link: null
title: 珠峰架构师成长计划
description: 咨询珠峰架构课请加微信
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=395 sentences=853, words=5520

咨询珠峰架构课请加微信□

1.与webpack类似的工具还有哪些?谈谈你为什么选择使用或放弃webpack?

1.1 grunt

- 一句话：自动化。对于需要反复重复的任务，例如压缩（minification）、编译、单元测试、linting等，自动化工具可以减轻你的劳动，简化你的工作。当你在 `Gruntfile` 文件正确配置好了任务，任务运行器就会自动帮你或你的小组完成大部分无聊的工作。
- 最老牌的打包工具,它运用配置的思想来写打包脚本，一切皆配置

1.1.1 优点

- 出现的比较早**1.1.2 缺点#**
- 配置项太多
- 而且不同的插件可能会有自己扩展字段
- 学习成本高,运用的时候需要明白各种插件的配置规则和配合方式

** 1.1.3 执行任务 #**

```
npm run build
```

1.2 gulp

- 基于 nodejs 的 `steam` 流打包
- 定位是基于任务流的自动化构建工具
- Gulp是通过task对整个开发过程进行构建

** 1.2.1 优点 #**

- 流式的写法简单直观
- API简单,代码量少
- 易于学习和使用
- 适合多页面应用开发

** 1.2.2 缺点 #**

- 异常处理比较麻烦
- 工作流程顺序难以精细控制
- 不太适合单页或者自定义模块的开发

** 1.2.3 执行任务 #**

```
npm run build
```

1.3 webpack

- webpack 是模块化管理工具和打包工具。通过 loader 的转换，任何形式的资源都可以视作模块，比如 CommonJs 模块、AMD 模块、ES6 模块、CSS、图片等。它可以将许多松散的模块按照依赖和规则打包成符合生产环境部署的前端资源
- 还可以将按需加载的模块进行代码分隔，等到实际需要的时候再异步加载
- 它定位是模块打包器，而 Gulp/Grunt 属于构建工具。Webpack 可以代替 Gulp/Grunt 的一些功能，但不是一个职能的工具，可以配合使用

** 1.3.1 优点 #**

- 可以模块化的打包任何资源
- 适配任何模块系统
- 适合SPA单页应用的开发

** 1.3.2 缺点 #**

- 学习成本高,配置复杂
- 通过babel编译后的js代码打包后体积过大

** 1.3.3 执行任务 #**

1.4 rollup

- rollup 下一代 ES6 模块化工具，最大的亮点是利用 ES6 模块设计，利用 tree-shaking生成更简洁、更简单的代码
- 一般而言，对于应用使用 Webpack，对于类库使用 Rollup
- 需要代码拆分(Code Splitting)，或者很多静态资源需要处理，再或者构建的项目需要引入很多 CommonJS 模块的依赖时，使用 webpack
- 代码库是基于 ES6 模块，而且希望代码能够被其他人直接使用，使用 Rollup

** 1.4.1 优点 #**

- 用标准化的格式（es6）来写代码，通过减少死代码尽可能地缩小包体积**1.4.2 缺点#**
- 对代码拆分、静态资源、CommonJS模块支持不好

** 1.4.3 执行任务 #**

```
npm run build
```

1.5 parcel

- Parcel 是快速、零配置的 Web 应用程序打包器
- 目前 Parcel 只能用来构建用于运行在浏览器中的网页，这也是他的出发点和专注点

** 1.5.1 优点 #**

- Parcel 内置了常见场景的构建方案及其依赖，无需再安装各种依赖
- Parcel 能以 HTML 为入口，自动检测和打包依赖资源
- Parcel 默认支持模块热替换，真正的开箱即用

** 1.5.2 缺点 #**

- 不支持 SourceMap
- 不支持剔除无效代码(TreeShaking)
- 配置不灵活**1.5.3 执行任务#**

```
npm run start
```

2 如何调试webpack代码

debug.js

```
const webpack = require("webpack");
const config = require("./webpack.config.js");
debugger;
const compiler = webpack(config);
function compilerCallback(err, stats) {
  const statsString = stats.toString();
  console.log(statsString);
}
debugger
compiler.run((err, stats) => {
  compilerCallback(err, stats);
});
```

3. Loader和Plugin的不同？#

- Loader直译为 加载器。Webpack将一切文件视为模块，但是webpack原生是只能解析js文件，如果想将其他文件也打包的话，就会用到loader。 所以Loader的作用是让webpack拥有了加载和解析非JavaScript文件的能力
- Plugin直译为 插件。Plugin可以扩展webpack的功能，让webpack具有更多的灵活性。 在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果

4. webpack的构建流程是什么？#

- 初始化参数：从配置文件和Shell语句中读取与合并参数，得出最终的参数；
- 开始编译：用上一步得到的参数初始化Compiler对象，加载所有配置的插件，执行对象的run方法开始执行编译； 确定入口：根据配置中的entry找出所有的入口文件
- 编译模块：从入口文件出发，调用所有配置的Loader对模块进行编译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；
- 完成模块编译：在经过第 4 步使用Loader翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个Chunk转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会
- 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统

> 在以上过程中，Webpack 会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用 Webpack 提供的 API 改变 Webpack 的运行结果

4.1 webpack.config.js

```
const path = require("path");
const RunPlugin = require("./plugins/RunPlugin");
const DonePlugin = require("./plugins/DonePlugin");
module.exports = {
  context: process.cwd(),
  mode: "development",
  devtool: false,
  entry: "./src/app.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "main.js",
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        use: {
          loader: "babel-loader",
          options: {
            presets: ["@babel/preset-env"],
          },
        },
        include: path.join(__dirname, "src"),
        exclude: /node_modules/,
      },
    ],
  },
  plugins: [new RunPlugin(),new DonePlugin()],
  devServer: {},
};
```

4.2 flow.js

flow.js

```
let fs = require('fs');
let path = require('path');
const { SyncHook } = require("tapable");

class Compiler {
  constructor(options) {
    this.options = options;
    this.hooks = {
      run: new SyncHook(),
      done: new SyncHook(),
    };
  }
  run() {
    this.hooks.run.call();
    let modules = [];
    let chunks = [];
    let files = [];

    let entry = path.join(this.options.context, this.options.entry);

    let entryContent = fs.readFileSync(entry, "utf8");
    let entrySource = babelLoader(entryContent);
    let entryModule = { id: entry, source: entrySource };
    modules.push(entryModule);

    let title = path.join(this.options.context, "./src/title.js");
    let titleContent = fs.readFileSync(title, "utf8");
    let titleSource = babelLoader(titleContent);
    let titleModule = { id: title, source: titleSource };
    modules.push(titleModule);

    let chunk = { name: "main", modules };
    chunks.push(chunk);

    let file = {
      file: this.options.output.filename,
      source: `
(function (modules) {
  function __webpack_require__(moduleId) {
    var module = { exports: {} };
    modules[moduleId].call(
    module.exports,
    module,
    module.exports,
    __webpack_require__
    );
    return module.exports;
  }
  return __webpack_require__("./src/app.js");
 })(
{
 "./src/app.js": function (module, exports, __webpack_require__) {
    var title = __webpack_require__("./src/title.js");
    console.log(title);
 },
 "./src/title.js": function (module) {
    module.exports = "title";
 },
});
`,
    };
    files.push(file);

    let outputPath = path.join(
      this.options.output.path,
      this.options.output.filename
    );
    fs.writeFileSync(outputPath, file.source,'utf8');
      this.hooks.done.call();
  }
}

let options = require('./webpack.config');

let compiler = new Compiler(options);

if (options.plugins && Array.isArray(options.plugins)) {
  for (const plugin of options.plugins) {
   plugin.apply(compiler);
  }
}

compiler.run();
function babelLoader(source) {
  return `var sum = function sum(a, b) {
              return a + b;
            };`;
}
```

