你了解 webpack 中插件的机制么

webpack 可谓是让人欣喜又让人忧，功能强大但需要一定的学习成本。在探寻 webpack 插件机制前，首先需要了解一件有意思的事情，

webpack 插件机制是整个 webpack 工具的骨架，而 webpack 本身也是利用这套插件机制构建出来的。因此在深入认识 webpack 插件机制后，再来进行项目的相关优化，想必会大有裨益。

感觉比较拗口，这个就得从 webpack 的 源码 看 webpack 的具体实现才能解释的清楚了，暂时可以理解为 webpack 的核心是一个编译器（Compiler），而这个编译器也是作为一个插件提供给 webpack 这个插件平台使用的。

webpack 插件

先简单明了的了解一下什么是 webpack 插件，如下面代码描述，some-webpack-plugin 就是一个简单的 webpack 插件了，这个插件专注处理 webpack 在编译过程中的某个特定的任务。

```
const webpack = require('webpack');

// 假设有这么一个 webpack plugin
const SomewebpackPlugin = require('some-webpack-plugin');

webpack({
    // ...
    plugins: [
        new SomewebpackPlugin({/* some plugin options */})
    ]
    // ...
});
```

那么怎么样的一个东西可以称之为 webpack 插件呢？一个完整的 webpack 插件需要满足以下几点规则和特征：

- 是一个独立的模块。
- 模块对外暴露一个 js 函数。
- 函数的原型 (prototype) 上定义了一个注入 compiler 对象的 apply 方法。

- apply 函数中需要有通过 compiler 对象挂载的 webpack 事件钩子，钩子的回调中能拿到当前编译的 compilation 对象，如果是异步编译插件的话可以拿到回调 callback。
- 完成自定义子编译流程并处理 complition 对象的内部数据。
- 如果异步编译插件的话，数据处理完成后执行 callback 回调。

这就描述了一个 webpack 插件的基础形态，具体每个插件的差异几乎只是在 自定义子编译流程 这一步，可以通过代码来深刻理解一下 webpack 插件的具体形态：

```
// 1、some-webpack-plugin.js 文件（独立模块）

// 2、模块对外暴露的 js 函数
function SomewebpackPlugin(pluginOpions) {
    this.options = pluginOptions;
}

// 3、原型定义一个 apply 函数，并注入了 compiler 对象
SomewebpackPlugin.prototype.apply = function (compiler) {
    // 4、挂载 webpack 事件钩子（这里挂载的是 emit 事件）
    compiler.plugin('emit', function (compilation, callback) {
        // ... 内部进行自定义的编译操作
        // 5、操作 compilation 对象的内部数据
        console.log(compilation);
        // 6、执行 callback 回调
        callback();
    });
};

// 暴露 js 函数
module.exports = SomewebpackPlugin;
```

compiler & compilation 对象

通过对 webpack 插件的初步了解，我们注意到了一个 Webpcak 插件中出现了两个对象，一个是 compiler 对象，一个是 compilation 对象，乍一看这两个对象肯定是一头雾水，compiler 和 compilation 对象是整个 webpack 最核心的两个对象，是扩展 webpack 功能的关键。为了更加利于后面对 webpack 插件机制的理解，先重点介绍一下这两个对象。

compiler 对象

compiler 对象是 webpack 的编译器对象，前文已经提到，webpack 的核心就是编译器，compiler 对象会在启动 webpack 的时候被一次性的初始化，compiler 对象中包含了所有 webpack 可自定义操作的配置，例如 loader 的配置，plugin 的配置，entry 的配置等各种原始 webpack 配置等，在 webpack 插件中的自定义子编译流程中，我们肯定会用到 compiler 对象中的相关配置信息，我们相当于可以通过 compiler 对象拿到 webpack 的主环境所有的信息。

源码如下：

```
// webpack/lib/webpack.js
const Compiler = require("./Compiler")

const webpack = (options, callback) => {
  ...
  options = new WebpackOptionsDefaulter().process(options) // 初始化 webpack 各配置参数
  let compiler = new Compiler(options.context)             // 初始化 compiler 对象，这里 options.context 为 process.cwd()
  compiler.options = options                               // 往 compiler 添加初始化参数
  new NodeEnvironmentPlugin().apply(compiler)              // 往 compiler 添加 Node 环境相关方法
  for (const plugin of options.plugins) {
    plugin.apply(compiler);
  }
  ...
}
```

compilation 对象

这里首先需要了解一下什么是编译资源，编译资源是 webpack 通过配置生成的一份静态资源管理 Map（一切都在内存中保存），以 key-value 的形式描述一个 webpack 打包后的文件，编译资源就是这一个个 key-value 组成的 Map。而编译资源就是需要由 compilation 对象生成的。

compilation 实例继承于 compiler，compilation 对象代表了一次单一的版本 webpack 构建和生成编译资源的过程。当运行 webpack 开发环境中间件时，每当检测到一个文件变化，一次新的编译将被创建，从而生成一组新的编译资源以及新的 compilation 对象。一个 compilation 对象包含了 当前的模块资源、编译生成资源、变化的文件、以

