Webpack5 核心原理

介绍

本章节我们主要学习：

- loader 原理
- 自定义常用 loader
- plugin 原理
- 自定义常用 plugin

Loader 原理

loader 概念

帮助 webpack 将不同类型的文件转换为 webpack 可识别的模块。

loader 执行顺序

1. 分类

- pre： 前置 loader
- normal： 普通 loader
- inline： 内联 loader
- post： 后置 loader

2. 执行顺序

- 4 类 loader 的执行优级为： `pre > normal > inline > post` 。
- 相同优先级的 loader 执行顺序为：从右到左，从下到上。

例如：

JavaScript 复制代码

```javascript
// 此时loader执行顺序：loader3 - loader2 - loader1
module: {
  rules: [
    {
      test: /\.js$/,
      loader: "loader1",
    },
    {
      test: /\.js$/,
      loader: "loader2",
    },
    {
      test: /\.js$/,
      loader: "loader3",
    },
  ],
},
```

JavaScript | 复制代码

```javascript
// 此时loader执行顺序：loader1 - loader2 - loader3
module: {
  rules: [
    {
      enforce: "pre",
      test: /\.js$/,
      loader: "loader1",
    },
    {
      // 没有enforce就是normal
      test: /\.js$/,
      loader: "loader2",
    },
    {
      enforce: "post",
      test: /\.js$/,
      loader: "loader3",
    },
  ],
},
```

3. 使用 loader 的方式

- 配置方式：在 `webpack.config.js` 文件中指定 loader。（pre、normal、post loader）
- 内联方式：在每个 `import` 语句中显式指定 loader。（inline loader）

4. inline loader

用法：`import Styles from 'style-loader!css-loader?modules!./styles.css';`

含义：

- 使用 `css-loader` 和 `style-loader` 处理 `styles.css` 文件
- 通过 `!` 将资源中的 loader 分开

`inline loader` 可以通过添加不同前缀，跳过其他类型 loader。

- `!` 跳过 normal loader。

`import Styles from '!style-loader!css-loader?modules!./styles.css';`

- `-!` 跳过 pre 和 normal loader。

```
import Styles from '-!style-loader!css-loader?modules!./styles.css';
```

- `!!` 跳过 pre、 normal 和 post loader。

```
import Styles from '!!style-loader!css-loader?modules!./styles.css';
```

开发一个 loader

1. 最简单的 loader

JavaScript | 复制代码

```
// loaders/loader1.js
module.exports = function loader1(content) {
  console.log("hello loader");
  return content;
};
```

它接受要处理的源码作为参数，输出转换后的 js 代码。

2. loader 接受的参数

- content 源文件的内容
- map SourceMap 数据
- meta 数据，可以是任何内容

loader 分类

1. 同步 loader

```javascript
module.exports = function (content, map, meta) {
  return content;
};
```

this.callback 方法则更灵活，因为它允许传递多个参数，而不仅仅是 content。

```javascript
module.exports = function (content, map, meta) {
  // 传递map，让source-map不中断
  // 传递meta，让下一个loader接收到其他参数
  this.callback(null, content, map, meta);
  return; // 当调用 callback() 函数时，总是返回 undefined
};
```

2. 异步 loader

```javascript
module.exports = function (content, map, meta) {
  const callback = this.async();
  // 进行异步操作
  setTimeout(() => {
    callback(null, result, map, meta);
  }, 1000);
};
```

由于同步计算过于耗时，在 Node.js 这样的单线程环境下进行此操作并不是好的方案，我们建议尽可能地使你的 loader 异步化。但如果计算量很小，同步 loader 也是可以的。

