---

link: null
title: 珠峰架构师成长计划
description: 插件向第三方开发者提供了 webpack 引擎中完整的能力。使用阶段式的构建回调，开发者可以引入它们自己的行为到 webpack 构建流程中。创建插件比创建 loader 更加高级，因为你将需要理解一些 webpack 底层的内部特性来做相应的钩子
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=69 sentences=153, words=849

---

1. plugin

插件向第三方开发者提供了 webpack 引擎中完整的能力。使用阶段式的构建回调，开发者可以引入它们自己的行为到 webpack 构建流程中。创建插件比创建 loader 更加高级，因为你将需要理解一些 webpack 底层的内部特性来做相应的钩子

1.1 为什么需要一个插件

- webpack基础配置无法满足需求
- 插件几乎能够任意更改webpack编译结果
- webpack内部也是通过大量内部插件实现的

1.2 可以加载插件的常用对象

对象 钩子
Compiler (https://github.com/webpack/webpack/blob/v4.39.3/lib/Compiler.js)

run,compile,compilation,make,emit,done
Compilation (https://github.com/webpack/webpack/blob/v4.39.3/lib/Compilation.js)

buildModule,normalModuleLoader,succeedModule,finishModules,seal,optimize,after-seal
Module Factory (https://github.com/webpack/webpack/blob/master/lib/ModuleFactory.js)

beforeResolver,afterResolver,module,parser Module
Parser (https://github.com/webpack/webpack/blob/master/lib/Parser.js)

] program,statement,call,expression
Template (https://github.com/webpack/webpack/blob/master/lib/Template.js)

hash,bootstrap,localVars,render

2. 创建插件

webpack 插件由以下组成：

- 一个 JavaScript 命名函数。
- 在插件函数的 prototype 上定义一个 apply 方法。
- 指定一个绑定到 webpack 自身的事件钩子。
- 处理 webpack 内部实例的特定数据。
- 功能完成后调用 webpack 提供的回调。

3. Compiler 和 Compilation

在插件开发中最重要的两个资源就是 `compiler`和 `compilation`对象。理解它们的角色是扩展webpack引擎重要的第一步。

- compiler 对象代表了完整的 webpack 环境配置。这个对象在启动 webpack 时被一次性建立，并配置好所有可操作的设置，包括 options，loader 和 plugin。当在 webpack 环境中应用一个插件时，插件将收到此 compiler 对象的引用。可以使用它来访问 webpack 的主环境。
- compilation 对象代表了一次资源版本构建。当运行 webpack 开发环境中间件时，每当检测到一个文件变化，就会创建一个新的 compilation，从而生成一组新的编译资源。一个 compilation 对象表现了当前的模块资源、编译生成资源、变化的文件、以及被跟踪依赖的状态信息。compilation 对象也提供了很多关键时机的回调，以供插件做自定义处理时选择使用。

4. 基本插件架构

- 插件是由「具有 apply 方法的 prototype 对象」所实例化出来的
- 这个 apply 方法在安装插件时，会被 webpack compiler 调用一次
- apply 方法可以接收一个 webpack compiler 对象的引用，从而可以在回调函数中访问到 compiler 对象

4.1 使用插件代码

- [使用插件] https://github.com/webpack/webpack/blob/master/lib/webpack.js#L60-L69 (https://github.com/webpack/webpack/blob/master/lib/webpack.js#L60-L69))

```
if (options.plugins && Array.isArray(options.plugins)) {
    for (const plugin of options.plugins) {
        plugin.apply(compiler);
    }
}
```

4.2 Compiler插件

- done: new AsyncSeriesHook(["stats"]) (https://github.com/webpack/webpack/blob/master/lib/Compiler.js#L105)

4.2.1 同步

```
class DonePlugin {
    constructor(options) {
        this.options = options;
    }
    apply(compiler) {
        compiler.hooks.done.tap('DonePlugin', (stats) => {
            console.log('Hello ', this.options.name);
        });
    }
}
module.exports = DonePlugin;
```

4.2.2 异步

```
class DonePlugin {
    constructor(options) {
        this.options = options;
    }
    apply(compiler) {
        compiler.hooks.done.tapAsync('DonePlugin', (stats, callback) => {
            console.log('Hello ', this.options.name);
            callback();
        });
    }
}
module.exports = DonePlugin;
```

4.3 使用插件

- 要安装这个插件，只需要在你的 webpack 配置的 plugin 数组中添加一个实例

```
const DonePlugin=require('./plugins/DonePlugin');
module.exports={
    entry: './src/index.js',
    output: {
        path: path.resolve('build'),
        filename:'bundle.js'
    },
    plugins: [
        new DonePlugin({name:'zfpx'})
    ]
}
```

