Jetpack Composeで Reduxっぽいアーキテクチャを試す

Jetpack Compose

Architecture

Mori Atsushi

Shibuya.apk #44
2022/09/01

Mori Atsushi

Twitter: @at_sushi_at

2019年度 未踏スーパークリエータ

LINE株式会社（2023年4月〜）

Android application engineer

詳解 Kotlin Coroutines [2021]

Katalog / Koject / InsetsX

Jetpack Compose進めています

Jetpack ComposeでLINE Design Systemを実装する

Atsushi Mori 2023-08-17
メッセンジャープロダクトアプリ開発1チーム

はじめに

こんにちは、コミュニケーションアプリ LINE のAndroidクライアントを開発している森です。

LINEでは現在、Jetpack Composeの導入を進めています。
導入を進めるにあたって、LINEのDesign Systemや独自機能である「着せかえ」機能を実装する必要がありました。

https://engineering.linecorp.com/ja/blog/lind_design_system_with_jetpack_compose

今回のゴール

Jetpack ComposeでReduxっぽいアーキテクチャを試す方法と、そのメリット/デメリットを知る

1. 標準的なアーキテクチャの辛み
2. Reduxっぽいアーキテクチャを試す
3. 非同期処理を扱う
4. Reducerを分割する
5. まとめ

1

標準的な
アーキテクチャの辛み

標準的なアーキテクチャ

ViewModelでStateを保持

```
@Composable
fun SomeScreen(
    modifier: Modifier = Modifier,
    viewModel: SomeViewModel = hiltViewModel(),
) {
    val uiState: UiState by viewModel.uiState
        .collectAsStateWithLifecycle()

    SomeScreen(
        uiState = uiState,
        onFollowClick = viewModel::follow
    )
}
```

```
@Composable
fun SomeScreen(
    uiState: UiState,
    onFollowClick: () -> Unit,
    modifier: Modifier = Modifier,
) {
    /* ... */
}
```

