webpack 构建流程

webpack 基本架构

webpack 的基本架构，是基于一种类似事件的方式。下面的代码中，对象可以使用 plugin 函数来注册一个事件，暂时可以理解为我们熟悉的 addEventListener。但为了区分概念，后续的讨论中会将事件名称为 **任务点**，比如下面有四个任务点 compilation, optimize, compile, before-resolve：

```
compiler.plugin("compilation", (compilation, callback) => {
    // 当 Compilation 实例生成时

    compilation.plugin("optimize", () => {
        // 当所有 modules 和 chunks 已生成，开始优化时
    })
})

compiler.plugin("compile", (params) => {
    // 当编译器开始编译模块时

    let nmf = params.normalModuleFactory
    nmf.plugin("before-resolve", (data) => {
        // 在 factory 开始解析模块前
    })
})
```

webpack 内部的大部分功能，都是通过这种注册任务点的形式来实现的，这在后面中我们很容易发现这一点。所以这里直接抛出结论：**webpack 的核心功能，是抽离成很多个内部插件来实现的。**

那这些内部插件是如何对 webpack 产生作用的呢？在我们开始运行 webpack 的时候，它会先创建一个 Compiler 实例，然后调用 WebpackOptionsApply 这个模块给 Compiler 实例添加内部插件：

```
compiler = new Compiler();
// 其他代码..
compiler.options = new WebpackOptionsApply().process(options, compiler);
```

在 WebpackOptionsApply 这个插件内部会根据我们传入的 webpack 配置来初始化需要的内部插件：

```
JsonpTemplatePlugin = require("./JsonpTemplatePlugin");
NodeSourcePlugin = require("./node/NodeSourcePlugin");
compiler.apply(
    new JsonpTemplatePlugin(options.output),
    new FunctionModulePlugin(options.output),
    new NodeSourcePlugin(options.node),
    new LoaderTargetPlugin(options.target)
);

// 其他代码..

compiler.apply(new EntryOptionPlugin());
compiler.applyPluginsBailResult("entry-option", options.context,
options.entry);

compiler.apply(
    new CompatibilityPlugin(),
    new HarmonyModulesPlugin(options.module),
    new AMDPlugin(options.module, options.amd || {}),
    new CommonJsPlugin(options.module),
    new LoaderPlugin(),
    new NodeStuffPlugin(options.node),
    new RequireJsStuffPlugin(),
    new APIPlugin(),
    new ConstPlugin(),
    new UseStrictPlugin(),
    new RequireIncludePlugin(),
    new RequireEnsurePlugin(),
    new RequireContextPlugin(options.resolve.modules,
options.resolve.extensions, options.resolve.mainFiles),
    new ImportPlugin(options.module),
    new SystemPlugin(options.module)
);
```

每一个内部插件，都是通过监听任务点的方式，来实现自定义的逻辑。比如 JsonpTemplatePlugin 这个插件，是通过监听 mainTemplate 对象的 require-ensure 任务点，来生成 jsonp 风格的代码：

```
mainTemplate.plugin("require-ensure", function(_, chunk, hash) {
    return this.asString([
        "var installedChunkData = installedChunks[chunkId];",
```

```
        "if(installedChunkData === 0) {",
        this.indent([
            "return new Promise(function(resolve) { resolve(); });"
        ]),
        "}",
        ""
        ,
        "// a Promise means \"currently loading\".",
        "if(installedChunkData) {",
        this.indent([
            "return installedChunkData[2];"
        ]),
        "}",
        ""
        ,
        "// setup Promise in chunk cache",
        "var promise = new Promise(function(resolve, reject) {",
        this.indent([
            "installedChunkData = installedChunks[chunkId] = [resolve,
reject];"
        ]),
        "});",
        "installedChunkData[2] = promise;",
        ""
        ,
        "// start chunk loading",
        "var head = document.getElementsByTagName('head')[0];",
        this.applyPluginsWaterfall("jsonp-script", "", chunk, hash),
        "head.appendChild(script);",
        ""
        ,
        "return promise;"
    ]);
});
```

