パッケージマネージャ Nix で実現する
宣言的でクリーンな PHP 開発環境の構築

たけてぃ @takeokunn

1. はじめに

PHP 開発者にとって環境構築は永遠の課題です。 PHP 8.3 や 8.1 といったバージョンの違い、拡張モジュールの違いなど、複数プロジェクトを並行開発する際に環境の混在は避けられません。 本業と OSS・副業プロジェクトの環境が混在し、マシン買い替え時の環境再構築に苦労した経験は誰しもあるのではないでしょうか。

環境構築で遭遇する典型的な問題を挙げてみます。

- `pecl install redis` が謎のコンパイルエラーで失敗する
- Intel Mac から Apple Silicon Mac への移行で Homebrew のパスが変わり環境が壊れる
- `brew upgrade` したら依存していた PHP バージョンが消えてプロジェクトが動かなくなる
- 新メンバーのオンボーディングで「README の手順とおりにやったのに動かない」といわれる
- 半年ぶりに触るプロジェクトの環境構築手順を思い出せない

これらの問題の多くは、環境構築が「手順の実行」に依存していることに起因します。 手順書は陳腐化し、環境は人によって微妙に異なり、再現性が担保されません。

本記事ではそんな環境構築の課題を解決する選択肢として、パッケージマネージャ Nix と devenv を使った宣言的な PHP 開発環境の構築について紹介します。

2. Nix について

2.1. 概要

Nix[1]は 2003 年に Eelco Dolstra 氏によって開発された純粋関数型パッケージマネージャです。 従来のパッケージマネージャと異なり、パッケージを副作用なく管理できることが特徴です。

Nix の最大の特徴は「宣言的管理」です。 「あるべき状態」を設定ファイルに記述し、Nix がその状態を実現します。 これにより、設定ファイル自体が環境のドキュメントとなり、半年後の自分でも同じ環境を再現できます。

Nix はパッケージマネージャとしてだけでなく、NixOS（Linux ディストリビューション）や nix-darwin（macOS 設定管理）の基盤技術でもあります。 本記事ではパッケージマネージャとしての Nix に焦点を当てます。

2.2. nixpkgs とバイナリキャッシュ

nixpkgs[2]は Nix の公式パッケージリポジトリで、12 万以上のパッケージが含まれています。 GitHub でもっともアクティブなリポジトリの 1 つで、PHP も複数バージョン（8.1〜8.4）が利用可能です。

利用可能な PHP バージョンは NixOS 公式サイト（https://search.nixos.org/packages?channel=unstable&query=php）で検索できます。 php82、php83、php84 などが見つかります。

Nix はデフォルトで cache.nixos.org からビルド済みバイナリをダウンロードします。 ソースからビルドする必要がないため、 `brew install` と同等の速度でパッケージをインストールできます。

```shell
# バイナリキャッシュから取得（数秒〜数十秒）
$ nix run nixpkgs#php

# キャッシュがない場合はソースビルド（数分〜数十分）
# → 通常のパッケージはほぼキャッシュが存在
```

2.3. Nix の仕組み

2.3.1. Nix ストアによるパッケージ分離

従来のパッケージマネージャでは `/usr/local/bin/php` のような共有パスを使うため、PHP 8.3 と 8.1 を同時にインストールすると衝突が発生します。

Nix はすべてのパッケージを `/nix/store` という専用ディレクトリに、ハッシュ値を含むパスで保存します。

```shell
/nix/store/abc123...-php-8.3.0/bin/php
/nix/store/def456...-php-8.1.27/bin/php
/nix/store/ghi789...-composer-2.7.1/bin/composer
```

