深入浅出 Webpack

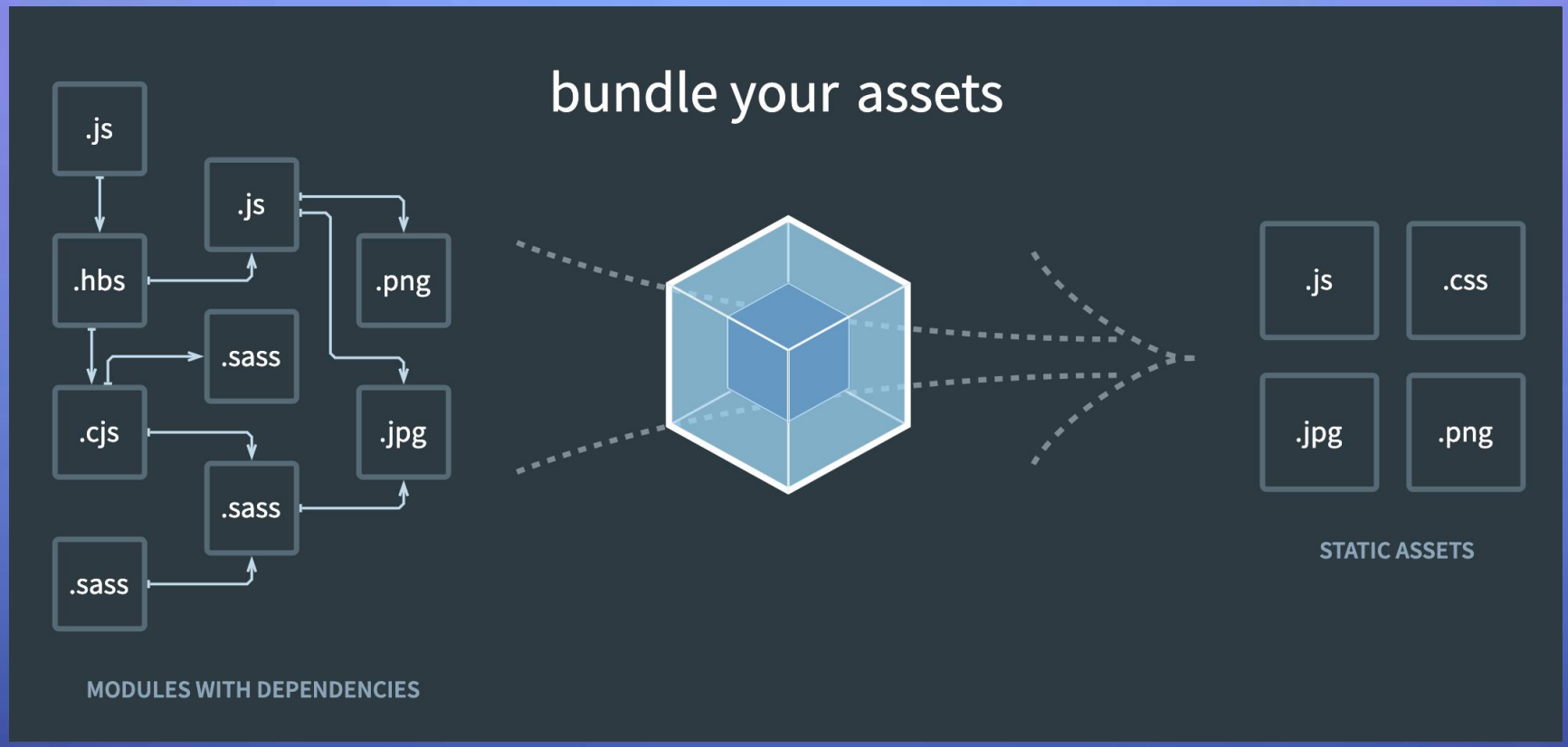
吕新
lvxin5@xiaomi.com

目录

1 什么是 Webpack 4.41.2

一、什么是Webpack

webpack 是一个现代 JavaScript 应用程序的*静态模块打包器(module bundler)*。当 webpack 处理应用程序时，它会递归地构建一个*依赖关系图(dependency graph)*，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundle。一切皆模块，俗称**模块打包机**。



实例: https://github.com/lewiscutey/Webpack-study

二、 基本概念

Entry：入口，Webpack 执行构建的第一步将从 Entry 开始，可抽象成输入，默认值为 ./src

Output：出口，webpack 在哪里输出它所创建的 bundles，以及如何命名这些文件，默认值为 ./dist。

Loader：模块转换器，用于把模块原内容按照需求转换成新内容。

Plugin：扩展插件，在 Webpack 构建流程中的特定时机会广播出对应的事件，插件可以监听这些事件的发生，在特定时机做对应的事情。

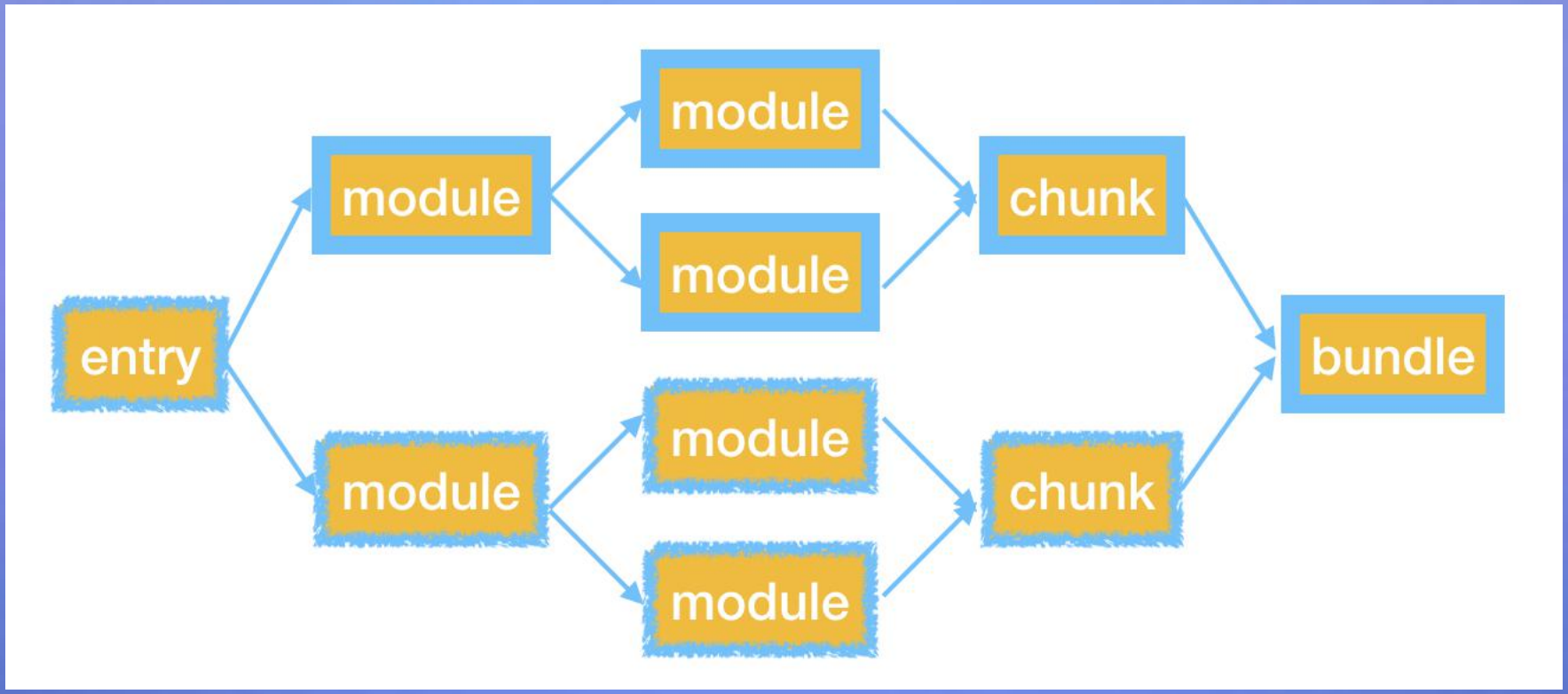
Module：模块，在 Webpack 里一切皆模块，一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。