4.4 触发钩子执行

- done (https://github.com/webpack/webpack/blob/master/lib/Compiler.js#L360-L363)

```
if (this.hooks.shouldEmit.call(compilation)
                const stats = new Stats(compilation);
                stats.startTime = startTime;
                stats.endTime = Date.now();
+                this.hooks.done.callAsync(stats, err => {
+                    if (err) return finalCallback(err);
+                    return finalCallback(null, stats);
+                });
                return;
            }
```

5. compilation插件

- 使用 compiler 对象时，你可以绑定提供了编译 compilation 引用的回调函数，然后拿到每次新的 compilation 对象。这些 compilation 对象提供了一些钩子函数，来钩入到构建流程的很多步骤中

5.1 asset-plugin.js

plugins\asset-plugin.js

```
class AssetPlugin {
    constructor(options) {
        this.options = options;
    }
    apply(compiler) {
        compiler.hooks.compilation.tap('AssetPlugin', function (compilation) {
            compilation.hooks.chunkAsset.tap('AssetPlugin', function (chunk, filename) {
                console.log('filename=', filename);
            });
        });
    }
}
module.exports = AssetPlugin;
```

5.2 compilation.call

- Compiler.js (https://github.com/webpack/webpack/blob/master/lib/Compiler.js#L853-L860)

```
newCompilation(params) {
        const compilation = this.createCompilation();
        this.hooks.compilation.call(compilation, params);
        return compilation;
    }
```

5.3 chunkAsset.call

- chunkAsset.call (https://github.com/webpack/webpack/blob/v4.39.3/lib/Compilation.js#L2019)

```
chunk.files.push(file);
+this.hooks.chunkAsset.call(chunk, file);
```

> 关于 compiler, compilation 的可用回调，和其它重要的对象的更多信息，请查看 插件 (https://webpack.docschina.org/api/compiler-hooks/) 文档。

6. 打包zip

- webpack-sources (https://www.npmjs.com/package/webpack-sources)**6.1 zip-plugin.js #**plugins\zip-plugin.js

```
const { RawSource } = require("webpack-sources");
const JSZip = require("jszip");
const path = require("path");
class ZipPlugin {
    constructor(options) {
        this.options = options;
    }
    apply(compiler) {
        let that = this;
        compiler.hooks.emit.tapAsync("ZipPlugin", (compilation, callback) => {
            var zip = new JSZip();
            for (let filename in compilation.assets) {
                const source = compilation.assets[filename].source();
                zip.file(filename, source);
            }
            zip.generateAsync({ type: "nodebuffer" }).then(content => {
                compilation.assets[that.options.filename] = new RawSource(content);
                callback();
            });
        });
    }
}
module.exports = ZipPlugin;
```

** 6.2 webpack.config.js #**

webpack.config.js

```
  plugins: [
+    new zipPlugin({
+      filename: 'assets.zip'
+    })
]
```

7.自动外链

** 7.1 使用外部类库 #**

- 手动指定 external
- 手动引入 script

> 能否检测代码中的import自动处理这个步骤?

** 7.2 思路 #**

- 解决import自动处理 external和 script的问题，需要怎么实现，该从哪方面开始考虑
- 依赖 当检测到有 import该 library时，将其设置为不打包类似 exteral,并在指定模版中加入script,那么如何检测import？这里就用 Parser
- external依赖 需要了解external是如何实现的，webpack的external是通过插件 ExternalsPlugin实现的，ExternalsPlugin通过 tap NormalModuleFactory 在每次创建Module的时候判断是否是 ExternalModule
- webpack4加入了模块类型之后， Parser获取需要指定类型moduleType,一般使用 javascript/auto即可

** 7.3 使用plugins #**

```
plugins: [
        new HtmlWebpackPlugin({
            template: './src/index.html',
            filename:'index.html'
        }),
        new AutoExternalPlugin({
            jquery: {
                expose: '{{content}}#x27;,
                url: 'https://cdn.bootcss.com/jquery/3.1.0/jquery.js'
            }
        })
    ]
```