複雑な画面だと…

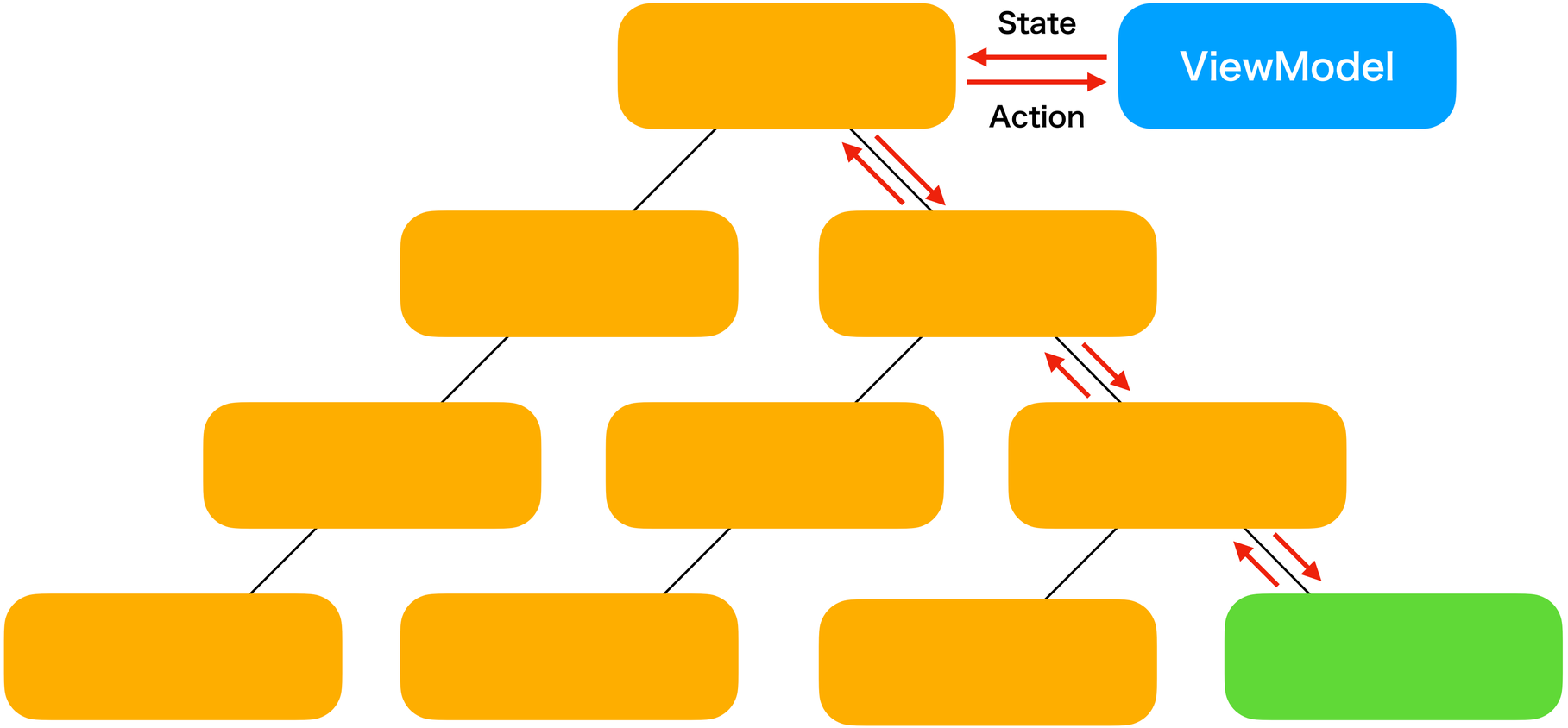
Composableの引数が爆増 🤯

際限なく増える
ステートとアクション

```
@Composable
fun SomeScreen(
    uiState1: UiState1,
    uiState2: UiState2,
    onFollowClick: () -> Unit,
    onAction1Click: () -> Unit,
    onAction2Click: () -> Unit,
    onAction3Click: () -> Unit,
    onAction4Click: () -> Unit,
    /* ... */
    modifier: Modifier = Modifier,
) {
    /* ... */
}
```

