chlorine 氯
Posted on 2023年3月31日
发布于 2023年3月31日

How does pnpm work pnpm 是如何工作的

#javascript JavaScript （英语） #npm #frontend 前端 #programming 编程

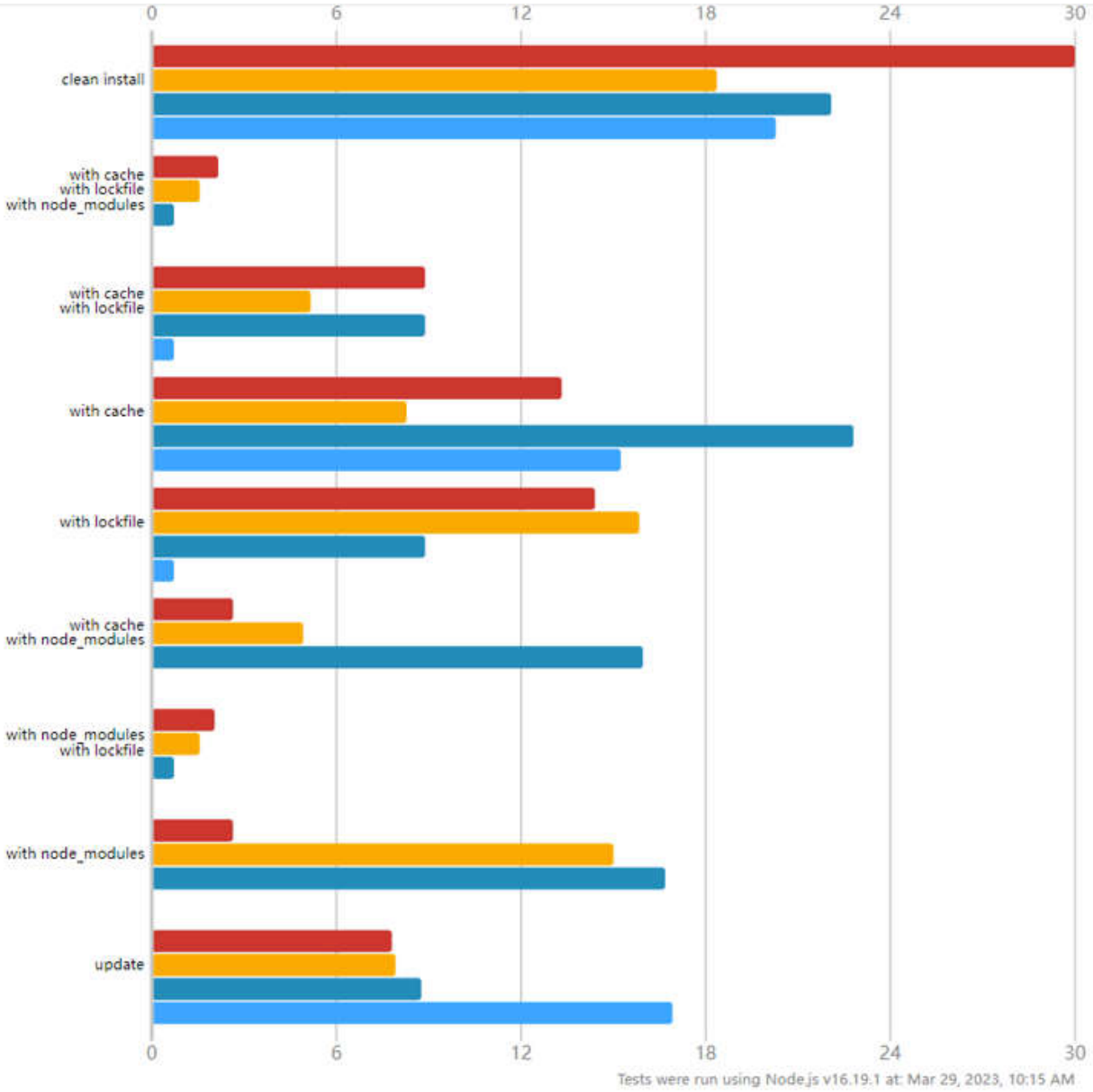
As one of the very popular package managers, pnpm is mainly characterized by fast speed and saving disk space. I will introduce how pnpm works to help you understand the principle of pnpm.

