

Webpack-Day3



sourceMap

- 1.为了开发时，快速的定位问题
- 2.线上代码，我们有时候也会开启 前端错误监控，快速的定位问题

源代码与打包后的代码的映射关系，通过sourceMap定位到源代码。

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,使用eval包裹模块代码,

source-map: 产生 .map 文件 外部产生 错误代码的准确信息和位置

cheap:较快, 不包含列信息

Module: 第三方模块, 包含loader的sourcemap (比如jsx to js, babel的sourcemap)

inline: 将 .map 作为DataURI嵌入, 不单独生成 .map 文件

配置推荐:

```
devtool: "cheap-module-eval-source-map", // 开发环境配置
```

```
//线上不推荐开启
```

```
devtool: "cheap-module-source-map", // 线上生成配置
```

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

- 安装

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

- 配置

修改下package.json

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

在webpack.config.js配置:

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

- 启动

```
npm run server
```

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

- 本地mock,解决跨域:

前后端分离

前端和后端是可以并行开发的,

前端会依赖后端的接口

先给接口文档，和接口联调日期的

我们前端就可以本地mock数据，不打断自己的开发节奏

开发模式：前后端分离

项目会议

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')

//node server.js

http://localhost:9092/api/info
```

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);
});
```

Hot Module Replacement (HMR:热模块替换)

- 不支持抽离出的css 我们要使用css-loader
-

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");
  div.innerHTML = "item";
  document.body.appendChild(div);
};

//index.css
div:nth-of-type(odd) {
  background: yellow;
}

```

注意启动HMR后，css抽离会不生效，还有不支持contenthash， chunkhash

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/>

中文网站: <https://www.babeljs.cn/>

Babel是JavaScript编译器, 能将ES6代码转换成ES5代码, 让我们开发过程中放心使用JS新特性而不用担心兼容性问题。并且还可以通过插件机制根据需求灵活的扩展。

Babel在执行编译的过程中，会从项目根目录下的 `.babelrc` JSON文件中读取配置。没有该文件会从 loader的options地方读取配置。

测试代码

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

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

安装

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

1.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

2.@babel/preset-env里包含了es, 6, 7, 8转es5的转换规则

Ecma 5 6 7 8... 草案（评审通过的，还有未通过的）

面向未来的

env是babel7之后推行的预设插件

```
env{
```

```
  ecma 5
```

```
  ecma 6
```

```
  ecma 7
```

```
  ecma 8
```

```
  ...
```

```
}
```

Webpack.config.js

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

通过上面的几步 还不够，默认的Babel只支持let等一些基础的特性转换，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

以全局变量的方式注入进来的。window.Promise，它会造成全局对象的污染

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

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

按需加载，减少冗余

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"
      },
      corejs: 2, //新版本需要指定核心库版本
      useBuiltIns: "entry" //按需注入
    }
  ]
}
}

```

`useBuiltIns` 选项是 `babel 7` 的新功能，这个选项告诉 `babel` 如何配置 `@babel/polyfill`。它有三个参数可以使用：①`entry`: 需要在 `webpack` 的入口文件里 `import "@babel/polyfill"` 一次。`babel` 会根据你的使用情况导入垫片，没有使用的功能不会被导入相应的垫片。②`usage`: 不需要 `import`，全自动检测，但是要安装 `@babel/polyfill`。（试验阶段）③`false`: 如果你 `import "@babel/polyfill"`，它不会排除掉没有使用的垫片，程序体积会庞大。（不推荐）

请注意： `usage` 的行为类似 `babel-transform-runtime`，不会造成全局污染，因此也不会不会对类似 `Array.prototype.includes()` 进行 `polyfill`。

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

{
  presets: [
    [
      "@babel/preset-env",
      {
        targets: {
          edge: "17",
          firefox: "60",
          chrome: "67",
          safari: "11.1"
        },
        corejs: 2, //新版本需要指定核心库版本
        useBuiltIns: "usage" //按需注入
      }
    ]
  ]
}

```

```
//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 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",
    "Android": "6.0"
  },
  "useBuiltIns": "usage", //按需注入
}
],
"@babel/preset-react"
]
}

```

如果是库的作者的话，提供模块的时候代码怎么打包的？

构建速度会越来越慢，怎么优化

扩展：

多页面打包通用方案

```

entry:{
  index:"./src/index",
  list:"./src/list",
  detail:"./src/detail"
}

new htmlWebpackPlugin({
  title: "index.html",
  template: path.join(__dirname, "./src/index/index.html"),
  filename: "index.html",
  chunks:[index]
})

```

1.目录结构调整

- src
 - index
 - index.js
 - index.html
 - list
 - index.js

- index.html
- detail
 - index.js
 - index.html
- 2.使用 glob.sync 第三方库来匹配路径

```
npm i glob -D
```

```
const glob = require("glob")
```

//MPA多页面打包通用方案

```
const setMPA = () => {
  const entry = {};
  const htmlWebpackPlugins = [];

  return {
    entry,
    htmlWebpackPlugins
  };
};

const { entry, htmlWebpackPlugins } = setMPA();
```

```
const setMPA = () => {
  const entry = {};
  const htmlWebpackPlugins = [];

  const entryFiles = glob.sync(path.join(__dirname, "./src/*/index.js"));

  entryFiles.map((item, index) => {
    const entryFile = entryFiles[index];
    const match = entryFile.match(/src\/(.*)\/index\.js$/);
    const pageName = match && match[1];
    entry[pageName] = entryFile;
    htmlWebpackPlugins.push(
      new htmlWebpackPlugin({
        title: pageName,
        template: path.join(__dirname, `src/${pageName}/index.html`),
        filename: `${pageName}.html`,
      })
    );
  });

  return {
    entry,
    htmlWebpackPlugins
  };
};
```

```
        chunks: [pageName],
        inject: true
      })
    );
  });
  return {
    entry,
    htmlWebpackPlugin
  };
};

const { entry, htmlWebpackPlugin } = setMPA();

module.exports = {
  entry,
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "[name].js"
  },
  plugins: [
    // ...
    ...htmlWebpackPlugin //展开数组
  ]
}
```