Webpack构建工具

15.1.6 Webpack简介

前言

Web 应用日益复杂，相关开发技术也百花齐放，这对前端构建工具提出了更高的要求。 Webpack 从众多构建工具中脱颖而出成为目前最流行的构建工具，几乎成为目前前端开发里的必备工具之一， 因此每位紧跟时代的前端工程师都应该掌握 Webpack。

前端的发展

近年来 Web 应用变得更加复杂与庞大，Web 前端技术的应用范围也更加广泛。 从复杂庞大的管理后台到对性能要求苛刻的移动网页，再到类似 ReactNative 的原生应用开发方案，Web 前端工程师在面临更多机遇的同时也会面临更大的挑战。 通过直接编写 JavaScript、CSS、HTML 开发 Web 应用的方式已经无法应对当前 Web 应用的发展。近年来前端社区涌现出许多新思想与框架，下面将一一介绍它们。

模块化

模块化是指把一个复杂的系统分解到多个模块以方便编码。

很久以前，开发网页要通过命名空间的方式来组织代码，例如 jQuery 库把它的API都放在了 window.$ 下，在加载完 jQuery 后其他模块再通过 window.$ 去使用 jQuery。 这样做有很多问题，其中包括：

- 命名空间冲突，两个库可能会使用同一个名称，例如 Zepto 也被放在 window.$ 下；
- 无法合理地管理项目的依赖和版本；
- 无法方便地控制依赖的加载顺序。 当项目变大时这种方式将变得难以维护，需要用模块化的思想来组织代码。

CommonJS

CommonJS 是一种使用广泛的 JavaScript 模块化规范，核心思想是通过 require 方法来同步地加载依赖的其他模块，通过 module.exports 导出需要暴露的接口。 CommonJS 规范的流行得益于 Node.js 采用了这种方式，后来这种方式被引入到了网页开发中。

采用 CommonJS 导入及导出时的代码如下：

```
// 导入
const moduleA = require('./moduleA');

// 导出
module.exports = moduleA.someFunc;
```

CommonJS 的优点在于：

- 代码可复用于 Node.js 环境下并运行，例如做同构应用；

- 通过 NPM 发布的很多第三方模块都采用了 CommonJS 规范。 CommonJS 的缺点在于这样的代码无法直接运行在浏览器环境下，必须通过工具转换成标准的 ES5。

> CommonJS 还可以细分为 CommonJS1 和 CommonJS2，区别在于 CommonJS1 只能通过 exports.XX = XX 的方式导出，CommonJS2 在 CommonJS1 的基础上加入了 module.exports = XX 的导出方式。CommonJS 通常指 CommonJS2。

AMD

AMD 也是一种 JavaScript 模块化规范，与 CommonJS 最大的不同在于它采用异步的方式去加载依赖的模块。 AMD 规范主要是为了解决针对浏览器环境的模块化问题，最具代表性的实现是 requirejs。

采用 AMD 导入及导出时的代码如下：

```
// 定义一个模块
define('module', ['dep'], function(dep) {
  return exports;
});

// 导入和使用
require(['module'], function(module) {
});
```

AMD 的优点在于：

- 可在不转换代码的情况下直接在浏览器中运行；
- 可异步加载依赖；
- 可并行加载多个依赖；
- 代码可运行在浏览器环境和 Node.js 环境下。 AMD 的缺点在于JavaScript 运行环境没有原生支持 AMD，需要先导入实现了 AMD 的库后才能正常使用。

ES6 模块化

ES6 模块化是欧洲计算机制造联合会 ECMA 提出的 JavaScript 模块化规范，它在语言的层面上实现了模块化。浏览器厂商和 Node.js 都宣布要原生支持该规范。它将逐渐取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

采用 ES6 模块化导入及导出时的代码如下：

```
// 导入
import { readFile } from 'fs';
import React from 'react';
// 导出
export function hello() {};
export default {
  // ...
};
```

ES6模块虽然是终极模块化方案，但它的缺点在于目前无法直接运行在大部分 JavaScript 运行环境下，必须通过工具转换成标准的 ES5 后才能正常运行。

样式文件中的模块化

除了 JavaScript 开始模块化改造，前端开发里的样式文件也支持模块化。 以 SCSS 为例，把一些常用的样式片段放进一个通用的文件里，再在另一个文件里通过 @import 语句去导入和使用这些样式片段。

```
// util.scss 文件

// 定义样式片段
@mixin center {
  // 水平竖直居中
  position: absolute;
  left: 50%;
  top: 50%;
  transform: translate(-50%,-50%);
}

// main.scss 文件

// 导入和使用 util.scss 中定义的样式片段
@import "util";
#box{
  @include center;
}
```

