结束语：课程回顾和未来展望

发布于 2022-05-09

在 2021 年 2 月，尤大正式推出了 Vite 2.0 版本，可以说是 Vite 的一个重要转折点，自此之后 Vite 的用户量发生了非常迅速的增长，很快达到了每周 100 万的 npm 下载量。同时，Vite 的社区也越来越活跃，目前已经形成非常庞大的社区生态(详情可见 Github 地址)，给整个前端领域带来了诸多的改变，如:

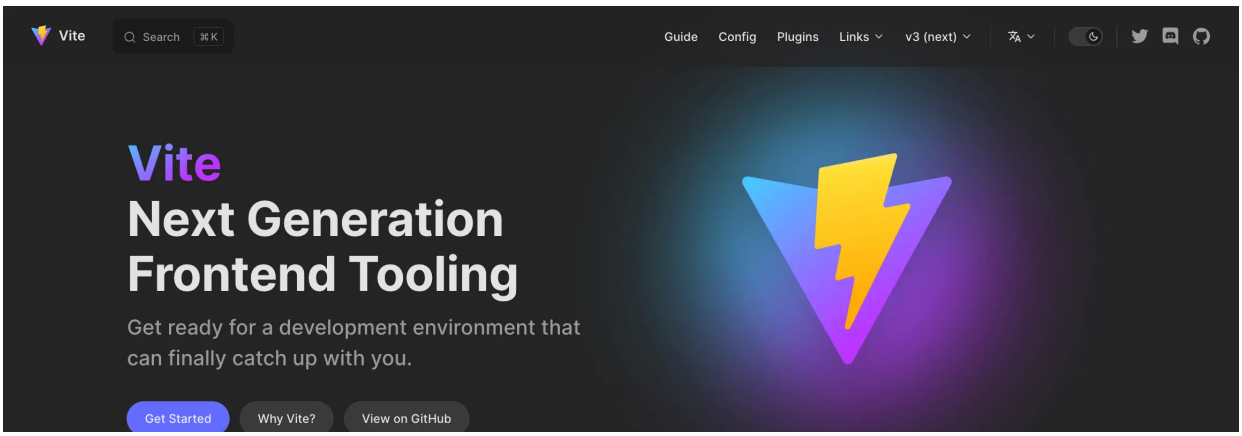
- Nuxt 3、SvelteKit、Astro、StoryBook 等在内的各大前端框架已经将 Vite 作为内置的构建方案。

- 基于 Vite 的测试工具 Vitest 诞生，成为替代 Jest 的新一代测试方案。

如今已经 2022 年 7 月，距离 v2 版本已经发布了 16 个月的时间，Vite 正式推出 3.0 版本，接下来就给大家介绍一下 Vite 3.0 带来的一些改变以及未来的规划。

一、全新的 VitePress 文档

对于用户侧来说，谈到框架的更新，文档自然是最重要的部分。现在你可以直接去 vitejs.dev 站点体验到 v3 版本的文档，目前文档同样是使用 VitePress 进行搭建。下面是暗黑模式下的一张截图:



怎么样，是不是比以前更加好看了呢?

不光是 Vite，也有 Vite 生态中其它的一些项目使用 VitePress 进行文档站点的搭建，比如 Vitest, vite-plugin-pwa 以及 VitePress 自身的文档，我也十分推荐大家使用 VitePress 作为自己的文档建站方案之一。

如果你需要查看 Vite 2.0 的文章，也可以访问 v2.vitejs.dev。

二、开发阶段的更新

1. CLI 的更新

在执行 `vite` 命令启动项目时，终端的界面和之前会有所不同，而更重要的是，为了避免 Vite 开发服务的端口和别的应用冲突，默认的端口号从之前的 `3000` 变成了 `5173` 。

```
VITE v3.0.0  ready in 320 ms

➜  Local:   http://127.0.0.1:5173/
➜  Network: use --host to expose
```


2. 开箱即用的 WebSocket 连接策略

Vite 2 中有存在一个痛点，即在存在代理的情况下(比如 Web IDE)需要我们手动配置 WebSocket 使 HMR 生效。目前 Vite 内置了一套更加完善的 WebSocket 连接策略，自动满足更多场景的 HMR 需求。

