

# Vue.js + Node.js

## 全栈开发实战

```
const http = require('http');
const server = http.createServer(function(req, res) {
    let data = '';
    req.on('data', function(chunk) {
        data += chunk;
    });
    req.on('end', function() {
        let method = req.method;
        let url = req.url;
        let headers = JSON.stringify(req.headers);
        let httpVersion = req.httpVersion;
        res.writeHead(200, {
            'content-type': 'text/html'
        });
    });
});
```

清华大学出版社

北京

let dataHtml = '<p>data:' + data + '</p>';
let methodHtml = '<p>method:' + method + '</p>';
let urlHtml = '<p>url:' + url + '</p>';
let headersHtml = '<p>headers:' + headers + '</p>';
let httpVersionHtml = '<p>httpVersion:' + httpVersion + '</p>';
let resData = dataHtml + methodHtml + urlHtml + headersHtml + httpVersionHtml;

## 内 容 简 介

本书着眼于实战开发，以 Node.js 和 Vue.js 原生知识和框架实战为主线，详细介绍 Node.js + Vue.js 开发的基础知识和相应案例实践。Node.js 后端包括 console、assert、fs、path、http、url、tcp、udp 等核心模块、与 MongoDB 和 MySQL 数据库的连接方法等。Vue.js 前端包括模板语法、生命周期、指令系统、样式绑定和路由等内容。同时，本书着重介绍基于 Node.js + Vue.js 开发的工具选择、环境搭建和项目构建等内容，所有案例都提供了详尽的源代码及其注释。

全书共 14 章，主要内容有 Node.js 和 Vue.js 环境搭建、Node.js 语法基础、Node.js 包管理机制、Node.js 网络开发、Node.js 文件模块使用、Node.js 数据库开发、Vue.js 数据、方法与生命周期、Vue.js 模板语法、Vue.js 指令系统、Vue.js 样式绑定和 Vue.js 路由，以及两个 Node.js+Vue.js 实战项目。

本书内容丰富、实例典型、实用性强，适合希望学习 Node.js+Vue.js 全栈开发的初学者，也适合作为高等院校和培训学校计算机及其相关专业师生的参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售

版权所有，侵权必究。举报：010-62782989，beiqinquan@tup.tsinghua.edu.cn。

### 图书在版编目（CIP）数据

Vue.js+Node.js 全栈开发实战 / 王金柱编著. —北京：清华大学出版社，2021.1

（Web 前端技术丛书）

ISBN 978-7-302-56719-6

I. ①V… II. ①王… III. ①网页制作工具—JAVA 语言—程序设计 IV. ①TP393.092.2②TP312.8

中国版本图书馆 CIP 数据核字（2020）第 210728 号

责任编辑：夏毓彦

封面设计：王 翔

责任校对：闫秀华

责任印制：宋 林

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者：三河市君旺印务有限公司

经 销：全国新华书店

开 本：190mm×260mm 印 张：18.5 字 数：474 千字

版 次：2021 年 1 月第 1 版 印 次：2021 年 1 月第 1 次印刷

定 价：69.00 元

---

产品编号：088868-01

# 前 言

Node.js 框架和 Vue.js 框架自发布伊始，就迅速掀起了一阵 Web 前端开发的热潮。随着最新的 Node.js 和 Vue.js 在功能上的日臻完善，其在 Web 开发领域已经牢牢占据了属于自己的一方天地。一方面，Node.js 使用 JavaScript 的语法使得服务器和客户端使用同一种语言进行开发成为可能；另一方面，Vue.js 通过“自底向上、增量开发”、渐进式的高效开发方式的加持，使得基于 Node.js + Vue.js 的前端应用开发具有独特的优势。Node.js 和 Vue.js 框架目前非常年轻、并且正处于高速发展时期，无数的开发者正准备或者已经进入这个领域，只有扎实的语言基础和丰富的实战开发经验才能在这个快速发展的领域立足。

目前图书市场上关于 Node.js + Vue.js 开发零基础入门的图书并不多，从语言基础开始并结合案例实践的书籍就更加少了。本书着眼于实战，通过介绍 Node.js 和 Vue.js 开发中最常用的原生模块和典型的项目案例，让读者全面、深入、透彻地理解 Node.js + Vue.js 开发的各种热门技术、各种主流框架及其整合使用，提高实际开发水平和项目实战能力。

## 本书特色

### 1. 内容全面、系统，结构合理

为了便于读者了解 Node.js 和 Vue.js 的开发，本书详细、系统地介绍入门级的原生模块技术，同时涵盖 Node.js 框架和 Vue.js 的实战案例。

### 2. 叙述完整，图文并茂

为了更好地帮助读者进行编程学习，书中附有大量的案例运行效果图，方便读者查看效果。

### 3. 结合实际，案例丰富

本书提供了大量的实际开发案例，便于读者在了解 Node.js 和 Vue.js 知识的同时进行案例实践，同时书中所有的案例都给出了完整的代码和详细的注释。

### 4. 涵盖基础和前沿知识

本书既介绍简单的网络开发、数据库开发等入门知识，也同时穿插基于 Node.js + Vue.js 框架开发的前沿知识，让读者在了解基础知识的同时紧跟 Web 前沿技术的步伐。

## 5. 提供大量的源代码

本书提供大量的源代码，所涉及的全部源代码都将开放给读者，以便于读者学习。读者也可以手动在 IDE 中输入源代码，通过实践提高动手能力。

## 本书内容

本书内容可分为 4 个部分。

第 1、2 章讲解 Node.js 和 Vue.js 的主要特点、发展历史和开发环境的搭建，主要包括基于 Node.js 和 Vue.js 框架的开发工具选择、开发环境搭建、以及构建项目框架的过程。

第 3~7 章讲解 Node.js 常用原生模块的开发基础，主要包括 Node.js 的包管理、模块机制以及 Node.js 开发中最常用的文件模块、网络开发模块、数据库开发模块等知识。

第 8~12 章讲解 Vue.js 在实际开发中的运用，主要包括 Vue.js 的数据、方法与生命周期、模板语法、指令系统、样式绑定和路由等知识。

第 13、14 章讲解两个基于 Vue.js + Node.js 框架设计的项目开发过程，主要包括学生成绩管理系统和全国城市信息查询系统的设计与实现。

## 源码下载

本书配套的源代码，请用微信扫描右侧二维码获取。

如果下载有问题，请联系 booksaga@163.com，邮件主题为“Vue.js+Node.js 全栈开发实战”。



## 本书读者

- Node.js+Vue.js 全栈开发人员
- Web 前端开发人员
- 需要一本案头必备查询手册的 Web 开发人员
- 高等院校和培训学校的师生

编者

2020 年 8 月

# 目 录

|  |    |
|--|----|
| 第 1 章 Node.js 基础与环境搭建 .....                    | 1  |
| 1.1 Node.js 基础.....                            | 1  |
| 1.1.1 Node.js 简介.....                          | 1  |
| 1.1.2 Node.js 发展历史.....                        | 2  |
| 1.1.3 Node.js 组织架构.....                        | 3  |
| 1.1.4 Node.js 特点.....                          | 4  |
| 1.1.5 Node.js 应用场景 .....                       | 6  |
| 1.1.6 Node.js 在国内的发展.....                      | 7  |
| 1.2 搭建 Node.js 开发环境.....                       | 8  |
| 1.2.1 Windows 10 系统下安装部署 Node.js 开发环境.....     | 8  |
| 1.2.2 测试 Node 开发环境.....                        | 13 |
| 1.2.3 通过 Node 运行 JavaScript 文件 .....           | 14 |
| 1.3 通过 Visual Studio Code 开发 Node 应用 .....     | 15 |
| 1.3.1 通过 Visual Studio Code 开发管理代码 .....       | 15 |
| 1.3.2 通过 Webpack 构建 Node 应用程序架构.....           | 21 |
| 1.3.3 通过 Visual Studio Code 开发调试 Node 应用 ..... | 23 |
| 第 2 章 Vue.js 基础介绍与环境搭建 .....                   | 40 |
| 2.1 Vue.js 基础.....                             | 40 |
| 2.1.1 Vue.js 简介.....                           | 40 |
| 2.1.2 Vue.js 发展历史 .....                        | 41 |
| 2.1.3 Vue.js 与 MVVM 架构模型 .....                 | 41 |
| 2.1.4 双向数据绑定 .....                             | 42 |
| 2.1.5 Vue.js 特点 .....                          | 42 |
| 2.2 Vue.js 快速开发环境.....                         | 43 |

|              |   |           |
|--------------|---|-----------|
| 2.2.1        | 直接通过<script>引入本地 Vue.js.....                          | 43        |
| 2.2.2        | 通过 CDN 方式引入 Vue.js .....                              | 44        |
| 2.2.3        | 兼容 ES Module 的方式 .....                                | 45        |
| 2.3          | Vue.js 脚手架开发环境.....                                   | 45        |
| 2.3.1        | 安装 Vue.js 脚手架 .....                                   | 45        |
| 2.3.2        | 通过 Vue.js 脚手架进行快速原型开发测试 .....                         | 47        |
| 2.3.3        | 通过 Vue.js 脚手架进行打包 .....                               | 48        |
| 2.3.4        | 通过 Vue.js 脚手架创建应用 .....                               | 50        |
| 2.3.5        | 通过 vue-cli 结合 Webpack 创建应用 .....                      | 53        |
| 2.3.6        | 通过 Visual Studio Code 开发调试 Vue 代码 .....               | 57        |
| <b>第 3 章</b> | <b>Node.js 语法基础 .....</b>                             | <b>62</b> |
| 3.1          | JavaScript 语法 .....                                   | 62        |
| 3.1.1        | 变量 .....  | 62        |
| 3.1.2        | 注释 .....  | 65        |
| 3.1.3        | 数据类型 .....  | 65        |
| 3.1.4        | 函数 .....  | 66        |
| 3.1.5        | 闭包 .....  | 68        |
| 3.2          | 命名规范与编程规范 .....                                       | 69        |
| 3.2.1        | 命名规范 .....  | 69        |
| 3.2.2        | 编程规范 .....  | 71        |
| 3.3          | Node.js 的控制台 console.....                             | 72        |
| 3.3.1        | console 对象下的各种函数 .....                                | 72        |
| 3.3.2        | console.log()函数 .....                                 | 73        |
| 3.3.3        | console.info()、console.warn()和 console.error()函数..... | 73        |
| 3.3.4        | console.dir()函数 .....                                 | 74        |
| 3.3.5        | console.time()和 console.timeEnd()函数 .....             | 74        |
| 3.3.6        | console.trace()函数.....                                | 75        |
| <b>第 4 章</b> | <b>Node.js 中的包管理.....</b>                             | <b>76</b> |
| 4.1          | NPM 介绍 .....  | 76        |
| 4.1.1        | NPM 常用命令 .....  | 76        |
| 4.1.2        | package.json 文件 .....                                 | 79        |
| 4.2          | 模块加载原理与加载方式 .....                                     | 81        |
| 4.2.1        | require 导入模块 .....                                    | 81        |
| 4.2.2        | exports 导出模块 .....                                    | 82        |

|                                      |            |
|--------------------------------------|------------|
| 4.3 Node.js 核心模块 .....               | 83         |
| 4.3.1 http 模块——创建 HTTP 服务器、客户端 ..... | 83         |
| 4.3.2 url 模块——url 地址处理 .....         | 87         |
| 4.3.3 querystring 模块——查询字符串处理 .....  | 88         |
| 4.4 Node.js 常用模块 .....               | 89         |
| 4.4.1 util 模块——实用工具 .....            | 89         |
| 4.4.2 path 模块——路径处理 .....            | 90         |
| 4.4.3 dns 模块 .....                   | 91         |
| <b>第 5 章 文件系统 .....</b>              | <b>93</b>  |
| 5.1 Node.js 文件系统介绍 .....             | 93         |
| 5.1.1 同步和异步 .....                    | 93         |
| 5.1.2 fs 模块中的类和文件的基本信息 .....         | 95         |
| 5.1.3 文件路径 .....                     | 96         |
| 5.2 基本文件操作 .....                     | 97         |
| 5.2.1 打开文件 .....                     | 97         |
| 5.2.2 关闭文件 .....                     | 98         |
| 5.2.3 读取文件 .....                     | 99         |
| 5.2.4 写入文件 .....                     | 100        |
| 5.3 其他文件操作 .....                     | 102        |
| <b>第 6 章 Node.js 网络开发 .....</b>      | <b>104</b> |
| 6.1 构建 TCP 服务器 .....                 | 104        |
| 6.1.1 使用 Node.js 创建 TCP 服务器 .....    | 104        |
| 6.1.2 监听客户端的连接 .....                 | 105        |
| 6.1.3 查看服务器监听的地址 .....               | 107        |
| 6.1.4 连接服务器的客户端数量 .....              | 107        |
| 6.1.5 获得客户端发送的数据 .....               | 108        |
| 6.1.6 发送数据给客户端 .....                 | 109        |
| 6.2 构建 TCP 客户端 .....                 | 111        |
| 6.2.1 使用 Node.js 创建 TCP 客户端 .....    | 111        |
| 6.2.2 连接 TCP 服务器 .....               | 112        |
| 6.2.3 获得从 TCP 服务器发送的数据 .....         | 112        |
| 6.2.4 向 TCP 服务器发送数据 .....            | 113        |
| 6.3 构建 HTTP 服务器 .....                | 114        |
| 6.3.1 创建 HTTP 服务器 .....              | 114        |

|   |            |
|---|------------|
| 6.3.2 HTTP 服务器的路由控制 .....                     | 115        |
| 6.4 利用 UDP 协议传输数据与发送消息 .....                  | 117        |
| 6.4.1 创建 UDP 服务器 .....                        | 117        |
| 6.4.2 创建 UDP 客户端 .....                        | 120        |
| <b>第 7 章 Node.js 数据库开发 .....</b>              | <b>123</b> |
| 7.1 使用 mongoose 连接 MongoDB.....               | 123        |
| 7.1.1 MongoDB 介绍 .....                        | 123        |
| 7.1.2 使用 mongoose 连接 MongoDB.....             | 125        |
| 7.1.3 使用 mongoose 操作 MongoDB.....             | 126        |
| 7.2 直接连接 MongoDB .....                        | 131        |
| 7.2.1 使用 node-mongodb-native 连接 MongoDB ..... | 131        |
| 7.2.2 使用 node-mongodb-native 操作 MongoDB ..... | 132        |
| 7.3 连接 MySQL .....                            | 139        |
| 7.3.1 MySQL 介绍 .....                          | 139        |
| 7.3.2 Node.js 连接 MySQL.....                   | 142        |
| 7.3.3 Node.js 操作 MySQL.....                   | 143        |
| <b>第 8 章 Vue.js 数据、方法与生命周期 .....</b>          | <b>146</b> |
| 8.1 Vue.js 数据 .....                           | 146        |
| 8.1.1 Vue.js 数据同步 .....                       | 146        |
| 8.1.2 Vue.js 数据冻结 .....                       | 150        |
| 8.1.3 Vue.js 实例 property 属性 .....             | 153        |
| 8.2 Vue.js 方法 .....                           | 156        |
| 8.2.1 观察属性方法 .....                            | 156        |
| 8.2.2 事件触发方法 .....                            | 163        |
| 8.2.3 自定义事件方法 .....                           | 165        |
| 8.3 Vue.js 生命周期 .....                         | 169        |
| 8.3.1 Vue.js 生命周期图示 .....                     | 169        |
| 8.3.2 Vue.js 生命周期钩子 .....                     | 171        |
| <b>第 9 章 Vue.js 模板语法 .....</b>                | <b>180</b> |
| 9.1 Vue.js 模板语法介绍 .....                       | 180        |
| 9.2 Vue.js 插值 .....                           | 180        |
| 9.2.1 文本插值 .....                              | 181        |
| 9.2.2 原始 HTML 插值 .....                        | 182        |

|                                    |     |
|------------------------------------|-----|
| 9.2.3 使用 JavaScript 表达式.....       | 184 |
| 9.3 Vue.js 指令.....                 | 185 |
| 9.3.1 Vue 指令概述 .....               | 186 |
| 9.3.2 v-if 条件表达式指令 .....           | 186 |
| 9.3.3 v-show 显示指令 .....            | 190 |
| 9.3.4 使用<template>元素渲染分组.....      | 192 |
| 9.3.5 v-for 循环指令 .....             | 195 |
| 9.4 Vue.js 指令参数.....               | 199 |
| 9.4.1 Vue.js 指令接收参数.....           | 199 |
| 9.4.2 Vue.js 指令接收动态参数.....         | 201 |
| 9.4.3 通过 Vue.js 指令动态参数改变元素类型 ..... | 203 |
| 9.5 Vue.js 指令修饰符.....              | 205 |
| 9.5.1 Vue.js 指令 prevent 修饰符 .....  | 205 |
| 9.5.2 Vue.js 指令 stop 修饰符 .....     | 210 |
| 9.5.3 Vue.js 指令 once 修饰符 .....     | 213 |
| 9.6 Vue.js 指令缩写.....               | 215 |
| 9.7 Vue.js 数据双向绑定.....             | 219 |
| 9.7.1 v-model 指令原理 .....           | 219 |
| 9.7.2 .lazy 修饰符 .....              | 224 |
| 9.7.3 .number 修饰符 .....            | 227 |
| 9.7.4 .trim 修饰符 .....              | 231 |
| 9.8 Vue.js 计算属性.....               | 232 |
| 第 10 章 Vue.js 样式绑定 .....           | 236 |
| 10.1 Vue.js 绑定 HTML Class .....    | 236 |
| 10.1.1 绑定静态 Class .....            | 236 |
| 10.1.2 绑定动态 Class .....            | 238 |
| 10.1.3 绑定多个 Class .....            | 242 |
| 10.2 通过数组语法绑定 Class.....           | 244 |
| 10.3 Vue.js 绑定 HTML Style .....    | 246 |
| 10.3.1 绑定静态 Style .....            | 246 |
| 10.3.2 绑定 Style 对象 .....           | 247 |
| 10.3.3 绑定多重值的 Style .....          | 249 |
| 10.4 通过计算属性绑定样式 .....              | 249 |

|        |   |     |
|--------|---|-----|
| 第 11 章 | Vue.js 组件基础 .....                       | 252 |
| 11.1   | Vue.js 全局组件 .....                       | 252 |
| 11.2   | Vue.js 局部组件 .....                       | 254 |
| 11.3   | 通过 Prop 向子组件传递数据 .....                  | 256 |
| 第 12 章 | Vue.js 路由 .....                         | 261 |
| 12.1   | 安装 vue-router 库的方法 .....                | 261 |
| 12.2   | 基于 vue-router 库开发单页面应用 .....            | 262 |
| 12.3   | 基于 vue-router 库实现动态路由 .....             | 264 |
| 第 13 章 | 项目实战：基于 Vue.js+Node.js 实现学生成绩管理系统 ..... | 266 |
| 13.1   | 学生成绩管理系统组织架构设计 .....                    | 266 |
| 13.2   | 构建项目应用框架 .....                          | 267 |
| 13.3   | 后台数据结构 .....                            | 269 |
| 13.4   | 功能模块组件设计 .....                          | 269 |
| 13.5   | 功能模块路由设计 .....                          | 275 |
| 13.6   | 测试应用 .....                              | 276 |
| 第 14 章 | 项目实战：基于 Vue.js+Node.js 实现城市信息查询系统 ..... | 279 |
| 14.1   | 全国城市信息查询系统组织架构设计 .....                  | 279 |
| 14.2   | 构建项目应用框架 .....                          | 280 |
| 14.3   | 后台数据获取方式 .....                          | 280 |
| 14.4   | 功能模块组件设计 .....                          | 281 |
| 14.5   | 功能模块路由设计 .....                          | 284 |
| 14.6   | 测试应用 .....                              | 286 |

# 第 1 章

## Node.js 基础与环境搭建

Node.js 是 JavaScript 的运行时环境，是一个基于 Google Chrome V8 引擎设计实现的、跨平台兼容的、可以运行在服务器端的脚本开发语言。如今，随着全栈开发技术的不断深入与日益盛行，Node.js 逐渐成为前端设计开发的通用标准框架。例如，大多数读者耳熟能详的 Angular、React 和 Vue.js 这三大渐进式前端开发框架，均与 Node.js 有着密不可分的关联关系。本书的重点，就是介绍关于 Node.js 与 Vue.js 前端全栈开发的内容。

本章主要对 Node.js 进行整体介绍，并对其发展历史和相关版本进行详细说明，同时也介绍后续开发中所涉及的基础知识。

通过本章的学习可以：

- 了解 Node.js 的发展历史和特点。
- 了解 V8 引擎的介绍及其与 Node.js 的关系。
- 掌握 Node.js 的一些应用场景。

### 1.1 Node.js 基础

本节包括 Node.js 的基础简介、发展历史、组织架构，以及具体应用等方面的内容。

#### 1.1.1 Node.js 简介

Node.js 是基于 Google Chrome 浏览器内置的 V8 引擎所开发的 JavaScript 运行时环境。它充分利用了 V8 引擎的强大性能，借鉴了其中的很多前沿技术（例如：GC 机制、事件驱动、非阻塞的 I/O 模型，等等），保证了 Node.js 的轻量与高效，进而受到了众多开发者的追捧。

Node.js 最显著的特点就是能够运行在服务器端（区别于其他脚本语言），以及良好的多平台兼容性（支持 Windows、Linux、Mac OS X、SunOS 和 FreeBSD 等多种系统平台），使其成为最重要的脚本程序设计语言。

我们都知道，一般的 JavaScript 脚本语言需要在浏览器环境下才可以解释执行。而 Node.js 是服务器端的脚本语言，可以直接在后端解释执行。下面就是最基本的 Node.js 命令执行方法。

```
#node filename.js // TODO: node 命令直接解释执行 filename.js 脚本文件
```

由于 Chrome V8 引擎执行 JavaScript 脚本的速度非常快，因此 Node.js 所开发出来的应用程序性能非常好。Node.js 已经成为全栈开发的首选语言之一，并且从它衍生出众多出色的全栈开发框架。Node.js 已经在全球被众多公司使用，包括创业公司 Voxer、Uber 以及沃尔玛、微软这样的知名公司。它们每天通过 Node.js 处理的请求数以亿计，可以说在要求苛刻的服务器系统，Node.js 也可以轻松胜任。

Node.js 还包括一个完善的社区。在 Node.js 的官方网站 <http://nodejs.org/> 可以找到大量的文档和示例程序，并且 Node.js 还有一个强大的 NPM 包管理器。渐渐地，越来越多的人参与到本项目中来，可用的第三方模块和扩展增长迅猛，而且质量也在不断提升，Node.js 已是全球较大的开源库生态系统之一。



Node.js 并不是一个 JavaScript 应用，而是一个 JavaScript 的运行时环境，其底层由 C++ 语言编写而成。

### 1.1.2 Node.js 发展历史

任何语言或框架都不是一天形成的，而是经过漫长的测试、发布、再测试、再发布的迭代过程，本节重点介绍一下 Node.js 的发展历史。

Node.js 的创始人就是大名鼎鼎的 Ryan Dahl，其实 Ryan Dahl 本人是数学专业的。在 2008 年年末，一个偶然的机会让他了解到 Google 推出了一个全新的 Chrome 浏览器及其 V8 引擎。而当他了解到，Chrome V8 是一个为了实现更快的 Web 体验而专门制作的 JavaScript 引擎时，非常希望找到一种语言能提供先进的推送功能并集成到自己的网站中去，从而避免采用传统的不断轮询拉取数据的方式。

Ryan Dahl 对 C/C++ 和系统调用非常熟悉，他使用系统调用（用 C）实现消息推送这样的功能。如果只使用非阻塞式 Socket，每个连接的开销都会非常小。在小规模测试中，它能同时处理几千个闲置连接，并可以实现相当大的吞吐量。但是，他并不想使用 C，他希望能采用另外一种漂亮灵活的动态语言。他最初也希望采用 Ruby 来写 Node.js，但是发现 Ruby 虚拟机的性能不能满足要求，后来便尝试采用 V8 引擎，所以选择了 C++ 语言。

2009 年 2 月，Ryan Dahl 首次在自己的博客上宣布准备基于 V8 创建一个轻量级的 Web 服务器并提供一套库，2009 年 5 月他正式在 GitHub 上发布最初版本的部分 Node.js 包。随后几个月里，有人开始使用 Node.js 开发应用。实践证明，JavaScript 与非阻塞 Socket 配合得相当完美，只需要简单的几行 JavaScript 代码就可以构建出非常复杂的非阻塞服务器。时间到了 2010 年年底，Node.js 获得云计算服务商 Joyent 的资助。创始人 Ryan Dahl 加入 Joyent，全职负责 Node.js 的发展。Node.js 从此以后迅猛发展，并成为一种流行的开发语言。

在官方网站上 Node.js 的版本号是从 0.1.14 开始的，每个发布版本对应不同的 V8 引擎版

本和 NPM 包管理器版本，截至作者写作时最新的 LTS 版本为 V12.16.3。当然，这期间 Node.js 发生了很曲折的故事，感兴趣的读者可以自行去了解一下。

总结一下，Node.js 的发展大致可以分为如下 4 个阶段。

### 1. 发展初期

创始人 Ryan Dahl 带着他的团队开发出以 Web 为中心的 Web.js，此时的一切都非常混乱，API 也大多都处于试验阶段。

### 2. 快速发展时期

Node.js 的核心用户 Isaac Z. Schlueter 开发出奠定了 Node.js 如今地位的重要工具——NPM 包管理工具。同时，这也是 Schlueter 未来成为 Ryan 接班人的重要条件。之后 Connect、Express、Socket.io 等库的出现吸引了一大批爱好者加入到 Node.js 开发者的阵营中来。CoffeeScript 的出现更是让不少 Ruby 和 Python 开发者找到了学习的理由。其间一大波以 Node.js 作为运行环境的 CLI 工具涌现，其中不乏用于加速前端开发的优秀工具，如 Babel、Less、Sass、UglifyJS、Browserify、Grunt、Gulp 等。在这个阶段，Node.js 的发展势如破竹。

### 3. 不稳定时期

经过了一大批一线工程师的探索实践后，Node.js 开始进入时代的更迭期，新模式代替旧模式，新技术代替旧技术，新实践代替旧实践。ECMAScript 6（ECMAScript 2015）也开始出现在 Node.js 世界中。ECMAScript 6 的发展越来越明显，V8 也对 ECMAScript 6 中的部分特性实现了支持，如 Generator 等。

### 4. 稳步发展时期

随着 ECMAScript 6 的发展和最终定稿，出现了一大批利用 ECMAScript 6 特性开发的新模块，如原 Express 核心团队所开发的 Koa。Node.js 之父 Ryan Dahl 退出 Node.js 的核心开发，转而做其他的研究项目。Ryan Dahl 的接任者 Isaac Schlueter 负责将 Node.js 一直开发下去并进行不断完善。

## 1.1.3 Node.js 组织架构

前面介绍了 Node.js 是一个完整的 JavaScript 开发环境，是基于 Google 的 Chrome V8 引擎对代码进行解释执行。它在设计之初就已经定位用来解决传统 Web 开发语言所遇到的诸多问题，所以 Node.js 有很多其他开发语言所不具备的优点。下面介绍 Node.js 的组织架构。

从图 1.1 中可以看到，只有最顶层的 Node 标准库部分是用 JavaScript 语言编写的，其余的底层部分均是用 C/C++ 语言编写的。

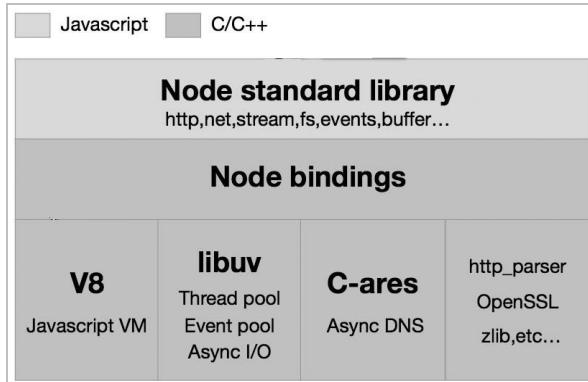


图 1.1 Node.js 组织架构图

我们继续分析图 1.1 中描述的 Node.js 组织架构，关于 Node.js 的结构大致可分为以下三个层次。

### 1. Node.js 标准库 (Node standard library)

这一层由 JavaScript 编写，是在使用过程中能直接调用的 API，在源码中的 lib 目录下可以看到，具体包括 http、net、stream、fs、buffer、events 等模块。

### 2. Node bindings

这一层是 JavaScript 与底层 C/C++能够沟通的关键，前者通过 bindings 调用后者，相互交换数据。

### 3. Node 基础构件

这一层包括支撑 Node.js 运行的基础构件，是使用 C/C++语言编写的，其中包括以下主要模块：

- **V8**: Google 推出的 JavaScript VM ( JavaScript 虚拟机 )，也是 Node.js 为什么用 JavaScript 的关键，它为 JavaScript 提供了在非浏览器端运行的环境，它的高效是 Node.js 之所以高效的原因之一。
- **libuv**: 为 Node.js 提供了跨平台、线程池、事件池、异步 I/O 等能力，是 Node.js 如此强大的关键。
- **C-ares**: 提供了异步处理 DNS 相关的能力。
- **http\_parser**、**OpenSSL**、**zlib** 等：提供了包括 HTTP 解析、OpenSSL、数据压缩等功能。

## 1.1.4 Node.js 特点

Node.js 的强大体现在很多方面，如事件驱动、异步处理、非阻塞 I/O 等。这里将介绍 Node.js 具备不同于其他框架的特点，包括事件驱动、异步编程等内容。

## 1. 事件驱动

在某些传统的网络编程语言中，都会使用到回调函数。比如：当 Socket 资源达到某种状态时，注册的回调函数就会执行。在 Node.js 的设计思想中，是以事件驱动为核心的，它提供的绝大多数 API 都是基于事件的、异步的风格。以 Net 模块为例，其中的 net.Socket 对象就有以下事件：connect、data、end、timeout、drain、error、close 等。使用 Node.js 的开发人员需要根据自己的业务逻辑注册相应的回调函数。这些回调函数都是异步执行的。这意味着虽然在代码结构中这些函数看起来是依次注册的，但是它们并不依赖于自身出现的顺序，而是等待相应的事件触发。

事件驱动的优势在于充分利用了系统资源，执行代码无须阻塞等待某种操作完成，有限的资源可以用于其他的任务。此类设计非常适合于后端的网络服务编程，Node.js 的目标也在于此。在服务器开发中，并发的请求处理是一个大问题，阻塞式的函数会导致资源浪费和时间延迟。通过事件注册、异步函数，开发人员可以提高资源的利用率，性能也会改善。

## 2. 异步、非阻塞 I/O

从 Node.js 提供的支持模块中，我们可以看到包括文件操作在内的许多函数都是异步执行的。这和传统语言存在区别。为了方便服务器开发，Node.js 的网络模块特别多，包括 HTTP、DNS、NET、UDP、HTTPS、TLS 等。开发人员可以在此基础上，快速构建 Web 服务器应用。一个异步 I/O 的大致流程如图 1.2 所示。

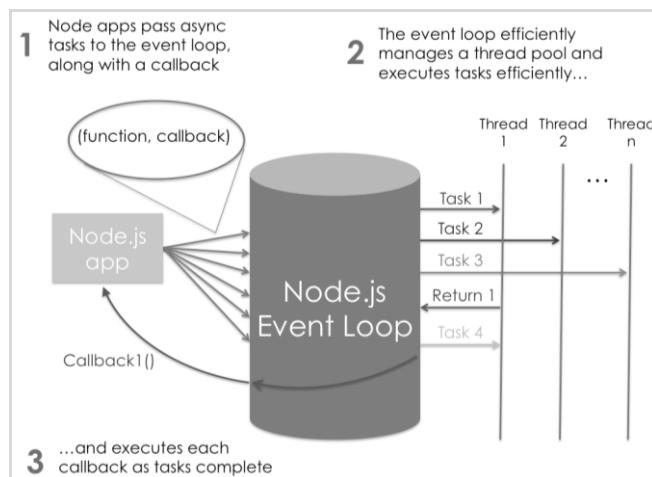


图 1.2 异步 I/O 的流程

如图 1.2 中所示，异步 I/O 流程主要包括以下过程：

### (1) 发起 I/O 调用

- ① 用户通过 JavaScript 代码调用 Node 核心模块，将参数和回调函数传入核心模块。
- ② Node 核心模块会将传入的参数和回调函数封装成一个请求对象。
- ③ 将这个请求对象推入 I/O 线程池等待执行。

④ JavaScript 发起的异步调用结束，JavaScript 线程继续执行后续操作。

## (2) 执行回调

① I/O 操作完成后会将结果存储到请求对象的 result 属性上，并发出操作完成的通知。

② 每次事件循环时会检查是否有完成的 I/O 操作，如果有就将请求对象加入 I/O 观察者队列中，之后当作事件处理。

③ 处理 I/O 观察者事件时会取出之前封装在请求对象中的回调函数，执行这个回调函数，并将 result 当作参数，以完成 JavaScript 回调的目的。

Node.js 的网络编程非常方便，提供的模块（在这里是 HTTP）开放了容易上手的 API 接口，短短几行代码就可以构建服务器。

## 3. 性能出众

创始人 Ryan Dahl 在设计的时候就考虑了性能方面的问题，选择 C++ 和 V8，而不是 Ruby 或者其他的虚拟机。Node.js 在设计上以单进程、单线程模式运行。事件驱动机制是 Node.js 通过内部单线程高效率地维护事件循环队列来实现的，没有多线程的资源占用和上下文切换。这意味着面对大规模的 HTTP 请求，Node.js 是凭借事件驱动来完成的。从大量的测试结果分析来看，Node.js 的处理性能非常出色，在 QPS (Query Per Second, 每秒查询率) 达到 16700 次时，内存仅占用 30MB (测试环境：RHEL 5.2、CPU 2.2GHz、内存 4GB)。

## 4. 单线程

Node.js 和大名鼎鼎的 Nginx 一样，都是以单线程为基础的。这正是 Node.js 保持轻量级和高性能的关键，也是 Ryan Dahl 设计 Node.js 的初衷。这里的单线程是指主线程为“单线程”，所有阻塞的部分交给一个线程池处理，然后这个主线程通过一个队列跟线程池协作。我们编写的 js 代码部分不用再关心线程问题，代码也主要由一堆 callback 回调构成，然后主线程在循环过程中适时调用这些代码。

单线程除了保证 Node.js 高性能之外，还保证了绝对的线程安全，使开发者不用担心同一变量同时被多个线程读写而造成的程序崩溃。

### 1.1.5 Node.js 应用场景

Node.js 可以应用到很多方面，可以说，从 Node.js 开始程序员们就能用 JavaScript 来开发服务器端的程序了。Node.js 为前端开发程序员们提供了便利，并在各大网站中承担重要角色，成为开发高并发大型网络应用的关键技术。Web 站点早已不仅限于内容的呈现，很多交互性和协作型环境也逐渐被搬到了网站上，而且这种需求还在不断地增长。这就是所谓的数据密集型实时 (Data-Intensive Real-Time) 应用程序，比如在线协作的白板、多人在线游戏等，这种 Web 应用程序需要一个能够实时响应大量并发用户请求的平台支撑它们，这正是 Node.js 擅长的领域。此外，Node.js 的跨平台特性也是使用 Node.js 语言开发流行的另一大原因。

Node.js 的主要应用场景如下：

- JSON APIs：构建一个 Rest/JSON API 服务，Node.js 可以充分发挥其非阻塞 IO 模型以及 JavaScript 对 JSON 的功能支持（如 `JSON.stringify` 函数）。
- 单页面、多 Ajax 请求应用：如 Gmail，前端有大量的异步请求，需要服务后端有极高的响应速度。
- 基于 Node.js 开发 UNIX 命令行工具：Node.js 可以大量生产子进程，并以流的方式输出，这使得它非常适合做 UNIX 命令行工具。
- 流式数据：传统的 Web 应用，通常会将 HTTP 请求和响应看成原子事件，而 Node.js 会充分利用流式数据这个特点，构建非常酷的应用，如实时文件上传系统 Transloadit。
- 准实时应用系统：如聊天系统、微博系统，但 JavaScript 是有垃圾回收机制的，这就意味着系统的响应时间是不平滑的（GC 垃圾回收会导致系统这一时刻停止工作）。如果想要构建硬实时应用系统，Erlang 是一个不错的选择。

例如，实时互动交互比较多的社交网站，像 Twitter 这样的公司，它必须接收 tweets 并将其写入数据库。实际上，每秒几乎有数千条 tweet 达到，数据库不可能及时处理高峰时段所需的写入数量。Node 成为这个问题解决方案的重要一环。Node 能处理数万条入站 tweet。它能快速而又轻松地将它们写入一个内存排队机制（例如 memcached），另一个单独进程可以从那里将它们写入数据库。Node 能处理每个连接而不会阻塞通道，从而能够捕获尽可能多的 tweets。

虽然看起来 Node.js 可以做很多事情，并且拥有很高的性能，但是 Node.js 并不是万能的，有一些类型的应用，Node.js 可能处理起来会比较吃力。例如，CPU 密集型的应用、模板渲染、压缩/解压缩、加/解密等操作都是 Node.js 的软肋。

### 1.1.6 Node.js 在国内的发展

Node.js 在初期发展的时候，国内就有大量的开发者开始持续关注了。随着 Node.js 的不断成熟，很多国内的公司都开始采用这一新技术。Node.js 开发者在国内的数量不断增加，并涌现出很多组织和机构来自发地进行推广和技术分享。

国内的各大视频培训网站上都有 Node.js 开发的培训教程，各大门户网站也都或多或少地采用了 Node.js 的开发技术，淘宝、网易、百度等有很多项目运行在 Node.js 之上。阿里云是在这方面比较靠前的公司，它们的云平台率先支持 Node.js 的开发。淘宝也为 Node.js 搭建了国内的 NPM 镜像网站，方便国内的开发者下载各种开发包。

以下是关于 Node.js 中文资源的汇总清单：

(1) Node.js 官方网站，是 Node.js 在国内的官方网站，有关于 Node.js 最新版本的下载和新闻，丰富的文档资料非常值得认真学习，是 Node.js 开发爱好者不容错过的网站，网址为 <https://nodejs.org/zh-cn/>。

(2) CNode 社区，由一批热爱 Node.js 技术的工程师发起，已经吸引了互联网各个公司的专业技术人员加入，是目前国内非常具有影响力的 Node.js 开源技术社区，致力于 Node.js 的技术研究，并有论坛会定期组织一些技术交流活动，网址为 <https://cnnodejs.org>。

(3) Node.js 中文网, 是一个专业的 Node.js 中文知识分享社区, 致力于普及 Node.js 知识, 分享 Node.js 研究成果, 努力推进 Node.js 在中国的应用和发展, 网站有大量的技术博客和文章, 各个级别的开发者都能找到适合自己学习的资料, 网址为 [www.nodejs.cn](http://www.nodejs.cn)。

(4) 淘宝 NPM 镜像, 是一个完整 [npmjs.org](https://npm.taobao.org/) 镜像, 可以用此代替官方版本, 同步频率为每 10 分钟一次, 以保证尽量与官方服务同步, 网址为 <https://npm.taobao.org/>。

(5) Node.js & HTML5 论坛, 也是一个学习 Node.js 和前端开发技术非常好的网站, 每天都有大量原创文章发布, 并且技术问题可以很快被回答。当然, 如果愿意为其他人解答技术问题或者进行技术分享也是非常受欢迎的。其网址为 <http://forjs.org/>。

每年的 JavaScript 中国开发者大会和各种 Node.js 分享沙龙也都是很好的学习 Node.js 开发技术和交流的机会。一个开发者要时刻保持谦虚的心态, 并不断学习新的技术。这对开发者来说是一种基本能力和素养。

## 1.2 搭建 Node.js 开发环境

对于学习任何一门编程语言, 第一步都是要先搭建好该语言的开发环境。Node.js 可以在多个不同的平台稳定运行, 并且均具有良好的兼容性。本节主要介绍如何在 Windows 系统平台下搭建 Node.js 的开发环境。至于其他操作系统平台下的操作方法大同小异, 读者可自行学习。

### 1.2.1 Windows 10 系统下安装部署 Node.js 开发环境

Node.js 可以在多个版本的 Windows 系统平台 (Windows 7、Windows 10 以及 Windows Server 系列等) 下稳定运行, 本书主要介绍在 Windows 10 系统下 Node.js 开发环境的安装部署过程。

在 Windows 10 系统中进行 Node.js 环境部署相对比较简单。首先, 从 Node.js 的官方网站 (<https://nodejs.org/en/download/>) 上下载新的 Node 安装包。如果下载网速较慢, 国内用户还可以通过 Node.js 官方中文站点 (<http://nodejs.cn/download/>) 进行下载。中文站点的下载页面与英文版的下载页面略有不同, 中文版页面只提供新的发布版本, 而英文站点同时提供新的长期支持 (LTS) 版本和新的发布版本的下载链接。

Node.js 的安装包在 Windows 平台分为 installer 和 binary 两个版本。installer 是通常的安装包发布版本 (.msi), binary 为二进制版本, 可以下载后直接运行 (.exe)。这里建议使用后缀为.msi 的安装版本。此外, Node.js 的安装包分为 32 位和 64 位, 在下载的时候请查看系统的具体信息, 并选择正确的安装包进行下载和安装。



Node.js 的其他发布版本可以从 <https://nodejs.org/dist/> 链接找到，本文以 v12.6.3(LTS) 64-bit 版本在 Windows 10 系统下进行安装为例。

打开英文网站的下载页面 (<https://nodejs.org/en/download/>)，如图 1.3 所示。

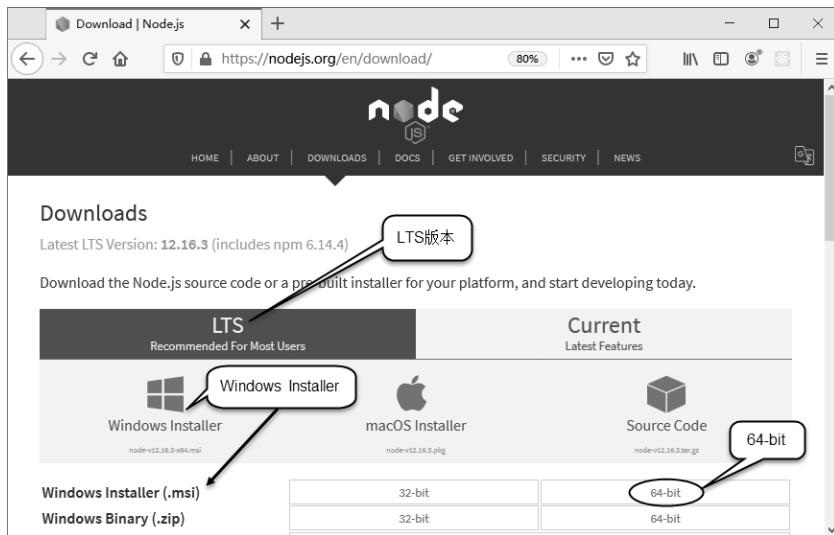


图 1.3 Node.js 官方网站下载页面

如图 1.3 中的标识所示，首先选择 LTS（长期支持版）版本，然后选择 Windows Installer 项，并找到 Windows Installer(.msi)子项，最后选择 64-bit 版本进行下载。这里，具体下载得到的安装包名称为“node-v12.16.3-x64.msi”，然后就可以进行安装了。

Node 安装包的具体安装步骤如下：

(1) Node 安装包是一个大约 20MB 大小的 msi 格式的 Windows 系统安装文件。双击该安装包开始安装，会弹出如图 1.4 所示的欢迎界面。



图 1.4 Node.js 安装步骤 (1)

(2) 如图 1.4 中的箭头所示, 通过单击 Next(下一步)按钮进入如图 1.5 所示的“End-User License Agreement(终端用户协议许可)”界面。

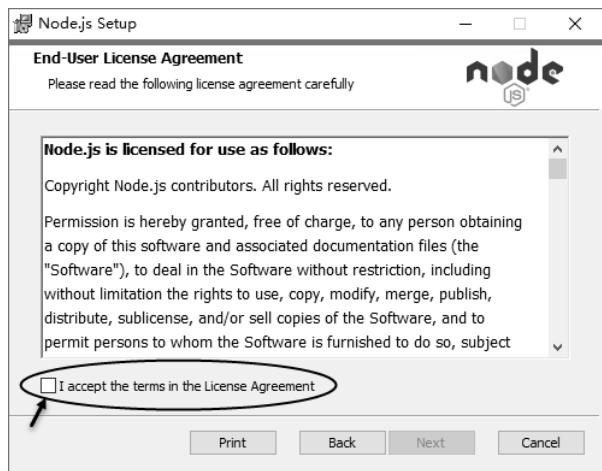


图 1.5 Node.js 安装步骤 (2)

(3) 如图 1.5 中的箭头所示, 勾选接受协议许可选项后, Next 按钮会变为如图 1.6 所示的可用状态。

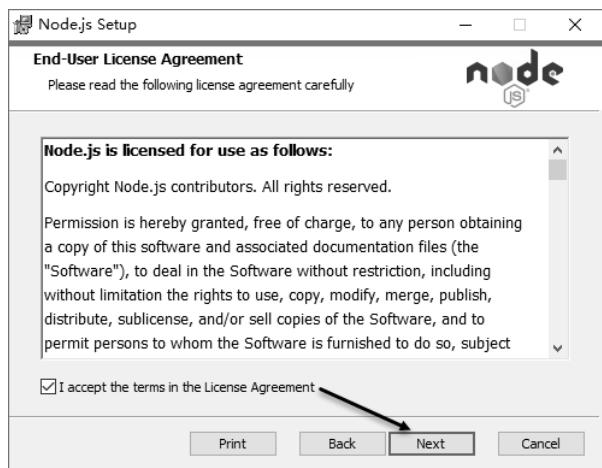


图 1.6 Node.js 安装步骤 (3)

(4) 如图 1.6 中的箭头所示, 单击 Next 按钮进入下一步, 此时会打开选择目标安装目录界面, 如图 1.7 所示。



图 1.7 Node.js 安装步骤（4）

如图 1.7 中的箭头所示，默认的安装目录为“C:\Program Files\nodejs\”，可以通过单击 Change...按钮选择自己的目标安装目录，这里建议安装在磁盘（可选任一磁盘，笔者选择的是 D 盘）的根目录下。

（5）单击图 1.7 中的 Next 按钮，进入如图 1.8 所示的“Custom Setup（自定义安装选项）”界面。

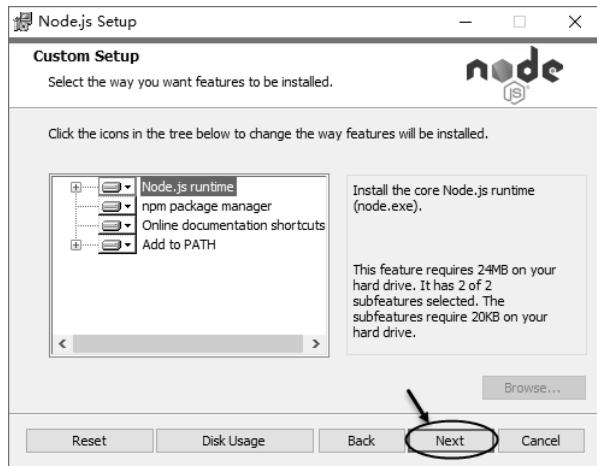


图 1.8 Node.js 安装步骤（5）

（6）如图 1.8 所示，默认会安装全部的“自定义安装选项”。这里，建议读者选择安装全部选项，尤其是“Add to PATH”选项是用来设置系统默认的环境变量 PATH 的。另外，在安装 Node.js 的同时默认也安装了 NPM（Npm Package Manager），NPM 是 Node.js 著名的包管理工具。

（7）单击 Next 按钮，进入“Ready to install Node.js（准备安装 Node.js）”界面，如图 1.9 所示。

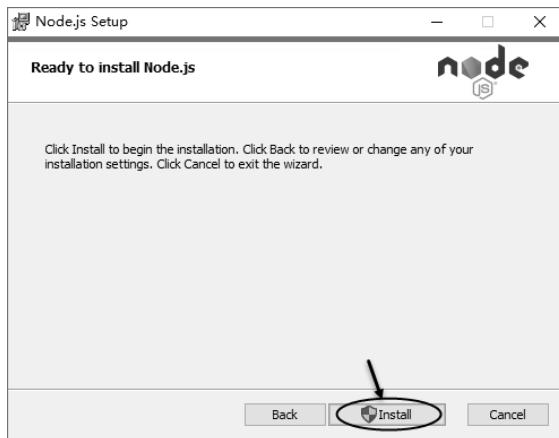


图 1.9 Node.js 安装步骤（6）

(8) 如图 1.9 中的箭头所示，单击 Install 按钮就会开始执行安装，如图 1.10 所示。

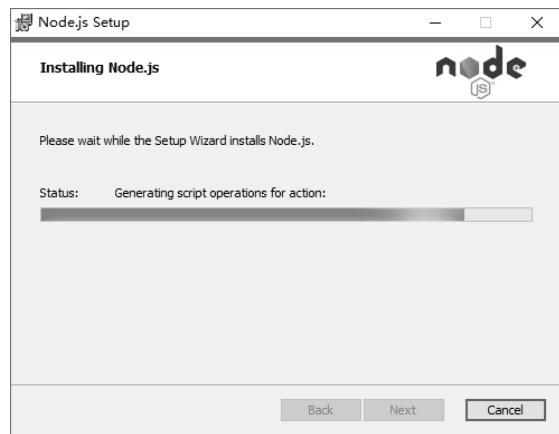


图 1.10 Node.js 安装步骤（7）

(9) 安装完毕后，会显示如图 1.11 所示的界面，单击 Finish 按钮完成 Node.js 的安装步骤。

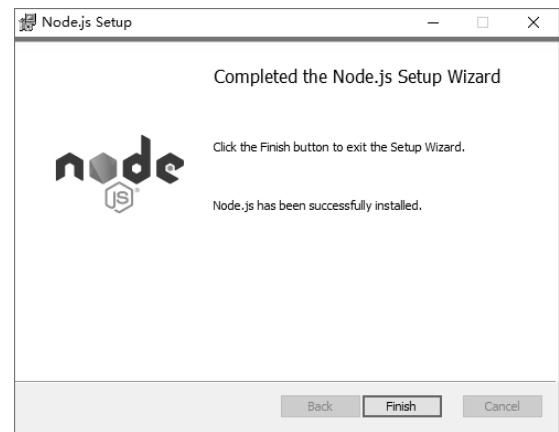


图 1.11 Node.js 安装步骤（8）

**提 示**

要查看 PATH 变量，需要右击“计算机”，选择“属性”|“高级系统属性”选项，打开“系统属性”对话框。然后单击“高级”|“环境变量”选项，在“用户变量”列表项中找到 PATH 变量，单击下方的“编辑”按钮就可以看到所有变量在这里的设置，主要是设置路径。

## 1.2.2 测试 Node 开发环境

在 Node.js 开发包安装完毕后，首先要测试 Node 的开发环境，以验证 Node 是否安装成功。测试方法很简单，在控制台命令行窗口通过输入以下 node 命令即可。

```
node --version      // 对于完整的命令参数，可查询 node 安装版本号
node -v           // 对于简化的命令参数，可查询 node 安装版本号
```

这里选择 Node.js 自带的控制台命令行工具（Node.js command prompt）进行测试，效果如图 1.12 所示。

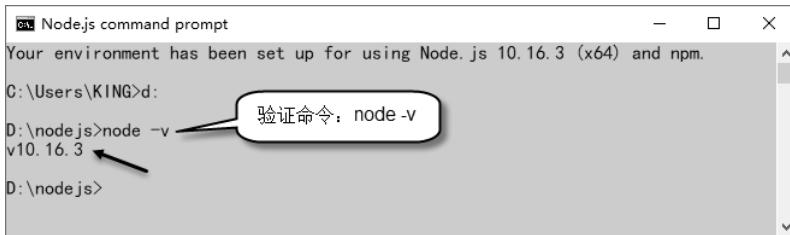


图 1.12 验证 Node.js 环境是否安装成功

如图 1.12 中的箭头和标识所示，通过输入“node -v”命令查询到了当前系统安装了 v10.16.3 版本，表明 Node 开发环境已经安装成功了。

前文中介绍了 node 自带的包管理工具 NPM，下面再验证一下 NPM 工具是否也安装成功了，方法是输入以下 node 命令。

```
npm --version      // 对于完整的命令参数，可查询 npm 安装版本号
npm -v           // 对于简化的命令参数，可查询 npm 安装版本号
```

效果如图 1.13 所示。

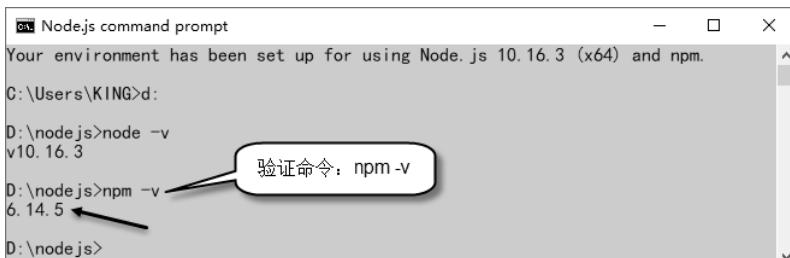


图 1.13 验证 NPM 包管理工具是否安装成功

如图 1.13 中的箭头和标识所示，通过输入“npm -v”命令查询到当前系统安装了 v6.14.5

版本，表明 NPM 包管理工具同步也安装成功了。

那么，Node 开发环境能做什么呢？最简单的一项功能就是可以直接在命令行运行 JavaScript 脚本程序，具体请参看图 1.14 所示的操作过程。



图 1.14 直接在命令行运行 js 脚本程序（一）

如图 1.14 所示，首先要在命令行通过输入 node 命令进入 Node 开发环境，然后就可以输入 js 脚本代码了。由于 Node 命令行开发环境是交互式的 js 解释器，因此在输入 js 代码并回车后就可以直接把运行结果打印出来。

其实，这种在输入 js 代码回车后直接输出结果的方式不是十分友好，如果想实现一些稍微复杂 js 代码就很困难。好在 js 代码是通过分号（;）断句的，可以将若干句 js 代码写在一行中来完成。具体请参看图 1.15 所示的操作过程。

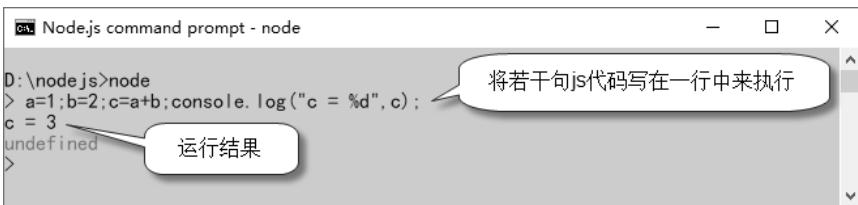


图 1.15 直接在命令行运行 js 脚本程序（二）

如图 1.15 所示，将若干句 js 代码写在一行中，就可以实现一个简单的求和运算。

不过，这种直接在 Node 命令行中编写 js 脚本代码的方式仅限于非常简单的场景。如果想完成复杂的代码功能，就需要通过 Node 环境来运行 js 脚本文件。

### 1.2.3 通过 Node 运行 JavaScript 文件

在 Node 命令行开发环境中，可以直接运行 js 脚本文件。方法也很简单，通过 node 命令指定 js 文件名即可，具体如下：

```
node filename.js // filename.js 指定具体 js 脚本文件名
```

接下来，我们将前一小节中测试的两段 js 脚本代码整合到同一个 js 脚本文件中，代码如下：

**【代码 1-1】**（详见源代码 cmdline 目录中的 cmdline.js 文件）

```
01 console.log("Hello Node.js!");
02 var a = 1;
03 var b = 2;
```

```
04 var c = a + b;
05 console.log("c = %d", c);
```

### 【代码说明】

在上面的代码中，将前一小节中测试的两端 js 脚本代码写在了一个 js 脚本文件中，然后通过 node 命令行工具运行该 js 脚本文件，如图 1.16 所示。



图 1.16 Node 命令行运行 js 脚本文件

如图 1.16 中的标识所示，通过 node 命令行运行 js 脚本文件（commandline.js），得到了同样的运行结果。

在实际的 Node.js 项目开发中，无论是使用轻量级代码开发工具，还是使用集成式的开发平台工具，且不论项目的 js 源码文件有多复杂（数量众多且关系嵌套），在后台均是通过上面的方式来运行 js 脚本文件。

## 1.3 通过 Visual Studio Code 开发 Node 应用

本节介绍如何开发一个完整的 Node 应用程序。这里，需要使用到 Visual Studio Code 作为 Node 代码开发和管理的工具，同时还需要使用到 Webpack 模块打包器作为构建 Node 应用框架的工具，这些都是开发 Web 前端应用的基础知识。

### 1.3.1 通过 Visual Studio Code 开发管理代码

Visual Studio Code（简称 VS Code）可谓是近年来风头日盛的代码开发和管理工具，该工具由微软（Microsoft）公司负责开发维护，并在 2015 年 4 月 30 日的 Build 开发者大会上第一次正式对外发布。

Visual Studio Code 的设计目标是一款能够满足跨平台运行的、轻量级的、可扩展的开发工具，主要用于编写当前流行的 Web 应用和云服务应用的源代码编辑器。虽然 VS Code 同样是隶属于 Visual Studio 系列开发平台，但与诸如 Visual Studio 2015、Visual Studio 2019 这类的重量级的、功能强大的、集成式的开发平台完全不同。VS Code 自身仅是一款源代码编辑器，编写好的 C 程序无法在 VS Code 中编译运行，js 脚本文件也是无法在 VS Code 中解释执行。

不过，如果因此而小看 VS Code 就大错特错了，VS Code 的特点就是提供很强的扩展功能，强大到让设计人员惊叹的程度。VS Code 的扩展功能是通过安装相应的插件实现的，前面

提到的 C 程序和 js 脚本，甚至包括 Java、Python、Node 程序，以及本书后面将要重点介绍的 Vue.js 程序，都可以通过安装相对应的功能插件而得到支持。因此，VS Code 在极短的时间内就得到了业内的充分认可，并迅速在源代码编辑器市场占据了一席之地。对此，相信你也只能感叹微软（Microsoft）研发能力的强大了。

在 Windows 平台下安装 Visual Studio Code 的方法非常简单，自家的操作系统平台自然会提供最好的支持。首先，进入 Visual Studio Code 官方网址的下载页面 (<https://code.visualstudio.com/Download>)，如图 1.17 所示。

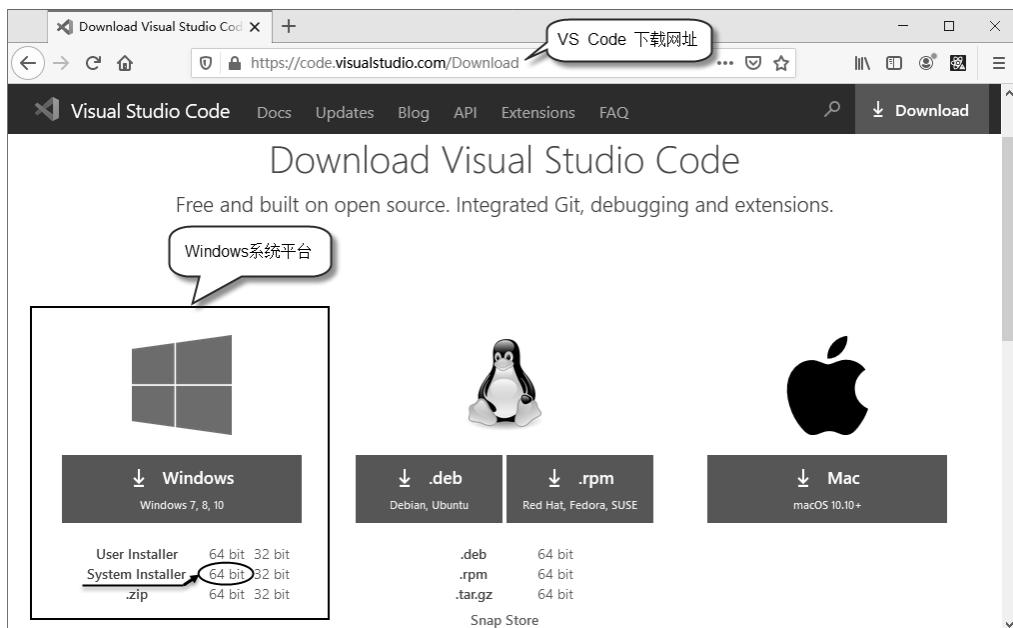


图 1.17 Visual Studio Code 官方下载页面

如图 1.17 所示，在 Windows 系统平台区域中的“System Installer”类别中选择 64-bit 版本的安装包（VSCodeSetup-x64-1.45.1.exe）进行下载，该版本是为系统用户配置的。另外，还有一种“User Installer”类别是为个人用户配置的版本。在图中，还可以看到 Visual Studio Code 的 Linux 版本和 Mac 版本，这表明了 Visual Studio Code 是一款跨平台的开发工具。

在 Windows 操作系统中安装 Visual Studio Code 安装包很简单，直接双击 VS Code 安装包开始安装，然后按照步骤进行安装即可。

(1) 首先会进入安装程序的“许可协议”界面，如图 1.18 所示。

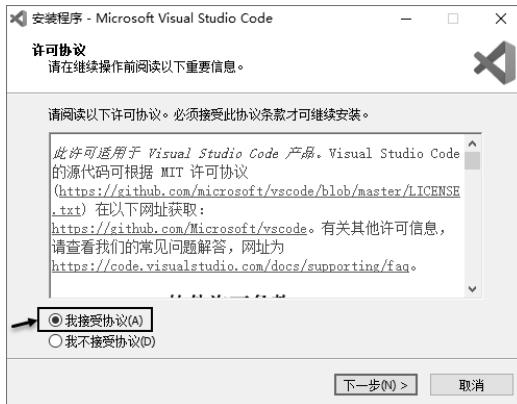


图 1.18 Visual Studio Code 安装过程（一）

（2）选择“我接受协议”选项才会激活“下一步”按钮，单击“下一步”按钮进入“选择目标位置”界面，如图 1.19 所示。



图 1.19 Visual Studio Code 安装过程（二）

（3）这里可以选择用户自定义的 VS Code 安装路径，继续单击“下一步”按钮。然后，就会进入实际安装过程，直到安装程序执行完毕，如图 1.20 所示。



图 1.20 Visual Studio Code 安装过程（三）

(4) 如图 1.20 所示, 如果勾选了“启动 Visual Studio Code”选项, 在单击“完成”按钮后会退出安装程序, 同时启动 Visual Studio Code 开发工具。初次运行 Visual Studio Code 时的界面如图 1.21 所示。



图 1.21 初次运行 Visual Studio Code 程序时的界面

(5) 如图 1.21 所示, Visual Studio Code 开发工具的界面非常简洁(轻量级), 窗口的顶部是一个主菜单, 窗口的左侧一列是快捷工具按钮列表(可以安装扩展功能), 窗口的中部是具体的文档编辑区域(包括一些开发功能的快捷方式)。

(6) 继续单击左侧快捷工具按钮列表中最上面的“资源管理器(快捷键为  $\text{Ctrl}+\text{Shift}+\text{E}$ )”按钮, 界面效果如图 1.22 所示。



图 1.22 Visual Studio Code 资源管理器

如图 1.22 中的箭头和标识所示，打开“资源管理器”后，会在窗口左侧显示一个区域，里面包括了工程目录、大纲和时间线等项目。设计人员可以在这里新建或引入自己的工程目录进行源代码的有效管理，本书的工程目录名称为“vueprojects”（注意，VS Code 默认会将工程目录名称全部显示为大写）。

接下来，我们尝试通过 Visual Studio Code 测试运行前面【代码 1-1】所创建的 commandline.js 文件。首先，在 VS Code 中通过“文件”菜单中的“打开文件夹...”菜单项来打开工程目录，效果如图 1.23 所示。



图 1.23 Visual Studio Code 打开工程目录

如图 1.23 中的箭头所示，单击后会打开一个“文件选择对话框”，选中之前创建好的“vueprojects”工程目录即可，效果如图 1.24 所示。



图 1.24 Visual Studio Code 打开工程目录

如图 1.24 中的箭头所示，已经能看到之前【代码 1-1】所创建的 commandline.js 文件。那么，如何通过 VS Code 工具运行该 js 脚本文件呢？

这里，需要在 VS Code 中安装一个名称为“Code Runner”的插件以运行 js 脚本文件，具体安装方法如下。在左侧快捷工具按钮列表中找到“扩展（快捷键为 Ctrl+Shift+X）”按钮，界面效果如图 1.25 所示。

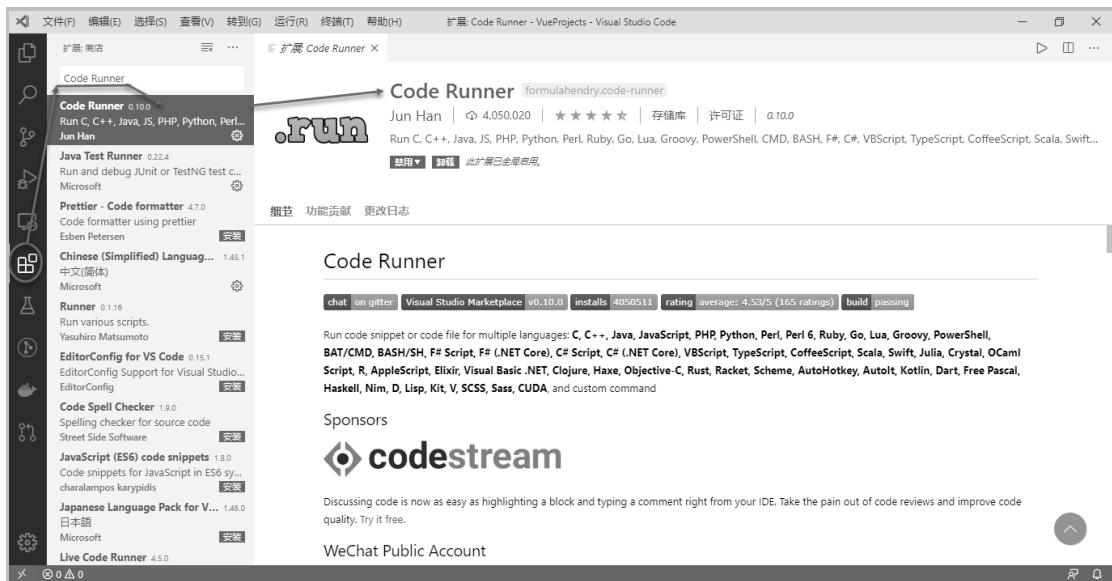


图 1.25 Visual Studio Code 安装扩展

如图 1.25 中的箭头和标识所示，打开“扩展”界面后在搜索栏输入“Code Runner”字符串，下面自动筛选出来的扩展列表中的第一项就是“Code Runner”插件，直接单击进行“安装”即可。由于笔者已经安装过该插件，因此图 1.25 中显示的是已安装状态，从右侧主窗口可以看到关于“Code Runner”插件的功能介绍。可以看到该插件除了支持 JavaScript 脚本语言外，还支持多种编程语言（C、C++、Java、PHP、Python、Go...），功能是十分的强大。

安装好“Code Runner”插件后，就可以直接在 VS Code 中测试运行 js 脚本文件了。我们返回图 1.24 所示的界面状态，选中 commandline.js 文件并右击，会弹出一个上下文功能菜单，效果如图 1.26 所示。

如图 1.26 中的箭头所示，在上下文功能菜单中单击“Run Code”菜单项，VS Code 开发工具会弹出一个“输出”界面窗口，显示 js 代码调试运行的输出结果，效果如图 1.27 所示。



图 1.26 通过“Code Runner”运行 JavaScript 脚本文件（一）

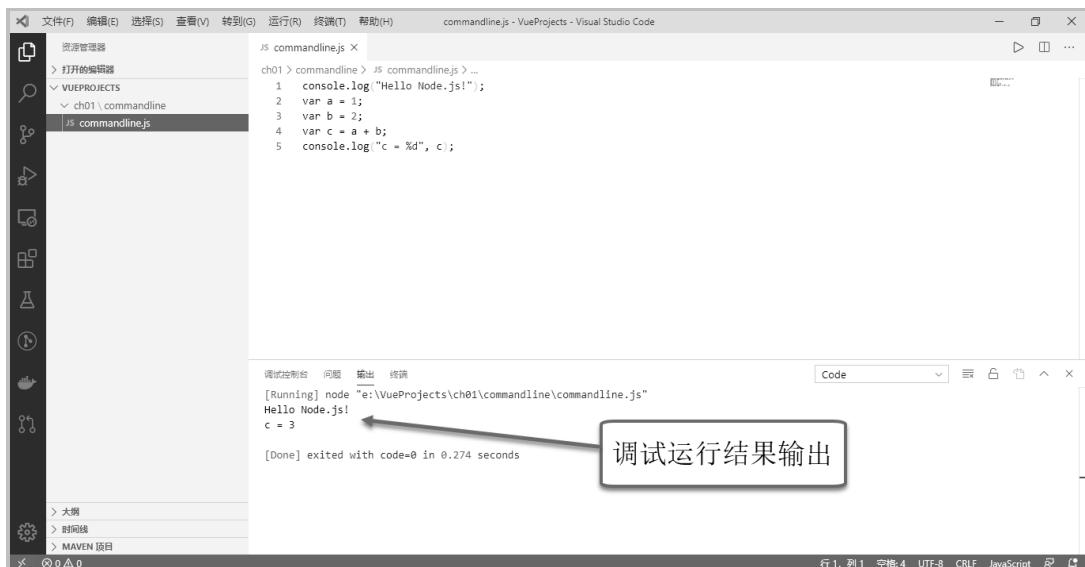


图 1.27 通过“Code Runner”运行 JavaScript 脚本文件（二）

如图 1.27 中的标识所示，调试运行的输出结果与前面图 1.16 所示的结果是一致的，说明 VS Code 开发工具的“Code Runner”插件可以完美替代 Node 程序在命令行终端的运行能力。

### 1.3.2 通过 Webpack 构建 Node 应用程序架构

在前文中，我们介绍了如何通过 Visual Studio Code 开发工具测试运行单个 js 脚本文件的 Node 程序。不过，这种开发方式在近年来已经被一种全新的、以自动化方式构建 Web 前端应用的方式所取代了。

所谓“自动化”方式，其实就是通过一种或几种自动化工具来构建 Web 前端应用的开发框架。这类开发框架基本都是通过一种或几种工具自动生成的，生成后的框架会包含大部分 Web 前端应用所需的基本类库、第三方插件和支撑配置文件，等等。可以说，这种全新的开发方式将 Web 前端应用开发提升到了一个全新的高度，并且符合将前端分离开来进行独立设计的大趋势。

Web 前端的自动化构建工具有很多种，其中著名的就是 Webpack 模块化打包器工具，其也是目前 Web 前端开发中流行的工具之一。Webpack 的功能十分强大，设计了“入口（entry）”“输出（output）”“加载器（loader）”和“插件（plugins）”这四个概念，以递归方式构建出一个应用程序主要资源的依赖关系图，并最终完成将 JavaScript 模块打包成一个或多个“包（bundle）”。

由于本书不是专门介绍 Webpack 工具的（感兴趣的读者可去参考 Webpack 官网的内容），这里仅就如何使用 Webpack 工具构建 Node 应用进行简单的介绍，过程如下。

（1）创建 Node 应用程序目录，并进入该目录，命令行如下：

```
mkdir vueprojects && cd vueprojects
```

（2）通过 npm 命令进行初始化项目的操作，命令行如下：

```
npm init -y
```

npm 就是包管理工具命令，参数 init 表示进行 Node 应用项目的初始化操作。该命令参数会生成一个 package.json 配置文件，用于描述该项目的详细配置信息。如果再添加“-y”参数，则表示项目使用默认的配置参数（省去了人工操作的过程），效果如图 1.28 所示。

```
E:\VueProjects>npm init -y
Write to E:\VueProjects\package.json:

{
  "name": "VueProjects",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

E:\VueProjects>
```

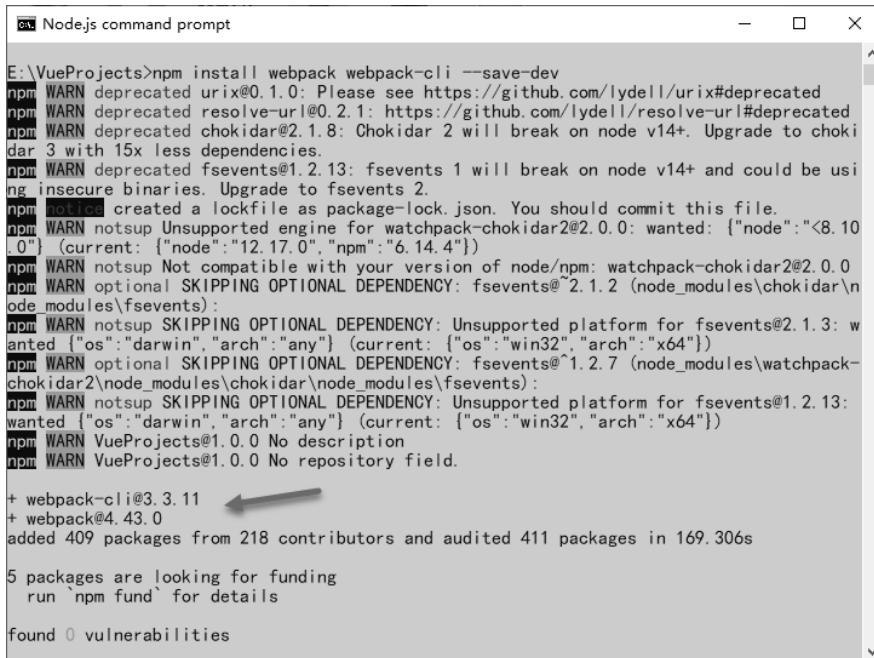
图 1.28 通过“npm init”命令初始化项目

如图 1.28 所示，命令行控制台中显示出生成的 package.json 文件的内容，这些信息都是默认生成的，后期用户可以自行修改。

（3）通过 npm 命令在本地安装 Webpack 开发包，命令如下：

```
npm install webpack webpack-cli --save-dev
```

