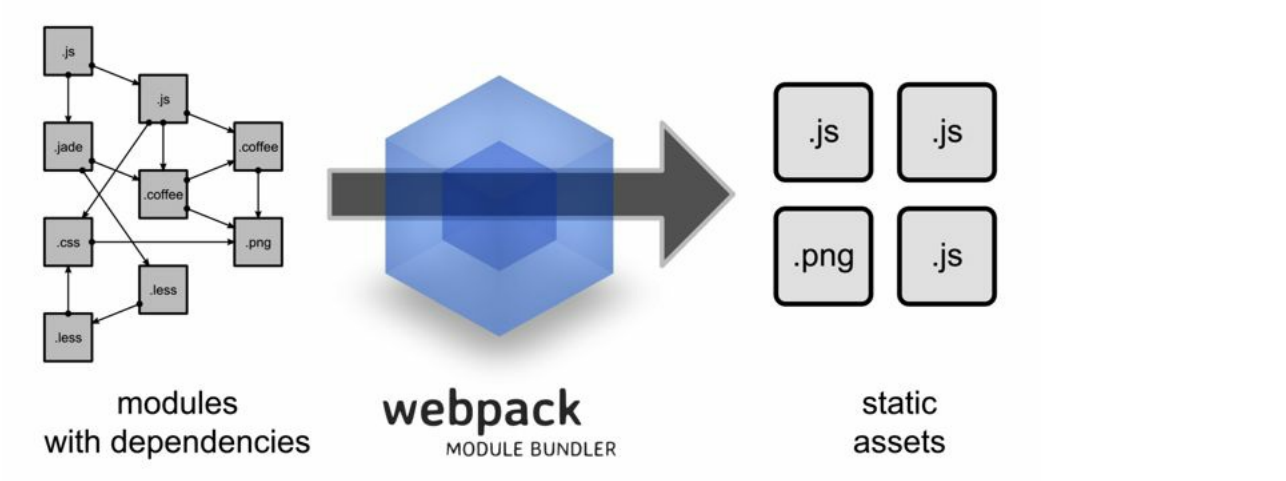
link: null
title: 珠峰架构师成长计划
description: WebPack可以看做是模块打包机：它做的事情是，分析你的项目结构，找到JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Scss，TypeScript等），并将其打包为合适的格式以供浏览器使用。
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=534 sentences=608, words=4252

1. 什么是WebPack

WebPack可以看做是模块打包机：它做的事情是，分析你的项目结构，找到JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Scss，TypeScript等），并将其打包为合适的格式以供浏览器使用。



构建就是把源代码转换成发布到线上的可执行 JavaScrip、CSS、HTML 代码，包括如下内容。

- 代码转换：TypeScript 编译成 JavaScript、SCSS 编译成 CSS 等。
- 文件优化：压缩 JavaScript、CSS、HTML 代码，压缩合并图片等。
- 代码分割：提取多个页面的公共代码、提取首屏不需要执行部分的代码让其异步加载。
- 模块合并：在采用模块化的项目里会有很多个模块和文件，需要构建功能把模块分类合并成一个文件。
- 自动刷新：监听本地源代码的变化，自动重新构建、刷新浏览器。
- 代码校验：在代码被提交到仓库前需要校验代码是否符合规范，以及单元测试是否通过。
- 自动发布：更新完代码后，自动构建出线上发布代码并传输给发布系统。

构建其实是工程化、自动化思想在前端开发中的体现，把一系列流程用代码去实现，让代码自动化地执行这一系列复杂的流程。 构建给前端开发注入了更大的活力，解放了我们的生产力。

2. 初始化项目

```
mkdir zhufeng-webpack
cd zhufeng-webpack
npm init -y
```

3. 快速上手

3.1 webpack核心概念

- Entry：入口，Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入。
- Module：模块，在 Webpack 里一切皆模块，一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。
- Chunk：代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。
- Loader：模块转换器，用于把模块原内容按照需求转换成新内容。
- Plugin：扩展插件，在 Webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要的事情。
- Output：输出结果，在 Webpack 经过一系列处理并得出最终想要的代码后输出结果。
- context: context即是项目打包的路径上下文，如果指定了context,那么entry和output都是相对于上下文路径的，contex必须是一个绝对路径

> Webpack 启动后会从 `Entry`里配置的 `Module`开始递归解析 Entry 依赖的所有 Module。 每找到一个 Module， 就会根据配置的 `Loader`去找出对应的转换规则，对 Module 进行转换后，再解析出当前 Module 依赖的 Module。 这些模块会以 Entry 为单位进行分组，一个 Entry 和其所有依赖的 Module 被分到一个组也就是一个 `Chunk`。最后 Webpack 会把所有 Chunk 转换成文件输出。 在整个流程中 Webpack 会在恰当的时机执行 Plugin 里定义的逻辑。

3.2 配置webpack

```
npm install webpack webpack-cli -D
```

3.2.1 创建src目录

```
mkdir src
```

3.2.2 创建dist目录

```
mkdir dist
```

3.2.3 基本配置文件

webpack.config.js

```
const path=require('path');
module.exports={
  context:process.cwd(),
    entry: './src/index.js',
    output: {
        path: path.resolve(__dirname,'dist'),
        filename:'bundle.js'
    },
    module: {},
    plugins: [],
    devServer: {}
}
```

- 创建dist
    - 创建index.html
- 配置文件webpack.config.js
    - entry：配置入口文件的地址
    - output：配置出口文件的地址
    - module：配置模块,主要用来配置不同文件的加载器
    - plugins：配置插件
    - devServer：配置开发服务器

```
const path=require('path');
module.exports={
    entry: './src/index.js',
    output: {
        path: path.resolve(__dirname,'dist'),
        filename:'bundle.js'
    },
    module: {},
    plugins: [],
    devServer: {}
}
```

3.2.4 创建index.html文件

在dist目录下创建index.html文件

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Documenttitle>
head>
<body>
<div id="root">div>
<script src="bundle.js">script>
body>
html>
```

3.2.5 mode

- webpack的mode (https://webpack.js.org/configuration/mode/#root)配置用于提供模式配置选项告诉webpack相应地使用其内置的优化，mode有以下三个可选值
- development
- production
- none

common

```
optimization.removeAvailableModules:true

optimization.removeEmptyChunks:true

optimization.mergeDuplicateChunks:true
```

development

```
devtool:eval

cache:true

module.unsafeCache:true

output.pathinfo:true

optimization.providedExports:true

optimization.splitChunks:true

optimization.runtimeChunk:true

optimization.noEmitOnErrors:true

optimization.namedModules:true

optimization.namedChunks:true
```

production

```
performance:{hints:"error"....}

optimization.flagIncludedChunks:true

optimization.occurrenceOrder:true

optimization.usedExports:true

optimization.sideEffects:true

optimization.concatenateModules:true

optimization.minimize:true
```

4. 配置开发服务器

```
npm i webpack-dev-server -D
```

- contentBase 配置开发服务运行时的文件根目录
- host：开发服务器监听的主机地址
- compress 开发服务器是否启动gzip等压缩
- port：开发服务器监听的端口

```
+ devServer:{
+        contentBase:path.resolve(__dirname,'dist'),
+        host:'localhost',
+        compress:true,
+        port:8080
+ }
```

```
+  "scripts": {
+    "build": "webpack",
+    "dev": "webpack-dev-server --open "
+  }
```

5. 支持加载css文件

5.1 什么是Loader

通过使用不同的Loader，Webpack可以要把不同的文件都转成JS文件,比如CSS、ES6/7、JSX等

- test：匹配处理文件的扩展名的正则表达式
- use：loader名称，就是你要使用模块的名称
- include/exclude:手动指定必须处理的文件夹或屏蔽不需要处理的文件夹
- query：为loaders提供额外的设置选项

5.2 loader三种写法

- css-loader (https://www.npmjs.com/package/css-loader)
- style-loader (https://www.npmjs.com/package/style-loader)

5.2.1 loader

加载CSS文件，CSS文件有可能在node_modules里，比如bootstrap和antd

