



珠峰前端架构直播课(每周)

- 晋级高级前端-对标阿里P6+
- 前端技术专家联合研发

JS高级 / VUE / REACT / NodeJS / 前端工程化 / 项目实战 / 测试 / 运维



长按识别二维码加微信
获取直播地址和历史精彩视频



1.项目初始化

1.1 安装

```
cnpm i webpack webpack-cli html-webpack-plugin webpack-dev-server cross-env -D
```

1.2 webpack.config.js

```
const path = require("path");
module.exports = {
  mode: "development",
  devtool: 'source-map',
  context: process.cwd(),
  entry: {
    main: "./src/index.js",
  },
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js"
  }
};
```

1.3 package.json

```
"scripts": {
  "build": "webpack",
  "start": "webpack serve"
},
```

2.数据分析

2.1 日志美化

- [friendly-errors-webpack-plugin](#)可以识别某些类别的webpack错误，并清理，聚合和优先级，以提供更好的开发人员体验

2.1.1 安装

```
cnpm i friendly-errors-webpack-plugin node-notifier -D
```

2.1.2 webpack.config.js

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
+const FriendlyErrorsWebpackPlugin = require('friendly-errors-webpack-plugin');
+const notifier = require('node-notifier');
+const ICON = path.join(__dirname, 'icon.jpg');
module.exports = {
  mode: "development",
  devtool: 'source-map',
  context: process.cwd(),
  entry: {
    main: "./src/index.js",
  },
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js"
  },
  plugins:[
    new HtmlWebpackPlugin(),
+   new FriendlyErrorsWebpackPlugin({
+     onErrors: (severity, errors) => {
+       const error = errors[0];
+       notifier.notify({
+         title: "webpack编译失败",
+         message: severity + ': ' + error.name,
+         subtitle: error.file || '',
+         icon: ICON
+       });
+     }
+   })
  ]
};
```

2.2 速度分析

- [speed-measure-webpack5-plugin](#)可以分析打包速度

2.2.1 安装

```
cnpm i speed-measure-webpack5-plugin -D
```

2.2.2 webpack.config.js

```
+const SpeedMeasureWebpackPlugin = require('speed-measure-webpack5-plugin');
+const smw = new SpeedMeasureWebpackPlugin();
+module.exports = smw.wrap({
  mode: "development",
  devtool: 'source-map',
  ...
+});
```

2.3 文件体积监控

- [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
- 它可以直观分析打包出的文件包含哪些，大小占比如何，模块包含关系，依赖项，文件是否重复，压缩后大小如何，针对这些，我们可以进行文件分割等操作。

2.3.1 安装

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

2.3.2 编译启动

2.3.2.1 webpack.config.js

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

2.3.2.2 package.json

```
"scripts": {
  "build": "webpack",
  "start": "webpack serve",
+  "dev": "webpack --progress"
},
```

2.3.3 单独启动

2.3.3.1 webpack.config.js

```
const {BundleAnalyzerPlugin} = require('webpack-bundle-analyzer')
module.exports={
  plugins: [
    new BundleAnalyzerPlugin({
+     analyzerMode: 'disabled', // 不启动展示打包报告的http服务器
+     generateStatsFile: true, // 是否生成stats.json文件
    }),
  ]
}
```

2.3.3.2 package.json

```
"scripts": {
  "build": "webpack",
  "start": "webpack serve",
  "dev": "webpack --progress",
+  "analyzer": "webpack-bundle-analyzer --port 8888 ./dist/stats.json"
}
```

3.编译时间优化

- 减少要处理的文件
- 缩小查找的范围

3.1 缩小查找范围

3.1.1 extensions

- 指定 `extensions` 之后可以不用在 `require` 或是 `import` 的时候加文件扩展名
- 查找的时候会依次尝试添加扩展名进行匹配

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

3.1.2 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

```
cnpm i bootstrap css-loader style-loader -S
```

```
+const bootstrap =
path.resolve(__dirname, 'node_modules/bootstrap/dist/css/bootstrap.css');
module.exports = smw.wrap({
  mode: "development",
  devtool: 'source-map',
  context: process.cwd(),
  entry: {
    main: "./src/index.js",
  },
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js"
  },
  resolve: {
    extensions: [".js",".jsx",".json"],
+   alias:{bootstrap}
  },
+  module:{
+    rules:[
+      {
+        test: /\.css$/,
+        use: ['style-loader', 'css-loader']
```

```
+         }
+     ]
+ }
+ });
```

3.1.3 modules

- 对于直接声明依赖名的模块webpack会使用类似 `Node.js` 一样进行路径搜索，搜索 `node_modules` 目录
- 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` 中的话可以直接指定
- 默认配置

```
resolve: {
  extensions: [".js", ".jsx", ".json"],
  alias: {bootstrap},
+ modules: ['node_modules']
},
```

- 直接指定

```
resolve: {
+ modules: ['C:/node_modules', 'node_modules'],
}
```

3.1.4 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: {
  // 配置 target === "web" 或者 target === "webworker" 时 mainFields 默认值是:
+ mainFields: ['browser', 'module', 'main'],
  // target 的值为其他时, mainFields 默认值为:
+ mainFields: ["module", "main"],
}
```

3.1.5 mainFiles

- 当目录下没有 `package.json` 文件时，我们会默认使用目录下的 `index.js` 这个文件

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

3.1.6 oneOf

- 每个文件对于rules中的所有规则都会遍历一遍，如果使用oneOf就可以解决这个问题，只要能匹配一个即可退出
- 在oneOf中不能两个配置处理同一种类型文件

webpack.config.js

```
rules: [
+ {
```

```

+   oneOf: [
     {
       test: /\.js$/,
       include: path.resolve(__dirname, "src"),
       exclude: /node_modules/,
       use: [
         {
           loader: 'thread-loader',
           options: {
             workers: 3
           }
         },
         {
           loader: 'babel-loader',
           options: {
             cachedDirectory: true
           }
         }
       ]
     },
     {
       test: /\.css$/,
       use: ['cache-loader', 'logger-loader', 'style-loader', 'css-loader']
     }
   ]
+   ]
   }
 ]

```

3.1.7 external

- 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`

3.1.7.1 安装

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

3.1.7.2 使用jquery

src/index.js

```

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

```

3.1.7.3 index.html

src/index.html

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

3.1.7.4 webpack.config.js

webpack.config.js

```
+ externals: {
+   jquery: 'jQuery',
+ },
  module: {
```

3.1.8 resolveLoader

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

3.1.8.1 logger-loader.js

loaders\logger-loader.js

```
function loader(source){
  console.log('logger-loader');
  return source;
}
module.exports = loader;
```

3.1.8.2 webpack.config.js

webpack.config.js

```
module.exports = {
  resolve: {
    extensions: ['.js', '.jsx', '.json'],
    alias: {bootstrap},
    modules: ['node_modules'],
  },
+  resolveLoader: {
+    modules: [path.resolve(__dirname, 'loaders'), 'node_modules'],
+  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
+          'logger-loader',
          'style-loader',
          'css-loader'
        ]
      }
    ]
  },
};
```

3.2 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
- 使用 `noParse` 进行忽略的模块文件中不能使用 `import`、`require` 等语法

3.2.1 src\index.js

src\index.js

```
let title = require('./title');
console.log(title);
```

3.2.2 src\title.js

src\title.js

```
let name = require('./name');
module.exports = `hello ${name}`;
```

3.2.3 src\name.js

src\name.js

```
module.exports = 'zhufeng';
```

3.2.4 webpack.config.js

```
module.exports = {
  module: {
    +   noParse: /title.js/, // 正则表达式
  }
}
```

3.3 IgnorePlugin

- [ignore-plugin](#)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
- requestRegExp 匹配(test)资源请求路径的正则表达式。
- contextRegExp (可选) 匹配(test)资源上下文（目录）的正则表达式。
- moment会将所有本地化内容和核心功能一起打包,你可使用 `IgnorePlugin` 在打包时忽略本地化内容

3.3.1 安装

```
cnpm i moment -S
```

3.3.2 src\index.js

src\index.js

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

3.3.3 webpack.config.js

webpack.config.js


```

plugins:[
+   new webpack.IgnorePlugin({
+     resourceRegExp: /^\.\/locale$/,
+     contextRegExp: /moment$/
+   })
]

```

3.4 thread-loader(多进程)

- 把[thread-loader](#)放置在其他 loader 之前，放置在这个 loader 之后的 loader 就会在一个单独的 worker 池(worker pool)中运行
- include 表示哪些目录中的 .js 文件需要进行 babel-loader
- exclude 表示哪些目录中的 .js 文件不要进行 babel-loader
- exclude 的优先级高于 include, 尽量避免 exclude, 更倾向于使用 include

3.4.1 安装

```

cnpm i thread-loader babel-loader @babel/core @babel/preset-env -D

