Refreshable API を TCA で使う

iOSアプリ開発のためのFunctional Architecture情報共有会5

Refreshable API とは

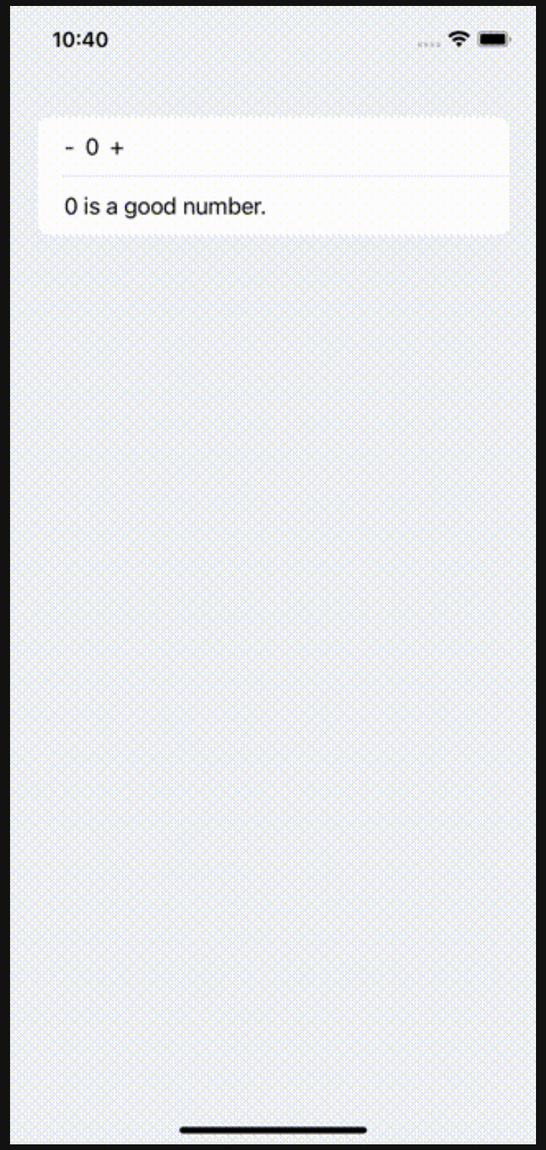
- iOS 15 から利用できるようになった View Modifier
- SwiftUI なら簡単に Pull to Refresh を実現できる
- `refreshable` を利用するだけ
  - これが取る closure は async な処理を要求する

```
List(mailbox.conversations) {
    ConversationCell($0)
}
.refreshable {
    await mailbox.fetch()
}
```

TCA ではどのように refreshable を利用できるか

- TCA では v0.23.0 から `ViewStore.send(_:while:)` というものが導入されている(現在時点では Beta)
- これを refreshable 内で利用すると TCA で refreshable が上手く扱える
- 「Async Refreshable: Composable Architecture」という Point-Free 内のエピソードをもとに、
  TCA でどのように refreshable が扱えるか見ていこうと思います

Refreshable with TCA を理解するために利用する例



API Client 的部分

```
struct FactClient {
  var fetch: (Int) → Effect<String, Error>
  struct Error: Swift.Error, Equatable {}
}

extension FactClient {
  static let live = Self(
    fetch: { number in
      URLSession.shared.dataTaskPublisher(
        for: URL(string: "http://numbersapi.com/\(number)/trivia")!
      )
      .map { data, _ in String(decoding: data, as: UTF8.self) }
      .catch { _ in
        Just("\(number) is a good number Brent")
          .delay(for: 1, scheduler: DispatchQueue.main)
      }
      .setFailureType(to: Error.self)
      .eraseToEffect()
    }
  )
}
```

State, Action, Environment

```
struct PullToRefreshState: Equtable {
  var count = 0
  var fact: String?
}

enum PullToRefreshAction: Equatable {
  case cancelButtonTapped
  case decrementButtonTapped
  case incrementButtonTapped
  case refresh
  case factResponse(Result<String, FactClient.Error>)
}

struct PullToRefreshEnvironment {
  var fact: FactClient
  var mainQueue: AnySchedulerOf<DispatchQueue>
}
```

