Webpack 工作流程

更新时间：2019-06-28 09:40:17

> 机会不会上门来找人，只有人去找机会。
>
> ——狄更斯

Webpack 主要工作是从一个入口开始，将小块独立的代码编制成更大而复杂的可以运行在浏览器中的代码，独立的代码就是一些 JavaScript 及其它可以被 JavaScript 引用的文件。今天章节来理解下 Webpack 的工作流程和基本原理。为了方便理解，我先将 Webpack 整个工作流程打个比方：

> Webpack 可以看做是一个工厂车间，`plugin`和`loader`是车间中的两类机器，工厂有一个车间主任和一个生产车间。车间主任叫`Compiler`，负责指挥生产车间机器`Compilation`进行生产劳动，`Compilation`会首先将进来的原材料（`entry`）使用一种叫做`loader`的机器进行加工，生产出来的产品就是`Chunk`；`Chunk`生产出来之后，会被组装成`Bundle`，然后通过一类`plugin`的机器继续加工，得到最后的`Bundle`，然后运输到对应的仓库（output）。这个工厂的生产线就是 Tapable，厂子运作的整个流程都是生产线控制的，车间中有好几条生产线，每个生产线有很多的操作步骤（`hook`），一步操作完毕，会进入到下一步操作，直到生产线全流程完成，再将产出传给下一个产品线处理。整个车间生产线也组成了一条最大的生产线。

上面的例子揭示了整个 Webpack 工作流程，其中我们可以看到我们配置的`webpack.config.js`当中的`entry`和`output`，也可以看到我们配置的`loader`。

> Tips：为了更好的理解 Webpack 的工作原理，推荐阅读下minipack这个项目，中文版本的在这里。

基本流程

Webpack 的基本流程可以分为三个阶段：

- 准备阶段：主要任务是创建 `Compiler` 和 `Compilation` 对象；
- 编译阶段：这个阶段任务是完成 modules 解析，并且生成 chunks；
- module 解析：包含了三个主要步骤，创建实例、loaders 应用和依赖收集；
- chunks 生成，主要步骤是找到每个 `chunk` 所需要包含的 `modules`。
- 产出阶段：这个阶段的主要任务是根据 `chunks` 生成最终文件，主要有三个步骤：模板 Hash 更新，模板渲染 chunk，生成文件。

细化到具体的代码层次，大概可以分为：

1. 初始化参数：包括从配置文件和 shell 中读取和合并参数，然后得出最终参数；**shell 中的参数要优于配置文件的**；
2. 使用上一步得到的参数实例化一个 Compiler 类，注册所有的插件，给对应的 Webpack 构建生命周期绑定 Hook；
3. 开始编译：执行 Compiler 类的 run 方法开始执行编译；
4. `compiler.run` 方法调用 `compiler.compile`，在 `compile` 内实例化一个 `Compilation` 类，`Compilation` 是做构建打包的事情，主要事情包括：
   1）查找入口：根据 entry 配置，找出全部的入口文件；
   2）编译模块：根据文件类型和 loader 配置，使用对应 loader 对文件进行转换处理；
   3）解析文件的 AST 语法树；
   4）找出文件依赖关系；
   5）递归编译依赖的模块。
5. 递归完后得到每个文件的最终结果，根据 entry 配置生成代码块 chunk；
6. 输出所有 chunk 到对应的 `output` 路径。

> Tips: shell 中的参数要优于配置文件。举例说明：配置文件指定了 mode 是 `development`，而 shell 中传入了 `--mode production`，则最终 mode 值为 `production`。

在 Webpack 工作流程里，`Tapable` 始终贯穿其中，Tapable 各种 Hook（钩子）组成了 Webpack 的生命周期。Tapable Hook 和生命周期的关系为：

- Hook：钩子，对应 Tapable 的 Hook；
- 生命周期：Webpack 的执行流程，钩子实际就是生命周期，一般类似 entryOption 的 Hook，在生命周期中 `entry-option`。

参与 Webpack 流程的两个重要模块是：`Compiler` 和 `Compilation`。关于 `Compiler` 和 `Compilation` 这里先简单做下介绍，本文主要讲解 Webpack 的工作流程，它俩在后续章节会继续详细解释。

