六星教育 WEB 前端面试圣经

谈谈你对 webpack 的看法？

WebPack 是一个模块打包工具，你可以使用 WebPack 管理你的模块依赖，并编绎输出模 块们所需的静态文件。它能够很好地管理、打包 Web 开发中所用到的 HTML 、Javascript 、 CSS 以及各种静态文件（图片、字体等），让开发过程更加高效。对于 不同类型的资源， webpack 有对应的模块加载器。 webpack 模块打包器会分析模块间的依赖关系，最后 生成了优化且合并后的静态资源

webpack 打包体积 优化思路

提取第三方库或通过引用外部文件的方式引入第三方库

代码压缩插件 UglifyJsPlugin

服务器启用gzip 压缩

按需加载资源文件 require.ensure

优化 devtool 中的 source-map

剥离 css 文件，单独打包

去除不必要插件，通常就是开发环境与生产环境用同一套配置文件导致

webpack 打包效率

开发环境采用增量构建，启用热更新

开发环境不做无意义的工作如提取 css 计算文件 hash 等

配置 devtool

选择合适的 loader

个别 loader 开启 cache 如 babel-loader

第三方库采用引入方式

提取公共代码

优化构建时的搜索路径 指明需要构建目录及不需要构建目录

模块化引入需要的部分

webpack 编写一个 loader

loader 就是一个 node 模块，它输出了一个函数。当某种资源需要用这个 loader 转换时，这个函数会被调用。并且，这个函数可以通过提供给它的 this 上下文访问 Loader API 。 reverse-txt-loader

```
// 定义
module.exports = function(src) {
    //src 是原文件内容（abcde），下面对内容进行处理，这里是反转
    var result = src.split('').reverse().join('');
    //返回 JavaScript 源码，必须是 String 或者 Buffer
    return `module.exports = '${result}'`;
}
//使用
{
    test: /\.txt$/,
    use: [
        {
            './path/reverse-txt-loader'
```

```
        }
      ]
    },
```

说一下 webpack 的一些 plugin，怎么使用webpack 对项目进行优化

构建优化

减少编译体积 ContextReplacementPugin 、 IgnorePlugin 、 babel-plugin-import 、 babel-plugin-transform-runtime

并行编译 happypack 、 thread-loader 、 uglifyjsWebpackPlugin 开启并行

缓存 cache-loader 、 hard-source-webpack-plugin 、 uglifyjsWebpackPlugin 开启缓存、 babel-loader 开启缓存预编译 dllWebpackPlugin && DllReferencePlugin 、 auto-dll-webapck-plugin

性能优化

减少编译体积 Tree-shaking 、 Scope Hositing

hash 缓存 webpack-md5-plugin

拆包 splitChunksPlugin 、 import() 、 require.ensure

webpack 优化打包速度

减少文件搜索范围

比如通过别名

loader 的 test ， include & exclude

Webpack4 默认压缩并行

Happypack 并发调用

babel 也可以缓存编译

webpack 如何实现一个插件

调用插件 apply 函数传入 compiler 对象

通过 compiler 对象监听事件

比如你想实现一个编译结束退出命令的插件

```
apply (compiler) {
    const afterEmit = (compilation, cb) => {
        cb()
        setTimeout(function () {
            process.exit(0)
        }, 1000)
    }
    compiler.plugin('after-emit', afterEmit)
}
}
module.exports = BuildEndPlugin
```

Webpack 性能优化

这部分的内容中，我们会聚焦于以下两个知识点，并且每一个知识点都属于高频考点：

有哪些方式可以减少 Webpack 的打包时间

有哪些方式可以让 Webpack 打出来的包更小

1 减少 Webpack 打包时间

1. 优化 Loader

对于 Loader 来说，影响打包效率首当其冲必属 Babel 了。因为 Babel 会将代码转为字符串生成 AST ，然后对 AST 继续进行转变最后再生成新的 代码，项目越大，转换代码越多，效率就越低。当然了，我们是有办法优化的

首先我们可以优化 Loader 的文件搜索范围

```
module.exports = {
    module: {
        rules: [
            {
                // js 文件才使用 babel
                test: /\.js$/,
                loader: 'babel-loader',
                // 只在 src 文件夹下查找
                include: [resolve('src')],
```

```
                // 不会去查找的路径
                exclude: /node_modules/
            }
        ]
    }
}
```

