代码分割：打包完产物体积太大，怎么拆包？

发布于 2022-05-09

在生产环境下，为了提高页面加载性能，构建工具一般将项目的代码打包(bundle)到一起，这样上线之后只需要请求少量的 JS 文件，大大减少 HTTP 请求。当然，Vite 也不例外，默认情况下 Vite 利用底层打包引擎 Rollup 来完成项目的模块打包。

某种意义上来说，对线上环境进行项目打包是一个必须的操作。但随着前端工程的日渐复杂，单份的打包产物体积越来越庞大，会出现一系列应用加载性能问题，而代码分割可以很好地解决它们。

在本小节中，我们将围绕 代码分割 展开学习。首先我们将一起分析 `Code Splitting` 解决了单产物打包模式下的哪些问题，然后用具体的项目示例体验一下 Vite 默认自带的 `Code Splitting` 效果。从中，你将了解到 Vite 的默认分包策略，以及底层所使用的 Rollup 拆包 API—— `munaulChunks` 。

当然，在实际的项目场景中，只用 Vite 默认的策略是不够的，我们会更深入一步，学习 Rollup 底层拆包的各种高级姿势，实现**自定义拆包**，同时我也会带大家通过实际案例复现 Rollup 自定义拆包经常遇到的坑—— 循环引用 问题，一起分析问题出现的原因，也分享一些自己的解决思路和方案，让大家对 Vite 及 Rollup 的代码分割有更加深入的掌握。

需要注意的是，文中会多次提到 `bundle` 、 `chunk` 、 `vendor` 这些构建领域的专业概念，这里给大家提前解释一下:

- `bundle` 指的是整体的打包产物，包含 JS 和各种静态资源。
- `chunk` 指的是打包后的 JS 文件，是 `bundle` 的子集。
- `vendor` 是指第三方包的打包产物，是一种特殊的 chunk。

Code Splitting 解决的问题

在传统的单 chunk 打包模式下，当项目代码越来越庞大，最后会导致浏览器下载一个巨大的文件，从页面加载性能的角度来说，主要会导致两个问题:

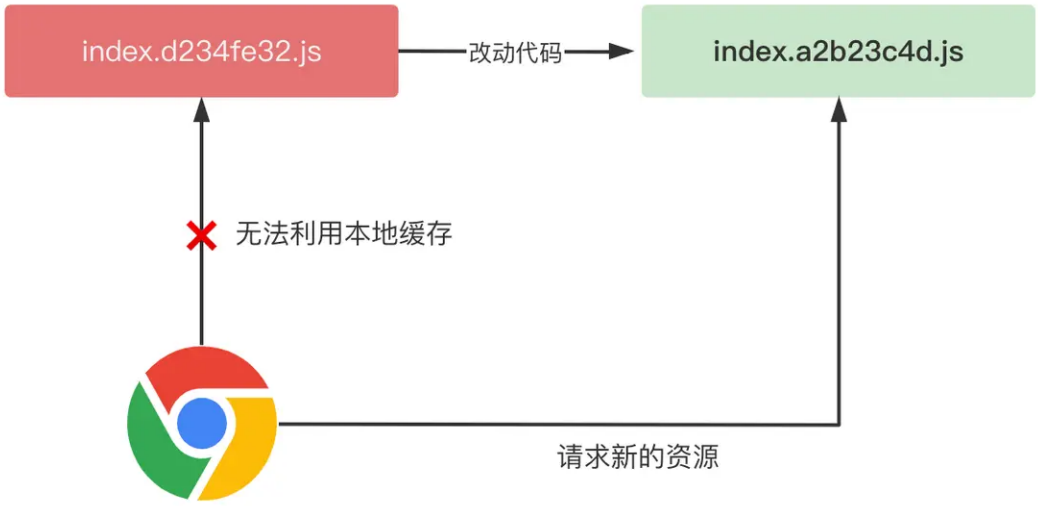
- 无法做到**按需加载**，即使是当前页面不需要的代码也会进行加载。
- 线上**缓存复用率**极低，改动一行代码即可导致整个 bundle 产物缓存失效。

