核心概念

* Entry：入口，Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入。
* Module：模块，在 Webpack 里一切皆模块，一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。
* 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。
* Loader：模块转换器，用于把模块原内容按照需求转换成新内容。
* Plugin：扩展插件，在 Webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要的事情。
* Output：输出结果，在 Webpack 经过一系列处理并得出最终想要的代码后输出结果。

2-1 Entry

entry是配置模块的入口，可抽象成输入，Webpack 执行构建的第一步将从入口开始搜寻及递归解析出所有入口依赖的模块。

Entry 类型

Entry 类型可以是以下三种中的一种或者相互组合：

|  |  |  |
| --- | --- | --- |
| 类型 | 例子 | 含义 |
| string | './app/entry' | 入口模块的文件路径，可以是相对路径。 |
| array | ['./app/entry1', './app/entry2'] | 入口模块的文件路径，可以是相对路径。 |
| object | { a: './app/entry-a', b: ['./app/entry-b1', './app/entry-b2']} | 配置多个入口，每个入口生成一个 Chunk |

如果是 array 类型，则搭配 output.library 配置项使用时，只有数组里的最后一个入口文件的模块会被导出。

Chunk 名称

Webpack 会为每个生成的 Chunk 取一个名称，Chunk 的名称和 Entry 的配置有关：

* 如果 entry 是一个 string 或 array，就只会生成一个 Chunk，这时 Chunk 的名称是 main；
* 如果 entry 是一个 object，就可能会出现多个 Chunk，这时 Chunk 的名称是 object 键值对里键的名称。

配置动态 Entry

假如项目里有多个页面需要为每个页面的入口配置一个 Entry ，但这些页面的数量可能会不断增长，则这时 Entry 的配置会受到到其他因素的影响导致不能写成静态的值。其解决方法是把 Entry 设置成一个函数去动态返回上面所说的配置，代码如下：

// 同步函数  
entry: () => {  
 return {  
 a:'./pages/a',  
 b:'./pages/b',  
 }  
};  
// 异步函数  
entry: () => {  
 return new Promise((resolve)=>{  
 resolve({  
 a:'./pages/a',  
 b:'./pages/b',  
 });  
 });  
};

2-2 Output

output 配置如何输出最终想要的代码。output 是一个 object，里面包含一系列配置项，下面分别介绍它们。

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。

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]')

2-3 Module

module 配置如何处理模块。

配置 Loader

rules 配置模块的读取和解析规则，通常用来配置 Loader。其类型是一个数组，数组里每一项都描述了如何去处理部分文件。 配置一项 rules 时大致通过以下方式：

1. 条件匹配：通过 test 、 include 、 exclude 三个配置项来命中 Loader 要应用规则的文件。
2. 应用规则：对选中后的文件通过 use 配置项来应用 Loader，可以只应用一个 Loader 或者按照从后往前的顺序应用一组 Loader，同时还可以分别给 Loader 传入参数。
3. 重置顺序：一组 Loader 的执行顺序默认是从右到左执行，通过 enforce 选项可以让其中一个 Loader 的执行顺序放到最前或者最后。

下面来通过一个例子来说明具体使用方法：

module: {  
 rules: [  
 {  
 // 命中 JavaScript 文件  
 test: /\.js$/,  
 // 用 babel-loader 转换 JavaScript 文件  
 // ?cacheDirectory 表示传给 babel-loader 的参数，用于缓存 babel 编译结果加快重新编译速度  
 use: ['babel-loader?cacheDirectory'],  
 // 只命中src目录里的js文件，加快 Webpack 搜索速度  
 include: path.resolve(\_\_dirname, 'src')  
 },  
 {  
 // 命中 SCSS 文件  
 test: /\.scss$/,  
 // 使用一组 Loader 去处理 SCSS 文件。  
 // 处理顺序为从后到前，即先交给 sass-loader 处理，再把结果交给 css-loader 最后再给 style-loader。  
 use: ['style-loader', 'css-loader', 'sass-loader'],  
 // 排除 node\_modules 目录下的文件  
 exclude: path.resolve(\_\_dirname, 'node\_modules'),  
 },  
 {  
 // 对非文本文件采用 file-loader 加载  
 test: /\.(gif|png|jpe?g|eot|woff|ttf|svg|pdf)$/,  
 use: ['file-loader'],  
 },  
 ]  
}

