这里讲述文件的配置方式，一般情况，通过配置文件导出对象中 `plugins` 属性传入 `new` 实例对象。如下所示：

JavaScript | 复制代码

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); // 通过 npm 安装
const webpack = require('webpack'); // 访问内置的插件
module.exports = {
  ...
  plugins: [
    new webpack.ProgressPlugin(),
    new HtmlWebpackPlugin({ template: './src/index.html' }),
  ],
};
```

6.2. 特性

其本质是一个具有 `apply` 方法 `javascript` 对象

`apply` 方法会被 `webpack compiler` 调用，并且在整个编译生命周期都可以访问 `compiler` 对象

JavaScript | 复制代码

```
const pluginName = 'ConsoleLogOnBuildWebpackPlugin';

class ConsoleLogOnBuildWebpackPlugin {
  apply(compiler) {
    compiler.hooks.run.tap(pluginName, (compilation) => {
      console.log('webpack 构建过程开始! ');
    });
  }
}

module.exports = ConsoleLogOnBuildWebpackPlugin;
```

`compiler hook` 的 `tap` 方法的第一个参数，应是驼峰式命名的插件名称

关于整个编译生命周期钩子，有如下：

- entry-option ： 初始化 option
- run
- compile： 真正开始的编译，在创建 compilation 对象之前
- compilation ： 生成好了 compilation 对象

- make 从 entry 开始递归分析依赖，准备对每个模块进行 build
- after-compile： 编译 build 过程结束
- emit ： 在将内存中 assets 内容写到磁盘文件夹之前
- after-emit ： 在将内存中 assets 内容写到磁盘文件夹之后
- done： 完成所有的编译过程
- failed： 编译失败的时候

6.3. 三、常见的Plugin

常见的 `plugin` 有如图所示：

AggressiveSplittingPlugin	将原来的 chunk 分成更小的 chunk
BabelMinifyWebpackPlugin	使用 babel-minify进行压缩
BannerPlugin	在每个生成的 chunk 顶部添加 banner
CommonsChunkPlugin	提取 chunks 之间共享的通用模块
CompressionWebpackPlugin	预先准备的资源压缩版本，使用 Content-Encoding 提供访问服务
ContextReplacementPlugin	重写 `require` 表达式的推断上下文
CopyWebpackPlugin	将单个文件或整个目录复制到构建目录
DefinePlugin	允许在编译时(compile time)配置的全局常量
DllPlugin	为了极大减少构建时间，进行分离打包
EnvironmentPlugin	`DefinePlugin` 中 `process.env` 键的简写方式。
ExtractTextWebpackPlugin	从 bundle 中提取文本（CSS）到单独的文件
HotModuleReplacementPlugin	启用模块热替换(Enable Hot Module Replacement - HMR)
HtmlWebpackPlugin	简单创建 HTML 文件，用于服务器访问
I18nWebpackPlugin	为 bundle 增加国际化支持
IgnorePlugin	从 bundle 中排除某些模块
LimitChunkCountPlugin	设置 chunk 的最小/最大限制，以微调和控制 chunk
LoaderOptionsPlugin	用于从 webpack 1 迁移到 webpack 2
MinChunkSizePlugin	确保 chunk 大小超过指定限制
NoEmitOnErrorsPlugin	在输出阶段时，遇到编译错误跳过
NormalModuleReplacementPlugin	替换与正则表达式匹配的资源

下面介绍几个常用的插件用法：

6.3.1. HtmlWebpackPlugin

在打包结束后，自动生成一个 `html` 文文件，并把打包生成的 `js` 模块引入到该 `html` 中

Bash 复制代码

```
npm install --save-dev html-webpack-plugin
```

JavaScript 复制代码

```
// webpack.config.js
const HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = {
 ...
  plugins: [
     new HtmlWebpackPlugin({
       title: "My App",
       filename: "app.html",
       template: "./src/html/index.html"
     })
  ]
};
```

HTML 复制代码