新框架

在 Web 应用变得庞大复杂时，采用直接操作 DOM 的方式去开发将会使代码变得复杂和难以维护， 许多新思想被引入到网页开发中以减少开发难度、提升开发效率。

React

React 框架引入 JSX 语法到 JavaScript 语言层面中，以更灵活地控制视图的渲染逻辑。

```
let has = true;
render(has ? <h1>hello,react</h1> : <div>404</div>);
```

这种语法无法直接在任何现成的 JavaScript 引擎里运行，必须经过转换。

Vue

Vue 框架把一个组件相关的 HTML 模版、JavaScript 逻辑代码、CSS 样式代码都写在一个文件里，这非常直观。

```
<!--HTML 模版-->
<template>
  <div class="example">{{ msg }}</div>
</template>

<!--JavaScript 组件逻辑-->
<script>
export default {
  data () {
    return {
      msg: 'Hello world!'
    }
  }
}
</script>

<!--CSS 样式-->
<style>
.example {
  font-weight: bold;
}
</style>
```

Angular

Angular 推崇采用 TypeScript 语言去开发应用，并且可以通过注解的语法描述组件的各种属性。

```
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>`
})
export class AppComponent {
  title = 'Tour of Heroes';
}
```

ES6

ECMAScript 6.0（简称 ES6）是 JavaScript 语言的下一代标准。它在语言层面为 JavaScript 引入了很多新语法和 API，使得 JavaScript 语言可以用来编写复杂的大型应用程序。例如：

- 规范模块化；
- Class 语法；
- 用 let 声明代码块内有效的变量，用 const 声明常量；
- 箭头函数；
- async 函数；
- Set 和 Map 数据结构。 通过这些新特性，可以更加高效地编写代码，专注于解决问题本身。但遗憾的是不同浏览器对这些特性的支持不一致，使用了这些特性的代码可能会在部分浏览器下无法运行。为

了解决兼容性问题，需要把 ES6 代码转换成 ES5 代码，Babel 是目前解决这个问题最好的工具。 Babel 的插件机制让它可灵活配置，支持把任何新语法转换成 ES5 的写法。

TypeScript

TypeScript 是 JavaScript 的一个超集，由 Microsoft 开发并开源，除了支持 ES6 的所有功能，还提供了静态类型检查。采用 TypeScript 编写的代码可以被编译成符合 ES5、ES6 标准的 JavaScript。 将 TypeScript 用于开发大型项目时，其优点才能体现出来，因为大型项目由多个模块组合而成，不同模块可能又由不同人编写，在对接不同模块时静态类型检查会在编译阶段找出可能存在的问题。 TypeScript 的缺点在于语法相对于 JavaScript 更加啰嗦，并且无法直接运行在浏览器或 Node.js 环境下。

```
// 静态类型检查机制会检查传给 hello 函数的数据类型
function hello(content: string) {
  return `Hello, ${content}`;
}
let content = 'word';
hello(content);
```

15.1.7 常见的构建工具及对比

Grunt

Grunt 和 Npm Script 类似，也是一个任务执行者。Grunt 有大量现成的插件封装了常见的任务，也能管理任务之间的依赖关系，自动化执行依赖的任务，每个任务的具体执行代码和依赖关系写在配置文件 Gruntfile.js 里，例如：

```
module.exports = function(grunt) {
  // 所有插件的配置信息
  grunt.initConfig({
    // uglify 插件的配置信息
    uglify: {
      app_task: {
        files: {
          'build/app.min.js': ['lib/index.js', 'lib/test.js']
        }
      }
    },
    // watch 插件的配置信息
    watch: {
      another: {
          files: ['lib/*.js'],
      }
    }
  });

  // 告诉 grunt 我们将使用这些插件
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-watch');
```

```
  // 告诉grunt当我们在终端中启动 grunt 时需要执行哪些任务
  grunt.registerTask('dev', ['uglify','watch']);
};
```

