

ACCESS テックブック 2

ACCESS 技術書典同好会 著

2022-01-22 版 ACCESS Co., Ltd. 発行

プロローグ

本書について

本書「ACCESS テックブック 2」は、株式会社 ACCESS に所属するエンジニアによって書かれた技術同人誌で、2019 年 9 月の技術書典 7 に出典したテックブックの 2 冊目です。1 冊目の続きの章もあれば、新しい題材を取り扱う章もあります。各章はそれぞれ完結しているので、好きなページからお読みください。

株式会社 ACCESS には、ブラウザエンジン、IoT、スマホアプリ、クラウド、ネットワーク、ハードウェア、ドローンと、とても幅広い技術分野に対し、専門的な深い知識を持ったマニアックなエンジニア達が在籍しています。

株式会社 ACCESS について

日本でインターネットの歩みが始まった 1980 年代、「すべてのモノをネットにつなぐ」という企業ビジョンとともに、株式会社 ACCESS は誕生しました。ACCESS は、このビジョンを DNA として成長し、インターネットの普及とともに「ネットにつなぐ技術」を進化させ続けてきました。IT 革命元年と呼ばれた 1999 年には、世界で初めて「携帯電話をネットにつなぐ技術」の実用化に成功。企業躍進の起爆剤となりました。さまざまなイノベーションを経て、IoT の時代がいよいよ幕を開けようとしています。創業時より思い描いていたビジョンが現実のものになる今、ACCESS は「ネットにつなぐ」技術で、世界により豊かな社会と暮らしを創造し、人々の次の未来の実現を目指します。
<https://www.access-company.com/recruit/>

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

プロローグ	2
本書について	2
株式会社 ACCESS について	2
免責事項	2
第 1 章 Flow でクリーンアーキテクチャーを最適化する	5
1.1 はじめに	5
1.2 単方向データフロー	5
1.2.1 双方向データバインディングとは	6
1.2.2 単方向データフローとは	6
1.2.3 単方向にするメリットとは	7
1.2.4 Android アプリの変化	8
1.2.5 iOS アプリの変化	8
1.3 クリーンアーキテクチャーと単方向データフロー	9
1.3.1 Presenter を State に置き換え	10
1.3.2 結局 ViewModel は必要なのか	12
1.4 Kotlin Coroutines Flow	13
1.4.1 コールドストリーム	13
1.4.2 ホットストリーム	14
1.4.3 Kotlin のストリーム事情	14
1.4.4 Flow の基本的な使い方	14
1.4.5 SharedFlow と StateFlow	18
1.4.6 Flow と LiveData、Rx との比較	20
1.5 Kotlin Coroutines Flow を Swift で observe する	21
1.5.1 環境構築	22
1.5.2 実装	24

目次

1.5.3	プレビューと実行	30
1.6	Combine を使って改善する	33
1.7	まとめ	35
第 2 章	2 章タイトル	36
第 3 章	3 章タイトル	37
第 4 章	4 章タイトル	38
第 5 章	5 章タイトル	39
第 6 章	6 章タイトル	40
第 7 章	7 章タイトル	41
	著者紹介	42

第 1 章

Flow でクリーンアーキテクチャーを最適化する

1.1 はじめに

弊社の一部チームでは、モバイルアプリ開発にクリーンアーキテクチャーを採用しています。プロジェクトが大きくなって新しい人が介入しても、基本の設計ポリシーが保たれるという利点があります。

しかし、モジュールやクラスの多さに伴ってデータの流がわかりにくいのは、クリーンアーキテクチャー採用前と比べてあまり変わっていないようです。

本書の前半では、クリーンアーキテクチャーの思想を守りつつデータの流をわかりやすくするため、Kotlin Coroutines Flow（以下、Flow）を用いて Unidirectional Data Flow（以下、単方向データフロー）のロジックを実装します。

そして後半では、ロジックを Kotlin Multiplatform Mobile（以下、KMM）のフレームワークとしてアプリに組み込みます。KMM を使うということは、Android だけでなく iOS でも Flow が使われるということですが、本書のサンプルでは iOS を使い、Kotlin の Flow が Swift にはどんな形で流れるのか、お楽しみの終盤展開となっています。

なお、クリーンアーキテクチャーの概要は、本書では触れていません。前作の ACCESS テックブック第 1 章、もしくは数ある書籍や検索をご参照ください。

1.2 単方向データフロー

まず、単方向データフローについて説明します。

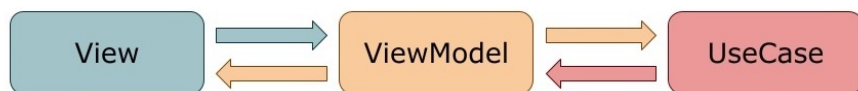
Web Frontend の方には馴染み深いと思いますが、2010 年代のモバイルアプリ開発では Two-way data binding（以下、双方向データバインディング）が主流でした。特に

Android は Data Binding や View Binding が公式から出ておりその傾向は顕著で、iOS も Rx 系を使って擬似的 or 結果的に双方向を実現しているケースが多いです。

ですが、2020 年代に入って SwiftUI/Combine や Jetpack Compose などの導入が進み、Web の Flux 系に似たような単方向データフローが浸透し始めています。ここでは、主にモバイル開発者向けにそれを一度おさらいします。

1.2.1 双方向データバインディングとは

単方向データフローと対の概念である双方向データバインディングとは、ある値がユーザー→View の入力に連動し、また通信結果などにも連動している状態のことです。値は ViewModel に置かれることが多いです。



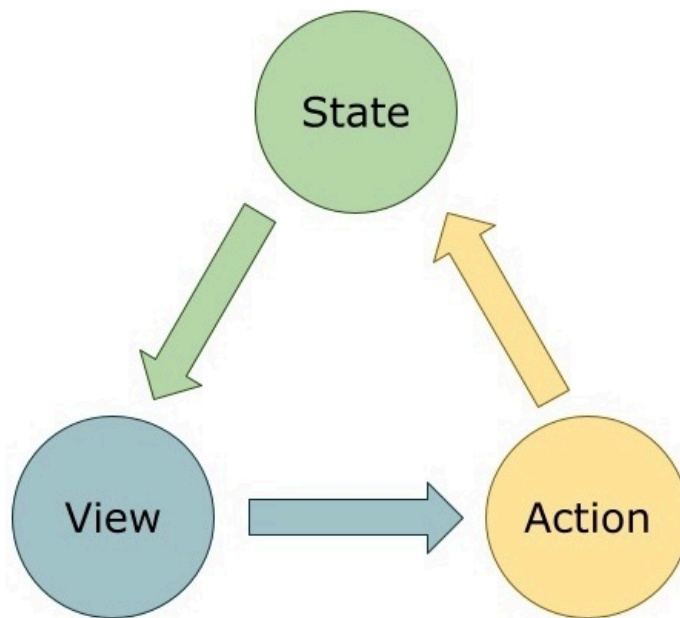
▲ 図 1.1 双方向データバインディングのデータの流れ

このように各境界が双方向に繋がっており、UseCase の変更が View に、View の変更が UseCase に伝播しやすく、責務を分割しながらコードを整理できるのが特徴です。

見かけのコード量は少なくできますが、各境界でのデータ型の変換や、ビューパーツごとにバインディングするための実装が必要になるため、全体のコード量は減らしづらい性質があります。

1.2.2 単方向データフローとは

かたや単方向データフローは、ユーザーが View に与えた変更を Action として受け取り、値は直接更新しません。Action はロジックを経て State を更新し、State が View に描画されるというサイクルの関係です。



▲図 1.2 単方向データフローのデータの流れ

単方向データフローでは、データは常に同じ方向へ流れ、逆流は許されません。Action は View からの命令、State は Action の結果、そして View は State の結果です。