4.3 RunPlugin.js

plugins\RunPlugin.js

```
module.exports = class RunPlugin {
  apply(compiler) {
    compiler.hooks.run.tap("RunPlugin", () => {
      console.log("RunPlugin");
    });
  }
}
```

4.4 DonePlugin.js

plugins\DonePlugin.js

```
module.exports = class DonePlugin {
  apply(compiler) {
    compiler.hooks.done.tap("DonePlugin", () => {
      console.log("DonePlugin");
    });
  }
};
```

5.有哪些常见的Loader和Plugin?他们是解决什么问题的？#

5.1 loader

loader 解决问题 babel-loader 把 ES6 或React转换成 ES5 css-loader 加载 CSS，支持模块化、压缩、文件导入等特性 eslint-loader 通过 ESLint 检查 JavaScript 代码 file-loader 把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件 url-loader 和 file-loader 类似，但是能在文件很小的情况下以 base64 的方式把文件内容注入到代码中去 sass-loader 把Sass/SCSS文件编译成CSS postcss-loader 使用PostCSS处理CSS css-loader 主要来处理background:(url)还有@import这些语法。让webpack能够正确的对其路径进行模块化处理 style-loader 把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS。

5.2 plugin

插件 解决问题 case-sensitive-paths-webpack-plugin 如果路径有误则直接报错 terser-webpack-plugin 使用terser来压缩JavaScript pnp-webpack-plugin Yarn Plug'n'Play插件 html-webpack-plugin 自动生成带有入口文件引用的index.html webpack-manifest-plugin 生产资产的显示清单文件 optimize-css-assets-webpack-plugin 用于优化或者压缩CSS资源 mini-css-extract-plugin 将CSS提取为独立的文件的插件，对每个包含css的js文件都会创建一个CSS文件，支持按需加载css和sourceMap ModuleScopePlugin 如果引用了src目录外的文件报警插件 InterpolateHtmlPlugin 和HtmlWebpackPlugin串行使用，允许在index.html中添加变量 ModuleNotFoundPlugin 找不到模块的时候提供一些更详细的上下文信息 DefinePlugin 创建一个在编译时可配置的全局常量,如果你自定义了一个全局变量PRODUCTION,可在此设置其值来区分开发还是生产环境 HotModuleReplacementPlugin 启用模块热替换 WatchMissingNodeModulesPlugin :此插件允许你安装库后自动重新构建打包文件

5.3 配置

** 5.3.1 webpack.config.js #**

```
;

const webpack = require('webpack');
const paths = require("./paths");
const CaseSensitivePathsPlugin = require("case-sensitive-paths-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const PnpWebpackPlugin = require("pnp-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const ManifestPlugin = require("webpack-manifest-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");
const InlineChunkHtmlPlugin = require("react-dev-utils/InlineChunkHtmlPlugin");
const WatchMissingNodeModulesPlugin = require("react-dev-utils/WatchMissingNodeModulesPlugin");
const ModuleScopePlugin = require("react-dev-utils/ModuleScopePlugin");
const InterpolateHtmlPlugin = require("react-dev-utils/InterpolateHtmlPlugin");
const ModuleNotFoundPlugin = require("react-dev-utils/ModuleNotFoundPlugin");
const getClientEnvironment = require("./env");
const cssRegex = /\.css$/;
const sassRegex = /\.(scss|sass)$/;
module.exports = function (webpackEnv) {
  console.log("webpackEnv", webpackEnv);

  const isEnvDevelopment = webpackEnv === "development";

  const isEnvProduction = webpackEnv === "production";

  const shouldUseSourceMap = process.env.GENERATE_SOURCEMAP !== "false";
  const shouldInlineRuntimeChunk = process.env.INLINE_RUNTIME_CHUNK !== "false";
  console.log("shouldInlineRuntimeChunk", shouldInlineRuntimeChunk);

  const env = getClientEnvironment(paths.publicUrlOrPath.slice(0, -1));
  console.log("env", env);
  const getStyleLoaders = (cssOptions, preProcessor) => {
    const loaders = [
      isEnvDevelopment && require.resolve("style-loader"),
      isEnvProduction && {
        loader: MiniCssExtractPlugin.loader,
      },
      {
        loader: require.resolve("css-loader"),
        options: cssOptions,
      },
      {
        loader: require.resolve("postcss-loader"),
      },
    ].filter(Boolean);
        if (preProcessor) {
          loaders.push(
            {
              loader: require.resolve("resolve-url-loader")
            },
            {
              loader: require.resolve(preProcessor),
              options: {
                sourceMap: true,
              },
            }
          );
        }
    return loaders;
  };
  return {
    mode: isEnvProduction ? "production" : "development",
    devtool: isEnvProduction
      ? shouldUseSourceMap
        ? "source-map"
        : false
      : isEnvDevelopment && "cheap-module-source-map",
    entry: [
      isEnvDevelopment &&
        require.resolve("react-dev-utils/webpackHotDevClient"),
      paths.appIndexJs,
    ].filter(Boolean),
    output: {
      path: isEnvProduction ? paths.appBuild : undefined,
```

```
    filename: isEnvProduction
      ? "static/js/[name].[contenthash:8].js"
      : "static/js/bundle.js",

    chunkFilename: isEnvProduction
      ? "static/js/[name].[contenthash:8].chunk.js"
      : "static/js/[name].chunk.js",

    publicPath: paths.publicUrlOrPath,
  },
  optimization: {
    minimize: isEnvProduction,
    minimizer: [

      new TerserPlugin({}),

      new OptimizeCSSAssetsPlugin({}),
    ],

    splitChunks: {
      chunks: "all",
      name: false,
    },

    runtimeChunk: {
      name: (entrypoint) => `runtime-${entrypoint.name}`,
    },
  },
  resolve: {

    modules: ["node_modules", paths.appNodeModules],

    extensions: paths.moduleFileExtensions.map((ext) => `.${ext}`),
    alias: {

      "react-native": "react-native-web",
    },
    plugins: [
      PnpWebpackPlugin,

      new ModuleScopePlugin(paths.appSrc, [paths.appPackageJson]),
    ],
  },
  resolveLoader: {
    plugins: [PnpWebpackPlugin.moduleLoader(module)],
  },
  module: {
    rules: [

      {
        test: /\.(js|mjs|jsx|ts|tsx)$/,
        enforce: "pre",
        use: [
          {
            loader: require.resolve("eslint-loader"),
          },
        ],
        include: paths.appSrc,
      },
      {

        oneOf: [
          {
            test: [/\.bmp$/, /\.gif$/, /\.jpe?g$/, /\.png$/],
            loader: require.resolve("url-loader"),
          },
          {
            test: /\.(js|mjs|jsx|ts|tsx)$/,
            include: paths.appSrc,
            loader: require.resolve("babel-loader"),
          },
          {
            test: /\.(js|mjs)$/,
            loader: require.resolve("babel-loader"),
          },
          {
            test: cssRegex,

            use: getStyleLoaders({ importLoaders: 1 }),
          },
          {
            test: sassRegex,

            use: getStyleLoaders({ importLoaders: 3 }, "sass-loader"),
          },
          {
            loader: require.resolve("file-loader"),
            exclude: [/\.(js|mjs|jsx|ts|tsx)$/, /\.html$/, /\.json$/],
            options: {
              name: "static/media/[name].[hash:8].[ext]",
            },
          },
        ],
      },
    ],
  },
  plugins: [

    new HtmlWebpackPlugin({
      inject: true,
      template: paths.appHtml,
    }),
    isEnvProduction &&
      shouldInlineRuntimeChunk &&
```

```
        new InlineChunkHtmlPlugin(HtmlWebpackPlugin, [/runtime-.+[.]js/]),

      new InterpolateHtmlPlugin(HtmlWebpackPlugin, env.raw),

      new ModuleNotFoundPlugin(paths.appPath),

      new webpack.DefinePlugin(env.stringified),

      isEnvDevelopment && new webpack.HotModuleReplacementPlugin(),

      isEnvDevelopment && new CaseSensitivePathsPlugin(),

      isEnvDevelopment &&
        new WatchMissingNodeModulesPlugin(paths.appNodeModules),

      isEnvProduction &&
        new MiniCssExtractPlugin({
          filename: "static/css/[name].[contenthash:8].css",
          chunkFilename: "static/css/[name].[contenthash:8].chunk.css",
        }),

      new ManifestPlugin({
        fileName: "asset-manifest.json",
        publicPath: paths.publicUrlOrPath,
        generate: (seed, files, entrypoints) => {
          const manifestFiles = files.reduce((manifest, file) => {
            manifest[file.name] = file.path;
            return manifest;
          }, seed);
          const entrypointFiles = entrypoints.main.filter(
            (fileName) => !fileName.endsWith(".map")
          );

          return {
            files: manifestFiles,
            entrypoints: entrypointFiles,
          };
        },
      }),
    ].filter(Boolean),
  };
};
```

