第 5 章

CHAPTER 5

打　包　器

6min

在软件工程领域，软件工程环境（Software Engineering Environment，SEE）是指在构筑一个新软件时所依赖的工具和基础设施等，包括软件环境和硬件环境。通常来讲，前端工程师会将工程环境区分为开发环境及生产环境等，用于更好地为各种环境的软件提供支撑。

对生产环境而言，为了能更好地提供应用功能、快速响应用户需求，前端工程师会对前端资源文件进行处理，打包器便由此而生。打包器，也称为打包软件或者封包软件，是将一个或多个文件或文件夹打包成一个单独的文件的工具。打包软件通常会对文件进行压缩和加密，以便在传输或存储过程中保护文件的安全性和完整性，其应用范围非常广泛，可以用于备份、存储、传输、共享和分发文件等方面。

注意：随着浏览器对 ESModule 的支持，基于捆绑打包（Bundle）和非捆绑打包（Bundleless）的打包构建方案便成为一个值得探讨的话题，前端工业环境中目前仍以捆绑打包（Bundle）构建方式为主。

前端打包器是一种开发工具，用于将前端项目中的多个文件和模块打包成一个或多个可供浏览器加载的文件，其能优化资源的加载和使用，提高前端应用的性能和响应速度。通常来讲，打包器能够处理 JavaScript、CSS、图像等各种前端资源，并且支持模块化开发，还要能够提供丰富的配置选项来定制打包的行为和输出的结果。使用前端打包器可以提高开发效率，同时也能够优化前端应用的性能和响应速度，并且通过合理地配置打包策略，可以灵活地处理各种前端资源，满足项目的需求。

注意：对于工程环境的区分，通常涉及分支的管理与划分，不同团队对分支的管理方式有所不同，本书会在后续章节进行简单介绍。

在前端领域中，打包器的发展大致经历了几个阶段，如手动构建阶段、文件构建阶段、模块构建阶段及多语言构建阶段，如图 5-1 所示。

在 2009 年之前，在早期的前端开发中，并没有像今天这样复杂的前端项目，因此也没有专门的打包工具。开发者通常会借助后端构建工具做支撑，手动管理文件依赖，将各个文件

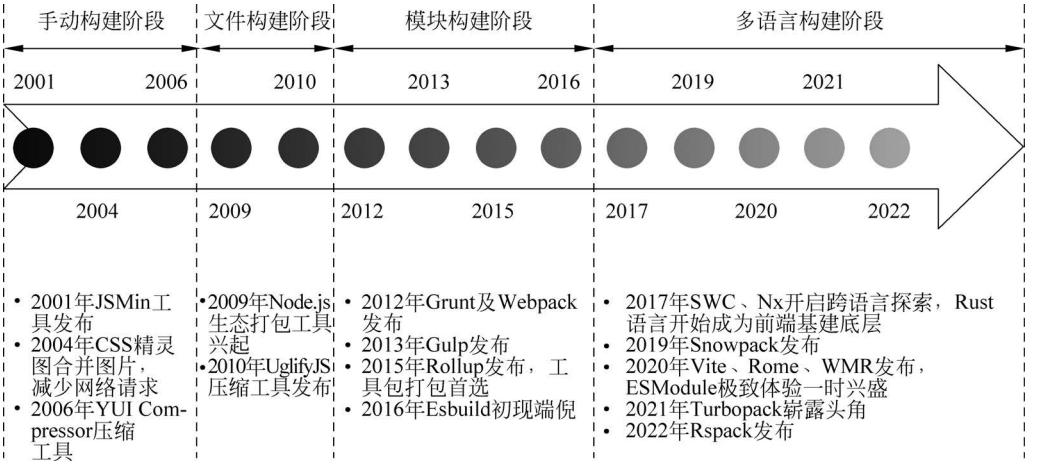

图 5-1 前端打包器发展历史

分别引入 HTML 文件中。

随着前端项目复杂性的增加，手动管理文件变得越来越困难。于是，随着 Node.js 的诞生，前端出现了一些手工打包工具。开发者可以使用这些工具来定义任务和文件的依赖关系，然后通过命令行运行任务来进行打包。

到了 2012 年，随着以 Webpack 为代表的打包器的发布，以模块化构建作为主流构建方案的前端打包器逐渐成为前端开发中的主流构建工具，但是，由于前端发展过程中的历史问题，前端模块化一直是前端开发领域的“阿喀琉斯之踵”，其包含纷繁的模块化方案，如 AMD(Asynchronous Module Definition)、UMD(Universal Module Definition)、CommonJS(Common JavaScript)、ESM(ECMAScript Modules)等。这一阶段的打包工具，通常会以一种模块化方案作为基准对不同模块化方案进行转化。

近年来，为了提升开发体验、加速打包构建流程，以 Rust 及 Go 语言为代表的多语言前端打包构建方案也相继出现，包括 ESBuild、Turbopack、Rome 等。

本章主要简单介绍前端业界中常见的 4 种打包工具，分别是 Webpack、Rollup、Gulp 及 Vite。对于前端工程而言，选择通用或者自研打包构建工具是前端工程化的一项重要内容。最后，本章也会简单地对比及总结业界已有的不同打包工具的优点及缺点，以便各位工程师在不同场景下更好地对使用哪种打包工具做出选择。

12min

5.1 Webpack

Webpack 是一个现代化的前端构建工具，被广泛地应用于前端开发中。Webpack 最早由德国开发者托比亚斯·科伯斯(Tobias Koppers)在 2012 年创建，其旨在解决前端开发中复杂的模块化和构建问题。

过去，在前端开发中常见的做法是手动引入和管理各个模块文件，这导致了代码复用性

差、难以维护和难以扩展的问题。为了解决这些问题，出现了一种名为模块加载器（Module Loader）的工具，用于将代码模块化并自动解析模块之间的依赖关系。

Webpack 正是基于这一需求而诞生的，其采用了类似 Node.js 的 CommonJS 规范，支持代码分割、模块依赖解析、文件压缩等功能，并且能够以配置的方式进行自定义。开发者通过 Webpack 可以将项目中的各个模块打包成一个或多个静态资源文件，以提高页面加载性能，减少网络请求次数。

回顾 Webpack 的发展历程，其大致可以分为初始阶段、改进阶段、增长阶段、繁荣阶段及革新阶段，如图 5-2 所示。

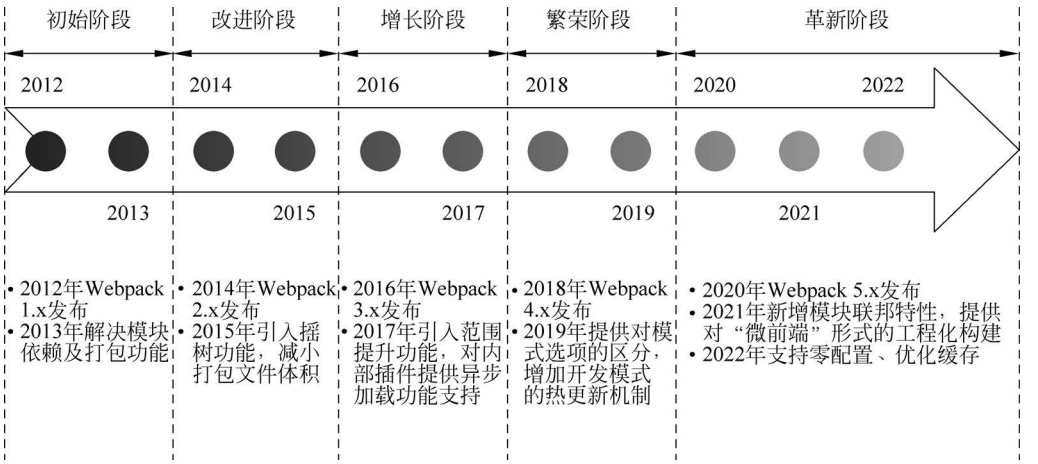

图 5-2 Webpack 发展历程

初始阶段大致可以追溯到 2012—2013 年，彼时的 Webpack 为了解决在前端开发中模块化的问题而主要关注模块之间的依赖关系和打包功能。到了 2014 年，Webpack 2.x 版本的发布标志着 Webpack 正式进入改进阶段，其最显著的特点是引入了摇树（Tree Shaking）功能，通过静态分析来确定哪些代码可以从最终的打包文件中删除，从而减小打包文件的体积。

时间来到 2016 年，Webpack 3.x 版本的发布进一步优化了性能和打包结果，其引入了范围提升（Scope Hoisting）功能，可将模块之间的依赖关系简化，从而减小打包后的代码体积。此外，Webpack 3.x 还优化一些其他的功能，包括 CommonsChunkPlugin 插件的改进和模块的异步加载的支持等。