```

3.4.2 webpack.config.js

webpack.config.js

```

rules: [
+   {
+     test: /\.js$/,
+     include: path.resolve(__dirname, "src"),
+     exclude: /node_modules/,
+     use: [
+       {
+         loader: 'thread-loader',
+         options: {workers: 3}
+       }, 'babel-loader']
+   },
+   {
+     test: /\.css$/,
+     use: ['logger-loader', 'style-loader', 'css-loader']
+   }
]

```

3.5 利用缓存

- 利用缓存可以提升重复构建的速度

3.5.1 babel-loader

- Babel在转义js文件过程中消耗性能较高，将[babel-loader](#)执行的结果缓存起来，当重新打包构建时会尝试读取缓存，从而提高打包构建速度、降低消耗
- 默认存放位置是 node_modules/.cache/babel-loader

```

{
  test: /\.js$/,
  include: path.resolve(__dirname, "src"),
  exclude: /node_modules/,

```

```

        use: [
          {
            loader: 'thread-loader',
            options: {
              workers: 3
            }
          },
+         {
+           loader: 'babel-loader',
+           options: {
+             cacheDirectory: true
+           }
+         }
        ],
      },
    ],
  },
},

```

3.5.2 cache-loader

- 在一些性能开销较大的[cache-loader](#)之前添加此 loader,可以以将结果缓存中磁盘中
- 默认保存在 `node_modules/.cache/cache-loader` 目录下

```
cnpm i cache-loader -D
```

```

{
  test: /\.css$/,
  use: [
+    'cache-loader',
    'logger-loader',
    'style-loader',
    'css-loader']
}

```

3.5.3 hard-source-webpack-plugin

- `HardSourceWebpackPlugin` 为 模块 提供了中间缓存,缓存默认的存放路径是 `node_modules/.cache/hard-source`
- 配置 `hard-source-webpack-plugin` 后, 首次构建时间并不会太大的变化,但是从第二次开始,构建时间大约可以减少 80% 左右
- [webpack5](#)中已经内置了模块缓存,不需要再使用此插件

3.5.3.1 安装

```
cnpm i hard-source-webpack-plugin -D
```

3.5.3.2 webpack.config.js

```

+let HardSourceWebpackPlugin = require('hard-source-webpack-plugin');

module.exports = {
  plugins: [
+    new HardSourceWebpackPlugin()
  ]
}

```

4.编译体积优化

4.1 压缩JS、CSS、HTML和图片

- [optimize-css-assets-webpack-plugin](#)是一个优化和压缩CSS资源的插件
- [terser-webpack-plugin](#)是一个优化和压缩JS资源的插件
- [image-webpack-loader](#)可以帮助我们图片进行压缩和优化

4.1.1 安装

```
cnpm i terser-webpack-plugin optimize-css-assets-webpack-plugin image-webpack-loader -D
```

4.1.2 webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
+const OptimizeCssAssetsWebpackPlugin = require('optimize-css-assets-webpack-plugin');
+const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  devtool: false,
  entry: './src/index.js',
+  optimization: {
+    minimize: true,
+    minimizer: [
+      new TerserPlugin(),
+    ],
+  },
  module: {
    rules: [
      {
        test: /\.?(png|svg|jpg|gif|jpeg|ico)$/,
        use: [
          'url-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
+       }
+     }
+   }
  ]
}
plugins: [
  new HtmlWebpackPlugin({
    template: './src/index.html',
+   minify: {
+     collapseWhitespace: true,
+     removeComments: true
+   }
  })
+ new OptimizeCssAssetsWebpackPlugin(),
],
};

```

4.2 清除无用的CSS

- [purgecss-webpack-plugin](#)单独提取CSS并清除用不到的CSS

4.2.1 安装

```
cnpm i purgecss-webpack-plugin mini-css-extract-plugin -D
```

4.2.3 src\index.js

```
import './index.css';
```

4.2.4 src\index.css

src\index.css

```

body{
  background-color: red;
}
#root{
  color:red;
}
#other{
  color:red;
}

```

4.2.5 src\index.html

src\index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div id="root"></div>
</body>
</html>
```

4.2.6 package.json

package.json

```
+ "sideEffects":["**/*.css"],
```

4.2.7 webpack.config.js

```
const path = require("path");
+const MiniCssExtractPlugin = require("mini-css-extract-plugin");
+const PurgecssPlugin = require("purgecss-webpack-plugin");
+const glob = require("glob");
+const PATHS = {
+  src: path.join(__dirname, "src"),
+};
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
+            loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      }
    ],
  },
  plugins: [
+    new MiniCssExtractPlugin({
+      filename: "[name].css",
+    }),
+    new PurgecssPlugin({
+      paths: glob.sync(`${PATHS.src}/**/*.`, { nodir: true }),
+    })
  ],
  devServer: {},
};
```

4.3 Tree Shaking

- 一个模块可以有多个方法，只要其中某个方法使用到了，则整个文件都会被打到bundle里面去，tree shaking就是只把用到的方法打入bundle,没用到的方法会在uglify阶段擦除掉
- 原理是利用es6模块的特点,只能作为模块顶层语句出现,import的模块名只能是字符串常量

4.3.1 开启

- webpack默认支持,可在production mode下默认开启
- 在package.json中配置:
 - "sideEffects": false 所有的代码都没有副作用 (都可以进行tree shaking)
 - 可能会把css和@babel/polyfill文件干掉 可以设置 "sideEffects":["*.css"]

4.3.2 没有导入和使用

functions.js

```
function func1(){
  return 'func1';
}
function func2(){
  return 'func2';
}
export {
  func1,
  func2
}
```

```
import {func2} from './functions';
var result2 = func2();
console.log(result2);
```

4.3.3 代码不会被执行，不可到达

```
if(false){
  console.log('false')
}
```

4.3.4 代码执行的结果不会被用到

```
import {func2} from './functions';
func2();
```

4.3.5 代码中只写不读的变量

```
var aabbcc='aabbcc';
aabbcc='eeffgg';
```

4.4 Scope Hoisting

- Scope Hoisting 可以让Webpack打包出来的代码文件更小、运行的更快，它又译作"作用域提升"，是在Webpack3中新推出的功能。
- scope hoisting的原理是将所有的模块按照引用顺序放在一个函数作用域里，然后适当地重命名一些变量以防止命名冲突

- 这个功能在mode为 production 下默认开启,开发环境要用 webpack.optimizeModuleConcatenationPlugin 插件

hello.js

```
export default 'Hello';
```

index.js

```
import str from './hello.js';  
console.log(str);
```

main.js

```
var hello = ('hello');  
console.log(hello);
```

5.运行速度优化

5.1 代码分割

- 对于大的Web应用来讲，将所有的代码都放在一个文件中显然是不够有效的，特别是当你的某些代码块是在某些特殊的时候才会被用到。
- webpack有一个功能就是将你的代码库分割成chunks语块，当代码运行到需要它们的时候再进行加载

5.1.1 入口点分割

- Entry Points：入口文件设置的时候可以配置
- 这种方法的问题
 - 如果入口 chunks 之间包含重复的模块(lodash)，那些重复模块都会被引入到各个 bundle 中
 - 不够灵活，并且不能将核心应用程序逻辑进行动态拆分代码

```
entry: {  
  index: "./src/index.js",  
  login: "./src/login.js"  
}
```

5.1.2 懒加载

- 用户当前需要什么功能就只加载这个功能对应的代码，也就是所谓的按需加载 在给单页应用做按需加载优化时
- 一般采用以下原则：
 - 对网站功能进行划分，每一类一个chunk
 - 对于首次打开页面需要的功能直接加载，尽快展示给用户,某些依赖大量代码的功能点可以按需加载
 - 被分割出去的代码需要一个按需加载的时机

5.1.2.1 hello.js

hello.js

```
module.exports = "hello";
```

5.1.2.2 index.js

```
document.querySelector('#clickBtn').addEventListener('click', () => {  
  import('./hello').then(result => {  
    console.log(result.default);  
  });  
});
```

5.1.2.3 index.html

```
<button id="clickBtn">点我</button>
```

5.1.3 prefetch

- 使用预先拉取，你表示该模块可能以后会用到。浏览器会在空闲时间下载该模块
- `prefetch` 的作用是告诉浏览器未来可能会使用到的某个资源，浏览器就会在闲时去加载对应的资源，若能预测到用户的行为，比如懒加载，点击到其它页面等则相当于提前预加载了需要的资源
- 此导入会让 `<link rel="prefetch" as="script" href="http://localhost:8080/hello.js">` 被添加至页面的头部。因此浏览器会在空闲时间预先拉取该文件。

webpack.config.js

```
document.querySelector('#clickBtn').addEventListener('click', () => {  
  import(/* webpackChunkName: 'hello', webpackPrefetch: true  
  */ './hello').then(result => {  
    console.log(result.default);  
  });  
});
```

5.1.6 提取公共代码

- 怎么配置单页应用?怎么配置多页应用?

