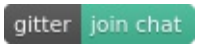
目錄

Webpack 中文指南



Webpack 是当下最热门的前端资源模块化管理和打包工具。它可以将许多松散的模块按照依赖和规则打包成符合生产环境部署的前端资源。还可以将按需加载的模块进行代码分隔，等到实际需要的时候再异步加载。通过 `loader` 的转换，任何形式的资源都可以视作模块，比如 CommonJs 模块、 AMD 模块、 ES6 模块、CSS、图片、 JSON、Coffeescript、 LESS 等。

主站 · 下载电子版 · 国内镜像1（掘金） · 国内镜像2（极客学院）

贡献内容

如果你想参与这本书的共同创作，修改或添加内容，可以先 Fork 这本书的仓库，然后将修改的内容提交 Pull requests ；或者创建 Issues。

Fork 后的仓库如何同步本仓库？

```
// 添加 upstream 源，只需执行一次
$ git remote add upstream git@github.com:zhaoda/webpack-handbook.git

// 拉取远程代码
$ git pull upstream master

// 提交修改
$ git add .
$ git commit

// 更新 fork 仓库
$ git push origin master
```

更多参考： Syncing a fork

注意，本书内容在 `/content` 目录中， gh-pages 分支和 Wiki 版是通过脚本自动生成的。

生成电子书

这本书使用 Gitbook 撰写并生成网站，请查看 `package.json` 中的 `scripts` 配置和 `/scripts` 目录中的脚本来了解这本书的构建和发布过程。

```
// 初始化 nodejs 依赖
$ npm install

// 安装 gitbook 插件
$ npm install gitbook-cli -g
$ gitbook install ./content

// 启动 gitbook 服务开始撰写工作
$ npm run serve-gitbook

// 生成 gitbook
$ npm run generate-gitbook

// 生成 wiki
$ npm run generate-wiki

// 发布到 gh-pages 分支
$ npm run deploy-gitbook

// 发布到 wiki
$ npm run deploy-wiki

// 生成并发布，是上面4条命令的快捷方式，通常编辑内容后只需要进行这个操作
$ npm run generate-and-deploy
```

更新日志

https://github.com/zhaoda/webpack-handbook/commits/master

版权许可

前言

介绍前端模块系统的演进历史，以及 Webpack 出现的背景及其特点。

现状

伴随着移动互联的大潮，当今越来越多的网站已经从网页模式进化到了 Webapp 模式。它们运行在现代的高级浏览器里，使用 HTML5、CSS3、ES6 等更新的技术来开发丰富的功能，网页已经不仅仅是完成浏览的基本需求，并且webapp通常是一个单页面应用，每一个视图通过异步的方式加载，这导致页面初始化和使用过程中会加载越来越多的 JavaScript 代码，这给前端开发的流程和资源组织带来了巨大的挑战。

前端开发和其他开发工作的主要区别，首先是前端是基于多语言、多层次的编码和组织工作，其次前端产品的交付是基于浏览器，这些资源是通过增量加载的方式运行到浏览器端，如何在开发环境组织好这些碎片化的代码和资源，并且保证他们在浏览器端快速、优雅的加载和更新，就需要一个模块化系统，这个理想中的模块化系统是前端工程师多年来一直探索的难题。

模块系统的演进

模块系统主要解决模块的定义、依赖和导出，先来看看已经存在的模块系统。

<script>标签

```
<script src="module1.js"></script>
<script src="module2.js"></script>
<script src="libraryA.js"></script>
<script src="module3.js"></script>
```

这是最原始的 JavaScript 文件加载方式，如果把每一个文件看做是一个模块，那么他们的接口通常是暴露在全局作用域下，也就是定义在 `window` 对象中，不同模块的接口调用都是一个作用域中，一些复杂的框架，会使用命名空间的概念来组织这些模块的接口，典型的例子如 YUI 库。

这种原始的加载方式暴露了一些显而易见的弊端：

- 全局作用域下容易造成变量冲突
- 文件只能按照 `<script>` 的书写顺序进行加载
- 开发人员必须主观解决模块和代码库的依赖关系
- 在大型项目中各种资源难以管理，长期积累的问题导致代码库混乱不堪