Reducer

```
refreshReducer = Reducer<PullToRefreshState, PullToRefreshAction, PullToRefreshEnvironment>
{ state, action, environment in
    struct CancelId: Hashable {}
    switch action {
    case .decrementButtonTapped:
        state.count -= 1
        return .none
    case .incrementButtonTapped:
        state.count += 1
        return .none
    case let .factResponse(.success(fact)):
        state.fact = fact
        return .none
    case .factResponse(.failure):
        return .none // TODO: エラーハンドリング
    case .refresh:
        return environment.fact.fetch(state.count)
            .receive(on: environment.mainQueue)
            .catchToEffect(PullToRefreshAction.factResponse)
            .cancellable(id: CancelId())
    case .cancelButtonTapped:
        return .cancel(id: CancelId())
    }
}
```

View(store 宣言部分)

```
struct PullToRefreshView: View {
    let store: Store<PullToRefreshState, PullToRefreshAction>

    var body: some View {
        //  ...
    }
}
```

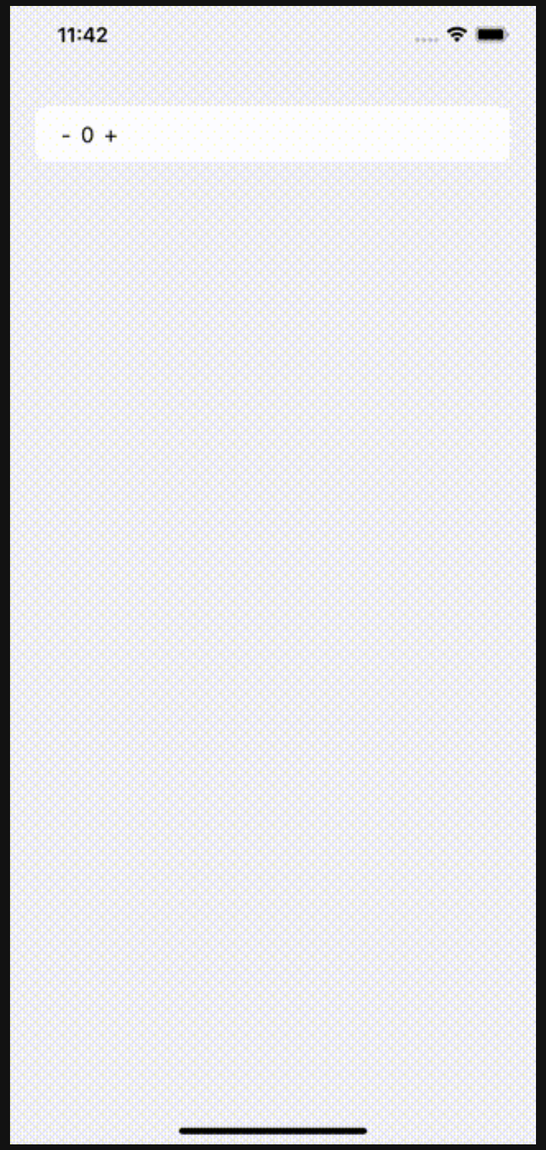
View(body)

```
var body: some View {
  WithViewStore(self.store) { viewStore in
    List {
      HStack {
        Button("-") { viewStore.send(.decrementButtonTapped) }
        Text("\(viewStore.count)")
        Button("+") { viewStore.send(.incrementButtonTapped) }
      }
      .buttonStyle(.plain)

      if let fact = viewStore.fact {
        Text(fact)
      }

      if viewStore.isLoading {
        Button("Cancel") {
          viewStore.send(.cancelButtonTapped)
        }
      }
    }
    .refreshable {
      viewStore.send(.refresh)
    }

  }
```

Preview

```
struct PullToRefreshView_Previews: PreviewProvider {
    static var previews: some View {
        PullToRefreshView(
            store: .init(
                initialState: .init(),
                reducer: pullToRefreshReducer,
                environment: .init(
                    fact: .live,
                    mainQueue: .main
                )
            )
        )
    }
}
```

