# webpack配置

1. **安装**

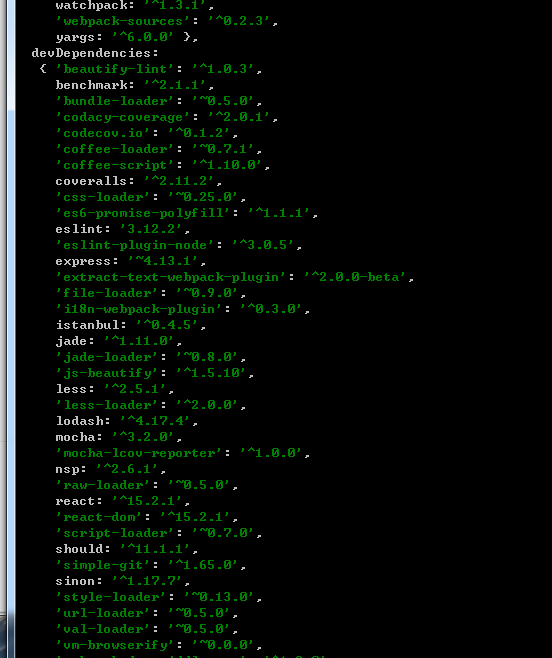
安装nodejs,运行命令npm intall wepack –g 或者npm install webpack –save-dev进行全局和本地安装。

安装指定版本，运行命令npm install webpack@1.12.x --save-dev

如果需要使用webpack开发工具，请自行安装npm install webpack-dev-server --save-dev

1. **查看版本信息**

运行 npm info webpack，即可查看webpack基本信息



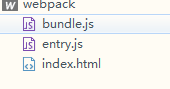
1. **使用**

1. 首先创建一个静态文件index.html,在静态文件里面引入bundle.js

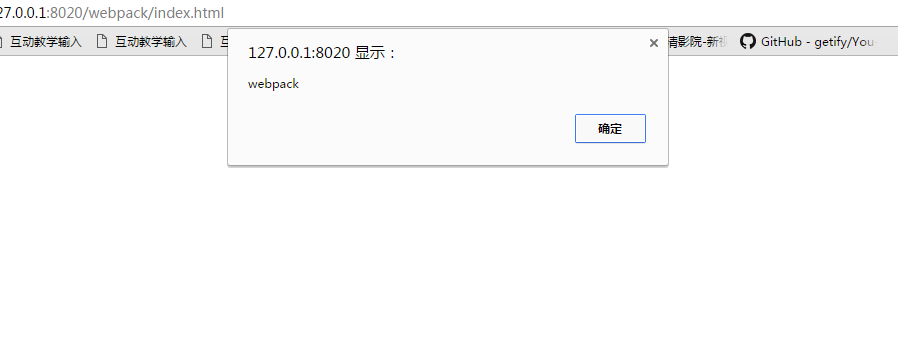
2. 创建一个entry.js文件，写一个alert(‘webpack’);

3. 运行命令 webpack entry.js bundle.js

4. 这样，bundle.js就可以生成了



5. 运行index.html

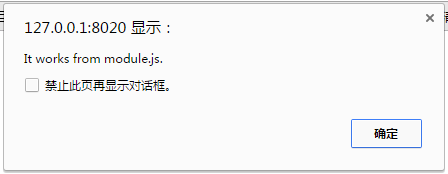


6. 添加一个模块，并修改entry.js

1. 新建module.js,写入module.exports = 'It works from module.js.'

2. entry.js添加如下代码alert(require('./module.js'))