```
<!--./src/html/index.html-->
<!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><%=htmlWebpackPlugin.options.title%></title>
</head>
<body>
    <h1>html-webpack-plugin</h1>
</body>
</html>
```

在 `html` 模板中，可以通过 `<%=htmlWebpackPlugin.options.XXX%>` 的方式获取配置的值

更多的配置可以自寻查找

6.3.2. clean-webpack-plugin

删除（清理）构建目录

Bash 复制代码

```
npm install --save-dev clean-webpack-plugin
```

JavaScript 复制代码

```
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
module.exports = {
 ...
  plugins: [
    ...,
    new CleanWebpackPlugin(),
    ...
  ]
}
```

6.3.3. mini-css-extract-plugin

提取 `CSS` 到一个单独的文件中

Bash 复制代码

```
npm install --save-dev mini-css-extract-plugin
```

```JavaScript
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
module.exports = {
 ...,
  module: {
   rules: [
    {
     test: /\.s[ac]ss$/,
     use: [
      {
       loader: MiniCssExtractPlugin.loader
      },
           'css-loader',
           'sass-loader'
         ]
    }
    ]
  },
   plugins: [
     ...,
     new MiniCssExtractPlugin({
      filename: '[name].css'
     }),
     ...
   ]
}
```

6.3.4. DefinePlugin

允许在编译时创建配置的全局对象，是一个 `webpack` 内置的插件，不需要安装

```JavaScript
const { DefinePlugun } = require('webpack')

module.exports = {
 ...
    plugins:[
        new DefinePlugin({
            BASE_URL:'"./"'
        })
    ]
}
```

这时候编译 `template` 模块的时候，就能通过下述形式获取全局对象

```HTML
<link rel="icon" href="<%= BASE_URL%>favicon.ico>"
```

6.3.5. copy-webpack-plugin

复制文件或目录到执行区域，如 `vue` 的打包过程中，如果我们将一些文件放到 `public` 的目录下，那么这个目录会被复制到 `dist` 文件夹中

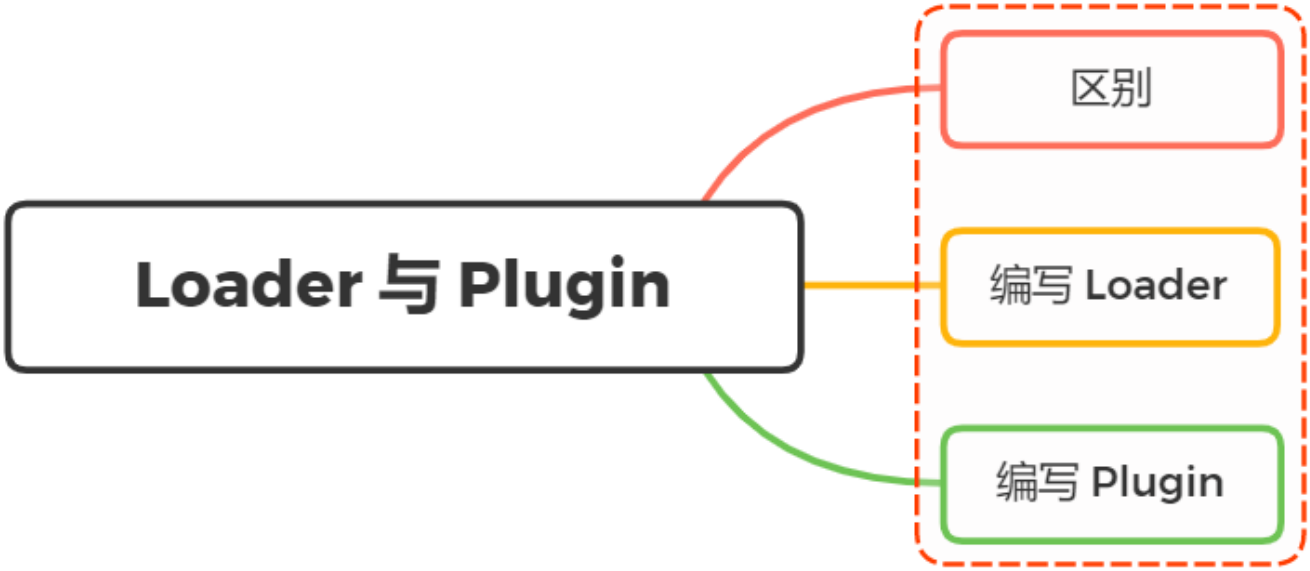
```Plain Text
npm install copy-webpack-plugin -D
```

