

良いアウトプットのためのインプットとノート整理

酒井文也 (Fumiya Sakai) Twitter & github: @fumiyasac

はじめに

技術的な知識を効果的に学ぶためには、単に情報を受け取るだけでは十分ではありません。真の理解は、得た情報を自分なりに噛み砕き、視覚的に整理することから生まれます。私のエンジニアリング学習において最も大切にしているのは、調査した技術トピックを手書きでノートにまとめるというアプローチです。

このノート作成は、ただの書き写しではありません。サンプルコードの実装における重要なポイントを、自分の視点で抽出し、言葉にすることで、技術的概念の本質的な理解に近づきます。手で書くという行為は、デジタル的な記録とは異なる、より深い思考のプロセスを引き出してくれます。指先で情報を紡ぐことで、抽象的な概念を具体的な知識に変換できるのです。

例えば、

- コードの背後にいる設計思想
- アルゴリズムの論理的な構造
- 実装上で工夫しているポイント

等、書籍やサンプル実装コード、動画教材等から得られた情報を自分なりの言葉や図解でノートに記録することで、技術の表面的な理解から、より深い洞察へと成長できると思います。

情報を受け取るだけでなく、自分なりの学びの地図を描くこと。それが、私の技術学習における最も大切な方法だと考えています。

進める際の個人的ポイント

エンジニアとして、技術的な知識を学ぶ際に大切にしているのは、ただ情報を集めるだけではなく、その本質を理解し、将来活用できる形で整理することです。

私の学習スタイルは、様々なリソースから得られる情報の断片を、パズルのピースのようにつなぎ合わせていくアプローチです。書籍、サンプルコード、オンライン講座、動画教材など、多様な情報源から学びを得ながら、それぞれの知識がどのようにつながっているのかを、できる限り丁寧に紐解いていきます。

新しい専門用語や技術的概念に出会うたびに、徹底的に調べて理解を深めます。関連する事例や周辺知識も積極的に探求し、自分の技術的な知識の幅を広げていきます。複雑な概念は、できるだけ簡潔な図解や視覚的な表現に置き換えることで、より直感的に理解しやすくなります。

手書きのノートは、私にとってはこの学習プロセスの中心的なツールです。単なるメモ帳ではなく、後で見返したときに、すぐに記憶が呼び起こせるような、自分だけの知識マップを作り上げる様なイメージでしょうか。ここで大切なのは、このノートを通じて得た知識が、将来の技術的な課題や問題解決に、柔軟かつ実践的に活用できる形で整理されていることです。

情報を取り入れ、整理し、そして最終的に自分のものとして活用できるようにする。

このサイクルが、私にとって技術者としての成長を支える原動力となります。

私にとってノート作成は、技術的な知識を生きた知恵へと変換するための方法かもしれません。

実践したメリット

技術を学ぶ上で大切なのは、ただ情報を集めるだけではなく、その本質を理解することです。私は、手書きのノートを通じて、技術的な知識を深く、そして自分なりに理解する方法を見つけてきました。

サンプルコードや技術資料から重要なポイントを丁寧に抽出し、整理していくこのアプローチは、表面的な学習を超えて、技術の本質に迫ろうとするものです。理論的な理解と、実際の実装イメージを同時に育むことで、単なる暗記とは異なる、深い学びを実現しています。

一見すると時間のかかる学習方法に思えるかもしれません。しかし、じっくりと知識を自分のものにしていくこのプロセスには、大きな利点があります。技術的な概念を自分なりに咀嚼し、視覚化することで、似たような課題に直面したときにも、落ち着いて対応できるようになります。

手書きでノートをまとめる行為は、ただの記録以上の意味を持つと考えています。得られた知識は、仕事や個人的な開発において、柔軟に活用できる「引き出し」となります。特に、実装イメージが似ている技術的な課題に取り組む際には、以前の学習が大きな助けとなり、問題解決のスピードを上げることができます。

デジタル技術が急速に進化する現代において、情報の洪水の中で本当の理解を得ることは簡単ではありません。このような学習方法は、ただ知識を詰め込むのではなく、技術の本質を理解し、実践的な応用力を育む、とても効果的なアプローチだと考えています。

エンジニアとして成長し続けるためには、自分に合った学習方法を常に模索し、改善していくことが大切です。手書きのノートを通じた学習は、そんな個人的な学びの探求の一つの形と言えるでしょう。

入力時に改行をすると、入力エリミネーターが伸縮する TextView を考えてみる

例とした OSS :

ResizingTextView の動きを見てみる

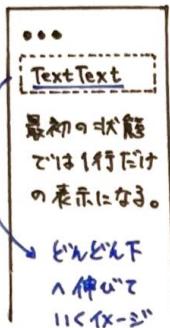
* Android の様なテキスト入力フォームを実現する形をつくる事ができる。



struct PreviewWrapper : View {

④ State var sampleText = "" 入力する内容と View で sampleText で Binding する

※ 自動でサイズを調整する形



⇒



この様に下へ入力エリミネーターがどんどん伸びていく
④ Binding で ④ State と 結合する形にする
ResizingTextView(text: \$sampleText, ① isScrollable: true, ② canHaveNewLineCharacters: true) ③
次の行で、文字数が三行を超えた時表示する
その他にも、最大行数を制限することや、Font サイズや色を設定
行数は三行まで、多くの文字を入力すると、Scroll 可能になります

