WebPack

这一节我们将系统的讲解webpack，包括：

一.webpack介绍
- 1.webpack是什么
- 2.为什么引入新的打包工具
- 3.webpack核心思想

二、webpack安装

三、webpack使用
- 1.命令行调用
- 2.配置文件

四、webpack参数配置
- 1.entry和output
- 2.单一入口
- 3.多个入口
- 4.多个打包目标

五、webpack支持Jsx和Es6

六、webpack loaders
- 1.loader定义
- 2.loader功能
- 3.loader配置
- 4.使用loader

七、webpack开发环境与生产环境

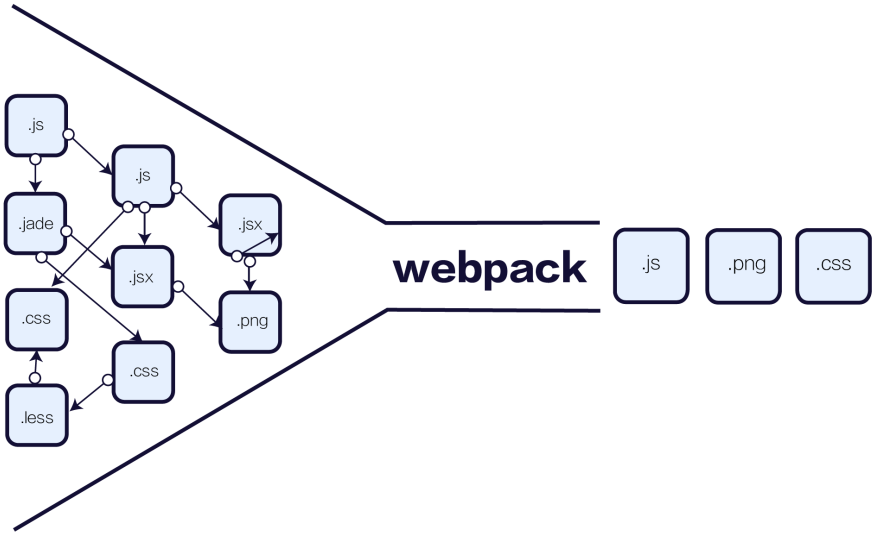
八、webpack分割vendor代码和应用业务代码

九、webpack develop server
- 1.安装webpack-Dev-server
- 2.启动webpack-Dev-server
- 3.代码监视
- 4.自动刷新
- 5.热加载（hot module replacement）
- 6.在webpack.config.js中配置webpack develop server

2.2.1 webpack介绍

webpack是什么

webpack is a module bunder.webpack takes modules with dependencies and generates static assets representing those modules



webpack是一个模块打包工具，输入为包含依赖关系的模块集，输出为打包合并的前端静态资源。webpack是同时支持AMD和CommonJS的模块定义方式，不仅如此，webpack可以将任何前端资源视为模块，如CSS，图片，文本。

为什么要引入新的打包工具

在webpack出现之前，已经有了一些打包工具，如Broweserify，grount，gulp，那为什么不优化这些工具，而是重复造轮子？

webpack之前的打包工具功能单一，只能完成特定的任务，然而web前端工程是复杂的，一个webapp对于业务代码的要求可能有：

1.代码可以分块，实现按需加载

2.首屏加载时间尽量要少

3.需要集成一些第三方库

对于模块打包工具，单一的支持CommonJS的打包在大型项目中是不够用的，为了满足一个大型项目的前端需求，那么一个打包工具应该包含一些这些功能：

1.支持多个bundler输出->解决代码分块问题

2.异步加载->按需加载，优化首屏加载时间

3.可定制化->可以集成第三方库，可以定制化打包过程

4.其他资源也可以定义为模块

webpack的出现正式解决了这些问题，在webpack中，提供了一下这些功能：

1.代码分块：webpack有两种类型的模块依赖，一种是同步的，一种是异步的。在打包的过程中可以将代码输出为代码块（chunk），代码块可以实现按需加载。异步加载的代码块通过分割点（spliting point）来确定。

