webpack打包中可以使用npx优先执行本地包

webpack中development（开发环境依赖）配置了的热重载

--只重新打包修改过的模块-在内存打包

--浏览器同步进行刷新修改区域

# //装饰器语法//支持generator//改写react中不适用constroutor

.修改babel配置，改写react组件  
  
  
可以不用constructor和静态绑定了  
  
1.安装babel插件  
    cnpm install @babelugin-proposal-class-properties -D  
2.修改.babelrc配置  
"plugins":[ ["@babel/plugin-proposal-decorators", {"legacy": true}], //装饰器语法  
[  
"@babel/plugin-transform-runtime", //支持generator  
{  
"absoluteRuntime": false,  
"corejs": 2,  
"helpers": true,  
"regenerator": true,  
"useESModules": false  
}  
],  
["@babel/plugin-proposal-class-properties", { "loose": true }//改写react中不使用constroutor  
]

# Wepack性能优化

给css样式自动加前缀

使用资源内联

html-webpack-externals-plugin插件 分离公共库

SplitChunksPlugin插件 -将多次引入的资源还有公共资源单独打包

@babel/plugin-syntax-dynamic-import动态加载插件，实现代码懒加载或import动态加载

安装react-loadable，实现异步加载组件

# 面试题：如何减少打包体积

1.使用html-webpack-extenrnals-plugin 插件，使用cdn服务器将公共库在cdn服务器拽取

2.使用webpack内置插件，SplitChunksPlugin 插件，分离公共库，和公共脚本，将公共脚本抽离

[**3.使用@babel/plugin-syntax-dynamic-import**](mailto:3.使用@babel/plugin-syntax-dynamic-import)**插件，实现动态加载插件，实现懒加载，当使用组件时通过import加载组件，后面跟一个.then成功回调函数，通过this。Setstate设置组件是否渲染**

**syntax 三 太 克 s dynamic带 内 米 科**

场景3:使用@babel/plugin-syntax-dynamic-import 动态加载插件，实现代码懒加载或import动态加载

代码懒加载，动态import

--将比较大组件或代码打包成独立文件，需要时候在从服务器加载

优点:打包文件体积变小 加载速度变快（页面加载的时候，并没有加载组件代码，而是使用的时候比如点击等才动态插入<script src="js/组件文件.js"></script>）从服务器加载对应脚本

cnpm install @babel/plugin-syntax-dynamic-import -D

2.修改.babelrc配置，添加

"plugins":[

…..,

…..,

"@babel/plugin-syntax-dynamic-import“

]

单击时动态引入组件，实现懒加载

，使用时需要使用默认导出的文件

handleClick(){

        import('./Child.jsx')

        .then(Child=>{

            this.setState({

                Child:Child.default

            })

        })

}

# 简书最后有一个异步加载组件的插件

React-loadable 读音：react-l od bo

## 1.安装Cnpm I -S react-loadable

## 2.创建Loadable组件

import React from 'react'

import Loadable from 'react-loadable';

const LoadableComponent = Loadable({

  loader: () => import('./index'),

  loading(){

      return <div>正在加载。。。。</div>

  }

});

export default ()=><LoadableComponent/>

## 3.修改路由，访问detailLoadable.js组件

## 4.引入高阶组件，包裹导出的组件，使其获得router的参数

import {withRouter} from 'react-router-dom'

export default connect(mapStateToProps,mapDispatchToProps)(withRouter(Detail))

*// withRouter有能力获取router里的所有参数和内容*

# 14.常用的loader

常用loader:

1. babel-loader 转换ES6、ES7、JSX等语法

2. css-loader 支持.css文件的加载和解析

css-loader :用于加载.css文件，并且转为commonjs模块对象，可以使用import进行导入,分析导入.css合并成一个

可以开启模块化，防止css样式重名

style-loader:将样式通过<style></style>标签插入到<head></head>之间

3. less-loader 支持less文件转换成css

4. ts-loader 将TS转换为js

5. file-loader 进行图片、字体等打包

6. thread-loader 多进程打包js和css

7. image-webpack-loader 将图片进行压缩

8. url-loader 拷贝图片，对于指定大小的图片进行base64转码

# 12.webpack中plugin的使用

使用plugin时候，引入后，new这个插件，然后在里面进行相应的配置

插件目的在于解决 [loader](https://www.webpackjs.com/concepts/loaders) 无法实现的其他事：

文件管理---拷贝模板到打包目录并且插入js文件--html-webpack-plugin

资源管理—清理打包目录---cleanWebpackPlugin

压缩js---UglifyjsWebpackPlugin

优化、混淆代码

13.常用的plugin

CleanWebpackPlugin 清理构建目录

CopyWebpackPlugin 将文件或文件夹拷贝到构建的输出目录

HtmlWebpackPlugin 拷贝模板文件到打包目录，并且插入生成的.js

//运行webpack命令时，在dist目录自动生成一个根据模板index.html生成一个index.html文件并且自动引入打包后的.js文件

CommonsChunkPlugin 将chunks相同的模块代码提取成公共js

ExtractTextWebpackPlugin 将CSS从bundle文件里提取成一个独立CSS文件

UglifyjsWebpackPlugin 压缩js

ZipWebpackPlugin 将打包资源生成一个zip包

mini-css-extract-plugin 将css提取成为独立文件.css,不打包到.js中

terser-webpack-plugin 压缩js文件

optimize-css-assets-webpack-plugin 压缩css文件

# webpack配置文件

webpack配置文件，默认名称 webpack.config.js

# 配置文件配置项:

module.exports={  
 entry:{},//入口配置项

output:{},//打包出口配置项

plugins:[],//插件配置项

module: {

rules: [ //loader配置项

{loader配置项},

{loader配置项}

]

},

Resolve：{extensions:[‘.jsx’,’.js’] //忽略后缀名

Dev-tool : source-map

mode:””/mode配置项，开发环境还是生产环境

devServer: {} //开发服务器配置项

}

entry配置项可以配置多入口

entry:{

index:"./src/index.js",

one:"./src/one"

},

# Babel-loader插件的功能

1. ES6/7/8转ES5

2.react jsx语法转js

# React环境构建:

安装babel核心

cnpm install --save-dev @babel/core

安装babel-loader转换器

cnpm install -D babel-loader //将es6转换成es5

安装es6语法插件集合

cnpm i -D @babel/preset-env //这两个插件决定是用什么语法转

安装react 语法插件 // Rjsx语法转换成js语法

cnpm i -D @babel/preset-react //这两个插件决定是用什么语法转

安装react react-dom库

cnpm i -S react react-dom //上线要用，安装在生产环境依赖下

## Babel-loader功能：

es678==》es5 react jsx语法转换成js语法

## babel-loader如何配置

module: {

rules: [

{

test: /\.m?js$/,

exclude: /(node\_modules|bower\_components)/,

use: {

loader: 'babel-loader',

options: {

presets: ['@babel/preset-env','@babel/preset-react'] 规则是这俩

}

## babelrc文件配置

{

"presets": [

'@babel/preset-env',

'@babel/preset-react'

]

}

# 安装装饰器语法支持插件

cnpm install --save-dev @babel/plugin-proposal-decorators

.babelrc中配置：

"plugins":[ ["@babel/plugin-proposal-decorators", {"legacy": true}]]

# 配置devtool，进行开发调式，找出对应位置的错误

devtool:"source-map",

# 11. 支持（generator，async，await）问题:ES6/7/8标准有一些新对象--这些对象在ES5中没有

解决:

方案一:@babel/polyfill

问题:polyfill体积太大

解决:按需加载:用哪个对象，就加载哪个对象

.babelrc //修改.babelrc使所有entry文件都支持polyfill

[

“@babel/preset-env”,

{

“useBuiltIns”: “usage” //entry🡪usage 按需加载polyfill太大

}

方案二:

cnpm i -S @babel/runtime //3的依赖项

cnpm i -S @babel/runtime-corejs2 //3的依赖项

cnpm I -D @babel/plugin-transform-runtime //代替polyfill

[

"@babel/plugin-transform-runtime",

{

"absoluteRuntime": false,

"corejs": 2,

"helpers": true,

"regenerator": true,

"useESModules": false

}

]