TextView がベースにある状態

(UIKit で作ってみる)

class CustomTextView : UITextView { ... }
↳ サイズ調整用の属性 property を定義する

④ MainActor struct
TextView : UIViewRepresentable
で SwiftUI で使える形にする

ホイントは、Binding<String> を渡していくこと * そのためも、initializer で設定できるのは
(@Binding private var text: String) 超意を満たすための設定を渡している。

* func makeUIView(context: Context) -> CustomTextView context.coordinator → coordinator の中に
→ CustomTextView で 1:2:3 で作って、view.delegate = context.coordinator ① での実装を定義
updateUIView(view, context: context) ②

* func updateUIView(_ view: CustomTextView, context: Context)
→ この部分で実行しているのは、様々な設定値の反映を行っている

実際のホイントを見ながら、ミニマルな形で実現できるか?

* Coordinator 関連の処理 func makeCoordinator() -> Coordinator {
Coordinator(self)}

final class Coordinator: NSObject, UITextViewDelegate {
• var parent: UITextView (UIViewRepresentable の要素)
• var selectedRange: NSRange? (選択範囲の情報)

* updateUIView 处理内で、View 要素に渡す Inset 値を更新することで、全体 Layout を更新している。

func textViewDidChange(_ textView: UITextView) {

(1) テキスト反映 (2) canHaveNewLineCharacters の反映

(3) 選択範囲 (selectedRange) の反映

}

c.f. Androidで入力量に応じて、フィールドの高さが変化する TextFieldを作成する方針
(with Jetpack Compose)

`This is my first testing.` → `This is my first testing. Unit testing is one of most import`

改行や文字入力をしても高さが変わってしまう。
Taがっていい。

`singleLine = false, maxLines = 未設定`で可!!

本題：iOS & SwiftUIでSimpleな形で作る
※細かな部分は要調整ではあるが、
基本の動きを作るのはこんな感じ？

※ 基本は、UITextViewを利用

べーとなるText入力用のView要素

```
struct WrappedTextView : UIViewRepresentable {
    typealias UIViewType = UITextView
    @Binding var text: String
    let textDidChange: (UITextView) → Void
    func makeUIView(context: Context) -> UITextView {
        let view = UITextView()
        view.isEditable = true
        view.delegate = context.coordinator
        return view
    }
}
```

```
class Coordinator: NSObject, UITextViewDelegate {
    @Binding var text: String
    let textDidChange: (UITextView) → Void
    init(text: Binding<String>, textDidChange: @escaping (UITextView) → Void) {
        self._text = text
        self.textDidChange = textDidChange
    }
    * UITextViewDelegateでの処理をつなげる
    func textViewDidChange(_ textView: UITextView) {
        self.text = textView.text
        self.TextDidChange(textView)
    }
}
```

ホバー modifier等で指定していない場合は自動的に高さが調整される（制限する時は`maxLines`を利用）

```
Box(modifier = Modifier.fillMaxSize(),
    contentAlignment = Alignment.TopCenter) {
    var sampleText by remember {
        mutableStateOf("....")
    }
    OutlinedTextField(
        value = sampleText,
        onValueChange = { sampleText = it },
        modifier = Modifier.padding(20.dp)
            .width(20.dp),
        ) singleLine = false, maxLines = 5
    ) 複数行を許可する MaxLines値
```

```
func updateUIView(
    uiView: UITextView,
    context: Context) {
    uiView.text = self.text
    DispatchQueue.main.async {
        self.textDidChange(text)
    }
} closureを実行する(更新)
```

```
func makeCoordinator() -> Coordinator {
    return Coordinator(text: $text,
        textDidChange: textDidChange)
}
```

* View ※他のView要素での使用例

WrappedTextView(text: \$text,
textDidChange: self.textDidChange)

- frame (height: height ?? minHeight)

↓ ④ Stateで持つ高さ

private 良軟を定義

self.height = max(
textView.contentSize
.height, minHeight)

\$textの部分は
④ Binding var
text: String
で定義する。

※ UITextViewの高さ
を調整している。

No. 2

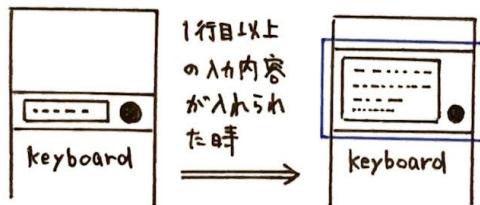
```

struct ExpandingTextView : View {
    * 前述したコードの全体はこの様になります。
    ④ Binding var text : String
        * UITextView の入力値と連動する所
    let minHeight : CGFloat = 150 → 初期の高さ値
    ④ State private var height : CGFloat?
        * 入力時の UITextView の高さを取って、反映する仕組み
    var body : some View {
        WrappedTextView(text: $text, textDidChange: self.textDidChange)
            .frame(height: height ?? minHeight)
    }
}

private func textDidChange(_ textView: UITextView) {
    self.height = max(textView.contentSize.height, minHeight)
}
    ④ State 値を更新して、View を再レンダリングする

