再来一打Webpack面试题

1.有哪些常见的Plugin？你用过哪些Plugin？

- `define-plugin`： 定义环境变量 (Webpack4 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`： 可方便地为单页应用输出 HTML，比 html-webpack-plugin 好用
- `uglifyjs-webpack-plugin`： 不支持 ES6 压缩 (Webpack4 以前)
- `terser-webpack-plugin`: 支持压缩 ES6 (Webpack4)
- `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
- `ModuleConcatenationPlugin`: 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)

更多 Plugin 请参考官网

(Double Kill)

2.那你再说一说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。

`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，每一项是一个 Plugin 的实例，参数都通过构造函数传入。

3.Webpack构建流程简单说一下

Webpack 的运行流程是一个串行的过程，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，递归地进行编译处理
- 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

4.使用webpack开发时，你用过哪些可以提高效率的插件?

(这道题还蛮注重实际，用户的体验还是要从小抓起的)

- `webpack-dashboard`：可以更友好的展示相关打包信息。
- `webpack-merge`：提取公共配置，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

5.source map是什么？生产环境怎么用？

`source map` 是将编译、打包、压缩后的代码映射回源代码的过程。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。

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 nginx 设置将 .map 文件只对白名单开放(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。

6.模块打包原理知道吗？

Webpack 实际上为每个模块创造了一个可以导出和导入的环境，本质上并没有修改 代码的执行逻辑，代码执行顺序与模块加载顺序也完全一致。

7.文件监听原理呢？

在发现源码发生变化时，自动重新构建出新的输出文件。

Webpack开启监听模式，有两种方式：

- 启动 webpack 命令时，带上 --watch 参数
- 在配置 webpack.config.js 中设置 watch:true

缺点：每次需要手动刷新浏览器

原理：轮询判断文件的最后编辑时间是否变化，如果某个文件发生了变化，并不会立刻告诉监听者，而是先缓存起来，等 `aggregateTimeout` 后再执行。

```
复制代码
module.export = {
    // 默认false,也就是不开启
    watch: true,
    // 只有开启监听模式时，watchOptions才有意义
    watchOptions: {
        // 默认为空，不监听的文件或者文件夹，支持正则匹配
        ignored: /node_modules/,
```

```
        // 监听到变化发生后会等300ms再去执行，默认300ms
        aggregateTimeout:300,
        // 判断文件是否发生变化是通过不停询问系统指定文件有没有变化实现的，默认每秒问1000次
        poll:1000
    }
}
```

8.说一下 Webpack 的热更新原理吧

(敲黑板，这道题必考)

`Webpack` 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。

HMR的核心就是客户端从服务端拉去更新后的文件，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` 请求来获取更改内容(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。

9.如何对bundle体积进行监控和分析?

`VSCode` 中有一个插件 `Import Cost` 可以帮助我们对引入模块的大小进行实时监测，还可以使用 `webpack-bundle-analyzer` 生成 `bundle` 的模块组成图，显示所占体积。

`bundlesize` 工具包可以进行自动化资源体积监控。

10.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。

```
复制代码
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。

```
复制代码
module.exports = {
    entry: {
        app: './scr/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

```
复制代码
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]'
                }
            }]
        }]
    }
}
```

11.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` 是所有 loader 之后执行。(inline 官方不推荐使用)

12.如何优化 Webpack 的构建速度?

(这个问题就像能不能说一说**「从URL输入到页面显示发生了什么」**一样)

(我只想说：您希望我讲多长时间呢？)

(面试官：。。。)

- 使用 `高版本` 的 Webpack 和 Node.js
- `多进程/多实例构建` ： HappyPack(不维护了)、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
- `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`：
    - babel-loader 开启缓存
    - terser-webpack-plugin 开启缓存
    - 使用 cache-loader 或者 hard-source-webpack-plugin

- `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)
- `Scope hoisting`
  - 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)

更多优化请参考官网-构建性能

13.你刚才也提到了代码分割，那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。

`阿卡丽：荣耀剑下取，均衡乱中求`

「用可接受的服务器性能压力增加来换取更好的用户体验。」

源代码直接上线：虽然过程可控，但是http请求多，性能开销大。

打包成唯一脚本：一把梭完自己爽，服务器压力小，但是页面空白期长，用户体验不好。

(Easy peezy right)

14.是否写过Loader？简单描述一下编写loader的思路？

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的API 可以去官网查阅

- 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
- Loader 是无状态的，我们不应该在 Loader 中保留状态
- 使用 loader-utils 和 schema-utils 为我们提供的实用工具
- 加载本地 Loader 方法
  - Npm link
  - ResolveLoader

15.是否写过Plugin？简单描述一下编写Plugin的思路？

webpack在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。

Plugin的API 可以去官网查阅

- 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 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

16.聊一聊Babel原理吧

大多数JavaScript Parser遵循 `estree` 规范，Babel 最初基于 `acorn` 项目(轻量级现代 JavaScript 解析器) Babel大概分为三大部分：

- 解析：将代码转换成 AST
  - 词法分析：将代码(字符串)分割为token流，即语法单元成的数组
  - 语法分析：分析token流(上面生成的数组)并生成 AST
- 转换：访问 AST 的节点进行变换操作生产新的 AST
  - Taro就是利用 babel 完成的小程序语法转换

- 生成：以新的 AST 为基础生成代码