Chunk：代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。

```
const path = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const {
  CleanWebpackPlugin
} = require('clean-webpack-plugin')

module.exports = {
  entry: {
    app: './src/index.js',
  },
  output: {
    filename: '[name].[hash].js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['css-loader', 'style-loader']
      },
    ]
  },
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'hello webpack'
    }),
  ]
}
```

三、 工作机制



webpack的过程是通过Compiler控制流程，Compilation专业解析，ModuleFactory生成模块，Parser解析源码，最后通过Template组合模块，输出打包文件的过程。

2

Webpack

核心概念

一、 Tapable基本概念

webpack 的插件架构主要基于 Tapable 实现的，Tapable 是 webpack 项目组的一个内部库，主要是抽象了一套插件机制。webpack 源代码中的一些 Tapable 实例都继承或混合了 Tapable 类。Tapable 能够让我们为 javaScript 模块添加并应用插件。它可以被其它模块继承或混合。它类似于 NodeJS 的 EventEmitter 类，专注于自定义事件的触发和操作。除此之外, Tapable 允许你通过回调函数的参数访问事件的生产者。
简单来说，就是我们熟悉的**发布-订阅模式**。

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();
//on的第一个参数是事件名，emit可以通过这个事件名触发这个方法
//on的第二个参数是回调函数，也就是此事件的执行方法
myEmitter.on('eventName', (param1, param2) => {
    console.log("eventName", param1, param2)
});
//emit的第一个参数是触发的事件名
//emit的第二个之后的参数是回调函数的参数。
myEmitter.emit('eventName', 111, 222);
```

```
// 广播出事件
// eventName 为事件名称，注意不要和现有的事件重名
// params 为附带的参数
compiler.apply('eventName', params);

// 监听名称为eventName的事件，当eventName事件发生时，函数就会被执行。
// 同时函数中的 params 参数为广播事件时附带的参数。
compiler.plugin('eventName',function(params) {
    console.log(params)
});
```

一、 Tapable的常用钩子

按被注册插件们的执行逻辑来分钩子：

1. 基本钩子：注册的插件顺序执行，如SyncHook、AsyncParallelHook、AsyncSeriesHook。

2.瀑布流钩子：前一个插件的返回值，是后一个插件的入参。如SyncWaterfallHook，AsyncSeriesWaterfallHook。

3.Bail钩子：Bail钩子是指一个插件返回非undefined的值，就不继续执行后续的插件。相同类型的事件满足其一条件则推出，如：SyncBailHook，AsyncSeriesBailHook

4.循环钩子：循环调用插件，直到插件的返回值是undefined。如SyncLoopHook。

按时序来区分钩子：

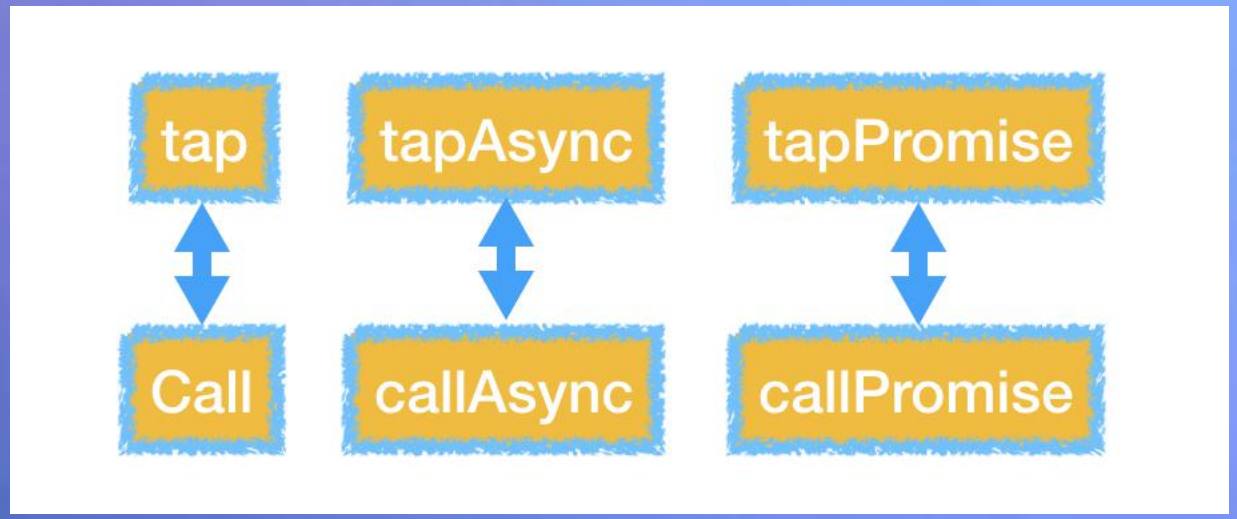
1.同步钩子：Sync开头的钩子。

2.异步串行钩子：AsyncSeries开头的钩子。

3.异步并行钩子：AsyncParallel开头的钩子。

钩子详解：
https://juejin.im/post/5d36faa9e51d45109725ff55#heading-18

一、 Tapable的使用




```
const { SyncHook } = require('tapable')
let h1 = new SyncHook(['options']);
h1.tap('A', function (arg) {
  console.log('A', arg);
  return 'b';
})
h1.tap('B', function () {
  console.log('b')
})
h1.intercept({
  call: (...args) => {
    console.log(...args, '-----intercept call');
  },
  register: (tap) => {
    console.log(tap, '-----intercept register');
    return tap;
  },
  loop: (...args) => {
    console.log(...args, '------intercept loop')
  },
  tap: (tap) => {
    console.log(tap, '------intercept tap')
  }
})
h1.call(666);
```

二、 Compiler

compiler 对象是 webpack 的编译器对象，compiler 对象会在启动 webpack 的时候被一次性的初始化，compiler 对象中包含了所有 webpack 可自定义操作的配置，例如 loader 的配置，plugin 的配置，entry 的配置等各种原始 webpack 配置等，在 webpack 插件中的自定义子编译流程中，我们肯定会用到 compiler 对象中的相关配置信息，我们相当于可以通过 compiler 对象拿到 webpack 的主环境所有的信息。