CommonJS

服务器端的 Node.js 遵循 CommonJS规范，该规范的核心思想是允许模块通过 `require` 方法来同步加载所要依赖的其他模块，然后通过 `exports` 或 `module.exports` 来导出需要暴露的接口。

```
require("module");
require("../file.js");
exports.doStuff = function() {};
module.exports = someValue;
```

优点：

- 服务器端模块便于重用
- NPM 中已经有将近20万个可以使用模块包
- 简单并容易使用

缺点：

- 同步的模块加载方式不适合在浏览器环境中，同步意味着阻塞加载，浏览器资源是异步加载的
- 不能非阻塞的并行加载多个模块

实现：

- 服务器端的 Node.js
- Browserify，浏览器端的 CommonJS 实现，可以使用 NPM 的模块，但是编译打包后的文件体积可能很大
- modules-webmake，类似Browserify，还不如 Browserify 灵活
- wreq，Browserify 的前身

AMD

Asynchronous Module Definition 规范其实只有一个主要接口 `define(id?, dependencies?, factory)`，它要在声明模块的时候指定所有的依赖 `dependencies`，并且还要当做形参传到 `factory` 中，对于依赖的模块提前执行，依赖前置。

```
define("module", ["dep1", "dep2"], function(d1, d2) {
  return someExportedValue;
});
require(["module", "../file"], function(module, file) { /* ... */ });
```

优点：

- 适合在浏览器环境中异步加载模块
- 可以并行加载多个模块

缺点：

- 提高了开发成本，代码的阅读和书写比较困难，模块定义方式的语义不顺畅
- 不符合通用的模块化思维方式，是一种妥协的实现

实现：

- RequireJS
- curl

CMD

Common Module Definition 规范和 AMD 很相似，尽量保持简单，并与 CommonJS 和 Node.js 的 Modules 规范保持了很大的兼容性。

```
define(function(require, exports, module) {
  var $ = require('jquery');
  var Spinning = require('./spinning');
  exports.doSomething = ...
  module.exports = ...
})
```

优点：

- 依赖就近，延迟执行
- 可以很容易在 Node.js 中运行

缺点：

- 依赖 SPM 打包，模块的加载逻辑偏重

实现：

- Sea.js
- coolie

UMD

Universal Module Definition 规范类似于兼容 CommonJS 和 AMD 的语法糖，是模块定义的跨平台解决方案。

ES6 模块

EcmaScript6 标准增加了 JavaScript 语言层面的模块体系定义。ES6 模块的设计思想，是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。CommonJS 和 AMD 模块，都只能在运行时确定这些东西。

```
import "jquery";
export function doStuff() {}
module "localModule" {}
```

优点：

- 容易进行静态分析
- 面向未来的 EcmaScript 标准

缺点：

- 原生浏览器端还没有实现该标准
- 全新的命令字，新版的 Node.js才支持

实现：

- Babel

期望的模块系统

可以兼容多种模块风格，尽量可以利用已有的代码，不仅仅只是 JavaScript 模块化，还有 CSS、图片、字体等资源也需要模块化。

前端模块加载

前端模块要在客户端中执行，所以他们需要增量加载到浏览器中。

模块的加载和传输，我们首先能想到两种极端的方式，一种是每个模块文件都单独请求，另一种是把所有模块打包成一个文件然后只请求一次。显而易见，每个模块都发起单独的请求造成了请求次数过多，导致应用启动速度慢；一次请求加载所有模块导致流量浪费、初始化过程慢。这两种方式都不是好的解决方案，它们过于简单粗暴。

分块传输，按需进行懒加载，在实际用到某些模块的时候再增量更新，才是较为合理的模块加载方案。

要实现模块的按需加载，就需要一个对整个代码库中的模块进行静态分析、编译打包的过程。

所有资源都是模块

在上面的分析过程中，我们提到的模块仅仅是指JavaScript模块文件。然而，在前端开发过程中还涉及到样式、图片、字体、HTML 模板等等众多的资源。这些资源还会以各种方言的形式存在，比如 coffeescript、 less、 sass、众多的模板库、多语言系统（i18n）等等。

