



nix.dev

Nix.dev 贡献者

Jan 01, 2025

CONTENTS

| | |
|---------------------------------|----|
| 1 安装 Nix | 1 |
| 1.1 验证安装 | 2 |
| 2 教程 | 3 |
| 2.1 第一步 | 3 |
| 2.1.1 临时 shell 环境 | 3 |
| 2.1.2 可复现的解释型脚本 | 7 |
| 2.1.3 使用 shell.nix 声明式 shell 环境 | 9 |
| 2.1.4 实现可复现性：固定 Nixpkgs 版本 | 12 |
| 2.2 Nix 语言基础 | 13 |
| 2.2.1 概述 | 13 |
| 2.2.2 名称与值 | 16 |
| 2.2.3 函数 | 26 |
| 2.2.4 函数库 | 32 |
| 2.2.5 杂质 | 35 |
| 2.2.6 推导 | 37 |
| 2.2.7 示例解析 | 38 |
| 2.2.8 参考文献 | 40 |
| 2.2.9 后续步骤 | 40 |
| 2.3 使用Nix打包现有软件 | 41 |
| 2.3.1 引言 | 41 |
| 2.3.2 你的第一个包 | 42 |
| 2.3.3 带依赖项的包 | 45 |
| 2.3.4 查找软件包 | 48 |
| 2.3.5 修复构建失败 | 51 |
| 2.3.6 构建成功 | 53 |
| 2.3.7 参考文献 | 53 |
| 2.3.8 后续步骤 | 53 |
| 2.4 使用 callPackage 实现包参数与覆写 | 53 |
| 2.4.1 概述 | 53 |
| 2.4.2 自动函数调用 | 54 |
| 2.4.3 参数化构建 | 55 |
| 2.5 覆盖机制 | 56 |
| 2.5.1 相互依赖的包集 | 57 |
| 2.5.2 概要 | 58 |
| 2.5.3 参考文献 | 58 |
| 2.5.4 后续步骤 | 58 |
| 2.6 处理本地文件 | 58 |
| 2.6.1 文件集 | 59 |
| 2.6.2 示例项目 | 60 |
| 2.6.3 向Nix存储添加文件 | 60 |
| 2.6.4 差异 | 61 |
| 2.6.5 缺失文件 | 63 |
| 2.6.6 联合（显式排除文件） | 64 |

| | |
|--|------------|
| 2.6.7 过滤 | 65 |
| 2.6.8 联合（显式包含文件） | 65 |
| 2.6.9 Git跟踪的匹配文件 | 66 |
| 2.6.10 交集 | 67 |
| 2.6.11 结论 | 68 |
| 2.7 交叉编译 | 68 |
| 2.7.1 你需要什么？ | 68 |
| 2.7.2 平台 | 68 |
| 2.7.3 什么是目标平台？ | 68 |
| 2.7.4 确定主机平台配置 | 69 |
| 2.7.5 使用 Nix 选择主机平台 | 69 |
| 2.7.6 指定主机平台 | 70 |
| 2.7.7 首次交叉编译 | 71 |
| 2.7.8 Hello World 实例的实际交叉编译 | 71 |
| 2.7.9 使用交叉编译器的开发环境 | 72 |
| 2.7.10 后续步骤 | 73 |
| 2.8 模块系统 | 73 |
| 2.8.1 你需要什么？ | 74 |
| 2.8.2 需要多长时间？ | 74 |
| 2.9 NixOS系统 | 98 |
| 2.9.1 创建NixOS镜像 | 98 |
| 2.9.2 测试与部署NixOS配置 | 98 |
| 2.9.3 扩展规模 | 98 |
| 3 操作指南 | 135 |
| 3.1 实用配方 | 135 |
| 3.1.1 配置Nix使用自定义二进制缓存 | 135 |
| 3.1.2 通过direnv实现环境自动激活 | 136 |
| 3.1.3 开发环境中的依赖管理 | 137 |
| 3.1.4 使用npins自动管理远程资源 | 138 |
| 3.1.5 搭建Python开发环境 | 140 |
| 3.1.6 设置构建后钩子 | 142 |
| 3.1.7 使用 GitHub Actions 实现持续集成 | 144 |
| 3.2 最佳实践 | 145 |
| 3.2.1 URL 规范 | 145 |
| 3.2.2 递归属性集 rec { ... } | 146 |
| 3.2.3 with 作用域 | 146 |
| 3.2.4 <...>查找路径 | 147 |
| 3.2.5 可复现的Nixpkgs配置 | 148 |
| 3.2.6 更新嵌套属性集 | 148 |
| 3.2.7 可复现的源路径 | 149 |
| 3.3 故障排除 | 149 |
| 3.3.1 二进制缓存不可用或无法访问时该怎么办？ | 149 |
| 3.3.2 如何强制 Nix 重新检查二进制缓存中是否存在某内容？ | 150 |
| 3.3.3 如何修复：错误：查询数据库中的路径：数据库镜像已损坏 | 150 |
| 3.3.4 如何修复：错误：当前 Nix 存储模式版本为 10，但仅支持版本 7 | 150 |
| 3.3.5 如何修复：写入文件时出错：连接被对方重置 | 150 |
| 3.3.6 macOS 更新导致 Nix 安装损坏 | 151 |
| 3.4 常见问题 | 151 |
| 3.4.1 Nix | 151 |
| 3.4.2 NixOS | 152 |
| 4 参考文献 | 155 |
| 4.1 术语表 | 155 |
| 4.2 Nix参考手册 | 156 |
| 4.3 延伸阅读 | 156 |

| | |
|-----------------------|------------|
| 4.3.1 Nix语言教程 | 156 |
| 4.3.2 其他文章 | 157 |
| 4.3.3 其他视频 | 157 |
| 4.4 固定Nixpkgs版本 | 158 |
| 4.4.1 可用URL值 | 158 |
| 4.4.2 示例 | 158 |
| 4.4.3 查找特定提交与版本 | 159 |
| 5 核心概念 | 161 |
| 5.1 Flakes | 161 |
| 5.1.1 什么是Flakes? | 161 |
| 5.1.2 为何Flakes存在争议? | 162 |
| 5.1.3 我的项目该采用Flakes吗? | 163 |
| 5.1.4 延伸阅读 | 164 |
| 5.2 常见问题 | 164 |
| 5.2.1 Nix名称的由来是什么? | 164 |
| 5.2.2 什么是flakes? | 164 |
| 5.2.3 应该使用哪个频道分支? | 164 |
| 5.2.4 沙盒构建中是否仍存在杂质? | 165 |
| 6 贡献指南 | 167 |
| 6.1 如何参与贡献 | 167 |
| 6.1.1 入门指引 | 167 |
| 6.1.2 问题报告 | 168 |
| 6.1.3 为Nix作贡献 | 168 |
| 6.1.4 为Nixpkgs作贡献 | 169 |
| 6.1.5 为NixOS做贡献 | 169 |
| 6.2 如何获取帮助 | 169 |
| 6.3 如何获取帮助 | 169 |
| 6.3.1 如何寻找维护者 | 170 |
| 6.3.2 应使用哪些沟通渠道 | 170 |
| 6.3.3 其他途径 | 171 |
| 6.4 贡献文档 | 171 |
| 6.4.1 入门指南 | 171 |
| 6.4.2 反馈意见 | 171 |
| 6.4.3 许可与署名 | 172 |
| 7 致谢 | 183 |
| 7.1 赞助支持 | 183 |
| 7.2 历史 | 183 |
| 索引 | 185 |

安装 NIX

系统要求：

- 安装前，您可能需要先安装 xz-utils 或类似工具，用于解压将通过以下脚本下载的 Nix 二进制压缩包 (.tar.xz 格式)。

Linux 系统

通过推荐的多用户安装方式³ 进行安装：

```
§ curl -L https://nixos.org/nix/install | sh -s -- --daemon
```

在 Arch Linux 上，您也可以通过 pacman⁴ 来安装 Nix。

macOS

通过推荐的多用户安装方式⁵ 安装 Nix：

```
§ curl -L https://nixos.org/nix/install | sh
```

重要提示：升级至 macOS 15 Sequoia

如果您最近升级到 macOS 15 Sequoia 并遇到以下错误

```
错误：用户组'nixbld'中的用户'_nixbld1'不存在
```

运行Nix命令时，请参考GitHub issue NixOS/nix#10892⁶ 获取无需重装即可修复安装的指导。

Windows (WSL2)

通过推荐的单用户安装方式安装Nix：

```
§ curl -L https://nixos.org/nix/install | sh -s -- --no-daemon
```

但若已启用systemd支持⁸，则通过推荐的多用户安装方式⁹ 安装Nix：

³ <https://nix.dev/manual/nix/stable/installation/multi-user.html>

⁴ <https://wiki.archlinux.org/title/Nix#Installation>

⁵ <https://nix.dev/manual/nix/stable/installation/multi-user.html>

⁶ <https://github.com/NixOS/nix/issues/10892>

⁷ <https://nix.dev/manual/nix/stable/installation/single-user.html>

⁸ <https://learn.microsoft.com/en-us/windows/wsl/wsl-config#systemd-support>

⁹ <https://nix.dev/manual/nix/stable/installation/multi-user.html>

```
$ curl -L https://nixos.org/nix/install | sh -s -- --daemon
```

Docker

启动带Nix的Docker shell：

```
§ docker run -it nixos/nix
```

或启动带Nix的Docker shell并挂载工作目录：

```
$ mkdir workdir  
§ docker run -it -v $(pwd)/workdir:/workdir nixos/nix
```

上述工作目錄示例也可用于开始对Nixpkgs进行修改：

```
$ git clone git@github.com:NixOS/nixpkgs  
$ docker run -it -v $(pwd)/nixpkgs:/nixpkgs nixos/nix  
bash-5.1# nix-build -I nixpkgs=/nixpkgs -A hello  
bash-5.1# find ./result # 此符号链接指向构建的软件包
```

1.1 验证安装

打开新终端并输入以下命令以检查安装：

```
§ nix --version  
nix (Nix) 2.11.0
```

These sections contain a series of lessons to get started.

2.1 入门指南

本系列教程是您开始学习Nix的最佳起点。

通过这些课程，您将使用基础Nix命令获取几乎任何软件、即时创建开发环境，并学会编写可复现的脚本。您还将学会阅读Nix语言，并最终用它构建可移植、可复现的开发环境。

2.1.1 临时Shell环境

在Nix的Shell环境中，您可以立即使用任何由Nix打包的程序，无需永久安装。

您还可以与他人共享调用该Shell的命令，这些命令可在所有Linux发行版、WSL及 macOS¹ 上运行。

创建一个 shell 环境

安装 Nix（第1页）后，您可以用它来创建包含所需程序的新 shell 环境。

本节将运行两个非常规程序 cowsay 和 lolcat——您的机器上很可能尚未安装它们：

```
$ cowsay 无法执行
当前未安装程序 'cowsay'

$ echo 没机会 | lolcat
程序 'lolcat' 当前未安装。
```

使用 nix-shell 时通过 -p (--packages) 选项指定需要 cowsay 和 lolcat 软件包：

注意：首次为这些软件包调用 nix-shell 时可能需要较长时间下载所有依赖项。

```
$ nix-shell -p cowsay lolcat
将构建以下 3 个派生项：
/nix/store/zx1j8gchgwfjn7sr4r8yx7a0afkjdg-builder.pl.drv
/nix/store/h9sbaa2k8ivnlihw2czhl5b58k0f7fsfh-lolcat-100.0.1.drv
...
```

(接续下一页)

¹ 并非所有软件包都同时支持Linux和macOS系统，图形界面程序的支持情况可能尤其不同。

(接上页)

```
[nix-shell:~]$
```

在 Nix shell 中，你可以使用这些软件包提供的程序：

```
[nix-shell:~]$ cowsay Hello, Nix! | lolcat
```

输入 exit 或按 CTRL-D 退出 shell，相关程序将不再可用。

```
[nix-shell:~]$ exit
```

退出

```
$ cowsay 不再有  
程序 'cowsay' 当前未安装。
```

```
§ echo 全部消失 | lolcat  
程序 'lolcat' 当前未安装。
```

程序单次运行

通过直接运行任何程序，你还能更快：

```
$ nix-shell -p cowsay --run "cowsay Nix"
```

若命令仅包含程序名，则无需引号：

```
$ nix-shell -p hello --run hello
```

搜索软件包

你能往Shell环境里放什么？只要你能想到的，很可能就有对应的Nix包。

提示：在 [search.nixos.org¹⁰](https://search.nixos.org) 中输入您想运行的程序名称，可查找提供该程序的软件包。

以下示例中，请找出这些程序对应的软件包名称：

- git
- nvim
- npm

搜索结果中，每项会显示软件包名称，详情列表则列出可用的程序。²

¹⁰ <https://search.nixos.org/packages>

² 软件包名称并不等同于程序名称。许多软件包会提供多个程序，若为库文件则可能完全不包含可执行程序。即使某个软件包仅提供一个程序，其包名与程序名也未必相同。

运行任意程序组合

获取包名后，即可启动包含该包的Shell。-p（--packages）参数支持输入多个包名。

启动包含git、nvim和npm包的Nix Shell。首次调用时可能需要较长时间下载所有依赖项。

```
$ nix-shell -p git neovim nodejs
```

将构建以下9个派生项：

```
/nix/store/7gz8jyn99kw4k74bgm4qp6z48715ap06-packdir-startdrv
/nix/store/d6fkgxc3b04m85wrhg6j015y0ray8217-packdir-optdrv
/nix/store/da6njv7r0zzc2n54n2j54g2a5sbi4a5i-manifest.vimdrv
/nix/store/zs4jb2ybr4rcyzwq0dagg9rlhlc368h6-builder.pldrv
/nix/store/g8sl2xnsshfrz9f39ki94k8p15vp3xd7-vim-pack-dirdrv
/nix/store/jmxkg8b1psk52awsfvfziy9nq6dwmxmjp-luajit-2.1.0-2022-10-04-envdrv
/nix/store/kn83q8yk6ds74zgyklrhvv5wkvbwmcch-python3-3.10.9-envdrv
/nix/store/m445wn3vizcg7syna2cdkkws3kk1gq8-neovim-ruby-envdrv
/nix/store/r2wa882mw99c311a4my7hcis9lq3kp3v-neovim-0.8.1drv
```

将获取以下151个路径（下载量186.43 MiB，解压后1018.20 MiB）：

```
/nix/store/046zxlxhq4srm3ggafkymx794bn1jksc-bzip2-1.0.8
/nix/store/0p1jxcb7b4p8jhhlf8qnjc4cqwy89460-unibilium-2.1.1
/nix/store/0q4fpnqmg81iqraj7zidylcyd062f6z0-perl5.36.0-URI-5.05
```

...

```
[nix-shell:~]$
```

检查软件包版本

请确认您使用的是Nix提供的这些程序的特定版本，即使您的机器上已安装过它们。

```
[nix-shell: ~] $ which git
/nix/store/3cdi52xh61k3h1fb51jkxs3p561p37wg-git-2.38.3/bin/git
```

```
[nix-shell:~]$ git --version
git 版本 2.38.3
```

```
[nix-shell:~]$ 查找 nvim 路径
/nix/store/ynskzgkf07lmrrs3cl2kzr9ah4871wab-neovim-0.8.1/bin/nvim
```

```
[nix-shell:~]$ 获取 nvim 版本 | 首行
NVIM v0.8.1
```

```
[nix-shell:~]$ 查找 npm 路径
/nix/store/q12w83z0i5pi1y0m6am7qmw1r73228sh-nodejs-18.12.1/bin/npm
```

```
[nix-shell:~]$ npm --version
8.19.2
```

嵌套的Shell会话

如需临时使用额外程序，可运行嵌套式Nix Shell。指定包提供的程序将被添加到当前环境中。

```
[nix-shell:~]$ nix-shell -p python3
将获取此路径（下载11.42 MiB，解压后62.64 MiB）：
/nix/store/pwy30a7siqrkki9r7xd1lksyv9fg4l1r-python3-3.10.11
正在从以下位置复制路径 '/nix/store/pwy30a7siqrkki9r7xd1lksyv9fg4l1r-python3-3.10.11'
→ 'https://cache.nixos.org' ...
[nix-shell:~]$ python --version
Python 3.10.11
```

照常退出 shell 即可返回之前的环境。

迈向可复现性

这些 shell 环境非常便利，但目前的示例尚不具备可复现性。在其他机器上运行这些命令可能会获取不同版本的软件包，具体取决于该机器安装 Nix 的时间。

何为可复现？完全可复现的示例意味着无论何时何地运行该命令，都将获得完全相同的结果。每次提供的环境都完全一致。

以下示例创建了一个完全可复现的环境。您可随时随地运行它，获取完全相同的git 版本。

```
$ nix-shell -p git --run "git --version" --pure -I nixpkgs=https://github.com/
-NixOS/nixpkgs/tarball/2a601aafdc5605a5133a2ca506a34a3a73377247
...
git version 2.33.1
```

此处涉及三个要点：

1. `--run` 在 Nix 创建的环境中执行给定的 Bash 命令¹¹，完成后立即退出。
当你需要快速运行本地未安装的程序时，可随时在 nix-shell 中使用此功能。

2. `--pure` 会在运行 shell 时丢弃系统设置的大部分环境变量。

这意味着该 shell 中仅存在 Nix 提供的 git 工具。这对示例中的简单单行命令很有用。但在开发时，通常需要保留编辑器和其他工具，因此建议开发环境省略 `--pure` 参数，仅在需要额外隔离时添加。

3. `-I` 用于指定软件包声明的来源。

这里我们提供了 nixpkgs¹² 的特定 Git 修订版，确保明确使用该集合中的软件包版本。

¹¹ <https://www.gnu.org/software/bash/manual/bash.html#Shell-Commands>

¹² <https://github.com/NixOS/nixpkgs/tree/2a601aafdc5605a5133a2ca506a34a3a73377247>

参考资料

- Nix手册：nix-shell¹³（或运行 man nix-shell）
- Nix手册：-I 选项¹⁴

后续步骤

- 可复现的解释型脚本（第7页） - 使用Nix实现脚本可复现性
- Nix语言基础（第13页） - 学习用于声明包和配置的Nix语言
- 声明式 shell 环境（shell.nix，第9页）——通过声明式配置文件创建可复现的 shell 环境
- 实现可复现性：固定 Nixpkgs（第12页）——学习指定软件包源精确版本的不同方法

若您暂时结束 Nix 试用，可通过以下命令释放示例程序下载的多个版本占用的磁盘空间：

```
$ nix-collect-garbage
```

2.1.2 可复现的解释型脚本

本教程将指导您使用 Nix 创建并运行可复现的解释型脚本（亦称 shebang¹⁵ 脚本）。

需求

- 可正常运行的Nix环境（第1页）
- 熟悉Bash¹⁶

依赖复杂但脚本简单

参考以下脚本：它获取URL的XML内容，转换为JSON格式，并进行美化输出：

```
#!/bin/bash

curl https://github.com/NixOS/nixpkgs/releases.atom | xml2json | jq
```

该命令需要curl、xml2json和jq程序，同时依赖bash解释器。若执行脚本的系统中缺少任一依赖项，脚本将部分或完全运行失败。

通过Nix，我们可以显式声明所有依赖项，并生成一个能在任何支持Nix的机器上运行的脚本，所需包均来自Nixpkgs。

¹³ <https://nix.dev/manual/nix/stable/command-ref/nix-shell>

¹⁴ <https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-I>

¹⁵ [https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))

¹⁶ <https://www.gnu.org/software/bash/>

脚本

Shebang¹⁷ 决定了使用哪个程序来运行解释型脚本。

我们将使用 shebang 行 `#!/usr/bin/env nix-shell`。

`env`¹⁸ 是一个在现代类 Unix 系统中通常位于 `/usr/bin/env` 路径的程序。它接收一个命令名作为参数，并在环境变量 `$PATH` 列出的目录中查找并运行第一个匹配的可执行文件。

我们使用 `nix-shell` 作为 shebang 解释器¹⁹。它接收以下与我们用例相关的参数：

- `-i` 指定用于解释文件剩余部分的程序
- `--pure` 选项在脚本运行时排除大部分环境变量
- `-p` 列出解释器环境中应存在的包
- `-I` 显式设置包的搜索路径²⁰

更多选项细节可查阅 `nix-shell` 参考文档²¹。

创建名为 `nixpkgs-releases.sh` 的文件，内容如下：

```
#!/usr/bin/env nix-shell
#! nix-shell -i bash --pure
#! nix-shell -p bash cacert curl jq python3Packages.xmljson
#! nix-shell -I nixpkgs=https://github.com/NixOS/nixpkgs/archive/
->2a601aafdc5605a5133a2ca506a34a3a73377247.tar.gz

curl https://github.com/NixOS/nixpkgs/releases.atom | xml2json | jq.
```

首行是标准的shebang。后续shebang行是Nix特有的语法：

- 使用 `-i` 选项时，会指定 `bash` 作为文件剩余部分的解释器。
- 此处启用 `--pure` 选项，以防止脚本隐式使用运行环境中可能已存在的程序。
- `-p` 选项会列出脚本运行所需的依赖包。

命令 `xml2json` 由 `python3Packages.xmljson` 包提供，而 `bash`、`jq` 和 `curl` 由同名包提供。`cacert` 必须存在以支持 SSL 认证。

提示： 使用 `search.nixos.org`²² 查找提供所需程序的包。

- `-I` 的参数指向 Nixpkgs 仓库的特定 Git 提交。

这确保脚本在任何地方都能始终使用完全相同的软件包版本运行。

使脚本可执行：

```
chmod +x nixpkgs-releases.sh
```

运行脚本：

```
./nixpkgs-releases.sh
```

¹⁷ [https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))

¹⁸ <https://pubs.opengroup.org/onlinepubs/9699919799/utilities/env.html>

¹⁹ <https://nix.dev/manual/nix/stable/command-ref/nix-shell.html#use-as-a--interpreter>

²⁰ <https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-I>

²¹ <https://nix.dev/manual/nix/stable/command-ref/nix-shell.html#options>

²² <https://search.nixos.org/packages>

后续步骤

- Nix语言基础（第13页）：学习用于声明软件包与配置的Nix语言。
- 使用shell.nix声明式Shell环境（第9页）：通过声明式配置文件创建可复现的Shell环境。
- 垃圾回收²³ - 释放通过Nix安装程序所占用的存储空间

2.1.3 使用shell.nix的声明式Shell环境

概述

声明式Shell环境允许您：

- 在环境激活时自动运行bash命令
- 自动设置环境变量
- 将环境定义纳入版本控制并在其他机器上复现

您将学到什么？

在《临时Shell环境（第3页）》教程中，您已学习如何通过nix-shell -p命令式创建Shell环境。这适用于快速获取工具而无需永久安装的场景。您还学会了如何通过Git提交作为参数，使用特定Nixpkgs修订版执行该命令，以复现之前使用的相同环境。

本教程将介绍如何通过Nix文件中的声明式配置创建可复现的Shell环境。该文件可共享给他人，以便在不同机器上重现相同环境。

需要多长时间？

30分钟

需要哪些准备？

- 熟悉Unix shell
- 对Nix语言有基本了解（第13页）

进入临时shell环境

假设我们需要一个能使用cowsay和lolcat的环境。最简单的实现方式是通过nix-shell -p命令：

```
$ nix-shell -p cowsay lolcat
```

该命令有效，但存在若干缺点：

- 每次进入shell时都必须重复输入-p cowsay lolcat
- （从人体工学角度）它不允许你对shell环境进行更多自定义

更好的解决方案是从 shell.nix 文件创建我们的 shell 环境。

²³ <https://nix.dev/manual/nix/stable/package-management/garbage-collection.html>

基础 shell.nix 文件

创建名为 shell.nix 的文件，内容如下：

```

let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
5
pkgs.mkShellNoCC {
  packages = with pkgs; [
    cowsay
    lolcat
10
  ];
11
}

```

详细说明

我们使用固定到发布分支的Nixpkgs版本（第158页）。若您遵循了临时Shell环境（第3页）教程且不希望重复下载所有依赖项，请指定与迈向可复现性（第6页）章节完全相同的修订版本：

```

let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/
->a601aafdc5605a5133a2ca506a34a3a73377247";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

```

我们显式设置 config 与 overlays 以避免其被全局配置²⁴ 意外覆盖。

`mkShellNoCC` 是接收属性集作为参数的函数，此处我们传入的 `packages` 属性包含来自 `pkgs` 属性集的两个条目。

关于 `mkShell` 的补充说明

`nix-shell` 和 `mkShell` 最初被构想为一种构建包含调试软件包构建所需工具（如 `Make` 或 `GCC`）的 shell 环境²⁵ 的方式，后来才广泛用作创建临时通用环境的通用方案。`mkShellNoCC` 是一个生成此类环境的函数，但不包含编译器工具链。

你可能会遇到将软件包添加到 `buildInputs` 或 `nativeBuildInputs` 属性的 `mkShell` 或 `mkShellNoCC` 示例。`mkShellNoCC` 是 `mkDerivation`²⁶ 的封装，因此它接受与 `mkDerivation` 相同的参数（如 `buildInputs` 或 `nativeBuildInputs`）。`mkShellNoCC` 的 `packages` 参数只是 `nativeBuildInputs` 的别名。通过在 `shell.nix` 所在目录运行 `nix-shell` 进入环境：

注意：首次在该文件上运行 `nix-shell` 可能需要较长时间以下载所有依赖项。

```
$ nix-shell
[nix-shell]$ cowsay hello | lolcat
```

默认情况下，`nix-shell` 会在当前目录查找名为 `shell.nix` 的文件，并根据该文件中的 Nix 表达式构建 shell 环境。`packages` 属性中定义的软件包将出现在 `$PATH` 中。

²⁴ <https://nixos.org/manual/nixpkgs/stable/#chap-packageconfig>

²⁵ <https://nixos.org/manual/nixpkgs/stable/#sec-tools-of-stdenv>

²⁶ <https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-mkShell>

环境变量

您可能希望在进入 shell 环境时自动导出某些环境变量。

设置 GREETING 以便在 shell 环境中使用：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs.mkShellNoCC {
  packages = with pkgs; [
    cowsay
    lolcat
  ];
+ GREETING = "Hello, Nix!";
}
```

任何传递给mkShellNoCC的属性名（未被保留且值可强制转换为字符串的）最终都会成为环境变量。

试试看！输入 exit 或按 Ctrl+D 退出 shell，然后通过 nix-shell 再次启动它。

```
[nix-shell] $ echo $GREETING
```

警告：部分变量受保护，无法按上述方式设置。

例如，大多数 shell 的提示符格式由 PS1 环境变量设置，但 nix-shell 默认已配置此变量，并会忽略参数中设置的 PS1 属性。

如需覆盖这些受保护的环境变量，请使用下一节所述的 shell Hook 属性。

启动命令

在进入 shell 环境前，您可能需要运行某些命令。这些命令可放置在提供给 mkShellNoCC 的 shellHook 属性中。

将 shellHook 设置为输出彩色欢迎语：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs.mkShellNoCC {
  packages = 使用 pkgs; [
    cowsay
    lolcat
  ];
  GREETING = "你好， Nix! ";
+
+ shellHook =
+   echo $GREETING | cowsay | lolcat
+   “ ;
}
```

再试一次！输入exit或按Ctrl+D退出shell，然后重新用nix-shell启动以观察效果。

参考资料

- [mkShell文档²⁷](#)
- [Nixpkgs shell函数与工具²⁸ 文档](#)
- [nix-shell文档²⁹](#)

下一步

- Nix语言基础（第13页）
- 使用direnv自动激活环境（第136页）
- 开发环境中的依赖项（第137页）
- 使用npins自动管理远程源（第138页）

2.1.4 实现可复现性：固定Nixpkgs版本

在各种Nix示例中，你经常会看到以下内容：

```
{ pkgs ? import <nixpkgs> {} }:
```

²
...

注：`<nixpkgs>`指向Nixpkgs某个修订版本的文件系统路径。更多关于查找路径的信息（第26页）详见Nix语言基础（第13页）。

这是快速演示Nix表达式并通过导入Nix包使其生效的便捷方式。
但该Nix表达式不具备完全可复现性。

在Nix表达式中通过URL锁定包版本

要创建完全可复现的Nix表达式，我们可以固定Nixpkgs的精确版本。
最简便的方法是将所需的Nixpkgs版本作为通过对应Git提交哈希指定的压缩包获取：

```
{ pkgs ? import (fetchTarball "https://github.com/NixOS/nixpkgs/archive/
→ 06278c77b5d162e62df170fec307e83f1812d94b.tar.gz") {} }:
```

³
...

可通过[status.nixos.org³⁰](https://status.nixos.org/)选择提交，该网站列出了所有版本及通过全部测试的最新提交。

选择提交时，建议遵循以下任一方式

- 使用特定版本（如nixos-21.05）跟踪最新稳定版NixOS发行版，或

²⁷ <https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-mkShell>

²⁸ <https://nixos.org/manual/nixpkgs/stable/#ssec-stdenv-functions>

²⁹ <https://nix.dev/manual/nix/stable/command-ref/nix-shell>

³⁰ <https://status.nixos.org/>

- 通过 nixos-unstable 获取最新的不稳定版本。

后续步骤

- 更多固定 nixpkgs 的示例和方法细节，请参阅《固定 Nixpkgs》（第158页）。
- 使用 npins 自动管理远程源（第138页）

2.2 Nix 语言基础

Nix 语言专为便捷创建和组合派生而设计——这些派生是对如何利用现有文件内容生成新文件的精确描述。它是一种领域特定、纯函数式、惰性求值且动态类型的编程语言。

Nix语言的显著用途

- **Nixpkgs**

全球规模最大、更新最及时的软件发行版，且完全采用Nix语言编写。

- **NixOS**

一款可完全声明式配置的Linux发行版，基于Nix与Nixpkgs构建。

其底层模块化配置系统由Nix语言编写，并采用Nixpkgs中的软件包。系统环境及其提供的服务均通过Nix语言配置。

您可能会很快遇到看似非常复杂的Nix语言表达式。与任何编程语言一样，所需的Nix语言代码量与其解决的问题复杂度紧密相关，并反映出对该问题及其解决方案的理解程度。构建软件是一项复杂任务，而Nix既揭示了这种复杂性，又允许通过Nix语言来管理它。然而，Nix语言本身只有少数几个基本概念，本教程将逐一介绍，这些概念可以任意组合。看似复杂的部分并非来自语言本身，而是源于其使用方式。

2.2.1 概述

本文旨在帮助读者理解Nix语言，以便更好地学习其他教程和示例。

在实践中使用Nix语言涉及多个方面：

- 语言：语法与语义
- 基础库：builtins 与 pkgs.lib
- 开发者工具：测试、调试、代码检查、格式化等
- 通用构建机制：stdenv.mkDerivation、简易构建器等
- 组合与配置机制：override、overrideAttrs、overlays、callPackage 等
- 生态专用打包机制：buildGoModule、buildPythonApplication 等
- NixOS 模块系统：config、option 等

本教程仅涵盖最重要的语言特性，简要讨论库相关内容，最后将引导您查阅其他组件的参考材料与资源。

你将学到什么？

本教程旨在让你能够阅读典型的Nix语言代码并理解其结构，重点展示Nix语言与你熟悉语言的差异之处。

因此教程展示了Nix语言中最常见且最具特色的模式：

- 命名赋值与值访问（第16页）
- 函数声明与调用（第26页）
- 内置函数与库函数（第32页）
- 杂质（第35页）用于获取构建输入
- 派生（第37页）用于描述构建任务

重要提示：本教程不会详细解释所有Nix语言特性，也不涉及语法规则的具体细节。例如，我们会跳过诸如if...then...else...等常见结构。
完整语言参考请查阅Nix手册³¹。

你需要什么？

- 熟悉软件开发
- 熟悉 Unix shell，以便阅读命令行示例
- 已安装 Nix（第1页）以运行示例

需要多长时间？

- 无函数式编程经验：2小时
- 熟悉函数式编程：1小时
- 精通函数式编程：30分钟

我们建议运行所有示例。通过实践来验证你的假设并测试所学内容。若想确保完全理解示例，请阅读详细解析。

如何运行这些示例？

- 一段 Nix 语言代码即是一个 Nix 表达式。
- 求值 Nix 表达式会生成一个 Nix 值。
- Nix 文件（扩展名为 .nix）的内容即为 Nix 表达式。

注：求值指根据语言规则将表达式转换为值的过程。

本教程包含大量Nix表达式示例，每个示例后都附有预期求值结果。

以下示例是一个将两个数字相加的Nix表达式：

```
1 + 2
```

```
3
```

³¹ <https://nix.dev/manual/nix/stable/language/index.html>

交互式求值

使用 nix repl³² 以交互方式评估 Nix 表达式（通过在命令行输入表达式）：

```
$ nix repl
欢迎使用 Nix 2.13.3。输入 ?: 获取帮助。

nix-repl> 1 + 2
3
```

注意：Nix 语言采用惰性求值，默认情况下 nix repl 仅在需要时计算值。

部分示例为清晰起见展示了完整求值的数据结构。若输出与示例不符，可尝试在输入表达式前添加 :p。

示例：

```
nix-repl> { a.b.c = 1; }
{ a = { ... }; }

nix-repl> :p { a.b.c = 1; }
{ a = { b = { c = 1; }; }; }
```

输入 :q 退出 nix repl³³。

评估Nix文件

使用nix-instantiate --eval³⁴ 来评估Nix文件中的表达式。

```
$ echo 1 + 2 > file.nix
$ nix-instantiate --eval file.nix
3
```

详细说明

第一条命令将1 + 2写入当前目录下的file.nix文件。现在file.nix的内容是1 + 2，可通过以下命令验证：

```
§ cat file.nix
1 + 2
```

第二条命令使用 --eval 选项对 file.nix 执行 nix-instantiate，该操作会读取文件并评估其中包含的 Nix 表达式。最终结果将作为输出打印。

--eval 的作用是仅评估文件而不执行其他操作。若省略 --eval，nix-instantiate 会要求给定文件中的表达式必须评估为一个名为 derivation 的特殊值，我们将在本教程末尾的《Derivations》（第37页）部分详述。

注意：若未指定文件名，nix-instantiate --eval 会尝试读取 default.nix。

```
$ echo 1 + 2 > default.nix
$ nix-instantiate --eval
3
```

³² <https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-repl.html>

³³ <https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-repl.html>

³⁴ <https://nix.dev/manual/nix/stable/command-ref/nix-instantiate.html>

注意：Nix语言采用惰性求值， 默认情况下nix-instantiate仅在需要时计算值。

部分示例为清晰展示会输出完整求值后的数据结构。若结果与示例不符，可尝试为nix-instantiate添加--strict选项。

示例：

```
$ echo "{ a.b.c = 1; }" > file.nix
$ nix-instantiate --eval file.nix
{ a = <CODE>; }
```

```
$ echo "{ a.b.c = 1; }" > file.nix
$ nix-instantiate --eval --strict file.nix
{ a = { b = { c = 1; }; }; }
```

关于空格的说明

空格用于在必要时分隔词法标记³⁵，除此之外无实际意义。

换行、缩进及额外空格仅为方便阅读而设。

以下是等价形式：

```
let
  x = 1;
  y = 2;
in x + y
```

3

```
let x=1;y=2;in x+y
```

3

2.2.2 名称与值

Nix语言中的值可以是基本数据类型、列表、属性集和函数。

我们在属性集（第17页）的上下文中展示基本数据类型和列表的示例。本节后续将介绍字符串的特殊特性：字符串插值（第23页）、文件系统路径（第25页）以及缩进字符串（第24页）。函数部分（第26页）将单独阐述。

属性集（第17页）和let表达式（第18页）用于为值分配名称。赋值通过单个等号（=）表示。

当你在Nix语言代码中遇到等号（=）时：

- 其左侧为被赋值的名称。
- 其右侧为值，以分号（;）分隔。

³⁵ https://en.wikipedia.org/wiki/Lexical_analysis#Lexical_token_and_lexical_tokenization

属性集 { ... }

属性集是名称-值对的集合，其中名称必须唯一。

以下示例展示了所有基本数据类型、列表及属性集。

注意：若熟悉JSON，可将Nix语言视为支持函数的JSON。

Nix语言中不含函数的数据类型，其行为与JSON中的对应类型完全一致且外观极为相似。

Nix

```
{
  字符串 = "你好";
  整数 = 1;
  浮点数 = 3.141;
  布尔值 = 真;
  空值 = 空;
  列表 = [1 "二" 假];
  属性集 = {
    a = "你好";
    b = 2;
    c = 2.718;
    d = 假;
  }; # 支持注释
}
```

14

JSON

```
{
  "字符串": "你好",
  "整数": 1,
  "float": 3.141,
  "bool": true,
  "null": null,
  "list": [1, "two", false],
  "object": {
    "a": "hello",
    "b": 1,
    "c": 2.718,
    "d": false
  }
}
```

注意：

- 属性名通常无需引号。⁶⁶
- 列表元素通过空格分隔。⁶⁸

⁶⁶ 详情参阅：Nix手册 - 属性集⁶⁷

⁶⁷ <https://nix.dev/manual/nix/stable/language/syntax#attrs-literal>
详情：Nix手册 - 列表⁶⁹

⁶⁹ <https://nix.dev/manual/nix/stable/language/syntax#list-literal>

递归属性集 rec { ... }

有时会看到属性集声明前加有rec。这允许在集合内部访问属性。

示例：

```
rec {
  one = 1;
  two = one + 1;
  three = two + 1;
}
```

```
{ one = 1; three = 3; two = 2; }
```

注：属性集中的元素可以按任意顺序声明，在求值时排序。

反例：

```
{
  one = 1;
  two = one + 1;
  three = two + 1;
}
```

错误：未定义变量 'one'

位于 «string»: 3 : 9 :

```
2 | one = 1;
3 two = one + 1;
|           A
4 three = two + 1;
```

let ... in ...

亦称"let表达式"或"let绑定"

let表达式允许为值分配名称以便重复使用。

示例：

```
let
  a = 1;
in
a + a
```

2

详细说明

赋值语句位于关键字 `let` 和 `in` 之间。本例中我们分配了一个 $= 1$ 。
`in` 之后是赋值生效的表达式，即可使用已赋值名称的区域。本例中表达式为 $a + a$ ，其中 a 指向 $a = 1$ 。
 通过将名称替换为其赋值， $a + a$ 的计算结果为 2。
 名称可按任意顺序赋值，且等号 (=) 右侧的表达式可引用其他已赋值的名称。

示例：

```
let
  b = a + 1;
  a = 1;
in
a + b
```

3

详细说明

赋值语句位于关键字 `let` 与 `in` 之间。本例中我们为 a 赋予 $= 1$ ，为 b 赋予 $= a + 1$ 。
 赋值顺序无关紧要。因此下面这个颠倒赋值顺序的例子与之等效：

```
let
  a = 1;
  b = a + 1;
in
a + b
```

3

注意 $b = a + 1$ 中的 a 指向 $a = 1$ 。

在表达式生效的作用域内，赋值操作有效。本例中表达式为 $a + b$ ，其中 a 指向 $a = 1$ ，而 b 指向 $b = a + 1$ 。

将名称替换为赋值后， $a + b$ 的计算结果为 3。

这与递归属性集（第18页）类似：两者中赋值顺序无关紧要，且左侧名称可用于赋值符 (=) 右侧的表达式。

示例：

`let ... in ...`

```
let
  b = a + 1;
  c = a + b;
  a = 1;
在 {c = c; a = a; b = b;} 中
```

{ $a = 1; b = 2; c = 3;$ }

记录 {...}

记录 {

```
b = a + 1;
c = a + b;
a = 1;
```

}

```
{a = 1; b = 2; c = 3; }
```

区别在于：递归属性集会求值为一个属性集（第17页），而in关键字后可跟随任意表达式。

在以下示例中，我们使用let表达式来构建列表：

let

```
b = a + 1;
c = a + b;
a = 1;
```

在 [a b c] 中

```
[1 2 3]
```

只有 let 表达式内部的表达式才能访问新声明的名称。我们称之为：绑定具有局部作用域。

反例：

```
{
  a = let x = 1; in x;
  b = x;
}
```

错误：未定义变量 'x'

在 «string»: 3 : 7 :

```
2| a = let x = 1; in x;
3| b = x;
   |           ^
4|
```

属性访问

集合中的属性通过点号(.)和属性名进行访问。

示例：

```
let
  属性集 = {x = 1;} ;
in
  属性集.x
```

```
1
```

访问嵌套属性的方式与此相同。

示例：

```
let
  属性集 = {a = {b = {c = 1; }; }; };
in
属性集.a.b.c
```

1

点号(.)也可用于属性赋值。

示例：

```
{ a.b.c = 1; }
```

```
{ a = { b = { c = 1; }; }; }
```

使用 ...;...

with表达式允许访问属性，而无需重复引用其属性集。

示例：

```
let
  a = {
    x = 1;
    y = 2;
    z = 3;
  };
in
with a; [x y z]
```

[1 2 3]

表达式

通过 a; [x y z]

等价于

```
[ a.x a.y a.z ]
```

通过with引入的属性仅在分号();后的表达式范围内有效

反例：

```
let
  a = {
    x = 1;
    y = 2;
    z = 3;
  };
in
{
  b = 使用 a; [ x y z ];
  c = x;
}
```

错误：未定义变量 'x'

位于 «string»:10 : 7 :

(接续下一页)

(continued from previous page)

```

9| b = 使用 a; [ x y z ];
10| c = x;
  ^
11 }
```

继承...

