上回说到 Webpack 的 Loader 机制，今天我要跟你分享 Webpack 的另外一个重要的核心特性：插件机制。

Webpack 插件机制的目的是为了增强 Webpack 在项目自动化构建方面的能力。通过上一讲的介绍你应该知道，Loader 就是负责完成项目中各种各样资源模块的加载，从而实现整体项目的模块化，而 Plugin 则是用来解决项目中除了资源模块打包以外的其他自动化工作，所以说 Plugin 的能力范围更广，用途自然也就更多。

我在这里先介绍几个插件最常见的应用场景：

实现自动在打包之前清除 dist 目录（上次的打包结果）；

自动生成应用所需要的 HTML 文件；

根据不同环境为代码注入类似 API 地址这种可能变化的部分；

拷贝不需要参与打包的资源文件到输出目录；

压缩 Webpack 打包完成后输出的文件；

自动发布打包结果到服务器实现自动部署。

总之，有了 Plugin 的 Webpack 几乎“无所不能”。借助插件，我们就可以轻松实现前端工程化中绝大多数经常用到的功能，这也正是很多初学者会认为 “Webpack 就是前端工程化，或者前端工程化就是 Webpack” 的原因。

那接下来我们通过一些常用插件的使用，具体聊聊 Webpack 的插件机制，最后再通过开发一个自己的插件，去理解插件的工作原理。

体验插件机制
这里我们先来体验几个最常见的插件，首先第一个就是用来自动清除输出目录的插件。

通过之前的尝试，你可能已经发现，Webpack 每次打包的结果都是直接覆盖到 dist 目录。而在打包之前，dist 目录中就可能已经存入了一些在上一次打包操作时遗留的文件，当我们再次打包时，只能覆盖掉同名文件，而那些已经移除的资源文件就会一直累积在里面，最终导致部署上线时出现多余文件，这显然非常不合理。

更为合理的做法就是在每次完整打包之前，自动清理 dist 目录，这样每次打包过后，dist 目录中就只会存在那些必要的文件。

clean-webpack-plugin 这个插件就很好的实现了这一需求。它是一个第三方的 npm 包，我们需要先通过 npm 安装一下，具体操作如下：

复制代码

$ npm install clean-webpack-plugin --save-dev
安装过后，我们回到 Webpack 的配置文件中，然后导入 clean-webpack-plugin 插件，这个插件模块导出了一个叫作 CleanWebpackPlugin 的成员，我们先把它解构出来，具体代码如下。

复制代码
```
const { CleanWebpackPlugin } = require('clean-webpack-plugin')
```
回到配置对象中，添加一个 plugins 属性，这个属性就是专门用来配置插件的地方，它是一个数组，添加一个插件就是在这个数组中添加一个元素。

绝大多数插件模块导出的都是一个类型，我们这里的 CleanWebpackPlugin 也不例外，使用它，就是通过这个类型创建一个实例，放入 plugins 数组中，具体代码如下：

复制代码
```
// ./webpack.config.js

const { CleanWebpackPlugin } = require('clean-webpack-plugin')

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js'
  },
  plugins: [
    new CleanWebpackPlugin()
  ]
}
```
完成以后我们来测试一下 clean-webpack-plugin 插件的效果。回到命令行终端，再次运行 Webpack 打包，此时之前的打包结果就不会存在了，dist 目录中存放的就都是我们本次打包的结果。

在这里，我只是希望通过这个非常简单的插件带你体验一下 Webpack 插件的使用。一般来说，当我们有了某个自动化的需求过后，可以先去找到一个合适的插件，然后安装这个插件，最后将它配置到 Webpack 配置对象的 plugins 数组中，这个过程唯一有可能不一样的地方就是，有的插件可能需要有一些配置参数。

用于生成 HTML 的插件
除了自动清理 dist 目录，我们还有一个非常常见的需求，就是自动生成使用打包结果的 HTML，所谓使用打包结果指的是在 HTML 中自动注入 Webpack 打包生成的 bundle。

