- 01 | loader的链式调用与执行顺序（从右到左）
  - 一个最简单的loader 代码结构
  - 多 loader 时的执行顺序
  - 函数组合的两种情况
  - 通过一个列子验证一下 loader 的执行顺序
- 02 | 使用loader-runner 高效进行loader的调试
  - loader-runner 介绍
  - loader-runner 的使用
  - 开发一个 raw-loader
  - 使用 loader-runner 调试 loader
- 03 | 更复杂的loader的开发场景
  - loader 的参数获取
  - loader 异常处理(同步loader中)
  - laoder 异步处理
  - 在 loader 中使用缓存
  - loader 如何惊醒文件输出?
- 04 | 实战开发一个自动合成雪碧图的loader
  - 支持的语法
  - 准备知识：如何将两张图片合成一张图片?
- 05 | 插件基本结构介绍
  - 插件的运行环境
  - 插件的基本结构
  - 搭建插件的运行环境
  - 开发一个最简单的插件
- 06 | 更复杂的插件开发场景
  - 插件中如何获取传递的参数?
  - 插件的错误处理
  - 通过 compilation 对象进行文件写入
  - 插件拓展：编写插件的插件
- 07 | 实战开发一个压缩构建资源为zip包的插件
  - 要求
  - 准备只是：Node.js 里面将文件压缩为.zip包
  - compiler 上负责文件生成的 hooks
  - 示例代码
- 08| 创新与自驱动的产出

01 | loader的链式调用与执行顺序（从右到左）

一个最简单的loader 代码结构

- 定义：loader 只是一个导出为函数的JS模块

```
module.exports = function (source) {
    // ...
```

```
    return source
}
```

多 loader 时的执行顺序

- 多个 loader 串行执行 series
- 顺序从后到前

```
module.exports = {
    // ...
    module: {
        rules: [
            {
                test: /\.less$/,
                use: [
                    'style-loader', // 3
                    'css-loader',   // 2
                    'less-loader'   // 1 执行顺序
                ]
            }
        ]
    }
}
```

函数组合的两种情况

- Unix 中的 pipline（从左往右）
- Compose（webpack 采取的是这种）从右往左

```
compose = (f, g) => (...args) => f(g(...args))
```

通过一个列子验证一下 loader 的执行顺序

> a-loader.js

```
module.exports = function(source) {
    console.log('loader a is executed')
    return source
}
```

> b-loader.js

```
module.exports = function(source) {
    console.log('loader b is executed')
    return source
}
```

02 | 使用loader-runner 高效进行loader的调试

loader-runner 介绍

- 定义： loader-runner 允许你在不安装 webpack 的情况下运行 loaders
- 作用：
    - 作为 webpack 的依赖，webpack 中使用它执行 loader
    - 进行loader 的开发和调试

loader-runner 的使用

```
import { runLoaders } from "loader-runner";

runLoaders({
    resource: "/abs/path/to/file.txt?query",
    // String: Absolute path to the resource (optionally including query
string)

    loaders: ["/abs/path/to/loader.js?query"],
    // String[]: Absolute paths to the loaders (optionally including query
string)
    // {loader, options}[]: Absolute paths to the loaders with options
object

    context: { minimize: true },
    // Additional loader context which is used as base context

    processResource: (loaderContext, resourcePath, callback) => { ... },
    // Optional: A function to process the resource
    // Must have signature function(context, path, function(err, buffer))
    // By default readResource is used and the resource is added a
fileDependency

    readResource: fs.readFile.bind(fs)
    // Optional: A function to read the resource
    // Only used when 'processResource' is not provided
    // Must have signature function(path, function(err, buffer))
    // By default fs.readFile is used
}, function(err, result) {
    // err: Error?

    // result.result: Buffer | String
    // The result
```

```
    // only available when no error occured

    // result.resourceBuffer: Buffer
    // The raw resource as Buffer (useful for SourceMaps)
    // only available when no error occured

    // result.cacheable: Bool
    // Is the result cacheable or do it require reexecution?

    // result.fileDependencies: String[]
    // An array of paths (existing files) on which the result depends on

    // result.missingDependencies: String[]
    // An array of paths (not existing files) on which the result depends
on

    // result.contextDependencies: String[]
    // An array of paths (directories) on which the result depends on
})
```

