前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

webpack 工程师 > 前端工程师（上）

说起前端工程化， webpack 必然在前端工具链中占有最重要的地位；说起前端工程师进阶，webpack 更是一个绕不开的话题。

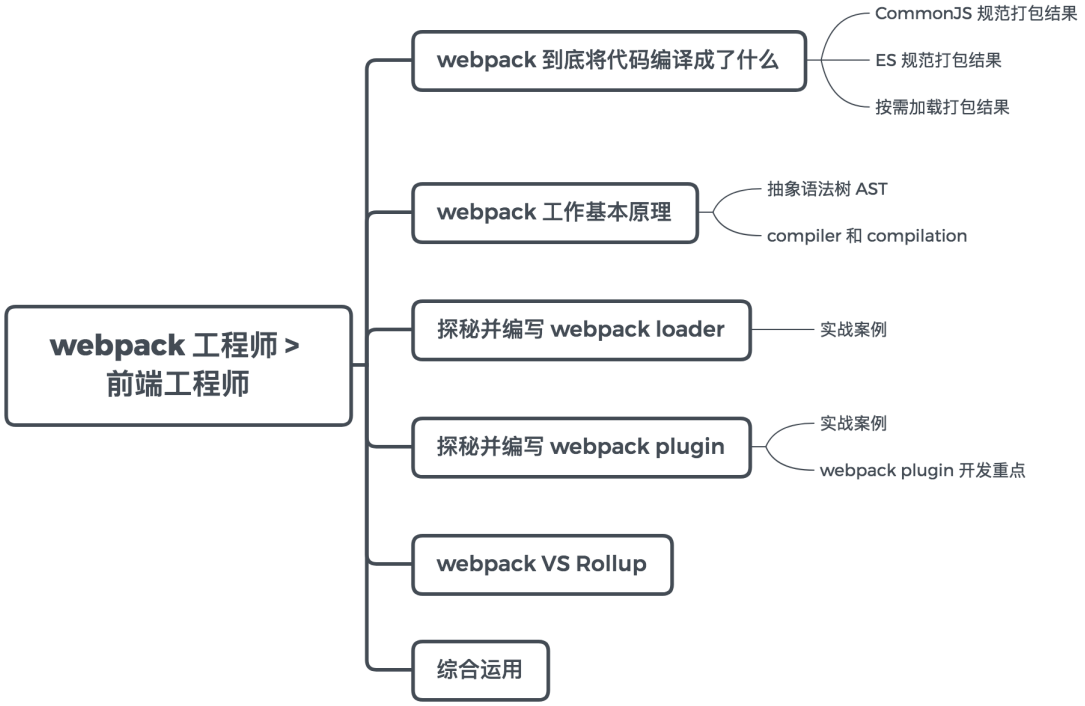
从原始的刀耕火种时代，到 Gulp、Grunt 等早期方案的横空出世，再到 webpack 通过其丰富的功能和开放的设计一举奠定「江湖地位」，我想每个前端工程师都需要熟悉各个时代的「打包神器」。

作为团队中不可或缺的高级工程师，能否玩转 webpack，能否通过工具搭建令人舒适的工作流和构建基础，能否不断适应技术发展打磨编译体系，将直接决定你的工作价值。

在这一系列课程里，赘述社区上大量存在的「webpack 配置 demo」，或者讲解一些现成的插件应用意义不大，这些知识都可以免费找到。

分析 webpack 工作原理，探究 webpack 能力边界，结合实践并加以应用将会是本讲的重点。

webpack 主题的知识点如下所示：



接下来，我们通过 2 节内容来学习这个主题。

webpack 到底将代码编译成了什么

项目中经过 webpack 打包后的代码究竟被编译成了什么？也许你认为并不重要。业务中的代码往往非常复杂，经过 webpack 编译后的代码可读性非常差。但是不管是复杂的项目还是最简单的一行代码，其经过 webpack 编译打包的**产出本质是相同的**。我们试图从最简单的情况开始，研究 webpack 打包产出的秘密。

CommonJS 规范打包结果

如何着手分析呢？首先创建并切入到项目，进行初始化：

```
mkdir webpack-demo
cd webpack-demo
npm init -y
```

安装 webpack 最新版本：

```
npm install --save-dev webpack
npm install --save-dev webpack-cli
```

根目录下创建 index.html：

创建 ./src 文件。因为我们要研究模块化打包产出，这一定涉及依赖关系，因此在 ./src 目录下创建 hello.js 和 index.js，其中 index.js 为入口脚本，它将依赖 hello.js：

```
const sayHello = require('./hello')
console.log(sayHello('lucas'))
```

hello.js：

```
module.exports = function (name) {
   return 'hello ' + name
}
```

