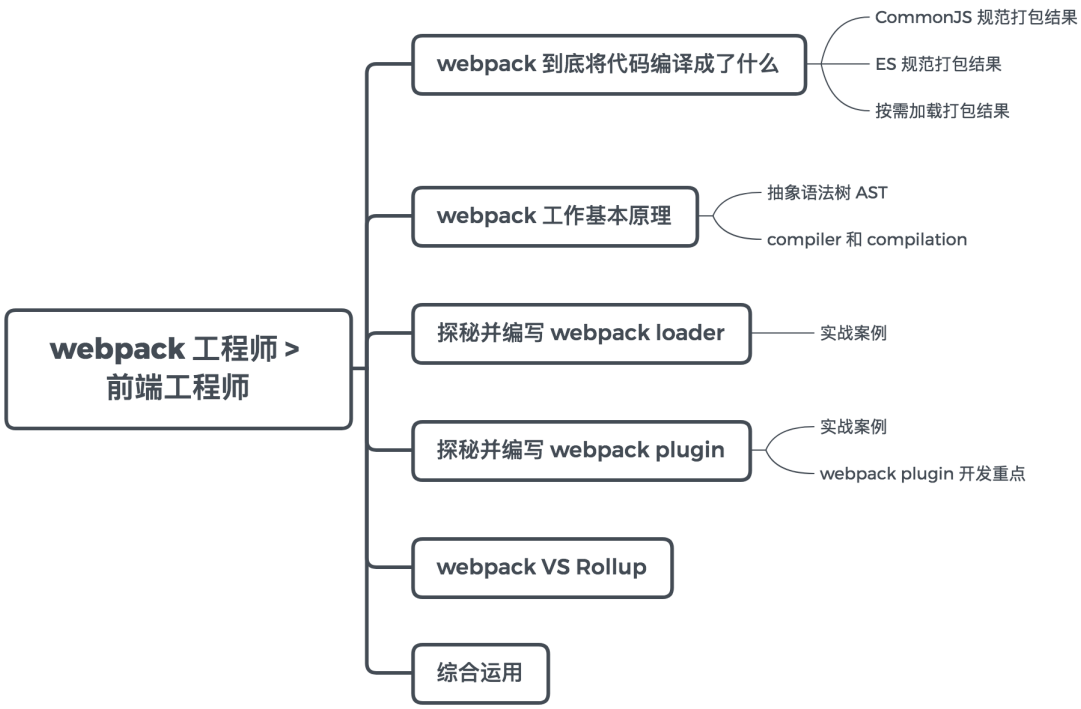
webpack 工程师 > 前端工程师（下）

上一节中，我们了解了 webpack 对于不同模块化标准的打包结果，分析了其自身的模块化解决方案。但是 webpack 绝不仅仅是一个打包器，它是一个完整的构建工具链。**那么它到底是如何工作的，原理是什么？了解了这些原理，我们又能如何扩展，以解决工作中的实际问题？**这一节，我们来一探究竟。

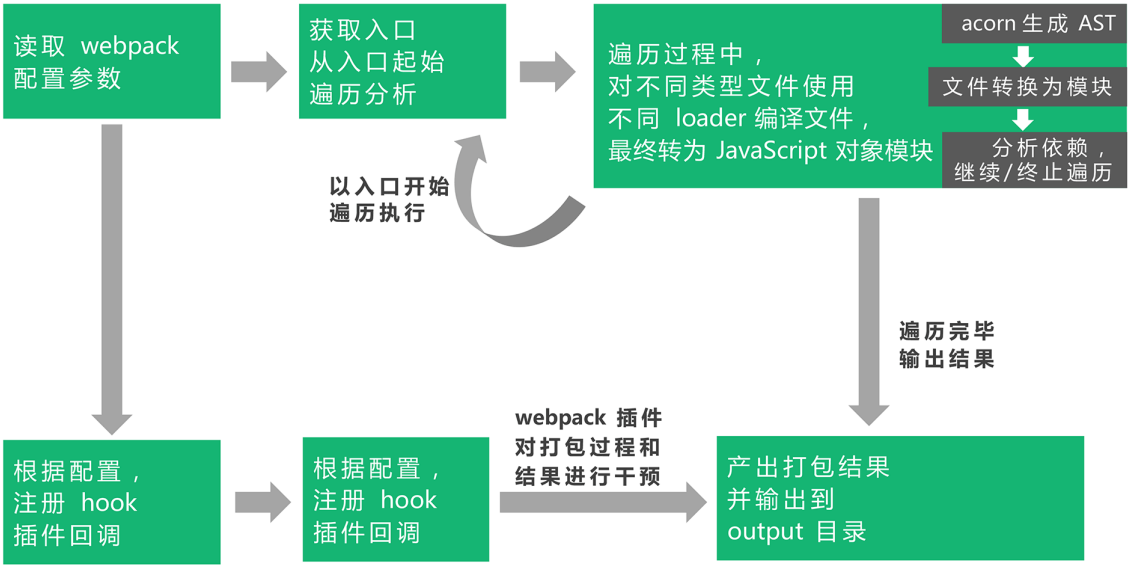
我们再次列出 webpack 主题的知识点：



webpack 工作基本原理

通过前文学习，我们知道了 webpack 编译产出，对结果进行分析。「知其然，知其所以然」，在知晓打包结果的基础上，接下来我们尝试分析产出过程，了解 webpack 工作的基本原理。

webpack 工作流程可以简单总结为下图：



首先，webpack 会读取项目中由开发者定义的 webpack.config.js 配置文件，或者从 shell 语句中获得必要的参数。这是 webpack 内部接收业务配置信息的方式。这就完成了配置读取的初步工作。

