

○、课程介绍

讲师：千锋教育--古艺散人

0.1 学习前提

0.2 授课方式

0.3 课程受众

0.4 课程安排

0.5 课程收获

一、基础应用篇

1.1 为什么需要 Webpack

1.1.1 如何解决作用域问题

1.1.2 如何解决代码拆分问题

1.1.3 如何让浏览器支持模块

1.1.4 Webpack 搞定这一切

1.1.5 Webpack 与竞品

1.2 小试 Webpack

1.2.1 开发准备

1.2.2 安装 Webpack

1.2.3 运行 Webpack

1.2.4 自定义 Webpack 配置

1.2.5 重新运行项目

1.3 自动引入资源

1.3.1 什么是插件

1.3.2 使用 HtmlWebpackPlugin

1.3.3 清理dist

1.4 搭建开发环境

1.4.1 mode 选项

1.4.2 使用 source map

1.4.3 使用 watch mode(观察模式)

1.4.4 使用 webpack-dev-server

1.5 资源模块

1.5.1 Resource 资源

1.5.2 inline 资源

1.5.3 source 资源

1.5.4 通用资源类型

1.6 管理资源

1.6.1 什么是loader

1.6.2 加载CSS

1.6.3 抽离和压缩CSS

1.6.4 加载 images 图像

1.6.5 加载 fonts 字体

1.6.6 加载数据

1.6.7 自定义 JSON 模块 parser

1.7 使用 babel-loader

- 1.7.1 为什么需要 babel-loader
 - 1.7.2 使用 babel-loader
 - 1.7.3 regeneratorRuntime 插件
 - 1.8 代码分离
 - 1.8.1 入口起点
 - 1.8.2 防止重复
 - 1.8.3 动态导入
 - 1.8.4 懒加载
 - 1.8.5 预获取/预加载模块
 - 1.9 缓存
 - 1.9.1 输出文件的文件名
 - 1.9.2 缓存第三方库
 - 1.9.3 将 js 文件放到一个文件夹中
 - 1.10 拆分开发环境和生产环境配置
 - 1.10.1 公共路径
 - 1.10.2 环境变量
 - 1.10.3 拆分配置文件
 - 1.10.4 npm 脚本
 - 1.10.5 提取公共配置
 - 1.10.6 合并配置文件
- ## 二、高级应用篇
- 2.1 提高开发效率，完善团队开发规范
 - 2.1.1 source-map
 - 2.1.2 devServer
 - 2.1.3 模块热替换与热加载
 - 2.1.4 eslint
 - 2.1.5 git-hooks 与 husky
 - 2.2 模块与依赖
 - 2.2.1 Webpack 模块与解析原理
 - 2.2.2 模块解析(resolve)
 - 2.2.3 外部扩展(Externals)
 - 2.2.4 依赖图(dependency graph)
 - 2.3 扩展功能
 - 2.3.1 PostCSS 与 CSS模块
 - 2.3.2 Web Works
 - 2.3.3 TypeScript
 - 2.4 多页面应用
 - 2.4.1 entry 配置
 - 2.4.2 配置 index.html 模板
 - 2.4.3 多页面应用
 - 2.5 Tree shaking
 - 2.5.1 tree-shaking实验
 - 2.5.2 sideEffects
 - 2.6 渐进式网络应用程序 PWA
 - 2.6.1 非离线环境下运行

- 2.6.2 添加 Workbox
- 2.6.3 注册 Service Worker
- 2.7 shimming 预置依赖
 - 2.7.1 Shimming 预置全局变量
 - 2.7.2 细粒度 Shimming
 - 2.7.3 全局 Exports
 - 2.7.4 加载 Polyfills
 - 2.7.5 进一步优化 Polyfills
- 2.8 创建 library
 - 2.8.1 创建一个 library
 - 2.8.2 Webpack 配置
 - 2.8.3 导出 Library
 - 2.8.4 外部化 lodash
 - 2.8.5 外部化限制
 - 2.8.6 优化输出
 - 2.8.7 发布为 npm package
- 2.9 模块联邦
 - 2.9.1 什么是模块联邦
 - 2.9.2 应用案例
- 2.10 提升构建性能
 - 2.10.1 通用环境
 - 2.10.2 开发环境
 - 2.10.3 生产环境

○、课程介绍

讲师：千锋教育--古艺散人

公众号：大前端私房菜
回复：webpack5
获取webpack课程导图

在讲述本节课程之前，我们先来思考一个问题：

为什么学习webpack?

如果你尚没有接触过webpack，那么你对构建和打包的概念恐怕是模糊不清的。你可能更习惯使用开箱即用的脚手架来生成你的项目配置，或者迭代着某个项目的业务，却对它的开发/生产环境搭建知之甚少。

要知道，前端架构最重要的点就在于前端工程化，而webpack则是我们搭建前端工程化环境的一个技术选型。

那么为什么是webpack呢？

在github上搜索webpack的时候，repositories的数量是157k。

事实上，无论是开源项目还是企业项目，最主流的前端工程化方案的技术选型都是webpack。

而全新版本的webpack5，则是具备了比以往版本更强大的功能，甚至是诸多企业级前端工程化技术选型的不二选择。

0.1 学习前提

学习webpack5需要什么前提？

在学习本课程之前，期望你能具备前端的基础知识如html，css，es6+。如果对nodejs和工程化有一定了解的话，那就更好不过了。

0.2 授课方式

在本课程中，我们将通过前后呼应的demo来一步步从0到1地进行webpack5教学，在课程后期我们也将学到更低层的原理知识。从而做到知其然并知其所以然的精熟掌握程度。

0.3 课程受众

webpack5课程适用于有一定前端知识基础的前端学习者及从业者——比如中高级前端工程师。如果你对webpack不甚了解，但又经常接触webpack配置相关的项目，那么毫无疑问，你此时正需要学习它。

0.4 课程安排

接下来我们了解下我们的webpack5课程安排：

我们的webpack5课程分为四大部分，分别是webpack基础应用篇，webpack高级应用篇，webpack项目实战篇以及webpack内部原理篇。

(配图)

其中，在基础应用篇我们将学习到webpack的基础配置方案，掌握webpack的各种基础配置项所对应的功能。

而在高级应用篇，我们将具体分析webpack的每个配置项，以及按需集成工程化模块，从而掌握定制项目配置的手段。

在项目应用篇中，我们将结合具体的项目案例，通过应用我们之前学到的webpack技术，来定制项目的工程化环境。真切地做到学以致用。

最后在内部原理篇中，我们将对webpack进行内部原理剖析，掌握webpack打包技术的底层实现。

0.5 课程收获

学习本课程后，会获得那些收获？

1. 首先通过本课程，你将学会webpack配置，并拥有工程化的前端思维，理解webpack在前端工程化领域的作用及原理。
2. 你将能够参与项目的打包配置，从工程化层面来优化开发环境、项目性能，落地面向前端业务的技术方案。
3. 学习了webpack，我们就具备了面向前端架构的核心竞争力。

最后，就让我们正式进入到webpack5的学习课程。

一、基础应用篇

1.1 为什么需要 Webpack

想要理解为什么要使用 webpack，我们先回顾下历史，在打包工具出现之前，我们是如何在 web 中使用 JavaScript 的。在浏览器中运行 JavaScript 有两种方法：

第一种方式，引用一些脚本来存放每个功能，比如下面这个文档：

01-why-webpack/index-1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-
scale=1.0">
<title>千锋大前端教研院-Webpack5学习指南</title>
</head>
<body>
<!-- HTML 代码 -->
<div>我的HTML代码</div>

<!-- 引入外部的 JavaScript 文件 -->
<script
src="https://cdn.bootcdn.net/ajax/libs/jquery/3.6.0/jquery.mi
n.js"></script>
<script
src="https://cdn.bootcdn.net/ajax/libs/lodash.js/4.17.21/loda
sh.core.min.js"></script>
```

```
<script src="https://cdn.bootcdn.net/ajax/libs/twitter-bootstrap/5.0.2/js/bootstrap.min.js"></script>

<!-- 引入我自己的 JavaScript 文件 --&gt;
&lt;script src="../scripts/common.js"&gt;&lt;/script&gt;
&lt;script src="../scripts/user.js"&gt;&lt;/script&gt;
&lt;script src="../scripts/authentication.js"&gt;&lt;/script&gt;
&lt;script src="../scripts/product.js"&gt;&lt;/script&gt;
&lt;script src="../scripts/inventory.js"&gt;&lt;/script&gt;
&lt;script src="../scripts/payment.js"&gt;&lt;/script&gt;
&lt;script src="../scripts/checkout.js"&gt;&lt;/script&gt;
&lt;script src="../scripts/shipping.js"&gt;&lt;/script&gt;
&lt;/body&gt;
&lt;/html&gt;</pre>
```

此解决方案很难扩展，因为加载太多脚本会导致网络瓶颈。同时如果你不小心更改了 JavaScript 文件的加载顺序，这个项目可能要崩溃。

第二种方式，使用一个包含所有项目代码的大型 .js 文件，对上面的文档做改进：

01-why-webpack/index-2.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>千锋大前端教研院-Webpack5学习指南</title>
</head>
<body>
<!-- HTML 代码 --&gt;
&lt;div&gt;我的HTML代码&lt;/div&gt;

<!-- 引入我自己的 JavaScript 文件 --&gt;
&lt;script src="../scripts/bundle.33520ba89e.js"&gt;&lt;/script&gt;
&lt;/body&gt;
&lt;/html&gt;</pre>
```

这种方式解决了方式一的问题，但会导致作用域、文件大小、可读性和可维护性方面的问题。如何解决这些问题，请往下阅读。

1.1.1 如何解决作用域问题

早先前，我们使用 Grunt 和 Gulp 两个工具来管理我们项目的资源。



这两个工具称为任务执行器，它们将所有项目文件拼接在一起。利用了立即调用函数表达式(IIFE) - `Immediately invoked function expressions`, 解决了大型项目的作用域问题；当脚本文件被封装在 IIFE 内部时，你可以安全地拼接或安全地组合所有文件，而不必担心作用域冲突。

什么是IIFE，参见下面的代码：

- 当函数变成立即执行的函数表达式时，表达式中的变量不能从外部访问。

```
(function () {
    var name = "Barry";
})();
// 无法从外部访问变量 name
name // 抛出错误: "Uncaught ReferenceError: name is not
defined"
```

- 将 IIFE 分配给一个变量，不是存储 IIFE 本身，而是存储 IIFE 执行后返回的结果。

```
var result = (function () {
  var name = "Barry";
  return name;
})();
// IIFE 执行后返回的结果:
result; // "Barry"
```

`Grunt`, `Gulp` 解决了作用域问题。但是，修改一个文件意味着必须重新构建整个文件。拼接可以做到很容易地跨文件重用脚本，却使构建结果的优化变得更加困难。如何判断代码是否实际被使用？

即使你只用到 `lodash` 中的某个函数，也必须在构建结果中加入整个库，然后将它们压缩在一起。大规模地实现延迟加载代码块及无用代码的去除，需要开发人员手动地进行大量工作。

01-why-webpack/index-3.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<script
src="https://cdn.bootcdn.net/ajax/libs/lodash.js/4.17.21/lodash.min.js"></script>
<title>千锋大前端教研院-Webpack5学习指南</title>
</head>
<body>
<script>
  // 这里我们只使用了一个join函数，确要引入整个lodash库
  const str = _.join(['千锋大前端教研院', 'Webpack5学习指南'], '-')
  console.log(str)
</script>
</body>
</html>
```

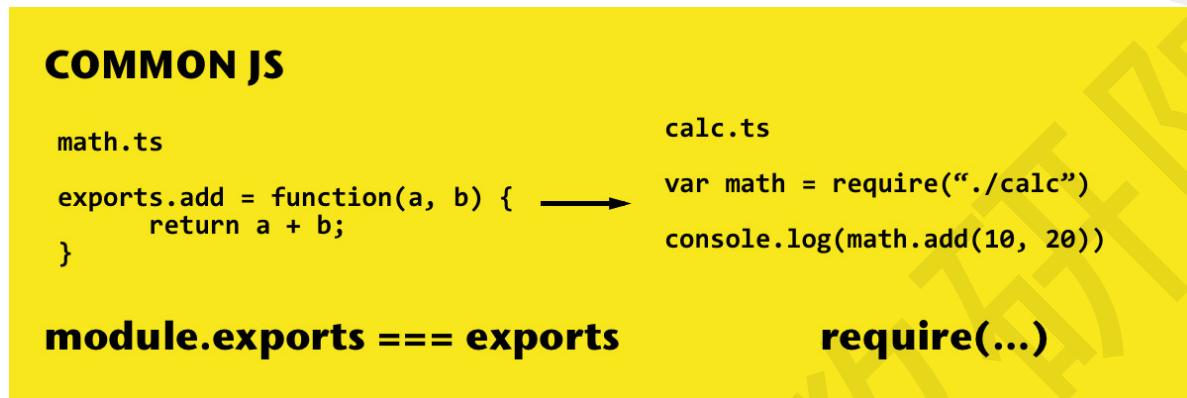
1.1.2 如何解决代码拆分问题

感谢 `Node.js`, `JavaScript` 模块诞生了！

`Node.js` 是一个 `JavaScript` 运行时，可以在浏览器环境之外的计算机和服务器中使用。`webpack` 运行在 `Node.js` 中。

当 Node.js 发布时，一个新的时代开始了，它带来了新的挑战。既然不是在浏览器中运行 JavaScript，现在已经没有了可以添加到浏览器中的 html 文件和 script 标签。那么 Node.js 应用程序要如何加载新的代码文件呢？

CommonJS 问世并引入了 `require` 机制，它允许你在当前文件中加载和使用某个模块。导入需要的每个模块，这一开箱即用的功能，帮助我们解决了代码拆分的问题。



Node.js 已经成为一种语言、一个平台和一种快速开发和创建快速应用程序的方式，接管了整个 JavaScript 世界。

但 CommonJS 没有浏览器支持。没有 [live binding\(实时绑定\)](#)。循环引用存在问题。同步执行的模块解析加载器速度很慢。虽然 CommonJS 是 Node.js 项目的绝佳解决方案，但浏览器不支持模块，我们似乎又遇到了新问题。

1.1.3 如何让浏览器支持模块

在早期，我们应用 Browserify 和 RequireJS 等打包工具编写能够在浏览器中运行的 CommonJS 模块：



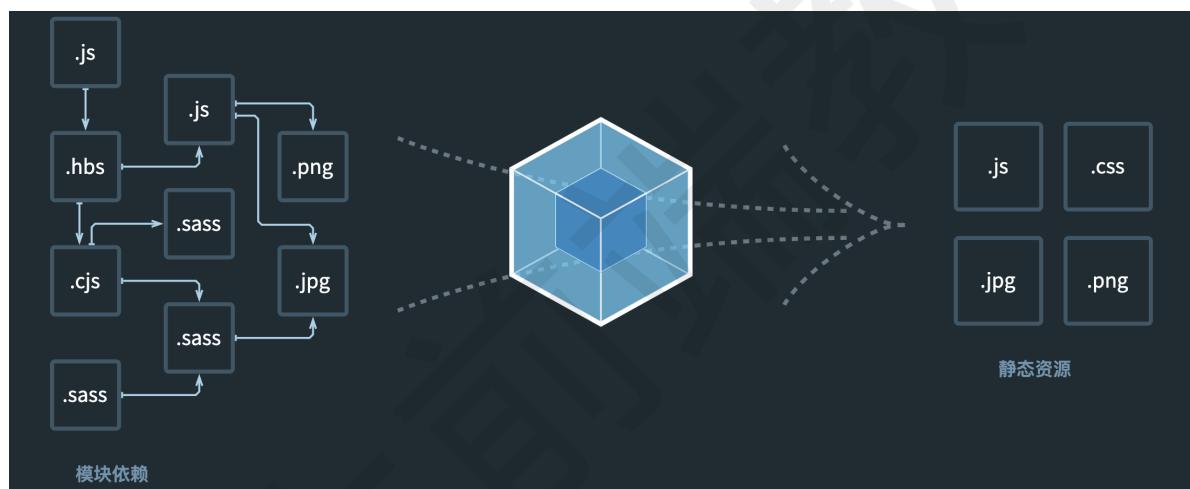
目前，我们还有一个选择，就是来自 Web 项目的好消息是，模块正在成为 ECMAScript 标准的官方功能。然而，浏览器支持不完整，版本迭代速度也不够快，还是推荐上面两个早期模块实现。早期的任务构建工具基于 Google 的 Closure 编译器，要求我们手动在顶部声明所有的依赖，开发体验不好。

1.1.4 Webpack 搞定这一切

是否可以有一种方式，不仅可以让我们编写模块，而且还支持任何模块格式（至少在我们到达 ESM 之前），并且可以同时处理 `resource` 和 `assets`？

这就是 `webpack` 存在的原因。它是一个工具，可以打包你的 JavaScript 应用程序（支持 ESM 和 CommonJS），可以扩展为支持许多不同的静态资源，例如：`images`, `fonts` 和 `stylesheets`。

`webpack` 关心性能和加载时间；它始终在改进或添加新功能，例如：异步地加载和预先加载代码文件，以便为你的项目和用户提供最佳体验。



1.1.5 Webpack 与竞品

- Webpack

`webpack` 为处理资源管理和分割代码而生，可以包含任何类型的文件。灵活，插件多。

- Parcel

`Parcel` 是 0 配置工具，用户一般无需再做其他配置即可开箱即用。

- Rollup

`Rollup` 用标准化的格式 (es6) 来写代码，通过减少死代码尽可能地缩小包体积。一般只用来打包 JS。

小结论：

构建一个简单的应用并让它快速运行起来？使用 Parcel。

构建一个类库只需要导入很少第三方库？使用 Rollup。

构建一个复杂的应用，需要集成很多第三方库？需要代码分拆，使用静态资源文件，还有 CommonJS 依赖？使用 webpack。

- Vite

在刚刚结束的 VueConf2021 中，除了 Vue 3.0 以外，另外一个亮点就是下一代构建工具 Vite 了。

在尤雨溪分享的 **【Vue 3 生态进展和计划】** 的演讲中，尤大神还特意提到 **Vite 将成为 Vue 的现代标配**。甚至最近新推出的 Petite Vue 从开发、编译、发布、Demo 几乎全都是使用 Vite 完成。



Vite 这种基于 ESM module 的构建方式会日益受到用户青睐，不仅因为 Vite 按需编译，热模块替换等特性，还有其丝滑的开发体验以及和 Vue 3 的完美结合。

按照这种说法，也许有人会问：是不是马上 **Webpack 就要被取代了，Vite 的时代就要到来了呢？**

Webpack、Vite 作为前端热门的工程化构建工具，它们都有各自的适用场景，并不存在“取代”这一说法。

1.2 小试 Webpack

1.2.1 开发准备

在进入 Webpack 世界之前，我们先来用原生的方法构建一个 Web 应用。

□ □ □ 一个 JavaScript 文件，编写一段通用的函数 `helloWorld`：

02-setup-app/src/hello-world.js

```
function helloWorld() {  
    console.log('Hello world')  
}
```

再创建一个JavaScript文件，调用这个函数：

02-setup-app/src/index.js

```
helloWorld()
```

最后创建一个 html 页面去引用这两个JS文件：

02-setup-app/index.html

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
<meta charset="UTF-8">  
<meta http-equiv="X-UA-Compatible" content="IE=edge">  
<meta name="viewport" content="width=device-width, initial-  
scale=1.0">  
<title>千锋大前端教研院-webpack5学习指南</title>  
</head>  
<body>  
    <!-- 注意这里的js文件的引用顺序要正确 -->  
    <script src=".src/index.js"></script>  
    <script src=".src/hello-world.js"></script>  
</body>  
</html>
```

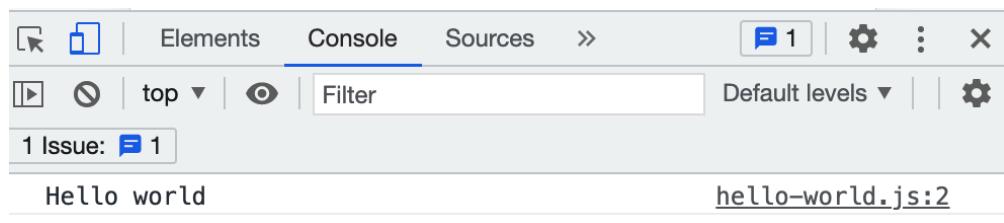
正常同步的 JavaScript 代码是按照在页面上加载的顺序执行的，上面html文件先引用 index.js 文件，后引用 hello-world.js 文件，由于两个文件的代码存在先定义后才能调用的顺序关系，所以浏览器运行后会报以下错误：

✖ ▶ **Uncaught ReferenceError: helloWorld is not defined** [index.js:1](#)
at [index.js:1](#)

调整 JS 文件的引用顺序：

```
<body>  
    <!-- 注意这里的js文件的引用顺序要正确 -->  
    <script src=".src/hello-world.js"></script>  
    <script src=".src/index.js"></script>  
</body>
```

在浏览器运行后输出如下：



如果页面引用的JS文件很少，我们可以手动的来调整顺序，但页面一旦引用大量的JS文件，调整顺序的心智负担和工作量可想而知，如何解决？我们就要有请 webpack 了。

1.2.2 安装 Webpack

- 前提条件

在开始之前，请确保安装了 [Node.js](#) 的最新版本。使用 Node.js 最新的长期支持版本(LTS - Long Term Support)，是理想的起步。使用旧版本，你可能遇到各种问题，因为它们可能缺少 webpack 功能，或者缺少相关 package。



Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

Download for macOS (x64)

14.17.6 LTS

Recommended For Most Users

16.10.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#) [Other Downloads](#) | [Changelog](#) | [API Docs](#)

Or have a look at the [Long Term Support \(LTS\) schedule](#)

你可以选择下载适合自己平台的安装包，自行安装即可，本文不再赘述。

- 本地安装

最新的 webpack 正式版本是：

webpack v5.64.2

要安装最新版本或特定版本，请运行以下命令之一：

```
npm install --save-dev webpack
# 或指定版本
npm install --save-dev webpack@<version>
```

提示:

是否使用 `--save-dev` 取决于你的应用场景。假设你仅使用 webpack 进行构建操作，那么建议你在安装时使用 `--save-dev` 选项，因为可能你不需要在生产环境上使用 webpack。如果需要应用于生产环境，请忽略 `--save-dev` 选项。

如果你使用 webpack v4+ 版本，并且想要在命令行中调用 `webpack`，你还需要安装 [CLI](#)。

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

对于大多数项目，我们建议本地安装。这可以在引入重大更新(breaking change)版本时，更容易分别升级项目。通常会通过运行一个或多个 [npm scripts](#) 以在本地 `node_modules` 目录中查找安装的 webpack，来运行 webpack：

```
"scripts": {
  "build": "webpack --config webpack.config.js"
}
```

提示:

想要运行本地安装的 webpack，你可以通过 `node_modules/.bin/webpack` 来访问它的二进制版本。另外，如果你使用的是 npm v5.2.0 或更高版本，则可以运行 `npx webpack` 来执行。

- 全局安装

通过以下 NPM 安装方式，可以使 `webpack` 在全局环境下可用：

```
npm install --global webpack
```

提示:

不推荐 全局安装 webpack。这会将你项目中的 webpack 锁定到指定版本，并且在使用不同的 webpack 版本的项目中，可能会导致构建失败。

- 最新体验版本

如果你热衷于使用最新版本的 webpack，你可以使用以下命令安装 beta 版本，或者直接从 webpack 的仓库中安装：

```
npm install --save-dev webpack@next
# 或特定的 tag/分支
npm install --save-dev webpack/webpack#<tagname/branchname>
```

提示：

安装这些最新体验版本时要小心！它们可能仍然包含 bug，因此不应该用于生产环境。

• 我们的安装

根据以上的各种情景，在我们的项目中安装 Webpack：

```
# 当前目录：任意你的目录/webpack5
[felix] webpack5 $ npm install webpack webpack-cli
```

1.2.3 运行 Webpack

Webpack安装好了以后，就可以在项目环境里运行了。在运行之前，我们先修改一下代码：

03-try-webpack/src/hello-world.js

```
function helloworld() {
  console.log('Hello world')
}

// 导出函数模块
export default helloworld
```

03-try-webpack/src/index.js

```
// 导入函数模块
import helloworld from './hello-world.js'

helloworld()
```

03-try-webpack/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>千锋大前端教研院-Webpack5学习指南</title>
</head>
<body>
<script src="./src/index.js"></script>
</body>
</html>
```

进入项目目录，运行 Webpack，结果如下：

```
[felix] 03-try-webpack $ npx webpack
asset main.js 50 bytes [emitted] [minimized] (name: main)
orphan modules 81 bytes [orphan] 1 module
./src/index.js + 1 modules 135 bytes [built] [code generated]

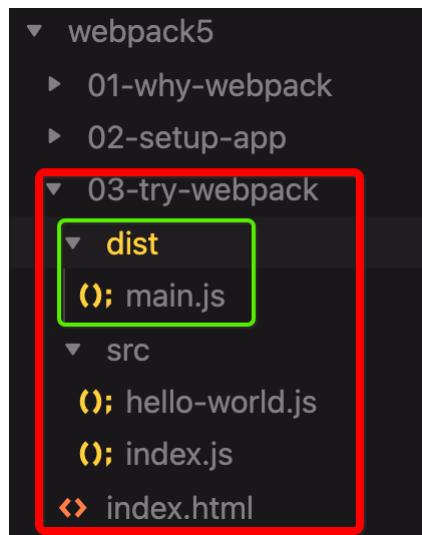
WARNING in configuration
The 'mode' option has not been set, webpack will fallback to
'production' for this value.
Set 'mode' option to 'development' or 'production' to enable
defaults for each environment.
You can also set it to 'none' to disable any default behavior.
Learn more: https://webpack.js.org/configuration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

webpack 5.54.0 compiled with 1 warning in 197 ms
```

在这里，我们在没有任何配置的情况下运行 Webpack（通常你会为 Webpack 提供一个配置文件，现在，自 Webpack4 开始，可以使用默认配置来打包文件了）。

这里还有一个警告：“mode” 选项尚未设置。我们将在本课程后面讨论“mode”选项。

从结果来看，webpack 为我们生成了一个 main.js 文件，具体见下图：



我们来看一下 `main.js` 里有什么：

03-try-webpack/dist/main.js

```
(()=>{"use strict";console.log("Hello world")})();
```

生成的代码非常简洁。这时你可能不禁会问，这个代码是从哪些文件里生成出来的呢？回到终端，我们再运行一下命令：

```
[felix] 03-try-webpack $ npx webpack --stats detailed
PublicPath: auto
asset main.js 50 bytes {179} [compared for emit] [minimized]
(name: main)
Entrypoint main 50 bytes = main.js
chunk {179} (runtime: main) main.js (main) 180 bytes [entry]
[rendered]
  > ./src main
orphan modules 103 bytes [orphan] 1 module
./src/index.js + 1 modules [860] 180 bytes {179} [depth 0]
[built] [code generated]
  [no exports]
  [no exports used]

.....
```

我们看到，`asset main.js` 是从入口 `> ./src main` 生成的。那么，我们能自己配置这个入口吗？请看下一节，自定义 Webpack 配置。

1.2.4 自定义 Webpack 配置

实际上，`webpack-cli` 给我们提供了丰富的终端命令行指令，可以通过 `webpack --help` 来查看：

<code>-w, --watch</code>	watch for files changes.
<code>--no-watch</code>	Do not watch for file changes.
<code>--watch-options-stdin</code>	Stop watching when stdin stream has ended.
<code>--no-watch-options-stdin</code>	Do not stop watching when stdin stream has ended.
 Global options:	
<code>--color</code>	Enable colors on console.
<code>--no-color</code>	Disable colors on console.
<code>-v, --version</code>	Output the version number of ' <code>webpack</code> ', ' <code>webpack-cli</code> ' and ' <code>webpack-dev-server</code> ' and commands.
<code>-h, --help [verbose]</code>	Display help for commands and options.
 Commands:	
<code>build bundle b [entries...]</code>	Run webpack (default command, can be omitted).
<code>configtest t [config-path]</code>	Validate a webpack configuration.
<code>help h [command] [option]</code>	Display help for commands and options.
<code>info i [options]</code>	Outputs information about your system.
<code>serve server s [entries...]</code>	Run the webpack dev server. To see all available options you need to install ' <code>webpack-dev-server</code> '.
<code>version v [commands...]</code>	Output the version number of ' <code>webpack</code> ', ' <code>webpack-cli</code> ' and ' <code>webpack-dev-server</code> ' and commands.
<code>watch w [entries...]</code>	Run webpack and watch for files changes.

To see list of all supported commands and options run '`webpack --help=verbose`'.

webpack documentation: <https://webpack.js.org/>.

CLI documentation: <https://webpack.js.org/api/cli/>.

Made with ❤ by the webpack team.

可是命令行不方便也不直观，而且还不利于保存配置的内容。因此，webpack 还给我们提供了通过配置文件，来自定义配置参数的能力。

03-try-webpack/webpack.config.js

```
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    // 输出文件夹必须定义为绝对路径
    path: './dist'
  },
  mode: 'none'
}
```

在项目目录下运行 `npx webpack`，可以通过配置文件来帮我们打包文件。

```
[felix] 03-try-webpack $ npx webpack
[webpack-cli] Invalid configuration object. Webpack has been
initialized using a configuration object that does not match the
API schema.
- configuration.output.path: The provided value "./dist" is not
an absolute path!
  -> The output directory as **absolute path** (required).
```

我们发现，打包并没有成功，因为 webpack 要求我们打包配置 `output.path` 的路径必须为绝对路径，通过 `path` 模块来定义输出路径为绝对路径：

03-try-webpack/webpack.config.js

```
const path = require('path')
module.exports = {
  entry: './src/index.js',

  output: {
    filename: 'bundle.js',

    // 输出文件夹必须定义为绝对路径
    path: path.resolve(__dirname, './dist')
  },

  mode: 'none'
}
```

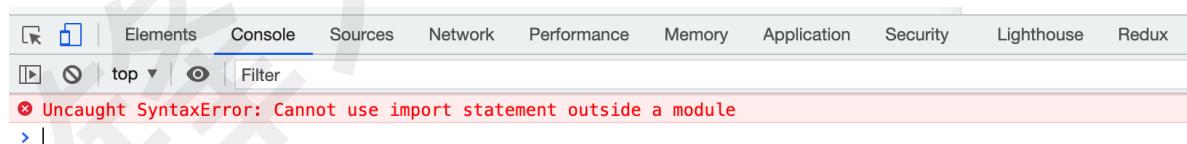
再次输入打包命令：

```
[felix] 03-try-webpack $ npx webpack
asset bundle.js 3.15 KiB [emitted] (name: main)
runtime modules 670 bytes 3 modules
cacheable modules 180 bytes
./src/index.js 77 bytes [built] [code generated]
./src/hello-world.js 103 bytes [built] [code generated]
webpack 5.54.0 compiled successfully in 81 ms
```

打包成功！

1.2.5 重新运行项目

项目文件通过 webpack 打包好了，可是我们在浏览器运行 `index.html` 提示如下错误：



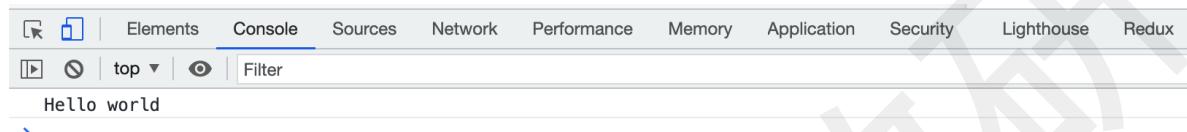
这是因为页面引用的JS代码，在浏览器里不能正确解析了，我们得去引用打包好了的JS才对。修改 `index.html`。

03-try-webpack/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>千锋大前端教研院-Webpack5学习指南</title>
</head>
<body>
<!-- <script src="./src/index.js"></script> -->
<!-- 引用打包好的 JS 文件 -->
<script src="./dist/bundle.js"></script>
</body>
</html>
```

在浏览器里再次运行 `index.html`:



大功告成!