现在我们理解了 webpack 的基本架构之后，可能会产生疑问，**每个插件应该监听哪个对象的哪个任务点，又如何对实现特定功能呢？**

要完全解答这个问题很难，原因在于 **webpack 中构建过程中，会涉及到非常多的对象和任务点**，要对每个对象和任务点都进行讨论是很困难的。但是，我们仍然可以挑选完整构建流程中涉及到的几个核心对象和任务点，把 webpack 的构建流程讲清楚，当我们需要实现某个特定内容的时候，再去找对应的模块源码查阅任务点。

webpack 的构建流程

三个阶段

1. webpack 的准备阶段
2. modules 和 chunks 的生成阶段
3. 文件生成阶段

webpack 的准备阶段

这个阶段的主要工作，是创建 Compiler 和 Compilation 实例。

首先我们从 webpack 的运行开始讲起，在前面我们大概地讲过，当我们开始运行 webpack 的时候，就会创建 Compiler 实例并且加载内部插件。这里跟构建流程相关性比较大的内部插件是 EntryOptionPlugin，我们来看看它到底做了什么：

```
// https://github.com/webpack/webpack/blob/master/lib/WebpackOptionsApply.js
compiler.apply(new EntryOptionPlugin());
compiler.applyPluginsBailResult("entry-option", options.context,
options.entry); // 马上触发任务点运行 EntryOptionPlugin 内部逻辑

// https://github.com/webpack/webpack/blob/master/lib/EntryOptionPlugin.js
module.exports = class EntryOptionPlugin {
    apply(compiler) {
        compiler.plugin("entry-option", (context, entry) => {
            if(typeof entry === "string" || Array.isArray(entry)) {
                compiler.apply(itemToPlugin(context, entry, "main"));
            } else if(typeof entry === "object") {
                Object.keys(entry).forEach(name =>
compiler.apply(itemToPlugin(context, entry[name], name)));
            } else if(typeof entry === "function") {
                compiler.apply(new DynamicEntryPlugin(context, entry));
            }
            return true;
        });
    }
};
```

EntryOptionPlugin 的代码只有寥寥数行但是非常重要，它会解析传给 webpack 的配置中的 entry 属性，然后生成不同的插件应用到 Compiler 实例上。这些插件可能是 SingleEntryPlugin, MultiEntryPlugin 或者 DynamicEntryPlugin。但不管是哪个插件，内部都会监听 Compiler 实例对象的 make 任务点，以 SingleEntryPlugin 为例：

```
// https://github.com/webpack/webpack/blob/master/lib/SingleEntryPlugin.js
```

```
class SingleEntryPlugin {
    // 其他代码..
    apply(compiler) {
        // 其他代码..
        compiler.plugin("make", (compilation, callback) => {
            const dep = SingleEntryPlugin.createDependency(this.entry,
this.name);
            compilation.addEntry(this.context, dep, this.name, callback);
        });
    }
}
```

这里的 make 任务点将成为后面解析 modules 和 chunks 的起点。

除了 EntryOptionPlugin，其他的内部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 实例加载完内部插件之后，下一步就会直接调用 compiler.run 方法来启动构建，任务点 run 也是在此时触发，值得注意的是此时基本只有 options 属性是解析完成的：

```
compiler.plugin("run", (compiler, callback) => {
    console.log(compiler.options) // 可以看到解析后的配置
    callback()
})
```

另外要注意的一点是，任务点 run 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)的模式运行 webpack，那么任务点 run 是不会触发的，但是会触发任务点 watch-run。

接下来， Compiler 对象会开始实例化两个核心的工厂对象，分别是 NormalModuleFactory 和 ContextModuleFactory。工厂对象顾名思义就是用来创建实例的，它们后续用来创建 NormalModule 以及 ContextModule 实例，这两个工厂对象会在任务点 compile 触发时传递过去，所以任务点 compile 是间接监听这两个对象的任务点的一个入口：

```
// 监听任务点 compile
compiler.plugin("compile", (params) => {
    let nmf = params.normalModuleFactory
    nmf.plugin("before-resolve", (data, callback) => {
        // ...
    })
})
```

