目录

这部分内容主要是帮助学员了解Web编程基础

第一篇：HTML - 构建网页的基石

目标：掌握HTML的基本语法与常用标签，能够构建简单的网页结构。

第一章：HTML基础

- HTML简介与历史
- HTML文档结构（<!DOCTYPE>、<html>、<head>、<body>）
- 常用标签（<h1>-<h6>、<p>、<a>、<img>、 、）
- 注释与代码格式

第二章：HTML表单与输入

- 表单标签（<form>）
- 输入类型（<input>、<textarea>、<select>、<button>）
- 表单属性（action、method、name、placeholder、required）
- 表单验证（HTML5原生验证）

第三章：HTML多媒体与语义化

- 多媒体标签（<audio>、<video>、<iframe>）
- 语义化标签（<header>、<footer>、<nav>、<article>、<section>）
- HTML5新特性（<canvas>、<svg>）

第四章：HTTP与Web基础

- HTTP简介（超文本传输协议）
- HTTP请求与响应的基本流程
- HTTP请求方法（GET、POST、PUT、DELETE）
- HTTP状态码（200、404、500等）
- HTTP头部（请求头与响应头）
- HTTPS与安全性

第二篇：CSS - 让网页焕发生机

目标：掌握CSS的基本语法与常用样式属性，能够实现网页的美化与布局。

第五章：CSS基础

- CSS简介与引入方式（内联、内部、外部）
- 选择器（标签选择器、类选择器、ID选择器、后代选择器、伪类选择器）

- 常用样式属性（颜色、字体、背景、边框、内外边距）

第六章：CSS布局

- 盒模型（Box Model）
- 浮动（float）与清除浮动
- 定位（position：static、relative、absolute、fixed）
- 弹性布局（Flexbox）
- 网格布局（Grid）

第七章：CSS进阶

- 响应式设计（媒体查询 @media）
- 动画与过渡（transition、animation）
- 变量与自定义属性（var()）
- CSS预处理器（Sass/SCSS 简介）

第三篇：JavaScript - 赋予网页灵魂

目标：掌握JavaScript的基本语法与核心概念，能够实现网页的动态交互。

第八章：JavaScript基础

- JavaScript简介与引入方式
- 变量与数据类型（let、const、var）
- 运算符与表达式
- 条件语句（if-else、switch）
- 循环语句（for、while、do-while）

第九章：函数与作用域

- 函数定义与调用
- 参数与返回值
- 作用域与闭包
- 箭头函数

第十章：DOM操作

- DOM简介与节点类型
- 选择元素（getElementById、querySelector、querySelectorAll）
- 操作元素（修改内容、样式、属性）
- 事件处理（addEventListener、常见事件类型）

第十一章：JavaScript进阶

- 数组与对象操作
- 异步编程（setTimeout、Promise、async/await）
- AJAX与Fetch API
- ES6+新特性（模板字符串、解构赋值、模块化）

第十二章：Web 存储与 Cookie

第四篇：工具与技巧 - 打造高效开发之路

目标：熟悉Web开发的常用工具与最佳实践，提升开发效率。

第十三章：开发工具

- 浏览器开发者工具（Chrome DevTools）
- 代码编辑器（VS Code 使用技巧）
- 版本控制（Git 基础）

第十四章：调试与优化

- JavaScript调试技巧
- 性能优化（减少HTTP请求、压缩资源、懒加载）
- 跨浏览器兼容性

第十五章：构建工具

- 包管理工具（npm、yarn）
- 模块打包工具（Webpack 简介）
- 自动化工具（Gulp、Grunt）

附录：探索未来 - 迈向更广阔的世界

目标：为后续深入学习Web开发打下基础。

附录：前端框架简介

- React、Vue、Angular 简介
- 前端框架的选择与学习路径

附录：后端基础

- HTTP协议与RESTful API
- Node.js 简介
- 前后端分离开发模式

附录：Web安全

- 常见Web攻击（XSS、CSRF、SQL注入）
- 安全防护措施

附录：HTTP与Web通信

- HTTP协议的作用与历史
- 请求-响应模型
- RESTful API简介
- HTTPS与安全性

附录：基于HTML+CSS的UI框架一览

第五篇：实战与创新 - 从零到一的蜕变

目标：通过实际项目，巩固所学知识，提升综合开发能力。

练习01：静态网页-个人简历网站

练习02：动态网页 - 在线留言板

练习03：响应式网页项目 - 在线商城首页

练习项目：前后端协作原理 - 新增用户功能

进阶练习：

练习04：UI组件封装 - 视频播放组件

练习05：后端服务调用

总结

这个完整的大纲涵盖了从HTML、CSS、JavaScript的基础到进阶内容，再到工具使用、项目实战和扩展知识，帮助学习者从零开始逐步掌握Web开发的核心技能。每个篇章和章节的目标明确，内容循序渐进，适合初学者和有一定基础的学习者

导论：关于前端技术演进

1. 基础层：Web 三剑客

前端技术的根基由三大核心语言构成，它们定义了网页的**结构**、**样式**和**行为**：

- HTML（HyperText Markup Language）：负责页面内容的结构化描述，如标题、段落、表单等。现代 HTML5 新增了语义化标签（<header>, <article>）和多媒体支持。
- CSS（Cascading Style Sheets）：控制网页的视觉呈现，包括布局（Flexbox、Grid）、动画、响应式设计（媒体查询）。CSS3 引入了变量、阴影等高级特性。
- JavaScript实现动态交互逻辑：（如表单验证、数据请求）。ES6+ 标准带来了模块化、箭头函数、Promise 等现代化语法。

意义：三者共同构成网页的“骨骼-皮肤-肌肉”模型，是前端开发的不可替代基石。

---

2. UI 框架（CSS 框架）：快速构建视觉层

在原生 CSS 基础上，UI 框架通过预定义样式和组件，解决**开发效率**和**一致性**问题：

- 经典代表：
  - Bootstrap：栅格系统 + 组件库，适合快速搭建企业级界面。
  - Tailwind CSS：原子化 CSS 框架，通过组合类名实现高度定制化设计。
  - Bulma：纯 CSS 框架，无 JavaScript 依赖。
- 核心价值：
  - 避免重复编写 CSS
  - 统一设计规范（如间距、配色）
  - 响应式适配自动化

---

3. 前端框架：工程化与组件化

为应对复杂应用开发，前端框架通过**组件化**、**数据驱动**和**状态管理**提升工程能力：

- 三大主流框架：
  - React（Facebook）：基于虚拟 DOM 和函数式编程，生态庞大（Redux、Next.js）。
  - Vue.js（尤雨溪）：渐进式框架，API 简洁，适合中小型项目。 Angular
  - （Google）：全功能 MVC 框架，强类型（TypeScript 原生支持）。
- 技术演进：
  - 从 jQuery 的 DOM 操作 → 声明式编程（数据驱动视图）
  - 从单体应用 → 微前端架构
  - 从 CSR（客户端渲染）→ SSR/SSG（服务端渲染/静态生成）

---

4. UI 组件库：企业级设计系统

基于前端框架的二次封装，提供开箱即用的**高阶组件**和**设计规范**：

- 典型案例：
  - Ant Design（React）：阿里系企业级组件库，强调设计一致性。
  - Element UI / Element Plus（Vue.js）：饿了么团队出品，轻量易用。
  - Material-UI（React）：遵循 Google Material Design 规范。
- 核心特点：
  - 封装复杂交互（如表格分页、表单校验）
  - 提供主题定制能力
  - 与脚手架工具深度集成

---

5. 脚手架与工具链：工业化开发流程

通过标准化工具链解决**环境配置**、**构建优化**和**工程管理**问题：

- 脚手架工具：
  - Create React App（React）
  - Vue CLI（Vue.js）
  - Vite：新一代极速构建工具，支持多框架。
- 现代工具链：
  - 打包工具：Webpack、Rollup
  - 编译器：Babel（ES6+ 转译）、TypeScript
  - 代码规范：ESLint、Prettier
  - 包管理：npm、Yarn、pnpm

---

6. 扩展领域：现代前端技术生态

- 跨端开发：
  - React Native / Flutter（移动端）
  - Electron（桌面端）
  - Taro / Uni-app（小程序）
- 前沿趋势：
  - WebAssembly：突破 JavaScript 性能瓶颈
  - 微前端：解耦巨型单体应用
  - 低代码平台：通过可视化降低开发门槛

总结：技术选型逻辑

1. 需求驱动：简单页面（HTML/CSS）→ 复杂应用（框架+组件库）

2. 团队适配：技术栈统一性 > 追逐最新技术
3. 生态考量：社区活跃度、文档质量、长期维护性

前端技术已从简单的“页面制作”演变为涵盖工程化、跨平台、性能优化的**系统性学科**，理解其分层架构是掌握现代 Web 开发的关键。

第一章：HTML基础

1. HTML简介与历史

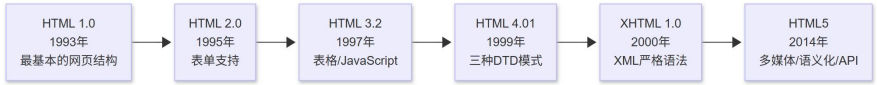
1.1 什么是 HTML?

HTML（HyperText Markup Language，超文本标记语言）是一种用于构建网页的标记语言，它通过 **标签（Tag）** 来定义网页的结构和内容，例如标题、段落、图片、链接等。

HTML 的特点：

- 采用 **标记语言**，不是编程语言。
- 主要用于 **描述网页结构**，而非逻辑控制。
- 结合 **CSS**（负责样式）和 **JavaScript**（负责交互）构建完整网页。

1.2 HTML 的发展历史



HTML 版本	发布年份	主要特性
HTML 1.0	1993年	最基本的网页结构
HTML 2.0	1995年	增加表单支持
HTML 3.2	1997年	增加表格、JavaScript支持
HTML 4.01	1999年	三种DTD模式（Strict、Transitional、Frameset）
XHTML 1.0	2000年	更严格的语法，基于XML
HTML5	**2014年**	**多媒体、语义化、API支持**

---

2. HTML 文档结构

2.1 HTML 基本结构

一个标准的 HTML 文档包含 <!DOCTYPE> 声明、<html> 结构以及 <head> 和 <body> 两部分。

```html
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>我的网页</title>
</head>
<body>
    <h1>欢迎来到我的网站</h1>
    <p>这是一个HTML基础教程。</p>
</body>
</html>

```

2.2 HTML 结构解析

标签	作用
<!DOCTYPE html>	声明 HTML5 文档类型
<html>	HTML 文档的根元素
<head>	页面头部，包含元数据
<meta charset="UTF-8">	设置字符编码
<meta name="viewport">	适配移动端
<title>	设置网页标题
<body>	页面可见内容

一、基础声明层

1. 文档类型声明 <!DOCTYPE html>

- 唯一作用：声明文档遵循 HTML5 规范
- 位置要求：必须位于文件首行
- 历史版本：HTML4.01/XHTML 需要更复杂的声明，HTML5 简化为此形式

二、根容器层 <html>

- 双标签结构：包裹整个文档内容
- 关键属性：

```
<html lang="en"> <!-- 定义文档语言（影响SEO/语音阅读） -->
```

- 嵌套层次：包含 <head> 和 <body> 两大模块

三、元数据层 <head>

- 不可见部分：存放浏览器和搜索引擎需要的关键信息
- 核心组件：

元素	作用	特殊属性
<meta charset="UTF-8">	定义字符编码	必须设置防止乱码
<title>	浏览器标签页标题	影响搜索引擎排名
<meta name="viewport">	移动端显示适配	width=device-width是关键
<link>	引入CSS/图标	rel="stylesheet"或rel="icon"
<script>	加载JavaScript	可设async/defer属性

四、内容呈现层 <body>

- 可见部分：展示给用户的所有内容
- 结构化元素：


```
<!-- 基础内容元素 -->
<h1>~<h6>   <!-- 标题分级（影响SEO权重） -->
<p>         <!-- 段落文本 -->
<img>       <!-- 图片（必须含 alt 属性） -->
<a>         <!-- 超链接（href 为必备属性） -->

<!-- 语义化容器（HTML5 新增） -->
<header>    <!-- 页眉 -->
<nav>       <!-- 导航栏 -->
<main>      <!-- 主要内容区 -->
<article>   <!-- 独立内容块 -->
<footer>    <!-- 页脚 -->
```

五、完整骨架示例

```html
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>页面标题</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <h1>网站主标题</h1>
        <nav>...</nav>
    </header>
    <main>
        <article>
            <h2>文章标题</h2>
            <p>正文内容...</p>
        </article>
    </main>
    <footer>版权信息</footer>
    <script src="app.js"></script>
</body>
</html>
```

六、关键技术特征

1. 树状结构：元素严格遵循父子嵌套关系
2. 标签分类：
   - 双标签： （含内容容器）
   - 单标签：<img>（自闭合元素）
3. 属性规则：

```
<input type="text" id="username" class="input-field">
<!-- 属性名=值 的键值对形式 -->
```

4. 注释语法：<!-- 注释内容 -->（不渲染到页面）

通过这种层次分明的结构设计，HTML 实现了内容与表现的分离，为 CSS 样式和 JavaScript 交互提供了清晰的接入点。

3. 常用HTML标签

3.1 标题标签（<h1> - <h6>）

```
<h1>一级标题</h1>
<h2>二级标题</h2>
<h3>三级标题</h3>

```

> **注意**：h1 代表最高级标题，一个网页通常只使用一个 h1，用于SEO优化。

3.2 段落标签（<p>）

```
<p>这是一个段落。</p>

```

> **特点**：每个 <p> 自动换行，适用于正文文本。

3.3 超链接标签（<a>）

```
<a href="https://www.example.com" target="_blank">访问示例网站</a>
```

属性	作用
href	设置链接地址
target="_blank"	_blank（在新窗口打开），_self（当前窗口打开）

3.4 图像标签（<img>）

```
<img src="image.jpg" alt="示例图片" width="300"/>
```

属性	作用
src	图片路径
alt	备用文本（SEO友好）
width	宽度

3.5 容器标签（ 和 ）

```
<div style="background-color: lightgray; padding: 10px;">
    这是一个 <div>，用于布局。
</div>
<span style="color: red;">这是一个 </span>，用于内联文本。</span>

```

标签	作用
	块级容器，适用于布局
	行内容器，适用于局部样式

---

4. 注释与代码格式

4.1 HTML 注释

```
<!-- 这是一个HTML注释 -->
<p>这是一段正常文本。</p>

```

> **作用**：注释不会显示在网页上，常用于代码说明和调试。

4.2 代码格式化规范

- 统一缩进（2 或 4 个空格）。
- 合理嵌套标签，避免错误结构。
- 代码属性加引号，如 class="example"。
- 避免多余空格，保持整洁。

---

5. 课后练习

5.1 实践任务

1. 搭建一个基础HTML页面，包含：
    - 一个标题（h1）
    - 一个段落（p）
    - 一个超链接（a）
    - 一张图片（img）
    - 一个 div 作为容器
2. 修正以下错误代码

```html
<html>
<head>
    <title>错误的HTML</title>
</head>
<body>
    <p>这里有一个段落
    <h1>标题</h1>
    <img src="image.jpg" alt=示例图片>
</body>
</html>
```

3. 思考题
   - 为什么 h1 只建议使用一次?
   - alt 属性的作用?
   - 和 的主要区别?

---

6. 总结

✓ **HTML 负责网页的结构和内容**

✓ **HTML5 提供了更丰富的标签与特性**

✓ **规范的代码格式提升可读性和可维护性**

下一步：学习 **HTML表单与输入控件**!

第二章：HTML表单与输入

1. 表单标签（<form>）

1.1 什么是 HTML 表单?

HTML 表单（Form）是用于收集用户输入数据的结构，常用于**用户注册、登录、搜索、提交信息等。**
表单的核心标签是 <form>，它包含多个输入控件，如文本框、按钮、复选框等。

1.2 <form> 标签的基本语法

```html
<form action="submit.php" method="POST">
    <label for="username">用户名：</label>
    <input type="text" id="username" name="username">
    <button type="submit">提交</button>
</form>

```

1.3 <form> 标签的常用属性

属性	作用
action	指定表单提交的目标 URL（如 submit.php）
method	表单提交方式（GET 或 POST）
enctype	数据编码方式（文件上传时需设为 multipart/form-data）
target	目标窗口（如 _blank、_self）
novalidate	取消 HTML5 原生表单验证

---

2. 输入类型（<input>、<textarea>、<select>、<button>）

HTML 提供多种表单输入控件，用于收集不同类型的数据。

2.1 文本输入（<input type="text">）

```
<label for="name">姓名：</label>
<input type="text" id="name" name="name" placeholder="请输入您的姓名"/>

```

- type="text"：用于输入普通文本。
- placeholder：提供提示文本。
- name：提交数据时的字段名称。

2.2 密码输入（<input type="password">）

```
<label for="password">密码：</label>
<input type="password" id="password" name="password"/>

```

- 输入时文本会被隐藏。

2.3 电子邮件输入（<input type="email">）

```
<label for="email">邮箱：</label>
<input type="email" id="email" name="email" required/>

```

- type="email"：只能输入符合电子邮件格式的数据。
- required：必填项。

2.4 数字输入（<input type="number">）

```
<label for="age">年龄：</label>
<input type="number" id="age" name="age" min="1" max="100"/>

```


- min/max：限制输入范围。

2.5 复选框（<input type="checkbox">）

```
<label>
    <input type="checkbox" name="agree"> 我同意条款</input>
</label>

```

- 适用于**多选**选项。

2.6 单选框（<input type="radio">）

```
<label><input type="radio" name="gender" value="male"> 男</label>
<label><input type="radio" name="gender" value="female"> 女</label>
```

- 多个单选框的 **name 属性相同，保证只能选择一个选项。**

2.7 文本区域（<textarea>）

```
<label for="message">留言：</label>
<textarea id="message" name="message" rows="4" cols="50"></textarea>
```

- rows 和 cols 控制大小，适用于**多行文本输入。**

2.8 下拉选择框（<select>）

```html
<label for="city">选择城市：</label>
<select id="city" name="city">
    <option value="beijing">北京</option>
    <option value="shanghai">上海</option>
    <option value="guangzhou">广州</option>
</select>
```

- <option> 定义可选项，**默认选项是第一个。**

2.9 提交按钮（<button>）

```html
<button type="submit">提交</button>
<button type="reset">重置</button>
<button type="button">普通按钮</button>
```

类型	作用
type="submit"	提交表单
type="reset"	重置表单

---

3. 表单属性（action、method、name、placeholder、required）

3.1 action（提交地址）

```html
<form action="submit.php">
```

- 指定**提交表单的目标 URL。**

3.2 method（提交方式）

```html
<form method="POST">
```

<form> 元素默认只支持 GET 和 POST 方法，不支持 PUT、DELETE 等其他 HTTP 方法。这是因为 HTML 表单的设计初衷是为了处理简单的数据提交，而 GET 和 POST 是最常用的两种方法。

提交方式	作用
GET	参数显示在 URL 上，适用于查询
POST	数据不会显示在 URL 上，适用于敏感数据

3.3 name（表单字段名称）

```html
<input type="text" name="username">
```

- name 用于**表单数据提交时的字段标识。**

3.4 placeholder（占位提示文本）

```html
<input type="text" placeholder="请输入您的姓名">
```

- 显示**提示信息**，但不会提交。

3.5 required（必填字段）

```html
<input type="email" required>
```

- 必须填写才能提交表单。

---

4. 表单验证（HTML5 原生验证）

HTML5 提供了原生的表单验证机制。

4.1 必填验证（required）

```
<input type="text" required>
```

- 用户必须输入值才能提交。

4.2 最小值/最大值（min / max）

```
<input type="number" min="18" max="60">
```

- 限制**数值范围**。

4.3 最小长度/最大长度（minlength / maxlength）

```
<input type="text" minlength="3" maxlength="10">
```

- 限制**字符长度**。

4.4 正则表达式匹配（pattern）

```
<input type="text" pattern="[A-Za-z]{3,10}" title="请输入3-10个字母"/>
```

- 只允许**3-10个字母**。

4.5 电子邮件格式验证（type="email"）


```
<input type="email">
```

- 只能输入符合**邮箱格式**的内容。

4.6 URL 格式验证（type="url"）

```
<input type="url">
```

- 只能输入符合**URL 格式**的内容。

---

5. 课后练习

5.1 实践任务

1. 创建一个简单的用户注册表单，包括：
    - 用户名（text）
    - 密码（password）
    - 性别（radio）
    - 爱好（checkbox）
    - 所在城市（select）
    - 自我介绍（textarea）
    - 提交按钮
2. 修正以下错误代码

```
<form action=submit.php method=get>
    <input type=text name=username>
    <button type=submit>提交</button>
</form>
```

3. 思考题
    - GET 和 POST 的区别?
    - 什么时候使用 required，什么时候使用 JavaScript 进行验证?