3. Raw Loader

默认情况下，资源文件会被转化为 UTF-8 字符串，然后传给 loader。通过设置 raw 为 true，loader 可以接收原始的 Buffer。

JavaScript 复制代码

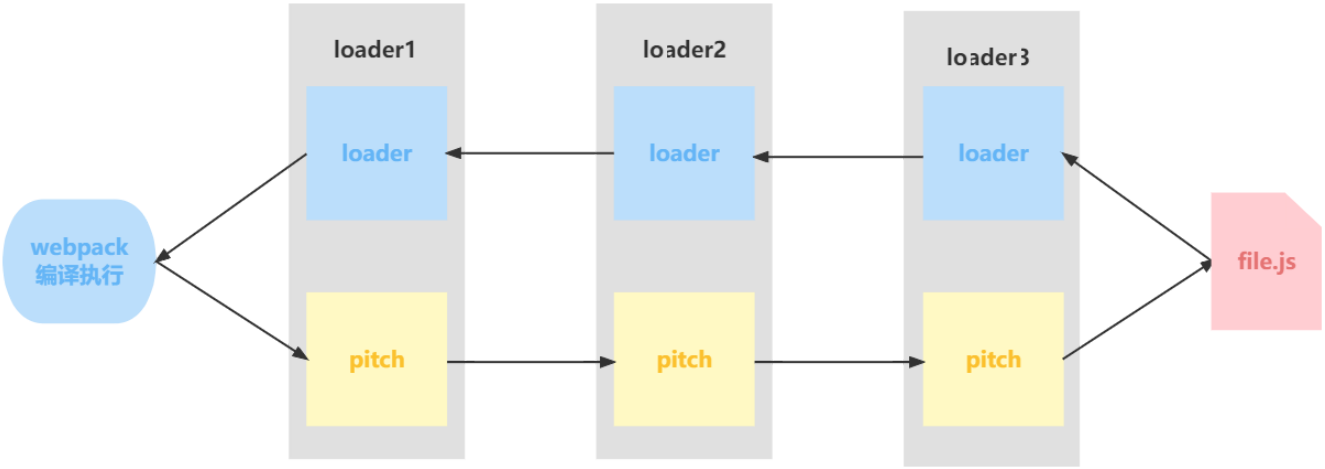
```
module.exports = function (content) {
  // content是一个Buffer数据
  return content;
};
module.exports.raw = true; // 开启 Raw Loader
```

4. Pitching Loader

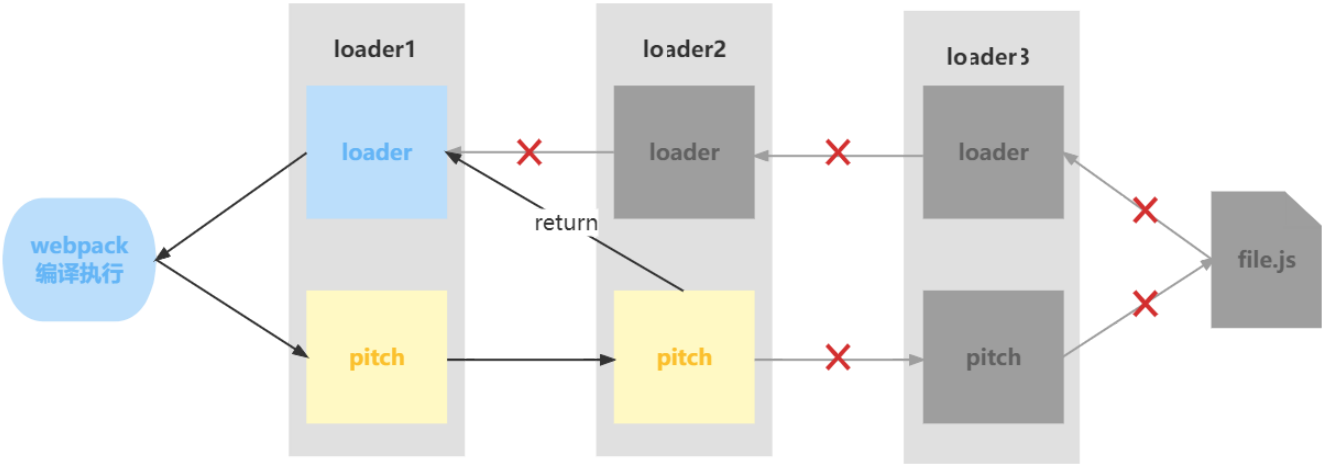
JavaScript 复制代码

```
module.exports = function (content) {
  return content;
};
module.exports.pitch = function (remainingRequest, precedingRequest, data)
{
  console.log("do somethings");
};
```

webpack 会先从左到右执行 loader 链中的每个 loader 上的 pitch 方法（如果有），然后再从右到左执行 loader 链中的每个 loader 上的普通 loader 方法。



