有点难的知识点： Webpack Chunk 分包规则详解

原创 范文杰 Tecvan 2021-05-13 17:30

收录于合集
#前端工程化 15 #Webpack 18 #前端 17 #JavaScript 7

全文 2500 字，阅读时长约 30 分钟。如果觉得文章有用，欢迎点赞关注，但写作实属不易，未经作者同意，禁止任何形式转载！！！

Tecvan
All or nothing, now or never 👉
48篇原创内容

公众号

背景

在前面系列文章提到，webpack 实现中，原始的资源模块以 `Module` 对象形式存在、流转、解析处理。

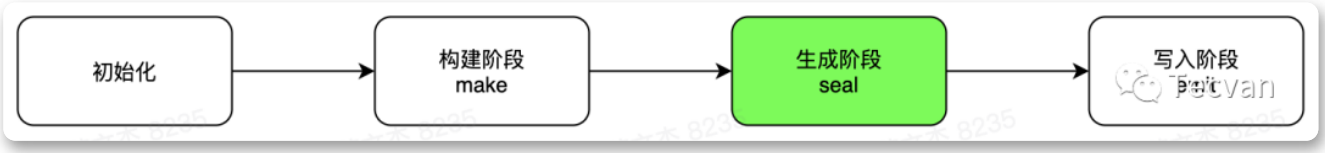
而 `Chunk` 则是输出产物的基本组织单位，在生成阶段 webpack 按规则将 `entry` 及其它 `Module` 插入 `Chunk` 中，之后再由 `SplitChunksPlugin` 插件根据优化规则与 `ChunkGraph` 对 `Chunk` 做一系列的变化、拆解、合并操作，重新组织成一批性能(可能)更高的 `Chunks` 。运行完毕之后 webpack 继续将 `chunk` 一一写入物理文件中，完成编译工作。

综上， `Module` 主要作用在 webpack 编译过程的前半段，解决原始资源“**「如何读」**”的问题；而 Chunk 对象则主要作用在编译的后半段，解决编译产物“**「如何写」**”的问题，两者合作搭建起 webpack 搭建主流程。

Chunk 的编排规则非常复杂，涉及 entry、optimization 等诸多配置项，我打算分成两篇文章分别讲解基本分包规则、 `SplitChunksPlugin` 分包优化规则，本文将集中在第一部分，讲解 entry、异步模块、runtime 三条规则的细节与原理。

默认分包规则

Webpack 4 之后编译过程大致上可以拆解为四个阶段(参考：[万字总结] 一文吃透 Webpack 核心原理)：



在构建(make) 阶段，webpack 从 entry 出发根据模块间的引用关系(require/import) 逐步构建出模块依赖关系图(ModuleDependencyGraph)，依赖关系图表达了模块与模块之间互相引用的先后次序，基于这种次序 webpack 就可以推断出模块运行之前需要先执行那些依赖模块，也就可以进一步推断出那些模块应该打包在一起，那些模块可以延后加载(异步执行)，关于模块依赖图的更多信息，可以参考我另一篇文章 《有点难的 webpack 知识点：Dependency Graph 深度解析》。

到了生成(seal) 阶段，webpack 会根据模块依赖图的内容组织分包 —— Chunk 对象，默认的分包规则有：

- 同一个 `entry` 下触达到的模块组织成一个 chunk
- 异步模块单独组织为一个 chunk
- `entry.runtime` 单独组织成一个 chunk

默认规则集中在 `compilation.seal` 函数实现，seal 核心逻辑运行结束后会生成一系列的 `Chunk`、`ChunkGroup`、`ChunkGraph` 对象，后续如 `SplitChunksPlugin` 插件会在 `Chunk` 系列对象上做进一步的拆解、优化，最终反映到输出上才会表现出复杂的分包结果。

我们聊聊默认生成规则。

Entry 分包处理

> 重点：seal 阶段遍历 entry 对象，为每一个 entry 单独生成 chunk，之后再根据模块依赖图将 entry 触达到的所有模块打包进 chunk 中。