在 npm 命令中使用参数 `install` 表示安装第三方开发包，后面指定了开发包的名称 `webpack` 和 `webpack-cli`。其中，`webpack-cli` 表示 `webpack` 的命令行工具，也就是在命令行中可以支持使用 Webpack。另外，使用参数“`--save-dev`”表示该安装包为调试开发时的依赖项，信息会记录到 `package.json` 文件中的 `devDependencies` 子项中，后期项目发布时不需要这些安装包。安装过程需要一些时间，请读者耐心等待，效果如图 1.29 所示。



```
cmd Node.js command prompt
E:\VueProjects>npm install webpack webpack-cli --save-dev
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
npm WARN deprecated chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependencies.
npm WARN deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade to fsevents 2.
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN notsup Unsupported engine for watchpack-chokidar2@2.0.0: wanted: {"node": "<8.10.0"} (current: {"node": "12.17.0", "npm": "6.14.4"})
npm WARN notsup Not compatible with your version of node/npm: watchpack-chokidar2@2.0.0
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@~2.1.2 (node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.1.3: wanted {"os": "darwin", "arch": "any"} (current: {"os": "win32", "arch": "x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@^1.2.7 (node_modules\watchpack-chokidar2\node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os": "darwin", "arch": "any"} (current: {"os": "win32", "arch": "x64"})
npm WARN VueProjects@1.0.0 No description
npm WARN VueProjects@1.0.0 No repository field.

+ webpack-cli@3.3.11
+ webpack@4.43.0
added 409 packages from 218 contributors and audited 411 packages in 169.306s

5 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

图 1.29 通过“npm”命令安装 Webpack

如图 1.29 中的箭头所示，已经成功安装了“`webpack@4.43.0`”和“`webpack-cli@3.3.11`”开发包，其中“`@`”符号后面标识的是开发包的版本号。

至此，通过 Webpack 工具构建 Node 应用程序框架的步骤就基本完成了。

### 1.3.3 通过 Visual Studio Code 开发调试 Node 应用

目前，有多种开发工具可以支持 Node.js 应用的开发，比如：jetBrains WebStorm、Eclipse、Visual Studio Code 等。这些开发工具原则上是“条条大路通罗马”，相互间各有优势、并无优劣之分。在本书中，我们选择 Visual Studio Code 开发工具，之所以选择该工具也是为了配合后面关于 Vue.js 的内容。

接下来，通过 Visual Studio Code 打开刚刚创建的 Node 应用目录（`vueprojects`），效果如图 1.30 所示。

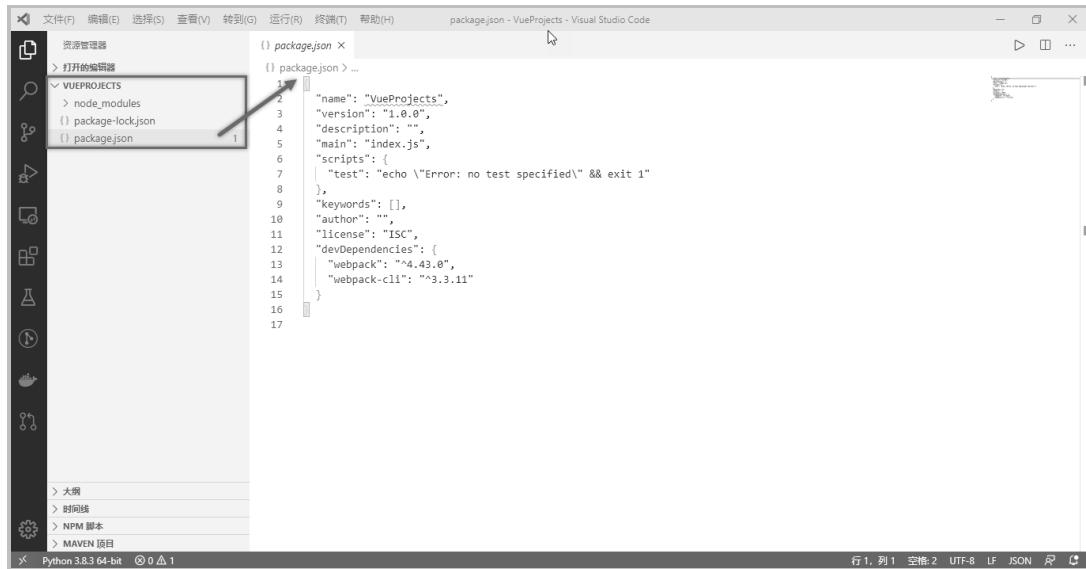


图 1.30 通过 VS Code 管理 Node 应用（一）

如图 1.30 中的方形区域标识所示，里面显示了 Node 应用的目录结构，具体内容如图 1.31 所示。

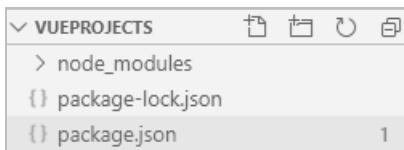


图 1.31 通过 VS Code 管理 Node 应用（二）

如图 1.31 所示，`node_modules` 目录存放了通过 `npm` 命令安装的各种开发包，里面不仅仅是刚刚安装的 `webpack` 和 `webpack_cli` 开发包，还包括了整个 Node 生态系统必要的依赖项。因此，`node_modules` 目录的体积通常是一个较大的数量级，这点也算是 Node 生态系统的不足之处。

再次返回图 1.30 中，箭头标识的是 `package.json` 配置文件的内容，具体代码如下：

### 【代码 1-2】

```

01  {
02      "name": "vueprojects",
03      "version": "1.0.0",
04      "description": "",
05      "main": "index.js",
06      "scripts": {
07          "test": "echo \"Error: no test specified\" && exit 1"
08      },
09      "keywords": [],
10      "author": "",
11      "license": "ISC",

```

```

12   "devDependencies": {
13     "webpack": "^4.43.0",
14     "webpack-cli": "^3.3.11"
15   }
16 }

```

### 【代码说明】

- 第 02 行代码中的“name”字段标识的是该项目的名称（vueprojects）。
- 第 03 行代码中的“version”字段标识的是该项目的版本号（1.0.0）。
- 第 05 行代码中的“main”字段标识的是该项目的主入口脚本文件。
- 第 06 行代码中的“scripts”字段用于执行自定义脚本命令。这里定义的“test”参数，就相当于执行“npm run test”脚本命令。
- 第 12~15 行代码中的“devDependencies”字段用于定义开发调试阶段安装的依赖项，其中第 13~14 行代码定义的依赖项印证了之前的安装命令（npm install webpack webpack-cli --save-dev）。

以上关于 package.json 配置文件的内容都是最基本的。功能越复杂，安装依赖项越多，package.json 配置文件的信息也会随之增加。

另外，还有一个 package-lock.json 配置文件，这个文件是在 NPM v5 版本以后新增加的功能。在通过“npm install”命令安装开发包后会自动生成该文件，该文件锁定了当前安装的小版本号。因此，当用户再次使用“npm install”命令后，就可以避免直接通过 package.json 配置文件将开发包升级到最新版本，有效地避免了因为版本升级带来的各种依赖冲突。但若用户真想升级到最新版本或某个指定版本，则必须在开发包名称后使用“@latest”或“@版本号”来执行指定版本号的升级。

接下来，具体介绍如何开发一个基本的 Node 应用程序，该程序实现了一个简单的、基于 Node 框架的 Web 服务器。

(1) 在应用程序根目录下创建一个 HTML 5 页面文件 (index.html)，我们通过 VS Code 开发工具来实现该操作。在应用程序根目录 (vueprojects) 上单击快捷方式“新建文件”图标，效果如图 1.32 所示。



图 1.32 通过 VS Code 新建 HTML 页面文件 (一)

单击“新建文件”图标后会自动创建一个文件。注意，这个文件没有名称，也没有文件类型后缀，全部需要设计人员手动输入 (index.html)，这点可能与其他开发平台差异比较大。因此，该新建文件是一个空白的 HTML 5 网页，如图 1.33 所示。

## Vue.js+Node.js 全栈开发实战

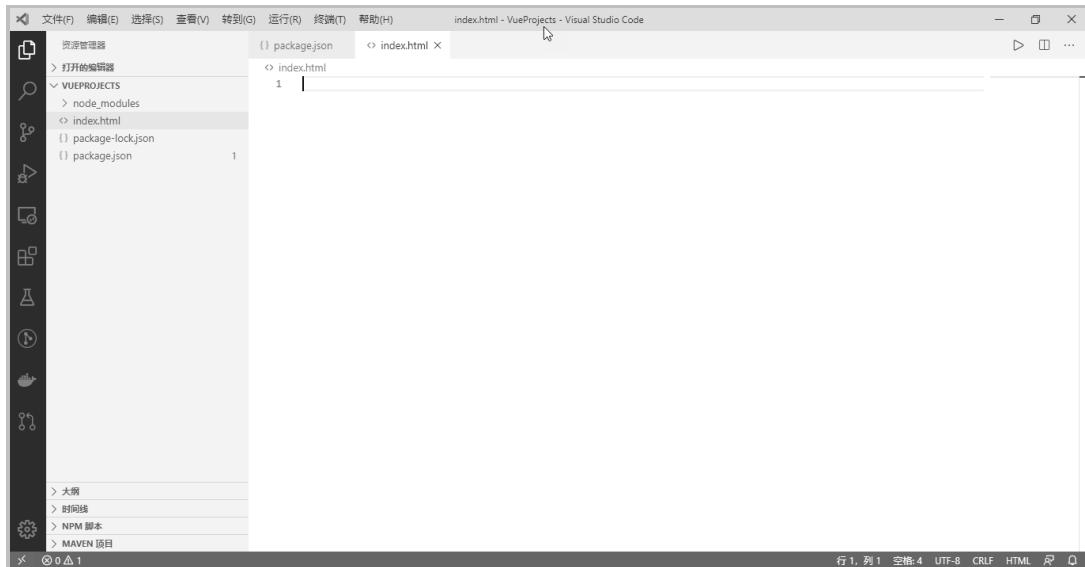


图 1.33 通过 VS Code 新建 HTML 页面文件（二）

新建的 index.html 网页全部空白，没有一行默认生成的代码。此时，读者如果对 VS Code 的开发工具产生质疑，那就大错特错了。VS Code 为设计人员提供很强大的自动代码生成功能，下面为读者演示如何操作。

在空白页面最开始处输入字符“!”，然后会激活并弹出一个快捷输入方式，如图 1.34 所示。

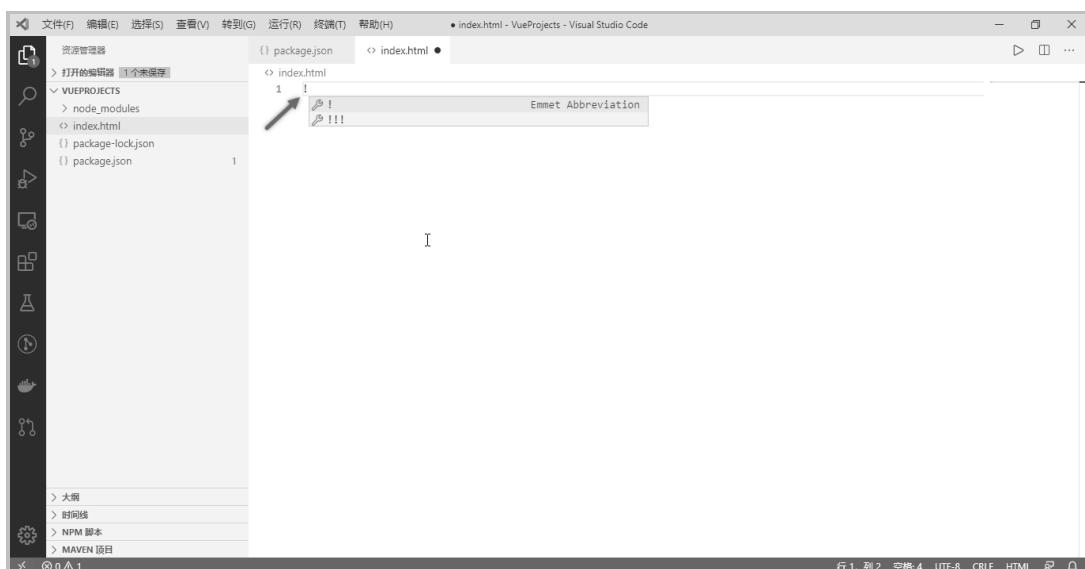


图 1.34 通过 VS Code 新建 HTML 页面文件（三）

如图 1.34 中的箭头所示，此时会弹出一个快捷输入方式，在里面的列表菜单中有一个“!”选项。注意，后面的提示信息为“Emmet Abbreviation”，说明该功能是由 Emmet 插件所提供

的。提到这个 Emmet 插件可谓是大名鼎鼎，在很多集成开发工具或轻量级代码编辑器中都有其身影，是一款非常好用的代码自动生成工具。因此，VS Code 索性就将 Emmet 插件内置其中了，用户不用再单独安装该插件了。

下面我们看一下 Emmet 插件的使用方法，在选中的“!”选项中直接按回车键或 Tab 键，代码自动生成的效果如图 1.35 所示。

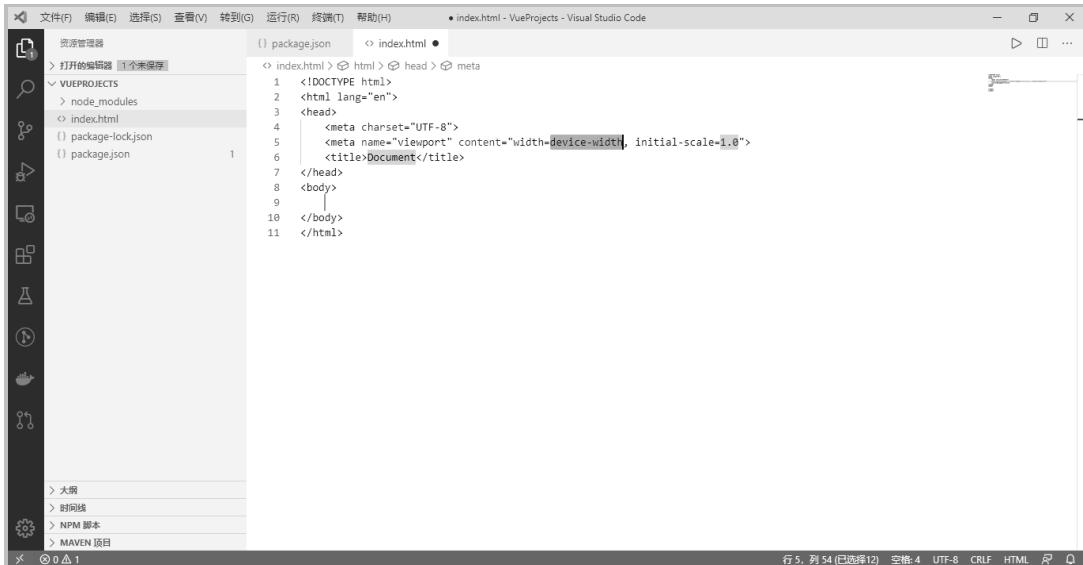


图 1.35 通过 VS Code 新建 HTML 页面文件（四）

如图 1.35 所示，Emmet 插件直接在 index.html 文件中自动生成了一个 HTML 5 网页模板。

(2) 在应用程序根目录下创建一个 src 子目录，在该子目录下新建一个 js 脚本文件(index.js)，方法同图 1.32 所示。在 index.js 脚本文件中，输入如下代码：

### 【代码 1-3】

```
01 document.write('Hello Node!');
```

### 【代码说明】

- 第 01 行代码通过调用 document 对象的 write() 方法向页面中写入一行信息。

(3) 对 index.html 页面代码进行适当修改，在 index.html 页面文件中引用 index.js 脚本文件，代码如下：

### 【代码 1-4】

```
01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04   <meta charset="UTF-8">
05   <meta name="viewport" content="width=device-width, initial-scale=1.0">
06   <title>Document</title>
```

```

07  </head>
08  <body>
09    <script src="./src/index.js"></script>
10  </body>
11  </html>

```

**【代码说明】**

- 第 09 行代码中，通过<script>标签引入了 index.js 脚本文件。

(4) 更新 package.json 配置文件的内容，代码如下：

**【代码 1-5】**

```

01  {
02    "name": "vueprojects",
03    "version": "1.0.0",
04    "description": "",
05    "private": true,
06    "scripts": {
07      "test": "echo \\"Error: no test specified\\" && exit 1"
08    },
09    "keywords": [],
10    "author": "",
11    "license": "ISC",
12    "devDependencies": {
13      "webpack": "^4.43.0",
14      "webpack-cli": "^3.3.11"
15    }
16  }

```

**【代码说明】**

- 第 05 行代码中，添加 private 参数，确保我们的安装包是私有的（private）；同时，移除 main 入口，防止意外发布应用代码。

(5) 此时就可以简单地测试一下该 Node 应用了，最直接的方法就是在浏览器中运行 index.html 页面文件，效果如图 1.36 所示。

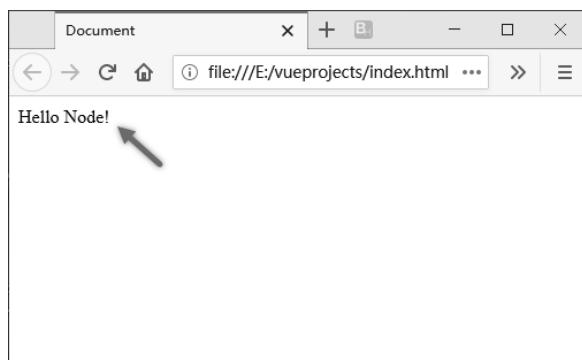


图 1.36 在浏览器中运行 HTML 页面文件

如图 1.36 中的箭头所示，浏览器页面中显示了 index.js 脚本代码执行的结果。不过，虽然上面的 Node 应用正确执行了，但远没有发挥出 Webpack 工具的能力。接下来，我们通过 Webpack 重构一下上面的 Node 应用。

#### (6) 创建“分发 (dist)”目录结构。

这里需要调整一下原始的目录结构，将“源 (src)”代码从“分发 (dist)”代码中分离出来。根据 Webpack 定义的规范，“源 (src)”代码是开发时使用的代码，“分发 (src)”代码是构建过程产生的代码最小化和优化后的“输出 (output)”目录，用于在浏览器中展示给终端用户。

具体操作还是在 VS Code 工具下完成，效果如图 1.37 所示。

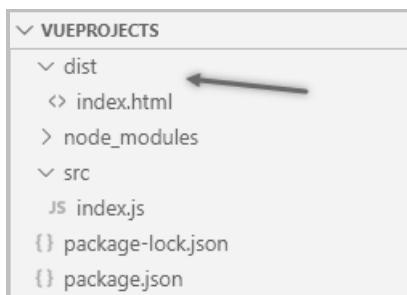


图 1.37 创建“分发 (dist)”目录结构

如图 1.37 中的箭头所示，我们还需要将 index.html 页面文件移入“分发 (dist)”目录，便于后期调试运行。

#### (7) 修改完善 index.js 脚本文件和重新更新 index.html 页面文件。

我们将直接在页面文件写入 (`document.write()`方法) 文本的方式，替换为通过层 (`div`) 标签插入文本的方式。另外，这里需要借助一个 Lodash 插件来完成该任务，Lodash 是一个高性能的、模块化的 JavaScript 实用工具库，简化了数组、字符串和对象等类型的操作难度，在业内十分受欢迎。Lodash 插件的使用方法有以下几种：

##### ① 直接引用方式：

```
<script src="lodash.js"></script>
```

##### ② CDN 方式：

```
<script src="https://unpkg.com/lodash@4.17.5"></script>
```

##### ③ 本地安装方式：

```
npm install --save lodash // --save 表示为生产环境
```

以上 3 种方式均可行，但从 Node 应用的角度看，推荐使用第 3 种方式。关于“本地安装方式”的过程，如图 1.38 所示。

```

E:\vueprojects>npm install --save lodash
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.1.3 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.1.3: wanted
  {"os": "darwin", "arch": "any"} (current: {"os": "win32", "arch": "x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\watchpack-chokidar2\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted
  {"os": "darwin", "arch": "any"} (current: {"os": "win32", "arch": "x64"})

+ lodash@4.17.15
added 1 package from 2 contributors and audited 412 packages in 12.028s

5 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

E:\vueprojects>

```

图 1.38 安装 Lodash 库

如图 1.38 中的箭头所示，默认安装的是 Lodash 库的最新稳定版（4.17.15）。另外，在上面的第 2 种 CDN 方式中，指定的安装版本号一般都是比较新的稳定版。

接下来，修改一下 index.js 脚本文件，具体修改后的代码如下：

### 【代码 1-6】

```

01 import _ from 'lodash';
02 /**
03  * func - create div component
04 */
05 function divComp() {
06     var eleDiv = document.createElement('div');
07     // TODO: use Lodash '_' to join string.
08     eleDiv.innerHTML = _.join(['Hello', 'Webpack', '!'], ' ');
09     // TODO: return element div
10     return eleDiv;
11 }
12 // TODO: append div to body
13 document.body.appendChild(divComp());

```

### 【代码说明】

- 第 01 行代码中，通过 import 方式导入已安装的 Lodash 模块，并将模块名称命名为“\_”（Lodash 通用命名方式）。
- 第 05~11 行代码定义了一个 divComp()方法，用于创建一个层（div）标签元素，然后在该标签元素内插入文本信息。具体内容如下：
  - 第 06 行代码中，通过 document 对象调用 createElement()方法创建了一个层（div）标签元素（eleDiv）。
  - 第 08 行代码中，通过“\_”调用 join()方法连接了一组字符串，并定义为层（eleDiv）标签元素的 innerHTML 属性。
  - 第 10 行代码中，返回创建好的层（eleDiv）标签元素。
- 第 13 行代码中，通过 document 对象的 appendChild()方法，将刚刚创建的层（eleDiv）元素追加到 body 元素中，实现了在页面中输入文本的效果。

在修改好 index.js 脚本文件后，还要修改一下 index.html 页面文件。因为现在需要通过 Webpack 工具打包成脚本文件，所以需要加载分发（dist）目录中的 bundle（包）脚本文件（main.js），就不再需要加载 src 目录的原始脚本文件了。index.html 页面文件修改后的代码如下：

### 【代码 1-7】

```
01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04     <meta charset="UTF-8">
05     <meta name="viewport" content="width=device-width, initial-scale=1.0">
06     <title>Document</title>
07 </head>
08 <body>
09     <!--<script src=".src/index.js"></script>-->
10     <script src="main.js"></script>
11 </body>
12 </html>
```

### 【代码说明】

- 第 09 行代码中，注销了通过<script>标签引入的 index.js 脚本文件。
- 第 10 行代码中，增加了通过<script>标签引入了 main.js 脚本文件。

### （8）通过 Webpack 工具打包 Node 应用。

通过执行 Webpack 打包命令，将会从 index.js 脚本作为入口起点，最终打包含并输出 main.js 脚本文件。命令如下：

```
npx webpack
```

npx 命令是从 NPM 5.2+ 版本开始提供的命令，可以自动执行 npm 包内的二进制安装文件。关于“npx webpack”命令的执行过程，如图 1.39 所示。

```
Node.js command prompt
E:\vueprojects>npx webpack
Hash: c46d81b69d32c9d70acb
Version: webpack 4.43.0
Time: 337ms
Built at: 2020-06-08 12:25:12 F10: PM-
  Asset      Size  Chunks      Chunk Names
main.js  72.1 KiB       0  [emitted]  main
Entrypoint main = main.js
[1] ./src/index.js 403 bytes { } [built]
[2] (webpack)/buildin/global.js 472 bytes { } [built]
[3] (webpack)/buildin/module.js 497 bytes { } [built]
+ 1 hidden module

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value. Set
'mode' option to 'development' or 'production' to enable defaults for each environment.
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack.js.org/configuration/mode/
E:\vueprojects>
```

图 1.39 通过 Webpack 打包 Node 应用

如图 1.39 中的箭头所示，通过 Webpack 打包后自动生成了 main.js 脚本文件。至于后面的警告信息先不用理会，后面会通过相关配置消除该警告信息。

下面我们返回 VS Code 开发工具的源代码目录（dist），看一下有没有什么变化，效果如图 1.40 所示。

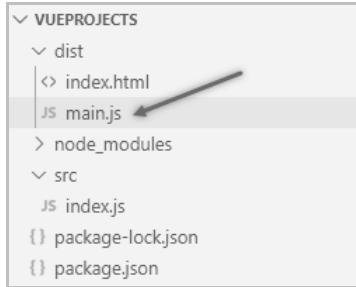


图 1.40 更新“分发（dist）”目录结构

如图 1.40 中的箭头所示，在“分发（dist）”目录下自动生成了一个 main.js 脚本文件，这就是通过 Webpack 工具打包生成的 bundle 包文件。下面，我们来见识一下这个 main.js 脚本文件的庐山真面目，如图 1.41 所示。

```
JS main.js ×
dist > JS main.js > ⓘ <function> ⓘ <function>
1 !function(n){var t={};function r(e){if(t[e])return t[e].exports;var u=t[e]=(i:e,l:!1,exports:{});return n[e].call(u,exports,u.u.exports,r),u.l=!0,u.exports}r.m=n,r.c=t,r.d=function(n,t,e){r.o(n,t)||Object.defineProperty(n,t,{enumerable:!0,get:e}),r.r=function(n){"undefined"!=typeof Symbol&&Symbol.toStringTag&&Object.defineProperty(n,t,{symbol:"Module"},{value:n});},r.t=function(n,t){if(1&t&&(n=r(n)),8&t) return n;if(4&t&&"object"==typeof n&&n.__esModule) return n;n=var e=Object.create(null);if(r.r(e),Object.defineProperty(e,"default",{enumerable:!0,value:n}),2&t&&"string"!=typeof n)for(var u in n)r.d(e,u,function(t){return [t].bind(null,u);},r.r=n=function(n){var t=n&&n.__esModule?function(){return n.default}:function(){return n};return r.d(t,"a",t),r.o=function(n,t){return Object.prototype.hasOwnProperty.call(n,t)},r.p="",r(r.s=1)([function(n,t){(function(n,e){var u
2 /**
3 * @license
4 * Lodash <https://lodash.com/>
5 * Copyright OpenJS Foundation and other contributors <https://openjsf.org/>
6 * Released under MIT license <https://lodash.com/license>
7 * Based on Underscore.js 1.8.3 <http://underscorejs.org/LICENSE>
8 * Copyright Jeremy Ashkenas, DocumentCloud and Investigative Reporters & Editors
9 */function() var i="Expected a function",o=_lodash_placeholder_,f=[["ary",128],["bind",1],["bindKey",2],["curry",8],["curryRight",16],["flip",512],["partial",32],["partialRight",64],["rearg",256]],a=[{"object Arguments":c=[["object Array"]],l=[["object Boolean"]],v=[["object Date"]],h=[["object Function"]],p=[["object GeneratorFunction"]],d=[["object Map"]],_=[["object Number"]],g=[["object Object"]],y=[["object RegExp"]],b=[["object Set"]],w=[["object String"]],m=[["object Symbol"]],x=[["object WeakMap"]],j=[["object ArrayBuffer"]],A=[["object DataView"]],o=[["object Float32Array"]],k=[["object Float64Array"]],s=[["object Int8Array"]],E=[["object Int16Array"]],I=[["object Int32Array"]],R=[["object Uint8Array"]],z=[["object Uint16Array"]],L=[["object Uint32Array"]],C=[["object _p \+_";g,W=[["b\_\_p \+_"]]\+/g,T=[("e\(\.*?\)\|b\_\_t\|")]\+\n';g,U=&{amp|lt|gt|quot#\39];g,B=[&<&>"/g,$=RegExp(U.source),P=RegExp(B.source),D=<%-(\s\S)+%>/g,M=<%-(\s\S)+%>/g,N=<%-(\s\S)+%>/g,F=/\.\[[?:\[\]]*\[[?:\[\]]*\((?:\?(1)\|\[\]\|\.\.)*\1)\]\]/,a=/^\w*\$/g,Z=/^[\[\]]+|[?:(-\d+(?:\.\d+)?)([""])(?:\?(1)\|\[\]\|\.\*)\?2]\]|(?:\?.|\[\]\?(?:\.\|\[\]\$))/g,K=/(^\w*\w+?)([\[\]\|]g,V=RegExp(K.source),G=/^\w+\s+\$g,H=/^\w+\$/,J=/\w+\$/,Y=/\{(?:\.\|\n\|\w*\|)[wrapped with .+]\*\}\?n?/,Q=/\{\n\|\w*\|)[wrapped with .+]\*\}/,X=/\? & /,nn=/^\w+x00-\w2f\x3a-\w40\x5b-\w60\x7b-\w7f]+/g,tn=/\w(\w)?/g,rn=/\$({\{\w\}\}}*(?:\.\[\w\]\*)*)\}/g,en=/\w*\$/g,un=/^[-+]0x[0-9a-f]+\$/i,on=/^0b[01]+\$/i,fn=/^[\object .+?constructor\$/],an=/^0o[0-7]+\$/i,cn=/^(?:0
1-9)\d*\$/i,ln=[/\xc0-\xd6\xd8-\xf6\xf8-\xff\u0100-\u017f]/g,sn=(\$^)/,vn=['\n\r\u2028\u2029\b/g]
```

图 1.41 在浏览器中运行 HTML 网面文件

如图 1.41 所示，main.js 脚本文件是一个经过代码压缩的打包文件（提高运行速度），但代码阅读起来是相当有难度的。另外，还可以看到 main.js 脚本文件将 Lodash 包也一并压缩进

去了。

(9) 此时可以再次测试这个 Node 应用，仍旧是在浏览器中运行 index.html 页面文件，效果如图 1.42 所示。

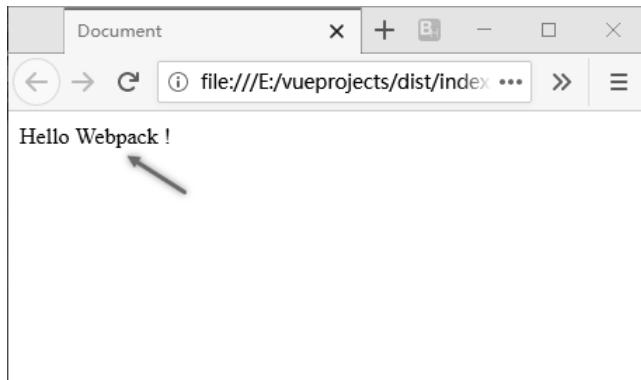


图 1.42 在浏览器中运行 HTML 页面文件

如图 1.42 中的箭头所示，浏览器页面中显示了 index.js 脚本代码第二次更新修改后的执行结果（Hello Webpack!），说明经过 Webpack 工具打包后生成的 main.js 脚本文件正确运行了。

#### (10) 使用 Webpack 配置文件。

细心的读者可能会发现上面的第（8）步有些奇怪，在 Webpack 打包过程中似乎隐藏了一些细节，在 main.js 脚本文件中看不到具体配置信息。其实，这是通过 Webpack 工具的默认配置所完成的，因此对于简单的应用通常不需要配置。

但是，Webpack 工具有一个配置文件（webpack.config.js），在应对较为复杂的 Node 应用时就会派上用场。在应用 Webpack 工具时，使用配置文件（webpack.config.js）可有效地避免在控制台终端以手动方式输入大量烦琐的命令，而是以自动化的方式完成压缩、合并和打包等复杂操作。

Webpack 配置文件（webpack.config.js）一般放置在 Node 应用的根目录下，与 Node 应用配置文件（package.json）处于同一级，效果如图 1.43 所示。

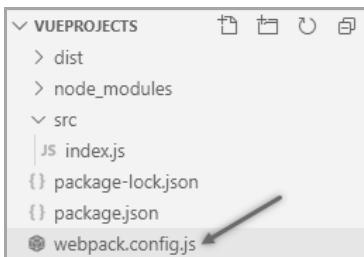


图 1.43 Webpack 配置文件——webpack.config.js

如图 1.43 中的箭头所示，将 Webpack 配置文件（webpack.config.js）放置于 Node 应用（vueprojects）根目录下。然后，简单地配置一下 Webpack 配置文件（webpack.config.js），代码如下：

**【代码 1-8】**

```

01 const path = require('path');
02 /**
03  * module : exports
04 */
05 module.exports = {
06   entry: './src/index.js',
07   output: {
08     filename: 'bundle.js',
09     path: path.resolve(__dirname, 'dist')
10   },
11   mode: 'development'
12 };

```

**【代码说明】**

- 第 06 行代码中，通过 entry 参数配置 Webpack 工具打包的入口（index.js 脚本文件的路径）。
- 第 07 行代码中，通过 output 参数配置 Webpack 工具打包的出口，具体内容如下：
  - 第 08 行代码中，通过 filename 参数配置打包后的出口脚本文件的名称（bundle.js）。
  - 第 09 行代码中，通过 path 参数解析了打包后的出口文件路径。
- 第 11 行代码中，通过 mode 参数配置打包模式（development 表示为开发模式，该模式会对脚本代码进行压缩）。

（11）重新配置 package.json 配置文件。

在原始 package.json 配置文件中的 scripts 节点下添加一个 build 参数，具体脚本代码如下：

**【代码 1-9】**

```

01 {
02   "name": "vueprojects",
03   "version": "1.0.0",
04   "description": "",
05   "private": true,
06   "scripts": {
07     "test": "echo \\"Error: no test specified\\" && exit 1",
08     "build": "webpack"
09   },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "devDependencies": {
14     "webpack": "^4.43.0",
15     "webpack-cli": "^3.3.11"
16   },
17   "dependencies": {
18     "lodash": "^4.17.15"

```

```
19 }
20 }
```

**【代码说明】**

- 第08行代码中，新添加了一个build参数，参数值指向webpack命令。

(12) 再次修改index.js脚本文件和index.html页面文件。

下面是再次修改index.js脚本文件后的具体代码：

**【代码 1-10】**

```
01 import _ from 'lodash';
02 /**
03  * func - create div component
04 */
05 function divComp() {
06     var eleDiv = document.createElement('div');
07     // TODO: use Lodash '' to join string.
08     eleDiv.innerHTML = _.join(['Hello', 'Webpack', '&', 'NodeJS', '!'], ' ');
09     // TODO: return element div
10     return eleDiv;
11 }
12 // TODO: append div to body
13 document.body.appendChild(divComp());
```

**【代码说明】**

- 第08行代码中，通过Lodash插件库重新连接了一组字符串，用于显示在页面中。

由于在Webpack配置文件(webpack.config.js)中重新定义了出口脚本文件(bundle.js)，因此还要再修改一下index.html页面文件，修改后的代码如下：

**【代码 1-11】**

```
01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04     <meta charset="UTF-8">
05     <meta name="viewport" content="width=device-width, initial-scale=1.0">
06     <title>Document</title>
07 </head>
08 <body>
09     <!--<script src="./src/index.js"></script>-->
10     <!--<script src="main.js"></script>-->
11     <script src="bundle.js"></script>
12 </body>
13 </html>
```

**【代码说明】**

- 第10行代码中，再次注销了通过<script>标签引入的main.js脚本文件。

- 第 11 行代码中，增加了通过<script>标签引入 bundle.js 脚本文件。

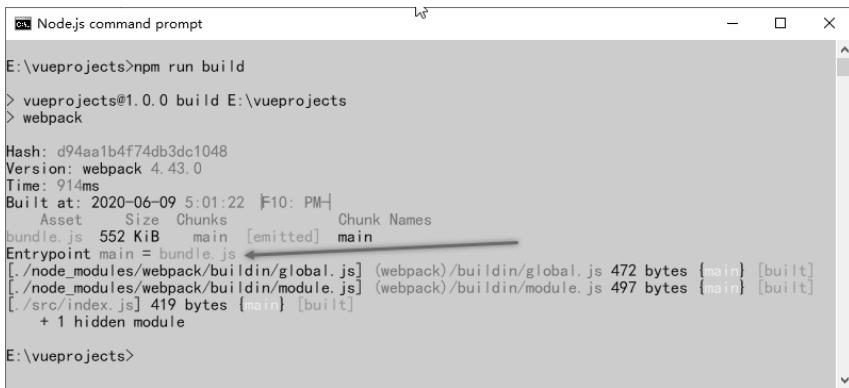
(13) 再次通过 Webpack 工具打包 Node 应用。

由于在 package.json 配置文件的 script 节点中，重新定义了脚本执行命令，因此需要输入新的命令来执行 Webpack 打包操作，命令如下：

```
npm run build // build parameter is defined in package.json
```

上面命令行中的 build 参数是在 package.json 配置文件中定义的，读者可以返回到配置文件中去查看一下。

“npm run build” 命令的执行过程，如图 1.44 所示。



```
Node.js command prompt
E:\vueprojects>npm run build
> vueprojects@1.0.0 build E:\vueprojects
> webpack

Hash: d94aa1b4f74db3dc1048
Version: webpack 4.43.0
Time: 914ms
Built at: 2020-06-09 5:01:22 | 10: PM
Asset      Size  Chunks   Chunk Names
bundle.js  552 KiB  main [emitted]  main
Entrypoint main = bundle.js
[./node_modules/webpack/buildin/global.js] (webpack)/buildin/global.js 472 bytes {main} [built]
[./node_modules/webpack/buildin/module.js] (webpack)/buildin/module.js 497 bytes {main} [built]
[./src/index.js] 419 bytes {main} [built]
  + 1 hidden module

E:\vueprojects>
```

图 1.44 通过 Webpack 配置文件打包 Node 应用

如图 1.44 中的箭头所示，通过 Webpack 打包后自动重新生成了 bundle.js 脚本文件。另外，读者可能注意到警告信息已经不见了，是因为在【代码 1-8】中的第 11 行代码添加配置了 mode 参数。

下面我们再次返回 VS Code 开发工具的源代码目录（dist），看一下有没有什么变化，效果如图 1.45 所示。

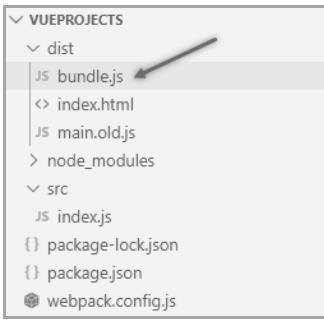


图 1.45 再次更新“分发（dist）”目录结构

如图 1.45 中的箭头所示，在“分发（dist）”目录下自动生成了一个 bundle.js 脚本文件，这就是通过 Webpack 工具配置文件重新打包生成的 bundle 包文件。这个重新生成的 bundle.js 脚本文件符合 ES6 标准规范，由于代码比较长在本书中就不列出来了。

(14) 此时可以再次测试一下这个 Node 应用，仍旧是在浏览器中运行 index.html 页面文

件，效果如图 1.46 所示。

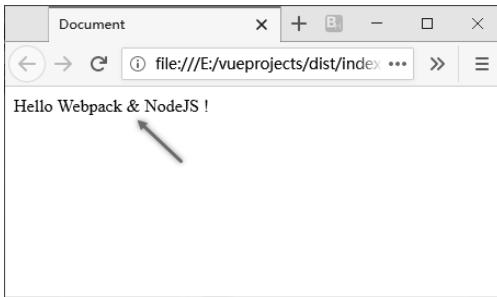


图 1.46 在浏览器中运行 HTML 页面文件

如图 1.46 中的箭头所示，浏览器页面中显示了 index.js 脚本代码第三次更新修改后的执行结果（Hello Webpack & NodeJS!），说明通过 Webpack 工具配置文件打包后生成的 bundle.js 脚本文件正确运行了。

#### （15）通过 webpack-dev-server 插件实现 Node 应用热加载。

Webpack 工具的打包操作功能固然十分强大，不过相信读者也发现了每次的打包操作都需要手动进行，之后的页面测试过程也很粗放（直接运行 HTML 页面文件）。于是，研发人员为了配合 Webpack 工具就设计了一个 webpack-dev-server 插件，可以实现 Node 应用的自动打包和热加载功能。webpack-dev-server 插件的安装方法与 webpack 方式一致，命令如下：

```
npm install webpack-dev-server --save-dev
```

安装完毕后，还需要在 package.json 配置文件的 script 节点中，新增一个 dev 参数的配置信息为插件名称（webpack-dev-server）。

然后，通过直接运行“npm run dev”命令就可以执行打包操作了，效果如图 1.47 所示。

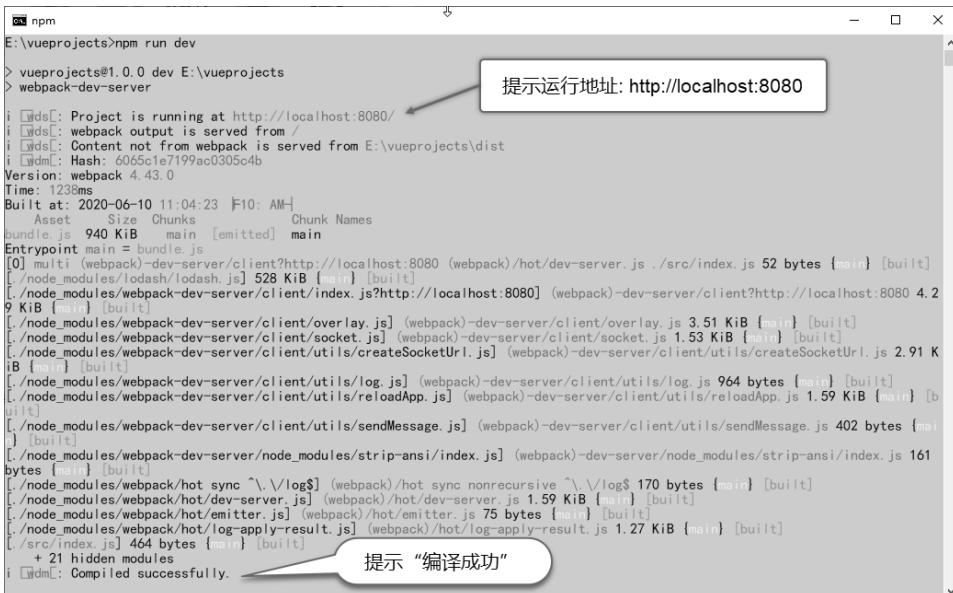


图 1.47 再次运行“npm run dev”命令进行打包

如图 1.47 中的标识所示，控制台终端中给出了 Node 应用服务端运行地址的提示信息（<http://localhost:8080/>），我们通过浏览器直接访问该地址就能测试该项目了。效果如图 1.48 所示。

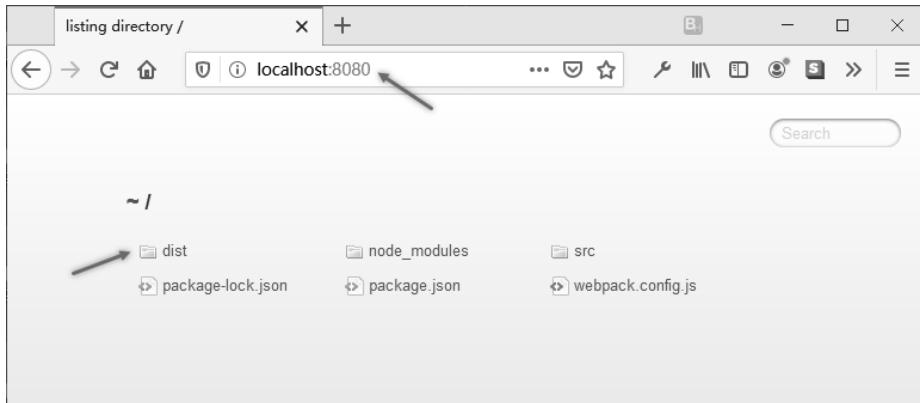


图 1.48 在浏览器中测试 Node 应用

如图 1.48 中的箭头所示，页面在打开后并不是预期的运行效果，显示的却是项目的目录结构。这里，需要再次单击“分发（dist）”目录才可以运行 index.html 页面。

接下来，我们体验一下 webpack-dev-server 插件的热加载功能。具体方法就是修改更新一下源文件（index.js）代码，然后注意观察一下控制台终端的变化，如图 1.49 所示。

```
cd npm
[./src/index.js] 464 bytes {main} [built]
  + 21 hidden modules
i  wdm: Compiled successfully.
i  wdm: Compiling...
i  wdm: Hash: eef78272d7f2bb8c5cc4
Version: webpack 4.43.0
Time: 37ms
Built at: 2020-06-10 11:17:34 F10: AM
          Asset      Size  Chunks             Chunk Names
 6065c1e7199ac0305c4b.hot-update.json  46 bytes           [emitted]  [immutable]  [hmr]
                                         bundle.js  940 KiB  main  [emitted]
main.6065c1e7199ac0305c4b.hot-update.js  1.14 KiB  main  [emitted]  [immutable]  [hmr]
Entrypoint main = bundle.js main.6065c1e7199ac0305c4b.hot-update.js
[./src/index.js] 483 bytes {main} [built]
  + 35 hidden modules
i  wdm: Compiled successfully.
```

图 1.49 通过 webpack-dev-server 实现热加载

如图 1.49 中的箭头所示，在没有经过手动命令操作的情况下，webpack-dev-server 插件重新编译了源文件，这就是所谓的“热加载”功能。不过，比较遗憾的是浏览器上显示的页面并不会随之更新，需要手动操作来完成更新。

#### （16）配合 HotModuleReplacementPlugin 插件实现 Node 应用热更新。

为了解决 webpack-dev-server 插件无法实现热更新操作的问题，这里需要借助 Webpack 工具自带的 HotModuleReplacementPlugin 插件来配合 webpack-dev-server 插件完成热更新功能。

使用 HotModuleReplacementPlugin 插件时，需要在 Webpack 配置文件（webpack.config.js）

中增加 plugins 节点的配置，代码如下：

```
plugins: [
  new webpack.HotModuleReplacementPlugin()
]
```

然后，再次尝试修改更新一下源文件（index.js）中的代码，注意观察一下浏览器页面显示内容的变化，如图 1.50 所示。

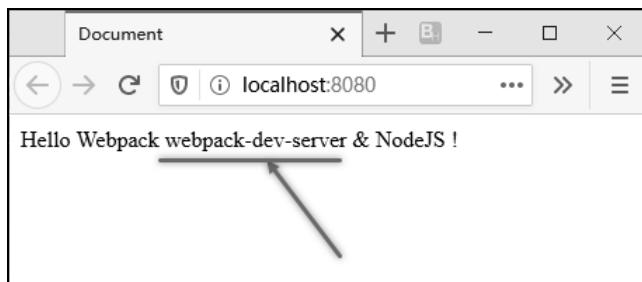


图 1.50 浏览器页面显示内容的热更新

如图 1.50 中的箭头所示，浏览器页面的内容通过“热更新”功能进行了自动更新。至此，关于使用 VS Code 开发工具通过 Webpack 开发 Node 应用的基本过程就介绍完了，希望能够对读者有所帮助。

# 第 2 章

## Vue.js 基础介绍与环境搭建

Vue.js（发音：/vju:/）是一个用于构建用户界面的渐进式 JavaScript 框架，是较早提出采用“自底向上、逐层设计”方式进行 Web 前端应用开发的开源框架。目前，随着 Vue.js 框架的不断完善与日趋成熟，已经可以与 Angular 框架和 React 框架“平起平坐”了。因此，Angular、React 和 Vue.js 这三大前端开发框架并称为 Web 前端设计框架的“三驾马车”。

本章针对 Vue.js 框架整体进行介绍，包括 Vue.js 的基础知识、发展历史、环境搭建以及基本开发等方面的内容，目的是为了后面详细讲解 Vue.js 开发做好前期铺垫。

通过本章的学习可以：

- 掌握 Vue.js 基础知识。
- 掌握 Vue.js 发展历史及性能特点。
- 掌握 Vue.js 开发环境搭建及简单应用的开发。

## 2.1 Vue.js 基础

本节介绍 Vue.js 的基础知识、发展历史、组织架构，以及具体应用等方面的内容。

### 2.1.1 Vue.js 简介

Vue.js 是一个用于构建用户界面的渐进式 JavaScript 框架，采用了“自底向上、增量开发”的设计方式。Vue.js 的核心是只关注视图层，便于与第三方库或既有项目进行整合。细心的读者会发现，这几句话里面提到了几个关键词汇（如：“渐进式”“自底向上、增量开发”“视图层”），下面就这些方面的内容展开讲解一下。

首先，这个“渐进式”的概念就比较新颖，那么具体是什么含义呢？所谓“渐进式”就是一开始不需要设计人员完全掌握其框架的全部功能特性，可以放到后续步骤中以逐步的增加方式来完成，这样在每一步都可以更专注于当前的任务。从本质上讲，这就是设计模式上的优化

与进步。而与 Vue.js 框架相对应的 Angular 框架和 React 框架均不是严格意义上的渐进式框架，均具有一定程度上的个性化及排他性。

其次，关于“自底向上、增量开发”这个概念主要描述的是设计方式。这种设计方式的思路就是，先设计好基础骨架，然后再逐步向上扩充，完善功能和效果。采用“自底向上、增量开发”的设计方式，可以有效地提高开发效率，避免不必要的重复工作。

最后，这个“视图层”的概念指的就是著名的 MVVM 架构模型中的 View 层。Vue.js 是一个基于 MVVM 架构模型实现了“双向数据绑定”功能的前端 JavaScript 库，其关注的核心点就是 View 层。另外，这里提到的关于 MVVM 架构模型和“双向数据绑定”的内容，会在下文中详细介绍。

## 2.1.2 Vue.js 发展历史

Vue.js 最早发布于 2014 年左右，开发者是曾在 Google 工作的中国籍开发人员——尤雨溪。根据作者本人的描述，Vue.js 框架的定位就是为前端开发者提供一个低门槛、高效率，又同时能够伴随用户成长的前端框架。

Vue.js 框架的发展历程主要如下：

- (1) 实验 (experiment) 阶段 (2013 年年中到 2014 年 2 月)。
- (2) 0.x 版本阶段 (2014 年 2 月 ~ 2015 年 10 月)。
- (3) 1.x 版本阶段 (2015 年 10 月 ~ 2016 年 9 月)，发行版名称为“Evangelion”。
- (4) 2.x 版本阶段 (2016 年 9 月 ~ 至今)，发行版名称为“Ghost in the Shell”。

关于 Vue.js 框架以上几个版本更新发展的过程，大致介绍如下：

在早期 0.x 版本阶段，内容更新主要集中在 Vue 模板语法上。而在 1.x 版本阶段期间，模板语法就日趋稳定了。在 2.x 版本阶段，内容更新专注于内部的渲染机制功能，这期间引入了著名的 Virtual DOM 机制，从而实现了服务端渲染、原生渲染、手写渲染函数等强大的设计功能。目前，它的 3.x 版本正处于开发过程中，更新目标主要集中于利用 ES6 (ECMAScript 2015) 版本的新特性、改进内部架构，以及性能优化等方面上。

## 2.1.3 Vue.js 与 MVVM 架构模型

软件设计的架构模型往往决定着一个开发框架的特性与性能，就好比基因对于人类的决定性因素一样。架构模型从 MVC 经过 MVP、再到 MVVM，每一步都体现了开发人员对于设计模式的不断完善。

MVVM (Model-View-ViewModel) 架构其实本质上就是对 MVC 架构的改进版。MVC 架构可谓是大名鼎鼎，相信大多数读者在刚开始接触架构模型时学习的就是该架构。从 MVVM 架构模型的命名来看，Model-View-ViewModel 中的 Model (模型) 和 View (视图) 沿用了下来，改变的就是 Controller (控制器) 被 ViewModel 替换了。那么，ViewModel 代表什么概念？Vue.js 借鉴了 MVVM 的什么设计理念呢？

ViewModel 在 MVVM 中负责在 Model（模型）和 View（视图）中间的桥接工作，当 Model（模型）改变时通过 ViewModel 通知 View（视图），反之亦然。Vue.js 框架专注于 View（视图层），将视图的状态和行为抽象化，并于业务逻辑分开来设计。Vue.js 虽然没有完全照搬 MVVM 模型，但对于 ViewModel 的设计有独到之处。当 View（视图）改变时会触发事件，通过 ViewModel 负责监听事件并同步更新 Model（模型）。

## 2.1.4 双向数据绑定

Vue.js 框架实现的一项核心功能就是“双向数据绑定”，所谓双向数据绑定就是指 View（视图）和 Model（模型）的数据相互同步。

Vue.js 框架是基于 MVVM 架构设计的。为了实现 View（视图）和 Model（模型）的数据相互同步，Vue.js 会通过 DOM Listeners 来监听并改变 Model（模型）中的数据，当 Model（模型）中的数据发生改变时，会通过 Data Bindings 来监听并改变 View（视图）中数据的展示。这一点也正是 MVVM 架构对于“双向数据绑定”的支持。

在 Vue.js 框架底层，是通过使用 JavaScript Object 对象的 `defineProperty()` 方法，重新定义了对象获取属性值和设置属性值的方法，来实现“双向数据绑定”操作的。因此，其原理仍旧是通过 JavaScript 方式实现的。

## 2.1.5 Vue.js 特点

Vue.js 是一款基于数据驱动思想开发的 JavaScript 框架，下面总结一下关于 Vue.js 框架的几个主要特点：

- Vue.js 是基于 MVVM 架构设计的、一套用于构建用户浏览器界面的、渐进式的前端 Web 框架。
- Vue.js 是基于数据驱动思想开发的 JavaScript 框架，实现了在尽可能的条件下减少繁杂的 DOM 操作。
- Vue.js 开发了一套自己的模板语言，采用虚拟 DOM 的方式渲染 HTML 页面，实现了将前后端进行分离的开发方式。
- Vue.js 的核心库只关注视图层，同时借助 MVVM 架构的特点实现了“双向数据绑定”的核心功能。
- Vue.js 只聚焦于视图层，具备能力实现单文件组件以及相对复杂的单页面应用。
- Vue.js 是一个轻巧的、高性能的、可组件化的 JavaScript 框架，设计了易于学习的 API 方法，能够非常方便地与其他前端库进行有效整合。

## 2.2 Vue.js 快速开发环境

本节主要介绍在 Windows 10 系统平台下，如何通过 Visual Studio Code 搭建 Vue.js 框架的快速开发环境。

### 2.2.1 直接通过<script>引入本地 Vue.js

Vue.js 框架本质上还是一个 JavaScript 开发库，因此仍旧可以直接通过<script>标签引入本地的 Vue.js 文件，这也是最原始的开发方式。如果读者打算使用该方式，就需要将 Vue.js 库文件下载到本地。

首先，访问 Vue.js 的中文官方网站（<https://cn.vuejs.org/>），在介绍“安装”方法的页面中可以找到 Vue.js 的库文件。Vue.js 库文件包含两个版本，分别是“开发版本”和“生产版本”。“开发版本”的下载地址为 <https://cn.vuejs.org/js/vue.js>，该版本包含完整的警告信息和调试模式。“生产版本”的下载地址为 <https://cn.vuejs.org/js/vue.min.js>，该版本删除了相关警告信息（体积更小、运行更快），用于最终打包发布时使用。



一般地，JavaScript 库文件为了区分“开发版本”和“生产版本”，会在“生产版本”的文件名中加入“min”字符串以示区别。

接下来，我们创建一个简单的 Vue 单页面文件，就是在 HTML 5 页面中直接引入 Vue.js 库文件，在页面中输出一行简单的欢迎信息（“Hello Vue.js!”），代码如下：

【代码 2-1】（详见源代码 hellovue 目录中的 hellovue-script.html 文件）

```

01 <!DOCTYPE html>
02 <html lang="en">
03   <head>
04     <meta charset="UTF-8">
05     <meta name="viewport" content="width=device-width, initial-scale=1.0">
06     <script src="vue.min.js"></script>
07     <title>Hello Vue.js</title>
08   </head>
09   <body>
10     <div id="app">
11       {{ message }}
12     </div>
13     <script>
14       var vApp = new Vue({
15         el: '#app',
16         data: {
17           message: 'Hello Vue.js!'

```

```

18      }
19    })
20  </script>
21 </body>
22 </html>

```

### 【代码说明】

- 第 06 行代码中，通过<script>标签引入了本地的 vue.min.js 库文件。
- 第 10~12 行代码中，通过<div>标签定义了一个层元素及其 id 属性值（id="app"）。第 11 行代码中的双大括号 “{{ }}” 是 Vue.js 框架专用的模板语法（Mustache 语法），双大括号内的 message 为数据绑定对象。（后文中会对 Vue.js 语法进行系统的介绍）
- 第 13~20 行代码定义的是 Vue.js 脚本语言，通过“new Vue()”构造函数实例化 Vue 对象，这是创建 Vue 对象的入口。具体内容如下：
  - 第 15 行代码通过 el 属性绑定 DOM 元素（"#app"），注意“#”前缀标识符的使用。
  - 第 16~18 行代码通过 data 属性定义具体数据，第 17 行代码定义的 message 属性对应第 11 行代码定义的数据绑定对象（message），从而实现将数据内容渲染到页面中指定的 DOM 元素上。

在上面的代码中，HTML 页面代码与 Vue.js 脚本代码是写在同一个页面文件中的，我们可以通过运行浏览器来进行测试，如图 2.1 所示。

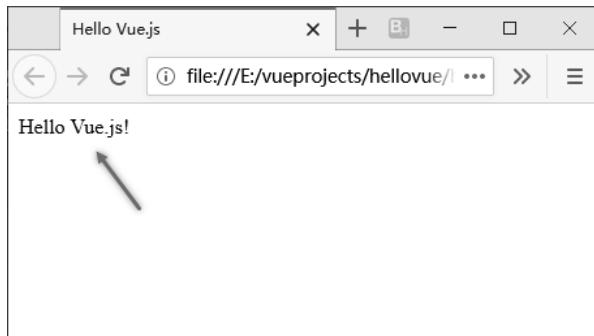


图 2.1 通过浏览器测试 Vue.js 代码

如图 2.1 中的箭头所示，【代码 2-1】中第 17 行代码定义的 message 字符串信息被成功渲染到页面中显示了。

## 2.2.2 通过 CDN 方式引入 Vue.js

对于 CDN 方式，相信大多数读者都比较熟悉，Vue.js 框架支持多种 CDN 的使用方式，下面进行详细介绍。

第 1 种是 cdnjs 方式，具体如下：

```
https://cdnjs.cloudflare.com/ajax/libs/vue/2.x.xx/vue.js      // 注意版本号
```

第2种是 unpkg 方式，具体如下：

```
https://unpkg.com/vue@2.x.xx/dist/vue.js      // 注意版本号
```

第3种是 Staticfile CDN（国内）方式，具体如下：

```
https://cdn.staticfile.org/vue/2.x.xx/vue.min.js      // 注意版本号
```

以上的3种方式中，前两种是国外的CDN源，最后一种是国内的CDN源，推荐使用国外的CDN源，相对稳定且保证及时更新。

### 2.2.3 兼容ES Module的方式

在Vue.js官网还推荐一种兼容ES Module的构建文件，适用于使用原生ES Modules的开发场景，具体如下：

```
<script type="module">
  import Vue from 'https://cdn.jsdelivr.net/npm/vue@2.x.xx/dist
  /vue.esm.browser.js'
</script>
```

注意“type="module””和import命令的使用，遵循的是ECMAScript 2015规范标准。另外，这种方式也支持使用CDN源，具体如下：

```
https://cdn.jsdelivr.net/npm/vue/dist/vue.js    // 开发环境
https://cdn.jsdelivr.net/npm/vue@2.6.11        // 生产环境，建议明确指定版本号
```

## 2.3 Vue.js 脚手架开发环境

本节主要介绍搭建Vue.js框架的脚手架（vue-cli）开发环境的方法，以及如何通过Visual Studio Code开发工具开发调试Vue.js程序等方面的内容。

### 2.3.1 安装Vue.js脚手架

所谓“脚手架”就是为了快速搭建应用程序开发框架而设计开发的自动构建工具。在当前各种Web开发框架流行的今天，大部分前端开发工具和框架都设计了自己的“脚手架”工具，而Vue.js框架的脚手架就是vue-cli命令行工具。

Vue.js框架自身的迭代速度很快，目前主流的是Vue 2版本和Vue 3+版本。相比较而言，Vue 3+版本在Vue 2版本的基础上增加了不少新特性和新功能。因此，也造成了Vue 3+版本与Vue 2版本在使用上多少有些差异。

在Vue.js脚手架（vue-cli）的使用上，Vue 3+版本与Vue 2版本也存在着不同，下面我们将主要以Vue 3+版本进行介绍。假设当前系统开发环境已经安装好新版的NPM工具和Webpack

工具，这样就可以继续安装 vue-cli 命令行工具了，具体命令如下：

```
npm install -g @vue/cli      // 注意：Vue 3+版本为“@vue/cli”，Vue 2 版本为“vue/cli”
# OR
yarn global add @vue/cli    // yarn 是 Facebook 提供的依赖管理工具，类似 npm 工具
```

在命令行控制台中输入以上的命令，安装效果如图 2.2 所示。

```
C:\Users\KingW>npm install -g @vue/cli
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
npm WARN deprecated chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependencies.
npm WARN deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade to fsevents 2.
[...]
  [ ] | extract:apollo-engine-reporting: sill tarball trying mimic-fn@2.1.0 by hash: sha512-0qb0k5oEqaZ8WXWylu9HJz
  [ ] | extract:indent-string: sill extract indent-string@3.2.0 extracted to C:\Users\KingW\AppData\Roaming\npm\node
C:\Users\KingW\AppData\Roaming\npm\vue -> C:\Users\KingW\AppData\Roaming\npm\node_modules@\vue\cli\bin\vue.js

> core-js@3.6.5 postinstall C:\Users\KingW\AppData\Roaming\npm\node_modules@\vue\cli\node_modules\core-js
> node -e "try[require('./postinstall')].catch(e)[]"

Thank you for using core-js (https://github.com/zloirock/core-js) for polyfilling JavaScript standard library!
The project needs your help! Please consider supporting of core-js on Open Collective or Patreon:
  > https://opencollective.com/core-js
  > https://www.patreon.com/zloirock

Also, the author of core-js (https://github.com/zloirock) is looking for a good job →

> @apollo/protobufjs@1.0.4 postinstall C:\Users\KingW\AppData\Roaming\npm\node_modules@\vue\cli\node_modules\@apollo\protobufjs
> node scripts/postinstall

> nodemon@1.19.4 postinstall C:\Users\KingW\AppData\Roaming\npm\node_modules@\vue\cli\node_modules\nodemon
> node bin/postinstall || exit 0

Love nodemon? You can now support the project via the open collective:
  > https://opencollective.com/nodemon/donate

> ej@2.7.4 postinstall C:\Users\KingW\AppData\Roaming\npm\node_modules@\vue\cli\node_modules\ej
> node ./postinstall.js

Thank you for installing EJS: built with the Jake JavaScript build tool (https://jakejs.com/)

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@^1.2.7 (node_modules@\vue\cli\node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
npm WARN jscodeshift@0.9.0 requires a peer of @babel/preset-env@^7.1.6 but none is installed. You must install peer dependencies yourself.

+ @vue/cli@4.5.2
added 1095 packages from 663 contributors in 493.715s
```

图 2.2 安装 vue-cli 命令行工具

如图 2.2 中的箭头所示，当前安装的是新版的“vue/cli@4.5.2”命令行工具，此外还自动安装其他一些依赖工具。

如果想验证一下 vue-cli 命令行工具是否安装成功了，可以通过“vue -V”命令查看 vue-cli 命令行工具的版本号来进行。注意，这里的参数“-V”要使用大写字母。效果如图 2.3 所示。

```
C:\Users\KING>vue -V
@vue/cli 4.5.2
```

图 2.3 验证 vue-cli 命令行工具版本号

如图 2.3 中的箭头所示，验证的版本号与安装后提示的版本号相对应，说明 vue-cli 命令行工具安装成功了。

**提 示**

如果想通过 Yarn 工具进行安装，首先要下载 Yarn 的工具包并安装在系统本地，然后在命令行控制台输入上面的命令即可。

### 2.3.2 通过 Vue.js 脚手架进行快速原型开发测试

Vue.js 脚手架支持快速原型开发（Instant Prototyping）方式，就是通过单个的 Vue 页面进行简单应用的设计。要使用这种快速原型开发（Instant Prototyping）方式，需要添加对 vue-cli 命令行工具的服务支持，具体操作方法如下：

```
npm install -g @vue/cli @vue/cli-service-global
# or
yarn global add @vue/cli @vue/cli-service-global
```

**提 示**

如果之前已经安装过“@vue/cli”工具包，这里就可以省略“@vue/cli”命令参数，直接安装“@vue/cli-service-global”服务即可。

在成功添加好对 vue-cli 命令行工具的服务支持后，就可以通过“vue serve”命令进行快速原型开发测试了。

这里，我们通过 Visual Studio Code 开发工具进行 Vue.js 脚手架的快速原型开发测试。在开发测试前，需要在 VS Code 开发工具中安装一个名称为“Vetur”的 Vue.js 扩展插件，该扩展插件添加了 VS Code 开发工具对 Vue 文件的支持。

然后，就可以通过 VS Code 开发工具创建一个真正的 Vue 单页面文件（hellovue.vue，注意文件后缀名为 vue），在页面中输出一行简单的欢迎信息（“Hello Vue.js!”），代码如下：

**【代码 2-2】**（详见源代码 hellovue 目录中的 hellovue.vue 文件）

```
01 <template>
02   <h3>Hello Vue.js!</h3>
03 </template>
```

**【代码说明】**

在上面的代码中，<template>是 Vue.js 框架的模板语法，而<h3>就是最基本的 HTML 标签语法。

最后，通过“vue serve”命令进行测试，具体命令如下：

```
vue serve hellovue.vue
```

在命令行控制台中执行上面的命令，效果如图 2.4 所示。

```

[1] Node.js command prompt - vue serve hellovue.vue
E:\vueprojects\hellovue>vue serve hellovue.vue
[NFO] Starting development server...
98% after emitting

[DONE] Compiled successfully in 12396ms
12:45:38 F10: PM

App running at:
- Local: http://localhost:8080/
- Network: http://192.168.199.181:8080/

Note that the development build is not optimized.
To create a production build, run npm run build.

```

打开  
http://localhost:8080  
网址进行测试

图 2.4 通过“vue serve”命令进行测试

如图 2.4 中的箭头和标识所示，命令行提示信息中显示通过“<http://localhost:8080/>”地址可以运行测试 hellovue.vue 文件，效果如图 2.5 所示。

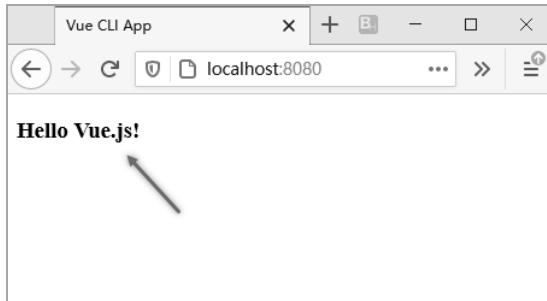


图 2.5 通过浏览器测试 Vue.js 文件

如图 2.5 中的箭头所示，【代码 2-2】中第 02 行代码定义的字符串信息被成功渲染到页面中显示了。

### 2.3.3 通过 Vue.js 脚手架进行打包

上一小节介绍的 vue-cli 命令行工具的服务（@vue/cli-service-global）支持，除了通过使用“vue serve”命令直接启动 HTTP 服务测试 Vue.js 单页面文件应用的方式，还支持使用“vue build”命令进行打包并测试的方式。

通过“vue build”命令进行打包的方法如下：

```
vue build hellovue.vue
```

在命令行控制台中执行上面的命令，效果如图 2.6 所示。

```
E:\vueprojects\hellovue>vue build hellovue.vue
\ Building for production...
[DONE] Compiled successfully in 5001ms
File Size Gzipped 2:31:07 F10: PM-
dist\js\chunk-vendors.d235b071.js 65.57 KiB 23.60 KiB
dist\js\app.cfacfa55.js 1.79 KiB 0.90 KiB
Images and other types of assets omitted.
[DONE] Build complete. The dist directory is ready to be deployed.
[INFO] Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html
```

图 2.6 通过“vue build”命令进行打包

如图 2.6 中的箭头所示，执行“`vue build hellovue.vue`”命令将通过生产模式对 Vue.js 文件进行打包，打包成功后的文件均输出到“dist”目录下。可以通过 VS Code 开发工具查看一下目录的状态变化，如图 2.7 所示。

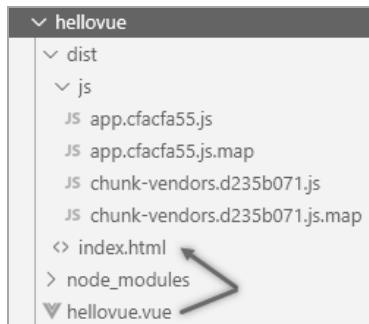


图 2.7 通过“vue build”命令打包输出到 dist 目录

如图 2.7 中的箭头所示，通过“`@vue/cli-service-global`”服务将单一 Vue.js 文件（`hellovue.vue`）打包输出到 dist 目录。在 dist 目录中，自动生成了一个 `index.html` 页面文件和一组 js 文件和 map 文件（js 子目录内）。最后，就可以将 dist 目录中的文件直接部署到服务器中去测试了。

为了加快效率，这里先不用那些常规的 Web 服务器来测试，可以通过 Node 内置的 `http-server` 扩展服务进行简单的测试。首先，在命令行控制台中进入 dist 目录，然后输入 `http-server` 或 `hs` 命令启动 Node 服务，效果如图 2.8 所示。

```
E:\vueprojects\hellovue\dist>hs
Starting up http-server, serving .
Available on:
http://10.147.18.42:8080
http://192.168.174.1:8080
http://192.168.234.1:8080
http://192.168.199.181:8080
http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

图 2.8 启动 http-server 服务

如图 2.8 中的箭头所示，通过浏览器运行“<http://localhost:8080>”地址启动 HTTP 服务，效果如图 2.9 所示。

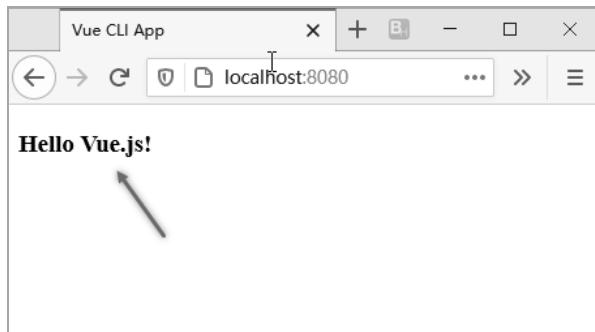


图 2.9 测试 dist 打包目录

如图 2.9 中的箭头所示，最终在浏览器中的显示效果与图 2.5 完全一致，说明打包输出的 dist 目录是正确的。

### 2.3.4 通过 Vue.js 脚手架创建应用

Vue.js 脚手架功能十分强大，通过 vue-cli 命令行工具的“vue create”命令，可以直接创建 Vue.js 应用项目，该应用项目会自动生成若干必要的框架文件，以及一个默认的 Vue 单文件页面。同时，这个默认的 Vue 单文件页面已经是一个最基本的 Vue.js 应用了，通过“npm run”命令就可以直接运行测试该应用项目。下面具体介绍一下通过 Vue.js 脚手架创建应用项目及进行运行测试的过程。

(1) 选定好打算创建 Vue.js 应用项目的目录，进入该目录的命令行控制台，通过“vue create”命令创建指定的项目名称（createvue），具体命令如下：

```
vue create createvue
```

(2) 在命令行控制台中执行上面的命令，效果如图 2.10 所示。

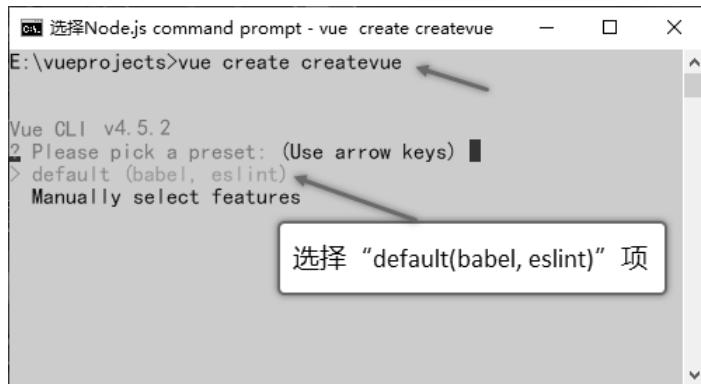


图 2.10 通过“vue create”命令创建应用项目（一）

(3) 如图 2.10 中的箭头和标识所示，继续选择默认的“default(babel, eslint)”项后确认

回车，效果如图 2.11 所示。

图 2.11 通过“vue create”命令创建应用项目（二）

(4) 如图 2.11 中所示，`vue-cli` 命令行工具将会创建刚刚指定的 `createvue` 项目，并安装项目所需的相关插件，如图 2.12 所示。

图 2.12 通过“vue create”命令创建应用项目（三）

(5) 如图 2.12 中的箭头和标识所示，安装过程需要一段时间，此时需要耐心等待一会儿。等到安装过程全部完成后，命令行控制台会提示如图 2.13 所示的信息。

```

added 1218 packages from 847 contributors in 483.917s
42 packages are looking for funding
  run `npm fund` for details
  □ Invoking generators...
  □ Installing additional dependencies...
added 53 packages from 36 contributors in 48.234s
45 packages are looking for funding
  run `npm fund` for details
  □ Running completion hooks...
  □ Generating README.md...
  □ Successfully created project createvue.
  □ Get started with the following commands:
    $ cd createvue
    $ npm run serve
  WARN Skipped git commit due to missing username and email in git config.
  You will need to perform the initial commit yourself.

```

图 2.13 通过“vue create”命令创建应用项目（三）

如图 2.13 中的箭头所示，命令行控制台中的提示信息告诉我们，进入项目目录后通过输入“npm run serve”命令就可以启动运行项目了。

(6) 接下来，我们就按照上面的提示信息测试一下，效果如图 2.14 所示。

```

E:\vueprojects\createvue>npm run serve
> createvue@0.1.0 serve E:\vueprojects\createvue
> vue-cli-service serve

[INFO] Starting development server...
98% after emitting CopyPlugin

[DONE] Compiled successfully in 11028ms
10:15:48 | 10: AM

App running at:
- Local: http://localhost:8080/
- Network: http://192.168.199.181:8080/

Note that the development build is not optimized.
To create a production build, run npm run build.
  