在 Loader 需要传入很多参数时，你还可以通过一个 Object 来描述，例如在上面的 babel-loader 配置中有如下代码：

* use: [  
   {  
   loader:'babel-loader',  
   options:{  
   cacheDirectory:true,  
   },  
   // enforce:'post' 的含义是把该 Loader 的执行顺序放到最后  
   // enforce 的值还可以是 pre，代表把 Loader 的执行顺序放到最前面  
   enforce:'post'  
   },  
   // 省略其它 Loader  
  ]

上面的例子中 test include exclude 这三个命中文件的配置项只传入了一个字符串或正则，其实它们还都支持数组类型，使用如下：

{  
 test:[  
 /\.jsx?$/,  
 /\.tsx?$/  
 ],  
 include:[  
 path.resolve(\_\_dirname, 'src'),  
 path.resolve(\_\_dirname, 'tests'),  
 ],  
 exclude:[  
 path.resolve(\_\_dirname, 'node\_modules'),  
 path.resolve(\_\_dirname, 'bower\_modules'),  
 ]  
}

数组里的每项之间是或的关系，即文件路径符合数组中的任何一个条件就会被命中。

5-3 编写Loader

loader 本质上是一个函数，输入参数是一个字符串，输出参数也是一个字符串。当然，输出的参数会被当成是 JS 代码，从而被 esprima 解析成 AST，触发进一步的依赖解析。webpack会按照从右到左的顺序执行loader。

Loader 基础

由于 Webpack 是运行在 Node.js 之上的，一个 Loader 其实就是一个 Node.js 模块，这个模块需要导出一个函数。 这个导出的函数的工作就是获得处理前的原内容，对原内容执行处理后，返回处理后的内容。

一个最简单的 Loader 的源码如下：

module.exports = function(source) {  
 // source 为 compiler 传递给 Loader 的一个文件的原内容  
 // 该函数需要返回处理后的内容，这里简单起见，直接把原内容返回了，相当于该 Loader 没有做任何转换  
 return source;  
};

由于 Loader 运行在 Node.js 中，你可以调用任何 Node.js 自带的 API，或者安装第三方模块进行调用：

const sass = require('node-sass');  
module.exports = function(source) {  
 return sass(source);  
};

获得 Loader 的 options

在最上面处理 SCSS 文件的 Webpack 配置中，给 css-loader 传了 options 参数，以控制 css-loader。 如何在自己编写的 Loader 中获取到用户传入的 options 呢？需要这样做：

const loaderUtils = require('loader-utils');  
module.exports = function(source) {  
 // 获取到用户给当前 Loader 传入的 options  
 const options = loaderUtils.getOptions(this);  
 return source;  
};

返回其它结果

上面的 Loader 都只是返回了原内容转换后的内容，但有些场景下还需要返回除了内容之外的东西。

例如以用 babel-loader 转换 ES6 代码为例，它还需要输出转换后的 ES5 代码对应的 Source Map，以方便调试源码。 为了把 Source Map 也一起随着 ES5 代码返回给 Webpack，可以这样写：

module.exports = function(source) {  
 // 通过 this.callback 告诉 Webpack 返回的结果  
 this.callback(null, source, sourceMaps);  
 // 当你使用 this.callback 返回内容时，该 Loader 必须返回 undefined，  
 // 以让 Webpack 知道该 Loader 返回的结果在 this.callback 中，而不是 return 中   
 return;  
};

其中的 this.callback 是 Webpack 给 Loader 注入的 API，以方便 Loader 和 Webpack 之间通信。 this.callback 的详细使用方法如下：

this.callback(  
 // 当无法转换原内容时，给 Webpack 返回一个 Error  
 err: Error | null,  
 // 原内容转换后的内容  
 content: string | Buffer,  
 // 用于把转换后的内容得出原内容的 Source Map，方便调试  
 sourceMap?: SourceMap,  
 // 如果本次转换为原内容生成了 AST 语法树，可以把这个 AST 返回，  
 // 以方便之后需要 AST 的 Loader 复用该 AST，以避免重复生成 AST，提升性能  
 abstractSyntaxTree?: AST  
);