在使用接下来这个插件之前，我们的 HTML 文件一般都是通过硬编码的方式，单独存放在项目根目录下的，这种方式有两个问题：
项目发布时，我们需要同时发布根目录下的 HTML 文件和 dist 目录中所有的打包结果，非常麻烦，而且上线过后还要确保 HTML 代码中的资源文件路径是正确的。

如果打包结果输出的目录或者文件名称发生变化，那 HTML 代码中所对应的 script 标签也需要我们手动修改路径。

解决这两个问题最好的办法就是让 Webpack 在打包的同时，自动生成对应的 HTML 文件，让 HTML 文件也参与到整个项目的构建过程。这样的话，在构建过程中，Webpack 就可以自动将打包的 bundle 文件引入到页面中。

相比于之前写死 HTML 文件的方式，自动生成 HTML 的优势在于：

HTML 也输出到 dist 目录中了，上线时我们只需要把 dist 目录发布出去就可以了；

HTML 中的 script 标签是自动引入的，所以可以确保资源文件的路径是正常的。

具体的实现方式就需要借助于 html-webpack-plugin 插件来实现，这个插件也是一个第三方的 npm 模块，我们这里同样需要单独安装这个模块，具体操作如下：

复制代码
```
$ npm install html-webpack-plugin --save-dev
```
安装完成过后，回到配置文件，载入这个模块，不同于 clean-webpack-plugin，html-webpack-plugin 插件默认导出的就是插件类型，不需要再解构内部成员，具体如下：

复制代码
```
const HtmlWebpackPlugin = require('html-webpack-plugin')
```
有了这个类型过后，回到配置对象的 plugins 属性中，同样需要添加一下这个类型的实例对象，完成这个插件的使用，具体配置代码如下：

复制代码
```
// ./webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin')
const { CleanWebpackPlugin } = require('clean-webpack-plugin')

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js'
  },
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin()
  ]
}
```
最后我们回到命令行终端，再次运行打包命令，此时打包过程中就会自动生成一个 index.html 文件到 dist 目录。我们找到这个文件，可以看到文件中的内容就是一段使用了 bundle.js 的空白 HTML，具体结果如下：

至此，Webpack 就可以动态生成应用所需的 HTML 文件了，但是这里仍然存在一些需要改进的地方：

对于生成的 HTML 文件，页面 title 必须要修改；

很多时候还需要我们自定义页面的一些 meta 标签和一些基础的 DOM 结构。

也就是说，还需要我们能够充分自定义这个插件最终输出的 HTML 文件。

如果只是简单的自定义，我们可以通过修改 HtmlWebpackPlugin 的参数来实现。

我们回到 Webpack 的配置文件中，这里我们给 HtmlWebpackPlugin 构造函数传入一个对象参数，用于指定配置选项。其中，title 属性设置的是 HTML 的标题，我们把它设置为 Webpack Plugin Simple。meta 属性需要以对象的形式设置页面中的元数据标签，这里我们尝试为页面添加一个 viewport 设置，具体代码如下：

复制代码

```
// ./webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin')
const { CleanWebpackPlugin } = require('clean-webpack-plugin')

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js'
  },
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Webpack Plugin Sample',
      meta: {
        viewport: 'width=device-width'
      }
    })
  ]
}
```

完成以后回到命令行终端，再次打包，然后我们再来看一下生成的 HTML 文件，此时这里的 title 和 meta 标签就会根据配置生成，具体结果如下：

如果需要对 HTML 进行大量的自定义，更好的做法是在源代码中添加一个用于生成 HTML

的模板，然后让 html-webpack-plugin 插件根据这个模板去生成页面文件。

我们这里在 src 目录下新建一个 index.html 文件作为 HTML 文件的模板，然后根据我们的需要在这个文件中添加相应的元素。对于模板中动态的内容，可以使用 Lodash 模板语法输出，模板中可以通过 htmlWebpackPlugin.options 访问这个插件的配置数据，例如我们这里输出配置中的 title 属性，具体代码如下：