接着，实例化所需 webpack 插件，在 webpack 事件流上挂载插件钩子，这样在合适的构建过程中，插件具备了改动产出结果的能力。

同时，根据配置所定义的入口文件，以入口文件（可以不止有一个）为起始，进行依赖收集：对所有依赖的文件进行编译，这个编译过程依赖 loaders，不同类型文件根据开发者定义的不同 loader 进行解析。编译好的内容使用 acorn 或其它抽象语法树能力，解析生成 AST 静态语法树，分析文件依赖关系，将不同模块化语法（如 require）等替换为 `__webpack_require__`，即使用 webpack 自己的加载器进行模块化实现。

上述过程进行完毕后，产出结果，根据开发者配置，将结果打包到相应目录。

值得一提的是，在这整个打包过程中，**webpack 和插件采用基于事件流的发布订阅模式，监听某些关键过程，在这些环节中执行插件任务**。到最后，所有文件的编译和转化都已经完成，输出最终资源。

如果深入源码，上述过程用更加专业的术语总结为——模块会经历**加载**（loaded）、**封存**（sealed）、**优化**（optimized）、**分块**（chunked）、**哈希**（hashed）和**重新创建**（restored）这几个经典步骤。在这里，我们了解大体流程即可。

梳理完 webpack 工作「流水账」，我们还需要在理论上熟悉以下概念。

抽象语法树 AST

即便大家没有接触过 AST，也应该不是第一次听说这个概念。

在计算机科学中，抽象语法树（Abstract Syntax Tree，简称 AST），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构和表达。

之所以说语法是「抽象」的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如类似于 if-condition-then 这样的条件跳转语句，可以使用带有两个分支的节点来表示。

AST 并不会被计算机所识别，更不会被运行，它是对编程语言的一种表达，为代码分析提供了基础。

webpack 将文件转换成 AST 的目的就是方便开发者提取模块文件中的关键信息。 这样一来，我们就可以「知晓开发者到底写了什么东西」，也就可以根据这些「写出的东西」，实现分析和扩展。在代码层面，我们可以把 AST 理解为一个 object：

```
var ast = 'AST demo'
```

这样的语句转换为 AST 就是：

```
{
 "type": "Program",
 "start": 0,
 "end": 20,
 "body": [
```

```
    {
      "type": "VariableDeclaration",
      "start": 0,
      "end": 20,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 4,
          "end": 20,
          "id": {
            "type": "Identifier",
            "start": 4,
            "end": 7,
            "name": "ast"
          },
          "init": {
            "type": "Literal",
            "start": 10,
            "end": 20,
            "value": "AST demo",
            "raw": "'AST demo'"
          }
        }
      ],
      "kind": "var"
    }
  ],
  "sourceType": "module"
}
```

从中我们可以看出，AST 结果精确地表明了这是一条变量声明语句，语句起始于哪里，赋值结果是什么等信息都被表达出来。

一个更复杂的例子：

```
let tips = [1, 2]
```

```
function printTips() {
 tips.forEach((tip, i) => console.log(`Tip ${i}:` + tip))
}
```

会转化为：

```
{
 "type": "Program",
 "start": 0,
 "end": 285,
 "body": [
   {
     "type": "VariableDeclaration",
     "start": 179,
     "end": 197,
     "declarations": [
       {
         "type": "VariableDeclarator",
         "start": 183,
         "end": 196,
         "id": {
           "type": "Identifier",
           "start": 183,
           "end": 187,
           "name": "tips"
         },
         "init": {
           "type": "ArrayExpression",
           "start": 190,
           "end": 196,
           "elements": [
             {
               "type": "Literal",
               "start": 191,
               "end": 192,
               "value": 1,
               "raw": "1"
```

```
            },
            {
              "type": "Literal",
              "start": 194,
              "end": 195,
              "value": 2,
              "raw": "2"
            }
          ]
        }
      }
    ],
    "kind": "let"
  },
  {
    "type": "FunctionDeclaration",
    "start": 199,
    "end": 283,
    "id": {
      "type": "Identifier",
      "start": 208,
      "end": 217,
      "name": "printTips"
    },
    "expression": false,
    "generator": false,
    "params": [],
    "body": {
      "type": "BlockStatement",
      "start": 220,
      "end": 283,
      "body": [
        {
          "type": "ExpressionStatement",
          "start": 224,
          "end": 281,
          "expression": {
```

```
"type": "CallExpression",
"start": 224,
"end": 280,
"callee": {
  "type": "MemberExpression",
  "start": 224,
  "end": 236,
  "object": {
    "type": "Identifier",
    "start": 224,
    "end": 228,
    "name": "tips"
  },
  "property": {
    "type": "Identifier",
    "start": 229,
    "end": 236,
    "name": "forEach"
  },
  "computed": false
},
"arguments": [
  {
    "type": "ArrowFunctionExpression",
    "start": 237,
    "end": 279,
    "id": null,
    "expression": true,
    "generator": false,
    "params": [
      {
        "type": "Identifier",
        "start": 238,
        "end": 241,
        "name": "tip"
      },
      {
```

```
      "type": "Identifier",
      "start": 243,
      "end": 244,
      "name": "i"
    }
  ],
  "body": {
    "type": "CallExpression",
    "start": 249,
    "end": 279,
    "callee": {
      "type": "MemberExpression",
      "start": 249,
      "end": 260,
      "object": {
        "type": "Identifier",
        "start": 249,
        "end": 256,
        "name": "console"
      },
      "property": {
        "type": "Identifier",
        "start": 257,
        "end": 260,
        "name": "log"
      },
      "computed": false
    },
    "arguments": [
      {
        "type": "BinaryExpression",
        "start": 261,
        "end": 278,
        "left": {
          "type": "TemplateLiteral",
          "start": 261,
          "end": 272,
```

```
      "expressions": [
        {
          "type": "Identifier",
          "start": 268,
          "end": 269,
          "name": "i"
        }
      ],
      "quasis": [
        {
          "type": "TemplateElement",
          "start": 262,
          "end": 266,
          "value": {
            "raw": "Tip ",
            "cooked": "Tip "
          },
          "tail": false
        },
        {
          "type": "TemplateElement",
          "start": 270,
          "end": 271,
          "value": {
            "raw": ":",
            "cooked": ":"
          },
          "tail": true
        }
      ]
    },
    "operator": "+",
    "right": {
      "type": "Identifier",
      "start": 275,
      "end": 278,
      "name": "tip"
```

```
                        }
                      }
                    ]
                  }
                }
              ]
            }
          }
        ]
      }
    }
  ],
  "sourceType": "module"
}
```