下一步 Compiler 实例将会开始创建 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。也就是说一次构建过程对应一个 Compilation 实例。在创建 Compilation 实例时会触发任务点 compilaiion 和 this-compilation：

```
class Compiler extends Tapable {

    // 其他代码..

    newCompilation(params) {
        const compilation = this.createCompilation();
        compilation.fileTimestamps = this.fileTimestamps;
        compilation.contextTimestamps = this.contextTimestamps;
        compilation.name = this.name;
        compilation.records = this.records;
        compilation.compilationDependencies = params.compilationDependencies;
        this.applyPlugins("this-compilation", compilation, params);
        this.applyPlugins("compilation", compilation, params);
        return compilation;
    }
}
```

这里为什么会有 compilation 和 this-compilation 两个任务点？其实是跟子编译器有关，Compil。er 实例通过 createChildCompiler 方法可以创建子编译器实例 childCompiler，创建时 childCompiler 会复制 compiler 实例的任务点监听器。**任务点 compilation 的监听器会被复制，而任务点 this-compilation 的监听器不会被复制。**

compilation 和 this-compilation 是最快能够获取到 Compilation 实例的任务点，如果你的插件功能需要尽早对 Compilation 实例进行一些操作，那么这两个任务点是首选：

```
compiler.plugin("this-compilation", (compilation, params) => {
    console.log(compilation.options === compiler.options) // true
    console.log(compilation.compiler === compiler) // true
    console.log(compilation)
})
```

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 modules 和 chunks 的生成阶段。

modules 和 chunks 的生成阶段

**这个阶段的主要内容，是先解析项目依赖的所有 modules，再根据 modules 生成 chunks。

module 解析，包含了三个主要步骤：创建实例、loaders 应用以及依赖收集。

chunks 生成，主要步骤是找到 chunk 所需要包含的 modules。**

当上一个阶段完成之后，下一个任务点 make 将被触发，此时内部插件 SingleEntryPlugin, MultiEntryPlugin, DynamicEntryPlugin 的监听器会开始执行。监听器都会调用 Compilation 实例的 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。

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，并触发任务点 finish-modules：

```
// 监听 finish-modules 任务点
compiler.plugin("this-compilation", (compilation) => {
    compilation.plugin("finish-modules", (modules) => {
        console.log(modules === compilation.modules) // true
        modules.forEach(module => {
            console.log(module._source.source()) // 处理后的源码
        })
    })
})
```


下面将以 NormalModule 为例讲解一下 module 的解析过程，ContextModule 等其他模块实例的处理是类似的。

第一个步骤是创建 NormalModule 实例。这里需要用到上一个阶段讲到的 NormalModuleFactory 实例， NormalModuleFactory 的 create 方法是创建 NormalModule 实例的入口，内部的主要过程是解析 module 需要用到的一些属性，比如需要用到的 loaders, 资源路径 resource 等等，最终将解析完毕的参数传给 NormalModule 构建函数直接实例化：

```
// 以 require("raw-loader!./a") 为例
// 并且对 .js 后缀配置了 babel-loader
createdModule = new NormalModule(
    result.request,      // <raw-loader>!<babel-loader>!/path/to/a.js
    result.userRequest,  // <raw-loader>!/path/to/a.js
```

```
    result.rawRequest,    // raw-loader!./a.js
    result.loaders,       // [<raw-loader>, <babel-loader>]
    result.resource,      // /path/to/a.js
    result.parser
);
```

这里在解析参数的过程中，有两个比较实用的任务点 before-resolve 和 after-resolve，分别对应了解析参数前和解析参数后的时间点。举个例子，在任务点 before-resolve 可以做到忽略某个 module 的解析，webpack 内部插件 IgnorePlugin 就是这么做的：

```
class IgnorePlugin {
    checkIgnore(result, callback) {
        // check if result is ignored
        if(this.checkResult(result)) {
            return callback(); // callback 第二个参数为 undefined 时会终止
module 解析
        }
        return callback(null, result);
    }

    apply(compiler) {
        compiler.plugin("normal-module-factory", (nmf) => {
            nmf.plugin("before-resolve", this.checkIgnore);
        });
        compiler.plugin("context-module-factory", (cmf) => {
            cmf.plugin("before-resolve", this.checkIgnore);
        });
    }
}
```

