Webpack 的 Compiler 和 Compilation

更新时间：2019-06-27 09:57:45

> 与有肝胆人共事，从无字句处读书。
>
> ——周恩来

在 Webpack 工作流程 文章中已经提到过，Compiler 和 Compilation 都是继承自Tapable ，不同点是`Compiler`是每个 Webpack 的配置，对应一个`Compiler`对象，记录着整个 Webpack 的生命周期；在构建的过程中，每次构建都会产生一次`Compilation`，`Compilation`则是构建周期的产物。本文将进一步介绍`Compiler`和`Compilation`，通过本章节的学习，将会对 Webpack 构建过程有一个更加宏观的掌握。

Compiler

`Compiler` 模块是 Webpack 最核心的模块。每次执行 Webpack 构建的时候，在 Webpack 内部，会首先实例化一个 Compiler 对象，然后调用它的 run 方法来开始一次完整的编译过程。我们直接使用 Webpack API `webpack(options)`的方式得到的就是一个`Compiler`实例化的对象，这时候 Webpack 并不会立即开始构建，需要我们手动执行`compiler.run()`才可以。

```
const webpack = require('webpack');
const webpackConfig = require('./webpack.config.js');

// 只传入 config
const compiler = webpack(webpackConfig);
// 开始执行
compiler.run(callback);

// 上面两句等价于
webpack(webpackConfig, callback);
```

> Tips：使用 webpack-dev-server API 方式时，只需要传入 compiler 对象给 dev server 即可，不需要手动执行`compiler.run()`。

我们如果要手动实例化一个`Compiler`对象，可以通过`const Compiler = webpack.Compiler`来获取它的类，一般只有一个父`Compiler`，而子`Compiler`可以用来处理一些特殊的事件。

在 webpack plugin 中，每个插件都有个`apply`方法。这个方法接收到的参数就是`Compiler`对象，我们可以通过在对应的钩子时机绑定处理函数来编写插件，下面主要介绍下`Compiler`对象的钩子。

Compiler 钩子

在Webpack 工作流程中，我们通过下面的代码，获取了对应的钩子名称：

```
const compiler = webpack(config);
// 遍历hooks，添加回调，输出`hookName`
Object.keys(compiler.hooks).forEach(hookName => {
    if (compiler.hooks[hookName].tap) {
        compiler.hooks[hookName].tap('anyString', () => {
            console.log(`run -> ${hookName}`);
        });
    }
});
// 触发webpack的编译流程
compiler.run();
```

得到`compiler.run()`之后的工作流程：

```
run -> beforeRun
run -> run
run -> normalModuleFactory
run -> contextModuleFactory
run -> beforeCompile
run -> compile
run -> thisCompilation
run -> compilation
run -> make
run -> afterCompile
run -> shouldEmit
run -> emit
run -> afterEmit
run -> done
```

上面的方式只是输出`compiler.run()`之后的一部分钩子，`Compiler`还有好多钩子。比如在 watch 模式下，还会有`watchRun`、`watchClose`和`invalid`。我们如果要绑定某个钩子，则可以使用下面的方法来绑定：

```
compiler.hooks.someHook.tap('MyPlugin', params => {
    /* ... */
});
```

下面通过一个表格来说明下对应的关系：

钩子名	Tapable 类型	触发时机	传入 callback 的参数
entryOption	SyncBailHook	在 webpack 中的 entry 配置处理过之后	`context`，`entry`
afterPlugins	SyncHook	初始化完内置插件之后	`compiler`
afterResolvers	SyncHook	resolver 完成之后（后面解释resolver是什么）	`compiler`
environment	SyncHook	准备编译环境，webpack plugins配置初始化完成之后	`compiler`
afterEnvironment	SyncHook	编译环境准备好之后	`compiler`
beforeRun	AsyncSeriesHook	开始正式编译之前	`compiler`
run	AsyncSeriesHook	开始编译之后，读取 records 之前；监听模式触发`watch-run`	`compiler`
watchRun	AsyncSeriesHook	监听模式下，一个新的编译触发之后	`compiler`