1.2.3 単方向にするメリットとは

一見すると、双方向のほうが便利そうで、階層化されているためとつきやすい印象です。しかし、単方向化には色々とメリットがあります。

1. 単一化/カプセル化...Mutable な State が複数箇所に散らばるのを防ぐ。双方向の場合、どこに置くかは曖昧である
2. 共有...1 つの State を複数の子 View で使いまわしやすい。双方向の場合もできるが、View を越えて共有するための実装が複雑化する
3. 分離...State が更新されたときの影響を最小限に留められる。双方向の場合、スレッドなどの考慮が必要。データの流れが単方向なら、切り出しや拡張が容易で、デバッグやテストもしやすい

もちろんメリットだけでなく、単方向化のデメリットも存在します。

1. 単一化/カプセル化...Mutable な State が 1 箇所なので、適用できるデザインパターンが限られる
2. 共有...View ライフサイクルの細かい考慮が必要。開いていない View の State までメモリ上に持つのは理想的ではない
3. 分離...View のコード量、コンポーネントが増える。パーツごとに描画用の Variable と入力の Listener を毎回書く必要がある。MVVM や MVC に慣れた人が理解しづらかったり、プロジェクトごとに実装スタイルの差が出やすい

1.2.4 Android アプリの変化

Android アプリの場合、今までは@BindingAdapter がよく使われており、推奨アーキテクチャが MVVM なことから、双方向データバインディングが暗黙的に推奨されていたと言えます。

これはアプリ規模が大きくなるにつれ、アプリ独自のバインディングが多数実装されることを意味しました。やりすぎるとブラックボックス化が起きたり、Android 初心者にとって学習ハードルが高い部分でした。

また、接続がアノテーション任せな上に双方向なので、デバッグやバグ調査がしづらい問題もありました。

2021 年 7 月、待望の Jetpack Compose が登場し、単方向データフローが公式推奨されました。これを導入すると、UI から State を直接変えられなくなるため、先述の複雑性からある程度解放されます。

また、デメリットの 1 つであるライフサイクル問題も、remember 宣言子によって State の生存期間を View ライフサイクルに合わせられるので回避できます。

1.2.5 iOS アプリの変化

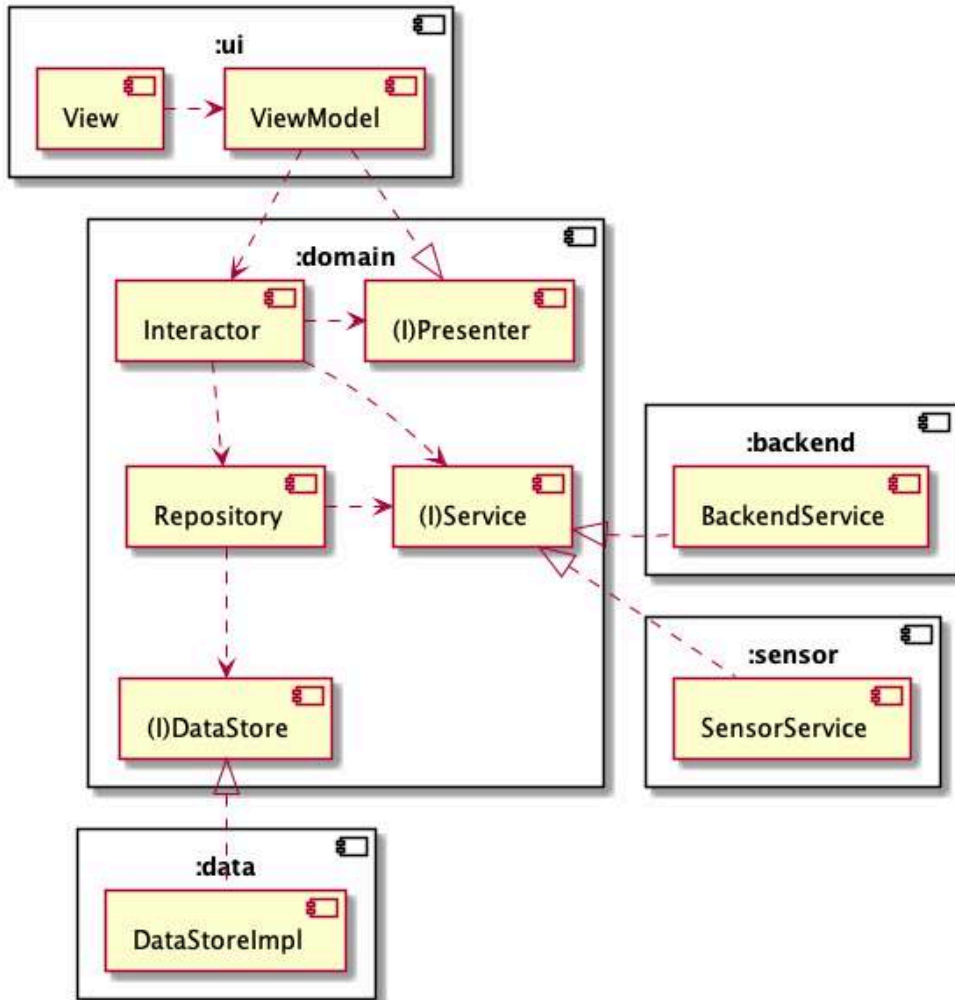
iOS アプリは、MVC や MVC+VM のようなアーキテクチャがよく使われてきました。

しかし、Rx 系をふんだんに使わないとあっという間に Controller が肥大化したり、外部のフレームワークに頼らざるをえないのが積年の問題でした。一部を Router に切り出したり、VIPER を導入したりなどの模索が続きました。

2019 年、Swift UI/Combine が登場し、こちらも単方向データフローが公式サポートのスタート地点に立ちました。非公式ですが TCA も登場し、着々と置き換えが進んでいます。

1.3 クリーンアーキテクチャと単方向データフロー

続いて、クリーンアーキテクチャと単方向データフローの親和性に関する考察です。



▲ 図 1.3 クリーンアーキテクチャの依存関係

domain から他へインターフェースを提供し、「この通りに実装して結果をください」(to backend, data, sensor)、「Presenter 型のオブジェクトを作ってくれば結果を代入

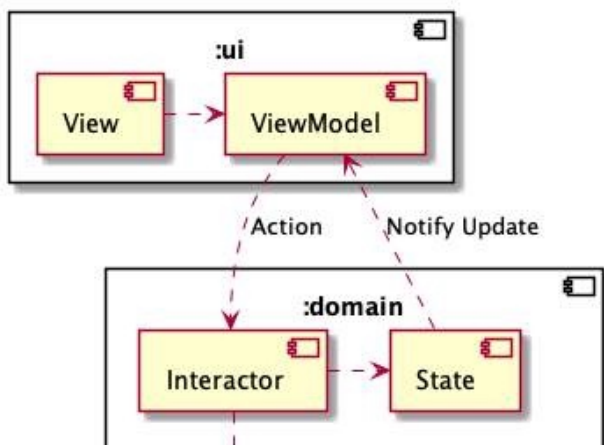
します」(to UI) とし、domain から他への依存を無くすのが、改善前のクリーンアーキテクチャーの特徴です。

そのクリーンアーキテクチャーのデータの流れを、本章でわかりやすくしていきます。

1.3.1 Presenter を State に置き換え

Presenter を採用していると、Action から View に戻るまでのルートを自由に決められるので、私の参画してる案件でも Application や AppDelegate なんかが Presenter を継承して、データの行き先がよくわからなくなってるロジックが多少ありました。

