Webpack 教程

一、优势：

1. webpack 是以 commonJS 的形式来书写脚本滴，但对 AMD/CMD 的支持也很全面，方便旧项目进行代码迁移。

2. 能被模块化的不仅仅是 JS 了。

3. 开发便捷，能替代部分 grunt/gulp 的工作，比如打包、压缩混淆、图片转 base64 等。

4. 扩展性强，插件机制完善，特别是支持 React 热插拔的功能让人眼前一亮。

安装配置：

1、安装：

我们常规直接使用 npm 的形式来安装：

$ npm install webpack -g

常规项目还是把依赖写入 package.json 包去更人性化：

$ npm init

$ npm install webpack --save-dev

2、配置：

每个项目下都必须配置有一个 webpack.config.js ，它的作用如同常规的 gulpfile.js/Gruntfile.js ，就是一个配置项，告诉 webpack 它需要做什么。

示例：

```
Var webpack = require(‘webpack’);

Var commonsPlugin = new webpack.optimize.CommonsChunKplugin(commin.js);

Module.exports = {
```

```
//插件项
Plugins:[commonsPlugin],
//页面入口文件配置
Entry:{
    Index:'./src/js/page/index.js'
},
//入口文件输出配置
Output:{
    Path:'dist/js/page',
    Filename:'[name].js'
}
Module:{
    //加载器配置
    Loaders:[
        {test:/\.css$/,loader:'style-loader!Css-loader'},
        {test:/\.js$/,loader:'jsx-loader?harmony'},
        {test:/\.scss$/,loader:'style!Css!Sass?sourceMap'},
        {test:/\.(png|jpg)$/,loader:'url-loader?Linmit=8192'}
    ]
},
//其他解决方案配置
Resolve:{
```

```
        Root:'c:/aaa/vvv/src',    //绝对路径
        Extensions:[ '','.js','.josn','.scss'],
        Alias:{
            AppStore:'js/stores/AppStores.js',
            ActionType:'js/actions/ActionType.js',
            AppAction:'js/actions/AppAction.js'
        }
    }
}
```

(1) plugins 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。

(2) entry 是页面入口文件配置，output 是对应输出项配置（即入口文件最终要生成什么名字的文件、存放到哪里），其语法大致为：

```
{
    Entry:{
        Page1:"./page1",
        //支持数组形式，将加载数组中的所有模块，但以最后一个模块作为输 出
        Page2:["./entry1","./entry2"]
        },
        Output:{
            Path:"dist/js/page",
```

```
        Filename:"[name].bundle.js"
    }
}
```

该段代码最终会生成一个 page1.bundle.js 和 page2.bundle.js，并存放到 ./dist/js/page 文件夹下。

(3) module.loaders 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$/,loader:’style-loader!Css-loader’},
        //.js 文件使用 jsx-loader 来编译处理
        {test:/\.js$/,loader:’jsx-loader?harmony’},
        //.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?Linmit=8192’}
        ]
}
```

如上，"-loader"其实是可以省略不写的，多个 loader 之间用“!”连接起来。

最后一个 url-loader ，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)*

⑷ 最后是 resolve 配置，这块很好理解，直接写注释了：

```
Resolve:{
    //查找 module 的话从这里开始查找
    Root:‘c:/abc/bba/src’,     //绝对路径
    //自动扩展文件后缀名，意味着我们 require 模块可以省略不谢后缀名
    Extensions : [ ‘’ ,’ .js’ ,’ .json’ ,’ .scss’ ],
    //模块别名定义，方便后续直接引用别名，无须多写长长的地址
    Alias:{
        AppStore:‘ js/stores/AppStores.js’,     //后续直接 require( ‘AppStore’ )即可
        ActionType:‘ js/actions/ActionType.js’,
        AppAction:js/actions/AppAction.js’
        }
}
```

运行 webpack：

webpack 的执行也很简单，直接执行

$ webpack --display-error-details

即可，后面的参数“--display-error-details”是推荐加上的，方便出错时能查阅更详尽的信息（比如 webpack 寻找模块的过程），从而更好定位到问题。

其他主要的参数有：

$ webpack --consig xxx.js　//使用另一份配置文件（比如 webpack.config2.js）来打包

$ webpack --watch　//监听变动并自动打包

$ webpack -p　//压缩混淆脚本，这个非常非常重要

$ webpack -d　//生成 map 映射文件，告知哪些模块被最终打包到哪里了

其中的 -p 是很重要的参数，曾经一个未压缩的 700kb 的文件，压缩后直接降到 180kb（主要是样式这块一句就独占一行脚本，导致未压缩脚本变得很大）。

模块引入：

一. HTML

直接在页面引入 webpack 最终生成的页面脚本即可，不用再写什么 data-main 或 seajs.use 了：