这里我们为了演示，采用了 CommonJS 规范，也没有加入 Babel 编译环节。

直接执行命令：

```
node_modules/.bin/webpack --mode development
```

便得到了产出 ./dist，打开 ./dist/main.js，得到最终编译结果：

```
(function(modules) {
   //缓存已经加载过的 module 的 exports，防止 module 在 exports 之前 JS 重复执行
   var installedModules = {};

   //类似 commonJS 的 require()，它是 webpack 加载函数，用来加载 webpack 定义的模块，返回 exports 导出的对象
   function __webpack_require__(moduleId) {
       //缓存中存在，则直接返回结果
       if (installedModules[moduleId]) {
           return installedModules[moduleId].exports
       }

       //第一次加载时，初始化模块对象，并进行缓存
       var module = installedModules[moduleId] = {
           i: moduleId, // 模块 ID
           l: false, // 是否已加载标识
           exports: {} // 模块导出对象
       };

       /**
       * 执行模块
       * @param module.exports -- 模块导出对象引用，改变模块包裹函数内部的 this 指向
       * @param module -- 当前模块对象引用
       * @param module.exports -- 模块导出对象引用
       * @param __webpack_require__ -- 用于在模块中加载其他模块
       */
       modules[moduleId].call(module.exports, module, module.exports, __webpack_require__);

       //标记是否已加载标识
       module.l = true;
```

```
        //返回模块导出对象引用
        return module.exports
    }

    __webpack_require__.m = modules;
    __webpack_require__.c = installedModules;
    //定义 exports 对象导出的属性
    __webpack_require__.d = function(exports, name, getter)
{
        //如果 exports （不含原型链上）没有 [name] 属性，定义该属
性的 getter
        if (!__webpack_require__.o(exports, name)) {
            Object.defineProperty(exports, name, {
                enumerable: true,
                get: getter
            })
        }
    };
    __webpack_require__.r = function(exports) {
        if (typeof Symbol !== 'undefined' &&
Symbol.toStringTag) {
            Object.defineProperty(exports,
Symbol.toStringTag, {
                value: 'Module'
            })
        }
        Object.defineProperty(exports, '__esModule', {
            value: true
        })
    };
    __webpack_require__.t = function(value, mode) {
        if (mode & 1) value = __webpack_require__(value);
        if (mode & 8) return value;
        if ((mode & 4) && typeof value === 'object' &&
value && value.__esModule) return value;
        var ns = Object.create(null);
        __webpack_require__.r(ns);
```

```
        Object.defineProperty(ns, 'default', {
            enumerable: true,
            value: value
        });
        if (mode & 2 && typeof value != 'string') for (var
key in value) __webpack_require__.d(ns, key, function(key)
{
            return value[key]
        }.bind(null, key));
        return ns
    };
    __webpack_require__.n = function(module) {
        var getter = module && module.__esModule ?
        function getDefault() {
            return module['default']
        } : function getModuleExports() {
            return module
        };
        __webpack_require__.d(getter, 'a', getter);
        return getter
    };
    __webpack_require__.o = function(object, property) {
        return Object.prototype.hasOwnProperty.call(object,
property)
    };
    // __webpack_public_path__
    __webpack_require__.p = "";

    //加载入口模块并返回入口模块的 exports
    return __webpack_require__(__webpack_require__.s =
"./src/index.js")
})({
    "./src/hello.js": (function(module, exports) {
        eval("module.exports = function(name) {\n    return
'hello ' + name\n}\n\n//#
sourceURL=webpack:///./src/hello.js?")
    }),
```

```
  "./src/index.js": (function(module, exports,
__webpack_require__) {
      eval("var sayHello = __webpack_require__(/*!
./hello */
\"./src/hello.js\")\nconsole.log(sayHello('lucas'))\n\n//#
sourceURL=webpack:///./src/index.js?")
  })
});
```

不要着急阅读，我们先把最核心的代码骨架提出来，上面的代码其实就是一个 IIFE（立即执行函数表达式）：

```
(function(modules){
 // ...
})({
 "./src/hello.js": (function(){
   // ...
 }),
 "./src/index.js": (function() {
   // ...
 })
})
```

Ben Cherry 的著名文章 JavaScript Module Pattern: In-Depth"> JavaScript Module Pattern: In-Depth 介绍了 IIFE 实现模块化的多种进阶尝试，阮一峰老师在其博客中也提到了相关内容。用 IIFE 实现模块化，我们并不陌生。

