

# Jetpack Compose 入门到精通

## 第一章 初识 Jetpack Compose

Jetpack Compose是用于构建原生Android UI的现代工具包。 Jetpack Compose使用更少的代码，强大的工具和直观的Kotlin API，简化并加速了Android上的UI开发。这是Android Developers 官网对它的描述。

本文不是教你Jetpack Compose 的一些基本使用方法，而是为啥我们需要Jetpack Compose 的一些简洁，让我们对Jetpack Compose 有更深层次的了解。如果你想看Jetpack Compose 的快速上手和基本使用，请看我前面的文章

[Android Jetpack Compose 最全上手指南](#)

### 1.1 为什么我们需要一个新的UI 工具？

在Android中，UI工具包的历史可追溯到至少10年前。自那时以来，情况发生了很大变化，例如我们使用的设备，用户的期望，以及开发人员对他们所使用的开发工具和语言的期望。

以上只是我们需要新UI工具的一个原因，另外一个重要的原因是 `View.java` 这个类实在是太大了，有太多的代码，它大到你甚至无法在Githubs上查看该文件，因为它实际上包含了 30000 行代码，这很疯狂，而我们所使用的几乎每一个Android UI 组件都需要继承于View。

GogleAndroid团队的Anna-Chiara表示，他们对已经实现的一些API感到遗憾，因为他们也无法在不破坏功能的情况下收回、修复或扩展这些API，因此现在是一个崭新起点的好时机。

这就是为什么Jetpack Compose 让我们看到了曙光。

### 1.2 Jetpack Compose的着重点

包括一下几个方面：

- 1. 加速开发
- 1. 强大的UI工具
- 1. 直观的Kotlin API

#### 1.2.1 加速开发

如果你是一个初级开发工程师，你总是希望有更多的时间来写业务逻辑，而不是花时间在一些如：动画、颜色变化等事情上，看看下面这个View:



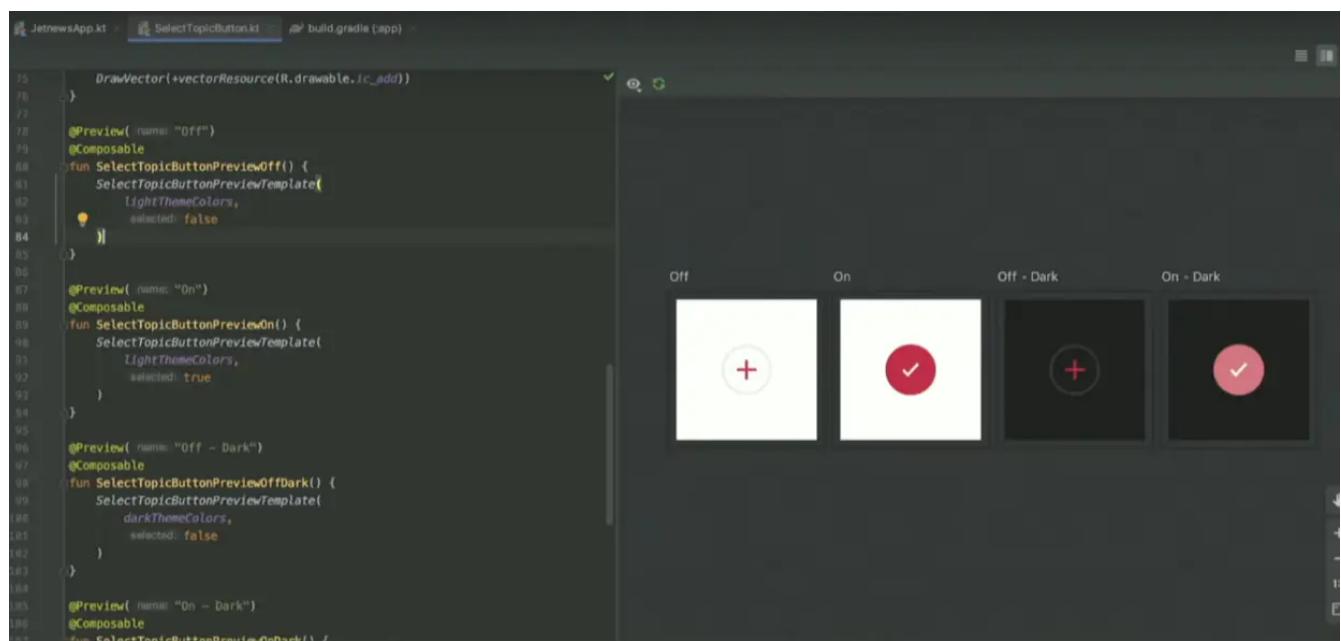
这个Material Edit-text 似乎看起来很简单，但是其实背后有许多东西需要关注，比如：动画、颜色改变、状态管理等等。

而Jetpack Compose 为我们提供了很多开箱即用的Material 组件，如果的APP是使用的material设计的话，那么使用Jetpack Compose 能让你节省不少精力。

### 1.2.2 强大的UI工具

没有正确工具的UI工具包是无用的，因此从过去10年的经验中能学到不少，Jetpack Compose 团队开始和 JetBrains 合作，以提供开发者强大的工具包，在Android Studio 上大规模的支持Compose 能力。

看一看在Android Studio上的表现：



上图是使用Jetpack Compose 开发UI时，在Android Studio 上的预览，你可以看到，在左边编码时，右边你能同时展现UI即时预览，比如在明/暗模式下的状态切换，都能在右边及时展示出来。

它与我们现在使用的Android Studio 中的 `text/Design` 相似，但是它更加先进，使用很简单，这个功能只能在 Android Studio4.0以上预览版，开发compose 时使用。

The screenshot shows the Android Studio interface with the XML code editor open. The code defines a series of `<TextView>` elements with specific styles and text content. Below the code editor is a toolbar with two tabs: "Design" and "Text", with "Text" being the active tab. To the right of the code editor is a preview window showing a white screen with a small blue icon in the top-left corner.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <merge xmlns:android="http://schemas.android.com/apk/res/android">
3
4     <TextView
5         style="@style/Widget.Shrine.Button.TextButton"
6         android:layout_width="wrap_content"
7         android:layout_height="wrap_content"
8         android:text="Featured" />
9
10    <TextView
11        style="@style/Widget.Shrine.Button.TextButton"
12        android:layout_width="wrap_content"
13        android:layout_height="wrap_content"
14        android:text="Apartment" />
15
16    <TextView
17        style="@style/Widget.Shrine.Button.TextButton"
18        android:layout_width="wrap_content"
19        android:layout_height="wrap_content"
20        android:text="Accessories" />
21
22    <TextView
23        style="@style/Widget.Shrine.Button.TextButton"
24        android:layout_width="wrap_content"
25        android:layout_height="wrap_content"
26        android:text="Shoes" />
27
28    <TextView
29        style="@style/Widget.Shrine.Button.TextButton"
30        android:layout_width="wrap_content"
31        android:layout_height="wrap_content"
32        android:text="Tops" />
33
34    <TextView
35        style="@style/Widget.Shrine.Button.TextButton"
36        android:layout_width="wrap_content"
37        android:layout_height="wrap_content"
38        android:text="Bottoms" />
39
40    <TextView
41        style="@style/Widget.Shrine.Button.TextButton"
42        merge ... TextView
```

### 1.2.3 直观的Kotlin API

对于开发者而言，Jetpack Compose 的用途不仅仅是Android UI，因此用Kotlin来编写他们并开源。当然，所有Android代码都是开源的，但特别强调的是Compose代码，它每天在这里更新（<https://android.googlesource.com/platform/frameworks/opt/compose/>）。因此，您可以查看和使用代码，同时也可以在此处提供反馈。

由于Compose仍在开发之中，因此每个开发人员的反馈都很重要。

## 1.3 API 设计

十多年来，Android团队在创建API和审查API方面拥有丰富的经验，但有一个收获-他们使用Java作为编程语言。但有一个问题-他们使用的是Java作为开发语言。



Jetpack Compose是第一个使用Kotlin正在开发中的大型项目，因此Android团队正在探索Kotlin API指南的新世界，以创建一组特定于Compose API的指南，该工作仍在进行中，仍然有很长的路要走。

## 1.4 Compose API 的原则

### 1.4.1 一切都是函数

正如我前面的文章所说，Compose是一个声明式UI系统，其中，我们用一组函数来声明UI，并且一个Compose函数可以嵌套另一个Compose函数，并以树的结构来构造所需要的UI。

在Compose中，我们称该树为UI图，当UI需要改变的时候会刷新此UI图，比如Compose函数中有 if 语句，那么Kotlin编译器就需要注意了。

### 1.4.2 顶层函数 ( Top-level function )

在Compose的世界中，没有类的概念，全都是函数，并且都是顶层函数，因此不会有任何继承和层次机构问题。

```
@Composable  
fun checkbox ( ... )  
  
@Composable  
fun TextView ( ... )  
  
@Composable  
fun EditText ( ... )  
  
@Composable  
fun Image ( ... )
```

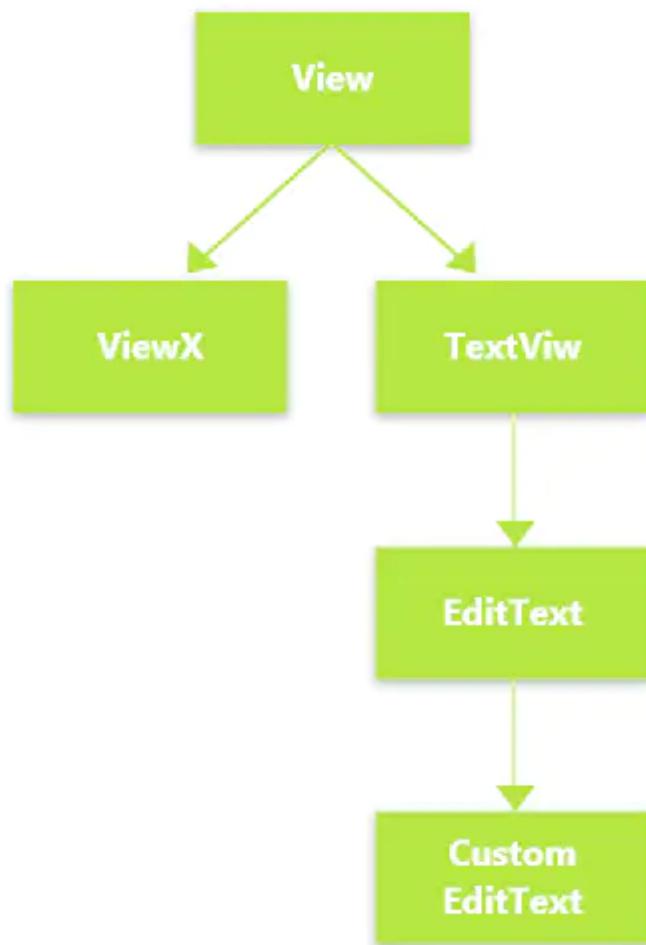
复制代码

在此过程中，Compose函数始终根据接收到的输入生成相同的UI，因此，放弃类结构不会有任何害处。从类结构构建UI过渡到顶层函数构建UI对开发者和Android 团队都是一个巨大的转变，顶层函数还在讨论之中，还没有发布 release 版。

### 1.4.3 组合优于继承

Jetpack Compose首选组合而不是继承。Compose会基于其他部分构建UI，但不会继承行为。

如果你经常关注Android或者对Android有所了解，你就会知道，Android中的几乎所有组件都继承于View类（直接或间接继承）。比如 `EditText` 继承于 `TextView`，而同时 `TextView` 又继承于其他一些View这样的继承机构最终会指向跟View即 `view.java`。并且 `view.java` 又非常多的功能。



而Compose团队则将整个系统从继承转移到了顶层函数。`Textview`，`EditText`，复选框 和所有UI组件都是它们自己的Compose函数，而它们构成了要创建UI的其他函数，代替了从另一个类继承。

Toggleable

Text

EditText

MyCustomTextview

#### 1.4.4. 信任单一来源

信任单一来源是构建整个Jetpack Compose一项非常重要的特性。如果您习惯了现有的UI工具包，那么您可能会知道执行点击的工作原理。如下代码所示：

```
@Override  
public boolean performClick(){  
    setChecked(!mChecked);  
    final boolean handled = super.performClick();  
    ...  
}
```

复制代码

首先，它改变view的状态，然后执行动作，这会导致许多bug，例如复选框，因为它首先从已选中状态变为未选中，反之亦然，然后由于某种原因，如果操作失败，开发人员必须手动分配先前的状态。

而在Compose中呢，功能正好相反。在此，复选框等功能具有两个参数。一个是在UI中显示状态，另一个是lambda函数，用于观察UI应相应更改的状态变化。

```
@Composable  
fun Checkbox(checked : Boolean,  
            onCheckedChange : ((Boolean) -> Unit)),  
            ....)
```

复制代码

## 1.5 深入了解Compose

## **Compose UI Material**

Surface, Buttons, Tabs, Themes

## **Compose UI Foundation**

Standard layouts, interactions

## **Compose UI Core**

Input, measure, layout, Drawing

## **Compose Runtime**

Tree management, Effects

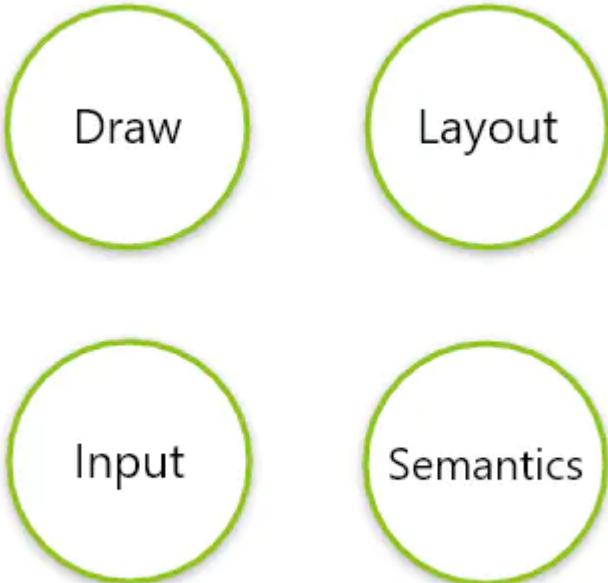
如上图所示，Compose在运行时分为四个部分。让我们——看一下。

### 1.5.1 Core

顾名思义，这是Compose的核心，如果您不想深入学习，可以跳过它。

基本上，核心包含四个构建模块：

- 绘制(Draw)
- 布局(Layout)
- 输入(Input)
- 语义(Semantics)



- 1、Draw — Draw 给了你访问Canvas的能力，因此你可以绘制你要的任何自定义View
- 2、Layout — 通过布局，我们可以测量事物并相应地放置视图。
- 3、Input — 开发人员可以通过输入访问事件并执行手势
- 4、Semantics — 我们可以提供有关树的语义信息。

### 1.5.2. Foundation

Foundation的核心是收集上面提到的所有内容，并共同创建一个 抽象层，以使开发人员更轻松调用。

### 1.5.3 Material

在这一层，所有的Material组件将会被提供，并且我们可以通过提供的这些组件来构建复杂的UI。

这是Compose团队所做的出色工作中最精彩的部分，在这里，所有提供的View都有Material支持，因此，使用Compose来构建APP，默认就Material风格的，这使得开发者少了很多工作。

## 1.6 插槽API

插槽API的出现是为了给开发人员留出了很多空间，以便他们可以执行所需的任何自定义操作，Android团队试图猜测开发人员可能会想到的许多自定义设置，但他们无法一直想象开发人员的想法，例如使用带 `drawable` 的 `TextView`。

因此，Compose团队为组件留出了空间，以便开发人员可以执行所需的任何操作，例如 使用按钮。你可以 保留文本 或 带有图标的文本 或 所需的任何内容，如下所示

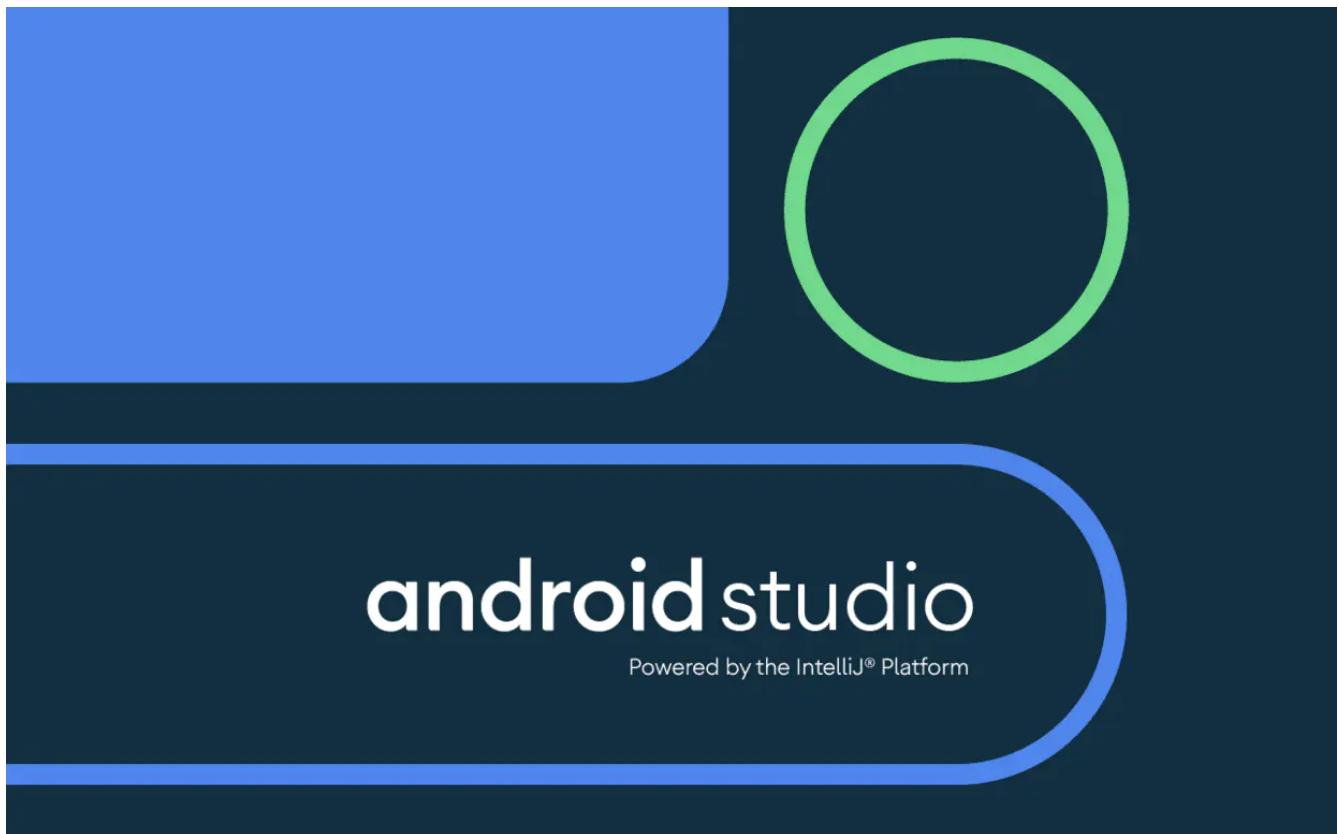
[Star](#) [Star](#)

## 第二章 Jetpack Compose构建Android UI

### 2.1 Android Jetpack Compose 最全上手指南

#### 2.1.1 Jetpack Compose 环境准备和Hello World

每当我们学习一门新的语言，我们都是从一个 `hello world` 开始，今天我们也从一个 `hello world` 来开始 Jetpack Compose 吧！要想获得Jetpack Compose 的最佳体验，我们需要下载最新版本的Android Studio 预览版本（即Android Studio 4.0）。因为Android Studio 4.0 添加了对Jetpack Compose 的支持，如新的Compose 模版和Compose 及时预览。



使用Jetpack Compose 来开始你的开发工作有2种方式：

- 将Jetpack Compose 添加到现有项目
- 创建一个支持Jetpack Compose的新应用

接下来分别介绍一下这两种方式。

## 1. 将Jetpack Compose 添加到现有项目

如果你想在现有的项目中使用Jetpack Compose，你需要配置一些必须的设置和依赖：

### (1) gradle 配置

在app目录下的 `build.gradle` 中将app支持的最低API 版本设置为21或更高，同时开启Jetpack Compose `enable` 开关，代码如下：

```
android {  
    defaultConfig {  
        ...  
        minSdkVersion 21  
    }  
  
    buildFeatures {  
        // Enables Jetpack Compose for this module  
        compose true  
    }  
    ...  
  
    // Set both the Java and Kotlin compilers to target Java 8.  
  
    compileOptions {  
        sourceCompatibility JavaVersion.VERSION_1_8  
        targetCompatibility JavaVersion.VERSION_1_8  
    }  
  
    kotlinOptions {  
        jvmTarget = "1.8"  
    }  
}
```

复制代码

### (2) 使用试验版Kotlin-Gradle 插件

Jetpack Compose 需要试验版的 `Kotlin-Gradle` 插件，在根目录下的 `build.gradle` 添加如下代码：

```
buildscript {  
    repositories {  
        google()  
        jcenter()  
        // To download the required version of the Kotlin-Gradle plugin,  
        // add the following repository.  
        maven { url 'https://dl.bintray.com/kotlin/kotlin-eap' }  
    ...  
    dependencies {  
        classpath 'com.android.tools.build:gradle:4.0.0-alpha01'  
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:1.3.60-eap-25'  
    }  
}
```

```
allprojects {
    repositories {
        google()
        jcenter()
        maven { url 'https://dl.bintray.com/kotlin/kotlin-eap' }
    }
}
```

复制代码

### (3) 添加Jetpack Compose工具包依赖项

在app目录下的 build.gradle 添加Jetpack Compose 工具包依赖项，代码如下：

```
dependencies {
    // You also need to include the following Compose toolkit dependencies.
    implementation 'androidx.ui:ui-tooling:0.1.0-dev02'
    implementation 'androidx.ui:ui-layout:0.1.0-dev02'
    implementation 'androidx.ui:ui-material:0.1.0-dev02'
    ...
}
```

复制代码

ok，到这儿准备工作就完毕，就可以开始写代码了，但是前面说了，还有一种方式接入Jetpack Compose，我们来一起看看。

## 2. 创建一个支持Jetpack Compose的新应用

比起在现有应用中接入Jetpack Compose，创建一个支持Jetpack Compose 的新项目则简单了许多，因为Android Studio 4.0 提供了一个新的Compose 模版，只要选择这个模版创建应用，则所有上面的那些配置项都自动帮我们完成了。

创建一个支持Jetpack Compose 的应用，如下几个步骤就可以了：

- 1. 如果你在Android Studio的欢迎窗口，点击 `Start a new Android Studio project`，如果你已经打开了Android Studio 项目，则在顶部菜单栏选择 `File > New > New Project`
- 1. 在 `Select a Project Template` 窗口，选择 `Empty Compose Activity` 并且点击下一步
- 1. 在 `Configure your project` 窗口，做如下几步：
  - a. 设置 项目名称，包名 和 保存位置
  - b. 注意，在语言下来菜单中，**Kotlin** 是唯一一个可选项，因为Jetpack Compose 只能用Kotlin来写的才能运行。
  - c. `Minimum API level` 下拉菜单中，选择21或者更高
- 1. 点击 `Finish`

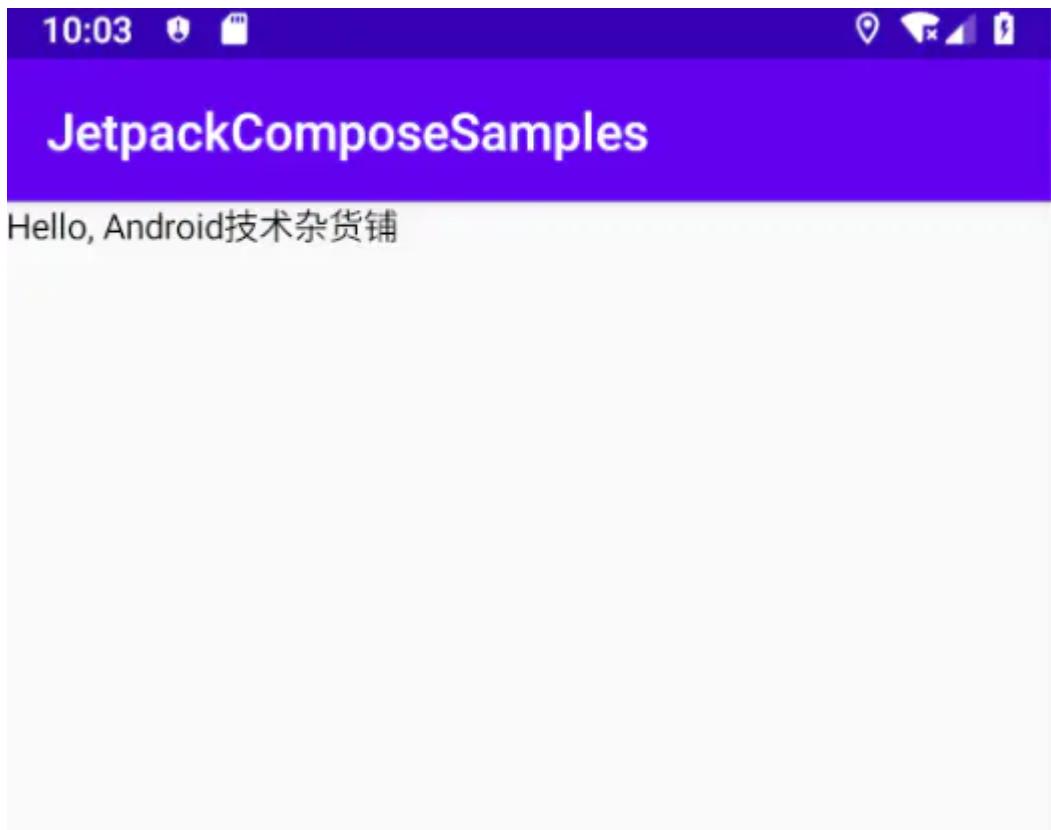
现在，你就可以使用Jetpack Compose 来编写你的应用了。

## 3. Hello world

在 `MainActivity.kt` 中添加如下代码：

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello, Android技术杂货铺")
        }
    }
}
```

复制代码



Jetpack Compose是围绕 `composable` 函数来构建的。这些函数使你可以通过描述应用程序的形状和数据依赖，以编程方式定义应用程序的UI，而不是着眼于UI的构建过程。要创建 `composable` 函数，只需要在函数名前面加上一个 `@composable` 注解即可，上面的 `Text` 就是一个 `composable` 函数。