Source Map 的生成很耗时，通常在开发环境下才会生成 Source Map，其它环境下不用生成，以加速构建。 为此 Webpack 为 Loader 提供了 this.sourceMap API 去告诉 Loader 当前构建环境下用户是否需要 Source Map。 如果你编写的 Loader 会生成 Source Map，请考虑到这点。

同步与异步

Loader 有同步和异步之分，上面介绍的 Loader 都是同步的 Loader，因为它们的转换流程都是同步的，转换完成后再返回结果。 但在有些场景下转换的步骤只能是异步完成的，例如你需要通过网络请求才能得出结果，如果采用同步的方式网络请求就会阻塞整个构建，导致构建非常缓慢。

在转换步骤是异步时，你可以这样：

module.exports = function(source) {  
 // 告诉 Webpack 本次转换是异步的，Loader 会在 callback 中回调结果  
 var callback = this.async();  
 someAsyncOperation(source, function(err, result, sourceMaps, ast) {  
 // 通过 callback 返回异步执行后的结果  
 callback(err, result, sourceMaps, ast);  
 });  
};

处理二进制数据

在默认的情况下，Webpack 传给 Loader 的原内容都是 UTF-8 格式编码的字符串。 但有些场景下 Loader 不是处理文本文件，而是处理二进制文件，例如 file-loader，就需要 Webpack 给 Loader 传入二进制格式的数据。 为此，你需要这样编写 Loader：

module.exports = function(source) {  
 // 在 exports.raw === true 时，Webpack 传给 Loader 的 source 是 Buffer 类型的  
 source instanceof Buffer === true;  
 // Loader 返回的类型也可以是 Buffer 类型的  
 // 在 exports.raw !== true 时，Loader 也可以返回 Buffer 类型的结果  
 return source;  
};  
// 通过 exports.raw 属性告诉 Webpack 该 Loader 是否需要二进制数据   
module.exports.raw = true;

以上代码中最关键的代码是最后一行 module.exports.raw = true;，没有该行 Loader 只能拿到字符串。

缓存加速

在有些情况下，有些转换操作需要大量计算非常耗时，如果每次构建都重新执行重复的转换操作，构建将会变得非常缓慢。 为此，Webpack 会默认缓存所有 Loader 的处理结果，也就是说在需要被处理的文件或者其依赖的文件没有发生变化时， 是不会重新调用对应的 Loader 去执行转换操作的。

如果你想让 Webpack 不缓存该 Loader 的处理结果，可以这样：

module.exports = function(source) {  
 // 关闭该 Loader 的缓存功能  
 this.cacheable(false);  
 return source;  
};

Loader其他常用API

其它 Loader API

除了以上提到的在 Loader 中能调用的 Webpack API 外，还存在以下常用 API：

* this.context：当前处理文件的所在目录，假如当前 Loader 处理的文件是 /src/main.js，则 this.context 就等于 /src。
* this.resource：当前处理文件的完整请求路径，包括 querystring，例如 /src/main.js?name=1。
* this.resourcePath：当前处理文件的路径，例如 /src/main.js。
* this.resourceQuery：当前处理文件的 querystring。
* this.target：等于 Webpack 配置中的 Target，详情见 [2-7其它配置项-Target](file:///E:\GitProject\%E6%B7%B1%E5%85%A5%E6%B5%85%E5%87%BAwebpack\2配置\2-7其它配置项.html#Target)。
* this.loadModule：当 Loader 在处理一个文件时，如果依赖其它文件的处理结果才能得出当前文件的结果时， 就可以通过 this.loadModule(request: string, callback: function(err, source, sourceMap, module)) 去获得 request 对应文件的处理结果。
* this.resolve：像 require 语句一样获得指定文件的完整路径，使用方法为 resolve(context: string, request: string, callback: function(err, result: string))。
* this.addDependency：给当前处理文件添加其依赖的文件，以便再其依赖的文件发生变化时，会重新调用 Loader 处理该文件。使用方法为 addDependency(file: string)。
* this.addContextDependency：和 addDependency 类似，但 addContextDependency 是把整个目录加入到当前正在处理文件的依赖中。使用方法为 addContextDependency(directory: string)。
* this.clearDependencies：清除当前正在处理文件的所有依赖，使用方法为 clearDependencies()。
* this.emitFile：输出一个文件，使用方法为 emitFile(name: string, content: Buffer|string, sourceMap: {...})。