** 5.3.2 paths.js#**

paths.js

```
;

const path = require('path');
const fs = require('fs');

const appDirectory = fs.realpathSync(process.cwd());

const resolveApp = relativePath => path.resolve(appDirectory, relativePath);

const publicUrlOrPath = require(resolveApp("package.json")).homepage || process.env.PUBLIC_URL || "";

const moduleFileExtensions = [
  'web.mjs',
  'mjs',
  'web.js',
  'js',
  'web.ts',
  'ts',
  'web.tsx',
  'tsx',
  'json',
  'web.jsx',
  'jsx',
];

const resolveModule = (resolveFn, filePath) => {
  const extension = moduleFileExtensions.find(extension =>
    fs.existsSync(resolveFn(`${filePath}.${extension}`))
  );

  if (extension) {
    return resolveFn(`${filePath}.${extension}`);
  }

  return resolveFn(`${filePath}.js`);
};

module.exports = {
  dotenv: resolveApp('.env'),
  appPath: resolveApp('.'),
  appBuild: resolveApp('build'),
  appPublic: resolveApp('public'),
  appHtml: resolveApp('public/index.html'),
  appIndexJs: resolveModule(resolveApp, 'src/index'),
  appPackageJson: resolveApp('package.json'),
  appSrc: resolveApp('src'),
  appTsConfig: resolveApp('tsconfig.json'),
  appJsConfig: resolveApp('jsconfig.json'),
  appNodeModules: resolveApp('node_modules'),
  publicUrlOrPath,
};

module.exports.moduleFileExtensions = moduleFileExtensions;
```

** 5.3.3 env.js#**

env.js

```
;

const fs = require('fs');
const path = require('path');
const paths = require('./paths');

const NODE_ENV = process.env.NODE_ENV;

const dotenvFiles = [
  `${paths.dotenv}.${NODE_ENV}.local`,
  `${paths.dotenv}.${NODE_ENV}`,

  NODE_ENV !== 'test' && `${paths.dotenv}.local`,
  paths.dotenv,
].filter(Boolean);

dotenvFiles.forEach(dotenvFile => {
  if (fs.existsSync(dotenvFile)) {
    require('dotenv-expand')(
      require('dotenv').config({
        path: dotenvFile,
      })
    );
  }
});

const appDirectory = fs.realpathSync(process.cwd());
process.env.NODE_PATH = (process.env.NODE_PATH || '')
  .split(path.delimiter)
  .filter(folder => folder && !path.isAbsolute(folder))
  .map(folder => path.resolve(appDirectory, folder))
  .join(path.delimiter);

const REACT_APP = /^REACT_APP_/i;
function getClientEnvironment(publicUrl) {
  const raw = Object.keys(process.env)
    .filter((key) => REACT_APP.test(key))
    .reduce(
      (env, key) => {
        env[key] = process.env[key];
        return env;
      },
      {

        NODE_ENV: process.env.NODE_ENV || "development",

        PUBLIC_URL: publicUrl,
      }
    );

  const stringified = {
    "process.env": Object.keys(raw).reduce((env, key) => {
      env[key] = JSON.stringify(raw[key]);
      return env;
    }, {}),
  };

  return { raw, stringified };
}

module.exports = getClientEnvironment;
```

** 5.3.4 webpackDevServer.config.js #**

```
;

const fs = require('fs');
const errorOverlayMiddleware = require('react-dev-utils/errorOverlayMiddleware');
const evalSourceMapMiddleware = require('react-dev-utils/evalSourceMapMiddleware');
const ignoredFiles = require('react-dev-utils/ignoredFiles');
const redirectServedPath = require('react-dev-utils/redirectServedPathMiddleware');
const paths = require('./paths');
const host = process.env.HOST || '0.0.0.0';

module.exports = function(proxy, allowedHost) {
  return {

    disableHostCheck: true,

    compress: true,

    clientLogLevel: "none",

    contentBase: paths.appPublic,
    contentBasePublicPath: paths.publicUrlOrPath,

    watchContentBase: true,

    hot: true,

    transportMode: "ws",

    injectClient: false,

    publicPath: paths.publicUrlOrPath.slice(0, -1),

    quiet: true,
    watchOptions: {
      ignored: ignoredFiles(paths.appSrc),
    },
    host,
    historyApiFallback: {

      disableDotRule: true,
      index: paths.publicUrlOrPath,
    },
    public: allowedHost,

    proxy,
    before(app, server) {

      app.use(evalSourceMapMiddleware(server));

      app.use(errorOverlayMiddleware());

      if (fs.existsSync(paths.proxySetup)) {
        require(paths.proxySetup)(app);
      }
    },
    after(app) {

      app.use(redirectServedPath(paths.publicUrlOrPath));
    },
  };
};
```

** 5.3.5 webpack.dev.config.js #**

webpack.dev.config.js

```

const { merge } = require("webpack-merge");
let config = require('./webpack.config');
let devServerConfig = require('./webpackDevServer.config');
module.exports = merge(config('development'), {
  devServer: devServerConfig()
});
```

** 5.3.6 postcss.config.js #**

postcss.config.js

```
module.exports = {
  plugins: [
    require("autoprefixer")({ overrideBrowserslist: ["> 0.15% in CN"] }),
  ],
};
```

** 5.3.7 .eslintrc.json #**

```
{
    "env": {
        "browser": true,
        "es2020": true
    },
    "parserOptions": {
        "ecmaVersion": 2020,
        "sourceType": "module"
    },
    "rules": {
    }
}
```