深入上述代码结果（已添加注释），我们可以提炼出以下关键几点。

> webpack 打包结果就是一个 IIFE，一般称它为 webpackBootstrap，这个 IIFE 接收一个对象 modules 作为参数，modules 对象的 key 是依赖路径，value 是经过简单处理后的脚本（它不完全等同于我们编写的业务脚本，而是被 webpack 进行包裹后的内容）。
>
> 打包结果中，定义了一个重要的模块加载函数 `__webpack_require__`。

我们首先使用 __webpack_require__ 加载函数去加载入口模块 ./src/index.js。

加载函数 __webpack_require__ 使用了闭包变量 installedModules，它的作用是将已加载过的模块结果保存在内存中。

如果读者对于产出结果源码存在不理解的地方，请继续阅读，我们将会在 webpack 工作基本原理部分进一步说明，同时欢迎随时在评论区跟我讨论。

ES 规范打包结果

以上是基于 CommonJS 规范的模块化写法，业务中我们的代码往往遵循 ES Next 模块化标准，并通过 Babel 进行编译，这样的流程下，会得到什么结果呢?

我们动手尝试一下，安装依赖：

```
npm install --save-dev webpack
npm install --save-dev webpack-cli
npm install --save-dev babel-loader
npm install --save-dev @babel/core
npm install --save-dev @babel/preset-env
```

同时配置 package.json，加入：

```
"scripts": {
   "build": "webpack --mode development --progress --
display-modules --colors --display-reasons"
s},
```

设置 npm script 以方便运行 webpack 构建，同时在 package.json 中加入 Babel 配置：

```
"babel": {
   "presets": ["@babel/preset-env"]
}
```

将 index.js 和 hello.js 改写为 ESM 方式：

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

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

执行：

```
npm run build
```

得到的打包主体与之前内容基本一致。但是细节上，我们发现 IIFE 传入参数 modules 对象的 value 部分，即执行脚本内容多了以下语句：

```
__webpack_require__.r(__webpack_exports__)
```

实际上 `__webpack_require__.r` 这个方法是给模块的 exports 对象加上 ES 模块化规范的标记。

具体标记方式为：如果支持 Symbol 对象，则通过 Object.defineProperty 为 exports 对象的 Symbol.toStringTag 属性赋值 Module，这样做的结果是 exports 对象在调用 toString 方法时会返回 Module；同时，将 `exports.__esModule` 赋值为 true。

除了 CommonJS 和 ES Module 规范，webpack 同样支持 AMD 规范，这里不再进行分析，读者可以重新打包来观察它们的区别。总之，希望大家记住 webpack 打包输出的结果就是一个 IIFE，通过这个 IIFE，以及 **`webpack_require`** 支持了各种模块化打包方案。

按需加载打包结果

现代化的业务，尤其是在单页应用中，我们往往使用「按需加载」，那么对于这种相对较新的依赖技术，webpack 又会产出什么样的代码呢？

我们加入 Babel 插件，以支持 dynamic import：

```
npm install --save-dev babel-plugin-dynamic-import-webpack
```

并在 webpack.config.js 中添加相关插件配置：

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

同时，将 index.js 使用 dynamic import 的方式实现按需加载：

```
import('./hello').then(sayHello => {
   console.log(sayHello('lucas'))
})
```

最后执行：

```
npm run build
```

这样一来，我们发现重新构建后会输出两个文件，分别是执行入口文件 main.js 和异步加载文件 0.js，因为异步按需加载显然不能把所有的代码再打到一个 bundle 当中了。

0.js 内容为：

```
(window["webpackJsonp"] = window["webpackJsonp"] ||
[]).push([
[0],
{
   "./src/hello.js": (function(module,
__webpack_exports__, __webpack_require__) {
       "use strict";
       eval("__webpack_require__.r(__webpack_exports__);\n
// module.exports = function(name) {\n//     return 'hello
' + name\n// }\nvar sayHello = function sayHello(name) {\n
 return \"hello \".concat(name);\n};\n\n/* harmony default
export */ __webpack_exports__[\"default\"] =
(sayHello);\n\n//# sourceURL=webpack:///./src/hello.js?")
   })
}])
```