在项目根目录下执行命令 grunt dev 就会启动 JavaScript 文件压缩和自动刷新功能。

Grunt的优点是：

- 灵活，它只负责执行你定义的任务；
- 大量的可复用插件封装好了常见的构建任务。 Grunt的缺点是集成度不高，要写很多配置后才可以用，无法做到开箱即用。

Gulp

Gulp 是一个基于流的自动化构建工具。 除了可以管理和执行任务，还支持监听文件、读写文件。Gulp 被设计得非常简单，只通过下面5个方法就可以胜任几乎所有构建场景：

- 通过 gulp.task 注册一个任务；
- 通过 gulp.run 执行任务；
- 通过 gulp.watch 监听文件变化；
- 通过 gulp.src 读取文件；
- 通过 gulp.dest 写文件。 Gulp 的最大特点是引入了流的概念，同时提供了一系列常用的插件去处理流，流可以在插件之间传递，大致使用如下：

```
// 引入 Gulp
var gulp = require('gulp');
// 引入插件
var jshint = require('gulp-jshint');
var sass = require('gulp-sass');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');

// 编译 SCSS 任务
gulp.task('sass', function() {
  // 读取文件通过管道喂给插件
  gulp.src('./scss/*.scss')
    // SCSS 插件把 scss 文件编译成 CSS 文件
    .pipe(sass())
    // 输出文件
    .pipe(gulp.dest('./css'));
});

// 合并压缩 JS
gulp.task('scripts', function() {
  gulp.src('./js/*.js')
    .pipe(concat('all.js'))
    .pipe(uglify())
    .pipe(gulp.dest('./dist'));
});

// 监听文件变化
```

```
gulp.task('watch', function(){
  // 当 scss 文件被编辑时执行 SCSS 任务
  gulp.watch('./scss/*.scss', ['sass']);
  gulp.watch('./js/*.js', ['scripts']);
});
```

Gulp 的优点是好用又不失灵活，既可以单独完成构建也可以和其它工具搭配使用。其缺点是和 Grunt 类似，集成度不高，要写很多配置后才可以用，无法做到开箱即用。

Fis3

Fis3 是一个来自百度的优秀国产构建工具。相对于 Grunt、Gulp 这些只提供基本功能的工具，Fis3 集成了 Web 开发中的常用构建功能，如下所述。

- 读写文件：通过 fis.match 读文件，release 配置文件输出路径。
- 资源定位：解析文件之间的依赖关系和文件位置。
- 文件指纹：通过 useHash 配置输出文件时给文件 URL 加上 md5 戳来优化浏览器缓存。
- 文件编译：通过 parser 配置文件解析器做文件转换，例如把 ES6 编译成 ES5。
- 压缩资源：通过 optimizer 配置代码压缩方法。
- 图片合并：通过 spriter 配置合并 CSS 里导入的图片到一个文件来减少 HTTP 请求数。

```
// 加 md5
fis.match('*.{js,css,png}', {
  useHash: true
});

// fis3-parser-typescript 插件把 TypeScript 文件转换成 JavaScript 文件
fis.match('*.ts', {
  parser: fis.plugin('typescript')
});

// 对 CSS 进行雪碧图合并
fis.match('*.css', {
  // 给匹配到的文件分配属性 `useSprite`
  useSprite: true
});

// 压缩 JavaScript
fis.match('*.js', {
  optimizer: fis.plugin('uglify-js')
});

// 压缩 CSS
fis.match('*.css', {
  optimizer: fis.plugin('clean-css')
});

// 压缩图片
fis.match('*.png', {
```

```
  optimizer: fis.plugin('png-compressor')
});
```

可以看出 Fis3 很强大，内置了许多功能，无须做太多配置就能完成大量工作。

Fis3的优点是集成了各种 Web 开发所需的构建功能，配置简单开箱即用。其缺点是目前官方已经不再更新和维护，不支持最新版本的 Node.js。 Fis3 是一种专注于 Web 开发的完整解决方案，如果将 Grunt、Gulp 比作汽车的发动机，则可以将Fis3 比作一辆完整的汽车.