在创建完 NormalModule 实例之后会调用 build 方法继续进行内部的构建。我们熟悉的 **loaders 将会在这里开始应用**，NormalModule 实例中的 loaders 属性已经记录了该模块需要应用的 loaders。应用 loaders 的过程相对简单，直接调用 loader-runner 这个模块即可：

```
const runLoaders = require("loader-runner").runLoaders;
```

```
// 其他代码..
class NormalModule extends Module {
    // 其他代码..
    doBuild(options, compilation, resolver, fs, callback) {
        this.cacheable = false;
        const loaderContext = this.createLoaderContext(resolver, options,
compilation, fs);

        runLoaders({
            resource: this.resource,
            loaders: this.loaders,
            context: loaderContext,
            readResource: fs.readFile.bind(fs)
        }, (err, result) => {
            // 其他代码..
        });
    }
}
```

webpack 中要求 NormalModule 最终都是 js 模块，所以 loader 的作用之一是将不同的资源文件转化成 js 模块。比如 html-loader 是将 html 转化成一个 js 模块。在应用完 loaders 之后，NormalModule 实例的源码必然就是 js 代码，这对下一个步骤很重要。

下一步我们需要得到这个 module 所依赖的其他模块，所以就有一个依赖收集的过程。webpack 的依赖收集过程是将 js 源码传给 js parser（webpack 使用的 parser 是 acorn）：

```
class NormalModule extends Module {
    // 其他代码..
    build(options, compilation, resolver, fs, callback) {
        // 其他代码..
        return this.doBuild(options, compilation, resolver, fs, (err) => {
            // 其他代码..
            try {
                this.parser.parse(this._source.source(), {
                    current: this,
```

```
                    module: this,
                    compilation: compilation,
                    options: options
                });
            } catch(e) {
                const source = this._source.source();
                const error = new ModuleParseError(this, source, e);
                this.markModuleAsErrored(error);
                return callback();
            }
            return callback();
        });
    }
}
```

parser 将 js 源码解析后得到对应的 AST(抽象语法树, Abstract Syntax Tree)。然后 **webpack 会遍历 AST，按照一定规则触发任务点。** 比如 js 源码中有一个表达式：a.b.c，那么 parser 对象就会触发任务点 expression a.b.c。更多相关的规则 webpack 在官网有罗列出来，大家可以对照着使用。

有了 AST 对应的任务点，依赖收集就相对简单了，比如遇到任务点 call require，说明在代码中是有调用了 require 函数，那么就应该给 module 添加新的依赖。webpack 关于这部分的处理是比较复杂的，因为 webpack 要兼容多种不同的依赖方式，比如 AMD 规范、CommonJS 规范，然后还要区分动态引用的情况，比如使用了 require.ensure, require.context。但这些细节对于我们讨论构建流程并不是必须的，因为不展开细节讨论。

当 parser 解析完成之后，module 的解析过程就完成了。每个 module 解析完成之后，都会触发 Compilation 实例对象的任务点 succeed-module，我们可以在这个任务点获取到刚解析完的 module 对象。正如前面所说，module 接下来还要继续递归解析它的依赖模块，最终我们会得到项目所依赖的所有 modules。**此时任务点 make 结束。**

继续往下走，Compialtion 实例的 seal 方法会被调用并马上触发任务点 seal。在这个任务点，我们可以拿到所有解析完成的 module：

```
// 监听 seal 任务点
compiler.plugin("this-compilation", (compilation) => {
    console.log(compilation.modules.length === 0) // true
    compilation.plugin("seal", () => {
        console.log(compilation.modules.length > 0) // true
    })
})
```

有了所有的 modules 之后，webpack 会开始生成 chunks。webpack 中的 chunk 概念，要不就是配置在 entry 中的模块，要不就是动态引入（比如 require.ensure）的模块。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