继承是一种简写方式，用于将现有作用域中的名称值赋给嵌套作用域中的同名变量。这种便利设计避免了重复书写相同名称。

示例：

```

let
  x = 1;
  y = 2;
in
{
  继承 x y;
}
```

```
{x = 1; y = 2; }
```

片段

```
继承 xy ;
```

等价于

```
x = x; y = y;
```

也可通过带括号的特定属性集继承名称 (inherit (...) ...)。示例：

```

let
  a = { x = 1; y = 2; };
in
{
  继承(a) xy ;
}
```

```
{x = 1; y = 2; }
```

该片段

```
继承(a) xy ;
```

等价于

```
x = a. x; y = a. y;
```

inherit 在 let 表达式中同样有效

示例：

```

让
  继承 ({x = 1; y = 2; })xy ;
于 [x  y]
```

[12]

详细说明

虽然这个例子是人为设计的，但在更复杂的代码中，你会经常看到嵌套的let表达式重用外层作用域的变量名。

这里我们使用属性集 `{x = 1; y = 2;}` 作为非平凡的继承来源。该let表达式通过()从该属性集继承了 `x` 和 `y`，其效果等同于编写：

```
let
  x = {x = 1; y = 2;.x;
  y = {x = 1; y = 2;.y;
in
```

新的内部作用域现在包含 `x` 和 `y`，它们被用于列表 `[x y]` 中。

字符串插值 \${ ... }

旧称“反引用”。

Nix表达式的值可通过美元符号加花括号 (`{}$`) 插入字符串中。

示例：

```
let
  name = "Nix";
in
"hello ${name}"
```

"你好 Nix"

仅允许使用字符串或可表示为字符串的值。

反例：

```
let
  x = 1;
in
"${x} + ${x} = ${x + x}"
```

错误：无法将整数强制转换为字符串

在 «string»:4:2 处：

```
3 | in
4 | "${x} + ${x} = ${x + x}"
  |
  5 |
```

插值表达式可以任意嵌套。

(这可能导致可读性下降，实践中建议避免。)

示例：

```
let
  a = "否";
in
"${a + " ${a + " ${a}}"}"
```

```
"不不不"
```

详细说明

任何可表示为字符串的Nix表达式都可用于 \${} 中。

上述表达式中的+号是字符串连接运算符³⁶，它接收两个字符串并生成新字符串。

示例中的表达式故意设计得令人困惑，以证明任意嵌套的字符串插值是可行的，但往往难以阅读。

它表示一个包含以下内容的字符串：将a的值与以空格开头、后接另一个插值字符串的字符串连接后的插值结果。该第二个插值字符串又是a的值与以空格开头、后接a的插值的另一个字符串的连接结果。

例如：

```
let
  a = "一";
  b = "二";
in
"${a + b}"
```

```
"一二"
```

内置函数将在后续章节讨论（第32页）。

警告：您可能会遇到在赋值名称前使用美元符号(\$)但无大括号({ })的字符串：这些并非插值字符串，通常表示shell脚本中的变量。

此类情况下，使用外围Nix表达式中的名称纯属巧合。

示例：

```
let
  out = "Nix";
in
"echo ${out} > $out"

"echo Nix > $out"
```

缩进字符串

亦称“多行字符串”。

Nix语言为具有共同缩进的多行字符串提供了便捷语法。

缩进字符串用双单引号（'''）表示。

示例：

```
...
多行
字符串
拼接
11
```

³⁶ <https://nix.dev/manual/nix/latest/language/operators#string-concatenation>

"多行\n字符串\n"

结果中会去除开头等量的空白字符。

示例：

```
...  
—  
—  
—  
11
```

"—\n—\n—\n"

注：缩进字符串同样支持字符串插值（第23页）。详情请查阅Nix语言中关于字符串字面量的文档³⁷。

文件系统路径

Nix语言为文件系统路径提供了便捷语法。

绝对路径始终以斜杠 (/) 开头。

示例：

/绝对路径

/绝对路径

当路径包含至少一个斜杠 (/) 但不以斜杠开头时，即为相对路径。其值会解析为相对于包含该表达式的文件的路径。

以下示例假设当前Nix文件位于/current/directory（或在/current/directory中运行nix repl）。

示例：

./相对路径

/当前目录/相对路径

示例：

相对路径

/当前目录/相对路径/路径

单个点(.)表示给定路径中的当前目录。

你会经常看到以下表达式，它指定了Nix文件所在目录。

示例：

./

/当前/目录

³⁷ <https://nix.dev/manual/nix/2.24/language/syntax#字符串字面量>

详细说明

由于相对路径必须包含斜杠 (/) 但不能以斜杠开头，而点号 (.) 表示不切换目录，因此组合 ./ 将当前目录指定为相对路径。

双点号 (..) 表示上级目录。

示例：

```
./.
```

```
/current
```

注意：路径可用于插值表达式——这是我们将稍后详细讲解的不纯操作（第35页）
(第35页)。

查找路径

亦称“尖括号语法”。

示例：

```
<nixpkgs>
```

```
/nix/var/nix/profiles/per-user/root/channels/nixpkgs
```

查找路径³⁸ 的值是一个文件系统路径，其取决于builtins.nixPath³⁹ 的值。

实践中，<nixpkgs>指向Nixpkgs某个修订版的文件系统路径。

例如，<nixpkgs/lib> 指向该文件系统路径下的 lib 子目录：

```
<nixpkgs/lib>
```

```
/nix/var/nix/profiles/per-user/root/channels/nixpkgs/lib
```

虽然你会遇到许多此类示例，但我们建议在生产代码中避免使用查找路径（第147页），因为它们是不可复现的杂质（第35页）。

2.2.3 函数

函数在 Nix 语言中无处不在，值得特别关注。

函数始终只接受一个参数。参数与函数体之间用冒号 (:) 分隔。

在Nix语言代码中遇到冒号(:)时：

- 其左侧为函数参数
- 其右侧为函数体。

函数参数是第三种为值命名的方式，区别于属性集（第17页）和let表达式（第18页）。值得注意的是，这些值并非预先已知：名称仅是占位符，在调用函数时才会被填充（第28页）。

Nix语言中的函数声明可呈现多种形式。下文将逐一解释，此处先作概览：

³⁸ <https://nix.dev/manual/nix/2.22/language/constructs/lookup-path>

³⁹ <https://nix.dev/manual/nix/2.22/language/builtin-constants#builtins-nixPath>

- 单一参数

```
x: x + 1
```

- 通过嵌套传递多参数

```
x: y: x + y
```

- 属性集参数

```
{ a, b }: a + b
```

- 带默认属性

```
{ a, b ? 0 }: a + b
```

- 允许包含额外属性

```
{ a, b, ... }: a + b
```

- 具名属性集参数

```
args@{a, b, ... }: a + b + args.c
```

or

```
{ a, b, ... }@args: a + b + args.c
```

Nix语言中的函数没有名称。我们称其为匿名函数，或称这类函数为lambda。⁷⁰示例：

```
x: x + 1
```

```
<LAMBDA>
```

<LAMBDA>表示结果值是一个匿名函数。

与其他值一样，函数可以被赋予一个名称。

示例：

```
let
  f = x: x + 1;
in f
```

```
<LAMBDA>
```

⁷⁰ 术语 lambda 是 lambda 演算⁷² 中 lambda 抽象⁷¹ 的简写。

71 https://en.wikipedia.org/wiki/Lambda_calculus#lambdaAbstr

72 https://en.wikipedia.org/wiki/Lambda_calculus

调用函数

亦称"函数应用"。

调用带参函数时，需将参数写在函数名之后。

示例：

```
let
  f = x: x + 1;
在 f 1
```

2

例如：

```
let
  f = x: x.a;
in
f { a = 1; }
```

1

上例在字面属性集上调用 f 。也可以通过名称传递参数。

例如：

```
let
  f = x: x.a;
  v = { a = 1; };
in
f v
```

1

由于函数与参数通过空格分隔，有时需用圆括号 (()) 才能获得预期结果。

例如：

$(x: x + 1) 1$

2

详细说明

该表达式将一个匿名函数 $x : x + 1$ 应用于参数1。函数必须写在括号内以区别于参数。

示例：

列表元素同样由空格分隔，因此以下两种情况不同：

```
let
  f = x: x + 1;
  a = 1;
在 [ (f a) ] 中
```

[2]

```
let
  f = x: x + 1;
  a = 1;
in [ f a ]
```

```
[ <LAMBDA> 1 ]
```

第一个示例意为：将 f 应用于 a ，并将结果放入列表中。生成的列表仅含一个元素。
 第二个示例意为：将 f 和 a 放入列表。生成的列表包含两个元素。

多参数

亦称“柯里化⁴⁰ 函数”。

Nix函数严格接收单一参数，多参数需通过函数嵌套实现。

此类嵌套函数可像多参数函数般使用，同时提供额外灵活性。

示例：

```
x: y: x + y
```

```
<LAMBDA>
```

上述函数等价于

```
x: (y: x + y)
```

```
<LAMBDA>
```

该函数接收一个参数并返回另一个函数 $y : x + y$ ，其中 x 被设置为该参数值。示例：

```
let
  f = x: y: x + y;
in
f 1
```

```
<LAMBDA>
```

将 $f 1$ 产生的函数应用于另一个参数时，会得到内部表达式 $x + y$ （此时 x 设为1且 y 设为另一参数），此时可进行完整求值。

```
令
  f = x: y: x + y;
in
f 1 2
```

```
3
```

⁴⁰ <https://en.wikipedia.org/wiki/Currying>

属性集参数

亦称“关键字参数”或“解构”。

Nix函数可声明要求特定结构的属性集作为参数。

通过在花括号({ })内列出以逗号(,)分隔的预期属性名来标识。

示例：

```
{a, b}: a + b
```

```
<LAMBDA>
```

该参数明确定义了集合中必须包含的精确属性。遗漏或传递额外属性均属错误。

示例：

```
let
  f = {a, b}: a + b;
in
f { a = 1; b = 2; }
```

```
3
```

反例：

```
让
  f = {a, b}: a + b;
in
f { a = 1; b = 2; c = 3; }
```

错误：'f' 在 (string):2:7 处被调用时传入了意外参数 'c'

位于 «string»:4:1:

```
 3 | in
  4 | f { a = 1; b = 2; c = 3; }
    1 ^
      5
```

默认值

亦称“默认参数”。

解构参数可为属性设置默认值。

通过在属性名与默认值间添加问号(?)标识。

若参数中的属性设有默认值，则非必填。

示例：

```
令
  f = {a, b ? 0}: a + b;
in
f { a = 1; }
```

```
1
```

示例：

```
let
  f = {a ? 0, b ? 0 }: a + b;
in
f {} # 空属性集
```

0

附加属性

允许通过省略号 (...) 添加附加属性：

```
{a, b, ...} : a + b
```

与前一个反例不同，传入包含附加属性的参数不会报错。

示例：

```
let
  f = {a, b, ...}: a + b;
in
f { a = 1; b = 2; c = 3; }
```

3

命名属性集参数

亦称“@模式”、“@语法”或“at语法”。

可为属性集参数指定名称以便整体访问。

通过在属性集参数前或后添加名称并以@符号分隔来表示。

示例：

```
{a, b, ...} @args: a + b + args.c
```

```
<LAMBDA>
```

or

```
args@{a, b, ...}: a + b + args.c
```

```
<LAMBDA>
```

示例：

```
让
  f = {a, b, ...}@参数: a + b + 参数.c;
in
f { a = 1; b = 2; c = 3; }
```

6

2.2.4 函数库

除了内置运算符

⁴¹ (+, ==, &&, etc.), there are two widely used libraries that together can be 这类Nix语言的标准组件外。理解这两者对于阅读和编写Nix代码都至关重要。
建议至少浏览这些内容以熟悉可用功能。

内置函数

亦称"原始操作"或"primops"。

Nix内置了大量语言基础函数，这些函数由C++实现并集成在Nix语言解释器中。

注：Nix手册列出了所有内置函数⁴²，并展示了其使用方法。

这些函数可通过 builtins 常量调用。

示例：

```
builtins.toString
```

```
<PRIMOP>
```

import

多数内置函数仅能通过 builtins 访问，但 import 是显著例外——它也可在顶层直接使用。

import 接受一个 Nix 文件路径，读取并评估其中包含的 Nix 表达式，最终返回结果值。若路径指向目录，则自动使用该目录下的 default.nix 文件。

示例：

```
§ echo 1 + 2 > file.nix
```

```
import ./file.nix
```

```
3
```

详细说明

上述 shell 命令将内容 $1 + 2$ 写入当前目录的 file.nix 文件中。

上述Nix表达式通过./file.nix引用此文件。import会读取该文件并将其解析为包含的Nix表达式。

若文件系统路径不存在则报错。

读取file.nix后，该Nix表达式等价于文件内容：

```
1 + 2
```

⁴¹ <https://nix.dev/manual/nix/stable/language/operators.html>

⁴² <https://nix.dev/manual/nix/stable/language/builtins.html>

3

由于Nix文件可包含任意Nix表达式，导入的函数可立即应用于参数。这意味着，当在import调用后发现额外标记时，其返回值应为函数，后续内容均为该函数的参数。

示例：

```
$ echo "x: x + 1" > file.nix
```

```
import ./file.nix 1
```

2

详细说明

前置的shell命令将内容 $x : x + 1$ 写入当前目录下的file.nix文件中。

上述Nix表达式通过./file.nix引用此文件。import ./file.nix会读取该文件并将其求值为包含的Nix表达式。

若文件系统路径不存在，则报错。

读取文件后，Nix表达式import ./file.nix等价于文件内容：

```
(x : x + 1)1
```

2

这将函数 $x : x + 1$ 应用于参数1，因此求值结果为2。

注意：必须使用括号分隔函数声明与函数应用。

pkgs.lib

nixpkgs⁴³ 代码库包含一个名为 lib⁴⁴ 的属性集，提供了大量实用函数。这些函数由 Nix语言实现，与语言内置函数（第32页）不同。

注：Nixpkgs手册列出了所有Nixpkgs库函数⁴⁵。

按照惯例，Nixpkgs属性集被命名为pkgs，因此这些函数通常通过pkgs.lib访问。

示例：

```
let
  pkgs = import <nixpkgs> {};
in
pkgs.lib.strings.toUpperCase "查找路径被认为有害"
```

查找路径被认为有害

⁴³ <https://github.com/NixOS/nixpkgs>

⁴⁴ <https://github.com/NixOS/nixpkgs/blob/master/lib/default.nix>

⁴⁵ <https://nixos.org/manual/nixpkgs/stable/#sec-functions-library>

详细解析

这是个稍复杂的示例，但至此你应已熟悉其中所有组件。

变量`pkgs`被声明为从某文件导入的表达式。该文件路径由查找路径`<nixpkgs>`的值决定，而该路径又取决于表达式求值时`$NIX_PATH`环境变量的值。由于此表达式恰为函数，求值需传入参数，本例中传递空属性集`{}`即可满足。

既然`pkgs`已在`let...in...`作用域内，便可访问其属性。查阅Nixpkgs手册可知，`lib.strings.toUpperCase`⁴⁶下存在某函数。

为简洁起见，本例通过查找路径获取某版本Nixpkgs。`toUpperCase`函数极为简单，可预期其在不同Nixpkgs版本下结果一致。但更复杂的软件可能受此类问题影响。因此完全可复现的示例如下：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/archive/
-06278c77b5d162e62df170fec307e83f1812d94b.tar.gz";
  pkgs = import nixpkgs {};
in
  pkgs.lib.strings.toUpperCase "务必固定源码版本"
```

务必固定源码版本

详见《实现可复现性：固定Nixpkgs版本》（第12页）。

您还会经常看到`pkgs`作为参数传递给函数。按照惯例，可以假定它指向Nixpkgs属性集，该集合包含一个`lib`属性：

```
{ pkgs, ... }:
  pkgs.lib.strings.removePrefix "no" "no true scotsman"
```

<LAMBDA>

要使该函数生成结果，可将其写入文件（如`file.nix`）并通过`nix-instantiate`传递参数：

```
$ nix-instantiate --eval file.nix --arg pkgs 'import <nixpkgs> {}
"true scotsman"
```

在NixOS配置及Nixpkgs中，常见到直接传递`lib`参数的情况。此时可认为该`lib`等价于仅当`pkgs`可用时的`pkgs.lib`。

例如：

```
{ lib, ... }:
let
  to-be = true;
in
  lib.trivial.or to-be (! to-be)
```

<LAMBDA>

要使该函数输出结果，可将其写入文件（如`file.nix`）并通过`nix-instantiate`传递参数：

```
$ nix-instantiate --eval file.nix --arg lib '(import <nixpkgs> {}).lib'
true
```

⁴⁶ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.strings.toUpperCase>

有时 `pkgs` 和 `lib` 会同时作为参数传递。这种情况下，可以认为 `pkgs.lib` 和 `lib` 是等价的。这样做的目的是通过避免重复使用 `pkgs.lib` 来提高可读性。

示例：

```
{ pkgs, lib, ... }:
#... 多次使用 `pkgs`、
#... 多次使用 `lib`
```

由于历史原因，`pkgs.lib` 中的某些函数与同名内置函数（第32页）功能相同。

2.2.5 杂质

到目前为止，我们仅讨论了所谓的纯表达式：声明数据并通过函数进行转换。

实际上，描述派生（Nix语言的定义性特征，使得能够结合文件系统进行函数式编程）需要观察外部世界。我们将在教程后续部分讨论派生（第37页）。

Nix语言中与此相关的杂质只有一种：从文件系统读取文件作为构建输入。

构建输入是派生所引用的文件，用于描述如何派生出新文件。运行时，派生只能访问显式声明的构建输入。

在Nix语言中，指定构建输入的唯一方式是显式使用：

- 文件系统路径
- 专用函数

Nix及Nix语言通过内容哈希引用文件。若文件内容未知，则表达式求值时读取文件不可避免。

注：Nix支持其他非纯表达式类型，如查找路径（第147页）或常量
`builtins.currentSystem`⁴⁷。此处不作详述，因其不影响Nix语言的核心原理，且因其破坏可重现性而不推荐使用。

路径

当文件系统路径用于字符串插值（第23页）时，该文件内容会作为副作用被复制到Nix存储的特殊位置。

求值后的字符串将包含分配给该文件的Nix存储路径。

示例：

```
§ echo 123 > data
```

```
"${./data}"
```

```
"/nix/store/h1qj5h5n05b5d15q4nldrqq8mdg7dhqk-data"
```

⁴⁷ <https://nix.dev/manual/nix/stable/language/builtin-constants.html#builtins-currentSystem>

详细说明

前述 shell 命令将字符 123 写入当前目录下的 data 文件。

上述 Nix 表达式将此文件引用为 ./data，并将文件系统路径转换为插值字符串（第23页） `${...}` 。

此类插值表达式必须能计算出可表示为字符串的值。文件系统路径即为此类值，其字符串表示形式为对应的 Nix 存储路径：

```
/nix/store/<hash>-<name>
```

Nix 存储路径通过计算文件内容的哈希值（`<hash>`）并与文件名（`<name>`）组合获得。文件会作为求值的副作用被复制到 /nix/store 目录下。若文件系统路径不存在则报错。

对于目录同样如此：整个目录（包括嵌套的文件和子目录）会被复制到 Nix 存储中，而求值后的字符串将成为该目录在 Nix 存储中的路径。

获取器

用作构建输入的文件不必来自文件系统。

Nix 语言提供了内置的非纯函数，用于在求值期间通过网络获取文件：

- `builtins.fetchurl`⁴⁸
- `builtins.fetchTarball`⁴⁹
- `builtins.fetchGit`⁵⁰
- `builtins.fetchClosures`⁵¹

这些函数会解析为 Nix 存储中的文件系统路径。

示例：

```
builtins.fetchurl "https://github.com/NixOS/nix/archive/
→ 7c3ab5751568a0bc63430b33a5169c5e4784a0ff.tar.gz"
```

```
"/nix/store/7dhgs330clj36384akg86140fgkghg2f-
-7c3ab5751568a0bc63430b33a5169c5e4784a0ff.tar.gz"
```

其中一些提供了额外的便利，例如自动解压归档文件。

示例：

```
builtins.fetchTarball "https://github.com/NixOS/nix/archive/
→ 7c3ab5751568a0bc63430b33a5169c5e4784a0ff.tar.gz"
```

```
"/nix/store/d59llm96vgis5fy231x6m7nrijs0ww36-source"
```

注：Nixpkgs 手册中关于获取器的章节⁵² 列举了大量通过网络获取文件的附加库函数。

若网络请求失败则报错。

⁴⁸ <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchurl>

⁴⁹ <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchTarball>

⁵⁰ <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchGit>

⁵¹ <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchClosure>

⁵² <https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers>

2.2.6 衍生

衍生是 Nix 和 Nix 语言的核心：

- Nix 语言用于描述衍生。
- Nix 通过运行衍生来生成构建结果。
- 构建结果又可作为其他衍生的输入。

声明衍生的 Nix 语言原语是内置非纯函数 derivation。

它通常由Nixpkgs构建机制stdenv.mkDerivation封装，该机制隐藏了非平凡构建过程中涉及的大部分复杂性。

注意：实践中您可能永远不会遇到derivation。

每当遇到mkDerivation时，都表示Nix最终会构建的内容。

示例：使用mkDerivation的软件包（第39页）

derivation（及mkDerivation）的求值结果是一个具有特定结构和特殊属性的集合（第17页）：可用于字符串插值（第23页），此时会求值为其构建结果的Nix存储路径。

示例：

让

```
pkgs = 导入 <nixpkgs> {};
于 "${pkgs.nix}"
```

```
"/nix/store/sv2srrjddrp2isghmrla8s6lazbzmikd-nix-2.11.0"
```

注：您的输出可能不同。可能会生成不同的哈希值甚至不同的软件包版本。

衍生输出路径完全由其输入决定，本例中的输入来自某个版本的Nixpkgs。

正因如此，我们建议避免使用查找路径（第147页）以确保结果可预测，除非示例仅用于演示目的。

详细说明

该示例从查找路径<nixpkgs>导入Nix表达式，并将结果函数应用于空属性集{}。其输出被命名为pkgs。

允许通过字符串插值（第23页）将属性pkgs.nix转换为字符串，因为pkgs.nix是一个派生。也就是说，最终pkgs.nix可归结为对derivation的调用。

结果字符串是该派生构建结果最终所在的文件系统路径。

关于派生内部运作机制还有更深入的内容，但现阶段只需知道这类表达式会求值为Nix存储路径即可。

推导中的字符串插值用于在声明新推导时，将其构建结果作为文件系统路径引用。

这使得能够用Nix语言构建任意复杂的推导组合。

2.2.7 实战示例

目前我们看到的都是用于演示Nix语言各种结构的人为示例。

现在你应该能阅读简单包和配置的Nix语言代码，并为以下实际案例写出类似的解析。

注意：下列练习的目的不是理解代码含义或工作原理，而是分析其如何通过函数、属性集等Nix语言数据类型构建结构。

Shell环境

```
{ pkgs ? import <nixpkgs> {} }:
让
  message = "你好世界";
in
pkgs.mkShellNoCC {
  packages = 使用 pkgs; [ cowsay ];
  shellHook =
    cowsay ${message}
  '';
}
```

此示例声明了一个Shell环境（初始化时会运行shell钩子）。

说明：

- 该表达式是一个以属性集为参数的函数。
- 若参数包含pkgs属性，则函数体内将使用该属性；否则默认导入查找路径<nixpkgs>中找到的Nix表达式（此处为函数），以空属性集调用该函数，并返回结果值。
- 名称message被绑定至字符串值"hello world"。
- pkgs集中的mkShellNoCC属性是一个以属性集为参数的函数，其返回值同时也是外层函数的结果。
- 传递给mkShellNoCC的属性集包含两个属性：packages（设置为仅含pkgs中thecowsay属性的单元素列表）和shellHook（设置为一个缩进字符串）。
- 缩进字符串包含一个插值表达式，该表达式将展开message的值以生成"hello world"。

NixOS 配置

```
{ config, pkgs, ... }: {
  imports = [ ./hardware-configuration.nix ];
  environment.systemPackages = with pkgs; [ git ];
  #...
}
```

此示例是 NixOS 配置的（部分）内容。

说明：

- 该表达式是一个函数，接受属性集作为参数，并返回一个属性集。

- 参数必须至少包含 config 和 pkgs 属性，也可包含更多属性。
- 返回的属性集会包含 imports 和 environment 属性。
- imports 是一个单元素列表：指向与此 Nix 文件同级的 hardware-configuration.nix 文件路径。

注意：此处的 imports 并非不纯的内置 import 函数，而是常规属性名！

- environment 本身是一个属性集，包含 systemPackages 属性，该属性将求值为单元素列表：即 pkgs 集中的 git 属性。
- (未见) config 参数被使用。

包

```
{ lib, stdenv, fetchurl }:

stdenv.mkDerivation rec {
    pname = "hello";
    version = "2.12";
    src = fetchurl {
        url = "mirror://gnu/${pname}/${pname}-${version}.tar.gz";
        sha256 = "layhp9v4m4rdhjmnl2bq3cibrbqqkgjbl3s7yk2nhlh8vj3ay16g";
    };
    meta = with lib; {
        license = licenses.gpl3Plus;
    };
}
```

此示例是Nixpkgs中(简化后的)软件包声明。

说明：

- 该表达式是一个函数，接收必须包含 lib、stdenv 和 fetchurl 这三个属性的属性集。
- 它返回对 stdenv 的属性 mkDerivation 求值的结果，该函数被应用于一个递归集合。
- 传递给 mkDerivation 的递归集合在 fetchurl 函数的参数中使用自身的 pname 和 version 属性，而 fetchurl 来自外部函数的参数。
- meta 属性本身是一个属性集，其中 license 属性的值被赋给了嵌套属性 lib.licenses.gpl3Plus。

2.2.8 参考文献

- Nix 手册：Nix 语言⁵³
- Nix 手册：字符串插值⁵⁴
- Nix 手册：内置函数⁵⁵
- Nix 手册：nix 仓库⁵⁶
- Nixpkgs 手册：函数参考⁵⁷
- Nixpkgs 手册：Fetchers⁵⁸

2.2.9 后续步骤

完成任务

- 使用 shell.nix 声明式 shell 环境（第9页）——通过 Nix 文件创建可复现的 shell 环境
- 用 Nix 打包现有软件（第41页）——让更多软件通过 Nix 可用

若想暂停学习 Nix，可通过以下命令清除 Nix 存储中未使用的构建结果：

```
$ nix-collect-garbage
```

了解更多

如果你完成了这些示例，你会注意到阅读 Nix 语言能揭示代码结构，但未必能说明代码的实际含义。

通常无法仅凭手头的代码确定

- 命名值或函数参数的数据类型。
- 被调用函数所接受的参数数据类型。
- 给定属性集中存在哪些属性。

示例：

```
{x, y, z} : (x y) z. a
```

我们如何知道...

- x 将是一个函数，该函数在给定参数时会返回另一个函数？
- 若 x 是函数，如何确保 y 能作为 x 的有效参数？
- 若 (xy) 是函数，如何确保 z.a 能作为 (xy) 的有效参数？
- z 是否会被定义为一个属性集？
- 假设 z 是一个属性集，它是否会包含属性 a？
- 数据类型 y 和 z.a 会是什么？
- 最终结果的数据类型是什么？

⁵³ <https://nix.dev/manual/nix/stable/language/index.html>

⁵⁴ <https://nix.dev/manual/nix/stable/language/string-interpolation.html>

⁵⁵ <https://nix.dev/manual/nix/stable/language/builtins.html>

⁵⁶ <https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-repl.html>

⁵⁷ <https://nixos.org/manual/nixpkgs/stable/#sec-functions-library>

⁵⁸ <https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers>

该函数的调用者如何知道它需要一个包含x、y、z属性的属性集？

回答这类问题需要了解给定表达式预期使用的上下文。

Nix生态系统和代码风格由惯例驱动。你在Nix语言代码中遇到的大多数名称都来自Nixpkgs：

- Nix Pills⁵⁹ - 详细解释了推导过程以及如何从基本原理构建Nixpkgs。Nixpkgs提供了广泛使用的通用构建机制：

- stdenv⁶⁰ - 最重要的是mkDerivation
- 简易构建器⁶¹ - 用于创建文件和shell脚本

Nixpkgs中的软件包可通过多种机制进行修改：

- 覆写(overrides)⁶² ——使用override和overrideAttrs对单个软件包进行特定修改
- 覆盖层(overlays)⁶³ ——通过逐个修改软件包生成自定义版本的Nixpkgs

不同语言生态和框架融入Nixpkgs时有不同需求：

- 语言与框架⁶⁴ 列出了Nixpkgs提供的工具，用于构建Nix中特定语言或框架的软件包

NixOS Linux发行版采用模块化配置系统，其自身遵循特定规范：

- NixOS 模块⁶⁵ 展示了NixOS配置的组织方式。

2.3 使用Nix打包现有软件

Nix的主要应用场景之一是解决软件打包过程中的常见难题，例如依赖项的指定与获取。

长期来看，Nix能极大缓解此类问题。但首次用Nix打包现有软件时，常会遇到看似晦涩的错误。

2.3.1 简介

本教程中，您将创建首个Nix派生⁷³ 来打包C/C++软件，利用Nixpkgs标准环境⁷⁴ (stdenv) 自动完成大部分工作。

⁵⁹ <https://nixos.org/guides/nix-pills/>

⁶⁰ <https://nixos.org/manual/nixpkgs/stable/#chap-stdenv>

⁶¹ <https://nixos.org/manual/nixpkgs/stable/#chap-trivial-builders>

⁶² <https://nixos.org/manual/nixpkgs/stable/#chap-overrides>

⁶³ <https://nixos.org/manual/nixpkgs/stable/#chap-overlays>

⁶⁴ <https://nixos.org/manual/nixpkgs/stable/#chap-language-support>

⁶⁵ <https://nixos.org/manual/nixos/stable/index.html#sec-writing-modules>

⁷³ <https://nix.dev/manual/nix/stable/language/derivations>

⁷⁴ <https://nixos.org/manual/nixpkgs/stable/#part-stdenv>

你将学到什么？

本教程从"hello"开始——一个仅需stdenv已提供依赖的"hello world"实现。接着你将构建更复杂的包及其专属依赖，逐步掌握更多派生功能的使用。

你将遇到并解决Nix错误信息、构建失败等各种问题，在此过程中培养迭代调试技巧。

需要哪些准备？

- 熟悉Unix shell和纯文本编辑器
- 应能流畅阅读Nix语言（第13页）。建议先完成教程再实践。

需要多长时间？

仔细完成所有步骤大约需要60分钟。

2.3.2 你的第一个软件包

注意：

软件包是一个宽泛定义的概念，既可指代文件及其他数据的集合，也可指代在生成前表示该集合的Nix表达式。Nixpkgs中的软件包遵循约定俗成的结构，使其能被搜索发现，并与其他软件包共同组合到环境中。

在本教程中，"软件包"指一个能求值为衍生的Nix语言函数。通过"用Nix打包现有软件"，它将使你或他人能生成可供实际使用的制品。

首先，参考这个基础推导结构：

```
2 { stdenv }:  
stdenv.mkDerivation {
```

这是一个接收包含 stdenv 的属性集合并生成推导（当前无实际功能）的函数。

包函数

GNU Hello 是"hello world"程序的实现，其源代码可通过 GNU 项目的FTP服务器⁷⁵获取。

首先，在传递给mkDerivation的集合中添加pname属性。每个包都需要名称和版本，否则Nix会报错：缺少衍生名称。

```
stdenv.mkDerivation {  
+ pname = "hello";  
+ version = "2.12.1";
```

⁷⁵ <https://ftp.gnu.org/gnu/hello/>

接下来，您将声明对最新版本hello的依赖，并指示Nix使用fetchzip下载源代码归档⁷⁶。

注意：fetchzip不仅能获取zip文件，还能下载更多类型的归档⁷⁷！

哈希值需在归档下解压后才能确定。若提供给fetchzip的哈希值不正确，Nix会报错。将hash属性设为空字符串，然后根据错误信息确定正确哈希值：

```
#hello.nix
1
{
2   stdenv
3   fetchzip,
4 }:
5
6 stdenv.mkDerivation {
7   pname = "hello";
8   version = "2.12.1";
9
10  src = fetchzip {
11    url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz";
12    sha256 = "";
13  };
14}
```

将此文件保存为 hello.nix 并运行 nix-build，即可观察到您的首次构建失败：

```
$ nix-build hello.nix
错误：无法评估一个含有未赋值参数（'stdenv'）的函数
      Nix 尝试将函数作为顶层表达式进行评估；
      此类情况下，必须通过默认值
      或显式传递 '--arg'/'--argstr' 来提供参数。详见
      https://nix.dev/manual/nix/stable/language/constructs.html#functions.

位于 /home/nix-user/hello.nix:3:3:
```

```
2 | {
3 | stdenv,
| ^
4 fetchzip,
```

问题：hello.nix 中的表达式是一个函数，只有传入正确的参数时才会生成预期输出。

使用 nix-build 构建

stdenv 可从 nixpkgs⁷⁸ 获取，需通过另一个 Nix 表达式导入才能将其作为参数传递给此派生。

推荐做法是在 hello.nix 同级目录创建 default.nix 文件，内容如下：

```
#default.nix
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
```

(接下一页)

⁷⁶ <https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz>

⁷⁷ <https://nixos.org/manual/nixpkgs/stable/#fetchurl>

⁷⁸ <https://github.com/NixOS/nixpkgs/>

(接上页)

```

6   in
7   {
8     hello = pkgs.callPackage ./hello.nix { };
9   }

```

这样你可以运行 nix-build -A hello 来实现 hello.nix 中的推导，类似于 Nixpkgs 当前的惯例。

注意：如果 pkgs 中的属性与函数参数属性集所需的属性匹配，callPackage 会自动将它们传递给该函数。本例中，callPackage 会向 hello.nix 定义的函数提供 stdenv 和 fetchzip。

教程《callPackage 的包参数与覆写》（第53页）详细解释了其工作原理。

现在用新参数运行 nix-build 命令：

```

$ nix-build -A hello
错误：固定输出推导 '/nix/store/ 中的哈希值不匹配
<pd2kiyfa0c06giparlhd1k31bvlypbb-source.drv':
    指定值: sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    实际值: sha256-1kJhtlsAkpNB7f6tZEs+dbKd8z7KoNHyDHEJ0tmhnc=
错误：推导 '/nix/store/b4mjwlv73nmiqgkdabsdj4zq9gnma11- 的 1 个依赖项出现问题
构建 hello-2.12.1.drv 失败

```

正在查找文件哈希值

正如预期，错误的文件哈希值引发了错误，而 Nix 贴心地提供了正确的哈希值。请在 hello.nix 中将空字符串替换为正确的哈希值：

```

#hello.nix
2 {
3   stdenv
4   fetchzip,
5   }: 
6 stdenv.mkDerivation {
7   pname = "hello";
8   version = "2.12.1";
9
10  src = fetchzip {
11    url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz";
12    sha256 = "sha256-1kJhtlsAkpNB7f6tZEs+dbKd8z7KoNHyDHEJ0tmhnc=";
13  };
14}
15

```

现在再次运行之前的命令：

```

$ nix-build -A hello
将构建此派生项：
/nix/store/rbq37s3r76rr77c7d8x8px7z04kw2mk7-hello.drv
正在构建 '/nix/store/rbq37s3r76rr77c7d8x8px7z04kw2mk7-hello.drv'...
...
配置中
...
configure: 正在生成 ./config.status
config.status: 正在生成 Makefile
...
构建中
... <其余多行已省略>

```

好消息：构建推导成功完成！

控制台输出显示已调用configure命令生成Makefile，随后用于构建项目。本例无需编写任何构建指令，因为stdenv构建系统基于GNU Autoconf⁷⁹，可自动检测项目目录结构。

构建结果

在工作目录查看结果：

```
§ ls
default.nix hello.nix result
```

该结果是一个指向包含已构建二进制文件的Nix存储位置的符号链接⁸⁰；你可以通过执行./result/bin/hello来运行此程序：

```
$ ./result/bin/hello
你好，世界！
```

恭喜，你已成功用Nix打包了第一个程序！

接下来，你将打包一个具有stdenv外部依赖的软件，这会带来新的挑战，需要你运用更多mkDerivation功能。

2.3.3 带依赖项的软件包

现在你将打包一个稍复杂的程序icat⁸¹，它允许你在终端中渲染图像。

修改上一节的 default.nix，为 icat 添加一个新属性：

```
#default.nix
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-24.05";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
{
  hello = pkgs.callPackage ./hello.nix {};
  icat = pkgs.callPackage ./icat.nix {};
}
```

将 hello.nix 复制为新文件 icat.nix，并更新该文件中的 pname 和 version 属性：

```
#icat.nix
{
  stdenv,
  fetchzip,
}:
6 stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";
10  src = fetchzip {
    #...
  };
13}
14}
```

⁷⁹ <https://www.gnu.org/software/autoconf/>

⁸⁰ https://en.wikipedia.org/wiki/Symbolic_link

⁸¹ <https://github.com/atextor/icat>

现在下载源代码。icat的上游仓库托管在GitHub⁸²，因此你需要替换之前的源码获取器⁸³。这次将改用fetchFromGitHub⁸⁴而非fetchzip，通过相应更新函数参数属性集来实现：

```
#icat.nix
2
{
  stdenv,
  fetchFromGitHub,
}:
6
stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";
10
  src = fetchFromGitHub {
    #...
12
  };
13
}
14
```

正在从GitHub获取源码

虽然fetchzip只需url和sha256参数，但fetchFromGitHub⁸⁵需要更多参数。

源码URL是<https://github.com/atextor/icat>，这已给出前两个参数：

- owner: 控制代码库的账户名称

```
owner = "atextor";
```

- repo: 要获取的代码库名称

```
repo = "icat";
```

前往项目的标签页⁸⁶查找合适的Git版本⁸⁷ (rev)，例如对应你要获取版本的Git提交哈希值或标签（如v1.0）。

本例中，最新发布标签是v0.5。

如hello示例所示，仍需提供哈希值。这次无需使用空字符串并让nix-build在报错时返回正确值，你可以直接通过nix-prefetch-url命令获取正确哈希。

你需要的是压缩包内容的SHA256哈希值（而非压缩包文件本身的哈希）。因此需传入--unpack和--type sha256参数：

```
$ nix-prefetch-url --unpack https://github.com/atextor/icat/archive/refs/tags/v0.5.
<tar.gz --type sha256
路径是'/nix/store/p8jl1jlqxcsc7ryiazbpm7c1mqb6848b-v0.5.tar.gz'
0wy2ksxp95vh71ybj1bbmqd5ggp13x3mk37pqr991js9awy8ka
```

为fetchFromGitHub设置正确的哈希值：

```
#icat.nix
2
{
  stdenv,
  fetchFromGitHub,
```

(接续下一页)

⁸² <https://github.com/atextor/icat>

⁸³ <https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers>

⁸⁴ <https://nixos.org/manual/nixpkgs/stable/#fetchfromgithub>

⁸⁵ <https://nixos.org/manual/nixpkgs/stable/#fetchfromgithub>

⁸⁶ <https://github.com/atextor/icat/tags>

⁸⁷ <https://git-scm.com/docs/revisions>

(continued from previous page)

```

}:

6 stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

10 src = fetchFromGitHub {
    owner = "atextor";
    repo = "icat";
    rev = "v0.5";
    sha256 = "0wy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pqr991js9awy8ka";
};

16 };

17 }

```

缺少依赖项

使用新的 icat 属性运行 nix-build 时，报告了一个全新问题：

```
$ nix-build -A icat
将构建以下 2 个派生项：
/nix/store/86q9x927hsyyzfr4lcqirmsbimysi6mb-source.drv
/nix/store/15wz9inkvkf0qh18kp139vpg2xfm2qpy-icat.drv

错误：构建 '/nix/store/15wz9inkvkf0qh18kp139vpg2xfm2qpy-icat.drv' 失败_
→ 退出代码 2;
      最后 10 行日志：
      > 来自 /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp41-glibc-2.
→ 37 – 8 – dev/include/stdio.h:27,
      >
      > /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp41-glibc-2.37-8-dev/include/
<features.h:195:3: 警告: #warning "_BSD_SOURCE 和 _SVID_SOURCE 已弃用,
• 请改用 _DEFAULT_SOURCE" [8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
-#index-Wcpp-Wcpp8;;]
      > 1 ~~~~~
      icat.c:39:10: 致命错误: Imlib2.h: 没有那个文件或目录
          39 | #include <Imlib2.h>
      >
      编译终止。
      make: *** [Makefile:16: icat.o] 错误 1
      查看完整日志，请运行 'nix log /nix/store/15wz9inkvkf0qh18kp139vpg2xfm2qpy-
→ icat.drv'。
```

编译器报错！icat源码从GitHub拉取后，Nix尝试构建时因缺失imlib2头文件依赖导致编译失败。

若在search.nixos.org⁸⁸搜索imlib2，你会发现该库已存在于Nixpkgs中。

将imlib2添加至icat.nix函数的参数列表，再将该参数值imlib2加入stdenv.mkDerivation的buildInputs列表即可引入构建环境：

```
#icat.nix
{
  stdenv,
  从GitHub获取,
  imlib2,
};
```

(接续下一页)

⁸⁸ <https://search.nixos.org/packages?query=imlib2>

(接上页)

```

7 stdenv.mkDerivation {
8   pname = "icat";
9   version = "v0.5";
10
11   src = fetchFromGitHub {
12     owner = "atextor";
13     repo = "icat";
14     rev = "v0.5";
15     sha256 = "0wy2ksxp95vnh71ybj1bbmqd5ggp13x3mk37pqr991js9awy8ka";
16   };
17
18   buildInputs = [ imlib2 ];
19 }
```

再次运行 nix-build -A icat 会遇到另一个错误，但这次编译会走得更远：

```

$ nix-build -A icat
将构建以下派生：
/nix/store/bw2d4rp2k115rg49hds199ma2mz36x47-icat.drv