复制代码

```
<!-- ./src/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title><%= htmlWebpackPlugin.options.title %></title>
</head>
<body>
  <div class="container">
    <h1>页面上的基础结构</h1>
    <div id="root"></div>
  </div>
</body>
</html>
```

有了模板文件过后，回到配置文件中，我们通过 HtmlWebpackPlugin 的 template 属性指定所使用的模板，具体配置如下：

复制代码

```
// ./webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin')
const { CleanWebpackPlugin } = require('clean-webpack-plugin')

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js'
  },
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Webpack Plugin Sample',
      template: './src/index.html'
    })
  ]
```

}
完成以后我们回到命令行终端，运行打包命令，然后再来看一下生成的 HTML 文件，此时 HTML 中就都是根据模板生成的内容了，具体结果如下：

至此，你应该了解了如何通过 html-webpack-plugin 自定义输出 HTML 文件内容。

关于 html-webpack-plugin 插件，除了自定义输出文件的内容，同时输出多个 HTML 文件也是一个非常常见的需求，除非我们的应用是一个单页应用程序，否则一定需要输出多个 HTML 文件。

如果需要同时输出多个 HTML 文件，其实也非常简单，我们回到配置文件中，这里通过 HtmlWebpackPlugin 创建的对象就是用于生成 index.html 的，那我们完全可以再创建一个新的实例对象，用于创建额外的 HTML 文件。

例如，这里我们再来添加一个 HtmlWebpackPlugin 实例用于创建一个 about.html 的页面文件，我们需要通过 filename 指定输出文件名，这个属性的默认值是 index.html，我们把它设置为 about.html，具体配置如下：

复制代码
```
// ./webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin')
const { CleanWebpackPlugin } = require('clean-webpack-plugin')

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js'
  },
  plugins: [
    new CleanWebpackPlugin(),
    // 用于生成 index.html
    new HtmlWebpackPlugin({
      title: 'Webpack Plugin Sample',
      template: './src/index.html'
    }),
    // 用于生成 about.html
    new HtmlWebpackPlugin({
      filename: 'about.html'
    })
  ]
}
```
完成以后我们再次回到命令行终端，运行打包命令，然后我们展开 dist 目录，此时 dist 目

录中就同时生成了 index.html 和 about.html 两个页面文件。

根据这个尝试我们就应该知道，如果需要创建多个页面，就需要在插件列表中加入多个 HtmlWebpackPlugin 的实例对象，让每个对象负责一个页面文件的生成。

当然了，对于同时输出多个 HTML，一般我们还会配合 Webpack 多入口打包的用法，这样就可以让不同的 HTML 使用不同的打包结果。不过关于多入口打包的用法不在本课时的讨论范畴内，我们后面再进行介绍。

用于复制文件的插件
在我们的项目中一般还有一些不需要参与构建的静态文件，那它们最终也需要发布到线上，例如网站的 favicon、robots.txt 等。

一般我们建议，把这类文件统一放在项目根目录下的 public 或者 static 目录中，我们希望 Webpack 在打包时一并将这个目录下所有的文件复制到输出目录。

对于这种需求，我们可以使用 copy-webpack-plugin 插件来帮我们实现。

同理，我们需要先安装一下 copy-webpack-plugin 插件，安装完成过后，回到配置文件中，导入这个插件类型。然后同样在 plugins 属性中添加一个这个类型的实例，具体代码如下：

复制代码
```
// ./webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin')
const CopyWebpackPlugin = require('copy-webpack-plugin')
const { CleanWebpackPlugin } = require('clean-webpack-plugin')

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js'
  },
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Webpack Plugin Sample',
      template: './src/index.html'
    }),
    new CopyWebpackPlugin({
      patterns: ['public'] //  需要拷贝的目录或者路径通配符
    })
  ]
}
```
这个插件类型的构造函数需要我们传入一个字符串数组，用于指定需要拷贝的文件路径。它