```

Message 入力欄アリの入力機能を考える Auto Resizable TextField Using Swift



* keyboard が閉じている時は。
入力用の CustomTextField() は。
一番下にひつついで配置される
placeholder 表示等で工夫が必要

入力欄の広さは。
• lineLimit(n) で
実現可能である

CustomTextField() という名前で要素を作成（あくまでベースは TextField）

<画面の構造例はこんな感じ>

```

struct ContentView : View {
    var body : some View {
        NavigationStack {
            VStack {
                Spacer() * Spacer() を入れる
                CustomTextField()
                    * Input Field
                Spacer() * 重ねづけ。下に <3
            }
            .background(.orange)
        }
        .onTapGesture {
            hideKeyboard()
        }
    }
}

    画面全体を Tap すると、
    keyboard を閉じる

```

SwiftUI の View Extension を定義

```

func hideKeyboard() {
    * keyboard を閉じる
    let resign = #selector(UIResponder.resignFirstResponder)
    UIApplication.shared.sendAction(resign, to: nil, from: nil,
                                    for: nil)
}

```

```

func placeholder<Content : View>(
    when shouldShow : Bool, * 表示可否
    alignment : Alignment = .leading, * 対象位置
    ④ ViewBuilder.placeholder : () -> Content) -> some View {
    ZStack(alignment: alignment) {
        placeholder().opacity(shouldShow ? 1 : 0)
        self
    }
}
    第3引数には placeholder へ表示する内容を設定

```

* CustomTextField は下記の様に定義する

```
struct CustomTextField: View {
    @State var message: String = ""
    var body: some View {
        HStack(alignment: .bottom) {
            HStack(spacing: 8.0) {
                withAnimation(.easeInOut) {
                    TextField("", text: $message, axis: .vertical)
                        .placeholder(when: message.isEmpty) {
                            Text("Message ....")
                                .foregroundColor(.secondary)
                        }
                        .lineLimit(7) → 最大7行まで表示する
                }
            }
        }
    }
}
```

\$message は String で Binding<String> となるので、入力を連動する形になる

※ padding / background / cornerRadius 等を設定する

送信ボタンの場合分け

```
if message != "" → Validation の種類は仕様次第
    (1) 送信可能なボタンを配置する →
    } else {
        (2) 押下できないタグマークボタンを配置する
    }
```

※ padding / background 等を設定する

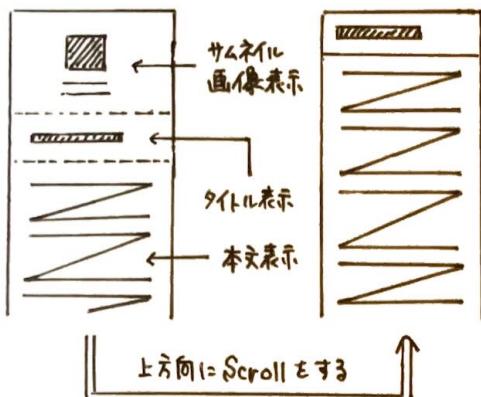
- frame(maxWidth: .infinity)
- frame(minHeight: 55.0) → 最小の高さを設定する
- ignoresSafeArea()
- animation(.easeInOut(duration: 0.3), value: message)

Animation の秒数は短くても良さそう

Diagram:

The diagram illustrates the visual structure of the CustomTextField. It shows a main HStack containing a nested HStack. The nested HStack has an alignment of .bottom and a spacing of 8.0. Inside this nested HStack is an HStack that contains a TextField and a Button. The TextField has a placeholder "Message" and a line limit of 7. The Button is positioned to the right of the TextField.

- Coordinator Layout + Nested ScrollView の組み合わせ



Scroll の位置に応じて見た目が変化する動き

- CoordinatorLayout → 親の Component
 - AppBarLayout
 - Collapsing Toolbar
 - ToolBar
- ↓
- Header 領域
- ConstraintLayout
 - Nested ScrollView
 - TextView
- ↓
- Contents 領域

ネストスクロールをするの実装でのポイント

- ↳ CoordinatorLayout
- ↳ AppBarLayout
- Header ↳ ToolBar
- ↳ Nested ScrollView
- ↳ ScrollView
- Content ↳ LinearLayout
- ↳ TextView

iOS と比べて大きく異なる点

- Android は提供されてる Component を上手く組み合いで、対応していくイメージ
- 自分で実装の微調整をせずとも実現する余地はある

CollapsingToolbarLayout 内に配置した要素に関する設定するポイント

* enterAlwaysCollapsed / exitUntilCollapsed を選択時

① <CollapsingToolbarLayout> で、運動部分を囲む。(ToolBar サイズを変動)

② app:layout_scrollFlags 属性を付す

③ 運動部分を app:layout_collapseMode 属性を付す

Android View を利用する場合は、要素の階層構造に注意が必要。

- スクロールと連動する Toolbar
- app:layout_scrollFlags 属性で Scroll の向き方が異なる。
ToolBar が縮小される場合