如果他们都可以视作模块，并且都可以通过 `require` 的方式来加载，将带来优雅的开发体验，比如：

```
require("./style.css");
require("./style.less");
require("./template.jade");
require("./image.png");
```

那么如何做到让 `require` 能加载各种资源呢？

静态分析

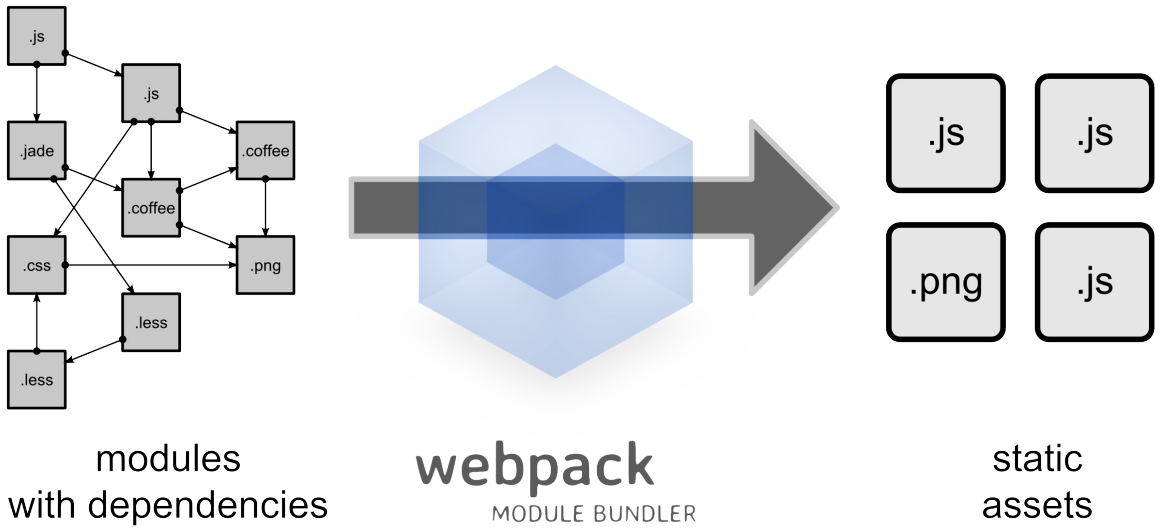
在编译的时候，要对整个代码进行静态分析，分析出各个模块的类型和它们依赖关系，然后将不同类型的模块提交给适配的加载器来处理。比如一个用 LESS 写的样式模块，可以先用 LESS 加载器将它转成一个CSS 模块，在通过 CSS 模块把他插入到页面的 `<style>` 标签中执行。Webpack 就是在这样的需求中应运而生。

同时，为了能利用已经存在的各种框架、库和已经写好的文件，我们还需要一个模块加载的兼容策略，来避免重写所有的模块。

那么接下来，让我们开始 Webpack 的神奇之旅吧。

什么是 Webpack

Webpack 是一个模块打包器。它将根据模块的依赖关系进行静态分析，然后将这些模块按照指定的规则生成对应的静态资源。



为什么重复造轮子

市面上已经存在的模块管理和打包工具并不适合大型的项目，尤其单页面 Web 应用程序。最紧迫的原因是如何在一个大规模的代码库中，维护各种模块资源的分割和存放，维护它们之间的依赖关系，并且无缝的将它们整合到一起生成适合浏览器端请求加载的静态资源。

这些已有的模块化工具并不能很好的完成如下的目标：

- 将依赖树拆分成按需加载的块
- 初始化加载的耗时尽量少
- 各种静态资源都可以视作模块
- 将第三方库整合成模块的能力
- 可以自定义打包逻辑的能力
- 适合大项目，无论是单页还是多页的 Web 应用

Webpack 的特点

Webpack 和其他模块化工具有什么区别呢？

代码拆分

Webpack 有两种组织模块依赖的方式，同步和异步。异步依赖作为分割点，形成一个新的块。在优化了依赖树后，每一个异步区块都作为一个文件被打包。

Loader

Webpack 本身只能处理原生的 JavaScript 模块，但是 loader 转换器可以将各种类型的资源转换成 JavaScript 模块。这样，任何资源都可以成为 Webpack 可以处理的模块。