在 2018 年，Webpack 4.x 版本正式发布，其可谓是奠定 Webpack 打包器领域王者地位的高光之作。Webpack 4.x 引入了一些重要的功能，其最重要的是引入了新的配置方式，通过 mode 选项可以自动启用相应的功能，例如开发模式下的热更新和生产模式下的压缩功能等。此外，Webpack 4.x 还优化了构建速度和打包后的体积。

随着单页应用单包构建体积的增大，也伴随着浏览器对 ESModule 的友好支持，前端领域出现了是否需要进行打包构建的大讨论。作为上一时代的王者，Webpack 毋庸置疑被推

到了风口浪尖。面对众多前端开发者对单包构建冗重的“口诛笔伐”,Webpack 以 5.x 版本的发布作为其突破创新的正式回应,其进一步改进了性能和功能,并引入了一些重要的特性,包括支持通过模块联邦(Module Federation)实现跨项目共享代码、支持零配置(Zero Configuration)构建等。此外,Webpack 5.x 还提供了许多其他功能,例如缓存策略及代码分割等。

随着时间的推移,Webpack 持续优化功能和提升性能,其逐渐成为前端开发中最流行的构建工具之一,并且在社区中积累了大量的插件和工具,而这些插件和工具可以进一步增加 Webpack 的功能和灵活性。可以说,Webpack 的诞生解决了在前端开发中的模块化和构建问题,并且持续发展演进,为开发者提供了更高效、更便捷的前端开发体验,因此,本节将分别介绍 Webpack 不同版本的实现方式及对应版本的特色,以期能够让读者全面地了解 Webpack 的原理与功能。

1. Webpack 1.x

Webpack 1.x 最初设计就是为了对在浏览器运行的 JavaScript 文件进行打包,其支持代码拆分和按需加载,可以将应用拆分为多个模块,通过 Tapable 钩子函数的丰富插件系统,扩展 Webpack 的功能。同时,Webpack 支持诸如 CSS、图片等各种资源的加载和处理,也提供了强大的 loader 机制,可以通过 loader 对各种资源进行预处理,如图 5-3 所示。

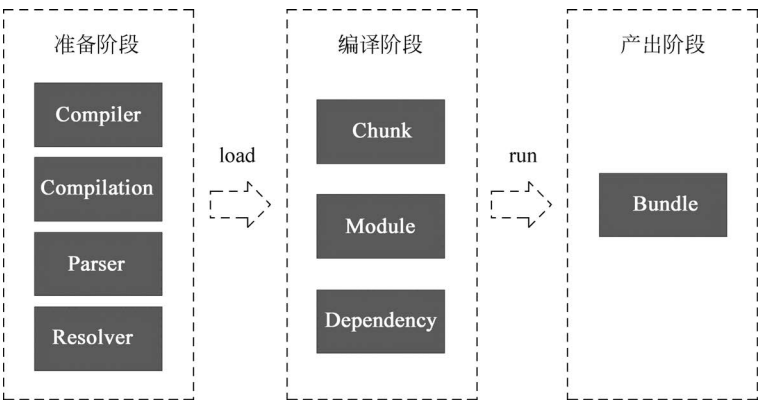

图 5-3 Webpack 1.x 原理

其中,Compiler 是用于 Webpack 构建时执行完整编译过程的对象,代码如下:

```
//第 5 章/webpack1.x.js
function Compiler() {
    //基于 Tapable 的操作
    Tapable.call(this);
}
Compiler.prototype.run = function(callback) {
    //执行 Tapable 中的 applyPluginsAsync 方法
    this.applyPluginsAsync("run", this);
};
```

```
Compiler.prototype.compile = function(callback) {
    this.applyPlugins("compile", params);
    //Compilation 是原始物料
    var compilation = this.newCompilation(params);
    //make 是核心打包过程
    this.applyPluginsParallel("make", compilation);
}
```

Compilation 则是针对项目文件通过编译解析成 module 的原始物料，其代码如下：

```
//第 5 章/webpack1.x.js
function Compilation(compiler) {
    Tapable.call(this);
    this.compiler = compiler;
    this.chunks = [];
    this.namedChunks = {};
    this.modules = [];
    this._modules = {};
}
Compilation.prototype.addModule = function(module, cacheGroup) {
    var identifier = module.identifier();
    this._modules[identifier] = module;
    //添加模块
    this.modules.push(module);
    return true;
};
Compilation.prototype.getModule = function(module) {
    var identifier = module.identifier();
    return this._modules[identifier];
};
Compilation.prototype.findModule = function(identifier) {
    return this._modules[identifier];
};
Compilation.prototype.buildModule = function(module, thisCallback) {
    this.applyPlugins("build-module", module);
    if(module.building) return module.building.push(thisCallback);
    var building = module.building = [thisCallback];
    //module 构建
    module.build();
};
//添加 chunk
Compilation.prototype.addChunk = function addChunk(name, module, loc) {
    var chunk;
    if(name) {
            if(Object.prototype.hasOwnProperty.call(this.namedChunks, name)) {
                    chunk = this.namedChunks[name];
                    if(module) {
                            chunk.addOrigin(module, loc);
                    }
                    return chunk;
```

```
                }
        }
        chunk = new Chunk(name, module, loc);
        this.chunks.push(chunk);
        if(name) {
                this.namedChunks[name] = chunk;
        }
        return chunk;
};
//创建 chunk 物料
Compilation.prototype.createChunkAssets = function createChunkAssets() {
        //遍历 module 物料
        for(var i = 0; i < this.modules.length; i++) {
                var module = this.modules[i];
                if(module.assets) {
                        Object.keys(module.assets).forEach(function(name) {

                                this.applyPlugins("module-asset", module, file);
                        }, this);
                }
        }
        //遍历 chunk 物料
        for(i = 0; i < this.chunks.length; i++) {
                var chunk = this.chunks[i];
                this.applyPlugins("chunk-asset", chunk, file);
        }
}
```

因此,对于 Webpack 1.x 而言,大致可以总结为插件系统、代码拆分、资源处理。

2. Webpack 2.x

Webpack 2.x 则开始支持 ESModule 规范,通过引入 Tree Shaking 功能,可以通过静态代码分析来消除未使用的代码,减小打包后的文件体积。由于支持 ES 模块语法,故而 Webpack 2.x 可以通过 import 和 export 语法对模块进行导入和导出,同时添加了异步动态导入功能,其可以在运行时动态地加载模块,如图 5-4 所示。

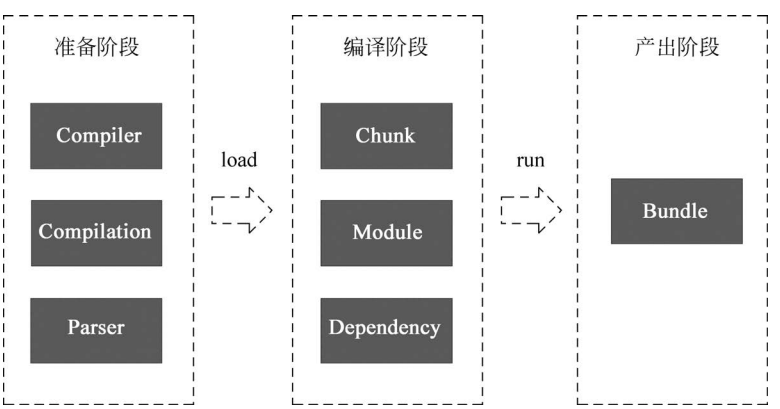

图 5-4 Webpack 2.x 原理

其中，对于 Module 部分可通过定义基类与编码文件进行相互映射，代码如下：

```
//第 5 章/webpack2.x.js
class Module extends DependenciesBlock {
    constructor() {
            super();
            this.context = null;
            //依赖
            this.reasons = [];
            this.chunks = [];
    }
    addReason(module, dependency) {
            //ModuleReason 类对 module 和 dependency 进行了解耦
            this.reasons.push(new ModuleReason(module, dependency));
    }
    removeReason(module, dependency) {
            for(let i = 0; i < this.reasons.length; i++) {
                    let r = this.reasons[i];
                    if(r.module === module && r.dependency === dependency) {
                            this.reasons.splice(i, 1);
                            return true;
                    }
            }
            return false;
    }
    hasReasonForChunk(chunk) {
            for(let r of this.reasons) {
                    if(r.chunks) {
                            if(r.chunks.indexOf(chunk) >= 0)
                                    return true;
                    } else if(r.module.chunks.indexOf(chunk) >= 0)
                            return true;
            }
            return false;
    }
    rewriteChunkInReasons(oldChunk, newChunks) {
            this.reasons.forEach(r => {
                    if(!r.chunks) {
                            if(r.module.chunks.indexOf(oldChunk) < 0)
                                    return;
                            r.chunks = r.module.chunks;
                    }
                    r.chunks = r.chunks.reduce((arr, c) => {
                            return arr;
                    }, []);
            });
    }
}
```

