**概述**

Webpack是一款用户打包前端模块的工具。主要是用来打包在浏览器端使用的javascript的。同时也能转换、捆绑、打包其他的静态资源，包括css、image、font file、template等。个人认为它的优点就是易用，而且常用功能基本都有,另外可以通过自己开发loader和plugin来满足自己的需求。这里就尽量详细的来介绍下一些基本功能的使用。

**安装**

npm install webpack

**运行webpack**

webpack需要编写一个config文件，然后根据这个文件来执行需要的打包功能。我们现在来编写一个最简单的config。新建一个文件，命名为webpack-config.js。config文件实际上就是一个Commonjs的模块。内容如下：

var webpack = require('webpack');

var path = require('path');

var buildPath = path.resolve(\_\_dirname,"build");

var nodemodulesPath = path.resolve(\_\_dirname,'node\_modules');

var config = {

//入口文件配置

entry:path.resolve(\_\_dirname,'src/main.js'),

resolve:{

extentions:["","js"]//当requrie的模块找不到时，添加这些后缀

},

//文件导出的配置

output:{

path:buildPath,

filename:"app.js"

}

}

module.exports = config;

我的目录结构是这样的：

webpack

|---index.html

|---webpack-config.js

|---src

|---main.js

|---js

|---a.js

main.js文件内容如下：

var a = require('./js/a');

a();

console.log('hello world');

document.getElementById("container").innerHTML = "<p>hello world</p>";

a.js文件内容如下：

module.exports = function(){

console.log('it is a ');

}

然后我们执行如下的命令：

webpack --config webpack-config.js --colors

这样我们就能在目录里面看到一个新生成的目录build，目录结构如下：

webpack

|---index.html

|---webpack-config.js

|---build

|---app.js

然后引用app.js就Ok啦。main.js和模块a.js的内容就都打包到app.js中了。这就演示了一个最简单的把模块的js打包到一个文件的过程了。

**介绍webpack config文件**

webpack是根据config里面描述的内容对一个项目进行打包的。接着我们来解释下config文件中的节点分别代表什么意思。一个config文件，基本都是由以下几个配置项组成的。

**entry**

配置要打包的文件的入口;可以配置多个入口文件，下面会有介绍。

**resolve**

配置文件后缀名(extensions)，除了js，还有jsx、coffee等等。alias配置项，可以为常用模块配置改属性，可以节省编译的搜索时间。例如：

resolve:{

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

alias:{

'react':path.join(nodeModulesPath,'react/react.js')

}

}

除了这个功能还可以配置其他有用的功能，由于我还不完全了解，有知道的朋友欢迎指教。

**output**

配置输出文件的路径，文件名等。

**module(loaders)**

配置要使用的loader。对文件进行一些相应的处理。比如babel-loader可以把es6的文件转换成es5。大部分的对文件的处理的功能都是通过loader实现的。loader就相当于gulp里的task。loader可以用来处理在入口文件中require的和其他方式引用进来的文件。loader一般是一个独立的node模块，要单独安装。

loader配置项：

test: /\.(js|jsx)$/, //注意是正则表达式，不要加引号，匹配要处理的文件

loader: 'eslint-loader', //要使用的loader，"-loader"可以省略

include: [path.resolve(\_\_dirname, "src/app")], //把要处理的目录包括进来

exclude: [nodeModulesPath] //排除不处理的目录

目前已有的loader列表：<https://webpack.github.io/docs/list-of-loaders.html>

一个module的例子:

module: {

preLoaders: [

{

test: /\.(js|jsx)$/,

loader: 'eslint-loader',

include: [path.resolve(\_\_dirname, "src/app")],

exclude: [nodeModulesPath]

},

],

loaders: [

{

test: /\.(js|jsx)$/, //正则表达式匹配 .js 和 .jsx 文件

loader: 'babel-loader?optional=runtime&stage=0',//对匹配的文件进行处理的loader

exclude: [nodeModulesPath]//排除node module中的文件

}

]

}

**plugins**

顾名思义，就是配置要使用的插件。plugin是比loader功能更强大的插件，能使用更多的wepack api。来看一个使用plugin的例子：

plugins: [

//压缩打包的文件

new webpack.optimize.UglifyJsPlugin({

compress: {

//supresses warnings, usually from module minification

warnings: false

}

}),

//允许错误不打断程序

new webpack.NoErrorsPlugin(),

//把指定文件夹xia的文件复制到指定的目录

new TransferWebpackPlugin([

{from: 'www'}

], path.resolve(\_\_dirname,"src"))

]

目前已有的plugins列表：<http://webpack.github.io/docs/list-of-plugins.html>

**如何压缩输出的文件**

plugins: [

//压缩打包的文件

new webpack.optimize.UglifyJsPlugin({

compress: {

//supresses warnings, usually from module minification

warnings: false

}

})]

**如何copy目录下的文件到输出目录**

copy文件需要通过插件"transfer-webpack-plugin"来完成。

安装：

npm install transfer-webpack-plugin -save

配置:

var TransferWebpackPlugin = require('transfer-webpack-plugin');

