開発ガイドライン - Mac開発・Windowsビルド前提

> 📋 **ステータス**: アクティブ（開発ガイドライン）
> 📅 **最終更新**: 2025-01-15
> 👤 **用途**: Mac開発・Windowsビルド前提の開発ガイドライン

📋 重要な前提条件

このプロジェクトは以下の前提で開発されています：

- **開発環境**: macOS（Macで開発）
- **ビルド・パッケージ化**: Windows（Windowsでビルドして配布用パッケージを作成）
- **エンドユーザー**: MacとWindowsの両方にスタンドアローンアプリとして提供
- **開発ツール**: MacではNode.jsを使用、Windowsでは開発しない

⚠ **重要**: すべてのコードは、MacとWindowsの両方で動作するように書く必要があります。

🎯 クロスプラットフォーム対応の基本原則

1. ファイルパスの扱い

❌ 避けるべき書き方

```
// 文字列連結でパスを構築（環境依存）
const filePath = `${basePath}/subfolder/file.txt`;
const filePath = basePath + "/subfolder/file.txt";
```

```
# Unix系コマンドを直接使用（Windowsで動作しない）
cp -r source dest
rm -rf directory
mkdir -p path/to/dir
```

✅ 推奨される書き方

```
// Node.jsのpathモジュールを使用
import * as path from 'path';

const filePath = path.join(basePath, 'subfolder', 'file.txt');
const dirPath = path.resolve(basePath, 'subfolder');
```

```
// package.jsonのスクリプトではshxを使用
{
  "scripts": {
    "copy:static": "shx mkdir -p .next/standalone/.next && shx cp -r
.next/static .next/standalone/.next/static",
    "clean": "shx rm -rf .next src-tauri/target/release"
  }
}
```

2. Rustコードでのパス処理

✅ **推奨される書き方**

```
use std::path::PathBuf;

// PathBufとjoin()を使用（クロスプラットフォーム対応）
let file_path = app_data_dir.join("subfolder").join("file.txt");

// パスを文字列に変換する際はto_string_lossy()を使用
let path_str = file_path.to_string_lossy().to_string();
```

3. file://プロトコルの扱い

❌ **避けるべき書き方**

```
// プラットフォームを考慮しないfile://URL
let file_url = format!("file://{}", file_path.to_string_lossy());
```

✅ **推奨される書き方**

```
// プラットフォーム判定を行い、適切な形式を生成
let file_url = if cfg!(target_os = "windows") {
    // Windows: file:///C:/path/to/file (3つのスラッシュ、スラッシュ区切り)
    let path_str = file_path.to_string_lossy().replace('\\', "/");
    format!("file:///{}", path_str)
} else {
    // Unix系: file:///path/to/file (2つのスラッシュ)
    format!("file://{}", file_path.to_string_lossy())
};
```

4. シェルスクリプトの扱い

原則

- **Mac側で実行するスクリプト**: `.sh`ファイルで問題なし
- **Windows側で実行するスクリプト**: `.bat`ファイルまたはNode.jsスクリプトを使用
- **package.jsonのスクリプト**: `shx`や`cross-env`などのクロスプラットフォーム対応ツールを使用

例

```
{
  "scripts": {
    "build": "cross-env TAURI_MODE=true NODE_ENV=production next build
&& npm run copy:static",
    "copy:static": "shx mkdir -p .next/standalone/.next && shx cp -r
.next/static .next/standalone/.next/static",
    "clean": "shx rm -rf .next src-tauri/target/release"
  }
}
```

🔧 開発時の注意点

TypeScript/JavaScriptコード

1. **パス構築**: 必ず`path.join()`または`path.resolve()`を使用
2. **環境変数**: `process.platform`でプラットフォーム判定が必要な場合のみ使用
3. **ファイル操作**: Node.jsの`fs`モジュールを使用（ブラウザ環境ではTauriコマンド経由）

Rustコード

1. **パス操作**: `std::path::PathBuf`と`join()`メソッドを使用
2. **プラットフォーム固有の処理**: `cfg!(target_os = "windows")`で条件分岐
3. **ファイルURL**: WindowsとUnix系で異なる形式を生成する必要がある

package.jsonスクリプト

1. **コマンド**: `shx`を使用（`cp`, `rm`, `mkdir`など）
2. **環境変数**: `cross-env`を使用
3. **プラットフォーム固有のスクリプト**: `package:win`のように別名で定義

🧪 テストとビルド確認

Macでの開発フロー

```
# 1. 開発サーバーの起動
npm run tauri:dev
```

```
# 2. ビルドテスト（copy:staticスクリプトの動作確認）
npm run build

# 3. リントチェック
npm run lint
```

Windowsでのビルド確認フロー

```
# 1. 依存関係のインストール
npm install

# 2. Next.jsのビルド（copy:staticスクリプトの動作確認）
npm run build

# 3. Tauriアプリのビルド
npm run tauri:build
```

📦 Windows転送用ファイルの作成

Mac側で実行

```
# Windows転送用ZIPファイルを作成
./create-windows-transfer-zip.sh
```

このスクリプトは以下を実行します：

- 必要なファイルのみをコピー
- 不要なファイル（node_modules、.next、targetなど）を除外
- macOS固有ファイル（.DS_Store、*.commandなど）を除外
- ZIPファイルを作成

Windows側での作業

1. ZIPファイルをWindowsマシンに転送
2. 解凍
3. `npm install`で依存関係をインストール
4. `npm run build`と`npm run tauri:build`でビルド確認

⚠ よくある問題と対処法

問題1: Windowsでビルドが失敗する

原因: `cp -r`や`rm -rf`などのUnix系コマンドが使用されている

対処法:

- `package.json`のスクリプトを`shx`を使用するように修正
- または、Node.jsスクリプトに置き換え

問題2: ファイルパスが正しく解決されない

原因: 文字列連結でパスを構築している

対処法:

- TypeScript: `path.join()`を使用
- Rust: `PathBuf::join()`を使用

問題3: file://URLがWindowsで動作しない

原因: Windows用の形式（`file:///C:/path`）になっていない

対処法:

- Rustコードで`cfg!(target_os = "windows")`で条件分岐
- パスをスラッシュ区切りに変換し、3つのスラッシュで開始

問題4: シェルスクリプトがWindowsで実行できない

原因: `.sh`ファイルはWindowsで直接実行できない

対処法:

- Windows側では使用しないスクリプトは`.sh`のままでも問題なし
- Windows側で実行する必要がある場合は`.bat`またはNode.jsスクリプトを作成

📝 コードレビューチェックリスト

新しい機能を追加する際は、以下を確認してください：

- ☐ ファイルパスは`path.join()`または`PathBuf::join()`を使用しているか
- ☐ `package.json`のスクリプトは`shx`や`cross-env`を使用しているか
- ☐ Rustコードで`file://`URLを生成する場合はプラットフォーム判定を行っているか
- ☐ Windowsでビルドできるか確認したか（可能であれば）
- ☐ Macでビルドが正常に動作するか確認したか

🔗 関連ドキュメント

- `architecture/port-and-server-design.md` - ポート設計とサーバー構成
- `database/database-design.md` - データベース設計
- `chromadb/CHROMADB_INTEGRATION_PLAN.md` - ChromaDB統合計画

💡 まとめ

重要なポイント：

1. **パスは必ず`path.join()`や`PathBuf::join()`を使用**
2. **package.jsonのスクリプトは`shx`を使用**
3. **file://URLはプラットフォーム判定を行う**
4. **Macで開発、Windowsでビルド確認**

これらの原則に従うことで、MacとWindowsの両方で正常に動作するコードを書くことができます。