main.js 内容也与之前相比变化较大：

```
(function(modules) {
   /***
   * webpackJsonp 用于从异步加载的文件中安装模块
   * 把 webpackJsonp 挂载到全局是为了方便在其他文件中调用
   *
   * @param chunkIds 异步加载的文件中存放的需要安装的模块对应的
Chunk ID
   * @param moreModules 异步加载的文件中存放的需要安装的模块列表
   * @param executeModules 在异步加载的文件中存放的需要安装的模
块都安装成功后，需要执行的模块对应的 index
   */
   function webpackJsonpCallback(data) {
       var chunkIds = data[0];
       var moreModules = data[1];
       var moduleId, chunkId, i = 0,
           resolves = [];
       // 把所有 chunkId 对应的模块都标记成已经加载成功
```

```
        for (; i < chunkIds.length; i++) {
            chunkId = chunkIds[i];
            if (installedChunks[chunkId]) {
                resolves.push(installedChunks[chunkId][0])
            }
            installedChunks[chunkId] = 0
        }
        for (moduleId in moreModules) {
            if
(Object.prototype.hasOwnProperty.call(moreModules,
moduleId)) {
                modules[moduleId] = moreModules[moduleId]
            }
        }
        if (parentJsonpFunction) parentJsonpFunction(data);
        while (resolves.length) {
            resolves.shift()()
        }
    };

    var installedModules = {};
    // 存储每个 Chunk 的加载状态
    // 键为 Chunk 的 ID，值为 0 代表已经加载成功
    var installedChunks = {
    "main": 0
    };

    function jsonpScriptSrc(chunkId) {
        return __webpack_require__.p + "" + ({}[chunkId] ||
chunkId) + ".js"
    }
    function __webpack_require__(moduleId) {
        if (installedModules[moduleId]) {
            return installedModules[moduleId].exports
        }
        var module = installedModules[moduleId] = {
            i: moduleId,
```

```
            l: false,
            exports: {}
        };
        modules[moduleId].call(module.exports, module,
module.exports, __webpack_require__);
        module.l = true;
        return module.exports
    }

    /**
    * 用于加载被分割出去的，需要异步加载的 Chunk 对应的文件
    * @param chunkId 需要异步加载的 Chunk 对应的 ID
    * @returns {Promise}
    */
    __webpack_require__.e = function requireEnsure(chunkId)
{
        var promises = [];
        var installedChunkData = installedChunks[chunkId];
        // 如果加载状态为 0 表示该 Chunk 已经加载成功了，直接返回
resolve Promise
        if (installedChunkData !== 0) {
            if (installedChunkData) {
                promises.push(installedChunkData[2])
            } else {
                var promise = new Promise(function(resolve,
reject) {
                    installedChunkData =
installedChunks[chunkId] = [resolve, reject]
                });
                promises.push(installedChunkData[2] =
promise);
                var script =
document.createElement('script');
                var onScriptComplete;
                script.charset = 'utf-8';
                // 设置异步加载的最长超时时间
                script.timeout = 120;
```

```
                if (__webpack_require__.nc) {
                    script.setAttribute("nonce",
__webpack_require__.nc)
                }
                // 文件的路径为配置的 publicPath、chunkId 拼接
而成
                script.src = jsonpScriptSrc(chunkId);
                onScriptComplete = function(event) {
                    script.onerror = script.onload = null;
                    clearTimeout(timeout);
                    var chunk = installedChunks[chunkId];
                    if (chunk !== 0) {
                        if (chunk) {
                            var errorType = event &&
(event.type === 'load' ? 'missing' : event.type);
                            var realSrc = event &&
event.target && event.target.src;
                            var error = new Error('Loading
chunk ' + chunkId + ' failed.\n(' + errorType + ': ' +
realSrc + ')');
                            error.type = errorType;
                            error.request = realSrc;
                            chunk[1](error)
                        }
                        installedChunks[chunkId] =
undefined
                    }
                };
                var timeout = setTimeout(function() {
                    onScriptComplete({
                        type: 'timeout',
                        target: script
                    })
                }, 120000);
                script.onerror = script.onload =
onScriptComplete;head
                // 通过 DOM 操作，往 HTML head 中插入一个
```

```
script 标签去异步加载 Chunk 对应的 JavaScript 文件
                document.head.appendChild(script)
            }
        }
        return Promise.all(promises)
    };

    __webpack_require__.m = modules;
    __webpack_require__.c = installedModules;
    __webpack_require__.d = function(exports, name, getter)
{
        if (!__webpack_require__.o(exports, name)) {
            Object.defineProperty(exports, name, {
                enumerable: true,
                get: getter
            })
        }
    };

    __webpack_require__.r = function(exports) {
        if (typeof Symbol !== 'undefined' &&
Symbol.toStringTag) {
            Object.defineProperty(exports,
Symbol.toStringTag, {
                value: 'Module'
            })
        }
        Object.defineProperty(exports, '__esModule', {
            value: true
        })
    };

    __webpack_require__.t = function(value, mode) {
        if (mode & 1) value = __webpack_require__(value);
        if (mode & 8) return value;
        if ((mode & 4) && typeof value === 'object' &&
value && value.__esModule) return value;
```

```
        var ns = Object.create(null);
        __webpack_require__.r(ns);
        Object.defineProperty(ns, 'default', {
            enumerable: true,
            value: value
        });
        if (mode & 2 && typeof value != 'string') for (var
key in value) __webpack_require__.d(ns, key, function(key)
{
            return value[key]
        }.bind(null, key));
        return ns
    };

    __webpack_require__.n = function(module) {
        var getter = module && module.__esModule ?
        function getDefault() {
            return module['default']
        } : function getModuleExports() {
            return module
    };
    __webpack_require__.d(getter, 'a', getter);
        return getter
    };
    __webpack_require__.o = function(object, property) {
        return Object.prototype.hasOwnProperty.call(object,
property)
    };
    __webpack_require__.p = "";
    __webpack_require__.oe = function(err) {
        console.error(err);
        throw err;
    };

    var jsonpArray = window["webpackJsonp"] =
window["webpackJsonp"] || [];
    var oldJsonpFunction =
```

```
jsonpArray.push.bind(jsonpArray);
   jsonpArray.push = webpackJsonpCallback;
   jsonpArray = jsonpArray.slice();
   for (var i = 0; i < jsonpArray.length; i++)
webpackJsonpCallback(jsonpArray[i]);

   var parentJsonpFunction = oldJsonpFunction;
   return __webpack_require__(__webpack_require__.s =
"./src/index.js")
   })({
       // 所有没有经过异步加载的，随着执行入口文件加载的模块
       "./src/index.js": (function(module, exports,
__webpack_require__) {
       eval("// var sayHello = require('./hello')\n//
console.log(sayHello('lucas'))\n// import sayHello from
'./hello.js'\n// console.log(sayHello('lucas'))\nnew
Promise(function (resolve) {\n  __webpack_require__.e(/*!
require.ensure */ 0).then((function (require) {\n
   resolve(__webpack_require__(/*! ./hello */
\"./src/hello.js\"));\n  }).bind(null,
__webpack_require__)).catch(__webpack_require__.oe);\n}).t
hen(function (sayHello) {\n
 console.log(sayHello('lucas'));\n});\n\n//#
sourceURL=webpack:///./src/index.js?")
   })
});
```

