

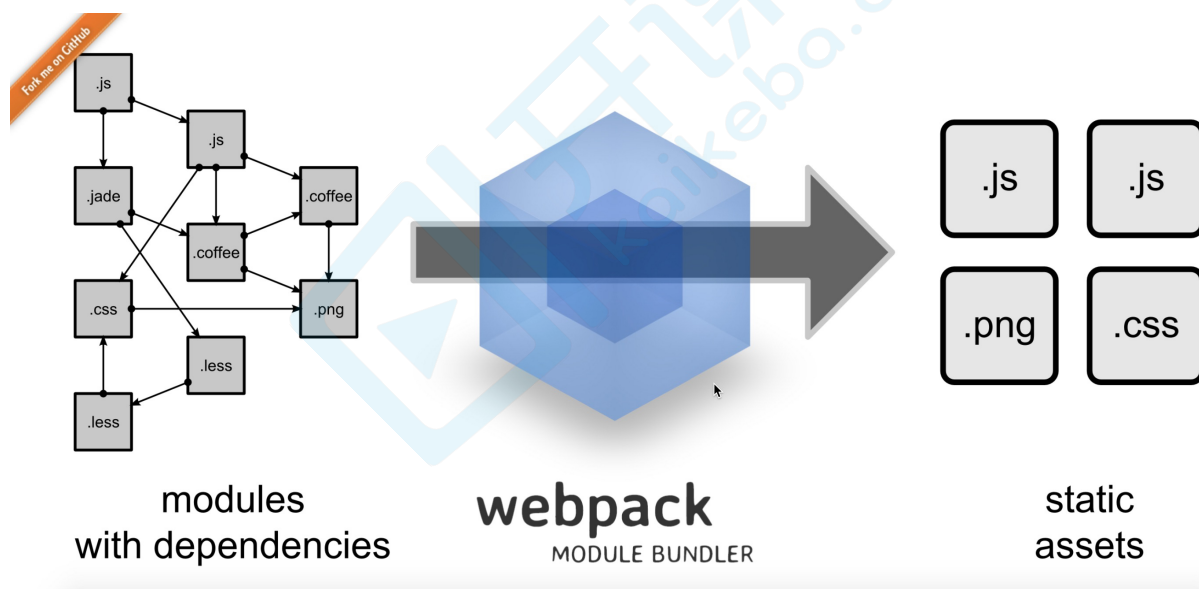
Webpack-Day1



课前准备

- nodeJS
- webpack

什么是webpack



webpack is a module bundler.(模块打包工具)