对于 Babel 来说，我们肯定是希望只作用在 JS 代码上的，然后 node_modules 中使用的代码都是编译过的，所以我们也完全没有必要再去 处理一遍

当然这样做还不够，我们还可以将 Babel 编译过的文件缓存起来，下次只需要编译更改过的代码文件即可，这样可以大幅度加快打包时间

```
loader: 'babel-loader?cacheDirectory=true'
```

2. HappyPack

受限于 Node 是单线程运行的，所以 Webpack 在打包的过程中也是单线程 的，特别是在执行 Loader 的时候， 时间编译的任务很多，这样就会导致 等待的情况。

HappyPack 可以将 Loader 的同步执行转换为并行的，这样就能充分利用 系统资源来加快打包效率了

```
module: {
    loaders: [
        {
            test: /\.js$/,
```

```
            include: [resolve('src')],
            exclude: /node_modules/,
            // id 后面的内容对应下面
            loader: 'happypack/loader?id=happybabel'
        }
    ]
    },
    plugins: [
        new HappyPack({
            id: 'happybabel',
            loaders: ['babel-loader?cacheDirectory'],
            // 开启 4 个线程
            threads: 4
        })
    ]
```

3. DllPlugin

DllPlugin 可以将特定的类库提前打包然后引入。这种方式可以极大的减少 打包类库的次数，只有当类库更新版本才有需要重新打包，并且也实现了将公 共代码抽离成单独文件的优化方案。

接下来我们就来学习如何使用 DllPlugin

```
// 单独配置在一个文件中
```

```
// webpack.dll.conf.js
const path = require('path')
const webpack = require('webpack')
module.exports = {
    entry: {
        // 想统一打包的类库
        vendor: ['react']
    },
    output: {
        path: path.join(__dirname, 'dist'),
        filename: '[name].dll.js',
        library: '[name]-[hash]'
    },
    plugins: [
        new webpack.DllPlugin({
        // name 必须和 output.library 一致
        name: '[name]-[hash]',
        // 该属性需要与 DllReferencePlugin 中一致
        context: __dirname,
        path: path.join(__dirname, 'dist', '[name]-manifest.json')
        })
    ]
```

```
}
```

然后我们需要执行这个配置文件生成依赖文件，接下来我们需要使用 DllReferencePlugin 将依赖文件引入项目中

```
// webpack.conf.js
module.exports = {
    // ...省略其他配置
    plugins: [
        new webpack.DllReferencePlugin({
            context: __dirname,
            // manifest 就是之前打包出来的 json 文件
            manifest: require('./dist/vendor-manifest.json'),
        })
    ]
}
```

4. 代码压缩

在 Webpack3 中，我们一般使用 UglifyJS 来压缩代码，但是这个是单线 程运行的，为了加快效率，我们可以使用 webpack-parallel-uglify- plugin 来并行运行 UglifyJS ，从而提高效率。

在 Webpack4 中，我们就不需要以上这些操作了，只需要将 mode 设置为 production 就可以默认开启以上功能。代码压缩也是我们必做的性能优化 方案，当然

我们不止可以压缩 JS 代码，还可以压缩 HTML 、 CSS 代码， 并且在压缩 JS 代码的过程中，我们还可以通过配置实现比如删除 console.log 这类代码的功能。

5. 一些小的优化点

我们还可以通过一些小的优化点来加快打包速度

resolve.extensions ：用来表明文件后缀列表，默认查找顺序是 ['.js','.json'] ，如果你的导入文件没有添加后缀就会按照这个顺序查找文件。我们应该尽可能减少后缀列表 度，然后将出现频率高的后缀排在前面

resolve.alias ：可以通过别名的方式来映射一个路径，能让 Webpack 更快找到路径

module.noParse ：如果你确定一个文件下没有其他依赖，就可以使用该属性让 Webpack 不扫描该文件，这种方式对于大型的类库很有帮助

2 减少 Webpack 打包后的文件体积

1. 按需加载

想必大家在开发 SPA 项目的时候，项目中都会存在十几甚至更多的路由 面。如果我们将这些 面全部打包进一个 JS 文件的话，虽然将多个请求合并 了，但是同样也加载了很多并不需要的代码，耗费了更 的时间。那么为了首 能更快地呈现给用户，我们肯定是希望首 能加载的文件体积越小越好，这 时候我们就可以使用按需加载，将每个路由 面单独打包为一个文件。当然不 仅仅路由可以按需加载，对于 loadash 这种大型类库同样可以使用这个功 能。

