# ChromaDB組み込みの実装計画

## 📊 調査結果のまとめ

ChromaDBの組み込み方法

### ❌ Rustクライアントのembedded modeは存在しない

Web検索の結果、以下のことが判明しました：

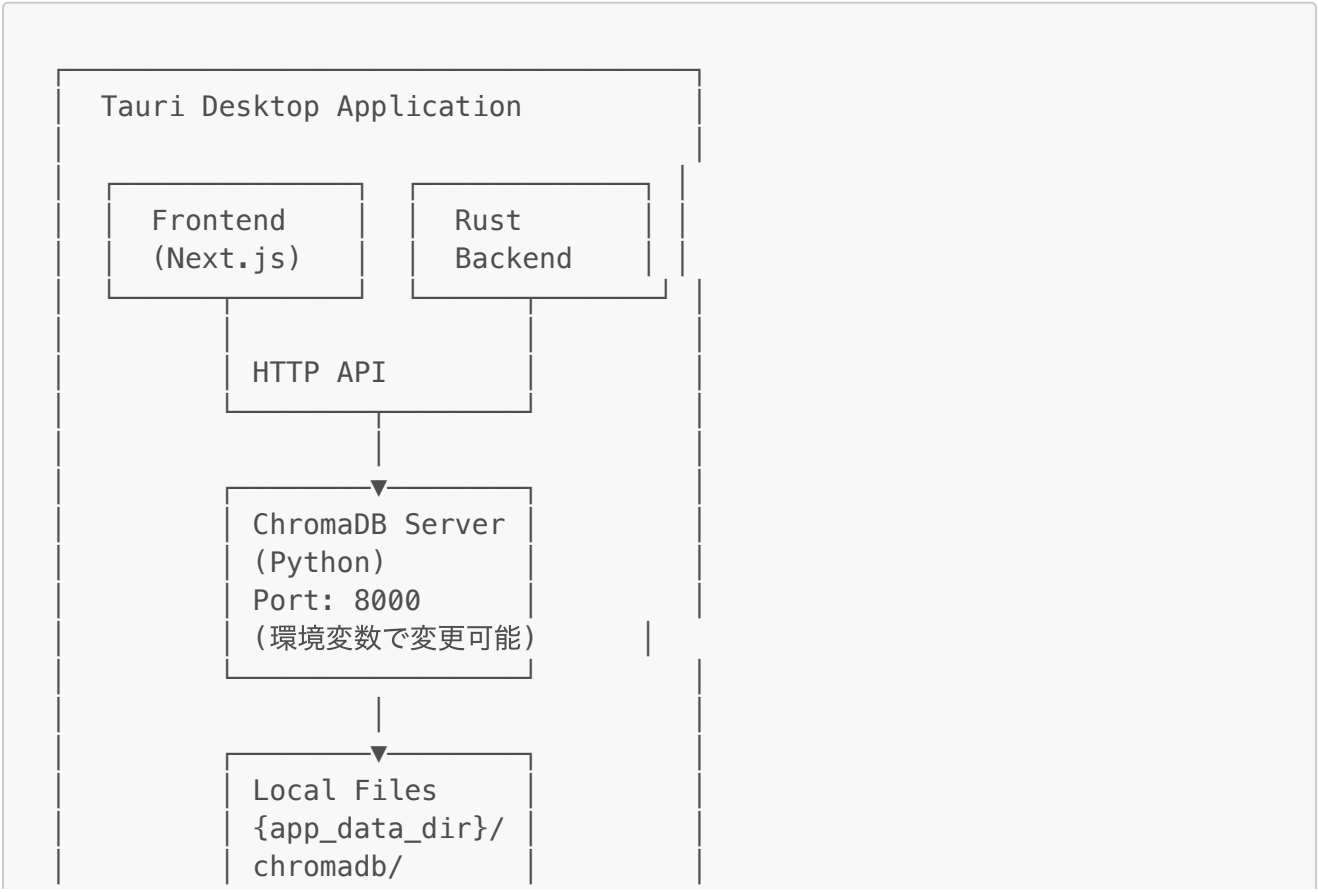
1. **ChromaDBのRustクレート（`chromadb`）はサーバー通信専用**
   - `persist_directory`はサーバーのデータ保存先を指定するだけ
   - サーバーなしで動作するembedded modeは**存在しない**
2. **ChromaDBのembedded modeはPythonクライアントでのみ利用可能**
   - `RustClient`はPythonバインディング経由でRust実装を使用
   - Python環境が必要（PyO3やPyOxidizerで埋め込む必要がある）