- 官方网站: <https://webpack.js.org/>
- 中文网站: <https://www.webpackjs.com/>

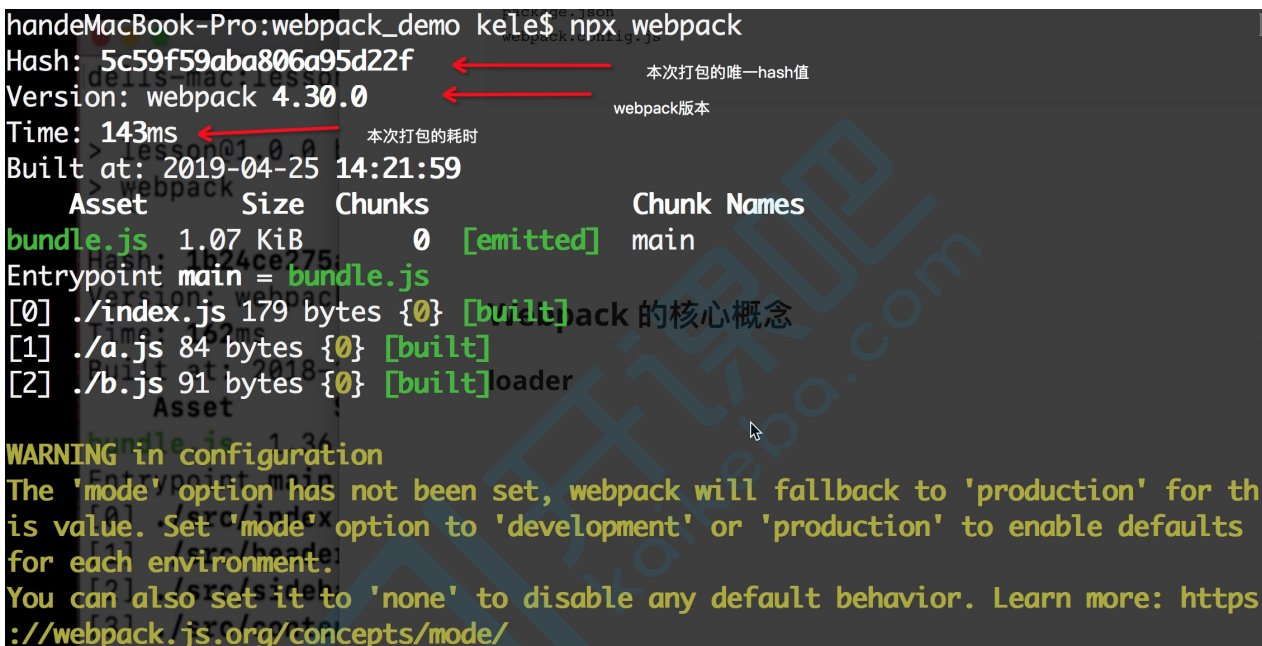
测试：启动webpack打包

```
// es module 模块引入
// commonJs 模块引入

import add from "./a"; //需要使用es module导出
import minux from "./b";////需要使用es module导出

npx webpack index.js //打包命令 使用webpack处理index.js这个文件
```

总结：webpack 是一个模块打包工具，可以识别出引入模块的语法，早起的webpack只是个js模块的打包工具，现在可以是css，png，vue的模块打包工具



Terminal output of the command `npx webpack` on a Mac. Red arrows point from Chinese annotations to specific parts of the output:

- Annotation: "本次打包的唯一-hash值" (Unique hash value for this build) points to `Hash: 5c59f59aba806a95d22f`.
- Annotation: "webpack版本" (Webpack version) points to `Version: webpack 4.30.0`.
- Annotation: "本次打包的耗时" (Time taken for this build) points to `Time: 143ms`.

The output also shows the built assets:

Asset	Size	Chunks	Chunk Names
<code>bundle.js</code>	1.07 KiB	0	<code>[emitted] main</code>

Entrypoint `main` = `bundle.js`

Files included in the bundle:

- `[0] ./index.js` 179 bytes `{0}` `[built]`
- `[1] ./a.js` 84 bytes `{0}` `[built]`
- `[2] ./b.js` 91 bytes `{0}` `[built]`

A warning message is displayed at the bottom:

```
WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value. Set 'mode' option to 'development' or 'production' to enable defaults for each environment.
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack.js.org/concepts/mode/
```

安装

- 环境：nodejs <https://nodejs.org/en/>
版本参考官网发布的最新版本，可以提升webpack的打包速度
- 全局 不推荐

```
npm install webpack webpack-cli -g//webpack-cli 可以帮助我们在命令行里使用npx ,webpack等相关指令
```

```
webpack -v
```

```
npm uninstall webpack webpack-cli -g
```

- 局部安装 项目内安装

```
npm install webpack webpack-cli --save-dev //-D
```

```
webpack -v //command not found 默认在全局环境中查找
```

```
npx webpack -v// npx帮助我们在项目中的node_modules里查找webpack
```

- 安装指定版本

```
npm info webpack//查看webpack的历史发布信息
```

```
npm install webpack@x.xx webpack-cli -D
```

webpack 配置文件

当我们使用`npx webpack index.js`时，表示的是使用webpack处理打包，名为`index.js`的入口模块。默认放在当前目录下的`dist`目录，打包后的模块名称是`main.js`，当然我们也可以修改

webpack有默认的配置文件的，叫`webpack.config.js`，我们可以对这个文件进行修改，进行个性化配置

- 默认的配置文件的： `webpack.config.js`

```
npx webpack //执行命令后，webpack会找到默认的配置文件的，并使用执行
```