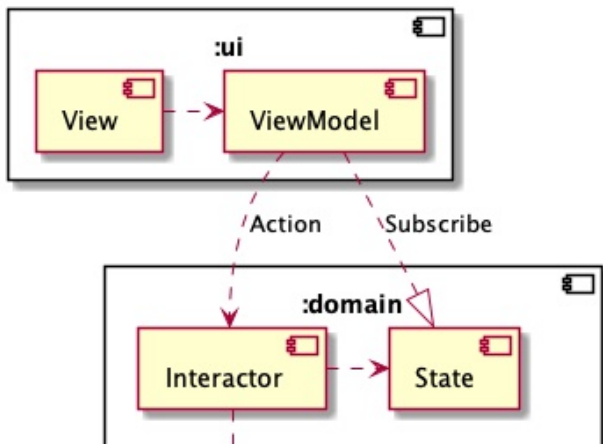
その Presenter をやめ、UI から domain に渡すものを Action、domain から UI には State の更新を通知するようにしてみます。



▲ 図 1.4 単方向データフローを適用した UI と domain 間の依存関係

図 1.2 っぽくなりました。ところが、このままではクリーンアーキテクチャーに反しています。Notify Update のところで、UI が domain に依存しているからです。

これは、UI 側から State を Subscribe することで回避可能です。

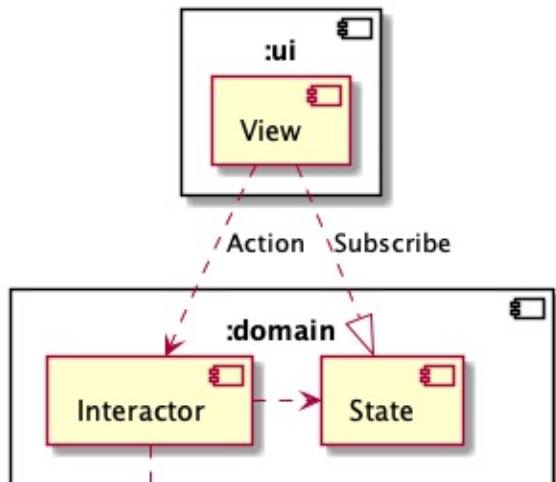


▲図 1.5 単方向データフローと DIP を適用した UI と domain 間の依存関係

さて、データの流れは View と ViewModel の間にもあります。

ViewModel から View へは、依存こそありませんが、データは流れます。つまり、View と ViewModel 間のデータの流れはまだ双方向です。このままでは、ViewModel のテストがしづらいです。

1.3.2 結局 ViewModel は必要なのか



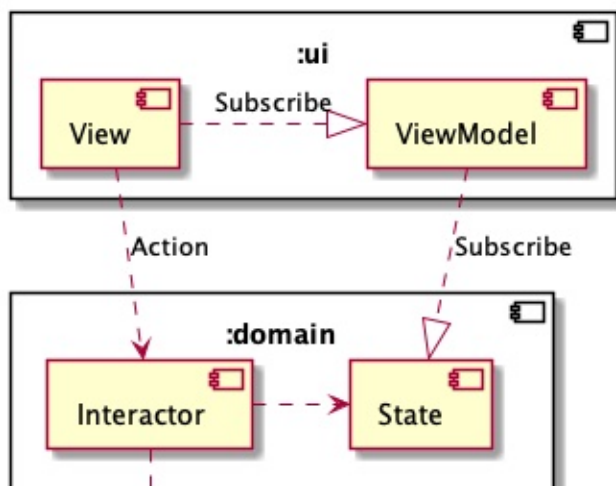
▲図 1.6 ViewModel を廃止した UI と domain 間の依存関係

思いきって ViewModel を取ってみました。すると、View に以下の責務が集中します。

1. 画面の表示
2. State の監視
3. State から画面パーツ向けの型変換
4. 画面の更新
5. 画面遷移

これなら、2 と 3 を ViewModel として分けての方がまだ読みやすいでしょうし、テストのしにくさに至っては悪くなっています。

では、View からのイベント発火で直接 `interactor` を呼び、ViewModel は上記 2 と 3 に専念させればどうでしょうか。



▲ 図 1.7 View から直接 Action を発火する場合の UI と domain 間の依存関係

スッキリした気がしますね。

この場合の ViewModel は、Subscribe したデータを View 向けに変換して View に送り出す役割ですが、iOS の TCA では ViewStore、Unio では ViewStream と呼ばれています。

ViewModel という呼び方が適切かどうかは諸説ありつつ、コンバーターとしての役割は View から切り離れたほうがよいと私は考えます。

1.4 Kotlin Coroutines Flow

ここまでは OS を限定せず記述しましたが、ここからは実際に Android アプリで Flow を使う例を紹介します。

Flow とは、Kotlin Coroutines の新しい非同期処理用ライブラリです。Rx や Promise に似た記述ができ、コールドストリームであることが特徴です。

1.4.1 コールドストリーム

Subscribe されたら初めて動きだす、Observable なストリームです。ストリームとは、データを連続して送り出す型を言います。

上司が来たら初めて働き出すぐうたら社員をイメージしてみてください。上司がいる間は、状況の変化をちゃんと逐次報告します。1 人の上司にのみ報告するというのも特徴で

す。そして、上司がいなくなったらすぐに自分から働くのをやめます。

社員だとどうかなと思うコールドな働き方ですが、プログラムとして、必要ないときに働かないのは実は強力な利点なんです。必要なときだけリソースを食い、不要になったら開放してくれるからです。

しかし、1 人の上司にしか報告できない点は、Observer が 2 つ登録されると、2 つのコールドストリームが必要なことを意味します。これはメモリ効率を考えればまいちなところですよ。

1.4.2 ホットストリーム

反対の型がホットストリームです。こちら Observable ですが、Subscriber がいなくても値を発行し、データを送り出します。Publisher と呼ばれることが多くあります。

こちらは上司が来る前から働き出す頑張り屋さんです。上司がいる間、状況の変化を逐次報告するのはコールドと同じですが、複数の上司がいても同じ報告を 1 人で請け負います。そして、上司が止めてくれるまで、いなくても働き続けるのです。

ちゃんと止めてあげないと必要ないときも働き続けてしまうのですが、Observer がいくつ登録されても、使うリソースが変わらないのは利点です。

これを応用すると、1 つのコールドストリームを受信し、複数の Subscriber に送信させるという中継地点の役割も担えます。

1.4.3 Kotlin のストリーム事情

元々 Kotlin には、Channel というホットストリームが存在していました。しかし、suspend の非同期処理をシーケンシャルに繋げたい場合、コールドストリームのほうが望ましく、それは遅れて登場した Flow を待つ必要がありました。

実際に Flow が登場すると、非同期処理を直感的に実装でき、安全で習得しやすく使いやすいと、次々と移行が進んでいます。

1.4.4 Flow の基本的な使い方

Android アプリで、100ms 毎に 0 から 100 までカウントする処理を、Flow を使って双方向データバインディングで実装してみましょう。

CounterUseCase.kt

```
class CounterUseCase {
    suspend fun countStream(): Flow<Int> = flow {
        repeat(100) { count ->
            delay(100)
            emit(count) // 0 1 2 3 4 ... 99
        }
    }
}
```

クリーンアーキテクチャーの図 1.3 での Interactor の部分が、無加工のデータを非同期に送ります。

CounterViewModel.kt

```
class CounterViewModel: ViewModel() {
    val showing = MutableLiveData<String>()

    fun showCountEvenNumbersSquared() {
        viewModelScope.launch(Dispatchers.Main) {
            useCase.countStream()
                .drop(1)
                .filter { it % 2 == 0 }
                .map { (it * it).toString() }
                .take(5)
                .collect { count.value = it } // 4 16 36 64 100
        }
    }
}
```

