Jetpack Composeにおける

「副作用」との賢い付き合い方

副作用とは？🤔

プログラミングにおける「副作用」：

> **本来の責務ではない処理が、ある操作の結果として発生する状態の変化**

例：`calculateTotal()`関数の場合

- 本来の責務：合計金額を計算して返す
- 副作用の例：
  - データベースに注文履歴を書き込む
  - ユーザーのポイントを更新する

Composeにおける副作用とは？ 🎨

Composeの基本：

- コンポーザブル関数の責務は**UIを描画すること**
- 理想は「純粋な関数」
  - 同じ入力 → 同じUI
  - 外部状態は変化させない

実際に必要になる副作用 📱

UIの描画以外で必要な処理：

- ネットワークからのデータ取得（APIコール）
- データベースへのアクセス
- ユーザーイベントに応じた画面遷移
- 分析ツールのログ送信
- Androidのシステムサービス連携

これらは全て「副作用」！

LaunchedEffect 🚀

```
@Composable
fun UserProfile(userId: String) {
    LaunchedEffect(userId) {
        // userIdが変わるたびにユーザーデータをフェッチ
    }
}
```

- 画面表示時にコルーチンを起動
- キーが変更されたら再起動

rememberCoroutineScope 🔄

```
@Composable
fun SaveButton() {
    val scope = rememberCoroutineScope()
    Button(onClick = {
        scope.launch {
            // ボタンクリックでデータを保存
        }
    })
}
```

- コンポーザブルのライフサイクルに紐付いたスコープ
- UIイベントでの非同期処理に最適

SideEffect 📝

```
@Composable
fun AnalyticsScreen(eventName: String) {
    SideEffect {
        // UIが描画される直前に分析イベントを送信
    }
}
```

- UIが描画される直前に実行される
- Composeの状態を外部システムと同期する際に使用
- 再コンポジションのたびに実行される

DisposableEffect 🧹

```
@Composable
fun LocationTracker() {
    DisposableEffect(Unit) {
        // セットアップ処理
        val listener = LocationListener { /* ... */ }
        locationManager.requestLocationUpdates(listener)

        onDispose {
            // クリーンアップ処理
            locationManager.removeUpdates(listener)
        }
    }
}
```

- リソースのセットアップとクリーンアップが必要な場合に使用
- コンポーザブルが破棄されるときに確実にクリーンアップされる

produceState 📊

非同期データをComposeの状態として公開：

```
@Composable
fun UserDataLoader(userId: String) {
    val user by produceState<User?>(
        initialValue = null,
        userId
    ) {
        // userIdが変わるたびにフェッチ
        value = ApiService.fetchUser(userId)
    }
}
```

まとめ

1. JetpackComposeの副作用 = UI描画以外の外部状態への操作
2. Composeは専用APIで安全に管理
3. 適切なAPIの使い分けが重要
    - `LaunchedEffect`
    - `rememberCoroutineScope`
    - `SideEffect`
    - `DisposableEffect`
    - `produceState`

> より純粋なUIコンポーザブルで、保守性の高いコードを！