在所有 chunks 生成之后，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，并且触发一系列相关的任务点：


```
class Compilation extends Tapable {
    // 其他代码 ..
    seal(callback) {
        // 生成 chunks 代码..
        self.applyPlugins0("optimize");

        while(self.applyPluginsBailResult1("optimize-modules-basic",
self.modules) ||
            self.applyPluginsBailResult1("optimize-modules", self.modules) ||
            self.applyPluginsBailResult1("optimize-modules-advanced",
self.modules)) { /* empty */ }
        self.applyPlugins1("after-optimize-modules", self.modules);

        while(self.applyPluginsBailResult1("optimize-chunks-basic",
self.chunks) ||
            self.applyPluginsBailResult1("optimize-chunks", self.chunks) ||
            self.applyPluginsBailResult1("optimize-chunks-advanced",
self.chunks)) { /* empty */ }
```

```
        self.applyPlugins1("after-optimize-chunks", self.chunks);

        self.applyPluginsAsyncSeries("optimize-tree", self.chunks,
self.modules, function sealPart2(err) {
            if(err) {
                return callback(err);
            }

            self.applyPlugins2("after-optimize-tree", self.chunks,
self.modules);

            while(self.applyPluginsBailResult("optimize-chunk-modules-basic",
self.chunks, self.modules) ||
                self.applyPluginsBailResult("optimize-chunk-modules",
self.chunks, self.modules) ||
                self.applyPluginsBailResult("optimize-chunk-modules-advanced",
self.chunks, self.modules)) { /* empty */ }
            self.applyPlugins2("after-optimize-chunk-modules", self.chunks,
self.modules);

            const shouldRecord = self.applyPluginsBailResult("should-
record") !== false;

            self.applyPlugins2("revive-modules", self.modules, self.records);
            self.applyPlugins1("optimize-module-order", self.modules);
            self.applyPlugins1("advanced-optimize-module-order",
self.modules);
            self.applyPlugins1("before-module-ids", self.modules);
            self.applyPlugins1("module-ids", self.modules);
            self.applyModuleIds();
            self.applyPlugins1("optimize-module-ids", self.modules);
            self.applyPlugins1("after-optimize-module-ids", self.modules);

            self.sortItemsWithModuleIds();

            self.applyPlugins2("revive-chunks", self.chunks, self.records);
            self.applyPlugins1("optimize-chunk-order", self.chunks);
            self.applyPlugins1("before-chunk-ids", self.chunks);
            self.applyChunkIds();
            self.applyPlugins1("optimize-chunk-ids", self.chunks);
            self.applyPlugins1("after-optimize-chunk-ids", self.chunks);
```

```
            // 其他代码..
        })
    }
}
```

这些任务点一般是 webpack.optimize 属性下的插件会使用到，比如 CommonsChunkPlugin 会使用到任务点 optimize-chunks，但这里我们不深入讨论。

至此，modules 和 chunks 的生成阶段结束。接下来是文件生成阶段。

文件生成阶段

这个阶段的主要内容，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

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。第一个对应了在 entry 配置的入口 chunk 的渲染模板，第二个是动态引入的非入口 chunk 的渲染模板，最后是 chunk 中的 module 的渲染模板。

在开始渲染之前，Compilation 实例会调用 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，createHash 也会为每一个 chunk 也创建一个 hash，对应 output.filename 的 [chunkhash] 占位符。

每个 hash 的影响因素比较多，首先三个模板对象会调用 updateHash 方法来更新 hash，在内部还会触发任务点 hash，传递 hash 到其他插件。 chunkhash 也是类似的原理：

```
class Compilation extends Tapable {
    // 其他代码..
    createHash() {
        // 其他代码..
        const hash = crypto.createHash(hashFunction);
        if(outputOptions.hashSalt)
        hash.update(outputOptions.hashSalt);
        this.mainTemplate.updateHash(hash);
        this.chunkTemplate.updateHash(hash);
        this.moduleTemplate.updateHash(hash);
        // 其他代码..
```

```
        for(let i = 0; i < chunks.length; i++) {
            const chunk = chunks[i];
            const chunkHash = crypto.createHash(hashFunction);
            if(outputOptions.hashSalt)
            chunkHash.update(outputOptions.hashSalt);
            chunk.updateHash(chunkHash);
            if(chunk.hasRuntime()) {
                this.mainTemplate.updateHashForChunk(chunkHash, chunk);
            } else {
                this.chunkTemplate.updateHashForChunk(chunkHash, chunk);
            }
            this.applyPlugins2("chunk-hash", chunk, chunkHash);
            chunk.hash = chunkHash.digest(hashDigest);
            hash.update(chunk.hash);
            chunk.renderedHash = chunk.hash.substr(0, hashDigestLength);
        }
        this.fullHash = hash.digest(hashDigest);
        this.hash = this.fullHash.substr(0, hashDigestLength);
    }
}
```