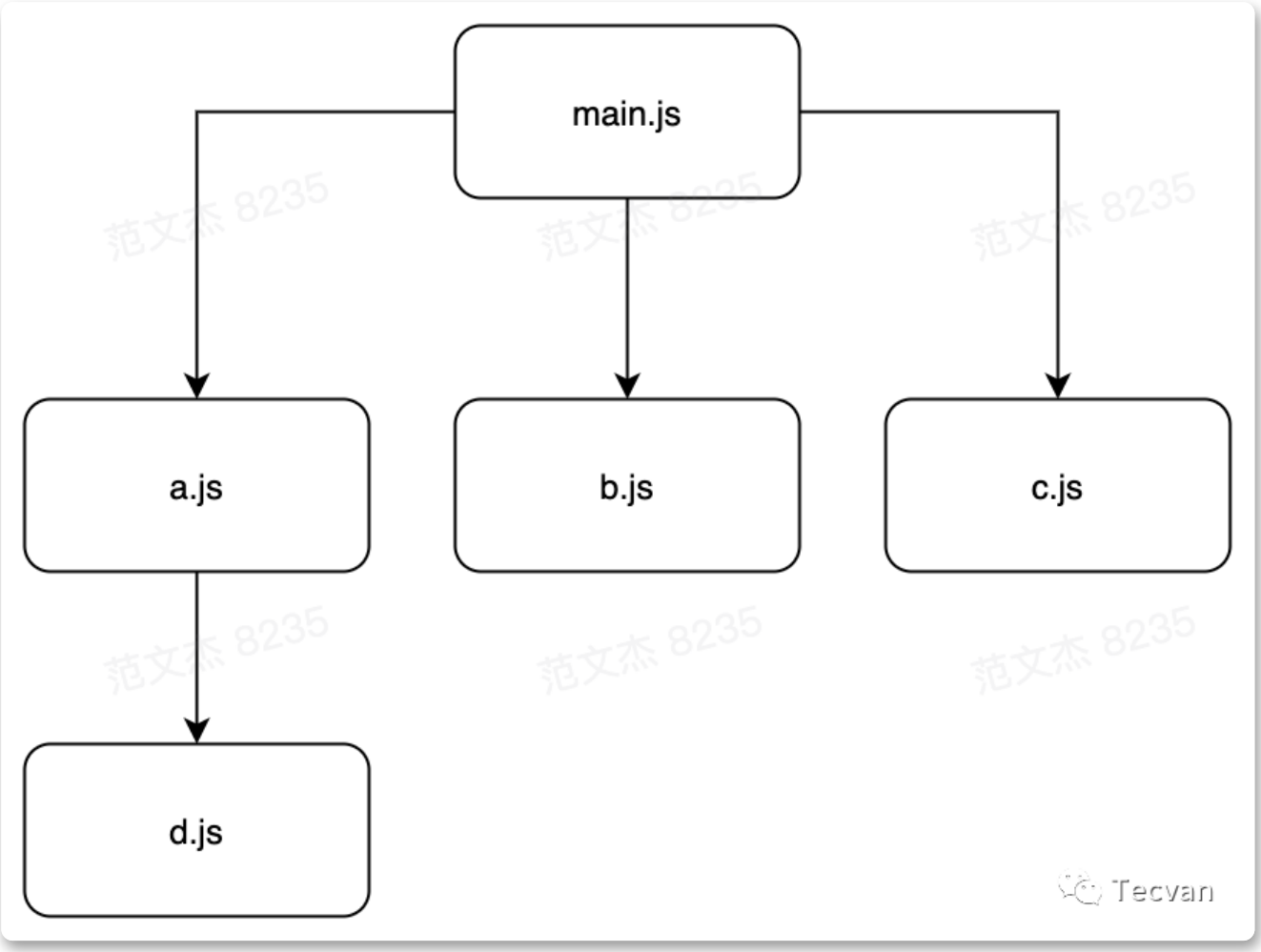
在生成阶段，Webpack 首先根据遍历用户提供的 entry 属性值，为每一个 entry 创建 Chunk 对象，比如对于如下配置：

```
module.exports = {
  entry: {
    main: "./src/main",
```