```

图 2.14 通过“run serve”命令运行应用项目

(7) 如图 2.14 中的箭头所示，命令行控制台中的提示信息告知我们，通过在浏览器中输入“<http://localhost:8080/>”地址就可以运行项目了，效果如图 2.15 所示。

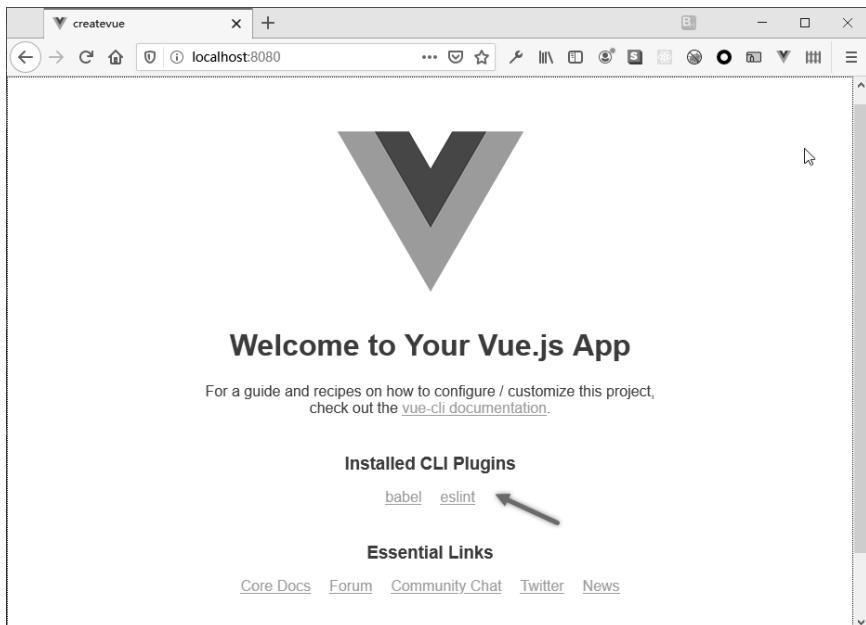


图 2.15 测试运行 Vue.js 应用项目

如图 2.15 中的箭头所示，浏览器中显示的插件信息（babel、eslint）与上面图 2.10 中的选择项是相对应的，说明这个 Vue.js 项目创建成功了。

现在，到这一步基本就完成了一个 Vue.js 项目的构建、测试和运行任务，设计人员就可以在这个 Vue.js 项目的骨架上按照自己项目的需求进行开发了。

在项目开发完成后，还有很重要一步就是项目的打包和发布，那么通过 vue-cli 命令行工具如何实现呢？其实，在上面的图 2.14 中命令行控制台已经给出了提示信息，如图 2.16 所示。

```
E:\vueprojects\createvue>npm run serve
> createvue@0.1.0 serve E:\vueprojects\createvue
> vue-cli-service serve

[INFO] Starting development server...
98% after emitting CopyPlugin

[DONE] Compiled successfully in 11028ms
          10:15:48 | F10: AM

App running at:
- Local:  http://localhost:8080/
- Network: http://192.168.199.181:8080/

Note that the development build is not optimized.
To create a production build, run npm run build.
```

图 2.16 通过“npm run build”命令打包发布应用项目（一）

如图 2.16 中的箭头所示，根据命令行控制台中的提示信息，进入项目目录后通过输入“npm run build”命令就可以对项目执行打包发布操作，效果如图 2.17 所示。

```
E:\vueprojects\createvue>npm run build
> createvue@0.1.0 build E:\vueprojects\createvue
> vue-cli-service build

| Building for production...

[DONE] Compiled successfully in 21139ms
          10:27:06 | F10: AM

File                      Size           Gzipped
dist\js\chunk-vendors.b0f460c7.js    89.18 KiB      31.93 KiB
dist\js\app.bb406f8a.js            4.62 KiB       1.64 KiB
dist\css\app.fb0c6e1c.css         0.33 KiB       0.23 KiB

Images and other types of assets omitted.

[DONE] Build complete. The dist directory is ready to be deployed.
[INFO] Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html

E:\vueprojects\createvue
```

图 2.17 通过“npm run build”命令打包发布应用项目（二）

如图 2.17 中所示，dist 目录已经构建完成，直接将该目录部署到服务器上就可以运行。这里就不再给出具体操作步骤了，读者可以参考前一小节中关于“Node 内置的 http-server 扩展服务”的内容。

### 2.3.5 通过 vue-cli 结合 Webpack 创建应用

通过 vue-cli 命令行工具还可以结合 Webpack 工具创建 Vue.js 项目应用。严格来讲，这其实是 Vue 2 版本下的操作方式，所结合的 Webpack 工具其实是 Vue.js 官方所推荐的模板。但

由于 Vue 2 版本目前也非常流行，且有大量的实际项目在使用这个版本，因此这里也向读者进行一个简单的介绍。

在 Vue 2 版本下创建应用项目同样会自动生成若干必要的框架文件，以及一个默认的 Vue 单文件页面。不过，该项目内生成的文件和目录与 Vue 3 版本下生成的略有不同，但迁移方式也不是很复杂（在后文中会具体介绍）。

下面，我们就具体介绍一下在 Vue 2 版本下通过 vue-cli 命令行工具创建应用项目及测试运行的过程。

(1) 对于 Vue 2 版本下 vue-cli 命令行工具，就不是通过“vue create”命令创建应用项目了，而是通过“vue init”命令来实现的，具体的命令格式如下：

```
vue init webpack your-project-name
```

其中，webpack 是指定的模板名称（webpack 是官方推荐的默认模板工具，当然也可以指定其他模板工具），your-project-name 是指定创建的项目名称。

另外需要注意的是，在使用“vue init”命令之前需要先安装该命令扩展工具，具体命令如下：

```
npm install -g @vue/cli-init
```

(2) 在命令行控制台中执行上面的命令，效果如图 2.18 所示。

```
Node.js command prompt
Your environment has been set up for using Node.js 12.18.1 (x64) and npm.

C:\Users\KingW>npm install -g @vue/cli-init
npm WARN deprecated vue-cli@2.9.6: This package has been deprecated in favour of @vue/cli
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/re
quest/request/issues/3142
npm WARN deprecated coffee-script@1.12.7: CoffeeScript on NPM has moved to "coffeescript"
(no hyphen)
+ @vue/cli-init@4.4.4 ←
added 253 packages from 207 contributors in 116.376s

C:\Users\KingW>
```

图 2.18 安装“vue init”命令扩展工具

(3) 如图 2.18 中的箭头所示，系统默认安装了新版本的“@vue/cli-init@4.4.4”。安装完毕后，通过“vue init”命令创建指定的项目名称（initvue），具体命令如下：

```
vue init webpack initvue
```

(4) 在命令行控制台中执行上面的命令，效果如图 2.19 所示。

```

? Project name initvue
? Project description A Vue.js project
? Author king
? Vue build standalone
? Install vue-router? No
? Use ESLint to lint your code? Yes
? Pick an ESLint preset Standard
? Set up unit tests Yes
? Pick a test runner jest
? Setup e2e tests with Nightwatch? Yes
? Should we run `npm install` for you after the project has been created? (recommended) npm
    · Generated "initvue".
  
```

图 2.19 通过“vue init”命令创建应用项目（一）

(5) 如图 2.19 中所示，命令行控制台会依次给出若干提示信息，我们按照实际需要依次进行配置即可。配置信息都完成后，就是一个相对漫长的下载安装过程了，如图 2.20 所示。

```

# npm
# Installing project dependencies ...
# =====
npm WARN deprecated extract-text-webpack-plugin@3.0.2: Deprecated. Please use https://github.com/webpack-contrib/mini-css-extract-plugin
npm WARN deprecated browserslist@2.11.3: Browserslist 2 could fail on reading Browserslist >3.0 config used in other tools.
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142
npm WARN deprecated core-js@2.6.11: core-js@<3 is no longer maintained and not recommended for usage due to the number of issues. Please, upgrade your dependencies to the actual version of core-js@3.
npm WARN deprecated chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependencies.
npm WARN deprecated bfj-node4@5.3.1: Switch to the `bfj` package for fixes and new features!
npm WARN deprecated mkdirp@0.5.1: Legacy versions of mkdirp are no longer supported. Please update to mkdirp 1.x. (Note that the API surface has changed to use Promises in 1.x.)
npm WARN deprecated json3@3.3.2: Please use the native JSON object instead of JSON 3
npm WARN deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade to fsevents 2.
npm WARN deprecated browserslist@1.7.7: Browserslist 2 could fail on reading Browserslist >3.0 config used in other tools.
npm WARN deprecated circular-json@0.3.3: CircularJSON is in maintenance only, flattened is its successor.
npm WARN deprecated socks@1.1.10: If using 2.x branch, please upgrade to at least 2.1.6 to avoid a serious bug with socket data flow and an import issue introduced in 2.1.0
npm WARN deprecated left-pad@1.3.0: use String.prototype.padStart()
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
[REDACTED] \ refresh-package-json:webpack-merge: timing action:finalize Completed in
  
```

图 2.20 通过“vue init”命令创建应用项目（二）

(6) 整个安装过程相对于 Vue 3 版本所需要的时间要长不少，需要耐心多等待一会儿。等到安装过程全部完成后，命令行控制台会提示如图 2.21 所示的信息。

如图 2.21 中的箭头所示，命令行控制台中的提示信息告诉我们，进入项目目录后通过输入“npm run dev”命令就可以启动运行项目了。

```

Node.js command prompt
added 1803 packages from 1110 contributors and audited 1811 packages in 594.168s
29 packages are looking for funding
  run 'npm fund' for details
found 99 vulnerabilities (76 low, 9 moderate, 13 high, 1 critical)
  run 'npm audit fix' to fix them, or 'npm audit' for details

Running eslint --fix to comply with chosen preset rules...
# =====

> initvue@1.0.0 lint E:\vueprojects\initvue
> eslint --ext .js,.vue src test/unit test/e2e/specs "--fix"

# Project initialization finished!
# =====

To get started:
  cd initvue
  npm run dev

Documentation can be found at https://vuejs-templates.github.io/webpack
  
```

图 2.21 通过“vue init”命令创建应用项目（三）

(7) 接下来，我们就按照上面的提示信息测试一下，效果如图 2.22 所示。

```

npm
  
```

DONE Compiled successfully in 14503ms  
12:26:11 | 10: PM |  
Your application is running here: http://localhost:8080

图 2.22 通过“npm run dev”命令运行应用项目

(8) 如图 2.22 中的箭头所示，命令行控制台中的提示信息告知我们，通过在浏览器中输入“<http://localhost:8080/>”地址就可以运行项目了，效果如图 2.23 所示。

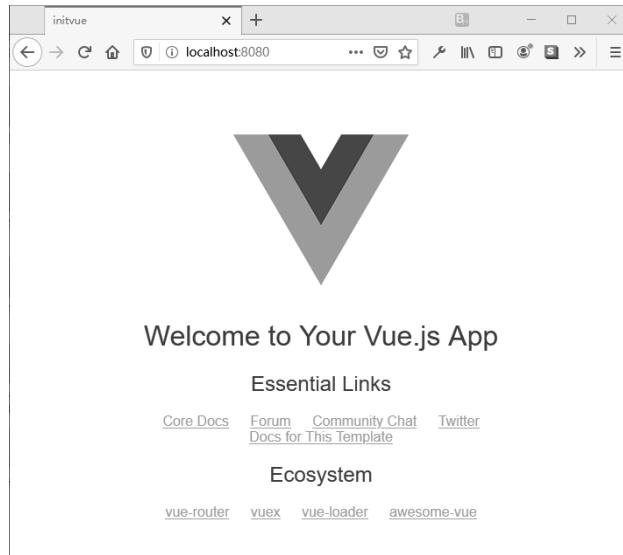


图 2.23 测试运行 Vue.js 应用项目

(9) 最后，还是通过输入“`npm run build`”命令对项目进行打包发布操作，效果如图 2.24 所示。

```

Node.js command prompt
E:\vueprojects\initvue>npm run build
> initvue@1.0.0 build E:\vueprojects\initvue
> node build/build.js

Hash: 0905bd3775288130eebe
Version: webpack 3.12.0
Time: 31407ms

Asset           Size  Chunks     Chunk Names
static/js/app.2f2e5edd9af2c59aa514.js    11.5 kB   0 [emitted]  app
static/js/vendor.ec0485e92c06cccd9446e.js  94.8 kB   1 [emitted]  vendor
static/js/manifest.2ae2e69a05c33dfc65f8.js 857 bytes  2 [emitted]  manifest
static/css/app.30790115300ab27614ce176899523b62.css 432 bytes  0 [emitted]  app
static/css/app.30790115300ab27614ce176899523b62.css.map 797 bytes  0 [emitted]
static/js/app.2f2e5edd9af2c59aa514.js.map 21.7 kB   0 [emitted]  app
static/js/vendor.ec0485e92c06cccd9446e.js.map 494 kB   1 [emitted]  vendor
static/js/manifest.2ae2e69a05c33dfc65f8.js.map 4.97 kB   2 [emitted]  manifest
index.html      509 bytes  [emitted]

Build complete.

Tip: built files are meant to be served over an HTTP server.
Opening index.html over file:// won't work.

```

图 2.24 通过“`npm run build`”命令打包发布应用项目

下面我们还是通过 VS Code 开发工具查看一下 initvue 项目中的变化，如图 2.25 所示。



图 2.25 生成的 dist 打包目录

如图 2.25 中所示，dist 目录已经自动生成，该目录下包括一个 index.html 页面文件和一个 static 目录，这个 static 就是通过 Webpack 打包后生成的脚本文件。至于测试运行的方法，直接将该目录部署到服务器上即可。

### 2.3.6 通过 Visual Studio Code 开发调试 Vue 代码

目前，通过 Visual Studio Code 开发工具开发调试 Vue.js 应用代码，几乎是最流行的标准配置了。Visual Studio Code 开发工具的强大之处就不必多说了，也正是因为 Visual Studio Code 开发工具强大的扩展能力，设计人员才在该工具平台下设计出了许多基于 Vue.js 应用开发的优秀插件。接下来，我们就借助前面编写完成的 initvue 项目应用，详细介绍一下通过 Visual Studio Code 开发调试 Vue 代码的基本过程。

(1) 在 Visual Studio Code 开发工具中安装一款名称为 `vetur` 的插件，该插件实现了 vue 代码基本语法的高亮功能，如图 2.26 所示。

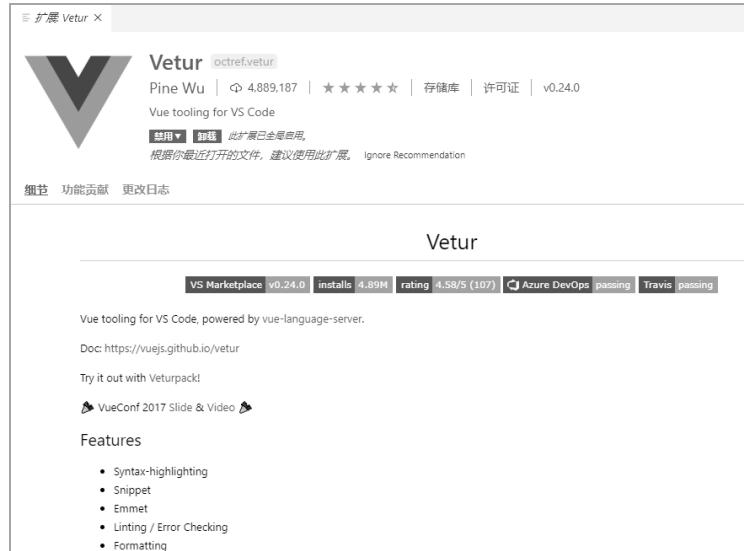


图 2.26 在 VS Code 工具中安装 vetur 插件

(2)为 VS Code 开发工具安装调试时所使用的浏览器插件。一般地,浏览器会选择 Chrome 和 Firefox 这两款,相关的安装地址如下:

- Debugger for Chrome

<https://marketplace.visualstudio.com/items?itemName=msjsdiag.debugger-for-chrome>  
在 VS Code 开发工具中,安装 Debugger for Chrome 插件后的效果如图 2.27 所示。

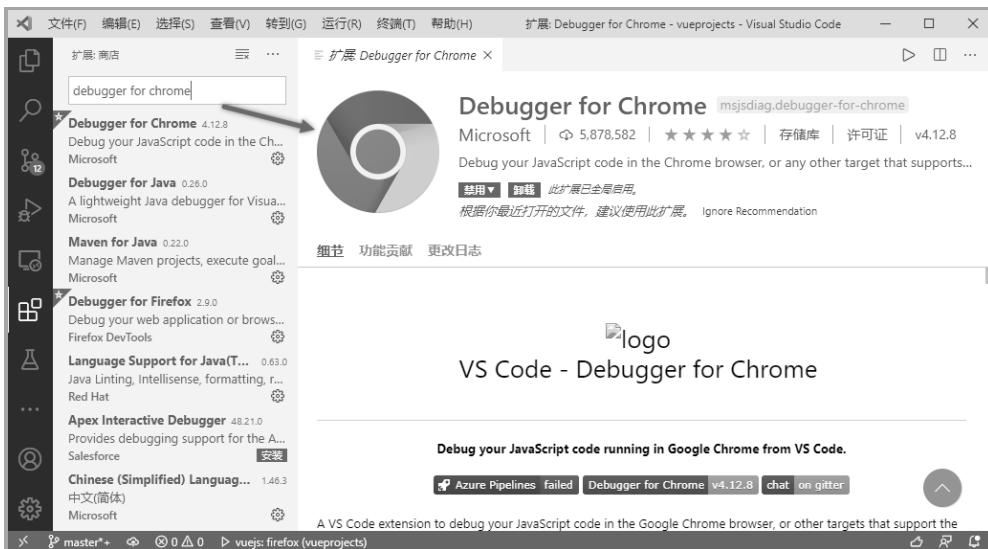


图 2.27 在 VS Code 工具中安装“Debugger for Chrome”插件

- Debugger for Firefox

<https://marketplace.visualstudio.com/items?itemName=hbenl.vscode-firefox-debug>  
在 VS Code 开发工具中,安装 Debugger for Firefox 插件后的效果如图 2.28 所示。

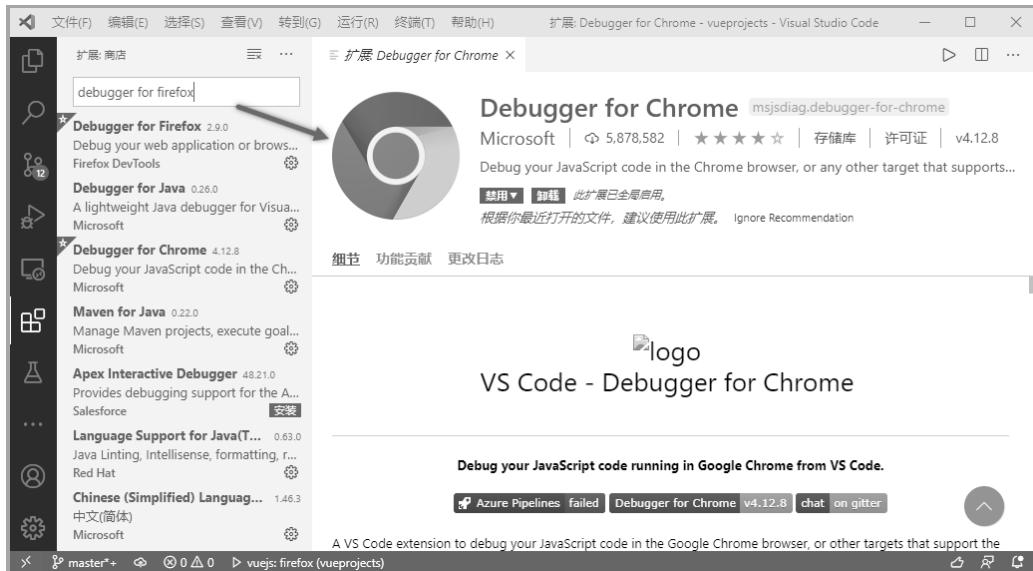


图 2.28 在 VS Code 工具中安装“Debugger for Firefox”插件

(3) 最后，通过前面编写完成的 initvue 项目应用测试一下调试过程。我们先打开“src\components”目录下的 HelloWorld.vue 代码文件，然后跳转到第 90 行代码中，并在该行代码设置断点，如图 2.29 所示。

```

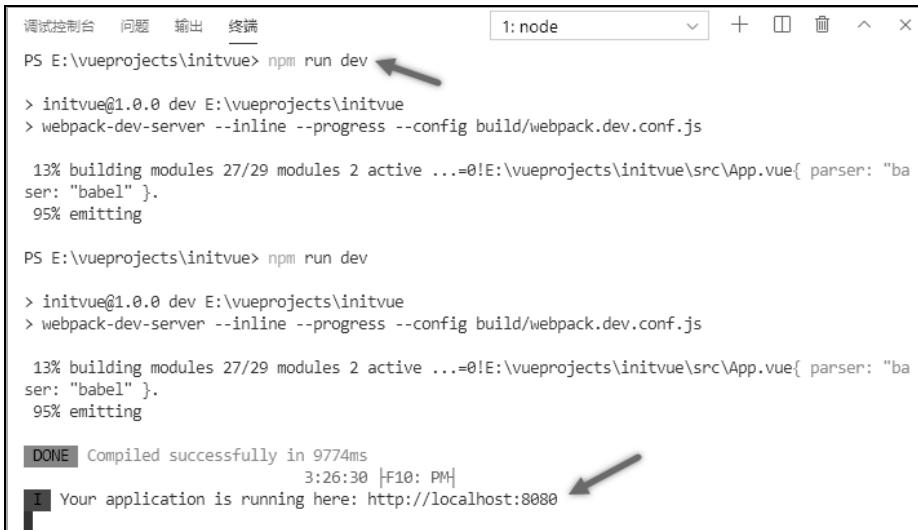
HelloWorld.vue
initvue > src > components > HelloWorld.vue > {} "HelloWorld.vue" > script > data
85
86  <script>
87  export default {
88    name: 'HelloWorld',
89    data () {
90      return {
91        msg: 'Welcome to Your Vue.js App'
92      }
93    }
94  </script>

```

图 2.29 在 Vue 代码中设置断点

(4) 如图 2.29 中所示，被设置断点的代码行的左侧会标记一个“实心圆点”。在 VS Code 开发工具中设置断点的方法很简单，按 F9 快捷键即可。

(5) 在 VS Code 开发工具自带的控制台窗口中输入“npm run dev”命令，启动该 Vue.js 应用项目（等同于命令行控制台方式），如图 2.30 所示。



```

调试控制台 问题 输出 终端 1: node + ×
PS E:\vueprojects\initvue> npm run dev
> initvue@1.0.0 dev E:\vueprojects\initvue
> webpack-dev-server --inline --progress --config build/webpack.dev.conf.js

13% building modules 27/29 modules 2 active ...=0!E:\vueprojects\initvue\src\App.vue{ parser: "ba
ser: "babel" }.
95% emitting

PS E:\vueprojects\initvue> npm run dev
> initvue@1.0.0 dev E:\vueprojects\initvue
> webpack-dev-server --inline --progress --config build/webpack.dev.conf.js

13% building modules 27/29 modules 2 active ...=0!E:\vueprojects\initvue\src\App.vue{ parser: "ba
ser: "babel" }.
95% emitting

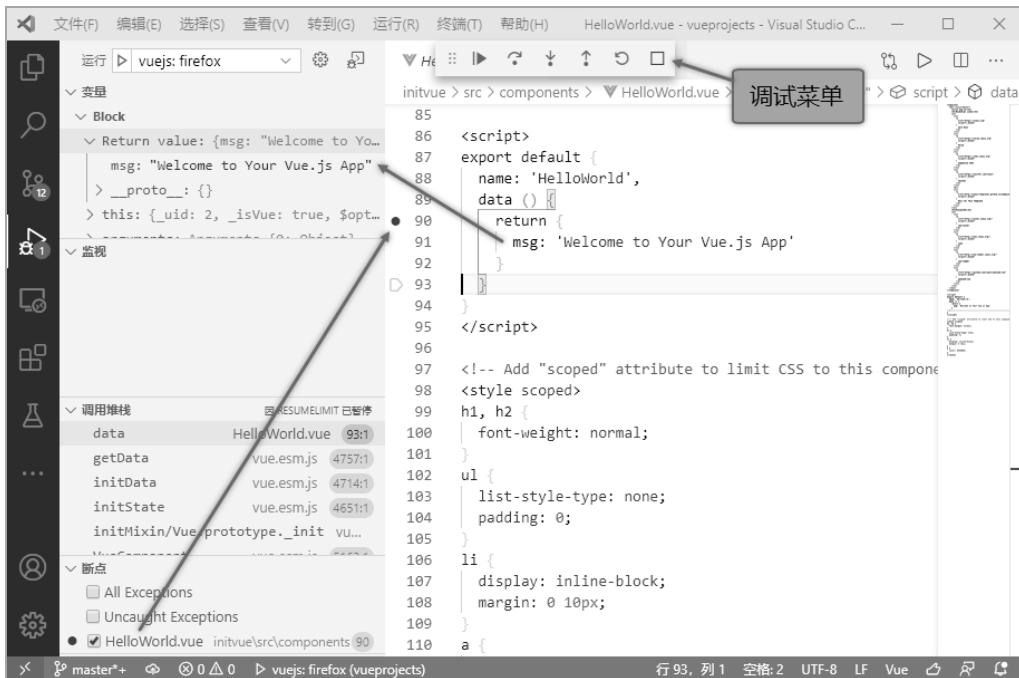
[DONE] Compiled successfully in 9774ms
3:26:30 |F10: PM|
[I] Your application is running here: http://localhost:8080

```

图 2.30 在 VS Code 开发工具自带的控制台窗口中启动项目

(6) 如图 2.30 中的箭头所示，控制台窗口中给出的提示信息与之前命令行控制台中的信息基本一致。不过，这里的方法不再是直接通过运行浏览器运行调试了，而是在 VS Code 开发工具中通过按 F5 快捷键来开启调试功能，此时 VS Code 会自动打开浏览器（Firefox）并通过访问“<http://localhost:8080>”地址运行 initvue 项目。

(7) 此时，浏览器的状态与图 2.23 完全一致，而我们要关注的是 VS Code 开发工具的状态，如图 2.31 所示。



The screenshot shows the VS Code interface with the following details:

- Top Bar:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H) HelloWorld.vue - vueprojects - Visual Studio C...
- Left Sidebar:**
  - 运行: vuejs: firefox
  - 变量: 显示当前作用域的变量，包括 `msg` 和 `\_\_proto\_\_`。
  - 调用堆栈: 显示当前调用链，包括 `HelloWorld.vue`、`getData`、`initData` 等。
  - 断点: 显示 `HelloWorld.vue` 中设置的断点。
- Center Area:**
  - 代码编辑器显示 `HelloWorld.vue` 的内容，高亮了 `msg` 变量。
  - 右侧是文件树视图，显示了项目的目录结构。
- Bottom Status Bar:** 行 93, 列 1 空格: 2 UTF-8 LF Vue
- Annotations:**
  - A callout bubble labeled "调试菜单" points to the "Debug" icon in the top bar.
  - An arrow points from the "Variables" section of the sidebar to the highlighted `msg` variable in the code editor.

图 2.31 在 VS Code 开发工具自带的控制台窗口中启动项目

(8) 如图 2.31 中的箭头和标识所示, VS Code 开发工具的界面中弹出了一个“调试菜单”工具条, 我们可以通过菜单中的“单步 (Step in)”按钮逐行运行 Vue 代码。

此外, 我们观察到源代码中的 msg 变量定义的字符串信息, 与调试 (变量) 窗口中所对应变量的数据完全一致。基于此功能, 设计人员就可以在 VS Code 开发工具中借助浏览器插件的配合, 调试 Vue.js 应用代码。

# 第 3 章

## Node.js 语法基础

Node.js 是建立在 V8 引擎之上的，也就意味着 Node.js 的语法几乎与 JavaScript 语法一致。这也是 Node.js 大受前端开发人员欢迎的原因，即通过一门语言便可打通前端的开发。在这一章中将介绍 JavaScript 的基本使用，为之后 Node.js 的学习提供必要的基础。

通过本章的学习可以：

- 掌握 JavaScript 的基础语法与使用。
- 掌握简单的 JavaScript 编程风格。
- 掌握基本的 Node.js 控制台的使用。

### 3.1 JavaScript 语法

JavaScript 是一门解释型、弱类型的脚本语言，也是 Web 开发最重要的语言之一。JavaScript 由 ECMAScript、DOM（文档对象模型）、BOM（浏览器对象模型）三部分组成。ECMAScript 规定了 JavaScript 的语法核心，这也是本节重点介绍的内容。

#### 3.1.1 变量

##### 1. 交互式运行环境——REPL

Node.js 提供了一个交互式运行环境——REPL。在这个交互式环境中可以运行简单的应用程序。在控制台直接输入 node 即可进入这个环境，此时控制台会显示一个提示符“>”，表明我们已经进入这个环境，如图 3.1 中的箭头所示。

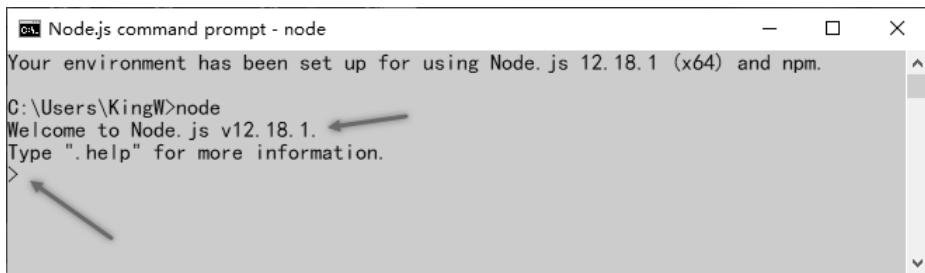


图3.1 进入REPL运行环境



如果要退出该运行环境，连续按两次 **Ctrl+C** 快捷键，或者输入**.exit**。Node.js 的命令需要在前面加点。例如，可用**.help** 查看所有命令。

本节中的所有代码都会在这个环境中使用和运行。

## 2. 浏览器环境——Firefox

当然，读者也可以在浏览器的控制台运行。以 Firefox 浏览器为例，通过按 F12 键或者 **Ctrl+Shift+I** 快捷键打开开发者工具，在开发者工具栏中选择 **Console** 面板，如图 3.2 所示。在 **Console** 中也是显示提示符“>”，和 **REPL** 的使用方法一致。



图3.2 浏览器的Console面板

## 3. 关键字 var

JavaScript 的变量通过关键字 **var** 来声明。前面说过 JavaScript 是一门弱类型的编程语言，JavaScript 的所有数据类型都可以用 **var** 关键字来声明，通过 **var 变量名=值** 的形式就可以对变量同时进行声明和赋值。和许多语言一样，JavaScript 通过分号 “;” 来分隔不同的语句，以下这段代码就声明了两个变量：

```
var a = "node.js";
var b = 10;
```

## 4. 变量的命名

JavaScript 规定变量名必须以字母、美元符 (\$)、下划线 (\_) 三者之一开头，同时 JavaScript 区分字母大小写，字母大小写不同也就意味着是不同的两个变量。同时，JavaScript 不区分单引号与双引号，因此上一个例子若把双引号换成单引号，程序表示的意思是一样的：

```
var a = 'node.js';
var b = 10;
```

## 5. 变量提升机制

JavaScript 中存在变量提升机制，也就是所有的变量声明在运行时都会提升到代码的最前方。例如，上个例子在运行时实际上会先声明两个变量再赋值：

```
var a;
var b;
a = "node.js";
b = 10;
```

通过一个更直观的例子或许会让读者更易理解变量提升。在 REPL 中试图使用一个未声明的变量时会出现 `is not defined` 错误，如图 3.3 所示。

```
$ node
> console.log(a);
ReferenceError: a is not defined
at repl:1:13
at sigintHandlersWrap (vm.js:22:35)
at sigintHandlersWrap (vm.js:96:12)
at ContextifyScript.Script.runInThisContext (vm.js:21:12)
at REPLServer.defaultEval (repl.js:313:29)
at bound (domain.js:280:14)
at REPLServer.runBound [as eval] (domain.js:293:12)
at REPLServer.<anonymous> (repl.js:513:10)
at emitOne (events.js:101:20)
at REPLServer.emit (events.js:188:7)
```

图 3.3 变量未声明错误

如果试图使用一个已经声明却未赋值的变量，那么这个变量所代表的是 `undefined`，如图 3.4 所示。

```
> var a; console.log(a);
undefined
undefined
>
```

图 3.4 已经声明却未赋值



图 3.4 中有两行 `undefined`：一个是白色，一个是半透明白色。执行 `node` 语句，在没有任何返回值的时候总是会输出一个 `undefined`，读者不必介意，这不是错误。白色语句是正常输出的内容。

使用一个在后来定义赋值的代码时会返回 `undefined`，如图 3.5 所示。

```
> console.log(a); var a = 10;
undefined
undefined
>
```

图 3.5 先使用后声明赋值

可以发现这段代码返回 `undefined` 正是因为变量提升，实际运行的代码如下：

```
var a;
console.log(a);
a = 10;
```

### 3.1.2 注释

JavaScript 中的注释和很多其他编程语言类似，以双斜杠（//）代表单行注释，以“/\*注释内容\*/”形式代表多行注释。

```
// 这是单行注释
/* 这是
   多行
   注释
*/
```

### 3.1.3 数据类型

JavaScript 中的数据类型可以分为简单数据类型和复杂数据类型。简单数据类型有 undefined、boolean、number、string、null，复杂数据类型只有 object。object 由一组无序的键值对组成。

#### 1. 利用 typeof 区分数据类型

利用操作符 typeof 可以部分区分以上数据类型。typeof 返回的值有 undefined、boolean、number、string、object 和 function。下面举一个例子。

##### 【代码 3-1】

```
01 var a;
02 var b 2;
03 var c node.js';
04 var d rue;
05 var e unction() {
06
07 }
08 var f ull;
09 var g
10     nul2
11 }
12 var ar [a,b,c,d,e,f,g];
13 for(va = 0, max = arr.length; i < max; i++) {
14     cole.log(typeof arr[i]);
15 }
16
17 // undned
18 // num
19 // str
20 // boon
21 // funon
22 // obj
23 // obj
```

**【代码说明】**

可以看到 null 和 object 都返回了 object，这是因为 null 实际上是一个空对象指针，当一个变量只声明但未赋值时就会返回 undefined。

number 和 string 数据类型分别指数字类型和字符串类型； boolean 类型和其他语言一样，仅有 true 和 false 两个值； null 仅有一个值 null。

## 2. 利用 Boolean()转化数据类型

JavaScript 中可以利用 Boolean()函数将其他数据类型转化为布尔值。需要注意的是，空字符串、0、null、undefined、NaN 都将转化为 false，其他值则会转化为 true。下面举一个例子。

**【代码 3-2】**

```

01 var a;
02 var b = null;
03 var c = 0;
04 var d = '';
05 var e = NaN;
06 var arr = [a,b,c,d,e];
07 for(var i = 0, max = arr.length; i < max; i++) {
08     console.log(Boolean(arr[i]));
09 }
10
11 // false
12 // false
13 // false
14 // false
15 // false

```

### 3.1.4 函数

在 JavaScript 中，声明一个函数只需要使用 function 关键字即可，如声明一个求和的函数，代码如下：

```

function add(num1, num2) {
    return num1 + num2;
}

```

当然，函数同样可以作为一个值传递给一个变量，例如：

```

var add = function(num1, num2) {
    return num1 + num2;
}

```

调用一个函数同样很简单，只需要在函数声明之后使用“函数名(参数)”的形式调用即可，如调用上面的函数：

```

function add(num1, num2) {
    return num1 + num2;
}

```

```

}
add(1, 2);
// 3
add(3,5);
//8

```

函数中默认带有一个 `arguments` 对象，这是一个类数组对象。`arguments` 记录了传递给函数的参数信息，因为 JavaScript 中的函数调用时，参数个数并不需要和定义函数时的个数一致。在上面的 `add()` 函数中多添加几个参数，函数仍然会正常执行，例如：

```

function add(num1, num2) {
    return num1 + num2;
}
add(1,2,4,5,5);
// 3
add(3,5,2,3);
//8

```

利用好这一点和 `arguments` 类数组特性可以对上述的 `add` 函数拓展一下，让这个函数无论接收多少个参数，总能返回这些数值的和：

```

function add() {
    var sum = 0;
    for(var i = 0, max = arguments.length; i < max; i++) {
        sum += arguments[i];
    }
    return sum;
}
add(2,3,4);
// 9
add(2,4,5);
// 11

```

还可以利用 JavaScript 中的 `arguments` 类数组对象模拟函数重载。当然，实际上 JavaScript 并不支持函数重载，比如通过检测 `arguments` 对象的 `length` 属性做出不同的反应来模拟重载。下面给出一个完整的例子。

### 【代码 3-3】

```

01 function operate() {
02     if(arguments.length == 2) {
03         return arguments[0] * arguments[1];
04     } else {
05         var sum = 0;
06         for(var i = 0, max = arguments.length; i < max; i++) {
07             sum += arguments[i];
08         }
09         return sum;
10     }
11 }
12 operate(3, 4);

```

```
13 // 12
14 operate(3, 4, 5);
15 // 12
```

以上代码的运行效果如图 3.6 所示。

```
D:\projects>node
> function operate() {
... if(arguments.length == 2) {
...     return arguments[0] * arguments[1];
... } else {
...     var sum = 0;
...     for(var i = 0, max = arguments.length; i < max; i++) {
...         sum += arguments[i];
...     }
...     return sum;
... }
...
undefined
> operate(3, 4);
12
> operate(3, 4, 5);
12
> -
```

图 3.6 运行效果

#### 【代码说明】

arguments 对象是一个类数组对象。通过数组的 slice() 函数可以把 arguments 对象转化为一个真正的数组，这样就可以使用数组的所有函数，而不用担心出现其他问题了。

```
function funName() {
    var arguments = [].slice.call(arguments);
    // the code of function
}
```

### 3.1.5 闭包

JavaScript 中的变量可以分为全局变量和局部变量。JavaScript 中的函数自然可以读取到全局变量，而函数外部并不能读取到函数内部定义的变量，例如：

#### 【代码 3-4】

```
01 var str = 'node.js';
02 function copy () {
03     var str2 = str;
04     console.log(str2);
05 }
06 copy();
07 // node.js
08 console.log(str2);
09 // str2 is not defined
```

当然，这需要在定义变量的时候使用 var 关键字定义。不用 var 关键字定义的话，实际上这个变量会成为全局对象的一个属性。在 Node.js 中，全局对象是 global，如果上面代码中的 str2 变量不使用 var 定义，str2 就会成为一个全局变量，函数外部也是可以读取到这个变量的，

例如：

**【代码3-5】**

```
01 var str = 'node.js';
02 function copy () {
03     str2 = str;
04     console.log(str2);
05 }
06 copy();
07 // node.js
08 console.log(str2);
09 // node.js
```



建议所有的变量都使用var关键字进行声明（或定义），以避免不必要的错误出现。

JavaScript中的闭包可以让函数读取到其他函数内部的变量，如下代码就可以让函数之外读取到函数内部定义的变量，这就是最简单的闭包。

**【代码3-6】**

```
01 function a() {
02     var str = 'node.js';
03     return function() {
04         var str2 = str + ' is powerful';
05         return str2;
06     }
07 }
08 a()();
09 // node.js is powerful
```

以上就是JavaScript的简要介绍。更多关于JavaScript的知识，读者可以阅读相关的书籍进行学习和掌握。进行Node.js的学习之前，读者应该对JavaScript有一定的了解。

## 3.2 命名规范与编程规范

与其他语言相比，JavaScript总是显得相对灵活，对代码的格式要求也显得相对宽松，因此对JavaScript编码制定一定的规范是非常重要的。一个良好的规范不仅能让阅读代码的人感到清晰愉悦，还能让整个项目更加容易维护。

### 3.2.1 命名规范

JavaScript作为一种弱类型的语言，命名的规范显得更加重要，因为开发人员并不能直接看出这个变量的作用。

## 1. var 关键字

在 JavaScript 中，所有的变量都应该通过 var 关键字来声明，而不是缺少 var 关键字，因为缺少 var 的变量声明会使得这个变量成为全局变量，在开发中则应尽量减少全局变量：

```
var a = 'node.js';
// 推荐
b = 12
// 不推荐
```

## 2. 驼峰命名法

在开发中，变量的命名常常是让开发人员头疼的问题，一般来说每个团队都会有自己的命名规范。近些年来更加流行的是驼峰命名法。如它的名字一样，驼峰命名法中第一个单词的开头小写，其他单词的开头字符大写，例如：

```
var myNumber;
var myString;
```

## 3. 常量

在其他语言中，会有常量这样一个概念。这是一种不允许在声明赋值之后再修改的变量。显然，JavaScript 中有着同样的需求。在开发人员不希望有些变量得到修改时，常量就显得格外重要了。例如，定义一个圆周率的常量。

在常量的命名中，开发人员往往采用变量名全部大写的方式来表示这是一个常量。当然，实际上，这样的变量依旧是可以被修改的。这需要开发人员共同遵守规定，把这样一个变量作为一个常量，而不是普通的 JavaScript 变量：

```
var PI = 3.14159
// 这是一个圆周率的常量
```

## 4. 内部变量

开发中还有一类就是内部变量。这类变量并不希望在局部作用域之外的其他作用域来存取这些变量。开发人员通常以下划线“\_”开头命名作为约定俗成的内部变量。当然，这同样需要协同的开发人员共同遵守这个约定：

```
var obj = {
    _num: 12,
// 这是一个内部变量
    put: function() {
        return this._num;
    }
}
console.log(obj.put())
```

## 5. 有意义的名字

命名规范中同样需要遵守的是，在命名中不应该使用一些无意义的变量名。这些无意义的

变量名往往会使开发人员摸不着头脑。变量的命名应该是一些有意义的、能够表示变量作用的，而不是一些无意义、混淆视听的变量名。

### 3.2.2 编程规范

在 JavaScript 中遵守编程规范，会使得整个项目得到更快的开发和更好的维护。同时，也可以让代码看起来更加优雅易读。

#### 1. 以分号结尾

在 JavaScript 代码中，所有的语句都应该以分号结尾，虽然 JavaScript 中并没有强制要求：

```
var n = 12;
// 以分号结尾，推荐
var n = 12
// 结尾缺少分号，不推荐
```

#### 2. 大括号

JavaScript 的所有语句块都应该有大括号，比如一个简单的 if 判断语句块里面只有一行时，不使用大括号并不会出现错误，但是这往往会让开发人员产生疑惑：

```
var n = 12;
if(n < 10) {
    console.log(n);
}
// 即使语句块里只有一行代码也应该有大括号
```

#### 3. ===

相等判断中应该尽量使用绝对等于 “`==`” ，因为等于 “`==`” 存在着类型转化，这在开发中可能会出现意想不到的错误。例如，使用`==`来判断 `null` 与 `undefined` 时会出现 `true` 的情况，而使用绝对等于 “`===`” 则不会：

```
console.log(1 == true);
// true
console.log(1 === true);
// false
console.log(null == undefined);
// true
console.log(null === undefined);
// false
```

#### 4. 空格的使用

关于 JavaScript 中的空格，永远不要吝啬，因为满屏连续的字符串会让人头疼。建议在数值操作符（如`+`、`-`、`*`、`/`、`%`等）前后留有一个空格，在赋值运算符和相等判断运算符的前后留有一个空格。在 json 对象中“键-值对”（Key-Value Pair）的冒号后应该留有一个空格，比

如以下的空格会让代码在开发人员的眼中更加优雅：

```
var num = 12;
// 赋值运算符前后留空格

if (num === 12) {
    // your coding
}
// 相等判断运算符的前后留出空格

var num2 = num * 2;
// 数值运算符的前后留出空格
var obj = {
    name: 'node.js'
}
// “键-值对”冒号后面留出一个空格
```



关于 JavaScript 中的注释，永远记住一条：所有的注释都应该是有意义的，无意义的注释只会让阅读代码的人员感到更加困惑。

关于 JavaScript 中的编程规范就简单介绍到这里。相比于别人介绍的规范，拥有一套属于自己团队内部的使用规范，并且让开发人员遵守这个规范则更加重要。

## 3.3 Node.js 的控制台 console

利用好 Node.js 提供的 console 控制台和 debug 可以有效地辅助开发和定位 bug。在 Node.js 中，console 代表控制台，可以通过 console 对象的各种方法向控制台进行标准输出。注意：在面向对象程序设计中，把类中定义的操作或功能称为方法（Method）。本书在没有特别说明以及不会造成混淆时也把方法称为函数，如果交替使用，则都是指一个意思。

### 3.3.1 console 对象下的各种函数

在 REPL 交互式运行环境中输入 console，可以看到 console 对象下各种函数组成的一个数组，如图 3.7 所示。

```
> console
Console {
  log: [Function: bound ],
  info: [Function: bound ],
  warn: [Function: bound ],
  error: [Function: bound ],
  dir: [Function: bound ],
  time: [Function: bound ],
  timeEnd: [Function: bound ],
  trace: [Function: bound trace],
  assert: [Function: bound ],
  Console: [Function: Console ] }
```

图 3.7 console 对象的函数

### 3.3.2 console.log()函数

console.log()函数用于标准输出流的输出，也就是在控制台中显示一行信息，例如：

```
console.log('node.js is powerful')
```

无论是在 RPEL 环境中运行这行代码，还是作为 Node.js 文件执行这行代码，都可以看到控制台输出了“node.js is powerful”字样。

console.log()方法并没有对参数的个数进行限制，当传递多个参数时，控制台输出时将以空格分隔这些参数，例如：

```
console.log('node.js', 'is', 'powerful');
```

运行之后，同样会在控制台输出“node.js is powerful”字样，这三个单词也依旧是以空格分隔开来的。

console.log()方法也可以利用占位符来定义输出的格式，如%d 表示数字、%s 表示字符串。



如果需要对后面的多个参数都定义格式，就要逐个设置，并且输出时将不会再以空格分隔；如果没有预定义格式，就将正常输出。

示例代码如下：

#### 【代码 3-7】

```
01 console.log('%s%s', 'node.js', 'is', 'powerful');
02 // node.jsis powerful
03 console.log('%s%s%s', 'node.js', 'is', 'powerful');
04 // node.jsispowerful
05 console.log('%d', 'node.js');
06 // NaN
07 console.log('%d', 'node.js', 'is', 'powerful');
08 // NaN is powerful
```

在这一段代码中，需要注意的是当使用%d 占位符后，如果对应的参数不是数字，控制台将会输出 NaN。

### 3.3.3 console.info()、console.warn()和 console.error()函数

console.info()、console.warn()以及 console.error()函数的使用方法和 console.log()函数一致，如果将 3.3.2 小节的代码换成 console.info()、console.warn()、console.error()函数，将得到同样的结果。

#### 【代码 3-8】

```
01 console.warn('%s%s', 'node.js', 'is', 'powerful');
02 // node.jsis powerful
03 console.warn('%s%s%s', 'node.js', 'is', 'powerful');
```

```

04 // node.js is powerful
05 console.info('%d', 'node.js');
06 // NaN
07 console.info('%d', 'node.js', 'is', 'powerful');
08 // NaN is powerful
09 console.error('%d', 'node.js');
10 // NaN
11 console.error('%d', 'node.js', 'is', 'powerful');
12 // NaN is powerful

```

### 3.3.4 console.dir()函数

console.dir()函数用于将一个对象的信息输出到控制台。如下代码将定义一个简单的对象。

#### 【代码 3-9】

```

01 const obj = {
02   name: 'node.js',
03   get: function() {
04     console.log('get');
05   },
06   set: function() {
07     console.log('set');
08   }
09 }
10 console.dir(obj);

```

在 RPEL 交互运行环境中运行这段代码，可以看到控制台输出了这个对象的信息，如图 3.8 所示。

```

> const obj = {
...   name: 'node.js',
...   get: function() {
...     console.log('get');
...   },
...   set: function() {
...     console.log('set');
...   }
... }
undefined
> console.dir(obj);
{ name: 'node.js', get: [Function: get], set: [Function: set] }
undefined

```

图 3.8 console.dir()函数输出对象信息

### 3.3.5 console.time()和 console.timeEnd()函数

console.time()和 console.timeEnd()函数主要用于统计一段代码运行的时间。console.time()函数置于代码起始处，console.timeEnd()方法置于代码结尾处。只需要向这两个方法传递同一个参数，就可以看到在控制台中输入了以毫秒计的代码运行时间。如下代码统计了两个循环执行后的时间以及各个循环分别使用的时间。

#### 【代码 3-10】

```
01 console.time('total time');
```

```

02 console.time('time1');
03 for(var i =0; i< 10000; i++) {
04 }
05 console.timeEnd('time1');
06 console.time('time2');
07 for(var i =0; i< 100000; i++) {
08 }
09 console.timeEnd('time2');
10 console.timeEnd('total time');

```

将这段代码保存为名为 time.js 的文件。利用 node time 命令运行这个文件，可以在控制台看到各个循环的使用时间统计，如图 3.9 所示。

```

$ node time
time1: 0.075ms
time2: 0.549ms
total time: 8.104ms

```

图 3.9 各个循环使用的时间统计

### 3.3.6 console.trace()函数

console.trace()用于输出当前位置的栈信息，可以向 console.trace()函数传递任意字符串作为标志，类似于 console.time()中的参数。在 RPEL 交互运行环境中执行以下代码：

```
console.trace('trace');
```

可以看到此处的栈信息已经在控制台中输出，如图 3.10 所示。

```

> console.trace('trace');
Trace: trace
  at repl:1:9
    at sigintHandlersWrap (vm.js:22:35)
    at sigintHandlersWrap (vm.js:96:12)
    at ContextifyScript.Script.runInThisContext (vm.js:21:12)
    at REPLServer.defaultEval (repl.js:313:29)
    at bound (domain.js:280:14)
    at REPLServer.runBound [as eval] (domain.js:293:12)
    at REPLServer.<anonymous> (repl.js:513:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
undefined

```

图 3.10 console.trace()输出栈信息

# 第 4 章

## Node.js 中的包管理

Node.js 的模块加载机制可以在开发时更好地划分程序的功能，从而更好地做到代码解耦，同时有利于进行模块化开发，保证编写出的 Node.js 代码优雅、易读。同时，Node.js 的包管理工具 NPM 可以很方便地下载使用第三方模块，简化开发工作，提高项目开发效率。本章将介绍 Node.js 的包管理工具 NPM 的使用和 Node.js 核心模块的使用。

通过本章的学习可以：

- 从 NPM 下载使用第三方模块，并了解 package.json 文件的使用。
- 了解 Node.js 的模块机制。
- 通过实例了解 Node.js 核心模块的使用。

## 4.1 NPM 介绍

NPM 是 Node.js 的包管理工具，它的重要性就像 gem 之于 Ruby 一样。简单来说，Node.js 与 NPM 的关系是密不可分的。

### 4.1.1 NPM 常用命令

NPM 默认与 Node.js 一起安装，可以在命令行中输入 npm，验证 NPM 是否安装，如图 4.1 所示。

```

Node.js command prompt
Your environment has been set up for using Node.js 12.18.1 (x64) and npm.
C:\Users\KingW>npm
Usage: npm <command>
where <command> is one of:
  access, adduser, audit, bin, bugs, c, cache, ci, cit,
  clean-install, clean-install-test, completion, config,
  create, ddp, dedupe, deprecate, dist-tag, docs, doctor,
  edit, explore, fund, get, help, help-search, hook, i, init,
  install, install-ci-test, install-test, it, link, list, ln,
  login, logout, ls, org, outdated, owner, pack, ping, prefix,
  profile, prune, publish, rb, rebuild, repo, restart, root,
  run, run-script, s, se, search, set, shrinkwrap, star,
  stars, start, stop, t, team, test, token, tst, un,
  uninstall, unpublish, unstar, up, update, v, version, view,
  whoami

  npm <command> -h  quick help on <command>
  npm -l  display full usage info
  npm help <term>  search for help on <term>
  npm help npm  involved overview

Specify configs in the ini-formatted file:
  C:\Users\KingW\.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

npm@6.14.5 C:\Users\KingW\AppData\Roaming\npm\node_modules\npm
C:\Users\KingW>

```

图 4.1 NPM 验证安装结果

### 1. npm -v、npm version

通过输入“npm -v”命令或者“npm –version”命令可以查看 NPM 的安装版本（v6.14.5），如图 4.2 所示。

```

Node.js command prompt
C:\Users\KingW>npm -v
6.14.5 ←
C:\Users\KingW>

```

图 4.2 NPM 查看版本结果

### 2. npm init

通过“npm init”命令可以生成一个 package.json 文件。这个文件是整个项目的描述文件。通过这个文件可以清楚地知道项目的包依赖关系、版本、作者等信息。每个 NPM 包都有自己的 package.json 文件，使用这个命令需要填写项目名、版本号、作者等信息，如图 4.3 所示。

```

Node.js command prompt
package name: (npminit)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords: npm
author: king
license: (ISC)
About to write to E:\vueprojects\ch04\npminit\package.json:

{
  "name": "npminit",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" & exit 1"
  },
  "keywords": [
    "npm"
  ],
  "author": "king",
  "license": "ISC"
}

Is this OK? (yes)

```

图 4.3 npm init 生成 package.json 文件

填写完毕后，可以看到在使用命令的文件夹中多了一个 package.json 文件。当然，如果读者不想填写这些内容，也可以在这条命令后添加参数 -y 或者 --yes，这样系统将会使用默认值生成 package.json 文件，例如：

```

npm init -y
//or
npm int --yes

```

### 3. npm install

通过“npm install”命令安装包，如安装 underscore 这个包（underscore 是一个强大的 JavaScript 工具库，使用这个库可以大大提高开发效率），如图 4.4 所示。

```

Node.js command prompt
E:\vueprojects\ch04\npminit>npm install underscore
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN npminit@1.0.0 No description
npm WARN npminit@1.0.0 No repository field.

+ underscore@1.10.2
added 1 package from 1 contributor and audited 1 package in 2.927s
found 0 vulnerabilities

```

图 4.4 安装 underscore 结果

命令运行完毕后，可以发现在运行命令的文件夹中多了一个名为 node-modules 的文件夹（用来存放安装包的文件夹）。打开这个文件夹就可以找到名为 underscore 的文件夹（用来存

放 underscore 包），如图 4.5 所示。

|                    |                    |       |
|--------------------|--------------------|-------|
| LICENSE            | 文件                 | 2 KB  |
| package.json       | JSON 文件            | 3 KB  |
| README.md          | MD 文件              | 2 KB  |
| underscore.js      | JS 文件              | 52 KB |
| underscore-min.js  | JS 文件              | 17 KB |
| underscore-min.map | Linker Address ... | 27 KB |

图 4.5 underscore 文件夹下的文件

在安装包的时候，同样可以在命令后添加--save 或者-S 参数，这样安装包的信息将会记录在 package.json 文件的 dependencies 字段中，如图 4.6 所示。这样可以很方便地管理包的依赖关系。

```
"dependencies": {
  "underscore": "^1.10.2"
}
```

图 4.6 使用--save 参数安装

当然如果这个包只是开发阶段需要的，可以继续添加-dev 参数。这样安装包的信息将会记录在 package.json 文件的 devDependencies 字段中，如图 4.7 所示。

```
"devDependencies": {
  "underscore": "^1.10.2"
}
```

图 4.7 使用--ave-dev 参数安装



将所有项目安装的包都记录在 package.json 文件中。当我们的 package.json 文件中有了依赖包的记录时，只需要运行“npm install”命令，系统就会自动安装所有项目需要的依赖包。

当不需要使用某个包时，可以运行“npm uninstall”命令来卸载这个包。

### 4.1.2 package.json 文件

上文提到 package.json 文件是提供包描述的文件。在 Node.js 中，一个包是一个文件夹，文件夹中的 package.json 文件以 json 格式存储该包的相关描述。一个典型的 package.json 文件内容（这是 underscore 的 package.json 文件，有删减）如下：

```
{
  "author": {
    "name": "Jeremy Ashkenas",
    "email": "jeremy@documentcloud.org"
  },
  "bugs": {
    "url": "https://github.com/jashkenas/underscore/issues"
```

```

},
"dependencies": {},
"description": "JavaScript's functional programming helper library.",
"devDependencies": {
  "docco": "*",
  "eslint": "0.6.x"
},
"directories": {},
"gitHead": "e4743ab712b8ab42ad4ccb48b155034d02394e4d",
"homepage": "http://underscorejs.org",
"keywords": [
  "util",
  "functional",
  "server"
],
"license": "MIT",
"main": "underscore.js",
"maintainers": [
  {
    "name": "jashkenas",
    "email": "jashkenas@gmail.com"
  },
  {
    "name": "jridgewell",
    "email": "justin+npm@ridgewell.name"
  }
],
"name": "underscore",
"repository": {
  "type": "git",
  "url": "git://github.com/jashkenas/underscore.git"
},
"version": "1.8.3"
}

```

以下对主要的字段进行说明：

- **Name:** 包的名字。
- **Repository:** 包存放的仓库地址。
- **Keywords:** 包的关键字，有利于别人通过搜索找到你的包。
- **License:** 遵循的协议。
- **Maintainers:** 包的维护者。
- **Author:** 包的作者。
- **Version:** 版本号，遵循版本命名规范。
- **Dependencies:** 包依赖的其他包。
- **devDependencies:** 包开发阶段所依赖的包。
- **homepage:** 包的官方主页。

当然以上仅仅是列举了最常见的字段，所有字段的说明读者可以在网站 <https://docs.npmjs.com/files/package.json> 上找到。

## 4.2 模块加载原理与加载方式

Node.js 中的模块可以分为原生模块和文件模块。在 Node.js 中可以通过 require 方法导入模块、exports 方法导出模块。



本节多用“方法”（Method）一词而非“函数”。

### 4.2.1 require 导入模块

对于原生模块（如 http），只需要使用 require('http') 导入这个模块并将其赋值给一个变量，即可使用这个模块导出的属性、方法（即函数）。

#### 【代码 4-1】

```
01 const http = require('http');
02 http.createServer(
03   // your code
04 )
```

对于文件模块，可以使用“.”前缀来指代当前路径，从而使用相对路径来加载模块。加载模块时，可以省略.js 扩展名。例如，在同级的文件夹 node 中有一个名为 myModule.js 的文件模块，可以这样导入：

```
const myModule = require('./node/myModule');
```

在 4.1 节中利用 NPM 下载了 underscore 模块，那么在 node\_modules 文件夹的同级目录可以这样加载：

```
const underscore = require('./underscore');
```

这是因为 Node.js 内部会自动查找加载 node\_modules 文件夹下的模块。

这里有必要了解一下 Node.js 尝试路径的顺序。例如，某个模块的绝对路径是 home/hello/hello.js，在该模块中导入其他模块，写法为 require("me/first")，则 Node.js 会依次尝试使用路径：

```
/home/hello/node_modules/me/first
/home/node_modules/me/first
node_modules/me/first
```

## 4.2.2 exports 导出模块

一个模块中的变量和方法只能用于这个模块，如果想要与其他模块共享一些方法、属性等，就可以用 `exports` 导出一个对象。这个对象可以包含想要与其他模块共享的方法和属性等。

假设一个模块中有两个想要与其他模块共享的方法，一个用于数组去重，一个用于计算数组之和，可以像下面这样导出：

### 【代码 4-2】

```
01 const util = {
02   noRepeat: function(arr) {
03     return arr.filter(function(ele, index) {
04       return arr.indexOf(ele) == index;
05     });
06   },
07   add: function(arr) {
08     return arr.reduce(function(ele1, ele2) {
09       return ele1 + ele2;
10     });
11   }
12 };
13 module.exports = util;
```

假设将这个模块保存为 `exports.js`，同级目录下通过 `require` 使用该模块，代码如下：

### 【代码 4-3】

```
01 const arrFn = require('./exports');
02 const arr = [1, 2, 3, 3, 2];
03 let noRepeatArr = arrFn.noRepeat(arr);
04 let num = arrFn.add(arr);
05 console.log(noRepeatArr);
06 console.log(num);
```

运行这段代码后，可以在控制台看到输出数组`[1, 2, 3]`和数字 11，说明模块导入成功，如图 4.8 所示。

The screenshot shows a terminal window titled "Nodejs command prompt". It displays two lines of output: "[1, 2, 3]" on the first line and "11" on the second line, indicating the execution of the imported module's methods.

图 4.8 导入模块

## 4.3 Node.js 核心模块

Node.js 的核心模块主要有 http、fs、url、querystring 模块。下面分别对这几个模块进行分析。fs 模块将在第 5 章详细介绍。http 模块在第 2 章的例子中使用过，本节将详细分析其方法和原理。

### 4.3.1 http 模块——创建 HTTP 服务器、客户端

使用 http 模块只需要在文件中通过 `require('http')` 引入即可。http 模块是 Node.js 原生模块中最为亮眼的模块。传统的 HTPP 服务器会由 Apache、Nginx、IIS 之类的软件来担任，但是 Node.js 并不需要。Node.js 的 http 模块本身就可以构建服务器，而且性能非常可靠。

#### 1. Node.js 服务器端

下面创建一个简单的 Node.js 服务器。

##### 【代码 4-4】

```
01 const http = require('http');
02 const server = http.createServer(function(req, res) {
03   res.writeHead(200, {
04     'content-type': 'text/plain'
05   });
06   res.end('Hello, Node.js!');
07 });
08 server.listen(3000, function() {
09   console.log('listening port 3000');
10 });
```

##### 【代码说明】

运行这段代码，在浏览器中打开 `http://localhost:3000/` 或者 `http://127.0.0.1:3000/`，页面中显示“Hello, Node.js!”文字。

`http.createServer()` 方法返回的是 http 模块封装的一个基于事件的 http 服务器。同样，`http.request` 是其封装的一个 http 客户端工具，可以用来向 http 服务器发起请求。上面的 `req` 和 `res` 分别是 `http.IncomingMessage` 和 `http.ServerResponse` 的实例。

http.Server 的事件主要有：

- `request`: 最常用的事件，当客户端请求到来时，该事件被触发，提供 `req` 和 `res` 两个参数，表示请求和响应信息。
- `connection`: 当 TCP 连接建立时，该事件被触发，提供一个 `socket` 参数，是 `net.Socket` 的实例。

- close: 当服务器关闭时，触发事件（注意不是在用户断开连接时）。

http.createServer()方法其实就是一个 request 事件监听，利用下面的代码同样可以实现【代码 4-4】的效果。

#### 【代码 4-5】

```
01 const http = require('http');
02 const server = new http.Server();
03 server.on('request', function(req, res) {
04     res.writeHead(200, {
05         'content-type': 'text/plain'
06     });
07     res.end('Hello, Node.js!');
08 });
09 server.listen(3000, function() {
10     console.log('listening port 3000');
11 });
```

http.IncomingMessage 是 HTTP 请求的信息，提供了以下 3 个事件：

- data: 当请求体数据到来时该事件被触发。该事件提供一个 chunk 参数，表示接受的数据。
- end: 当请求体数据传输完毕时该事件被触发，此后不会再有数据。
- close: 用户当前请求结束时，该事件被触发。

http.IncomingMessage 提供的主要属性有：

- method: HTTP 请求的方法，如 GET。
- headers: HTTP 请求头。
- url: 请求路径。
- httpVersion: HTTP 协议的版本。

将上面提到的知识融合到【代码 4-4】的服务器代码中。

#### 【代码 4-6】

```
01 const http = require('http');
02 const server = http.createServer(function(req, res) {
03     let data = '';
04     req.on('data', function(chunk) {
05         data += chunk;
06     });
07     req.on('end', function() {
08         let method = req.method;
09         let url = req.url;
10         let headers = JSON.stringify(req.headers);
11         let httpVersion = req.httpVersion;
12         res.writeHead(200, {
13             'content-type': 'text/html'
```

```

14      });
15      let dataHtml = '<p>data:' + data + '</p>';
16      let methodHtml = '<p>method:' + method + '</p>';
17      let urlHtml = '<p>url:' + url + '</p>';
18      let headersHtml = '<p>headers:' + headers + '</p>';
19      let httpVersionHtml = '<p>httpVersion:' + httpVersion + '</p>';
20      let resData = dataHtml + methodHtml + urlHtml + headersHtml + httpVersionHtml;
21      res.end(resData);
22  });
23 });
24 server.listen(3000, function() {
25   console.log('listening port 3000');
26 });

```

打开浏览器输入地址后，可以在浏览器页面中看到如图 4.9 所示的信息。

```

data:
method:GET
url:/
headers:{"host":"localhost:3000","connection":"keep-alive","cache-control":"max-
age=0","upgrade-insecure-requests":"1","user-agent":"Mozilla/5.0 (Windows NT 10.0;
WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.87
Safari/537.36","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
encoding":"gzip, deflate, sdch, br","accept-language":"zh-CN,zh;q=0.8"}
httpVersion:1.1

```

图 4.9 浏览器效果

`http.ServerResponse` 是返回给客户端的信息，其常用的方法为：

- `res.writeHead(statusCode,[headers])`: 向请求的客户端发送响应头。
- `res.write(data,[encoding])`: 向请求发送内容。
- `res.end([data],[encoding])`: 结束请求。

这些方法在上面的代码中已经演示过了，这里就不再演示了。

## 2. 客户端向 http 服务器发起请求

以上方法都是 `http` 模块在服务器端的调用，接下来看客户端的调用方式。向 `http` 服务器发起请求的方法有：

- `http.request(option[,callback])`: `option` 为 `json` 对象，主要字段有 `host`、`port`(默认为 80)、`method`(默认为 GET)、`path`(请求的相对于根的路径，默认为“/”)、`headers` 等。该方法返回一个 `httpClientRequest` 实例。
- `http.get(option[,callback])`: `http.request()` 调用 HTTP 请求方式 GET 方法。

同时运行【代码 4-4】【代码 4-7】，我们可以发现命令行中输出“Hello, Node.js!”字样，表明一个简单的 GET 请求发送成功了。

**【代码 4-7】**

```

01 const http = require('http');
02 let reqData = '';
03 http.request({
04     'host': '127.0.0.1',
05     'port': '3000',
06     'method': 'get'
07 }, function(res) {
08     res.on('data', function(chunk) {
09         reqData += chunk;
10     });
11     res.on('end', function() {
12         console.log(reqData);
13     });
14 }).end();

```

调用 `http.get()`方法也可以实现同样的效果。

**【代码 4-8】**

```

01 const http = require('http');
02 let reqData = '';
03 http.get({
04     'host': '127.0.0.1',
05     'port': '3000'
06 }, function(res) {
07     res.on('data', function(chunk) {
08         reqData += chunk;
09     });
10     res.on('end', function() {
11         console.log(reqData);
12     });
13 }).end();

```

与服务端一样，`http.request()`和`http.get()`方法返回的是一个`http.ClientRequest()`实例。  
`http.ClientRequest()`类主要的事件和方法有：

- `response`: 当接收到响应时触发。
- `request.write(chunk[,encoding][,callback])`: 发送请求数据。
- `res.end([data][,encoding][,callback])`: 发送请求完毕，应该始终指定这个方法。

同样可以改写上述代码为【代码 4-9】。

**【代码 4-9】**

```

01 const http = require('http');
02 let reqData = '';
03 let option= {
04     'host': '127.0.0.1',
05     'port': '3000'

```

```

06  };
07  const req = http.request(option);
08  req.on('response', function(res) {
09      res.on('data', function(chunk) {
10          reqData += chunk;
11      });
12      res.on('end', function() {
13          console.log(reqData);
14      });
15  });

```

### 4.3.2 url 模块——url 地址处理

使用 url 模块，只需要在文件中通过 require('url') 引入即可。url 模块是一个分析、解析 url 的模块，主要提供以下三种方法：

- url.parse(urlStr[,parseQueryString][,slashesDenoteHost]): 解析一个 url 地址，返回一个 url 对象。
- url.format(urlObj): 接受一个 url 对象为参数，返回一个完整的 url 地址。
- url.resolve(from, to): 接受一个 base url 对象和一个 href url 对象，像浏览器那样解析，返回一个完整地址。

#### 【代码 4-10】

```

01  const url = require('url');
02  let parseUrl = 'https://www.google.com/?q=node.js';
03  let urlObj = url.parse(parseUrl);
04  console.log(urlObj);

```

在控制台中输出如图 4.10 所示的信息，说明解析成功。

```

Url {
  protocol: 'https:',
  slashes: true,
  auth: null,
  host: 'www.google.com',
  port: null,
  hostname: 'www.google.com',
  hash: null,
  search: '?q=node.js',
  query: 'q=node.js',
  pathname: '/',
  path: '/?q=node.js',
  href: 'https://www.google.com/?q=node.js' }

```

图 4.10 解析 url

调用 url.format() 方法返回上述完整地址的代码如下：

#### 【代码 4-11】

```

01  const url = require('url');
02  let urlObj = {
03      'host': 'www.google.com',
04      'port': 80,
05      'protocol': 'https',

```

```

06     'search': '?q=node.js',
07     'query': 'q=node.js',
08     'path': '/'
09   };
10 let urlAdress = url.format(urlObj);
11 console.log(urlAdress);

```

运行代码后，可以在控制台看到完整的 url 地址。

resolve 的调用方式如下：

#### 【代码 4-12】

```

01 const url = require('url');
02 let urlAdress = url.resolve('https://www.google.cn', '/image');
03 console.log(urlAdress);

```

运行代码可以在控制台看到完整的 url 地址 https://www.google.cn/image。

### 4.3.3 querystring 模块——查询字符串处理

使用 querystring 模块，只需要在文件中通过 require('querystring') 引入即可。querystring 模块是一个处理查询字符串的模块。这个模块的主要方法有：

- querystring.parse(): 将查询字符串反序列化为一个对象，类似 JSON.parse()。
- querystring.stringify(): 将一个对象序列化为一个字符串，类似 JSON.stringify()。

下面演示调用它们的方式。

将查询字符串反序列化为一个对象。

#### 【代码 4-13】

```

01 const querystring = require('querystring');
02 let str = 'keyWord=node.js&name=huruji';
03 let obj = querystring.parse(str);
04 console.log(obj);

```

将对象序列化为一个查询字符串。

#### 【代码 4-14】

```

01 const querystring = require('querystring');
02 let obj = {
03   keyWord: 'node.js',
04   name: 'huruji'
05 };
06 let str = querystring.stringify(obj);
07 console.log(str);

```

## 4.4 Node.js 常用模块

除了上述提到的核心模块外，Node.js 还有一些常用的模块。

### 4.4.1 util 模块——实用工具

util 模块是一个工具模块，提供的主要方法有：

- `util.inspect()`: 返回一个对象反序列化形成的字符串。
- `util.format()`: 返回一个使用占位符格式化的字符串，类似于 C 语言的 `printf`。可以使用的占位符有`%s`、`%d`、`%j`。
- `util.log()`: 在控制台输出，类似于 `console.log()`，但这个方法带有时间戳。

下面通过代码说明这些方法的调用方式。

【代码 4-15】

```
01 const util = require('util');
02 let obj = {
03   keyWord: 'node.js',
04   name: 'huruji'
05 };
06 let str = util.inspect(obj);
07 console.log(str);
```

上面这段代码已经将对象反序列化为一个字符串。之所以说这个方法在调试的时候非常有用，是因为我们还可以让控制台输出字符串带有颜色和风格，这有利于区分各种数据类型。Node.js 默认的风格可参考表 4.1 所示。

表 4.1 Node.js 默认的风格

| 数据类型      | 风格 | 数据类型 | 风格  |
|-----------|----|------|-----|
| 数字        | 黄色 | 字符串  | 绿色  |
| 布尔值       | 黄色 | 日期   | 洋红色 |
| 正则表达式     | 红色 | Null | 粗体  |
| Undefined | 斜体 |      |     |

只需要在这个方法添加一个 `json` 对象参数、将 `color` 字段设置为 `true` 即可。

【代码 4-16】

```
01 const util = require('util');
02 let obj = {
03   keyWord: 'node.js',
```

```

04     name: 'huruji'
05   };
06 let str = util.inspect(obj, {
07   'color': true
08 });
09 console.log(str);

```

如果 `util.format` 方法中的参数少于占位符，那么多余的占位符不会被替换；如果参数多于占位符，那么剩余的参数将通过 `util.spect()` 方法转换为字符串；如果没有占位符，就将以空格分隔各个参数并拼接成字符串。

#### 【代码 4-17】

```

01 const util = require('util');
02 util.format('%s is %d', 'huruji', 12);
03 // huruji is 12
04 util.format('%s is a %s%s', 'huruji', 'FE');
05 // huruji is a FE%
06 util.format('%s is a ', 'huruji', 'FE');
07 // huruji is a FE
08 util.format('huruji', 'is', 'a', 'FE');
09 // huruji is a FE

```

除了这些方法外，`util` 模块还提供了一些判断数据类型的方法（即函数），如 `util.isArray()`、`util.isRegExp()`、`util.isDate()` 等。

### 4.4.2 path 模块——路径处理

`path` 模块提供了一系列处理文件路径的工具，主要的方法有：

- `path.join()`：将所有的参数连接起来，返回一个路径。
- `path.extname()`：返回路径参数的扩展名，无扩展名时返回空字符串。
- `path.parse()`：将路径解析为一个路径对象。
- `path.format()`：接受一个路径对象为参数，返回一个完整的路径地址。

下面用代码说明这些方法的调用方式。

#### 【代码 4-18】

```

01 const path = require('path');
02 let outputPath = path.join(__dirname, 'node', 'node.js');
03 console.log(outputPath);

```

假设上面这段代码文件存放的文件夹为 C 盘目录下的 `frontEnd` 文件夹下，即 `__dirname` 表示 `C:\frontEnd`，则返回：

```
C:\frontEnd\node\node.js
```

调用 `path.extname()` 方法解析上面代码返回路径中的扩展名 `.js`，代码如下：

**【代码 4-19】**

```
01 const path = require('path');
02 let ext = path.extname(path.join(__dirname, 'node', 'node.js'));
03 console.log(ext);
```

在 Node.js 中，一个文件对象有 root、dir、base、ext、name 五个字段，分别对应根目录（一般是磁盘名）、完整目录、路径最后一个部分（可能是文件名或文件夹名，是文件名时带扩展名）、扩展名、文件名（不带扩展名），可以利用以下代码将上面的地址解析成一个路径对象。

**【代码 4-20】**

```
01 const path = require('path');
02 const str = 'C:/frontEnd/node/node.js';
03 let obj = path.parse(str);
04 console.log(obj);
```

我们将在控制台看到如图 4.11 所示的输出。

```
{ root: 'C:/',
  dir: 'C:/frontEnd/node',
  base: 'node.js',
  ext: '.js',
  name: 'node' }
```

图 4.11 解析路径

如果我们将控制台输出的这个对象作为 path.format() 的参数使用，就将会得到上面的路径字符串。

### 4.4.3 dns 模块

dns 模块的功能是域名处理和域名解析，常用方法有：

- dns.resolve(): 将一个域名解析为一个指定类型的数组。
- dns.lookup(): 返回第一个被发现的 IPv4 或者 IPv6 的地址。
- dns.reverse(): 通过 IP 解析域名。

下面用代码说明这些方法的使用。

可以通过 dns.resolve() 方法解析一下百度的 IPv4 地址。

**【代码 4-21】**

```
01 const dns = require('dns');
02 let domain = 'baidu.com';
03 dns.resolve(domain, function(err, address) {
04   if(err) {
05     console.log(err);
06     return;
07   }
08   console.log(address);
09 })
```

运行代码后，可以在控制台中看到如图 4.12 所示的数组输出。

```
[ '180.149.132.47',
  '111.13.101.208',
  '123.125.114.144',
  '220.181.57.217' ]
```

图 4.12 解析 DNS

调用 dns.lookup()方法会返回上面这个数组中的一个元素。

#### 【代码 4-22】

```
01 const dns = require('dns');
02 let domain = 'baidu.com';
03 dns.lookup(domain, function(err, address) {
04   if(err) {
05     console.log(err);
06     return;
07   }
08   console.log(address);
09 })
```

笔者控制台输出的便是上面数组中的第二个元素，即 111.13.101.208 这个地址。读者可以自行查看一下自己网络返回的地址。

我们将一个 IP 地址 114.114.114.114 传递给 dns.reverse()方法，就会得到一个域名数组，不过这个数组中只有 public1.114dns.com 这个域名。

#### 【代码 4-23】

```
01 const dns = require('dns');
02 dns.reverse('114.114.114.114', function(err, domain) {
03   console.log(domain);
04 })
```

# 第 5 章

# 文件系统

文件的操作在 Node.js 的编程中非常重要。Node.js 可以跨平台运行，所以在处理文件的操作时需要考虑不同操作系统的区别。Node.js 同时提供大量核心的 API 和外部模块供开发人员轻松地进行文件的读写和其他操作，并以高效率著称。本章将深入讨论如何在 Node.js 中使用这些模块来完成文件操作。

通过本章的学习可以：

- 处理文件的路径和从文件路径中获取信息。
- 打开一个已存在的文件和在文件操作结束后正常关闭文件。
- 使用 Node.js 包读取和写入纯文本文件、XML 文件、CSV 文件和 JSON 文件。
- 从文本文件中读取数据，并修改数据格式重新生成对应的 CSV 文件。



本章内容不包含对数据库的操作。

## 5.1 Node.js 文件系统介绍

Node.js 为文件操作提供了大量的 API。这些 API 基本上和 UNIX (POSIX) 中的 API 相对应。Node.js 在操作文件时使用的是 fs (File System) 模块。文件系统模块均有两种不同的方法，分别是异步和同步版本。

### 5.1.1 同步和异步

为了使用 Node.js 进行文件操作，首先要调用 `require('fs')` 来加载文件系统模块。异步方法的最后一个参数总是一个完整的回调函数（callback 函数）。传递给回调函数的参数一般取决于这个方法本身，但是第一个参数永远是异常（err）。如果方法执行成功，第一个参数将会是 `null` 或者 `undefined`。当使用同步方法来执行时，任何异常都会立刻引发。我们可以使用 `try`

或者 catch 来处理异常并将错误信息显示出来。

下面给出一个异步方法的例子，其中 tmp 文件夹下有一个 hello 文件。

### 【代码 5-1】

```
01 const fs = require('fs');
02 //异步操作读取文件
03 fs.unlink('./tmp/hello', (err) => {
04     if (err) throw err;
05     console.log('successfully deleted ./tmp/hello');
06 });

```

### 【代码说明】

这段代码将删除在 tmp 目录下的 hello 文件，如果删除成功就在 console 中打印删除成功的信息。也可以使用同步的方法实现同样的功能。在下面的代码中，将使用同步的方法执行相同的操作：