** 5.3.8 package.json #**

```
{
   "scripts": {
     "build": "webpack --env=production",
     "start": "webpack-dev-server --env=development --config webpack.dev.config.js"
   }
}
```

6. source map是什么？生产环境怎么用？#

- sourcemap是为了解决开发代码与实际运行代码不一致时帮助我们debug到原始开发代码的技术
- webpack通过配置可以自动给我们 source maps文件， map文件是一种对应编译文件和源文件的方法
- whyeval (https://github.com/webpack/docs/wiki/build-performance#sourcemaps)
- source-map (https://github.com/mozilla/source-map)
- javascript_source_map算法 (http://www.ruanyifeng.com/blog/2013/01/javascript_source_map.html)

6.1 source map的类型

类型 含义 source-map 原始代码 最好的sourcemap质量有完整的结果，但是会很慢 eval-source-map 原始代码 同样道理，但是最高的质量和最低的性能 cheap-module-eval-source-map 原始代码（只有行内） 同样道理，但是更高的质量和更低的性能 cheap-eval-source-map 转换代码（行内） 每个模块被eval执行，并且sourcemap作为eval的一个dataurl eval 生成代码 每个模块都被eval执行，并且存在@sourceURL,带eval的构建模式能cache SourceMap cheap-source-map 转换代码（行内） 生成的sourcemap没有列映射，从loaders生成的sourcemap没有被使用 cheap-module-source-map 原始代码（只有行内） 与上面一样除了每行特点的从loader中进行映射

看似配置项很多， 其实只是五个关键字eval、source-map、cheap、module和inline的任意组合

关键字 含义 eval 使用eval包裹模块代码 source-map 产生.map文件 cheap 不包含列信息（关于列信息的解释下面会有详细介绍)也不包含loader的sourcemap module 包含loader的sourcemap（比如jsx to js，babel的sourcemap）,否则无法定义源文件 inline 将.map作为DataURI嵌入，不单独生成.map文件

- eval eval执行
- eval-source-map 生成sourcemap
- cheap-module-eval-source-map 不包含列
- cheap-eval-source-map 无法看到真正的源码

6.2 如何选择source map的类型

- 首先在源代码的列信息是没有意义的，只要有行信息就能完整的建立打包前后代码之间的依赖关系。因此，不管是开发还是生产环境都会增加cheap属性来忽略模块打包后的列信息关联
- 不管是生产环境还是开发环境，我们都需要定位debug到最原始的资源，比如定位错误到jsx，ts的原始代码，而不是经编译后的js代码。所以不可以忽略掉module属性
- 需要生成.map文件，所以得有source-map属性
- 总结
    - 开发环境使用：cheap-module-eval-source-map
    - 生产环境使用：cheap-module-source-map

7.如何利用webpack来优化前端性能？#

7.1. 安装

```
cnpm i react react-dom  -S
cnpm install  webpack webpack-cli webpack-dev-server image-webpack-loader mini-css-extract-plugin purgecss-webpack-plugin babel-loader @babel/core
@babel/preset-env @babel/preset-react terser-webpack-plugin html-webpack-plugin optimize-css-assets-webpack-plugin mini-css-extract-plugin qiniu -D
```

7.2.压缩JS

```
  optimization: {
    minimize: true,
    minimizer: [
      //压缩JS
+     new TerserPlugin({})
    ]
  },
```

7.3. 压缩CSS

```
  optimization: {
    minimize: true,
    minimizer: [
      //压缩CSS
+      new OptimizeCSSAssetsPlugin({}),
    ]
  },
```

7.4. 压缩图片

```
      {
        test: /\.(png|svg|jpg|gif|jpeg|ico)$/,
        use: [
          "file-loader",
          {
+            loader: "image-webpack-loader",
+            options: {
+              mozjpeg: {
+                progressive: true,
+                quality: 65,
+              },
+              optipng: {
+                enabled: false,
+              },
+              pngquant: {
+                quality: "65-90",
+                speed: 4,
+              },
+              gifsicle: {
+                interlaced: false,
+              },
+              webp: {
+                quality: 75,
+              }
+            }
+          }
        ]
      }
```

7.5. 清除无用的CSS

- 单独提取CSS并清除用不到的CSS

```
const path = require("path");
+const MiniCssExtractPlugin = require("mini-css-extract-plugin");
+const PurgecssPlugin = require("purgecss-webpack-plugin");
module.exports = {
  module: {
    rules: [
       {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
+           loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      }
    ]
  },
  plugins: [
+    new MiniCssExtractPlugin({
+      filename: "[name].css",
+    }),
+    new PurgecssPlugin({
+      paths: glob.sync(`${PATHS.src}/**/*`,  { nodir: true }),
+    })
  ]
  devServer: {},
};
```

7.6. Tree Shaking

- 一个模块可以有多个方法，只要其中某个方法使用到了，则整个文件都会被打到bundle里面去，tree shaking就是只把用到的方法打入bundle,没用到的方法会uglify阶段擦除掉
- 原理是利用es6模块的特点,只能作为模块顶层语句出现,import的模块名只能是字符串常量
- webpack默认支持，在.babelrc里设置module:false即可在production mode下默认开启

```
module.exports = {
+    mode:'production',
+    devtool:false,
     module: {
        rules: [
            {
                test: /\.js/,
                include: path.resolve(__dirname, "src"),
                use: [
                    {
                        loader: "babel-loader",
                        options: {
+                           presets: [["@babel/preset-env", { "modules": false }]],
                        },
                    },
                ],
            }
     }
}
```

7.7. Scope Hoisting

- Scope Hoisting 可以让 Webpack 打包出来的代码文件更小、运行的更快， 它又译作 "作用域提升"，是在 Webpack3 中新推出的功能。
- scope hoisting的原理是将所有的模块按照引用顺序放在一个函数作用域里，然后适当地重命名一些变量以防止命名冲突
- 这个功能在mode为 `production`下默认开启,开发环境要用 `webpack.optimize.ModuleConcatenationPlugin`插件

hello.js

```
export default 'Hello';
```

index.js

```
import str from './hello.js';
console.log(str);
```

main.js

```
var hello = ('hello');
console.log(hello);
```

7.8. 代码分割

- 对于大的Web应用来讲，将所有的代码都放在一个文件中显然是不够有效的，特别是当你的某些代码块是在某些特殊的时候才会被用到。
- webpack有一个功能就是将你的代码库分割成chunks语块，当代码运行到需要它们的时候再进行加载

** 7.8.1 入口点分割 #**

- Entry Points：入口文件设置的时候可以配置
- 这种方法的问题
    - 如果入口 chunks 之间包含重复的模块(lodash)，那些重复模块都会被引入到各个 bundle 中
    - 不够灵活，并且不能将核心应用程序逻辑进行动态拆分代码

```
entry: {
        index: "./src/index.js",
        login: "./src/login.js"
}
```

** 7.8.2 动态导入和懒加载 #**

- 用户当前需要用什么功能就只加载这个功能对应的代码，也就是所谓的按需加载 在给单页应用做按需加载优化时
- 一般采用以下原则：
    - 对网站功能进行划分，每一类一个chunk
    - 对于首次打开页面需要的功能直接加载，尽快展示给用户,某些依赖大量代码的功能点可以按需加载
    - 被分割出去的代码需要一个按需加载的时机

7.8.2.1 hello.js

hello.js

```
module.exports = "hello";
```

index.js

```
document.querySelector('#clickBtn').addEventListener('click',() => {
    import('./hello').then(result => {
        console.log(result.default);
    });
});
```

index.html

```
<button id="clickBtn">点我button>
```

7.8.2.2 按需加载

- 如何在react项目中实现按需加载?