```JavaScript
new CopyWebpackPlugin({
    parrerns:[
        {
            from:"public",
            globOptions:{
                ignore:[
                    '**/index.html'
                ]
            }
        }
    ]
})
```

复制的规则在 `patterns` 属性中设置：

- from：设置从哪一个源中开始复制
- to：复制到的位置，可以省略，会默认复制到打包的目录下
- globOptions：设置一些额外的选项，其中可以编写需要忽略的文件

7. 说说Loader和Plugin的区别？编写Loader，Plugin的思路？

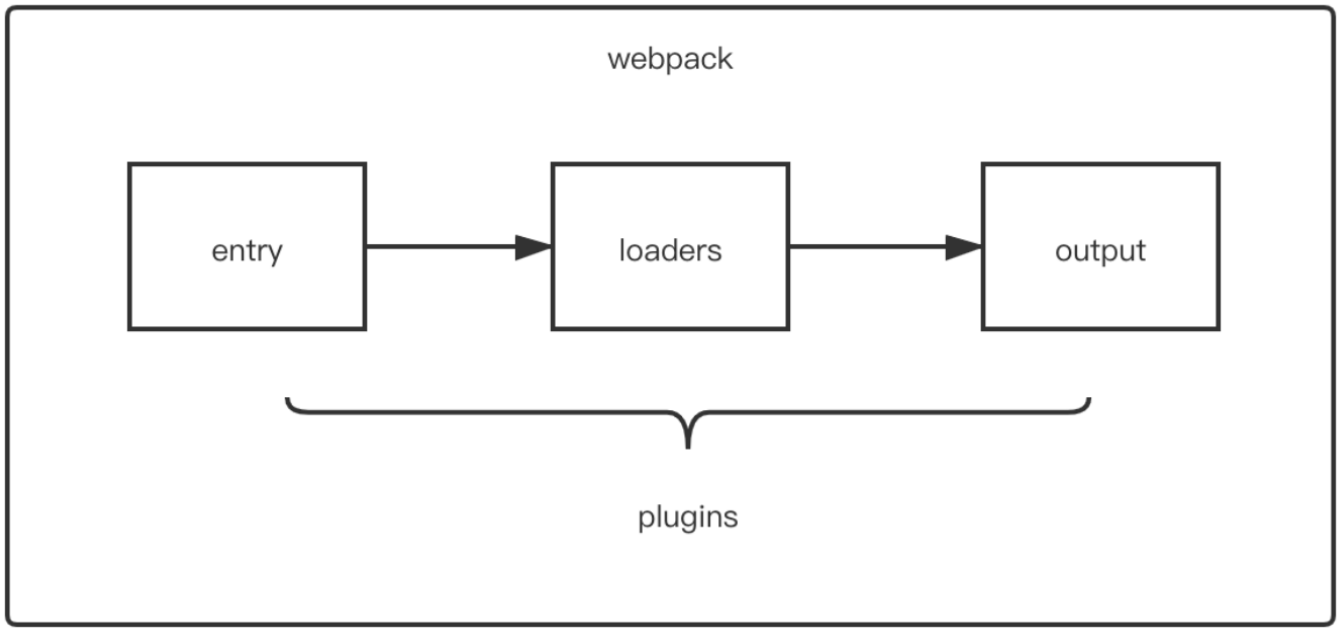


7.1. 区别

前面两节我们有提到 `Loader` 与 `Plugin` 对应的概念，先来回顾下

- loader 是文件加载器，能够加载资源文件，并对这些文件进行一些处理，诸如编译、压缩等，最终一起打包到指定的文件中
- plugin 赋予了 webpack 各种灵活的功能，例如打包优化、资源管理、环境变量注入等，目的是解决 loader 无法实现的其他事

从整个运行时机上来看，如下图所示：



可以看到，两者在运行时机上的区别：

- loader 运行在打包文件之前
- plugins 在整个编译周期都起作用

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

对于 `loader` ，实质是一个转换器，将A文件进行编译形成B文件，操作的是文件，比如将 `A.scss` 或 `A.less` 转变为 `B.css` ，单纯的文件转换过程

7.2. 编写loader

在编写 `loader` 前，我们首先需要了解 `loader` 的本质

其本质为函数，函数中的 `this` 作为上下文会被 `webpack` 填充，因此我们不能将 `loader` 设为一个箭头函数

函数接受一个参数，为 `webpack` 传递给 `loader` 的文件源内容

函数中 `this` 是由 `webpack` 提供的对象，能够获取当前 `loader` 所需要的各种信息

函数中有异步操作或同步操作，异步操作通过 `this.callback` 返回，返回值要求为 `string` 或者 `Buffer`

代码如下所示：