ViewModel では、それを表示向けに加工します。

Presenter の代わりに Flow の Observer がおり、データが流れてきたら drop や take など Flow の様々なオペレーターを使い、加工や除外を行います。

MainActivity.kt

```
counterViewModel.showCountEvenNumbersSquared()
```

activity_main.xml

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{viewModel.showing}" />
```

View は、計算実行と表示を担当します。

データの流れは、図 1.1 と同等になります。

では、単方向データフローにするとどう変わるでしょうか。

CounterState.kt

```
sealed class CounterState {
    data class Init(val count: Int = 0) : CounterState()
    data class Success(val count: Int) : CounterState()
    data class Error(val exception: Exception) : CounterState()
}
```

図 1.2 に適合するため、State クラスを作成します。

ただのカウンターに Success や Error を持たせるのは若干大袈裟ですが、何かの処理を行い結果を返す場合の基本構成です。

CounterUseCase.kt

```
class CounterUseCase {
    private val mutableState = MutableStateFlow(CounterState.Init() as CounterState)
    val state: StateFlow<CounterState> = mutableState
    suspend fun countStream() {
        repeat(100) { count ->
            delay(100)
            mutableState.emit(count) // 0 1 2 3 4 ... 99
        }
    }
}
```

Interactor が、無加工のデータを非同期に送ります。

先ほどと違うのは、Flow インスタンスを State クラス型で外部に公開している点です。

StateFlow は、Flow を継承した状態管理用のホットストリームな Flow で、LiveData に似たものです。詳しくは後述しますが、State を導入する上で最も容易な手段なので採用しています。

データ操作は MutableStateFlow でないといえませんが、データ更新をどこからでも行えるのはリスクのある設計なので、MutableStateFlow 型は非公開にします。そのため、CounterUseCase のみが State の更新が可能です。

CounterViewModel.kt


```

class CounterViewModel: ViewModel(
    useCase: CounterUseCase
) {
    private val mutableShowing = MutableStateFlow(String)
    val showing: StateFlow<String> = mutableShowing

    init {
        useCase.state
            .drop(1)
            .filter { it is Success && it.list % 2 == 0 }
            .map { it is Success && (it * it).toString() }
            .take(5)
            .onEach { new ->
                (new as? Success)?.count?.let {
                    mutableShowing.value = it // 4 16 36 64 100
                }
            }
            .launchIn(viewModelScope)
    }

    fun counter() {
        viewModelScope.launch(Dispatchers.Main) {
            useCase.countStream()
        }
    }
}

```

ViewModel が、それを表示向けに加工します。

先ほどと違うのは、ここでも StateFlow を外部に公開している点です。今まで説明した依存関係に従い、公開先は View です。

MainActivity.kt

```

counterViewModel.showCountEvenNumbersSquared()

```

activity_main.xml

```

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{viewModel.showing}" />

```

View は、計算実行と表示を担当します。

データの流れは、図 1.7 と同等になります。

サンプルコードに出てきた ViewModelScope と Dispatcher は、Flow を使う上では理

解を欠かせないキーワードですが、本書では省略します。

1.4.5 SharedFlow と StateFlow

さて、先ほど出てきた StateFlow と、異なる性質を持つ SharedFlow について説明します。

Flow での値の更新は、上記 UseCase の `flow { ... }` ラムダ式中でしかできません。つまり ViewModel 側では値を更新できず、また `.value` のように値の参照もできません。

`subscribe` してる数だけ `flow { ... }` ラムダ式が呼ばれてしまうのも特徴です。

それでは状態保持とか処理リソースの節約には向いてないということで、ホットストリームな Flow として登場したのが、ここで紹介する SharedFlow と StateFlow です。

SharedFlow とは

複数箇所での `subscribe` でデータや状態を共有できる Flow で、処理リソースの節約に向いています。

ただの Flow と違う点は以下です。

```
sharedFlow.onEach {  
    println("1")  
}.launchIn(scope)  
  