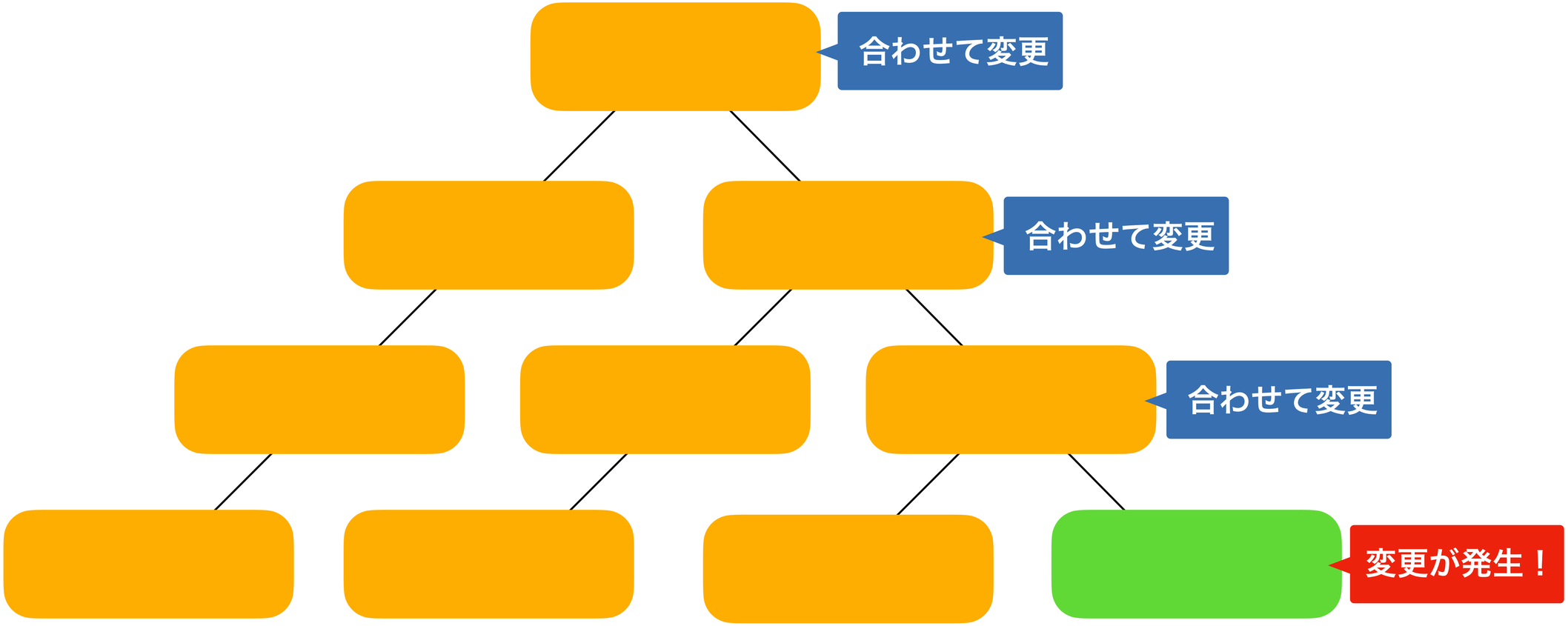
バケツリレー地獄

末端のコンポーネントと情報をやりとりするのは大変



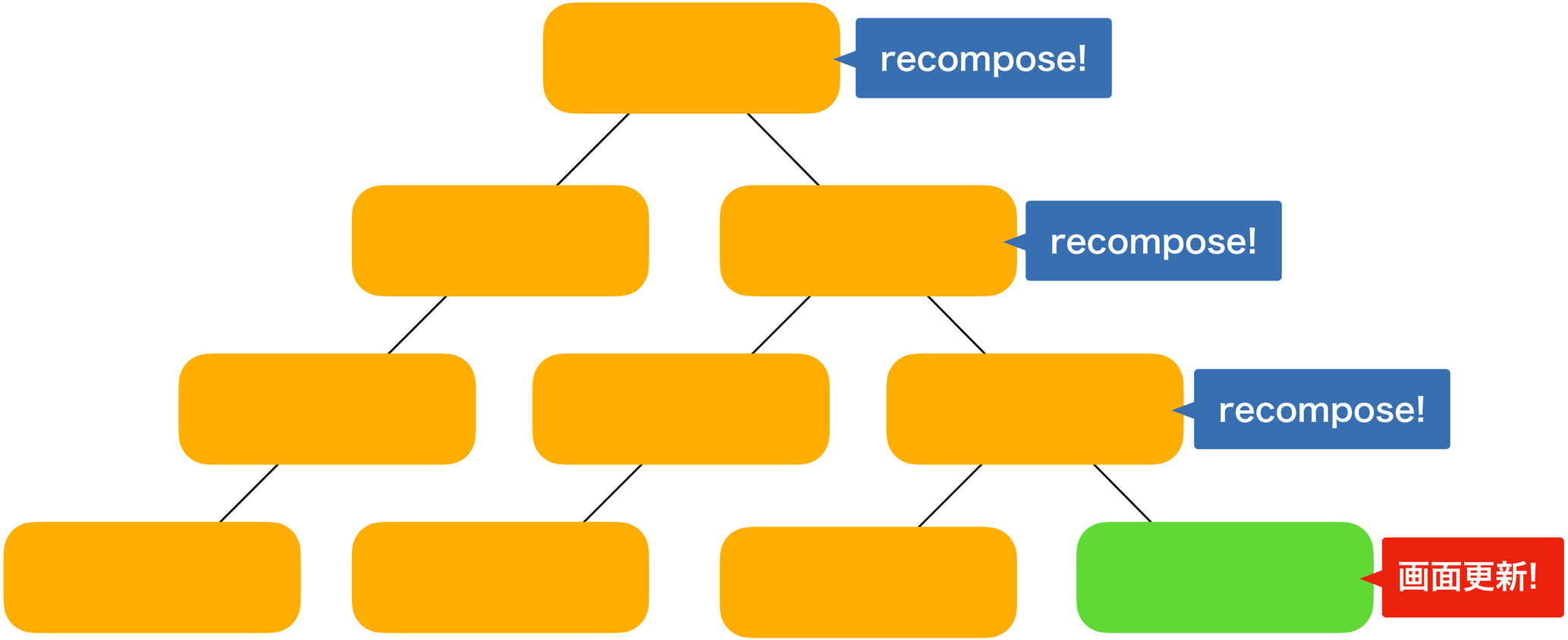
単一責任の原則の違反

末端の変更に合わせて、親も変更する必要が出てくる



不要なrecomposeの発生

末端の更新のために親もrecomposeされる

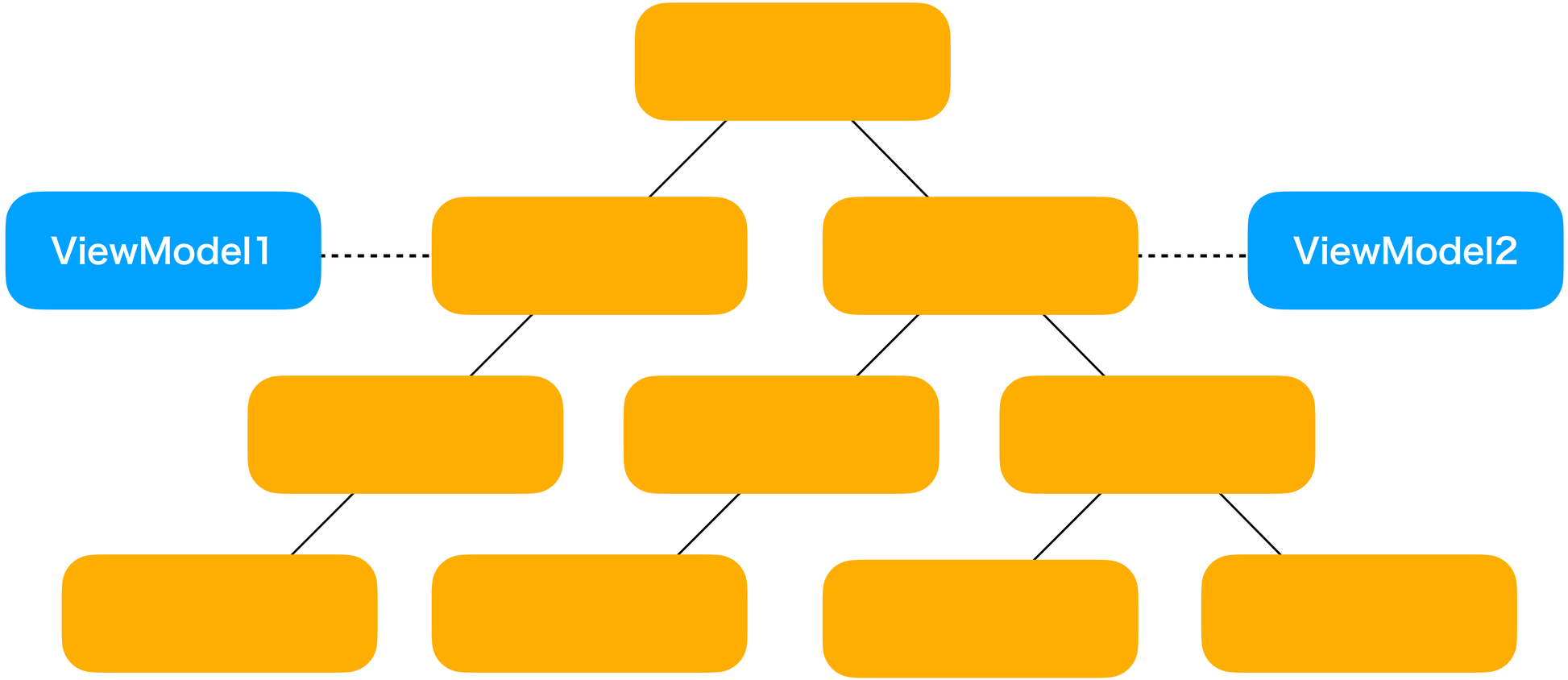


ViewModelも肥大しがち…

```
class SampleViewModel(
    val repository1: Repository1,
    val repository2: Repository2
): ViewModel {
    private val _uiState1 =
      MutableStateFlow(UiState1.Loading)
    val uiState1: StateFlow<UiState1> = _uiState1

    private val _uiState2 =
      MutableStateFlow(UiState2.Loading)
    val uiState1: StateFlow<UiState2> = _uiState2

    fun follow() { /* ... */ }

    fun action1() { /* ... */ }

    fun action2() { /* ... */ }

    fun action3() { /* ... */ }
```