1.3 自动引入资源

到目前为止，我们都是在 `index.html` 文件中手动引入所有资源，然而随着应用程序增长，如果继续手动管理 `index.html` 文件，就会变得困难起来。然而，通过一些插件可以使这个过程更容易管控。

1.3.1 什么是插件

插件 是 webpack 的核心功能。插件可以用于执行一些特定的任务，包括：打包优化，资源管理，注入环境变量等。Webpack 自身也是构建于你在 webpack 配置中用到的 **相同的插件系统** 之上！

想要使用一个插件，你只需要 `require()` 它，然后把它添加到 `plugins` 数组中。多数插件可以通过选项(option)自定义。你也可以在一个配置文件中因为不同目的而多次使用同一个插件，这时需要通过使用 `new` 操作符来创建一个插件实例。

Webpack 提供很多开箱即用的 [插件](#)。

1.3.2 使用 HtmlWebpackPlugin

首先安装插件：

```
npm install --save-dev html-webpack-plugin
```

并且调整 `webpack.config.js` 文件：

```
plugins: [
  // 实例化 html-webpack-plugin 插件
  new HtmlWebpackPlugin()
]
```

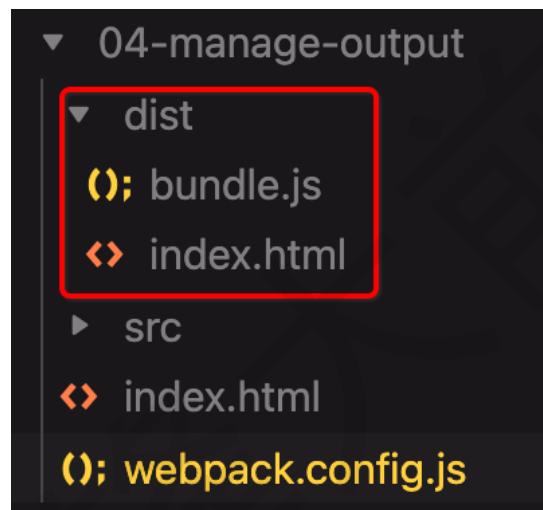
04-manage-output/webpack.config.js

```
//...
module.exports = {
//...

plugins: [
  // 实例化 html-webpack-plugin 插件
  new HtmlWebpackPlugin()
]
}
```

打包：

```
[felix] 04-manage-output $ npx webpack
```



打包生产的 `index.html` 内容如下：

04-manage-output/dist/index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>webpack App</title>
<meta name="viewport" content="width=device-width, initial-scale=1"><script defer src="bundle.js"></script></head>
<body>
</body>
</html>
```

打包后，我们发现这个 `dist/index.html` 似乎与先前的 `index.html` 并没有关系，`HtmlWebpackPlugin` 会默认生成它自己的 `index.html` 文件，并且所有的 bundle (bundle.js) 会自动添加到 html 中。

能否基于原有的 `index.html` 文件打包生成新的 `index.html` 呢？可以通过阅读 [HtmlWebpackPlugin](#) 插件提供的全部的功能和选项来找到答案。

首先删除 `index.html` 手工引入的 `js` 文件：

04-manage-output/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>千锋大前端教研院-Webpack5学习指南</title>
</head>
<body>
</body>
</html>
```

再次调整 `webpack.config.js` 文件：

```
plugins: [
  // 实例化 html-webpack-plugin 插件
  new HtmlWebpackPlugin({
    template: './index.html', // 打包生成的文件的模板
    filename: 'app.html', // 打包生成的文件名称。默认为index.html
    // 设置所有资源文件注入模板的位置。可以设置的值
    true | 'head' | 'body' | false, 默认值为 true
    inject: 'body'
  })
]
```

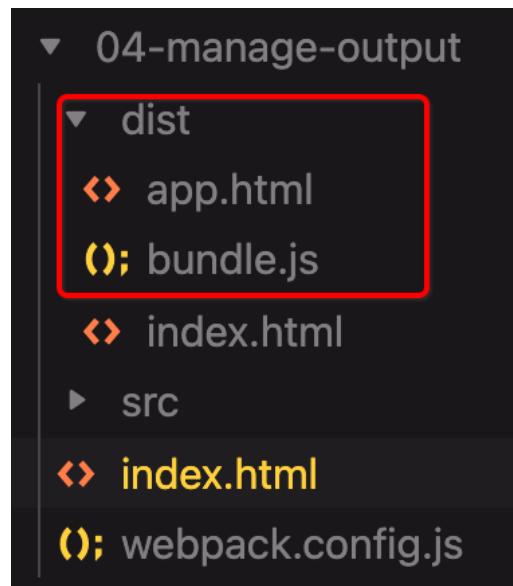
04-manage-output/webpack.config.js

```
//...
module.exports = {
//...

plugins: [
  // 实例化 html-webpack-plugin 插件
  new HtmlWebpackPlugin({
    template: './index.html', // 打包生成的文件的模板
    filename: 'app.html', // 打包生成的文件名称。默认为
    index.html
    inject: 'body' // 设置所有资源文件注入模板的位置。可以设置的值
    true | 'head' | 'body' | false, 默认值为 true
  })
]
```

打包：

```
[felix] 04-manage-output $ npx webpack
asset bundle.js 3.15 KiB [compared for emit] (name: main)
asset app.html 414 bytes [emitted]
runtime modules 670 bytes 3 modules
cacheable modules 180 bytes
./src/index.js 77 bytes [built] [code generated]
./src/hello-world.js 103 bytes [built] [code generated]
webpack 5.54.0 compiled successfully in 95 ms
```



查看 `app.html` 内容：

04-manage-output/dist/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>千锋大前端教研院 - webpack5 学习指南</title>
</head>
<body>
<script defer src="bundle.js"></script></body>
</html>
```

这次打包应用到了我们的模板文件 `index.html`, 并且生成了新的文件 `app.html`, 文件里自动引用的 `bundle.js` 也从 `<header>` 迁移到了 `<body>` 里。

1.3.3 清理dist

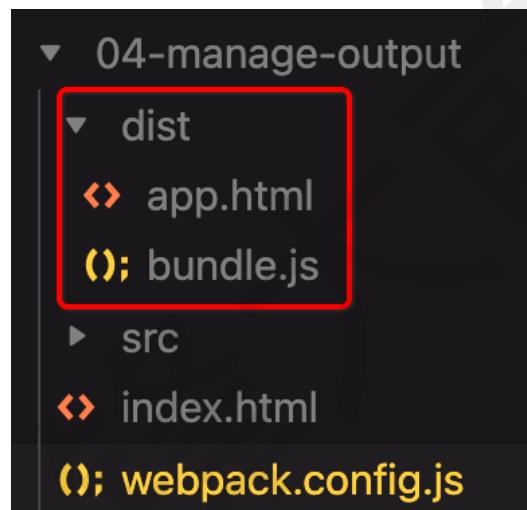
仔细留意一下，我们发现 `dist/index.html` 仍旧存在，这个文件是上次生成的残留文件，已经没有用了。可见，webpack 将生成文件并放置在 `/dist` 文件夹中，但是它不会追踪哪些文件是实际在项目中用到的。通常比较推荐的做法是，在每次构建前清理 `/dist` 文件夹，这样只会生成用到的文件。让我们使用 `output.clean` 配置项实现这个需求。

```
output: {  
    // 打包前清理 dist 文件夹  
    clean: true  
}
```

04-manage-output/webpack.config.js

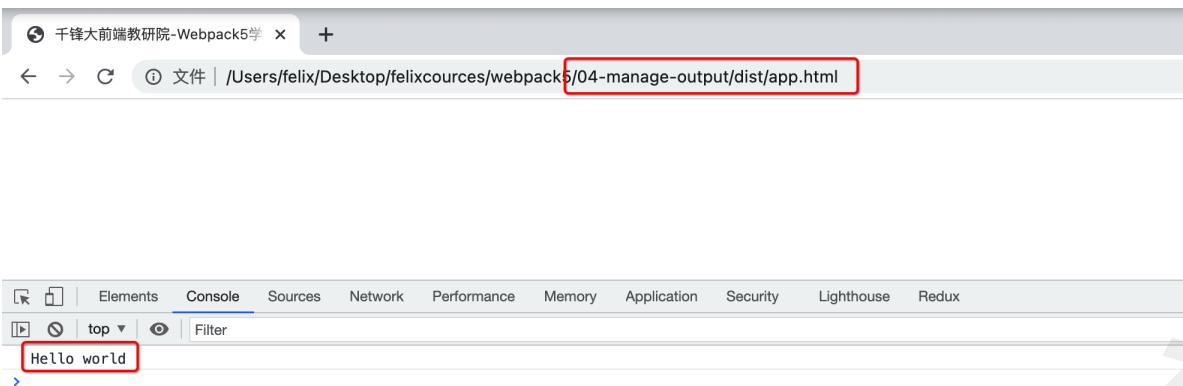
```
//...  
module.exports = {  
//...  
  
output: {  
    //...  
  
    // 打包前清理 dist 文件夹  
    clean: true  
},  
  
//...  
}  
}
```

再次打包：



检查 /dist 文件夹。现在只会看到构建后生成的文件，而没有旧文件！

最后，在浏览器里运行我们打包好的页面：



1.4 搭建开发环境

截止目前，我们只能通过复制 `dist/index.html` 完整物理路径到浏览器地址栏里访问页面。现在来看看如何设置一个开发环境，使我们的开发体验变得更轻松一些。

1.4.1 mode 选项

在开始前，我们先将 `mode` 设置为 `'development'`

```
module.exports = {
  // 开发模式
  mode: 'development',
}
```

05-development/webpack.config.js

```
//...
module.exports = {
//...

// 开发模式
mode: 'development',

//...
}
```

1.4.2 使用 source map

当 webpack 打包源代码时，可能会很难追踪到 error(错误) 和 warning(警告) 在源代码中的原始位置。例如，如果将三个源文件 (`a.js`, `b.js` 和 `c.js`) 打包到一个 bundle (`bundle.js`) 中，而其中一个源文件包含一个错误，那么堆栈跟踪就会直接指向到 `bundle.js`。你可能需要准确地知道错误来自于哪个源文件，所以这种提示这通常不会提供太多帮助。

为了更容易地追踪 error 和 warning, JavaScript 提供了 [source maps](#) 功能, 可以将编译后的代码映射回原始源代码。如果一个错误来自于 `b.js`, source map 就会明确的告诉你。

在本篇中, 我们将使用 `inline-source-map` 选项:

```
module.exports = {
  // 在开发模式下追踪代码
  devtool: 'inline-source-map',
}
```

05-development/webpack.config.js

```
//...
module.exports = {
//...

// 在开发模式下追踪代码
devtool: 'inline-source-map',

//...
}
```

现在, 让我们来做一些调试, 在 `src/hello-world.js` 文件中生成一个错误:

05-development/src/hello-world.js

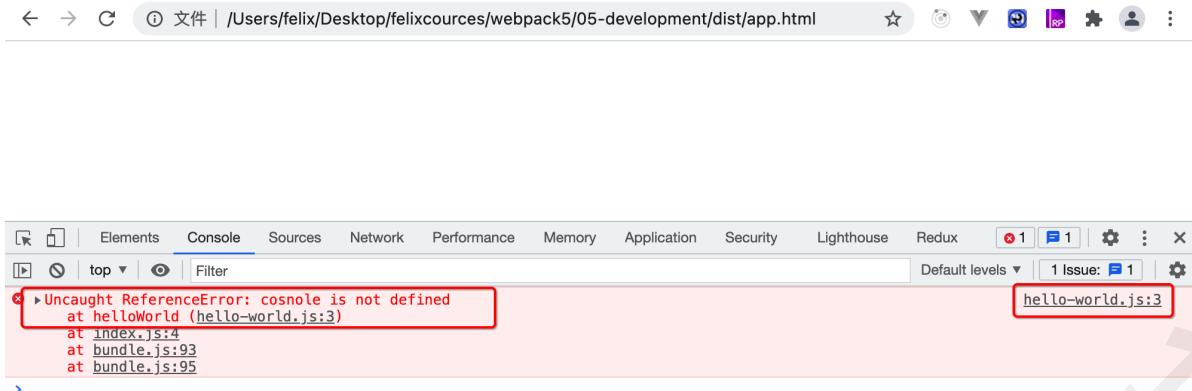
```
function helloworld() {
  // console 单词拼写错误
  cosnole.log('Hello world')
}

// 导出函数模块
export default helloworld
```

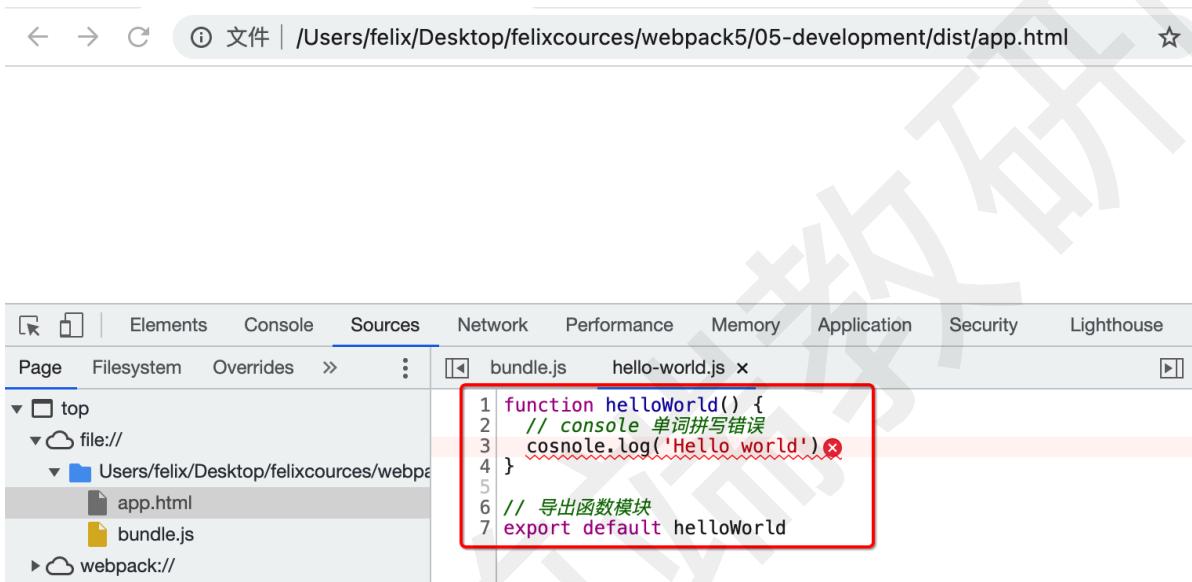
再次编译:

```
[felix] 05-development $ npx webpack
```

现在, 在浏览器中打开生成的 `index.html` 文件, 并且在控制台查看显示的错误。错误如下:



在浏览器里点击 `hello-world.js:3`, 查看具体错误:



我们可以精确定位错误的行数。

1.4.3 使用 watch mode(观察模式)

在每次编译代码时，手动运行 `npx webpack` 会显得很麻烦。

我们可以在 webpack 启动时添加 "watch" 参数。如果其中一个文件被更新，代码将被重新编译，所以你不必再去手动运行整个构建。

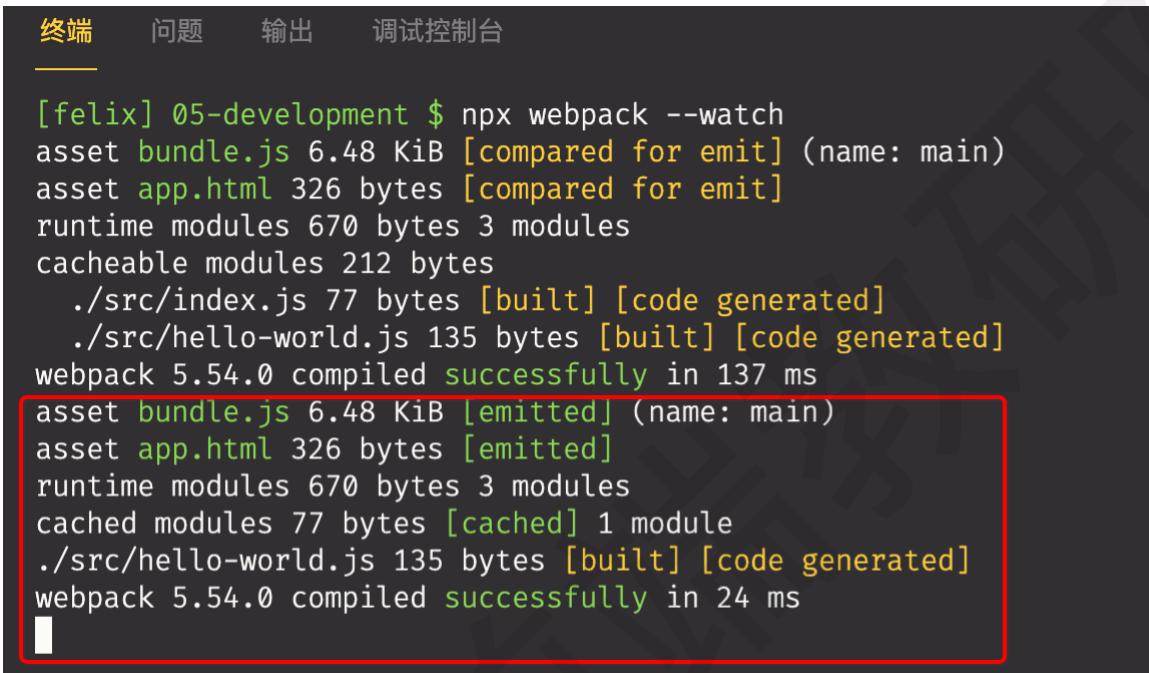
```
[felix] 05-development $ npx webpack --watch
```

```
[felix] 05-development $ npx webpack --watch
asset bundle.js 6.48 KiB [compared for emit] (name: main)
asset app.html 326 bytes [compared for emit]
runtime modules 670 bytes 3 modules
cacheable modules 212 bytes
./src/index.js 77 bytes [built] [code generated]
./src/hello-world.js 135 bytes [built] [code generated]
webpack 5.54.0 compiled successfully in 137 ms
```

现在命令行中，光标停留在尾部，监测文件的变化。修改 `hello-world.js` 文件：

```
function helloworld() {
    console.log('Hello world')
}

// 导出函数模块
export default helloworld
```



```
[felix] 05-development $ npx webpack --watch
asset bundle.js 6.48 KiB [compared for emit] (name: main)
asset app.html 326 bytes [compared for emit]
runtime modules 670 bytes 3 modules
cacheable modules 212 bytes
./src/index.js 77 bytes [built] [code generated]
./src/hello-world.js 135 bytes [built] [code generated]
webpack 5.54.0 compiled successfully in 137 ms
asset bundle.js 6.48 KiB [emitted] (name: main)
asset app.html 326 bytes [emitted]
runtime modules 670 bytes 3 modules
cached modules 77 bytes [cached] 1 module
./src/hello-world.js 135 bytes [built] [code generated]
webpack 5.54.0 compiled successfully in 24 ms
```

现在，保存文件并检查 terminal(终端) 窗口。应该可以看到 webpack 自动地重新编译修改后的模块！

唯一的缺点是，为了看到修改后的实际效果，你需要刷新浏览器。如果能够自动刷新浏览器就更好了，因此接下来我们会尝试通过 `webpack-dev-server` 实现此功能。

1.4.4 使用 webpack-dev-server

`webpack-dev-server` 为你提供了一个基本的 web server，并且具有 live reloading(实时重新加载) 功能。先安装：

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

修改配置文件，告知 dev server，从什么位置查找文件：

```
module.exports = {
  // dev-server
  devServer: {
    static: './dist'
  }
}
```

05-development/webpack.config.js

```
//...
module.exports = {
  //...

  // dev-server
  devServer: {
    static: './dist'
  }
}
```

以上配置告知 `webpack-dev-server`，将 `dist` 目录下的文件作为 web 服务的根目录。

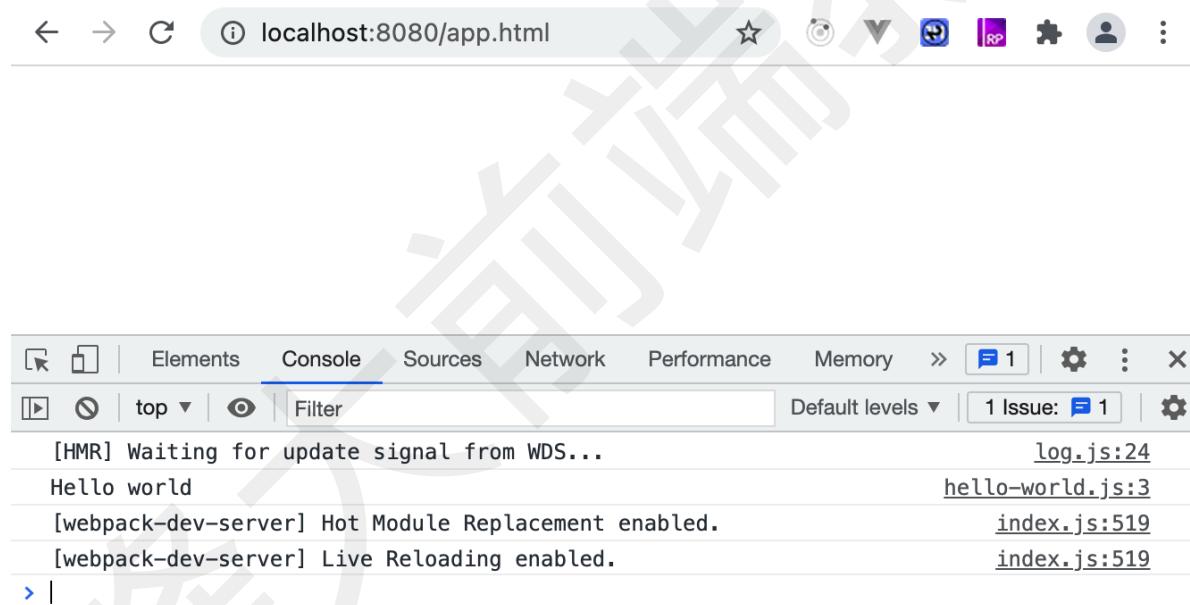
提示：

`webpack-dev-server` 在编译之后不会写入到任何输出文件。而是将 bundle 文件保留在内存中，然后将它们 serve 到 server 中，就好像它们是挂载在 server 根路径上的真实文件一样。

执行命令：

```
终端 问题 输出 调试控制台
[felix] 05-development $ npx webpack serve --open
<i> [webpack-dev-server] Project is running at:
<i> [webpack-dev-server] Loopback: http://localhost:8080/
<i> [webpack-dev-server] On Your Network (IPv4): http://192.168.2.10:8080/
<i> [webpack-dev-server] On Your Network (IPv6): http://[fe80::1]:8080/
<i> [webpack-dev-server] Content not from webpack is served from './dist' directory
<i> [webpack-dev-middleware] wait until bundle finished: /
asset bundle.js 670 KiB [emitted] (name: main)
asset app.html 326 bytes [emitted]
runtime modules 27 KiB 13 modules
modules by path ../node_modules/ 198 KiB
  modules by path ../node_modules/webpack-dev-server/client/ 51.8 KiB 12 modules
  modules by path ../node_modules/webpack/hot/*.js 4.3 KiB 4 modules
  modules by path ../node_modules/html-entities/lib/*.js 81.3 KiB 4 modules
  modules by path ../node_modules/querystring/*.js 4.51 KiB 3 modules
  modules by path ../node_modules/url/*.js 23.1 KiB
    ../node_modules/url/url.js 22.8 KiB [built] [code generated]
    ../node_modules/url/util.js 314 bytes [built] [code generated]
  ../node_modules/ansi-html-community/index.js 4.16 KiB [built] [code generated]
  ../node_modules/events/events.js 14.5 KiB [built] [code generated]
  ../node_modules/punycode/punycode.js 14.3 KiB [built] [code generated]
modules by path ./src/*.js 212 bytes
  ./src/index.js 77 bytes [built] [code generated]
  ./src/hello-world.js 135 bytes [built] [code generated]
webpack 5.54.0 compiled successfully in 518 ms
```

在浏览器里可以直接访问页面：



修改一下 `hello-world.js` 文件：

05-development/src/hello-world.js

```
function helloworld() {
  console.log('Hello world~~~')
}

// 导出函数模块
export default helloworld
```

这时我们不用刷新浏览器页面，在控制台上能看到 `Hello world~~~` 自动更新了。

1.5 资源模块

目前为止，我们的项目可以在控制台上显示 "Hello world~~~"。现在我们尝试混合一些其他资源，比如 images，看看 webpack 如何处理。

在 webpack 出现之前，前端开发人员会使用 [grunt](#) 和 [gulp](#) 等工具来处理资源，并将它们从 `/src` 文件夹移动到 `/dist` 或 `/build` 目录中。webpack 最出色的功能之一就是，除了引入 JavaScript，还可以内置的资源模块 [Asset Modules](#) 引入任何其他类型的文件。

资源模块(asset module)是一种模块类型，它允许我们应用Webpack来打包其他资源文件（如字体，图标等）

资源模块类型(asset module type)，通过添加 4 种新的模块类型，来替换所有这些 loader：

- `asset/resource` 发送一个单独的文件并导出 URL。
- `asset/inline` 导出一个资源的 data URI。
- `asset/source` 导出资源的源代码。
- `asset` 在导出一个 data URI 和发送一个单独的文件之间自动选择。

1.5.1 Resource 资源

修改 `webpack.config.js` 配置：

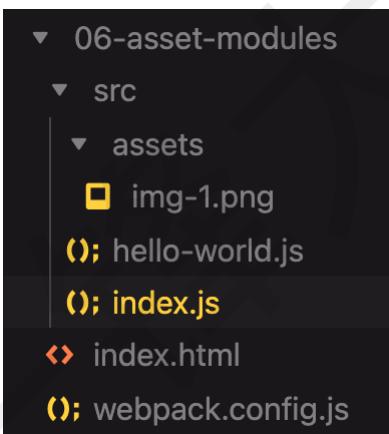
```
// 配置资源文件
module: {
  rules: [{
    test: /\.png/,
    type: 'asset/resource'
  }]
},
```

06-asset-modules/webpack.config.js

```
const path = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
  //...

  // 配置资源文件
  module: {
    rules: [{
      test: /\.png/,
      type: 'asset/resource'
    }]
  },
  //...
}
```

准备资源文件，在入口文件中引入，并显示在页面上：



06-asset-modules/src/index.js

```
// 导入函数模块
import helloworld from './hello-world.js'
import imgsrc from './assets/img-1.png'

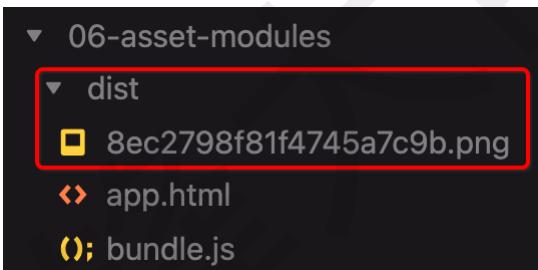
helloworld()

const img = document.createElement('img')
img.src = imgsrc
document.body.appendChild(img)
```

执行打包命令：

```
[felix] 06-asset-modules $ npx webpack
asset 8ec2798f81f4745a7c9b.png 101 KiB [emitted] [immutable]
[from: src/assets/img-1.png] (auxiliary name: main)
asset bundle.js 10.5 KiB [emitted] (name: main)
asset app.html 326 bytes [emitted]
runtime modules 1.72 KiB 5 modules
cacheable modules 388 bytes (javascript) 101 KiB (asset)
./src/index.js 208 bytes [built] [code generated]
./src/hello-world.js 138 bytes [built] [code generated]
./src/assets/img-1.png 42 bytes (javascript) 101 KiB (asset)
[built] [code generated]
webpack 5.54.0 compiled successfully in 114 ms
```

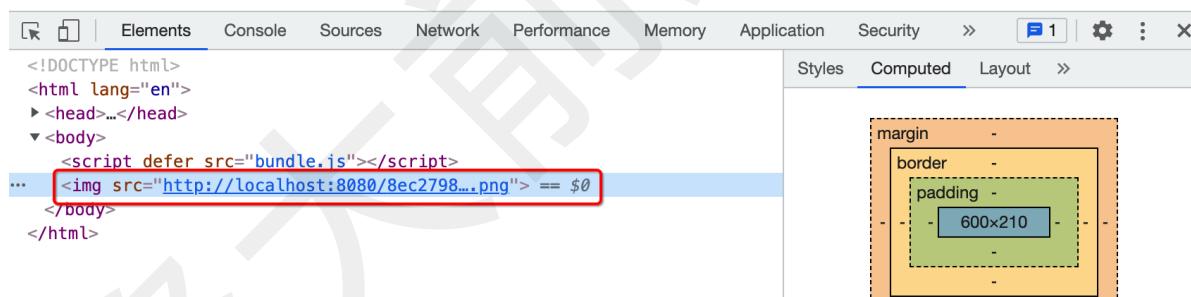
发现图片(.png)文件已经打包到了dist目录下：



执行启动服务命令：

```
[felix] 06-asset-modules $ npx webpack serve --open
<i> [webpack-dev-server] Project is running at:
<i> [webpack-dev-server] Loopback: http://localhost:8080/
<i> [webpack-dev-server] On Your Network (IPv4): http://192.168.2.10:8080/
<i> [webpack-dev-server] On Your Network (IPv6): http://[fe80::1]:8080/
<i> [webpack-dev-server] Content not from webpack is served from './dist' directory
<i> [webpack-dev-middleware] wait until bundle finished: /
asset bundle.js 671 KiB [emitted] (name: main)
asset 8ec2798f81f4745a7c9b.png 101 KiB [emitted] [immutable] [from: src/assets/img-1.png] (auxiliary name: main)
asset app.html 326 bytes [emitted]
runtime modules 27 KiB 13 modules
modules by path ../node_modules/ 198 KiB
  modules by path ../node_modules/webpack-dev-server/client/ 51.8 KiB 12 modules
  modules by path ../node_modules/webpack/hot/*.js 4.3 KiB 4 modules
  modules by path ../node_modules/html-entities/lib/*.js 81.3 KiB 4 modules
  modules by path ../node_modules/querystring/*.js 4.51 KiB 3 modules
  modules by path ../node_modules/url/*.js 23.1 KiB 2 modules
  ../node_modules/ansi-html-community/index.js 4.16 KiB [built] [code generated]
  ../node_modules/events/events.js 14.5 KiB [built] [code generated]
  ../node_modules/punycode/punycode.js 14.3 KiB [built] [code generated]
modules by path ./src/ 388 bytes (javascript) 101 KiB (asset)
./src/index.js 208 bytes [built] [code generated]
./src/hello-world.js 138 bytes [built] [code generated]
./src/assets/img-1.png 42 bytes (javascript) 101 KiB (asset) [built] [code generated]
webpack 5.54.0 compiled successfully in 593 ms
```

打开浏览器：



- 自定义输出文件名

默认情况下，`asset/resource` 模块以 `[contenthash][ext][query]` 文件名发送到输出目录。

可以通过在 webpack 配置中设置 `output.assetModuleFilename` 来修改此模板字符串：

```
output: {
  assetModuleFilename: 'images/[contenthash][ext][query]'
},
```

06-asset-modules/webpack.config.js

```
//...
module.exports = {
//...

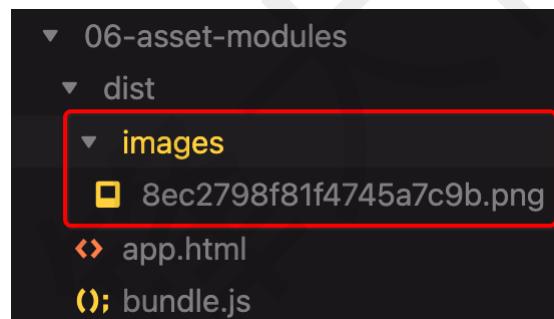
output: {
//...

assetModuleFilename: 'images/[contenthash][ext][query]'
},

//...
}
```

执行编译：

```
[felix] 06-asset-modules $ npx webpack
assets by status 101 KiB [cached] 1 asset
asset bundle.js 10.5 KiB [compared for emit] (name: main)
asset app.html 326 bytes [compared for emit]
runtime modules 1.72 KiB 5 modules
cacheable modules 356 bytes (javascript) 101 KiB (asset)
./src/index.js 208 bytes [built] [code generated]
./src/hello-world.js 106 bytes [built] [code generated]
./src/assets/img-1.png 42 bytes (javascript) 101 KiB (asset)
[built] [code generated]
webpack 5.54.0 compiled successfully in 125 ms
```



另一种自定义输出文件名的方式是，将某些资源发送到指定目录，修改配置：

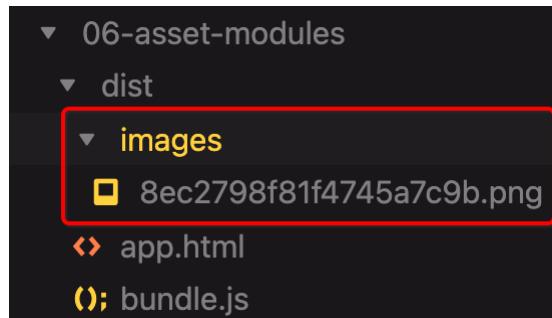
```
rules: [{  
  test: /\.png/,  
  type: 'asset/resource',  
  
  // 优先级高于 assetModuleFilename  
  generator: {  
    filename: 'images/[contenthash][ext][query]'  
  }  
}]
```

06-asset-modules/webpack.config.js

```
//...  
module.exports = {  
//...  
  
output: {  
  //...  
  
  // 配置资源文件  
  module: {  
    rules: [  
      {  
        test: /\.png/,  
        type: 'asset/resource',  
  
        // 优先级高于 assetModuleFilename  
        generator: {  
          filename: 'images/[contenthash][ext][query]'  
        }  
      }  
    ],  
    }  
  },  
  //...  
}
```

执行编译:

```
[felix] 06-asset-modules $ npx webpack
assets by status 102 KiB [cached] 2 assets
assets by path . 10.5 KiB
  asset bundle.js 10.5 KiB [compared for emit] (name: main)
  asset app.html 71 bytes [compared for emit]
runtime modules 1.72 KiB 5 modules
cacheable modules 356 bytes (javascript) 101 KiB (asset)
  ./src/index.js 208 bytes [built] [code generated]
  ./src/hello-world.js 106 bytes [built] [code generated]
  ./src/assets/img-1.png 42 bytes (javascript) 101 KiB (asset)
[built] [code generated]
webpack 5.54.0 compiled successfully in 113 ms
```



输出结果与 `assetModuleFilename` 设置一样。

1.5.2 inline 资源

修改 `webpack.config.js` 配置：

```
// 配置资源文件
module: [
  {
    test: /\.svg$/,
    type: 'asset/inline'
  }
]
```

06-asset-modules/webpack.config.js

```
//...
module.exports = {
//...

