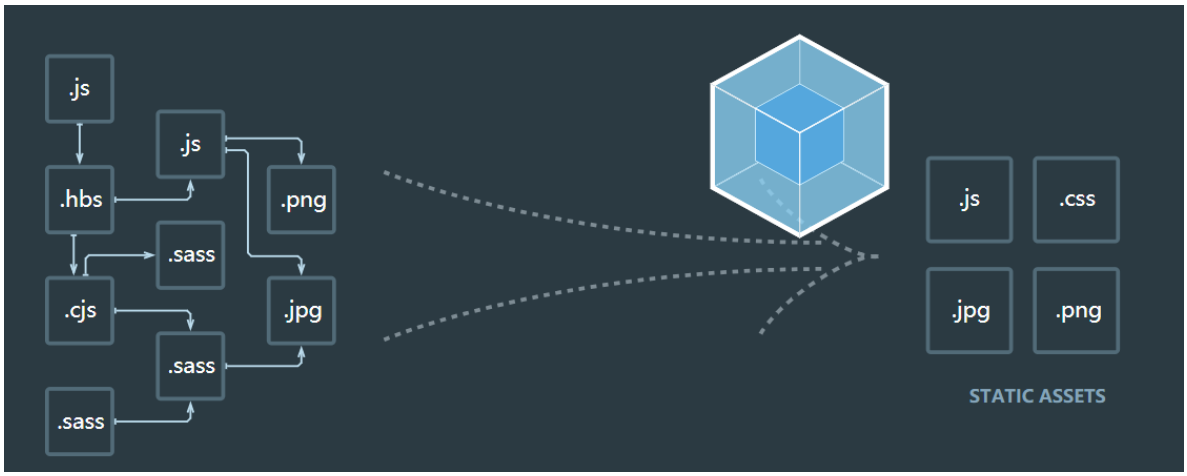
Webpack笔记整理

一、Webpack是什么的?

就目前来说打包工具有很多

1. `gulp`：用代码方式来写打包脚本，并且代码采用流式(stream)的写法，只抽象出了gulp.src, gulp.pipe, gulp.dest, gulp.watch 接口，运用相当简单。更易于学习和使用，使用gulp的代码量能比grunt少一半左右。
2. `Grunt`：最老牌的打包工具，它运用配置的思想来写打包脚本，一切皆配置，所以会出现比较多的配置项，诸如option,src,dest等等。而且不同的插件可能会有自己扩展字段，认知成本高，运用的时候需要明白各种插件的配置规则。
3. `Webpack`: 是模块化管理工具和打包工具。通过 loader 的转换，任何形式的资源都可以视作模块，比如 CommonJs 模块、AMD 模块、ES6 模块、CSS、图片等。它可以将许多松散的模块按照依赖和规则打包成符合生产环境部署的前端资源。还可以将按需加载的模块进行代码分隔，等到实际需要的时候再异步加载。它定位是模块打包器，而 Gulp/Grunt 属于构建工具。Webpack 可以代替 Gulp/Grunt 的一些功能，但不是一个职能的工具，可以配合使用。
4. `Rollup`：下一代 ES6 模块化工具，最大的亮点是利用 ES6 模块设计，利用 tree-shaking生成更简洁、更简单的代码。一般而言，对于应用使用 Webpack，对于类库使用 Rollup；需要代码拆分(Code Splitting)，或者很多静态资源需要处理，再或者构建的项目需要引入很多 CommonJS 模块的依赖时，使用 webpack。代码库是基于 ES6 模块，而且希望代码能够被其他人直接使用，使用 Rollup。

现在的主流打包工具里面还是在使用webpack,可以将webpack看成是下面的一张图



二、Webpack的四大核心是什么?