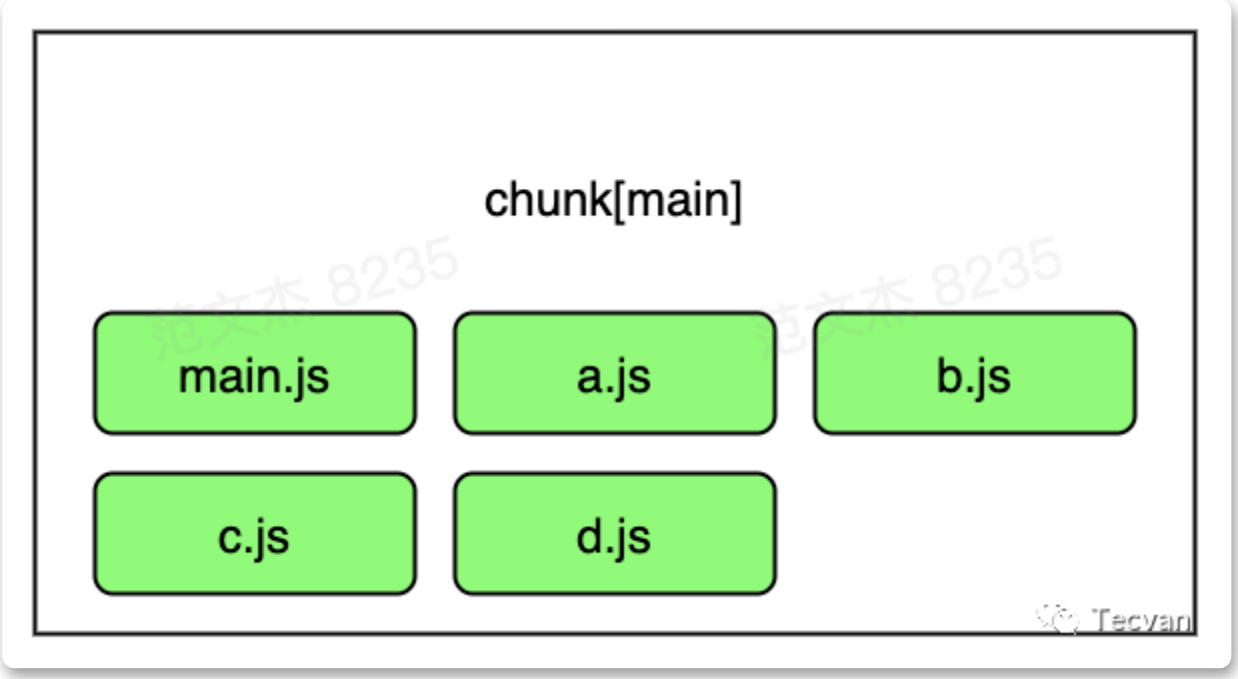
```
    home: "./src/home",
  }
};
```

Webpack 遍历 entry 对象属性并创建出 `chunk[main]` 、 `chunk[home]` 两个对象，此时两个 chunk 分别包含 `main` 、 `home` 模块：

初始化完毕后，Webpack 会读取 `ModuleDependencyGraph` 的内容，将 entry 所对应的内容塞入对应的 chunk（发生在 `webpack/lib/buildChunkGrap.js` 文件）。比如对于如下文件依赖：



main.js 以同步方式直接或间接引用了 a/b/c/d 四个文件，分析 ModuleDependencyGraph 过程会逐步将 a/b/c/d 模块逐步添加到 chunk[main] 中，最终形成：



