基本的 webpack 配置详解（一）

为什要使用 Weback

现今的很多网页其实可以看做是功能丰富的应用，它们拥有着复杂的 JavaScript 代码和一大堆依赖包。为了简化开发的复杂度，前端社区涌现出了很多好的实践方法

模块化，让我们可以把复杂的程序细化为小的文件;
类似于 TypeScript 这种在 JavaScript 基础上拓展的开发语言：使我们能够实现目前版本的 JavaScript 不能直接使用的特性，并且之后还能转换为 JavaScript 文件使浏览器可以识别；
Scss，less 等 CSS 预处理器
...
这些改进确实大大的提高了我们的开发效率，但是利用它们开发的文件往往需要进行额外的处理才能让浏览器识别,而手动处理又是非常繁琐的，这就为 WebPack 类的工具的出现提供了需求。
WebPack 可以看做是模块打包机：它做的事情是，分析你的项目结构，找到 JavaScript 模块以及其它的一些浏览器不能直接运行的拓展语言（Scss，TypeScript 等），并将其转换和打包为合适的格式供浏览器使用。

webpack 有哪些重要特征？

插件化：webpack 本身非常灵活，提供了丰富的插件接口。基于这些接口，webpack 开发了很多插件作为内置功能。
速度快：webpack 使用异步 IO 以及多级缓存机制。所以 webpack 的速度是很快的，尤其是增量更新。
丰富的 Loaders：loaders 用来对文件做预处理。这样 webpack 就可以打包任何静态文件。
高适配性：webpack 同时支持 AMD/CommonJs/ES6 模块方案。webpack 会静态解析你的代码，自动帮你管理他们的依赖关系。此外，webpack 对第三方库的兼容性很好。
代码拆分：webpack 可以将你的代码分片，从而实现按需打包。这种机制

可以保证页面只加载需要的 JS 代码，减少首次请求的时间。
优化：webpack 提供了很多优化机制来减少打包输出的文件大小，不仅如此，它还提供了 hash 机制，来解决浏览器缓存问题。
开发模式友好：webpack 为开发模式也提供了很多辅助功能。比如 SourceMap、热更新等。
使用场景多：webpack 不仅适用于 web 应用场景，也适用于 Webworkers、Node.js 场景。

接下来基于 Webpack4.0 做一个基本的配置，也是最简单的配置，毕竟 webpack 内容很多，便于大家学习，今天的内容只做最简单的打包的过程，我们会陆续的更新 webpack 的相关文章，以及每篇文章都会详解 webpack 配置中的每个字段的含义。

Webpack 的基本配置

初始化项目

```
npm install webpack webpack-cli -D
```

webpack4 抽离出了 webpack-cli,所以我们需要下载 2 个依赖。

不推荐全局安装 webpack。这会将你项目中的 webpack 锁定到指定版本，并且在使用不同的 webpack 版本的项目中，可能会导致构建失败。

Webpack 启动后会从 Entry 里配置的 Module 开始递归解析 Entry 依赖的所有 Module。 每找到一个 Module， 就会根据配置的 Loader 去找出对应的转换规则，对 Module 进行转换后，再解析出当前 Module 依赖的 Module。 这些模块会以 Entry 为单位进行分组，一个 Entry 和其所有依赖的 Module 被分到一个组也就是一个 Chunk。最后 Webpack 会把所有 Chunk 转换成文件输出。 在整个流程中 Webpack 会在恰当的时机执行 Plugin 里定义的逻辑。
webpack 需要在项目根目录下创建一个 webpack.config.js 来导出 webpack 的配置,配置多样化,可以自行定制,下面讲讲最基础的配置。

```
module.exports = {
    entry: './src/index.js',
    output: {
        path: path.join(__dirname, './dist'),
        filename: 'main.js',
```

```
    }
}
```

entry 代表入口，webpack 会找到该文件进行解析。
output 代表输入文件配置。
path 把最终输出的文件放在哪里。
filename 输出文件的名字。

有时候我们的项目并不是 spa，需要生成多个 js html，那么我们就需要配置多入口。

```
module.exports = {
    entry: {
        pageA: './src/pageA.js',
        pageB: './src/pageB.js'
    },
    output: {
        path: path.join(__dirname, './dist'),
        filename: '[name].[hash:8].js',
    },
}
```

entry 配置一个对象，key 值就是 chunk： 代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。看看 filename[name]: 这个 name 指的就是 chunk 的名字，我们配置的 key 值 pageA pageB，这样打包出来的文件名是不同的，再来看看[hash]，这个是给输出文件一个 hash 值，避免缓存，那么:8 是取前 8 位。
以上内容可以对基本的文件进行打包。接下来的文章会继续给大家讲解 webpack 更多的进阶的知识内容。

扩展：常用的 output 配置

这里再补充一下常用的 output 配置：
filename
output.filename 配置输出文件的名称，为 string 类型。 如果只有一个输出文件，则可以把它写成静态不变的：

filename: 'bundle.js'
但是在有多个 Chunk 要输出时，就需要借助模版和变量了。前面说到 Webpack 会为每个 Chunk 取一个名称，可以根据 Chunk 的名称来区分输出的文件名：

filename: '[name].js'
代码里的 [name] 代表用内置的 name 变量去替换 [name] ，这时你可以把它看作一个字符串模块函数，每个要输出的 Chunk 都会通过这个函数去拼接出输出的文件名称。

内置变量除了 name 还包括：

变量名 含义
id Chunk 的唯一标识，从 0 开始
name Chunk 的名称
hash Chunk 的唯一标识的 Hash 值
chunkhash Chunk 内容的 Hash 值
其中 hash 和 chunkhash 的长度是可指定的，[hash:8] 代表取 8 位 Hash 值，默认是 20 位。