在这个过程中如果任何 pitch 有返回值，则 loader 链被阻断。webpack 会跳过后面所有的的 pitch 和 loader，直接进入上一个 loader 。



loader API

方法名	含义	用法
this.async	异步回调 loader。返回 this.callback	const callback = this.async()
this.callback	可以同步或者异步调用的并返回多个结果的函数	this.callback(err, content, sourceMap?, meta?)
this.getOptions(schema)	获取 loader 的 options	this.getOptions(schema)
this.emitFile	产生一个文件	this.emitFile(name, content, sourceMap)

this.utils.contextify	返回一个相对路径	this.utils.contextify(context, request)
this.utils.absolutify	返回一个绝对路径	this.utils.absolutify(context, request)

更多文档，请查阅 webpack 官方 loader api 文档open in new window

手写 clean-log-loader

作用：用来清理 js 代码中的console.log

JavaScript | 复制代码

```
// loaders/clean-log-loader.js
module.exports = function cleanLogLoader(content) {
  // 将console.log替换为空
  return content.replace(/console\.log\(.*\);?/g, "");
};
```

手写 banner-loader

作用：给 js 代码添加文本注释

- loaders/banner-loader/index.js

```JavaScript
const schema = require("./schema.json");

module.exports = function (content) {
  // 获取loader的options，同时对options内容进行校验
  // schema是options的校验规则（符合 JSON schema 规则）
  const options = this.getOptions(schema);

  const prefix = `
    /*
    * Author: ${options.author}
    */
  `;

  return `${prefix} \n ${content}`;
};
```

- loaders/banner-loader/schema.json

```JSON
{
  "type": "object",
  "properties": {
    "author": {
      "type": "string"
    }
  },
  "additionalProperties": false
}
```

手写 babel-loader

作用：编译 js 代码，将 ES6+语法编译成 ES5-语法。

- 下载依赖

```
npm i @babel/core @babel/preset-env -D
```

- loaders/babel-loader/index.js

```javascript
const schema = require("./schema.json");
const babel = require("@babel/core");

module.exports = function (content) {
  const options = this.getOptions(schema);
  // 使用异步loader
  const callback = this.async();
  // 使用babel对js代码进行编译
  babel.transform(content, options, function (err, result) {
    callback(err, result.code);
  });
};
```

- loaders/banner-loader/schema.json

```json
{
  "type": "object",
  "properties": {
    "presets": {
      "type": "array"
    }
  },
  "additionalProperties": true
}
```

手写 file-loader

作用：将文件原封不动输出出去

- 下载包

```
npm i loader-utils -D
```

- loaders/file-loader.js

```
const loaderUtils = require("loader-utils");

function fileLoader(content) {
  // 根据文件内容生产一个新的文件名称
  const filename = loaderUtils.interpolateName(this, "[hash].[ext]", {
    content,
  });
  // 输出文件
  this.emitFile(filename, content);
  // 暴露出去，给js引用。
  // 记得加上''
  return `export default '${filename}'`;
}

// loader 解决的是二进制的内容
// 图片是 Buffer 数据
fileLoader.raw = true;

module.exports = fileLoader;
```

- loader 配置

```
{
  test: /\.(png|jpe?g|gif)$/,
  loader: "./loaders/file-loader.js",
  type: "javascript/auto", // 解决图片重复打包问题
},
```

手写 style-loader

作用：动态创建 style 标签，插入 js 中的样式代码，使样式生效。

- loaders/style-loader.js

JavaScript | 复制代码

```
const styleLoader = () => {};

styleLoader.pitch = function (remainingRequest) {
  /*
    remainingRequest: C:\Users\86176\Desktop\source\node_modules\css-loader\dist\cjs.js!C:\Users\86176\Desktop\source\src\css\index.css
      这里是inline loader用法，代表后面还有一个css-loader等待处理

    最终我们需要将remainingRequest中的路径转化成相对路径，webpack才能处理
      希望得到：../../node_modules/css-loader/dist/cjs.js!./index.css

    所以：需要将绝对路径转化成相对路径
    要求：
      1. 必须是相对路径
      2. 相对路径必须以 ./ 或 ../ 开头
      3. 相对路径的路径分隔符必须是 / ，不能是 \
  */
  const relativeRequest = remainingRequest
    .split("!")
    .map((part) => {
      // 将路径转化为相对路径
      const relativePath = this.utils.contextify(this.context, part);
      return relativePath;
    })
    .join("!");

  /*
    !!${relativeRequest}
      relativeRequest：../../node_modules/css-loader/dist/cjs.js!./index.css
      relativeRequest是inline loader用法，代表要处理的index.css资源，使用css-loader处理
      !!代表禁用所有配置的loader，只使用inline loader。（也就是外面我们style-loader和css-loader）,它们被禁用了，只是用我们指定的inline loader，也就是css-loader

    import style from "!!${relativeRequest}"
      引入css-loader处理后的css文件
      为什么需要css-loader处理css文件，不是我们直接读取css文件使用呢?
      因为可能存在@import导入css语法，这些语法就要通过css-loader解析才能变成一个css文件，否则我们引入的css资源会缺少
    const styleEl = document.createElement('style')
      动态创建style标签
    styleEl.innerHTML = style
      将style标签内容设置为处理后的css代码
    document.head.appendChild(styleEl)
```

```
41          添加到head中生效
42      */
43      const script = `
44        import style from "!!${relativeRequest}"
45        const styleEl = document.createElement('style')
46        styleEl.innerHTML = style
47        document.head.appendChild(styleEl)
48      `;
49
50      // style-loader是第一个loader, 由于return导致熔断，所以其他loader不执行了（不管
    是normal还是pitch）
51      return script;
52    };
53
54    module.exports = styleLoader;
```

