Web 开发讲义

南京大学 软件学院 刘钦 2021.3

第 1 章 前端开发基础

1.1 Web 前端

本节介绍 Web 前端的基础知识。主要是浏览器架构、HTML、CSS 和 JavaScript 的介绍。

1.1.1 浏览器架构

通常浏览器如图 1-1 所示，最上层是用户界面（User Interface）构件，主要负责与用户的交互，比如包括地址栏输入、刷新/向前/后退按钮、菜单中书签、浏览记录、偏好设置等功能。

下面是浏览器引擎（Browser Engine）。协调上层的 User Interface 和下层的渲染引擎（Rendering Engine）。Browser Engine 主要是实现浏览器的动作（初始化加载，刷新、向前、后退等）。

Rendering Engine 则是为给定的 URL 提供可视化展示，它解析 HTML、XML、JavaScript。不同的浏览器使用不同的 Rendering Engine。例如 IE 使用 Trident，Firefox 使用 Gecko，Safai、Chrome 和 Opera 使用 Webkit。

最底层是一些组件库。XML Parser 负责解析 XML。FireFox 用的是 Expat 库，Chrome 使用的是 libXML 库；Networking 负责基于 HTTP 和 FTP 协议处理网络请求，还提供文档的缓存功能以减少网络传输；JavaScript Interpreter 解释器负责解释执行页面的 js 代码，得到的结果传输给 Rendering Engine。Chrome 用的是 C++实现的 V8 引擎，FireFox 用的 SpiderMonkey；UI Backend 是 UI 的基础控件，隔离平台的实现，提供平台无关的接口给上层。Data Storage 则是负责对用户数据、书签、Cookie 和偏好设置等数据的持久化工作。

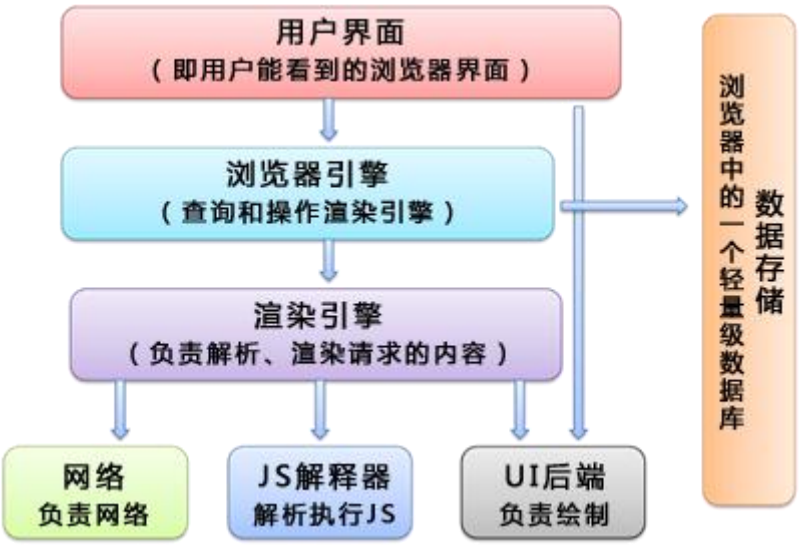

图 1-1 浏览器框架

1.1.2 用 HTML 生成内容

超文本标记语言（HyperText Markup Language，HTML）是一种用于创建网页的标准标记语言，用来描述网页的内容。HTML 使用标记标签（Tag）来描述网页，通常包含了 HTML 标签及文本内容。HTML 标签通常是成对出现的，比如 和 。标签内可以嵌套标签。下列代码在浏览器中显示如图 1-2 所示。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Title</title>
</head>

<body>
<h1>Heading 1</h1>
<p>This is a paragraph.</p>
<p>This is a paragraph.</p>
</body>



</html>
```

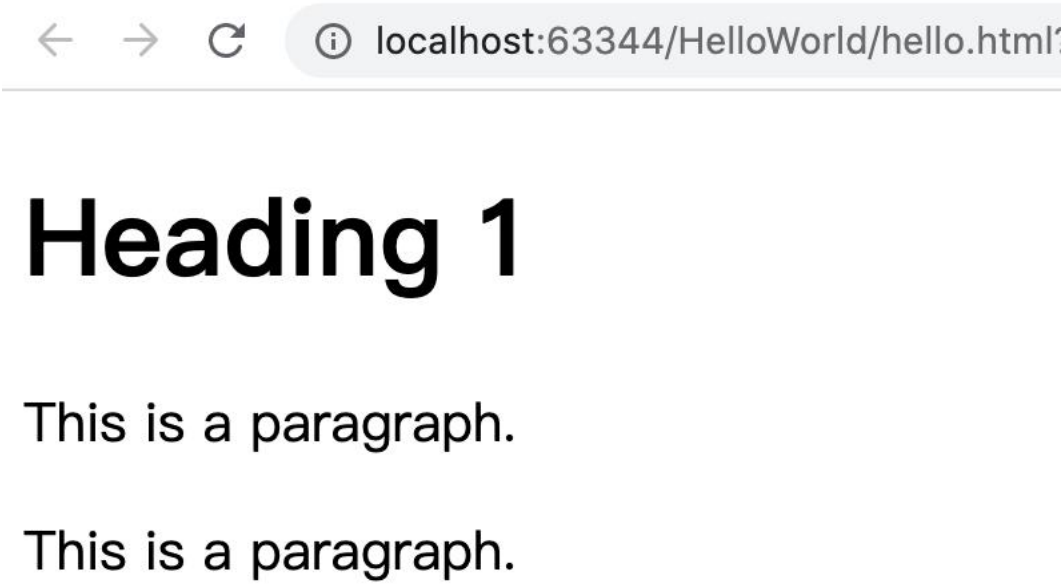

图 1-2 HTML 显示

HTML 运行在浏览器上，由浏览器来解析，解析生成 DOM(Document Object Model) Tree。DOM Tree 是由 HTML 文件解析 (parse) 生成代表内容的树状结构。比如下列 HTML 代码生成的 DOM Tree 如图 1-3 所示。

```
<doc>
<title>A few quotes</title>
<para>
  Franklin said that <quote>"A penny saved is a penny earned."</quote>
</para>
<para>
  FDR said <quote>"We have nothing to fear but <span>fear itself.</span>"</quote>
</para>
</doc>
```

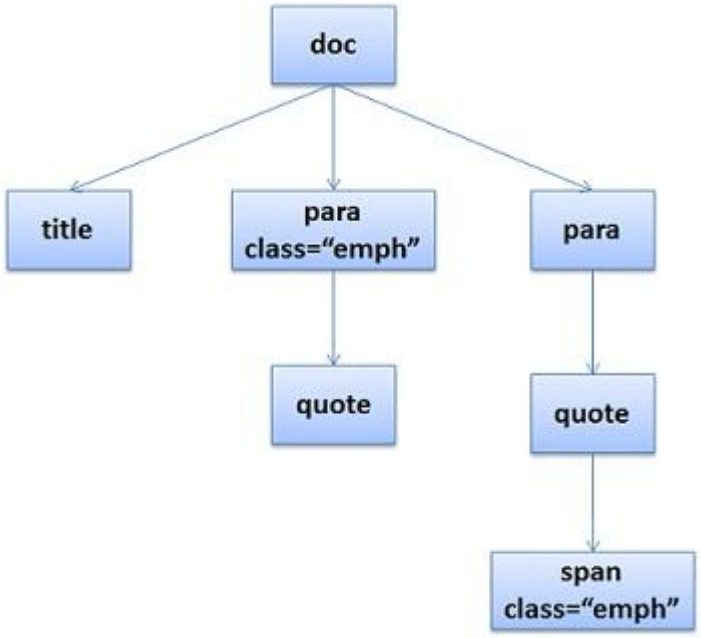

图 1-3 DOM Tree

1.1.3 用 CSS 生成样式

层叠样式表（Cascading Style Sheets，CSS） 用于控制网页的样式和布局。CSS3 是最新的 CSS 标准。CSS 不能单独使用，必须与 HTML 或 XML 一起协同工作，为 HTML 或 XML 起装饰作用。用于装饰 HTML 网页的 CSS 技术。其中 HTML 负责确定网页中有哪些内容，CSS 确定以何种外观（大小、粗细、颜色、对齐和位置）展现这些元素。CSS 可以用于设定页面布局、设定页面元素样式、设定适用于所有网页的全局样式。CSS 可以零散地直接添加在要应用样式的网页元素上，也可以集中化内置于网页、链接式引入网页以及导入式引入网页。

下列 CSS 代码中定义了 doc、title、para 的样式，以及 class 值为”emph”的元素的样式。

```
/* rule 1 */ doc { display: block; text-indent: 1em; }
/* rule 2 */ title { display: block; font-size: 3em; }
/* rule 3 */ para { display: block; }
/* rule 4 */ [class="emph"] { font-style: italic; }
```

同样，由 CSS 代码 parse 生成的样式规则的树状结构成为 CSS Tree(CSS Object Model Tree) ，如图 1-4 所示。

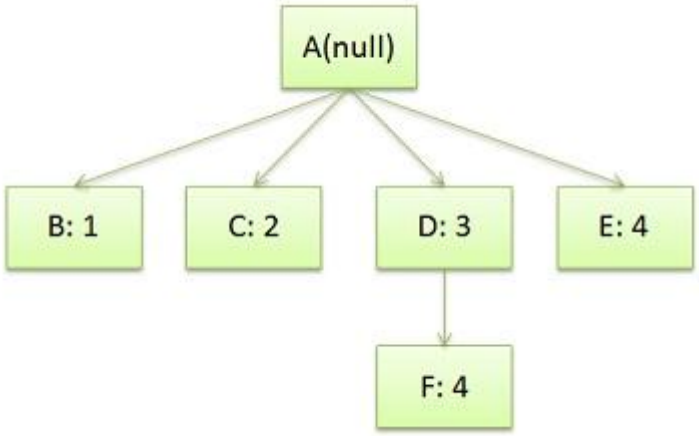

图 1-4 CSS Tree

图 1-5 所示的 Content Tree 则是同时包含内容和样式信息。

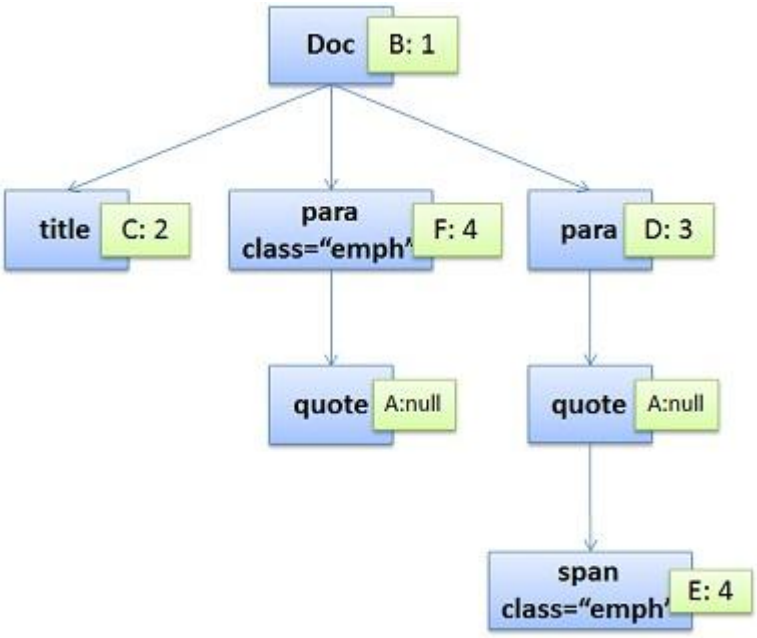

图 1-5 Content Tree

1.1.4 渲染流程

渲染树（Render Tree）是渲染流程的输出目标。它包含具有显示属性（颜色和大小）的长方形组成的树状结构。如图 1-6 所示，渲染的目标就是将最初写的页面内容 HTML、样式 CSS 渲染为最后的 Render Tree，然后调用系统图形 API 来显示。

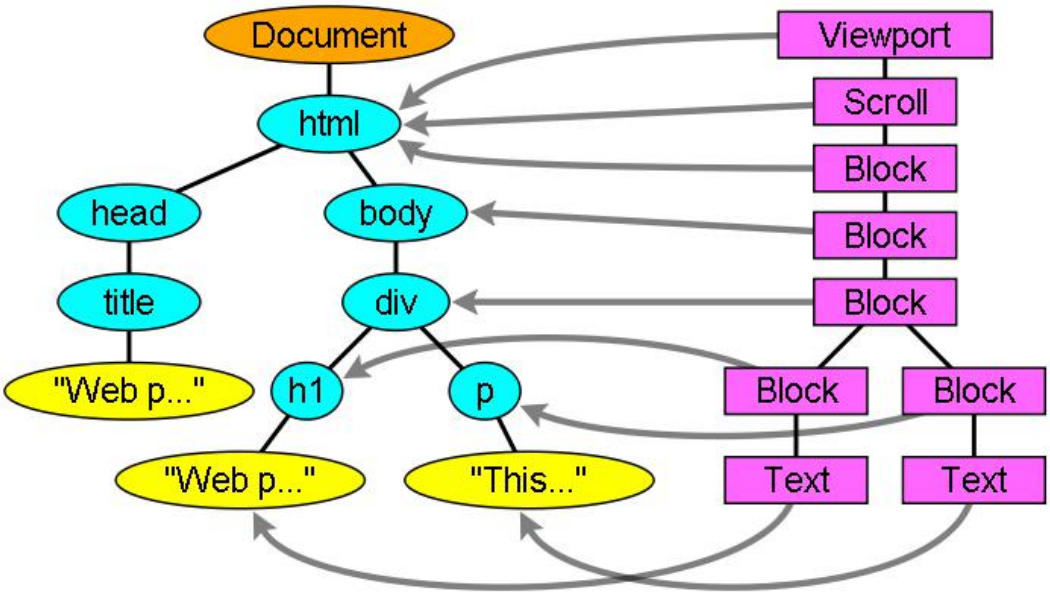

图 1-6 Render Tree

如图 1-7 所以，渲染过程如下：

(1) 解析（parse）：parse HTML/SVG/XHTML 文件，产生 DOM Tree；parse CSS 文件生成 CSS Rule Tree。

(2) 附着合成（construct）：DOM 树和 CSS 规则树连接在一起 construct 形成 Render Tree（渲染树）。

(3) 布局（reflow/layout）：计算出 Render Tree 每个节点的具体位置。

(4) 绘制（paint）：调用系统图形 API，通过显卡，将 Layout 后的节点内容分别呈现到屏幕上。

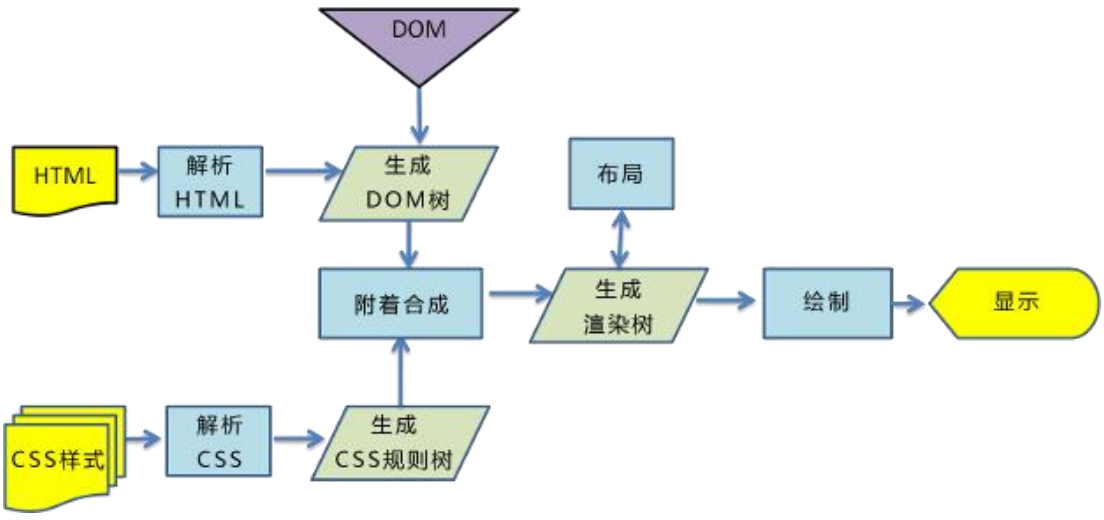

图 1-7 渲染过程

1.1.5 用 JavaScript 完成交互

用 HTML 和 CSS 已经可以显示页面的静态内容，但是网页还需要和用户和服务器进行交互。所以，可以用 JavaScript 语言完成与用户的交互。可以直接在 HTML 文件中加入 JavaScript 脚本，也可以使用单独的文件，再导入。JavaScript 语言可以通过 API 完成对 DOM Tree 和 CSS Tree 的操作，可以完成和用户的交互，完成和远端服务器的交互，也可以执行简单的前端和业务的逻辑。

1. script in HTML

下列是一个脚本在 HTML 文件中的例子。通过<script>标签，加入 JavaScript 脚本。脚本在文档中的位置非常重要。这可能会导致浏览器在屏幕上处理和呈现页面时出现严重的延迟。当浏览器遇到脚本标记时，DOM 构造暂停，直到脚本完成执行。JavaScript 可以查询和修改 DOM 和 CSSOM。JavaScript 会暂停执行直到 CSSOM 就绪。

案例中的脚本，修改了 DOM 的文字内容，修改了 CSSOM 的样式属性。并且创建了一个新的元素，插入了 DOM Tree 中。

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script>
      var span = document.getElementsByTagName('span')[0];
      span.textContent = 'interactive'; // change DOM text content
      span.style.display = 'inline';  // change CSSOM property
      // create a new element, style it, and append it to the DOM
      var loadTime = document.createElement('div');
      loadTime.textContent = 'You loaded this page on: ' + new Date();
      loadTime.style.color = 'blue';
      document.body.appendChild(loadTime);
    </script>
  </body>
</html>
```