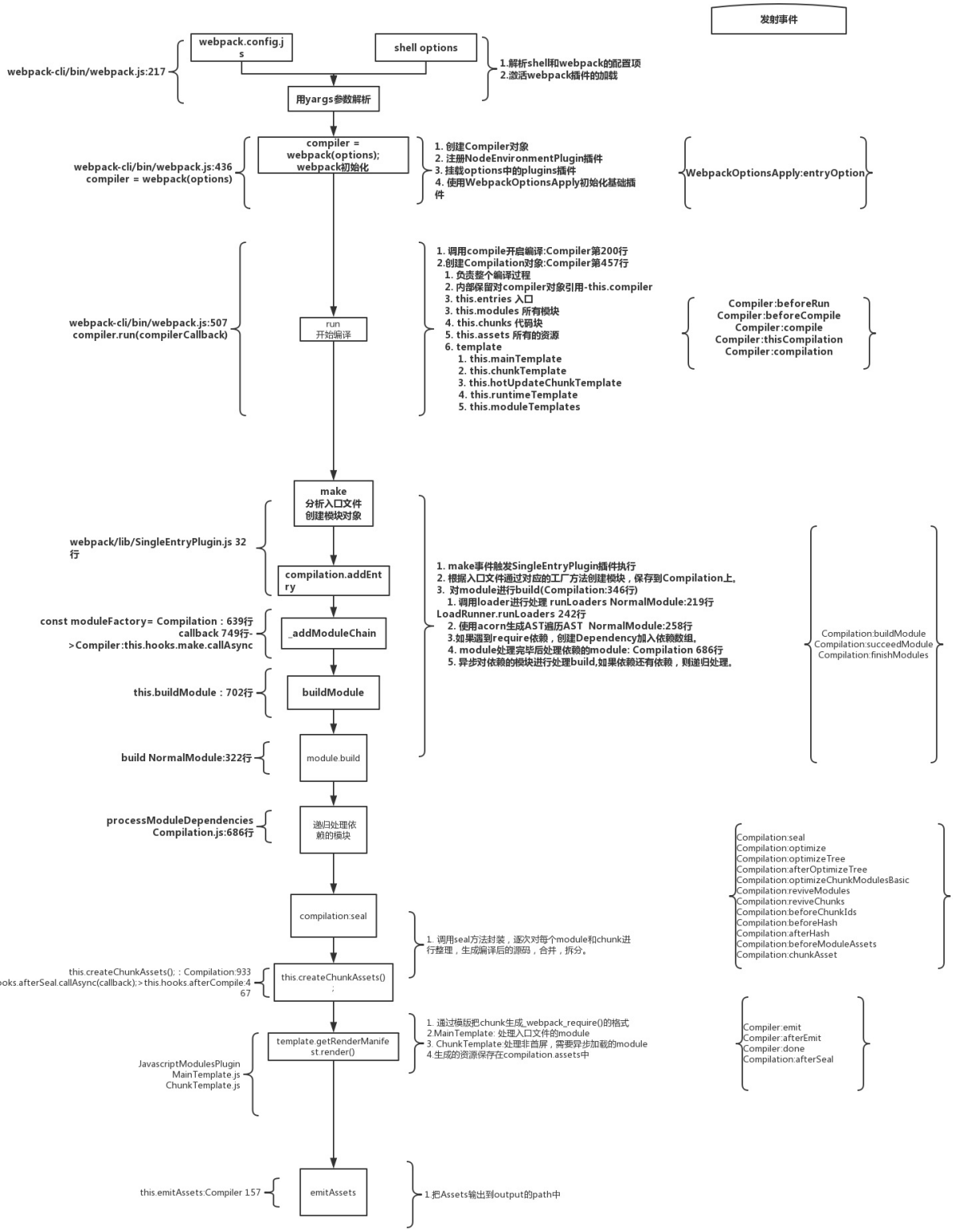
** 7.4 AutoExternalPlugin #**

- ExternalsPlugin.js (https://github.com/webpack/webpack/blob/0d4607c68e04a659fa58499e1332c97d5376368a/lib/ExternalsPlugin.js)
- ExternalModuleFactoryPlugin (https://github.com/webpack/webpack/blob/eeafeee32ad5a1469e39ce66df671e3710332608/lib/ExternalModuleFactoryPlugin.js)
- ExternalModule.js (https://github.com/webpack/webpack/blob/eeafeee32ad5a1469e39ce66df671e3710332608/lib/ExternalModule.js)
- parser (https://github.com/zhufengnodejs/webpack-analysis/blob/master/node_modules/_webpack%404.20.2%40webpack/lib/NormalModuleFactory.js#L87)
- factory (https://github.com/zhufengnodejs/webpack-analysis/blob/master/node_modules/_webpack%404.20.2%40webpack/lib/NormalModuleFactory.js#L66)
- htmlWebpackPluginAlterAssetTags (https://github.com/jantimon/html-webpack-plugin/blob/v3.2.0/index.js#L62)

```
const ExternalModule = require('webpack/lib/ExternalModule');
class AutoExternalPlugin {
    constructor(options) {
        this.options = options;
        this.externalModules = {};
    }
    apply(compiler) {

        compiler.hooks.normalModuleFactory.tap('AutoExternalPlugin', normalModuleFactory => {
            normalModuleFactory.hooks.parser
                .for('javascript/auto')
                .tap('AutoExternalPlugin', parser => {
                    parser.hooks.import.tap('AutoExternalPlugin', (statement, source) => {
                        if (this.options[source])
                            this.externalModules[source] = true;
                    });
                });

            normalModuleFactory.hooks.factory.tap('AutoExternalPlugin', factory => (data, callback) => {
                const dependency = data.dependencies[0];
                let value = dependency.request;
                if (this.externalModules[value]) {
                    let varName = this.options[value].expose;
                    callback(null, new ExternalModule(varName, 'window'));
                } else {
                    factory(data, callback);
                }
            });
        });
        compiler.hooks.compilation.tap('AutoExternalPlugin', compilation => {

            compilation.hooks.htmlWebpackPluginAlterAssetTags.tapAsync('normalModuleFactory', (htmlPluginData, callback) => {
                Object.keys(this.externalModules).forEach(name => {
                    let src = this.options[name].url;
                    htmlPluginData.body.unshift({
                        tagName: 'script',
                        closeTag: true,
                        attributes: { type: 'text/javascript', src }
                    });
                });
                callback(null, htmlPluginData);
            });
        });
    }
}
module.exports = AutoExternalPlugin;
```