1. 入口(entry)
2. 出口(output)
3. `loader`模块(module)
4. 插件`plugins`

入口决定了程序从什么地方开始运行，出口决定程序最终打包在什么目录下面，loader决定了webpack到底以什么方式来处理源代码，plugin则可以实现一些第三方的功能

三、Webpack怎么安装使用

```
$ yarn add webpack webpack-cli --dev
```

在安装的过程当中一个要安装`webpack-cli`它是必须的

安装完成以后，我们可以偿试着去写两个部分试一下

Person.js

```
export default class Person{
    constructor(userName){
        this.userName = userName;
    }
}
```

Student.js

```
import Person from "./Person";
class Student extends Person{
    constructor(userName,sex){
        super(userName);
        this.sex = sex;
    }
}

let s = new Student("张三","男");
console.log(s.userName);
```

最后，我们只需要在控制台输入命令

```
$ webpack ./js/Student.js -o ./dist
```

上面的打包过程完成以后，我们在dist的目录下面生成了一个main.js的文件，然后去执行这个文件，这外时候我们就发现可以正常运行了，这样webpack就间接的让我们实现了ES6的模块化

> 为什么要实现ES6的模块化，是因为之前讲过，ES6的模块化可以导入任何文件，这样我们后期可以使用`import`这个命令去导入非JS的文件，如图片，scss，css等，这样这些文件被导入到JS以后，我们还可以再次通过JS去调整里面的东西

上面只是一个简单的打包过程，正常情况下，它是要配置文件的，所以我们可以直接书写配置文件

四、webpack的配置

1. 生产模式

配置入口与出口

webpack无论是哪种模式的配置都离不开4大点，所以我们先在当前项目的根目录下面创建一个`webpack.config.js`的目录，代码如下

```
const path = require("path");
const config = {
    //将模式设置为生产模式
    mode:"production",
    entry:path.join(__dirname,"./js/Student.js"),
    output:{
        filename:"bundle.js",
        path:path.join(__dirname,"./dist"),
        publicPath:"./"
    }
};
module.exports = config;
```

配置`html-webpack-plugin`

这个插件主要的目的就是将生成的js文件或css文件自动的插入到html页面当中去

安装插件

```
$ yarn add html-webpac-plugin --dev
```

配置

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const path = require("path");
const config = {
    entry:path.join(__dirname,"./js/Student.js"),
    output:{
        filename:"bundle.js",
        path:path.join(__dirname,"./dist"),
        publicPath:"./"
    },
    plugins:[
        //配置插件
        new HtmlWebpackPlugin({
            //生成新的文件的名称
            filename:"index.html",
            //模板文件的位置
            template:path.join(__dirname,"./index.html"),
            //将打包生成好的HTML与CSS自动插入
            inject:true,
            title:"标哥哥的webpack第一课"
        })
    ]
};
module.exports = config;
```

我们在上面配置了`htmlWebpackPlugin`以后，同时也配置了title的属性，这样在网页当当中可以通过下面的方式把标题插入进去

```
<title><%=htmlWebpackPlugin.options.title%></title>
```

配置`clean-webpack-plugin`插件

这个插件主要的目的自动清除output下面的东西

安装

```
$ yarn add clean-webpack-plugin --dev
```

配置

```
const {CleanWebpackPlugin} = require("clean-webpack-plugin");
const config ={
    //省略代码....
    plugins:[
        new CleanWebpackPlugin();
        //省略代码....
    ]
}
module.exports = config;
```

babel的配置

安装依赖插件

```
$ yarn add babel-loader @babel/core --dev
```

配置webpack.config.js

```
const config = {
    //省略部分代码
    module:{
        rules:[
            {
                test:/\.js$/,
                exclude:/node_modules/,
                loader:"babel-loader"
            }
        ]
    }
}
```

配置预设信息

1. 在当前项目下面创建一个`.babelrc`的文件
2. 设置preset的信息

在设置`preset`的时候，我们需要第三方的插件，分别是下面三个包

1. `@babel/preset-env`这个包的作用是环境配置环境来决定怎么样转换JS代码
2. `@babel/polyfill`这个包主要的作用就是实现ES6中的一些扩展功能，如`Map,Symbol.....`
3. `core-js`这个包主要的作用就是用于实现`Generater function`生成器函数的功能