# 15.HtmlWebpackPlugin 拷贝模板文件到打包目录，并且插入生成的.js

<title><%= htmlWebpackPlugin.options.title %></title>

plugins:[

//配置多个应用

new HtmlWebpackPlugin({ //假设是前台应用入口

title: '首页',

  filename:"index.html",

template:“./public/index.html”, //模板拷贝的

chunks:[“index”] //chunks指定需要引入的入口模块的键名 index:"./src/index.js"

}),

# 16. devServer:配置项

 {

        contentBase: path.join(\_\_dirname, 'dist'),

        // 指定web服务器的根目录

        compress: true, //会 gzip(压缩) 和 serve(服务) 所有来自项目根路径下 dist/ 目录的文件

        port: 9000,

        proxy: { //配置服务器代理

            "/data": { //地址

                "target": "http://www.bjlink32.com/data.php",

                //接口地址,跨域访问

                //公司测试服务器地址

                // secure: false,// 如果是https接口，需要配置这个参数

                "changeOrigin": true, //开启跨域

                "pathRewrite": {

                    "^/data": ""

                } //如果接口本身没有/data需要通过pathRewrite来重写了地址

            }

        }

    }

# 17大中型公司开发业务流程:

后台写好接口放到开发服务器

接口文档发给前端

前端在开发环境通过代理服务器访问接口进行开发

后台写完真实接口后，同步到开发服务器

前端开发完毕，同步到开发服务器

前端联调，只需要在开发环境即可

# 18.文件后缀名如何忽略不写

添加配置项: //设置配置项，这些扩展名可以忽略不写

resolve:{ //解决

extensions:['.jsx','.less','.js','.css'] //扩展

},

# 19安装webpackbar插件，显示打包进度条

cnpm i webpackbar -D

# 20//覆盖层，错误信息显示在浏览器中

1.修改webpack配置文件，webpack.config.dev.js

devServer: {

contentBase: path.join(\_\_dirname, 'dist'),

compress: true,//会 gzip(压缩) 和 serve(服务) 所有来自项目根路径下 dist/ 目录的文件

port: 9000,

overlay: { //覆盖层，错误信息显示在浏览器中

warnings: true,

errors: true

},

}

# 开发场景1:CSS3样式如何自动加前缀

解决:PostCSS-loader及autoprefixer

1.postCSS-loader功能

引入其他插件，并且配置其他插件

问题:移动端开发、CSS3开发为了兼容指定浏览器需要加前缀，如果所有样式都加前缀，兼容好，CSS文件体积太大；如果按照前端开发规范中兼容类别进行添加前缀，还得需要自己去caniuse.com网站去查兼容性，再决定哪个样式加还是不加，效率太低了

我们希望，前端开发规范给我们一个兼容列表，自动就能够根据兼容类别决定哪些样式加，哪些样式不加前缀

解决:PostCSS-loader及autoprefixer

在.browserList 配置好兼容浏览器列表， autoprefixer自动按照兼容列表上的浏览器自动给需要样式加前缀

说明:postCSS-loader:引入配置其他css插件

Atuoprefixer 插件可以自动按需添加样式的前缀

cnpm install postcss-loader@3.0.0 autoprefixer@9.8.0 -D

{

loader:'postcss-loader',

options:{

plugins:()=>[

require('autoprefixer')({

//指定需要兼容的浏览器版本

//autoprefixer会参考caniuse.com进行加前缀

browsers:['last 2 version','>1%','ios 7','last 3 ie version']

//兼容最近2个版本，用户量大于1%的浏览器，兼容ios 7以上的系统上的浏览器

})

]