#### 4. 定义一个 `composable` 函数

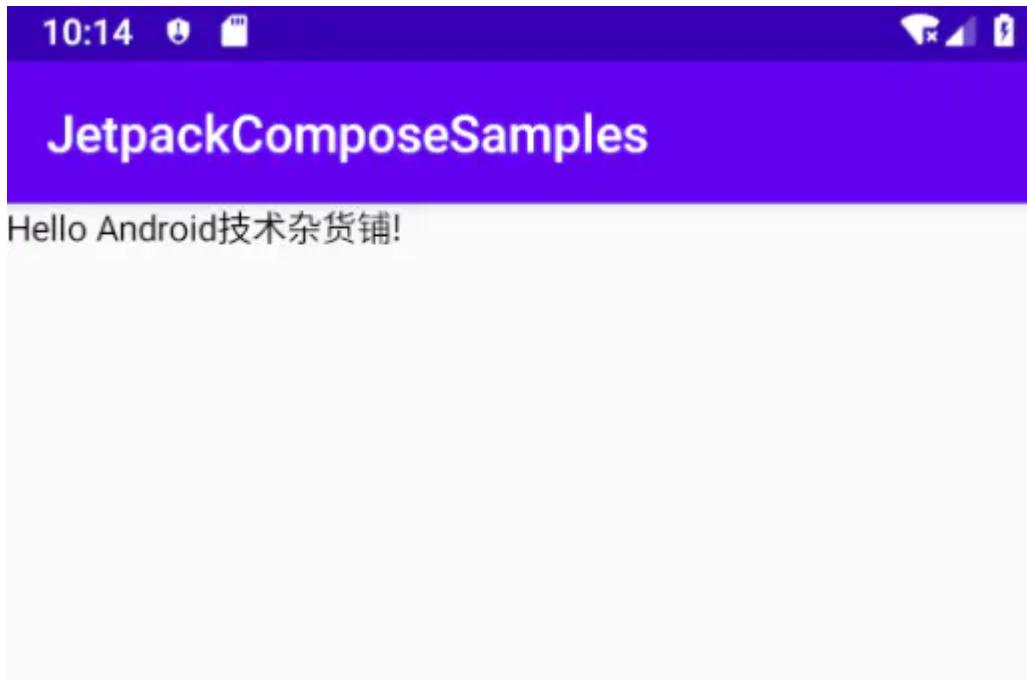
一个 `composable` 函数只能在另一个 `composable` 函数的作用域内调用，要使一个函数变为 `composable` 函数，只需在函数名前加上 `@composable` 注解，我们把上面的代码中，`setContent` 中的部分移到外面，抽取出一个 `composable` 函数中，然后传递一个参数 `name` 给 `text` 元素。

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Greeting("Android技术杂货铺")
        }
    }
}
```

```
    }
}

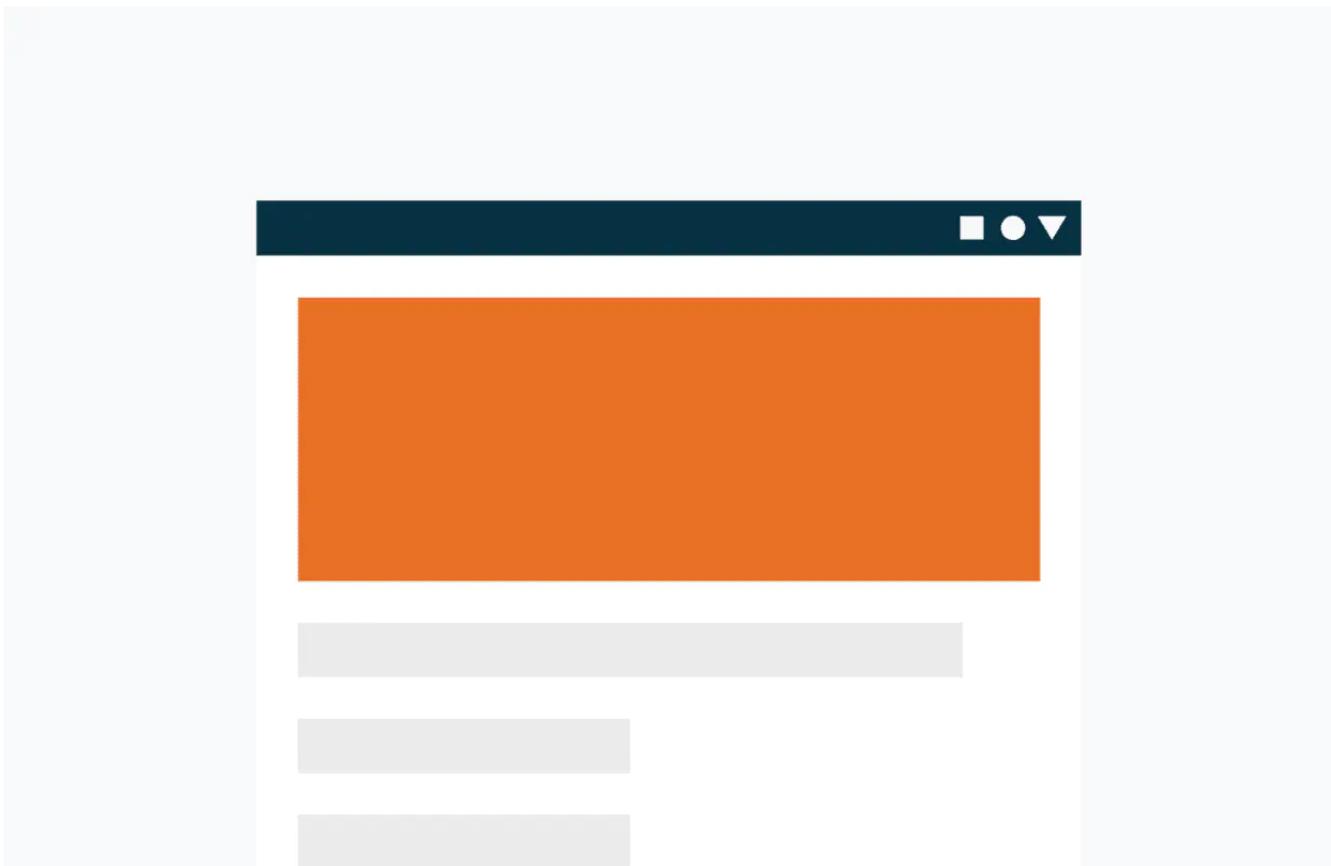
@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}
```

复制代码



## 2.1.2 布局

UI元素是分层级的，元素包含在其他元素中。在Jetpack Compose中，你可以通过从其他 `composable` 函数中调用 `composable` 函数来构建UI层次结构。



在Android的xml布局中，如果要显示一个垂直结构的布局，最长用的就是`LinearLayout`，设置`android:orientation`值为`vertical`，子元素就会垂直排列，那么，在Jetpack Compose中，如何来实现垂直布局呢？先添加几个`Text`来看一下。

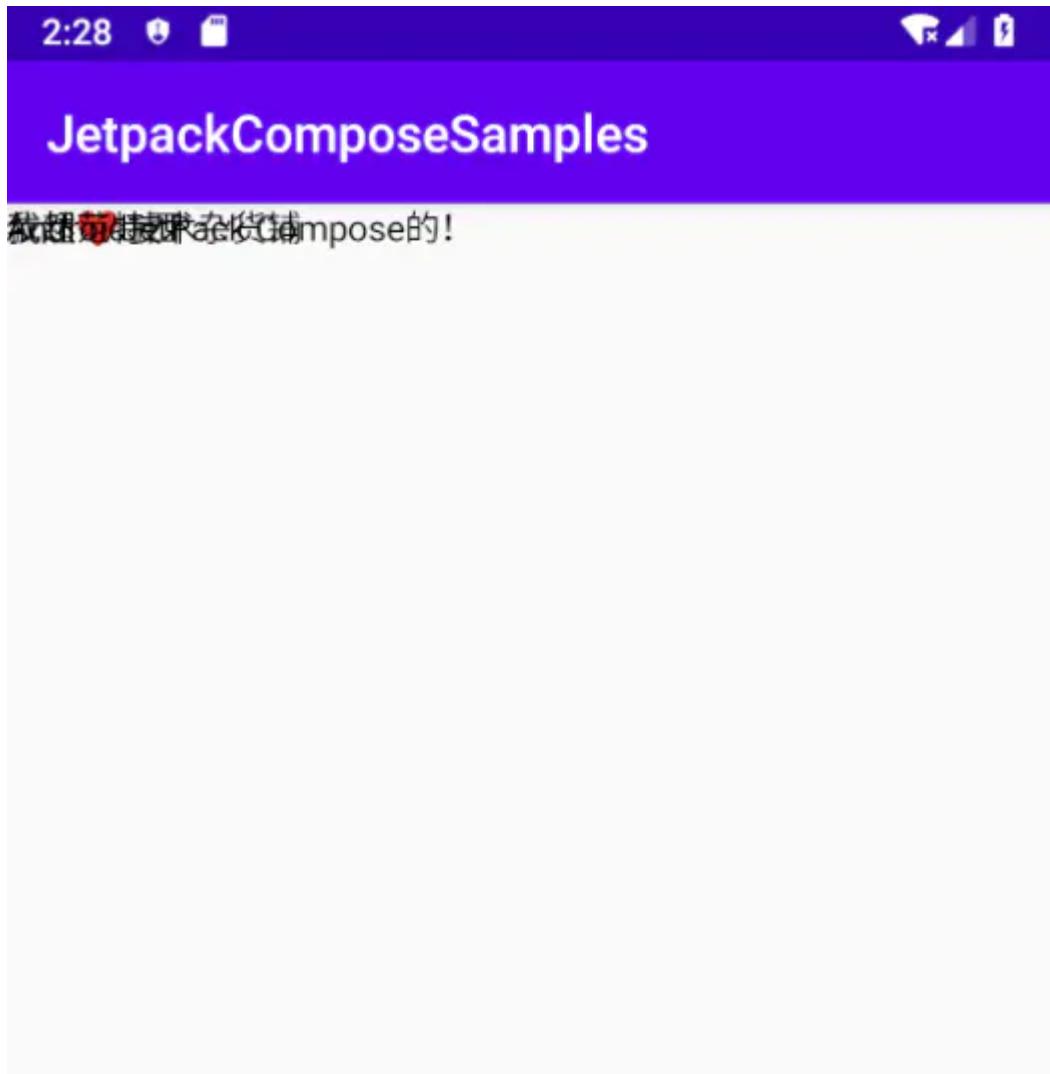
## 1. 添加多个Text

在上面的例子中，我们添加了一个`Text`显示文本，现在我们添加三个文本，代码如下：

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

@Composable
fun NewsStory() {
    Text("我超❤️JetPack Compose的！")
    Text("Android技术杂货铺")
    Text("依然范特西")
}
```

复制代码



从上图可以看到，我们添加了3个文本，但是，由于我们还没有提供有关如何排列它们的任何信息，因此三个文本元素相互重叠绘制，使得文本不可读。

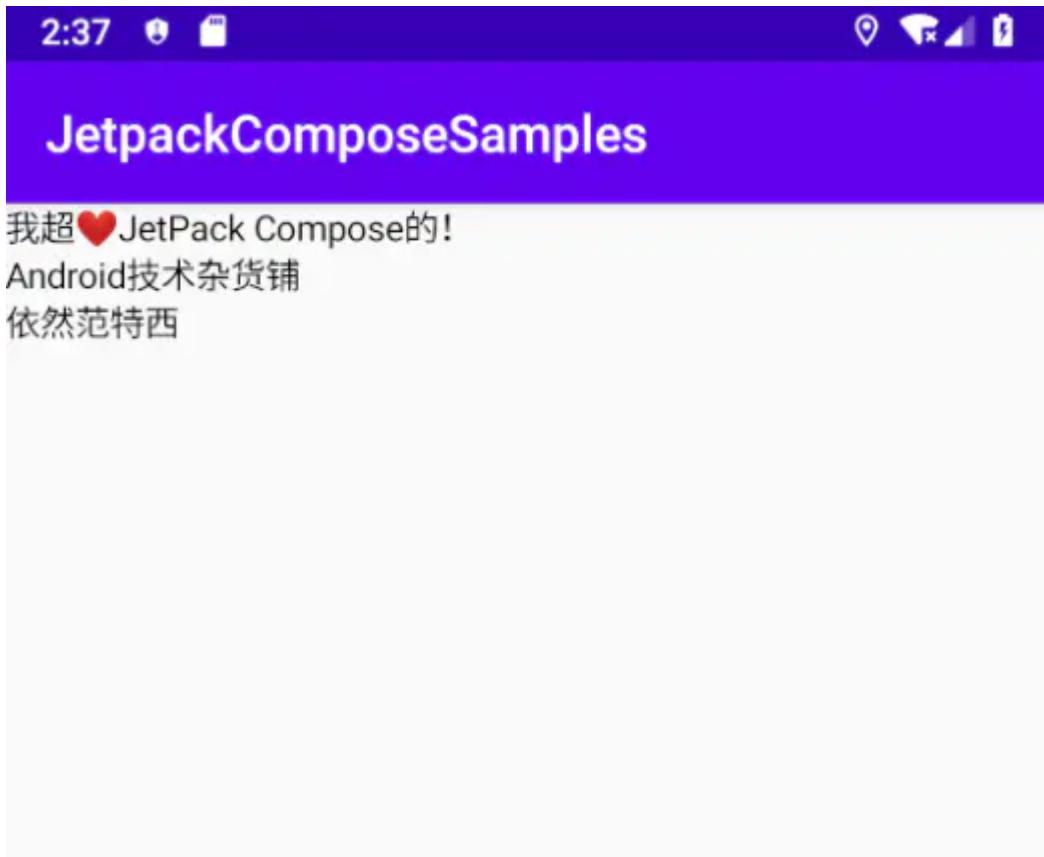
## 2. 使用Column

要使重叠绘制的 `Text` 文本能够垂直排列，我们就需要使用到 `Column` 函数，**写过flutter的同学看起来是不是很眼熟？是的，跟flutter里面的Column Widget 名字和功能完全一样，甚至连他们的属性都一摸一样。**

```
@Composable
fun NewsStory() {
    Column { // 添加Column，使布局垂直排列
        Text("我超❤️JetPack Compose的!")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

复制代码

效果如下：



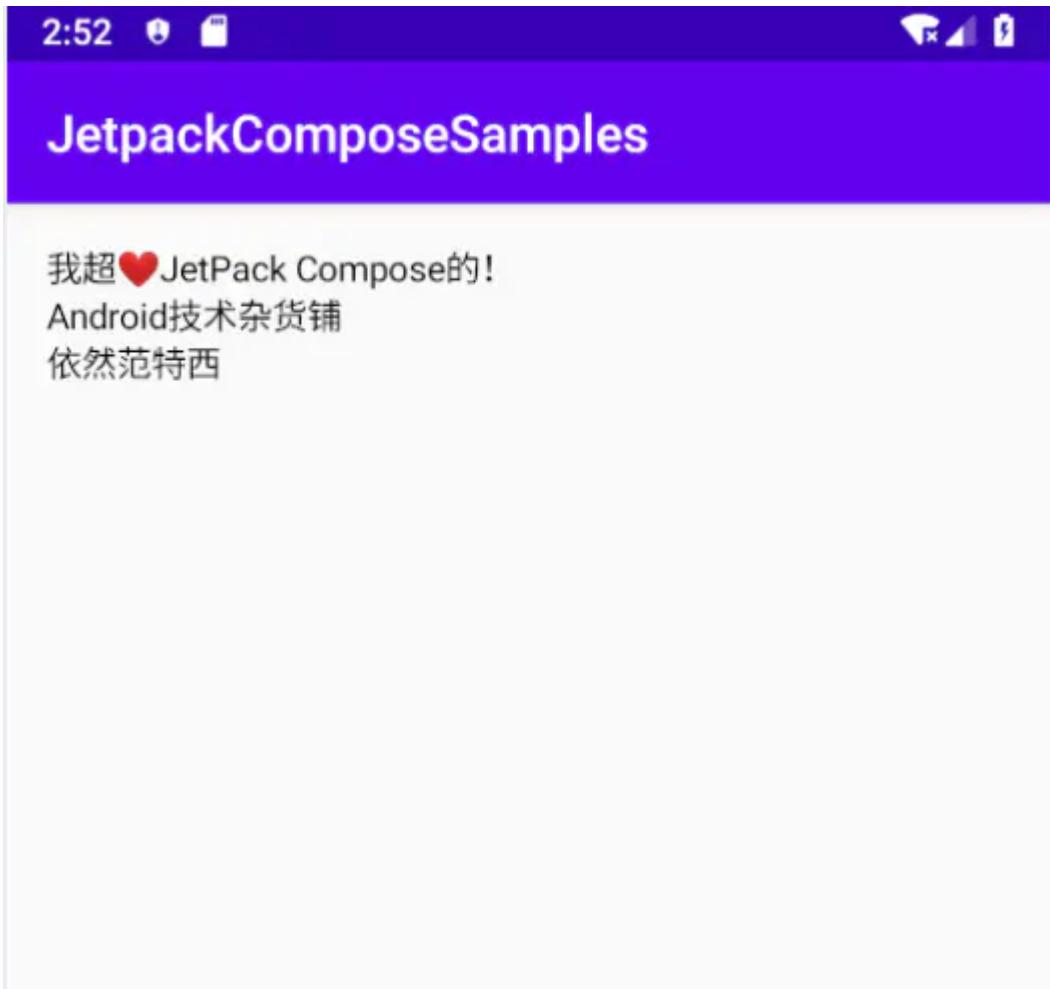
可以看到，前面重叠的布局，现在已经垂直排列了，但是，默认情况下，从左上角开始，一个接一个的排列，没有任何间距。接下来，我们给 `Column` 设置一些样式。

### 3. 给 `Column` 添加样式

在调用 `Column( )` 时，可以传递参数给 `Column( )` 来配置 `Column` 的大小、位置以及设置子元素的排列方式。

```
@Composable
fun NewsStory() {
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        Text("我超❤️ JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

复制代码



如上图所示，我们填充了padding，其他效果几乎一摸一样，上面代码中的设置属性解释如下：

- `crossAxisSize`: 指定 `Column` 组件（注：Compose 中，所有的组件都是 `composable` 函数，文中的组件都是指代 `composable` 函数）在水平方向的大小，设置 `crossAxisSize` 为 `LayoutSize.Expand` 即表示 `Column` 宽度应为其父组件允许的最大宽度，相当于传统布局中的 `match_parent`，还有一个值为 `LayoutSize.Wrap`，看名字就知道，包裹内容，相当于传统布局中的 `wrap_content`。
- `modifier`: 使你可以进行其他格式更改。在这种情况下，我们将应用一个 `spacing` 修改器，该设置将 `Column` 与周围的视图产生间距。

#### 4. 如何显示一张图片？

在原来的安卓原生布局中，显示图片有相应的控件 `ImageView`，设置本地图片地址或者 `Bitmap` 就能展示，在 Jetpack Compose 中该如何显示图片呢？



我们先下载这张图片到本地，添加到资源管理器中，命名为 header.png，我们更改一下上面的 NewsStory() 方法，先从资源文件夹获取图片image，然后通过 DrawImage() 将图片绘制出来：

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.mipmap.header)
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        // 显示图片
        DrawImage(image)

        Text("我超❤️JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

复制代码

3:57



# JetpackComposeSamples

我超❤️JetPack Compose的！

Android技术杂货铺

依然范特西



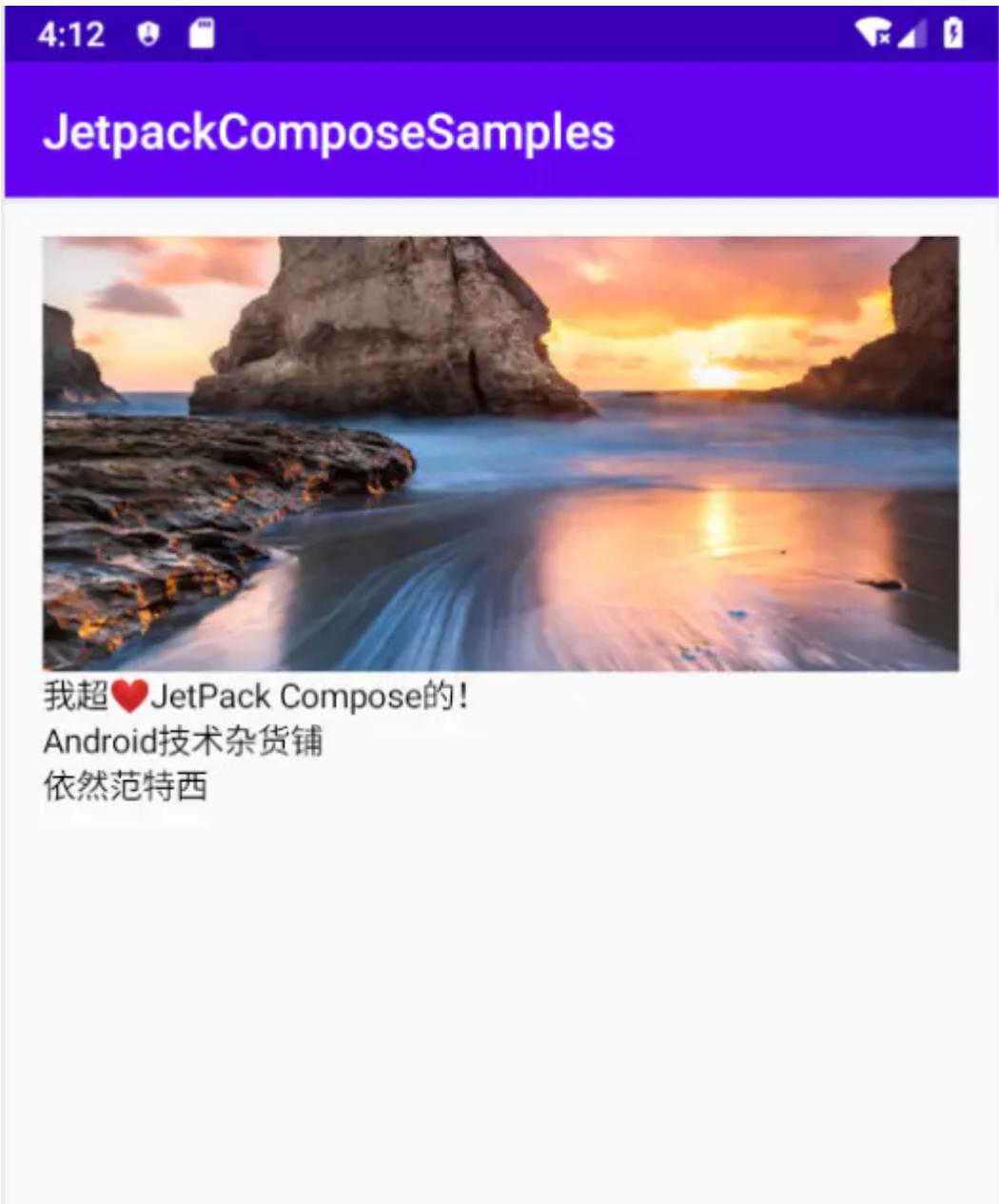
可以看到，图片不会按正确的比例显示，接下来，我们来修复它。

图片已添加到布局中，但会展开以填充整个视图，并和文本是拼叠排列，文字显示在上层。要设置图形样式，请将其放入 `Container` (又一个和flutter中一样的控件)

- `Container`: 一个通用的内容对象，用于保存和安排其他UI元素。然后，你可以将大小和位置的设置应用于容器。

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.drawable.header)
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        // 放在容器中，设置大小
        Container(expanded = true, height = 180.dp) {
            // 显示图片
            DrawImage(image)
        }
        Text("我超❤️JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

复制代码



- `expanded` : 指定Container的大小，默认是`false` ( Container的大小是子组件的大小，相当于`wrap_content` )，如果将它设置为`true`，就指定Container的大小为父控件所允许的最大size, 相当于`match_parent`。
- `height` : 设置Container容器的高度，`height`属性的优先级高于`expanded`,因此会覆盖`expanded`，如上面的例子，设置`height`为`180dp`,也就是容器宽为父控件宽度，高为`180dp`

## 5. 添加间距 Spacer

我们看到，图片和文本之间没有间距，传统布局中，我们可以添加`Margin`属性，设置间距，在Jetpack Compose 中，我们可以使用`HeightSpacer()` 和`widthSpacer()` 来设置垂直和水平间距

```
HeightSpacer(height = 20.dp) //设置垂直间距20dp  
widthSpacer(width = 20.dp) // 设置水平间距20dp
```

复制代码

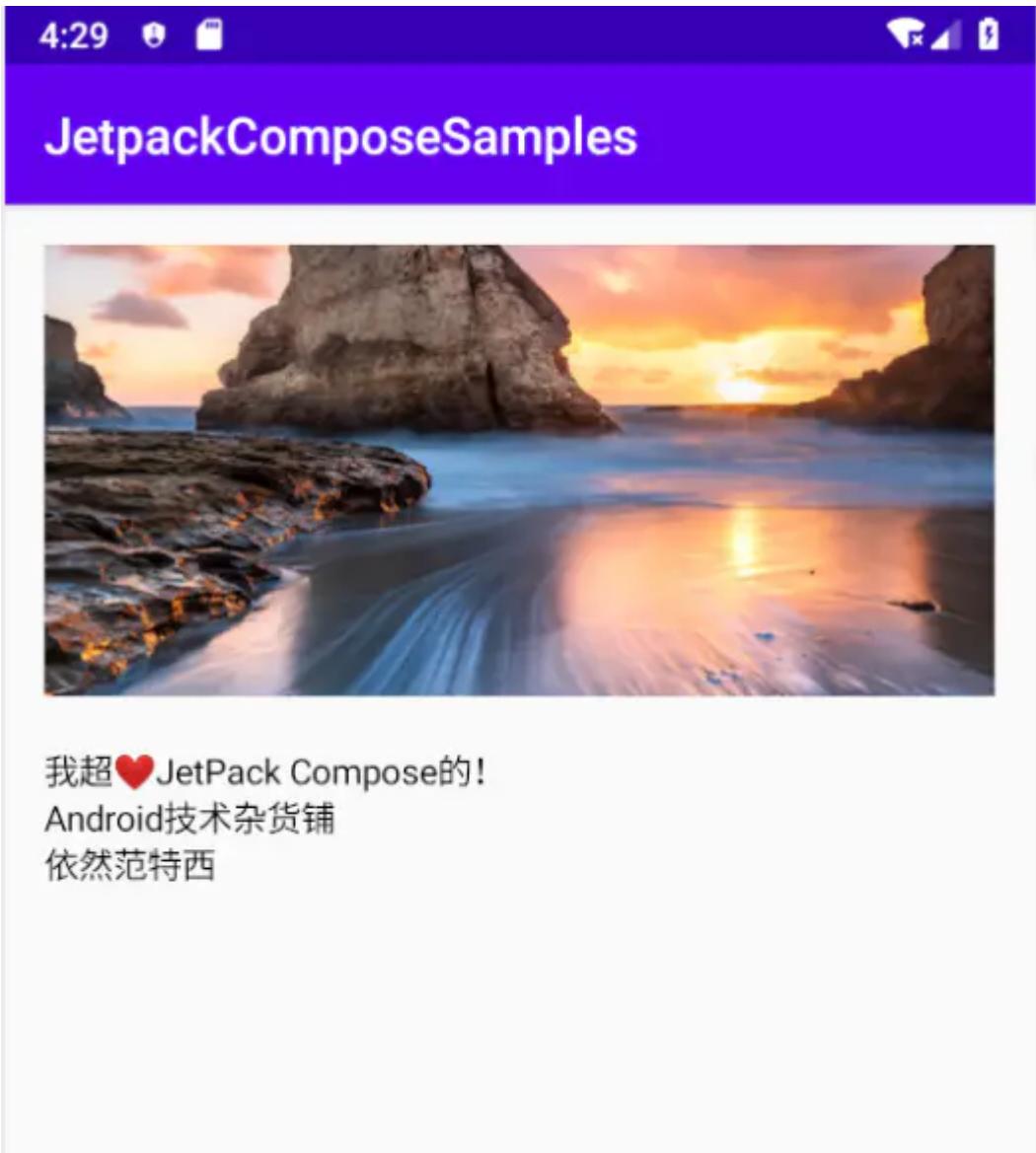
在上面的例子中，我们来为图片和文本之间添加 20dp 的间距：

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.mipmap.header)
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        // 放在容器中，设置大小
        Container(expanded = true, height = 180.dp) {
            // 显示图片
            DrawImage(image)
        }

        HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp

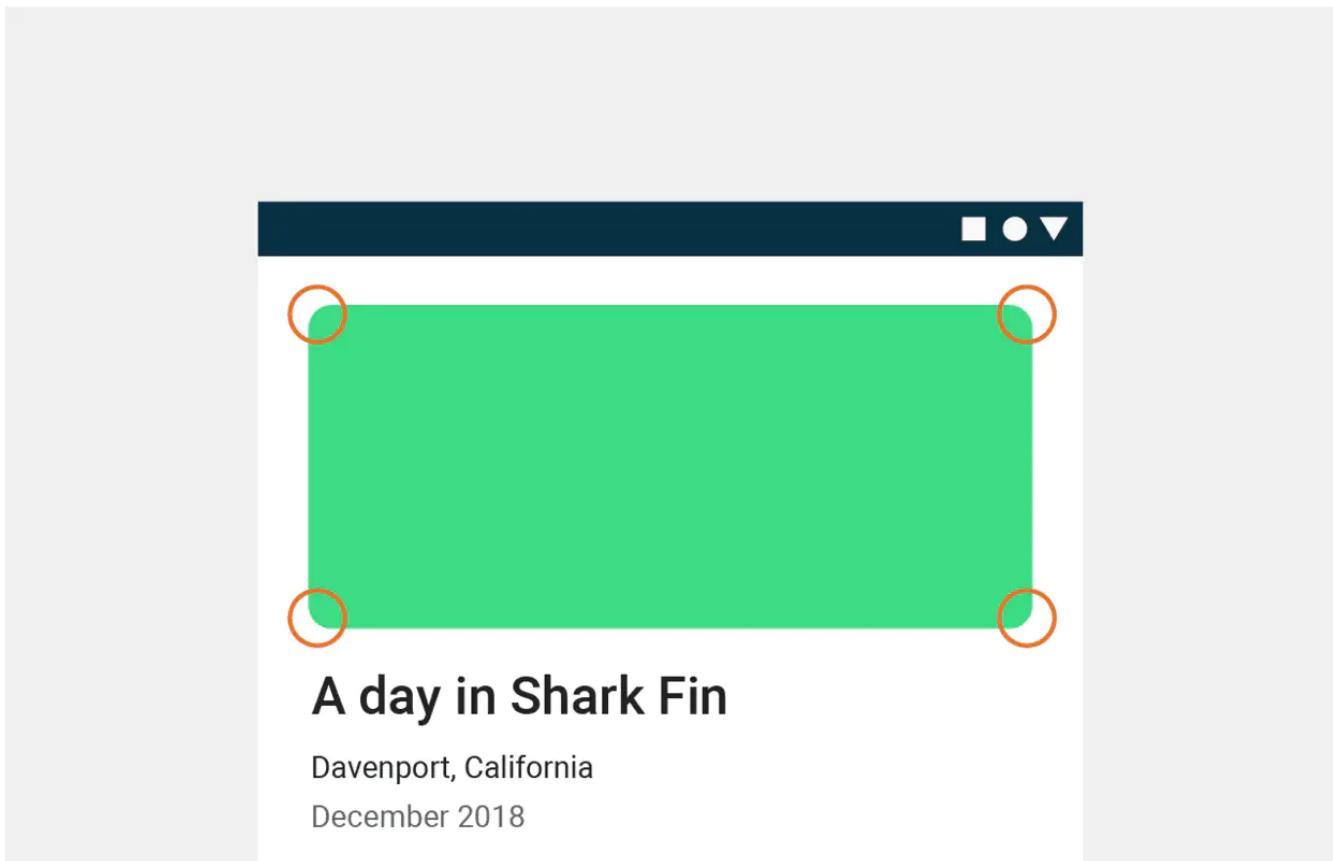
        Text("我超❤️JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

复制代码



### 2.1.3 使用Material design 设计

Compose 旨在支持Material Design 设计原则，许多组件都实现了Material Design 设计，可以开箱即用，在这一节中，将使用一些Material小组件来对app进行样式设置



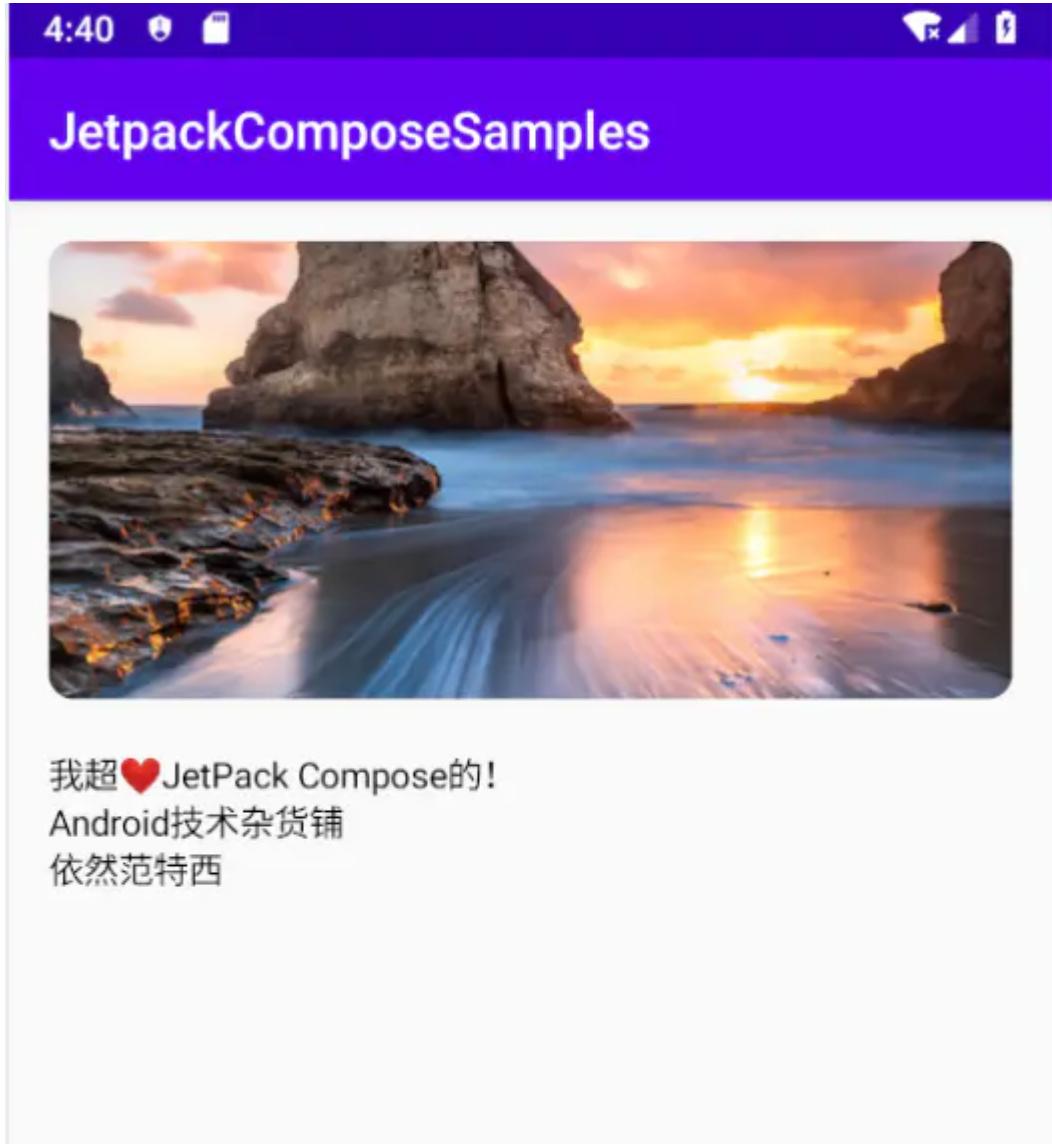
## 1. 添加 Shape 样式

`shape` 是 Material Design 系统中的支柱之一，我们来用 `clip` 函数对图片进行圆角裁剪。

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.drawable.header)
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        // 放在容器中，设置大小
        Container(expanded = true, height = 180.dp) {
            Clip(shape = RoundedCornerShape(10.dp)) {
                // 显示图片
                DrawImage(image)
            }
        }
        HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp
        Text("我超❤️JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

```
}
```

复制代码



形状是不可见的，但是我们的图片已经被裁剪成了设置的形状样式，因此如上图，图片已经有圆角了。

## 2. 给 `Text` 添加一些样式

通过Compose，可以轻松利用Material Design原则。将 `MaterialTheme()` 应用于创建的组件

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.mipmap.header)
    // 使用Material Design 设计
    MaterialTheme() {
        Column (
            crossAxisSize = LayoutSize.Expand,
```

```
    modifier = Spacing(16.dp)
) { // 添加Column，使布局垂直排列
    // 放在容器中，设置大小
    Container(expanded = true, height = 180.dp) {
        clip(shape = RoundedCornerShape(10.dp)) {
            // 显示图片
            DrawImage(image)
        }
    }
}

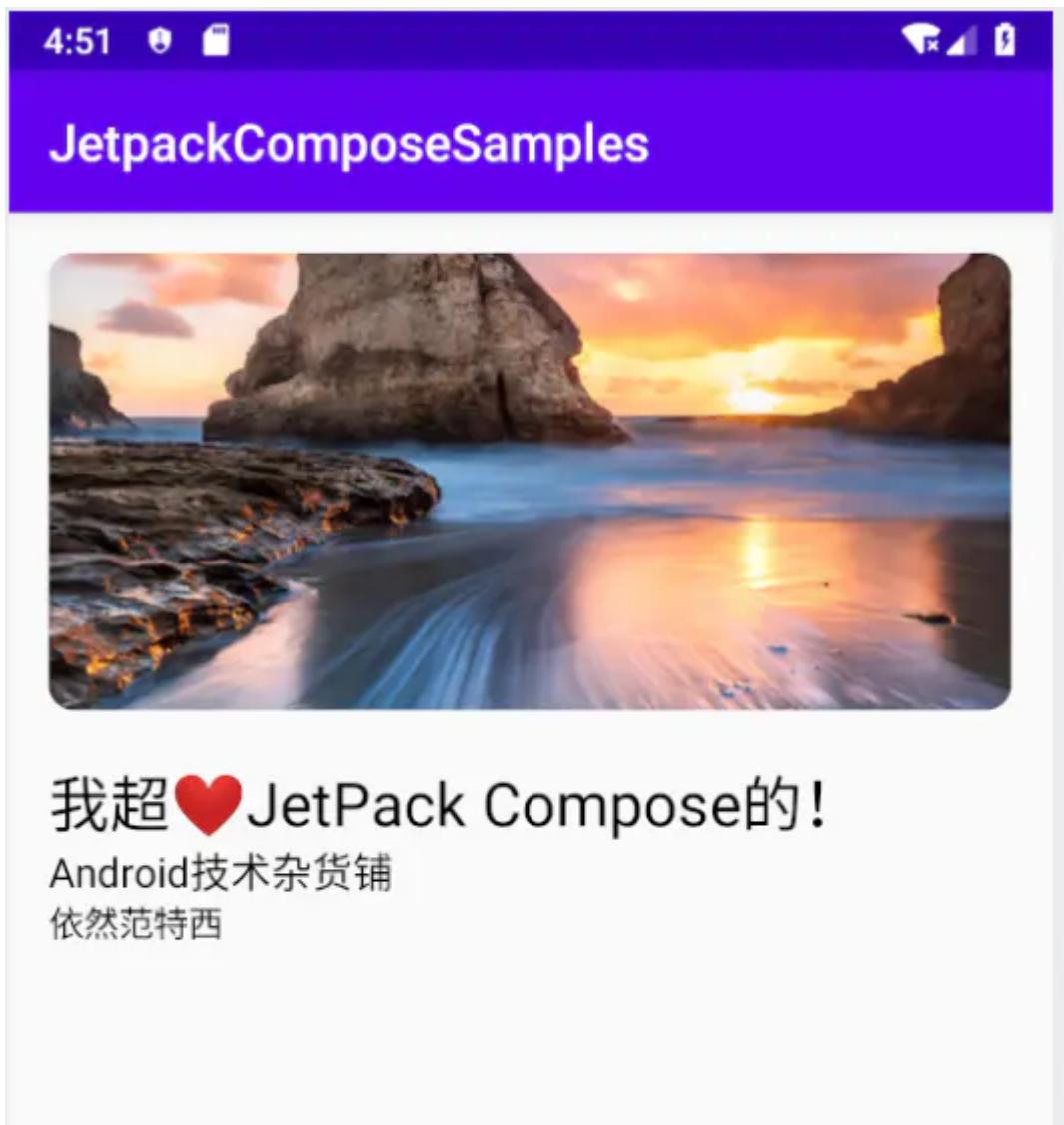
HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp

Text("我超❤JetPack Compose的！")
Text("Android技术杂货铺")
Text("依然范特西")
}

}
}
```

如上面的代码，添加了 `MaterialTheme` 后，重新运行，效果没有任何变化，文本现在使用了 `MaterialTheme` 的默认文本样式。接下来，我们将特定的段落样式应用于每个文本元素。

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.mipmap.header)
    // 使用Material Design 设计
    MaterialTheme() {
        column (
            crossAxisSize = LayoutSize.Expand,
            modifier = Spacing(16.dp)
        ) { // 添加column，使布局垂直排列
            // 放在容器中，设置大小
            Container(expanded = true, height = 180.dp) {
                clip(shape = RoundedCornerShape(10.dp)) {
                    // 显示图片
                    DrawImage(image)
                }
            }
        }
        HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp
        Text("我超❤️JetPack Compose的！", style = +themeTextStyle { h5 }) // 注意添加了
        style
        Text("Android技术杂货铺", style = +themeTextStyle { body1 }) // 注意添加了style
        Text("依然范特西", style = +themeTextStyle { body2 }) // 注意添加了style
    }
}
}
```

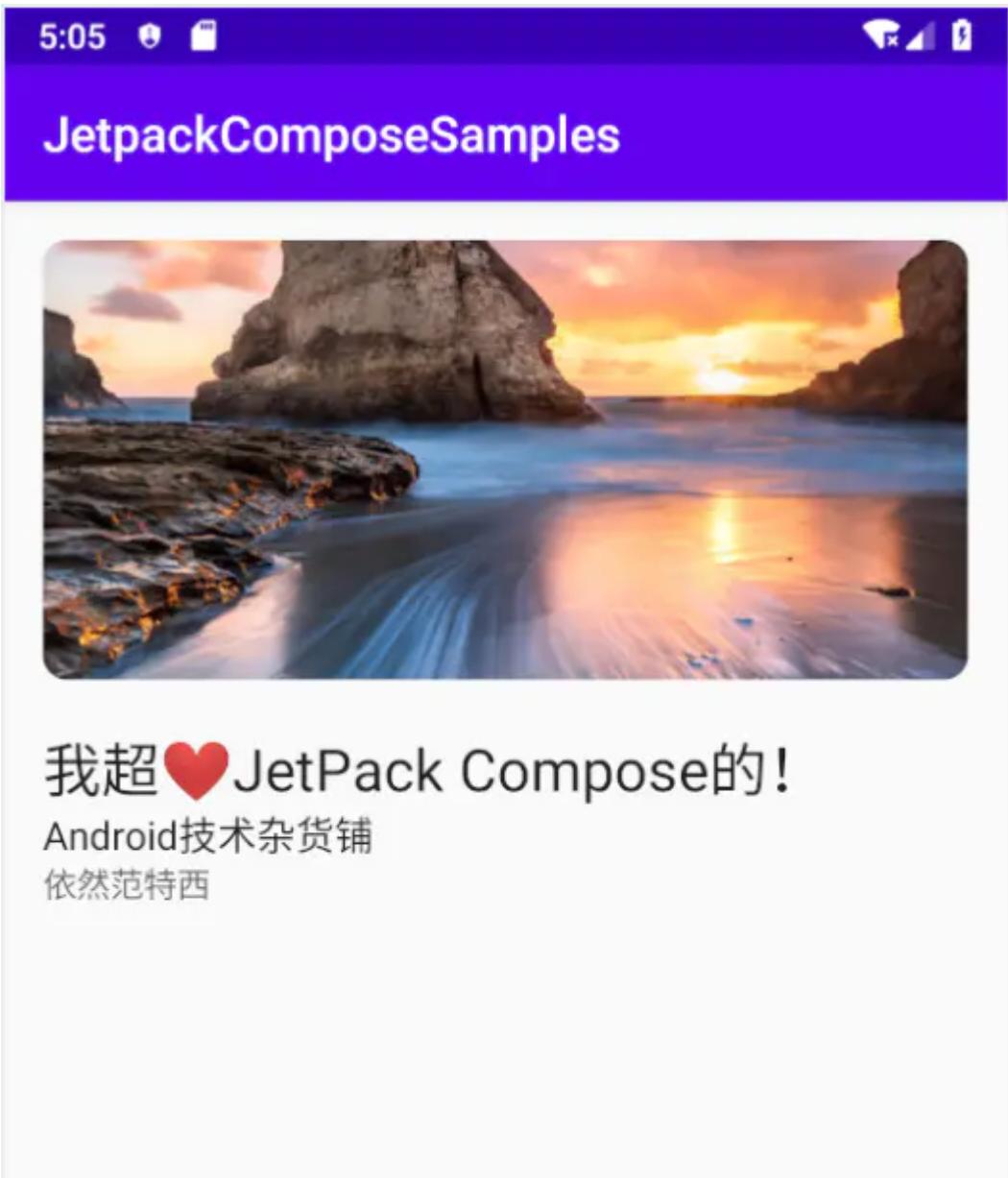


现在看看，我们的文本样式已经有变化了，标题有6中样式 `h1-h6`，其实 `HTML` 中的样式很像，内容文本有 `body1` 和 `body2` 2中样式。

Material 调色版使用了一些基本颜色，如果要强调文本，可以调整文本的 不透明度：

```
Text("我超❤️JetPack Compose的！", style = (+themeTextStyle { h5 }).withOpacity(0.87f))  
Text("Android技术杂货铺", style = (+themeTextStyle { body1 }).withOpacity(0.87f))  
Text("依然范特西", style = (+themeTextStyle { body2 }).withOpacity(0.6f))
```

复制代码

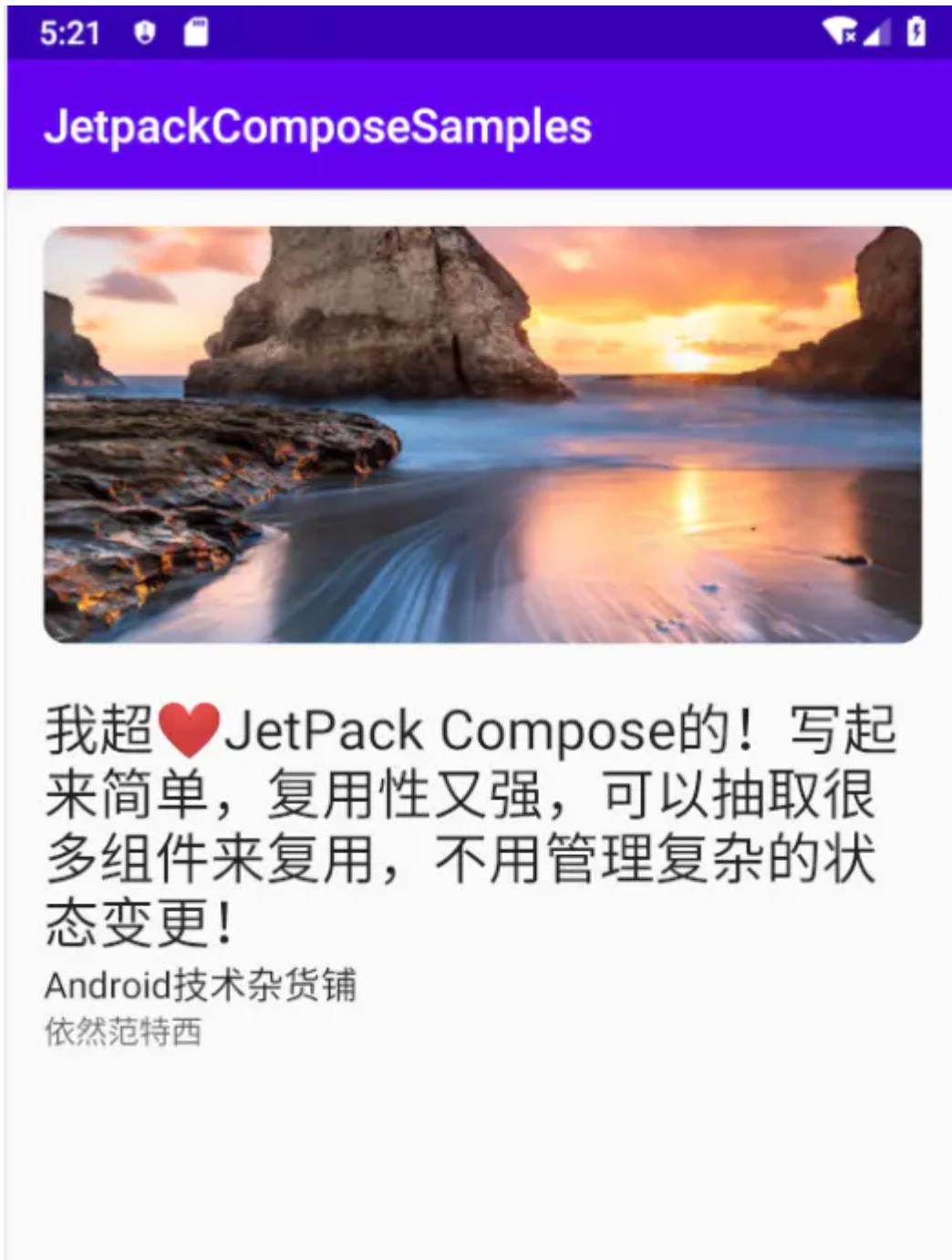


我超❤️JetPack Compose的！

Android技术杂货铺

依然范特西

有些时候，标题很长，但是我们又不想长标题换行从而影响我们的app UI，因此，我们可以设置文本的最大显示行数，超过部分就截断。



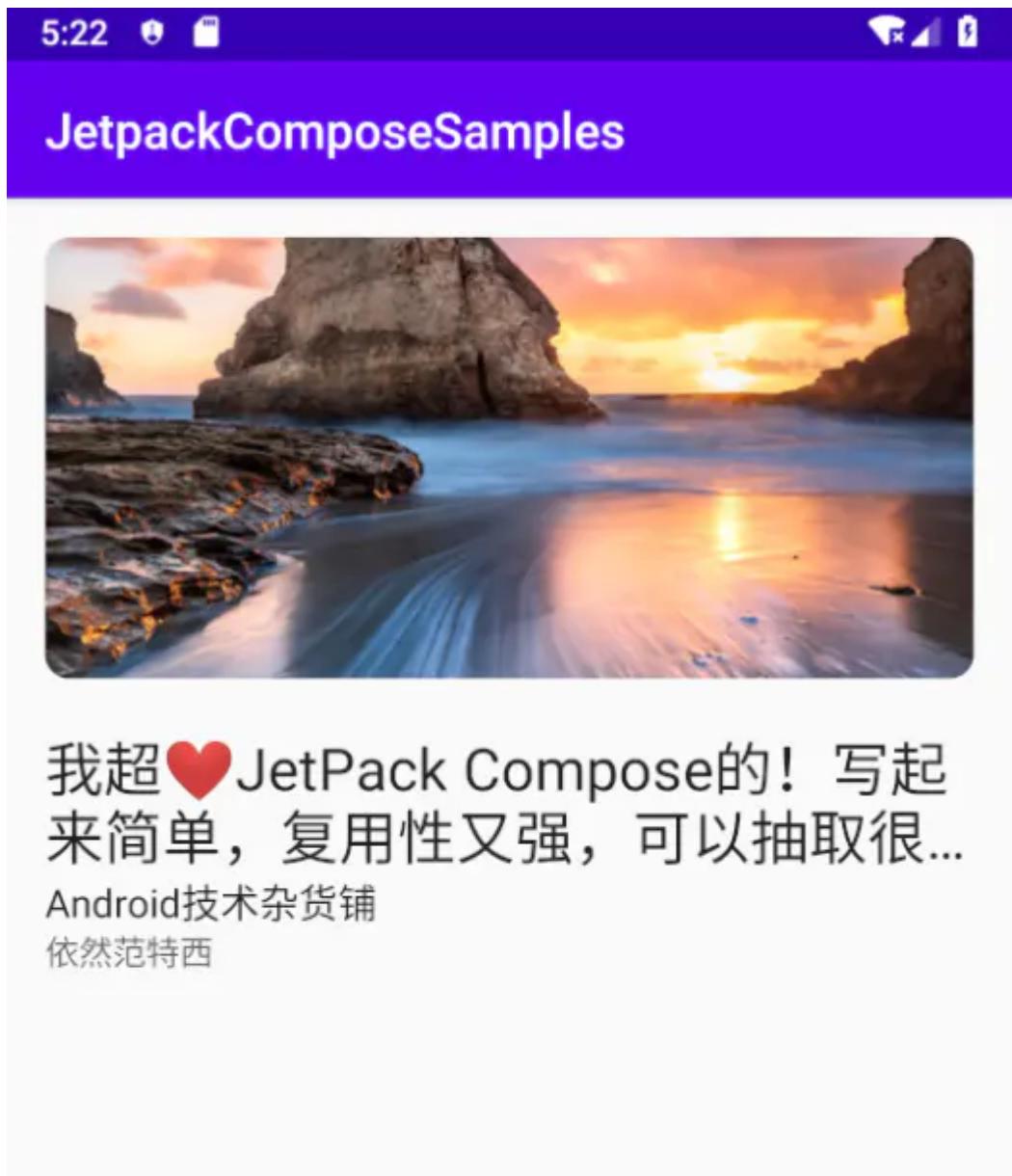
我超❤️JetPack Compose的！写起来简单，复用性又强，可以抽取很多组件来复用，不用管理复杂的状态变更！

Android技术杂货铺  
依然范特西

如本例所示，我们设置显示最大行数为 2，多余的部分截断处理：

```
Text("我超❤️JetPack Compose的！写起来简单，复用性又强，可以抽取很多组件来复用，不用管理复杂的状态变  
更！",  
    maxLines = 2, overflow = TextOverflow.Ellipsis,  
    style = (+themeTextStyle { h5 }).withOpacity(0.87f))
```