sharedFlow.onEach {  
    println("2")  
}.launchIn(scope)
```

- このように複数 `subscribe` してても `flow { ... }` ラムダ式側は 1 回しか呼ばれない
- 処理開始/subscribe 終了のタイミングを選択できるが、これを適切に指定しないと `subscribe` され続ける
 - LiveData に変換し、`observe` 引数に `LifecycleOwner` を設定すれば、表示中だけの `subscribe` も可能
- 色々高機能
 - `replay`: `subscribe` した瞬間、過去の `n` 回の値を受信する
 - `buffer`: 複数 `subscribe` かつ処理に時間がかかるとき、1 回目に行われた処理をバッファリングして、2 回目以降を早くしてくれる

1 つだけの Flow インスタンスを全ての場所で参照し、監視は必要な間だけ動作させる

とか、永続的に監視しつつ、`replay` で最後に発行された 10 個を常に監視するといったトリックが可能です。

StateFlow とは

状態保持に特化した `SharedFlow` です。LiveData に似ています、というか実質の後継機能です。

- 初期値が必須
- 現在の状態を `.value` で受け取れる
- `MutableStateFlow` を使えば、`.value` への代入も可能
 - その際 Coroutines Scope は不要
- `launchIn` で直近の値を 1 つ受信する
- 同じ値の代入は受信しない
- `wait` とかを挟まず連続して値が変更されたとき、最後の値しか受信しない
 - つまり「状態」が「保持」されないと「状態変化」とみなされない

`sharedFlow` では、View を開いたタイミングで flow がサーバー通信などの処理中なら、直近の値をどう表示するのかで迷います。しかし、`stateFlow` では `.value` に直近の値がキャッシュされているので、迷わずに済みます。

初期化方法

`MutableSharedFlow`、`MutableStateFlow` を使って初期化するか、`shareIn`、`stateIn` を使って Flow から変換します。`shareIn` は `sharedFlow` インスタンスを、`stateIn` は `stateFlow` インスタンスを返します。

注意点

関数の戻り値で `shareIn` や `stateIn` をしてはなりません。それをする、関数の呼び出しごとに新しい `SharedFlow` または `StateFlow` が作成され、リソースの無駄遣いになります。

また、ユーザー ID のような入力値を持つ Flow は、異なる入力値で複数回開始した場合、`subscribe` が共有されていると新旧 ID が混じって誤動作するリスクがあります。`shareIn` や `stateIn` で安易に共有してはならないパターンです。

処理開始タイミングの指定

`flow { ... }` ラムダ式の処理開始タイミングは、

1. `shareIn` の `SharingStarted` オプションで、`shareIn` 直後から永続的に開始する `Eagerly`
2. `subscribe` が行われてから永続的に開始する `Lazily`
3. `subscribe` が行われている間だけ有効にする `WhileSubscribed`

を選択することができます。

結局どれがいいのか

…は、場合によって異なります。大事なのは、要件に応じて `SharedFlow`/`StateFlow` を適切に使い分けることです。

どうしても迷うときは、

1. `subscribe` 場所の結果に狂いが生じないこと
2. リソースの無駄遣いにならないこと

を念頭に置いて判断しましょう。

1.4.6 Flow と LiveData、Rx との比較

Android アプリでは、`LiveData` や `RxJava` から `Flow` への置き換えが少しずつ進んでいます。が、`Flow` の何が良いのかわからないまま周りに釣られてやってる方も少なくないのではと思います。

私も最初はその 1 人だったので、`Flow` を使うメリットを見ていきましょう。

従来手法の問題点

まず、`Rx` 系を使う上で避けては通れない問題が、OS のライフサイクルへの適合やオーバーヘッド対策を盛り込むこと、あるいはそれらの考慮抜けによるバグです。

`LiveData` は Android Jetpack の一部なので、Android のライフサイクルやメモリ/キャッシュとの親和性は抜群です。これは初心者のぶっつけ実装でもその類のバグが起きにくいことで `LiveData` の優秀さを実感できると思います。もちろん `LifecycleOwner` の指定間違いのようなものは致命的です。

一方、次々と起こる状態変化の `subscribe` や、`Model` → `ViewModel` のデータ変換部分に注目すると、そこは複雑化やバグの温床を抱えたままです。

Flow 移行のメリット 1. 構造化された並行性

`Flow` にあって `LiveData` にない主な 1 つは、`map` の再計算やデータ変換を `flowOn` により簡単に他スレッドへ投げ、結果だけ UI 側で受け取れることです。

コールドストリームなので、次々と状態変化が起きても無駄なくスレッドを使い破棄もしてくれます。

Flow 移行のメリット 2. さまざまな演算子

map、filter、onEach、reduce など、LiveData には無い多くの演算子で効率的なデータ変換を標準サポートしてくれます。

Flow 移行のメリット 3. テスタビリティ

テストのしやすさも Flow に軍配が上がります。

LiveData + Coroutines のテストはピタゴラスイッチのようなもので、他のテストの干渉を受けないよう上手く作らないと Fail が起きたり、observe のチェーンになってしまいます。

Flow では flowOn する際の dispatcher に TestCoroutineDispatcher を使い、runBlockingTest で走らせることで必ず決まった順序でテストが可能なので、無用意な影響を排除できます。

その順序をチェックする collectIndexed など便利なオペレーターも揃っています。一部だけモックを DI することも容易です。

まとめ

LiveData から Flow に移行すると、無駄のない非同期処理が書けて、いろんな演算子も使え、テストも捗ります。

ただし、Flow は Android ではなく Kotlin の機能なので、Android で使う場合 flowWithLifecycle などの導入を忘れないようにしましょう。

結局、LiveData や Rx より便利なのかは、作ろうとしてるモノが何か次第だと思います。上記のメリットを生かせると判断すれば完全 Flow で、確信がなければ最初は LiveData+Flow という判断で良いと思います。

Android の Flow には asLiveData という変換オペレーターもあります。

1.5 Kotlin Coroutines Flow を Swift で observe する

ここまで Flow について説明しましたが、例えば UI 以外を Kotlin Multiplatform Mobile (以下、KMM) で実装し、UI は Swift で実装する iOS アプリの場合、**Kotlin の Flow は Swift でも受け取れるのか?** という疑問が起きます。

結論としては受け取れますし、Flow の強みである各種オペレーターも、そのままとはいきませんが Swift の事情に合わせる形で容易に利用できます。

では、それを確かめましょう。

1.5.1 環境構築

- 必須要件：Android Studio 4.2 以上 / Xcode 11.3 以上 / macOS Mojave 10.14.4
- 私の環境：Android Studio Arctic Fox 2020.3.1 / Xcode 13.1 (13A1030d) / macOS Big Sur 11.5.2

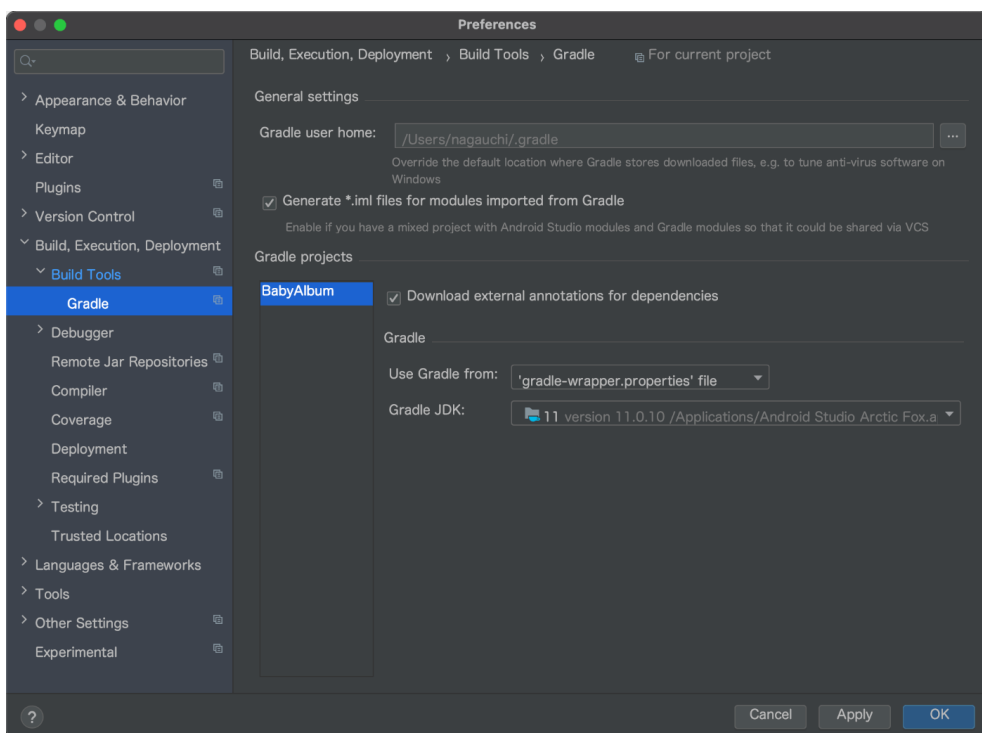
せっかくなので、KMM アプリを 1 から作る手順を載せます。

1. Android Studio で、Android Studio → Preferences → Plugins で、Kotlin Multiplatform Mobile をインストール
2. Android Studio で、File → New → New Project、KMM Application を選択
3. パッケージ名など入れる
4. Add sample tests for Shared module にチェック入れ（本記事ではテストコード書かないですが今後のため）、iOS framework distribution は Regular framework とする
5. フォルダ階層を Android から Project に変える
6. iOS のデバッグ設定を Edit Configurations から行う
7. Execution Target を好みの Simulator に指定
8. Simulator が出てこない場合は Xcode → Window → Devices and Simulators → Simulators タブの+ボタンで追加

これで準備完了。

ちなみに Gradle Setting で JDK 1.8 だとビルド時に `Android Gradle plugin requires Java 11 to run. You are currently using Java 1.8.` というエラーになるので、Android Studio → Preferences → Build, Execution, Deployment → Build Tools → Gradle で Gradle JDK を 11 以上に設定します。

1.5 Kotlin Coroutines Flow を Swift で observe する



▲図 1.8

最後に、`shared/build.gradle.kts` に common で使う依存関係を追加します。