Webpack

Webpack 是一个打包模块化 JavaScript 的工具，在 Webpack 里一切文件皆模块，通过 Loader 转换文件，通过 Plugin 注入钩子，最后输出由多个模块组合成的文件。Webpack 专注于构建模块化项目。

一切文件：JavaScript、CSS、SCSS、图片、模板，在 Webpack 眼中都是一个个模块，这样的好处是能清晰的描述出各个模块之间的依赖关系，以方便 Webpack 对模块进行组合和打包。 经过 Webpack 的处理，最终会输出浏览器能使用的静态资源。

Webpack 具有很大的灵活性，能配置如何处理文件，大致使用如下：

```
module.exports = {
  // 所有模块的入口，Webpack 从入口开始递归解析出所有依赖的模块
  entry: './app.js',
  output: {
    // 把入口所依赖的所有模块打包成一个文件 bundle.js 输出
    filename: 'bundle.js'
  }
}
```

Webpack的优点是：

- 专注于处理模块化的项目，能做到开箱即用一步到位；
- 通过 Plugin 扩展，完整好用又不失灵活；
- 使用场景不仅限于 Web 开发；
- 社区庞大活跃，经常引入紧跟时代发展的新特性，能为大多数场景找到已有的开源扩展；
- 良好的开发体验。 Webpack的缺点是只能用于采用模块化开发的项目。

15.1.8 安装 Webpack

在用 Webpack 执行构建任务时需要通过 webpack 可执行文件去启动构建任务，所以需要安装 webpack 可执行文件。

安装 Webpack 到全局

```
npm i -g webpack
npm i -g webpack-cli
```

安装 Webpack 到本项目

选择文件夹创建项目目录

```
mkdir webpack-demo
cd webpack-demo
```

初始化项目

```
npm init
```

要安装 Webpack 到本项目，可按照你的需要选择以下任意命令运行：

```
npm i -D webpack
npm i -D webpack-cli
```

使用webpack

创建项目结构`src`、`dist`、`public`、`webpack.config.js`

在`src`下创建`show.js`文件，并写入以下代码

```
function show(content) {
    window.document.getElementById('root').innerText = 'Hello,' + content;
}
module.exports = show
```

在`src`下创建`index.js`主文件，并写入以下代码

```
// 通过 CommonJS 规范导入 show 函数
const show = require('./show.js');
// 执行 show 函数
show('iwen');
```

在项目的根目录下创建`webpack.config.js`文件并输入以下代码

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

module.exports = {
  // JavaScript 执行入口文件
  entry: './src/index.js',
  output: {
```

```
    // 把所有依赖的模块合并输出到一个 bundle.js 文件
    filename: 'bundle.js',
    // 输出文件都放到 dist 目录下
    path: path.resolve(__dirname, './dist'),
  }
};
```

运行webpack查看生成的bundle.js文件

```
webpack
```

在dist创建index.html文件引入bundle.js文件查看运行效果

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="root"></div>
    <script src="./bundle.js"></script>
</body>
</html>
```

15.1.9 使用 Loader

在上一节中使用 Webpack 构建了一个采用 CommonJS 规范的模块化项目，本节将继续优化这个网页的 UI，为项目引入 CSS 代码让文字居中显示，main.css 的内容如下： 在src下创建assets/css/main.css并输入以下代码

```
#root{
  text-align: center;
}
```

Webpack 把一切文件看作模块，CSS 文件也不例外，要引入 main.css 需要像引入 JavaScript 文件那样，修改入口文件 index.js 如下：

```
// 通过 CommonJS 规范导入 CSS 模块
require('./assets/css/main.css');
// 通过 CommonJS 规范导入 show 函数
const show = require('./show.js');
// 执行 show 函数
```

```
show('iwen');
```

但是这样修改后去执行 Webpack 构建是会报错的，因为 Webpack 不原生支持解析 CSS 文件。要支持非 JavaScript 类型的文件，需要使用 Webpack 的 Loader 机制。Webpack的配置修改使用如下：

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

module.exports = {
    // JavaScript 执行入口文件
    entry: './src/index.js',
    output: {
        // 把所有依赖的模块合并输出到一个 bundle.js 文件
        filename: 'bundle.js',
        // 输出文件都放到 dist 目录下
        path: path.resolve(__dirname, './dist'),
    },
    module: {
        rules: [
            {
                // 用正则去匹配要用该 loader 转换的 CSS 文件
                test: /\.css$/,
                use: ['style-loader', 'css-loader'],
            }
        ]
    }
};
```

然后安装CSS需要的依赖

```
npm i -D style-loader css-loader
```

修改快捷运行方案

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

运行一下命令查看CSS样式效果

