了解Webpack编译流程

前置知识

- **webpack** 是 JavaScript 应用程序静态资源模块打包器。它的源码设计采用了插件架构，由 `Tapable` 提供注册和调用插件的能力。因此，webpack 中应用了很多内置插件，来完成对每一个 config 配置项功能的实现。
- **compiler** 理解为**编译器**，仅在 webpack 初始化时创建一次实例，是 webpack 打包流程上的支柱引擎。在 compiler 实例上提供了大量 `hooks` 来面向用户实现自定义插件，在 `run` 启动打包之后会创建 `compilation` 实例处理模块的编译工作。
- **compilation** 理解为 **编译**，是由 `compiler` 编译器创建而成，一个编译器可能会创建一次或多次编译，则 `compilation` 可能会被创建多次。compilation 会从入口模块开始，编译模块以及它的依赖子模块。所有的模块的编译都会经过：加载(loaded)、封存(sealed)、优化(optimized)、分块(chunked)、哈希(hashed) 和 重新创建(restored)。
- **Dependence**：webpack 用于它来记录模块间依赖关系。在模块中引用其它模块，会将引用关系表述为Dependency 子类并关联 module 对象，等到当前 module 内容都解析完毕之后，启动下次循环开始将 Dependency 对象转换为新的 Module 子类。
- **Module**：webpack 处理每一个资源文件时，都会以 module 对象形式存在，包含了资源的路径、上下文、依赖、内容等信息。所有关于资源的构建、转译、合并也都是以 module 为基本单位进行。
- **Chunk**：编译完成准备输出时，webpack 会将 module 按特定的规则组织成一个一个的 chunk，这些 chunk 某种程度上跟最终输出的文件一一对应。
- **Assets**： asset 代表了最终要输出写入磁盘的文件内容，它与 chunk 一一对应。
- **Tapable**：Tapable 提供注册和调用插件的能力，来串联 webapck 整个打包流程的插件工作。

调试环境

1. git clone webpack源码
2. 安装依赖 `yarn install`
3. 新建 `dmeo` 文件夹
4. 创建简单的webpack.config.js配置文件

```
1 //本配置以打包vue项目调试
2 //需要安装less,vue-loader,vue-template-compiler, webpack,webpack-cli
```

```
//src目录下写业务代码
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { VueLoaderPlugin } = require("vue-loader/dist/index");

module.exports = {
  mode: "development",
  devtool: "source-map",
  context: path.join(__dirname, "."),
  entry: "./src/main.js",
  output: {
    path: path.join(__dirname, "dist"),
    filename: "js/[name]_[contentHash:10].js",
    clean: true
  },
  module: {
    rules: [
      {
        test: /\.less$/,
        use: ["style-loader", "css-loader", "less-loader"]
      },
      {
        test: /\.js$/,
        loader: "babel-loader"
      },
      {
        test: /\.vue$/,
        loader: "vue-loader"
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: path.join(__dirname, "./public/index.html"),
      filename: "index.html",
      title: "webpack source"
    }),
    new VueLoaderPlugin()
  ]
};
```

5. demo下新建build.js文件，作为webpack的执行文件

```
const webpack = require("../lib/webpack.js");
const config = require("./webpack.config.js");
```

```
3
4  //1.创建一个对象：compiler--编译对象
5  //另外一个非常重要的对象：compilation
6  const compiler = webpack(config);
7
8  //2.执行run方法，开始对代码进行编译和打包
9  compiler.run((err, stat) => {
10   if (err) {
11     console.log(`err${err}`);
12   } else {
13     console.log(stat);
14   }
15 });
16
```

6. 打断点调试

流程概览

1. 初始化阶段
2. 构建编译阶段（make）
3. 生成阶段（seal）
4. 写入阶段（emit）

初始化阶段

> ❗
> 1. 生成compile编译器、watch监听器
> 2. 注册所有的plugin：传入的(entry、output也会转化为plugin)、内置的
> 3. 初始化编译环境

主要步骤

1. 初始化参数：将用户传入配置与默认配置结合得到最终配置参数；
2. 创建编译器对象：根据配置参数创建 `Compiler` 实例对象；
3. 初始化编译环境：注册用户配置插件及内置插件；
4. 运行编译：执行 `compiler.run` 方法；