### 【代码 5-2】

```
01 const fs = require('fs');
02 //同步操作读取文件
03 fs.unlinkSync('./tmp/hello');
04 console.log('successfully deleted /tmp/hello');
```

### 【代码说明】

异步操作的方法不能保证执行一定成功，所以文件操作的顺序在代码执行过程中非常重要。例如下面的代码将会引发一个错误。

### 【代码 5-3】

```
01 //重命名 hello 文件为 world 文件
02 fs.rename('./tmp/hello', './tmp/world', (err) => {
03     if (err) throw err;
04     console.log('renamed complete');
05 });
06 //获取 world 文件的信息
07 fs.stat('./tmp/world', (err, stats) => {
08     if (err) throw err;
09     console.log(`stats: ${JSON.stringify(stats)}`);
10 });
```

### 【代码说明】

fs.stat 将在 fs.rename 之前执行，正确的方法是使用回调函数来执行。

下面的代码是正确使用回调函数来处理程序执行过程中的异常：

### 【代码 5-4】

```
01 fs.rename('./tmp/hello', './tmp/world', (err) => {
02     if (err) throw err;
03     fs.stat('./tmp/world', (err, stats) => {
```

```

04         if (err) throw err;
05         console.log(`stats: ${JSON.stringify(stats)} `);
06     });
07 });

```



在一个大型的系统中，建议使用异步方法，同步方法将会导致进程被锁死。和同步方法相比，异步方法性能更高，速度更快，而且阻塞更少。本书以介绍异步方法为主、同步方法为辅。

### 5.1.2 fs 模块中的类和文件的基本信息

Node.js 在文件模块中只有 4 个类，分别为 `fs.FSWatcher`、`fs.ReadStream`、`fs.Stats` 和 `fs.WriteStream`。其中，`fs.ReadStream` 和 `fs.WriteStream` 分别是读取流和写入流，我们将在后面的内容中进行介绍；`fs.FSWatcher` 和 `fs.Stats` 可以获取文件的相关信息。

关于 `stats` 类中的方法有：

- `stats.isFile()`: 如果是标准文件就返回 `true`，如果是目录、套接字、符号连接或设备等就返回 `false`。
- `stats.isDirectory()`: 如果是目录就返回 `true`。
- `stats.isBlockDevice()`: 如果是块设备就返回 `true`。大多数情况下，类 UNIX 系统的块设备都位于`/dev` 目录下。
- `stats.isCharacterDevice()`: 如果是字符设备就返回 `true`。
- `stats.isSymbolicLink()`: 如果是符号连接就返回 `true`。`fs.lstat()`方法返回的 `stats` 对象才有此方法。
- `stats.isFIFO()`: 如果是 FIFO 就返回 `true`。FIFO 是 UNIX 中一种特殊类型的命令管道。
- `stats.isSocket()`: 如果是 UNIX 套接字就返回 `true`。

使用 `fs.stat()`、`fs.lstat()` 和 `fs.fstat()` 方法都将返回文件的一些特征信息，如文件的大小、创建时间或者权限。一个典型的查询文件元信息的代码如下：

#### 【代码 5-5】

```

01 var fs = require('fs');
02 fs.stat('/Users/liuht/code/itbilu/demo/path.js', function (err, stats) {
03     console.log(stats);
04 })

```

运行上面的代码，将会输出如图 5.1 所示的文件信息。

```

1  {
2    dev: 2114,
3    ino: 48064969,
4    mode: 33188,
5    nlink: 1,
6    uid: 85,
7    gid: 100,
8    rdev: 0,
9    size: 527,
10   blksize: 4096,
11   blocks: 8,
12   atime: Mon, 10 Oct 2019 23:24:11 GMT,
13   mtime: Mon, 10 Oct 2019 23:24:11 GMT,
14   ctime: Mon, 10 Oct 2019 23:24:11 GMT,
15   birthtime: Mon, 10 Oct 2019 23:24:11 GMT
16 }

```

图 5.1 文件信息查询结果

### 5.1.3 文件路径

在 Node.js 中访问文件，既可以使用相对路径又可以使用绝对路径。我们可以使用 path 模块功能来修改链接、解析路径，还可以将路径进行转换和规范化。需要注意的是，在不同操作系统的路径分隔也不一样，如有的需要带“/”，有的不需要。因此，处理文件路径会比较困难，使用 path 模块可以很好地解决这些问题。

在使用 path 模块时，首先要调用 require('path')方法进行引用。关于 path 模块主要有以下几个主要功能：

- 规范化路径。
- 连接路径。
- 路径解析。
- 查找路径之间的关系。
- 提取路径中的部分内容。

下面的代码调用 normalize()方法来规范化路径字符串，在存储和使用路径之前，将其规范化可以避免之后的错误引用。

#### 【代码 5-6】

```

01 var path = require('path');
02 path.normalize('/foo/bar//baz/asdf/quux/..');
03 // 处理后
04 '/foo/bar/baz/asdf'

```

join()方法可以连接任意多个路径字符串。要连接的多个路径可作为参数传入。下面给出一个路径连接的示例。

#### 【代码 5-7】

```

01 var path = require('path');
02 // 合法的字符串连接
03 path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')

```

```

04 //连接后
05 '/foo/bar/baz/asdf'
06 //不合法的字符串将抛出异常
07 path.join('foo', {}, 'bar')
08 //抛出的异常
09 TypeError: Arguments to path.join must be strings'

```

### 【代码说明】

`path.relative()`方法可以找出一个绝对路径到另一个绝对路径的相对关系。例如，下面的代码判定两个路径的相对关系，结果将输出为`'../../zszzgc/lib'`。

```

01 var path = require('path');
02 path.relative('/Users/code/itbilu/demo', '/Users/code/zszzgc/lib');

```



在使用相对路径的时候，路径的相对性应该与 `process.cwd()`一致。

## 5.2 基本文件操作

Node.js 使用流（Stream）的方式来处理文件。这种处理方式和处理网络数据几乎是一样的，操作起来非常方便。使用流的方式操作一般会有一个问题，即无法在文件的指定位置进行读写。但是 Node.js 进行了更底层的操作，除了可以在文件的尾部写入，也可以在文件的特定位置写入数据。

Node.js 中有丰富的 API 支持对文件的各种操作，包括获取文件信息、创建和删除文件、打开和关闭文件、读写数据。在本节中将会介绍文件的一些基本操作，下一节会针对具体格式的文件操作进行讲解。

### 5.2.1 打开文件

在处理文件之前都需要调用 Node.js 中的 `fs.open` 方法来打开文件，然后才能使用文件描述符调用所提供的回调函数。在异步模式下打开文件的语法如下：

```
fs.open(path, flags[, mode], callback)
```

参数使用说明如下：

- `path`: 文件的路径。
- `flags`: 文件打开的方式，具体说明可参见表 5.1。
- `mode`: 设置文件模式（权限），文件创建默认权限为可读写。
- `callback`: 回调函数，同时带有两个参数。

表 5.1 fs.open 中 flag 参数说明

| flag 值 | 说明                        |
|--------|---------------------------|
| r      | 以读取模式打开文件，如果文件不存在就抛出异常    |
| r+     | 以读写模式打开文件，如果文件不存在就抛出异常    |
| rs     | 以同步的方式读取文件                |
| rs+    | 以同步的方式读取和写入文件             |
| w      | 以写入模式打开文件，如果文件不存在就创建      |
| wx     | 类似 'w'，但是如果文件路径存在则文件写入失败  |
| w+     | 以读写模式打开文件，如果文件不存在则创建      |
| wx+    | 类似 'w+'，但是如果文件路径存在则文件读写失败 |
| a      | 以追加模式打开文件，如果文件不存在则创建      |
| ax     | 类似 'a'，但是如果文件路径存在则文件追加失败  |

下面的代码将打开一个文件，并在打开之前和打开成功之后在 console 中显示相对应的消息。

#### 【代码 5-8】

```
01 var fs = require('fs');
02 // 打开文件
03 console.log("准备打开文件！");
04 fs.open('text.txt', 'r+', function(err, fd) {
05     if (err) {
06         return console.error(err);
07     }
08     console.log("成功打开文件");
09 });


```

### 5.2.2 关闭文件

关闭文件将调用 fs.close 和 fs.closeSync 方法。其中，fs.closeSync 为同步操作的方法。我们在这里主要介绍调用异步的 fs.close 方法。它一共有两个参数可以设定，具体语法如下：

```
fs.close(fd, callback)
```

参数使用说明如下：

- fd：通过 fs.open()方法返回的文件描述符。
- callback：回调函数，没有参数。

在实际开发过程中，如果打开了一个文件，就应该在文件操作完成之后尽快关闭该文件，为此可能需要跟踪那些已经打开的文件描述符，并在操作完成之后确保文件正确关闭。下面的代码将建立一个新的文本文件，并进行打开文件和关闭文件的操作。

#### 【代码 5-9】

```
01 var fs = require('fs');
```

```

02 console.log("准备打开文件！");
03 fs.open('input.txt', 'r+', function(err, fd) {
04     if (err) {
05         return console.error(err);
06     }
07     console.log("文件打开成功！");
08     // 关闭文件
09     fs.close(fd, function(err) {
10         if (err) {
11             console.log(err);
12         }
13         console.log("文件关闭成功");
14     });
15 });
16 });

```

### 【代码说明】

事实上，并不需要经常调用 `fs.close` 来关闭文件。除了几种特例之外，Node.js 在进程退出之后将自动关闭所有文件。原因在于，在使用 `fs.readFile`、`fs.writeFile` 或 `fs.append` 之后，它们并不返回任何 `fd`，Node.js 将在文件操作之后进行判断并自动关闭文件。例如，在执行下面的代码后并不需要调用 `fs.close` 来关闭文件。

```

01 var fs = require('fs');
02 // 在/home/text.txt 中写入字符串 abc
03 fs.writeFile("/home/text.txt", "abc");

```



在调用一些方法的时候，例如 `fs.createReadStream`，在 `option` 中含有 `autoClose` 选项。`autoClose` 选项设置为 `true` 时，才会在文件操作之后自动关闭，详细内容请参见相关方法的具体说明或 Node.js 的官方手册。

## 5.2.3 读取文件

Node.js 目前支持 `utf-8`、`ucs2`、`ascii`、`binary`、`base64`、`hex` 编码的文件，并不支持中文 `GBK` 或 `GB2312` 之类的编码，所以无法操作 `GBK` 或 `GB2312` 格式文件的中文内容。如果想读取 `GBK` 或 `GB2312` 格式的文件，需要第三方的模块支持，建议使用 `iconv` 模块或 `iconv-lite` 模块。其中，`iconv` 模块仅支持 `Linux`，不支持 `Windows`。

在 Node.js 中读取文件一般调用 `fs.read` 方法。该方法从一个特定的文件描述符（`fd`）中读取数据，语法格式如下：

```
fs.read(fd, buffer, offset, length, position, callback)
```

参数使用说明如下：

- `fd`: 通过 `fs.open()` 方法返回的文件描述符。
- `buffer`: 数据写入的缓冲区。
- `offset`: 缓冲区写入的写入偏移量。

- `length`: 要从文件中读取的字节数。
- `position`: 文件读取的起始位置, 如果 `position` 的值为 `null`, 就会从当前文件指针的位置读取。
- `callback`: 回调函数, 有 `err`、`bytesRead`、`buffer` 三个参数。其中 `err` 为错误信息, `bytesRead` 表示读取的字节数, `buffer` 为缓冲区对象。

下面的代码是一个文件读取的示例。首先, 调用 `fs.open()` 方法将文件打开; 然后, 从第 100 个字节开始, 读取后面的 1024 个字节的数据; 读取完成后, `fs.open()` 会使用回调方法返回数据, 再处理读取到的缓冲数据。

#### 【代码 5-10】

```
01 var fs = require('fs');
02 fs.open('/Users/liuht/code/itbilu/demo/fs.js', 'r', function (err, fd) {
03   var readBuffer = new Buffer(1024),
04     offset = 0,
05     len = readBuffer.length,
06     filePostion = 100;
07   fs.read(fd, readBuffer, offset, len, filePostion, function(err, readByte) {
08     console.log('读取数据总数: '+readByte+' bytes' );
09     // ==>读取数据总数
10     console.log(readBuffer.slice(0, readByte)); // 数据已被填充到 readBuffer 中
11   })
12 })
```

读取文件也可以调用 `fs.readFile()` 方法, 语法格式如下:

```
fs.readFile(filename[, options], callback)
```

- `filename`: 要读取的文件。
- `options`: 一个包含可选值的对象。
  - `encoding {String | Null}`: 默认为 `null`。
  - `flag {String}`: 默认为 '`r`'。
- `callback`: 回调函数。

`fs.readFile` 方法是在 `fs.read` 上的进一步封装, 两者的主要区别是 `fs.readFile` 方法只能读取文件的全部内容。



js 文件必须保存为 UTF8 编码格式。使用 Node.js 开发时, 无论是代码文件还是要读写的其他文件都建议使用 UTF8 编码格式保存, 这样可以无须额外的模块支持。

### 5.2.4 写入文件

写入文件一般调用 `fs.writeFile` 和 `fs.appendFile` 方法。两者都可以将字符串或者缓存区中的内容直接写入文件, 如果检测到文件不存在将创建新的文件。`fs.writeFile` 和 `fs.appendFile` 的语法格式也非常接近, 分别如下:

## (1) fs.writeFile 语法:

```
fs.writeFile(filename, data[, options], callback)
```

参数使用说明如下：

- path：文件路径。
- data：写入文件的数据，可以是 string（字符串）或 buffer（流）对象。
- options：该参数是一个对象，包含{encoding, mode, flag}，默认编码为 utf-8，模式为 0666，flag 为 'w'。
- callback：回调函数，只包含错误信息参数（err）。

## (2) fs.appendFile 语法:

```
fs.appendFile(file, data[, options], callback)
```

参数说明如下：

- file：文件名或者文件描述符。
- data：可以是 string（字符串）或 buffer（流）对象。
- options：该参数是一个对象，包含{encoding, mode, flag}，默认编码为 utf-8，模式为 0666，flag 为 'w'。
- callback：回调函数，只包含错误信息参数（err）。

下面将字符串（string）和流（buffer）作为数据源写入一个文件中：

## 【代码 5-11】

```
01 // 使用 string 写入文件
02 fs.appendFile('message.txt', 'data to append', 'utf8', callback);
03 // 使用 buffer 写入文件
04 fs.appendFile('message.txt', 'data to append', (err) => {
05     if (err) throw err;
06     console.log('The "data to append" was appended to file!');
07 });
```

## 【代码说明】

在执行写入文件之后，不要使用提供的缓存区，因为一旦将其传递给写入函数，缓存区就处于写入操作的控制之下，直到函数结束之后才可以重新使用。



在写入文件时一般要包含写入信息的具体位置，如果以追加模式打开文件，那么文件的游标位于文件的尾部，因此写入的数据也处于文件的尾部。

## 5.3 其他文件操作

在实际的编程过程中，我们需要操作多种不同格式的文件。Node.js 除了提供官方的 API 对文件操作进行支持，也可以通过 NPM 安装第三方的模块来进行文件操作。本节主要介绍如何通过 Node.js 和第三方模块来操作 CSV 文件、XML 文件和 JSON 文件。本节我们以 CSV 文件为例来详细介绍。

CSV 是一种常见的数据格式。Node.js 中有很多模块可以解析 CSV 文件，这里建议使用 node-CSV 来进行文件的解析操作。node-CSV 遵循开源的 BSD 协议，项目在 git 网站的网址为 <https://github.com/wdavidw/node-CSV>。它一共包含 4 个包，分别为 CSV-generate、CSV-parse、stream-transform 和 CSV-stringify。各个包的功能具体如下：

- CSV-generate：用来生成标准的 CSV 文件。
- CSV-parse：将 CSV 文件解析为数组变量。
- stream-transform：一个转换框架。
- CSV-stringify：将记录转换为 CSV 的文本。

使用 node-CSV 时，需要先通过 npm 命令来安装 CSV 的包，具体命令如下：

```
npm install csv
```

其中每个包都与 stream2 和 stream3 的标准相兼容，并且提供一个简单的回调函数。CSV-parse 解析方法可以使用多种选项，但所有的选项都是可选的，而不是必需的，参见表 5.2。

表 5.2 fs.open 中 flag 参数说明

| flag 值                           | 说明   |
|----------------------------------|--|
| delimiter (char)                 | 设定分隔符，只能设定一个字符，默认为','                                    |
| rowDelimiter (chars constant)    | 定义行分隔符，默认值为'auto'，也可以设定为'unix'、'mac'、'windows'、'unicode' |
| quote (char)                     | 默认为双引号，可以用来限定一个范围  |
| escape (char)                    | 设定转义字符，只能设定一个字符，默认为双引号                                   |
| columns (array boolean function) | 将一段数据设置为数组，默认为 null                                      |
| comment (char)                   | 注释，将后面的字符串都当作注释字段，默认为"                                   |
| objname (string)                 | 设置标题名称   |
| skip_empty_lines (boolean)       | 忽略内容为空的行   |
| trim (boolean)                   | 默认值为 false。如果设定为 true，就将忽略分隔符附近的空格                       |
| ltrim (boolean)                  | 默认值为 false。如果设置为 true，就将忽略分隔符后面的空格                       |
| rtrim (boolean)                  | 默认值为 false。如果设置为 true，就将忽略分隔符前面的空格                       |
| auto_parse (boolean)             | 如果设置为 true，将从默认读取数据类型转换为 native 类型                       |
| auto_parse_date (boolean)        | 如果设置为 true，将尝试转换读取数据类型为 dates 类型                         |

下面的代码将使用 CSV 模块中的 stream 来读取、解析和转换 CSV 文件。

### 【代码 5-12】

```

01 var cvs = require('CSV');
02 var generator = CSV.generate({seed: 1, columns: 2, length: 20});
03 var parser = CSV.parse();
04 var transformer = CSV.transform(function(data) {
05   return data.map(function(value){return value.toUpperCase()});
06 });
07 var stringifier = CSV.stringify();
08 generator.on('readable', function() {
09   while(data = generator.read()){
10     parser.write(data);
11   }
12 });
13 //解析生成的 csv 文件
14 parser.on('readable', function(){
15   while(data = parser.read()){
16     transformer.write(data);
17   }
18 });
19 //将 csv 文件转换为 txt 文件
20 transformer.on('readable', function(){
21   while(data = transformer.read()){
22     stringifier.write(data);
23   }
24 });
25 stringifier.on('readable', function(){
26   while(data = stringifier.read()){
27     process.stdout.write(data);
28   }
29 });

```

### 【代码说明】

首先要调用 `require('csv')` 引用 csv 模块，引用之后，就可以直接调用它封装的方法和属性了。`csv()` 相当于实例化一个对象，`.from()` 和 `.to()` 都是 csv 封装的方法。

- `.from()` 方法：从源文件中读取数据，参数既可以像上面一样直接传字符串，也可以像下面的高级应用传源文件的路径。
- `.to()` 方法：将从 `form()` 方法中读取出来的数据输出，既可以输出到控制台，也可以输出到目标文件。此例子是输出到控制台。

# 第6章

# Node.js网络开发

网络是通信互联的基础，Node.js 提供了 net、http、dgram 等模块，分别用来实现 TCP、HTTP、UDP 的通信。

通过本章的学习可以：

- 掌握 TCP 服务器和客户端的创建。
- 掌握 HTTP 的路由控制思想。
- 掌握 UDP 数据通信的实现。

## 6.1 构建 TCP 服务器

OSI 参考模型将网络通信功能划分为 7 层，即物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。TCP 协议就是位于传输层的协议。Node.js 在创建一个 TCP 服务器的时候使用的是 net（网络）模块。

### 6.1.1 使用 Node.js 创建 TCP 服务器

为了使用 Node.js 创建 TCP 服务器，首先要调用 require('net') 来加载 net 模块，然后调用 net 模块的 createServer 方法就可以轻松地创建一个 TCP 服务器。

```
net.createServer([options], [connectionListener])
```

- options 是一个对象参数值，有两个布尔类型的属性 allowHalfOpen 和 pauseOnConnect。这两个属性默认都是 false。
- connectionListener 是一个当客户端与服务端建立连接时的回调函数，这个回调函数以 socket 端口对象作为参数。

如下代码构建一个 TCP 服务器。

**【代码 6-1】**

```

01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建 TCP 服务器*/
04 var server = net.createServer(function(socket) {
05     console.log('someone connects');
06 });

```

### 6.1.2 监听客户端的连接

使用 TCP 服务器的 `listen` 方法就可以开始监听客户端的连接：

```
server.listen(port[, host] [, backlog] [, callback]);
```

- `port`: 为需要监听的端口号。此参数值为 0 的时候将随机分配一个端口号。
- `host`: 服务器地址。
- `backlog`: 连接等待队列的最大长度。
- `callback`: 回调函数。

如下代码可以创建一个 TCP 服务器并监听 8001 端口。

**【代码 6-2】**

```

01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建 TCP 服务器*/
04 var server = net.createServer(function(socket) {
05     console.log('someone connects');
06 });
07 /*设置监听端口*/
08 server.listen(8001, function() {
09     console.log('server is listening');
10 });

```

运行这段代码，可以在控制台看到执行了 `listen` 方法的回调函数，如图 6.1 所示。

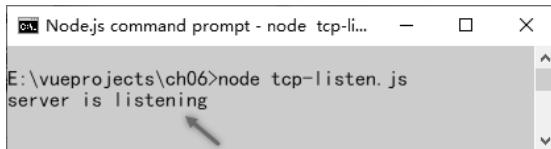


图 6.1 执行了 `listen` 方法的回调函数

可以使用相应的 TCP 客户端或者调试工具来连接这个已经创建好的 TCP 服务器。例如，要使用 Windows 的 Telnet 就可以用以下命令来连接：

```
open localhost 8001
```

连接成功后可以看到控制台打印了“`someone connects`”字样，表明 `createServer` 方法的回调函数已经执行，说明已经成功连接到这个创建好的 TCP 服务器。

server.listen()方法其实触发的是 server 下的 listening 事件，所以也可以手动监听 listening 事件。如下代码同样实现了创建一个 TCP 服务器并监听 8001 端口。

### 【代码 6-3】

```
01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建 TCP 服务器*/
04 var server = net.createServer(function(socket) {
05     console.log('someone connects');
06 });
07 /*设置监听端口*/
08 server.listen(8001);
09 /*设置监听时的回调函数*/
10 server.on('listening',function(){
11     console.log('server is listening');
12 });
```

除了 listening 事件外，TCP 服务器还支持以下事件：

- connection：当有新的链接创建时触发，回调函数的参数为 socket 连接对象。
- close：TCP 服务器关闭的时候触发，回调函数没有参数。
- error：TCP 服务器发生错误的时候触发，回调函数的参数为 error 对象。

下列代码通过 net.Server 类来创建一个 TCP 服务器，添加以上事件。

### 【代码 6-4】

```
01 /*引入 net 模块*/
02 var net = require('net');
03 /*实例化一个服务器对象*/
04 var server = new net.Server();
05 /*监听 connection 事件*/
06 server.on('connection', function(socket) {
07     console.log('someone connects');
08 });
09 /*设置监听端口*/
10 server.listen(18001);
11 /*设置监听时的回调函数*/
12 server.on('listening', function() {
13     console.log('server is listening');
14 });
15 /*设置关闭时的回调函数*/
16 server.on('close', function() {
17     console.log('server closed');
18 });
19 /*设置出错时的回调函数*/
20 server.on('error', function(err) {
21     console.log('error');
22 });
```

运行以上这段代码并用 Telnet 等工具连接这个创建的 TCP 服务器，可以发现效果和【代码 6-3】代码的效果一致。

### 6.1.3 查看服务器监听的地址

当创建了一个 TCP 服务器后，可以通过 `server.address()` 方法来查看这个 TCP 服务器监听的地址，并返回一个 JSON 对象。这个对象的属性有：

- `port`: TCP 服务器监听的端口号。
- `family`: 说明 TCP 服务器监听的地址是 IPv6 还是 IPv4。
- `address`: TCP 服务器监听的地址。

因为这个方法返回的是 TCP 服务器监听的地址信息，所以应该在调用了 `server.listen()` 方法或者绑定了事件 `listening` 中的回调函数中调用该方法。

#### 【代码 6-5】

```

01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建服务器*/
04 var server = net.createServer(function(socket) {
05     console.log('someone connects');
06 });
07 /*设置监听端口*/
08 server.listen(8001,function() {
09 /*获取地址信息*/
10     var address = server.address();
11 /*获取地址端口*/
12     console.log('the port of server is ' + address.port);
13     console.log('the address of server is ' + address.address);
14     console.log('the family of server is ' + address.family);
15 });

```

运行这段代码可以发现已经在控制台打印出 TCP 服务器监听的地址信息，如图 6.2 所示。

```

E:\vueprojects\ch06>node tcp-address.js
the port of server is 8001
the address of server is ::
the family of server is IPv6

```

图 6.2 TCP 服务器监听的地址信息

### 6.1.4 连接服务器的客户端数量

创建一个 TCP 服务器后，可以通过 `server.getConnections()` 方法获取连接这个 TCP 服务器的客户端数量。这个方法是一个异步的方法，回调函数有两个参数：

- 第一个参数为 error 对象。
- 第二个参数为连接 TCP 服务器的客户端数量。

除了获取连接数外，也可以通过设置 TCP 服务器的 maxConnections 属性来设置这个 TCP 服务器的最大连接数。当连接数超过最大连接数的时候，服务器将拒绝新的连接。如下代码设置这个 TCP 服务器的最大连接数为 3。

### 【代码 6-6】

```

01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建服务器*/
04 var server = net.createServer(function(socket) {
05     console.log('someone connects');
06     /*设置最大连接数量*/
07     server.maxConnections = 3;
08     server.getConnections(function(err, count) {
09         console.log('the count of client is ' + count);
10     });
11 });
12 /*设置监听端口*/
13 server.listen(8001,function() {
14     console.log('server is listening');
15 });

```

运行这段代码，并尝试用多个客户端连接。可以发现当客户端连接数超过 3 的时候，新的客户端就无法连接这个服务器了，如图 6.3 所示。

```

  Node.js command prompt - node tcp-cl...
E:\vueprojects\ch06>node tcp-cl...
server is listening
someone connects
the count of client is 1
someone connects
the count of client is 2
someone connects
the count of client is 3

```

图 6.3 设置 TCP 服务器的最大连接数量

### 6.1.5 获取客户端发送的数据

上文提到 createServer 方法的回调函数参数是一个 net.Socket 对象（服务器所监听的端口对象）。这个对象同样也有一个 address()方法，用来获取 TCP 服务器绑定的地址，同样也是返回一个含有 port、family、address 属性的对象。

通过 socket 对象可以获取客户端发送的流数据，每次接收到数据的时候触发 data 事件，通过监听这个事件就可以在回调函数中获取客户端发送的数据，代码如下：

**【代码 6-7】**

```

01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建服务器*/
04 var server = net.createServer(function(socket) {
05 /*监听 data 事件*/
06     socket.on('data', function(data) {
07         /*打印 data*/
08         console.log(data.toString());
09     });
10 });
11 /*设置监听端口*/
12 server.listen(8001, function() {
13     console.log('server is listening');
14 });

```

运行这段代码之后，通过 Telnet 等工具连接后，发送一段数据给服务端，在命令行中就可以发现数据已经被打印出来了，如图 6.4 所示。

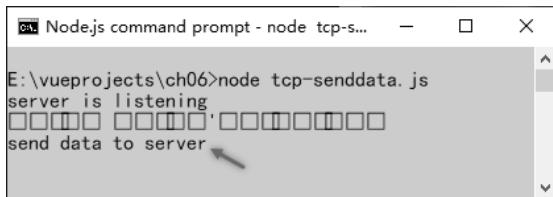


图 6.4 打印客户端发送的数据

socket 对象除了有 data 事件外，还有 connect、end、error、timeout 等事件。

### 6.1.6 发送数据给客户端

调用 socket.write()可以使 TCP 服务器发送数据。这个方法只有一个必需参数，就是需要发送的数据；第二个参数为编码格式，可选。同时，可以为这个方法设置一个回调函数。当有用户连接 TCP 服务器的时候，将发送数据给客户端，代码如下：

**【代码 6-8】**

```

01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建服务器*/
04 var server = net.createServer(function(socket) {
05 /*获取地址信息*/
06     var address = server.address();
07     var message = 'client, the server address is ' + JSON.stringify(address);
08 /*发送数据*/
09     socket.write(message, function() {
10         var writeSize = socket.bytesWritten;
11         console.log(message + ' has send');
12         console.log('the size of message is ' + writeSize);

```

```

13      });
14  /*监听 data 事件*/
15  socket.on('data', function(data) {
16      console.log(data.toString());
17      var readSize = socket.bytesRead;
18      console.log('the size of data is ' + readSize);
19  });
20 });
21 /*设置监听端口*/
22 server.listen(8001, function() {
23     console.log('server is listening');
24 });

```

运行这段代码并连接 TCP 服务器，可以看到 Telnet 中收到了 TCP 服务器发送的数据，Telnet 也可以发送数据给 TCP 服务器，如图 6.5 所示。

```

Nodejs command prompt - node tcp-sendtoclient.js
E:\vueprojects\ch06>node tcp-sendtoclient.js
server is listening
client, the server address is {"address":":8001","family":"IPv6","port":8001}has send
the size of message is 74
□□□□□□□□□□□□□□□□□□□□□□□□□□
the size of data is 21

```

图 6.5 TCP 服务器发送数据

在上面这段代码中还用到了 socket 对象的 bytesWritten 和 bytesRead 属性，这两个属性分别代表着发送数据的字节数和接收数据的字节数。

除了上面这两个属性外，socket 对象还有以下属性：

- socket.localPort：本地端口的地址。
- socket.localAddress：本地 IP 地址。
- socket.remotePort：远程端口地址。
- socket.remoteFamily：远程 IP 协议族。
- socket.remoteAddress：远程 IP 地址。

以下这段代码就将这些属性打印在控制台上。

### 【代码 6-9】

```

01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建服务器*/
04 var server = net.createServer(function(socket) {
05     /*本地端口*/
06     console.log('localPort: ' + socket.localPort);
07     /*本地 IP 地址*/
08     console.log('localAddress: ' + socket.localAddress);
09     /*远程端口*/

```

```

10     console.log('remotePort: ' + socket.remotePort);
11  /*远程 IP 协议族*/
12      console.log('remoteFamily: ' + socket.remoteFamily);
13  /*远程 IP 地址*/
14      console.log('remoteAddress: ' + socket.remoteAddress);
15  });
16  /*设置监听端口*/
17 server.listen(8001,function() {
18     console.log('server is listening');
19 });

```

运行这段代码并连接 TCP 服务器，可以在命令行中看到图 6.6 所示的信息。

```

Node.js command prompt - node tcp-rem...
E:\vueprojects\ch06>node tcp-rem...
server is listening
localPort: 8001
localAddress: ::1
remotePort: 11926
remoteFamily: IPv6
remoteAddress: ::1

```

图 6.6 socket 的相关属性

## 6.2 构建 TCP 客户端

Node.js 在创建一个 TCP 客户端的时候同样使用的是 net（网络）模块。

### 6.2.1 使用 Node.js 创建 TCP 客户端

为了使用 Node.js 创建 TCP 客户端，首先要调用 require('net') 来加载 net 模块。创建一个 TCP 客户端只需要创建一个连接 TCP 客户端的 socket 对象即可：

```

/*引入 net 模块*/
var net = require('net');
/*创建客户端*/
var client = new net.Socket();

```

创建一个 socket 对象的时候可以传入一个 json 对象。这个对象有以下属性：

- fd：指定一个存在的文件描述符，默认值为 null。
- readable：是否允许在这个 socket 上读，默认值为 false。
- writeable：是否允许在这个 socket 上写，默认值为 false。
- allowHalfOpen：该属性为 false 时，TCP 服务器接收到客户端发送的一个 FIN 包后，将会回发一个 FIN 包；该属性为 true 时，TCP 服务器接收到客户端发送的一个 FIN 包后不会回发 FIN 包。

## 6.2.2 连接 TCP 服务器

创建了一个 socket 对象后，调用 socket 对象的 connect()方法就可以连接一个 TCP 服务器。例如，连接【代码 6-10】中创建的 TCP 服务器，可以使用以下代码：

**【代码 6-10】**

```
01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建客户端*/
04 var client = net.Socket();
05 /*设置连接的服务器*/
06 client.connect(8001, '127.0.0.1', function() {
07   console.log('connect the server');
08 });
```

运行【代码 6-10】启动 TCP 服务器之后，可以在命令行中发现打印了一些字样，说明 connect()方法的回调函数已经执行了，即已经成功连接上 TCP 服务器，如图 6.7 所示。

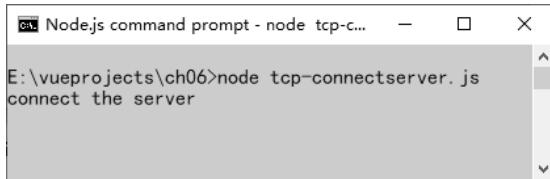


图 6.7 连接 TCP 服务器

## 6.2.3 获取从 TCP 服务器发送的数据

在 6.1 节中已经介绍了一个 socket 对象有 data、error、close、end 等事件，因此也可以通过监听 data 事件来获取从 TCP 服务器发送的数据。

**【代码 6-11】**

```
01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建客户端*/
04 var client = net.Socket();
05 /*设置连接的服务器*/
06 client.connect(8001, '127.0.0.1', function() {
07   console.log('connect the server');
08 });
09 /*监听 data 事件*/
10 client.on('data', function(data) {
11   console.log('the data of server is ' + data.toString());
12 });
```

运行【代码 6-8】启动 TCP 服务器之后，再运行上面这段代码，可以发现命令行中已经输出了来自服务端的数据，说明此时已经实现了服务端和客户端之间的通信。

### 6.2.4 向 TCP 服务器发送数据

因为 TCP 客户端是一个 socket 对象，在 6.1 节中提到的 write()方法以及 localPort、localAddress 等属性依旧可用，所以可以使用以下代码来向 TCP 服务器发送数据。

【代码 6-12】

```

01 /*引入 net 模块*/
02 var net = require('net');
03 /*创建客户端*/
04 var client = net.Socket();
05 /*设置连接的服务器*/
06 client.connect(8001, '127.0.0.1', function() {
07     console.log('connect the server');
08     /*发送数据*/
09     client.write('message from client');
10 });
11 /*监听 data 事件*/
12 client.on('data', function(data) {
13     console.log('the data of server is ' + data.toString());
14 });
15 /*监听 end 事件*/
16 client.on('end', function(){
17     console.log('data end');
18 });

```

运行【代码 6-8】启动 TCP 服务器之后，再运行上面这段代码，可以发现服务器已经接收到客户端的数据，客户端也已经接收到服务端的数据，如图 6.8 和图 6.9 所示。

```

Node.js command prompt - node tcp-sock...
E:\vueprojects\ch06>node tcp-socketwrite.js
connect the server
the data of server is client, the server address is
{"address": "::", "family": "IPv6", "port": 8001}

```

图 6.8 TCP 客户端

```

Node.js command prompt - node tcp-sendtoclient.js
E:\vueprojects\ch06>node tcp-sendtoclient.js
server is listening
client, the server address is {"address": "::", "family": "IPv6", "port": 8001} has send
the size of message is 74
message from client
the size of data is 19

```

图 6.9 TCP 服务端

当然，客户端和服务端也可以通过流的形式将文件中的数据发送出去，相关的知识可以在文件模块中进行学习。

# 6.3 构建 HTTP 服务器

在如今 Web 大行其道的时代，支撑无数网页运行的正是 HTTP 服务器。Node.js 之所以受到大量 Web 开发者的青睐，与 Node.js 有能力自己构建服务器是分不开的。

## 6.3.1 创建 HTTP 服务器

在本书的第 4 章中已经提到 HTTP 服务器。只需要使用以下代码就可以创建一个简单的 HTTP 服务器。

【代码 6-13】

```
01 /*引入 http 模块*/
02 var http = require('http');
03 /*创建 HTTP 服务器*/
04 var server = http.createServer(function(req, res) {
05     /*设置响应的头部*/
06     res.writeHead(200, {
07         'content-type': 'text/plain'
08     });
09     /*设置响应的数据*/
10     res.end('Hello, Node.js!');
11 });
12 /*设置服务器端口*/
13 server.listen(3000, function() {
14     console.log('listening port 3000');
15 });
```

通过这段代码可以在浏览器中看到创建的服务器发送给浏览器的数据。在第 4 章中已经说明了 http 模块的主要应用，这里不再讲解，将重点放在 HTTP 服务器优化上。

上面这个 HTTP 服务器只是实现了将一行字符串的数据发送给浏览器。很明显，如果服务器仅仅能发送一些字符串，那几乎是不可用的，因此需要对上面这个服务器的功能进行拓展。通过文件模块读取文件并发送给浏览器就是一个不错的选择，将上面的代码修改为以下代码：

【代码 6-14】

```
01 /*引入 http 模块*/
02 var http = require('http');
03 /*引入 fs 模块*/
04 var fs = require('fs');
05 /*创建 HTTP 服务器*/
06 var server = http.createServer(function(req, res) {
07     /*设置响应的头部*/
08     res.writeHead(200, {
```

```

09         'content-type': 'text/html'
10     });
11     /*读取文件数据*/
12     var data = fs.readFileSync('./index.html');
13     /*响应数据*/
14     res.write(data);
15     res.end();
16   });
17   /*设置服务器端口*/
18   server.listen(3000, function() {
19     console.log('listening port 3000');
20   });

```

同时在同级目录中创建一个名为 index.html 的文件，写入以下代码：

#### 【代码 6-15】

```

01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04   <meta charset="UTF-8">
05   <title>Node.js</title>
06   <style>
07     h1 {
08       color:red;
09     }
10   </style>
11 </head>
12 <body>
13 <h1>Hello,Node.js</h1>
14 </body>
15 </html>

```

运行代码，利用浏览器访问 localhost:3000 这个地址，即可看到如图 6.10 所示的显示信息。



图 6.10 HTTP 服务器发送文件信息

需要提及的是，这里的 HTTP 服务器在发送给浏览器的头部信息中会把 content-type 修改为 text/html。content-type 的作用就是用来表示客户端或者服务器传输数据的类型，服务器和客户端通过这个值来做相应的解析。如果将这个值修改为原来的 text/plain，浏览器中将显示 index.htm 文件中的所有代码，这显然不是我们所希望的。

### 6.3.2 HTTP 服务器的路由控制

上一节中的服务器虽然已经可以通过读取文件数据来发送给客户端了，但是并没有做任何的路由控制，在浏览器中输入任何 URL 都将返回同样的内容。简单来说，路由就是 URL 到函

数的映射。

要做到路由控制，通过上面的学习可以预想到，需要设定的必然有 `content-type`。这里假定只需要处理 `html`、`js`、`css` 和图片文件，创建一个名为 `mime.js` 的文件，写入以下代码：

```
module.exports = {
  ".html": "text/html",
  ".css": "text/css",
  ".js": "text/javascript",
  ".gif": "image/gif",
  ".ico": "image/x-icon",
  ".jpeg": "image/jpeg",
  ".jpg": "image/jpeg",
  ".png": "image/png"
};
```

需要做到路由控制，也就需要知道用户请求的 URL 地址，也就是 `req.url`，所以通过这个属性获取到 URL 后就可以对路由进行控制了，如以下代码所示：

### 【代码 6-16】

```
01 /*引入 http 模块*/
02 var http = require('http');
03 /*引入 fs 模块*/
04 var fs = require('fs');
05 /*引入 url 模块*/
06 var url = require('url');
07 /*引入 mime 文件*/
08 var mime = require('../mime');
09 /*引入 path 模块*/
10 var path = require('path');
11 /*创建 HTTP 服务器*/
12 var server = http.createServer(function(req, res) {
13   var filePath = '.' + url.parse(req.url).pathname;
14   if(filePath === './'){
15     filePath = './index.html';
16   }
17   /*判断相应的文件是否存在*/
18   fs.exists(filePath, function(exist){
19     /*存在则返回相应文件数据*/
20       if(exist) {
21         var data = fs.readFileSync(filePath);
22         var contentType = mime[path.extname(filePath)];
23         res.writeHead(200, {
24           'content-type': contentType
25         });
26         res.write(data);
27         res.end();
28       }else{
29         /*不存在则返回 404 */
30         res.end('404');
31       }
32     }
33   });
34   server.listen(3000, function(){
35     console.log('服务器启动成功');
36   });
37 }
```

```

31      }
32    })
33  });
34 /*设置服务器端口*/
35 server.listen(3000, function() {
36   console.log('listening port 3000');
37 });

```

这里通过 req.url 对路径进行处理来返回不同的资源，从而做到简单的路由控制。

## 6.4 利用 UDP 协议传输数据与发送消息

在前文中，我们所提到的 TCP 数据传输是一种可靠的数据传输方式，在数据传输之前必须建立客户端与服务端之间的连接。UDP 是一种面向非连接的协议，所以其传输速度比 TCP 更加快速。

### 6.4.1 创建 UDP 服务器

为了使用 Node.js 创建 UDP 服务器，首先要调用 require('dgram') 加载 dgram 模块。

调用 dgram 模块中的 createSocket() 方法来创建一个 UDP 服务器。这个方法接收一个必需参数和一个可选参数，必需参数是一个表示 UDP 协议的类型，可指定为 udp4 或者 udp6，代码如下：

```

/*引入 dgram 模块*/
var dgram = require('dgram');
/*创建 UDP 服务器*/
var socket = dgram.createSocket('udp4');

```

这个方法中的可选参数为一个回调函数，是 UDP 服务器接收数据时触发的回调函数，可接收两个参数，一个为接收到的数据，另一个为存放发送者信息的对象，代码如下：

```

/*引入 dgram 模块*/
var dgram = require('dgram');
/*创建 UDP 服务器*/
var socket = dgram.createSocket('udp4', function (msg, rinfo) {
  // your code
});

```

rinfo 对象的属性及属性值如下：

- address：表示发送者地址。
- family：表示发送者使用的地址为 ipv4 或者 ipv6。
- port：表示发送者的端口号。
- size：表示发送者发送数据的字节数大小。

创建完一个 socket 端口对象后还需要绑定一个端口号才能创建 UDP 服务器，可利用 socket.bind()方法绑定一个端口号。这个方法接收一个必需参数、两个可选参数。必需参数为需要绑定的端口号，两个可选参数为地址和回调函数，代码如下：

#### 【代码 6-17】

```
01 /*引入 dgram 模块*/
02 var dgram = require('dgram');
03 /*创建 UDP 服务器*/
04 var socket = dgram.createSocket('udp4', function (msg, rinfo) {
05   // your code
06 });
07 /*绑定端口*/
08 socket.bind(41234, 'localhost', function () {
09   console.log('bind 41234');
10 });
```

这是一个简单的 UDP 服务器，与 net 和 http 方法类似，因为 createSocket 方法返回的是一个 socket 对象。一个 socket 对象主要有以下事件：

- message：接收数据时触发。
- listening：开始监听数据报文时触发。
- close：关闭 socket 时触发。
- error：发生错误时触发。

显然上文中调用 createSocket()方法中的回调函数就是监听 message 事件，因此调用 createSocket()方法时可以不指定回调函数，直接显式监听 message 事件，同样可以达到相应的效果：

#### 【代码 6-18】

```
01 /*引入 dgram 模块*/
02 var dgram = require('dgram');
03 /*创建 UDP 服务器*/
04 var socket = dgram.createSocket('udp4');
05 /*绑定端口*/
06 socket.bind(41234, 'localhost', function () {
07   console.log('bind 41234');
08 });
09 /*监听 message 事件*/
10 socket.on('message', function (msg, rinfo) {
11   console.log(msg.toString());
12 });
```

将事件综合使用，代码如下所示：

#### 【代码 6-19】

```
01 /*引入 dgram 模块*/
02 var dgram = require('dgram');
```

```

03 /*创建 UDP 服务器*/
04 var socket = dgram.createSocket('udp4');
05 /*绑定端口*/
06 socket.bind(41234, 'localhost', function () {
07   console.log('bind 41234');
08 });
09 /*监听 message 事件*/
10 socket.on('message', function (msg, rinfo) {
11   console.log(msg.toString());
12 });
13 /*监听 listening 事件*/
14 socket.on('listening', function() {
15   console.log('listening begin');
16 });
17 /*监听 close 事件*/
18 socket.on('close', function(){
19   console.log('server closed');
20 });
21 /*监听 error 事件*/
22 socket.on('error', function (err) {
23   console.log(err);
24 });

```

一个 socket 对象主要有以下方法：

- bind(): 绑定端口号。
- send(): 发送数据。
- address(): 获取该 socket 端口对象相关的地址信息。
- close(): 关闭 socket 对象。

bind()方法在上文中已经介绍，send()方法用来发送数据，其完整的参数使用如下：

```
socket.send(buf, offset, length, port, address[,callback])
```

- buf: 代表需要发送的消息，可以是缓存对象或者字符串。
- offset: 是一个整数数字，代表消息在缓存中的偏移量。
- length: 是一个整数数字，代表消息的比特数。
- port: 代表发送数据的端口号。
- address: 代表接收数据的 socket 端口对象的地址。
- callback: 为数据发送完毕所需调用的回调函数。这个回调函数的第一个参数是 error 对象，第二个参数为数据发送的比特数。

因此使用这个方法的代码看起来会是这样：

#### 【代码 6-20】

```

01 /*引入 dgram 模块*/
02 var dgram = require('dgram');
03 /*创建 buffer*/

```

```

04 var message = new Buffer('some message');
05 /*创建 UDP 服务器*/
06 var socket = dgram.createSocket('udp4', function (msg, rinfo) {
07   console.log(msg.toString());
08   /*发送数据*/
09   socket.send(message, 0, message.length, rinfo.port, rinfo.address, function
10     (err, bytes) {
11       if(err) {
12         console.log(error);
13         return;
14       }
15       console.log("send " + bytes + ' message');
16     })
17   /*绑定端口*/
18   socket.bind(41234, 'localhost', function () {
19     console.log('bind 41234');
20   });

```

## 6.4.2 创建 UDP 客户端

因为 UDP 客户端本质上其实也是一个 socket 端口对象，所以同样可以通过创建一个 socket 对象来构建 UDP 客户端，这样得到的是一个 socket 对象，所以同样可以使用上述介绍的相关方法。如下代码就可以实现一个简单的 UDP 客户端：

### 【代码 6-21】

```

01 /*引入 dgram 模块*/
02 var dgram = require('dgram');
03 /*创建 buffer*/
04 var message = new Buffer('some message from client');
05 /*创建 UDP 服务器*/
06 var socket = dgram.createSocket('udp4');
07 /*发送数据*/
08 socket.send(message, 0, message.length, 41234, 'localhost',
09   function (err, bytes) {
10     if(err) {
11       console.log(err);
12       return;
13     }
14     console.log('client send ' + bytes + 'message');
15   });
16 /*监听 message 事件*/
17 socket.on('message', function (msg, rinfo) {
18   console.log("some message form server");
19 });

```

因此通过创建一个 socket 对象作为客户端和一个 socket 对象作为服务端就可以实现 UDP 协议的通信了。

**【代码 6-22】**

```

01 /*引入 dgram 模块*/
02 var dgram = require('dgram');
03 /*创建 buffer*/
04 var message = new Buffer('some message from server');
05 /*创建 UDP 服务器*/
06 var socket = dgram.createSocket('udp4', function (msg, rinfo) {
07   console.log(msg.toString());
08   /*发送数据*/
09   socket.send(message, 0, message.length, rinfo.port, rinfo.address,
10     function (err, bytes) {
11       if(err) {
12         console.log(error);
13         return;
14       }
15       console.log("send " + bytes + ' message');
16     })
17   });
18 /*绑定端口*/
19 socket.bind(41234, 'localhost', function () {
20   console.log('bind 41234');
21 });

```

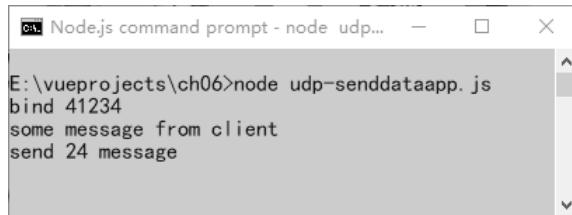
**【代码 6-23】**

```

01 /*引入 dgram 模块*/
02 var dgram = require('dgram');
03 /*创建 buffer*/
04 var message = new Buffer('some message from client');
05 /*创建 UDP 服务器*/
06 var socket = dgram.createSocket('udp4');
07 /*发送数据*/
08 socket.send(message, 0, message.length, 41234, 'localhost',
09   function (err, bytes) {
10     if(err) {
11       console.log(err);
12       return;
13     }
14     console.log('client send ' + bytes + 'message');
15   });
16 /*监听 message 事件*/
17 socket.on('message', function (msg, rinfo) {
18   console.log("some message form server");
19 });

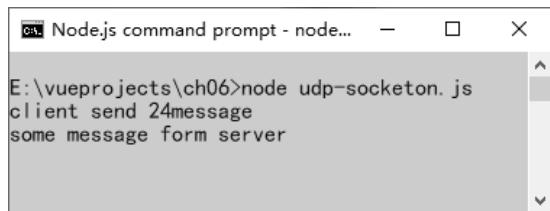
```

运行【代码 6-22】和【代码 6-23】，依次启动 UDP 服务器和 UDP 客户端，可以发现已经实现了 UDP 服务器和 UDP 客户端的通信，如图 6.11 和图 6.12 所示。



```
Node.js command prompt - node udp... ━ ━ X  
E:\vueprojects\ch06>node udp-senddataapp.js  
bind 41234  
some message from client  
send 24 message
```

图 6.11 UDP 服务器



```
Node.js command prompt - node... ━ ━ X  
E:\vueprojects\ch06>node udp-socketon.js  
client send 24message  
some message form server
```

图 6.12 UDP 客户端

# 第 7 章

## Node.js 数据库开发

数据库在互联网中的重要性不言而喻。数据库存放着大量的信息，是很多互联网公司的命脉。目前运用广泛的有关系型数据库和非关系型数据库。MySQL 数据库是关系型数据库的杰出代表，MongoDB 则是近几年大热的非关系型数据库。本章将主要介绍 Node.js 与这两种数据库的连接和交互操作。

通过本章的学习可以：

- 掌握连接 MySQL 数据库并进行操作的方法。
- 掌握连接 MongoDB 数据库并进行操作的方法。
- 了解数据库的基础知识。

### 7.1 使用 mongoose 连接 MongoDB

MongoDB 是一个基于分布式文件存储的数据库，由 C++ 语言编写，目的是让 Web 应用提供可拓展的高性能数据存储解决方案。

#### 7.1.1 MongoDB 介绍

目前，MongoDB 是非关系型数据库中功能最丰富、最像关系型数据库的产品。MongoDB 由 10gen 团队在 2007 年发起，2009 年 2 月首度推出。MongoDB 支持的数据结构类似于 json 的 bson 格式。这种数据结构非常松散，可以很方便地存储比较复杂的数据类型。MongoDB 的主要特点是高性能、易存储、易使用、易部署。

MongoDB 的最小数据单位是文档（类似于关系型数据库中的行）。文档是由多个键及其对应的值组成的（类似于 json），一组文档共同组成了一个集合。集合类似于关系型数据库中的表，但是一个集合中的文档可以是各种各样的，一组集合就组成了一个数据库。MongoDB 可以承载多个数据库，这些数据库可以看作是相互独立的。

(1) MongoDB 的官方网站是 <https://www.mongodb.com/>。读者可以在 MongoDB 官方网站的下载页面 <https://www.mongodb.com/download-center> 选择相应的版本进行下载, 如图 7.1 所示。

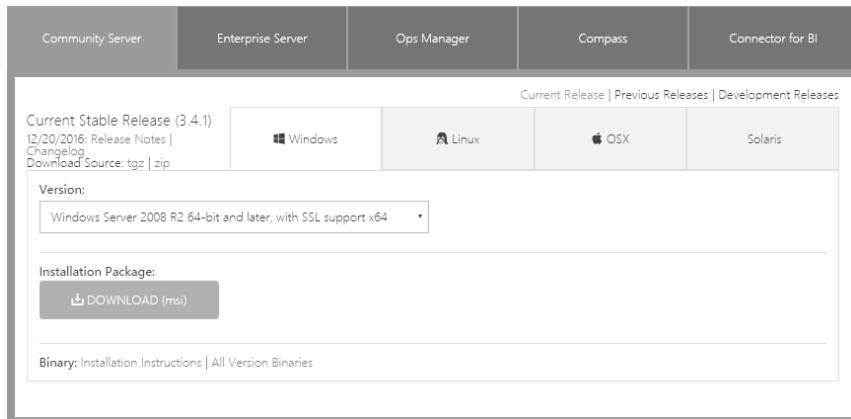


图 7.1 MongoDB 下载页面

(2) 在这里以 Windows 版本为例, 将下载下来的 MongoDB 软件按照常规软件的步骤安装即可。需要提醒的是, 在安装过程中, MongoDB 默认安装在 C 盘, 可以在安装过程中选择安装的路径(因为 MongoDB 的操作需要用到这个路径, 所以读者要选择一个合适的路径进行安装), 如图 7.2 和图 7.3 所示。

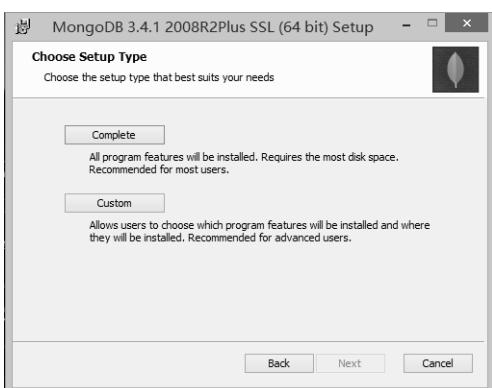


图 7.2 MongoDB 选择 Custom

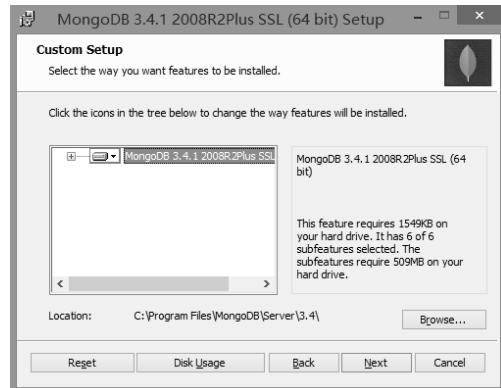


图 7.3 MongoDB 选择 Browse 自定义路径

(3) 安装完成后, 需要配置数据存储的文件夹和 MongoDB 的日志文件夹。在 MongoDB 安装的路径中新建一个名为 db 的文件夹作为数据库存储的文件夹, 同时新建一个名为 mongolog 的文件夹作为日志文件存储的文件夹。在 db 文件夹的同级目录下新建一个名为 mongo.config 的文件作为配置文件, 写入以下内容:

```
##数据文件
dbpath= ##你的数据存储文件夹地址
##日志文件
logpath= ##你的日志文件地址
```

此时，整个目录结构如图 7.4 所示。



图 7.4 MongoDB 配置目录

(4) 打开 CMD 工具，输入以下命令，就可以启动 MongoDB 了：

```
mongod --dbpath 你的 db 文件夹地址
```

这里仅是对 MongoDB 进行简单的介绍，读者可以通过阅读相关的书籍学习更多的知识，比如清华大学出版社出版的《MongoDB 游记之轻松入门到进阶》可以参考。

### 7.1.2 使用 mongoose 连接 MongoDB

mongoose 是一个基于 node-mongodb-native 开发的 MongoDB 的 Node.js 驱动，可以很方便地在异步环境中使用。

mongoose 的 GitHub 地址是 <https://github.com/Automattic/mongoose>，mongoose 的官方网站是 <http://mongoosejs.com/>，读者可以在 mongoose 的官方网站中阅读相应的说明文档。

使用 mongoose 这个模块前，首先需要通过 NPM 安装这个模块：

```
npm install mongoose
```

mongoose 模块通过 connect()方法与 MongoDB 创建连接。connect()方法中需要传递一个 URI 地址，用来说明需要连接的 MongoDB 数据库。如下代码就和本地的 MongoDB 数据库 article 建立了连接。

#### 【代码 7-1】

```
01 /*引入 mongoose 模块*/
02 const mongoose = require('mongoose');
03 /*定义 mongodb 地址*/
04 const uri = 'mongodb://localhost/article';
05 /*连接 mongodb*/
06 mongoose.connect(uri, function(err) {
07     if(err) {
08         console.log('connect failed');
09         console.log(err);
10         return;
11     }
12     console.log('connect success');
13 });


```

#### 【代码说明】

connect()方法创建 MongoDB 连接，回调函数中 err 为参数，出现连接错误则打印出“connect failed”，连接成功则打印出“connect success”。

运行这段代码，如果 MongoDB 服务已经正常开启，就会在控制台打印出“connect success”字样。

需要说明的是，connect()方法中 uri 参数的完整示例应该是：

```
mongodb://user:pass@localhost:port/database
```

- user：代表 MongoDB 的用户名。
- pass：代表用户名对应的密码。
- port：代表 MongoDB 服务的端口号。

### 7.1.3 使用 mongoose 操作 MongoDB

mongoose 中的一切由 schema 开始。schema 是一种以文件形式存储的数据库模型骨架，并不具备数据库的操作能力。schema 中定义了 model 中的所有属性，而 model 则是对应一个 MongoDB 中的 collection。以下代码定义了一个 schema 并且注册成了一个 model。

#### 【代码 7-2】

```
01 /*引入 mongoose 模块*/
02 const mongoose = require('mongoose');
03 /*定义 mongodb 地址*/
04 const uri = 'mongodb://localhost/article';
05 /*连接 mongodb*/
06 mongoose.connect(uri, function(err) {
07     if(err) {
08         console.log('connect failed');
09         console.1go(err);
10         return;
11     }
12     console.log('connect success');
13 });
14 /*定义 Schema*/
15 const ArticleSchema = new mongoose.Schema({
16     title: String,
17     author: String,
18     content: String,
19     publishTime: Date
20 });
21 mongoose.model('Article', ArticleSchema);
```

#### 【代码说明】

这段代码通过实例化一个 mongoose.Schema() 对象定义一个 model 的所有属性，类似于关系型数据库中的字段和字段的数据类型。schema 合法的类型有 String、Number、Date、Buffer、Boolean、Mixed、Objectid 和 Array。mongoose 中通过 mongoose.model() 方法注册一个 model。

在 mongoose 中可以调用 save() 方法将一个新的文档插入到已有的 collection 中，请看下面的代码。

**【代码 7-3】**

```

01 /*引入 mongoose 模块*/
02 const mongoose = require('mongoose');
03 /*定义 mongodb 地址*/
04 const uri = 'mongodb://localhost/article';
05 /*连接 mongodb*/
06 mongoose.connect(uri, function(err) {
07     if(err) {
08         console.log('connect failed');
09         console.log(err);
10         return;
11     }
12     console.log('connect success');
13 });
14 /*定义 Schema*/
15 const ArticleSchema = new mongoose.Schema({
16     title: String,
17     author: String,
18     content: String,
19     publishTime: Date
20 });
21 mongoose.model('Article', ArticleSchema);
22 const Article = mongoose.model('Article');
23 var art = new Article({
24     title: 'node.js',
25     author: 'node',
26     content: 'node.js is great!',
27     publishTime: new Date()
28 });
29 /*将文档插入到集合中*/
30 art.save(function(err) {
31     if(err) {
32         console.log('save failed');
33         console.log(err);
34     }else{
35         console.log('save successed');
36     }
37 });

```

**【代码说明】**

这段代码调用名为 Article 的 model，之后定义了一个 Article 的文档，最后使用 save 将记录插入到相应的 collection 中。save()方法中的回调函数监听是否出错。

运行这段代码，在 MongoDB 运行正常的情况下，控制台将输出“save successed”字样。

可以在控制台对 MongoDB 进行操作来查看 MongoDB 中是否存在这样一条记录，连接完 MongoDB 后打开控制台，输入以下命令切换至 article 数据库：

```
use article
```

切换成 article 数据库后可以使用以下命令来查看 article 这个数据库存在的所有 collection：

```
show collections
```

这时，可以看到控制台中存在一个名为 articles 的 collection，如图 7.5 所示。

```
> show collections
articles
```

图 7.5 查看数据库中的 collection

通过以下命令可以查看这个名为 articles 的 collection 中的所有文档：

```
db.articles.find()
```

如果刚才的这个文档插入成功，控制台就将会显示这个文档，如图 7.6 所示。

```
> db.articles.find()
[{"_id": ObjectId("587a01f9b65e3707d477afb0"), "title": "node.js", "author": "node", "content": "node.js is great!", "publishTime": ISODate("2017-01-14T10:48:25.386Z"), "__v": 0}]
```

图 7.6 文档插入成功

名为 articles 的 collection 中出现了插入的这个文档，说明 MongoDB 中的的确保存了刚刚插入的这条记录。

当然，使用 mongoose 同样可以查询相应的数据。如下代码的功能就是将 articles 这个 collection 中的所有文档查询出来。

#### 【代码 7-4】

```
01 /*引入 mongoose 模块*/
02 const mongoose = require('mongoose');
03 /*定义 mongodb 地址*/
04 const uri = 'mongodb://localhost/article';
05 /*连接 mongodb*/
06 mongoose.connect(uri, function(err) {
07     if(err) {
08         console.log('connect failed');
09         console.log(err);
10         return;
11     }
12     console.log('connect success');
13 });
14 /*定义 Schema*/
15 const ArticleSchema = new mongoose.Schema({
16     title: String,
17     author: String,
18     content: String,
19     publishTime: Date
20 });
21 mongoose.model('Article', ArticleSchema);
22 const Article = mongoose.model('Article');
23 /*查询 mongodb*/
24 Article.find({},function(err, docs) {
25     if(err) {
```

```

26         console.log('error');
27         return;
28     }
29     console.log("result: " + docs);
30 });

```

**【代码说明】**

这段代码通过 `find()`方法查找相应的数据记录。`find()`方法中的第一个参数是一个 json 对象，定义查找的条件，第二个参数为回调函数。回调函数中的第一个参数是 `error`，第二个参数是查询的结果。

运行这段代码，可以发现控制台输出了相应的数据记录，如图 7.7 所示。

```

connect success
result: {_id: 587a01f9b65e3707d477af80,
  title: 'node.js',
  author: 'node',
  content: 'node.js is great!',
  publishTime: 2017-01-14T10:48:25.386Z,
  __v: 0 }

```

图 7.7 mongoose 查询记录

在 `find()`方法中的第一个参数中可以传入筛选条件，以便更加精确地查找出需要查找的数据。现将 `find()`方法的代码修改为以下代码：

**【代码 7-5】**

```

01 Article.find({title:'node.js'},function(err, docs) {
02   if(err) {
03     console.log('error');
04     return;
05   }
06   console.log("result: " + docs);
07 });

```

运行这段代码同样可以查询出记录。

与 `find` 方法类似的还有 `findOne()`方法，`find()`方法是查询完所有符合要求的数据后返回结果，而 `findOne()`方法则是查询一条数据，返回的是查询得到的第一条数据。

在 `mongoose` 中可以直接在查询记录后修改记录的值，修改后直接调用保存即可。如下代码查询数据后直接修改数据的 `title` 值为 `javascript`。

**【代码 7-6】**

```

01 /*引入 mongoose 模块*/
02 const mongoose = require('mongoose');
03 /*定义 mongodb 地址*/
04 const uri = 'mongodb://localhost/article';
05 /*连接 mongodb*/
06 mongoose.connect(uri, function(err) {
07   if(err) {
08     console.log(err);

```

```

09      }
10  });
11 /*定义 Schema*/
12 const ArticleSchema = new mongoose.Schema({
13   title: String,
14   author: String,
15   content: String,
16   publishTime: Date
17 });
18 mongoose.model('Article', ArticleSchema);
19 const Article = mongoose.model('Article');
20 /*查询 mongodb*/
21 Article.find({title:'node.js'},function(err, docs) {
22   if(err) {
23     console.log('error');
24     return;
25   }
26 /*修改数据*/
27   docs[0].title = 'javascript';
28 /*保存修改后的数据*/
29   docs[0].save();
30   console.log("result: " + docs);
31 });

```

同样，在命令行中查询记录 MongoDB，可以发现原来这个文档中的 title 值已经被修改，如图 7.8 所示。

```

> db.articles.find()
{ "_id" : ObjectId("587a01f9b65e3707d477afb0"), "title" : "javascript", "author" : "node", "content" : "node.js is great!", "publishTime" : ISODate("2017-01-14T10:48:25.386Z"), "__v" : 0 }
>

```

图 7.8 mongoose 修改数据

类似于修改数据，删除 MongoDB 的文档也可以在查询出文档后直接调用 remove 方法。如下代码可以删除 articles 集合中的所有文档。

### 【代码 7-7】

```

01 /*引入 mongoose 模块*/
02 const
03 /*定义 mongodb 地址*/
04 const uri = 'mongodb://localhost/article';
05 /*连接 mongodb*/
06 mongoose.connect(uri, function(err) {
07   if(err) {
08     console.lgo(err);
09   }
10 });
11 /*定义 Schema*/
12 const ArticleSchema = new mongoose.Schema({

```

```

13     title: String,
14     author: String,
15     content: String,
16     publishTime: Date
17 });
18 mongoose.model('Article', ArticleSchema);
19 const Article = mongoose.model('Article');
20 /*查询 mongodb*/
21 Article.find({}, function(err, docs) {
22     if(err) {
23         console.log('error');
24         return;
25     }
26     if(docs) {
27 /*删除数据*/
28         docs.forEach(function(ele) {
29             ele.remove();
30         })
31     }
32 });

```



只有单个文档可以调用 `remove()`方法，因为 `find()`方法返回的是一个符合查询条件的所有文档组成的数组，所以这里调用数组的 `forEach()`方法逐个删除所有的文档。同样，在命令行中查询记录 MongoDB，可以发现原来 `articles` 这个集合的所有文档已经为空了。

以上的知识实现了使用 `mongoose` 对 MongoDB 数据库进行简单的增删改查。更多关于 `mongoose` 的使用，读者可以通过阅读 `mongoose` 的官方文档进行学习。

## 7.2 直接连接 MongoDB

在前面的 7.1 节中，提到了 `mongoose` 模块是基于 `node-mongodb-native` 开发的 MongoDB 的 Node.js 驱动，同样使用 `node-mongodb-native` 这个原生 MongoDB 驱动也可以对 MongoDB 进行相应的操作。`node-mongodb-native` 模块的 GitHub 地址是 <https://github.com/mongodb/node-mongodb-native>，官方网站为 <http://mongodb.github.io/node-mongodb-native/>，读者可以在官方网站查看相应的说明和文档进行学习。

### 7.2.1 使用 `node-mongodb-native` 连接 MongoDB

使用 `node-mongodb-native` 这个模块前需要通过 NPM 安装这个模块：

```
npm install mongodb
```

node-mongodb-native 通过 `connect()`方法传递一个 URI 地址，用来说明需要连接的 MongoDB 数据库。如下代码即和本地的 MongoDB 数据库 `student` 建立了连接。

### 【代码 7-8】

```
01 /*引入模块*/
02 var MongoClient = require('mongodb').MongoClient;
03 /*定义 mongodb 地址*/
04 var url = 'mongodb://localhost:27017/student';
05 /*连接 mongodb*/
06 MongoClient.connect(url, function(err, db) {
07     if(err) {
08         console.log('connect failed');
09         console.log(err);
10         return;
11     }
12     console.log('connect success!');
13 })
```

### 【代码说明】

因为 `mongoose` 是基于 `node-mongodb-native` 开发的，所以两者的 API 还是有相似的地方。运行以上这段代码，如果 MongoDB 运行正常，将会打印出“`connect success!`”字样。

## 7.2.2 使用 node-mongodb-native 操作 MongoDB

使用 `node-mongodb-native` 驱动需要注意：每次操作完 MongoDB 都应该调用 `close` 方法来关闭 MongoDB，否则会影响其他代码对 MongoDB 的操作。调用 `insertOne` 方法可以插入一条数据，如前面提到的一样，`node-mongodb-native` 插入的数据依旧是 json 格式，代码如下：

### 【代码 7-9】

```
01 /*引入模块*/
02 var MongoClient = require('mongodb').MongoClient;
03 var Db = require('mongodb').Db;
04 var server = require('mongodb').Server;
05 var studentDb = new Db('student', new server('localhost', '27017'));
06 /*定义数据*/
07 var student = {
08     id: '1101',
09     name: 'jack',
10     age: 12
11 };
12 /*打开数据库*/
13 studentDb.open(function(err, db) {
14     if(err) {
15         console.log('open err');
16         console.log(err);
17         return;
18     }
```

```

19 /*打开集合*/
20     db.collection('student', function(err, collection) {
21         if(err) {
22             console.log('collection error');
23             studentDb.close();
24             console.log(err);
25             return;
26         }
27     /*插入文档*/
28     collection.insertOne(student, function(err, doc) {
29     /*关闭数据库*/
30         studentDb.close();
31         if(err) {
32             console.log('document error');
33             consle.log(err);
34             return;
35         }
36         console.log(doc[0]);
37     });
38 });
39 });

```

**【代码说明】**

这段代码将一个名为 jack 的学生数据插入到 student 数据库下的 student 集合中，整个过程是打开数据库→打开集合→插入数据→关闭数据库。

运行这段代码，在 MongoDB 数据库运行正常的情况下将打印出 undefined。因为这是插入数据操作，并没有文档会被查询出来，所以 console.log(doc[0])语句打印出 undefined。同样，通过命令行工具可以查询出 student 数据库中的 student 集合已经存在这样一条记录，如图 7.9 所示。

```

> use student
switched to db student
> show collections
student
> db.student.find()
{ "_id" : ObjectId("587b94f71d8a2b258cb40ff2"), "id" : "1101", "name" :
"jack", "age" : 12 }

```

图 7.9 通过命令查看数据

调用 node-mongodb-native 提供的 findOne 方法也可以将这条数据查询出来，代码如下所示：

**【代码 7-10】**

```

01 /*引入模块*/
02 var MongoClient = require('mongodb').MongoClient;
03 var Db = require('mongodb').Db;
04 var server = require('mongodb').Server;
05 var studentDb = new Db('student', new server('localhost', '27017'));
06 /*打开数据库*/
07 studentDb.open(function(err, db) {
08     if(err) {
09         console.log('open err');

```

```

10         console.log(err);
11         return;
12     }
13     /*打开集合*/
14     db.collection('student', function(err, collection) {
15     /*出错则关闭数据库*/
16     if(err) {
17         console.log('collection error');
18         studentDb.close();
19         console.log(err);
20         return;
21     }
22     /*查找文档*/
23     collection.findOne({}, function(err, doc) {
24     /*关闭数据库*/
25         studentDb.close();
26         if(err) {
27             console.log('document error');
28             consle.log(err);
29             return;
30         }
31         console.log(doc);
32     });
33 });
34 });

```

**【代码说明】**

整个过程也是按照打开数据库→打开集合→查询数据→关闭数据库这个流程严格执行的。整段代码依旧严格遵循打开 MongoDB 数据后必须关闭的原则。

运行这段代码，在 MongoDB 数据库运行正常的情况下，将会打印出 jack 这条数据，如图 7.10 所示。

```
{
  _id: 587b94f71d8a2b258cb40ff2,
  id: '1101',
  name: 'jack',
  age: 12 }
```

图 7.10 调用 findOne()方法查询数据

node-mongodb-native 模块也支持一次插入多条数据和查询多条数据，只需要调用 insertMany() 和 find() 方法即可。其中，在 insertMany 中插入多条数据时，只要将这些数据组成一个数组传递给 insertMany 方法即可。如下代码就一次性插入了三条记录。

**【代码 7-11】**

```

01 /*引入模块*/
02 var MongoClient = require('mongodb').MongoClient;
03 var Db = require('mongodb').Db;
04 var server = require('mongodb').Server;
05 var studentDb = new Db('student', new server('localhost', '27017'));

```

```
06 /*定义数据*/
07 var student1 = {
08     id: 1201,
09     name: '张三',
10     age:13
11 };
12 var student2 = {
13     id: 1202,
14     name: '李四',
15     age:14
16 };
17 var student3 = {
18     id: 1203,
19     name: '王五',
20     age:10
21 };
22 /*打开数据库*/
23 studentDb.open(function(err, db) {
24     if(err) {
25         console.log('open err');
26         console.log(err);
27         return;
28     }
29     /*打开集合*/
30     db.collection('student', function(err, collection) {
31     /*出错则关闭数据库*/
32         if(err) {
33             console.log('collection error');
34             studentDb.close();
35             console.log(err);
36             return;
37         }
38     /*插入多条数据*/
39         collection.insertMany([student1, student2, student3], function(err,
doc) {
40             studentDb.close();
41             if(err) {
42                 console.log('document error');
43                 consle.log(err);
44                 return;
45             }
46             console.log('insert success');
47         });
48     });
49 });
```

### 【代码说明】

运行这段代码，在 MongoDB 运行正常的情况下，可以发现控制台打印出“insert success”字样。

调用 find()方法可以验证 MongoDB 是否真的存在刚刚插入的这些数据。使用 find 方法之

后需要调用 `toArray()`方法将这些数据转化为一个数组。以下代码可以查询出 `student` 数据库下 `student` 集合中所有的数据记录。

### 【代码 7-12】

```
01 /*引入模块*/
02 var MongoClient = require('mongodb').MongoClient;
03 var Db = require('mongodb').Db;
04 var server = require('mongodb').Server;
05 var studentDb = new Db('student', new server('localhost', '27017'));
06 /*定义数据*/
07 var student1 = {
08     id: 1201,
09     name: '张三',
10     age:13
11 };
12 var student2 = {
13     id: 1202,
14     name: '李四',
15     age:14
16 };
17 var student3 = {
18     id: 1203,
19     name: '王五',
20     age:10
21 };
22 /*打开数据库*/
23 studentDb.open(function(err, db) {
24     if(err) {
25         console.log('open err');
26         console.log(err);
27         return;
28     }
29 /*打开集合*/
30     db.collection('student', function(err, collection) {
31 /*出错则关闭数据库*/
32         if(err) {
33             console.log('collection error');
34             studentDb.close();
35             console.log(err);
36             return;
37         }
38 /*将查询记录转化为数组*/
39         collection.find().toArray(function(err, docs) {
40             studentDb.close();
41             if(err) {
42                 console.log('document error');
43                 console.log(err);
44                 return;
45         }
```

```

46         console.log(docs);
47     });
48   });
49 }

```

**【代码说明】**

运行这段代码，在 MongoDB 运行正常的情况下，可以发现所有的数据都已经组织成一个数组，刚刚的数据也在这个数组内，说明上面的插入操作和查询操作都是成功的，如图 7.11 所示。

```

[ { _id: 587b94f71d8a2b258cb40ff2,
  id: '1101',
  name: 'jack',
  age: 12 },
{ _id: 587b9ebe51c2fd36283cbe9f, id: 1201, name: '张三', age: 13 },
{ _id: 587b9ebe51c2fd36283cbea0, id: 1202, name: '李四', age: 14 },
{ _id: 587b9ebe51c2fd36283cbea1, id: 1203, name: '王五', age: 10 } ]

```

图 7.11 调用 find()方法查询数据

调用 node-mongodb-native 模块提供的 deleteOne()方法可以对数据进行删除，与 findOne()方法类似。delete()方法的第一个参数是查询条件，第二个参数是一个处理错误和结果的回调函数。如下代码可以删除最初插入的 jack 数据。