Plugin 原理

Plugin 的作用

通过插件我们可以扩展 webpack，加入自定义的构建行为，使 webpack 可以执行更广泛的任务，拥有更强的构建能力。

Plugin 工作原理

> 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 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。 ——「深入浅出 Webpack」

站在代码逻辑的角度就是：webpack 在编译代码过程中，会触发一系列 `Tapable` 钩子事件，插件所做的，就是找到相应的钩子，往上面挂上自己的任务，也就是注册事件，这样，当

webpack 构建的时候，插件注册的事件就会随着钩子的触发而执行了。

Webpack 内部的钩子

什么是钩子

钩子的本质就是：事件。为了方便我们直接介入和控制编译过程，webpack 把编译过程中触发的各类关键事件封装成事件接口暴露了出来。这些接口被很形象地称做：`hooks`（钩子）。开发插件，离不开这些钩子。

Tapable

`Tapable` 为 webpack 提供了统一的插件接口（钩子）类型定义，它是 webpack 的核心功能库。webpack 中目前有十种 `hooks`，在 `Tapable` 源码中可以看到，他们是：

JavaScript | 复制代码

```
// https://github.com/webpack/tapable/blob/master/lib/index.js
exports.SyncHook = require("./SyncHook");
exports.SyncBailHook = require("./SyncBailHook");
exports.SyncWaterfallHook = require("./SyncWaterfallHook");
exports.SyncLoopHook = require("./SyncLoopHook");
exports.AsyncParallelHook = require("./AsyncParallelHook");
exports.AsyncParallelBailHook = require("./AsyncParallelBailHook");
exports.AsyncSeriesHook = require("./AsyncSeriesHook");
exports.AsyncSeriesBailHook = require("./AsyncSeriesBailHook");
exports.AsyncSeriesLoopHook = require("./AsyncSeriesLoopHook");
exports.AsyncSeriesWaterfallHook = require("./AsyncSeriesWaterfallHook");
exports.HookMap = require("./HookMap");
exports.MultiHook = require("./MultiHook");
```

`Tapable` 还统一暴露了三个方法给插件，用于注入不同类型的自定义构建行为：

- `tap`：可以注册同步钩子和异步钩子。
- `tapAsync`：回调方式注册异步钩子。
- `tapPromise`：Promise 方式注册异步钩子。

Plugin 构建对象

Compiler

compiler 对象中保存着完整的 Webpack 环境配置，每次启动 webpack 构建时它都是一个独一无二，仅仅会创建一次的对象。

这个对象会在首次启动 Webpack 时创建，我们可以通过 compiler 对象上访问到 Webapck 的主环境配置，比如 loader 、 plugin 等等配置信息。