5. 确定入口：根据配置 `entry` 找寻所有入口文件，并转换为 `dependence` 对象，等待执行 `compilition.addEntry` 编译工作。

webpack(config,callback)入口函数

```
//webpack(config, callback)
//根据是否传入callback来判断，传入callback则自动执行run方法，否则手动执行

const webpack = (options, callback) => {
  const create = () => {
    const webpackOptions = options;
    const compiler = createCompiler(webpackOptions);
    return compiler
  }

  if (callback) {
    const { compiler } = create();
    compiler.run((err, stats) => {
      compiler.close(err2 => {
        callback(err || err2, stats);
      });
    });
    return compiler;
  } else {
    const { compiler } = create();
    return compiler;
  }
}

```

createCompiler(options)创建编译器

```
//重要的函数逻辑
const createCompiler = rawOptions => {
  //合并配置参数 （默认的，传入的）
  const options = getNormalizedWebpackOptions(rawOptions);
  applyWebpackOptionsBaseDefaults(options);
  //1.通过new方法创建一个compiler的实例
  const compiler = new Compiler(
    /** @type {string} */ (options.context),
```

```
    options
  );
  //初始化编译环境
  new NodeEnvironmentPlugin({
    infrastructureLogging: options.infrastructureLogging
  }).apply(compiler);
  //2.注册我们传入的所有的webpack插件
  if (Array.isArray(options.plugins)) {
    for (const plugin of options.plugins) {
      if (typeof plugin === "function") {
        //函数
        plugin.call(compiler, compiler);
      } else if (plugin) {
        //对象必须要有apply方法
        plugin.apply(compiler);
      }
    }
  }
  applyWebpackOptionsDefaults(options);

  //3.调用钩子environment/afterEnvironment函数
  //environment钩子：执行webpack编译构建前的注册到environment钩子的所有函数
  compiler.hooks.environment.call();
  //afterEnvironment钩子：执行初始化环境之后（构建之前）的afterEnvironment钩子的所有函数
  compiler.hooks.afterEnvironment.call();
  //4.使用process函数处理其他选项，比如entry/output/devtool等 让webpack传入的其它选项起作用
  new WebpackOptionsApply().process(options, compiler);
  compiler.hooks.initialize.call();
  return compiler;
};
```

这里重点提及两处：

1. new Compiler(options.context, options) Compiler 是一个 ES6 class 构造函数，到这里我们只认识到了一个 run 方法，它的基础结构如下：

```
// webpack/lib/Compiler.js
class Compiler {
  constructor(context, options = {}) {
    this.hooks = Object.freeze({
      initialize: new SyncHook([]),
      run: new AsyncSeriesHook(["compiler"]),
      done: new AsyncSeriesHook(["stats"]),
```

```
      emit: new AsyncSeriesHook(["compilation"]),
      make: new AsyncParallelHook(["compilation"]),
      ... 很多很多
    });
    this.options = options;
    this.context = context;
  }
  run(callback) {}
}

```

2. new WebpackOptionsApply().process(options, compiler) 上面「前置知识」中我们了解到：webpack 是一个插件结构化的设计架构，即每一个功能的实现都是由一个插件来完成的，比如模块的编译入口 `entry` 是由 `EntryPlugin` 来管理和执行。

`WebpackOptionsApply().process` 中注册的内置插件很多，这里我们只关心会涉及到的插件配置。

```
// webpack/lib/WebpackOptionsApply.js
class WebpackOptionsApply extends OptionsApply {
  constructor() {
    super();
  }
  process(options, compiler) {
    new JavascriptModulesPlugin().apply(compiler);
    // entry 插件
    new EntryOptionPlugin().apply(compiler);
    compiler.hooks.entryOption.call(options.context, options.entry);
    ...
  }
}
```

- `EntryOptionPlugin` 会为 `config.entry` 中的每个配置应用 `EntryPlugin` 来注册编译入口，等后续 `hooks.make` 时机开始入口模块的编译。
- `JavascriptModulesPlugin` 提供了 `parse` AST 的核心实现，后续收集模块内的所引入的 `deps` 依赖模块时会用到。

到这里，compiler 实例创建完成，并且将相关插件注册成功。

接下来会执行 `compiler.run()` 开启打包。由于还没有走到真正的编译，将这部分内容放在初始化阶段一并介绍。

compiler.run(callback)

```
// webpack/lib/Compiler.js
class Compiler {
  ...
  run(callback) {
    const finalCallback = (err, stats) => {} // 所有工作都完成后的最终执行函数
    //等到compilation 执行完所有函数之后会执行的回调函数
    const onCompiled = (err, compilation) => {};

    //beforeRun ==> run ==> compile
    this.hooks.beforeRun.callAsync(this, err => {
      if (err) return finalCallback(err);
      this.hooks.run.callAsync(this, err => {
        if (err) return finalCallback(err);
        this.compile(onCompiled);
      });
    });
  }
}

