

Webpack-Day2

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/>

中文网站: <https://www.babeljs.cn/>

```
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

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
```

Webpack.config.js

```
{
  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

它不会造成全局污染

```
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",
        corejs: 2
      }
    ]
  ],
  "plugins": [
    [
      "@babel/plugin-transform-runtime",
      {
```

```

    "absoluteRuntime": false,
    "corejs": 2,
    "helpers": true,
    "regenerator": true,
    "useESModules": false
  }
]
}

```

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

```

tree Shaking

webpack2.x开始支持 tree shaking概念，顾名思义，"摇树"，只支持ES module的引入方式！！！！，

```

//webpack.config.js

optimization: {
  usedExports: true
}

//package.json
"sideEffects":false 正常对所有模块进行tree shaking 或者 "sideEffects":
['*.css', '@babel/polyfill']

```

开发模式设置后，不会帮助我们删掉没有引用的代码

案例：

```

//expo.js
export const add = (a, b) => {
  console.log(a + b);
};

export const minus = (a, b) => {
  console.log(a - b);
};

```



```
//index.js
import { add } from "./expo";
add(1, 2);
```

development vs Production模式区分打包

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

案例

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

module.exports = merge(commonConfig,devConfig)

//package.js
"scripts":{
  "dev":"webpack-dev-server --config ./build/webpack.dev.js",
  "build":"webpack --config ./build/webpack.prod.js"
}
```

案例2

基于环境变量

```
//外部传入的全局变量
module.exports = (env)=>{
  if(env && env.production){
    return merge(commonConfig,prodConfig)
  }else{
    return merge(commonConfig,devConfig)
  }
}

//外部传入变量
scripts:" --env.production"
```

代码分割 code Splitting

```
import _ from "lodash";

console.log(_.join(['a','b','c','****']))
```

假如我们引入一个第三方的工具库，体积为1mb，而我们的业务逻辑代码也有1mb，那么打包出来的体积大小会在2mb

导致问题：

体积大，加载时间长

业务逻辑会变化，第三方工具库不会，所以业务逻辑一变更，第三方工具库也要跟着变。

引入代码分割的概念：

```
//lodash.js

import _ from "lodash";

window._ = _;

//index.js 注释掉lodash引用
//import _ from "lodash";

console.log(_.join(['a','b','c','****']))
```

```
//webpack.config.js
entry: {
  lodash: "./lodash.js",
  index: "./index.js"
},
//指定打包后的资源位置
output: {
  path: path.resolve(__dirname, "./build"),
  filename: "[name].js"
}
```

其实code Splitting概念 与 webpack并没有直接的关系，只不过webpack中提供了一种更加方便的方法供我们实现代码分割

基于<https://webpack.js.org/plugins/split-chunks-plugin/>

```
optimization: {
  splitChunks: {
    chunks: 'async', //对同步，异步，所有的模块有效
    minSize: 30000, //当模块大于30kb
    maxSize: 0, //对模块进行二次分割时使用，不推荐使用
    minChunks: 1, //打包生成的chunk文件最少有几个chunk引用了这个模块
    maxAsyncRequests: 5, //模块请求5次
    maxInitialRequests: 3, //入口文件同步请求3次
    automaticNameDelimiter: '~',
    name: true,
    cacheGroups: {
      vendors: {
        test: /[\\/]node_modules[\\/]/,
        priority: -10 //优先级 数字越大，优先级越高
      },
      default: {
        minChunks: 2,
        priority: -20,
        reuseExistingChunk: true
      }
    }
  }
}
```

使用下面配置即可：

```
optimization:{  
  //帮我们自动做代码分割  
  splitChunks:{  
    chunks:"all",//默认是支持异步，我们使用all  
  }  
}
```

打包分析

<https://github.com/webpack/analyze>

官方推荐工具

<https://webpack.js.org/guides/code-splitting/#bundle-analysis>

webpack 官方推荐的编码方式

```
optimization:{  
  //帮我们自动做代码分割  
  splitChunks:{  
    chunks:"async",//默认是支持异步  
  }  
}
```

代码利用率的问题

```
//index.js

document.addEventListener("click", () => {
  const element = document.createElement("div");
  element.innerHTML = "welcome to webpack4.x";
  document.body.appendChild(element);
});
```

通过控制台看看代码利用率

把里面异步代码抽离出来

```
//index.js

document.addEventListener("click", () => {
  import("./click.js").then(({ default: func }) => {
    //需要用到 npm install --save-dev @babel/plugin-syntax-dynamic-import

    func();
  });
});

//click.js
function handleClick() {
  const element = document.createElement("div");
  element.innerHTML = "welcome to webpack4.x";
  document.body.appendChild(element);
}

export default handleClick;
```