错误：'/nix/store/bw2d4rp2k115rg49hds199ma2mz36x47-icat.drv' 的构建器失败_
→ 退出码为 2;
    最后 10 行日志：
    > 来自 icat.c:31:
    > /nix/store/hkj250rjsvxcbr31fr1v81cv88cdfp41-glibc-2.37-8-dev/include/
</factures.h:195:3: 警告: #warning "_BSD_SOURCE 和 _SVID_SOURCE 已被弃用,
• 请改用 _DEFAULT_SOURCE" [8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
→ #index-Wcpp-Wcpp8;;]
    > 195 | # warning "_BSD_SOURCE 和 _SVID_SOURCE 已被弃用, 请使用 _
→ DEFAULT_SOURCE"
    >
    > 包含自 icat.c 文件第39行:
    > /nix/store/4fvrh0sjc8sbkbqda7dfsh7q0gxmnh9p-imlib2-1.11.1-dev/include/
<Inib2.h:45:10: 致命错误: X11/Xlib.h: 没有那个文件或目录
    > 45 | #include <X11/Xlib.h>
    >
    > 编译终止。
    > make: *** [Makefile:16: icat.o] 错误 1
    查看完整日志, 请运行 'nix log /nix/store/bw2d4rp2k115rg49hds199ma2mz36x47-
→ icat.drv'。
```

您会看到一些应在上游代码中修正的警告。但本教程的关键是致命错误：X11/Xlib.h：没有那个文件或目录：这表示缺少另一个依赖项。

2.3.4 查找软件包

目前确定依赖项的来源有些复杂，因为软件包名称并不总是与库或程序名称对应。

您需要从X11 C包中获取Xlib.h头文件，其在Nixpkgs中的派生名为libX11，位于xorg软件包集中。有几种方法可以解决这个问题：

search.nixos.org

提示：查找所需内容最便捷的方式是访问 search.nixos.org/packages⁸⁹。

但本例中，搜索 `x11`⁹⁰ 会返回过多无关结果，因为 X11 普遍存在。左侧边栏有软件包集列表，选择 `xorg`⁹¹ 会显示有效线索。
若其他方法均无效，熟悉如何在 Nixpkgs 源代码⁹² 中搜索关键词会很有帮助。

本地代码搜索

要在源码中查找名称赋值，可搜索"`<关键词> =`"。例如，这是 Github 上搜索 "`x11 = " "3` 或 `"libx11 = " "4` 的结果。
或克隆Nixpkgs仓库⁹⁵的副本，然后在本地代码中搜索。

启动一个提供所需工具的shell环境——版本控制用的git，以及代码搜索用的rg（由 ripgrep包⁹⁶ 提供）：

```
$ nix-shell -p git ripgrep
[nix-shell:~]$
```

Nixpkgs仓库体积庞大。仅克隆最新版本以避免长时间等待完整克隆：

```
[nix-shell:~]$ git clone https://github.com/NixOS/nixpkgs --depth 1
...
[nix-shell:~]$ cd nixpkgs
```

为缩小搜索范围，仅检索包含所有软件包配置的pkgs子目录：

```
[nix-shell:~]$ rg "x11 =" pkgs
pkgs/tools/X11/primus/default.nix
21: primus = if useNvidia then primusLib_ else primusLib_.override { nvidia_x11 =_
→ null; };
22: primus_i686 = 若使用Nvidia则取primusLib_i686_否则取primusLib_i686_.override
-{ nvidia_x11 = null; };

pkgs/应用程序/图形/imv/default.nix
38: x11 = [ libGLU xorg.libxch xorg.libX11 ];

pkgs/工具/X11/primus/lib.nix
14: 若nvidia_x11为null则取libGL

pkgs/top-level/linux-kernels.nix
573: ati_drivers_x11 = throw "ATI驱动已不再受任何内核支持...
-->=4.1"; # 添加于2021-05-18;
... <更多结果>
```

由于rg默认区分大小写，添加-i参数确保不遗漏任何结果：

⁸⁹ [http://search.nixos.org/packages](https://search.nixos.org/packages)

⁹⁰ <https://search.nixos.org/packages?query=x11>

⁹¹ https://search.nixos.org/packages?buckets=%7B%22package_attr_set%22%3A%5B%22xorg%22%5D%2C%22package_license_set%22%3A%5B%5D%2C%22package_maintainers_set%22%3A%5B%5D%2C%22package_platforms%22%3A%5B%5D%7D&query=x11

⁹² <https://github.com/nixos/nixpkgs>

⁹³ <https://github.com/search?q=repo%3ANixOS%2Fnixpkgs+%22x11%22&type=code>

⁹⁴ <https://github.com/search?q=repo%3ANixOS%2Fnixpkgs+%22libx11%22&type=code>

⁹⁵ <https://github.com/nixos/nixpkgs>

⁹⁶ <https://search.nixos.org/packages?show=ripgrep>

```
[nix-shell:~] $ rg -i "libx11 =" pkgs
pkgs/applications/version-management/monotone-viz/graphviz-2.0.nix
55: ++ lib.optional (libX11 == null) "-without-x";

pkgs/top-level/all-packages.nix
14191: libX11 = xorg.libX11;

pkgs/servers/x11/xorg/default.nix
1119: libX11 = callPackage ({ stdenv, pkg-config, fetchurl, xorgproto,
slippthreadstubs, libxcb, xtrans, testers }: stdenv.mkDerivation (finalAttrs: {

pkgs/servers/x11/xorg/overrides.nix
147: libX11 = super.libX11.overrideAttrs (attrs: {
```

本地派生搜索

要在命令行搜索派生，请使用 nix-index⁹⁷ 中的 nix-locate 工具。

将软件包集添加为依赖项

将 xorg 添加到您的 derivation 输入属性集中，并在 buildInputs 中使用 xorg.libX11：

```
#icat.nix
{
  stdenv
  fetchFromGitHub,
  imlib2,
  xorg,
}:

stdenv.mkDerivation {
  pname = "icat";
  version = "v0.5";

src = fetchFromGitHub {
  owner = "atextor";
  repo = "icat";
  rev = "v0.5";
  sha256 = "0wy2ksxp95vnh71ybjlbbmqd5ggp13x3mk37pzs99ljs9awy8ka";
};

buildInputs = [ imlib2 xorg.libX11 ];
}
```

注：由于 Nix 语言采用惰性求值，仅访问 xorg.libX11 意味着 xorg 属性集的其他内容不会被处理。

⁹⁷ <https://github.com/nix-community/nix-index>

2.3.5 修复构建失败

重新运行上一条命令：

```
§ nix-build -A icat
即将构建此派生：
/nix/store/x1d791d8jxqqla5zw2b47d2s187mf56k-icat.drv

错误：'/nix/store/x1d791d8jxqqla5zw2b47d2s187mf56k-icat.drv' 的构建器失败
退出码2表示什么；
    最后10行日志：
        > 195 | # 警告"_BSD_SOURCE和_SVID_SOURCE已弃用，请改用_DEFAULT_SOURCE"
<DEFAULT_SOURCE"
        >
        > icat.c: 在函数'main'中：
        > icat.c:319:33: 警告: 忽略'write'声明的返回值，该函数声明带有...
→ 属性 'warn_unused_result' [8;https://gcc.gnu.org/onlinedocs/gcc/Warning-
<Options.html#index-Wunused-result-Wunused-result8;;]
        > 319 |                               write(tempfile, &buf, 1);
        >                                         →
        > gcc -o icat icat.o -lImlib2
        > 正在安装
        > 安装参数: SHELL=/nix/store/8fv91097mbh5049i9rglc73dx6kjg3qk-bash-5.2-
→ p15/bin/bash 安装
        > 编译错误：*** 没有规则可以创建目标 'install'。停止。
        查看完整日志，请运行 'nix log /nix/store/xld79ld8jxqdl45zw2b47d2s187mf56k-
→ icat.drv'。
```

缺失依赖项的问题已解决，但出现新问题：编译错误：*** 没有规则可以创建目标 'install'。停止。

安装阶段

stdenv 正自动处理 icat 附带的 Makefile。控制台输出显示 configure 和 make 均顺利执行，因此 icat 二进制文件正在成功编译。

失败发生在 stdenv 尝试运行 make install 时。该项目自带的 Makefile 恰好缺少 install 目标。icat 仓库的 README 仅提及使用 make 构建工具，将安装步骤留给用户自行处理。

若要将此步骤加入您的 derivation，请使用 installPhase 属性⁹⁸。它包含执行安装操作所需的命令字符串列表。

由于 make 已成功完成，icat 可执行文件存在于构建目录中。您只需将其从该目录复制到输出目录即可。

在 Nix 中，输出目录存储在 \$out 变量中。该变量可在 derivation 的 builder 执行环境⁹⁹ 中访问。请在 fout 目录内创建 bin 目录，并将 icat 二进制文件复制至此：

```
#icat.nix
{
  标准环境,
  从GitHub获取,
  imlib2库,
  xorg,
}:
标准环境.mkDerivation {
  包名 = "icat";
```

(接下一页)

⁹⁸ <https://nixos.org/manual/nixpkgs/stable/#ssec-install-phase>

⁹⁹ <https://nix.dev/manual/nix/2.19/language/derivations#builder-execution>

(continued from previous page)

```

12  版本 = "v0.5";
13
14  源 = 从GitHub获取 {
15      所有者 = "atextor";
16      仓库 = "icat";
17      版本号 = "v0.5";
18      sha256 = "0wy2ksxp95vh71ybj1bbmqd5ggp13x3mk37pqr991js9awy8ka";
19  };
20
21  buildInputs = [ imlib2 xorg.libX11 ];
22
23  installPhase =
24      mkdir -p $out/bin
25      cp icat $out/bin
26
}

```

构建阶段与钩子

Nixpkgs stdenv.mkDerivation 派生过程被划分为多个阶段¹⁰⁰，每个阶段旨在控制构建过程的某些方面。

此前您已观察到，stdenv.mkDerivation要求项目的Makefile必须包含install目标，否则会构建失败。为此，您定义了一个自定义的installPhase，其中包含将icat二进制文件复制到正确输出位置的指令，从而实现了安装功能。在此之前，stdenv.mkDerivation始终自动确定icat包的buildPhase信息。

在实现派生过程时，每个构建阶段都可能执行若干Shell函数（Nixpkgs中称为“hooks”）。这些钩子函数负责设置变量、引入文件、创建目录等操作。

这些钩子函数是各阶段特有的，会在阶段执行前后运行。它们通过修改构建环境来支持构建过程中的常规操作。

使用Nix打包软件时，最佳实践是在自定义的派生阶段调用这些钩子函数——即使您并不直接使用它们。这样便于后续¹⁰¹轻松覆盖派生的特定部分，同时保持代码整洁易读。

请调整您的installPhase以调用相应钩子：

```

#icat.nix
2
#...
4
    安装阶段 =
        运行钩子 preInstall
        创建目录 -p $out/bin
        复制 icat 到 $out/bin
        运行钩子 postInstall
10
    '';
11
#...

```

¹⁰⁰ <https://nixos.org/manual/nixpkgs/stable/#sec-stdenv-phases>

¹⁰¹ <https://nixos.org/manual/nixpkgs/stable/#chap-overrides>

2.3.6 构建成功

再次运行 nix-build 命令将最终达成目标，且可重复执行。在本地目录调用 ls 可找到指向 Nix 存储位置的 result 符号链接：

```
§ ls
default.nix hello.nix icat.nix result
```

result/bin/icat 是先前构建的可执行文件。成功！

2.3.7 参考资料

- Nixpkgs 手册 - 标准环境¹⁰²

2.3.8 后续步骤

- 使用 callPackage 进行包参数与覆盖（第53页）
- 开发环境中的依赖项（第137页）
- 使用 direnv 自动激活环境（第136页）
- 配置 Python 开发环境（第140页）
- 将您自己的新软件包加入 Nixpkgs¹⁰³
 - 如何贡献（第167页）
 - 如何获取帮助（第169页）

2.4 使用 callPackage 实现包参数与覆写

Nix 自带一门专用于创建软件包与配置的编程语言：Nix 语言。它被用来构建 Nix 软件包集合，即 Nixpkgs。

作为纯函数式语言，Nix 允许声明自定义函数以抽象通用模式。Nixpkgs 中最显著的模式之一便是软件包配方的参数化。

2.4.1 概述

Nixpkgs本身是一个规模可观的软件项目，经过多年发展形成了独特的编码规范与惯用模式。它确立了一种通过名为callPackage¹⁰⁵ 的函数来自动配置参数化包的惯例¹⁰⁴。本教程将展示其使用方法及优势。

¹⁰² <https://nixos.org/manual/nixpkgs/unstable/#part-stdevn>

¹⁰³ <https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md>

¹⁰⁴ <https://github.com/nixos/nixpkgs/commit/d17f0f9cbc38fabbb71624f069cd4c0d6feace92>

¹⁰⁵ <https://github.com/NixOS/nixpkgs/commit/fd268b4852d39c18e604c584dd49a611dc795a9b>

你将学到什么？

- 使用 callPackage 调用遵循 Nixpkgs 规范的包配方
- 覆盖包参数
- 创建相互依赖的包集合

你需要什么？

- 熟悉 Nix 语言（第13页）
- 首次体验打包现有软件（第41页）

需要多长时间？

- 45分钟

2.4.2 自动函数调用

新建文件hello.nix，这是一个典型的Nixpkgs包配方：该函数接收一个属性集（其属性对应顶层包集中派生项），并返回一个派生项。

清单1：hello.nix

```
{ writeShellScriptBin }:
writeShellScriptBin "hello" "
  echo "Hello, world!"
```

详细说明

hello.nix声明了一个函数，其参数为包含writeShellScript-Bin元素的属性集。

writeShellScriptBin¹⁰⁶是Nixpkgs中存在的函数，作为构建辅助工具¹⁰⁷，它返回一个衍生项。此衍生项的输出包含位于Sout/bin/hello的可执行shell脚本，运行时会打印"Hello world"。

现在创建default.nix文件，内容如下：

清单2：default.nix

```
let
  pkgs = import <nixpkgs> {};
in
pkgs.callPackage ./hello.nix { }
```

在default.nix中实现派生并运行生成的可执行文件：

```
$ nix-build
$ ./result/bin/hello
你好，世界！
```

当hello.nix中的函数被求值时，参数writeShellScriptBin会自动填充。对于函数参数中的每个属性，若pkgs属性集中存在对应属性，callPackage会将其传入。

¹⁰⁶ <https://nixos.org/manual/nixpkgs/unstable/#trivial-builder-writeShellScriptBin>

¹⁰⁷ <https://nixos.org/manual/nixpkgs/unstable/#part-builders>

在这种简单配置中为软件包创建额外的hello.nix文件可能显得繁琐。我们这样做是因为Nixpkgs正是这样组织的：每个软件包配方都是一个声明函数的文件，该函数以软件包的依赖项作为参数。

2.4.3 参数化构建

修改default.nix以生成派生属性集，其中hello属性包含原始派生：

清单3： default.nix

```
let
  pkgs = import <nixpkgs> { };
in
{
  hello = pkgs.callPackage ./hello.nix { };
}
```

当通过 `-A` / `--` 属性选项¹⁰⁸ 访问并构建hello属性时，结果将与之前相同：

```
§ nix-build -A hello
$ ./result/bin/hello
你好，世界！
```

同时修改hello.nix，添加一个默认值为"world"的额外参数audience：

清单4： hello.nix

```
{
  writeShellScriptBin,
  audience ? "world",
};

writeShellScriptBin "hello" "
  echo "Hello, ${audience}!""
...
```

这同样不会改变结果。

当修改default.nix以利用这个新参数时，事情变得更有趣了。在callPackage的第二个参数中传递audience参数：

清单5： default.nix

```
let
  pkgs = import <nixpkgs> { };
in
{
  -• hello = pkgs.callPackage ./hello.nix { };
  + hello = pkgs.callPackage ./hello.nix { audience = "people"; };
}
```

该属性会传递给hello.nix中定义的函数参数：同样的语法也可用于显式设置自动发现的参数（如writeShellScriptBin），但在此场景下并无意义。

尝试执行：

¹⁰⁸ <https://nix.dev/manual/nix/2.19/command-ref/nix-build#opt-attr>

```
$ nix-build -A hello
$ ./result/bin/hello
你好，人们！
```

这种模式在Nixpkgs中被广泛使用：例如，表示Go程序的函数通常有一个参数`buildGoModule`。常见到类似`callPackage ./go-program.nix { build-GoModule = buildGo116Module; }`的表达式，用于更改默认的Go编译器版本。因此Nixpkgs不只是一个预配置包的庞大库，而是一系列函数的集合——这些函数作为包的配方，能够动态定制包甚至整个生态系统（例如“所有使用我自定义解释器的Python包”），而无需重复代码。

2.5 覆写机制

`callPackage`通过允许使用返回派生对象的`override`函数在事后自定义参数，提供了更多便利。

在`default.nix`中添加第三个属性`hello-folks`，并将其设置为调用`hello.override`并传入新参数`audience`的值：

清单6：`default.nix`

```
let
  pkgs = import <nixpkgs> { };
in
- {
+rec {
  hello = pkgs.callPackage ./hello.nix { audience = "people"; };
+ hello-folks = hello.override { audience = "folks"; };
}
```

注意：生成的属性集现在是递归的（通过关键字`rec`）。这意味着属性值可以引用同一属性集内的名称。

`override`将`audience`传递给`hello.nix`中的原始函数——它会覆盖原始`callPackage`中生成派生`hello`时传入的任何参数，其余参数保持不变。这一特性尤其实用，常见于提供多种自定义选项的包中。

构建`hello-folks`属性并运行生成的可执行文件，将再次生成脚本的新版本：

```
§ nix-build -A hello-folks
§ ./result/bin/hello
大家好！
```

一个现实案例是 neovim¹⁰⁹ 的包配方，它包含可覆盖参数如`extraLuaPackages`、`extraPythonPackages`或`withRuby`。目前这些参数只能通过阅读源代码发现，而源代码可通过search.nixos.org/packages上的“2 Source”链接¹¹⁰找到。

¹⁰⁹ <https://search.nixos.org/packages?show=neovim>

¹¹⁰ <https://search.nixos.org/packages>

2.5.1 相互依赖的包集合

您实际上可以创建自己的callPackage版本！这在处理相互依赖的包集合时非常实用。

注意：以下示例未展示"被调用"文件，因其对理解原理并非必需。

请看以下递归推导属性集：

代码清单7：default.nix

```
let
  pkgs = import <nixpkgs> {};
in
rec {
  a = pkgs.callPackage ./a.nix {};
  b = pkgs.callPackage ./b.nix { inherit a; };
  c = pkgs.callPackage ./c.nix { inherit b; };
  d = pkgs.callPackage ./d.nix {};
  e = pkgs.callPackage ./e.nix { 继承 c d; };
}
```

注意：此处的 `inherit a;` 等价于 `a = a;`。

先前声明的派生会通过 callPackage 作为参数传递给其他派生。

这种情况下，你必须记得在每个 Nix 文件中手动指定所有未被 Nixpkgs 包含的包所需参数。若 `./b.nix` 需要参数 `a` 但不存在 `pkgs.a`，函数调用将报错。对于较大的包集合，这会很快变得相当繁琐。

使用 `lib.callPackageWith` 基于属性集创建自定义的 callPackage。

清单8：default.nix

```
让
pkgs = 导入 <nixpkgs> {};
callPackage = pkgs.lib.callPackageWith (pkgs // packages);
packages = {
  a = callPackage ./a.nix {};
  b = callPackage ./b.nix {};
  c = callPackage ./c.nix {};
  d = callPackage ./d.nix {};
  e = callPackage ./e.nix {};
};
in
软件包集
```

这需要稍加说明。

首先注意，我们操作的名称现在是通过let绑定分配的，而非递归属性集。它具有与递归集相同的特性：等号(=)左侧的名称可以在右侧表达式中使用。正是通过这种方式，我们才能在用//运算符将其内容与预先存在的属性集pkgs合并时引用这些软件包。

您的自定义调用 Packages 现在将使 pkgs 和 packages 中的所有属性对调用的 packagefunction 可用（同名属性以 packages 中的优先），且 packages 会在每次调用时递归构建。

最后一点可能令人费解。这种构造之所以可行，是因为 Nix 语言采用惰性求值——即仅在真正需要时才计算值。它允许传递尚未完全定义的 packages。

每个包的依赖关系在此层级已隐式体现（在各包文件中仍显式声明），callPackage 会自动解析这些依赖。这使您无需手动处理它们，并避免了可能直到冗长构建过程后期才暴露的配置错误。

当然这个小例子在原形式下仍可管理。但对于不熟悉惰性求值的开发者，隐式递归变量可能模糊代码结构，使其比原先更难阅读。但这一优势在大型构建中真正显现——当代码量本身会遮蔽结构时，当手动修改变得繁琐且易错时。

2.5.2 小结

使用 callPackage 不仅遵循 Nixpkgs 惯例（使有经验的 Nix 用户更易理解您的代码），还免费带来以下优势：

1. 参数化构建
2. 可覆盖式构建
3. 相互依赖包集的简洁实现

2.5.3 参考资料

- Nixpkgs 手册：callPackageWith¹¹¹

2.5.4 后续步骤

- 处理本地文件（第58页）——学习用Nix打包自己的项目
- 模块系统深入（第76页）——掌握NixOS背后的函数式编程魔法

2.6 处理本地文件

要在 Nix 派生中构建本地项目，其构建器可执行文件¹¹² 必须能访问源文件。由于默认情况下，构建器在隔离环境¹¹³ 中运行（该环境仅允许从 Nix 存储读取），Nix 语言内置了将本地文件复制到存储并暴露结果存储路径的功能。

但直接使用这些功能可能很棘手：

- 将路径强制转换为字符串（如常见的 src = ./ 模式）会使派生依赖于当前目录名称。此外，它总是将整个目录添加到存储中（包括不需要的文件），当这些文件变更时会导致不必要的重新构建。
- builtins.path¹¹⁴ 函数（及等效的 lib.sources.cleanSourceWith¹¹⁵）可解决这些问题。但通过过滤器函数接口往往难以表达所需的路径选择逻辑。

本教程将教你如何在派生中使用 Nixpkgs 的 lib.filesset 库¹¹⁶ 处理本地文件。它基于内置功能进行封装，提供了更安全便捷的接口。

¹¹¹ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.customisation.callPackageWith>

¹¹² <https://nix.dev/manual/nix/stable/language/derivations#attr-builder>

¹¹³ <https://nix.dev/manual/nix/stable/command-ref/conf-file.html#conf-sandbox>

¹¹⁴ <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-path>

¹¹⁵ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.sources.cleanSourceWith>

¹¹⁶ <https://nixos.org/manual/nixpkgs/stable/#sec-functions-library-filesset>

2.6.1 文件集

文件集是一种表示本地文件集合的数据类型。可通过库提供的各种函数对文件集进行创建、组合和操作。

您可以使用 nix repl¹¹⁷ 探索并学习该库：

```
$ nix repl -f channel:nixos-23.11
...
nix-repl> fs = lib.fileset
```

trace¹¹⁸ 函数会美观地打印给定文件集中包含的文件：

```
nix-repl> fs.trace ./null
跟踪: /home/user (目录下所有文件)
null
```

所有需要文件集作为参数的函数也可接受路径¹¹⁹。此类路径参数会隐式转换为包含该路径下所有文件的集合¹²⁰。前述跟踪结果中的（目录下所有文件）即表示此意。

提示：trace函数会美化输出其第一个参数并返回第二个参数。但在nix repl中若仅需美化输出，可省略第二参数：

```
nix-repl> fs.trace ./
跟踪: /home/user (目录下所有文件)
<lambd@/nix/store/1c278x24s3b16qdnifpvm5z03wfi2p-nixpkgs-src/lib/filesset/
<default.nix:555:8>
```

尽管文件集在概念上包含本地文件，但这些文件除非被显式请求，否则永远不会被添加到Nix存储中。因此您无需过度担心意外将敏感信息复制到全局可读的存储中。

本例中，虽然我们漂亮地打印了主目录路径，但并未复制任何文件。这与将路径强制转换为字符串（如“\${. /}”）形成鲜明对比——后者在求值时会将整个目录复制到Nix存储！

警告：当使用flakes和nix-command实验特性¹²¹时，Flake中的本地目录除非是Git仓库，否则总是会被完整复制到Nix存储中！

这种隐式强制转换同样适用于文件：

```
§ touch some-file
```

```
nix-repl> fs.trace ./some-file
trace: /home/user
trace: - some-file (regular)
```

除了包含的文件外，此处还会输出其文件类型¹²²。

¹¹⁷ <https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-repl>
¹¹⁸ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.trace>
¹¹⁹ <https://nix.dev/manual/nix/stable/language/values#type-path>
¹²⁰ <https://nixos.org/manual/nixpkgs/stable/#sec-files-set-path-coercion>
¹²¹ <https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake>
¹²² <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-readFileType>

2.6.2 示例项目

要进一步试验该库，可创建一个示例项目。新建目录并进入，随后使用npins工具固定Nixpkgs依赖：

```
$ mkdir fileset
$ cd fileset
$ nix-shell -p npins --run "npins init --bare; npins add github nixos nixpkgs --
-branch nixos-23.11"
```

随后创建一个包含以下内容的 default.nix 文件：

清单9： default.nix

```
{
  system ? builtins.currentSystem,
  sources ? import ./npins,
}:
let
  pkgs = import sources.nixpkgs {
    config = {};
    overlays = [];
    继承系统;
  };
in
pkgs.callPackage ./build.nix {}
```

添加两个源文件以进行操作：

```
$ echo hello > hello.txt
$ echo world > world.txt
```

2.6.3 将文件添加至Nix存储

通过toSource¹²³ 可将指定文件集中的文件添加至Nix存储。该函数需以root属性作为参数，以确定将哪个源目录复制到存储中。只有fileset属性中的文件会被包含在结果中。

按如下方式定义build.nix：

清单10： build.nix

```
{ stdenv, lib }:
let
  fs = lib.fileset;
  sourceFiles = ./hello.txt;
in
fs.trace sourceFiles

stdenv.mkDerivation {
  name = "文件集";
  src = fs.toSource {
    root = ./;
    fileset = sourceFiles;
  };
  postInstall = "
    mkdir $out
    cp -v hello.txt $out
  ";
}
```

(接下一页)

¹²³ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.toSource>

(接上页)

```

";
}
```

调用 `fs.trace` 会打印将用作推导输入的文件集。

尝试构建它：

注意：首次获取 Nixpkgs 需要较长时间。

```
$ nix-build
trace: /home/user/fileset
跟踪: - hello.txt (常规文件)
将构建此派生:
/nix/store/3ci6avmjaijx5g8jhb218i183xi7bi2n-fileset.drv
...
'hello.txt' -> '/nix/store/sa4g6h13v0zbpfw6pzva860kp5aks44n-fileset/hello.txt'
...
/nix/store/sa4g6h13v0zbpfw6pzva860kp5aks44n-fileset
```

但文件集库的真正优势在于其以不同方式组合文件集的功能。

2.6.4 差异

若需同时复制 `hello.txt` 和 `world.txt` 到输出目录，请再次将整个项目目录添加为源：

清单11：build.nix

```
{ stdenv, lib }:
let
  fs = lib.fileset;
  sourceFiles = ./hello.txt;
  • sourceFiles = ./;
in
fs.trace 源文件

stdenv.mkDerivation {
  name = "文件集";
  src = fs.toSource {
    根目录 = ./;
    文件集 = sourceFiles;
  };
  安装后操作 =
    创建目录 $out
    复制 -v hello.txt 至 $out
    复制 -v {hello,world}.txt 至 $out
  ";
}
```

这将按预期工作：

```
§ nix构建
跟踪: /home/user/fileset (目录内所有文件)
将构建此派生项:
/nix/store/fsihp8872vv9ngbkc7si5jcbigs81727-fileset.drv
...
'hello.txt' -> '/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset/hello.txt'
```

(接下一页)

(接上页)

```
'world.txt' -> '/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset/world.txt'
...
/nix/store/wmsxfgbylagmf033nkazr3qfc96y7mwk-fileset
```

但若再次运行 nix-build，输出路径将不同！

```
$ nix-build
跟踪: /home/user/fileset (目录内所有文件)
将构建以下派生:
/nix/store/nlh7ismrf27xsn13m20vfz6rvwlbca-fileset.drv
...
'hello.txt' -> '/nix/store/xknflcvja8dj6a6vkg629zmcrgz10rh-fileset/hello.txt'
'world.txt' -> '/nix/store/xknflcvja8dj6a6vkg629zmcrgz10rh-fileset/world.txt'
...
/nix/store/xknflcvja8dj6a6vkg629zmcrgz10rh-fileset
```

此处问题在于，nix-build 默认会在工作目录创建指向刚生成存储路径的 result 符号链接：

```
$ ls -1 result
result -> /nix/store/xknflcvja8dj6a6vkg629zmcrgz10rh-fileset
```

由于src指向整个目录，且其内容会随nix-build成功而改变，Nix每次都必须重新开始。

注意：不使用文件集库时也会发生此情况，例如直接设置src = ./;时。

¹²⁴ 函数用于从一个文件集中减去另一个文件集，结果将生成包含第一个参数中存在但第二个参数中不存在所有文件的新文件集。

通过修改sourceFiles定义来过滤掉./result：

清单12：build.nix

```
{ stdenv, lib }:
let
  fs = lib.fileset;
  sourceFiles = ./;
+ sourceFiles = fs.difference ./ ./result;
in
```

构建时，文件集库将指定从目录中选取哪些文件：

```
$ nix-build
跟踪: /home/user/fileset
跟踪: - build.nix (常规文件)
跟踪: - default.nix (常规文件)
跟踪: - hello.txt (常规文件)
跟踪: - npins (目录下所有文件)
跟踪: - world.txt (常规文件)
将构建此派生:
/nix/store/zr19bv51085zz005yk7pw4s9sglmafvn-fileset.drv
...
'hello.txt' -> '/nix/store/vhyhk6ij39gjapqavz1jlx3zbiy3qc1a-fileset/hello.txt'
'world.txt' -> '/nix/store/vhyhk6ij39gjapqavz1jlx3zbiy3qc1a-fileset/world.txt'
...
/nix/store/vhyhk6ij39gjapqavz1jlx3zbiy3qc1a-fileset
```

重复构建将直接复用现有存储路径：

¹²⁴ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.difference>

```
$ nix-build
跟踪: /home/user/fileset
跟踪: - build.nix (常规文件)
跟踪: - default.nix (常规文件)
跟踪: - hello.txt (常规文件)
跟踪: - npins (目录下所有文件)
跟踪: - world.txt (常规文件)
/nix/store/vhyhk6ij39gjapqavz1jlx3zbiy3qc1a-fileset
```

2.6.5 缺失文件

然而，删除 `./result` 符号链接会引发新问题：

```
$ rm result
$ nix-build
错误: lib.fileset.difference: 第二个参数 (负集)
(/home/user/fileset/result) 是不存在的路径。
若要从可能不存在的路径创建文件集，请使用 `lib.fileset.
maybeMissing`。
```

遵循错误提示中的说明，使用 `maybeMissing`¹²⁵ 从可能不存在的路径创建文件集（此时文件集将为空）：

清单13：build.nix

```
{ stdenv, lib }:
let
  fs = lib.fileset;
• sourceFiles = fs.difference ./ ./result;
+ sourceFiles = fs.difference ./ (fs.maybeMissing ./result);
in
```

现在可以正常工作了，由于 `./result` 不存在，故使用整个目录：

```
$ nix-build
跟踪: /home/user/fileset (目录下所有文件)
将构建此派生:
/nix/store/zr19bv51085zz005yk7pw4s9sglmfvn-fileset.drv
...
/nix/store/vhyhk6ij39gjapqavz1jlx3zbiy3qc1a-fileset
```

再次构建将生成不同的跟踪记录，但输出路径相同：

```
$ nix-build
跟踪: /home/user/fileset
跟踪: - build.nix (常规文件)
跟踪: - default.nix (常规文件)
跟踪: - hello.txt (常规文件)
跟踪: - npins (目录下所有文件)
跟踪: - world.txt (常规文件)
/nix/store/vhyhk6ij39gjapqavz1jlx3zbiy3qc1a-fileset
```

¹²⁵ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.maybeMissing>

2.6.6 联合（显式排除文件）

仍存在一个问题：修改任何被包含文件都会导致派生被重新构建，尽管它并不依赖这些文件。

在 build.nix 末尾追加空行：

```
$ echo >> build.nix
```

再次地，Nix 会从头开始构建：

```
$ nix-build
跟踪: /home/user/fileset
跟踪: - default.nix (常规文件)
跟踪: - npins (目录下所有文件)
跟踪: - build.nix (常规文件)
跟踪: - string.txt (常规文件)
将构建此派生:
/nix/store/zmngpqlpfz2jq0w9rdacsnp8ni4n77cn-filesets.drv
...
/nix/store/6pffjljy3c7kla60nljk3fad4qkkzn-filesets
```

解决方法之一是使用并集¹²⁶。

创建一个包含待排除文件并集 (fs.unions [...]) 的文件集，再从完整目录 (./) 中减去 (difference) 该集合：

代码清单14：build.nix

```
sourceFiles =
  文件系统差异
  ./.
  (文件系统联合 [
    (可能缺失的文件系统 ./result)
    ./default.nix
    ./build.nix
    ./npins
  ]);
};
```

这将按预期工作：

```
$ nix-build
跟踪: /home/user/fileset
跟踪: - hello.txt (常规文件)
跟踪: - world.txt (常规文件)
将构建此派生项:
/nix/store/gr2hw3gdjc28fmv0as1ikpj71ya4r51f-fileset.drv
...
/nix/store/ckn40y7hgqphhbhyrq64h9r6rvdh973r-fileset
```

现在修改任何被排除的文件不一定会触发重新构建：

```
$ echo >> build.nix
```

```
$ nix-build
跟踪: /home/user/fileset
跟踪: - hello.txt (常规文件)
跟踪: - world.txt (常规文件)
/nix/store/ckn40y7hgqphhbhyrq64h9r6rvdh973r-fileset
```

¹²⁶ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.unions>

2.6.7 过滤器

fileFilter¹²⁷ 函数可对文件集进行筛选，使每个包含的文件均满足指定条件。

用于选取所有以.nix结尾的文件：

代码清单15：build.nix

```
sourceFiles =
  fs.difference
    ./
    (fs.unions [
      (fs.maybeMissing ./result)
      ./default.nix
      ./build.nix
      + (fs.fileFilter (文件: 文件.hasExt "nix") ./)
        ./npins
    ]);

```

即使新增.nix文件也不会改变结果

```
$ nix-build
跟踪: /home/user/fileset
跟踪: - hello.txt (常规文件)
跟踪: - world.txt (常规文件)
/nix/store/ckn40y7hgqphhhbyrq64h9r6rvdh973r-fileset
```

值得注意的是，使用差异路径`./`的方法会明确选择要排除的文件，这意味着默认会包含源目录中新添加的文件。根据项目需求，这可能比下一节的替代方案更合适。

2.6.8 联合模式（显式包含文件）

与前一种方法相反，联合模式也可用于仅选择要包含的文件。这意味着默认会忽略当前目录中新添加的文件。

创建一些附加文件：

```
$ mkdir src
$ touch build.sh src/select.{c,h}
```

随后从需要显式包含的文件中创建文件集：

清单16：build.nix

```
{ stdenv, lib }:
let
  fs = lib.filesset;
  sourceFiles = fs.unions [
    ./你好.txt
    ./世界.txt
    ./构建.sh
    (fs.fileFilter
      (文件: 文件.hasExt "c" || 文件.hasExt "h")
      ./src
    )
  ];
in
```

(下页继续)

¹²⁷ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.filesset.fileFilter>

(continued from previous page)

文件系统追踪源文件

```
标准环境构建派生函数 {
  名称 = "文件集";
  源文件 = 文件系统转源函数 {
    根目录 = ./;
    文件集 = 源文件;
  };
  postInstall = "
    cp -vr . $out
  ";
}
```

后安装脚本已简化为依赖预先过滤的源码：

```
$ nix-build
跟踪: /home/user/fileset
跟踪: - build.sh (常规文件)
跟踪: - hello.txt (常规文件)
跟踪: - src (目录内所有文件)
跟踪: - world.txt (常规文件)
将构建此派生:
/nix/store/sjzkn07d6a4qfp60p6dc64pzvmmmdafff-fileset.drv

...
'.' -> '/nix/store/zl4n1g6is4cmsqf02dc5b2h5zd0ia4r-fileset'
'./build.sh' -> '/nix/store/zl4n1g6is4cmsqf02dc5b2h5zd0ia4r-fileset/build.sh'
'./hello.txt' -> '/nix/store/zl4n1g6is4cmsqf02dc15b2h5zd0ia4r-fileset/hello.txt'
'./world.txt' -> '/nix/store/zl4n1g6is4cmsqf02dc5b2h5zd0ia4r-fileset/world.txt'
'./src' -> '/nix/store/zl4n1g6is4cmsqf02dc5b2h5zd0ia4r-fileset/src'
'./src/select.c' -> '/nix/store/zl4n1g6is4cmsqf02dc15b2h5zd0ia4r-fileset/src/
→ select.c'
'./src/select.h' -> '/nix/store/zl4n1g6is4cmsqf02dc15b2h5zd0ia4r-fileset/src/
eselect.h'

...
/nix/store/zl4n1g6is4cmsqf02dc5b2h5zd0ia4r-fileset
```

即使新增文件，也仅使用指定文件：

```
$ touch src/select.o README.md

$ nix-build
跟踪: - build.sh (常规文件)
跟踪: - hello.txt (常规文件)
跟踪: - src
跟踪: - select.c (常规文件)
跟踪: - select.h (常规文件)
跟踪: - world.txt (常规文件)
/nix/store/zl4n1g6is4cmsqf02dc5b2h5zd0ia4r-fileset
```

2.6.9 匹配Git追踪的文件

若某目录属于Git仓库，将其传递给git Tracked¹²⁸ 可生成仅包含Git追踪文件的文件集。

创建本地Git仓库，并将除src/select.o和./result外的所有文件添加至仓库：

```
§ git init
已初始化空Git仓库于/home/user/fileset/.git/
```

(接下一页)

¹²⁸ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.fileset.gitTracked>

(接上页)

```
§ git add -A
$ git reset src/select.o result
```

通过 `gitTracked` 复用此文件选择：

清单 17: build.nix

```
sourceFiles = fs.gitTracked ./;
```

重新构建：

```
$ nix-build
警告：Git仓库/home/user/fileset'存在未提交更改
跟踪：/home/vg/src/nix.dev/fileset
跟踪：- README.md (常规文件)
跟踪：- build.nix (常规文件)
跟踪：- build.sh (常规文件)
跟踪：- default.nix (常规文件)
跟踪：- hello.txt (常规文件)
跟踪：- npins (目录内所有文件)
跟踪：- src
跟踪：- select.c (常规文件)
跟踪：- select.h (常规文件)
跟踪：- world.txt (常规文件)
将构建以下派生：
/nix/store/p9aw3f15xcjbogg9yagykywvskzgrmk5y-fileset.drv
...
/nix/store/cw4bza1r27iimzrdbf14yn5xr36d6k51-fileset
```

但这包含的内容过多，因为并非所有这些文件都是最初构建该派生所需的。

注：当使用 `flakes` 和 `nix-command` 实验性功能¹²⁹时，此函数并非必需，因为 `nix build` 默认仅允许访问 Git 跟踪的文件。但为了在稳定版 Nix 中提供相同的开发者体验，仍建议使用此函数。

2.6.10 交集

这正是交集功能的用途——它能创建仅包含两个给定文件集中共有文件的新文件集。

筛选出同时被 Git 跟踪且与构建相关的所有文件：

代码清单 18: build.nix

```
sourceFiles =
  fs.交集
    (fs.git跟踪 ./)
    (fs.并集 [
      ./hello.txt
      ./world.txt
      ./build.sh
      ./src
    ]);
```

这将产生与其他方法相同的输出，因此会复用之前的构建结果：

¹²⁹ <https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake>

```
$ nix-build
警告: Git 仓库 '/home/user/fileset' 存在未提交修改
跟踪: - build.sh (常规文件)
跟踪: - hello.txt (常规文件)
跟踪: - src
跟踪: - select.c (常规文件)
跟踪: - select.h (常规文件)
跟踪: - world.txt (常规文件)
/nix/store/zl4n1g6is4cmsqf02dci5b2h5zd0ia4r-fileset
```

2.6.11 结论

我们已演示如何运用所有基础文件集函数的示例。针对更复杂的场景，可按需组合使用。

完整列表及详细说明，请参阅 lib.fileset 参考文档¹³⁰。

2.7 交叉编译

Nixpkgs 提供强大的工具来为各类系统交叉编译软件。

2.7.1 需要准备什么？

- 使用 C 编译器的经验
- Nix 语言基础知识（第13页）

2.7.2 平台

在编译代码时，我们可以区分构建平台（生成可执行文件的环境）与运行平台（运行编译后程序的环境）。¹³⁹

当这两个平台相同时称为本地编译，不同时则称为交叉编译。

当运行平台资源有限（如CPU性能不足）或难以直接进行开发时，就需要交叉编译。

经过Nix社区多年的努力，nixpkgs软件包集在交叉编译支持方面已达到世界级水平。

2.7.3 什么是目标平台？

我们定义的第三个平台概念称为目标平台。

目标平台与需要构建编译器二进制文件的情况相关。此时，您将在构建平台上构建编译器，在主机平台上运行它以编译代码，最终在目标平台上运行生成的可执行文件。

由于这种情况较为罕见，我们默认目标平台与主机平台相同。

¹³⁰ <https://nixos.org/manual/nixpkgs/stable/#sec-functions-library-fileset>

¹³⁹ 不同构建系统对交叉编译平台的术语存在差异，我们选择遵循 autoconf 的术语¹⁴⁰。

¹⁴⁰ https://www.gnu.org/software/autoconf/manual/autoconf-2.69/html_node/Hosts-and-Cross_002dCompilation.html

2.7.4 确定宿主平台配置

构建平台由Nix在配置阶段自动确定。

在宿主平台上运行以下命令可精准确定宿主平台：

```
$ $(nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 -A gnu-config)/config.
→ guess
aarch64-unknown-linux-gnu
```

若无法实现此操作（例如，当主机平台不易进行开发访问时），则需通过以下模板手动构建platformconfig：