```

首先执行了 `beforeRun` 和 `run` 两个 hook 钩子，如果有插件中注册了这两类钩子，注册的回调函数就会立刻执行。下面移步到 `this.compile` 之中。

```
// webpack/lib/Compiler.js
class Compiler {
  ...
  compile(callback) {
    const params = this.newCompilationParams();
    //1. 创建compilation对象
    const compilation = this.newCompilation(params);

    //顺序: beforeCompile ==> compile ==> make ==> finalMake ==> seal ==> afterCompile ==> done ==> callback
    this.hooks.beforeCompile.callAsync(params, err => {
      this.hooks.compile.call(params);
      const compilation = this.newCompilation(params);
      //2. 执行compile 派发事件 callAsync  tapAsync监听事件
      this.hooks.make.callAsync(compilation, err => {
        compilation.finish(err => {
          compilation.seal(err => {
            return callback(null, compilation);
```

```
          });
        });
      });
    });
  }
}
```

`compile()` 是启动编译的关键，编译实例 `compilation` 的参数定义、实例创建以及编译完成后的收尾工作都在这里实现。

1. 首先是 `this.newCompilationParams` 创建 compilation 编译模块所需的参数：

```
// webpack/lib/Compiler.js
newCompilationParams() {
  const params = {
    normalModuleFactory: this.createNormalModuleFactory(),
    contextModuleFactory: this.createContextModuleFactory()
  };
  return params;
}
```

这里，我们需要留意一下 `createNormalModuleFactory`，在 webpack 中，每一个依赖模块都可以看作是一个 `Module` 对象，通常会有很多模块要处理，这里创建一个模块工厂 Factory。

```
// webpack/lib/Compiler.js
createNormalModuleFactory() {
  const normalModuleFactory = new NormalModuleFactory({
    context: this.options.context,
    fs: this.inputFileSystem,
    resolverFactory: this.resolverFactory, // resolve 模块时使用
    options: this.options.module,
    associatedObjectForCache: this.root,
    layers: this.options.experiments.layers
  });
  this._lastNormalModuleFactory = normalModuleFactory;
  this.hooks.normalModuleFactory.call(normalModuleFactory);
  return normalModuleFactory;
}
```

2. 创建 `compilation` 实例对象：

```
// webpack/lib/Compiler.js
newCompilation(params) {
  this._cleanupLastCompilation(); // 清除上次 compilation
  const compilation = this._lastCompilation = new Compilation(this, params);
  this.hooks.thisCompilation.call(compilation, params);
  this.hooks.compilation.call(compilation, params);
  return compilation;
}
```

`Compilation` 和 `Compiler` 都是一个 class 构造函数，实例上也包含了非常多的属性和方法。

```
// webpack/lib/Compilation.js
class Compilation {
  constructor(compiler, params) {
    this.hooks = Object.freeze({ ... });
    this.compiler = compiler;
    this.params = params;
    this.options = compiler.options;
    this.entries = new Map(); // 存储 entry module
    this.modules = new Set();
    this._modules = new Map(); // 存储所有 module
    ...
  }
}
```

3. 调用 `hooks.make` 这一步很关键，在上面创建得到 `compilation` 之后，就可以进入编译阶段，编译会从入口模块开始进行。

入口模块的准备工作是在注册的 `EntryOptionPlugin` 之中：

```
// webpack/lib/EntryOptionsPlugin.js
class EntryOptionPlugin {
  apply(compiler) {
    compiler.hooks.entryOption.tap("EntryOptionPlugin", (context, entry) => {
      EntryOptionPlugin.applyEntryOption(compiler, context, entry);
      return true;
    });
  }
  static applyEntryOption(compiler, context, entry) {
    const EntryPlugin = require("./EntryPlugin");
    for (const name of Object.keys(entry)) {
      const desc = entry[name];
```

```
      const options = EntryOptionPlugin.entryDescriptionToOptions(compiler, name, desc);
      for (const entry of desc.import) {
        new EntryPlugin(context, entry, options).apply(compiler); // 注册 entry 插件
      }
    }
  }
}
```