// 配置资源文件
module: {
```

```

rules: [
  //...
  {
    test: /\.svg/,
    type: 'asset/inline'
  }
],
},
//...
}

```

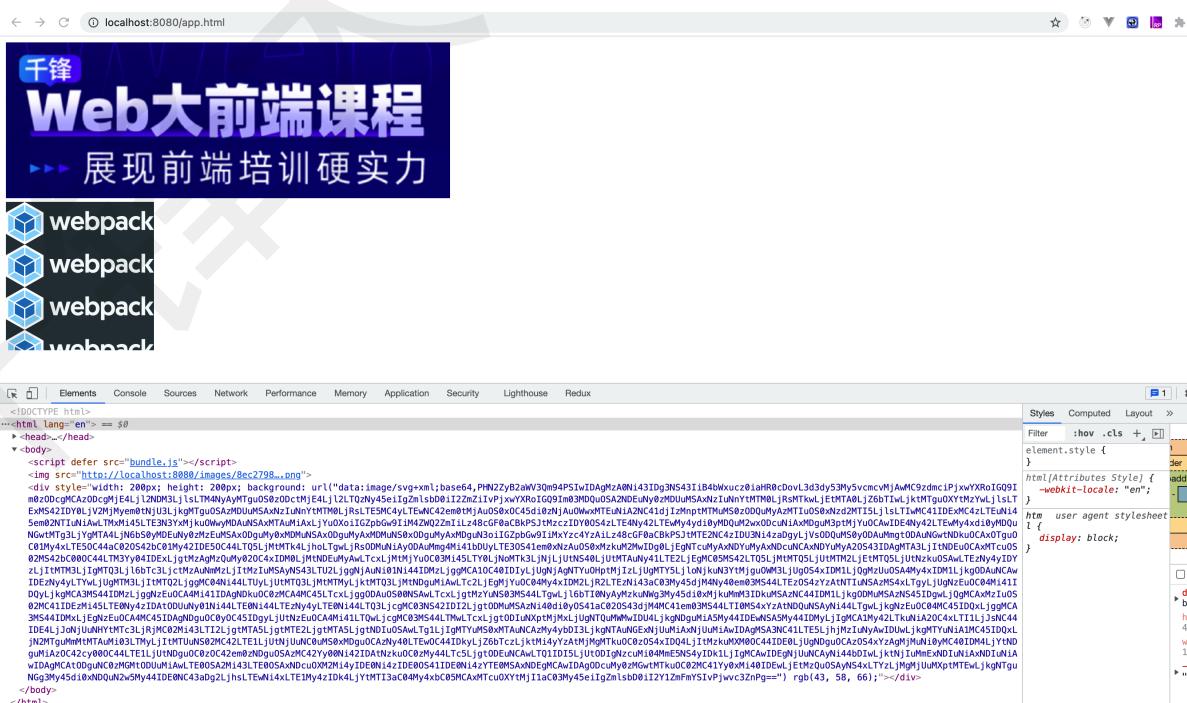
执行启动服务命令：

```

[felix] 06-asset-modules $ npx webpack serve --open
<i> [webpack-dev-server] Project is running at:
<i> [webpack-dev-server] Loopback: http://localhost:8080/
<i> [webpack-dev-server] On Your Network (IPv4): http://192.168.2.10:8080/
<i> [webpack-dev-server] On Your Network (IPv6): http://[fe80::1]:8080/
<i> [webpack-dev-server] Content not from webpack is served from './dist' directory
<i> [webpack-dev-middleware] wait until bundle finished: /
asset bundle.js 675 KiB [emitted] (name: main)
asset images/8ec2798f81f4745a7c9b.png 101 KiB [emitted] [immutable] [from: src/assets/img-1.png] (auxiliary name: main)
asset app.html 326 bytes [emitted]
runtime modules 27 KiB 13 modules
modules by path ..../node_modules/ 198 KiB
  modules by path ../node_modules/webpack-dev-server/client/ 51.8 KiB 12 modules
    modules by path ../node_modules/webpack/hot/*.js 4.3 KiB 4 modules
    modules by path ../node_modules/html-entities/lib/*.js 81.3 KiB 4 modules
    modules by path ../node_modules/queryString/*.js 4.51 KiB 3 modules
    modules by path ../node_modules/url/*.js 23.1 KiB 2 modules
  3 modules
modules by path ./src/ 3.55 KiB (javascript) 101 KiB (asset)
  javascript modules 529 bytes
    ./src/index.js 423 bytes [built] [code generated]
    ./src/hello-world.js 106 bytes [built] [code generated]
  asset modules 3.03 KiB (javascript) 101 KiB (asset)
    ./src/assets/img-1.png 42 bytes (javascript) 101 KiB (asset) [built] [code generated]
    ./src/assets/webpack-logo.svg 2.99 KiB [built] [code generated]
webpack 5.54.0 compiled successfully in 513 ms

```

打开浏览器：



可见，`.svg` 文件都将作为 data URI 注入到 bundle 中。

- 自定义 data URI 生成器

webpack 输出的 data URI，默认是呈现为使用 Base64 算法编码的文件内容。

如果要使用自定义编码算法，则可以指定一个自定义函数来编码文件内容。

安装自定义函数模块：

```
[felix] webpack5 $ npm install mini-svg-data-uri -D
```

修改配置文件：

```
const svgToMiniDataURI = require('mini-svg-data-uri')

rules: [
  {
    test: /\.svg/,
    type: 'asset/inline',
    generator: {
      dataUrl: content => {
        content = content.toString();
        return svgToMiniDataURI(content);
      }
    }
  }
]
```

06-asset-modules/webpack.config.js

```
//...
const svgToMiniDataURI = require('mini-svg-data-uri')

module.exports = {
//...

// 配置资源文件
module: {
  rules: [
//...
  {
    test: /\.svg/,
    type: 'asset/inline',
    generator: {
      dataUrl: content => {
        content = content.toString();
        return svgToMiniDataURI(content);
      }
    }
  }
]
```

```
        }
    }
}
],
},
//...
}
```

现在，所有 `.svg` 文件都将通过 `mini-svg-data-uri` 包进行编码。重新启动服务，在浏览器查看效果：



1.5.3 source 资源

source资源，导出资源的源代码。修改配置文件，添加：

```
module: {
  rules: [
    test: /\.txt$/,
    type: 'asset/source'
  ]
}
```

06-asset-modules/webpack.config.js

```
//...

module.exports = {
//...

// 配置资源文件
module: {
  rules: [
    //...
  ]
}
```

```
        test: /\.txt/,
        type: 'asset/source'
    }
],
},
//...
}
```

在assets里创建一个 example.txt 文件:

```
06-asset-modules/src/assets/example.txt
```

```
hello webpack
```

在入口文件里引入一个 .txt 文件，添加内容:

```
import exampleText from './assets/example.txt'

const block = document.createElement('div')
block.style.cssText = `width: 200px; height: 200px; background:
aliceblue`
block.textContent = exampleText
document.body.appendChild(block)
```

```
06-asset-modules/src/index.js
```

```
// 导入函数模块
//...
import exampleText from './assets/example.txt'

//...

const block = document.createElement('div')
block.style.cssText = `width: 200px; height: 200px;
background: aliceblue`
block.textContent = exampleText
document.body.appendChild(block)

//...
```

启动服务，打开浏览器:



所有 `.txt` 文件将原样注入到 bundle 中。

1.5.4 通用资源类型

通用资源类型 `asset`，在导出一个 data URI 和发送一个单独的文件之间自动选择。

修改配置文件：

```
module: {  
  rules: [  
    test: /\.jpg/,  
    type: 'asset'  
  ]  
}
```

现在，webpack 将按照默认条件，自动地在 `resource` 和 `inline` 之间进行选择：小于 8kb 的文件，将会视为 `inline` 模块类型，否则会被视为 `resource` 模块类型。

可以通过在 webpack 配置的 module rule 层级中，设置

[`Rule.parser.dataurlCondition.maxsize`](#) 选项来修改此条件：

```
rules: [
  {
    test: /\.jpg/,
    type: 'asset',
    parser: {
      dataUrlCondition: {
        maxSize: 4 * 1024 // 4kb
      }
    }
  }
]
```

06-asset-modules/webpack.config.js

```
//...

module.exports = {
//...

// 配置资源文件
module: {
  rules: [
    //...

    {
      test: /\.jpg/,
      type: 'asset',
      parser: {
        dataUrlCondition: {
          maxSize: 4 * 1024 // 4kb
        }
      }
    }
  ],
}

//...
}
```

在 assets 目录下创建 .jpg 文件，然后在入口文件中引入：

```
import jpgMap from './assets/qianfeng-sem.jpg'

const img3 = document.createElement('img')
img3.style.cssText = 'width: 600px; height: 240px; display:
block'
img3.src = jpgMap
document.body.appendChild(img3)
```

06-asset-modules/src/index.js

```
// 导入函数模块
//...
import jpgMap from './assets/qianfeng-sem.jpg'

//...
const img3 = document.createElement('img')
img3.style.cssText = 'width: 600px; height: 240px; display:
block'
img3.src = jpgMap
document.body.appendChild(img3)
```

启动服务，打开浏览器：



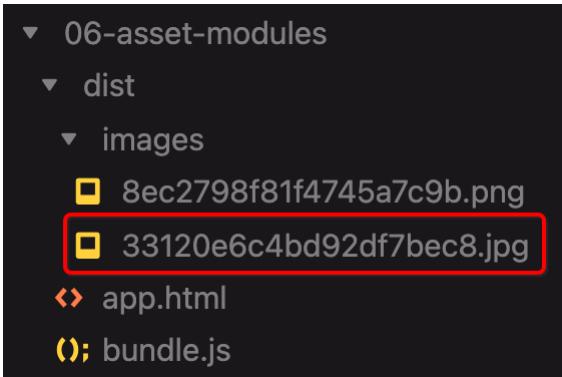
hello webpack



执行编译命令：

```
[felix] 06-asset-modules $ npx webpack
assets by status 101 KiB [cached] 1 asset
assets by status 653 KiB [emitted]
  asset images/33120e6c4bd92df7bec8.jpg 637 KiB [emitted]
  [immutable] [from: src/assets/qianfeng-sem.jpg] (auxiliary name:
  main)
  asset bundle.js 15.7 KiB [emitted] (name: main)
  asset app.html 326 bytes [compared for emit]
  runtime modules 1.72 KiB 5 modules
  cacheable modules 4.03 KiB (javascript) 738 KiB (asset)
  asset modules 3.1 KiB (javascript) 738 KiB (asset)
    ./src/assets/img-1.png 42 bytes (javascript) 101 KiB (asset)
  [built] [code generated]
    ./src/assets/webpack-logo.svg 2.99 KiB [built] [code
  generated]
    ./src/assets/example.txt 25 bytes [built] [code generated]
```

```
./src/assets/qianfeng-sem.jpg 42 bytes (javascript) 637 KiB
(asset) [built] [code generated]
  javascript modules 949 bytes
    ./src/index.js 843 bytes [built] [code generated]
    ./src/hello-world.js 106 bytes [built] [code generated]
webpack 5.54.0 compiled successfully in 139 ms
```



发现当前的 `.jpg` 文件被打包成了单独的文件，因为此文件大小超过了 `4kb`。

1.6 管理资源

在上一章，我们讲解了四种资源模块引入外部资源。除了资源模块，我们还可以通过 `loader` 引入其他类型的文件。

1.6.1 什么是loader

webpack 只能理解 JavaScript 和 JSON 文件，这是 webpack 开箱可用的自带能力。`loader` 让 webpack 能够去处理其他类型的文件，并将它们转换为有效 `模块`，以供应用程序使用，以及被添加到依赖图中。

在 webpack 的配置中，`loader` 有两个属性：

1. `test` 属性，识别出哪些文件会被转换。
2. `use` 属性，定义出在进行转换时，应该使用哪个 loader。

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

module.exports = {
  output: {
    filename: 'my-first-webpack.bundle.js',
  },
  module: {
    rules: [{ test: /\.txt$/, use: 'raw-loader' }],
  },
};
```

以上配置中，对一个单独的 module 对象定义了 `rules` 属性，里面包含两个必须属性：`test` 和 `use`。这告诉 webpack 编译器(compiler) 如下信息：

“嘿，webpack 编译器，当你碰到「在 `require()`/`import` 语句中被解析为 `'.txt'` 的路径」时，在你对它打包之前，先 `use(使用)` `raw-loader` 转换一下。”

1.6.2 加载CSS

为了在 JavaScript 模块中 `import` 一个 CSS 文件，你需要安装 [style-loader](#) 和 [css-loader](#)，并在 `module` 配置 中添加这些 loader：

```
[felix] webpack5 $ npm install --save-dev style-loader css-loader
```

修改配置文件：

```
module: {  
  rules: [  
    {  
      test: /\.css$/i,  
      use: ['style-loader', 'css-loader'],  
    },  
  ],  
},
```

07-manage-assets/webpack.config.js

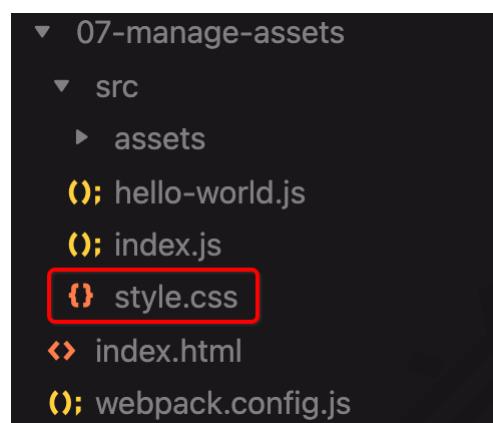
```
//...  
  
module.exports = {  
//...  
  
// 配置资源文件  
module: {  
  rules: [  
    //...  
  
    {  
      test: /\.css$/i,  
      use: ['style-loader', 'css-loader'],  
    },  
  ],  
},  
  
//...  
}
```

模块 loader 可以链式调用。链中的每个 loader 都将对资源进行转换。链会逆序执行。第一个 loader 将其结果（被转换后的资源）传递给下一个 loader，依此类推。最后，webpack 期望链中的最后的 loader 返回 JavaScript。

应保证 loader 的先后顺序：'`'style-loader'` 在前，而 '`'css-loader'` 在后。如果不遵守此约定，webpack 可能会抛出错误。webpack 根据正则表达式，来确定应该查找哪些文件，并将其提供给指定的 loader。在这个示例中，所有以 `.css` 结尾的文件，都将被提供给 `style-loader` 和 `css-loader`。

这使你可以在依赖于此样式的 js 文件中 `import './style.css'`。现在，在此模块执行过程中，含有 CSS 字符串的 `<style>` 标签，将被插入到 html 文件的 `<head>` 中。

我们尝试一下，通过在项目中添加一个新的 `style.css` 文件，并将其 import 到我们的 `index.js` 中：



07-manage-assets/src/style.css

```
.hello {  
  color: #f9efd4;  
}
```

在入口文件里导入 `.css` 文件：

```
import './style.css'  
  
document.body.classList.add('hello')
```

07-manage-assets/src/index.js

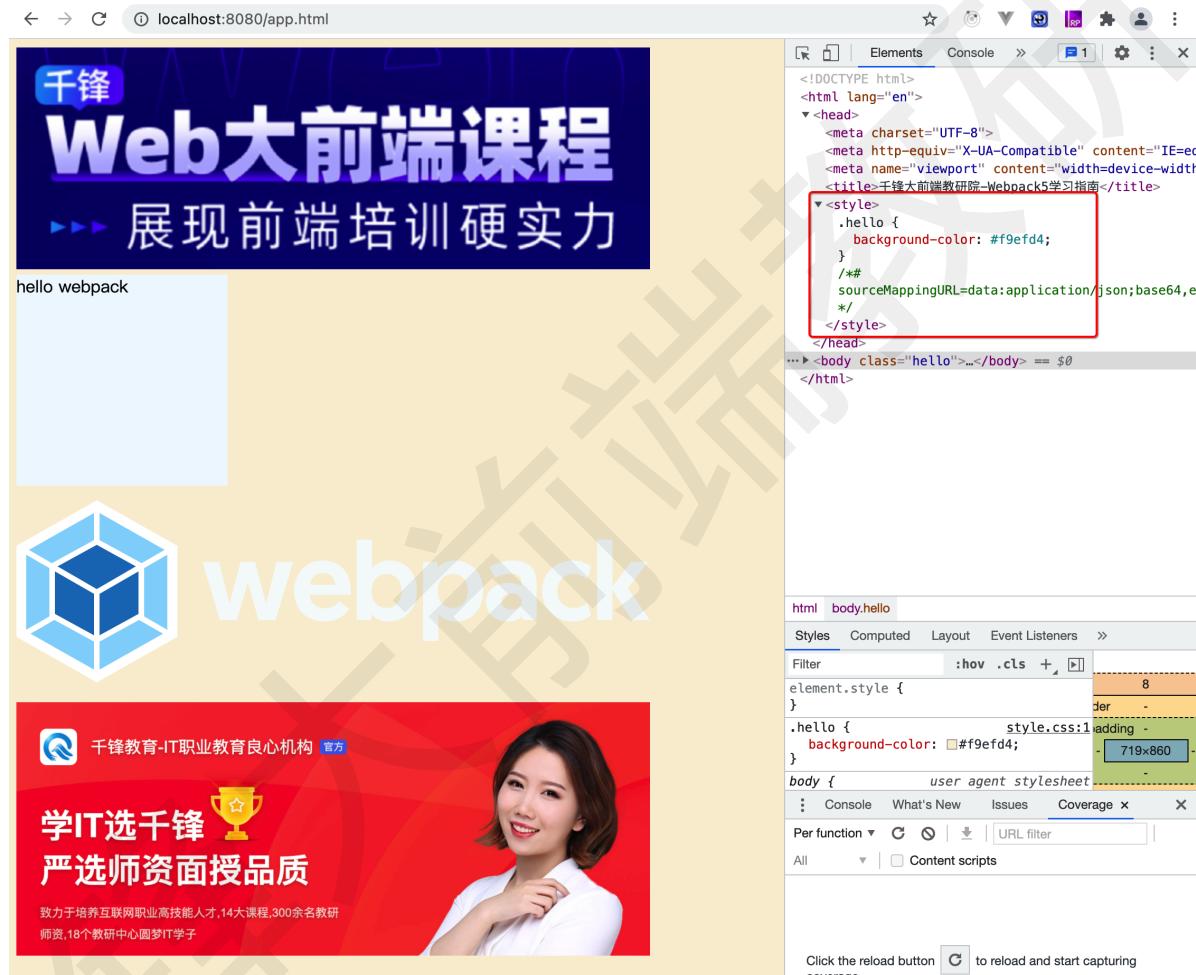
```
// 导入函数模块
//...
import './style.css'

//...

document.body.classList.add('hello')
```

启动服务，打开浏览器：

```
[felix] 07-manage-assets $ npx webpack serve --open
```



你应该看到页面背景颜色是浅黄色。要查看 webpack 做了什么，请检查页面（不要查看页面源代码，它不会显示结果，因为 `<style>` 标签是由 JavaScript 动态创建的），并查看页面的 head 标签，包含 style 块元素，也就是我们在 `index.js` 中 import 的 css 文件中的样式。

现有的 loader 可以支持任何你可以想到的 CSS 风格 - [sass](#) 和 [less](#) 等。安装less-loader：

```
[felix] webpack5 $ npm install less less-loader --save-dev
```

修改配置文件：

```
module: {
  rules: [
    {
      test: /\.less$/i,
      use: ['style-loader', 'css-loader', 'less-loader'],
    }
  ],
},
```

07-manage-assets/webpack.config.js

```
//...
module.exports = {
//...

// 配置资源文件
module: {
  rules: [
    //...

    {
      test: /\.less$/i,
      use: ['style-loader', 'css-loader', 'less-loader'],
    },
    //...
  ],
},
//...
}
```

在项目 `src` 目录下创建 `style.less` 文件:

```
@color: red;
.world {
  color: @color;
}
```

在入口文件中引入 `.less` 文件:

```
import './style.less'

document.body.classList.add('world')
```

07-manage-assets/src/index.js

```
// 导入模块
//...
import './style.less'

//...

document.body.classList.add('world')
```



The screenshot shows the Chrome DevTools Elements tab with the page's HTML code. A red box highlights the CSS rule `.world { color: red; }` located in the head section's style block. The right panel shows the CSS Styles tab with a list of styles applied to elements on the page, including `.world` and `.hello`.

由预览的效果可见，页面的文字都添加了“红色”的样式。

1.6.3 抽离和压缩CSS

在多数情况下，我们也可以进行压缩CSS，以便在生产环境中节省加载时间，同时还可以将CSS文件抽离成一个单独的文件。实现这个功能，需要 `mini-css-extract-plugin` 这个插件来帮忙。安装插件：

```
[felix] webpack5 $ npm install mini-css-extract-plugin --save-dev
```

本插件会将 CSS 提取到单独的文件中，为每个包含 CSS 的 JS 文件创建一个 CSS 文件，并且支持 CSS 和 SourceMaps 的按需加载。

本插件基于 webpack v5 的新特性构建，并且需要 webpack 5 才能正常工作。

之后将 loader 与 plugin 添加到你的 `webpack` 配置文件中：

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin")

module: {
  rules: [
    {
      test: /\.css$/i,
      use: [MiniCssExtractPlugin.loader, 'css-loader'],
    },
  ]
}
```

07-manage-assets/webpack.config.js

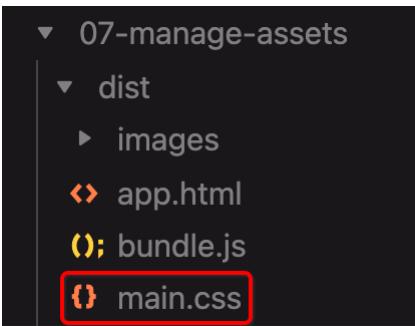
```
//...
const MiniCssExtractPlugin = require("mini-css-extract-
plugin")

module.exports = {
//...

// 配置资源文件
module: {
  rules: [
    {
      test: /\.css$/i,
      use: [MiniCssExtractPlugin.loader, 'css-loader'],
    },
    //...
  ],
},
//...
}
```

执行编译：

```
[felix] 07-manage-assets $ npx webpack
```



单独的 `mini-css-extract-plugin` 插件不会将这些 CSS 加载到页面中。这里 [html-webpack-plugin](#) 帮助我们自动生成 `link` 标签或者在创建 `index.html` 文件时使用 `link` 标签。

07-manage-assets/dist/app.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>千锋大前端教研院-webpack5学习指南</title>
<link href="main.css" rel="stylesheet"></head>
<body>
<script defer src="bundle.js"></script></body>
</html>
```

这时，`link` 标签已经生成出来了，把我们打包好的 `main.css` 文件加载进来。我们发现，`main.css` 文件被打包抽离到 `dist` 根目录下，能否将其打包到一个单独的文件夹里呢？修改配置文件：

```
plugins: [
  new MiniCssExtractPlugin({
    filename: 'styles/[contenthash].css'
  })
],
```

07-manage-assets/webpack.config.js

```
//...

const MiniCssExtractPlugin = require("mini-css-extract-plugin")

module.exports = {
```

```
//...

plugins: [
  //...

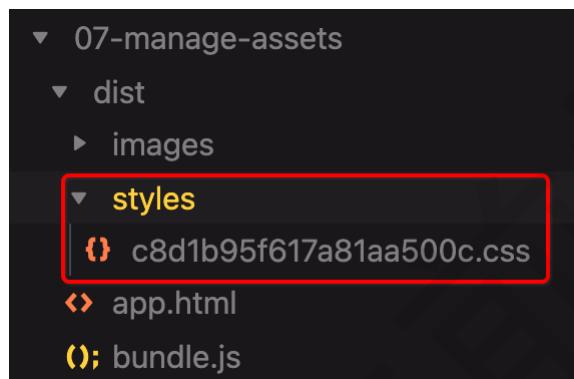
  new MiniCssExtractPlugin({
    filename: 'styles/[contenthash].css'
  })
],

//...
}
```

再次执行编译：

```
[felix] 07-manage-assets $ npx webpack
```

查看打包完成后的目录和文件：



07-manage-assets/dist/app.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>千锋大前端教研院-webpack5学习指南</title>
<link href="styles/c8d1b95f617a81aa500c.css" rel="stylesheet"></head>
<body>
<script defer src="bundle.js"></script></body>
</html>
```

现在，`app.html`文件引用的路径同样更新了。

打开查看 .css 文件:

07-manage-assets/dist/styles/c8d1b95f617a81aa500c.css

```
/*! ****!*\\
!*** css ./node_modules/css-
loader/dist/cjs.js!./src/style.css ***!
\*****
*****/
.hello {
    background-color: #f9efd4;
}

/*# sourceMappingURL=data:application/json;charset=utf-
8;base64,eyJ2ZXJzaW9uIjozLCJmaWx1Ijoic3R5bGVzLzRhMDUzMTlkYWm5
MDJlMjc5ODM5LmNzcyIsIm1hcHBpbmdzIjoiOzs7QUFBQTtFQUNFLHlCQUF5Q
jtBQUMZQixDIiwic291cmNlcyI6WyI3ZWJwYWNrOi8vLy4vc3JjL3N0ewx1Lm
NzcyJdLCJzb3vyY2vzQ29udGvudCI6WyIuaGVsbG8ge1xuICBiYWNrZ3Jvdw5
kLWNvbG9yOiAjzj11zmq0O1xufSJdLCJuYw11cyI6w10sInNvdXJjZVJvb3Qi
oiifQ==*/

```

发现文件并没有压缩和优化，为了压缩输出文件，请使用类似于 [css-minimizer-webpack-plugin](#) 这样的插件。安装插件：

```
[felix] webpack5 $ npm install css-minimizer-webpack-plugin --  
save-dev
```

配置插件：

```
const CssMinimizerPlugin = require('css-minimizer-webpack-plugin')

module.exports = {
    // 生产模式
    mode: 'production',

    // 优化配置
    optimization: {
        minimizer: [
            new CssMinimizerPlugin(),
        ],
    },
}
```

07-manage-assets/webpack.config.js

```
//...

const CssMinimizerPlugin = require('css-minimizer-webpack-
plugin')

module.exports = {
//...

// 开发模式
mode: 'production',

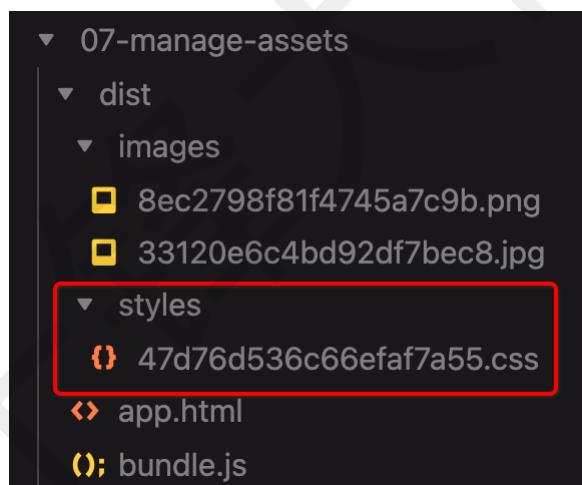
//...

optimization: {
minimizer: [
new CssMinimizerPlugin(),
],
},
}
}
```

再次执行编译：

```
[felix] 07-manage-assets $ npx webpack
```

查看打包完成后的目录和文件：



查看 `47d76d536c66efaf7a55.css` 文件：

07-manage-assets/dist/styles/47d76d536c66efaf7a55.css

```
.hello{background-color:#f9efd4}
/*# sourceMappingURL=data:application/json;charset=utf-
8;base64,eyJ2ZXJzaW9uIjoxLCJmaWxLIjoic3R5bGVzLzQ3ZDc2ZDUzNmM2
NmVmYwY3YTU1LmNzcyIsIm1hcHBpbmdzIjoiQUFBQSxPQUNFLhdCQUNGIiwic
291cmNlcycI6wyJ3ZWJwYWNrOi8vLy4vc3JjL3N0ewx1LmNzcyJdLCJzb3VyY2
vzQ29udGVudCI6wyIuaGVsbG8ge1xuICB1YWNRZ3Jvdw5kLWNvbG9yOiaJZj1
1ZmQ001xufSJdLCJuYW1lcycI6w10sInNvdXJjZVJvb3QioiiifQ==*/
```

css 文件优化成功！

1.6.4 加载 images 图像

假如，现在我们正在下载 CSS，但是像 background 和 icon 这样的图像，要如何处理呢？在 webpack 5 中，可以使用内置的 [Asset Modules](#)，我们可以轻松地将这些内容混入我们的系统中，这个我们在“资源模块”一节中已经介绍了。这里再补充一个知识点，在 `css` 文件里也可以直接引用文件，修改 `style.css` 和入口 `index.js`：

```
.block-bg {
  background-image: url("./assets/webpack-logo.svg");
}
```

```
block.style.cssText = `width: 200px; height: 200px; background-
color: #2b3a42`;
block.classList.add('block-bg')
```

07-manage-assets/src/style.css

```
.hello {
  background-color: #f9efd4;
}

.block-bg {
  background-image: url("./assets/webpack-logo.svg");
}
```

07-manage-assets/src/index.js

```

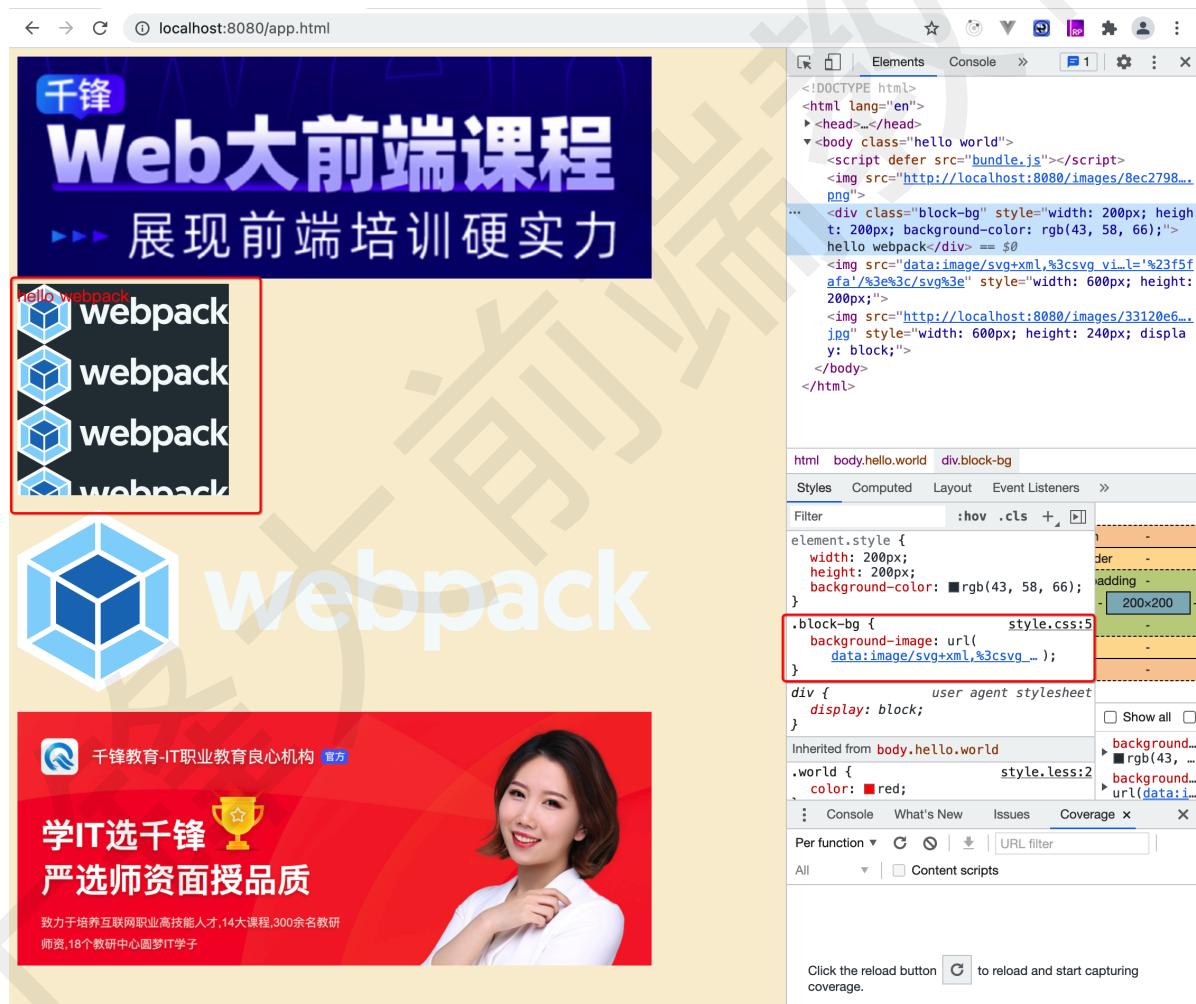
// 导入模块
//...
import './style.css'

//...
block.style.cssText = `width: 200px; height: 200px;
background-color: #2b3a42`;
block.textContent = exampleText
block.classList.add('block-bg')
document.body.appendChild(block)

//...

```

启动服务，打开浏览器：



The screenshot shows a browser window displaying a webpage from `localhost:8080/app.html`. The page content includes a large banner with the text "千锋 Web 大前端课程" and "展现前端培训硬实力". Below the banner, there is a logo consisting of four blue cubes with the word "webpack" next to each one. To the right of the logo, the word "webpack" is repeated in a larger, stylized font. At the bottom left, there is a red banner with the text "学IT选千锋" and "严选师资面授品质", along with a small trophy icon. The developer tools are open, specifically the Elements tab, which shows the HTML structure and the CSS styles applied to the page. A red box highlights the CSS rule for the `.block-bg` class, which defines a width of 200px, a height of 200px, and a background image using the `url(data:image/svg+xml,%3csvg%3e)` syntax.

我们看到，通过样式把背景图片加到了页面中。

1.6.5 加载 fonts 字体

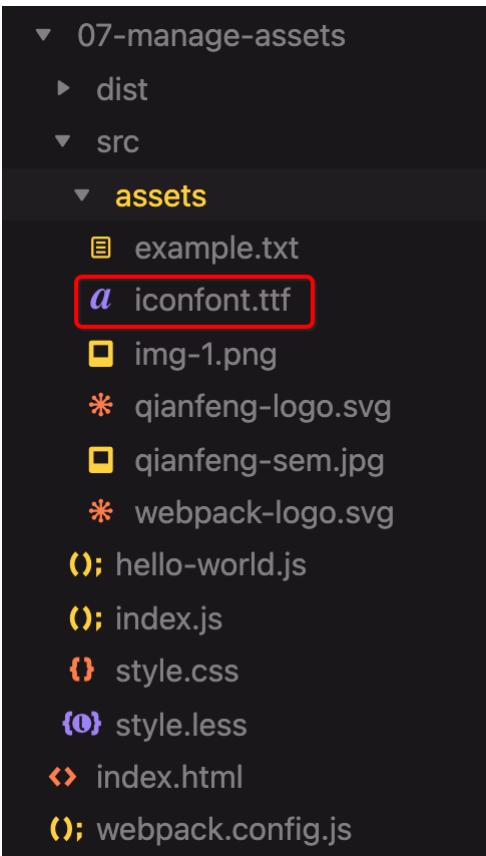
那么，像字体这样的其他资源如何处理呢？使用 Asset Modules 可以接收并加载任何文件，然后将其输出到构建目录。这就是说，我们可以将它们用于任何类型的文件，也包括字体。让我们更新 `webpack.config.js` 来处理字体文件：

```
module: {  
  rules: [  
    {  
      test: /\.woff|woff2|eot|ttf|otf$/i,  
      type: 'asset/resource',  
    },  
  ],  
}
```

07-manage-assets/webpack.config.js

```
//...  
  
module.exports = {  
//...  
  