钩子名	Tapable 类型	触发时机	传入 callback 的参数
normalModuleFactory	SyncHook	NormalModuleFactory 创建之后	`normalModuleFactory` 实例
contextModuleFactory	SyncHook	ContextModuleFactory 创建之后	`contextModuleFactory` 实例
beforeCompile	AsyncSeriesHook	compilation 实例化需要的参数创建完毕之后	`compilationParams`
compile	SyncHook	一次 compilation 编译创建之前	`compilationParams`
thisCompilation	SyncHook	触发 compilation 事件之前执行	`compilation`，`compilationParams`
compilation	SyncHook	compilation创建成功之后	`compilation`，`compilationParams`
make	AsyncParallelHook	完成编译之前	`compilation`
afterCompile	AsyncSeriesHook	完成编译和封存（seal）编译产出之后	`compilation`
shouldEmit	SyncBailHook	发布构建后资源之前触发，回调必须返回`true`/`false`，`true`则继续	`compilation`
emit	AsyncSeriesHook	生成资源到 output 目录之前	`compilation`
afterEmit	AsyncSeriesHook	生成资源到 output 目录之后	`compilation`
done	AsyncSeriesHook	compilation完成之后	`stats`
failed	SyncHook	compilation失败	`error`
invalid	SyncHook	监听模式下，编译无效时	`fileName`，`changeTime`
watchClose	SyncHook	监听模式停止	无

> Tips：整个`Compiler`完整地展现了 Webpack 的构建流程：
>
> - 准备阶段：`make`之前做的事情都属于准备阶段，这阶段的`calback`入参以`compiler`为主；
> - 编译阶段：这阶段以`compilation`的钩子为主，`calback`入参以`compilation`为主；
> - 产出阶段：这阶段从`compilation`开始，最后回到`Compiler`钩子上，`calback`传入参数是跟结果相关的数据，包括`stats`、`error`。

注解1：Resolver

Compiler 的 Resolver 是指来自于enhanced-resolve模块，它主要功能是一个提供异步`require.resolve()`，即从哪里去查找文件的路径，可以通过Webpack的`resolve`和`resolveLoader`来配置。Compiler类有三种类型的内置Resolver：

- Normal：通过绝对路径或相对路径，解析一个模块；
- Context：通过给定的上下文（context）解析一个模块；
- Loader：解析一个 webpack loader。

注解2：`thisCompilation`和`compilation`

这里为什么会有`thisCompilation`和`compilation`两个钩子呢？其实是跟子编译（child compiler）有关，Compiler 实例通过`createChildCompiler`方法可以创建子编译实例`childCompiler`。创建`childCompiler`时，`childCompiler`会复制 compiler 实例的任务点监听器。`compilation`**的钩子会被复制，而**`thisCompilation`**钩子则不会被复制**。

Compilation

在 Compilation 阶段，模块会被加载(loaded)、封存(sealed)、优化(optimized)、分块(chunked)、哈希(hashed)和重新创建(restored)，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 以监听（watch）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。

Compilation 钩子

在 Compilation 中处理的对象分别是`module`、`chunk`、`asset`，由 modules 组成`chunks`，由`chunks`生成`assets`，处理顺序是：`module → modules → chunks → assets`，先从单个 module 开始处理，查找依赖关系，最后完成单个 module 处理，完成全部modules 之后，开始 chunks 阶段处理，最后在根据优化配置，按需生成 assets。

