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"]
    }
  }
}


  "browserslist": [
    "last 2 version",
    "> 1%",
    "not ie < 11",
    "cover 99.5%",
    "dead"
  ]
```

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。windows.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： ①entry: 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。 ②usage: 不需要`import`，全自动检测，但是要安装`@babel/polyfill`。（试验阶段） ③false: 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"
}
```

暗号：做人嘛，最重要的是开心

作业：实现text-webpack-plugin 自定义插件

要求：提交代码截图 ，在emit阶段，往资源列表里插入一个新的txt文档，文档的内容和体积不限

配置React打包环境

安装

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

解析插件

转换插件

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

webpack性能优化

- 优化开发体验
- 优化输出质量

优化开发体验

- 提升效率
- 优化构建速度
- 优化使用体验

优化输出质量

- 优化要发布到线上的代码，减少用户能感知到的加载时间
- 提升代码性能，性能好，执行就快

缩小搜索Loader的文件范围

优化loader配置

- test include exclude三个配置项来缩小loader的处理范围
- 推荐include

```
//string
include: path.resolve(__dirname, "./src"),


//array
include: [
          path.resolve(__dirname, 'app/styles'),
          path.resolve(__dirname, 'vendor/styles')
        ]
```

优化resolve.modules配置

resolve.modules用于配置webpack去哪些目录下寻找第三方模块，默认是['node_modules']

寻找第三方模块，默认是在当前项目目录下的node_modules里面去找，如果没有找到，就会去上一级目录../node_modules找，再没有会去../../node_modules中找，以此类推，和Node.js的模块寻找机制很类似。

如果我们的第三方模块都安装在了项目根目录下，就可以直接指明这个路径。

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

优化resolve.alias配置

resolve.alias配置通过别名来将原导入路径映射成一个新的导入路径

拿react为例，我们引入的react库，一般存在两套代码

- cjs
  - 采用commonJS规范的模块化代码
- umd
  - 已经打包好的完整代码，没有采用模块化，可以直接执行

默认情况下，webpack会从入口文件./node_modules/bin/react/index开始递归解析和处理依赖的文件。我们可以直接指定文件，避免这处的耗时。

```
alias: {
  "@": path.join(__dirname, "./pages"),
  react: path.resolve(
  __dirname,
  "./node_modules/react/umd/react.production.min.js"
  ),
  "react-dom": path.resolve(
  __dirname,
  "./node_modules/react-dom/umd/react-dom.production.min.js"
  )
}
```

```
resolve: {
    alias: {
      "@assets": path.resolve(__dirname, "../src/images/"),
    },
  },

  //html-css中使用
  .sprite3 {
  background: url("~@assets/s3.png");
}
```

优化resolve.extensions配置

resolve.extensions在导入语句没带文件后缀时，webpack会自动带上后缀后，去尝试查找文件是否存在。

默认值：

```
extensions:['.js','.json','.jsx','.ts']
```

- 后缀尝试列表尽量的小
- `导入语句尽量的带上后缀。`

使用externals优化cdn静态资源

//公司有cdn

//静态资源有部署到cdn 有链接了

// 我想使用cdn！！！！！！！！

我的bundle文件里，就不用打包进去这个依赖了，体积会小

我们可以将一些JS文件存储在 `CDN` 上(减少 `Webpack` 打包出来的 `js` 体积)，在 `index.html` 中通过 标签引入，如:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <div id="root">root</div>
    <script src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js"></script>
</body>
</html>
```

我们希望在使用时，仍然可以通过 `import` 的方式去引用(如 `import $ from 'jquery'`)，并且希望 `webpack` 不会对其进行打包，此时就可以配置 `externals`。

```
//webpack.config.js
module.exports = {
    //...
    externals: {
        //jquery通过script引入之后，全局中即有了 jQuery 变量
        'jquery': 'jQuery'
    }
}
```

使用静态资源路径publicPath(CDN)

CDN通过将资源部署到世界各地，使得用户可以就近访问资源，加快访问速度。要接入CDN，需要把网页的静态资源上传到CDN服务上，在访问这些资源时，使用CDN服务提供的URL。