```
module: {
        rules: [
            {
                test: /\.css/,
+                loader:['style-loader','css-loader']
            }
        ]
    }
```

5.2.2 use

```
module: {
        rules: [
            {
                test: /\.css/,
+                use:['style-loader','css-loader']
            }
        ]
    },
```

5.2.3 use+loader

```
    module: {
        rules: [
            {
                test: /\.css/,
                include: path.resolve(__dirname,'src'),
                exclude: /node_modules/,
                use: [{
                    loader: 'style-loader',
                    options: {
                        insert:'top'
                    }
                },'css-loader']
            }
        ]
    }
```

6. 插件

- 在 webpack 的构建流程中，plugin 用于处理更多其他的一些构建任务
- 模块代码转换的工作由 loader 来处理
- 除此之外的其他任何工作都可以交由 plugin 来完成

6.1 自动产出html

- 我们希望自动能产出HTML文件，并在里面引入产出后的资源
- chunksSortMode (https://github.com/jaketrent/html-webpack-template/blob/86f285d5c790a6c15263f5cc50fd666d51f974fd/index.html)还可以控制引入的顺序

```
cnpm i html-webpack-plugin -D
```

- minify 是对html文件进行压缩，removeAttrubuteQuotes是去掉属性的双引号
- hash 引入产出资源的时候加上查询参数，值为哈希避免缓存
- template 模版路径

```
+    +entry:{
+        index:'./src/index.js',  // chunk名字 index
+        common:'./src/common.js' //chunk名字 common
+    },

    plugins: [
+       new HtmlWebpackPlugin({
+           template:'./src/index.html',//指定模板文件
+           filename:'index.html',//产出后的文件名
+           inject:false,
+           hash:true,//为了避免缓存，可以在产出的资源后面添加hash值
+           chunks:['common','index'],
+           chunksSortMode:'manual'//对引入代码块进行排序的模式
+       }),
    )]
```

```
+
+
+

+
+
+
```

7. 支持图片

7.1 手动添加图片

```
npm i file-loader url-loader -D
```

- file-loader (http://npmjs.com/package/file-loader) 解决CSS等文件中的引入图片路径问题
- url-loader (https://www.npmjs.com/package/url-loader) 当图片小于limit的时候会把图片BASE64编码，大于limit参数的时候还是使用file-loader 进行拷贝

7.2 JS中引入图片

7.2.1 JS

```
let logo=require('./images/logo.png');
let img=new Image();
img.src=logo;
document.body.appendChild(img);
```

7.2.2 webpack.config.js

```
{
  test:/\.(jpg|png|bmp|gif|svg)/,
    use:[
    {
       loader:'url-loader',
       options:{limit:4096}
    }
  ]
}
```

7.3 在CSS中引入图片

还可以在CSS文件中引入图片

7.3.1 CSS

```
.logo{
    width:355px;
    height:133px;
    background-image: url(./images/logo.png);
    background-size: cover;
}
```

7.3.2 HTML

```
<div class="logo">div>
```

8. 分离CSS

因为CSS的下载和JS可以并行,当一个HTML文件很大的时候，我们可以把CSS单独提取出来加载

- mini-css-extract-plugin (https://github.com/webpack-contrib/mini-css-extract-plugin)
- filename 打包入口文件
- chunkFilename 用来打包 `import('module')`方法中引入的模块

8.1 安装依赖模块

```
npm install --save-dev mini-css-extract-plugin
```

8.2 配置webpack.config.js

```
plugins: [
       //参数类似于webpackOptions.output
+        new MiniCssExtractPlugin({
+            filename: '[name].css',
+            chunkFilename:'[id].css'
+        }),

{
                test: /\.css/,
                include: path.resolve(__dirname,'src'),
                exclude: /node_modules/,
                use: [{
+                    loader: MiniCssExtractPlugin.loader
                },'css-loader']
           }
```

8.3 内联CSS

- 注意此插件要放在 HtmlWebpackPlugin的下面
- HtmlWebpackPlugin的 inject设置为 true

```
cnpm i html-inline-css-webpack-plugin -D
```

```
+const HtmlInlineCssWebpackPlugin= require('html-inline-css-webpack-plugin').default;

plugins:[
  new HtmlWebpackPlugin({}),
+  new HtmlInlineCssWebpackPlugin()
]
```

8.4 压缩JS和CSS

- 用 terser-webpack-plugin替换掉 uglifyjs-webpack-plugin解决uglifyjs不支持es6语法问题

```
cnpm i uglifyjs-webpack-plugin terser-webpack-plugin optimize-css-assets-webpack-plugin -D
```

```
const UglifyJsPlugin = require("uglifyjs-webpack-plugin");
const TerserPlugin = require('terser-webpack-plugin');
const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");
module.exports = {
    mode: 'production',
    optimization: {
        minimizer: [

            new TerserPlugin({
                 parallel: true,
                 cache: true
            }),

            new OptimizeCSSAssetsPlugin({
                 assetNameRegExp:/\.css$/g,

                 cssProcessor:require('cssnano')
            })
        ]
    },
```

8.5 css和image存放单独目录

- 去掉 HtmlInlineCssWebpackPlugin
- outputPath 输出路径
- publicPath指定的是构建后在html里的路径
- 如果在CSS文件中引入图片，而图片放在了image目录下，就需要配置图片的publicPath为 /images,或者

```
{
   loader:MiniCssExtractPlugin.loader,
      options:{
+        publicPath:'/'
      }
```

```
{
  test:/\.(jpg|jpeg|png|bmp|gif|svg|ttf|woff|woff2|eot)/,
  use:[
        {
          loader:'url-loader',
          options:{
              limit: 4096,
+              outputPath: 'images',
+              publicPath:'/images'
          }
        }
     ]
}
```

```
output: {
        path: path.resolve(__dirname,'dist'),
        filename: 'bundle.js',
+        publicPath:'/'
    },
{
  test:/\.(jpg|jpeg|png|bmp|gif|svg|ttf|woff|woff2|eot)/,
  use:[
        {
          loader:'url-loader',
          options:{
              limit: 4096,
+              outputPath: 'images',
+              publicPath:'/images'
          }
        }
     ]
}

plugins: [
    new MiniCssExtractPlugin({
-       //filename: '[name].css',
-       //chunkFilename: '[id].css',
+       chunkFilename: 'css/[id].css',
+       filename: 'css/[name].[hash].[chunkhash].[contenthash].css',//name是代码码chunk的名字
    }),
```

8.6 文件指纹

- 打包后输出的文件名和后缀
- hash一般是结合CDN缓存来使用，通过webpack构建之后，生成对应文件名自动带上对应的MD5值。如果文件内容改变的话，那么对应文件哈希值也会改变，对应的HTML引用的URL地址也会改变，触发CDN服务器从源服务器上拉取对应数据，进而更新本地缓存。

8.6.1 文件指纹如何生成

- Hash 是整个项目的hash值，其根据每次编译内容计算得到，每次编译之后都会生成新的hash,即修改任何文件都会导致所有文件的hash发生改变，在一个项目中虽然入口不同，但是hash是相同的，hash无法实现前端静态资源的浏览器长缓存，如果有这个需求应该使用chunkhash
- chunkhash 采用hash计算的话，每一次构建后生成的哈希值都不一样，即使文件内容压根没有改变。这样子是没办法实现缓存效果，我们需要换另一种哈希值计算方式，即chunkhash,chunkhash和hash不一样，它根据不同的入口文件(Entry)进行依赖文件解析、构建对应的chunk，生成对应的哈希值。我们在生产环境里把一些公共库和程序入口文件区分开，单独打包构建，接着我们采用chunkhash的方式生成哈希值，那么只要我们不改动公共库的代码，就可以保证其哈希值不会受影响
- contenthash 使用chunkhash存在一个问题，就是当在一个JS文件中引入CSS文件，编译后它们的hash是相同的，而且只要js文件发生改变 ，关联的css文件hash也会改变,这个时候可以使用 `mini-css-extract-plugin`里的 `contenthash`值，保证即使css文件所处的模块里就算其他文件内容改变，只要css文件内容不变，那么不会重复构建

指纹占位符

占位符名称 含义 ext 资源后缀名 name 文件名称 path 文件的相对路径 folder 文件所在的文件夹 contenthash 文件的内容hash,默认是md5生成 hash 文件内容的hash,默认是md5生成 emoji 一个随机的指代文件内容的emoj

9. 编译less 和 sass

9.1 安装less