及 被跟踪依赖的状态信息。编译对象也提供了很多关键点回调供插件做自定义处理时选择使用。

由此可见，如果开发者需要通过一个插件的方式完成一个自定义的编译工作的话，如果涉及到需要改变编译后的资源产物，必定离不开这个 compilation 对象。

源码如下：

```
// webpack/lib/Compiler.js
const Compilation = require("./Compilation");

newCompilation(params) {
  const compilation = new Compilation(this);
  ...
  return compilation;
}
```

webpack 插件机制

webpack 以插件的形式提供了灵活强大的自定义 api 功能。使用插件，我们可以为 webpack 添加功能。另外，webpack 提供生命周期钩子以便注册插件。在每个生命周期点，webpack 会运行所有注册的插件，并提供当前 webpack 编译状态信息。

作为 webpack 的使用者和开发者，如果想要玩转 webpack，自定义一些自己的 webpack 插件是非常有必要的，而想要更好的写出更加完善的 webpack 插件，需要更加深刻的了解 webpack 的插件机制，以及了解整个 webpack 插件机制是如何运作起来的，webpack 插件机制为 webpack 平台带来了极大的灵活性，而这一插件机制追根溯源却离不开一个叫做 Tapable 的库。

Tapable & Tapable 实例

webpack 的插件架构主要基于 Tapable 实现的，Tapable 是 webpack 项目组的一个内部库，主要是抽象了一套插件机制。webpack 源代码中的一些 Tapable 实例都继承或混合了 Tapable 类。Tapable 能够让我们为 javaScript 模块添加并应用插件。 它可以被其它模块继承或混合。它类似于 NodeJS 的 EventEmitter 类，专注于自定义事件的触发和操作。 除此之外，Tapable 允许你通过回调函数的参数访问事件的生产者。

Tapable 实例对象都有四组成员函数:

- plugin(name<string>, handler<function>) - 这个方法允许给 Tapable 实例事件注册一个自定义插件。 这个操作类似于 EventEmitter 的 on(), 注册一个处

理函数 -> 监听器到某个信号 -> 事件发生时执行（开发者自定义的插件需要频繁的用到此方法来自定义事件钩子的处理函数，以便被主编译流程 emit 触发）。

- apply(...pluginInstances<AnyPlugin|function>[]) - AnyPlugin 是 AbstractPlugin 的子类，或者是一个有 apply 方法的类（或者，少数情况下是一个对象），或者只是一个有注册代码的函数。这个方法只是 apply 插件的定义，所以正真的事件监听器会被注册到 Tapable 实例的注册表。
- applyPlugins*(name<string>, ...) - 这是一组函数，使用这组函数，Tapable 实例可以对指定 hash 下的所有插件执行 apply。 这些方法执行类似于 EventEmitter 的 emit(), 可以针对不同的使用情况采用不同的策略控制事件发射（webpack 内部实现机制中在主流程的编译过程中频繁的使用此方法来 emit 外界插件的自定义的插件自定义的事件钩子）。
- mixin(pt<Object>) - 一个简单的方法能够以混合的方式扩展 Tapable 的原型，而非继承。

Tapable 的 README 中也有详细的描述，值得注意的是这组 applyPlugins* 方法，* 表示着不同情况的事件注册，这组 applyPlugins* 方法在 webpack 的源码中随处可见，它们也涉及到 webpack 插件的执行顺序，不同的 applyPlugins* 对应着以下不同的情况：

- 同步串行执行插件 - applyPlugins()
- 并行执行插件 - applyPluginsParallel()
- 插件一个接一个的执行,并且每个插件接收上一个插件的返回值(瀑布) - applyPluginsWaterfall()
- 异步执行插件 - applyPluginsAsync()
- 保护模式终止插件执行: 一旦某个插件返回 非 undefined，会退出运行流程并返回 这个插件的返回值。这看起来像 EventEmitter 的 once()，但他们是完全不同的 - applyPluginsBailResult()