3. 服务冷启动性能提升

Vite 3.0 在服务冷启动方面做了非常多的工作，来最大程度提升项目启动的速度。

首先我们来盘点一下 Vite 2.x 阶段服务冷启动的一些问题。

从 Vite 2.0 到 2.9 版本之前，Vite 会在服务启动之前进行依赖预构建，也就是使用 Esbuild 将项目中使用到的依赖扫描出来(Scan)，然后分别进行一次打包(Optimize)。

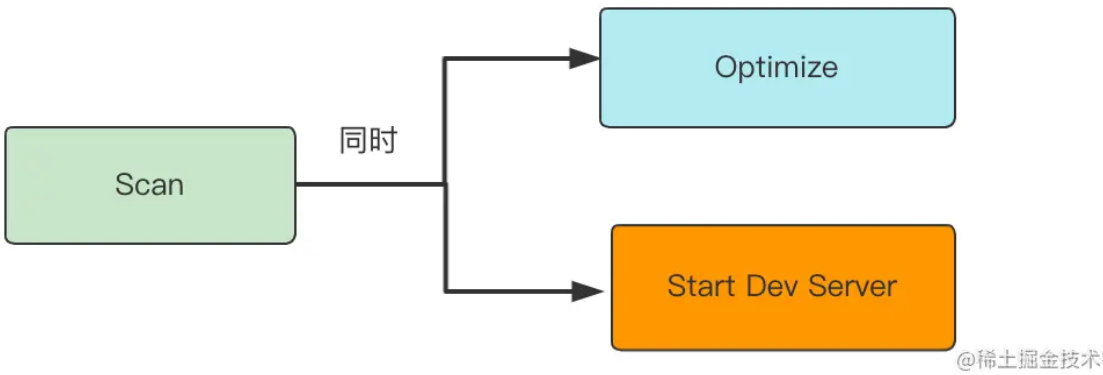
这样会造成两个问题:

- 依赖预构建会阻塞 Dev Server 启动，但其实不阻塞的情况下，Dev Server 也可以正常启动。
- 当某些 Vite 插件手动注入了 import 语句，比如调用 `babel-plugin-import` 添加 `import Button from 'antd/lib/button'` ，就会导致 Vite 的二次预构建，因为 `antd/lib/button` 的引入代码由 Vite 插件注入，属于 Dev Server 运行时发现的依赖，冷启动阶段无法扫描到。

所谓的二次预构建包含两个步骤，一是需要将所有的依赖全量预构建，二是由于依赖更新，页面需要进行 reload，加载最新的依赖代码。这样会导致 Dev Server 性能明显下降，尤其是在新增依赖较多的场景下，很容易出现浏览器 卡住 的情况。因此二次预构建也是需要极力避免的。当时 vite-plugin-optimize-persist 就是为了解决二次预构建带来的问题，通过持久化的方式记录 Dev Server 运行时扫描到的依赖，从而让首次预构建便可以感知到，避免二次预构建的发生。

到了 2.9 版本，Vite 将预构建的逻辑做了一次整体的重构，最后的效果是下面这样的:

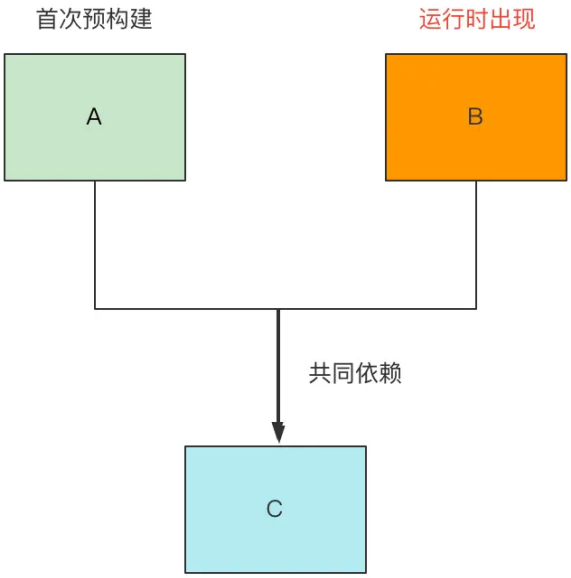
- Dev Server 启动后预构建(Optimize 阶段)在后台执行，也就是预构建不再阻塞 Dev Server 的启动，只需要等待 Scan 阶段完成，不过通常这个阶段的开销非常小。