```
<cpu> <vendor>-<os>-<abi>
```

出于历史原因，nixpkgs中采用此字符串表示形式。

注意：<vendor>常为未知，且<abi>为可选字段。平台亦无唯一标识符，例如unknown与pc可互换（因此该脚本名为config.guess）。

若无法安装Nix，请设法在主机平台可运行的操作系统上执行config.guess（通常随autoconf软件包提供）。

其他常见平台配置示例：

- aarch64-apple-darwin14
- aarch64-pc-linux-gnu
- x86_64-w64-mingw32
- aarch64-apple-ios

注：macOS/Darwin是特例，因其并非整个操作系统都开源。仅支持在aarch64-darwin与x86_64-darwin之间交叉编译。aarch64-darwin支持是近期新增的，故交叉编译功能几乎未经测试。

2.7.5 使用Nix选择宿主平台

nixpkgs 提供了一组预定义的交叉编译主机平台，称为 pkgsCross。

可以通过 nix repl 查看这些平台：

注意：从 Nix 2.19¹³¹ 开始，nix repl 需要 `-f/ --` 文件标志：

```
$ nix repl -f '<nixpkgs>' -I nixpkgs=channel:nixos-23.11
```

```
$ nix repl '<nixpkgs>' -I nixpkg=channel:nixos-23.11
欢迎使用 Nix 2.18.1。输入 :? 获取帮助。
```

正在加载'<nixpkgs>'...

已添加14200个变量。

```
piy-road phasCross (TAP)
pkgsCross.aarch64-androi
pkgsCross.aarch64-android-
phasCress parab64-darwin
天际交叉.阿迪克104-阿迪米特
```

德鲁西奥斯.伊乌斯洛
包裹交叉.穆斯尔皮

(接续下一页)

¹³¹ <https://nix.dev/manual/nix/latest/release-notes/r1-2.19>

(接上页)

| | |
|--------------------------------------|--------------------------------|
| pkgsCross.aarch64-multiplatform | pkgsCross.or1k |
| pkgsCross.aarch64-multiplatform-musl | pkgsCross.pogoplug4 |
| pkgsCross.aarch64be-embedded | pkgsCross.powerenv |
| pkgsCross.amd64-netbsd | pkgsCross.ppc-嵌入式 |
| pkgsCross.arm-嵌入式 | pkgsCross.ppc64 |
| pkgsCross.armhf-嵌入式 | pkgsCross.ppc64-musl |
| pkgsCross.armv7a-android-prebuilt | pkgsCross.prole-embedded |
| pkgsCross.armv71-hf-multiplatform | pkgsCross.raspberryPi |
| pkgsCross.avr | pkgsCross.remarkable1 |
| pkgsCross.ben-nanonote | pkgsCross.remarkable2 |
| pkgsCross.fuloongminipc | pkgsCross.riscv32 |
| pkgsCross.ghcjs | pkgsCross.riscv32-嵌入式 |
| pkgsCross.gnu32 | pkgsCross.riscv64 |
| pkgsCross.gnu64 | pkgsCross.riscv64-嵌入式 |
| pkgsCross.i686-嵌入式 | pkgsCross.scaleway-c1 |
| pkgsCross.iphone32 | pkgsCross.sheevaplug |
| pkgsCross.iphone32-simulator | pkgsCross.vc4 |
| pkgsCross.iphone64 | pkgsCross.wasi32 |
| pkgsCross.iphone64模拟器 | pkgsCross.x86_64-嵌入式 |
| pkgsCross.mingw32 | pkgsCross.x86_64-netbsd |
| pkgsCross.mingwW64 | pkgsCross.x86_64-netbsd-llvm |
| pkgsCross.xmix | pkgsCross.x86_64-unknown-redox |
| pkgsCross.msp430 | |

这些交叉编译包的属性名称在历史演进中较为随意，通常与对应平台配置字符串不匹配。

可通过 `pkgsCross.<平台>.stdenv.hostPlatform.config` 获取平台字符串：

```
nix-repl> pkgsCross.aarch64-multiplatform.stdenv.hostPlatform.config
"aarch64-unknown-linux-gnu"
```

若您所需的主机平台尚未定义，请向上游贡献该配置¹³²。

2.7.6 指定主机平台

交叉编译的配置机制如下：

1. 获取构建平台配置并应用到当前包集合（惯例称为pkgs）。

在 `pkgs = import <nixpkgs> {}` 中隐含构建平台为当前系统。这会生成一个包含构建平台所需全部依赖的编译环境 `pkgs.stdenv`。

2. 对 `pkgsCross` 中的所有包应用适当的主机平台配置。

使用 `pkgs.pkgsCross.<host>.hello` 将生成在构建平台编译、在<host>平台运行的 hello 包。

有多种等效方式可访问针对主机平台的软件包。

1. 从构建平台环境中显式选择主机平台包：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.
-11";
  pkgs = import nixpkgs {};
in
pkgs.pkgsCross.aarch64-multiplatform.hello
```

2. 在导入nixpkgs时向crossSystem传递宿主机平台参数。这将配置nixpkgs使其所有软件包均针对宿主机平台构建：

¹³² <https://github.com/NixOS/nixpkgs/blob/master/lib/systems/examples.nix>

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.
-11";
  pkgs = import nixpkgs { crossSystem = { config = "aarch64-unknown-linux-gnu";
- }; };
in
pkgs.hello
```

等效地，你可以将宿主平台作为参数传递给 nix-build：

```
$ nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 \
--arg crossSystem '{ config = "aarch64-unknown-linux-gnu"; }' \
-A hello
```

2.7.7 首次交叉编译

要交叉编译像 hello¹³³ 这样的包，选择平台属性（本例中是 aarch64-multiplatform）并运行：

```
$ nix-build '<nixpkgs>' -I nixpkgs=channel:nixos-23.11 \
-A pkgsCross.aarch64-multiplatform.hello
...
/nix/store/ldx8715rav86791qigf9xxkb7wvh2m4k-hello-aarch64-unknown-linux-gnu-2.12.1
```

注：存储路径中的软件包哈希值会随频道更新而改变。

通过¹³⁴ 属性名搜索软件包，找到您需要构建的目标。

2.7.8 实战演练：Hello World示例的跨平台编译

为展示Nix跨编译的强大功能，我们将通过静态编译方式，将自写的Hello World程序分别交叉编译至armv6l-unknown-linux-gnueabihf和x86_64-w64-mingw32（Windows）平台，并使用模拟器¹³⁵运行生成的可执行文件。

假设我们已有cross-compile.nix文件：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.11";
  pkgs = import nixpkgs {};
4
#创建一个打印Hello World的C程序
helloWorld = pkgs.writeText "hello.c" "
#include <stdio.h>
8
int main (void)
{
  printf ("你好， 世界！ \n");
  return 0;
10
}
11
12
#接收主机平台包的函数
crossCompileFor = hostPkgs:
```

(接续下一页)

133 <https://www.gnu.org/software/hello/>

134 <https://search.nixos.org/packages>

135 <https://en.wikipedia.org/wiki/Emulator>

(continued from previous page)

```

#运行一个简单的命令，使用当前可用的编译器
hostPkgs.runCommandCC "hello-world-cross-test" {} "
    #Wine需要家目录
    HOME=$PWD

22      #使用针对宿主机平台特定的编译器编译示例
$CC ${helloWorld} -o hello

25      #使用用户模式仿真运行编译后的程序（Qemu/Wine）
#传递buildPackages以确保仿真器针对构建平台编译
${hostPkgs.stdenv.hostPlatform.emulator hostPkgs.buildPackages} hello > $out

29      #输出到标准输出
cat $out
30
31      '';
in {
    #使用两个平台主机静态编译我们的示例
    rpi = 为 pkgs.pkgsCross.raspberryPi 进行交叉编译;
    windows = 为 pkgs.pkgsCross.mingwW64 进行交叉编译;
32 }
33
34
35
36
37

```

若构建此示例并打印两个派生结果，应看到各自的"Hello, world!"：

```
$ cat $(nix-build cross-compile.nix)
Hello, world!
你好，世界！
```

2.7.9 使用交叉编译器的开发环境

在声明式可重现环境教程（第9页）中，我们探讨了Nix如何为项目提供工具链和系统库支持。

还可以配置一个使用musl进行静态二进制文件交叉编译的编译器环境。

假设我们有一个shell.nix文件：

```

let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/release-23.11";
  pkgs = (import nixpkgs {}).pkgsCross.aarch64-multiplatform;
in
5
#由于 https://github.com/NixOS/nixpkgs/pull/126844 需要 callPackage
pkgs.pkgsStatic.callPackage ({ mkShell, zlib, pkg-config, file }: mkShell {
    #这些工具在构建平台上运行，但被配置为针对宿主平台..
    → 平台
    nativeBuildInputs = [ pkg-config file ];
    #宿主平台所需的库
    buildInputs = [ zlib ];
}) {}

```

以及 hello.c 文件：

```
#include <stdio.h>

int main (void)
{
    printf ("你好， 世界！ \n");
    return 0;
}
```

我们可以进行交叉编译：

```
$ nix-shell --run '$CC hello.c -o hello' shell.nix
```

并确认这是aarch64架构：

```
$ nix-shell --run 'file hello' shell.nix
hello: ELF 64位 LSB可执行文件, ARM aarch64架构, 版本1 (SYSV), 静态链接,
→ 含调试信息, 未剥离符号
```

2.7.10 后续步骤

- 官方二进制缓存¹³⁶ 仅提供有限数量的跨编译包二进制文件，为节省重新编译时间，建议配置自己的二进制缓存并使用 GitHub Actions 实现持续集成（第144页）。
 - 虽然 Nixpkgs 中许多编译器支持跨编译，但并非全部如此。
此外，支持跨编译并非易事，由于需要测试的潜在组合过多，某些软件包可能无法构建成功。
 - 关于 Nix¹³⁷ 中跨编译实现原理的详细说明，可协助解决这些问题。
 - Nix 社区设有专门的 Matrix 聊天室¹³⁸，可提供跨编译相关帮助。
-

2.8 模块系统

Nixpkgs 和 NixOS 的强大功能很大程度上源于其模块系统。

模块系统是一个 Nix 语言库，它允许您：

- 通过多个独立的 Nix 表达式声明一个属性集。
- 对该属性集中的值施加类型约束。
- 在不同 Nix 表达式中为同一属性定义值，并根据类型自动合并这些值。

这些Nix表达式称为模块，必须遵循特定结构。

本教程系列将带您了解

- 模块的定义与创建方法
- 选项的概念及声明方式
- 如何表达模块间的依赖关系

¹³⁶ <https://cache.nixos.org>

¹³⁷ <https://nixos.org/manual/nixpkgs/stable/#chap-cross>

¹³⁸ <https://matrix.to/#/#cross-compiling:nixos.org>

2.8.1 你需要准备什么？

- 熟悉数据类型和通用编程概念
- 已安装Nix（第1页）以运行示例
- 具备Nix语言的中级读写能力（第13页）

2.8.2 需要多长时间？

本教程篇幅较长，请预留至少3小时的学习时间。

基础模块

什么是模块？

- 模块是一个接收属性集合并返回属性集合的函数。
- 它可以声明选项，规定最终结果中允许包含哪些属性。
- 它可以为自身或其他模块声明的选项定义值。
- 当被模块系统评估时，它会根据声明和定义生成一个属性集合。

最简单的模块是一个接收任意属性并返回空属性集的函数：

清单19：options.nix

```
{ ... }:
```

```
{
}
```

要定义任何值，模块系统首先需要知道哪些值是允许的。这通过声明选项来实现，这些选项指定了哪些属性可以被设置并在其他地方使用。

声明选项

选项在顶层的options属性下通过lib.mkOption¹⁴¹进行声明。

清单20：options.nix

```
{ lib, ... }:
{
  options = {
    name = lib.mkOption { type = lib.types.str; };
  };
}
```

注意：lib参数由模块系统自动传递。这使得Nixpkgs库函数¹⁴²在每个模块的函数体中可用。

省略号...是必需的，因为模块系统可以向模块传递任意参数。

lib.mkOption参数中的type属性指定了选项的有效值类型。lib.types¹⁴³下提供了多种类型。

此处我们声明了一个str类型的选项名称：当定义值时，模块系统将期望接收字符串。

¹⁴¹ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.options.mkOption>

¹⁴² <https://nixos.org/manual/nixpkgs/stable/#chap-functions>

¹⁴³ <https://nixos.org/manual/nixos/stable/#sec-option-types-basic>

既然已声明选项，自然要为其赋值。

定义值

选项在顶层config属性下设置或定义：

清单21： config.nix

```
{ ... }:
{
  config = {
    name = "Boaty McBoatface";
  };
}
```

在选项声明中，我们创建了一个字符串类型的选项名称。而在选项定义中，我们将该选项同样设置为字符串类型。

选项声明与选项定义无需位于同一文件。具体哪些模块会参与最终属性集的构建，是在配置模块系统评估时指定的。

模块评估

模块通过Nixpkgs库中的lib.evalModules¹⁴⁴ 进行评估。该函数接收一个属性集作为参数，其中modules属性是要合并评估的模块列表。

evalModules的输出包含所有已评估模块的信息，最终值会出现在config属性中。

清单22： default.nix

```
让
pkgs = 导入 <nixpkgs> { };
result = pkgs.lib.evalModules {
  modules = [
    ./options.nix
    ./config.nix
  ];
};
in
result.config
```

这是一个辅助脚本，用于通过 nix-instantiate --eval¹⁴⁵ 解析并评估我们的 default.nix 文件，并将输出以 JSON 格式打印：

清单23： eval.bash

```
nix-shell -p jq --run "nix-instantiate --eval --json --strict | jq"
```

只要每个定义都有对应的声明，评估就会成功。如果存在未声明的选项定义，或定义值类型错误，模块系统将抛出错误。

运行脚本 (./eval.bash) 应显示与我们配置相符的输出：

```
{ "name": "船船长船脸"
}
```

¹⁴⁴ <https://nixos.org/manual/nixpkgs/stable/#module-system-lib-evalModules>

¹⁴⁵ <https://nix.dev/manual/nix/stable/command-ref/nix-instantiate>

模块系统深入探究

或称：用模块封装世界

本教程将通过完整演示，指导您如何用Nix模块封装现有API。

概述

本教程基于@infinisil¹⁴⁶，关于模块的演讲¹⁴⁷（来源¹⁴⁸），面向2021年Nix之夏¹⁴⁹的参与者。

配合视频学习本教程，可更清晰掌握代码修改轨迹。

你将学习什么？

你将编写模块与谷歌地图API¹⁵⁰交互，声明代表地图几何形状、位置标记等模块选项。

教程中，你将先编写一些错误配置，借此讨论产生的错误信息及解决方法，特别是在涉及类型检查时。

你需要什么？

本练习需使用两个辅助脚本。将map.sh和geocode.sh下载至工作目录。

警告：运行本教程示例需在\$XDG_DATA_HOME/google-api/key中放置谷歌API密钥¹⁵¹。

空模块

将以下内容写入名为 default.nix 的文件：

146 <https://github.com/infinisil>

147 <https://infinisil.com/modules.mp4>

148 <https://github.com/tweag/summer-of-nix-modules>

149 <https://github.com/ngi-nix/summer-of-nix>

150 <https://developers.google.com/maps/documentation/maps-static>

151 <https://developers.google.com/maps/documentation/maps-static/start#before-you-begin>

代码清单24： default.nix

```
{ ... }:
```

声明选项

我们需要一些辅助函数，这些函数来自Nixpkgs库¹⁵²，它由模块系统作为lib参数传入：

代码清单25： default.nix

```
- { ... }:
+ { lib, ... }:
{
}
```

使用lib.mkOption¹⁵³，将scripts.output选项声明为lines类型：

清单26： default.nix

```
{ lib, ... }:{

  options = {
    scripts.output = lib.mkOption {
      type = lib.types.lines;
    };
  };
}
```

lines类型表示唯一有效的值是字符串，且多个定义应当用换行符连接。

注：该选项的名称与属性路径可任意定义。此处选用"scripts"是因后续将添加另一脚本，而命名"output"是因它将输出最终生成的映射。

¹⁵² <https://github.com/NixOS/nixpkgs/tree/master/lib>

¹⁵³ <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.options.mkOption>

评估模块

新建 eval.nix 文件调用 lib.evalModules¹⁵⁴，并评估 default.nix 中的模块：

代码清单 27: eval.nix

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
pkgs.lib.evalModules {
  modules = [
    ./default.nix
  ];
}
```

运行以下命令：

警告：这将导致错误。

nix-instantiate --eval eval.nix -A config/scripts.output

详细说明

nix-instantiate --eval¹⁵⁵ 会解析并评估指定路径下的Nix文件，然后输出结果。evalModules生成一个属性集，最终配置值出现在config属性中。因此我们在eval.nix文件中评估Nix表达式时使用属性路径¹⁵⁶ config/scripts.output。

错误信息表明scripts.output选项被使用但未定义：在访问该选项前必须为其设置值。您将在后续步骤中完成此操作。

类型检查

如前所述，lines类型仅允许字符串值。

警告：本节中您将设置一个无效值并遇到类型错误。

若尝试将整数值赋给该选项会怎样？

将以下内容添加至 default.nix：

清单28： default.nix

```
{ lib, ... }: {
  options = {
    scripts.output = lib.mkOption {
      类型 = lib.types.lines;
    };
  };
  + 配置 = {
```

(下页继续)

¹⁵⁴ <https://nixos.org/manual/nixpkgs/unstable/#module-system-lib-evalModules>

¹⁵⁵ <https://nix.dev/manual/nix/stable/command-ref/nix-instantiate>

¹⁵⁶ <https://nix.dev/manual/nix/stable/language/operators#attribute-selection>

(接上页)

```
+ scripts.output = 42;
+ };
}
```

现在尝试执行上一条命令，你将看到第一个模块错误：

```
$ nix-instantiate --eval eval.nix -A config.scripts.output
错误:
...
    错误：选项 `scripts.output` 的定义不符合 `strings_` 类型
以"\n"连接的定义值：
    • 在 '/home/nix-user/default.nix' 文件中第42行:
```

定义 `scripts.output = 42;` 引发类型错误：整数不能与换行符字符串连接。

为使该模块通过类型检查并成功解析 `scripts.output` 选项，现需为其分配字符串值。

本例中，将分配一个用于执行当前目录下地图脚本的shell命令。该脚本调用Google Maps Static API生成世界地图，输出结果通过 feh¹⁵⁷（极简图像查看器）显示。

修改 `default.nix` 文件，将 `scripts.output` 的值更新为以下字符串：

清单29： default.nix

```
config = {
-   scripts.output = 42;
+   scripts.output =
+     ./map.sh size=640x640 scale=2 | feh -
+     '';
};
```

插曲：可复现的脚本

这条简单命令很可能无法在您的系统上按预期运行，因为它可能缺少必要的依赖项（curl 和 feh）。我们可以通过用 `pkgs.writeShellApplication` 打包原始地图脚本来解决这个问题。

首先，通过添加设置 `config._module.args` 的模块，确保在模块评估中提供 `pkgs` 参数：

清单30： eval.nix

```
pkgs.lib.evalModules {
  modules = [
+   ({ config, ... }: { config._module.args = { inherit pkgs; }; })
    ./default.nix
  ];
}
```

注：该机制目前仅在模块系统代码¹⁵⁸ 中有记载，且文档不完整且已过时。

随后将 `default.nix` 内容修改如下：

¹⁵⁷ <https://feh.finalrewind.org/>

¹⁵⁸ <https://github.com/NixOS/nixpkgs/blob/master/lib/modules.nix#L140-L182>

清单31： default.nix

```
{
  pkgs, lib, ... }: {
    options = {
      scripts.output = lib.mkOption {
        type = lib.types.package;
      };
    };

    config = {
      scripts.output = pkgs.writeShellApplication {
        name = "map";
        runtimeInputs = with pkgs; [ curl feh ];
        text = "
          ${./map.sh} size=640x640 scale=2 | feh -
        ";
      };
    };
  };
}
```

这将访问先前添加的 `pkgs` 参数以便使用依赖项，并将当前目录中的 `map` 文件复制到 Nix 存储中，使其对封装脚本可用（该脚本同样位于 Nix 存储中）。

运行脚本命令：

```
nix-build eval.nix -A config.scripts.output
./result/bin/map
```

如需快速迭代，请打开新终端并设置 `entr`¹⁵⁹，以便当前目录中任何源文件变更时自动重新运行脚本：

```
nix-shell -p entr findutils bash --run \
"ls *.nix |
  entr -rs '\
    nix-build eval.nix -A config.scripts.output --no-out-link \
    | xargs printf -- '\"$s/bin/map\"' \
    | xargs bash \
  '
"
```

该命令执行以下操作：

- 列出所有 `.nix` 文件
- 监控这些文件的变更。每次变更时以 `-r` 参数终止正在运行的命令。
- 每次变更时：
 - 按上述方式执行 `nix-build` 命令，但不创建 `./result` 符号链接
 - 获取生成的存储路径并追加 `/bin/map`
 - 运行以这种方式构建的路径下的可执行文件

¹⁵⁹ <https://github.com/eradman/entr>

声明更多选项

我们将通过模块系统来设置脚本参数，而非直接配置所有参数。这不仅会通过类型检查增加安全性，还能构建抽象层以管理日益增长的复杂性和需求变更。

首先引入一个新选项 `requestParams`，它将表示向 Google Maps API 发起请求时所用的参数。

其类型为 `<element Type>` 列表，即由单一类型元素构成的集合。

本例中，您需要将列表元素的类型指定为 `str`（通用字符串类型），而非 `lines`。

`str` 与 `lines` 的区别在于合并行为：模块选项类型不仅会校验有效值，还会指定多个选项定义应如何合并为一个。

- 对于行类型，多个定义将通过换行符连接合并。
- 对于字符串类型，不允许存在多个定义。此处不成问题，因为无法多次定义列表元素。

将以下内容添加到你的 `default.nix` 文件中：

代码清单 32：`default.nix`

```
scripts.output = lib.mkOption {
    type = lib.types.package;
};

+ requestParams = lib.mkOption {
+   type = lib.types.listOf lib.types.str;
+ };
};

config = {
    scripts.output = pkgs.writeShellApplication {
        name = "map";
        runtimeInputs = with pkgs; [ curl feh ];
        text = "
            ${./map.sh} 尺寸=640x640 缩放=2 | feh -
        ";
    };
};

+ 请求参数 = [
+   "尺寸=640x640"
+   "缩放=2"
+ ];
};

}
```

选项间的依赖关系

给定模块通常声明一个生成结果的选项供其他部分使用，本例中即 `scripts.output`。

选项可依赖其他选项，从而构建更有用的抽象。

此处我们希望 `scripts.output` 选项将 `requestParams` 的值作为参数传递给 `./mapscript`。

访问选项值

要使模块能获取选项值，声明模块的函数参数必须包含config属性。

更新default.nix以添加config属性：

清单33： default.nix

```
-{ pkgs, lib, ... }: {
+{ pkgs, lib, config, ... }: {
```

当评估设置选项的模块时，可通过config下对应的属性名访问最终值。

注意：选项值不能直接从同一模块中访问。

模块系统会评估接收的所有模块，其中任一模块均可定义特定选项的值。当多个模块设置同一选项时，其行为由该选项的类型决定。

警告： config参数与config属性不同：

- config参数包含模块系统延迟评估的结果，该评估会考虑所有传递给evalModules的模块及其导入项。
- 模块的config属性向模块系统暴露该模块的选项值以供评估。

现在对 default.nix 进行如下修改：

代码清单 34： default.nix

```
config = {
  scripts.output = pkgs.writeShellApplication {
    name = "map";
    runtimeInputs = with pkgs; [ curl feh ];
    text = "
      ${./map.sh} size=640x640 scale=2 | feh -
      ${./map.sh} ${lib.concatStringsSep " "
        config.requestParams} | feh -
    ";
  };
}
```

此处， config.requestParams 属性的值由模块系统根据同一文件中的定义填充。

注： Nix语言的惰性求值特性允许模块系统在传递给定义该值的模块的config参数中提供该值。

lib.concatStringsSep " " 用于将 config.requestParams 值中的每个列表元素连接成单个字符串，其中 requestParams 的列表元素以空格分隔。
该结果表示要传递给 ./map 脚本的命令行参数列表。

条件式定义

有时您会希望选项值是可选的。这在定义非必填选项值时非常实用，如下例所示。

您将定义一个新选项 `map.zoom` 来控制地图缩放级别。若未传递相应参数，Google Maps API 会自动推断缩放级别——这种情况可用 `nullOr <type>` 表示，它代表 `<type>` 类型值或 `null`。但这不意味着未定义选项时其值自动为 `null`，我们仍需设定默认值。

将带有 `zoom` 选项的 `map` 属性集添加到顶层 `options` 声明中，如下所示：

代码清单 35: default.nix

```
requestParams = lib.mkOption {
    类型 = lib.types.listOf lib.types.str;
};

+ 映射 = {
+     缩放 = lib.mkOption {
+         类型 = lib.types.nullOr lib.types.int;
+         默认值 = null;
+     };
+ };
};
```

要使用此功能，请使用 `mkIf <条件> <定义>` 函数，该函数仅在条件评估为真时添加定义。请对配置块中的 `requestParams` 列表进行以下添加：

清单36：default.nix

```
requestParams = [
    "size=640x640"
    "scale=2"
    (lib.mkIf (config.map.zoom != null)
        "zoom=${toString config.map.zoom}")
];
};
```

仅当 `config.map.zoom` 的值不为空时，才会向脚本调用添加缩放参数。

默认值

假设在我们的应用中，我们希望设置默认缩放级别为10，这样必须显式启用自动缩放功能。

这可以通过 `mkOption`¹⁶⁰ 的 `default` 参数实现。当选项未被显式赋值时，将采用该默认值。

修改对应行：

代码清单37：default.nix

```
地图 = {
    缩放 = lib.mkOption {
        类型 = lib.types.nullOr lib.types.int;
        - 默认值 = null;
        + 默认值 = 10;
    };
};
```

(下页继续)

¹⁶⁰ <https://github.com/NixOS/nixpkgs/blob/master/lib/options.nix>

(continued from previous page)

```
    };
};
```

封装 shell 命令

您已声明了控制地图尺寸和缩放级别的选项，但尚未提供指定地图中心位置的方式。

现在添加 center 选项，建议将您当前位置设为默认值：

清单 38： default.nix

```
type = lib.types.nullOr lib.types.int;
default = 10;
};

+   center = lib.mkOption {
+     type = lib.types.nullOr lib.types.str;
+     default = "switzerland";
+   };
};

};
```

为实现此功能，您将使用地理编码工具（geocode），它能将位置名称转换为坐标。虽然可通过多种方式引入新包，但作为练习，您将选择通过模块系统以选项形式添加。

首先，添加一个新选项来配置该包：

清单39： default.nix

```
options = {
  scripts.output = lib.mkOption {
    type = lib.types.package;
  };

  scripts.geocode = lib.mkOption {
    type = lib.types.package;
  };
};
```

随后在定义该选项值时，通过用writeShellApplication包裹调用使其成为可复现的原始脚本：

清单40： default.nix

```
config = {
  scripts.geocode = pkgs.writeShellApplication {
    name = "geocode";
    runtimeInputs = with pkgs; [ curl jq ];
    text = "exec ${./geocode.sh} \"${@}\"";
  };

  scripts.output = pkgs.writeShellApplication {
    name = "map";
    runtimeInputs = with pkgs; [ curl feh ];
```

现在向 requestParams 列表中添加另一个 mkIf 调用，通过 config.scripts.geocode 访问封装包，并运行其中的 /bin/geocode 可执行文件：

清单41： default.nix

```
"scale=2"
  (lib.mkIf (config.map.zoom != null)
    "zoom=${toString config.map.zoom}")
+  (lib.mkIf (config.map.center != null)
+    "center=\"$(${config.scripts.geocode}/bin/geocode ${{
+      lib.escapeShellArg config.map.center
+    }})\"")
  ];
};
```

这次你使用了`escapeShellArg`将`config.map.center`值作为命令行参数传递给`geocode`，并通过字符串插值将结果返回到设置中心值的`requestParams`字符串中。在Nix模块中封装shell命令执行是控制系统变更的有效技巧，因为它使用更符合人体工程学的属性和值接口，而非手动处理转义的特殊情况。

模块拆分

模块模式¹⁶¹ 包含`imports`属性，允许引入更多模块，例如将大型配置拆分到多个文件中。

这尤其能让你将选项声明与配置中的使用位置分离。

新建一个模块 `marker.nix`，用于声明地图位置标记和其他图钉的配置选项：

代码清单42： marker.nix

```
{ lib, config, ... }: {
```

在 `default.nix` 中通过 `imports` 属性引用该文件：

代码清单43： default.nix

```
{ pkgs, lib, config, ... }: {
```

```
+   imports = [
+     ./marker.nix
+   ];
+
```

子模块类型

我们需要在地图上设置多个标记点。每个标记点都是包含多个字段的复合类型。

模块系统类型体系中最实用的类型之一在此发挥作用：子模块。该类型允许定义具有独立配置选项的嵌套模块。

这里将定义一个新的地图标记点选项，其类型为子模块列表，每个子模块包含嵌套的坐标类型，从而支持在地图上定义标记点列表。

每个标记的赋值都将在顶层配置评估期间进行类型检查。

对 `marker.nix` 作出以下修改：

¹⁶¹ <https://nixos.org/manual/nixos/stable/#sec-writing-modules>

清单44：marker.nix

```
-{ lib, config, ... }: {
+{ lib, config, ... }:
+let
+  markerType = lib.types.submodule {
+    options = {
+      location = lib.mkOption {
+        type = lib.types.nullOr lib.types.str;
+        default = null;
+      };
+    };
+  };
+in {
+
+  options = {
+    map.markers = lib.mkOption {
+      类型 = lib.types.listOf markerType;
+    };
+  };
+}
```

在其他模块中定义选项

由于模块系统组合选项定义的方式，您可以自由地为其他模块中定义的选项赋值。

本例中，您将通过 `map.markers` 选项生成新元素并添加到 `requestParams` 列表，使声明的标记显示在返回的地图上——但这些标记实际来自 `marker.nix` 模块。

要实现此功能，请将以下配置块添加到 `marker.nix` 中：

清单45：marker.nix

```
+ config = {
+
+  map.markers = [
+    { location = "纽约"; }
+  ];
+
+  requestParams = let
+    paramForMarker =
+      builtins.map (marker: "${config/scripts/geocode}/bin/geocode ${lib.escapeShellArg marker.location}") config.map.markers;
+    in [ "markers=${lib.concatStringsSep "\|" paramForMarker}\\" ];
+  };
+}
```

警告：为避免地图选项设置与最终配置的`map`值混淆，此处显式使用`builtins.map`函数。

这里再次使用`escapeShellArg`和字符串插值生成Nix字符串，最终输出以竖线分隔的地理编码位置属性列表。

`requestParams`值也被设置为生成的字符串列表，得益于列表类型的默认合并行为，该值会追加到`default.nix`中定义的`requestParams`列表。

定义多个标记点时，手动确定合适的地图中心点或缩放级别可能较困难，不如让API自动处理。

为此，请在 `requestParams` 声明上方的 `marker.nix` 中添加以下内容：

清单46：marker.nix

```
+ map.center = lib.mkIf
+   (lib.length config.map.markers >= 1)
+   null;
+
+ map.zoom = lib.mkIf
+   (lib.length config.map.markers >= 2)
+   null;
+
+ requestParams = let
+   paramForMarker = marker:
+     let
```

在此情况下，若未传入中心点或缩放级别，Google Maps API 的默认行为是选择所有给定标记的几何中心，并设置适合一次性查看所有标记的缩放级别。

嵌套子模块

接下来，我们希望允许多个指定用户各自定义标记点列表。

为此，您需要添加一个类型为lib.types.attrsOf <subtype>的users选项，这将允许您将用户定义为属性集，其值具有<subtype>类型。

此处，该子类型将是另一个允许声明出发标记点的子模块，适用于向API查询行程推荐路线。

这将再次利用markerType子模块，形成嵌套的子模块结构。

要将标记点定义从用户传递到map.markers选项，请进行以下更改。

在let块中：

清单47：marker.nix

```
+ userType = lib.types.submodule {
+   options = {
+     departure = lib.mkOption {
+       type = markerType;
+       default = {};
+     };
+   };
+ };
+
in {
```

此处定义用户子模块类型，包含一个标记类型的离开选项。

在配置块中，map.markers上方：

代码清单48：marker.nix

```
users = lib.mkOption {
  type = lib.types.attrsOf userType;
};
```

这允许在任何导入 marker.nix 的子模块中，添加一个设置为 config 的 users 属性集，其中每个属性都将如先前步骤所声明的那样属于 userType 类型。

在 config 块中，map.center 上方：

清单49：marker.nix

```
config = {
  map.markers = [
    { location = "纽约"; }
  ];
  map.markers = lib.filter
    (marker: marker.location != null)
    (lib.concatMap (用户: [
      用户.departure
    ]) (lib.attrValues config.users));
  map.center = lib.mkIf
    (lib.length config.map.markers >= 1)
```

此操作从 config 参数中所有用户的 departure 标记中提取数据，当 location 属性非空时将其添加到 map.markers 中。

config.users 属性集被传递给 attrValues，后者返回该集合中每个属性的值列表（此处指您定义的 config.users 集合），并按字母顺序排序（这是 Nix 语言中存储属性名的方式）。

回到 default.nix 中，生成的 map.markers 选项值仍由 requestParams 访问，而 requestParams 又用于生成最终调用 Google Maps API 的脚本参数。

通过这种方式定义选项，您可以设置多个 users.<name>.departure.location 值，并生成具有适当缩放和中心位置的地图，其中图钉对应所有用户的出发位置值集合。

在 2021 年 Nix 之夏活动中，这构成了一个交互式多人地图演示的基础。

strMatching 类型

既然地图现在可以渲染多个标记，是时候添加一些样式自定义了。

为区分不同标记，需在标记类型子模块中新增选项，允许为每个标记引脚添加标签。

API 文档明确指出这些标签必须是大写字母或数字¹⁶²。

可通过 strMatching "<negex>" 类型实现，其中<regex>是接受匹配值的正则表达式，此处即大写字母或数字。

在 let 代码块中：

清单50：marker.nix

```
type = lib.types.nullOr lib.types.str;
默认值 = null;
};

+
+ style.label = lib.mkOption {
+   类型 = lib.types.nullOr
+     (lib.types.strMatching "[A-Z0-9]");
+   默认值 = null;
+ };
};
```

types.nullOr 同样允许空值，且默认值已设为 null。

在 paramForMarker 函数中：

¹⁶² <https://developers.google.com/maps/documentation/maps-static/start#MarkerStyles>

代码清单51：marker.nix

```

requestParams = let
-   paramForMarker =
-     builtins.map (marker: "${${config/scripts/geocode}/bin/geocode ${lib.escapeShellArg marker.location})}" config.map.markers;
-   in [ "markers=\\"${lib.concatStringsSep "|" paramForMarker}\\""];
+   paramForMarker = 标记:
+   let
+     attributes =
+       lib.optional (标记.style.label != null)
+       "label:${标记.style.label}"
+       ++
+       [
+         "${${config/scripts/geocode}/bin/geocode ${lib.escapeShellArg 标记位置
+         })}"
+       ];
+   在 "markers=\\"${lib.concatStringsSep "|" attributes}\\" 中
+   in
+     builtins.map 标记参数 config.map.markers

```

注意我们现在如何通过将标签和位置属性拼接起来，为每个用户创建唯一标记，并将其分配给 requestParams。仅当 marker.style.label 被设置时，每个标记的标签才会传播到 CLI 参数。

作为子模块参数的函数

目前若未显式设置标签，则不会显示任何内容。但由于每个用户属性都有名称，我们可以将其作为默认值使用。

firstUpperAlnum 函数用于获取用户名的首字符，其返回类型可直接用于 departure.style.label：

清单52：marker.nix

```

{ lib, config, ... }:
let
  #返回字符串的首字母大写形式
  #或首数字
  firstUpperAlnum = 字符串:
    lib.mapNullable lib.head
    (builtins.match "[^A-Z0-9] * ([A-Z0-9]) . * "
      (lib.toUpperCase 字符串));

  markerType = lib.types.submodule {
    options = {

```

通过将 lib.types.submodule 的参数转换为函数，您可以在其中访问其他参数。

子模块自动获得的一个特殊参数是 name，当在 attrsOf 中使用时，它会给出该子模块所定义属性名称：

代码清单53：marker.nix

```

• userType = lib.types.submodule {
+ userType = lib.types.submodule ({ name, ... }: {
  options = {
    departure = lib.mkOption {
      type = markerType;
      default = {};

```

(下页继续)

(接上页)

```

    };
};

- };

```

这种情况下，您无法轻易从标记子模块的标签选项中获取名称，而该选项本可设置默认值。

但您可以通过用户子模块的配置项来设置默认值，如下所示：

代码清单 54: marker.nix

```

+ config = {
+   departure.style.label = lib.mkDefault
+     (首字母大写的字母数字组合 name);
+   );
+ });

in

```

注意：模块选项具有优先级（以整数表示），该优先级决定了将选项设置为特定值时的优先顺序。合并值时，数值最小的优先级胜出。

`lib.mkDefault` 修饰符将其参数值的优先级设为1000，即最低优先级。

这确保了为同一选项设置的其他值将优先生效。

either 与 enum 类型

为提升视觉对比度，若能更改标记颜色将很有帮助。

此处你将使用两种新的类型函数来实现：

- `either <this> <that>`，接受两个类型参数，允许其中任意一个
- `enum [<allowed values>]`，接受允许值列表，允许其中任意一个

在`let`块中，添加以下`colorType`选项，该选项可存储包含给定颜色名或RGB值的字符串，并添加新的复合类型：

代码清单55：marker.nix

```

... (builtins.match "[^A-Z0-9]*([A-Z0-9]).*"
(lib.toUpperCase str));

#颜色名称或`0xRRGGBB`格式
colorType = lib.types.either
  (lib.types.strMatching "0x[0-9A-F]{6}")
  (lib.types.enum [
    "black" "brown" "green" "purple" "yellow"
    "蓝色" "灰色" "橙色" "红色" "白色" ]);

markerType = lib.types.submodule {
  options = {
    location = lib.mkOption {

```

允许使用符合24位十六进制数的字符串或等于指定颜色名称之一的字符串。

在`let`块底部添加`style.color`选项并指定默认值：

清单56： marker.nix

```
(lib.types.strMatching "[A-Z0-9]");
默认值 = null;
};

+ style.color = lib.mkOption {
+   类型 = colorType;
+   默认值 = "red";
+ };
};

};
```

现在向paramForMarker列表添加一个条目，以利用新选项：

代码清单57： marker.nix

```
(marker.style.label != null)
"label:${marker.style.label}"
++ [
  "color:${marker.style.color}"
  "${${config/scripts/geocode}/bin/geocode ${lib.escapeShellArg 标记位置
})}"
```

若需设置多个不同标记，支持单独调整其尺寸将非常实用。

为 marker.nix 新增样式尺寸选项，可从预定义尺寸集中选择：

代码清单58： marker.nix

```
type = 颜色类型;
default = "红色";
};

+ style.size = lib.mkOption {
+   type = lib.types.enum
+   [ "tiny" "small" "medium" "large" ];
+   default = "medium";
+ };
};

};
```

现在在paramForMarker中添加尺寸参数的映射，该映射会选择合适的字符串传递给API：

清单59： marker.nix

```
请求参数 = let
  标记参数 = marker:
    let
      尺寸 = {
        极小 = "tiny";
        小 = "small";
        medium = "中";
        large = null;
      }.${marker.style.size};
    
```

最后，在属性字符串中再添加一个 lib.optional 调用，利用所选尺寸：

清单60：marker.nix

```

属性 =
  lib.optional
    (marker.style.label != null)
    "label:${marker.style.label}"
++ lib.optional
(大小不为空)
"尺寸:${size}"
++ [
  "颜色:${marker.style.color}"
  "${${config.scripts.geocode}/bin/geocode ${{

```

路径类型模块

至此，您已创建了用于声明出发标记的选项，以及多个用于配置标记视觉呈现的选项。

现在我们需要计算并显示从用户当前位置到目的地的路线。

下一节定义的新选项将允许您设置到达标记，结合出发标记即可使用下文定义的新模块在地图上绘制路径。

首先，创建包含以下内容的path.nix新文件：

代码清单61：path.nix

```

{ lib, config, ... }:
let
  pathType = lib.types.submodule {
    options = {
      locations = lib.mkOption {
        type = lib.types.listOf lib.types.str;
      };
    };
  };
in
{
  options = {
    map.paths = lib.mkOption {
      类型 = 库.类型.列表类型 路径类型;
    };
  };
  config = {
    请求参数 =
    让
      attrForLocation = 定位:
        "${${config.scripts.geocode}/bin/geocode ${lib.escapeShellArg loc}}";
    paramForPath = 路径:
      let
        attributes =
          builtins.map attrForLocation 路径.locations;
      in
        "路径=S{lib.concatStringsSep "|" attributes}";
      in
        builtins.map paramForPath config.map.paths;
  };
}

```

path.nix 模块声明了一个选项，用于定义地图上的路径列表，其中每条路径都是由表示地理位置字符串组成的列表。

在配置属性中，我们通过将经适当转换的坐标设置为requestParams选项值来增强API调用，该值将与其它地方设置的请求参数拼接。

现在从您的marker.nix模块导入这个新的path.nix模块：

代码清单62：marker.nix

```
in {
+   imports = [
+     ./path.nix
+   ];
+
+   options = {
+     users = lib.mkOption {
```

将出发选项声明复制到marker.nix中的新到达选项，以完成初始路径实现：

清单63：marker.nix

```
      type = markerType;
      default = {};
    };
+
+   arrival = lib.mkOption {
+     type = markerType;
+     default = {};
+   };
}
```

接着在配置块中添加 arrival.style.label 属性，与 departure.style.label 保持对称：

代码清单64：marker.nix