7.8.2.2.1 index.js

index.js

```
import React, { Component, Suspense } from "react";
import ReactDOM from "react-dom";
import Loading from "./components/Loading";

const AppTitle = React.lazy(() =>
  import( "./components/Title")
);

class App extends Component {
  constructor(){
    super();
    this.state = {visible:false};
  }
  show(){
    this.setState({ visible: true });
  };
  render() {
    return (
      <>
        {this.state.visible && (
          <Suspense fallback={<Loading />}>
            <AppTitle />
          Suspense>
        )}
        <button onClick={this.show.bind(this)}>加载button>
      </>
    );
  }
}
ReactDOM.render(<App />, document.querySelector("#root"));
```

7.8.2.2.2 Loading.js

src\components\Loading.js

```
import React, { Component, Suspense } from "react";
export default (props) => {
  return <p>Loadingp>;
};
```

7.8.2.2.3 Title.js

src\components\Title.js

```
import React, { Component, Suspense } from "react";
export default props=>{
  return <p>Titlep>;
}
```

** 7.8.3 preload(预先加载) #**

- preload通常用于本页面要用到的关键资源，包括关键js、字体、css文件
- preload将会把资源得下载顺序权重提高，使得关键数据提前下载好,优化页面打开速度
- 在资源上添加预先加载的注释，你指明该模块需要立即被使用
- 一个资源的加载的优先级被分为五个级别,分别是
    - Highest 最高
    - High 高
    - Medium 中等
    - Low 低
    - Lowest 最低
- 异步/延迟/插入的脚本（无论在什么位置）在网络优先级中是 Low


```
"preload" as="script" href="utils.js">
```

```
import(
  `./utils.js`

)
```

** 7.8.4 prefetch(预先拉取) #**

- prefetch 跟 preload 不同，它的作用是告诉浏览器未来可能会使用到的某个资源，浏览器就会在闲时去加载对应的资源，若能预测到用户的行为，比如懒加载，点击到其它页面等则相当于提前预加载了需要的

资源

```
<link rel="prefetch" href="utils.js" as="script">
```

```
button.addEventListener('click', () => {
  import(
    `./utils.js`

  ).then(result => {
    result.default.log('hello');
  })
});
```

** 7.8.5 preload vs prefetch #**

- preload 是告诉浏览器页面必定需要的资源，浏览器一定会加载这些资源
- 而 prefetch 是告诉浏览器页面可能需要的资源，浏览器不一定会加载这些资源
- 所以建议：对于当前页面很有必要的资源使用 preload,对于可能在将来的页面中使用的资源使用 prefetch

** 7.8.6 提取公共代码 #**

- 怎么配置单页应用?怎么配置多页应用?**7.8.6.1 为什么需要提取公共代码#**
- 大网站有多个页面，每个页面由于采用相同技术栈和样式代码，会包含很多公共代码，如果都包含进来会有问题
- 相同的资源被重复的加载，浪费用户的流量和服务器的成本；
- 每个页面需要加载的资源太大，导致网页首屏加载缓慢，影响用户体验。
- 如果能把公共代码抽离成单独文件进行加载能进行优化，可以减少网络传输流量，降低服务器成本

** 7.8.6.2 如何提取 #**

- 基础类库，方便长期缓存
- 页面之间的公用代码
- 各个页面单独生成文件

** 7.8.6.3 splitChunks #**** 7.8.6.3.1 module chunk bundle #**

- module：就是js的模块化webpack支持commonJS、ES6等模块化规范，简单来说就是你通过import语句引入的代码
- chunk: chunk是webpack根据功能拆分出来的，包含三种情况
    - 你的项目入口（entry）
    - 通过import()动态引入的代码
    - 通过splitChunks拆分出来的代码
- bundle：bundle是webpack打包之后的各个文件，一般就是和chunk是一对一的关系，bundle就是对chunk进行编译压缩打包等处理之后的产出

** 7.8.6.3.2 默认配置 #**

webpack.config.js

```
  entry: {
    page1: "./src/page1.js",
    page2: "./src/page2.js",
    page3: "./src/page3.js",
  },
 optimization: {
  splitChunks: {
      chunks: "all",
      minSize: 0,
      minChunks: 1,
      maxAsyncRequests: 2,
      maxInitialRequests: 4,
      name: true,
      automaticNameDelimiter: "~",
      cacheGroups: {

        vendors: {
          chunks: "all",
          test: /node_modules/,
          priority: -10,
        },
        commons: {
          chunks: "all",
          minSize: 0,
          minChunks: 2,
          priority: -20
        }
      }
    }
```

src\page1.js

```
import utils1 from "./module1";
import utils2 from "./module2";
import $ from "jquery";
console.log(utils1, utils2, $);
import( "./asyncModule1");
```

src\page2.js

```
import utils1 from "./module1";
import utils2 from "./module2";
import $ from "jquery";
console.log(utils1, utils2, $);
```

src\page3.js

```
import utils1 from "./module1";
import utils3 from "./module3";
import $ from "jquery";
console.log(utils1, utils3, $);
```

src\module1.js

```
console.log("module1");
```

src\module2.js

```
console.log("module2");
```

src\module3.js

```
console.log("module3");
```

src\asyncModule1.js

```
import _ from 'lodash';
console.log(_);
```

```
                          Asset       Size                  Chunks             Chunk Names
            asyncModule1.chunk.js  740 bytes              asyncModule1  [emitted]  asyncModule1
                       index.html  498 bytes                           [emitted]
                         page1.js   10.6 KiB                    page1  [emitted]  page1
             page1~page2.chunk.js  302 bytes               page1~page2  [emitted]  page1~page2
       page1~page2~page3.chunk.js  308 bytes         page1~page2~page3  [emitted]  page1~page2~page3
                         page2.js   7.52 KiB                    page2  [emitted]  page2
                         page3.js   7.72 KiB                    page3  [emitted]  page3
    vendors~asyncModule1.chunk.js    532 KiB      vendors~asyncModule1  [emitted]  vendors~asyncModule1
vendors~page1~page2~page3.chunk.js    282 KiB  vendors~page1~page2~page3  [emitted]  vendors~page1~page2~page3
Entrypoint page1 = vendors~page1~page2~page3.chunk.js page1~page2~page3.chunk.js page1~page2.chunk.js page1.js
Entrypoint page2 = vendors~page1~page2~page3.chunk.js page1~page2~page3.chunk.js page1~page2.chunk.js page2.js
Entrypoint page3 = vendors~page1~page2~page3.chunk.js page1~page2~page3.chunk.js page3.js
```

