# Webpack & Babel 面试题

**你知道webpack的作用是什么吗?**

- **模块打包。**可以将不同模块的文件打包整合在一起，并且保证它们之间的引用正确，执行有序。利用打包我们就可以在开发的时候根据我们自己的业务自由划分文件模块，保证项目结构的清晰和可读性
- **编译兼容。**通过 webpack 的 loader 机制，不仅可以帮助我们对代码做 ployfill，还可以编译转换诸如 .less、.vue、.jsx 这类在浏览器无法识别的格式文件，让我们在开发的时候可以使用新特性和新语法做开发，提高开发效率
- **能力扩展。**通过webpack 的 plugin 机制，我们可以在实现模块化和编译兼容的基础上，进一步实现诸如按需加载、代码压缩等一系列功能，帮助我们进一步提高自动化程度，工程效率以及打包输出的质量

**说一下模块打包运行原理?**

Webpack 的运行过程就是一个串行的过程，从启动到结束会依次执行以下流程：

- **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 对模块进行翻译，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。

简单说：

- 初始化：启动构建，读取与合并配置参数，加载 Plugin，实例化 Compiler
- 编译：从 Entry 出发，针对每个 Module 串行调用对应的 Loader 去翻译文件的内容，再找到该 Module 依赖的 Module，递归地进行编译
- 输出：将编译后的 Module 组合成 Chunk，将 Chunk 转换成文件，输出到文件系统中

具体请阅读 传送门

**webpack 的生命周期及钩子**

compiler 对象包含了 webpack 环境所有的配置信息。这个对象在启动 webpack时被一次性建立，并配置好所有可操作的设置，包括 options、loader 和 plugin。当在 webpack 环境中应用一个插件时，插件将收到此 compiler对象的引用。可以使用它来访问 webpack 的主环境

compilation 对象包含了当前的模块资源、编译生成资源、变化的文件等。当运行 webpack 开发环境中间件时，每当检测到一个文件变化，就会创建一个新的 compilation，从而生成了一组新的编译资源。compilation 对象也提供了很多关键时机的回调，以供插件做自定义处理时选择使用

compiler 代表了整个 webpack 从启动到关闭的生命周期，而 compilation 只是代表了一次新的编译过程

**有哪些常见的Loader？你用过哪些常见Loader?**

- raw-loader：加载文件原始内容（utf-8）
- file-loader：把文件输出到一个文件夹中，在代码中通过 URL 去引用输出的文件（处理图片和字体）
- url-loader：与 file-loader类似，区别是用户可以设置一个阈值，大于阈值会交给 file-loader 处理，小于阈值时返回文本 base64 形式编码（处理图片和字体）
- source-map-loader：加载额外的 Source Map 文件，以方便断点调试
- svg-inline-loader：将压缩后的SVG内容注入代码中
- image-loader：加载并且压缩图片文件
- json-loader：加载JSON文件（默认包含）
- handlebars-loader：将 Handlebars 模板编译成函数并返回
- babel-loader：将 ES6 转化成 ES5
- ts-loader：将 TypeScript 转换成 JavaScript
- awesome-typescript-loader：将 TypeScript 转换成 JavaScript，性能优于 ts-loader
- sass-loader：将SCSS/SASS代码转换成CSS
- css-loader：加载 CSS，支持模块化、压缩、文件导入等特性
- style-loader：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS
- postcss-loader：扩展 CSS 语法，使用下一代 CSS，可以配合 autoprefixer 插件自动补齐 CSS3 前缀
- eslint-loader：通过 ESLint 检查 JavaScript 代码
- tslint-loader：通过 TSLint检查 TypeScript 代码
- mocha-loader：加载 Mocha 测试用例的代码

- coverjs-loader：计算测试的覆盖率
- vue-loader：加载 Vue.js 单文件组件
- i18n-loader：国际化
- cache-loader：可以在一些开销较大的 Loader 之前添加，目的是将结果缓存到磁盘中

**有哪些常见的Plugin？你用过哪些Plugin?**

- define-plugin：定义环境变量（Webpack 4 之后指定 mode 会自动配置）
- ignore-plugin：忽略部分文件
- html-webpack-plugin：简化HTML文件创建（依赖于html-loader）
- web-webpack-plugin：可方便为单页应用输出HTML，比如html-webpack-plugin好用
- uglifyjs-webpack-plugin：不支持ES6压缩
- terser-webpack-plugin：支持压缩ES6（Webpack 4）
- webpack-parallel-uglify-plugin：多进程执行代码压缩，提升构建速度
- mini-css-extract-plugin：分离样式文件，CSS提取为独立文件，支持按需加载（替代extract-text-webpack-plugin）
- serviceworker-webpack-plugin：为网页应用增加离线缓存功能
- clean-webpack-plugin：目录清理
- ModuleConcatentationPlugin：开启Scope Hoisting
- speed-measure-webpack-plugin：可以看到每个 Loader 和 Plugin 执行耗时 (整个打包耗时、每个 Plugin 和 Loader 耗时)
- webpack-bundle-analyzer：可视化 Webpack 输出文件的体积（业务组件、依赖第三方模块）