---

6. 总结

✅ **<form> 结构用于收集用户数据**

✅ **不同类型的 <input> 控件适用于不同的数据输入**

✅ **HTML5 提供了简单易用的表单验证机制**

下一步：学习 **HTML 多媒体与语义化标签**！

第三章：HTML多媒体与语义化

1. 多媒体标签（<audio>、<video>、<iframe>）

1.1 audio 标签（音频）

HTML5 提供了 <audio> 标签，用于在网页中嵌入音频文件。

```
<audio controls>
    <source src="au io.mp3" type="audio/mpeg"/>
    <source src="au io.ogg" type="audio/ogg"/>
    您的浏览器不支持 audio 标签。
</audio>
```

常用属性

属性	作用
controls	显示播放控制按钮（播放、暂停、音量）
autoplay	自动播放（部分浏览器可能需要用户交互）
loop	循环播放
muted	默认静音

1.2 video 标签（视频）

<video> 标签用于嵌入视频文件。

```
<video controls width="500">
    <source src="vi eo.mp4" type="video/mp4"/>
    <source src="vi eo.ogg" type="video/ogg"/>
    您的浏览器不支持 video 标签。
</video>
```

常用属性

属性	作用
controls	显示播放控制按钮
autoplay	自动播放
loop	循环播放
muted	默认静音
poster="image.jpg"	指定视频加载前的预览图片

1.3 iframe 标签（嵌入网页内容）

<iframe> 可用于嵌入其他网页、地图、视频等内容。

```
<iframe src="https://www.example.com" width="600" height="400"></iframe>
```

常见应用

- 嵌入 YouTube 视频：

```
<iframe width="560" height="315" src="https://www.youtube.com/embed/视频ID"
frameborder="0" allowfullscreen></iframe>
ameborder="0" allowfullscreen></iframe>
```

- 嵌入 Google 地图：

```
<iframe src="https://www.google.com/maps/embed?..."></iframe>
```

---

2. 语义化标签（<header>、<footer>、<nav>、<article>、<section>）

HTML5 引入了**语义化标签**，提高代码可读性和 SEO 友好性。

2.1 header（头部）

定义网页或一个内容块的标题部分。

```
<header>
    <h1>我的网站</h1>
    <p>欢迎访问我的网站！</p>
</header>
```

2.2 footer（底部）

定义网页的底部信息，如版权声明、联系方式等。

```
<footer>
    <p>版权所有 © 2025 我的公司</p>
</footer>
```

2.3 nav（导航）

定义页面的导航菜单。

```
<nav>
    <ul>
        <li><a href="index.html">首页</a></li>
        <li><a href="about.html">关于我们</a></li>
        <li><a href="contact.html">联系我们</a></li>
    </ul>
</nav>
```

2.4 article（文章块）

定义独立的文章或内容块，适用于博客、新闻等。


```
<article>
    <h2>HTML 语义化标签</h2>
    <p>使用 `article` 标签可以提高可读性和 SEO 友好性。</p>
</article>
```

2.5 section（页面分区）

用于划分网页中的不同部分。

```
<section>
    <h2>关于我们</h2>
    <p>我们是一家专业的软件开发公司。</p>
</section>

```

---

3. HTML5 新特性（<canvas>、<svg>）

3.1 canvas（绘制图形）

HTML5 的 <canvas> 标签用于绘制 2D 图形，如线条、矩形、圆形等。

```
<canvas id="myCanvas" width="400" height="200" style="border:1px solid #000;"></canvas>

<script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("2d");
    ctx.fillStyle = "blue";
    ctx.fillRect(50, 50, 100, 100); // 绘制蓝色矩形
</script>

```

> **特点**：基于 JavaScript 绘制图形，适合游戏开发、数据可视化等。
> E-Charts：https://echarts.apache.org/examples/zh/index.html
> Antv X6：https://x6.antv.antgroup.com/examples

3.2 svg（矢量图形）

<svg> 用于绘制矢量图形，具有**不失真、可缩放**的特点。

```
<svg width="100" height="100">
    <circle cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="red" />
</svg>
```

> **区别**：
>
> **Canvas**：基于像素绘制（适用于动画）。
> **SVG**：基于矢量绘制（适用于图标、图表）。
> bootstrap Icons：https://icons.getbootstrap.com

4. 课后练习

4.1 实践任务

1. 创建一个包含音频和视频的网页：
   - 添加 <audio> 播放一首音乐。
   - 添加 <video> 播放一个视频。
   - 使用 <iframe> 嵌入一个 YouTube 视频。
2. 使用 HTML5 语义化标签创建博客页面：
   - 包含 header（标题）。
   - nav（导航菜单）。
   - article（文章）。
   - section（不同部分）。
   - footer（版权信息）。
3. 绘制一个简单的图形：
   - 使用 <canvas> 画一个矩形。
   - 使用 <svg> 画一个圆形。

5. 总结

✓ **<audio> 和 <video> 可用于嵌入多媒体文件**
✓ **<iframe> 可用于嵌入网页、地图、视频等**
✓ **HTML5 语义化标签提高可读性和 SEO 友好性**
✓ **<canvas> 和 <svg> 适用于绘制图形，前者基于像素，后者基于矢量**

下一步：学习 **HTTP 与 Web 基础**！

第四章：HTTP 与 Web 基础

1. HTTP 简介（超文本传输协议）

1.1 什么是 HTTP?

HTTP（HyperText Transfer Protocol，超文本传输协议）是**客户端（浏览器）与服务器之间的通信协议**，用于请求和传输网页数据。

1.2 HTTP 的特点

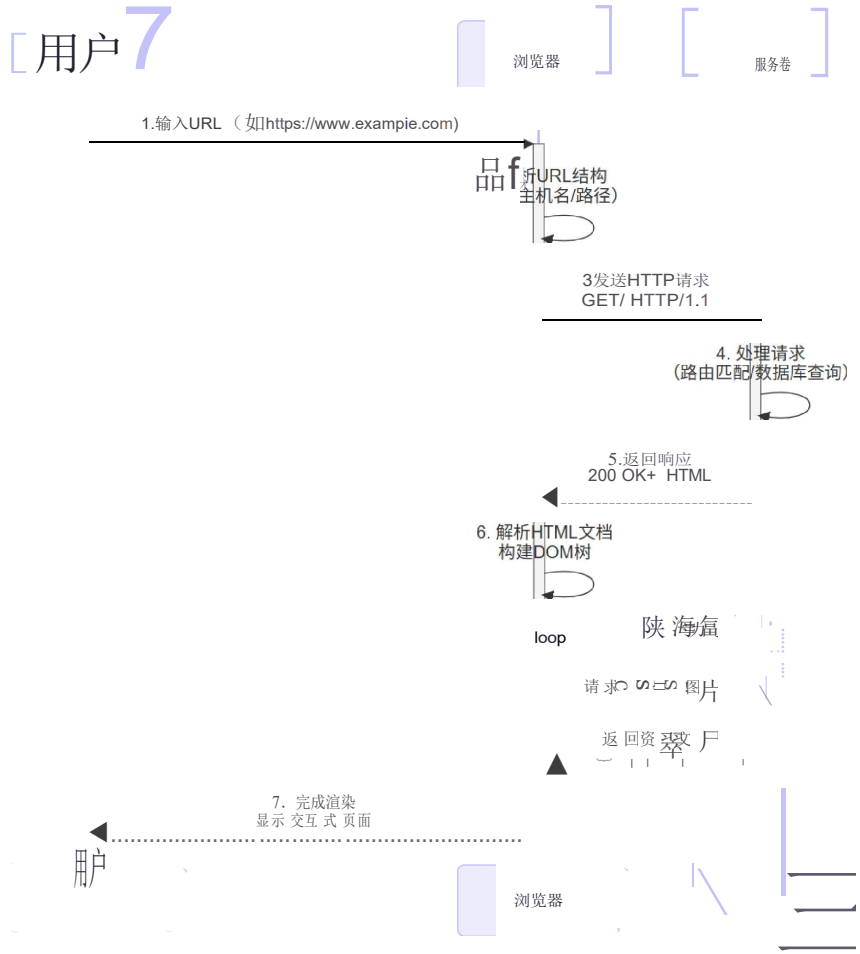
- 无状态：每次请求都是独立的，服务器不会记住上一次的请求状态。
- 基于请求-响应模式：浏览器（客户端）发送请求，服务器返回响应。
- 明文传输：默认情况下，HTTP 数据不加密，容易被窃听（可使用 HTTPS 解决）。

1.3 HTTP 的作用

- 负责**网页数据的传输**，如 HTML、CSS、JavaScript、图片等资源。
- 允许**前端与后端通信**，实现动态网页交互。

---

2. HTTP 请求与响应的基本流程



```
---
config:
  theme: default
---
sequenceDiagram
    participant 用户
    participant 浏览器
    participant 服务器

    用户 ->> 浏览器: 1. 输入URL（如 https://www.example.com）
    activate 浏览器
    浏览器 ->> 浏览器: 2. 解析URL结构<br/>（协议/主机名/路径）
    浏览器 ->> 服务器: 3. 发送HTTP请求<br/>GET / HTTP/1.1
    activate 服务器
    服务器 ->> 服务器: 4. 处理请求<br/>（路由匹配/数据库查询）
    服务器 -->> 浏览器: 5. 返回响应<br/>200 OK + HTML
    deactivate 服务器

    浏览器 ->> 浏览器: 6. 解析HTML文档<br/>构建DOM树
    loop 资源加载
        浏览器 ->> 服务器: 请求CSS/JS/图片
        activate 服务器
        服务器 -->> 浏览器: 返回资源文件
        deactivate 服务器
    end

    浏览器 -->> 用户: 7. 完成渲染<br/>显示交互式页面
    deactivate 浏览器
```

2.1 HTTP 工作流程

1. 用户输入 URL（如 https://www.example.com）。
2. 浏览器解析 URL，提取 **协议（http/https）**、**主机名**、**资源路径**。
3. 浏览器发送 HTTP 请求（请求方法 + 请求头）。
4. 服务器处理请求，返回响应（状态码 + 响应头 + 响应体）。
5. 浏览器解析 HTML，加载资源（CSS、JS、图片等）。
6. 页面展示给用户。

2.2 示例：HTTP 请求-响应

请求示例

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

响应示例

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024

<html>
  <body>
    <h1>欢迎访问我的网站</h1>
  </body>
</html>
```

---

3. HTTP 请求方法（GET、POST、PUT、DELETE）

HTTP 定义了多种请求方法，常见的有：

方法	作用	适用场景
GET	请求数据	访问网页、查询数据
POST	发送数据	提交表单、上传文件
PUT	更新数据	修改资源（如更新用户信息）
DELETE	删除数据	删除资源

3.1 GET 请求示例

```
GET /search?q=html HTTP/1.1
Host: www.example.com
```

- 参数放在 URL 中，如 ?q=html
- 适用于**无副作用的请求**（查询数据）

3.2 POST 请求示例

```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

username=admin&password=123456
```

- 参数放在请求体中
- 适用于**有副作用的请求**（提交数据）

---

4. HTTP 状态码（200、404、500 等）

HTTP 响应包含 **状态码**，表示请求的处理情况。

状态码	分类	说明
1xx	信息	请求已接收，继续处理
2xx	成功	请求成功（如 200 OK）
3xx	重定向	资源已移动（如 301 Moved Permanently）
4xx	客户端错误	请求错误（如 404 Not Found）
5xx	服务器错误	服务器异常（如 500 Internal Server Error）

4.1 常见状态码

状态码	说明
200 OK	请求成功
301 Moved Permanently	资源永久重定向
302 Found	资源临时重定向
400 Bad Request	错误请求
401 Unauthorized	未授权（需要身份验证）
403 Forbidden	服务器拒绝访问
404 Not Found	资源未找到
500 Internal Server Error	服务器内部错误

---

5. HTTP 头部（请求头与响应头）

5.1 请求头

请求头包含客户端发送给服务器的信息，如浏览器信息、接受数据类型等。

头部字段	说明
Host	服务器主机名
User-Agent	浏览器或客户端类型
Accept	可接受的响应数据类型
Content-Type	请求体的数据类型（如 application/json）
Authorization	认证信息（如 Bearer token）

请求头示例

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

5.2 响应头

服务器返回的头部信息，包括数据格式、缓存策略等。

头部字段	说明
Content-Type	响应数据类型（如 text/html）
Content-Length	响应数据的字节大小
Set-Cookie	设置 Cookie
Cache-Control	缓存控制（如 no-cache）
Location	重定向目标地址

响应头示例

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 512
Cache-Control: no-cache

```

---

6. HTTPS 与安全性

6.1 什么是 HTTPS?

HTTPS（HyperText Transfer Protocol Secure）是**加密版的 HTTP**，通过 **SSL/TLS** 加密数据传输，提高安全性。

6.2 HTTPS 的优势

- 数据加密：防止窃听，保证数据安全。
- 身份验证：防止中间人攻击（MITM）。
- 数据完整性：防止数据被篡改。

6.3 HTTP 与 HTTPS 的区别

特性	HTTP	HTTPS
安全性	明文传输，易被窃听	数据加密，安全性高
端口	80	443
证书	不需要	需要 SSL 证书
速度	较快	略慢（因加密过程）
SEO 友好	一般	更有利于 SEO 排名

---

7. 课后练习

7.1 实践任务

1. 分析以下 HTTP 请求和响应
   - 使用浏览器 **开发者工具（F12）** 观察 HTTP 请求和响应头。
   - 访问不同网站，查看 GET 和 POST 请求。
2. 使用 **curl 或 Postman 发送 HTTP 请求**
   - 使用 curl 发送 GET 请求：

```
curl -X GET "https://www.example.com"
```

   - 发送 POST 请求：

```
curl -X POST -d "username=admin&password=123" "https://www.example.com/login"
```

Postman 下载地址： https://www.postman.com/downloads/

3. 思考题
   - GET 和 POST 的主要区别?
   - 为什么 403 Forbidden 比 404 Not Found 更严重?
   - HTTPS 为什么比 HTTP 更安全?

---

8. 总结

✅ **HTTP 是 Web 的核心通信协议**

✅ **请求-响应模式决定了浏览器如何获取网页数据**
✅ **GET、POST、PUT、DELETE 用于不同的场景**
✅ **HTTPS 通过加密保证数据安全**

下一步：学习 **CSS 让网页焕发生机！**

第五章：CSS 基础

1. CSS 简介与引入方式

1.1 什么是 CSS?

CSS（Cascading Style Sheets，层叠样式表）用于**控制网页的外观和布局**，使 HTML 结构更加美观和易读。

1.2 CSS 的作用

✅ **美化网页**（颜色、字体、背景等）
✅ **布局控制**（位置、大小、间距）
✅ **提升用户体验**（动画、响应式设计）

1.3 CSS 的三种引入方式

1.3.1 内联样式（Inline CSS）

直接在 HTML 标签内使用 style 属性定义 CSS。

```
<p style="color: red; font-size: 20px;">这是一个红色的段落</p>
```

✅ **优点**：简单、适用于单个元素
❌ **缺点**：难以维护，代码冗余

1.3.2 内部样式（Internal CSS）

在 HTML 的 <head> 部分使用 <style> 标签定义样式。

```
<head>
    <style>
        p {
            color: blue;
            font-size: 18px;
        }
    </style>
</head>
<body>
    <p>这是一个蓝色的段落</p>
</body>

```

✓ **优点**：适用于单个页面的样式定义

✗ **缺点**：样式无法复用，影响代码结构

1.3.3 外部样式（External CSS）

将样式写在独立的 .css 文件中，并通过 <link> 引入。

CSS 文件（styles.css）：

```
p {
    color: green;
    font-size: 16px;
}
```

HTML 文件（index.html）：

```
<head>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <p>这是一个绿色的段落</p>
</body>

```


✓ **优点**：样式**可复用**，维护方便

✗ **缺点**：**需额外的 HTTP 请求**加载 CSS 文件

---

2. 选择器（Selectors）

选择器用于**匹配 HTML 元素**，以应用 CSS 样式。

CSS 核心选择器详解 —— 精准定位页面元素的工具

1. 基础选择器（核心三要素）

- Type Selectors 标签选择器
- Class Selectors 类选择器
- ID Selectors ID选择器
- Descendant Selector 后代选择器

---

2. 扩展选择器（复杂场景利器）

- Pseudo Selectors 伪类选择器`a:hover { color: blue; }`*定义元素特定状态（如悬停、焦点）*
- Attribute Selectors 属性选择器

2.1 标签选择器（Element Selector）

直接选中 HTML 标签，应用样式到所有相同标签。

```
p {
    color: blue;
}
```

作用：**所有 <p> 元素都会变蓝**

2.2 类选择器（Class Selector）

使用 . 选择具有特定 class 属性的元素。

```
.title {
    font-weight: bold;
    color: red;
}
```

```html
<p class="title">这是一个加粗的红色标题</p>
```

> 多个元素可共享同一个 class
> 可重复使用

2.3 ID 选择器（ID Selector）

使用 # 选择具有特定 id 的元素。

```css
#main-title {
    font-size: 24px;
    text-align: center;
}

```

```html
<h1 id="main-title">这是页面的主标题</h1>
```

✕ **注意**：id 在页面中应该是**唯一的。**

2.4 后代选择器（Descendant Selector）

匹配 **某个元素内的特定子元素。**

```css
div p {
    color: purple;
}

```



```html
<div>
    <p>这个段落是紫色的</p>
</div>
<p>这个段落不会受影响</p>

```

2.5 伪类选择器（Pseudo-classes）

用于选择**特定状态**的元素，如鼠标悬停、选中状态等。

伪类	作用
:hover	鼠标悬停时触发
:focus	输入框获得焦点时触发
:first-child	选中父元素的第一个子元素
:last-child	选中父元素的最后一个子元素

示例：鼠标悬停变色

```css
a:hover {
    color: orange;
}
```

```html
<a href="#">鼠标悬停我会变色</a>
```

2.6 属性选择器（Attribute Selectors）

一、基础类型

1. 存在性选择器

匹配具有指定属性的元素，不限制属性值
语法：[attribute]

```
[disabled] {
    opacity: 0.5;  /* 所有带disabled属性的元素变半透明 */
}

```

2. 精确匹配选择器
属性值必须完全等于目标值
语法：[attribute="value"]

```
input[type="text"] {
    border: 2px solid red;  /* 精确匹配type为text的输入框 */
}

```

二、模糊匹配类型
属性值中包含指定字符串即可
语法：[attribute*="value"]

```
img[src*="logo"] {
    filter: grayscale(100%);  /* 图片路径包含logo的变灰 */
}

```

1. 开头匹配选择器
属性值以指定字符串开头
语法：[attribute^="value"]


```
a[href^="tel:"]::after {
    content: "☎";  /* 电话链接添加电话图标 */
}

```

2. 结尾匹配选择器
属性值以指定字符串结尾
语法：[attribute$="value"]

```
a[href$=".pdf"] {
    color: red;  /* PDF链接显示为红色 */
}

```

三、特殊场景类型
属性值为多个空格分隔的单词时匹配
语法：[attribute~="value"]

```
[data-tags~="urgent"] {
    background: #ffeb3b;  /* 标签含urgent的元素高亮 */
}

```

1. 连字符前缀匹配
匹配属性值以连字符分隔且前缀符合的情况（常用于语言代码）
语法：[attribute|="value"]

```
[lang|="zh"] {
    font-family: "PingFang SC";  /* 中文语言版本特殊字体 */
}

```

四、组合使用技巧

```
input[type="email"][required] {
    border: 2px solid #4CAF50;  /* 必填的邮箱输入框 */
}

```

• 伪类联动：结合交互状态

```
a[href^="http"]:hover::after {
    content: "（外部链接）";  /* 鼠标悬停时提示外部链接 */
}

```

五、实际应用场景

1. 表单验证增强

```
input:invalid[required] {
    box-shadow: 0 0 8px red;  /* 必填项未填时显示红框 */
}

```

2. 国际化适配

```
[lang|="en"] { quotes: '"' '"'; }  /* 英文引号样式 */

```

3. 动态数据标记

```
tr[data-status="expired"] td {
    text-decoration: line-through;  /* 过期数据划除显示 */
}

```

优先级注意：属性选择器优先级与类选择器相同，低于ID选择器。建议结合浏览器开发者工具验证样式覆盖关系。

---

3. 常用样式属性

3.1 颜色（Color）

```
h1 {
    color: red; /* 颜色名称 */
    color: #ff0000; /* 十六进制颜色 */
    color: rgb(255, 0, 0); /* RGB 颜色 */
}

```

3.2 字体（Font）

属性	作用
font-size	字体大小
font-weight	字体粗细（bold、normal）
font-style	斜体（italic）
font-family	字体

示例

```
p {
    font-size: 18px;
    font-weight: bold;
    font-style: italic;
    font-family: Arial, sans-serif;
}

```

3.3 背景（Background）

属性	作用
background-color	背景颜色
background-image	背景图片
background-size	背景大小（cover、contain）

示例

```
body {
    background-color: #f0f0f0;
    background-image: url('background.jpg');
    background-size: cover;

}

```

3.4 边框（Border）

```
div {
    border: 2px solid black;
    border-radius: 10px; /* 圆角 */
}

```

关键字	作用
solid	实线
dashed	虚线

dotted	点状线

3.5 内外边距（Margin & Padding）

属性	作用
margin	设置外边距
padding	设置内边距

示例

```
div {
    margin: 20px;
    padding: 10px;
}

```

4. 课后练习

4.1 实践任务

1. 创建一个 HTML 页面并使用 CSS 样式
:
    - 让 <h1> 变红色
    - 让 <p> 变蓝色
    - 添加一个带背景图片的
    - 给 <a> 添加鼠标悬停变色效果
2. 选择器练习
    - 使用 **类选择器** 让多个元素共享样式
    - 使用 **ID 选择器** 定义唯一样式
    - 使用 **后代选择器** 让 div 内的 p 变紫色
3. 思考题
    - 为什么建议使用外部 CSS 而不是内联 CSS?
    - 类选择器和 ID 选择器的区别是什么?
    - 伪类 **:hover 可以用于哪些 HTML 元素?**

5. 总结

✓ **CSS 通过不同方式引入，外部 CSS 最推荐**

✓ **使用选择器可以灵活地应用样式**

✓ **常用样式包括颜色、字体、背景、边框、内外边距**

✓ **伪类可以用于用户交互，如 :hover 和 :focus**

下一步：学习 **CSS 布局（盒模型、浮动、定位、Flexbox）**！

第六章：CSS 布局

1. 盒模型（Box Model）

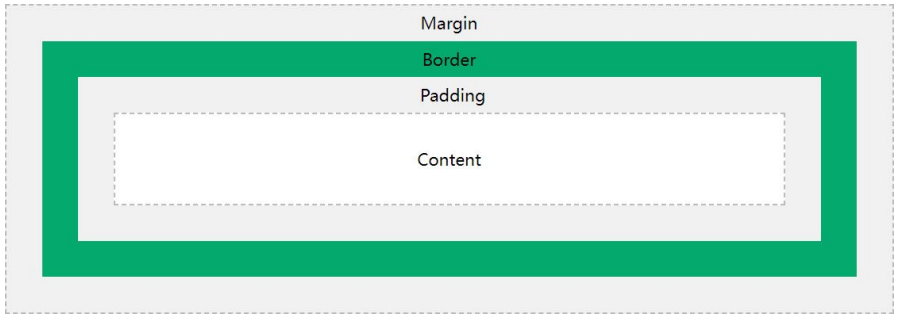
1.1 什么是 CSS 盒模型?