```
config = {
  departure.style.label = lib.mkDefault
    (firstUpperAlnum name);
+
+   arrival.style.label = lib.mkDefault
+     (firstUpperAlnum name);
};
});
```

最后，更新传递给concatMap的函数中的返回列表，使其同时包含每个用户的arrivalmarker：

清单65：marker.nix

```
地图标记 = 库过滤
  (标记项: 标记项.位置 != 空)
  (库.concatMap (用户: [
    用户.出发地
    用户.出发地 用户.到达地
  ]) (库.attrValues 配置.用户));
map.center = lib.mkIf
```

现在您已具备基础，可以在地图上定义连接出发点和到达点的路径。

在路径模块中，定义连接每位用户出发地与目的地的路径：

清单66： path.nix

```
config = {
  map.paths = builtins.map(user: {
    locations = [
      user.departure.location
      user.arrival.location
    ];
  }) (库.筛选(用户:
    用户.出发地.位置 != 空值
    && 用户.到达地.位置 != 空值
  ) (库.属性值 配置.用户));
  请求参数 = 令
  位置属性 = 定位:
  "${geocode ${lib.escapeShellArg loc}}";
}
```

新地图的paths属性包含为所有用户定义的有效路径列表。
仅当为用户设置了departure和arrival属性时，路径才有效。

整数值的between约束

您的用户已发声，他们要求能够通过weight选项自定义路径样式。
和之前一样，您现在需要为路径样式声明一个新的子模块。
虽然您也可以直接声明 style.weight 选项，但在此情况下应使用子模块以便后续复用路径样式类型。
将 pathStyleType 子模块选项添加到 path.nix 的 let 块中：

代码清单 67： path.nix

```
{ lib, config, ... }:
let
+
+  pathStyleType = lib.types.submodule {
+    选项 = {
+      权重 = lib.mkOption {
+        类型 = lib.types.ints.between 120;
+        默认值 = 5;
+      };
+    };
+  };
+
+  路径类型 = lib.types.submodule {
```

注： ints.between <下限> <上限> 类型允许指定闭区间范围内的整数。

路径权重默认值为5，但可设置为1至20范围内的任意整数值，权重越高，地图上显示的路径越粗。

现在向文件下方选项集中添加一个样式选项：

代码清单68： path.nix

```
options = {
  locations = lib.mkOption {
```

(下页续)

(接上页)

```

        type = lib.types.listOf lib.types.str;
    );
+
+    style = lib.mkOption {
+        type = pathStyleType;
+        default = {};
+    };
};

};

```

最后，更新 paramForPath 中的属性列表：

清单69：path.nix

```

paramForPath = 路径:
let
    attributes =
        builtins.map 位置属性 path.locations;
        [
            "weight:${toString path.style.weight}"
        ]
        ++ 内置函数.map 属性定位 path.locations;
in "path=\"$${lib.concatStringsSep "|" attributes}\"";

```

路径样式子模块

用户目前仍无法实际自定义路径样式。为每位用户新增一个pathStyle选项。模块系统允许多次声明选项值，若类型允许，会自动合并每次声明的值。

这使得可以在marker.nix模块中定义users选项，同时在path.nix中定义users：

清单70：path.nix

```

位于 {
    options = {
+
+        users = lib.mkOption {
+            type = lib.types.attrsOf (lib.types.submodule {
+                options.pathStyle = lib.mkOption {
+                    类型 = 路径样式类型;
+                    默认值 = {};
+                };
+            });
+        );
+
+        映射.路径 = 库.可选项 {
+            类型 = 库.类型.路径类型列表;
+        );
    };
}

```

然后在映射.路径中使用用户.路径样式选项添加一行，用于处理每个用户的路径：

清单71：路径.nix

```

用户出发地
用户到达地
];

```

(接续下一页)

(continued from previous page)

```
+ 样式 = 用户.路径样式;
}) (库.过滤(用户:
    用户.出发地.位置 != 空
    && 用户.到达地.位置 != 空
```

路径样式：颜色

与标记点类似，路径应支持颜色自定义。

通过目前已掌握的类型即可实现此功能。

在path.nix中添加新的colorType块，指定允许的颜色名称及RGB/RGBA十六进制值：

代码清单72：path.nix

```
{ lib, config, ... }:
let

#颜色名称、`0xRRGGBB`或`0xRRGGBBAA`
colorType = lib.types.either
  (lib.types.strMatching "0x[0-9A-F]{6}([0-9A-F]{2})?")
  (lib.types.enum [
    "黑色" "棕色" "绿色" "紫色" "黄色"
    "蓝色" "灰色" "橙色" "红色" "白色"
  ]);
pathStyleType = lib.types.submodule {
```

在权重选项下，新增颜色选项以使用新的colorType值：

清单73：path.nix

```
type = lib.types.ints.between 120;
default = 5;
};

+ color = lib.mkOption {
+   type = colorType;
+   default = "blue";
+ };
};

};
```

最后，在属性列表中添加一行使用颜色选项：

清单74： path.nix

```
属性 =
[
  "weight:${toString path.style.weight}"
+   "color:${path.style.color}"
]
++ map attrForLocation path.locations;
in "path=${
```

进一步美化

既然已进展至此，为提升地图渲染的美观度，可新增样式选项：允许路径以大地线形式绘制，即地球上两点间最短的“直线距离”。

由于此功能可开关，建议使用布尔类型（bool）实现，其值可为 true 或 false。

现在对 path.nix 进行如下修改：

代码清单 75： path.nix

```
type = colorType;
default = "blue";
};

+
geodesic = lib.mkOption {
  type = lib.types.bool;
  default = false;
};
};

};
```

确保同时在属性列表中添加一行以使用该值，这样选项值就会包含在 API 调用中：

代码清单 76： path.nix

```
[ "粗细:${toString path.style.weight}"
  "颜色:${path.style.color}"
+  "测地线:${lib.boolToString path.style.geodesic}"
]
++ 地图 位置属性 path.locations;
位于 "路径=${
```

收尾工作

本教程中，您已学习如何借助 Nixpkgs 库提供的多个新工具函数编写自定义 Nix 模块，从而将外部服务纳入声明式控制。

您在多个文件中定义了若干模块，每个模块利用类型检查子系统实现独立子模块功能。

这些模块以声明式方式对外部 API 功能进行了封装。

现在您可以用 Nix 征服世界了。

2.9 NixOS

学习如何配置、测试及安装或部署NixOS系统。

2.9.1 创建NixOS镜像

- NixOS虚拟机（第98页）
- 构建可启动的ISO镜像（第103页）
- 构建与运行Docker镜像（第104页）

2.9.2 测试与部署 NixOS 配置

- 使用 NixOS 虚拟机进行集成测试（第107页）
- 通过 SSH 远程配置机器（第112页）
- 在树莓派上安装 NixOS（第117页）
- 使用 Terraform 部署 NixOS（第121页）

2.9.3 扩展应用

- 设置HTTP二进制缓存（第124页）
- 设置分布式构建（第128页）

NixOS虚拟机

NixOS最重要的特性之一就是能够以声明式配置整个系统，包括要安装的软件包、要运行的服务以及其他设置和选项。

NixOS配置可用于通过虚拟机测试和使用NixOS，独立于"裸机"计算机上的安装。

你将学到什么？

本教程是创建NixOS虚拟机的入门指南。虚拟机是用于实验或调试NixOS配置的实用工具。

你需要什么？

- 支持虚拟化的Linux系统
- （可选）用于运行图形化虚拟机的图形环境
- 可正常运行的Nix安装¹⁶³
- Nix语言基础知识（第13页）

重要提示：NixOS配置是一个遵循NixOS模块¹⁶⁴约定的Nix语言函数。要深入了解模块系统，请参阅《模块系统深度解析》（第76页）教程。

¹⁶³ <https://nix.dev/install-nix>

¹⁶⁴ <https://nixos.org/manual/nixos/stable/index.html#sec-writing-modules>

从默认的NixOS配置开始

注：本教程将从零开始构建您的configuration.nix文件，逐步解释每个步骤。若您希望直接参考，可跳转至示例配置章节（第99页）。

我们从一个最简化的configuration.nix开始：

```

1 { config, pkgs, ... }:
2
3 {
4     boot.loader.systemd-boot.enable = true;
5     boot.loader.efi.canTouchEfiVariables = true;
6
7     system.stateVersion = "24.05";
8 }
```

要允许登录，请将以下内容添加到返回的属性集中：

```

1 users.users.alice = {
2     isNormalUser = true;
3     extraGroups = [ "wheel" ];
4 };
```

此外，您需要为该用户指定密码。仅出于演示目的，您可以通过在用户配置中添加 initialPassword 选项来指定不安全的明文密码：

```
initialPassword = "test";
```

我们以两个轻量级程序为例：

```

1 environment.systemPackages = with pkgs; [
2     cowsay
3     lolcat
4 ];
```

警告：除非你清楚操作风险，否则请勿在本示例之外使用明文密码。更安全的替代方案请参阅initialHashedPassword¹⁶⁵ 或ssh.authorizedKeys¹⁶⁶。

示例配置

完整的 configuration.nix 文件如下所示：

```

1 { config, pkgs, ... }:
2
3 {
4     boot.loader.systemd-boot.enable = true;
5     boot.loader.efi.canTouchEfiVariables = true;
6
7     users.users.alice = {
8         isNormalUser = true;
9         extraGroups = [ "wheel" ]; # 为用户启用 'sudo' 权限
10        initialPassword = "test";
11    };
12
13    environment.systemPackages = with pkgs; [
```

(接下一页)

¹⁶⁵ https://nixos.org/manual/nixos/stable/options.html#opt-users.extraUsers._name_.initialHashedPassword

¹⁶⁶ https://nixos.org/manual/nixos/stable/options.html#opt-users.extraUsers._name_.openssh.authorizedKeys.keys

(continued from previous page)

```

15   奶牛说
16   彩虹猫
    ];
17
18   system.stateVersion = "24.05";
}

```

基于NixOS配置创建QEMU虚拟机

通过nix-build命令创建NixOS虚拟机：

```

$ nix-build '<nixpkgs/nixos>' -A vm -I nixpkg=channel:nixos-24.05 -I nixos-
  ↳ config- ./configuration.nix

```

该命令基于NixOS 24.05版本的nixos构建vm属性，使用相对路径中指定的NixOS配置文件。

详细说明

- nix-build的位置参数¹⁶⁷是要构建的derivation路径。该路径可通过求值为derivation的Nix表达式获取（第37页）。
 虚拟机构建助手定义于nixpkgs仓库¹⁶⁸中的NixOS模块，因此我们使用查找路径<nixpkgs/nixos>（第26页）。
- -A选项¹⁶⁹指定从提供的Nix表达式<nixpkgs/nixos>中选取的属性。
 为构建虚拟机，我们选用定义于nixos/default.nix中的vm属性¹⁷⁰。
- -I选项¹⁷¹会将条目添加到搜索路径前端。
 此处我们将nixpkgs指向特定版本的Nixpkgs（第158页），并将nix-config设置为当前目录下的configuration.nix文件。

运行虚拟机

上一条命令在工作目录中创建了名为result的链接，该链接指向包含虚拟机的目录。

```

$ ls -R ./result
结果:
二进制系统

结果/二进制:
运行-nixos-虚拟机

```

启动虚拟机：

```

$ QEMU_KERREL_PARAMS=console=ttyS0 ./result/bin/run-nixos-vm -nographic; reset

```

由于使用了-nographic参数，该命令将在当前终端运行QEMU。console=ttyS0还会显示启动过程，最终停留在控制台登录界面。

使用密码test以alice身份登录。确认程序是否按指定要求可用：

¹⁶⁷ <https://nix.dev/manual/nix/stable/command-ref/nix-build.html>

¹⁶⁸ <https://github.com/NixOS/nixpkgs>

¹⁶⁹ <https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-attr>

¹⁷⁰ <https://github.com/NixOS/nixpkgs/blob/7c164f4bea71d74d98780ab7be4f9105630a2eba/nixos/default.nix#L19>

¹⁷¹ <https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-I>

```
§ cowsay hello | lolcat
```

通过关机退出虚拟机：

```
$ sudo poweroff
```

注意：若忘记将用户加入wheel组或未设置密码，请从其他终端停止虚拟机：

```
§ sudo pkill qemu
```

运行虚拟机将在当前目录生成nixos.qcow2文件。该磁盘映像文件包含虚拟机的动态状态，由于会保留之前运行的状态（例如用户密码），可能干扰调试。

更改配置时请删除此文件：

```
§ rm nixos.qcow2
```

在图形化虚拟机上运行GNOME

如需创建带图形用户界面的虚拟机，请将以下内容添加至配置：

```
#启用X11窗口系统
services.xserver.enable = true;

#启用GNOME桌面环境。
services.xserver.displayManager.gdm.enable = true;
services.xserver.desktopManager.gnome.enable = true;
```

这三行代码分别激活X11系统、GDM显示管理器（用于登录）以及Gnome桌面管理器。

提示：您也可使用installation-cd-graphical-gnome.nix模块从头生成配置文件：

```
nix-shell -I nixpkgs=channel:nixos-24.05 -p "$(cat <<EOF
让
  pkgs = import <nixpkgs> { config = {}; overlays = []; };
  iso-config = pkgs.path + /nixos/modules/installer/cd-dvd/installation-cd-
→ graphical-gnome.nix;
  nixos = pkgs.nixos iso-config;
  in nixos.config.system.build.nixos-generate-config
EOF
)"
```

```
$ nixos-generate-config --dir ./
```

完整的 configuration.nix 文件如下所示：

```
{ config, pkgs, ... }:
{
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;
  services.xserver.enable = true;
```

(接下一页)

(continued from previous page)

```

7   services.xserver.displayManager.gdm.enable = true;
10  services.xserver.desktopManager.gnome.enable = true;

11  users.users.alice = {
12      isNormalUser = true;
13      extraGroups = [ "wheel" ];
14      initialPassword = "test";
15  };
16
17  system.stateVersion = "24.05";
18 }
```

要获得图形输出，请在不使用特殊选项的情况下运行虚拟机：

```
$ nix-build '<nixpkgs/nixos>' -A vm -I nixpkgs=channel:nixos-24.05 -I nixos-
→ config = ./configuration.nix
§ ./result/bin/run-nixos-vm
```

在虚拟机上以Wayland合成器身份运行Sway

要切换至Wayland合成器，需禁用`services.xserver.desktopManager.gnome`服务并启用`programs.sway`：

代码清单77：configuration.nix

```
• services.xserver.desktopManager.gnome.enable = true;
+ programs.sway.enable = true;
```

注意：在虚拟机中运行Wayland合成器可能导致QEMU所用显示驱动不兼容。需从可用驱动中选择与Sway兼容的选项。详见QEMU用户文档¹⁷²。例如virtio-vga驱动：

```
$ ./result/bin/run-nixos-vm -device virtio-vga
```

QEMU 的参数也可添加到配置文件中：

```

{ config, pkgs, ... }:
{
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  services.xserver.enable = true;

  services.xserver.displayManager.gdm.enable = true;
  programs.sway.enable = true;

  imports = [ <nixpkgs/nixos/modules/virtualisation/qemu-vm.nix> ];
  virtualisation.qemu.options = [
    "-device virtio-vga"
  ];

  users.users.alice = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
    initialPassword = "test";
  };
}
```

(接续下一页)

¹⁷² <https://www.qemu.org/docs/master/system/qemu-manpage.html>

(接上页)

```

20   };
21
22     system.stateVersion = "24.05";
23 }
```

NixOS手册包含关于X11¹⁷³与Wayland¹⁷⁴的章节，列举了备选窗口管理器。

参考文献

- NixOS手册：NixOS配置¹⁷⁵
- NixOS手册：模块¹⁷⁶
- NixOS 手册选项参考¹⁷⁷。
- NixOS 手册：修改配置¹⁷⁸。
- NixOS 源代码：tools.ni 中的配置模板¹⁷⁹。
- NixOS 源代码：default.nix 中的 vm 属性¹⁸⁰。
- Nix 手册：nix-build 命令¹⁸¹。
- Nix 手册：常用命令行选项¹⁸²。
- QEMU 用户文档¹⁸³查看更多运行时选项
- NixOS 选项搜索：virtualisation.qemu¹⁸⁴用于声明式虚拟机配置

后续步骤

- 模块系统深入解析（第76页）
- 使用 NixOS 虚拟机进行集成测试（第107页）
- 构建可启动的 ISO 镜像（第103页）

构建可启动的ISO镜像

注意：如需为其他平台构建镜像，请参阅交叉编译¹⁸⁵。

您可能会发现官方安装镜像缺少某些硬件支持。

解决方案是创建myimage.nix文件，通过最小安装ISO指向最新内核：

¹⁷³ <https://nixos.org/manual/nixos/stable/#sec-x11>
¹⁷⁴ <https://nixos.org/manual/nixos/stable/#sec-wayland>
¹⁷⁵ <https://nixos.org/manual/nixos/stable/index.html#ch-configuration>
¹⁷⁶ <https://nixos.org/manual/nixos/stable/index.html#sec-writing-modules>
¹⁷⁷ <https://nixos.org/manual/nixos/stable/options.html>
¹⁷⁸ <https://nixos.org/manual/nixos/stable/#sec-changing-config>
¹⁷⁹ <https://github.com/NixOS/nixpkgs/blob/4e0525a8cdb370d31c1e1ba2641ad2a91fded57d/nixos/modules/installer/tools/tools.nix#L122-L226>
¹⁸⁰ <https://github.com/NixOS/nixpkgs/blob/master/nixos/default.nix>
¹⁸¹ <https://nix.dev/manual/nix/stable/command-ref/nix-build.html>
¹⁸² <https://nix.dev/manual/nix/stable/command-ref/opt-common.html>
¹⁸³ <https://www.qemu.org/docs/master/system/qemu-manpage.html>
¹⁸⁴ <https://search.nixos.org/options?query=virtualisation.qemu>
¹⁸⁵ <https://github.com/nix-community/nixos-generators#user-content-cross-compiling>

```

4 { pkgs, modulesPath, lib, ... }: {
5   imports = [
6     "${modulesPath}/installer/cd-dvd/installation-cd-minimal.nix"
7   ];
8
9   # 使用最新的 Linux 内核
10  boot.kernelPackages = pkgs.linuxPackages_latest;
11
12  # 解决 https://github.com/NixOS/nixpkgs/issues/58959 所需
13  boot.supportedFilesystems = lib.mkForce [ "btrfs" "reiserfs" "vfat" "f2fs" "xfs"
14    • "ntfs" "cifs" ];
15
16 }

```

生成包含上述配置的ISO镜像：

```

§ NIX_PATH=nixpkgs=https://github.com/NixOS/nixpkgs/archive/
-74e2faf5965a12e8fa5cff799b1b19c6cd26b0e3.tar.gz nix-shell -p nixos-generators --
<运行 "nixos-generate --format iso --configuration ./myimage.nix -o result"

```

将新镜像复制到您的U盘，请将sdX替换为您的设备名称：

```

§ dd if=result/iso/*.iso of=/dev/sdX status=progress
§ sync

```

后续步骤

- 查看生成器支持的格式列表¹⁸⁶，找到您的云服务商或虚拟化技术。
- 参阅创建NixOS live的替代指南 CD¹⁸⁷

构建与运行Docker镜像

Docker¹⁸⁸是一套用于构建、管理和部署容器的工具与服务。

由于许多云平台提供基于Docker的容器托管服务，为指定服务创建Docker容器成为构建可复现软件时的常见任务。本教程将指导您如何使用Nix构建Docker容器。

先决条件

您需同时安装Nix和Docker¹⁸⁹。nixpkgs中提供了Docker，这是在NixOS上安装的首选方式。但若您使用其他Linux发行版或macOS，也可采用系统原生的Docker安装方式。

¹⁸⁶ <https://github.com/nix-community/nixos-generators#user-content-supported-formats>

¹⁸⁷ https://wiki.nixos.org/wiki/Creating_a_NixOS_live_CD

¹⁸⁸ <https://www.docker.com/>

¹⁸⁹ <https://docs.docker.com/get-docker/>

构建你的第一个容器

Nixpkgs¹⁹⁰ 提供 dockerTools 来创建 Docker 镜像：

```

4 { pkgs ? import <nixpkgs> { }
, pkgsLinux ? import <nixpkgs> { system = "x86_64-linux"; }
}:
9
5   pkgs.dockerTools.buildImage {
6     name = "hello-docker";
7     config = {
8       Cmd = [ "${pkgsLinux.hello}/bin/hello" ];
9     };
}

```

注意：若您运行的是macOS或非x86_64-linux平台，需选择以下方案：

- 配置远程构建机器（第128页）在Linux上编译
- 通过将pkgsLinux.hello替换为pkgs.pkgsCross.mus164.hello实现Linux交叉编译（第68页）

我们调用dockerTools.buildImage并传入若干参数：

- 为我们的镜像命名
- 配置包含命令 Cmd，该命令会在镜像启动后于容器内执行。此处我们引用 nixpkgs 中的 GNU hello 包，并在容器中运行其可执行文件。

将其保存为 hello-docker.nix 并构建：

```

$ nix-build hello-docker.nix
以下派生将被构建：
/nix/store/qpgdp0qpd8ddi11d72w02zkmm7n87b92-docker-layer-hello-docker.drv
/nix/store/m4xyfyviwb38sfplq3xx54j6k7mccfb-runtime-deps.drv
/nix/store/v0bvy9qxa79izc7s03fhpq5nqs2h4sr5-docker-image-hello-docker.tar.gz.drv
警告：未知设置项 'experimental-features'
正在构建 '/nix/store/qpgdp0qpd8ddi11d72w02zkmm7n87b92-docker-layer-hello-docker.drv
'...
无内容可添加至该层
正在打包层数据...
正在计算层校验和...
已完成构建层 'hello-docker'
正在构建 '/nix/store/m4xyfyviwb38sfplq3xx54j6k7mccfb-runtime-deps.drv'...
正在构建 '/nix/store/v0bvy9qxa79izc7s03fhpq5nqs2h4sr5-docker-image-hello-docker.
tar.gz.drv'...
正在添加层...
tar: 正在移除成员名称前的反斜杠`/
正在添加元数据...
正在构建镜像...
已完成。
/nix/store/y74sb4nrhx975xs7h83izgm8z75x5fc-docker-image-hello-docker.tar.gz

```

镜像标签(y74sb4nrhx975xs7h83izgm8z75x5fc)对应Nix构建哈希值，确保Docker镜像与Nix构建一致。输出最后行的存储路径指向该Docker镜像。

¹⁹⁰ <https://github.com/NixOS/nixpkgs>

运行容器

要操作容器，请通过 nix-build 创建的默认结果符号链接将此镜像加载到 Docker 的镜像仓库：

```
§ docker load < result  
已加载镜像：hello-docker:y74sb4nrhx975xs7h83izgm8z75x5fc
```

您也可以使用存储路径加载镜像，以避免依赖 result 文件的存在：

```
$ docker load < /nix/store/y74sb4nrhx975xs7h83izgm8z75x5fc-docker-image-hello-docker.tar.gz  
已加载镜像：hello-docker:y74sb4nrhx975xs7h83izgm8z75x5fc
```

更便捷的是，您可以用一条命令完成所有操作。这种方式的优势在于：若有任何改动，nix-build 将重新构建镜像，并将新的存储路径传递给 docker load：

```
§ docker load < $(nix-build hello-docker.nix)  
已加载镜像：hello-docker:y74sb4nrhx975xs7h83izgm8z75x5fc
```

现在您已将镜像加载到 Docker 中，可以运行它：

```
§ docker run -t hello-docker:y74sb4nrhx975xs7h83izgm8z75x5fc  
你好，世界！
```

使用 Docker 镜像

本教程不涉及 Docker 镜像的通用介绍。官方 Docker 文档¹⁹¹ 是更合适的学习资源。

请注意，当使用 Nix 构建 Docker 镜像时，您可能不需要编写 Dockerfile，因为 Nix 替代了 Docker 生态中的 Dockerfile 功能。但理解 Dockerfile 的构成仍有助了解 Nix 如何逐一替代其功能。另一方面，根据具体场景，使用 Docker CLI、Docker Compose、Docker Swarm 或 Docker Hub 可能仍然适用。

后续步骤

- 更多关于如何使用 dockerTools 的细节可查阅参考文档¹⁹²。
- 您可能想浏览更多用 Nix¹⁹³ 构建的 Docker 镜像示例。
- 不妨看看 Arion¹⁹⁴，这是一个对 Nix 提供一流支持的 docker-compose 封装工具。
- 使用 GitHub Actions 在 CI 上构建 docker 镜像（第144页）。

¹⁹¹ <https://docs.docker.com/>

¹⁹² <https://nixos.org/nixpkgs/manual/#sec-pkgs-dockerTools>

¹⁹³ <https://github.com/NixOS/nixpkgs/blob/master/pkgs/build-support/docker/examples.nix>

¹⁹⁴ <https://docs.hercules-ci.com/arion/>

使用NixOS虚拟机进行集成测试

你将学到什么？

本教程介绍用于测试NixOS配置的Nixpkgs功能，并展示如何设置涉及多台机器的分布式测试场景。

需要准备什么？

- Linux系统上可用的Nix安装（第1页）或NixOS¹⁹⁵
- Nix语言基础知识（第13页）
- NixOS 配置基础知识（第98页）

简介

Nixpkgs 提供了一套测试环境¹⁹⁶，用于自动化分布式系统的集成测试。它允许基于一组声明式 NixOS 配置定义测试，并通过 QEMU¹⁹⁷ 作为后端使用 Python shell 与这些配置交互。这些测试被广泛用于确保 NixOS 按预期工作，因此通常被称为 NixOS 测试¹⁹⁸。它们可以在任何 Linux 机器²⁰⁸ 上编写和运行，无需依赖 NixOS 系统。得益于 Nix 的设计特性，集成测试具备可复现性，这使其成为持续集成（CI）流程中的重要组成部分。

testers.runNixOSTest 函数

NixOS 虚拟机测试通过 testers.runNixOSTest 函数定义，其基本模式如下：

```

let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
5
pkgs.testers.runNixOSTest {
  name = "test-name";
  nodes = {
    machine1 = { config, pkgs, ... }: {
      #...
    };
    machine2 = { config, pkgs, ... }: {
      #...
    };
  };
  testScript = { nodes, ... }: "
    #...
  ";
}
19

```

¹⁹⁵ <https://nixos.org/manual/nixos/stable/index.html#sec-installation>

¹⁹⁶ <https://nixos.org/manual/nixos/stable/index.html#sec-nixos-tests>

¹⁹⁷ <https://www.qemu.org/>

¹⁹⁸ <https://nixos.org/manual/nixos/stable/index.html#sec-nixos-tests>

²⁰⁸ 在 macOS 上运行 NixOS 虚拟机测试的支持^{Page 107,209}也已实现，但目前未记录文档²¹⁰。

<https://github.com/NixOS/nixpkgs/issues/108984>

¹⁰ <https://github.com/NixOS/nixpkgs/issues/254552>

函数`testers.runNixOSTest`接受一个模块¹⁹⁹ 来指定测试选项²⁰⁰。由于该模块仅用于设置配置值，可采用简化的模块语法。

必须设置以下配置值：

- `name`²⁰¹ 定义测试名称。
- `nodes`²⁰² 包含一组命名配置（因测试脚本可能涉及多台虚拟机），每台虚拟机均基于NixOS配置创建。
- `testScript`²⁰³ 定义Python测试脚本（可为字面字符串或接收`nodes`属性的函数）。该脚本可通过节点名称访问虚拟机，并拥有虚拟机超级用户权限。脚本中每个虚拟机可通过`machine`对象访问，NixOS提供多种方法²⁰⁴ 对这些配置进行测试。

测试框架会自动启动虚拟机并执行Python脚本。

最小示例

作为对默认配置的最小化测试，我们将检查用户`root`和`alice`是否能运行Firefox。我们将从零开始逐步构建这个示例。

1. 使用固定版本的`Nixpkgs`（第158页），并显式设置配置选项和覆盖层（第148页），以避免它们被全局配置意外覆盖：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11
  ->`;
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
5
pkgs testers.runNixOSTest {
  #...
}
```

2. 为测试标注描述性名称：

```
name = "minimal-test";
```

3. 由于本示例仅使用一台虚拟机，我们指定的节点简称为`machine`。该名称可任意选择。配置时采用默认配置的相关部分（我们在前文教程第98页已使用过）：

```
nodes.machine = { config, pkgs, ... }: {
  users.users.alice = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
    packages = with pkgs; [
      firefox
      tree
    ];
  };
  system.stateVersion = "23.11";
};
```

4. 以下是测试脚本：

<https://nixos.org/manual/nixos/stable/#sec-writing-modules>
<https://nixos.org/manual/nixos/stable/index.html#sec-test-options-reference>
<https://nixos.org/manual/nixos/stable/index.html#test-opt-name>
²⁰² <https://nixos.org/manual/nixos/stable/index.html#test-opt-nodes>
³ <https://nixos.org/manual/nixos/stable/index.html#test-opt-testScript>
⁰⁴ <https://nixos.org/manual/nixos/stable/index.html#ssec-machine-objects>

```
machine.wait_for_unit("default.target")
machine.succeed("su -- alice -c 'which firefox'")
machine.fail("su -- root -c 'which firefox'")
```

此Python脚本中的machine指代虚拟机配置名称，该配置用于节点属性集的nodes属性。

脚本会等待systemd达到default.target状态。通过su命令切换用户，并用which命令检查用户是否有firefox访问权限。预期which firefox命令对用户alice成功执行，对root用户执行失败。

该脚本将作为testScript属性的值。

完整的 minimal-test.nix 文件内容如下所示：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
5
pkgs.testers.runNixOSTest {
  name = "minimal-test";
8
  nodes.machine = { config, pkgs, ... }: {
10
    users.users.alice = {
      isNormalUser = true;
      extraGroups = [ "wheel" ];
      packages = with pkgs; [
        firefox
        树
      ];
16
    };
17
    system.stateVersion = "23.11";
18
  };
19
  testScript =
20
    machine.wait_for_unit("default.target")
    machine.succeed("su -- alice -c 'which firefox'")
    machine.fail("su -- root -c 'which firefox'")
21
    '';
22
}
```

运行测试

设置所有机器并运行测试脚本：

```
$ nix-build minimal-test.nix
```

```
...
测试脚本于10.96秒完成
清理中
终止机器进程 (pid 10)
(0.00 秒)
/nix/store/bx7z3imvxxpwkkza10vb23czhw7873w2-vm-test-run-minimal-test
```

虚拟机中的交互式 Python shell

在开发测试或出现故障时，能够交互式调试测试或访问机器终端非常有用。

要启动带测试框架的交互式 Python 会话：

```
§ $(nix-build -A driverInteractive minimal-test.nix)/bin/nixos-test-driver
```

此处可运行任何测试操作。通过 `test_script()` 函数执行 `minimal-test.nix` 中的 `testScript` 属性。

若虚拟机尚未启动，测试环境会在首次调用机器对象方法时自动处理。

但你也可以手动触发虚拟机启动，通过以下命令：

```
>>> machine.start()
```

针对特定节点，

or

```
>>> start_all()
```

针对所有节点。

你可以通过以下方式进入虚拟机的交互式终端：

```
>>> machine.shell_interact()
```

并运行如下 shell 命令：

```
uname -a
```

```
Linux server 5.10.37 #1-NixOS SMP Fri May 14 07:50:46 UTC 2021 x86_64 GNU/Linux
```

重新运行成功测试

由于测试结果保存在 Nix 存储中，成功的测试会被缓存。这意味着只要测试设置（节点配置和测试脚本）在语义上保持不变，Nix 不会重复运行该测试。因此，要重新运行测试，需先删除结果。

若尝试通过符号链接删除结果，将出现以下错误：

```
nix-store --delete ./result
```

```
正在查找垃圾回收器根节点...
```

```
0个存储路径被删除，释放0.00 MiB空间
```

```
错误：无法删除路径'/nix/store/4klj06bsilkqkn6h2sia8dcsi72wbcfl-vm-test-run-unnamed'，因该路径仍在使用中。查询原因请使用： nix-store --query --roots
```

相反，应先移除符号链接，再删除缓存结果：

```
rm ./result  
nix-store --delete /nix/store/4klj06bsilkqkn6h2sia8dcsi72wbcfl-vm-test-run-unnamed
```

也可通过一条命令完成：

```
result=$(readlink -f ./result) rm ./result && nix-store --delete $result
```

多虚拟机测试场景

测试可涉及多个虚拟机，例如用于测试客户端-服务器通信。

以下示例配置包含：

- 名为server的虚拟机，运行默认配置的nginx²⁰⁵。
- 名为client的虚拟机，配备curl工具以发起HTTP请求。
- 用于协调客户端与服务器间测试逻辑的testScript。

完整的 client-server-test.nix 文件内容如下所示：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in

pkgs testers.runNixOSTest {
  name = "client-server-test";

  nodes.server = { pkgs, ... }: {
    networking = {
      firewall = {
        allowedTCPPorts = [ 80 ];
      };
    };
    services.nginx = {
      enable = true;
      virtualHosts."server" = {};
    };
  };

  nodes.client = { pkgs, ... }: {
    environment.systemPackages = with pkgs; [
      curl
    ];
  };

  testScript =
    "server.wait_for_unit(\"default.target\")
     client.wait_for_unit(\"default.target\")
     client.succeed(\"curl http://server/ | grep -o \"Welcome to nginx!\"\"")
    ";
}
}
```

测试脚本执行以下步骤：

- 启动服务器并等待其准备就绪。
- 启动客户端并等待其准备就绪。
- 在客户端运行curl并使用grep检查预期返回字符串。测试根据返回值判断通过或失败。

运行测试：

```
$ nix-build client-server-test.nix
```

²⁰⁵ <https://nginx.org/en/>

关于NixOS测试的补充信息

- 在CI上运行集成测试需要硬件加速，但许多CI平台并不支持。
若要在GitHub Actions（第144页）中运行集成测试，请参阅如何禁用硬件加速²⁰⁶。
- NixOS自带大量测试用例，可作为学习范例。
一个很好的参考是Matrix与IRC桥接的实现²⁰⁷。

后续步骤

- 模块系统深度解析（第76页）
- 构建可启动的ISO镜像（第103页）
- 构建并运行Docker镜像（第104页）

通过SSH远程配置机器

借助nixos-anywhere²¹¹ 和disko²¹² 工具，可在运行中的Linux系统上直接替换为NixOS配置。

简介

本教程将指导您如何将NixOS配置部署到运行中的计算机。

您将学到什么？

您将学习如何：

- 通过声明式磁盘布局和SSH访问指定最小化NixOS配置
- 验证配置的有效性
- 在远程机器上部署及更新NixOS配置

你需要什么？

- 熟悉 Nix 语言（第13页）
- 熟悉模块系统（第73页）

为确保无人值守安装成功，请确认目标机器满足以下条件：

- 是运行 Linux 的 QEMU 虚拟机
 - 支持 kexec²¹³
 - 基于 x86-64 或 aarch64 指令集架构 (ISA)
 - 至少 1 GB 内存

²⁰⁶ <https://github.com/cachix/install-nix-action#how-do-i-run-nixos-tests>

²⁰⁷ <https://github.com/NixOS/nixpkgs/blob/master/nixos/tests/matrix/appservice-irc.nix>

²¹¹ <https://nix-community.github.io/nixos-anywhere/>

²¹² <https://github.com/nix-community/disko>

²¹³ <https://zh.wikipedia.org/wiki/Kexec>

这也可能是从USB启动的实时系统，例如NixOS安装程序²¹⁴。

- IP地址通过DHCP自动配置
- 您可以通过SSH登录
 - 使用公钥认证（推荐）或密码
 - 以root用户或具有sudo权限的其他用户身份

本地机器仅需安装可运行的Nix（第1页）。

本教程中将目标机器称为target-machine，请替换为实际的主机名或IP地址。

准备环境

新建项目目录并通过shell进入：

```
mkdir remote
cd remote
```

指定对nixpkgs、disko及nixos-anywhere的依赖（第138页）：

```
$ nix-shell -p npins
[nix-shell:remote]$ npins init
[nix-shell:remote]$ npins add github nix-community disk
[nix-shell:remote]$ npins add github nix-community nixos-anywhere
```

新建一个shell.nix文件，使用固定的依赖项提供所有必要工具：

```
let
  sources = 导入 ./npins;
  pkgs = 导入 sources.nixpkgs {};
in

pkgs.mkShell {
  nativeBuildInputs = 使用 pkgs 包含 [
    npins
    nixos-anywhere
    nixos-rebuild
  ];
  shellHook =
    export NIX_PATH="${sources.nixpkgs}:nixos-config=$PWD/configuration.nix"
    ->"";
}
```

现在退出临时环境并进入新指定的环境：

```
[nix-shell:remote]$ exit
$ nix-shell
```

该Shell环境已配置就绪，可通过nixos-anywhere和nixos-rebuild使用明确定义的Nixpkgs。

重要提示：所有后续命令均需在此环境中运行。

²¹⁴ <https://nixos.org/download/#download-nixos-accordion>

创建NixOS配置

新的NixOS配置将包含通用系统设置和磁盘布局规范

本示例的磁盘布局描述了一个单磁盘结构：包含主引导记录²¹⁵ (MBR)、EFI系统分区²¹⁶ (ESP)以及占用全部剩余空间的根文件系统。该方案同时兼容EFI和BIOS系统

创建新文件single-disk-layout.nix并写入磁盘布局规范：

```

1 { ... } :
2
3 {
4     disko.devices.disk.main = {
5         type = "disk";
6         内容 = {
7             类型 = "gpt";
8             分区 = {
9                 MBR = {
10                    优先级 = 0;
11                    大小 = "1M";
12                    类型 = "EF02";
13                };
14                ESP = {
15                    优先级 = 1;
16                    大小 = "500M";
17                    类型 = "EF00";
18                    内容 = {
19                        类型 = "文件系统";
20                        格式 = "vfat";
21                        挂载点 = "/boot";
22                    };
23                };
24            };
25            根目录 = {
26                优先级 = 2;
27                大小 = "100%";
28                内容 = {
29                    类型 = "文件系统";
30                    格式 = "ext4";
31                    挂载点 = "/";
32                };
33            };
34        };
35    };
36 }

```

创建 configuration.nix 文件，该文件需导入磁盘布局定义并指定要格式化的磁盘：

提示：若不清楚目标磁盘的设备标识符，可用 `lsblk` 列出目标机器上的所有设备：

```

$ ssh 目标机器 lsblk
名称 主:次 可移动 大小 只读 类型 挂载点
sda 8 : 0 0256G 0 磁盘
-sda1 8:1 0 248.5G 0 分区 /nix/store
/
L-sda2 8:2 0 7.5G 0 分区 [交换区]
sr0 11:0 1 1024M 0 光驱

```

215 <https://zh.wikipedia.org/wiki/主引导记录>

216 <https://zh.wikipedia.org/wiki/EFI系统分区>

在此示例中，磁盘名称为sda，块设备路径即为/dev/sda。请记下该值以备后用。

```

1 { modulesPath, ... }:
2
3 let
4   diskDevice = "/dev/sda";
5   sources = import ./npins;
6 in
7 {
8   imports = [
9     (modulesPath + "/profiles/qemu-guest.nix")
10    (sources.disko + "/module.nix")
11    ./single-disk-layout.nix
12  ];
13
14  disko.devices.disk.main.device = diskDevice;
15
16  boot.loader.grub = {
17    devices = [ diskDevice ];
18    efiSupport = 启用;
19    efiInstallAsRemovable = 启用;
20  };
21
22  services.openssh.enable = 是;
23
24  users.users.root.openssh.authorizedKeys.keys = [
25    "<在此填写您的SSH密钥>"
26  ];
27
28  system.version = "24.11";
29 }

```

重要提示：将 /dev/sda 替换为您的磁盘块设备路径。

将 <your SSH key here> 字符串替换为您希望用于未来以 root 用户身份登录的 SSH 公钥。

详细说明

let 块中的 diskDevice 变量定义了磁盘块设备的路径：

```

let
  diskDevice = "/dev/sda";
  sources = import ./npins;
in

```

用于设置磁盘分区与格式化的目标设备，如磁盘布局规范所述。同时该配置也用于引导加载程序，使其同时兼容传统BIOS与UEFI系统：

```

15 disko.devices.disk.main.device = diskDevice;
16
17 boot.loader.grub = {
18   devices = [ diskDevice ];
19   efiSupport = true;
20   efiInstallAsRemovable = true;
}

```

qemu-guest.nix 模块使该系统兼容在 QEMU 虚拟机内运行：

```
12 imports = [
  (modulesPath + "/profiles/qemu-guest.nix")
  (sources.disko + "/module.nix")
  ./single-disk-layout.nix
];
```

disko库根据磁盘布局规范生成分区脚本，并在NixOS配置中生成启动时自动挂载分区的对应部分。首行引入该库，次行应用磁盘布局：

```
12 imports = [
  (modulesPath + "/profiles/qemu-guest.nix")
  (sources.disko + "/module.nix")
  ./single-disk-layout.nix
];
```

测试磁盘布局

检查磁盘布局是否有效：

```
nix-build -E "((import <nixpkgs> {}).nixos [ ./configuration.nix ]).installTest"
```

该命令通过在disko模块提供的installTest属性中构建派生，在虚拟机中运行完整安装。

部署系统

要部署系统，需构建配置及对应的磁盘格式化脚本，并使用结果运行nixos-anywhere：

重要提示：请将target-host替换为目标主机的主机名或IP地址。

```
toplevel=$(nixos-rebuild build --no-flake)
diskoScript=$(nix-build -E "((import <nixpkgs> {}).nixos [ ./configuration.nix ]).<diskoScript")
nixos-anywhere --store-paths "$diskoScript" "$toplevel" root@目标主机
```

注意：若未启用公钥认证：请将环境变量SSH_PASS设为密码，并在nixos-anywhere命令后添加--env-password标志。

nixos-anywhere将登录目标系统，完成分区、格式化、挂载磁盘及安装NixOS配置，随后重启系统。

更新系统

```
npins update nixpkgs
nixos-rebuild switch --no-flake --target-host root@target-host
```