首先说第一个问题，一般而言，一个前端页面中的 JS 代码可以分为两个部分: `Initital Chunk` 和 `Async Chunk` ，前者指页面首屏所需要的 JS 代码，而后者当前页面并不一定需要，一个典型的例子就是 `路由组件` ，与当前路由无关的组件并不用加载。而项目被打包成单 bundle 之后，无论是 `Initial Chunk` 还是 `Async Chunk` ，都会打包进同一个产物，也就是说，浏览器加载产物代码的时候，会将两者一起加载，导致许多冗余的加载过程，从而影响页面性能。而通过 `Code Splitting` 我们可以将按需加载的代码拆分出单独的 chunk，这样应用在首屏加载时只需要加载 `Initial Chunk` 即可，避免了冗余的加载过程，使页面性能得到提升。

其次，线上的 `缓存命中率` 是一个重要的性能衡量标准。对于线上站点而言，服务端一般在响应资源时加上一些 HTTP 响应头，最常见的响应头之一就是 `cache-control` ，它可以指定浏览器的**强缓存**，比如设置为下面这样:

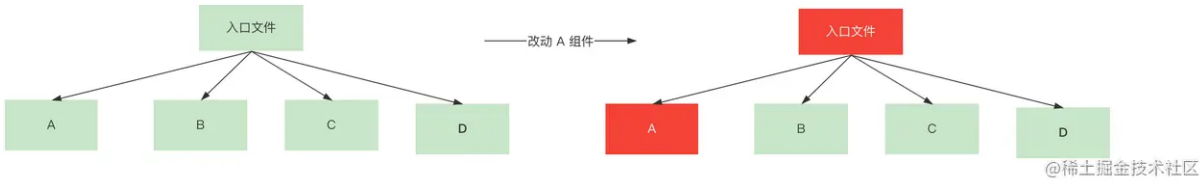
```
cache-control: max-age=31536000
```

表示资源过期时间为一年，在过期之前，访问**相同的资源 url**，浏览器直接利用本地的缓存，并不用给服务端发请求，这就大大降低了页面加载的网络开销。不过，在单 chunk 打包模式下面，一旦有一行代码变动，整个 chunk 的 url 地址都会变化，比如下图所示的场景:



由于构建工具一般会根据产物的内容生成哈希值，一旦内容变化就会导致整个 chunk 产物的强缓存失效，所以单 chunk 打包模式下的缓存命中率极低，基本为零。

而进行 `Code Splitting` 之后，代码的改动只会影响部分的 chunk 哈希改动，如下图所示:



入口文件引用了 `A` 、 `B` 、 `C` 、 `D` 四个组件，当我们修改 A 的代码后，变动的 Chunk 就只有 `A` 以及 `依赖 A 的 Chunk` 中，A 对应的 chunk 会变动，这很好理解，后者也会变动是因为相应的引入语句会变化，如这里的入口文件会发生如下内容变动:

```
import CompA from './A.d3e2f17a.js'
// 更新 import 语句
import CompA from './A.a5d2f82b.js'
```

也就是说，在改动 `A` 的代码后， `B` 、 `C` 、 `D` 的 chunk 产物 url 并没有发生变化，从而可以让浏览器复用本地的强缓存，大大提升线上应用的加载性能。

Vite 默认拆包策略

刚刚我们说到了为什么要进行拆包，实际上 Vite 中已经内置了一份拆包的策略，接下来让我们来看看 Vite 默认的拆包模式是怎样的。

在生产环境下 Vite 完全利用 Rollup 进行构建，因此拆包也是基于 Rollup 来完成的，但 Rollup 本身是一个专注 JS 库打包的工具，对应用构建的能力还尚为欠缺，Vite 正好是补足了 Rollup 应用构建的能力，在拆包能力这一块的扩展就是很好的体现。

我们先通过具体的项目来体验一下 Vite 拆包，示例项目我已经放到了小册的 Gihub 仓库中，你可以对照着来学习。

在项目中执行 `npm run build` ，接着终端会出现如下的构建信息:

> 项目示例使用的是 Vite 2.9 之前的版本，点击进入项目。Vite 2.9 及以后的版本拆包策略会有所不同，后文会介绍。

这里我来解释一下产物的结构:

```
.
├── assets
│   ├── Dynamic.3df51f7a.js    // Async Chunk
│   ├── Dynamic.f2cbf023.css   // Async Chunk (CSS)
│   ├── favicon.17e50649.svg   // 静态资源
│   ├── index.1e236845.css     // Initial Chunk (CSS)
│   ├── index.6773c114.js      // Initial Chunk
│   └── vendor.ab4b9e1f.js     // 第三方包产物 Chunk
└── index.html                 // 入口 HTML
```

对于 Vite 的拆包能力，从产物结构中就可见一斑。

一方面 Vite 实现了自动 **CSS 代码分割**的能力，即实现一个 chunk 对应一个 css 文件，比如上面产物中 `index.js` 对应一份 `index.css` ，而按需加载的 chunk `Danamic.js` 也对

应单独的一份 `Danamic.css` 文件，与 JS 文件的代码分割同理，这样做也能提升 CSS 文件的缓存复用率。

而另一方面， Vite 基于 Rollup 的 `manualChunks` API 实现了 应用拆包 的策略:

- 对于 `Initital Chunk` 而言，业务代码和第三方包代码分别打包为单独的 chunk，在上述的例子中分别对应 `index.js` 和 `vendor.js` 。需要说明的是，这是 Vite 2.9 版本之前的做法，而在 Vite 2.9 及以后的版本，默认打包策略更加简单粗暴，将所有的 js 代码全部打包到 `index.js` 中。

- 对于 `Async Chunk` 而言 ， 动态 import 的代码会被拆分成单独的 chunk，如上述的 `Dynacmic` 组件。

小结一下，Vite 默认拆包的优势在于实现了 CSS 代码分割与业务代码、第三方库代码、动态 import 模块代码三者的分离，但缺点也比较直观，第三方库的打包产物容易变得比较臃肿，上述例子中的 `vendor.js` 的大小已经达到 500 KB 以上，显然是有进一步拆包的优化空间的，这个时候我们就需要用到 Rollup 中的拆包 API —— `manualChunks` 了。

自定义拆包策略

针对更细粒度的拆包，Vite 的底层打包引擎 Rollup 提供了 `manualChunks` ，让我们能自定义拆包策略，它属于 Vite 配置的一部分，示例如下:

```
// vite.config.ts
export default {
  build: {
    rollupOptions: {
      output: {
        // manualChunks 配置
        manualChunks: {},
      },
    }
  },
}
```

`manualChunks` 主要有两种配置的形式，可以配置为一个对象或者一个函数。我们先来看看对象的配置，也是最简单的配置方式，你可以在上述的示例项目中添加如下的 `manualChunks` 配置代码:

```
// vite.config.ts
{
  build: {
    rollupOptions: {
      output: {
        // manualChunks 配置
        manualChunks: {
          // 将 React 相关库打包成单独的 chunk 中
          'react-vendor': ['react', 'react-dom'],
          // 将 Lodash 库的代码单独打包
          'lodash': ['lodash-es'],
          // 将组件库的代码打包
          'library': ['antd', '@arco-design/web-react'],
        },
      },
    }
  },
}
```

在对象格式的配置中， `key` 代表 chunk 的名称， `value` 为一个字符串数组，每一项为第三方包的包名。在进行了如上的配置之后，我们可以执行 `npm run build` 尝试一下打包:

```
> vite-code-splitting@0.0.0 build
> tsc && vite build

vite v2.8.4 building for production...
✓ 2367 modules transformed.
dist/assets/favicon.17e50649.svg        1.49 KiB
dist/index.html                         0.65 KiB
dist/assets/index.0db21541.js           25.60 KiB / gzip: 7.13 KiB
dist/assets/Dynamic.38b4ea51.js         0.50 KiB / gzip: 0.32 KiB
dist/assets/index.1e236845.css          47.92 KiB / gzip: 4.00 KiB
dist/assets/Dynamic.d0005171.css        0.03 KiB / gzip: 0.05 KiB
dist/assets/lodash.0b61700c.js          199.99 KiB / gzip: 43.89 KiB
dist/assets/react-vendor.2fa3f57c.js    195.36 KiB / gzip: 48.57 KiB
dist/assets/library.3e3496eb.js         269.80 KiB / gzip: 57.47 KiB
```