---

## ✅ ChromaDB組み込みの実現可能な方法

方法1: ChromaDB Serverをバンドルして起動（推奨）

#### 実装アーキテクチャ



**実装手順**

**1. ChromaDB Serverの起動（Rust側）**

```
// src-tauri/src/database/chromadb.rs
use std::process::{Command, Stdio};
use std::path::PathBuf;
use tokio::process::Command as TokioCommand;

pub struct ChromaDBServer {
    process: Option<tokio::process::Child>,
    port: u16,
    data_dir: PathBuf,
}

impl ChromaDBServer {
    pub async fn start(data_dir: PathBuf, port: u16) -> Result<Self,
String> {
        // Python環境の確認
        let python_path = Self::find_python()?;

        // chromaコマンドを探す（優先順位: chroma > chromadb）
        let chroma_cmd = Self::find_chroma_command()?;

        // ChromaDBサーバーを起動
        let mut child = TokioCommand::new(&chroma_cmd)
            .arg("run")
            .arg("--host")
            .arg("localhost")
            .arg("--port")
            .arg(port.to_string())
            .arg("--path")
            .arg(data_dir.to_string_lossy().as_ref())
            .stdout(Stdio::piped())
            .stderr(Stdio::piped())
            .spawn()
            .map_err(|e| format!("ChromaDBサーバーの起動に失敗しました: {}",
e))?;

        // サーバーが起動するまで待機（最大10秒、20回 × 500ms）
        for i in 0..20 {

tokio::time::sleep(tokio::time::Duration::from_millis(500)).await;

            // ヘルスチェック
            let health_check = reqwest::Client::new()
                .get(&format!("http://localhost:{}/api/v1/heartbeat",
```

```
port))
                .timeout(Duration::from_secs(1))
                .send()
                .await;

            if health_check.is_ok() {
                return Ok(Self {
                    process: Some(child),
                    port,
                    data_dir,
                });
            }
        }

        // 起動に失敗した場合、プロセスを終了
        let _ = child.kill().await;
        Err("ChromaDB Serverの起動確認に失敗しました（10秒以内に起動しませんでし
た）".to_string())
    }

    fn find_python() -> Result<String, String> {
        // Python 3.8-3.12を探す
        let candidates = vec!["python3.12", "python3.11", "python3.10",
"python3.9", "python3.8", "python3", "python"];

        for cmd in candidates {
            if Command::new(cmd)
                .arg("--version")
                .output()
                .is_ok()
            {
                return Ok(cmd.to_string());
            }
        }

        Err("Python環境が見つかりません。Python 3.8以上が必要で
す。".to_string())
    }

    fn find_chroma_command() -> Result<String, String> {
        // chromaコマンドを探す（優先順位: chroma > chromadb）
        let candidates = vec!["chroma", "chromadb"];

        for cmd in candidates {
            if Command::new(cmd)
                .arg("--version")
                .output()
                .is_ok()
            {
                return Ok(cmd.to_string());
            }
        }
```

```
            Err("chromaコマンドが見つかりません。`pip3 install chromadb`でインスト
ールしてください。".to_string())
        }

        pub async fn stop(&mut self) -> Result<(), String> {
            if let Some(mut process) = self.process.take() {
                process.kill().await
                    .map_err(|e| format!("ChromaDBサーバーの停止に失敗しました:
{}", e))?;
            }
            Ok(())
        }
    }
```

**2. Rust側からChromaDBに接続**

```
// src-tauri/src/database/chromadb.rs
use chromadb::client::{ChromaClient, ChromaClientOptions};

pub async fn init_chroma_client(port: u16) -> Result<ChromaClient,
String> {
    let options = ChromaClientOptions {
        url: Some(format!("http://localhost:{}", port)),
        database: "default".to_string(),
        auth: None,
    };

    ChromaClient::new(options)
        .map_err(|e| format!("ChromaDBクライアントの初期化に失敗しました: {}",
e))
}
```

**3. Tauriコマンドの追加**

```
// src-tauri/src/main.rs
#[tauri::command]
async fn init_chromadb(data_dir: String) -> Result<(), String> {
    let data_path = PathBuf::from(data_dir);
    let server = ChromaDBServer::start(data_path, 8000).await?;
    // サーバーインスタンスをグローバルに保持
    // ...
    Ok(())
}
```

---

方法2: Pythonインタープリターを埋め込む（複雑）

**実装方法**

PyO3やPyOxidizerを使ってPythonインタープリターをRustアプリケーションに埋め込み、ChromaDBのembedded modeを使用する。

**デメリット**

- ❌ **実装が非常に複雑**
- ❌ **配布パッケージサイズが大きい**（+150-200MB）
- ❌ **クロスプラットフォーム対応が困難**
- ❌ **メモリ使用量が高い**