**使用webpack开发时，你用过哪些可以提高效率的插件?**

- webpack-dashboard：可以更友好的展示相关打包信息
- webpack-merge：提供公共配置，减少重复配置代码
- speed-measure-webpack-plugin：简称 SMP，分析出 Webpack 打包过程中 Loader 和 Plugin 的耗时，有助于找到构建过程中的性能瓶颈
- size-plugin：监控资源体积变化，尽早发现问题
- HotModuleReplacementPlugin：模块热替换

**Loader 和 Plugin 的区别?**

Loader 本质就是一个函数，在该函数中对接收到的内容进行转换，返回转换后的结果。因为 Webpack 只认识 JavaScript，所以 loader 就成了翻译官，对其他类型的资源进行转译的预处理工作

Plugin 就是插件，基于事件流框架Tapable，插件可以扩展 Webpack 的功能，在Webpack运行的生命周期会广播出许多事件，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

Loader 在 module.rules 中配置，作为模块的解析规则，类型为数组。每一项都是一个Object，内部包含了test（类型文件）、loader、options（参数）等属性

Plugin 在 plugins 中单独配置，类型为数组，每一项都是一个Plugin的实例，参数都通过构造函数传入

**你知道sourceMap是什么吗?**

sourceMap 是一项将编译、打包、压缩后的代码映射会源代码的技术，打包压缩后的代码不具备良好的可读性，想要调试源码就需要 soucre map。sourceMap 可以帮助我们快速定位到源代码的位置，提高我们的开发效率。

map文件只要不打开开发者工具，浏览器是不会加载的。

线上环境一般有三种处理方案：

- hidden-source-map：借助第三方错误监控平台 Sentry 使用
- nosources-source-map：只会显示具体行数以及查看源代码的错误栈。安全性比 sourcemap 高
- sourcemap：通过 nignx 设置将 .map 文件只对白名单开发（公司内网）

注意：避免在生产中使用 inline- 和 eval-，因为它们会增加 bundle 体积大小，并降低整体性能

**文件指纹是什么？怎么用?**

文件指纹是打包后输出的文件名的后缀

- Hash：和整个项目的构建相关，只要项目文件有修改，整个项目构建的 hash 值就会更改
- Chunkhash：和 Webpack 打包的 chunk 有关，不同的 entry 会生出不同的chunkhash
- Contenthash：根据文件内容来定义 hash，文件内容不变，则 contenthash 不变

JS 的文件指纹设置：设置 output 的 filename，用 chunkhash

```
module.exports = {
    entry: {
        app: './scr/app.js',
        search: './src/search.js'
    },
    output: {
        filename: '[name][chunkhash:8].js',
        path:__dirname + '/dist'
    }
}
```

CSS 的文件指纹设置：设置 MiniCssExtractPlugin 的 filename，使用 contenthash

```
module.exports = {
    entry: {
        app: '.src/app.js',
        search: './src/search.js'
    },
    output: {
        filename: '[name][chunkhash:8].js',
        path: __dirname + '/dist'
    },
    plugins: [
        new MiniCssExtractPlugin({
            filename: `[name][contenthash: 8].css`
        })
    ]
}
```

图片的文件指纹设置：设置 file-loader 的 name，使用 hash

占位符名称及含义：

- 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
- contenthash 文件的内容hash，默认是md5生成
- hash 文件内容的hash，默认是md5生成
- emoji 一个随机的指代文件内容的emoj

```
const path = require('path')

module.exports = {
    entry: './src/index.js',
    output: {
        filename: 'bundle.js',
        path: path.resolve(__dirname, 'dist')
    },
    module: {
        rules: [{
            test: /\.(png|svg|jpg|gif)$/,
```

```
            use: [{
                loader: 'file-loader',
                options: {
                    name: 'img/[name][hash:8].[ext]'
                }
            }]
        }]
    }
}
```