PS：基于动态加载生成的 chunk 在 webpack 官方文档中，通常称之为 「Initial chunk」 。

异步模块分包处理

> 重点：分析 ModuleDependencyGraph 时，每次遇到异步模块都会为之创建单独的 Chunk 对象，单独打包异步模块。

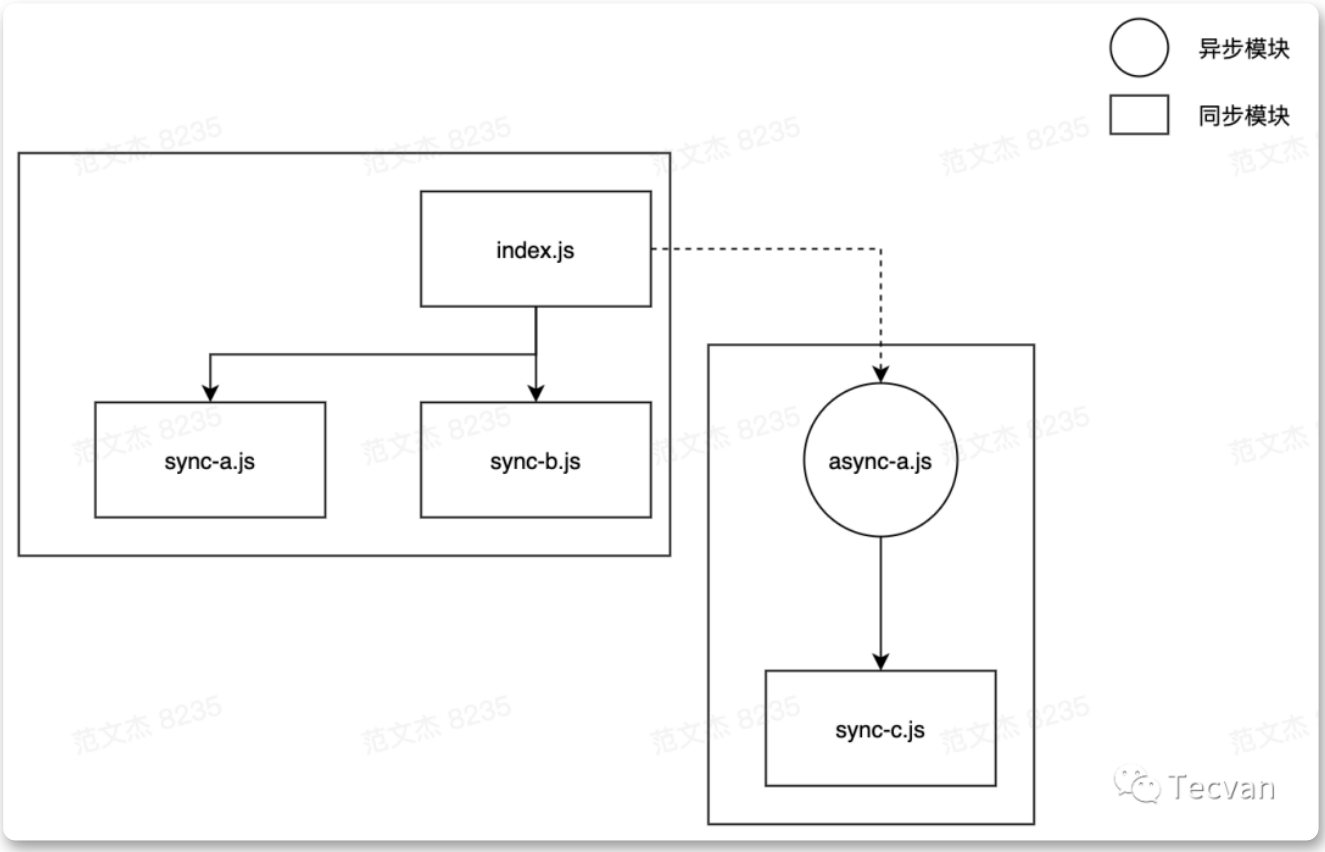
Webpack 4 之后，只需要用异步语句 require.ensure("./xx.js") 或 import("./xx.js") 方式引入模块，就可以实现模块的动态加载，这种能力本质也是基于 Chunk 实现的。

Webpack 生成阶段中，遇到异步引入语句时会为该模块单独生成一个 chunk 对象，并将其子模块都加入这个 chunk 中。例如对于下面的例子：

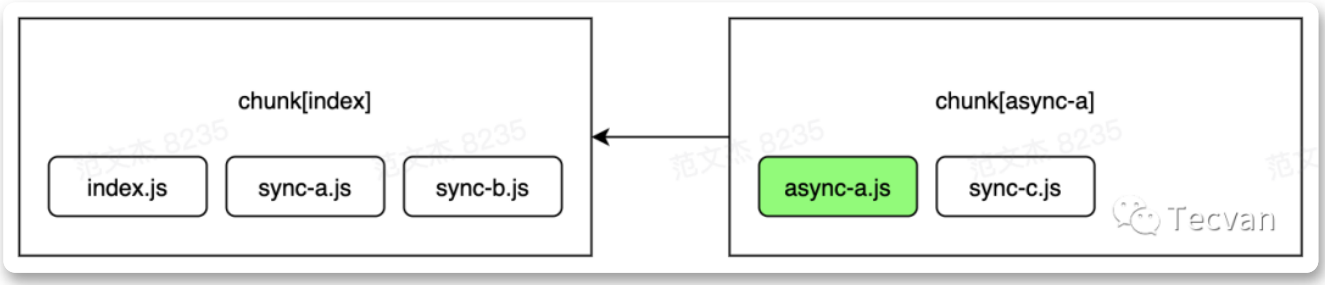
```
// index.js, entry 文件
import 'sync-a'
import 'sync-b'

import('async-c')
```