因而，对于 Webpack 2.x 而言，其特点可以大致总结为静态分析、异步导入、动态加载。

3. Webpack 3.x

Webpack 3.x 则通过作用域提升(Scope Hoisting)来减少打包后的代码体积和运行时的开销,同时支持生成 NamedChunks 提供更友好的命名方式,也支持配置文件合并和条件配置,如图 5-5 所示。

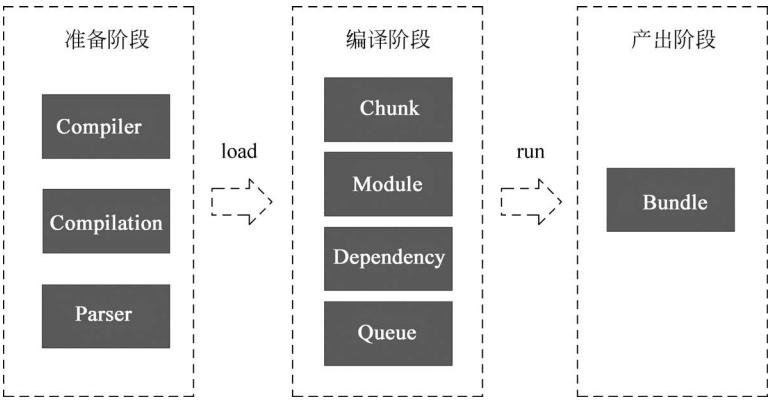

图 5-5 Webpack 3.x 原理

对于 Chunk 而言,其可以通过不同 Module 之间的关系构建生成,代码如下:

```
//第 5 章/webpack3.x.js
class Chunk {
    constructor(name, module, loc) {
            this.id = null;
            this.ids = null;
            this.name = name;
            this._modules = {};
            this.chunks = [];
            this.parents = [];
            this.blocks = [];
            this.origins = [];
            this.files = [];
            this.rendered = false;
            if(module) {
                    this.origins.push({
                            module,
                            loc,
                            name
                    });
            }
    }
    addOrigin(module, loc) {
            this.origins.push({
                    module,
```

```
                    loc,
                    name: this.name
                });
        }
}
```

因而，对于 Webpack 3.x 而言，其特点可以大致总结为范域提升、友好命名、条件配置。

4. Webpack 4.x

Webpack 4.x 则为了更好地支持环境区分，引入了新的默认模式(mode)选项，其可以根据开发或生产环境自动优化配置。除此之外，Webpack 4.x 替代了 CommonsChunkPlugin 和 UglifyJsPlugin 等插件，使用 optimization 选项优化配置，从而提升性能，并且通过持久化缓存和多线程构建来加快打包速度，如图 5-6 所示。

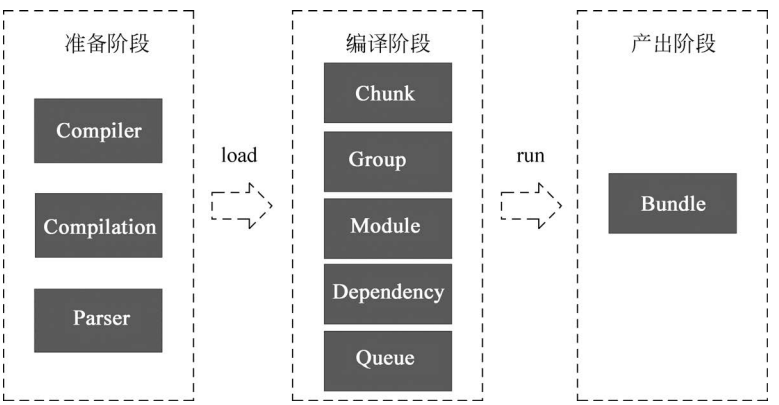

图 5-6 Webpack 4.x 原理

其中，Webpack 4.x 引入了 ChunkGroup 的概念，用于更好地聚合 Chunk 来构建最终的 Bundle 文件，代码如下：

```
//第 5 章/webpack4.x.js
class ChunkGroup {
    constructor(options) {
            this.options = options;
            this.chunks = [];
    }
    //当 Chunk 加入 ChunkGroup 时执行
    addOptions(options) {
            for (const key of Object.keys(options)) {
                    if (this.options[key] === undefined) {
                            this.options[key] = options[key];
                    } else if (this.options[key] !== options[key]) {
                            if (key.endsWith("Order")) {
                                    this.options[key] = Math.max(this.options[key], options
[key]);
```

```
                                } else {
                                        throw new Error(
                                            `ChunkGroup.addOptions: No option merge strategy for
    ${key}`
                                        );
                                }
                            }
                        }
                    }
                }
```

因而,对于 Webpack 4.x 而言,其特点可以总结为模式选项、配置优化、多线并行。

5. Webpack 5.x

近年来,随着微前端理念的盛行,Webpack 5.x 也通过引入模块联邦功能将多个独立的 Webpack 构建在客户端而共享模块之间的依赖与生成。同时,随着单包构建时间缓慢的问题的影响,Webpack 5.x 也开始支持 WebAssembly 模块的导入和使用,并且也开始支持增量构建,用于缩短每次构建的时间,如图 5-7 所示。

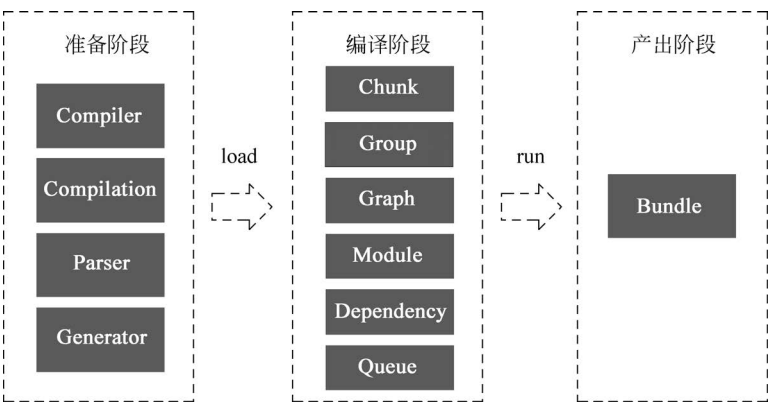

图 5-7 Webpack 5.x 原理

为了更好地支持 Chunk 之间的依赖检索,Webpack 5.x 通过构建 ChunkGraph 来明晰各个 Chunk 之间的相互依赖关系,其中,Dependency 用于构建依赖的数据结构,其代码如下:

```
//第 5 章/webpack5.x.js
class Dependency {
    constructor() {
            this._parentModule = undefined;
            this._parentDependenciesBlock = undefined;
            this._parentDependenciesBlockIndex = -1;
            this.weak = false;
            this.optional = false;
            this._locSL = 0;
            this._locSC = 0;
```

```
            this._locEL = 0;
            this._locEC = 0;
            this._locI = undefined;
            this._locN = undefined;
            this._loc = undefined;
        }
    }
```

对于 Dependency 之间的关系，则可以通过 ChunkGraph 更好地进行关联与管理，代码如下：

```
    //第 5 章/webpack5.x.js
    class ChunkGraph {
        constructor(moduleGraph, hashFunction = "md4") {
                this._modules = new WeakMap();
                this._chunks = new WeakMap();
                this._blockChunkGroups = new WeakMap();
                this._runtimeIds = new Map();
                this.moduleGraph = moduleGraph;
        }
        _getChunkGraphModule(module) {
                let cgm = this._modules.get(module);
                if (cgm === undefined) {
                        //ChunkGraphModule 用于记录 module 与外界的关系，其中 chunks 参数记录了
    //module 关联的 chunk
                        cgm = new ChunkGraphModule();
                        this._modules.set(module, cgm);
                }
                return cgm;
        }
        _getChunkGraphChunk(chunk) {
                let cgc = this._chunks.get(chunk);
                if (cgc === undefined) {
                        //ChunkGraphChunk 用于记录 chunk 与外界的关系，其中 module 参数记录了
    //chunk 关联的 modules
                        cgc = new ChunkGraphChunk();
                        this._chunks.set(chunk, cgc);
                }
                return cgc;
        }
        connectChunkAndModule(chunk, module) {
                const cgm = this._getChunkGraphModule(module);
                const cgc = this._getChunkGraphChunk(chunk);
                cgm.chunks.add(chunk);
                cgc.modules.add(module);
        }
        disconnectChunkAndModule(chunk, module) {
                const cgm = this._getChunkGraphModule(module);
                const cgc = this._getChunkGraphChunk(chunk);
                cgc.modules.delete(module);
                if (cgc.sourceTypesByModule) cgc.sourceTypesByModule.delete(module);
```

```
                cgm.chunks.delete(chunk);
            }
        }
```