```
const Compiler = require("./Compiler")
const webpack = (options, callback) => {
    ...
    options = new WebpackOptionsDefaulter().process(options)
    let compiler = new Compiler(options.context)
    compiler.options = options
    new NodeEnvironmentPlugin().apply(compiler)
    for (const plugin of options.plugins) {
        plugin.apply(compiler);
    }
    ...
}
```

1.整合options，merge我们自定义的配置和Webpack默认的配置；

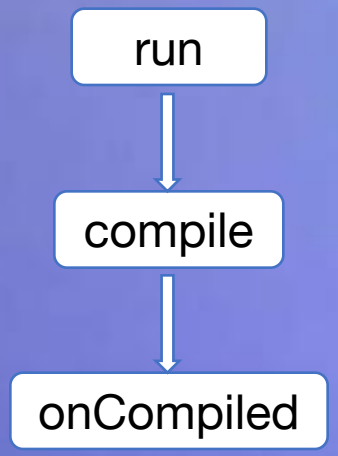
2.实例化compiler对象；

3.实例化所有的（内置的和我们配置的）Webpack插件。调用它们的apply方法（广播出去）；

4. 返回compiler对象；

webpack入口函数，主要就是以上读取配置，实例compiler，实例化和挂载插件。

二、 Compiler之核心




```
class Compiler extends Tapable {
  constructor(context) {
    this.hooks = {
      ...
      beforeRun,
      run
    }
  }
  ...
  run(callback) {
    ...
    this.hooks.beforeRun.callAsync(this, err => {
      this.hooks.run.callAsync(this, err => {
        ...
        this.compile(onCompiled)
      });
    });
  }
}
```

```
compile(callback) {
  this.hooks.beforeCompile.callAsync(params, err => {
    this.hooks.compile.call(params);
    const compilation = this.newCompilation(params);
    this.hooks.make.callAsync(compilation, err => {
      compilation.finish(err => {
        compilation.seal(err => {
          this.hooks.afterCompile.callAsync(compilation, err => {
            return callback(null, compilation);
          });
        });
      });
    });
  });
}
```

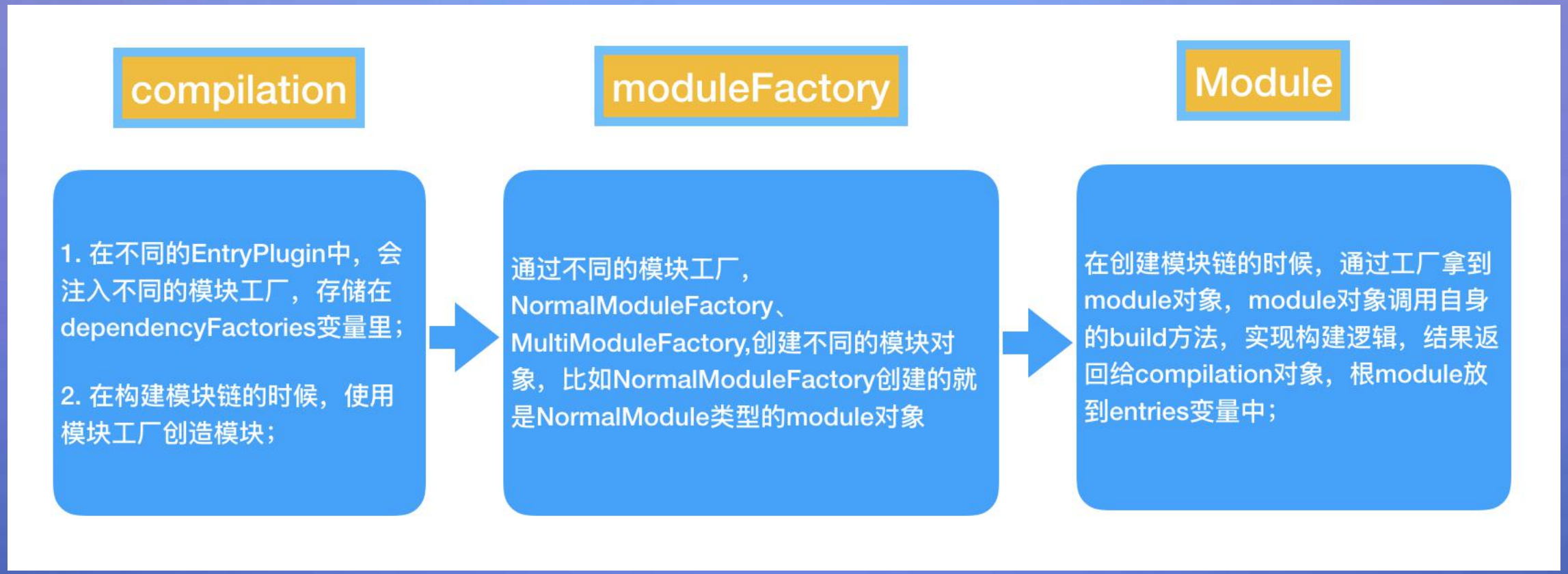
```
const onCompiled = (err, compilation) => {
    if (this.hooks.shouldEmit.call(compilation) === false) {
        ...
        this.hooks.done.callAsync(stats, err => {
            ...
        });
        return;
    }

    this.emitAssets(compilation, err => {
        this.hooks.done.callAsync(stats, err => {
            ...
        });
        return;
    });
};
```