**【代码 7-13】**

```

01 /*引入模块*/
02 var MongoClient = require('mongodb').MongoClient;
03 var Db = require('mongodb').Db;
04 var server = require('mongodb').Server;
05 var studentDb = new Db('student', new server('localhost', '27017'));
06 /*打开数据库*/
07 studentDb.open(function(err, db) {
08     if(err) {
09         console.log('open err');
10         console.log(err);
11         return;
12     }
13 /*打开集合*/
14     db.collection('student', function(err, collection) {
15 /*出错则关闭数据库*/
16     if(err) {
17         console.log('collection error');
18         studentDb.close();
19         console.log(err);
20         return;
21     }
22 /*删除单个数据*/
23     collection.deleteOne({name:'jack'}, function(err, doc) {
24         studentDb.close();
25         if(err) {
26             console.log('delete failed');
27             console.log(err);

```

```

28         return;
29     }
30     console.log('delete success')
31   });
32 });
33 });

```

**【代码说明】**

运行这段代码，在 MongoDB 运行正常的情况下，可以发现控制台打印出了“delete success”字样。利用控制台工具查询 student 集合下的所有文档，同样可以发现 jack 这条数据已经被删除。

node-mongodb-native 模块的 updateOne()方法可以更改数据，与查询方法类似。updateOne()方法的第一个参数是查询条件，第二个参数是更改后的数据，第三个参数是一个处理错误和结果的回调函数。如下代码就可以将“张三”这条数据的名字改为“张四”。

**【代码 7-14】**

```

01 /*引入模块*/
02 var MongoClient = require('mongodb').MongoClient;
03 var Db = require('mongodb').Db;
04 var server = require('mongodb').Server;
05 var studentDb = new Db('student', new server('localhost', '27017'));
06 /*打开数据库*/
07 studentDb.open(function(err, db) {
08   if(err) {
09     console.log('open err');
10     console.log(err);
11     return;
12   }
13 /*打开集合*/
14   db.collection('student', function(err, collection) {
15     if(err) {
16       console.log('collection error');
17       studentDb.close();
18       console.log(err);
19       return;
20     }
21 /*更新数据*/
22   collection.updateOne({name:'张三'}, {$set:{name:'张四'}}, function(err,doc) {
23     studentDb.close();
24     if(err) {
25       console.log('update failed');
26       console.log(err);
27       return;
28     }
29     console.log('update success')
30   });
31 });
32 });

```

### 【代码说明】

运行这段代码，在 MongoDB 运行正常的情况下，可以发现控制台打印出了“update success”字样。利用控制台工具查询 student 集合下的所有文档，同样可以发现“张三”这条数据的名字已经被改为“张四”。

以上示例代码就已经实现了使用 node-mongodb-native 对 MongoDB 数据库进行简单的增删查改。更多关于 node-mongodb-native 的使用，读者可以通过阅读 node-mongodb-native 的官方文档进行学习。

这里需要提出的是，使用 mongoose 会相对简单一点，毕竟 mongoose 是基于 node-mongodb-native 开发的。如果读者对其感兴趣，可以学习一下 node-mongodb-native，对 mongoose 的使用很有帮助。另外，读者应该掌握 MongoDB 基本的增删查改操作，以便在学习过程中验证数据的操作是否成功。

## 7.3 连接 MySQL

MySQL 作为一种典型的关系型数据库在互联网中被大量使用。本节将使用 mysql 模块进行 MySQL 数据库的连接。

### 7.3.1 MySQL 介绍

MySQL 数据库由瑞典 MySQL AB 公司开发，目前属于 Oracle 公司。MySQL 采用双授权模式，分为商业版和社区版。MySQL 数据库凭借其体积小、速度快、总成本低等特点被广泛应用在 Web 开发中。经典的开源软件架构 LAMP 中的 M 便是指 MySQL。

MySQL 的官方网站是 <http://www.mysql.com/>。读者可以在社区版下载网址 <https://dev.mysql.com/downloads/mysql/> 中选择相应系统的版本（见图 7.12）。

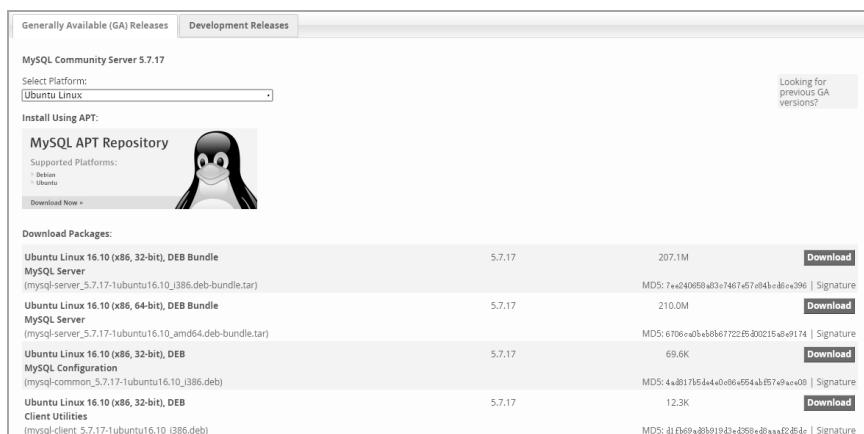


图 7.12 MySQL 下载页面

这里以 Windows 版本为例，安装过程大致如下：

(1) 将下载的 MySQL 软件按照常规软件的步骤安装即可。需要提醒的是，在安装过程中，可以设置 MySQL 服务的端口，默认为 3306，如图 7.13 所示。

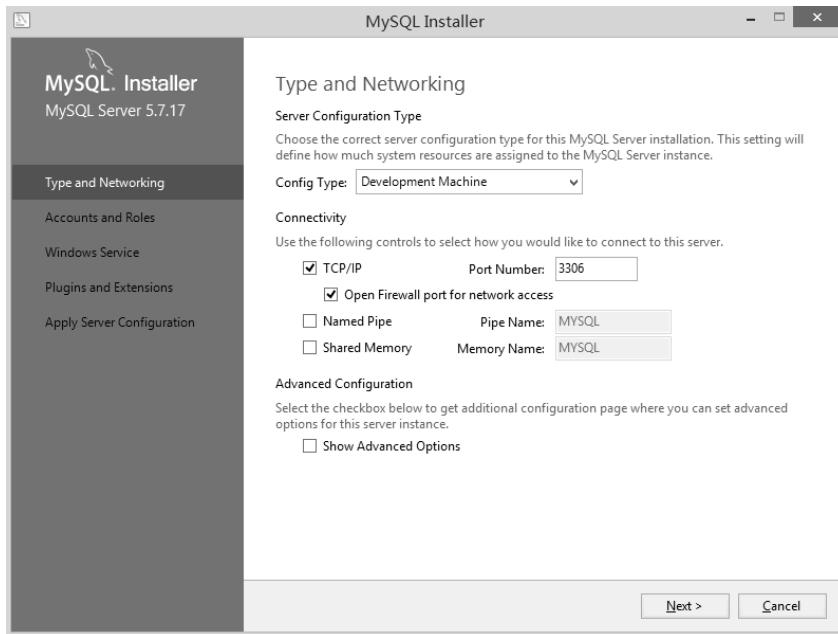


图 7.13 MySQL 设置端口

(2) 在安装过程中可以设置 root 用户的密码、添加用户，如图 7.14 所示。

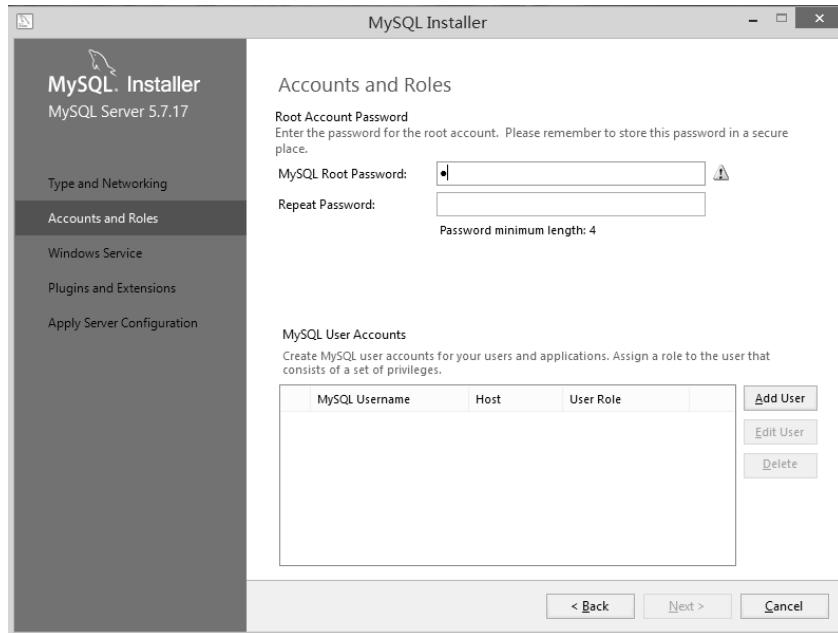


图 7.14 添加 MySQL 用户并设置密码

(3) 安装完成后，可以在控制台使用以下命令来启动 MySQL：

```
// net start commandline
net start mysql5.7
```



在这里 mysql5.7 是已经安装的带有版本号的 MySQL 软件名。

同样，也可以在 Windows 的服务中找到 MySQL 服务，然后进行启动、停止操作，如图 7.15 所示。



图 7.15 从 Windows 服务中启动 MySQL

(4) 启动 MySQL 后，通过以下命令可以进入 MySQL：

```
mysql -u root -p
```



root 是用户名，MySQL 自带 root 用户，读者可以设置自己的用户名再进入 MySQL。

(5) 输入命令后，紧接着会要求输入密码。输入用户的密码后，可以看到图 7.16 所示的界面，表示成功进入了 MySQL。

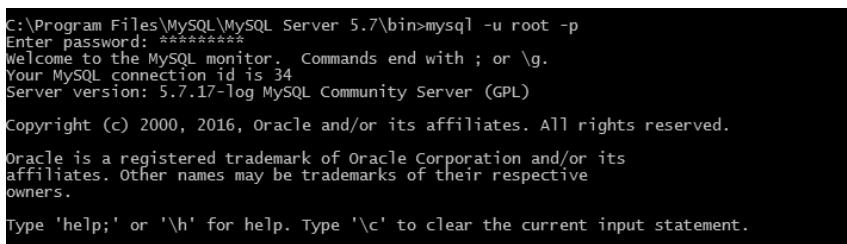


图 7.16 进入 MySQL

(6) 进入 MySQL 后，可以通过输入 help 或者 “\h” 来查看 MySQL 命令的帮助，如图

7.17 所示。

```
List of all MySQL commands:
Note that all text commands must be first on line and end with ';'
?
      ((?) Synonym for 'help'.
clear
      ((c) Clear the current input statement.
connect ((n) Reconnect to the server. Optional arguments are db and host.
delimiter ((d) Set statement delimiter.
ego
      ((G) Send command to mysql server, display result vertically.
exit
      ((q) Exit mysql. Same as quit.
go
      ((g) Send command to mysql server.
help
      ((h) Display this help.
notee
      ((t) Don't write into outfile.
print
      ((p) Print current command.
prompt
      ((R) Change your mysql prompt.
quit
      ((q) Quit mysql.
rehash
      ((#) Rebuild completion hash.
source ((.) Execute an SQL script file. Takes a file name as an argument.
status
      ((s) Get status information from the server.
tee
      ((\T) Set outfile [to_outfile]. Append everything into given outfile.
use
      ((u) Use another database. Takes database name as argument.
charset ((C) Switch to another charset. Might be needed for processing binlog with multi-byte charsets.
warnings ((W) Show warnings after every statement.
nowarning ((w) Don't show warnings after every statement.
resetconnection((x) Clean session context.
```

图 7.17 MySQL 的帮助

(7) 当我们不需要使用 MySQL 时, 可以通过 quit 命令退出 MySQL, 如图 7.18 所示。

```
mysql> quit
Bye
```

图 7.18 退出 MySQL

这里仅仅是对 MySQL 进行简单的介绍, 读者可以通过阅读相关的书籍学习更多知识。

### 7.3.2 Node.js 连接 MySQL

Node.js 连接 MySQL 使用的是 mysql 模块。mysql 模块的 GitHub 地址是 <https://github.com/mysqljs/mysql>, 从中可以查到官方文档。使用这个模块前需要通过 NPM 来安装:

```
npm install mysql
```

mysql 模块通过 createConnection()方法创建 MySQL 连接。如下代码即和本地的 MySQL 数据库建立了连接。

#### 【代码 7-15】

```
/*引入 mysql 模块*/
const mysql = require('mysql');

/*创建连接*/
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password : 'secret'
});

/*连接 mysql*/
connection.connect(function(err) {
/*连接出错的处理*/
if (err) {
```

```

        console.error('error connecting: ' + err.stack);
        return;
    }
    console.log('connected as id ' + connection.threadId);
});

```

**【代码说明】**

CreateConnection()方法用于创建连接，connection.connect()方法用于判断连接是否成功。

CreateConnection()方法用于接受一个 json 对象参数。json 对象主要使用的字段有：

- host：需要连接数据库地址，默认为 localhost。
- port：连接地址端口默认为 3306。
- user：连接 MySQL 时使用的用户名。
- password：用户名对应的密码。
- database：所需要连接的数据库的名称。

通过 end()方法可以正常地终止一个连接：

```

connection.end(function(err) {
    console.log(err);
})

```

当然，使用 destroy()方法也可以终止连接。该方法会立即终止底层套接字，不会触发更多的事件和回调函数。

```
connection.destroy()
```

### 7.3.3 Node.js 操作 MySQL

连接 MySQL 成功后，就需要通过 Node.js 来操作数据库了。mysql 模块提供了一个名为 query() 的方法，可以用来执行 SQL 语句，从而对 MySQL 数据库进行相应的操作。

假设我们连接的数据库有一个名为 data 的数据表，可以使用以下代码将这个 data 数据表的所有记录查询出来。

**【代码 7-16】**

```

/*引入 mysql 模块*/
const mysql = require('mysql');

/*创建连接*/
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password : 'secret',
    database : 'database'
});

/*连接 mysql*/
connection.connect(function(err) {

```

```

/*连接出错的处理*/
if (err) {
    console.error('error connecting: ' + err.stack);
    return;
}
console.log('connected as id ' + connection.threadId);
});

/*查询数据*/
connection.query('SELECT * FROM data', function(err, rows) {
    if(err) {
        console.log(err);
    } else {
        console.log(rows);
    }
});

```

运行这段代码就可以看到所有的记录被打印出来了。

上述代码中 `connection.query()` 方法的第一个参数是一条 SQL 语句，第二个参数是一个回调函数。回调函数中的第一个参数是 `err`，第二个参数是执行 SQL 语句后返回的记录。

`connection.query()` 方法还有一个 `paramInfo` 参数可选。当 SQL 语句中含有一些变量时，可以将“?”作为占位符放置在 SQL 语句中，通过 `paramInfo` 参数传递给 SQL 语句。

### 【代码 7-17】

```

/*引入 mysql 模块*/
const mysql = require('mysql');

/*创建连接*/
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password : 'secret',
    database : 'database'
});

/*连接 mysql*/
connection.connect(function(err) {
/*连接出错的处理*/
    if (err) {
        console.error('error connecting: ' + err.stack);
        return;
    }
    console.log('connected as id ' + connection.threadId);
});
const table = 'mytable';

/*查询数据*/
connection.query('SELECT * FROM ?', [table], function(err, rows) {
    if(err) {

```

```

        console.log(err);
    } else {
        console.log(rows);
    }
});
```

运行这段代码，同样可以从 mytable 数据表中取出所有的数据记录。

mysql 模块还提供一个 escape()方法，用来防止 SQL 注入攻击。SQL 注入攻击的本质就是黑客在提交给服务器的数据中带有 SQL 语句，试图欺骗服务器，让服务器运行自己的恶意 SQL 语句，因此调用 escape 方法处理用户提交的数据可以防止 SQL 注入攻击。

假设 userid 为用户提供的数据，可以先通过 connection.escape()方法处理一遍，之后再执行相关的 SQL 语句。

### 【代码 7-18】

```

/*引入 mysql 模块*/
const mysql = require('mysql');

/*创建连接*/
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password : 'secret',
    database : 'database'
});

/*连接 mysql*/
connection.connect(function(err) {
/*连接出错的处理*/
    if (err) {
        console.error('error connecting: ' + err.stack);
        return;
    }
    console.log('connected as id ' + connection.threadId);
});

/*定义 SQL 语句*/
let sql = 'SELECT * FROM users WHERE userid=' + connection.escape(userid);

/*执行 SQL 语句*/
connection.query(sql, function(err, rows) {
    if(err) {
        console.log(err);
    } else {
        console.log(rows);
    }
});
```

# 第 8 章

# Vue.js 数据、方法与生命周期

目前使用 Vue.js 技术的开发人员大都是前端开发人员。如今前端技术发展迅速，了解当前的主流前端开发技术是每个 Web 开发人员的必修课。

通过本章的学习可以：

- 了解 Vue.js 的数据与方法。
- 了解 Vue.js 生命周期的概念。
- 掌握通过 Vue.js 设计一个单页面应用。

## 8.1 Vue.js 数据

本节介绍 Vue.js 数据属性与实例属性方面的内容，Vue.js 针对数据与视图进行了特殊设计，并内置了一系列属性实现数据与视图的交互响应功能。

### 8.1.1 Vue.js 数据同步

对于 Vue.js 框架编程而言，当创建一个新的 Vue 实例对象时，其会将数据（data）对象中的所有的 property 属性加入到 Vue.js 框架的响应式系统当中去。该操作带来的最直接效果就是，当这些 property 属性值发生改变时，视图（View）将会随之发生“响应”——也就是同时更新为新匹配的属性值。以上关于 Vue.js 数据的描述听起来会感觉比较晦涩，下面我们通过具体实例进行解释。

（1）在页面中定义一个层（<div>）元素，用于显示 Vue 组件定义的对象，代码如下：

【代码 8-1】（详见源代码 vuedata 目录中的 vuedata.html 文件）

```
01 <div id="id-div-number">
02   {{ dNum }}
03 </div>
```

**【代码说明】**

- 第 01 行代码中，定义了<div>元素的 id 属性（"id-div-number"）。
- 第 02 行代码中，通过 Vue.js 框架的插值模板语法（{{ }}），引用了一个对象（dNum）。

（2）通过 js 脚本代码定义一个对象（oNum），在该对象内定义一个属性（n），并进行初始化操作，代码如下：

**【代码 8-2】**（详见源代码 vuedata 目录中的 vuedata.html 文件）

```
01 var oNum = {
02     n: 1
03 };
```

（3）通过 Vue 脚本代码定义一个 Vue 对象（vm），将对象（oNum）所定义的数据写进该 Vue 对象（vm），代码如下：

**【代码 8-3】**（详见源代码 vuedata 目录中的 vuedata.html 文件）

```
01 var vm = new Vue({
02     el: '#id-div-number',
03     data: {
04         dNum: oNum
05     }
06 })
```

**【代码说明】**

- 第 01~06 行代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。同时，这段代码创建了 Vue 对象的入口，并将该对象所定义的内容渲染到页面中对应的 DOM 元素中。具体说明如下：
  - 第 02 行代码中，通过“el”属性绑定 DOM 元素（id = 'id-div-number'），该 DOM 元素定义在【代码 8-1】中。
  - 第 03~05 行代码中，通过“data”属性绑定数据操作。其中，在第 04 行代码中定义了一个 property 属性（dNum），其将该属性值初始化为【代码 8-2】中定义的对象（oNum）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuedata.html 页面，效果如图 8.1 所示。

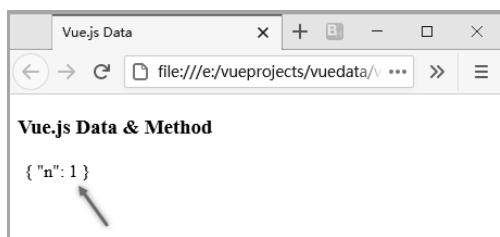


图 8.1 测试 Vue.js 数据对象（一）

如图 8.1 中的箭头所示，【代码 8-3】中定义的 Vue 对象成功渲染到【代码 8-1】中定义的页面 DOM 元素中。

这里需要特别关注的是，Vue 对象中 data 属性的 property 属性的“响应式”特性。也就是当这些数据发生改变时，页面视图会随之进行重新渲染。为了验证这个 Vue.js 数据的特性，我们在页面中添加一个文本输入框和一个关联按钮，通过人工输入修改 property 属性（dNum）的值，并观察页面视图的变化。代码如下：

#### 【代码 8-4】（详见源代码 vuedata 目录中的 vuedata.html 文件）

```
01 <div>
02   ViewModel:
03     <input type="text" id='id-input-text-vm' value=''/>
04     <input
05       type="button"
06       id='id-input-btn-vm'
07       value='Set VM'
08       onclick="onBtnVMClk(this.id)" />
09 </div>
```

#### 【代码说明】

- 第 03 行代码中，通过<input type="text">元素定义了一个文本输入框。
- 第 04~08 行代码中，<input type="button">元素定义了一个按钮。

然后，定义上面按钮控件（id='id-input-btn-vm'）的 onclick 方法（onBtnVMClk()），以修改 property 属性（dNum）的值，代码如下：

#### 【代码 8-5】（详见源代码 vuedata 目录中的 vuedata.html 文件）

```
01 function onBtnVMClk(thisid) {
02   let vData = document.getElementById('id-input-text-vm').value;
03   console.log("onBtnVMClk: " + vData);
04   vm.dNum.n = parseInt(vData);
05 }
```

#### 【代码说明】

- 第 04 行代码中，将从文本输入框中获取的用户输入值，赋值给 vm 对象的 property 属性（dNum）。

接下来，通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuedata.html 页面，效果如图 8.2 所示。

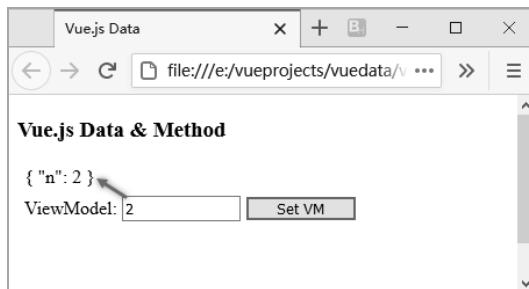


图 8.2 测试 Vue.js 数据对象（二）

如图 8.2 中的箭头所示，【代码 8-5】通过获取人工输入的数值并赋值给 property 属性（dNum）后，页面视图中的 DOM 元素的内容也同步进行了更新。注意，这个过程是通过修改 vm 对象实现的。那如果直接修改【代码 8-2】中定义的对象（oNum）呢？

为了验证这个 Vue.js 数据的特性，我们在页面中再添加一个文本输入框和一个关联按钮，通过人工输入修改对象（oNum）的值，并观察页面视图的变化。代码如下：

#### 【代码 8-6】（详见源代码 vuedata 目录中的 vuedata.html 文件）

```

01 <div>
02   Data:
03     <input type="text" id='id-input-text-data' value=''/>
04     <input
05       type="button"
06       id='id-input-btn-data'
07       value='Set Data'
08       onclick="onBtnDataClk(this.id)" />
09 </div>

```

#### 【代码说明】

- 第 03 行代码中，通过<input type="text">元素定义了一个文本输入框。
- 第 04~08 行代码中，<input type="button">元素定义了一个按钮。

然后，定义上面按钮控件（id='id-input-btn-data'）的 onclick 方法（onBtnDataClk()），以修改对象（oNum）的值，代码如下：

#### 【代码 8-7】（详见源代码 vuedata 目录中的 vuedata.html 文件）

```

01 function onBtnDataClk(thisid) {
02   let vData = document.getElementById('id-input-text-data').value;
03   console.log("onBtnDataClk: " + vData);
04   oNum.n = parseInt(vData);
05 }

```

#### 【代码说明】

- 第 04 行代码中，将从文本输入框中获取的用户输入值，赋值给对象（oNum）的属性（n）。

下面我们还是通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuedata.html 页面，

效果如图 8.3 所示。

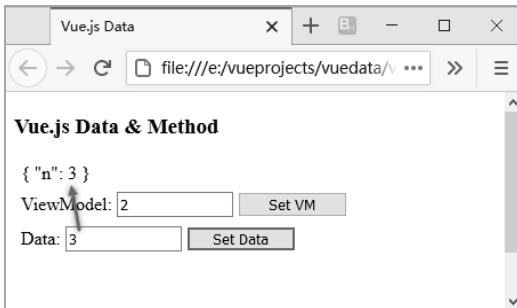


图 8.3 测试 Vue.js 数据对象（三）

如图 8.3 中的箭头所示，【代码 8-7】通过获取人工输入的数值并赋值给对象（oNum）的属性（n）后，页面视图中的 DOM 元素的内容也同步进行了更新。



只有当 Vue 对象实例被创建时，就已经存在于 data 属性中的 property 属性才是响应式的。如果在晚些时候才需要使用某个 property 属性，那么还是需要在一开始初始化时定义好该 property 属性。即使一开始该 property 属性为空或不存在，那么也需要设置一些初始值（比如：空字符串）。

## 8.1.2 Vue.js 数据冻结

Vue.js 数据“同步更新”的功能很实用，页面渲染效果也很惊艳。不过，我们不总是需要全部数据都保持同步更新，那么该如何操作呢？Vue.js 框架为数据对象定义了“冻结”方法，可以实现阻止 property 属性同步更新的功能。

这个方法就是由 Object 对象所提供的 freeze() 方法，通过在一个对象上使用 freeze() 方法，就会阻止修改现有的 property 属性，也就意味着 Vue.js 框架的视图响应系统无法追踪 property 属性的变化。下面通过一个具体的代码实例进行详细介绍。

（1）在页面中定义一个层（<div>）元素，用于显示 Vue 组件定义的对象，代码如下：

**【代码 8-8】**（详见源代码 vuedata 目录中的 vuedata.html 文件）

```
01 <div id="id-div-number">
02   <span>{{ dNumA }}</span><br>
03   <span>{{ dNumB }}</span><br>
04 </div>
```

**【代码说明】**

- 第 02 行和第 03 行代码中，分别通过 Vue.js 框架的插值模板语法（{{ }}），引用了两个对象（dNumA 和 dNumB）。

（2）通过 JS 脚本代码定义两个对象（oNumA 和 oNumB），分别在两个对象内定义一个属性（a 和 b），并进行初始化操作，代码如下：

**【代码 8-9】**（详见源代码 vuedata 目录中的 vuedata.html 文件）

```
01 var oNumA = {
02     a: 1
03 };
04 var oNumB = {
05     b: 1
06 };
```

**【代码说明】**

我们的设计想法是，通过 `Object.freeze()` 方法冻结其中一个对象（`oNumB`），这样就可以与另一个对象（`oNumA`）进行对比参照。

（3）通过 Vue 脚本代码定义一个 Vue 对象（`vm`），将对象（`oNumA` 和 `oNumB`）所定义的数据写进该 Vue 对象（`vm`），代码如下：

**【代码 8-10】**（详见源代码 vuedata 目录中的 vuedata.html 文件）

```
01 var vm = new Vue({
02     el: '#id-div-number',
03     data: {
04         dNumA: oNumA,
05         dNumB: oNumB,
06     }
07 })
```

**【代码说明】**

- 第 03~06 行代码中，通过“`data`”属性进行绑定数据操作。具体说明如下：
  - 第 04 行代码中定义了一个 `property` 属性（`dNumA`），它将该属性值初始化为【代码 8-8】中定义的对象（`oNumA`）。
  - 第 05 行代码中定义了一个 `property` 属性（`dNumB`），它将该属性值初始化为【代码 8-8】中定义的对象（`oNumB`）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 `vuedata.html` 页面，效果如图 8.4 所示。



图 8.4 测试 Vue.js 数据冻结（一）

如图 8.4 所示，【代码 8-10】中定义的 Vue 对象成功渲染到【代码 8-8】中定义的页面 DOM 元素中。

(4) 然后是本例中关键的一步，通过 `Object.freeze()` 方法冻结其中对象（`dNumB`），代码如下：

**【代码 8-11】**（详见源代码 `vuedata` 目录中的 `vuedata.html` 文件）

```
01 Object.freeze(oNumB);
```

为了验证上面这个 `Vue.js` 数据冻结的特性，我们在页面中添加相应的文本输入框和关联按钮，通过人工输入修改 `property` 属性的值，并观察页面视图的变化。代码如下：

**【代码 8-12】**（详见源代码 `vuedata` 目录中的 `vuedata.html` 文件）

```
01 <div>
02     ViewModel:
03         <input type="text" id='id-input-text-vm' value='' />
04         <input
05             type="button"
06             id='id-input-btn-vm'
07             value='Set VM'
08             onclick="onBtnVMClk(this.id)" />
09     </div>
10     <div>
11         Data:
12         <input type="text" id='id-input-text-data' value='' />
13         <input
14             type="button"
15             id='id-input-btn-data'
16             value='Set Data'
17             onclick="onBtnDataClk(this.id)" />
18     </div>
19     <script>
20         function onBtnVMClk(thisid) {
21             let vData = document.getElementById('id-input-text-vm').value;
22             console.log("onBtnVMClk: " + vData);
23             vm.dNumA.a = parseInt(vData);
24             vm.dNumB.b = parseInt(vData);
25         }
26         function onBtnDataClk(thisid) {
27             let vData = document.getElementById('id-input-text-data').value;
28             console.log("onBtnDataClk: " + vData);
29             oNumA.a = parseInt(vData);
30             oNumB.b = parseInt(vData);
31         }
32     </script>
```

**【代码说明】**

- 第 23 行和第 24 行代码中，将从文本输入框中获取的用户输入值，分别赋值给 `vm` 对

象的 property 属性 (dNumA 和 dNumB)。

- 第 29 行和第 30 行代码中，将从文本输入框中获取的用户输入值，分别赋值给对象 (oNumA 和 oNumB) 的属性 (a 和 b)。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuedata.html 页面，效果如图 8.5 和图 8.6 所示。

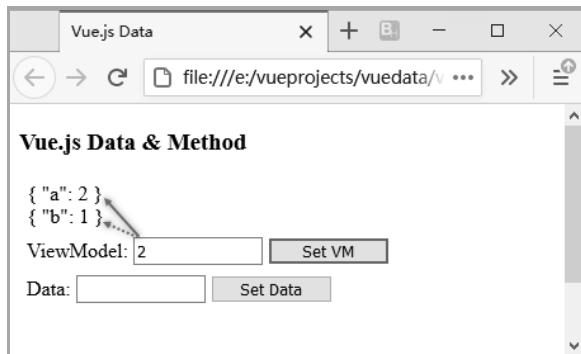


图 8.5 测试 Vue.js 数据冻结 (二)

如图 8.5 中的箭头(实线)所示，通过获取人工输入的数值并赋值给 property 属性 (dNumA) 后，页面视图中的 DOM 元素的内容也同步进行了更新。而图 8.5 中的箭头 (虚线) 表示，通过获取人工输入的数值并赋值给 property 属性 (dNumB) 后，页面视图中的 DOM 元素的内容没有进行同步更新，这就表明【代码 8-11】中的数据冻结操作生效了。

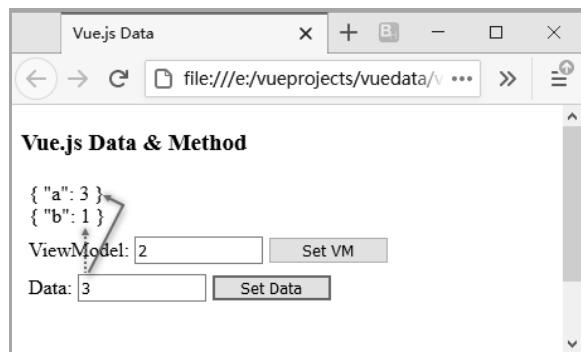


图 8.6 测试 Vue.js 数据冻结 (三)

同样如图 8.6 中的箭头 (实线和虚线) 所示，通过获取人工输入的数值并赋值给对象 (oNumA) 的属性 (a) 和对象 (oNumB) 的属性 (b) 后，页面视图中的 DOM 元素 (对应 oNumA 对象) 的内容同样也同步进行了更新，而 DOM 元素 (对应 oNumB 对象) 的内容没有进行同步更新，同样表明【代码 8-11】中的数据冻结操作生效了。

### 8.1.3 Vue.js 实例 property 属性

除了前面介绍的 Vue 数据 property 属性之外，Vue.js 框架还定义了一个非常有用的“Vue 实例 property 属性”的概念。需要注意的是，在使用该功能时必须要加上前缀“\$”符号，主

要是便于与用户定义的数据 property 属性进行区分。

首先，我们介绍一下“Vue 实例 property 属性”的内容。为了便于理解这个“Vue 实例 property 属性”的术语，我们通过具体实例进行解释。请看下面的代码。

**【代码 8-13】**（详见源代码 vuedata 目录中的 vuedata.html 文件）

```

01 <div id="id-div-number">
02   {{ dNum }}
03 </div>
04 <div>
05   Test:
06   <input
07     type="button"
08     id='id-input-btn-test'
09     value='Test Method'
10     onclick="onBtnTestClk(this.id)" />
11 </div>
12 <script>
13   var oNum = {
14     n: 1
15   };
16   var vm = new Vue({
17     el: '#id-div-number',
18     data: {
19       dNum: oNum
20     }
21   })
22   function onBtnTestClk(thisid) {
23     console.log(vm.$data.dNum.n);
24     console.log(vm.$el);
25   }
26 </script>
```

### 【代码说明】

- 第 23 行代码中，通过 vm 对象实例 property 属性（\$data）的引用，代理了对 vm 对象中 data 属性中 property 属性（dNum）的访问。这里，通过命令行控制台输出了“vm.\$data.dNum.n”的数值。
- 第 24 行代码中，通过 vm 对象实例 property 属性（\$el）的引用，代理了对 vm 对象中 el 属性 DOM 元素的访问。这里，通过命令行控制台输出了 vm.\$el 代表的内容。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuedata.html 页面，效果如图 8.7 所示。

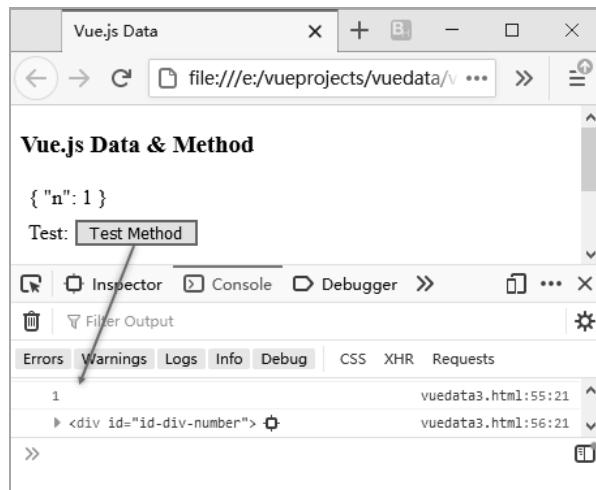


图 8.7 Vue 实例 property 属性

如图 8.7 中的箭头所示，在浏览器的命令行控制台中，【代码 8-13】的第 23 行代码输出了数值“1”，第 24 行代码输出了 DOM 元素。

现在，估计读者已经能大概理解“Vue 实例 property 属性”的含义了。而 Vue.js 框架定义这个概念的目的，就是为了帮助设计人员以更简洁的方式操作 Vue 对象中定义的各个字段属性。

接下来，我们在【代码 8-13】的基础上稍作修改，看一下如何实际使用“Vue 实例 property 属性”进行操作。请看下面的代码。

#### 【代码 8-14】（详见源代码 vuedata 目录中的 vuedata.html 文件）

```

01 <div id="id-div-number">
02   {{ dNum }}
03 </div>
04 <div>
05   Test:
06   <input
07     type="button"
08     id='id-input-btn-test'
09     value='Test Method'
10     onclick="onBtnTestClk(this.id)" />
11 </div>
12 <script>
13   var oNum = {
14     n: 1
15   };
16   var vm = new Vue({
17     el: '#id-div-number',
18     data: {
19       dNum: oNum
20     }
21   })

```

```

22     function onBtnTestClk(thisid) {
23         let divText = vm.$el.innerText;
24         console.log(divText);
25         vm.$data.dNum.n += 1;
26     }
27 </script>

```

### 【代码说明】

- 第 23~24 行代码中，通过 vm 对象实例 property (\$el) 的引用，获取了其 DOM 元素的 innerText 属性内容，并将该内容输出到命令行控制台中。
- 第 25 行代码中，通过 vm 对象实例 property (\$data) 的引用，代理了对其 data 属性中 property 属性 (dNum) 的访问，并对其 n 属性值进行了“+1”操作。

下面，通过 VS Code 开发工具启动 FireFox 浏览器运行测试 vuedata.html 页面，效果如图 8.8 所示。在浏览器的命令行控制台中，【代码 8-14】中的第 24 行代码输出了 DOM 元素的 innerText 属性内容，第 25 行代码输出了 n 属性经过“+1”操作后的数值。

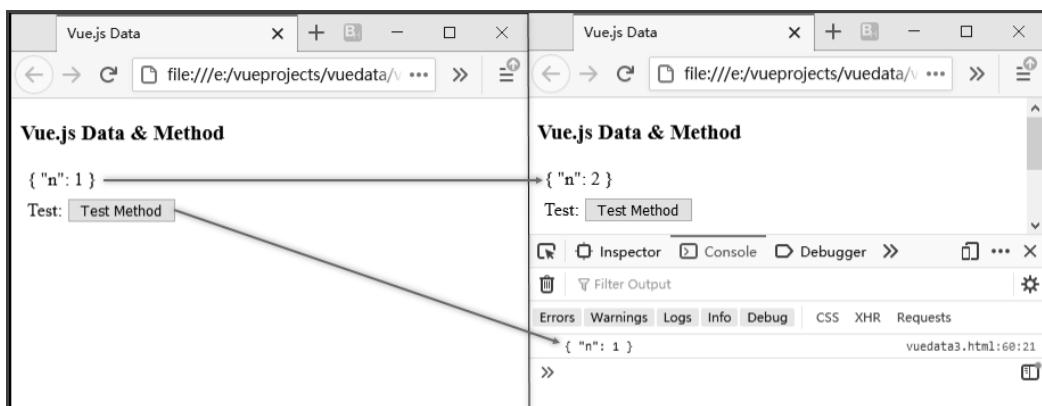


图 8.8 Vue 实例 property 属性应用

## 8.2 Vue.js 方法

本节介绍 Vue.js 实例方法的相关内容。Vue.js 实例方法包含了几大类，这里先介绍基于数据和基于事件这两类。

### 8.2.1 观察属性方法

Vue.js 框架设计了一个\$watch 方法，用于“观察”Vue 实例上的属性是否发生变化。这里的属性变化，具体可以是一个属性表达式的变化，也可以是稍微复杂的一个函数计算结果的变化。如果\$watch 方法观察到了变化，就会通过一个回调函数得到两个参数，分别表示变化后

的新值和变化前的旧值。

下面是关于\$watch 方法的基本语法格式：

```
语法: vm.$watch(expOrFn, callback, [options])>
参数说明:
{string | Function} expOrFn
{Function | Object} callback
返回值: {Function} unwatch // 返回一个取消观察函数, 用来停止触发回调
```

关于这个\$watch 方法如何实现“观察”Vue 实例上的属性变化，我们还是通过具体的代码实例进行讲解。请看下面这个“观察”计数器变化的代码。

**【代码 8-15】**（详见源代码 vuemethod 目录中的 vuewatch.html 文件）

```
01 <div id="id-div-counter">
02     Current counter is <b>{{ counter }}</b>.
03 </div>
04 <div>
05     Click "Start Watch" to start watch & add counter:
06     <input
07         type="button"
08         id='id-btn-start-watch'
09         value='Start Watch'
10         onclick="onBtnStartWatch(this.id)"/>
11     Click "Cancel Watch" to stop watch:
12     <input
13         type="button"
14         id='id-btn-cancel-watch'
15         value='Cancel Watch'
16         onclick="onBtnCancelWatch(this.id)"/>
17 </div>
18 <script>
19     // define Vue entry
20     var vm = new Vue({
21         el: '#id-div-counter',
22         data: {
23             counter: 1
24         }
25     })
26     // define watch on counter
27     var unwatch = vm.$watch('counter', function(newVal, oldVal) {
28         console.log("counter from " + oldVal + " turns to " + newVal + ".");
29     })
30     // start watch
31     function onBtnStartWatch(thisid) {
32         vm.$data.counter += 1;
33         console.log('counter = ' + vm.$data.counter);
34     }
35     // cancel watch
36     function onBtnCancelWatch(thisid) {
37         unwatch();
```

```
38      }
39  </script>
```

### 【代码说明】

- 第 01~03 行代码中，在页面中通过<div>元素定义了一个层，并定义了其 id 属性值（"id-div-counter"）。在第 02 行代码中，通过 Vue.js 框架的插值模板语法（{{ }}）引用了一个对象（counter），实现了一个计数器的展示功能。
- 第 20~25 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。同时，这段代码创建了 Vue 对象的入口，并将该对象所定义的内容渲染到页面中对应的 DOM 元素中。具体说明如下：
  - 第 21 行代码中，通过 el 属性绑定 DOM 元素（"id-div-counter"）。
  - 第 22~24 行代码中，通过 data 属性进行绑定数据操作。其中，在第 23 行代码中定义了一个计数器属性（counter），并初始化为数值 1。该计数器属性（counter）对应第 02 行代码引用的对象（counter），实现了页面数据同步渲染的功能。
- 第 27~29 行的脚本代码定义的就是 vm.\$watch()“观察”函数，具体说明如下：
  - 第 27 行代码中，指定的观察对象就是计数器（counter）。
  - 第 27 行代码中，回调函数定义了两个参数（newVal, oldVal），分别表示计数器对象（counter）变化后和变化前的值。
  - 第 28 行代码中，将两个参数（newVal, oldVal）的调试信息输出到命令行控制台中显示。
  - 另外，第 27 行代码中 vm.\$watch() 函数的返回值（unwatch），定义的就是取消观察函数，用来停止触发回调。
- 第 31~34 行代码中定义的 onBtnStartWatch() 函数，实现了第 06~10 行代码定义的<input>控件的单击事件方法。其中，第 32 行代码通过对 vm 对象实例 property(\$data) 的引用，将计数器对象（counter）的数值进行累加（+1）。
- 第 36~38 行代码中定义的 onBtnCancelWatch() 函数，实现了第 12~16 行代码定义的<input>控件的单击事件方法。其中，第 37 行代码通过调用 unwatch() 方法，实现了取消观察函数、并停止触发回调的操作。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuewatch.html 页面，页面初始效果如图 8.9 所示。

如图 8.9 中的箭头所示，页面中显示了计数器（counter）的初始值（1）。然后，我们尝试单击“Start Watch”按钮，页面更新效果如图 8.10 所示。

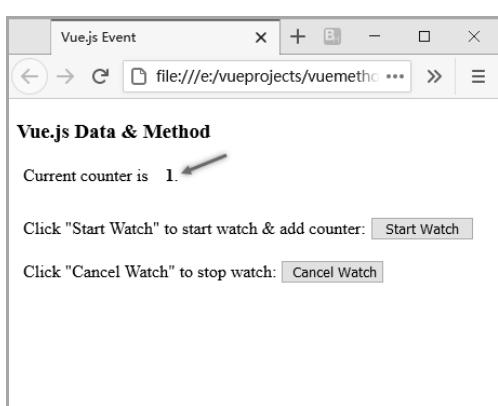


图 8.9 Vue 实例\$watch 方法（一）

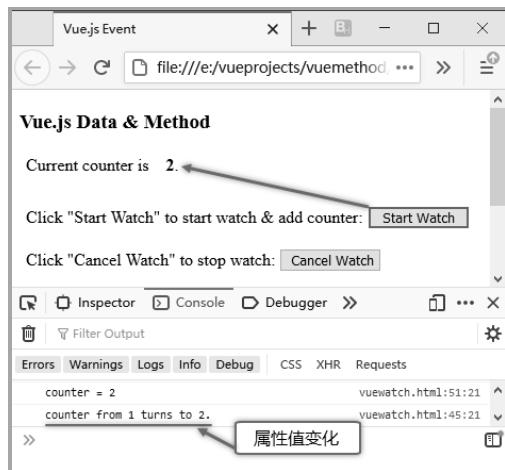


图 8.10 Vue 实例\$watch 方法（二）

如图 8.10 中的箭头和标识所示，在单击“Start Watch”按钮后，计数器（counter）的数值从 1 变为 2。同时，命令行控制台中也同步输出了体现计数器（counter）旧值到新值变化的调试信息。我们可以进行多次尝试，观察浏览器页面的变化更新以及命令行控制台跟踪的调试信息，如图 8.11 所示。

最后，尝试单击“Cancel Watch”按钮，取消观察函数及其回调函数。然后，再进行单击“Start Watch”按钮的测试，效果如图 8.12 所示。

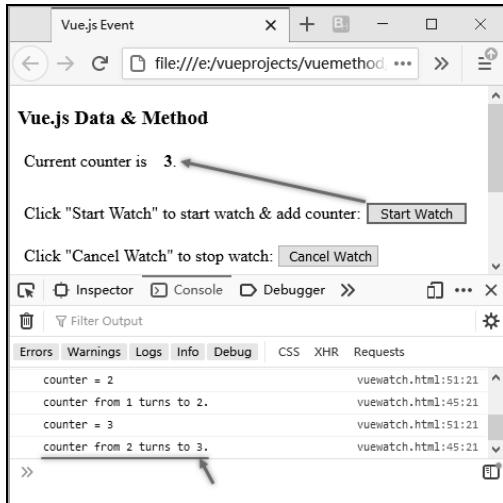


图 8.11 Vue 实例\$watch 方法（三）

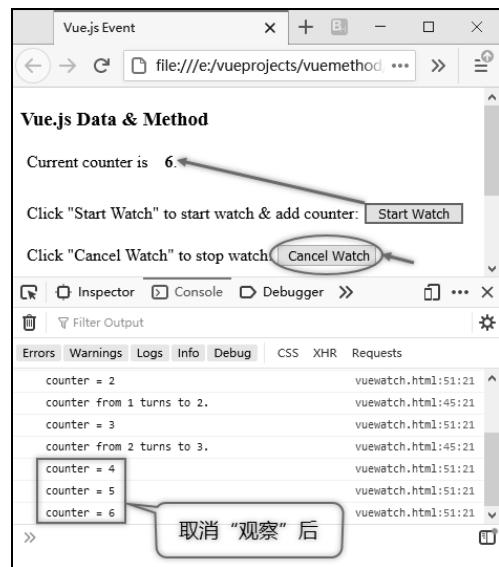


图 8.12 Vue 实例 unwatch 方法

如图 8.12 中的箭头和标识所示，在单击“Cancel Watch”按钮取消“观察”函数后，每次单击“Start Watch”按钮后，页面视图中的数值会同步增加，但命令行控制台中就不再同步输出体现计数器（counter）从旧值到新值变化的调试信息了。

在【代码 8-15】中“观察”的是一个对象表达式，根据\$watch 方法的语法描述，还可以

“观察”一个函数方法，这种情况适用于较复杂的场景。下面，我们还是通过具体的代码实例进行讲解，请看下面这个“观察”计算算术和变化的代码：

【代码 8-16】（详见源代码 vuemethod 目录中的 vuewatchfunc.html 文件）

```

01 <div id="id-div-sum">
02   Expression : {{a}} + {{b}} = {{sum}}
03 </div>
04 <div>
05   Click "Start Watch" to start watch sum:
06   <input
07     type="button"
08     id='id-btn-start-watch'
09     value='Start Watch'
10     onclick="onBtnStartWatch(this.id)" />
11   Click "Cancel Watch" to stop watch:
12   <input
13     type="button"
14     id='id-btn-cancel-watch'
15     value='Cancel Watch'
16     onclick="onBtnCancelWatch(this.id)" />
17 </div>
18 <script>
19   // define Vue entry
20   var vm = new Vue({
21     el: '#id-div-sum',
22     data: {
23       a: 1,
24       b: 1,
25       sum: 2
26     }
27   })
28   // define watch on a & b
29   var unwatch = vm.$watch(function() {
30     return this.a + this.b;
31   }, function(newVal, oldVal) {
32     console.log("sum from " + oldVal + " turns to " + newVal + ".");
33   })
34   // start watch
35   function onBtnStartWatch(thisid) {
36     vm.$data.a = Math.round(Math.random() * 100);
37     console.log('a = ' + vm.$data.a);
38     vm.$data.b = Math.round(Math.random() * 100);
39     console.log('b = ' + vm.$data.b);
40     vm.$data.sum = vm.$data.a + vm.$data.b;
41     console.log('sum = ' + vm.$data.sum);
42   }
43   // cancel watch
44   function onBtnCancelWatch(thisid) {
45     unwatch();

```

```
46      }
47  </script>
```

### 【代码说明】

- 第 01~03 行代码中，在页面中通过<div>元素定义了一个层，并定义了其 id 属性值 ("id-div-sum")。其中在第 02 行代码中，通过 Vue.js 框架的插值模板语法 ({{ }}) 引用了一个表达式 (a + b = sum)，实现了一个计算算术和的展示功能。
- 第 20~27 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象 (vm)。同时，这段代码创建了 Vue 对象的入口，并将该对象所定义的内容渲染到页面中对应的 DOM 元素中。具体说明如下：
  - 第 21 行代码中，通过 el 属性绑定 DOM 元素 ("id-div-sum")。
  - 第 22~26 行代码中，通过 data 属性进行绑定数据操作。其中，在第 23、24 行代码分别定义了两个加数属性 (a 和 b)，并均初始化为数值 1。第 25 行代码定义了算术和属性 (sum)，并初始化为计算结果数值 2。这三个属性 (a、b、sum) 对应第 02 行代码引用的对象 (a、b、sum)，实现了页面数据同步渲染的功能。
- 第 29~33 行的脚本代码定义的就是 vm.\$watch() “观察” 函数，具体说明如下：
  - 第 29~31 行代码中，指定的观察对象是一个自定义函数，该函数返回两个加数属性 (a 和 b) 的算术和，实际对应的是属性 (sum)。
  - 第 31~33 行代码中，回调函数定义了两个参数 (newVal, oldVal)，分别表示算术和属性 (sum) 变化后和变化前的值。第 32 行代码中，将两个参数 (newVal, oldVal) 的调试信息输出到命令行控制台中显示。
  - 另外，第 29 行代码中 vm.\$watch() 函数的返回值 (unwatch)，定义的就是取消观察函数，用来停止触发回调。
- 第 35~42 行代码中定义的 onBtnStartWatch() 函数，实现了第 06~10 行代码定义的<input>控件的单击事件方法，具体说明如下：
  - 第 36 行和第 38 行代码通过对 vm 对象实例 property (\$data) 的引用，分别将两个加数属性 (a 和 b) 重新赋值为一个随机自然数 (0~100)。
  - 第 40 行代码通过对 vm 对象实例 property (\$data) 的引用，将两个加数属性 (a 和 b) 的算术和赋值给属性 (sum)。
- 第 44~46 行代码中定义的 onBtnCancelWatch() 函数，实现了第 12~16 行代码定义的<input>控件的单击事件方法。其中，第 45 行代码通过调用 unwatch() 方法，实现了取消观察函数，并停止触发回调的操作。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuewatch.html 页面，页面初始效果如图 8.13 所示。

如图 8.13 中的箭头所示，页面中显示了计算算术和的表达式初始值 (1+1=2)。然后，尝试单击“Start Watch”按钮，页面更新效果如图 8.14 所示。

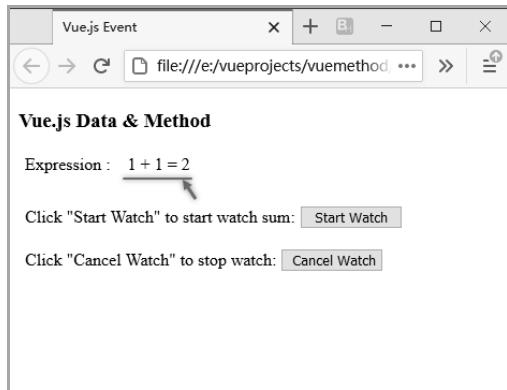


图 8.13 Vue 实例\$watch 方法——“观察”函数（一）

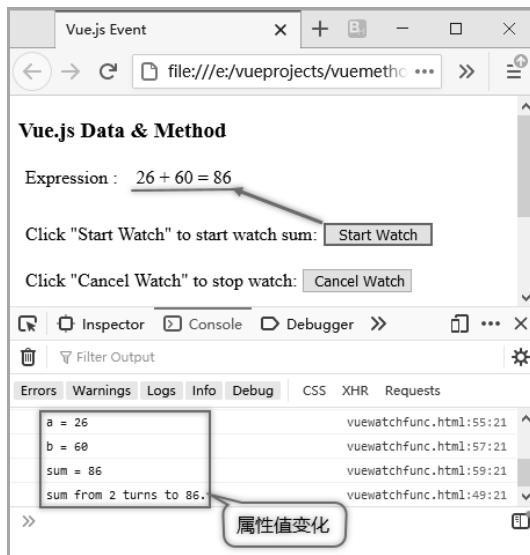


图 8.14 Vue 实例\$watch 方法——“观察”函数（二）

如图 8.14 中的箭头和标识所示，在单击“Start Watch”按钮后，算术表达式从 $(1+1=2)$ 变化为 $(26+60=86)$ 。同时，命令行控制台中也同步输出了算术和 $(sum)$ 从旧值变化到新值的调试信息。

最后，尝试单击一下“Cancel Watch”按钮，取消观察函数及其回调函数。然后，再进行单击“Start Watch”按钮的测试，效果如图 8.15 所示。

如图 8.15 中的箭头和标识所示，在单击“Cancel Watch”按钮取消“观察”函数后，每次单击“Start Watch”按钮后，页面视图中的算术表达式会同步更新，但命令行控制台中就不再同步输出体现算术和 $(sum)$ 从旧值到新值变化的调试信息了。

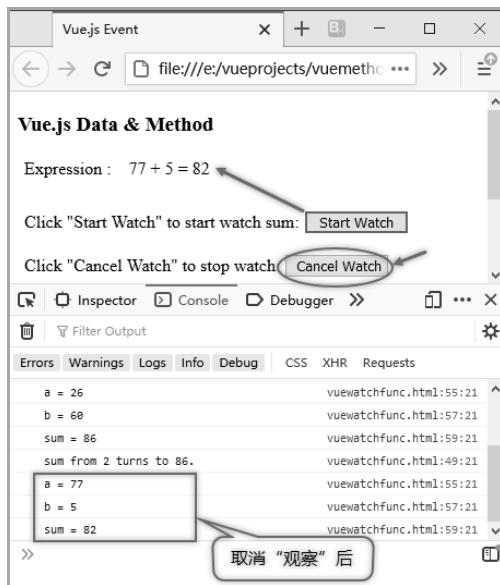


图 8.15 Vue 实例\$watch 方法——“观察”函数（三）

## 8.2.2 事件触发方法

Vue.js 框架设计了一个\$emit 事件触发方法，用于触发 Vue 实例上定义的事件。下面是\$emit 方法的基本语法格式：

语法: `vm.$emit(eventName, [...args])`  
 参数说明:  
`{string} eventName` // 事件名称  
`[...args]` // 附加参数

关于这个\$emit 事件触发方法如何使用，我们还是通过具体的代码实例进行讲解。请看下面这个“消息按钮”的代码。

【代码 8-17】（详见源代码 vuemethod 目录中的 vueemit.html 文件）

```

01 <div id="id-div-vue-emit">
02   <welbutton v-on:evhello="sayHello"></welbutton>
03 </div>
04 <script>
05   // define vue component
06   Vue.component('welbutton', {
07     template: '<button v-on:click="$emit('evhello')">
08       Click me to say hello
09     </button>'
10   });
11   // define vm
12   var vm = new Vue({
13     el: '#id-div-vue-emit',
14     methods: {
15       sayHello: function() {

```

```

16           console.log('Hello Vue --- vm.$emit!');
17       }
18   }
19 })
20 </script>

```

### 【代码说明】

- 第 01~03 行代码中，在页面中通过<div>元素定义了一个层，并定义了其 id 属性值（"id-div-vue-emit"）。在第 02 行代码中，通过 Vue.js 框架的组件（Component）语法（{{ }}）引用了一个“消息按钮<welbutton>”，实现了一个可以输出消息的按钮控件功能。
- 第 06~10 行的脚本代码中，通过“Vue.component()”方法实现了第 02 行代码中引用的“消息按钮<welbutton>”组件。具体说明如下：
  - 第 06 行代码中，定义了组件的名称（'welbutton'）。
  - 第 07~09 行代码中，通过 template 模板属性定义了组件的内容，一个基于<button>元素实现的功能按钮。同时，通过 Vue.js 框架定义的 v-on 指令绑定了单击（click）事件，实现了\$emit('evhello')事件触发方法。注意，参数（'evhello'）为定义在“消息按钮<welbutton>”组件上的自定义事件名称，在实际使用时需要绑定该自定义事件（见第 02 行代码）。
- 第 12~19 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象的（vm）。具体说明如下：
  - 第 13 行代码中，通过 el 属性绑定 DOM 元素（"id-div-vue-emit"）。
  - 第 14~18 行代码中，通过 methods 属性进行绑定方法操作。其中，在第 15~17 行代码中实现了 sayHello 方法，该方法对应第 02 行代码中自定义事件（'evhello'）所触发的方法。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vueemit.html 页面，页面初始效果如图 8.16 所示。

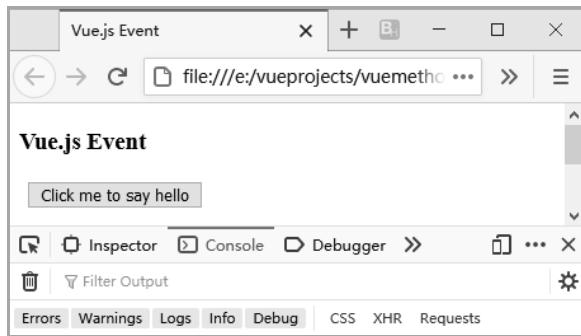


图 8.16 Vue 实例\$emit 方法（一）

如图 8.16 所示，尝试单击页面中的“Click me to say hello”按钮，页面更新效果如图 8.17 所示。

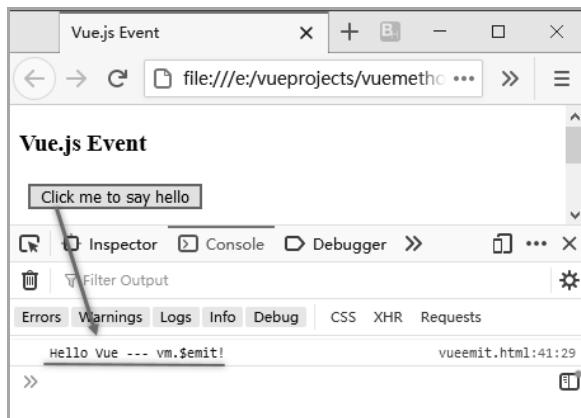


图 8.17 Vue 实例\$emit 方法（二）

如图 8.17 中的箭头所示，命令行控制台中输出了【代码 8-17】中第 16 行代码中定义的日志消息，我们定义的“消息按钮<welbutton>”组件，通过\$emit 事件触发方法实现了单击响应功能。

### 8.2.3 自定义事件方法

Vue.js 框架还设计了一组自定义事件触发方法（\$on 和\$once），用于触发 Vue 实例上用户自定义的事件。其中，\$once 方法只能触发一次，而\$on 方法在触发次数上是没有限制的。

首先，介绍\$on 自定义事件触发方法的基本语法格式。

```
语法: vm.$on(event, callback)
参数说明:
{string | Array<string>} event      // 数组只在 2.2.0+ 版本中支持
{Function} callback                 // 回调函数
```

关于这个\$on 自定义事件触发方法如何使用，我们还是通过具体的代码实例进行讲解。请看下面这个自定义“测试事件”按钮的代码：

**【代码 8-18】**（详见源代码 vuemethod 目录中的 vueon.html 文件）

```
01 <div id="id-div-event-on">
02   {{ msg }}
03 </div>
04 <div>
05   Test Event On:
06   <button
07     id='id-btn-event-on'
08     onclick="onBtnTestOn(this.id)">
09   Test On
10   </button>
11 </div>
12 <script>
13   // define global variables
14   var _times = 0;
```

```

15     // define vm
16     var vm = new Vue({
17         el: '#id-div-event-on',
18         data: {
19             msg: 'vm.$on()    ' + _times++ + '    times.'
20         }
21     })
22     // vm event --- $on
23     vm.$on('test', function(msg) {
24         console.log(msg);
25         vm.$data.msg = msg;
26     })
27     //
28     function onBtnTestOn(thisid) {
29         // generate msg
30         let i_msg = 'vm.$on()    ' + _times++ + '    times.';
31         // vm event --- $emit
32         vm.$emit('test', i_msg.toString());
33     }
34 </script>

```

### 【代码说明】

- 第 01~03 行代码中，在页面中通过<div>元素定义了一个层，并定义了其 id 属性值（"id-div-vue-on"）。在第 02 行代码中，通过 Vue.js 框架的插值模板语法（{{ }}）引用了一个对象（msg），实现了一个页面消息展示的功能。
- 第 06~10 行代码中，通过<button id='id-btn-event-on'>元素定义了一个按钮，用于触发下面用户自定义的“test”事件。
- 第 14 行的脚本代码中，定义了一个全局计数器变量（\_times），初始化数值为 0。
- 第 16~21 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 17 行代码中，通过 el 属性绑定 DOM 元素（"id-div-vue-on"）。
  - 第 18~20 行代码中，通过 data 属性进行绑定数据操作。其中，在第 19 行代码中定义了一个消息属性（msg），其对应第 02 行代码引用的对象（msg），用于显示用户单击按钮（<button id='id-btn-event-on'>）次数的信息，该信息是在页面中同步渲染出来的。
- 第 23~26 行代码中，通过 vm.\$on()方法定义了用户自定义 test 事件。具体说明如下：
  - 在第 23~26 行代码定义的回调函数中，接收一个 msg 消息参数。
  - 第 25 行代码中，通过 vm.\$data 将 msg 参数绑定到第 19 行代码定义的消息属性（msg）上。
- 第 28~33 行代码是第 08 行代码定义的按钮单击事件（onBtnTestOn()）的具体实现过程。说明如下：
  - 第 30 行代码中，通过自增表达式（\_times++）对计数器变量（\_times）进行（+1）

算术运算。

- 第 32 行代码中，通过\$emit()事件触发方法实现了对用户自定义 test 事件的触发操作。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vueon.html 页面，页面初始效果如图 8.18 所示。

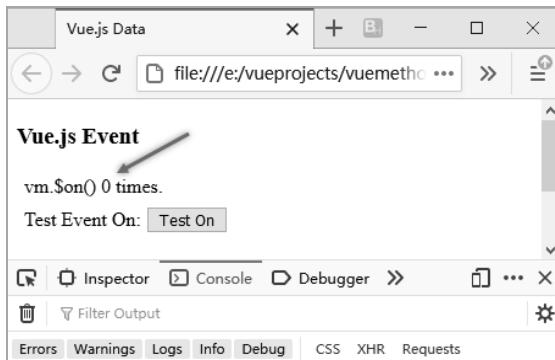


图 8.18 Vue 实例\$on 方法（一）

如图 8.18 中的箭头所示，页面初始时显示的单击次数为 0。然后，尝试单击页面中的“Test On”按钮，页面更新效果如图 8.19 所示。

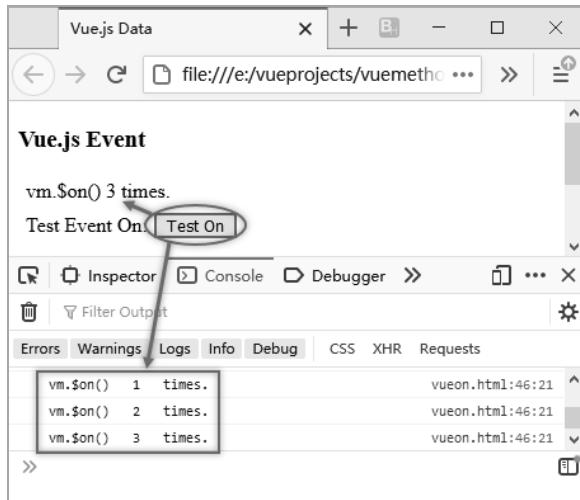


图 8.19 Vue 实例\$on 方法（二）

如图 8.19 中的箭头所示，每单击一次“Test On”按钮，就会触发一次用户自定义的 test 事件，页面中渲染更新的信息与命令行控制台中输出的日志消息是同步的。

在 Vue.js 框架中，除了这个 on 自定义事件，还定义了一个类似的\$once 自定义事件。如上文所述，二者的区别主要就体现在触发次数上，\$once 自定义事件只能触发一次。\$once 自定义事件触发方法的基本语法格式如下：

语法：vm.\$once(event, callback)

参数说明：

```
{string} event
{Function} callback      // 回调函数
```

关于这个\$once 自定义事件触发方法如何使用，我们还是通过具体的代码实例进行讲解。请看下面这个自定义“测试事件”按钮的代码。

**【代码 8-19】**（详见源代码 vuemethod 目录中的 vueonce.html 文件）

```
01 <div id="id-div-event-once">
02   {{ msg }}
03 </div>
04 <div>
05   Test Event Once:
06   <button
07     id='id-btn-event-once'
08     onclick="onBtnTestOnce(this.id)">
09   Test Once
10  </button>
11 </div>
12 <script>
13   // define global variables
14   var _times = 0;
15   // define vm
16   var vm = new Vue({
17     el: '#id-div-event-once',
18     data: {
19       msg: 'vm.$once()  ' + _times++ + '  times.'
20     }
21   })
22   // vm event --- $once
23   vm.$once('test', function(msg) {
24     console.log(msg);
25     vm.$data.msg = msg;
26   })
27   //
28   function onBtnTestOnce(thisid) {
29     // log click times
30     console.log('click button  ' + times + '  times.');
31     // generate msg
32     let i msg = 'vm.$once()  ' + times++ + '  times.';
33     // vm event --- $emit
34     vm.$emit('test', i_msg.toString());
35   }
36 </script>
```

### 【代码说明】

- 【代码 8-19】与【代码 8-18】基本类似，主要的区别如下：
  - 在第 23~26 行代码中，是通过 vm.\$once()方法（仅仅触发一次）定义了用户自

定义 test 事件。

- 第 34 行代码中，另外记录了用户单击按钮`<button id='id-btn-event-once'>`的次数，用以和用户自定义 test 事件触发次数区分开来。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 `vueonce.html` 页面，页面初始效果如图 8.20 所示。

如图 8.20 中的箭头所示，页面初始时显示的单击次数为 0。然后，尝试单击页面中的“Test Once”按钮，页面更新效果如图 8.21 所示。

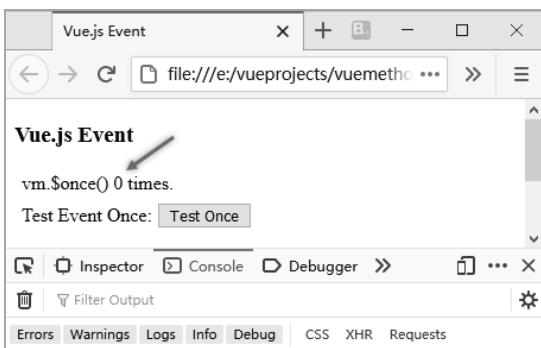


图 8.20 Vue 实例\$once 方法（一）

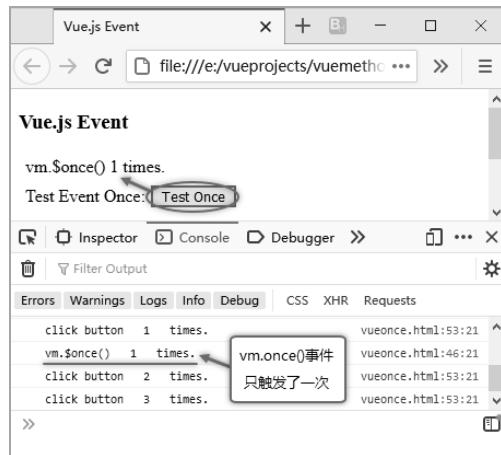


图 8.21 Vue 实例\$once 方法（二）

如图 8.21 中的箭头和标识所示，无论单击多少次“Test Once”按钮，只会触发一次用户自定义的 test 事件（由 `vm.$once()` 事件触发），而用户单击按钮的次数是同步增加的。另外，对比页面中渲染更新的信息与命令行控制台中输出的日志消息，可以清楚地看到二者的区别。

## 8.3 Vue.js 生命周期

本节介绍 Vue.js 框架中关于生命周期和生命周期钩子方面的内容。Vue.js 生命周期是创建 Vue.js 应用的核心基础。

### 8.3.1 Vue.js 生命周期图示

在 Vue.js 应用中，每个 Vue 实例在被创建（`new Vue()`）时，都要经过一系列的初始化过程，例如：设置数据监听、编译模板、将实例挂载到 DOM 上，以及在数据变化时更新 DOM 等。一般地，在前端应用框架中将这个过程称为应用的“生命周期”。

同时，在 Vue 应用的“生命周期”过程中，会根据进程演化的不同阶段定义一组相关过程方法（回调函数的形式）。前端框架会将这组过程方法称为“生命周期的钩子函数”，所谓

“钩子函数”就是给用户提供了在这些回调函数中添加自定义代码的机会。因此，这些“生命周期的钩子函数”类似于事件方法中的回调函数，只不过是只有存在于前端框架的“生命周期”中才会有意义。

关于 Vue.js 框架的“生命周期”的详细内容，其官网上提供了一幅非常详细的示意图（地址：<https://cn.vuejs.org/v2/guide/instance.html#生命周期图示>），如图 8.22 所示。

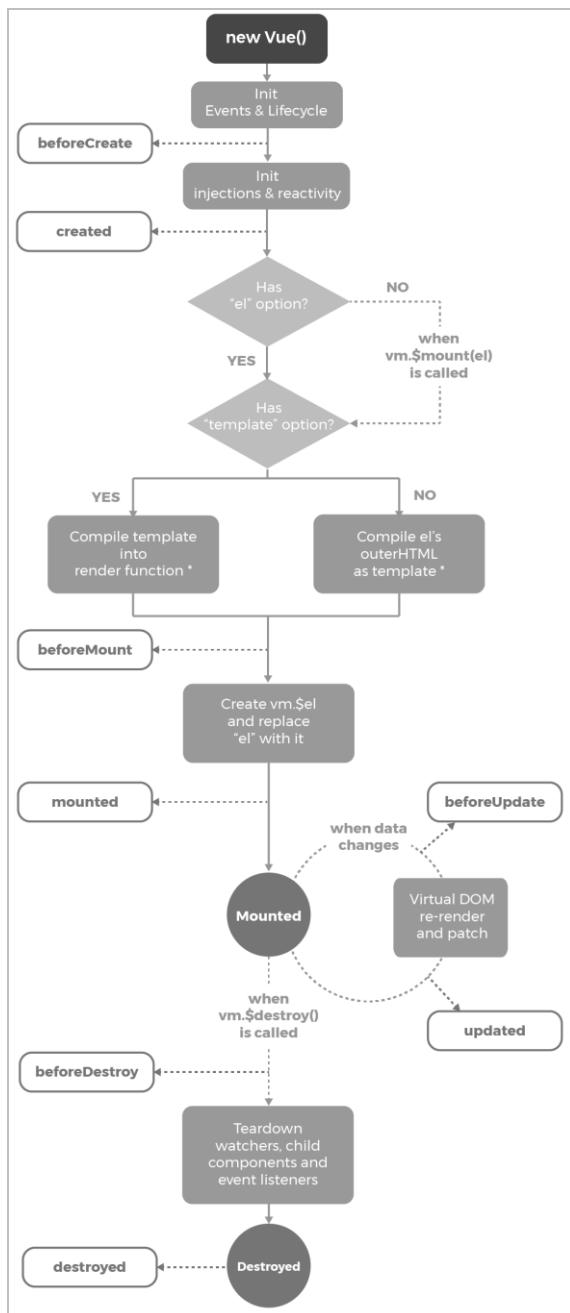


图 8.22 Vue.js 框架“生命周期”图示

如图 8.22 所示，从 Vue 实例的创建（“new Vue()”）开始，其“生命周期”就按部就班地开始了。接下来，我们列举几个最常用的“生命周期”阶段进行介绍。

- beforeCreate：在 Vue 实例（vm）初始化之后，在数据观测（data observer）和（event/watcher）事件配置之前被调用。
- created：在 Vue 实例（vm）创建完成后被立即调用。
- beforeMount：在 Vue 实例（vm）挂载开始之前被调用，此时相关的 render 函数首次被调用。
- mounted：在 Vue 实例（vm）挂载后调用，这时 el 已被新创建的 vm.\$el 替换了。
- beforeUpdate：在数据更新时调用，发生在虚拟 DOM 打补丁之前。
- updated：由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子函数。
- activated：被 keep-alive 缓存的组件激活时调用。
- deactivated：被 keep-alive 缓存的组件停用时调用。
- beforeDestroy：在 Vue 实例（vm）销毁之前调用，在这一阶段的 Vue 实例仍然完全可用。
- destroyed：在 Vue 实例（vm）销毁后调用。该钩子函数被调用后，对应 Vue 实例的所有指令都被解绑，所有的事件监听器被移除，所有的子实例也都会被销毁。

### 8.3.2 Vue.js 生命周期钩子

前一小节详细介绍了 Vue.js 框架的“生命周期”和“生命周期钩子”的相关内容。在 Vue.js 应用中，利用“生命周期钩子函数”可以为设计人员实现功能丰富的自定义代码功能。下面通过具体的代码实例进行介绍。

首先，先看下面这个关于 beforeCreate 和 created 钩子的代码实例。

**【代码 8-20】**（详见源代码 vuelifecycle 目录中的 vuelifecycle.html 文件）

```

01 <div class="text-wrapper" id="id-div-vue-lifecycle" v-html="msg">
02   {{ msg }}
03 </div>
04 <script>
05   // define Vue entry
06   var vm = new Vue({
07     el: '#id-div-vue-lifecycle',
08     data: {
09       msg: ''
10     },
11     beforeCreate: function() {
12       // this.$data.msg += 'beforeCreate' + '<br/>';
13       console.log('Lifecycle hook --- beforeCreate.');
14       console.log('$el: ' + this.$el);
15       console.log('$data: ' + this.$data);
16     },

```

```

17     created: function() {
18         this.$data.msg += 'created' + '<br/>';
19         console.log('Lifecycle hook --- created.');
20         console.log('$el: ' + this.$el);
21         console.log('$data: ' + this.$data);
22     },
23 }
24 </script>

```

### 【代码说明】

- 第 01~03 行代码中，在页面中通过<div>元素定义了一个层，并定义了其 id 属性值（"id-div-vue-lifecycle"）。在第 02 行代码中，通过 Vue.js 框架的插值模板语法（{{ }}）引用了一个对象（msg），实现了一个页面消息展示的功能。
- 第 06~23 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 07 行代码中，通过 el 属性绑定 DOM 元素（"id-div-vue-lifecycle"）。
  - 第 08~10 行代码中，通过 data 属性进行绑定数据操作。其中，第 09 行代码定义了一个消息属性（msg），其对应第 02 行代码引用的对象（msg），用于在页面中进行同步渲染操作。
  - 第 11~16 行代码定义了 beforeCreate 钩子函数，尝试将 el 和 data 属性作为日志信息在浏览器控制台中输出。另外，第 12 行代码通过实例 property 属性（\$data）更新了 msg 属性（不过，该行代码先是注释的状态）。
  - 第 17~22 行代码定义了 created 钩子函数，尝试将 el 和 data 属性作为日志信息在浏览器控制台中输出。另外，第 18 行代码通过实例 property 属性（\$data）更新了 msg 属性。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuelifecycle.html 页面，页面初始效果如图 8.23 所示。