8. 参考

- Node.js SDK (https://developer.qiniu.com/kodo/sdk/1289/nodejs)
- writing-a-plugin (https://webpack.js.org/contribute/writing-a-plugin/)
- api/plugins (https://webpack.js.org/api/plugins/)


发射事件
webpack.config.js
shell options
webpack-cli/bin/webpack.js:217
1.解析shell和webpack的配置项
2.激活webpack插件的加载
用yargs参数解析
compiler = webpack(options);
webpack初始化
webpack-cli/bin/webpack.js:436
compiler = webpack(options)
1. 创建Compiler对象
2. 注册NodeEnvironmentPlugin插件
3. 挂载options中的plugins插件
4. 使用WebpackOptionsApply初始化基础插件
WebpackOptionsApply:entryOption
webpack-cli/bin/webpack.js:507
compiler.run(compilerCallback)
run
开始编译
1. 调用compile开启编译:Compiler第200行
2.创建Compilation对象:Compiler第457行
1. 负责整个编译过程
2. 内部保留对compiler对象引用-this.compiler
3. this.entries 入口
3. this.modules 所有模块
4. this.chunks 代码块
5. this.assets 所有的资源
6. template
1. this.mainTemplate
2. this.chunkTemplate
3. this.hotUpdateChunkTemplate
4. this.runtimeTemplate
5. this.moduleTemplates
Compiler:beforeRun
Compiler:beforeCompile
Compiler:compile
Compiler:thisCompilation
Compiler:compilation
make
分析入口文件
创建模块对象
webpack/lib/SingleEntryPlugin.js 32行
compilation.addEntry
const moduleFactory= Compilation：639行
callback 749行->Compiler:this.hooks.make.callAsync
_addModuleChain
this.buildModule：702行
buildModule
build NormalModule:322行
module.build
processModuleDependencies
Compilation.js:686行
递归处理依赖的模块
1. make事件触发SingleEntryPlugin插件执行
2. 根据入口文件通过对应的工厂方法创建模块，保存到Compilation上,
3. 对module进行build(Compilation:346行)
1. 调用loader进行处理 runLoaders NormalModule:219行
LoadRunner.runLoaders 242行
2. 使用acorn生成AST遍历AST NormalModule:258行
3.如果遇到require依赖，创建Dependency加入依赖数组。
4. module处理完毕后处理依赖的module: Compilation 686行
5. 异步对依赖的模块进行处理build,如果依赖还有依赖，则递归处理。
Compilation:buildModule
Compilation:succeedModule
Compilation:finishModules
compilation:seal
1. 调用seal方法封装，逐次对每个module和chunk进行整理，生成编译后的源码，合并，拆分。
Compilation:seal
Compilation:optimize
Compilation:optimizeTree
Compilation:afterOptimizeTree
Compilation:optimizeChunkModulesBasic
Compilation:reviveModules
Compilation:reviveChunks
Compilation:beforeChunkIds
Compilation:beforeHash
Compilation:afterHash
Compilation:beforeModuleAssets
Compilation:chunkAsset
this.createChunkAssets(): Compilation:933
this.hooks.afterSeal.callAsync(callback):> this.hooks.afterCompile:467
this.createChunkAssets();
template.getRenderManifest.render()
JavascriptModulesPlugin
MainTemplate.js
ChunkTemplate.js
1. 通过模版把chunk生成_webpack_require()的格式
2.MainTemplate: 处理入口文件的module
3. ChunkTemplate:处理非首屏，需要异步加载的module
4.生成的资源保存在compilation.assets中
Compiler:emit
Compiler:afterEmit
Compiler:done
Compilation:afterSeal
this.emitAssets:Compiler 157
emitAssets
1.把Assets输出到output的path中