```
createCompilation() {
  return new Compilation(this);
}

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

三、 Compilation的主要对象



构建模块和Chunk，并利用插件优化构建过程。

三、 Compilation的主要过程

addEntry

调用在_addModuleChain方法，调用完毕后执行addEntry的callback，通知make钩子的插件编译工作完成。

_addModuleChain

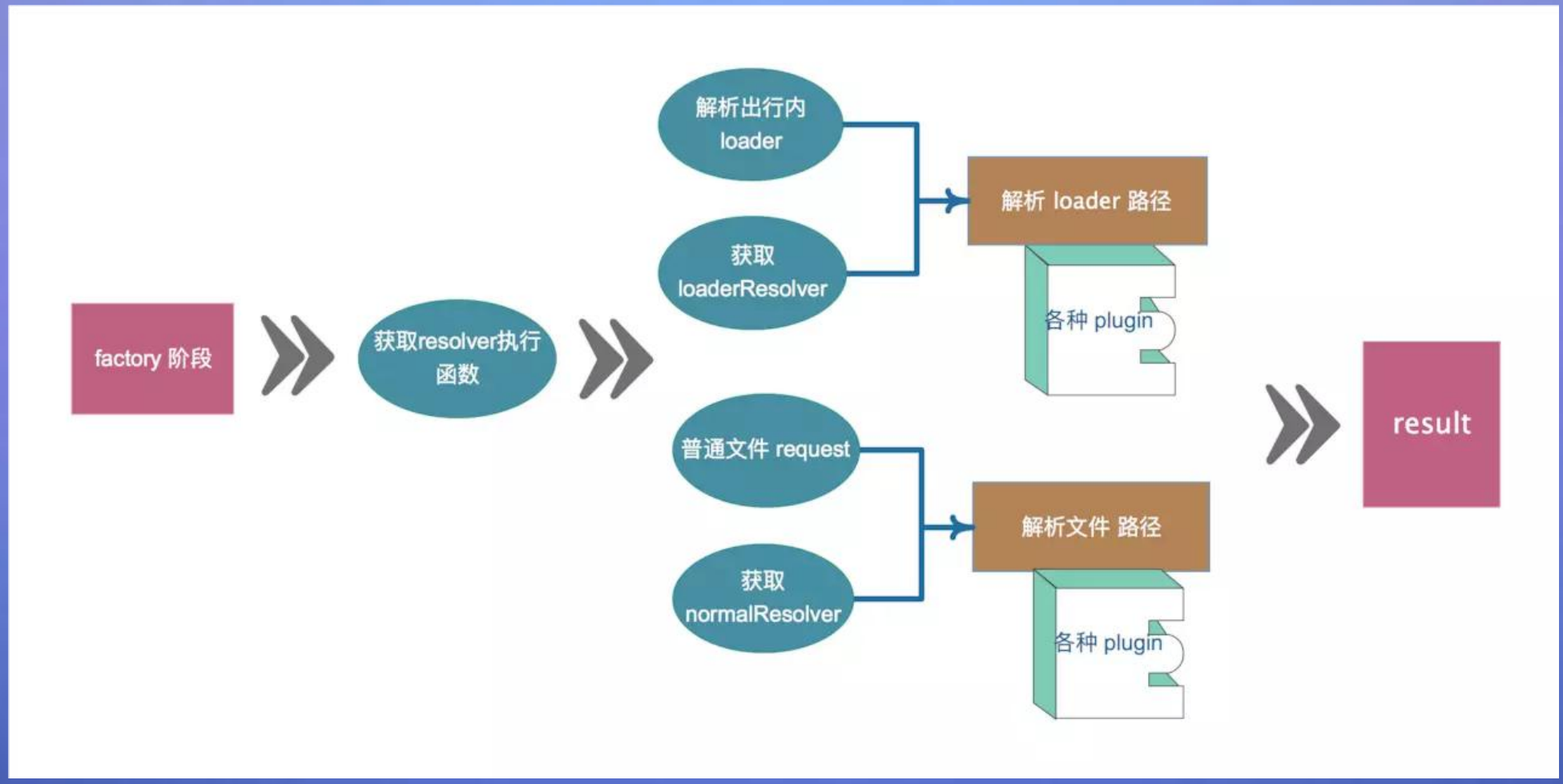
1. 添加模块链，当它执行完，模块都构建好了，并且形成了链。
2. 根据dep找到moduleFactory，我们找到的是normalModuleFactory。
3. 调用moduleFactory.create方法。在返回值中拿到module，这里的module是个NormalModule类型的对象。
4. 得到moudle对象后，调用buildModule方法。此方法内调用buildModule钩子，然后调用moudle自身的build方法。build方法就是调用loader，构建模块，获取依赖的逻辑。
5. 得到编译后的moudle，调用afterBuild方法。在afterBuild判断模块依赖了哪些模块，递归的用模块工厂创建它们，重复3，4，5流程。直到所有关联的模块构建完毕，我们就拿到模块链了。

从make钩子的插件开始，插件可以操作compilation，调用了compilation的addEntry。编译工作就从入口文件开始了，参数dep就是为了在compilation.dependencyFactories找到normalModuleFactory。

三、 Compilation的主要过程

addEntry执行结束、SingleEntryPlugin执行结束、make钩子调用结束。该执行make钩子的回调函数了。make钩子回调中调用了compilation的seal方法。开始了Chunk的构建和打包的优化过程。