智能解析

Webpack 有一个智能解析器，几乎可以处理任何第三方库，无论它们的模块形式是 CommonJS、 AMD 还是普通的 JS 文件。甚至在加载依赖的时候，允许使用动态表达式 `require("./templates/" + name + ".jade")` 。

插件系统

Webpack 还有一个功能丰富的插件系统。大多数内容功能都是基于这个插件系统运行的，还可以开发和使用开源的 Webpack 插件，来满足各式各样的需求。

快速运行

Webpack 使用异步 I/O 和多级缓存提高运行效率，这使得 Webpack 能够以令人难以置信的速度快速增量编译。

准备开始

我们通过具体案例来快速上手 Webpack。以下章节中的案例源码可以在 https://github.com/zhaoda/webpack-handbook/tree/master/examples/start 查看。

安装

首先要安装 Node.js， Node.js 自带了软件包管理器 npm，Webpack 需要 Node.js v0.6 以上支持，建议使用最新版 Node.js。

用 npm 安装 Webpack：

```
$ npm install webpack -g
```

此时 Webpack 已经安装到了全局环境下，可以通过命令行 `webpack -h` 试试。

通常我们会将 Webpack 安装到项目的依赖中，这样就可以使用项目本地版本的 Webpack。

```
# 进入项目目录
# 确定已经有 package.json，没有就通过 npm init 创建
# 安装 webpack 依赖
$ npm install webpack --save-dev
```

Webpack 目前有两个主版本，一个是在 master 主干的稳定版，一个是在 webpack-2 分支的测试版，测试版拥有一些实验性功能并且和稳定版不兼容，在正式项目中应该使用稳定版。

```
# 查看 webpack 版本信息
$ npm info webpack

# 安装指定版本的 webpack
$ npm install webpack@1.12.x --save-dev
```

如果需要使用 Webpack 开发工具，要单独安装：

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

使用

首先创建一个静态页面 index.html 和一个 JS 入口文件 entry.js：

```
<!-- index.html -->
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <script src="bundle.js"></script>
</body>
</html>
```

```
// entry.js
document.write('It works.')
```

然后编译 entry.js 并打包到 bundle.js：

```
$ webpack entry.js bundle.js
```

打包过程会显示日志：

```
Hash: e964f90ec65eb2c29bb9
Version: webpack 1.12.2
Time: 54ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.42 kB       0  [emitted]  main
   [0] ./entry.js 27 bytes {0} [built]
```

用浏览器打开 `index.html` 将会看到 `It works.` 。

接下来添加一个模块 `module.js` 并修改入口 `entry.js` ：

```
// module.js
module.exports = 'It works from module.js.'
```

```
// entry.js
document.write('It works.')
document.write(require('./module.js')) // 添加模块
```

重新打包 `webpack entry.js bundle.js` 后刷新页面看到变化 `It works.It works from module.js.`

```
Hash: 279c7601d5d08396e751
Version: webpack 1.12.2
Time: 63ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.57 kB       0  [emitted]  main
   [0] ./entry.js 66 bytes {0} [built]
   [1] ./module.js 43 bytes {0} [built]
```

Webpack 会分析入口文件，解析包含依赖关系的各个文件。这些文件（模块）都打包到 bundle.js 。Webpack 会给每个模块分配一个唯一的 id 并通过这个 id 索引和访问模块。在页面启动时，会先执行 entry.js 中的代码，其它模块会在运行 `require` 的时候再执行。

Loader

Webpack 本身只能处理 JavaScript 模块，如果要处理其他类型的文件，就需要使用 loader 进行转换。

Loader 可以理解为是模块和资源的转换器，它本身是一个函数，接受源文件作为参数，返回转换的结果。这样，我们就可以通过 `require` 来加载任何类型的模块或文件，比如 CoffeeScript、 JSX、 LESS 或图片。