你可以看到原来的 vendor 大文件被拆分成了我们手动指定的几个小 chunk，每个 chunk 大概 200 KB 左右，是一个比较理想的 chunk 体积。这样，当第三方包更新的时候，也只会更新其中一个 chunk 的 url，而不会全量更新，从而提高了第三方包产物的缓存命中率。

除了对象的配置方式之外，我们还可以通过函数进行更加灵活的配置，而 Vite 中的默认拆包策略也是通过函数的方式来进行配置的，我们可以在 Vite 的实现中瞧一瞧:

```
// Vite 部分源码
function createMoveToVendorChunkFn(config: ResolvedConfig): GetManualChunk {
  const cache = new Map<string, boolean>()
  // 返回值为 manualChunks 的配置
  return (id, { getModuleInfo }) => {
    // Vite 默认的配置逻辑其实很简单
    // 主要是为了把 Initial Chunk 中的第三方包代码单独打包成`vendor.[hash].js`
```

```
    if (
      id.includes('node_modules') &&
      !isCSSRequest(id) &&
      // 判断是否为 Initial Chunk
      staticImportedByEntry(id, getModuleInfo, cache)
    ) {
      return 'vendor'
    }
  }
}
```

Rollup 会对每一个模块调用 manualChunks 函数，在 manualChunks 的函数入参中你可以拿到 模块 id 及 模块详情信息，经过一定的处理后返回 chunk 文件的名称，这样当前 id 代表的模块便会打包到你所指定的 chunk 文件中。

我们现在来试着把刚才的拆包逻辑用函数来实现一遍:

```
manualChunks(id) {
  if (id.includes('antd') || id.includes('@arco-design/web-react')) {
    return 'library';
  }
  if (id.includes('lodash')) {
    return 'lodash';
  }
  if (id.includes('react')) {
    return 'react';
  }
}
```

打包后结果如下:

```
vite v2.8.4 building for production...
✓ 2367 modules transformed.
dist/assets/favicon.17e50649.svg        1.49 KiB
dist/index.html                         0.78 KiB
dist/assets/library.bc4700ad.js         74.86 KiB / gzip: 15.48 KiB
dist/assets/library.23b57768.css        47.56 KiB / gzip: 3.77 KiB
dist/assets/index.fa07d900.css          0.39 KiB / gzip: 0.27 KiB
dist/assets/index.37a7b2eb.js           188.11 KiB / gzip: 42.46 KiB
dist/assets/lodash.0b61700c.js          199.99 KiB / gzip: 43.89 KiB
dist/assets/react-vendor.e2c4883f.js    234.99 KiB / gzip: 56.46 KiB  @稀土掘金技术社区
```

看上去好像各个第三方包的 chunk (如 lodash 、 react 等等)都能拆分出来，但实际上你可以运行 npx vite preview 预览产物，会发现产物根本没有办法运行起来，页面出现白屏，同时控制台出现如下的报错:

```
▸Uncaught TypeError: l$1 is not a function                    react-vendor.e2c4883f.js:77
    at react-vendor.e2c4883f.js:77:1                          @稀土掘金技术社区
```

这也就是函数配置的坑点所在了，虽然灵活而方便，但稍不注意就陷入此类的产物错误问题当中。那上面的这个报错究竟是什么原因导致的呢?

