Webpack面试刷题

你对 webpack 了解多少？(我说了下 webpack 的一些优化手段，打包时间方面，预编译、缓存、缩小构建目标，说了大概十个插件，然后打包体积上，JS 和 CSS 的Tree-Shaking 怎么配置)

- 你觉得 CommonJS 为什么不能做 Tree-Shaking ？
  - ESModule 既然是编译时加载，那它可以做到运行时加载吗，想过这个问题吗？(愣了一会，说webpack 有动态 import 的方式)
    - 写过 loader 和 plugin 吗？(实话实说，没有)那你知道两者有什么差异吗？(先loader后plugin)

所有的配置都在webpack.config.js里面进行配置

mode：代表打包模式有两种，development代表开发模式，production代表生产环境，区别就是开发环境对代码不会压缩，而生产环境会进行压缩。

entry：代表需要打包的文件

output：代表打包完成之后的文件，可以通过path设置路径，通过filename设置文件名

devtool：用来设置打包之前代码与打包之后代码之间的映射关系，方便进行错误提示，生产模式一般配置为保存在单独文件中。

常见的loader:

webpack本质上是一个node.js的模块打包工具，所以其只能用来处理JS文件，而不能处理其他文件，而loader的作用就是将其他类型的文件转换成webpack能够处理的模块，从而方便webpack进行打包。

注意当我们配置多个loader时，多个loader会按照**从右至左，从下往上**的顺序执行。

JavaScript 复制代码

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

JavaScript 复制代码

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

- **source-map-loader**：保存打包之前与打包之后代码之间的映射关系方便调试，通过devtool参数来配置
- **file-loader**：将webpack不能识别的文件转换为可以识别的类型，比如可以**处理图片，处理字体图标**
- **url-loader**：与file-loader功能几乎一样只不过可以设置一个阈值，当小于这个阈值时会采用base64进行编码（将之前需要引入的图片转换为Data URL后直接嵌入了页面中，从而减少了发送HTTP请求的次数，比较适用于图片数据比较小的情况下）然后在文件中引入，这样可以减少发送HTTP请求的数量，也是用来**处理图片以及字体图标**

url-loader是依托于file-loader实现的，其并没有对文件内容有什么修改，只是简单的将文件内容复制一份，并根据配置为其生成一个唯一的文件名。

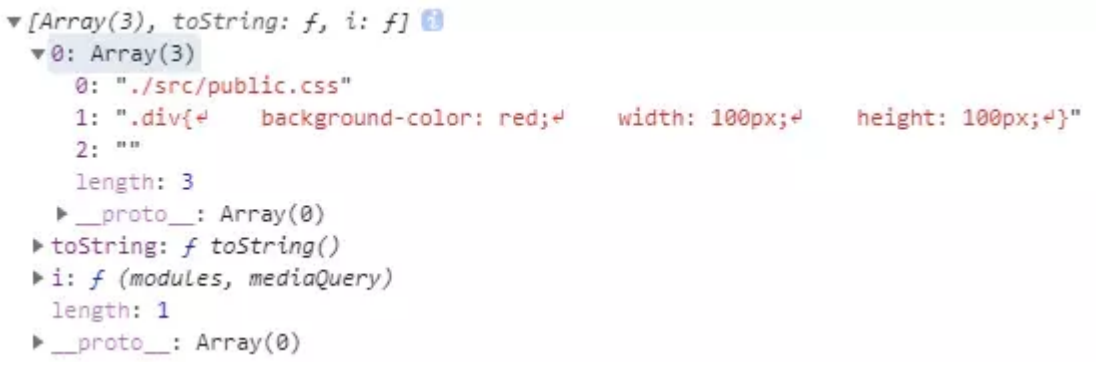
其实就是将import/require解析为url，并将需要引入的图片或者字体图标拷贝一份放入打包后的文件中。

- **css-loader**：加载css（也就是处理我们在js文件中引入的css），处理css文件导入（css文件中使用@import导入），压缩，支持模块化等

```JavaScript
import css from './public.css';

console.log(css);
```

将其打印出来一般是这样：



我们可以这样使用：

```JavaScript
import css from './public.css';

let div = document.createElement("div");
div.className = "rect";
div.innerText = "hello div";

let body = document.getElementsByTagName("body")[0];

let style = document.createElement("style");
style.innerText = css[0][1];

body.appendChild(style);
body.appendChild(div);
```

由于我们手动创建style标签太过于麻烦所以我们可以利用style-loader来帮助我们直接将处理好的css内容插入到html的<style>中。

这里所指的模块化就是我们可以将css文件像一个模块一样的引入进来，然后调用这样模块上的某一类。

比如这是index.css文件：

```JavaScript
.size {
    display: block;
    width: 100px;
    height: 100px;
    margin: 0 auto;
}
```