先来看看 loader 有哪些特性？

- Loader 可以通过管道方式链式调用，每个 loader 可以把资源转换成任意格式并传递给下一个 loader ，但是最后一个 loader 必须返回 JavaScript。
- Loader 可以同步或异步执行。
- Loader 运行在 node.js 环境中，所以可以做任何可能的事情。
- Loader 可以接受参数，以此来传递配置项给 loader。
- Loader 可以通过文件扩展名（或正则表达式）绑定给不同类型的文件。
- Loader 可以通过 `npm` 发布和安装。
- 除了通过 `package.json` 的 `main` 指定，通常的模块也可以导出一个 loader 来使用。
- Loader 可以访问配置。
- 插件可以让 loader 拥有更多特性。
- Loader 可以分发出附加的任意文件。

Loader 本身也是运行在 node.js 环境中的 JavaScript 模块，它通常会返回一个函数。大多数情况下，我们通过 npm 来管理 loader，但是你也可以在项目中自己写 loader 模块。

按照惯例，而非必须，loader 一般以 `xxx-loader` 的方式命名， `xxx` 代表了这个 loader 要做的转换功能，比如 `json-loader`。

在引用 loader 的时候可以使用全名 `json-loader`，或者使用短名 `json`。这个命名规则和搜索优先级顺序在 webpack 的 `resolveLoader.moduleTemplates` api 中定义。

```
Default: ["*-webpack-loader", "*-web-loader", "*-loader", "*"]
```

Loader 可以在 `require()` 引用模块的时候添加，也可以在 webpack 全局配置中进行绑定，还可以通过命令行的方式使用。

接上一节的例子，我们要在页面中引入一个 CSS 文件 style.css，首页将 style.css 也看成是一个模块，然后用 `css-loader` 来读取它，再用 `style-loader` 把它插入到页面中。

```
/* style.css */
body { background: yellow; }
```

修改 entry.js：

```
require("!style!css!./style.css") // 载入 style.css
document.write('It works.')
document.write(require('./module.js'))
```

安装 loader：

```
npm install css-loader style-loader
```

重新编译打包，刷新页面，就可以看到黄色的页面背景了。

如果每次 `require` CSS 文件的时候都要写 loader 前缀，是一件很繁琐的事情。我们可以根据模块类型（扩展名）来自动绑定需要的 loader。

将 entry.js 中的 `require("!style!css!./style.css")` 修改为 `require("./style.css")` ，然后执行：

```
$ webpack entry.js bundle.js --module-bind 'css=style!css'
```

显然，这两种使用 loader 的方式，效果是一样的。

配置文件

Webpack 在执行的时候，除了在命令行传入参数，还可以通过指定的配置文件来执行。默认情况下，会搜索当前目录的 `webpack.config.js` 文件，这个文件是一个 node.js 模块，返回一个 json 格式的配置信息对象，或者通过 `--config` 选项来指定配置文件。

继续我们的案例，在根目录创建 `package.json` 来添加 webpack 需要的依赖：

```
{
  "name": "webpack-example",
  "version": "1.0.0",
  "description": "A simple webpack example.",
  "main": "bundle.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "webpack"
  ],
  "author": "zhaoda",
  "license": "MIT",
  "devDependencies": {
    "css-loader": "^0.21.0",
    "style-loader": "^0.13.0",
    "webpack": "^1.12.2"
  }
}
```

别忘了运行 `npm install` 。

然后创建一个配置文件 `webpack.config.js` ：

```
var webpack = require('webpack')

module.exports = {
  entry: './entry.js',
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {test: /\.css$/, loader: 'style!css'}
    ]
  }
}
```

同时简化 `entry.js` 中的 `style.css` 加载方式：

```
require('./style.css')
```

最后运行 `webpack` ，可以看到 webpack 通过配置文件执行的结果和上一章节通过命令行 `webpack entry.js bundle.js --module-bind 'css=style!css'` 执行的结果是一样的。

插件

插件可以完成更多 loader 不能完成的功能。

插件的使用一般是在 webpack 的配置信息 `plugins` 选项中指定。

Webpack 本身内置了一些常用的插件，还可以通过 npm 安装第三方插件。

接下来，我们利用一个最简单的 `BannerPlugin` 内置插件来实践插件的配置和运行，这个插件的作用是给输出的文件头部添加注释信息。

修改 `webpack.config.js`，添加 `plugins`：

```
var webpack = require('webpack')

module.exports = {
  entry: './entry.js',
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {test: /\.css$/, loader: 'style!css'}
    ]
  },
  plugins: [
    new webpack.BannerPlugin('This file is created by zhaoda')
  ]
}
```