要更新系统，请运行npins并重新部署配置：除非需要更改磁盘布局，否则不再需要nixos-anywhere。

后续步骤

- 设置HTTP二进制缓存（第124页）
- 设置构建后钩子（第142页）

参考文献

- [nixos-anywhere项目页面](#)²¹⁷
- [disko项目仓库](#)²¹⁸
- [磁盘布局示例集](#)²¹⁹

在树莓派上安装NixOS

本教程假设您使用的是树莓派4B型4GB内存版本²²⁰。

开始前请确保备齐以下硬件²²¹：

- HDMI线缆/转换器
- 8GB以上容量的SD卡
- SD读卡器（若主机无SD插槽）
- 树莓派电源线。
- USB键盘。

注意：本教程针对树莓派4B编写。使用早期支持的型号如3B或3B+需对本教程进行适当修改。

启动NixOS live镜像

注意：从USB启动可能需要升级EEPROM固件。本教程选择从SD卡启动以避免此类问题。

若需在其他已安装Nix的设备上准备AArch64镜像，请运行以下命令：

²¹⁷ <https://nix-community.github.io/nixos-anywhere/>
²¹⁸ <https://github.com/nix-community/disko>
²¹⁹ <https://github.com/nix-community/disko/tree/master/example>
²²⁰ <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>
²²¹ <https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up/1>

```
$ nix-shell -p wget zstd  
[nix-shell:~]$ wget https://hydra.nixos.org/build/226381178/download/1/nixos-sd-image-23.11pre500597.0fbe93c5a7c-aarch64-linux.img.zst  
[nix-shell:~]$ unzstd -d nixos-sd-image-23.11pre500597.0fbe93c5a7c-aarch64-linux.img.zst  
[nix-shell:~]$ dmesg --follow
```

注：您可从 Hydra²²² 下载最新镜像，点击标记绿色对勾的最新成功构建，并复制构建产物镜像的链接。

注：若系统支持，使用 Etcher²²³ 等工具将镜像烧录至SD卡可能更为便捷。

终端会实时打印接收到的内核消息。

插入SD卡后，终端将打印其分配的设备名（例如 /dev/sdX）。

按 Ctrl+C 可停止 dmesg --follow 命令。

通过以下命令将NixOS复制到SD卡（请将sdX替换为您的设备名）：

```
[nix-shell:~]$ sudo dd if=nixos-sd-image-23.11pre500597.0fbe93c5a7c-aarch64-linux.  
↪img of=/dev/sdX bs=4096 conv=fsync status=progress
```

该命令执行完毕后，将SD卡插入树莓派并通电启动。

您将看到一个全新的终端界面！

若镜像无法启动，建议先更新固件²²⁴ 后重新引导镜像。

连接网络

执行sudo -i以获取root权限，便于完成后续教程。

此时您需要联网。若可使用以太网线，请插入并跳转至下一章节。

若需连接WiFi，请先运行iwconfig命令获取无线网卡名称。若为wlan0，请替换以下SSID和密码并执行：

```
#wpa_supplicant -B -i wlan0 -c <(wpa_passphrase 'SSID' 'passphrase') &
```

当终端显示连接成功后，运行host nixos.org命令检查DNS解析是否正常。

若输入有误，请执行pkill wpa_supplicant后重新开始。

222 https://hydra.nixos.org/job/nixos/trunk-combined/nixos_sd_image.aarch64-linux

223 <https://www.balena.io/etcher/>

224 https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#bootloader_update_stable

更新固件

为获取厂商提供的更新与错误修复，我们将首先更新树莓派固件：

```
#nix-shell -p raspberrypi-eeprom
#mount /dev/disk/by-label/FIRMWARE /mnt
#BOOTFS=/mnt FIRMWARE_RELEASE_STATUS=stable rpi-eeprom-update -d -a
```

安装并配置NixOS

现在我们将使用自定义配置安装NixOS，这里会创建一个访客用户并启用SSH守护进程。在下面的let绑定中，将SSID和SSIDpassword变量的值更改为您之前使用的SSID和密码短语：

```
{ config, pkgs, lib, ... }:

2  let
3    user = "guest";
4    password = "quest";
5    SSID = "mywifi";
6    SSID密码 = "mypassword";
7    接口 = "wlan0";
8    主机名 = "myhostname";
9    in {
10
11      boot = {
12        kernelPackages = pkgs.linuxKernel.packages.linux_rpi4;
13        initrd.availableKernelModules = [ "xhci_pci" "usbhid" "usb_storage" ];
14        loader = {
15          grub.enable = false;
16          generic-extlinux-compatible.enable = true;
17        };
18      };
19
20      fileSystems = {
21        "/" = {
22          device = "/dev/disk/by-label/NIXOS_SD";
23          fsType = "ext4";
24          options = [ "noatime" ];
25        };
26      };
27
28      网络配置 =
29        hostName = 主机名;
30        wireless = {
31          启用 = 真;
32          网络."${SSID}".密码 = SSID密码;
33          接口 = [ 接口 ];
34        };
35      };
36    };
37
38    环境.系统包 = 带 pkgs; [ vim ];
39
40    服务.openssh.启用 = 真;
41
42    用户 = {
43      mutableUsers = false;
44      users."${user}" = {
45        isNormalUser = true;
```

(接续下一页)

(continued from previous page)

```

48     password = password;
49     extraGroups = [ "wheel" ];
50   };
51
52   hardware.enableRedistributableFirstware = true;
53   system.stateVersion = "23.11";
}

```

为节省输入完整配置的时间，可下载现成文件：

```
#curl -L https://tinyurl.com/tutorial-nixos-install-rpi4 > /etc/nixos/
configuration.nix
```

注意：写入NixOS配置的凭证会在构建时以明文形式存储在/nix/store中。

若需避免此情况，可在控制台输入凭证或使用社区提供的加密方案。

由于nixos-sd-image的设计机制，此时系统已安装完成，仅需用新配置执行nixos-rebuild：

```
#nixos-rebuild boot
#reboot
```

若系统无法启动，请在引导加载器菜单中选择最旧的配置以返回实时映像并重新开始。

进行更改

已成功启动，恭喜！

如需进一步修改配置，请查阅NixOS选项²²⁵，编辑/etc/nixos/configuration.nix文件后更新系统：

```
§ sudo -i
#nixos-rebuild switch
```

后续步骤

- 当系统可运行后，可通过`nixos-rebuild switch --upgrade`升级以安装更新的软件包版本。若出现故障，可重启回滚至旧配置。
- 若要启用硬件加速以获得流畅的图形桌面体验，请在配置中添加nixos-hardware²²⁶模块：

```
imports = [
  "${fetchTarball "https://github.com/NixOS/nixos-hardware/tarball/master"}"
  <raspberry-pi/4"
];
```

我们建议将引用固定到nixos-hardware：固定Nixpkgs（第158页）

²²⁵ <https://search.nixos.org/options>

²²⁶ <https://github.com/nixos/nixos-hardware>

- 要调整影响硬件的引导加载程序选项，请参阅 config.txt 配置项²²⁷。可通过执行 mount /dev/disk/by-label/FIRMWARE /mnt 并编辑 /mnt/config.txt 来修改这些选项。

使用Terraform部署NixOS

假设您已掌握 Terraform 基础²²⁸，本教程结束时，您将能用 Terraform 配置亚马逊云服务(AWS)实例，并能使用 Nix 对该实例运行的 NixOS 进行增量变更部署。

我们将介绍如何启动 NixOS 机器及如何部署增量变更。

启动NixOS镜像

- 首先提供 Terraform 可执行文件：

```
$ nix-shell -p terraform
```

- 我们使用 Terraform Cloud²²⁹ 作为状态/锁定后端²³⁰：

```
$ terraform login
```

- 确保在你的 Terraform Cloud 账户中创建一个组织²³¹，例如 myorganization。
- 在 myorganization 内，通过选择 CLI 驱动的工作流创建一个工作区²³²，并指定名称，例如 myapp。
- 在你的工作区内，进入 Settings / General，将 Execution Mode 改为 Local。
- 在新目录中创建 main.tf 文件，内容如下。这将使用一个 SSH 密钥对和 SSH 安全组启动带有 NixOS 镜像的 AWS 实例：

```
terraform {
  backend "remote" {
    organization = "myorganization"

    workspaces {
      name = "myapp"
    }
  }
}

provider "aws" {
  region = "eu-central-1"
}

module "nixos_image" {
  source = "git::https://github.com/tweag/terraform-nixos.git//aws_image_nixos?
<ref=5f5a0408b299874d6a29d1271e9bffffee4c9ca71"
  release = "20.09"
}

resource "aws_security_group" "ssh_and_egress" {
  ingress {
    type        = "ssh"
    protocol   = "tcp"
    port        = 22
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

(接续下一页)

²²⁷ <https://www.raspberrypi.org/documentation/configuration/config-txt/>

²²⁸ <https://www.terraform.io/intro/index.html>

²³⁰ <https://www.terraform.io/docs/state/purpose.html>

²³¹ <https://app.terraform.io/app/organizations/new>

²³² <https://app.terraform.io/app/cachix/workspaces/new>

(continued from previous page)

```

    协议 = "tcp"
    cidr块 = [ "0.0.0.0/0" ]
}

出口 {
    起始端口      = 0
    目标端口      = 0
    协议          = "-1"
    CIDR区块     = ["0.0.0.0/0"]
}
}

资源 "tls_private_key" "state_ssh_key" {
    算法 = "RSA"
}

资源 "local_file" "machine_ssh_key" {
    敏感内容 = tls_private_key.state_ssh_key.private_key_pem
    文件名  = "${path.module}/id_rsa.pem"
    文件权限 = "0600"
}

资源 "aws_key_pair" "generated_key" {
    密钥名称 = "生成密钥-${sha256(tls_private_key.state_ssh_key.public_key_
.openssh) }"
    公钥 = tls_private_key.state_ssh_key.public_key_openssh
}

资源 "aws_instance" "machine" {
    ami = module.nixos_image.ami
    instance_type = "t3.micro"
    security_groups = [ aws_security_group.ssh_and_egress.name ]
    key_name = aws_key_pair.generated_key.key_name

    root_block_device {
        volume_size = 50 # GiB
    }
}

输出 "public_dns" {
    value = aws_instance.machine.public_dns
}

```

唯一与NixOS相关的代码片段是：

```

module "nixos_image" {
    source = "git::https://github.com/tweag/terraform-nixos.git/aws_image_nixos?
<\ref=5f5a0408b299874d6a29d1271e9bffffee4c9ca71"
    release = "20.09"
}

```

注意：aws_image_nixos模块将根据NixOS版本号²³³返回对应的NixOS AMI，以便aws_instance资源能在instance_type²³⁴参数中引用该AMI。

5. 确保配置好AWS凭证²³⁵。

6. 应用Terraform配置后即可获得运行中的NixOS系统：

²³³ <https://status.nixos.org>

²³⁴ <https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance-type>

²³⁵ <https://registry.terraform.io/providers/hashicorp/aws/latest/docs#认证>

```
$ terraform init
$ terraform apply
```

部署 NixOS 变更

当 AWS 实例通过 Terraform 运行 NixOS 镜像后，我们可以配置 Terraform 始终构建最新的 NixOS 配置并将变更应用到实例。

1. 创建包含以下内容的 configuration.nix 文件：

```
{ config, lib, pkgs, ... }:
  imports = [ <nixpkgs/nixos/modules/virtualisation/amazon-image.nix> ];
  #访问 https://search.nixos.org/options 查看所有选项
}
```

2. 将以下代码片段追加到您的 main.tf 文件中：

```
module "deploy_nixos" {
  source = "git::https://github.com/tweag/terraform-nixos.git//deploy_nixos?
<ref=5f5a0408b299874d6a29d1271e9bffeee4c9ca71"
  nixos_config = "${path.module}/configuration.nix"
  target_host = aws_instance.machine.public_ip
  ssh_private_key_file = local_file.machine_ssh_key.filename
  ssh_agent = false
}
```

3. 部署步骤：

```
$ terraform init
$ terraform apply
```

注意事项

- deploy_nixos 模块要求目标机器已安装 NixOS 且宿主机已安装 Nix
- 当客户端与目标机器架构不同时，deploy_nixos 模块将无法工作（除非使用分布式构建²³⁶）
- 如需向 Nix 注入值，目前尚无优雅解决方案
- 每台机器需单独评估，请注意内存需求会随机器数量线性增长

后续步骤

- 可切换至 Google Compute Engine²³⁷。
- deploy_nixos 模块²³⁸ 支持多种参数，例如上传密钥。

²³⁶ <https://nix.dev/manual/nix/stable/advanced-topics/distributed-builds.html>

²³⁷ https://github.com/tweag/terraform-nixos/tree/master/google_image_nixos#readme

²³⁸ https://github.com/tweag/terraform-nixos/tree/master/deploy_nixos#readme

配置HTTP二进制缓存

二进制缓存用于存储预构建的Nix存储对象²³⁹，并通过网络提供给其他机器。任何拥有Nix存储的机器均可作为其他机器的二进制缓存。

简介

本教程将指导您搭建一个Nix二进制缓存，通过HTTP或HTTPS从NixOS机器提供存储对象。

您将学到什么？

您将学习如何：

- 为您的缓存设置签名密钥
- 在作为缓存服务的NixOS机器上启用相应服务
- 验证配置是否按预期工作

您需要什么？

- 本地机器上可用的Nix安装（第1页）

- 用作缓存的NixOS机器的SSH访问权限

若初次接触NixOS，请先学习模块系统（第73页），并通过NixOS虚拟机（第98页）配置您的首个系统。

- （可选）公网IP与DNS域名

若不自建主机，可查阅NixOS Wiki推荐的兼容主机商²⁴⁰。按照《通过SSH配置远程主机》（第112页）教程部署NixOS配置。

针对本地网络缓存，我们假设：

- 主机名为cache（请替换为您的主机名或IP地址）

- 主机通过80端口提供HTTP存储服务（此为默认设置）

对于公开可访问的缓存，我们假设：

- 域名为cache.example.com（请替换为您自己的域名）

- 主机通过HTTPS在端口443上提供存储对象（此为默认设置）

需要多长时间？

- 25分钟

²³⁹ <https://nix.dev/manual/nix/latest/store/store-object>

²⁴⁰ <https://wiki.nixos.org/wiki/NixOS友好主机商>

设置服务

为托管缓存服务的NixOS主机，在binary-cache.nix中创建新配置模块：

```
{ config, ... }:

{
  services.nix-serve = {
    enable = true;
    secretKeyFile = "/var/secrets/cache-private-key.pem";
  };

  services.nginx = {
    enable = true;
    recommendedProxySettings = true;
    virtualHosts.cache = {
      locations."/".proxyPass = "http://${config.services.nix-serve.bindAddress}:$→{toString config.services.nix-serve.port}";
    };
  };

  networking.firewall.allowedTCPPorts = [
    config.services.nginx.defaultHTTPListenPort
  ];
}
```

services.nix-serve²⁴¹ 下的选项用于配置二进制缓存服务。

nix-serve 不支持 IPv6 或 SSL/HTTPS。需通过 services.nginx²⁴² 选项设置代理（支持 IPv6）来处理主机名缓存请求。

重要提示：本教程末尾包含可选的 HTTPS 章节（第127页）。

将新 NixOS 模块添加至现有机器配置：

```
{ config, ... }:

{
  imports = [
    ./binary-cache.nix
  ];

  #...
}
```

从本地机器部署新配置：

```
nixos-rebuild switch --no-flake --target-host root@cache
```

注意：二进制缓存守护进程将报错，因为尚无密钥文件。

²⁴¹ <https://search.nixos.org/options?query=services.nix-serve>

²⁴² <https://search.nixos.org/options?query=services.nginx>

生成签名密钥对

需要一对公私钥来确保缓存中的存储对象真实可信。

为二进制缓存生成密钥对时，请将示例主机名 cache.example.com 替换为你的主机名：

```
nix-store --generate-binary-cache-key cache.example.com cache-private-key.pem  
→ cache-public-key.pem
```

cache-private-key.pem 将由二进制缓存守护程序用于在提供二进制文件时进行签名。请将其复制到托管缓存的机器上 services.nix-serve.secretKeyFile 配置的路径中：

```
将 cache-private-key.pem 通过 scp 传输到缓存服务器：scp cache-private-key.pem root@cache:/var/secrets/cache-private-key.pem
```

此前由于缺少密钥文件，二进制缓存守护进程一直处于重启循环状态。请验证其当前是否正常运行：

```
通过 ssh 检查服务状态：ssh root@cache systemctl status nix-serve.service
```

重要提示：在本地机器上配置 Nix 使用自定义二进制缓存（第135页），需用到 cache-public-key.pem 文件

测试可用性

以下步骤用于验证所有配置是否正确，并有助于排查潜在问题

检查通用可用性

通过查询缓存测试二进制缓存、反向代理和防火墙规则是否按预期工作：

```
$ curl http://cache/nix-cache-info  
StoreDir: /nix/store  
WantMassQuery: 1  
Priority: 30
```

检查存储对象签名

要测试存储对象是否正确签名，请检查示例衍生的元数据。在二进制缓存主机上构建 hello 包，并从缓存中获取.narinfo 文件：

```
§ hash=$(nix-build '<nixpkgs>' -A pkgs.hello | awk -F '/' '{print $4}' | awk -F '-'  
<'{print $1}'  
$ curl "http://cache/$hash.narinfo" | grep "Sig: "  
...  
Sig: cache.example.  
→ org:GyBFzocLAeLEFdOhr2noK84VzPUw0ArCNYEnrm1YXakdsC5FkO2Bkj2JH8Xjou+wxexMjFKa0YP2AML7nBWsaG==
```

确保输出包含以 Sig: 为前缀的该行，并显示您生成的公钥。

通过HTTPS提供二进制缓存服务

若二进制缓存可公开访问，可通过Let's Encrypt²⁴³ SSL证书强制启用HTTPS。按如下方式编辑binary-cache.nix文件，并确保将示例URL和邮箱替换为您自己的：

```
services.nginx = {
  enable = true;
  recommendedProxySettings = true;
-  virtualHosts.cache = {
+  virtualHosts."cache.example.com" = {
+    enableACME = true;
+    forceSSL = true;
    locations."/".proxyPass = "http://${config.services.nix-serve.bindAddress}:${
→ {toString config.services.nix-serve.port}";
    };
  };

+  security.acme = {
+    acceptTerms = true;
+    certs = {
+      "cache.example.com".email = "you@example.com";
+    };
+  };
};

networking.firewall.allowedTCPPorts = [
  config.services.nginx.defaultHTTPListenPort
  config.services.nginx.defaultSSLListenPort
];
```

重建系统以部署这些更改：

```
nixos-rebuild switch --no-flake --target-host root@cache.example.com
```

后续步骤

若您的二进制缓存已是远程构建机器²⁴⁴，它将提供其Nix存储中所有存储对象。

- 配置Nix使用自定义二进制缓存（第135页），需指定缓存主机名及生成的公钥
- 设置构建后钩子（第142页）将存储对象上传至二进制缓存
- 配置分布式构建（第128页）

为节省存储空间，请参考以下NixOS配置属性：

- nix.gc²⁴⁵：自动垃圾回收的选项
- nix.optimize²⁴⁶：定期优化Nix存储的选项

²⁴³ <https://letsencrypt.org/>

²⁴⁴ <https://nix.dev/manual/nix/latest/advanced-topics/distributed-builds>

²⁴⁵ <https://search.nixos.org/options?query=nix.gc>

²⁴⁶ <https://search.nixos.org/options?query=nix.optimize>

替代方案

- nix-serve-ng²⁴⁷：用 Haskell 编写的 nix-serve 直接替代品
- SSH 存储²⁴⁸、实验性 SSH 存储²⁴⁹ 和 S3 二进制缓存存储²⁵⁰ 也可用于提供缓存服务。有许多提供 S3 兼容存储的商业供应商，例如：
 - 亚马逊 S3
 - Tigris
 - Cloudflare R2
- attic²⁵¹：基于S3兼容存储的Nix二进制缓存服务器
- Cachix²⁵²：托管式Nix二进制缓存服务

参考文献

- Nix手册关于HTTP二进制缓存存储的说明²⁵³
- services.nix-serve模块选项²⁵⁴
- services.nginx模块选项²⁵⁵

配置分布式构建

Nix 可通过将构建任务分散到多台计算机来加速构建过程

简介

本教程将指导您配置独立构建机，并设置本地机将构建任务卸载至该机器

您将学到什么？

您将学习如何

- 为远程构建访问创建新用户，实现本地机器到远程构建器的连接
- 以可持续配置方式设置远程构建器
- 测试远程构建器连接与认证
- 配置本地机器实现构建任务自动分发

247 <https://github.com/aristanetworks/nix-serve-ng>

248 <https://nix.dev/manual/nix/latest/store/types/ssh-store>

249 <https://nix.dev/manual/nix/latest/store/types/experimental-ssh-store>

250 <https://nix.dev/manual/nix/latest/store/types/s3-binary-cache-store>

251 <https://github.com/zhaofengli/attic>

252 <https://www.cachix.org>

253 <https://nix.dev/manual/nix/latest/store/types/http-binary-cache-store>

254 <https://search.nixos.org/options?query=services.nix-serve>

255 <https://search.nixos.org/options?query=services.nginx>

你需要什么？

- 熟悉 Nix 语言（第 13 页）
- 熟悉模块系统（第 73 页）
- 本地机器（示例主机名：localmachine）
 - 已安装 Nix 的计算机（第 1 页），用于将构建任务分发至其他机器。
- 远程机器（示例主机名：remotemachine）
 - 一台运行NixOS的计算机，用于接收来自本地机器的构建任务。按照《通过 SSH配置远程机器》（第112页）的步骤来设置远程NixOS系统。

需要多长时间？

- 25分钟

创建SSH密钥对

本地机器的Nix守护进程以root用户身份运行，需要私钥文件向远程机器验证身份。远程机器则需要公钥来识别本地机器。

在本地机器上，以root身份运行以下命令来创建SSH密钥对：

```
#ssh-keygen -f /root/.ssh/remotebuild
```

注：密钥对文件的名称和存储位置可自由选择。

配置远程构建器

在远程主机的NixOS配置目录中，创建remote-builder.nix文件：

```
{
  users.users.remotebuild = {
    isSystemUser = true;
    group = "remotebuild";
    useDefaultShell = true;

    openssh.authorizedKeys.keyFiles = [ ./remotebuild.pub ];
  };

  users.groups.remotebuild = {};
  nix.settings.trusted-users = [ "remotebuild" ];
}
```

将remotebuild.pub文件复制到本目录。

该配置模块将创建一个无主目录的新用户remotebuild。本地机器的root用户可通过之前生成的SSH密钥登录远程构建器。

将新NixOS模块添加至远程机器的现有配置中：

```
{
  imports = [
    ./remote-builder.nix
  ];
  #...
}
```

以 root 身份激活新配置：

```
nixos-rebuild switch --no-flake --target-host root@remotemachine
```

测试认证

确保 SSH 连接和认证正常工作。在本地机器上以 root 身份运行：

```
#ssh remotebuild@remotemachine -i /root/.ssh/remotebuild "echo hello"
hello
```

若能看到 hello 消息，则认证成功。

此次测试登录还会将远程构建器的主机密钥添加到本地机器的 /root/.ssh/known_hosts 文件中。后续登录将不再受主机密钥检查干扰。

配置分布式构建

注意：若本地机器运行 NixOS，请跳过本节并通过模块选项配置 Nix（见第131页）。

通过以root身份在Nix配置文件²⁵⁶ 中添加以下内容，配置Nix使用远程构建器：

```
#cat << EOF >> /etc/nix/nix.conf
builders = ssh-ng://remotebuild@remotebuilder $(nix-instantiate --eval -E builtins.
<currentSystem>/root/.ssh/remotemachine - - nixos-test,big-parallel,kvm
builders-use-substitutes = true
```

详细说明

首行通过指定以下内容将远程机器注册为远程构建器

- 协议、用户及主机名
- 本地机器的系统类型²⁵⁷

这将把该类型的系统任务委托给远程机器处理。

- SSH密钥的位置
- 支持的系统特性列表²⁵⁸

必须指定此特定列表，以便将构建编译器和运行NixOS虚拟机测试（第107页）的任务委托给远程机器。

详情请参阅构建器设置²⁵⁹ 的参考文档。

²⁵⁶ <https://nix.dev/manual/nix/2.23/command-ref/conf-file>

²⁵⁷ <https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-system>

²⁵⁸ <https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-system-features>

²⁵⁹ <https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-builders>

第二行指令要求所有远程构建器从其自身的二进制缓存获取依赖项，而非本地机器。这默认远程构建器的网络连接速度至少与本地机器相当。

激活此配置需重启Nix守护进程：

Linux

在systemd管理的Linux系统中，以root身份执行：

```
#systemctl restart nix-daemon.service
```

macOS

在macOS上，以root身份运行：

```
#sudo launchctl stop org.nixos.nix-daemon
#sudo launchctl start org.nixos.nix-daemon
```

NixOS

若本地机器运行NixOS，请在其配置目录中创建distributed-builds.nix文件：

```
{ pkgs, ... }:
{
  nix.distributedBuilds = true;
  nix.settings.builders-use-substitutes = true;

  nix.buildMachines = [
    {
      hostName = "remotebuilder";
      sshUser = "remotebuild";
      sshKey = "/root/.ssh/remotebuild";
      system = pkgs.stdenv.hostPlatform.system;
      supportedFeatures = [ "nixos-test" "big-parallel" "kvm" ];
    }
  ];
}
```

详细说明

该配置模块启用分布式构建并添加远程构建器，具体指定：

- SSH主机名与用户名
- SSH密钥的存储位置
- 本地机器的系统类型²⁶⁰

这将把该类型系统的任务委派给远程机器执行。

- 支持的系统特性列表²⁶¹

必须指定此特定列表，才能将构建编译器及运行NixOS虚拟机测试（第107页）的任务委派给远程机器。

²⁶⁰ <https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-system>

²⁶¹ <https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-system-features>

详情请参阅 nix.buildMachines²⁶² 的 NixOS 选项文档。

builders-use-substitutes 指令要求所有远程构建机从自身的二进制缓存获取依赖项，而非本地机器。此操作前提是远程构建机的网络连接速度至少与本地机器相当。

将新的 NixOS 模块添加至现有机器配置：

```
{
  imports = [
    ./distributed-builds.nix
  ];
  #...
}
```

以 root 身份激活新配置：

```
#nixos-rebuild switch
```

测试分布式构建

尝试在本地机器上构建新派生：

```
$ nix-build --max-jobs 0 -E "$(cat << EOF
(import <nixpkgs> {}).writeText "test" "$(date)"
EOF
)"
将构建此派生：
/nix/store/9csjdx6ir8ccnj16ijs36izswjgchn0-test.drv
正在 'ssh://' 上构建 '/nix/store/9csjdx6ir8ccnj16ijs36izswjgchn0-test.drv'
<remotebuilder'...
正在复制 0 个路径...
正在复制 1 个路径...
正在从 'ssh://' 复制路径 '/nix/store/hvj5vyg4723nly1qh5a8daifbi1yisb3-test'
<remotebuilder'...
/nix/store/hvj5vyg4723nly1qh5a8daifbi1yisb3-test
```

由于依赖当前系统时间，每次调用都会导致衍生结果不同，因此该结果永远无法存入本地缓存。命令行参数 --max-jobs 0²⁶³ 会强制 Nix 在远程构建器上执行构建。

最后一行输出包含输出路径，表明构建分发功能符合预期。

优化远程构建器配置

为最大化并行效率、启用自动垃圾回收并防止 Nix 构建占用全部内存，请将以下内容添加到 remote-builder.nix 配置模块中：

```
{
  users.users.remotebuild = {
    isNormalUser = true;
    createHome = false;
    group = "remotebuild";
  };
  openssh.authorizedKeys.keyFiles = [ ./remotebuild.pub ];
```

(下页继续)

²⁶² <https://search.nixos.org/options?query=nix.buildMachines>

²⁶³ <https://nix.dev/manual/nix/2.23/command-ref/conf-file#conf-max-jobs>

(continued from previous page)

```

};

users.groups.remotebuild = {};

nix.settings.trusted-users = [ "remotebuild" ];
nix = {
  nrBuildUsers = 64;
  settings = {
    trusted-users = [ "remotebuild" ];

    最小空闲 = 10 * 1024 * 1024;
    最大空闲 = 200 * 1024 * 1024;

    +  最大任务数 = "自动";
    +  核心数 = 0;
    +  };
    +  };

  systemd.services.nix-daemon.serviceConfig = {
    +  内存统计 = true;
    +  MemoryMax = "90%";
    +  OOMScoreAdjust = 500;
    +  };
  };
}

```

提示：参阅 Nix 参考手册²⁶⁴ 了解 nix.settings 中可用选项的详细信息²⁶⁵。

远程构建器可能具有不同的性能特征。请为每个 nix.buildMachines 条目正确设置 maxJobs、speedFactor 和 supportedFeatures 属性，以适应不同的远程构建器。这有助于本地 Nix 以最优方式分配构建任务。

提示：参阅 NixOS 关于 nix.buildMachines 的选项文档²⁶⁶ 获取详细信息。

设置 nix.buildMachines.*.publicHostKey 字段为每个远程构建器的公共主机密钥，以防止构建分发过程中的中间人攻击。

后续步骤

- 在每个远程构建器上配置 Nix 以使用自定义二进制缓存（第135页）
- 设置构建后钩子（第142页）将存储对象上传至二进制缓存

如需配置多个构建器，请为每个远程构建器重复“设置远程构建器（第129页）”章节的步骤，并将所有新增构建器添加到“设置分布式构建（第130页）”章节所示的 [nix.buildMachines 属性中](#)。

²⁶⁴ <https://nix.dev/manual/nix/2.23/command-ref/conf-file>

²⁶⁵ <https://search.nixos.org/options?show=nix.settings>

²⁶⁶ <https://search.nixos.org/options?query=nix.buildMachines>

替代方案

- nixbuild.net²⁶⁷ - 作为服务的Nix远程构建器
- Hercules CI²⁶⁸ - 支持自动构建分发的持续集成
- garnix²⁶⁹ - 托管式支持构建分发的持续集成

参考文献

- Nix参考手册：分布式构建配置项²⁷⁰

²⁶⁷ <https://nixbuild.net>

²⁶⁸ <https://hercules-ci.com/>

²⁶⁹ <https://garnix.io/>

²⁷⁰ <https://nix.dev/manual/nix/latest/command-ref/conf-file#conf-builders>

这些章节包含实用操作指南。

3.1 配方集

3.1.1 配置Nix使用自定义二进制缓存

可通过设置 `substituters`²⁷¹ 与 `trusted-public-keys`²⁷² 将Nix配置为使用指定二进制缓存，既可单独使用也可与 `cache.nixos.org`²⁷³ 共存。

提示：按照教程设置HTTP二进制缓存（第124页）并创建用于签名存储对象的密钥对。

例如，假设存在一个位于 `https://example.org` 且公钥为 `My56...Q== - %` 的二进制缓存，且 `default.nix` 中包含某些派生项，可通过命令行参数²⁷⁴ 临时指定Nix独占使用该缓存：

```
$ nix-build --substituters https://example.org --trusted-public-keys example.  
<org: My 56...Q == -
```

若要永久使用自定义缓存（同时保留公共缓存），请将以下内容添加至Nix配置文件²⁷⁵：

```
$ echo "extra-substituters = https://example.org" >> /etc/nix/nix.conf  
$ echo "extra-trusted-public-keys = example.org:My56...Q=="" >> /etc/nix/nix.conf
```

若希望始终仅使用自定义缓存：

```
$ echo "substituters = https://example.org" >> /etc/nix/nix.conf  
$ echo "trusted-public-keys = example.org:My56...Q==%">> /etc/nix/nix.conf
```

NixOS

在NixOS系统中，Nix通过`nix.settings`²⁷⁶ 选项进行配置：

```
{ ...}: {  
    nix.settings = {  
        substituters = [ "https://example.org" ];  
        trusted-public-keys = [ "example.org:My56...Q==," ];  
    };  
}
```

²⁷¹ <https://nix.dev/manual/nix/2.21/command-ref/conf-file.html#conf-substituters>

²⁷² <https://nix.dev/manual/nix/2.21/command-ref/conf-file.html#conf-trusted-public-keys>

²⁷³ <http://cache.nixos.org>

²⁷⁴ <https://nix.dev/manual/nix/2.21/command-ref/conf-file#command-line-flags>

²⁷⁵ <https://nix.dev/manual/nix/2.21/command-ref/conf-file#configuration-file>

²⁷⁶ <https://search.nixos.org/options?show=nix.settings>

3.1.2 使用 direnv 自动激活环境

无需手动为每个项目激活环境，每当进入项目目录或修改其中的 shell.nix 文件时，都可重新加载声明式 shell（第9页）。

1. 确保 nix-direnv 可用²⁷⁷

2. 将其集成至你的 shell²⁷⁸

例如，编写包含以下内容的 shell.nix 文件：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
5
pkgs.mkShellNoCC {
  packages = with pkgs;
  hello
9
};;
10}
```

从项目根目录运行：

```
$ echo "use nix" > .envrc && direnv allow
```

下次启动终端并进入项目根目录时，direnv 将自动加载 shell.nix 中定义的 shell。

```
$ cd myproject
$ which hello
/nix/store/1gxz5nfzfnhyxjdyzi04r86sh61y4i00-hello-2.12.1/bin/hello
```

direnv 还会检测 shell.nix 文件的变更。

进行如下添加：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
pkgs.mkShellNoCC {
  packages = with pkgs;
  你好
};;
+
+ shellHook =
+ 你好
+ ";
}
```

运行环境应在首次交互（执行任意命令或按回车键）后自动重新加载。

```
你好，世界！
```

²⁷⁷ <https://github.com/nix-community/nix-direnv>

²⁷⁸ <https://direnv.net/docs/hook.html>

3.1.3 开发环境中的依赖项

在 default.nix 中打包软件（第41页）时，您会需要在 shell.nix（第9页）中配置开发环境，以便通过 nix-shell 便捷进入，或通过 direnv（第136页）自动加载。

如何将 default.nix 中的软件包依赖项共享给 shell.nix 中的开发环境？

总结

使用 inputsFrom 属性传递给 pkgs.mkShellNoCC²⁷⁹：

```
#default.nix
let
  pkgs = import <nixpkgs> {};
  build = pkgs.callPackage ./build.nix {};
in
{
  继承 build;
  shell = pkgs.mkShellNoCC {
    inputsFrom = [ build ];
  };
}
10
11 }
```

在 shell.nix 中导入 shell 属性：

```
#shell.nix
(import ./).shell
```

完整示例

假设你的构建定义在 build.nix 中：

```
#build.nix
{ cowsay, runCommand }:
runCommand "cowsay-output" { buildInputs = [ cowsay ]; } "
  cowsay 你好, Nix! > $out
11
```

本示例中，cowsay 通过 buildInputs 被声明为构建时依赖项。

进一步假设你的项目定义在 default.nix 中：

```
#default.nix
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
in
{
  build = pkgs.callPackage ./build.nix {};
```

在 default.nix 中添加一个指定环境的属性：

```
let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
  in
  {
```

(接下一页)

²⁷⁹ <https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-mkShell-attributes>

(continued from previous page)

```

build = pkgs.callPackage ./build.nix {};
+ shell = pkgs.mkShellNoCC {
+ };
}

```

将 build 属性移至 let 绑定中以便复用，随后通过 inputsFrom²⁸⁰ 将包的依赖引入环境：

```

let
  nixpkgs = fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11";
  pkgs = import nixpkgs { config = {}; overlays = []; };
+ build = pkgs.callPackage ./build.nix {};
in
{
-  build = pkgs.callPackage ./build.nix {};
+ 继承 build;
  shell = pkgs.mkShellNoCC {
+   inputsFrom = [ build ];
  };
}

```

最后，在 shell.nix 中导入 shell 属性：

```
#shell.nix
(import ./).shell
```

检查开发环境，其中包含构建时依赖 cowsay：

```
$ nix-shell --pure
[nix-shell]$ cowsay shell.nix
```

后续步骤

- 实现可复现性：固定Nixpkgs版本（第12页）
- 使用direnv自动激活环境（第136页）
- 搭建Python开发环境（第140页）
- 用Nix打包现有软件（第41页）

3.1.4 使用npins自动管理远程源

Nix语言可用于描述由Nix管理的文件间依赖关系。Nix表达式本身可依赖远程源，其来源有多种指定方式，如《实现可复现性：固定Nixpkgs版本》（第12页）所示。

如需实现远程源处理的更多自动化，请在项目中配置 npins²⁸¹：

```
$ nix-shell -p npins --run "npins init --bare; npins add github nixos nixpkgs --
-branck nixos-23.11"
```

此命令将获取 Nixpkgs 23.11 发布分支的最新版本。当前目录下会生成 npins/sources.json 文件，其中包含对获取版本的固定引用，同时创建 npins/default.nix 文件，将这些依赖项暴露为属性集。
将生成的 npins/default.nix 作为参数默认值导入 default.nix 中的函数，并用于引用 Nixpkgs 源目录：

²⁸⁰ <https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-mkShell-attributes>

²⁸¹ <https://github.com/andir/npins/>

```

{
  sources ? 导入 ./npins,
  system? 内置函数.currentSystem,
  pkgs ? 导入 sources.nixpkgs { 继承 system; config = {}; overlays = []; },
}:
{
  package = pkgs.hello;
}

```

nix-build 将使用空属性集 {} 调用顶层函数，或通过 --arg²⁸² 和 --argstr²⁸³ 传递参数。这种模式支持以编程方式覆盖远程源码（第139页）。

将 npins 添加到项目开发环境中以便随时使用：

```

{
  源 ? 导入 ./npins,
  系统 ? builtins.currentSystem,
  pkgs ? 导入 sources.nixpkgs { 继承 system; config = {}; overlays = []; },
}:
- {
+递归 {
  包 = pkgs.hello;
  shell = pkgs.mkShellNoCC {
    inputsFrom = [ 包 ];
    packages = with pkgs; [
      npins
    ];
  };
}

```

同时添加 shell.nix 以便更便捷地进入该环境：

```
(import ./ {}).shell
```

详情参阅开发环境中的依赖项（第137页），注意此处需向导入的表达式传递空属性集，因 default.nix 现在包含一个函数。

覆盖源

例如，我们将使用先前创建的表达式搭配旧版Nixpkgs。

进入开发环境，创建新目录，并用另一版本的Nixpkgs初始化npins：

```
$ nix-shell
[nix-shell]$ mkdir old
[nix-shell]$ cd old
[nix-shell]$ npins init --bare
[nix-shell]$ npins add github nixos nixpkgs --branch nixos-21.11
```

在新目录中创建 default.nix 文件，用于导入原始文件及刚创建的源。

```
import ./default.nix { sources = import ./npins; }
```

这将导致构建出不同版本：

```
$ nix-build -A build
$ ./result/bin/hello --version | head -1
hello (GNU Hello) 2.10
```

²⁸² <https://nix.dev/manual/nix/stable/command-ref/nix-build#opt-arg>

²⁸³ <https://nix.dev/manual/nix/stable/command-ref/nix-build#opt-argstr>

源文件也可通过命令行参数覆盖：

```
nix-build .. -A build --arg sources 'import ./npins'
```

从niv迁移

本指南旧版曾推荐使用niv²⁸⁴——一个基于Haskell编写的类似依赖锁定工具。

若现有项目使用niv，可通过以下命令将远程源定义导入npins：

```
npins import-niv
```

警告：所有导入的条目都将被更新，因此它们不一定指向与之前相同的提交！

后续步骤

- 查看内置帮助以获取更多信息：

```
npins --help
```

- 有关指定远程源的不同方式的详细说明和示例，请参阅《实现可复现性：固定Nixpkgs》（第12页）。

3.1.5 设置Python开发环境

在本示例中，您将通过练习使用Flask²⁸⁵ 网络框架构建一个Python网络应用程序。为充分发挥其作用，您应当熟悉声明式Shell环境的定义（第9页）。

新建名为myapp.py的文件并添加以下代码：

```
#!/usr/bin/env python

from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return {
        "message": "Hello, Nix!"
    }

def run():
    app.run(host="0.0.0.0", port=5000)

if name == "main__":
    run()
```

这是一个简单的Flask应用，用于提供包含消息“Hello, Nix!”的JSON文档。

新建 shell.nix 文件以声明开发环境：

²⁸⁴ <https://github.com/nmattia/niv/>

²⁸⁵ <https://flask.palletsprojects.com>

```
{ pkgs ? import (fetchTarball "https://github.com/NixOS/nixpkgs/tarball/nixos-23.11
→") { } } :

pkgs.mkShellNoCC {
  packages = with pkgs; [
    (python3.withPackages (ps: [ ps.flask ]))
    curl
    jq
  ];
}
```

这里描述了一个包含python3实例的shell环境，该实例通过python3.withPackages²⁸⁶ 集成了flask包。同时包含curl²⁸⁷（一个执行网络请求的工具）和jq²⁸⁸（一个解析与格式化JSON文档的工具）。这两者都不是Python包。若使用Python的virtualenv²⁸⁹，则需额外手动操作才能将这些工具添加到开发环境。

运行nix-shell即可进入刚声明的环境：

```
$ nix-shell
将构建以下2个派生项:
/nix/store/5yvz7zf8yzck6r9z4f1br9sh71vqkimk-builder.pl.drv
/nix/store/aihgjkf856dbpjqlgrdmxyyd8a5j2m-python3-3.9.13-env.drv
将获取以下93个路径 (下载量109.50 MiB, 解压后468.52 MiB):
/nix/store/0xxjx37fcy2nl3yz6igmv4mag2a7giq6-glibc-2.33-123
/nix/store/138azk9hs5a2yp3zzx6iy1vdw19q26wv-hook
...
[nix-shell:~]$
```

在此Shell环境中启动Web应用程序：

