

Webpack学习笔记

作者：韩涛

[第一章 webpack介绍 3](#_Toc772189119)

[前言 3](#_Toc1520179276)

[模块系统的演进 3](#_Toc1215166002)

[<script>标签 3](#_Toc1128411083)

[CommonJS 4](#_Toc560864132)

[AMD 4](#_Toc1677578350)

[CMD 5](#_Toc1825158044)

[UMD 6](#_Toc924851377)

[ES6模块 6](#_Toc1893845981)

[期望的模块系统 6](#_Toc855279645)

[前端模块加载 6](#_Toc46520305)

[所有资源都是模块 7](#_Toc800363462)

[静态分析 7](#_Toc2127155195)

[第二章 Webpack之旅 7](#_Toc361008311)

[什么是Webpack 7](#_Toc868537392)

[为什么重复造轮子 8](#_Toc241295487)

[Webpack的特点 8](#_Toc743658203)

[代码拆分 8](#_Toc440372192)

[Loader 9](#_Toc1122366675)

[智能解析 9](#_Toc239321757)

[插件系统 9](#_Toc1479364691)

[快速运行 9](#_Toc1846389611)

# webpack介绍

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。

## 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。

前端开发的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 中已经有将尽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。

# Webpack之旅

## 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

Webapck 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。

## 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(注意package.json里的name字段不能与webpack重名)。

$ 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

### Webpack命令行

安装webpack后，可以使用webpack这个命令行工具。主要命令：

webpack <entry> <output>。

可以切换到包含webpack.config.js的目录运行命令：

* webpack 执行一次开发时的编译
* webpack -p 执行一次生成环境的编译（压缩）
* webpack --watch 在开发时持续监控增量编译（很快）
* webpack -d 让他生成SourceMaps
* webpack --progress 显示编译进度
* webpack --colors 显示静态资源的颜色
* webpack --sort-modules-by, --sort-chunks-by, --sort-assets-by 将modules/chunks/assets进行列表排序
* webpack --display-chunks 展示编译后的分块
* webpack --display-reasons 显示更多引用模块原因
* webapck --display-error-details 显示更多报错信息

### webpack.config.js

每个项目下都必须配置有一个webpack.config.js，就是一个配置项，告诉 webpack 它需要做什么。  
下面看一个简单的示例：

var webpack = require('webpack');

module.exports = {

//插件项

plugins: [

//提公用js到common.js文件中

new webpack.optimize.CommonsChunkPlugin('common.js'), //将样式统一发布到style.css中

new ExtractTextPlugin("style.css", { allChunks: true, disable: false }), //使用ProvidePlugin加载使用频率高的模块

new webpack.ProvidePlugin({ $: "webpack-zepto" })

],

//页面入口文件配置

entry: { index : './src/main.js' }, //入口文件输出配置

output: { path: \_\_dirname +'/dist/', filename: '[name].js' },

module: {

//加载器配置

loaders: [

{ test: /\.css$/, loader: 'style-loader!css-loader' },

{ test: /\.scss$/, loader: 'style!css!sass?sourceMap'},

{ test: /\.(png|jpg)$/, loader: 'url-loader?limit=8192'}

]

},

//其它解决方案配置

resolve: {

extensions: ['', '.js', '.json', '.scss'],

alias: { filter: path.join(\_\_dirname, 'src/filters')

}

}

};

**entry**是页面入口文件配置，可以是一个文件或者多个入口文件，可以是对象格式或者数组格式。

对象形式：entry: { index : './src/main.js' }

数组形式：entry:['./src/main.js','./src/index.js']

#### output **output**是对应输出项配置,主要包括path,filename和publishPath属性。path代表输出的路径，filename代表输出的文件名称，publishPath代表静态资源发布后的前缀地址。

output: { path: \_\_dirname +'/dist/', filename: '[name].js' },

**[module.loaders](http://webpack.github.io/docs/using-loaders.html)**是最关键的一块配置。它告知 webpack 每一种文件都需要使用什么加载器来处理。

module: {

//加载器配置

loaders: [

//.css 文件使用 style-loader 和 css-loader 来处理

{ test: /\.css$/, loader: 'style-loader!css-loader' },

//.scss 文件使用 style-loader、css-loader 和 sass-loader 来编译处理

{ test: /\.scss$/, loader: 'style!css!sass?sourceMap'},

//图片文件使用 url-loader 来处理，小于8kb的直接转为base64

{ test: /\.(png|jpg)$/, loader: 'url-loader?limit=8192'} ]

}

**loader**主要有3种使用方式：

1、在页面里面引用资源使用

require("url-loader?mimetype=image/png!./file.png");

2、在webpack.config.js文件夹中使用

{ test: /.png$/, loader: "url?mimetype=image/png" };

3、在命令行中编译使用

webpack --module-bind "png=url-loader?mimetype=image/png";

如上，"-loader"其实是可以省略不写的，多个loader之间用“!”连接起来。  
注意所有的加载器都需要通过 npm 来加载，并建议查阅它们对应的 readme 来看看如何使用。  
拿最后一个 url-loader 来说，它会将样式中引用到的图片转为模块来处理，使用该加载器需要先进行安装：  
 npm install url-loader -save-dev  
配置信息的参数“?limit=8192”表示将所有小于8kb的图片都转为base64形式（其实应该说超过8kb的才使用 url-loader 来映射到文件，否则转为data url形式）。也可以使用file-loader来加载资源文件。

**[plugins](http://webpack.github.io/docs/using-plugins.html)**是插件项，这里我们使用了一个 CommonsChunkPlugin 的插件，它用于提取多个入口文件的公共脚本部分，然后生成一个 common.js 来方便多页面之间进行复用。

plugins: [

//提公用js到common.js文件中

new webpack.optimize.CommonsChunkPlugin('common.js'),

//将样式统一发布到style.css中

new ExtractTextPlugin("style.css", { allChunks: true, disable: false }),

//使用ProvidePlugin加载使用频率高的模块

new webpack.ProvidePlugin({ $: "webpack-zepto" })

]

**resolve** 配置，这块很好理解，直接写注释了：

resolve: {

// require时省略的扩展名，如：require('module') 不需要module.js

extension: ['', '.js'],

//别名

alias: {

filter: path.join(\_\_dirname, 'src/filters')

}

}