```
npm start
```

15.1.10 使用 Plugin

插件安装与使用

Plugin 是用来扩展 Webpack 功能的，通过在构建流程里注入钩子实现，它给 Webpack 带来了很大的灵活性。

在上一节中通过 Loader 加载了 CSS 文件，本节将通过 Plugin 把注入到 bundle.js 文件里的 CSS 提取到单独的文件中，配置修改如下：

```
const path = require('path');
// 压缩CSS代码
const optimizeCss = require('optimize-css-assets-webpack-plugin');
// 分离CSS代码
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    // JavaScript 执行入口文件
    entry: './src/index.js',
    output: {
        // 把所有依赖的模块合并输出到一个 bundle.js 文件
        filename: 'bundle.js',
        // 输出文件都放到 dist 目录下
        path: path.resolve(__dirname, './dist'),
    },
    module: {
        rules: [
            {
                // 用正则去匹配要用该 loader 转换的 CSS 文件
                test: /\.css$/,
                use: [
                    {
                        loader: MiniCssExtractPlugin.loader
                    },
                    "css-loader"
                ]
            },
        ]
    },
    plugins: [
        // css代码压缩
        new optimizeCss(),
        // 单独提取CSS文件
        new MiniCssExtractPlugin({
            filename: "main.css",
            chunkFilename: "[id].css"
        }),
    ],
};
```

需要安装依赖

```
npm install --save-dev optimize-css-assets-webpack-plugin
npm install --save-dev mini-css-extract-plugin
```

此时css文件已经分离，需要在html文件中引入

HTML文件分离

此时我们的文件是放入在dist文件夹中，但是dist本事是动态生成的生产环境，所以。我们此时创建public文件夹，然后配置HTML文件分离

安装依赖

```
npm install -D html-webpack-plugin
```

创建public文件夹，然后移动index.html到public文件夹中

修改配置文件

```
const HTMLPlugin = require('html-webpack-plugin')
plugins: [
    new HTMLPlugin({
        template: './public/index.html'
    })
]
```

15.1.11 使用 DevServer

前面的几节只是让 Webpack 正常运行起来了，但在实际开发中你可能会需要：

- 提供 HTTP 服务而不是使用本地文件预览；
- 监听文件的变化并自动刷新网页，做到实时预览；
- 支持 Source Map，以方便调试。 对于这些， Webpack 都为你考虑好了。Webpack 原生支持上述第2、3点内容，再结合官方提供的开发工具 DevServer 也可以很方便地做到第1点。 DevServer 会启动一个 HTTP 服务器用于服务网页请求，同时会帮助启动 Webpack ，并接收 Webpack 发出的文件更变信号，通过 WebSocket 协议自动刷新网页做到实时预览。

先安装依赖，注意由于版本问题，此处修改`webpack-cli`安装版本为`3.3.12`

```
npm install -D webpack-dev-server
```

增加一下关于服务器配置

```

```

```
    devServer: {
        // 服务器打开目录
        contentBase: path.join(__dirname, 'dist'),
        // 压缩
        compress: true,
        port: 9000,
        hot: true,
        open: true
    },
```

修改package.json文件

```
    "dev": "webpack-dev-server --config webpack.config.js"
```

运行服务器看效果

```
npm run dev
```

关于更多配置，请查看webpack官网

核心概念

通过之前几节的学习，相信你已经对 Webpack 有了一个初步的认识。虽然Webpack 功能强大且配置项多，但只要你理解了其中的几个核心概念，就能随心应手地使用它。 Webpack 有以下几个核心概念。

- Entry：入口，Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入。
- Module：模块，在 Webpack 里一切皆模块，一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。
- Chunk：代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。
- Loader：模块转换器，用于把模块原内容按照需求转换成新内容。
- Plugin：扩展插件，在 Webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要的事情。
- Output：输出结果，在 Webpack 经过一系列处理并得出最终想要的代码后输出结果。

15.1.12 更多配置

Entry

entry是配置模块的入口，可抽象成输入，Webpack 执行构建的第一步将从入口开始搜寻及递归解析出所有入口依赖的模块。

entry 配置是必填的，若不填则将导致 Webpack 报错退出。

我们常用的可以是多入口的配置，例如，在一个项目中，可以存在多个入口地址，此时可以在entry中配置。

增加其他入口文件dist/login.html和src/login/login.js