我们看到，AST 结果除了表达出变量赋值 VariableDeclaration 信息以外，对函数声明 FunctionDeclaration 也做了精确的「解剖」，哪里出现了一个花括号，哪里实现了 API 调用，通过 AST 全部一览无余。

设想一下，有了这样的语法树，开发者便可以针对源文件进行一些「分析、加工或转换」操作。

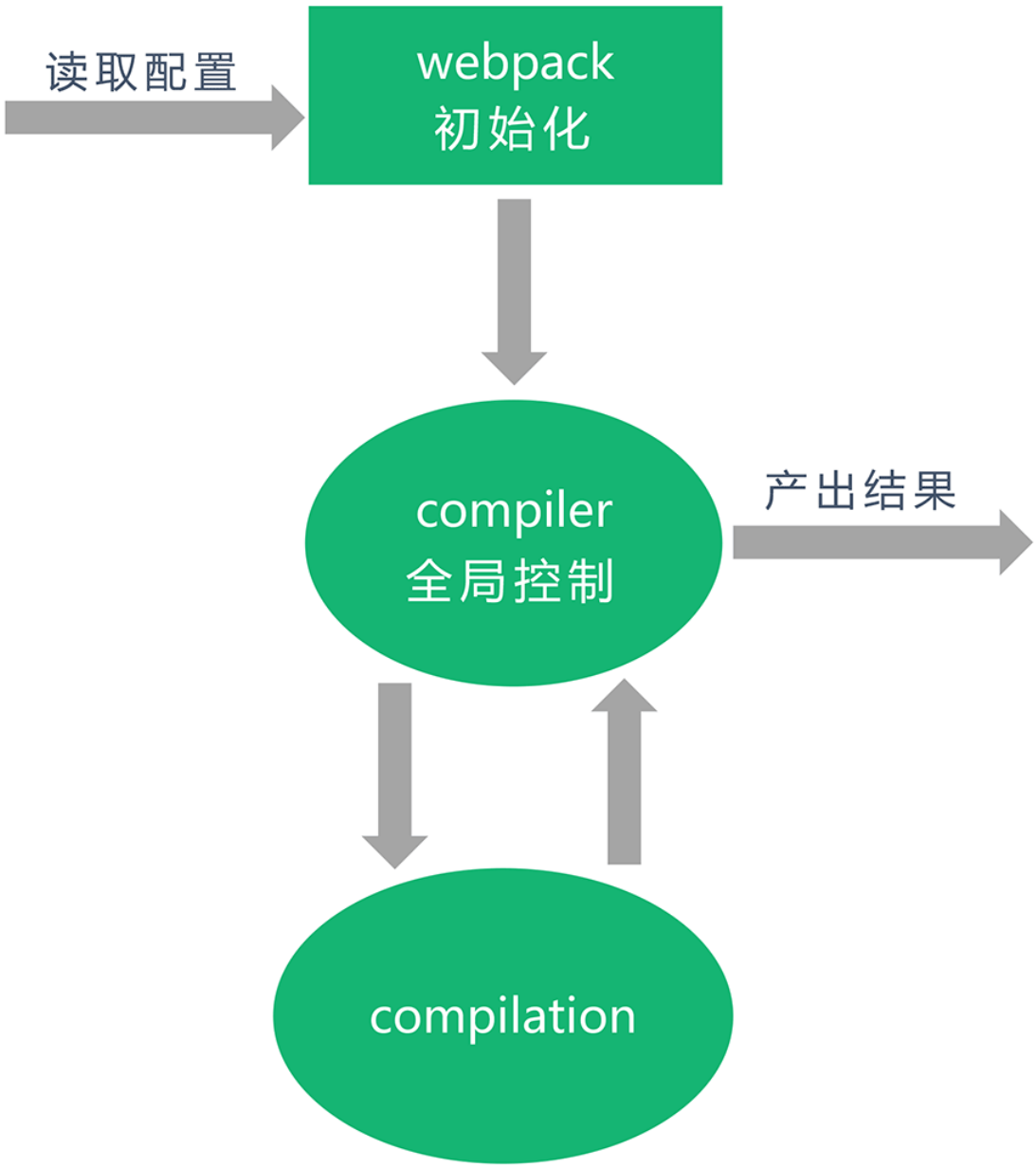
compiler 和 compilation

compiler 和 compilation 这两个对象是 webpack 核心原理中最重要的概念。它们是理解 webpack 工作原理、loader 和插件工作的基础。

> compiler 对象：它的实例包含了完整的 webpack 配置，全局只有一个 compiler 实例，因此它就像 webpack 的骨架或神经中枢。当插件被实例化的时候，会收到一个 compiler 对象，通过这个对象可以访问 webpack 的内部环境。

> compilation 对象：当 webpack 以开发模式运行时，每当检测到文件变化，一个新的 compilation 对象将被创建。这个对象包含了当前的模块资源、编译生成资源、变化的文件等信息。也就是说，所有构建过程中产生的构建数据都存储在该对象上，它也掌控着构建过程中的每一个环节。该对象也提供了很多事件回调供插件做扩展。

两者的关系可以通过以下图示说明：



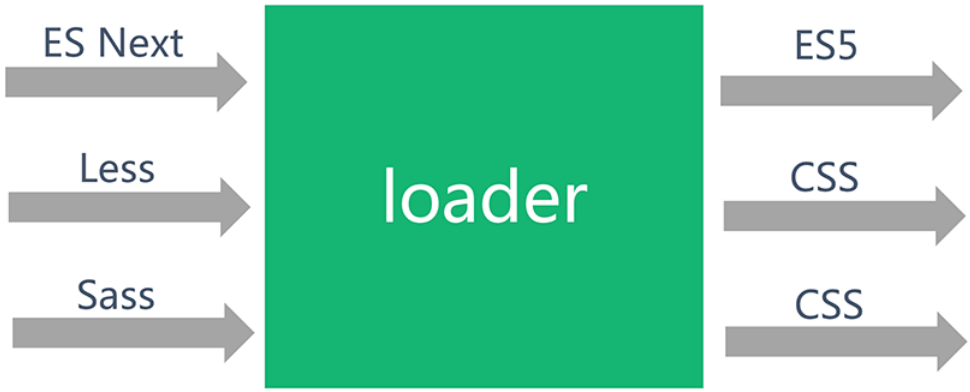
webpack 的构建过程是通过 compiler 控制流程，compilation 进行解析。**在开发插件时，我们可以从 compiler 对象中拿到所有和 webpack 主环境相关的内容，包括事件钩子。**更多信息我们将在下文介绍。

compiler 对象和 compilation 对象都继承自 tapable，tapable.js 这个库暴露了所有和事件相关的 pub/sub 的方法。webpack 基于事件流的 tapable 库，不仅能保证插件的有序性，还使得整个系统扩展性更好。

关于 tapable 库的解读我们到这里不再深入，感兴趣的读者可以参加后续讨论和学习后续文章内容。