在 CSS 中，谈论设计和布局时会使用术语“盒子模型”。

CSS 盒子模型本质上是一个包裹每个 HTML 元素的盒子。它由以下部分组成：内容、填充、边框和边距。

盒模型（Box Model）是**HTML 元素的布局基础**，它定义了元素的 **内容（content）**、**内边距（padding）**、**边框（border）** 和 **外边距（margin）**。

1.2 盒模型结构



不同部分的解释：

- Content（内容） - 框的内容，文本和图像出现的位置
- Padding（内边距） - 清除内容周围的区域。填充是透明的
- Border（边框） - 围绕填充和内容的边框 Margin
- （外边距） - 清除边框外的区域。边距是透明的

框模型允许我们在元素周围添加边框，并定义元素之间的空间。

```
div {
    width: 300px;
    border: 15px solid green;
    padding: 50px;
    margin: 20px;
}
```

1.3 盒模型属性

属性	作用
content	盒子内容区域
padding	内边距，**增大盒子大小**
border	边框
margin	外边距，**控制盒子间距**

```
.box {
    width: 200px;
    height: 100px;
    padding: 20px;
    border: 5px solid black;
    margin: 10px;
}

```

1.4 box-sizing 控制盒模型

box-sizing	计算方式
content-box	**默认模式**，width 只包含 content
border-box	width 包含 padding 和 border

```
.box {
    box-sizing: border-box;
}

```

✅ **推荐使用 border-box 以避免计算宽度时的额外问题！**

---

2. 浮动（float）与清除浮动

2.1 什么是浮动？

float 让元素脱离文档流，**可以左对齐或右对齐**，通常用于**图片环绕文本、简易布局。**

属性值	作用
float: left	元素向左浮动
float: right	元素向右浮动
float: none	默认，不浮动

```
.img {
    float: left;
    margin-right: 10px;
}

```

2.2 清除浮动

浮动元素不会影响父元素高度，需要使用 clear 清除浮动。

```
.clearfix::after {
    content: "";
    display: block;
    clear: both;
}
```

```
<div class="clearfix">
    <img src="image.jpg" class="img">
    <p>这是一段文字，旁边有图片。</p>
</div>

```

✅ **推荐使用 clearfix 解决浮动问题！**

---

3. 定位（position）

3.1 position 属性

属性值	作用
static	默认值，**元素按正常流排列**
relative	**相对自身原位置** 进行偏移
absolute	**相对最近的已定位父元素** 进行偏移
fixed	**相对浏览器窗口** 进行固定定位

3.2 relative 相对定位

```
.box {
    position: relative;
    top: 20px;
    left: 10px;
}

```

✓ **元素原本的位置仍占据空间，但偏移了一定距离！**

3.3 absolute 绝对定位

```
.box {
    position: absolute;
    top: 50px;
    left: 100px;
}

```

✓ **绝对定位的元素不会占据原位置，而是脱离文档流！**

3.4 fixed 固定定位

```
.fixed-box {
    position: fixed;
    bottom: 10px;
    right: 10px;
}

```

✓ **适用于固定导航栏、返回顶部按钮等！**

---

4. 弹性布局（Flexbox）

4.1 什么是 Flexbox?

Flexbox（弹性盒子模型）用于**灵活控制子元素排列、对齐和分布**，特别适用于**自适应布局。**

4.2 display: flex 启用弹性布局

```
.container {
    display: flex;
}

```

4.3 justify-content（主轴对齐）

属性值	作用
flex-start	**默认**，子元素靠左对齐
flex-end	子元素靠右对齐
center	子元素居中对齐
space-between	**两端对齐，间距均匀**
space-around	**两侧保留间距，间距均匀**

```
.container {
    display: flex;
    justify-content: space-between;
}

```

4.4 align-items（交叉轴对齐）

属性值	作用
stretch	**默认值**，子元素拉伸至充满容器
flex-start	顶部对齐
flex-end	底部对齐
center	垂直居中对齐

```
.container {
    display: flex;
    align-items: center;
}
```

✅ **常见的 Flexbox 布局**


```
.container {
    display: flex;
    justify-content: center;
    align-items: center;
}
```

---

5. 网格布局（Grid）

5.1 什么是 CSS Grid?

Grid（网格布局）是一种**强大的二维布局方式**，适用于复杂网页布局。

5.2 display: grid 启用网格

```
.grid-container {
    display: grid;
    grid-template-columns: 100px 100px 100px;
    grid-template-rows: 50px 50px;
}

```

✅ **定义了 3 列，每列 100px 宽，2 行，每行 50px 高!**

5.3 grid-template-columns（列定义）

语法	作用
100px 200px auto	设定固定列宽
repeat(3, 1fr)	3 列，等比例分配

5.4 grid-gap（网格间距）

```
.grid-container {
    grid-gap: 10px;
}

```

✓ **适用于设置网格间距，避免元素紧贴！**

5.5 grid-area（区域布局）

```
.grid-container {
    display: grid;
    grid-template-areas:
        "header header"
        "sidebar main";
}
.header { grid-area: header; }
.sidebar { grid-area: sidebar; }
.main { grid-area: main; }

```

✓ **适用于复杂的网页布局设计！**

---

6. 课后练习

6.1 实践任务

1. 创建一个包含 **box-sizing: border-box 的盒模型**
2. 使用 **float 实现左右布局**
3. 使用 **position: absolute 定位一个按钮**
4. 使用 **Flexbox 实现水平居中**
5. 使用 **Grid 创建 3 列 2 行的网格**

---

7. 总结

✓ **盒模型是网页布局的基础**

✓ **float 适用于简单布局，但要清除浮动**

✓ **position 提供灵活定位方式**

✓ **Flexbox 适用于一维布局，Grid 适用于二维布局**

下一步：学习 **CSS 进阶（响应式设计、动画、CSS 变量）！**

第七章：CSS 进阶

1. 响应式设计（媒体查询 @media）

1.1 什么是响应式设计?

响应式设计（Responsive Design）让网页在**不同设备**（桌面、平板、手机）上都能良好显示，**核心技术是 CSS 的 @media 媒体查询。**

✓ **适应不同屏幕尺寸**

✓ **提升用户体验**

✓ **减少维护多套代码的需求**

1.2 @media 语法

```
@media 条件 {
    /* CSS 规则 */
}

```

示例：不同屏幕大小的样式


```
/* 桌面端（默认样式） */
body {
    background-color: white;
}
/* 平板端（屏幕宽度 ≤ 768px） */
@media (max-width: 768px) {
    body {
        background-color: lightblue;
    }
}
/* 手机端（屏幕宽度 ≤ 480px） */
@media (max-width: 480px) {
    body {
        background-color: lightgreen;
    }
}
```

✓ **页面背景色会根据设备屏幕大小变化!**

1.3 @media 断点推荐

设备	屏幕宽度（px）
超小屏幕（手机）	max-width: 480px
小屏幕（平板）	max-width: 768px
中等屏幕（笔记本）	max-width: 1024px
大屏幕（桌面）	min-width: 1025px

---

2. 动画与过渡（transition、animation）

2.1 transition 过渡效果

transition 让元素在**属性变化时**产生平滑动画。

示例：鼠标悬停变色

```css
.button {
    background-color: blue;
    color: white;
    padding: 10px 20px;
    transition: background-color 0.5s ease;
}

.button:hover {
    background-color: red;
}

```

✅ **鼠标悬停时背景色从蓝色渐变为红色！**

2.2 animation 关键帧动画

animation 让元素可以执行**复杂动画**。

示例：文本闪烁

```css
@keyframes blink {
    0% { opacity: 1; }
    50% { opacity: 0; }
    100% { opacity: 1; }
}

.text {
    animation: blink 1s infinite;
}

```

✅ **文本每秒闪烁一次！**

2.3 animation 语法


```css
@keyframes 动画名称 {
    0% { 初始状态 }
    100% { 结束状态 }
}

.元素 {
    animation: 动画名称 时长 速度 无限循环;
}

```

示例：移动盒子

```css
@keyframes move-box {
    from { transform: translateX(0); }
    to { transform: translateX(100px); }
}

.box {
    animation: move-box 2s ease-in-out infinite;
}

```

✅ **盒子会左右移动！**

---

3. 变量与自定义属性（var()）

3.1 CSS 变量

CSS 变量（Custom Properties）使样式更加灵活、可复用。

定义变量

```
:root {
    --primary-color: blue;
    --text-size: 18px;
}

button {
    background-color: var(--primary-color);
    font-size: var(--text-size);
}

```

✓ **修改 --primary-color 即可统一改变所有按钮颜色！**

3.2 var() 语法

```
var(--变量名, 默认值)

```

示例：按钮颜色切换

```
:root {
    --btn-color: green;
}

button {
    background-color: var(--btn-color, gray);
}

```

✓ **如果 --btn-color 未定义，则使用默认值 gray。**

---

4. CSS 预处理器（Sass/SCSS 简介）

4.1 什么是 CSS 预处理器?

CSS 预处理器（Sass/SCSS）是**增强版 CSS**，提供 **变量、嵌套、函数** 等高级功能，提高开发效率。

✓ **更易维护**（支持变量、模块化）

✓ **减少代码重复**（支持 mixin）

✓ **增强可读性**（支持嵌套规则）

4.2 Sass vs SCSS

语法	说明
Sass	**无 {} 和 ;，简洁但不兼容 CSS**
SCSS	**兼容 CSS 语法，更流行**

4.3 变量

```
$primary-color: blue;

button {
    background-color: $primary-color;
}

```

✓ **Sass 变量类似 CSS 变量，但支持更多功能！**

4.4 嵌套

```scss
nav {
    ul {
        list-style: none;

        li {
            display: inline-block;
            a {
                text-decoration: none;
            }
        }
    }
}
```

✓ **避免重复书写选择器，提高代码结构清晰度！**

4.5 Mixin（代码复用）

```scss
@mixin button-style {
    padding: 10px;
    border-radius: 5px;
    background-color: red;
}

button {
    @include button-style;
}

```

✓ **@mixin 定义代码块，@include 复用样式！**

---

5. 课后练习

5.1 实践任务

1. 使用 **@media 让网页在不同屏幕尺寸下自适应**
2. 给按钮添加 **transition 过渡动画**
3. 实现 **animation 让元素来回移动**
4. 定义 **CSS 变量 统一管理颜色**
5. 用 **Sass 创建 mixin 复用按钮样式**

---

6. 总结

✓ **@media 让网页适配不同设备**
✓ **transition 让样式变化更平滑**
✓ **animation 让网页更具动感**
✓ **CSS 变量减少重复代码，提高可维护性**
✓ **Sass/SCSS 提供变量、嵌套、复用功能，增强 CSS 能力**

下一步：学习 **JavaScript，赋予网页交互功能！**

第八章：JavaScript 基础

notes: 附录：JavaScript基础练习环境搭建 | 高级环境参照 - https://gitee.com/eip-xauat-y22/basic-javascript

1. JavaScript 简介与引入方式

1.1 什么是 JavaScript?

JavaScript（简称 JS）是一种**基于浏览器的脚本语言**，用于**实现网页交互**，如**动态内容、事件处理、数据操作**等。

✓ **前端三大核心技术**

- HTML：负责网页结构
- CSS：负责网页样式
- JavaScript：负责网页交互

✓ **JavaScript 主要用途**

- 动态修改 HTML 和 CSS（如按钮点击改变颜色）
- 处理用户输入（如表单验证）
- 发送网络请求（如 Ajax 获取数据）
- 实现动画效果（如轮播图）

1.2 JavaScript 的引入方式

1. 内联 JavaScript

直接在 HTML 标签内使用 onclick 等事件属性。

```
<button onclick="alert('Hello, JavaScript!')">点击我</button>
```

✓ **适用于简单功能，不推荐大量使用**

---

2. 内部 JavaScript

在 HTML 页面中的 <script> 标签内编写 JS 代码。


```
<script>
    alert("欢迎访问我的网站！");
</script>
```

✓ **适用于小型脚本**

---

3. 外部 JavaScript

将 JavaScript 代码写在 .js 文件中，并用 <script> 引入。

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

script.js 文件

```
alert("这是外部 JavaScript 文件！");
```

✓ **推荐使用，代码结构清晰，可复用**

---

2. 变量与数据类型

2.1 变量的声明

在 JavaScript 中，可以使用 var、let 和 const 声明变量。

关键字	作用	作用域	是否可修改
var	旧版变量声明方式，**不推荐**	函数作用域	✓ 可修改
let	**推荐**，支持块级作用域	块级作用域	✓ 可修改
const	**常量**，不能重新赋值	块级作用域	✗ 不能修改

```
var name = "张三";  // 旧方式（不推荐）
let age = 25;       // 推荐
const PI = 3.1415;  // 常量，不能修改

```

✅ **推荐使用 let 和 const，避免 var 造成作用域问题**

2.2 数据类型

JavaScript 主要有 6 种基本数据类型：

数据类型	说明	示例
String	字符串	"Hello"
Number	数字（整数或小数）	100, 3.14
Boolean	布尔值	true, false
Undefined	未定义	let x;
Null	空值	let y = null;
Symbol	唯一值	Symbol('id')

---

3. 运算符与表达式

JavaScript 主要有**算术运算符、比较运算符、逻辑运算符**等。

3.1 算术运算符

运算符	作用	示例
+	加法	5 + 3 = 8
-	减法	10 - 5 = 5
*	乘法	4 * 2 = 8
/	除法	8 / 2 = 4
%	取余	10 % 3 = 1
**	幂运算	2 ** 3 = 8

```
let a = 10;
let b = 3;
console.log(a + b);  // 输出 13
console.log(a % b);  // 输出 1
```

3.2 比较运算符

运算符	作用	示例
==	等于（值相等）	"5" == 5 ✅
===	**全等（值和类型相等）**	"5" === 5 ❌
!=	不等于	10 != 5 ✅
<	小于	5 < 10 ✅
>	大于	20 > 15 ✅

```
console.log(5 == "5");  // true
console.log(5 === "5"); // false
```

✅ **推荐使用 ===，避免类型转换问题**

3.3 逻辑运算符

运算符	作用	示例
&&	**与（AND）**	true && false ❌
||	**或（OR）**	true && false ✅
!	**非（NOT）**	!true ❌

```
let x = 5;
let y = 10;
console.log(x > 0 && y > 5); // true
console.log(x < 0 || y > 5); // true
console.log(!(x > 0));       // false
```

---

4. 条件语句（if-else、switch）

4.1 if-else 语句

```
let score = 85;

if (score >= 90) {
    console.log("优秀");
} else if (score >= 60) {
    console.log("及格");
} else {
    console.log("不及格");
}

```

✅ **if-else 用于判断不同情况**

4.2 switch 语句


```
let fruit = "apple";

switch (fruit) {
    case "apple":
        console.log("苹果");
        break;
    case "banana":
        console.log("香蕉");
        break;
    default:
        console.log("未知水果");
}

```

✅ **switch 适用于多选一的情况**

---

5. 循环语句（for、while、do-while）

5.1 for 循环

```
for (let i = 0; i < 5; i++) {
    console.log("循环次数：" + i);
}

```

✅ **适用于已知循环次数的情况**

5.2 while 循环

```
let i = 0;
while (i < 5) {
    console.log("while循环：" + i);
    i++;
}
```

✅ **适用于循环次数不确定的情况**

5.3 do-while 循环

```
let i = 0;
do {
    console.log("do-while 循环：" + i);
    i++;
} while (i < 5);
```

✅ **即使条件不满足，也会至少执行一次！**

---

6. 课后练习

6.1 实践任务

1. 创建一个 **if-else 语句，判断输入的年龄是否成年**
2. 编写 **for 循环，输出 1 到 10**
3. 用 **switch 语句实现简单的计算器**
4. 使用 **while 计算 1+2+3+...+100 的和**

---

7. 总结

✅ **JavaScript 可内联、内部、外部引入**
✅ **推荐使用 let 和 const 声明变量**
✅ **逻辑运算符 &&、||、！处理布尔逻辑**
✅ **if-else 处理条件，switch 处理多选一**
✅ **for、while 适用于不同场景的循环**

下一步：学习 **JavaScript 函数与作用域！**

第九章：函数与作用域

1. 函数定义与调用

1.1 什么是函数?