```
// login.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="root"></div>
    <script src="./login.js"></script>
</body>
</html>
```

```
console.log("登录页面")
```

修改配置文件入口如下

```
    entry: {
        index: "./src/index.js",
        login: "./src/login/login.js"
    },
    // 模式区分：生产和开发环境
    mode: 'development',
    output: {
        // 把所有依赖的模块合并输出到一个 bundle.js 文件
        filename: '[name].js',
        // 输出文件都放到 dist 目录下
        path: path.resolve(__dirname, './dist'),
    }
```

运行代码之后，我们可以访问两个入口,在login的路径中可以看到打印

```
http://localhost:9000
http://localhost:9000/login
```

Resolve

Webpack 在启动后会从配置的入口模块出发找出所有依赖的模块，Resolve 配置 Webpack 如何寻找模块所对应的文件。 Webpack 内置 JavaScript 模块化语法解析功能，默认会采用模块化标准里约定好的规则去寻找，但你也可以根据自己的需要修改默认的规则。

resolve下面有很多配置，我们这里介绍一个简单的,取消添加后缀

```
resolve: {
    extensions: ['.js', '.json']
},
```

使用ES6语言

通常我们需要把采用 ES6 编写的代码转换成目前已经支持良好的 ES5 代码，这包含2件事：

- 把新的 ES6 语法用 ES5 实现，例如 ES6 的 class 语法用 ES5 的 prototype 实现。
- 给新的 API 注入 polyfill，例如项目使用 fetch API 时，只有注入对应的 polyfill 后，才能在低版本浏览器中正常运行。

安装babel相关依赖

```
$ npm install --save-dev @babel/core
# 最新转码规则
$ npm install --save-dev @babel/preset-env
# babel依赖
$ npm install --save-dev babel-loader
```

在项目根目录下，创建`.babelrc`文件然后，将这些规则加入`.babelrc`.

```
{
  "presets": [
    "@babel/env"
  ],
  "plugins": []
}
```

修改`webpack.config.js`文件

```
const path = require('path');
const optimizeCss = require('optimize-css-assets-webpack-plugin');
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    // JavaScript 执行入口文件
    entry: {
        index: "./src/index.js",
        login: "./src/login/login.js"
    },
    // 模式区分：生产和开发环境
    mode: 'development',
    output: {
        // 把所有依赖的模块合并输出到一个 bundle.js 文件
        filename: '[name].js',
```

```
        // 输出文件都放到 dist 目录下
        path: path.resolve(__dirname, './dist'),
    },
    module: {
        rules: [
            {
                // 用正则去匹配要用该 loader 转换的 CSS 文件
                test: /\.css$/,
                use: [
                    {
                        loader: MiniCssExtractPlugin.loader
                    },
                    "css-loader"
                ]
            },
            {
                test: /\.js$/,
                use: ['babel-loader'],
                include: path.resolve(__dirname, 'src')
            },
        ]
    },
    plugins: [
        // css代码压缩
        new optimizeCss(),
        // 单独提取CSS文件
        new MiniCssExtractPlugin({
            filename: "main.css",
            chunkFilename: "[id].css"
        }),
    ],
    devServer: {
        contentBase: path.join(__dirname, 'dist'),
        compress: true,
        port: 9000,
        hot: true,
        open: true
    },
    resolve: {
        extensions: ['.js', '.json']
    },
};
```

接下来将代码修改为ES6，index.js修改如下

```
import "../assets/css/main.css"
import show from "./show"
show('iwen!');
```

show.js修改如下

```
function show(content) {
    window.document.getElementById('root').innerText = 'Hello,' + content;
}

export default show
```

15.1.13 使用React框架

接下来我们构建一个React基础环境,在React中我们需要解决的jsx语法问题

安装依赖:

```
npm install -D @babel/preset-react
npm install -S react
npm install -S react-dom
```

在配置文件中，我们也需要添加jsx语法的解析

```
{
    test: /\.(js|jsx)$/,//一个匹配loaders所处理的文件的拓展名的正则表达式，这里用来匹配js和jsx文件（必须）
    exclude: /node_modules/,//屏蔽不需要处理的文件（文件夹）（可选）
    loader: 'babel-loader',//loader的名称（必须）
}
```

同时我们需要修改.babelrc文件