探秘并编写 webpack loader

熟悉了概念，我们就来进行实战：了解如何编写一个 webpack loader。事实上，在 webpack 中，loader 是魔法真正发生的阶段之一：Babel 将 ES Next 编译成 ES5，sass-loader 将 SCSS/Sass 编译成 CSS 等，都是由相关 loader 或者 plugin 完成的。因此，直观上理解，**loader 就是接受源文件，对源文件进行处理，返回编译后文件**。如图：



我们看到一个 loader 秉承单一职责，完成最小单元的文件转换。当然，**一个源文件可能需要经历多步转换才能正常使用**，比如 Sass 文件先通过 sass-loader 输出 CSS，之后将内容交给 css-loader 处理，甚至 css-loader 输出的内容还需要交给 style-loader 处理，转换成通过脚本加载的 JavaScript 代码。如下使用方式：

```
module.exports = {
 ...
 module: {
   rules: [{
     test: /\.less$/,
     use: [{
       loader: 'style-loader' // 通过 JS 字符串，创建 style
node
     }, {
       loader: 'css-loader' // 编译 css 使其符合 CommonJS 规
范
     }, {
       loader: 'less-loader' // 编译 less 为 css
```

```
      }]
    }]
  }
}
```

当我们调用多个 loader 串联去转换一个文件时，每个 loader 会链式地顺序执行。webpack 中，在同一文件存在多个匹配 loader 的情况下，遵循以下原则：

> loader 的执行顺序是和配置顺序相反的，即配置的最后一个 loader 最先执行，第一个 loader 最后执行。
>
> 第一个执行的 loader 接收源文件内容作为参数，其他 loader 接收前一个执行的 loader 的返回值作为参数。最后执行的 loader 会返回最终结果。