Compiler

`Compiler` 继承自 `Tapable`，是 Webpack 的整个生命周期管理，代表了完整的 Webpack 环境配置。每个 Webpack 的配置，对应一个 `Compiler` 对象，记录了 Webpack 的 `options`、`loader` 和 `plugin` 等信息，并且通过 `Tapable` 的 Hook 机制管理整个打包流程的生命周期。

Compilation

`Compilation` 也继承自 `Tapable`，代表了一次资源版本构建，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。每次构建过程都会产生一次 `Compilation`，比如我们启动 watch 功能的时候，每当检测到一个文件变化，就会重新创建一个新的 `Compilation`，从而生成一组新的编译资源。

> Tips：Webpack 的插件是在 `apply` 方法接收 `Compiler` 对象来给某个流程添加钩子回调，钩子回调函数接收的是记录当前状态的 `Compilation` 对象，后面 plugin 小节内容继续做介绍。

Compiler 和 Compilation 关系

- Compiler：代表的是不变的 Webpack 环境，是针对 Webpack 的。例如 watch 模式下，传入的 Webpack 配置是不变的，不管执行几次 Compilation 都不变；
- Compilation：针对的是随时可变的项目文件，只要文件有改动，Compilation 就会被重新创建。

Webpack 流程源码解析

准备阶段

当我们开始运行 Webpack 的时候，就会创建 `Compiler` 实例并且加载内置插件。这里跟构建流程相关性比较大的内置插件是 `EntryOptionPlugin`，它会解析传给 Webpack 的配置中的 `entry`。这里**不同类型的 entry**包括：`SingleEntryPlugin`、`MultiEntryPlugin`、`DynamicEntryPlugin` 三类，分别对应着单文件入口、多文件入口和动态文件入口（忘记的翻下 Webpack 基础概念里面 entry 部分的内容），对应代码如下：

```
// webpack 4.29.6
// lib/EntryOptionPlugin.js
const itemToPlugin = (context, item, name) => {
    if (Array.isArray(item)) {
        return new MultiEntryPlugin(context, item, name);
    }
    return new SingleEntryPlugin(context, item, name);
};

module.exports = class EntryOptionPlugin {
    apply(compiler) {
        compiler.hooks.entryOption.tap('EntryOptionPlugin', (context, entry) => {
            // 看这里，判断webpack.config中entry的类型判断，然后选择对应的Entry
            if (typeof entry === 'string' || Array.isArray(entry)) {
                itemToPlugin(context, entry, 'main').apply(compiler);
            } else if (typeof entry === 'object') {
                for (const name of Object.keys(entry)) {
                    itemToPlugin(context, entry[name], name).apply(compiler);
                }
            } else if (typeof entry === 'function') {
                new DynamicEntryPlugin(context, entry).apply(compiler);
            }
            return true;
        });
    }
};
```

除了 `EntryOptionPlugin`，其他的内置插件也会监听特定的任务点来完成特定的逻辑，但我们这里不再仔细讨论。当 `Compiler` 实例加载完内置插件之后，下一步就会直接调用 `compiler.run` 方法来启动构建，这时候 Compiler 的 `run` 钩子被触发，在 `run` 钩子回调中可以得到解析后的 `compiler.options`。

> Tips：`run` 钩子只有在 Webpack 以正常模式运行的情况下会触发，如果我们以监听 （watch）模式运行 Webpack，那么 `run` 是不会触发的，但是会触发 `watchRun` 钩子。

完成了 Webpack 配置的处理，接下来开始构建流程，构建流程主要是在`Compiler`的`Compiler.compile`内：

```
// webpack v4.29.6
// lib/Compiler.js#L610-L636
compile(callback) {
    // Compilation类的参数
    const params = this.newCompilationParams();
    // 1. 执行beforeCompile 钩子回调
    this.hooks.beforeCompile.callAsync(params, err => {
        if (err) return callback(err);
        // 2. 执行 Compiler.compile 钩子回调
        this.hooks.compile.call(params);
        // 3. 实例化 Compilation
        const compilation = this.newCompilation(params);
        // 4. 执行 Compiler.make 钩子回调
        // make内实际主要是执行的compilation的addEntry方法(**注意这里**)
        this.hooks.make.callAsync(compilation, err => {
            if (err) return callback(err);

            compilation.finish();
            // seal方法整理构建之后的chunk产出
            // 这里会做一些优化相关的事情，比如压缩代码等
            compilation.seal(err => {
                if (err) return callback(err);
                // 执行 Compiler.afterCompile 钩子回调
                this.hooks.afterCompile.callAsync(compilation, err => {
                    if (err) return callback(err);

                    return callback(null, compilation);
                });
            });
        });
    });
}
```

