

Jetpack Composeにおける 「副作用」との賢い付き合い方

はじめに

皆さん、こんにちは！

今日はJetpack Compose開発で避けて通れない

「副作用（Side Effect）」について、

その概念との賢い付き合い方についてお話しします。

1. 「テスト容易性」って何だろう？

- **テスト容易性** = コードが思い通りに動くか確かめやすいこと
- 関数に対してスコープを絞ると：
 - 「実装された関数が仕様通りかを証明する」
- これがテストの本質

1.1. 数学でいうところの関数

- 同じ入力 (Input) → 同じ出力 (Output)
- 入力と出力だけで振る舞いを予測できる
- 2つの関数の等価性を証明するには：
 - 「任意の入力に対して出力が一致する」ことを示せばOK

「等しい」とは？

- 同じ入力（Input）→同じ出力（Output）
- 入力と出力だけで振る舞いを予測できる
- テストがしやすい！

2. Composable関数は「UIを生成する関数」

- **Composable関数** = 状態 (State) → UI
- 数学の関数 $f(x) = y$ のイメージ
- 入力と同じなら出力も同じ

例：シンプルなComposable

```
@Composable  
fun Greeting(name: String) {  
    Text("Hello, $name!")  
}
```






Composable関数のテスト容易性

- 入力Stateが同じなら、UIも同じ
- テストは「このStateでこのUIになるか？」だけ
- **Composable関数として公開されているAPIはこの性質を持つ**
- これが**高いテスト容易性**の理由

3. 副作用は「UIの一意性」を乱す存在

- 副作用 = 本来の責務外で外部状態を変更すること
- 例：UI表示中にユーザーデータをサーバー送信
- Jetpack Composeでも同じ

Composeでの副作用の例

-  ネットワークからデータ取得
-  データベース保存
-  アニメーション開始
-  ログ記録
-  画面遷移

副作用があると...

- 同じStateでもUIが一意に定まらない
- 外部状態が変わることで予測困難に
- テスト容易性や予測可能性が損なわれる

4. でも、副作用は悪ではない！

- 副作用 = 「悪」ではない
- アプリには不可欠な要素
- 例：
 - ボタン押下で画面遷移
 - サーバーから最新情報取得

副作用は「適切に」「予測可能に」扱う

- 正しく管理された副作用は
 - 実装を容易に
 - コードの可読性UP
- Androidはライフサイクルが複雑 → 副作用管理が重要

5. 副作用の適切な取り扱い方：「作用」 API

- Jetpack Composeは副作用管理のために
「作用 (Effect)」 APIを提供
- UI描画とは別に副作用を明示的・制御的に記述

作用APIとは？

- UIを出力せず、コンポジション完了時に副作用を実行するComposable
- 例：
 - 「このUI状態になったら、この副作用を実行」

6. 主な「作用」API一覧①

API	概要	主な使い所
rememberCoroutineScope	ライフサイクルに紐付く CoroutineScope取得	UIイベントで非同期 処理
LaunchedEffect	コンポジション時やキー変更時 にコルーチン起動	画面表示時のデー タ取得など
rememberUpdatedState	コルーチン等で最新値を参照	コールバックや値 のキャプチャ
DisposableEffect	セットアップとクリーンアップ	リソース管理・リス ナー登録解除

6. 主な「作用」API一覧②

API	概要	主な使い所
SideEffect	コミット後、UI描画前に実行	外部状態と同期・ログ送信
produceState	非同期ソースをState化	API/Flowからのデータ取得
derivedStateOf	複数Stateから新State導出	再計算の最適化
snapshotFlow	StateをFlowに変換	State変化をFlowで監視

6. 主な「作用」API一覧③

API	概要	主な使い所
NoWrapper	デバッグ用途のコンポーザブル	主にCompose内部の動作確認

主要APIの使い分け例

- `rememberCoroutineScope` : UIイベントで非同期処理
- `LaunchedEffect` : 画面表示や引数変更時の処理
- `DisposableEffect` : リソースのセットアップ/クリーンアップ
- `SideEffect` : 外部システムとの同期
- `produceState` : 非同期データのUI反映

7. まとめ

- Jetpack Composeの副作用 = UI描画以外の外部状態操作
- Composable関数は「純粋な関数」として保つとテスト容易
- でも副作用は不可欠 → **作用APIで安全に管理！**
- 適切なAPIを使い分けて、
 - 堅牢で管理しやすい
 - 高品質なUIを実現

ご清聴ありがとうございました！