如图，对应上面代码：

因此，在你开发一个 loader 时，请保持其职责的单一性，只需关心输入和输出。

不难理解：loader 本质就是函数，其最简单的结构为：

```
module.exports = function(source){
    // some magic...
    return content
}
```

loader 就是一个基于 CommonJS 规范的函数模块，它接受内容（这个内容可能是源文件也可能是经过其他 loader 处理后的结果），并返回新的内容。

更进一步，我们知道在配置 webpack 时，对于 loader 可以增加一些配置，比如著名的 babel-loader 的简单配置：

```
module:{
   rules:[
       {
           test: /\.js$/,
           exclude: /node_modules/,
           loader: "babel-loader",
           options: {
               "plugins": [
                   "dynamic-import-webpack"
               ]
           }
       }
   ]
}
```

这样一来，上文简单的 loader 写法便不能满足需求了，因为我们除了 source 以外，还需要根据开发者配置的 options 信息进行处理，以输出最后结果。那么如何获取 options 呢？这时候就需要 loader-utils 模块了：

```
const loaderUtils = require("loader-utils")
module.exports = function(source) {
   // 获取开发者配置的 options
   const options = loaderUtils.getOptions(this)
   // some magic...
   return content
}
```

另外，对于 loader 返回的内容，在实际开发中，单纯对 content 进行改写并返回也许是不够的。

比如，我们想对 loader 处理过程中的错误进行捕获，或者又想导出 sourceMap 等信息，该如何做呢？

这种情况需要用到 loader 中的 this.callback 进行内容的返回。this.callback 可以传入四个参数，分别是：

error：Error | null，当 loader 出错时向外抛出一个 error

content：String | Buffer，经过 loader 编译后需要导出的内容

sourceMap：为方便调试生成的编译后内容的 source map

ast：本次编译生成的 AST 静态语法树，之后执行的 loader 可以直接使用这个 AST，进而省去重复生成 AST 的过程

这样，我们的 loader 代码变得更加复杂，同时也能够处理更多样的需求：

```
module.exports = function(source) {
   // 获取开发者配置的 options
   const options = loaderUtils.getOptions(this)
   // some magic...
   // return content
   this.callback(null, content)
}
```

注意 当我们使用 `this.callback` 返回内容时，该 loader 必须返回 `undefined`，这样 webpack 就知道该 loader 返回的结果在 `this.callback` 中，而不是 return 中。

细心的读者会问，这里的 this 指向谁？事实上，这个 this 是一个叫 loaderContext 的 loader-runner 特有对象。如果刨根问底，就要细读 webpack loader 部分相关源码了，这并不是我们的主题，感兴趣的读者可以针对 webpack 源码再进行分析。

默认情况下，webpack 传给 loader 的内容源都是 UTF-8 格式编码的字符串。但请思考 file-loader 这个常用的 loader，它不是处理文本文件，而是处理二进制文件的，这种情况下，我们可以通过：source instanceof Buffer === true 来判断内容源类型：

```
module.exports = function(source) {
   source instanceof Buffer === true
```

```
    return source
}
```

如果自定义的 loader 也会返回二进制文件，需要在文件中显式注明：

```
module.exports.raw = true
```

当然，还存在异步 loader 的情况，即对 source 的处理并不能同步完成，这时候使用简单的 async-await 即可：

```
module.exports = async function(source) {
    function timeout(delay) {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve(source)
            }, delay)
        })
    }
    const content = await timeout(1000)
    this.callback(null, content)
}
```

另一种异步 loader 解决方案是使用 webpack 提供的 this.async，调用 this.async 会返回一个 callback Function，在异步完成之后，我们进行调用。上面的示例代码可以改写为：

```
module.exports = async function(source) {
    function timeout(delay) {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve(source)
            }, delay)
        })
    }
    const callback = this.async()
    timeout(1000).then(data => {
```

```
        callback(null, data)
    })
}
```

实际上，对于我们熟悉的 less-loader，翻看其源码，就能发现它的核心是利用 less 这个库来解析 less 代码，less 会返回一个 promise，因此 less-loader 是异步的，其实现正是运用了 this.async() 来完成。

到此，我们了解了 loader 的编写套路，更多细节内容，比如 loader 缓存开关、全程传参 pitch 等用法不再过多讨论，读者可以根据需要进行了解，也欢迎在课程评论区大家一起讨论沟通。

实战案例

工程师想要进阶，一定要「学以致用」，解决实际问题。我们现在来编写一个 path-replace-loader，这个 loader 将允许自定义替换 require 语句中的 base path 为动态指定 path，使用和配置方式为：

```
module.exports = {
    module: {
        rules: [{
            test: /\.js$/,
            loader: 'path-replace-loader',
            exclude: /(mode_modules)/,
            options: {
                path: 'ORIGINAL_PATH',
                replacePath: 'REPLACE_PATH'
            }
        }]
    }
}
```

根据上面所介绍内容，我们给出 path-replace-loader 源码如下：

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

```
module.exports = function(source) {
   this.cacheable && this.cacheable()
   const callback = this.async()
   const options = loaderUtils.getOptions(this)

   if (this.resourcePath.indexOf(options.path) > -1) {
       const newPath =
this.resourcePath.replace(options.path,
options.replacePath)

       fs.readFile(newPath, (err, data) => {
           if (err) {
               if (err.code === 'ENOENT') return
callback(null, source)
               return callback(err)
           }

           this.addDependency(newPath)
           callback(null, data)
       })
   }
   else {
       callback(null, source)
   }
}

module.exports.raw = true
```