7.9 CDN

- 最影响用户体验的是网页首次打开时的加载等待。 导致这个问题的根本是网络传输过程耗时大，CDN的作用就是加速网络传输。
- CDN 又叫内容分发网络，通过把资源部署到世界各地，用户在访问时按照就近原则从离用户最近的服务器获取资源，从而加速资源的获取速度
- 用户使用浏览器第一次访问我们的站点时，该页面引入了各式各样的静态资源，如果我们能做到持久化缓存的话，可以在 http 响应头加上 Cache-control 或 Expires 字段来设置缓存，浏览器可以将这些资源一一缓存到本地
- 用户在后续访问的时候，如果需要再次请求同样的静态资源，且静态资源没有过期，那么浏览器可以直接走本地缓存而不用再通过网络请求资源
- 缓存配置
    - HTML文件不缓存，放在自己的服务器上，关闭自己服务器的缓存，静态资源的URL变成指向CDN服务器的地址
    - 静态的JavaScript、CSS、图片等文件开启CDN和缓存，并且文件名带上HASH值
    - 为了并行加载不阻塞，把不同的静态资源分配到不同的CDN服务器上
- 域名限制
    - 同一时刻针对同一个域名的资源并行请求是有限制
    - 可以把这些静态资源分散到不同的 CDN 服务上去
    - 多个域名后会增加域名解析时间
    - 可以通过在 HTML HEAD 标签中 加入去预解析域名，以降低域名解析带来的延迟

** 7.9.1 webpack.config.js #**

```
const path = require("path");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const PurgecssPlugin = require("purgecss-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const UploadPlugin = require("./plugins/UploadPlugin");
const glob = require("glob");
const PATHS = {
    src: path.join(__dirname, "src"),
};
module.exports = {
  mode: "development",
  devtool: false,
  context: process.cwd(),
  entry: {
    main: "./src/index.js",
  },
  output: {
    path: path.resolve(__dirname, "dist"),
+    filename: "[name].[hash].js",
+    chunkFilename: "[name].[hash].chunk.js",
+    publicPath: "http://img.zhufengpeixun.cn/",
  },
  optimization: {
    minimize: true,
    minimizer: [
      //压缩JS
      /* new TerserPlugin({
        sourceMap: false,
        extractComments: false,
      }),
      //压缩CSS
      new OptimizeCSSAssetsPlugin({}), */
    ],
    //自动分割第三方模块和公共模块
    splitChunks: {
      chunks: "all", //默认作用于异步chunk，值为all/initial/async
      minSize: 0, //默认值是30kb,代码块的最小尺寸
      minChunks: 1, //被多少模块共享,在分割之前模块的被引用次数
      maxAsyncRequests: 2, //限制异步模块内部的并行最大请求数的，说白了你可以理解为是每个import()它里面的最大并行请求数量
      maxInitialRequests: 4, //限制入口的拆分数量
      name: true, //打包后的名称，默认是chunk的名字通过分隔符（默认是~）分隔开，如vendor~
      automaticNameDelimiter: "~", //默认webpack将会使用入口名和代码块的名称生成命名,比如 'vendors~main.js'
      cacheGroups: {
        //设置缓存组用来抽取满足不同规则的chunk,下面以生成common为例
        vendors: {
          chunks: "all",
          test: /node_modules/, //条件
          priority: -10, ///优先级，一个chunk很可能满足多个缓存组，会被抽取到优先级高的缓存组中,为了能够让自定义缓存组有更高的优先级(默认0),默认缓存组的priority属性为负值.

        },
        commons: {
          chunks: "all",
          minSize: 0, //最小提取字节数
          minChunks: 2, //最少被几个chunk引用
          priority: -20,
          reuseExistingChunk: true, //如果该chunk中引用了已经被抽取的chunk，直接引用该chunk，不会重复打包代码
        },
      },
    },
```

```
    //为了长期缓存保持运行时代码块是单独的文件
    /*  runtimeChunk: {
      name: (entrypoint) => `runtime-${entrypoint.name}`,
    }, */
  },
  module: {
    rules: [
      {
        test: /\.js/,
        include: path.resolve(__dirname, "src"),
        use: [
          {
            loader: "babel-loader",
            options: {
              presets: [
                ["@babel/preset-env", { modules: false }],
                "@babel/preset-react",
              ],
            },
          },
        ],
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      },
      {
        test: /\.(png|svg|jpg|gif|jpeg|ico)$/,
        use: [
          "file-loader",
          {
            loader: "image-webpack-loader",
            options: {
              mozjpeg: {
                progressive: true,
                quality: 65,
              },
              optipng: {
                enabled: false,
              },
              pngquant: {
                quality: "65-90",
                speed: 4,
              },
              gifsicle: {
                interlaced: false,
              },
              webp: {
                quality: 75,
              },
            },
          },
        ],
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      inject: true,
      template: "./src/index.html",
    }),
    new MiniCssExtractPlugin({
+      filename: "[name].[hash].css",
    }),
    new PurgecssPlugin({
      paths: glob.sync(`${PATHS.src}/**/*`, { nodir: true }),
    }),
    new UploadPlugin({}),
  ],
  devServer: {},
};
```

** 7.9.2 UploadPlugin.js#**

```
const qiniu = require("qiniu");
const path = require("path");

require("dotenv").config();
const defaultAccessKey = process.env.accessKey;
const defaultSecretKey = process.env.secretKey;
class UploadPlugin {
  constructor(options) {
    this.options = options || {};
  }
  apply(compiler) {
    compiler.hooks.afterEmit.tap("UploadPlugin", (compilation) => {
      let assets = compilation.assets;
      let promises = Object.keys(assets).filter(item=>!item.includes('.html')).map(this.upload.bind(this));
      Promise.all(promises).then((err, data) => console.log(err, data));
    });
  }
  upload(filename) {
    return new Promise((resolve, reject) => {
      let {
        bucket = "cnpmjs",
        accessKey = defaultAccessKey,
        secretKey = defaultSecretKey,
      } = this.options;
      let mac = new qiniu.auth.digest.Mac(accessKey, secretKey);
      let options = {
        scope: bucket,
      };

      let putPolicy = new qiniu.rs.PutPolicy(options);
      let uploadToken = putPolicy.uploadToken(mac);
      let config = new qiniu.conf.Config();

      let localFile = path.resolve(__dirname, "../dist", filename);
      let formUploader = new qiniu.form_up.FormUploader(config);
      let putExtra = new qiniu.form_up.PutExtra();
      formUploader.putFile(
        uploadToken,
        filename,
        localFile,
        putExtra,
        (err, body, info) => {
          err ? reject(err) : resolve(body);
        }
      );
    });
  }
}

module.exports = UploadPlugin;
```

8. webpack中hash、chunkhash、contenthash区别？#

- 文件指纹
- 打包后输出的文件名和后缀
- hash一般是结合CDN缓存来使用，通过webpack构建之后，生成对应文件名自动带上对应的MD5值。如果文件内容改变的话，那么对应文件哈希值也会改变，对应的HTML引用的URL地址也会改变，触发CDN服务器从源服务器上拉取对应数据，进而更新本地缓存。

指纹占位符

占位符名称 含义 ext 资源后缀名 name 文件名称 path 文件的相对路径 folder 文件所在的文件夹 hash 每次webpack构建时生成一个唯一的hash值 chunkhash 根据chunk生成hash值，来源于同一个chunk，则hash值就一样 contenthash 根据内容生成hash值，文件内容相同hash值就相同