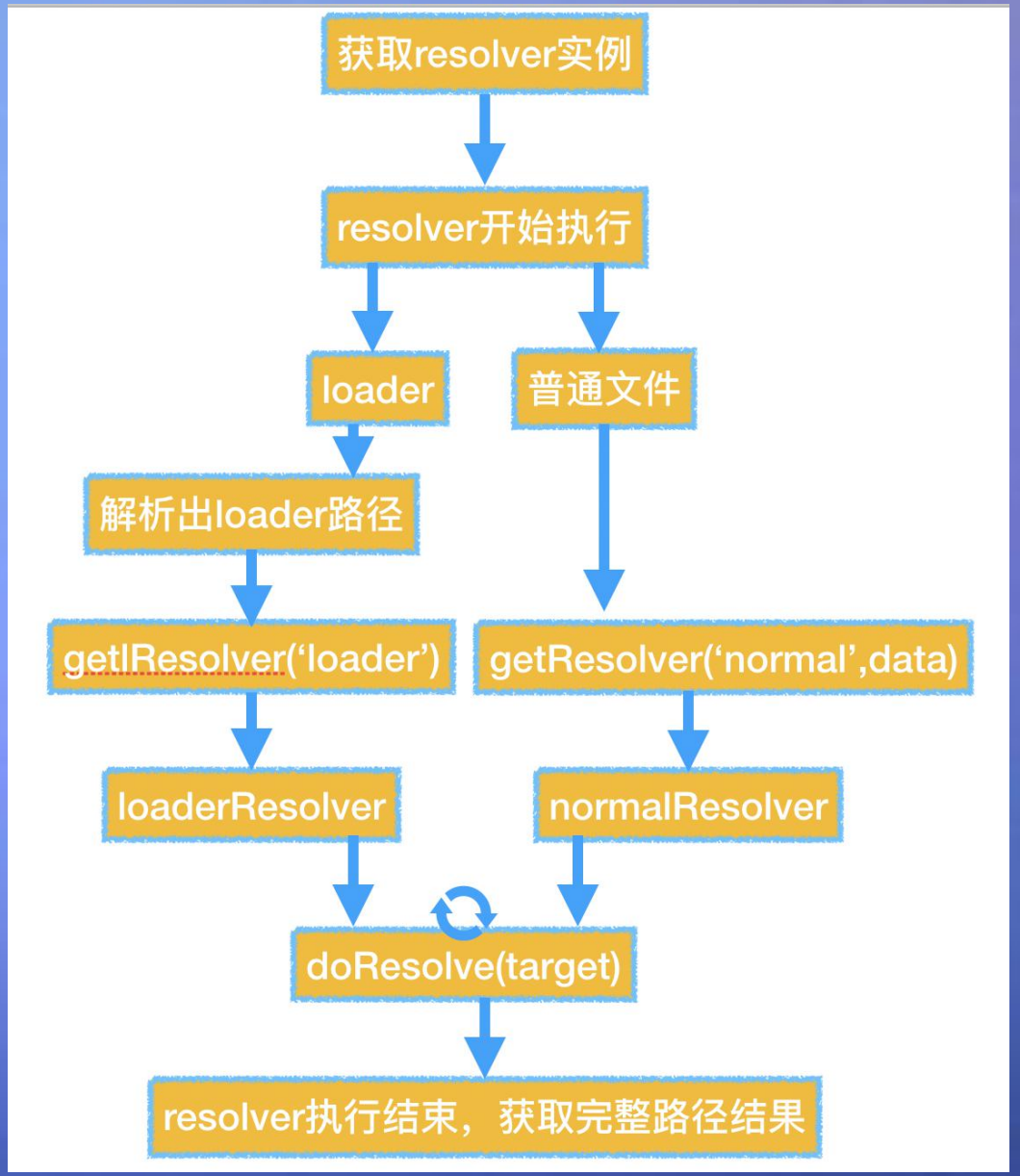
四、 Resolve整体流程



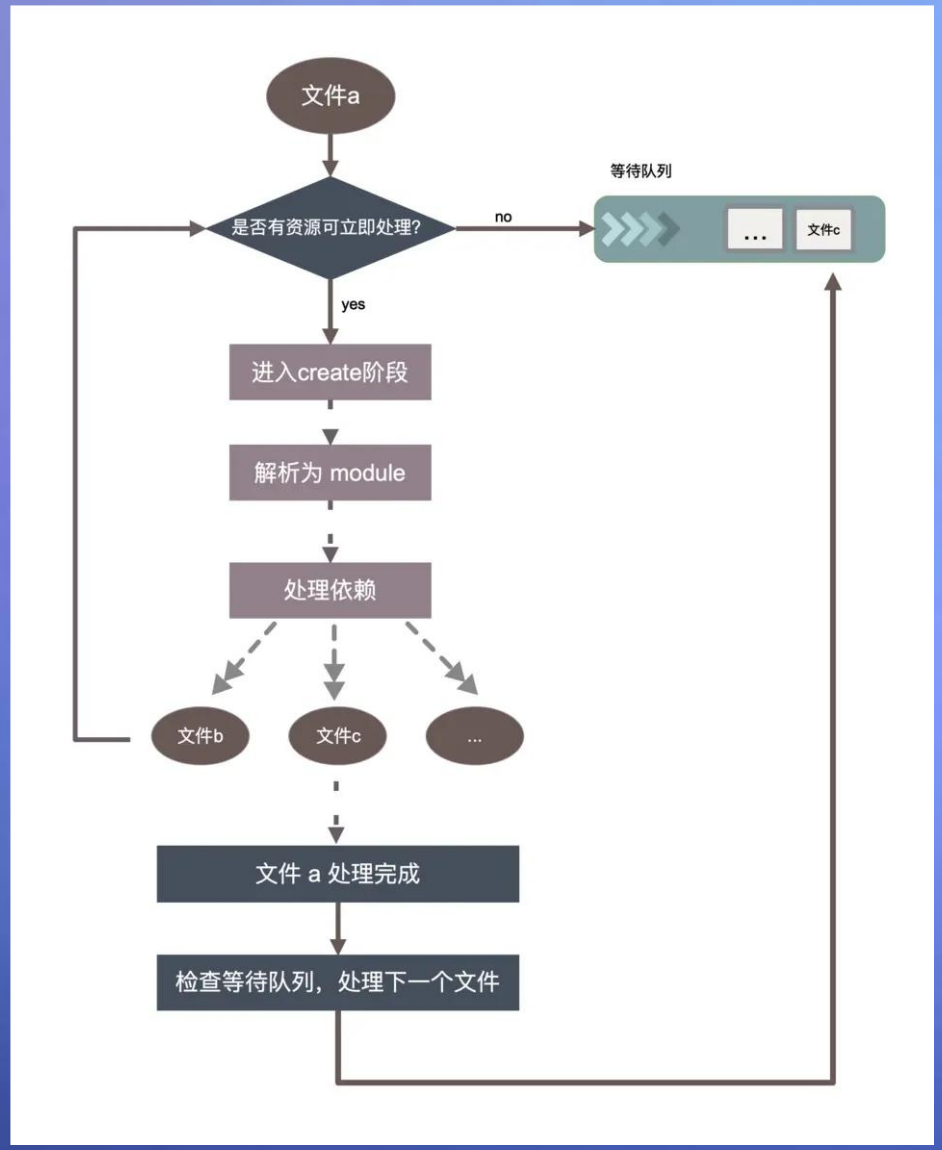
(图片来源于 https://juejin.im/post/5c6b78cdf265da2da15db125)

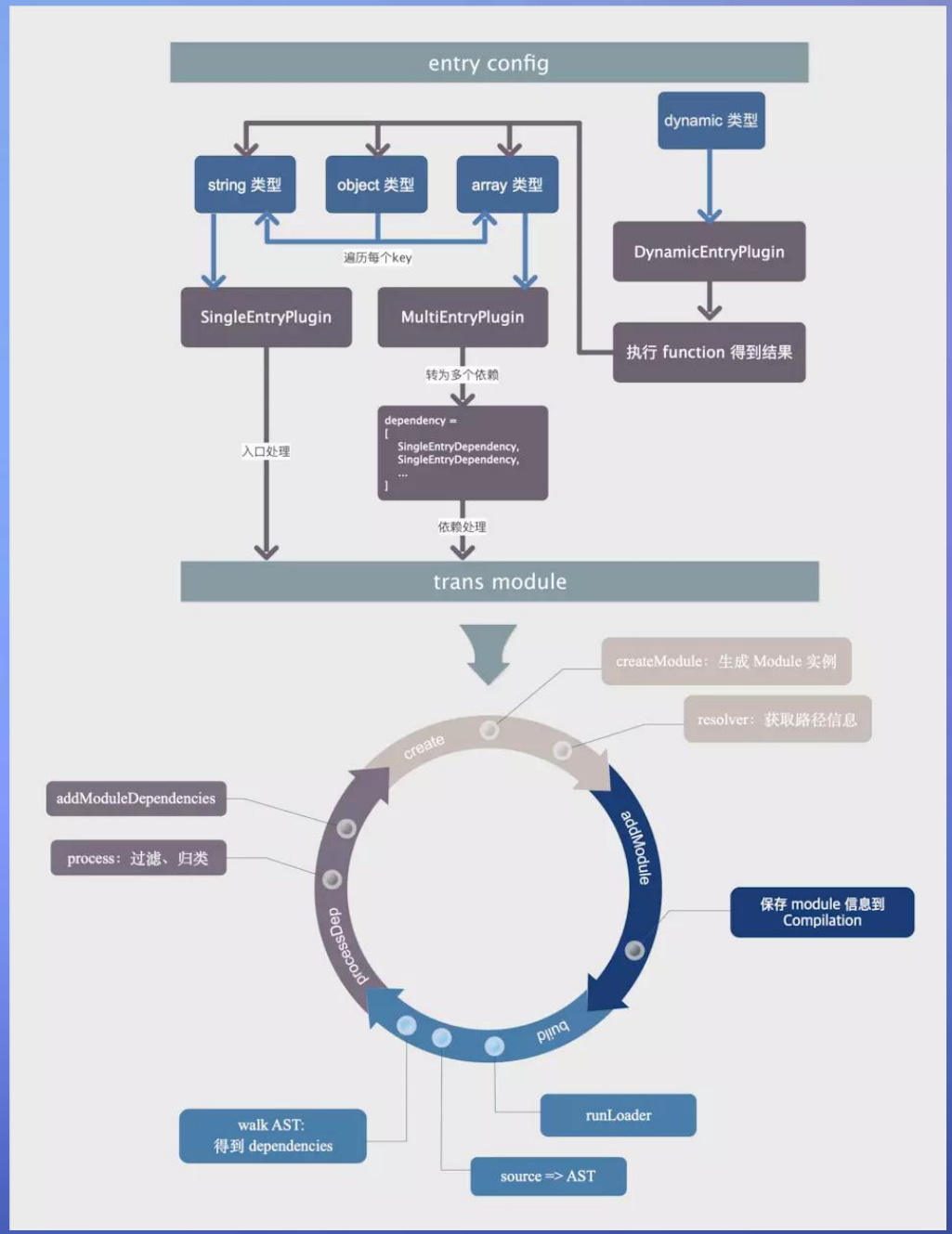
四、 Resolve具体流程

```
class NormalModuleFactory extends Tapable {
    constructor(context, resolverFactory, options) {
        this.hooks = {
            resolver: new SyncWaterfallHook(["resolver"]),
            factory: new SyncWaterfallHook(["factory"]),
            beforeResolve: new AsyncSeriesWaterfallHook(["data"]),
            afterResolve: new AsyncSeriesWaterfallHook(["data"]),
            createModule: new SyncBailHook(["data"]),
            module: new SyncWaterfallHook(["module", "data"]),
            createParser: new HookMap(() => new SyncBailHook(["parserOptions"])),
            parser: new HookMap(() => new SyncHook(["parser", "parserOptions"])),
            createGenerator: new HookMap(
                () => new SyncBailHook(["generatorOptions"])
            ),
            generator: new HookMap(
                () => new SyncHook(["generator", "generatorOptions"])
            )
        };
    }
}
```