所以整个 Compilation 的生命周期钩子虽然比较多，但是大规律上是围绕这个顺序进行的，下面是大部分钩子表格如下，其中带★的是比较重要的钩子：

钩子名	Tapable 类型	触发时机	传入 callback 的参数
buildModule	SyncHook	在模块构建开始之前触发	`module`
rebuildModule	SyncHook	在重新构建一个模块之前触发	`module`
failedModule	SyncHook	模块构建失败时执行	`module`，`error`
succeedModule	SyncHook	模块构建成功时执行	`module`
finishModules	SyncHook	所有模块都完成构建	`module`
finishRebuildingModule	SyncHook	一个模块完成重新构建	`module`
seal	SyncHook	★编译（compilation）停止接收新模块时触发	`module`
unseal	SyncHook	编译（compilation）开始接收新模块时触发	`module`
optimizeDependencies	SyncBailHook	依赖优化开始时触发	`modules`
afterOptimizeDependencies	SyncHook	依赖优化结束时触发	`modules`
optimize	SyncHook	★优化阶段开始时触发	`modules`
optimizeModules	SyncBailHook	★模块的优化	`modules`
afterOptimizeModules	SyncHook	模块优化结束时触发	`modules`
optimizeChunks	SyncBailHook	★优化 chunks	`chunks`
afterOptimizeChunks	SyncHook	chunk 优化完成之后触发	`chunks`
optimizeTree	AsyncSeriesHook	异步优化依赖树	`chunks`，`modules`
afterOptimizeTree	SyncHook	异步优化依赖树完成时	`chunks`，`modules`
optimizeChunkModules	SyncBailHook	优化单个chunk中的 modules 开始	`chunks`
afterOptimizeChunkModules	SyncHook	优化单个chunk中的 modules 结束	`chunks`
shouldRecord	SyncHook		`chunks`
reviveModules	SyncHook	从 records 中恢复模块信息	`modules`，`records`
optimizeModuleOrder	SyncHook	将模块从最重要的到最不重要的进行排序	`chunks`
beforeModuleIds	SyncHook	处理 modulesId 之前	`modules`
moduleIds	SyncHook	处理 modulesId	`modules`
optimizeModuleIds	SyncHook	优化 modulesId	`chunks`
afterOptimizeModuleIds	SyncHook	优化 modulesId之后	`chunks`
reviveChunks	SyncHook	从 records 中恢复 chunk 信息	`modules`，`records`
optimizeChunkOrder	SyncHook	将 chunk 从最重要的到最不重要的进行排序	`chunks`
beforeOptimizeChunkIds	SyncHook	chunk id 优化之前触发	`chunks`
optimizeChunkIds	SyncHook	chunk id 优化开始触发	`chunks`
afterOptimizeChunkIds	SyncHook	chunk id 优化结束触发	`chunks`
recordModules	SyncHook	将模块信息存储到 records	`modules`，`records`
recordChunks	SyncHook	将 chunk 信息存储到 records	`chunks`，`records`
beforeHash	SyncHook	在编译被哈希（hashed）之前	-

钩子名	Tapable 类型	触发时机	传入 callback 的参数
afterHash	SyncHook	在编译被哈希（hashed）之后	-
record	SyncHook	将 compilation 相关信息存储到 records 中	`compilation`，`records`
beforeChunkAssets	SyncHook	在创建 chunk 资源（asset）之前	`chunks`
additionalChunkAssets	SyncHook	为 chunk 创建附加资源（asset）	`chunks`
additionalAssets	AsyncSeriesHook	★为编译（compilation）创建附加资源（asset）	-
optimizeChunkAssets	AsyncSeriesHook	★优化所有 chunk 资源（asset）	`chunks`
afterOptimizeChunkAssets	SyncHook	chunk 资源（asset）已经被优化	`chunks`
optimizeAssets	AsyncSeriesHook	★优化存储在 compilation.assets 中的所有资源（asset）	`assets`
afterOptimizeAssets	SyncHook	优化compilation.assets 中的所有资源（asset）之后	`assets`
moduleAsset	SyncHook	一个模块中的一个资源被添加到编译中	`module`，`filename`
chunkAsset	SyncHook	一个 chunk 中的一个资源被添加到编译中	`chunk`，`filename`
assetPath	SyncWaterfallHook	asset 路径确认之后	`filename`，`data`
childCompiler	SyncHook	子编译（compiler）触发	`childCompiler`，`compilerName`，`compilerIndex`
normalModuleLoader	SyncHook	★普通模块 loader，真正（一个接一个地）加载模块图（graph）中所有模块的函数	`loaderContext`，`module`

Compilation 和 Stats 对象

在 Webpack 的回调函数中会得到`stats`对象。这个对象实际来自于`Compilation.getStats()`，返回的是主要含有`modules`、`chunks`和`assets`三个属性值的对象。

- modules：记录了所有解析后的模块；
- chunks：记录了所有chunk；
- assets：记录了所有要生成的文件。

Stats对象本质上来自于`lib/Stats.js`的类实例，常用的方法`stats.hasWarnings()`、`stats.hasErrors()`、`stats.toJson()`和`stats.toString()`都可以在这个类里面找到对应的实现。