2.loaders：webpack本身只会处理javascript，为了实现将其他资源也定义为模块，并转化为JavaScript，webpack定义loaders，不同的loader可以将对应的资源转化为JavaScript模块。

3.智能的模块解析：webpack可以很容易将第三方库转化为模块集成到项目代码中，模块的依赖可以用表达式的方式（这在其他打包工具中是没有支持的），这种模块依赖叫做动态模块依赖。

4.插件系统：webpack的可定制化在于其插件系统，其本身的很多功能也是通过插件的方式实现，插件系统形成了webpack的生态，是的可以使用很多开源的第三方插件。

webpack核心思想

webpack的三个核心

1.万物皆模块：在webpack的世界中，除了javascript，其他任何资源都可以当做模块的方式引用

2.按需加载：webpack的优化关键在于代码体积.当应用体积增大，实现代码的按需加载是刚需，这也是webpack出现的根本原因

3.可定制化：任何一个工具都不可能解决所有问题，提供解决方案才是最可行的，webpack基于可定制化的理念构建，通过插件系统，配置文件，可以实现大型项目的定制需求。

2.2.2安装配置

第一步：node.js

webpack是node实现，首先需要到node.js下载安装最新版本的Node.js

第二步：webpack-cli

Node.js安装好过后，打开命令行终端，通过npm命令安装：

```
// -g 参数表示全局安装
$ npm install webpack -g
```

第三步：新建空前端项目

为了使用 webpack，先新建一个空前端项目，创建一个目录，目录结构如下:

```
.
├── index.html      // 入口 HTML
├── dist            // dist 目录放置编译过后的文件文件
└── src             // src 目录放置源文件
    └── index.js    // 入口 js
```

其中 html 内容:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Hello React!</title>
</head>
<body>
  <div id="AppRoot"></div>
```

```
  <script src="dist/index.js"></script>
</body>
</html>
```

index.js 内容为:

```
alert('hello world webpack');
```

第四步：在项目中安装 webpack

```
// 初始化 package.json,  根据提示填写 package.json 的相关信息
$ npm init

// 下载 webpack 依赖
// --save-dev 表示将依赖添加到 package.json 中的 'devDependencies' 对象中
$  npm install webpack --save-dev
```

*** 第五步：Develop Server 工具（可选）**

dev server 可以实现一个基于 node + express 的前端 server

```
$ npm install webpack-dev-server --save-dev
```

2.2.3 webpack 使用

命令行调用

在之前创建的目录下执行：

```
$ webpack src/index.js dist/index.js
```

执行成功过后会出现如下信息：

```
Hash: 9a8e7e83864a07c0842f
Version: webpack 1.13.1
Time: 37ms
   Asset     Size  Chunks             Chunk Names
index.js  1.42 kB       0  [emitted]  main
   [0] ./src/index.js 29 bytes {0} [built]
```

可以查看 dist/index.js 的编译结果:

```
/******/ (function(modules) { // webpackBootstrap
//           .......... UMD 定义内容
/******/ })
/************************************************************************/
/******/ ([
/* 0 */
/***/ function(module, exports) {
    // index.js 的内容被打包进来
    alert('hello world webpack');

/***/ }
/******/ ]);
```

在浏览器中打开 index.html ：

配置文件

以命令执行的方式需要填写很长的参数，所以 webpack 提供了通过配置的方式执行，在项目目录下创建 webpack.config.js 如下：

```
var webpack = require('webpack')
module.exports = {
  entry: './src/index.js',
  output: {
    path: './dist/',
    filename: 'index.js'
  }
}
```

执行：

```
$ webpack
```

会和通过命令执行有同样的输出

2.2.4 webpack 配置

entry 和 output

webpack 的配置中主要的两个配置 key 是，entry 和 output。