5.1.6.1 为什么需要提取公共代码

- 大网站有多个页面，每个页面由于采用相同技术栈和样式代码，会包含很多公共代码，如果都包含进来会有问题
- 相同的资源被重复的加载，浪费用户的流量和服务器的成本；
- 每个页面需要加载的资源太大，导致网页首屏加载缓慢，影响用户体验。
- 如果能把公共代码抽离成单独文件进行加载能进行优化，可以减少网络传输流量，降低服务器成本

5.1.6.2 如何提取

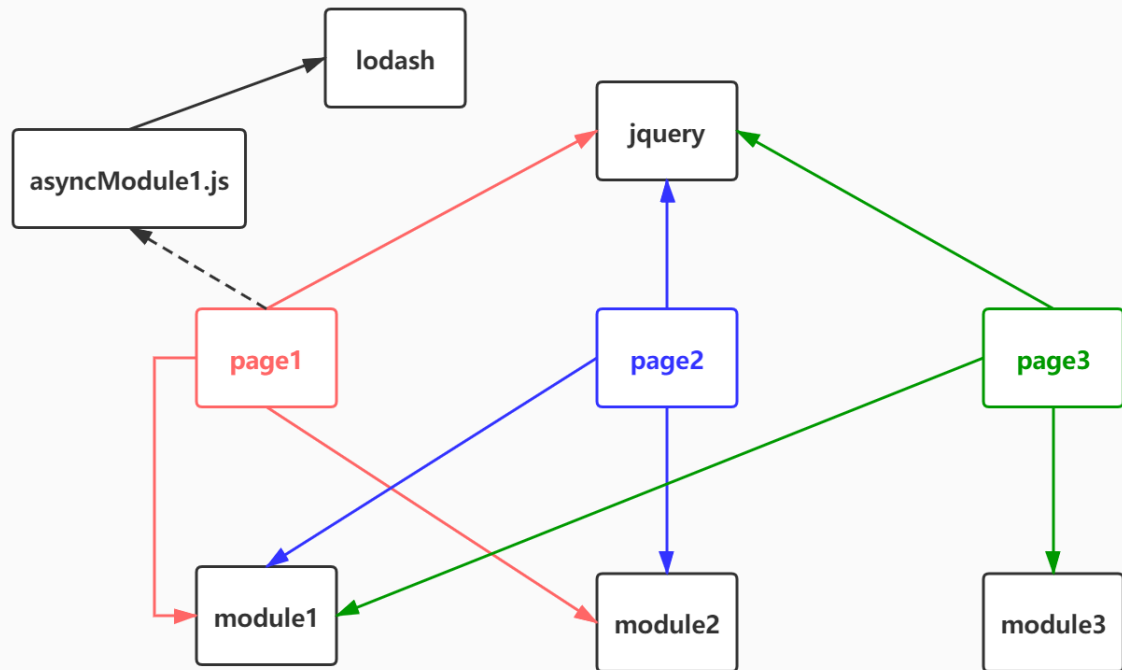
- 基础类库，方便长期缓存
- 页面之间的公用代码
- 各个页面单独生成文件

5.1.6.3 splitChunks

- [split-chunks-plugin](#)

5.1.6.3.1 module chunk bundle

- module: 就是js的模块化webpack支持commonJS、ES6等模块化规范, 简单来说就是你通过import语句引入的代码
- chunk: chunk是webpack根据功能拆分出来的, 包含三种情况
 - 你的项目入口 (entry)
 - 通过import()动态引入的代码
 - 通过splitChunks拆分出来的代码
- bundle: bundle是webpack打包之后的各个文件, 一般就是和chunk是一对一的关系, bundle就是对chunk进行编译压缩打包等处理之后的产出



5.1.6.3.2 默认配置

webpack.config.js

```

module.exports = {
  output: {
    filename: '[name].js',
    chunkFilename: '[name].js'
  },
  entry: {

```

```

    page1: "./src/page1.js",
    page2: "./src/page2.js",
    page3: "./src/page3.js",
  },
  optimization: {
    splitChunks: {
      chunks: 'all'
    }
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
      filename: 'page1.html',
      chunks: ['page1']
    }),
    new HtmlWebpackPlugin({
      template: './src/index.html',
      filename: 'page2.html',
      chunks: ['page2']
    }),
    new HtmlWebpackPlugin({
      template: './src/index.html',
      filename: 'page3.html',
      chunks: ['page3']
    })
  ]
}

```

src\page1.js

```

import utils1 from "./module1";
import utils2 from "./module2";
import $ from "jquery";
console.log(utils1, utils2, $);
import(/* webpackChunkName: "asyncModule1" */ "./asyncModule1");

```

src\page2.js

```

import utils1 from "./module1";
import utils2 from "./module2";
import $ from "jquery";
console.log(utils1, utils2, $);

```

src\page3.js

```

import utils1 from "./module1";
import utils3 from "./module3";
import $ from "jquery";
console.log(utils1, utils3, $);

```

src\module1.js

```

console.log("module1");

```

src\module2.js

```
console.log("module2");
```

src/module3.js

```
console.log("module3");
```

src/asyncModule1.js

```
import _ from 'lodash';  
console.log(_);
```

```
assets by chunk 813 KiB (id hint: vendors)  
  asset vendors-node_modules__lodash_4_17_20_lodash_lodash.js.js 531 KiB  
[emitted] (id hint: vendors) 1 related asset  
  asset vendors-node_modules__jquery_3_5_1_jquery_dist_jquery.js.js 282 KiB  
[emitted] (id hint: vendors) 1 related asset  
asset page1.js 15.8 KiB [emitted] (name: page1) 1 related asset  
asset page2.js 8.55 KiB [emitted] (name: page2) 1 related asset  
asset page3.js 8.55 KiB [emitted] (name: page3) 1 related asset  
asset asyncModule1.js 1.01 KiB [emitted] (name: asyncModule1) 1 related asset  
asset index.html 456 bytes [emitted]  
Entrypoint page1 298 KiB (381 KiB) = vendors-  
node_modules__jquery_3_5_1_jquery_dist_jquery.js.js 282 KiB page1.js 15.8 KiB 2  
auxiliary assets  
Entrypoint page2 291 KiB (374 KiB) = vendors-  
node_modules__jquery_3_5_1_jquery_dist_jquery.js.js 282 KiB page2.js 8.55 KiB 2  
auxiliary assets  
Entrypoint page3 291 KiB (374 KiB) = vendors-  
node_modules__jquery_3_5_1_jquery_dist_jquery.js.js 282 KiB page3.js 8.55 KiB 2  
auxiliary assets  
runtime modules 15.5 KiB 23 modules  
cacheable modules 811 KiB  
  modules by path ./src/*.js 531 bytes  
    ./src/page1.js 185 bytes [built] [code generated]  
    ./src/page2.js 119 bytes [built] [code generated]  
    ./src/page3.js 119 bytes [built] [code generated]  
    ./src/module1.js 23 bytes [built] [code generated]  
    ./src/module2.js 23 bytes [built] [code generated]  
    ./src/module3.js 23 bytes [built] [code generated]  
    ./src/asyncModule1.js 39 bytes [built] [code generated]  
  modules by path ./node_modules/ 811 KiB  
    ./node_modules/_jquery@3.5.1@jquery/dist/jquery.js 281 KiB [built] [code  
generated]  
    ./node_modules/_lodash@4.17.20@lodash/lodash.js 530 KiB [built] [code  
generated]  
webpack 5.9.0 compiled successfully in 6375 ms
```

```
page1.js => page1.js module1.js module2.js  
page2.js => page2.js module1.js module2.js  
page3.js => page3.js module1.js module3.js  
asyncModule1.js => asyncModule1.js  
vendors-node_modules__lodash_4_17_20_lodash_lodash.js=>lodash  
vendors-node_modules__jquery_3_5_1_jquery_dist_jquery.js=>jquery
```