8.1 hash

- Hash 是整个项目的hash值，其根据每次编译内容计算得到，每次编译之后都会生成新的hash,即修改任何文件都会导致所有文件的hash发生改变

```
const path = require("path");
const glob = require("glob");
const PurgecssPlugin = require("purgecss-webpack-plugin");
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const PATHS = {
  src: path.join(__dirname, 'src')
}
module.exports = {
  mode: "production",
+  entry: {
+    main: './src/index.js',
+    vender:['lodash']
+  },
  output:{
    path:path.resolve(__dirname,'dist'),
+    filename:'[name].[hash].js'
  },
  devServer:{
    hot:false
  },
  module: {
    rules: [
      {
        test: /\.js/,
        include: path.resolve(__dirname, "src"),
        use: [
          {
            loader:'thread-loader',
            options:{
              workers:3
            }
          },
          {
            loader: "babel-loader",
            options: {
              presets: ["@babel/preset-env", "@babel/preset-react"],
            },
          },
        ],
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
+      filename: "[name].[hash].css"
    }),
    new PurgecssPlugin({
      paths: glob.sync(`${PATHS.src}/**/*`,  { nodir: true }),
    }),
  ],
};
```

8.2 chunkhash

- chunkhash 采用hash计算的话，每一次构建后生成的哈希值都不一样，即使文件内容压根没有改变。这样子是没办法实现缓存效果，我们需要换另一种哈希值计算方式，即chunkhash
- chunkhash和hash不一样，它根据不同的入口文件(Entry)进行依赖文件解析、构建对应的chunk，生成对应的哈希值。我们在生产环境里把一些公共库和程序入口文件区分开，单独打包构建，接着我们采用chunkhash的方式生成哈希值，那么只要我们不改动公共库的代码，就可以保证其哈希值不会受影响

```
const path = require("path");
const glob = require("glob");
const PurgecssPlugin = require("purgecss-webpack-plugin");
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const PATHS = {
  src: path.join(__dirname, 'src')
}
module.exports = {
  mode: "production",
  entry: {
    main: './src/index.js',
    vender:['lodash']
  },
  output:{
    path:path.resolve(__dirname,'dist'),
+    filename:'[name].[chunkhash].js'
  },
  devServer:{
    hot:false
  },
  module: {
    rules: [
      {
        test: /\.js/,
        include: path.resolve(__dirname, "src"),
        use: [
          {
            loader:'thread-loader',
            options:{
              workers:3
            }
          },
          {
            loader: "babel-loader",
            options: {
              presets: ["@babel/preset-env", "@babel/preset-react"],
            },
          },
        ],
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
+      filename: "[name].[chunkhash].css"
    }),
    new PurgecssPlugin({
      paths: glob.sync(`${PATHS.src}/**/*`,  { nodir: true }),
    }),
  ],
};
```

8.3 contenthash

- 使用chunkhash存在一个问题，就是当在一个JS文件中引入CSS文件，编译后它们的hash是相同的，而且只要js文件发生改变 ， 关联的css文件hash也会改变,这个时候可以使用 `mini-css-extract-plugin` 里的 `contenthash`值，保证即使css文件所处的模块里就算其他文件内容改变，只要css文件内容不变，那么不会重复构建

```
const path = require("path");
const glob = require("glob");
const PurgecssPlugin = require("purgecss-webpack-plugin");
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const PATHS = {
  src: path.join(__dirname, 'src')
}
module.exports = {
  mode: "production",
  entry: {
    main: './src/index.js',
    vender:['lodash']
  },
  output:{
    path:path.resolve(__dirname,'dist'),
    filename:'[name].[chunkhash].js'
  },
  devServer:{
    hot:false
  },
  module: {
    rules: [
      {
        test: /\.js/,
        include: path.resolve(__dirname, "src"),
        use: [
          {
            loader:'thread-loader',
            options:{
              workers:3
            }
          },
          {
            loader: "babel-loader",
            options: {
              presets: ["@babel/preset-env", "@babel/preset-react"],
            },
          },
        ],
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
+      filename: "[name].[contenthash].css"
    }),
    new PurgecssPlugin({
      paths: glob.sync(`${PATHS.src}/**/*`,  { nodir: true }),
    }),
  ],
};
```

9.如何对bundle体积进行监控和分析？

- 是一个webpack的插件，需要配合webpack和webpack-cli一起使用。这个插件的功能是生成代码分析报告，帮助提升代码质量和网站性能

9.1 安装

```
cnpm i webpack-bundle-analyzer -D
```

```
const {BundleAnalyzerPlugin} = require('webpack-bundle-analyzer')
module.exports={
  plugins: [
    new BundleAnalyzerPlugin()

  ]
}
```

```
{
 "scripts": {
    "dev": "webpack --config webpack.dev.js --progress"
  }
}
```

9.2 先生成文件再分析

webpack.config.js

```
const {BundleAnalyzerPlugin} = require('webpack-bundle-analyzer')
module.exports={
  plugins: [
    new BundleAnalyzerPlugin({
      analyzerMode: 'disabled',
      generateStatsFile: true,
    }),
  ]
}
```

```
{
 "scripts": {
    "generateAnalyzFile": "webpack --profile --json > stats.json",
    "analyz": "webpack-bundle-analyzer --port 8888 ./dist/stats.json"
  }
}
```

```
npm run generateAnalyzFile
npm run analyz
```

10.如何提高webpack的构建速度？#

10.1 费时分析

```
const SpeedMeasureWebpackPlugin = require('speed-measure-webpack-plugin');
const smw = new SpeedMeasureWebpackPlugin();
module.exports =smw.wrap({
});
```

10.2 缩小范围

** 10.2.1 extensions #**

指定extension之后可以不用在 require或是 import的时候加文件扩展名,会依次尝试添加扩展名进行匹配

```
resolve: {
  extensions: [".js",".jsx",".json",".css"]
},
```

** 10.2.2 alias #**

配置别名可以加快webpack查找模块的速度

- 每当引入bootstrap模块的时候，它会直接引入 bootstrap,而不需要从 node_modules文件夹中按模块的查找规则查找

```
const bootstrap = path.resolve(__dirname,'node_modules/_bootstrap@3.3.7@bootstrap/dist/css/bootstrap.css');
resolve: {
+    alias:{
+        "bootstrap":bootstrap
+    }
},
```

** 10.2.3 modules #**

- 对于直接声明依赖名的模块（如 react ），webpack 会类似 Node.js 一样进行路径搜索，搜索 node_modules目录
- 这个目录就是使用 resolve.modules字段进行配置的 默认配置

```
resolve: {
modules: ['node_modules'],
}
```

如果可以确定项目内所有的第三方依赖模块都是在项目根目录下的 node_modules 中的话

```
resolve: {
modules: [path.resolve(__dirname, 'node_modules')],
}
```

** 10.2.4 mainFields #**

默认情况下package.json 文件则按照文件中 main 字段的文件名来查找文件

```
resolve: {

  mainFields: ['browser', 'module', 'main'],

  mainFields: ["module", "main"],
}
```

** 10.2.5 mainFiles #**

当目录下没有 package.json 文件时，我们说会默认使用目录下的 index.js 这个文件，其实这个也是可以配置的

```
resolve: {
  mainFiles: ['index'],
},
```

** 10.2.6 resolveLoader #**
resolve.resolveLoader用于配置解析 loader 时的 resolve 配置,默认的配置：

```
module.exports = {
  resolveLoader: {
    modules: [ 'node_modules' ],
    extensions: [ '.js', '.json' ],
    mainFields: [ 'loader', 'main' ]
  }
};
```