按需加载的代码实现这里就不详细展开了，因为鉴于用的框架不同，实现起来 都是不一样的。当然了，虽然他们的用法可能不同，但是底层的机制都是一样 的。都是当使用的时候再去下载对应文件，返回一个 Promise ，当 Promise 成功以后去执行回调。

2. Scope Hoisting

Scope Hoisting 会分析出模块之间的依赖关系，尽可能的把打包出来的模 块合并到一个函数中去。

比如我们希望打包两个文件

```
// test.js
export const a = 1
// index.js
import { a } from './test.js'
```

对于这种情况，我们打包出来的代码会类似这样

```
[
    /* 0 */
    function (module, exports, require) {
        //...
    },
    /* 1 */
    function (module, exports, require) {
        //...
```

```
    }
]
```

但是如果我们使用 Scope Hoisting 的话，代码就会尽可能的合并到一个函 数中去，也就变成了这样的类似代码

```
[
    /* 0 */
    function (module, exports, require) {
    //...
}
```

这样的打包方式生成的代码明显比之前的少多了。如果在 Webpack4 中你希 望开启这个功能，只需要启用 optimization.concatenateModules 就可以 了。

```
module.exports = {
    optimization: {
        concatenateModules: true
    }
```

3. Tree Shaking

Tree Shaking 可以实现删除项目中未被引用的代码，比如

```
// test.js
export const a = 1
```

```
export const b = 2
// index.js
import { a } from './test.js'
```

对于以上情况， test 文件中的变量 b 如果没有在项目中使用到的话，就不会被打包到文件中。

如果你使用 Webpack 4 的话，开启生产环境就会自动启动这个优化功能。

实现小型 Javascript 打包工具

该工具可以实现以下两个功能

将 ES6 转换为 ES5

支持在 JS 文件中 import CSS 文件

通过这个工具的实现，大家可以理解到打包工具的原理到底是什么实现

因为涉及到 ES6 转 ES5 ，所以我们首先需要安装一些 Babel 相关的工具

```
yarn add babylon babel-traverse babel-core babel-preset-env
```

接下来我们将这些工具引入文件中

```
const fs = require('fs')
const path = require('path')
const babylon = require('babylon')
const traverse = require('babel-traverse').default
```

```
const { transformFromAst } = require('babel-core')
```

首先，我们先来实现如何使用 Babel 转换代码

```
function readCode(filePath) {
    // 读取文件内容
    const content = fs.readFileSync(filePath, 'utf-8')
    // 生成 AST
    const ast = babylon.parse(content, {
        sourceType: 'module'
    })
    // 寻找当前文件的依赖关系
    const dependencies = []
    traverse(ast, {
        ImportDeclaration: ({ node }) => {
            dependencies.push(node.source.value)
        }
    })
    // 通过 AST 将代码转为 ES5
    const { code } = transformFromAst(ast, null, {
        presets: ['env']
    })
    return {
        filePath,
```

```
        dependencies,
        code
    }
}
```

首先我们传入一个文件路径参数，然后通过 fs 将文件中的内容读取出来

接下来我们通过 babylon 解析代码获取 AST ，目的是为了分析代码中是否还引入了别的文件

通过 dependencies 来存储文件中的依赖，然后再将 AST 转换为 ES5 代码

最后函数返回了一个对象，对象中包含了当前文件路径、当前文件依赖和当前文件转换后的代码

接下来我们需要实现一个函数，这个函数的功能有以下几点

调用 readCode 函数，传入入口文件

分析入口文件的依赖

识别 JS 和 CSS 文件

```
function getDependencies(entry) {
    // 读取入口文件
    const entryObject = readCode(entry)
    const dependencies = [entryObject]
    // 遍历所有文件依赖关系
```

```
    for (const asset of dependencies) {
        // 获得文件目录
        const dirname = path.dirname(asset.filePath)
        // 遍历当前文件依赖关系
        asset.dependencies.forEach(relativePath => {
        // 获得绝对路径
            const absolutePath = path.join(dirname, relativePath)
            // CSS 文件逻辑就是将代码插入到 `style` 标签中
            if (/\.css$/.test(absolutePath)) {
                const content = fs.readFileSync(absolutePath, 'utf-8')
                const code = `
                const style = document.createElement('style')
                style.innerText                                    =
${JSON.stringify(content).replace(/\\r\\n/g, ''
                document.head.appendChild(style)
                `
                dependencies.push({
                    filePath: absolutePath,
                    relativePath,
                    dependencies: [],
                    code
                })
```

```
                } else {
                    // JS 代码需要继续查找是否有依赖关系
                    const child = readCode(absolutePath)
                    child.relativePath = relativePath
                    dependencies.push(child)
                }
            })
        }
        return dependencies
    }
```