```
##webpack.config.js
output:{
 publicPath: '//cdnURL.com', //指定存放JS文件的CDN地址
}
```

- 咱们公司得有cdn服务器地址
- 确保静态资源文件的上传与否

css文件的处理

- 使用less或者sass当做css技术栈

```
$ npm install less less-loader --save-dev
```

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

- 使用postcss为样式自动补齐浏览器前缀
  - https://caniuse.com/

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

```
##新建postcss.config.js
module.exports = {
  plugins: [
    require("autoprefixer")({
      overrideBrowserslist: ["last 2 versions", ">1%"]
    })
  ]
};

##index.less
body {
  div {
    display: flex;
    border: 1px red solid;
  }
}

##webpack.config.js
{
  test: /\.less$/,
  include: path.resolve(__dirname, "./src"),
  use: [
    "style-loader",
    {
      loader: "css-loader",
      options: {}
    },
    "less-loader",
    "postcss-loader"
  ]
},
```

如果不做抽取配置，我们的`css`是直接打包进`js`里面的，我们希望能单独生成`css`文件。 因为单独生成css,css可以和js并行下载，提高页面加载效率

借助MiniCssExtractPlugin 完成抽离css

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

const MiniCssExtractPlugin = require("mini-css-extract-plugin");

 {
   test: /\.scss$/,
     use: [
       // "style-loader", // 不再需要style-loader，用MiniCssExtractPlugin.loader
代替
       MiniCssExtractPlugin.loader,
       //{
            loader: MiniCssExtractPlugin.loader,
            options: {
              publicPath: "../",
            },
          },//
       "css-loader", // 编译css
       "postcss-loader",
       "sass-loader" // 编译scss
     ]
 },

 plugins: [
    new MiniCssExtractPlugin({
      filename: "css/[name]_[contenthash:6].css",
      chunkFilename: "[id].css"
    })
  ]
```

压缩css

- 借助 optimize-css-assets-webpack-plugin
- 借助cssnano

```
##安装

npm install cssnano -D
npm i optimize-css-assets-webpack-plugin -D

const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");

new OptimizeCSSAssetsPlugin({
  cssProcessor: require("cssnano"), //引入cssnano配置压缩选项
  cssProcessorOptions: {
    discardComments: { removeAll: true }
  }
})
```

压缩HTML

- 借助html-webpack-plugin

```
new htmlWebpackPlugin({
      title: "京东商城",
      template: "./index.html",
      filename: "index.html",
      minify: {
        // 压缩HTML文件
        removeComments: true, // 移除HTML中的注释
        collapseWhitespace: true, // 删除空白符与换行符
        minifyCSS: true // 压缩内联css
      }
    }),
```

Hash ChunkHash ContentHash的区别

图片优化

在 Webpack 中可以借助img-webpack-loader来对使用到的图片进行优化。它支持 JPG、PNG、GIF 和 SVG 格式的图片，因此我们在碰到所有这些类型的图片都会使用它。

```
npm install image-webpack-loader --save-dev


macos系统版本需要依赖libpng,需要翻墙！！！！！
```

使用

```
rules: [{
  test: /\.(gif|png|jpe?g|svg)$/i,
  use: [
    'file-loader',
    {
      loader: 'image-webpack-loader',
      options: {
        mozjpeg: {
          progressive: true,
          quality: 65
        },
        // optipng.enabled: false will disable optipng
        optipng: {
          enabled: false,
        },
        pngquant: {
          quality: [0.65, 0.90],
          speed: 4
        },
        gifsicle: {
          interlaced: false,
        },
        // the webp option will enable WEBP
        webp: {
          quality: 75
        }
      }
    },
  ],
}]
```

github地址：https://github.com/tcoopman/image-webpack-loader

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"
}
```

基于环境变量区分

- 借助cross-env

```
npm i cross-env -D
```

package里面配置命令脚本，传入参数

```
##package.json

"test": "cross-env NODE_ENV=test webpack --config ./webpack.config.test.js",
```

在webpack.config.js里拿到参数