其中关键部分是为每个 entry 注册 `EntryPlugin`，在 EntryPlugin 中就会看到与 `hook.make` 相关的逻辑：

```
// webpack/lib/EntryPlugin.js
class EntryPlugin {
  apply(compiler) {
    // 1、记录 entry 模块解析时使用 normalModuleFactory
    compiler.hooks.compilation.tap(
      "EntryPlugin",
      (compilation, { normalModuleFactory }) => {
        compilation.dependencyFactories.set(
          EntryDependency, // key
          normalModuleFactory // value
        );
      }
    );
    const { entry, options, context } = this;
    // 2、为 entry 创建 Dependency 对象
    const dep = EntryPlugin.createDependency(entry, options);
    // 3、监听 hook.make，执行 compilation.addEntry
    compiler.hooks.make.tapAsync("EntryPlugin", (compilation, callback) => {
      compilation.addEntry(context, dep, options, err => {
        callback(err);
      });
    });
  }
}
```

当执行 `hooks.make.callAsync` 时，其实就是执行 `compilation.addEntry` 开始入口模块的编译构建阶段。这也是很关键的一步：找到入口进行构建。

构建阶段

`hooks.make` 是触发入口模块编译的开始，在 webpack 的构建阶段，流程如下：

1. 为 `entry` 入口模块创建 entryData 存储在 `compilation.entries`，用于后续为每个入口输出 `chunk`；
2. 拿到处理 `entry` 模块的工厂方法 `moduleFactory`，开始 `ModuleTree` 的创建，后面每个文件模块都会先生成一个 `Module`；
3. 执行 `handleModuleCreation` 开始处理入口模块的构建，当然，入口模块中所引入的依赖模块，构建也都是从这里开始；
4. 构建过程会经历 `feactorize（创建 module）`、`addModule`、`buildModule` 三个阶段，build 阶段涉及到 loader 代码转换和依赖收集；
5. 模块构建完成后，若存在子依赖（`module.dependencies`），回到第三步开始子依赖的构建。

构建 EntryModuleTree

从 `compilation.addEntry` 开始进入 entry 模块的编译，调用 `_addEntryItem` 创建 `entryData` 加入到 `this.entries` 集合中

```
// webpack/lib/Compilation.js
class Compilation {
  constructor(compiler, params) {
    this.entries = new Map();
    ...
  }

  addEntry(context, entry, options, callback) {
    //私有方法
    this._addEntryItem(context, entry, "dependencies", options, callback);
  }

  _addEntryItem(context, entry, target, options, callback) {
    //将入口添加到模块树中
    this.addModuleTree({ context, dependency: entry, contextInfo: undefined },
(err, module) => {
      this.hooks.succeedEntry.call(entry, options, module);
      return callback(null, module);
    });
  }
}
```

接着，执行 `addModuleTree` 获取 `moduleFactory` 即上文存储的 `normalModuleFactory`

```
// webpack/lib/Compilation.js
addModuleTree({ context, dependency, contextInfo }, callback) {
  //分析依赖
  const Dep = dependency.constructor;
  // dependencyFactories.get(EntryDependency) = normalModuleFactory
  const moduleFactory = this.dependencyFactories.get(Dep); // 用于后续执行 moduleFactory.create()
  this.handleModuleCreation({
    factory: moduleFactory,
    dependencies: [dependency],
    originModule: null, contextInfo, context
  }, (err, result) => {
    callback(null, result);
  });
}
```

然后，执行 `handleModuleCreation`：

```
// webpack/lib/Compilation.js
//处理模块、并且创建模块
handleModuleCreation(
  {
    factory, // moduleFactory
    dependencies, // [dep]
    ...
  },
  callback
) {
  const moduleGraph = this.moduleGraph;
  this.factorizeModule(
    {
      currentProfile: false,
      factory,
      dependencies,
      factoryResult: true,
      originModule,
      contextInfo,
      context
    },
    (err, factoryResult) => {
      const newModule = factoryResult.module;
      this.addModule(newModule, (err, module) => {
        ...
      });
    }
```

```
  );
}
```

`factorizeModule` 有分解模块的意思，可以理解为：为 entry 创建一个 `Module`。它的函数体逻辑十分简单：

```
// webpack/lib/Compilation.js
// 因式分解、将模块分解进行处理
Compilation.prototype.factorizeModule = function (options, callback) {
  this.factorizeQueue.add(options, callback);

  //添加模块到模块队列中
  this.addModule(newModule, (err, module) => {})

  //处理模块的依赖和构建
  this._handleModuleBuildAndDependencies(
    originModule,
    module,
    recursive,
    callback
  );
}
```