我们可以这样使用：

```JavaScript
import cssModule from "./index.css";

import cssModule from "./index.css";

let oImg = document.createElement("img");
oImg.setAttribute("class", cssModule.size);
```

这就是模块化，我们可以将css文件当做一个模块一样，按需引入我们所需要的类。

- **stylt-loader**：将打包好的css代码以<style>标签的形式插入到html页面中
- **less-loader**：将less编译成css
- **sass-loader**：将sass编译成css
- **postcss-loader**：css转换工具，配合**autoprefixer**可以实现自动补全浏览器的前缀，配合**pxtorem**可以自动将px转换为rem
- **babel-loader**：将ES678高级语法转换为ES5低级语法
- **eslint-loader**：检查js代码规范

常见的plugin：

- **html-plugin**：将打包好的文件自动引入到生成的html文件中，我们可以指定html文件模板
- **clean-plugin**：在打包之前先清空打包目录
- **copy-plugin**：将指定的文件或者目录复制到对应的目录中
- **mini-css-extract-plugin**：将打包好的css文件内容提前到一个单独的文件中
- **terser-webpack-plugin**：js代码压缩（支持ES6）
- **optimize-css-assets-webpack-plugin**：css代码压缩
- **IgnorePlugin**：忽略第三方库的指定目录，让其打包时不被打包进去

还有webpack的一些工具：

- **webpack-watch**：文件修改后自动进行重新打包
- **webpack-dev-server**：将打包好的文件运行在一个服务器环境下，可以解决开发阶段的跨域问题
- **HotModuleReplacementPlugin**：动态更新页面但不刷新网页

说说loader与plugin的区别：

作用上的不同：

loader：

直译为“加载器”，webpack将一切文件视为模块，但是原生webpack只能处理js文件，为了让webpack可以处理其他类型的文件于是就有了loader，其本质是一个函数，将接收到的内容进行转换，返回转换后的结果，让webpack有了加载与解析其他文件的能力。

plugin：

直译为“插件”，其主要用来丰富webpack的功能。在webpack的生命周期中会广播许多的事件，plugin对这些事件进行监听，在恰当的时机通过webpack提供的API改变输出的结果。

配置上的不同：

loader：

loader主要在**module.rules**这个数组中进行配置，每一项都是一个对象，也就是说其作为文件解析规则存在，里面有test表示处理什么类型的文件，loader表示用什么loader进行处理，options表示loader需要用到的一些配置参数等。

plugin：

其在plugins这个数组中单独配置，每一项都是一个实例对象，参数通过构造函数传入。

Tree Shaking：

基本配置：

对于js：

在webpack4中production模式会自动激活**Tree Shaking**，我们不再需要单独配置UglifyjsWebpackPlugin来进行Tree Shaking。

但需要注意如果我们想要将引入的库也进行Tree Shaking的话一定要引入ES Module模块系统的。

对于css：

如果想要对css进行Tree Shaking我们需要使用PurifyCSS这个插件，之后通过glob-all这个插件告诉PurifyCSS具体要Tree Shaking哪个路径下的文件。

JavaScript 复制代码

```
const PurifyCSS = require("purifycss-webpack");
const glob = require("glob-all");

let purifyCSS = new PurifyCSS({
  paths: glob.sync([
    // 要做CSS Tree Shaking的路径文件
    path.resolve(__dirname, "./*.html"),
    path.resolve(__dirname, "./src/*.js")
  ])
});
```

我们知道Tree Shaking是基于ES5的**静态结构**（其实就是在代码编译时其就会完成加载与解析相关依赖了）特性，但是如果我们有使用的bable-loader将ES Module转化为Common js（原因就是服务器都是使用node.js编写的，为了与服务器兼容）那么我们需要让bable不再进行转换。我们设置modules:false便好。

JavaScript 复制代码

```
{
   loader: "babel-loader",
   options: {
     presets: [
        [
          "@babel/preset-env",
          {
            useBuiltIns: "usage",
            corejs: { version: 3, proposals: true },
           // 禁止将import/export转为require写法
           modules: false
          }
        ]
      ]
   }
}
```

但是由于webpack在设计时考虑到比较周到，有些文件虽然没有被用到但是如果删除了还是会影响整个项目的，比如之前我们在js文件里引入的css样式，我们**可能**并不会在js文件里使用而是利用style-loader将其插入到html文件的header中，所以我们的webpack默认所有的文件都是重要的，都是对整个项目有影响的，我们需要告诉它哪些文件比较重要不能进行删除来让其放心的进行Tree Shaking。

我们需要在package.json中配置某些文件不进行Tree Shaking：

JavaScript 复制代码

```
  "sideEffects": [
    "*.css",
    "*.less",
    "*.scss"
  ],
```

或者默认所有的文件都进行：

JavaScript 复制代码

```
// 所有文件都有副作用，全都不可 tree-shaking
{
 "sideEffects": true
}
// 没有文件有副作用，全都可以 tree-shaking
{
 "sideEffects": false
}
// 只有这些文件有副作用，所有其他文件都可以 tree-shaking，但会保留这些文件
{
 "sideEffects": [
  "./src/file1.js",
  "./src/file2.js"
 ]
}
```

为什么使用Tree Shaking只有ES module才可以?