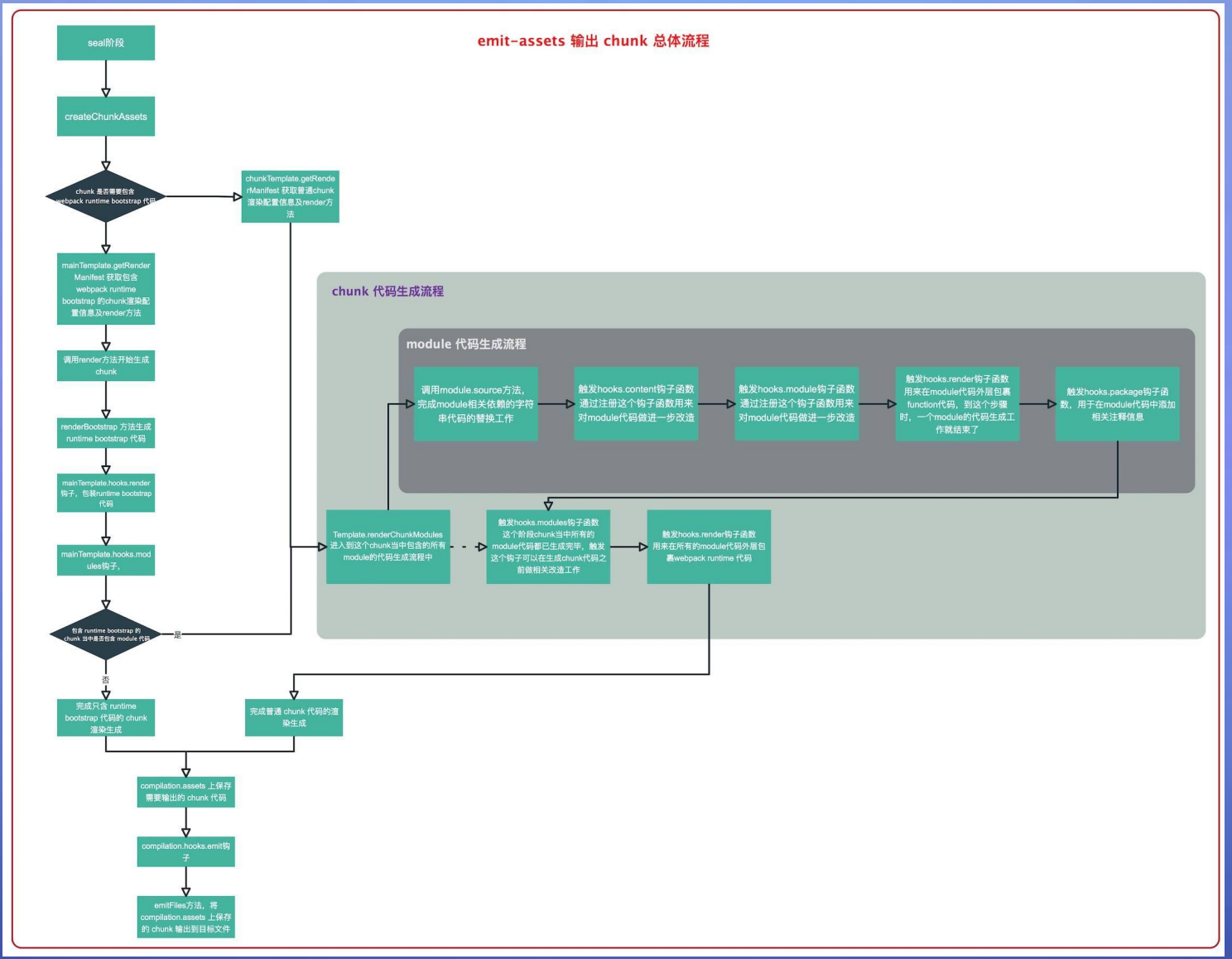
五、 Module生成





（图片来源于 https://juejin.im/post/5cc51b79518825250c76aac0）

五、Chunk生成

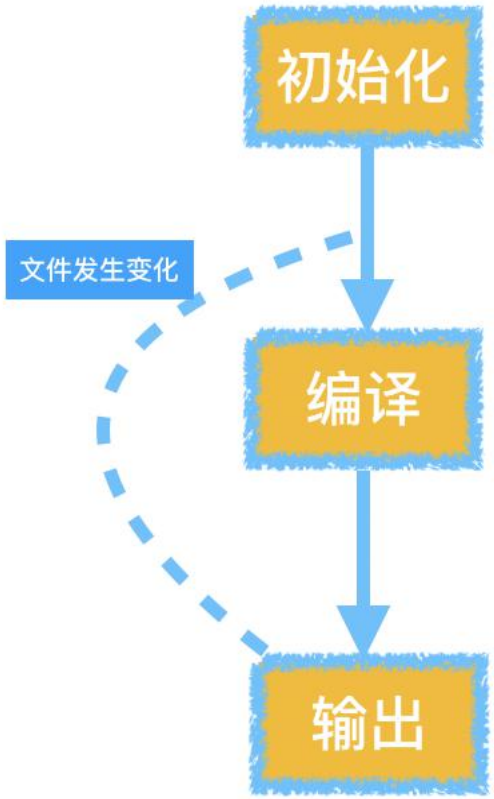


(图片来源于 https://juejin.im/post/5cf4a53ae51d45775b419b7e)

3

Webpack
打包原理

一、 Webpack打包流程



启动构建，读取与合并配置参数，加载 Plugin，实例化 Compiler。

从 Entry 发出，针对每个 Module 串行调用对应的 Loader 去翻译文件内容，再找到该 Module 依赖的 Module，递归地进行编译处理。

对编译后的 Module 组合成 Chunk，把 Chunk 转换成文件，输出到文件系统。

二、 初始化阶段

事件名称	解释
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。 这个过程中还会执行配置文件中的插件实例化语句 new Plugin()。
实例化 Compiler	用上一步得到的参数初始化 Compiler 实例，Compiler 负责文件监听和启动编译。Compiler 实例中包含了完整的 Webpack 配置，全局只有一个 Compiler 实例。
加载插件	依次调用插件的 apply 方法，让插件可以监听后续的所有事件节点。同时给插件传入 compiler 实例的引用，以方便插件通过 compiler 调用 Webpack 提供的 API。
environment	开始应用 Node.js 风格的文件系统到 compiler 对象，以方便后续的文件寻找和读取。
entry-option	读取配置的 Entrys，为每个 Entry 实例化一个对应的 EntryPlugin，为后面该 Entry 的递归解析工作做准备。
after-plugins	调用完所有内置的和配置的插件的 apply 方法。
after-resolvers	根据配置初始化完 resolver，resolver 负责在文件系统中寻找指定路径的文件。