复制代码



可以看到，设置了

maxLines

和

overflow

之后，超出部分就截断处理了，不会影响到整个布局样式。

## 2.1.4 Compose 布局实时预览

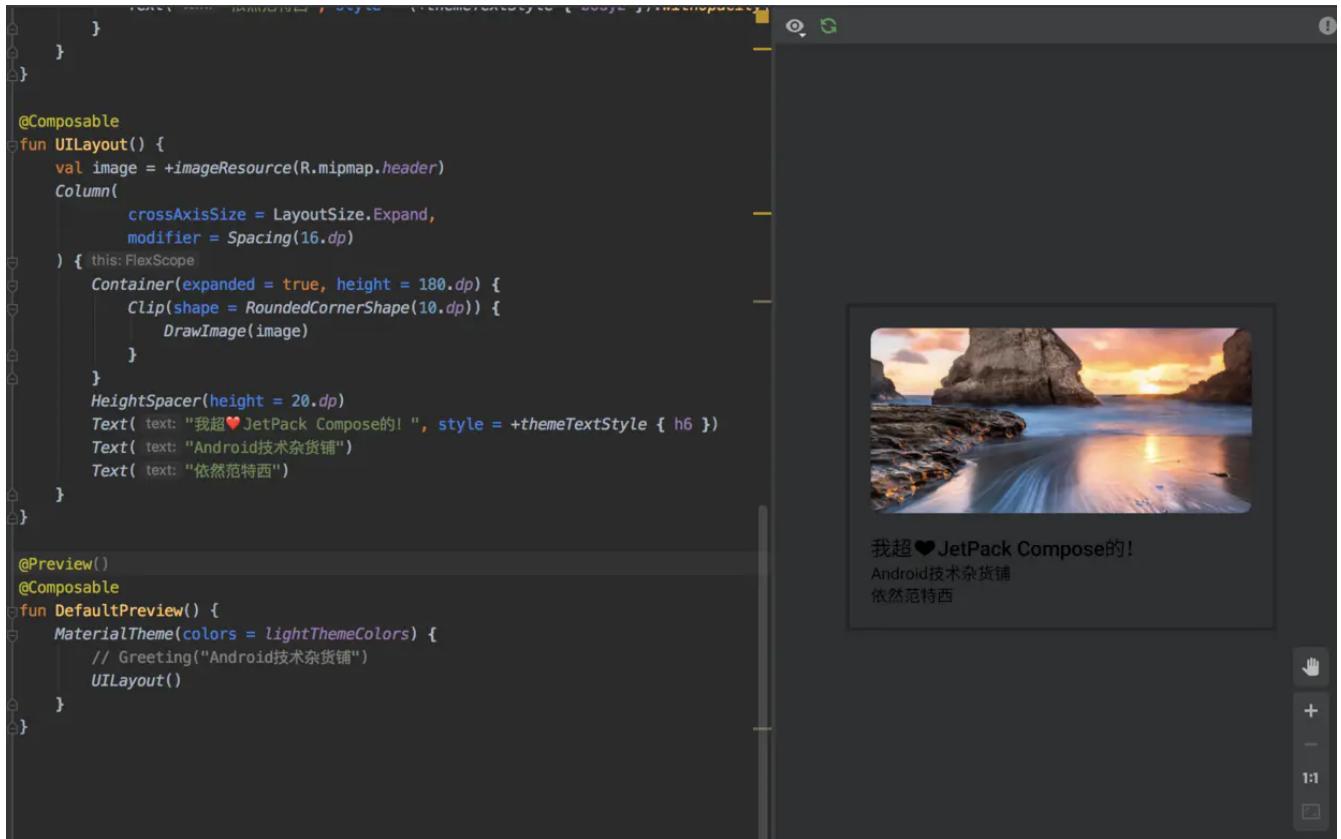
从Android Studio 4.0 开始，提供了在IDE中预览 composable 函数的功能，不用像以前那样，要先下载一个模拟器，然后将app状态模拟器上，运行app才能看到效果。

**但是有一个限制，那就是composable函数不能有参数**

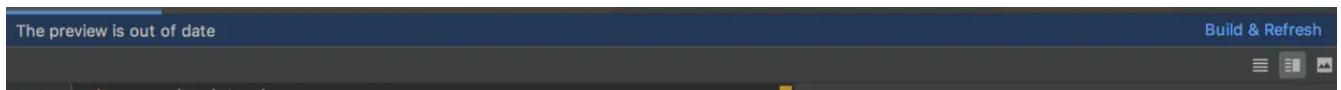
满足下面两个条件：

- 函数没有参数
- 在函数前面添加 `@Preview` 注解

预览效果图如下：



当布局改变了之后，顶部会出现一个导航条，显示预览已经过期，点击 `build&refresh` 就可以刷新预览



这真的是一个非常棒的功能，像其他声明式布局，如React、flutter 是没有这个功能的，布局了之后，要重新运行才能看到效果，虽然可以热启动，但是还是没有这个预览来得直接。

还有一个非常牛逼的地方是，compose 的预览可以同时预览多个composable函数。

效果如下：

The screenshot shows the Android Studio code editor on the left and the Preview tool window on the right. The code editor displays a portion of a file named `TutorialPreviewTemplate.kt` with several `@Preview` annotations. The Preview window shows three cards representing different preview configurations:

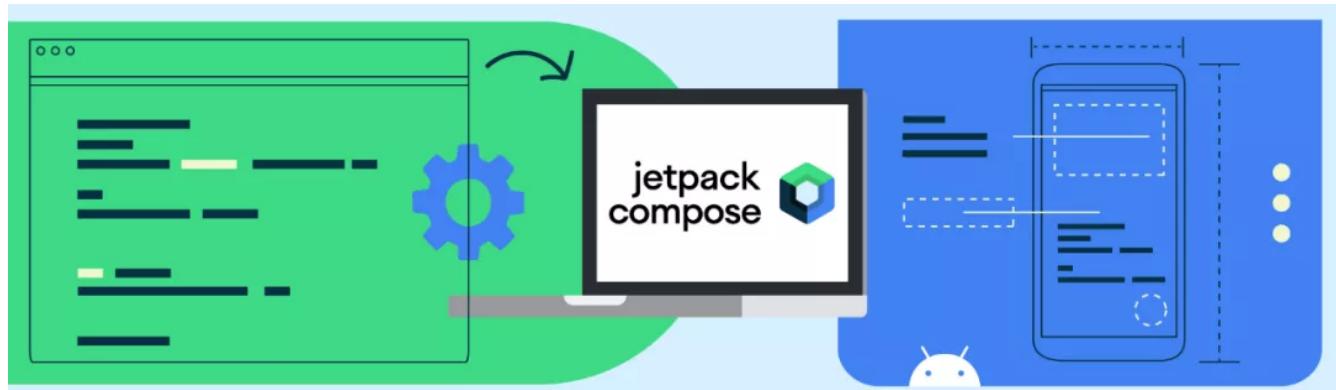
- Default colors**: Shows a light-colored interface with a blue header.
- Dark colors**: Shows a dark-themed interface with a blue header.
- Font scaling 1.5**: Shows a light-colored interface with a blue header and increased font size.

Each card includes a thumbnail, the title "Dagger in Kotlin: Gotchas and Optimizations", the author "Manuel Vivo", and a timestamp "July 30 - 3 min read".

## 2.1.5 小结

Jetpack Compose 目前还是试验版，所以肯定还存在很多问题，还不能现在将其用于商业项目中，但是这并不能妨碍我们学习和体验它，声明式 UI 框架近年来飞速发展，React 为声明式 UI 奠定了坚实基础并。Flutter 的发布将声明式 UI 的思想成功带到移动端开发领域，Apple 和 Google 分别先后发布了自己的声明式 UI 框架 SwiftUI 和 Jetpack Compose，以后，原生 UI 布局，声明式可能将会是主流。

## 2.2 深入详解 Jetpack Compose | 优化 UI 构建



人们对于 UI 开发的预期已经不同往昔。现如今，为了满足用户的需求，我们构建的应用必须包含完善的用户界面，其中必然包括动画 (animation) 和动效 (motion)，这些诉求在 UI 工具包创建之初时并不存在。为了解决如何快速而高效地创建完善的 UI 这一技术难题，我们引入了 Jetpack Compose —— 这是一个现代的 UI 工具包，能够帮助开发者们在新的趋势下取得成功。

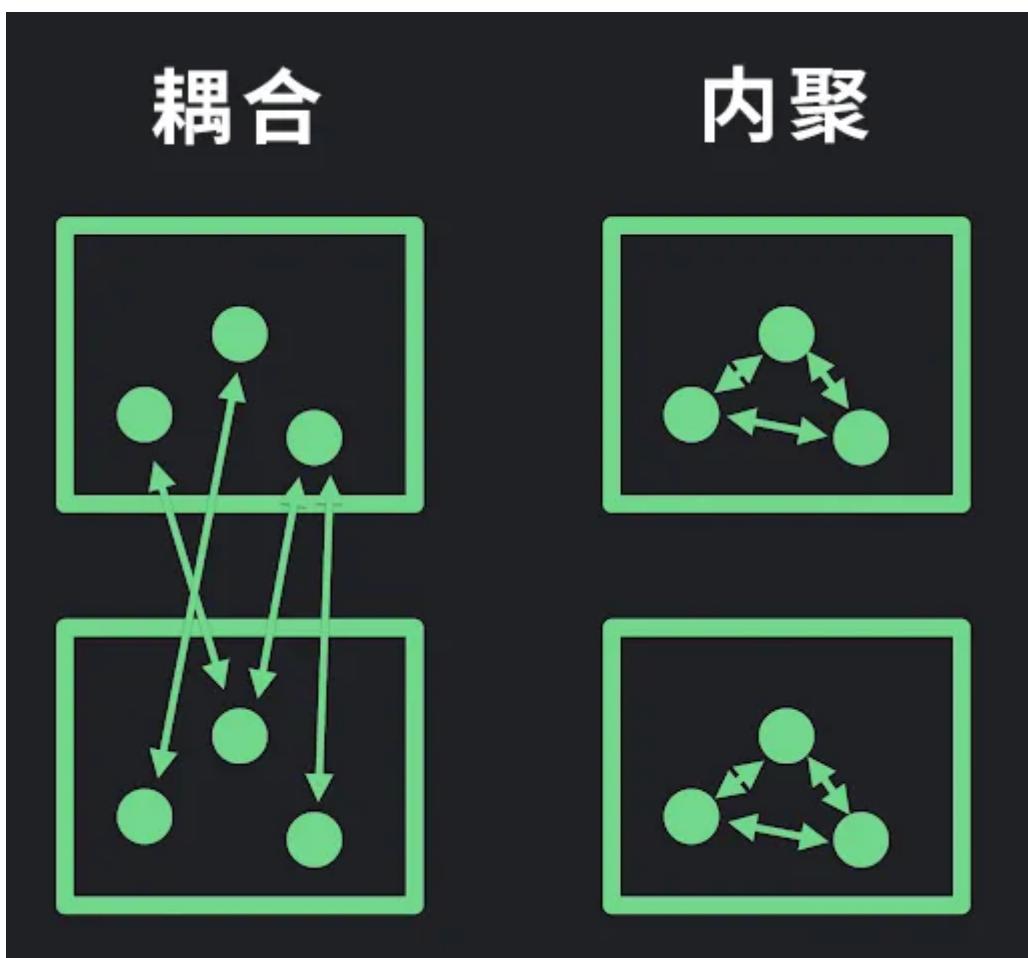
在本系列的两篇文章中，我们将阐述 Compose 的优势，并探讨它背后的工作原理。作为开篇，在本文中，我会分享 Compose 所解决的问题、一些设计决策背后的原因，以及这些决策如何帮助开发者。此外，我还会分享 Compose 的思维模型，您应如何考虑在 Compose 中编写代码，以及如何创建您自己的 API。

## 2.2.1 Compose 所解决的问题

关注点分离 (Separation of concerns, SOC) 是一个众所周知的软件设计原则，这是我们作为开发者所要学习的基础知识之一。然而，尽管其广为人知，但在实践中却常常难以把握是否应当遵循该原则。面对这样的问题，从 "耦合" 和 "内聚" 的角度去考虑这一原则可能会有所帮助。

编写代码时，我们会创建包含多个单元的模块。"耦合" 便是不同模块中单元之间的依赖关系，它反映了一个模块中的各部分是如何影响另一个模块的各个部分的。"内聚" 则表示的是一个模块中各个单元之间的关系，它指示了模块中各个单元相互组合的合理程度。

在编写可维护的软件时，我们的目标是最大程度地 **减少耦合并增加内聚**。



当我们处理**紧耦合**的模块时，对一个地方的代码改动，便意味对其他的模块作出许多其他的改动。更糟的是，耦合常常是隐式的，以至于看起来毫无关联的修改，却会造成了意料之外的错误发生。

关注点分离是尽可能的将相关的代码组织在一起，以便我们可以轻松地维护它们，并方便我们随着应用规模的增长而扩展我们的代码。

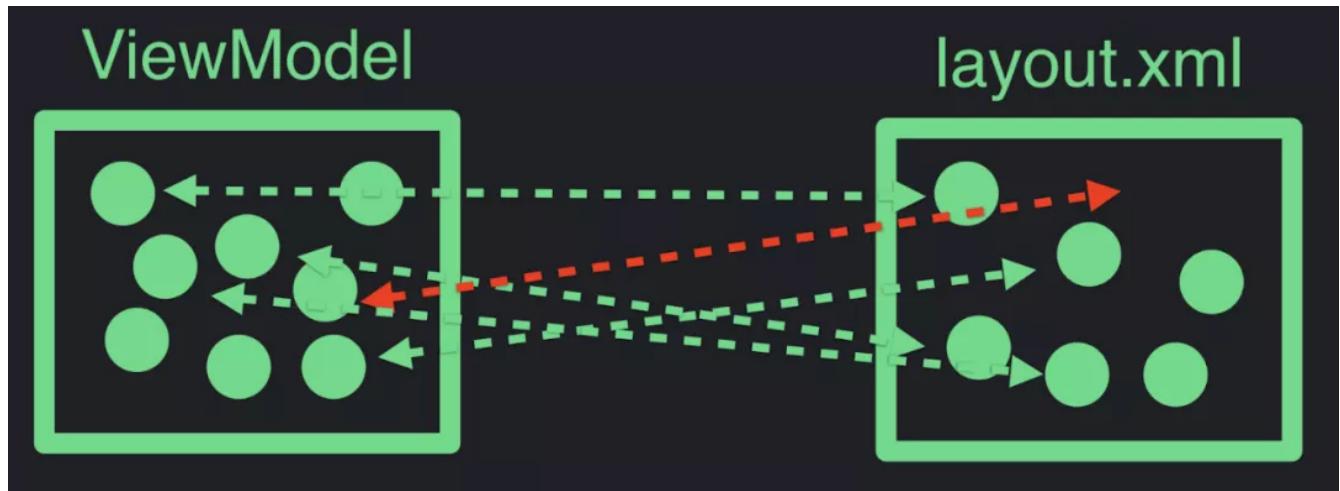
让我们在当前 Android 开发的上下文中进行更为实际的操作，并以视图模型 (view model) 和 XML 布局为例：



视图模型会向布局提供数据。事实证明，这里隐藏了很多依赖关系：视图模型与布局间存在许多耦合。一个更为熟悉的可以让您查看这一清单的方式是通过一些 API，例如 `findViewById`。使用这些 API 需要对 XML 布局的形式和内容有一定了解。

使用这些 API 需要了解 XML 布局是如何定义并与视图模型产生耦合的。由于应用规模会随着时间增长，我们还必须保证这些依赖不会过时。

大多数现代应用会动态展示 UI，并且会在执行过程中不断演变。结果导致应用不仅要验证布局 XML 是否静态地满足了这些依赖关系，而且还需要保证在应用的生命周期内满足这些依赖。如果一个元素在运行时离开了视图层级，一些依赖关系可能会被破坏，并导致诸如 `NullPointerExceptions` 一类的问题。

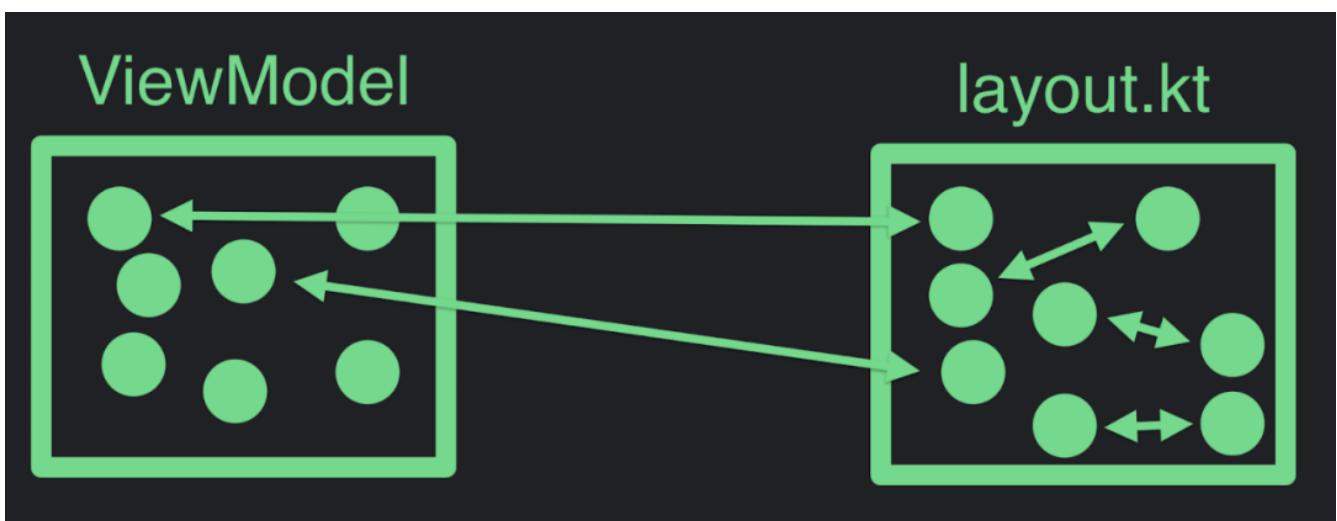


通常，视图模型会使用像 Kotlin 这样的编程语言进行定义，而布局则使用 XML。由于这两种语言的差异，使得它们之间存在一条强制的分隔线。然而即使存在这种情况，视图模型与布局 XML 还是可以关联得十分紧密。换句话说，它们二者紧密耦合。

这就引出了一个问题：如果我们开始用相同的语言定义布局与 UI 结构会怎样？如果我们选用 Kotlin 来做这件事会怎样？



由于我们可以使用相同的语言，一些以往隐式的依赖关系可能会变得更加明显。我们也可以重构代码并将其移动至那些可以使它们减少耦合和增加内聚的位置。



现在，您可能会以为这是建议您将逻辑与 UI 混合起来。不过现实的情况是，无论您如何组织架构，您的应用中都将出现与 UI 相关联的逻辑。框架本身并不会改变这一点。

不过框架可以为您提供一些工具，从而帮您更加简单地实现关注点分离：这一工具便是 Composable 函数，长久以来您在代码的其他地方实现关注点分离所使用的方法，您在进行这类重构以及编写简洁、可靠、可维护的代码时所获得的技巧，都可以应用在 Composable 函数上。

## 2.2.2 Composable 函数剖析

这是一个 Composable 函数的示例：

```
@Composable
fun App(appData: AppData) {
    val derivedData = compute(appData)
    Header()
    if (appData.isOwner) {
        EditButton()
    }
    Body {
        for (item in derivedData.items) {
            Item(item)
        }
    }
}
```

在示例中，函数从 AppData 类接收数据作为参数。理想情况下，这一数据是不可变数据，而且 Composable 函数也不会改变：Composable 函数应当成为这一数据的转换函数。这样一来，我们便可以使用任何 Kotlin 代码来获取这一数据，并利用它来描述的我们的层级结构，例如 Header() 与 Body() 调用。

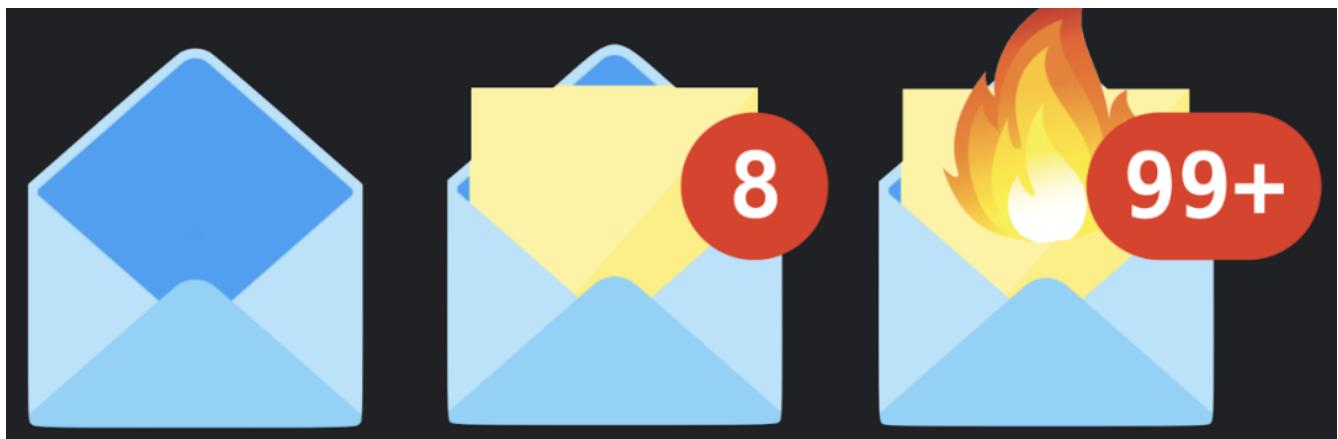
这意味着我们调用了其他 Composable 函数，并且这些调用代表了我们层次结构中的 UI。我们可以使用 Kotlin 中语言级别的原语来动态执行各种操作。我们也可以使用 if 语句与 for 循环来实现控制流，来处理更为复杂的 UI 逻辑。

Composable 函数通常利用 Kotlin 的尾随 lambda 语法，所以 Body() 是一个含有 Composable lambda 参数的 Composable 函数。这种关系意味着层级或结构，所以这里 Body() 可以包含多个元素组成的多个元素组成的集合。

## 2.2.3 声明式 UI

"声明式" 是一个流行词，但也是一个很重要的字眼。当我们谈论声明式编程时，我们谈论的是与命令式相反的编程方式。让我们来看一个例子：

假设有一个带有未读消息图标的电子邮件应用。如果没有消息，应用会绘制一个空信封；如果有一些消息，我们会在信封中绘制一些纸张；而如果有 100 条消息，我们就把图标绘制成好像在着火的样子……



使用命令式接口，我们可能会写出一个下面这样的更新数

```
fun updateCount(count: Int) {
    if (count > 0 && !hasBadge()) {
        addBadge()
    } else if (count == 0 && hasBadge()) {
        removeBadge()
    }
}
```

```
    }
    if (count > 99 && !hasFire()) {
        addFire()
        setBadgeText("99+")
    } else if (count <= 99 && hasFire()) {
        removeFire()
    }
    if (count > 0 && !hasPaper()) {
        addPaper()
    } else if (count == 0 && hasPaper()) {
        removePaper()
    }
    if (count <= 99) {
        setBadgeText("$count")
    }
}
```

在这段代码中，我们接收新的数量并且必须搞清楚如何更新当前的 UI 来反映对应的状态。尽管是一个相对简单的示例，这里仍然出现了许多极端情况，而且这里的逻辑也不简单。

作为替代，使用声明式接口编写这一逻辑则会看起来像下面这样：

```
@Composable
fun BadgedEnvelope(count: Int) {
    Envelope(fire=count > 99, paper=count > 0) {
        if (count > 0) {
            Badge(text="$count")
        }
    }
}
```

这里我们定义：

- 当数量大于 99 时，显示火焰；
- 当数量大于 0 时，显示纸张；
- 当数量大于 0 时，绘制数量气泡。

这便是声明式 API 的含义。我们编写代码来按我们的想法描述 UI，而不是如何转换到对应的状态。这里的关键是，编写像这样的声明式代码时，您不需要关注您的 UI 在先前是什么状态，而只需要指定当前应当处于的状态。框架控制着如何从一个状态转到其他状态，所以我们不再需要考虑它。

## 2.2.4 组合 vs 继承

在软件开发领域，Composition (组合) 指的是多个简单的代码单元如何结合到一起，从而构成更为复杂的代码单元。在面向对象编程模型中，最常见的组合形式之一便是基于类的继承。在 Jetpack Compose 的世界中，由于我们使用函数替代了类型，因此实现组合的方法颇为不同，但相比于继承也拥有许多优点，让我们来看一个例子：

假设我们有一个视图，并且我们想要添加一个输入。在继承模型中，我们的代码可能会像下面这样：

```
class Input : View() { /* ... */ }
class ValidatedInput : Input() { /* ... */ }
class DateInput : ValidatedInput() { /* ... */ }
class DateRangeInput : ??? { /* ... */ }
```

View 是基类，ValidatedInput 使用了 Input 的子类。为了验证日期，DateInput 使用了 ValidatedInput 的子类。但是接下来挑战来了：我们要创建一个日期范围的输入，这意味着需要验证两个日期——开始和结束日期。您可以继承 DateInput，但是您无法执行两次，这便是继承的限制：我们只能继承自一个父类。

在 Compose 中，这个问题变得很简单。假设我们从一个基础的 Input Composable 函数开始：

```
@Composable
fun <T> Input(value: T, onChange: (T) -> Unit) {
    /* ... */
}
```

当我们创建 ValidatedInput 时，只需要在方法体中调用 Input 即可。我们随后可以对其进行装饰以实现验证逻辑：

```
@Composable
fun ValidatedInput(value: T, onChange: (T) -> Unit, isValid: Boolean) {
    InputDecoration(color = if(isValid) blue else red) {
        Input(value, onChange)
    }
}
```

接下来，对于 DateInput，我们可以直接调用 ValidatedInput：

```
@Composable
fun DateInput(value: DateTime, onChange: (DateTime) -> Unit) {
    ValidatedInput(
        value,
        onChange = { ... onChange(...) },
        isValid = isValidDate(value)
    )
}
```

现在，当我们实现日期范围输入时，这里不再会有任何挑战：只需要调用两次即可。示例如下：

```
@Composable
fun DateRangeInput(value: DateRange, onChange: (DateRange) -> Unit) {
    DateInput(value = value.start, ...)
    DateInput(value = value.end, ...)
}
```

在 Compose 的组合模型中，我们不再有单个父类的限制，这样一来便解决了我们在继承模型中所遭遇的问题。

另一种类型的组合问题是装饰类型的抽象。为了能够说明这一情况，请您考虑接下来的继承示例：

```
class FancyBox : View() { /* ... */ }
class Story : View() { /* ... */ }
class EditForm : FormView() { /* ... */ }
class FancyStory : ??? { /* ... */ }
class FancyEditForm : ??? { /* ... */ }
```

FancyBox 是一个用于装饰其他视图的视图，本例中将用来装饰 Story 和 EditForm。我们想要编写 FancyStory 与 FancyEditForm，但是如何做到呢？我们要继承自 FancyBox 还是 Story？又因为继承链中单个父类的限制，使这里变得十分含糊。

相反，Compose 可以很好地处理这一问题：

```
@Composable
fun FancyBox(children: @Composable () -> Unit) {
    Box(fancy) { children() }
}

@Composable fun Story(...) { /* ... */ }
@Composable fun EditForm(...) { /* ... */ }
@Composable fun FancyStory(...) {
    FancyBox { Story(...) }
}

@Composable fun FancyEditForm(...) {
    FancyBox { EditForm(...) }
}
```

我们将 Composable lambda 作为子级，使得我们可以定义一些可以包裹其他函数的函数。这样一来，当我们要创建 FancyStory 时，可以在 FancyBox 的子级中调用 Story，并且可以使用 FancyEditForm 进行同样的操作。这便是 Compose 的组合模型。

## 2.2.5 封装

Compose 做的很好的另一个方面是 "封装"。这是您在创建公共 Composable 函数 API 时需要考虑的问题：公共的 Composable API 只是一组其接收的参数而已，所以 Compose 无法控制它们。另一方面，Composable 函数可以管理和创建状态，然后将该状态及它接收到的任何数据作为参数传递给其他的 Composable 函数。

现在，由于它正管理该状态，如果您想要改变状态，您可以启用您的子级 Composable 函数通过回调通知当前的改变。

## 2.2.6 重组

"重组" 指的是任何 Composable 函数在任何时候都可以被重新调用。如果您有一个庞大的 Composable 层级结构，当您的层级中的某一部分发生改变时，您不会希望重新计算整个层级结构。所以 Composable 函数是可重启动 (restartable) 的，您可以利用这一特性来实现一些强大的功能。

举个例子，这里有一个 Bind 函数，里面是一些 Android 开发的常见代码：

```
fun bind(liveMsgs: LiveData<MessageData>) {
    liveMsgs.observe(this) { msgs ->
        updateBody(msgs)
    }
}
```

我们有一个 LiveData，并且希望视图可以订阅它。为此，我们调用 observe 方法并传入一个 LifecycleOwner，并在接下来传入 lambda。lambda 会在每次 LiveData 更新被调用，并且发生这种情况时，我们会想要更新视图。

使用 Compose，我们可以反转这种关系。

```
@Composable
fun Messages(liveMsgs: LiveData<MessageData>) {
    val msgs by liveMsgs.observeAsState()
    for (msg in msgs) {
        Message(msg)
    }
}
```