函数（Function） 是一组**可复用的代码块**，用于执行特定任务，提高代码复用性。

✓ **函数的作用**

- 封装代码逻辑，避免重复代码
- 提高代码可读性
- 降低维护成本

1.2 定义函数的方式

JavaScript 提供 **三种** 定义函数的方式：

1.2.1 传统函数（function 关键字）

```
function greet() {
    console.log("Hello, JavaScript!");
}
greet(); // 调用函数

```

1.2.2 函数表达式（匿名函数）

```
let greet = function() {
    console.log("Hello, JavaScript!");
};
greet();
```

✓ **区别：函数表达式不会被提升（hoisting）**

1.2.3 箭头函数（ES6）


```
const greet = () => {
    console.log("Hello, JavaScript!");
};
greet();
```

✓ **箭头函数语法更简洁（后面详细讲解）**

---

2. 参数与返回值

2.1 传递参数

函数可以接收参数，用于动态计算结果。

```
function add(a, b) {
    console.log(a + b);
}
add(5, 3); // 输出 8

```

2.2 返回值

return 语句可返回计算结果。

```
function multiply(a, b) {
    return a * b;
}
let result = multiply(4, 5);
console.log(result); // 输出 20

```

2.3 默认参数

ES6 允许给参数指定默认值：

```
function greet(name = "Guest") {
    console.log("Hello, " + name);
}
greet();        // 输出 Hello, Guest
greet("Alice"); // 输出 Hello, Alice

```

✓ **适用于未传递参数的情况**

---

3. 作用域与闭包

3.1 作用域

作用域（Scope） 指**变量的可见范围**，分为：

类型	说明
全局作用域	在任何地方都能访问
函数作用域	变量仅在函数内部可用
块级作用域	仅在 {} 内部有效（let、const）

```
let globalVar = "我是全局变量";

function test() {
    let localVar = "我是函数内变量";
    console.log(globalVar); // ✓ 可访问
}
test();
console.log(localVar); // ✕ 报错（超出作用域）
```

✓ **全局变量在任何地方都能访问，而局部变量仅限于函数内**

3.2 闭包（Closure）

闭包是指：函数内部可以访问外部函数的变量，即使外部函数已执行完毕。


```
function outer() {
    let count = 0;
    return function inner() {
        count++;
        console.log(count);
    };
}
let counter = outer();
counter(); // 输出 1
counter(); // 输出 2

```

✓ **闭包特性**

1. 能访问外部函数变量
2. 变量不会被销毁
3. 适用于计数器、数据缓存等场景

---

4. 箭头函数

4.1 箭头函数的语法

ES6 引入 **箭头函数（Arrow Function）**，语法更简洁。

```
// 传统函数
function add(a, b) {
    return a + b;
}

// 箭头函数
const add = (a, b) => a + b;

console.log(add(5, 3)); // 输出 8

```

✓ **箭头函数特点**

1. 省略 **function 关键字**

2. 参数只有一个时，括号可省略

3. 只有一行返回值时，**return 可省略**

4.2 this 指向问题

箭头函数不会创建自己的 this，而是继承外部作用域的 this。

示例 1：普通函数的 this

```
const obj = {
    name: "Alice",
    greet: function() {
        console.log(this.name);
    }
};
obj.greet(); // 输出 Alice

```

示例 2：箭头函数的 this

```
const obj = {
    name: "Alice",
    greet: () => {
        console.log(this.name);
    }
};
obj.greet(); // 输出 undefined（`this` 不是 `obj`）

```

✅ **箭头函数适合 setTimeout 等回调函数**

```
setTimeout(() => {
    console.log("1 秒后执行");
}, 1000);

```

5. 课后练习

5.1 实践任务

1. 创建一个函数 **sum，计算两个数的和，并返回结果**
2. 使用函数表达式定义一个 **square，计算平方**
3. 写一个 **factorial(n) 递归函数，计算 n!**
4. 使用闭包创建一个计数器函数
5. 用箭头函数改写 **sum 和 square**

6. 总结

✅ **JavaScript 函数支持三种定义方式**（普通、表达式、箭头）
✅ **return 语句可返回计算结果**
✅ **作用域决定变量可见范围，闭包可保持变量状态**
✅ **箭头函数语法简洁，但 this 继承外部作用域**

下一步：学习 **JavaScript 进阶（数组、对象、异步编程）！**

第十章：JavaScript 进阶

1. 数组与对象操作

1.1 数组（Array）

数组是一种**有序的数据集合**，用于存储多个值。

```
let fruits = ["苹果", "香蕉", "橙子"];
console.log(fruits[0]); // 输出 "苹果"
```

✅ **数组索引从 0 开始**

1.2 常见数组方法

方法	作用
push()	**添加** 元素到数组末尾
pop()	**删除** 最后一个元素
shift()	**删除** 第一个元素
unshift()	**添加** 元素到数组开头
splice()	**删除/替换/插入** 元素
map()	**遍历数组并返回新数组**
filter()	**筛选符合条件的元素**
reduce()	**数组累加计算**

```
let numbers = [1, 2, 3, 4, 5];
let squared = numbers.map(num => num * num);
console.log(squared); // [1, 4, 9, 16, 25]
```

✅ **map() 适用于转换数组**

1.3 对象（Object）

对象是**键值对的集合**，用于存储复杂数据。

```
let person = {
    name: "张三",
    age: 25,
    greet: function() {
        console.log("你好，我叫 " + this.name);
    }
};
person.greet(); // 输出 "你好，我叫 张三"

```

✓ **对象属性可包含函数（方法）**

1.4 遍历对象

```
for (let key in person) {
    console.log(key + ": " + person[key]);
}
```

✓ **for...in 遍历对象属性**

---

2. 异步编程

JavaScript 是**单线程**的，异步编程用于执行 **耗时任务（网络请求、定时器）**。

2.1 setTimeout()

setTimeout() **延迟执行代码。**


```
console.log("开始");
setTimeout(() => {
    console.log("延迟 2 秒后执行");
}, 2000);
console.log("结束");

```

✓ **不会阻塞后续代码执行**

2.2 Promise

Promise 解决了回调地狱问题，提供**then、catch、finally** 处理异步操作。

```
let fetchData = new Promise((resolve, reject) => {
    setTimeout(() => {
        let success = true;
        if (success) resolve("数据加载成功");
        else reject("加载失败");
    }, 2000);
});

fetchData
    .then(data => console.log(data))
    .catch(error => console.log(error));
```

✓ **resolve 触发 then()，reject 触发 catch()**

2.3 async/await

async/await 是 **Promise 的语法糖**，让异步代码更像同步代码。

```
async function fetchData() {
    try {
        let response = await new Promise(resolve => setTimeout(() => resolve("数据加载成功"), 2000));
        console.log(response);
    } catch (error) {
        console.log(error);
    }
}
fetchData();

```

✓ **await 等待 Promise 结果，代码更清晰**

---

3. AJAX 与 Fetch API

3.1 什么是 AJAX?

AJAX（Asynchronous JavaScript and XML）用于 **在不刷新页面的情况下与服务器通信。**

✓ **主要用途**

- 获取数据（如 JSON）
- 发送表单
- 动态更新页面

3.2 fetch() 请求数据

fetch() 是 ES6 提供的 **现代 AJAX 方式**，替代 XMLHttpRequest。

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
    .then(response => response.json()) // 解析 JSON
    .then(data => console.log(data))
    .catch(error => console.log("请求失败", error));

```



```
// 要发送的数据
const postData = {
    title: "foo",
    body: "bar",
    userId: 1,
};

// 发起 POST 请求
fetch("https://jsonplaceholder.typicode.com/posts", {
    method: "POST", // 请求方法
    headers: {
        "Content-Type": "application/json", // 请求头，指定发送的数据类型为 JSON
    },
    body: JSON.stringify(postData), // 请求体，将数据转换为 JSON 字符串
})
    .then((response) => response.json()) // 解析响应为 JSON
    .then((data) => console.log("请求成功:", data)) // 处理响应数据
    .catch((error) => console.log("请求失败:", error)); // 捕获错误
```

代码说明:

✓ **比 XMLHttpRequest 更简单**

```
// 创建一个新的 XMLHttpRequest 对象
var xhr = new XMLHttpRequest();

// 配置请求类型、URL 和异步标志
xhr.open("GET", "https://jsonplaceholder.typicode.com/posts/1", true);

// 设置请求完成时的回调函数
xhr.onload = function () {
    // 检查请求是否成功（状态码 200 表示成功）
    if (xhr.status >= 200 && xhr.status < 300) {
        // 解析响应数据为 JSON
        var data = JSON.parse(xhr.responseText);
        console.log(data);
    } else {
        // 请求失败时输出错误信息
        console.log("请求失败", xhr.statusText);
    }
};

// 设置请求出错时的回调函数
xhr.onerror = function () {
    console.log("请求失败", xhr.statusText);
};

// 发送请求
xhr.send();
```

代码说明：

3.3 async/await 结合 fetch()


```
async function getData() {
    const url = "https://jsonplaceholder.typicode.com/posts/1";
    try {
        let response = await fetch(url);
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.log("请求失败", error);
    }
}
getData();

```

✅ **更易读、易维护**

Promise 版本

```
function getData() {
    fetch("https://jsonplaceholder.typicode.com/posts/1")
        .then(response => {
            if (!response.ok) {
                throw new Error("网络响应不正常");
            }
            return response.json();
        })
        .then(data => {
            console.log(data);
        })
        .catch(error => {
            console.log("请求失败", error);
        });
}

getData();
```

解释：

---

4. ES6+ 新特性

4.1 模板字符串（Template Literals）

ES6 允许使用反引号 ` 创建 **多行字符串**，并用 ${} 插入变量。

```
let name = "张三";
let message = `你好，${name}! 欢迎学习 JavaScript。`;
console.log(message);
```

✅ **避免字符串拼接的麻烦**

4.2 解构赋值

快速提取数组和对象的值。

```
let [a, b] = [10, 20];
console.log(a, b); // 输出 10 20

let person = { name: "李四", age: 30 };
let { name, age } = person;
console.log(name, age); // 输出 "李四" 30

```

✅ **代码更简洁**

4.3 扩展运算符（...）

用于数组合并、克隆对象等。


```
let arr1 = [1, 2, 3];
let arr2 = [...arr1, 4, 5];
console.log(arr2); // [1, 2, 3, 4, 5]

let obj1 = { a: 1, b: 2 };
let obj2 = { ...obj1, c: 3 };
console.log(obj2); // { a: 1, b: 2, c: 3 }

```

✅ **避免 concat() 和 Object.assign() 的冗长写法**

4.4 模块化（import/export）

ES6 支持模块化开发，可拆分代码，提高可维护性。

导出模块

```
// math.js
export function add(a, b) {
    return a + b;
}
export const PI = 3.14;

```

导入模块

```
// main.js
import { add, PI } from './math.js';
console.log(add(2, 3)); // 5
console.log(PI); // 3.14

```

✅ **避免全局变量污染，适用于大型项目**

---

5. 课后练习

5.1 实践任务

1. 创建数组并使用 **map() 计算每个元素的平方**
2. 使用 **Promise 模拟 2 秒后返回数据**
3. 用 **fetch() 获取 https://jsonplaceholder.typicode.com/posts/1 并打印标题**
4. 用 **async/await 发送 AJAX 请求**
5. 使用解构赋值获取对象 **name 和 age**

---

6. 总结

✅ **数组方法 map()、filter()、reduce() 操作数据**
✅ **Promise 解决异步回调问题，async/await 让代码更直观**
✅ **fetch() 现代化 AJAX 请求，支持 JSON**
✅ **ES6+ 提供模板字符串、解构赋值、扩展运算符**
✅ **模块化 import/export 让代码更结构化**

下一步：学习 **DOM 操作，控制网页内容！**

第十一章：DOM 操作

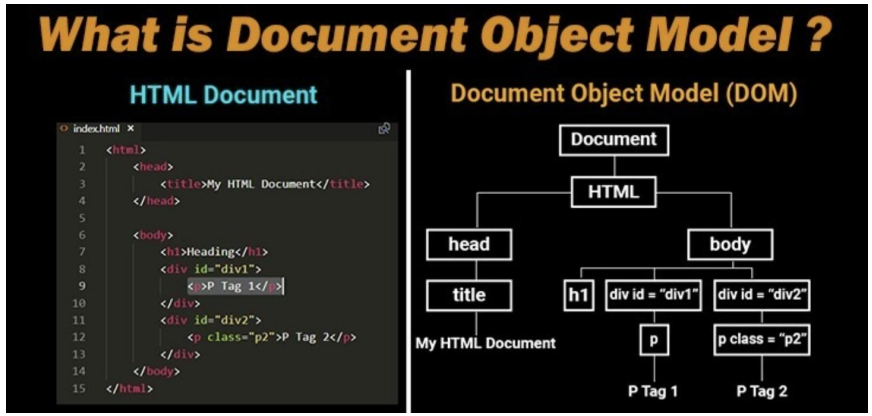
1. DOM 简介与节点类型

1.1 什么是 DOM?

DOM（Document Object Model，文档对象模型） 是浏览器解析 HTML 生成的 **树状结构**，用于 **JavaScript 操作网页内容。**

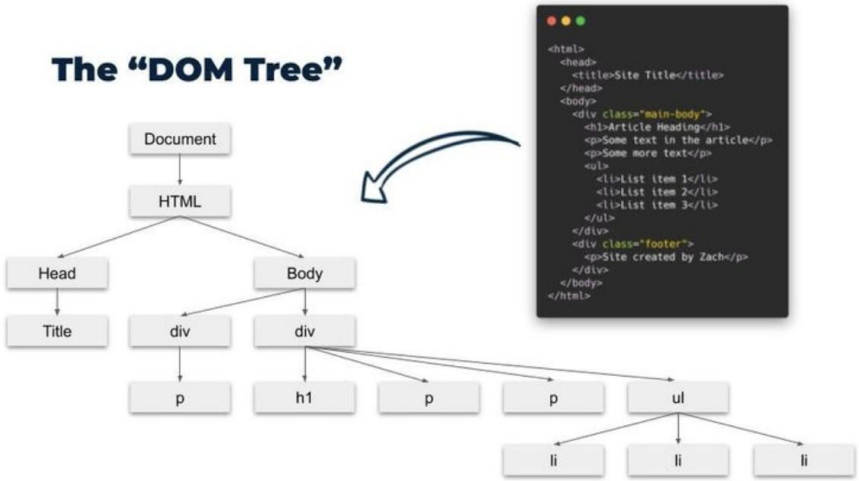
✅ **作用**：

- 查找 HTML 元素
- 修改内容
- 更改样式
- 处理用户交互（事件监听）



1.2 DOM 结构

HTML 文档在 JavaScript 中被解析为 **DOM 树**：




```html
<html>
  <head>
    <title>文档标题</title>
  </head>
  <body>
    <h1>标题</h1>
    <p>段落文本</p>
  </body>
</html>

```

➡ 转换成 DOM 结构：

```
document
  ├── <html>
  │    ├── <head>
  │    │    ├── <title>文档标题</title>
  │    ├── <body>
  │         ├── <h1>标题</h1>
  │         ├── <p>段落文本</p>

```

1.3 常见节点类型

节点类型	说明	示例
文档节点	整个 HTML 文档	document
元素节点	HTML 标签	<p> <h1>
文本节点	标签内文本	文本内容
属性节点	标签的属性	class="box"

```javascript
console.log(document.body); // 获取 `<body>` 元素
console.log(document.title); // 获取 `<title>` 文本
```

---

2. 选择元素

2.1 getElementById()

按 id 选择**唯一**元素：

```html
<p id="text">Hello, JavaScript!</p>
<script>
    let p = document.getElementById("text");
    console.log(p.innerText); // 输出 Hello, JavaScript!
</script>

```

✅ **适用于唯一 ID 选择**

2.2 querySelector()

按 **CSS 选择器** 选择**第一个**匹配的元素：

```javascript
let firstParagraph = document.querySelector("p");
let header = document.querySelector("#header"); // 选择 id="header"
let button = document.querySelector(".btn"); // 选择 class="btn"
```

✅ **适用于单个匹配**

2.3 querySelectorAll()

获取**所有匹配的元素（NodeList）**：

```html
<p>第一段</p>
<p>第二段</p>
<script>
    let paragraphs = document.querySelectorAll("p");
    console.log(paragraphs.length); // 输出 2
</script>
```

✅ **适用于多个匹配（类数组结构）**

---

3. 操作元素

3.1 修改内容

方法	作用
innerText	修改文本内容（不解析 HTML）
innerHTML	修改 HTML 内容（解析 HTML）
textContent	获取/修改文本（推荐使用）

```html
<p id="demo">原文本</p>
<script>
    let demo = document.getElementById("demo");
    demo.innerText = "新文本"; // 纯文本
    demo.innerHTML = "<strong>加粗文本</strong>"; // 解析 HTML
</script>
```

✅ **innerText 更安全，不解析 HTML**

3.2 修改样式

方法	作用
element.style.property	修改单个样式
classList.add()	添加类
classList.remove()	移除类
classList.toggle()	切换类

```html
<p id="styled-text">改变样式</p>
<script>
    let text = document.getElementById("styled-text");
    text.style.color = "red"; // 改变颜色
    text.classList.add("highlight"); // 添加类
</script>
```

✓ **推荐使用 classList，更易维护**

3.3 修改属性

方法	作用
element.setAttribute(name, value)	设置属性
element.getAttribute(name)	获取属性值
element.removeAttribute(name)	移除属性

```
<img id="image" src="old.jpg">
<script>
    let img = document.getElementById("image");
    img.setAttribute("src", "new.jpg"); // 修改图片
</script>
```

✓ **适用于修改 href、src、alt 等属性**

---

4. 事件处理

4.1 监听事件

事件是用户在页面上的**交互操作**（点击、输入、悬停等）。

✓ **常见事件**

事件	说明
click	点击
mouseover	鼠标悬停
keydown	按键按下
input	输入框变化

4.2 addEventListener()

最推荐的事件绑定方式：


```
<button id="btn">点击我</button>
<script>
    let btn = document.getElementById("btn");
    btn.addEventListener("click", function() {
        alert("按钮被点击了！");
    });
</script>

```

✓ **优点**

1. 可同时绑定多个事件
2. 可动态添加和移除事件

4.3 事件对象（event）

事件处理函数**可接收 event 对象**，获取事件详细信息。

```
<input id="input-box" type="text">
<script>
    let inputBox = document.getElementById("input-box");
    inputBox.addEventListener("input", function(event) {
        console.log("当前值：" + event.target.value);
    });
</script>

```

✓ **适用于表单输入监控**

4.4 事件冒泡与捕获

事件传播有 **冒泡（默认）** 和 **捕获** 两种模式：

```html
<div id="parent">
    <button id="child">点击按钮</button>
</div>
<script>
    document.getElementById("parent").addEventListener("click", () => {
        console.log("父元素被点击");
    });

    document.getElementById("child").addEventListener("click", (event) => {
        console.log("子元素被点击");
        event.stopPropagation(); // 阻止冒泡
    });
</script>
```

✅ **stopPropagation() 阻止事件继续传播**

---

5. 课后练习

5.1 实践任务

1. 使用 **getElementById 选择一个 <p> 元素，并修改其文本**
2. 使用 **querySelector 选择 .btn 按钮，并改变背景色**
3. 使用 **setAttribute 修改 <img> 图片地址**
4. 给 **<button> 绑定 click 事件，点击后弹出提示**
5. 创建一个输入框，输入内容时实时显示在 **<p> 中**

---

6. 总结

✅ **DOM 是 HTML 的 JavaScript 表现形式**
✅ **getElementById、querySelector 选择元素**
✅ **innerText 修改文本，style 修改样式**
✅ **addEventListener 绑定事件，避免 onclick**
✅ **事件对象 event 提供更多信息**

第十二章：Web 存储与 Cookie

1. Web 存储概述

在 JavaScript 中，有 3 种主要的 Web 存储方式：

存储方式	特点	适用场景
localStorage	**长期存储**，数据不会随页面关闭而丢失	存储用户偏好、持久化数据（如 JWT 令牌、用户设置等）
sessionStorage	**会话级存储**，页面关闭后数据清除	临时存储数据（如表单数据、当前登录状态）
Cookie	**可设置过期时间**，支持服务器读取	用于存储登录凭证、跨页面数据传输

---

2. localStorage（本地存储）

- 数据存储在浏览器本地，不会随页面关闭而丢失。
- 存储容量：最大 5MB，适合存储 **非敏感** 持久化数据。

操作 localStorage

```javascript
// 存储数据
localStorage.setItem("username", "张三");
// 读取数据
console.log(localStorage.getItem("username")); // 输出 "张三"
// 删除单个数据
localStorage.removeItem("username");
// 清空所有数据
localStorage.clear();
```

✅ **应用场景**：

- 持久化存储用户偏好（如深色模式）
- 存储 JWT 令牌（但建议使用 HttpOnly Cookie 代替）
- 存储前端缓存数据（如搜索历史）

---

3. sessionStorage（会话存储）

- 数据仅在当前页面会话中有效，页面关闭后数据清除。

- 适合存储临时数据（如用户输入的表单数据）。

操作 sessionStorage

```
// 存储数据
sessionStorage.setItem("sessionKey", "临时数据");
// 读取数据
console.log(sessionStorage.getItem("sessionKey"));
// 删除数据
sessionStorage.removeItem("sessionKey");
// 清空所有 sessionStorage 数据
sessionStorage.clear();
```

✅ **应用场景**：
- 临时存储表单数据，避免页面刷新丢失数据。
- 临时存储搜索结果、筛选条件等会话信息。

---

4. Cookie（用于服务器通信）

- 数据可以在前端 & 服务器之间传输，支持设置过期时间。
- 大小限制：每个 cookie 最大 4KB。
- 可设置 HttpOnly，使 JavaScript 无法访问，提升安全性。

操作 Cookie

```
// 设置 cookie（expires 过期时间，path 适用路径）
document.cookie = "username=张三; expires=Fri, 31 Dec 2025 23:59:59 GMT; path=/";

// 读取 cookie
console.log(document.cookie);

// 删除 cookie（设置过期时间为过去）
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 GMT";

```

✅ **应用场景**：

- 存储登录状态（如 session ID）
- 跨页面共享数据（如用户偏好设置）
- 用于服务器身份认证（如 token）

---

5. localStorage vs sessionStorage vs Cookie 对比

特性	localStorage	sessionStorage	Cookie
生命周期	**永久存储**（手动删除）	**会话存储**（页面关闭即清除）	**可自定义过期时间**
存储大小	约 **5MB**	约 **5MB**	仅 **4KB**
数据作用范围	**整个网站**（同源共享）	**当前页面会话**	**同源共享，可与服务器交互**
是否随请求发送	✕ **不会**	✕ **不会**	✓ **自动随 HTTP 请求发送**
适用场景	持久化数据（如用户偏好）	临时数据（如搜索结果）	服务器身份验证、跨页面数据

课后练习

✓ **练习 1**：使用 localStorage 存储深色模式开关状态，并在页面刷新后保持用户选择的模式。

✓ **练习 2**：使用 sessionStorage 记录当前页面的表单输入，刷新后仍能恢复数据。

✓ **练习 3**：使用 JavaScript 操作 Cookie，存储一个用户登录状态，并在控制台读取它。

总结

✓ **localStorage** 适合长期存储数据，不会随页面关闭丢失。

✓ **sessionStorage** 适合临时存储数据，页面关闭后自动清除。

✓ **Cookie** 适用于跨页面存储 & 服务器通信，但大小有限。

建议更新你的 JavaScript 大纲，加入这个新章节！

第十三章：开发工具

1. 浏览器开发者工具（Chrome DevTools）

1.1 什么是开发者工具?

浏览器开发者工具（DevTools）是 **前端调试、优化、分析网页性能的必备工具**，主要用于：

- 调试 JavaScript 代码
- 检查 HTML 结构
- 修改 CSS 样式
- 分析网页加载性能

✓ **推荐使用**：F12 或 Ctrl + Shift + I 打开 DevTools

1.2 DevTools 常用面板

面板	作用
Elements	查看/修改 HTML 结构和 CSS 样式
Console	输出日志、调试 JavaScript
Network	监控 HTTP 请求、分析加载时间
Sources	设置断点、调试 JS 代码
Performance	分析网页性能
Application	管理存储（LocalStorage、Cookies）

1.3 Console 调试

查看错误