三、 编译阶段

事件名称	解释
run	启动一次新的编译。
watch-run	和 run 类似，区别在于它是在监听模式下启动的编译，在这个事件中可以获取到是哪些文件发生了变化导致重新启动一次新的编译。
compile	该事件是为了告诉插件一次新的编译将要启动，同时会给插件带上 compiler 对象。
compilation	当 Webpack 以开发模式运行时，每当检测到文件变化，一次新的 Compilation 将被创建。一个 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。Compilation 对象也提供了很多事件回调供插件做扩展。
make	一个新的 Compilation 创建完毕，即将从 Entry 开始读取文件，根据文件类型和配置的 Loader 对文件进行编译，编译完后再找出该文件依赖的文件，递归的编译和解析。
after-compile	一次 Compilation 执行完成。
invalid	当遇到文件不存在、文件编译错误等异常时会触发该事件，该事件不会导致 Webpack 退出。

四、 编译阶段之compliation

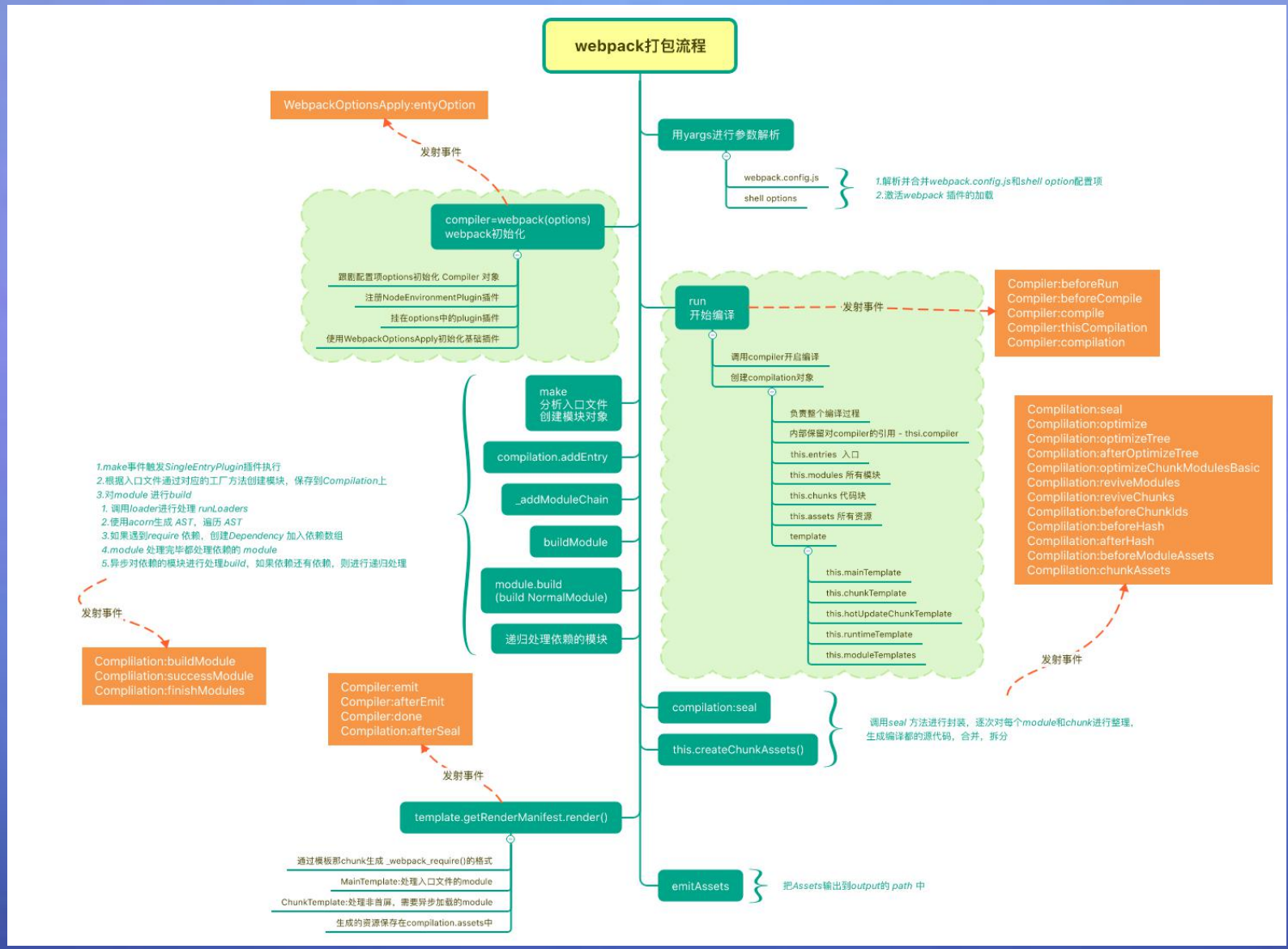
事件名称	解释
build-module	使用对应的 Loader 去转换一个模块。
normal-module-loader	在用 Loader 对一个模块转换完后，使用 acorn 解析转换后的内容，输出对应的抽象语法树（AST），以方便 Webpack 后面对代码的分析。
program	从配置的入口模块开始，分析其 AST，当遇到 require 等导入其它模块语句时，便将其加入到依赖的模块列表，同时对新找出的依赖模块递归分析，最终搞清所有模块的依赖关系。
seal	所有模块及其依赖的模块都通过 Loader 转换完成后，根据依赖关系开始生成 Chunk。

五、 输出阶段

事件名称	解释
should-emit	所有需要输出的文件已经生成好，询问插件哪些文件需要输出，哪些不需要。
emit	确定好要输出哪些文件后，执行文件输出，可以在这里获取和修改输出内容。
after-emit	文件输出完毕。
done	成功完成一次完成的编译和输出流程。
failed	如果在编译和输出流程中遇到异常导致 Webpack 退出时，就会直接跳转到本步骤，插件可以在本事件中获取到具体的错误原因。

在输出阶段已经得到了各个模块经过转换后的结果和其依赖关系，并且把相关模块组合在一起形成一个个 Chunk。 在输出阶段会根据 Chunk 的类型，使用对应的模版生成最终要要输出的文件内容。

六、 完整流程图



（图片来源于 https://www.goyth.com/2018/12/10/webpackFlow）

4

Webpack 优化实践

优化开发体验

1. 缩小文件搜索范围
2. 使用DllPlugin
3. 使用HappyPack
4. 使用ParallelUglifyPlugin
5. 使用自动刷新
6. 开启模块热替换

优化输出质量

1. 压缩代码
2. CDN加速
3. 抽取公共代码
4. 使用Tree Shaking
5. 使用Code Spliting
6. 开启Scope Hoisting

5 详解 MiniCssExtractPlugin

https://github.com/webpack-contrib/mini-css-extract-plugin

一、 功能特点

将CSS提取为独立文件的插件，对每个包含css的js文件都会创建一个同名CSS文件，不会再内联到js文件中了，支持按需加载css和sourceMap；

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
module.exports = {
  plugins: [
    new MiniCssExtractPlugin({
      filename: '[name].css',
      chunkFilename: '[id].css',
      ignoreOrder: false,
    }),
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
            options: {
              publicPath: '../',
              hmr: process.env.NODE_ENV === 'development',
            },
          },
          'css-loader',
        ],
      },
    ],
  },
};
```

二、对比ExtractTextWebpackPlugin

ExtractTextWebpackPlugin

优点	缺点
更少的style标签	额外的 HTTP 请求
CSS SourceMap (使用 devtool: "source-map" 和 css-loader?sourceMap 配置)	更长的编译时间
CSS 请求并行	不支持HMR，只能用在production环境中
CSS 单独缓存	只支持webpack4以下
更快的浏览器运行时 (更少代码和 DOM 的运行)	没有运行时的公共路径修改

MiniCssExtractPlugin

优点	缺点
异步加载	支持HMR，最好用在production环境中
不重复编译，性能更好	和style-loader不能在一个Loader链上共存
更容易使用	额外的 HTTP 请求
只针对CSS，且不重复	只支持webpack4+
支持publicPath配置	进阶配置稍复杂

如果是老配置升级webpack4的可以使用beta版的extract-text-webpack-plugin@next;

三、原理解析