```
{
    entry: [String | Array | Object], // 入口模块
    output: {
        path: String,      // 输出路径
        filename: String   // 输出名称或名称 pattern
        publicPath: String // 指定静态资源的位置
```

```
        ...                 // 其他配置
    }
}
```

单一入口

如果只有一个入口文件，可以有如下几种配置方式

```
// 第一种 String
{
  entry: './src/index.js',
  output: {
    path: './dist/',
    filename: 'index.js'
  }
}

// 第二种 Array
{
  entry: ['./src/index.js'],
  output: {
    path: './dist/',
    filename: 'index.js'
  }
}

// 第三种 Object
{
  entry: {
    index: './src/index.js',
  },
  output: {
    path: './dist/',
    filename: 'index.js'
  }
}
```

多个入口文件

当存在多个入口时 ，可以使用 Array 的方式，比如依赖第三方库 bootstrap ，最终 bootstrap 会被追加到打包好的 index.js 中，数组中的最后一个会被 export。

```
{
  entry: ['./src/index.js', './vendor/bootstrap.min.js'],
  output: {
    path: './dist',
    filename: "index.js"
  }
}
```

最终的输出结果如：

```
/******/ ([
/* 0 */
/***/ function(module, exports, __webpack_require__) {

    __webpack_require__(1);

    // export 最后一个
    module.exports = __webpack_require__(2);


/***/ },
/* 1 */
/***/ function(module, exports) {

    alert('hello world webpack');

/***/ },
/* 2 */
/***/ function(module, exports) {
    // bootstrap 的内容被追加到模块中
```

```
    console.log('bootstrap file');


/***/ }
/******/ ])
```

多个打包目标

上面的例子中都是打包出一个 index.js 文件，如果项目有多个页面，那么需要打包出多个文件，webpack 可以用对象的方式配置多个打包文件

```
{
  entry: {
    index: './src/index.js',
    a: './src/a.js'
  },
  output: {
    path: './dist/',
    filename: '[name].js'
  }
}
```

最终会打包出：

```
.
├── a.js
└── index.js
```

文件名称 pattern

- [name] entry 对应的名称
- [hash] webpack 命令执行结果显示的 Hash 值
- [chunkhash] chunk 的 hash

为了让编译的结果名称是唯一的，可以利用 hash 。

2.2.5 webpack 支持 Jsx

现在我们已经可以使用 webpack 来打包基于 CommonJs 的 Javascript 模块了，但是还没法解析 JSX 语法和 Es6 语法。下面我们将利用 Babel 让 webpack 能够解析 Es6 和 Babel

第一步：npm install 依赖模块

```
// babel 相关的模块
$ npm install babel-loader babel-preset-es2015 babel-preset-stage-0 babel-preset-react babel-polyfill
--save-dev

// react 相关的模块
$ npm install react react-dom --save
```

第二步：webpack.config.js 中添加 babel loader 配置

```
{
    entry: {
        index: './src/index.js',
        a: './src/a.js'
    },
    output: {
        path: './dist/',
        filename: '[name].js'
    },
    module: {
        loaders: [{
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel',
            query: {
                presets: ['es2015', 'stage-0', 'react']
            }
        }]
    }
}
```

第三步: 修改 index.js 为 React 的语法

src/index.js 内容改为:

Es6 的知识在后面的章节中讲解，目前我们暂时以 Es5 的方式来写，但是配置已经支持了 Es6 的编译，熟悉 Es6 的读者也可以直接写 Es6

```
// 通过 require 的方式依赖 React，ReactDOM
var React = require('react');
var ReactDOM = require('react-dom');

var Hello = React.createClass({
  render: function render() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(
  <Hello name="World" />,
  document.getElementById('AppRoot')
);
```

第四步：运行 webpack