- ① enterAlways → 下 Scroll 時消失、上 Scroll 時出現、サイズ不変
- ② enterAlways - Collapsed → 下 Scroll 時消失、上 Scroll 時は上端到達時出現、サイズ変動
- ③ exitUntil - Collapsed → 下 Scroll 時は一部だけ出現、上端到達時完全出現、サイズ変動
- ⇒ enterAlwaysCollapsed / exitUntilCollapsed では、運動部分を <CollapsingToolbarLayout> で囲ってあげること
Scroll に運動して Toolbar のサイズを変更できる Layout 部品

* CollapsingToolbarLayout について

Scroll に運動して Toolbar のサイズを変更する Layout 部品

↳ CoordinatorLayout app:layout_scrollFlags
↳ AppBarLayout ↳ を付与する

↳ CollapsingToolbarLayout
↳ Toolbar

↳ (Scroll 対象の部分)

app:layout_collapseMode を付す

※ enterAlwaysCollapsed / exitUntilCollapsed モードを選択時の Toolbar 対応

- ↓ ↓
 - 下に Scroll した時:
 - 消失
 - 上に Scroll した時:
 - 上端到達時のみ出現
 - 下に Scroll した時:
 - 部分出現
 - 上に Scroll した時:
 - 上端到達時のみ完全に出現

(* どちらも Toolbar のサイズ)
は変動するところになる。)

* exitUntilCollapsed を選択した時の流れとポイント

⇒ サイズの縮小下限を指定する必要がある

↓ (特別な Mode)
app:layout-collapseMode 属性の分類

- pin: 縮小時の表示 Contents は全て表示
縮小時の Header 移動はそのまま
上方へ移動する
- parallax: 縮小時の表示 Contents は
タイトルのみ表示
縮小時の Header 移動は
視差方式で上方へ移動
- none: 縮小時の表示 Contents は
タイトルのみ表示
縮小時の Header 移動は
そのまま上方へ移動する

<全体的な設定例> * XML での例

↳ CoordinatorLayout → 全体で共通化
↳ AppBarLayout → サイズの初期値設定

↳ CollapsingToolbarLayout

app:layout-scrollFlags =
"scroll | enterAlwaysCollapsed"

↳ Toolbar

app:layout-collapseMode = "pin"

↳ NestedScrollView

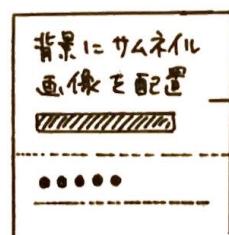
app:layout-behavior = "@string/
appbar_scrolling_view_behavior"



CoordinatorLayout と
連携するためには必要な設定

サイズ縮小
の下限
を指定する

例. どの様な構成になる?



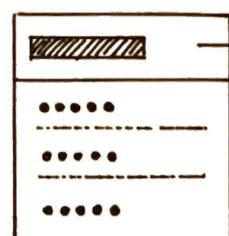
※ タイトルは、■■■■■ の部分

↳ CollapsingToolbarLayout を
利用した構造を取る方が
ポイントになる
(重なった状態になる)

- * 条件を満たす
- scrollFlags
- collapseMode

を組み合せ

※ 重ねる
注意



↑ 上方向に Scroll をしていく。
Toolbar にタイトルが表示
された状態となる。

Contents 表示部分は、
ScrollView や RecyclerView
を活用して構築する

Item や記事 Detail 画面
でよく見る表現の例

↳ CoordinatorLayout

↳ AppBarLayout

↳ CollapsingToolbarLayout
app:layout-scrollFlags =
"scroll | exitUntilCollapsed"

• ImageView

app:layout-collapseMode =
"parallax"

• Toolbar

app:layout-collapseMode =
"pin"

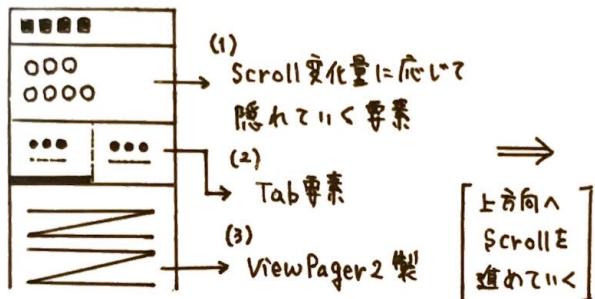
↳ NestedScrollView

↳ RecyclerView

応用する：たとえ StickyTabBar の様な Layout を作る

- ある地点まで到達すると、Tab が折りたたむ①
- ↓ Tab が Toolbar に隠れる②
- Contents 表示部分は、ViewPager2 で作成③
- CoordinatorLayout + AppBarLayout の組み合わせ④

* ①の挙動は工夫が必要になる部分
→ Component の組合せだけでは難しい部分

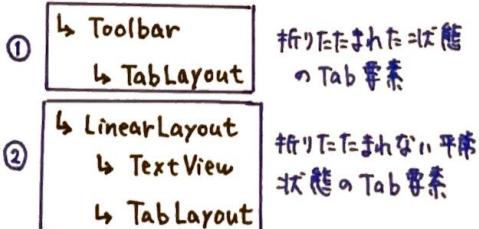


* TabLayout の高さを変えるのではなく。
表示の異なる TabLayout を 2 種用意する

↳ CoordinatorLayout

↳ AppBarLayout

↳ CollapsingToolbarLayout



↳ ViewPager

→ offset 値の変更があった時に発火する処理

* binding.appBarLayout.addOnOffsetChangedListener 内の処理ポイント

isCollapsed : Boolean? E private; 定義

(この値を元にどちらの Tab を使うかを決定する)

override fun onOffsetChanged 内での処理

```
val shouldCollapse = appBarHeight + verticalOffset
= collapsedTabHeight
```

offset 値の変化量を
加算していく

if (shouldCollapse == isCollapsed) { return }

* 变数を切替える時は以降の処理を実行

toggleTabVisibility(shouldCollapse)

isCollapsed = shouldCollapse

※ Tab型の Layout の作成

No.2

↳ (ConstraintLayout)

