webpack.config.js 的配置项

mode

- production
- development
- none：保留原始打包结果

context

表示入口 entry 是以哪个目录为起点的

entry

`entry`有 5 种形式

- 字符串
- 数组，相当于是最后面的是入口文件，其余的在入口文件中用`import`引入
- 对象，每一组键值对都是一个入口
- 函数，返回以上任一即可
- 描述符形式（暂未研究）

output

`output`是一个对象，有 4 个属性

path

资源输出目录，如果不设置，则默认为`dist`文件夹

filename 的占位符

- `[chunkhash]`
- `[contenthash]`
- `[hash]`：在`webpack5`中被废弃，但仍然可以使用
- `[fullhash]`
- `[name]`：`chunk`的名字，也即如果`entry`是一个对象，则是这个对象的键名；如果`entry`是一个字符串，则是`main`
- `[id]`：`webpack`打包过程中为每个`chunk`生成的唯一序号

publicPath

资源访问目录，默认值为`auto`（和`output.path`位置一样）

假设当前页面为`http://www.example.org/w3c/index.html`，那么

`./`开头的相对地址

```
//设置资源访问目录为http://www.example.org/w3c/
output.publicPath:""

//设置资源访问目录为http://www.example.org/diist/
output.publicPath:"../dist/"
```

若以/开头表示以服务器地址根目录作为基础路径

```
//设置资源访问目录为http://www.example.org/
output.publicPath:"/"

//设置资源访问目录为http://www.example.org/dist/
output.publicPath:"/dist/"
```

绝对http协议地址

```
//设置资源访问目录为http://www.example.org/
output.publicPath:"http://www.example.org/"
```

相对http协议地址

```
//设置资源访问目录为http://www.example.org/
output.publicPath:"//www.example.org/"
```

chunkFilename

非入口文件的chunk名称，如import()动态导入的文件，支持占位符

hash、fullhash、chunkhash 和 contenthash 的区别

主要与浏览器缓存行为有关

fullhash和hash一样，是webpack通过**所有文件**内容计算来的一个hash值，[hash:8]表示取hash值的前8位

chunkhash是根据**每个 chunk**计算出的hash值

contenthash是根据**文件内容**计算出的hash值，像使用提取css文件的loader的时候一般用这个

预处理器

这个配置项为什么叫module？因为在webpack中每个文件都是一个模块，可以用其他loader加载不同类型的文件（模块），webpack只认识js和json，其他文件需要借助loader

module的写法

```
//...
module: {
  rules: [
    {
      test: /\.js$/,

      // use: ["style-loader", "css-loader"],

      //如果是一个loader可以写成
      //use:"css-loader"

      //如果预处理器可以配置额外参数可以写成
      // use:[{
      //   loader:"babel-loader",
      //   option:{/*...*/}
      // }]

      //也可写成
      use:{
        loader:"babel-loader",
        option:{/*...*/}
      }

      //exclude值可以是字符串或者正则
      //表示会处理除了node_modules以外的文件夹
      exclude:/node_modules/,

      //表示只对src文件夹进行处理
      //如果exclude和include同时存在，webpack会优先使用exclude配置
      include:/src/

    },
  ];
}
//...
```

> use中的loader处理流程是从后往前处理的

es6 转 es5 babel-loader

```
npm install -D @babel/preset-env@7.13.10

npm install -D @babel/core@7.13.10 babel-loader@8.2.2
```

```
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
```

```
            loader: "babel-loader",
            options: {
              presets: ["@babel/preset-env"],
              cacheDirectory:true//开启缓存，有助于提高编译速度
            },
          },
        },
      ],
    },
```

图片处理 url-loader

`npm install -D url-loader file-loader`

如果出来的图片路径不对，出现`[object module]`，试试加上`esModule: false`

可以使用的占位符 `[hash]`、 `[name]`、 `[contenthash]`

```
module: {
    rules: [
      {
        test: /\.(png|jpe?g|gif)$/i,
        use: {
          loader: "url-loader",
          options: {
            limit: 1024 * 8,
            esModule: false,
            //name:"[name]-[contenthash:8].[ext]"
            //publicPath
          },
        },
      },
    ],
    type: "javascript/auto"
  },
```

读取文本文件为字符串 raw-loader

插件

插件是在`webpack`编译阶段，通过调用`webpack`对外暴露的`api`来扩展`webpack`的功能

清除目录

`npm install clean-webpack-plugin@3.0.0 -D`

```
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
module.exports = {
```

```
  //...
  plugins: [new CleanWebpackPlugin()],
};
```