- 如果某些依赖是 Dev Server 运行时才发现的，那么 Vite 会尽可能地复用已有预构建产物，尽量不进行 page reload。

> 具体实现大家可以去查看这个 PR

那问题就完全解决了吗？其实并不是，在某些场景下，Vite 仍然不可避免地需要二次预构建。如下面的这个例子:



A 和 B 都是项目的第三方依赖，它们也同时依赖 C。那么当 Vite 预构建 A 的时候，将会 A 和 C 一起进行打包。但 Vite 在运行时发现了依赖 B，而 A 和 B 需要共享 C 的代码，这样 C 的代码可能就会被抽离成一个公共的 chunk，因此之前 A 的预构建产物可能就发生变化了，那么此时 Vite 必须要强制刷新页面，让浏览器使用最新的预构建产物。这仍然是一个二次预构建(所有依赖再次打包 + page reload)的过程。

总体而言，2.9 版本解决了预构建阻塞服务启动的问题，但并没有完全解决二次预构建的问题。

但在 Vite 3.0，二次预构建的问题也得到了根本的解决。那 Vite 3.0 是如何做到的呢?

核心的解决思路在于 延迟处理 ，即把预构建的行为延迟到**页面加载的最后阶段**进行，此时 Vite 已经编译完了所有的源文件，可以准确地记录下所有需要预构建的依赖(包括

的 Vite 已经编译完了所有的源文件，可以准确地记录下所有需要预构建的依赖(包括 Vite 插件添加的一些依赖)，然后统一进行预构建，将预构建的产物响应给给浏览器即可。

> 依赖预构建的代码在 Vite 中先后重构了多次，目前的版本实现比较复杂，后续会单独写一篇文章讨论实现细节。

因此，与 Vite 2.0 相比，Vite 3.0 在冷启动阶段所做的优化主要有两个方面:

- 预构建不再阻塞 Dev Server 的启动，真正做到服务秒启动的效果；
- 从根本上防止二次预构建的发生。

4. import.meta.glob 语法更新

Vite 3.0 中重写对 `import.meta.glob` 的实现进行了重写，支持了更加灵活的 glob 语法，增加了如下的一些特性:

- 多种模式匹配:

```
import.meta.glob(["./dir/*.js", "./another/*.js"]);
```

- 否定模式(!):

```
import.meta.glob(["./dir/*.js", "!**/bar.js"]);
```

- 命名导入，可以更好地做到 Tree Shaking:

```
import.meta.glob("./dir/*.js", { import: "setup" });
```

- 自定义 query 参数:

```
import.meta.glob("./dir/*.js", { query: { custom: "data" } });
```

- 指定 eager 模式，替换掉原来 `import.meta.globEager`：

```
import.meta.glob("./dir/*.js", { eager: true });
```

三、生产阶段的更新

1. SSR 产物默认使用 ESM 格式

在当下的社区生态中，众多 SSR 框架已经在使用 ESM 格式作为默认的产物格式。Vite 3.0 也积极拥抱社区，支持 SSR 构建默认打包出 ESM 格式的产物。

2. Relative Base 支持

Vite 3.0 正式支持 Relative Base(即配置 `base: ''`)，主要用于构建时无法确定 base 地址的场景。

四、实验性功能

1. 更细粒度的 base 配置

在某些场景下，我们需要将不同的资源部署到不同的 CDN 上，比如将图片部署到单独的 CDN，和 JS/CSS 的部署服务区分开来。但 2.x 的版本仅支持统一的部署域名，即 `base` 配置。在 3.0 中，你可以通过 `renderBuiltUrl` 进行更细粒度的配置:

```
{
  experimental: {
    renderBuiltUrl: (filename: string, { hostType: 'js' | 'css' | 'html' }) => {
      if (hostType === 'js') {
        return { runtime: `window.__toCdnUrl(${JSON.stringify(filename)})` }
      } else {
        return 'https://cdn.domain.com/assets/' + filename
      }
    }
  }
}
```

> 目前该配置项还不稳定，可能会在之后的 minor 版本修改。具体文档见 vitejs.dev/guide/build...