因为ES module是在文件解析前就进行加载执行模块了（无论你在文件的哪里引入），且我们知道JS代码在执行前会先进行编译，所以这样我们就可以提前知道在文件中哪里有使用该导入的模块如果没有使用的模块我们就将其删除掉。

> 因为`ES6`模块是运行时加载（静态加载），即可以在编译时就完成模块加载，使得编译时就能确定模块的依赖关系，可以进行可靠的静态分析，这就是`tree shaking`的基础。 而`CommonsJs`必须在跑起来运行的时候才能确定依赖关系，所以与不能`tree-shaking`。

webpack热更新原理：

webpack常用优化手段：

代码分割：

splitChunks是webpack自带的plugin主要是帮助我们处理分割文件的作用。

在多入口文件的情况下，如果我们两个入口文件都导入相同的文件，那么其可能相同的文件可能会重复两次被打包两次，所以我们需要将公共的文件分离出来，这就需要用到代码分割了，我们可以这样配置。

```JavaScript
    optimization: {
        splitChunks: {
            chunks: "all"
        }
    },
```

all代表所有引入的文件都要进行代码分割，还有其他取值。

- all: 不管文件是动态还是非动态载入，统一将文件分离。当页面首次载入会引入所有的包
- async： 将异步加载的文件分离，首次一般不引入，到需要异步引入的组件才会引入。
- initial：将异步和非异步的文件分离，如果一个文件被异步引入也被非异步引入，那它会被打包两次（注意和all区别），用于分离页面首次需要加载的包。

一般情况下webpack默认会帮助我们进行代码分割splitChunks的默认配置是这样：

```JavaScript
module.exports = {
  //...
  optimization: {
    splitChunks: {
      chunks: 'async',
      minSize: 20000,
      minRemainingSize: 0,
      maxSize: 0,
      minChunks: 1,
      maxAsyncRequests: 30,
      maxInitialRequests: 30,
      enforceSizeThreshold: 50000,
      cacheGroups: {
        defaultVendors: {
          test: /[\\/]node_modules[\\/]/,
          priority: -10,
          reuseExistingChunk: true,
        },
        default: {
          minChunks: 2,
          priority: -20,
          reuseExistingChunk: true,
        },
      },
    },
  },
};
```

我们可以通过splitChunks配置更多的定制分包规则。

闲时加载：

虽然异步加载可以帮助我们提高首页加载的速度，但是在需要的时候再去加载可能会造成用户等待的时间过长，所以我们通过在异步加载时添加魔法注释的方式来让相关模块在闲时加载。

```JavaScript
    const { default: $ } = await import(/* webpackPrefetch: true *//* webpackChunkName: "jquery" */"jquery");
```

Provide-Plugin：

如果我们有多个入口文件，都需要用到jQuery那么我们就需要在每个模块中都进行import这样重复写非常麻烦所以我们使用Provide-Plugin在全局一次性导入之后在文件中就可以尽情的使用导入的公共库了。

webpack-merge：

将webpack.config.js配置文件分隔开，这样分开写开发模式与生产模式的配置模式式代码更简明。

利用webpack来优化前端性能：

利用UglityjsWebpackPlugin与PurifyCSS给js与css代码做Tree Shaking减少无用的代码量，从而提高加载速度。

利用CDN加速，将我们需要引入的一些第三方库设置为CDN上对应的路径，publicPath来设置。

利用**terser-webpack-plugin**做js代码压缩，利用**optimize-css-assets-webpack-plugin**做css代码压缩。

利用webpack自带的splitChunks做代码分割（默认生产环境会进行代码分割的），异步代码按需异步引入，prefetching魔法注释做闲时加载。

利用CommonChunkPlugin提取公共代码。

长缓存优化：

- **利用CommonChunkPlugin提取公共代码**
- 配置contenthash（文件的内容改变对应文件名的hash才改变）
  - 如果是hash那么整个项目有一个文件改变那么整个项目对应的hash都会改变
  - 如果是chunkhash，那么chunk里任何改变都会导致chunkhash改变，包括我们在js文件里引入的css
  - 所以最终我们选择使用了contenthash只有chunk里的具体内容改变后hash才会改变
- 利用CommonChunkPlugin提前runtime
- 利用NamedChunksPlugin，NamedModulesPlugin固定chunk ID与module ID