开发一个 raw-loader

- src/raw-loader.js

```
module.exports = function(source) {
    const json = JSON.stringfy(source)
        .replace(/\u2028/g,'\\u2028 ')
        .replace(/\u2029/g,'\\u2029 ')

    return `export default ${json}`
}
```

- src/demo.text
    - foobar

使用 loader-runner 调试 loader

- run-loader.js

```
const fs  = require("fs");
const path  = require("path");
const { runLoaders }  = require("loader-runner");

runLoaders({
    resource: path.join(__dirname, './src/demo.text'),
    loaders: [
        path.resolve(__dirname, './loader/raw-loader.js')
    ],
    readResource: fs.readFile.bind(fs)
```

```
}, (err, result) => {
    err ? console.error(err) : console.log(result)
})
```

- 运行查看结果：node run-loader.js

03 | 更复杂的loader的开发场景

loader 的参数获取

- 通过 loader-utils 的 getOptions 方法获取

```
const loaderUtils = require('loader-utils')

module.exports = function(content) {
    const { name } = loaderUtils.getOptions(this)
}
```

loader 异常处理(同步loader中)

- loader 内直接通过 throw 抛出
- 通过 this.callback 传递错误

```
this.callback(
    err: Error | null,
    content: string | Buffer,
    sourceMap?:SourceMap,
    meta?:any
)
```

laoder 异步处理

- 通过 this.async 来返回一个异步函数
    - 第一个参数是 Error，第二个参数是处理的结果
- 示例代码

```
module.exports = function(source) {
    this.callback = this.async()
    // No callback -> return synchronous results
    // if (callback) { ... }

    callback(null, input + input)
}
```

在 loader 中使用缓存

- webpack 中默认开启loader缓存
  - 可以使用 this.cacheable(false) 关掉缓存
- 缓存条件：loader 的结果在相同的输入下有确定的输出
  - 有依赖的 loader 无法使用缓存

loader 如何惊醒文件输出?

- 通过 this.emitFile 进行文件写入

```

```

04 | 实战开发一个自动合成雪碧图的loader

支持的语法

```
    background: url('a.png?__sprite');
                                        =>  background: url('sprite.png');
    background: url('b.png?__sprite');
```

准备知识：如何将两张图片合成一张图片?

- 使用 spritesmith
- 示例代码

```

```

05 | 插件基本结构介绍

插件的运行环境

- 插件没有像 loader 那样的独立运行环境（loader 不能做的 plugin 都可以使用）
- 只能在webpack 里面运行

插件的基本结构

- 基本结构

```
class TempWebpackPlugin {                                   // 1. 插件类名
    apply(compiler) {                                       // 2. 插件上的
```

```
apply 方法(必须)
        compiler.hooks.done.tap('Temp webpack plugin', (    // 3. 插件上的
hooks 监听
            stats
            /* stats is passed as arguments when done hook is tapped */
        ) => {
            console.log('Hello temp webpack plugin')        // 4. 插件的逻辑
处理
        })
    }
}
module.exports = TempWebpackPlugin
```

- 插件使用: plugins: [new MyPlugin()]

搭建插件的运行环境

```
const MyPlugin = require('./plugins/temp-webpack-plugin.js')

module.exports = {
    // ...
    plugins: [
        new MyPlugin()
    ]
}
```

开发一个最简单的插件

- src/plugins/demo-plugin.js

```
class DemoPlugin {
    constructor(options) {
        this.options = options
    }
    apply(compiler) {
        console.log('executed apply function :', compiler)
    }
}

module.exports = DemoPlugin
```

- 加入到webpack配置中

```
const MyPlugin = require('./plugins/temp-webpack-plugin.js')

module.exports = {
    // ...
```

```
    plugins: [
        new MyPlugin()
    ]
}
```

06 | 更复杂的插件开发场景

插件中如何获取传递的参数?