```
npm i less less-loader -D
npm i node-sass sass-loader -D
```

9.2 编写样式

less

```
@color:red;
.less-container{
    color:@color;
}
```

scss

```
$color:green;
.sass-container{
    color:$color;
}
```

webpack.config.js

```
  {
      test: /\.less/,
      include: path.resolve(__dirname,'src'),
      exclude: /node_modules/,
      use: [{
          loader: MiniCssExtractPlugin.loader,
      },'css-loader','less-loader']
  },
  {
      test: /\.scss/,
      include: path.resolve(__dirname,'src'),
      exclude: /node_modules/,
      use: [{
          loader: MiniCssExtractPlugin.loader,
      },'css-loader','sass-loader']
  },
```

10. 处理CSS3属性前缀

- 为了浏览器的兼容性，有时候我们必须加入-webkit,-ms,-o,-moz这些前缀
    - Trident内核：主要代表为IE浏览器, 前缀为-ms
    - Gecko内核：主要代表为Firefox, 前缀为-moz
    - Presto内核：主要代表为Opera, 前缀为-o
    - Webkit内核：产要代表为Chrome和Safari, 前缀为-webkit
- caniuse (https://caniuse.com)

```
npm i postcss-loader autoprefixer -D
```

- PostCSS 的主要功能只有两个

- 第一个就是前面提到的把 CSS 解析成 JavaScript 可以操作的 抽象语法树结构(Abstract Syntax Tree，AST)
- 第二个就是调用插件来处理 AST 并得到结果

postcss-loader (https://github.com/postcss/postcss-loader)

index.css

```
::placeholder {
    color: red;
}
```

postcss.config.js

```
module.exports={
    plugins:[require('autoprefixer')]
}
```

webpack.config.js

```
{
   test:/\.css$/,
   use:[MiniCssExtractPlugin.loader,'css-loader','postcss-loader'],
   include:path.join(__dirname,'./src'),
   exclude:/node_modules/
}
```

11. 转义ES6/ES7/JSX

- Babel其实是一个编译JavaScript的平台,可以把ES6/ES7,React的JSX转义为ES5
- babel-plugin-proposal-decorators (https://babeljs.io/docs/en/babel-plugin-proposal-decorators)

11.1 安装依赖包

```
npm i babel-loader @babel/core @babel/preset-env  @babel/preset-react  -D
npm i @babel/plugin-proposal-decorators @babel/plugin-proposal-class-properties -D
```

11.2 decorator

```

function readonly(target,key,discriptor) {
    discriptor.writable=false;
}

class Person{
    @readonly PI=3.14;
}
let p1=new Person();
p1.PI=3.15;
console.log(p1)
```

jsconfig.json

```
{
    "compilerOptions": {
        "experimentalDecorators": true
    }
}
```

11.3 webpack.config.js

```
{
    test: /\.jsx?$/,
    use: {
        loader: 'babel-loader',
        options:{
         "presets": ["@babel/preset-env"],
         "plugins": [
            ["@babel/plugin-proposal-decorators", { "legacy": true }],
            ["@babel/plugin-proposal-class-properties", { "loose" : true }]
         ]
        }
    },
    include: path.join(__dirname,'src'),
    exclude:/node_modules/
}
```

.babelrc

```
{
  "presets": ["@babel/preset-env"],
  "plugins": [
    ["@babel/plugin-proposal-decorators", { "legacy": true }],
    ["@babel/plugin-proposal-class-properties", { "loose" : true }]
  ]
}
```

11.4 babel runtime

- babel 在每个文件都插入了辅助代码，使代码体积过大
- babel 对一些公共方法使用了非常小的辅助代码，比如 _extend
- 默认情况下会被添加到每一个需要它的文件中。你可以引入 @babel/runtime 作为一个独立模块，来避免重复引入
- babel-plugin-transform-runtime (https://babeljs.io/docs/en/babel-plugin-transform-runtime)
- babel-plugin-proposal-decorators (https://babeljs.io/docs/en/babel-plugin-proposal-decorators)
- babel-plugin-proposal-class-properties (https://babeljs.io/docs/en/babel-plugin-proposal-class-properties)
- loose为true的时候,属性是直接赋值,loose为false的时候会使用 Object.defineProperty
- @babel/preset-env 中的 useBuiltIns 选项，如果你设置了 usage，babel 编绎的时候就不用整个 polyfills，只加载你使用 polyfills，这样就可以减少包的大小
- @babel/plugin-transform-runtime 是开发时引入, @babel/runtime 是运行时引用
- plugin-transform-runtime 已经默认包括了 @babel/polyfill，因此不用在独立引入
- corejs 是一个给低版本的浏览器提供接口的库，如 Promise、Map和Set 等
- 在 babel 中你设置成 false 或者不设置，就是引入的是 corejs 中的库，而且在全局中引入，也就是说侵入了全局的变量

```
npm install --save-dev @babel/plugin-transform-runtime
npm install --save @babel/runtime
```

.babelrc

```
{
  "presets": ["@babel/preset-env"],
  "plugins": [
    ["@babel/plugin-proposal-decorators", { "legacy": true }],
    ["@babel/plugin-proposal-class-properties", { "loose" : true }]
    [
         "@babel/plugin-transform-runtime",
         {
            "corejs": false,
            "helpers": true,
            "regenerator": true,
            "useESModules": true
        }
    ]
  ]
}
```

> webpack打包的时候，会自动优化重复引入公共方法的问题

11.4.1 区别

- 区别 (https://www.arayzou.com/2019/10/15/plugin-transform-runtime%E4%B8%8Epolyfill/)

11.5 ESLint校验代码格式规范

- eslint (https://eslint.org/docs/developer-guide/nodejs-api#cliengine)
- eslint-loader (https://www.npmjs.com/package/eslint-loader)
- configuring (https://eslint.org/docs/user-guide/configuring)
- babel-eslint (https://www.npmjs.com/package/babel-eslint)
- Rules (https://cloud.tencent.com/developer/chapter/12618)
- ESlint 语法检测配置说明 (https://segmentfault.com/a/1190000008742240)

11.5.1 标准配置

- 建议制定团队的eslint规范
- 基于eslint:recommend配置进行改进
- 发现代码错误的规则尽可能多的开启
- 帮助保持团队的代码风格统一而不要限制开发体验

```
npm install eslint eslint-loader babel-eslint --D
```

.eslintrc.js

```
module.exports = {
    root: true,

    parserOptions: {
        sourceType: 'module',
        ecmaVersion: 2015
    },

    env: {
        browser: true,
    },

    rules: {
        "indent": ["error", 4],
        "quotes": ["error", "double"],
        "semi": ["error", "always"],
        "no-console": "error",
        "arrow-parens": 0
    }
}
```

webpack.config.js

```
module: {

        rules: [
            {
                test: /\.js$/,
                loader: 'eslint-loader',
                enforce: "pre",
                include: [path.resolve(__dirname, 'src')],
                options: { fix: true }
            },
```

11.5.2 继承airbnb

- eslint-config-airbnb (https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb)

```
cnpm i eslint-config-airbnb eslint-loader eslint eslint-plugin-import eslint-plugin-react eslint-plugin-react-hooks and eslint-plugin-jsx-a11y -D
```

.eslintrc.js

```
module.exports = {
    "parser":"babel-eslint",
    "extends":"airbnb",
    "rules":{
        "semi":"error",
        "no-console":"off",
        "linebreak-style":"off",
        "eol-last":"off"