作为非常流行的包管理器之一，pnpm 的主要特点是速度快，节省磁盘空间。我将介绍 pnpm 的工作原理，帮助您理解 pnpm 的原理。

Introduction 介绍

The meaning of pnpm is performant npm. From the benchmarks in pnpm website, we can see in many scenarios pnpm has good performance advantages compared with npm/yarn/yarn_pnp.

pnpm 的含义是高性能 npm。从 pnpm 网站的基准测试中，我们可以看到在很多场景中 pnpm 相比 npm/yarn/yarn_pnp 具有良好的性能优势。



Directory structure of node_modules
node_modules 的目录结构

Nested structure　嵌套结构

In earlier versions of npm@2, corresponding to Node.js 4.x and previous version, node_modules is a nested structure while installing.

在早期版本的 npm@2 中，对应于 Node.js 4.x 和早期版本，node_modules 是安装时的嵌套结构。

A simple case here, both demo-foo and demo-baz depend on demo-bar. When demo-foo and demo-baz are installed in the same repository, we can get the

demo-baz 安装在同一个仓库中时，我们可以得到以下node_modules结构：

```
node_modules
└─ demo-foo
   ├─ index.js
   ├─ package.json
   └─ node_modules
      └─ demo-bar
         ├─ index.js
         └─ package.json
└─ demo-baz
   ├─ index.js
   ├─ package.json
   └─ node_modules
      └─ demo-bar
         ├─ index.js
         └─ package.json
```

Although the directory structure is relatively clear at this time, each dependent package will have its own node_modules directory, and the same dependency has not been reused. For the above example, the same dependency demo-bar has been installed twice.

虽然此时目录结构比较清晰，但每个依赖的 package 都会有自己的node_modules目录，同样的依赖没有被复用。在上面的例子中，同一个依赖 demo-bar 已经安装了两次。

Another problem is the Maximum Path Limitation of windows. In some complex cases when the project has a deep dependency level, the dependent path often exceeds the length limit.

另一个问题是窗口的最大路径限制。在一些复杂情况下，当项目具有深层依赖级别时，依赖路径通常会超过长度限制。

Flat structure 扁平结构

In order to solve the above problems, yarn proposed a flat structure design: flattening all dependencies in node_modules. And the implementation of the later

为了解决上述问题，yarn 提出了一种扁平化结构设计：将 node_modules 中的所有依赖扁平化。和后面的 npm v3 版本的实现类似，所以使用 yarn 或者 npm@3+ 安装上面的例子，你会得到如下扁平的目录结构：

```
node_modules
└─ demo-bar
   ├─ index.js
   └─ package.json
└─ demo-baz
   ├─ index.js
   └─ package.json
└─ demo-foo
   ├─ index.js
   └─ package.json
```

In addition, for different versions of the same dependency in this way, only one of them will be hoisted, and the remaining versions will still be nested in the corresponding packages.

此外，对于以这种方式对同一依赖的不同版本，只会提升其中一个，其余版本仍会嵌套在对应的包中。

For instance, if we upgrade the above demo-bar to v1.0.1 (its dependency demo-foo is also v1.0.1), you will get the following structure, which version will be hosited to the top depends on the order of installation:

例如，如果我们把上面的 demo-bar 升级到 v1.0.1（它的依赖 demo-foo 也是 v1.0.1），你会得到这样的结构，哪个版本会放在最上面，取决于安装顺序：

```
node_modules
└─ demo-bar
   ├─ index.js
   └─ package.json
└─ demo-baz
   ├─ index.js
   ├─ package.json
```

```
        └─ package.json
└─ demo-foo
   ├─ index.js
   ├─ package.json
   └─ node_modules
      └─ demo-bar
         ├─ index.js
         └─ package.json
```

Problems with flat structures
平面结构的问题

The flat solution is not perfect, and accompanies with some new problems:

扁平解决方案并不完美，并且伴随着一些新问题：

Phantom dependencies　虚拟依赖项

Phantom dependencies means the dependencies that are not declared in package.json but can be directly used in your project. This issue is caused by the flat structure, and the dependencies of the dependencies will also be hoisted to the top level of node_modules, so that you can reference it directly in the project. And if some day this sub-dependency is no longer a dependency of the reference package, there will be problems with the references in the project.