```
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "modules": false,
                "targets": {
                    "browsers": [
                        "> 1%",
                        "last 2 versions",
                        "not ie <= 8"
                    ]
                }
            }
        ],
        "@babel/preset-react"
    ]
}
```

修改主文件，键入React代码

```
import React from 'react'
import ReactDom from 'react-dom'

class App extends React.Component {
    render(){
        return (
            <div style={{color:"#333"}}>
                Hello
            </div>
        )
    }
}
ReactDom.render(<App/>,document.getElementById("root"))
```

15.1.14 使用 Vue 框架

Vue 是一个渐进式的 MVVM 框架，相比于 React、Angular 它更灵活轻量。 它不会强制性地内置一些功能和语法，你可以根据自己的需要一点点地添加功能。 虽然采用 Vue 的项目能用可直接运行在浏览器环境里的代码编写，但为了方便编码大多数项目都会采用 Vue 官方的单文件组件的写法去编写项目。 由于直接引用 Vue 是古老和成熟的做法，本书只专注于讲解如何用 Webpack 构建 Vue 单文件组件。

接下来我们配置一下相关环境 css预处理器

```
npm i -D stylus stylus-loader
npm i -D postcss-loader
```

预处理器工具

```
npm i -D autoprefixer
```

处理html文件

```
npm i -D html-webpack-plugin
```

处理vue文件

```
npm i -D vue vue-loader vue-style-loader vue-template-loader vue-template-
compiler
```

修改`webpack.config.js`文件

```
const path = require('path');
const optimizeCss = require('optimize-css-assets-webpack-plugin');
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const VueLoaderPlugin = require('vue-loader/lib/plugin')
const HTMLPlugin = require('html-webpack-plugin')
const webpack = require('webpack')


module.exports = {
    // JavaScript 执行入口文件
    entry: {
        index: "./src/index.js",
        login: "./src/login/login.js"
    },
    // 模式区分：生产和开发环境
    mode: 'development',
    output: {
        // 把所有依赖的模块合并输出到一个 bundle.js 文件
        filename: '[name].js',
        // 输出文件都放到 dist 目录下
        path: path.resolve(__dirname, './dist'),
    },
    module: {
        rules: [
            {
                // 用正则去匹配要用该 loader 转换的 CSS 文件
                test: /\.css$/,
                use: [
                    {
                        loader: MiniCssExtractPlugin.loader
                    },
                    "css-loader"
                ]
            },
            {
                test: /\.js$/,
                use: ['babel-loader'],
                include: path.resolve(__dirname, 'src')
            },
            {
                test: /\.vue$/,
                use: {
                    loader: 'vue-loader'
                }
            },
            {
                test: /\.styl/,
                use: [
                    'style-loader',
                    'css-loader',
                    'postcss-loader',
                    'stylus-loader'
```

```
                ]
            }
        ]
    },
    plugins: [
        // css代码压缩
        new optimizeCss(),
        // 单独提取CSS文件
        new MiniCssExtractPlugin({
            filename: "main.css",
            chunkFilename: "[id].css"
        }),
        new VueLoaderPlugin(),
        new HTMLPlugin({
            filename: './public/index.html',
            template: './public/index.html'
        }),
        new webpack.HotModuleReplacementPlugin()
    ],
    devServer: {
        contentBase: path.join(__dirname, 'dist'),
        compress: true,
        port: 9000,
        hot: true,
        open: true
    },
    resolve: {
        extensions: ['.js', '.json']
    }
};
```

增加Vue文件`components/hello.vue

```
<template>
  <div>
      Vue文件环境测试
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

修改主入口文件index.js如下

```
import "../assets/css/main.css"
import Vue from 'vue';
import App from './components/hello.vue';
new Vue({
  el: '#root',
  render: h => h(App)
});
```

复制index.html入口文件到public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" href="main.css">
</head>
<body>
    <div id="root"></div>
    <script src="./index.js"></script>
</body>
</html>
```

同时需要处理css，在项目根目录下创建postcss.config.js

```
const autoprefixer = require('autoprefixer')
module.exports = {
    plugins:[
        autoprefixer()
    ]
}
```

15.1.15 加载图片

在网页中不可避免的会依赖图片资源，例如 PNG、JPG、GIF，下面来教你如何用 Webpack 加载图片资源。

安装依赖如下：

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

修改配置文件如下：