复制文件夹的文件到指定目录

```
npm install copy-webpack-plugin -D
```

好像只起到复制作用，但是手动也可以?

```
const CopyPlugin = require("copy-webpack-plugin");
module.exports = {
  //...
  plugins: [
    new CopyPlugin({
      patterns: [
        {
          from: path.resolve(__dirname, "./img"),
          to: path.resolve(__dirname, "./dist/image"),
        },
      ],
    }),
  ],
};
```

HTML 模板插件

```
npm install html-webpack-plugin -D
```

可以使用html模板或者自动生成html文件，可以自动将生成的css或js文件插入到html文件中

src/index.ejs

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title><%= htmlWebpackPlugin.options.title%></title>
  </head>
  <body>
    <h2>使用模板</h2>
  </body>
</html>
```

webpack.config.js

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
```

```
module.exports = {
  plugins: [
    new HtmlWebpackPlugin({
      title: "webpack-learn",
      //不配置模板的话会默认生成一个html文件
      template: "src/index.ejs",
      filename: "home.html",
    }),
  ],
};
```

webpack 开发环境配置

文件监听模式

`npx webpack --watch`

webpack-dev-server

安装：`npm i webpack-dev-server -D` 启动：`npx webpack server`

```
module.exports = {
  devServer: {
    open: true,
    hot: true,
    //开启html5 history模式，这会让任何请求都会返回index.html
    historyApiFallback: true,
    port: 8089,
    //是否为静态资源开启Gzip压缩
    compress: true,
    //web服务请求资源的路径
    publicPath: "/", //默认取值
  },
};
```

HMR

`webpack5`中将`devServer.hot`设置为`true`会自动添加该插件

```
devServer: {
  //...
    hot: true,
  },
```

在 `js` 文件尾部中添加以下代码使文件支持热更新

```
if (module.hot) {
  module.hot.accept();
}
```

添加 devtool 以调试编译后的代码

devtool 取值

```
module.exports = {
  devtool: "eval-cheap-module-source-map",
};
```

开发环境推荐用`eval-cheap-module-source-map`

生产环境下，推荐使用`hidden-source-map`或白名单策略

webpack5 内置的文件资源处理模块

Asset Modules 是用来替换`url-loader`、`file-loader`、`raw-loader`的

```
module.exports = {
  //...

  module: {
    rules: [
      {
        test: /\.(jpg|png)$/,

        //type = "asset/resource" 相当于file-loader的功能
        // type: "asset/resource",
        // generator: {
        //   //也可以在output.assetModuleFilename配置输出文件名
        //   filename: "static/[hash:8][ext][query]",
        // },

        //type = "asset/inline" 强制将图片转为base64
        // type: "asset/inline",

        //type = "asset" 看情况讨论
        //默认大于8kb以asset/resource处理，小于等于8kb以asset/inline处理
        //可配置parser.dataUrlCondition.maxSize来自定义阈值
        type: "asset",
        parser: {
          dataUrlCondition: {
            maxSize: 6 * 1024, //6kb
          },
        },
        generator: {
```

```
            //也可以在output.assetModuleFilename配置输出文件名
            filename: "static/[hash:8][ext][query]",
          },
        },
      ],
    },
  };
```

webpack 生产环境配置

用 cross-env 包来设置 node 环境变量

因为在windows（用set）和mac（用export）设置环境变量的方式不同，可以使用cross-env包来兼容设置环境变量

npm install cross-env -D

```
// package.json
"scripts": {
  //会先设置环境变量，然后执行webpack编译
    "build": "cross-env NODE_ENV=_development webpack"
  },
```

配置 webpack 环境变量

webpack 和 node 的环境变量的区别

webpack环境变量是可以在打包的文件运行在浏览器中可以获得的环境变量

而node环境变量是指在运行在node环境中可以获得的环境变量

两者不是同一个东西，配置了其中一个不会影响另一个

使用 webpack.DefinePlugin 配置 webpack 环境变量

注意当设置环境变量为一个字符串时，要用JSON.stringify()包裹起来 或 是像"'jack'"这样使用，否则在文件中使用后编译后会成为一个全局变量，一个例子：

webpack.config.js

```
const webpack = require("webpack");
module.exports = {
  //...
  entry: "./src/index.js",
  plugins: [
    new webpack.DefinePlugin({
      IS_OLD: true,
      MY_ENV: JSON.stringify("dev"),
```

```
      NAME: "jack", //正确写法："'jack'"或JSON.stringify("jack")  JSON.stringify会
生成："\"jack\""
      NODE_ENV: JSON.stringify("production"),
    }),
  ],
};
```

src/index

```
console.log(IS_OLD);
console.log(MY_ENV);
console.log(NAME);
console.log(NODE_ENV);
```