```
[nix-shell:~]$ python ./myapp.py
* 正在运行 Flask 应用 'myapp'
* 调试模式: 关闭
警告: 这是开发服务器。请勿在生产环境中使用。
→ 请改用生产级 WSGI 服务器。
* 正在所有地址上运行 (0.0.0.0)
* 运行于 http://127.0.0.1:5000
* 运行于 http://192.168.1.100:5000
按 CTRL+C 退出
```

现在您已拥有一个运行中的 Python 网络应用，试试看吧！

打开新终端以启动另一个 shell 环境会话，并执行以下命令：

```
$ nix-shell

[nix-shell:~]$ curl 127.0.0.1:5000
{"message": "你好, Nix! "}

[nix-shell:~]$ curl 127.0.0.1:5000 jq '.message'
% 总量 % 已接收 % 传输 平均速度 时间 时间 时间 当前
下载 上传 总计 耗时 剩余 速度
100 26 100 26 00 0 13785 0 --:--:-- --:--:-- 2:000
"你好, Nix! "
```

如示例所示，您无需手动安装即可同时使用curl和jq来测试运行的Web应用程序。

²⁸⁶ <https://nixos.org/manual/nixpkgs/stable/#python.withpackages-function>

²⁸⁷ <https://search.nixos.org/packages?show=curl>

²⁸⁸ <https://search.nixos.org/packages?show=jq>

²⁸⁹ <https://virtualenv.pypa.io/en/latest/>

您可以将我们创建的文件提交到版本控制系统，并与其他人员共享。只要安装了Nix（第1页），其他人现在就能使用相同的Shell环境。

后续步骤

- 使用Nix打包现有软件（第41页）
- 处理本地文件（第58页）
- 使用direnv自动激活环境（第136页）
- 使用npins自动管理远程源（第138页）

3.1.6 设置构建后钩子

本指南展示如何通过Nix的post-build-hook²⁹⁰ 配置选项，自动将构建结果上传至兼容S3的二进制缓存²⁹¹。

实现注意事项

这是一个简单可用的示例，但并非适用于所有场景。

构建后钩子程序会在每次构建完成后运行，并阻塞构建循环。若钩子程序失败，构建循环将终止。

具体而言，当网络连接缓慢或不稳定时，此实现会导致Nix变慢或不可用。更高级的实现可将存储路径传递至用户提供的守护进程或队列，以便在构建循环外处理存储路径。

前提条件

本教程假设您已配置兼容S3的二进制缓存²⁹²，且root用户的默认AWS配置可上传至该存储桶。

设置签名密钥

使用nix-store --generate-binary-cache-key²⁹³ 创建一对加密密钥。私钥用于路径签名，公钥则分发给用户以验证路径真实性。

```
$ nix-store --generate-binary-cache-key example-nix-cache-1 /etc/nix/key.private /  
→ etc/nix/key.public  
$ cat /etc/nix/key.public  
example-nix-cache-1:1:cKDz3QCCOmwcztD2eV6Coggp6rqc9DGjWv7C0G+rM=
```

配置Nix以在访问存储桶的任何机器上使用自定义二进制缓存（第135页）。例如，将缓存URL添加到substituters²⁹⁴，并将公钥添加到nix.conf中的trusted-public-keys²⁹⁵：

```
substituters = https://cache.nixos.org/ s3://example-nix-cache  
trusted-public-keys = cache.nixos.org-  
↔ 1 : 6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY = example-nix-cache-1:1/  
↔ cKDz3QCCOmwcztD2eV6Coggp6rqc9DGjWv7C0G+rM=
```

²⁹⁰ <https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-post-build-hook>

²⁹¹ <https://nix.dev/manual/nix/2.22/store/types/s3-binary-cache-store>

²⁹² <https://nix.dev/manual/nix/2.22/store/types/s3-binary-cache-store#authenticated-writes-to-your-s3-compatible-binary-cache>

²⁹³ <https://nix.dev/manual/nix/2.22/command-ref/nix-store/generate-binary-cache-key>

²⁹⁴ <https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-substituters>

²⁹⁵ <https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-trusted-public-keys>

为缓存构建的机器必须使用私钥对派生进行签名。需将包含刚生成私钥的文件路径添加到这些机器的 secret-key-files²⁹⁶ 配置中：

```
secret-key-files = /etc/nix/key.private
```

实现构建钩子

将以下脚本写入 /etc/nix/upload-to-cache.sh：

```
#!/bin/sh
set -eu
set -f # 禁用通配符扩展
export IFS=' '
echo "正在上传路径" SOUT_PATHS
exec nix copy --to "s3://example-nix-cache" SOUT_PATHS
```

\$SOUT_PATHS变量是以空格分隔的Nix存储路径列表。此处我们期望且需要shell执行单词分割，使每个输出路径成为nix store sign的独立参数。Nix保证路径不会包含空格，但存储路径可能包含通配字符。set -f用于禁用shell的通配符扩展。

确保root用户可执行钩子程序：

```
#chmod +x /etc/nix/upload-to-cache.sh
```

更新Nix配置

在本地机器上设置构建后钩子²⁹⁷ 配置选项以运行钩子：

```
post-build-hook = /etc/nix/upload-to-cache.sh
```

然后在所有相关机器上重启nix-daemon，例如执行：

```
pkkill nix-daemon
```

测试

构建任意派生项，例如：

```
$ nix-build -E '(import <nixpkgs> {}).writeText "example" (builtins.toString_
<builtins.currentTime>)'
将构建此派生项：
/nix/store/s4pnfbkalzy5qz57qs6yybna8wylkig6-example.drv
构建 '/nix/store/s4pnfbkalzy5qz57qs6yybna8wylkig6-example.drv' 中...
运行构建后钩子 '/home/grahamc/projects/github.com/NixOS/nix/post-hook.sh'
...
post-build-hook: 正在签名路径 /nix/store/ibcyipq5gf918381dx40mjs0b8w9n18-example
post-build-hook: 正在上传路径 /nix/store/ibcyipq5gf918381dx40mjs0b8w9n18-
↔ example
/nix/store/ibcyipq5gf918381dx40mjs0b8w9n18-example
```

要验证钩子是否生效，请从存储中删除该路径，并尝试从二进制缓存中替换它：

²⁹⁶ <https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-secret-key-files>

²⁹⁷ <https://nix.dev/manual/nix/2.22/command-ref/conf-file#conf-post-build-hook>

```
$ rm ./result
$ nix-store --delete /nix/store/ibcyipq5gf918381dx40mjsp0b8w9n18-example
$ nix-store --realise /nix/store/ibcyipq5gf918381dx40mjsp0b8w9n18-example
正在从's3://复制路径'/nix/store/m8bmqrch613h8s0k3d673xpmjpcdpsa-example
-example-nix-cache'...
警告：未指定'--add-root'；结果可能被垃圾回收...
垃圾回收器
/nix/store/m8bmqrch613h8s0k3d673xpmjpcdpsa-example
```

结论

您已配置Nix，使其自动签名并将每次本地构建上传至远程S3兼容的二进制缓存。在生产环境部署前，请务必考虑实现中的注意事项（第142页）。

3.1.7 使用GitHub Actions进行持续集成

本指南将简要介绍几个步骤，帮助您开始使用 GitHub Actions²⁹⁸ 作为提交和拉取请求的持续集成(CI)工作流。

Nix 的一个优势在于，CI 能通过二进制缓存为每个项目的每个分支构建并缓存开发者环境。

CI 的关键考量是反馈循环——构建完成需要多少分钟？

虽然有几个不错的选择，但 Cachix（如下）是最直接的选择。

使用 Cachix 缓存构建

通过 Cachix²⁹⁹，您无需重复构建派生项，还能与所有开发者共享已构建的派生项。

每次任务结束后，新构建的派生项会被推送到您的二进制缓存中。

每次任务开始前，待构建的派生项会优先从您的二进制缓存中进行替换（若存在）。

1. 创建您的第一个二进制缓存

建议按团队设置不同的二进制缓存，具体取决于谁拥有其读写权限。

填写创建二进制缓存³⁰⁰页面上的表单。

在新建的二进制缓存上，按照“推送二进制文件”标签页的指引操作。

298 <https://github.com/features/actions>

299 <https://cachix.org/>

300 <https://app.cachix.org/cache>

2. 配置密钥

在您的 GitHub 仓库或组织中（适用于所有仓库）：

1. 点击 Settings（设置）。
2. 点击 Secrets（密钥）。
3. 添加您先前生成的密钥（CACHIX_SIGNING_KEY 和/或 CACHIX_AUTH_TOKEN）。

3. 配置 GitHub Actions

创建 .github/workflows/test.yml 内容如下：

```
name: "Test"
on:
  pull_request:
  push:
jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v25
        with:
          nix_path: nixpkgs=channel:nixos-unstable
      - uses: cachix/cachix-action@v14
        with:
          name: mycache
          #若选择使用签名密钥获取写入权限
          #签名密钥: '${{ secrets.CACHIX_SIGNING_KEY }}'
          #若选择API令牌获取写入权限 或 使用私有缓存
          #authToken: '${{ secrets.CACHIX_AUTH_TOKEN }}'
      - run: nix-build
      - run: nix-shell --run "echo OK"
```

提交并推送至GitHub仓库后，您将在提交记录和PR中看到状态检查项。

后续步骤

- 参阅GitHub Actions工作流语法³⁰¹
- 要快速建立Nix项目，请阅读入门指南《Nix模板》³⁰²。

3.2 最佳实践

3.2.1 URL格式

Nix语言语法支持裸URL，因此可直接写https://example.com而非"https://example.com"

RFC 45³⁰³已通过决议弃用未加引号的URL，并列举了该特性弊大于利的多项依据。

³⁰¹ <https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions>

³⁰² <https://github.com/nix-dot-dev/getting-started-nix-template>

³⁰³ <https://github.com/NixOS/rfcs/pull/45>

提示：始终为URL添加引号。

3.2.2 递归属性集 rec { ... }

rec允许你在同一属性集中引用名称。

示例：

```
rec {
  a = 1;
  b = a + 2;
}
```

```
{a = 1; b = 3;}
```

常见陷阱是在遮蔽名称时引发难以调试的无限递归错误。最简单的例子如下：

```
let a = 1 ; in rec {a = a;}
```

提示：避免使用rec。推荐使用 let ... in。

示例：

```
let
  a = 1;
in {
  a = a;
  b = a + 2;
}
```

提示：可通过显式命名属性集实现自引用：

```
let
  argset = {
    a = 1;
    b = argset.a + 2;
  };
in
  argset
```

3.2.3 作用域

以下表达式在实践中仍很常见：

```
with (import <nixpkgs> {});
#... 大量代码
```

这将导入表达式的所有属性引入当前表达式的作用域中。

该方法存在若干问题：

- 静态分析无法推导代码逻辑，因为必须实际执行该文件才能确定作用域内的名称。

- 当使用多个 with 时，名称来源将变得不明确。
- with 的作用域规则不符合直觉，详见此 Nix 议题³⁰⁴。

提示：不要在 Nix 文件顶部使用 with。应在 let 表达式中显式声明名称。

示例：

```
let
  pkgs = import <nixpkgs> {};
  继承 (pkgs) curl jq;
in
#...
```

较小作用域通常问题较少，但仍可能因作用域规则导致意外。

提示：若想完全避免 with，可尝试替换此类表达式

```
buildInputs = with pkgs; [ curl jq ];
```

为以下形式：

```
buildInputs = builtins.attrValues {
  继承 (pkgs) curl jq;
};
```

3.2.4 <...> 查找路径

你会经常遇到引用<nixpkgs>的Nix语言代码示例。

<...>是 2011³⁰⁵ 引入的特殊语法，用于便捷地从环境变量\$NIX_PATH³⁰⁶ 中获取值。

这意味着查找路径的值取决于外部系统状态。使用查找路径时，相同的Nix表达式可能产生不同结果。

多数情况下，安装Nix时\$NIX_PATH会被设置为最新频道，因此不同机器间该值很可能不同。

注：频道³⁰⁷ 是一种引用远程 Nix 表达式并获取其最新版本的机制。

订阅频道的状态独立于依赖它的 Nix 表达式，难以跨机器移植，这可能影响可复现性。

例如，不同机器上的两位开发者可能使 <nixpkgs> 指向 Nixpkgs 仓库的不同版本，导致构建结果因人而异，引发混淆。

提示：使用《实现可复现性：固定 Nixpkgs》（第12页）中的技术显式声明依赖。

除极简示例外，避免使用查找路径。

³⁰⁴ <https://github.com/NixOS/nix/issues/490>

³⁰⁵ <https://github.com/NixOS/nix/commit/1ecc97b6bdb27e56d832ca48cdaf3dbb5185a04>

³⁰⁶ https://nix.dev/manual/nix/stable/command-ref/env-common.html#env-NIX_PATH

³⁰⁷ <https://nix.dev/manual/nix/stable/command-ref/nix-channel.html>

某些工具要求预先设置查找路径。此时：

提示：在版本控制下的中央位置将 \$NIX_PATH 设为已知值。

NixOS

在NixOS系统中，可通过nix.nixPath³⁰⁸ 选项永久设置\$NIX_PATH。

3.2.5 可复现的Nixpkgs配置

为快速获取演示用软件包，我们采用以下简明模式：

```
导入 <nixpkgs> {}
```

然而，即便如《实现可复现性：固定Nixpkgs版本》（第12页）所示替换<nixpkgs>，结果仍可能无法完全复现。这是由于历史原因，Nixpkgs顶层表达式³⁰⁹ 默认会不纯地从文件系统读取配置参数。拥有相应文件的系统可能产生不同结果。

这是一个众所周知的难题，若不破坏现有配置则无法解决。

提示：导入Nixpkgs时显式设置config³¹⁰ 和overlays³¹¹：

```
导入 <nixpkgs> { config = {}; overlays = []; }
```

我们在教程中采用此方式以确保示例行为完全符合预期。为保持简洁性，最小化示例中会省略该设置。

3.2.6 更新嵌套属性集

属性集更新运算符³¹² 用于合并两个属性集。

示例：

```
{a = 1; b = 2;}//{b = 3; c = 4;}
```

```
{a = 1; b = 3; c = 4;}
```

但右侧的命名具有优先级，且更新为浅层操作。

示例：

```
{ a = { b = 1; }; } // { a = { c = 3; }; }
```

```
{ a = { c = 3; }; }
```

此处键b被完全移除，因为整个a值已被替换。

提示：使用pkgs.lib.recursiveUpdate³¹³ Nixpkgs函数：

³⁰⁸ <https://search.nixos.org/options?show=nix.nixPath>

³⁰⁹ <https://github.com/NixOS/nixpkgs/blob/master/default.nix>

³¹⁰ <https://nixos.org/manual/nixpkgs/stable/#chap-packageconfig>

311 <https://nixos.org/manual/nixpkgs/stable/#chap-overlays>312 <https://nix.dev/manual/nix/stable/language/operators.html#update>313 <https://nixos.org/manual/nixpkgs/stable/#function-library-lib.attrssets.recursiveUpdate>

```
let pkgs = import <nixpkgs> {}; in
pkgs.lib.recursiveUpdate { a = { b = 1; }; } { a = { c = 3; }; }
```

```
{a = {b = 1; c = 3; };}
```

3.2.7 可复现的源码路径

```
let pkgs = import <nixpkgs> {}; in
pkgs.stdenv.mkDerivation {
    name = "foo";
    src = ./;
}
```

如果包含此表达式的Nix文件位于/home/myuser/myproject目录下，那么src的存储路径将是/nix/store/<hash>-myproject。

问题在于此时你的构建不再可重现，因为它依赖于父目录名称。这种依赖无法在源代码中声明，从而导致不纯性。

如果其他人在不同命名的目录中构建该项目，他们将获得不同的src存储路径，以及所有依赖它的内容。这可能导致不必要的重新构建。

提示：使用带固定名称属性的builtins.path³¹⁴。

这将从name属性而非工作目录派生存储路径的符号名称：

```
let pkgs = import <nixpkgs> {}; in
pkgs.stdenv.mkDerivation {
    name = "foo";
    src = builtins.path { path = ./; name = "myproject"; };
}
```

3.3 故障排除

本页汇集了解决使用Nix时可能遇到问题的实用技巧。

3.3.1 二进制缓存不可用或无法访问时该如何处理？

向Nix命令传递参数 --option substitute false³¹⁵。

³¹⁴ <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-path>

³¹⁵ <https://nix.dev/manual/nix/stable/command-ref/conf-file#conf-substitute>

3.3.2 如何强制 Nix 重新检查二进制缓存中是否存在某内容？

Nix 会记录二进制缓存中的可用内容，因此无需每次执行命令时都进行查询。这包括否定答案，即当特定存储路径无法被替代时。
通过 `--narinfo-cache-negative-ttl`³¹⁶ 选项设置缓存超时时间（秒）。

3.3.3 如何修复：错误：查询数据库中的路径：数据库磁盘映像已损坏

这是一个已知问题³¹⁷。可尝试：

```
§ sqlite3 /nix/var/nix/db/db.sqlite "pragma integrity_check"
```

这将打印数据库中的错误³¹⁸。若错误由缺失引用引起，以下方法可能奏效：

```
$ mv /nix/var/nix/db/db.sqlite /nix/var/nix/db/db.sqlite-bkp  
§ sqlite3 /nix/var/nix/db/db.sqlite-bkp ".dump" | sqlite3 /nix/var/nix/db/db.sqlite
```

3.3.4 如何修复：错误：当前Nix存储库架构版本为10，但仅支持版本7

此为已知问题³¹⁹。

这意味着新版本Nix升级了数据库的SQLite架构³²⁰，而您随后尝试使用了旧版Nix。

解决方案是转储数据库，并使用旧版Nix重新导入数据：

```
$ /path/to/nix/unstable/bin/nix-store --dump-db > /tmp/db.dump  
$ mv /nix/var/nix/db /nix/var/nix/db.toonew  
§ mkdir /nix/var/nix/db  
$ nix-store --load-db < /tmp/db.dump
```

3.3.5 修复方法：写入文件时出现：Connection reset by peer

这可能意味着您尝试将过大的文件或目录导入Nix存储³²¹，或您的机器资源（如磁盘空间或内存）即将耗尽。

尝试减小待导入目录的大小，或运行垃圾回收³²²。

³¹⁶ <https://nix.dev/manual/nix/stable/command-ref/conf-file.html#conf-narinfo-cache-negative-ttl>

³¹⁷ <https://github.com/NixOS/nix/issues/1353>

³¹⁸ <https://nix.dev/manual/nix/stable/glossary#gloss-nix-database>

³¹⁹ <https://github.com/NixOS/nix/issues/1251>

³²⁰ <https://nix.dev/manual/nix/stable/glossary#gloss-nix-database>

321 <https://nix.dev/manual/nix/stable/glossary#gloss-store>322 <https://nix.dev/manual/nix/stable/command-ref/nix-collect-garbage>

3.3.6 macOS 更新导致 Nix 安装中断

这是已知问题³²³。Nix安装程序³²⁴会修改/etc/zshrc文件。当macOS更新时，系统通常会再次覆盖该文件。

临时解决方案：将以下代码片段添加到/etc/zshrc末尾并重启shell：

```
if [-e '/nix/var/nix/profiles/default/etc/profile.d/nix-daemon.sh']; then
    . '/nix/var/nix/profiles/default/etc/profile.d/nix-daemon.sh'
fi
```

3.4 常见问题解答

3.4.1 Nix

如何自动格式化 Nix 语言代码？

nixfmt³²⁵是Nix语言的官方格式化工具。安装说明请参阅其源代码仓库。

nixfmt用于格式化Nixpkgs中的所有代码³²⁶。

如何在 Nix 语言中实现路径与字符串的相互转换？

参阅 Nix 参考手册中关于字符串插值³²⁷及路径与字符串运算符³²⁸的章节。

如何构建软件包的反向依赖？

```
$ nix-shell -p nixpkgs-review --run "nixpkgs-review wip"
```

如何用Nix管理\$HOME目录下的点文件？

参见 <https://github.com/nix-community/home-manager>

构建自定义软件包的推荐流程是什么？

请阅读《使用Nix打包现有软件》（第41页）。

³²³ <https://github.com/NixOS/nix/issues/3616>

³²⁴ <https://nix.dev/manual/nix/latest/installation/installing-binary>

³²⁵ <https://github.com/NixOS/nixfmt>

³²⁶ <https://github.com/NixOS/nixpkgs/blob/master/ci/default.nix>

³²⁷ <https://nix.dev/manual/nix/2.19/language/string-interpolation>

³²⁸ <https://nix.dev/manual/nix/2.19/language/operators#string-concatenation>

如何使用Nixpkgs仓库的克隆来更新或编写新包？

请阅读《用Nix打包现有软件》（第41页）及Nixpkgs贡献指南³²⁹。

3.4.2 NixOS

如何运行非Nix可执行文件？

NixOS无法直接运行为通用Linux环境设计的动态链接可执行文件。这是由于其设计上既没有全局库路径，也不遵循文件系统层次标准³³⁰ (FHS)。

有几种方法可以解决这种环境预期不匹配的问题：

- 若Nixpkgs中已有打包版本，优先使用。可通过<https://search.nixos.org/packages>搜索可用软件包。
- 为程序编写Nix表达式，将其打包至个人配置中。

存在多种实现方式：

- 从源码构建。

许多开源程序在编译时可灵活配置文件安装路径。相关入门指南参见《使用Nix打包现有软件》（第41页）。

- 修改程序的ELF头³³¹，通过autoPatchelfHook³³²添加库文件路径。

若从源码构建不可行，则采用此方案。

- 使用buildFHSEnv³³³将程序包装在类FHS环境中运行。

这是最终手段，但有时不可避免，例如当程序需下载并运行其他可执行文件时。

- 通过nix-ld³³⁴创建仅适用于未打包程序的库路径。将此添加至configuration.nix：

```
programs.nix-ld.enable = true;
programs.nix-ld.libraries = with pkgs; [
    #为未打包程序添加缺失的动态链接库
    #在此处添加，而非environment.systemPackages中
];
```

随后运行nixos-rebuild switch，并重新登录以使新环境变量生效。（仅首次启用nix-ld时需要此操作；所含库的变更在重建后立即生效。）

注意：nix-ld在x86_64机器上不兼容32位可执行文件。

- 使用steam-run³³⁵在专为Steam包打造的类FHS环境中运行程序：

```
$ nix-shell -p steam-run --run "steam-run <命令>"
```

³²⁹ <https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md>

³³⁰ https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html

³³¹ https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

³³² <https://nixos.org/manual/nixpkgs/stable/#setup-hook-autopatchelfhook>

³³³ <https://nixos.org/manual/nixpkgs/stable/#sec-fhs-environments>

³³⁴ <https://github.com/Mic92/nix-ld>

³³⁵ <https://nixos.org/manual/nixpkgs/stable/#sec-steam-run>

如何构建自己的ISO?

参见 <http://nixos.org/nixos/manual/index.html#sec-building-image>

如何连接到NixOS测试中的任意机器?

应用以下补丁:

```
diff --git a/nixos/lib/test-driver/test-driver.pl b/nixos/lib/test-driver/test-
→ driver.pl
索引 8ad0d67..838fbdd 100644
--- a/nixos/lib/test-driver/test-driver.pl
+++ b/nixos/lib/test-driver/test-driver.pl
Q@ -34,7 +34,7 && 对每个我的 $vlan (分割 //, $环境变量{VLANS} || "") {
    如果 ($pid == 0) {
        复制文件描述符 (文件句柄($pty->从端), 0);
        dup2 (fileno($stdoutW), 1);
        exec "vde_switch -s $socket" or _exit(1);
+       exec "vde_switch -tap tap0 -s $socket" or _exit(1);
    }
    close $stdoutW;
    print $pty "version\n";
```

随后应能在本地访问vde_switch网络。

如何在现有 Linux 系统中引导安装 NixOS?

有以下几种工具:

- <https://github.com/nix-community/nixos-anywhere>
- <https://github.com/jeaye/nixos-in-place>
- <https://github.com/elitak/nixos-infect>
- <https://github.com/cleverca22/nix-tests/tree/master/kexec>

REFERENCE

这些章节包含详细技术说明的集合。

4.1 术语表

Nix

构建系统与包管理器。

读作 /nrks/ ("Niks")。

另请参阅：

- Nix 参考手册 (第156页)
- Nix 源代码³³⁶

Nix 语言

用于声明 Nix 软件包与配置的编程语言。

另请参阅：

- Nix 语言基础 (第13页)
- Nix 语言参考³³⁷

Nix 表达式

用 Nix 语言编写的表达式。

Nix 文件

包含 Nix 表达式的 .nix 文件。

Nixpkgs

使用Nix构建的软件发行版。

读作/nrks 'pækɪdʒɪz/ ("Nix包")。

另请参阅：

- Nixpkgs 参考手册³³⁸
- Nixpkgs 源代码³³⁹

NixOS

基于Nix和Nixpkgs的Linux发行版。

读作/nɪks oʊs/ ("Niks Oh Es")。

另见：

³³⁶ <https://github.com/NixOS/nix>

³³⁷ <https://nix.dev/manual/nix/stable/language>

³³⁸ <https://nixos.org/manual/nixpkgs>

³³⁹ <https://github.com/NixOS/nixpkgs>

- NixOS 参考手册³⁴⁰
- NixOS 源代码³⁴¹

4.2 Nix 参考手册

Nix 参考手册提供多版本查阅：

- Nix 预发布版³⁴²
 - Nix 代码库主分支的开发构建版³⁴³
 - Nix 2.32³⁴⁴ (单页版³⁴⁵)
 - 最新 Nix 版本
 - Nix 2.31³⁴⁶ (单页版³⁴⁷)
 - 随 Nixpkgs 与 NixOS 的滚动更新版本发布
 - Nix 2.28³⁴⁸ (单页版³⁴⁹)
 - 随当前 Nixpkgs 与 NixOS 稳定版发布：25.05
 - Nix 2.24³⁵⁰ (单页版³⁵¹)
 - 随 Nixpkgs 与 NixOS 上一稳定版本 24.11 发布

提示：关于 Nixpkgs 和 NixOS 版本的更多信息：我该使用哪个通道分支？（第164页）

4.3 扩展阅读

4.3.1 Nix 语言教程

- Nix 语言速查手册³⁵² (Vincent Ambo, 2019-2021)
 - 通用语言特性与常用惯用法的概览。
- Nix 语言导览³⁵³ (Joachim Schiele, 2015-2022)
 - Nix 语言的交互式练习。
 - 视频：Nix 语言概览³⁵⁴ (Wil Taylor, 2021)
 - 语言特性概览。

³⁴⁰ <https://nixos.org/manual/nixos>

³⁴¹ <https://github.com/NixOS/nixpkgs/tree/master/nixos>

³⁴² <https://nix.dev/manual/nix/development/>

³⁴³ <https://github.com/NixOS/nix>

³⁴⁴ <https://nix.dev/manual/nix/latest/>

³⁴⁵ <https://nix.dev/manual/nix/latest/nix-2.32.html>

³⁴⁶ <https://nix.dev/manual/nix/rolling/>

³⁴⁷ <https://nix.dev/manual/nix/rolling/nix-2.31.html>

³⁴⁸ <https://nix.dev/manual/nix/stable/>

³⁴⁹ <https://nix.dev/manual/nix/stable/nix-2.28.html>

³⁵⁰ <https://nix.dev/manual/nix/prev-stable/>

³⁵¹ <https://nix.dev/manual/nix/prev-stable/nix-2.24.html>

³⁵² <https://github.com/tazjin/nix-1p>

³⁵³ <https://nixcloud.io/tour>

³⁵⁴ https://www.youtube.com/watch?v=eCapIx9heBw&list=PL-saUBvIJzOkjAw_vOac75v-x6EzNzZq-&index=5

- 视频：Nix语言阅读指南³⁵⁵ (Jonas Chevalier, 2019)
 - Nix语言代码阅读入门
- 视频：工作原理及缘由³⁵⁶ (Graham Christensen, 2019)
 - 派生编写入门

4.3.2 其他文章

- Nix Pills³⁵⁷
 - 一份关于使用Nix构建软件包的初级教程，详细展示Nixpkgs的构建过程。
- Nix中的软件包定制方法³⁵⁸ (2022)
 - 概述定制Nix软件包的不同方法。
- 使用Home Manager管理点文件³⁵⁹ (Mattia Gheda, 2021)
 - 一份Home Manager入门教程。
- Nix短篇指南³⁶⁰
 - 一系列关于Nix打包基础原理的帖子。
- NixOS与Flakes - 面向初学者的非官方书籍³⁶¹ (2023)
 - 本教程是通过实验性Nix Flakes功能（第161页）对NixOS的入门介绍。

4.3.3 其他视频

- Nixology³⁶² (Burke Libbey, 2020)
 - 介绍Nix基础概念的系列视频。
- Nix时刻³⁶³ (Silvan Mosberger, 2022年起)
 - 每周探讨Nix生态各类话题并解答问题的系列节目。
- Nixpkgs³⁶⁴ (Jon Ringer, 2020-22年)
 - 围绕Nixpkgs各项操作提供教程的视频系列。
- NixOS³⁶⁵ (Wil Taylor, 2021年)
 - NixOS入门教程系列。
- NixOS 基金会 YouTube 频道³⁶⁶
 - NixOS 基金会官方频道。
- NixCon 大会 YouTube 频道³⁶⁷
 - NixCon 演讲与讨论实录。

355 <https://youtu.be/hbJkMl631FE?t=1533>

356 <https://youtu.be/hbJkMl631FE?t=4806>

357 <https://nixos.org/nixos/nix-pills/index.html>

358 <https://bobvanderlinden.me/customizing-packages-in-nix/>

359 <https://ghedam.at/24353/tutorial-getting-started-with-home-manager-for-nix>

360 <https://github.com/justinwoo/nix-shorts>

361 <https://nixos-and-flakes.thiscute.world>

362 https://www.youtube.com/playlist?list=PLRG19KQ3_HPOFRG6R-p4iFgMSK1t5BHs

363 <https://www.youtube.com/playlist?list=PLyzwHTVJIRc8yjlx4VR4LU5A5044og9in>

364 <https://www.youtube.com/@jonringer117/videos>

365 https://www.youtube.com/playlist?list=PL-saUBvIJzOkjAw_vOac75v-x6EzNzZq-

366 <https://www.youtube.com/@NixOS-Foundation/playlists>

367 <https://www.youtube.com/@NixCon>

4.4 固定 Nixpkgs 版本

指定远程 Nix 表达式（例如 Nixpkgs 提供的表达式）可通过多种方式实现：

- [\\$NIX_PATH 环境变量](#)³⁶⁸
- 向 nix-build、nix-shell 等大多数命令传递 -I 选项³⁶⁹
- 在 Nix 表达式中使用 fetchurl³⁷⁰、fetchTarball³⁷¹、fetchGit³⁷² 或 Nixpkgs 抓取器³⁷³

4.4.1 可用的 URL 值

- 本地文件路径：

```
./path/to/expression.nix
```

使用 ./ 表示表达式位于当前目录的 default.nix 文件中。

- 锁定到特定提交：

```
https://github.com/NixOS/nixpkgs/archive/  
-eabc38219184cc3e04a974fe31857d8e0eac098d.tar.gz
```

- 使用最新通道版本（意味着所有测试均已通过）：

```
http://nixos.org/channels/nixos-22.11/nixexprs.tar.xz
```

- 通道简写语法：

```
channel:nixos-22.11
```

- 使用 GitHub 托管的最新通道版本：

```
https://github.com/NixOS/nixpkgs/archive/nixos-22.11.tar.gz
```

- 使用发布分支的最新提交（尚未测试）：

```
https://github.com/NixOS/nixpkgs/archive/release-21.11.tar.gz
```

4.4.2 示例

```
§ nix-build -I ~/dev
```

- `$ nix-build -I nixpkgs=http://nixos.org/channels/nixos-22.11/nixexprs.tar.xz`

```
$ nix-build -I nixpkgs=channel:nixos-22.11
```

- `$ NIX_PATH=nixpkgs=http://nixos.org/channels/nixos-22.11/nixexprs.tar.xz nix-build`

- `$ NIX_PATH=nixpkgs=channel:nixos-22.11 nix-build`

³⁶⁸ https://nix.dev/manual/nix/stable/command-ref/env-common.html#env-NIX_PATH

³⁶⁹ <https://nix.dev/manual/nix/stable/command-ref/opt-common.html#opt-I>

³⁷⁰ <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchurl>

³⁷¹ <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchTarball>

³⁷² <https://nix.dev/manual/nix/stable/language/builtins.html#builtins-fetchGit>

³⁷³ <https://nixos.org/manual/nixpkgs/stable/#chap-pkgs-fetchers>

- 在Nix语言中：