很多 webpack 中的对象都继承了 Tapable 类，暴露了一个 plugin 方法。插件可以使用 plugin 方法注入自定义的构建步骤。在各种 webpack 插件中你可以看到 compiler.plugin 和 compilation.plugin 被频繁使用。基本上，每个插件的调用都在构建流程中绑定了回调来触发特定的步骤。每个插件会在 webpack 启动时被安装一次，webpack 通过调用插件的 apply 方法来安装它们，并且传递一个 webpack compiler 对象的引用。然后你可以调用 compiler.plugin 来访问资源的编译和它们独立的构建步骤。

示例：

```
// MyPlugin.js

function MyPlugin(options) {
```

```
    // Configure your plugin with options...
}

MyPlugin.prototype.apply = function (compiler) {
    compiler.plugin('compile', function (params) {
        console.log('The compiler is starting to compile...');
    });

    compiler.plugin('compilation', function (compilation) {
        console.log('The compiler is starting a new compilation...');

        compilation.plugin('optimize', function () {
            console.log('The compilation is starting to optimize files...');
        });
    });

    // 异步的事件钩子
    compiler.plugin('emit', function (compilation, callback) {
        console.log('The compilation is going to emit files...');
        callback();
    });
};

module.exports = MyPlugin;
```


webpack 插件相关的事件钩子

通过以上的了解，webpack 插件中的自定义子编译流程都是需要配合 webpack 主编译流程发挥功效的，我们如何保证我们的插件中所定义的编译逻辑能够准确的在合适的时机运行呢？

其实，之前已经了解过 webpack 的两个重要的对象，compiler 和 compilation 对象在这里发挥了重要的作用，我们也已经了解到这两个对象都是 Tapable 实例，webpack 通过继承的 Tapable 实例的方法，分别在 compile 对象和 compilation 对象都注册了一系列的事件钩子，这样可以使得开发者能够在 webpack 编译的任何过程中都能够插入自己的自定义处理逻辑。

webpack 的做法就是使用 Tapable 实例的 applyPlugins* 方法来预先设定好这些事件钩子，当然，webpack 在一些其他的 Tapable 实例对象中也定义了一些内部或外部的事件钩子，在这里我们主要了解和插件相关的 compiler 对象和 compilation 对象一共有哪些事件钩子。

compiler 事件钩子

为了让开发者能够方便的写出 webpack plugin，官方也给出了 compiler 对象的 事件钩子(Event Hooks)。

在这里，重点介绍几个我们在写插件的过程中可能会经常用到的一些事件钩子。

事件钩子	触发时机	得到的内容	类型
entry-option	初始化 option	-	同步
run	开始编译	compiler	异步
compile	真正开始的编译，在创建 compilation 对象之前	compilation 参数	同步
compilation	生成好了 compilation 对象，可以操作这个对象啦	compilation	同步
make	从 entry 开始递归分析依赖，准备对每个模块进行 build	compilation	并行
after-compile	编译 build 过程结束	compliation	异步
emit	在将内存中 assets 内容写到磁盘文件夹之前	compilation	异步
after-emit	在将内存中 assets 内容写到磁盘文件夹之后	compilation	异步
done	完成所有的编译过程	stats	同步
failed	编译失败的时候	error	同步

compiler 对象如何绑定事件钩子呢？前面已经介绍过了 Tapable 的原理了，由于 webpack 自身在继承于 Tapable 的 compiler 对象的各个关键时间点已经通过 applyPlugins*() 方法注册了事件钩子，开发者只需要绑定事件就行，compiler 会在合适的时机去 emit 开发者绑定的事件，compiler 的绑定事件钩子的方式如下：

```
// 前提是先要拿到 compiler 对象，apply 方法的回调中就能拿到，这里假设能拿到
compiler 对象
compiler.plugin('emit', function (compilation, callback) {
 // 可以得到 compilation 对象，如果是异步的事件钩子，能拿到 callback 回调。
 // 做一些异步的事情
 setTimeout(function () {
 console.log("Done with async work...");
 callback();
 }, 1000);
});
```

可以明显的看出，compiler 的事件钩子是建立在整个编译过程的基础上的，粒度较粗，通常对编译的结果要做细粒度的处理的时候，少不了 compilation 对象上定义的事件钩子。

compilation 事件钩子

前面已经介绍过 compilation 对象，compilation 对象代表了一次单一的版本 webpack 构建和生成编译资源的过程，compilation 对象可以访问所有的模块和它们的依赖（大部分是循环依赖）。在编译阶段，模块被 加载，封闭，优化，分块，哈希 和 重建 等等，这将是编译中任何操作主要的生命周期。