它有以下主要属性：

- `compiler.options` 可以访问本次启动 webpack 时候所有的配置文件，包括但不限于 `loaders` 、 `entry` 、 `output` 、 `plugin` 等等完整配置信息。
- `compiler.inputFileSystem` 和 `compiler.outputFileSystem` 可以进行文件操作，相当于 `Nodejs` 中 `fs` 。
- `compiler.hooks` 可以注册 `tapable` 的不同种类 `Hook` ，从而可以在 `compiler` 生命周期中植入不同的逻辑。

compiler hooks 文档open in new window

Compilation

`compilation` 对象代表一次资源的构建， `compilation` 实例能够访问所有的模块和它们的依赖。

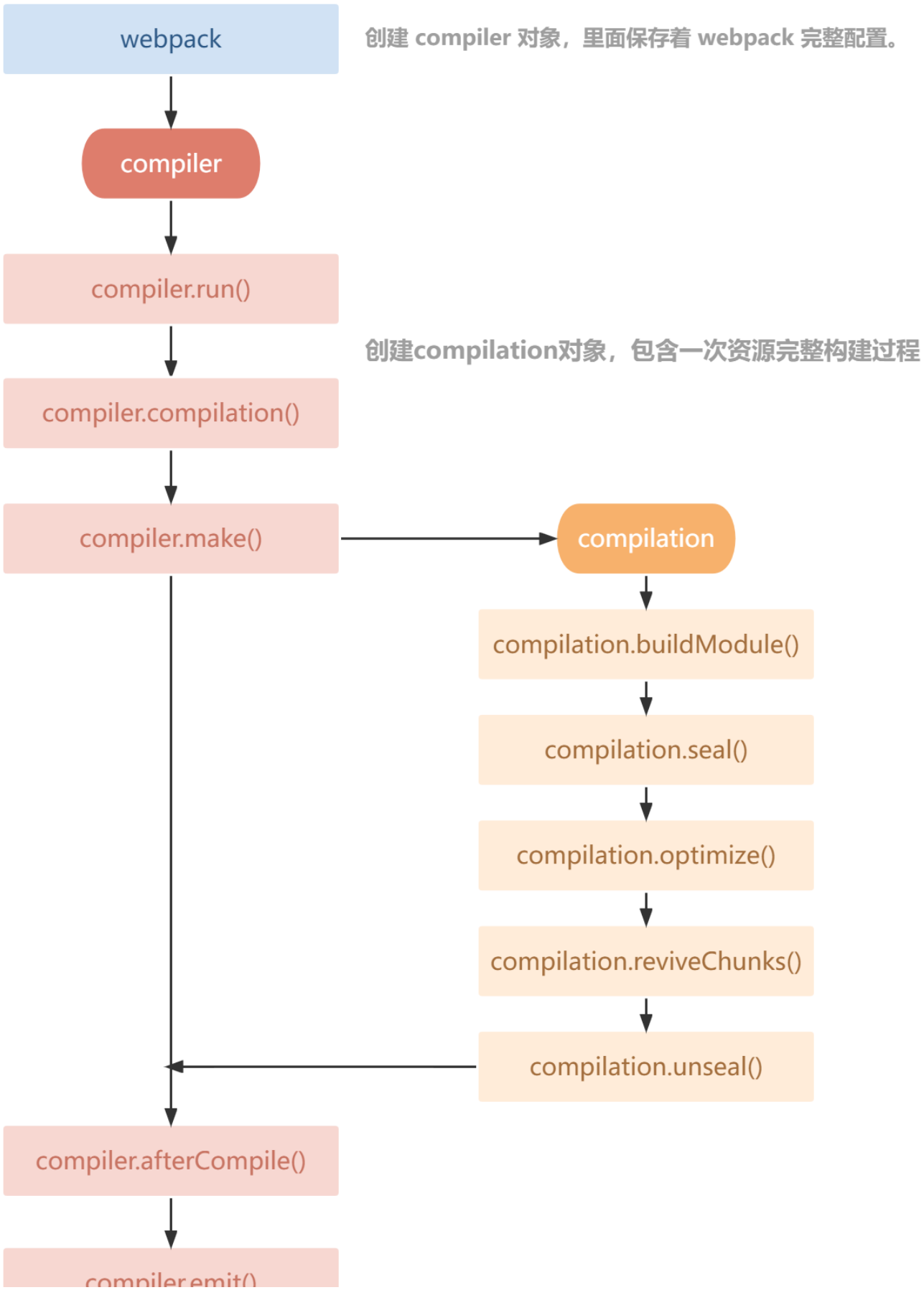
一个 `compilation` 对象会对构建依赖图中所有模块，进行编译。 在编译阶段，模块会被加载(`load`)、封存(`seal`)、优化(`optimize`)、 分块(`chunk`)、哈希(`hash`)和重新创建(`restore`)。