Phantom dependencies 是指未在 package.json 中声明但可以直接在项目中使用的依赖项。这个问题是由扁平结构引起的，依赖的依赖也会被提升到node_modules的顶层，这样就可以直接在项目中引用了。如果有一天，这个子依赖项不再是引用包的依赖项，那么项目中的引用就会出现问题。

For example, in the project containing demo-foo and demo-baz as the dependencies, demo-bar also appears in node_modules as a dependent dependency:

例如，在包含 demo-foo 和 demo-baz 作为依赖项的项目中，demo-bar 也作为依赖依赖项出现在 node_modules 中：

```
node_modules
```

```
└─ demo-baz
   ├─ index.js
   └─ package.json
└─ demo-foo
   ├─ index.js
   └─ package.json
```

Npm doppelgangers Npm 分身

NPM doppelgangers refer to different versions of the same dependency, due to the hoist mechanism, only one version will be hoisted, and other versions may be installed repeatedly. The same example as above, when we upgrade demo-bar to v1.0.1, the 1.0.0 version that demo-baz and demo-foo depends on will be repeatedly installed in a nested way:

NPM 分身是指同一依赖关系的不同版本，由于提升机构的原因，只会吊装一个版本，其他版本可能会重复安装。和上面一样的例子，当我们把 demo-bar 升级到 v1.0.1 时，demo-baz 和 demo-foo 依赖的 1.0.0 版本会以嵌套的方式重复安装：

```
node_modules
└─ demo-bar // v1.0.1
   ├─ index.js
   └─ package.json
└─ demo-baz
   ├─ index.js
   ├─ package.json
   └─ node_modules
      └─ demo-bar // v1.0.0
         ├─ index.js
         └─ package.json
└─ demo-foo
   ├─ index.js
   ├─ package.json
   └─ node_modules
      └─ demo-bar // v1.0.0
         ├─ index.js
         └─ package.json
```

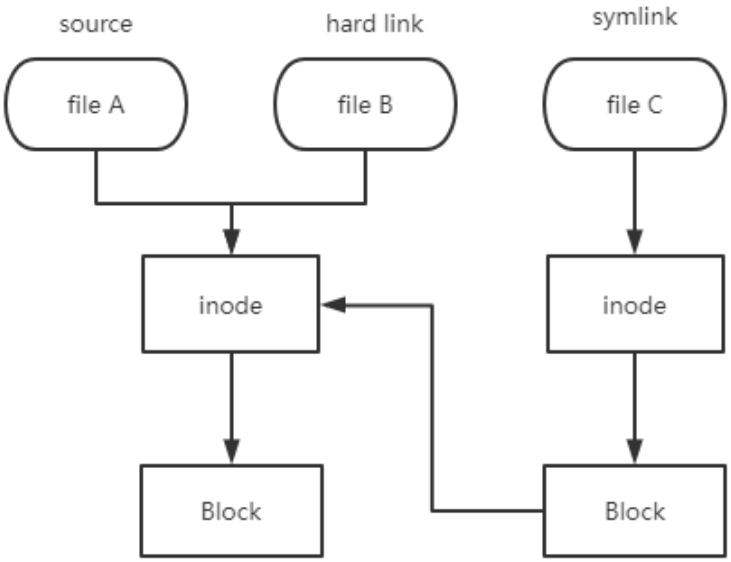
First of all, pnpm installs the dependencies to the global store, and then use symbolic link and hard link to organize directory structure. It links the global dependencies into the project, and links the direct dependencies into top level of node_modules directory. All dependencies are flattened under the `node_modules/.pnpm` directory, which realizes the global dependencies shared of all projects, and solves the problem of phantom dependencies and NPM doppelgangers.