```JavaScript
// 导出一个函数，source为webpack传递给loader的文件源内容
module.exports = function(source) {
    const content = doSomeThing2JsString(source);

    // 如果 loader 配置了 options 对象，那么this.query将指向 options
    const options = this.query;

    // 可以用作解析其他模块路径的上下文
    console.log('this.context');

    /*
     * this.callback 参数：
     * error：Error | null，当 loader 出错时向外抛出一个 error
     * content：String | Buffer，经过 loader 编译后需要导出的内容
     * sourceMap：为方便调试生成的编译后内容的 source map
     * ast：本次编译生成的 AST 静态语法树，之后执行的 loader 可以直接使用这个 AST，进而省去重复生成 AST 的过程
     */
    this.callback(null, content); // 异步
    return content; // 同步
}
```

一般在编写 `loader` 的过程中，保持功能单一，避免做多种功能

如 `less` 文件转换成 `css` 文件也不是一步到位，而是 `less-loader` 、 `css-loader` 、 `style-loader` 几个 `loader` 的链式调用才能完成转换

7.3. 编写plugin

由于 `webpack` 基于发布订阅模式，在运行的生命周期中会广播出许多事件，插件通过监听这些事件，就可以在特定的阶段执行自己的插件任务

在之前也了解过， `webpack` 编译会创建两个核心对象：

- compiler：包含了 webpack 环境的所有的配置信息，包括 options，loader 和 plugin，和 webpack 整个生命周期相关的钩子
- compilation：作为 plugin 内置事件回调函数的参数，包含了当前的模块资源、编译生成资源、变化的文件以及被跟踪依赖的状态信息。当检测到一个文件变化，一次新的 Compilation 将被创建

如果自己要实现 `plugin` ，也需要遵循一定的规范：

- 插件必须是一个函数或者是一个包含 `apply` 方法的对象，这样才能访问 `compiler` 实例

- 传给每个插件的 `compiler` 和 `compilation` 对象都是同一个引用，因此不建议修改
- 异步的事件需要在插件处理完任务时调用回调函数通知 `Webpack` 进入下一个流程，不然会卡住

实现 `plugin` 的模板如下：

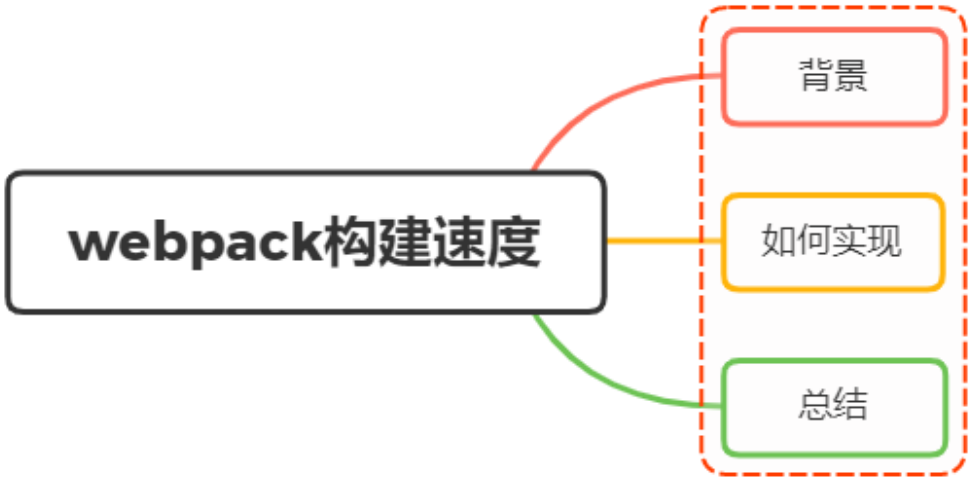
JavaScript | 复制代码