- 不使用默认的配置文件的： `webpackconfig.js`

```
npx webpack --config webpackconfig.js //指定webpack使用webpackconfig.js文件来作为配置文件的并执行
```

- 修改package.json scripts字段：有过vue react开发经验的同学 习惯使用npm run来启动，我们也可以修改下

```
"scripts":{  
  "bundle":"webpack"//这个地方不要添加npx ,因为npm run执行的命令，会优先使用项目工程里的包，效果和npx非常类似  
}  
  
npm run bundle
```

项目结构优化

```
dist  
  //打包后的资源目录  
node_modules  
  //第三方模块  
src  
  //源代码  
  css  
  images  
  index.js  
  
package.json  
webpack.config.js
```

Webpack 的核心概念

entry:

指定打包入口文件

```
entry:{
  main: './src/index.js'
}
=====
entry:"./src/index.js"

entry:
```

output:

打包后的文件位置

```
output: {
  publicPath:"xxx",
  filename: "bundle.js",
  // 必须是绝对路径
  path: path.resolve(__dirname, "dist")
},
```

loader

webpack是模块打包工具，而模块不仅仅是js，还可以是css，图片或者其他格式

但是webpack默认只知道如何处理js模块，那么其他格式的模块处理，和处理方式就需要loader了

```
module:{
  rules:[
    {
      test:/\.xxx$/,
      use:{
        loader: 'xxx-load'
      }
    }
  ]
}
```

当webpack处理到不认识的模块时，需要在webpack中的module处进行配置，当检测到是什么格式的模块，使用什么loader来处理。

- loader: file-loader: 处理静态资源模块

loader: file-loader

原理是把打包入口中识别出的资源模块，移动到输出目录，并且返回一个地址名称

所以我们什么时候用file-loader呢？

场景：就是当我们需要模块，仅仅是从源代码挪移到打包目录，就可以使用file-loader来处理，txt, svg, csv, excel, 图片资源啦等等

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

案例：

```
module: {
  rules: [
    {
      test: /\.?(png|jpe?g|gif)$/ ,
      //use使用一个loader可以用对象，字符串，两个loader需要用数组
      use: {
        loader: "file-loader",
        // options额外的配置，比如资源名称
        options: {
          // placeholder 占位符 [name]老资源模块的名称
          // [ext]老资源模块的后缀
          // https://webpack.js.org/loaders/file-loader#placeholders
          name: "[name].[hash].[ext]",
          //打包后的存放位置
          outputPath: "images/"
        }
      }
    }
  ]
},
```

- url-loader

可以处理file-loader所有的事情，但是遇到jpg格式的模块，会把该图片转换成base64格式字符串，并打包到js里。对小体积的图片比较合适，大图片不合适。

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

案例；

```
module: {
  rules: [
    {
      test: /\. (png|jpe?g|gif)$/ ,
      use: {
        loader: "url-loader",
        options: {
          name: "[name]_[hash].[ext]",
          outputPath: "images/",
          //小于2048, 才转换成base64
          limit: 2048
        }
      }
    }
  ]
},
```

样式处理：

Css-loader 分析css模块之间的关系，并合成一个css

Style-loader 会把css-loader生成的内容，以style挂载到页面的heade部分

```
npm install style-loader css-loader -D
```

```
{
  test: /\.css$/,
  use: ["style-loader", "css-loader"]
}
```

sass样式处理

sass-load 把sass语法转换成css，依赖node-sass模块

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

案例：

loader有顺序，从右到左，从下到上

```
{
  test: /\.scss$/,
  use: ["style-loader", "css-loader", "sass-loader"]
}
```

样式自动添加前缀：

Postcss-loader

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

webpack.config.js

```
{
  test: /\.css$/,
  use: ["style-loader", "css-loader", "postcss-loader"]
},
```