// 配置资源文件  
  module: {  
    rules: [  
      //...  
      {  
        test: /\.woff|woff2|eot|ttf|otf$/i,  
        type: 'asset/resource',  
      },  
    ],  
  },  
//...  
}
```

在项目中添加一些字体文件：



配置好 loader 并将字体文件放在合适的位置后，你可以通过一个 `@font-face` 声明将其混合。本地的 `url(...)` 指令会被 webpack 获取处理，就像它处理图片一样：

```
@font-face {  
    font-family: 'iconfont';  
    src: url('./assets/iconfont.ttf') format('truetype');  
}  
  
.icon {  
    font-family: "iconfont" !important;  
    font-size: 30px;  
    font-style: normal;  
    -webkit-font-smoothing: antialiased;  
    -moz-osx-font-smoothing: grayscale;  
}
```

07-manage-assets/src/style.css

```
@font-face {  
    font-family: 'iconfont';  
    src: url('./assets/iconfont.ttf') format('truetype');  
}  
  
.hello {  
    background-color: #f9efd4;
```

```
}

.icon {
font-family: "iconfont" !important;
font-size: 30px;
font-style: normal;
-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: grayscale;
}

.block-bg {
background-image: url("./assets/webpack-logo.svg");
}
```

```
const span = document.createElement('span')
span.classList.add('icon')
span.innerHTML = ''
document.body.appendChild(span)
```

07-manage-assets/src/index.js

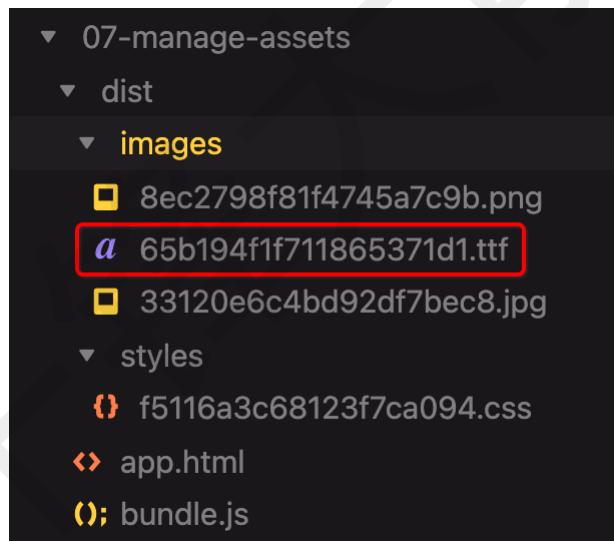
```
// 导入模块
//...

const span = document.createElement('span')
span.classList.add('icon')
span.innerHTML = ''
document.body.appendChild(span)
```

启动服务，打开浏览器：

我们再打包一下，看看输出的文件：

```
[felix] 07-manage-assets $ npx webpack
```



再看一下打包好的 `styles/.css` 文件：

07-manage-assets/dist/styles/4a9cff551c7a105e1554.css

```
/*! ****!*\
*** css ..../node_modules/css-
loader/dist/cjs.js!./src/style.css ***!
```

```
\******/@font-face{font-
family:iconfont;src:url(..../images/65b194f1f711865371d1.ttf)
format("truetype")}.hello{background-color:#f9efd4}.icon{-
webkit-font-smoothing:antialiased;-moz-osx-font-
smoothing:grayscale;font-family:iconfont!important;font-
size:16px;font-style:normal}.block-bg{background-
image:url("data:image/svg+xml; charset=utf-8,%3Csvg viewBox='0
0 3046.7 875.7' xmlns='http://www.w3.org/2000/svg'%3E%3Cpath
d='m387 0 387 218.9v437.9L387 875.7 0 656.8v218.9z'
fill='%23fff'/%3E%3Cpath d='M704.9 641.7 399.8
814.3v679.9l190.1-104.6zm20.9-18.9v261.9l-111.6
64.5v232zM67.9 641.7 373 814.3v679.9L182.8 575.3zM47
622.8v261.9l111.6 64.5v232zm13.1-384.3L373 61.5v129.9L172.5
301.7l-1.6.9zm652.6 0-312.9-177v129.9l200.5 110.2 1.6.9z'
fill='%238ed6fb'/%3E%3Cpath d='M373 649.3 185.4
546.1v341.8L373 450.1zm26.8 0 187.6-103.1v341.8L399.8
450.1zm198.1 318.2l188.3-103.5 188.3 103.5-188.3 108.7z'
fill='%231c78c0'/%3E%3Cpath d='M1164.3 576.3h82.5l84.1-
280.2h-80.4l-49.8 198.8-53.1-198.8h1078l-53.6 198.8-49.3-
198.8h-80.4l83.6 280.2h82.5l52-179.5zM1335.2 437c0 84.1 57.3
146.3 147.4 146.3 69.7 0 107.2-41.8 117.9-61.6l-48.8-37c-8
11.8-30 34.3-68.1 34.3-41.3 0-71.3-26.8-72.9-64.3H1608c.5-
5.4.5-10.7.5-16.1 0-91.6-49.3-149.5-136.1-149.5-79.9 0-137.2
63.2-137.2 147.9zm77.7-30.6c3.2-32.1 25.7-56.8 60.6-56.8 33.8
0 58.4 22.5 60 56.8zm223.5 169.9h69.7v-28.9c7.5 9.1 35.4 35.9
83.1 35.9 80.4 0 137.2-60.5 137.2-146.8 0-86.8-52.5-147.3-
132.9-147.3-48.2 0-76.1 26.8-83.1 36.4v188.9h-
73.9v387.4zm71.8-139.3c0-52.5 31.1-82.5 71.8-82.5 42.9 0 71.8
33.8 71.8 82.5 0 49.8-30 80.9-71.8 80.9-45 0-71.8-36.5-71.8-
80.9zm247 239.5h73.9v547.3c7 9.1 34.8 35.9 83.1 35.9 80.4 0
132.9-60.5 132.9-147.3 0-85.7-56.8-146.8-137.2-146.8-47.7 0-
75.6 26.8-83.1 36.4v296h-69.7v380.5zm71.8-241.1c0-44.5 26.8-
80.9 71.8-80.9 41.8 0 71.8 31.1 71.8 80.9 0 48.8-28.9 82.5-
71.8 82.5-40.7 0-71.8-30-71.8-82.5zm231.5 54.1c0 58.9 48.2
93.8 105 93.8 32.2 0 53.6-9.6 68.1-25.2l4.8
18.2h65.4v398.9c0-62.7-26.8-109.8-116.8-109.8-42.9 0-85.2
16.1-110.4 33.2l27.9 50.4a165.2 165.2 0 0 1 74.5-19.8c32.7 0
50.9 16.6 50.9 41.3v18.2c-10.2-7-32.2-15.5-60.6-15.5-65.4-.1-
108.8 37.4-108.8 92.6zm73.9-2.2c0-23 19.8-39.1 48.2-39.1s48.8
14.5 48.8 39.1c0 23.6-20.4 38.6-48.2 38.6s-48.8-15.5-48.8-
38.6zm348.9 30.6c-46.6 0-79.8-33.8-79.8-81.4 0-45 29.5-82
77.2-82a95.2 95.2 0 0 1 65.4 26.8l20.9-62.2a142.6 142.6 0 0
0-88.4-30c-85.2 0-149 62.7-149 147.9s62.2 146.3 149.5
146.3a141 141 0 0 0 87.3-30l-19.8-60.5c-12.4 10.1-34.9 25.1-
```

63.3 25.1zm110.9 58.4h73.9v431.6l93.8 144.7h86.8L2940.6
423198.6-127h-83.1l-90 117.9v-225h-73.9z'
fill='%23f5fafa'/%3E%3C/svg%3E")}




```
uMi0xNDYuOCAwLTg2LjgtNTIuNS0xNDcuMy0xMzIuOS0xNDcuMy000C4yIDAt
NzYuMSAyNi44LTgzLjEgMzYuNHYtMTM2LjdoLTCzLj12Mzg3LjR6bTcxLjgtM
TM5LjnjMC01Mi41IDMXLjEtODIuNSA3MS44LTgyLjUgNDIuOSAwIDCXLjggMz
MuOCA3MS44IDgyLjUgMCA0OS44LTmwidgwLjktNzEuOCA4MC45LTQ1IDATNZE
uOC0zNi41LTcxLjgtODAuoxptMjQ3IDIZOS41aDczLj12LTEyOS4yYzcgOS4x
IDM0LjggMzUuOSA4My4xIDM1LjkgODAUncAwIDEZMi45LTywLjUgMTMyLjktM
TQ3LjMgMC04NS43LTU2LjgtMTQ2LjgtMTM3LjItMTQ2LjgtNDcuNyAwLTc1Lj
YgMjYuOC04My4xIDM2LjR2LTi5LjVOLTy5Ljd2MzgwLjV6bTcxLjgtMjQxLjF
jMC00NC41IDI2LjgtODAUOSA3MS44LTgwLjkgNDEuOCAwIDCXLjggMzEuMSA3
MS44IDgwLjkgMCA0OC44LTi4LjkgODIuNS03MS44IDgyLjUtNDAuNyAwLTcXL
jgtMzAtNzEuOC04Mi41em0yMzEuNSA1NC4xYZAgNTguOSA0OC4yIDkzLjggMT
A1IDkzLjggMzIuMiAwIDUzLjYtOs42IDY4LjEtMjUuMmw0LjggMTguMmg2Ns4
0di0XNzcungMwlTYyLjctMjYuOC0xMDkuOC0xMTYuOC0xMDkuOC00Mi45IDAT
ODUuMiAxNi4xLTExMC40IDMzLjJsmjcuOSA1MC40YTE2NS4yIDE2NS4yIDAgM
CAXIDC0LjutMTkuOGMzMi43IDAgNTAuOSAxNi42IDUwLjkgNDEuM3YxOC4yYy
0xMC4yLTctMzIuMi0xNS41LTywLjYtMTuuNS02NS40LS4xLTewOC44IDM3LjQ
tMTA4LjggOTIuNnptNzMuOS0yLjJjMC0yMyAxOS44LTm5LjEgNDguMi0zOS4x
czQ4LjggMTQuNSA0OC44IDM5LjFjMCAyMy42LTiwLjQgMzguNi000C4yIDM4L
jzzLTQ4LjgtMTuuNS000C44LTm4LjZ6btM00C45IDMwLjZjLTQ2LjYgMC03oS
44LTmzLjgtNzkuOC04MS40IDATNDugMjkuns04MiA3Ny4yLTgyYTk1LjIgotU
uMiAwIDAgMSA2NS40IDI2LjhsMjAuOS02Mi4yYTE0Mi42IDE0Mi42IDAgMCAw
IC040C40LTmwy04NS4yIDATMTQ5IDYyLjctMTQ5IDE0Ny45czYyLjIgMTQ2Lj
jMqMTQ5LjUgMTQ2LjNhMTQxIDE0MSAwIDAgMCA4Ny4zLTmwbC0x0s44LTywLj
VjLTEyLjQgMTAuMS0zNC45IDI1LjEtNjMuMyAyNS4xem0xMTAuOSA1OC40aDc
zLj12LTE0NC43bdKzLjggMTQ0LjdoODYuOGwtMTA2LjEtMTUzLjMgOTguNi0x
Mjd0LTgzLjFSLTkWIDE Ny45di0yMjV0LTCzLj16jyBmaWxsPSclMjNmNWzhz
mEnLyUZZSUzYy9zdmc1M2vciikgo1xufvxuIiwiQGZvbnQtZmFjZSB7XG4gIG
ZvbnQtZmFtaWx50iAnawNvbmZvbnQn01xuICBzcmM6IHVybCgnLi9hc3NldHM
vaWNvbmZvbnQudHRmJykgZm9ybWF0Kcd0cnvlhdhwzScp01xufvxuXG4uaGVs
bG8ge1xuICBiYWRz3Jvdw5kLwNvbG9yoiajzj11zmQ001xufvxuXG4uaWNvb
ib7XG4gIGZvbnQtZmFtaWx50iBcIm1jb25mb250XCIGIwltcG9ydGFudDtcbi
AgZm9udc1zaXploiaxNnb401xuICBmb250Lxn0ewx1oibub3jtYww7XG4gIC1
3zwJraxQtZm9udc1zbw9vdghpbmc6IGFudG1hbG1hc2vk01xuICAtbw96Lw9z
ec1mb250Lxntb290agluzzogZ3jhexnjYwx1o1xufvxuXG4uYmxvY2stYmcge
1xuICBiYWRz3Jvdw5kLw1tYwdloib1cmwoLi9hc3NldHMvd2VicGFjay1sb2
dvLnN2Zykg01xufSjdLCJuYw11cyI6w10sInNvdXjjZVjvb3QioiiifQ==*/
```

由于在前面我们应用了如下配置，使生产环境css文件也进行了压缩处理。我们可以注释它：

```
optimization: {
  // minimize: true,
}
```

1.6.6 加载数据

此外，可以加载的有用资源还有数据，如 JSON 文件，CSV、TSV 和 XML。类似于 NodeJS，JSON 支持实际上是内置的，也就是说 `import data from './data.json'` 默认将正常运行。要导入 CSV、TSV 和 XML，你可以使用 [CSV-loader](#) 和 [xml-loader](#)。让我们处理加载这三类文件：

```
[felix] webpack5 $ npm install --save-dev csv-loader xml-loader
```

添加配置：

```
module: {
  rules: [
    {
      test: /\.csv|tsv$/i,
      use: ['csv-loader'],
    },
    {
      test: /\.xml$/i,
      use: ['xml-loader'],
    },
  ]
}
```

07-manage-assets/webpack.config.js

```
//...
module.exports = {
//...
// 配置资源文件
module: {
  rules: [
    //...
    {
      test: /\.csv|tsv$/i,
      use: ['csv-loader'],
    },
    {
      test: /\.xml$/i,
      use: ['xml-loader'],
    },
  ],
}
```

```
  ] ,  
  } ,  
  
  // ...  
}
```

现在，你可以 `import` 这四种类型的数据(JSON, CSV, TSV, XML)中的任何一种，所导入的 `Data` 变量，将包含可直接使用的已解析 JSON：

创建两个文件：

07-manage-assets/src/assets/data.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<note>  
  <to>Mary</to>  
  <from>John</from>  
  <heading>Reminder</heading>  
  <body>Call Cindy on Tuesday</body>  
</note>
```

07-manage-assets/src/assets/data.csv

```
to,from,heading,body  
Mary,John,Reminder,Call Cindy on Tuesday  
Zoe,Bill,Reminder,Buy orange juice  
Autumn,Lindsey,Letter,I miss you
```

在入口文件里加载数据模块，并在控制台上打印导入内容：

```
import Data from './assets/data.xml'  
import Notes from './assets/data.csv'  
  
console.log(Data)  
console.log(Notes)
```

07-manage-assets/src/index.js

```
// 导入模块
//...

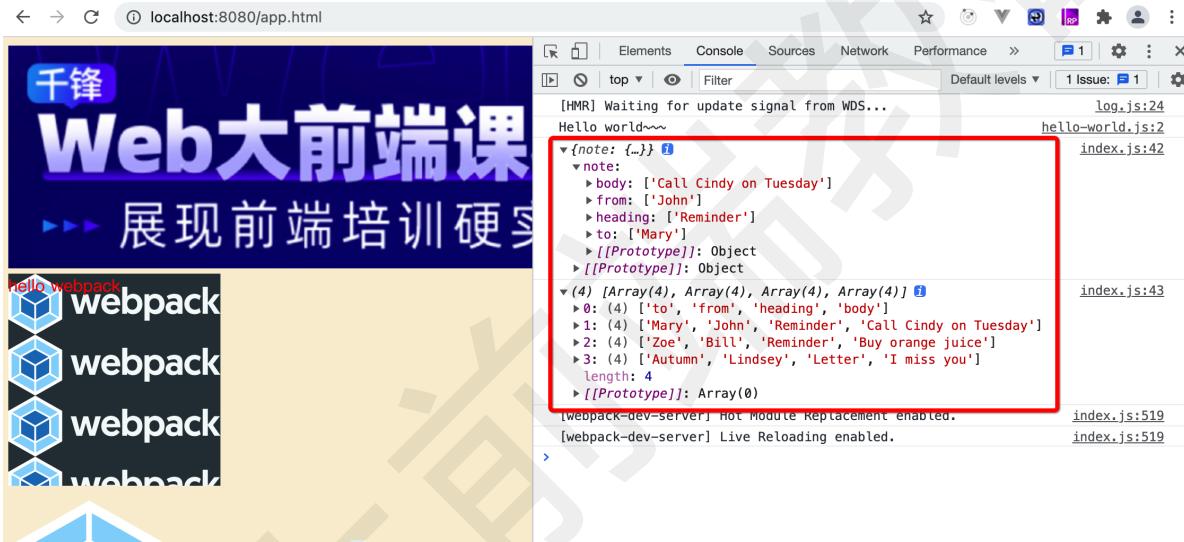
import Data from './assets/data.xml'
import Notes from './assets/data.csv'

//...

console.log(Data)
console.log(Notes)
```

查看开发者工具中的控制台，你应该能够看到导入的数据会被打印出来！

```
[felix] 07-manage-assets $ npx webpack serve
```



由此可见，`data.xml`文件转化为一个JS对象，`data.csv`转化为一个数组。

1.6.7 自定义 JSON 模块 parser

通过使用 [自定义 parser](#) 替代特定的 webpack loader，可以将任何 `toml`、`yaml` 或 `json5` 文件作为 JSON 模块导入。

假设你在 `src` 文件夹下有一个 `data.toml`、一个 `data.yaml` 以及一个 `data.json5` 文件：

```
07-manage-assets/src/assets/json/data.toml
```

```
title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
organization = "GitHub"
bio = "GitHub Cofounder & CEO\nLikes tater tots and beer."
dob = 1979-05-27T07:32:00Z
```

07-manage-assets/src/assets/json/data.yaml

```
title: YAML Example
owner:
name: Tom Preston-Werner
organization: GitHub
bio: |-  
GitHub Cofounder & CEO  
Likes tater tots and beer.
dob: 1979-05-27T07:32:00.000Z
```

07-manage-assets/src/assets/json/data.json5

```
{
// comment
title: 'JSON5 Example',
owner: {
  name: 'Tom Preston-Werner',
  organization: 'GitHub',
  bio: 'GitHub Cofounder & CEO\n\
Likes tater tots and beer.',
  dob: '1979-05-27T07:32:00.000Z',
},
}
```

首先安装 `toml`, `yamljs` 和 `json5` 的 packages:

```
[felix] webpack5 $ npm install toml yamljs json5 --save-dev
```

并在你的 webpack 中配置它们:

```
const toml = require('toml');
const yaml = require('yamljs');
const json5 = require('json5');

module.exports = {
```

```
module: {
  rules: [
    {
      test: /\.toml$/i,
      type: 'json',
      parser: {
        parse: toml.parse,
      },
    },
    {
      test: /\.yaml$/i,
      type: 'json',
      parser: {
        parse: yaml.parse,
      },
    },
    {
      test: /\.json5$/i,
      type: 'json',
      parser: {
        parse: json5.parse,
      },
    },
  ],
}
```

07-manage-assets/webpack.config.js

```
//...

const toml = require('toml')
const yaml = require('yamljs')
const json5 = require('json5')

module.exports = {
//...

// 配置资源文件
module: {
  rules: [
    //...

    {
      test: /\.toml$/i,
```

```
        type: 'json',
        parser: {
          parse: toml.parse,
        },
      },
      {
        test: /\.yaml$/i,
        type: 'json',
        parser: {
          parse: yaml.parse,
        },
      },
      {
        test: /\.json5$/i,
        type: 'json',
        parser: {
          parse: json5.parse,
        },
      },
    ],
  },
}

//...
}
```

在主文件中引入模块，并打印内容：

```
import toml from './data.toml';
import yaml from './data.yaml';
import json from './data.json5';

console.log(toml.title); // output `TOML Example`
console.log(toml.owner.name); // output `Tom Preston-Werner`

console.log(yaml.title); // output `YAML Example`
console.log(yaml.owner.name); // output `Tom Preston-Werner`

console.log(json.title); // output `JSON5 Example`
console.log(json.owner.name); // output `Tom Preston-Werner`
```

07-manage-assets/src/index.js

```
// 导入模块
//...
import toml from './assets/json/data.toml'
```

```
import yaml from './assets/json/data.yaml'
import json from './assets/json/data.json5'

//...

console.log(toml.title); // output `TOML Example`
console.log(toml.owner.name); // output `Tom Preston-Werner`

console.log(yaml.title); // output `YAML Example`
console.log(yaml.owner.name); // output `Tom Preston-Werner`

console.log(json.title); // output `JSON5 Example`
console.log(json.owner.name); // output `Tom Preston-Werner`
```

启动服务，打开浏览器：

```
[felix] 07-manage-assets $ npx webpack serve
```

现在，`toml`、`yaml`和`json5`几个类型的文件都正常输出了结果。

1.7 使用 babel-loader

前面的章节里，我们应用`less-loader`编译过`less`文件，应用`xml-loader`编译过`xml`文件，那`js`文件需要编译吗？我们来做一个实验，修改`hello-world.js`文件：

08-babel-loader/src/hello-world.js

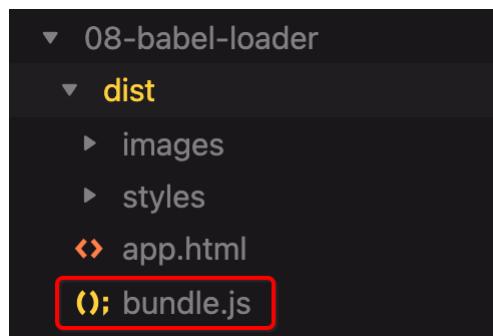
```
function getString() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Hello world~~~')
    }, 2000)
  })
}

async function helloworld() {
  let string = await getString()
  console.log(string)
}

// 导出函数模块
export default helloworld
```

执行编译:

```
[felix] 08-babel-loader $ npx webpack
```



查看 bundle.js 文件:

08-babel-loader/dist/bundle.js

```
//...

/**/
"./src/hello-world.js":
/*! ****!*\
  !*** ./src/hello-world.js ***!
  \****/
((__unused_webpack_module, __webpack_exports__, __webpack_require__) => {

  "use strict";
  __webpack_require__.r(__webpack_exports__);
  /* harmony export */
  __webpack_require__.d(__webpack_exports__, {
    /* harmony export */
    "default": () => (__WEBPACK_DEFAULT_EXPORT__)
    /* harmony export */
  });
}

function getString() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Hello world~~~')
    }, 2000)
  })
}

async function helloworld() {
```

```

let string = await getString()
console.log(string)
}

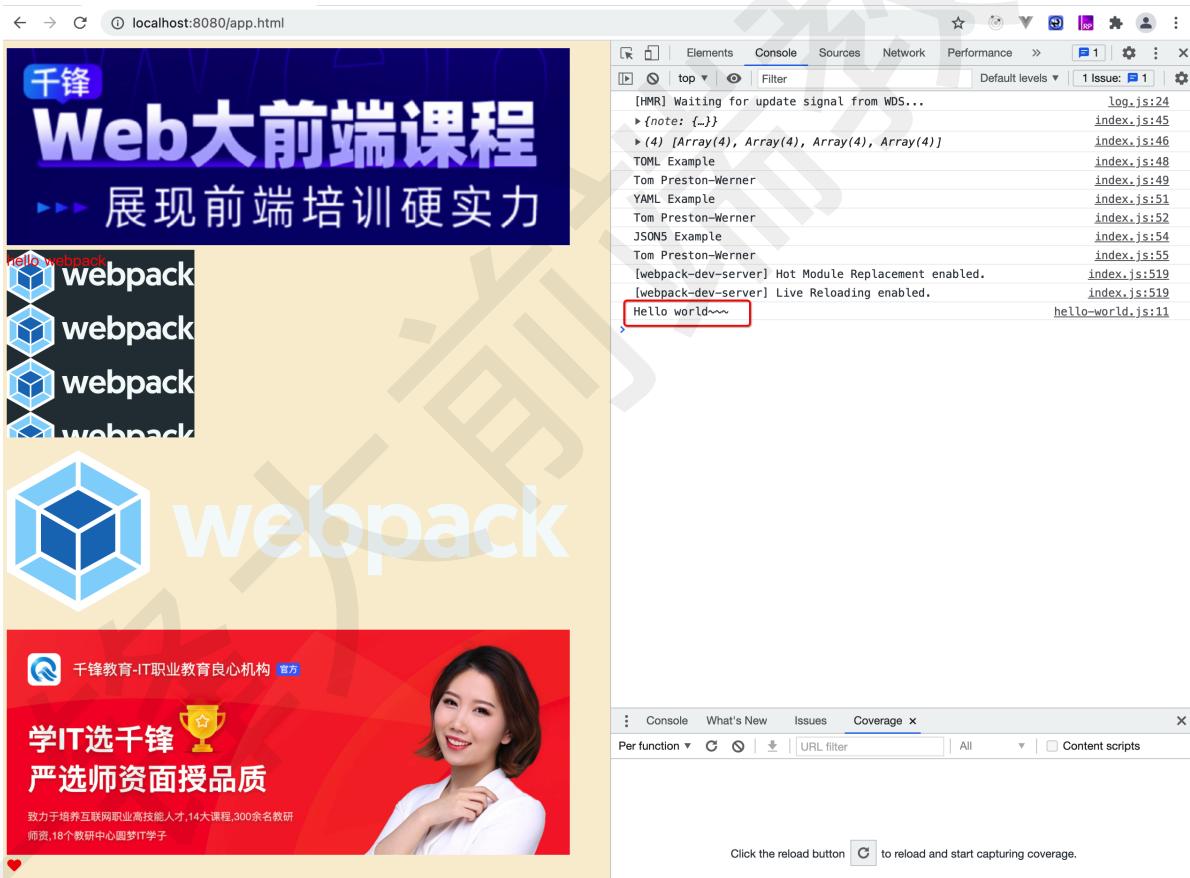
// 导出函数模块
/* harmony default export */
const __WEBPACK_DEFAULT_EXPORT__ = (helloworld);

/**/
},
//...

```

我们发现，编写的ES6代码原样输出了。启动服务，打开浏览器：

```
[felix] 08-babel-loader $ npx webpack serve
```



`Hello world~~` 两秒后正常输出，说明浏览器能够运行我们的ES6代码。但如果浏览器版本过低，就很难保证代码正常运行了。

1.7.1 为什么需要 babel-loader

webpack 自身可以自动加载JS文件，就像加载JSON文件一样，无需任何 loader。可是，加载的JS文件会原样输出，即使你的JS文件里包含ES6+的代码，也不会做任何的转化。这时我们就需要Babel来帮忙。Babel 是一个 JavaScript 编译器，可以将 ES6+转化成ES5。在Webpack里使用Babel，需要使用 `babel-loader`。

1.7.2 使用 babel-loader

安装：

```
npm install -D babel-loader @babel/core @babel/preset-env
```

- `babel-loader`: 在webpack里应用 babel 解析ES6的桥梁
- `@babel/core`: babel核心模块
- `@babel/preset-env`: babel预设，一组 babel 插件的集合

在 webpack 配置中，需要将 `babel-loader` 添加到 `module` 列表中，就像下面这样：

```
module: {
  rules: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env']
        }
      }
    }
  ]
}
```

执行编译：

```
[felix] 08-babel-loader $ npx webpack
```

查看 `bundle.js` 文件：

08-babel-loader/dist/bundle.js

```
/**/
"./src/hello-world.js":
/*!*****!*\
 !*** ./src/hello-world.js ***!

```

```
\*****
((__unused_webpack_module, __webpack_exports__,
__webpack_require__) => {

"use strict";
__webpack_require__.r(__webpack_exports__);
/* harmony export */
__webpack_require__.d(__webpack_exports__, {
/* harmony export */
"default": () => (__WEBPACK_DEFAULT_EXPORT__)
/* harmony export */
});

function asyncGeneratorStep(gen, resolve, reject, _next,
_throw, key, arg) {
try {
var info = gen[key](arg);
var value = info.value;
} catch (error) {
reject(error);
return;
}
if (info.done) {
resolve(value);
} else {
Promise.resolve(value).then(_next, _throw);
}
}

function _asyncToGenerator(fn) {
return function () {
var self = this,
args = arguments;
return new Promise(function (resolve, reject) {
var gen = fn.apply(self, args);

function _next(value) {
asyncGeneratorStep(gen, resolve, reject, _next, _throw,
"next", value);
}

function _throw(err) {
asyncGeneratorStep(gen, resolve, reject, _next, _throw,
"throw", err);
}
});
}
}
```

```
        }
        _next(undefined);
    });
};

}

function getString() {
return new Promise(function (resolve, reject) {
setTimeout(function () {
    resolve('Hello world~~~');
}, 2000);
});
}

function helloworld() {
return _helloworld.apply(this, arguments);
} // 导出函数模块

function _helloworld() {
_helloworld = _asyncToGenerator(/*#__PURE__*/
regeneratorRuntime.mark(function _callee() {
    var string;
    return regeneratorRuntime.wrap(function _callee$(_context) {
        while (1) {
            switch (_context.prev = _context.next) {
                case 0:
                    _context.next = 2;
                    return getString();

                case 2:
                    string = _context.sent;
                    console.log(string);

                case 4:
                case "end":
                    return _context.stop();
            }
        }
    }, _callee);
}))();
return _helloworld.apply(this, arguments);
}

/* harmony default export */
```

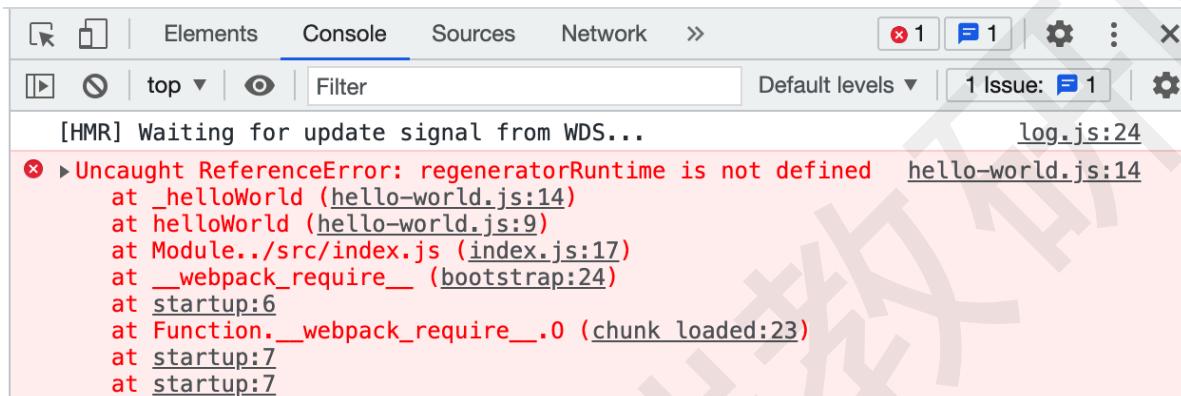
```
const __WEBPACK_DEFAULT_EXPORT__ = (helloworld);

/**/
},
```

从编译完的结果可以看出，`async/await` 的ES6语法被 `babel` 编译了。

1.7.3 regeneratorRuntime 插件

此时执行编译，在浏览器里打开项目发现报了一个致命错误：



`regeneratorRuntime` 是 webpack 打包生成的全局辅助函数，由 babel 生成，用于兼容 `async/await` 的语法。

`regeneratorRuntime is not defined` 这个错误显然是未能正确配置 babel。

正确的做法需要添加以下的插件和配置：

```
# 这个包中包含了regeneratorRuntime，运行时需要
npm install --save @babel/runtime

# 这个插件会在需要regeneratorRuntime的地方自动require导包，编译时需要
npm install --save-dev @babel/plugin-transform-runtime

# 更多参考这里
https://babeljs.io/docs/en/babel-plugin-transform-runtime
```

接着改一下 babel 的配置：

```
module: {
  rules: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env'],
          plugins: ['@babel/plugin-transform-runtime']
        }
      }
    }
  ]
}
```

```
        options: {
          presets: ['@babel/preset-env'],
          plugins: [
            [
              '@babel/plugin-transform-runtime'
            ]
          ]
        }
      }
    ]
}
```