```
class MyPlugin {
    // Webpack 会调用 MyPlugin 实例的 apply 方法给插件实例传入 compiler 对象
  apply (compiler) {
    // 找到合适的事件钩子，实现自己的插件功能
    compiler.hooks.emit.tap('MyPlugin', compilation => {
        // compilation: 当前打包构建流程的上下文
        console.log(compilation);

        // do something...
    })
  }
}
```

在 `emit` 事件发生时，代表源文件的转换和组装已经完成，可以读取到最终将输出的资源、代码块、模块及其依赖，并且可以修改输出资源的内容

8. 如何提高webpack的构建速度?



8.1. 背景

随着我们的项目涉及到页面越来越多，功能和业务代码也会随着越多，相应的 `webpack` 的构建时间也会越来越久

构建时间与我们日常开发效率密切相关，当我们本地开发启动 `devServer` 或者 `build` 的时候，如果时间过长，会大大降低我们的工作效率

所以，优化 `webpack` 构建速度是十分重要的环节

8.2. 如何优化

常见的提升构建速度的手段有如下：

- 优化 loader 配置
- 合理使用 resolve.extensions
- 优化 resolve.modules
- 优化 resolve.alias
- 使用 DLLPlugin 插件
- 使用 cache-loader
- terser 启动多线程
- 合理使用 sourceMap

8.2.1. 优化loader配置

在使用 `loader` 时，可以通过配置 `include` 、 `exclude` 、 `test` 属性来匹配文件，接触 `include` 、 `exclude` 规定哪些匹配应用 `loader`

如采用 ES6 的项目为例，在配置 `babel-loader` 时，可以这样：

```JavaScript
module.exports = {
  module: {
    rules: [
      {
        // 如果项目源码中只有 js 文件就不要写成 /\.jsx?$/，提升正则表达式性能
        test: /\.js$/,
        // babel-loader 支持缓存转换出的结果，通过 cacheDirectory 选项开启
        use: ['babel-loader?cacheDirectory'],
        // 只对项目根目录下的 src 目录中的文件采用 babel-loader
        include: path.resolve(__dirname, 'src'),
      },
    ]
  },
};
```

8.2.2. 合理使用 resolve.extensions

在开发中我们会有各种各样的模块依赖，这些模块可能来自于自己编写的代码，也可能来自第三方库，`resolve` 可以帮助 `webpack` 从每个 `require/import` 语句中，找到需要引入到合适的模块代码

通过 `resolve.extensions` 是解析到文件时自动添加拓展名，默认情况如下：

```JavaScript
module.exports = {
    ...
    extensions:[".warm",".mjs",".js",".json"]
}
```

当我们引入文件的时候，若没有文件后缀名，则会根据数组内的值依次查找

当我们配置的时候，则不要随便把所有后缀都写在里面，这会调用多次文件的查找，这样就会减慢打包速度

8.2.3. 优化 resolve.modules

`resolve.modules` 用于配置 `webpack` 去哪些目录下寻找第三方模块。默认值为 `['node_modules']`，所以默认会从 `node_modules` 中查找文件

当安装的第三方模块都放在项目根目录下的 `./node_modules` 目录下时，所以可以指明存放第三方模块的绝对路径，以减少寻找，配置如下：

```plain text
module.exports = {
  resolve: {
    // 使用绝对路径指明第三方模块存放的位置，以减少搜索步骤
    // 其中 __dirname 表示当前工作目录，也就是项目根目录
    modules: [path.resolve(__dirname, 'node_modules')]
  },
};
```

8.2.4. 优化 resolve.alias

`alias` 给一些常用的路径起一个别名，特别当我们的项目目录结构比较深的时候，一个文件的路径可能是 `./../../` 的形式

通过配置 `alias` 以减少查找过程