然后运行 `webpack`，打开 `bundle.js`，可以看到文件头部出现了我们指定的注释信息：

```
/*! This file is created by zhaoda */
/******/ (function(modules) { // webpackBootstrap
/******/  // The module cache
/******/  var installedModules = {};
// 后面代码省略
```

开发环境

当项目逐渐变大，webpack 的编译时间会变长，可以通过参数让编译的输出内容带有进度和颜色。

```
$ webpack --progress --colors
```

如果不想每次修改模块后都重新编译，那么可以启动监听模式。开启监听模式后，没有变化的模块会在编译后缓存到内存中，而不会每次都被重新编译，所以监听模式的整体速度是很快的。

```
$ webpack --progress --colors --watch
```

当然，使用 `webpack-dev-server` 开发服务是一个更好的选择。它将在 localhost:8080 启动一个 express 静态资源 web 服务器，并且会以监听模式自动运行 webpack，在浏览器打开 http://localhost:8080/ 或 http://localhost:8080/webpack-dev-server/ 可以浏览项目中的页面和编译后的资源输出，并且通过一个 socket.io 服务实时监听它们的变化并自动刷新页面。

```
# 安装
$ npm install webpack-dev-server -g

# 运行
$ webpack-dev-server --progress --colors
```

故障处理

Webpack 的配置比较复杂，很容出现错误，下面是一些通常的故障处理手段。

一般情况下，webpack 如果出问题，会打印一些简单的错误信息，比如模块没有找到。我们还可以通过参数 `--display-error-details` 来打印错误详情。

```
$ webpack --display-error-details

Hash: a40fbc6d852c51fceadb
Version: webpack 1.12.2
Time: 586ms
    Asset     Size  Chunks             Chunk Names
bundle.js  12.1 kB       0  [emitted]  main
   [0] ./entry.js 153 bytes {0} [built] [1 error]
   [5] ./module.js 43 bytes {0} [built]
    + 4 hidden modules

ERROR in ./entry.js
Module not found: Error: Cannot resolve 'file' or 'directory' ./badpathmodule in /Users/z
resolve file
  /Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule doesn't exist
  /Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule.webpack.js doesn't
  /Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule.js doesn't exist
  /Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule.web.js doesn't exis
  /Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule.json doesn't exist
resolve directory
  /Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule doesn't exist (dire
  /Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule/package.json doesn'
[/Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule]
[/Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule.webpack.js]
[/Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule.js]
[/Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule.web.js]
[/Users/zhaoda/data/projects/webpack-handbook/examples/badpathmodule.json]
 @ ./entry.js 3:0-26
```

Webpack 的配置提供了 `resolve` 和 `resolveLoader` 参数来设置模块解析的处理细节，`resolve` 用来配置应用层的模块（要被打包的模块）解析，`resolveLoader` 用来配置 `loader` 模块的解析。

当引入通过 npm 安装的 node.js 模块时，可能出现找不到依赖的错误。Node.js 模块的依赖解析算法很简单，是通过查看模块的每一层父目录中的 `node_modules` 文件夹来查询依赖的。当出现 Node.js 模块依赖查找失败的时候，可以尝试设置 `resolve.fallback` 和 `resolveLoader.fallback` 来解决问题。

```
module.exports = {
  resolve: { fallback: path.join(__dirname, "node_modules") },
  resolveLoader: { fallback: path.join(__dirname, "node_modules") }
};
```

Webpack 中涉及路径配置最好使用绝对路径，建议通过 `path.resolve(__dirname, "app/folder")` 或 `path.join(__dirname, "app", "folder")` 的方式来配置，以兼容 Windows 环境。

高级

以下章节是进阶内容。

CommonJS 规范

CommonJS 是以在浏览器环境之外构建 JavaScript 生态系统为目标而产生的项目，比如在服务器和桌面环境中。

这个项目最开始是由 Mozilla 的工程师 Kevin Dangoor 在2009年1月创建的，当时的名字是 ServerJS。

> 我在这里描述的并不是一个技术问题，而是一件重大的事情，让大家走到一起来做决定，迈出第一步，来建立一个更大更酷的东西。 —— Kevin Dangoor's What Server Side JavaScript needs