```
$ yarn add @babel/preset-env @babel/polyfill core-js@2
```

.babelrc的配置

```
{
    "presets": [
```

```
        [
            "@babel/preset-env",
            {
                "useBuiltIns":"usage",    //这里是用于设置pollfily
                "corejs":"2",
                "targets": {
                    "browsers": [
                        "last 2 version",
                        "> 1%",
                        "not dead"
                    ]
                }
            }
        ]
    ]
}
```

> **代码说明**:
>
> 1. `"useBuiltIns":"usage"`代表使用了的才进行`polyfill`的转换，它还有另一个属性`entry`
> 2. `"corejs":"2"`设置版本
> 3. `browsers`代表浏览器的版本信息

配置CSS文件的加载

webpack可以通过`loader`实现非js文件的导入，只需要有相应的`loader`及配置文件就可以了

它主要使用的点包就是三个

1. `css-loader`这个包用于加载CSS文件
2. `postcss-loader`这个包用于处理CSS的兼容性
3. `mini-css-extract-plugin`用于后期生成CSS文件（JS与CSS代码分离）

安装包

```
$ yarn add css-loader postcss-loader mini-css-extract-plugin --dev
```

配置包的信息

```
//压缩CSS并进行分离
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const config = {
    module:{
        rules:[
            //省略部分代码
            {
                test:/\.css$/,
                use:[
                    MiniCssExtractPlugin.loader,
                    {
                        loader:"css-loader",
                        options:{
                            importLoaders:1
                        }
                    },
                    "postcss-loader"
```

```
                ]
            }
        ]
    },
    plugins:[
        //省略部分代码 .....

        //将JS与CSS进行分离的插件
        new MiniCssExtractPlugin({
            filename: "css/index.[hash:8].css",
            ignoreOrder: false
        }),
    ]
}
```

> 代码说明：
>
> 1. `css-loader`里面的`importLoaders:1`代表的是如果在CSS文件当中又使用了`@import`命令，则向下再1层再交给`postcss-loader`去处理一次

当加载了postcss-loader以后，它个包了是要加载第三方插件的

postcss的插件

1. `postcss-import`
2. `postcss-cssnext`
3. `cssnano`

安装上面的三个插件

```
$ yarn add postcss-loader postcss-import postcss-cssnext cssnano --dev
```

在当前项目的根目录创建**postcss.config.js**，请注意，文件名不能变更，然后进行如下配置

```
//这是postcss的配置文件
module.exports = {
    //postcss处理css文件的时候所需要使用的插件
    plugins:[
        require("postcss-import"),
        require("postcss-cssnext"),
        require("cssnano")
    ]
}
```

配置postcss 的预设env，它的预设信息比较特殊在`package.json`文件里面，我们要添加一个属性叫`browserslist`

```
"browserslist": [
    "last 2 version",
    "> 1%"
]
```

到目前我们的postcss以及CSS的预设就完成了

配置`scss、sass`的加载

这个的加载与CSS的加载差不多，所以不需要太多的设置，只需要安装相应的包就可以了

```
$ yarn add node-sass sass-loader --dev
```

在webpack.config.js中进行配置

```
const config = {
    //省略部分代码
    module:{
        rules:[
            //省略部分代码
             {
                test: /\.s(c|a)ss$/,
                use: [
                    MiniCssExtractPlugin.loader,
                    {
                        loader: "css-loader",
                        options: {
                            importLoaders: 2
                        }
                    },
                    "postcss-loader",
                    "sass-loader"
                ]
            }
        ]
    }
}
```

CSS中的图片与字体文件的处理

当在CSS中使用了图片或字体文件以后，也需要进行处理，这个时候它要使用第三方的包