2. script in js file

如果希望可读性更强，可以将 script 写在单独的文件中。不过我们必须在<script>标签中通过 src="app.js"属性来定义所要导入的 js 文件。

```
<!DOCTYPE html><html>
```

```
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script External</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js"></script>
  </body>
</html>
```

下面的 js 文件如上面的例子中的代码相同。

```
var span = document.getElementsByTagName('span')[0];
span.textContent = 'interactive'; // change DOM text content
span.style.display = 'inline';  // change CSSOM property
// create a new element, style it, and append it to the DOM
var loadTime = document.createElement('div');
loadTime.textContent = 'You loaded this page on: ' + new Date();
loadTime.style.color = 'blue';
document.body.appendChild(loadTime);
```

3. async

如果将 async 关键字添加到 script 标签中，会告诉浏览器在等待脚本可用时不要阻止 DOM 构造，将脚本的执行和 DOM 的构造异步化，这可以显著提高性能。

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script Async</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js" async></script>
  </body>
</html>
```

1.2 Vue.js 入门

本节介绍 Vue 的特性、安装步骤和简单的入门案例。

1.2.1 Vue.js 介绍

Vue.js 是一个用于创建用户界面的开源 JavaScript 框架，也是一个创建单页应用的 Web 应用框架。

Vue.js 具有如下的特性，可以随意组合需要用到的模块 vue + components + vue-router + vuex + vue-cli。

图 1-8 Vue.js 特性

1. 声明式渲染

Vue.js 的核心是一个允许采用简洁的模板语法来声明式地将数据渲染进 DOM 的系统。下列代码中界面 标签内的内容和数据 message 实现了双向绑定，当 message 改变的时候，界面也会自动随之而改变。

```
<div id="app">
  {{ message }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

2. 组件化应用构建

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。如图 1-9 所示，几乎任意类型的应用界面都可以抽象为一个组件树：

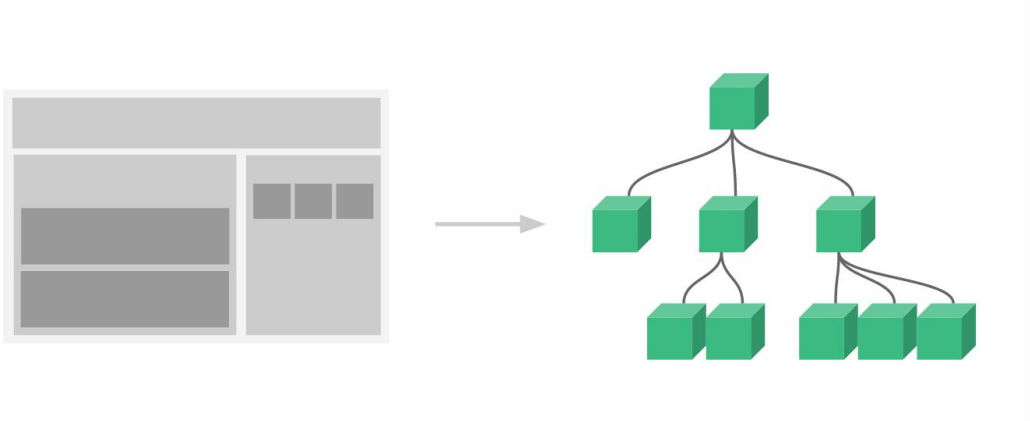

图 1-9 组件树

在 Vue 里，一个组件本质上是一个拥有预定义选项的一个 Vue 实例。在 Vue 中注册组件很简单：

```
// 定义名为 todo-item 的新组件
Vue.component('todo-item', {
  template: '<li>这是个待办项</li>'
})

var app = new Vue(...)
```

3. 前端路由

路由是将 URL 请求映射到代码的过程。当我们在 Web 应用中点击链接。URL 的改变将会给用户提供新的数据或者跳转新的网页。

传统的路由是服务器端路由，或者说后端路由。对于如下网页上的连接，

Hello!

操作的过程如下图 1-10 所示：

(1) 浏览器向当前 URL 路径下的子路径/hello 发送 GET 请求（ContentType 一般为 text/html）；

(2) 服务端解析请求并进行路由，向浏览器发送 HTML；

(3) 浏览器解析 DOM、CSS、JS 并进行渲染。

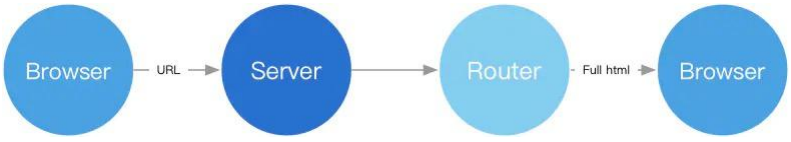

图 1-10 后端路由

后端路由每次都是一个完整网页的加载，网页的 HTML 一般是在后端服务器里通过模板引擎渲染好再交给前端的。但是对于同一个网站的页面中很多 DOM 结构（导航栏、脚注、

广告位）都是相似甚至是重复的。每次都反复请求是网络资源的浪费。

如果点击链接之后，不需要后端发送完整的页面，甚至不需要向后端请求。这就是单页面应用。（Single Page Appplication，SPA）。单页面的应用一般采用的就是前端路由。

SPA 中点击链接加载的完整过程是：

(1) 在首屏加载时，需要对应用需要的资源进行完整请求；
(2) 点击链接时检测请求事件，阻止浏览器发送 GET 请求；
(3) 前端路由代码改变地址栏 URL（利用 Hash、HTML5 History API 等）；
(4) 如果需要从服务端接收数据，发送 Ajax 请求获取需要的数据即可(JSON 等)；
(5) 路由代码对页面需要改变的 DOM 结构进行加载和渲染.

图 1-11 前端路由

这样虽然首次搜词加载的耗时较长，但是之后的加载速度大幅提高。浏览器从服务端拿到的初始 HTML。这里面空荡荡的可能只有一个 这个入口的 div 以及下面配套的一系列 js 文件。之后看到的页面其实是通过那些 js 渲染出来的。前端渲染把渲染的任务交给了浏览器，通过客户端的算力来解决页面的构建，这个很大程度上缓解了服务端的压力。而且配合前端路由，无缝的页面切换体验自然是对用户友好的。

而 vue-router 提供了如下功能：

1.前端路由：让页面中的部分内容可以无刷新的跳转，就像原生 APP 一样。

2.懒加载：结合异步组件以及在组件的 created 钩子上触发获取数据的 Ajax 请求可以最大化的降低加载时间，减少流量消耗。

3.重定向：可以实现某些需要根据特定逻辑改变页面原本路由的需求，比如说未登录状态下访问“个人信息”时应该重定向到登录页面。

2.美化 URL：通过 HTML5 History 模式优化 URL。

4. 大规模状态管理

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态。，

Vuex 其中包含以下几个部分：

(1) state，驱动应用的数据源；
(2) view，以声明方式将 state 映射到视图；
(3) actions，响应在 view 上的用户输入导致的状态变化。

Vuex 强调数据流的单向流动。界面 view 被点击之后，回调 action，然后改变状态 state，又重新渲染 view。但是，当我们的应用遇到多个组件共享状态时，单向数据流的简洁性很容易被破坏：

(1) 多个视图依赖于同一状态。
(2) 来自不同视图的行为需要变更同一状态。

对于问题一，传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。对于问题二，我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致无法维护的代码。

因此，我们为什么不把组件的共享状态抽取出来，以一个全局单例模式管理呢？在这种

模式下，我们的组件树构成了一个巨大的“视图”，不管在树的哪个位置，任何组件都能获取状态或者触发行为!

通过定义和隔离状态管理中的各种概念并通过强制规则维持视图和状态间的独立性，我们的代码将会变得更结构化且易维护。

这就是 Vuex 背后的基本思想。如图 1-12 所示，Vue 组件接收交互行为，调用 dispatch 方法触发 action 相关处理，若页面状态需要改变，则调用 commit 方法提交 mutation 修改 state，通过 getters 获取到 state 新值，重新渲染 Vue Components，界面随之更新。

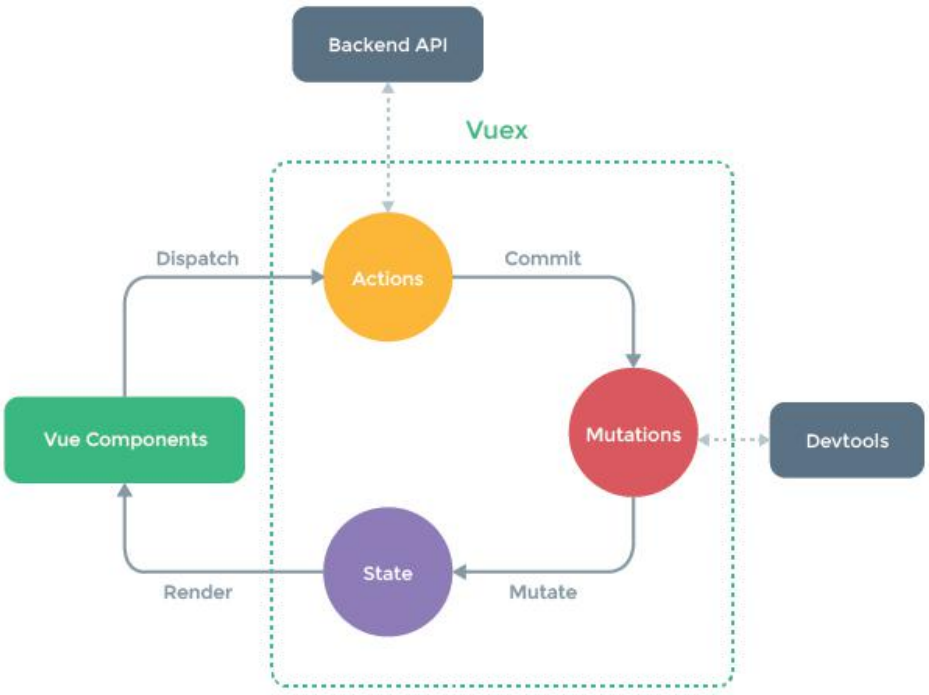

图 1-12 Vuex 原理

5. 强大的构建工具

Vue CLI 是一个基于 Vue.js 进行快速开发的完整系统，提供：

(1) 通过 @vue/cli 实现的交互式的项目脚手架。

(2) 通过 @vue/cli + @vue/cli-service-global 实现的零配置原型开发。

(3) 一个运行时依赖 (@vue/cli-service)，该依赖：

- 可升级；
- 基于 webpack 构建，并带有合理的默认配置；
- 可以通过项目内的配置文件进行配置；
- 可以通过插件进行扩展。

(4) 一个丰富的官方插件集合，集成了前端生态中最好的工具。

(5) 一套完全图形化的创建和管理 Vue.js 项目的用户界面。

Vue CLI 致力于将 Vue 生态中的工具基础标准化。它确保了各种构建工具能够基于智能的默认配置即可平稳衔接，这样你可以专注在撰写应用上，而不必花好几天去纠结配置的问题。

1.2.2 Vue.js 安装配置

1. 安装 Node.js 和 npm

如图 1-13，访问 https://nodejs.org/en/官方网址，选择对应操作系统的安装软件。

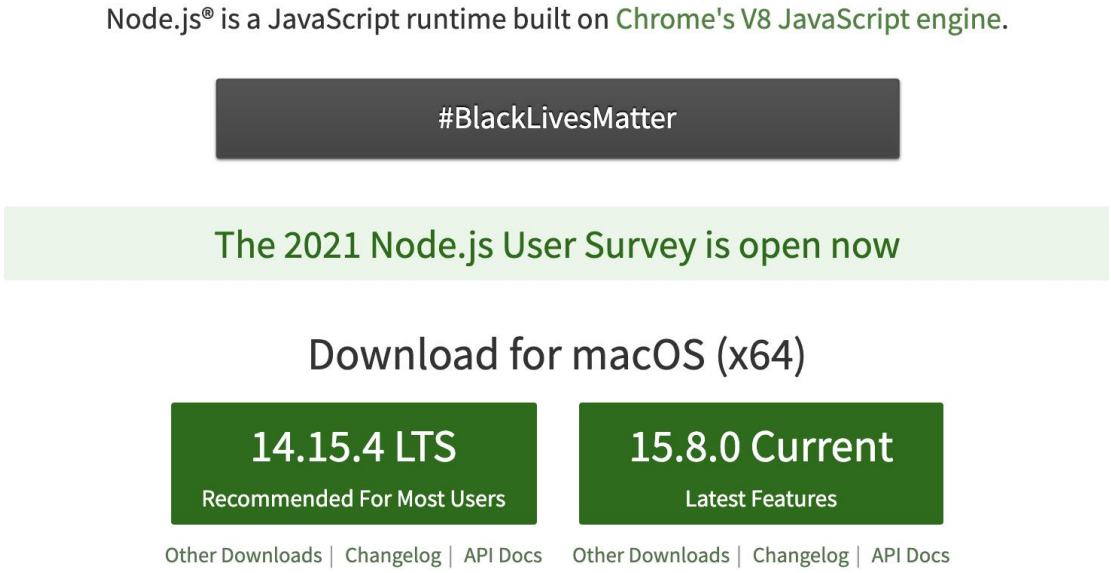

图 1-13 Node.js 官方网址安装页面

npm:为 Nodejs 下的包管理器。一般安装 Node.js 后，会自动安装 npm。如图 1-14，运行：node -v 查看 Node.js 版本。运行：npm -v 产看 npm 版本。

```
[QindeMacBook-Pro:~ qinliu$ node -v
v10.11.0
[QindeMacBook-Pro:~ qinliu$ npm -v
6.14.3
```