模块编译所经历的阶段

一个模块的编译会经过 `factorize 创建模块`、`addModule 添加模块`、`buildQueue 构建模块`、`needBuild`、`module.Build` 和 `processDependencies 递归处理子依赖模块（如果有）` 几个阶段。

而每个阶段的真正执行函数绑定在 `Queue.processor` 处理器上。

```
// webpack/lib/Compilation.js
class Compilation {
  constructor(compiler, params) {
    this.processDependenciesQueue = new AsyncQueue({
      name: "processDependencies",
      parallelism: options.parallelism || 100,
      processor: this._processModuleDependencies.bind(this)
    });
    this.addModuleQueue = new AsyncQueue({
      name: "addModule",
      parent: this.processDependenciesQueue,
      getKey: module => module.identifier(),
```

```
      processor: this._addModule.bind(this)
    });
    this.factorizeQueue = new AsyncQueue({
      name: "factorize",
      parent: this.addModuleQueue,
      processor: this._factorizeModule.bind(this)
    });
    this.buildQueue = new AsyncQueue({
      name: "build",
      parent: this.factorizeQueue,
      processor: this._buildModule.bind(this)
    });
  }
  _processModuleDependencies(module, callback) { }
  _addModule(module, callback) { }
  _factorizeModule(params, callback) { }
  _buildModule(module, callback) { }
}
```

factorize 创建模块

```
// webpack/lib/Compilation.js
_factorizeModule(
  {
    currentProfile,
    factory,
    dependencies,
    originModule,
    factoryResult,
    contextInfo,
    context
  },
  callback
) {
  factory.create({ context, dependencies, ... }, (err, result) => {
    callback(null, factoryResult ? result : result.module);
  });
}
```

addModule 存储模块

```
// webpack/lib/Compilation.js
_addModule(module, callback) {
```

```
  const identifier = module.identifier();
  const alreadyAddedModule = this._modules.get(identifier);
  if (alreadyAddedModule) {
    return callback(null, alreadyAddedModule);
  }

  this._modulesCache.get(identifier, null, (err, cacheModule) => {
    if (cacheModule) {
      cacheModule.updateCacheModule(module);
      module = cacheModule;
    }
    this._modules.set(identifier, module);
    this.modules.add(module);
    callback(null, module);
  });
}
```

buildModule 构建模块

```
// webpack/lib/Compilation.js
_handleModuleBuildAndDependencies(originModule, module, recursive, callback) {
  this.buildModule(module, err => {
    ...
  })
}
```

processModuleDependencies

如果模块存在 `dependencies` 依赖，则会对子模块调用 `handleModuleCreation()` 进行上述构建步骤，否则执行 callback 模块编译结束。

```
// webpack/lib/Compilation.js
_processModuleDependencies(module, callback) {
  // 没有要处理的依赖
  if (sortedDependencies.length === 0 && inProgressTransitive === 1) {
    return callback();
  }
  // 处理子依赖
  for (const item of sortedDependencies) {
    this.handleModuleCreation(item, err => {
      ...
```

```
    });
  }
}
```

compilation.finish

进入了 `compilation.finish` 意味着模块的 `make` 打包制作阶段完成。在这里，调用 `hooks.finishModules` 并收集模块构建过程中产生的 `errors` 和 `warnings`。

```
// webpack/lib/Compilation.js
class Compilation {
  constructor(compiler, params) {
    this.errors = [];
    this.warnings = [];
  }
  finish(callback) {
    this.factorizeQueue.clear();
    const { modules } = this;
    this.hooks.finishModules.callAsync(modules, err => {
      for (const module of modules) {
        // 收集 error
        const errors = module.getErrors();
        if (errors !== undefined) {
          for (const error of errors) {
            this.errors.push(error);
          }
        }
        // 收集 warning
        const warnings = module.getWarnings();
        if (warnings !== undefined) {
          for (const warning of warnings) {
            this.warnings.push(warning);
          }
        }
      }
      this.moduleGraph.unfreeze();
      callback();
    });
  }
}
```