解决循环引用问题

从报错信息追溯到产物中，可以发现 `react-vendor.js` 与 `index.js` 发生了循环引用:

```
// react-vendor.e2c4883f.js
import { q as objectAssign } from "./index.37a7b2eb.js";

// index.37a7b2eb.js
import { R as React } from "./react-vendor.e2c4883f.js";
```

这是很典型的 ES 模块循环引用的场景，我们可以用一个最基本的例子来复原这个场景:

```
// a.js
import { funcB } from './b.js';

funcB();

export var funcA = () => {
  console.log('a');
}
// b.js
import { funcA } from './a.js';

funcA();

export var funcB = () => {
  console.log('b')
}
```

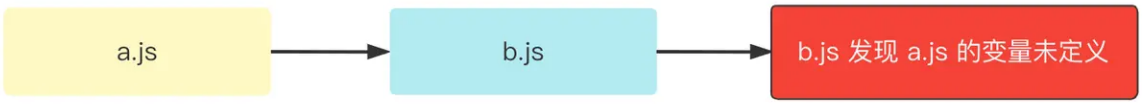
接着我们可以执行一下 `a.js` 文件:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <script type="module" src="/a.js"></script>
</body>
</html>
```

在浏览器中打开会出现类似的报错:

代码的执行原理如下:

- JS 引擎执行 `a.js` 时，发现引入了 `b.js` ，于是去执行 `b.js`
- 引擎执行 `b.js` ，发现里面引入了 `a.js` (出现循环引用)，认为 `a.js` 已经加载完成，继续往下执行
- 执行到 `funcA()` 语句时发现 funcA 并没有定义，于是报错。



而对于如上打包产物的执行过程也是同理:

可能你会有疑问: `react-vendor` 为什么需要引用 `index.js` 的代码呢？其实也很好理解，我们之前在 `munaulChunks` 中仅仅将路径包含 `react` 的模块打包到 `react-vendor` 中，殊不知，像 `object-assign` 这种 react 本身的依赖并没有打包进 `react-vendor` 中，而是打包到另外的 chunk 当中，从而导致如下的循环依赖关系:

那我们能不能避免这种问题呢？当然是可以的，之前的 `manualChunks` 逻辑过于简单粗暴，仅仅通过路径 id 来决定打包到哪个 chunk 中，而漏掉了间接依赖的情况。如果针对像 `object-assign` 这种间接依赖，我们也能识别出它属于 react 的依赖，将其自动打包到 `react-vendor` 中，这样就可以避免循环引用的问题。

我们来梳理一下解决的思路:

- 确定 react 相关包的入口路径。
- 在 manualChunks 中拿到模块的详细信息，向上追溯它的引用者，如果命中 react 的路径，则将模块放到 `react-vendor` 中。

接下来让我们进行实际代码的实现:

```
// 确定 react 相关包的入口路径
const chunkGroups = {
  'react-vendor': [
    require.resolve('react'),
    require.resolve('react-dom')
  ],
}

// Vite 中的 manualChunks 配置
function manualChunks(id, { getModuleInfo }) {
  for (const group of Object.keys(chunkGroups)) {
    const deps = chunkGroups[group];
    if (
      id.includes('node_modules') &&
      // 递归向上查找引用者，检查是否命中 chunkGroups 声明的包
      isDepInclude(id, deps, [], getModuleInfo)
     ) {
      return group;
    }
  }
}
```

实际上核心逻辑包含在 `isDepInclude` 函数，用来递归向上查找引用者模块:

```
// 缓存对象
const cache = new Map();

function isDepInclude (id: string, depPaths: string[], importChain: string[], getModuleInfo):
  const key = `${id}-${depPaths.join('|')}`;
  // 出现循环依赖，不考虑
  if (importChain.includes(id)) {
    cache.set(key, false);
    return false;
  }
  // 验证缓存
  if (cache.has(key)) {
    return cache.get(key);
  }
  // 命中依赖列表
  if (depPaths.includes(id)) {
    // 引用链中的文件都记录到缓存中
    importChain.forEach(item => cache.set(`${item}-${depPaths.join('|')}`, true));
    return true;
  }
  const moduleInfo = getModuleInfo(id);
  if (!moduleInfo || !moduleInfo.importers) {
    cache.set(key, false);
    return false;
  }
  // 核心逻辑，递归查找上层引用者
  const isInclude = moduleInfo.importers.some(
    importer => isDepInclude(importer, depPaths, importChain.concat(id), getModuleInfo)
```