```
<!DOCTYPE html>
<html>
<head lang=" en" >
<mate charset=" UTF-8" >
<title>demo</title>
</head>
<body>
<script src=" dist/js/page/common.js" ></script>
<script src=" dist/js/page/index.js" ></script>
```

```
</body>

</html>
```

可以看到我们连样式都不用引入，毕竟脚本执行时会动态生成<style>并标签打到 head 里。

二. JS

各脚本模块可以直接使用 commonJS 来书写，并可以直接引入未经编译的模块，比如 JSX、sass、coffee 等（只要你在 webpack.config.js 里配置好了对应的加载器）。

我们再看看编译前的页面入口文件（index.js）：

```
Require(‘../../css/reset.scss’);    //加载初始化样式

Require(‘../../allComponent.scss’);  //加载组件样式

Var React = require(‘react’);

Var AppWrap = require(../component/AppWrap’);  //加载组件

Var createRedux = require(‘redux’).createRedux;

Var Provider = require(‘redux/react’).Provider;

Var stores = require(‘AppStore’);

Var redux = createRedux(stores);

Var App = React.createClass({

    Render.function(){

        Return(

            <provider redux ={redux}>

                {function(){return <AppWrap/>;}}
```

```
            </provider>
            );
        }
});
React.render(
    <App/>,document.body
);
```

一切就是这么简单么么哒~ 后续各种有的没的，webpack 都会帮你进行处理。

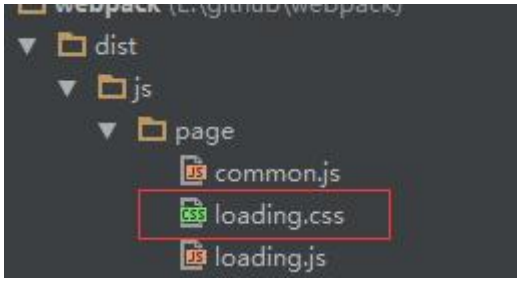
补充技巧：

一. 独立打包样式文件

有时候可能希望项目的样式能不要被打包到脚本中，而是独立出来作为.css，然后在页面中以<link>标签引入。这时候我们需要 extract-text-webpack-plugin 来帮忙：

```
Var webpack = require(‘webpack’);
Var commonsPlugin = new webpack.optimize.CommonsChunkplugin(‘common.js’);
Var ExtractTextplugin = require(‘extract-text-webpack-plugin’);
Module.exports = {
    Plugins:[commonPlugin,new ExtractTextPlugin(“[name].css”)],
    Entry:{
        //省略其他配置
    }
}
```

最终 webpack 执行后会乖乖地把样式文件提取出来：



二. 使用 CDN/远程文件

有时候我们希望某些模块走 CDN 并以<script>的形式挂载到页面上来加载，但又希望能在 webpack 的模块中使用上。

这时候我们可以在配置文件里使用 externals 属性来帮忙：

```
{
    Externals:{
        //require("jquery")是引用自外部模块对应全局变量 jQuery
        "jquery" : "jquery"
    }
}
```

需要留意的是，得确保 CDN 文件必须在 webpack 打包文件引入之前先引入。

我们倒也可以使用 script.js 在脚本中来加载我们的模块：

```
Var $script = require("scriptjs");
$script("//ajac.googleapis.com/ajax/libs/jquery/2.0.0/jquery.min.js",function(){
    $('body').html('it works!')
})
```

三. 与 grunt/gulp 配合

以 gulp 为示例，我们可以这样混搭：

```
Gulp.task("webpack",function(callback){
    //run webpack
    Webpack({
        //configuration
    },function(err,stats){
        If(err) throw new gutil.PluginError("webpack",err);
        Gutil.log("[webpack]",stats.toString({
            //output options
        }));
        Callback();
    })
})
```

当然我们只需要把配置写到 webpack({ ... }) 中去即可，无须再写 webpack.config.js 了。