Webpack：

安装npm和webpack，还有webpack-cli

安装好后新建文件夹，初始化工程npm init -y

配置package.json文件：

“scripts”: {} 配置自定义脚本，我们可以在命令行通过npm run xxx调用在这里定义的脚本，例如定义“dev”: “webpack”，在命令行运行npm run dev就会进行一次文件打包构建，后期安装了伺服服务器后，将该脚本改为“dev”: “webpack server”，那么在运行npm run dev就会开启下载好的静态资源伺服服务器，进行实时打包

配置webpack.config.js文件，common.js语法，这个文件在wp打包开始之前告诉wp基于怎么样的配置去打包

wp文件module.exports = {}暴露一个配置对象，{}里面可以配置：

1. model: “ ”, development或者production作为属性值(dev模式不会压缩优化代码，效率高时间快，pro模式只在项目上线时使用，开发时不用)
2. entry: path.join(__dirname, “./src/index.js”)配置入口文件，其中path模块要提前引入才能使用，通过const path = require(“path”)，引入的是node.js的模块，__dirname指当前目录的绝对路径。
3. output: {} 配置出口文件，其中填写path，即出口文件路径path.join(__dirname, ./dist)，还有filename，即出口文件名。
4. plugin:[] 数组形式引入插件，使插件生效
5. devServer: {} 帮助我们配置wp server的更多设, devServer: { open: true, //初次打包完成后，自动打开浏览器host: “127.0.0.1”, //实时打包所使用的主机地址port：80, //实时打包所使用的端口号 }
6. module: {rules: [{test: //, use: [“”, “”]}]}用来通知webpack打包时如果使

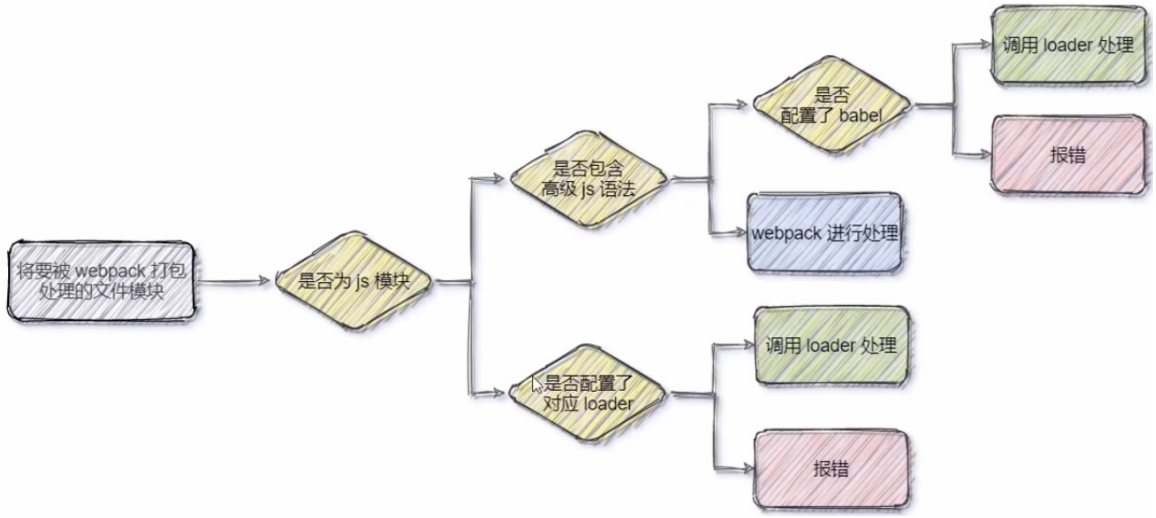
用下载好的loader处理非js文件

静态资源伺服服务器，webpack-dev-server
通过npm install … -D安装
监听根目录下静态资源的变化，并在文件保存时实时打包，出口文件不再存入原有的物理磁盘目录，而是在内存中进行缓存(提高效率)，在根目录下生成一个看不到的虚拟出口文件，所以，如果我们想要浏览器的渲染结果也随着实时打包而实时更新，就需要更改index.html的script src目录，原有目录是物理磁盘目录，是不会随着实时打包更新的，将该目录修改为根目录/下内存中的虚拟出口文件，才能实现浏览器渲染结果的实时更新

html-webpack-plugin插件，通过这个插件可以自定制index.html页面的内容，解决什么问题？
通过浏览器打开的服务器地址是根目录，很目录中没有index.html文件，想要查看效果需要点进src文件才可以看到index.html的渲染效果，所以我们需要在根目录中复制一份index.html，这个插件可以实现此需求
三个步骤:
1. 通过require引入该插件作为一个构造函数const XXX = require("html-webpack-plugin")
2. 实例化这个构造函数，const xxx = new XXX({template: "", filename: ""})
其中template为模版文件的存放路径，filename为生成文件的存放路径
3. 在module.exports={}中通过plugin:[xxx]，让上面插件的实例对象生效
生效后每次打包构建wp都会在根目录下拷贝一个存在内存里的虚拟index.html文件，并且该文件末尾还会在拷贝的时候自动通过script src注入打包后的出口文件，也就是说，在这样的情况下，index.html中我们不需要自己用src引入根目录中的虚拟出口文件了。