ViewModelを分割？

状態を共有する場合等、上手く行かないケースも

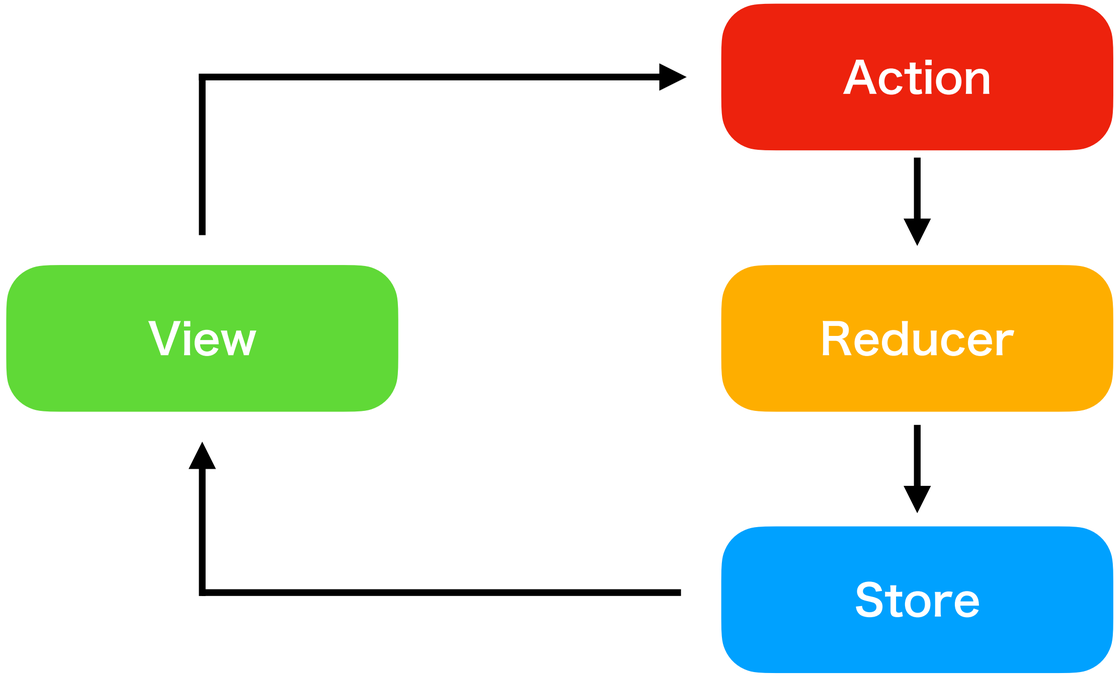


2

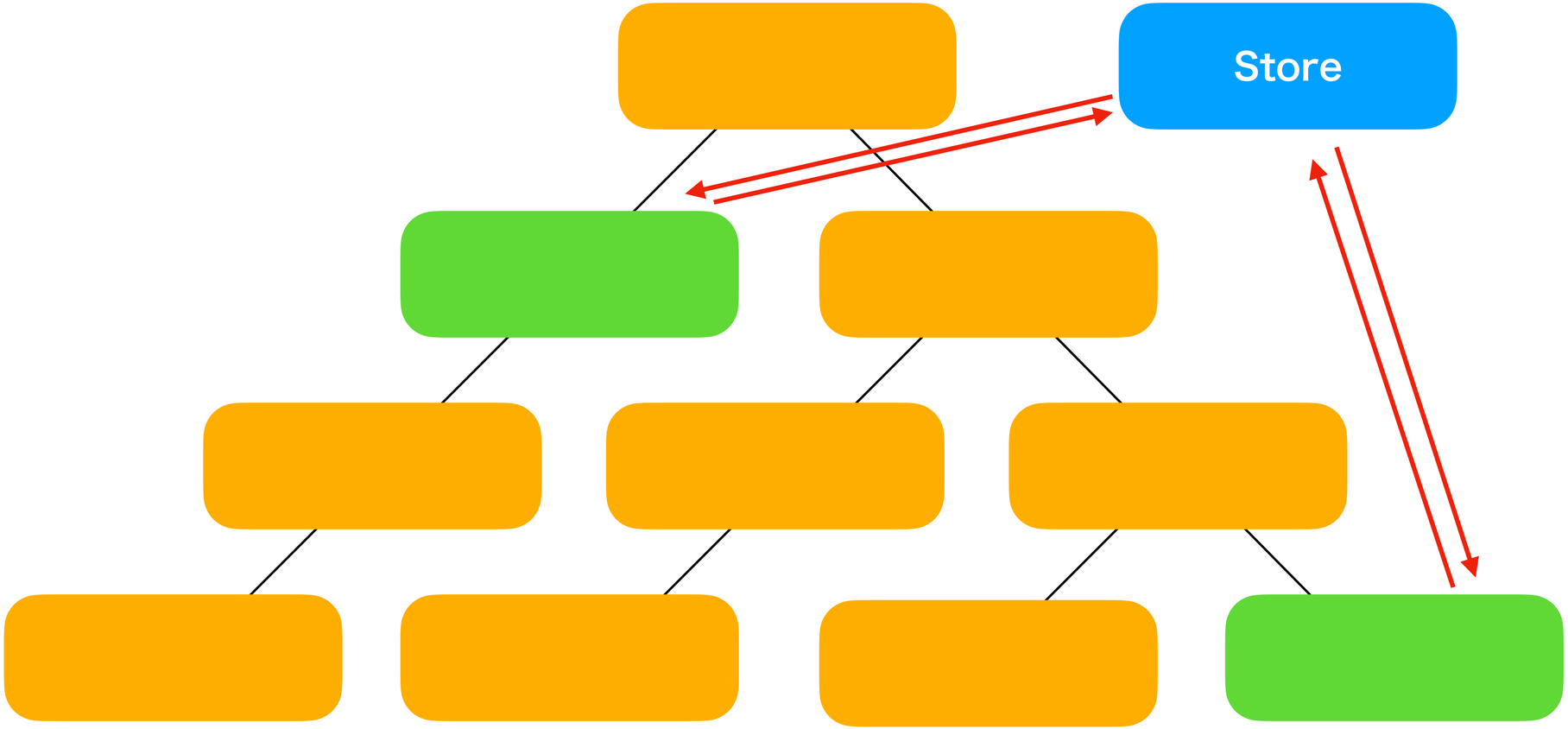
Reduxっぽい
アーキテクチャを試す

Reudxとは？

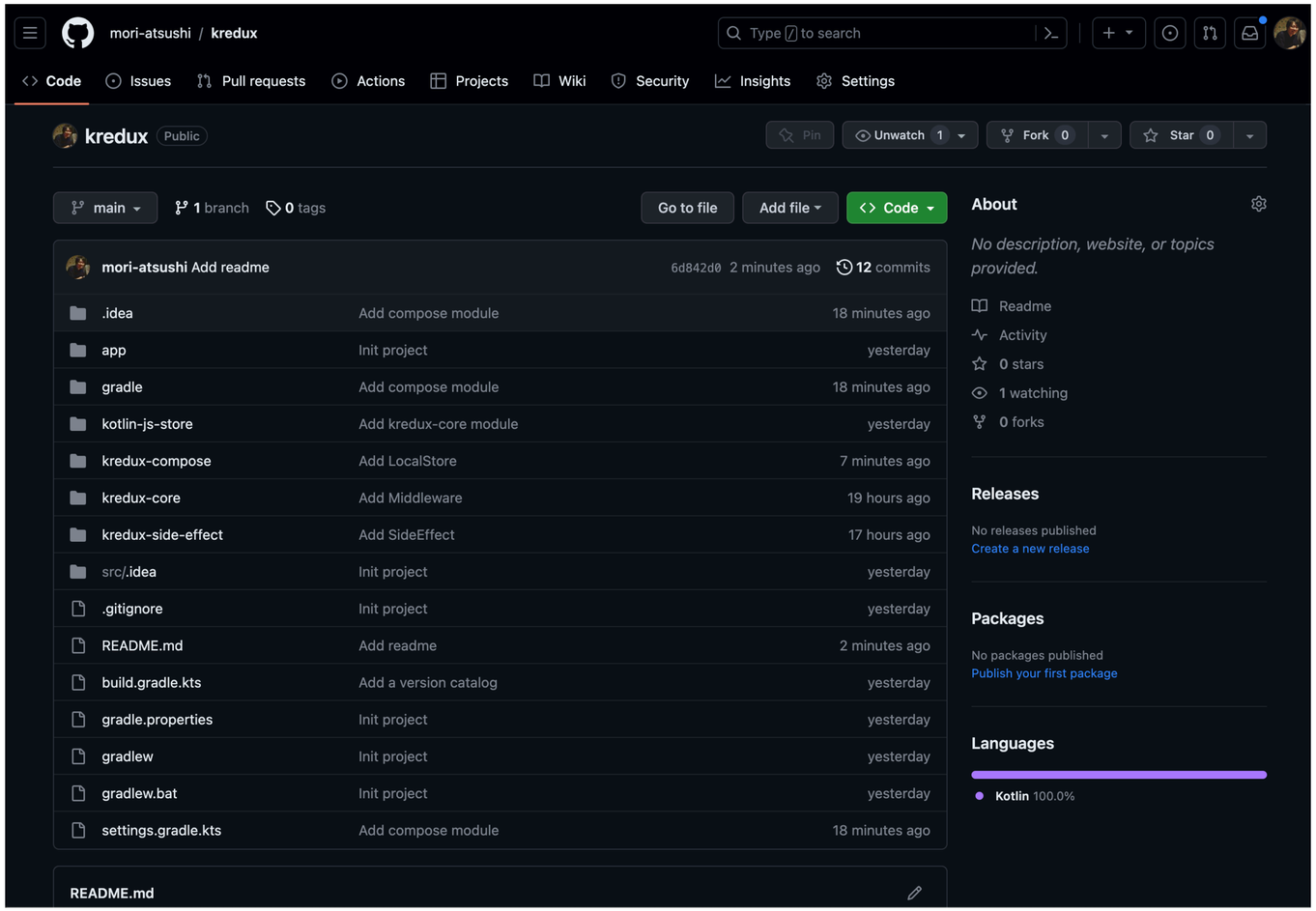
JavaScriptアプリケーションのための状態管理ライブラリ



Storeをコンポーネントが参照できる



GitHubに実装を上げています



https://github.com/mori-atsushi/kredux

StateとActionを定義する

```
data class CounterState(
    val count: Int,
)

sealed interface CounterAction {
    object Increment : CounterAction
    object Decrement : CounterAction
}
```

Reducerを作る

```
data class CounterState(
    val count: Int,
)

sealed interface CounterAction {
    object Increment : CounterAction
    object Decrement : CounterAction
}

val counterReducer: Reducer<CounterState, CounterAction> =
    createReducer(CounterState(0)) { acc, action ->
        when (action) {
            CounterAction.Increment -> acc.copy(count = acc.count + 1)
            CounterAction.Decrement -> acc.copy(count = acc.count - 1)
        }
    }
```

Storeを作る

```
class CounterViewModel: ViewModel() {
    val store = createStore(
        reducer = counterReducer,
        coroutineScope = viewModelScope
    )
}
```

StoreをCompositionLocalで提供

```
@Composable
fun Counter(
    store: Store<CounterState, CounterAction> = viewModel<CounterViewModel>().store
) {
    CompositionLocalProvider(
        LocalCounterStore provides store
    ) {
        /* ... */
    }
}

val LocalCounterStore = localStoreOf<Store<CounterState, CounterAction>>()
```