↳ androidx.viewpager2.widget

· ViewPager2

↳ com.google.android.material.tabs

· TabLayout

の様な形で組み合わせて作るのがポイント



Tab 部分は位置によって、
見え方が変わる様にする。

----- の部分
が表示から、
消える様な
形になら

Toolbar 到達時

(注意) AppBarLayout と連動するためには
ViewPager で管理する Fragment には。

- Nested ScrollView のいずれかにす
- RecyclerView

① collapsedTabLayout

② tabLayout

下準備として、2つの ViewPager をセットアップしておく。

binding.tabLayout.setupViewPager(viewPager)
(collapsedTabLayout)

*1. appBarHeight : AppBar の高さ

*2. collapsedTabHeight : 折りたたみ状態の Tab 高さ

shouldCollapse(true) :

- tabLayout : 非表示
- collapsedTabLayout : 表示

shouldCollapse(false) :

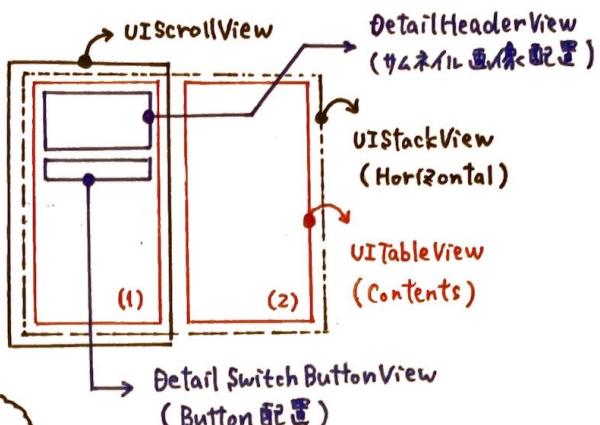
- tabLayout : 表示
- collapsedTabLayout : 非表示

iOSでは似た様な表現をどう作るか?

(1番 Simple に作る with Storyboard)



上方向へ
scroll を
実行する



InterfaceBuilder 上の 配置

↳ ViewController
↳ View

↳ DetailScrollView → 全体の ScrollView

↳ StackView → 表示リストを系統に並べる

↳ UITableView(1) → コニティア表示

↳ UITableView(2)

↳ DetailHeaderView → Header用の View 要素

↳ DetailSwitchButtonView → Button用の View 要素

* ScrollView は
上下左右で Scroll
可能にしておく。

* Tab型 Button を押下した時

→ ふるまい = カル

↓

Slideさせて表示する

UITableView を切り替

(Delegate を定義して、処理
を 2 通り 様に作る)

AutoLayout の制約には注意する

要素同士の制約のつけ方 + Scroll に合わせた変化を合わせる

① DetailHeaderView と Parallax 表現にする場合

→ ScrollView の offset 値が変化した時に発火

UIScrollViewDelegate の scrollViewDidScroll (- scrollView : UIScrollView) を活用

↳ Y 軸 方向の offset 値を調整して AutoLayout 制約へ加算する

② Headerとして表示している要素が隠れても、Tab切り替 Button 用 View は表示させたい!

↳ DetailHeaderView の Top に付与した制約を調整する

→ ④ IB Outlet で切り出して、
Offset 値の変化に合わせて調整

③ コニティア用の UITableView に対して、Y 軸 方向の offset 値を調整する

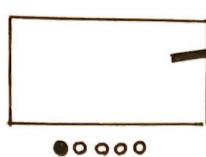
• Case 1: 画像の Parallax 効果付きの View が見えている

→ 配置した各 UITableView の Y 軸 方向の offset 値も一緒に更新

• Case 2: 画像の Parallax 効果付きの View が隠れている

→ 現在重ねかかっていない UITableView の top 位置を調整する

Auto Carousel



一定時間が
経過したら
要素が切り
替わる。

- 無限Carousel = な3
(0 → 1 → 2 → 3 → 4 → 0 → 1 → ...)
- Carousel要素はSwipeで
切替可能(左右)
※ Swipe切替の部分は。
.simultaneousGesture(
 DragGesture(...))