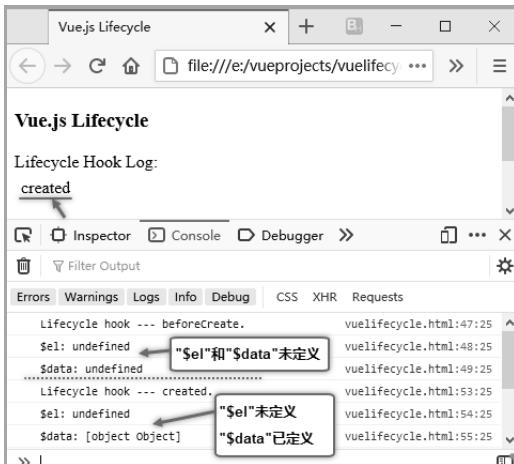


图 8.23 Vue.js 框架“生命周期钩子函数”（一）

如图 8.23 中的箭头和标识所示，在 Vue.js 生命周期的 beforeCreate 钩子阶段，\$el 和 \$data 均是“未定义”的状态。而在 Vue.js 生命周期的 created 钩子阶段，\$el 仍旧是“未定义”的状态，但是 \$data 已经是“已定义”的状态了，这一点与 Vue.js 官方文档中对于 created 钩子的描述是一致的。

另外，上面的第 12 行代码是注释的状态，那为什么要注释掉这行代码呢？因为，此时的 msg 属性还未定义，所以执行该行代码会导致 JavaScript 解释器报错，感兴趣的读者可以自行测试一下。

然后，再看下面这个关于 beforeMount 和 mounted 钩子的代码实例。

#### 【代码 8-21】（详见源代码 vuelifecycle 目录中的 vuelifecycle.html 文件）

```

01 <div class="text-wrapper" id="id-div-vue-lifecycle" v-html="msg">
02   {{ msg }}
03 </div>
04 <script>
05   // define Vue entry
06   var vm = new Vue({
07     el: '#id-div-vue-lifecycle',
08     data: {
09       msg: ''
10     },
11     beforeMount: function() {
12       this.$data.msg += 'beforeMount' + '<br/>';
13       console.log('Lifecycle hook --- beforeMount.');
14       console.log(this.$el);
15       console.log(this.$data);
16     },
17     mounted: function() {
18       this.$data.msg += 'mounted' + '<br/>';
19       console.log('Lifecycle hook --- mounted.');
20       console.log(this.$el);
21       console.log(this.$data);
22     },
23   })
24 </script>
```

#### 【代码说明】

- 【代码 8-21】与【代码 8-20】类似，区别就是使用的是 beforeMount 和 mounted 钩子函数。具体说明如下：
  - 第 11~16 行代码定义了 beforeMount 钩子函数，尝试将 el 和 data 属性作为日志信息在浏览器控制台中输出。另外，第 12 行代码通过实例 property 属性（\$data）更新了 msg 属性。
  - 第 17~22 行代码定义了 mounted 钩子函数，尝试将 el 和 data 属性作为日志信息在浏览器控制台中输出。另外，第 18 行代码通过实例 property 属性（\$data）更新了 msg 属性。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuecycle.html 页面，页面初始效果如图 8.24 所示。

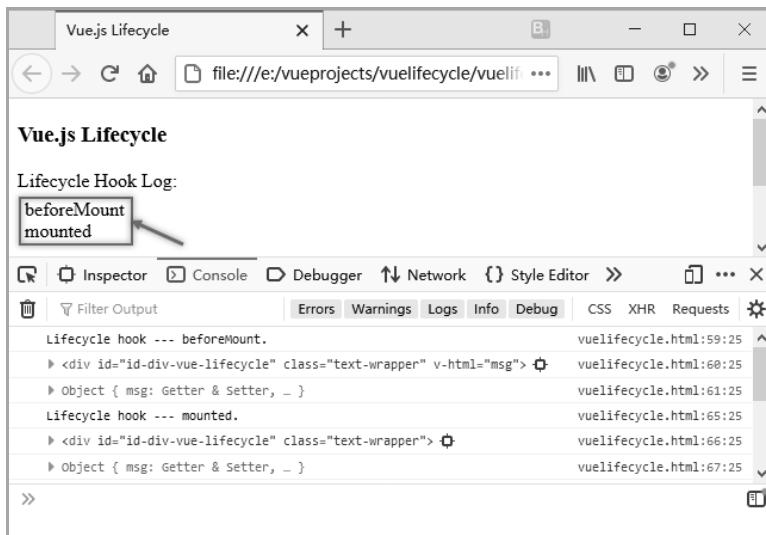


图 8.24 Vue.js 框架“生命周期钩子函数”（二）

如图 8.24 中的箭头和标识所示，在 Vue.js 生命周期的 beforeMount 和 mounted 钩子阶段，\$el 和 \$data 均是“已定义”的状态。这一点与 Vue.js 官方文档中对于 beforeMount 和 mounted 钩子的描述是一致的。

不过，beforeMount 和 mounted 这两个钩子还是有些区别的。在 beforeMount 钩子阶段，DOM 元素虽然是已定义、但还未加载进页面。而在 mounted 钩子阶段，DOM 元素才会被加载进页面。关于这一点，可以借助浏览器内置的 JavaScript 调试器进行测试验证，具体页面效果如图 8.25 和图 8.26 所示。

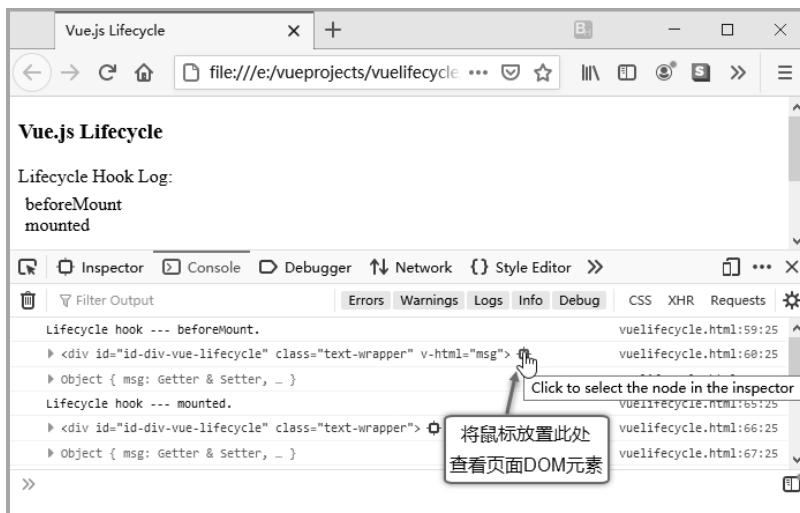


图 8.25 Vue.js 框架“生命周期钩子函数”（三）



图 8.26 Vue.js 框架“生命周期钩子函数”（四）

如图 8.25 和图 8.26 中的箭头和标识所示，两幅图对比的效果比较清楚。在 beforeMount 钩子阶段时，页面中还没有加载 DOM 元素（`<div id="id-div-vue-lifecycle">`），而在 mounted 钩子阶段时页面中，已经加载了 DOM 元素（`<div id="id-div-vue-lifecycle">`）。

下面再看一下这个关于 beforeUpdate 和 updated 钩子的代码实例。

#### 【代码 8-22】（详见源代码 vuelifecycle 目录中的 vuelifecycle.html 文件）

```

01 <div class="text-wrapper" id="id-div-vue-lifecycle" v-html="msg">
02   {{ msg }}
03 </div>
04 <div>
05   Lifecycle Hook:
06   <button id='id-btn-hook-update' onclick="onBtnHookUpdate(this.id)">
07     Hook Update
08   </button>
09 </div>
10 <script>
11   // define Vue entry
12   var vm = new Vue({
13     el: '#id-div-vue-lifecycle',
14     data: {
15       msg: ''
16     },
17     beforeUpdate: function() {
18       this.$data.msg += 'beforeUpdate' + '<br/>';
19       console.log('Lifecycle hook --- beforeUpdate.');
20       this.$data.msg += 'updated' + '<br/>';
21     },
22     updated: function() {
23       console.log('Lifecycle hook --- updated.');
24     }
25   })

```

```

26      // func --- Hook Update
27      function onBtnHookUpdate(thisid) {
28          vm.$data.msg += 'manual to update' + '<br/>';
29          // vm.$forceUpdate();
30      }
31  </script>

```

### 【代码说明】

- 【代码 8-22】在【代码 8-20】的基础上修改而成，它们的区别就是是否使用 beforeUpdate 和 updated 钩子函数。由于 beforeUpdate 和 updated 钩子在 data 属性的数据被修改后才会触发，因此在这段代码中是通过人工修改 msg 属性值来实现的。具体说明如下：
  - 第 06~08 行代码中，通过<button>元素定义了一个按钮及其单击事件（onBtnHookUpdate()）方法，用于触发实现人工修改 msg 属性值。
  - 第 17~21 行代码定义了 beforeUpdate 钩子函数，通过实例 property 属性（\$data）更新了 msg 属性值，并在浏览器控制台中输出相关的日志信息。
  - 第 22~24 行代码定义了 updated 钩子函数，在浏览器控制台中输出相关的日志信息。
  - 第 27~30 行代码是单击事件（onBtnHookUpdate()）方法的实现过程，以人工方式通过实例 property 属性（\$data）更新了 msg 属性值。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuecycle.html 页面，页面初始效果如图 8.27 所示。

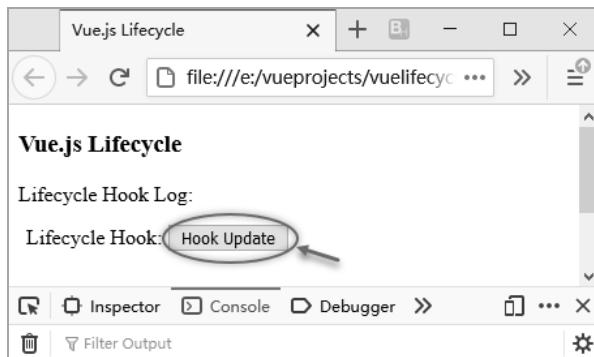


图 8.27 Vue.js 框架“生命周期钩子函数”（五）

如图 8.27 中的箭头和标识所示，由于 data 属性的数据没有发生改变，因此 beforeUpdate 和 updated 钩子也没有被触发。尝试单击“Hook Update”按钮以人工方式改变 data 属性的数据，页面效果如图 8.28 所示。

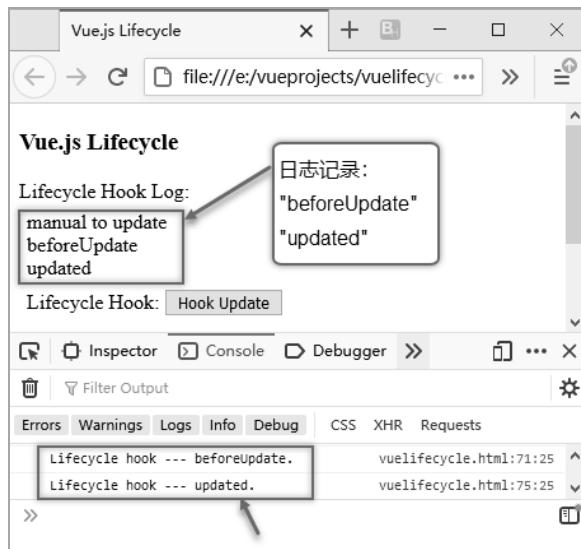


图 8.28 Vue.js 框架“生命周期钩子函数”（六）

如图 8.28 中的箭头和标识所示，单击“Hook Update”按钮以人工方式改变 data 属性的数据后，beforeUpdate 和 updated 钩子函数被触发了。页面中的信息显示按钮触发了并显示出 data 属性数据修改后的内容，浏览器控制台输出的是 beforeUpdate 和 updated 钩子函数定义的日志信息。

最后，再看一下这个关于 beforeDestroy 和 destroyed 钩子的代码实例。

**【代码 8-23】**（详见源代码 vuelifecycle 目录中的 vuelifecycle.html 文件）

```

01 <div class="text-wrapper" id="id-div-vue-lifecycle" v-html="msg">
02   {{ msg }}
03 </div>
04 <div>
05   Lifecycle Hook:
06   <button id='id-btn-hook-destroy' onclick="onBtnHookDestroy(this.id)">
07     Hook Destroy
08   </button>
09   <button id='id-btn-hook-destroy2' onclick="onBtnHookDestroy2(this.id)">
10     Hook Destroy Again
11   </button>
12 </div>
13 <script>
14   // define Vue entry
15   var vm = new Vue({
16     el: '#id-div-vue-lifecycle',
17     data: {
18       msg: ''
19     },
20     beforeDestroy: function() {
21       console.log('Lifecycle hook --- beforeDestroy.');
22       console.log(this.$el);

```

```

23         console.log(this.$data);
24     },
25     destroyed: function() {
26         console.log('Lifecycle hook --- destroyed.');
27         console.log(this.$el);
28         console.log(this.$data);
29     },
30     errorCapture: function() {}
31 }
32 // func --- Hook Destroy
33 function onBtnHookDestroy(thisid) {
34     console.log('manual destroy.');
35     vm.$destroy();
36 }
37 // func --- Hook Destroy Again
38 function onBtnHookDestroy2(thisid) {
39     console.log('manual destroy again.');
40     vm.$destroy();
41 }
42 </script>

```

### 【代码说明】

- 【代码 8-23】在【代码 8-20】的基础上修改而成，它们的区别就是是否使用 beforeDestroy 和 destroyed 钩子函数。由于 beforeDestroy 和 destroyed 钩子在 Vue 实例销毁之后被触发，因此在这段代码中是通过人工销毁 Vue 实例的方式来实现的。具体说明如下：
  - 第 06~08 行代码中，通过<button>元素定义了第一个按钮及其单击事件（onBtnHookDestroy()）方法，用于实现人工销毁 Vue 实例（vm）。
  - 第 09~11 行代码中，通过<button>元素定义了第二个按钮，定义了单击事件（onBtnHookDestroy2()）方法，用于尝试再次人工销毁 Vue 实例（vm）。
  - 第 20~24 行代码定义了 beforeDestroy 钩子函数，尝试将 el 和 data 属性作为日志信息在浏览器控制台中输出，并在浏览器控制台中输出相关的日志信息。
  - 第 25~29 行代码定义了 destroyed 钩子函数，同样尝试将 el 和 data 属性作为日志信息在浏览器控制台中输出，并在浏览器控制台中输出相关的日志信息。
  - 第 33~36 行代码是单击事件（onBtnHookDestroy()）方法的实现过程，其中第 35 行代码通过调用\$destroy()方法以人工方式销毁 Vue 实例（vm）。
  - 第 38~41 行代码是单击事件（onBtnHookDestroy2()）方法的实现过程，其中第 40 行代码通过调用\$destroy()方法销毁尝试再次以人工方式销毁 Vue 实例（vm）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuecycle.html 页面，页面初始效果如图 8.29 所示。

如图 8.29 中的箭头和标识所示，尝试单击“Hook Destroy”按钮以人工方式销毁 Vue 实例（vm），页面效果如图 8.30 所示。

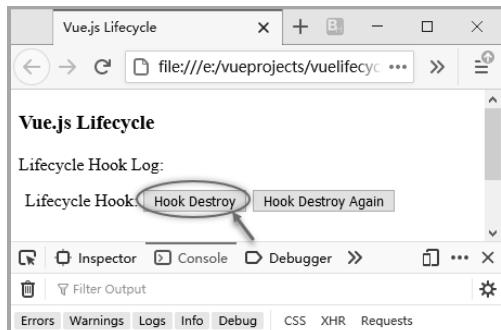


图 8.29 Vue.js 框架“生命周期钩子函数”（七）

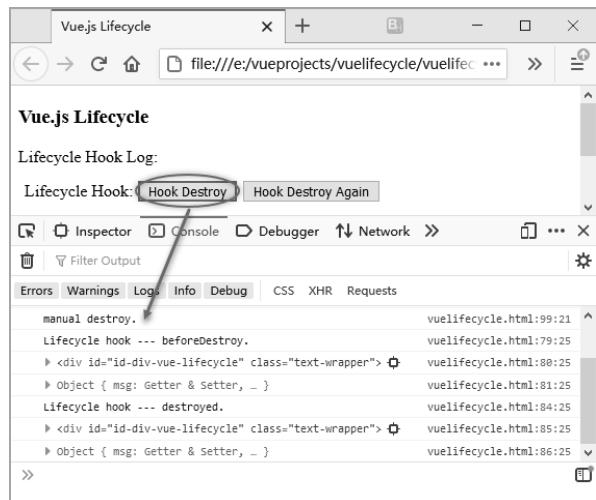


图 8.30 Vue.js 框架“生命周期钩子函数”（八）

如图 8.30 中的箭头和标识所示，浏览器控制台输出了“beforeDestroy”和“destroyed”钩子函数定义的日志信息。然后，再次尝试单击“Hook Destroy Again”按钮以人工方式再次销毁 Vue 实例（vm），页面效果如图 8.31 所示。

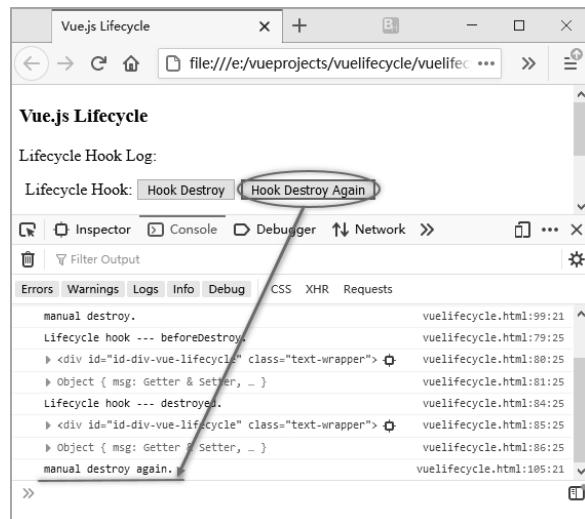


图 8.31 Vue.js 框架“生命周期钩子函数”（九）

如图 8.31 中的箭头和标识所示，浏览器控制台输出了“Hook Destroy Again”按钮单击事件处理方法中定义的日志信息，却没有再次输出 beforeDestroy 和 destroyed 钩子函数定义的日志信息，说明 Vue 实例（vm）此时已经被销毁了。

# 第 9 章

## Vue.js 模板语法

Vue.js 框架使用了基于 HTML 语义的模板语法，这也是 Vue.js 框架能够受到大多数前端开发人员喜爱的原因之一。本章的重点内容就是介绍 Vue.js 框架中关于模板语法的相关知识点。

通过本章的学习可以：

- 了解 Vue.js 的插值方法。
- 掌握 Vue.js 指令的使用方式。
- 理解关于 Vue.js 模板语法中缩写的内容。

### 9.1 Vue.js 模板语法介绍

Vue.js 框架使用了基于 HTML 语义的模板语法，允许开发者声明式地将 DOM 元素绑定至底层 Vue 实例的数据上。在 Vue.js 框架中，所有的模板都是合法的 HTML 语法，自然也能被遵循规范的浏览器和 HTML 解析器进行解析。

Vue.js 框架的底层是通过将模板语法编译成虚拟 DOM 元素的渲染函数来实现的。结合 Vue.js 框架自身的响应系统，Vue 应用能够智能地计算出需要重新渲染的最少组件，同时把操作 DOM 的次数减至最少。

另外，如果开发者熟悉虚拟 DOM 并且偏爱原生 JavaScript 代码的开发方式，Vue.js 框架也允许不使用模板，而采用直接写渲染（render）函数的方式，并借助于选用 JSX 语法进行开发。

### 9.2 Vue.js 插值

本节介绍 Vue.js 框架模板语法中插值的内容，并讲解如何通过插值实现数据绑定的功能。

## 9.2.1 文本插值

在 Vue.js 框架中，进行数据绑定最常见的方式就是使用遵循 Mustache 语法的“双大括号（{{ }}）”形式的文本插值。下面看一个最简单的、使用文本插值的代码实例。

**【代码 9-1】**（详见源代码 vuetemplate 目录中的 vuetemplate.html 文件）

```

01 <div id="id-div-templ-text">
02   <table>
03     <caption></caption>
04     <tr>
05       <th>id</th>
06       <th>Name</th>
07       <th>Age</th>
08     </tr>
09     <tr>
10       <td>{{ id }}</td>
11       <td>{{ name }}</td>
12       <td>{{ age }}</td>
13     </tr>
14   </table>
15 </div>
16 <script>
17   var vm = new Vue({
18     el: '#id-div-templ-text',
19     data: {
20       id: 1,
21       name: 'King',
22       age: '26'
23     }
24   })
25 </script>

```

### 【代码说明】

- 第 01~15 行代码中，通过<div>元素定义了一个层（`id="id-div-templ-text"`），在该层内部定义了一个表格（`<table>`）。在该表格内，通过 Vue.js 框架的文本插值模板语法（{{ }}）引用了一组对象（`id`、`name` 和 `age`）。
- 第 17~24 行代码中，通过“`new Vue()`”构造函数实例化 `Vue` 对象（`vm`）。同时，这段代码创建了 `Vue` 对象的入口，并将该对象所定义的内容渲染到页面中对应的 DOM 元素中。具体说明如下：
  - 第 18 行代码中，通过 `el` 属性绑定了第 01~15 行代码中定义的 DOM 元素（`id="id-div-templ-text"`）。
  - 第 19~23 行代码中，通过 `data` 属性进行绑定数据操作。其中，在第 20~22 行代码中定义了一组 `property` 属性（`id`、`name` 和 `age`）并进行了初始化操作，一一对应于上面表格（`<table>`）中通过 `Vue` 文本插值模板语法引用的对象（`id`、`name`

和 age)。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuemaple.html 页面，效果如图 9.1 所示。

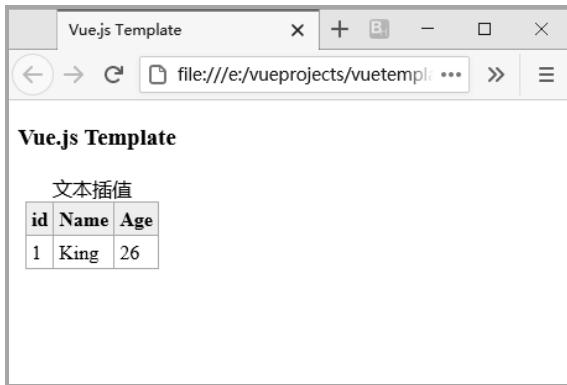


图 9.1 Vue.js 之文本插值模板语法应用

如图 9.1 所示，Mustache 语法的“双大括号（{{ }}）”文本插值标签被替代为对应 data 对象上的 property 属性值。在 Vue.js 框架下，当绑定的数据对象上的 property 属性值发生改变，插值标签处的内容也都会随之进行更新。

## 9.2.2 原始 HTML 插值

在 Vue.js 框架中，使用 Mustache 语法的“双大括号（{{ }}）”形式的文本插值，会将任何内容转换为文本形式。这样就会带来一个问题，如果想在页面展示 HTML 代码定义的内容就无法实现了，因为文本插值会将 HTML 标签直接作为文本输出。

因此，Vue.js 框架设计了一个 v-html 指令，用于直接输出原始 HTML 代码定义的内容。下面来看一个使用原始 HTML 文本插值的代码实例。

**【代码 9-2】**（详见源代码 vuemaple 目录中的 vuemaple.html 文件）

```

01 <div id="id-div-templ-html">
02   <table>
03     <caption>原始 HTML 插值</caption>
04     <tr>
05       <th>Msg</th>
06       <td>{{ msg }}</td>
07     </tr>
08     <tr>
09       <th>Msg HTML</th>
10       <td v-html="msgHtml"></td>
11     </tr>
12   </table>
13 </div>
14 <script>
15   var vm = new Vue({

```

```

16         el: '#id-div-templ-html',
17         data: {
18             msg: 'King is a leader.<br>He is a good leader.',
19             msgHtml: 'King is a leader.<br>He is a good leader.'
20         }
21     })
22 </script>

```

### 【代码说明】

- 第 06 行代码中，通过 Vue.js 框架的文本插值模板语法（{{ }}）引用了第一个对象（msg），对象（msg）的定义在 Vue 构造函数中的第 18 行代码。
- 第 10 行代码中，通过 Vue.js 框架的 v-html 指令插值语法引用了第二个对象（msgHtml），对象（msgHtml）的定义在 Vue 构造函数中的第 19 行代码。
- 第 15~21 行代码中，定义了 Vue 构造函数。其中，第 17~20 行代码通过 data 属性进行绑定数据操作。具体说明如下：
  - 第 18 行代码中，定义了第一个 property 属性（msg）并初始化为一行字符串信息，对应第 06 行代码中通过 Vue 文本插值模板语法引用的对象（msg）。不过请读者注意，该字符串中包含了一个换行元素（<br>），因此实际上是一段 HTML 代码。
  - 第 19 行代码中，定义了第二个 property 属性（msgHtml）并初始化为与上面（msg）相同的字符串信息，对应第 10 行代码中通过 v-html 指令插值语法引用的对象（msgHtml）。通过第 18 行和第 19 行代码页面显示效果的对比，可以看到文本插值与 v-html 指令插值的功能差别。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuemap.html 页面，效果如图 9.2 所示。

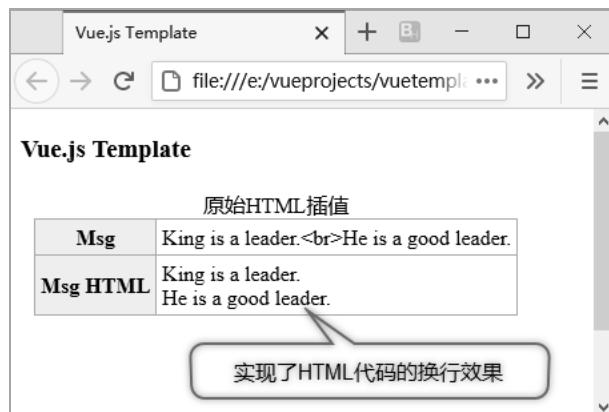


图 9.2 Vue.js 之 v-html 指令插值语法应用

如图 9.2 中的标识所示，使用 Mustache 语法的“双大括号（{{ }}）”文本插值方式，将 HTML 换行标签（<br>）识别成普通文本进行显示。而使用 v-html 指令插值语法方式，才

可以实现 HTML 代码的换行效果。

### 9.2.3 使用 JavaScript 表达式

在前面的代码实例中，使用 Mustache 语法的文本插值基本都是绑定了简单的 property 属性值。其实，Vue.js 框架对于所有的数据绑定，都是支持一个完整的 JavaScript 表达式的。不过需要注意，这里支持的仅仅是 JavaScript 表达式，而不支持 JavaScript 语句。下面看一个使用 JavaScript 表达式插值的代码实例。

**【代码 9-3】**（详见源代码 vuetemplate 目录中的 vuetemplate.html 文件）

```

01 <div id="id-div-templ-js">
02   <table>
03     <caption>JavaScript 表达式</caption>
04     <tr>
05       <th>id</th>
06       <th>Name</th>
07       <th>Gender</th>
08     </tr>
09     <tr>
10       <td>{{ id + 1 }}</td>
11       <td>{{ name.toLowerCase() }}</td>
12       <td>{{ gender ? 'male' : 'female' }}</td>
13     </tr>
14     <tr>
15       <td>{{ id * id }}</td>
16       <td>{{ name.toUpperCase() }}</td>
17       <td>{{ gender ? 'female' : 'male' }}</td>
18     </tr>
19     <tr>
20       <td>{{ Math.round(Math.random() * 100) }}</td>
21       <td>{{ name.split('').reverse().join('') }}</td>
22       <td>{{ (new Date()).getFullYear() + 1900 }}</td>
23     </tr>
24   </table>
25 </div>
26 <script>
27   var vm = new Vue({
28     el: '#id-div-templ-js',
29     data: {
30       id: 1,
31       name: 'King',
32       gender: true
33     }
34   })
35 </script>
```

**【代码说明】**

- 第 10 行代码中，通过文本插值模板语法（{{ }}）引用了 property 属性（id），并

改写成为 JavaScript 算术运算表达式 (`id + 1`)。同样地，第 15 行代码也引用了 `property` 属性 (`id`)，并改写成为 JavaScript 算术运算表达式 (`id * id`)。

- 第 11 行代码中，通过文本插值模板语法 (`{{ }}`) 引用 `property` 属性 (`name`)，并通过 JavaScript 的 `String` 对象方法 `toLowerCase()`，将属性 (`name`) 的字符串修改为小写格式。同样地，第 15 行代码将属性 (`name`) 的字符串修改为大写格式，第 21 行代码将属性 (`name`) 的字符串进行反转操作。
- 第 12 行和第 17 行代码中，通过文本插值模板语法 (`{{ }}`) 引用 `property` 属性 (`gender`)，并通过三元表达式判断属性 (`gender`) 的布尔值，并计算出结果 (`'male'` or `'female'`)。
- 在第 20 行代码定义的文本插值模板语法 (`{{ }}`) 中，通过引用 JavaScript 的 `Math` 对象方法计算出了一个随机数 (100 以内)。
- 在第 22 行代码定义的文本插值模板语法 (`{{ }}`) 中，通过引用 JavaScript 的 `Date` 对象方法获取当前时间的年份。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 `vuetemplate.html` 页面，效果如图 9.3 所示。

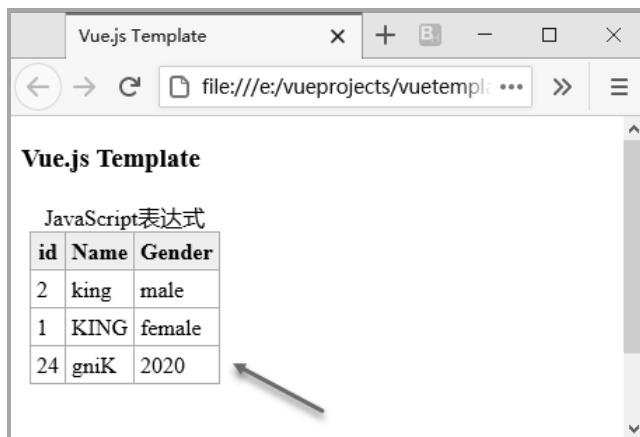


图 9.3 Vue.js 之 JavaScript 表达式插值应用

如图 9.3 中的箭头所示，在 Mustache 语法的文本插值方式中，使用 JavaScript 表达式插值语法可以被 Vue.js 框架正确解析。

## 9.3 Vue.js 指令

本节介绍 Vue.js 框架中指令的内容。Vue.js 指令可以绑定到 HTML 页面代码中使用，从而实现较为复杂的功能。

### 9.3.1 Vue 指令概述

Vue.js 框架设计了一个复杂且完整的“指令”系统，用于实现较为复杂的动态页面渲染功能。在 Vue.js 框架下，一般的指令（Directives）都是指带有前缀（“v-”）的特殊属性（Attribute）。

Vue 指令的功能是，当表达式的值（一般对应 property 属性）发生改变时，将其产生的连带变化“响应式”地作用于 DOM，从而完成页面的渲染操作。关于 Vue 指令的具体形式，请参考下面的简单示例：

```
<p v-if="seen">Now, you can see me!</p>
```

上面代码示例中的 v-if 就是一个 Vue 指令，由前缀（v-）开头，连接具体参数指令（if）。顾名思义，v-if 就是一个条件表达式指令。从这个代码示例中可以看到，Vue 指令的预期值是单个的 JavaScript 表达式。当然也有例外情况，比如：v-for 指令，后面会单独对其进行介绍。

### 9.3.2 v-if 条件表达式指令

Vue.js 框架设计了一个 v-if 指令，用于实现条件表达式的判断功能。这个 v-if 指令实现了 JavaScript 脚本语言的“if | if else | if elseif else”条件表达式的功能，也就是将条件表达式逻辑嵌入到 Vue 代码中去执行。

在 Vue 代码中使用 v-if 指令，与在 JavaScript 代码中使用“if-elseif-else”语法略有不同，使用 v-if 指令的 Vue 代码在格式上略显烦琐，需要设计人员仔细审查代码逻辑，从而避免出现逻辑错误。

下面还是通过具体的代码实例进行讲解，请看这个通过 v-if 指令“显示”和“隐藏”页面元素的应用。

**【代码 9-4】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```
01 <div id="id-div-derectives-if">
02   <p>Which language do you like, JavaScript or Vue.js?</p>
03   <p>I like <b v-if="js">JavaScript</b><b v-if="vue">Vue.js</b>.</p>
04 </div>
05 <script>
06   var vm = new Vue({
07     el: '#id-div-derectives-if',
08     data: {
09       js: true,
10       vue: false
11     }
12   })
13 </script>
```

**【代码说明】**

- 第 01~04 行代码中，在页面中通过<div>元素定义了一个层，并定义其 id 属性值（"id-div-derectives-if"）。在第 03 行代码中，分别通过 v-if 指令判断两个对象（js

和 vue) 的布尔值，从而实现在页面中“显示”和“隐藏”元素的功能。

- 第 06~12 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。同时，这段代码创建 Vue 对象的入口，并将该对象所定义的内容渲染到页面中对应的 DOM 元素中。具体说明如下：
  - 第 07 行代码中，通过 el 属性绑定 DOM 元素（"id-div-derectives-if"）。
  - 第 08~11 行代码中，通过 data 属性进行绑定数据操作。其中，第 09、10 行代码分别定义了两个布尔值属性（js 和 vue），并初始化为布尔值（true 和 false），对应第 03 行代码中引用的两个对象（js 和 vue）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.4 所示。



图 9.4 v-if 指令应用

如图 9.4 中的箭头所示，页面中仅显示了“JavaScript”信息，而“Vue.js”信息没有显示出来。这个结果与第 09、10 行代码定义的两个布尔值属性值（js: true 和 vue: false）相对应，v-if 指令判断为 true，则显示页面元素，v-if 指令判断为 false，则隐藏页面元素。

下面再介绍一个通过 v-if 和 v-else 指令实现“显示”和“隐藏”页面元素的应用。

**【代码 9-5】**（详见源代码 vuederectives 目录中的 vuederectives.html 文件）

```

01 <div id="id-div-derectives-if">
02   <p>Which language do you like, JavaScript or Vue.js?</p>
03   <p>I like <b v-if="b_tf">Angular</b><b v-else>Vue.js</b>.</p>
04 </div>
05 <script>
06   var vm = new Vue({
07     el: '#id-div-derectives-if',
08     data: {
09       b_tf: false
10     }
11   })
12 </script>
```

**【代码说明】**

- 第 01~04 行代码中，在页面中通过<div>元素定义了一个层及其 id 属性值

( "id-div-derectives-if" )。在第 03 行代码中，分别通过 v-if 指令判断对象 ( b\_tf ) 的布尔值，如果值为 false，则执行相对应的 v-else 指令，从而实现在页面中“显示”和“隐藏”元素的功能。

- 第 06~12 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象 ( vm )。其中，第 09 行代码定义了一个布尔值属性 ( b\_tf )，并初始化为布尔值 ( false )，对应第 03 行代码中引用的对象 ( b\_tf )。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.5 所示。

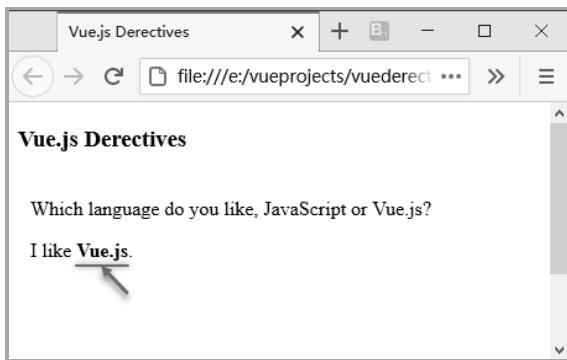


图 9.5 “v-if | v-else” 指令应用

如图 9.5 中的箭头所示，页面中仅显示了“Vue.js”信息，而“JavaScript”信息没有显示出来，这个结果与第 09 行代码定义的布尔值属性值 ( b\_tf: false ) 相对应，v-if 指令判断为 false，则会去执行 v-else 指令引用的页面元素。

最后再介绍一下如何通过完整的 v-if, v-else-if 和 v-else 指令，来实现“显示”和“隐藏”页面元素的方法。

#### 【代码 9-6】（详见源代码 vuederectives 目录中的 vuederectives.html 文件）

```

01 <div id="id-div-derectives-if">
02   <p>Which language do you like, JavaScript, Vue.js or both of them?</p>
03   <p>I like
04     <b v-if="type == 'J'">JavaScript</b>
05     <b v-else-if="type == 'V'">Vue.js</b>
06     <b v-else-if="type == 'A'">both JavaScript and Vue.js</b>
07     <b v-else>None</b>.
08   </p>
09 </div>
10 <script>
11   var vm = new Vue({
12     el: '#id-div-derectives-if',
13     data: {
14       type: 'A'
15     }
16   })

```

```
17 </script>
```

### 【代码说明】

- 第 01~09 行代码中，在页面中通过<div>元素定义了一个层，并定义其 id 属性值（"id-div-derecrtives-if"）。其中，第 04~07 行代码通过 v-if，v-else-if 和 v-else 指令实现多重条件的判断，具体说明如下：
  - 在第 04 行代码中，通过 v-if 指令判断对象（type）的值是否等于字符 J，如果结果为 true（真），则在页面中显示字符串“JavaScript”。
  - 在第 05 行代码中，通过 v-else-if 指令判断对象（type）的值是否等于字符 V，如果结果为 true，则在页面中显示字符串“Vue.js”。
  - 在第 06 行代码中，通过 v-else-if 指令判断对象（type）的值是否等于字符 A，如果结果为 true，则在页面中显示字符串“both JavaScript and Vue.js”。
  - 在第 07 行代码中，如果前面的判断结果没有一个为 true，则在页面中显示字符串“None”。
- 第 11~16 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。其中，第 14 行代码定义了一个属性（type），并初始化为字符（'A'），对应第 04~07 行代码中引用的对象（type）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.6 所示。

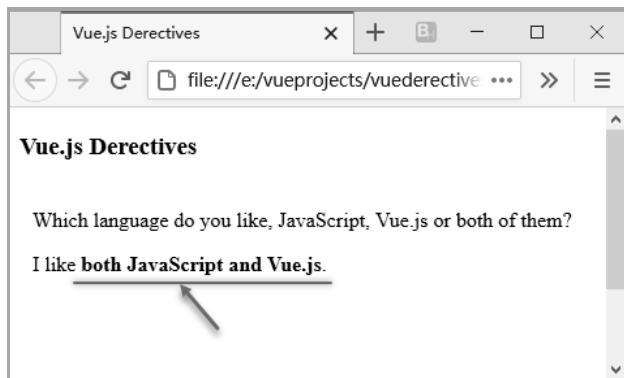


图 9.6 “v-if | v-else-if | v-else” 指令应用（一）

如图 9.6 中的箭头所示，页面中显示了“type='A'" 对应的信息。假设在第 14 行代码中将属性（type）的值初始化为任意字符（例如：'N'），则页面效果如图 9.7 所示。

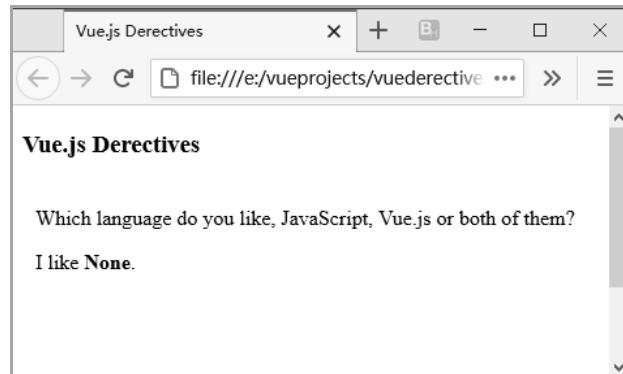


图 9.7 “v-if | v-else-if | v-else” 指令应用（二）

如图 9.7 中所示,由于条件判断结果没有任一项为真,则页面中显示了第 07 行代码中 v-else 指令对应的信息。

### 9.3.3 v-show 显示指令

Vue.js 框架还设计了一个 v-show 指令,用于实现“显示”或“隐藏”页面元素的功能。虽然看起来,这个 v-show 指令实现的效果与 v-if 指令实现的效果类似,但是在底层实现上还是有区别的。

使用 v-if 指令“隐藏”的页面元素会真正地被删除,在最终的 HTML 页面 DOM 树中不会出现这些元素。而使用 v-show 指令“隐藏”的页面元素仅仅就是隐藏起来,在最终的 HTML 页面 DOM 树中这些元素还是存在的,只是被 CSS 代码隐藏不显示而已。

下面通过具体的代码实例进行讲解,请看这个通过 v-show 和 v-if 指令“显示”和“隐藏”页面元素的应用。

**【代码 9-7】** (详见源代码 vuederectives 目录中的 vuederectives.html 文件)

```

01 <div id="id-div-derectives-show">
02   <p>Which language do you like, JavaScript or Vue.js?</p>
03   <p>I like
04     <b v-show="noshow">JavaScript</b>
05     <b v-show="show">Vue.js</b>.
06   </p>
07   <p>I like
08     <b v-if="js">JavaScript</b>
09     <b v-if="vue">Vue.js</b>.
10   </p>
11 </div>
12 <script>
13   var vm = new Vue({
14     el: '#id-div-derectives-show',
15     data: {
16       noshow: false,
17       show: true,
18       js: true,

```

```

19           vue: false
20       }
21   })
22 </script>

```

### 【代码说明】

- 第 01~11 行代码中，在页面中通过<div>元素定义了一个层，并定义其 id 属性值（"id-div-derecives-show"）。第 04、05 行代码分别通过 v-show 指令判断两个对象（noshow 和 show）的布尔值。第 08、09 行代码分别通过 v-if 指令判断另外两个对象（js 和 vue）的布尔值。这样写代码，我们就可以针对 v-show 指令和 v-if 指令在页面中的执行结果进行对比。
- 第 13~21 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。其中，第 15~20 行代码通过 data 属性进行绑定数据操作，具体说明如下：
  - 第 16、17 行代码分别定义了两个布尔值属性（noshow 和 show），并初始化为布尔值（false 和 true），对应第 04 行和第 05 行代码中引用的两个对象（noshow 和 show）。
  - 第 18、19 行代码分别定义了两个布尔值属性（js 和 vue），并初始化为布尔值（true 和 false），对应第 08 行和第 09 行代码中引用的两个对象（js 和 vue）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.8 所示。



图 9.8 v-show 指令应用（一）

如图 9.8 中的箭头所示，页面中显示了第 05 行代码定义的“Vue.js”信息，第 04 行代码定义的“JavaScript”信息没有显示出来。这个结果与第 16、17 行代码中定义的两个布尔值属性值(noshow: false 和 show: true)相对应。另外，页面中显示了第 08 行代码定义的“JavaScript”信息，第 09 行代码定义的“Vue.js”信息没有显示出来。这个结果与第 18、19 行代码中定义的两个布尔值属性值 (js: true 和 vue: false) 相对应。

可以看到，v-show 指令和 v-if 指令执行后，在页面中得到了相同的显示效果。但是请注意，在底层代码逻辑上二者是不同的，如图 9.9 所示。

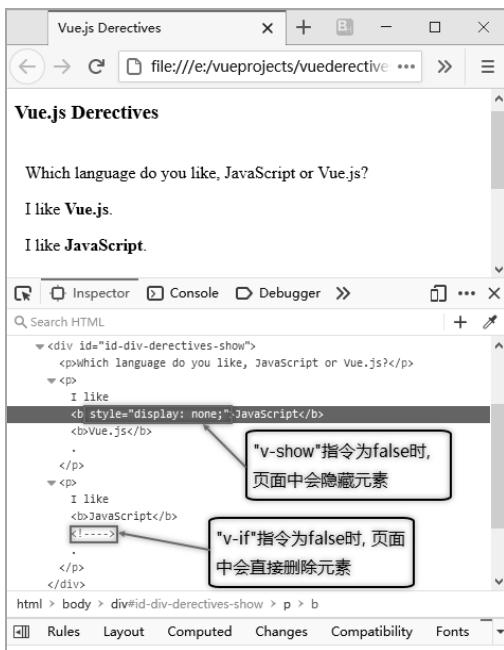


图 9.9 v-show 指令应用（二）

如图 9.9 中的箭头和标识所示，v-show 指令判断结果为 false 时，会通过 CSS 代码将元素隐藏 (display: none) 起来。而 v-if 指令判断结果为 false 时，则会将元素直接删除掉。

#### 9.3.4 使用<template>元素渲染分组

本小节介绍 Vue.js 框架的<template>元素，这个<template>元素本质上是一个虚拟元素，在最终的页面代码中是不体现的。但是，这个<template>元素在 Vue.js 框架下又十分有用，可以配合 v-if 指令完成很多强大的功能。

在 Vue.js 框架中，v-if 指令必须和页面元素配合在一起使用。从前面介绍的代码实例中可以看到，通过 v-if 指令的条件选择功能，可以实现一组页面元素的切换效果。这时候，就可以通过<template>元素将这组页面元素包裹起来，而这个<template>元素在最终的页面中不会渲染出来。从这个意义上讲，Vue.js 框架定义的<template>元素更像是一个一个抽象的页面元素包裹器。

下面通过具体的代码实例进行讲解，请看这个通过`<template>`元素包裹一组页面元素的应用。

### 【代码 9-8】（详见源代码 vuederectives 目录中的 vuederectives.html 文件）

```

01 <div id="id-div-derectives-templ">
02     <p>Login Region</p>
03     <template>
04         <p><label>Username:</label></p>
05     </template>
06     <template>
07         <p><label>Email:</label></p>
08     </template>
09 </div>
10 <script>
11     // Vue Entry
12     var vm = new Vue({
13         el: '#id-div-derectives-templ'
14     })
15 </script>

```

### 【代码说明】

- 第 01~09 行代码中，在页面中通过`<div>`元素定义了一个层，并定义其 `id` 属性值（"id-div-derectives-templ"）。第 03~05 行代码和第 06~08 行代码分别通过`<template>`元素包裹“用户名（Username）”和“邮箱（Email）”两组页面元素，后面会在这段代码的基础上进行功能扩展。
- 第 13~21 行的脚本代码中，通过“`new Vue()`”构造函数实例化 Vue 对象（`vm`）。其中，第 13 行代码通过 `el` 属性绑定了页面元素（"id-div-derectives-templ"）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 `vuederectives.html` 页面，页面效果如图 9.10 所示。

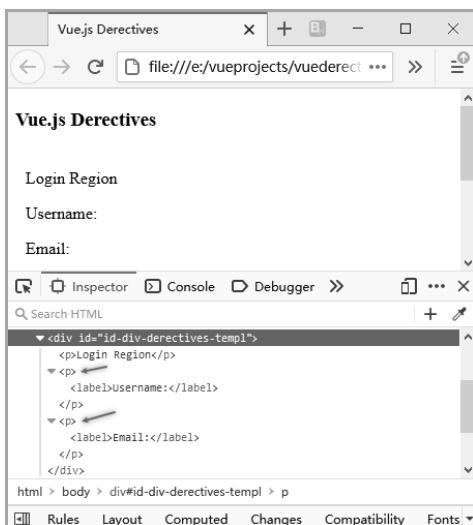


图 9.10 `<template>`元素应用（一）

如图 9.10 中的箭头所示，页面中显示了第 04 行和第 07 行代码定义的登录信息，但从浏览器调试窗口中是看不到<template>元素信息的。这一点印证了前文中关于<template>元素的介绍，该元素在最终的页面中不会被渲染出来。

那么，<template>元素在实际中如何使用呢？请看这个通过<template>元素切换显示登录信息的应用。

【代码 9-9】（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-templ">
02   <p>Login Region</p>
03   <template v-if="logintype">
04     <p><label>Username:</label></p>
05   </template>
06   <template v-else>
07     <p><label>Email:</label></p>
08   </template>
09   <p>
10     <button
11       id="id-btn-logintype"
12       onclick="on_btn_logintype(this.id)">
13       Change Login Type
14     </button>
15   </p>
16 </div>
17 <script>
18   // Vue Entry
19   var vm = new Vue({
20     el: '#id-div-derectives-templ',
21     data: {
22       logintype: true
23     }
24   })
25   // func - button click eventlogintype
26   function on_btn_logintype(thisid) {
27     vm.$data.logintype = !vm.$data.logintype;
28   }
29 </script>

```

### 【代码说明】

- 在第 03 行代码中，通过<template>元素包裹了一个“用户名（Username）”页面元素，并通过 v-if 指令判断对象（logintype）的布尔值，根据判断结果显示该“用户名（Username）”页面元素。
- 在第 07 行代码中，通过<template>元素包裹一个“邮箱（Email）”页面元素，并通过 v-else 指令根据前面 v-if 指令的判断结果显示该“邮箱（Email）”页面元素。
- 第 10~14 行代码中，通过<button>元素定义了一个按钮，并注册它的单击事件方法（on\_btn\_logintype()），用于执行切换对象（logintype）布尔值的操作。

- 第 19~24 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。其中，第 21~23 行代码通过 data 属性进行绑定数据操作，具体说明如下：
  - 第 22 行定义了一个布尔值属性（logintype），并初始化为布尔值（true），对应第 03 行代码中引用的对象（logintype）。
- 第 26~28 行定义的脚本代码是单击事件方法（on\_btn\_logintype()）的具体实现过程。其中，第 27 行代码通过 vm 对象的\$data 实例引用对象（logintype），并进行布尔值取反操作，从而实现在页面中动态切换显示登录信息的效果。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面初始效果如图 9.11 所示。

如图 9.11 中的箭头和标识所示，初始页面中只显示第 04 行代码定义的“用户名（Username）”信息。尝试单击“Change Login Type”按钮来改变登录信息，页面效果如图 9.12 所示。

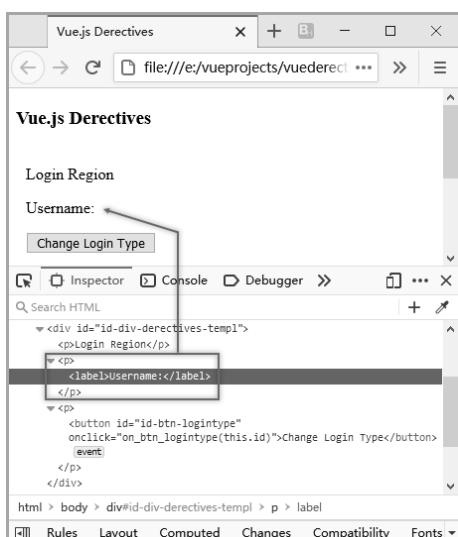


图 9.11 <template>元素应用（二）

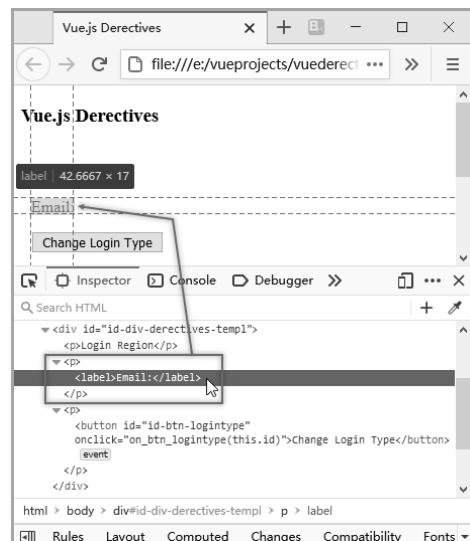


图 9.12 <template>元素应用（三）

如图 9.12 中的箭头和标识所示，单击“Change Login Type”按钮后，页面中的显示信息切换成第 07 行代码定义的“邮箱（Email）”信息。

另外，从图 9.11 和图 9.12 中可以看到，<template>元素在最终的页面中并没有被加载进去，在 Vue.js 框架中仅仅是作为虚拟元素使用的。

### 9.3.5 v-for 循环指令

既然 Vue.js 框架设计了 v-if 条件指令，自然也不会忽略掉 v-for 循环指令。这个 v-for 指令基本实现了 JavaScript 脚本语言的 for 循环语句功能，将循环表达式逻辑嵌入到 Vue 代码中去执行。

下面通过具体的代码实例进行讲解，请看这个通过 v-for 指令定义页面列表的应用。

## 【代码 9-10】（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-for">
02   <ul>
03     <li v-for="li in liArr">
04       {{ li.txt }}
05     </li>
06   </ul>
07 </div>
08 <script>
09   // Vue Entry
10  var vm = new Vue({
11    el: '#id-div-derectives-for',
12    data: {
13      liArr: [
14        { txt: 'JavaScript' },
15        { txt: 'Vue.js' },
16        { txt: 'Vue-cli' },
17        { txt: 'Vue router' },
18        { txt: 'Vuex' }
19      ]
20    }
21  })
22 </script>
23
24
25
26 </script>

```

## 【代码说明】

- 第 01~07 行代码中，在页面中通过<div>元素定义了一个层，并定义其 id 属性值（"id-div-derectives-for"）。具体说明如下：
  - 第 02~06 行代码中，通过<ol><li>元素定义了一个列表。
  - 第 03 行代码中，在<li>元素中通过 v-for 指令定义一个循环语句块（li in liArr）。其中，变量（li）是循环自变量，变量（liArr）是一个对象数组，变量（li）在对象数组（liArr）中迭代。
  - 第 04 行代码中，通过变量（li）迭代对象（liArr）的 txt 属性值，从而在页面中生成一个列表。
- 第 10~25 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 11 行代码中，通过 el 属性绑定 DOM 元素（"id-div-derectives-for"）。
  - 第 12~24 行代码中，通过 data 属性进行绑定数据操作。其中，第 13~23 行代码定义了一个对象数组（liArr）和一个 txt 属性，并进行初始化操作，对应第 04 行代码中引用的对象（li.txt）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.13 所示。

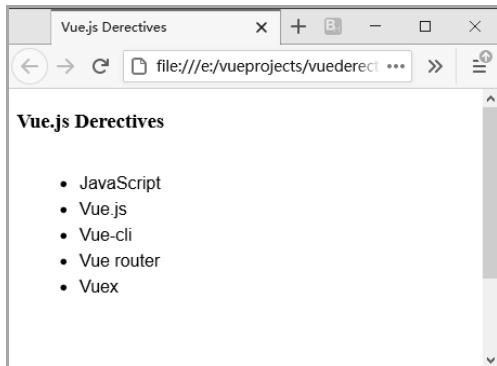


图 9.13 v-for 指令应用（一）

如图 9.13 中的箭头所示，页面中通过 v-for 循环指令，显示出由对象数组（btnArr）定义的一个列表。

通过上面的代码实例，我们已经体会到了 v-for 循环指令的强大。通常，在复杂的 HTML 页面设计中需要定义很多类型相同的元素，此时通过传统 JavaScript 脚本自动生成元素属性（如：id、class、style 等）的工作就会很烦琐。

而在 Vue.js 框架下，通过 v-for 循环指令自动生成这些元素属性就很方便，相信这也是优秀的 Vue.js 前端框架能够被广大设计人员喜爱的重要原因之一。下面通过具体的代码实例，讲解一下如何自动生成一组<button>元素的 id 属性的方法。

**【代码 9-11】**（详见源代码 vuederectives 目录中的 vuederectives.html 文件）

```

01 <div id="id-div-derectives-for">
02   <template>
03     <button v-for="btn in btnArr" v-bind:id="genId(btn.id)">
04       {{ btn.txt }}
05     </button>
06   </template>
07 </div>
08 <script>
09   // Vue Entry
10   var vm = new Vue({
11     el: '#id-div-derectives-for',
12     data: {
13       btnArr: [
14         { id: 1,
15           txt: 'JavaScript',
16         },
17         { id: 2,
18           txt: 'Vue.js',
19         },
20         { id: 3,
21           txt: 'Vue-cli',
22         }
23       ]
24     }
25   })
26 
```

```

22      },
23      id: 4,
24      txt: 'Vue router',
25    },
26    id: 5,
27    txt: 'Vuex',
28  ],
29},
30methods: {
31  genId: function(index) {
32    return "id-btn-" + index;
33  }
34}
35})
36</script>

```

### 【代码说明】

- 第 01 ~ 07 行代码中，在页面中通过<div>元素定义了一个层及其 id 属性值（"id-div-derectives-for"）。具体说明如下：
  - 第 02 ~ 06 行代码中，通过<template>元素包裹一个<button>按钮元素。
  - 第 03 行代码中，在<button>元素中通过 v-for 指令定义了一个循环语句块（btn in btnArr），变量（btn）是循环自变量，变量（btnArr）是一个对象数组，变量（btn）在对象数组（btnArr）中迭代。另外，通过 v-bind:id 指令绑定 id 属性，id 属性值通过一个自定义方法（genId()）自动获取。
  - 第 04 行代码中，通过变量（btn）迭代对象（btnArr）的 txt 属性值，从而在页面中自动生成一组按钮。
- 第 10 ~ 35 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 11 行代码中，通过 el 属性绑定 DOM 元素（"id-div-derectives-for"）。
  - 第 12 ~ 29 行代码中，通过 data 属性进行绑定数据操作。其中，第 13 ~ 28 行代码定义了一个对象数组（btnArr）、一个 id 属性和一个 txt 属性，并进行初始化操作。这里的 id 属性对应第 03 行代码中 genId()方法的参数（btn.id），txt 属性对应第 04 行代码中引用的对象（btn.txt）。
  - 第 31 ~ 33 行代码是 genId()方法的具体实现过程，通过参数返回按钮<button>元素的 id 属性值。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.14 所示。

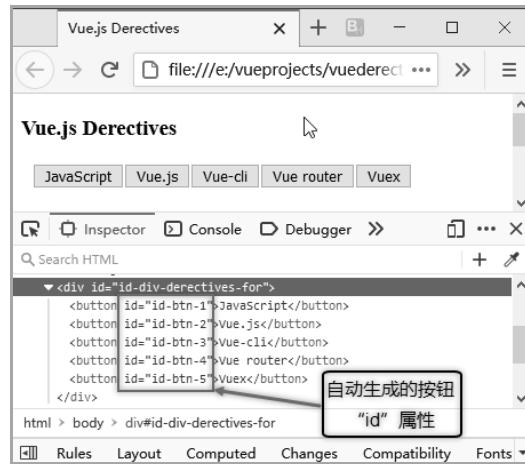


图 9.14 v-for 指令应用（二）

如图 9.14 中的箭头和标识所示，页面中显示了一组自动生成的按钮，在浏览器调试窗口中可查看到按钮自动生成的 id 属性值。

## 9.4 Vue.js 指令参数

本节介绍 Vue.js 框架中指令参数方面的内容。Vue.js 指令参数扩展了 Vue 指令的使用方式，可以实现更强大的页面功能。

### 9.4.1 Vue.js 指令接收参数

针对 Vue.js 框架中的一些指令，可以通过接收一个“参数”来绑定一些功能，这个接收的参数需要在指令名称之后用冒号（:）来连接。例如，在前面的【代码 9-10】中用到过绑定元素 id 属性，就是通过 v-bind 指令用冒号（:）连接 id 属性描述符（v-bind:id）来实现的。

下面介绍一个通过 v-bind 指令接收 href 参数，绑定超链接元素的地址属性的代码实例

**【代码 9-12】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-bind-href">
02   <p>Which language do you like, JavaScript or Vue.js?</p>
03   <p>I like <a v-bind:href="url">Vue.js</a>.</p>
04 </div>
05 <script>
06   // Vue Entry
07   var vm = new Vue({
08     el: '#id-div-derectives-bind-href',
09     data: {
10       url: "https://cn.vuejs.org"

```

```

11      }
12    })
13  </script>

```

### 【代码说明】

- 第 01~04 行代码中，在页面中通过<div>元素定义了一个层，并定义其 id 属性值（"id-div-derectives-bind-href"）。具体说明如下：
  - 第 03 行代码中，在<a>元素中通过 v-bind 指令接收一个参数 href，并绑定超链接<a>元素的地址属性，属性值为一个对象（url）。
- 第 07~12 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 08 行代码中，通过 el 属性绑定 DOM 元素（"id-div-derectives-bind-href"）。
  - 第 09~11 行代码中，通过 data 属性绑定数据操作。其中，第 10 行代码定义一个对象（url），并初始化为 Vue.js 框架的中文官方地址（"https://cn.vuejs.org"）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.15 所示。

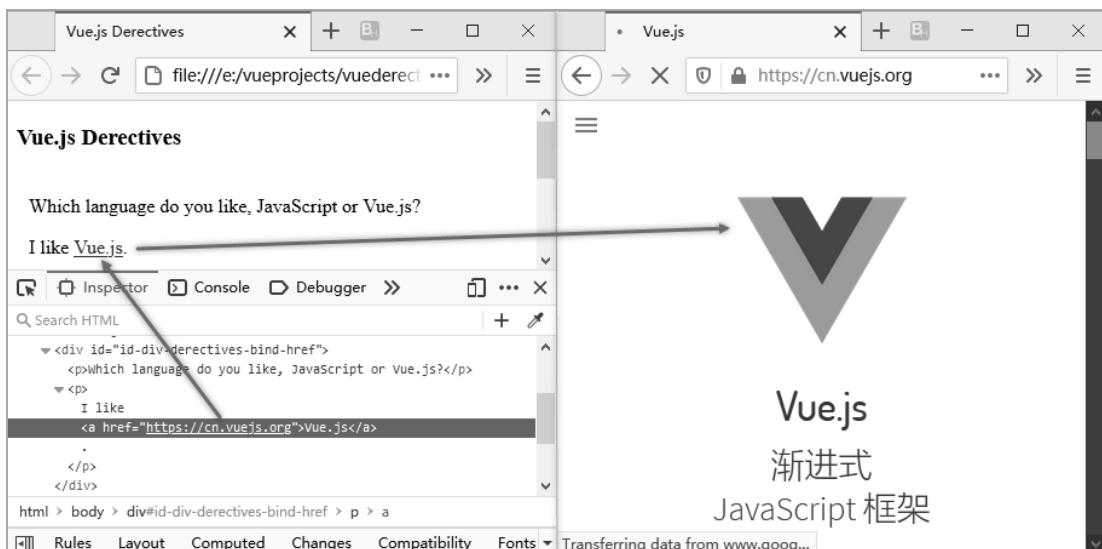


图 9.15 v-bind:href 指令应用

如图 9.15 中的箭头所示，页面中显示了第 03 行代码定义生成的超链接（目标地址见浏览器控制台中的信息），单击该超链接会跳转到目标地址（"https://cn.vuejs.org"）。

除了前面介绍的通过 v-bind 指令接收 id 属性参数，本节介绍的通过 v-bind 指令接收 href 属性参数，还有通过 v-on 指令绑定监听 click 事件，都属于通过 Vue.js 指令接受参数的范畴。设计人员在 Vue.js 框架下，可以大胆地使用这些指令来接受相关参数来实现绑定 HTML 元素属性的操作。

## 9.4.2 Vue.js 指令接收动态参数

上一小节介绍了 Vue.js 指令接收参数的用法，读者应该会注意到，这些接收的参数都是已经定义好的“静态”参数。其实，还可以将这些接收的参数设计成为“动态”的，就是通过 JS 表达式来定义这些参数。

将 Vue.js 指令接收的参数用 JS 表达式来定义，就可以事先不指定具体取值，而是事后通过表达式动态运算来获取具体值，从而实现“动态”参数的功能。Vue.js 指令接收的动态参数需要使用方括号（[]）来定义，方括号（[]）内为通过 JS 表达式表示的参数。

下面介绍一个通过 v-on 指令，在<input>元素上接收“动态”事件参数的代码实例。

**【代码 9-13】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-bind-input">
02     <p>Which language do you like, JavaScript or Vue.js?</p>
03     <p>I like
04     <input type="text" v-on:[type?focus:change]="event" value="Vue.js"/>.
05     </p>
06     <button v-on:click="type=true">event:focus</button>
07     <button v-on:click="type=false">event:change</button>
08 </div>
09 <script>
10     // Vue Entry
11     var vm = new Vue({
12         el: '#id-div-derectives-bind-input',
13         data: {
14             type: null,
15             focus: 'focus',
16             change: 'change'
17         },
18         methods: {
19             event: function() {
20                 console.log("Emit " +
21                     (this.type ? 'focus' : 'change') +
22                     " event.");
23             }
24         }
25     })
26 </script>
```

**【代码说明】**

- 第 01~08 行代码中，在页面中通过<div>元素定义了一个层，并定义其 id 属性值（"id-div-derectives-bind-input"）。具体说明如下：
  - 第 04 行代码中，在<input type="text">元素中通过 v-on 指令接收一个动态参数 “[type ? focus : change]”，通过判断对象“type”的布尔值来选取具体事件（focus 事件或者 change 事件），参数值为一个事件方法（event）。

- 第 06、07 行代码定义一组<button>按钮元素，通过 v-on 指令接收单击事件 click 参数，用于切换对象（type）的布尔值。
- 第 11~25 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 13~17 行代码中，通过 data 属性进行绑定数据操作。其中，第 14 行代码定义一个对象（type），并初始化为空（null）值。第 15 行代码定义一个对象（focus），初始化为 focus 事件名称。第 16 行代码定义一个对象（change），并初始化为 change 事件名称。
  - 第 18~24 行代码中，通过 methods 属性进行绑定方法操作。其中，第 19~23 行代码是事件方法（event）的具体实现，通过判断对象（type）的布尔值向浏览器控制台中输出相应的日志信息。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面初始效果如图 9.16 所示。

如图 9.16 中的箭头所示，尝试在页面中单击文本输入框，使其获取用户输入焦点（focus），页面效果如图 9.17 所示。

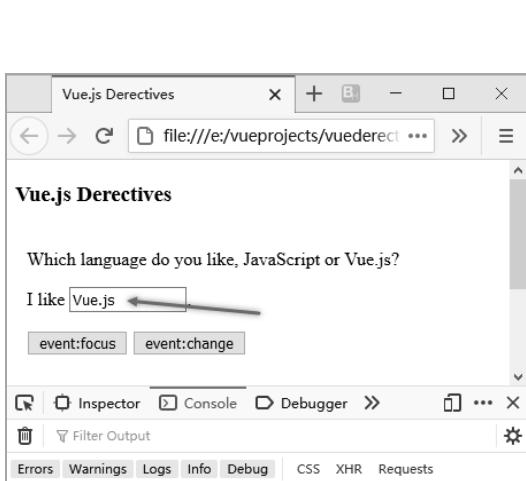


图 9.16 v-bind 指令接收动态参数应用（一）

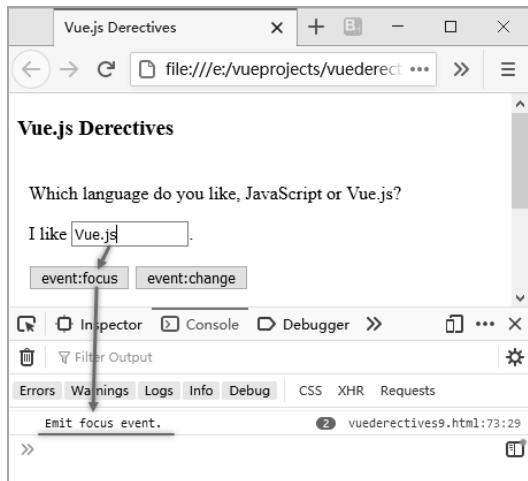


图 9.17 v-bind 指令接收动态参数应用（二）

如图 9.17 中的箭头所示，浏览器控制台中输出了文本输入框获取用户输入焦点事件的日志信息。假如上述操作无信息反馈，可以先单击“event:focus”按钮，将当前<button>按钮的响应事件切换到 focus 事件上。

继续单击“event:change”按钮，将当前<button>按钮的响应事件切换到 change 事件上。然后，尝试在页面中修改文本输入框中的文本内容，触发其文本改变事件（change），页面效果如图 9.18 所示。

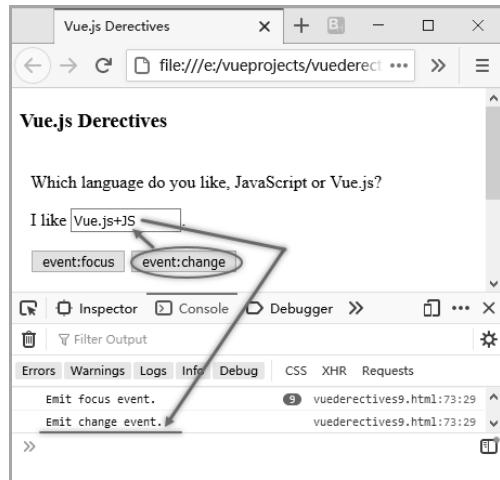


图 9.18 v-bind 指令接收动态参数应用（三）

如图 9.18 中的箭头和标识所示，当在页面中的文本输入框中修改文本后，浏览器控制台中输出了文本改变事件（change）的日志信息。

### 9.4.3 通过 Vue.js 指令动态参数改变元素类型

Vue.js 指令接收动态参数的使用方式非常灵活，除了动态修改事件类型外，还可以动态修改元素类型。

下面介绍一个通过 Vue.js 指令接收动态参数的方式，动态切换文本输入框和按钮的代码实例。

**【代码 9-14】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-bind-input">
02   <p>Which language do you like, JavaScript or Vue.js?</p>
03   <p>I like
04     <input
05       v-bind:[attr.type.name]="attr.type.val"
06       v-bind:value="attr.value" />.
07   </p>
08   <p>
09     <button v-on:click="event(true)">type:text</button>
10     <button v-on:click="event(false)">type:button</button>
11   </p>
12 </div>
13 <script>
14   // Vue Entry
15   var vm = new Vue({
16     el: '#id-div-derectives-bind-input',
17     data: {
18       // property - attr
19       attr: {
20         type: {

```

```

21             name: "type",
22             val: "text"
23         },
24         value: "Vue.js"
25     }
26 },
27 methods: {
28     // function - event
29     event: function(b) {
30         if (b) {
31             this.attr.type.name = 'type';
32             this.attr.type.val = 'text';
33             this.attr.value = 'Vue.js';
34         } else {
35             this.attr.type.name = 'type';
36             this.attr.type.val = 'button';
37             this.attr.value = 'Vue Button';
38         }
39     }
40 }
41 })
42 </script>

```

### 【代码说明】

- 第 01~12 行代码中，在页面中通过<div>元素定义了一个层，并定义其 id 属性值（"id-div-derecives-bind-input"）。具体说明如下：
  - 第 04~06 行代码中，在<input>元素中通过 v-bind 指令接收了一个动态参数对象 [attr.type.name]，参数值为对象（attr.type.val）。还通过 v-bind 指令接收一个参数 value，参数值为对象（attr.value）。
  - 第 09 行和第 10 行代码定义了一组<button>按钮元素，通过 v-on 指令接收单击事件 click 参数，参数值为事件方法（event），通过在调用该方法时传递布尔值参数来实现元素类型的切换功能。
- 第 15~41 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 17~26 行代码中，通过 data 属性进行绑定数据操作。其中，第 19~25 行代码定义一个 JSON 对象（attr），该对象内部定义了切换<input>元素类型所需的参数属性。
  - 第 27~40 行代码中，通过 methods 属性进行绑定方法操作。其中，第 29~39 行代码是事件方法（event）的具体实现，通过判断参数（b）的布尔值实现<input>元素类型的动态切换操作。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面初始效果如图 9.19 所示。

如图 9.19 中的箭头所示，页面中初始显示的是一个文本输入框元素。然后，尝试在页面

中单击“type:button”按钮来切换元素类型，页面效果如图 9.20 所示。

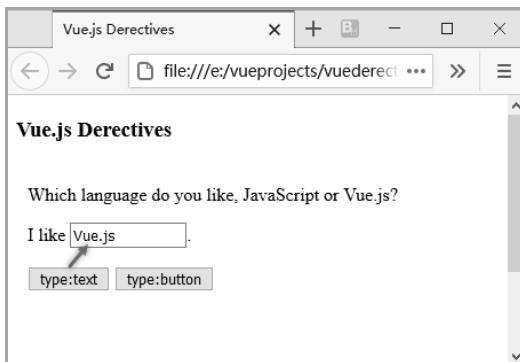


图 9.19 v-bind 指令接收动态参数切换元素类型（一）

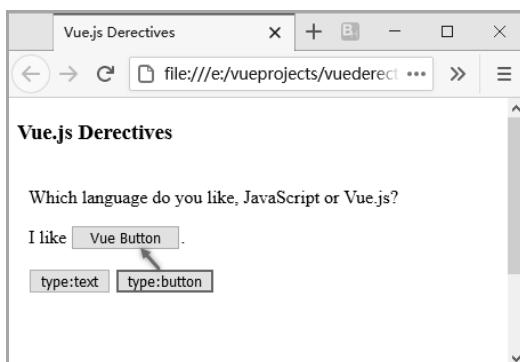


图 9.20 v-bind 指令接收动态参数切换元素类型（二）

如图 9.20 中的箭头所示，页面中所显示的元素已经由一个文本输入框元素切换为一个按钮了。

## 9.5 Vue.js 指令修饰符

本节介绍 Vue.js 框架中指令修饰符方面的内容。Vue.js 指令修饰符扩展了 Vue 指令的使用方式，可以通过更简洁的方式实现传统的页面功能。Vue.js 框架中定义了很多指令修饰符，这里介绍几个比较常用的修饰符来帮助读者学习理解。

### 9.5.1 Vue.js 指令 prevent 修饰符

在 Vue.js 指令中使用 prevent 修饰符可以阻止控件元素的默认行为，相当于调用了 event.preventDefault()方法。例如，表单的提交（Submit）行为和超链接元素的跳转行为，就是该控件元素的默认事件行为。

在 Vue.js 指令中使用修饰符，需要在指令名称之后用英文句号（.）来连接，类似于引用对象属性的方式。例如，针对表单的提交（Submit）行为使用 prevent 修饰符，就要写成（v-on:submit.prevent）的形式。针对超链接元素的跳转行为使用 prevent 修饰符，就要写成（v-on:click.prevent）的形式。