//其他节点省略

plugins: [

//把指定文件夹下的文件复制到指定的目录

new TransferWebpackPlugin([

{from: 'www'}

], path.resolve(\_\_dirname,"src"))

]

**打包javascript模块**

支持的js模块化方案包括：

* ES6 模块

import MyModule from './MyModule.js';

* CommonJS

var MyModule = require('./MyModule.js');

* AMD
* define(['./MyModule.js'], function (MyModule) {

});

上面已经演示了打包js模块,这里不再重复。ES6的模块需要配置babel-loader来先把处理一下js文件。下面展示下打包ES模块的配置文件:

var webpack = require('webpack');

var path = require('path');

var buildPath = path.resolve(\_\_dirname, 'build');

var nodeModulesPath = path.resolve(\_\_dirname, 'node\_modules');

var TransferWebpackPlugin = require('transfer-webpack-plugin');

var config = {

entry: [path.join(\_\_dirname, 'src/main.js')],

resolve: {

extensions: ["", ".js", ".jsx"]

//node\_modules: ["web\_modules", "node\_modules"] (Default Settings)

},

output: {

path: buildPath,

filename: 'app.js'

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compress: {

warnings: false

}

}),

new webpack.NoErrorsPlugin(),

new TransferWebpackPlugin([

{from: 'www'}

], path.resolve(\_\_dirname,"src"))

],

module: {

preLoaders: [

{

test: /\.(js|jsx)$/,

loader: 'eslint-loader',

include: [path.resolve(\_\_dirname, "src/app")],

exclude: [nodeModulesPath]

},

],

loaders: [

{

test: /\.js$/, //注意是正则表达式，不要加引号

loader: 'babel-loader?optional=runtime&stage=0',//babel模块相关的功能请自查，这里不做介绍

exclude: [nodeModulesPath]

}

]

},

//Eslint config

eslint: {

configFile: '.eslintrc' //Rules for eslint

},

};

module.exports = config;

**打包静态资源**

**css/sass/less**

安装css-loader和style-loader

npm install css-loader --save -dev

npm install style-loader --save -dev

config配置：

var config = {

entry:path.resolve(\_\_dirname,'src/main.js'),

resolve:{

extentions:["","js"]

},

output:{

path:buildPath,

filename:"app.js"

},

module:{

loaders:[{

test:/\.css$/,

loader:'style!css',

exclude:nodemodulesPath

}]

}

}

style-loader会把css文件嵌入到html的style标签里，css-loader会把css按字符串导出，这两个基本都是组合使用的。打包完成的文件，引用执行后，会发现css的内容都插入到了head里的一个style标签里。如果是sass或less配置方式与上面类似。

**images**

可以通过url-loader把较小的图片转换成base64的字符串内嵌在生成的文件里。安装：

npm install url-loader --save -dev

config配置:

var config = {

entry:path.resolve(\_\_dirname,'src/main.js'),

resolve:{

extentions:["","js"]

},

output:{

path:buildPath,

filename:"app.js"

},

module:{

loaders:[{

test:/\.css$/,

loader:'style!css',//

exclude:nodemodulesPath

},

{ test:/\.png$/,loader:'url-loader?limit=10000'}//限制大小小于10k的

]

}

}

css文件内容：

#container{

color: #f00;

background:url(images/logo-201305.png);

/\*生成完图片会被处理成base64的字符串 注意：不要写'/images/logo-201305.png',否则图片不被处理\*/

}

**iconfont**

内嵌iconfont的使用方法其实和上述处理png图片的方法一致。通过url-loader来处理。

config配置：

var config = {

entry:path.resolve(\_\_dirname,'src/main.js'),

resolve:{

extentions:["","js"]

},

output:{

path:buildPath,

filename:"app.js"

},

module:{

loaders:[{

test:/\.css$/,

loader:'style!css',//

exclude:nodemodulesPath

},

{ test:/\.(png|woff|svg|ttf|eot)$/,loader:'url-loader?limit=10000'}//限制大小小于10k的

]

}

}

css文件内容：

@font-face {font-family: 'iconfont';

src: url('fonts/iconfont.eot'); /\* IE9\*/

src: url('fonts/iconfont.eot?#iefix') format('embedded-opentype'), /\* IE6-IE8 \*/

url('fonts/iconfont.woff') format('woff'), /\* chrome、firefox \*/

url('fonts/iconfont.ttf') format('truetype'), /\* chrome、firefox、opera、Safari, Android, iOS 4.2+\*/

url('fonts/iconfont.svg#iconfont') format('svg'); /\* iOS 4.1- \*/

}

执行打包后会把字体文件都转换成base64字符串内容到文件里.这里有个头疼的问题，就是每个浏览器支持的字体格式不一样，由于把全部格式的字体打包进去，造成不必要的资源浪费。