```javascript
module.exports = {
    ...
    resolve:{
        alias:{
            "@":path.resolve(__dirname,'./src')
        }
    }
}
```

8.2.5. 使用 DLLPlugin 插件

`DLL` 全称是 动态链接库，是为软件在winodw种实现共享函数库的一种实现方式，而Webpack也内置了DLL的功能，为的就是可以共享，不经常改变的代码，抽成一个共享的库。这个库在之后的编译过程中，会被引入到其他项目的代码中

使用步骤分成两部分：

- 打包一个 DLL 库
- 引入 DLL 库

8.2.5.1. 打包一个 DLL 库

`webpack` 内置了一个 `DllPlugin` 可以帮助我们打包一个DLL的库文件

```JavaScript
module.exports = {
    ...
    plugins:[
        new webpack.DllPlugin({
            name:'dll_[name]',
            path:path.resolve(__dirname,"./dll/[name].mainfest.json")
        })
    ]
}
```

8.2.5.2. 引入 DLL 库

使用 `webpack` 自带的 `DllReferencePlugin` 插件对 `mainfest.json` 映射文件进行分析，获取要使用的 `DLL` 库

然后再通过 `AddAssetHtmlPlugin` 插件，将我们打包的 `DLL` 库引入到 `Html` 模块中

```JavaScript
module.exports = {
    ...
    new webpack.DllReferencePlugin({
        context:path.resolve(__dirname,"./dll/dll_react.js"),
        mainfest:path.resolve(__dirname,"./dll/react.mainfest.json")
    }),
    new AddAssetHtmlPlugin({
        outputPath:"./auto",
        filepath:path.resolve(__dirname,"./dll/dll_react.js")
    })
}
```

8.2.6. 使用 cache-loader

在一些性能开销较大的 `loader` 之前添加 `cache-loader`，以将结果缓存到磁盘里，显著提升二次构建速度

保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`

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

8.2.7. terser 启动多线程

使用多进程并行运行来提高构建速度

```javascript
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        parallel: true,
      }),
    ],
  },
};
```

8.2.8. 合理使用 sourceMap

打包生成 `sourceMap` 的时候，如果信息越详细，打包速度就会越慢。对应属性取值如下所示：

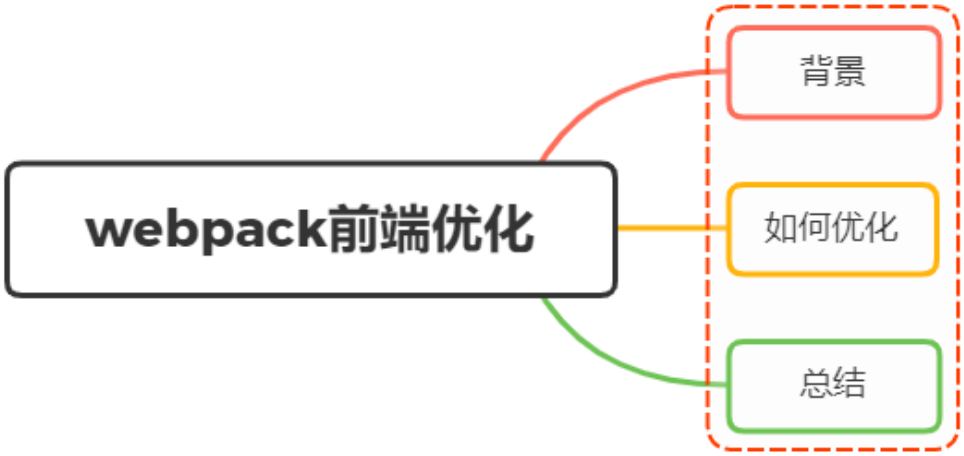
devtool	构建速度	重新构建速度	生产环境	品质(quality)
(none)	+++	+++	yes	打包后的代码
eval	+++	+++	no	生成后的代码
cheap-eval-source-map	+	++	no	转换过的代码（仅限行）
cheap-module-eval-source-map	o	++	no	原始源代码（仅限行）
eval-source-map	--	+	no	原始源代码
cheap-source-map	+	o	yes	转换过的代码（仅限行）
cheap-module-source-map	o	-	yes	原始源代码（仅限行）
inline-cheap-source-map	+	o	no	转换过的代码（仅限行）
inline-cheap-module-source-map	o	-	no	原始源代码（仅限行）
source-map	--	--	yes	原始源代码
inline-source-map	--	--	no	原始源代码
hidden-source-map	--	--	yes	原始源代码
nosources-source-map	--	--	yes	无源代码内容

+++ 非常快速 ++ 快速 + 比较快 o 中等 - 比较慢 -- 慢

8.3. 总结

可以看到，优化 `webpack` 构建的方式有很多，主要可以从优化搜索时间、缩小文件搜索范围、减少不必要的编译等方面入手

9. 说说如何借助webpack来优化前端性能?



9.1. 背景

随着前端的项目逐渐扩大，必然会带来的一个问题就是性能

尤其在大型复杂的项目中，前端业务可能因为一个小小的数据依赖，导致整个页面卡顿甚至奔溃

一般项目在完成后，会通过 `webpack` 进行打包，利用 `webpack` 对前端项目性能优化是一个十分重要的环节

9.2. 如何优化

通过 `webpack` 优化前端的手段有：

- JS代码压缩
- CSS代码压缩
- Html文件代码压缩
- 文件大小压缩
- 图片压缩
- Tree Shaking
- 代码分离
- 内联 chunk

9.2.1. JS代码压缩

`terser` 是一个 `JavaScript` 的解释、绞肉机、压缩机的工具集，可以帮助我们压缩、丑化我们的代码，让 `bundle` 更小

在 `production` 模式下， `webpack` 默认就是使用 `TerserPlugin` 来处理我们的代码的。如果想要自定义配置它，配置方法如下：

JavaScript | 复制代码