```
val commonMain by getting {
    dependencies {
        implementation("org.jetbrains.kotlin:kotlin-stdlib:1.6.0")
        implementation(
            "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.2-native-mt"){
            version {
                strictly("1.5.2-native-mt")
            }
        }
    }
}
```

`native-mt` を付与しないと、iOS からの Coroutines 呼び出しで落ちます。後々 Ktor などの依存関係を追加する場合に備え、`strictly` も設定します。

1.5.2 実装

この章で作るのは「赤ちゃんに見せるアルバムアプリ」です。

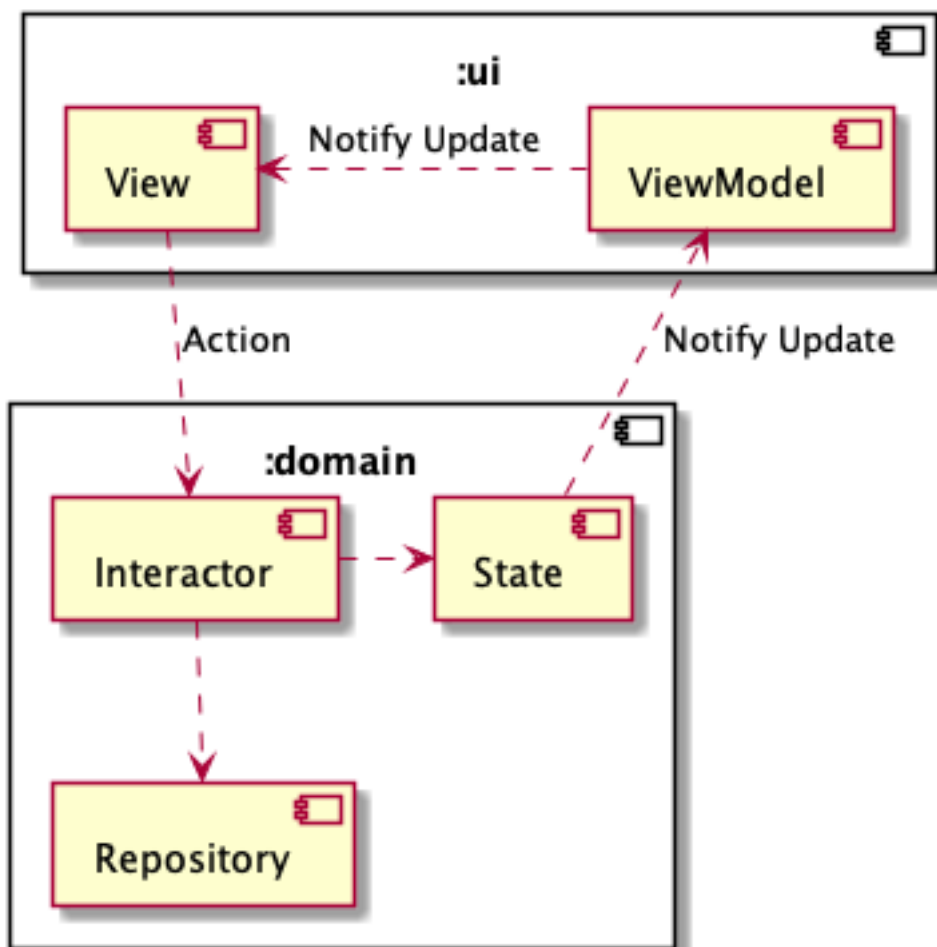
1 歳の娘が、自分やいとこの写真をスマホで見るのが大好きなんです。しかし、OS 標準のフォトアプリで写真を見せてると、手を伸ばして触れていろんな操作をしてしまいます。

たとえば削除してゴミ箱を空にしたりとか、共有を開きメールを開いて画像を誰かに送信なんてことがあると困ります。実際タッチパネルが敏感すぎて予期せぬことが意外と起きます。

なので、アルバムアプリの仕様は次のようにします。

1. 15 秒ごとに異なる画像をランダムに表示する
2. ユーザー入力は一切受け付けない
3. Home ボタンを押さないとアプリを抜けられない

設計はこうします。Repository より先はありません。



▲図 1.9

サーバーやデータ保存まで作り込むと本書の記事が長くなり伝えたい本質がどこかわからなくなるので、極シンプルなプログラム仕様にします。

1. Interactor と Repository は、15 秒ごとに [0-4] の数字をランダムで State に反映
2. ViewModel は、State を Subscribe し、その数字に応じた画像名を View に Publish
3. View は、@Published 変数を Subscribe し、その画像をアセットから見つけて表示

その通りコードを書いていきます。Kotlin のソースコードは全て shared の common-

第 1 章 Flow でクリーンアーキテクチャーを最適化する

Main 階層下です。

SwiftStateFlow.kt

まずは、本記事のメインテーマとなる StateFlow を Swift で使うための SwiftStateFlow を実装します。

```
package com.vitantonio.nagauzzi.babyalbum

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.launchIn
import kotlinx.coroutines.flow.onEach

class SwiftStateFlow<T>(private val kotlinStateFlow: StateFlow<T>) : Flow<T> {
    by kotlinStateFlow {
        val value = kotlinStateFlow.value
        var job: Job? = null

        fun observe(continuation: ((T) -> Unit)) {
            kotlinStateFlow.onEach {
                continuation(it)
            }.launchIn(
                CoroutineScope(Dispatchers.Main + Job().also { job = it })
            )
        }

        fun close() {
            job?.cancel()
            job = null
        }
    }
}
```

呼び元は continuation クロージャの引数でデータを受け取ります。

Flow でなく StateFlow を選択したのは、前回と同じ値を連続発行しないためのチェックを Interactor に設ける予定だからです。

ここで、Flow の豊富なオペレーターを駆使すればいいのでは？ と思われた方は鋭いですね！ ぜひ後述の **Combine** と **Flow** を連動してより使いやすくするところを見てください。

呼び元が破棄されるときに close() を呼ぶと中断もできます。実際、後述の View-Model の deinit で呼んでいます。

PublishNumber.kt (Interactor)

次に、KMM 側の Interactor/Repository/State を実装します。

```
package com.vitantonio.nagauzzi.babyalbum.domain.interactor

import com.vitantonio.nagauzzi.babyalbum.SwiftStateFlow
import com.vitantonio.nagauzzi.babyalbum.domain.repository.NumberRepository
import com.vitantonio.nagauzzi.babyalbum.domain.state.NumberState
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

class PublishNumber(
    private val repository: NumberRepository
) {
    private val mutableState = MutableStateFlow(NumberState.Init(0) as NumberState)
    private val swiftMutableState = SwiftStateFlow(mutableState)

    val state: StateFlow<NumberState>
        get() = mutableState
    val swiftState: SwiftStateFlow<NumberState>
        get() = swiftMutableState

    fun execute(min: Int, max: Int, times: Int) =
        CoroutineScope(Dispatchers.Default).launch {
            repeat(times) {
                delay(15000)
                mutableState.value = NumberState.Updated(
                    repository.getChangedRandom(min, max,
                        before = state.value.number)
                )
            }
        }
}
```

NumberRepository.kt (Repository)

```
package com.vitantonio.nagauzzi.babyalbum.domain.repository

class NumberRepository {
    fun getChangedRandom(min: Int, max: Int, before: Int): Int {
        val random = (min..max).random()
        return if (random == before) {
            getChangedRandom(min, max, before)
        } else {
            random
        }
    }
}
```

NumberState.kt (State)

第1章 Flow でクリーンアーキテクチャーを最適化する

```
package com.vitantonio.nagauzzi.babyalbum.domain.state

sealed class NumberState(open val number: Int) {
    data class Init(override val number: Int) : NumberState(number)
    data class Updated(override val number: Int) : NumberState(number)
}
```

この程度なら Interactor/Repository/State に分けるメリットがあまり無いのですが、行く行くはサーバーから画像をダウンロードして ByteArray か何かで Interactor に渡し、State は Init/Success/Failure に分かれることを想定しているので、今のうちにと分けてしまいました。

AlbumViewModel.swift (ViewModel)

さて、ここからは Swift 側です。

KMM のプロジェクトを作るだけで SwiftUI がデフォルト生成されているので、まず ViewModel を作ります。

```
import shared

class AlbumViewModel: ObservableObject {
    private let photos = (1...5).map { "BabyImage\($0)" }
    private let interactor: PublishNumber