    },
    "env":{
        "browser":true,
        "node":true
    }
}
```

11.6 引入字体

- OTF—— opentype 苹果机与PC机都能很好应用的兼容字体

- HabanoST (http://img.zhufengpeixun.cn/HabanoST.otf)

11.6.1 配置loader

```
{
 test:/\.(woff|ttf|eot|svg|otf)$/,
     use:{

        loader:'url-loader',
        options:{
       limit:10*1024
        }
   }
 },
```

11.6.2 使用字体

```
@font-face {
    src: url('./fonts/HabanoST.otf') format('truetype');
    font-family: 'HabanoST';
}

.welcome {
    font-size:100px;
    font-family: 'HabanoST';
}
```

12. 如何调试打包后的代码

- sourcemap是为了解决开发代码与实际运行代码不一致时帮助我们debug到原始开发代码的技术
- webpack通过配置可以自动给我们 source maps文件， map文件是一种对应编译文件和源文件的方法
- whyeval (https://github.com/webpack/docs/wiki/build-performance#sourcemaps)
- source-map (https://github.com/mozilla/source-map)
- javascript_source_map算法 (http://www.ruanyifeng.com/blog/2013/01/javascript_source_map.html)

类型 含义 source-map 原始代码 最好的sourcemap质量有完整的结果，但是会很慢 eval-source-map 原始代码 同样道理，但是最高的质量和最低的性能 cheap-module-eval-source-map 原始代码（只有行内） 同样道理，但是更高的质量和更低的性能 cheap-eval-source-map 转换代码（行内） 每个模块被eval执行，并且sourcemap作为eval的一个dataurl eval 生成代码 每个模块都被eval执行，并且存在@sourceURL,带eval的构建模式能cache SourceMap cheap-source-map 转换代码（行内） 生成的sourcemap没有列映射，从loaders生成的sourcemap没有被使用 cheap-module-source-map 原始代码（只有行内） 与上面一样除了每行特点的从loader中进行映射

看似配置项很多， 其实只是五个关键字eval、source-map、cheap、module和inline的任意组合

关键字 含义 eval 使用eval包裹模块代码 source-map 产生.map文件 cheap 不包含列信息（关于列信息的解释下面会有详细介绍)也不包含loader的sourcemap module 包含loader的sourcemap（比如jsx to js，babel的sourcemap）,否则无法定义源文件 inline 将.map作为DataURI嵌入，不单独生成.map文件

- eval eval执行
- eval-source-map 生成sourcemap
- cheap-module-eval-source-map 不包含列
- cheap-eval-source-map 无法看到真正的源码

12.1 sourcemap

- compiler官方下载 (https://developers.google.com/closure/compiler)
- compiler蜂蜂镜像 (http://img.zhufengpeixun.cn/compiler.jar)

12.1 生成sourcemap

script.js

```
let a=1;
let b=2;
let c=3;
```

```
java -jar compiler.jar --js script.js --create_source_map ./script-min.js.map --source_map_format=V3 --js_output_file script-min.js
```

script-min.js

```
var a=1,b=2,c=3;
```

script-min.js.map

```
{
"version":3,
"file":"script-min.js",
"lineCount":1,
"mappings":"AAAA,IAAIA,EAAE,CAAN,CAACC,EAAE,CAAHD,CAAKE,EAAE,CAAP;",
"sources":["script.js"],
"names":["a","b","c"]
}
```

字段 含义 version：Source Source map的版本，目前为3 file：转换后的文件名。 转换后的文件名 sourceRoot 转换前的文件所在的目录。如果与转换前的文件在同一目录，该项为空。 sources 转换前的文件。该项是一个数组，表示可能存在多个文件合并。 names 转换前的所有变量名和属性名 mappings 记录位置信息的字符串

12.2 mappings属性

- 关键就是map文件的mappings属性。这是一个很长的字符串，它分成三层

对应 含义 第一层是行对应 以分号（;）表示，每个分号对应转换后源码的一行。所以，第一个分号前的内容，就对应源码的第一行，以此类推。 第二层是位置对应 以逗号（,）表示，每个逗号对应转换后源码的一个位置。所以，第一个逗号前的内容，就对应该行源码的第一个位置，以此类推。 第三层是位置转换 以VLQ编码表示，代表该位置对应的转换前的源码位置。

```
"mappings":"AAAA,IAAIA,EAAE,CAAN,CAACC,EAAE,CAAHD,CAAKE,EAAE,CAAP;",
```

12.3 位置对应的原理

- 每个位置使用五位，表示五个字段

位置 含义 第一位 表示这个位置在（转换后的代码的）的第几列 第二位 表示这个位置属于sources属性中的哪一个文件 第三位 表示这个位置属于转换前代码的第几行 第四位 表示这个位置属于转换前代码的第几 第五位 表示这个位置属于names属性中的哪一个变量

> 首先，所有的值都是以0作为基数的。其次，第五位不是必需的，如果该位置没有对应names属性中的变量，可以省略第五位,再次，每一位都采用VLQ编码表示；由于VLQ编码是变长的，所以每一位可以由多个字符构成

> 如果某个位置是AAAAA，由于A在VLQ编码中表示0，因此这个位置的五个位实际上都是0。它的意思是，该位置在转换后代码的第0列，对应sources属性中第0个文件，属于转换前代码的第0行第0列，对应names属性中的第0个变量。

12.4 VLQ编码

- VLQ 是 Variable-length quantity 的缩写,它的特点就是可以非常精简地表示很大的数值
- VLQ编码是变长的。如果（整）数值在-15到+15之间（含两个端点），用一个字符表示；超出这个范围，就需要用多个字符表示。它规定，每个字符使用6个两进制位，正好可以借用Base 64编码的字符表

- 在这6个位中，左边的第一位（最高位）表示是否"连续"（continuation）。如果是1，代表这 6 个位后面的6个位也属于同一个数；如果是0，表示该数值到这6个位结束。
- 这6个位中的右边最后一位（最低位）的含义，取决于这6个位是否是某个数值的VLQ编码的第一个字符。如果是的，这个位代表"符号"（sign），0为正，1为负（Source map的符号固定为0）；如果不是，这个位没有特殊含义，被算作数值的一部分。

base64vlq在线转换 (http://murzwin.com/base64vlq.html)

以16来做示例吧

1. 将16改写成二进制形式10000
2. 在最右边补充符号位。因为16大于0，所以符号位为0，整个数变成100000
3. 从右边的最低位开始，将整个数每隔5位，进行分段，即变成1和00000两段。如果最高位所在的段不足5位，则前面补0，因此两段变成00001和00000
4. 将两段的顺序倒过来，即00000和00001
5. 在每一段的最前面添加一个"连续位"，除了最后一段为0，其他都为1，即变成100000和000001
6. 将每一段转成Base 64编码。
7. 查表可知，100000为g，000001为B。因此，数值16的VLQ编码为gB

```
let base64 = [
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
    'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
    'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
    'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '/'
];

function encode(num) {
    debugger;
    let binary = (num).toString(2);
    binary = num > 0 ? binary + '0' : binary + '1';

    let zero = 5 - (binary.length % 5);
    if (zero > 0) {
        binary = binary.padStart(Math.ceil(binary.length / 5) * 5, '0');
    }
    let parts = [];
    for (let i = 0; i < binary.length; i += 5) {
        parts.push(binary.slice(i, i + 5));
    }
    parts.reverse();
    for (let i = 0; i < parts.length; i++) {
        if (i === parts.length - 1) {
            parts[i] = '0' + parts[i];
        } else {
            parts[i] = '1' + parts[i];
        }
    }
    let chars = [];
    for (let i = 0; i < parts.length; i++) {
        chars.push(base64[parseInt(parts[i], 2)]);
    }
    return chars.join('')
}

let ret = encode(16);
console.log(ret);

function getValue(char) {
    let index = base64.findIndex(item => item == char);
    let str = (index).toString(2);
    str = str.padStart(6, '0');
    str = str.slice(1, -1);
    return parseInt(str, 2);
}

function decode(chars) {
    let values = [];
    for (let i = 0; i < chars.length; i++) {
        values.push(getValue(chars[i]));
    }
    return values;
}
function desc(values) {
    return `
    第${values[1] + 1}个源文件中
    的第1行
    第${values[0] + 1}列,
    对应转换后的第${values[2] + 1}行
    第${values[3] + 1}列,
    对应第${values[4] + 1}个变量`;
}
let ret2 = decode('IAAIA');
let message = desc(ret2);
console.log(ret2, message);
```

13.打包第三方类库

13.1 直接引入