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

```
class Compilation extends Tapable {
    // 其他代码..
    createChunkAssets() {
        // 其他代码..
        for(let i = 0; i < this.chunks.length; i++) {
            const chunk = this.chunks[i];
            // 其他代码..
            if(chunk.hasRuntime()) {
                source = this.mainTemplate.render(this.hash, chunk,
this.moduleTemplate, this.dependencyTemplates);
            } else {
                source = this.chunkTemplate.render(chunk, this.moduleTemplate,
this.dependencyTemplates);
            }
            file = this.getPath(filenameTemplate, {
```

```
                noChunkHash: !useChunkHash,
                chunk
            });
            this.assets[file] = source;
            // 其他代码..
        }
    }
}
```

这里注意到 ModuleTemplate 实例会被传递下去，在实际渲染时将会用 ModuleTemplate 来渲染每一个 module，其实更多是往 module 前后添加一些"包装"代码，因为 module 的源码实际上是已经渲染完毕的。

MainTemplate 的渲染跟 ChunkTemplate 的不同点在于，入口 chunk 的源码中会带有启动 webpack 的代码，而非入口 chunk 的源码是不需要的。这个只要查看 webpack 构建后的文件就可以比较清楚地看到区别：

```
// 入口 chunk
/******/ (function(modules) { // webpackBootstrap
/******/     // install a JSONP callback for chunk loading
/******/     var parentJsonpFunction = window["webpackJsonp"];
/******/     window["webpackJsonp"] = function webpackJsonpCallback(chunkIds,
moreModules, executeModules) {
/******/         // add "moreModules" to the modules object,
/******/         // then flag all "chunkIds" as loaded and fire callback
/******/         var moduleId, chunkId, i = 0, resolves = [], result;
/******/         for(;i < chunkIds.length; i++) {
/******/             chunkId = chunkIds[i];
/******/             if(installedChunks[chunkId]) {
/******/                 resolves.push(installedChunks[chunkId][0]);
/******/             }
/******/             installedChunks[chunkId] = 0;
/******/         }
/******/         for(moduleId in moreModules) {
/******/             if(Object.prototype.hasOwnProperty.call(moreModules,
moduleId)) {
/******/                 modules[moduleId] = moreModules[moduleId];
/******/             }
/******/         }
```

```
/******/         if(parentJsonpFunction) parentJsonpFunction(chunkIds,
moreModules, executeModules);
/******/         while(resolves.length) {
/******/             resolves.shift()();
/******/         }
/******/
/******/     };
/******/     // 其他代码..
/******/ })(/* modules 代码 */);

// 动态引入的 chunk
webpackJsonp([0],[
    /* modules 代码.. */
]);
```

当每个 chunk 的源码生成之后，就会添加在 Compilation 实例的 assets 属性中。

assets 对象的 key 是最终要生成的文件名称，因此这里要用到前面创建的 hash。调用 Compilation 实例内部的 getPath 方法会根据配置中的 output.filename 来生成文件名称。

assets 对象的 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。此时 Compilation 实例还会触发几个任务点，例如 addtional-chunk-assets，addintial-assets 等，在这些任务点可以修改 assets 属性来改变最终要生成的文件。

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，接下来只有文件生成的步骤。**

在 Compiler 实例开始生成文件前，**最后一个修改最终文件生成的任务点 emit 会被触发：**

```
// 监听 emit 任务点，修改最终文件的最后机会
compiler.plugin("emit", (compilation, callback) => {
    let data = "abcd"
    compilation.assets["newFile.js"] = {
        source() {
            return data
```

```
        }
        size() {
            return data.length
        }
    }
})
```

当任务点 emit 被触发之后，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。

总结

我们将 webpack 的基本架构以及核心的构建流程都过了一遍，希望在阅读完全文之后，对大家了解 webpack 原理有所帮助。