因而,Webpack 5.x 的特点可以总结为模块联邦、持久缓存、增量构建。

注意:Webpack 生态体系十分庞大且繁杂,本书仅仅对整个 Webpack 的设计主线进行了剖析,对于其丰富的插件(plugin)生态设计及加载器(loader)等都十分值得玩味,感兴趣的读者可阅读源码及资料深入地进行学习。

8min

5.2 Rollup

Rollup 是一种现代化的前端构建工具,用于打包 JavaScript 代码。Rollup 于 2014 年由里奇·哈里斯(Rich Harris)创建,并在 Web 开发社区中得到了广泛认可和采纳。

相较于其他构建工具,Rollup 的核心特性是利用 ES 模块化系统对代码进行打包。ES 模块化是 JavaScript 的官方模块化方案,具有静态引用、摇树和作用域分析等优势,可将代码打包得更小并且更高效。

Rollup 的设计理念是只生成实际被使用的代码,而不是将整个库或框架打包到最终的输出文件中,其可以减小生成的代码体积,提升运行时性能。另外,Rollup 对于第三方库的导入也能灵活地进行处理,其能够将第三方库中暴露的功能按需引入,避免打包整个库。

同样地,回顾 Rollup 整个发展历史,其大致可以分为启动阶段、初创阶段、专注阶段及复兴阶段,如图 5-8 所示。

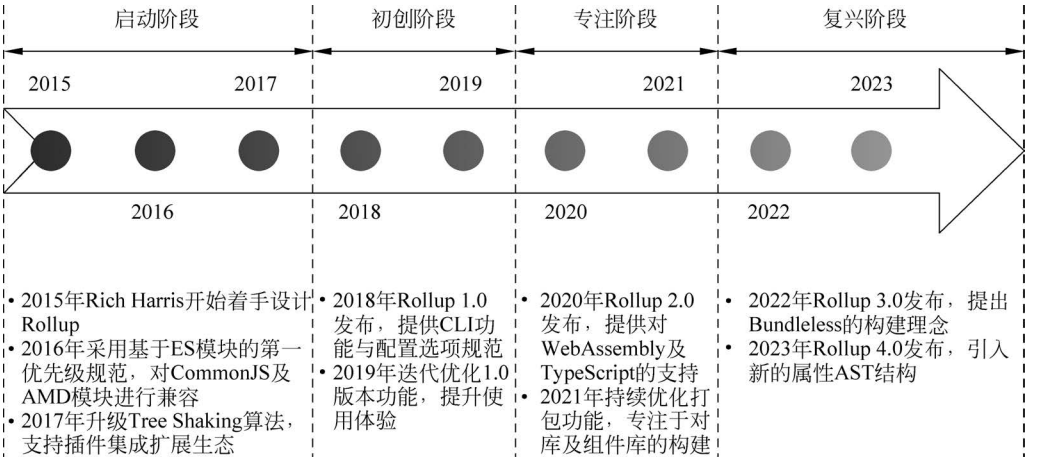

图 5-8 Rollup 发展历程

早在2015年，里奇·哈里斯(Rich Harris)就开始着手打包器相关的设计与开发。到了2018年，Rollup正式发布了1.0版本，其最初是为了解决JavaScript模块打包的问题。与Webpack所不同的是，由于Rollup诞生的时机恰好与ECMAScript第6个版本的发布重合，所以其采用了ES模块的格式规范，而不是以前的CommonJS和AMD等特殊解决方案。正因如此，Rollup从创建之初便有了Tree Shaking等功能，并且也具备使用各种插件来扩展自身功能的生态。

到了2020年，Rollup发布了2.0版本，其带来了一些重要的改进，包括对WebAssembly模块的支持及更好地对TypeScript支持，但是，相较于Webpack在应用打包场景的风生水起，Rollup将自己的精力专注于对JavaScript包或组件库的打包构建场景之中。

随着浏览器对ES模块的支持力度加大，Rollup 3.0进行了重大重构和改进，尽管其仍然是将捆绑打包(Bundle)作为最终目标，但是随着非捆绑打包(Bundleless)的构建理念兴起，这也为后续诸如Vite、Snowpack等前端打包器的设计思路提供了借鉴和参考。

本质上来讲，Rollup是一种基于ES模块化的前端构建工具，为开发者提供了高效、轻量级的代码打包解决方案，其设计理念和特性都使生成的代码更小、运行更高效。到目前为止，Rollup已经发布了最新的4.0版本，其更进一步地提升了打包构建的效率与性能。事实上，真正生产环境下的应用打包器选择仍旧以Webpack作为首选方案居多，但Rollup在开发环境下的体验无疑是对前端工程化的发展起到了很好的推动作用，因此，本节将分别介绍Rollup不同版本的实现方式及对应版本的特色，以期能够让读者全面地了解Rollup的原理与功能。

1. Rollup 1.x

Rollup 1.x作为Rollup的第1个正式版本，其采用非常直观的配置和命令行接口，并天然地支持Tree Shaking机制，可以通过静态分析来删除未使用的代码，从而最大程度地减小输出文件的大小。可以说，Rollup 1.x的发布对Rollup的后续发展起到了至关重要的作用，如图5-9所示。

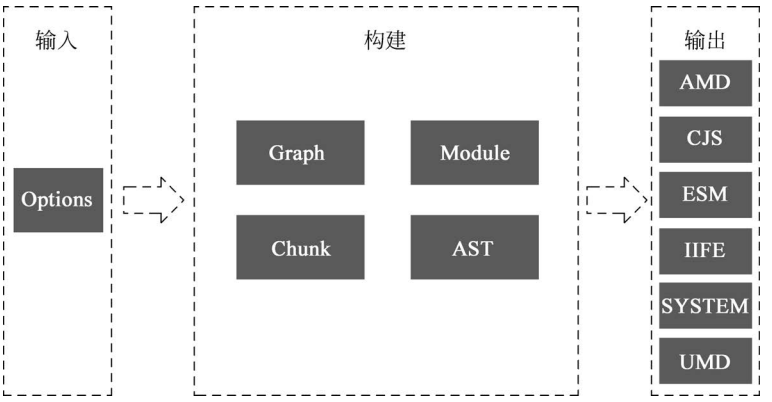

图5-9　Rollup 1.x原理

对于 Rollup 1.x 而言,其提供了简单直观的配置格式,方便开发者进行不同格式的输出,代码如下:

```
//第 5 章/rollup1.x.js
//打包 AMD 格式
export default function amd() {}
//打包 CJS 模块
export default function cjs(magicString, {snippets}) {
    const { _ } = snippets;
    //输出 CJS 格式
    magicString.append(`module.exports ${_} = ${_}`);
}
//打包 ESM 模块
export default function es(magicString) {
    //输出 ESM 格式 Rollup 默认将 ESM 作为第一优先格式
    magicString.trim();
}
//打包 IIFE 模块
export default function iife() {}
//打包 SYSTEM 格式
export default function system() {}
//打包 UMD 格式
export default function umd() {}
```

因此,Rollup 1.x 的特点可以总结为直观配置、命令接口、多格式输出。

2. Rollup 2.x

Rollup 2.x 则改进了对 TypeScript 的支持,其通过引入新的插件使对 TypeScript 的支持更加完善和可靠,并且 Rollup 也开始兼容不同模块的转化,新增了对 CommonJS 和 AMD 模块的解析和转换,使开发者可以更方便地使用这些模块系统,同时也在不断地优化打包性能、提升打包速度,如图 5-10 所示。

和 Webpack 一样,Rollup 同样需要提供对 Module 基于依赖 Graph 的构建生成 Chunk。所不同的是,Rollup 是天然支持 ESModule 的,故其天然就有 Tree Shaking 能力,代码如下:

```
//第 5 章/rollup2.x.ts
export default class Graph {
    private modules = []
    constructor() {
            //ModuleLoader 是 module 加载器的基类
            this.moduleLoader = new ModuleLoader();
    }
    build(
```