新建postcss.config.js

安装autoprefixer

```
//npm i autoprefixer -D

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

Plugins

plugin 可以在webpack运行到某个阶段的时候，帮你做一些事情，类似于生命周期的概念

HtmlWebpackPlugin

htmlwebpackplugin会在打包结束后，自动生成一个html文件，并把打包生成的js模块引入到该html中。

```
npm install --save-dev html-webpack-plugin
```

clean-webpack-plugin

```
npm install --save-dev clean-webpack-plugin
```

sourceMap

源代码与打包后的代码的映射关系

在dev模式中，默认开启，关闭的话 可以在配置文件里

```
devtool: "none"
```

devtool的介绍：<https://webpack.js.org/configuration/devtool#devtool>

eval:速度最快

cheap:较快，不用管列的报错

Module：第三方模块

开发环境推荐：

```
devtool: "cheap-module-eval-source-map"
```

线上环境可以不开启：如果要看到一些错误信息，推荐；

```
devtool: "cheap-module-source-map"
```

WebpackDevServer

提升开发效率的利器

每次改完代码都需要重新打包一次，打开浏览器，刷新一次，很麻烦

我们可以安装使用webpackdevserver来改善这块的体验

启动服务后，会发现dist目录没有了，这是因为devServer把打包后的模块不会放在dist目录下，而是放到内存中，从而提升速度

```
npm install webpack-dev-server -D
```

修改下package.json

```
"scripts": {  
  "server": "webpack-dev-server"  
},
```

在webpack.config.js配置：

```
devServer: {  
  contentBase: "./dist",  
  open: true,  
  port: 8081  
},
```

跨域：

联调期间，前后端分离，直接获取数据会跨域，上线后我们使用nginx转发，开发期间，webpack就可以搞定这件事

启动一个服务器，mock一个接口：

```
// npm i express -D  
// 创建一个server.js 修改scripts "server":"node server.js"  
  
//server.js  
const express = require('express')  
  
const app = express()  
  
app.get('/api/info', (req,res)=>{  
  res.json({  
    name:'开课吧',  
    age:5,  
    msg:'欢迎来到开课吧学习前端高级课程'  
  })  
})
```

```
})  
  
app.listen('9092')
```

项目中安装axios工具

```
//npm i axios -D  
  
//index.js  
import axios from 'axios'  
axios.get('http://localhost:9092/api/info').then(res=>{  
  console.log(res)  
})
```

会有跨域问题

修改webpack.config.js 设置服务器代理

```
proxy: {  
  "/api": {  
    target: "http://localhost:9092"  
  }  
}
```

修改index.js

```
axios.get("/api/info").then(res => {  
  console.log(res);  
});
```

搞定!

Hot Module Replacement (HMR:热模块替换)

启动hmr

```
devServer: {
  contentBase: "./dist",
  open: true,
  hot: true,
  //即便HMR不生效，浏览器也不自动刷新，就开启hotOnly
  hotOnly: true
},
```

配置文件头部引入webpack

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

const webpack = require("webpack");
```

在插件配置处添加：

```
plugins: [
  new CleanWebpackPlugin(),
  new HtmlWebpackPlugin({
    template: "src/index.html"
  }),
  new webpack.HotModuleReplacementPlugin()
],
```

案例：

```
//index.js
import "../css/index.css";

var btn = document.createElement("button");
btn.innerHTML = "新增";
document.body.appendChild(btn);

btn.onclick = function() {
  var div = document.createElement("div");
  console.log("1");
  div.innerHTML = "item";
  document.body.appendChild(div);
};

//index.css
```

```
div:nth-of-type(odd) {  
  background: yellow;  
}
```

处理js模块HMR

需要使用module.hot.accept来观察模块更新 从而更新

案例：

```
//counter.js  
function counter() {  
  var div = document.createElement("div");  
  div.setAttribute("id", "counter");  
  div.innerHTML = 1;  
  div.onclick = function() {  
    div.innerHTML = parseInt(div.innerHTML, 10) + 1;  
  };  
  document.body.appendChild(div);  
}  
export default counter;  
  
//number.js  
function number() {  
  var div = document.createElement("div");  
  div.setAttribute("id", "number");  
  div.innerHTML = 13000;  
  document.body.appendChild(div);  
}  
export default number;  
  
//index.js  
  
import counter from "./counter";  
import number from "./number";  
  
counter();  
number();  
  
if (module.hot) {
```

```
module.hot.accept("./b", function() {
  document.body.removeChild(document.getElementById("number"));
  number();
});
}
```