このハッシュ値はソースコード、ビルド手順、依存関係から計算されます。 同じ入力からは必ず同じハッシュが生成されるため、ビルドの再現性が保証されます。

2.3.2. 再現性の保証

Nix のビルドはサンドボックス環境で実行され、ネットワークアクセスや環境変数が遮断されます。 これにより「自分のマシンでは動くのに」という問題を大幅に減らせます。

```shell
# 同じflake.lockを使えば、誰がいつどこでビルドしても同じ結果
$ nix build .#myapp  # → /nix/store/xyz789...-myapp

# 1年後に別のマシンで実行しても同じハッシュ
```

PHP の `composer.lock` が依存ライブラリを固定するように、Nix は実行環境全体（PHP 本体、拡張モジュール、システムライブラリ）を固定します。

2.3.3. シンボリックリンクによる環境切り替え

`/nix/store` 内の各パッケージは「プロファイル」というシンボリックリンクの集合を通じて使います。 Nix は必要なパッケージだけを参照するプロファイルを作成します。

```shell
~/.nix-profile/bin/php -> /nix/store/abc123...-php-8.3.0/bin/php
~/.nix-profile/bin/composer -> /nix/store/ghi789...-composer-2.7.1/bin/composer
```

プロジェクトごとに異なるプロファイルを持てるため、A プロジェクトでは PHP 8.3、B プロジェクトでは PHP 8.1 といった使い分けが可能です。 実体は `/nix/store` に共存しており、シンボリックリンクの切り替えだけで環境が変わります。

2.3.4. ガベージコレクションとロールバック

`/nix/store` には使用中・未使用のパッケージが蓄積されていきます。 不要なパッケージは `nix-collect-garbage` で削除できます。

```shell
# 未使用パッケージの削除
$ nix-collect-garbage

# 古い世代も含めて削除（より多くの容量を解放）
$ nix-collect-garbage -d

# ストアの容量確認
$ du -sh /nix/store
```

Nix は環境の「世代」を保持しているため、問題が発生した場合は以前の状態にロールバックできます。 PHP のバージョンアップで不具合が出ても、すぐに元の環境に戻せる安心感があります。

2.3.5. Flakes による厳密なバージョン管理

`flake.lock` は nixpkgs のリビジョン（コミットハッシュ）を記録します。 nixpkgs には 10 万以上のパッケージが含まれており、リビジョンを固定することで、PHP、MySQL、Node.js などすべてのパッケージのバージョンが一意に決まります。

```json
// flake.lock（一部抜粋）
{
  "nodes": {
    "nixpkgs": {
      "locked": {
        "rev": "abc123...",
        "type": "github"
      }
    }
  }
}
```

`flake.lock` を Git 管理すれば、チーム全員が同じバージョンのパッケージを使えます。 `composer.lock` が PHP ライブラリを固定するように、 `flake.lock` は実行環境全体を固定します。

2.4. Homebrew との共存

「すでに Homebrew を使っているけど、Nix に完全移行しないといけない？」という疑問があるかもしれません。 答えは No です。Nix と Homebrew は問題なく共存できます。

Nix は `/nix/store` という独立したディレクトリにパッケージを配置し、Homebrew の `/opt/homebrew` （Apple Silicon）や `/usr/local` （Intel）とは完全に分離されています。 PATH の優先順位を調整すれば、プロジェクトでは Nix、それ以外では Homebrew という使い分けも可能です。

devenv と direnv を組み合わせれば、プロジェクトディレクトリに入ったときだけ Nix の環境が有効になり、ディレクトリを出れば Homebrew の環境に戻ります。 既存の環境を壊すことなく、段階的に Nix を導入できるのは大きなメリットです。

2.5. 既存ツールの課題

PHP 開発環境の構築にはさまざまなツールが存在しますが、それぞれに課題があります。

Homebrew はグローバル環境を汚染し、 `brew unlink` / `brew link` によるバージョン切り替えは煩雑です。 `php@8.1` と `php@8.3` を同時にアクティブにできず、プロジェクト間の切り替えが手動になります。