这只是一个简单的实例，但是涵盖了 loader 编写的不少内容，我们来简单分析一下：这是一个异步 loader，我们使用了下面，

```
const callback = this.async()
// ...
callback(null, data)
```

的返回方式。通过：

```
const options = loaderUtils.getOptions(this)
// ...
const newPath = this.resourcePath.replace(options.path,
options.replacePath)
```

获取开发者的配置信息，并与 this.resourcePath（当前资源文件路径）比对，进行路径替换。

对于错误的处理也很简单：如果新的目标路径文件不存在，则返回原路径文件：

```
if (err.code === 'ENOENT') return callback(null, source)
```

其它错误也一并通过 return callback(err) 抛出。

主逻辑使用了 this.addDependency(newPath) 将新的文件加入到 webpack 依赖当中，并返回内容 callback(null, data)。

这个过程并不复杂，同时思路非常清晰，通过这个案例，读者可以根据自身团队需求，编写不同复杂度的 wepback loader，实现不同程度的拓展。

探秘并编写 webpack plugin

除了 webpack loader 这个核心概念以外，webpack plugin 是另一个重要话题。loader 和 plugin 就像 webpack 的双子星，有着共同之处，但是分工却很明晰。

我们反复提到过 webpack **事件流机制**，也就是说在 webpack 构建的生命周期中，会广播许多事件。这时候，开发中注册的各种插件，便可以根据需要监听与自身相关的事件。捕获事件后，在合适的时机通过 webpack 提供的 API 去改变编译输出结果。

因此，我们可以总结出 loader 和 plugin 的**差异**。

loader 其实就是一个转换器，执行单纯的文件转换操作。

plugin 是一个扩展器，它丰富了 webpack 本身，在 loader 过程结束后，webpack 打包的整个过程中，weback plugin 并不直接操作文件，而是基于事件机制工作，监听 webpack 打包过程中的某些事件，见缝插针，修改打包结果。

究竟应该如何从零开始，编写一个 webpack 插件呢?

首先我们要清楚当前插件要解决什么问题，根据问题，找到相应的钩子事件，在相关事件中进行操作，改变输出结果。这就需要清楚开发中都有哪些钩子了，下面列举一些常用的，完整内容可以在官网找到：Compiler 暴露的所有事件钩子。

我们知道 compiler 对象暴露了和 webpack 整个生命周期相关的钩子，通过如下的方式访问：

```
//基本写法
compiler.hooks.someHook.tap(...)
```

例如，如果希望 webpack 在读取 entry 配置完后就执行某项工作，我们可以：

```
compiler.hooks.entryOption.tap(...)
```

因为名字为 entryOption 的 SyncBailHook 类型 hook，就表明了入口配置信息执行完毕的事件，在相关 tap 函数中我们可以在这个时间节点插入操作。

又如，如果希望在生成的资源输出之前执行某个功能，我们可以：

```
compiler.hooks.emit.tap(...)
```

因为名字为 emit 的 AsyncSeriesHook 类型 hook，就表明了资源输出前的时间节点。

一个自定义 webpack plugin 的骨架结构就是一个带有 apply 方法的 class（用 prototype 实现同理 CustomPlugin.prototype.apply = function () {...}）：

```
class CustomPlugin {
   constructor(options) {
       this.options = options
   }
   apply(compiler) {
       // 相关钩子注册回调
       compiler.hooks.someHook.tap('CustomPlugin', () => {
           // magic here...
       })

       // 打印出此时 compiler 暴露的钩子
       for(var hook of Object.keys(compiler.hooks)){
           console.log(hook)
       }
   }
}

module.exports = customPlugin
```

除了 compiler 暴露了与 webpack 整体构建生命周期相关的钩子以外，compilation 也暴露了与模块和依赖有关的粒度更小的钩子，读者可以参考：compilation 暴露的所有事件钩子，找到合适的时机插入自定义行为。

其实 compilation 是 compiler 生命周期中的一个步骤，使用 compilation 相关钩子的通用写法为：

```
class CustomPlugin {
   constructor(options) {
       this.options = options
   }
   apply(compiler) {
       compiler.hooks.compilation.tap('CustomPlugin',
function(compilation, callback) {
           compilation.hooks.someOtherHook.tap('SomePlugin
',function() {
               // some magic here
           })
```

```
        })
    }
}

module.exports = customPlugin
```

最终，我们可以总结一下 webpack 插件的**套路**。

定义一个 JavaScript class 函数，或在函数原型（prototype）中定义一个以 compiler 对象为参数的 apply 方法。

apply 函数中通过 compiler 插入指定的事件钩子，在钩子回调中拿到 compilation 对象。

使用 compilation 操纵修改 webapack 打包内容。

当然，plugin 也存在异步的情况，一些事件钩子是异步的。相应地，我们可以使用 tapAsync 和 tapPromise 方法来处理：

```
class CustomAsyncPlugin {
    constructor(options) {
        this.options = options
    }
    apply(compiler) {
        compiler.hooks.emit.tapAsync('CustomAsyncPlugin',
function(compilation, callback) {
            setTimeout(() => {
                callback()
            }, 1000)
        })

        compiler.hooks.emit.tapPromise('CustomAsyncPlugin',
function(compilation, callback) {
            return asyncFun().then(() => {
                //...
            })
```

```
        })
    }
}
```