生成阶段

在经过 **构建阶段** 后，我们可以在 `compilation` 实例上拿到 `entires`、 `modules` 以及每个 module 的源代码。进入 `seal` 阶段，就是根据 `entires` 创建对应 `chunk` 文件，并将它所依赖 module 的代码拼接生成 `assets` 对象。

`seal` 意为密封的意思，是从 `module` 到 `chunk` 再到 `assets` 的转换过程。

```
暂未过流程
```

写入阶段

`compiler.emitAssets` 是输入阶段的开始，在生成阶段 `seal` 确定好输出内容后，根据配置的 output，将文件内容写入到文件系统。

走到这里，就是对编译完成的 `Assets` 进行写入输出。具体分为以下几步：

1. 调用 `this.emitAssets`，先执行 `compiler.hooks.emit`，再根据 config.output 创建输出目录，遍历 `asstes` 对资源文件进行写入；
2. 执行 `new Stats(compilation)` 生成模块统计数据；
3. 接着触发 `compiler.hooks.done` 通知打包资源写入完成；
4. 最后，将 `stats` 统计数据传入并执行最初调用 `run()` 方法时所传入的 `callback`。

```
// webpack/lib/Compiler.js
const onCompiled = (err, compilation) => {
  // 1. 写入资源
  this.emitAssets(compilation, err => {
    // 2. 生成统计数据
    const stats = new Stats(compilation);
    // 3. 触发 hooks.done
    this.hooks.done.callAsync(stats, err => {
      // 4. 执行 compiler.run 时传入的 callback
      finalCallback(null, stats);
    });
  });
}

emitAssets(compilation, callback) {
  // 触发 hooks.emit
```