在 index.js 中，以同步方式引入 sync-a 、 sync-b ；以异步方式引入 async-a 模块；同时，在 async-a 中以同步方式引入 sync-c 模块。对应的模块依赖如：



此时，webpack 会为入口 index.js 、异步模块 async-a.js 分别创建分包，形成如下数据：



这里需要引入一个新的概念 —— Chunk 间的父子关系。由 entry 生成的 Chunk 之间相互孤立，没有必然的前后依赖关系，但异步生成的 Chunk 则不同，引用者(上例 index.js 块)需要在特定场景下使用被引用者(上例 async-a 块)，两者间存在单向依赖关系，在 webpack 中称引用者为 parent、被引用者为 child，分别存放在 ChunkGroup._parents 、 ChunkGroup._children 属性中。

上述分包方案默认情况下会生成两个文件：

- 入口 `index` 对应的 `index.js`
- 异步模块 `async-a` 对应的 `src_async-a_js.js`

运行时，webpack 在 index.js 中使用 promise 及 `__webpack_require__.e` 方法异步载入并运行文件 `src_async-a_js.js` ，从而实现动态加载。

PS: 基于异步模块的 chunk 在 webpack 官方文档中，通常称之为 「Async chunk」。

Runtime 分包

> 重点：Webpack 5 之后还能根据 `entry.runtime` 配置单独打包运行时代码。

除了 entry、异步模块外，webpack 5之后还支持基于 `runtime` 的分包规则。除业务代码外，Webpack 编译产物中还需要包含一些用于支持 webpack 模块化、异步加载等特性的支撑性代码，这类代码在 webpack 中被统称为 `runtime`。举个例子，产物中通常会包含如下代码：

```
/******/ (() => {
  // webpackBootstrap
  /******/ var __webpack_modules__ = {}; // The module cache
  /************************************************************************/
  /******/ /******/ var __webpack_module_cache__ = {}; // The require function
  /******/
  /******/ /******/ function __webpack_require__(moduleId) {
    /******/ /******/ __webpack_modules__[moduleId](
      module,
      module.exports,
      __webpack_require__
    ); // Return the exports of the module
    /******/
    /******/ /******/ return module.exports;
    /******/
  } // expose the modules object (__webpack_modules__)
  /******/
  /******/ /******/ __webpack_require__.m = __webpack_modules__; /* webpack/runtime/compat get de
```

```
  /******/

  // ...
})();
```

编译时，Webpack 会根据业务代码决定输出那些支撑特性的运行时代码(基于 Dependency 子类)，例如：

- 需要 __webpack_require__.f 、__webpack_require__.r 等功能实现最起码的模块化支持
- 如果用到动态加载特性，则需要写入 __webpack_require__.e 函数
- 如果用到 Module Federation 特性，则需要写入 __webpack_require__.o 函数
- 等等

虽然每段运行时代码可能都很小，但随着特性的增加，最终结果会越来越大，特别对于多 entry 应用，在每个入口都重复打包一份相似的运行时代码显得有点浪费，为此 webpack 5 专门提供了 entry.runtime 配置项用于声明如何打包运行时代码。用法上只需在 entry 项中增加字符串形式的 runtime 值，例如：

```
module.exports = {
  entry: {
    index: { import: "./src/index", runtime: "solid-runtime" },
  }
};
```

Webpack 执行完 entry 、异步模块分包后，开始遍历 entry 配置判断是否带有 runtime 属性，如果有则创建以 runtime 值为名的 Chunk，因此，上例配置将生成两个 chunk：chunk[index.js] 、chunk[solid-runtime]，并据此最终产出两个文件：