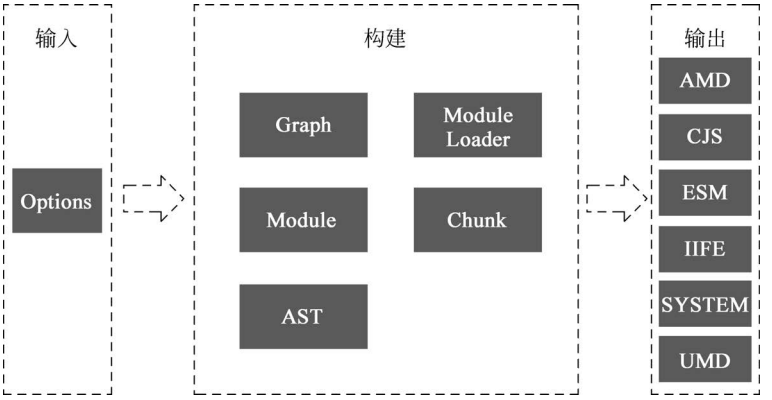

图 5-10　Rollup 2.x 原理

```
            entryModules,
            manualChunks
    ) {
            //阶段 1: 查询需要加载的入口模块
            return Promise.all([
                    this.moduleLoader.addEntryModules()
            ]).then(([{ entryModules, manualChunks }]) => {
                    //阶段 2: 链接到拓扑关系
                    this.link(entryModules);
                    //阶段 3: 标记状态
                    for (const module of entryModules) {
                            module.includeAllExports();
                    }
                    this.includeMarked(this.modules);
                    //阶段 4: 构建 chunk
                    const chunks = [];
                    for (const chunk of chunks) {
                            chunk.link();
                    }
                    const facades = [];
                    for (const chunk of chunks) {
                            facades.push(...chunk.generateFacades());
                    }
                    return [...chunks, ...facades];
            });
    }
    private link(entryModules) {
            for (const module of this.modules) {
                    module.linkDependencies();
            }
            for (const module of this.modules) {
```

```
                        module.bindReferences();
                }
        }
}
```

对于 Module 而言，Rollup 2.x 提供了 Module Loader 对 Module 进行预处理，其代码如下：

```
//第 5 章/rollup2.x.ts
export class ModuleLoader {
    constructor(
            graph,
            modulesById,
            pluginDriver,
            external,
    ) {
            this.graph = graph;
            this.modulesById = modulesById;
            this.pluginDriver = pluginDriver;
            this.isExternal = getIdMatcher(external);
    }
}
```

通过对 Module 的预处理来增加用户侧的介入，提升扩展性，Module 的核心代码如下：

```
//第 5 章/rollup2.x.ts
export default class Module {
    chunk;
    code;
    id;
    sources = new Set();
    dependencies = new Set();
    resolvedIds;
    originalCode;
    transformFiles;
    private ast;
    private esTreeAst;
    private graph;
    private transformDependencies = [];
    constructor(graph, id, moduleSideEffects) {
            this.id = id;
            this.graph = graph;
            this.context = graph.getModuleContext(id);
            this.moduleSideEffects = moduleSideEffects;
    }
    setSource({ast, code, resolvedIds, transformDependencies, transformFiles}) {}
```

```
    toJSON() {
            return {
                    ast: this.esTreeAst,
                    code: this.code,
                    dependencies: Array.from(this.dependencies).map(module => module.id),
                    id: this.id,
                    originalCode: this.originalCode,
                    resolvedIds: this.resolvedIds,
                    transformDependencies: this.transformDependencies,
                    transformFiles: this.transformFiles
            };
    }
}
```

因此，Rollup 2.x 的核心特点可以归纳为类型改进、模块转换、性能提升。

3. Rollup 3.x

Rollup 3.x 新增了对多级文件的引用输出，开发者可以方便地将代码分割为多个输出文件，实现更好的代码组织，从而优化加载性能，方便开发者按需加载和更好地利用缓存。除此之外，Rollup 3.x 还对自身插件生态系统进行了改进，丰富自身插件生态并提供更加灵活的插件编写接口，如图 5-11 所示。

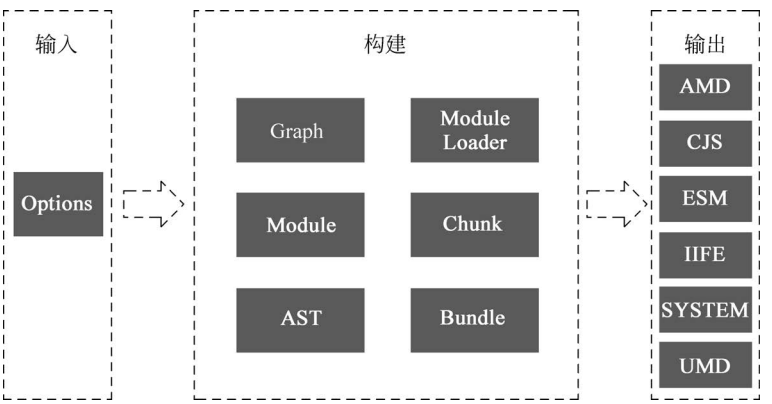

图 5-11　Rollup 3.x 原理

对于多个 Module 生成的 Chunk，其自身依据特殊的 AST 转化后进行融合，代码如下：

```
//第 5 章/rollup3.x.ts
export default class Chunk {
    readonly entryModules = [];
    execIndex;
    private readonly dynamicEntryModules = [];
    private readonly exports = new Set();
    private implicitEntryModules = [];
    constructor(
```

```
                private readonly orderedModules,
                private readonly chunkByModule,
                private readonly includedNamespaces,
        ) {
                this.execIndex = orderedModules.length > 0 ?
orderedModules[0].execIndex : Infinity;
                const chunkModules = new Set(orderedModules);

                for (const module of orderedModules) {
                        chunkByModule.set(module, this);
                        if (module.namespace.included) {
                                includedNamespaces.add(module);
                    }
                    if (module.info.isEntry || outputOptions.preserveModules) {
                                this.entryModules.push(module);
                    }
                    for (const importer of module.includedDynamicImporters) {
                                if (!chunkModules.has(importer)) {

        this.dynamicEntryModules.push(module);
                            //具有合成导出的模块需要为动态导入提供一个人工命名空间
                                if (module.info.syntheticNamedExports && !outputOptions.
preserveModules) {

includedNamespaces.add(module);

this.exports.add(module.namespace);
                                }
                            }
                    }
                    if (module.implicitlyLoadedAfter.size > 0) {
                            this.implicitEntryModules.push(module);
                    }
            }
        }
}
```

最后,根据所需导出的格式可将输入打成 Bundle 进行输出,代码如下:

```
//第 5 章/rollup3.x.ts
async function renderChunks(chunks) {
    //render chunks 开始
    const renderedChunks = await Promise.all(chunks.map(chunk => chunk.render()));
    //render chunks 结束
}
export default class Bundle {
    constructor(
            private readonly outputOptions,
            private readonly inputOptions,
            private readonly pluginDriver,
```

```
                private readonly graph
        ) {}
        async generate() {
                const outputBundleBase = Object.create(null);
                const outputBundle = {…outputBundleBase};
                this.pluginDriver.setOutputBundle(outputBundle, this.outputOptions);
                //initialize render 开始
                await this.pluginDriver.hookParallel('renderStart', [this.outputOptions, this.
    inputOptions]);
                //initialize render 结束
                //generate chunks 开始
                const chunks = await this.generateChunks(outputBundle);
                for (const chunk of chunks) {
                        chunk.generateExports();
                }
                //generate chunks 结束
                await renderChunks(chunks);
                //generate bundle 开始
                await this.pluginDriver.hookSeq('generateBundle', [
                        this.outputOptions,
                        outputBundle
                ]);
                //generate bundle 结束
                return outputBundleBase;
        }
        private async generateChunks(bundle) {
                const chunks = [];
                const facades = [];
                for (const chunk of chunks) {
                        facades.push(...chunk.generateFacades());
                }
                return [...chunks, ...facades];
        }
    }
```

注意：Rollup 的插件系统虽然不像 Webpack 那样丰富多样，但也有其独特的运行机制，感兴趣的读者可以阅读源码及资料进行对比分析。

5.3　Gulp

6min

Gulp 也是一种前端构建工具，可用于自动化任务的执行和前端资源的处理。Gulp 于 2013 年由艾瑞克·斯科夫斯托尔(Eric Schoffstall)创建，其与众多前端构建工具所不同的

流式构建方式迅速在前端开发社区中流行起来。

事实上,在 Gulp 出现之前,前端工程通常会使用 Grunt 进行构建,然而,由于 Grunt 本身配置文件仍需编写大量的配置代码而使配置变得冗长和复杂,其成为开发 Gulp 的诱因。为了解决 Grunt 的复杂配置问题,Gulp 采用了一种基于代码的任务流程描述方式,帮助开发者通过 JavaScript 代码来描述任务和流程,使任务的配置和管理更直观和更灵活。