这里有一个相似的 Composable 函数—— Messages。它接收了 LiveData 作为参数并调用了 Compose 的 observeAsState 方法。observeAsState 方法会把 LiveData 映射为 State，这意味着您可以在函数体的范围使用其值。State 实例订阅了 LiveData 实例，这意味着 State 会在 LiveData 发生改变的任何地方更新，也意味着，无论在何处读取 State 实例，包裹它的、已被读取的 Composable 函数将会自动订阅这些改变。结果就是，这里不再需要指定 LifecycleOwner 或者更新回调，Composable 可以隐式地实现这两者的功能。

## 2.2.6 小结

Compose 提供了一种现代的方法来定义您的 UI，这使您可以有效地实现关注点分离。由于 Composable 函数与普通 Kotlin 函数很相似，因此您使用 Compose 编写和重构 UI 所使用的工具与您进行 Android 开发的知识储备和所使用的工具将会无缝衔接。

# 2.3 深入详解 Jetpack Compose | 实现原理

本节是 Compose 深入详解 系列的第二篇文章。在[上一节](#)中，我已经阐述了 Compose 的优点、Compose 所解决的问题、一些设计决策背后的原因，以及这些内容是如何帮助开发者的。此外，我还讨论了 Compose 的思维模型、您应如何考虑使用 Compose 编写代码，以及如何创建您自己的 API。

在本文中，我将着眼于 Compose 背后的工作原理。但在开始之前，我想要强调的是，**使用 Compose 并不一定需要您理解它是如何实现的**。接下来的内容纯粹是为了满足您的求知欲而撰写的。

## 2.3.1 @Composable 注解意味着什么？

如果您已经了解过 Compose，您大概已经在一些代码示例中看到过 @Composable 注解。这里有件很重要的事情需要注意—— Compose 并不是一个注解处理器。Compose 在 Kotlin 编译器的类型检测与代码生成阶段依赖 Kotlin 编译器插件工作，所以无需注解处理器即可使用 Compose。

这一注解更接近于一个语言关键字。作为类比，可以参考 Kotlin 的 [suspend 关键字](#)：

```
// 函数声明
suspend fun MyFun() { ... }

// Lambda 声明
val myLambda = suspend { ... }

// 函数类型
fun MyFun(myParam: suspend () -> Unit) { ... }
```

Kotlin 的 [suspend 关键字](#)适用于处理函数类型: 您可以将函数、lambda 或者函数类型声明为 suspend。Compose 与其工作方式相同: 它可以改变函数类型。

```
// 函数声明
@Composable fun MyFun() { ... }

// Lambda 声明
val myLambda = @Composable { ... }

// 函数类型
fun MyFun(myParam: @Composable () -> Unit) { ... }
```

这里的重点是，**当您使用 @Composable 注解一个函数类型时，会导致它类型的改变**: 未被注解的相同函数类型与注解后的类型互不兼容。同样的，挂起(suspend) 函数需要调用上下文作为参数，这意味着您只能在其他挂起函数中调用挂起函数:

```
fun Example(a: () -> Unit, b: suspend () -> Unit) {
    a() // 允许
    b() // 不允许
}

suspend
fun Example(a: () -> Unit, b: suspend () -> Unit) {
    a() // 允许
    b() // 允许
}
```

Composable 的工作方式与其相同。这是因为我们需要一个贯穿所有的上下文调用对象。

```
fun Example(a: () -> Unit, b: @Composable () -> Unit) {
    a() // 允许
    b() // 不允许
}

@Composable
fun Example(a: () -> Unit, b: @Composable () -> Unit) {
    a() // 允许
    b() // 允许
}
```

## 2.3.2 执行模式

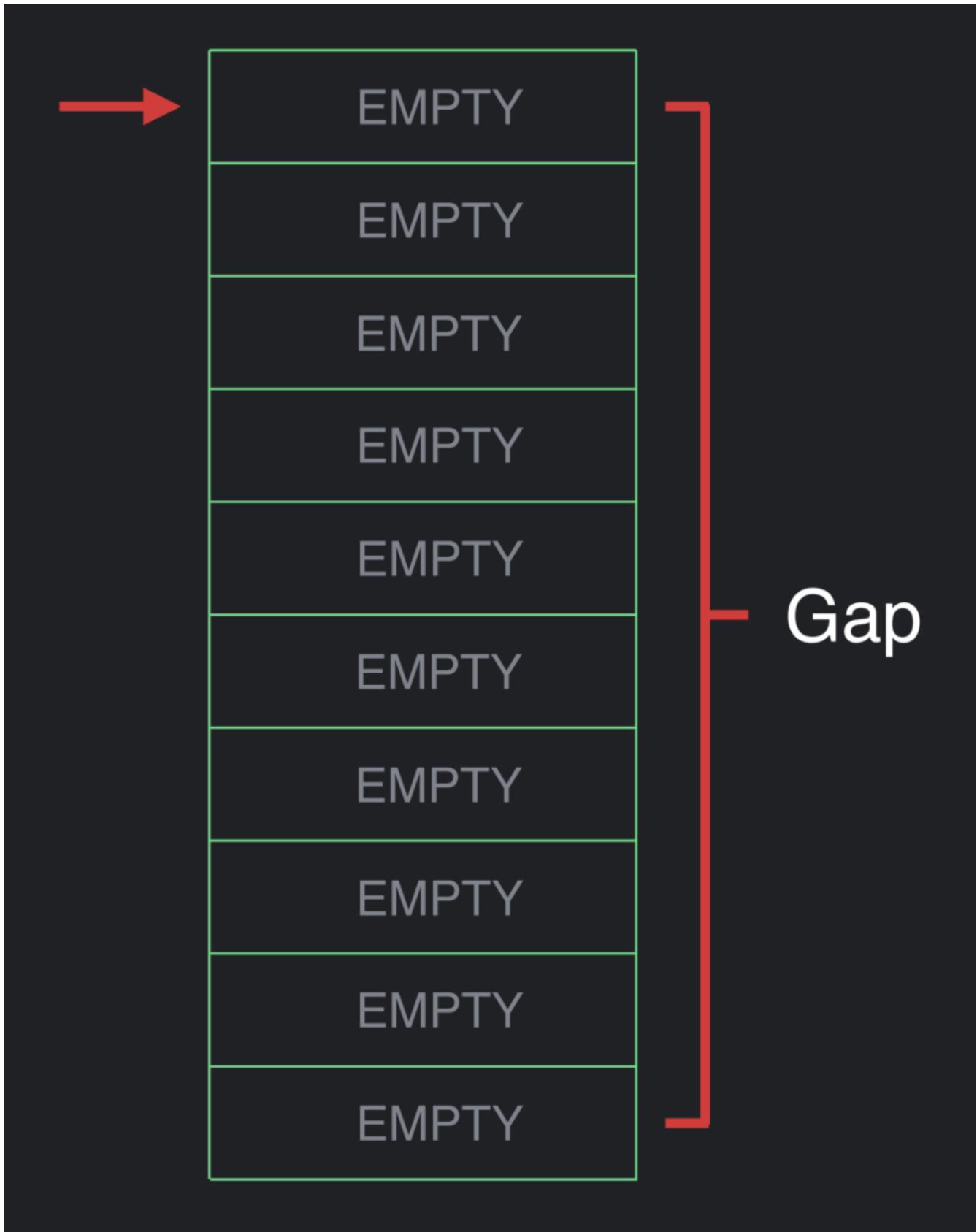
所以，我们正在传递的调用上下文究竟是什么？还有，我们为什么需要传递它？

我们将其称之为 "Composer"。Composer 的实现包含了一个与 Gap Buffer (间隙缓冲区) 密切相关的数据结构，这一数据结构通常应用于文本编辑器。

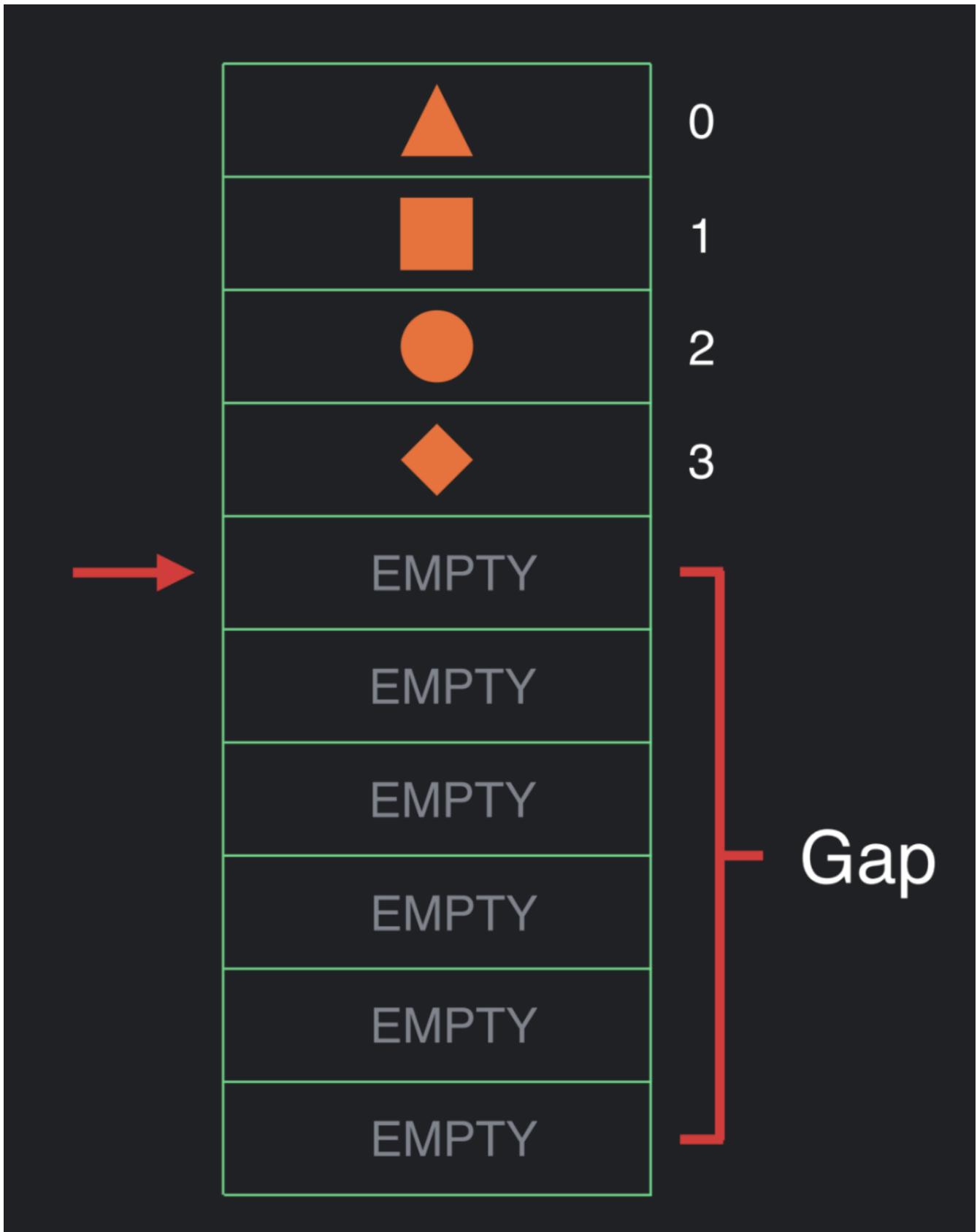
- Gap Buffer

[https://en.wikipedia.org/wiki/Gap\\_buffer](https://en.wikipedia.org/wiki/Gap_buffer)

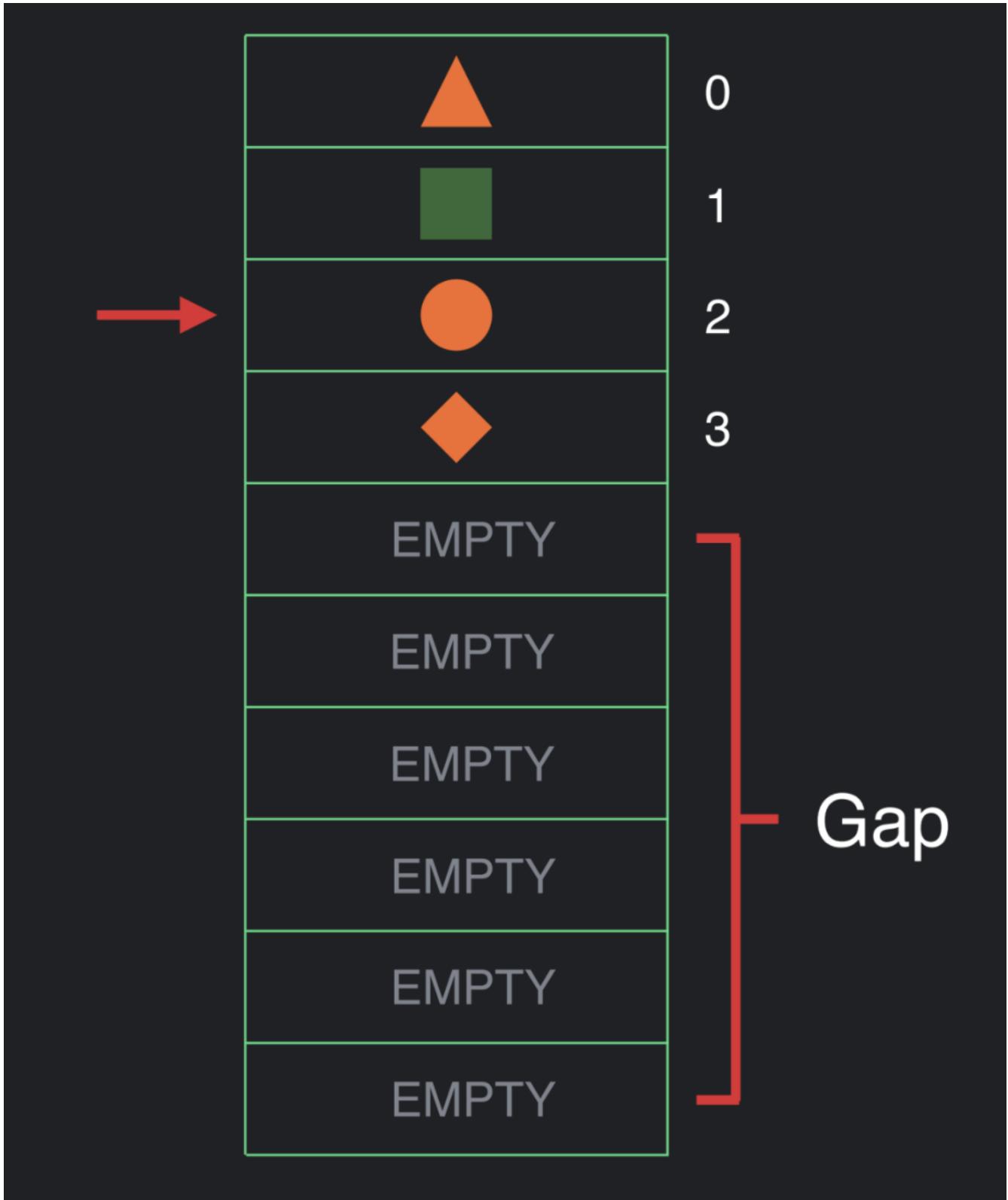
间隙缓冲区是一个含有当前索引或游标的集合，它在内存中使用扁平数组 (flat array) 实现。这一扁平数组比它代表的数据集合要大，而那些没有使用的空间就被称为间隙。



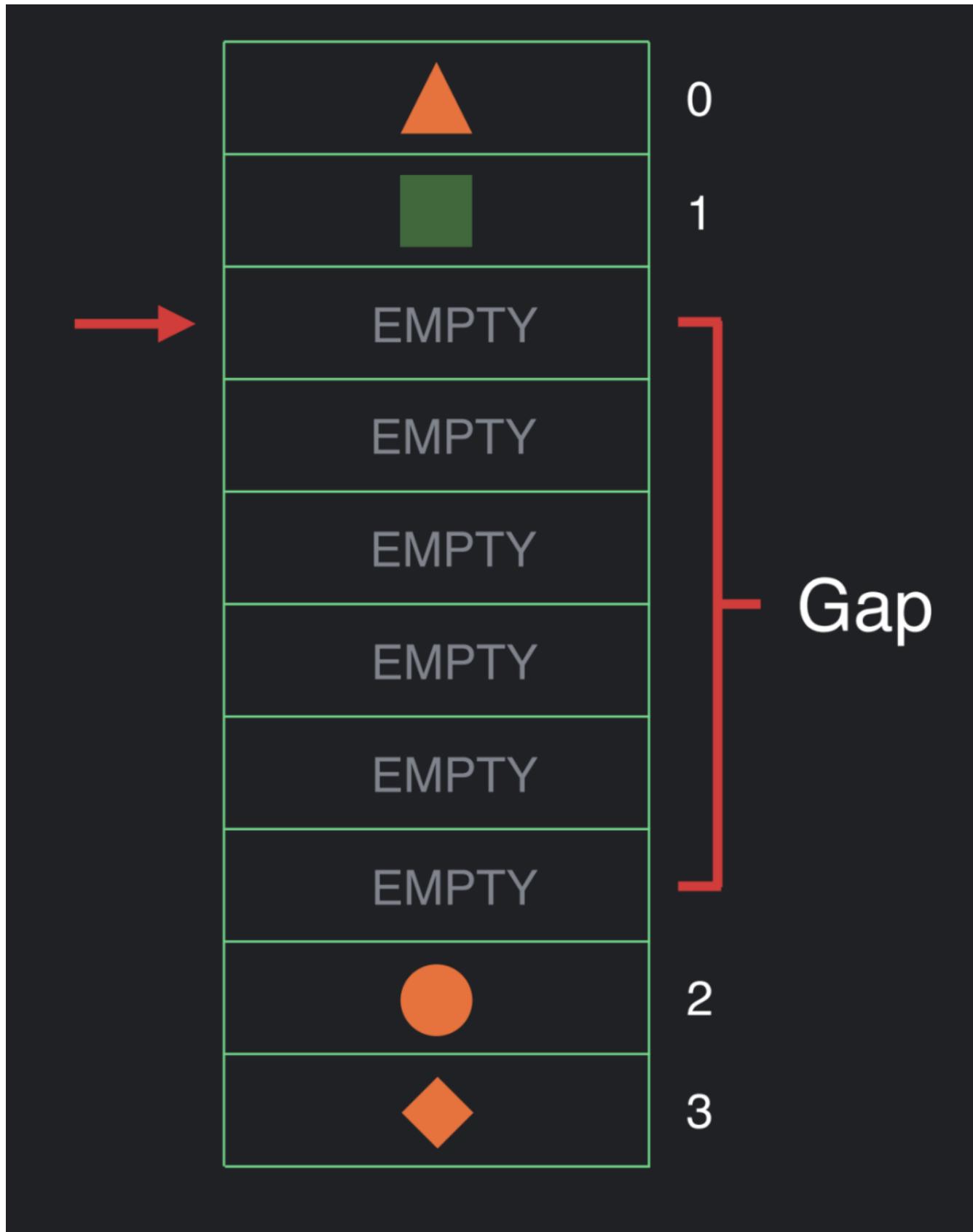
一个正在执行的 Composable 的层级结构可以使用这个数据结构，而且我们可以在其中插入一些东西。



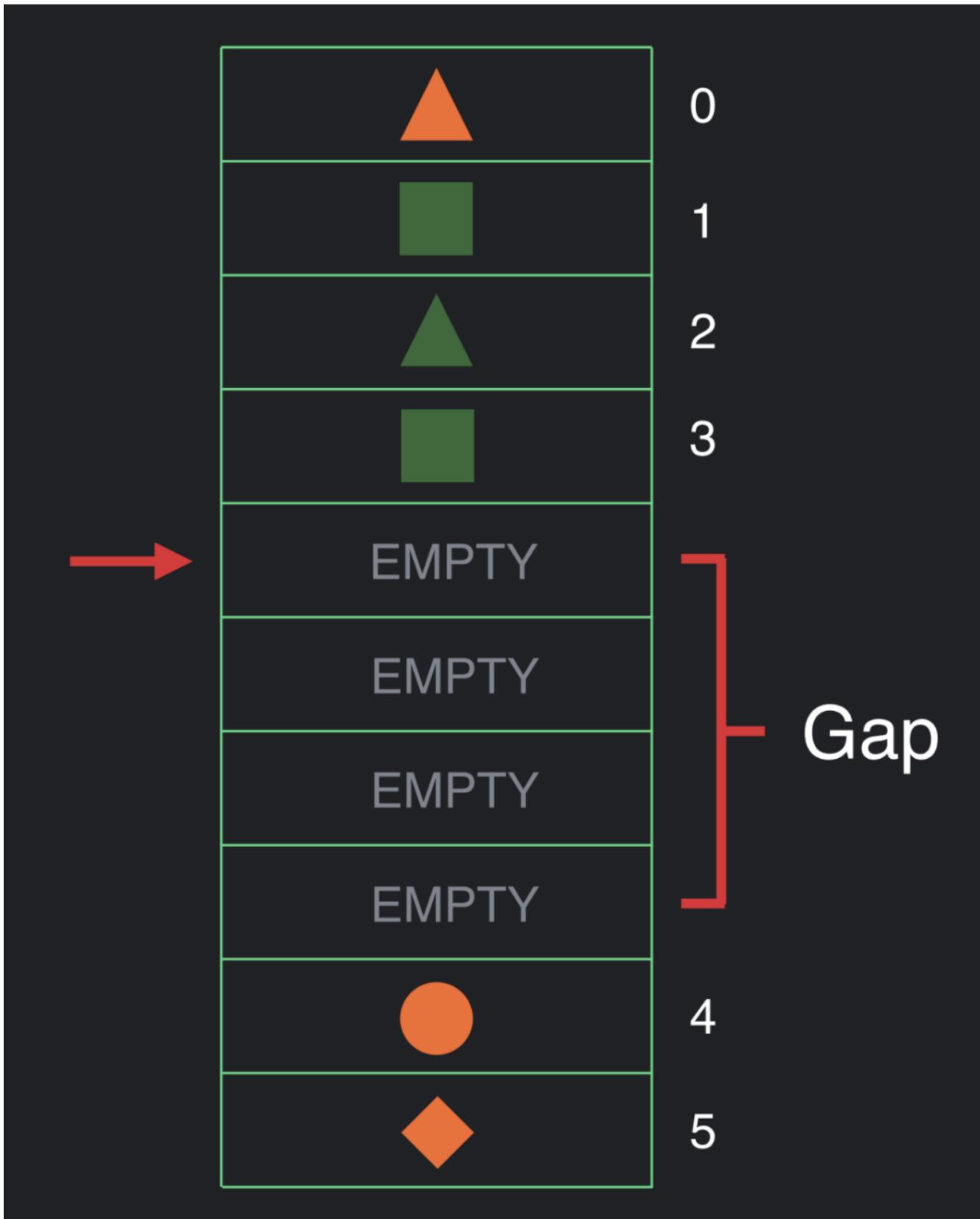
让我们假设已经完成了层级结构的执行。在某个时候，我们会重新组合一些东西。所以我们将游标重置回数组的顶部并再次遍历执行。在我们执行时，可以选择仅仅查看数据并且什么都不做，或是更新数据的值。



我们也许会决定改变 UI 的结构，并且希望进行一次插入操作。在这个时候，我们会把间隙移动至当前位置。



现在，我们可以进行插入操作了。



在了解此数据结构时，很重要的一点是除了移动间隙，它的所有其他操作包括获取 (get)、移动 (move)、插入 (insert)、删除 (delete) 都是常数时间操作。移动间隙的时间复杂度为  $O(n)$ 。我们选择这一数据结构是因为 UI 的结构通常不会频繁地改变。当我们处理动态 UI 时，它们的值虽然发生了改变，却通常不会频繁地改变结构。当它们确实需要改变结构时，则很可能需要做出大块的改动，此时进行  $O(n)$  的间隙移动操作便是一个很合理的权衡。

让我们来看一个计数器示例：

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Button(
        text="Count: $count",
        onPress={ count += 1 }
    )
}
```

这是我们编写的代码，不过我们要看的是编译器做了什么。

当编译器看到 Composable 注解时，它会在函数体中插入额外的参数和调用。

首先，编译器会添加一个 composer.start 方法的调用，并向其传递一个编译时生成的整数 key。

```
fun Counter($composer: Composer) {
    $composer.start(123)
    var count by remember { mutableStateOf(0) }
    Button(
        text="Count: $count",
        onPress={ count += 1 }
    )
    $composer.end()
}
```

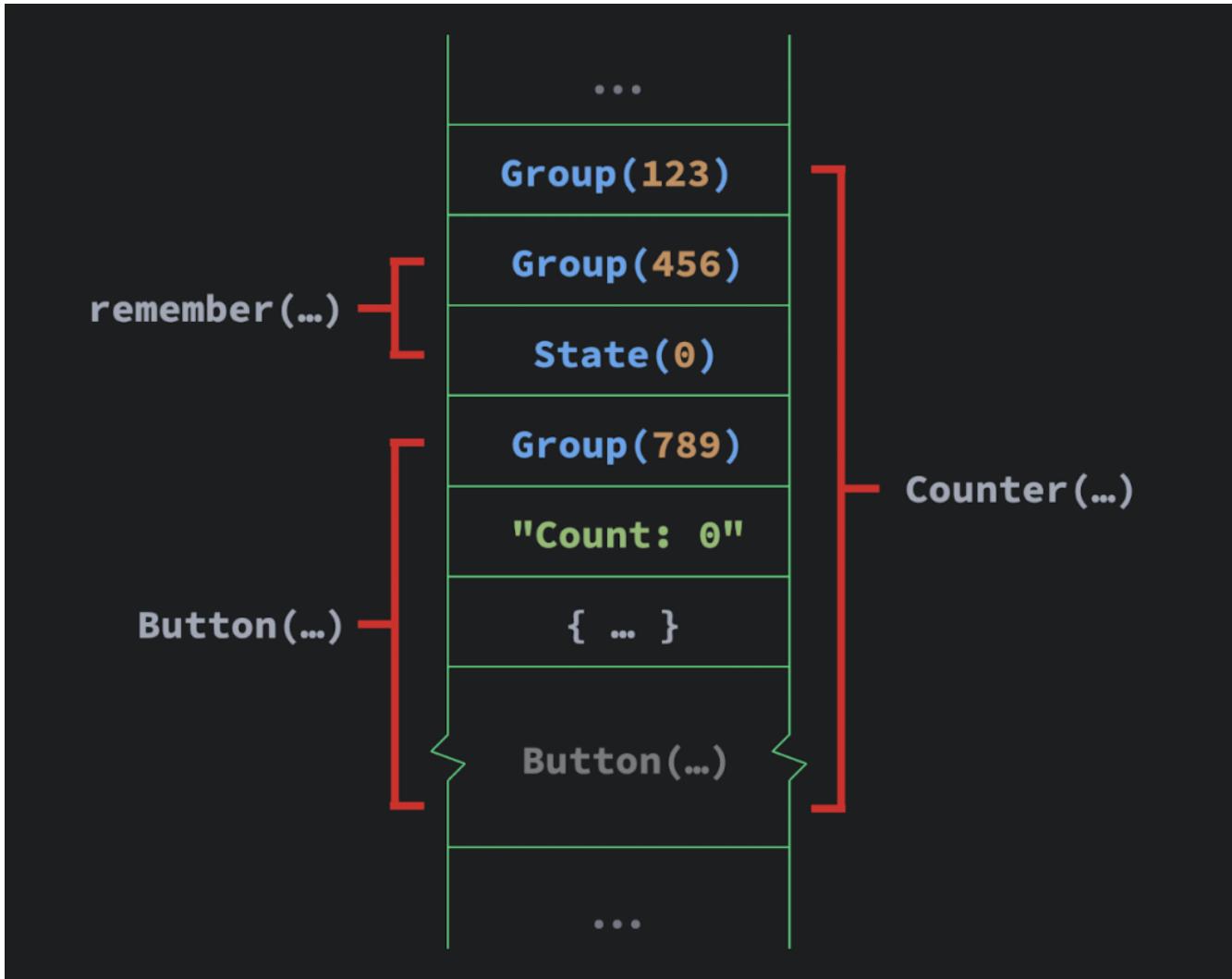
编译器也会将 composer 对象传递到函数体里的所有 composable 调用中。

