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、お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ろうとする今、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

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れ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、データの流れをわかりやすくするために Kotlin Coroutines 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、そのロジックを Kotlin Multiplatform Mobile のフレームワークとしてアプリに組み込みます。Kotlin を標準サポートする Android アプリではなく、あえて SwiftUI ベースの iOS アプリを選び、X-Platform への適合を確認します。また、Kotlin の Flow が Swift にはどんな形で流れるのか、単方向データフローや Concurrency は保たれるのか、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。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

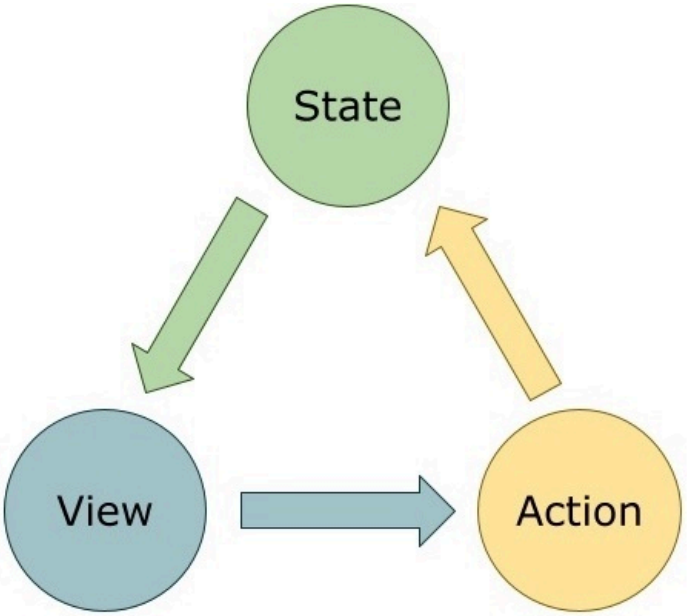
5

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っ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 なことからも、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

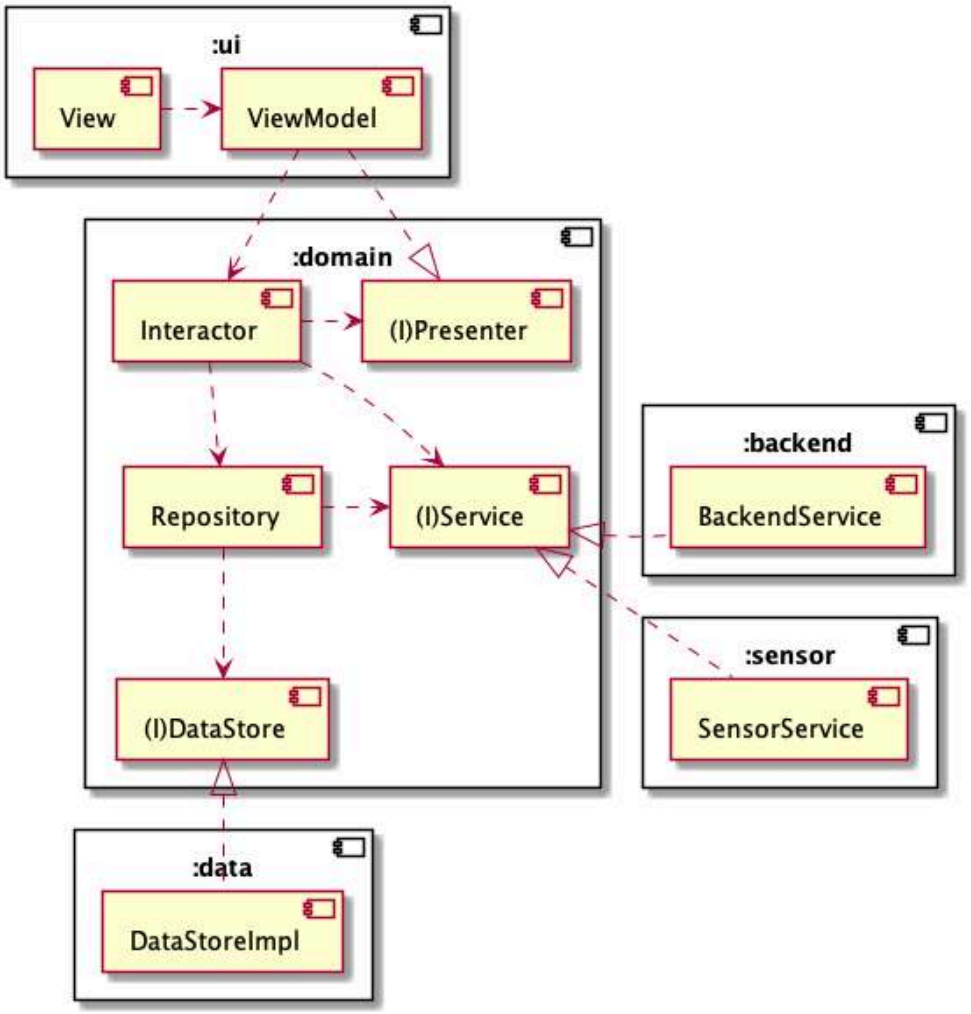
iOS アプリは、MVC や MVC+VM のようなアーキテクチャがよく使われてきました。