08-babel-load/webpack.config.js

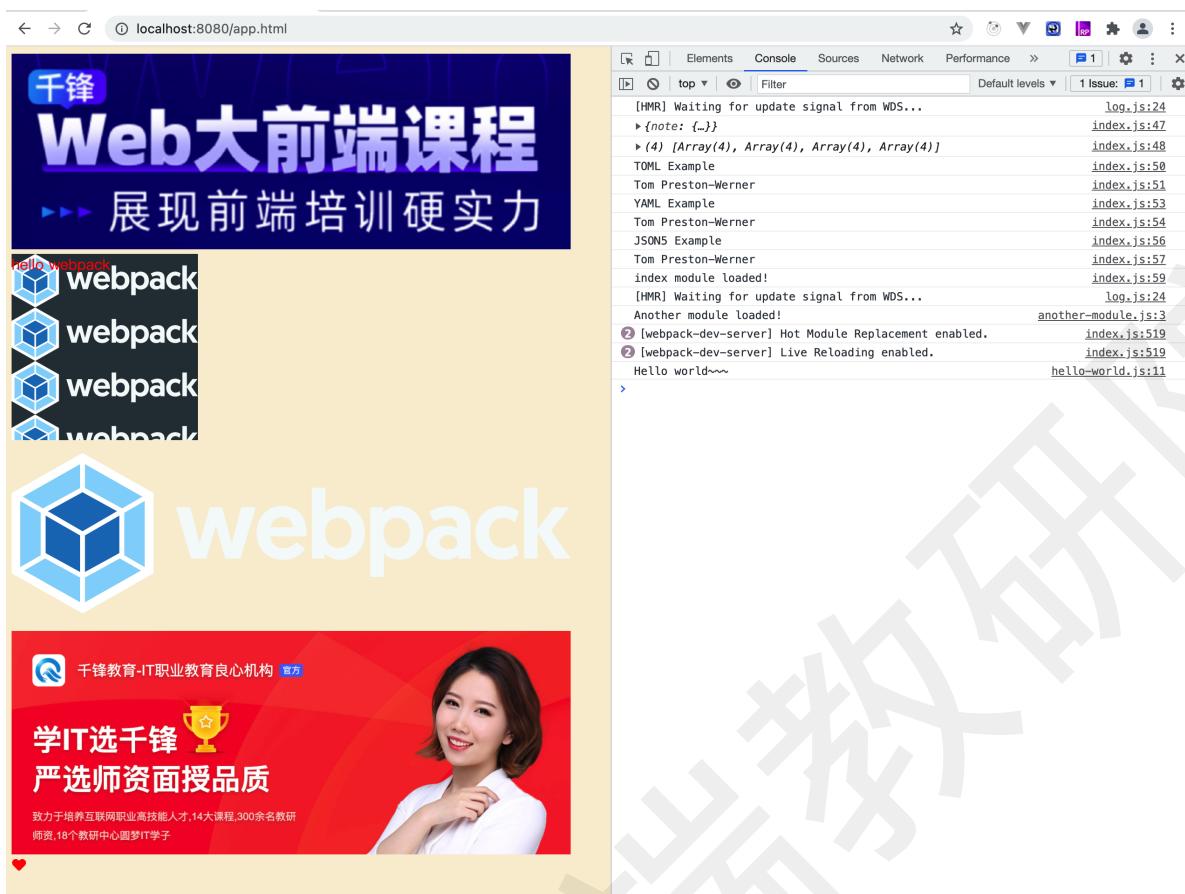
```
//...

module.exports = {
//...

// 配置资源文件
module: {
  rules: [{
    test: /\.js$/,
    exclude: /node_modules/,
    use: {
      loader: 'babel-loader',
      options: {
        presets: ['@babel/preset-env'],
        plugins: [
          [
            '@babel/plugin-transform-runtime'
          ]
        ]
      }
    },
    ],
  },
  //...
},

//...
],
```

启动服务，打开浏览器：



成功运行。

1.8 代码分离

代码分离是 webpack 中最引人注目的特性之一。此特性能够把代码分离到不同的 bundle 中，然后可以按需加载或并行加载这些文件。代码分离可以用于获取更小的 bundle，以及控制资源加载优先级，如果使用合理，会极大影响加载时间。

常用的代码分离方法有三种：

- **入口起点**：使用 `entry` 配置手动地分离代码。
- **防止重复**：使用 `Entry dependencies` 或者 `splitChunksPlugin` 去重和分离 chunk。
- **动态导入**：通过模块的内联函数调用来分离代码。

1.8.1 入口起点

这是迄今为止最简单直观的分离代码的方式。不过，这种方式手动配置较多，并有一些隐患，我们将会解决这些问题。先来看看如何从 main bundle 中分离 another module(另一个模块)：

在 `src` 目录下创建 `another-module.js` 文件：

09-code-splitting/src/another-module.js

```
import _ from 'lodash'

console.log(_.join(['Another', 'module', 'loaded!'], ' '))
```

这个模块依赖了 `lodash`, 需要安装一下:

```
[felix] webpack5 $ npm install lodash --save-dev
```

修改配置文件:

```
module.exports = {
  entry: {
    index: './src/index.js',
    another: './src/another-module.js',
  },
  output: {
    filename: '[name].bundle.js'
  },
}
```

09-code-splitting/webpack.config.js

```
//...

module.exports = {
  entry: {
    index: './src/index.js',
    another: './src/another-module.js',
  },

  output: {
    filename: '[name].bundle.js'
    //...
  },
}

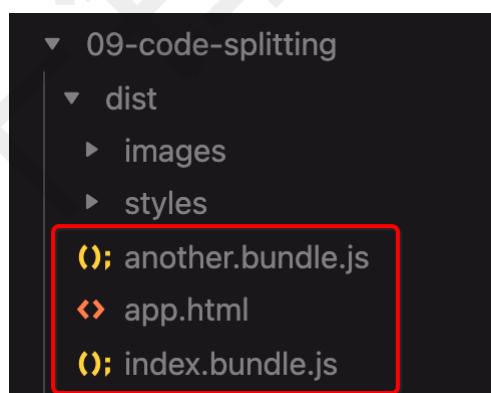
//...
```

执行编译:

```
[felix] 09-code-splitting $ npx webpack
assets by status 744 KiB [cached] 4 assets
assets by status 1.44 MiB [emitted]
  asset another.bundle.js 1.38 MiB [emitted] (name: another)
```

```
asset index.bundle.js 65.1 KiB [emitted] (name: index)
asset app.html 441 bytes [emitted]
Entrypoint index 68.9 KiB (740 KiB) =
  styles/4a9cff551c7a105e1554.css 3.81 KiB index.bundle.js 65.1 KiB
  3 auxiliary assets
Entrypoint another 1.38 MiB = another.bundle.js
runtime modules 3.23 KiB 12 modules
cacheable modules 549 KiB (javascript) 738 KiB (asset) 2.65 KiB
(css/mini-extract)
  javascript modules 546 KiB
    modules by path ../node_modules/ 540 KiB 9 modules
    modules by path ./src/ 5.48 KiB 8 modules
  asset modules 3.1 KiB (javascript) 738 KiB (asset)
    ./src/assets/img-1.png 42 bytes (javascript) 101 KiB (asset)
[built] [code generated]
  ./src/assets/webpack-logo.svg 2.99 KiB [built] [code
generated]
  ./src/assets/example.txt 25 bytes [built] [code generated]
  ./src/assets/qianfeng-sem.jpg 42 bytes (javascript) 637 KiB
(asset) [built] [code generated]
  json modules 565 bytes
    ./src/assets/json/data.toml 188 bytes [built] [code
generated]
    ./src/assets/json/data.yaml 188 bytes [built] [code
generated]
    ./src/assets/json/data.json5 189 bytes [built] [code
generated]
  css ../node_modules/css-loader/dist/cjs.js!./src/style.css 2.65
KiB [built] [code generated]
webpack 5.54.0 compiled successfully in 854 ms
```

asset another.bundle.js 1.38 MiB [emitted] (name: another), 我们发现 lodash.js
也被打包到 another.bundle.js 中。



查看 app.html:

09-code-splitting/dist/app

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

<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>千锋大前端教研院-webpack5学习指南</title>
<link href="styles/4a9cff551c7a105e1554.css" rel="stylesheet">
</head>

<body>
<script defer src="index.bundle.js"></script>
<script defer src="another.bundle.js"></script>
</body>

</html>
```

两个入口的 `bundle` 文件都被链接到了 `app.html` 中。

我们再来修改一下 `index.js` 文件：

```
import _ from 'lodash'

console.log(_.join(['index', 'module', 'loaded!'], ' '))
```

09-code-splitting/src/index.js

```
// 导入模块
//...
import _ from 'lodash'

//...

console.log(_.join(['index', 'module', 'loaded!'], ' '))
```

执行编译：

```
[felix] 09-code-splitting $ npx webpack
assets by status 744 KiB [cached] 4 assets
assets by path . 2.82 MiB
```

```
asset index.bundle.js 1.44 MiB [emitted] (name: index)
asset another.bundle.js 1.38 MiB [compared for emit] (name: another)
asset app.html 441 bytes [compared for emit]
Entrypoint index 1.44 MiB (740 KiB) =
  styles/4a9cff551c7a105e1554.css 3.81 KiB index.bundle.js 1.44 MiB
  3 auxiliary assets
Entrypoint another 1.38 MiB = another.bundle.js
runtime modules 3.35 KiB 13 modules
cacheable modules 549 KiB (javascript) 738 KiB (asset) 2.65 KiB
(css/mini-extract)
  javascript modules 546 KiB
    modules by path ../node_modules/ 540 KiB 9 modules
    modules by path ./src/ 5.57 KiB 8 modules
  asset modules 3.1 KiB (javascript) 738 KiB (asset)
    ./src/assets/img-1.png 42 bytes (javascript) 101 KiB (asset)
[built] [code generated]
  ./src/assets/webpack-logo.svg 2.99 KiB [built] [code
generated]
  ./src/assets/example.txt 25 bytes [built] [code generated]
  ./src/assets/qianfeng-sem.jpg 42 bytes (javascript) 637 KiB
(asset) [built] [code generated]
  json modules 565 bytes
    ./src/assets/json/data.toml 188 bytes [built] [code
generated]
    ./src/assets/json/data.yaml 188 bytes [built] [code
generated]
    ./src/assets/json/data.json5 189 bytes [built] [code
generated]
  css ../node_modules/css-loader/dist/cjs.js!./src/style.css 2.65
KiB [built] [code generated]
webpack 5.54.0 compiled successfully in 898 ms
```

观察一下：

```
assets by path . 2.82 MiB
  asset index.bundle.js 1.44 MiB [emitted] (name: index)
  asset another.bundle.js 1.38 MiB [compared for emit] (name: another)
  asset app.html 441 bytes [compared for emit]
```

我们发现：`index.bundle.js` 文件大小也骤然增大了，可以`lodash.js`也被打包到了`index.bundle.js`中了。

正如前面提到的，这种方式的确存在一些隐患：

- 如果入口 chunk 之间包含一些重复的模块，那些重复模块都会被引入到各个 bundle 中。
- 这种方法不够灵活，并且不能动态地将核心应用程序逻辑中的代码拆分出来。

以上两点中，第一点对我们的示例来说无疑是个问题，因为之前我们在 `./src/index.js` 中也引入过 `lodash`，这样就在两个 bundle 中造成重复引用。

1.8.2 防止重复

- 入口依赖

配置 [dependOn option](#) 选项，这样可以在多个 chunk 之间共享模块：

```
module.exports = {
  entry: {
    index: {
      import: './src/index.js',
      dependOn: 'shared',
    },
    another: {
      import: './src/another-module.js',
      dependOn: 'shared',
    },
    shared: 'lodash',
  }
}
```

09-code-splitting/webpack.config.js

```
//...

module.exports = {
  entry: {
    index: {
      import: './src/index.js',
      dependOn: 'shared',
    },
    another: {
      import: './src/another-module.js',
      dependOn: 'shared',
    },
    shared: 'lodash',
  },
}

//...
```

```
}
```

执行编译：

```
[felix] 09-code-splitting $ npx webpack
assets by status 744 KiB [cached] 4 assets
assets by status 1.45 MiB [emitted]
asset shared.bundle.js 1.39 MiB [emitted] (name: shared)
asset index.bundle.js 57.1 KiB [emitted] (name: index)
asset another.bundle.js 1.53 KiB [emitted] (name: another)
asset app.html 487 bytes [emitted]
Entrypoint index 60.9 KiB (740 KiB) =
  styles/4a9cff551c7a105e1554.css 3.81 KiB index.bundle.js
  57.1 KiB 3 auxiliary assets
Entrypoint another 1.53 KiB = another.bundle.js
Entrypoint shared 1.39 MiB = shared.bundle.js
runtime modules 4.47 KiB 9 modules
cacheable modules 549 KiB (javascript) 738 KiB (asset)
  2.65 KiB (css/mini-extract)
javascript modules 546 KiB
modules by path ../node_modules/ 540 KiB 9 modules
modules by path ./src/ 5.57 KiB 8 modules
asset modules 3.1 KiB (javascript) 738 KiB (asset)
  ./src/assets/img-1.png 42 bytes (javascript) 101 KiB
  (asset) [built] [code generated]
  ./src/assets/webpack-logo.svg 2.99 KiB [built] [code
  generated]
  ./src/assets/example.txt 25 bytes [built] [code generated]
  ./src/assets/qianfeng-sem.jpg 42 bytes (javascript) 637
  KiB (asset) [built] [code generated]
json modules 565 bytes
  ./src/assets/json/data.toml 188 bytes [built] [code
  generated]
  ./src/assets/json/data.yaml 188 bytes [built] [code
  generated]
  ./src/assets/json/data.json5 189 bytes [built] [code
  generated]
css ../node_modules/css-loader/dist/cjs.js!./src/style.css
  2.65 KiB [built] [code generated]
webpack 5.54.0 compiled successfully in 1237 ms
```

观察一下：

```
assets by status 1.45 MiB [emitted]
  asset shared.bundle.js 1.39 MiB [emitted] (name: shared)
  asset index.bundle.js 57.1 KiB [emitted] (name: index)
  asset another.bundle.js 1.53 KiB [emitted] (name: another)
  asset app.html 487 bytes [emitted]
```

`index.bundle.js` 与 `another.bundle.js` 共享的模块 `lodash.js` 被打包到一个单独的文件 `shared.bundle.js` 中。

- **SplitChunksPlugin**

[SplitChunksPlugin](#) 插件可以将公共的依赖模块提取到已有的入口 chunk 中，或者提取到一个新生成的 chunk。让我们使用这个插件，将之前的示例中重复的 `lodash` 模块去除：

```
entry: {
  index: './src/index.js',
  another: './src/another-module.js'
},

optimization: {
  splitChunks: {
    chunks: 'all',
  },
},
```

09-code-splitting/webpack.config.js

```
//...

module.exports = {
// entry: {
//   index: {
//     import: './src/index.js',
//     dependon: 'shared',
//   },
//   another: {
//     import: './src/another-module.js',
//     dependon: 'shared',
//   },
//   shared: 'lodash',
// },
entry: {
  index: './src/index.js',
  another: './src/another-module.js'
```

```
},  
  
//...  
  
optimization: {  
  //...  
  
  splitChunks: {  
    chunks: 'all',  
  }  
},  
}
```

执行编译：

```
[felix] 09-code-splitting $ npx webpack  
assets by status 744 KiB [cached] 4 assets  
assets by status 1.46 MiB [emitted]  
  asset vendors-node_modules_lodash_lodash_js.bundle.js 1.37  
MiB [emitted] (id hint: vendors)  
  asset index.bundle.js 75.3 KiB [emitted] (name: index)  
  asset another.bundle.js 17.2 KiB [emitted] (name: another)  
  asset app.html 518 bytes [emitted]  
Entrypoint index 1.45 MiB (740 KiB) = vendors-  
node_modules_lodash_lodash_js.bundle.js 1.37 MiB  
styles/4a9cff551c7a105e1554.css 3.81 KiB index.bundle.js 75.3  
KiB 3 auxiliary assets  
Entrypoint another 1.39 MiB = vendors-  
node_modules_lodash_lodash_js.bundle.js 1.37 MiB  
another.bundle.js 17.2 KiB  
runtime modules 8.1 KiB 17 modules  
cacheable modules 549 KiB (javascript) 738 KiB (asset) 2.65  
KiB (css/mini-extract)  
  javascript modules 546 KiB  
    modules by path ../node_modules/ 540 KiB 9 modules  
    modules by path ./src/ 5.57 KiB 8 modules  
    asset modules 3.1 KiB (javascript) 738 KiB (asset)  
      ./src/assets/img-1.png 42 bytes (javascript) 101 KiB  
(asset) [built] [code generated]  
      ./src/assets/webpack-logo.svg 2.99 KiB [built] [code  
generated]  
      ./src/assets/example.txt 25 bytes [built] [code generated]  
      ./src/assets/qianfeng-sem.jpg 42 bytes (javascript) 637  
KiB (asset) [built] [code generated]  
  json modules 565 bytes
```

```
./src/assets/json/data.toml 188 bytes [built] [code generated]
./src/assets/json/data.yaml 188 bytes [built] [code generated]
./src/assets/json/data.json5 189 bytes [built] [code generated]
css ../node_modules/css-loader/dist/cjs.js!./src/style.css 2.65 KiB [built] [code generated]
webpack 5.54.0 compiled successfully in 914 ms
```

观察一下：

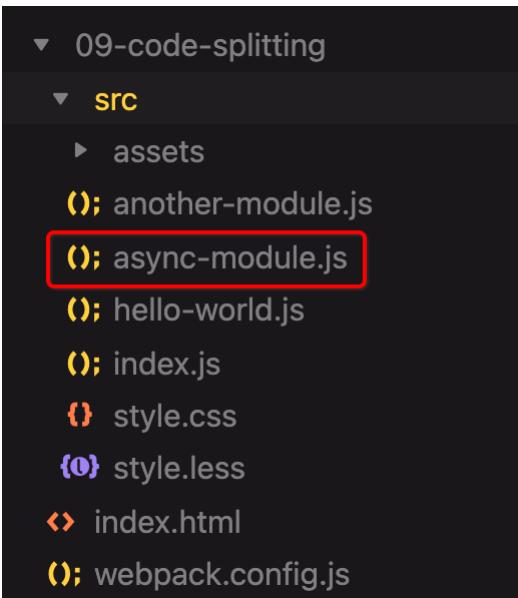
```
assets by status 1.46 MiB [emitted]
asset vendors-node_modules_lodash_lodash_js.bundle.js 1.37 MiB [emitted] (id hint: vendors)
asset index.bundle.js 75.3 KiB [emitted] (name: index)
asset another.bundle.js 17.2 KiB [emitted] (name: another)
asset app.html 518 bytes [emitted]
```

使用 [optimization.splitChunks](#) 配置选项之后，现在应该可以看出，`index.bundle.js` 和 `another.bundle.js` 中已经移除了重复的依赖模块。需要注意的是，插件将 `lodash` 分离到单独的 chunk，并且将其从 main bundle 中移除，减轻了大小。

1.8.3 动态导入

当涉及到动态代码拆分时，webpack 提供了两个类似的技术。第一种，也是推荐选择的方式是，使用符合 [ECMAScript 提案](#) 的 [import\(\)](#) 语法 来实现动态导入。第二种，则是 webpack 的遗留功能，使用 webpack 特定的 [require.ensure](#)。让我们先尝试使用第一种.....

创建 `async-module.js` 文件：



内容如下：

09-code-splitting/src/async-module.js

```
function getComponent() {
  return import('lodash')
    .then(({_
      default: _
    }) => {
      const element = document.createElement('div')

      element.innerHTML = _.join(['Hello', 'webpack'], ' ')
      return element
    })
    .catch((error) => 'An error occurred while loading the
component')
}

getComponent().then((component) => {
  document.body.appendChild(component)
})
```

在入口文件中导入：

```
import './async-module'
```

09-code-splitting/src/index.js

```
// 导入模块  
//...  
import './async-module'  
//...
```

执行编译：

```
[felix] 09-code-splitting $ npx webpack  
assets by status 744 KiB [cached] 4 assets  
assets by status 1.53 MiB [compared for emit]  
  assets by chunk 1.46 MiB (id hint: vendors)  
    asset vendors-node_modules_lodash_lodash_js.bundle.js 1.37  
MiB [compared for emit] (id hint: vendors)  
      asset vendors-  
node_modules_babel_runtime_regenerator_index_js-node_modules_css-  
loader_dist_runtime_-86adfe.bundle.js 93.8 KiB [compared for  
emit] (id hint: vendors)  
  asset index.bundle.js 54.3 KiB [compared for emit] (name:  
index)  
  asset another.bundle.js 17.2 KiB [compared for emit] (name:  
another)  
  asset app.html 658 bytes [compared for emit]  
Entrypoint index 1.52 MiB (740 KiB) = vendors-  
node_modules_lodash_lodash_js.bundle.js 1.37 MiB vendors-  
node_modules_babel_runtime_regenerator_index_js-node_modules_css-  
loader_dist_runtime_-86adfe.bundle.js 93.8 KiB  
styles/4a9cff551c7a105e1554.css 3.81 KiB index.bundle.js 54.3 KiB  
3 auxiliary assets  
Entrypoint another 1.39 MiB = vendors-  
node_modules_lodash_lodash_js.bundle.js 1.37 MiB  
another.bundle.js 17.2 KiB  
runtime modules 9.21 KiB 18 modules  
....
```

从打印的结果看，除了公共的 `lodash` 代码被单独打包到一个文件外，还生成了一个 `vendors-node_modules_babel_runtime_regenerator_index_js-node_modules_css-loader_dist_runtime_-86adfe.bundle.js` 文件。



我们看到，静态和动态载入的模块都正常工作了。

1.8.4 懒加载

懒加载或者按需加载，是一种很好的优化网页或应用的方式。这种方式实际上是先把你的代码在一些逻辑断点处分离开，然后在一些代码块中完成某些操作后，立即引用或即将引用另外一些新的代码块。这样加快了应用的初始加载速度，减轻了它的总体体积，因为某些代码块可能永远不会被加载。

创建一个 `math.js` 文件，在主页面中通过点击按钮调用其中的函数：

09-code-splitting/src/math.js

```
export const add = () => {
    return x + y
}

export const minus = () => {
    return x - y
}
```

编辑 `index.js` 文件：

```

const button = document.createElement('button')
button.textContent = '点击执行加法运算'
button.addEventListener('click', () => {
  import(/* webpackChunkName: 'math' */ './math.js').then(({ add })
}) => {
  console.log(add(4, 5))
})
}
document.body.appendChild(button)

```

这里有句注释，我们把它称为 webpack 魔法注释： `webpackChunkName: 'math'`，告诉webpack打包生成的文件名为 `math`。

启动服务，在浏览器上查看：



第一次加载完页面，`math.bundle.js`不会加载，当点击按钮后，才加载`math.bundle.js`文件。

1.8.5 预获取/预加载模块

Webpack v4.6.0+ 增加了对预获取和预加载的支持。

在声明 `import` 时，使用下面这些内置指令，可以让 webpack 输出 "resource hint(资源提示)"，来告知浏览器：

- **prefetch**(预获取): 将来某些导航下可能需要的资源
- **preload**(预加载): 当前导航下可能需要资源

下面这个 prefetch 的简单示例中，编辑 `index.js` 文件：

```
const button = document.createElement('button')
button.textContent = '点击执行加法运算'
button.addEventListener('click', () => {
  import(/* webpackChunkName: 'math', webpackPrefetch: true */ './math.js').then(({ add }) => {
    console.log(add(4, 5))
  })
})
document.body.appendChild(button)
```

添加第二句魔法注释：`webpackPrefetch: true`

告诉 webpack 执行预获取。这会生成 `<link rel="prefetch" href="math.js">` 并追加到页面头部，指示着浏览器在闲置时间预取 `math.js` 文件。

09-code-splitting/src/index.js

```
// 导入模块
//...
import './async-module'

//...

const button = document.createElement('button')
button.textContent = '点击执行加法运算'
button.addEventListener('click', () => {
  import( /* webpackChunkName: 'math', webpackPrefetch: true */ './math.js').then(({ add }) => {
    console.log(add(4, 5))
  })
})
document.body.appendChild(button)
```

启动服务，在浏览器上查看：

The screenshot shows a webpage with the title 'Web大前端课程' (Large Front-end Course) by QianFeng. The page includes a 'Hello webpack' logo and a large 'webpack' logo. On the right side, the Chrome Network tab is open, showing a list of resources being loaded. One resource, 'math.bundle.js', is highlighted with a red box.

我们发现，在还没有点击按钮时，`math.bundle.js`就已经下载下来了。同时，在`app.html`里webpack自动添加了一句：

The screenshot shows the same webpage as the previous one, but the developer tools Network tab is now visible. A red box highlights a specific line of code in the head section of the HTML document, which contains a `<link rel='prefetch' as='script'` tag pointing to 'math.bundle.js'.

点击按钮，会立即调用已经下载好的`math.bundle.js`文件中的`add`方法：

点击按钮，执行 `4+5` 的加法运算。

与 `prefetch` 指令相比，`preload` 指令有许多不同之处：

- `preload chunk` 会在父 `chunk` 加载时，以并行方式开始加载。 `prefetch chunk` 会在父 `chunk` 加载结束后开始加载。
- `preload chunk` 具有中等优先级，并立即下载。 `prefetch chunk` 在浏览器闲置时下载。
- `preload chunk` 会在父 `chunk` 中立即请求，用于当下时刻。 `prefetch chunk` 会用于未来的某个时刻。
- 浏览器支持程度不同。

创建一个 `print.js` 文件：

```
export const print = () => {
  console.log('preload chunk.')
}
```

修改 `index.js` 文件：

```
const button2 = document.createElement('button')
button2.textContent = '点击执行字符串打印'
button2.addEventListener('click', () => {
  import(/* webpackChunkName: 'print', webpackPreload: true */ './print.js').then(({ print }) => {
    print(4, 5)
  })
})
document.body.appendChild(button2)
```

09-code-splitting/src/index.js

```
// 导入模块
//...

import './async-module'

//...

const button2 = document.createElement('button')
button2.textContent = '点击执行字符串打印'
button2.addEventListener('click', () => {
  import( /* webpackChunkName: 'print', webpackPreload: true */ './print.js').then(({ print }) => {
    print
  }) => {
    print()
  }
})
document.body.appendChild(button2)
```

启动服务，打开浏览器：

Name	Status	Type	S.	Waterfall
app.html	200	document	9...	
4a9cff51c7a105e1554.css	200	stylesheet	2...	
vendors-node_modules_lodash lodash_js...	200	script	4...	
vendors-node_modules_babel_runtime_re...	200	script	3...	
index.bundle.js	200	script	3...	
another.bundle.js	200	script	2...	
math.bundle.js	200	javascript	1...	
8ec2798f814745a7c9b.png	200	png	1...	
data:image/svg+xml,...	200	svg+xml	(...)	
33120e6c4bd92df7bec8.jpg	200	jpeg	6...	
65b194f1f711865371d1.ttf	200	font	1...	
ws	101	websocket	0...	
ws	101	websocket	0...	

仔细观察，发现 `print.bundle.js` 未被下载，因为我们配置的是 `webpackPreload`，是在父 chunk 加载时，以并行方式开始加载。点击按钮才加载的模块不会事先加载的。

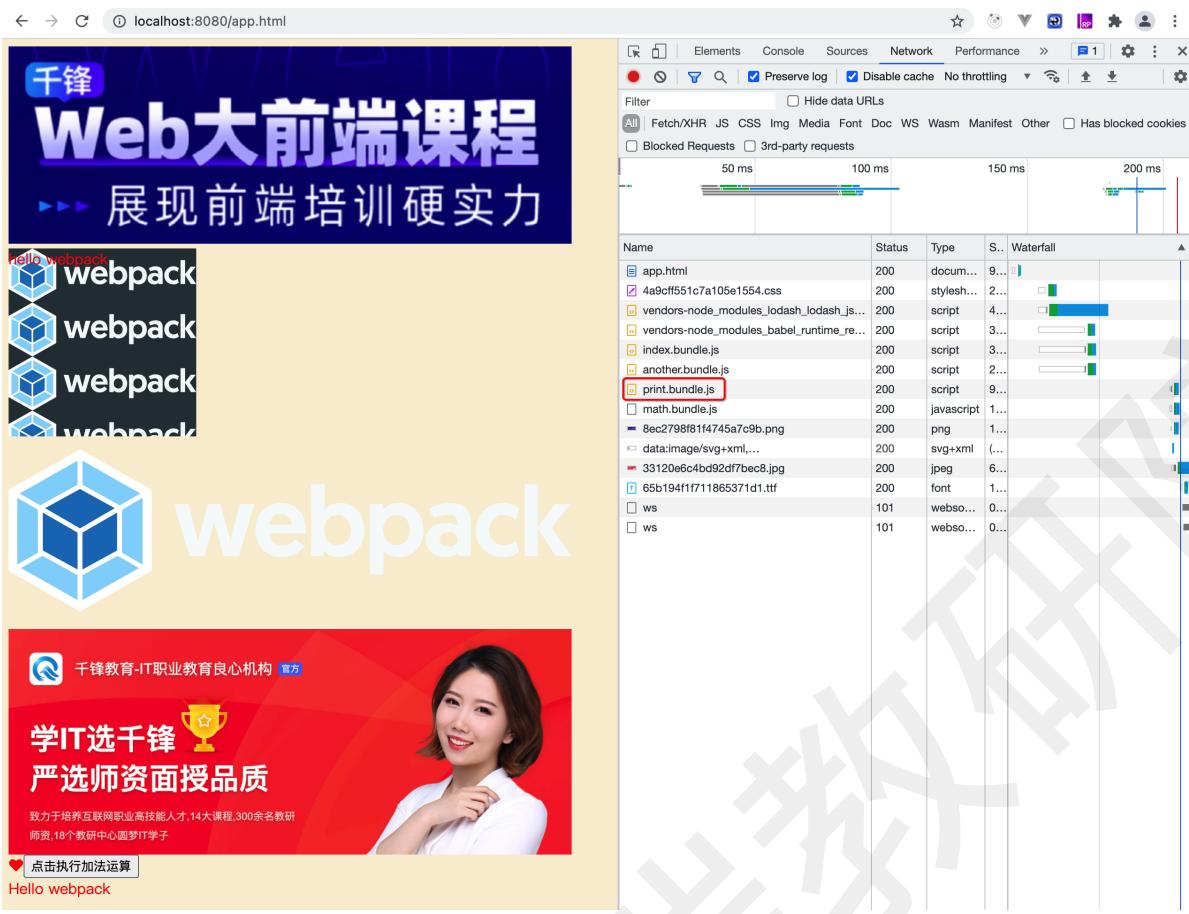
我们修改一下引入方式：

09-code-splitting/src/index.js

```
//...
import(/* webpackChunkName: 'print', webpackPreload: true */)
  './print.js').then(({

  print
}) => {
  print()
})
```

再次刷新浏览器页面：



print.bundle.js 被加载下来，是和当前 index.bundle.js 并行加载的。

1.9 缓存

以上，我们使用 webpack 来打包我们的模块化后的应用程序，webpack 会生成一个可部署的 `/dist` 目录，然后把打包后的内容放置在此目录中。只要 `/dist` 目录中的内容部署到 server 上，client（通常是浏览器）就能够访问此 server 的网站及其资源。而最后一步获取资源是比较耗费时间的，这就是为什么浏览器使用一种名为 **缓存** 的技术。可以通过命中缓存，以降低网络流量，使网站加载速度更快，然而，如果我们在部署新版本时不更改资源的文件名，浏览器可能会认为它没有被更新，就会使用它的缓存版本。由于缓存的存在，当你需要获取新的代码时，就会显得很棘手。

本节通过必要的配置，以确保 webpack 编译生成的文件能够被客户端缓存，而在文件内容变化后，能够请求到新的文件。

1.9.1 输出文件的文件名

我们可以通过替换 `output.filename` 中的 `substitutions` 设置，来定义输出文件的名称。webpack 提供了一种使用称为 **substitution(可替换模板字符串)** 的方式，通过带括号字符串来模板化文件名。其中，`[contenthash]` substitution 将根据资源内容创建出唯一 hash。当资源内容发生变化时，`[contenthash]` 也会发生变化。

修改配置文件：

```
module.exports = {  
  output: {  
    filename: '[name].[contenthash].js',  
  },  
};
```

10-caching/webpack.config.js

```
//...  
  
module.exports = {  
//...  
output: {  
  filename: '[name].[contenthash].js',  
  
  //...  
},  
  
//...  
}
```

执行打包编译：

```
▼ 10-caching  
  ▼ dist  
    ▶ images  
    ▶ styles  
    ⚠ another.ae60e625de36e4f2cdaf.js  
    ⚠ app.html  
    ⚠ index.ebc440b51e90b14c84bc.js  
    ⚠ math.08eaa54ca2e0547b22d4.js  
    ⚠ print.15cb6b5ff55c3bbf13c3.js  
    ⚠ vendors-node_modules_babel_runtime_rege...  
    ⚠ vendors-node_modules_lodash lodash_js.b...
```

可以看到，bundle 的名称是它内容（通过 hash）的映射。如果我们不做修改，然后再次运行构建，文件名会保持不变。

1.9.2 缓存第三方库

将第三方库(library)（例如 `lodash`）提取到单独的 `vendor` chunk 文件中，是比较推荐的做法，这是因为，它们很少像本地的源代码那样频繁修改。因此通过实现以上步骤，利用 client 的长效缓存机制，命中缓存来消除请求，并减少向 server 获取资源，同时还能保证 client 代码和 server 代码版本一致。我们在 `optimization.splitChunks` 添加如下 `cacheGroups` 参数并构建：

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

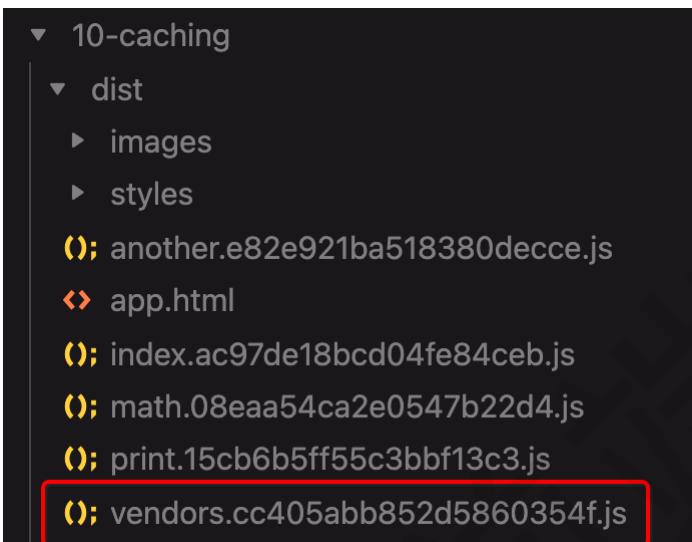
10-caching/webpack.config.js

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

执行编译：

```
[felix] 10-caching $ npx webpack
assets by status 746 KiB [cached] 6 assets
assets by status 1.55 MiB [emitted]
  asset vendors.cc405abb852d5860354f.js 1.46 MiB [emitted]
  [immutable] (name: vendors) (id hint: vendor)
    asset index.ac97de18bcd04fe84ceb.js 67.4 KiB [emitted]
  [immutable] (name: index)
    asset another.e82e921ba518380decce.js 17.2 KiB [emitted]
  [immutable] (name: another)
    asset app.html 530 bytes [emitted]
  ...

```



1.9.3 将 js 文件放到一个文件夹中

目前，全部 js 文件都在 `dist` 文件夹根目录下，我们尝试把它们放到一个文件夹中，这个其实也简单，修改配置文件：

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

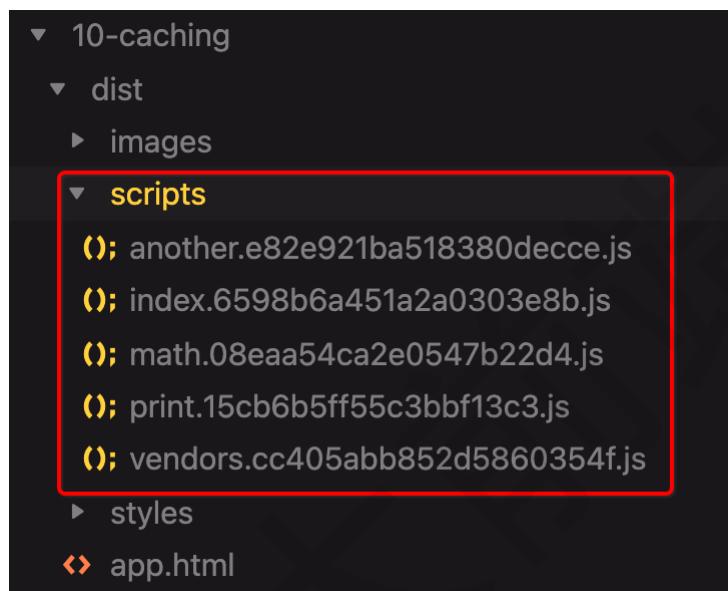
10-caching/webpack.config.js

```
//...

module.exports = {
//...