実行してみる

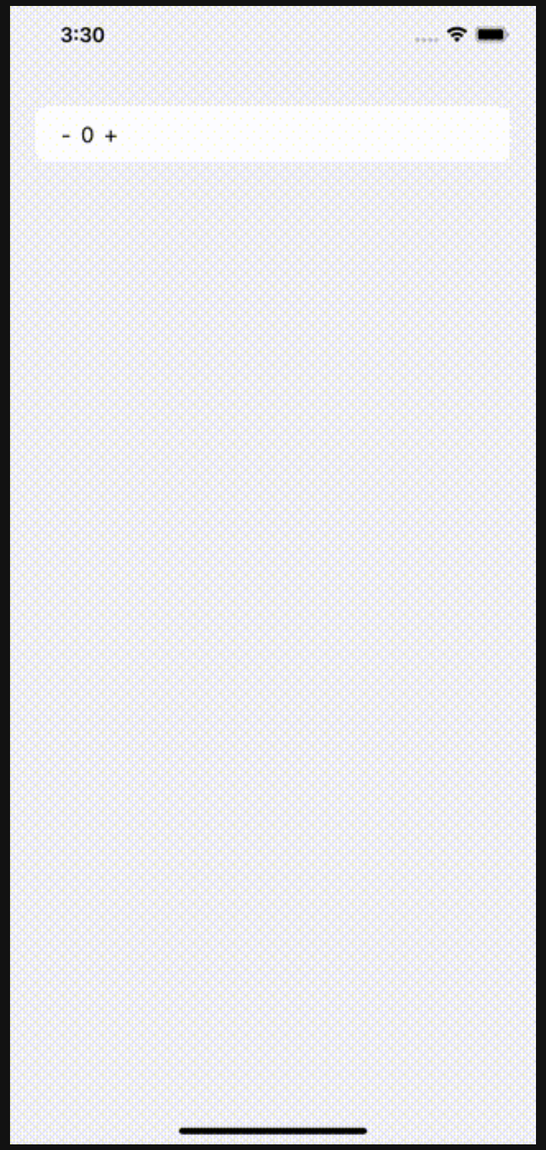


動作的には問題なさそうに見える？

コードを少し変更してみる

```
case .refresh:
    return environment.fact.fetch(state.count)
            .delay(for: 2, scheduler: environment.mainQueue)
            .catchToEffect(PullToRefreshAction.factResponse)
            .cancellable(id: CancelId())
```

通信が完了してないのに indicator が消えてしまう



何が問題なのか

- `refreshable` View Modifier は closure に async な処理を要求する
  - 提供された非同期な処理が実行されている限り loading indicator が留まるというものになっている
- 現在実装している `viewStore.send(.refresh)` は async ではない同期的な処理
- TCA でこの問題を解決するためには少し工夫する必要がある

State に isLoading を導入

```
struct PullToRefreshState: Equatable {
    var count = 0
    var fact: String?
    var isLoading = false
}
```

isLoading を reducer で操作

```
switch action {
case let .factResponse(.success(fact)):
    state.fact = fact
    state.isLoading = false
    return .none
case .factResponse(.failure):
    state.isLoading = false
    return .none
case .refresh:
    state.isLoading = true
    // ...
case .cancelButtonTapped:
    state.isLoading = false
    return .cancel(id: CancelId())
}
```

あとは async 的に利用できる send があると良さそう

```
// こんな感じ
.refreshable {
    await viewStore.send(.refresh, while: \.isLoading)
}
```

async な send の signature はこのような形

```
extension ViewStore {
    func send(
        _ action: Action,
        `while`: (State) → Bool
    ) async {
        // 実装
    }
}
```

実装を考えてみる

```
func send(
    _ action: Action,
    `while`: (State) → Bool
) async {
    // まずは何よりも Action を発火させる必要がある
    self.send(action)
    // ViewStore には全ての state の変化が流れてくる publisher があるため、それを監視する
    self.publisher
        .filter { !`while`($0) } // `while` は escaping でないためエラーが発生する
}
```