下面介绍一个通过在 Vue.js 指令中使用 prevent 修饰符，阻止超链接元素默认跳转行为的代码实例。

### 【代码 9-15】（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-prevent">
02     <p>Which language do you like, JavaScript or Vue.js?</p>
03     <p>I like
04         <a name="language"
05             v-bind:href="url"
06             v-on:click.prevent="clk_prevent">
07             {{ language }}
08         </a>.
09     </p>
10 </div>
11 <script>
12 // Vue Entry
13 var vm = new Vue({
14     el: '#id-div-derectives-prevent',
15     data: {
16         url: "https://cn.vuejs.org",
17         language: "Vue.js"
18     },
19     methods: {
20         clk_prevent: function() {
21             console.log("href is prevented by modifier '.prevent'!");
22         }
23     }
24 })
25 </script>

```

### 【代码说明】

- 第 01~10 行代码中，在页面中通过<div>元素定义了一个层，并定义其 id 属性值（"id-div-derectives-prevent"）。具体说明如下：
  - 第 04~08 行代码中，通过<a>元素定义了一个超链接。其中，第 05 行代码通过 v-bind 指令接收一个参数 href，并绑定超链接<a>元素的地址属性，属性值为一个对象（url）。第 06 行代码通过 v-on 指令绑定单击 click 事件，并定义 prevent 修饰符，事件方法名称为 clk\_prevent。
- 第 13~24 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 14 行代码中，通过 el 属性绑定 DOM 元素（"id-div-derectives-prevent"）。

- 第 15~18 行代码中，通过 data 属性进行绑定数据操作。其中，第 16 行代码定义了一个对象（url），并初始化为 Vue.js 框架的中文官方地址（"https://cn.vuejs.org"）。
- 第 19~23 行代码中，通过 methods 属性进行绑定方法操作。其中，第 20~22 行代码是事件方法（clk\_prevent）的具体实现，第 04~08 行代码定义的超链接元素所默认的跳转地址被阻止后会调用该方法。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuedirectives.html 页面，页面初始效果如图 9.21 所示。

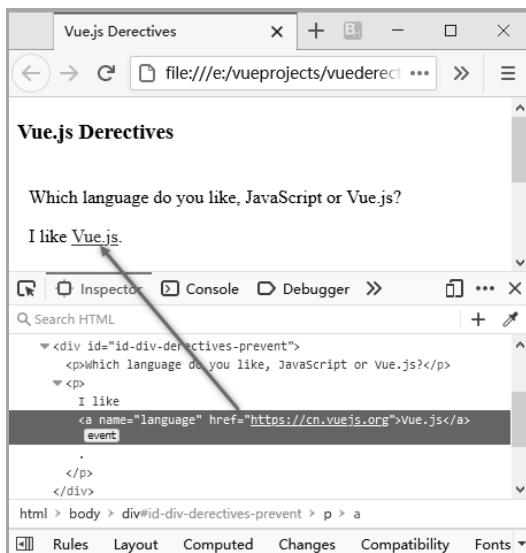


图 9.21 在超链接元素上使用 prevent 修饰符（一）

如图 9.21 中的箭头所示，我们尝试单击页面中的超链接，预期会跳转到浏览器控制台中显示的目标地址（"https://cn.vuejs.org"）上去。但是，实际的结果会怎么样呢？页面效果如图 9.22 所示。

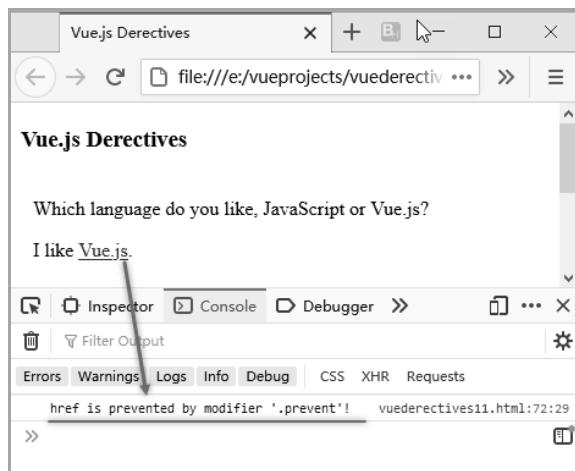


图 9.22 在超链接元素上使用 prevent 修饰符（二）

如图 9.22 中的箭头所示，页面并没有跳转到超链接的目标地址上去，而是在浏览器控制台中打印了一行由第 21 行代码定义的日志信息。以上结果说明：在第 06 行代码中，通过 v-on 指令绑定的单击 click 事件被 prevent 修饰符成功阻止了。

下面再介绍一个通过在 Vue.js 指令中使用 prevent 修饰符，阻止表单<form>元素默认提交行为的代码实例。

**【代码 9-16】**（详见源代码 vuoderectives 目录中的 vuoderectives.html 文件）

```

01 <div id="id-div-derectives-prevent">
02   <form
03     v-on:submit.prevent="submit_prevent"
04     action="server.php"
05     method="GET">
06     <p>Which language do you like, JavaScript or Vue.js?</p>
07     <p>I like
08     <input type="text" name="language" v-bind:value="language">.
09     </p>
10     <input type="submit" value="Submit" />
11   </form>
12 </div>
13 <script>
14   // Vue Entry
15   var vm = new Vue({
16     el: '#id-div-derectives-prevent',
17     data: {
18       language: 'Vue.js'
19     },
20     methods: {
21       submit_prevent: function() {
22         console.log("submit is prevented by modifier '.prevent'!");
23       }
24     }
25   })
26 </script>

```

**【代码说明】**

- 第 02~11 行代码中，通过<form>元素定义了一个表单，具体说明如下：
  - 第 03 行代码中，通过 v-on 指令接收参数 submit，并绑定表单的提交操作。同时，在参数 submit 上定义 prevent 修饰符，事件方法名称为 ( submit\_prevent )。
  - 第 04 行代码中，通过表单的 action 属性定义提交的服务器端文件为 server.php。
  - 第 05 行代码中，通过表单的 method 属性定义提交方式为 GET。
  - 第 08 行代码中，通过<input type="text">元素定义一个文本输入框，并通过“v-bind”指令接收参数 value，初始化为对象 ( language ) 的值。
  - 第 08 行代码中，通过<input type="submit">元素定义一个表单提交按钮。
- 第 15~25 行的脚本代码中，通过 “new Vue()” 构造函数实例化 Vue 对象 ( vm )。

具体说明如下：

- 第 17~19 行代码中，通过 data 属性进行绑定数据操作。其中，第 18 行代码定义了一个对象（language），并初始化为字符串“Vue.js”。
- 第 20~24 行代码中，通过 methods 属性进行绑定方法操作。其中，第 21~23 行代码是事件方法（submit\_prevent）的具体实现，通过第 22 行代码向浏览器控制台输出了一行日志信息。

为了测试如何通过 prevent 修饰符阻止表单<form>的默认提交行为，我们先将第 03 行代码注释掉。然后，通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.23 所示。



图 9.23 在表单<form>元素上使用 prevent 修饰符（一）

如图 9.23 中的箭头和标识所示，我们尝试单击左侧页面中的提交按钮（Submit），可以看到表单中文本输入框中的信息被成功提交到服务器页面上去了。

图 9.23 中的页面效果是完全符合 HTML 表单提交的操作结果的，如果将被注释的第 03 行代码加入运行后会怎么样呢？下面，再次通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.24 所示。

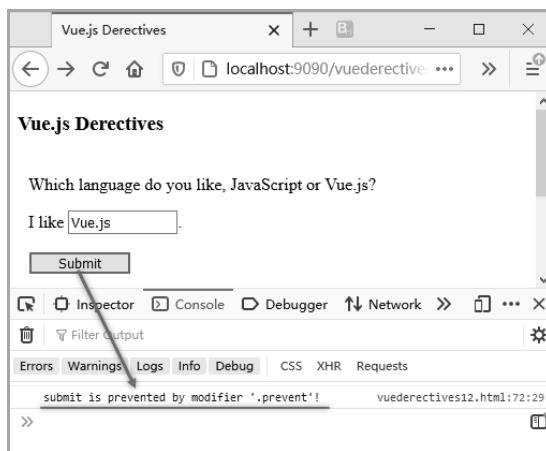


图 9.24 在表单<form>元素上使用 prevent 修饰符（二）

如图 9.24 中的箭头和标识所示，当我们尝试单击页面中的提交按钮（Submit）后，并没

有看到表单中文本输入框中的信息被提交到服务器页面上，而是被 prevent 修饰符阻止，并在浏览器控制台中输出第 22 行代码定义的日志信息。

### 9.5.2 Vue.js 指令 stop 修饰符

在 Vue.js 指令中使用 stop 修饰符可以阻止事件冒泡，相当于调用了 event.stopPropagation() 方法。关于 JavaScript 事件体系中冒泡原理就不深入介绍了，这里主要讲解一下 stop 修饰符的使用方法。

下面介绍一个通过在 Vue.js 指令中使用 stop 修饰符，阻止多层父子关系的<div>元素事件冒泡行为的代码实例。

**【代码 9-17】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-stop">
02     <div id="outer" v-on:click="clk_outer">
03         {{ outerTxt }}
04         <div
05             id="inner"
06             v-on:click="clk_inner"
07             v-on:click.stop="clk_stop_inner">
08             {{ innerTxt }}
09         </div>
10     </div>
11 </div>
12 <script>
13     // Vue Entry
14     var vm = new Vue({
15         el: '#id-div-derectives-stop',
16         data: {
17             outerTxt: "JavaScript",
18             innerTxt: "Vue.js"
19         },
20         methods: {
21             clk_outer: function() {
22                 console.log("you have clicked outer div.");
23             },
24             clk_inner: function() {
25                 console.log("you have clicked inner div.");
26             },
27             clk_stop_inner: function() {
28                 console.log("you have stopped inner div click event.");
29             }
30         }
31     })
32 </script>
```

**【代码说明】**

- 第 02~10 行代码中，在页面中通过<div>元素定义了具有父子关系的两个层，并分别

定义其 id 属性值（`id="outer"`和`id="inner"`）。具体说明如下：

- 在第 02 行代码定义的父级`<div id="outer">`层中，通过`v-on`指令绑定单击`click`事件，事件方法名称为`clk_outer`。
- 第 04~09 行代码定义的子级`<div id="inner">`层中，第 06 行代码通过`v-on`指令绑定单击`click`事件，事件方法名称为`clk_inner`。第 07 行代码再次通过`v-on`指令绑定单击`click`事件，并定义`stop`修饰符，事件方法名称为（`clk_stop_inner`）。
- 第 20~30 行代码中，通过`methods`属性进行绑定方法操作。具体说明如下：
  - 第 21~23 行代码是事件方法（`clk_outer`）的具体实现，其中第 22 行代码中向浏览器控制台中输出了一行日志信息。
  - 第 24~26 行代码是事件方法（`clk_inner`）的具体实现，其中第 25 行代码中向浏览器控制台中输出了一行日志信息。
  - 第 27~29 行代码是事件方法（`clk_stop_inner`）的具体实现，其中第 28 行代码中向浏览器控制台中输出了一行日志信息。

为了测试如何通过`stop`修饰符阻止事件的默认冒泡行为，我们先将第 07 行代码注释掉。然后，通过 VS Code 开发工具启动 FireFox 浏览器，运行测试`vuedirectives.html`页面，页面效果如图 9.25 所示。



图 9.25 在层`<div>`元素上使用`stop`修饰符（一）

如图 9.25 中的箭头和标识所示，我们尝试单击父级`<div id="outer">`层的区域，浏览器控制台中输出了第 21~23 行代码中事件方法（`clk_outer`）定义的日志信息。然后，我们再尝试单击子级`<div id="inner">`层的区域，页面效果如图 9.26 所示。



图 9.26 在层&lt;div&gt;元素上使用 stop 修饰符（二）

如图 9.26 中的箭头和标识所示，我们尝试单击子级`<div id="inner">`层的区域，浏览器控制台中先输出了第 24~26 行代码中事件方法（`clk_inner`）定义的日志信息，然后再次输出了第 21~23 行代码中事件方法（`clk_outer`）定义的日志信息。该效果就是 JavaScript 的事件冒泡原理所产生的，事件会逐级向上“冒泡”传递，直到“根节点”才会结束。

如果我们将被注释掉的第 07 行代码恢复运行，效果会如何呢？我们再次通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 `vuedirectives.html` 页面，页面效果如图 9.27 所示。

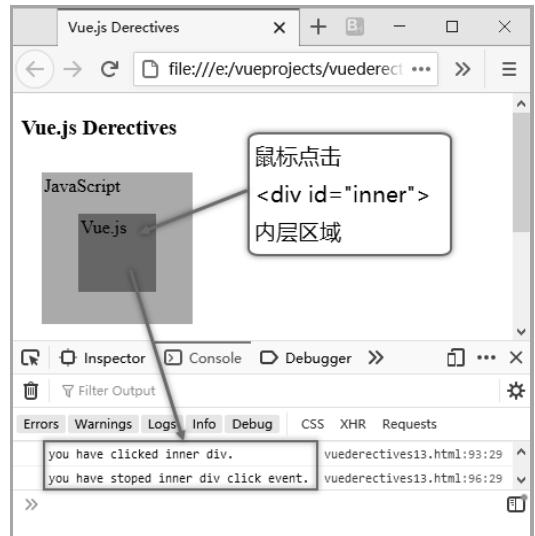


图 9.27 在层&lt;div&gt;元素上使用 stop 修饰符（三）

如图 9.27 中的箭头和标识所示，我们尝试单击子级`<div id="inner">`层的区域，浏览器控制台中输出了第 24~26 行代码中事件方法（`clk_inner`）定义的日志信息，接着又输出第 27~

29 行代码中事件方法（clk\_stop\_inner）定义的日志信息。然而，我们发现并没有出现图 9.26 所示的效果，单击 click 事件并没有传递到父级<div id="outer">层的区域，说明事件冒泡被 stop 修饰符事件阻止了。

### 9.5.3 Vue.js 指令 once 修饰符

在 Vue.js 指令中使用 once 修饰符可以强制执行仅仅一次有效的事件行为，再一次之后就不再起作用了。

下面介绍一个通过在 Vue.js 指令中使用 once 修饰符，模拟实现一个网络投票器的代码实例。

**【代码 9-18】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-once">
02     <span>Please choose your favorite language.</span>
03     <template>
04         <button v-on:click.once="clk once js">JavaScript</button>
05         <button v-on:click.once="clk once node">Node.js</button>
06         <button v-on:click ="clk vue">Vue.js</button>
07     </template>
08     <span>Your selected favorite language:<br>
09         <b v-html="language"></b>
10     </span>
11 </div>
12 <script>
13     // Vue Entry
14     var vm = new Vue({
15         el: '#id-div-derectives-once',
16         data: {
17             language: ""
18         },
19         methods: {
20             clk_once_js: function() {
21                 this.language += " &nbsp;&nbsp;JavaScript<br>";
22                 console.log("you have clicked JavaScript.");
23             },
24             clk_once_node: function() {
25                 this.language += " &nbsp;&nbsp;Node.js<br>";
26                 console.log("you have clicked Node.js.");
27             },
28             clk_vue: function() {
29                 this.language += " &nbsp;&nbsp;Vue.js<br>";
30                 console.log("you have clicked Vue.js.");
31             }
32         }
33     })
34 </script>
```

### 【代码说明】

- 第 01~11 行代码中，在页面中通过<div>元素定义了一个层，并分别定义其 id 属性值（id="id-div-derecives-once"）。具体说明如下：
  - 在第 03~07 行代码定义的<template>元素中，定义一组按钮<button>元素，分别用于用户选择 3 种脚本语言（JavaScript、Node.js 和 Vue.js）。在前两个按钮<button>元素中，通过 v-on 指令绑定单击 click 事件，并定义了 once 修饰符及其对应事件方法。而第三个按钮<button>元素中，通过 v-on 指令绑定单击 click 事件，但没有使用 once 修饰符（这样就可以直观地看到是否使用 once 修饰符的对比效果）。
  - 第 08~10 行代码中，通过 v-html 指令引用一个对象（language），用于输出用户的选择。
- 第 14~33 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 15 行代码中，通过 el 属性绑定 DOM 元素（"id-div-derecives-once"）。
  - 第 16~18 行代码中，通过 data 属性进行绑定数据操作。其中，第 17 行代码定义了一个对象（language），并初始化为空字符串，对应第 09 行代码引用的对象（language）。
  - 第 19~32 行代码中，通过 methods 属性进行绑定方法操作，定义了每个按钮<button>元素中各自单击 click 事件所对应的方法。在每个事件方法内，更新对象（language）的取值，并向浏览器控制台中输出一行日志信息。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.28 所示。

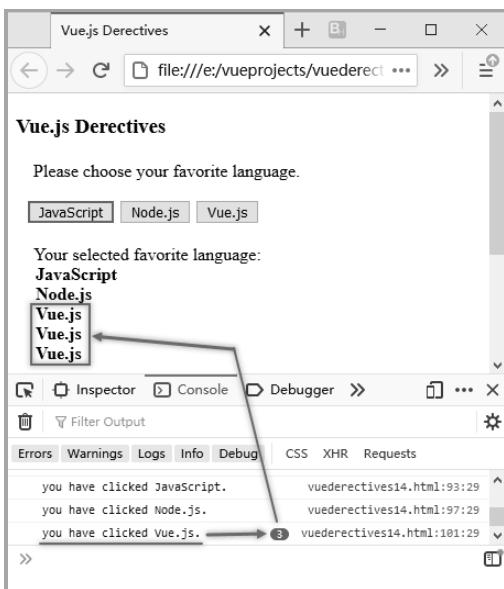


图 9.28 在按钮<button>元素上使用 once 修饰符

如图 9.28 中的箭头和标识所示，当单击按钮（JavaScript）和按钮（Node.js）时，浏览器控制台中在输出一次日志信息后就再没有任何反馈了。当单击按钮（Vue.js）时，浏览器控制台中输出每一次操作后日志信息（数量显示单击了 3 次）。以上区别说明在使用 once 修饰符后，事件行为被强制限制为仅仅使用一次有效。

## 9.6 Vue.js 指令缩写

本节介绍 Vue.js 框架中指令缩写方面的内容。设计人员通过指令缩写可以使用更简洁的方式来使用指令，这对于基于 Vue.js 框架构建的单页面应用程序(Single Page Application, SPA)起着令人惊喜的效果。

什么是 Vue.js 指令缩写呢？在前面我们所学习到的指令中，均是通过“v-”前缀来定义的。使用“v-”前缀的作用就是给出识别提示，帮助 Vue.js 框架来解析模板中具有特定行为的属性。而且，“v-”前缀具有很好的视觉提示效果，设计人员在查看代码时一眼就可以判断出这是 Vue 代码。

然而，虽然“v-”前缀的作用多多，但对于一些频繁用到指令的场景就会感到很烦琐。尤其是在基于 Vue.js 框架所构建的单页面应用程序中，“v-”前缀也是可用可不用的。因此，Vue.js 框架为最常用的这两个 v-bind 和 v-on 指令提供了简写支持，对于 v-bind 指令可以直接省略，而对于 v-on 指令可以用“@”符号替代。

下面介绍一个通过 Vue.js 指令缩写接收 href 参数实现超链接的代码实例。

**【代码 9-19】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-bind-href">
02   <p>Which language do you like, JavaScript or Vue.js?</p>
03   <p>I like <a v-bind:href="url">Vue.js</a>.</p>
04   <p>I like <a :href="url">Vue.js</a>.</p>
05 </div>
06 <script>
07   // Vue Entry
08   var vm = new Vue({
09     el: '#id-div-derectives-bind-href',
10     data: {
11       url: "https://cn.vuejs.org"
12     }
13   })
14 </script>
```

**【代码说明】**

- 第 01~05 行代码中，在页面中通过<div>元素定义了一个层，具体说明如下：
  - 在第 03 行代码中，通过<a>元素定义了一个超链接。其中，使用 v-bind 指令接收

href 参数，并绑定对象（url）。这是一个通过标准的 v-bind 指令绑定超链接地址的写法。

- 第 04 行代码中，还是通过<a>元素定义了一个超链接。注意，这里使用 v-bind 指令的缩写方式来接收 href 参数（:href），同样绑定了对象（url）。这样，通过将第 04 行代码与第 03 行代码进行对比，来验证一下 v-bind 指令缩写完成与标准 v-bind 指令同样的功能。
- 第 08~13 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 10~12 行代码中，通过 data 属性进行绑定数据操作。其中，第 11 行代码定义了一个对象（url），并初始化为 Vue.js 中文官方网址，对应第 03、04 行代码引用的对象（url）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.29 所示。



图 9.29 使用 v-bind 指令缩写

如图 9.29 中的箭头和标识所示，第 03 行代码中通过标准 v-bind 指令绑定的超链接地址，与第 04 行代码通过 v-bind 指令缩写绑定的超链接地址是完全相同的。而且，v-bind 指令无论是标准方式还是缩写方式，在最终的页面代码中都不体现出来。

下面介绍一个通过 Vue.js 指令缩写接收 click 参数，实现单击事件处理的代码实例。

**【代码 9-20】**（详见源代码 vuederectives 目录中的 vuederectives.html 文件）

```
01 <div id="id-div-derectives-bind-click">
```

```

02     <p>Which language do you like, JavaScript or Vue.js?</p>
03     <p>
04         <button v-on:click="clk_event_js">JavaScript</button>
05         <button @click="clk_event_vue">Vue.js</button>
06     </p>
07     <p>I like <a :href="url">{{language}}</a>.</p>
08 </div>
09 <script>
10     // Vue Entry
11     var vm = new Vue({
12         el: '#id-div-directives-bind-click',
13         data: {
14             url: "",
15             language: ""
16         },
17         methods: {
18             clk_event_js: function() {
19                 console.log("javascret button clicked.");
20                 this.url = "https://www.javascript.com/";
21                 this.language = "JavaScript";
22             },
23             clk_event_vue: function() {
24                 console.log("vue button clicked.");
25                 this.url = "https://cn.vuejs.org";
26                 this.language = "Vue.js";
27             }
28         }
29     })
30 </script>

```

### 【代码说明】

- 第 01~08 行代码中，在页面中通过<div>元素定义了一个层，具体说明如下：
  - 在第 04 行代码中，通过<button>元素定义了一个按钮。其中，使用 v-on 指令绑定单击（click）事件，并定义事件处理方法（clk\_event\_js）。这是一个通过标准的 v-on 指令绑定事件的写法。
  - 在第 05 行代码中，还是通过<button>元素定义了一个按钮。注意，这里使用 v-on 指令的缩写方式（@符号）来绑定单击（click）事件，同样定义事件处理方法（clk\_event\_vue）。
  - 在第 07 行代码中，通过<a>元素定义了一个超链接。其中，使用 v-bind 指令缩写方式接收 href 参数，并绑定对象（url）。另外，使用文本插值方式引用一个对象（language），用于显示用户通过单击按钮所选择的编程语言名称（JavaScript 或 Vue.js）。
- 第 11~29 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 13~16 行代码中，通过 data 属性进行绑定数据操作。其中，第 14 行代码定义

了一个对象（url），对应第 07 行代码引用的对象（url）。第 15 行代码定义了一个对象（language），对应第 07 行代码引用的对象（language）。

- 第 17~28 行代码中，通过 methods 属性进行绑定方法操作，定义了上面两个按钮<button>元素中的单击 click 事件处理方法（clk\_event\_js 和 clk\_event\_vue）。在每个事件方法内，均更新了各自对象（url 和 language）的取值，并向浏览器控制台中输出一行日志信息。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuedirectives.html 页面，页面初始效果如图 9.30 所示。

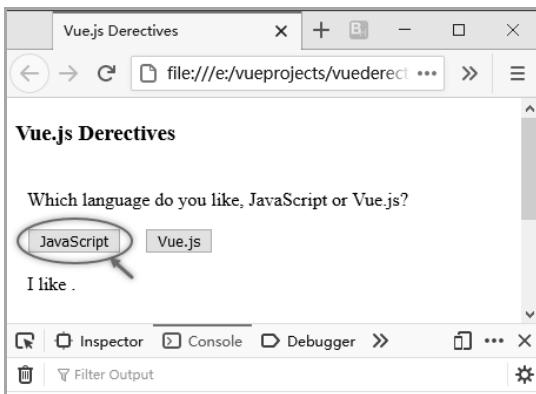


图 9.30 使用 v-on 指令缩写（一）

如图 9.30 中的箭头和标识所示，我们尝试单击一下按钮（JavaScript），页面效果如图 9.31 所示。

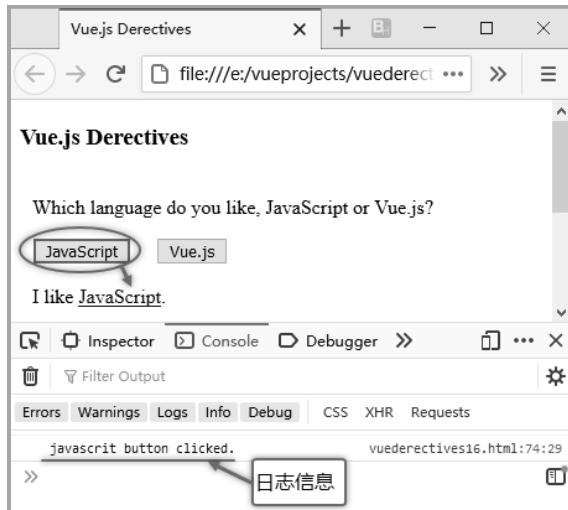


图 9.31 使用 v-on 指令缩写（二）

如图 9.31 中的箭头和标识所示，在单击按钮（JavaScript）后，页面更新显示用户选择的“JavaScript”信息。然后，我们尝试单击一下第二个按钮（Vue.js），页面效果如图 9.32 所示。

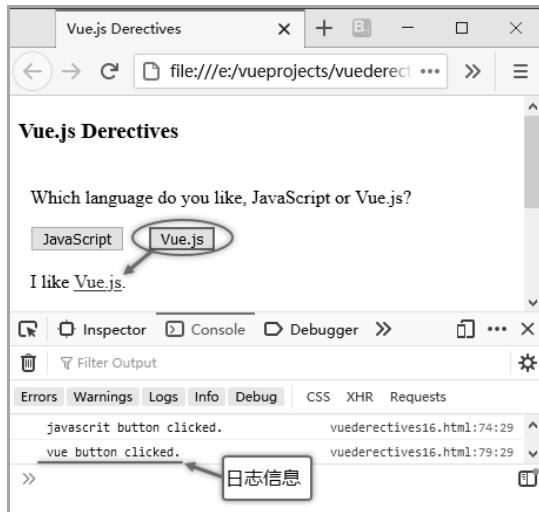


图 9.32 使用 v-on 指令缩写（三）

如图 9.32 中的箭头和标识所示，在单击按钮(Vue.js)后，页面更新显示用户选择的“Vue.js”信息。这说明，第 05 行代码中使用 v-on 指令的缩写方式(@符号)绑定的单击(click)事件，与第 04 行代码中使用标准 v-on 指令绑定的单击(click)事件，二者在功能上完全一致。

## 9.7 Vue.js 数据双向绑定

本节介绍 Vue.js 框架中数据双向绑定方面的内容。数据双向绑定功能通过 v-model 指令来实现，Vue.js 框架的该项功能是其能够成为目前最优秀前端框架的重要基础之一。

### 9.7.1 v-model 指令原理

在 Vue.js 框架的指令系统中，v-model 是其中最特殊、也是最受欢迎的指令之一。相信大部分读者都清楚，Vue.js 框架的核心特性之一就是数据的双向绑定功能，Vue.js 的响应式原理就是实现了“数据→视图”与“数据←视图”的同步更新功能。

Vue.js 的数据双向绑定功能通过 v-model 指令实现，该指令限制在<input>元素、<select>元素、<textarea>元素和 components 组件（后面章节会介绍）中使用。一般会通过修饰符（例如：.lazy、.number 和.trim 等）来配合使用。其实，v-model 指令本质上就是一个语法糖。下面这个代码实例就是通过 v-model 指令，实现一个非常简单的数据双向绑定功能。

**【代码 9-21】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```
01 <div id="id-div-derectives-model-msg">
02   <p>Test v-model derectives</p>
03   <p>Please enter
```

```

04      <input
05          type="text"
06          v-model="msg"
07          @input="input_msg"
08          placeholder="pls enter..." />.
09    </p>
10    <p>Your enter message: {{msg}}</p>
11  </div>
12  <script>
13      // Vue Entry
14  var vm = new Vue({
15      el: '#id-div-derectives-model-msg',
16      data: {
17          msg: ""
18      },
19      methods: {
20          input_msg: function() {
21              console.log("msg changed: " + this.msg);
22          }
23      }
24  });
25 </script>

```

### 【代码说明】

- 第 01~11 行代码中，在页面中通过<div>元素定义了一个层，具体说明如下：
  - 在第 04~08 行代码中，通过<input type="text">元素定义了一个文本输入框，用于测试双向绑定的功能。其中，第 06 行代码通过 v-model 指令双向绑定数据对象 (msg)。第 07 行代码通过 v-on 指令缩写绑定 input 输入事件方法 (input\_msg)。
  - 第 10 行代码中，使用文本插值方式引用一个对象 (msg)，用于同步显示用户在第 04~08 行代码定义的文本输入框中所输入的内容。
- 第 14~24 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象 (vm)。具体说明如下：
  - 第 16~18 行代码中，通过 data 属性进行绑定数据操作。其中，第 17 行代码定义了一个数据对象 (msg)，分别对应第 06 行和第 10 行代码引用的数据对象 (msg)。
  - 第 20~22 行代码是第 07 行代码中 input 输入事件处理方法 (input\_msg) 的具体实现，第 21 行代码向浏览器控制台中输出了一行日志信息。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面初始效果如图 9.33 所示。

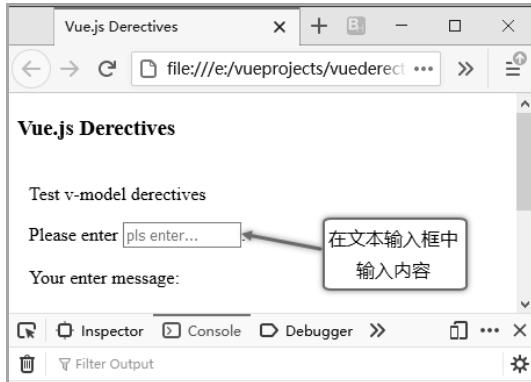


图 9.33 使用 v-model 指令实现数据双向绑定（一）

如图 9.33 中的箭头和标识所示，我们尝试在文本输入框中输入一些内容（例如：vue.js），页面效果如图 9.34 所示。

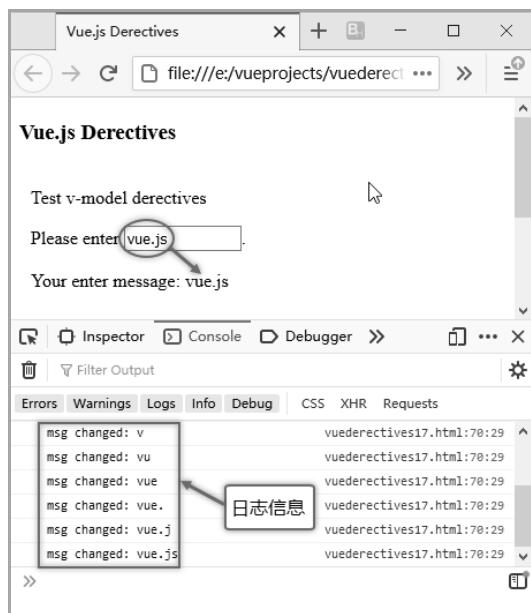


图 9.34 使用 v-model 指令实现数据双向绑定（二）

如图 9.34 中的箭头和标识所示，当我们在文本输入框中输入字符串“vue.js”后，页面中同步更新了我们输入的内容，说明 v-model 指令实现了针对数据对象（msg）的双向绑定。同时，在浏览器控制台中输出用户输入过程的日志记录。

那么，v-model 指令实现的双向绑定功能的原理是什么呢？这个 v-model 指令的原理非常简单，下面我们就从数据双向绑定的过程来分析，进而阐述一下 v-model 指令的设计原理。

所谓 v-model 指令数据双向绑定，就是负责在视图中监听用户输入事件，从而更新 Vue 实例中的数据对象。另外，v-model 指令会忽略所有表单<form>元素的 value、checked 和 selected 特性的初始值，其将 Vue 实例中的数据对象作为数据来源，然后当输入事件发生时去实时更新 Vue 实例中的数据对象。

我们以文本输入框<input>元素为例，具体描述一下针对 value 属性的数据双向绑定过程。

- (1) 需要在 Vue 实例的初始化过程中定义一个数据对象（例如：msg）。
- (2) 将文本输入框<input>元素中的 value 属性与 Vue 实例的数据对象（例如：msg）进行绑定。
- (3) 在文本输入框<input>元素中监听用户输入事件，将用户的输入与 Vue 实例的数据对象（例如：msg）进行同步。

接下来，我们将上面的【代码 9-21】按照上述过程修改一下，以印证一下 v-model 指令的实现原理。

**【代码 9-22】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-derectives-model-msg">
02   <p>Test v-model derectives</p>
03   <p>Please enter
04   <input
05     type="text"
06     @input="msg=$event.target.value"
07     :value="msg"
08     placeholder="pls enter..." />.
09 </p>
10 <p>Please enter
11 <input
12   type="text"
13   @input="input_msg($event)"
14   :value="msg"
15   placeholder="pls enter..." />.
16 </p>
17 <p>Your enter message: {{msg}}</p>
18 </div>
19 <script>
20   // Vue Entry
21   var vm = new Vue({
22     el: '#id-div-derectives-model-msg',
23     data: {
24       msg: ""
25     },
26     methods: {
27       input msg: function(ev) {
28         this.msg = ev.target.value;
29       }
30     }
31   })
32 </script>

```

### 【代码说明】

- 第 01~18 行代码中，在页面中通过<div>元素定义了一个层，具体说明如下：

- 在第 04~08 行代码中，通过<input type="text">元素定义了第一个文本输入框，用于测试双向绑定的功能。其中，第 06 行代码通过 v-on 指令缩写绑定用户输入事件（input），将用户输入的数据（\$event.target.value）与数据对象（msg）进行同步。第 07 行代码通过 v-bind 指令缩写接收文本输入框的 value 属性参数，并与数据对象（msg）进行绑定。
- 在第 11~15 行代码中，通过<input type="text">元素定义了第二个文本输入框，用于测试双向绑定的功能。该文本输入框与第 04~08 行代码定义的文本输入框的区别是：第 13 行代码中的定义用户输入事件（input）是通过事件方法（input\_msg）的方式实现的。
- 第 17 行代码中，使用文本插值方式引用一个对象（msg），用于同步显示用户在第 04~08 行代码定义的文本输入框或第 11~15 行代码定义的文本输入框中所输入的内容。
- 第 21~31 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 23~25 行代码中，通过 data 属性进行绑定数据操作。其中，第 24 行代码定义了一个数据对象（msg），分别对应第 06~07 行和第 13~14 行代码引用的数据对象（msg）。
  - 第 27~29 行代码是第 13 行代码中 input 输入事件处理方法（input\_msg）的具体实现，在第 28 行代码中将用户输入的内容赋值给数据对象（msg）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面初始效果如图 9.35 所示。

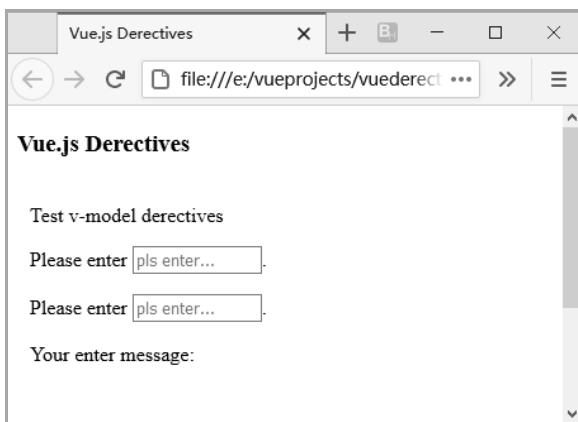


图 9.35 v-model 指令数据双向绑定原理（一）

如图 9.35 中所示，我们尝试在第一个文本输入框中输入一些内容（例如：vue），页面效果如图 9.36 所示。

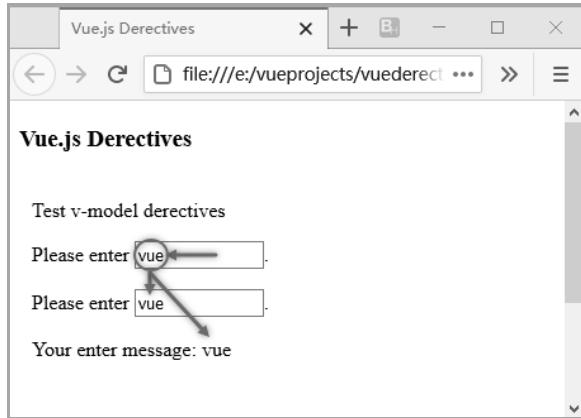


图 9.36 v-model 指令数据双向绑定原理（二）

如图 9.36 中的箭头和标识所示，当我们在第一个文本输入框中输入字符串“vue”后，页面中的第二个文本输入框以及文本信息中均同步更新了我们输入的内容，说明第 06 行代码实现了针对数据对象（msg）的双向绑定功能。

我们尝试继续在第二个文本输入框中输入一些内容（例如：.js），页面效果如图 9.37 所示。

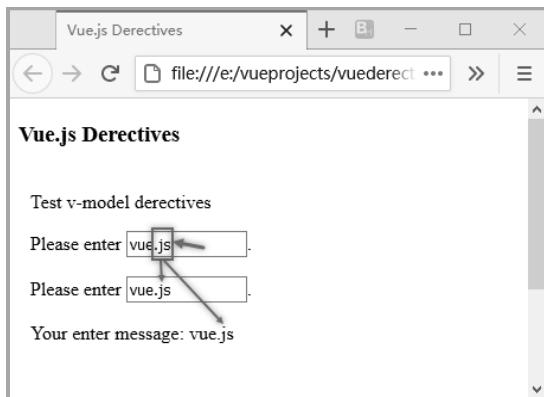


图 9.37 v-model 指令数据双向绑定原理（三）

如图 9.37 中的箭头和标识所示，当我们在第二个文本输入框中输入字符串“.js”后，页面中的第一个文本输入框以及文本信息中均同步更新了我们输入的内容，说明第 27~29 行代码定义的 input 输入事件处理方法（input\_msg）同样实现了针对数据对象（msg）的双向绑定功能。

从【代码 9-22】的运行结果来看，完全替代了【代码 9-21】中使用 v-model 指令所实现的数据双向绑定功能，进而证实了上述关于 v-model 指令的设计原理过程。

## 9.7.2 .lazy 修饰符

在 Vue.js 框架所设计的 v-model 指令固然功能十分强大，但在某些场景下也会有恼人的情况。比如，通过 v-model 指令实现的数据双向绑定功能，会在文本输入框内容改变的同时，同

步更新 Vue 数据对象的内容。但有时候我们希望在用户输入完毕、光标离开文本输入框时，才去完成数据更新（类似传统的 change 事件），此时就需要用到.lazy 修饰符了。

Vue.js 框架所设计的.lazy 修饰符，支持在用户光标离开文本输入框中后才去更新数据，相当于用 change 事件取代 input 输入事件。下面这个代码实例通过在 v-model 指令上使用.lazy 修饰符，实现一个非常简单的数据更新功能。

【代码 9-23】（详见源代码 vuederectives 目录中的 vuederectives.html 文件）

```

01 <div id="id-div-derectives-model-lazy-msg">
02   <p>Test v-model derectives</p>
03   <p>Please enter
04   <input
05     type="text"
06     v-model.lazy="msg"
07     @input="input_event"
08     @change="change_event"
09     placeholder="pls enter..." />.
10  </p>
11  <p>Your enter message: {{msg}}</p>
12 </div>
13 <script>
14   // Vue Entry
15   var vm = new Vue({
16     el: '#id-div-derectives-model-lazy-msg',
17     data: {
18       msg: ""
19     },
20     methods: {
21       input_event: function() {
22         console.log("input event: " + this.msg);
23       },
24       change_event: function() {
25         console.log("change event: " + this.msg);
26       }
27     }
28   })
29 </script>

```

### 【代码说明】

- 第 04~09 行代码中，通过<input type="text">元素定义了一个文本输入框，用于测试“.lazy”修饰符的功能。具体说明如下：
  - 第 06 行代码通过 v-model 指令双向绑定数据对象（msg），注意该 v-model 指令添加了.lazy 修饰符。
  - 第 07 行代码通过 v-on 指令缩写绑定 input 输入事件方法（input\_msg）。
  - 第 08 行代码通过 v-on 指令缩写绑定文本框 change 事件方法（change\_msg）。
  - 第 07~08 行代码定义的这两个事件，可以清楚地对比出.lazy 修饰符对于 input

输入事件和文本框 change 事件的不同作用。

- 第 11 行代码中，使用文本插值方式引用了一个对象（msg），用于同步显示用户在第 04~09 行代码定义的文本输入框中所输入的内容。
- 第 15~28 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 17~19 行代码中，通过 data 属性进行绑定数据操作。其中，第 18 行代码定义了一个数据对象（msg），分别对应第 06 行和第 11 行代码引用的数据对象（msg）。
  - 第 21~23 行代码是第 07 行代码中 input 输入事件处理方法（input\_msg）的具体实现，第 22 行代码向浏览器控制台中输出一行日志信息。
  - 第 24~26 行代码是第 08 行代码中文本框 change 事件处理方法（change\_msg）的具体实现，第 25 行代码向浏览器控制台中输出一行日志信息。

接下来，通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面初始效果如图 9.38 所示。

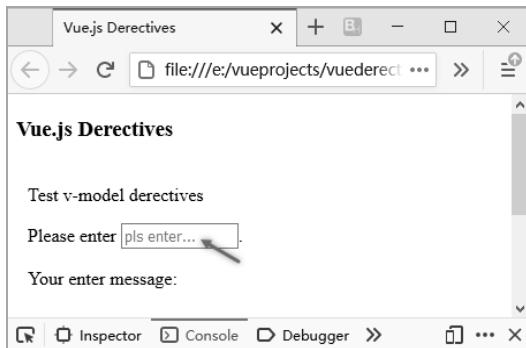


图 9.38 在 v-model 指令上使用.lazy 修饰符（一）

如图 9.38 中的箭头所示，我们尝试在文本输入框中输入一些内容（例如：vue.js），页面效果如图 9.39 所示。

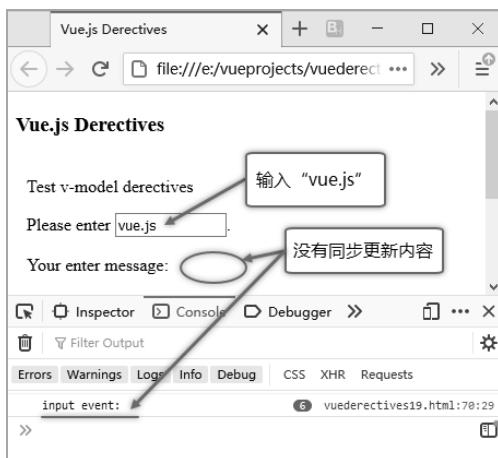


图 9.39 在 v-model 指令上使用.lazy 修饰符（二）

如图 9.39 中的箭头和标识所示，当我们在文本输入框中输入字符串“vue.js”后（光标焦点没有离开文本输入框），页面中没有同步更新所输入的内容。这说明在 v-model 指令上使用.lazy 修饰符后，数据对象（msg）的双向绑定功能失效了。同时，在浏览器控制台中输出的日志记录也印证了上述结果。

接下来，我们将光标输入焦点离开该文本输入框，观察一下页面内容的更新，页面效果如图 9.40 所示。

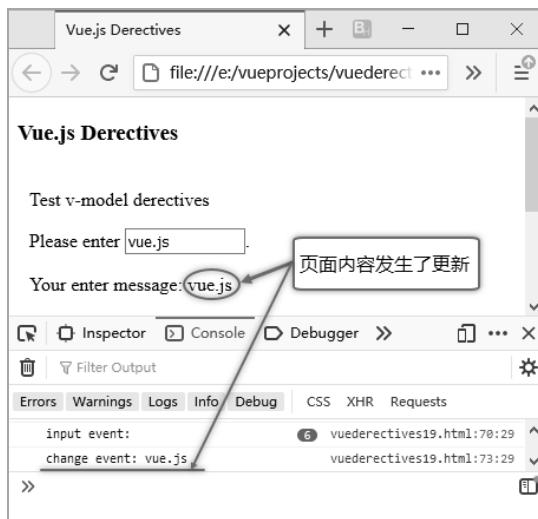


图 9.40 在 v-model 指令上使用.lazy 修饰符（三）

如图 9.40 中的箭头和标识所示，当我们光标输入焦点离开该文本输入框后，文本输入框中的 change 事件被激活了。页面中的内容更新了，浏览器控制台中输出的日志记录也印证了上述结果。

通过图 9.39 和图 9.40 的对比可以看到，在 v-model 指令上使用.lazy 修饰符会屏蔽掉 input 输入事件，并将该事件替换为 change 事件。

### 9.7.3 .number 修饰符

Vue.js 框架还设计了一个.number 修饰符。在 v-model 指令上使用.number 修饰符后，如果在文本框中先输入数字就会限制只能输入数字，如果先输入字符串就相当于没有加.number 修饰符（恢复为普通文本框）。

下面看一个在 v-model 指令上使用.number 修饰符的代码实例，了解一下.number 修饰符的使用方法。

**【代码 9-24】**（详见源代码 vuederectives 目录中的 vuederectives.html 文件）

```

01 <div id="id-div-derectives-model-number">
02   <p>Test v-model derectives</p>
03   <p>Please enter number</p>
04   <input
05     type="text"

```

```

06      v-model.number="num"
07      @input="input_num"
08      placeholder="pls enter..." />.
09  </p>
10  <p>Your enter number: {{num}}</p>
11 </div>
12 <script>
13 // Vue Entry
14 var vm = new Vue({
15   el: '#id-div-derectives-model-number',
16   data: {
17     num: null
18   },
19   methods: {
20     input_num: function() {
21       console.log("input event: " + this.num);
22     }
23   }
24 })
25 </script>

```

### 【代码说明】

- 第 04~08 行代码中，通过<input type="text">元素定义了一个文本输入框，用于测试.number 修饰符的功能。具体说明如下：
  - 第 06 行代码通过 v-model 指令双向绑定数据对象（num），注意该 v-model 指令添加了 .number 修饰符。
  - 第 07 行代码通过 v-on 指令缩写绑定 input 输入事件方法（input\_num）。
- 第 10 行代码中，使用文本插值方式引用一个对象（num），用于同步显示用户在第 04~08 行代码定义的文本输入框中所输入的内容。
- 第 14~24 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 16~18 行代码中，通过 data 属性进行绑定数据操作。其中，第 17 行代码定义了一个数据对象（num），分别对应第 06 行和第 10 行代码引用的数据对象（num）。
  - 第 20~22 行代码是第 07 行代码中 input 输入事件处理方法（input\_num）的具体实现，第 21 行代码向浏览器控制台中输出一行日志信息。

接下来，通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面初始效果如图 9.41 所示。

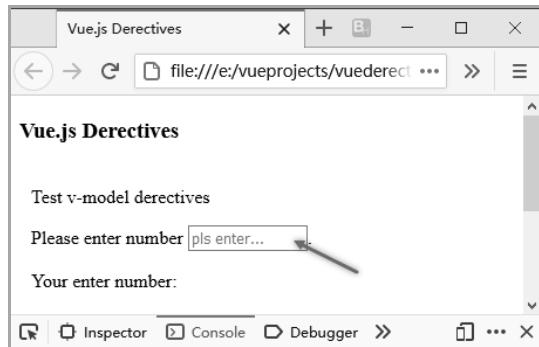


图 9.41 在 v-model 指令上使用.number 修饰符（一）

如图 9.41 中的箭头所示，我们先尝试在文本输入框中输入数字（例如：123），页面效果如图 9.42 所示。

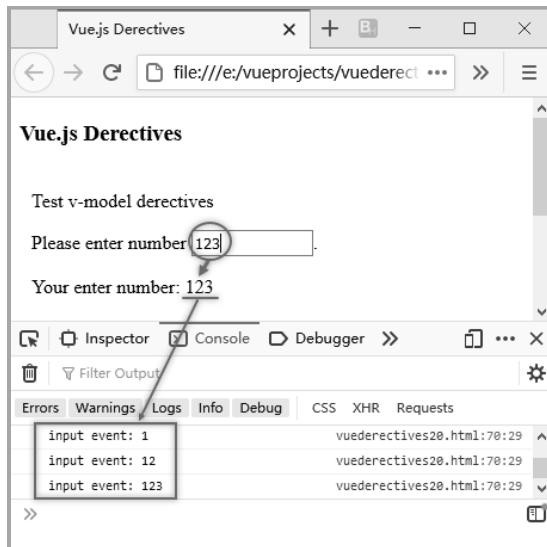


图 9.42 在 v-model 指令上使用.number 修饰符（二）

如图 9.42 中的箭头和标识所示，当我们在文本输入框中输入数字“123”后，页面中同步更新所输入的数字。同时，在浏览器控制台中输出的日志记录也印证了上述结果。

我们尝试继续输入字符串（例如：abc），观察一下页面内容有无变化更新，页面效果如图 9.43 所示。

如图 9.43 中的箭头和标识所示，当我们输入字符串“abc”后，页面中的内容没有发生更新，浏览器控制台中输出的日志记录也印证了上述结果。然后，我们将光标焦点离开文本输入框，观察一下页面内容有无变化更新，页面效果如图 9.44 所示。

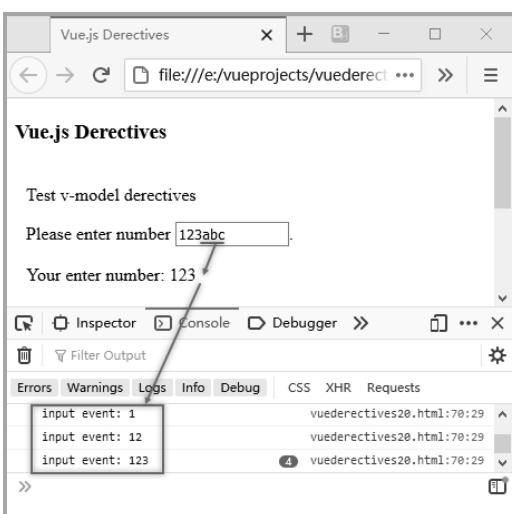


图 9.43 在 v-model 指令上使用.number 修饰符（三）

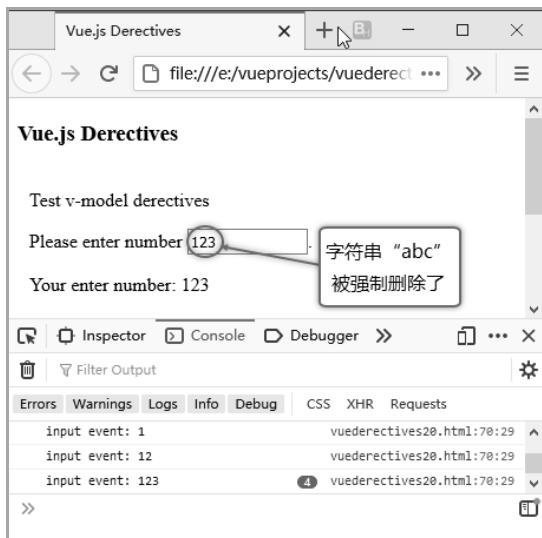


图 9.44 在 v-model 指令上使用.number 修饰符（四）

如图 9.44 中的箭头和标识所示，当我们将光标焦点离开文本输入框后，刚刚输入的字符串“abc”被强制删除了。这说明对于使用了.number 修饰符的文本输入框，如果先输入数字，就只能继续输入数字，输入的字符串是不被接受的。

如果用户先输入字符串，数字还会被文本输入框接收吗？我们继续测试一下，页面效果如图 9.45 所示。

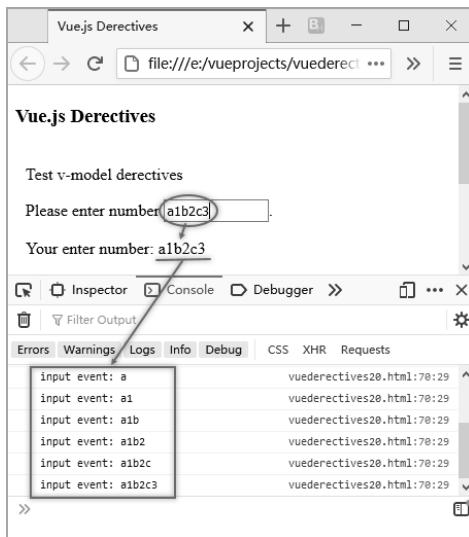


图 9.45 在 v-model 指令上使用.number 修饰符（五）

如图 9.45 中的箭头和标识所示，当我们在文本输入框先输入字符串后，再输入数字也不会受到影响。这说明对于使用了.number 修饰符的文本输入框，如果先输入字符串，那该文本输入框就转成普通的文本输入框了。

### 9.7.4 .trim 修饰符

Vue.js 框架还设计了一个.trim 修饰符，顾名思义在 v-model 指令上使用.trim 修饰符后，会过滤并删除掉文本输入框中内容首尾的空格。

下面看一个在 v-model 指令上使用.trim 修饰符的代码实例，了解一下.trim 修饰符的使用方法。

**【代码 9-25】**（详见源代码 vuederectives 目录中的 vuederectives.html 文件）

```

01 <div id="id-div-derectives-model-trim">
02   <p>Test v-model derectives</p>
03   <p>Please enter
04   <input
05     type="text"
06     v-model.trim="trim"
07     @input="input_trim"
08     placeholder="pls enter..." />.
09   </p>
10   <p>Your enter: {{trim}}</p>
11 </div>
12 <script>
13   // Vue Entry
14   var vm = new Vue({
15     el: '#id-div-derectives-model-trim',
16     data: {
17       trim: null
18     },
19     methods: {
20       input_trim: function() {
21         console.log("input event: " + this.trim);
22       }
23     }
24   })
25 </script>
```

**【代码说明】**

- 第 04~08 行代码中，通过<input type="text">元素定义了一个文本输入框，用于测试.trim 修饰符的功能。具体说明如下：
  - 第 06 行代码通过 v-model 指令双向绑定一个对象（trim），注意该 v-model 指令添加了.trim 修饰符。
  - 第 07 行代码通过 v-on 指令缩写绑定 input 输入事件方法（input\_trim）。
- 第 10 行代码中，使用文本插值方式引用一个对象（trim），用于同步显示用户在第 04~08 行代码定义的文本输入框中所输入的内容。
- 第 14~24 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：

- 第 16~18 行代码中，通过 data 属性进行绑定数据操作。其中，第 17 行代码定义了一个对象（trim），分别对应第 06 行和第 10 行代码引用的数据对象（trim）。
- 第 20~22 行代码是第 07 行代码中 input 输入事件处理方法（input\_trim）的具体实现，第 21 行代码向浏览器控制台中输出一行日志信息。

接下来，通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.46 所示。

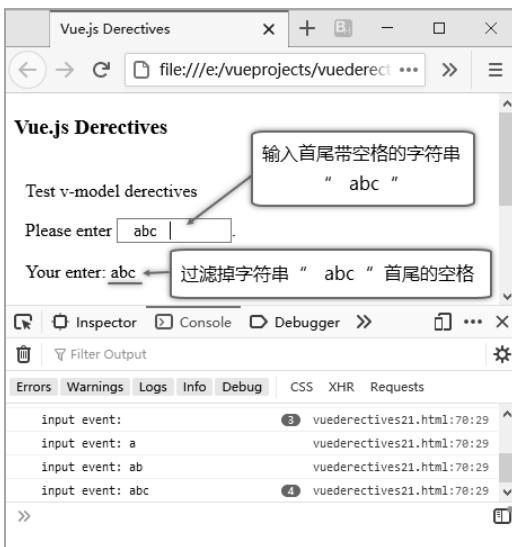


图 9.46 在 v-model 指令上使用.trim 修饰符（一）

如图 9.46 中的箭头所示，我们尝试在文本输入框中输入首尾带空格的字符串“ abc ”，页面中更新的内容强制将首尾的空格过滤并删除了。这说明对于使用了.trim 修饰符的文本输入框，会将输入内容中的首尾空格强制过滤并删除。

## 9.8 Vue.js 计算属性

本节介绍 Vue.js 框架中计算属性方面的内容。设计人员通过计算属性可以避免在模板内使用太过复杂的逻辑表达式，从而减轻了模板维护的工作量。在 Vue.js 框架中，计算属性通过 computed 关键字来声明。

为了详细说明计算属性的使用方法，我们通过一个基本的字符串反序计算的应用来逐步讲解。提到字符串反序计算，大多数读者会想到 String 类的 reverse() 方法，直接调用该方法就可以了。于是，就可能会写出下面的代码实例。

**【代码 9-26】**（详见源代码 vuecomputed 目录中的 vuecomputed.html 文件）

```
01 <div id="id-div-computed-msg">
```

```

02      <p>This is a vue computed test.</p>
03      <p>Original message: {{msg}}.</p>
04      <p>Reverse message:
05          {{msg.split('').reverse().join('')}}.
06      </p>
07  </div>
08  <script>
09      var vm = new Vue({
10          el: '#id-div-computed-msg',
11          data: {
12              msg: "Hello Vue.js!"
13          }
14      })
15  </script>

```

### 【代码说明】

- 在第 03 行代码中，通过文本插值模板引入一个对象（msg）。
- 在第 05 行代码中，再次通过文本插值模板引入一个表达式，该表达式通过 String 类的 reverse()方法将字符串对象（msg）进行反序计算。
- 第 09~14 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。具体说明如下：
  - 第 11~13 行代码中，通过 data 属性进行绑定数据操作。其中，第 12 行代码定义了一个对象（msg），并进行初始化。该对象对应第 03 行和第 05 行代码引用的对象（msg）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuederectives.html 页面，页面效果如图 9.47 所示。

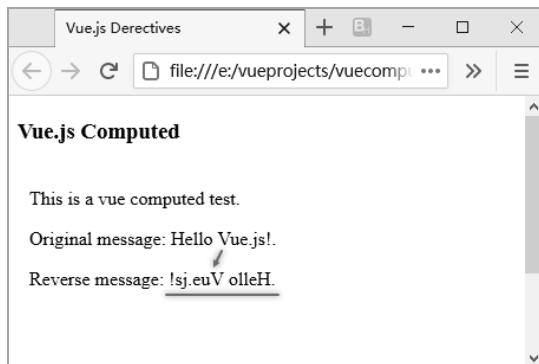


图 9.47 使用 Vue.js 的计算属性（一）

如图 9.47 中的箭头所示，第 05 行代码中的计算表达式将字符串对象（msg）成功进行了反序计算。

但是，在 Vue.js 框架应用中文本插值模板中，使用类似第 05 行代码中的计算表达式都会带来一些困扰，过于复杂的计算逻辑不利于设计人员维护代码，也会给其他设计人员阅读代码

带来困难。因此，在 Vue.js 框架应用的文本插值模板中，不建议使用复杂的表达式，应该使用简洁直观的对象名称。于是，计算属性这个概念就应运而生了。下面我们看看如何通过使用计算属性改写【代码 9-26】。

**【代码 9-27】**（详见源代码 vuedirectives 目录中的 vuedirectives.html 文件）

```

01 <div id="id-div-computed-msg">
02   <p>This is a vue computed test.</p>
03   <p>Original message: {{msg}}.</p>
04   <p>Computed reverse message: {{reverseMsg}}.</p>
05 </div>
06 <script>
07   var vm = new Vue({
08     el: '#id-div-computed-msg',
09     data: {
10       msg: "Hello Vue.js!"
11     },
12     computed: {
13       reverseMsg: function() {
14         return this.msg.split('').reverse().join('');
15       }
16     }
17   })
18 </script>
```

### 【代码说明】

- 在第 03 行代码中，通过文本插值模板引入一个对象（msg）。
- 在第 04 行代码中，再次通过文本插值模板引入一个计算属性（reverseMsg）。
- 第 07~17 行的脚本代码中，通过“new Vue()”构造函数实例化 Vue 对象（vm）。  
具体说明如下：
  - 第 09~11 行代码中，通过 data 属性进行绑定数据操作。其中，第 12 行代码定义一个字符串对象（msg），并进行初始化。该对象对应第 03 行代码引用的对象（msg）。
  - 第 12~16 行代码中，通过 computed 属性进行绑定计算属性的操作。其中，第 13 ~ 15 行代码定义了一个计算属性（reverseMsg），该属性对应第 04 行代码引用的计算属性（reverseMsg）。在第 14 行代码中，通过 String 类的 reverse() 方法将字符串对象（msg）进行反序计算，并将计算结果返回给计算属性（reverseMsg）。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuedirectives.html 页面，页面初始效果如图 9.48 所示。

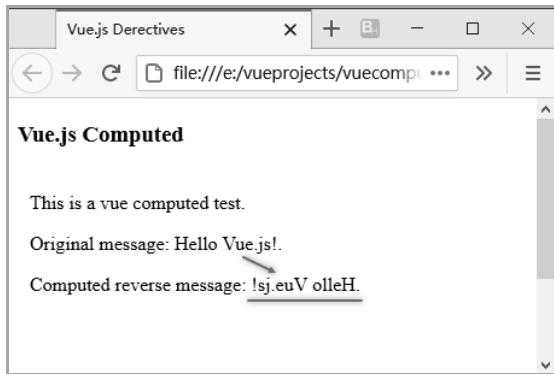


图 9.48 使用 Vue.js 的计算属性（二）

如图 9.48 中的箭头所示，第 04 行代码中的计算属性（reverseMsg）显示了正确的反序计算结果。这说明 Vue.js 框架所设计的计算属性，可以完美替代在文本插值模板中使用的复杂表达式。

# 第 10 章

## Vue.js 样式绑定

本章主要介绍 Vue.js 框架中样式（class 和 style）绑定的内容。前端代码设计离不开使用样式表，Vue.js 针对样式也同样进行了增强设计。

通过本章的学习可以：

- 掌握使用 Vue.js 绑定 HTML Class 的方法。
- 掌握使用 Vue.js 绑定 HTML 内联样式（style）的方法。
- 了解 Vue.js 样式绑定的数组语法。

### 10.1 Vue.js 绑定 HTML Class

本节介绍 Vue.js 绑定 HTML Class 方面的内容。Vue.js 绑定 Class 仍旧通过 v-bind 指令完成，一般写成标准语法形式为“v-bind:class”，也可以使用 Vue 指令的缩写形式“:class”。

#### 10.1.1 绑定静态 Class

对于 Vue.js 框架而言，绑定静态 HTML Class 是最基本的一种方式，等同于直接固定元素的样式类。下面的代码实例中，在页面中定义了一组（3 个）段落（`<p>`）元素，然后通过静态绑定方式为每个`<p>`元素定义样式类，代码如下：

【代码 10-1】（详见源代码 `vuestyle` 目录中的 `vuestyle1.html` 文件）

```
01 <style>
02   p.font-small {
03     font-size: 18px;
04   }
05   p.font-medium {
06     font-size: 24px;
07   }
08   p.font-big {
```

```

09         font-size: 32px;
10     }
11 </style>
12 <div id="id-div-class-p">
13     <p>This is a vue class test.</p>
14     <p :class="pBig">{{msg}}</p>
15     <p :class="pMedium">{{msg}}</p>
16     <p :class="pSmall">{{msg}}</p>
17 </div>
18 <script>
19     // Vue Entry
20     var vm = new Vue({
21         el: '#id-div-class-p',
22         data: {
23             msg: "Hello Vue.js!",
24             pSmall: "font-small",
25             pMedium: "font-medium",
26             pBig: "font-big"
27         },
28     })
29 </script>

```

### 【代码说明】

- 第 01~11 行代码中，通过<style>标签定义了一组（3个）样式类，分别用于修饰不同字体大小的段落。
- 第 12~17 行代码中，通过<div>元素定义一个层，层内定义一组（3个）段落<p>元素，每个元素均通过文本插值模板引用一个对象（msg）。同时，这3个段落<p>元素内部通过 v-bind 指令缩写方式绑定样式参数 class，每个 class 的值分别为第 01~11 行代码中所定义的3个样式类。
- 第 22~27 行代码中，通过 data 属性绑定数据操作。具体说明如下：
  - 第 23 行代码中定义了一个属性（msg），并进行初始化。该属性对应第 14~16 行代码中引用的对象（msg）。
  - 第 24~26 行代码中分别定义 3 个属性（pSmall、pMedium 和 pBig），分别初始化为第 01~11 行代码中所定义的 3 个样式类。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuestyle.html 页面，效果如图 10.1 所示。

如图 10.1 中的箭头和标识所示，第 01~11 行代码中定义的 3 个样式类，成功渲染到第 14~16 行代码中定义的段落<p>元素上去了。

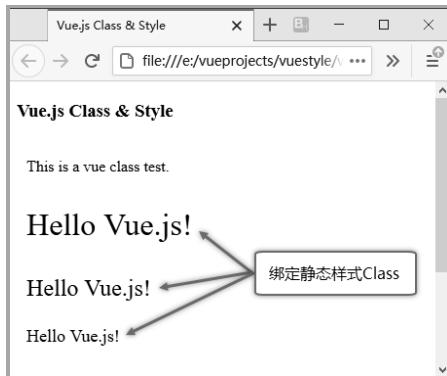


图 10.1 Vue.js 绑定静态 Class

### 10.1.2 绑定动态 Class

对于 Vue.js 框架而言，真正强大之处是绑定动态 HTML Class 的方式，动态绑定方式可以根据用户编程来动态切换样式类。下面的代码实例是在【代码 10-1】的基础上略作修改，通过动态绑定方式为每个段落<p>元素定义样式类。

**【代码 10-2】**（详见源代码 vuestyle 目录中的 vuestyle2.html 文件）

```

01  <style>
02      p.font-small {
03          font-size: 18px;
04      }
05      p.font-medium {
06          font-size: 24px;
07      }
08      p.font-big {
09          font-size: 32px;
10     }
11  </style>
12  <div id="id-div-class-p">
13      <p>This is a vue class test.</p>
14      <p :class="{ 'font-big':pBigActive }">{{msg}}</p>
15      <p :class="{ 'font-medium':pMediumActive }">{{msg}}</p>
16      <p :class="{ 'font-small':pSmallActive }">{{msg}}</p>
17  </div>
18  <script>
19      // Vue Entry
20      var vm = new Vue({
21          el: '#id-div-class-p',
22          data: {
23              msg: "Hello Vue.js!",
24              pSmallActive: true,
25              pMediumActive: false,
26              pBigActive: true
27          }
28      })

```

```
29 </script>
```

### 【代码说明】

- 第 01~11 行代码中，通过<style>标签定义了一组（3个）样式类，分别用于修饰不同字体大小的段落。
- 第 12~17 行代码中，通过<div>元素定义了一个层，层内定义一组（3个）段落<p>元素，每个元素均通过文本插值模板引用了一个对象（msg）。同时，这3个段落<p>元素内部通过 v-bind 指令缩写方式绑定了样式参数 class，每个 class 的值分别指定为一个对象，该对象的名称分别是第 01~11 行代码中所定义的 3 个样式类名称，而每个对象的值分别是一个布尔型的属性值（pSmallActive、pMediumActive 和 pBigActive）。
- 第 22~27 行代码中，通过 data 属性进行绑定数据操作。具体说明如下：
  - 第 23 行代码中定义了一个属性（msg），并进行初始化。该属性对应第 14~16 行代码中引用的对象（msg）。
  - 第 24~26 行代码中分别定义了 3 个布尔型属性（pSmallActive、pMediumActive 和 pBigActive），并进行初始化。这 3 个布尔型属性分别对应第 14~16 行代码中所引用的 3 个布尔型属性值，如此定义就实现了绑定动态 Class 的操作，因为可以通过编程动态修改这 3 个布尔型属性的取值。

接下来，通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuestyle.html 页面，效果如图 10.2 所示。

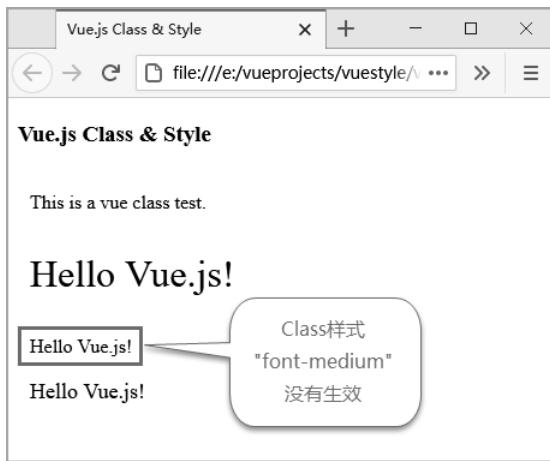


图 10.2 Vue.js 绑定动态 Class

如图 10.2 中的箭头和标识所示，第 15 行代码中引用的样式类 font-medium 没有生效，该段落<p>元素的字体大小还是默认的。这就说明第 25 行代码将布尔型属性（pMediumActive）值初始化为 false 后，在第 15 行代码中通过判断布尔型属性（pMediumActive）的逻辑值，将引用的样式类 font-medium 屏蔽了。

这个 Vue.js 框架绑定动态 Class 的功能很实用，比如，在很多的网页界面中都会设置有切

换页面风格样式的按钮，这个效果通过绑定动态 Class 的操作就很容易实现。下面就是一个通过绑定动态 Class 操作实现切换字体大小的应用实例。

**【代码 10-3】**（详见源代码 vuestyle 目录中的 vuestyle3.html 文件）

```

01 <style>
02   p {
03     font-size: 16px;
04   }
05   p.font-big {
06     font-size: 32px;
07   }
08 </style>
09 <div id="id-div-class-p">
10   <p>This is a vue class test.</p>
11   <p :class="{ 'font-big':pBigActive }">{{msg}}</p>
12   <button @click="event_click_font">Toggle Class Font</button>
13 </div>
14 <script>
15   // Vue Entry
16   var vm = new Vue({
17     el: '#id-div-class-p',
18     data: {
19       msg: "Hello Vue.js!",
20       pBigActive: false
21     },
22     methods: {
23       event_click_font: function() {
24         if (this.pBigActive) {
25           this.pBigActive = false;
26         } else {
27           this.pBigActive = true;
28         }
29         console.log(this.pBigActive);
30       }
31     }
32   })
33 </script>

```

**【代码说明】**

- 第 01~08 行代码中，通过<style>标签定义了一组（共 2 个）关于段落<p>元素的样式类，分别用于修饰默认字体大小和加大字体大小的段落。
- 第 09~13 行代码中，通过<div>元素定义一个层。具体说明如下：
  - 第 11 行代码中定义了一个段落<p>元素，通过文本插值模板引用一个对象（msg）。同时，这个段落<p>元素内部通过 v-bind 指令缩写方式绑定样式参数 class，其值定义为一个对象。该对象的名称是第 05~07 行代码中所定义的样式类名称，其值是一个布尔型的属性值（pBigActive）。

- 第 12 行代码中定义了一个按钮<button>元素，通过 v-on 指令缩写绑定单击 click 事件方法（event\_click\_font）。
- 第 18~21 行代码中，通过 data 属性进行绑定数据操作。具体说明如下：
  - 第 19 行代码中定义了一个属性（msg），并进行初始化。该属性对应第 11 行代码中引用的对象（msg）。
  - 第 20 行代码中分别定义了一个布尔型属性（pBigActive），并进行初始化（false）。这个布尔型属性对应第 11 行代码中所引用的布尔型属性值，该定义实现绑定动态 Class 的操作，用户可以通过单击按钮动态修改这个布尔型属性的取值。
- 第 23~30 行代码中，是第 12 行代码定义的按钮<button>元素中单击 click 事件方法（event\_click\_font）的具体实现。其中，第 24~28 行代码通过条件语句判断布尔型属性（pBigActive）的逻辑值，完成动态修改该属性值的操作，进而实现切换页面中段落<p>元素字体大小的效果。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuestyle.html 页面，具体初始效果如图 10.3 所示。

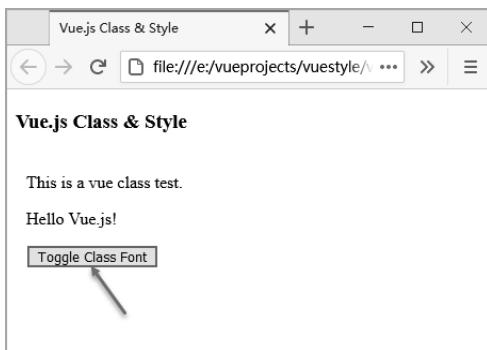


图 10.3 Vue.js 通过绑定动态 Class 实现字体大小切换（一）

如图 10.3 中的箭头所示，我们尝试单击页面中的“Toggle Class Font”按钮，页面效果如图 10.4 所示。

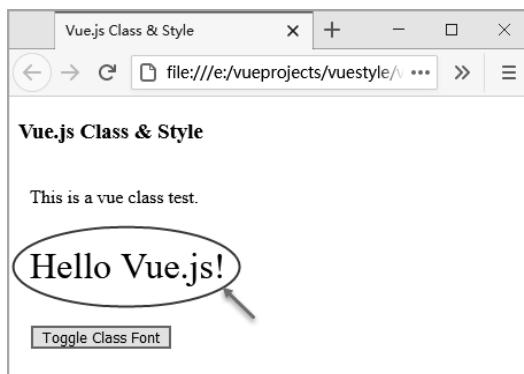


图 10.4 Vue.js 通过绑定动态 Class 实现字体大小切换（二）

如图 10.4 中的箭头和标识所示，在单击页面中的“Toggle Class Font”按钮后，段落中的

字体成功切换为大字体了。

### 10.1.3 绑定多个 Class

Vue.js 框架还可以实现同时绑定多个 Class 的操作，该功能在实际开发中经常会使用到。下面的代码实例是在【代码 10-1】和【代码 10-2】的基础上略作修改而成的，通过动态绑定多个 Class 方式来定义段落

元素的样式类。

【代码 10-4】（详见源代码 vuestyle 目录中的 vuestyle4.html 文件）

```
01 <style>
02     p.font-small {
03         font-size: 18px;
04     }
05     p.font-medium {
06         font-size: 24px;
07     }
08     p.font-big {
09         font-size: 32px;
10     }
11     p.font-normal {
12         font-style: normal;
13     }
14     p.font-italic {
15         font-style: italic;
16     }
17     p.font-weight {
18         font-weight: bold;
19     }
20 </style>
21 <div id="id-div-class-p">
22     <p>This is a vue class test.</p>
23     <p :class="{'font-big':pBigActive, 'font-normal':pNormalActive}">
24         {{msg}}
25     </p>
26     <p :class="{'font-medium':pMediumActive, 'font-italic':pItalicActive}">
27         {{msg}}
28     </p>
29     <p class="font-weight" :class="{'font-small':pSmallActive}">
30         {{msg}}
31     </p>
32 </div>
33 <script>
34     // Vue Entry
35     var vm = new Vue({
36         el: '#id-div-class-p',
37         data: {
38             msg: "Hello Vue.js!",
39             pSmallActive: true,
40             pMediumActive: false,
```

```

41         pBigActive: true,
42         pNormalActive: false,
43         pItalicActive: true
44     }
45 }
46 </script>

```

### 【代码说明】

- 第 01~20 行代码中，通过<style>标签定义一组（6个）样式类，分别用于修饰不同字体大小和不同风格样式的段落。
- 第 21~32 行代码中，通过<div>元素定义了一个层，层内定义一组（3个）段落<p>元素。具体说明如下：
  - 在第 23~25 行代码中定义的第 1 个段落<p>元素中，通过 v-bind 指令缩写方式绑定了样式参数 class，而该 class 的值指定两个对象，分别是第 08~10 行代码中所定义的样式类 font-big 和第 11~13 行代码中所定义的样式类 font-normal，且对象的值分别是两个布尔型的属性值（pBigActive 和 pNormalActive）。
  - 同样，在第 26~28 行代码中定义的第 2 个段落<p>元素中，通过 v-bind 指令缩写方式绑定样式参数 class，而该 class 的值也指定两个对象，分别是第 05~07 行代码中所定义的样式类 font-medium 和第 14~16 行代码中所定义的样式类 font-italic，且对象的值分别是两个布尔型的属性值（pMediumActive 和 pItalicActive）。
  - 而在第 29~31 行代码中定义的第 3 个段落<p>元素中，先是通过传统的 class 属性方式引用样式类 font-weight，然后又通过 v-bind 指令缩写方式绑定样式类 font-small，这其实是在 Vue.js 框架下使用混合语法引用样式类的方式。
- 第 37~44 行代码中，通过 data 属性进行绑定数据操作。具体说明如下：
  - 第 38 行代码中定义了一个属性（msg），并进行初始化。该属性对应第 24 行、第 27 行和第 30 行代码中引用的对象（msg）。
  - 第 39~43 行代码中定义了 6 个布尔型属性，分别对应第 23~31 行代码中段落<p>元素所引用的 6 个布尔型属性值，用于实现绑定动态 Class 的操作。