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

我们在输出配置中修改 `filename`，在前面加上路径即可。执行编译：



截至目前，我们已经把 JS 文件、样式文件及图片等资源文件分别放到了 `scripts`、`styles`、`images` 三个文件夹中。

1.10 拆分开发环境和生产环境配置

现在，我们只能手工的来调整 `mode` 选项，实现生产环境和开发环境的切换，且很多配置在生产环境和开发环境中存在不一致的情况，比如开发环境没有必要设置缓存，生产环境还需要设置公共路径等等。

本节介绍拆分开发环境和生产环境，让打包更灵活。

1.10.1 公共路径

`publicPath` 配置选项在各种场景中都非常有用。你可以通过它来指定应用程序中所有资源的基础路径。

- 基于环境设置

在开发环境中，我们通常有一个 `assets/` 文件夹，它与索引页面位于同一级别。这没太大问题，但是，如果我们将所有静态资源托管至 CDN，然后想在生产环境中使用呢？

想要解决这个问题，可以直接使用一个 environment variable(环境变量)。假设我们有一个变量 `ASSET_PATH`：

```
import webpack from 'webpack';

// 尝试使用环境变量，否则使用根路径
const ASSET_PATH = process.env.ASSET_PATH || '/';

export default {
  output: {
    publicPath: ASSET_PATH,
  },

  plugins: [
    // 这可以帮助我们在代码中安全地使用环境变量
    new webpack.DefinePlugin({
      'process.env.ASSET_PATH': JSON.stringify(ASSET_PATH),
    }),
  ],
};

};
```

11-multiple-env/webpack.config.js

```
//...
import webpack from 'webpack';

// 尝试使用环境变量，否则使用根路径
const ASSET_PATH = process.env.ASSET_PATH || '/';

export default {
  output: {
    //...

    publicPath: ASSET_PATH,
```

```
},  
  
plugins: [  
    // 这可以帮助我们在代码中安全地使用环境变量  
    new webpack.DefinePlugin({  
        'process.env.ASSET_PATH': JSON.stringify(ASSET_PATH),  
    }),  
  
    //...  
],  
};
```

• Automatic publicPath

有可能你事先不知道 publicPath 是什么，webpack 会自动根据 `import.meta.url`、`document.currentScript`、`script.src` 或者 `self.location` 变量设置 publicPath。你需要做的是将 `output.publicPath` 设为 `'auto'`：

```
module.exports = {  
    output: {  
        publicPath: 'auto',  
    },  
};
```

请注意在某些情况下不支持 `document.currentScript`，例如：IE 浏览器，你不得不引入一个 polyfill，例如 [currentScript Polyfill](#)。

1.10.2 环境变量

想要消除 `webpack.config.js` 在 [开发环境](#) 和 [生产环境](#) 之间的差异，你可能需要环境变量(environment variable)。

webpack 命令行 [环境配置](#) 的 `--env` 参数，可以允许你传入任意数量的环境变量。而在 `webpack.config.js` 中可以访问到这些环境变量。例如，`--env production` 或 `--env goal=local`。

```
npx webpack --env goal=local --env production --progress
```

对于我们的 webpack 配置，有一个必须要修改之处。通常，`module.exports` 指向配置对象。要使用 `env` 变量，你必须将 `module.exports` 转换成一个函数：

```
//...
module.exports = (env) => {
  return {
    //...
    // 根据命令行参数 env 来设置不同环境的 mode
    mode: env.production ? 'production' : 'development',
    //...
  }
}
```

1.10.3 拆分配置文件

目前，生产环境和开发环境使用的是一个配置文件，我们需要将这两个文件单独放到不同的配置文件中。如 `webpack.config.dev.js`（开发环境配置）和 `webpack.config.prod.js`（生产环境配置）。在项目根目录下创建一个配置文件夹 `config` 来存放他们。

`webpack.config.dev.js` 配置如下：

11-multiple-env/config/webpack.config.dev.js

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

const toml = require('toml')
const yaml = require('yaml')
const json5 = require('json5')

module.exports = {
  entry: {
    index: './src/index.js',
    another: './src/another-module.js'
  },

  output: {
    filename: 'scripts/[name].js',
    path: path.resolve(__dirname, './dist'),
    clean: true,
    assetModuleFilename: 'images/[contenthash][ext]'
  },

  mode: 'development',
```

```
devtool: 'inline-source-map',  
  
plugins: [  
  new HtmlWebpackPlugin({  
    template: './index.html',  
    filename: 'app.html',  
    inject: 'body'  
  }),  
  
  new MiniCssExtractPlugin({  
    filename: 'styles/[contenthash].css'  
  })  
,  
  
devServer: {  
  static: './dist'  
},  
  
module: {  
  rules: [  
    {  
      test: /\.png$/,  
      type: 'asset/resource',  
      generator: {  
        filename: 'images/[contenthash][ext]'  
      }  
    },  
  
    {  
      test: /\.svg$/,  
      type: 'asset/inline'  
    },  
  
    {  
      test: /\.txt$/,  
      type: 'asset/source'  
    },  
  
    {  
      test: /\.jpg$/,  
      type: 'asset',  
      parser: {  
        dataUrlCondition: {  
          maxSize: 4 * 1024 * 1024  
        }  
      }  
    }  
  ]  
},
```

```
        },
    },
    {
        test: /\.css|less$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader', 'less-loader']
    },
    {
        test: /\.(woff|woff2|eot|ttf|otf)$/,
        type: 'asset/resource'
    },
    {
        test: /\.csv|tsv$/,
        use: 'csv-loader'
    },
    {
        test: /\.xml$/,
        use: 'xml-loader'
    },
    {
        test: /\.toml$/,
        type: 'json',
        parser: {
            parse: toml.parse
        }
    },
    {
        test: /\.yaml$/,
        type: 'json',
        parser: {
            parse: yaml.parse
        }
    },
    {
        test: /\.json5$/,
        type: 'json',
        parser: {
            parse: json5.parse
        }
    }
]
```

```
        }
    },
    {
        test: /\.js$/,
        exclude: /node_modules/,
        use: [
            {
                loader: 'babel-loader',
                options: {
                    presets: ['@babel/preset-env'],
                    plugins: [
                        [
                            '@babel/plugin-transform-runtime'
                        ]
                    ]
                }
            }
        ]
    },
    optimization: {
        splitChunks: {
            cacheGroups: {
                vendor: {
                    test: /[\\/]node_modules[\\/]/,
                    name: 'vendors',
                    chunks: 'all'
                }
            }
        }
    }
}
```

webpack.config.prod.js 配置如下：

11-multiple-env/config/webpack.config.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')
```

```
const toml = require('toml')
const yaml = require('yaml')
const json5 = require('json5')

module.exports = {
entry: {
index: './src/index.js',
another: './src/another-module.js'
},

output: {
filename: 'scripts/[name].[contenthash].js',
// 打包的dist文件夹要放到上一层目录
path: path.resolve(__dirname, '../dist'),
clean: true,
assetModuleFilename: 'images/[contenthash][ext]',
publicPath: 'http://localhost:8080/'
},
mode: 'production',

plugins: [
new HtmlWebpackPlugin({
template: './index.html',
filename: 'app.html',
inject: 'body'
}),
new MiniCssExtractPlugin({
filename: 'styles/[contenthash].css'
})
],
module: {
rules: [
{
test: /\.png$/,
type: 'asset/resource',
generator: {
filename: 'images/[contenthash][ext]'
}
},
{
test: /\.svg$/,

```

```
        type: 'asset/inline'
    },
    {
        test: /\.txt$/,
        type: 'asset/source'
    },
    {
        test: /\.jpg$/,
        type: 'asset',
        parser: {
            dataurlCondition: {
                maxSize: 4 * 1024 * 1024
            }
        }
    },
    {
        test: /\.(css|less)\$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader', 'less-loader']
    },
    {
        test: /\.(woff|woff2|eot|ttf|otf)\$/,
        type: 'asset/resource'
    },
    {
        test: /\.csv$/,
        use: 'csv-loader'
    },
    {
        test: /\.xml$/,
        use: 'xml-loader'
    },
    {
        test: /\.toml$/,
        type: 'json',
        parser: {
            parse: toml.parse
        }
    }
]
```

```
},  
  
{  
  test: /\.yaml$/,
  type: 'json',
  parser: {
    parse: yaml.parse
  }
},  
  
{  
  test: /\.json5$/,
  type: 'json',
  parser: {
    parse: json5.parse
  }
},  
  
{  
  test: /\.js$/,
  exclude: /node_modules/,
  use: [
    loader: 'babel-loader',
    options: {
      presets: ['@babel/preset-env'],
      plugins: [
        [
          '@babel/plugin-transform-runtime'
        ]
      ]
    }
  ]
},  
  
optimization: {  
  minimizer: [  
    new CssMinimizerPlugin()  
  ],  
  
  splitChunks: {  
    cacheGroups: {  
      vendor: {  
        test: /[\\/]node_modules[\\/]/,
```

```
        name: 'vendors',
        chunks: 'all'
    }
}
},
//关闭 webpack 的性能提示
performance: {
hints:false
}
}
```

拆分成两个配置文件后，分别运行这两个文件：

开发环境：

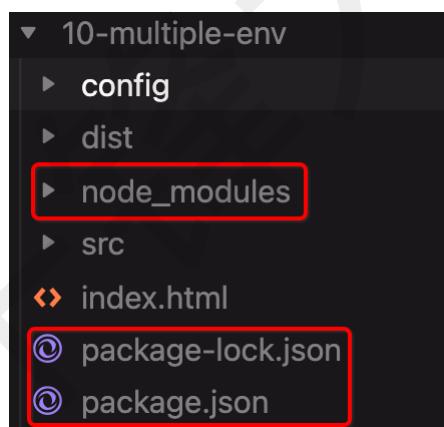
```
[felix] 10-multiple-env $ npx webpack serve -c
./config/webpack.config.dev.js
```

生产环境：

```
[felix] 10-multiple-env $ npx webpack -c
./config/webpack.config.prod.js
```

1.10.4 npm 脚本

每次打包或启动服务时，都需要在命令行里输入一长串的命令。我们将父目录的 `package.json`、`node_modules` 与 `package-lock.json` 拷贝到当前目录下，



配置 npm 脚本来简化命令行的输入，这时可以省略 `npx`：

```
11-multiple-env/package.json
```

```
{  
  "scripts": {  
    "start": "webpack serve -c ./config/webpack.config.dev.js",  
    "build": "webpack -c ./config/webpack.config.prod.js"  
  }  
}
```

开发环境运行脚本：

```
[felix] 10-multiple-env $ npm run start
```

```
[felix] 10-multiple-env $ npm run build
```

1.10.5 提取公共配置

这时，我们发现这两个配置文件里存在大量的重复代码，可以手动的将这些重复的代码单独提取到一个文件里，

创建 `webpack.config.common.js`，配置公共的内容：

11-multiple-env/config/webpack.config.common.js

```
const path = require('path')  
const HtmlWebpackPlugin = require('html-webpack-plugin')  
const MiniCssExtractPlugin = require('mini-css-extract-plugin')  
  
const toml = require('toml')  
const yaml = require('yaml')  
const json5 = require('json5')  
  
module.exports = {  
  entry: {  
    index: './src/index.js',  
    another: './src/another-module.js'  
  },  
  
  output: {  
    // 注意这个dist的路径设置成上一级  
    path: path.resolve(__dirname, '../dist'),  
    clean: true,  
    assetModuleFilename: 'images/[contenthash][ext]',  
  },  
  
  plugins: [
```

```
new HtmlWebpackPlugin({
  template: './index.html',
  filename: 'app.html',
  inject: 'body'
}),

new MiniCssExtractPlugin({
  filename: 'styles/[contenthash].css'
})
],

module: {
  rules: [
    {
      test: /\.png$/,
      type: 'asset/resource',
      generator: {
        filename: 'images/[contenthash][ext]'
      }
    },
    {
      test: /\.svg$/,
      type: 'asset/inline'
    },
    {
      test: /\.txt$/,
      type: 'asset/source'
    },
    {
      test: /\.jpg$/,
      type: 'asset',
      parser: {
        dataUrlCondition: {
          maxSize: 4 * 1024
        }
      }
    },
    {
      test: /\.(css|less)$/,
      use: [MiniCssExtractPlugin.loader, 'css-loader', 'less-loader']
    }
  ]
}
```

```
},  
{  
  test: /\.woff|woff2|eot|ttf|otf$/,
  type: 'asset/resource'
},  
  
{  
  test: /\.csv|tsv$/,
  use: 'csv-loader'
},  
  
{  
  test: /\.xml$/,
  use: 'xml-loader'
},  
  
{  
  test: /\.toml$/,
  type: 'json',
  parser: {
    parse: toml.parse
  }
},  
  
{  
  test: /\.yaml$/,
  type: 'json',
  parser: {
    parse: yaml.parse
  }
},  
,  
{  
  test: /\.json5$/,
  type: 'json',
  parser: {
    parse: json5.parse
  }
},  
,  
{  
  test: /\.js$/,
  exclude: /node_modules/,
  use: {
    loader: 'babel-loader',
    options: {
      presets: ['es2015']
    }
  }
}
```

```
    loader: 'babel-loader',
    options: {
      presets: ['@babel/preset-env'],
      plugins: [
        [
          '@babel/plugin-transform-runtime'
        ]
      ]
    }
  }
],
},
optimization: {
  splitChunks: {
    cacheGroups: {
      vendor: {
        test: /[\\/]node_modules[\\/]/,
        name: 'vendors',
        chunks: 'all'
      }
    }
  }
},
//关闭 webpack 的性能提示
performance: {
  hints: false
}
}
```

改写 `webpack.config.dev.js`:

11-multiple-env/config/webpack.config.dev.js

```
module.exports = {
  // 开发环境不需要配置缓存
  output: {
    filename: 'scripts/[name].js',
  },
  // 开发模式
  mode: 'development',
```

```
// 配置 source-map
devtool: 'inline-source-map',

// 本地服务配置
devServer: {
  static: './dist'
}
```

修改 `webpack.config.prod.js`:

11-multiple-env/config/webpack.config.prod.js

```
const CssMinimizerPlugin = require('css-minimizer-webpack-
plugin')

module.exports = {
  // 生产环境需要缓存
  output: {
    filename: 'scripts/[name].[contenthash].js',
    publicPath: 'http://localhost:8080/'
  },

  // 生产环境模式
  mode: 'production',

  // 生产环境 css 压缩
  optimization: {
    minimizer: [
      new CssMinimizerPlugin()
    ]
  }
}
```

1.10.6 合并配置文件

配置文件拆分好后，新的问题来了，如何保证配置合并没用问题呢？[webpack-merge](#) 这个工具可以完美解决这个问题。

npmjs.com/package/webpack-merge

Search packages

Search Sign Up Sign In

webpack-merge TS

5.8.0 • Public • Published 5 months ago

Readme Explore BETA 2 Dependencies 4,387 Dependents 87 Versions

financial contributors build passing codecov 99%

webpack-merge - Merge designed for Webpack

webpack-merge provides a `merge` function that concatenates arrays and merges objects creating a new object. If functions are encountered, it will execute them, run the results through the algorithm, and then wrap the returned values within a function again.

This behavior is particularly useful in configuring webpack although it has uses beyond it. Whenever you need to merge configuration objects, **webpack-merge** can come in handy.

`merge(...configuration | [...configuration])`

`merge` is the core, and the most important idea, of the API. Often this is all you need unless you

Install

```
> npm i webpack-merge
```

Repository [github.com/survivejs/webpack-merge](#)

Homepage [github.com/survivejs/webpack-merge](#)

Weekly Downloads 7,825,713

Version 5.8.0 License MIT

安装 `webpack-merge`:

```
[felix] felixcourses $ npm install webpack-merge -D
```

创建 `webpack.config.js`, 合并代码:

11-multiple-env/config/webpack.config.js

```
const { merge } = require('webpack-merge')

const commonConfig = require('./webpack.config.common.js')

const productionConfig = require('./webpack.config.prod.js')

const developmentConfig = require('./webpack.config.dev')

module.exports = (env) => {
  switch(true) {
    case env.development:
      return merge(commonConfig, developmentConfig)
    case env.production:
      return merge(commonConfig, productionConfig)
    default:
      throw new Error('No matching configuration was found!')
  }
}
```

-- 本篇完 --

二、高级应用篇

上述我们基于webpack构建了我们的基础工程化环境，将我们认为需要的功能配置了上去。除开公共基础配置之外，我们意识到两点：

1. 开发环境(mode=development),追求强大的开发功能和效率，配置各种方便开发的功能；
2. 生产环境(mode=production),追求更小更轻量的bundle(即打包产物)；

接下来基于我们的开发需求，完善我们的工程化配置的同时，来介绍一些常用并强大的工具。

2.1 提高开发效率，完善团队开发规范

2.1.1 source-map

作为一个开发工程师——无论是什么开发，要求开发环境最不可少的一点功能就是——debug功能。之前我们通过webpack, 将我们的源码打包成了 bundle.js。试想：实际上客户端(浏览器)读取的是打包后的 bundle.js,那么当浏览器执行代码报错的时候，报错的信息自然也是bundle的内容。我们如何将报错信息(bundle错误的语句及其所在行列)映射到源码上？

是的，source-map。

webpack已经内置了sourcemap的功能，我们只需要通过简单的配置，将可以开启它。

```
module.exports = {
  // 开启 source map
  // 开发中推荐使用 'source-map'
  // 生产环境一般不开启 sourcemap
  devtool: 'source-map',
}
```

当我们执行打包命令之后，我们发现bundle的最后一行总是会多出一个注释，指向打包出的bundle.map.js(sourcemap文件)。sourcemap文件用来描述 源码文件和bundle文件的代码位置映射关系。基于它，我们将bundle文件的错误信息映射到源码文件上。

除开'source-map'外，还可以基于我们的需求设置其他值，webpack——devtool一共提供了7种SourceMap模式：

模式	解释
eval	每个module会封装到 eval 里包裹起来执行，并且会在末尾追加注释 // @ sourceURL.
source-map	生成一个SourceMap文件.
hidden-source-map	和 source-map 一样，但不会在 bundle 末尾追加注释.
inline-source-map	生成一个 DataUrl 形式的 SourceMap 文件.
eval-source-map	每个module会通过eval()来执行，并且生成一个DataUrl形式的 SourceMap.
cheap-source-map	生成一个没有列信息 (column-mappings) 的SourceMaps文件，不包含loader的 sourcemap (譬如 babel 的 sourcemap)
cheap-module-source-map	生成一个没有列信息 (column-mappings) 的SourceMaps文件，同时 loader 的 sourcemap 也被简化为只包含对应行的。

要注意的是，生产环境我们一般不会开启sourcemap功能，主要有两点原因：

1. 通过bundle和sourcemap文件，可以反编译出源码———也就是说，线上产物有sourcemap文件的话，就意味着有暴漏源码的风险。
2. 我们可以观察到，sourcemap文件的体积相对比较大，这跟我们生产环境的追求不同(生产环境追求更小更轻量的bundle)。

一道思考题：有时候我们期望能第一时间通过线上的错误信息，来追踪到源码位置，从而快速解决掉bug以减轻损失。但又不希望sourcemap文件报漏在生产环境，有比较好的方案吗？

2.1.2 devServer

开发环境下，我们往往需要启动一个web服务，方便我们模拟一个用户从浏览器中访问我们的web服务，读取我们的打包产物，以观测我们的代码在客户端的表现。webpack内置了这样的功能，我们只需要简单的配置就可以开启它。

在此之前，我们需要安装它

```
yarn add -D webpack-dev-server
```

devServer.proxy基于强大的中间件 `http-proxy-middleware` 实现的，因此它支持很多的配置项，我们基于此，可以做应对绝大多数开发场景的定制化配置。

基础使用：

```
const path = require('path');
module.exports = {
  //...
  devServer: {
    // static: {
    //   directory: path.join(__dirname, 'dist'),
    // }, // 默认是把/dist目录当作web服务的根目录
    compress: true, //可选择开启gzsips压缩功能，对应静态资源请求的响应头里的
    Content-Encoding: gzip
    port: 3000, // 端口号
  },
};
```

为了方便，我们配置一下工程的脚本命令，在`package.json`的`scripts`里。

```
{
  //...
  "scripts": {
    //...
    "dev": "webpack serve --mode development"
  }
}
```

注意！如果您需要指定配置文件的路径，请在命令的后面添加 `--config [path]`，比如：

```
webpack serve --mode development --config webpack.config.js
```

这时，当我们`yarn dev`(或者`npm run dev`)时，就可以在日志里看到———它启动了一个http服务。`(webpack-dev-server的最底层实现是源自于node的http模块。)`

```
> webpack serve --mode development

<i> [webpack-dev-server] Project is running at:
<i> [webpack-dev-server] Loopback: http://localhost:3000/
<i> [webpack-dev-server] On Your Network (IPv4):
http://192.168.0.107:3000/
<i> [webpack-dev-server] On Your Network (IPv6):
http://[fe80::1]:3000/
```

```
<i> [webpack-dev-server] Content not from webpack is served from
'/Users/wxy/codeworks/githubPros/demos/webpack5demo/public'
directory
asset bundle.js 289 KiB [emitted] (name: main) 1 related asset
asset index.html 161 bytes [emitted]
runtime modules 27.2 KiB 13 modules
modules by path ./node_modules/ 207 KiB 36 modules
modules by path ./src/ 6.06 KiB
  modules by path ./src/*.css 2.92 KiB
    ./src/styles.css 2.25 KiB [built] [code generated]
      ./node_modules/css-loader/dist/cjs.js!./src/styles.css 684
bytes [built] [code generated]
  modules by path ./src/*.less 3.07 KiB
    ./src/styles.less 2.37 KiB [built] [code generated]
      ./node_modules/css-loader/dist/cjs.js!./node_modules/less-
loader/dist/cjs.js!./src/styles.less 717 bytes [built] [code
generated]
  ./src/index.js 75 bytes [built] [code generated]
webpack 5.60.0 compiled successfully in 1004 ms
```

上述是一个基本的示例，我们可以根据自己的需求定制化devServer的参数对象，比如添加响应头，开启代理来解决跨域问题，http2, https等功能。

- **添加响应头**

有些场景需求下，我们需要为所有响应添加headers, 来对资源的请求和响应打入标志，以便做一些安全防范，或者方便发生异常后做请求的链路追踪。比如：

```
// webpack-config
module.exports = {
  //...
  devServer: {
    headers: {
      'X-Fast-Id': 'p3fdg42njghm34gi9ukj',
    },
  },
};
```

这时，在浏览器中右键检查(或者使用f12快捷键)，在Network一栏查看任意资源访问，我们发现响应头里成功打入了一个FastId。

```
Response Headers
/** some others */
X-Fast-Id: p3fdg42njghm34gi9ukj
```

headers的配置也可以传一个函数：

```
module.exports = {
  //...
  devServer: {
    headers: () => {
      return { 'X-Bar': ['key1=value1', 'key2=value2'] };
    },
  },
};
```

比如我们的标志ID(X-Fast-Id)，很明显这个id不应该写死，而是随机生成的——这时候你就可以用函数实现这个功能。

• 开启代理

我们打包出的 js bundle 里有时会含有一些对特定接口的网络请求.ajax/fetch).要注意，此时客户端地址是在 <http://localhost:3000/> 下，假设我们的接口来自 <http://localhost:4001/>，那么毫无疑问，此时控制台里会报错并提示你跨域。如何解决这个问题？在开发环境下，我们可以使用devServer自带的proxy功能：

```
module.exports = {
  //...
  devServer: {
    proxy: {
      '/api': 'http://localhost:4001',
    },
  },
};
```

现在，对 /api/users 的请求会将请求代理到 <http://localhost:4001/api/users> 。如果不希望传递/api，则需要重写路径：

```
module.exports = {
  //...
  devServer: {
    proxy: {
      '/api': {
        target: 'http://localhost:4001',
        pathRewrite: { '^/api': '' },
      },
    },
  },
};
```

默认情况下，将不接受在 HTTPS 上运行且证书无效的后端服务器。如果需要，可以这样修改配置：

```
module.exports = {
  //...
  devServer: {
    proxy: {
      '/api': {
        target: 'https://other-server.example.com',
        secure: false,
      },
    },
  },
};
```

- **https**

如果想让我们的本地http服务变成https服务，我们只需要这样配置：

```
module.exports = {
  //...
  devServer: {
    https: true, // https://localhost...
  },
};
```

注意，此时我们访问<http://localhost:port>是无法访问我们的服务的，我们需要在地址栏里加前缀：https: 注意：由于默认配置使用的是自签名证书，所以有得浏览器会告诉你这是不安全的，但我们依然可以继续访问它。当然我们也可以提供自己的证书——如果有的话：

```
module.exports = {
  devServer: {
    https: {
      cacert: './server.pem',
      pfx: './server.pfx',
      key: './server.key',
      cert: './server.crt',
      passphrase: 'webpack-dev-server',
      requestCert: true,
    },
  },
};
```

- **http2**

如果想要配置http2，那么直接设置：

```
devServer: {  
  http2: true, // https://localhost...  
},
```

即可，http2默认自带https自签名证书，当然我们仍然可以通过https配置项来使用自己的证书。

- **historyApiFallback**

如果我们的应用是个SPA(单页面应用)，当路由到/some时(可以直接在地址栏里输入)，会发现此时刷新页面后，控制台会报错。

```
GET http://localhost:3000/some 404 (Not Found)
```

此时打开network，刷新并查看，就会发现问题所在——浏览器把这个路由当作了静态资源地址去请求，然而我们并没有打包出/some这样的资源，所以这个访问无疑是404的。如何解决它？这种时候，我们可以通过配置来提供页面代替任何404的静态资源响应：

```
module.exports = {  
  //...  
  devServer: {  
    historyApiFallback: true,  
  },  
};
```

此时重启服务刷新后发现请求变成了index.html。当然，在多数业务场景下，我们需要根据不同的访问路径定制替代的页面，这种情况下，我们可以使用rewrites这个配置项。类似这样：

```
module.exports = {  
  //...  
  devServer: {  
    historyApiFallback: {  
      rewrites: [  
        { from: /^\/$/, to: '/views/landing.html' },  
        { from: /^\/subpage/, to: '/views/subpage.html' },  
        { from: './', to: '/views/404.html' },  
      ],  
    },  
  },  
};
```

- **开发服务器主机**

如果你在开发环境中起了一个devserve服务，并期望你的同事能访问到它，你只需要配置：

```
module.exports = {  
  //...  
  devServer: {  
    host: '0.0.0.0',  
  },  
};
```

这时候，如果你的同事跟你处在同一局域网下，就可以通过局域网ip来访问你的服务啦。

2.1.3 模块热替换与热加载

- **模块热替换**

模块热替换(HMR - hot module replacement)功能会在应用程序运行过程中，替换、添加或删除 [模块](#)，而无需重新加载整个页面。

启用 webpack 的 热模块替换 特性，需要配置devServer.hot参数：

```
module.exports = {  
  //...  
  devServer: {  
    hot: true,  
  },  
};
```

此时我们实现了基本的模块热替换功能。

- HMR 加载样式 如果你配置了style-loader，那么现在已经同样支持样式文件的热替换功能了。

```
module.exports={  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        use: ['style-loader', 'css-loader'],  
      },  
    ],  
  },  
}
```

这是因为style-loader的实现使用了module.hot.accept，在CSS依赖模块更新之后，会对style标签打补丁。从而实现了这个功能。

- 热加载(文件更新时，自动刷新我们的服务和页面) 新版的webpack-dev-server默认已经开启了热加载的功能。它对应的参数是devServer.liveReload，默认为true。注意，如果想要关掉它，要将liveReload设置为false的同时，也要关掉hot

```
module.exports = {
  //...
  devServer: {
    liveReload: false, //默认为true，即开启热更新功能。
  },
};
```

2.1.4 eslint

eslint是用来扫描我们所写的代码是否符合规范的工具。往往我们的项目是多人协作开发的，我们期望统一的代码规范，这时候可以让eslint来对我们进行约束。严格意义上来说，eslint配置跟webpack无关，但在工程化开发环境中，它往往是不可或缺的。

```
yarn add eslint -D
```

配置eslint，只需要在根目录下添加一个.eslintrc文件(或者.eslintrc.json, .js等)。当然，我们可以使用eslint工具来自动生成它：

```
npx eslint --init
```

我们可以看到控制台里的展示：

```
wxy@me1odywxydeMacBook-Pro webpack5demo % npx eslint --init
✓ How would you like to use ESLint? · syntax
✓ What type of modules does your project use? · esm
✓ Which framework does your project use? · react
✓ Does your project use TypeScript? · No / Yes
✓ Where does your code run? · browser
✓ What format do you want your config file to be in? · JSON
```

并生成了一个配置文件 (.eslintrc.json)，这样我们就完成了eslint的基本规则配置。eslint配置文件里的配置项含义如下：

1. env 指定脚本的运行环境。每种环境都有一组特定的预定义全局变量。此处使用的browser预定义了浏览器环境中的全局变量，es6启用除了modules以外的

所有 ECMAScript 6 特性（该选项会自动设置 ecmaVersion 解析器选项为 6）。

2. globals 脚本在执行期间访问的额外的全局变量。也就是 env 中未预定义，但我们需要使用的全局变量。

3. extends 检测中使用的预定义的规则集合。
4. rules 启用的规则及其各自的错误级别，会合并 extends 中的同名规则，定义冲突时优先级更高。

5. parserOptions ESLint 允许你指定你想要支持的 JavaScript 语言选项。

ecmaFeatures 是个对象，表示你想使用的额外的语言特性，这里 jsx 代表启用 JSX。ecmaVersion 用来指定支持的 ECMAScript 版本。默认为 5，即仅支持 es5，你可以使用 6、7、8、9 或 10 来指定你想要使用的 ECMAScript 版本。你也可以使用年份命名的版本号指定为 2015（同 6），2016（同 7），或 2017（同 8）或 2018（同 9）或 2019 (same as 10)。上面的 env 中启用了 es6，自动设置了 ecmaVersion 解析器选项为 6。plugins plugins 是一个 npm 包，通常输出 eslint 内部未定义的规则实现。rules 和 extends 中定义的规则，并不都在 eslint 内部中有实现。比如 extends 中的 plugin:react/recommended，其中定义了规则开关和等级，但是这些规则如何生效的逻辑是在其对应的插件 'react' 中实现的。

接下来，我们在这个配置文件里额外添加一个规则：

```
{  
  // ...others  
  "rules": {  
    "no-console": "warn" // 我们在rules里自定义我们的约束规范  
  }  
}
```

我们通过命令来让eslint检测代码——在我们的package.json里添加一个脚本命令：

```
// package.json  
{  
  "scripts": {  
    // ...others  
    "eslint": "eslint ./src"  
  }  
}
```

然后执行它：

```
xxx@MacBook-Pro webpack5demo % npm run eslint
> eslint src

/Users/wxy/codeworks/githubPros/demos/webpack5demo/src/index.js
 3:1  warning  Unexpected console statement  no-console
 4:1  warning  Unexpected console statement  no-console

✖ 2 problems (0 errors, 2 warnings)
```

果然，因为代码中含有console.log,所以被警告了。

- 结合webpack使用

我们期望eslint能够实时提示报错而不必等待执行命令。这个功能可以通过给自己的IDE(代码编辑器)安装对应的eslint插件来实现。然而，不是每个IDE都有插件，如果不想使用插件，又想实时提示报错，那么我们可以结合 webpack 的打包编译功能来实现。

```
//...
{
  test: /\.js|jsx$/,
  exclude: /node-modules/,
  use: ['babel-loader', 'eslint-loader']
},
// ...
```

因为我们使用了devServer，因此需要在devServer下添加一个对应的配置参数：

```
module.exports = {
  //...
  devServer: {
    liveReload: false, //默认为true，即开启热更新功能。
  },
};
```

现在我们就可以实时地看到代码里的不规范报错啦。

2.1.5 git-hooks 与 husky

为了保证团队里的开发人员提交的代码符合规范，我们可以在开发者上传代码时进行校验。我们常用 husky 来协助进行代码提交时的 eslint 校验。在使用husky之前，我们先来研究一下 `git-hooks`。

- `git-hooks`

我们回到项目的根目录下。运行 ls -a 命令 ——— “-a”可以显示隐藏目录(目录名的第一位是.)。

我们可以看到，存在一个".git"名称的文件夹。

事实上，在我们项目中根目录下运行git命令时，git会根据它来工作。

接下来我们进入到这个文件夹，进一步查看它内部的内容。

```
cd .git  
ls -a
```

我们发现它内部还挺有料！不慌，我们这节课仅仅只讲到其中的一个内容 ——— hooks，可以看到，当前目录下存在一个hooks文件夹，顾名思义，这个文件夹提供了git 命令相关的钩子。