5.1.6.3.3 自定义配置

webpack.config.js

```
module.exports = {
  optimization: {
    splitChunks: {
      chunks: 'all', //分割同步和异步代码块
      minSize: 0, //最小体积
      minRemainingSize: 0, //代码分割后的最小保留体积,默认等于minSize
      maxSize: 0, //最大体积
      minChunks: 1, //最小代码块
      maxAsyncRequests: 30, //最大异步请求数
      maxInitialRequests: 30, //最小异步请求数
      automaticNameDelimiter: '~', //名称分隔符
      enforceSizeThreshold: 50000, //执行拆分的大小阈值,忽略其他限制
      (minRemainingSize, maxAsyncRequests, maxInitialRequests)
    },
    cacheGroups: {
      defaultVendors: {
        test: /[\\/]node_modules[\\/]$/, //控制此缓存组选择哪些模块
        priority: -10, //一个模块属于多个缓存组,默认缓存组的优先级是负数,自定义缓存组的
        //如果当前代码块包含已经主代码块中分离出来的模块,那么它将被重用,而不是生成新的模
        //块。这可能会影响块的结果文件名。
        priority: 0
      },
      default: {
        minChunks: 2,
        priority: -20
      }
    }
  }
}
```

```
assets by chunk 813 KiB (id hint: defaultVendors)
  asset defaultVendors-node_modules__lodash_4_17_20_lodash_lodash.js.js 531 KiB
  [compared for emit] (id hint: defaultVendors) 1 related asset
  asset defaultVendors-node_modules__jquery_3_5_1_jquery_dist_jquery.js.js 282
  KiB [compared for emit] (id hint: defaultVendors) 1 related asset
assets by chunk 888 bytes (id hint: default)
  asset default-src_module1.js.js 444 bytes [compared for emit] (id hint:
  default) 1 related asset
  asset default-src_module2.js.js 444 bytes [compared for emit] (id hint:
  default) 1 related asset
asset page1.js 15.5 KiB [compared for emit] (name: page1) 1 related asset
asset page3~src_page3_js_601f375c.js 8.21 KiB [compared for emit] (name:
  page3~src_page3_js_601f375c) 1 related asset
asset page2.js 8.16 KiB [compared for emit] (name: page2) 1 related asset
asset asyncModule1.js 1.01 KiB [compared for emit] (name: asyncModule1) 1
  related asset
asset index.html 639 bytes [compared for emit]
asset page3~src_module3_js_b7ae565d.js 458 bytes [compared for emit] (name:
  page3~src_module3_js_b7ae565d) 1 related asset
Entrypoint page1 298 KiB (381 KiB) = defaultVendors-
  node_modules__jquery_3_5_1_jquery_dist_jquery.js.js 282 KiB default-
  src_module1.js.js 444 bytes default-src_module2.js.js 444 bytes page1.js 15.5
  KiB 4 auxiliary assets
```

```
Entrypoint page2 291 KiB (375 KiB) = defaultVendors-
node_modules__jquery_3_5_1_jquery_dist_jquery_js.js 282 KiB default-
src_module1_js.js 444 bytes default-src_module2_js.js 444 bytes page2.js 8.16
KiB 4 auxiliary assets
Entrypoint page3 291 KiB (375 KiB) = defaultVendors-
node_modules__jquery_3_5_1_jquery_dist_jquery_js.js 282 KiB default-
src_module1_js.js 444 bytes page3~src_module3_js_b7ae565d.js 458 bytes
page3~src_page3_js_601f375c.js 8.21 KiB 4 auxiliary assets
```

```
page1.js => page1.js
page2.js => page2.js
page3.js => page3.js
default-src_module1_js.js=>module1.js
default-src_module2_js.js=>module2.js
page3~src_module3_js_b7ae565d.js=>module3.js
asyncModule1.js => asyncModule1.js
vendors-node_modules__lodash_4_17_20_lodash_lodash_js.js=>lodash
vendors-node_modules__jquery_3_5_1_jquery_dist_jquery_js.js=>jquery
```

5.1.6.3.4 reuseExistingChunk

webpack.config.js

```
module.exports = {
  entry: {
    entry1: "./src/entry1.js",
    entry2: "./src/entry2.js"
  },
  optimization: {
    splitChunks: {
      chunks: "all",
      minSize: 0,
      maxSize: 0,
      cacheGroups: {
        default: false,
        commons: {
          minChunks: 1,
          reuseExistingChunk: true
        }
      }
    }
  }
}
```

src\A.js

```
export default 'A';
```

src\entry1.js

```
import(/* webpackChunkName:"A1" */'./A').then(item=>{
  console.log(item);
});
```

src\entry2.js

```
import(/* webpackChunkName:"A1" */'./A').then(item=>{
  console.log(item);
});
```

```
assets by status 5.52 KiB [emitted]
  asset commons-src_entry1_js.js 1.84 KiB [emitted] (id hint: commons)
  asset commons-src_entry2_js.js 1.84 KiB [emitted] (id hint: commons)
  asset A1.js 1.84 KiB [emitted] (name: A1) (id hint: commons)
assets by status 26.7 KiB [compared for emit]
  asset entry1.js 13.4 KiB [compared for emit] (name: entry1)
  asset entry2.js 13.4 KiB [compared for emit] (name: entry2)
```

```
assets by status 28.6 KiB [compared for emit]
  asset entry1.js 13.4 KiB [compared for emit] (name: entry1)
  asset entry2.js 13.4 KiB [compared for emit] (name: entry2)
  asset commons-src_A_js.js 1.85 KiB [compared for emit] (id hint: commons)
assets by status 3.71 KiB [emitted]
  asset commons-src_entry1_js.js 1.85 KiB [emitted] (id hint: commons)
  asset commons-src_entry2_js.js 1.85 KiB [emitted] (id hint: commons)
```

5.2 CDN

- 最影响用户体验的是网页首次打开时的加载等待。导致这个问题的根本是网络传输过程耗时大，CDN的作用就是加速网络传输。
- CDN 又叫内容分发网络，通过把资源部署到世界各地，用户在访问时按照就近原则从离用户最近的服务器获取资源，从而加速资源的获取速度
- 用户使用浏览器第一次访问我们的站点时，该页面引入了各式各样的静态资源，如果我们能做到持久化缓存的话，可以在 http 响应头加上 Cache-control 或 Expires 字段来设置缓存，浏览器可以将这些资源——缓存到本地
- 用户在后续访问的时候，如果需要再次请求同样的静态资源，且静态资源没有过期，那么浏览器可以直接走本地缓存而不再通过网络请求资源
- 缓存配置
 - HTML文件不缓存，放在自己的服务器上，关闭自己服务器的缓存，静态资源的URL变成指向CDN服务器的地址
 - 静态的JavaScript、CSS、图片等文件开启CDN和缓存，并且文件名带上HASH值
 - 为了并行加载不阻塞，把不同的静态资源分配到不同的CDN服务器上
- 域名限制
 - 同一时刻针对同一个域名的资源并行请求是有限制
 - 可以把这些静态资源分散到不同的 CDN 服务上去
 - 多个域名后会增加域名解析时间
 - 可以通过在 HTML HEAD 标签中 加入去预解析域名，以降低域名解析带来的延迟

webpack.config.js

```
const path = require("path");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const PurgecssPlugin = require("purgecss-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const UploadPlugin = require("../plugins/UploadPlugin");
```

```

const glob = require("glob");
const PATHS = {
  src: path.join(__dirname, "src"),
};
module.exports = {
  mode: "development",
  devtool: false,
  context: process.cwd(),
  entry: {
    main: "./src/index.js",
  },
  output: {
    path: path.resolve(__dirname, "dist"),
+   filename: "[name].[hash].js",
+   chunkFilename: "[name].[hash].chunk.js",
+   publicPath: "http://img.zhufengpeixun.cn/",
  },
  optimization: {
    minimize: true,
    minimizer: [
      //压缩JS
      /* new TerserPlugin({
        sourceMap: false,
        extractComments: false,
      }),
      //压缩CSS
      new OptimizeCSSAssetsPlugin({}), */
    ],
    //自动分割第三方模块和公共模块
    splitChunks: {
      chunks: "all", //默认作用于异步chunk, 值为all/initial/async
      minSize: 0, //默认值是30kb,代码块的最小尺寸
      minChunks: 1, //被多少模块共享,在分割之前模块的被引用次数
      maxAsyncRequests: 2, //限制异步模块内部的并行最大请求数的,说白了你可以理解为是每个
import()它里面的最大并行请求数量
      maxInitialRequests: 4, //限制入口的拆分数量
      name: true, //打包后的名称,默认是chunk的名字通过分隔符(默认是~)分隔开,如vendor~
      automaticNameDelimiter: "~", //默认webpack将会使用入口名和代码块的名称生成命名,比
如 'vendors~main.js'
      cacheGroups: {
        //设置缓存组用来抽取满足不同规则的chunk,下面以生成common为例
        vendors: {
          chunks: "all",
          test: /node_modules/, //条件
          priority: -10, ///优先级,一个chunk很可能满足多个缓存组,会被抽取到优先级高的缓存组中,为了能够让自定义缓存组有更高的优先级(默认0),默认缓存组的priority属性为负值.
        },
        commons: {
          chunks: "all",
          minSize: 0, //最小提取字节数
          minChunks: 2, //最少被几个chunk引用
          priority: -20,
          reuseExistingChunk: true, //如果该chunk中引用了已经被抽取的chunk,直接引用该
chunk,不会重复打包代码
        },
      },
    },
  },
  //为了长期缓存保持运行时代码块是单独的文件

```

```

    /* runtimeChunk: {
      name: (entrypoint) => `runtime-${entrypoint.name}`,
    }, */
  },
  module: {
    rules: [
      {
        test: /\.js/,
        include: path.resolve(__dirname, "src"),
        use: [
          {
            loader: "babel-loader",
            options: {
              presets: [
                ["@babel/preset-env", { modules: false }],
                "@babel/preset-react",
              ],
            },
          },
        ],
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      },
      {
        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,
              },
            },
          },
        ],
      },
    ],
  },