图 1-14 查看 Node.js 和 npm 版本

2. 安装 cnpm

cnpm 是中国 npm 镜像的客户端，用以访问 npm 相关的镜像文件在国内的备份站点，以提高国内用户镜像文件的下载速度。

运行：npm install -g cnpm --registry=http://registry.npm.taobao.org 安装 cnpm。

3. 安装 vue-cli

vue-cli 是一种全局脚手架用于帮助搭建所需的模板框架。

运行：npm install -g vue-cli 安装 vue-cli。

运行：vue -v 查看 vue 版本。

2. 创建第一个 Vue.js 项目

运行 vue init webpack helloworld。设置你的项目名称，其它可以默认回车。具体项目

的目录可以查看图 1-15。

5. 安装依赖

运行 npm install 或者 cnpm install。安装完成之后，会在我们的项目目录文件夹中多出一个 node_modules 文件夹，这里边就是我们项目需要的依赖包资源。

6. 运行项目

如图 1-15，进入你项目的目录，运行 npm run dev，启动本地服务器。最后，如图 1-16 通过浏览器访问 http://localhost:8080/#/ 网址。

图 1-15 运行项目

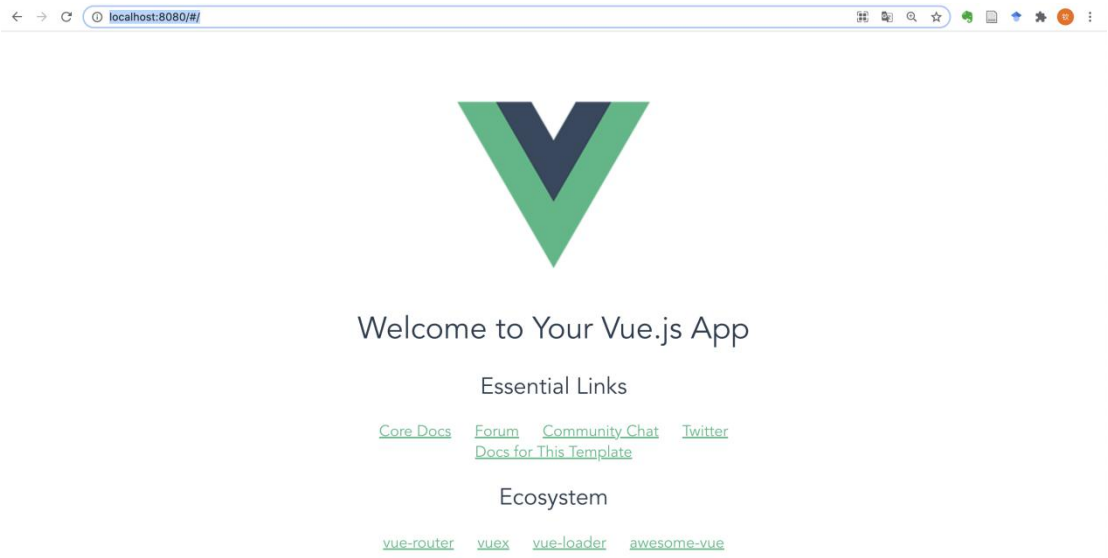

图 1-16 浏览器显示

1.2.3 Vue.js 基本使用

1. 计数器

下列代码是一个简单计数器的案例。点击按钮，显示计数的次数。counter 数据和 View 中的点击次数进行双向绑定。事件响应每次按按钮，counter 数据+1。View 中的点击次数也随之改变。 点击 3 次之后，显示效果如图 1-17 所示。

```
<!DOCTYPE html>

<head>
    <meta charset="utf-8">
    <title>Vue Example</title>
    <script src="https://cdn.staticfile.org/vue/2.2.2/vue.min.js"></script>
</head>
<body>
<div id="app">
    <button v-on:click="increment">plus 1</button>
    <p>This button is clicked {{ count }} times。</p>
</div>

<script>
    const app=new Vue({
        el:'#app',
        // state
        data:{
            count: 0
        },
        // actions
        methods: {
            increment:function () {
                this.count++
            }
        }
    })
</script>
</body>
</html>
```

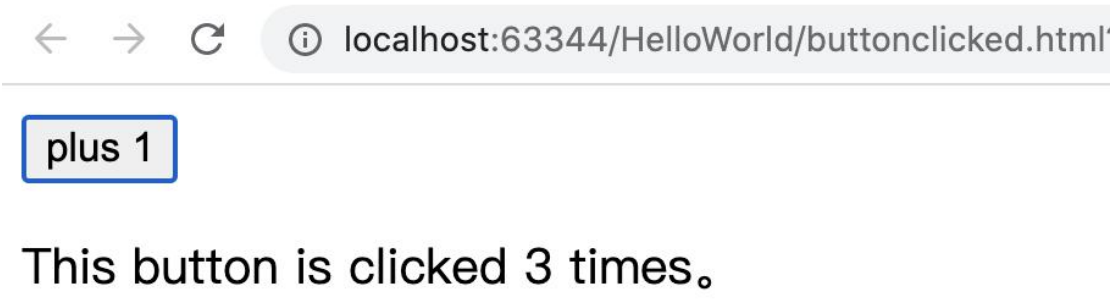

图 1-17 计数器案例显示效果

2. 网站列表

客户端 ViewModel 有时候会异步请求 Model 的数据。比如下列代码的 info 数据是向服务器 GET 请求得来的。当服务器回答后，再根据 response 数据设置 ViewModel，从而达到 View 的改变。response.data.sites 数据赋给了 info。具体显示效果，如图 1-18 所示。

```
<div id="app">
  <h1>网站列表</h1>
  <div
    v-for="site in info"
  >
    {{ site.name }}
  </div>
</div>
<script type = "text/javascript">
new Vue({
  el: '#app',
  data () {
    return {
      info: null
    }
  },
  mounted () {
    axios
      .get('https://www.runoob.com/try/ajax/json_demo.json')
      .then(response => (this.info = response.data.sites))
      .catch(function (error) { // 请求失败处理
        console.log(error);
      });
  }
})
</script>
```

这个代码如果是在本地运行，由于要访问 www.runoob.com 域名的接口，则会出现跨域问题。可以在 https://www.runoob.com/try/try.php?filename=vue2-ajax-axios3 页面查

看效果。

图 1-18 网站列表案例显示效果

3. Vuex 实现计数器

下面我们将以一个 Vuex 实现的计数器的实现介绍 Vue.js 的基本使用。如图 1-19 所示，项目的主要目录文件如下。

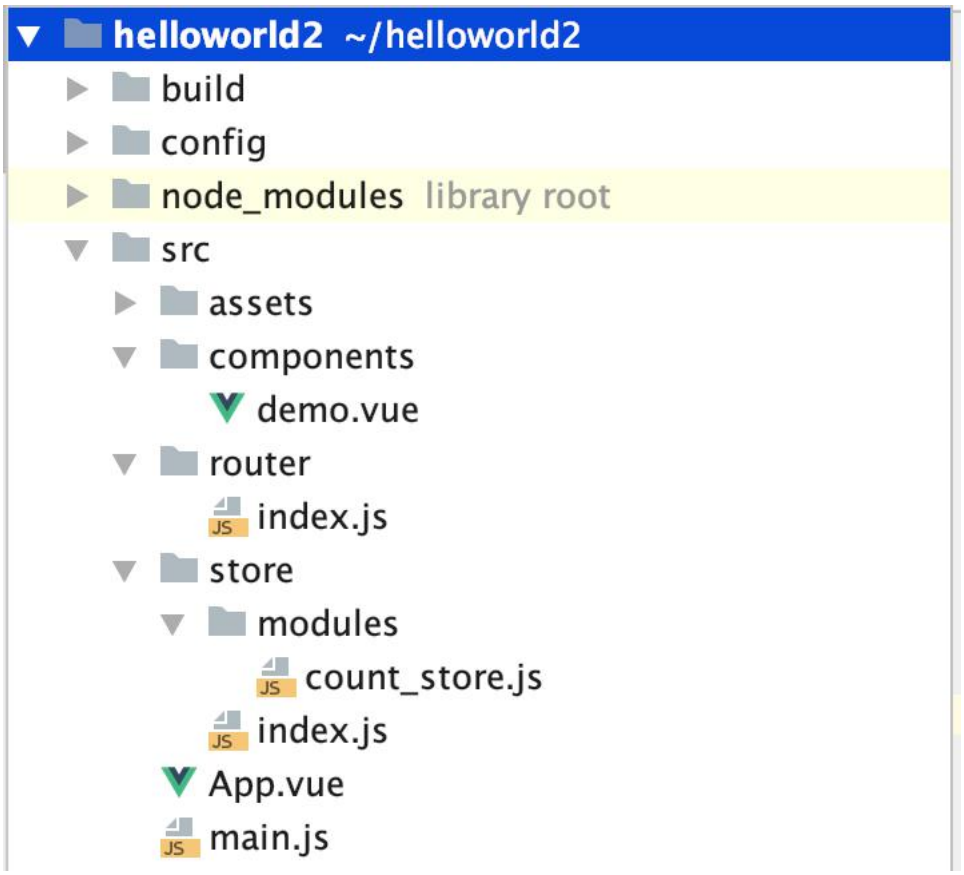

图 1-19 Vuex 实现的计数器项目主要目录文件

App.vue 是根页面级组件，main.js 是入口文件。在 src 目录下，router 目录下有 index.js，实现了路由。store 目录下有 index.js 文件和 modules 文件夹。把相关的状态（store）分离到 modules 目录下，再在 index.js 中引入。component 目录是界面组件。

App.vue 根页面级组件，引入了 components 目录的 demo 组件。

```
<template>
  <div id="app">
    <Demo></Demo>
  </div>
</template>

<script>
import Demo from './components/demo.vue'

export default {
  components: {
    Demo
  }
}
</script>
```

main.js 中引入了 router 和 store。

```
import Vue from 'vue'
import App from './App'
import router from './router'
import store from './store'

Vue.config.productionTip = false

new Vue({
  el: '#app',
  router,
  store,
  components: { App },
  template: '<App/>'
})
```

下面是 store 目录的 index.js 代码。当应用比较简单时，可以把状态都写在一个 store 对象 index.js 里, 但当应用变得非常复杂时, index.js 就有可能很臃肿。所以我们将 store 分割成模块 (module) 。这里定义了一个的 count 模块, 来自 modules 目录的 count_store.js。

```
import Vue from 'vue'
import Vuex from 'vuex'
import countStore from './modules/count_store.js'

Vue.use(Vuex)

export default new Vuex.Store({
  modules: {
    count: countStore
```

```
  }
})
```

下面是 count_store.js 代码。里面定义了保存状态 count。Commit mutation（提交更改）是更改 Vuex 的 store 中的状态的唯一方法。Vuex 中的 mutation 非常类似于事件：每个 mutation 都有一个字符串的事件类型（type）和 一个 回调函数 （handler）。这个回调函数就是我们实际进行状态更改的地方，并且它会接受 state 作为第一个参数。代码中的 increment 是计数加一，decrement 是计数减一。

```
export default{
  state: {
    count: 1
  },
  mutations: {
    increment (state) {
      state.count++
    },
    decrement (state) {
      state.count--
    }
  }
}
```

demo.vue 代码如下。定义一个 <p> 标签，MVVM 双向绑定了 {{ $store.state.count.count }} 值。另外有两个按钮，点击之后的回调分别执行 "$store.commit('increment')"和"$store.commit('decrement')"提交变更。

```
<template>
  <div>
    <p>{{ $store.state.count.count }}</p>
    <div>
      <button @click="$store.commit('increment')">+</button>
      <button @click="$store.commit('decrement')">-</button>
    </div>
  </div>
</template>

<script>
export default {
  name: 'demo'
}
</script>

<style scoped>
```

```
</style>
```

可以通过命令行执行 npm run dev 命令来运行项目。如果使用类似 WebStorm 的 IDE，成功编译并启动服务器的话，可以看到如图 1-20 的输出。如果失败，及时修改代码，系统会自动帮你运行命令来完成编译和启动服务器的工作。

图 1-20 在 WebStorm IDE 的 Terminal 中编译运行项目

如图 1-21 所示，访问 http://localhost:8080 网站，就可以看到项目的显示效果。点击+号 4 次，点击-号 1 次，数字显示 3。

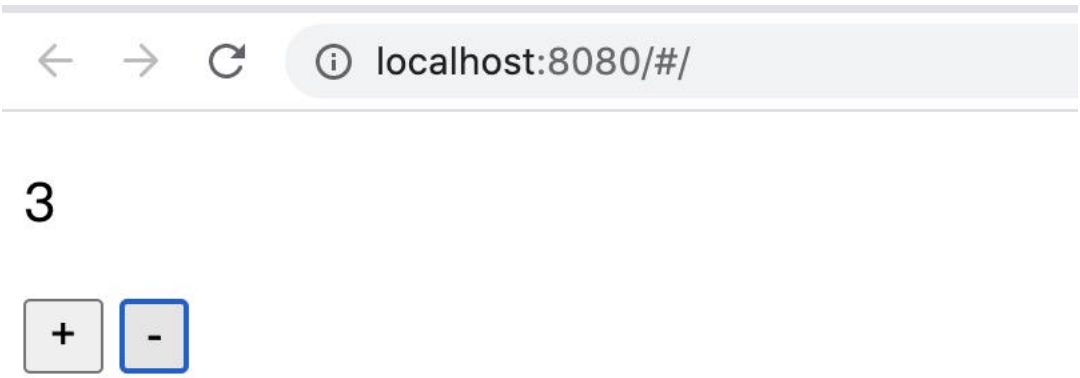

图 1-21 Vuex 实现的计数器运行效果

1.3 Node.js 入门

本节介绍 Node.js 的特性、Hello World 项目和简单的基于 Node.js 的服务器案例。

3.1.1 Node.js 介绍

Node.js 就是能够在服务端运行的 JavaScript 开放源代码、跨平台 JavaScript 环境。Node.js 采用 Google 开发的 V8 运行代码，使用事件驱动、非阻塞和异步输入输出模型等技术来提高性能，可优化应用程序的传输量和规模。这些技术通常用于资料密集的即时应用

程序。

Node.js 大部分基本模块都用 JavaScript 语言编写。在 Node.js 出现之前，JavaScript 通常作为客户端程序设计语言使用，用来完成前端的交互逻辑，以 JavaScript 写出的程序常在用户的浏览器上运行。Node.js 的出现之后，使得 JavaScript 应用于服务端编程成为可能。Node.js 的一系列内置模块，使得程序可以脱离 Apache HTTP Server 或 IIS，作为独立服务器运行。这使得 JavaScript 成为了全栈开放语言，大大提升了其在编程语言中的地位。

由于大量的第三方库的存在，在安装和使用时会给用户带来相当大的困扰。Node.js 附带了包管理器，npm。npm 是一个命令行工具，用于从 NPM Registry 中下载、安装 Node.js 程序，同时解决依赖问题，大大提供了开发的速度。

3.1.2 Node.js 安装配置

参考 1.2.2 节关于 Node.js 的安装。

1.3.3 Node.js 基本使用

1. Hello World

在任何文本编辑器上输入下列代码，然后保存为 hello.js。

```
'use strict';


console.log('Hello, world.');
```