```
$ yarn add file-loader url-loader --dev
```

开始配置

```
const config = {
    //省略部分代码
    module:{
        rules:[
            //配置图片
            {
                test:/\.(jpe?g|png|svg|bmp|gif|ico|tif)$/,
                use:[
                    {
                        loader:"url-loader",
                        options:{
                            //这里小于8KB的就会被转换成base64
                            limit: 8 * 1024,
                            name:'[name].[hash:8].[ext]',
                            outputPath:"img/",
                            publicPath:"../img"
                        }
                    }
```

```
                ]
            },
            //配置字体
            {
                test:/\.(ttf|eot|woff|woff2)$/,
                use:[
                    loader:"url-loader",
                        options:{
                            limit: 10,
                            name:'[name].[hash:8].[ext]',
                            outputPath:"fonts/",
                            publicPath:"../fonts"
                        }
                ]
            }
        ]
    }
}
```

配置打包进度条

因为打包的速度非常慢，我们希望看到打包的时候可以看到进度，这个时候就可以使用进度条的插件

```
$ yarn add chalk progress-bar-webpack-plugin --dev
```

接下来进行配置

```
//导入打包进度条的插件
const ProgressBarWebpackPlugin = require("progress-bar-webpack-plugin");
//导入画笔包，用于设置进度条文字的颜色
const chalk = require("chalk");
const config ={
    //省略部分代码
    plugins:[
        //省略部分代码
        new ProgressBarWebpackPlugin({
            format:chalk.green("进度：")+chalk.white("
[:bar]")+chalk.green(":percent")+chalk.red(" (:elapsed seconds) "),
            clear:false
        })
    ]
}
```

启动

当所有的配置都进行完成以后，我们只需要在`package.json`中的`scripts`里面添加启动命令

```
"scripts":{
    "build":"webpack --config ./webpack.config.js"
}
```

2. 开发模式

大多数的配置都是差不多的，只有少部分的配置是不一样的

1. 关于CSS的配置，它不需要进行分离了，直接使用`style-loader`插入到页面的`style`标签中去

2. 关于图片与字体，它不用再做limit的限制了，它全部转base64直接展示
3. 它需要使用`webpack-dev-server`来启动一个服务器调试页面
4. 它可以配置热模块替换

首先，我们在当前的项目下面新建一个文件叫`webpack.config.dev.js`,配置如下

```
//开发者模式下面的webpack配置
const path = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
const ProgressBarWebpackPlugin = require("progress-bar-webpack-plugin");
const chalk = require("chalk");
//导入webpack插件，配置热模块替换
const webpack = require("webpack");

const config = {
    mode: "development",
    entry: path.join(__dirname, "./js/Student.js"),
    output: {
        filename: "bundle.js",
        path: path.join(__dirname, "./dist")
        //这里千万不要配publicPath,开发者模式不允许配置这个
    },
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /node_modules/,
                loader: "babel-loader"
            }, {
                test: /\.css$/,
                use: [
                    //在开发者模式下面，是不需要进行CSS样式分离的，所以不用调用分离的插
                    //而直接应该插入在style标签中
                    "style-loader",
                    {
                        loader: "css-loader",
                        options: {
                            importLoaders: 1
                        }
                    },
                    "postcss-loader"
                ]
            }, {
                test: /\.s(c|a)ss$/,
                use: [
                    "style-loader",
                    {
                        loader: "css-loader",
                        options: {
                            importLoaders: 1
                        }
                    },
                    "postcss-loader",
                    "sass-loader"
                ]
```