首先我们读取入口文件，然后创建一个数组，该数组的目的是存储代码中涉及到的所有文件

接下来我们遍历这个数组，一开始这个数组中只有入口文件，在遍历的过程中，如果入口文件有依赖其他的文件，那么就会被 push 到这个数组中

在遍历的过程中，我们先获得该文件对应的目录，然后遍历当前文件的依赖关系

在遍历当前文件依赖关系的过程中，首先生成依赖文件的绝对路径，然后判断当前文件是 CSS 文件还是 JS 文件

如果是 CSS 文件的话，我们就不能用 Babel 去编译了，只需要读取 CSS 文件中的代码，然后创建一个 style 标签，将代码插入进标签并且放入 head 中即可

如果是 JS 文件的话，我们还需要分析 JS 文件是否还有别的依赖关系

最后将读取文件后的对象 push 进数组中

现在我们已经获取到了所有的依赖文件，接下来就是实现打包的功能了

```
function bundle(dependencies, entry) {
    let modules = ''
    // 构建函数参数，生成的结构为
    // { './entry.js': function(module, exports, require) { 代码 } }
    dependencies.forEach(dep => {
        const filePath = dep.relativePath || entry
        modules += `'${filePath}': (
            function (module, exports, require) { ${dep.code} }
        ),`
    })
    // 构建 require 函数，目的是为了获取模块暴露出来的内容
    const result = `
        (function(modules) {
            function require(id) {
            const module = { exports : {} }
            modules[id](module, module.exports, require)
            return module.exports
        }
        require('${entry}')
        })({${modules}})
```

```
    // 当生成的内容写入到文件中
    fs.writeFileSync('./bundle.js', result)
}
```

这段代码需要结合着 Babel 转换后的代码来看，这样大家就能理解为什么需 要这样写了

```
// entry.js
var _a = require('./a.js')
var _a2 = _interopRequireDefault(_a)
function _interopRequireDefault(obj) {
    return obj && obj.__esModule ? obj : { default: obj }
}
console.log(_a2.default)
// a.js
Object.defineProperty(exports, '__esModule', {
    value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS 的代码，但是浏览器 是不支持 CommonJS 的，所以如果这段代码需要在浏览器环境下运行的话， 我们需要自己实现 CommonJS 相关的代码，这就是 bundle 函数做的大部 分事情。

接下来我们再来逐行解析 bundle 函数

首先遍历所有依赖文件，构建出一个函数参数对象

对象的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 module 、 exports 、 require module 参数对应 CommonJS 中的 module exports 参数对应 CommonJS 中的 module.export require 参数对应我们自己创建的 require 函数

接下来就是构造一个使用参数的函数了，函数做的事情很简单，就是内部创建一个 require 函数，然后调用 require(entry) ，也就是 require(‘./entry.js’) ，这样

就会从函数参数中找到 ./entry.js 对应的函数并执行，最后将导出的内容通过 module.export 的方式让外部获取到

最后再将打包出来的内容写入到单独的文件中

如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码

```
;(function(modules) {
    function require(id) {
        // 构造一个 CommonJS 导出代码
        const module = { exports: {} }
        // 去参数中获取文件对应的函数并执行
        modules[id](module, module.exports, require)
        return module.exports
    }
```

```
    require('./entry.js')
})({
    './entry.js': function(module, exports, require) {
    // 这里继续通过构造的 require 去找到 a.js 文件对应的函数
    var _a = require('./a.js')
    console.log(_a2.default)
    },
    './a.js': function(module, exports, require) {
    var a = 1
    // 将 require 函数中的变量 module 变成了这样的结构
    // module.exports = 1
    // 这样就能在外部取到导出的内容了
    exports.default = a
    }
    // 省略
})
```

虽然实现这个工具只写了不到 100 行的代码，但是打包工具的核心原理就是 这些了

找出入口文件所有的依赖关系

然后通过构建 CommonJS 代码来获取 exports 导出的内容