```
const TerserPlugin = require('terser-webpack-plugin')
module.exports = {
    ...
    optimization: {
        minimize: true,
        minimizer: [
            new TerserPlugin({
                parallel: true // 电脑cpu核数-1
            })
        ]
    }
}
```

属性介绍如下

- extractComments：默认值为true，表示会将注释抽取到一个单独的文件中，开发阶段，我们可设置为 false ，不保留注释
- parallel：使用多进程并发运行提高构建的速度，默认值是true，并发运行的默认数量：os.cpus().length – 1
- terserOptions：设置我们的terser相关的配置：
- compress：设置压缩相关的选项，mangle：设置丑化相关的选项，可以直接设置为true
- mangle：设置丑化相关的选项，可以直接设置为true
- toplevel：底层变量是否进行转换
- keep_classnames：保留类的名称
- keep_fnames：保留函数的名称

9.2.2. CSS代码压缩

`CSS` 压缩通常是去除无用的空格等，因为很难去修改选择器、属性的名称、值等

CSS的压缩我们可以使用另外一个插件： `css-minimizer-webpack-plugin`

Plain Text | 复制代码

```
npm install css-minimizer-webpack-plugin -D
```

配置方法如下：

```JavaScript
const CssMinimizerPlugin = require('css-minimizer-webpack-plugin')
module.exports = {
    // ...
    optimization: {
        minimize: true,
        minimizer: [
            new CssMinimizerPlugin({
                parallel: true
            })
        ]
    }
}
```

9.2.3. Html文件代码压缩

使用 `HtmlWebpackPlugin` 插件来生成 `HTML` 的模板时候，通过配置属性 `minify` 进行 `html` 优化

```JavaScript
module.exports = {
    ...
    plugin:[
        new HtmlwebpackPlugin({
            ...
            minify:{
                minifyCSS:false, // 是否压缩css
                collapseWhitespace:false, // 是否折叠空格
                removeComments:true // 是否移除注释
            }
        })
    ]
}
```

设置了 `minify` ，实际会使用另一个插件 `html-minifier-terser`

9.2.4. 文件大小压缩

对文件的大小进行压缩，减少 `http` 传输过程中宽带的损耗

JavaScript | 复制代码