首先，pnpm 将依赖项安装到全局存储中，然后使用符号链接和硬链接来组织目录结构。它将全局依赖项链接到项目中，并将直接依赖项链接到 node_modules 目录的顶层。所有依赖都扁平化在 `node_modules/.pnpm` 目录下，实现了所有项目的全局依赖共享，解决了幻影依赖和 NPM 分身的问题。

Symbolic link and hard link
符号链接和硬链接

Link is the way of file sharing in the operating system, where the one is symbolic link, also known as soft link, and the onther one is hard link. From the point of view of use, there is no difference between them, both supporting reading and writing, and if it's an excutable file, it can also be excuted directly. The main difference is the core principle:

链接是操作系统中文件共享的方式，其中一是符号链接，也称为软链接，二是硬链接。从使用的角度来看，它们之间没有区别，都支持读写，如果是可执行的文件，也可以直接执行。主要区别在于核心原则：



the same index node

硬链接不会创建新的索引节点，源文件和硬链接指向同一个索引节点

- Hard links don't support directories, only file level, and don't support cross disk partitions

  硬链接不支持目录，仅支持文件级，并且不支持跨磁盘分区

- The file is not actually deleted until the source file and all hard links are deleted

  在删除源文件和所有硬链接之前，文件实际上不会被删除

Symbolic link 符号链接

- The symbolic link stores the path of the source file, pointing to the source file, similar to the shortcut of Windows

  符号链接存储源文件的路径，指向源文件，类似于 Windows 的快捷方式

- The symbolic link supports directories and files, which are different files from source files. It has different inode values and different file types, so the symbolic link can be accessed across partitions

  符号链接支持目录和文件，它们是与源文件不同的文件。它具有不同的 inode 值和不同的文件类型，因此可以跨分区访问符号链接

- After deleting the source file, the symbolic link still exists, but the source file cannot be accessed through it

  删除源文件后，符号链接仍然存在，但无法通过它访问源文件

How to create links through command
如何通过命令创建链接

```
# symbolic ink
ln -s myfile mysymlink

# hard link
ln myfile myhardlink
```