    @Published var photoName: String

    init(interactor: PublishNumber) {
        self.interactor = interactor
        self.photoName = self.photos[0]
        interactor.swiftState.observe { newState in
            if 0...4 ~= newState!.number {
                self.photoName = self.photos[Int(newState!.number)]
            } else {
                fatalError("newNumber isn't supported number")
            }
        }
    }

    deinit {
        self.interactor.swiftState.close()
    }
}
```

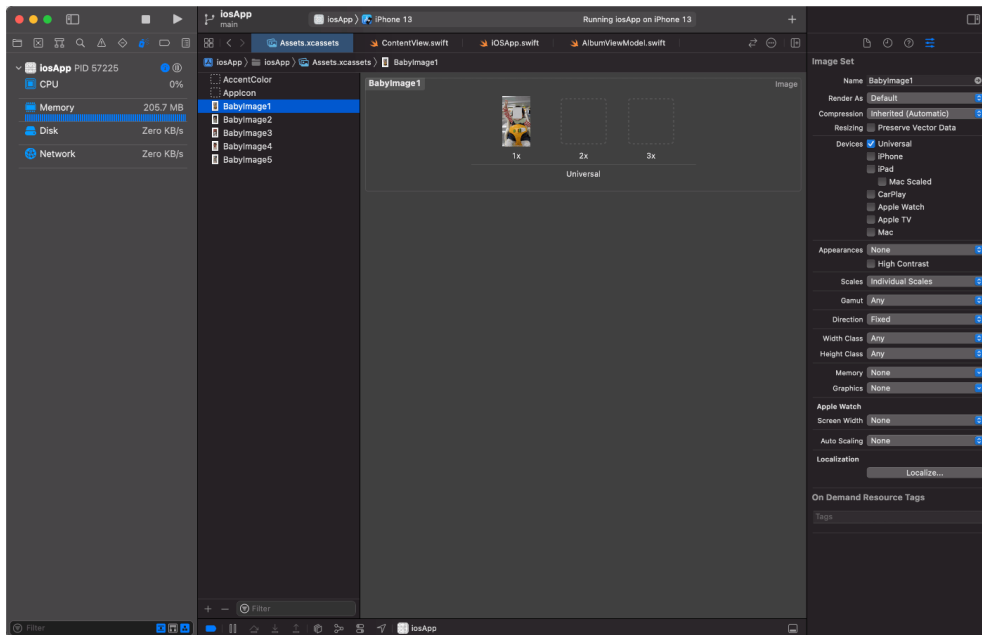
先頭の `import shared` で、KMM で作った `SwiftStateFlow` や `Interactor` にアクセスできるわけです。始めれば難しくないですよ、KMM！

そして、`observe` が Flow を監視・データ受信する部分です。画像名に変えて `photoName` に入れると、`@Published` が付いているので View の更新トリガーの役割を果たしてくれます。

`photos` が配列で画像名を持っていますが、実際の画像は `BabyImage1`～`BabyImage5`

1.5 Kotlin Coroutines Flow を Swift で observe する

を asset に登録済みです。



▲図 1.10

ContentView.swift (ViewModel)

最後に、ContentView をデフォルトから変更します。これが設計図の View にあたります。

```
import shared

struct ContentView: View {
-   let greet = Greeting().greeting()
-
+   @ObservedObject var viewModel: AlbumViewModel
+   let interactor: PublishNumber
+
+   init() {
+       self.interactor = PublishNumber(repository: NumberRepository())
+       self.viewModel = AlbumViewModel(interactor: interactor)
+   }
+
+   var body: some View {
```

```
-      Text(greet)
+      ZStack {
+          Color.black
+          .ignoresSafeArea()
+          Image(viewModel.photoName)
+          .resizable()
+          .aspectRatio(contentMode: .fill)
+      }.task {
+          self.interactor.execute(min: 0, max: 4, times: 100)
+      }
+  }
```

1.5.3 プレビューと実行

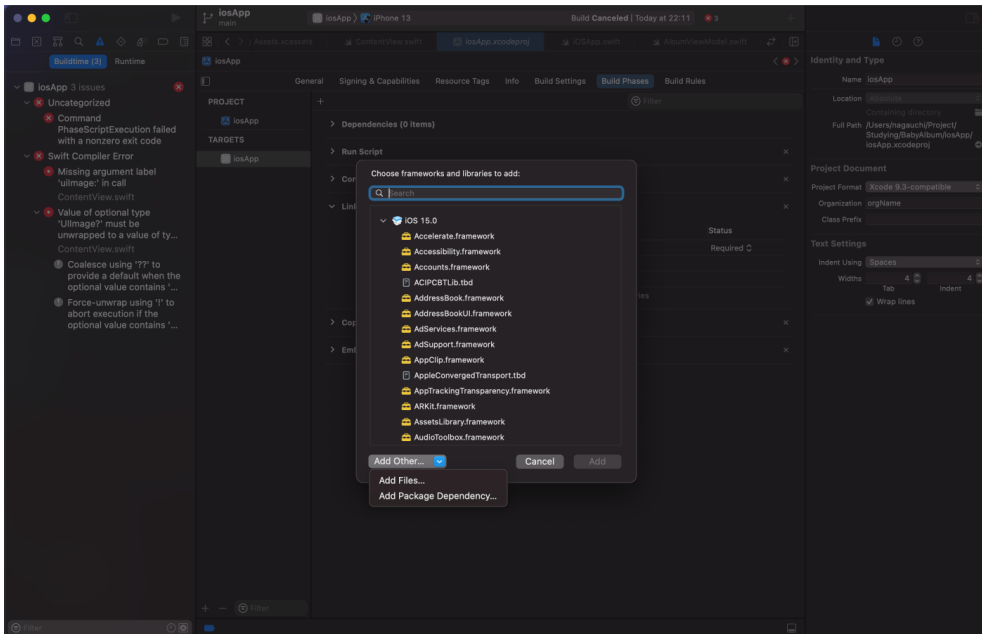
ここで、SwiftUI のプレビューが動いていないことに気づきました。

Android Studio 側で Run するとビルドできるのですが、Xcode 側ではまだ xcframework をインポートしてないので、そのビルドエラーによってプレビューやデバッグ、エラー調査などができないのです。

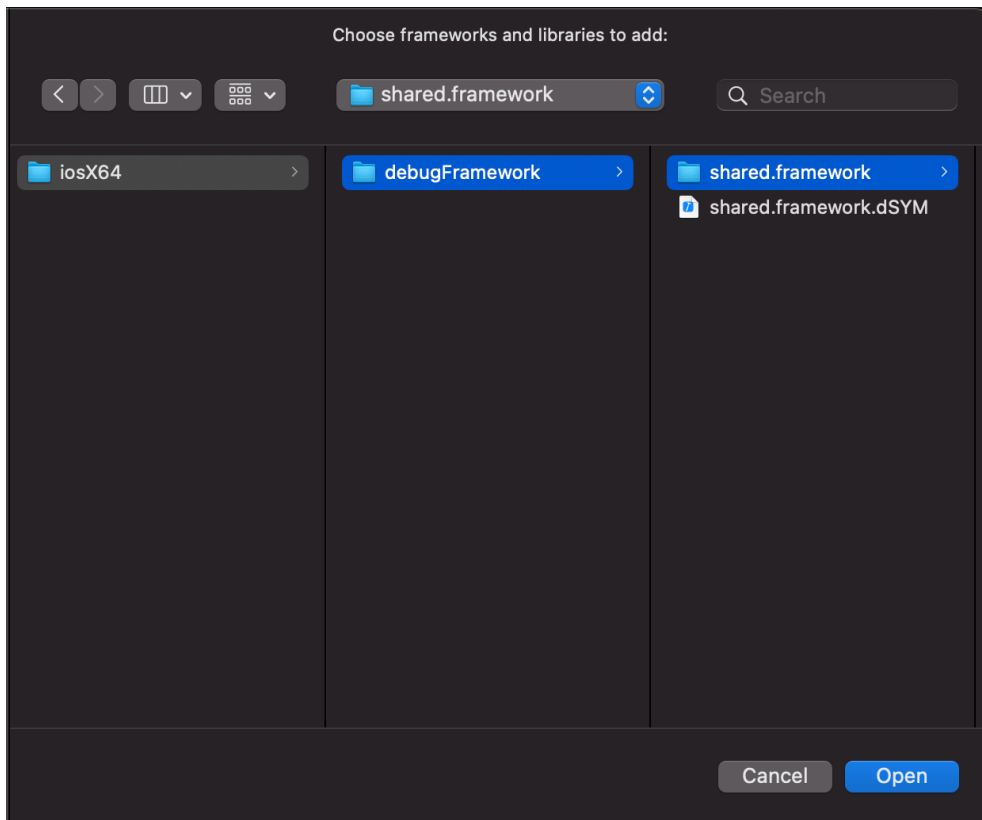
Xcode でそれらができたら開発効率が上がるので、直しましょう。

iosApp の TARGETS → Build Phases → Link Binary With Libraries で、`./BabyAlbum/shared/build/bin/iosX64/debugFramework/shared.framework` を追加します。

1.5 Kotlin Coroutines Flow を Swift で observe する



▲ 1.11

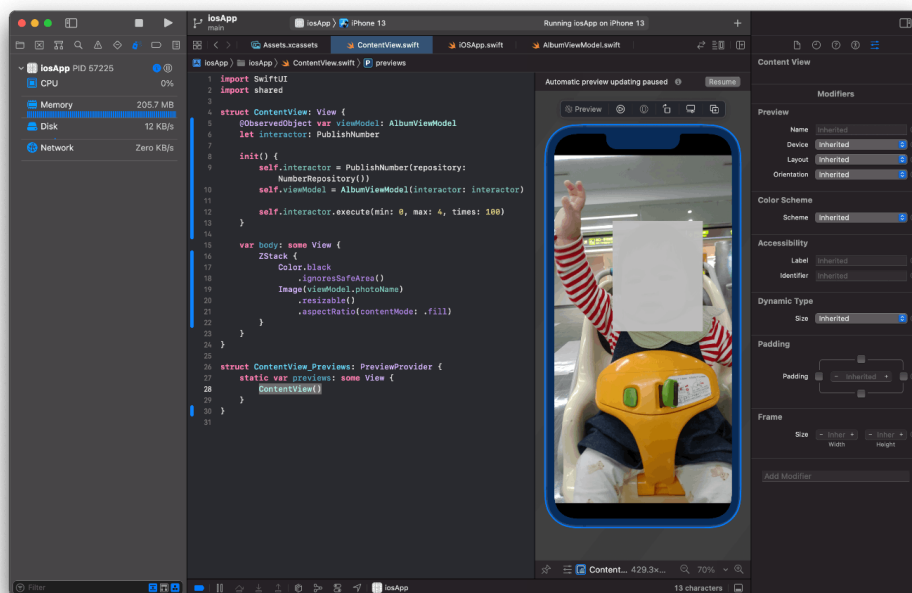


▲ 図 1.12

ただし、とりあえずビルドを通したいだけなので、CPU アーキテクチャーによっては上記出力先の framework ではダメなこともあります。

いちいち設定を弄るのは面倒なので、実際に動かすためのビルドは Android Studio からがよいでしょう。

これでひとまず仕様は満たせました。



▲図 1.13

プレビューもされてるし、実行すると 15 秒毎に違う画像に変わります。これは Swift-StateFlow が 15 秒毎の Flow のデータ送出を Swift にうまく伝えているからです。

iPad に入れて赤ちゃんに見せてあげたいと思います

1.6 Combine を使って改善する

さて、ここまでで満足かという、「本当に作りたいものはこれじゃない」感があります。

SwiftStateFlow が onEach を逐次横流ししてるだけじゃん！ みたいな。

Flow の強みである様々なオペレーター (map、filter、onEach、reduce など) を使いたいのです。

とはいえ Flow と同じオペレーターを Swift で 1 から実装するのは大変だし、shared.h に定義されてる Kotlinx.coroutinescore 系の型を使って Flow の扉をこじ開けるのもちょっと根気が必要そうです。

そもそも、無理して Flow に合わせるより、Swift なら Swift らしい実装をしたいですね。

そこで、Swift 標準の非同期フレームワークである **Combine** の出番です！
AlbumViewModel を以下のように書き直します。

AlbumViewModel.swift (ViewModel)