加载本地Loader

在开发 Loader 的过程中，为了测试编写的 Loader 是否能正常工作，需要把它配置到 Webpack 中后，才可能会调用该 Loader。 在前面的章节中，使用的 Loader 都是通过 Npm 安装的，要使用 Loader 时会直接使用 Loader 的名称，代码如下：

module.exports = {  
 module: {  
 rules: [  
 {  
 test: /\.css$/,  
 use: ['style-loader'],  
 },  
 ]  
 },  
};

如果还采取以上的方法去使用本地开发的 Loader 将会很麻烦，因为你需要确保编写的 Loader 的源码是在 node\_modules 目录下。 为此你需要先把编写的 Loader 发布到 Npm 仓库后再安装到本地项目使用。

解决以上问题的便捷方法有两种，分别如下：

Npm link

Npm link 专门用于开发和调试本地 Npm 模块，能做到在不发布模块的情况下，把本地的一个正在开发的模块的源码链接到项目的 node\_modules 目录下，让项目可以直接使用本地的 Npm 模块。 由于是通过软链接的方式实现的，编辑了本地的 Npm 模块代码，在项目中也能使用到编辑后的代码。

完成 Npm link 的步骤如下：

1. 确保正在开发的本地 Npm 模块（也就是正在开发的 Loader）的 package.json 已经正确配置好；
2. 在本地 Npm 模块根目录下执行 npm link，把本地模块注册到全局；
3. 在项目根目录下执行 npm link loader-name，把第2步注册到全局的本地 Npm 模块链接到项目的 node\_moduels 下，其中的 loader-name 是指在第1步中的 package.json 文件中配置的模块名称。

链接好 Loader 到项目后你就可以像使用一个真正的 Npm 模块一样使用本地的 Loader 了。

ResolveLoader

在 [2-7其它配置项](file:///E:\GitProject\%E6%B7%B1%E5%85%A5%E6%B5%85%E5%87%BAwebpack\2配置\2-7其它配置项.html#ResolveLoader) 中曾介绍过 ResolveLoader 用于配置 Webpack 如何寻找 Loader。 默认情况下只会去 node\_modules 目录下寻找，为了让 Webpack 加载放在本地项目中的 Loader 需要修改 resolveLoader.modules。

假如本地的 Loader 在项目目录中的 ./loaders/loader-name 中，则需要如下配置：

module.exports = {  
 resolveLoader:{  
 // 去哪些目录下寻找 Loader，有先后顺序之分  
 modules: ['node\_modules','./loaders/'],  
 }  
}

加上以上配置后， Webpack 会先去 node\_modules 项目下寻找 Loader，如果找不到，会再去 ./loaders/ 目录下寻找。

5-4编写Plugin

Webpack 通过 Plugin 机制让其更加灵活，以适应各种应用场景。 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。

一个最基础的 Plugin 的代码是这样的：

class BasicPlugin{  
 // 在构造函数中获取用户给该插件传入的配置  
 constructor(options){  
 }

// Webpack 会调用 BasicPlugin 实例的 apply 方法给插件实例传入 compiler 对象  
 apply(compiler){  
 compiler.plugin('compilation',function(compilation) {  
 })  
 }  
}

// 导出 Plugin  
module.exports = BasicPlugin;

在使用这个 Plugin 时，相关配置代码如下：

const BasicPlugin = require('./BasicPlugin.js');  
module.export = {  
 plugins:[  
 new BasicPlugin(options),  
 ]  
}

Webpack 启动后，在读取配置的过程中会先执行 new BasicPlugin(options) 初始化一个 BasicPlugin 获得其实例。 在初始化 compiler 对象后，再调用 basicPlugin.apply(compiler) 给插件实例传入 compiler 对象。 插件实例在获取到 compiler 对象后，就可以通过 compiler.plugin(事件名称, 回调函数) 监听到 Webpack 广播出来的事件。 并且可以通过 compiler 对象去操作 Webpack。

Compiler 和 Compilation

在开发 Plugin 时最常用的两个对象就是 Compiler 和 Compilation，它们是 Plugin 和 Webpack 之间的桥梁。 Compiler 和 Compilation 的含义如下：