实战案例

接下来，我们来编写一个简单的 webpack 插件。相信不少 React 开发者了解：在使用 create-react-app 开发项目时，如果发生错误，会出现 error overlay 提示。我们来开发一个类似的功能，使用如下代码：

```
module.exports = {
    // ...
    plugins: [new ErrorOverlayPlugin()],
    devtool: 'cheap-module-source-map',
    devServer: {}
}
```

我们借助 errorOverlayMiddleware 中间件来进行错误拦截并展示：

```
import errorOverlayMiddleware fomt 'react-dev-
utils/errorOverlayMiddleware'

class ErrorOverlayPlugin {
    apply(compiler) {
        const className = this.constructor.name
        if (compiler.options.mode !== 'development') return

        compiler.hooks.entryOption.tap(className, (context,
entry) => {
            const chunkPath = require.resolve('./entry')
            adjustEntry(entry, chunkPath)
        })

        compiler.hooks.afterResolvers.tap(className, ({
options }) => {
            if (options.devServer) {
                const originalBefore =
```

```
options.devServer.before
                option.devServer.before = (app, server) =>
{
                    if (originalBefore) {
                        originalBefore(app, server)
                    }
                    app.use(errorOverlayMiddleware())
                }

            }
        })
    }
}

function adjustEntry(entry, chunkPath) {
    if (Array.isArray(entry)) {
        if (!entry.includes(chunkPath)) {
            entry.unshift(chunkPath)
        }
    }
    else {
        Object.keys(entry).forEach(entryName => {
            entry[name] = adjustEntry(entry[entryName],
chunkPath)
        })
    }
}

module.exports = ErrorOverlayPlugin
```

参考实现源码，我们发现，编写一个 webpack plugin 确实并不困难，只需要开发者了解相关步骤，熟记相关钩子，并多加尝试即可。

简单分析一下上面代码，在非生产环境下，不打开错误窗口，而是直接返回，以免影响线上体验：

```
if (compiler.options.mode !== 'development') return
```

在 entryOption hook 中，获取开发者配置的 entry 并通过 adjustEntry 方法获取正确的入口模块，该方法支持 entry 配置为 array 和 object 两种形式。在 afterResolvers hook 中，判断开发者是否开启 devServer，并对相关中间件进行调用 app.use(errorOverlayMiddleware())。

实际生产环境当中，webpack pulgin 生态丰富多样，一般已有插件就可以满足大部分开发需求。如果团队结合自身业务需求，自主编写 webpack plugin，进而反哺生态，非常值得鼓励。

webpack plugin 开发重点

本节目前为止所介绍的内容已经可以带领大家入门插件开发。学习过程中我们会发现，webpack 插件开发重点在于对 compilation 和 compiler 以及两者对应钩子事件的理解、运用。我们提到 webpack 的事件机制基于 tapable 库，因此想完全理解 webpack 事件和钩子，有必要学习 tapable 。

事实上，tapable 更加复杂而「神通广大」，它除了提供同步和异步类型的钩子以外，又根据执行方式，串行/并行，衍生出 Bail、Waterfall、Loop 多种类型。站在 tapable 等的肩膀上，webpack 插件的开发更加灵活，可扩张性更强。

学习的目的在于应用。相信通过本小节的学习，读者已经能够理解 webpack 开发插件的流程。根据项目需要和业务特点，手握 webpack 插件开发的理论钥匙，在实践中多摸索、多尝试，每个人都一定会有所收获。

webpack VS Rollup

Rollup 号称下一代打包方案，它的功能和特点非常突出：

依赖解析，打包构建

仅支持 ES Next 模块

Tree shaking

Rollup 凭借其清新且友好的配置，以及强大的功能横空出世，吸睛无数。

可以说，Webpack 算得上目前最流行的打包方案，而 Rollup 是下一代打包方案，两者有何区别？目前业界对两者的定位，可以总结为一句话：**建库使用 Rollup，其他场景使用 webpack。**

为什么这么说呢？还记得我们在前面提的 webpack 打包结果吗？从结果上看，webpack 方案会生成比较多的冗余代码，这对于业务代码来说没什么问题，能保证较强的程序健硕性和语法还原度，兼容性保障更有利。也许开发者会关心代码量多带来的冗余问题，但衡量其优缺点和开发效率性价比，webpack 始终是业务开发的首选；但对于库来说就不一样了，相同的脚本，使用 Rollup 产出，复杂的模块冗余会完全消失。Rollup 通过将代码顺序引入同一个文件来解决模块依赖问题，因此，Rollup 做拆包的话就会有问题，原因是模块完全透明了，而在复杂应用中我们往往需要进行拆包，在库的编写中很少用到这样的功能。

当然，「库使用 Rollup，其他场景使用 webpack」——这不是一个绝对的原则。如果你需要代码拆分（Code Splitting），或者有很多静态资源需要处理，或者你构建的项目需要引入很多 CommonJS 规范的模块，再或者你需要拥有相对更大的社区支持，那么 webpack 是不错的选择。

如果你的代码库基于 ES Next 模块，且希望自己写的代码能够被其他人直接使用，那么，你需要的打包工具可能就是 Rollup 。

我们借用前面小节的代码，来看看经过 Rollup 编译之后的代码会成什么样子。

main.js：

```
import sayHello from './hello.js'
console.log(sayHello('lucas'))
```

hello.js：

```
const sayHello = name => `hello ${name}`
export default sayHello
```