**在实际工程中，配置文件上百行乃是常事，如何保证各个loader按照预想方式工作?**

可以使用 enforce 强制执行 loader 的作用顺序

- pre 代表在所有正常 loader 之前执行
- post 代表所有 loader之后执行
- inline 官方不推荐使用

**如何优化Webpack的构建速度?**

- 使用高版本的Webpack和Node.js
- 多核编译：happypack 项目接入多核编译，理解为 happypack 将编译工作灌满所有线程
- 多进程/多实例构建：thread-loader
- 压缩代码：
  - 多进程并行压缩
    - webpack-paralle-uglify-plugin
    - uglifyjs-webpack-plugin 开启 parallel 参数 (不支持ES6)
    - terser-webpack-plugin 开启 parallel 参数
  - 通过 mini-css-extract-plugin 提取 Chunk 中的CSS代码到单独文件，通过 css-loader 的 minimize 选项开启 cssnano 压缩 CSS
- 图片压缩：
  - 使用基于 Node 库的 imagemin (很多定制选项、可以处理多种图片格式)
  - 配置 image-webpack-loader
- 缩小打包作用域：
  - exclude/include (确定 loader 规则范围)
  - resolve.modules 指明第三方模块的绝对路径 (减少不必要的查找)
  - resolve.mainFields 只采用 main 字段作为入口文件描述字段 (减少搜索步骤，需要考虑到所有运行时依赖的第三方模块的入口文件描述字段)

    - resolve.extensions 尽可能减少后缀尝试的可能性
    - noParse 对完全不需要解析的库进行忽略 (不去解析但仍会打包到 bundle 中，注意被忽略掉的文件里不应该包含 import、require、define 等模块化语句)
    - IgnorePlugin (完全排除模块)
    - 合理使用alias
- 提取页面公共资源：
    - 基础包分离
        - 使用 html-webpack-externals-plugin，将基础包通过 CDN 引入，不打入 bundle 中
        - 使用 SplitChunksPlugin 进行(公共脚本、基础包、页面公共文件)分离(Webpack4内置)，替代了 CommonsChunkPlugin 插件
    - 抽离：
        - webpack-dll-plugin 将这些静态依赖从每一次的构建逻辑中抽离出去，静态依赖单独打包
        - Externals 将不需要打包的静态资源从构建逻辑剔除出去，使用 CDN 引用
- DLL：
    - 使用 DllPlugin 进行分包，使用 DllReferencePlugin(索引链接) 对 manifest.json 引用，让一些基本不会改动的代码先打包成静态资源，避免反复编译浪费时间。
    - HashedModuleIdsPlugin 可以解决模块数字id问题
- 充分利用缓存提升二次构建速度：、
    - 大部分 loader、plugin 都提供了 cache 配置项，比如 babel-loader、terser-webpack-plugin，可以开启缓存
    - 用 cache-loader 或者 hard-source-webpack-plugin，cache-loader 可以将loader的编译结果写入硬盘缓存
- Tree shaking
    - 打包过程中检测工程中没有引用过的模块并进行标记，在资源压缩时将它们从最终的bundle中去掉(只能对ES6 Modlue生效) 开发中尽可能使用ES6 Module的模块，提高tree shaking效率
    - 禁用 babel-loader 的模块依赖解析，否则 Webpack 接收到的就都是转换过的 CommonJS 形式的模块，无法进行 tree-shaking
    - 使用 PurifyCSS(不在维护) 或者 uncss 去除无用 CSS 代码
        - purgecss-webpack-plugin 和 mini-css-extract-plugin配合使用(建议)
    - 对组件库引用的优化
        - 使用 babel 插件 babel-plugin-import-fix，缩小引用范围
- Scope hositing
    - 构建后的代码会存在大量闭包，造成体积增大，运行代码时创建的函数作用域变多，内存开销变大。Scope hoisting 将所有模块的代码按照引用顺序放在一个函数作用域里，然后适当的重命名一些变量以防止变量名冲突
    - 必须是ES6的语法，因为有很多第三方库仍采用 CommonJS 语法，为了充分发挥 Scope

hoisting 的作用，需要配置 mainFields 对第三方模块优先采用 jsnext:main 中指向的ES6模块化语法

- 动态Polyfill
  - 建议采用 polyfill-service 只给用户返回需要的polyfill，社区维护。 (部分国内奇葩浏览器UA可能无法识别，但可以降级返回所需全部polyfill)

具体请阅读 传送门

**Webpack 代码分割的本质是什么？有什么意义呢?**

代码分割的本质其实就是在 源代码直接上线 和 打包成唯一脚本 main.bundle.js 这两种极端方案之间的一种更适合实际场景的中间状态

用可接受的服务器性能压力增加来换取更好的用户体验

**是否写过 Loader？简单描述一下编写Loader的思路?**

loader 函数中的 this 上下文由 webpack 提供，可以通过 this 对象提供的相关属性，获取当前 loader 需要的各种信息数据，事实上，这个 this 指向了一个叫 loaderContext 的 loader-runner 特定对象

Loader 支持链式调用，所以开发上需要严格遵循“单一职责”，每个 Loader 只负责自己需要负责的事情

- Loader 运行在 Node.js 中，我们可以调用任意 Node.js 自带的 API 或者安装第三方模块进行调用
- Webpack 传给 Loader 的原内容都是 UTF-8 格式编码的字符串，当某些场景下 Loader 处理二进制文件时，需要通过exports.raw = true 告诉 Webpack 该 loader 是否需要二进制数据
- 尽可能的异步化 Loader，如果计算量很小，同步也可以
- 使用 Loader-utils 和 schema-utils 为我们提供的使用工具
- 加载本地 Loader 方法
  - Npm link
  - ResolveLoader

```
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)
}
```