```
console.error("这是一个错误信息");
```

警告信息

```
console.warn("警告信息");

```

表格格式化数据

```
console.table([{ name: "张三", age: 25 }, { name: "李四", age: 30 }]);

```

✅ **开发者工具让调试 JavaScript 更高效！**

---

2. 代码编辑器（VS Code 使用技巧）

2.1 为什么选择 VS Code?

VS Code（Visual Studio Code）是目前**最流行的前端开发工具**，提供：

- 轻量级但功能强大
- 丰富的插件生态
- 内置终端
- Git 集成

2.2 常用快捷键

快捷键	作用
Ctrl + P	快速打开文件
Ctrl + Shift + P	命令面板
Alt + Shift + ↓	复制当前行
Ctrl + D	选中多个相同文本
Ctrl + /	添加/取消注释
Ctrl + Shift + K	删除当前行

2.3 必装插件

插件	作用
ESLint	代码规范检查
Prettier	自动格式化代码
Live Server	本地服务器，自动刷新
Debugger for Chrome	浏览器调试
Path Intellisense	智能路径补全

✅ **安装这些插件，提高编码效率！**

---

3. 版本控制（Git 基础）

3.1 什么是 Git?

Git 是**最流行的版本控制系统**，用于：

- 代码管理
- 多人协作
- 版本回滚

✅ **Git 是现代开发必备工具**

3.2 Git 常用命令

命令	作用
git init	初始化 Git 仓库
git clone	克隆远程仓库
git add .	添加所有更改到暂存区
git commit -m "说明"	提交更改
git push origin main	推送代码到远程仓库
git pull origin main	拉取远程最新代码
git log	查看提交历史
git status	查看当前状态

✅ **推荐使用 GitHub / GitLab 托管代码！**

第十四章：构建工具

1. 包管理工具（npm、yarn）

1.1 什么是包管理器?

包管理器用于**安装、管理 JavaScript 依赖**，主流工具：

- npm（Node.js 自带）
- yarn（Facebook 开发，更快）

1.2 常用命令

命令	作用
npm init	初始化项目
npm install package	安装包
npm uninstall package	卸载包
npm update	更新依赖

✅ **推荐使用 yarn 提高速度**

---

2. 模块打包工具（Webpack）

2.1 什么是 Webpack?

Webpack 是**前端打包工具**，用于：

- 合并 JavaScript 文件
- 压缩 CSS 和 JS
- 加载图片、字体等静态资源

✅ **适用于大型前端项目**

2.2 Webpack 基本配置

```
module.exports = {
    entry: './src/index.js', // 入口文件
    output: {
        filename: 'bundle.js', // 输出文件
        path: __dirname + '/dist'
    },
    module: {
        rules: [
            { test: /\.css$/, use: ['style-loader', 'css-loader'] }
        ]
    }
};

```

✓ **安装 webpack 并运行**

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

```

---

3. 自动化工具（Gulp、Grunt）

3.1 Gulp

Gulp 是**前端自动化构建工具**，用于：

- 压缩 CSS/JS
- 自动刷新浏览器
- 图片优化

✓ **Gulp 配置示例**


```
const { src, dest, watch, series } = require("gulp");
const cssnano = require("gulp-cssnano");

function minifyCSS() {
    return src("src/*.css")
        .pipe(cssnano())
        .pipe(dest("dist"));
}

exports.default = series(minifyCSS);

```

✓ **运行 Gulp**

```
npm install gulp gulp-cssnano --save-dev
npx gulp

```

总结

✓ **开发工具：VS Code、DevTools、Git 提高效率**

✓ **调试技巧：debugger、断点、console.log 解决问题**

✓ **优化：减少 HTTP 请求、懒加载、缓存**

✓ **构建工具：Webpack、Gulp 自动化开发**

下一步：进入 **第五篇：实战与创新 - 从零到一的蜕变！**

第十五章：调试与优化

1. JavaScript 调试技巧

1.1 使用 debugger

```
function test() {
    let a = 10;
    debugger; // 代码在此暂停，可查看变量值
    console.log(a);
}
test();

```

✓ **配合 DevTools 进行代码调试**

1.2 设置断点

1. 打开 DevTools（F12）
2. 进入 **Sources 面板**
3. 找到 **JS 文件，单击行号设置断点**
4. 执行代码，查看变量值

✓ **更精准地调试代码**

---

2. 性能优化

2.1 减少 HTTP 请求

合并 CSS 和 JS 文件，减少 HTTP 请求次数。

2.2 资源压缩

资源	工具
JavaScript	UglifyJS
CSS	CSSNano
图片	TinyPNG

✓ **减少文件大小，提高加载速度**

2.3 懒加载

按需加载资源，减少首屏加载时间

```
<img src="placeholder.jpg" data-src="real-image.jpg" class="lazy">
<script>
    document.addEventListener("DOMContentLoaded", () => {
        document.querySelectorAll(".lazy").forEach(img => {
            img.src = img.dataset.src;
        });
    });
</script>

```

✓ **提高网页首屏加载速度**

2.4 使用缓存

- 启用浏览器缓存
- 使用 **localStorage 存储数据**
- CDN 加速静态资源

---

3. 跨浏览器兼容性

3.1 caniuse.com 查询兼容性

使用 caniuse.com 检查 CSS/JS 兼容性。

3.2 @supports 处理不兼容情况

```
@supports (display: grid) {
    .container {
        display: grid;
    }
}

```

✓ **让代码适应不同浏览器**

第十六章：Webpack 基础配置

一、为什么需要 Webpack?

1. 前端开发的痛点

- 模块化问题：浏览器原生不支持 import/export 等 ES6 模块语法
- 资源依赖管理：手动处理 CSS、图片、字体等非 JS 资源
- 性能优化需求：代码压缩、按需加载、Tree Shaking 等
- 开发效率：实时刷新、热更新（HMR）、Source Map

2. Webpack 的核心作用

将**所有资源（JS、CSS、图片等）视为模块**，通过依赖分析打包成浏览器可识别的静态文件。

二、快速上手 Webpack

1. 安装

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

这两行命令需要在**项目目录**下执行。

原因：

npm init -y：

- 该命令会在当前目录下初始化一个 npm 项目。
- 自动生成一个 package.json 文件，用于记录项目的依赖、版本等信息。
- 如果不在项目目录下执行，可能会在错误的位置生成 package.json，导致项目配置混乱。

npm install webpack webpack-cli --save-dev：

- 该命令会在当前目录下的 node_modules 文件夹中安装 webpack 和 webpack-cli。
- 同时，会将它们作为开发依赖（devDependencies）记录到 package.json 文件中。
- 如果不在项目目录下执行，安装的包不会被正确关联到你的项目，后续使用时可能会找不到依赖。

正确操作步骤：

1. 进入你的项目目录（包含 src 文件夹和其他项目文件的目录）。
2. 执行以下命令：

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

3. 确保在项目目录下创建 webpack.config.js 文件并进行配置。
这样可以确保项目的依赖管理和配置是正确的。

2. 基础配置

创建 webpack.config.js：

```
const path = require('path');

module.exports = {
  entry: './src/index.js',        // 入口文件
  output: {
    filename: 'bundle.js',       // 输出文件名
    path: path.resolve(__dirname, 'dist') // 输出目录
  },
  mode: 'production' // 模式：development 或 production
};
```

3. 运行打包

```
npx webpack
```

三、核心概念

1. Entry（入口）

定义依赖分析的起点：


```
entry: {
  main: './src/index.js',
  vendor: './src/vendor.js'
}
```

2. Output（出口）

指定打包文件的位置和命名规则：

```
output: {
  filename: '[name].[contenthash].js', // 使用哈希避免缓存
  path: path.resolve(__dirname, 'dist')
}
```

3. Loader（加载器）

处理非 JS 文件（Webpack 默认只理解 JS）：

```
module: {
  rules: [
    {
      test: /\.css$/,        // 匹配 .css 文件
      use: ['style-loader', 'css-loader'] // 从右向左执行
    },
    {
      test: /\.(png|svg|jpg)$/,
      type: 'asset/resource' // Webpack 5 内置资源处理
    }
  ]
}
```

4. Plugin（插件）

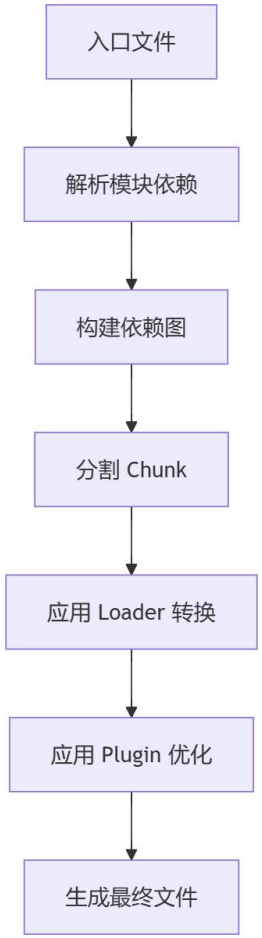
扩展 Webpack 功能（如 HTML 模板生成）：

```
const HtmlWebpackPlugin = require('html-webpack-plugin');

p: ugins: [
  new
    HtmlWebpackPlugin({ template
    : './src/index.html'
  })
]
```

四、Webpack 工作原理

1. 整体流程




```
graph TD
  A[入口文件] --> B[解析模块依赖]
  B --> C[构建依赖图]
  C --> D[分割 Chunk]
  D --> E[应用 Loader 转换]
  E --> F[应用 Plugin 优化]
  F --> G[生成最终文件]
```

2. 关键步骤详解

(1) 模块解析

- 从 entry 开始，递归分析 import/require 语句
- 生成 **AST（抽象语法树）** 分析依赖关系

(2) 依赖图构建

- 将所有模块及其依赖关系组合成 **依赖图（Dependency Graph）**

(3) Chunk 生成（详见附录：深入理解Chunk）

- 根据配置（如 splitChunks）将代码分割成多个 Chunk
- 常见的 Chunk 类型：
  - Initial Chunk（入口直接依赖）
  - Async Chunk（动态导入的模块）
  - Runtime Chunk（Webpack 运行时代码）

(4) 输出文件

- 将 Chunk 转换为最终文件
- 应用 Loader 转换（如 SCSS → CSS → JS 内联）
- 执行 Plugin 的优化逻辑（如压缩、添加 Hash）

五、配置实战

1. 开发服务器配置

```
devServer: {
  static: './dist',
  hot: true,       // 热更新
  port: 8080,
  open: true       // 自动打开浏览器
}
```

2. 处理样式


```
// 安装：npm install style-loader css-loader sass-loader sass --save-dev
{
  test: /\.scss$/,
  use: ['style-loader', 'css-loader', 'sass-loader']
}
```

3. Babel 转译

```
// 安装：npm install babel-loader @babel/core @babel/preset-env --save-dev
{
  test: /\.js$/,
  exclude: /node_modules/,
  use: {
    loader: 'babel-loader',
    options: {
      presets: ['@babel/preset-env']
    }
  }
}
```

六、高级特性

1. 代码分割（Code Splitting）

```
// 动态导入（自动生成 Async Chunk）
import(/* webpackChunkName: "lodash" */ 'lodash').then(...);
```

2. Tree Shaking

- 自动删除未使用的代码（需配合 ES6 模块语法）
- 在 package.json 中添加：

```
"sideEffects": false
```

七、常见问题

1. 路径配置错误

```
// 错误：找不到模块
import MyComponent from './components/MyComponent';
// 正确：明确文件扩展名
import MyComponent from './components/MyComponent.js';
```

2. Loader 执行顺序

```
// 从后向前执行（先执行 sass-loader，最后 style-loader）
use: ['style-loader', 'css-loader', 'sass-loader']
```

3. 旧浏览器兼容

使用 @babel/preset-env + core-js：

```
// .babelrc
{
  "presets": [
    ["@babel/preset-env", {
      "useBuiltIns": "usage",
      "corejs": 3
    }]
  ]
}
```

八、总结

✓ **Webpack 核心能力**：模块打包、资源转换、代码优化

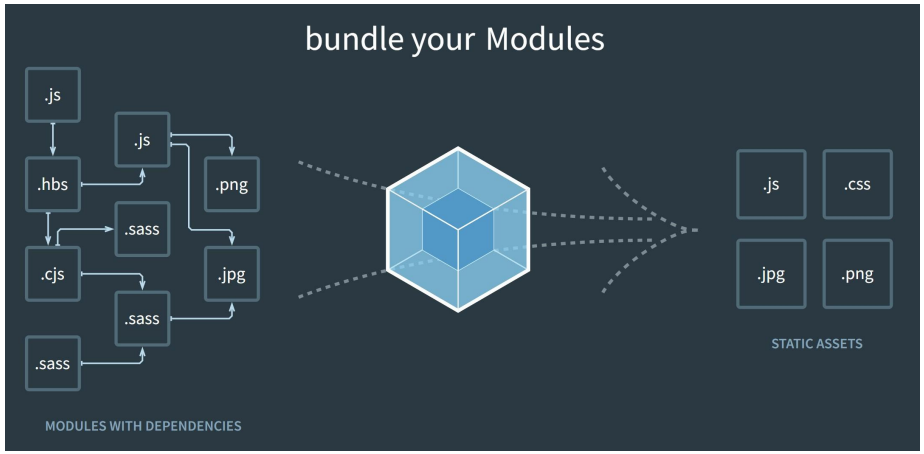
✓ **关键配置**：entry/output/loader/plugin

✓ **工作流程**：依赖分析 → 构建依赖图 → 生成 Chunk → 输出文件

进阶方向：自定义 Loader/Plugin、性能优化、微前端集成

第十七章：Webpack 深度教学指南

注意：学习本章前：请完成综合项目02：购物车功能开发



一、为什么需要 Webpack?

1. 前端开发的演变

- 原始开发模式：手动管理 HTML/CSS/JS 文件，缺乏模块化
- 模块化需求：ES6 的 import/export，但浏览器兼容性差
- 资源依赖管理：非 JS 资源（如图片、CSS）无法直接引入
- 性能优化瓶颈：代码压缩、按需加载、缓存策略难以手动实现

2. Webpack 的核心价值

- 模块化打包：将分散的模块整合为少数文件
- 资源统一处理：通过 Loader 处理 CSS/图片等非 JS 资源
- 开发效率提升：热更新（HMR）、Source Map、DevServer
- 生产优化能力：Tree Shaking、代码分割、压缩

二、核心概念

1. 模块（Module）

- 定义：任何文件（JS、CSS、图片等）均可视为模块
- 特点：模块间通过 import/require 建立依赖关系
- 示例：


```
import styles from './styles.css'; // CSS 模块
import logo from './logo.png';     // 图片模块
```

2. 入口（Entry）

- 作用：指定依赖分析的起点
- 单入口：

```
entry: './src/index.js'
```

- 多入口（多页面应用）：

```
entry: {
  home: './src/home.js',
  about: './src/about.js'
}
```

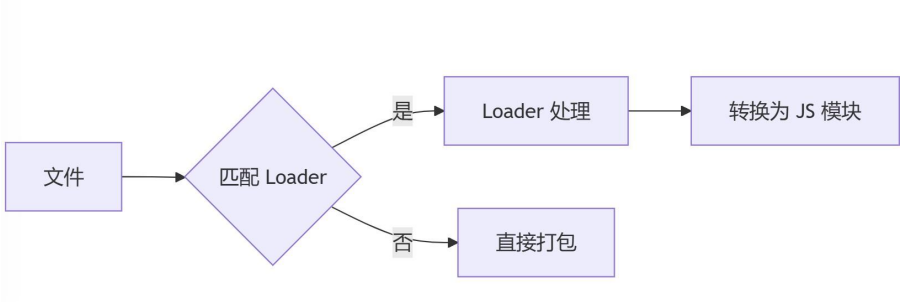
3. 出口（Output）

- 作用：定义打包文件的输出规则
- 关键配置：

```
output: {
  filename: '[name].[contenthash].js', // 动态文件名（入口名 + 哈希）
  path: path.resolve(__dirname, 'dist'),
  publicPath: '/assets/',              // 资源公共路径（CDN 场景）
  clean: true                          // 自动清理旧文件
}
```

4. Loader

- 作用：处理非 JS 模块（Webpack 默认只能处理 JS）
- 工作流程：




```
graph LR
  A[文件] --> B{匹配 Loader}
  B -->|是| C[Loader 处理]
  B -->|否| D[直接打包]
  C --> E[转换为 JS 模块]
```

- 常用 Loader：

Loader	作用	示例配置
css-loader	解析 CSS 文件	{ test: /\.css$/, use: ['style-loader', 'css-loader'] }
babel-loader	转译 ES6+ 代码	{ test: /\.js$/, use: 'babel-loader' }
file-loader	处理文件（如图片）	{ test: /\.png$/, type: 'asset/resource' } (Webpack 5+)

5. 插件（Plugin）

- 作用：扩展 Webpack 功能（如生成 HTML、压缩代码）
- 常用插件：

插件	作用	示例配置
HtmlWebpackPlugin	生成 HTML 文件	new HtmlWebpackPlugin({ template: './src/index.html' })
MiniCssExtractPlugin	提取 CSS 为独立文件	new MiniCssExtractPlugin({ filename: '[name].css' })
CleanWebpackPlugin	清理输出目录	new CleanWebpackPlugin()

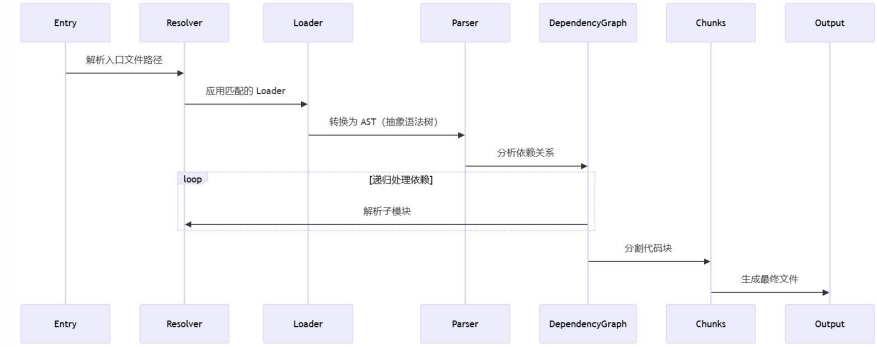
6. 模式（Mode）

- 作用：内置优化策略（开发/生产环境）

```
mode: 'development', // 或 'production'
// 等同于：
moule.exports =
  { optimization: {
    minimize: mode === 'production',
    // 其他生产环境优化（如 Tree Shaking）
  },
  devtool: mode === 'development' ? 'eval-source-map'                    : 'hidden-source-map'
}
```

三、Webpack 工作原理

1. 打包流程总览



```
sequenceDiagram
  participant Entry
  participant Resolver
  participant Loader
  participant Parser
  participant DependencyGraph
  participant Chunks
  participant Output

  Entry->>Resolver: 解析入口文件路径
  Resolver->>Loader: 应用匹配的 Loader
  Loader->>Parser: 转换为 AST（抽象语法树）
  Parser->>DependencyGraph: 分析依赖关系
  loop 递归处理依赖
    DependencyGraph->>Resolver: 解析子模块
  end
  DependencyGraph->>Chunks: 分割代码块
  Chunks->>Output: 生成最终文件
```

2. 关键步骤详解

- 模块解析（Resolution）：
  - 根据 resolve.alias、resolve.extensions 等配置解析文件路径
- 加载与转换（Loading & Transpiling）：
  - 按 module.rules 匹配 Loader，从右向左执行
  - 示例：SCSS → Sass-loader → CSS-loader → Style-loader
- 依赖图构建（Dependency Graph）：
  - 从入口开始，递归分析所有依赖关系
  - 形成包含所有模块及其依赖关系的图结构
- 代码分割（Code Splitting）：
  - 通过 optimization.splitChunks 或动态导入 (import()) 分割代码
  - 生成 Initial Chunk（入口代码）、Async Chunk（懒加载代码）、Runtime Chunk（Webpack 运行时）
- 资源生成（Asset Generation）：
  - 将 Chunk 转换为最终文件（JS、CSS、Source Map 等）
  - 应用 Plugin 的优化逻辑（如压缩、添加 Hash）

四、完整配置示例

1. 开发环境配置 (webpack.dev.js)

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  mode: 'development',
  entry: './src/index.js',
  output: {
    filename: '[name].js',
    path: path.resolve(__dirname, 'dist'),
  },
  devtool: 'eval-source-map',
  devServer: {
    static: './dist',
    hot: true,
    port: 3000,
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: 'babel-loader',
      },
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
  ],
```

```
};
```


2. 生产环境配置 (webpack.prod.js)

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const CssMinimizerPlugin = require('css-minimizer-webpack-plugin');

module.exports = {
  mode: 'production',
  entry: './src/index.js',
  output: {
    filename: '[name].[contenthash].js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  optimization: {
    minimizer: [
      new CssMinimizerPlugin(), // 压缩 CSS
    ],
    splitChunks: {
      chunks: 'all', // 提取公共依赖
    },
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader'], // 提取 CSS
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
    new MiniCssExtractPlugin({
      filename: '[name].[contenthash].css',
    }),
  ],
};
```