```



```

    ],
  },
],
},
plugins: [
  new HtmlWebpackPlugin({
    inject: true,
    template: './src/index.html',
  }),
  new MiniCssExtractPlugin({
+   filename: '[name].[hash].css',
  }),
  new PurgecssPlugin({
    paths: glob.sync(`${PATHS.src}/**/*`, { nodir: true }),
  }),
  new UploadPlugin({}),
],
devServer: {},
};

```

6.开发体验

6.1 HMR

6.1.1 src\title.js

src\title.js

```
module.exports = `hello`;
```

6.1.2 src\index.js

src\index.js

```

import './index.css';
function render(){
  document.getElementById('root').innerHTML = require('./title');
}
render();
if(module.hot){
  module.hot.accept('./title', render);
}

```

6.1.3 src\index.css

src\index.css

```

body{
  background-color: green;
}

```

6.1.4 webpack.config.js

```
module.exports = {
  devServer: {
    compress: true,
    port: 3000,
    open: true,
    hot: true
  }
}
```

6.2 PWA

- PWA(Progressive Web Apps)PWA 借助 Service Worker 缓存网站的静态资源，甚至是网络请求，使网站在离线时也能访问。并且我们能够为网站指定一个图标添加在手机桌面，实现点击桌面图标即可访问网站
- Service Worker 是浏览器在后台独立于网页运行的脚本。是它让 PWA 拥有极快的访问速度和离线运行能力
- workbox 是由谷歌浏览器团队发布，用来协助创建 PWA 应用的 JavaScript 库
- [workbox-webpack-plugin](#)

6.2.1 下载

```
cnpm i workbox-webpack-plugin -D
```

6.2.2 webpack.config.js

```
const workboxWebpackPlugin = require('workbox-webpack-plugin');
module.exports = {
  plugins:[
    new workboxWebpackPlugin.GenerateSW({
      clientsClaim: true,//不允许遗留任何旧的ServiceWorkers
      skipwaiting: true//帮助ServiceWorkers 快速启用
    })
  ]
}
```

7.附录

7.1 环境

7.1.1 模式(mode)

- 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

选项	描述
development	会将 process.env.NODE_ENV 的值设为 development。启用 NamedChunksPlugin 和 NamedModulesPlugin
production	会将 process.env.NODE_ENV 的值设为 production。启用 FlagDependencyUsagePlugin, FlagIncludedChunksPlugin, ModuleConcatenationPlugin, NoEmitOnErrorsPlugin, OccurrenceOrderPlugin, SideEffectsFlagPlugin 和 UglifyJsPlugin

7.1.2 环境差异

- 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
- 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
- 其默认值为 production

7.1.3 区分环境

- `--mode` 用来设置模块内的 `process.env.NODE_ENV`
- `--env` 用来设置webpack配置文件的函数参数
- [cross-env](#)用来设置node环境的 `process.env.NODE_ENV`
- [dotenv](#)可以按需加载不同的环境变量文件
- [define-plugin](#)用来配置在 编译时候 用的 全局常量

7.1.3.1 安装

```
cnpm i cross-env dotenv terser-webpack-plugin optimize-css-assets-webpack-plugin -D
```

7.1.3.2 mode默认值

- webpack的mode默认为 `production`
- `webpack serve` 的mode默认为 `development`
- 可以在模块内通过 `process.env.NODE_ENV` 获取当前的环境变量,无法在 `webpack`配置文件 中获取此变量

```
"scripts": {
  "build": "webpack",
  "start": "webpack serve"
},
```

index.js

```
console.log(process.env.NODE_ENV); // development | production
```

webpack.config.js

```
console.log('NODE_ENV', process.env.NODE_ENV); // undefined
```

7.1.3.3 命令行传mode

- 同配置1

```
"scripts": {  
  "build": "webpack --mode=production",  
  "start": "webpack --mode=development serve"  
},
```

7.1.3.4 命令行配置env

- 无法在模块内通过 `process.env.NODE_ENV` 访问
- 可以通过 webpack 配置文件中 中通过 函数 获取当前环境变量

```
"scripts": {  
  "dev": "webpack serve --env=development",  
  "build": "webpack --env=production",  
}
```

index.js

```
console.log(process.env.NODE_ENV); // undefined
```

webpack.config.js

```
console.log('NODE_ENV', process.env.NODE_ENV); // undefined
```

```
const TerserWebpackPlugin = require('terser-webpack-plugin');  
const OptimizeCssAssetsWebpackPlugin = require('optimize-css-assets-webpack-plugin');  
  
module.exports = (env) => {  
  console.log('env', env); // {development:true} | {production:true}  
  return {  
    + optimization: {  
    +   minimize: env && env.production,  
    +   minimizer: (env && env.production) ? [  
    +     new TerserWebpackPlugin({  
    +       parallel: true // 开启多进程并行压缩  
    +     }),  
    +     new OptimizeCssAssetsWebpackPlugin({})  
    +   ] : []  
    + },  
  }  
};
```

7.1.3.5 mode配置

- 和命令行配置2一样

```
module.exports = {
  mode: 'development'
}
```

7.1.3.6 DefinePlugin

- 设置全局变量(不是 window),所有模块都能读取到该变量的值
- 可以在任意模块内通过 `process.env.NODE_ENV` 获取当前的环境变量
- 但无法在 node 环境 (webpack 配置文件中) 下获取当前的环境变量

7.1.3.6.1 webpack.config.js

```
console.log('process.env.NODE_ENV', process.env.NODE_ENV); // undefined
console.log('NODE_ENV', NODE_ENV); // error !!!
module.exports = {
  plugins: [
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify('development'), // 注意用双引号引起来, 否则就成变量了
      'NODE_ENV': JSON.stringify('production'),
    })
  ]
}
```

7.1.3.6.2 src/index.js

```
console.log(NODE_ENV); // production
```

7.1.3.6.3 src/logger.js

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

7.1.4 cross-env

- 只能设置 node 环境 下的变量 `NODE_ENV`

package.json

```
"scripts": {
  "build": "cross-env NODE_ENV=development webpack"
}
```

webpack.config.js

```
console.log('process.env.NODE_ENV', process.env.NODE_ENV); // development
```

7.1.4 env

7.1.4.1 .env

```
NODE_ENV=development
```

7.1.4.2 webpack.config.js

webpack.config.js

```
+require('dotenv').config({path: path.resolve(__dirname, '.env')})  
+console.log(process.env.NODE_ENV);
```

7.2 JavaScript兼容性

7.2.1 Babel

- [Babel](#) 是一个 JavaScript 编译器
- Babel 是一个工具链，主要用于将 ECMAScript 2015+ 版本的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中
- Babel 能为你做的事情
 - 语法转换

7.2.1.1 安装

```
cnpm i @babel/core @babel/cli -D
```

7.2.1.2 命令行使用

7.2.1.2.1 src\index.js

```
const sum = (a,b)=>a+b;  
console.log(sum(1,2));
```

7.2.1.2.2 package.json

```
"scripts": {  
  "build": "babel src --out-dir dist --watch"  
},
```

7.2.2 插件

- Babel 是一个编译器（输入源码 => 输出编译后的代码）。就像其他编译器一样，编译过程分为三个阶段：解析、转换和打印输出。
- 现在，Babel 虽然开箱即用，但是什么动作都不做。它基本上类似于 `const babel = code => code;`，将代码解析之后再输出同样的代码。如果想要 Babel 做一些实际的工作，就需要为其添加[插件](#)

7.2.2.1 安装

```
cnpm i @babel/plugin-transform-arrow-functions -D
```

7.2.2.2 .babelrc

```
{
  "plugins": ["@babel/plugin-transform-arrow-functions"]
}
```

7.2.3 预设

- [@babel/preset-env](#)可以让你使用最新的JavaScript语法,而不需要去管理语法转换器(并且可选的支持目标浏览器环境的polyfills)

```
module.exports = function () {
  return { plugins: ["pluginA", "pluginB", "pluginC"] }
}
```

7.2.3.1 安装

```
cnpm install --save-dev @babel/preset-env
```

7.2.3.2 .babelrc

```
{
  "presets": ["@babel/preset-env"]
}
```

7.2.3.3 browsers

- [@babel/preset-env](#) 会根据你配置的目标环境,生成插件列表来编译
- 可以使用使用 `.browserslistrc` 文件来指定目标环境
- [browserslist](#)详细配置

```
//.browserslistrc
> 0.25%
not dead
```

```
last 2 Chrome versions
```