現在位置を示すための変数

④ State private var

activePage : Int = 0



自動で動きくCarousel要素
に対して、変数: activePage
と連動する様にするのが
ポイントとなる

NavigationStack {

VStack(spacing: 16.0) {

// 1. Custom Carousel

④ Stateなので

中には@Bindingで

つづけられる

CustomCarousel(activePage: activePage) {

自作要素 ForEach(mockItems) { item in -----> 表示対象のItem要素
の配達をす } RoundedRectangle(cornerRadius: 16.0)

.fill(item.color.gradient)

.padding(.horizontal, 16.0)

}

.frame(height: 220.0)

// 2. Custom Indicator

HStack(spacing: 6.0) {

ForEach(mockItems.indices, id: \.self) { index in }

Circle()

.fill(activePage == index ? .primary : .secondary)

.frame(width: 8.0, height: 8.0)

}

---- padding ----

.background(.bar, in: .capsule)

.animation(.snappy(duration: 0.35, extraBounce: 0), value: activePage)

} navigationTitle("■■■■")

Indicator
(PageControlの複合形)
ntのを自作してみる

変数: activePage が
変化すると Animation
が発動する



CustomCarouselに関するポイント解説

```
struct CustomCarousel<Content: View>
```

- Internal Property

④ Binding var activePage: Int ↗ 現在、選択中のIndex値

④ ViewBuilder var content: Content ↗ 表示対象のView要素

- private Property

④ State private var scrollPosition: Int? → Scroll位置とパインドするための変数

④ State private var offsetBasedPosition: Int = 0 → (offset位置から算出した) Scroll位置変数

④ State private var isSettled: Bool = false → Carousel要素が固定するかを判定する変数

④ State private var isScrolling: Bool = false → スクロール状態かを判定する変数

* Gesture関連のコントロール

④ GestureState private var isHoldingScreen: Bool = false

Gestureが非activeになると、暗黙的にResetされる

* Timer関連処理

④ State private var timer = Timer.publisher(every: autoScrollingDuration, on: .main, in: .default).autoconnect()

* Timer発火時
の秒数を定義
(1.2)

* View要素の構築 (body要素の断片)

```
GeometryReader {
```

```
    let size = $0.size
```

* Group (subviews: ■■■) * iOS 18 ~ 利用可能

ContainerViewで作成したSubViewを取る

```
    Group (subviews: content) { collection in
        ScrollView (.horizontal) {
            HStack (spacing: 0.0) {
```

③ if let firstItem = collection.first {

firstItem

.frame (width: size.width,
 height: size.height)

} .id (collection.count)

① if let lastItem = collection.last {

lastItem

.frame (width: size.width, height: size.height)

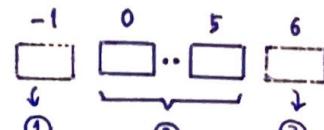
} .id (-1)

② ForEach (collection.indices, id: 1.self) { index in

collection [index]

.frame (width: size.width, height: size.height)

} .id (index)



この様に表示される形を取る

(①, ③を余分に表示する例 - 3)

Geometry Reader

Group
ScrollView

HStack

scrollTargetLayout()scrollPosition (id: \$scrollPosition)scrollTargetBehavior (+paging)scrollIndicators (.hidden)onAppear {

```
    scrollPosition = 0  ↪ position <= 0 は LT
}                                リセットする
```

onScrollPhaseChange { oldPhase, newPhase inisScrolling = newPhase.isScrolling

スクロール状態かを判定する変数を更新

```
if !isScrolling && scrollPosition == -1 {
```

scrollPosition = collection.count - 1

}

@GestureStateで管理する変数 (Gestureの動作状態)

```
if !isScrolling && !isHoldingScreen && scrollPosition == collection.count {
```

scrollPosition = 0

}

Drag Gestureを利用して Slide する動作を実行する

simultaneousGesture (DragGesture (minimumDistance: 0). updating (\$isHoldingScreen, body: { -, out, - in

})

out = true * @GestureStateを更新する

→ @GestureStateの値が変化した時 (Drag 時) の処理

onChange (of: isHoldingScreen, { oldValue, newValue inif newValue {timer.upstream.connect().cancel()

→ 動かしてない時は Timer をキャンセルする

{ else {

if issettled && scrollPosition != offsetBasedPosition {scrollPosition = offsetBasedPosition

→ スクロールを更新して Timer を再セットする

}

timer = Timer.publish(every: Self.autoscrollDuration, on: .main,in: .default).autoconnect()

})

→ == タイマー Timer がスタートする

★

iOS17 ~ 利用可能な Modifier

scrollTargetLayout (isEnabled: Bool = true)

ScrollView内の主要なレイアウトコンテナに追加

→ これが Scroll 対象の要素

scrollPosition (id: anchor:)

ScrollViewにおけるスクロール位置をハイド

→ @State private var scrollPosition: Int?

※ 取得できるのは、index 値の様な形

scrollTargetBehavior (-:)

ScrollView の動作を指定する * 今日は .paging

• `onReceive(timer) { - in` → Timer の設定時間が来たら実行する処理

guard !isHoldingScreen && !isScrolling else { return } // 動かしている時は
// 1X 降の動作をしない

`let nextIndex = (scrollPosition ?? 0) + 1`

`withAnimation(.snappy(duration: 0.25, extraBounce: 0)) {`

`scrollPosition = (nextIndex == collection.count + 1) ? 0 : nextIndex`
★ collection は Group から取得して container

Animation 処理

時間経過したら

Carousel を移動

• `onChange(of: scrollPosition) { oldValue, newValue in` → 実数 scrollPosition が変化した時に
activePage を更新して index 値を変える

`if let newValue {`

★ activePage の算出

条件分岐 ① `newValue == -1` → `collection.count - 1`

② `newValue == collection.count` → `0`

★ 少し複雑な算出ではある...