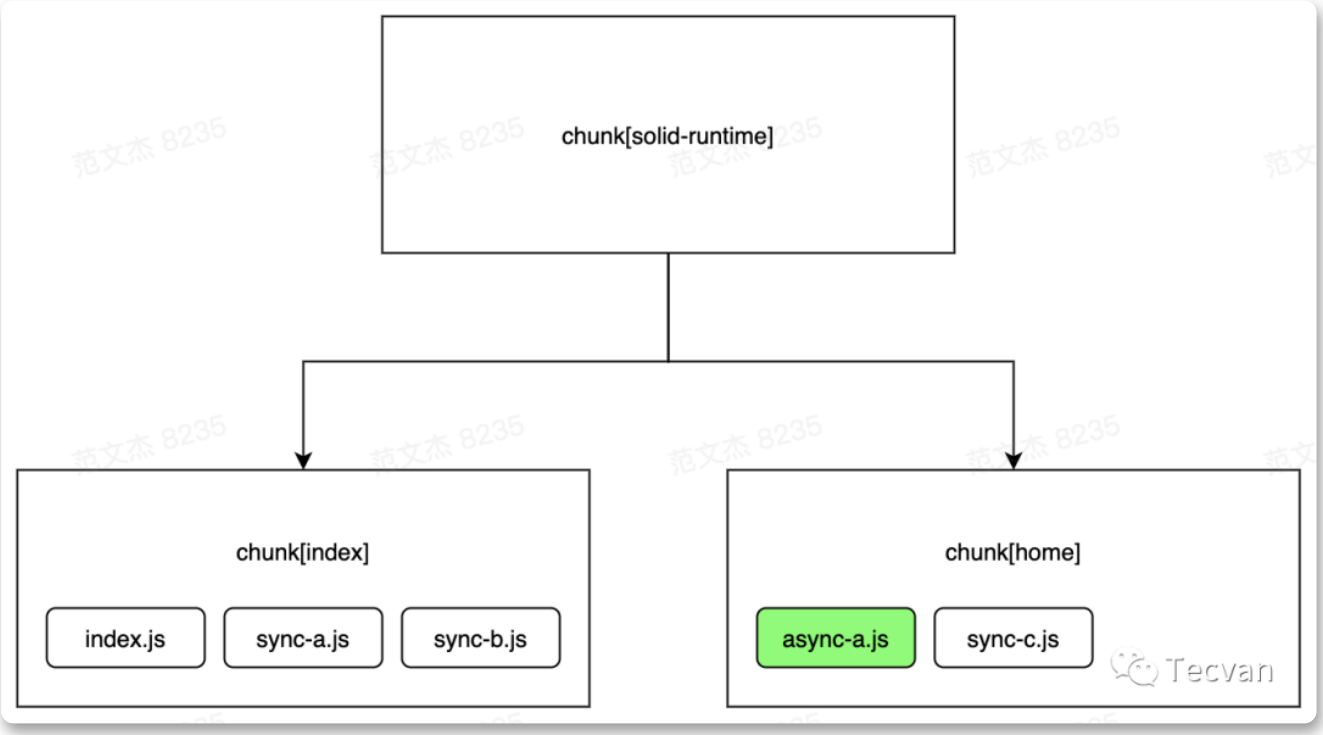
- 入口 index 对应的 index.js 文件
- 运行时配置对应的 solid-runtime.js 文件

在多 entry 场景中，只要为每个 entry 都设定相同的 runtime 值，webpack 运行时代码最终就会集中写入到同一个 chunk，例如对于如下配置：

```
module.exports = {
  entry: {
    index: { import: "./src/index", runtime: "solid-runtime" },
    home: { import: "./src/home", runtime: "solid-runtime" },
```

```
  }
};
```