```
  );
  // 设置缓存
  cache.set(key, isInclude);
  return isInclude;
};
```

对于这个函数的实现，有两个地方需要大家注意:

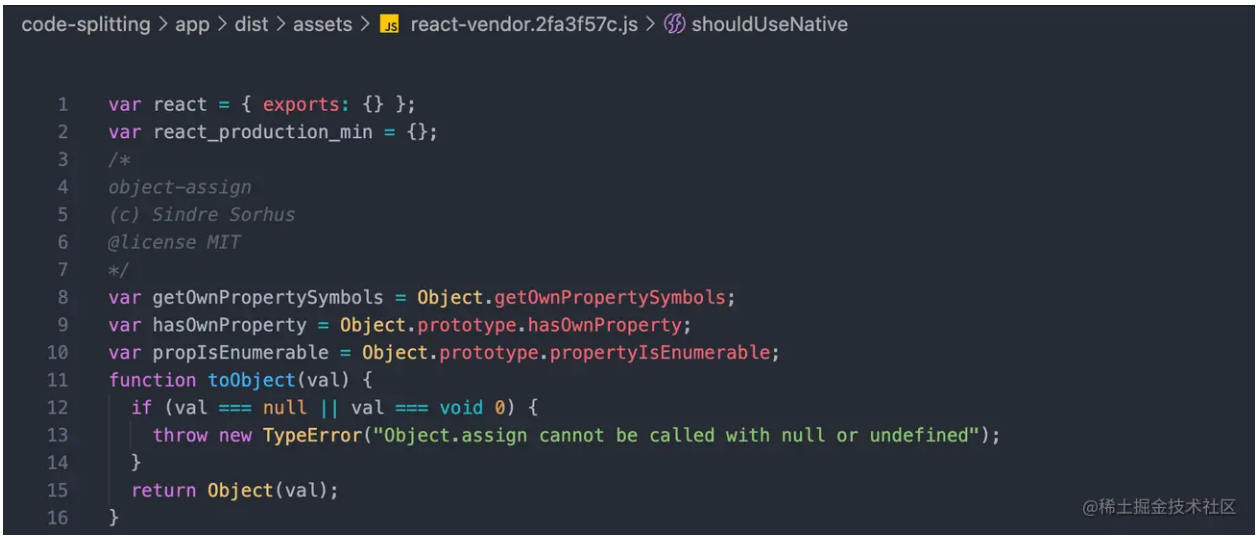
- 我们可以通过 manualChunks 提供的入参 `getModuleInfo` 来获取模块的详情 `moduleInfo` ，然后通过 `moduleInfo.importers` 拿到模块的引用者，针对每个引用者又可以递归地执行这一过程，从而获取引用链的信息。
- 尽量使用缓存。由于第三方包模块数量一般比较多，对每个模块都向上查找一遍引用链会导致开销非常大，并且会产生很多重复的逻辑，使用缓存会极大加速这一过程。

完成上述 `manualChunks` 的完整逻辑后，现在我们来执行 `npm run build` 来进行打包:

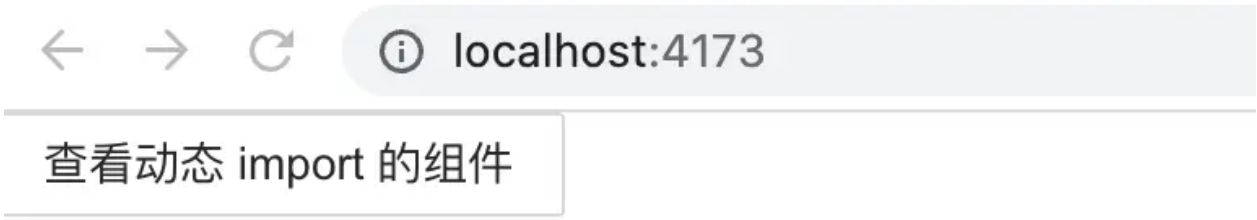
```
> vite-code-splitting@0.0.0 build
> tsc && vite build