```
$ webpack
```

执行结果：

```
Hash: ae2a037c191c18195b6a
Version: webpack 1.13.1
Time: 1016ms
   Asset     Size  Chunks             Chunk Names
    a.js  1.42 kB       0  [emitted]  a
index.js   700 kB       1  [emitted]  index
    + 169 hidden modules
```

浏览器中打开 index.html 会显示 Hello World

2.2.6 webpack loaders

在配置 JSX 的过程中，使用到了 loader， 前面已经介绍过 webpack 的核心功能包含 loader，通过 loader 可以将任意资源转化为 javascript 模块。

loader 定义

Loaders are transformations that are applied on a resource file of your app.
(Loaders 是应用中源码文件的编译转换器)
也就是说在 webpack 中，通过 loader 可以实现 JSX 、Es6、CoffeeScript 等的转换，

loader 功能

1. loader 管道：在同一种类型的源文件上，可以同时执行多个 loader ， loader 的执行方式可以类似管道的方式。
2. loader 可以支持同步和异步
3. loader 可以接收配置参数
4. loader 可以通过正则表达式或者文件后缀指定特定类型的源文件
5. 插件可以提供给 loader 更多功能
6. loader 除了做文件转换以外，还可以创建额外的文件

loader 配置

新增 loader 可以在 webpack.config.js 的 module.loaders 数组中新增一个 loader 配置。
一个 loader 的配置为：

```
{
    // 通过扩展名称和正则表达式来匹配资源文件
    test: String ,
    // 匹配到的资源会应用 loader， loader 可以为 string 也可以为数组
    loader: String | Array
}
```

感叹号和数组可以定义 loader 管道:

```
{
    module: {
        loaders: [
            { test: /\.jade$/, loader: "jade" },
            // => .jade 文件应用  "jade" loader

            { test: /\.css$/, loader: "style!css" },
            { test: /\.css$/, loaders: ["style", "css"] },
            // => .css 文件应用  "style" 和 "css" loader
        ]
    }
}
```

loader 可以配置参数

```
{
    module: {
        loaders: [
            // => url-loader 配置  mimetype=image/png 参数
            {
                test: /\.png$/,
                loader: "url-loader?mimetype=image/png"
            }, {
                test: /\.png$/,
                loader: "url-loader",
                query: { mimetype: "image/png" }
            }

        ]
    }
}
```

使用 loader

第一步: 安装

loader 和 webpack 一样都是 Node.js 实现，发布到 npm 当中，需要使用 loader 的时候，只需要

```
$ npm install xx-loader --save-dev

// eg css loader
$ npm install css-loader style-loader --save-dev
```

第二步：修改配置

```
{
    entry: {
        index: './src/index.js',
        a: './src/a.js'
    },
    output: {
        path: './dist/',
        filename: '[name].js'
    },
    module: {
        loaders: [{
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel',
            query: {
                presets: ['es2015', 'stage-0', 'react']
            }
        }, {
            test: /\.css$/,
            loader: "style-loader!css-loader"
        }]
    }
}
```

第三步：使用

前面我们已经使用过 jsx loader 了， loader 的使用方式有多种

1. 在配置文件中配置
2. 显示的通过 require 调用
3. 命令行调用

显示的调用 require 会增加模块的耦合度，应尽量避免这种方式

以 css-loader 为例子，在项目 src 下面创建一个 css

src/style.css

```
body {
    background: red;
    color: white;
}
```

修改 webpack 配置 entry 添加

```
entry: {
    index: ['./src/index.js', './src/style.css']
}
```

执行 webpack 命令然后打开 index.html 会看到页面背景被改为红色。
最终的编译结果为：

```
....
function(module, exports, __webpack_require__) {
    exports = module.exports = __webpack_require__(171)();
    exports.push([module.id, "\nbody {\n background: red;\n color: white;\n}\n", ""]);
}
....
```

可以看到 css 被转化为了 javascript, 在页面中并非调用 <link rel="stylesheet" href=""> 的方式， 而是使用 inline 的<style>.....</style>
另外一种方法是直接 require， 修改 src/index.js:

```
var css = require("css!./style.css");
```

编译结果相同。

2.2.7 webpack 开发环境与生产环境

前端开发环境通常分为两种，开发环境和生成环境，在开发环境中，可能我们需要日志输出，sourcemap ，错误报告等功能，在生成环境中，需要做代码压缩，hash 值生成。两种环境在其他的一些配置上也可能不同。
所以为了区分，我们可以创建两个文件：

- webpack.config.js // 开发环境
- webpack.config.prod.js // 生产环境

生产环境 build 用如下命令：

```
$ webpack --config webpack.config.prod.js
```

在本章深入 webpack 小节中会更多的介绍生产环境中的优化

2.2.8 webpack 插件

webpack 提供插件机制，可以对每次 build 的结果进行处理。配置 plugin 的方法为在 webpack.config.js 中添加：

```
{
  plugins: [
   new BellOnBundlerErrorPlugin()
  ]
}
```

plugin 也是一个 npm 模块，安装一个 plugin ：

```
$ npm install bell-on-bundler-error-plugin --save-dev
```

2.2.9 webpack 分割 vendor 代码和应用业务代码

在上面的 jsx 配置中，我们将 React 和 ReactDOM 一起打包进了项目代码。为了实现业务代码和第三方代码的分离，我们可以利用 CommonsChunkPlugin 插件.
修改 webpack.config.js

```
{
    entry: {
        index: './src/index.js',
        a: './src/a.js',
        // 第三方包
        vendor: [
          'react',
          'react-dom'
        ]
    },
    output: {
        path: './dist/',
        filename: '[name].js'
    },
    module: {
        loaders: [{
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel',
            query: {
                presets: ['es2015', 'stage-0', 'react']
            }
        }, {
            test: /\.css$/,
            loader: "style-loader!css-loader"
        }]
```

```
    },
    plugins: [
      new webpack.optimize.CommonsChunkPlugin(/* chunkName= */"vendor", /* filename=
*/"vendor.bundle.js")
    ]
}
```

执行 webpack 命令，输出日志：

```
Hash: f1256dc00b9d4bde8f7f
Version: webpack 1.13.1
Time: 1459ms
           Asset       Size  Chunks             Chunk Names
            a.js  109 bytes       0  [emitted]  a
          index.js    10.9 kB       1  [emitted]  index
vendor.bundle.js     702 kB       2  [emitted]  vendor
   [0] multi vendor 40 bytes {2} [built]
   [0] multi index 40 bytes {1} [built]
    + 173 hidden modules
```

index.js 体积变小了，多出了 vendor.bundle.js

2.2.10 webpack develop server

在前端开发的过程中，通常需要启动一个服务器，把开发打包好的前端代码放在服务器上，通过访问服务器访问并测试（因为可以有些情况需要 ajax 请求）。 webpack 提供了一个机遇 node.js Express 的服务器 - webpack-dev-server 来帮助我们简化服务器的搭建，并提供服务器资源访问的一些简单配置。

安装 webpack-dev-server

```
$ npm install webpack-dev-server -g
```

启动 webpack-dev-server

```
$ webpack-dev-server --content-base ./
```

--content-base ./ 参数表示将当前目录作为 server 根目录。 命令启动过后，会在 8080 端口启动一个 http 服务，通过访问 http://localhost:8080/index.html 可以访问 index.html 内容。

如果访问提示报错：

```
Uncaught ReferenceError: webpackJsonp is not defined
```

原因是 html 中没有引用 vendor.bundle.js, 修改 html ：

```
<!-- vendor 必须先于 index.js -->
<script src="dist/vendor.bundle.js"></script>
<script src="dist/index.js"></script>
```

修改 index.html 过后可以看到正确结果

代码监控

webpack-dev-server 除了提供 server 服务以外， 还会监控源文件的修改，如果源文件改变了，会调用 webpack 重新打包

修改 style.css 中的内容为：