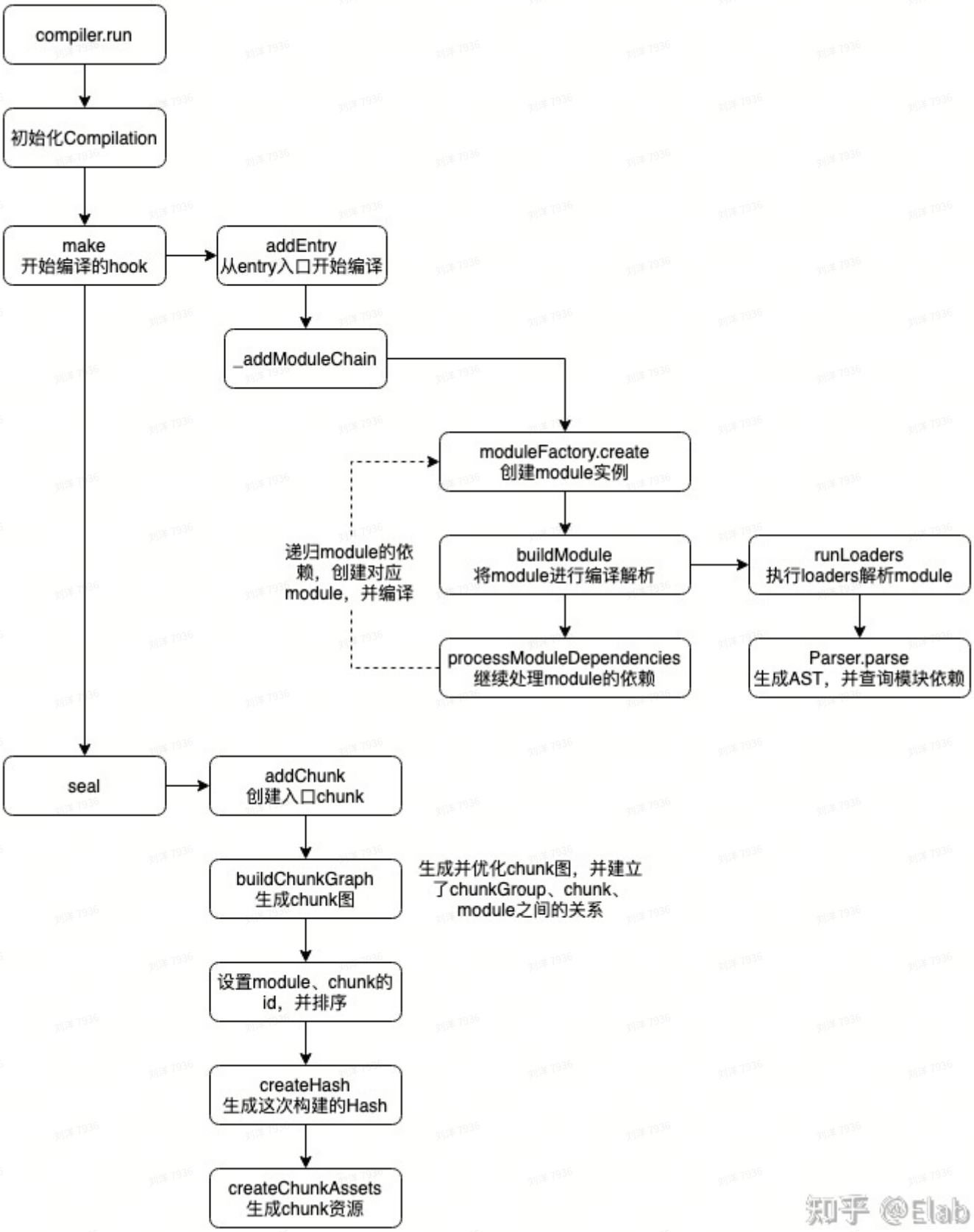
```
  this.hooks.emit.callAsync(compilation, err => {
    // 创建 build 目录。outputPath 为我们的 config.output.path
    mkdirp(this.outputFileSystem, outputPath, emitFiles);
  });
}

const emitFiles = err => {
  // 1. 首先获取 compilation 编译完成的 assets
  const assets = compilation.getAssets();
  // 2. 遍历 assets 进行文件输出
  asyncLib.forEachLimit( // async 异步库，理解为 forEach assets 即可
    assets,
    15, // 一次运行 15 个异步
    ({ name: file, source, info }, callback) => {
      // 3. 若输出资源存在多级目录，依次创建目录，创建完成后执行 writeOut 写入
      if (targetFile.match(/\/|\\/)) {
        const fs = this.outputFileSystem;
        const dir = dirname(fs, join(fs, outputPath, targetFile));
        mkdirp(fs, dir, writeOut);
      } else {
        writeOut();
      }
    },
    err => {
      // 4. 写入完成
      this.hooks.afterEmit.callAsync(compilation, err => {
        return callback();
      });
    }
  )
}

const writeOut = err => {
  // 1. 拼接 output.path 得到一个绝对路径
  const targetPath = join(this.outputFileSystem, outputPath, targetFile);
  // 2. 判断文件是否存在，不存在则执行 processMissingFile
  this.outputFileSystem.stat(targetPath, (err, stats) => {
    const exists = !err && stats.isFile();
    if (exists) { // build 目录下存在这个文件
      processExistingFile(stats);
    } else {
      processMissingFile();
    }
  });
}

const processMissingFile = () => {
```

```
64    const getContent = () => {
65      if (typeof source.buffer === "function") {
66        return source.buffer();
67      } else {
68        const bufferOrString = source.source();
69        if (Buffer.isBuffer(bufferOrString)) {
70          return bufferOrString;
71        } else {
72          return Buffer.from(bufferOrString, "utf8");
73        }
74      }
75    };
76    // 1. 获取文件内容
77    const content = getContent();
78    // 2. 写入磁盘
79    return doWrite(content);
80  };
81
82  const doWrite = content => {
83    this.outputFileSystem.writeFile(targetPath, content, err => {
84      compilation.emittedAssets.add(file);
85      this.hooks.assetEmitted.callAsync(file);
86    })
87  }
88
89  const finalCallback = (err, stats) => {
90    this.running = false;
91    if (callback !== undefined) callback(err, stats);
92    this.hooks.afterDone.call(stats);
93  };
94
```

Webpack的构建流程总结

1. 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；
2. 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；
3. 确定入口：根据配置中的 entry 找出所有的入口文件；
4. 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行编译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；
5. 完成模块编译：在经过第4步使用 Loader 编译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；

6. 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
7. 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。


compiler.run
初始化Compilation
make
开始编译的hook
addEntry
从entry入口开始编译
_addModuleChain
moduleFactory.create
创建module实例
递归module的依赖，创建对应module，并编译
buildModule
将module进行编译解析
runLoaders
执行loaders解析module
processModuleDependencies
继续处理module的依赖
Parser.parse
生成AST，并查询模块依赖
seal
addChunk
创建入口chunk
buildChunkGraph
生成chunk图
生成并优化chunk图，并建立了chunkGroup、chunk、module之间的关系
设置module、chunk的id，并排序
createHash
生成这次构建的Hash
createChunkAssets
生成chunk资源