产出的main.js

```
console.log(!0),
  console.log("dev"),
  console.log(jack), //这里使用了jack全局变量
  console.log("production");
```

提取和压缩 CSS 文件

npm install -D mini-css-extract-plugin css-loader

```
const MiniCssPlugin = require("mini-css-extract-plugin");
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.css$/i,
        use: [MiniCssPlugin.loader, "css-loader"],
      },
    ],
  },
  plugins: [
    new MiniCssPlugin({
      filename: "[name]-[contenthash:8].css",
      chunkFilename: "[id].css", //异步代码提取的css文件名
    }),
  ],
};
```

sass 处理

```
npm install -D sass sass-loader
```

```
const MiniCssPlugin = require("mini-css-extract-plugin");
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.(css|scss)$/i,
        use: [MiniCssPlugin.loader, "css-loader", "sass-loader"], //注意顺序
      },
    ],
  },
  plugins: [
    new MiniCssPlugin({
      filename: "[name]-[contenthash:8].css",
      chunkFilename: "[id].css", //异步代码提取的css文件名
    }),
  ],
};
```

使用 postcss-loader 添加厂商前缀

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

```
npm i postcss-preset-env -D
```

postcss-loader通过postcss-preset-env来实现对厂商添加前缀，需要配置package.json文件，并添加postcss.config.js配置文件

> @babel/preset-env也会解析在package.json中添加的browserslist

```
//package.json
{
  "browserslist": [
    "defaults",
    "not ie < 11",
    "last 3 versions",
    "> 0.2%",
    "iOS 7",
    "last 3 iOS versions"
  ]
}

//postcss.config.js
module.exports = {
  plugins: [
    //自动添加前缀
    require("postcss-preset-env"),
  ],
```

```
};

//webpack.config.js
const path = require("path");
const MiniCssPlugin = require("mini-css-extract-plugin");
module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "main.js",
  },
  module: {
    rules: [
      {
        test: /\.css$/i,
        use: [MiniCssPlugin.loader, "css-loader", "postcss-loader", "scss-
loader"], //要保证postcss-loader在css-loader之前运行就可以
      },
    ],
  },
  plugins: [
    new MiniCssPlugin({
      filename: "[name]-[contenthash:8].css",
      chunkFilename: "[id].css",
    }),
  ],
};
```

使用 webpack-merge 合并 webpack 配置

```
npm install -D webpack-merge
```

```
//webpack.common.js
const path = require("path")
const HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "./dist/"),
    filename:"main.js"
  },
  module: {

  },
  plugins: [new HtmlWebpackPlugin()],
  mode:"none"
}

//webpack.development.js
const { merge } = require("webpack-merge");
const common = require("./webpack.common");
```

```
module.exports = merge(common, {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ["style-loader", "css-loader"],
      },
    ],
  },
});

//webpack.production.js
const { merge } = require("webpack-merge");
const common = require("./webpack.common");
const MiniCssPlugin = require("mini-css-extract-plugin");
module.exports = merge(common, {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [MiniCssPlugin.loader, "css-loader"],
      },
    ],
  },
  plugins: [new MiniCssPlugin()],
});

//package.json
{
//...
  "scripts": {
    //...
    "build": "cross-env NODE_ENV=production webpack --
config=webpack.production.js",
    "start": "cross-env NODE_ENV=development webpack server --
config=webpack.development.js"
  },
}
```

监控打包的文件大小

```
module.exports = {
  performance: {
    maxEntrypointSize: 512000, //512k,默认是250 000  即250k
    maxAssetSize: 512000, //512k,默认是250 000  即250k
    // hints:"error",//提示类型 error 或 warning默认值
    // assetFilter:(assetFilename)=>!/\.map$/.test(assetFilename)//配置对哪些文件进
行监控
```

```
  },
};
```

webpack 性能优化

可视化打包体积分析工具 webpack-bundle-analyzer

```
npm i webpack-bundle-analyzer -D
```

```
//webpack.production.js
const { merge } = require("webpack-merge");
const common = require("./webpack.common");
const BundleAnalyzerPlugin =
  require("webpack-bundle-analyzer").BundleAnalyzerPlugin;
module.exports = merge(common, {
  //...
  plugins: [new BundleAnalyzerPlugin()],
});
```