可以是一个通配符，也可以是一个目录或者文件的相对路径。我们这里传入的是 public 目录，表示将这个目录下所有文件全部拷贝到输出目录中。当然了，你还可以在这个数组中继续添加其它路径，这样它在工作时可以同时拷贝。

配置完成以后回到命令行终端，再次运行 Webpack，此时 public 目录下的文件就会同时拷贝到输出目录中。

至此，我们简单了解了几个非常常用的插件，这里的重点是，你不仅要学会使用这几个插件的使用，还要能够总结出大多数插件在使用上的共性。

开发一个插件
通过前面的介绍，我们知道相比于 Loader，插件的能力范围更宽，因为 Loader 只是在模块的加载环节工作，而插件的作用范围几乎可以触及 Webpack 工作的每一个环节。

那么，这种插件机制是如何实现的呢？

其实说起来也非常简单，Webpack 的插件机制就是我们在软件开发中最常见的钩子机制。

钩子机制也特别容易理解，它有点类似于 Web 中的事件。在 Webpack 整个工作过程会有很多环节，为了便于插件的扩展，Webpack 几乎在每一个环节都埋下了一个钩子。这样我们在开发插件的时候，通过往这些不同节点上挂载不同的任务，就可以轻松扩展 Webpack 的能力。

具体有哪些预先定义好的钩子，我们可以参考官方文档的 API：

Compiler Hooks；

Compilation Hooks；

JavascriptParser Hooks。

接下来，我们来开发一个自己的插件，看看具体如何往这些钩子上挂载任务。

这里我的需求是，希望我们开发的这个插件能够自动清除 Webpack 打包结果中的注释，这样一来，我们的 bundle.js 将更容易阅读，如下图所示：

那这里我们同样在项目根目录下添加一个单独的 JS 文件。

复制代码
```
└── 04-webpack-plugins ····· sample root dir
```

```
├──── public ····· static dir
│     └──── favicon.ico ····· static file
├──── src ····· source dir
│     └──── main.js ····· entry module
├──── package.json ····· package file
+    ├──── remove-comments-plugin.js ····· my plugin
└──── webpack.config.js ····· webpack config file
```

Webpack 要求我们的插件必须是一个函数或者是一个包含 apply 方法的对象，一般我们都会定义一个类型，在这个类型中定义 apply 方法。然后在使用时，再通过这个类型来创建一个实例对象去使用这个插件。

所以我们这里定义一个 RemoveCommentsPlugin 类型，然后在这个类型中定义一个 apply 方法，这个方法会在 Webpack 启动时被调用，它接收一个 compiler 对象参数，这个对象是 Webpack 工作过程中最核心的对象，里面包含了我们此次构建的所有配置信息，我们就是通过这个对象去注册钩子函数，具体代码如下：

复制代码

```
// ./remove-comments-plugin.js
class RemoveCommentsPlugin {
  apply (compiler) {
    console.log('RemoveCommentsPlugin 启动')
    // compiler => 包含了我们此次构建的所有配置信息
  }
}
```

知道这些过后，还需要明确我们这个任务的执行时机，也就是到底应该把这个任务挂载到哪个钩子上。

我们的需求是删除 bundle.js 中的注释，也就是说只有当 Webpack 需要生成的 bundle.js 文件内容明确过后才可能实施。

那根据 API 文档中的介绍，我们找到一个叫作 emit 的钩子，这个钩子会在 Webpack 即将向输出目录输出文件时执行，非常符合我们的需求。

我们回到代码中，通过 compiler 对象的 hooks 属性访问到 emit 钩子，再通过 tap 方法注册一个钩子函数，这个方法接收两个参数：

第一个是插件的名称，我们这里的插件名称是 RemoveCommentsPlugin；

第二个是要挂载到这个钩子上的函数；

根据 API 文档中的提示，这里我们在这个函数中接收一个 compilation 对象参数，这个对象可以理解为此次运行打包的上下文，所有打包过程中产生的结果，都会放到这个对象中。

