|  |
| --- |
| Webpack 学习笔记 　　　　　　　　　　　　　　　　　　　作者：韩涛 |



[第一章 webpack介绍 3](#_Toc1065144906)

# 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。