```
fun Counter($composer: Composer) {
    $composer.start(123)
    var count by remember($composer) { mutableStateOf(0) }
    Button(
        $composer,
        text="Count: $count",
        onPress={ count += 1 },
    )
    $composer.end()
}
```

当此 composer 执行时，它会进行以下操作：

- Composer.start 被调用并存储了一个组对象 (group object)
- remember 插入了一个组对象
- mutableStateOf 的值被返回，而 state 实例会被存储起来
- Button 基于它的每个参数存储了一个分组

最后，当我们到达 composer.end 时：



数据结构现在已经持有了来自组合的所有对象，整个树的节点也已经按照深度优先遍历的执行顺序排列。

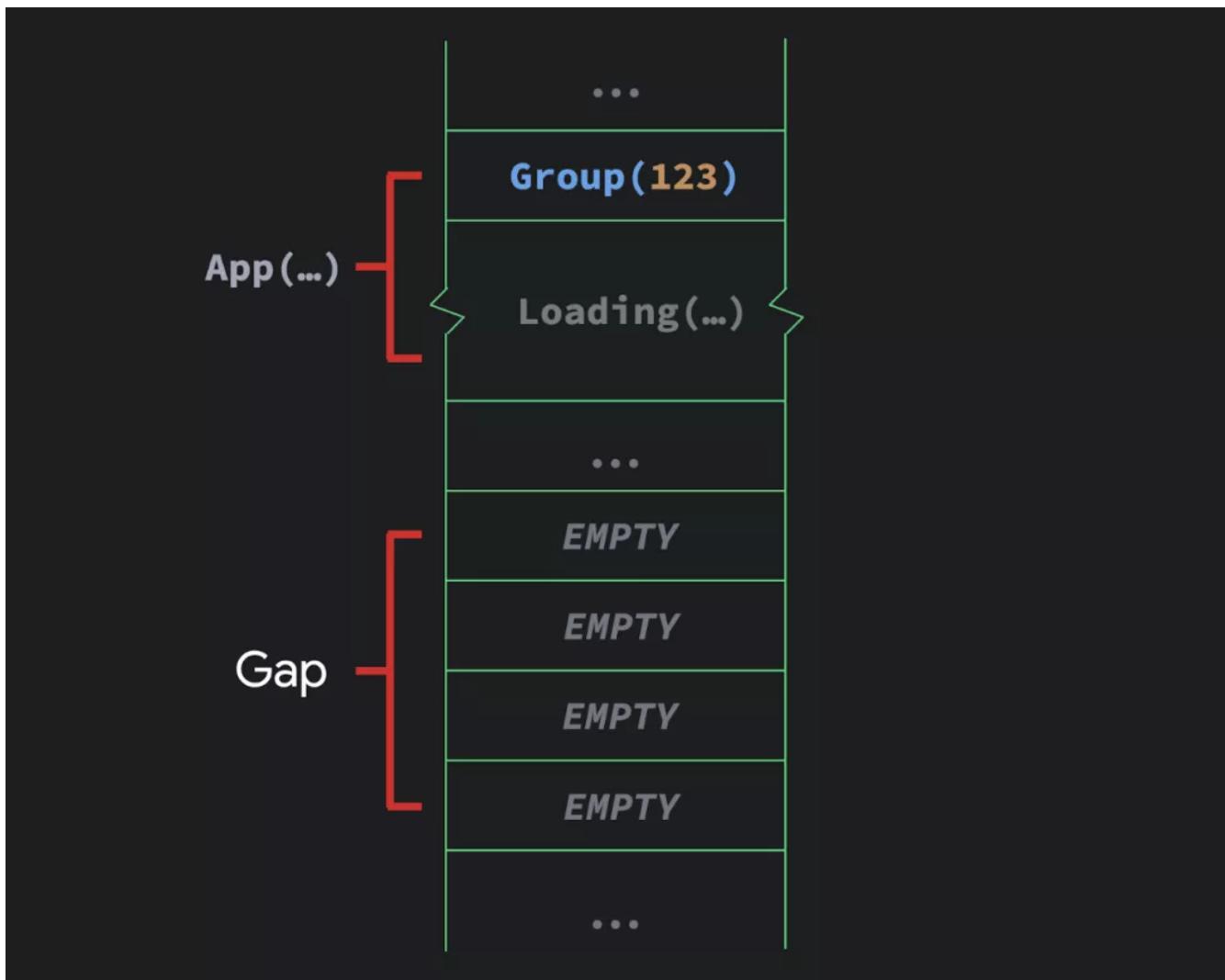
现在，所有这些组对象已经占据了很多的空间，它们为什么要占据这些空间呢？这些组对象是用来管理动态 UI 可能发生的移动和插入的。编译器知道哪些代码会改变 UI 的结构，所以它可以有条件地插入这些分组。大部分情况下，编译器不需要它们，所以它不会向插槽表 (slot table) 中插入过多的分组。为了说明这一点，请您查看以下条件逻辑：

```
@Composable fun App() {
    val result = getData()
    if (result == null) {
        Loading(...)
    } else {
        Header(result)
        Body(result)
    }
}
```

在这个 Composable 函数中，`getData` 函数返回了一些结果并在某个情况下绘制了一个 `Loading` composable 函数；而在另一个情况下，它绘制了 `Header` 和 `Body` 函数。编译器会在 `if` 语句的每个分支间插入分隔关键字。

```
fun App($composer: Composer) {  
    val result = getData()  
    if (result == null) {  
        $composer.start(123)  
        Loading(...)  
        $composer.end()  
    } else {  
        $composer.start(456)  
        Header(result)  
        Body(result)  
        $composer.end()  
    }  
}
```

让我们假设这段代码第一次执行的结果是 null。这会使一个分组插入空隙并运行载入界面。

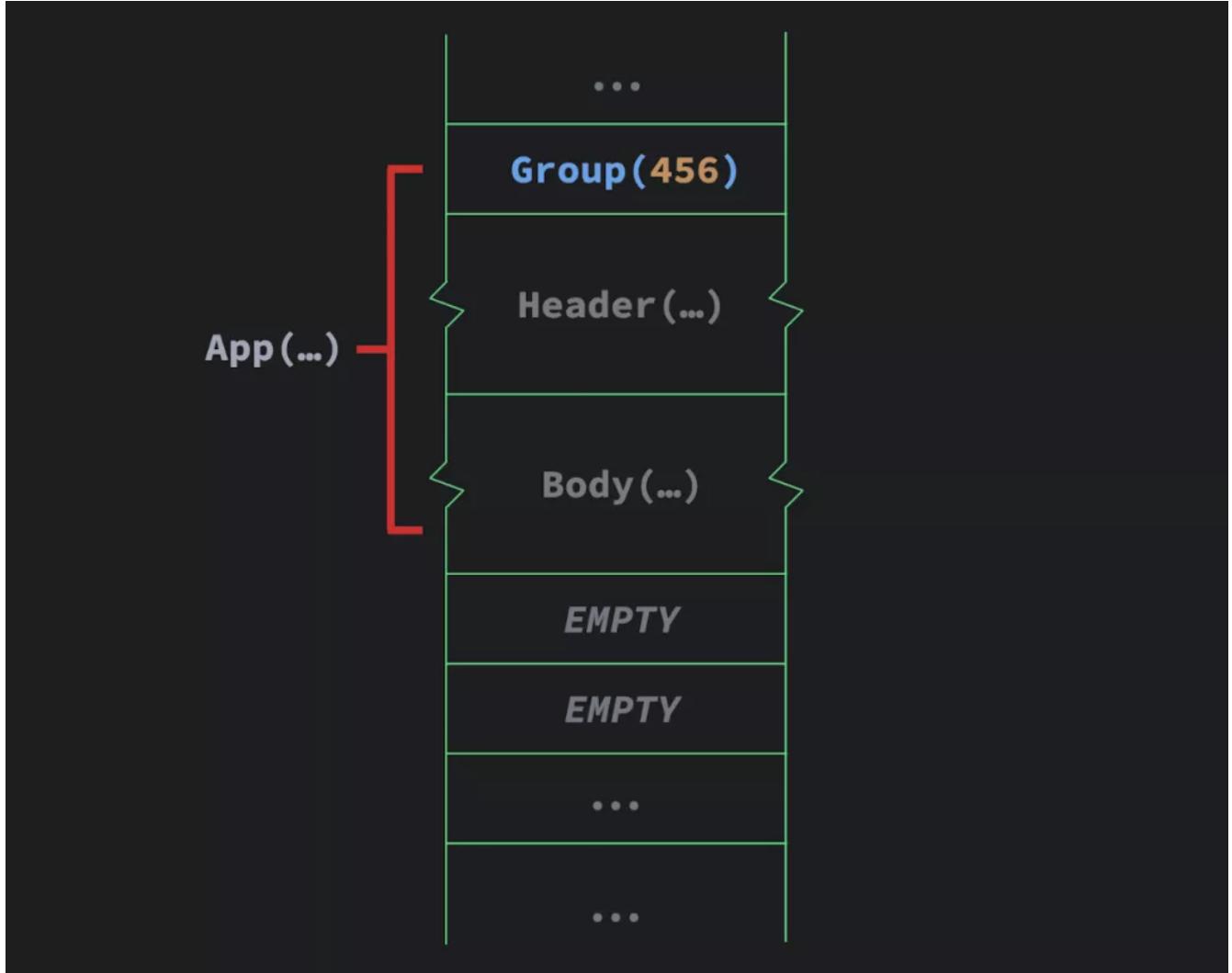


函数第二次执行时，让我们假设它的结果不再是 null，这样一来第二个分支就会执行。这里便是它变得有趣的地方。

对 `composer.start` 的调用有一个 key 为 456 的分组。编译器会看到插槽表中 key 为 123 分组与之并不匹配，所以此时它知道 UI 的结构发生了改变。

于是编译器将缝隙移动至当前游标位置并使其在以前 UI 的位置进行扩展，从而有效地消除了旧的 UI。

此时，代码已经会像一般的情况一样执行，而且新的 UI —— header 和 body —— 也已被插入其中。



在这种情况下，if 语句的开销为插槽表中的单个条目。通过插入单个组，我们可以在 UI 中任意实现控制流，同时启用编译器对 UI 的管理，使其可以在处理 UI 时利用这种类缓存的数据结构。

这是一种我们称之为 Positional Memoization 的概念，同时也是自创建伊始便贯穿整个 Compose 的概念。

### 2.3.3 Positional Memoization (位置记忆化)

通常，我们所说的全局记忆化，指的是编译器基于函数的输入缓存了其结果。下面是一个正在执行计算的函数，我们用它作为位置记忆化的示例：

```
@Composable
fun App(items: List<String>, query: String) {
    val results = items.filter { it.matches(query) }
    // ...
}
```

该函数接收一个字符串列表与一个要查找的字符串，并在接下来对列表进行了过滤计算。我们可以将该计算包装至对 remember 函数的调用中——remember 函数知道如何利用插槽列表。remember 函数会查看列表中的字符串，同时也会存储列表并在插槽表中对其进行查询。过滤计算会在之后运行，并且 remember 函数会在结果传回之前对其进行存储。

函数第二次执行时，remember 函数会查看新传入的值并将其与旧值进行对比，如果所有的值都没有发生改变，过滤操作就会在跳过的同时将之前的结果返回。这便是位置记忆化。

有趣的是，这一操作的开销十分低廉：编译器必须存储一个先前的调用。这一计算可以发生在您的 UI 的各个地方，由于您是基于位置对其进行存储，因此只会为该位置进行存储。

下面是 remember 的函数签名，它可以接收任意多的输入与一个 calculation 函数。

```
@Composable fun <T> remember(vararg inputs: Any?, calculation: () -> T): T
```

不过，这里没有输入时会产生一个有趣的退化情况。我们可以故意误用这一 API，比如记忆一个像 Math.random 这样不输出稳定结果的计算：

```
@Composable fun App() {  
    val x = remember { Math.random() }  
    // ...  
}
```

使用全局记忆化来进行这一操作将不会有任何意义，但如果换做使用位置记忆化，此操作将最终呈现出一种新的语义。每当我们在 Composable 层级中使用 App 函数时，都将会返回一个新的 Math.random 值。不过，每次 Composable 被重新组合时，它将会返回相同的 Math.random 值。这一特性使得持久化成为可能，而持久化又使得状态成为可能。

## 2.3.4 存储参数

下面，让我们用 Google Composable 函数来说明 Composable 是如何存储函数的参数的。这个函数接收一个数字作为参数，并且通过调用 Address Composable 函数来绘制地址。

```
@Composable fun Google(number: Int) {  
    Address(  
        number=number,  
        street="Amphitheatre Pkwy",  
        city="Mountain View",  
        state="CA"  
        zip="94043"  
    )  
}  
  
@Composable fun Address(  
    number: Int,  
    street: String,  
    city: String,  
    state: String,  
    zip: String  
) {  
    Text("$number $street")  
    Text(city)  
    Text(", ")  
    Text(state)  
    Text(" ")  
    Text(zip)
```

```
}
```

Compose 将 Composable 函数的参数存储在插槽表中。在本例中，我们可以看到一些冗余: Address 调用中添加的 "Mountain View" 与 "CA" 会在下面的文本调用被再次存储，所以这些字符串会被存储两次。

我们可以在编译器级为 Composable 函数添加 static 参数来消除这种冗余。

```
fun Google(
    $composer: Composer,
    $static: Int,
    number: Int
) {
    Address(
        $composer,
        0b11110 or ($static and 0b1),
        number=number,
        street="Amphitheatre Pkwy",
        city="Mountain View",
        state="CA"
        zip="94043"
    )
}
```

本例中，static 参数是一个用于指示运行时是否知道参数不会改变的位字段。如果已知一个参数不会改变，则无需存储该参数。所以这一 Google 函数示例中，编译器传递了一个位字段来表示所有参数都不会发生改变。

接下来，在 Address 函数中，编译器可以执行相同的操作并将参数传递给 text。

```
fun Address(
    $composer: Composer,
    $static: Int,
    number: Int, street: String,
    city: String, state: String, zip: String
) {
    Text($composer, ($static and 0b11) and ((($static and 0b10) shr 1), "$number $street"))
    Text($composer, ($static and 0b100) shr 2, city)
    Text($composer, 0b1, ", ")
    Text($composer, ($static and 0b1000) shr 3, state)
    Text($composer, 0b1, " ")
    Text($composer, ($static and 0b10000) shr 4, zip)
}
```

这些位操作逻辑难以阅读且令人困惑，但我们也没有必要理解它们：编译器擅长于此，而人类则不然。

在 Google 函数的实例中，我们看到这里不仅有冗余，而且有一些常量。事实证明，我们也不需要存储它们。这样一来，number 参数便可以决定整个层级，它也是唯一一个需要编译器进行存储的值。

有赖于此，我们可以更进一步，生成可以理解 number 是唯一一个会发生改变的值的代码。接下来这段代码可以在 number 没有发生改变时直接跳过整个函数体，而我们也可以指导 Composer 将当前索引移动至函数已经执行到的位置。

```
fun Google(
    $composer: Composer,
    number: Int
) {
    if (number == $composer.next()) {
        Address(
            $composer,
            number=number,
            street="Amphitheatre Pkwy",
            city="Mountain View",
            state="CA"
            zip="94043"
        )
    } else {
        $composer.skip()
    }
}
```

Composer 知道快进至需要恢复的位置的距离。

### 2.3.5 重组

为了解释重组是如何工作的，我们需要回到计数器的例子：

```
fun Counter($composer: Composer) {
    $composer.start(123)
    var count = remember($composer) { mutableStateOf(0) }
    Button(
        $composer,
        text="Count: ${count.value}",
        onPress={ count.value += 1 },
    )
    $composer.end()
}
```

编译器为 Counter 函数生成的代码含有一个 composer.start 和一个 compose.end。每当 Counter 执行时，运行时就会理解：当它调用 count.value 时，它会读取一个 appmodel 实例的属性。在运行时，每当我们调用 compose.end，我们都可以选择返回一个值。

```
$composer.end()?.updateScope { nextComposer ->
    Counter(nextComposer)
}
```

接下来，我们可以在该返回值上使用 lambda 来调用 updateScope 方法，从而告诉运行时在有需要时如何重启当前的 Composable。这一方法等同于 LiveData 接收的 lambda 参数。在这里使用问号的原因——可空的原因——是因为如果我们在执行 Counter 的过程中不读取任何模型对象，则没有理由告诉运行时如何更新它，因为我们知道它永远不会更新。

### 2.3.6 小结

您一定要记得的重要一点是，这些细节中的绝大部分只是实现细节。与标准的 Kotlin 函数相比，Composable 函数具有不同的行为和功能。有时候理解如何实现十分有用，但是未来 Composable 函数的行为与功能不会改变，而实现则有可能发生变化。

同样的，Compose 编译器在某些状况下可以生成更为高效的代码。随着时间流逝，我们也期待优化这些改进。

## 第三章 Jetpack Compose 项目实战演练（附Demo）

### 3.1 Jetpack Compose应用1

#### 3.1.1 开始前的准备

说一千道一万，不如自己动手试一下，使用之前还有条件

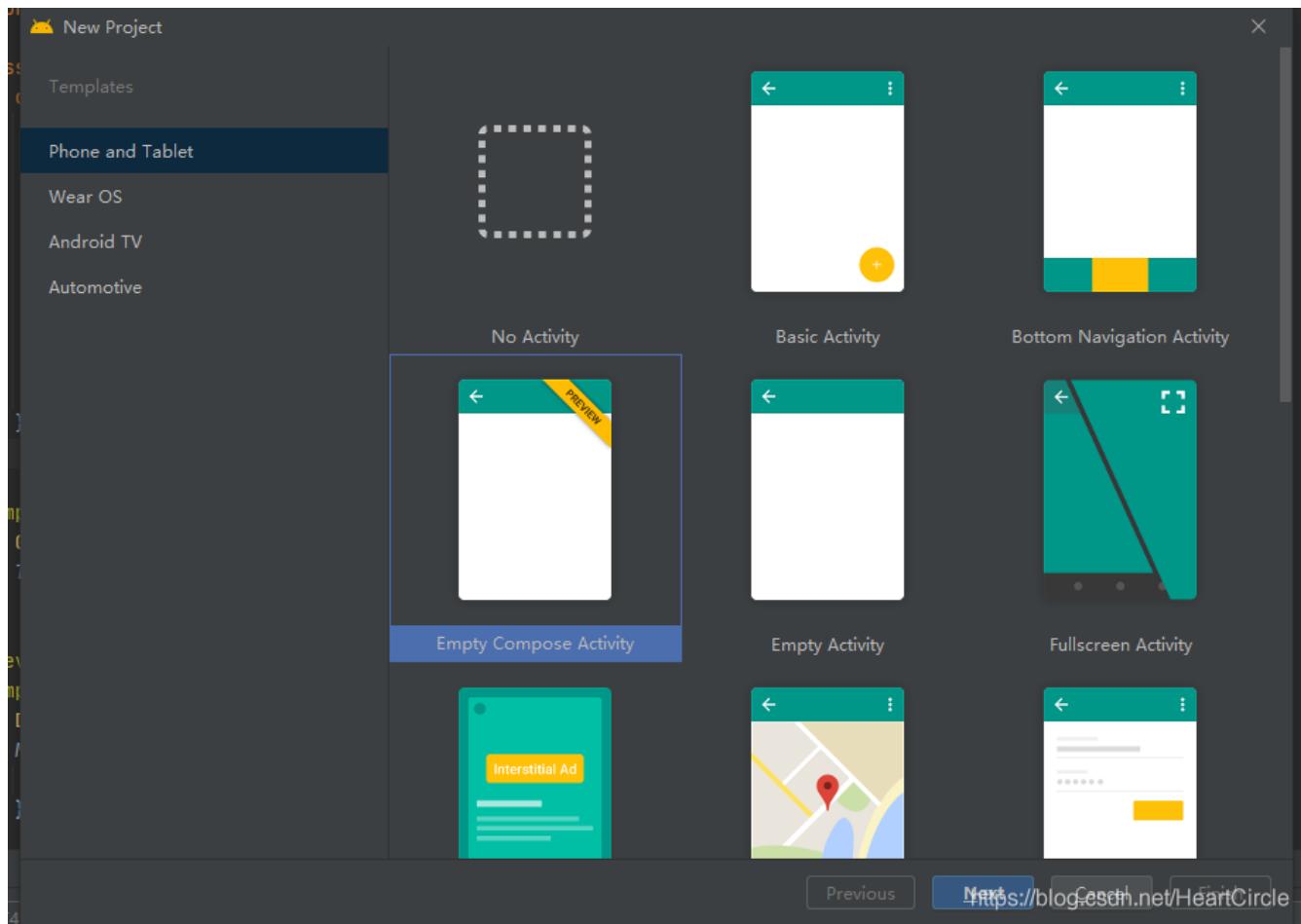
3.1.1.1 需要重新配置一下新的 AndroidStudio，必须使用新版本 金丝雀版本的AndroidStudio，必须区分开来才行

3.1.1.2 必须使用kotlin语言，不支持Java语言

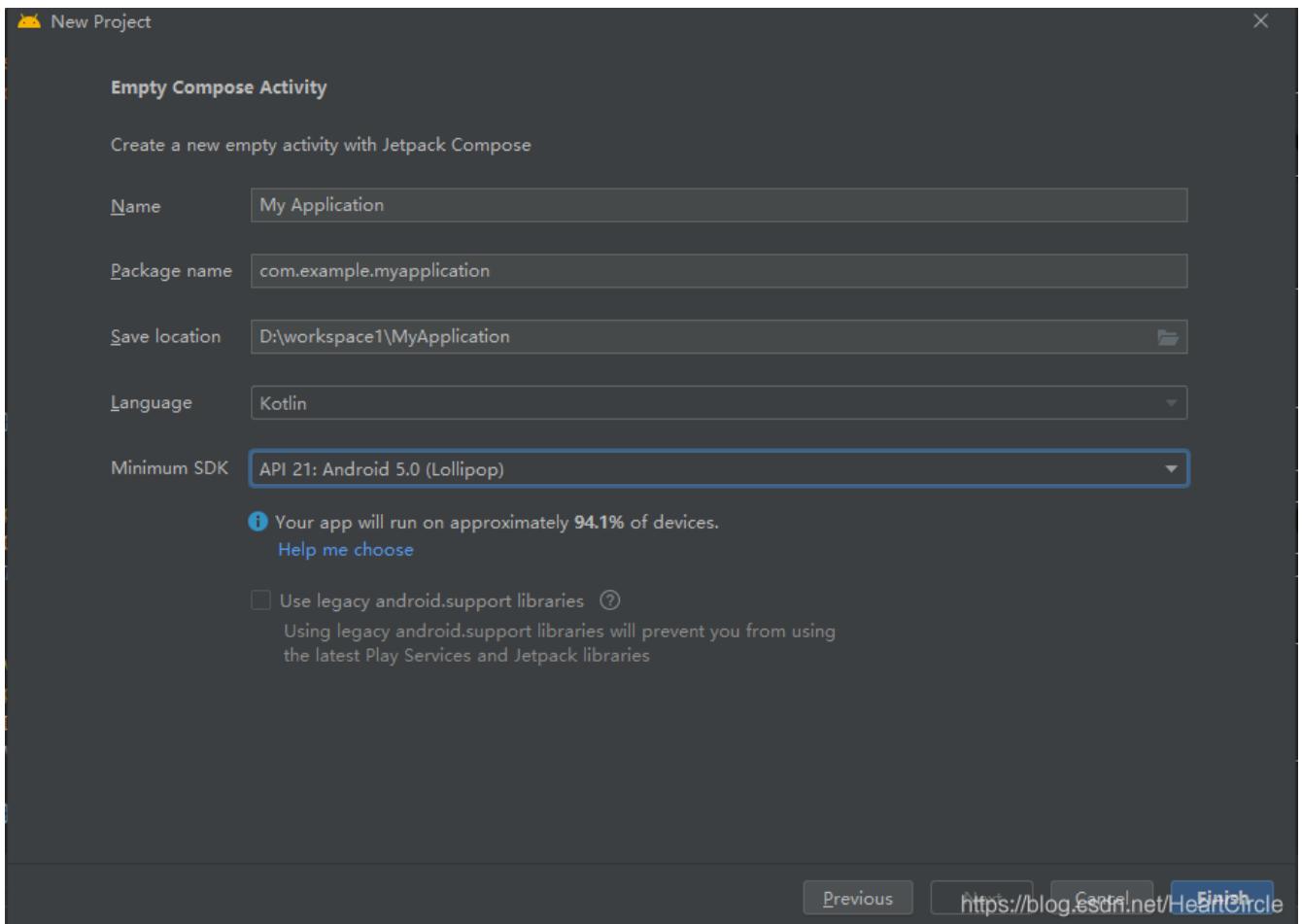
#### 3.1.2 创建DEMO

##### 3.1.2.1 New Project

必须选择 Empty Compose Activity

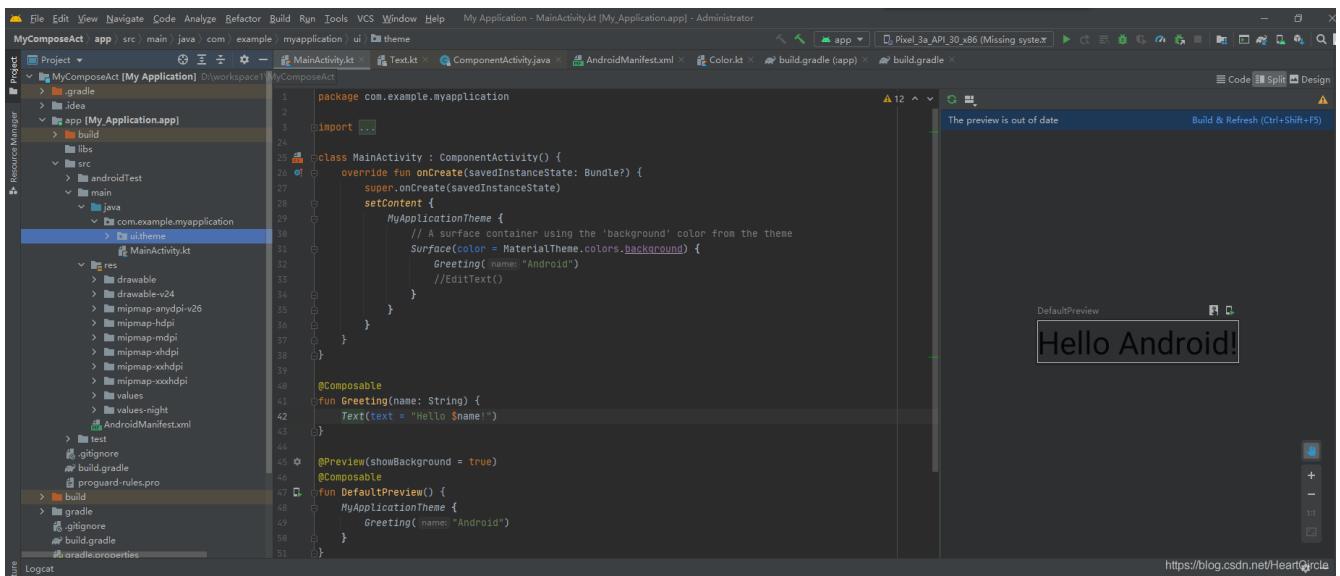


3.1.2.2 项目配置 看到语言只支持Kotlin，而且最低版本要在 5.0以上才行。



### 3.1.3 项目整体变化

**3.1.3.1 布局部分** 左侧的Layout文件夹已经不存在了，中间就是代码部分，右侧是代码布局预览，值得一提的是，现在Jetpack Compose还并不完善，预览的话还得添加一个 @Preview才行。



### 3.1.3.2 依赖的变化 Project.buildGradle