}

}

删除webpack.config.dev.js和webpack.config.prod.js中browsers配置项

在项目根目录下创建autoprefixer配置文件.browserslistrc

添加支持的浏览器即可

last 2 version

>1%

ios 7

last 3 ie version

# 开发场景2:资源内联

**--row-loader**

为什么使用资源内联:移动端每个页面都需要写很多meta标记，如果一个一个粘过去不好维护，可以使用HTML内联

---css或js嵌入到html中，html文件下载完毕，css或js也加载完毕

--小图标转成base64，嵌入到css/js中 url-loader

url-loader可以将小图标打包到css中

场景1:移动端开发每个页面都有很多meta一个页面一个页面添加太麻烦，

所以用raw-loader内联到html中

移动端响应式开发每个页面都需要用到flexible，需要将代码内联到html中

<script> //flexible代码</script> 也是使用raw-loader

cnpm install raw-loader@0.5.1 -D

src文件夹下新建meta.html

<meta http-equiv="Cache-Control" content="max-age=180">

<meta name="apple-mobile-web-app-capable" content="yes">

。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。

(3) 安装lib-flexible包,当做初始化脚本

cnpm install lib-flexible -S

(4)修改public文件夹下的模板文件index.html

内联HTML资源:

${require('raw-loader!../src/meta.html')}

将src目录下的meta。Html文件通过raw-loder引入到当前位置

内联js资源:

<script>

${require('raw-loader!babel-loader!../node\_modules/lib-flexible/flexible.js')}

</script>

将../node\_modules/lib-flexible/flexible.js 文件通过babel-loader转码后，再通过raw-loader插入到当前位置

5.运行npm run build,

模板页面中index.html、one.html中 meta.html内容及flexible.js内容全部内联（插入）到HTML中

资源内联的内容，所有html文件都会生效

<meta http-equiv="Cache-Control" content="max-age=180">

<meta name="apple-mobile-web-app-capable" content="yes">

<meta name="apple-mobile-web-app-status-bar-style" content="black">

<meta name="format-detection" content="telephone=no">

<link rel="dns-prefetch" href="//m.360buyimg.com">

<meta name="format-detection" content="telephone=no">

<meta http-equiv="x-dns-prefetch-control" content="on" />

<link rel="dns-prefetch" href="//img10.360buyimg.com">

<link rel="dns-prefetch" href="//img11.360buyimg.com">

<link rel="dns-prefetch" href="//img12.360buyimg.com">

<link rel="dns-prefetch" href="//img13.360buyimg.com">

<link rel="dns-prefetch" href="//img14.360buyimg.com">

<link rel="dns-prefetch" href="//img20.360buyimg.com">

<link rel="dns-prefetch" href="//img30.360buyimg.com">

<link rel="dns-prefetch" href="//wq.360buyimg.com">

# 开发场景3:减少打包文件体积

3.安装插件，讲react，react-dom 通过cdn服务器拽取，叫内容分发网络

问题:如果所有代码都打包到bundle.js文件，造成文件过大，加载速度慢

场景1:html-webpack-externals-plugin插件 分离公共库，例如react react-dom 可以cdn库

公共库使用cdn，不打包到打包文件中，会把cdn地址自动插入到.html文件中

cnpm install html-webpack-externals-plugin -D

const HtmlWebpackExternalsPlugin = require('html-webpack-externals-plugin');

  new HtmlWebpackExternalsPlugin({

      externals: [{

          module: 'react', //引入的模块

          entry: 'https://unpkg.com/react@16/umd/react.development.js', //cdn服务器

          global: 'React', //全局对象window.React，可以全局使用了

        },

        {

          module: 'react-dom',

          entry: 'https://unpkg.com/react-dom@16/umd/react-dom.development.js',

          global: 'ReactDOM',

        },

      ],

    })

可以使用html-webpack- externals -plugin插件，配置plugin插件，实现公共资源包在cdn服务器上拽取

场景2:使用 SplitChunksPlugin插件 ---webpack 内置插件

代码分离，将react等公共库 分离成独立文件

--分离公共脚本，比如react 、react-dom、或自己创建的对象、函数等，，也就是说分离出来的脚本不会打包到bundle中去了，公共脚本单独打包成一个文件