```
npm install compression-webpack-plugin -D
```

JavaScript | 复制代码

```
new ComepressionPlugin({
    test:/\.(css|js)$/,  // 哪些文件需要压缩
    threshold:500, // 设置文件多大开始压缩
    minRatio:0.7, // 至少压缩的比例
    algorithm:"gzip", // 采用的压缩算法
})
```

9.2.5. 图片压缩

一般来说在打包之后，一些图片文件的大小是远远要比 `js` 或者 `css` 文件要来的大，所以图片压缩较为重要

配置方法如下：

JavaScript | 复制代码

```javascript
module: {
  rules: [
    {
      test: /\.(png|jpg|gif)$/,
      use: [
        {
          loader: 'file-loader',
          options: {
            name: '[name]_[hash].[ext]',
            outputPath: 'images/',
          }
        },
        {
          loader: 'image-webpack-loader',
          options: {
            // 压缩 jpeg 的配置
            mozjpeg: {
              progressive: true,
              quality: 65
            },
            // 使用 imagemin**-optipng 压缩 png, enable: false 为关闭
            optipng: {
              enabled: false,
            },
            // 使用 imagemin-pngquant 压缩 png
            pngquant: {
              quality: '65-90',
              speed: 4
            },
            // 压缩 gif 的配置
            gifsicle: {
              interlaced: false,
            },
            // 开启 webp, 会把 jpg 和 png 图片压缩为 webp 格式
            webp: {
              quality: 75
            }
          }
        }
      ]
    },
  ]
}
```

9.2.6. Tree Shaking

`Tree Shaking` 是一个术语，在计算机中表示消除死代码，依赖于 `ES Module` 的静态语法分析（不执行任何的代码，可以明确知道模块的依赖关系）

在 `webpack` 实现 `Trss shaking` 有两种不同的方案：

- usedExports：通过标记某些函数是否被使用，之后通过Terser来进行优化的
- sideEffects：跳过整个模块/文件，直接查看该文件是否有副作用

两种不同的配置方案， 有不同的效果

9.2.6.1. usedExports

配置方法也很简单，只需要将 `usedExports` 设为 `true`

JavaScript | 复制代码

```
module.exports = {
    ...
    optimization:{
        usedExports
    }
}
```

使用之后，没被用上的代码在 `webpack` 打包中会加入 `unused harmony export mul` 注释，用来告知 `Terser` 在优化时，可以删除掉这段代码

如下面 `sum` 函数没被用到， `webpack` 打包会添加注释， `terser` 在优化时，则将该函数去掉

```
/* unused harmony export mul */
function sum(num1, num2) {
  return num1 + num2;
}
```

9.2.6.2. sideEffects

`sideEffects` 用于告知 `webpack compiler` 哪些模块时有副作用，配置方法是在 `package.json` 中设置 `sideEffects` 属性

如果 `sideEffects` 设置为false，就是告知 `webpack` 可以安全的删除未用到的 `exports`

如果有些文件需要保留，可以设置为数组的形式

```JavaScript
"sideEffecis":[
    "./src/util/format.js",
    "*.css" // 所有的css文件
]
```

上述都是关于 `javascript` 的 `tree shaking`， `css` 同样也能够实现 `tree shaking`

9.2.6.3. css tree shaking

`css` 进行 `tree shaking` 优化可以安装 `PurgeCss` 插件

```Plain Text
npm install purgecss-plugin-webpack -D
```

```JavaScript
const PurgeCssPlugin = require('purgecss-webpack-plugin')
module.exports = {
    ...
    plugins:[
        new PurgeCssPlugin({
            path:glob.sync(`${path.resolve('./src')}/**/*`), {nodir:true}
// src里面的所有文件
            satelist:function(){
                return {
                    standard:["html"]
                }
            }
        })
    ]
}
```

- paths：表示要检测哪些目录下的内容需要被分析，配合使用glob
- 默认情况下，Purgecss会将我们的html标签的样式移除掉，如果我们希望保留，可以添加一个safelist的属性

9.2.7. 代码分离

将代码分离到不同的 `bundle` 中，之后我们可以按需加载，或者并行加载这些文件