```

```

```
{
    test: /\.(png|jpe?g|gif)$/i,
    use: [
        {
            loader: 'file-loader',
        },
    ],
},
```

然后在Vue文件中引入图片组件

```
<template>
  <div>Vue文件环境测试<img :src="myimg" alt=""></div>
</template>

<script>
import myimg from "../assets/images/timg.jpg";
export default {
  data() {
    return {
      myimg,
    };
  },
};
</script>

<style>
</style>
```

15.1.16 优化

缩小文件搜索范围

Webpack 启动后会从配置的 Entry 出发，解析出文件中的导入语句，再递归的解析。 在遇到导入语句时 Webpack 会做两件事情：

根据导入语句去寻找对应的要导入的文件。例如 require('react') 导入语句对应的文件是 ./node_modules/react/react.js，require('./util') 对应的文件是 ./util.js。 根据找到的要导入文件的后缀，使用配置中的 Loader 去处理文件。例如使用 ES6 开发的 JavaScript 文件需要使用 babel-loader 去处理。 以上两件事情虽然对于处理一个文件非常快，但是当项目大了以后文件量会变的非常多，这时候构建速度慢的问题就会暴露出来。 虽然以上两件事情无法避免，但需要尽量减少以上两件事情的发生，以提高速度。

接下来一一介绍可以优化它们的途径。

优化 loader 配置

以采用 ES6 的项目为例，在配置 babel-loader 时，可以这样：

```
module.exports = {
  module: {
    rules: [
      {
        // 如果项目源码中只有 js 文件就不要写成 /\.jsx?$/，提升正则表达式性能
        test: /\.js$/,
        use: ['babel-loader'],
        // 只对项目根目录下的 src 目录中的文件采用 babel-loader
        include: path.resolve(__dirname, 'src'),
      },
    ]
  },
};
```

区分环境

我们在以下配置好的环境下发现，一些运行时分环境的，开发模式与生产模式。 创建 `webpack.production.config.js`,将不需要在运行环境下的配置全部删除，如下：

```
const path = require('path');
const optimizeCss = require('optimize-css-assets-webpack-plugin');
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const VueLoaderPlugin = require('vue-loader/lib/plugin')
const HTMLPlugin = require('html-webpack-plugin')
const webpack = require('webpack')


module.exports = {
    // JavaScript 执行入口文件
    entry: {
        index: "./src/index.js",
        login: "./src/login/login.js"
    },
    // 模式区分：生产和开发环境
    mode: 'production',
    output: {
        // 把所有依赖的模块合并输出到一个 bundle.js 文件
        filename: '[name].js',
        // 输出文件都放到 dist 目录下
        path: path.resolve(__dirname, './dist'),
    },
    module: {
        rules: [
            {
                // 用正则去匹配要用该 loader 转换的 CSS 文件
                test: /\.css$/,
                use: [
                    {
                        loader: MiniCssExtractPlugin.loader
                    },
```

```
                    "css-loader"
                ]
            },
            {
                test: /\.js$/,
                use: ['babel-loader'],
                // 这里主要是配置优化，缩小范围，只处理src文件夹下内容
                include: path.resolve(__dirname, 'src')
            },
            {
                test: /\.vue$/,
                use: {
                    loader: 'vue-loader'
                }
            },
            {
                test: /\.styl/,
                use: [
                    'style-loader',
                    'css-loader',
                    'postcss-loader',
                    'stylus-loader'
                ]
            },
            {
                test: /\.(png|jpe?g|gif)$/i,
                use: [
                    {
                        loader: 'file-loader',
                    },
                ],
            }
        ]
    },
    plugins: [
        // css代码压缩
        new optimizeCss(),
        // 单独提取CSS文件
        new MiniCssExtractPlugin({
            filename: "main.css",
            chunkFilename: "[id].css"
        }),
        new VueLoaderPlugin(),
        new HTMLPlugin({
            filename: './public/index.html',
            template: './public/index.html'
        }),
        new webpack.HotModuleReplacementPlugin()
    ],
    resolve: {
        extensions: ['.js', '.json']
    }
};
```

然后我们修改package.json文件

```
"scripts": {
    "start": "webpack --config webpack.production.config.js",
    "build": "webpack --config webpack.production.config.js",
    "dev": "webpack-dev-server --config webpack.config.js"
},
```

这样我们可以运行npm run build和npm run dev来区分开发与生产环境