五、高级特性与优化

1. Tree Shaking

- 条件：使用 ES6 模块语法 (import/export)，设置 mode: 'production'
- 验证：在 package.json 中标记无副作用文件：

```
{
  "sideEffects": ["*.css", "*.global.js"]
}
```

2. 持久化缓存

- 配置文件名哈希：

```
output: {
  filename: '[name].[contenthash].js'
}
```

- 优化模块 ID：

```
optimization: {
  moduleIds: 'deterministic' // 防止模块 ID 变化导致哈希变动
}
```

3. 懒加载（Lazy Loading）

- 动态导入语法：

```
button.addEventListener('click', () => {
  import('./lazy-module.js').then(module => {
    module.run();
  });
```

```
});
```


六、常见问题与调试

1. 性能分析

- 生成统计文件：

```
npx webpack --profile --json > stats.json
```

- 可视化工具：
  - Webpack Bundle Analyzer
  - Webpack Visualizer

2. 调试配置

- 打印完整配置：

```
console.log(require('util').inspect(config, { depth: null }));
```

- 逐步排查 Loader/Plugin：禁用插件或简化 Loader 规则定位问题

七、延伸学习

1. 自定义 Loader

- 编写一个简单的 Loader (src/loaders/uppercase-loader.js)：

```
module.exports = function(source) {
  return source.toUpperCase(); // 将文本转为大写
};
```

- 配置使用：

```
{
  test: /\.txt$/,
  use: path.resolve(__dirname, 'src/loaders/uppercase-loader.js')
}
```

2. 自定义 Plugin

- 基本结构：

```
class MyPlugin {
  apply(compiler) {
    compiler.hooks.done.tap('MyPlugin', stats => {
      console.log('编译完成! ');
    });
  }
}
```

八、总结

- ✓ **核心能力**：模块打包、资源处理、代码优化
- ✓ **关键配置**：entry、output、loader、plugin、mode
- ✓ **优化方向**：代码分割、缓存策略、Tree Shaking
- **调试技巧**：分析工具、配置打印、逐步排查
- **进阶路径**：自定义 Loader/Plugin、微前端集成、性能调优

通过理解 Webpack 的底层机制，你可以更高效地配置和优化前端构建流程！

一、基础条件

1. JavaScript 运行环境支持
   - 浏览器：现代浏览器（如 Chrome、Edge、Firefox、Safari）已原生支持大部分 ES6 特性。
     - 检查兼容性：Can I use。
   - Node.js：Node.js 6+ 开始支持部分 ES6 特性，建议使用 Node.js 12+ 或更高版本以获得完整支持。
2. 开发工具支持
   - 代码编辑器/IDE：如 VS Code、WebStorm 等需支持 ES6 语法高亮和提示（现代编辑器默认支持）。

二、兼容旧环境的额外条件

如果代码需在**不支持 ES6 的旧环境**（如 IE11、旧版移动浏览器）中运行，需通过以下工具链处理：

1. 转译器（Transpiler）

- Babel：将 ES6+ 代码转换为 ES5 代码。
  - 安装 Babel 核心包和预设：

```
npm install @babel/core @babel/preset-env --save-dev
```

  - 配置 .babelrc：

```
{
  "presets": ["@babel/preset-env"]
}
```

2. 打包工具（Bundler）

- Webpack/Rollup/Parcel：配合 Babel 实现代码打包和转译。
  - 示例（Webpack + Babel）：
    i. 安装依赖：

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

    ii. 配置 webpack.config.js：

```
module.exports = {
  module:
    { rules: [{
      test: /\.js$/,
      exclude: /node_modules/,
      use: { loader: 'babel-loader' }
    }]
  }
};
```

3. Polyfill

- core-js 或 **@babel/polyfill**：为旧环境提供 ES6+ 新 API（如 Promise、Array.from）。
  - 安装并引入：

```
npm install core-js --save
```

```
import "core-js/stable";
```

三、开发流程条件

1. 模块化语法支持
   - 使用 import/export 模块化语法时，需通过打包工具（Webpack 等）处理，或在浏览器中声明 type="module"：

```
<script type="module" src="app.js"></script>
```

2. 严格模式
   - ES6 模块和 class 默认启用严格模式，需避免使用非严格模式语法（如未声明的变量）。

四、其他注意事项

1. 浏览器特性支持
   - 部分 ES6+ 特性（如 Proxy、Web Workers）可能依赖浏览器 API，需单独测试兼容性。
2. Node.js 的特殊配置
   - 在 Node.js 中使用 ES6 模块需：
     - 将文件扩展名改为 .mjs，或在 package.json 中设置 "type": "module"。
3. TypeScript 支持
   - 如果使用 TypeScript，需配置 tsconfig.json 的 target 为 ES6 或更高：

```
{
  "compilerOptions":
    { "target": "ES6"
  }
}
```

五、总结

- 现代环境：直接使用 ES6，无需额外条件。
- 旧环境：需通过 **Babel + 打包工具 + Polyfill** 实现兼容。
- Node.js：建议使用 Node.js 12+，并通过 package.json 配置模块类型。

合理配置工具链后，ES6 可安全用于生产环境，提升代码可读性和开发效率。

Bootstrap 5 的网格系统是基于 CSS 的 Flexbox 布局来实现的，而不再是之前版本中使用的浮动（float）布局。以下是这句话的详细解释：

1. Flexbox 布局

Flexbox 是 CSS3 引入的一种一维布局模式，全称是 "Flexible Box"，即灵活的盒子布局。它允许容器内的子元素（项目）按照一定的规则进行排列，可以很容易地实现各种复杂的布局，而无需使用浮动（float）或其他复杂的定位技巧。

Flexbox 的主要特点包括：

- 父容器可以定义子元素的排列方向（水平或垂直）。
- 子元素可以灵活地分配空间。
- 子元素可以定义对齐方式（如居中、两端对齐等）。
- 布局方向、对齐方式等都可以通过简单的 CSS 属性进行控制。

2. Bootstrap 5 基于 Flexbox 实现网格系统

Bootstrap 是一个流行的前端框架，提供了快速构建响应式网站的工具和组件。在 Bootstrap 5 中，网格系统是基于 Flexbox 实现的，这意味着：

- 网格布局的实现更加灵活和强大。
- 开发者可以更轻松地创建复杂的响应式布局。
- 不再需要依赖浮动（float）来实现布局，简化了代码。

3. 取代了旧版的浮动布局

在 Bootstrap 4 及之前的版本中，网格系统是基于浮动（float）布局实现的。浮动布局有一些局限性，比如：

- 需要手动清除浮动（clearfloat）。
- 布局的灵活性较低。
- 在响应式设计中，实现某些复杂的布局可能比较困难。

Bootstrap 5 选择用 Flexbox 取代浮动布局，主要是为了：

- 提高布局的灵活性和响应式能力。
- 简化代码，减少开发者的工作量。
- 提供更现代化的布局方式，符合现代前端开发的趋势。

总结

这句话的意思是，Bootstrap 5 的网格系统采用了更现代化的 Flexbox 布局方式，而不是以前的浮动布局。这种方式使得网格系统的使用更加灵活、强大，同时简化了开发过程，提高了响应式设计的能力

HTTP 与 Web 通信

1. HTTP 协议的作用

✓ **客户端（浏览器）和服务器之间的通信标准**

---

2. 请求-响应模型

```
GET /index.html HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0

```

✓ **请求头包含 User-Agent、Accept 等信息**

---

3. RESTful API 简介

方法	作用	示例
GET	获取资源	/users/1
POST	创建资源	/users
PUT	更新资源	/users/1
DELETE	删除资源	/users/1

✓ **JSON 是 API 主要数据格式**

```
{
    "id": 1,
    "name": "张三"
}

```

---

4. HTTPS 与安全性

4.1 HTTPS 如何工作?

1. 浏览器请求 HTTPS 站点
2. 服务器返回 SSL 证书
3. 浏览器验证证书，并建立加密连接
4. 数据加密传输，防止窃听

✅ **HTTPS 保护用户隐私，防止攻击**

总结

✅ **前端框架（React、Vue、Angular）提升开发效率**
✅ **后端基础（RESTful API、Node.js）实现前后端分离**
✅ **Web 安全（XSS、CSRF、HTTPS）保护数据**
✅ **HTTP 让前后端通信更加标准化**

下一步：持续学习，迈向全栈开发！

一、模拟类的主要方式

1. 构造函数模式（Constructor Pattern）

- 实现：使用构造函数和new关键字创建实例，方法定义在原型上。

```
function Person(name) {
  this.name = name;
}
Person.prototype.sayHello = function() {
  console.log(`Hello, I'm ${this.name}`);
};
const alice = new Person("Alice");
```

2. 工厂函数模式（Factory Function）

- 实现：返回对象的函数，手动绑定方法（通常不共享方法）。

```
function createPerson(name) {
  return {
    name,
    sayHello() { console.log(`Hello, I'm ${this.name}`); }
  };
}
const bob = createPerson("Bob");
```

3. Object.create()

- 实现：基于现有对象作为原型创建实例。

```
const personProto = {
  sayHello() { console.log(`Hello, I'm ${this.name}`); }
};
const carol = Object.create(personProto);
carol.name = "Carol";
```

4. 原型链继承（Prototypal Inheritance）

- 实现：通过原型链实现继承（需手动修复构造函数）。

```
function Parent(name) { this.name = name; }
function Child(name) { Parent.call(this, name); }
Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;
```

5. 模块模式（Module Pattern）

- 实现：使用闭包封装私有状态。

```
const Person = (() => {
  let privateField = 0; // 私有变量
  function Person(name) { this.name = name; }
  Person.prototype.getSecret = () => privateField;
  return Person;
})();
```

6. ES6 class语法（语法糖）

- 实现：使用class和extends关键字。

```
class Person {
  constructor(name) { this.name = name; }
  sayHello() { console.log(`Hello, I'm ${this.name}`); }
}
class Student extends Person { /* ... */ }
```

二、为什么要模拟类?

1. 代码组织与可维护性

- 类将数据和方法封装在一起，结构清晰，符合面向对象编程（OOP）思维，便于维护。

2. 代码复用

- 通过继承和原型链，可以复用方法和属性，减少重复代码。

3. 封装与数据隐藏

- 使用闭包或私有字段（如ES2022的#语法）实现私有成员，保护内部状态。

4. 多态性

- 子类可以重写父类方法，实现不同行为，增强灵活性。

5. 兼容性与历史原因

- 在ES6之前，开发者必须通过原型链模拟类，以构建复杂应用；理解这些方式有助于维护旧代码。

6. 开发习惯

- 类语法降低了传统OOP语言（如Java/C++）开发者的学习成本，提供更直观的语法。

三、选择方式的考量

- 简单性：ES6的class语法最简洁，推荐使用。
- 继承需求：复杂继承可选用“寄生组合继承”或class。
- 私有成员：模块模式或ES2022+的#私有字段。
- 性能：原型方法共享内存，工厂函数每个实例独立方法（可能浪费内存）。

总结

JavaScript模拟类的方式从原型链到class语法逐步演进，核心目标是通过封装、继承和多态提高代码质量。尽管class是语法糖，理解底层原型机制仍对解决复杂问题至关重要。

JavaScript 中的 Promise

1. 什么是 Promise?

Promise 是 JavaScript 中用于处理异步操作的对象。它代表一个异步操作的最终完成（或失败）及其结果值。Promise 有三种状态：

- Pending（等待）：初始状态，既不是成功，也不是失败。
- Fulfilled（已完成）：操作成功完成。 Rejected
- （已拒绝）：操作失败。

2. 创建 Promise

Promise 通过 new Promise() 构造函数创建，接受一个执行器函数（executor），该函数有两个参数：resolve 和 reject。

```
const myPromise = new Promise((resolve, reject) => {
  // 异步操作
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve("操作成功! ");
    } else {
      reject("操作失败! ");
    }
  }, 1000);
});
```

resolve 和 reject 的本质

- resolve：将 Promise 的状态从 pending（等待）变为 fulfilled（已完成），并传递一个值作为成功的结果。
- reject：将 Promise 的状态从 pending（等待）变为 rejected（已拒绝），并传递一个值（通常是错误信息）作为失败的原因。

它们只是普通的 JavaScript 函数，由 Promise 内部实现并传递给执行器函数。

---

resolve 和 reject 的参数

- resolve(value)：
  - value 是 Promise 成功时传递的值，可以是任意类型（如字符串、数字、对象等）。
  - 如果 value 是一个 Promise，则当前 Promise 的状态会由这个传入的 Promise 决定（即“跟随”这个 Promise 的状态）。
  - 如果 value 是一个普通值，则 Promise 直接变为 fulfilled，并将 value 作为结果。

- reject(reason)：
  - reason 是 Promise 失败时传递的原因，通常是错误信息（如字符串、Error 对象等）。
  - reason 可以是任意类型，但通常是一个 Error 对象，以便更好地追踪错误。

3. 使用 Promise

Promise 对象有两个主要方法用于处理结果：

- then()：处理成功状态。
- catch()：处理失败状态。

```
myPromise
  .then((result) => {
    console.log(result); // 输出: 操作成功!
  })
  .catch((error) => {
    console.error(error); // 输出: 操作失败!
  });
```

then() 和 catch() 是 Promise 对象的核心方法，用于处理 Promise 的成功或失败状态。它们的参数是回调函数，具体如下：

---

then() 方法

then() 用于处理 Promise 的成功状态（fulfilled）。它接受两个可选参数：

- 第一个参数：成功时的回调函数（onFulfilled）。
- 第二个参数：失败时的回调函数（onRejected）。

语法：

```
promise.then(onFulfilled, onRejected);
```

参数说明：

- onFulfilled(value)：
  - 当 Promise 成功时调用。
  - value 是 resolve 传递的值。
  - 如果未提供此函数，则成功状态会向下传递到链中的下一个 then。
- onRejected(reason)：
  - 当 Promise 失败时调用。
  - reason 是 reject 传递的原因（通常是错误信息）。

- 如果未提供此函数，则失败状态会向下传递到链中的下一个 catch 或 then 的第二个参数。

示例：

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("成功！");
    // reject("失败！");
  }, 1000);
});

myPromise.then(
  (value) => {
    console.log("成功：", value); // 输出：成功： 成功！
  },
  (reason) => {
    console.error("失败：", reason); // 如果 reject 被调用，输出：失败： 失败！
  }
);
```

---

catch() 方法

catch() 用于处理 Promise 的失败状态（rejected）。它相当于 then(null, onRejected) 的简写形式。

语法：

```
promise.catch(onRejected);
```

参数说明：

- onRejected(reason)：
  - 当 Promise 失败时调用。
  - reason 是 reject 传递的原因（通常是错误信息）。

示例：

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("失败！");
  }, 1000);
});

myPromise
  .then((value) => {
    console.log("成功：", value); // 不会执行
  })
  .catch((reason) => {
    console.error("失败：", reason); // 输出：失败： 失败！
  });
```

---

then() 和 catch() 的链式调用

then() 和 catch() 都会返回一个新的 Promise，因此可以链式调用。

链式调用示例：

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(10);
  }, 1000);
});

myPromise
  .then((value) => {
    console.log("第一步成功：", value); // 输出：第一步成功： 10
    return value * 2; // 返回一个新值，传递给下一个 then
  })
  .then((value) => {
    console.log("第二步成功：", value); // 输出：第二步成功： 20
    throw new Error("手动抛出错误"); // 抛出错误，触发 catch
  })
  .catch((reason) => {
    console.error("捕获错误：", reason.message); // 输出：捕获错误： 手动抛出错误
```

```
});
```


then() 和 catch() 的返回值

- then() 和 catch() 的返回值：
  - 如果回调函数返回一个值，则新的 Promise 会以该值作为结果。
  - 如果回调函数抛出错误，则新的 Promise 会以该错误作为失败原因。
  - 如果回调函数返回一个 Promise，则新的 Promise 会“跟随”这个返回的 Promise 的状态。

示例：

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(10);
  }, 1000);
});

myPromise
  .then((value) => {
    console.log("第一步成功：", value); // 输出：第一步成功： 10
    return new Promise((resolve) => {
      setTimeout(() => resolve(value * 2), 1000); // 返回一个新的 Promise
    });
  })
  .then((value) => {
    console.log("第二步成功：", value); // 1秒后输出：第二步成功： 20
  })
  .catch((reason) => {
    console.error("捕获错误：", reason);
  });
```

小结

- then(onFulfilled, onRejected)：
  - 用于处理成功或失败状态。
  - 第一个参数是成功回调，第二个参数是失败回调（可选）。
- catch(onRejected)：
  - 用于处理失败状态，是 then(null, onRejected) 的简写。
- 链式调用：
  - then() 和 catch() 都会返回一个新的 Promise，支持链式调用。
- 返回值：

- 回调函数的返回值会决定新的 Promise 的状态和结果。

通过 then() 和 catch()，你可以清晰地处理 Promise 的成功和失败状态，并构建复杂的异步逻辑。

4. Promise 链

then() 方法可以链式调用，每个 then() 返回一个新的 Promise，允许进一步的异步操作。

```
myPromise
  .then((result) =>
    { console.log(result);  输出: 操作成功!
    // return "继续处理";
  })
  .then((newResult) =>
    { console.log(newResult // 输出: 继续处理
    );
  })
  .catch((error) =>
    { console.error(error);
  });
```

5. Promise 的静态方法

- Promise.resolve(value)：返回一个已解决的 Promise。
- Promise.reject(reason)：返回一个已拒绝的 Promise。
- Promise.all(iterable)：所有 Promise 都成功时返回结果数组，任何一个失败则立即拒绝。
- Promise.race(iterable)：第一个完成的 Promise 决定结果。


```
const promise1 = Promise.resolve(3);
const promise2 = new Promise((resolve, reject) => setTimeout(resolve, 100, 'foo'));
const promise3 = Promise.reject("出错");

Promise.all([promise1, promise2])
  .then((values) => {
    console.log(values); // 输出: [3, "foo"]
  })
  .catch((error) => {
    console.error(error);
  });

Promise.race([promise1, promise2])
  .then((value) => {
    console.log(value); // 输出: 3
  });
```

6. 错误处理

除了 catch()，还可以在 then() 中传入第二个函数处理错误。

```
myPromise
  .then(
    (result) => {
      console.log(result);
    },
    (error) => {
      console.error(error);
    }
  );
```

7. 异步函数与 async/await

async/await 是处理 Promise 的语法糖，使异步代码看起来像同步代码。

```
async function myAsyncFunction() {
  try {
    const result = await myPromise;
    console.log(result); // 输出: 操作成功!
  } catch (error) {
    console.error(error);
  }
}

myAsyncFunction();
```

总结

Promise 是 JavaScript 中处理异步操作的核心工具，通过 then() 和 catch() 方法处理结果，支持链式调用和错误处理。async/await 进一步简化了 Promise 的使用，使代码更易读和维护。

JavaScript 基础练习环境设置文档

1. 基础开发环境准备

1.1 安装 Node.js

JavaScript 运行环境，推荐安装 LTS 版本：

- Node.js 官网下载
- 安装后验证：

```
node -v
npm -v
```

1.2 安装代码编辑器

推荐使用：

- Visual Studio Code (免费)
- WebStorm (付费)

1.3 浏览器

推荐 Chrome 或 Firefox，用于前端调试

2. 项目初始化

2.1 创建项目目录

```
mkdir js-basics-practice
cd js-basics-practice
```

2.2 初始化 npm 项目

```
npm init -y
```

2.3 创建基础文件结构

```
js-basics-practice/
├── index.html       # HTML 入口文件
├── script.js        # JavaScript 主文件
├── style.css        # 可选样式文件
└── exercises/       # 练习文件目录
    ├── variables.js
    ├── functions.js
    ├── arrays.js
    └── objects.js

```

3. HTML 基础模板

在 index.html 中添加：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JavaScript 基础练习</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>JavaScript 基础练习</h1>
    <div id="output"></div>

    <!-- 引入你的 JavaScript 文件 -->
    <script src="script.js"></script>

    <!-- 练习文件按需引入 -->
    <!-- <script src="exercises/variables.js"></script> -->
</body>
</html>
```


4. JavaScript 基础测试代码

在 script.js 中添加：

```
console.log("JavaScript 环境已准备好！");

// 简单的 DOM 操作测试
document.getElementById('output').innerHTML =
    '<p>环境已设置完成，打开开发者工具(Console)查看日志。</p>';

// 练习区域
// 在这里或单独的练习文件中编写你的代码
```

5. 运行和测试

5.1 浏览器中测试

1. 直接在文件管理器中双击打开 index.html
2. 或使用 VS Code 的 Live Server 扩展

5.2 使用 Node.js 运行纯 JavaScript 文件

```
node exercises/variables.js
```

6. 可选工具安装

6.1 安装 ESLint (代码风格检查)

```
npm install eslint --save-dev
npx eslint --init
```

6.2 安装 Prettier (代码格式化)

```
npm install --save-dev prettier
```

6.3 安装 nodemon (自动重启)

```
npm install -g nodemon
```

使用方式:

```
nodemon exercises/functions.js
```

7. 练习建议

1. 从 exercises/variables.js 开始，逐步练习
2. 每个练习文件顶部添加练习说明
3. 使用 console.log() 输出结果
4. 完成后在浏览器控制台或终端查看输出

示例练习文件 (exercises/variables.js):