```
process.env.NODE_ENV
```

```
//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"
```

tree Shaking

Rollup

webpack2.x开始支持 tree shaking概念，顾名思义，"摇树"，清除无用 css,js(Dead Code)

Dead Code 一般具有以下几个特征

- 代码不会被执行，不可到达
- 代码执行的结果不会被用到
- 代码只会影响死变量（只写不读）
- Js tree shaking只支持ES module的引入方式！！！！，

Css tree shaking

```
npm i glob-all purify-css purifycss-webpack --save-dev

const PurifyCSS = require('purifycss-webpack')
const glob = require('glob-all')
plugins:[
    // 清除无用 css
    new PurifyCSS({
      paths: glob.sync([
        // 要做 CSS Tree Shaking 的路径文件
        path.resolve(__dirname, './src/*.html'), // 请注意，我们同样需要对 html 文件进行 tree shaking
        path.resolve(__dirname, './src/*.js')
      ])
    })
]
```

JS tree shaking

只支持import方式引入，不支持commonjs的方式引入

案例：

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

export const minus = (a, b) => {
  return a - b;
};


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

```
//webpack.config.js
optimization: {
   usedExports: true // 哪些导出的模块被使用了，再做打包
}
```

只要mode是production就会生效，develpoment的tree shaking是不生效的，因为webpack为了方便你的调试

可以查看打包后的代码注释以辨别是否生效。

生产模式不需要配置，默认开启

sideEffects 处理副作用

```
//package.json
"sideEffects":false  //正常对所有模块进行tree shaking  , 仅生产模式有效，需要配合usedExports

或者 在数组里面排除不需要tree shaking的模块

"sideEffects":['*.css','@babel/polyfill']
```

代码分割 code Splitting

单页面应用spa：

打包完后，所有页面只生成了一个bundle.js

- 代码体积变大，不利于下载
- 没有合理利用浏览器资源

多页面应用mpa:

如果多个页面引入了一些公共模块，那么可以把这些公共的模块抽离出来，单独打包。公共代码只需要下载一次就缓存起来了，避免了重复下载。

main.js 170kb

Lodash.js 150kb

Main.js 20kb

```
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。
```

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: "all", // 所有的 chunks 代码公共的部分分离出来成为一个单独的文件
    },
  },
```

```
optimization: {
    splitChunks: {
      chunks: 'async',//对同步 initial，异步 async，所有的模块有效 all
      minSize: 30000,//最小尺寸，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,//最大异步请求数，默认5
      maxInitialRequests: 3,//最大初始化请求书，入口文件同步请求，默认3
      automaticNameDelimiter: '-',//打包分割符号
      name: true,//打包后的名称，除了布尔值，还可以接收一个函数function
      cacheGroups: {//缓存组
        vendors: {
          test: /[\\/]node_modules[\\/]/,
          name:"vendor", // 要缓存的 分隔出来的 chunk 名称
          priority: -10//缓存组优先级 数字越大，优先级越高
        },
        other:{
          chunks: "initial", // 必须三选一： "initial" | "all" | "async"(默认就是
async)
          test: /react|lodash/, // 正则规则验证，如果符合就提取 chunk,
          name:"other",
          minSize: 30000,
          minChunks: 1,
        },
        default: {
          minChunks: 2,
          priority: -20,
          reuseExistingChunk: true//可设置是否重用该chunk
        }
      }
    }
  }
```

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
  }
}
```

Scope Hoisting

作用域提升（Scope Hoisting）是指 webpack 通过 ES6 语法的静态分析，分析出模块之间的依赖关系，尽可能地把模块放到同一个函数中。下面通过代码示例来理解：

-

```
// hello.js
export default 'Hello, Webpack';
// index.js
import str from './hello.js';
console.log(str);
```

打包后，`hello.js`的内容和`index.js`会分开

通过配置 optimization.concatenateModules=true`：开启 Scope Hoisting