実装を考えてみる2

```
func send(
    _ action: Action,
    `while` isInFlight: @escaping (State) → Bool // escaping + internal argument
) async {
    self.send(action)
    self.publisher
        .filter { !isInFlight($0) }
        .prefix(1) // isLoading の変化の監視は最初のものだけ判別できれば良い
        .sink { _ in
          // 実装
        }
}
```

- ここで生じる問題点
  - `sink` は `cancellable` を返すがどうする？
  - 最終的には async な task を構築する必要があるがどうする？

publisher -> async にするための Bridge

- Swift はそのための Bridge となる function を用意してくれている
  - `withUnsafeContinuation`
  - non-async/await なコードを async/await なコードに変えられる

```
// signature
withUnsafeContinuation(<#(UnsafeContinuation<T, Never>) → Void#>)

// 使い方
let number = await WithUnsafeContinutation { continuation in
  DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
    continuation.resume(returning: 42)
  }
}
```

`withUnsafeContinuation` を `send` で利用する

```
func send(
    _ action: Action,
    `while` isInFlight: @escaping (State) → Bool
) async {
  self.send(action)
  await withUnsafeContinuation { continuation in
    self.publisher
      .filter { !isInFlight($0) }
      .prefix(1)
      .sink { _ in
        continuation.resume()
      }
  }
}
```

`cancellable` の取り扱い方

```
func send(
    _ action: Action,
    `while` isInFlight: @escaping (State) → Bool
) async {
  self.send(action)

  var cancellable: Cancellable?
  await withUnsafeContinuation { (continuation: UnsafeContinuation<Void, Never>) in // 型推論ができなくなるため型を明示
    cancellable = self.publisher
      .filter { !isInFlight($0) }
      .prefix(1)
      .sink { _ in
        continuation.resume()
        _ = cancellable // strongly capture
      }
  }
}
```

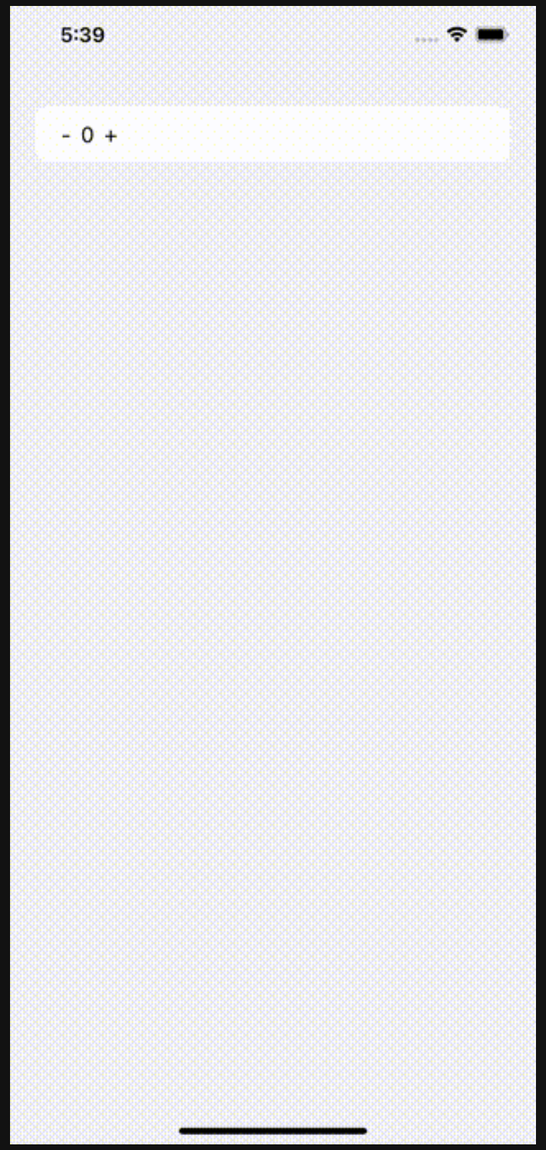
現在時点の Beta ディレクトリにある実装方法

```
func send(
  _ action: Action,
  while predicate: @escaping (State) → Bool
) async {
  self.send(action)
  await self.suspend(while: predicate)
}

func suspend(while predicate: @escaping (State) → Bool) async {
  _ = await self.publisher
    .values // AsyncPublisher<Self>
    .first(where: { !predicate($0) }) // AnyCancellable を返却しないため、そのための対処が必要ない
}
```