```
import _ from 'lodash';
alert(_.join(['a','b','c'],'@'));
```

13.2 插件引入

- webpack配置ProvidePlugin后，在使用时将不再需要import和require进行引入，直接使用即可
- _ 函数会自动添加到当前模块的上下文，无需显示声明

```
+ new webpack.ProvidePlugin({
+     _:'lodash'
+ })
```

没有全局的 s函数，所以导入依赖全局变量的插件依旧会失败

13.3 expose-loader

- The expose loader adds modules to the global object. This is useful for debugging
- 不需要任何其他的插件配合，只要将下面的代码添加到所有的loader之前

```
require("expose-loader?libraryName!./file.js");
```

```
{
  test: require.resolve("jquery"),
  loader: "expose-loader?jQuery"
}
```

```
require("expose-loader?$!jquery");
```

13.4 externals

如果我们想引用一个库，但是又不想让webpack打包，并且又不影响我们在程序中以CMD、AMD或者window/global全局等方式进行使用，那就可以通过配置externals

```
const jQuery = require("jquery");
import jQuery from 'jquery';
```

```
<script src="https://cdn.bootcss.com/jquery/3.4.1/jquery.js">script>
```

```
+ externals: {
+         jquery: 'jQuery'//如果要在浏览器中运行，那么不用添加什么前缀，默认设置就是global
+ },
module: {
```

13.5 html-webpack-externals-plugin

- 外链CDN

```
+ const htmlWebpackExternalsPlugin= require('html-webpack-externals-plugin');
new htmlWebpackExternalsPlugin({
            externals:[
                {
                    module:'react',
                    entry:'https://cdn.bootcss.com/react/15.6.1/react.js',
                    global:'React'
                },
                 {
                    module:'react-dom',
                    entry:'https://cdn.bootcss.com/react/15.6.1/react-dom.js',
                    global:'ReactDOM'
                }
            ]
})
```

14. watch

当代码发生修改后可以自动重新编译

```
module.exports = {

    watch:true,

    watchOptions:{

        ignored:/node_modules/,

        aggregateTimeout:300,

        poll:1000
    }
}
```

- webpack定时获取文件的更新时间，并跟上次保存的时间进行比对，不一致就表示发生了变化,poll就用来配置每秒问多少次
- 当检测文件不再发生变化，会先缓存起来，等待一段时间后之后再通知监听者，这个等待时间通过 aggregateTimeout配置
- webpack只会监听entry依赖的文件
- 我们需要尽可能减少需要监听的文件数量和检查频率，当然频率的降低会导致灵敏度下降

15. 添加商标

```
+ new webpack.BannerPlugin('珠峰培训'),
```

16. 拷贝静态文件

有时项目中没有引用的文件也需要打包到目标目录

```
npm i copy-webpack-plugin -D
```

```
new CopyWebpackPlugin([{
  from: path.resolve(__dirname,'src/assets'),
  to:path.resolve(__dirname,'dist/assets')
}])
```

17. 打包前先清空输出目录

- clean-webpack-plugin (https://www.npmjs.com/package/clean-webpack-plugin)

```
npm i  clean-webpack-plugin -D
```

```
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
plugins:[
new CleanWebpackPlugin({cleanOnceBeforeBuildPatterns: ['**/*', '!static-files*'],})
]
```

18. 服务器代理

如果你有单独的后端开发服务器 API，并且希望在同域名下发送 API 请求 ，那么代理某些 URL 会很有用。

18.1 不修改路径

- 请求到 /api/users 现在会被代理到请求http://localhost:3000/api/users。 (http://localhost:3000/api/users。)

```
proxy: {
  "/api": 'http://localhost:3000'
}
```

18.2 修改路径

```
proxy: {
    "/api": {
       target: 'http://localhost:3000',
       pathRewrite:{"^/api":""}
    }
}
```

18.3 before after

before 在 webpack-dev-server 静态资源中间件处理之前，可以用于拦截部分请求返回特定内容，或者实现简单的数据 mock。

```
before(app){
  app.get('/api/users', function(req, res) {
    res.json([{id:1,name:'zfpx1'}])
  })
}
```

18.4 webpack-dev-middleware

webpack-dev-middleware (https://www.npmjs.com/package/)就是在 Express 中提供 webpack-dev-server 静态服务能力的一个中间件

```
npm install webpack-dev-middleware --save-dev
```

```
const express = require('express');
const app = express();
const webpack = require('webpack');
const webpackDevMiddleware = require('webpack-dev-middleware');
const webpackOptions = require('./webpack.config');
webpackOptions.mode = 'development';
const compiler = webpack(webpackOptions);
app.use(webpackDevMiddleware(compiler, {}));
app.listen(3000);
```

- webpack-dev-server 的好处是相对简单，直接安装依赖后执行命令即可
- 而使用 webpack-dev-middleware的好处是可以在既有的 Express 代码基础上快速添加 webpack-dev-server 的功能，同时利用 Express 来根据需要添加更多的功能，如 mock 服务、代理 API 请求等

19. resolve解析

19.1 extensions

指定extension之后可以不用在 require或是 import的时候加文件扩展名,会依次尝试添加扩展名进行匹配

```
resolve: {
  extensions: [".js",".jsx",".json",".css"]
},
```

19.2 alias

配置别名可以加快webpack查找模块的速度

- 每当引入bootstrap模块的时候，它会直接引入 bootstrap,而不需要从 node_modules文件夹中按模块的查找规则查找

```
const bootstrap = path.resolve(__dirname,'node_modules/_bootstrap@3.3.7@bootstrap/dist/css/bootstrap.css');
resolve: {
+    alias:{
+        "bootstrap":bootstrap
+    }
},
```

19.3 modules

- 对于直接声明依赖名的模块（如 react ），webpack 会类似 Node.js 一样进行路径搜索，搜索 node_modules目录
- 这个目录就是使用 resolve.modules字段进行配置的 默认配置

```
resolve: {
modules: ['node_modules'],
}
```

如果可以确定项目内所有的第三方依赖模块都是在项目根目录下的 node_modules 中的话

```
resolve: {
modules: [path.resolve(__dirname, 'node_modules')],
}
```

19.4 mainFields

默认情况下package.json 文件则按照文件中 main 字段的文件名来查找文件

```
resolve: {

  mainFields: ['browser', 'module', 'main'],

  mainFields: ["module", "main"],
}
```

19.5 mainFiles

当目录下没有 package.json 文件时，我们说会默认使用目录下的 index.js 这个文件，其实这个也是可以配置的

```
resolve: {
  mainFiles: ['index'],
},
```

19.6 resolveLoader

resolve.resolveLoader用于配置解析 loader 时的 resolve 配置,默认的配置：

```
module.exports = {
  resolveLoader: {
    modules: [ 'node_modules' ],
    extensions: [ '.js', '.json' ],
    mainFields: [ 'loader', 'main' ]
  }
};
```

20. noParse

- module.noParse 字段，可以用于配置哪些模块文件的内容不需要进行解析
- 不需要解析依赖（即无依赖） 的第三方大型类库等，可以通过这个字段来配置，以提高整体的构建速度

```
module.exports = {

module: {
  noParse: /jquery|lodash/,

  noParse(content) {
    return /jquery|lodash/.test(content)
  },
}
}...
```

> 使用 noParse 进行忽略的模块文件中不能使用 import、require、define 等导入机制

21. DefinePlugin

DefinePlugin创建一些在编译时可以配置的全局常量