按需加载相比常规打包产出结果变化较大，也更加复杂。我们仔细对比其中差异，发现 main.js：

多了一个 __webpack_require__.e

多了一个 webpackJsonp

其中 __webpack_require__.e 实现非常重要，它初始化了一个 promise 数组，使用 Promise.all() 进行异步插入 script 脚本；webpackJsonp 会挂在到全

局对象 window 上，进行模块安装。

熟悉 webpack 的读者可能会知道 CommonsChunkPlugin 插件（在 webpack v4 版本中已经被取代），这个插件用来分割第三方依赖或者公共库的代码，将业务逻辑和稳定的库脚本分离，以达到优化代码体积、合理使用缓存的目的。实际上，这样的思路和上述「按需加载」不谋而合，具体实现思路也一致。我们可以推测开发者在使用 CommonsChunkPlugin 插件打包后的代码结果和上面的代码结构类似，都存在 **webpack_require**.e 和 webpackJsonp。**因为提取公共代码和异步加载本质上都是前置进行代码分割，再在必要时加载，具体实现可以观察 webpack_require.e 和 webpackJsonp**。

到此，我们分析了业务中几乎所有的打包方式以及 webpack 产出结果。虽然这些内容较为晦涩，源码冗长而难以阅读，但是这对我们理解 webpack 内部工作原理，编写 loader、plugin 意义重大。只有分析过所有这些最基本的编译后代码，我们才能对上线代码的质量做到「心里有底」。在出现问题时，能够驾轻就熟，独当一面。这也是高级 Web 工程师所必备的素养。

如果读者在阅读 webpack 打包后代码存在一些困难，也没有关系，细节实现相对打包思想设计并没有那么重要。也许你试着去设计一个模块系统，了解一下 require.js 或者 sea.js 的实现，这些内容也就不再「那么高深」了。这些代码实现细节可以放在一边，通过后续章节的学习之后，再返回来看，可能效果更好。

点击查看下一节

webpack 工程师 > 前端工程师（下）