在上面的配置下，实时打包实时渲染环境中，即使删除dist文件，也不会影响打包以及渲染效果，因为浏览器渲染的是根目录下的拷贝而来的虚拟index.html文件，而该文件中注入的js出口文件，也不来自于实际物理磁盘路径dist，而是拷贝过程中自动注入的内存上的实时更新的打包出口文件，该文件也存在于根目录下，但我们看不到。

loader帮助wp打包非js文件，因为wp本身只能打包构建以.js结尾的文件，并且对js版本也有一定要求，高版本js有不支持的现象，同样需要bable-loader转化高版本js再打包



webpack中一切皆模块，都可以用import的方式引入index.js文件中，css文件，图片等也不例外

打包处理css文件

通常需要两个loader，也是通过npm i xxx -D来下载，它们分别是css-loader和style-loader，下载完相应的loader，我们需要通过配置文件中的module: {}告诉wp如何使用这些loader:

② 在 webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
module: { // 所有第三方文件模块的匹配规则
    rules: [ // 文件后缀名的匹配规则
        { test: /\.css$/, use: ['style-loader', 'css-loader'] }
    ]
}
```

其中，test 表示匹配的文件类型， use 表示对应要调用的 loader

打包处理less文件，跟上面的css文件差不多，需要下载两个loader，分别是less-loader和less，其中less是less-loader的内置依赖项，并不需要体现在use数组中

```
module: { // 所有第三方文件模块的匹配规则
    rules: [ // 文件后缀名的匹配规则
        { test: /\.less$/, use: ['style-loader', 'css-loader', 'less-loader'] },
    ]
}
```

打包处理样式表中的url文件，下载两个loader，分别是url-loader和file-loader，其中file-loader是url-loader的内置依赖，不需要体现在use中

```
module: { // 所有第三方文件模块的匹配规则
    rules: [ // 文件后缀名的匹配规则
        { test: /\.jpg|png|gif$/, use: 'url-loader?limit=22229' },
    ]
}
```

以上use也可以用对象的方式来表示：

use: {loader: “url-loader”, options: {limit: 22228}}

打包高级js语法文件，需要用到valve-loader，具体用法如下：

运行如下的命令安装对应的依赖包：

```
npm i babel-loader@8.2.1  @babel/core@7.12.3  @babel/plugin-proposal-class-properties@7.12.1  -D
```

包的名称及版本号列表如下（红色是包的名称、黑色是包的版本号）：

- babel-loader@8.2.1
- @babel/core@7.12.3
- @babel/plugin-proposal-class-properties@7.12.1

在 webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```
{
    test: /\.js$/,
    // exclude 为排除项，
    // 表示 babel-loader 只需处理开发者编写的 js 文件，不需要处理 node_modules 下的 js 文件
    exclude: /node_modules/,
    use: {
      loader: 'babel-loader',
      options: { // 参数项
        // 声明一个 babel 插件，此插件用来转化 class 中的高级语法
        plugins: ['@babel/plugin-proposal-class-properties'],
      },
    },
}
```

为什么要进行打包发布？

为了拿到存在内存中的隐藏文件，并且实现代码压缩性能优化，使代码上线后效率提高

如何配置打包发布？

在package.json中添加新脚本命令“build”=“webpack - -model production”

当我们在项目需要打包发布的时候在终端运行npm run build，会执行这个脚本，由于不再有server，所以打包生成的文件不再是存于内存上的看不到的文件，而会存于物理磁盘中，如果我们根目录中没有dist文件夹，那么就会新生成一个dist文件夹用来存放打包好的文件，另外，- -model production会覆盖掉wp.config.js中设置的model: “development”，让这次打包的wp运行环境是产品模式，实现代码的压缩和性能优化

如何整理打包后dist文件夹中杂乱无章的文件，让他们各自分类？

将js文件统一生成到dist文件夹下的js文件夹中：

在 webpack.config.js 配置文件的 output 节点中，进行如下的配置：

```
output: {
    path: path.join(__dirname, 'dist'),
    // 明确告诉 webpack 把生成的 bundle.js 文件存放到 dist 目录下的 js 子目录中
    filename: 'js/bundle.js',
}
```

将图片文件统一生成到dist文件夹下的image文件夹中：

修改 webpack.config.js 中的 url-loader 配置项，新增 outputPath 选项即可指定图片文件的输出路径：

```
{
    test: /\.jpg|png|gif$/,
    use: {
        loader: 'url-loader',
        options: {
            limit: 22228,
            // 明确指定把打包生成的图片文件，存储到 dist 目录下的 image 文件夹中
            outputPath: 'image',
        },
    },
}
```

利用插件自动清理dist文件夹下的旧文件：

为了在每次打包发布时自动清理掉 dist 目录中的旧文件，可以安装并配置 clean-webpack-plugin 插件

```
// 1. 安装清理 dist 目录的 webpack 插件
npm install clean-webpack-plugin@3.0.0 -D

// 2. 按需导入插件，得到插件的构造函数之后，创建插件的实例对象
const { CleanWebpackPlugin } = require('clean-webpack-plugin')
const cleanPlugin = new CleanWebpackPlugin()

// 3. 把创建的 cleanPlugin 插件实例对象，挂载到 plugins 节点中
plugins: [htmlPlugin, cleanPlugin], // 挂载插件
```

Source map的使用推荐

① 开发环境下：

- 建议把 devtool 的值设置为 eval-source-map
- 好处：可以精准定位到具体的错误行

② 生产环境下：

- 建议关闭 Source Map 或将 devtool 的值设置为 nosources-source-map
- 好处：防止源码泄露，提高网站的安全性