打包速度分析工具 speed-measure-webpack-plugin

```
npm i speed-measure-webpack-plugin -D
```

这个工具可以看到每个loader或者plugin用到的时间

```
//webpack.common.js
const SpeedMeasurePlugin = require("speed-measure-webpack-plugin");

module.exports = new SpeedMeasurePlugin().wrap({
  //...
  mode: "none",
});
```

压缩 js 和 css 文件

使用到terser-webpack-plugin、css-minimizer-webpack-plugin两个插件，前一个已经在webpack5中内置了

```
const TerserWebpackPlugin = require("terser-webpack-plugin");
const CssMinimizerWebpackPlugin = require("css-minimizer-webpack-plugin");
module.exports = {
  //...
  optimization: {
    minimize: true,
    minimizer: [new TerserWebpackPlugin(), new CssMinimizerWebpackPlugin()],
```

```
  },
};
```

缩小查找范围

- 配置预处理器的exclude和include

```
rules: [
  {
    test: /\.js$/,
    exclude: /node_modules/,
    use: {
      loader: "babel-loader",
    },
  },
];
```

- 通过module.noParse设置不需要被loader解析的模块

```
module: {
  noParse:/jquery|lodash/;
}
```

- 通过resolve.modules设置第三方模块位置，这样就不需要沿着目录一直向上级查找

```
module.exports = {
  //...
  resolve: {
    modules: [path.resolve(__dirname, "./node_modules")],
  },
};
```

- 通过resolve.extensions设置匹配文件后缀规则，这样webpack在匹配不到文件的时候不会尝试去加后缀查找了

```
module.exports = {
  //...
  resolve: {
    extensions: [".js", ".json", ".wasm"], //webpack5默认配置
  },
};
```

代码分割 optimization.splitChunks

摇树优化 tree shaking

使用mode = production默认开启tree shaking

使用terser-plugin会自动进行tree shaking

可以在package.json使用sideEffects来告诉tree shaking不能删除某些文件

> webpack4中不能优化那些导出的未使用到的嵌套的代码，比如export default {name,age}，另一个文件import * as person from "./person.js";console.log(person.name)，这样age没有被使用，而webpack5会删掉age进行优化

```
{
  //...
  "sideEffects": ["./polyfill.js"]
}
```

使用 webpack5 新增的文件缓存

```
module.exports = {
  //开发模式下，默认cache=true，这与type=memory一样的效果
  //生产模式下，默认cache=false，会禁用缓存
  cache: {
    type: "filesystem", //或memory
  },
};
```