vite v2.8.4 building for production...
✓ 2367 modules transformed.
dist/assets/favicon.17e50649.svg        1.49 KiB
dist/index.html                         0.53 KiB
dist/assets/Dynamic.01efda59.js         0.43 KiB / gzip: 0.30 KiB
dist/assets/Dynamic.d0005171.css        0.03 KiB / gzip: 0.05 KiB
dist/assets/index.1e236845.css          47.92 KiB / gzip: 4.00 KiB
dist/assets/react-vendor.2fa3f57c.js    195.36 KiB / gzip: 48.57 KiB
dist/assets/index.8411a8fb.js           496.64 KiB / gzip: 108.14 KiB
```


可以发现 `react-vendor` 可以正常拆分出来，查看它的内容:



从中你可以看出 `react` 的一些间接依赖已经成功打包到了 `react-vendor` 当中，执行 `npx view preview` 预览产物页面也能正常渲染了:



说明循环依赖的问题已经被我们解决掉了。

终极解决方案

尽管上述的解决方案已经能帮我们正常进行产物拆包，但从实现上来看，还是显得略微繁琐，那么有没有开箱即用的拆包方案，能让我们直接用到项目中呢?

答案是肯定的，接下来我就给大家介绍 Vite 自定义拆包的终极解决方案—— `vite-plugin-chunk-split` 。

首先安装一下这个插件:

```
pnpm i vite-plugin-chunk-split -D
```

然后你可以在项目中引入并使用:

```
// vite.config.ts
import { chunkSplitPlugin } from 'vite-plugin-chunk-split';

export default {
  chunkSplitPlugin({
    // 指定拆包策略
    customSplitting: {
      // 1. 支持填包名。`react` 和 `react-dom` 会被打包到一个名为`render-vendor`的 chunk 里面(包
      'react-vendor': ['react', 'react-dom'],
      // 2. 支持填正则表达式。src 中 components 和 utils 下的所有文件被会被打包为`component-util`
      'components-util': [/src\/components/, /src\/utils/]
    }
  })
}
```

相比于手动操作依赖关系，使用插件只需几行配置就能完成，非常方便。当然，这个插件还可以支持多种打包策略，包括 unbundle 模式打包，你可以去 使用文档 探索更多使用

姿势。

小结

恭喜你，学习完了本小节的内容，我们最后来小结和回顾一下。在本小节，你需要重点掌握**拆包的意义**、**Vite 中的默认拆包策略**和**自定义拆包方法**。

首先我给你分析了拆包技术所解决的问题，主要包括**无法按需加载**以及**线上缓存命中率低**两个问题，然后我们一起通过具体的项目示例尝试了 Vite 默认的拆包策略，但默认的策略也是有缺陷的，比如对第三方包模块无法做进一步的拆包，这就需要我们自定义拆包策略了。

由于 Vite 生产环境使用 Rollup 进行打包，在后续的内容我们深入学习了 Rollup 底层的拆包 API—— `manualChunks` ，用 对象配置 和 函数配置 两种方式来自定义拆包策略，对象配置使用上比较简单，但函数配置更加灵活。随后我和你分析了函数配置中容易遇到的坑——chunk 循环依赖问题，并分享了我的解决思路和方案。不过一般情况下，大家将 `manualChunks` 配置为一个对象即可，如果需要进行深度的拆包优化可以采用函数的方式，相信经过今天的学习你也能很好地驾驭函数拆包配置。

最后，欢迎你在评论区记录你的学习心得和收获，也欢迎和我一起讨论。我们下一节再见👏。

> 扩展阅读: 阮一峰 ECMAScript 6 入门——模块循环依赖加载

上一篇： HMR API 及原理：代码改动后，如何进行毫秒级别的局部更新?

下一篇：语法降级与Polyfill：联合前端编译工具链，消灭低版本浏览器兼容问题