`newCompilationParams`主要是生成对应 compilation 用到的参数：

```
// webpack 4.29.6
// lib/Compiler.js#L601
newCompilationParams() {
    const params = {
        // normal module
        normalModuleFactory: this.createNormalModuleFactory(),
        // context module
        contextModuleFactory: this.createContextModuleFactory(),
        // 依赖关系表
        compilationDependencies: new Set()
    };
    return params;
}
```

紧接着 `Compiler` 实例开始创建 `Compilation` 对象，这个对象是后续构建流程中**最核心最重要的对象**，它包含了一次构建过程中所有的数据，一次构建过程对应一个 `Compilation` 实例。当 `Compilation` 实例创建完成之后，Webpack 的准备阶段已经完成，下一步将开始编译阶段。

编译阶段

从 Compiler 的`make`钩子触发开始，此时内置插件`SingleEntryPlugin`、`MultiEntryPlugin`、`DynamicEntryPlugin`（根据不同类型 entry）的监听器会开始执行。监听器都会调用 Compilation 实例的 `compilation.addEntry()` 方法，该方法将会触发第一批 module 的解析，这些 module 就是 entry 中配置的模块。

```
// webpack 4.29.6
// lib/Compilation.js#L1019
addEntry(context, entry, name, callback) {
    // ...
    // 执行内部_addModuleChain方法
    this._addModuleChain(
        context,
        entry,
        module => {
            this.entries.push(module);
        },
        (err, module) => {
            // ...
        }
    );
}
```

我们先讲一个 module 解析完成之后的操作，它会递归调用它所依赖的 modules 进行解析，所以当解析停止时，我们就能够得到项目中所有依赖的 modules，它们将存储在 Compilation 实例的 modules 属性中，并触发`compilation`的`finishModules`钩子。

`module`对象有`NormalModule`、`MultiModule`、`ContextModule`、`DLLModule`等多种类型（分别在对应的`lib/*Module.js`中实现）。下面以`NormalModule`为例讲解下 module 的解析流程，其他类型的解析都是类似。

NormalModule 的实例化需要借用对应的`NormalModuleFactory.create()`，`NormalModuleFactory`则来自于上一阶段创建`Compilation`对象传入的参数。创建`NormalModule`之前会调用`resolver`来获取一个 module 的属性，比如解析这个 module 需要用到的`loaders`，资源路径`resource`等等。

`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。

在创建完 NormalModule 实例之后会调用`NormalModule.build()`方法继续进行内部的构建，`NormalModule.build()`会调用`NormalModule.doBuild()`，在`doBuild()`中执行 loader，生成 AST 语法树。

```
// webpack 4.29.6
// lib/NormalModule.js#L274
doBuild(options, compilation, resolver, fs, callback) {
    // 生成loader上下文
    const loaderContext = this.createLoaderContext(
        resolver,
        options,
        compilation,
        fs
    );
    // 开始执行loader
    runLoaders(
        {
            resource: this.resource,
            loaders: this.loaders,
            context: loaderContext,
            readResource: fs.readFile.bind(fs)
        },
        (err, result) => {
            // ...

            if (err) {
                // ...
                return callback(error);
            }

            const resourceBuffer = result.resourceBuffer;
            const source = result.result[0];
            const sourceMap = result.result.length >= 1 ? result.result[1] : null;
            const extraInfo = result.result.length >= 2 ? result.result[2] : null;

            // ...
            // 这里是处理后的源码
            this._source = this.createSource(
                this.binary ? asBuffer(source) : asString(source),
                resourceBuffer,
                sourceMap
            );
            // 这里是ast
            this._ast =
                typeof extraInfo === "object" &&
                extraInfo !== null &&
                extraInfo.webpackAST !== undefined
                    ? extraInfo.webpackAST
                    : null;
            return callback();
        }
    );
}
```