它有以下主要属性：

- `compilation.modules` 可以访问所有模块，打包的每一个文件都是一个模块。
- `compilation.chunks` `chunk` 即是多个 `modules` 组成而来的一个代码块。入口文件引入的资源组成一个 `chunk` ，通过代码分割的模块又是另外的 `chunk` 。
- `compilation.assets` 可以访问本次打包生成所有文件的结果。
- `compilation.hooks` 可以注册 `tapable` 的不同种类 `Hook` ，用于在 `compilation` 编译模块阶段进行逻辑添加以及修改。

compilation hooks 文档open in new window

生命周期简图


webpack
创建 compiler 对象，里面保存着 webpack 完整配置。
compiler
compiler.run()
创建compilation对象，包含一次资源完整构建过程
compiler.compilation()
compiler.make()
compilation
compilation.buildModule()
compilation.seal()
compilation.optimize()
compilation.reviveChunks()
compilation.unseal()
compiler.afterCompile()
compiler.emit()

开发一个插件

最简单的插件

- plugins/test-plugin.js

JavaScript | 复制代码

```
class TestPlugin {
  constructor() {
    console.log("TestPlugin constructor()");
  }
  // 1. webpack读取配置时，new TestPlugin() ，会执行插件 constructor 方法
  // 2. webpack创建 compiler 对象
  // 3. 遍历所有插件，调用插件的 apply 方法
  apply(compiler) {
    console.log("TestPlugin apply()");
  }
}

module.exports = TestPlugin;
```

注册 hook

JavaScript | 复制代码

```javascript
class TestPlugin {
  constructor() {
    console.log("TestPlugin constructor()");
  }
  // 1. webpack读取配置时，new TestPlugin() ，会执行插件 constructor 方法
  // 2. webpack创建 compiler 对象
  // 3. 遍历所有插件，调用插件的 apply 方法
  apply(compiler) {
    console.log("TestPlugin apply()");

    // 从文档可知，compile hook 是 SyncHook，也就是同步钩子，只能用tap注册
    compiler.hooks.compile.tap("TestPlugin", (compilationParams) => {
      console.log("compiler.compile()");
    });

    // 从文档可知，make 是 AsyncParallelHook，也就是异步并行钩子，特点就是异步任务同时执行
    // 可以使用 tap、tapAsync、tapPromise 注册。
    // 如果使用tap注册的话，进行异步操作是不会等待异步操作执行完成的。
    compiler.hooks.make.tap("TestPlugin", (compilation) => {
      setTimeout(() => {
        console.log("compiler.make() 111");
      }, 2000);
    });

    // 使用tapAsync、tapPromise注册，进行异步操作会等异步操作做完再继续往下执行
    compiler.hooks.make.tapAsync("TestPlugin", (compilation, callback) => {
      setTimeout(() => {
        console.log("compiler.make() 222");
        // 必须调用
        callback();
      }, 1000);
    });

    compiler.hooks.make.tapPromise("TestPlugin", (compilation) => {
      console.log("compiler.make() 333");
      // 必须返回promise
      return new Promise((resolve) => {
        resolve();
      });
    });

    // 从文档可知，emit 是 AsyncSeriesHook，也就是异步串行钩子，特点就是异步任务顺序执行
```

```
    compiler.hooks.emit.tapAsync("TestPlugin", (compilation, callback) =>
{
      setTimeout(() => {
        console.log("compiler.emit() 111");
        callback();
      }, 3000);
    });

    compiler.hooks.emit.tapAsync("TestPlugin", (compilation, callback) =>
{
      setTimeout(() => {
        console.log("compiler.emit() 222");
        callback();
      }, 2000);
    });

    compiler.hooks.emit.tapAsync("TestPlugin", (compilation, callback) =>
{
      setTimeout(() => {
        console.log("compiler.emit() 333");
        callback();
      }, 1000);
    });
  }
}

module.exports = TestPlugin;
```

启动调试

通过调试查看 `compiler` 和 `compilation` 对象数据情况。

1. `package.json` 配置指令

JSON 复制代码

```json
{
  "name": "source",
  "version": "1.0.0",
  "scripts": {
    "debug": "node --inspect-brk ./node_modules/webpack-cli/bin/cli.js"
  },
  "keywords": [],
  "author": "xiongjian",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.17.10",
    "@babel/preset-env": "^7.17.10",
    "css-loader": "^6.7.1",
    "loader-utils": "^3.2.0",
    "webpack": "^5.72.0",
    "webpack-cli": "^4.9.2"
  }
}
```