```
1 // Top-level build file where you can add configuration options common to all sub-proj
2 buildscript {
3     ext {
4         compose_version = '1.0.0-beta01'
5     }
6     repositories {
7         google()
8         mavenCentral()
9     }
10    dependencies {
11        classpath "com.android.tools.build:gradle:7.0.0-alpha10"
12        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.4.30"
13
14        // NOTE: Do not place your application dependencies here; they belong
15        // in the individual module build.gradle files
16    }
17 }
18
19 task clean(type: Delete) {
20     delete rootProject.buildDir
21 }
```

<https://blog.csdn.net/HeartCircle>

#### App.BuildGradle

**3.1.3.3 包体大小的变化** 这么多依赖下来，包体整体增加了6m左右，不过随着手机的更新换代，app性能的提升完全能够盖过包体大小的影响。

**3.1.3.4 代码编写的变化** 可以看到 setContentView没有了，取而代之的是使用代码生成布局。以前使用xml形式是方便我们观察布局的排列变化，如果使用代码编写也能达到这种效果，使用代码性能肯定会比加载xml文件的要高。唯一麻烦的是我们需要一定事件去适应这种方式，毕竟老的xml布局都写了那么些年了，突然的改变肯定会有不适应，而且还需要去学习一下Kotlin，如果之前没开发过。

### 3.1.4 遇到的问题

**3.1.4.1 EditText 数据不会变化** 我之前是这么写的，发现输入时日志有打印，但是EditText却没有展示。

```
package com.example.myapplication

import android.os.Bundle
import android.util.Log
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.material.Text
import androidx.compose.material.TextField
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
```

```
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.example.myapplication.ui.theme.MyApplicationTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyApplicationTheme {
                // A surface container using the 'background' color from the theme
                Surface(color = MaterialTheme.colors.background) {
                    Greeting("Android")
                    EditText()
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    MyApplicationTheme {
        Greeting("Android")
    }
}

@Preview(showBackground = true)
@Composable
private fun EditText() {
    var text11: String by mutableStateOf("")
    Box(
        modifier = Modifier
            .size(300.dp, 120.dp),
        contentAlignment = Alignment.Center
    ) {
        Log.e("lbs", "EditText222  ")
        TextField(
```
        modifier = Modifier
            .size(200.dp, 60.dp),
        onValueChange = {

```

```
        Log.e("lbs", "EditText222 $it ")
        text11 = it
    },
    value = text11,
    label = { Text("Countdown Seconds") },
    maxLines = 1,
    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Text)
)
```
}
}
```

后来发现不太对，需要将 var text11: String by mutableStateOf("") 提到全局才行，否则应该是每次进来都是新的。这种方式比较像数据驱动试图改变的形式，有点数据绑定的意思，跟VUE还有点像。

**3.1.4.2 导入其他人写的一个项目，在mumu模拟器报错 "Snapshot is not open" 不知道为啥回去验证这个功能，我看了下代码也没有地方用到啊，还有待研究。不过安装在手机上没有问题，应该是模拟器没有相机的问题。**

03-22 19:48:39.751 1714-1714/com.example.androiddevchallenge E/AndroidRuntime: FATAL EXCEPTION: main  
Process: com.example.androiddevchallenge, PID: 1714  
java.lang.IllegalStateException: Snapshot is not open  
    at androidx.compose.runtime.snapshots.SnapshotKt.validateOpen(Snapshot.kt:1457)  
    at androidx.compose.runtime.snapshots.SnapshotKt.access\$validateOpen(Snapshot.kt:1)  
    at androidx.compose.runtime.snapshots.MutableSnapshot.apply(Snapshot.kt:584)  
    at androidx.compose.runtime.Recomposer.applyAndCheck(Recomposer.kt:759)  
    at androidx.compose.runtime.Recomposer.access\$applyAndCheck(Recomposer.kt:100)  
    at androidx.compose.runtime.Recomposer.performRecompose(Recomposer.kt:1003)  
    at androidx.compose.runtime.Recomposer.access\$performRecompose(Recomposer.kt:100)  
    at  
    androidx.compose.runtime.Recomposer\$runRecomposeAndApplyChanges\$2\$2.invoke(Recomposer.kt:43)  
7)  
    at  
    androidx.compose.runtime.Recomposer\$runRecomposeAndApplyChanges\$2\$2.invoke(Recomposer.kt:41)  
1)  
    at  
    androidx.compose.ui.platform.AndroidUiFrameClock\$withFrameNanos\$2\$callback\$1.doFrame(AndroidUiFrameClock.android.kt:34)  
    at  
    androidx.compose.ui.platform.AndroidUiDispatcher.performFrameDispatch(AndroidUiDispatcher.android.kt:112)  
    at  
    androidx.compose.ui.platform.AndroidUiDispatcher.access\$performFrameDispatch(AndroidUiDispatcher.android.kt:43)  
    at  
    androidx.compose.ui.platform.AndroidUiDispatcher\$dispatchCallback\$1.doFrame(AndroidUiDispatcher.android.kt:72)  
    at android.view.Choreographer\$CallbackRecord.run(Choreographer.java:856)  
    at android.view.Choreographer.doCallbacks(Choreographer.java:670)  
    at android.view.Choreographer.doFrame(Choreographer.java:603)  
    at android.view.Choreographer\$FrameDisplayEventReceiver.run(Choreographer.java:844)  
    at android.os.Handler.handleCallback(Handler.java:739)

```
at android.os.Handler.dispatchMessage(Handler.java:95)
at android.os.Looper.loop(Looper.java:148)
at android.app.ActivityThread.main(ActivityThread.java:5539)
at java.lang.reflect.Method.invoke(Native Method)
at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:745)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:635)
```

## 3.2 Jetpack Compose应用2

刚才已经说了，项目要求非常简单，只要包含一个小狗列表界面，以及每只小狗的详细信息界面即可。首周的挑战项目必须在3月3日下午3:59前完成并提交，因此如果你也有兴趣的话，现在动手可能还来得及（我是用了周六一天时间完成的）。

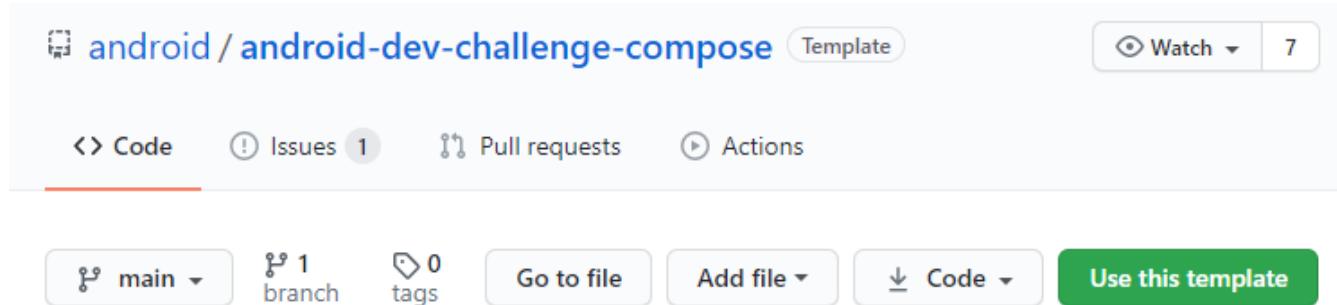
不管是使用Jetpack Compose，还是使用传统的写法去实现，首先你必须要拥有用于展示的数据才行。于是在Google上找了一个专门介绍小狗的网站：

<https://dogtime.com/>

在这个网站上面搜集了一些小狗的信息和图片，这样数据就算是准备好了。

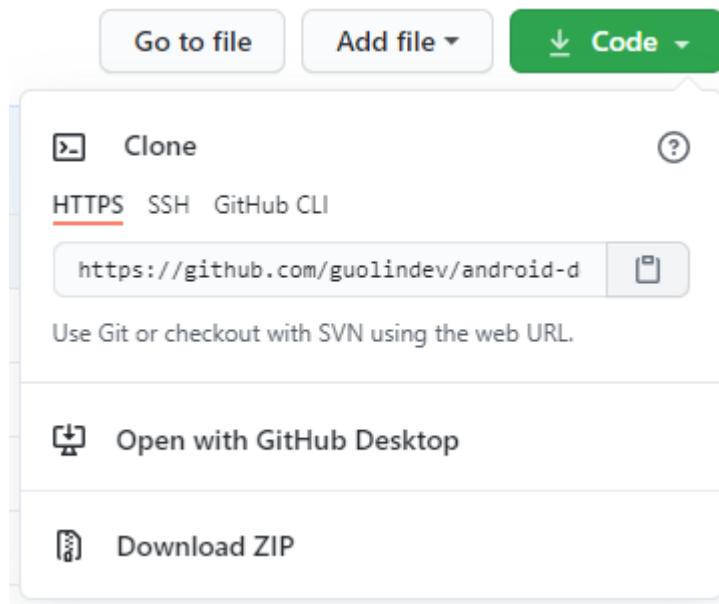
另外，你是不可以自己随便创建一个项目就开始写代码的，Google给我们提供了一个模板，必须在它的模板下编写代码才行。模板地址是：

<https://github.com/android/android-dev-challenge-compose>



打开这个地址，并点击Use this template按钮，就会将这个模板复制一份到你自己的GitHub仓库。

接下来在你自己的GitHub仓库中将项目clone到本地，然后在这里编写代码即可。



至于具体的代码我就不贴出来了，因为基本都是Jetpack Compose相关的代码，而我在本篇文章中是不准备讲解Jetpack Compose的。

尽管很多东西我都还不会，就这样边查边写，我还是在一天时间内把这个项目给做出来了。效果非常简单，不过最基本的项目要求都满足了，如下图所示：

## Puppies Home



可以看到，一个小狗列表界面，以及每只小狗的详细信息界面都有了，另外我们还可以通过点击按钮来领养小狗。

虽然代码已经写完了，但是我在提交代码时才意识到，Google的挑战赛项目并没有那么容易。因为Google设置了一套严格的代码检查机制，你的代码必须是完全符合规范的才能编译通过。

我自认为自己平时的编程风格是非常规范的，并且微软也有这种类似的代码审核机制，但完全没有Google的这套严格。在Google的这套规则中，每个类的头部都要按照固定的格式声明版权。代码中import的包不能使用\*通配符，得一个个手动引入，而且引入的包必须按照字符表的顺序排列。所有代码的换行，空格等等都有严格的规范，少写一个空格，少加一个换行都会导致编译失败。

我基本是按照报错的提示一个个进行修改，但是改了一处，提交代码，又会有其他地方报错。改了老半天，我竟然没有一次代码是可以编译通过的，简直快把我搞崩溃了。

✖ Adjust for spotlessKotlinCheck.	main	16 hours ago	...
Check #9: Commit 953d45d pushed by guolindev		1m 31s	
✖ Make status bar always be light mode.	main	16 hours ago	...
Check #8: Commit 811bc65 pushed by guolindev		1m 37s	
✖ Adjust for spotlessKotlinCheck.	main	yesterday	...
Check #7: Commit 0bb3be1 pushed by guolindev		1m 30s	
✖ Declare copyright for each file.	main	yesterday	...
Check #6: Commit adad3c0 pushed by guolindev		1m 34s	
✖ Remove Wildcard import.	main	yesterday	...
Check #5: Commit 019e2dd pushed by guolindev		1m 38s	
✖ Show loading progress on screen.	main	yesterday	...
Check #4: Commit 3f724eb pushed by guolindev		1m 31s	
✖ Show dog list in MainActivity.	main	yesterday	...
Check #3: Commit cc82c5f pushed by guolindev		1m 28s	
✖ Prepared data to display.	main	2 days ago	...
Check #2: Commit 949778c pushed by guolindev		1m 35s	
✓ Initial commit	main	3 days ago	...
Check #1: Commit 4896250 pushed by guolindev		14m 40s	

后来重新打开了这个挑战项目的GitHub主页，发现主页上有这样一段描述：

## Code formatting

The CI uses [Spotless](#) to check if your code is formatted correctly and contains the right licenses. Internally, Spotless uses [ktlint](#) to check the formatting of your code. To set up ktlint correctly with Android Studio, follow one of the [listed setup options](#).

Before committing your code, run `./gradlew app:spotlessApply` to automatically format your code.

原来项目中会使用Spotless来检查我们的代码是否规范。虽然我并不知道Spotless是个什么工具，但是这个名字听起来就感觉很变态，因为它是完美无暇的意思。也就是说，Google会用这个工具来检查我们的代码是否完美无瑕，一个空格都不会放过。

不过好在它还是给了我们一个简单的解决方案，运行`gradlew app:spotlessApply`命令可以自动将所有代码按照Spotless的规范进行格式化，这样就能够通过检查了。

最后我是通过这种方式来让项目编译通过的。但是在正式项目当中，并不推荐大家这样使用，这就好像我从不建议大家使用`Ctrl + Shift + L`来格式化代码一样。因为用这种快捷的方式可能会导致许多并非你自己编写的代码发生变动，从而对以后追查代码的变动历史造成很大的困扰。

项目编译通过之后，构建任务会转变成绿色，如下图所示：

✓ Update README.md and screenshots.	main	15 hours ago	...
Check #12: Commit 7532e7e pushed by guolindev		11m 41s	
✓ Remove the tricky code.	main	16 hours ago	...
Check #11: Commit 0b5eea3 pushed by guolindev		11m 38s	
✓ Adjust for spotlessKotlinCheck.	main	16 hours ago	...
Check #10: Commit a8e8bc5 pushed by guolindev		12m 39s	

最后，我们还需要按照Google给出的模板修改README.md文件。这个倒是挺简单的，模板都是给好的，往里面填内容就行了。随便写点项目描述，以及这个项目有什么优势即可。

然后刷新仓库页面，你会发现这个模板已经自动帮你生成一个比较漂亮的项目主页了。

# Puppies Home

 Check passing

## Description

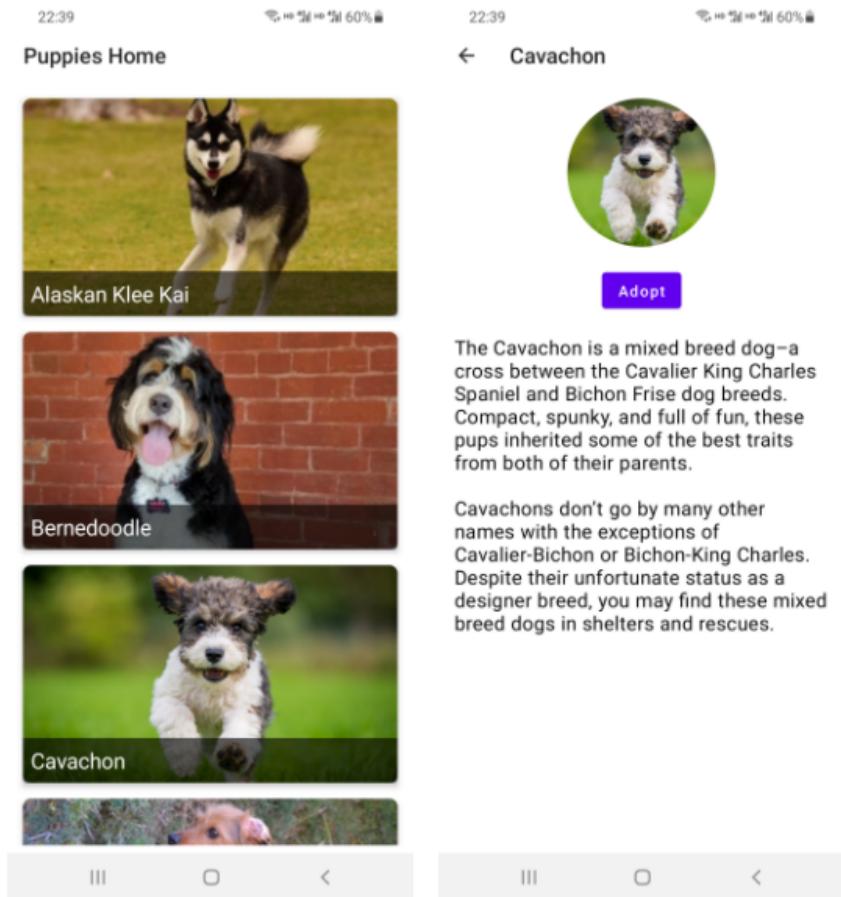
This is a challenge app for adopting puppies written by Jetpack Compose.

The app contains an overview screen that displays a list of puppies, and a detail screen showing each puppy's details.

## Motivation and Context

The UI layer is all implemented by Jetpack Compose. This is a simple and clean project for beginners to read and learn. Also the project included other Jetpack components, so you can learn how Jetpack Compose can work with them as well.

## Screenshots



注意，这个主页的顶部有个Check passing的图标，只有你的项目编译通过了才会显示这个图标。因此从这里就可以迅速看出你提交的代码有没有通过Google的检查。

所有工作都完成了之后，接下来就可以去提交比赛作品了，提交地址是：

<http://goo.gle/dev-challenge-week-1-submission>

令我没想到的是，在提交作品时，Google竟然还要求参赛者必须发条推文：

Tweeted screenshots: the direct link to a Tweet containing 2-3 screenshots of your completed entry project as well as the hashtag #AndroidDevChallenge \*



于是我又去找回了多年没有使用过的Twitter账号，发了一条久违的推文。

Lin Guo @sinyu890807 · 18小时  
#AndroidDevChallenge

Bernedoodle

Cavachon

22:39

← Cavachon

Adopt

The Cavachon is a mixed breed dog—a cross between the Cavalier King Charles Spaniel and Bichon Frise dog breeds. Compact, spunky, and full of fun, these pups inherited some of the best traits

三 O <

回复 转发 收藏 分享

至此，整个参赛过程结束，耗时一天。

希望本文的整理也能对其他有意愿参赛的选手有所帮助，即使首周的比赛可能已经有些来不及了，但是接下来还有3周的比赛，并且后面奖品只会更加丰富。

不过我不知道我自己是否会继续参加后面的比赛，一是未必每周都有时间，二是如果后面挑战项目的难度升级，我可能就做不出来了。所以走一步看一步吧。

如果是想要借助这个项目来学习Jetpack Compose的朋友，也可以参考一下我的实现，源码地址是：

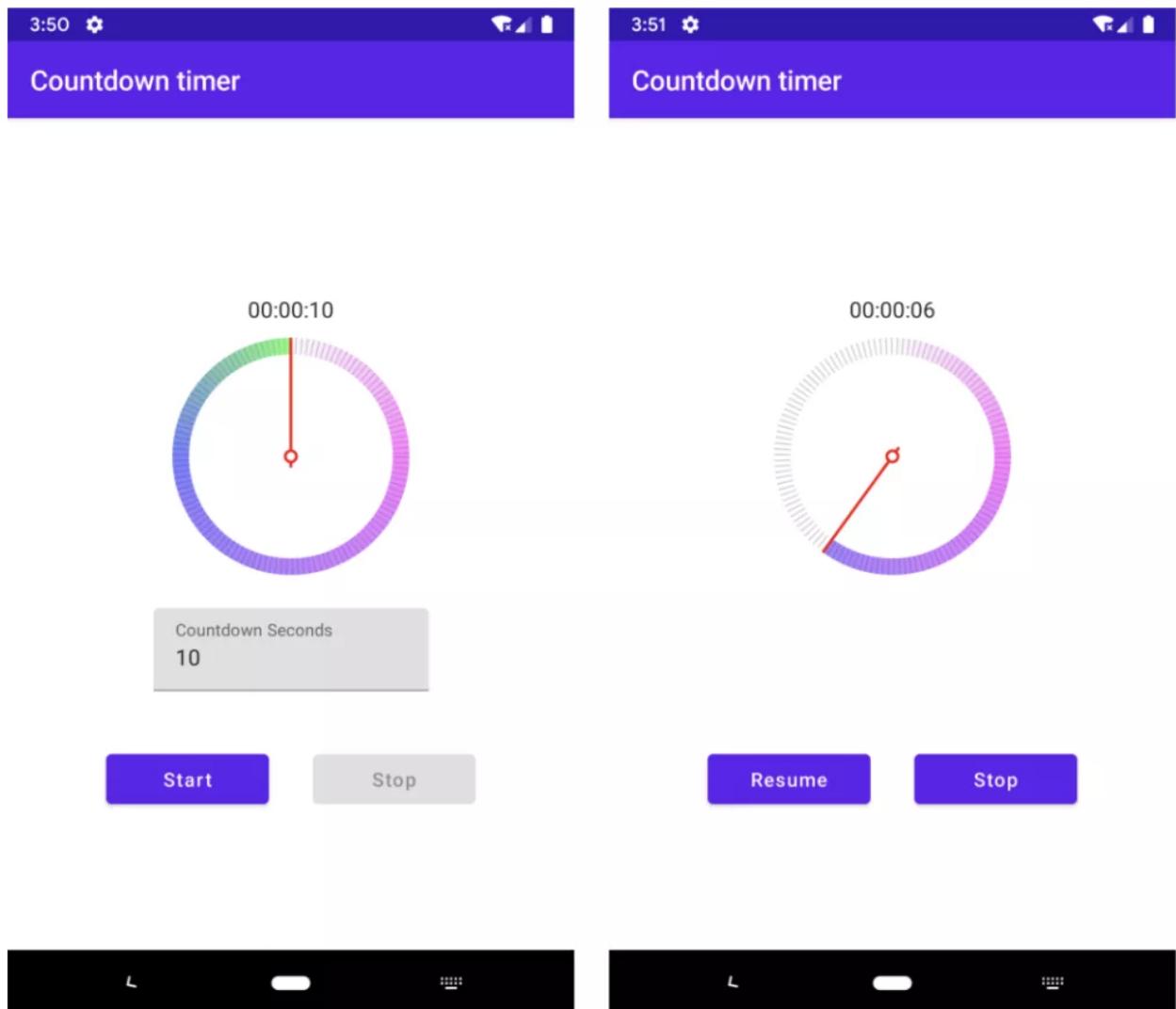
<https://github.com/guolindev/android-dev-challenge-compose>

### 3.3 Jetpack Compose应用做一个倒计时器

虽然是个小项目，但 Compose 的资料实在是太少了，不断地摸索，加上同事的帮助，花费了一天的工夫才做出来，效果如下：



## Screenshots



### 3.3.1 数据结构

首先分析数据结构，我们需要保存用户设置的总时间、当前倒计时剩余时间。除此之外就没有其他数据需要保存了。

TimerViewModel 类如下：

```
// Max input length limit, it's used to prevent number grows too big.  
const val MAX_INPUT_LENGTH = 5  
  
class TimerViewModel : viewModel() {  
  
    /**  
     * Total time user set in seconds  
     */  
    var totalTime: Long by mutableStateOf(0)
```

```

    /**
     * Time left during countdown in seconds
     */
    var timeLeft: Long by mutableStateOf(0)

    /**
     * Update value when EditText content changed
     * @param text new content in EditText
     */
    fun updateValue(text: String) {
        // Just in case the number is too big
        if (text.length > MAX_INPUT_LENGTH) return
        // Remove non-numeric elements
        var value = text.replace("\\D".toRegex(), "")
        // Zero cannot appear in the first place
        if (value.startsWith("0")) value = value.substring(1)
        // Set a default value to prevent NumberFormatException
        if (value.isBlank()) value = "0"
        totalTime = value.toLong()
        timeLeft = value.toLong()
    }
}

```

其中，updateValue 函数用于当用户输入倒计时总秒数后，更新 TimerViewModel 中的 totalTime 和 timeLeft 的值。

为了防止数字过大，我们只允许用户输入 5 位数，并且用正则表达式过滤掉用户输入的小数点、负号、逗号分隔符等非数值。并且数字首位不允许出现 0。经过层层处理，将安全的数值赋给 totalTime 和 timeLeft。

### 3.3.2 倒计时功能

实现倒计时有很多种方式，比如：

- 我们熟悉的 handler.postDelayed 的方式
- 在协程中 repeat + delay 的方式
- 使用 ValueAnimator 的方式

我采用的是第三种方式，因为动画相对来说较容易控制，pause、resume、cancel 函数都是现成的，可以很方便的实现暂停、继续、停止等功能。

AnimatorController 类如下：

```

class AnimatorController(private val viewModel: TimerViewModel) {

    private var valueAnimator: ValueAnimator? = null

    fun start() {
        if (viewModel.totalTime == 0L) return
        if (valueAnimator == null) {
            // Animator: totalTime -> 0
            valueAnimator = ValueAnimator.ofInt(viewModel.totalTime.toInt(), 0)
            valueAnimator?.interpolator = LinearInterpolator()
            // Update timeLeft in viewModel
            valueAnimator?.addUpdateListener {

```

```

        viewModel.timeLeft = (it.animatedValue as Int).toLong()
    }
    valueAnimator?.addListener(object : AnimatorListenerAdapter() {
        override fun onAnimationEnd(animation: Animator?) {
            super.onAnimationEnd(animation)
            complete()
        }
    })
} else {
    valueAnimator?.setIntValues(viewModel.totalTime.toInt(), 0)
}
// (LinearInterpolator + duration) aim to set the interval as 1 second.
valueAnimator?.duration = viewModel.totalTime * 1000L
valueAnimator?.start()
}

fun pause() {
    valueAnimator?.pause()
}

fun resume() {
    valueAnimator?.resume()
}

fun stop() {
    valueAnimator?.cancel()
    viewModel.timeLeft = 0
}

fun complete() {
    viewModel.totalTime = 0
}
}

```

在这个类中，我们处理了动画的启动、暂停、恢复、停止和完成五个功能。通过将 ValueAnimator 设置为在 totalTime 秒内从 totalTime 线性变化到 0 的方式设置出动画的间隔时间为 1s。

为了方便使用，我们将创建的 AnimatorController 对象放到 TimerViewModel 中：

```

class TimerViewModel : ViewModel() {
    //...

    var animatorController = AnimatorController(this)
}

```

### 3.3.3 状态模式

分析可知，倒计时 App 可分为四个状态：

- 尚未开始
- 已经开始
- 暂停
- 完成

由此，我们很容易想到用状态模式来设计此 App，只要为每个状态创建一个状态类，就可以减少大量的 if-else 语句和 when 语句。

状态模式（State Pattern）：当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

在不同状态下，App 的表现和行为是不同的，我们先将这些不同的部分提取到接口中，大致有如下几个方法：

```
interface IStatus {  
    /**  
     * The content string displayed in Start Button.  
     * include: Start, Pause, Resume.  
     */  
    fun startButtonDisplayString(): String  
  
    /**  
     * The behaviour when click Start Button.  
     */  
    fun clickStartButton()  
  
    /**  
     * Stop Button enable status  
     */  
    fun stopButtonEnabled(): Boolean  
  
    /**  
     * The behaviour when click Stop Button.  
     */  
    fun clickStopButton()  
  
    /**  
     * Show or hide EditText  
     */  
    fun showEditText(): Boolean  
}
```

接口中抽出了五个函数，对应 App 在不同状态下的表现和行为：

- fun startButtonDisplayString(): String 用于控制 Start 按钮上的文字显示，在尚未开始/完成状态下，按钮显示的文字为“Start”，在已经开始状态下，按钮显示文字为“Pause”，在暂停状态下，按钮显示文字为“Resume”。
- fun clickStartButton() 用于控制 Start 按钮的点击事件，在尚未开始/完成状态下，点击 Start 按钮启动 ValueAnimator，在已经开始状态下，点击 Start 按钮暂停 ValueAnimator，在暂停状态下，点击 Start 按钮恢复 ValueAnimator。
- fun stopButtonEnabled(): Boolean 用于控制 Stop 按钮是否可点击，在尚未开始/完成状态下，Stop 按钮不可点击，在已经开始/暂停状态下，Stop 按钮可点击。
- fun clickStopButton() 用于控制 Stop 按钮的点击事件，在尚未开始/完成状态下，Stop 按钮不可点击，点击事件为空，在已经开始/暂停状态下，点击 Stop 按钮停止 ValueAnimator。
- fun showEditText(): Boolean 用于控制 EditText 的显示和隐藏，在尚未开始/完成状态下，EditText 显示，在已经开始/暂停状态下，EditText 隐藏。

通过以上分析，我们可以写出以下四个状态类：

尚未开始状态：