当一个模块编译完成之后，有会根据其`AST`查找依赖，递归整个构建流程，直到整个所有依赖都被处理完毕。得到所有的 modules 之后，Webpack 会开始生成对应的 chunk。这些 chunk 对象是 Webpack 生成最终文件的一个重要依据。每个 chunk 的生成就是找到需要包含的 modules。这里大致描述一下 chunk 的生成算法：

1. Webpack 先将 `entry` 中对应的 module 都生成一个新的 chunk；
2. 遍历 module 的依赖列表，将依赖的 module 也加入到 chunk 中；
3. 如果一个依赖 module 是动态引入（`import()`、`require.ensure()`）的模块，那么就会根据这个 module 创建一个新的 chunk，继续遍历依赖；
4. 重复上面的过程，直至得到所有的 chunks。

得到所有 chunks 之后，Webpack 会对 chunks 和 modules 进行一些优化相关的操作，比如分配 id、排序等，即进入到`compilation.seal()`内，这时候会触发`webpack.optimize`配置中用到的一些插件。

至此，编译阶段处理完成，进入产出阶段。

产出阶段

在产出阶段，webpack 会根据 chunks 生成最终文件。主要有三个步骤：模板 hash 更新，模板渲染 chunk，生成 bunlde 文件。

Compilation 在实例化的时候，就会同时实例化三个对象：`MainTemplate`，`ChunkTemplate`，`ModuleTemplate`，这三个对象是用来渲染 chunk 对象，得到最终代码的模板。

- `MainTemplate`：对应了在 entry 配置的入口 chunk 的渲染模板；
- `ChunkTemplate`：动态引入的非入口 chunk 的渲染模板；
- `ModuleTemplate`：chunk 中的 module 的渲染模板。

在开始渲染之前， Compilation 实例会调用 `compilation.createHash` 方法来生成这次构建的 Hash，在 Webpack 的配置中，我们可以在 `output.filename` 中配置 `[hash]` 占位符，最终就会替换成这个 Hash。同样， `compilation.createHash` 也会为每一个 chunk 也创建一个 Hash，对应 `[chunkhash]` 占位符。

当 Hash 都创建完成之后，下一步就会遍历 `compilation` 对象的 `chunks` 属性，来渲染每一个 chunk。如果一个 chunk 是入口（entry） chunk，那么就会调用 `MainTemplate` 实例的 `render` 方法，否则调用 `ChunkTemplate` 的 render 方法：

```
// webpack 4.29.6
// lib/Compilation.js#L2314
for (let i = 0; i < chunks.length; i++) {
    const chunk = chunks[i];
    const chunkHash = createHash(hashFunction);
    try {
        if (outputOptions.hashSalt) {
            chunkHash.update(outputOptions.hashSalt);
        }
        chunk.updateHash(chunkHash);
        // 根据类型选择模板
        const template = chunk.hasRuntime() ? this.mainTemplate : this.chunkTemplate;
        template.updateHashForChunk(chunkHash, chunk, this.moduleTemplates.javascript, this.dependencyTemplates);
        this.hooks.chunkHash.call(chunk, chunkHash);
        chunk.hash = chunkHash.digest(hashDigest);
        hash.update(chunk.hash);
        chunk.renderedHash = chunk.hash.substr(0, hashDigestLength);
        this.hooks.contentHash.call(chunk);
    } catch (err) {
        this.errors.push(new ChunkRenderError(chunk, '', err));
    }
}
```

当每个 chunk 的源码生成之后，就会添加在 Compilation 实例对象的 `assets` 属性中。`assets` 中的对象 `key` 是最终要生成的文件名称， `value` 是一个对象，对象需要包含两个方法， `source` 和 `size` 分别返回文件内容和文件大小。当所有的 chunk 都渲染完成之后， assets 就是最终更要生成的文件列表。

完成上面的操作之后， Compilation 实例的 `seal` 方法结束，进入到 Compiler 实例的 `emitAssets` 方法。Compilation 实例的所有工作到此也全部结束，意味着一次构建过程已经结束，接下来 Webpack 会直接遍历 compilation.assets 生成所有文件，然后触发任务点 done，结束构建流程。