Gulp 的发展历程大体可以分为起步阶段、重构阶段、完善阶段,如图 5-12 所示。

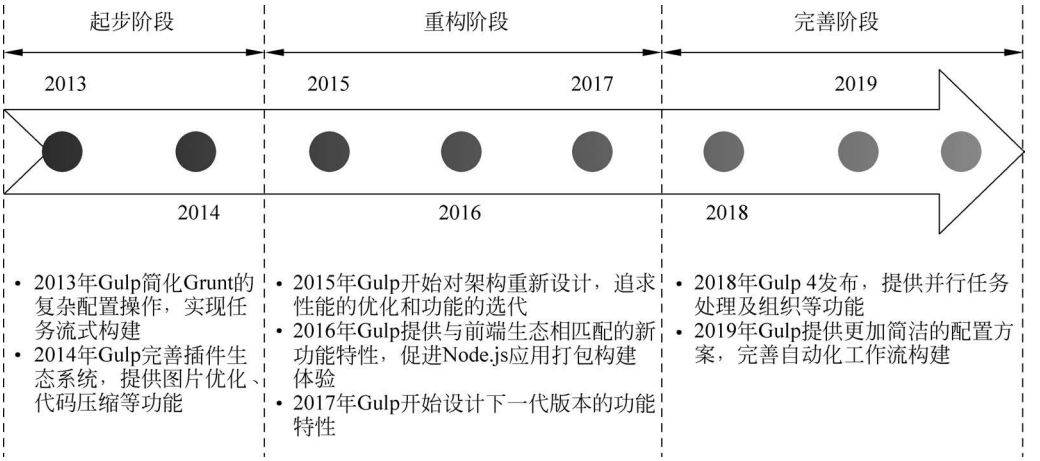

图 5-12 Gulp 发展历程

时间拨回 2013 年,最初的 Gulp 其实只是一个很小的项目,其只为了简化 Grunt 的复杂操作而提供了一些基本的功能,如文件复制、文件合并和文件压缩等,帮助开发者可以轻松地定义和执行这些常见的任务。随着 Gulp 的流行,其庞大的插件生态系统也开始形成,可以根据自己的需求来扩展和定制 Gulp 的功能,如 Sass 编译、代码压缩、图片优化等。

Gulp 的发布一直遵循“如无必要,勿增实体”的原则,其在沉寂许久后开始重新审视前端生态的现状与自身的定位。随着对架构的重新设计,Gulp 再次开启了功能的迭代优化工作,也促使了许多插件的重构。

Gulp 4 于 2018 年发布,其也是目前最新的版本,引入了一些新特性,如更好的错误处理并行任务执行、更灵活的任务组织等。尽管 Gulp 并不是严格意义上的前端打包器,但其简单的流式操作却十分契合 Node.js 应用场景的打包构建。Gulp 正是利用了 JavaScript 中的流(Stream)概念,通过连接各个任务并处理数据流,实现了高效的构建过程。

严格来讲,Gulp 只不过是一种基于 JavaScript 代码的流式构建工具。虽然 Gulp 起步很早,但其追求的极致简化理念却最终没有抢到前端打包器市场的先机,因此,本节将分别介绍 Gulp 不同版本的实现方式及对应版本的特色,以期能够让读者全面地了解 Gulp 的原理与功能。

1. Gulp 3.x

Gulp 3.x 是 Gulp 使用较为广泛的一个版本,其任务使用 task()函数进行注册并通过

回调的方式进行串行处理，而错误处理则是通过插件的方式进行接入兜底，如图 5-13 所示。

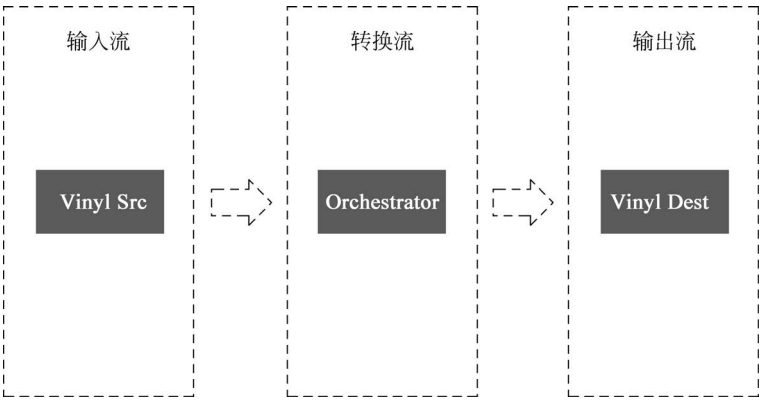

图 5-13　Gulp 3.x 原理

Gulp 3.x 任务调度的核心是通过 orchestrator 包进行实现的，代码如下：

```
//第 5 章/gulp3.x.js
var Orchestrator = function () {
    EventEmitter.call(this);
  //当队列中的所有任务都完成时调用
    this.doneCallback = undefined;
  //顺序执行任务
    this.seq = [];
  //task 包括 name、dep (依赖名称列表)和 fn (执行的任务)
    this.tasks = {};
  //是否在执行
    this.isRunning = false;
};
Orchestrator.prototype.add = function (name, dep, fn) {
  this.tasks[name] = {
    fn: fn,
    dep: dep,
    name: name
  };
  return this;
};
Orchestrator.prototype.start = function() {
  this.seq = seq;
  //发布订阅模式
  this.emit('start', {message:'seq: ' + this.seq.join(',')});
  return this;
};
Orchestrator.prototype.stop = function (err, successfulFinish) {
  this.emit('stop', {message:'orchestration succeeded'});
};
```

```
//任务执行的核心方法
function runTask(task, done) {
    var that = this, finish, cb, isDone = false, start, r;
    finish = function (err, runMethod) {
            isDone = true;
            done.call(that, err, {
                    runMethod
            });
    };
    cb = function (err) {
            finish(err, 'callback');
    };
    r = task(cb);
  r.then(function () {
    finish(null, 'promise');
  }, function(err) {
    finish(err, 'promise');
  });
}
Orchestrator.prototype._runTask = function (task) {
  runTask(task.fn.bind(this), function (err, meta) {});
}
```

Gulp 3.x 使调度操作继承自本身的构造函数,代码如下:

```
//第 5 章/gulp3.x.js
function Gulp(){
  Orchestrator.call(this);
}
Gulp.prototype.Gulp = Gulp;
```

因此,Gulp 3.x 的特点可以总结为回调任务、串行执行、文件监听。

2. Gulp 4.x

Gulp 4.x 则对 Gulp 3.x 的设计方案进行了改进,其任务可通过在 gulpfile 文件中定义普通 JavaScript 函数进行注册并返回基于 Promise 的方式进行串并行处理,而错误处理则被内置在核心包,如图 5-14 所示。

Gulp 4.x 任务调度的核心则通过 Gulp 自身独立开发的 undertaker 包进行处理,代码如下:

```
//第 5 章/gulp4.x.js
function Undertaker(customRegistry) {
  EventEmitter.call(this);
}
Undertaker.prototype.task = task;
Undertaker.prototype.series = series;
```

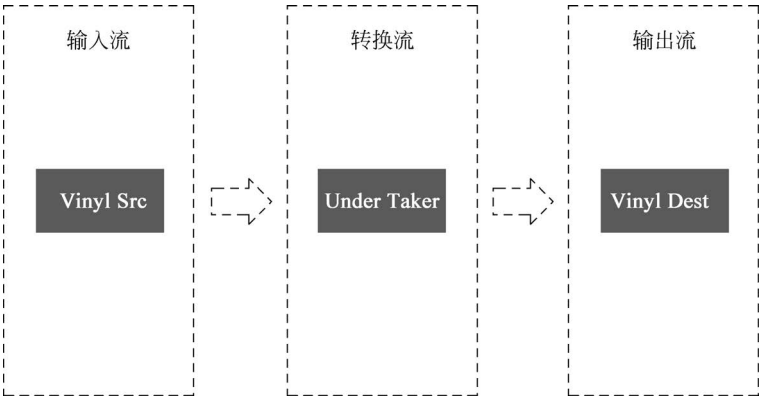

图 5-14　Gulp 4.x 原理

Gulp 4.x 同样通过构造函数进行继承，代码如下：

```
//第 5 章/gulp4.x.js
function Gulp() {
  Undertaker.call(this);
}
Gulp.prototype.Gulp = Gulp;
```

因此，Gulp 4.x 的特点可总结为约定处理并行任务、错误内置。

5.4　Vite

8min

正如开篇所提到的模块化纷争背景，Vite 的产生正是源自 Webpack 称霸时代捆绑打包而导致开发服务器臃肿，由此启动体验问题探索。Vite 是一个轻量级、快速的前端构建工具，由于其近乎即时的代码编译和快速的热模块更换，迅速受到广大前端开发者青睐。

Vite 主要由两部分组成，一个是通过本机 ES 模块提供源文件的开发服务器，另一个是可执行的命令行界面工具(Command Line Interface)。此外，Vite 还可以通过其插件 API 和 JavaScript API 提供具有高扩展性和全面性的功能支持。