我们可以使用这个对象中的 assets 属性获取即将写入输出目录的资源文件信息，它是一个对象，我们这里通过 for in 去遍历这个对象，其中键就是每个文件的名称，我们尝试把它打印出来，具体代码如下：

复制代码
```
// ./remove-comments-plugin.js
class RemoveCommentsPlugin {
  apply (compiler) {
    compiler.hooks.emit.tap('RemoveCommentsPlugin', compilation => {
      // compilation => 可以理解为此次打包的上下文
      for (const name in compilation.assets) {
        console.log(name) // 输出文件名称
      }
    })
  }
}
```
完成以后，我们将这个插件应用到 Webpack 的配置中，然后回到命令行重新打包，此时打包过程就会打印我们输出的文件名称，代码如下：

我们再回到代码中，来打印一下每个资源文件的内容，文件内容需要通过遍历的值对象中的 source 方法获取，具体代码如下：

复制代码
```
// ./remove-comments-plugin.js
class RemoveCommentsPlugin {
  apply (compiler) {
    compiler.hooks.emit.tap('RemoveCommentsPlugin', compilation => {
      // compilation => 可以理解为此次打包的上下文
      for (const name in compilation.assets) {
        // console.log(name)
        console.log(compilation.assets[name].source()) // 输出文件内容
      }
    })
  }
}
```
回到命令行，再次打包，此时输出的文件内容也可以正常被打印。

能够拿到文件名和文件内容后，我们回到代码中。这里需要先判断文件名是不是以 .js 结尾，因为 Webpack 打包还有可能输出别的文件，而我们的需求只需要处理 JS 文件。

那如果是 JS 文件，我们将文件内容得到，再通过正则替换的方式移除掉代码中的注释，最

后覆盖掉 compilation.assets 中对应的对象，在覆盖的对象中，我们同样暴露一个 source 方法用来返回新的内容。另外还需要再暴露一个 size 方法，用来返回内容大小，这是 Webpack 内部要求的格式，具体代码如下：

复制代码

```
// ./remove-comments-plugin.js
class RemoveCommentsPlugin {
  apply (compiler) {
    compiler.hooks.emit.tap('RemoveCommentsPlugin', compilation => {
      // compilation => 可以理解为此次打包的上下文
      for (const name in compilation.assets) {
        if (name.endsWith('.js')) {
          const contents = compilation.assets[name].source()
          const noComments = contents.replace(/\/\*{2,}\/\s?/g, '')
          compilation.assets[name] = {
            source: () => noComments,
            size: () => noComments.length
          }
        }
      }
    })
  }
}
```

完成以后回到命令行终端，再次打包，打包完成过后，我们再来看一下 bundle.js，此时 bundle.js 中每行开头的注释就都被移除了。

以上就是我们实现一个移除注释插件的过程，通过这个过程我们了解了：插件都是通过往 Webpack 生命周期的钩子中挂载任务函数实现的。

写在最后

最后我们再来总结一下今天的内容：

首先，我们简单了解了几个非常常用的插件，这些插件一般都适用于任何类型的项目。不管你有没有使用框架，或者使用的是哪一个框架，它们基本上都会用到，所以说，在这之后你最好能够仔细过一遍这些插件的官方说明，看看它们还可以有哪些特别的用法，做到心中有数。

除此之外，社区中还提供了成百上千的插件，你并不需要也不可能全部认识。当你遇到一些具体的构建需求时，再去提炼你需求中的关键词然后搜索它们，例如，我想要压缩输出的图片，我会搜索 imagemin webpack plugin。虽然说每个插件的作用不尽相同，但是在用法上基本都是类似的。