7.2.4 polyfill

- [@babel/preset-env](#)默认只转换新的javascript语法,而不转换新的API,比如 Iterator, Generator, Set, Maps, Proxy, Reflect, Symbol, Promise 等全局对象。以及一些在全局对象上的方法(比如 Object.assign)都不会转码
- 比如说, ES6在Array对象上新增了Array.from方法, Babel就不会转码这个方法,如果想让这个方法运行,必须使用 babel-polyfill来转换等
- polyfill 的中文意思是垫片,所谓垫片就是垫平不同浏览器或者不同环境下的差异,让新的内置函数、实例方法等在低版本浏览器中也可以使用
- 官方给出[@babel/polyfill](#)和 `babel-runtime` 两种解决方案来解决这种全局对象或全局对象方法不足的问题
- `babel-runtime` 适合在组件和类库项目中使用,而 `babel-polyfill` 适合在业务项目中使用。

7.2.4.1 polyfill

- [@babel/polyfill](#)模块可以模拟完整的 ES2015+ 环境
- 这意味着可以使用诸如 `Promise` 和 `WeakMap` 之类的新的内置对象、`Array.from` 或 `Object.assign` 之类的静态方法、`Array.prototype.includes` 之类的实例方法以及生成器函

数

- 它是通过向全局对象和内置对象的 `prototype` 上添加方法来实现的。比如运行环境中不支持 `Array.prototype.find` 方法，引入 `polyfill`，我们就可以使用 es6 方法来编写了，但是缺点就是会造成全局空间污染
- V7.4.0 版本开始，`@babel/polyfill` 已经被废弃

7.2.4.1.1 安装

```
cnpm install --save @babel/polyfill core-js@3
cnpm install --save-dev webpack webpack-cli babel-loader
```

7.2.4.1.2 useBuiltIns:false

- "useBuiltIns": false 此时不对 `polyfill` 做操作。如果引入 `@babel/polyfill`，则无视配置的浏览器兼容，引入所有的 `polyfill`

src\index.js

```
import "@babel/polyfill";
console.log(Array.isArray([]));
let p = new Promise();
```

webpack.config.js

```
const path = require('path');
module.exports = {
  mode: 'development',
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].js',
  },
  module: {
    rules: [
      {
        test: /\.js?$/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: [
              ["@babel/preset-env",
                {
                  useBuiltIns: false
                }
              ]
            ]
          },
        },
      },
    ],
  },
  plugins: []
};
```

package.json


```
"scripts": {
  "build": "babel src --out-dir dist --watch",
  "pack": "webpack --mode=development"
},
```

7.2.4.1.3 useBuiltIns:entry

- "useBuiltIns": "entry" 根据配置的浏览器兼容，引入浏览器不兼容的 polyfill。需要在入口文件手动添加 `import '@babel/polyfill'`，会自动根据 `browserslist` 替换成浏览器不兼容的所有 polyfill
- 这里需要指定 core-js 的版本, 如果 "corejs": 3, 则 `import '@babel/polyfill'` 需要改成 `import 'core-js/stable';import 'regenerator-runtime/runtime'`
- `core-js@2` 分支中已经不会再添加新特性，新特性都会添加到 `core-js@3`, 比如 `Array.prototype.flat()`

webpack.config.js

```
module: {
  rules: [
    {
      test: /\.js?$/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: [
            ["@babel/preset-env",
+             {
+               useBuiltIns: 'entry',
+               corejs: 2,
+               targets: "last 2 Chrome versions"
            }
          ]
        },
      },
    ],
  ],
},
```

src/index.js

```
import "@babel/polyfill";
console.log(Array.isArray([]));
let p = new Promise(resolve=>resolve('ok'));
p.then(result=>console.log(result));
```

webpack.config.js

```
module: {
  rules: [
    {
      test: /\.js?$/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: [
```

```

        ["@babel/preset-env",
        {
            useBuiltIns: 'entry',
+           corejs: 3,
            targets: "last 2 Chrome versions",
        }]
    ],
    },
    },
    ],
    },
}

```

src/index.js

```

+import 'core-js/stable';
+import 'regenerator-runtime/runtime'
console.log(Array.isArray([]));
let p = new Promise(resolve=>resolve('ok'));
p.then(result=>console.log(result));

```

7.2.4.1.4 useBuiltIns:usage

- `"useBuiltIns": "usage"` usage 会根据配置的浏览器兼容，以及你代码中用到的 API 来进行 polyfill，实现了按需添加

src\index.js

```

console.log(Array.isArray([]));
let p = new Promise(resolve=>resolve('ok'));
p.then(result=>console.log(result));
async function fn(){

```

webpack.config.js

```

module: {
  rules: [
    {
      test: /\.js?$/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: [
            ["@babel/preset-env",
+           {
+             useBuiltIns: 'usage',
+             corejs: 3,
              targets: "> 0.25%, not dead",
            }]
          ]
        }
      },
    },
  ],
}

```

```

"use strict";

require("core-js/stable");

require("regenerator-runtime/runtime");

console.log(Array.isArray([]));
let p = new Promise(resolve => resolve('ok'));
p.then(result => console.log(result));
async function d() {}

```

7.2.5 babel-runtime

- Babel为了解决全局空间污染的问题，提供了单独的包[babel-runtime](#)用以提供编译模块的工具函数
- 简单说 `babel-runtime` 更像是一种按需加载的实现，比如你哪里需要使用 Promise，只要在这个文件头部 `require Promise from 'babel-runtime/core-js/promise'` 就行了

7.2.5.1 安装

```
cnpm i babel-runtime -D
```

7.2.5.2 src/index.js

```

import Promise from 'babel-runtime/core-js/promise';
const p = new Promise((resolve)=> {
  resolve('ok');
});
console.log(p);

```

7.2.5.3 webpack.config.js

```

module: {
  rules: [
    {
      test: /\.js?$/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: [
+           ["@babel/preset-env"]
          ]
        },
      },
    ],
  ],
},

```

7.2.6 babel-plugin-transform-runtime

- 启用插件 `babel-plugin-transform-runtime` 后，Babel就会使用 `babel-runtime` 下的工具函数。

- `babel-plugin-transform-runtime` 插件能够将这些工具函数的代码转换成 `require` 语句，指向为对 `babel-runtime` 的引用
- `babel-plugin-transform-runtime` 就是可以在我们使用新 API 时自动 `import babel-runtime` 里面的 `polyfill`
 - 当我们使用 `async/await` 时，自动引入 `babel-runtime/regenerator`
 - 当我们使用 ES6 的静态事件或内置对象时，自动引入 `babel-runtime/core-js`
 - 移除内联 `babel helpers` 并替换使用 `babel-runtime/helpers` 来替换
- `babel-plugin-transform-runtime` 自带的是 `core-js@3`，如果配置 `corejs` 配置为 2 需要单独安装 `@babel/runtime-corejs2`
- `@babel/polyfill` 自带的 `core-js@2`，如果配置 `corejs` 配置为 3 需要单独安装 `core-js@3`
- `@babel/plugin-transform-runtime` 可以减少编译后代码的体积外，我们使用它还有一个好处，它可以为代码创建一个沙盒环境，如果使用 `@babel/polyfill` 及其提供的内置程序（例如 `Promise`，`Set` 和 `Map`），则它们将污染全局范围。虽然这对于应用程序或命令行工具可能是可以的，但是如果你的代码是要发布供他人使用的库，或者无法完全控制代码运行的环境，则将成为一个问题

7.2.6.1 安装

```
cnpm i @babel/plugin-transform-runtime @babel/runtime-corejs2 @babel/runtime-corejs3 -D
```

7.2.6.2 webpack.config.js

```
{
  test: /\.jsx?$/,
  use: {
    loader: 'babel-loader',
    options: {
      presets: [['@babel/preset-env', {
        targets: "> 0.25%, not dead",
      }], '@babel/preset-react'],
      plugins: [
+       [
+         "@babel/plugin-transform-runtime",
+         {
+           corejs: 2, // 当我们使用 ES6 的静态事件或内置对象时自动引入 babel-
runtime/core-js
+           helpers: true, // 移除内联 babel helpers 并替换使用 babel-
runtime/helpers 来替换
+           regenerator: true, // 是否开启 generator 函数转换成使用 regenerator
runtime 来避免污染全局域
+         ],
+       ],
      ['@babel/plugin-proposal-decorators', { legacy: true }],
      ['@babel/plugin-proposal-class-properties', { loose: true }],
    ],
  },
},
},
```

7.2.6.3 src/index.js

```
//corejs
```

```

const p = new Promise(()=> {});
console.log(p);

//helpers
class A {

}
class B extends A {

}
console.log(new B());
//regenerator
function* gen() {

}
console.log(gen());

```

7.2.7 执行顺序

- 插件在 Presets 前运行
- 插件顺序从前往后排列
- presets顺序从后往

webpack.config.js

```

const path = require('path');
+const plugin1 = path.resolve(__dirname, 'plugins', 'plugin1.js');
+const plugin2 = path.resolve(__dirname, 'plugins', 'plugin2.js');
+const plugin3 = path.resolve(__dirname, 'plugins', 'plugin3.js');
+const plugin4 = path.resolve(__dirname, 'plugins', 'plugin4.js');
+const plugin5 = path.resolve(__dirname, 'plugins', 'plugin5.js');
+const plugin6 = path.resolve(__dirname, 'plugins', 'plugin6.js');
function preset1() {
  return { plugins: [plugin5, plugin6] }
}
function preset2() {
  return { plugins: [plugin3, plugin4] }
}
module.exports = {
  mode: 'development',
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].js',
  },
  module: {
    rules: [
      {
        test: /\.js?$/,
        use: {
          loader: 'babel-loader',
+          options: {
+            plugins: [
+              plugin1,
+              plugin2
+            ],
+            presets: [