入口 index、home 共享相同的 `runtime` ，最终生成三个 `chunk` ，分别为：



同时生成三个文件：

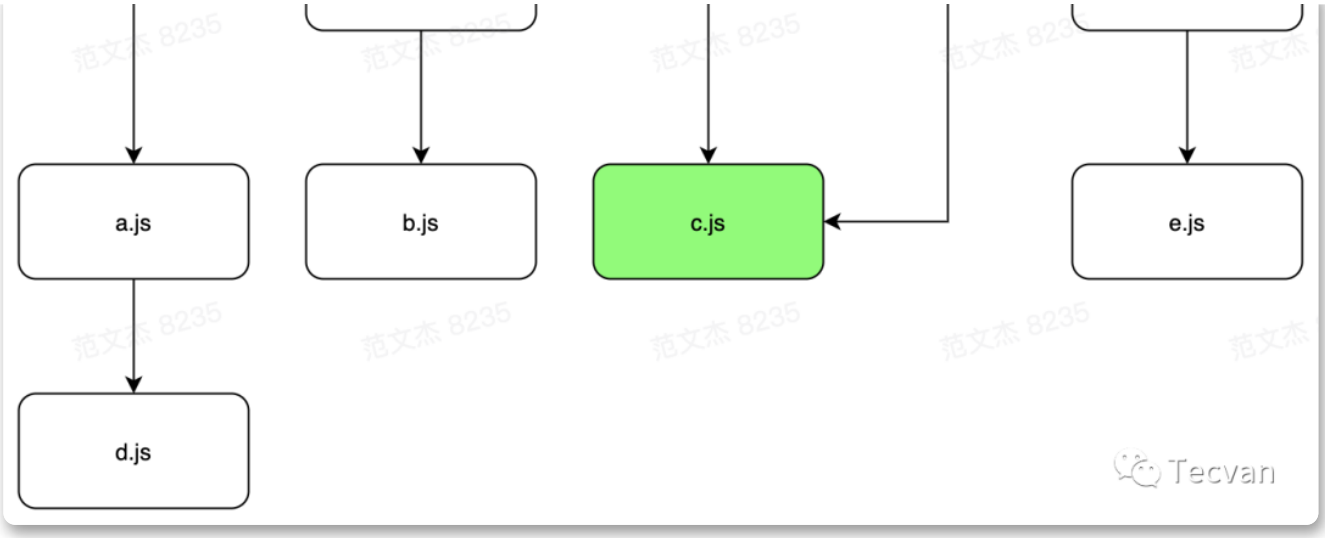
- 入口 index 对应的 `index.js`
- 入口 index 对应的 `home.js`
- 运行时代码对应的 `solid-runtime.js`

分包规则的问题

至此，webpack 分包规则的基本逻辑就介绍完毕了，实现上，大部分功能代码都集中在：

- `webpack/lib/compilation.js` 文件的 `seal` 函数
- `webpack/lib/buildChunkGraph.js` 的 `buildChunkGraph` 函数

默认分包规则最大的问题是无法解决模块重复，如果多个 chunk 同时包含同一个 `module` ，那么这个 `module` 会被不受限制地重复打包进这些 chunk。比如假设我们有两个入口 `main/index` 同时依赖了同一个模块：



默认情况下，webpack 不会对此做额外处理，只是单纯地将 c 模块同时打包进 `main/index` 两个 chunk，最终形成：

可以看到 chunk 间互相孤立，模块 c 被重复打包，对最终产物可能造成不必要的性能损耗！

为了解决这个问题，webpack 3 引入 `CommonChunkPlugin` 插件试图将 entry 之间的公共依赖提取成单独的 chunk，但 `CommonChunkPlugin` 本质上是基于 Chunk 之间简单的父子关系链实现的，很难推断出提取出的第三个包应该作为 entry 的父 chunk 还是子 chunk，`CommonChunkPlugin` 统一处理为父 chunk，某些情况下反而对性能造成了不小的负面影响。

在 webpack 4 之后则引入了更负责的设计 —— `ChunkGroup` 专门实现关系链管理，配合 `SplitChunksPlugin` 能够更高效、智能地实现**「启发式分包」**，这里的内容很复杂，我打算拆开来在下一篇文章再讲，感兴趣的同学记得关注。

下节预告

后面我还会继续 focus 在 chunk 相关功能与核心实现原理，内容包括：

- webpack 4 之后引入 `ChunkGroup` 的引入解决了什么问题，为什么能极大优化分包功能
- webpack 5 引入的 `ChunkGraph` 解决了什么问题
- Chunk、ChunkGroup、ChunkGraph 分别实现什么能力，互相之间如何协作，为什么要做这样的拆分
- SplitChunksPlugin 插件做了那些分包优化，以及我们可以从中学到什么插件开发技巧
- 站在应用、性能的角度，有那些分包最佳实践

感兴趣的同学一定要记得点赞关注，您的反馈将是我持续创作的巨大动力！

Tecvan

All or nothing, now or never 👉

48篇原创内容

公众号

往期文章：

- [万字总结] 一文吃透 Webpack 核心原理
- [源码解读] Webpack 插件架构深度讲解
- 十分钟精进 Webpack：module.issuer 属性详解
- 分享几个 Webpack 实用分析工具
- 有点难的 webpack 知识点：Dependency Graph 深度解析

收录于合集 #Webpack 18

上一篇
Webpack 原理系列六： 彻底理解 Webpack 运行时

下一篇
有点难的 webpack 知识点：Dependency Graph 深度解析

文章已于2021-05-13修改

阅读原文

喜欢此内容的人还喜欢

[科普] JS中Object的keys是无序的吗

Tecvan