* Compiler 对象包含了 Webpack 环境所有的的配置信息，包含 options，loaders，plugins 这些信息，这个对象在 Webpack 启动时候被实例化，它是全局唯一的，可以简单地把它理解为 Webpack 实例；
* 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 Webpack 以开发模式运行时，每当检测到一个文件变化，一次新的 Compilation 将被创建。Compilation 对象也提供了很多事件回调供插件做扩展。通过 Compilation 也能读取到 Compiler 对象。

Compiler 和 Compilation 的区别在于：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。

事件流

Webpack 就像一条生产线，要经过一系列处理流程后才能将源文件转换成输出结果。 这条生产线上的每个处理流程的职责都是单一的，多个流程之间有存在依赖关系，只有完成当前处理后才能交给下一个流程去处理。 插件就像是一个插入到生产线中的一个功能，在特定的时机对生产线上的资源做处理。

Webpack 通过 [Tapable](https://github.com/webpack/tapable) 来组织这条复杂的生产线。 Webpack 在运行过程中会广播事件，插件只需要监听它所关心的事件，就能加入到这条生产线中，去改变生产线的运作。 Webpack 的事件流机制保证了插件的有序性，使得整个系统扩展性很好。

Webpack 的事件流机制应用了观察者模式，和 Node.js 中的 EventEmitter 非常相似。 Compiler 和 Compilation 都继承自 Tapable，可以直接在 Compiler 和 Compilation 对象上广播和监听事件，方法如下：

/\*\*  
\* 广播出事件  
\* event-name 为事件名称，注意不要和现有的事件重名  
\* params 为附带的参数  
\*/  
compiler.apply('event-name',params);

/\*\*  
\* 监听名称为 event-name 的事件，当 event-name 事件发生时，函数就会被执行。  
\* 同时函数中的 params 参数为广播事件时附带的参数。  
\*/  
compiler.plugin('event-name',function(params) {

});

同理，compilation.apply 和 compilation.plugin 使用方法和上面一致。

在开发插件时，你可能会不知道该如何下手，因为你不知道该监听哪个事件才能完成任务。

在开发插件时，还需要注意以下两点：

* 只要能拿到 Compiler 或 Compilation 对象，就能广播出新的事件，所以在新开发的插件中也能广播出事件，给其它插件监听使用。
* 传给每个插件的 Compiler 和 Compilation 对象都是同一个引用。也就是说在一个插件中修改了 Compiler 或 Compilation 对象上的属性，会影响到后面的插件。
* 有些事件是异步的，这些异步的事件会附带两个参数，第二个参数为回调函数，在插件处理完任务时需要调用回调函数通知 Webpack，才会进入下一处理流程。例如：  
   compiler.plugin('emit',function(compilation, callback) {  
   // 支持处理逻辑  
    
  // 处理完毕后执行 callback 以通知 Webpack   
   // 如果不执行 callback，运行流程将会一直卡在这不往下执行   
   callback();  
   });

插件常用API

读取输出资源、代码块、模块及其依赖

有些插件可能需要读取 Webpack 的处理结果，例如输出资源、代码块、模块及其依赖，以便做下一步处理。

在 emit 事件发生时，代表源文件的转换和组装已经完成，在这里可以读取到最终将输出的资源、代码块、模块及其依赖，并且可以修改输出资源的内容。 插件代码如下：

class Plugin {  
 apply(compiler) {  
 compiler.plugin('emit', function (compilation, callback) {  
 // compilation.chunks 存放所有代码块，是一个数组  
 compilation.chunks.forEach(function (chunk) {  
 // chunk 代表一个代码块  
 // 代码块由多个模块组成，通过 chunk.forEachModule 能读取组成代码块的每个模块  
 chunk.forEachModule(function (module) {  
 // module 代表一个模块  
 // module.fileDependencies 存放当前模块的所有依赖的文件路径，是一个数组  
 module.fileDependencies.forEach(function (filepath) {  
 });  
 });

// Webpack 会根据 Chunk 去生成输出的文件资源，每个 Chunk 都对应一个及其以上的输出文件  
 // 例如在 Chunk 中包含了 CSS 模块并且使用了 ExtractTextPlugin 时，  
 // 该 Chunk 就会生成 .js 和 .css 两个文件  
 chunk.files.forEach(function (filename) {  
 // compilation.assets 存放当前所有即将输出的资源  
 // 调用一个输出资源的 source() 方法能获取到输出资源的内容  
 let source = compilation.assets[filename].source();  
 });  
 });