第一行总是写上'use strict';是因为以严格模式运行 JavaScript 代码，避免各种潜在陷阱。

在控制台，进入 hello.js 所在目录，执行 node hello.js，控制台就会得到如下输出。

```
Hello, world.
```

2. Hello World Server

服务器端使用 required 指令来载入 CommonJS 模块。require()是同步加载，后面的代码必须等待这个命令执行完，才会执行。如果是浏览器端，由于浏览器通过网络的加载 ES6 模块无法保证速度，所以一般以 import 命令进行异步加载，或者更准确地说，ES6 模块有一个独立的静态解析阶段，依赖关系的分析是在那个阶段完成的，最底层的模块第一个执行。

Node.js 中自带 http 的模块，在代码中请求它并返回它的实例化对象赋给一个本地变量，通过这个本地变量可以调用 http 模块所提供的公共方法。然后，使用 http.createServer()方法来创建服务器，并使用 listen 方法绑定 8888 端口。方法通过 request，response 参数来接收和响应数据。

```
var http = require('http');


http.createServer(function (request, response) {


    // 发送 HTTP 头部
```

```
    // HTTP 状态值: 200 : OK
    // 内容类型: text/plain
    response.writeHead(200, {'Content-Type': 'text/plain'});

    // 发送响应数据 "Hello World"
    response.end('Hello World\n');
}).listen(8888);

// 终端打印如下信息
console.log('Server running at http://127.0.0.1:8888/');
```

如图 1-22 所示，运行：node server.js 启动服务器。通过浏览器访问 http://127.0.0.1:8888/ 网址，可以看到如图 1-23 所示效果。

```
QindeMacBook-Pro:nodehelloworld qinliu$ node server.js
Server running at http://127.0.0.1:8888/
```

图 1-22 启动 Node.js 服务器

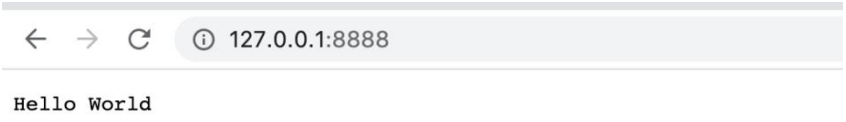

图 1-23 Node.js HelloWorld 案例显示效果

3. 网站列表服务器

Vue.js 章节中访问网站列表是通过第三方服务器得到的数据。如果我们想从自己的 Node.js 服务器得到数据，我们只需要向我们自己的服务地址发出 REST 请求。

修改之前创建第一个 Vue.js 项目的 HelloWorld 项目中 router 文件夹下的 index.js。

```
import Vue from 'vue'
import Router from 'vue-router'
import WebSites from '@/components/WebSites'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'OurWebSites',
```

```
      component: WebSites
    }
  ]
})
```

在 component 文件夹下创建我们新的 Vue component：WebSites.vue。代码如下：

```
<template>

  <div >
    <h1>网站列表</h1>
      <div v-for="site in info" v-bind:key="site">{{ site.name }}</div>
  </div>
</template>

<script>
import axios from 'axios'

export default {
  name: 'OurWebSites',
  data () {
    return {
      info: null
    }
  },
  mounted () {
    axios
      .get('http://localhost:8081/getWebSites')
      .then(response => (this.info = response.data.sites))
      .catch(function (error) { // 请求失败处理
        console.log(error)
      })
  }
}
</script>

<style scoped>

</style>
```

Express 是一种保持最低程度规模的灵活 Node.js Web 应用程序框架，为 Web 和移动应用程序提供一组强大的功能。Express 框架核心特性：可以设置中间件来响应 HTTP 请求；定义了路由表用于执行不同的 HTTP 请求动作；可以通过向模板传递参数来动态渲染 HTML 页面。下面案例，服务器根据 URL 映射到具体的处理函数，通过 req 得到请求，经过业务逻辑处理后，通过 res 设置回复。通过 JSON.stringfy()将数据转成 JSON 格式返回。

__dirname 获得当前目录名称。下面代码中用以构建数据文件的路径。

```
var express = require('express');
var app = express();
var fs = require("fs");
var cors = require('cors')

//解决跨域问题
app.use(cors({
    credentials: true,
    origin: '*',
}))

//映射 URL 和处理函数
app.get('/getWebSites', function (req, res) {
   fs.readFile( __dirname + "/" + "data.json", 'utf8', function (err, data) {
      data = JSON.parse( data );
      console.log( data);
      res.end( JSON.stringify(data));
   });
})

//设置监听端口，启动服务器
var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

本例子服务器是从本地一个 JSON 文件中读取的数据。JSON 文件内容如下。

```
{
    "sites":[
        { "name":"Sina", "info":[ "News", "Blog"] },
       { "name":"Baidu", "info":[ "Search", "Netdisk"] },
       { "name":"Taobao", "info":[ "Shopping"] }
    ]
}
```

最后，显示的效果如图 1-24 所示。

图 1-24 Node.js 网站列表案例显示效果

练习 1

1. 【多选题】浏览器架构有以下哪些部分组成?
   A. 用户界面
   B. 浏览器引擎
   C. 渲染引擎
   D. 数据存储、网络、JS 引擎、UI 后端等基本库
2. 【多选题】浏览器渲染包括以下几个步骤?
   A. 解析
   B. 附着合成
   C. 布局
   D. 绘制
3. 【单选题】下列说法不正确的是:
   A. HTML 使用标记标签（Tag）来描述网页的内容。
   B. CSS 确定以何种外观（大小、粗细、颜色、对齐和位置）展现这些元素。
   C. JavaScript 语言可以通过 API 完成对 DOM Tree 和 CSS Tree 的操作。
   D. JavaScript 脚本只能写在 HTML 里面才有效，不能作为单独的文件引用。
4. 【多选题】下列哪些是 Vue.js 的特点?
   A. 声明式渲染
   B. 组件化应用构建
   C. 前端路由
   D. 大规模状态管理
   E. 强大的构建工具
5. 【简答题】论述一下前端路由和后端路由的区别，和优缺点。
6. 【单选题】下列关于 Vuex 说法错误的是?
   A. Vuex 强调数据流的单向流动。
   B. Vuex 采用集中式存储管理应用的所有组件的状态,
   C. Vue 组件接收交互行为，调用 dispatch 方法触发 action 相关处理，若页面状态需要改变，则调用 commit 方法提交 mutation 修改 state，通过 getters 获取到 state 新值,

重新渲染 Vue Components，界面随之更新。

D. Vuex 既可以直接调用 mutation 方法，也可以得通过 commit 来提交 mutation。

7. 【简答题】简述 Vuex 项目各文件夹与文件的作用。
8. 【简答题】简述 Node.js Express 框架的功能。

第 2 章 Java Web 开发

2.1 HTTP 协议

本节我们介绍协议栈、数据包、HTTP 的顺序让大家了解 HTTP 协议的基础知识。

2.1.1 协议栈

网络中计算机之间互相通信，就必须制定一些规则，比如怎么搜索到目标计算机，怎么开始和结束通信，怎么保证可靠性。这些达成一致的规则称为协议。TCP/IP 是互联网相关的各类协议族的总称，计算机网络就是在 TCP/IP 协议族的基础上运作，通常抽象为 4 层的协议栈。从上到下，依次为：应用层、传输层、网络层和网络接口层。应用层为操作系统或网络应用程序提供访问网络服务的接口。HTTP 属于它内部的一个子集，提供了超文本数据的传输。传输层将上层数据分段并提供端到端的、可靠的或不可靠的传输以及端到端的差错控制和流量控制问题；主要用到的是 TCP 协议、UDP 协议；网络层对子网间的数据包进行路由选择，还可以实现拥塞控制，网络互连等功能；网络接口层则为网络层提供物理上的可靠的数据传输。

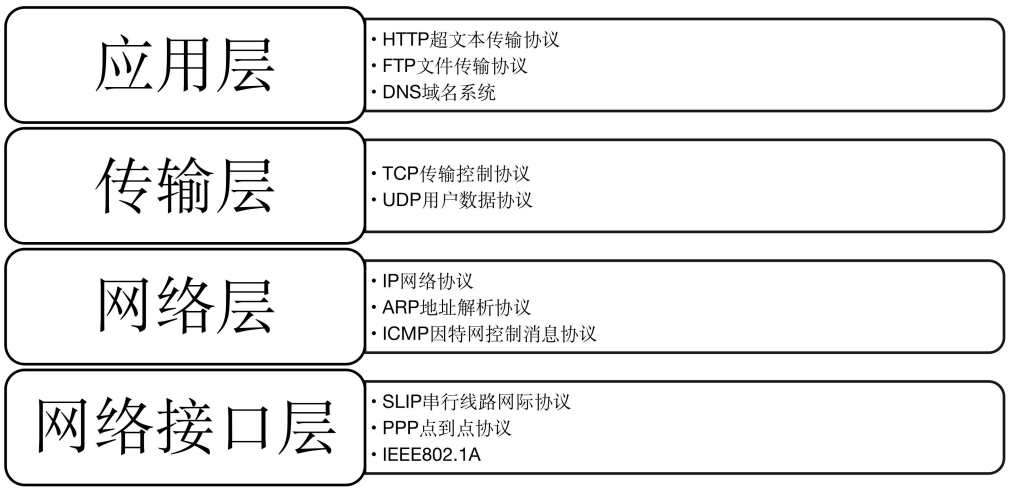

图 2-1 TCP/IP 协议栈

2.1.2 数据包

当应用层协议使用 TCP/IP 协议传输数据时，TCP/IP 协议簇可能会将应用层发送的数据分成多个包依次发送，而数据的接收方收到的数据可能是分段的或者拼接的，所以它需要对接收的数据进行拆分或者重组。应用层数据的单位称为消息。IP 和 UDP 协议会分片传输过大的数据包（Packet）避免物理设备的限制；TCP 协议会分段传输过大的数据段（Segment）保证传输的性能。在网络接口层中的数据链路子层，传输的单位是帧。我们在传输数据的时候，其实是按相应的单位来传输的。由于在各层都有相应的传输功能要实现，所以，发送时必须在各层的数据单位中增加头信息。如图 2-2 所示，各层会增加相应的头信息，再通过下一层传输过去。网络中的机器收到数据后，同样要根据头信息逐层解开各级数据，直到目标机器收到并解析出源机器最初要传输的应用层数据消息。

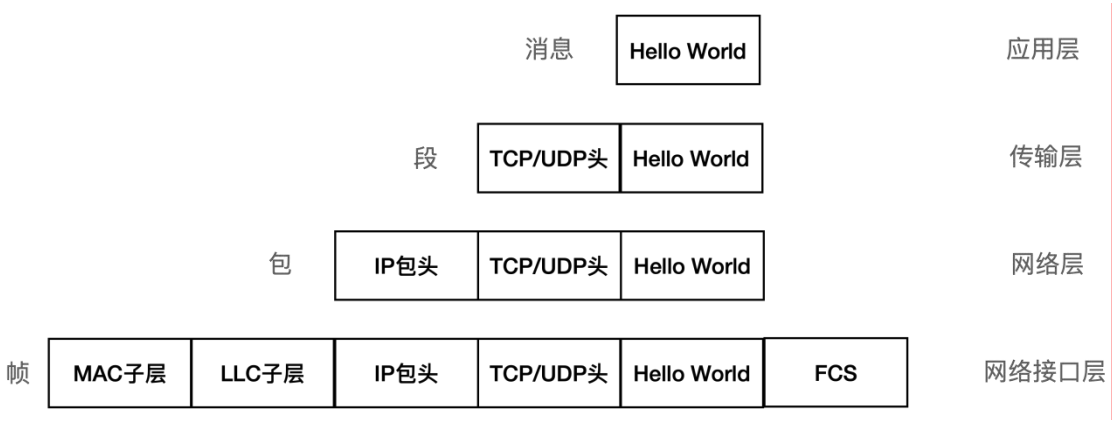

图 2-2 各层数据包

2.1.3 HTTP 协议

1. 客户端服务器模式

如图 2-3 所示，HTTP 是 TCP/IP 协议栈应用层的一个协议，是一个客户端服务器模型（C/S）。客户端（Client）发起请求（Request），服务器（Server）回送响应（Response）。HTTP 是一个无状态的协议。无状态是指客户机（Web 浏览器）和服务器之间的每个请求、如果在非 Keep-Alive 模式下，每次建立连接都是完全独立的，这意味着当一个客户端向服务器端发出请求，然后服务器返回响应，连接就被关闭了，在服务器端不保留连接的有关信息。当使用 Keep-Alive 模式时，Keep-Alive 功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接。

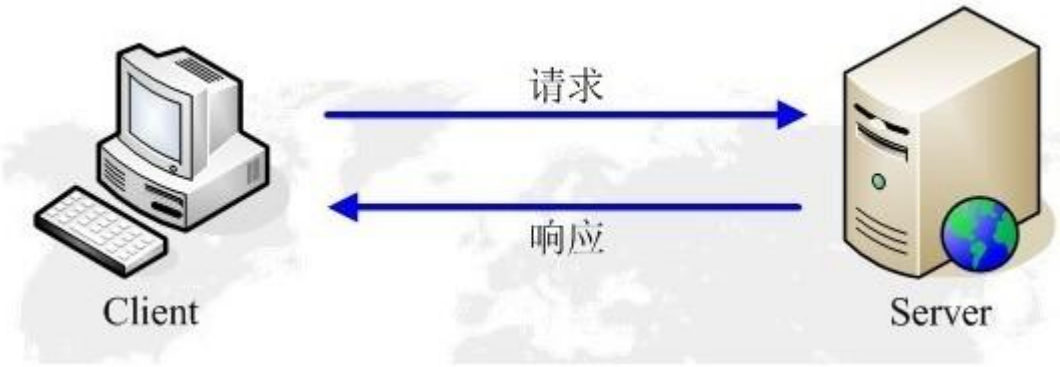

图 2-3 HTTP 客户端服务器模型

2. HTTP 工作过程

一次 HTTP 操作称为一个事务，其工作整个过程如下：

(1) 地址解析：如客户用浏览器请求某个网站页面，需要通过其 URL 来指定访问目标。例如，http://localhost.com:8080/index.htm，。我们从 URL 中分解结果如下：（协议名：http、主机名：localhost.com、端口：8080、对象路径：/index.htm）。真实访问时，还需要通过域名系统 DNS 解析域名 localhost.com，得到主机的 IP 地址。

(2) 封装 HTTP 请求数据包：把以上部分结合本机自己的信息，封装成一个 HTTP 请求数据包。

(3) 封装成 TCP 包，建立 TCP 连接（TCP 的三次握手）：在 HTTP 工作开始之前，客户机（Web 浏览器）首先要通过网络与服务器建立连接，该连接是通过 TCP 来完成的，该协议与 IP 协议共同构建 Internet，即著名的 TCP/IP 协议族。HTTP 是比 TCP 更高层次的应用层协议，根据规则，只有低层协议建立之后，才能进行更高层协议的连接，因此，首先要建立 TCP 连接，一般 TCP 连接的端口号是 80。这里是 8080 端口

(4) 客户机发送请求命令：建立连接后，客户机发送一个请求给服务器。例如：GET/sample/hello.jsp HTTP/1.1。

(5) 服务器响应：服务器接到请求后，给予相应的响应信息。例如： HTTP/1.1 200 OK。服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以 Content-Type 应答头信息所描述的格式发送用户所请求的实际数据。

(6) 服务器关闭 TCP 连接：一般情况下，一旦 Web 服务器向浏览器发送了请求数据，它就要关闭 TCP 连接。但如果浏览器或者服务器在其头信息加入了这行代码：Connection:Keep-Alive，TCP 连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

3. URL 格式请求

统一资源定位符（Uniform Resource Locator，URL）是因特网上标准的资源的地址，如同在网络上的门牌。URL 标准格式由三部分组成：协议类型、服务器地址(和端口号)、路径(Path)。HTTP URL 可以携带键值对形式的参数。参数以问号？开始并采用 name=value 的格式。如果存在多个 URL 参数，则参数之间用一个&符隔开。具体格式和示例如下：

(1) 格式： 协议类型://服务器地址[:端口号]路径

(2) 示例： http://blog.csdn.com/users?gender=male

4. Request

图 2-4 是 HTTP Request 请求的格式。主要有方法、路径、HTTP 版本、头信息和 Body。其中，常用的方法有 GET 和 POST。GET 方法用于获取资源，不会对服务器数据进行修改，所以不发送 Body。例如：