```
let
pkgs = import (fetchTarball "https://github.com/NixOS/nixpkgs/archive/nixos-
-22.11.tar.gz") {};
in pkgs.stdenv.mkDerivation { ... }
```

4.4.3 查找特定提交与版本

status.nixos.org³⁷⁴ 提供：

- 各版本最新测试通过的提交——适用于固定到特定提交时
- 活跃发布通道列表——适用于追踪通道最新版本时

完整通道列表详见 nixos.org/channels³⁷⁵。

提示：更多关于 Nixpkgs 和 NixOS 版本的信息：我该使用哪个通道分支？（第164页）

³⁷⁴ <https://status.nixos.org/>

³⁷⁵ <https://nixos.org/channels>

CONCEPTS

这些章节阐述了Nix生态中的历史背景与核心理念。

5.1 Flakes

通常所说的"flakes"是指：

- 一套管理Nix表达式间依赖关系的策略。
- Nix中的实验性功能³⁷⁶，用于实现该策略并提供支持功能。

5.1.1 什么是flakes？

从技术上讲，flake³⁷⁷是一个文件系统树，其根目录下包含名为 flake.nix 的文件。

Flakes 为 Nix 添加了以下行为：

1. flake.nix 文件强制遵循模式³⁷⁸，其中：
 - 其他 flakes 被引用为提供 Nix 语言代码或其他文件的依赖项。
 - flake.nix 中 Nix 表达式生成的值根据预定义用例进行结构化。
2. 可使用专用类 URL 语法³⁷⁹ 指定对其他 flakes 的引用。flake 注册表³⁸⁰ 允许使用符号标识符进一步简化。引用可自动锁定到当前特定版本，并后续通过编程方式更新。
3. 作为独立实验性功能实现的新命令行界面³⁸¹，通过接受flake引用来构建、运行或部署定义为flake的软件。

Nix处理flake与常规Nix文件有以下差异：

- flake.nix文件会进行模式有效性检查。

特别说明：元数据字段不能是任意Nix表达式。这是为了防止查询元数据时出现复杂且可能无法终止的计算。

- 整个flake目录在评估前会被复制到Nix存储区。

这能实现有效的评估缓存（对Nixpkgs等大型表达式尤为重要），但也需要在每次变更时重新复制整个flake目录。

³⁷⁶ <https://nix.dev/manual/nix/stable/contributing/experimental-features>

³⁷⁷ <https://nix.dev/manual/nix/stable/Command-ref/new-cli/nix3-flake.html#description>

³⁷⁸ <https://nix.dev/manual/nix/stable/Command-ref/new-cli/nix3-flake.html#flake-format>

³⁷⁹ <https://nix.dev/manual/nix/stable/Command-ref/new-cli/nix3-flake.html#flake-references>

³⁸⁰ <https://nix.dev/manual/nix/stable/Command-ref/new-cli/nix3-registry.html>

³⁸¹ <https://nix.dev/manual/nix/stable/Command-ref/new-cli/nix.html>

- 不允许使用外部变量、参数或不纯的语言值。

这意味着Nix表达式默认具备完全可重现性，进而构建指令亦然，但也禁止使用者通过参数化定制结果。

5.1.2 Flakes为何存在争议？

Flakes（第161页）灵感源自Shea Levy在NixCon 2018的演讲³⁸²，经RFC 49³⁸³正式提案，自201年起持续开发。Nix于2021年将其作为首个实验性功能³⁸⁴引入。

该主题在Nix用户和开发者中因设计、实现质量及决策流程而存在争议，具体表现为：

- RFC未达成结论即被关闭，部分核心问题悬而未决。例如：
 - 全局flake注册中心³⁸⁵的概念遭到大量未解决的批评³⁸⁶。虽然注册条目可固定源码引用³⁸⁷，但Nix表达式中的本地注册名会引入可变系统状态³⁸⁸，因此在这方面与nix-channel³⁸⁹管理的通道相比并无改进。
 - 无法对flakes进行参数化³⁹⁰。这意味着对于生产者和消费者而言，flakes降低了派生系统参数³⁹¹的易用性。
 - flakes提案因试图一次性解决过多问题³⁹²且选错抽象层³⁹³而受到批评。部分原因在于新命令行界面与flakes紧密耦合³⁹⁴。
- 正如RFC评审者³⁹⁵所预言的，原始实现在Nix 2.4版本³⁹⁷中引入了回归问题³⁹⁶，导致部分稳定功能在未进行主版本号³⁹⁸升级的情况下失效。
- 在评估前将源代码复制到Nix存储库会带来显著的性能损耗³⁹⁹，对于Nixpkgs等大型代码库尤为明显。自2022年5月⁴⁰⁰起已着手解决该问题，但可能引发新的系列问题⁴⁰¹。
- 至今仍有不少人在引导新Nix用户采用flakes功能，尽管该实验既无稳定性保证，也无明确结束时间表。

这导致稳定接口多年来仅得到零星维护，并因持续开发而反复出现故障。与此同时，新接口的广泛采用使得在不影响用户的情况下持续演进设计变得极其困难。

根据2023⁴⁰²调查，84%的受访者依赖实验性功能。作为对比案例的Nixpkgs，虽然提供了flake.nix以实现兼容性，但其代码库并不依赖Nix实验性功能。

382 <https://www.youtube.com/watch?v=DHOLjsyXPtM>

383 <https://github.com/NixOS/rfcs/pull/49>

384 <https://nix.dev/manual/nix/stable/contributing/experimental-features>

385 <https://github.com/NixOS/flake-registry>

386 <https://github.com/NixOS/rfcs/pull/49#issuecomment-635635333>

387 <https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-registry-pin>

388 <https://github.com/NixOS/nix/issues/7422>

389 <https://nix.dev/manual/nix/stable/command-ref/nix-channel>

390 <https://github.com/NixOS/nix/issues/2861>

391 <https://github.com/NixOS/nix/issues/3843>

392 <https://github.com/nixos/rfcs/pull/49#issuecomment-521998933>

393 <https://discourse.nixos.org/t/nixpkgs-clients-working-group-member-search/30517>

394 <https://discourse.nixos.org/t/2023-03-06-nix-team-meeting-minutes-38/26056#cli-stabilisation-announcement-draft-4>

395 <https://github.com/NixOS/rfcs/pull/49#issuecomment-588990425>

396 <https://discourse.nixos.org/t/nix-2-4-and-what-s-next/16257>

397 <https://nix.dev/manual/nix/stable/release-notes/r1-2.4.html>

398 <https://semwer.org/>

399 <https://github.com/NixOS/nix/issues/3121>

400 <https://github.com/NixOS/nix/pull/6530>

401 <https://github.com/NixOS/nix/pull/6530#issuecomment-1850565931>

402 <https://discourse.nixos.org/t/nix-community-survey-2023-results/33124>

5.1.3 我应该在项目中使用 Flakes 吗？

您需要根据自身需求做出判断。

Flakes（第161页）强调产物的可重现性及使用者的便利性，而经典 Nix 工具则聚焦开发者的模块化构建与定制选项。两种范式各有独特的理念体系和支持工具，其学习曲线、实现质量和支持状态各不相同。目前无论是稳定版还是实验版接口，都未在所有方面展现出绝对优势。

Flakes 和 nix 命令套件为软件用户和包作者带来了多项改进：

- 新命令行接口配合 Flakes 能在多数场景显著提升现有软件包的使用便利性
- Flakes 的默认约束机制增强了可重现性，并在与大型代码库（如 Nixpkgs）交互时实现了性能优化
- Flake引用机制使得现有软件包或项目依赖的版本升级更易处理。
- Flake模式⁴⁰³有助于从多源头有序组合Nix项目。

与此同时，Flake存在根本性架构缺陷（第162页）及若干实现问题⁴⁰⁴，且缺乏系统性解决的协同努力。关于Nix命令行界面⁴⁰⁵仍存在许多未决设计问题，其中部分正在推进解决。

尽管Flake在某些方面降低了复杂度，但其引入的额外机制也带来了新的复杂性。必须深入理解系统才能完全掌握其运作方式。

除此之外，在Flake模式表层之下，两种场景中Nix及Nix语言的运作机制完全相同。无论是否使用Flake，原则上均可实现同等程度的可复现性。特别是向Nixpkgs添加软件或维护NixOS模块及配置的过程完全不受Flake影响。目前也无证据表明Flake能帮助解决二者的可扩展性挑战。

总体而言，依赖实验性功能⁴⁰⁶存在以下弊端：

- Nix开发者可能随时变更实验性功能的接口与行为。这要求您未来必须调整代码，而随着复杂度增加，调整成本将更高。目前稳定flakes功能尚无具体时间表⁴⁰⁷。相比之下，Nix的稳定功能可视为永久稳定。
- Nix维护团队⁴⁰⁸优先修复稳定接口的缺陷与回归问题，支持成熟用例，并优化内部设计及贡献者体验以促进未来发展。实验性功能的改进优先级较低。
- Nix文档团队⁴⁰⁹专注于完善稳定功能及通用原理的文档与学习资料。使用flakes时，您将更依赖用户间互助、第三方文档及源代码。

⁴⁰³ <https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake.html#flake-format>

⁴⁰⁴ <https://github.com/NixOS/nix/issues?q=is%3Aissue+is%3Aopen+label%3Aflakes+sort%3Areactions-%2B1-desc>

⁴⁰⁵ <https://github.com/NixOS/nix/issues?q=is%3Aissue+is%3Aopen+label%3Anew-cli+sort%3Areactions-%2B1-desc>

⁴⁰⁶ <https://nix.dev/manual/nix/stable/contributing/experimental-features>

⁴⁰⁷ <https://discourse.nixos.org/t/stabilising-the-new-nix-command-line-interface/35531#p-123372-how-does-this-relate-to-flakes-3>

⁴⁰⁸ <https://nixos.org/community/teams/nix.html>

⁴⁰⁹ <https://nixos.org/community/teams/documentation.html>

5.1.4 延伸阅读

- Flakes并非真实存在且不会伤害你：以非flake方式使用Nix flakes指南⁴¹⁰ (Jade Lovelace, 2024年1月)
- Nix Flakes是一次尝试过多功能的实验...⁴¹¹ (评论⁴¹²) (Samuel Dionne-Riel, 2023年9月)
- 实验性并不意味着不稳定⁴¹³ (评论⁴¹⁴) (Graham Christensen, 2023年9月)
- Nix时刻：flakes与传统Nix的对比⁴¹⁵ (Silvan Mosberger, 2022年11月)

5.2 常见问题

5.2.1 Nix 名称的起源是什么？

Nix 名称源自荷兰语单词 niks，意为“空无”；构建操作不会感知任何未显式声明为输入的内容。

——《Nix：一种安全无策略的软件部署系统》⁴¹⁶，LISA XVIII, 2004

Nix 徽标的灵感来自 Haskell 徽标的创意⁴¹⁷，且 nix 在拉丁语中意为“雪”⁴¹⁸。

5.2.2 什么是 flakes？

参见 Flakes (第161页)。

5.2.3 我应该使用哪个通道分支？

Nixpkgs 和 NixOS 同时提供稳定版和滚动更新版。

这些版本通过称为“通道分支”的变体分发：即用于发布的 Git 分支，它们也会被转换为 Nix 通道。

提示：查阅 Nix 参考手册中的 nix-channel⁴¹⁹ 条目获取更多通道信息，以及 Nixpkgs 贡献指南⁴²⁰ 了解分支策略。

410 <https://jade.fyi/blog/flakes-arent-real/>

411 <https://samuel.dionne-riel.com/blog/2023/09/06/flakes-is-an-experiment-that-did-too-much-at-once.html>

412 <https://discourse.nixos.org/t/nix-flakes-is-an-experiment-that-did-too-much-at-once/32707>

413 <https://determinate.systems/posts/experimental-does-not-mean-unstable>

414 <https://discourse.nixos.org/t/experimental-does-not-mean-unstable-detsyss-perspective-on-nix-flakes/32703>

415 <https://www.youtube.com/watch?v=atmoYyBAhF4>

416 <https://edolstra.github.io/pubs/nsfssd-lisa2004-final.pdf>

417 <https://wiki.haskell.org/File:Sgf-logo-blue.png>

418 <https://nix-dev.science.uu.narkive.com/VDaaP1BY/nix-logo>

419 <https://nix.dev/manual/nix/2.22/command-ref/nix-channel>

420 <https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md#branch-conventions>

稳定版

稳定版仅接收保守更新以修复错误或安全漏洞；其他情况下软件包版本保持不变。每六个月发布一个新的稳定版。

- 在 Linux（包括 NixOS 和 WSL）上，使用 nixos-*⁴²¹

这些分支指向已预构建大多数 Linux 软件包的提交，且可从二进制缓存获取。此外，这些提交已通过完整的 NixOS 测试套件。

- 在 macOS/Darwin 上，使用 nixpkgs-*darwin⁴²²

这些分支指向已预构建大多数 Darwin 软件包的提交，且可从二进制缓存获取。

- 在其他平台上，使用上述哪一种都无关紧要。

Hydra 不会为其他平台预构建任何二进制文件。

所有这些“频道分支”都遵循相应的 release-*⁴²³ 分支。

示例

nixos-23.05 和 nixpkgs-23.05-darwin 都基于 release-23.05。

滚动更新

滚动发布遵循主开发分支 master⁴²⁴。

- 在 Linux（含 NixOS 和 WSL）上使用 nixos-unstable⁴²⁵。

- 其他平台使用 nixpkgs-unstable⁴²⁶。

*-small⁴²⁷ 通道分支已通过精简测试套件，这意味着它们与基础分支同步更快，但稳定性保障较低。

5.2.4 沙盒构建中是否仍存在杂质？

是的。包括：

- CPU 架构——正全力避免编译原生指令，优先采用硬编码支持的指令集。

- 系统当前时间/日期。

- 用于构建的文件系统（另见 TMPD IR⁴²⁸）。

- Linux 内核参数，例如：

- IPv6 支持能力⁴²⁹。

- binfmt 解释器，例如通过 boot.binfmt.emulatedSystems⁴³⁰ 配置的实例。

- 构建系统的时间行为问题——在某些情况下，并行 Make 构建可能无法获取正确的输入。

⁴²¹ <https://github.com/NixOS/nixpkgs/branches/all?query=nixos>

⁴²² <https://github.com/NixOS/nixpkgs/branches/all?query=nixpkgs>

⁴²³ <https://github.com/NixOS/nixpkgs/branches/all?query=release->

⁴²⁴ <https://github.com/NixOS/nixpkgs/branches/all?query=master>

⁴²⁵ <https://github.com/NixOS/nixpkgs/branches/all?query=nixos-unstable>

⁴²⁶ <https://github.com/NixOS/nixpkgs/branches/all?query=nixpkgs-unstable>

⁴²⁷ <https://github.com/NixOS/nixpkgs/branches/all?query=-small>

⁴²⁸ <https://nix.dev/manual/nix/stable/command-ref/env-common.html#env-TMPDIR>

⁴²⁹ <https://github.com/NixOS/nix/issues/5615>

⁴³⁰ <https://search.nixos.org/options?show=boot.binfmt.emulatedSystems>

- 插入随机值，例如来自 /dev/random 或 /dev/urandom 的数据。
- Nix版本间的差异。例如，新版本可能引入新的环境变量。像 `env > $ out` 这样的语句，Nix不保证未来会产生相同的输出。

贡献指南

6.1 如何参与贡献

Nix生态系统由众多志愿者和少数受薪开发者共同构建，维护着全球规模最大的开源软件发行版之一。若没有您的支持，保持其正常运转、持续更新并不断改进将无从实现！

本指南介绍如何为Nix、Nixpkgs或NixOS贡献力量。假设您已初步掌握基础概念和工作流程（入门教程系列第3页所述），重点包括Nix语言（第13页）、构建软件衍生的Nixpkgs机制（第41页）、模块系统（第73页）以及NixOS集成测试（第107页）。

重要提示：若无法投入时间，请考虑通过Open Collective向NixOS基金会捐款⁴³¹。当前重点资助线下活动⁴³²以分享知识、壮大Nix开发者社区。若有充足预算，我们将能支付关键基础设施和代码的持续维护工作——这些繁重任务难以永远依赖志愿者完成。

6.1.1 入门指南

在阅读参考文档（第155页）和关注的相关代码后，从提出有依据的问题开始。

加入我们的社区交流平台⁴³³与其他用户和开发者建立联系。若对特定主题感兴趣，可探索并考虑参与我们的社区团队⁴³⁴。

所有源代码和文档均托管于GitHub⁴³⁵，提交变更需拥有GitHub账号。技术讨论通常在议题和拉取请求的评论区进行。

提示：若初次接触Nix，建议优先贡献文档（第171页）。

这里是我们最需要帮助且最适合新手起步的领域。

文档和贡献指南往往不完整或过时，尽管我们并不希望如此。我们正在努力改进。您可以通过在遇到贡献流程问题时立即解决它们，来帮助改善所有人的处境。这可能会延缓您处理原本关注的问题，但将使未来每个人都能更轻松地做出有意义的贡献，并最终带来更好的代码和文档。

⁴³¹ <https://opencollective.com/nixos>

⁴³² <https://github.com/NixOS/foundation/issues?q=is%3Aissue%20label%3Afunding-request%20>

⁴³³ <https://nixos.org/community>

⁴³⁴ <https://nixos.org/community/#governance-teams>

⁴³⁵ <https://github.com/NixOS>

6.1.2 问题反馈

注：关于代码的通用问题或操作咨询，请使用我们的社区交流平台⁴³⁶

技术问题及解决方案提议请提交GitHub工单，问题解决或失效后及时关闭。

唯有知晓的问题方能修复，请务必上报您遇到的任何问题。

- Nix相关问题（含Nix参考手册⁴³⁷）请提交至<https://github.com/NixOS/nix/issues>
- Nixpkgs或NixOS相关问题（含软件包、配置模块、Nixpkgs手册⁴³⁸及NixOS手册⁴³⁹）请提交至<https://github.com/NixOS/nixpkgs/issues>

请确认您的问题尚未有未解决的议题。请遵循议题模板并填写所有要求的信息。

特别注意提供一个最小化、易于理解的示例来复现您遇到的问题。您还应展示自己尝试解决问题时的发现。这能极大提高问题最终被解决的可能性，且至关重要：

- 可复现的示例简洁且明确。

这有助于分类议题、理解问题、定位根本原因及制定解决方案。您的初步研究还能进一步协助维护者分析。

- 它让任何人能判断该议题是否仍有意义。

议题可能长期未被处理。即使数月或数年后，决定如何处理它们仍需检查根本问题是否仍存在或已解决。这一过程必须简单：这样任何人都能协助分类，并通知维护者关闭或重新调整议题优先级。

- 该样本可用于解决问题时的回归测试。

提示：理想情况下您还应提出或草拟解决方案。完美的问题报告实际上应是一个直接解决问题的拉取请求，并通过测试确保问题不会再次出现。

重要说明：请仅在您愿意且能够自行实现的情况下，提交新功能需求（如软件包、模块、命令等）的问题报告。此类问题可用于评估用户需求、判断功能是否适合项目，并讨论实现策略。

6.1.3 为Nix作贡献

Nix是生态系统的基石，主要采用 C++ 编写。

如需参与开发，请查阅GitHub上Nix仓库的贡献指南⁴⁴⁰。

436 <https://nixos.org/community>

437 <https://nix.dev/manual/nix/stable>

438 <https://nixos.org/manual/nixpkgs/stable>

439 <https://nixos.org/manual/nixos/stable>

440 <https://github.com/NixOS/nix/blob/master/CONTRIBUTING.md>

6.1.4 为 Nixpkgs 做贡献

提示：观看 NixCon 2024 演讲《成为 Nixpkgs 贡献者》⁴⁴¹ 获取入门讲解。

Nixpkgs 是涵盖多个开发领域的大型软件项目，您可以在 Nixpkgs 问题跟踪器⁴⁴² 中找到改进灵感。

若想参与贡献，请先阅读 GitHub 上 Nixpkgs 仓库的贡献指南⁴⁴³ 了解代码结构与贡献流程，另有针对不同编程语言的包添加说明⁴⁴⁴。

6.1.5 为 NixOS 做贡献

NixOS 是通过声明式编程接口实现高度灵活配置的集体开发 Linux 发行版，其模块代码与默认配置位于 nixpkgs GitHub 仓库的 nixos 目录⁴⁴⁵。

参阅 NixOS 手册的开发章节⁴⁴⁶ 开始参与改进。虽然目前缺乏针对 NixOS 的贡献者文档，但大多数 Nixpkgs 贡献规范（第169页）同样适用。协助完善文档将备受赞赏。

新贡献者可查阅标记为 good-first-bug⁴⁴⁷ 的议题。若已熟悉流程，解决热门议题⁴⁴⁸ 将获得 NixOS 用户的高度认可。

6.2 获取帮助途径

若已准备拉取请求但需进一步协助，请查阅 contributing-how-to-get-help⁴⁴⁹ 获取更多信息。

6.3 获取帮助方式

若您在贡献过程中需要协助，可通过以下渠道获取帮助。

⁴⁴¹ <https://www.youtube.com/watch?v=ejjTOBBbCv4>

⁴⁴² <https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+is%3Aissue+-label%3A%226.topic%3A+nixos%22+-label%3A%226.topic%3A+module+system%22+-label%3A%226.+topic%3A+nixos-container%22+sort%3Areactions-%2B1-desc>

⁴⁴³ <https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md>

⁴⁴⁴ <https://nixos.org/manual/nixpkgs/unstable/#chap-language-support>

⁴⁴⁵ <https://github.com/NixOS/nixpkgs/tree/master/nixos>

⁴⁴⁶ <https://nixos.org/manual/nixos/stable/index.html#ch-development>

⁴⁴⁷ <https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+label%3A%223.skill%3A+good-first-bug%22+label%3A%226.topic%3A+nixos%22>

⁴⁴⁸ <https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+is%3Aissue+label%3A%226.topic%3A+nixos%22+sort%3Areactions-%2B1-desc>

⁴⁴⁹ <https://nix.dev/contributing/how-to-get-help#contributing-how-to-get-help>

6.3.1 如何寻找维护者

为提高效率与成功率，您应优先联系掌握更具体知识的个人或团队：

- 若您的贡献针对Nixpkgs中的软件包，请在maintainers⁴⁵⁰ 属性中查找其维护者。
- 检查是否有团队负责相关子系统：
 - NixOS官网⁴⁵¹ 上。
 - Nixpkgs维护团队列表⁴⁵² 中。
 - 在 Nixpkgs⁴⁵³ 或 Nix⁴⁵⁴ 的 CODEOWNERS 文件中。
- 对需要帮助的文件执行 git blame⁴⁵⁵ 或 git log⁴⁵⁶ 并查看输出。记录提交过相关代码的人员邮箱地址。

6.3.2 应使用的沟通渠道

找到目标人员后，可通过以下任一社区沟通平台⁴⁵⁷ 联系他们：

- GitHub⁴⁵⁸
所有源代码均托管于 GitHub。此处是讨论实现细节的理想平台。
在议题评论或拉取请求描述中，提及maintainers-list.nix文件⁴⁶⁰ 中列出的GitHub用户名⁴⁵⁹。
- Discourse⁴⁶¹
Discourse用于公告发布、协作讨论及开放式提问。
尝试使用maintainers-list.nix文件⁴⁶² 中的GitHub用户名来提及或直接联系特定用户。请注意，部分用户在Discourse上的用户名可能不同。
- Matrix⁴⁶³
Matrix适用于即时短对话及私信交流。
如需联系维护者，请使用 maintainers-list.nix 文件中列出的 Matrix 账号⁴⁶⁴。若某位维护者未提供 Matrix 账号，可尝试搜索其 GitHub 用户名，多数人会在不同平台使用相同名称。
维护者团队通常拥有专属的公开 Matrix 聊天室。
- 电子邮件
使用 git log 查询到的电子邮件地址。

⁴⁵⁰ <https://nixos.org/manual/nixpkgs/stable/#var-meta-maintainers>

⁴⁵¹ <https://nixos.org/community/#governance-teams>

⁴⁵² <https://github.com/NixOS/nixpkgs/blob/master/maintainers/team-list.nix>

⁴⁵³ <https://github.com/NixOS/nixpkgs/blob/master/ci/OWNERS>

⁴⁵⁴ <https://github.com/NixOS/nix/blob/master/.github/CODEOWNERS>

⁴⁵⁵ <https://git-scm.com/docs/git-blame>

⁴⁵⁶ <https://www.git-scm.com/docs/git-log>

⁴⁵⁷ <https://nixos.org/community>

⁴⁵⁸ <https://github.com/nixos>

⁴⁵⁹ <https://docs.github.com/zh/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax#mentioning-people-and-teams>

⁴⁶⁰ <https://github.com/NixOS/nixpkgs/blob/master/maintainers/maintainer-list.nix>

⁴⁶¹ <https://discourse.nixos.org>

⁴⁶² <https://github.com/NixOS/nixpkgs/blob/master/maintainers/maintainer-list.nix>

⁴⁶³ <https://matrix.to/#/#community:nixos.org>

⁴⁶⁴ <https://github.com/NixOS/nixpkgs/blob/master/maintainers/maintainer-list.nix>

- **会议与活动**

查看官方 NixOS 日历⁴⁶⁵ 和 Discourse 社区日历⁴⁶⁶ 获取实时或线下活动信息。部分社区团队会定期举行会议并公开会议记录。

6.3.3 其他渠道

若未找到能协助您贡献的特定用户或群组，可通过以下任一官方沟通渠道向整个社区提问：

- NixOS Matrix 空间⁴⁶⁷ 中与您问题相关的聊天室。
- Discourse 上的求助板块⁴⁶⁸。
- Matrix 上的通用 #nix⁴⁶⁹ 聊天室。

6.4 文档贡献

感谢您有意帮助改进 Nix 生态系统的文档！没有您的支持，本项目将无法实现。

6.4.1 入门指南

查阅文档资源概览（第172页）。文档贡献需遵循风格指南（第176页）。

如需进一步指导，请联系 Nix 文档团队⁴⁷⁰。

重要提示：若您无法贡献时间，请考虑通过Open Collective⁴⁷¹ 向NixOS基金会文档项目捐款，以资助参考文档和学习资料的持续维护与开发。

6.4.2 反馈

反馈同样是有价值的贡献。请在Discourse⁴⁷² 的文档分类中分享您的想法。

Nix初学者

请尽量以官方文档作为主要学习资源，即使它可能看起来不够完善。

请提交问题并报告遇到的所有困难或疑问。同时请说明您的学习目标及当前已尝试的学习路径。

分享您的第一手经验将有助于指导我们的工作，并为您和其他所有人解决文档中反复出现的问题。

⁴⁶⁵ <https://calendar.google.com/calendar/u/0/embed?src=b9052fobqjak8oq8lfkhg3t0qg@group.calendar.google.com>

⁴⁶⁶ <https://discourse.nixos.org/t/community-calendar/18589>

⁴⁶⁷ <https://matrix.to/#/#community:nixos.org>

⁴⁶⁸ <https://discourse.nixos.org/c/learn/9>

⁴⁶⁹ <https://matrix.to/#/#nix:nixos.org>

⁴⁷⁰ <https://nixos.org/community/teams/documentation>

⁴⁷¹ <https://opencollective.com/nixos/projects/nix-documentation>

⁴⁷² <https://discourse.nixos.org/c/dev/documentation/25>

Nix教育工作者

您可能已注意到学习者最常遇到的难点及其典型需求与疑问。或许您还拥有自己的课堂笔记、培训资料或演示文稿。

请分享您的经验，帮助我们改进上游文档和初学者材料，让您能更专注于为学生提供核心价值。

使用Nix的领域专家

若您擅长用Nix解决特定领域问题，并希望分享最佳实践，请先查阅现有内容。

- 现有相关材料是否符合您的标准？
- 我们该如何改进它？
- Nix 的功能是否有尚未涵盖的热门应用场景？
- 我们很乐意吸纳您的见解。

6.4.3 许可与署名

当提交包含您个人贡献的拉取请求时，即表示您同意根据 CC-BY-SA 4.0⁴⁷³ 许可协议授权您的工作。若添加他人创作的材料，请确保其许可证允许此类使用。此时需在新添加内容中明确标注来源、作者及许可信息。理想情况下，使用前应事先通知原作者。

将原作者添加为合著者⁴⁷⁴ 至您拉取请求的首个提交（该提交应包含原始文档的逐字内容），以便我们通过版本历史追踪作者身份与修改记录。

个别文档可采用CC-BY-SA 4.0以外的自由许可协议。当您对这些文档贡献修改时，即表示同意按相应协议授权您的工作成果。

注意：若您编写了与Nix相关的教程或指南，请考虑采用CC-BY-SA 4.0协议授权！这将允许我们在您的作品补充或改进现有材料时，将其收录为官方文档。

文档资源

本文概述Nix、Nixpkgs和NixOS的文档资源，并提供参与改进的建议方案。

参考手册

参考手册记录了接口与行为规范，提供示例说明，并定义组件专用术语。

- Nix参考手册（第156页）
 - source⁴⁷⁵
 - issues⁴⁷⁶
 - 拉取请求⁴⁷⁷

⁴⁷³ <https://creativecommons.org/licenses/by-sa/4.0/>

⁴⁷⁴ <https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/creating-a-commit-with-multiple-authors>

⁴⁷⁵ <https://github.com/NixOS/nix/tree/master/doc/manual>

⁴⁷⁶ <https://github.com/NixOS/nix/issues?q=is%3Aopen+is%3Aissue+label%3Adocumentation>

⁴⁷⁷ <https://github.com/NixOS/nix/pulls?q=is%3Aopen+is%3Apr+label%3Adocumentation>

- Nixpkgs 参考手册⁴⁷⁸

- 源代码⁴⁷⁹
 - 问题⁴⁸⁰
 - 拉取请求⁴⁸¹
- NixOS 参考手册⁴⁸²
- 源代码⁴⁸³
 - 问题⁴⁸⁴
 - 拉取请求⁴⁸⁵

各手册章节由被记录代码的开发者维护。

如何提供帮助：

- 在仅提及名称处添加定义、命令、选项等链接
- 确保技术术语使用一致
- 确保示例自包含且遵循最佳实践
- 完善看似不完整的章节

NixOS 维基

NixOS 维基⁴⁸⁶ 是NixOS用户指南、配置示例和故障排除技巧的集合，旨在作为 NixOS参考手册的补充。

由NixOS用户社区共同编辑。

如何提供帮助：

- 通过添加分类和参考文档链接提升可发现性
- 移除冗余或过时信息
- 为您的使用场景添加指南和配置示例

nix.dev

nix.dev⁴⁸⁷（来源⁴⁸⁸）旨在为Nix生态初学者提供指引。

文档团队以编辑身份维护nix.dev。

如何参与：

- 处理待解决问题⁴⁸⁹

⁴⁷⁸ <https://nixos.org/manual/nixpkgs>

⁴⁷⁹ <https://github.com/NixOS/nixpkgs/tree/master/doc>

⁴⁸⁰ <https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+is%3Aissue+label%3A%226.topic%3A+documentation%22+-label%3A%226.topic%3A+nixos%22>

⁴⁸¹ <https://github.com/NixOS/nixpkgs/pulls?q=is%3Aopen+is%3Apr+label%3A%226.主题%3A+文档%22+label%3A%226.主题%3A+nixos%22>

⁴⁸² <https://nixos.org/manual/nixos>

⁴⁸³ <https://github.com/NixOS/nixpkgs/tree/master/nixos/doc/manual>

⁴⁸⁴ <https://github.com/NixOS/nixpkgs/issues?q=is%3Aopen+is%3Aissue+label%3A%226.主题%3A+文档%22+label%3A%226.主题%3A+nixos%22>

⁴⁸⁵ <https://github.com/NixOS/nixpkgs/pulls?q=is%3Aopen+is%3Apr+label%3A%226.主题%3A+文档%22+label%3A%226.主题%3A+nixos%22>

⁴⁸⁶ <https://wiki.nixos.org/>

⁴⁸⁷ <https://nix.dev>

⁴⁸⁸ <https://github.com/nixos/nix.dev>

⁴⁸⁹ <https://github.com/nixos/nix.dev/issues>

- 审查拉取请求⁴⁹⁰
- 按拟定大纲添加指南或教程⁴⁹¹。新文章可基于以下视频创作：
 - Nix Hour⁴⁹² 录像
 - 部分 ~ 100NiXCon⁴⁹³ 录像
 - @jonringer 制作的 Nix 视频指南⁴⁹⁴
 - 2022年Nix之夏讲座⁴⁹⁵

由于编写指南或教程工作量较大，请务必与维护者协调，例如通过评论或提交议题。

论坛

Nix 用户在这些论坛版块中交流信息并互相支持：

- 求助⁴⁹⁶
- 指南⁴⁹⁷
- 链接⁴⁹⁸

如何提供帮助：

- 提出有见地的问题，展示你的工作
- 解答他人的疑问
- 通过更新或添加 NixOS 维基条目、nix.dev 指南/教程或参考手册，解决重复性问题
- 鼓励并协助他人将见解整合到官方文档中

Nix Pills

Nix Pills⁴⁹⁹ 是一系列关于使用 Nix 构建软件包的低阶教程，详细展示如何从第一性原理构建 Nixpkgs。

Nix Pills 目前未积极维护。

文档框架

我们致力于依据 Diátaxis 技术文档框架⁵⁰⁰ 构建文档，该框架将文档划分为四类：

- 教程（第175页）
- 指南（第175页）
- 参考（第175页）
- 概念（第175页）

490 <https://github.com/nixos/nix.dev/pulls>

491 <https://github.com/NixOS/nix.dev/issues/572>

492 <https://www.youtube.com/watch?v=wwV1204mCtE&list=PLyzwHTVJIRc8yjlx4VR4LU5A5044og9in>

493 <https://www.youtube.com/c/NixCon>

494 <https://www.youtube.com/user/elitespartan117j27>

495 https://www.youtube.com/playlist?list=PLt4_lkyRrOMWyp5G-m_d1wtTcbBaOxZk

496 <https://discourse.nixos.org/c/learn/9>

497 <https://discourse.nixos.org/c/howto/15>

498 <https://discourse.nixos.org/c/links/12>

499 <https://nixos.org/guides/nix-pills/>

500 <https://diataxis.fr>

我们发现贡献者往往难以理解这些分类间的区别，虽然强烈建议研读Diátaxis框架，但可简要总结如下：

参考

参考资料应当

- 聚焦“现有内容”，仅列举存在的函数、类等要素
- 采用简洁语言，文本与版式需便于速览和随机查阅
- 提供完整且相关的用法示例
- 关联相关内容以提升可发现性

教程

教程通过引导用户完成特定活动，帮助其了解生态系统中常用工具与模式。活动本身固然重要，但目标还包括串联读者已学知识的关联性。

教程结构应最大限度降低学习者的认知负荷，主动规避可能导致用户出错的选择与操作。

指南

指南是逐步实现特定目标或解决问题的操作列表，旨在帮助读者达成具体结果，而非理解底层理论或宏观背景。

指南默认读者已具备理解当前主题的基础知识，因此无需逐一解释新概念的引入。

概念

概念用于描述代码的内部机制，或阐释生态系统中特定想法或实体的思考方式。概念也可解释某事物现行运作方式背后的历史成因。

若您需要详述某事物的具体运作细节，很可能应撰写解释性说明。

指南与教程之辨

我们发现贡献者主要困惑在于指南与教程的区别。

以下是帮助您理解差异的几点说明。

- 指南适用于“工作”场景，读者只需按步骤操作即可达成目标。
 - 在此场景中，读者可能已知或不在意这些步骤的原理，他们只关心如何达成预期结果。
- 教程适用于“学习”场景，读者通过逐步操作来练习特定任务。
 - 此场景中少量激励或解释有助于读者串联已学知识，但核心仍是操作本身而非原理。

一个贴切的类比是在两种不同情境下驾驶飞机降落。

假设飞行员昏迷，此时你必须操纵飞机降落以避免坠毁。这种情况下你只想知道如何保命，不在乎原理或缘由，只求平安着陆。这就是指南类内容的适用场景。

飞行模拟器中的学员需要练习降落操作。这位准飞行员要掌握何时放下起落架、何时调整襟翼等具体技能。相比成功降落这个结果，模拟飞行中更关键的是分解动作的专项训练。这正是教程类内容的定位。

最后，关于操作指南与教程的区别还可以这样理解：

- 指南： "第一步做A，第二步做B，依此类推"
- 教程： "牵着我的手，我来示范如何完成"

风格指南

本文档概述了编写文档时遵循的准则。

写作风格

力求清晰简洁

我本想写一封更短的信，只是时间不够。

——布莱兹·帕斯卡⁵⁰¹

读者的时间和注意力是有限的。请格外珍惜他们的认知资源。

对贡献者和维护者的沟通同样如此：这是一个公共项目，许多人会阅读你写的内容。谨慎运用这种影响力。

- 遵循基于证据的简明语言指南⁵⁰²。
 - 避免使用行话。读者可能不熟悉特定技术术语。
 - 如果有更短、更简单的词能表达相同意思，就不要使用冗长复杂的词。
- 给出指令时使用祈使语气。例如，写成：

将 python310 包添加至 buildInputs。

避免使用对话语气，以免分散内容焦点。例如，不要写成：

"现在让我们像前一个教程中那样，将 python310 包添加到 buildInputs 中。"

使用包容性语言

改编自《贡献者公约》⁵⁰³ 与《卡彭特里斯行为准则》⁵⁰⁴：

- 采用友好且包容的措辞
- 对他人展现同理心与尊重
- 尊重不同的观点与经历
- 提出并优雅接受建设性批评

⁵⁰¹ https://en.m.wikiquote.org/w/index.php?title=Blaise_Pascal&oldid=2978584#Quotes

⁵⁰² <https://www.plainlanguage.gov/guidelines/>

⁵⁰³

https://github.com/EthicalSource/contributor_covenant/blob/cd7fcf684249786b7f7d47ba49c23a6bcb3233eb/content/version/2/1/code_of_conduct.md

⁵⁰⁴ <https://github.com/carpentries/docs.carpentries.org/blob/fb188fa8d7f57ad85eb525091e335ed0d8fea16d/source/policies/coc/index.md#L13-L19>

- 聚焦于对社区最有益的事项

避免使用习语，非英语母语者可能难以理解。

不要试图幽默。幽默具有高度文化敏感性。往好了说，玩笑可能模糊关键指引；往坏了说，玩笑可能冒犯读者并使我们帮助其学习的努力付诸东流。

不要引用流行文化。你认为众所周知的内容，对来自不同背景的人可能完全陌生且会分散注意力。

语态

客观描述主题，在直接指引中使用祈使语气。

避免与读者建立个人化关系，追求清晰简洁而非情感诉求。

用“你”指代读者，仅用“我们”指代作者。两者都应谨慎使用。

例如：

你需要将密钥部署到远程机器。我们选择在此展示显式的手动操作过程（使用`s cp`），但实际存在多种自动化工具可选。

确保准确，引用来源

比没有文档更糟糕的只有错误文档。确保准确性的方法之一是引用来源。若对某功能原理作出声明（如命令行参数存在），请链接至该主题的官方文档。我们希望构建文档网络，相互链接能强化文档生态体系。

明确鼓励在适当情况下更新或重构手册，以提升整体使用体验。

标记与源代码

代码示例

在展示代码前务必说明其动机，用文字描述其用途或功能。

反例

运行以下命令：

```
```bash
:(){ :|:& };:
```

---

复杂示例可能需要额外说明，尤其是涉及给定上下文之外的概念时。使用折叠内容框放置会打断阅读流畅性的解释。

---

#### 示例

触发一个[fork炸弹]([https://en.wikipedia.org/wiki/Fork\\_bomb](https://en.wikipedia.org/wiki/Fork_bomb))：

```
```bash
():{ :|:& };:
```

(续下页)

:::{dropdown} 详细说明
该 Bash 命令定义并执行了一个递归生成自身副本的函数`:`，会快速消耗系统资源。
副本，迅速耗尽系统资源。
:::

始终在正文中解释代码。代码示例中的注释应极其精简，仅用于突出特定重点。

读者浏览信息时往往会略过大段代码——即使大部分是注释。初学者尤其会觉得阅读复杂代码吃力，可能因此完全跳过。

若某代码示例需要大量行内解释，建议替换为更简单的示例。若不可行，则将其拆分为多个部分分别讲解，最后展示完整合并结果。

预期能运行的代码示例应当确保可运行。

若需展示不可运行的示例（如演示常见错误），须事先说明。许多读者会执着于让示例代码运行起来，而不会继续阅读以发现该代码本就不该运行。

代码示例在适用时都应标注编程语言以便渲染时语法高亮，例如：

```
```python
print("Hello, World!")
...```

```

## 标题

将最大标题 (#) 保留用于文档标题。

使用 Markdown 的 ## 到 ### 级标题划分文档正文内容。并非标题层级越细越好。

## 每行一句

每行只写一个句子。

这样能立即凸显长句，也便于审查修改和直接提出建议——因为 GitHub 的审阅界面是基于行操作的。

## 链接

谨慎使用参考链接<sup>505</sup> 以提升源码可读性。将定义紧贴其首次使用位置。

## 示例

我们采用[Diátaxis](https://diataxis.fr/)框架来组织文档。该体系区分了[教程]、[指南]、[参考]和... → [说明]四种类型。

[教程]: <https://diataxis.fr/tutorials/>

(下页继续)

505 https://github.github.com/gfm/#reference-link

(接上页)

[指南]: <https://diataxis.fr/how-to-guides/>  
 [参考]: <https://diataxis.fr/reference/>  
 [说明]: <https://diataxis.fr/explanation/>

除非明确要求指向外部资源的最新版本，否则所有引用均应为永久链接<sup>506</sup>。

许多网络服务提供永久链接，例如：

- 指向特定提交的 GitHub 网址<sup>507</sup>
- 指向特定页面版本的维基百科网址<sup>508</sup>
- 使用互联网档案馆的“立即保存页面”功能永久保存网页<sup>509</sup>

## 如何编写教程

这是一份关于编写 Nix 教程的指南。

我们所说的教程是指技术文档框架 Diátaxis 中描述的课程形式<sup>510</sup>，建议在继续之前先熟悉 Diátaxis。特别要注意教程与指南之间的区别<sup>511</sup>。

## 目标受众

Nix 教程的主要目标受众是具有至少基础 Linux 命令行经验的软件开发人员。

专家即时解答问题、个性化指导与培训以及其他形式的师徒传授，已被公认为学习 Nix 最有效的支持方式。本教程面向无法获得这些资源的学习者，因此撰写时应适合自主学习。这通过遵循此处概述的结构来实现，其核心特点是力求避免并填补学习者所有信息缺口。

作为副产品，编写精良的教程也可作为互动培训课程的讲义使用。因此，次要目标受众是教授 Nix 的讲师。

## 流程

撰写高质量教程需要投入时间——无论对你还是他人。大部分时间通常用于协作：

- 找到正确的方法
- 确保指导内容对学习者既不过于简略也不过于冗长
- 寻找更简洁清晰的方式传达概念

遵循以下步骤避免重复劳动

506 <https://en.wikipedia.org/wiki/Permalink>

507 <https://docs.github.com/en/repositories/working-with-files/using-files/getting-permanent-links-to-files>

508 <https://zh.wikipedia.org/wiki/Wikipedia:链接至维基百科#页面旧版本的永久链接>

509 <https://web.archive.org/save>

510 <https://diataxis.fr/>

511 <https://diataxis.fr/tutorials-how-to/>

## 选择一个主题

文档团队已确定教程系列应包含的教程存在跟踪问题<sup>512</sup>。请选择一个你熟悉或特别感兴趣的主题对应的问题。

检查相关问题和拉取请求，确保不会重复他人已开始的工作！

现有教程请求<sup>513</sup>比大纲中记录的更多。若你想贡献的内容未被任何问题追踪，请新建问题<sup>514</sup>。这是明确目标的好机会，也能让他人了解该主题的需求。

## 提交包含大纲的拉取请求

按照推荐结构（第180页）提交教程大纲的拉取请求。大纲应包含各章节内容的要点。在拉取请求描述中引用跟踪问题，以宣告你正在编写教程。

评审将确保您的学习目标与技术实施方案方向正确。

## 扩展大纲内容

根据您的大纲和风格指南（第176页）详细阐述教程内容。

评审将确保您以合理顺序呈现所有必要信息，同时避免让学习者感到压力。

## 处理评审意见

根据详细反馈修改教程。建议邀请朋友或同事测试教程，这既能发现隐性前提条件，又能预估实际阅读时长。

最终审核将确保所有内容在技术上准确无误。

## 结构

每个教程都应回答以下问题。

此外，我们强烈推荐《学习是如何发生的》（摘要）<sup>515</sup>作为设计学习材料的指导手册。

## 你将学到什么？

描述问题陈述和学习目标。

教程的学习目标始终是掌握一项技能，其特点在于可应用于具有重复模式的一系列情境。

---

512 <https://github.com/NixOS/nix.dev/issues/572>

513 <https://github.com/NixOS/nix.dev/issues?q=is%3Aopen+is%3Aissue+label%3Atutorial>

514 <https://github.com/NixOS/nix.dev/issues/new?&template=tutorial.md>

515 <https://www.lesswrong.com/posts/mAdMkFqWzbJRB544m/book-review-how-learning-works>

## 你需要什么？

说明必备知识与技能。教程编写时需确保所述前提条件足以达成学习目标。

示例：

- 指向之前章节的链接
- 特定领域的技能或知识

## 需要多长时间？

预估阅读时间。这对学习者至关重要，能确保其有能力完成计划任务，从而避免因挫败感而中断探索Nix生态系统的旅程。

预估时间取决于学习者已有的知识储备和熟练程度。可标注可选技能或知识对阅读时长的影响。

## 该做什么？

提供达成学习目标的具体步骤。应以可重复执行的直接指令形式呈现，确保导向预期结果。

在::: {dropdown}区块中添加背景说明也很有价值。这能在最小化干扰的同时辅助理解。

## 你学到了什么？

提供练习或示例解析，以及其他自我评估手段。

此处也是向读者提供反馈渠道或向作者提问方式的理想位置，以便持续改进教程。

## 后续步骤

根据用例的探讨深度，引导读者查阅

- 参考手册
- 指南或其他教程
- 链接到已知优质外部资源，并附摘要说明
- 可用支持工具概览及其成熟度与维护状态
- 构想概览及社区讨论进展状态

我们建议明确区分实践性与理论性学习资源，这样读者能快速决定是要解决问题还是深入学习。

外部资源应附摘要以明确内容预期，最好包含阅读时长。博客文章链接需保留原标题，并标注 (<作者>, <年份>)：既尊重作者，也让读者了解信息时效性。



## 致谢

### 7.1 赞助方

以下个人与组织为本次工作提供了支持：

- @fricklerhandwerk<sup>516</sup> 自2023-02起担任团队负责人，2023-02至2024-04期间由Antithesis<sup>517</sup> 赞助
- @zmitchell<sup>518</sup> 于2023-03至2023-08期间领导学习之旅工作组，由flox<sup>519</sup> 赞助
- @infinisil<sup>520</sup> 在2022-11至2024-05期间参与团队工作，由Tweag<sup>521</sup> 赞助
- @lucperkins<sup>522</sup> 于2022年11月至2023年1月担任团队负责人，由Determinate Systems<sup>523</sup> 赞助
- @fricklerhandwerk<sup>524</sup> 于2022年5月至2022年10月担任团队负责人，由Tweag<sup>525</sup> 赞助

### 7.2 历史

衷心感谢过往贡献者，他们帮助塑造了Nix文档的今日面貌：

- @infinisil<sup>526</sup> 在2022年11月至2024年5月期间协助领导团队。期间他勤勉地完成了无数贡献的技术评审，重构了Nixpkgs<sup>527</sup> 的贡献指南，并重写了模块系统教程<sup>528</sup> 以供发布。
- @olafklingt<sup>529</sup> 于2022年10月至2024年5月期间在团队中担任志愿者，并在2022年10月至2024年5月间成为正式成员。他撰写了NixOS虚拟机简介<sup>530</sup>，大幅简化了NixOS虚拟机测试教程<sup>531</sup>，并保持内容持续更新。这两篇文章广受欢迎，是我们教程系列的核心内容。
- @brianmcge<sup>532</sup> 于2023年3月至2023年10月期间加入团队，为学习之旅工作组做出了贡献。

---

516 <https://github.com/fricklerhandwerk>

517 <https://antithesis.com>

518 <https://github.com/zmitchell>

519 <https://floxdev.com>

520 <https://github.com/infinisil>

521 <https://tweag.io>

522 <https://github.com/lucperkins>

523 <https://determinate.systems>

524 <https://github.com/fricklerhandwerk>

525 <https://tweag.io>

526 <https://github.com/infinisil>

527 <https://github.com/NixOS/nixpkgs/blob/master/CONTRIBUTING.md>

528 <https://nix.dev/tutorials/module-system/deep-dive>

529 <https://github.com/olafklingt>

530 <https://nix.dev/tutorials/nixos/nixos-configuration-on-vm>

531 <https://nix.dev/tutorials/nixos/integration-testing-using-virtual-machines>

532 <https://github.com/brianmcgee>

- @zmitchell<sup>533</sup> 在2023年3月至2023年8月期间领导学习路径工作组<sup>534</sup>，新增了多项教程。他定期发布该阶段文档进展的更新<sup>535</sup>。
- @Mic92<sup>536</sup> 是创始成员之一，于2022年5月至2023年1月在团队任职。Jörg在NixOS维基上撰写了大量文档，并分享经验为文档团队确立方向。
- @domenkozar<sup>537</sup> 是创始成员之一，于2022年5月至2023年1月在团队任职。Domen最初创建了nix.dev，编写了许多基础教程，并通过Cachix<sup>538</sup>资助编辑工作。他协助组建文档团队、分配权限，并在多方面提供建议。Domen于2023年7月将nix.dev捐赠给NixOS基金会。

---

533 <https://github.com/zmitchell>

534 [https://discourse.nixos.org/search?q=%E5%8A%A0%E6%96%87%E8%8A%A8%E6%95%BF%E5%9B%9E%20-%20%E4%BA%A4%E6%8D%A2%E8%AE%A8%E5%91%98%20in%3Atitle%20order%3Alatest\\_topic](https://discourse.nixos.org/search?q=%E5%8A%A0%E6%96%87%E8%8A%A8%E6%95%BF%E5%9B%9E%20-%20%E4%BA%A4%E6%8D%A2%E8%AE%A8%E5%91%98%20in%3Atitle%20order%3Alatest_topic)

535 [https://discourse.nixos.org/search?q=Nix%20%E6%96%87%E8%8A%A8%20in%3Atitle%20before%3A2023-10-30%20order%3Alatest\\_topic](https://discourse.nixos.org/search?q=Nix%20%E6%96%87%E8%8A%A8%20in%3Atitle%20before%3A2023-10-30%20order%3Alatest_topic)

536 <https://github.com/Mic92>

537 <https://github.com/domenkozar>

538 <https://www.cachix.org/>

## **INDEX**

### N

- Nix, 155
- Nix表达式, 155
- Nix文件, 155
- Nix语言, 155
- NixOS, 155
- Nixpkgs, 155