Docker はオーバーヘッド（起動時間、メモリ消費）が大きく、macOS ではファイル同期の遅延が開発体験を損ないます。企業規模によっては Docker Desktop のライセンス費用も考慮が必要です。

phpenv/asdf/mise は古い PHP バージョンのビルド失敗が頻発します。 macOS のシステムライブラリ変更（特に OpenSSL、libxml2）の影響を受けやすく、 `brew install openssl` してからパスを通すなどの作業が必要になることがあります。

Nix と Docker の使い分けについても触れておきます。 **ローカル開発** では Nix/devenv が適しています。起動が一瞬で、ファイル I/O もネイティブ速度です。 **本番環境・CI/CD** では Docker が依然として有力です。コンテナイメージによるデプロイの一貫性、Kubernetes との親和性は大きなメリットです。両者は排他的ではなく、「ローカルは Nix、デプロイは Docker」という組み合わせも現実的な選択肢です。

2.6. 宣言的管理の特徴

Nix は IaC（Infrastructure as Code）の考え方を開発環境に適用したものです。 Terraform が AWS リソースを `.tf` ファイルで定義するように、Nix は開発環境を設定ファイルで定義します。

```shell
# 命令的（Homebrew）- 手順を実行
$ brew install php@8.3 && pecl install redis xdebug
# → 手順書が必要、環境差異が発生しやすい
```

```nix
# 宣言的（Nix）- あるべき状態を記述
{ pkgs, ... }: {
  languages.php = {
    enable = true;
    package = pkgs.php83.buildEnv {
      extensions = { all, enabled }: with all; enabled ++
      [ redis xdebug ];
      extraConfig = "memory_limit = 256M";
    };
  };
}
# → 設定ファイル自体がドキュメント
```

命令的アプローチは「どうやって構築するか」を記述し、宣言的アプローチは「どうあるべきか」を記述します。 Nix が差分を検出し、あるべき状態に収束させます。

2.7. 言語仕様

Nix 言語は純粋関数型言語で、パッケージのビルド定義に特化しています。 構文は JSON を拡張したような形式で、関数や変数束縛が追加されています。

```nix
# 基本的な値
"hello"                 # 文字列
42                      # 数値
true                    # 真偽値
[ 1 2 3 ]               # リスト
{ name = "php"; }       # アトリビュートセット
```

関数は `引数: 本体` の形式で定義します。

```nix
# 関数定義
add = x: y: x + y;    # add 1 2 => 3

# パターンマッチ引数（devenv.nixでよく見る形）
{ pkgs, ... }: { languages.php.enable = true; }
```

`let ... in` で変数を束縛し、 `with` でスコープを簡略化できます。

```nix
let
  phpVersion = "8.3";
in {
  packages = with pkgs; [ php composer nodejs ];
  # pkgs.php, pkgs.composer... と書く代わりに短縮
}
```

詳細は公式マニュアル[3]を参照してください。

3. PHP を取り巻く Nix 環境

Nix で PHP 開発環境を構築する方法は主に 3 つあります。

従来の **nix-shell** は `shell.nix` を記述して環境に入る方法ですが、依存関係のバージョン固定が弱く再現性に課題があります。 **Nix Flakes** は `flake.nix` と `flake.lock` で厳密なバージョン管理を実現しますが、Nix 言語の知識が必要で PHP 向けの設定は冗長になりがちです。

Nix を簡単に使うためのラッパーツールとして **devbox**[4] と **devenv**[5] があります。

devbox は Jetpack 社が開発するツールで、JSON ベースの設定ファイルを使用します。 Nix 言語を一切書かずに環境構築できるため、導入障壁がもっとも低いのが特徴です。

```JSON
{
  "packages": [
    "php@latest",
    "php83Packages.composer@latest",
    "nodejs@20"
  ]
}
```