```
// webpack.config.js
module.exports = {
    optimization: {
        concatenateModules: true
    }
};
```

我们发现hello.js内容和index.js的内容合并在一起了！所以通过 Scope Hoisting 的功能可以让 Webpack 打包出来的代码文件更小、运行的更快。

HardSourceWebpackPlugin

- 提供中间缓存的作用
- 首次构建没有太大的变化
- 第二次构建时间就会有较大的节省

```
const HardSourceWebpackPlugin = require('hard-source-webpack-plugin')

const plugins = [
  new HardSourceWebpackPlugin()
]
```

使用happypack并发执行任务

构建时间较久

项目复杂度较高

运行在 Node.之上的Webpack是单线程模型的，也就是说Webpack需要一个一个地处理任务，不能同时处理多个任务。 **Happy Pack** 就能让Webpack做到这一点，它将任务分解给多个子进程去并发执行，子进程处理完后再将结果发送给主进程。从而发挥多核 CPU 电脑的威力。

```
npm i -D happypack
  var happyThreadPool = HappyPack.ThreadPool({ size: 5 });
  //const happyThreadPool = HappyPack.ThreadPool({ size: os.cpus().length })

// webpack.config.js
 rules: [
     {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        use: [
          {
            // 一个loader对应一个id
            loader: "happypack/loader?id=babel"
          }
        ]
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "./src"),
        use: ["happypack/loader?id=css"]
      },
  ]

//在plugins中增加
plugins:[
      new HappyPack({
```

```
        // 用唯一的标识符id，来代表当前的HappyPack是用来处理一类特定的文件
        id:'babel',
        // 如何处理.js文件，用法和Loader配置中一样
        loaders:['babel-loader?cacheDirectory'],
        threadPool: happyThreadPool,
    }),
    new HappyPack({
        id: "css",
        loaders: ["style-loader", "css-loader"]
      }),
]
```

https://github.com/webpack-contrib/mini-css-extract-plugin/issues/273

https://github.com/amireh/happypack/issues/242

如何自己编写一个Plugin

Plugin: 开始打包，在某个时刻，帮助我们处理一些什么事情的机制

plugin要比loader稍微复杂一些，在webpack的源码中，用plugin的机制还是占有非常大的场景，可以说plugin是webpack的灵魂

设计模式

事件驱动

发布订阅

plugin是一个类，里面包含一个apply函数，接受一个参数，compiler

官方文档：https://webpack.js.org/contribute/writing-a-plugin/

案例：

- 创建copyright-webpack-plugin.js

```
class CopyrightWebpackPlugin {
  constructor() {
  }

  //compiler：webpack实例
  apply(compiler) {

  }
}
module.exports = CopyrightWebpackPlugin;
```

- 配置文件里使用

```
const CopyrightWebpackPlugin = require("./plugin/copyright-webpack-plugin");

plugins: [new CopyrightWebpackPlugin()]
```

- 如何传递参数

```
//webpack配置文件
plugins: [
    new CopyrightWebpackPlugin({
      name: "开课吧"
    })
  ]

//copyright-webpack-plugin.js
class CopyrightWebpackPlugin {
  constructor(options) {
    //接受参数
    console.log(options);
  }

  apply(compiler) {}
}
module.exports = CopyrightWebpackPlugin;
```

- 配置plugin在什么时刻进行

```
class CopyrightWebpackPlugin {
  constructor(options) {
```

```
    // console.log(options);
  }

  apply(compiler) {
    //hooks.emit 定义在某个时刻
    compiler.hooks.emit.tapAsync(
      "CopyrightWebpackPlugin",
      (compilation, cb) => {
        compilation.assets["copyright.txt"] = {
          source: function() {
            return "hello copy";
          },
          size: function() {
            return 20;
          }
        };
        cb();
      }
    );

    //同步的写法
    //compiler.hooks.compile.tap("CopyrightWebpackPlugin", compilation => {
    //  console.log("开始了");
    //});
  }
}
module.exports = CopyrightWebpackPlugin;
```

参考：compiler-hooks

https://webpack.js.org/api/compiler-hooks