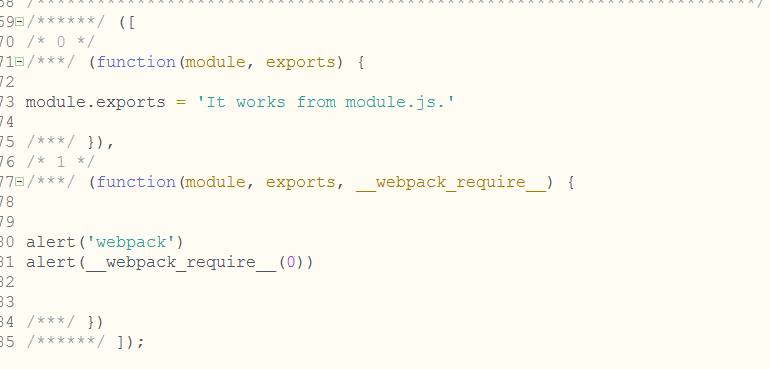
3. 重新编译bundle.js,运行index.html



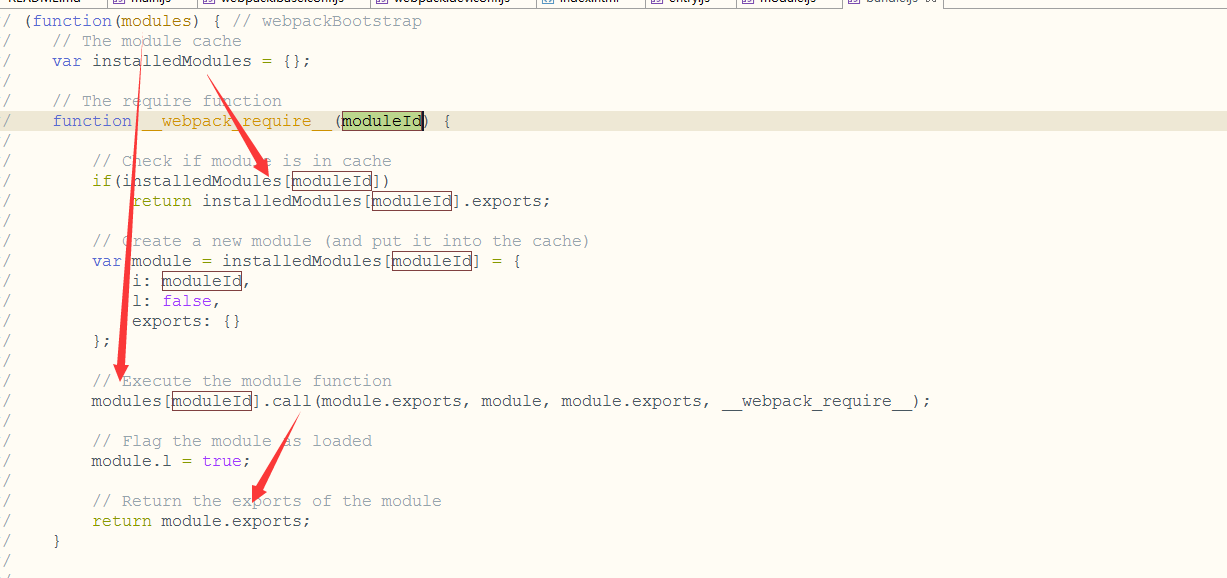
1. **原理：**

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。

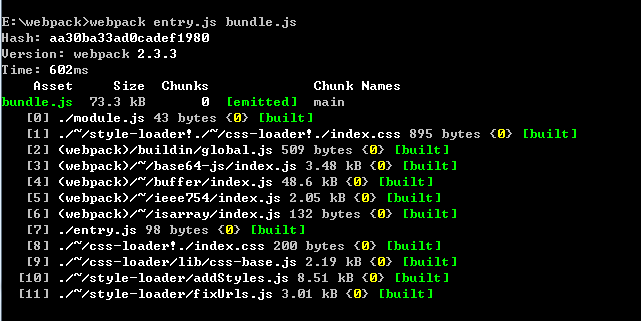
编译后文件如下图

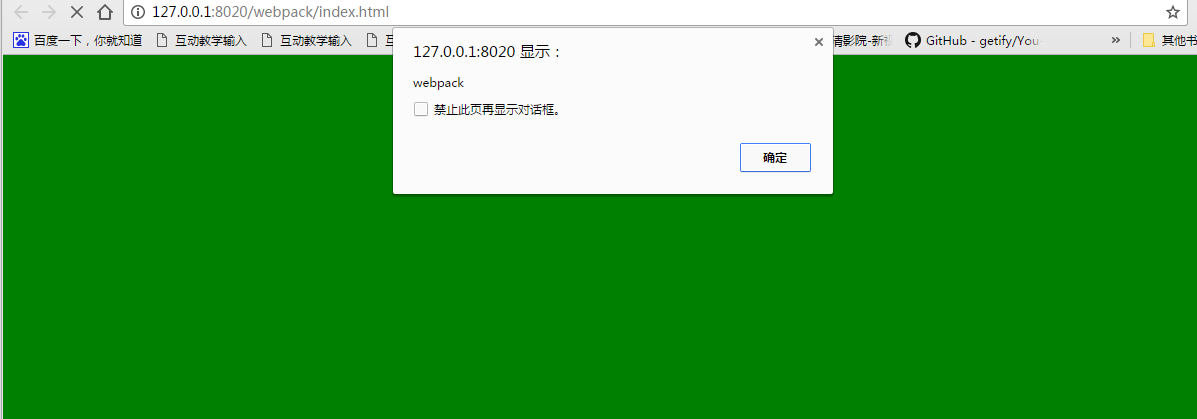




  
（通过对象的赋值和引用，形成一个调用关系链）

1. **Loader**
2. 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
3. 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。
4. 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。正是由于loader运行在node环境里面，所以，loader读取文件变得轻而易举。
5. 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", "\*"]
6. 安装 loader
7. 运行npm intall css-loader style-loader
8. 新建index.css文件，指定背景色为green
9. 修改entry.js文件require("!style-loader!css-loader!./style.css")，载入style-loader css-loader
10. 重新打包编译bundle.js,运行index.html