// 这是一个异步事件，要记得调用 callback 通知 Webpack 本次事件监听处理结束。  
 // 如果忘记了调用 callback，Webpack 将一直卡在这里而不会往后执行。  
 callback();  
 })  
 }  
}

监听文件变化

在[4-5使用自动刷新](file:///E:\GitProject\%E6%B7%B1%E5%85%A5%E6%B5%85%E5%87%BAwebpack\4优化\4-5使用自动刷新.html) 中介绍过 Webpack 会从配置的入口模块出发，依次找出所有的依赖模块，当入口模块或者其依赖的模块发生变化时， 就会触发一次新的 Compilation。

在开发插件时经常需要知道是哪个文件发生变化导致了新的 Compilation，为此可以使用如下代码：

// 当依赖的文件发生变化时会触发 watch-run 事件  
compiler.plugin('watch-run', (watching, callback) => {  
 // 获取发生变化的文件列表  
 const changedFiles = watching.compiler.watchFileSystem.watcher.mtimes;  
 // changedFiles 格式为键值对，键为发生变化的文件路径。  
 if (changedFiles[filePath] !== undefined) {  
 // filePath 对应的文件发生了变化  
 }  
 callback();  
});

默认情况下 Webpack 只会监视入口和其依赖的模块是否发生变化，在有些情况下项目可能需要引入新的文件，例如引入一个 HTML 文件。 由于 JavaScript 文件不会去导入 HTML 文件，Webpack 就不会监听 HTML 文件的变化，编辑 HTML 文件时就不会重新触发新的 Compilation。 为了监听 HTML 文件的变化，我们需要把 HTML 文件加入到依赖列表中，为此可以使用如下代码：

compiler.plugin('after-compile', (compilation, callback) => {  
 // 把 HTML 文件添加到文件依赖列表，好让 Webpack 去监听 HTML 模块文件，在 HTML 模版文件发生变化时重新启动一次编译  
 compilation.fileDependencies.push(filePath);  
 callback();  
});

修改输出资源

有些场景下插件需要修改、增加、删除输出的资源，要做到这点需要监听 emit 事件，因为发生 emit 事件时所有模块的转换和代码块对应的文件已经生成好， 需要输出的资源即将输出，因此 emit 事件是修改 Webpack 输出资源的最后时机。

所有需要输出的资源会存放在 compilation.assets 中，compilation.assets 是一个键值对，键为需要输出的文件名称，值为文件对应的内容。

设置 compilation.assets 的代码如下：

compiler.plugin('emit', (compilation, callback) => {  
 // 设置名称为 fileName 的输出资源  
 compilation.assets[fileName] = {  
 // 返回文件内容  
 source: () => {  
 // fileContent 既可以是代表文本文件的字符串，也可以是代表二进制文件的 Buffer  
 return fileContent;  
 },  
 // 返回文件大小  
 size: () => {  
 return Buffer.byteLength(fileContent, 'utf8');  
 }  
 };  
 callback();  
});

读取 compilation.assets 的代码如下：

compiler.plugin('emit', (compilation, callback) => {  
 // 读取名称为 fileName 的输出资源  
 const asset = compilation.assets[fileName];  
 // 获取输出资源的内容  
 asset.source();  
 // 获取输出资源的文件大小  
 asset.size();  
 callback();  
});

判断 Webpack 使用了哪些插件

在开发一个插件时可能需要根据当前配置是否使用了其它某个插件而做下一步决定，因此需要读取 Webpack 当前的插件配置情况。 以判断当前是否使用了 ExtractTextPlugin 为例，可以使用如下代码：

// 判断当前配置使用使用了 ExtractTextPlugin，  
// compiler 参数即为 Webpack 在 apply(compiler) 中传入的参数  
function hasExtractTextPlugin(compiler) {  
 // 当前配置所有使用的插件列表  
 const plugins = compiler.options.plugins;  
 // 去 plugins 中寻找有没有 ExtractTextPlugin 的实例  
 return plugins.find(plugin=>plugin.\_\_proto\_\_.constructor === ExtractTextPlugin) != null;

}