一方、 **devenv** は Cachix 社が開発する Flakes ベースのツールです。 Nix 言語で記述しますが、言語ごとに最適化されたオプションを提供します。

```Nix
# devenv.nix
{ pkgs, ... }: {
  languages.php.enable = true;
  languages.php.package = pkgs.php83.buildEnv {
    extensions = { all, enabled }: with all; enabled ++
    [ redis ];
  };
}
```

devbox は手軽さが魅力ですが、devenv はサービス統合（MySQL、Redis 等）、pre-commit フック、カスタムスクリプトなど、より高度な機能を備えています。 PHP 開発では拡張モジュールの細かな設定や php.ini のカスタマイズが必要になることが多いため、本記事ではより自由度の高い devenv を使った環境構築を紹介します。

4. devenv について

4.1. 概要

devenv を使えば、学習コストを最小限に抑えてすぐに Nix を使い始められます。

devenv は `devenv.nix` と `devenv.lock` の 2 つのファイルでプロジェクトの開発環境を定義します。 `devenv.nix` には言語、パッケージ、サービス、スクリプトなどを記述し、`devenv.lock` で依存関係のバージョンを固定します。 `devenv shell` コマンドで定義した環境に入り、 `devenv up` で MySQL や Redis などのサービスを起動できます。

セットアップは次の手順で完了します。

```Shell
# Nixのインストール(例)
$ curl -L https://nixos.org/nix/install | sh

# Flakesの有効化（~/.config/nix/nix.conf）
# experimental-features = nix-command flakes

# devenvのインストール
$ nix-shell -p devenv

# プロジェクトの初期化
$ cd myproject
$ devenv init
```

4.2. PHP 環境の設定

devenv.nix ファイルに PHP 環境を記述します。 PHP のバージョン、拡張モジュール、php.ini の設定、追加パッケージなどを一箇所で管理できます。

```Nix
# devenv.nix
{ pkgs, ... }:
{
  languages.php = {
    enable = true;
    package = pkgs.php83.buildEnv {
      extensions = { all, enabled }:
        with all; enabled ++ [ redis xdebug pdo_mysql
        mbstring gd ];
      extraConfig = ''
        memory_limit = 256M
        display_errors = On
        xdebug.mode = debug
      '';
    };
  };

  packages = with pkgs; [ nodejs_20 yarn ];

  enterShell = ''
    echo "PHP $(php -v | head -n1)"
  '';
}
```

各ディレクトリに devenv.nix を配置するだけで、本業プロジェクト（PHP 8.3）、副業案件（PHP 8.1）、個人 OSS（PHP 8.4）といった複数環境を完全分離できます。

4.3. スクリプトとフック

devenv では `scripts` でカスタムコマンドを、 `git-hooks.hooks` で Git フックを定義できます。

```Nix
# devenv.nix - カスタムスクリプト
{ pkgs, ... }:
{
  languages.php.enable = true;

  scripts = {
    test.exec = "./vendor/bin/phpunit";
    lint.exec = "./vendor/bin/phpcs --standard=PSR12 src/";
    format.exec = "./vendor/bin/php-cs-fixer fix";
    analyze.exec = "./vendor/bin/phpstan analyse src/ --
    level=max";
  };
}
```

```Nix
# devenv.nix - Git pre-commitフック
{ pkgs, ... }:
{
  languages.php.enable = true;

  git-hooks.hooks = {
    # 組み込みフック
    php-cs-fixer.enable = true;

    # カスタムフック
    phpstan = {
      enable = true;
      name = "PHPStan";
      entry = "./vendor/bin/phpstan analyse --no-progress";
      files = "\.php$";
      pass_filenames = false;
    };
  };
}
```

`test` という名前で定義したスクリプトは `test` コマンドとしてシェル内で直接実行できます。 pre-commit フックは `devenv shell` に入ると自動的にセットアップされ、コミット時に実行されます。

