Creative Research and Development in 2022

【宣言的 UI と命令的 UI の比較】

Comparison of declarative UI and imperative UI

電気系コース 長田 陽太(OSADA Yota)

1 はじめに

私は、学校生活の中で、学校からの連絡がわかりづらいなどの不便な点を感じていた。そこで不便な点を解決するようなアプリケーションを開発しようと考えた。その中でアプリケーションにおいて、UI は非常に重要であることがわかった。その UI の実装方法には宣言的 UI と命令的 UI の 2 つがあり、どちらの手法が開発に適しているか調べることが本研究の目的である。

本資料ではそれらの比較や優位性、考察についてまとめる。

2 使用したツール

① Android Studio
② Kotlin
③ Jetpack Compose

3 比較

本研究では、同一機能を持つ単純なアプリケーションを、それぞれ命令的 UI と宣言的 UI で実装し、そのときに必要になったコードの量や、可読性などを比較する。

なお、命令的 UI では UI の記述のために XML ファイルが必要になるので、そちらも合わせて上記の比較を行っていく。

3.1 Hello World

命令的 UI

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

宣言的 UI

```
setContent {
    HelloWorldDeclarativeTheme {
        // A surface container using the 'background' color from the theme
        Surface(modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colorScheme.background) {
            Text(text = "Hello World!")
        }
    }
}
```

結果

命令的 UI の方は Kotlin から XML ファイルを呼び出しているだけなので、大きく書き方が異なってるが可読性にはそこまで差はない。

3.2 カウントアプリ

命令的 UI

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val countText = findViewById<TextView>(R.id.countText)
        val countUp = findViewById<Button>(R.id.countUp)
        val countDown = findViewById<Button>(R.id.countDown)
        val countReset = findViewById<Button>(R.id.countReset)

        var count = 0

        countUp.setOnClickListener { it: View!
            count++
            countText.text = "${count}回"
        }

        countDown.setOnClickListener { it: View!
            if (count > 0) count--
            countText.text = "${count}回"
        }

        countReset.setOnClickListener { it: View!
            count = 0
            countText.text = "${count}回"
        }
    }
}
```

```
TextView
    android:id="@+id/countText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="カウント:0"
    android:textSize="36sp"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"/>

Button
    android:id="@+id/countUp"
    style="@android:style/Widget.Material.Button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="104dp"
    android:text="カウントアップ"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/countText" />
Button
    android:text="カウントダウン"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/countDown"
    style="@android:style/Widget.Material.Button"
    android:layout_marginTop="16dp"
    app:layout_constraintTop_toBottomOf="@+id/countUp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"/>
Button
    android:text="カウントリセット"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/countReset"
    style="@android:style/Widget.Material.Button"
    android:layout_marginTop="16dp"
    app:layout_constraintTop_toBottomOf="@+id/countDown"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"/>
```

宣言的 UI

```
Column(
    modifier = Modifier.fillMaxSize(),
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.Center
) { this: ColumnScope
    var count by remember { mutableStateOf( value: 0) }
    Text(
        modifier = Modifier.padding(vertical = 64.dp),
        text = "カウント：$count"
    )
    TextButton(
        modifier = Modifier.padding(4.dp),
        onClick = { count++ }
    ) { this: RowScope
        Text(text = "カウントアップ")
    }
    TextButton(
        modifier = Modifier.padding(4.dp),
        onClick = { count-- }
    ) { this: RowScope
        Text(text = "カウントダウン")
    }
    Button(
        modifier = Modifier.padding(8.dp),
        onClick = { count = 0 }
    ) { this: RowScope
        Text(text = "カウントリセット")
    }
}
```

結果

スペースの都合上、宣言的 UI のコードを分割して表示する。少し応用的な内容になると、ソースコードの書き方が全く異なるものになる。宣言的 UI ではロジック部分が UI 部分の中に書くことができるため、どのボタンがどの役割をしているのかをソースコードを見るだけでわかるようになっている。

4 考察

小さなプログラムだとそこまで差はないが、大きなものになっていくにつれて、宣言的 UI で実装されたコードのほうが読みやすくなっていってるように感じた。やはり一つの言語やファイルで UI とロジックの双方の実装ができるのが大きいメリットだと考えられる。また、一つのソースファイルで管理できるという点から、保守性も向上するという利点もある。

しかし、宣言的 UI にも問題点は少なからずある。例えば、この手法自体、比較的新しいものなので情報が少なかったり、何も考えずに書いていくとネストが深くなってしまったりする問題がある。前者の問題は時間が経てば改善していくので、あとはこの書き方や考え方にいかにプログラマが慣れていく必要がある。保守性の向上が騒がれる現在で、この概念は今後ますます重要になっていくだろう。

5 まとめ

本研究を通して、UI の実装方法について知ることができた。宣言的 UI は宣言的プログラミングから着想を得ているようで、正しく書くことができれば非常に読みやすいコードを書ける。実際、ネストを浅くするように適宜関数化すると、どこにどのボタンがあるのかが分かりやすかった。また、Kotlin には引数の順序を変更できたり、ラムダ式の引数を関数の括弧の外に書くことができたりする機能がある。そういった機能のおかげで、コーディング中のストレスがかなり低く、コードもそれらの機能のおかげで非常に読みやすかった。さらに今回使用した IDE には、Java と Kotlin のソースコードを相互に変換する機能が備わっており、古いライブラリを使う際に大いに役立った。

ネストを深くしないようなか書き方さえできれば、宣言的 UI は非常に読みやすいコードになるので、勉強や実践がより大事になってくると思われる。ただし、命令的 UI のコードを宣言的 UI に置き換えるようなツールは今のところないので、古くからあるサービスは命令的 UI、今後できるサービスは宣言的 UI という住み分けになっていくと予想される。