1. 运行指令

Plain Text 复制代码

```
npm run debug
```

此时控制台输出以下内容：

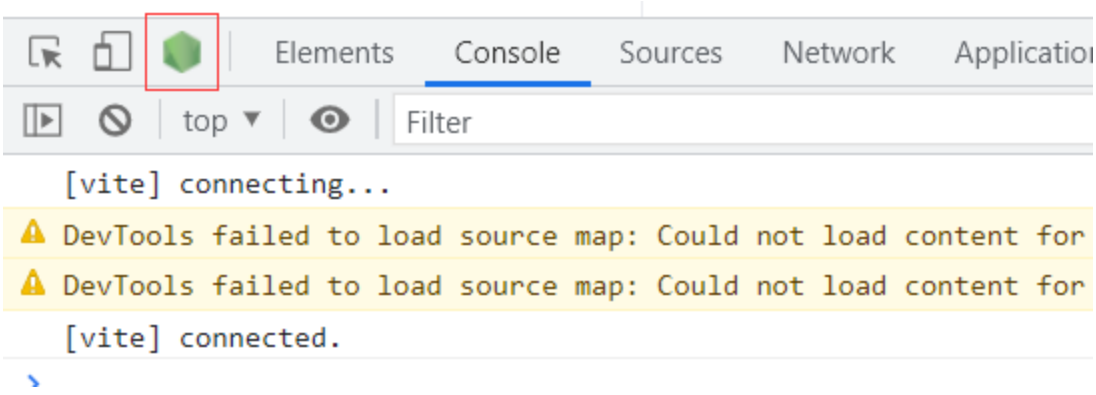
```
PS C:\Users\86176\Desktop\source> npm run debug

> source@1.0.0 debug
> node --inspect-brk ./node_modules/webpack-cli/bin/cli.js

Debugger listening on ws://127.0.0.1:9229/629ea097-7b52-4011-93a7-02f83c75c797
For help, see: https://nodejs.org/en/docs/inspecto
```

2. 打开 `Chrome` 浏览器，F12 打开浏览器调试控制台。

此时控制台会显示一个绿色的图标



1. 点击绿色的图标进入调试模式。
2. 在需要调试代码处用 `debugger` 打断点，代码就会停止运行，从而调试查看数据情况。

BannerWebpackPlugin

1. 作用：给打包输出文件添加注释。
2. 开发思路:

- 需要打包输出前添加注释：需要使用 compiler.hooks.emit 钩子, 它是打包输出前触发。
- 如何获取打包输出的资源？compilation.assets 可以获取所有即将输出的资源文件。

1. 实现：

JavaScript | 复制代码

```javascript
// plugins/banner-webpack-plugin.js
class BannerWebpackPlugin {
  constructor(options = {}) {
    this.options = options;
  }

  apply(compiler) {
    // 需要处理文件
    const extensions = ["js", "css"];

    // emit是异步串行钩子
    compiler.hooks.emit.tapAsync("BannerWebpackPlugin", (compilation, callback) => {
      // compilation.assets包含所有即将输出的资源
      // 通过过滤只保留需要处理的文件
      const assetPaths = Object.keys(compilation.assets).filter((path) => {
        const splitted = path.split(".");
        return extensions.includes(splitted[splitted.length - 1]);
      });

      assetPaths.forEach((assetPath) => {
        const asset = compilation.assets[assetPath];

        const source = `/*
* Author: ${this.options.author}
*/\n${asset.source()}`;

        // 覆盖资源
        compilation.assets[assetPath] = {
          // 资源内容
          source() {
            return source;
          },
          // 资源大小
          size() {
            return source.length;
          },
        };
      });

      callback();
    });
  }
}
```

```
44
45
    module.exports = BannerWebpackPlugin;
```

CleanWebpackPlugin

1. 作用：在 webpack 打包输出前将上次打包内容清空。
2. 开发思路：

- 如何在打包输出前执行？需要使用 compiler.hooks.emit 钩子，它是打包输出前触发。
- 如何清空上次打包内容？
    - 获取打包输出目录：通过 compiler 对象。
    - 通过文件操作清空内容：通过 compiler.outputFileSystem 操作文件。