```
import Combine
import shared

class AlbumViewModel: ObservableObject {
    private let photos = (1...5).map { "BabyImage\($0)" }
    private let interactor: PublishNumber
    private var cancellable: Cancellable?

    @Published var photoName: String

    init(interactor: PublishNumber) {
        self.interactor = interactor
        self.photoName = self.photos[0]
        self.cancellable = NumberStatePublisher(stateFlow: interactor.swiftState)
            .map { newValue in
                self.photos[Int(newValue.number)]
            }
            .assign(to: \.photoName, on: self)
    }

    deinit {
        self.cancellable?.cancel()
    }
}

public struct NumberStatePublisher: Publisher {
    public typealias Output = NumberState
    public typealias Failure = Never

    private let stateFlow: SwiftStateFlow<Output>

    public init(stateFlow: SwiftStateFlow<Output>) {
        self.stateFlow = stateFlow
    }

    public func receive<S: Subscriber>(subscriber: S)
        where S.Input == Output, S.Failure == Failure {
        let subscription = NumberStateSubscription(stateFlow: stateFlow, subscriber: subscriber)
        subscriber.receive(subscription: subscription)
    }
}

final class NumberStateSubscription<S: Subscriber>: Subscription
    where S.Input == NumberState, S.Failure == Never {
    private let stateFlow: SwiftStateFlow<S.Input>
    private var subscriber: S?

    public init(stateFlow: SwiftStateFlow<S.Input>, subscriber: S) {
        self.stateFlow = stateFlow
        self.subscriber = subscriber
    }
}
```

```
        stateFlow.observe { newValue in
            _ = subscriber.receive(newValue!)
        }
    }

    func cancel() {
        subscriber = nil
        stateFlow.close()
    }

    func request(_ demand: Subscribers.Demand) {}
}
```

Combine の仕組みは省略しますが、必要な Publisher と Subscription を作りました。

最も肝心なのは `func startSubscribe()` の部分です。受け取った数字を `map` で文字列加工して、直接 `photoName` にバインドしています。この数行だけでロジックから流れてきたデータを View 向きに加工して再描画まで行っていることを考えれば、とてもシンプルですね。もちろん `map` 以外の Combine オペレーターも使用可能です。

これが標準フレームワークのオペレーターを使用できる大きな利点です。改良前の `onEach` を横流しするだけの `ViewModel` もコード量は少ないのですが、複雑な実装になると `observe` 部分が膨れ上がり、じわじわと痛めつけられるでしょう。

また、Publisher を何かに `assign` すると `Cancellable` オブジェクトが取れるので、その参照を取っておいて `ViewModel` が解放される際に監視を止めることも可能です。改良前は直接 `Interactor` を介して `SwiftState` の `cancel` を呼ぶ必要があり、複数箇所監視している場合の考慮ができていませんでした。

1.7 まとめ

まとめる力などもう残ってない（笑）2022/1/11(火) に続きを書きます。

第 2 章

2 章タイトル

第 3 章

3 章タイトル

第 4 章

4 章タイトル

第 5 章

5 章タイトル

第 6 章

6 章タイトル

第 7 章

7 章タイトル

著者紹介

第 1 章 **tonionagauzzi / @tonionagauzzi**

Smartphone App Engineer, "You decide you're happy or not."

第 2 章

.....

第 3 章

.....

第 4 章

.....

第 5 章

.....

第 6 章

.....

第 7 章

.....

ACCESS テックブック 2

2022 年 1 月 22 日 ACCESS テックブック 2 v1.0.0

著 者 ACCESS 技術書典同好会

編 集 tonionagauzzi

発行所 ACCESS Co., Ltd.

(C) 2022 ACCESS Co., Ltd.