由于 Vite 起步于开发体验的优化革新，故其发展历程大体可以分为探索阶段、起始阶段和增长阶段，如图 5-15 所示。

早在 2018 年，Vue.js 框架的作者尤雨溪(Evan You)便开始设想如何提升 Vue 脚手架在大型项目中热更新体验问题。最初，Vite 仅仅是为了提供 Vue 脚手架生态相关的原生 ESM 的服务器。到了 2019 年，随着浏览器对原生 ESM 支持的不断推广，Vite 开始不断借鉴已有的非捆绑打包(Bundleless)的服务器实现方案。

2020 年 4 月 21 日，Vite 0.1 发布，其能够转化 Vue 的单文件组件(Single File Component)

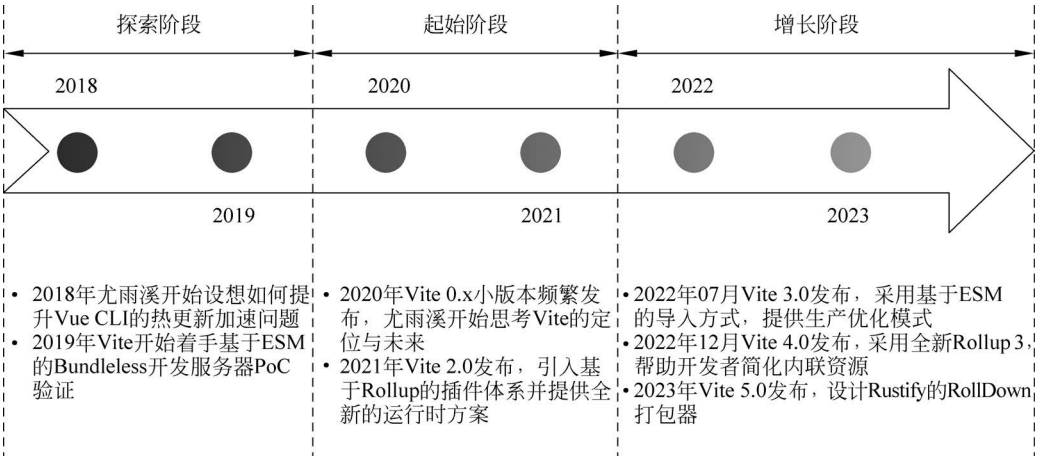

图 5-15　Vite 发展历程

并处理原生 ESM 的热更新。尽管该版本的核心逻辑比较粗糙，并且仅支持 Vue 组件，但却为后来的发展埋下了种子。紧接着，到了同年 11 月，Vite 发布了大概 91 个小版本，但考虑到 Vite 更广阔的场景与格局而并未诞生真正意义上的 Vite 1.0 版本。

2021 年 2 月 16 日，Vite 2.0 经过重构设计正式发布。在 2.0 版本中，Vite 受到 WMR 的启发而引入了基于 Rollup 的底层插件系统，提供了更好的开发体验，并且提升了构建性能，例如全新的 SSR 运行时及依赖预打包方案等。

2022 年 7 月 13 日，Vite 3.0 发布，其采用了 ES 模块的导入方式而可以按需加载模块，使开发者可以快速启动开发服务器，并且在开发过程中修改代码后能够实时更新。同时，Vite 3.x 还引入了一个叫作生产优化模式的功能，可以在构建时自动优化代码，减少构建时间和最终生成的文件大小。同年 12 月，Vite 4.0 发布，其采用了全新的 Rollup 3，可以帮助开发者简化内联资源并提升性能。到目前为止，Vite 已经发布了 5.0 版本，其在不断探索更深层次的前端工程化发展，提供 Rust 化的 Rolldown 前端打包工具。

通过不断地迭代和改进，Vite 已经成为前端开发者钟爱的工具之一，其快速、轻量级的特点使开发者能够更加高效地构建现代化的 Web 应用程序，并享受更好的开发体验，因此，本节将分别介绍 Vite 不同版本的实现方式及对应版本的特色，以期能够让读者全面地了解 Vite 的原理与功能。

1. Vite 1.x

Vite 1.x 并未真正意义上进行发版，其仅仅提供了对 Vue 的基于 ES 模块的开发服务器，可以实现按需编译，提高了开发效率。除此之外，Vite 1.x 还引入了快速热更新，可以在修改代码后立即更新浏览器中的内容，无须手动刷新页面，如图 5-16 所示。

对于开发服务器，其本质是基于 Koa 的一个 Node.js 应用服务器，代码如下：

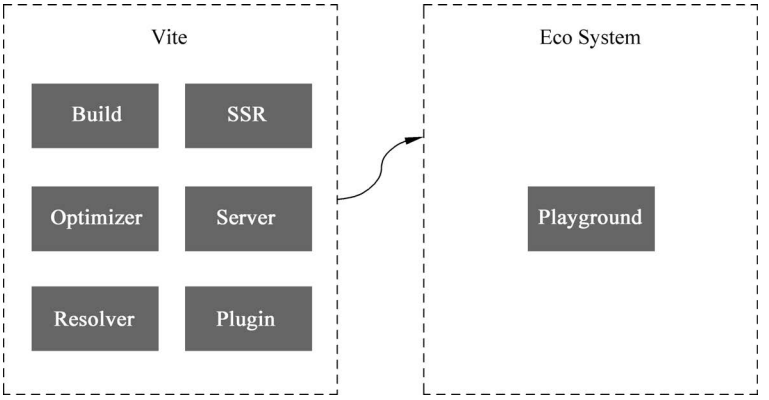

图 5-16　Vite 1.x 原理

```
//第 5 章/vite1.x.ts
export interface ServerPluginContext {
  root: string
  app: Koa < State, Context >
  server: Server
  watcher: HMRWatcher
  resolver: InternalResolver
  config: ServerConfig & { __path?: string }
  port: number
}
export function createServer(config) {
  const app = new Koa()
  const server = resolveServer(config, app.callback());
  const context: ServerPluginContext = {};
  //Koa 中间件
  app.use((ctx, next) => {
    Object.assign(ctx, context)
    return next()
  })
  return server
}
function resolveServer(
  { https = false, httpsOptions = {}, proxy },
  requestListener
) {
  if (!https) {
    return require('http').createServer(requestListener)
  }
  if (proxy) {
    return require('https').createServer(requestListener)
  } else {
```

```
    return require('http2').createSecureServer(requestListener)
  }
}
```

对于非 ES 模块则需进行解析,代码如下:

```
//第 5 章/vite1.x.ts
export function transformCjsImport(
  exp: string,
  id: string,
  resolvedPath: string,
  importIndex: number
): string {
  const ast = parse(exp)[0] as ImportDeclaration
  const importNames: ImportNameSpecifier[] = []
  ast.specifiers.forEach((obj) => {
    if (obj.type === 'ImportSpecifier' && obj.imported.type === 'Identifier') {
      const importedName = obj.imported.name
      const localName = obj.local.name
      importNames.push({ importedName, localName })
    } else if (obj.type === 'ImportDefaultSpecifier') {
      importNames.push({ importedName: 'default', localName: obj.local.name })
    } else if (obj.type === 'ImportNamespaceSpecifier') {
      importNames.push({ importedName: '*', localName: obj.local.name })
    }
  })
  return generateCjsImport(importNames, id, resolvedPath, importIndex)
}

function generateCjsImport(
  importNames: ImportNameSpecifier[],
  id: string,
  resolvedPath: string,
  importIndex: number
): string {
  //如果在一个文件中针对相同的 id 存在多个导入,则防止 CJS 模块名出现重复现象
  const cjsModuleName = makeLegalIdentifier(
    `$viteCjsImport${importIndex}_${id}`
  )
  const lines: string[] = [`import ${cjsModuleName} from "${resolvedPath}";`]
  importNames.forEach(({ importedName, localName }) => {
    if (importedName === '*' || importedName === 'default') {
      lines.push(`const ${localName} = ${cjsModuleName};`)
    } else {
```

```
        lines.push(`const ${localName} = ${cjsModuleName}["${importedName}"];`)
      }
    })
    return lines.join('\n')
  }
```

因此，Vite 1.x 的特点可以总结为按需编译、无捆构建、快速更新。

2. Vite 2.x

Vite 2.x 扩展了对其他框架的支持，包括 React、Preact、Svelte 等。Vite 2.x 还引入了基于 ESBuild 的快速打包，可以极大地提高构建速度，同时也引入了静态资源优化和代码压缩功能，帮助减小生成的文件大小，如图 5-17 所示。