Storeの値を監視する

```
@Composable
fun CounterPanel(
    modifier: Modifier = Modifier
) {
    val count by LocalCounterStore.select { it.count }
    val dispatch = LocalCounterStore.dispatch

    Column(modifier = modifier) {
        Text(text = count.toString())
        Row {
            Button(onClick = { dispatch(CounterAction.Increment) }) {
                Text(text = "+")
            }
            Button(onClick = { dispatch(CounterAction.Decrement) }) {
                Text(text = "-")
            }
        }
    }
}
```

Selectで必要な値だけ監視する

ActionをDispatchする

```
@Composable
fun CounterPanel(
    modifier: Modifier = Modifier
) {
    val count by LocalCounterStore.select { it.count }
    val dispatch = LocalCounterStore.dispatch

    Column(modifier = modifier) {
        Text(text = count.toString())
        Row {
            Button(onClick = { dispatch(CounterAction.Increment) }) {
                Text(text = "+")
            }
            Button(onClick = { dispatch(CounterAction.Decrement) }) {
                Text(text = "-")
            }
        }
    }
}
```

Dispatcherを取得

Actionを呼び出し

アプリケーションロジックを CompositionLocalで提供する？

暗黙的なデータ渡しになる
公式では推奨されていない
Reduxを再現するため、今回は許容

メリット/デメリット

単一方向のデータフローが強制される

複雑なステートが管理しやすくなる

入力と出力がはっきりしてるのでテストが書きやすい

デメリット：単純なステート管理には書くコードが増える

バケツリレー地獄からの脱却

子コンポーネントで直接Storeを参照できる

デメリット：やりすぎると再利用性に支障が出る

3

非同期処理を扱う

非同期処理は苦手

Action -> Stateのデータフローから逸脱するため

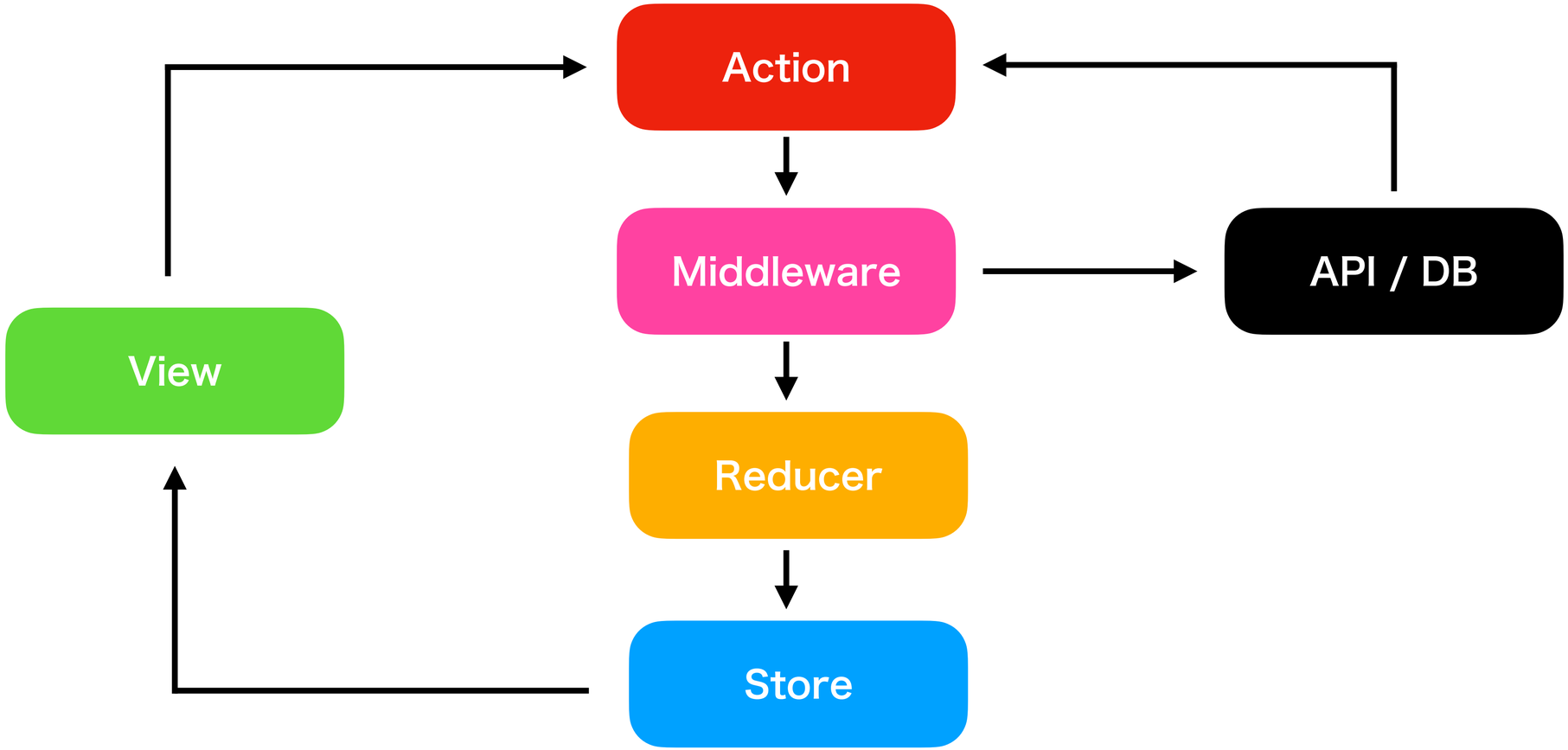
いろんな対応方法が存在

ActionCreator内で処理する

ActionとIntentを分けてその間で処理する

Reducer内で処理する（the Composable Architecture）

MiddlewareでSideEffectをもたせる



Actionを定義する

UIからのActionに、非同期処理のActionを加える

```
sealed interface AsyncAction {
    data class Request(val id: Int) : AsyncAction
    data class Success(val data: String) : AsyncAction
    data class Failure(val error: Throwable) : AsyncAction
}
```

SideEffectでActionを監視する

```
val sideEffects = sideEffects<AsyncState, AsyncAction> {
    collect<AsyncAction.Request> { request ->
        try {
            val data = Api.request(request.id)
            dispatch(AsyncAction.Success(data))
        } catch (e: Throwable) {
            dispatch(AsyncAction.Failure(e))
        }
    }
}
```

RequestのActionを監視する

SideEffectでActionを監視する

```
val sideEffects = sideEffects<AsyncState, AsyncAction> {
    collect<AsyncAction.Request> { request ->
        try {
            val data = Api.request(request.id)