```
GET /users/1 HTTP/1.1
Host: api.github.com
```

POST 方法用于增加或修改资源 ，它会将发送给服务器的内容写在 Body 里面。例如：

```
POST /users HTTP/1.1
```

```
Host: api.github.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 13
name=rengwuxian&gender=male
```

图 2-4 是 HTTP Request 请求的格式

5. Response

图 2-5 是 HTTP Response 回答的格式。其中包括 HTTP 版本、状态码、状态消息、头信息、空行、Body。

图 2-5 HTTP Response 回答的格式

状态码是个三位数字，用于对响应结果做出类型化描述（如“获取成功”、“内容未找到”）常用状态码对应的含义如下：

(1) 1xx：临时性消息。如：100（继续发送）、101（正在切换协议）；
(2) 2xx：成功。最典型的是 200（OK）、201（创建成功）；
(3) 3xx：重定向。如：301（永久移动）302（暂时移动）、304（内容未改变）；
(4) 4xx：客户端错误。如：400（客户端请求错误）、401（认证失败）、403（被禁止）、404（找不到内容）；
(5) 5xx：服务器错误。如：500（服务器内部错误）。

2.2 Tomcat 和 Servlet 原理

本节介绍了 Tomcat 的职责，以及 Servlet 在 Tomcat 中如何工作的。

2.2.1 Tomcat

如图 2-6 所示, Apache Tomcat 是一个 Java 语言编写的运行在 JVM 之上的 Web 服务器（Java Servlet 容器），它和 HTTP 服务器一样，绑定 IP 地址并监听 TCP 端口，同时还包含以下职责：
(1) 管理 Servlet 程序的生命周期；
(2) 将 URL 映射到指定的 Servlet 进行处理；
(3) 与 Servlet 程序合作处理由 Web 浏览器发来的 HTTP 请求：根据 HTTP 请求生成 HttpServletResponse 对象并传递给 Servlet 进行处理，将 Servlet 中的 HttpServletResponse 对象生成的内容返回给浏览器。

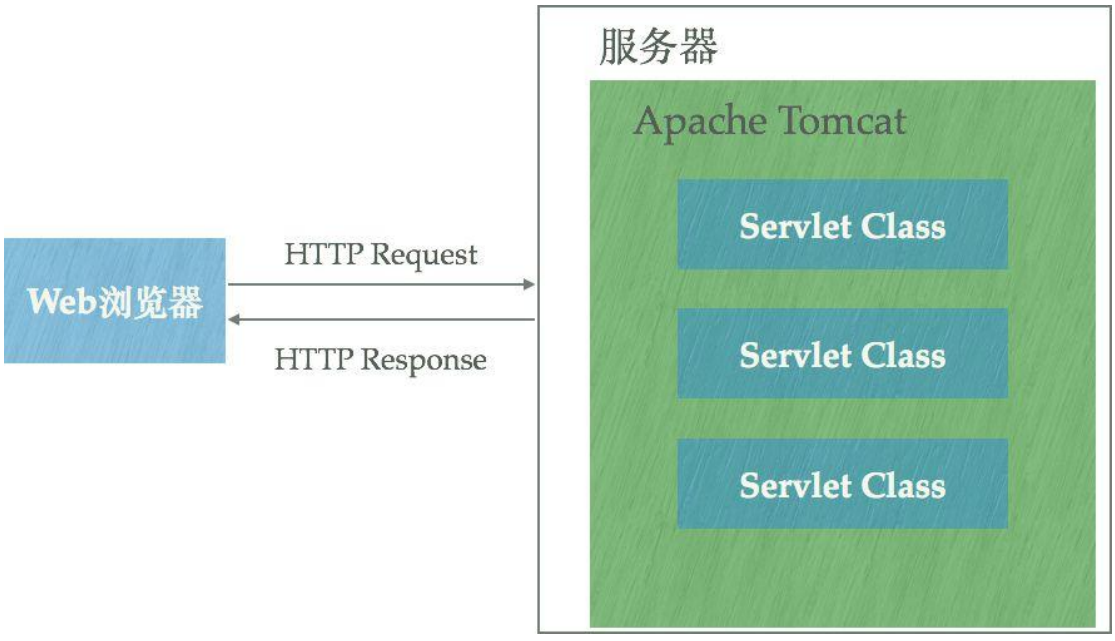

图 2-6 Apache Tomcat 工作原理

2.2.2 Servlet

Java Servlet 是是用 Java 编写的服务器端程序。其主要功能在于交互式地浏览和修改

数据，生成动态 Web 内容。Java Servlet 运行在 Web 服务器或应用服务器上，作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的中间层。

1. 生命周期

如图 2-7 所示，一个典型的 Servlet 生命周期方案如下：

(1) 到达服务器的 HTTP 请求被委派到 Servlet 容器；

(2) Servlet 容器在调用 service() 方法之前加载 Servlet 类；

(3) 然后 Servlet 容器处理由多个线程产生的多个请求，每个线程执行一个单一的 Servlet 对象的 service() 方法。

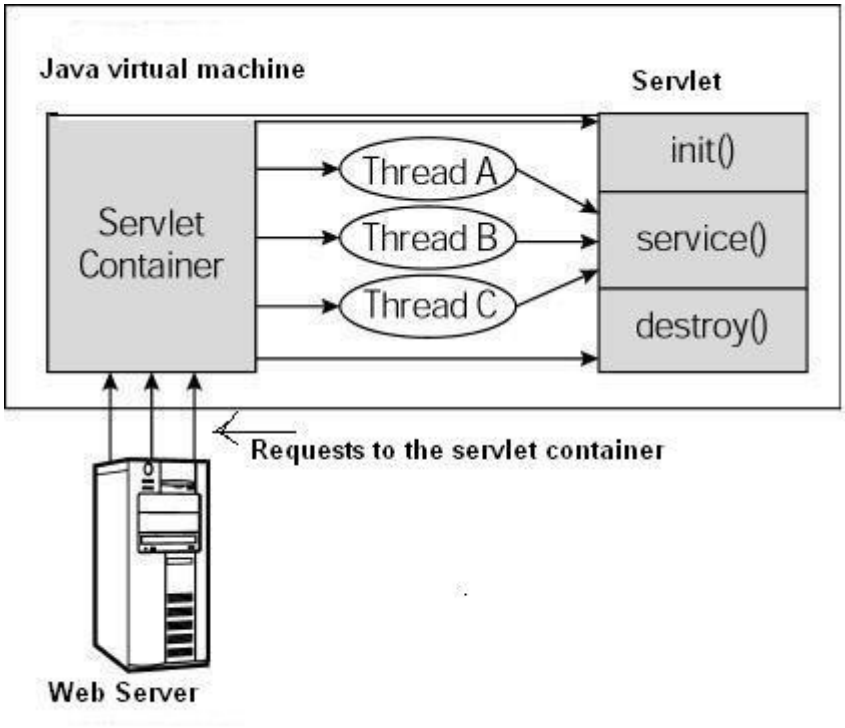

图 2-7 Servlet 工作原理

2. service() 方法

service() 方法是执行实际任务的主要方法。Servlet 容器（即 Web 服务器）调用 service() 方法来处理来自客户端（浏览器）的请求，并把格式化的响应写回给客户端。每次服务器接收到一个 Servlet 请求时，服务器会产生一个新的线程并调用服务。service() 方法检查 HTTP 请求类型（GET、POST、PUT、DELETE 等），并在适当的时候调用 doGet、doPost、doPut，doDelete 等方法。

下面是该方法的接口：

```
public void service(ServletRequest request,
                    ServletResponse response)
     throws ServletException, IOException{
}
```

service() 方法由容器调用，service 方法在适当的时候调用 doGet、doPost、doPut、doDelete 等方法。所以，不用对 service() 方法做任何动作，只需要根据来自客户端的请求类型来重写 doGet() 或 doPost() 即可。

doGet() 和 doPost() 方法是每次服务请求中最常用的方法:

(1) doGet() 方法：GET 请求来自于一个 URL 的正常请求，或者来自于一个未指定 METHOD 的 HTML 表单，它由 doGet() 方法处理。

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet 代码
}
```

(2) doPost() 方法: POST 请求来自于一个特别指定了 METHOD 为 POST 的 HTML 表单，它由 doPost() 方法处理。

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet 代码
}
```

3. Hello World, Servlet

下面代码是一个 Servlet 的 Hello World 程序：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloWorld extends HttpServlet {

  private String message;

  public void init() throws ServletException
  {
      // 执行必需的初始化
      message = "Hello World";
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
```

```
      // 设置响应内容类型
      response.setContentType("text/html");

      // 实际的逻辑是在这里
      PrintWriter out = response.getWriter();
      out.println("<h1>" + message + "</h1>");
  }

  public void destroy()
  {
      //
  }
}
```

首先通过如下命令编译 Servlet：

```
$ javac HelloWorld.java
```

接着，部署 Servlet。默认情况下，Servlet 应用程序位于路径 /webapps/ROOT 下，且类文件放在 /webapps/ROOT/WEB-INF/classes 中。如果您有一个完全合格的类名称 com.myorg.MyServlet，那么这个 Servlet 类必须位于 WEB-INF/classes/com/myorg/MyServlet.class 中。把 HelloWorld.class 复制到 /webapps/ROOT/WEB-INF/classes 中，并在位于 /webapps/ROOT/WEB-INF/ 的 web.xml 文件中创建以下条目：

```
<web-app>
   <servlet>
      <servlet-name>HelloWorld</servlet-name>
      <servlet-class>HelloWorld</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>HelloWorld</servlet-name>
      <url-pattern>/HelloWorld</url-pattern>
   </servlet-mapping>
</web-app>
```

最后，运行 Servlet。启动 Tomcat 服务器，最后在浏览器的地址栏中输入 http://localhost:8080/HelloWorld。浏览器显示结果如图 2-8 所示。

图 2-8 Servlet Hello World 项目运行效果

2.3 REST 和 JSON

本节介绍了 REST 风格和 JSON 数据格式。

2.1.1 REST 风格

1. REST 风格的提出

“The modern Web is one instance of a REST-style architecture.”

自从 Roy Fielding 博士在 2000 年他的博士论文中提出 REST（Representational State Transfer）风格的软件架构模式后，REST 就基本上迅速取代了复杂而笨重的 SOAP，成为 Web API 的标准了。图 2-9，给出了 REST 风格的软件架构模式满足的各种要求：可重复、分层、无状态、可靠、统一接口、可扩展、可重用等。

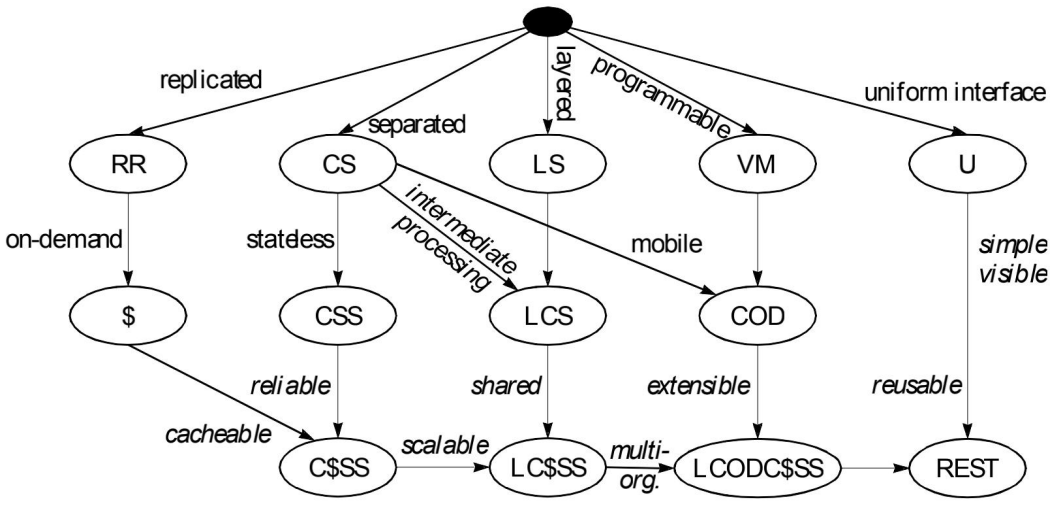

图 2-9 REST 风格的软件架构模式

2. Web API 与前后端分离

如果想要获取电商网站的某个商品信息，输入 http://localhost:3000/products/123，就可以看到 id 为 123 的商品页面，但这个结果是 HTML 页面，它同时混合包含了 Product 的数据和 Product 的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从 HTML 中解析出 Product 的数据。

如果一个 URL 返回的不是 HTML，而是机器能直接解析的数据，这个 URL 就可以看成是一个 Web API。比如，读取 http://localhost:3000/api/products/123，如果能直接返回 Product 的数据，那么机器就可以直接读取。

REST 风格就是一种设计 Web API 的风格。由于 JSON 能直接被 JavaScript 读取，所以，以 JSON 格式编写的 REST 风格的 Web API，即 REST API，具有简单、易读、易用的特点。

编写 REST API 有什么好处呢？如图 2-10 所示，由于 REST API 就是把 Web App 的功能全部封装了，所以通过 REST API 操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

此外，如果我们把前端页面看作是一种用于展示的客户端，那么 REST API 就是为客户端提供数据、操作数据的接口。这种设计可以获得极高的扩展性。例如，当用户需要在手机上购买商品时，只需要开发针对 iOS 和 Android 的两个客户端，通过客户端访问 REST API，就可以完成通过浏览器页面提供的功能，而后端代码基本无需改动。

当一个 Web 应用以 REST API 的形式对外提供功能时，整个应用的结构就扩展为：

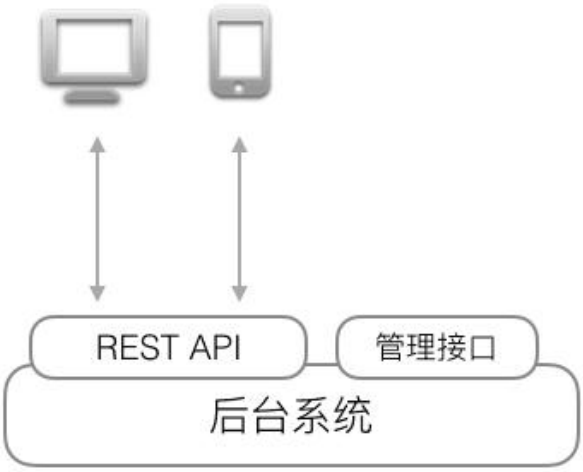

图 2-10 REST API 工作原理

3. REST API 规范

REST API 是遵从 REST 体系结构风格的 Web Service。REST 请求和普通的 HTTP 有所不同，通常会遵循下列规范：

(1) REST 请求仍然是标准的 HTTP 请求，但是，除了 GET 请求外，POST、PUT 等请求的 body 是 JSON 数据格式，请求的 Content-Type 为 application/json；