---

## 📊 方法1（ChromaDB Server）の詳細

### メリット

1. **ChromaDBの機能をフルに活用**
    - メタデータ管理
    - コレクション管理
    - 高度なクエリ機能
    - チャンク化対応
2. **成熟したソリューション**
    - 実績がある
    - ドキュメントが充実
    - コミュニティサポート
3. **将来の拡張性**
    - AI統合が容易
    - チャンク化に対応
    - 複雑なクエリに対応

### デメリット

1. **起動時間**
    - サーバー起動に3-5秒かかる
    - **解決策**: バックグラウンドで起動、起動完了を待機
2. **メモリ使用量**
    - Python + ChromaDBサーバーで約250-400MB
    - **許容範囲**: ユーザーが「多少の設定は許容」と明言
3. **Python環境の配布**
    - Python環境をバンドルする必要がある

- **解決策**:
  - **Mac**: Pythonをバンドル（約30-50MB）
  - **Windows**: Pythonをバンドル（約30-50MB）
  - または、ユーザーにPythonのインストールを要求

4. **配布パッケージサイズ**
   - +70-130MB（Python環境を含む）
   - **許容範囲**: ユーザーが「多少の設定は許容」と明言

---

## 🎯 実装計画

### フェーズ1: ChromaDB Serverの起動機能

1. **Python環境の検出**
   - システムのPythonを探す（Python 3.8-3.12をサポート）
   - 見つからない場合はエラーを表示
2. **ChromaDB Serverの起動**
   - バックグラウンドプロセスとして起動
   - `chroma run`コマンドを優先的に使用（`chroma`または`chromadb`コマンドを検出）
   - 起動完了を待機（最大10秒、20回 × 500ms）
3. **ヘルスチェック**
   - サーバーが正常に起動したか確認
   - エラーの場合は適切なメッセージを表示

### フェーズ2: Rust側からの接続

1. **ChromaDBクライアントの初期化**
   - Rustクライアントでサーバーに接続
   - コレクションの作成・管理
2. **エンティティ埋め込みの保存・検索**
   - 既存のSQLite実装をChromaDBに移行
   - フォールバック機能を維持

### フェーズ3: 配布時の設定

1. **Python環境のバンドル**
   - Mac: Pythonをアプリケーションバンドルに含める
   - Windows: Pythonをインストーラーに含める
2. **起動時の自動起動**

- アプリケーション起動時にChromaDBサーバーを自動起動
- 終了時に自動停止

---

## 📝 配布時の設定

### Mac

```
# Python環境をバンドル
# アプリケーションバンドル内にPythonを含める
# または、Homebrewでインストールされたpythonを使用
```

### Windows

```
# Python環境をバンドル
# インストーラーにPythonを含める
# または、ユーザーにPythonのインストールを要求
```

---

## ポート設定とフォールバック仕様

### ポート設定

- **デフォルトポート**: 8000
- **環境変数**: `CHROMADB_PORT`で変更可能
- **ポート競合時**: 自動的に利用可能なポートを検出

詳細はポート設計とサーバー構成の設計書を参照してください。

### フォールバック仕様

ChromaDB Serverが起動しない場合の挙動:

1. **SQLite全文検索モードにフォールバック**
   - ベクトル検索は利用不可
   - 全文検索（FTS5）は利用可能
2. **フロントエンドに通知**
   - TauriイベントでChromaDBの状態を通知
   - UIを調整（検索機能の無効化など）
3. **次回起動時に自動再試行**
   - 前回の起動状態を確認
   - 失敗時は再試行

4. **プロセス監視と自動再起動**
    - 30秒ごとにChromaDBプロセスの状態を確認
    - 停止時は自動的に再起動を試みる

詳細はポート設計とサーバー構成の設計書の「ChromaDBフォールバック仕様」セクションを参照してください。

## 🎯 まとめ

結論

**ChromaDB Serverをバンドルして起動する方法が最適です。**

**理由:**

1. ✅ ChromaDBの機能をフルに活用できる
2. ✅ 将来の拡張性（チャンク化、AI統合）に対応できる
3. ✅ 実装が比較的シンプル
4. ✅ ユーザーが「多少の設定は許容」と明言している

**デメリットは許容範囲内:**

- ⚠ 起動時間: 3-5秒（バックグラウンド起動で対応）
- ⚠ メモリ使用量: 250-400MB（許容範囲）
- ⚠ 配布パッケージサイズ: +70-130MB（許容範囲）

次のステップ

1. **ChromaDB Serverの起動機能を実装**
2. **Rust側からの接続を実装**
3. **既存のSQLite実装をChromaDBに移行**
4. **配布時の設定を整備**
5. **ポート動的割り当てとフォールバック仕様の実装**