验证全流程

通过 Tapable 文章的内容得知，我们可以给 Tapable 的实例使用 tap 的方式添加回调函数，再结合 Webpack 的 API 章节内容得知`webpack(config)`返回的实际是`compiler`，所以我们可以遍历`comipler.hooks`，使用`hook.tap`的方法添加回调函数，将 hookName 打印出来，通过这样的方式，可以把 webpack compiler 部分的流程全部输出出来。代码如下：

```
const webpack = require('webpack');
const config = {
    mode: 'development',
    devtool: false,
    // 下面是只有一个entry的情况
    // 没有output则默认输出是到dist的main
    entry: './src/app.js'
};

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
```

上面的方式是得到了`compiler.run()`之后的流程，这个流程缺少了环境变量和参数处理的流程，因为这些事情已经在`run()`操作之前的`webpack()`调用期间实例化`Compiler`就做完了。再进一步，我们直接修改`node_modules/webpack/lib/Compiler.js`的代码，在`Compiler.constructor`最后添加代码：

```
Object.keys(this.hooks).forEach(hookName => {
    const hook = this.hooks[hookName];
    if (hook.tap) {
        hook.tap('anyString', () => {
            console.log(`compiler -> ${hookName}`);
        });
    }
});
```

这样得到更加完整的执行过程：

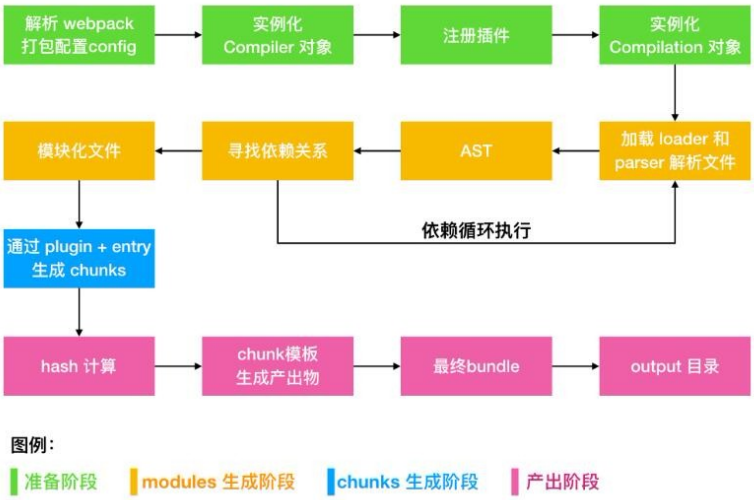
```
compiler -> environment
compiler -> afterEnvironment
compiler -> entryOption
compiler -> afterPlugins
compiler -> afterResolvers
compiler -> beforeRun
compiler -> run
compiler -> normalModuleFactory
compiler -> contextModuleFactory
compiler -> beforeCompile
compiler -> compile
compiler -> thisCompilation
compiler -> compilation
compiler -> make
compiler -> afterCompile
compiler -> shouldEmit
compiler -> emit
compiler -> afterEmit
compiler -> done
```

综上，在`compiler.run()`之前有一下流程：

```
environment
afterEnvironment
entryOption
afterPlugins
afterResolvers
```

全流程图

通过上面的解释、源码分析和嵌入式代码验证，我们已经了解了 Webpack 打包的全流程，下面是结合上面的内容，整理的一张 Webpack 工作流程图，供大家进一步学习和巩固本文的内容。