```
new webpack.DefinePlugin({
    PRODUCTION: JSON.stringify(true),
    VERSION: "1",
    EXPRESSION: "1+2",
    COPYRIGHT: {
        AUTHOR: JSON.stringify("珠峰培训")
    }
})
```

```
console.log(PRODUCTION);
console.log(VERSION);
console.log(EXPRESSION);
console.log(COPYRIGHT);
```

- 如果配置的值是字符串，那么整个字符串会被当成代码片段来执行，其结果作为最终变量的值
- 如果配置的值不是字符串，也不是一个对象字面量，那么该值会被转为一个字符串，如 true，最后的结果是 'true'
- 如果配置的是一个对象字面量，那么该对象的所有 key 会以同样的方式去定义
- JSON.stringify(true) 的结果是 'true'

22. IgnorePlugin

IgnorePlugin用于忽略某些特定的模块，让 webpack 不把这些指定的模块打包进去

```
import moment from  'moment';
console.log(moment);
```

```
new webpack.IgnorePlugin(/^\.\/locale/,/moment$/)
```

- 第一个是匹配引入模块路径的正则表达式
- 第二个是匹配模块的对应上下文，即所在目录名

20. 区分环境变量

- 日常的前端开发工作中，一般都会有两套构建环境
- 一套开发时使用，构建结果用于本地开发调试，不进行代码压缩，打印 debug 信息，包含 sourcemap 文件
- 一套构建后的结果是直接应用于线上的，即代码都是压缩后，运行时不打印 debug 信息，静态文件不包括 sourcemap
- webpack 4.x 版本引入了 mode 的概念
- 当你指定使用 production mode 时，默认会启用各种性能优化的功能，包括构建结果优化以及 webpack 运行性能优化
- 而如果是 development mode 的话，则会开启 debug 工具，运行时打印详细的错误信息，以及更加快速的增量编译构建

20.1 环境差异

- 生产环境
  - 可能需要分离 CSS 成单独的文件，以便多个页面共享同一个 CSS 文件
  - 需要压缩 HTML/CSS/JS 代码
  - 需要压缩图片
- 开发环境
  - 需要生成 sourcemap 文件
  - 需要打印 debug 信息
  - 需要 live reload 或者 hot reload 的功能...

20.2 获取mode参数

```
npm install --save-dev optimize-css-assets-webpack-plugin
```

```
  "scripts": {
+    "dev": "webpack-dev-server --env=development --open"
  },
```

```
const TerserWebpackPlugin = require('terser-webpack-plugin');
const OptimizeCssAssetsPlugin = require('optimize-css-assets-webpack-plugin');
module.exports=(env,argv) => ({
    optimization: {
        minimizer: argv.mode == 'production'?[
            new TerserWebpackPlugin({
               parallel:true,
               cache:true
      }),
            new OptimizeCssAssetsWebpackPlugin({})
        ]:[]
    }
})
```

20.3 封装log方法

- webpack时传递的 mode 参数，是可以在我们的应用代码运行时，通过 process.env.NODE_ENV 这个变量获取

```
export default function log(...args) {
    if (process.env.NODE_ENV == 'development') {
        console.log.apply(console,args);
    }
}
```

20.4 拆分配置

可以把 webpack 的配置按照不同的环境拆分成多个文件，运行时直接根据环境变量加载对应的配置即可

- webpack.base.js：基础部分，即多个文件中共享的配置
- webpack.development.js：开发环境使用的配置
- webpack.production.js：生产环境使用的配置
- webpack.test.js：测试环境使用的配置...
- webpack-merge (https://github.com/survivejs/webpack-merge)

```
const { smart } = require('webpack-merge')
const webpack = require('webpack')
const base = require('./webpack.base.js')
module.exports = smart(base, {
  module: {
    rules: [],
  }
})
```

21. 对图片进行压缩和优化

image-webpack-loader (https://www.npmjs.com/package/image-webpack-loader)可以帮助我们对图片进行压缩和优化

```
npm install image-webpack-loader --save-dev
```

```
 {
         test: /\.(png|svg|jpg|gif|jpeg|ico)$/,
         use: [
           'file-loader',
+          {
+            loader: 'image-webpack-loader',
+            options: {
+              mozjpeg: {
+                progressive: true,
+                quality: 65
+              },
+              optipng: {
+                enabled: false,
+              },
+              pngquant: {
+                quality: '65-90',
+                speed: 4
+              },
+              gifsicle: {
+                interlaced: false,
+              },
+              webp: {
+                quality: 75
+              }
+            }
+          },
         ]
       }
```

22. 多入口MPA#

- 有时候我们的页面可以不止一个HTML页面，会有多个页面，所以就需要多入口
- 每一次页面跳转的时候，后台服务器都会返回一个新的html文档，这种类型的网站就是多页网站，也叫多页应用

```
const path=require('path');
const HtmlWebpackPlugin=require('html-webpack-plugin');
const htmlWebpackPlugins=[];
const glob = require('glob');
const entry={};
const entryFiles = glob.sync('./src/**/index.js');
entryFiles.forEach((entryFile,index)=>{
    let entryName = path.dirname(entryFile).split('/').pop();
    entry[entryName]=entryFile;
    htmlWebpackPlugins.push(new HtmlWebpackPlugin({
        template:`./src/${entryName}/index.html`,
        filename:`${entryName}/index.html`,
        chunks:[entryName],
        inject:true,
        minify:{
            html5:true,
            collapseWhitespace:true,
            preserveLineBreaks:false,
            minifyCSS:true,
            minifyJS:true,
            removeComments:false
        }
    }));
});

module.exports={
    entry,
    plugins: [

        ...htmlWebpackPlugins
    ]
}
```

23. 日志优化

- 日志太多太少都不美观
- 可以修改stats

预设 替代 描述 errors-only none 只在错误时输出 minimal none 发生错误和新的编译时输出 none false 没有输出 normal true 标准输出 verbose none 全部输出

22.1 friendly-errors-webpack-plugin

- friendly-errors-webpack-plugin (https://www.npmjs.com/package/friendly-errors-webpack-plugin)
- success 构建成功的日志提示
- warning 构建警告的日志提示
- error 构建报错的日志提示

```
cnpm i friendly-errors-webpack-plugin
```

```
+ stats:'verbose',
  plugins:[
+   new FriendlyErrorsWebpackPlugin()
  ]
```

> 编译完成后可以通过 `echo $?`获取错误码，0为成功，非0为失败

24. 日志输出

```
  "scripts": {
    "build": "webpack",
+    "build:stats":"webpack --json > stats.json",
    "dev": "webpack-dev-server --open"
  },
```

```
const webpack = require('webpack');
const config = require('./webpack.config.js');
webpack(config,(err,stats)=>{
  if(err){
    console.log(err);
  }
  if(stats.hasErrors()){
    return console.error(stats.toString("errors-only"));
  }
  console.log(stats);
});
```

25.费时分析

```
const SpeedMeasureWebpackPlugin = require('speed-measure-webpack-plugin');
const smw = new SpeedMeasureWebpackPlugin();
module.exports =smw.wrap({
});
```

26.webpack-bundle-analyzer

- 是一个webpack的插件，需要配合webpack和webpack-cli一起使用。这个插件的功能是生成代码分析报告，帮助提升代码质量和网站性能

```
cnpm i webpack-bundle-analyzer -D
```

```
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin
module.exports={
  plugins: [
    new BundleAnalyzerPlugin()

  ]
}
```

```
{
 "scripts": {
    "dev": "webpack --config webpack.dev.js --progress"
  }
}
```

webpack.config.js

```
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin
module.exports={
  plugins: [
    new BundleAnalyzerPlugin({
      analyzerMode: 'disabled',
      generateStatsFile: true,
    }),
  ]
}
```