如何提高webpack打包速度：

利用Happypack实现多线程打包。

使用`webpack-uglify-parallel`来提升`uglifyPlugin`的压缩速度。

利用externals设置一些常用库不进行打包。

配置动态链接库将那些不常用的第三方库只打包一次，利用`DllPlugin`和`DllReferencePlugin`预编译资源模块 通过`DllPlugin`来对那些我们引用但是绝对不会修改的npm包来进行预编译，再通过`DllReferencePlugin`将预编译的模块加载进来。

如何打包单页面应用？如何打包多页面应用？

单页面应用设置一个entry便好。

多页面应用可以配置多个入口文件，但是入口文件的名称要尽量灵活不然添加页面后还要重新修改配置，之后通过html-plugin配置不同的出口文件名称。

如何实现一个Loader？

- 通过loader-utils获取配置loader的参数

- 通过schema-utils进行loader参数值的校验

JavaScript 复制代码

```
const loaderUtils = require("loader-utils");
const validateOptions = require("schema-utils");
// 注意点:
// 1.webpack在使用Loader的时候，会将打包好的内容传递给当前的loader
// 2.webpack在使用Loader的时候，会修改loader中的this，所以在定义loader的时候只能使用ES5的函数
module.exports = function (source) {
    // 可以通过这个Loader获得options中的参数
    let options = loaderUtils.getOptions(this);
    // 制定校验规则
    let schema = {
        type: "object",
        // 指定配置中可以传递的参数
        properties: {
            // 可以传递name参数
            name: {
                // 参数必须为字符型
                type: "string"
            }
        },
        additionalProperties: false
    }
    // 利用校验规则进行校验
    validateOptions(schema, options, "ReplaceLoader");
    source = source.replace(/lmm/g, options.name);
    return source;
}
```

最后通过resolveLoader设置我们自己写的loader所在路径。

JavaScript 复制代码

```
    resolveLoader: {
        // 寻找Loader时首先在node_modules寻找，然后再loader目录下
        // modules: ['node_modules', './loader']
        // 给loader配置别名
        alias: {
            ReplaceLoader: path.resolve(__dirname, "./loader/ReplaceLoader.js")
        }
    },
```

如何实现一个Plugin：

plugin就是一个class类，我们可以在其中的apply方法通过实参complire.options中获取到配置文件，可以在complire.hooks获取所有的钩子，如果我们希望在entry配置完成后完成某些功能可以complire.hooks.entryOptins.tap中进行一些设置。

JavaScript 复制代码

```
const fs = require("fs");
const path = require("path");
class CleanWebpackPlugin {
    constructor(options) {
        console.log("插件被创建了", options);
    }
    apply(complire) {
        // 可以通过complire.options拿到webpack的配置文件
        // console.log(complire.options);
        // complire.hooks保存了所有的发布者
        // console.log(complire.hooks);
        let outputPath = complire.options.output.path;
        complire.hooks.entryOption.tap("CleanWebpackPlugin", () => {
            this.cleanDir(outputPath);
        })
    }
    // 注意点：node.js中不能直接删除非空目录
    cleanDir(dirPath) {
        // 首先判断是否为一个非空目录
        if (fs.statSync(dirPath).isDirectory() && fs.readdirSync(dirPath).length !== 0) {
            let files = fs.readdirSync(dirPath);
            // 如果非空首先删除里面的文件
            files.forEach((file) => {
                let filePath = path.resolve(dirPath, file);
                if (fs.statSync(filePath).isDirectory()) {
                    // 然后清空里面的文件夹
                    this.cleanDir(filePath);
                } else {
                    fs.unlinkSync(filePath);
                }
            })
        }
        fs.rmdirSync(dirPath);
    }
}

module.exports = CleanWebpackPlugin;
```

说说webpack的构建流程：

webpack本质上是一种工作流机制，通过类似发布订阅模式的**tapable**将各种插件连接起来在不同的阶段执行不同的插件。

- 首先合并shell与配置文件里的参数
- 传入参数初始化我们的 Compiler 对象，加载所有的插件，执行对象的run方法，run方法会构建 compilation 用于保存在一次构建中所有的数据
- 将我们的入口文件传入所有的 Loader 进行处理，然后找到其依赖文件递归调用 Loader 进行处理
- 根据入口以及打包之后模块的依赖关系组装成chunk
- 然后将所有的chunk放到我们的输出文件夹中（bundle）