**webpack笔记整理**

webpack的四个核心概念

1，入口（entry）

2，输出（output）

3，loader

4，插件（pulgins）

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin'); // 通过 npm 安装

const webpack = require('webpack'); // 用于访问内置插件

module.exports = {

entry: './path/to/my/entry/file.js',

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'my-first-webpack.bundle.js'

},

module: {

rules: [

{ test: /\.txt$/, use: 'raw-loader' }

]

},

plugins: [

    new HtmlWebpackPlugin({template: './src/index.html'})  
  ]

};

以上为基础用法

**入口**

简写

entry: './path/to/my/entry/file.js' 等于

entry: {

main: './path/to/my/entry/file.js'

}

对象语法

entry: { app: './src/app.js', vendors: './src/vendors.js' }

对象语法最繁琐也是最可扩展的方式

分离应用程序和三方库入口（上面对象语法代码）

多页面应用

entry: { pageOne: './src/pageOne/index.js', pageTwo: './src/pageTwo/index.js', pageThree: './src/pageThree/index.js' }

**出口**

出口最少要在output中配置filename和path（只能配置绝对路径）两个属性

多入口起点

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

// 写入到硬盘：./dist/app.js, ./dist/search.js

使用CDN和资源hash的示例

output: { path: "/home/proj/cdn/assets/[hash]", publicPath: "http://cdn.example.com/assets/[hash]/"}

在编译时不知道最终输出文件的 publicPath 的情况下，publicPath 可以留空，并且在入口起点文件运行时动态设置。如果你在编译时不知道 publicPath，你可以先忽略它，并且在入口起点设置 \_\_webpack\_public\_path\_\_。

\_\_webpack\_public\_path\_\_ = myRuntimePublicPath

模式mode

仅有两个值 development和production

Development：会将 process.env.NODE\_ENV 的值设为 development。启用 NamedChunksPlugin 和 NamedModulesPlugin。

Production：会将 process.env.NODE\_ENV 的值设为 production。启用 FlagDependencyUsagePlugin, FlagIncludedChunksPlugin, ModuleConcatenationPlugin, NoEmitOnErrorsPlugin, OccurrenceOrderPlugin, SideEffectsFlagPlugin 和 UglifyJsPlugin.

例：

module.exports = { mode: 'production'};

cli传参的形式：

webpack --mode=production

**loader**

用于对模块的源代码进行转换。可以加不同语言转为js

安装loader：npm install --save-dev css-loader

配置：

module.exports = { module: { rules: [ { test: /\.css$/, use: 'css-loader' }, { test: /\.ts$/, use: 'ts-loader' } ] }};

使用loader的三种方式

1，配置：在webpack.config.js文件中配置

module: { rules: [ { test: /\.css$/, use: [ { loader: 'style-loader' }, { loader: 'css-loader', options: { modules: true } } ] } ] }

2，内联：在每个import语句中显示指定loader

import Styles from 'style-loader!css-loader?modules!./styles.css';

3，CLI：在shell命令中指定它们

webpack --module-bind jade-loader --module-bind 'css=style-loader!css-loader'

特性

loader 支持链式传递。能够对资源使用流水线(pipeline)。一组链式的 loader 将按照相反的顺序执行。loader 链中的第一个 loader 返回值给下一个 loader。在最后一个 loader，返回 webpack 所预期的 JavaScript。loader 可以是同步的，也可以是异步的。loader 运行在 Node.js 中，并且能够执行任何可能的操作。loader 接收查询参数。用于对 loader 传递配置。loader 也能够使用 options 对象进行配置。除了使用 package.json 常见的 main 属性，还可以将普通的 npm 模块导出为 loader，做法是在 package.json 里定义一个 loader 字段。插件(plugin)可以为 loader 带来更多特性。loader 能够产生额外的任意文件。解析

loader 遵循标准的模块解析。多数情况下，loader 将从模块路径（通常将模块路径认为是 npm install, node\_modules）解析。loader 模块需要导出为一个函数，并且使用 Node.js 兼容的 JavaScript 编写。通常使用 npm 进行管理，但是也可以将自定义 loader 作为应用程序中的文件。按照约定，loader 通常被命名为 xxx-loader（例如 json-loader）。 **插件plugins**

插件的目的是为了解决loader无法实现的其他事。插件是一个具有apply属性的js对象。Apply属性会被webpack compiler调用，并且compiler对象可在整个编译生命周期访问

const pluginName = 'ConsoleLogOnBuildWebpackPlugin';class ConsoleLogOnBuildWebpackPlugin { apply(compiler) { compiler.hooks.run.tap(pluginName, compilation => { console.log("webpack 构建过程开始！"); }); }}

compiler hook 的 tap 方法的第一个参数，应该是驼峰式命名的插件名称。建议为此使用一个常量，以便它可以在所有 hook 中复用。

由于插件可以携带参数/选项，你必须在 webpack 配置中，向 plugins 属性传入 new 实例。根据你的 webpack 用法，这里有多种方式使用插件

例子

const HtmlWebpackPlugin = require('html-webpack-plugin'); //通过 npm 安装const webpack = require('webpack'); //访问内置的插件const path = require('path');const config = { entry: './path/to/my/entry/file.js', output: { filename: 'my-first-webpack.bundle.js', path: path.resolve(\_\_dirname, 'dist') }, module: { rules: [ { test: /\.(js|jsx)$/, use: 'babel-loader' } ] }, plugins: [ new webpack.optimize.UglifyJsPlugin(), new HtmlWebpackPlugin({template: './src/index.html'}) ]};module.exports = config;

node API

const webpack = require('webpack'); //访问 webpack 运行时(runtime) const configuration = require('./webpack.config.js'); let compiler = webpack(configuration); compiler.apply(new webpack.ProgressPlugin()); compiler.run(function(err, stats) { // ... });

配置：

通过 require(...) 导入其他文件通过 require(...) 使用 npm 的工具函数使用 JavaScript 控制流表达式，例如 ?: 操作符对常用值使用常量或变量编写并执行函数来生成部分配置应避免

在使用 webpack 命令行接口(CLI)（应该编写自己的命令行接口(CLI)，或使用 --env）时，访问命令行接口(CLI)参数导出不确定的值（调用 webpack 两次应该产生同样的输出文件）编写很长的配置（应该将配置拆分为多个文件）