在prod 开发环境进行配置

optimization: {

    minimize: true,

    minimizer: [

      // 对js压缩的

      new TerserPlugin({

        cache: true,

        parallel: true

      }),

      // 对css进行压缩的

      new OptimizeCSSAssetsPlugin({})

    ],

    // 代码分离，将react和react-dom分离出来，分离包名叫vendors，无论同步异步还是按需加载all分离

    splitChunks: {

      cacheGroups: {

        commons: { //分离公共包

          test: /(react|react-dom)/, // 要分离的包名

          name: 'vendors', // 分离包的名字

          chunks: 'all'

        },

      }

    },

  },

在base配置项中加上vendors index。Html页面加载index和vendors的js文件

 plugins:[

      //配置多个应用

      new HtmlWebpackPlugin({

        title: 'index1',

        filename:"index.html",

        template:"./public/index.html",

        chunks:['vendors',"index"]

      }),

可以配置大于两次引用的文件，单独打包成文件，不用每个引入的都打包

splitChunks: {

      minSize: 0,

      cacheGroups: {

        commons: { // 打包两个页面的公共代码

          name: 'common', // 分离包的名字

          chunks: 'all',

          minChunks: 2      //引用次数大于两次的文件分离

        },

      }

    },

# ESlint:团队开发JavaScript强制代码规范

规范团队代码--形成统一的代码风格、书写更改 保证整个团队开发代码风格一致、减少错误发生、提升开发效率

5.eslint安装

公司使用的eslint规范是什么

# 开发场景4:打包速度分析

--开发中发现打包速度非常慢，分析哪些loader或 plugin造成的，然后解决

cnpm install speed-measure-webpack-plugin -D

2.修改webpack.config.prod.js，用smp.wrap包裹所有配置

const SpeedMeasurePlugin = require("speed-measure-webpack-plugin");  
const smp = new SpeedMeasurePlugin();

module.exports=smp.wrap(等号右侧所有内容包裹);

# 开发场景5:打包文件体积分析

--开发中如果发现打包文件体积过大，分析插件对象打包文件进行分析，详细查看每个包体积，以便我们进一步进行分离或优化

cnpm install webpack-bundle-analyzer -D

2.修改webpack.config.prod.js

const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;

plugins:[

new BundleAnalyzerPlugin()

],

# 开发场景6:提升打包的速度

--thread-loader实现多进程打包

2.修改webpack.config.base.js

{

test: /\.(js|jsx)$/,

exclude: /(node\_modules|bower\_components)/,

use: [

{

loader:'thread-loader',

options:{

worker:3 //三进程打包

}

如何提升打包速度：

Js压缩，css压缩，将公共代码抽离，再cdn服务器拽取资源，多进程打包

optimization: {

    minimize: true,

    minimizer: [

      // 对js压缩的

      new TerserPlugin({

        cache: true,

        parallel: true

      }),

      // 对css进行压缩的

      new OptimizeCSSAssetsPlugin({})

    ],

    // 代码分离，将react和react-dom分离出来，分离包名叫vendors，无论同步异步还是按需加载all分离

    splitChunks: {

      cacheGroups: {

        commons: { //分离公共包

          test: /(react|react-dom)/, // 要分离的包名

          name: 'vendors', // 分离包的名字

          chunks: 'all'

        },

      }

    },

  },

简书最后有一个异步加载组件的插件

React-loadable 读音：react-l od bo

1.安装Cnpm I -S react-loadable

2.创建Loadable组件

import React from 'react'

import Loadable from 'react-loadable';

const LoadableComponent = Loadable({

  loader: () => import('./index'),

  loading(){

      return <div>正在加载。。。。</div>

  }

});

export default ()=><LoadableComponent/>

3.修改路由，访问detailLoadable.js组件

4.引入高阶组件，包裹导出的组件，使其获得router的参数

import {withRouter} from 'react-router-dom'

export default connect(mapStateToProps,mapDispatchToProps)(withRouter(Detail))

// withRouter有能力获取router里的所有参数和内容