继续往里看。

```
cd hooks  
ls -a
```

ok，那我们可以看到有很多git命令相关的文件名。比如"pre-commit.sample pre-push.sample"。

回到正题——我们期望在git提交(commit)前，对我们的代码进行检测，如果不能通过检测，就无法提交我们的代码。

自然而然的，这个动作的时机应该是？ ———"pre commit",也就是 commit之前。

现在，我们查看一下pre-commit.sample的内容。

```
# cat命令可以查看一个文件的内容  
cat pre-commit.sample
```

OK，它返回了这样的内容，是一串shell注释。翻译过来大概意思是，这是个示例钩子，然后我们看到了这一句话

```
# To enable this hook, rename this file to "pre-commit".
```

意思是要启用这个钩子的话，我们就把这个文件的后缀名去掉。

虽然这样对我们本地来讲是可行的，但要注意，.git文件夹的改动无法同步到远端仓库。

所以我们期望将git-hook的执行权移交到外面来。

好的，我们回到项目的根目录下，然后我们新建一个文件夹，暂时命名为".mygithooks"

然后在此文件夹下，新增一个git-hook文件,命名为"pre-commit"，并写入以下内容：

```
echo pre-commit执行啦
```

好了，我们新建了自己的git-hook，但此时git并不能识别。下面我们执行这行命令：

```
# 项目根目录下  
git config core.hooksPath .mygithooks
```

上述命令给我们自己的文件，配置了git-hook的执行权限。

但这个时候我们git commit的话，可能会报这样的waring，并且没有执行我们的shell：

```
hint: The 'pre-commit' hook was ignored because it's not set as  
executable.  
hint: You can disable this warning with `git config  
advice.ignoredHook false`
```

这是因为我们的操作系统没有给出这个文件的可执行权限。

因此我们得再执行这样一句命令：

```
chmod +x .mygithooks/pre-commit
```

ok！现在我们尝试执行git add . && git commit -m "any meesage"。

我们发现控制台日志会先打印“pre-commit执行啦”。

这意味着成功啦！

总结：

也就是说，我们搞git-hook的话，要分三步走：

1. 新增任意名称文件夹以及文件pre-commit(这个文件名字比如跟要使用的git-hook名字一致)！
2. 执行以下命令来移交git-hook的配置权限

```
git config core.hooksPath .mygithooks
```

3. 给这个文件添加可执行权限：

```
chmod +x .mygithooks/pre-commit
```

然后就成功啦。

这时候我们可以在pre-commit里写任意脚本，比如：

```
eslint src
```

当eslint扫描代码，出现error时，会在结束扫描时将退出码设为大于0的数字。也就是会报错，这时候commit就无法往下执行啦，我们成功的拦截了此次错误操作。

- **husky**

husky在升级到7.x后，做了跟我们上述同样的事。

安装它之前，我们需要在package.json中的script里，先添加

```
"script": {  
    //...others  
    "prepare": "husky install"  
}
```

prepare是一个npm钩子，意思是安装依赖的时候，会先执行husky install命令。

这个命令就做了上述的123这三件事！

我们安装了7.x的husky会发现，项目根目录下生成了.husky的文件夹。

当然，7.x的husky似乎是有bug的，如果不能正常使用，那么我们只需要验证两件事：

1. 是否移交了git-hook的配置权限？

执行命令 "git config --list" 查看core.hooksPath配置是否存在，是否正确指向了.husky。

如果没有，我们只需要手动的给加上就行：

```
git config core.hooksPath .husky
```

2. 是否是可执行文件？

参考上述总结中的3即可

这时我们的husky就正常了。

2.2 模块与依赖

在模块化编程中，开发者将程序分解为功能离散的文件，并称之为模块。每个模块都拥有小于完整程序的体积，使得验证、调试及测试变得轻而易举。精心编写的模块提供了可靠的抽象和封装界限，使得应用程序中每个模块都具备了条理清晰的设计和明确的目的。

Node.js 从一开始就支持模块化编程。但，浏览器端的模块化还在缓慢支持中——截止到2021，大多主流浏览器已支持ESM模块化，因此基于ESM的打包工具生态逐渐开始活跃。

在前端工程化圈子里存在多种支持 JavaScript 模块化的工具，这些工具各有优势和限制。Webpack从这些系统中汲取了经验和教训，并将模块的概念应用到项目的任何文件中。

2.2.1 Webpack 模块与解析原理

在讲webpack模块解析之前，我们先了解下webpack模块的概念，以及简单探究下webpack的具体实现。

1、webpack 模块

- 何为 webpack 模块

能在webpack工程化环境里成功导入的模块，都可以视作webpack模块。与Node.js 模块相比，webpack 模块能以各种方式表达它们的依赖关系。下面是一些示例：

- ES2015 import 语句
- CommonJS require() 语句
- AMD define 和 require 语句
- css/sass/less 文件中的 @import 语句
- stylesheet url(...) 或者 HTML 文件中的图片链接

- 支持的模块类型 Webpack 天生支持如下模块类型：

- ECMAScript 模块
- CommonJS 模块
- AMD 模块
- Assets
- WebAssembly 模块

而我们早就发现——通过 loader 可以使 webpack 支持多种语言和预处理器语法编写的模块。loader 向 webpack 描述了如何处理非原生模块，并将相关依赖引入到你的 bundles 中。包括且不限于：

- TypeScript
- Sass
- Less
- JSON

- YAML

总的来讲，这些都可以被认为是webpack模块。

2、compiler与Resolvers

在我们运行webpack的时候(就是我们执行webpack命令进行打包时)，其实就是相当于执行了下面的代码：

```
const webpack = require('webpack');
const compiler = webpack({
  // ...这是我们配置的webpackconfig对象
})
```

webpack的执行会返回一个描述webpack打包编译整个流程的对象，我们将其称之为compiler。 compiler对象描述整个webpack打包流程——它内置了一个打包状态，随着打包过程的进行，状态也会实时变更，同时触发对应的webpack生命周期钩子。(简单点讲，我们可以将其类比为一个Promise对象，状态从打包前，打包中到打包完成或者打包失败。)每一次webpack打包，就是创建一个compiler对象，走完整个生命周期的过程。

而webpack中所有关于模块的解析，都是compiler对象里的内置模块解析器去工作的——简单点讲，你可以理解为这个对象上的一个属性，我们称之为Resolvers。 webpack的Resolvers解析器的主体功能就是模块解析，它是基于enhanced-resolve这个包实现的。换句话讲，在webpack中，无论你使用怎样的模块引入语句，本质其实都是在调用这个包的api进行模块路径解析。

2.2.2 模块解析(resolve)

webpack通过Resolvers实现了模块之间的依赖和引用。举个例子：

```
import _ from 'lodash';
// 或者
const add = require('./utils/add');
```

所引用的模块可以是来自应用程序的代码，也可以是第三方库。resolver帮助webpack从每个require/import语句中，找到需要引入到bundle中的模块代码。当打包模块时，webpack使用enhanced-resolve来解析文件路径。(webpack_resolver的代码实现很有思想，webpack基于此进行treeshaking，这个概念我们后面会讲到)。

1、webpack中的模块路径解析规则

通过内置的enhanced-resolve，webpack能解析三种文件路径：

- 绝对路径

```
import '/home/me/file';
import 'c:\\users\\me\\file';
```

由于已经获得文件的绝对路径，因此不需要再做进一步解析。

- 相对路径

```
import '../utils/reqFetch';
import './styles.css';
```

这种情况下，使用 import 或 require 的资源文件所处的目录，被认为是上下文目录。在 import/require 中给定的相对路径，enhanced-resolve会拼接此上下文路径，来生成模块的绝对路径(path.resolve(_dirname, RelativePath))。这也是我们在写代码时最常用的方式之一，另一种最常用的方式则是模块路径。

- 模块路径

```
import 'module';
import 'module/lib/file';
```

也就是在resolve.modules中指定的所有目录检索模块(node_modules里的模块已经被默认配置了)。你可以通过配置别名的方式来替换初始模块路径，具体请参照下面 resolve.alias 配置选项。

2、resolve

- alias

上文中提到我们可以通过 resolve.alias 来自定义配置模块路径。现在我们来是实现一下：首先，我们src目录下新建一个utils文件夹，并新建一个add.js文件，对外暴露出一个add函数。

```
// src/utils/add.js
export default function add(a, b){
  return a + b;
}
```

然后我们在src/index.js中基于相对路径引用并使用它：

```
import add from './utils/add';

console.log(add);
```

很好，代码跑起来了并且没有报错。这时我们期望能用@utils/add的方式去引用它，于是我们这样写了：

```
import add from '@utils/add';
console.log(add(a,b));
```

很明显它会报错，因为webpack会将其当做一个模块路径来识别——所以无法找到@utils这个模块。这时，我们配置下resolve：

```
// webpack.config.js
const path = require('path');
module.exports = {
  //...
  resolve: {
    alias: {
      "@utils": path.resolve(__dirname, 'src/utils/')
    },
  },
};
```

如代码所示，我们讲utils文件夹的绝对路径配置为一个模块路径，起一个别名为“@utils”。重启服务发现，代码跑起来了。模块识别成功了。

- **extentions**

上述代码中我们发现，只需要“import add from '@utils/add'”，webpack就可以帮我们找到add.js。事实上，这与import add from '@utils/add.js' 的效果是一致的。为什么会这样？原来webpack的内置解析器已经默认定义好了一些文件/目录的路径解析规则。比如当我们

```
import utils from './utils';
```

utils是一个文件目录而不是模块(文件)，但webpack在这种情况下默认帮我们添加了后缀"/index.js"，从而将此相对路径指向到utils里的index.js。这是webpack解析器默认内置好的规则。那么现在有一个问题：当utils文件夹下同时拥有add.js add.json时，“@utils/add”会指向谁呢？@utils/add.json

```
{
  "name": "add"
}
```

我们发现仍然指向到add.js。当我们删掉add.js，会发现此时的引入的add变成了一个json对象。上述现象似乎表明了这是一个默认配置的优先级的问题。而webpack对外暴露了配置属性: `resolve.extensions`，它的用法形如：

```
module.exports = {
  //...
  resolve: {
    extensions: ['.js', '.json', '.wasm'],
  },
};
```

webpack会按照数组顺序去解析这些后缀名，对于同名的文件，webpack总是会先解析列在数组首位的后缀名的文件。

2.2.3 外部扩展(Externals)

有时候我们为了减小bundle的体积，从而把一些不变的第三方库用cdn的形式引入进来，比如jQuery: index.html

```
<script
  src="https://cdn.bootcdn.net/ajax/libs/jquery/3.6.0/jquery.js"
></script>
```

这个时候我们想在我们的代码里使用引入的jquery——但似乎三种模块引入方式都不行，这时候怎么办呢？webpack给我们提供了Externals的配置属性，让我们可以配置外部扩展模块：

```
module.exports = {
  //...
  externals: {
    jquery: 'jQuery',
  },
};
```

我们尝试在代码中使用jQuery：

```
// index.js
import $ from 'jquery';
console.log($);
```

发现打印成功，这说明我们已经在代码中使用它。注意：我们如何得知 { jquery: 'jQuery' } 中的 'jQuery'？其实就是cdn里打入到window中的变量名，比如jQuery不仅有jQuery变量名，还有\$，那么我们也可以写成这样子：

```
module.exports = {
  //...
  externals: {
    jquery: '$',
  },
};
```

重启服务后发现，效果是一样的。

2.2.4 依赖图(dependency graph)

每当一个文件依赖另一个文件时，webpack 会直接将文件视为存在依赖关系。这使得 webpack 可以获取非代码资源，如 images 或 web 字体等。并会把它们作为依赖提供给应用程序。当 webpack 开始工作时，它会根据我们写好的配置，从入口(entry)开始，webpack 会递归的构建一个 依赖关系图，这个依赖图包含着应用程序中所需的每个模块，然后将所有模块打包为bundle(也就是output的配置项)。

单纯讲似乎很抽象，我们更期望能够可视化打包产物的依赖图，下边列示了一些 bundle 分析工具。

bundle 分析(bundle analysis) 工具：

[官方分析工具](#) 是一个不错的开始。还有一些其他社区支持的可选项：

- [webpack-chart](#): webpack stats 可交互饼图。
- [webpack-visualizer](#): 可视化并分析你的 bundle，检查哪些模块占用空间，哪些可能是重复使用的。
- [webpack-bundle-analyzer](#): 一个 plugin 和 CLI 工具，它将 bundle 内容展示为一个便捷的、交互式、可缩放的树状图形式。
- [webpack bundle optimize helper](#): 这个工具会分析你的 bundle，并提供可操作的改进措施，以减少 bundle 的大小。
- [bundle-stats](#): 生成一个 bundle 报告 (bundle 大小、资源、模块)，并比较不同构建之间的结果。

我们来使用 `webpack-bundle-analyzer` 实现。

```
# 首先安装这个插件作为开发依赖
# NPM
npm install --save-dev webpack-bundle-analyzer
# Yarn
yarn add -D webpack-bundle-analyzer
```

然后我们配置它：

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

module.exports = {
  plugins: [
    // ...others
    new BundleAnalyzerPlugin()
  ]
}
```

这时我们执行打包命令，发现控制台里打印出下面这样的日志：

```
webpack Bundle Analyzer is started at http://127.0.0.1:8888
use Ctrl+C to close it
asset bundle.js 5.46 KiB [emitted] [minimized] (name: main) 1
related asset
asset index.html 352 bytes [emitted]
orphan modules 2.35 KiB [orphan] 3 modules
...
```

我们在浏览器中打开<http://127.0.0.1:8888>，我们成功可视化了打包产物依赖图！

注意：对于 HTTP/1.1 的应用程序来说，由 webpack 构建的 bundle 非常强大。当浏览器发起请求时，它能最大程度的减少应用的等待时间。而对于 HTTP/2 来说，我们还可以使用代码分割进行进一步优化。(开发环境观测的话需要在DevServer里进行配置{http2:true, https:false})。这个我们会在之后的课程里讲。

2.3 扩展功能

2.3.1 PostCSS 与 CSS模块

[PostCSS](#) 是一个用 JavaScript 工具和插件转换 CSS 代码的工具。比如可以使用 [Autoprefixer](#) 插件自动获取浏览器的流行度和能够支持的属性，并根据这些数据帮我们自动的为 CSS 规则添加前缀，将最新的 CSS 语法转换成大多数浏览器都能理解的语法。

[CSS 模块](#) 能让你永远不用担心命名太大众化而造成冲突，只要用最有意义的名字就行了。

- [PostCSS](#)

[PostCSS](#) 与 [Webpack](#) 结合，需要安装 `style-loader`, `css-loader`, `postcss-loader` 三个 loader:

```
module.exports = {
```

```
module: {
  rules: [
    {
      test: /\.css$/,
      exclude: /node_modules/,
      use: [
        {
          loader: 'style-loader',
        },
        {
          loader: 'css-loader',
          options: {
            importLoaders: 1,
          }
        },
        {
          loader: 'postcss-loader'
        }
      ]
    }
  ]
}
```

然后在项目根目录下创建 `postcss.config.js`:

```
module.exports = {
  plugins: [
    require('autoprefixer'),
    require('postcss-nested')
  ]
}
```

插件 `autoprefixer` 提供自动给样式加前缀去兼容浏览器，`postcss-nested` 提供编写嵌套的样式语法。

最后，在 `package.json` 内增加如下实例内容：

```
"browserslist": [
  "> 1%",
  "last 2 versions"
]
```

数组内的值对应的含义：

1. `last 2 versions`: 每个浏览器中最新的两个版本。
2. `> 1%` or `>= 1%`: 全球浏览器使用率大于1%或大于等于1%。

- CSS 模块

目前还有一个问题，就是多人编写的样式可能会冲突，开启 CSS 模块可以解决这个问题。`webpack.config.js` 配置：

```
module: {  
  rules: [  
    {  
      test: /\.css$/,  
      use: [  
        'style-loader',  
        {  
          loader: 'css-loader',  
          options: {  
            // 开启css模块  
            modules: true  
          }  
        },  
        'postcss-loader'  
      ]  
    }  
  ]  
}
```

加入项目里有样式文件 `style.css`:

```
body {  
  display: flex;  
  flex-direction: column;  
  .box {  
    width: 100px;  
    height: 100px;  
    background: red;  
  }  
}
```

在 js 文件里导入 css 文件:

```
// 开启 css 模块后，可以导入模块
import style from './style.css'

const div = document.createElement('div')
div.textContent = 'hello webpack'

// style 里可以识别 class 样式
div.classList.add(style.box)
document.body.appendChild(div)
```

也可以部分开启 CSS 模块模式，比如全局样式可以冠以 `.global` 前缀，如：

1. `*.global.css` 普通模式
2. `*.css` CSS module 模式

这里统一用 `global` 关键词进行识别。用正则表达式匹配文件：

```
// CSS module
{
  test: new RegExp(`^(!.*\\.global).*\\.css`),
  use: [
    {
      loader: 'style-loader'
    },
    {
      loader: 'css-loader',
      options: {
        modules: true,
        localIdentName: '[hash:base64:6]'
      }
    },
    {
      loader: 'postcss-loader'
    }
  ],
  exclude: [path.resolve(__dirname, '..', 'node_modules')]
}

// 普通模式
{
  test: new RegExp(`^(.*)\\.global.*\\.css`),
  use: [
    {
      loader: 'style-loader'
    },
    {
```

```
    loader: 'css-loader',
  },
{
  loader: 'postcss-loader'
}
],
exclude:[path.resolve(__dirname, '.', 'node_modules')]
}
```

2.3.2 Web Works

有时我们需要在客户端进行大量的运算，但又不想让它阻塞我们的js主线程。你可能第一时间考虑到的是异步。

但事实上，运算量过大(执行时间过长)的异步也会阻塞js事件循环，甚至会导致浏览器假死状态。

这时候，HTML5的新特性 WebWorker就派上了用场。

在此之前，我们简单的了解下这个特性。

html5之前，打开一个常规的网页，浏览器会启用几个线程？

一般而言，至少存在三个线程(公用线程不计入在内):

分别是js引擎线程(处理js)、GUI渲染线程(渲染页面)、浏览器事件触发线程(控制交互)。

当一段JS脚本长时间占用着处理器，就会挂起浏览器的GUI更新，而后面的事件响应也被排在队列中得不到处理，从而造成了浏览器被锁定进入假死状态。

现在如果遇到了这种情况，我们可以做的不仅仅是优化代码———html5提供了解决方案，webworker。

webWorkers提供了js的后台处理线程的API，它允许将复杂耗时的单纯js逻辑处理放在浏览器后台线程中进行处理，让js线程不阻塞UI线程的渲染。

多个线程间也是可以通过相同的方法进行数据传递。

它的使用方式如下：

```
//new Worker(scriptURL: string | URL, options?: WorkerOptions)
new Worker("someworker.js");
```

也就是说，需要单独写一个js脚本，然后使用new Worker来创建一个Work线程实例。

这意味着并不是将这个脚本当做一个模块引入进来，而是单独开一个线程去执行这个脚本。

我们知道，常规模式下，我们的webpack工程化环境只会打包出一个bundle.js，那我们的worker脚本怎么办？

也许你会想到设置多入口(Entry)多出口(output)的方式。

事实上不需要那么麻烦，webpack4的时候就提供了worker-loader专门配置webWorker。

令人开心的是，webpack5之后就不需要用loader啦，因为webpack5内置了这个功能。

我们来试验一下：

- 第一步

创建一个work脚本 work.js, 我们甚至不需要写任何内容，我们的重点不是 webWorker的使用，而是在webpack环境中使用这个特性。

当然，也可以写点什么，比如：

```
self.onmessage = ({ data: { question } }) => {
  self.postMessage({
    answer: 42,
  })
}
```

- 在 index.js 中使用它

```
// 下面的代码属于业务逻辑
const worker = new Worker(new URL('./work.js', import.meta.url));
worker.postMessage({
  question:
    'hi, 那边的workder线程, 请告诉我今天的幸运数字是多少? ',
});
worker.onmessage = ({ data: { answer } }) => {
  console.log(answer);
};
```

(import.meta.url这个参数能够锁定我们当前的这个模块——注意，它不能在 commonjs中使用。)

这时候我们执行打包命令，会发现dist目录下除了bundle.js之外，还有另外一个 xxx.bundle.js!

这说明我们的webpack5自动的将被new Work使用的脚本单独打出了一个bundle。

我们加上刚才的问答代码，执行npm run dev，发现它是能够正常工作。
并且在network里也可以发现多了一个src_worker_js.bundle.js。

总结：

webpack5以来内置了很多功能，让我们不需要过多的配置，比如之前讲过的hot模式，和现在的web worker。

2.3.3 TypeScript

在前端生态里，TS扮演着越来越重要的角色。

我们直入正题，讲下如何在webpack工程化环境中集成TS。

首先，当然是安装我们的ts和对应的loader。

```
npm install --save-dev typescript ts-loader
```

接下来我们需要在项目根目录下添加一个ts的配置文件——tsconfig.json，我们可以用ts自带的工具来自动化生成它。

```
npx tsc --init
```

我们发现生成了一个tsconfig.json，里面注释掉了绝大多数配置。

现在，根据我们想要的效果来打开对应的配置。

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "noImplicitAny": true,
    "sourceMap": true,
    "module": "es6",
    "target": "es5",
    "jsx": "react",
    "allowJs": true,
    "moduleResolution": "node"
  }
}
```

好了，接下来我们新增一个src/index.ts，内置一些内容。

然后我们别忘了更改我们的entry及配置对应的loader。

当然，还有resolve.extensions，将.ts放在.js之前，这样它会先找.ts。

注意，如果我们使用了sourceMap，一定记得和上面的ts配置一样，设置sourcemap为true。

也别忘记在我们的webpack.config.js里，添加sourcemap,就像我们之前课程里讲的那样。

更改如下：

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

module.exports = {
  entry: './src/index.ts',
  devtool: 'inline-source-map',
```

```
module: {
  rules: [
    {
      test: /\.ts|tsx$/,
      use: 'ts-loader',
      exclude: /node_modules/,
    },
  ],
},
resolve: {
  extensions: [ '.tsx', '.ts', '.js' ],
},
output: {
  filename: 'bundle.js',
  path: path.resolve(__dirname, 'dist'),
},
};
```

运行我们的项目，我们发现完全没有问题呢！

- 使用第三方类库

在从 npm 上安装第三方库时，一定要记得同时安装这个库的类型声明文件(typing definition)。

我们可以从 TypeSearch中找到并安装这些第三方库的类型声明文件(<https://www.typescriptlang.org/dt/search?search=>)。

举个例子，如果想安装 lodash 类型声明文件，我们可以运行下面的命令：

```
npm install --save-dev @types/lodash
```

- eslint & ts

注意，如果要使用eslint，使用初始化命令的时候，记得选择“使用了typescript”。

```
npx eslint --init
# 往下选择的时候选择使用了typescript
```

如果已经配置了eslint，但没有配置ts相关的配置，那么我们需要先安装对应的 plugin

```
yarn add -D @typescript-eslint/eslint-plugin@latest
@typescript-eslint/parser@latest
```

注意如果需要用到react的话，记得也要安装

```
yarn add -D eslint-plugin-react@latest
```

vue或者其他常用框架同样如此，一般都会有专门的plugin。

然后我们队.esilntrc进行更改~

```
{
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": [
    "eslint:recommended", // 如果需要react的话
    "plugin:react/recommended",
    "plugin:@typescript-eslint/recommended"
  ],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaFeatures": {
      "jsx": true
    }, // 如果需要react的话
    "ecmaVersion": 13,
    "sourceType": "module"
  },
  "plugins": [
    "react",
    "@typescript-eslint"
  ],
  "rules": {
    // ...一些自定义的rules
    "no-console": "error"
  }
};
```

执行npm run eslint试一下！

大功告成！

2.4 多页面应用

2.4.1 entry 配置

- 单个入口（简写）语法

用法：entry: string | [string]

webpack.config.js

```
module.exports = {  
  entry: './path/to/my/entry/file.js',  
};
```

`entry` 属性的单个入口语法，参考下面的简写：

webpack.config.js

```
module.exports = {  
  entry: {  
    main: './path/to/my/entry/file.js',  
  },  
};
```

我们也可以将一个文件路径数组传递给 `entry` 属性，这将创建一个所谓的 "**multi-main entry**"。在你想要一次注入多个依赖文件，并且将它们的依赖关系绘制在一个 "chunk" 中时，这种方式就很有用。

webpack.config.js

```
module.exports = {  
  entry: ['./src/file_1.js', './src/file_2.js'],  
  output: {  
    filename: 'bundle.js',  
  },  
};
```

当你希望通过一个入口（例如一个库）为应用程序或工具快速设置 webpack 配置时，单一入口的语法方式是不错的选择。然而，使用这种语法方式来扩展或调整配置的灵活性不大。

• 对象语法

用法： `entry: { <entryChunkName> string | [string] } | {}`

webpack.config.js

```
module.exports = {  
  entry: {  
    app: './src/app.js',  
    adminApp: './src/adminApp.js',  
  },  
};
```

对象语法会比较繁琐。然而，这是应用程序中定义入口的最可扩展的方式。

描述入口的对象：

用于描述入口的对象。你可以使用如下属性：

- `dependon`: 当前入口所依赖的入口。它们必须在该入口被加载前被加载。
- `filename`: 指定要输出的文件名称。
- `import`: 启动时需加载的模块。
- `library`: 指定 `library` 选项，为当前 `entry` 构建一个 `library`。
- `runtime`: 运行时 `chunk` 的名字。如果设置了，就会创建一个新的运行时 `chunk`。在 webpack 5.43.0 之后可将其设为 `false` 以避免一个新的运行时 `chunk`。
- `publicPath`: 当该入口的输出文件在浏览器中被引用时，为它们指定一个公共 URL 地址。请查看 [output.publicPath](#)。

webpack.config.js

```
module.exports = {
  entry: {
    a2: 'dependingfile.js',
    b2: {
      dependon: 'a2',
      import: './src/app.js',
    },
  },
};
```

`runtime` 和 `dependon` 不应在同一个入口上同时使用，所以如下配置无效，并且会抛出错误：

webpack.config.js

```
module.exports = {
  entry: {
    a2: './a',
    b2: {
      runtime: 'x2',
      dependon: 'a2',
      import: './b',
    },
  },
};
```

确保 `runtime` 不能指向已存在的入口名称，例如下面配置会抛出一个错误：

```
module.exports = {
  entry: {
    a1: './a',
    b1: {
      runtime: 'a1',
      import: './b',
    },
  },
};
```

另外 `dependon` 不能是循环引用的，下面的例子也会出现错误：

```
module.exports = {
  entry: {
    a3: {
      import: './a',
      dependon: 'b3',
    },
    b3: {
      import: './b',
      dependon: 'a3',
    },
  },
};
```

2.4.2 配置 index.html 模板

- 生成多个HTML文件

要生成多个HTML文件，请在插件数组中多次声明插件。

webpack.config.js

```
{
  entry: 'index.js',
  output: {
    path: __dirname + '/dist',
    filename: 'index_bundle.js'
  },
  plugins: [
    new HtmlWebpackPlugin(), // Generates default index.html
    new HtmlWebpackPlugin({ // Also generate a test.html
      filename: 'test.html',
      template: 'src/assets/test.html'
    })
}
```

```
    ]  
}
```

- 编写自己的模板

如果默认生成的HTML不能满足您的需要，您可以提供自己的模板。最简单的方法是使用 `template` 选项并传递自定义HTML文件。html 网页包插件将自动将所有必要的 CSS、JS、manifest 和 favicon 文件注入标记中。

```
plugins: [  
  new HtmlWebpackPlugin({  
    title: 'Custom template',  
    // Load a custom template (lodash by default)  
    template: 'index.html'  
  })  
]
```

index.html

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="utf-8"/>  
    <title><%= htmlWebpackPlugin.options.title %></title>  
  </head>  
  <body>  
  </body>  
</html>
```

2.4.3 多页面应用

webpack.config.js

```
module.exports = {  
  entry: {  
    pageOne: './src/pageOne/index.js',  
    pageTwo: './src/pageTwo/index.js',  
    pageThree: './src/pageThree/index.js',  
  },  
};
```

这是什么？ 我们告诉 webpack 需要三个独立分离的依赖图（如上面的示例）。

为什么? 在多页面应用程序中, server 会拉取一个新的 HTML 文档给你的客户端。页面重新加载此新文档, 并且资源被重新下载。然而, 这给了我们特殊的机会去做很多事, 例如使用 [optimization.splitChunks](#) 为页面间共享的应用程序代码创建 bundle。由于入口起点数量的增多, 多页应用能够复用多个入口起点之间的大量代码/模块, 从而可以极大地从这些技术中受益。

2.5 Tree shaking

tree shaking 是一个术语, 通常用于描述移除 JavaScript 上下文中的未引用代码 (dead-code)。它依赖于 ES2015 模块语法的 [静态结构](#) 特性, 例如 [import](#) 和 [export](#)。这个术语和概念实际上是由 ES2015 模块打包工具 [rollup](#) 普及起来的。

webpack 2 正式版本内置支持 ES2015 模块 (也叫做 *harmony modules*) 和未使用模块检测能力。新的 webpack 4 正式版本扩展了此检测能力, 通过 `package.json` 的 `"sideEffects"` 属性作为标记, 向 compiler 提供提示, 表明项目中的哪些文件是 "pure(纯正 ES2015 模块)", 由此可以安全地删除文件中未使用的部分。

2.5.1 tree-shaking实验

- `src/math.js`

```
export function square(x) {
  return x * x;
}

export function cube(x) {
  return x * x * x;
}
```

需要将 `mode` 配置设置成[development](#), 以确定 bundle 不会被压缩:

- `webpack.config.js`

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  mode: 'development',
  optimization: {
    usedExports: true,
  },
};
```

配置完这些后，更新入口脚本，使用其中一个新方法：

```
import { cube } from './math.js';

function component() {
  const element = document.createElement('pre');

  element.innerHTML = [
    'Hello webpack!',
    '5 cubed is equal to ' + cube(5)
  ].join('\n\n');

  return element;
}

document.body.appendChild(component());
```

注意，我们没有从 `src/math.js` 模块中 `import` 另外一个 `square` 方法。这个函数就是所谓的“未引用代码(dead code)”，也就是说，应该删除掉未被引用的 `export`。

现在运行 npm script `npm run build`，并查看输出的 bundle：

- `dist/bundle.js`

```
/* 1 */
/**/ (function (module, __webpack_exports__,
__webpack_require__) {
  'use strict';
  /* unused harmony export square */
  /* harmony export (immutable) */ __webpack_exports__['a'] =
cube;
  function square(x) {
    return x * x;
  }

  function cube(x) {
    return x * x * x;
  }
});
```

注意，上面的 `unused harmony export square` 注释。如果你观察它下面的代码，你会注意到虽然我们没有引用 `square`，但它仍然被包含在 bundle 中。

- mode: production

如果此时修改配置：

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  mode: 'production',
};
```

打包后发现无用的代码全部都消失了。

处于好奇，webpack是如何完美的避开没有使用的代码的呢？

很简单：就是 Webpack 没看到你使用的代码。Webpack 跟踪整个应用程序的 `import/export` 语句，因此，如果它看到导入的东西最终没有被使用，它会认为那是未引用代码(或叫做“死代码”—— dead-code)，并会对其进行 tree-shaking。

死代码并不总是那么明确的。下面是一些死代码和“活”代码的例子：

```
// 这会被看作“活”代码，不会做 tree-shaking
import { add } from './math'
```

```
console.log(add(5, 6))

// 导入并赋值给 JavaScript 对象，但在接下来的代码里没有用到
// 这就会被当做“死”代码，会被 tree-shaking
import { add, minus } from './math'
console.log(add(5, 6))

// 导入但没有赋值给 Javascript 对象，也没有在代码里用到
// 这会被当做“死”代码，会被 tree-shaking
import { add, minus } from './math'
console.log('hello webpack')

// 导入整个库，但是没有赋值给 JavaScript 对象，也没有在代码里用到
// 非常奇怪，这竟然被当做“活”代码，因为 webpack 对库的导入和本地代码导入的处理
// 方式不同。
import { add, minus } from './math'
import 'lodash'
console.log('hello webpack')
```

2.5.2 sideEffects

注意 Webpack 不能百分百安全地进行 tree-shaking。有些模块导入，只要被引入，就会对应用程序产生重要的影响。一个很好的例子就是全局样式表，或者设置全局配置的JavaScript 文件。

Webpack 认为这样的文件有“副作用”。具有副作用的文件不应该做 tree-shaking，因为这将破坏整个应用程序。

Webpack 的设计者清楚地认识到不知道哪些文件有副作用的情况下打包代码的风险，因此webpack4默认地将所有代码视为有副作用。这可以保护你免于删除必要的文件，但这意味着 Webpack 的默认行为实际上是不进行 tree-shaking。值得注意的是webpack5默认会进行 tree-shaking。

如何告诉 Webpack 你的代码无副作用，可以通过 package.json 有一个特殊的属性 sideEffects，就是为此而存在的。

它有三个可能的值：

- true

如果不指定其他值的话。这意味着所有的文件都有副作用，也就是没有一个文件可以 tree-shaking。

- false

告诉 Webpack 没有文件有副作用，所有文件都可以 tree-shaking。

- 数组[...]