**打包template**

我一大包handlebars的模块为例，来演示下打包模块的过程。有的模板对应的loader，有可能没有现车的，恐怕要自己实现loader。

先安装必须的node模块

npm install handlebars-loader --save -dev

npm install handlebars -save//是必须的

config配置:

var config = {

entry:path.resolve(\_\_dirname,'src/main.js'),

resolve:{

extentions:["","js"]

},

output:{

path:buildPath,

filename:"app.js"

},

module:{

loaders:[

{ test: /\.html$/, loader: "handlebars-loader" }

]

}

}

新建一个模板文件tb.html，目录结构：

webpack

|---index.html

|---webpack-config.js

|---src

|---template

| |---tb.html

|---main.js

main.js中调用模块的代码如下：

var template = require("./template/tp.html");

var data={say\_hello:"it is handlebars"};

var html = template(data);

document.getElementById('tmpl\_container').innerHTML = html;

**公用的模块分开打包**

这需要通过插件“CommonsChunkPlugin”来实现。这个插件不需要安装，因为webpack已经把他包含进去了。接着我们来看配置文件：

var config = {

entry:{app:path.resolve(\_\_dirname,'src/main.js'),

vendor: ["./src/js/common"]},//【1】注意这里

resolve:{

extentions:["","js"]

},

output:{

path:buildPath,

filename:"app.js"

},

module:{

loaders:[{

test:/\.css$/,

loader:'style!css',

exclude:nodemodulesPath

}

]

},

plugins:[

new webpack.optimize.UglifyJsPlugin({

compress: {

warnings: false

}

}),

//【2】注意这里 这两个地方市用来配置common.js模块单独打包的

new webpack.optimize.CommonsChunkPlugin({

name: "vendor",//和上面配置的入口对应

filename: "vendor.js"//导出的文件的名称

})

]

}

目录结构现在是这样的：

webpack

|---index.html

|---webpack-config.js

|---src

|---main.js

|---js

|---a.js //a里面require了common

|---common.js

执行webpack会生成app.js和common.js两个文件.

**多个入口**

config配置：

var config = {

entry:{

m1:path.resolve(\_\_dirname,'src/main.js'),

m2:path.resolve(\_\_dirname,'src/main1.js')

},//注意在这里添加文件的入口

resolve:{

extentions:["","js"]

},

output:{

path:buildPath,

filename:"[name].js"//注意这里使用了name变量

}

}

**webpack-dev-server**

在开发的过程中个，我们肯定不希望，每次修改完都手动执行webpack命令来调试程序。所以我们可以用webpack-dev-server这个模块来取代烦人的执行命令。它会监听文件，在文件修改后，自动编译、刷新浏览器的页面。另外，编译的结果是保存在内存中的，而不是实体的文件，所以是看不到的，因为这样会编译的更快。它就想到与一个轻量的express服务器。安装：

npm install webpack-dev-server --save -dev

config配置：

var config = {

entry:path.resolve(\_\_dirname,'src/main.js'),

resolve:{

extentions:["","js"]

},

//Server Configuration options

devServer:{

contentBase: '', //静态资源的目录 相对路径,相对于当前路径 默认为当前config所在的目录

devtool: 'eval',

hot: true, //自动刷新

inline: true,

port: 3005

},

devtool: 'eval',

output:{

path:buildPath,

filename:"app.js"

},

plugins: [

new webpack.HotModuleReplacementPlugin(),//这个好像也是必须的，虽然我还没搞懂它的作用

new webpack.NoErrorsPlugin()

]

}

我的目录结构:

webpack

|---index.html

|---webpack-config.js//我把静态资源目录配置在了这里

|---src

|---main.js

|---js

|---a.js

|---common.js

执行命令：

webpack-dev-server --config webpack-dev-config.js --inline --colors

默认访问地址：

http://localhost:3000/index.html(根据配置会不一样)

有一点需要声明，在index.html（引用导出结果的html文件）里直接引用“app.js”，不要加父级目录,因为此时app.js在内存里与output配置的目录无关：

<script type="text/javascript" src="app.js"></script>

详细文档在这里查看：<http://webpack.github.io/docs/webpack-dev-server.html>

**如何区分开发及生产环境**

在webpack.config.js使用process.env.NODE\_ENV进行判断在package.json里面的script设置环境变量，注意mac与windows的设置方式不一样

"scripts": {

"publish-mac": "export NODE\_ENV=prod&&webpack -p --progress --colors",

"publish-win": "set NODE\_ENV=prod&&webpack -p --progress --colors",

"dev-mac": "export NODE\_ENV=dev&&webpack-dev-server",

"dev-win": "set NODE\_ENV=dev&&webpack-dev-server

}

**常用plugins**

* 代码热替换, HotModuleReplacementPlugin
* 将css成生文件，而非内联，ExtractTextPlugin
* 报错但不退出webpack进程，NoErrorsPlugin
* 代码丑化，UglifyJsPlugin，开发过程中不建议打开
* 多个 html共用一个js文件(chunk)，可用CommonsChunkPlugin
* 清理文件夹，Clean
* 调用模块的别名ProvidePlugin，例如想在js中用$，如果通过webpack加载，需要将$与jQuery对应起来