具体请阅读 传送门

**是否写过Plugin？简单描述一下编写plugin的思路？**

具体请阅读 传送门

webpack 在运行的生命周期中会广播出很多事件，Plugin 可以监听这些事件，在特定的阶段钩入想要添加的自定义功能。Webpack 的 Tapable 事件流机制保证了插件的有序性，使得整个系统扩展性良好

- compiler 暴露了和 webpack 整个生命周期相关的钩子
- compilation 暴露了与模块和依赖有关的粒度更小的事件钩子
- 插件需要在其原型上绑定apply方法，才能访问到 compiler 实例
- 传给每个插件的 compiler 和 compilation 对象都是同一个引用，若在一个插件中修改了它们身上的属性，会影响到后面的插件
- 找出合适的事件点去完成想要的功能
  - emit 事件发生时，可以读取到最终输出的资源、代码块、模块及其依赖，并进行修改（emit事件是修改Webpack输出资源的最后时机）
  - watch-run 当依赖的文件发生变化时会触发
- 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

**Tree Shaking 原理?**

Tree Shaking 是基于 ESM 模块基础进行处理的。ES Module 在 js 编译阶段就可以确定模块之间的依赖关系（import）以及模块的导出（export），所以我们并不需要代码执行就可以根据ESM确定模块之间的规划从而实现 Tree Shaking，我们称之为静态分析特性。它会在运行过程中静态分析模块之间的导入导出，确定ESM模块中哪些导出值未曾被其它模块使用，并将其删除，依次实现打包产物的优化

**wepack 和 gulp 的区别（模块化与流的区别）**

- gulp 强调的是前端开发的工作流程，我们可以通过配置一系列的task，定义task处理的事务（例如文件压缩合并、雪碧图、启动 server、版本控制等），然后定义执行顺序，来让 gulp 执行这些 task，从而构建项目的整个前端开发流程
- webpack 是一个前端模块化方案，更侧重模块打包，我们可以把开发中的所有资源（图片、js文件、css文件）都看成模块，通过 loader（加载器）和 plugins（插件）对资源进行处理，打包成符合生产环境部署的前端资源

**说一下 Webpack 的热更新原理吧**

Webpakkc 的热更新又称热替换（Hot Module Replacement），缩写为 HMR。这个机制可以做到不用刷新浏览器而将新变更的模块替换掉旧的模块。

HRM 的核心就是客户端从服务端拉取更新后的文件，准确的说是 chunk diff（chunk 需要更新的部分），实际上 WDS 与浏览器之间维护了一个 Websocket，当本地资源发生变化时，WDS 会向浏览器推送更新，并带上构建时的 hash，让客户端与上一次资源进行对比，客户端对比出差异后会向 WDS 发起 Ajax 请求来获取更给内容（文件列表、hash），这样客户端就可以再借助这些信息继续向 WDS 发起 jsonp 请求获取该 chunk 的增量更新

后续的部分(拿到增量更新之后如何处理？哪些状态该保留？哪些又需要更新？)由 HotModulePlugin 来完成，提供了相关 API 以供开发者针对自身场景进行处理，像 react-hot-loader 和 vue-loader 都是借助这些 API 实现 HMR。

**热更新流程总结：**

- 启动本地 server，让浏览器可以请求本地的静态资源
- 页面首次打开后，服务端与客户端通过 websocket 建立通信渠道，把下次的 hash 返回前端
- 客户端获取到 hash，这个 hash 将作为下一次请求服务端 hot-update.js 和 hot-update.json 的hash
- 修改页面代码后，Webpack 监听到文件修改后，开始编译，编译完成后，发送 build 消息给客户端
- 客户端获取到hash，成功后客户端构造hot-update.js script 链接，然后插入主文档
- hot-update.js 插入成功后，执行 hotAPI 的createRecord 和 reload 方法，获取到 Vue 组件的 render 方法，重新 render 组件，继而实现 UI 无刷新更新

**为什么 Vite 启动这么快**

Webpack 会先打包，然后启动开发服务器，请求服务器时直接给予打包结果。

而 Vite 是直接启动开发服务器，请求哪个模块再对该模块进行实时编译

Vite 将开发环境下的模块文件，就作为浏览器要执行的文件，而不是像 Webpack 那样进行打包合并

由于 Vite 在启动的时候是通过esbuild方式进行EsModel原生引入浏览器且按需更新。也就意味着不需要分析模块的依赖、

不需要编译。因此启动速度非常快。当浏览器请求某个模块时，再根据需要对模块内容进行编译。