> Tips: 我们还可以用同样的方法修改`node_modules/webpack/lib/Compilation.js`，也添加上 hooks 的遍历，这样可以通过输出的 hookName 把全流程就串起来了！最终得到全流程如下：

```
compiler -> environment
compiler -> afterEnvironment
compiler -> entryOption
compiler -> afterPlugins
compiler -> afterResolvers
compiler -> beforeRun
compiler -> run
```

```
compiler -> normalModuleFactory
compiler -> contextModuleFactory
compiler -> beforeCompile
compiler -> compile
compiler -> thisCompilation
compiler -> compilation
compiler -> make
    compilation -> addEntry
    compilation -> buildModule
    compilation -> normalModuleLoader
    compilation -> succeedModule
    compilation -> buildModule
    compilation -> normalModuleLoader
    compilation -> succeedModule
    compilation -> succeedEntry
    compilation -> finishModules
    compilation -> seal
    compilation -> optimizeDependenciesBasic
    compilation -> optimizeDependencies
    compilation -> optimizeDependenciesAdvanced
    compilation -> afterOptimizeDependencies
    compilation -> beforeChunks
    compilation -> dependencyReference
    compilation -> afterChunks
    compilation -> optimize
    compilation -> optimizeModulesBasic
    compilation -> optimizeModules
    compilation -> optimizeModulesAdvanced
    compilation -> afterOptimizeModules
    compilation -> optimizeChunksBasic
    compilation -> optimizeChunks
    compilation -> optimizeChunksAdvanced
    compilation -> afterOptimizeChunks
    compilation -> optimizeTree
    compilation -> afterOptimizeTree
    compilation -> optimizeChunkModulesBasic
    compilation -> optimizeChunkModules
    compilation -> optimizeChunkModulesAdvanced
    compilation -> afterOptimizeChunkModules
    compilation -> shouldRecord
    compilation -> reviveModules
    compilation -> optimizeModuleOrder
    compilation -> advancedOptimizeModuleOrder
    compilation -> beforeModuleIds
    compilation -> moduleIds
    compilation -> optimizeModuleIds
    compilation -> afterOptimizeModuleIds
    compilation -> reviveChunks
    compilation -> optimizeChunkOrder
    compilation -> beforeChunkIds
    compilation -> optimizeChunkIds
    compilation -> afterOptimizeChunkIds
    compilation -> recordModules
    compilation -> recordChunks
    compilation -> beforeHash
    compilation -> chunkHash
    compilation -> contentHash
    compilation -> afterHash
    compilation -> recordHash
    compilation -> beforeModuleAssets
    compilation -> shouldGenerateChunkAssets
    compilation -> beforeChunkAssets
    compilation -> chunkAsset
    compilation -> additionalChunkAssets
    compilation -> record
    compilation -> additionalAssets
    compilation -> optimizeChunkAssets
    compilation -> afterOptimizeChunkAssets
    compilation -> optimizeAssets
    compilation -> afterOptimizeAssets
    compilation -> needAdditionalSeal
    compilation -> afterSeal
compiler -> afterCompile
```

```
compiler -> shouldEmit
compiler -> emit
compiler -> afterEmit
    compilation -> needAdditionalPass
compiler -> done
```

为了更好理解 compiler 和 Compilation 的区别，再将`webpack.config.js`修改下，增加`watch`配置，这样只要 entry 文件修改过，就会重新启动一次编译。我们来看下 watch 编译的流程和第一次编译流程有什么区别：

```
// webpack.config.js
module.exports = {
    mode: 'development',
    devtool: false,
    // 下面是只有一个entry的情况
    // 没有output则默认输出是到dist的main
    entry: './src/app.js',
    // 增加watch功能
    watch: true
};
```

这时候执行`webpack --config webpack.config.js`，第一次编译触发，然后修改下`app.js`，通过 watch 监控，发生了第二次编译，得到下面的输出：

```
compiler -> invalid
compiler -> watchRun
compiler -> normalModuleFactory
compiler -> contextModuleFactory
compiler -> beforeCompile
compiler -> compile
compiler -> thisCompilation
compiler -> compilation
compiler -> make
    compilation -> addEntry
    compilation -> buildModule
    compilation -> normalModuleLoader
    compilation -> succeedModule
    compilation -> succeedEntry
    compilation -> finishModules
    compilation -> seal
    compilation -> optimizeDependenciesBasic
    compilation -> optimizeDependencies
    compilation -> optimizeDependenciesAdvanced
    compilation -> afterOptimizeDependencies
    compilation -> beforeChunks
    compilation -> dependencyReference
    compilation -> afterChunks
    compilation -> optimize
    compilation -> optimizeModulesBasic
    compilation -> optimizeModules
    compilation -> optimizeModulesAdvanced
    compilation -> afterOptimizeModules
    compilation -> optimizeChunksBasic
    compilation -> optimizeChunks
    compilation -> optimizeChunksAdvanced
    compilation -> afterOptimizeChunks
    compilation -> optimizeTree
    compilation -> afterOptimizeTree
    compilation -> optimizeChunkModulesBasic
    compilation -> optimizeChunkModules
    compilation -> optimizeChunkModulesAdvanced
    compilation -> afterOptimizeChunkModules
    compilation -> shouldRecord
    compilation -> reviveModules
    compilation -> optimizeModuleOrder
    compilation -> advancedOptimizeModuleOrder
    compilation -> beforeModuleIds
    compilation -> moduleIds
    compilation -> optimizeModuleIds
    compilation -> afterOptimizeModuleIds
```

```
    compilation -> reviveChunks
    compilation -> optimizeChunkOrder
    compilation -> beforeChunkIds
    compilation -> optimizeChunkIds
    compilation -> afterOptimizeChunkIds
    compilation -> recordModules
    compilation -> recordChunks
    compilation -> beforeHash
    compilation -> chunkHash
    compilation -> contentHash
    compilation -> afterHash
    compilation -> recordHash
    compilation -> beforeModuleAssets
    compilation -> shouldGenerateChunkAssets
    compilation -> beforeChunkAssets
    compilation -> chunkAsset
    compilation -> additionalChunkAssets
    compilation -> record
    compilation -> additionalAssets
    compilation -> optimizeChunkAssets
    compilation -> afterOptimizeChunkAssets
    compilation -> optimizeAssets
    compilation -> afterOptimizeAssets
    compilation -> needAdditionalSeal
    compilation -> afterSeal
compiler -> afterCompile
compiler -> shouldEmit
compiler -> emit
compiler -> afterEmit
    compilation -> needAdditionalPass
compiler -> done
```