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る印象です。

1.3　クリーンアーキテクチャーと単方向データフロー

続いて、クリーンアーキテクチャー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ーの依存関係

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

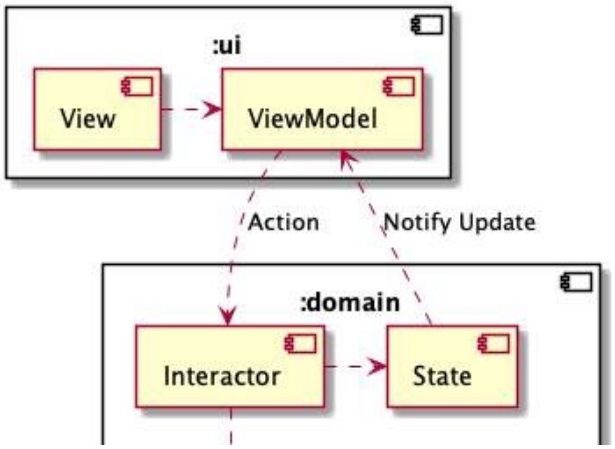
します」（to UI）とし、domain から他への依存を無くすのが、改善前のクリーンアーキテクチャーの特徴です。

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。

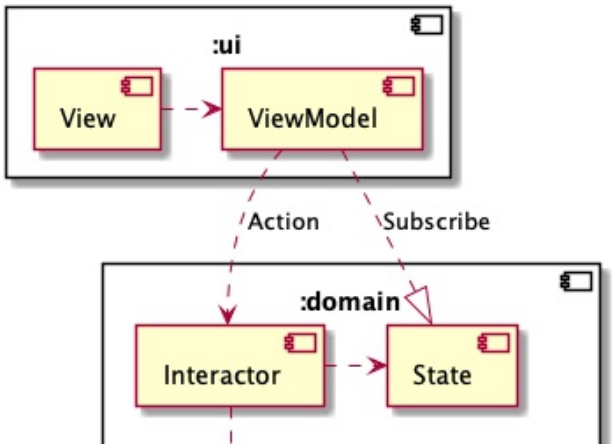
その Presenter をやめ、UI から domain に渡すものを Action、domain から UI には State の更新を通知するようにしてみます。



▲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。

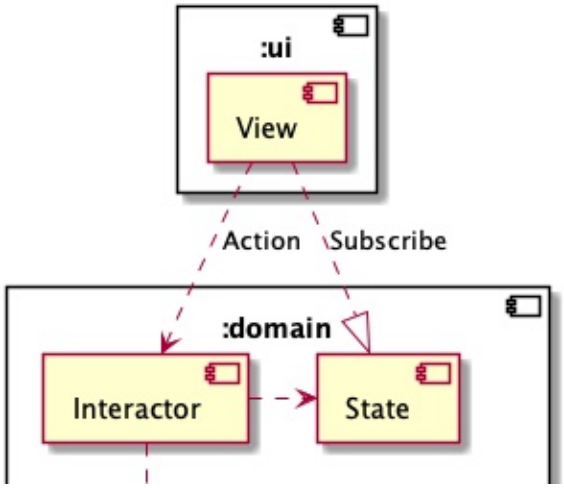


▲図 1.5　単方向データフローと DIP を適用した UI と domain 間の依存関係

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

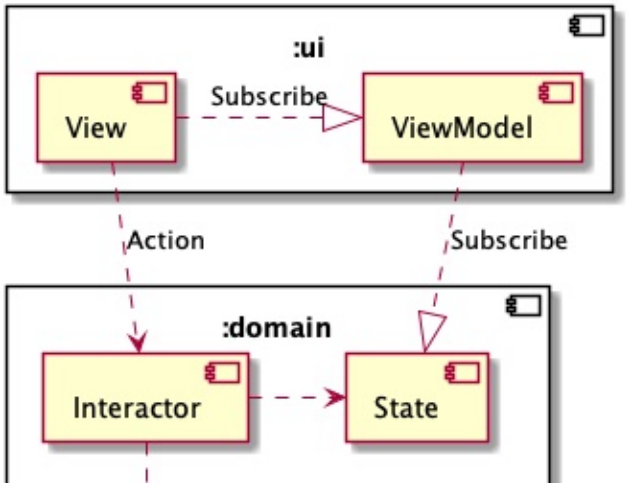
1
1

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 として分けてた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 から切り離し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 Kotlin Coroutine Flow を用いていきます。

まず、説明です。Kotlin Coroutine 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であることを意味します。メモリ効率的には良くない面もあるのです。

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。こちらも 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

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

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

ViewModel がデータを受け取り、表示向けに加工します。

```
val count = MutableLiveData<String>()

fun counter() {
    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
    }
}
```

それを View が表示します。

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

以下が、その全体図です。（画像にする予定）

```
View    |  ViewModel    | UseCase〜深層
-----------------------------------------------
        |               |
イベント  →  Coroutine起動 → 時間のかかる処理（非同期）
        |               |       ↓↓↓
描画  ←←←  出力データ加工 ←←← 出力データ送信（複数回）
        |               |
```

Flow にはオペレーターがたくさんあり、この図で言う左向き時のデータ加工に優れています。

続きは以下 https://qiita.com/tonionagauzzi/items/12aa1a4400256cece72c#viewmodelscope%E3%81%AE%E8

1.5　Kotlin Multiplatform Mobile で UI 以外を実装する

執筆中

1.6　SwiftUI に KMM を取り込む

執筆中

著者紹介

第 1 章 tonionagauzzi / @tonionagauzzi

Smartphone App Engineer

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.