注意 ExtractTextWebpackPlugin 插件是使用 contenthash 来代表哈希值而不是 chunkhash，原因在于 ExtractTextWebpackPlugin 提取出来的内容是代码内容本身而不是由一组模块组成的 Chunk。

chunkFilename
output.chunkFilename 配置无入口的 Chunk 在输出时的文件名称。chunkFilename 和上面的 filename 非常类似，但 chunkFilename 只用于指定在运行过程中生成的 Chunk 在输出时的文件名称。常见的会在运行时生成 Chunk 场景有在使用 CommonChunkPlugin 、使用 import('path/to/module') 动态加载等时。chunkFilename 支持和 filename 一致的内置变量。

path
output.path 配置输出文件存放在本地的目录，必须是 string 类型的绝对路径。通常通过 Node.js 的 path 模块去获取绝对路径：

path: path.resolve(__dirname, 'dist_[hash]')

publicPath
在复杂的项目里可能会有一些构建出的资源需要异步加载，加载这些异步资源需要对应的 URL 地址。

output.publicPath 配置发布到线上资源的 URL 前缀，为 string 类型。 默认值是空字符串 ''，即使用相对路径。

这样说可能有点抽象，举个例子，需要把构建出的资源文件上传到 CDN 服务上，以利于加快页面的打开速度。配置代码如下：

filename:'[name]_[chunkhash:8].js'
publicPath: 'https://cdn.example.com/assets/'
这时发布到线上的 HTML 在引入 JavaScript 文件时就需要：

<script src='https://cdn.example.com/assets/a_12345678.js'></script>
使用该配置项时要小心，稍有不慎将导致资源加载 404 错误。

output.path 和 output.publicPath 都支持字符串模版，内置变量只有一个：hash 代表一次编译操作的 Hash 值。

crossOriginLoading
Webpack 输出的部分代码块可能需要异步加载，而异步加载是通过 JSONP 方式实现的。 JSONP 的原理是动态地向 HTML 中插入一个 <script src="url"></script> 标签去加载异步资源。output.crossOriginLoading 则是用于配置这个异步插入的标签的 crossorigin 值。

script 标签的 crossorigin 属性可以取以下值：

anonymous(默认) 在加载此脚本资源时不会带上用户的 Cookies；
use-credentials 在加载此脚本资源时会带上用户的 Cookies。
通常用设置 crossorigin 来获取异步加载的脚本执行时的详细错误信息。

libraryTarget 和 library
当用 Webpack 去构建一个可以被其他模块导入使用的库时需要用到它们。

output.libraryTarget 配置以何种方式导出库。

output.library 配置导出库的名称。
它们通常搭配在一起使用。

output.libraryTarget 是字符串的枚举类型，支持以下配置。

var (默认)
编写的库将通过 var 被赋值给通过 library 指定名称的变量。

假如配置了 output.library='LibraryName'，则输出和使用的代码如下：

```
// Webpack 输出的代码
var LibraryName = lib_code;

// 使用库的方法
LibraryName.doSomething();
```

假如 output.library 为空，则将直接输出：

```
lib_code
```

其中
lib_code
代指导出库的代码内容，是有返回值的一个自执行函数。
commonjs
编写的库将通过 CommonJS 规范导出。

假如配置了 output.library='LibraryName'，则输出和使用的代码如下：

```
// Webpack 输出的代码
exports['LibraryName'] = lib_code;

// 使用库的方法
require('library-name-in-npm')['LibraryName'].doSomething();
```

其中
library-name-in-npm
是指模块发布到 Npm 代码仓库时的名称。
commonjs2
编写的库将通过 CommonJS2 规范导出，输出和使用的代码如下：

```
// Webpack 输出的代码
module.exports = lib_code;

// 使用库的方法
require('library-name-in-npm').doSomething();
```

CommonJS2 和 CommonJS 规范很相似，差别在于 CommonJS 只能用 exports 导出，而 CommonJS2 在 CommonJS 的基础上增加了 module.exports 的导出方式。

在 output.libraryTarget 为 commonjs2 时，配置 output.library 将没有意义。

this

编写的库将通过 this 被赋值给通过 library 指定的名称，输出和使用的代码如下：

```
// Webpack 输出的代码
this['LibraryName'] = lib_code;

// 使用库的方法
this.LibraryName.doSomething();
```

window

编写的库将通过 window 被赋值给通过 library 指定的名称，即把库挂载到 window 上，输出和使用的代码如下：

```
// Webpack 输出的代码
window['LibraryName'] = lib_code;

// 使用库的方法
window.LibraryName.doSomething();
```

global

编写的库将通过 global 被赋值给通过 library 指定的名称，即把库挂载到 global 上，输出和使用的代码如下：

```
// Webpack 输出的代码
global['LibraryName'] = lib_code;

// 使用库的方法
global.LibraryName.doSomething();
```

libraryExport

output.libraryExport 配置要导出的模块中哪些子模块需要被导出。 它只有在 output.libraryTarget 被设置成 commonjs 或者 commonjs2 时使用才有意义。

假如要导出的模块源代码是：

```
export const a=1;
export default b=2;
```

现在你想让构建输出的代码只导出其中的 a，可以把 output.libraryExport 设置成 a，那么构建输出的代码和使用方法将变成如下：

```
// Webpack 输出的代码
module.exports = lib_code['a'];

// 使用库的方法
require('library-name-in-npm')===1;
```

以上只是

output

里常用的配置项，还有部分几乎用不上的配置项没有一一列举