(2) REST 响应返回的结果是 JSON 数据格式，因此，响应的 Content-Type 也是 application/json。

(3) REST 规范定义了资源的通用访问格式，虽然它不是一个强制要求，但遵守该规范可以让人易于理解。

- 假如商品 Product 就是一种资源。获取所有 Product 的 URL 是：
  GET /api/products
- 获取某个指定的 Product，例如，id 为 123 的 Product。URL 是：
  GET /api/products/123
- 新建一个 Product 使用 POST 请求，JSON 数据包含在 body 中。URL 是：
  POST /api/products
- 更新一个 Product 使用 PUT 请求，例如，更新 id 为 123 的 Product。URL 是：
  PUT /api/products/123
- 删除一个 Product 使用 DELETE 请求，例如，删除 id 为 123 的 Product。URL 是：
  DELETE /api/products/123
- 资源还可以按层次组织。例如，获取某个 Product 的所有评论，使用的 URL 是：
  GET /api/products/123/reviews
- 当我们只需要获取部分数据时，可通过参数限制返回的结果集，例如，返回第 2 页评论，每页 10 项，按时间排序，使用的 URL 是：
  GET /api/products/123/reviews?page=2&size=10&sort=time

2.1.2 JSON

JSON（JavaScript Object Notation，JavaScript 对象表示法，读作/ˈdʒeɪsən/）是一种由道格拉斯·克罗克福特构想和设计、轻量级的数据交换语言，该语言以易于让人阅读的文字为基础，用来传输由属性值或者序列性的值组成的数据对象。尽管 JSON 是 JavaScript 的一个子集，但 JSON 是独立于语言的文本格式，并且采用了类似于 C 语言家族的一些习惯。

JSON 数据格式与语言无关。即便它源自 JavaScript，但当前很多编程语言都支持 JSON 格式数据的生成和解析。JSON 的官方 MIME 类型是 application/json，文件扩展名是 .json。

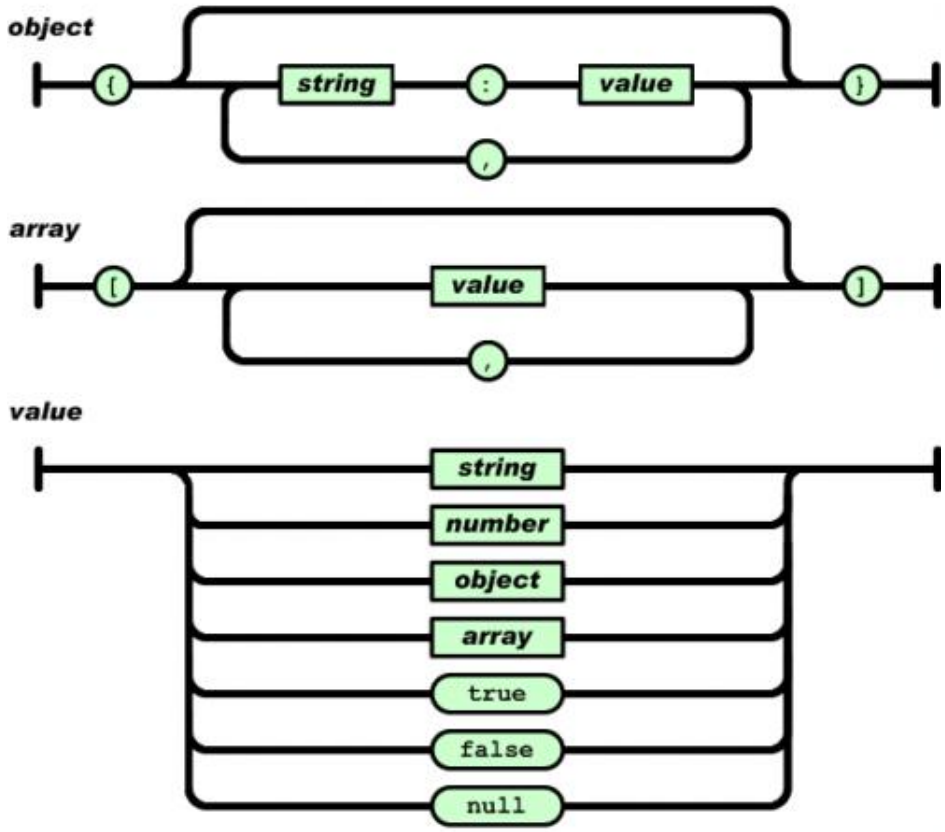

图 2-11 JSON 数据格式

如图 2-11 所示，JSON 本质是一个正则表达式，JSON 的 parser 是一个有限状态机。下面代码是一个 JSON 的示例。

```
{
    "firstName": "John",
    "lastName": "Smith",
    "sex": "male",
    "age": 25,
    "address":
    {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumber":
    [
        {
          "type": "home",
          "number": "212 555-1234"
        },
        {
```

```
        "type": "fax",
        "number": "646 555-4567"
      }
    ]
}
```

2.4 Spring Boot 入门

本节介绍 Spring Boot 的特色和 Hello World 项目。

2.2.1 Spring Boot 简介

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。其具有如下特色：

(1) 简化依赖，提供整合的依赖项，告别逐一添加依赖项的烦恼；
(2) 简化配置，提供约定俗成的默认配置，告别编写各种配置的繁琐；
(3) 简化部署，内置 Servlet 容器，开发时一键即运行。可打包为 jar 文件，部署时一行命令即启动；
(4) 简化监控，提供简单方便的运行监控方式。

2.2.2 Hello World

1. POM 文件

Spring Boot 的 Hello World 项目中 Maven 的 POM 文件如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project                              xmlns="http://maven.apache.org/POM/2.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/2.0.0
http://maven.apache.org/xsd/maven-2.0.0.xsd">
    <modelVersion>2.0.0</modelVersion>

    <groupId>com.dudu</groupId>
    <artifactId>chapter1</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>chapter1</name>
    <description>Demo project for Spring Boot</description>
```

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

2. 程序入口

Spring Boot 要求 main()方法所在的启动类必须放到根 package 下，命名不做要求。启动 Spring Boot 应用程序只需要一行代码加上一个注解@SpringBootApplication。

```
// Application.java
package hello;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

3. Controller

启动之后，对 URL “/” 路径的响应是通过 HelloController 类来实现的。需要添加 @RestController 和 @RequestMapping("/") 注解来说明。如果是其它路径，修改 RequestMapping 注解的参数为对应的路径即可。

```
// HelloController.java
package hello;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class HelloController {

    @RequestMapping("/")
    public String index() {
        return "Greetings from Spring Boot!";
    }

}
```

2. 启动和浏览器访问

启动项目有三种方式：

(1) 在 IDE 运行 Application 的 main 方法。

(2) 使用命令 mvn spring-boot:run”在命令行启动该应用。

(3) 运行“mvn package”进行打包时，会打包成一个可以直接运行的 JAR 文件，使用“java -jar”命令就可以直接运行。

启动项目之后，通过浏览器访问 URL，得到图 2-12 所示的效果。

图 2-12 Spring Boot 的 Hello World 项目运行效果

2.5 数据设计

本节介绍服务器如何通过关系型数据库进行数据持久。

2.5.1 数据持久化

为什么持久化？因为数据只有能够被记录（record）、存储（storage）、回忆（recall），才能成为知识，才能被我们人类所利用。在人类漫长的历史中出现了很多持久化的方法：实物记录（绳结、泥板、竹片、绢布、纸张）、电子文件、以及数据库管理系统。

2.5.2 关系型数据库

1. 关系

在关系型数据库中，我们用一张表表达一个关系。表（关系 Relation）是以行（值组 Tuple）和列（属性 Attribute）的形式组织起来的数据的集合。一个数据库包括一个或多个表（关系 Relation）。例如，可能有一个有关作者信息的名为 authors 的表（关系 Relation）。每列（属性 Attribute）都包含特定类型的信息，如作者的姓氏。每行（值组 Tuple）都包含有关特定作者的所有信息：姓、名、住址等等。

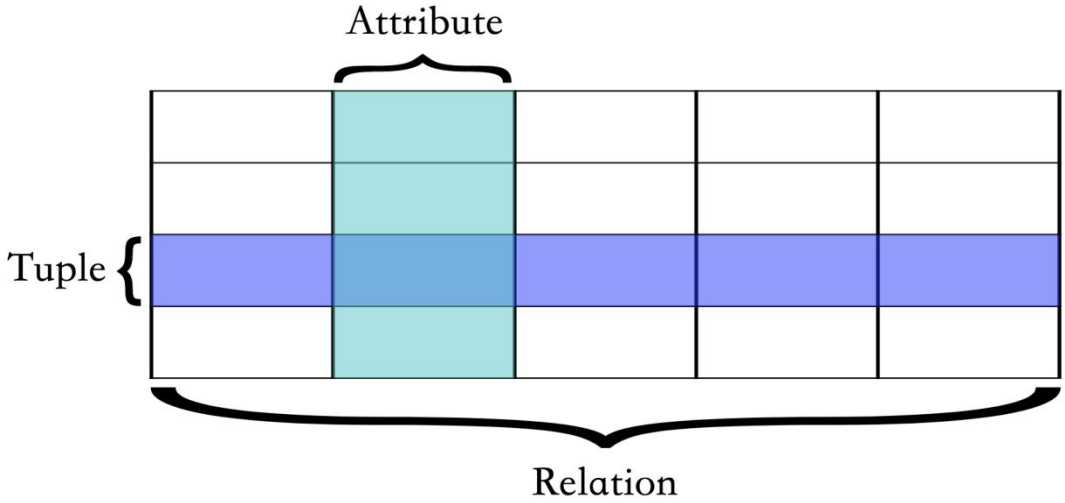

图 2-13 关系数据库中的表

2. 概念模型

概念模式是从用户的需求及其业务领域出发，进过分析和总结，提炼出来用以描述用户业务的概念类，以及概念类之间的关系。如图 2-14 所示，一个顾客可能买了多个商品。这时候，可以不关注概念的属性，主要关注的是类和类之间的关系。

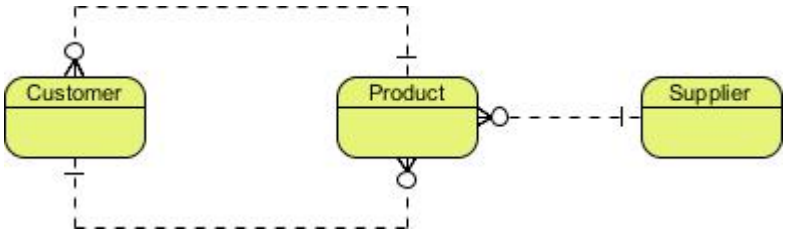

图 2-14 概念模型

3. 逻辑模型

逻辑模型就是将概念模型具体化，要实现概念的功能时候具体需要的实体和属性。如图 2-15 所示，新的订单实体和客户的属性被加入到逻辑模型中。概念模型只表明系统要实现什么，但怎样实现，用什么工具实现还没有讲。

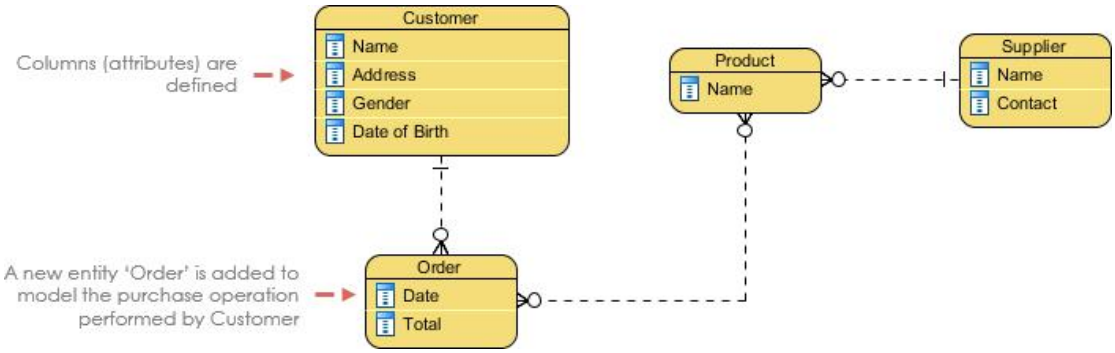

图 2-15 逻辑模型

4. 物理模型

物理模型则是对逻辑模型在真实数据库中实现的描述。物理模型包括数据表、主键、外键、字段、数据类型、长度、是否为空、默认值等。比如图 2-16 中所示，订单类被转换为 Purchase_Order 表。其中 ID 是主键，是 Integer 类型的。

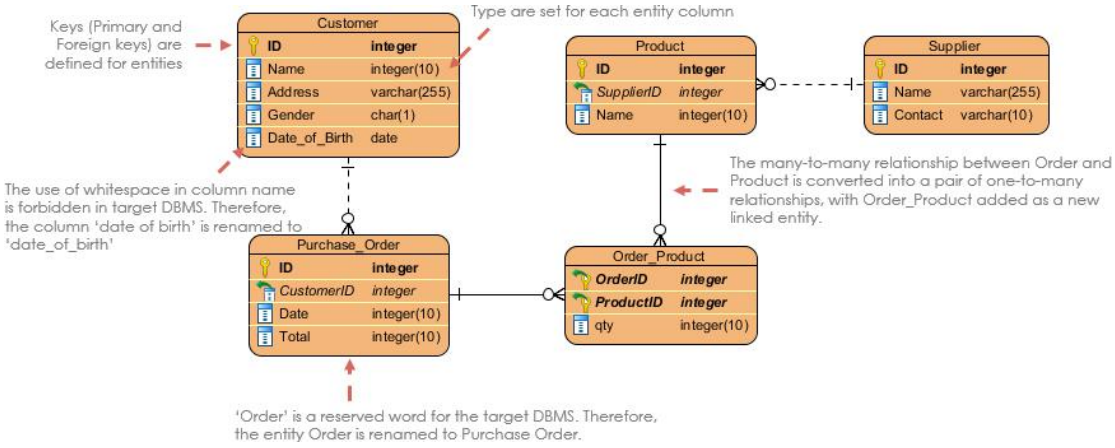

图 2-16 物理模型

5. 将类图转换为关系表

通常用 Java 面向对象设计类图实现了逻辑模型之后，我们主要要解决的问题就是向物理模型的转换。这样用面向对象编程范式设计的业务逻辑，就可以方便的和底层关系型数据对应起来。通常的转换过程如下:

(1) 通常一个类/对象映射为一张表
   a) 类的属性映射为表的列名
   b) 类的实例对象就是表的行
(2) 要为每个转换后的表建立主键
   a) 类/对象通过引用来唯一标识自己
   b) 关系/表通过主键类唯一标识自己
(3) 处理关联
   a) 类/对象通过链接实现关联
   b) 关系/表通过主键/外键对实现关联
      i. 1:1 关联：其中一个表的主键，作为另一个表的外键；两端等价
      ii. 1:N 关联：将 1 端表的主键，作为 N 端表的外键
      iii. M:N 关联：建立中间表，将 M 的主键和 N 的主键都作为中间表的外键
      iv. 在包含/聚合关联中，将整体的主键放在部分中作为外键
      v. 关联的最小基数为 0，意味着相应外键可以为 NULL

2.5.3 SQL 语句

结构化查询语言（Structured Query Language，SQL） 是用于管理关系数据库管理系统的一套标准语言。 SQL 的范围包括数据插入、查询、更新和删除，数据库模式创建和修改，以及数据访问控制。

一些重要的 SQL 命令如下所示：

(1) SELECT - 从数据库中提取数据
(2) UPDATE - 更新数据库中的数据
(3) DELETE - 从数据库中删除数据
(4) INSERT INTO - 向数据库中插入新数据
(5) CREATE DATABASE - 创建新数据库