```
            }, {
                test: /\.(jpe?g|gif|png|bmp|svg|ico|tif)$/,
                use: [
                    //在开发者模式下面，不需要考虑是否转换成文件或base64,全部调用base64
                    "url-loader"
                ]
            }, {
                //解析字体
                test: /\.(ttf|eot|woff|woff2)$/,
                use: ["url-loader"]
            }
        ]
    },
    plugins: [
        //清理内容的插件
        new CleanWebpackPlugin(),
        new HtmlWebpackPlugin({
            filename: "index.html",
            template: path.join(__dirname, "./index.html"),
            inject: true
        }),
        new ProgressBarWebpackPlugin({
            format: chalk.green("进度") + chalk.white("[:bar]") + chalk.green(":percent") + chalk.red(" :elapsed seconds"),
            clear: false
        }),

        //热模块替换插件
        new webpack.NamedModulesPlugin(),
        new webpack.HotModuleReplacementPlugin(),
    ],
    //进一步配置服务器webpack-dev-server
    devServer: {
        //服务器启动的端口号
        port: 9999,
        //生成的文件在dist目录下面
        contentBase: "./dist",
        // 启动成功以后直接打开浏览器
        open: true,
        //开启热模块替换
        hot: true
    }
}
module.exports = config;
```

启动

在package.json的文件的`scripts`下面添加如下启动命令

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack --config ./webpack.config.js",
    "dev": "webpack-dev-server --config ./webpack.config.dev.js"
},
```

上面的`dev`就是开发模式的启动命令

五、扩展

如果在webapck当中使用了第三方的插件，如`jQuery`或`bootstrap`则一定要注意，要进行`jquery`全局别名处理

```
const webpack = require("webpack");
const config = {
    plugins:[
         //配置全局别名
        new webpack.ProvidePlugin({
            "jquery": "jquery",
            "$": "jquery",
            "jQuery": "jquery"
        })
    ]
}
```

当配置完上面的东西以后，我们就可以直接使用jQuery以及jQuery的第三方插件了

```
import $ from "jquery";

// 根据bootstrap的package.json配置，导入的是dist/js/npm.js
import "bootstrap";
import "bootstrap/dist/css/bootstrap.css";
```

六、扩展ESModule

现在的webpack是支持ES6的模块化的，所以我们就根据目前的情况来具体的学习ESModule

AppConfig.js文件，代码如下

```
// default是默认导出的意思
export default function abc(){
    console.log("我是abc");
}
```

index.js文件，代码如下

```
import abc from "./AppConfig";
abc();
```

这个时候会正常的运行代码，思考一下，这里的`default`是什么意思

export default是指默认导出对象，后面的import代表导入默认对象

现在的问题就是，如果我们要导入多个变量出去，怎么办呢?

AppConfig.js文件

```
export const a = "张三";
export const b = "李四";
// default本身是关键字，后面不能再接关键字 ，所以不用接const
const c = "王五";
export default c;
```

index.js文件

```
import c, { a, b } from "./AppConfig";
console.log(a, b);
console.log(c);
```

这个时候我们其实已经发现了规律，加了`default`的我们直接导入，而没有加入`default`的则需要解构

注意：每个JS文件里面，只能有一个默认导出，所以`export default`只能出现一次，并且default的后面不能再跟其它的定义的型的关键字了，class/function除外

ES6的模块化是可以根`CommonJS`混合使用的

a.js文件的代码

```
function sayHello(){
    console.log("大家好啊，我用CommonJS导出的");
}
function abc(){
    console.log("我是abc");
}
module.exports = {
    sayHello,abc
}
```

index.js文件的代码

```
import {sayHello,abc} from "./a";
abc();
sayHello();
```

> 通过`CommonJS`模块化解构导出的对象，是可以在import里面再次解构的，但是export default则不行

这里是使用CommonJS导出，ESModule导入

同时它们还可以反过来使用

a.js的文件代码

```
export default function sayHello(){
    console.log("我是ESMdoule导出的");
}
export const userName = "标哥";
```

index.js文件的代码

```
const sayHello = require("./a").default;
const userName = require("./a").userName;
sayHello();
console.log(userName);
```

这里是使用ESModule导出，CommonJS导入