③ インデックス外の場合 → `max(min(newValue, collection.count - 1), 0)`

`}`

→ スクロール位置を取って、任意の処理が

できる様になる (iOS 18 ~)

• `onScrollGeometryChange(for: CGFloat.self) {`

`$0.contentOffset.x`

`} action: { oldValue, newValue in`

★ size.width は GeometryReader で取得した値

`let index = size.width > 0 ? Int((newValue / size.width).rounded() - 1) : 0`

`isSettled = size.width > 0 ? (Int(newValue) % Int(size.width) == 0) : false`

Carousel 要素が
固定されているか
を判定する

`offsetBasedPosition = index`

GeometryReader の size.width & contentOffset.x を用いて計算で出した index 値

`if isSettled && !isScrolling && !isHoldingScreen && (scrollPosition != index || index == collection.count) {`

固定中とき

スクロール中とき

Gesture の実行中では

ないとき

① 算出した index が 整数 の

scrollPosition と一致しない (or)

② 算出した index が 配置要素の

総数と等しい

`scrollPosition = index == collection.count ? 0`

: index

固定中でかつ、index 値の条件に該当する時には。

実数: scrollPosition を更新する

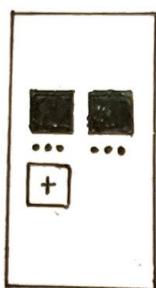
`}`

< Netflix の「あの動き」を clone(?) で実現する >

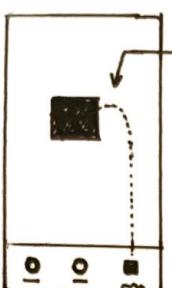
No.1

特定の場所に吸い込まれる様な Animation

例. Netflix の視聴対象を選ぶ画面



誰が視聴する
のかをボタンを
押す
細かな形が
動きそう...



TabBar表示
(左ナビゲイ션)
に切替わる時
Animation 付き

Animation
動作イメージ
滑らかで放物線の
様な軌跡を描く?
Tab要素の中へ吸い
込まれていく様な形

- ① Path に三段で Animation
をする形で実現する
or
② GeometryReader の値
を使って Customize する

どちらも選択可かは、後は matchedGeometryEffect?

結構 View構造次

集かも...??

* GeometryReader を利用した座標計算

Pros:

コードの分量が少なくて余地がある?

- GeometryReader や GeometryProxy
から得られた値を使うのがポイント
- modifier を利用して、位置情報を適用する

- position ↗
この値を調節する
事がポイント
- scaleEffect
この値を調節する
事がポイント
- opacity
する事がポイント

Cons:

滑らかでない Animation の様には
いけないケースがある....

実際の Sample を元にまとめ

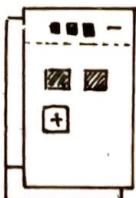
* 余談 → Geometry Readerだけではキツイ点

- そのままの Animation では、動きが直線的になる
- View の階層によっては、うまく重なる形の実現が難しい

滑らかではあるものの...
⇒ ではねば「放物線を描く自由落下」
の方が「等速直角運動」より望ましい



元々は TabBar
用の画面が
ベースにある



おもとの
Content View 内
で ZStack で重ねる

- まずはおもとの
画面があって →
重ねてある

- 状態管理をする
クラスの内容を →
更新する

- Animation に対する
処理が実行され、
TabBar が表示

* この Sample 実装では、
Application 全体の状態
管理をする部分では、
"③ Observation" で ViewModel
の様な形で状態管理をしている。

* Application 全体の
状態管理クラス
のインスタンスは
③ Bindable
var で

ZStack {

MainView()

* 条件に応じた処理を
諸々書き足していく

★ ③ Bindable var
利用することによって、
変化に連動する

ProfileView()

animateToCenter:

\$bindableData.animateToCenter,

animateToMain:

\$bindableData.animateToMain,

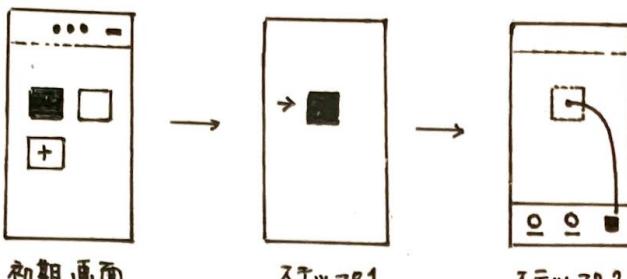
progress:

\$bindableData.progress

↓

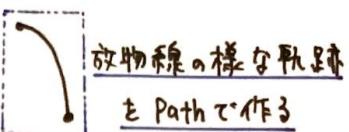
- 例.
animateToCenter → 対象が中心まで移動したか?
animateToMain → Xaxi の画面が表示されたか?
progress → Animation 実行割合

* Profile画面で Animation をどうやって作っていい?