要处理模块层面的逻辑，非常有必要了解 compilation 的事件钩子，当然，非常多的 webpack 插件在都巧妙的应用这些事件钩子完成了很多不可思议的工作。具体的 compilation 事件钩子都有哪些呢？还是挑一些比较常用和重要的来解释一下，具体完整的 complation 的事件钩子可以参考官方文档

normal-module-loader

普通模块 loader，真实地一个一个加载模块图(分析之后的所有模块一种数据结构)中所有的模块的函数。

模块，就是通常所说的 AMD, CMD 等模块化的模块。

```
// 前提是能先取到 complation 对象（可以通过 compiler 事件钩子取到）
compilation.plugin('normal-module-loader', function (loaderContext, module) {
 // 这里是所有模块被加载的地方
 // 一个接一个，此时还没有依赖被创建，想拿到啥模块直接通过 module 取
});
```

seal

编译的封闭已经开始，这个时候再也收不到任何的模块了，进入编译封闭阶段（参考 webpack 流程图）。

```
compilation.plugin('seal', function () {
 // 你已经不能再接收到任何模块
 // 回调没有参数
});
```

optimize

优化编译，这个事件钩子特别重要，很多插件的优化工作都是基于这个事件钩子，表示 webpack 已经进入优化阶段。

```
compilation.plugin('optimize', function () {
 // webpack 已经进入优化阶段
 // 回调没有参数
});
```


optimize-modules

模块的优化

```
compilation.plugin('optimize-modules', function (modules) {
 // 等待处理的模块数组
 console.log(modules);
});
```

optimize-chunks

这是个重要的事件钩子，webpack 的 chunk 优化阶段。可以拿到模块的依赖，loader 等，并进行相应的处理。

```
compilation.plugin('optimize-chunks', function (chunks) {
 //这里一般只有一个 chunk，除非你在配置中指定了多个入口
 chunks.forEach(function (chunk) {
 // chunk 含有模块的循环引用
```

```
 chunk.modules.forEach(function (module) {
 console.log(module);
 // module.loaders, module.rawRequest, module.dependencies 等。
 });
 });
});
```

additional-assets

这是一个异步的事件钩子，在这个阶段可以为 compilation 对象创建额外的 assets，也就是说可以异步的在最后的产物中加入自己自定义的一些资源，可以看一下往 assets 里面新增一个 svg 资源的例子：

```
compiler.plugin('compilation', function (compilation) {
 compilation.plugin('additional-assets', function (callback) {
   download('https://some.host/some/path/some.svg', function (resp) {
     if (resp.status === 200) {
       compilation.assets['webpack-version.svg'] = toAsset(resp);
       callback();
     } else {
       callback(new Error('[webpack-example-plugin] Unable to download the
image'));
     }
    })
   });
});
```

optimize-chunk-assets

优化 chunk 的 assets 的事件钩子，这个优化阶段可以改变 chunk 的 assets 以达到重新改变资源内容的目。assets 被存储在 this.assets 中，但是它们并不都是 chunk 的 assets。一个 chunk 有一个 files 属性指出这个 chunk 创建的所有文件。附加的 assets 被存储在 this.additionalChunkAssets 中。

下面是一个为每个 chunk 添加注释头信息的例子：

```
compilation.plugin("optimize-chunk-assets", function (chunks, callback) {
 chunks.forEach(function (chunk) {
 chunk.files.forEach(function (file) {
```

```
 compilation.assets[file] = '/**some comments info**/\n' +
compilation.assets[file];
 });
 });
 callback();
});
```

optimize-assets

优化所有的 assets 的异步事件钩子，在这个阶段可以直接通过 this.assets 拿到所有的 assets，并进行自定义操作。类似 optimize-chunk-assets，但是这个事件钩子的回调是拿不到 chunks 的。

```
compilation.plugin("optimize-assets", function (asstes, callback) {
 console.log(assets);
 // 可以直接操作 assets 里面的 file
 callback();
});
```

compilation 对象还有一些其他的事件钩子，可以直接阅读 webpack 官方文档，文档这块写的不是很详细，最好是都试一试这些事件钩子，放到代码中 console.log() 跑一跑，看看具体的实现和结果就能够加深理解了。

写在最后

想要了解 webpack 的插件的机制，需要了解这么几件事情：

- webpack 插件形态
- webpack 运行流程
- Tapable & Tapable 实例，以及 Tapable 的实例方法
- compiler 和 compilation 对象以及事件钩子

剩下的事情就是搞清楚自己的需求写合适的插件。webpack 内置的插件以及很多优秀的开源的 webpack 插件的源码是很好的学习的例子，多多阅读，多多学习。