```
/*
 * 变量与数据类型练习
 * 1. 声明不同数据类型的变量
 * 2. 进行类型转换练习
 * 3. 模板字符串使用
 */

// 1. 变量声明
let name = "Alice";
const age = 25;
var isStudent = true;

// 2. 类型转换
let strToNumber = Number("123");
let boolToStr = String(true);

// 3. 模板字符串
console.log(`姓名: ${name}, 年龄: ${age}, 学生: ${isStudent ? '是' : '否'}`);

// 更多练习...

```

Web 安全

1. 常见 Web 攻击

✅ **防止数据泄露，确保用户信息安全**

攻击类型	说明	解决方案
XSS（跨站脚本攻击）	插入恶意 JS	输入过滤、CSP
CSRF（跨站请求伪造）	伪造用户操作	CSRF Token 验证
SQL 注入	通过输入框执行 SQL	使用预处理语句
DDoS 攻击	短时间大量请求导致服务器崩溃	限流、CDN 防护

---

2. 防御措施

2.1 防止 XSS

```
<input type="text" oninput="alert('XSS 攻击！')">

```

✅ **禁止直接执行用户输入的 HTML**

```
element.innerText = userInput; // 而不是 innerHTML

```

2.2 防止 CSRF

✅ **使用 CSRF Token**

```
<input type="hidden" name="csrf_token" value="安全的 token">

```


2.3 使用 HTTPS

✅ **数据加密，防止中间人攻击**

```
HTTP → 明文传输（不安全）
HTTPS → 加密传输（安全）

```

1. 什么是 Webpack Bundle Analyzer?

Webpack Bundle Analyzer 是一个可视化工具，用于分析 Webpack 打包后的文件：

- 生成依赖图：展示每个模块的大小及其依赖关系。
- 优化打包结果：帮助识别冗余代码、未使用的模块以及过大的文件。

---

2. 安装与配置

2.1 安装

通过 npm 或 yarn 安装：

```
npm install --save-dev webpack-bundle-analyzer
```

或

```
yarn add --dev webpack-bundle-analyzer
```

2.2 配置

在 webpack.config.js 中引入并配置插件：


```
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;

mo ule.exports =
  { plugins: [
    new BundleAnalyzerPlugin({
      analyzerMode: 'server', // 启动本地服务器查看报告
      openAnalyzer: true,     // 打包完成后自动打开浏览器
      generateStatsFile: true, // 生成 stats.json 文件
      statsFilename: 'stats.json', // 指定 stats.json 文件名
    }),
  ],
};
```

2.3 运行

运行 Webpack 打包命令：

```
npx webpack --mode production
```

打包完成后，Webpack Bundle Analyzer 会自动启动一个本地服务器，并在浏览器中打开分析报告。

---

3. 分析报告解读

3.1 报告界面

- 左侧树状图：展示所有打包文件的依赖关系。
- 右侧矩形图：每个矩形代表一个模块，面积越大表示文件体积越大。
- 颜色区分：
  - 绿色：业务代码。
  - 橙色：第三方库（如 node_modules 中的代码）。
  - 红色：重复代码或未使用的模块。

3.2 关键功能

1. 查看模块大小：
   - 将鼠标悬停在矩形上，显示模块的详细信息（文件名、大小、路径等）。

2. 搜索模块：
   - 在搜索框中输入模块名称，快速定位特定模块。
3. 切换视图：
   - Treemap：默认视图，展示模块的体积和依赖关系。
   - Network：模拟模块加载顺序和依赖关系。
4. 导出报告：
   - 点击右上角的 Export 按钮，导出分析结果为 JSON 文件。

---

4. 高级配置

4.1 自定义端口

默认情况下，Webpack Bundle Analyzer 会使用 8888 端口。如果端口被占用，可以指定其他端口：

```
new BundleAnalyzerPlugin({
  analyzerPort: 9999, // 使用 9999 端口
});
```

4.2 生成静态报告

如果不希望启动本地服务器，可以生成静态 HTML 报告：

```
new BundleAnalyzerPlugin({
  analyzerMode: 'static', // 生成静态 HTML 文件
  reportFilename: 'report.html', // 指定报告文件名
});
```

4.3 仅生成 stats.json 文件

如果只需要生成 stats.json 文件，可以禁用本地服务器：


```
new BundleAnalyzerPlugin({
  analyzerMode: 'disabled', // 禁用本地服务器
  generateStatsFile: true,  // 生成 stats.json 文件
});
```

---

5. 使用场景

5.1 优化代码分割

- 识别重复代码：通过分析报告，找到被多个入口引用的模块，提取公共代码。
- 优化动态加载：检查异步加载的模块是否合理。

5.2 减少打包体积

- 删除未使用的模块：通过分析报告，找到未使用的模块并移除。
- 压缩大文件：识别体积过大的模块，优化其加载方式（如懒加载）。

5.3 调试依赖关系

- 分析依赖图：查看模块之间的依赖关系，解决循环依赖问题。
- 验证 Tree Shaking：检查是否成功移除未使用的代码。

---

6. 示例

6.1 配置示例

```
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;

module.exports = {
  mode: 'production',
  entry: {
    home: './src/home.js',
    about: './src/about.js',
  },
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  plugins: [
    new BundleAnalyzerPlugin({
      analyzerMode: 'server',
      openAnalyzer: true,
    }),
  ],
};
```

6.2 分析报告示例

- 首页入口：home.bundle.js（包含业务代码和部分第三方库）。
- 关于页入口：about.bundle.js（包含业务代码和部分第三方库）。
- 公共代码：vendors.bundle.js（包含 node_modules 中的代码）。

---

7. 总结

- 作用：Webpack Bundle Analyzer 是一个强大的工具，帮助开发者分析和优化 Webpack 打包结果。
- 核心功能：可视化依赖图、识别冗余代码、优化打包体积。
- 适用场景：代码分割、Tree Shaking、性能优化。

通过合理使用 Webpack Bundle Analyzer，你可以显著提升应用的性能和加载速度！

教学目标

1. 理解 Webpack 多页面应用（MPA）的配置方法。
2. 掌握如何为每个 HTML 页面单独引用对应的入口 JS 文件。
3. 学会提取公共代码，优化打包结果。

教学内容

1. 问题背景

默认情况下，HtmlWebpackPlugin 会将所有入口的 JS 文件注入到每个 HTML 页面中。但在多页面应用中，我们希望每个 HTML 页面只引用其对应的入口 JS 文件。

---

2. 解决方案

通过配置 HtmlWebpackPlugin 的 chunks 选项，为每个页面指定需要注入的 JS 文件。

---

3. 具体步骤

3.1 项目结构

```
src/
├── pages/
│   ├── home/
│   │   ├── index.html      # 首页模板
│   │   └── index.js        # 首页入口
│   └── about/
│       ├── index.html      # 关于页模板
│       └── index.js        # 关于页入口
public/                     # 静态资源（图片等）
webpack.config.js
```

3.2 修改 webpack.config.js

在 HtmlWebpackPlugin 中为每个页面指定 chunks，确保只注入相关的 JS 文件。

```
const path = require('path');
```



```
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: {
    home: './src/pages/home/index.js', // 首页入口
    about: './src/pages/about/index.js', // 关于页入口
  },
  output: {
    filename: '[name].bundle.js', // 动态生成文件名
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  plugins: [
    // 首页配置
    new HtmlWebpackPlugin({
      template: './src/pages/home/index.html',
      filename: 'home.html',
      chunks: ['home'], // 只注入 home 入口的 JS
    }),
    // 关于页配置
    new HtmlWebpackPlugin({
      template: './src/pages/about/index.html',
      filename: 'about.html',
      chunks: ['about'], // 只注入 about 入口的 JS
    }),
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: 'babel-loader',
      },
    ],
```

```
  },
  devServer:
    { static:
    './dist', hot:
    true,                    默认打开首页
    open: ['/home.html'], //
  },
};
```

3.3 示例代码

src/pages/home/index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>首页</title>
</head>
<body>
  <h1>这是首页</h1>
  <div id="app"></div>
</body>
</html>
```

src/pages/home/index.js

```
console.log('这是首页的 JS 文件');
```

src/pages/about/index.html


```
<!DOCTYPE html>
<html>
<head>
  <title>关于我们</title>
</head>
<body>
  <h1>这是关于页</h1>
  <div id="app"></div>
</body>
</html>
```

src/pages/about/index.js

```
console.log('这是关于页的 JS 文件');
```

3.4 运行结果

- 打包后生成 dist/home.html 和 dist/about.html。
- home.html 只引用 home.bundle.js。
- about.html 只引用 about.bundle.js。

4. 高级配置

4.1 提取公共代码

如果多个页面共享某些代码（如公共库），可以使用 optimization.splitChunks 提取公共代码：

```
optimization: {
  splitChunks:
    { chunks:
    'all',
    cacheGroups: {
      vendor: {
        test: /[\\/]node_modules[\\/]/,
        name: 'vendors',
      },
    },
  },
},
```

然后在 HtmlWebpackPlugin 中注入公共代码：

```
chunks: ['vendors', 'home'] // 首页注入 vendors 和 home
```

4.2 动态生成配置

如果页面较多，可以通过动态生成配置减少重复代码：


```
const pages = [
  { name: 'home', title: '首页' },
  { name: 'about', title: '关于我们' },
];

module.exports = {
  entry: pages.reduce((entries, page) => {
    entries[page.name] = `./src/pages/${page.name}/index.js`;
    return entries;
  }, {}),
  plugins: pages.map(page =>
    new HtmlWebpackPlugin({
      template: `./src/pages/${page.name}/index.html`,
      filename: `${page.name}.html`,
      chunks: [page.name], // 动态注入对应入口
      title: page.title,
    })
  ),
};
```

---

5. 总结

- 核心配置：通过 HtmlWebpackPlugin 的 chunks 选项控制每个 HTML 页面引用的 JS 文件。
- 优化建议：
  - 提取公共代码，减少重复加载。
  - 动态生成配置，简化多页面管理。
- 适用场景：多页面应用（MPA）或需要独立入口的复杂项目。

课后练习

1. 创建一个包含 3 个页面的项目，分别为 home、about 和 contact，每个页面引用不同的入口 JS 文件。
2. 配置 optimization.splitChunks，提取 node_modules 中的代码到 vendors.bundle.js。
3. 使用 Webpack Bundle Analyzer 分析打包结果，验证公共代码是否被正确提取。

参考资料

- Webpack 官方文档：https://webpack.js.org/
- HtmlWebpackPlugin 文档：https://github.com/jantimon/html-webpack-plugin
- Webpack Bundle Analyzer：https://github.com/webpack-contrib/webpack-bundle-analyzer

1. 安装 Bootstrap

通过 npm 安装 Bootstrap：

```
npm install bootstrap
```

---

2. 安装 Bootstrap 的依赖

Bootstrap 5 依赖于 **Popper.js**（用于工具提示和弹出框），因此需要安装 @popperjs/core：

```
npm install @popperjs/core
```

如果你使用的是 Bootstrap 4，还需要安装 jquery 和 popper.js：

```
npm install jquery popper.js
```

---

3. 在项目中引入 Bootstrap

在你的 JavaScript 入口文件（如 src/index.js）中引入 Bootstrap 的 CSS 和 JS 文件：

```
// 引入 Bootstrap 的 CSS 文件
import 'bootstrap/dist/css/bootstrap.min.css';

// 引入 Bootstrap 的 JS 文件
import 'bootstrap';
```

---

4. 配置 Webpack 处理 CSS 文件

Webpack 默认只能处理 JavaScript 文件，因此需要安装 style-loader 和 css-loader 来处理 CSS 文件。

安装 Loader：


```
npm install style-loader css-loader --save-dev
```

在 webpack.config.js 中配置 Loader：

```
module.exports = {
  module:
    { rules:
    [
      {
        test: /\.css$/, // 匹配 CSS 文件
        use: ['style-loader', 'css-loader'], // 使用 style-loader 和 css-loader
      },
    ],
  },
};
```

---

5. 配置 Webpack 处理 Bootstrap 的字体和图片

Bootstrap 的 CSS 文件中可能会引用字体和图片文件，因此需要配置 Webpack 处理这些静态资源。

安装 file-loader：

```
npm install file-loader --save-dev
```

在 webpack.config.js 中配置 file-loader：

```
module.exports = {
  module:
    { rules:
    [
      {
        test: /\.css$/, // 匹配 CSS 文件
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.(woff|woff2|eot|ttf|otf|svg)$/, // 匹配字体文件
        type: 'asset/resource', // 使用内置的 asset/resource 处理
      },
      {
        test: /\.(png|jpg|jpeg|gif)$/, // 匹配图片文件
        type: 'asset/resource', // 使用内置的 asset/resource 处理
      },
    ],
  },
};
```

---

6. 运行 Webpack 打包

运行以下命令进行打包：

```
npx webpack
```

打包完成后，Bootstrap 的样式和功能将会被包含在你的项目中。

---

7. 在 HTML 中使用 Bootstrap

在 HTML 文件中使用 Bootstrap 的样式和组件：


```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Webpack + Bootstrap</title>
</head>
<body>
  <div class="container">
    <h1 class="text-primary">Hello, Bootstrap!</h1>
    <button class="btn btn-success">Click Me</button>
  </div>
  <script src="bundle.js"></script> <!-- Webpack 打包后的文件 -->
</body>
</html>
```

---

8. 可选：使用 Sass 版本的 Bootstrap

如果你希望使用 Bootstrap 的 Sass 版本，可以安装 bootstrap 的 Sass 文件：

```
npm install bootstrap
```

然后在项目中引入 Bootstrap 的 Sass 文件：

```
// 引入 Bootstrap 的 Sass 文件
import 'bootstrap/scss/bootstrap.scss';
```

安装 sass-loader 和 sass：

```
npm install sass-loader sass --save-dev
```

在 webpack.config.js 中配置 sass-loader：

```
module.exports = {
  module:
    { rules:
    [
      {
        test: /\.scss$/, // 匹配 SCSS 文件
        use: ['style-loader', 'css-loader', 'sass-loader'],
      },
    ],
  },
};
```

总结

1. 安装 Bootstrap 及其依赖。
2. 在 JavaScript 中引入 Bootstrap 的 CSS 和 JS 文件。
3. 配置 Webpack 处理 CSS、字体和图片文件。
4. 运行 Webpack 打包。
5. 在 HTML 中使用 Bootstrap。

通过以上步骤，你就可以在 Webpack 项目中使用 Bootstrap 了！

后端基础

1. HTTP 协议与 RESTful API

✅ **前端与后端的核心交互方式**

1.1 HTTP 请求示例

```
GET /api/users HTTP/1.1
Host: example.com

```

- GET 获取数据
- POST 发送数据
- PUT 更新数据
- DELETE 删除数据

1.2 RESTful API 设计

```
{
    "id": 1,
    "name": "张三",
    "email": "zhangsan@example.com"
}

```

✅ **JSON 作为标准数据格式**

---

2. Node.js 简介

✅ **Node.js 是 JavaScript 运行环境**，可用于后端开发。

```
const http = require("http");

const server = http.createServer((req, res) => {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hello, Node.js!");
});

server.listen(3000);
console.log("服务器运行在 http://localhost:3000");

```

✅ **可用于构建 API、服务器端渲染等**

---

3. 前后端分离开发模式

✅ **前端负责 UI 和交互，后端提供 API**

```
前端（React/Vue） → 发送请求 → 后端（Node.js/Python/Java） → 返回 JSON 数据

```

✅ **提升开发效率，便于团队协作**

CSS 框架对比与推荐指南

在现代 Web 开发中，选择合适的 CSS 框架可以大幅提升开发效率。然而，面对众多框架，我们需要了解它们各自的特点和差异。本文通过对比常见 CSS 框架的关键特性，为不同需求提供框架选择指南，并推荐几个值得深入学习的框架。

1. CSS 框架对比分析

下表从多个维度对常见 CSS 框架进行比较，包括框架类型、布局方式、响应式支持、自定义程度、内置 JavaScript 插件、设计风格、学习曲线，以及社区支持情况：

框架名称	框架类型	布局方式	响应式设计	自定义程度	JavaScript 插件	设计风格	学习曲线	社区与支持
Bootstrap	UI 框架 (HTML+CSS+JS)	网格系统，Flexbox	内建	中等，可定制主题	丰富（模态框、轮播、弹窗等）	类 Material Design	低	大型社区，文档完善
Tailwind CSS	CSS 工具类框架	Flexbox，Grid，工具类	内建	**高**，自定义性极强	无（需手动添加）	极简风格	中等	较大社区，文档丰富
Semantic UI	UI 框架 (HTML+CSS+JS)	Flexbox，Grid	内建	中等，可定制主题	包含（模态框、菜单、弹出框等）	类 Material Design	中等	中型社区，活跃度适中
Foundation	UI 框架 (HTML+CSS+JS)	网格系统，Flexbox	内建	中等，可定制主题	包含（模态框、轮播、弹窗等）	专业风格	中等	较大社区，活跃但不及 Bootstrap
Bulma	CSS 框架 (仅 HTML+CSS)	Flexbox	内建	**高**，自定义性强	无（需手动添加）	简约现代风格	低	较小社区，但在不断成长
Materialize	UI 框架 (HTML+CSS+JS)	网格系统，Flexbox	内建	中等，可定制主题	包含（模态框、下拉菜单、卡片等）	Material Design	低	中型社区，支持良好
Ulkit	UI 框架 (HTML+CSS+JS)	Flexbox，Grid	内建	中等，可定制主题	包含（弹出框、导航等）	极简风格	中等	较小社区，但依然活跃
Material-UI	UI 框架 (React 组件库)	Flexbox，Grid	内建	**高**，自定义性强	基于 React 的丰富组件	Material Design	中等	非常大的社区，活跃度高 (主要用于 React 项目)

2. CSS 框架选择指南

根据不同的开发需求，选择最合适的 CSS 框架非常重要。以下针对几种常见需求给出框架选择建议：

- 构建企业级项目：优先选择 **Bootstrap。**它是最流行的框架，功能全面且文档完善，已经被广泛应用于各类企业级项目。
- 高度定制、自由设计 UI：建议使用 **Tailwind CSS。**该框架采用原子化工具类设计，灵活度极高，能够避免冗余 CSS，非常适合现代前端开发。

- 偏好 Material Design 风格：选择 **Material-UI**（基于 React）。它遵循 Google 的 Material Design 规范，提供丰富的 React UI 组件，适用于现代 Web 应用开发。
- 快速开发简洁 Web 界面：可以选用 **Bulma。**Bulma 较为轻量，简单易学，基于 Flexbox，非常适合用来快速构建简洁的用户界面。
- 注重强大的响应式设计：考虑 **Foundation。**Foundation 在 Web 和移动端都有出色表现，并且被许多企业项目广泛采用，响应式网格系统非常灵活。
- 追求丰富现代的 UI 组件：可以使用 **Ulkit。**Ulkit 提供种类丰富的组件，语法简洁，对于不想深度编写 CSS 的开发者来说非常友好。

3. 深入学习重点推荐

如果希望对某个框架进行深入研究，以下几个框架值得重点关注：

- Tailwind CSS（最推荐）
  - 提供完全的 UI 设计掌控，不依赖任何预制的组件库。
  - 可避免产生冗余 CSS，从而提高开发速度。
  - 维护性极高，不会产生全局样式冲突问题。
  - 能无缝配合 React、Vue、Next.js 等现代前端框架使用。
  - 流行度迅速提升，目前 GitHub 上的 star 数已超越 Bootstrap，发展趋势强劲。
  - 深入学习建议：理解 Tailwind CSS 的核心概念，掌握配置定制优化，并练习将其与 React/Vue/Next.js 项目相结合。
- Bootstrap（备选）
  - 业界最常用的 CSS 框架，在企业项目中非常常见。
  - 拥有庞大的生态，文档完善，非常适合团队协作开发。
  - 内置大量常用的 JavaScript 组件，适合构建功能完备的企业级页面。
  - 深入学习建议：重点学习响应式布局网格、自定义主题配置，并结合 SCSS 进行深度定制开发。
- Material-UI（适合 React 生态）
  - React 生态中最流行的 UI 组件库之一。
  - 遵循 Google 的 Material Design 规范，提供专业且统一的用户界面体验。
  - 非常适合用于构建大型企业级 React 应用。
  - 深入学习建议：学习主题定制技巧、响应式布局的实现，并结合 Redux 等状态管理工具开发复杂应用。

4. 最终建议

最后，根据以上对比分析，针对不同的学习或项目目标可做出以下选择：

- 掌握主流 UI 框架，适合大部分项目：选择 **Bootstrap。**
- 追求更灵活的样式系统，适用于现代前端开发：选择 **Tailwind CSS。**

- React 开发者，想用最强 UI 组件库：选择 **Material-UI**。

最终结论：

- 如果希望深入理解 CSS 的样式设计并提高开发效率，推荐学习 **Tailwind CSS**。
- 如果想熟练掌握业界最常用的 UI 框架以胜任各类企业项目，建议选择 **Bootstrap**。
- 对于使用 React 的开发者，建议学习 **Material-UI** 以利用 React 生态中最强大的 UI 组件库。

核心免费，增值付费

这些框架基础功能可以 **免费** 使用，无论是个人项目还是商业项目：

- Bootstrap（MIT 许可证）官网：https://getbootstrap.com
- Tailwind CSS（MIT 许可证）官网：https://tailwindcss.com
- Bulma（MIT 许可证）官网：https://bulma.io
- Foundation（MIT 许可证）官网：https://get.foundation
- Materialize（MIT 许可证）官网：https://materializecss.com
- UIkit（MIT 许可证）官网：https://getuikit.com

提供付费增值服务的 CSS 框架

虽然以下框架的核心是免费的，但它们提供了一些**额外的付费功能**，如 **专业 UI 组件、企业支持或主题**：

- Tailwind CSS（付费增值功能）
  - Tailwind UI（官网）：提供预构建的高级 UI 组件，需要 **付费** 购买。
  - Tailwind Enterprise：为企业提供技术支持、定制化功能和优化指导。
- Bootstrap（付费增值功能）
  - Bootstrap Themes（官网）：官方提供的 **高级主题**，需要 **付费** 购买。
  - 企业支持服务（部分第三方公司提供）。
- Material-UI（现更名为 MUI）（免费+付费增值）
  - 核心框架（免费）：MIT 许可证，适用于个人和商业项目。
  - MUI X Pro（官网）：提供**更高级的数据网格、日历、图表等组件**，需要 **付费** 订阅。
  - MUI Design Kits：Figma 设计套件，付费获取。

总结

框架	免费版	付费版内容
Bootstrap	✓✓ 完全免费	付费主题和企业支持
Tailwind CSS	✓✓ 完全免费	付费 UI 组件（Tailwind UI）
Bulma	✓✓ 完全免费	无付费内容
Foundation	✓✓ 完全免费	无付费内容
Material-UI (MUI)	✓✓ 基础版免费	付费高级组件（MUI X Pro）

大多数开发者 **无需付费** 就可以使用这些 CSS 框架，并且能够满足大部分项目需求。如果你需要更高效的开发，或者企业项目有更复杂的 UI 需求，可以考虑购买它们的付费增值服务。

前端框架简介

1. 为什么需要前端框架?