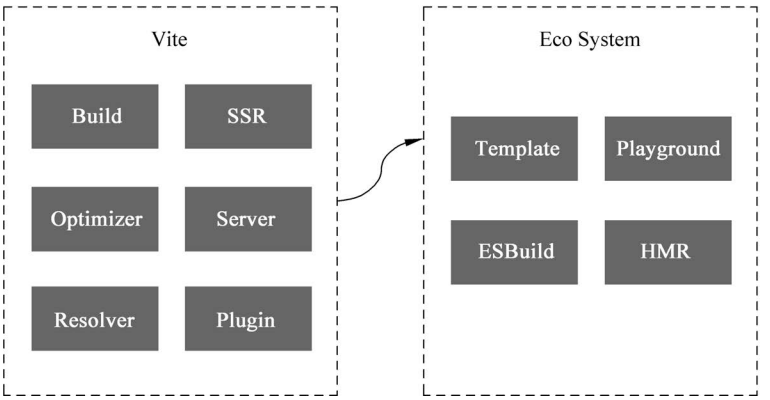

图 5-17　Vite 2.x 原理

Vite 2.x 在打包构建中对于代码转换则是基于 ESBuild 的跨语言转换，代码如下：

```
//第 5 章/vite2.x.ts
//ESBuild 中的 startService 在后边版本中被移除，使用 transform 方法实现
import { startService } from 'esbuild';
const ensureService = async () => {
  return await startService()
}
export async function transformWithEsbuild(code) {
  const service = await ensureService();
  const result = await service.transform(code)
}
export async function transformWithEsbuild(
  code,
  filename,
  options,
) {
  let loader = options?.loader
```

```
  const resolvedOptions = {
    sourcemap: true,
    sourcefile: filename,
    ...options,
    loader,
  }
  const result = await transform(code, resolvedOptions)
  return result;
}
```

注意：ESBuild 是基于 Go 语言实现的前端打包工具，将在后续的章节进行介绍。

因此，Vite 2.x 的特点可以总结为框架扩展、快速打包、资源优化。

3. Vite 3.x

Vite 3.x 进一步提升了构建性能，引入了 Hybrid 模式，可以在开发和构建过程中同时使用预编译和实时编译，提高了打包速度和开发体验。此外，Vite 3.x 还增强了对 TypeScript 的支持，并提供了更好的类型推断和错误提示，如图 5-18 所示。

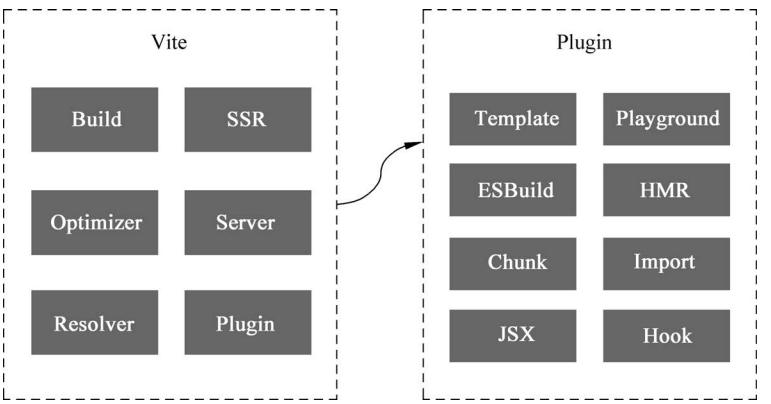

图 5-18　Vite 3.x 原理

对于构建部分，Vite 3.x 会先通过 ESBuild 进行预编译，再通过 Rollup 整体地进行打包构建，代码如下：

```
//第 5 章/vite3.x.ts
async function build(inlineConfig) {return await doBuild(inlineConfig)}
async function doBuild(config) {
  const plugins = [], external = {}, input = config.input;
  const rollupOptions: RollupOptions = {
    input,
    plugins,
    external,
```

```
    }
    const output = [];
    //使用 Rollup 进行构建
    const { rollup } = await import('rollup')
    const bundle = await rollup(rollupOptions)
    const generate = (output) => {
      return bundle['generate'](output)
    }
    return await generate(output)
  }
```

可以说，Vite 是整合了 ESBuild 和 Rollup 这两个打包器进行构建的，其特点可以总结为混合模式、预先编译、类型推断。

4. Vite 4.x

Vite 4.x 继续优化构建速度和性能，其引入了全新的缓存机制，可以更有效地利用缓存，减少重复的构建过程。Vite 4.x 还加强了对框架的支持，提供了更多的插件和工具，方便开发者进行定制，如图 5-19 所示。

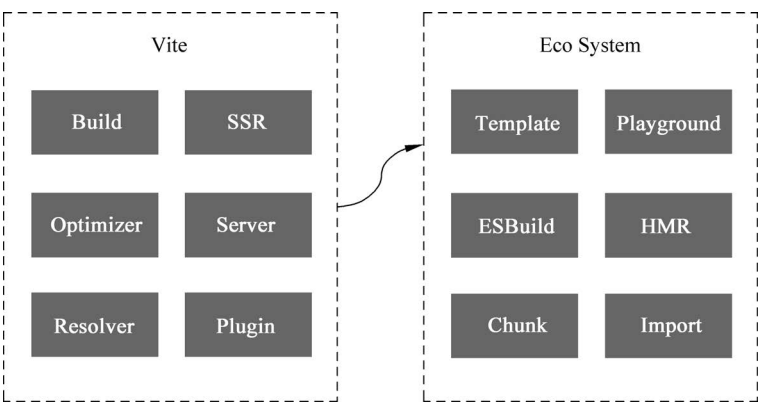

图 5-19　Vite 4.x 原理

Vite 4.x 优化通过 optimizer 模块进行处理，代码如下：

```
//第 5 章/vite4.x.ts
//执行 vite optimize 命令，扫描并优化项目中的依赖关系
export async function optimizeDeps(config) {
  const metadata = {};
  return metadata;
}
```

因此，Vite 4.x 的特点可以总结为缓存机制、性能提高、生态扩展。

5min

5.5 本章小结

本章以前端在开发过程中常见的几大打包器入手，分别介绍了 Webpack、Rollup、Gulp 及 Vite 等内容，也简要地介绍了每个打包工具所适用的场景。为了更好地了解业界打包工具的全貌，本节将通过对比分析已成熟的不同打包器来对本章内容进行总结，见表 5-1。

表 5-1 前端打包工具对比

工具名称	语言	时间	优　点	缺　点
Webpack	JavaScript	2012	模块化/代码分离/高度可配置/开箱即用/插件系统/生态丰富	构建速度慢/体积大/配置复杂/依赖项管理
Gulp	JavaScript	2013	易用/速度快/可扩展/可定制/跨平台/生态丰富	配置复杂/插件质量不一/功能较少/过于灵活
Rollup	JavaScript	2015	Tree Shaking/ES6 模块支持/插件系统/第三方库支持/多种输出格式	复杂性高/CJS 支持不足
ESBuild	Go	2016	极快速/通用/易于使用/高级压缩/静态分析	社区不完善/场景支持弱/配置灵活度低
Parcel	JavaScript	2017	零配置/自动化/易于维护/多种技术栈/快速	生态不完善/配置项少/高级功能少
SWC	Rust	2017	高性能/压缩效果好/最新 ES 标准/支持 TypeScript/易于集成	不稳定/生态薄弱/兼容性差
Nx	TypeScript	2017	高效/可扩展/平台无关/依赖管理	依赖复杂/项目结构固定/配置复杂
Snowpack	JavaScript	2019	直接加载/极速构建/支持生态完善/集成性好/易配置	不支持 CSS 打包/不适用大型项目
Vite	TypeScript	2020	快速开发服务器/热更新/支持多种框架/内置 Rollup/插件系统/简单易用	兼容性较差/生态系统不完善/CJS 模块兼容弱
Rome	Rust	2020	统一 AST/类型检测/零配置/全新工具链/多语言支持	生态薄弱/初级阶段/资源消耗高
WMR	JavaScript	2020	快速开发/热重载/零配置/自动优化/简单易用	JavaScript 支持不完全/缺乏对构建控制/性能受限
Turbopack	Rust	2021	自动计算依赖/快速打包/智能增量编译/内置 AST 转换/Node.js 集成	定制程度低/生态不完善/社区支持弱
Rspack	Rust	2022	极速启动/闪电热更新/兼容 Webpack/内置构建能力/默认生产优化/框架无关	社区生态小/兼容性差

最后，前端打包器在前端工程化领域扮演着重要的角色，其是现代在前端开发中不可或缺的工具之一。通过打包器，前端工程师可以将各种前端资源进行整合并优化以提高网页的加载速度和性能表现，包括模块化开发、资源优化、提高开发效率、跨浏览器兼容等方面。

从第 6 章开始，本书将会对前端工程中的规范标准进行阐述。“不以规矩，不能成方圆”，所有的工程项目都有其各自的规范，前端工程方案同样离不开规范的约束，希望各位读者通过规范篇章的学习，能够根据各自团队的特点制定出相应的团队规范。