1. 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。

1. 将 entry.js 中的 require("!style!css!./style.css") 修改为 require("./style.css")，2. 运行$webpack entry.js bundle.js --module-bind 'css=style-loader!css-loader'或者webpack entry.js bundle.js --module-bind "css=style-loader!css-loader"，效果是一样的

**6. 配置文件**

1. 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。
2. 通过npm init，我们来创建package的基本信息

{

"name": "webpack",

"version": "1.0.0",

"description": "'webapck demo'",

"main": "bundle.js",

"dependencies": {

"css-loader": "^0.28.0",

"style-loader": "^0.16.1"

},

"devDependencies": {},

"scripts": {

"test": "'test'"

},

"keywords": [

"'webpack'"

],

"author": "djl",

"license": "ISC"

}

1. 如上图所示，为安装初始化后的配置信息，我们看到，没有webpck的dependencies依赖，是因为我们安装的webpack为全局的，通过wbpack -v我们来查看一下webpack的版本信息，添加本地环境依赖



{

"name": "webpack",

"version": "1.0.0",

"description": "'webapck demo'",

"main": "bundle.js",

"dependencies": {

"css-loader": "^0.28.0",

"style-loader": "^0.16.1",

"webpack": "^2.3.3"

},

"devDependencies": {},

"scripts": {

"test": "'test'"

},

"keywords": [

"'webpack'"

],

"author": "djl",

"license": "ISC"

}

1. 运行 npm install 即可安装本地webpack
2. 创建webpack.config.js文件，放置到根目录下，写入如下信息

**var webpack = require('webpack') //引入webpack模块**

**module.exports = {//webpack导出模块**

**entry: './entry.js',//入口文件，也就是我们的entry.js文件，只有这一个**

**output: {//输出目录配置**

**path: \_\_dirname, //当前目录为导出文件，如果对node不太熟悉，自行脑补**

**filename: 'bundle.js'//导出文件名**

**},**

**module: {**

**loaders: [//webpack的loader模块配置**

**{test: /\.css$/, loader: 'style-loader!css-loader'}//test为正则匹配css文件，loader为loader名称**

**]**

**}**

**}**

1. 最后entry.js我们只需要require(‘./index.css’),重新运行编译bundle.js,即可

**7. 插件**

1. 插件可以完成更多 loader 不能完成的功能。

2．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

3. 我们以BannerPluigin为例，来给输出文件添加注释信息

4. 修改webpack.config.js文件，在modlue对象下方添加插件如下

module: {

loaders: [

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

]

},

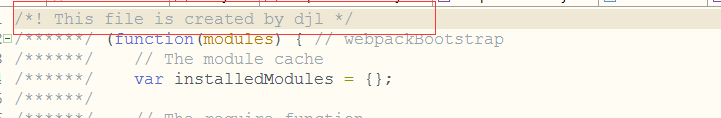
plugins: [

new webpack.BannerPlugin('This file is created by djl')

]

}

5. 重新编译bundle.js，打开文件，我们已经看到注释信息了



**8.开发环境**

1. 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

2. 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

3. 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。

**安装服务工具npm install webpack-dev-server –g 运行webpack-dev-server --progress –colors**

**9. webpack补充说明**

**1.entry**

entry可以是个字符串或数组或者是对象。

当entry是个**字符串**的时候，用来定义入口文件：

entry: './js/main.js'

当entry是个**数组**的时候，里面同样包含入口js文件，另外一个参数可以是用来配置webpack提供的一个静态资源服务器，webpack-dev-server。webpack-dev-server会监控项目中每一个文件的变化，实时的进行构建，并且自动刷新页面：

entry: [

'webpack/hot/only-dev-server',

'./js/app.js'

]

当entry是个**对象**的时候，我们可以将不同的文件构建成不同的文件，按需使用，比如在我的hello页面中只要\引入hello.js即可：

entry: {

hello: './js/hello.js',

form: './js/form.js'

}

**2.output**

output参数是个对象，用于定义构建后的文件的输出。其中包含path和filename：

当我们在entry中定义构建多个文件时，filename可以对应的更改为[name].js用于定义不同文件构建后的名字。

publicPath：publicPath”项则被许多Webpack的插件用于在生产模式下更新内嵌到css、html文件里的url值。

path：“path”仅仅告诉Webpack结果存储在哪里

**3. resolve**

webpack在构建包的时候会按目录的进行文件的查找，resolve属性中的extensions数组中用于配置程序可以自行补全哪些文件后缀：

resolve:{

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

}

**4. externals**

当我们想在项目中require一些其他的类库或者API，而又不想让这些类库的源码被构建到运行时文件中，这在实际开发中很有必要。此时我们就可以通过配置externals参数来解决这个问题：

externals: {

"jquery": "jQuery"

}

**5. loader**

**loader的参数项随插件的不同而不同，具体请参考具体的插件设置**