2. Esbuild 预构建用于生产环境

这应该是 Vite 架构上非常大的一个改动: 将原来仅仅用于开发阶段的依赖预构建功能应用在生产环境。在 Vite 2.x 中，开发阶段使用 Esbuild 来打包依赖，而在生产环境使用

用生产环境。在 Vite 2.x 中，开发阶段使用 Esbuild 来打包依赖，而生产环境使用 Rollup 进行打包，用 `@rollupjs/plugin-commonjs` 来处理 cjs 的依赖，这样做会导致依赖处理的不一致问题，造成一些生产构建中的 bug。

但 Vite 3.0 中支持通过配置将 Esbulid 预构建同时用于开发环境和生产环境，仅添加 `optimizeDeps.disabled: false` 的配置即可。不过这个改动确实比较大，Vite 团队不打算将此作为 v3 的正式更新内容，而是一个实验性质的功能，不会默认开启。

顺便提一句，Rollup 将在接下来的几个月发布 v3 的大版本，要知道，Rollup 2.0 发布至今已经过去 2 年多的时间了，无论是 Rollup 还是 Vite 来讲，这都是一次非常重大的变更。由于 Vite 的架构非常依赖 Rollup，在 Rollup 发布 v3 之后，Vite 也将跟随着发布 Vite 的第 4 个 major 版本。所以，Vite 4.0 的到来也不远啦：）

五、仓库内部的变化

除了本身功能上的演进，Vite 的仓库本身也产生了不少的变化，从中我们也能了解到社区的一些动向:

- 不再支持 Nodejs 12，需要 Node.js 14.18+ 的版本。
- 单元测试和 E2E 测试从 Jest 完全迁移到 Vitest，一方面 Vitest 更快、体验更好，另一方面也能在 Vite 这样大型的仓库完善 Vitest 的生态，进一步提升 Vitest 稳定性。
- VitePress 文档部分也参与 CI 流程。
- 包管理器 pnpm 迁移至 v7。
- 不管是 `Vite` 本身的包还是 E2E 中测试的项目，都在 package.json 中声明 `type: "module"`，即 Pure ESM 包，对外提供 ESM 格式的产物，将社区 Pure ESM 的趋势又推动了一步。
- 官方所有的 Vite 插件都采用 `unbuild` (新一代库构建工具) 进行构建， `pluin-vue-jsx` 和 `plugin-legacy` 均迁移到了 TS 上。
- 包体积优化。3.0 进一步优化 Vite 本身的产物和 node_modules 体积，将 `terser` 和 `node-forge` 的依赖移除，让用户进行按需安装(`node-forge` 的功能是实现 https 证书生成，可用 `@vitejs/plugin-basic-ssl` 插件替代)，效果如下:

	Publish Size	Install Size
Vite 2.9.14	4.38MB	19.1MB
Vite 3.0.0	3.05MB	17.8MB
Reduction	-30%	-7%

不得不说在自身包体积的优化方面， Vite 还是做的很细致的，这也是很多库开发者忽视的一点，有时候加个插件就得安装动辄上百 MB 的依赖，导致项目的 node_modules 最后变得非常臃肿，此时不妨学习一下 Vite 是怎么优化自身体积的。

六、未来规划

首先在 Vite 3.0 发布之后会重点保证 3.0 的稳定性，解决目前的一系列 issue。

其次，Rollup 团队将在接下来的几个月发布新的 major 版本，Vite 将持续跟进，紧接着发布 v4 版本，并在 v4 版本中将目前的一些实践性功能稳定下来。

小结

Vite 3.0 带来了一些比较大的架构变动，比如依赖预构建的重构、支持生产环境 Esbuild 预打包依赖以及全面支持 Pure ESM，当然也有一些比较小的 break change 在这个版本集中发布，比如 import.meta.glob 语法的变更。

总之，在这一年多的时间里，Vite 团队做了非常多的功能改进和架构升级，目前的 Github Star 已经达到了 44 k+，并且还在持续维护。与此同时，Vite 的社区生态也逐步完善，比如 Vitest、VitePress、丰富的社区插件以及众多内置 Vite 的社区框架等等，可以预见的是，Vite 将在未来的很长一段时间内继续发展，持续迭代，提供更好的用户体验，成为下一代前端工具链。

上一篇：手写 Bundler: 实现代码打包、 Tree Shaking