Stats 对象数据结构

Stats 对象的 JSON 数据结构，除了通过`Compilation.getStats()`获得，还可以在 webpack 回调中，通过 callback 参数获得：

```
webpack(config, (err, stats) => {
    console.log(stats.toJson());
});
```

还可以通过 webpack-cli 的选项将本次打包的 stats 存入一个 json 文件：`webpack --profile --json > compilation-stats.json`。

`stats.toJson()`得到的数据结构格式如下：

```
{
    "version": "4.29.6", // 用来编译的 webpack 的版本
    "hash": "11593e3b3ac85436984a", // 编译使用的 hash
    "time": 2469, // 编译耗时 (ms)
    "filteredModules": 0, // 当 `exclude`传入`toJson` 函数时，统计被无视的模块的数量
    "outputPath": "/", // path to webpack 输出目录的 path 路径
    "assetsByChunkName": {
        // 用作映射的 chunk 的名称
        "main": "web.js?h=11593e3b3ac85436984a",
        "named-chunk": "named-chunk.web.js",
        "other-chunk": ["other-chunk.js", "other-chunk.css"]
    },
    "assets": [
        // asset 对象 (asset objects) 的数组
    ],
    "chunks": [
        // chunk 对象 (chunk objects) 的数组
    ],
    "modules": [
        // 模块对象 (module objects) 的数组
    ],
    "errors": [
        // 错误字符串 (error string) 的数组
    ],
    "warnings": [
        // 警告字符串 (warning string) 的数组
    ]
}
```

这里面除了编译基本信息、报错和 warnings 之外，就是`modules`、`chunks`和`assets`三个对应的数组，里面是单个的`module`、`chunk`和`asset`。

> Tips：`stats.toJson`可以指定要不要输出对应的数据。例如不想输出`modules`和`chunks`，可以使用`stats.toJson({modules: false, chunks: false})`。

module

在每个 module 中，我们可以得到它的所有信息，这些信息可以分为四大类：

1. 基本信息：包括最基本的内容、大小、id；
2. 依赖关系：`module.reasons`对象描述了这个模块被加入依赖图表的理由，包含了引入的方式、引入的 module 信息及其对应代码在第几行第几列等，可以通过这个计算出 module 之间的依赖关系图表（graph）；
3. chunks 和 assets 关系：`module.chunks`和`module.assets`包含到 chunks 和 assets 中的对应 id 等；
4. 被 webpack 处理的后的信息：包含`module.failed`、`module.errors`、`module.warnings`等。

```
{
    "assets": [
        // asset对象 (asset objects)的数组
    ],
    "built": true, // 表示这个模块会参与 Loaders , 解析, 并被编译
    "cacheable": true, // 表示这个模块是否会被缓存
    "chunks": [
        // 包含这个模块的 chunks 的 id
    ],
    "errors": 0, // 处理这个模块发现的错误的数量
    "failed": false, // 编译是否失败
    "id": 0, // 这个模块的ID (类似于 `module.id`)
    "identifier": "(webpack)\\test\\browsertest\\lib\\index.web.js", // webpack内部使用的唯一的标识
    "name": "./lib/index.web.js", // 实际文件的地址
    "optional": false, // 每一个对这个模块的请求都会包裹在 `try... catch` 内 (与ESM无关)
    "prefetched": false, // 表示这个模块是否会被 prefetched
    "profile": {
        // 有关 `--profile` flag 的这个模块特有的编译数据 (ms)
        "building": 73, // 载入和解析
        "dependencies": 242, // 编译依赖
        "factory": 11 // 解决依赖
    },
    "reasons": [
        // 见下文描述
    ],
    "size": 3593, // 预估模块的大小 (byte)
    "source": "// Should not break it...\r\nif(typeof...", // 字符串化的输入
    "warnings": 0 // 处理模块时警告的数量
}
```