```
{
 "scripts": {
    "generateAnalyzFile": "webpack --profile --json > stats.json",
    "analyz": "webpack-bundle-analyzer --port 8888 ./dist/stats.json"
  }
}
```

```
npm run generateAnalyzFile
npm run analyz
```

27.polyfill

27.1 babel-polyfill

- babel-polyfill是React官方推荐，缺点是体积大
- babel-polyfill用正确的姿势安装之后，引用方式有三种：
- 1. require("babel-polyfill");
- 1. import "babel-polyfill";
- 1. module.exports = { entry: ["babel-polyfill", "./app/js"] };

27.2 polyfill-service

- 自动化的 JavaScript Polyfill 服务
- Polyfill.io 通过分析请求头信息中的 UserAgent 实现自动加载浏览器所需的 polyfills
- polyfill-service (https://polyfill.io/v3/)
- polyfill-io (https://c7sky.com/polyfill-io.html)

28. libraryTarget 和 library

- outputlibrarytarget (https://webpack.js.org/configuration/output/#outputlibrarytarget)
- 当用 Webpack 去构建一个可以被其他模块导入使用的库时需要用到它们
- output.library 配置导出库的名称
- output.libraryExport 配置要导出的模块中哪些子模块需要被导出。 它只有在 output.libraryTarget 被设置成 commonjs 或者 commonjs2 时使用才有意义
- output.libraryTarget 配置以何种方式导出库,是字符串的枚举类型，支持以下配置

libraryTarget 使用者的引入方式 使用者提供给被使用者的模块的方式 var 只能以script标签的形式引入我们的库 只能以全局变量的形式提供这些被依赖的模块 commonjs 只能按照commonjs的规范引入我们的库 被依赖模块需要按照commonjs规范引入 amd 只能按amd规范引入 被依赖的模块需要按照amd规范引入 umd 可以用script、commonjs、amd引入 按对应的方式引入

28.1 var (默认)

编写的库将通过 var被赋值给通过 library指定名称的变量。

28.1.1 index.js

```
module.exports =  {
    add(a,b) {
        return a+b;
    }
}
```

28.1.2 bundle.js

```
var calculator=(function (modules) {}({}))
```

29.1.3 index.html

```
    let ret = calculator.add(1,2);
    console.log(ret);
```

28.2 commonjs

编写的库将通过 CommonJS 规范导出。

28.2.1 导出方式

```
exports["calculator"] = (function (modules) {}({}))
```

28.2.2 使用方式

```
require('npm-name')['calculator'].add(1,2);
```

> npm-name是指模块发布到 Npm 代码仓库时的名称

28.3 commonjs2

编写的库将通过 CommonJS 规范导出。

28.3.1 导出方式

```
module.exports = (function (modules) {}({}))
```

28.3.2 使用方式

```
require('npm-name').add();
```

> 在 output.libraryTarget 为 commonjs2 时，配置 output.library 将没有意义。

28.4 this

编写的库将通过 this 被赋值给通过 library 指定的名称，输出和使用的代码如下：

28.4.1 导出方式

```
this["calculator"]= (function (modules) {}({})
```

28.4.2 使用方式

```
this.calculator.add();
```

28.5 window

编写的库将通过 window 被赋值给通过 library 指定的名称，即把库挂载到 window 上，输出和使用的代码如下：

28.5.1 导出方式

```
window["calculator"]= (function (modules) {}({})
```

28.5.2 使用方式

```
window.calculator.add();
```

28.6 global

编写的库将通过 global 被赋值给通过 library 指定的名称，即把库挂载到 global 上，输出和使用的代码如下：

28.6.1 导出方式

```
global["calculator"]= (function (modules) {}({})
```

28.6.2 使用方式

```
global.calculator.add();
```

28.6 umd

```
(function webpackUniversalModuleDefinition(root, factory) {
  if(typeof exports === 'object' && typeof module === 'object')
    module.exports = factory();
  else if(typeof define === 'function' && define.amd)
    define([], factory);
  else if(typeof exports === 'object')
    exports['MyLibrary'] = factory();
  else
    root['MyLibrary'] = factory();
})(typeof self !== 'undefined' ? self : this, function() {
  return _entry_return_;
});
```

29. 打包库和组件

- webpack还可以用来打包JS库
- 打包成压缩版和非压缩版
- 支持AMD/CJS/ESM方式导入

29.1 编写库文件

29.1.1 webpack.config.js

```
const path = require('path');
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const TerserPlugin = require('terser-webpack-plugin');
module.exports = {
    mode:'none',
    entry:{
        'zhufengmath':'./src/index.js',
        'zhufengmath.min':'./src/index.js'
    },
    optimization:{
        minimize:true,
        minimizer:[

            new TerserPlugin({
                include:/\.min\.js/
            })
        ]
    },
    output:{
        filename:'[name].js',
        library:'zhufengmath',
        libraryExport:'default',
        libraryTarget:'umd'
    }
};
```

29.1.2 package.json

```
 "scripts": {
+    "build": "webpack",
```

29.1.3 index.js

```
if(process.env.NODE_ENV == 'production'){
    module.exports = require('./dist/zhufengmath.min.js');
}else{
    module.exports = require('./dist/zhufengmath.js');
}
```

29.1.4 src\index.js

```
export function add(a,b){
  return a+b;
}
export function minus(a,b){
  return a-b;
}
export function multiply(a,b){
  return a*b;
}
export function divide(a,b){
  return a/b;
}
export default {
  add,minus,multiply,divide
}
```

29.2 git规范和changelog

29.2.1 良好的git commit好处

- 可以加快code review 的流程
- 可以根据git commit 的元数据生成changelog
- 可以让其它开发者知道修改的原因

29.2.2 良好的commit

- commitizen (https://www.npmjs.com/package/commitizen)
- validate-commit-msg (https://www.npmjs.com/package/validate-commit-msg)
- conventional-changelog-cli (https://www.npmjs.com/package/conventional-changelog-cli)
- commit_message_change_log (http://www.ruanyifeng.com/blog/2016/01/commit_message_change_log.html)
- 统一团队的git commit 标准
- 可以使用angular的git commit日志作为基本规范
    - 提交的类型限制为 feat、fix、docs、style、refactor、perf、test、chore、revert等
    - 提交信息分为两部分，标题(首字母不大写，末尾不要加标点)、主体内容(描述修改内容)
- 日志提交友好的类型选择提示 使用commitize工具
- 不符合要求格式的日志拒绝提交 的保障机制
    - 需要使用validate-commit-msg工具
- 统一changelog文档信息生成
    - 使用conventional-changelog-cli工具

```
cnpm i commitizen  validate-commit-msg conventional-changelog-cli -S
commitizen init cz-conventional-changelog --save --save-exact
git cz
```

29.2.3 提交的格式

```
():
```

- 代表某次提交的类型，比如是修复bug还是增加feature
- 表示作用域，比如一个页面或一个组件
- 主题 ，概述本次提交的内容
- 详细的影响内容
- 修复的bug和issue链接

类型 含义 feat 新增feature fix 修复bug docs 仅仅修改了文档，比如README、CHANGELOG、CONTRIBUTE等 style 仅仅修改了空格、格式缩进、偏好等信息，不改变代码逻辑 refactor 代码重构，没有新增功能或修复bug perf 优化相关，提升了性能和体验 test 测试用例，包括单元测试和集成测试 chore 改变构建流程，或者添加了依赖库和工具 revert 回滚到上一个版本 ci CI 配置，脚本文件等更新

29.2.4 precommit钩子

```
cnpm i husky validate-commit-msg conventional-changelog-cli --save-dev
```

```
"scripts": {
    "commitmsg": "validate-commit-msg",
    "changelog": "conventional-changelog -p angular -i CHANGELOG.md -s -r 0"
}
```

29.2.4.1 conventional-changelog-cli

- `conventional-changelog-cli` 默认推荐的 commit 标准是来自angular项目