下面通过 VS Code 开发工具启动 FireFox 浏览器，运行测试 vuestyle.html 页面，效果如图 10.5 所示。

如图 10.5 中的箭头所示，在第 23~31 行代码中，段落<p>元素绑定多个 Class 的操作均得到预期的显示效果。

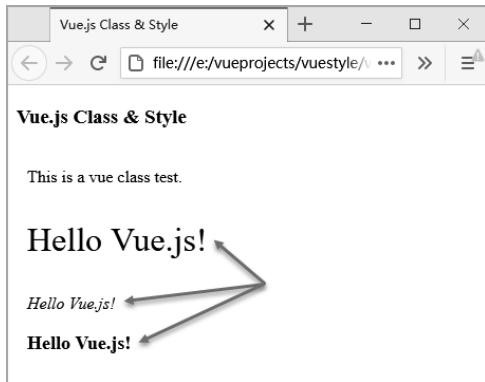


图 10.5 Vue.js 绑定多个 Class

## 10.2 通过数组语法绑定 Class

在 Vue.js 框架下绑定多个 Class，还支持一种数组语法的方式，可以进一步简化代码的编写。在下面的代码实例中，页面中定义了一个段落（`<p>`）元素，然后通过数组语法方式为该段落`<p>`元素引用多个样式类。

**【代码 10-5】**（详见源代码 `vuestyle` 目录中的 `vuestyle5.html` 文件）

```

01  <style>
02      p.font-medium {
03          font-size: 24px;
04      }
05      p.font-weight {
06          font-weight: bold;
07      }
08      p.font-italic {
09          font-style: italic;
10     }
11  </style>
12  <div id="id-div-class-p">
13      <p>This is a vue class test.</p>
14      <p :class="[pMediumClass,pWeightClass,pItalicClass]">{{msg}}</p>
15  </div>
16  <script>
17      // Vue Entry
18      var vm = new Vue({
19          el: '#id-div-class-p',
20          data: {
21              msg: "Hello Vue.js!",
22              pMediumClass: 'font-medium',
23              pWeightClass: 'font-weight',

```

```

24         pItalicClass: 'font-italic'
25     }
26   })
27 </script>

```

### 【代码说明】

- 第 01~11 行代码中，通过<style>标签定义了一组（3个）样式类，分别用于修饰不同字体大小、粗细和字体风格的段落。
- 第 14 行代码中定义了一个段落<p>元素，通过文本插值模板引用一个对象（msg）。同时，该段落<p>元素内部通过 v-bind 指令缩写方式绑定样式参数 class，其 class 的值就是通过数组语法的方式，分别引用第 22~24 行代码中所定义的 3 个属性（pMediumClass、pWeightClass 和 pItalicClass）。
- 第 20~25 行代码中，通过 data 属性进行绑定数据操作。具体说明如下：
  - 第 21 行代码中定义了一个属性（msg），并进行初始化。该属性对应第 14 行代码中引用的对象（msg）。
  - 第 22~24 行代码中分别定义了 3 个属性（pMediumClass、pWeightClass 和 pItalicClass），分别初始化为第 01~11 行代码中所定义的 3 个样式类（font-medium、font-weight 和 font-italic），对应于第 14 行代码中 class 所引用的 3 个属性（pMediumClass、pWeightClass 和 pItalicClass）。

接下来，运行测试 vuestyle.html 页面，效果如图 10.6 所示。

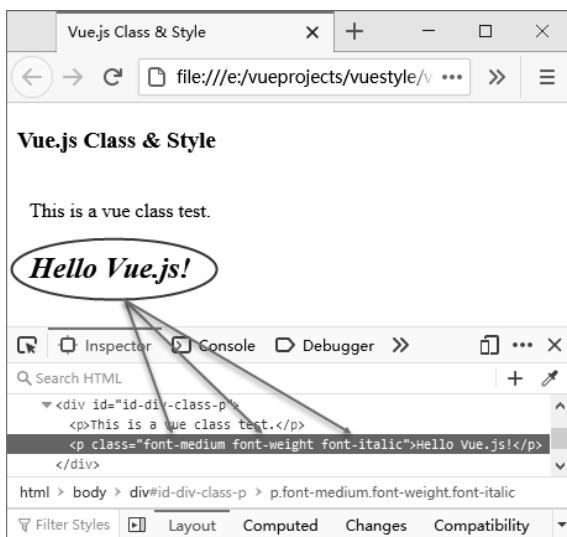


图 10.6 Vue.js 通过数组语法绑定 Class

如图 10.6 中的箭头和标识所示，在浏览器控制台中可以看到第 01~11 行代码中定义的 3 个样式类，成功渲染到第 14 行代码中定义的段落<p>元素中。

# 10.3 Vue.js 绑定 HTML Style

本节介绍 Vue.js 绑定 HTML Style（内联样式）方面的内容。Vue.js 绑定 Style 也是通过 v-bind 指令完成的，一般写成标准语法形式为“v-bind:style”，也可以使用 Vue 指令缩写形式“:style”。

## 10.3.1 绑定静态 Style

对于 Vue.js 框架而言，绑定静态 HTML Style 是最基本的一种方式，该方式看上去更加直观。虽然使用“v-bind:style”语法看上去与“v-bind:class”语法类似，但是“v-bind:class”语法基于 CSS 方式，而“v-bind:style”语法基于最基本的 JavaScript 对象方式。

在下面的代码实例中，页面中定义了一个段落（

）元素，然后通过静态绑定方式为该 

元素定义 Style。

【代码 10-6】（详见源代码 vuestyle 目录中的 vuestyle6.html 文件）

```

01 <div id="id-div-class-p">
02   <p>This is a vue class test.</p>
03   <p :style="{"
04     fontSize:ftSize,
05     fontWeight:ftWeight,
06     fontStyle:ftStyle}">
07     {{msg}}
08   </p>
09 </div>
10 <script>
11   // Vue Entry
12   var vm = new Vue({
13     el: '#id-div-class-p',
14     data: {
15       msg: "Hello Vue.js!",
16       ftSize: '32px',
17       ftWeight: 'bold',
18       ftStyle: 'italic'
19     }
20   })
21 </script>
```

### 【代码说明】

- 第 01~09 行代码中，通过<div>元素定义了一个层，层内定义一个段落<p>元素，通过文本插值模板引用一个对象（msg）。其中，在第 03~06 行代码中通过 v-bind 指

令缩写方式绑定内联样式参数 style，其参数值为一组样式对象（ftSize、ftWeight 和 ftStyle）。

- 第 14~19 行代码中，通过 data 属性进行绑定数据操作。具体说明如下：
  - 第 15 行代码中定义了一个属性（msg），并进行初始化。该属性对应第 07 行代码中引用的对象（msg）。
  - 第 16~18 行代码中分别定义了 3 个属性（ftSize、ftWeight 和 ftStyle），初始化为具体的字体大小、字体粗细和字体风格，其对应于第 04~06 行代码中段落<p>元素引用的样式对象（ftSize、ftWeight 和 ftStyle）。

接下来，运行测试 vuestyle.html 页面，效果如图 10.7 所示。

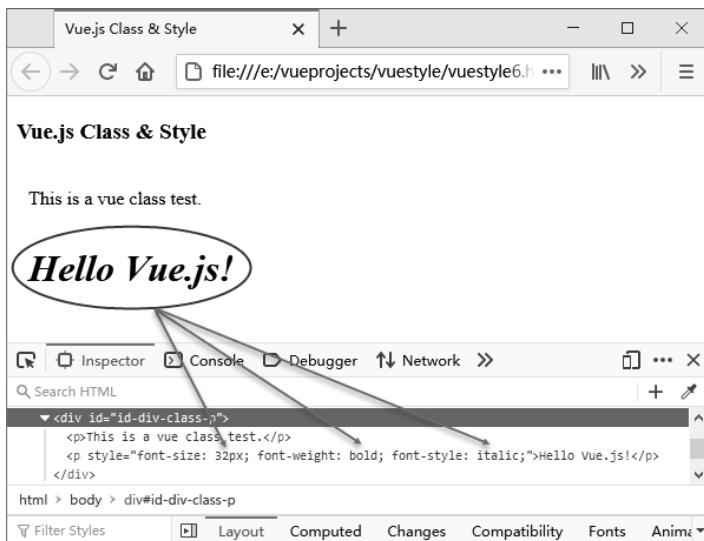


图 10.7 Vue.js 绑定静态内联样式 Style

如图 10.7 中的箭头和标识所示，第 16~18 行代码中所定义的 3 个字体内联样式，成功渲染到该段落<p>元素中。

### 10.3.2 绑定 Style 对象

在前一小节中，【代码 10-6】所使用的绑定静态 Style 虽然看上去更加直观，但如果需要定义的内联样式较多就会显得 HTML 代码很烦琐。因此，我们可以使用 Style 对象（内联样式对象）的方式来定义样式，这样 HTML 代码看上去会非常简洁。

下面的代码实例，是在【代码 10-6】的基础上修改而成的，将具体的内联样式通过 Style 对象方式进行改写（实现了同样的字体效果）。

**【代码 10-7】**（详见源代码 vuestyle 目录中的 vuestyle7.html 文件）

```
01 <div id="id-div-class-p">
02   <p>This is a vue class test.</p>
03   <p :style="myFontStyle">{{msg}}</p>
```

```

04  </div>
05  <script>
06      // Vue Entry
07      var vm = new Vue({
08          el: '#id-div-class-p',
09          data: {
10              msg: "Hello Vue.js!",
11              myFontStyle: {
12                  fontSize: '36px',
13                  fontWeight: 'bolder',
14                  fontStyle: 'italic'
15              }
16          }
17      })
18  </script>

```

### 【代码说明】

- 第 01~04 行代码中，通过<div>元素定义了一个层，层内定义一个段落<p>元素，通过文本插值模板引用一个对象（msg）。其中，在第 03 行代码中通过 v-bind 指令缩写方式绑定内联样式参数 style，其参数值为一个样式对象（myFontStyle）。
- 第 09~16 行代码中，通过 data 属性进行绑定数据操作。具体说明如下：
  - 第 10 行代码中定义了一个属性（msg），并进行初始化。该属性对应第 03 行代码中引用的对象（msg）。
  - 第 11~15 行代码中定义了一个对象（myFontStyle），该对象定义 3 个样式字段，分别为字体大小、字体粗细和字体风格，对应于第 03 行代码中段落<p>元素引用的样式对象（myFontStyle）。

接下来，运行测试 vuestyle.html 页面，效果如图 10.8 所示。

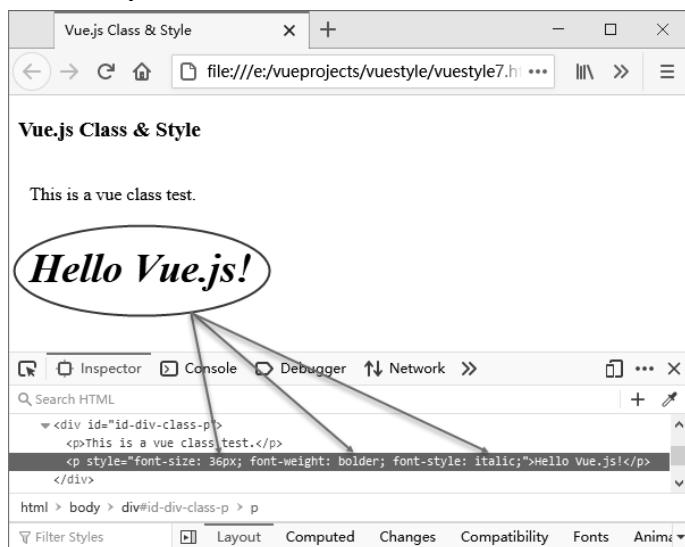


图 10.8 Vue.js 绑定静态内联样式 Style 对象

如图 10.8 中的箭头和标识所示，第 12~14 行代码中所定义的 3 个字体内联样式，成功渲染到该段落<p>元素中。

### 10.3.3 绑定多重值的 Style

从 Vue.js 框架的 v2.3.0 版本开始，支持设计人员为内联样式 style 绑定一个包含多个值的数组，该方式常用于提供多个带前缀的值。代码示例如下：

```
<div :style="{display: ['-webkit-box', '-ms-flexbox', 'flex']}"></div>
```

按照上面示例中代码的写法，浏览器只会渲染数组中最后一个被浏览器支持的值，也就是说，如果浏览器支持不带浏览器前缀的“flexbox”，那就只会按照样式“display: flex”进行渲染。

## 10.4 通过计算属性绑定样式

前面介绍过计算属性的使用方法，我们知道 Vue.js 框架提供的计算属性是一个方便而又强大的工具。下面，我们将介绍如何通过 Vue.js 框架的计算属性，实现样式绑定的应用，示例如下：

**【代码 10-8】**（详见源代码 vuestyle 目录中的 vuestyle8.html 文件）

```
01 <style>
02   p.font-medium {
03     font-size: 32px;
04   }
05   p.font-weight {
06     font-weight: bold;
07   }
08   p.font-italic {
09     font-style: italic;
10   }
11 </style>
12 <div id="id-div-class-p">
13   <p>This is a vue class test.</p>
14   <p :class="myFontComputed1">{ {msg} }</p>
15   <p :class="myFontComputed2">{ {msg} }</p>
16   <p :class="myFontComputed3">{ {msg} }</p>
17 </div>
18 <script>
19   // Vue Entry
20   var vm = new Vue({
21     el: '#id-div-class-p',
22     data: {
```

```

23         msg: "Hello Vue.js!"
24     },
25     computed: {
26         myFontComputed1: function() {
27             return {
28                 'font-medium': true,
29                 'font-weight': true,
30                 'font-italic': true
31             }
32         },
33         myFontComputed2: function() {
34             return {
35                 'font-medium': true,
36                 'font-weight': false,
37                 'font-italic': false
38             }
39         },
40         myFontComputed3: function() {
41             return {
42                 'font-medium': false,
43                 'font-weight': false,
44                 'font-italic': false
45             }
46         },
47     }
48 }
49 </script>

```

### 【代码说明】

- 第 01~11 行代码中，通过<style>标签定义了一组（3个）样式类，分别用于修饰不同字体大小、字体粗细和字体风格的段落。
- 第 14~16 行代码中定义了一组（3个）段落<p>元素，通过文本插值模板引用一个对象（msg）。同时，在每个段落<p>元素内部通过 v-bind 指令缩写方式绑定样式参数 class，其 class 的值分别引用 3 个对象（myFontComputed1、myFontComputed2 和 myFontComputed3）。
- 第 22~24 行代码中，通过 data 属性绑定数据操作。其中，第 23 行代码中定义了一个属性（msg），并进行初始化，该属性对应第 14~16 行代码中段落<p>元素引用的对象（msg）。
- 第 25~47 行代码中，通过 computed 属性定义了一组（共 3 个）计算属性（myFontComputed1、myFontComputed2 和 myFontComputed3），对应第 14~16 行代码每个段落<p>元素引用的样式对象（myFontComputed1、myFontComputed2 和 myFontComputed3）。

运行测试 vuestyle.html 页面，效果如图 10.9 所示。

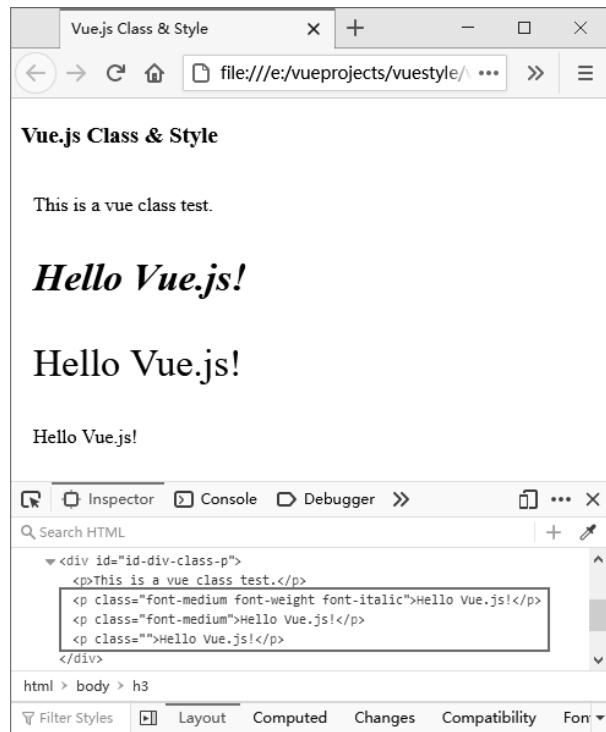


图 10.9 Vue.js 通过计算属性绑定样式

如图 10.9 中的标识所示，在浏览器控制台中可以看到：第 24~47 行代码计算属性定义的 3 个样式对象，成功渲染到第 14~16 行代码中定义的段落

元素中。

# 第 11 章

## Vue.js 组件基础

本章主要介绍 Vue.js 框架中组件（Component）方面的内容。目前，优秀的前端框架（例如：Angular、React 和 Vue.js）均支持组件的设计，这一点也正是 Vue.js 框架设计理念先进的主要原因所在。

通过本章的学习可以：

- 学会 Vue.js 框架设计基本组件的基础。
- 掌握 Vue.js 框架组件复用的设计方法。
- 了解 Vue.js 框架组件传递参数的方式。

### 11.1 Vue.js 全局组件

本节主要介绍 Vue.js 全局组件基本构成方面的内容，以及如何设计一个简单的、功能完整的 Vue 组件。

Vue.js 框架的全局组件，是通过 `Vue.Component` 命令来定义的，具体语法如下：

```
Vue.component(tagName, options)
```

在 `Vue.Component` 代码内，一般是先通过 `tagName` 来定义组件名称，然后通过 `options` 来配置组件内容（例如：通过 `template` 属性来定义 HTML 模板）。另外，Vue 组件还支持通过 `data` 属性来定义数据对象，然后将这些属性提供给 `template` 模板来使用。

下面通过一个简单的代码实例，介绍一下如何设计一个最基本的 Vue 全局组件，以及 Vue 组件的基本构成。

【代码 11-1】（详见源代码 vuecomp 目录中的 `vuecomp.html` 文件）

```
01 <div id="id-div-comp">
02   <comp-title></comp-title>
03   <comp-content></comp-content>
04   <comp-counter></comp-counter>
```

```

05  </div>
06  <script>
07      // Vue Global Component - title
08      Vue.component('comp-title', {
09          template: '<h3>Title - Vue.js Global Component</h3>'
10      });
11      // Vue Global Component - content
12      Vue.component('comp-content', {
13          template: '<p>This is a Vue.js component paragraph.</p>'
14      });
15      // Vue Global Component - button
16      Vue.component('comp-counter', {
17          data: function() {
18              return {
19                  count: 0
20              }
21          },
22          template: '<button v-on:click="count++">
23                  You clicked me {{ count }} times.
24              </button>'
25      });
26      // Vue Entry
27      var vm = new Vue({
28          el: '#id-div-comp'
29      });
30  </script>

```

### 【代码说明】

- 在第 01~05 行代码中，通过<div>元素定义了一个层，层内定义 3 个 Vue 组件。另外，这 3 个 Vue 组件分别模拟表示页面中几个比较常用的功能模块（标题、内容和按钮）。
- 第 08~10 行代码中定义的是标题 Vue 组件，组件名称定义为 comp-title。其中，在第 09 行代码中通过 template 属性定义组件的具体内容。
- 第 12~14 行代码中定义的是内容 Vue 组件，组件名称定义为 comp-content。其中，在第 13 行代码中通过 template 属性定义组件的具体内容。
- 第 16~25 行代码中定义的是按钮 Vue 组件，组件名称定义为 comp-counter。具体说明如下：
  - 第 17~21 行代码中，通过 data 属性绑定数据操作。在第 19 行代码中，通过函数方式返回一个属性（count），用于表示用户单击该按钮的次数。这里特别提醒一下，data 属性一般建议写成函数形式（不建议写成对象形式），这样可以避免组件之间产生不必要的相互影响。
  - 第 22~24 行代码中，通过 template 属性定义组件的具体内容。这里是一个按钮<button>组件，在代码中通过对属性（count）执行累加操作，实现记录用户单击按钮次数的操作。

下面运行测试 vuecomp.html 页面，页面初始效果如图 11.1 所示。

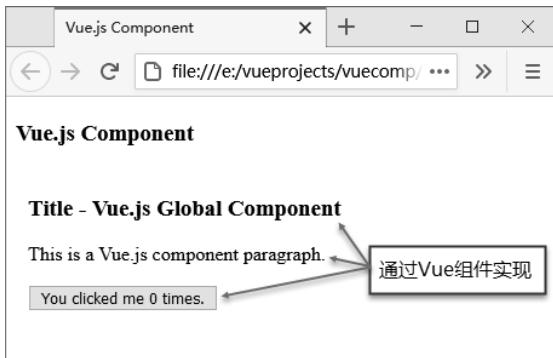


图 11.1 Vue.js 全局组件（一）

如图 11.1 中的箭头和标识所示，页面中成功渲染了第 08~10 行代码、第 12~14 行代码和第 16~25 行代码定义的三个 Vue 组件。

我们可以尝试单击一下页面中的按钮，观察一下通过 Vue 组件定义的按钮所实现的功能，效果如图 11.2 所示。

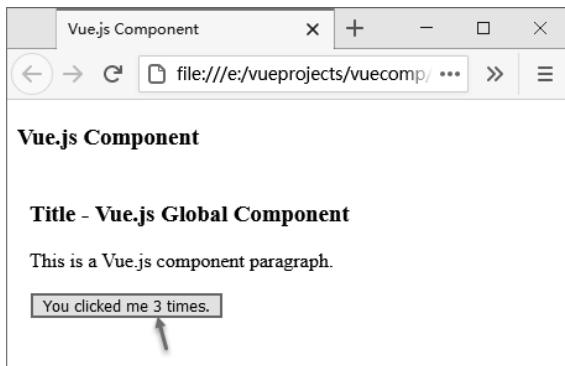


图 11.2 Vue.js 全局组件（二）

如图 11.2 中的箭头所示，在用户单击 3 次按钮后，按钮标题内容中的次数也随之进行更新（从 0 次到 3 次）。

## 11.2 Vue.js 局部组件

在 Vue.js 组件设计中，除了支持全局组件的设计之外，还支持局部组件的设计。二者的主要区别是，全局组件对于全部 Vue 实例均可用，而局部组件只能在注册该组件的 Vue 实例中使用。此外，二者在功能上基本一致。

在前一小节中，我们知道 Vue 全局组件是通过 `Vue.Component` 命令来定义的。而定义 Vue

局部组件的方式类似于定义对象，具体语法如下：

```
var tagName = { options: values }
```

这里的变量 tagName 定义的是组件名称，然后通过 options 对象来配置组件内容（例如：通过 data 属性来定义数据对象，然后通过 template 属性来定义 HTML 模板）。

我们将【代码 11-1】通过局部组件的方式进行改写，来实现一个相同的页面效果，示例如下：

**【代码 11-2】**（详见源代码 vuecomp 目录中的 vuecomp.html 文件）

```
01 <div id="id-div-comp">
02   <comp-title></comp-title>
03   <comp-content></comp-content>
04   <comp-counter></comp-counter>
05 </div>
06 <script>
07   // Vue Local Component - title
08   var CompTitle = {
09     template: '<h3>Title - Vue.js Local Component</h3>'
10   };
11   // Vue Local Component - content
12   var CompContent = {
13     template: '<p>This is a Vue.js component paragraph.</p>'
14   };
15   // Vue Local Component - button
16   var CompCounter = {
17     data: function() {
18       return {
19         count: 0
20       }
21     },
22     template: '<button v-on:click="count++">
23       You clicked me {{ count }} times.
24     </button>'
25   };
26   // Vue Entry
27   var vm = new Vue({
28     el: '#id-div-comp',
29     components: {
30       'comp-title': CompTitle,
31       'comp-content': CompContent,
32       'comp-counter': CompCounter
33     }
34   });
35 </script>
```

**【代码说明】**

- 在第 01~05 行代码中，通过<div>元素定义了一个层，层内定义 3 个 Vue 组件。另

外，这 3 个 Vue 组件分别模拟表示页面中几个比较常用的功能模块（标题、内容和按钮）。

- 第 08~10 行代码中定义的是标题 Vue 组件，组件名称定义为 CompTitle，注意这是通过局部组件方式实现的。其中，在第 09 行代码中通过 template 属性定义组件的具体内容。
- 第 12~14 行代码中定义的是内容 Vue 组件，组件名称定义为 CompContent，这也是一个局部组件。其中，在第 13 行代码中通过 template 属性定义了组件的具体内容。
- 第 16~25 行代码中定义的是按钮 Vue 组件，组件名称定义为 CompCounter，这同样是一个局部组件，与【代码 11-1】中全局组件的定义形式是类似的。
- 第 29~33 行代码中，在 Vue 构造函数中通过 components 属性来引入自定义的局部组件。具体说明如下：
  - 在第 30 行代码中，定义了一个 comp-title 属性，其属性值为自定义局部组件 CompTitle。
  - 在第 31 行代码中，定义了一个 comp-content 属性，其属性值为自定义局部组件 CompContent。
  - 在第 32 行代码中，定义了一个 comp-counter 属性，其属性值为自定义局部组件 CompContent。

下面运行测试 vuecomp.html 页面，页面初始效果如图 11.3 所示。

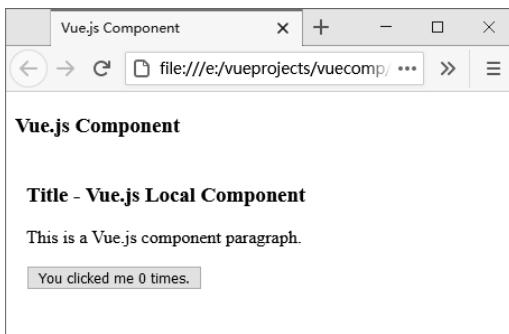


图 11.3 Vue.js 局部组件

如图 11.3 中所示，【代码 11-2】通过 Vue 局部组件实现与【代码 11-1】通过全局组件相同的页面效果。

## 11.3 通过 Prop 向子组件传递数据

Vue.js 组件设计提供了一个 Prop 属性，用来支持向子组件传递数据的功能。这是一个非常实用的设计，设计人员可以通过在组件的 Prop 属性中注册一些自定义属性，当一个值传递

给一个 Prop 的属性时，它就转变成了该组件实例的一个属性对象，从而也就实现了传递数据的功能。

Prop 属性的语法格式相对比较灵活，大致可以遵循下面的书写形式：

```
Vue.component(tagName, {
  props: [props]
  options...
})
```

注意，定义 Prop 属性时要写成 props 数组或对象的格式。

我们通过 Prop 的方式设计一个组件，来实现一个标题新闻的页面效果，示例如下：

**【代码 11-3】**（详见源代码 vuecomp 目录中的 vuecomp.html 文件）

```
01 <div id="id-div-comp-news">
02   <news-title title="News Title"></news-title>
03   <news-content content="Hello, Vue.js!"></news-content>
04   <news-author author="King"></news-author>
05 </div>
06 <script>
07   // Vue Global Component - title
08   Vue.component('news-title', {
09     props: ['title'],
10     template: '<h3>{{ title }}</h3>'
11   });
12   // Vue Global Component - content
13   Vue.component('news-content', {
14     props: ['content'],
15     template: '<p>This is news content : {{ content }}.</p>'
16   );
17   // Vue Global Component - author
18   Vue.component('news-author', {
19     props: ['author'],
20     template: '<p class="p-right">Edit by {{ author }}.</p>'
21   );
22   // Vue Entry
23   var vm = new Vue({
24     el: '#id-div-comp-news'
25   );
26 </script>
```

**【代码说明】**

- 在第 01~05 行代码中，通过<div>元素定义了一个层，层内定义 3 个 Vue 组件（分别模拟了新闻标题、新闻内容和新闻作者）。需要注意的是，在每个组件内都增加了一个属性（title、content 和 author）的定义，这个增加的属性就是使用 Prop 传递数据的关键。
- 第 08~11 行代码中定义的是新闻标题 Vue 组件，组件名称定义为 news-title。具体说明如下：

- 第 09 行代码中通过 props 属性注册一个自定义属性（title）。
- 第 10 行代码中通过 template 属性定义组件的具体内容，这里引用自定义属性（title）。
- 第 13~16 行代码中定义的是新闻内容 Vue 组件，组件名称定义为 news-content。具体说明如下：
  - 第 14 行代码中通过 props 属性注册一个自定义属性（content）。
  - 第 15 行代码中通过 template 属性定义组件的具体内容，这里引用自定义属性（content）。
- 第 18~21 行代码中定义的是新闻作者 Vue 组件，组件名称定义为 news-author。具体说明如下：
  - 第 19 行代码中通过 props 属性注册一个自定义属性（author）。
  - 第 20 行代码中通过 template 属性定义组件的具体内容，这里引用自定义属性（author）。

下面运行测试 vuecomp.html 页面，页面初始效果如图 11.4 所示。



图 11.4 通过 Prop 向子组件传递数据

如图 11.4 中的标识所示，从页面中渲染的内容可以看出来，在第 02~04 行代码中定义的属性（title、content 和 author）值成功传递给每一个新闻子组件（news-title、news-content 和 news-author）了。

在【代码 11-3】中，我们使用静态的 Prop 属性设计新闻组件。接下来，我们通过使用动态的 Prop 属性来设计一个逆序字符串的组件，以增强一些互动的功能效果，示例如下：

#### 【代码 11-4】（详见源代码 vuecomp 目录中的 vuecomp.html 文件）

```

01 <div id="id-div-comp-reverse">
02   Enter: <input v-model="instr">
03   <reverse-string v-bind:msg="instr"></reverse-string>
04 </div>
05 <script>
06   // Vue Global Component - reverse string
07   Vue.component('reverse-string', {

```

```

08     props: ['msg'],
09     computed: {
10         revstr: function() {
11             return this.msg.split('').reverse().join('');
12         }
13     },
14     template: '<p>Reverse string: {{ revstr }}.</p>'
15 );
16 // Vue Entry
17 var vm = new Vue({
18     el: '#id-div-comp-reverse',
19     data: {
20         instr: "Pls enter string to reverse..."
21     }
22 });
23 </script>

```

### 【代码说明】

- 在第01~04行代码中，通过<div>元素定义了一个层，层内定义一个文本输入框和一个用于显示逆序字符串的Vue组件。具体说明如下：
  - 第02行代码中，通过<input>元素定义了一个文本输入框，并通过v-model指令双向绑定一个对象（instr）。
  - 第03行代码中，通过Vue组件reverse-string定义一个用于显示逆序字符串的段落，并通过v-bind指令绑定一个参数属性（msg），其参数值为第02行代码中引用的对象（instr）。
- 第07~15行代码中定义的是逆序字符串Vue组件，组件名称定义为reverse-string。具体说明如下：
  - 第08行代码中通过props属性注册一个自定义属性（msg），对应于第03行代码绑定的参数属性（msg）。
  - 第09~13行代码通过computed属性定义一个计算属性（revstr），实现了逆序字符串的操作。
  - 第14行代码中通过template属性定义组件的具体内容，这里通过引用计算属性（revstr）来显示逆序的字符串内容。
- 在第17~22行代码的Vue构造函数中，通过data属性绑定数据操作。其中，第20行代码中定义了一个属性（instr），并进行初始化，该属性对应第02行代码中文本输入框<input>元素双向绑定的对象（instr）。

接下来，运行测试vuecomp.html页面，页面初始效果如图11.5所示。

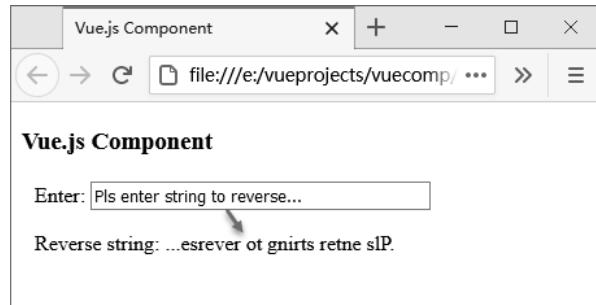


图 11.5 通过动态 Prop 实现逆序字符串组件（一）

如图 11.5 中的箭头所示，从页面中显示的内容可以看出，文本输入框中的默认数据已经被逆序显示了。

我们尝试在文本输入框中填入一些随意的字符串，页面效果如图 11.6 所示。



图 11.6 通过动态 Prop 实现逆序字符串组件（二）

如图 11.6 中的箭头和标识所示，在文本输入框中输入的字符串（"abcdefghijklmn"），已经被自动逆序显示在页面中了。

# 第 12 章

## Vue.js 路由

本章主要介绍 Vue.js 框架中路由的内容。路由允许设计人员通过不同的 URL 访问不同的内容，使用 Vue.js 路由需要载入 vue-router 库。使用 Vue.js 路由最大的好处就是，可以实现多视图的单页 Web 应用（SPA）。

通过本章的学习可以：

- 掌握 Vue.js 路由库 vue-router 的安装与使用。
- 掌握基于 vue-router 路由的导航设计。
- 掌握 vue-router 路由配置的详细信息。

### 12.1 安装 vue-router 库的方法

目前，在 Vue.js 框架下使用路由功能主要推荐使用 vue-router 库，该库实现了完整且强大的路由功能。本节主要介绍安装 vue-router 库的基本方式，以及使用该路由库的基本方法。

在 Vue.js 框架下安装 vue-router 库，主要有 CDN 和 NPM 两种最常用的方式。

对于 CDN 方式，可以直接下载 vue-router 库文件，也可以直接应用 URL 地址。具体地址如下：

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

对于 NPM 方式，直接使用包管理工具命令安装即可，这里推荐使用国内镜像（如：淘宝镜像）。具体命令如下：

```
npm install vue-router
```

或者

```
cnpm install vue-router // 淘宝镜像方式
```

通过上面的方法安装好 vue-router 路由库后，就可以在 Vue.js 框架使用该库进行路由应用

的开发了。

## 12.2 基于 vue-router 库开发单页面应用

基于 Vue.js 框架和 vue-router 库可以设计出很简单的单页面应用，这里主要通过使用一个<router-link>组件进行操作。这个<router-link>组件用于设置一个导航链接，可以实现切换不同 HTML 内容或页面的功能。在<router-link>组件中，通过一个 to 属性可以设定目标地址，也就是要显示的内容。

下面示例是通过在 Vue.js 框架下使用 vue-router 库来配置组件和路由映射，实现一个单页面应用。

【代码 12-1】（详见源代码 vuerouter 目录中的 vuerouter.html 文件）

```
01 <div id="id-div-vue-router">
02   <p>
03     <!-- 使用 router-link 组件来导航 -->
04     <router-link to="/home" active-class="active">Home</router-link>
05     <router-link to="/news" active-class="active">News</router-link>
06     <router-link to="/about" active-class="active">About</router-link>
07   </p>
08   <!-- 路由出口 -->
09   <router-view></router-view>
10 </div>
11 <script>
12   // 路由组件
13   const Home = {
14     template: '<div>This is home page.</div>'
15   };
16   const News = {
17     template: '<div>This is news page.</div>'
18   };
19   const About = {
20     template: '<div>About us.</div>'
21   };
22   // 定义路由配置
23   const routes = [
24     { path: '/home',
25       component: Home
26     },
27     { path: '/news',
28       component: News
29     },
30     { path: '/about',
31       component: About
32     }
33   ];
34   // 导航守卫
35   router.beforeEach((to, from, next) => {
36     if (to.meta.title) {
37       document.title = to.meta.title
38     }
39     next()
40   })
41 </script>
```

```

32      },
33      {
34         path: '*',
35         redirect: '/home'
36     ];
37     // 创建 router 实例
38     const router = new VueRouter({
39         routes: routes
40     });
41     // Vue Enter
42     const app = new Vue({
43         router
44     }).$mount('#id-div-vue-router');
45 </script>

```

### 【代码说明】

- 第 01~10 行代码中，通过<div>元素定义了一个层，层内定义一组（3个）路由链接<router-link>组件，在每个<router-link>组件内部通过 to 属性指定具体的路由目标地址。然后，第 09 行代码中通过<router-view>组件定义路由出口，用于显示路由匹配到此处的组件。
- 第 23~35 行代码中，定义了一组路由组件的配置对象（routers）。其中，通过 path 属性定义匹配组件的路径，通过 component 属性指定匹配组件的名称，还有就是通过 component 属性指定默认路由的组件名称。
- 第 23~35 行代码中，通过“new VueRouter()”构造函数创建路由实例（router）。其中第 38 行代码定义的内容，负责将路由组件的配置对象（routers）传递给路由实例（router）。
- 第 41~43 行代码中，通过“new Vue()”构造函数创建 Vue 实例（app）。其中第 42 行代码定义的内容，负责将路由实例（router）传递给 Vue 实例（app），然后第 43 行代码负责挂载页面组件到 Vue 实例（app）。

下面运行测试 vuestyle.html 页面，页面初始效果如图 12.1 所示。

如图 12.1 中的标识所示，页面中显示的是第 33~34 行代码定义的默认路由组件。我们尝试单击一下“News”链接，页面效果如图 12.2 所示。



图 12.1 vue-router 路由实例（一）



图 12.2 vue-router 路由实例（二）

如图 12.2 中的箭头和标识所示，当单击“News”链接后，页面切换到了“News”路由组件界面，在这个过程中页面是没有刷新操作的。读者可以继续单击“About”链接测试，效果是一样的。这就是通过 vue-router 库设计开发一个单页面应用的最基本过程。

## 12.3 基于 vue-router 库实现动态路由

在前一节中，我们设计的单页面应用只具有静态路由的功能。本节，我们介绍一下如何基于 Vue.js 框架和 vue-router 库，设计一个具有动态路由功能的单页面应用。

下面的示例是在【代码 12-1】的基础上修改而成的，主要添加了动态路由功能。

**【代码 12-2】**（详见源代码 vuerouter 目录中的 vuerouter.html 文件）

```

01 <div id="id-div-vue-router">
02   <p>
03     <!-- 使用 router-link 组件来导航-->
04     <router-link to="/home" active-class="active">Home</router-link>
05     <router-link to="/user/king" active-class="active">King</router-link>
06     <router-link to="/user/tina" active-class="active">Tina</router-link>
07     <router-link to="/user/cici" active-class="active">Cici</router-link>
08   </p>
09   <!-- 路由出口 -->
10   <router-view></router-view>
11 </div>
12 <script>
13   // 路由组件
14   const Home = {
15     template: '<div>This is home page.</div>'
16   };
17   const User = {
18     template: '<div>This is {{ $route.params.name }} page.</div>'
19   };
20   // 定义路由配置
21   const routes = [
22     {
23       path: '/home',
24       component: Home
25     },
26     {
27       path: '/user/:name',
28       component: User
29     },
30     {
31       path: '*',
32       redirect: '/home'
33     }];
34   // 创建 router 实例
35   const router = new VueRouter({
36     //routes // (缩写) 相当于 routes: routes

```

```

34         routes: routes
35     });
36     // Vue Enter
37     const app = new Vue({
38         router
39     }).$mount('#id-div-vue-router');
40 </script>

```

### 【代码说明】

- 这段代码与【代码 12-1】的主要区别是第 21~30 行代码中定义的路由组件配置对象 (routers)。其中，第 25~26 行代码定义的 User 组件中，path 属性配置的就是动态路径参数 name (注意要以“:”开头)。
- 第 01~11 行代码中，通过<div>元素定义了一个层，层内定义一组 (4 个) 路由链接<router-link>组件，在每个<router-link>组件内部通过 to 属性指定具体的路由目标地址。注意，第 01~11 行代码中定义的 to 属性，先是指定相同的路径 (/user)，再是配置不同的动态路径参数 (/king、/tina 和/cici)，这就是动态路由的关键之处。然后，第 09 行代码中通过<router-view>组件定义了路由出口，用于显示路由匹配到此处的组件。
- 再看一下第 17~19 行代码定义的路由组件 (User)，通过\$route.params 对象获取 name 参数属性。

接下来，运行测试 vuestyle.html 页面，页面初始效果如图 12.3 所示。

如图 12.3 中的标识所示，页面中显示的是第 22~23 行代码定义的默认路由组件。我们尝试单击一下“King”链接，页面效果如图 12.4 所示。

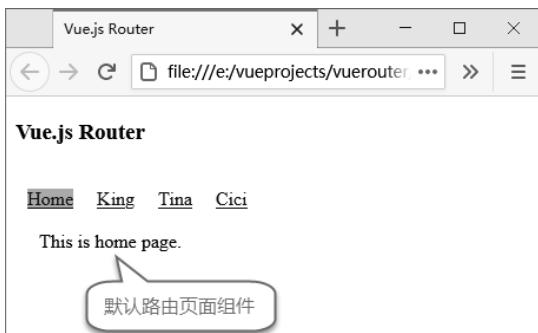


图 12.3 vue-router 动态路由实例（一）



图 12.4 vue-router 动态路由实例（二）

如图 12.4 中的箭头和标识所示，当单击“King”链接后，页面切换到“/User/king”动态路由组件界面，同样在这个过程中页面是没有刷新操作的。读者可以继续单击“Tina”链接和“Cici”链接测试，效果是一样的。

# 第 13 章

## 项目实战：基于 Vue.js+ Node.js 实现学生成绩管理系统

本章介绍一个基于 Vue.js 框架和 Node.js 框架实现的学生成绩管理系统。该系统设计了最基本的功能模块，包括学生成绩展示模块、新增模块、编辑模块和删除模块。我们设计该实战项目的目标就是，帮助读者快速掌握基于 Vue.js 框架的开发流程，并让读者能够体会到 Vue.js 框架在前端 Web 开发中的优势所在。

通过本章的学习可以：

- 掌握基于 Vue.js 框架构建项目应用。
- 掌握 Vue.js 框架中基于组件的设计模式。
- 掌握基于 vue-router 路由的路径导航。

### 13.1 学生成绩管理系统组织架构设计

基于 Vue.js 框架设计的学生成绩管理系统应用，主要包括基于 Vue 组件设计的学生成绩展示模块、新增模块、编辑模块和删除模块，功能模块之间的导航由 vue-router 路由进行操作，而后台数据库则使用 HTML 5 的 localStorage 本地存储来模拟完成。

关于本应用的组织架构，如图 13.1 所示。

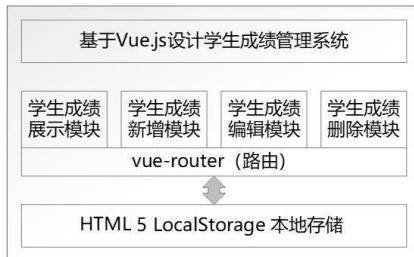


图 13.1 学生成绩管理系统组织架构图

## 13.2 构建项目应用框架

(1) 安装最新的稳定版 Node.js 框架、NPM 管理工具和 Webpack 构建工具。

安装的具体方法在前面的章节中已经介绍过，必要时可以参考相关工具官网上的文档说明，这里就不再赘述了。在上述框架和工具安装完毕后，一定要检查操作系统的环境来查看是否已经正确安装，具体命令如下：

```
node -v          // check node environment
npm -v          // check npm environment
webpack -v      // check webpack environment
```

如果上述框架和工具安装成功，那么在命令行控制台中运行上述命令后，会显示正确的框架和工具的版本号，如图 13.2 所示。

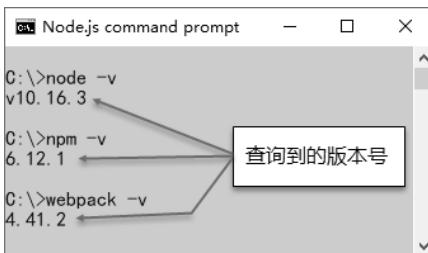


图 13.2 检查相关框架和工具的安装环境

如图 13.2 中的箭头和标识所示，查询后如果能够显示正确的版本号，就表示 Node.js、NPM 和 Webpack 在操作系统环境中是正确可用的。

(2) 安装 vue-cli 工具。

因为我们是通过 Vue.js 框架开发应用程序，所以必须安装 vue-cli “脚手架” 工具，通过这个工具自动搭建好 Vue 应用框架所需的大部分组件。安装命令如下：

```
npm install -g vue-cli
```

同样地，在 vue-cli 工具安装完毕后，一定要检查操作系统的环境来查看是否已经正确安

装，命令如下：

```
vue -V      // check vue environment, parameters must be '-V'
```



上述命令的参数“-V”一定要是大写字母。

如果上述 vue-cli 工具安装成功，那么在命令行控制台中运行上述命令后会显示正确的框架和工具的版本号，如图 13.3 所示。

```
C:\>node -v
v10.16.3

C:\>npm -v
6.12.1

C:\>webpack -v
4.41.2

C:\>vue -V
@vue/cli 4.5.2
```

图 13.3 检查相关框架和工具的安装环境

如图 13.3 中的箭头和标识所示，查询后如果能够显示正确的版本号，就表示 vue-cli 工具在操作系统环境中是正确可用的。

(3) 通过 vue-cli “脚手架” 工具自动构建 Vue 应用程序的框架，命令如下：

```
vue init webpack vuestu    // create by webpack,
vuestu is project name.
```

上述命令中，参数 init 表示初始化 Vue 应用框架，加上 webpack 表示基于该工具来构建，vuestu 则是具体项目应用的名称。

在上述命令中加上 Webpack 工具有很多好处，Vue.js 框架会自动构建 Webpack 工具的配置，避免了设计人员后续再进行手动配置 Webpack 工具的繁琐工作。同时，借助 Webpack 工具的 webpack-web-server 功能，还可以自动搭建一个简单的 Web 服务器进行调试。

如果 Vue 应用框架构建成功，我们就可以在本地项目目录（事先确定好的项目位置）中查看到该应用的整体架构了。接下来，我们通过 VS Code 工具查看一下应用架构，如图 13.4 所示。

如图 13.4 中的箭头和标识所示，vue-cli 工具自动

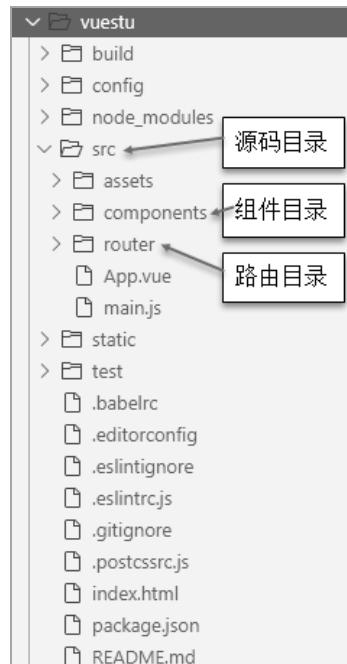


图 13.4 通过 VS Code 查看项目应用架构

构建好了整个项目应用架构，为设计人员避免了大量重复的繁琐工作。

## 13.3 后台数据结构

这里，我们在后台数据结构上选择使用简洁快速的 HTML 5 localStorage 本地存储方式。数据字段也尽量精简，只包括了学生 id、姓名、性别和语数英三科成绩，具体如下：

|          |           |
|----------|-----------|
| id:      | 学生 id，唯一项 |
| name:    | 学生姓名      |
| gender:  | 学生性别      |
| chinese: | 语文成绩      |
| math:    | 数学成绩      |
| english: | 英语成绩      |

localStorage 本地存储方式与传统的关系型数据库不同，更接近最新的非关系型数据库（实际上也有差别）。我们选择通过 localStorage 本地存储方式进行数据存储，就是因为其简洁与快速。

操作 localStorage 本地存储的方式也很简单，通过支持 HTML 5 的浏览器控制台工具就可以实现，如图 13.5 所示。

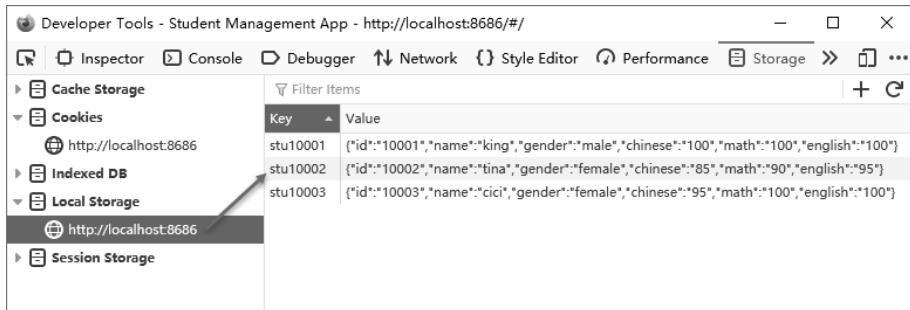


图 13.5 查看 localStorage 本地存储

如图 13.5 中的箭头所示，通过浏览器控制台工具中“Local Storage”选项，就可以查询到后台的数据结构。

## 13.4 功能模块组件设计

在前文中已经介绍了，该学生成绩管理系统应用主要包括基于 Vue 组件设计的学生成绩展示模块、新增模块、编辑模块和删除模块。下面我们分别介绍这几个基于 Vue.js 框架设计

的组件模块。

首先介绍学生成绩信息展示模块以及删除模块，代码如下：

**【代码 13-1】**（详见源代码 vuestu 目录中的 src\components\StudentInfo.vue 文件）

```

01 <template>
02   <div>
03     <h3>{{ msg }}</h3>
04     <h4>Student Scores Table</h4>
05     <table>
06       <tr>
07         <th>No</th>
08         <th>Id</th>
09         <th>Name</th>
10         <th>Gender</th>
11         <th>Chinese</th>
12         <th>Math</th>
13         <th>English</th>
14         <th>Admin</th>
15     </tr>
16     <tr v-for="(s, index) in stu" :key="index">
17       <td>{{ index+1 }}</td>
18       <td>{{ s.id }}</td>
19       <td>{{ s.name }}</td>
20       <td>{{ s.gender }}</td>
21       <td>{{ s.chinese }}</td>
22       <td>{{ s.math }}</td>
23       <td>{{ s.english }}</td>
24       <td>
25         <router-link to="/insert" active-class="active">
26           Insert
27         </router-link>
28         <router-link :to="'/edit/' + s.id" active-class="active">
29           Edit
30         </router-link>
31         <a href @click="del stu info(s.id)" active-class="active">
32           Del
33         </a>
34       </td>
35     </tr>
36   </table>
37 </div>
38 </template>
39 <script>
40 export default {
41   name: "StudentInfo",
42   data() {
43     return {
44       msg: "Welcome to Student Management App",
45     };

```

```

46     },
47     computed: {
48       stu: function () {
49         let len = localStorage.length;
50         let stuinfo;
51         let stu = Array();
52         for (let i = 0; i < len; i++) {
53           let stukey = localStorage.key(i);
54           if (stukey.substr(0, 3) == "stu") {
55             stuinfo = JSON.parse(localStorage.getItem(localStorage.key(i)));
56             stu.push(stuinfo);
57           }
58         }
59         return stu;
60       },
61     },
62     methods: {
63       del_stu_info: function (stuid) {
64         localStorage.removeItem("stu" + stuid);
65       },
66     },
67   };
68 </script>

```

**【代码说明】**

- 第 01~38 行代码中，通过<template>元素定义了一个虚拟模板，内部包含一个用于显示学生成绩信息的表格<table>元素，以及一组用于操作新增、编辑和删除学生成绩数据的路由和链接。
- 第 40~67 行代码中，定义并导出一个学生成绩信息组件（StudentInfo），该组件实现学生成绩信息的初始化与显示。其中，第 63~65 行代码定义的方法（del\_stu\_info），实现删除学生成绩信息的功能。

接下来，再介绍一下学生成绩信息新增模块，代码如下：

**【代码 13-2】**（详见源代码 vuestu 目录中的 src\components\StudentInsert.vue 文件）

```

01 <template>
02   <div class="divCen">
03     <h3>{{ msg }}</h3>
04     <h4>Student Scores Insert</h4>
05     <p class="p-right">
06       <router-link to="/" active-class="active">Back to home</router-link>
07     </p>
08     <table>
09       <tr>
10         <th>Id</th>
11         <td>
12           <input type="text" v-model="id" />
13         </td>

```

```
14      </tr>
15      <tr>
16          <th>Name</th>
17          <input type="text" v-model="name" />
18      </tr>
19      <tr>
20          <th>Gender</th>
21          <input type="text" v-model="gender" />
22      </tr>
23      <tr>
24          <th>Chinese</th>
25          <input type="text" v-model="chinese" />
26      </tr>
27      <tr>
28          <th>Math</th>
29          <input type="text" v-model="math" />
30      </tr>
31      <tr>
32          <th>English</th>
33          <input type="text" v-model="english" />
34      </tr>
35  </table>
36  <p>
37      <button @click="btn_add_stuinfo">Add to Save</button>
38  </p>
39 </div>
40 </template>
41 <script>
42 export default {
43     name: "StudentInsert",
44     data() {
45         return {
46             msg: "Welcome to Student Management App",
47             id: "",
48             name: "",
49             gender: "",
50             chinese: "",
51             math: "",
52             english: "",
53         };
54     },
55     created: function () {
56         let len = localStorage.length;
57         let stuinfo;
58         let id_max = 10001;
59         for (let i = 0; i < len; i++) {
60             let stukey = localStorage.key(i);
61             if (stukey.substr(0, 3) == "stu") {
62                 stuinfo = JSON.parse(localStorage.getItem(localStorage.key(i)));
63                 if (id_max < stuinfo.id) {
64                     id_max = stuinfo.id;
```

```

65         }
66     }
67   }
68   this.id = parseInt(id_max) + 1;
69 },
70 methods: {
71   btn_add_stuinfo: function () {
72     let oStu = {
73       id: this.id,
74       name: this.name,
75       gender: this.gender,
76       chinese: this.chinese,
77       math: this.math,
78       english: this.english,
79     };
80     var key = "stu" + oStu.id;
81     var stu = JSON.stringify(oStu);
82     localStorage.setItem(key, stu);
83     this.$router.push({ path: "/info" });
84   },
85 },
86 };
87 </script>

```

### 【代码说明】

- 第 01~40 行代码中，通过<template>元素定义了一个虚拟模板，内部包含一个用于新增学生成绩信息的表格<table>元素，以及一个用于操作新增学生成绩信息的按钮<button>元素。
- 第 42~86 行代码中，定义并导出一个新增学生成绩信息组件（StudentInsert），该组件实现新增学生成绩信息的功能。

最后，再介绍一下学生成绩信息编辑模块，代码如下：

### 【代码 13-3】（详见源代码 vuestu 目录中的 src\components\StudentEdit.vue 文件）

```

01 <template>
02   <div class="divCen">
03     <h3>{{ msg }}</h3>
04     <h4>Student Scores Edit</h4>
05     <p class="p-right">
06       <router-link to="/" active-class="active">Back to home</router-link>
07     </p>
08     <table>
09       <tr>
10         <th>Id</th>
11         <td>
12           <input type="text" :value="id" readonly />
13         </td>
14       </tr>

```

```
15      <tr>
16        <th>Name</th>
17        <input type="text" v-model="name" />
18      </tr>
19      <tr>
20        <th>Gender</th>
21        <input type="text" v-model="gender" />
22      </tr>
23      <tr>
24        <th>Chinese</th>
25        <input type="text" v-model="chinese" />
26      </tr>
27      <tr>
28        <th>Math</th>
29        <input type="text" v-model="math" />
30      </tr>
31      <tr>
32        <th>English</th>
33        <input type="text" v-model="english" />
34      </tr>
35    </table>
36    <p>
37      <button @click="btn_edit_stuinfo">Edit to Save</button>
38    </p>
39  </div>
40 </template>
41 <script>
42 export default {
43   name: "StudentEdit",
44   data() {
45     return {
46       msg: "Welcome to Student Management App",
47       id: "",
48       name: "",
49       gender: "",
50       chinese: "",
51       math: "",
52       english: "",
53     };
54   },
55   created: function () {
56     this.id = this.$route.params.id;
57     let stuinfo = JSON.parse(localStorage.getItem("stu" + this.id));
58     this.name = stuinfo.name;
59     this.gender = stuinfo.gender;
60     this.chinese = stuinfo.chinese;
61     this.math = stuinfo.math;
62     this.english = stuinfo.english;
63   },
64   methods: {
65     btn_edit_stuinfo: function () {
```

```

66     let oStu = {
67       id: this.id,
68       name: this.name,
69       gender: this.gender,
70       chinese: this.chinese,
71       math: this.math,
72       english: this.english,
73     };
74     var key = "stu" + oStu.id;
75     var stu = JSON.stringify(oStu);
76     localStorage.setItem(key, stu);
77     this.$router.push({ path: "/info" });
78   },
79 },
80 };
81 </script>

```

**【代码说明】**

- 第 01~40 行代码中，通过<template>元素定义了一个虚拟模板，内部包含一个用于编辑学生成绩信息的表格<table>元素，以及一个用于操作编辑学生成绩信息的按钮<button>元素。
- 第 42~80 行代码中，定义并导出一个编辑学生成绩信息组件（StudentEdit），该组件实现编辑学生成绩信息的功能。

## 13.5 功能模块路由设计

功能模块之间的导航可以利用 vue-router 路由完成。下面，我们介绍一下本应用基于 vue-router 的路由设计，代码如下：

**【代码 13-4】**（详见源代码 vuestu 目录中的 src\router\index.js 文件）

```

01 import Vue from 'vue'
02 import Router from 'vue-router'
03 import StudentInfo from '@/components/StudentInfo'
04 import StudentInsert from '@/components/StudentInsert'
05 import StudentEdit from '@/components/StudentEdit'
06 // Vue use Router
07 Vue.use(Router)
08 // export Router
09 export default new Router({
10   routes: [
11     {
12       path: '/',
13       name: 'StudentInfo',
14       component: StudentInfo

```

```

14     },
15     path: '/info',
16     name: 'StudentInfo',
17     component: StudentInfo
18   },
19   {
20     path: '/insert',
21     name: 'StudentInsert',
22     component: StudentInsert
23   },
24   {
25     path: '/edit/:id',
26     name: 'StudentEdit',
27     component: StudentEdit
28   ]
29 })

```

### 【代码说明】

- 第 01~05 行代码中，通过 import 命令导入必要的模块组件。
- 第 07 行代码中，通过 Vue.use()命令注入 Router 路由组件（关键步骤）。
- 第 09~27 行代码中，定义并导出一个路由对象（Router），该对象定义了一组基于功能模块的路由规则信息。

## 13.6 测试应用

我们测试一下学生成绩管理系统应用（vuestu），页面初始效果如图 13.6 所示。

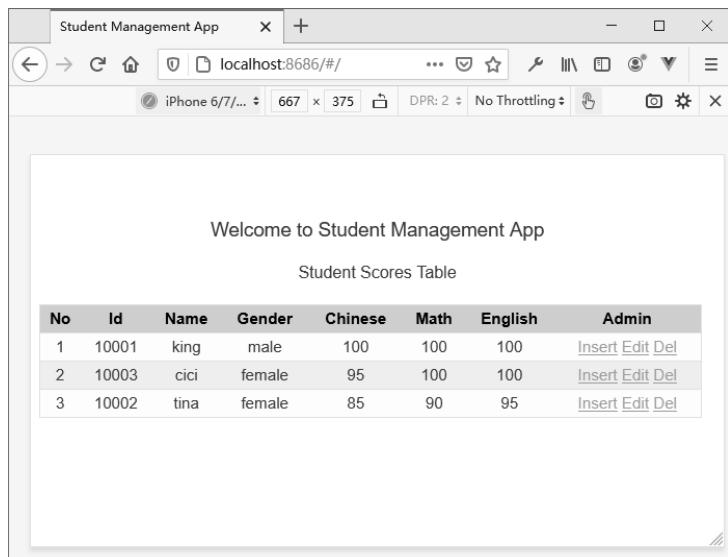


图 13.6 测试学生成绩管理系统（一）

如图 13.6 中所示，我们尝试单击一下“insert”新增链接，页面会路由跳转到新增学生成绩信息模块，页面效果如图 13.7 所示。

|                |                                    |
|----------------|------------------------------------|
| <b>Id</b>      | <input type="text" value="10004"/> |
| <b>Name</b>    | <input type="text" value="tom"/>   |
| <b>Gender</b>  | <input type="text" value="male"/>  |
| <b>Chinese</b> | <input type="text" value="70"/>    |
| <b>Math</b>    | <input type="text" value="75"/>    |
| <b>English</b> | <input type="text" value="65"/>    |

[Add to Save](#)

图 13.7 测试学生成绩管理系统（二）

如图 13.7 中所示，我们新增一条学生成绩信息，然后单击表单下面的“Add to Save”按钮，系统会自动添加该条信息，并返回学生信息浏览首页，页面效果如图 13.8 所示。

| No | Id    | Name | Gender | Chinese | Math | English | Admin   |
|----|-------|------|--------|---------|------|---------|---|
| 1  | 10001 | king | male   | 100     | 100  | 100     | <a href="#">Insert</a> <a href="#">Edit</a> <a href="#">Del</a> |
| 2  | 10003 | cici | female | 95      | 100  | 100     | <a href="#">Insert</a> <a href="#">Edit</a> <a href="#">Del</a> |
| 3  | 10002 | tina | female | 85      | 90   | 95      | <a href="#">Insert</a> <a href="#">Edit</a> <a href="#">Del</a> |
| 4  | 10004 | tom  | male   | 70      | 75   | 65      | <a href="#">Insert</a> <a href="#">Edit</a> <a href="#">Del</a> |

图 13.8 测试学生成绩管理系统（三）

如图 13.8 中所示，新增的学生成绩信息已经显示出来了。然后，我们可以再尝试单击“Edit”链接去编辑修改一下该条信息，页面效果如图 13.9 所示。

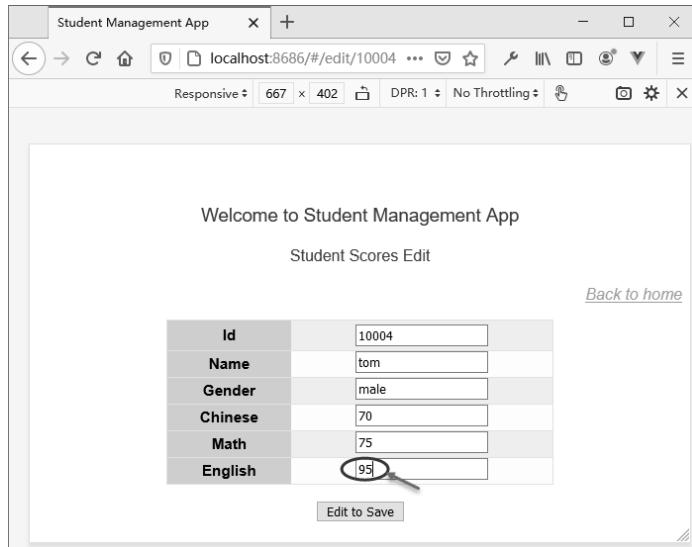


图 13.9 测试学生成绩管理系统（四）

如图 13.9 中的箭头所示，我们可以选择一项或几项学生成绩信息进行修改并提交，系统会自动保存该条信息，并返回学生信息浏览首页，页面效果如图 13.10 所示。

The screenshot shows a browser window titled 'Student Management App' at the URL 'localhost:8686/#/info'. The page displays a table of student scores. The table has columns: No, Id, Name, Gender, Chinese, Math, English, and Admin. The English column for student ID 10004 is highlighted with a red circle and an arrow pointing to the 'Edit' link in the Admin column.

| No | Id    | Name | Gender | Chinese | Math | English | Admin   |
|----|-------|------|--------|---------|------|---------|---|
| 1  | 10001 | king | male   | 100     | 100  | 100     | <a href="#">Insert</a> <a href="#">Edit</a> <a href="#">Del</a> |
| 2  | 10003 | cici | female | 95      | 100  | 100     | <a href="#">Insert</a> <a href="#">Edit</a> <a href="#">Del</a> |
| 3  | 10002 | tina | female | 85      | 90   | 95      | <a href="#">Insert</a> <a href="#">Edit</a> <a href="#">Del</a> |
| 4  | 10004 | tom  | male   | 70      | 75   | 95      | <a href="#">Insert</a> <a href="#">Edit</a> <a href="#">Del</a> |

图 13.10 测试学生成绩管理系统（五）

如图 13.10 中的箭头所示，刚刚修改的学生成绩信息已经保存下来了。另外，读者可以自行测试一下删除功能，这里就不具体介绍了。

# 第 14 章

## 项目实战：基于 Vue.js+ Node.js 实现城市信息查询系统

本章介绍一个基于 Vue.js 框架和 Node.js 框架实现的单页面应用（SPA）——全国城市信息查询系统。该城市信息查询系统包括省份查询模块和重点城市查询模块。这些功能模块利用 Vue.js 框架的组件功能来实现，功能模块之间的交互利用 vue-router 路由操作来完成，而后台数据则使用 jsonp 方式获取由免费的公共 API 接口所提供的全国省市信息。

通过本章的学习可以：

- 掌握基于 Vue.js 框架组件设计单页面（SPA）应用的方式。
- 掌握在 Vue.js 框架处理 JSON 格式数据的方法。
- 掌握基于 jsonp 方式进行跨域请求的步骤。

### 14.1 全国城市信息查询系统组织架构设计

基于 Vue.js 框架设计的全国城市信息查询系统是一个纯粹的单页面（SPA）应用，主要包括基于 Vue 组件设计的全国城市信息查询模块，模块之间的基于 vue-router 路由的导航操作（设计单页面的关键），后台数据则通过 jsonp 方式获取由免费的公共 API 接口提供的全国城市数据信息。

本应用的组织架构，如图 14.1 所示。

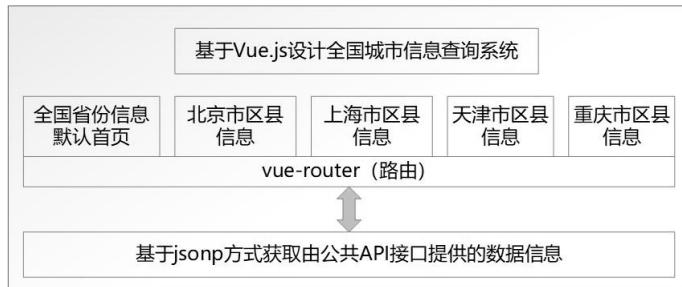


图 14.1 全国城市信息查询系统组织架构图

## 14.2 构建项目应用框架

构建项目应用框架与前一章中的步骤基本相同。因为整个项目应用基于 Vue.js 框架设计，还是要通过 vue-cli “脚手架” 工具自动搭建 Vue 应用程序的框架，命令如下：

```
vue init webpack vuespa      // create by webpack,
vuespa is project name.
```

上述命令中，参数 init 表示初始化 Vue 应用框架，加上 webpack 表示使用该工具来构建，vuespa 是具体项目应用的名称。

如果 Vue 应用框架构建成功，我们就可以在本地项目目录（事先确定好的项目位置）中查看到该应用的整体架构了。我们通过 VS Code 工具查看一下应用架构，如图 14.2 所示。

如图 14.2 中的箭头和标识所示，vue-cli 工具自动构建好整个项目应用的架构，在组件（components）目录中定义了全国及重点城市信息的组件。



图 14.2 通过 VS Code 查看应用架构

## 14.3 后台数据获取方式

在后台数据获取的方式上，我们可以通过 jsonp 方式，从免费的公共 API 接口中取得全国城市的数据信息。使用 jsonp 方式的基本语法如下：

```
$http.jsonp({
```

```

url: ""
params: {}
callback: function(data) {}
error: function(err) {}
complete: function() {}
})

```

其中，url 表示请求地址；params 表示请求地址中参数对象；callback 表示回调函数，返回的请求数据在该函数中进行处理；error 表示错误返回；complete 表示请求结束返回。另外，上面关于 jsonp 方式的基本语法，只描述了主要常用的参数，还有一些不常用的参数没有列进去，读者可以自行查阅官方文档了解一下。

通过 jsonp 方式，从免费的公共 API 接口中跨域请求数据也很简单，使用浏览器就可以进行测试，如图 14.3 所示。



图 14.3 通过 jsonp 方式跨域请求数据

如图 14.3 中的标识所示，通过浏览器输入免费的公共 API 接口地址，就可以返回全国省份城市的主要数据信息。如果想获取比较重要或更详细的数据信息，一般都需要为这些公共 API 接口支付一定的服务费，读者可以根据项目实际需要进行选择。

## 14.4 功能模块组件设计

在前文中已经介绍了，全国城市信息查询系统应用主要包括全国省份信息和几个重点城市信息模块。

首先介绍一下全国省份信息模块，代码如下：

## 【代码 14-1】（详见源代码 vuespa 目录中的 src\components\home.vue 文件）

```
01 <template>
02   <div class="home">
03     <h1>{{ msg }}</h1>
04     <ul>
05       <li v-for="rs in results" :key="rs.id">
06         <div class="m-content inline-block">
07           <span class="span-left">id:&nbsp;&nbsp;{{ rs.id }}</span>
08           <span class="span-left">城市名称:&nbsp;&nbsp;{{ rs.name }}</span>
09           <span class="span-left">
10             >邮编(Zip Code):&nbsp;&nbsp;{{ rs.zipcode }}</span>
11           >
12         </div>
13       </li>
14     </ul>
15   </div>
16 </template>
17 <script>
18 export default {
19   name: "home",
20   data() {
21     return {
22       msg: "City Lists",
23       results: []
24     };
25   },
26   created: function() {
27     let url = "https://api.jisuapi.com/area/province?appkey=";
28     let myappkey = "";
29     let json_url = url + myappkey;
30     // 请求 api
31     this.$http
32       .jsonp(
33         json_url,
34         {},
35         {
36           headers: {},
37           emulateJSON: true
38         }
39       )
40       .then(
41         function(response) {
42           // 这里是处理正确的回调
43           //console.log(JSON.parse(response.bodyText).result);
44           this.results = JSON.parse(response.bodyText).result;
45         },
46         function(response) {
47           // 这里是处理错误的回调
48           console.log(response);
49         }
50       )
51     }
52   }
53 }
```

```

50      );
51  }
52 };
53 </script>

```

### 【代码说明】

- 第 01~16 行代码中，通过<template>元素定义了一个虚拟模板，内部包含一个用于全国省份信息的列表<ul><li>元素。
- 第 18~52 行代码中，定义并导出一个全国省份信息组件（Home），该组件实现了全国省份列表信息的初始化与显示。其中，第 31~50 行代码中通过调用\$http.jsonp()方法跨域请求，以获得全国省份的数据信息（JSON 格式）。

然后，再介绍一下北京市区县信息模块，代码如下：

### 【代码 14-2】（详见源代码 vuespa 目录中的 src\components\Beijing.vue 文件）

```

01 <template>
02   <div class="home">
03     <h1>{{ msg }}</h1>
04     <ul>
05       <li v-for="rs in results" :key="rs.id">
06         <div class="m-content inline-block">
07           <span class="span-left">id:{{ rs.id }}</span>
08           <span class="span-left">区县名称:{{ rs.name }}</span>
09           <span class="span-left">邮编(Zip Code):{{ rs.zipcode }}</span>
10         </div>
11       </li>
12     </ul>
13   </div>
14 </template>
15 <script>
16 export default {
17   name: "beijing",
18   data() {
19     return {
20       msg: "City Lists",
21       results: []
22     };
23   },
24   created: function() {
25     let url = "https://api.jisuapi.com/area/city?";
26     let myappkey = "8db15f4d5a5370da";
27     let paramsData = {
28       parentid: 1,
29       appkey: myappkey
30     };
31     let json_url = url + myappkey;
32     // 请求 api
33     this.$http

```

```

34     .jsonp(
35         json_url,
36         { params: paramsData },
37         {
38             headers: {},
39             emulateJSON: true
40         }
41     )
42     .then(
43         function(response) {
44             // 这里是处理正确的回调
45             //console.log(JSON.parse(response.bodyText).result);
46             this.results = JSON.parse(response.bodyText).result;
47         },
48         function(response) {
49             // 这里是处理错误的回调
50             console.log(response);
51         }
52     );
53 }
54 );
55 </script>

```

#### 【代码说明】

- 第 01~14 行代码中，通过<template>元素定义了一个虚拟模板，内部包含一个用于北京市区县信息的列表<ul><li>元素。
- 第 16~54 行代码中，定义并导出一个北京市区县信息组件（Beijing），该组件实现了北京市区县列表信息的初始化与显示。其中，第 34~52 行代码中通过调用\$http.jsonp()方法跨域请求，以获得北京市区县的数据信息（JSON 格式）。

## 14.5 功能模块路由设计

在前文中已经介绍了，功能模块之间的导航可以利用 vue-router 路由完成。下面，我们介绍一下本查询系统基于 vue-router 的路由设计，代码如下：

#### 【代码 14-3】（详见源代码 vuespa 目录中的 src\router\index.js 文件）

```

01 import Vue from 'vue'
02 import Router from 'vue-router'
03 import Home from '@/components/Home'
04 import Beijing from '@/components/Beijing'
05 import Shanghai from '@/components/Shanghai'
06 import Tianjin from '@/components/Tianjin'
07 import Chongqing from '@/components/Chongqing'

```

```
08 import About from '@/components/About'
09 Vue.use(Router)
10 export default new Router({
11   routes: [
12     {
13       path: '/',
14       name: 'Home',
15       component: Home
16     },
17     {
18       path: '/home',
19       name: 'Home',
20       component: Home
21     },
22     {
23       path: '/beijing',
24       name: 'Beijing',
25       component: Beijing
26     },
27     {
28       path: '/shanghai',
29       name: 'Shanghai',
30       component: Shanghai
31     },
32     {
33       path: '/tianjin',
34       name: 'Tianjin',
35       component: Tianjin
36     },
37     {
38       path: '/chongqing',
39       name: 'Chongqing',
40       component: Chongqing
41     },
42     {
43       path: '/about',
44       name: 'About',
45       component: About
46     }
47   ]
48 })
49 })
```

### 【代码说明】

- 第 01~08 行代码中，通过 import 命令导入必要的模块组件。
- 第 09 行代码中，通过 Vue.use()命令注入 Router 路由组件（关键步骤）。
- 第 10~47 行代码中，定义并导出一个路由对象（Router），该对象定义了一组基于功能模块的路由规则信息。

# 14.6 测试应用

我们测试一下全国城市信息查询系统应用（vuespa），页面初始效果如图 14.4 所示。

| City Lists |          |                      |
|------------|----------|----------------------|
| id: 1      | 城市名称: 北京 | 邮编(Zip Code): 100000 |
| id: 2      | 城市名称: 安徽 | 邮编(Zip Code):        |
| id: 3      | 城市名称: 福建 | 邮编(Zip Code):        |
| id: 4      | 城市名称: 甘肃 | 邮编(Zip Code):        |
| id: 5      | 城市名称: 广东 | 邮编(Zip Code):        |
| id: 6      | 城市名称: 广西 | 邮编(Zip Code):        |
| id: 7      | 城市名称: 贵州 | 邮编(Zip Code):        |
| id: 8      | 城市名称: 海南 | 邮编(Zip Code):        |

图 14.4 测试全国城市信息查询系统（一）

如图 14.4 中所示，我们尝试单击一下左侧导航链接中“北京”项，页面会路由跳转到北京市区县信息查询模块，页面效果如图 14.5 所示。

| City Lists |            |                      |
|------------|------------|----------------------|
| id: 499    | 区县名称: 东城区  | 邮编(Zip Code): 100000 |
| id: 500    | 区县名称: 西城区  | 邮编(Zip Code): 100000 |
| id: 501    | 区县名称: 海淀区  | 邮编(Zip Code): 100000 |
| id: 502    | 区县名称: 朝阳区  | 邮编(Zip Code): 100000 |
| id: 505    | 区县名称: 丰台区  | 邮编(Zip Code): 100000 |
| id: 506    | 区县名称: 石景山区 | 邮编(Zip Code): 100000 |
| id: 507    | 区县名称: 房山区  | 邮编(Zip Code): 102400 |
| id: 508    | 区县名称: 门头沟区 | 邮编(Zip Code): 102300 |

图 14.5 测试全国城市信息查询系统（二）

读者可以自行测试其他重点城市的列表，或者通过 Vue.js 框架提供的功能进一步完善该全国城市信息查询系统应用。