2009年8月，这个项目改名为 CommonJS，以显示其 API 的更广泛实用性。CommonJS 是一套规范，它的创建和核准是开放的。这个规范已经有很多版本和具体实现。CommonJS 并不是属于 ECMAScript TC39 小组的工作，但 TC39 中的一些成员参与 CommonJS 的制定。2013年5月，Node.js 的包管理器 NPM 的作者 Isaac Z. Schlueter 说 CommonJS 已经过时，Node.js 的内核开发者已经废弃了该规范。

CommonJS 规范是为了解决 JavaScript 的作用域问题而定义的模块形式，可以使每个模块它自身的命名空间中执行。该规范的主要内容是，模块必须通过 `module.exports` 导出对外的变量或接口，通过 `require()` 来导入其他模块的输出到当前模块作用域中。

一个直观的例子：

```
// moduleA.js
module.exports = function( value ){
    return value * 2;
}
```

```
// moduleB.js
var multiplyBy2 = require('./moduleA');
var result = multiplyBy2(4);
```

CommonJS 是同步加载模块，但其实也有浏览器端的实现，其原理是现将所有模块都定义好并通过 `id` 索引，这样就可以方便的在浏览器环境中解析了，可以参考 require1k 和 tiny-browser-require 的源码来理解其解析（resolve）的过程。

更多关于 CommonJS 规范的内容请查看 http://wiki.commonjs.org/wiki/CommonJS。

AMD 规范

AMD（异步模块定义）是为浏览器环境设计的，因为 CommonJS 模块系统是同步加载的，当前浏览器环境还没有准备好同步加载模块的条件。

AMD 定义了一套 JavaScript 模块依赖异步加载标准，来解决同步加载的问题。

模块通过 `define` 函数定义在闭包中，格式如下：

```
define(id?: String, dependencies?: String[], factory: Function|Object);
```

`id` 是模块的名字，它是可选的参数。

`dependencies` 指定了所要依赖的模块列表，它是一个数组，也是可选的参数，每个依赖的模块的输出将作为参数一次传入 `factory` 中。如果没有指定 `dependencies`，那么它的默认值是 `["require", "exports", "module"]`。

```
define(function(require, exports, module) {})
```

`factory` 是最后一个参数，它包裹了模块的具体实现，它是一个函数或者对象。如果是函数，那么它的返回值就是模块的输出接口或值。

一些用例：

定义一个名为 `myModule` 的模块，它依赖 `jQuery` 模块：

```
define('myModule', ['jquery'], function($) {
    // $ 是 jquery 模块的输出
    $('body').text('hello world');
});
// 使用
define(['myModule'], function(myModule) {});
```

注意：在 webpack 中，模块名只有局部作用域，在 Require.js 中模块名是全局作用域，可以在全局引用。

定义一个没有 `id` 值的匿名模块，通常作为应用的启动函数：

```
define(['jquery'], function($) {
    $('body').text('hello world');
});
```

依赖多个模块的定义：

```
define(['jquery', './math.js'], function($, math) {
    // $ 和 math 一次传入 factory
    $('body').text('hello world');
});
```

模块输出：

```
define(['jquery'], function($) {

    var HelloWorldize = function(selector){
        $(selector).text('hello world');
    };

    // HelloWorldize 是该模块输出的对外接口
    return HelloWorldize;
});
```

在模块定义内部引用依赖：

```
define(function(require) {
    var $ = require('jquery');
    $('body').text('hello world');
});
```

模块规范

- CommonJS 规范
- Asynchronous Module Definition
- Common Module Definition
- CMD 模块定义规范
- Universal Module Definition
- ECMAScript 6 Module
- 什么是 AMD、CommonJS、UMD
- 关于 CommonJS AMD CMD UMD
- 为什么我推荐 requirejs 而不是 seajs
- AMD 和 CMD 的区别有哪些
- 前端模块化开发的价值
- What Server Side JavaScript needs

模块系统

- RequireJS
- curl
- Sea.js
- coolie
- Browserify
- modules-webmake
- wreq

Webpack

- Webpack 官方文档
- React Webpack cookbook

编译

- Babel