```
$ conventional-changelog -p angular -i CHANGELOG.md -s
```

- 参数 `-i CHANGELOG.md`表示从 `CHANGELOG.md` 读取 `changelog`
- 参数 -s 表示读写 `changelog` 为同一文件
- 这条命令产生的 changelog 是基于上次 tag 版本之后的变更（Feature、Fix、Breaking Changes等等）所产生的

如果你想生成之前所有 commit 信息产生的 changelog 则需要使用这条命令

```
conventional-changelog -p angular -i CHANGELOG.md -s -r 0
```

- 其中 -r 表示生成 changelog 所需要使用的 release 版本数量，默认为1，全部则是0。

29.3 semver版本规范

- 可以避免循环依赖并减少依赖冲突
- 开源项目的版本号通常由三位组件，比如 `x.y.z`

29.3.1 版本号说明

- 主版本号 `x`： 重大升级，做了不兼容的API修改
- 次版本号 `y`： 做了向下的兼容的功能新增
- 修订号 `z`: 做了向下兼容和问题修复

- 版本是严格递增的，比如1.1.1 -> 1.1.2 -> 1.2.1

29.3.2 先行版本

- 在发布重要版本的时候，可以先发布alpha,rc等先行版本
- 格式是在修订版本后面加上一个连接号(-),再加上一连串以(.)分割的标识符，标识符可以由英文、数字和连接号([0-9A-Za-z-])组成
    - alpha: 是内部测试版，一般不向外发表，会有比较多的Bug,只给测试人员用
    - bate: 也是测试版，这个阶段版本会增加新的功能，在Alpha之后推出
    - rc(Release Candidate) 候选版本，这个阶段不会再加入新的功能，主要是用于解决错误
- 15.0.0 -> 16.0.0-alpha.0 -> 16.0.0-alpha.1-> 16.0.0-bate.0 -> 16.0.0-bate.1-> 16.0.0-rc.1-> 16.0.0-rc.2

29.3.3 升级版本

```
npm version
npm version patch 升级补丁版本号
npm version minor 升级小版本号
npm version major 升级大版本号
```

29.3.4 发布

```
npm adduser
npm login
npm publish
```

29.4 使用

UMD=ES Module + CJS + AMD CDN

```
console.log(window.zhufengmath);
```

30. px 自动转成rem

- 使用px2rem-loader
- 页面渲染时计算根元素的 font-size值
- lib-flexible (https://github.com/amfe/lib-flexible)

30.1 安装

```
cnpm i px2rem-loader lib-flexible -D
```

30.2 index.html

index.html

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>主页title>
    <script>
      let docEle = document.documentElement;
      function setRemUnit () {

        docEle.style.fontSize = docEle.clientWidth / 10 + 'px';
      }
      setRemUnit();
      window.addEventListener('resize', setRemUnit);
    script>
head>
<body>
    <div id="root">div>
body>
```

30.3 reset.css

```
*{
    padding: 0;
    margin: 0;
}
#root{
    width:375px;
    height:375px;
    border:1px solid red;
    box-sizing: border-box;
}
```

30.4 webpack.config.js

```
{
        test:/\.css$/,//如果要require或import的文件是css的文件的话
        //从右向左处理CSS文件,loader是一个函数
        use:[{
                loader:MiniCssExtractPlugin.loader,
                options:{
                     publicPath: (resourcePath, context) => {
                        return '/';
                    }
                    //publicPath: '/'
                }
        },{
                    loader:'css-loader',
                    options:{
                        //Enables/Disables or setups number of loaders applied before CSS loader.

                        importLoaders:0
                    }
                },{
                    loader:'postcss-loader',
                    options:{
                        plugins:[
                            require('autoprefixer')
                        ]
                    }
                },{
+                    loader:'px2rem-loader',
+                    options:{
+                        remUnit:75,
+                        remPrecesion:8
+                    }
+                }]
+            },
```

31. 内联资源

- 可以在页面框架加载时进行初始化
- 可以上报打点数据
- CSS的内联可以避免页面闪动
- 可以减少HTTP网络请求的数量

31.1 webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin  = require("html-webpack-plugin");
module.exports = {
    entry: './src/index.js',
    output: {
        filename: '[name].bundle.js',
        path: path.resolve(__dirname, 'dist')
    },
    plugins:[
        new HtmlWebpackPlugin({
            title: "实现 Html、JavaScript、CSS 内联",
            template:"./src/index.html",
        }),
    ]
};
```

31.2 安装raw-loader

```
cnpm install raw-loader --save-dev
```

31.3 内联

```

${require("raw-loader!./inline.html").default}

<script>
${require("raw-loader!./inline.js").default}
script>

<style>
${require("!!raw-loader!./inline.css").default}
style>
```

32.参考

32.1 参考文档

- webpack-start (https://gitee.com/zhufengpeixun/webpack-start/commits/master)
- resolve (https://webpack.docschina.org/configuration/resolve/)

32.2 常用loader列表

- webpack 可以使用 loader 来预处理文件。这允许你打包除 JavaScript 之外的任何静态资源。你可以使用 Node.js 来很简单地编写自己的 loader。
- awesome-loaders (https://github.com/webpack-contrib/awesome-webpack#loaders)

32.3 文件

- raw-loader 加载文件原始内容（utf-8）
- val-loader 将代码作为模块执行，并将 exports 转为 JS 代码
- url-loader 像 file loader 一样工作，但如果文件小于限制，可以返回 data URL
- file-loader 将文件发送到输出文件夹，并返回（相对）URL

32.4 JSON

- json-loader 加载 JSON 文件（默认包含）
- json5-loader 加载和转译 JSON 5 文件
- cson-loader 加载和转译 CSON 文件

32.4 转换编译(Transpiling)

- script-loader 在全局上下文中执行一次 JavaScript 文件（如在 script 标签），不需要解析
- babel-loader 加载 ES2015+ 代码，然后使用 Babel 转译为 ES5
- buble-loader 使用 Bublé 加载 ES2015+ 代码，并且将代码转译为 ES5
- traceur-loader 加载 ES2015+ 代码，然后使用 Traceur 转译为 ES5
- ts-loader 或 awesome-typescript-loader 像 JavaScript 一样加载 TypeScript 2.0+
- coffee-loader 像 JavaScript 一样加载 CoffeeScript

32.5 模板(Templating) #

- html-loader 导出 HTML 为字符串，需要引用静态资源
- pug-loader 加载 Pug 模板并返回一个函数
- jade-loader 加载 Jade 模板并返回一个函数
- markdown-loader 将 Markdown 转译为 HTML
- react-markdown-loader 使用 markdown-parse parser(解析器) 将 Markdown 编译为 React 组件
- posthtml-loader 使用 PostHTML 加载并转换 HTML 文件
- handlebars-loader 将 Handlebars 转移为 HTML
- markup-inline-loader 将内联的 SVG/MathML 文件转换为 HTML。在应用于图标字体，或将 CSS 动画应用于 SVG 时非常有用

32.6 样式 #

- style-loader 将模块的导出作为样式添加到 DOM 中
- css-loader 解析 CSS 文件后，使用 import 加载，并且返回 CSS 代码
- less-loader 加载和转译 LESS 文件
- sass-loader 加载和转译 SASS/SCSS 文件
- postcss-loader 使用 PostCSS 加载和转译 CSS/SSS 文件
- stylus-loader 加载和转译 Stylus 文件

32.7 清理和测试(Linting && Testing) #

- mocha-loader 使用 mocha 测试（浏览器/NodeJS）
- eslint-loader PreLoader，使用 ESLint 清理代码
- jshint-loader PreLoader，使用 JSHint 清理代码
- jscs-loader PreLoader，使用 JSCS 检查代码样式
- coverjs-loader PreLoader，使用 CoverJS 确定测试覆盖率

32.8 框架(Frameworks) #

- vue-loader 加载和转译 Vue 组件
- polymer-loader 使用选择预处理器(preprocessor)处理，并且 require() 类似一等模块(first-class)的 Web 组件
- angular2-template-loader 加载和转译 Angular 组件