编译结果非常简单：

```
const sayHello = name => `hello ${name}`
console.log(sayHello('lucas'))
```

这与 webpack 的打包产出形成了鲜明差异。这种打包方式，天然支持 tree shaking，我们改写上例，加入一个没有用到的 sayHi 函数：

main.js：

```
import { sayHello } from './hello.js'

console.log( sayHello( 'lucas' ) )
```

hello.js：

```
export const sayHi = name => `hi ${name}`

export const sayHello = name => `hello ${name}`
```

打包结果：

```
'use strict';

const sayHello = name => `hello ${name}`;
console.log( sayHello( 'lucas' ) );
```

通过顺序引入依赖，非常简单、清晰，并且自动做到了 tree shaking，其中的原理和更多话题我们将在「深入浅出模块化」相关内容继续说明。

综合运用

至此，我们对于 webpack 已经有了较为深入的理解。但是，以上实战代码都是些较小型的 demo，综合运用这些知识到底能解决哪些问题呢?

我这里有一个很好的例子。

我们知道，2018 年号称小程序元年。以微信小程序为首，百度智能小程序、支付宝小程序、头条小程序纷纷入局。作为开发人员应该注意到，在带给开发无限红利的同时，由于各平台小程序的开发语法和技术方案不尽相同，因而也带来了巨大的多端开发成本。

如果团队能够实现这样一个脚手架：**以微信小程序为基础，将微信小程序的代码平滑转换为各端小程序，岂不大幅提高开发效率?**

可是技术方案上，应该如何实现呢？受 cantonjs 启发，我们团队打造了一款跨多端小程序脚手架，其**基本原理**正是以 webpack 开发架构为基础，对于微信小程序的规范化打包，以及不同平台的差异化编译，主要依靠自定义实现 webpack loader 和 webpack plugin 来填平。

在这套脚手架基础上，开发者可以选择任何一套小程序源代码（基于微信小程序/支付宝小程序/百度小程序）来开发多端小程序。脚手架支持自动编译 wxml 文件（微信小程序）为 axml 文件（支付宝小程序）或 swan 文件（百度小程序），能够转换基础平台 API： wx（微信小程序核心对象） 为 my（支付宝小程序核心对象） 或 swan（百度小程序核心对象），反之亦然。对于个别接口在平台上的天生差异，开发者可以通过 `__WECHAT__` 或 `__ALIPAY__` 或 `__BAIDU__` 来动态处理。

具体细节，我们可以通过 DefinePlugin 这个 webpack 内置插件在 webpack 编译阶段注册全局变量： `__WECHAT__` 或 `__ALIPAY__` 或 `__BAIDU__`。

```
new webpack.DefinePlugin({
 // Definitions...
})
```

通过 webpack loader 使 webpack 能编译或处理 *.wxml 上引用的文件，并将原 App 中的 API 进行转换，使用方式与正常的 webpack 配置 loader 完全相同：

```
{
 test: /\.wxml$/,
 include: /src/,
 use: [
   {
```

```
      loader: 'file-loader',
      options: {
        name: '[name].[ext]',
        useRelativePath: true,
        context: resolve('src'),
      },
    },
    {
      loader: 'mini-program-loader',
      options: {
        root: resolve('src'),
        enforceRelativePath: true,
      },
    },
  ],
}
```

注意，我们声明 loader 的顺序表明先通过 mini-program-loader 处理，其结果交给 file-loader 处理。mini-program-loader 的实现并不复杂，我们通过 sax.js 解析 wxml （XML 风格）文件，进行 API 转换。sax.js 是解析 XML 或者 HTML 的基础库，正好适用于我们各端小程序的主文档文件（wxml、swan、axml）。

通过 webpack-plugin 插件实现自动分析 ./app.js 入口文件，并智能打包，同时抹平 API 差异。

```
import MiniProgramWebpackPlugin from 'mini-program-
webpack-plugin'
export default {
  // ...configs,
  plugins: [
    // ...other,
    new MiniProgramWebpackPlugin(options)
  ],
}
```

在这两个 loader 和 plugin 的基础上，我们实现的这个脚手架构建，通过 script 脚本，启动不同目标的小程序平台编译：yarn start、yarn start:alipay、yarn start:baidu，同时开发者可以根据自身项目特点，添加 prettier 和 lint 标准等。

到此，一个基于 webpack、webpack loader、webpack plugin 的脚手架综合应用从场景到实现已经简要介绍完毕。

通过这个案例，我们发现 webpack 的能力边界是无穷的，以高级前端工程师为目标的程序员，应该尽最大努力来开发 webpack 的潜能。

总结

正如本课程的标题所示：**webpack 工程师 > 前端工程师**。 webpack 要求的不仅仅是「配置工程师」那么简单，其后蕴含的 Node.js 知识、AST 知识、架构设计、代码设计原则等非常值得玩味。我们不应该畏难，社区为我们提供了大量的开箱即用工具，借助这些工具，希望大家能够掌握这方面的知识，并在此基础上运用自如。

课程代码仓库：https://github.com/HOUCe/lucas-gitchat-courses

点击查看下一节

前端工程化背后的项目组织设计（上）