(6) ALTER DATABASE - 修改数据库
(7) CREATE TABLE - 创建新表
(8) ALTER TABLE - 变更（改变）数据库表
(9) DROP TABLE - 删除表
(10) CREATE INDEX - 创建索引（搜索键）
(11) DROP INDEX - 删除索引

2.5.4 JDBC 原理

Java 数据库连接（Java Database Connecitivity，JDBC）是 Java 语言中用来规范客户端程序如何访问数据库管理系统的应用程序接口，提供了诸如查询和更新数据库中工属具的方法。

如 2-17 所示，我们客户端通过各种方式访问 Java 应用服务器，我们通过 ORM 将对业务对象的操作，转换为统一的 JDBC 方式，再由 JDBC 和数据库服务器进行通信。其中主要的步骤是：

(1) 登记并加载 JDBC 驱动器；
(2) 建立与 SQL 数据库的连接；
(3) 传送一个 SQL 操作；
(4) 获得数据结果。

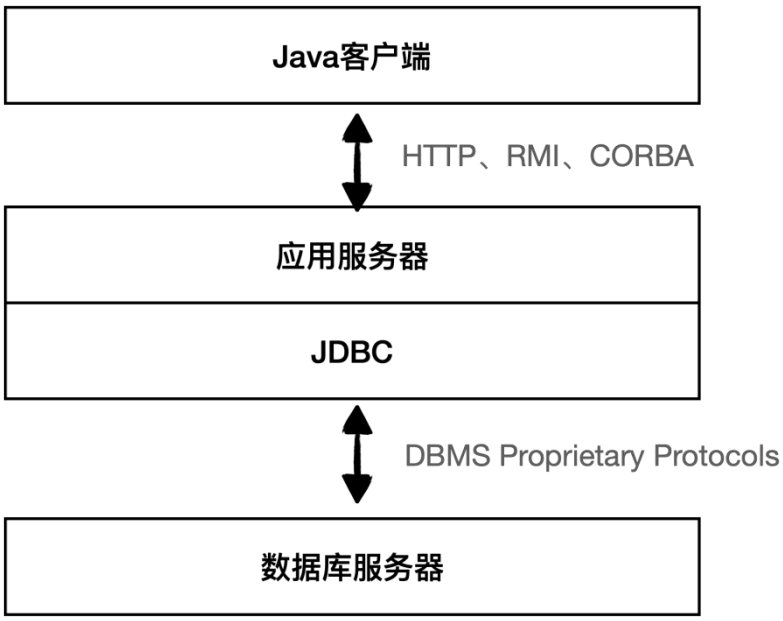

图 2-17 JDBC 原理

下面是使用 JDBC 完成一个 SQL 操作的示例代码：

```
//1.加载驱动程序
Class.forName("com.mysql.jdbc.Driver");
//2. 获得数据库连接
Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
//3.操作数据库，实现增删改查
Statement stmt = conn.createStatement();
```

```
ResultSet rs = stmt.executeQuery("SELECT user_name, age FROM imooc_goddess");
//如果有数据，rs.next()返回 true
while(rs.next()){
  System.out.println(rs.getString("user_name")+" 年龄："+rs.getInt("age"));
}
```

2.5.5 DAO

1. DAO

数据存取对象（Data Access objects，DAO）是指位于业务逻辑层和数据层之间实现以面向对象 API 为形式的对持久化数据进行 CRUD 操作的对象。通俗来讲，就是将数据库操作都封装起来。这样对逻辑对象的操作，就直接转换为对数据库中记录的操作。

DAO 的优势就在于：

(1) 提供了数据访问的 DAO 接口：隔离了数据访问代码和业务逻辑代码。业务逻辑代码直接调用 DAO 方法即可，完全感觉不到数据库表的存在，都是对逻辑对象的操作方法。

(2) 隔离了不同数据库实现：采用面向接口编程，如果底层数据库变化，如由 MySQL 变成 Oracle 只要增加 DAO 接口的新实现类即可，原有 MySQL 实现不用修改。这符合 "开-闭" 原则。该原则降低了代码的耦合性，提高了代码扩展性和系统的可移植性。

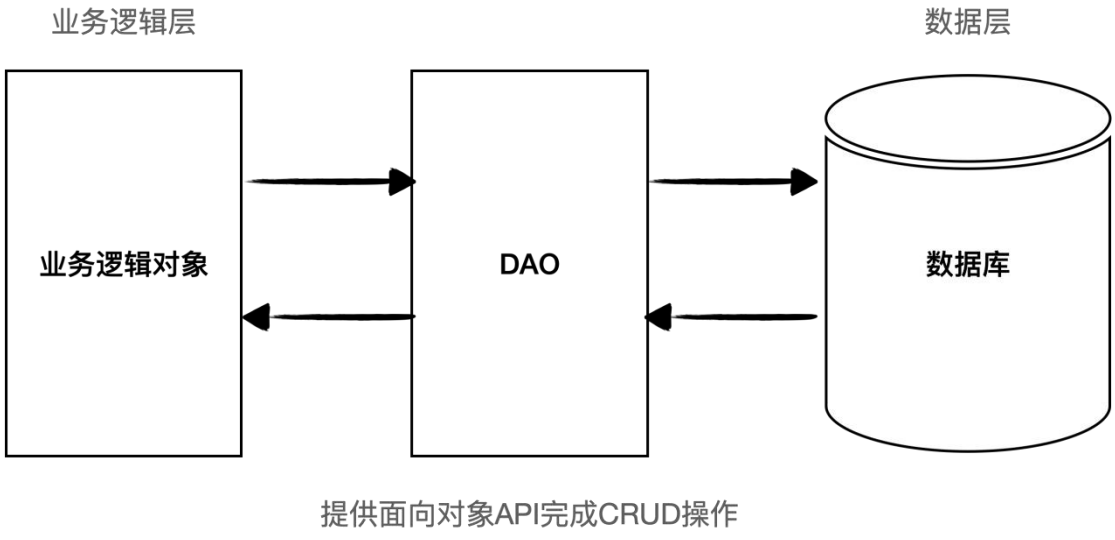

图 2-18 DAO 原理

2. 组成部分

实现 DAO 的时候需要实现以下几个部分：

(1) DAO 接口： 把对数据库的所有操作定义成抽象方法，可以提供多种实现。

(2) DAO 实现类： 针对不同数据库给出 DAO 接口定义方法的具体实现。

(3) 实体类：用于存放与传输对象数据。

(4) 数据库连接和关闭工具类： 避免了数据库连接和关闭代码的重复使用，方便修改。

下面是 DAO 的样例代码：

```
//DAO 接口：
public interface PetDao {
    /**
     * 查询所有宠物
     */
    List<Pet> findAllPets() throws Exception;
}
```

```
//DAO 实现类：
public class PetDaoImpl extends BaseDao implements PetDao {
    /**
     * 查询所有宠物
     */
    public List<Pet> findAllPets() throws Exception {
        Connection conn=BaseDao.getConnection();
        String sql="select * from pet";
        PreparedStatement stmt= conn.prepareStatement(sql);
        ResultSet rs=    stmt.executeQuery();
        List<Pet> petList=new ArrayList<Pet>();
        while(rs.next()) {
            Pet pet=new Pet(
                    rs.getInt("id"),
                    rs.getInt("owner_id"),
                    rs.getInt("store_id"),
                    rs.getString("name"),
                    rs.getString("type_name"),
                    rs.getInt("health"),
                    rs.getInt("love"),
                    rs.getDate("birthday")
                    );
                petList.add(pet);
        }
        BaseDao.closeAll(conn, stmt, rs);
        return petList;
    }
}
```

```
//宠物实体类(省略了 getter 和 setter 方法)
public class Pet {
    private Integer id;
    private Integer ownerId;    //主人 ID
```

```
    private Integer storeId;    //商店 ID
    private String name;    //姓名
    private String typeName;    //类型
    private int health;    //健康值
    private int love;    //爱心值
    private Date birthday;    //生日
}
```

```
//通过 JDBC 连接数据库
public class BaseDao {
    private static String driver="com.mysql.jdbc.Driver";
    private static String url="jdbc:mysql://127.0.0.1:3306/epet";
    private static String user="root";
    private static String password="root";
        static {
            try {
                Class.forName(driver);
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(url, user, password);
    }

    public static void closeAll(Connection conn,Statement stmt,ResultSet rs)
throws SQLException {
        if(rs!=null) {
            rs.close();
        }
        if(stmt!=null) {
            stmt.close();
        }
        if(conn!=null) {
            conn.close();
        }
    }

    public int executeSQL(String preparedSql, Object[] param) throws
ClassNotFoundException {
        Connection conn = null;
        PreparedStatement pstmt = null;
        /* 处理 SQL,执行 SQL */
```

```
        try {
            conn = getConnection(); // 得到数据库连接
            pstmt = conn.prepareStatement(preparedSql); // 得到 PreparedStatement
对象
            if (param != null) {
                for (int i = 0; i < param.length; i++) {
                    pstmt.setObject(i + 1, param[i]); // 为预编译 sql 设置参数
                }
            }
        ResultSet num = pstmt.executeQuery(); // 执行 SQL 语句
        } catch (SQLException e) {
            e.printStackTrace(); // 处理 SQLException 异常
        } finally {
            try {
                BaseDao.closeAll(conn, pstmt, null);
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return 0;
    }
}
```

2.5.6 MySql 数据库

MySQL 是现在最流行的关系型数据库管理系统。具体可以参考官方网站的说明，安装并使用。https://dev.mysql.com/doc/refman/8.0/en/installing.html。注意安装完毕之后要设置 root 用户的密码。

使用 Mysql 的时候，在 POM 文件中添加如下依赖：

```
        <!--mysql-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
```

在 Spring Boot 项目中使用前需要在 application.yml 文件修改 MySql 的用户和密码。

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mysql_test
    username: root
    password: root
    driver: com.mysql.jdbc.Driver
```

2.6 Mybatis

本节从 ORM 出发，介绍 Mybatis 框架的机制。

2.6.1 ORM

DAO 实现了业务逻辑和数据访问的隔离，而在 DAO 具体实现的过程中，我们往往会使用到关系型数据库。结合面向对象编程的思想，可以将关系型数据库中的关系用对象来表达，这其中就是要解决面向对象和关系型数据库之间存在的如表 2-1 的映射，也就是对象关系映射（Object Relation Mapping）。

表 2-1 关系型数据库和面向对象的映射

关系型数据库	面向对象
表（Table）	类（Class）
记录（Record）	对象（Object）
字段（Field）	属性

举例来说，下面是一行 SQL 语句：

```
SELECT id, first_name, last_name, phone, birth_date, sex FROM persons WHERE id = 10
```

如果我们直接在逻辑代码直接通过 SQL 操作数据库，那么具体代码写法如下：

```
res = db.execSql(sql);
name = res[0]["FIRST_NAME"];
```

如果改成 ORM 的写法，则具体代码写法如下：

```
p = Person.get(10);
name = p.first_name;
```

这样逻辑代码就直接操作对象，而不是具体的 SQL 数据操作了。具体的 SQL 操作由 ORM 框架自身来实现。开发者可以将注意力更加关注于逻辑代码的编写。ORM 具体框架如图 2-18 所示。最上层是接口层提供数据访问接口，中间是数据处理层，通过参数映射、SQL 解析、SQL 执行、结果映射完成对象和关系型数据库的映射，最底层是基础支撑层提供一个基础管理的服务。

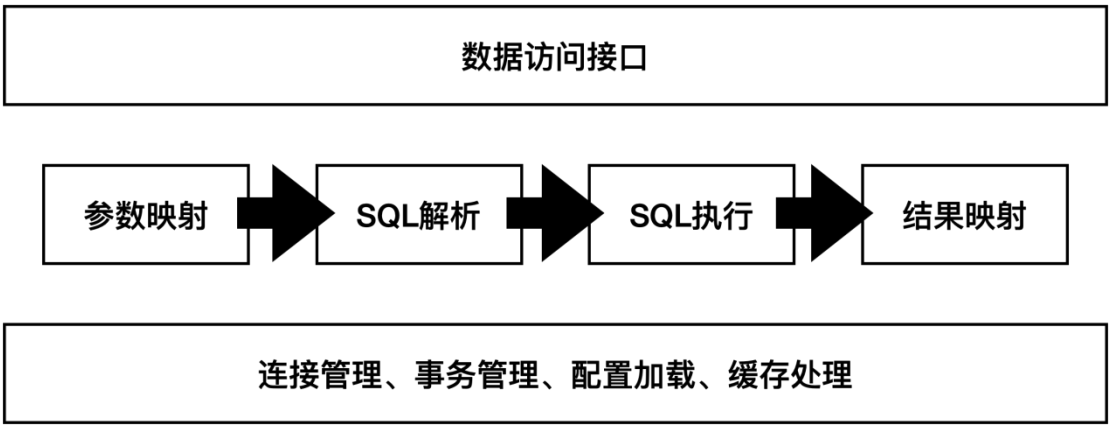

图 2-18 ORM 原理

ORM 是 Object 和 Relation 之间的映射，包括 Object->Relation 和 Relation->Object 两方面。Hibernate 和 MyBatis 是常见的两种 ORM 框架。Hibernate 是个完整的 ORM 框架，而 MyBatis 完成的是 Relation->Object，也就是其所说的 Data Mapper Framework。Hibernate 完全可以通过对象关系模型实现对数据库的操作，拥有完整的 JavaBean 对象与数据库的映射结构来自动生成 SQL。而 MyBatis 仅有基本的字段映射，对象数据以及对象实际关系仍然需要通过手写 SQL 来实现和管理。

2.6.2 Mybatis

MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以对配置和原生 Map 使用简单的 XML 或注解，将接口和持久化对象（Persistent Object，PO）映射成数据库中的记录。PO 通常使用 Java 的 普通的 Java 对象（Plain Old Java Objects，POJOs）来实现。

主流的 ORM 框架都实现了 Java 持久化 API（Java Persistence API，JPA）。MyBatis 没有实现 JPA，它和 ORM 框架的设计思路不完全一样。MyBatis 是拥抱 SQL，而 ORM 则更靠近面向对象，不建议写 SQL，则推荐用框架自带的类 SQL 代替。所以，更准确的说法是，MyBatis 是一种持久层框架，是一种 SQL 映射框架，而不是完整的 ORM 框架。

2.6.3 Mybatis 的安装和使用

要使用 MyBatis， 只需将 mybatis-x.x.x.jar 文件置于类路径（classpath）中即可。如果使用 Maven 来构建项目，则需将下面的依赖代码置于 pom.xml 文件中：

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>
```

Springboot 中可以使用如下依赖。

```
      <!--spring boot mybatis-->
```

```
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>2.1.4</version>
        </dependency>
```

下面是 Mybatis 的示例代码。具体的 MyBatis 相关的文件包括：

(1) Mapper 接口：业务逻辑使用的 API。
(2) PO 对象：接口传递的持久化数据对象。
(3) Mapper 实现的 XML 文件：用 SQL 语句实现 Mapper 接口。
(4) 项目 POM 文件：包括 Mybatis 和 MySQL 相关类库的依赖。
(5) 项目 application.yml 文件：其中包括 JDBC 和 Mybatis 的配置。

```
@Mapper
@Repository
public interface AdminMapper {

    int addManager(User user);

    List<User> getAllManagers();
}
```

```
public class User {  //省略 getter、setter
    private Integer id;
    private String email;
    private String password;
    private String userName;
    private String phoneNumber;
    private double credit;
    private UserType userType;
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-1-mapper.dtd">
<mapper namespace="com.example.hotel.data.admin.AdminMapper">
    <insert id="addManager" parameterType="com.example.hotel.po.User"
useGeneratedKeys="true" keyProperty="id">
        insert into User(email,password,usertype)
        values(#{email},#{password},#{userType})
    </insert>
    <select id="getAllManagers" resultMap="User">
        select * from User where usertype='HotelManager'
    </select>
```

```
    <resultMap id="User" type="com.example.hotel.po.User">
        <id column="id" property="id"></id>
        <result column="email" property="email"></result>
        <result column="password" property="password"></result>
        <result column="username" property="userNzame"></result>
        <result column="phonenumber" property="phoneNumber"></result>
        <result column="credit" property="credit"></result>
        <result column="usertype" property="userType"></result>
    </resultMap>
</mapper>
```

```
<!--mybatis-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.0</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