```

```
+ preset1,  
+ preset2  
+ ]  
+ },  
+ },  
+ ],  
+ },  
+ plugins: []  
};
```

```
{
  "presets": ["preset1", "preset"],
  "plugins": ["plugin1", "plugin2"]
}
```

plugin1
plugin2
plugin3
plugin4
plugin5
plugin6

7.3 sourcemap

- [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](#)可以单独缓存map，重建性能更高
- [source-map](#)

7.3.1 配置项

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

7.3.2 关键字

- 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
- 关键字可以任意组合，但是有顺序要求

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

7.3.3 webpack.config.js

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

7.3.4 组合规则

- [inline-|hidden-|eval-][nosources-][cheap-[module-]]source-map
- source-map 单独在外部生成完整的sourcemap文件，并且在目标文件里建立关联,能提示错误代码的准确原始位置
- inline-source-map 以base64格式内联在打包后的文件中，内联构建速度更快,也能提示错误代码的准确原始位置
- hidden-source-map 会在外部生成sourcemap文件,但是在目标文件里没有建立关联,不能提示错误代码的准确原始位置
- eval-source-map 会为每一个模块生成一个单独的sourcemap文件进行内联，并使用 eval 执行
- nosources-source-map 也会在外部分生成sourcemap文件,能找到源代码位置，但源代码内容为空
- cheap-source-map 外部生成sourcemap文件,不包含列和loader的map
- cheap-module-source-map 外部生成sourcemap文件,不包含列的信息但包含loader的map

7.3.5 最佳实践

7.3.5.1 开发环境

- 我们在开发环境对sourceMap的要求是：速度快，调试更友好
- 要想速度快 推荐 eval-cheap-source-map
- 如果想调试更友好 cheap-module-source-map
- 折中的选择就是 eval-source-map

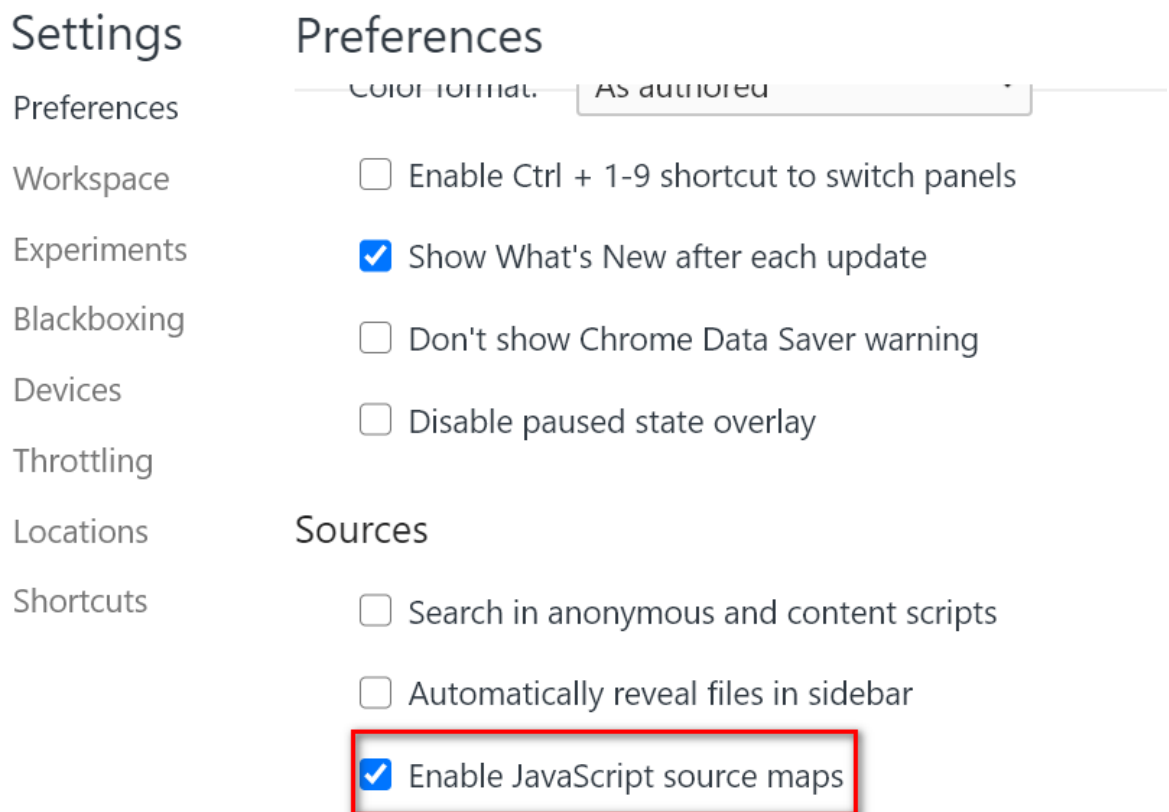
7.3.5.2 生产环境

- 首先排除内联，因为一方面我们隐藏了源代码，另一方面要减少文件体积
- 要想调试友好 `sourcemap>cheap-source-map/cheap-module-source-map>hidden-source-map/nosources-sourcemap`
- 要想速度快 优先选择 `cheap`
- 折中的选择就是 `hidden-source-map`

7.3.6 调试代码

7.3.6.1 测试环境调试

- [source-map-dev-tool-plugin](#)实现了对 source map 生成，进行更细粒度的控制
 - filename (string): 定义生成的 source map 的名称（如果没有值将会变成 inlined）。
 - append (string): 在原始资源后追加给定值。通常是 #sourceMappingURL 注释。[url] 被替换成 source map 文件的 URL
- 市面上流行两种形式的文件指定，分别是以 @ 和 # 符号开头的, @ 开头的已经被废弃



webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const OptimizeCssAssetsWebpackPlugin = require('optimize-css-assets-webpack-plugin');
const TerserPlugin = require('terser-webpack-plugin');
+const FileManagerPlugin = require('filemanager-webpack-plugin');
+const webpack = require('webpack');

module.exports = {
  mode: 'none',
  devtool: false,
  entry: './src/index.js',
  optimization: {
```



```

    minimize: true,
    minimizer: [
      new TerserPlugin(),
    ],
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].js',
    publicPath: '/',
  },
  devServer: {
    contentBase: path.resolve(__dirname, 'dist'),
    compress: true,
    port: 8080,
    open: true,
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        loader: 'eslint-loader',
        enforce: 'pre',
        options: { fix: true },
        exclude: /node_modules/,
      },
      {
        test: /\.jsx?$/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: [[
              '@babel/preset-env',
              {
                useBuiltIns: 'usage', // 按需要加载polyfill
                corejs: {
                  version: 3, // 指定core-js版本
                },
                targets: { // 指定要兼容到哪些版本的浏览器
                  chrome: '60',
                  firefox: '60',
                  ie: '9',
                  safari: '10',
                  edge: '17',
                },
              },
            ]],
            '@babel/preset-react',
          },
          plugins: [
            ['@babel/plugin-proposal-decorators', { legacy: true }],
            ['@babel/plugin-proposal-class-properties', { loose: true }],
          ],
        },
      },
      {
        include: path.join(__dirname, 'src'),
        exclude: /node_modules/,
      },
      { test: /\.txt$/, use: 'raw-loader' },
      { test: /\.css$/, use: [MiniCssExtractPlugin.loader, 'css-loader', 'postcss-loader'] },
    ],
  },

```

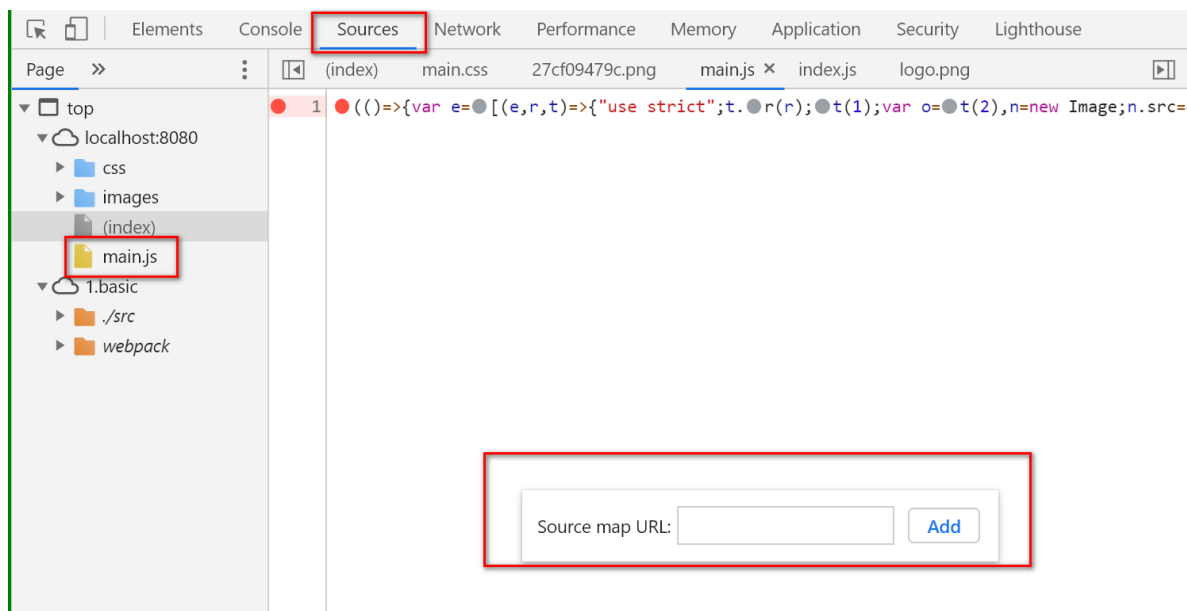
```