4.4. 環境変数とシークレット管理

開発用の環境変数は `env` で直接定義するか、 `dotenv` で `.env` ファイルから読み込めます。

```nix
{ pkgs, ... }:
{
  # 環境変数の直接定義
  env = {
    APP_ENV = "development";
    APP_DEBUG = "true";
    LOG_LEVEL = "debug";
  };

  # .envファイルから読み込み
  dotenv.enable = true;

  # シェル起動時に動的に設定
  enterShell = ''
    export APP_KEY=$(cat .app-key 2>/dev/null || echo "base64:dummy")
    export PATH="$PWD/vendor/bin:$PATH"
  '';
}
```

本番用シークレットは `.env` を gitignore に追加し、 `.env.example` をリポジトリに含める従来のパターンがそのまま使えます。

4.5. サービス統合

devenv は Docker なしでデータベースやキャッシュサーバを起動できます。[6] `devenv up` でサービスを起動し、データは `.devenv/` に保存されます。

```nix
{ pkgs, ... }:
{
  services.mysql = {
    enable = true;
    initialDatabases = [{ name = "myapp"; }];
  };

  services.redis.enable = true;
  services.mailhog.enable = true;
}
```

4.6. direnv との連携

direnv[7]と連携することで、ディレクトリ移動だけで環境が自動的に切り替わります。 プロジェクトルートに `.envrc` ファイルを作成し、以下を記述します。

```shell
# direnvのセットアップ
eval "$(devenv direnvrc)"
use devenv
```

`direnv allow` で許可すると、以降は `cd` するだけで環境が切り替わります。

```shell
$ cd ~/projects/legacy-app     # PHP 8.1環境に自動切り替え
direnv: loading .envrc
PHP 8.1.x

$ cd ~/projects/new-app        # PHP 8.3環境に自動切り替え
direnv: loading .envrc
PHP 8.3.x
```

「このプロジェクトは PHP 何だっけ？」と悩む必要がなくなります。

5. 終わりに

筆者は本業・副業・OSS の全プロジェクトで devenv を使っており、環境構築で悩む時間が大幅に減りました。 devenv は非常に便利なツールです。

Nix には学習コストがありますが、devenv を使えばその壁を大幅に下げられます。 興味のある方はぜひ Nix 公式サイト[1]や devenv 公式ドキュメント[5]を参照してみてください。

Nix の適用範囲は開発環境だけではありません。 home-manager[8]を使えば git、vim、zsh などの開発ツールや、シェルの設定ファイル（dotfiles）も宣言的に管理できます。さらに nix-darwin（macOS）や NixOS（Linux）を使えば、OS 全体の設定を Nix で管理することも可能です。 「開発環境の IaC」から始めて、徐々に管理範囲を広げていくのもよいでしょう。

また、Software Design 2025 年 9 月号[9]〜2026 年 1 月号にて Nix の連載記事を執筆しました。 本記事で興味を持たれた方は、ぜひそちらもご覧ください。

図 1: Software Design 2025 年 9 月号表紙

参考文献

[1] 「Nix & NixOS ¦ Reproducible builds and deployments」. [Online]. 入手先: https://nixos.org/

[2] 「NixOS/nixpkgs: Nix Packages collection & NixOS」. [Online]. 入手先: https://github.com/NixOS/nixpkgs

[3] 「Nix Language - Nix Reference Manual」. [Online]. 入手先: https://nix.dev/manual/nix/latest/language/

[4] 「Devbox by Jetify」. [Online]. 入手先: https://www.jetify.com/devbox

[5] 「devenv - Fast, Declarative, Reproducible, and Composable Developer Environments」. [Online]. 入手先: https://devenv.sh/

[6] 「Supported Services - devenv」. [Online]. 入手先: https://devenv.sh/supported-services/

[7] 「direnv – unclutter your .profile」. [Online]. 入手先: https://direnv.net/

[8] 「Home Manager Manual」. [Online]. 入手先: https://nix-community.github.io/home-manager/

[9] 技術評論社, 「Software Design 2025 年 9 月号」. [Online]. 入手先: https://gihyo.jp/magazine/SD/archive/2025/202509