```
spring:
  datasource:
     url:
jdbc:mysql://localhost:3306/Hotel?serverTimezone=CTT&characterEncoding=UTF-8
     username: root
     password: ******
     driver-class-name: com.mysql.cj.jdbc.Driver
     max-active: 200
     max-idle: 20
     min-idle: 10
  thymeleaf:
     cache: false
  jackson:
    time-zone: GMT+8

mybatis:
  mapper-locations: classpath:dataImpl/*/*Mapper.xml
```

2.7 Spring Boot+Mybatis 案例解析

本节介绍了结合了 Mybatis 的 Spring Boot 的实战案例的代码级解析。

2.7.1 项目结构

图 2-18 是 Spring Boot+Mybatis 项目的结构。其中主要有以下包组成：

(1) contoller:负责处理 REST API 请求
(2) bl：业务逻辑层接口
(3) blImpl:业务逻辑层实现
(4) data：数据层接口
(5) po：持久化对象，用于数据层和逻辑层之间数据传递
(6) vo：值对象，用于逻辑层和展示层（前端）数据传递
(7) dataImpl：数据层实现，Mybatis 是 XML 文件

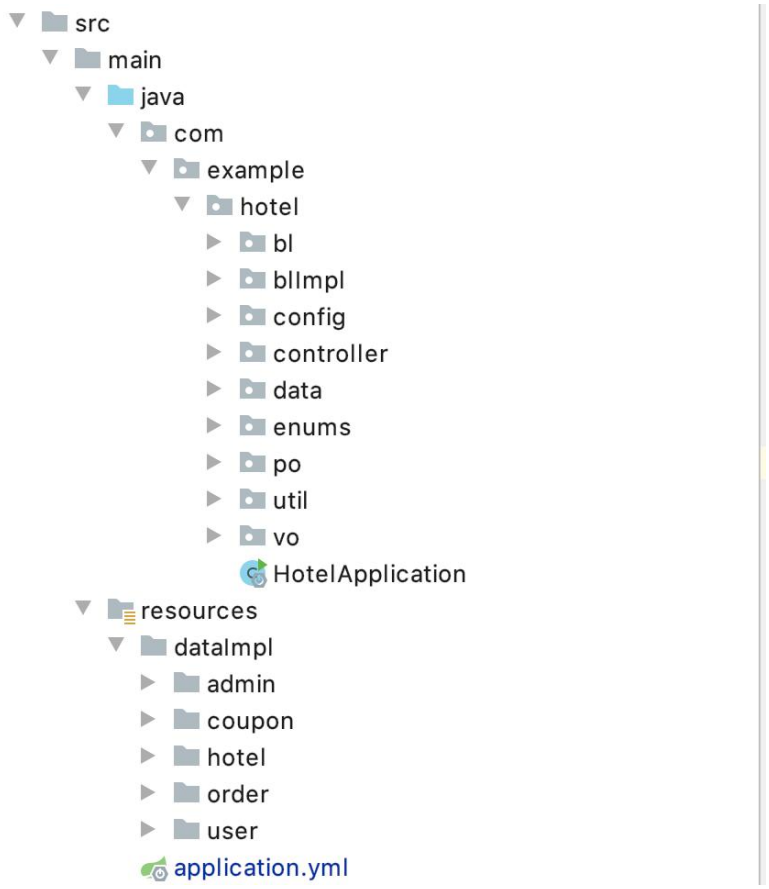

图 2-19 Spring Boot + Mybatis 项目结构

2.7.2 后端调用流程

整个后端调用的流程是 controller->bl->blImpl->data->dataImpl。

controller 是控制器。承担了接受 REST API 请求之后，根据 URL 找到对应的 bl 来处理。bl 是业务逻辑的接口。blImpl 才是业务逻辑真正的实现。blImpl 会调用数据层的接口

data。dataImpl 则是通过 Mybatis 给出的关于数据层接口的实现。将 data 接口与 SQL 语句相映射。

2.7.3 源代码解析

1. controller

controller.coupon.CouponController 类。其中：

(1) @RestController REST 风格 API 的控制器
(2) @RequestMapping("/api/coupon") 请求的 URL
(3) @PostMapping("/hotelTargetMoney") Post 请求子路径
(4) @RequestParam 请求参数
(5) @Autowired 自动装配
(6) couponVO 传给前端的数据

```
package com.example.hotel.controller.coupon;

import com.example.hotel.bl.coupon.CouponService;
import com.example.hotel.vo.CouponVO;
import com.example.hotel.vo.HotelTargetMoneyCouponVO;
import com.example.hotel.vo.OrderVO;
import com.example.hotel.vo.ResponseVO;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/coupon")
public class CouponController {

    @Autowired
    private CouponService couponService;

    @PostMapping("/hotelTargetMoney")
    public          ResponseVO          addHotelTargetMoneyCoupon(@RequestBody
HotelTargetMoneyCouponVO hotelTargetMoneyCouponVO) {

        CouponVO                              couponVO                              =
couponService.addHotelTargetMoneyCoupon(hotelTargetMoneyCouponVO);

        return ResponseVO.buildSuccess(couponVO);
    }

    @GetMapping("/hotelAllCoupons")
    public ResponseVO getHotelAllCoupons(@RequestParam Integer hotelId) {
```

```
        return
ResponseVO.buildSuccess(couponService.getHotelAllCoupon(hotelId));
    }

    @GetMapping("/orderMatchCoupons")
    public ResponseVO getOrderMatchCoupons(@RequestParam Integer userId,
                                           @RequestParam Integer hotelId,
                                           @RequestParam Double orderPrice,
                                           @RequestParam Integer roomNum,
                                           @RequestParam String checkIn,
                                           @RequestParam String checkOut) {
        OrderVO requestOrderVO = new OrderVO();
        requestOrderVO.setUserId(userId);
        requestOrderVO.setHotelId(hotelId);
        requestOrderVO.setPrice(orderPrice);
        requestOrderVO.setRoomNum(roomNum);
        requestOrderVO.setCheckInDate(checkIn);
        requestOrderVO.setCheckOutDate(checkOut);
        return
ResponseVO.buildSuccess(couponService.getMatchOrderCoupon(requestOrderVO));
    }


}
```

2. bl

下面是 bl.coupon.CouponService 类。其中：

(1) HotelTargetMoneyCouponVO、hotelId 是前端输入的参数；

(2) OrderVO 是逻辑层返回的结果。

```
package com.example.hotel.bl.coupon;

import com.example.hotel.po.Coupon;
import com.example.hotel.vo.CouponVO;
import com.example.hotel.vo.HotelTargetMoneyCouponVO;
import com.example.hotel.vo.OrderVO;

import java.util.List;

public interface CouponService {
    /**
```

```
     * 返回某一订单可用的优惠策略列表
     * @param orderVO
     * @return
     */
    List<Coupon> getMatchOrderCoupon(OrderVO orderVO);  ## Use CouponVO instead
of Coupon

    /**
     * 查看某个酒店提供的所有优惠策略（包括失效的）
     * @param hotelId
     * @return
     */
    List<Coupon> getHotelAllCoupon(Integer hotelId);

    /**
     * 添加酒店满减优惠策略
     * @param couponVO
     * @return
     */
    CouponVO addHotelTargetMoneyCoupon(HotelTargetMoneyCouponVO couponVO);
}
```

3. blImpl

下面是 blImpl.coupon.CouponServiceImpl 类的代码，其中：
@Service 是逻辑对象的注解。
逻辑层服务的实现调用了数据层 Mapper 的接口。

```
```

```
@Service
public class CouponServiceImpl implements CouponService {


    private final  TargetMoneyCouponStrategyImpl targetMoneyCouponStrategy;

    private final  TimeCouponStrategyImpl timeCouponStrategy;
    private final CouponMapper couponMapper;

    private static List<CouponMatchStrategy> strategyList = new ArrayList<>();

    @Autowired
    public                      CouponServiceImpl(TargetMoneyCouponStrategyImpl
targetMoneyCouponStrategy,
                          TimeCouponStrategyImpl timeCouponStrategy,
                          CouponMapper couponMapper) {
        this.couponMapper = couponMapper;
        this.targetMoneyCouponStrategy = targetMoneyCouponStrategy;
        this.timeCouponStrategy = timeCouponStrategy;
        strategyList.add(targetMoneyCouponStrategy);
        strategyList.add(timeCouponStrategy);
    }



    @Override
    public List<Coupon> getMatchOrderCoupon(OrderVO orderVO) {

        List<Coupon> hotelCoupons = getHotelAllCoupon(orderVO.getHotelId());

        List<Coupon> availAbleCoupons = new ArrayList<>();

        for (int i = 0; i < hotelCoupons.size(); i++) {
            for (CouponMatchStrategy strategy : strategyList) {
                if (strategy.isMatch(orderVO, hotelCoupons.get(i))) {
                    availAbleCoupons.add(hotelCoupons.get(i));
                }
            }
        }

        return availAbleCoupons;
    }
```

```
    @Override
    public List<Coupon> getHotelAllCoupon(Integer hotelId) {
        List<Coupon> hotelCoupons = couponMapper.selectByHotelId(hotelId);
        return hotelCoupons;
    }

    @Override
    public     CouponVO     addHotelTargetMoneyCoupon(HotelTargetMoneyCouponVO
couponVO) {
        Coupon coupon = new Coupon();
        coupon.setCouponName(couponVO.getName());
        coupon.setDescription(couponVO.getDescription());
        coupon.setCouponType(couponVO.getType());
        coupon.setTargetMoney(couponVO.getTargetMoney());
        coupon.setHotelId(couponVO.getHotelId());
        coupon.setDiscountMoney(couponVO.getDiscountMoney());
        coupon.setStatus(1);
        int result = couponMapper.insertCoupon(coupon);
        couponVO.setId(result);
        return couponVO;
    }
}
```

2. data

下面是 Data.coupon.CouponMapper 的代码，其中：

(1) @Mapper 注解是 mybatis 的注解，是用来说明这个是一个 Mapper，对应的 xxxMapper.xml 就是来实现这个 Mapper。

(2) @Repository 注解是 Spring 的注解，使用该注解把当前类注册成一个 Java Bean，并将其标记为数据持久层。

```
package com.example.hotel.data.coupon;

import com.example.hotel.po.Coupon;
import org.apache.ibatis.annotations.Mapper;
import org.springframework.stereotype.Repository;

import java.util.List;

@Mapper
@Repository
public interface CouponMapper {
```

```
    int insertCoupon(Coupon coupon);

    List<Coupon> selectByHotelId(Integer hotelId);
}
```

5. dataImpl

下面是 dataImpl.coupon.CouponMapper.xml 文件，其中:

(1) namespace: 接口的 ID

(2) id: 方法的 ID

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-1-mapper.dtd" >
<mapper namespace="com.example.hotel.data.coupon.CouponMapper">

    <insert id="insertCoupon" parameterType="com.example.hotel.po.Coupon"
            useGeneratedKeys="true" keyProperty="id">
        insert into
coupon(description,couponName,target_money,discount_money,start_time,end_tim
e,hotelId,couponType,discount,status)

values(#{description},#{couponName},#{targetMoney},#{discountMoney},#{startT
ime},#{endTime},#{hotelId},#{couponType},#{discount},#{status})
    </insert>  <!--  插入一个优惠券  -->

    <select id="selectByHotelId" resultMap="Coupon">
        select * from Coupon where hotelId=#{hotelId}
    </select>


    <resultMap id="Coupon" type="com.example.hotel.po.Coupon">
        <result column="description" property="description"></result>
        <result column="id" property="id"></result>
        <result column="couponName" property="couponName"></result>
        <result column="hotelId" property="hotelId"></result>
        <result column="couponType" property="couponType"></result>
        <result column="discount" property="discount"></result>
        <result column="status" property="status"></result>
        <result column="target_money" property="targetMoney"></result>
        <result column="discount_money" property="discountMoney"></result>
        <result column="start_time" property="startTime"></result>
```

```
        <result column="end_time" property="endTime"></result>
    </resultMap>
</mapper>
```

练习 2

1. 【单选题】下列说法错误的是?
   A. TCP/IP 协议栈分为四层，分别是应用层、传输层、网络层和网络接口层。
   B. HTTP 协议属于应用层。
   C. TCP 与 UDP 协议是传输层协议。
   D. 网络层为上层提供物理上的可靠的数据传输。
2. 【判断题】当应用层协议使用 TCP/IP 协议传输数据时，TCP/IP 协议簇可能会将应用层发送的数据分成多个包依次发送，而数据的接收方收到的数据可能是分段的或者拼接的，所以它需要对接收的数据进行拆分或者重组。
3. 【判断题】HTTP 是一个基于客户端和服务器模型的无状态的协议。
4. 【简答题】简述 HTTP 一次操作的工作流程。
5. 【简答题】如果 HTTP 请求的头信息加入了 Connection:keep-alive，则服务器会如何处理?
6. 【单选题】HTTP 协议中 URL 携带多个参数时用什么符号隔开?
   A. ?
   B. &
   C. =
   D. /
7. 【判断题】HTTP 请求中，GET 请求用于获取资源，POST 请求用户增加或修改资源。
8. 【判断题】HTTP 回答中状态码，4xx：服务器错误。例如 404（找不到内容）。
9. 【简单题】简述 Tomcat 的工作原理和 Servlet 的作用。
10. 【判断题】每次服务器接收到一个 Servlet 请求时，服务器会产生一个新的线程并调用服务。service() 方法检查 HTTP 请求类型（GET、POST、PUT、DELETE 等），并在适当的时候调用 doGet、doPost、doPut，doDelete 等方法。
11. 【简答题】前后端分离的好处是什么？为什么 REST API 可以实现前后端分离?
12. 【单选题】获取 id 为 123 的 Product 用哪一个 REST API?
    A. GET /api/products/123
    B. GET /api/products

C. POST /api/products/123

D. PUT /api/products/123

13. 【判断题】JSON 本质是一个正则表达式，JSON 的 parser 是一个有限状态机。

14. 【判断题】启动 Spring Boot 应用程序只需要一行代码加上一个注解@SpringBootApplication。

15. 【判断题】对 URL “/” 路径的响应是通过 HelloController 类来实现的。需要添加@RestController 和@RequestMapping("/")注解来说明。

16. 【简答题】简述启动 Springboot 项目的 3 种方式。

17. 【判断题】表（关系 Relation）是以行（值组 Tuple）和列（属性 Attribute）的形式组织起来的数据的集合。

18. 【简答题】论述概念模型、逻辑模型和物理模型的区别，和转换过程。

19. 【简答题】简述如何将类图转换为关系表。

20. 【简答题】论述 JDBC 完成 SQL 操作的主要步骤。

21. 【单选题】下列关于 DAO 的描述错误的是?

A. DAO 是实现以面向对象 API 为形式的对持久化数据进行 CRUD 操作的对象。

B. DAO 是位于业务逻辑层和数据层之间。

C. DAO 只是为了实现对不同数据库有统一的访问 API。

D. 利用 DAO 可以将对逻辑对象的操作就直接转换为对数据库中记录的操作。

22. 【判断题】Hibernate 和 MyBatis 是常见的两种 ORM 框架，都实现了关系到对象的双向映射。

9.