    { test: /\.less$/, use: [MiniCssExtractPlugin.loader, 'css-loader',
'postcss-loader', 'less-loader'] },
    { test: /\.scss$/, use: [MiniCssExtractPlugin.loader, 'css-loader',
'postcss-loader', 'sass-loader'] },
    {
      test: /\. (jpg|png|bmp|gif|svg)$/,
      use: [{
        loader: 'url-loader',
        options: {
          esModule: false,
          name: '[hash:10].[ext]',
          limit: 8 * 1024,
          outputPath: 'images',
          publicPath: '/images',
        },
      }],
    },
    {
      test: /\.html$/,
      loader: 'html-loader',
    },
  ],
},
plugins: [
  new HtmlWebpackPlugin({
    template: './src/index.html',
    minify: {
      collapseWhitespace: true,
      removeComments: true,
    },
  }),
  new MiniCssExtractPlugin({
    filename: 'css/[name].css',
  }),
  new OptimizeCssAssetsWebpackPlugin(),
+   new webpack.SourceMapDevToolPlugin({
+     append: '\n//# sourceMappingURL=http://127.0.0.1:8081/[url]',
+     filename: '[file].map',
+   }),
+   new FileManagerPlugin({
+     events: {
+       onEnd: {
+         copy: [{
+           source: './dist/*.map',
+           destination: 'C:/aprepare/zhufengwebpack2021/1.basic/sourcemap',
+         }],
+         delete: ['./dist/*.map'],
+       },
+     },
+   }),
],
};

```

7.3.6.2 生产环境调试

- webpack打包仍然生成sourceMap，但是将map文件挑出放到本地服务器，将不含有map文件的部署到服务器



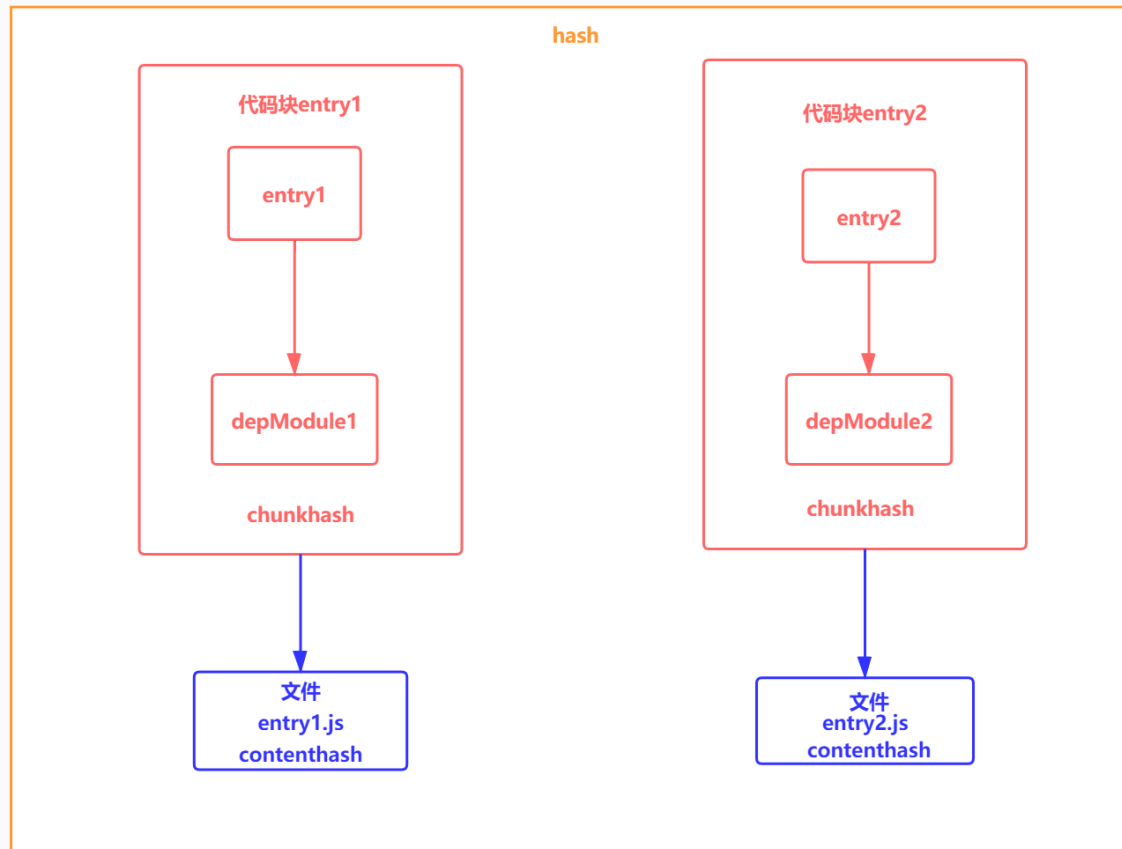
7.4 hash、chunkhash和contenthash

- 文件指纹 是指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。

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
hash	每次webpack构建时生成一个唯一的hash值
chunkhash	根据chunk生成hash值，来源于同一个chunk，则hash值就一样
contenthash	根据内容生成hash值，文件内容相同hash值就相同

7.4.1 hash计算



```
function createHash(){
  return require('crypto').createHash('md5');
}
let entry = {
  entry1: 'entry1',
  entry2: 'entry2'
}
let entry1 = 'require depModule1';//模块entry1
let entry2 = 'require depModule2';//模块entry2

let depModule1 = 'depModule1';//模块depModule1
let depModule2 = 'depModule2';//模块depModule2
//如果都使用hash的话，因为这是工程级别的，即每次修改任何一个文件，所有文件名的hash至都将改变。
所以一旦修改了任何一个文件，整个项目的文件缓存都将失效
let hash = createHash()
  .update(entry1)
  .update(entry2)
  .update(depModule1)
  .update(depModule2)
  .digest('hex');
console.log('hash',hash)
//chunkhash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。
//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
let entry1ChunkHash = createHash()
  .update(entry1)
  .update(depModule1).digest('hex');;
console.log('entry1ChunkHash',entry1ChunkHash);

let entry2ChunkHash = createHash()
  .update(entry2)
```

```

.update(depModule2).digest('hex');;
console.log('entry2ChunkHash',entry2ChunkHash);

let entry1File = entry1+depModule1;
let entry1ContentHash = createHash()
.update(entry1File).digest('hex');;
console.log('entry1ContentHash',entry1ContentHash);

let entry2File = entry2+depModule2;
let entry2ContentHash = createHash()
.update(entry2File).digest('hex');;
console.log('entry2ContentHash',entry2ContentHash);

```

7.4.2 hash

- 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

```

module.exports = {
+  entry: {
+    main: './src/index.js',
+    vender:['lodash']
+  },
  output:{
    path:path.resolve(__dirname,'dist'),
+    filename:'[name].[hash].js'
  },
  plugins: [
    new MiniCssExtractPlugin({
+    filename: "css/[name].[hash].css"
    })
  ]
};

```

7.4.2 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。这样子是无法实现缓存效果，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

```

module.exports = {
  entry: {
    main: './src/index.js',
    vender:['lodash']
  },
  output:{
    path:path.resolve(__dirname,'dist'),
+    filename:'[name].[chunkhash].js'
  },
  plugins: [
    new MiniCssExtractPlugin({

```

```
+     filename: "css/[name].[chunkhash].css"
    })
  ]
};
```

7.4.3 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

```
module.exports = {
  plugins: [
    new MiniCssExtractPlugin({
+     filename: "css/[name].[contenthash].css"
    })
  ],
};
```

8.更多内容

- Webpack5新特性实战
- 从零实现tree-shaking
- Webpack5模块联邦实现微前端