初期画面の構造でポイント1=なる所

* この動きには何で
の部分を実現する際には、
Path Animation が利用可能
かを考えてみる。



VStack {
 ----- 必要な要素 -----
 } [■の部分は ProfileCardView (profile) の様な View 要素で
 個別に表示して並べる形をとる]

- opacity (animateToCenter: 0 : 1)
- opacity (animateToMainView: 0 : 1) → Animation を実行する時は非表示にして

• overlayPreferenceValue (RectAnchorKey.self) { value in
 AnimatedLayerView (value)
 }

* 子 View の frame 値を取得
は直接できないので、
子 View 側で取得してから
親 View へ伝達する

View からこの指定された preference 値を使って別の View で

最初の View 上に overlay (かぶせもの) として生成する

- overlayPreferenceValue で値を取ってくるため、PreferenceKey を受け取るために準備をする
→ RectAnchorKey.swift を利用する

struct RectAnchorKey : PreferenceKey {

static var defaultValue : [String : Anchor<CGRect>] = [:]

static func reduce(value: inout [String : Anchor<CGRect>],

nextValue: () -> [String : Anchor<CGRect>]) {

value.merge(nextValue())

} } * Anchor という仕組が用意されてるので = これを利用す

- ProfileCardView の part 1

• anchorPreference(key: RectAnchorKey.self, value: .bounds, transform: {
 anchor in

return [... .sourceAnchorID : anchor]

* こちらは直接 bounds 値を
取得する方法

})

```
• overlayPreferenceValue (RectAnchorKey.self) { value in
```

AnimationLayerView (value)

}



④ ViewBuilder

* 子Viewの値を利用する → Animation の終端を決定する

```
func AnimationLayerView (_ value: [String : Anchor<CGRect>]) -> some View {
```

GeometryReader を利用して、その位置情報を利用する

1. [String : Anchor<CGRect>] から、GeometryReader の値を取ってみる

* proxy [sourceAnchor] , proxy.frame(in: .global) を利用

↓
value の値を元に取ってきた値

let sRect

↓
let screenRect

Position 計算の実行

let sourcePosition → sRect.midX (midY) の値を使って CGPoint を算出する

let centerPosition → x: screenRect.width / 2 y: (screenRect.height / 2) - 40 で
CGPoint を算出する

let destinationPosition → Tab 集素の Position を使って CGPoint を算出する

Animation 用の Path を描く * Path を利用して放物線を描く

let animationPath = Path { path in

① path.move(to: centerPosition) 真ん中の Position が始点となる

path.addQuadCurve(to: destinationPosition, ② 終点を設定する

control: CGPoint(x: centerPosition.x * 2,

③ 2つの点を結ぶベジエ曲線を作成するための
y: centerPosition.y - (centerPosition.y / 0.8))

)
}

基準点を設ける

* 跡を描いた時の処理

animationPath.stroke(.white.opacity(0.5), lineWidth: 1)

Path をつなげて曲線を書く

変化量を算出する (trimmedPathを使って調整)

* 元々は Path の一部を切り取る

let endPosition ↗ 位置の計算をする形
let currentPosition animationPath.trimmedPath(from: 0, to: ■■■)
· currentPoint ?? destinationPosition

let diff = CGSize(width: height:)
=⇒ diff は offset 値で利用 endPosition.x(.y) - currentPosition.x(.y)
* 位置計算を実行する

実際に動かす所の構造 ↗ * offset が Modifier に適用する

ZStack {

Image(profile.icon)

* 各種 Modifier 定義

[frame, clipShape, animation, opacity 等について。
渡された変数を元に条件分岐をしている]

· offset (animateToMainView ? diff : .zero)

· modifier (

 AnimatedPositionModifier (

 ... 値を Modifier に三度して、上手に Animation を実行する ...

)) * 三度された値を利用して .position を更新する

まとめ

技術の世界は日々変化し続けています。その中で、真の学びとは単に情報を集めることではなく、得た知識を自分のものとし、実践に活かせる知恵に変えていくプロセスそのものです。私が大切にしている手書きノートによる学習は、まさにこの技術者としての成長の本質を体現しているのかもしれません。

デジタル時代には、膨大な情報があふれていますが、本当の理解は情報の量ではなく、その質によって決まります。サンプルコード、技術書、オンライン教材から得られる断片的な知識を、自分の手と頭で丁寧につなぎ合わせていくこと。それが、表面的な知識のCollectionから、技術の本質的な理解への道だと考えています。

手書きのノートは、単なるメモ帳ではありません。それは思考を可視化するキャンバスであり、抽象的な概念を具体的なイメージに変換する、最も個人的で創造的な学びの方法です。理論と実践をつなぐ、私だけの知識の地図とも言えるでしょう。

このアプローチの真の価値は、知識を暗記することではなく、学んだことを実際に活かせるようになります。似たような技術的な課題に直面したとき、過去の学びがすぐに思い出せ、落ち着いて対応できる柔軟性が身につきます。蓄積された知識は、問題解決の力を大きく後押しする、目に見えない財産となります。

エンジニアとして成長することは、常に自分に合った学び方を探し、改善し続けることです。手書きのノートによる学習は、単なる技術の記録方法ではなく、自分の知的好奇心を大切にする姿勢そのものなのかもしれません。

最終的に、私たちエンジニアに求められるのは、技術の表面をなぞるのではなく、その本質を理解し、創造的に活用する力です。情報を受け取り、自分なりに咀嚼し、血肉化する。このサイクルが、技術者としての成長を支える原動力となるのです。