コンパイルが通るようになる

```
.refreshable {
  await viewStore.send(.refresh, while: \.isLoading)
}
```

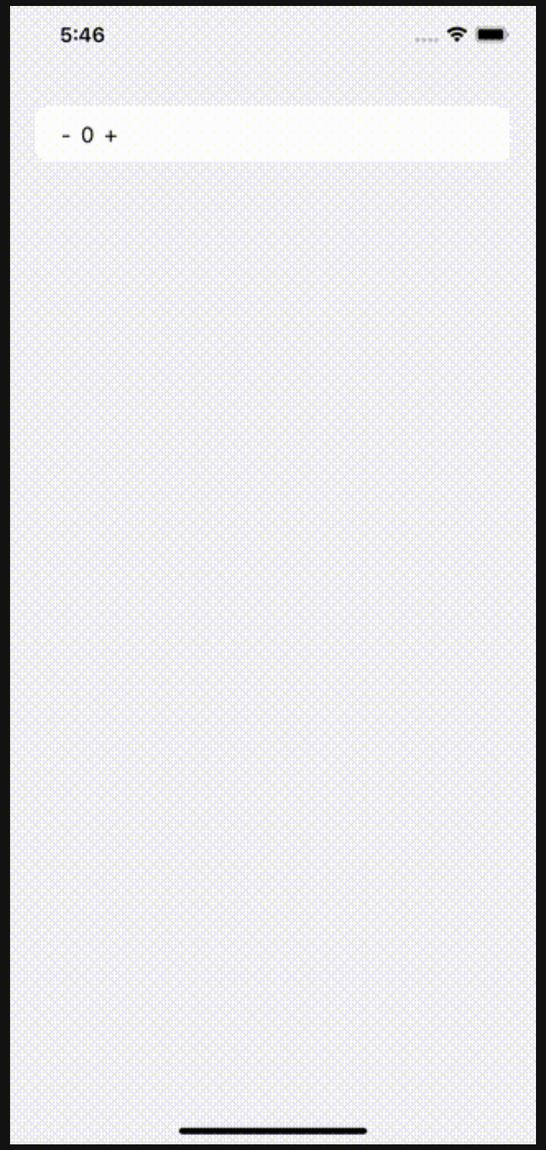
cancel 時の animation がない問題がある



combine-schedulers の animation 機能を使って解決

```
case .refresh:
  state.isLoading = true
  state.fact = nil
  return environment.fact.fetch(state.count)
    .delay(for: 2, scheduler: environment.mainQueue.animation())
    // ...

// ...

Button("Cancel") {
  viewStore.send(.cancelButtonTapped, animation: .default)
}
```

無事 cancel 時の animation が行われるようになる



まとめ

- SwiftUI の `refreshable` View Modifier は簡単に Pull To Refresh を表現できる
- `refreshable` は async な処理を要求するため、TCA で利用するためには工夫が必要
- 現在時点では Beta だが、TCA にはそのための `viewStore.send(_:while:)` が用意されている
- 発表では紹介しなかったが、TCA を利用すると非常に網羅的なテストが可能となる
  - 網羅的なテストができることが TCA の売り
  - 例えば State を追加したりしたら、その State の変化を検証しないとテストは失敗する
  - 発生しうる Action も `receive` 等によって網羅する必要がある
- 素の SwiftUI だと以下のような部分でテストが厳しくなると述べられていた
  - 詳しくは Point-Free の「Async Refreshable: SwiftUI」を参照して頂ければと思います🙏
  - API リクエストをキャンセルする際のフローがテストできない(する方法がわからない)
    - Xcode Beta 版のバグか、Swift の custom executors を使う必要があるのかはっきりしていないらしい
  - async な処理中の `isLoading` の変化をテストするために、テスト内で Sleep を行う必要がある