是文件路径数组。它告诉 webpack，除了数组中包含的文件外，你的任何文件都没有副作用。因此，除了指定的文件之外，其他文件都可以安全地进行 tree-shaking。

webpack4 曾经不进行对 CommonJs 导出和 require() 调用时的导出使用分析。webpack 5 增加了对一些 CommonJs 构造的支持，允许消除未使用的 CommonJs 导出，并从 require() 调用中跟踪引用的导出名称。

2.6 渐进式网络应用程序 PWA

渐进式网络应用程序(progressive web application - PWA)，是一种可以提供类似于 native app(原生应用程序) 体验的 web app(网络应用程序)。PWA 可以用来做很多事情。其中最重要的是，在离线([offline](#))时应用程序能够继续运行功能。这是通过使用名为 [Service Workers](#) 的 web 技术来实现的。

2.6.1 非离线环境下运行

到目前为止，我们一直是直接查看本地文件系统的输出结果。通常情况下，真正的用户是通过网络访问 web app；用户的浏览器会与一个提供所需资源（例如，`.html`, `.js` 和 `.css` 文件）的 **server** 通讯。

我们通过搭建一个拥有更多基础特性的 server 来测试下这种离线体验。这里使用 [http-server](#) package: `npm install http-server --save-dev`。还要修改 `package.json` 的 `scripts` 部分，来添加一个 `start` script:

package.json

```
{  
  ...  
  "scripts": {  
    "start": "http-server dist"  
  },  
  ...  
}
```

注意：默认情况下，[webpack DevServer](#) 会写入到内存。我们需要启用 [devserverdevmiddleware.writeToDisk](#) 配置项，来让 http-server 处理 `./dist` 目录中的文件。

```
devServer: {  
  devMiddleware: {  
    index: true,  
    writeToDisk: true,  
  },  
},
```

如果你之前没有操作过，先得运行命令 `npm run build` 来构建你的项目。然后运行命令 `npm start`。应该产生以下输出：

```
> http-server dist  
  
Starting up http-server, serving dist  
Available on:  
  http://xx.x.x.x:8080  
  http://127.0.0.1:8080  
  http://xxx.xxx.x.x:8080  
Hit CTRL-C to stop the server
```

如果你打开浏览器访问 `http://localhost:8080` (即 `http://127.0.0.1`)，你应该会看到 webpack 应用程序被 serve 到 `dist` 目录。如果停止 server 然后刷新，则 webpack 应用程序不再可访问。

这就是我们为实现离线体验所需要的改变。在本章结束时，我们应该要实现的是，停止 server 然后刷新，仍然可以看到应用程序正常运行。

2.6.2 添加 Workbox

添加 `workbox-webpack-plugin` 插件，然后调整 `webpack.config.js` 文件：

```
npm install workbox-webpack-plugin --save-dev
```

webpack.config.js

```
const path = require('path');  
const HtmlWebpackPlugin = require('html-webpack-plugin');  
const workboxPlugin = require('workbox-webpack-plugin');  
  
module.exports = {  
  entry: {  
    app: './src/index.js',  
  },  
  plugins: [  
    new HtmlWebpackPlugin(),  
    new workboxPlugin({  
      cacheName: 'app-cash',  
      runtimeCaching: [ {  
        url: '**/*.{js,css,html}',  
        method: 'GET',  
        handler: 'CacheFirst',  
        options: {  
          cacheableResponse: {  
            statuses: [200],  
            expiration: {  
              maxAgeSeconds: 31536000,  
              minFreshSeconds: 0,  
            },  
            updateStaleWhileRevalidate: {  
              maxAgeSeconds: 31536000,  
              minFreshSeconds: 0,  
            },  
            strategy: 'cache-and-network',  
          },  
        },  
      }]  
    }],  
};
```

```
new workboxPlugin.GenerateSW({
  // 这些选项帮助快速启用 ServiceWorkers
  // 不允许遗留任何“旧的”ServiceWorkers
  clientsClaim: true,
  skipWaiting: true,
}),
],
output: {
  filename: '[name].bundle.js',
  path: path.resolve(__dirname, 'dist'),
  clean: true,
},
}
```

执行: `npx webpack`

```
[felix] 06-pwa $ npx webpack
assets by status 121 KiB [emitted]
  asset workbox-718aa5be.js 118 KiB [emitted]
  asset service-worker.js 3.23 KiB [emitted]
assets by status 1.44 KiB [compared for emit]
  asset app.bundle.js 1.21 KiB [compared for emit] (name: app)
  asset index.html 237 bytes [compared for emit]
./src/index.js 29 bytes [built] [code generated]

LOG from GenerateSW
<i> The service worker at service-worker.js will precache
<i>      2 URLs, totaling 1.47 kB.

webpack 5.61.0 compiled successfully in 1140 ms
```

现在你可以看到，生成了两个额外的文件：`service-worker.js` 和名称冗长的 `workbox-718aa5be.js`。`service-worker.js` 是 Service Worker 文件，`workbox-718aa5be.js` 是 `service-worker.js` 引用的文件，所以它也可以运行。你本地生成的文件可能会有所不同；但是应该会有一个 `service-worker.js` 文件。

所以，值得高兴的是，我们现在已经创建出一个 Service Worker。接下来该做什么？

2.6.3 注册 Service Worker

接下来我们注册 Service Worker，使其出场并开始表演。通过添加以下注册代码来完成此操作：

`index.js`

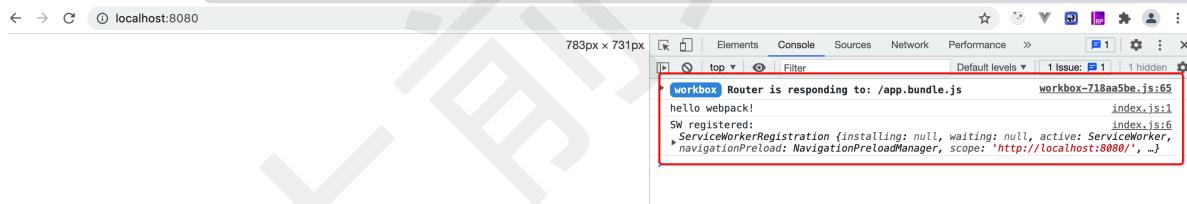
```
//...

if ('serviceworker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/service-worker.js').then(registration => {
      console.log('SW registered: ', registration);
    }).catch(registrationError => {
      console.log('SW registration failed: ', registrationError);
    });
  });
}
```

再次运行 `npx webpack` 来构建包含注册代码版本的应用程序。然后用 `npm start` 启动服务。访问 `http://localhost:8080` 并查看 `console` 控制台。在那里你应该看到：

```
SW registered
```

现在来进行测试。停止 server 并刷新页面。如果浏览器能够支持 Service Worker，应该可以看到你的应用程序还在正常运行。然而，server 已经停止 serve 整个 dist 文件夹，此刻是 Service Worker 在进行 serve。



2.7 shimming 预置依赖

`webpack` compiler 能够识别遵循 ES2015 模块语法、CommonJS 或 AMD 规范编写的模块。然而，一些 third party(第三方库) 可能会引用一些全局依赖（例如 `jQuery` 中的 `$`）。因此这些 library 也可能创建一些需要导出的全局变量。这些 "broken modules(不符合规范的模块)" 就是 *shimming(预置依赖)* 发挥作用的地方。

shim 另外一个极其有用的使用场景就是：当你希望 [polyfill](#) 扩展浏览器能力，来支持到更多用户时。在这种情况下，你可能只是想要将这些 polyfills 提供给需要修补 (patch) 的浏览器（也就是实现按需加载）。

2.7.1 Shimming 预置全局变量

让我们开始第一个 shimming 全局变量的用例。还记得我们之前用过的 `lodash` 吗？出于演示目的，例如把这个应用程序中的模块依赖，改为一个全局变量依赖。要实现这些，我们需要使用 `ProvidePlugin` 插件。

使用 `ProvidePlugin` 后，能够在 webpack 编译的每个模块中，通过访问一个变量来获取一个 package。如果 webpack 看到模块中用到这个变量，它将在最终 bundle 中引入给定的 package。让我们先移除 `lodash` 的 `import` 语句，改为通过插件提供它：

- `src/index.js`

```
console.log(_.join(['hello', 'webpack'], ' '))
```

- `webpack.config.js`

```
const webpack = require('webpack')
module.exports = {
  mode: 'development',
  entry: './src/index.js',

  plugins: [
    new webpack.ProvidePlugin({
      _: 'lodash'
    })
  ]
}
```

我们本质上所做的，就是告诉 webpack.....

如果你遇到了至少一处用到 `_` 变量的模块实例，那请你将 `lodash` package 引入进来，并将其提供给需要用到它的模块。

运行我们的构建脚本，将会看到同样的输出：

```
[felix] 01-third-party-shimming $ npx webpack
asset main.js 549 KiB [emitted] (name: main)
runtime modules 344 bytes 2 modules
cacheable modules 528 KiB
  ./src/index.js 46 bytes [built] [code generated]
  .../.../.../.../node_modules/lodash/lodash.js 528 KiB [built]
  [code generated]
webpack 5.61.0 compiled successfully in 275 ms
```

还可以使用 `ProvidePlugin` 暴露出某个模块中单个导出，通过配置一个“数组路径”（例如 `[module, child, ...children?]`）实现此功能。所以，我们假想如下，无论 `join` 方法在何处调用，我们都只会获取到 `lodash` 中提供的 `join` 方法。

- `src/index.js`

```
console.log(join(['hello', 'webpack'], ' '))
```

- `webpack.config.js`

```
const webpack = require('webpack')
module.exports = {
  mode: 'development',
  entry: './src/index.js',

  plugins: [
    new webpack.ProvidePlugin({
      // _: 'lodash'
      join: ['lodash', 'join'],
    })
  ]
}
```

这样就能很好的与 [tree shaking](#) 配合，将 `lodash` library 中的其余没有用到的导出去除。

2.7.2 细粒度 Shimming

一些遗留模块依赖的 `this` 指向的是 `window` 对象。在接下来的用例中，调整我们的 `index.js`：

```
this.alert('hello webpack')
```

当模块运行在 CommonJS 上下文中，这将会变成一个问题，也就是说此时的 `this` 指向的是 `module.exports`。在这种情况下，你可以通过使用 [imports-loader](#) 覆盖 `this` 指向：

```
const webpack = require('webpack')
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
  mode: 'production',
  entry: './src/index.js',

  module: {
```

```
rules: [
  {
    test: require.resolve('./src/index.js'),
    use: 'imports-loader?wrapper>window',
  },
],
},
plugins: [
  new webpack.ProvidePlugin({
    _: 'lodash'
  }),
  new HtmlWebpackPlugin()
]
}
```

2.7.3 全局 Exports

让我们假设，某个 library 创建出一个全局变量，它期望 consumer(使用者) 使用这个变量。为此，我们可以在项目配置中，添加一个小模块来演示说明：

- `src/globals.js`

```
const file = 'example.txt';
const helpers = {
  test: function () {
    console.log('test something')
  },
  parse: function () {
    console.log('parse something')
  },
}
```

- `webpack.config.js`

```
const webpack = require('webpack')
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
  mode: 'production',
  entry: './src/index.js',

  module: {
    rules: [
      {
        test: require.resolve('./src/index.js'),
```

```
use: 'imports-loader?wrapper>window',
} ,


{
  test: require.resolve('./src/globals.js'),
  use: 'exports-loader?
type=commonjs&exports=file,multiple|helpers.parse|parse',
  },
]
},


plugins: [
  new webpack.ProvidePlugin({
    _: 'lodash'
  }),
  new HtmlWebpackPlugin()
]
}
```

此时，在我们的 entry 入口文件中（即 `src/index.js`），可以使用 `const { file, parse } = require('./globals.js');`，可以保证一切将顺利运行。

2.7.4 加载 Polyfills

目前为止，我们讨论的所有内容都是处理那些遗留的 package，让我们进入到第二个话题：**polyfill**。

有很多方法来加载 polyfill。例如，想要引入 [@babel/polyfill](#) 我们只需如下操作：

```
npm install --save @babel/polyfill
```

然后，使用 `import` 将其引入到我们的主 bundle 文件：

```
import '@babel/polyfill'
console.log(Array.from([1, 2, 3], x => x + x))
```

注意，这种方式优先考虑正确性，而不考虑 bundle 体积大小。为了安全和可靠，polyfill/shim 必须运行于所有其他代码之前，而且需要同步加载，或者说，需要在所有 polyfill/shim 加载之后，再去加载所有应用程序代码。社区中存在许多误解，即现代浏览器“不需要”polyfill，或者 polyfill/shim 仅用于添加缺失功能 - 实际上，它们通常用于修复损坏实现(**repair broken implementation**)，即使是在最现代的浏览器中，也会出现这种情况。因此，最佳实践仍然是，不加选择地和同步地加载所有 polyfill/shim，尽管这会导致额外的 bundle 体积成本。

2.7.5 进一步优化 Polyfills

不建议使用 `import '@babel/polyfill'`。因为这样做的缺点是会全局引入整个 polyfill 包，比如 `Array.from` 会全局引入，不但包的体积大，而且还会污染全局环境。

`babel-preset-env` package 通过 [browserslist](#) 来转译那些你浏览器中不支持的特性。这个 preset 使用 `useBuiltIns` 选项，默认值是 `false`，这种方式可以将全局 `babel-polyfill` 导入，改进为更细粒度的 `import` 格式：

```
import 'core-js/modules/es7.string.pad-start';
import 'core-js/modules/es7.string.pad-end';
import 'core-js/modules/web.timers';
import 'core-js/modules/web.immediate';
import 'core-js/modules/web.dom.iterable';
```

- 安装 `@babel/preset-env` 及相关的包

```
npm i babel-loader @babel/core @babel/preset-env -D
```

- `webpack.config.js`

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
  mode: 'production',
  entry: './src/index.js',
  plugins: [
    new HtmlWebpackPlugin()
  ],
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: [
              [
                '@babel/preset-env',
                {
                  targets: [
                    "last 1 version",
                    "> 1%"
                  ]
                }
              ]
            ]
          }
        }
      }
    ]
  }
}
```

```
        ],
        useBuiltIns: 'usage'
    }
]
}
}
]
}
}
}
```

useBuiltIns: 参数有 “entry”、“usage”、false 三个值

默认值是 `false`，此参数决定了babel打包时如何处理`@babel/polyfill`语句。

“entry”: 会将文件中 `import @babel/polyfill` 语句 结合 targets，转换为一系列引入语句，去掉目标浏览器已支持的 polyfill 模块，不管代码里有没有用到，只要目标浏览器不支持都会引入对应的 polyfill 模块。

“usage”: 不需要手动在代码里写 `import @babel/polyfill`，打包时会自动根据实际代码的使用情况，结合 targets 引入代码里实际用到部分 polyfill 模块

false: 对 `import '@babel/polyfill'` 不作任何处理，也不会自动引入 polyfill 模块。

需要注意的是在 webpack 打包文件配置的 entry 中引入的 `@babel/polyfill` 不会根据 `useBuiltIns` 配置任何转换处理。

由于`@babel/polyfill`在7.4.0中被弃用，我们建议直接添加corejs并通过corejs选项设置版本。

- 执行编译 `npx webpack`

```
[felix] 02-polyfill $ npx webpack
```

```
WARNING (@babel/preset-env): We noticed you're using the
`useBuiltIns` option without declaring a core-js version.
Currently, we assume version 2.x when no version is passed. Since
this default version will likely change in future versions of
Babel, we recommend explicitly setting the core-js version you
are using via the `corejs` option.
```

```
You should also be sure that the version you pass to the `corejs`
option matches the version specified in your `package.json`'s
`dependencies` section. If it doesn't, you need to run one of the
following commands:
```

```
npm install --save core-js@2          npm install --save core-js@3
yarn add core-js@2                   yarn add core-js@3
```

```
More info about useBuiltIns: https://babeljs.io/docs/en/babel-preset-env#usebuiltins
More info about core-js: https://babeljs.io/docs/en/babel-preset-env#corejs
asset main.js 16.7 KiB [emitted] [minimized] (name: main)
asset index.html 214 bytes [compared for emit]
runtime modules 663 bytes 3 modules
modules by path ./node_modules/core-js/modules/*.js 38.9 KiB 68
modules
./src/index.js 374 bytes [built] [code generated]
webpack 5.61.0 compiled successfully in 1613 ms
```

提示我们需要安装 core-js。

```
npm i core-js@3 -s
```

此时还需要添加一个配置：

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
  mode: 'production',
  entry: './src/index.js',
  plugins: [
    new HtmlWebpackPlugin()
  ],
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: [
              [
                '@babel/preset-env',
                {
                  targets: [
                    "last 1 version",
                    "> 1%",
                  ],
                  useBuiltIns: 'usage',
                }
              ]
            ]
          }
        }
      }
    ]
  }
}
```

成功优化!

2.8 创建 library

除了打包应用程序，webpack 还可以用于打包 JavaScript library。

2.8.1 创建一个 library

假设我们正在编写一个名为 `webpack-numbers` 的小的 library，可以将数字 1 到 5 转换为文本表示，反之亦然，例如将 2 转换为 'two'。

使用 npm 初始化项目，然后安装 webpack，webpack-cli 和 lodash：

```
npm i webpack webpack-cli lodash -D
```

我们将 `lodash` 安装为 `devDependencies` 而不是 `dependencies`，因为我们不需要将其打包到我们的库中，否则我们的库体积很容易变大。

src/ref.json

```
[  
  {  
    "num": 1,  
    "word": "One"  
  },  
  {  
    "num": 2,  
    "word": "Two"  
  },  
  {  
    "num": 3,  
    "word": "Three"  
  },  
]
```

```
{  
  "num": 4,  
  "word": "Four"  
,  
{  
  "num": 5,  
  "word": "Five"  
,  
{  
  "num": 0,  
  "word": "zero"  
}  
]
```

src/index.js

```
import _ from 'lodash';  
import numRef from './ref.json';  
  
export function numToword(num) {  
  return _.reduce(  
    numRef,  
    (accum, ref) => {  
      return ref.num === num ? ref.word : accum;  
    },  
    ''  
  );  
}  
  
export function wordToNum(word) {  
  return _.reduce(  
    numRef,  
    (accum, ref) => {  
      return ref.word === word && word.toLowerCase() ? ref.num :  
        accum;  
    },  
    -1  
  );  
}
```

2.8.2 Webpack 配置

我们可以从如下 webpack 基本配置开始：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js',
  },
};
```

在上面的例子中，我们将通知 webpack 将 `src/index.js` 打包到 `dist/webpack-numbers.js` 中。

2.8.3 导出 Library

到目前为止，一切都应该与打包应用程序一样，这里是不同的部分 - 我们需要通过 `output.library` 配置项暴露从入口导出的内容。

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js',
    library: "webpackNumbers",
  },
};
```

我们暴露了 `webpackNumbers`，以便用户可以通过 `script` 标签使用。

```
<script src="https://example.org/webpack-numbers.js"></script>
<script>
  window.webpackNumbers.wordToNum('Five');
</script>
```

然而它只能通过被 script 标签引用而发挥作用，它不能运行在 CommonJS、AMD、Node.js 等环境中。

作为一个库作者，我们希望它能够兼容不同的环境，也就是说，用户应该能够通过以下方式使用打包后的库：

- **CommonJS module require:**

```
const webpackNumbers = require('webpack-numbers');
// ...
webpackNumbers.wordToNum('Two');
```

- **AMD module require:**

```
require(['webpackNumbers'], function (webpackNumbers) {
    // ...
    webpackNumbers.wordToNum('Two');
});
```

- **script tag:**

```
<!DOCTYPE html>
<html>
    ...
    <script src="https://example.org/webpack-numbers.js">
</script>
    <script>
        // ...
        // Global variable
        webpackNumbers.wordToNum('Five');
        // Property in the window object
        window.webpackNumbers.wordToNum('Five');
        // ...
    </script>
</html>
```

我们更新 `output.library` 配置项，将其 `type` 设置为 `'umd'`：

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js',
    library: {
      name: 'webpackNumbers',
      type: 'umd',
    },
  },
};
```

现在 webpack 将打包一个库，其可以与 CommonJS、AMD 以及 script 标签使用。

2.8.4 外部化 lodash

现在，如果执行 `webpack`，你会发现创建了一个体积相当大的文件。如果你查看这个文件，会看到 lodash 也被打包到代码中。在这种场景中，我们更倾向于把 `lodash` 当作 `peerDependency`。也就是说，consumer(使用者) 应该已经安装过 `lodash`。因此，你就可以放弃控制此外部 library，而是将控制权让给使用 library 的 consumer。

这可以使用 `externals` 配置来完成：

webpack.config.js

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js',
    library: {
      name: "webpackNumbers",
      type: "umd"
    },
    // 修补bug
    globalObject: 'globalThis',
  },
  externals: {
    lodash: {
      commonjs: 'lodash',
    }
  }
};
```

```
commonjs2: 'lodash',
amd: 'lodash',
root: '_',
},
},
};
```

这意味着你的 library 需要一个名为 `lodash` 的依赖，这个依赖在 consumer 环境中必须存在且可用。

2.8.5 外部化限制

对于想要实现从一个依赖中调用多个文件的那些 library：

```
import A from 'library/one';
import B from 'library/two';

// ...
```

无法通过在 `externals` 中指定整个 `library` 的方式，将它们从 bundle 中排除。而是需要逐个或者使用一个正则表达式，来排除它们。

```
module.exports = {
//...
externals: [
  'library/one',
  'library/two',
  // 匹配以 "library/" 开始的所有依赖
  /^library\/.+$/,
],
};
```

2.8.6 优化输出

为优化生产环境下的输出结果，我们还需要将生成 bundle 的文件路径，添加到 `package.json` 中的 `main` 字段中。

package.json

```
{
  ...
  "main": "dist/webpack-numbers.js",
  ...
}
```

或者，按照这个 [指南](#)，将其添加为标准模块：

```
{  
  ...  
  "module": "src/index.js",  
  ...  
}
```

这里的 key(键) `main` 是参照 [package.json 标准](#)，而 `module` 是参照 [一个提案](#)，此提案允许 JavaScript 生态系统升级使用 ES2015 模块，而不会破坏向后兼容性。

2.8.7 发布为 npm package

现在，你可以 [将其发布为一个 npm package](#)，并且在 [unpkg.com](#) 找到它，并分发给你的用户。

2.9 模块联邦

2.9.1 什么是模块联邦

多个独立的构建可以组成一个应用程序，这些独立的构建之间不应该存在依赖关系，因此可以单独开发和部署它们。

这通常被称作微前端，但并不仅限于此。

Webpack5 模块联邦让 Webpack 达到了线上 Runtime 的效果，让代码直接在项目间利用 CDN 直接共享，不再需要本地安装 Npm 包、构建再发布了！

我们知道 Webpack 可以通过 DLL 或者 Externals 做代码共享时 Common Chunk，但不同应用和项目间这个任务就变得困难了，我们几乎无法在项目之间做到按需热插拔。

- **NPM 方式共享模块**

想象一下正常的共享模块方式，对，就是 NPM。

如下图所示，正常的代码共享需要将依赖作为 Lib 安装到项目，进行 Webpack 打包构建再上线，如下图：



对于项目 Home 与 Search，需要共享一个模块时，最常见的办法就是将其抽成通用依赖并分别安装在各自项目中。

虽然 Monorepo 可以一定程度解决重复安装和修改困难的问题，但依然需要走本地编译。

• UMD 方式共享模块

真正 Runtime 的方式可能是 UMD 方式共享代码模块，即将模块用 Webpack UMD 模式打包，并输出到其他项目中。这是非常普遍的模块共享方式：



对于项目 Home 与 Search，直接利用 UMD 包复用一个模块。但这种技术方案问题也很明显，就是包体积无法达到本地编译时的优化效果，且库之间容易冲突。

• 微前端方式共享模块

微前端：micro-frontends (MFE) 也是最近比较火的模块共享管理方式，微前端就是要解决多项目并存问题，多项目并存的最大问题是模块共享，不能有冲突。



由于微前端还要考虑样式冲突、生命周期管理，所以本文只聚焦在资源加载方式上。微前端一般有两种打包方式：

1. 子应用独立打包，模块更解耦，但无法抽取公共依赖等。
2. 整体应用一起打包，很好解决上面的问题，但打包速度实在是太慢了，不具备水平扩展能力。

• 模块联邦方式

终于提到本文的主角了，作为 Webpack5 内置核心特性之一的 Federated Module：



从图中可以看到，这个方案是直接将一个应用的包应用于另一个应用，同时具备整体应用一起打包的公共依赖抽取能力。

2.9.2 应用案例

本案例模拟三个应用：`Nav`、`Search` 及 `Home`。每个应用都是独立的，又通过模块联邦联系到了一起。

1、Nav 导航

`src/header.js`

```
const Header = () => {
  const header = document.createElement('h1')
  header.textContent = '公共头部内容'
  return header
}

export default Header
```

src/index.js

```
import Header from './Header'
const div = document.createElement('div')
div.appendChild(Header())
document.body.appendChild(div)
```

webpack.config.js

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

const {
  ModuleFederationPlugin
} = require('webpack').container

module.exports = {
  mode: 'production',
  entry: './src/index.js',
  plugins: [
    new HtmlWebpackPlugin(),
    new ModuleFederationPlugin({
      // 模块联邦名字
      name: 'nav',
      // 外部访问的资源名字
      filename: 'remoteEntry.js',
      // 引用的外部资源列表
      remotes: {},
      // 暴露给外部资源列表
      exposes: {
        './Header': './src/Header.js',
      },
      // 共享模块，如lodash
      shared: {},
    }),
  ]
}
```

应用 webpack 运行服务：

```
[felix] nav $ npx webpack serve --port 3003
```

2、Home 首页

src/HomeList

```
const HomeList = (num) => {
  let str = '<ul>'
  for (let i = 0; i < num; i++) {
    str += '<li>item ' + i + '</li>'
  }
  str += '</ul>'
  return str
}

export default HomeList
```

src/index.js

```
import HomeList from './HomeList'
import('nav/Header').then((Header) => {
  const body = document.createElement('div')
  body.appendChild(Header.default())
  document.body.appendChild(body)
  document.body.innerHTML += HomeList(5)
})
```

webpack.config.js

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

const {
  ModuleFederationPlugin
} = require('webpack').container

module.exports = {
  mode: 'production',
  entry: './src/index.js',
  plugins: [
    new HtmlWebpackPlugin(),
    new ModuleFederationPlugin({
      name: "home",
      filename: "remoteEntry.js",
    })
  ]
}
```

```
remotes: {
    nav: "nav@http://localhost:3003/remoteEntry.js",
},
exposes: {
    './HomeList': './src/HomeList.js',
},
shared: {},
}),
]
}
```

应用 webpack 运行服务：

```
[felix] nav $ npx webpack serve --port 3001
```

3、search 搜索

src/index

```
Promise.all([import('nav/Header'), import('home/HomeList')])
.then(([{
    default: Header
}, {
    default: HomeList
}]) => {
    document.body.appendChild(Header())
    document.body.innerHTML += HomeList(4)
})
```

webpack.config.js

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

const {
    ModuleFederationPlugin
} = require('webpack').container

module.exports = {
    mode: 'production',
    entry: './src/index.js',
    plugins: [
        new HtmlWebpackPlugin(),
        new ModuleFederationPlugin({
            name: 'search',
            filename: 'remoteEntry.js',
        })
    ]
}
```

```
        remotes: {
          nav: "nav@http://localhost:3003/remoteEntry.js",
          home: "home@http://localhost:3001/remoteEntry.js"
        },
        exposes: {},
        shared: {}
      ),
    ],
}
```

应用 webpack 运行服务：

```
[felix] nav $ npx webpack serve --port 3002
```

2.10 提升构建性能

2.10.1 通用环境

无论你是在 [开发环境](#) 还是在 [生产环境](#) 下运行构建脚本，以下最佳实践都会有所帮助。

1、更新到最新版本

使用最新的 webpack 版本。我们会经常进行性能优化。webpack 的最新稳定版本是：

将 [Node.js](#) 更新到最新版本，也有助于提高性能。除此之外，将你的 package 管理工具（例如 `npm` 或者 `yarn`）更新到最新版本，也有助于提高性能。较新的版本能够建立更高效的模块树以及提高解析速度。

2、loader

将 loader 应用于最少数量的必要模块。而非如下：

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

通过使用 `include` 字段，仅将 loader 应用在实际需要将其转换的模块：

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

module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        include: path.resolve(__dirname, 'src'),
        loader: 'babel-loader',
      },
    ],
  },
};
```

3、引导(bootstrap)

每个额外的 loader/plugin 都有其启动时间。尽量少地使用工具。

4、解析

以下步骤可以提高解析速度：

- 减少 `resolve.modules`, `resolve.extensions`, `resolve.mainFiles`, `resolve.descriptionFiles` 中条目数量，因为他们会增加文件系统调用的次数。
- 如果你不使用 symlinks (例如 `npm link` 或者 `yarn link`)，可以设置 `resolve.symlinks: false`。
- 如果你使用自定义 resolve plugin 规则，并且没有指定 context 上下文，可以设置 `resolve.cacheWithContext: false`。

5、小即是快(smaller = faster)

减少编译结果的整体大小，以提高构建性能。尽量保持 chunk 体积小。

- 使用数量更少/体积更小的 library。
- 在多页面应用程序中使用 `SplitChunksPlugin`。
- 在多页面应用程序中使用 `SplitChunksPlugin`，并开启 `async` 模式。
- 移除未引用代码。
- 只编译你当前正在开发的那些代码。

6、持久化缓存

在 webpack 配置中使用 `cache` 选项。使用 `package.json` 中的 `"postinstall"` 清除缓存目录。

将 `cache` 类型设置为内存或者文件系统。`memory` 选项很简单，它告诉 webpack 在内存中存储缓存，不允许额外的配置：

webpack.config.js

```
module.exports = {
  //...
  cache: {
    type: 'memory',
  },
};
```

7、自定义 plugin/loader

对它们进行概要分析，以免在此处引入性能问题。

8、dll

使用 `DllPlugin` 为更改不频繁的代码生成单独的编译结果。这可以提高应用程序的编译速度，尽管它增加了构建过程的复杂度。

9、worker 池(worker pool)

`thread-loader` 可以将非常消耗资源的 loader 分流给一个 worker pool。

不要使用太多的 worker，因为 Node.js 的 runtime 和 loader 都有启动开销。

最小化 worker 和 main process(主进程) 之间的模块传输。进程间通讯(IPC, inter process communication)是非常消耗资源的。

10、Progress plugin

将 `ProgressPlugin` 从 webpack 中删除，可以缩短构建时间。请注意，`ProgressPlugin` 可能不会为快速构建提供太多价值，因此，请权衡利弊再使用。

2.10.2 开发环境

以下步骤对于 **开发环境** 特别有帮助。

1、增量编译

使用 webpack 的 `watch mode`(监听模式)。而不使用其他工具来 `watch` 文件和调用 `webpack`。内置的 `watch mode` 会记录时间戳并将此信息传递给 `compilation` 以使缓存失效。

在某些配置环境中，`watch mode` 会回退到 `poll mode`(轮询模式)。监听许多文件会导致 CPU 大量负载。在这些情况下，可以使用 `watchOptions.poll` 来增加轮询的间隔时间。

2、在内存中编译

下面几个工具通过在内存中（而不是写入磁盘）编译和 `serve` 资源来提高性能：

- `webpack-dev-server`
- `webpack-hot-middleware`
- `webpack-dev-middleware`

3、`stats.toJson` 加速

webpack 4 默认使用 `stats.toJson()` 输出大量数据。除非在增量步骤中做必要的统计，否则请避免获取 `stats` 对象的部分内容。`webpack-dev-server` 在 v3.1.3 以后的版本，包含一个重要的性能修复，即最小化每个增量构建步骤中，从 `stats` 对象获取的数据量。

4、Devtool

需要注意的是不同的 `devtool` 设置，会导致性能差异。

- `"eval"` 具有最好的性能，但并不能帮助你转译代码。
- 如果你能接受稍差一些的 map 质量，可以使用 `cheap-source-map` 变体配置来提高性能
- 使用 `eval-source-map` 变体配置进行增量编译。

在大多数情况下，最佳选择是 `eval-cheap-module-source-map`。

5、避免在生产环境下才会用到的工具

某些 utility, plugin 和 loader 都只用于生产环境。例如，在开发环境下使用 `TerserPlugin` 来 minify(压缩) 和 mangle(混淆破坏) 代码是没有意义的。通常在开发环境下，应该排除以下这些工具：

- `TerserPlugin`
- `[fullhash] / [chunkhash] / [contenthash]`

- `AggressiveSplittingPlugin`
- `AggressiveMergingPlugin`
- `ModuleConcatenationPlugin`

6. 最小化 entry chunk

Webpack 只会在文件系统中输出已经更新的 chunk。某些配置选项 (HMR, `output.chunkFilename` 的 `[name]` / `[chunkhash]` / `[contenthash]`, `[fullhash]`) 来说，除了对已经更新的 chunk 无效之外，对于 entry chunk 也不会生效。

确保在生成 entry chunk 时，尽量减少其体积以提高性能。下面的配置为运行时代码创建了一个额外的 chunk，所以它的生成代价较低：

```
module.exports = {
  // ...
  optimization: {
    runtimeChunk: true,
  },
};
```

7. 避免额外的优化步骤

Webpack 通过执行额外的算法任务，来优化输出结果的体积和加载性能。这些优化适用于小型代码库，但是在大型代码库中却非常耗费性能：

```
module.exports = {
  // ...
  optimization: {
    removeAvailableModules: false,
    removeEmptyChunks: false,
    splitChunks: false,
  },
};
```

8. 输出结果不携带路径信息

Webpack 会在输出的 bundle 中生成路径信息。然而，在打包数千个模块的项目中，这会导致造成垃圾回收性能压力。在 `options.output.pathinfo` 设置中关闭：

```
module.exports = {
  // ...
  output: {
    pathinfo: false,
  },
};
```

9、Node.js 版本 8.9.10-9.11.1

Node.js v8.9.10 - v9.11.1 中的 ES2015 `Map` 和 `Set` 实现，存在 [性能回退](#)。Webpack 大量地使用这些数据结构，因此这次回退也会影响编译时间。

之前和之后的 Node.js 版本不受影响。

10、TypeScript loader

你可以为 loader 传入 `transpileOnly` 选项，以缩短使用 `ts-loader` 时的构建时间。使用此选项，会关闭类型检查。如果要再次开启类型检查，请使用 [ForkTsCheckerwebpackPlugin](#)。使用此插件会将检查过程移至单独的进程，可以加快 TypeScript 的类型检查和 ESLint 插入的速度。

```
module.exports = {
  // ...
  test: /\.tsx?$/,
  use: [
    {
      loader: 'ts-loader',
      options: {
        transpileOnly: true,
      },
    },
  ],
};
```

2.10.3 生产环境

以下步骤对于 [生产环境](#) 特别有帮助。

Source Maps

source map 相当消耗资源。你真的需要它们？

--本篇完--

#三、项目实战篇

- Webpack与React
- Webpack与Vue
- Webpack与jQuery
- Webpck与Node/Express

#四、内部原理篇

webpack原理

开发loader plugin