Babel处理ES6

官方网站: <https://babeljs.io/>

```
npm i babel-loader @babel/core @babel/preset-env -D
```

//babel-loader是webpack 与 babel的通信桥梁, 不会做把es6转成es5的工作, 这部分工作需要用到@babel/preset-env来做

//@babel/preset-env里包含了es6转es5的转换规则

```
//index.js
const arr = [new Promise(() => {}), new Promise(() => {})];

arr.map(item => {
  console.log(item);
});
```

通过上面的几步 还不够, Promise等一些还有转换过来, 这时候需要借助@babel/polyfill, 把es的新特性都装进来, 来弥补低版本浏览器中缺失的特性

@babel/polyfill

```
npm install --save @babel/polyfill
```

```
{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: "babel-loader",
  options: {
    presets: ["@babel/preset-env"]
  }
}
```

```
//index.js 顶部
import "@babel/polyfill";
```

会发现打包的体积大了很多，这是因为polyfill默认会把所有特性注入进来，假如我想我用到的es6+，才会注入，没用到的不注入，从而减少打包的体积，可不可以呢

当然可以

修改Webpack.config.js

```
options: {
  presets: [
    [
      "@babel/preset-env",
      {
        targets: {
          edge: "17",
          firefox: "60",
          chrome: "67",
          safari: "11.1"
        },
        useBuiltIns: "usage" //按需注入
      }
    ]
  ]
}
```

当我们开发的是组件库，工具库这些场景的时候，polyfill就不适合了，因为polyfill是注入到全局变量，window下的，会污染全局环境，所以推荐闭包方式：@babel/plugin-transform-runtime

@babel/plugin-transform-runtime

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

```
npm install --save @babel/runtime
```

怎么使用？

先注释掉index.js里的polyfill

```
//import "@babel/polyfill";

const arr = [new Promise(() => {}), new Promise(() => {})];

arr.map(item => {
  console.log(item);
});
```

修改配置文件：注释掉之前的presets，添加plugins

```
options: {
  //  presets: [
  //    [
  //      "@babel/preset-env",
  //      {
  //        targets: {
  //          edge: "17",
  //          firefox: "60",
  //          chrome: "67",
  //          safari: "11.1"
  //        },
  //        useBuiltIns: "usage"
  //      }
  //    ]
  //  ]
  "plugins": [
    [
      "@babel/plugin-transform-runtime",
      {
        "absoluteRuntime": false,
        "corejs": 2,

```



```

    "helpers": true,
    "regenerator": true,
    "useESModules": false
  }
]
}

```

扩展：

babelrc文件：

新建.babelrc文件，把options部分移入到该文件中，就可以了

```

//.babelrc
{
  "plugins": [
    [
      "@babel/plugin-transform-runtime",
      {
        "absoluteRuntime": false,
        "corejs": 2,
        "helpers": true,
        "regenerator": true,
        "useESModules": false
      }
    ]
  ]
}

```

```

//webpack.config.js

```

```

{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: "babel-loader"
}

```

配置React打包环境

安装

```
npm install react react-dom --save
```

编写react代码:

```
//index.js
import "@babel/polyfill";

import React, { Component } from "react";
import ReactDOM from "react-dom";

class App extends Component {
  render() {
    return <div>hello world</div>;
  }
}

ReactDOM.render(<App />, document.getElementById("app"));
```

安装babel与react转换的插件:

```
npm install --save-dev @babel/preset-react
```

在babelrc文件里添加:

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "edge": "17",
          "firefox": "60",
          "chrome": "67",
          "safari": "11.1"
        },
        "useBuiltIns": "usage" //按需注入
      }
    ],
    "@babel/preset-react"
  ]
}
```