```
body {
 background: whitesmoke;
 color: #333;
 font-size: 100px;
}
```

可以看到输出以下日志：

```
  [168] ./~/react/lib/renderSubtreeIntoContainer.js 466 bytes {2} [built]
webpack: bundle is now VALID.
webpack: bundle is now INVALID.
Hash: cc7d7720b1a0fcbef972
Version: webpack 1.13.0
Time: 76ms
chunk    {0} a.js (a) 32 bytes {2}
     + 1 hidden modules
chunk    {1} index.js (index) 10.3 kB {2}
  [170] ./~/css-loader!./src/style.css 230 bytes {1} [built]
     + 5 hidden modules
chunk    {2} vendor.bundle.js (vendor) 665 kB
     + 168 hidden modules
webpack: bundle is now VALID.
```

这个时候说明代码已经修改了，但是这个时候刷新浏览器过后，背景是没有改变的，原因是 webpack-dev-server 的打包结果是放在内存的，查看 dist/index.js 的内容实际上是没有改变的，那如何访问内存中的打包内容呢？

修改 webpack.config.js 的 output.publicPath:

```
  output: {
      path: './dist/',
```

```
    filename: '[name].js',
    publicPath: '/dist'
    // webpack-dev-server 启动目录是  `/`, `/dist` 目录是打包的目标目录相对于启动目录的路径
  },
```

重新启动

```
$ ctrl + c 结束进程
$ webpack-dev-server
```

修改 style.css 再刷新页面，修改的内容会反映出来。

自动刷新

上面的配置已经能做到自动监控代码，每次修改完代码，刷新浏览器就可以看到最新结果，但是 webpack-dev-server 还提供了自动刷新功能，有两种模式。

Iframe 模式

修改访问的路径： **http://localhost:8080/index.html** -> **http://localhost:8080/webpack-dev-server/index.html** 。这个时候每次修改代码，打包完成过后都会自动刷新页面。

- 不需要额外配置，只用修改路径
- 应用被嵌入了一个 iframe 内部，页面顶部可以展示打包进度信息
- 因为 iframe 的关系，如果应用有多个页面，无法看到当前应用的 url 信息

inline 模式

启动 webpack-dev-server 的时候添加 --inline 参数

- 需要添加 --inline 配置参数
- 没有顶部信息提示条，提示信息在控制台中展现

热加载（hot module replacement)

webpack-dev-server 还提供了模块热加载的方式，在不刷新浏览器的条件下，应用最新的代码更新，启动 webpack-dev-server 的时候添加 --inline --hot 参数就可以体验。

```
$ webpack-dev-server --inline --hot
```

修改代码在浏览器控制台中会看到这样的日志输出：

```
[HMR] Waiting for update signal from WDS...
vendor.bundle.js:670 [WDS] Hot Module Replacement enabled.
2vendor.bundle.js:673 [WDS] App updated. Recompiling...
vendor.bundle.js:738 [WDS] App hot update...
vendor.bundle.js:8152 [HMR] Checking for updates on the server...
vendor.bundle.js:8186 [HMR] Updated modules:
vendor.bundle.js:8188 [HMR]  - 245
vendor.bundle.js:8138 [HMR] App is up to date.
```

在 webpack.config.js 中配置 webpack develop server

修改 webpack.config.js 添加：

```
plugins: [
  new webpack.optimize.CommonsChunkPlugin(
    /* chunkName= */"vendor",
    /* filename= */"vendor.bundle.js", Infinity),
  // 需要手动添加 HotModuleReplacementPlugin , 命令行的方式会自动添加
  new webpack.HotModuleReplacementPlugin()
],
devServer: {
  hot: true,
  inline: true
}
```

不加参数直接执行 webpack-dev-server

```
$ webpack-dev-server
```

webpack-dev-server 还提供了其他的一些功能，如：

1. 配置 proxy
2. 访问 node.js API
3. 和现有的 node 服务集成

基于这些功能可以实现很多自定义的配置。