            dispatch(AsyncAction.Success(data))
        } catch (e: Throwable) {
            dispatch(AsyncAction.Failure(e))
        }
    }
}
```

Kotlin Coroutinesで
非同期処理を呼び出し

SideEffectでActionを監視する

```
val sideEffects = sideEffects<AsyncState, AsyncAction> {
    collect<AsyncAction.Request> { request ->
        try {
            val data = Api.request(request.id)
            dispatch(AsyncAction.Success(data))
        } catch (e: Throwable) {
            dispatch(AsyncAction.Failure(e))
        }
    }
}
```

非同期処理が完了したら
Actionを呼び出す

Flowも監視してActionに変換可能

```
sealed interface AsyncAction {
    object Initial : AsyncAction
    data class Event(val value: String) : AsyncAction
}

val sideEffects = sideEffects<AsyncState, AsyncAction> {
    collect<AsyncAction.Initial> {
        eventFlow.collect {
            dispatch(AsyncAction.Event(it))
        }
    }
}
```

4

Reducerを分割する

Reducerが肥大化する…

```
val counterReducer: Reducer<SampleState, SampleAction> =
    createReducer(SampleState()) { acc, action ->
        when(action) {
            SampleAction.Action1 -> /* ... */
            SampleAction.Action2 -> /* ... */
            SampleAction.Action3 -> /* ... */
            SampleAction.Action4 -> /* ... */
            SampleAction.Action5 -> /* ... */
            SampleAction.Action6 -> /* ... */
            /* ... */
        }
    }
```

際限なく増える
Action

Stateをネストさせる

```
data class SampleState(
    val state1: State1,
    val state2: State2
) {
    data class State1(
        val count: Int,
        /* ... */
    )

    data class State2(
        val text: String,
        /* ... */
    )
}
```

各Stateに対してReducerを作る

```
val reducer1 = createReducer<SampleState.State1, SampleAction>(
    SampleState.State1(/* ... */)
) { acc, action ->
    when(action) {
        /* ... */
    }
}

val reducer2 = createReducer<SampleState.State2, SampleAction>(
    SampleState.State2(/* ... */)
) { acc, action ->
    when(action) {
        /* ... */
    }
}
```

複数のReducerを組み合わせる

```
val combinedReducer = combineReducers(
    child(reducer1) { it.state1 },
    child(reducer2) { it.state2 },
) { state1, state2 ->
    SampleState(
        state1 = state1,
        state2 = state2,
    )
}
```

親Stateを分解

親Stateを組み立てる

まとめ

Reduxっぽいアーキテクチャを導入することで、
いくつかの課題が解決できそう
軽量なアーキテクチャなので、コピペして使うことができる

検討事項：
簡単な画面には不要な複雑さをもたらす
学習コストがかかる
標準的なアーキテクチャを逸脱することによる不安感