1. 实现：

JavaScript | 复制代码

```javascript
// plugins/clean-webpack-plugin.js
class CleanWebpackPlugin {
  apply(compiler) {
    // 获取操作文件的对象
    const fs = compiler.outputFileSystem;
    // emit是异步串行钩子
    compiler.hooks.emit.tapAsync("CleanWebpackPlugin", (compilation, callback) => {
      // 获取输出文件目录
      const outputPath = compiler.options.output.path;
      // 删除目录所有文件
      const err = this.removeFiles(fs, outputPath);
      // 执行成功err为undefined，执行失败err就是错误原因
      callback(err);
    });
  }

  removeFiles(fs, path) {
    try {
      // 读取当前目录下所有文件
      const files = fs.readdirSync(path);

      // 遍历文件，删除
      files.forEach((file) => {
        // 获取文件完整路径
        const filePath = `${path}/${file}`;
        // 分析文件
        const fileStat = fs.statSync(filePath);
        // 判断是否是文件夹
        if (fileStat.isDirectory()) {
          // 是文件夹需要递归遍历删除下面所有文件
          this.removeFiles(fs, filePath);
        } else {
          // 不是文件夹就是文件，直接删除
          fs.unlinkSync(filePath);
        }
      });

      // 最后删除当前目录
      fs.rmdirSync(path);
    } catch (e) {
      // 将产生的错误返回出去
      return e;
    }
  }
```

```
45  }
46
47  module.exports = CleanWebpackPlugin;
```

AnalyzeWebpackPlugin

1. 作用：分析 webpack 打包资源大小，并输出分析文件。
2. 开发思路:

- 在哪做? compiler.hooks.emit, 它是在打包输出前触发，我们需要分析资源大小同时添加上分析后的 md 文件。

1. 实现：

JavaScript | 复制代码

```javascript
// plugins/analyze-webpack-plugin.js
class AnalyzeWebpackPlugin {
  apply(compiler) {
    // emit是异步串行钩子
    compiler.hooks.emit.tap("AnalyzeWebpackPlugin", (compilation) => {
      // Object.entries将对象变成二维数组。二维数组中第一项值是key，第二项值是value
      const assets = Object.entries(compilation.assets);

      let source = "# 分析打包资源大小 \n| 名称 | 大小 |\n| --- | --- |";

      assets.forEach(([filename, file]) => {
        source += `\n| ${filename} | ${file.size()} |`;
      });

      // 添加资源
      compilation.assets["analyze.md"] = {
        source() {
          return source;
        },
        size() {
          return source.length;
        },
      };
    });
  }
}

module.exports = AnalyzeWebpackPlugin;
```

InlineChunkWebpackPlugin

1. 作用：webpack 打包生成的 runtime 文件太小了，额外发送请求性能不好，所以需要将其内联到 js 中，从而减少请求数量。
2. 开发思路:

- 我们需要借助 html-webpack-plugin 来实现
  - 在 html-webpack-plugin 输出 index.html 前将内联 runtime 注入进去

    - 删除多余的 runtime 文件
- 如何操作 html-webpack-plugin? 官方文档open in new window

1. 实现：

JavaScript | 复制代码

```javascript
// plugins/inline-chunk-webpack-plugin.js
const HtmlWebpackPlugin = require("safe-require")("html-webpack-plugin");

class InlineChunkWebpackPlugin {
  constructor(tests) {
    this.tests = tests;
  }

  apply(compiler) {
    compiler.hooks.compilation.tap("InlineChunkWebpackPlugin", (compilation) => {
      const hooks = HtmlWebpackPlugin.getHooks(compilation);

      hooks.alterAssetTagGroups.tap("InlineChunkWebpackPlugin", (assets) => {
        assets.headTags = this.getInlineTag(assets.headTags, compilation.assets);
        assets.bodyTags = this.getInlineTag(assets.bodyTags, compilation.assets);
      });

      hooks.afterEmit.tap("InlineChunkHtmlPlugin", () => {
        Object.keys(compilation.assets).forEach((assetName) => {
          if (this.tests.some((test) => assetName.match(test))) {
            delete compilation.assets[assetName];
          }
        });
      });
    });
  }

  getInlineTag(tags, assets) {
    return tags.map((tag) => {
      if (tag.tagName !== "script") return tag;

      const scriptName = tag.attributes.src;

      if (!this.tests.some((test) => scriptName.match(test))) return tag;

      return { tagName: "script", innerHTML: assets[scriptName].source(), closeTag: true };
    });
  }
}

```

```
41  module.exports = InlineChunkWebpackPlugin;
```