10.3 noParse

- module.noParse 字段，可以用于配置哪些模块文件的内容不需要进行解析
- 不需要解析依赖（即无依赖） 的第三方大型类库等，可以通过这个字段来配置，以提高整体的构建速度

```
module.exports = {

module: {
  noParse: /jquery|lodash/,

  noParse(content) {
    return /jquery|lodash/.test(content)
  },
}
}...
```

> 使用 noParse 进行忽略的模块文件中不能使用 import、require、define 等导入机制

10.4 IgnorePlugin

IgnorePlugin用于忽略某些特定的模块，让 webpack 不把这些指定的模块打包进去

```
import moment from  'moment';
console.log(moment);
```

```
new webpack.IgnorePlugin(/^\.\/locale/,/moment$/)
```

- 第一个是匹配引入模块路径的正则表达式
- 第二个是匹配模块的对应上下文，即所在目录名

10.5 日志优化

- 日志太多太少都不美观
- 可以修改stats

预设 替代 描述 errors-only none 只在错误时输出 minimal none 发生错误和新的编译时输出 none false 没有输出 normal true 标准输出 verbose none 全部输出
** 10.5.1 friendly-errors-webpack-plugin #**

- friendly-errors-webpack-plugin (https://www.npmjs.com/package/friendly-errors-webpack-plugin)
- success 构建成功的日志提示
- warning 构建警告的日志提示
- error 构建报错的日志提示

```
cnpm i friendly-errors-webpack-plugin
```

```
+ stats:'verbose',
  plugins:[
+   new FriendlyErrorsWebpackPlugin()
  ]
```

> 编译完成后可以通过 `echo $?`获取错误码，0为成功，非0为失败

10.6.DLL

- .dll为后缀的文件称为动态链接库，在一个动态链接库中可以包含给其他模块调用的函数和数据
- 把基础模块独立出来打包到单独的动态连接库里
- 当需要导入的模块在动态连接库里的时候，模块不能再次被打包，而是去动态连接库里获取
- dll-plugin (https://webpack.js.org/plugins/dll-plugin/)

** 10.6.1 定义Dll #**

- DllPlugin插件： 用于打包出一个个动态连接库
- DllReferencePlugin: 在配置文件中引入DllPlugin插件打包好的动态连接库

webpack.dll.config.js

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

const DllPlugin = require("webpack/lib/DllPlugin");
module.exports = {
  mode: "development",
  entry: {
    react: ["react", "react-dom"],
  },
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "[name].dll.js",
    library: "_dll_[name]",
  },
  plugins: [
    new DllPlugin({
      name: "_dll_[name]",
      path: path.join(__dirname, "dist", "[name].manifest.json"),
    }),
  ],
};
```

```
webpack --config webpack.dll.config.js --mode=development
```

** 10.6.2 使用动态链接库文件 #**

```
const path = require("path");
const glob = require("glob");
const PurgecssPlugin = require("purgecss-webpack-plugin");
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
+const DllReferencePlugin = require("webpack/lib/DllReferencePlugin.js");
const PATHS = {
  src: path.join(__dirname, 'src')
}
module.exports = {
  mode: "development",
  entry: "./src/index.js",
  module: {
    rules: [
      {
        test: /\.js/,
        include: path.resolve(__dirname, "src"),
        use: [
          {
            loader: "babel-loader",
            options: {
              presets: ["@babel/preset-env", "@babel/preset-react"],
            },
          },
        ],
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: "[name].css",
    }),
    new PurgecssPlugin({
      paths: glob.sync(`${PATHS.src}/**/*`,  { nodir: true }),
    }),
+    new DllReferencePlugin({
+      manifest: require("./dist/react.manifest.json"),
+    }),
  ],
};
```

```
webpack --config webpack.config.js --mode development
```

** 10.6.3 html中使用 #**

```
<script src="react.dll.js">script>
```

10.7 利用缓存

- webpack中利用缓存一般有以下几种思路：
    - babel-loader开启缓存
    - 使用cache-loader
    - 使用hard-source-webpack-plugin

** 10.7.1 babel-loader #**

- Babel在转义js文件过程中消耗性能较高，将babel-loader执行的结果缓存起来，当重新打包构建时会尝试读取缓存，从而提高打包构建速度、降低消耗

```
{
   test: /\.js$/,
   exclude: /node_modules/,
   use: [{
     loader: "babel-loader",
     options: {
       cacheDirectory: true
     }
   }]
 },
```

** 10.7.2 cache-loader #**

- 在一些性能开销较大的 loader 之前添加此 loader,以将结果缓存到磁盘里
- 存和读取这些缓存文件会有一些时间开销,所以请只对性能开销较大的 loader 使用此 loader

```
cnpm i  cache-loader -D
```

```
const loaders = ['babel-loader'];
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          'cache-loader',
          ...loaders
        ],
        include: path.resolve('src')
      }
    ]
  }
}
```

** 10.7.3 hard-source-webpack-plugin #**

- HardSourceWebpackPlugin为模块提供了中间缓存,缓存默认的存放路径是 node_modules/.cache/hard-source。`
- 配置 hard-source-webpack-plugin后，首次构建时间并不会有太大的变化，但是从第二次开始，构建时间大约可以减少 80%左右
- webpack5中会内置 hard-source-webpack-plugin

```
cnpm i  hard-source-webpack-plugin -D
```

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

module.exports = {
  entry:
  output:
  plugins: [
    new HardSourceWebpackPlugin()
  ]
}
```

** 10.7.4 oneOf #**

- 每个文件对于rules中的所有规则都会遍历一遍，如果使用oneOf就可以解决该问题，只要能匹配一个即可退出。(注意：在oneOf中不能两个配置处理同一种类型文件)

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,

        enforce: 'pre',
        loader: 'eslint-loader',
        options: {
          fix: true
        }
      },
      {

        oneOf: [
          ...,
          {},
          {}
        ]
      }
    ]
  }
}
```

10.8 多进程处理

** 10.8.1 thread-loader #**

- 把这个 loader 放置在其他 loader 之前， 放置在这个 loader 之后的 loader 就会在一个单独的 worker 池(worker pool)中运行
- thread-loader (https://webpack.js.org/loaders/thread-loader/)

```
cnpm  i thread-loader- D
```

```
const path = require("path");
const glob = require("glob");
const PurgecssPlugin = require("purgecss-webpack-plugin");
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const DllReferencePlugin = require("webpack/lib/DllReferencePlugin.js");
const PATHS = {
  src: path.join(__dirname, 'src')
}
module.exports = {
  mode: "development",
  entry: "./src/index.js",
  module: {
    rules: [
      {
        test: /\.js/,
        include: path.resolve(__dirname, "src"),
        use: [
+          {
+            loader:'thread-loader',
+            options:{
+              workers:3
+            }
+          },
          {
            loader: "babel-loader",
            options: {
              presets: ["@babel/preset-env", "@babel/preset-react"],
            },
          },
        ],
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "src"),
        exclude: /node_modules/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
          },
          "css-loader",
        ],
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: "[name].css",
    }),
    new PurgecssPlugin({
      paths: glob.sync(`${PATHS.src}/**/*`,  { nodir: true }),
    }),
    new DllReferencePlugin({
      manifest: require("./dist/react.manifest.json"),
    }),
  ],
};
```

** 10.8.2 parallel #**

terser-webpack-plugin 开启 parallel 参数

```
const TerserPlugin = require('terser-webpack-plugin');
module.exports = {
    optimization: {
        minimizer: [
            new TerserPlugin({
                parallel: true,
            }),
        ],
    },
};
```