- 通过插件的构造函数进行获取

```
class DemoPlugin {
    constructor(options) {
        this.options = options
    }
    apply(compiler) {
        console.log('Demo plugin options :', this.options)
    }
}

module.exports = DemoPlugin
```

插件的错误处理

- 参数校验阶段可以直接throw的方式抛出：throw new Error('Error Message')
- 通过 compilation 对象的 warnings 和 errors 接收

```
compilation.warnings.push('Warning')
compilation.errors.push('Error')
```

通过 compilation 对象进行文件写入

- compilation 上的 assets 可以用于文件写入
  - 可以将 zip 资源包设置到 compilation.assets 对象上
  - 文件写入需要使用 webpack-source

```
const { RawSource } = require('webpack-source')

class DemoPlugin {
  constructor(options) {
      this.options = options
  }
  apply(compiler) {
      const { name } = this.options
```

```
        compiler.hooks.emit.tap('emit', (compilation, cb) => {
            compilation.assets[name] = new RawSource('demo')
            cb()
        })
    }
}

module.exports = DemoPlugin
```

插件拓展：编写插件的插件

- 插件自身也可以通过暴露hooks的方式进行自身拓展，以 html-webpack-plugin 为例
  - html-webpack-plugin-after-chunks（sync）
  - html-webpack-plugin-before-html-generation（Async）
  - html-webpack-plugin-after-asset-tag（Async）
  - html-webpack-plugin-after-html-processing（Async）
  - html-webpack-plugin-after-emit（Async）

07 | 实战开发一个压缩构建资源为zip包的插件

要求

- 生成的 zip 包文件名称可以通过插件传入
- 需要使用 compiler 对象上的 特定 hooks 进行资源的生成

准备只是：Node.js 里面将文件压缩为.zip包

- 使用 jszip
- 示例代码

```
var zip = new JSZip();
zip.file("Hello.txt", "Hello World\n");

var img = zip.folder("images");
img.file("smile.gif", imgData, {base64: true});

zip.generateAsync({type:"blob"}).then(function(content) {
    // see FileSaver.js
    saveAs(content, "example.zip");
});
```

compiler 上负责文件生成的 hooks

- Hooks 是 emit，是一个异步 hook（AsyncSeriesHook）
- emit 文件生成阶段，读取的是 compilation.assets 对象的值
  - 可以将 zip 资源包 设置到 compilation.assets 对象上

示例代码

```
const JSZip = require('jszip')
const path = require('path')
const { RawSource } = require('webpack-sources')

const zip = new JSZip()

class ZipPlugin {
    constructor(options) {
        this.options = options
    }
    apply(compiler) {
        compiler.hooks.emit.tapAsync('ZipPlugin', (compilation, callback)
=> {
            const { filename } = this.options
            const { assets } = compilation
            const folder = zip.folder(filename) // 使用 jszip 实例 zip 创建一
个目录

            for (const filename in assets) { // 将assets上的资源遍历写入
folder目录
                const source = assets[filename].source() // 得到源码
                folder.file(filename, source) // 往目录写入文件
            }

            zip.generateAsync({
                type:"nodebuffer" // 指定 content 数据格式
            }).then((content) => {
                const outputPath = path.join(
compilation.options.output.path, `${filename}.zip` ) // 压缩包输出的 绝对路径
                const outputRelativePath = path.relative(
compilation.options.output.path, outputPath) // 压缩包输出的 相对路径（相对
dist）
                compilation.assets[outputRelativePath] = new
RawSource(content) // 将压缩好的资源添加到 assets中（注意是相对路径）
                callback()
            })
        })
    }
}

module.exports = ZipPlugin
```

08| 创新与自驱动的产出

- 1）准备好一些好的xx-loader 或者 xx库（比较适合中台类型的项目）
- 2）准备一些runtime的插件，比如做一个UBB的js-parser，还能兼容时髦的 rn / flutter / 小程序 等

```
wordpress --> blog语言
    [a]
        [span]
            [fontSize][/fontSize]
        [/span]

    [/a]
```

- 3）多学一些非js的东西，比如 shell / php 等(最好能有)
- 4）做一些中间件，比如登陆业务 / 内部的业务RPC调用或特殊格式解析 cronttab 定期发一些消息