webpack 原理与扩展

生成的文件分析

观察打包后生成的文件，整体是一个立即执行函数，大体的流程如下

1. 定义了一个modules数组，每一项是用函数形式将 js 文件包裹起来的函数，当然webpack已经对js文件进行了更改，函数接收module,exports,require三个参数
2. 定义一个缓存对象（key是moduleId，value是一个含有exports属性的module）和一个require函数，接收参数moduleId（其实就是modules数组的索引），如果缓存中有这个模块会直接返回这个模块的exports属性，否则函数体会定义一个有exports属性的对象module，将其缓存起来，然后会拿到modules[moduleId]函数，传入module,module.exports,require执行，这个函数可能会改变module.exports，也就是做的导出操作，执行完后最后返回module.exports
3. 执行require(0)

```
/******/ (() => {
  // webpackBootstrap
  /******/ "use strict";
```

```
//__webpack_modules__数组有两个匿名函数
/******/ var __webpack_modules__ = [
  /* 0 */
  /***/ (
    __unused_webpack_module,
    __webpack_exports__,
    __webpack_require__
  ) => {
    debugger;
    __webpack_require__.r(__webpack_exports__);
    /* harmony import */ var _b_js__WEBPACK_IMPORTED_MODULE_0__ =
      __webpack_require__(1);
    debugger;
    console.log(_b_js__WEBPACK_IMPORTED_MODULE_0__.year);
    /***/
  },
  /* 1 */
  /***/ (
    __unused_webpack_module,
    __webpack_exports__,
    __webpack_require__
  ) => {
    debugger;
    __webpack_require__.r(__webpack_exports__);
    /* harmony export */ __webpack_require__.d(__webpack_exports__, {
      /* harmony export */ year: () => /* binding */ year,
      /* harmony export */
    });
    var year = 2022;
    /***/
  },
  /******/
];

//缓存、require函数
/************************************************************************/
/******/ // The module cache
/******/ var __webpack_module_cache__ = {};
/******/
/******/ // The require function
/******/ function __webpack_require__(moduleId) {
  /******/ // Check if module is in cache
  /******/ if (__webpack_module_cache__[moduleId]) {
    /******/ return __webpack_module_cache__[moduleId].exports;
    /******/
  }
  /******/ // Create a new module (and put it into the cache)
  /******/ var module = (__webpack_module_cache__[moduleId] = {
    /******/ // no module.id needed
    /******/ // no module.loaded needed
    /******/ exports: {},
    /******/
  });
```

```
    /******/
    /******/ // Execute the module function
    /******/ __webpack_modules__[moduleId](
      module,
      module.exports,
      __webpack_require__
    );
    /******/
    /******/ // Return the exports of the module
    /******/ return module.exports;
    /******/
  }

  /******/ //   3个立即执行函数
  /************************************************************************/
  /******/ /* 给require函数对象定义d方法，作用是如果definition中有一个exports中没有的
属性，就用getter附给exports*/
  /******/ (() => {
    /******/ // define getter functions for harmony exports
    /******/ __webpack_require__.d = (exports, definition) => {
      /******/ for (var key in definition) {
        /******/ if (
          __webpack_require__.o(definition, key) &&
          !__webpack_require__.o(exports, key)
        ) {
          /******/ Object.defineProperty(exports, key, {
            enumerable: true,
            get: definition[key],
          });
          /******/
        }
        /******/
      }
      /******/
    };
    /******/
  })();
  /******/

  /******/ /* 给require函数对象定义o方法，作用是判断对象是否拥有指定属性名*/
  /******/ (() => {
    /******/ __webpack_require__.o = (obj, prop) =>
      Object.prototype.hasOwnProperty.call(obj, prop);
    /******/
  })();
  /******/

  /******/ /* 给require函数对象定义r方法，作用是将exports的Symbol.toStringTag属性赋值
为‘Module’；将exports的__esModule属性赋值为true*/
  /******/ (() => {
    /******/ // define __esModule on exports
    /******/ __webpack_require__.r = (exports) => {
      /******/ if (typeof Symbol !== "undefined" && Symbol.toStringTag) {
        /******/ Object.defineProperty(exports, Symbol.toStringTag, {
```

```
            value: "Module",
          });
          /******/
        }
        /******/ Object.defineProperty(exports, "__esModule", { value: true });
        /******/
      };
      /******/
    })();
    /******/

    /************************************************************************/
    /******/ // startup
    /******/ // Load entry module
    /******/ __webpack_require__(0);
    /******/ // This entry module used 'exports' so it can't be inlined
    /******/
})();
```

自定义 loader

loader API

txt-loader

```
//src是上一个loader处理后的结果，如果这个loader是第一个处理该文件的loader，那么src就是文件内容
//如果这个loader是处理一个文件中的最后一个loader，那么返回的内容必须是js能够执行的字符串（相当于把源文件替换为该字符串然后webpack再进行打包）
//this.query是传入的参数
//同步模式下使用this.callback(null|error,result:String|Buffer,source map,自定义参数value:any)
//this.async()返回this.callback  告诉loader-runner 这个loader将会异步回调
module.exports = function (src, ...args) {
  console.log("----------------------");
  console.log("this.query", this.query); //{key:"value"}
  console.log("args", args); //[]
  console.log("----------------------");
  if (this.cacheable) this.cacheable(); //启用缓存
  return `module.exports = "${src}"`;
};

//webpack.config.js
rules: [
  {
    test: /\.txt$/,
    use: {
      loader: "./txt-loader.js",
      options: {
        key: "value",
      },
```

```
    },
  },
];
```

自定义插件

plugin API

```
module.exports = class HelloPlugin {
  constructor(options) {
    this.options = options;
  }

  apply(compiler) {
    compiler.hooks.done.tap("HelloPlugin", (compilation) => {
      console.log("Hello Plugin");
    });
  }
};
```