其中`module.reasons`数据结构如下：

```
{
    "loc": "33:24-93", // 导致这个被加入依赖图标的代码行数
    "module": "./lib/index.web.js", // 所基于模块的相对地址 context
    "moduleId": 0, // 模块的 ID
    "moduleIdentifier": "(webpack)\\test\\browsertest\\lib\\index.web.js", // 模块的地址
    "moduleName": "./lib/index.web.js", // 可读性更好的模块名称 (用于 "更好的打印 (pretty-printing)")
    "type": "require.context", // 使用的请求的种类 (type of request)
    "userRequest": "../../cases" // 用来 `import` 或者 `require` 的源字符串
}
```

chunk

在每个 chunk 中，信息也可以分为四大类：

1. 基本信息：包括最基本的内容、大小、id；
2. 来源：`chunk.origins`对象描述了这个模块被加入的理由，包含了引入的方式、引入的 module 信息及其对应代码在第几行第几列等，可以通过这个计算出 module 之间的依赖关系图表（graph）；
3. 引用关系：`chunk.parents`和`chunk.children`被引用和引用的 ids；
4. 包含和被包含：`chunk.files`和`chunk.modules`包含到 assets 和自己包含 modules 中信息等。

```
{
    "entry": true, // 表示这个 chunk 是否包含 webpack 的运行时
    "files": [
        // 一个包含这个 chunk 的文件名的数组
    ],
    "filteredModules": 0, // 见上文的 结构
    "id": 0, // 这个 chunk 的id
    "initial": true, // 表示这个 chunk 是开始就要加载还是 懒加载(lazy-loading)
    "modules": [
        // 模块对象 (module objects)的数组
        "web.js?h=11593e3b3ac85436984a"
    ],
    "names": [
        // 包含在这个 chunk 内的 chunk 的名字的数组
    ],
    "origins": [
        // 下文详述
    ],
    "parents": [], // 父 chunk 的 ids
    // 生成 assets 的原因
    "reason": "split chunk (cache group: asyncVendors) (name: async)",
    "hash": "170746935298270ad813",
    // 自己引用谁
    "children": [],
    // 引用的顺序
    "childrenByOrder": {},
    "modules": [],
    "rendered": true, // 表示这个 chunk 是否会参与进编译
    "size": 188057 // chunk 的大小(byte)
}
```

`chunk.origins`对应的格式如下：

```
{
    "loc": "", // 具体是哪行生成了这个chunk
    "module": "(webpack)\\test\\browsertest\\lib\\index.web.js", // 模块的位置
    "moduleId": 0, // 模块的ID
    "moduleIdentifier": "(webpack)\\test\\browsertest\\lib\\index.web.js", // 模块的地址
    "moduleName": "./lib/index.web.js", // 模块的相对地址
    "name": "main", // chunk的名称
    "reasons": [
        // 模块对象中`reason`的数组
    ]
}
```

asset

`asset`相对简单一些，内容如下：

```
{
    "chunkNames": [], // 这个 asset 包含的 chunk
    "chunks": [10, 6], // 这个 asset 包含的 chunk 的 id
    "emitted": true, // 表示这个 asset 是否会让它输出到 output 目录
    "name": "10.web.js", // 输出的文件名
    "size": 1058 // 文件的大小
}
```

总结

本小节主要介绍 Webpack 中两个核心的类 Compiler 和 Compilation。Compiler 是每次 Webpack 全部生命周期的对象，而 Compilation 是 Webpack 中每次构建过程的生命周期对象，Compilation 是通过 Compiler 创建的实例。两个类都有自己生命周期，即有自己不同的 Hook，通过添加对应 Hook 事件，可以拿到各自生命周期关键数据和对象。Compilation 有个很重要的对象是 Stats 对象，通过这个对象可以得到 Webpack 打包后的所有module、chunk 和 assets 信息，通过分析Stats 对象可以得到很多有用的信息，比如 webpack-bundle-analyzer这类分析打包结果的插件都是通过分析 Stats 对象来得到分析报告的。另外 Webpack 中`lib/Stats.js`的源码也可以看下，对于分析打包结果和编写插件都有很大的启发。

> 本小节 Webpack 相关面试题：
>
> 1. Webpack 的 Compiler 和 Compilation 对象有什么区别？
> 2. Webpack 的 Compiler 和 Compilation 对象有哪些重要的 Hooks？
> 3. 怎么获取 Webpack 的 Compiler 和 Compilation 对象？
> 4. Webpack 的 Stats 对象可以做什么？有什么用？怎么获取？

Tapable —— Webpack 的核心模块

Webpack 工作流程

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论