```
class SplitChunksPlugin {
  constructor(options) {
    this.options = SplitChunksPlugin.normalizeOptions(options);
  }
  static normalizeOptions(options = {}) {...}
  static normalizeName() {}
  ...
  apply(compiler) {
    compiler.hooks.thisCompilation.tap("SplitChunksPlugin", compilation => {
      let alreadyOptimized = false;
      compilation.hooks.unseal.tap("SplitChunksPlugin", () => {
        alreadyOptimized = false;
      });
      compilation.hooks.optimizeChunksAdvanced.tap("SplitChunksPlugin", chunks => {
        const indexMap = new Map();
        ...
        const chunkSetsByCount = new Map();
        ...
        const combinationsCache = new Map();
        ...
        const selectedChunksCacheByChunksSet = new WeakMap();
        ...
        const incorrectMinMaxSizeSet = new Set();
        ...
        const chunksInfoMap = new Map();
      })
    }
  }
}
```

三、原理解析

```
export function pitch(request) {
  const options = loaderUtils.getOptions(this) || {};
  validateOptions(schema, options, 'Mini CSS Extract Plugin Loader');
  const loaders = this.loaders.slice(this.loaderIndex + 1);
  this.addDependency(this.resourcePath);
  const childCompiler = this._compilation.createChildCompiler(
    `${pluginName} ${request}`,
    outputOptions
  );
  childCompiler.hooks.thisCompilation.tap(`${pluginName} loader`, (compilation) => {})
  childCompiler.hooks.afterCompile.tap(pluginName, (compilation) => {})
  childCompiler.runAsChild((err, entries, compilation) => {})
}
```

6 详解 SplitChunksPlugin

一、 功能特点

最初，chunks(代码块)和导入他们中的模块通过webpack内部的父子关系图连接.在webpack3中，通过CommonsChunkPlugin来避免他们之间的依赖重复。但是无法进一步优化，webpack4中直接取而代之，使用optimization.splitChunks 和 optimization.runtimeChunk 配置项。

在默认情况下，SplitChunksPlugin 仅仅影响按需加载的代码块，因为更改初始块会影响HTML文件应包含的脚本标记以运行项目。

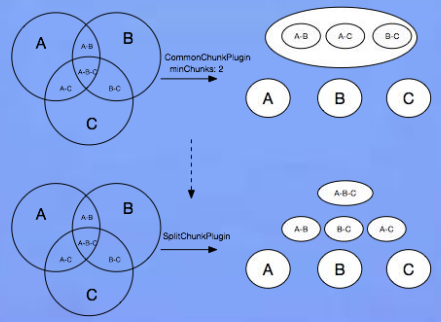
webpack将根据以下条件自动拆分代码块：

- 会被共享的代码块或者 node_mudules 文件夹中的代码块
- 体积大于30KB的代码块（在gz压缩前）
- 按需加载代码块时的并行请求数量不超过5个
- 加载初始页面时的并行请求数量不超过3个

```
module.exports = {
  //...
  optimization: {
    splitChunks: {
      chunks: 'async',
      minSize: 30000,
      minRemainingSize: 0,
      maxSize: 0,
      minChunks: 1,
      maxAsyncRequests: 6,
      maxInitialRequests: 4,
      automaticNameDelimiter: '~',
      automaticNameMaxLength: 30,
      cacheGroups: {
        vendors: {
          test: /[\\/]node_modules[\\/]/,
          priority: -10
        },
        default: {
          minChunks: 2,
          priority: -20,
          reuseExistingChunk: true
        }
      }
    }
  }
};
```

二、对比CommonsChunkPlugin

CommonsChunkPlugin



SplitChunksPlugin

优点	缺点
打开页面速度提升	首页文件体积增大
避免代码重复加载	公共chunk过大
配置简单易用	只支持webpack4以下

优点	缺点
打开页面速度提升	进阶后配置稍复杂
默认情况下只对按需异步块有影响，开箱即用	更多的chunkGroup
自动化的解决懒加载模块之间的代码重复问题	只支持webpack4+

三、原理解析

```
class SplitChunksPlugin {
  constructor(options) {
    this.options = SplitChunksPlugin.normalizeOptions(options);
  }
  static normalizeOptions(options = {}) {...}
  static normalizeName() {}
  ...
  apply(compiler) {
    compiler.hooks.thisCompilation.tap("SplitChunksPlugin", compilation => {
        let alreadyOptimized = false;
        compilation.hooks.unseal.tap("SplitChunksPlugin", () => {
            alreadyOptimized = false;
        });
        compilation.hooks.optimizeChunksAdvanced.tap("SplitChunksPlugin", chunks => {
            const indexMap = new Map();
            ...
            const chunkSetsByCount = new Map();
            ...
            const combinationsCache = new Map();
            ...
            const selectedChunksCacheByChunksSet = new WeakMap();
            ...
            const incorrectMinMaxSizeSet = new Set();
            ...
            const chunksInfoMap = new Map();
        })
    })
  }
}
```


加餐

开发Webpack之plugin或loader常用的方法：

```
// 判断当前配置使用使用了 ExtractTextPlugin,
// compiler 参数即为 Webpack 在 apply(compiler) 中传入的参数
function hasExtractTextPlugin(compiler) {
  // 当前配置所有使用的插件列表
  const plugins = compiler.options.plugins;
  // 去 plugins 中寻找有没有 ExtractTextPlugin 的实例
  return plugins.find(plugin=>plugin.__proto__.constructor === ExtractTextPlugin) != null;
}
```

```
module.exports = function(source) {
    // 关闭该 Loader 的缓存功能
    this.cacheable(false);
    // 告诉 Webpack 本次转换是异步的，Loader 会在 callback 中回调结果
    var callback = this.async();
    doSomeAsyncOperation(source, function(err, result, sourceMaps, ast) {
        // 通过 callback 返回异步执行后的结果
        if(err) return callback(err)
        callback(null, result, sourceMaps, ast);
    });
    return;
};
```

```
module.exports = function(source) {
  // 在 exports.raw === true 时，
  // Webpack 传给 Loader 的 source 是 Buffer 类型的；
  // 在 exports.raw !== true 时，
  // Loader 也可以返回 Buffer 类型的结果；
  source instanceof Buffer === true;
  return source;
};
// 通过exports.raw属性告诉Webpack该Loader是否需要二进制数据
module.exports.raw = true;
```

参考资料:

1. https://www.webpackjs.com/concepts/ -- Webpack官方文档

2. https://juejin.im/post/5d176df3518825032216 93dc -- Lewis的掘金专栏

3. https://juejin.im/post/5d36faa9e51d45109725ff55 -- tapable 使用研究

Q&A