前端框架可以**提高开发效率**，帮助开发者更快地构建复杂的交互式 Web 应用。

✓ **前端框架的优势**：

- 组件化开发（代码复用）
- 提高性能（虚拟 DOM、数据绑定）
- 简化状态管理
- 增强可维护性

---

2. 三大主流前端框架

2.1 React

✓ **由 Facebook 开发**，基于 **组件化** 和 **虚拟 DOM**，适用于**大型单页应用。**

特点:

- JSX（类 HTML 语法）
- 虚拟 DOM 提高性能
- 单向数据流（数据管理清晰）
- 支持 Hooks 和状态管理（Redux、MobX）

```
function App() {
    return <h1>Hello, React!</h1>;
}

```

2.2 Vue.js

✓ **由尤雨溪开发**，语法简单，适用于**小型项目和渐进式开发。**

特点:

- 双向数据绑定（v-model）
- 指令系统（v-if、v-for）
- 组件化
- Vue Router + Vuex（路由和状态管理）


```
<template>
    <h1>{{ message }}</h1>
</template>
<script>
export default {
    data() {
        return { message: "Hello, Vue!" };
    }
};
</script>

```

2.3 Angular

✓ **由 Google 开发**，基于 **TypeScript**，适用于**大型企业应用。**

特点:

- 完整 MVC 结构
- 双向数据绑定
- 内置依赖注入
- 模块化开发

```
@Component({
    selector: "app-root",
    template: `<h1>{{ title }}</h1>`
})
export class AppComponent {
    title = "Hello, Angular!";
}

```

---

3. 前端框架的选择

适用场景	推荐框架
小型项目	Vue.js
中型项目	React
大型企业项目	Angular
单页应用（SPA）	React / Vue
复杂状态管理	React（Redux）、Vue（Vuex）

静态网页项目：个人简历网站

项目简介

目标：创建一个**个人简历网站**，展示个人信息、技能、项目经验等，适用于求职或作品展示。

功能：

- 个人简介
- 技能列表
- 教育 & 工作经历
- 项目展示
- 联系方式
- 响应式布局（适配 PC & 移动端）

技术栈

✓ **HTML**：网页结构

✓ **CSS（Flexbox/Grid）**：布局 & 样式

✓ **FontAwesome**：图标库（如社交媒体图标）

✓ **Google Fonts**：美化字体

✓ **CSS 动画**：增强交互体验

任务拆解

✓ **任务 1**：搭建 HTML 页面结构

✓ **任务 2**：使用 CSS 美化页面

✓ **任务 3**：添加个人信息、技能、经历等内容

✓ **任务 4**：优化排版 & 适配移动端

✓ **任务 5**：添加 CSS 过渡效果，让页面更美观

代码示例

HTML 结构

```
<!DOCTYPE html>
```



```
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>个人简历</title>
    <link rel="stylesheet" href="styles.css">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css">
</head>
<body>
    <header>
        <h1>张三</h1>
        <p>前端开发工程师</p>
    </header>

    <section class="about">
        <h2>关于我</h2>
        <p>热爱编程，擅长 HTML、CSS、JavaScript，喜欢学习新技术。</p>
    </section>

    <section class="skills">
        <h2>技能</h2>
        <ul>
            <li><i class="fa-brands fa-html5"></i> HTML</li>
            <li><i class="fa-brands fa-css3-alt"></i> CSS</li>
            <li><i class="fa-brands fa-js"></i> JavaScript</li>
            <li><i class="fa-brands fa-react"></i> React</li>
        </ul>
    </section>

    <section class="experience">
        <h2>工作经历</h2>
        <div class="job">
            <h3>前端工程师 - ABC 公司</h3>
            <p>2021 - 现在</p>
            <ul>
                <li>负责公司官网开发与维护</li>
                <li>优化前端性能，提高加载速度</li>
            </ul>
```

```
        </div>
    </section>

    <section class="projects">
        <h2>项目经验</h2>
        <div class="project">
            <h3>个人博客</h3>
            <p>基于 HTML + CSS + JavaScript 开发的个人博客网站。</p>
            <button>查看项目</button>
        </div>
    </section>

    <section class="contact">
        <h2>联系方式</h2>
        <p>Email: zhangsan@example.com</p>
        <p>GitHub: <a href="#">github.com/zhangsan</a></p>
        <p>LinkedIn: <a href="#">linkedin.com/in/zhangsan</a></p>
    </section>

    <footer>
        <p>© 2025 张三 | 个人简历</p>
    </footer>
</body>
</html>
```

CSS 样式


```
/* 通用样式 */
```

```
body {
    font-family: 'Arial', sans-serif;
    margin: 0;
    padding: 0;
    line-height: 1.6;
    background-color: #f4f4f4;
    color: #333;
}

header {
    background-color: #333;
    color: #fff;
    text-align: center;
    padding: 20px;
}

h1, h2 {
    margin-bottom: 10px;
}

section {
    max-width: 800px;
    margin: 20px auto;
    background: white;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
}

ul {
    list-style: none;
    padding: 0;
}

ul li {
    display: flex;
    align-items: center;
    margin: 5px 0;
```



```
}

ul li i {
    margin-right: 10px;
    color: #007BFF;
}

footer {
    text-align: center;
    padding: 10px;
    background-color: #333;
    color: white;
}

/* 响应式布局 */
@media (max-width: 600px) {
    body {
        font-size: 16px;
    }
}
```

✓ **添加 @media 适配移动端，确保手机上显示良好**

进阶优化

✓ **添加 CSS 过渡效果**：

```css
button {
    background: #007BFF;
    color: white;
    padding: 10px;
    border: none;
    cursor: pointer;
    transition: background 0.3s ease-in-out;
}

button:hover {
    background: #0056b3;
}
```

✓ **使用动画增强用户体验**

```css
@keyframes fadeIn {
    from { opacity: 0; }
    to { opacity: 1; }
}

section {
    animation: fadeIn 1s ease-in-out;
}
```

课后挑战

✓ **挑战 1**：为每个项目添加图片和更多详情
✓ **挑战 2**：使用 Flexbox 或 Grid 重新布局
✓ **挑战 3**：添加“下载 PDF 简历”按钮
✓ **挑战 4**：制作夜间模式，使用 JavaScript 切换 dark mode

总结

✓ **通过本项目掌握 HTML + CSS 基础**
✓ **学习 Flexbox/Grid 布局**

✓ **掌握 CSS 动画 & 过渡效果**
✓ **使用 @media 适配移动端**

下一步：将静态简历升级为**动态页面**，加入 JavaScript 交互！

练习二：动态网页项目 - 在线留言板

项目简介

目标：开发一个**在线留言板**，用户可以提交留言，留言将动态显示在页面上，并且可以存储在本地存储（LocalStorage）。

主要功能：

- 用户输入留言并提交
- JavaScript 处理表单验证
- 使用 **AJAX/Fetch API** 发送留言
- 在页面上 **动态显示留言**
- 使用 **LocalStorage** 存储留言（页面刷新后仍然存在）

技术栈

✅ **HTML + CSS**（页面结构与样式）

✅ **JavaScript（DOM 操作、事件监听）**

✅ **AJAX + JSON（异步提交留言）**

✅ **LocalStorage（本地存储数据）**

任务拆解

✅ **任务 1**：创建留言表单

✅ **任务 2**：使用 JavaScript 处理表单验证

✅ **任务 3**：用 AJAX 发送留言数据（模拟）

✅ **任务 4**：在页面上动态显示留言

✅ **任务 5**：使用 LocalStorage 存储留言，刷新页面后数据仍存在

代码示例

HTML 结构


```
html复制编辑<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>在线留言板</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>留言板</h1>
    <form id="messageForm">
        <input type="text" id="username" placeholder="输入你的名字" required>
        <textarea id="message" placeholder="输入你的留言" required></textarea>
        <button type="submit">提交</button>
    </form>
    <h2>留言列表</h2>
    <ul id="messageList"></ul>

    <script src="script.js"></script>
</body>
</html>

```

JavaScript 处理留言

```
    document.getElementById("messageForm").addEventListener("submit", function(event) {
    event.preventDefault(); // 阻止默认提交

    let username = document.getElementById("username").value.trim();
    let message = document.getElementById("message").value.trim();

    if (!username || !message)
        { alert("请填写完整信息!
        "); return;
    }

    let newMessage = { username, message };

    // 读取 LocalStorage 中的留言列表
    let messages = JSON.parse(localStorage.getItem("messages")) || [];
    messages.push(newMessage);
    localStorage.setItem("messages", JSON.stringify(messages));

    displayMessages();
    document.getElementById("messageForm").reset();
});

// 显示留言
function displayMessages() {
    let messages = JSON.parse(localStorage.getItem("messages")) || [];
    let messageList = document.getElementById("messageList");
    messageList.innerHTML = messages.map(msg => `<li><strong>${msg.username}:
</strong>${msg.message}</li>`).join("");
}

// 页面加载时显示已存储的留言
displayMessages();

```

✅ **提交留言后，数据存储到 LocalStorage，并实时显示!**

CSS 美化


```
body {
    font-family: Arial, sans-serif;
    text-align: center;
    background-color: #f4f4f4;
}

form {
    margin: 20px auto;
    max-width: 400px;
    background: white;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
}

input, textarea {
    width: 100%;
    padding: 10px;
    margin: 10px 0;
    border: 1px solid #ddd;
    border-radius: 5px;
}

button {
    background: #007BFF;
    color: white;
    padding: 10px;
    border: none;
    cursor: pointer;
    transition: background 0.3s ease-in-out;
}

button:hover {
    background: #0056b3;
}

```

课后挑战

✓ **挑战 1**：添加“删除留言”功能

✓ **挑战 2**：使用 fetch() 发送留言到后端 API（可以使用 JSONPlaceholder 模拟）

✓ **挑战 3**：增加点赞功能

下一步：练习 **响应式网页项目 - 在线商城首页**！

练习三：响应式网页项目 - 在线商城首页

项目简介

目标：开发一个**响应式在线商城首页**，展示商品列表，适配移动端，支持搜索和筛选功能。

主要功能：

- 商品卡片展示（图片 + 价格 + 购买按钮）
- 响应式布局（适配 PC & 手机）
- CSS Flexbox/Grid 布局
- 媒体查询 **@media** 适配不同设备
- JavaScript 实现搜索功能

技术栈

✓ **HTML + CSS**（页面结构与样式）

✓ **CSS Flexbox/Grid**（实现响应式布局）

✓ **媒体查询 @media**（适配不同设备）

✓ **JavaScript（搜索 & 过滤商品）**

任务拆解

✓ **任务 1**：设计商城首页，包含商品列表

✓ **任务 2**：使用 Flexbox / Grid 布局

✓ **任务 3**：使用 @media 实现自适应

✓ **任务 4**：添加搜索栏，实现商品筛选

✓ **任务 5**：优化性能（懒加载）

代码示例

HTML 结构

```html
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>在线商城</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>商城</h1>
        <input type="text" id="search" placeholder="搜索商品...">
    </header>
    <section class="products">
        <div class="product" data-name="苹果">🍎 苹果 - ¥10</div>
        <div class="product" data-name="香蕉">🍌 香蕉 - ¥5</div>
        <div class="product" data-name="橙子">🍊 橙子 - ¥8</div>
    </section>

    <script src="script.js"></script>
</body>
</html>

```

CSS 响应式布局


```css
body {
    font-family: Arial, sans-serif;
    text-align: center;
}

header {
    background: #007BFF;
    color: white;
    padding: 10px;
}

input {
    padding: 5px;
    width: 80%;
    margin-top: 10px;
}

.products {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
    gap: 10px;
    padding: 20px;
}

.product {
    background: white;
    padding: 20px;
    border-radius: 5px;
    box-shadow: 0px 0px 5px rgba(0, 0, 0, 0.1);
}

@media (max-width: 600px) {
    .products { grid-template-columns: 1fr; }
}

```

JavaScript 搜索功能

```
document.getElementById("search").addEventListener("input", function() {
    let filter = this.value.toLowerCase();
    let products = document.querySelectorAll(".product");

    products.forEach(product => {
        let name = product.dataset.name.toLowerCase();
        product.style.display = name.includes(filter) ? "block" : "none";
    });
});

```

✓ **输入关键词，商品列表会动态筛选！**

课后挑战

✓ **挑战 1**：增加“加入购物车”功能

✓ **挑战 2**：使用 localStorage 记录购物车数据

✓ **挑战 3**：加载更多商品（滚动分页）

下一步：整合动态数据，升级为 **完整在线商城**！

总结

✓ **留言板练习 JavaScript 交互 & LocalStorage**

✓ **在线商城练习 响应式布局 & 搜索筛选**

✓ **结合 JavaScript，使网页更加动态化**

下一步：综合前后端，开发完整 Web 应用！

后台管理系统开发需求

概述

开发一个后台管理系统，实现用户登录、主页导航和用户管理功能，使用 Bootstrap 和 localStorage。

1. 功能需求

1. 用户登录：
   - 输入用户名和密码，点击登录后跳转到主页。
2. 主页：
   - 显示欢迎信息。
   - 提供侧边栏菜单，包含“用户管理”和“退出登录”选项。
3. 用户管理：
   - 展示用户列表（ID、用户名、邮箱）。
   - 支持添加、编辑、删除用户。

---

2. 技术要求

1. 使用 HTML、CSS、JavaScript 开发。
2. 使用 Bootstrap 实现页面布局和样式。
3. 使用 localStorage 存储用户数据。

---

3. 开发任务

1. 实现登录页面，点击登录后跳转到主页。
2. 实现主页，包含侧边栏菜单和欢迎信息。
3. 实现用户管理页面，支持用户信息的增删改查。

---

4. 扩展功能（可选）

1. 增加表单验证（如邮箱格式验证）。
2. 增加分页功能，支持大量用户数据的展示。
3. 增加搜索功能，支持按用户名或邮箱搜索用户。

综合项目：购物车功能开发

目标

1. 使用 Webpack 打包模块化代码
2. 通过 fetch() 模拟异步获取商品数据
3. 实现购物车添加/删除/计算总价功能
4. 应用 ES6+ 特性（解构、模板字符串、箭头函数）

---

1. 项目结构

```
webpack-demo/
├── src/
│   ├── index.js        # 入口文件
│   ├── api.js          # 模拟数据请求
│   ├── cart.js         # 购物车逻辑
│   └── styles.css      # 样式
├── public/
│   └── index.html      # HTML 模板
├── package.json
└── webpack.config.js
```

空文件创建好之后，请执行：

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

2. 核心代码实现

2.1 api.js - 模拟异步请求

```
// 模拟获取商品数据（使用 Promise）
export const fetchProducts = () => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve([
        { id: 1, name: "苹果", price: 5 },
        { id: 2, name: "香蕉", price: 3 },
        { id: 3, name: "橙子", price: 4 }
      ]);
    }, 1000);
  });
};
```

2.2 cart.js - 购物车逻辑


```
export class Cart {
  constructor() {
    this.items = [];
  }

  // 添加商品（使用数组操作）
  addItem(product, quantity = 1) {
    const existingItem = this.items.find(item => item.id === product.id);
    if (existingItem) {
      existingItem.quantity += quantity;
    } else {
      this.items.push({ ...product, quantity });
    }
  }

  // 删除商品（使用 filter）
  removeItem(productId) {
    this.items = this.items.filter(item => item.id !== productId);
  }

  // 计算总价（使用 reduce）
  calculateTotal() {
    return this.items.reduce((total, item) => total + item.price * item.quantity, 0);
  }

  // 获取商品列表（返回副本避免直接修改）
  getItems() {
    return [...this.items];
  }
}
```

2.3 index.js - 主入口

```
import { fetchProducts } from './api';
```



```
import { Cart } from './cart';
import './styles.css';

// 初始化购物车
const cart = new Cart();
// 全局变量（注意：实际项目中应避免过多全局变量，这里为简化示例）
let products = [];
// DOM 操作
const renderProducts = async () => {
  products = await fetchProducts();
  const productList = document.getElementById('product-list');

  // 使用模板字符串生成 HTML
  productList.innerHTML = products.map(product => `
    <div class="product">
      <h3>${product.name}</h3>
      <p>价格：¥${product.price}</p>
      <button onclick="addToCart(${product.id})">加入购物车</button>
    </div>
  `).join('');
};

// 全局函数（实际项目应避免，这里简化演示）
window.addToCart = (productId) => {
  const product = products.find(p => p.id === productId);
  cart.addItem(product);
  updateCartDisplay();
};

const updateCartDisplay = () => {
  const cartItems = document.getElementById('cart-items');
  const total = document.getElementById('total-price');

  // 使用解构和 map
  cartItems.innerHTML = cart.getItems().map(({ id, name, price, quantity }) => `
    <li>
      ${name} × ${quantity}
      <button onclick="removeFromCart(${id})">删除</button>
```

```
    </li>
  `).join('');

  total.textContent = `总价：¥${cart.calculateTotal()}`;
};

window.removeFromCart = (productId) => {
  cart.removeItem(productId);
  updateCartDisplay();
};

// 初始化渲染
renderProducts();
```

---

3. Webpack 配置 (webpack.config.js)


```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  mode: 'development',
  devServer: {
    static: './dist',
    hot: true,
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env'],
          },
        },
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html',
    }),
  ],
```

```
};
```

---

4. 配套文件

public/index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>购物车练习</title>
</head>
<body>
  <h1>商品列表</h1>
  <div id="product-list"></div>

  <h2>购物车</h2>
  <ul id="cart-items"></ul>
  <p id="total-price">总价：¥0</p>
</body>
</html>
```

src/styles.css


```
.product {
  border: 1px solid #ccc;
  padding: 10px;
  margin: 10px;
  border-radius: 5px;
}

button {
  background: #007bff;
  color: white;
  border: none;
  padding: 5px 10px;
  cursor: pointer;
}
```

---

5. 运行流程

1. 安装依赖：

```
npm install webpack webpack-cli webpack-dev-server html-webpack-plugin style-loader
css-loader babel-loader @babel/core @babel/preset-env --save-dev

or

npm install webpack webpack-cli webpack-dev-server --save-dev
npm install html-webpack-plugin --save-dev
npm install style-loader css-loader --save-dev
npm install babel-loader @babel/core @babel/preset-env --save-dev

or

npm install webpack --save-dev
npm install webpack-cli --save-dev
npm install webpack-dev-server --save-dev
npm install html-webpack-plugin --save-dev
npm install style-loader --save-dev
npm install css-loader --save-dev
npm install babel-loader --save-dev
npm install @babel/core --save-dev
npm install @babel/preset-env --save-dev
```

1. 启动开发服务器：

```
npx webpack serve
```

1. 访问 http://localhost:8080 查看效果

---

6. 关键知识点验证

1. 模块化：import/export 分割代码
2. 异步操作：fetchProducts 使用 Promise 模拟
3. Webpack 打包：CSS 和 JS 合并到 bundle.js
4. ES6+ 特性：箭头函数、模板字符串、解构赋值
5. 数据操作：数组方法 map/filter/reduce 的应用

通过这个练习，你可以直观看到 Webpack 如何整合模块、Babel 如何转译代码、以及现代 JavaScript 如何组织复杂功能。