How does pnpm work pnpm 是如何工作的

首先，pnpm 会将依赖项安装到当前分区的 `<home dir>/.pnpm-store` 中。可以通过以下命令获取当前商店位置：

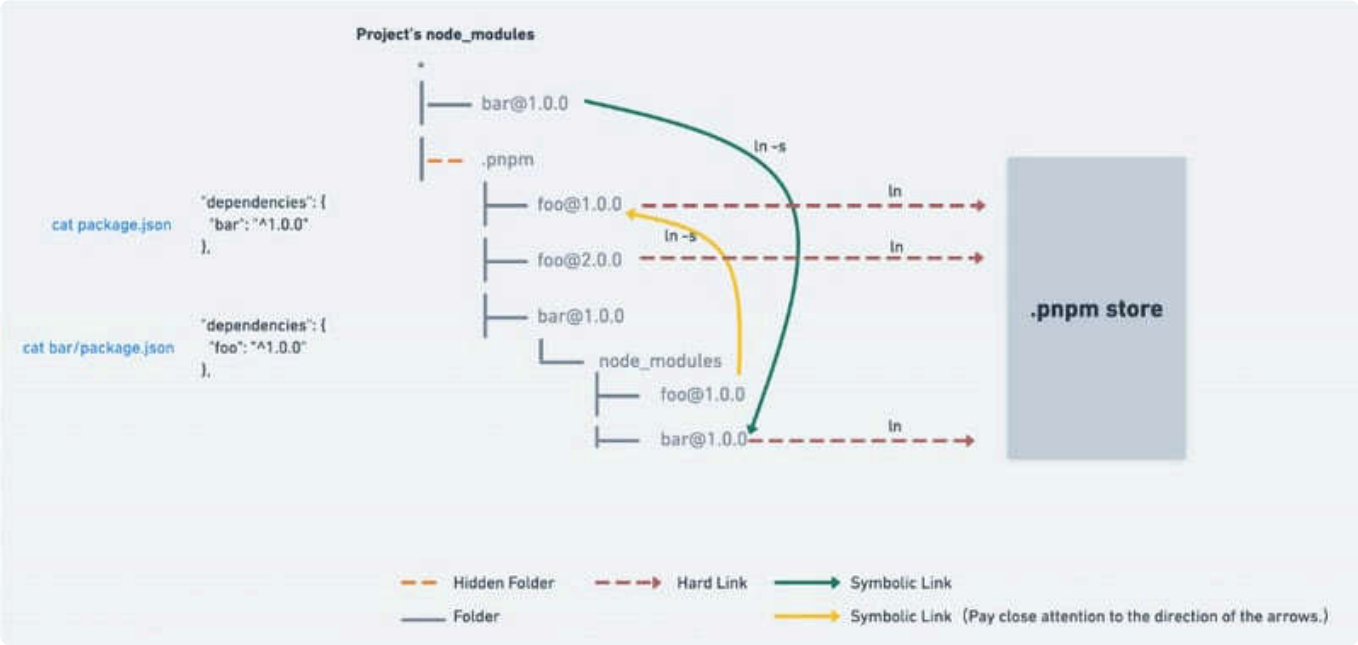
```
pnpm store path
```

And then hard link the required packages from `node_modules/.pnpm` into the store path. Finally, symbolically link the top-level dependencies and dependent dependencies in node_modules to `node_modules/ .pnpm`, an example that depends on demo-foo@ 1.0.1 and demo-baz@ 1.0.0, the node_modules structure is:

然后将所需的包从 `node_modules/.pnpm` 硬链接到 store 路径。最后，将 node_modules 中的顶级依赖和依赖node_modules符号化地链接到 `/ .pnpm`，一个依赖 demo-foo@ 1.0.1 和 demo-baz@ 1.0.0 的例子，node_modules结构为：

```
node_modules
└─ .pnpm
   └─ demo-bar@1.0.0
      └─ node_modules
         └─ demo-bar -> <store>/demo-bar
   └─ demo-bar@1.0.1
      └─ node_modules
         └─ demo-bar -> <store>/demo-bar
   └─ demo-baz@1.0.0
      └─ node_modules
         ├─ demo-bar -> ../../demo-bar@1.0.0/node_modules/demo-bar
         └─ demo-baz -> <store>/demo-baz
   └─ demo-foo@1.0.1
      └─ node_modules
         ├─ demo-bar -> ../../demo-bar@1.0.1/node_modules/demo-bar
         └─ demo-foo -> <store>/demo-foo
└─ demo-baz -> ./pnpm/demo-baz@1.0.0/node_modules/demo-baz
└─ demo-foo -> ./pnpm/demo-baz@1.0.1/node_modules/demo-foo
```

以下是 pnpm 网站的屏幕截图，可帮助您更好地了解符号链接和硬链接在项目结构中的组织方式：



For the actual usage of the link in pnpm, the following is the relevant source code:

对于 pnpm 中链接的实际使用，以下是相关源码：

```
function createImportPackage (packageImportMethod?: 'auto' | 'hardlink' | 'copy' | '
  // this works in the following way:
  // - hardlink: hardlink the packages, no fallback
  // - clone: clone the packages, no fallback
  // - auto: try to clone or hardlink the packages, if it fails, fallback to copy
  // - copy: copy the packages, do not try to link them first
  switch (packageImportMethod ?? 'auto') {
  case 'clone':
    packageImportMethodLogger.debug({ method: 'clone' })
    return clonePkg
```

```
    packageImportMethodLogger.debug({ method: 'hardlink' })
    return hardlinkPkg.bind(null, linkOrCopy)
  case 'auto': {
    return createAutoImporter()
  }
  case 'clone-or-copy':
    return createCloneOrCopyImporter()
  case 'copy':
    packageImportMethodLogger.debug({ method: 'copy' })
    return copyPkg
  default:
    throw new Error(Unknown package import method </span><span class="p">${</span><s
  }
}
```

Other abilities 其他能力

At present pnpm can be installed and used out of Node.js runtime. And pnpm can also manage Node.js version through pnpm env, similar to nvm. For a full feature