通过上面输出的内容可以发现，compiler 只是从`invalid -> watchRun` 开始，没有重新走流程，而`compilation`却是走了一个完整的流程，所以我们更好地理解了：**compiler 是管理整个生命周期的，而 compilation 是每次编译触发都会重新生成一次的**。

> Tips: 当处于监听模式时，compiler 会触发诸如 watchRun, watchClose 和 invalid 等额外的事件，这个跟普通模式下的流程稍有不同，主要区别在下面的内容：

```
# 普通模式
compiler -> afterResolvers
compiler -> beforeRun （不同）
compiler -> run （不同）
compiler -> normalModuleFactory

# watch模式
compiler -> afterResolvers
compiler -> watchRun （不同）
compiler -> normalModuleFactory
```

总结

本小节主要是结合之前的 Webpack 内核知识，来讲解一次打包的过程 Webpack 做的事情有哪些。Webpack 打包流程从配置文件的读取开始，分别经过了准备阶段、modules 产出阶段、chunks 产出阶段和 bundle 产出物产出阶段。在各自阶段，分别有不同的「角色」参与，整个 Webpack 的打包流程是通过 Compiler 来控制的，而每次打包的过程是通过 Compilation 来控制的。在普通打包模式下，webpack 的 Compiler 和 Compilation 是一一对应的关系； watch 模式下，Webpack 的 Compiler 会因为文件变化而产生多次打包流程，所以 Compiler 和 Compilation 是一对多关系，通过 Hook Compiler 的流程，可以得到每次打包过程的回调。

本小节知识量较大，所以继续总结下 webpack 工作流程中涉及到的类（对象）的作用，我们也可以从这些对象的角度来梳理和记忆 Webpack 的工作流程：

- Tapbale：Webpack 事件流程核心类；
- Compiler：Webpack 工作流程中最高层的对象，初始化配置，提供 Webpack 流程的全局钩子，比如`done`、`compilation`这类；
- Compilation：由 Compiler 来创建的实例对象，是每次打包流程最核心的流程，该对象内进行模块依赖解析、优化资源、渲染 runtime 代码等事情，下面在 Compilation 中还有用到的一些对象：
  - Resolver：解析模块（module）、loader 等路径，帮助查找对应的位置；
  - ModuleFactory：负责构造模块的实例，将 Resolver 解析成功的组件中把源码从文件中读取出来，然后创建模块对象；
  - Template：主要是来生成 runtime 代码，将解析后的代码按照依赖顺序处理之后，套上 Template 就是我们最终打包出来的代码。

> 本小节 Webpack 相关面试题：
>
> 1. 能否说下 Webpack 的完整打包流程，从读取配置到输出文件这个过程尽量说全？
> 2. 介绍几个你了解过的 Webpack 中的类，有什么用？

Webpack 的 Compiler 和 Compilation

从 Webpack 的产出代码来看 Webpack 是怎么执行的