来管理Node.js版本，类似于 nvm。有关与 npm/yarn 的完整功能比较，请参阅 feature-comparison

Limitations 局限性

- Due to compatibility issues with symbolic links in certain scenarios, pnpm cannot currently be used on applications deployed on Electron and Lambda, as outlined in: discussion

  由于某些情况下符号链接的兼容性问题，pnpm 目前不能用于部署在 Electron 和 Lambda 上的应用程序，如：讨论

By adding `node-linker=hoisted` to .npmrc, a flat node_modules directory without symbolic links can be created, which is similar to the directory structure created via npm/yarn.

通过将 `node-linker=hoisted` 添加到 .npmrc 中，可以创建一个没有符号链接的平面 node_modules 目录，类似于通过 npm/yarn 创建的目录结构。

- As the same store is shared globally, modifying the contents within node_modules will directly affect the corresponding content in the global store, which will also have an impact on other projects.

  由于同一个 store 是全局共享的，修改 node_modules 内的内容会直接影响全局 store 中的相应内容，也会对其他项目产生影响。

For the above issue, the most recommended approach is to use clone(copy-on-write). By default, multiple references point to the same file, and only when the user needs to modify it, a copy is made so that it will not affect other references that are reading the content of the source file.

对于上述问题，最推荐的方法是使用 clone（copy-on-write）。默认情况下，多个引用指向同一个文件，并且仅当用户需要修改该文件时，才会创建一个副本，以便它不会影响正在读取源文件内容的其他引用。

However, not all operating systems support this. By default, pnpm attempts to use clone. If it is not supported, it falls back to using hard link. You can also manually set the package reference method by specifying package-import-method in your .npmrc

支持，则回退到使用硬链接。您还可以通过在 .npmrc 配置文件中指定 package-import-method 来手动设置包引用方法。

- For other limitations, see: https://pnpm.io/limitations

  有关其他限制，请参阅： https://pnpm.io/limitations

Other tools 其他工具

- bun: https://github.com/oven-sh/bun

  包子： https://github.com/oven-sh/bun

  - bun is a JS runtime written in Zig. It also offers a package management tool, but it may encounter some compatibility issues.

    bun 是用 Zig 编写的 JS 运行时。它还提供了一个包管理工具，但可能会遇到一些兼容性问题。

- Volt：https://github.com/dimensionhq/volt

  电压：https://github.com/dimensionhq/volt

  - volt is a Node.js package manager written in Rust, which is known for its extremely fast performance. It is currently in beta, but it has not been updated for several months.

    volt 是一个用 Rust 编写的 Node.js 包管理器，以其极快的性能而闻名。它目前处于测试阶段，但已经几个月没有更新了。

- tnpm: https://github.com/cnpm/npminstall

  TNPM：https://github.com/cnpm/npminstall

  - tnpm implements a virtual mapped directory by utilizing the features of Filesystem in Userspace(FUSE) and OverlayFS.

    tnpm 通过利用 Filesystem in Userspace（FUSE）和 OverlayFS 的功能实现一个虚拟映射目录。

Reference 参考

- Flat node_modules is not the only way

  平node_modules并不是唯一的方法

- In-depth of tnpm rapid mode - how we managed to be 10 second faster than pnpm

  深入了解 tnpm 快速模式 - 我们如何设法比 pnpm 快 10 秒
- Complete code samples 完整的代码示例

Top comments (0)

chlorine 氯

Just code for fun. 只是为了好玩而编码。

JOINED 加入
2022年11月17日

More from chlorine

更多来自 chlorine 的相关产品

Execute E2E Test Cases Using Natural Language with Intelli-Browser

通过 Intelli-Browser 使用自然语言执行 E2E 测试用例

Create up to 10 Postgres Databases on Neon's free plan.

在 Neon 的免费计划中创建多达 10 个 Postgres 数据库。

If you're starting a new project, Neon has got your databases covered. No credit cards. No trials. No getting in your way.

如果您正在开始一个新项目，Neon 已经涵盖了您的数据库。没有信用卡。没有试验。不要妨碍你。