```
class NotStartedStatus(private val viewModel: TimerViewModel) : IStatus {

    override fun startButtonDisplayString() = "Start"

    override fun clickStartButton() = viewModel.animatorController.start()

    override fun stopButtonEnabled() = false

    override fun clickStopButton() {}

    override fun showEditText() = true
}
```

已经开始状态：

```
class StartedStatus(private val viewModel: TimerViewModel) : IStatus {

    override fun startButtonDisplayString() = "Pause"

    override fun clickStartButton() = viewModel.animatorController.pause()

    override fun stopButtonEnabled() = true

    override fun clickStopButton() = viewModel.animatorController.stop()

    override fun showEditText() = false
}
```

暂停状态：

```
class PausedStatus(private val viewModel: TimerViewModel) : IStatus {

    override fun startButtonDisplayString() = "Resume"

    override fun clickStartButton() = viewModel.animatorController.resume()

    override fun stopButtonEnabled() = true

    override fun clickStopButton() = viewModel.animatorController.stop()

    override fun showEditText() = false
}
```

完成状态：

```
class CompletedStatus(private val viewModel: TimerViewModel) : IStatus {  
  
    override fun startButtonDisplayString() = "Start"  
  
    override fun clickStartButton() = viewModel.animatorController.start()  
  
    override fun stopButtonEnabled() = false  
  
    override fun clickStopButton() {}  
  
    override fun showEditText() = true  
}
```

同样地，将状态值保存到 ViewModel 中：

```
class TimerViewModel : ViewModel() {  
    //...  
  
    var status: IStatus by mutableStateOf(NotStartedStatus(this))  
}
```

最后，因为四种状态的改变和动画的状态是息息相关的，所以我们可以将状态转移的代码添加到 AnimatorController 类中：

```
class AnimatorController(private val viewModel: TimerViewModel) {  
    //...  
  
    fun start() {  
        //...  
        viewModel.status = StartedStatus(viewModel)  
    }  
  
    fun pause() {  
        //...  
        viewModel.status = PausedStatus(viewModel)  
    }  
  
    fun resume() {  
        //...  
        viewModel.status = StartedStatus(viewModel)  
    }  
  
    fun stop() {  
        //...  
        viewModel.status = NotStartedStatus(viewModel)  
    }  
  
    fun complete() {  
        //...  
        viewModel.status = CompletedStatus(viewModel)  
    }  
}
```

### 3.3.4 Compose 布局

整个布局中，除了时钟的绘制稍微复杂一点外，其他的 UI 都还比较简单。时钟的绘制我们稍后再讲，先从简单的讲起。

#### TimeLeftText

Compose 中，对应 TextView 的函数是 Text，展示剩余时间的 Text 如下：

```
@Composable
private fun TimeLeftText(viewModel: TimerViewModel) {
    Text(
        text = TimeFormatUtils.formatTime(viewModel.timeLeft),
        modifier = Modifier.padding(16.dp)
    )
}
```

通过 text 属性为其设置文字，modifier 属性为其添加了一个 16dp 的 padding。

其中，TimeFormatUtils 工具类代码如下：

```
object TimeFormatUtils {

    fun formatTime(time: Long): String {
        var value = time
        val seconds = value % 60
        value /= 60
        val minutes = value % 60
        value /= 60
        val hours = value % 60
        return String.format("%02d:%02d:%02d", hours, minutes, seconds)
    }
}
```

此工具用于格式化时间，测试类：

```
class TimeFormatUtilsTest : TestCase() {
    @Test
    fun test() {
        Assert.assertEquals("00:00:00", TimeFormatUtils.formatTime(0))
        Assert.assertEquals("00:00:30", TimeFormatUtils.formatTime(30))
        Assert.assertEquals("00:01:00", TimeFormatUtils.formatTime(60))
        Assert.assertEquals("00:10:30", TimeFormatUtils.formatTime(630))
        Assert.assertEquals("01:40:00", TimeFormatUtils.formatTime(6000))
        Assert.assertEquals("27:46:39", TimeFormatUtils.formatTime(99999))
    }
}
```

#### EditText

Compose 中，对应 EditText 的函数是 TextField，本例中，TextField 用于提供给用户输入倒计时总秒数。代码如下：

```
@Composable
private fun EditText(viewModel: TimerviewModel) {
    Box(
        modifier = Modifier
            .size(300.dp, 120.dp),
        contentAlignment = Alignment.Center
    ) {
        if (viewModel.status.showEditText()) {
            TextField(
                modifier = Modifier
                    .size(200.dp, 60.dp),
                value = if (viewModel.totalTime == 0L) "" else
                    viewModel.totalTime.toString(),
                onValueChange = viewModel::updateValue,
                label = { Text("Countdown Seconds") },
                maxLines = 1,
                keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number)
            )
        }
    }
}
```

其中，我们通过状态类中的 showEditText 函数来控制是否需要绘制 TextField，需要注意的是，Compose 中没有 View 的可见行这一概念（也可能是我没找到...），只能通过 if 条件句来控制 View 是否绘制。这在实现 GONE 效果时非常方便，View 也比以前的 GONE 效果消失得更彻底，但本例中，我想实现的效果其实是 INVISIBLE，因为 TextField 的 GONE 效果会导致 Column 中其他控件移位。

在 Compose 中，想要实现 INVISIBLE 效果就只能在 View 外层加一层嵌套，使其仍然占有之前控件的位置。Compose 中的 Box 函数类似之前的 FrameLayout 效果。（如果读者有更好的实现方案欢迎指正。）

当 TextField 中的值将要发生改变时，onValueChange 代码块就会被调用，我们通过 viewModel 的 updateValue 函数干预此过程，保证 viewModel 中只会赋上安全的值。

keyboardOptions 属性用于控制输入类型，虽然这里指定为输入 Number 类型了，但小数点、负号、逗号分隔符无法被过滤掉，这也是 updateValue 中使用正则表达式对输入的字符再过滤一次的原因。

### StartButton

在 Compose 中，Button 不再是一个单纯的 View，Button 函数的最后一个参数是 RowScope，它更像是一个 ViewGroup，其中的内容由我们自己定义：

```
@Composable
private fun StartButton(viewModel: TimerViewModel) {
    Button(
        modifier = Modifier
            .width(150.dp)
            .padding(16.dp),
        enabled = viewModel.totalTime > 0,
        onClick = viewModel.status::clickStartButton
    ) {
        Text(text = viewModel.status.startButtonDisplayString())
    }
}
```

比如此例中，我们在 Button 中绘制了一个 Text，这样去实现一个普通的 Button 效果。Compose 中的 Button 使用起来更为灵活，也就是说，为 XLayout 添加点击事件的时代已经过去，在 Compose 中，可以直接使用 Button 应付这类场景。

### StopButton

StopButton 和 StartButton 是类似的：

```
@Composable
private fun StopButton(viewModel: TimerViewModel) {
    Button(
        modifier = Modifier
            .width(150.dp)
            .padding(16.dp),
        enabled = viewModel.status.stopButtonEnabled(),
        onClick = viewModel.status::clickStopButton
    ) {
        Text(text = "Stop")
    }
}
```

读者可能会产生疑惑，我们在写这几个控件时，只在初始化的时候定义了一下控件的状态，如显示的文字、enabled 状态等。却没有看到 viewModel.status 改变后，更新控件状态的代码。但实际上此时控件的状态已经会自动随着 viewModel.status 的改变而改变了。Compose 是如何做到这点的呢？

魔法就在这一句中：

```
var status: IStatus by mutableStateOf(NotStartedStatus(this))
```

只要是通过 by mutableStateOf 代理的属性，Compose 中使用了此属性的组件就会与此属性自动建立起订阅关系，当此属性发生改变时，Compose 中使用此属性的位置就会自动更新。这和 DataBinding、LiveData 等是类似的，都是观察者模式的运用。

### Scaffold

以上各个控件都已经写好，只需将他们组合起来就行了：

```
class MainActivity : AppCompatActivity() {
    private val viewModel: TimerViewModel by viewModels()
```

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        MyTheme {
            MyApp()
        }
    }
}

override fun onDestroy() {
    super.onDestroy()
    // Release memory
    viewModel.animatorController.stop()
}
}

// Start building your app here!
@Composable
fun MyApp() {
    val viewModel: TimerViewModel = viewModel()
    Scaffold(
        Modifier.fillMaxSize(),
        topBar = {
            TopAppBar(
                title = {
                    Text(
                        text = stringResource(id = R.string.app_name)
                    )
                }
            )
        },
    ) {
        Column(
            modifier = Modifier.fillMaxSize(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            TimeLeftText(viewModel)
            EditText(viewModel)
            Row {
                StartButton(viewModel)
                StopButton(viewModel)
            }
        }
    }
}

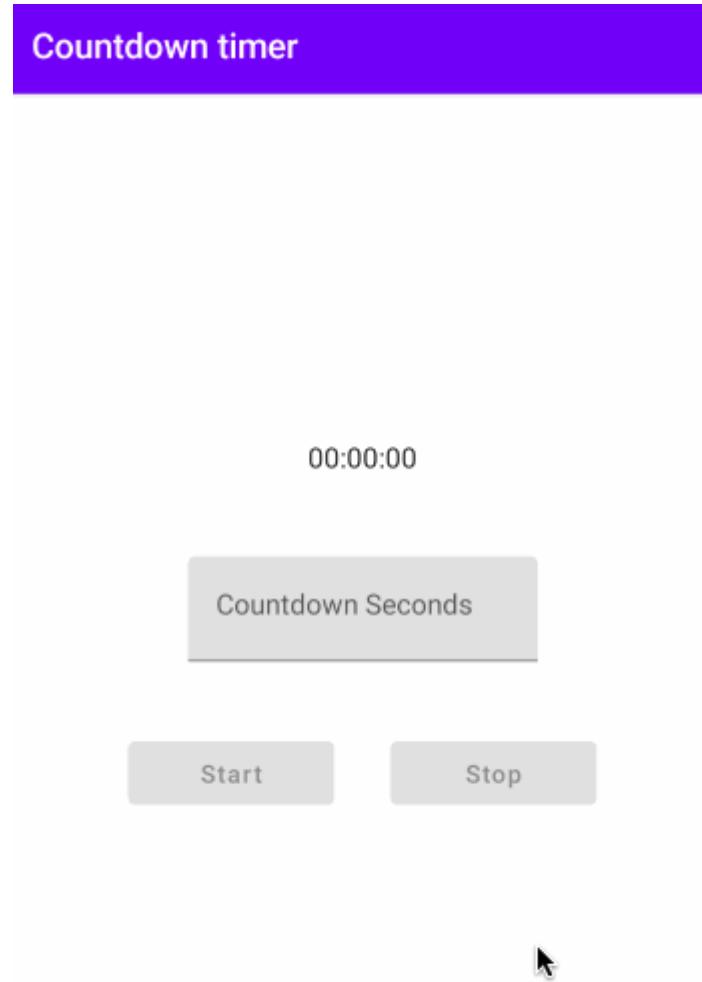
```

其中，Scaffold 是 Material Design 中的概念，它通常包含一个 topBar、一个 bottomBar、一个 floatingActionButton，剩余部分称之为 body。

在此例中，我们给 topBar 声明为带一个 Text 的 TopAppBar，实现之前 ActionBar 的效果。

控件的布局采用了基础的 Column 和 Row (列和行)，分别对应之前 LinearLayout 的 android:orientation="vertical" 和 android:orientation="horizontal"。

Ok，到这里我们可以先运行一下看看效果了，如下图所示：



可以看出，我们需要的功能都成功实现了。

接下来我们在中心区域绘制一个时钟，装饰一下我们的 App，顺便看下如何用 Compose 自定义 View。

### 3.3.5 绘制时钟

绘制时钟前，需要先在 IStatus 中新增一个方法：

```
interface IStatus {  
    //...  
  
    /**  
     * Sweep angle of progress circle  
     */  
    fun progressSweepAngle(): Float  
}
```

此方法用于表示圆环扫过的度数值。

在 StartedStatus 和 PausedStatus 中，此数值的计算方式为：

```
override fun progressSweepAngle() = viewModel.timeLeft * 1.0f / viewModel.totalTime * 360
```

在 NotStartedStatus 中，此数值为：

```
override fun progressSweepAngle() = if (viewModel.totalTime > 0) 360f else 0f
```

在 CompletedStatus 中，此数值为 0f：

```
override fun progressSweepAngle() = 0f
```

然后再通过此数值绘制对应的形状即可。

Compose 中自定义 View 需要使用 androidx.compose.foundation.Canvas 类，其中的 drawXXX 方法与之前 Android 中的 Canvas 都是类似的。Compose 之所以要写一个自己的 Canvas 类，而不是直接使用 Android 中现成的 Canvas 类，就是为了使 Compose 不需要依赖 Android 平台，为以后实现跨平台先铺好路。

```
@Composable
fun ProgressCircle(viewModel: TimerViewModel) {
    // Circle diameter
    val size = 160.dp
    Box(contentAlignment = Alignment.Center) {
        Canvas(
            modifier = Modifier.size(size)
        ) {
            val sweepAngle = viewModel.status.progressSweepAngle()
            // Circle radius
            val r = size.toPx() / 2
            // The width of Ring
            val stokewidth = 12.dp.toPx()
            // Draw dial plate
            drawCircle(
                color = Color.LightGray,
                style = Stroke(
                    width = stokewidth,
                    pathEffect = PathEffect.dashPathEffect(
                        intervals = floatArrayOf(1.dp.toPx(), 3.dp.toPx())
                    )
                )
            )
            // Draw ring
            drawArc(
                brush = Brush.sweepGradient(
                    0f to Color.Magenta,
                    0.5f to Color.Blue,
                    0.75f to Color.Green,
                    0.75f to Color.Transparent,
                    1f to Color.Magenta
                )
            )
        }
    }
}
```

```
        ),
        startAngle = -90f,
        sweepAngle = sweepAngle,
        useCenter = false,
        style = Stroke(
            width = stokewidth
        ),
        alpha = 0.5f
    )
    // Pointer
    val angle = (360 - sweepAngle) / 180 * Math.PI
    val pointTailLength = 8.dp.toPx()
    drawLine(
        color = Color.Red,
        start = Offset(r + pointTailLength * sin(angle).toFloat(), r +
pointTailLength * cos(angle).toFloat()),
        end = Offset((r - r * sin(angle) - sin(angle) * stokewidth / 2).toFloat(),
(r - r * cos(angle) - cos(angle) * stokewidth / 2).toFloat()),
        strokewidth = 2.dp.toPx()
    )
    drawCircle(
        color = Color.Red,
        radius = 5.dp.toPx()
    )
    drawCircle(
        color = Color.White,
        radius = 3.dp.toPx()
    )
}
}
```

可以看到，我们先通过设置 `drawCircle` 函数中的 `pathEffect` 参数，绘制出底部灰色的刻度盘，其中，`intervals` 中的第一个 `Float` 参数指刻度的宽度，第二个 `Float` 参数指刻度的间距。

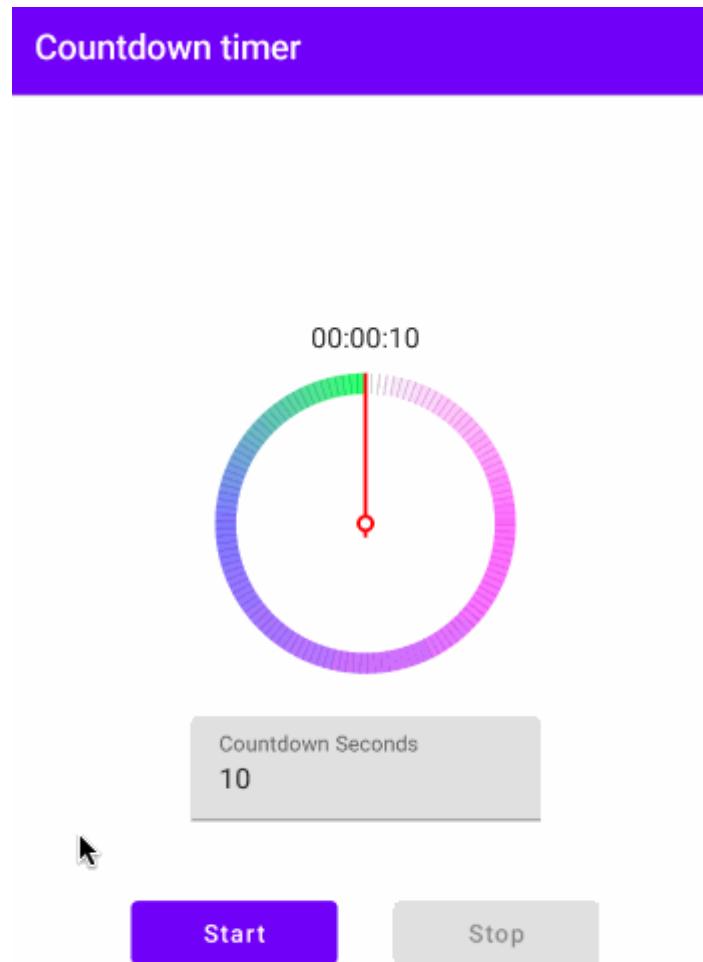
然后从  $-90^\circ$  开始绘制圆环，通过 `sweepAngle` 的变化使得圆环从  $360^\circ$  减到  $0^\circ$ ，通过 `brush` 参数设置出渐变色。

中间的指针部分通过正弦函数、余弦函数的变换计算出起点和终点，绘制出一条直线和两个圆，组成指针的形状。

然后将此控件添加到 TimeLeftText 下方：

```
Column(  
    modifier = Modifier.fillMaxSize(),  
    verticalArrangement = Arrangement.Center,  
    horizontalAlignment = Alignment.CenterHorizontally  
) {  
    TimeLeftText(viewModel)  
    ProgressCircle(viewModel)  
    EditText(viewModel)  
    Row {  
        StartButton(viewModel)  
        StopButton(viewModel)  
    }  
}
```

运行效果如下：



总体看起来还不错，但指针的动画效果还不够丝滑。这是因为我们每隔 1s 才会去更新一次指针扫过的角度，当时间很短时，指针的变化看起来就很突兀。

解决方案也很简单，将更新指针的时间设置得短一些就可以了。前文说到，我们通过将 ValueAnimator 设置为在 totalTime 秒内从 totalTime 线性变化到 0 的方式设置出动画的间隔时间为 1s。想要加快更新的频率，我们可以将动画的初始值扩大一个倍数，总时长保持不变。这时，就不能再使用 timeLeft 来计算扫过的角度了，我们需要一个新的值来保存动画过程中的值。

```
class TimerViewModel : ViewModel() {  
    //...  
  
    /**  
     * Temp value when anim is active  
     */  
    var animValue: Float by mutableStateOf(0.0f)  
}
```

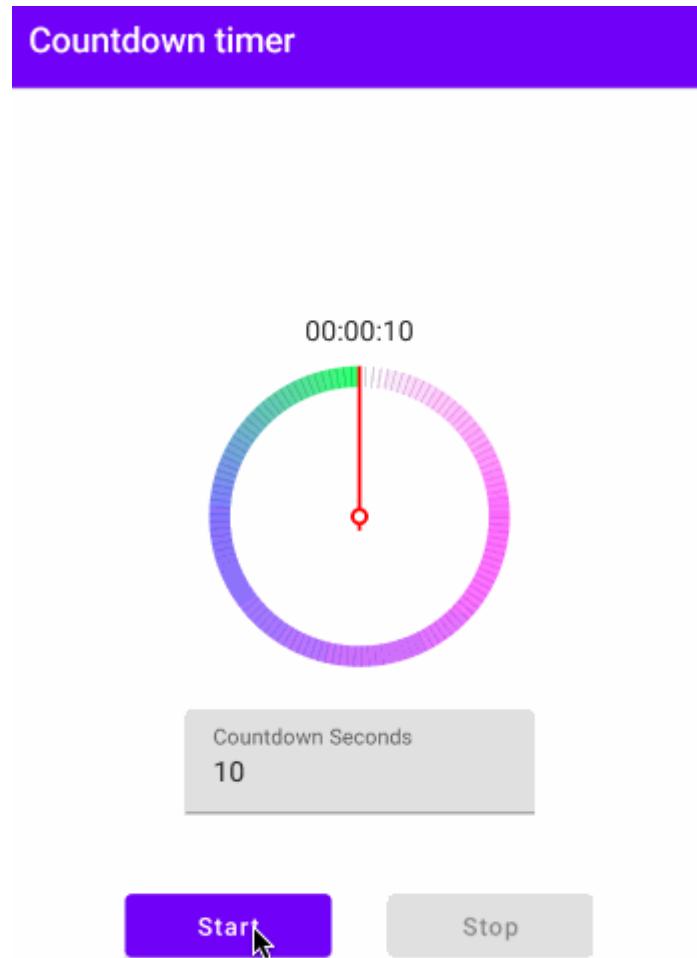
在 AnimatorController 中，加快动画更新的频率：

```
// Control how many times the pointer will be updated in a second  
const val SPEED = 100  
  
class AnimatorController(private val viewModel: TimerViewModel) {  
    //...  
  
    fun start() {  
        if (viewModel.totalTime == 0L) return  
        if (valueAnimator == null) {  
            // Animator: totalTime -> 0  
            valueAnimator = ValueAnimator.ofInt(viewModel.totalTime.toInt() * SPEED, 0)  
            valueAnimator?.interpolator = LinearInterpolator()  
            // Update timeLeft in viewModel  
            valueAnimator?.addUpdateListener {  
                viewModel.animValue = (it.animatedValue as Int) / SPEED.toFloat()  
                viewModel.timeLeft = (it.animatedValue as Int).toLong() / SPEED  
            }  
            valueAnimator?.addListener(object : AnimatorListenerAdapter() {  
                override fun onAnimationEnd(animation: Animator?) {  
                    super.onAnimationEnd(animation)  
                    complete()  
                }  
            })  
        } else {  
            valueAnimator?.setIntValues(viewModel.totalTime.toInt() * SPEED, 0)  
        }  
        // (LinearInterpolator + duration) aim to set the interval as 1 second.  
        valueAnimator?.duration = viewModel.totalTime * 1000L  
        valueAnimator?.start()  
        viewModel.status = StartedStatus(viewModel)  
    }  
}
```

在 StartedStatus 和 PausedStatus 中，使用这个 Float 值来更新 progressSweepAngle：

```
override fun progressSweepAngle() = viewModel.animValue / viewModel.totalTime * 360
```

这里我们将 SPEED 设置为 100，表示每秒钟更新 100 次指针的位置。修改后效果如下（实际效果比 gif 流畅很多，已经看不出卡顿）：



### CompletedText

最后，我们在倒计时结束时添加一个回调，一般的倒计时 App 会在结束时播放一段铃声，简单起见，我们在结束时展示一个“Completed!”文字即可。

同样地，为了避免状态判断，我们利用状态模式的优点，在 IStatus 中添加一个方法：

```
interface IStatus {
    //...

    /**
     * Completed string
     */
    fun completedString(): String
}
```

在 CompletedStatus 中，重载此方法为：

```
override fun completedString() = "Completed!"
```

在其他 Status 子类中，重载此方法为：

```
override fun completedString() = ""
```

创建 CompletedText 控件：

```
@Composable
private fun completedText(viewModel: TimerViewModel) {
    Text(
        text = viewModel.status.completedString(),
        color = MaterialTheme.colors.primary
    )
}
```

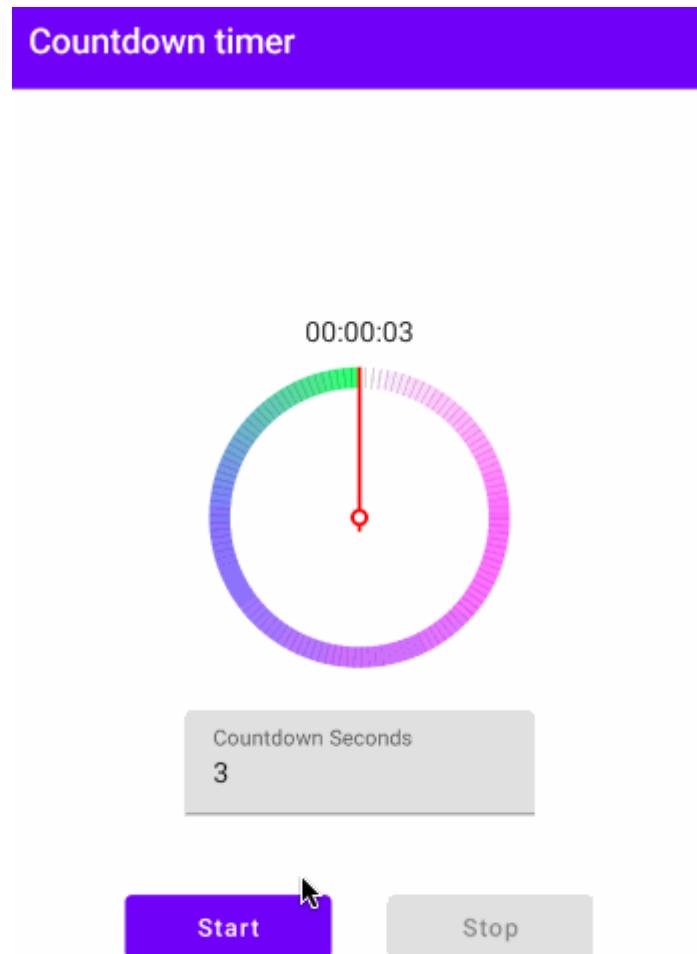
然后，将这个控件添加到 TimeLeftText 上方：

```
Column(
    modifier = Modifier.fillMaxSize(),
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally
) {
    completedText(viewModel)
    TimeLeftText(viewModel)
    ProgressCircle(viewModel)
    EditText(viewModel)
    Row {
        StartButton(viewModel)
        StopButton(viewModel)
    }
}
```

最后，在 EditText 更新时，如果状态是完成，则将其修改为尚未开始状态，因为当用户编辑 EditText 时，说明新一轮倒计时即将到来。

```
fun updateValue(text: String) {  
    //...  
    // After user clicks EditText, CompletedStatus turns to NotStartedStatus.  
    if (status is CompletedStatus) status = NotStartedStatus(this)  
}
```

最终运行效果如下：



这样，我们就使用 Compose 实现了一个完整的 Countdown Timer。源码已上传Github：

<https://github.com/wkxjc/CountdownTimer>

## 3.4 用Jetpack Compose写一个玩安卓App

### 3.4.1 准备工作

我个人的一个小习惯，学习什么新东西的时候就会写个 Demo，之前我写过一个 MVVM 版的玩安卓，而且还为这个项目写过一个系列的文章，感兴趣的可以去我的文章列表看看。

这次写完官方比赛的小 Demo 之后觉得 Compose 挺好玩，并且好多大佬都说 Compose 是未来的趋势，于是就想着把那个 MVVM 版的玩安卓改用 Compose 实现一下试试。

## 先来看看成品



The screenshot shows a mobile application interface. At the top, there's a blue header bar with the text "文章详情". Below it is a white navigation bar with a back arrow, a "首页" button with a dropdown arrow, a search bar containing "探索掘金", and a "登录" button. The main content area features a user profile for "RainyJiang" (Level 1), showing their profile picture, name, and the date "2021年02月25日 阅读 484". A green "关注" (Follow) button is also present. The title of the article is "【疯狂Android之Kotlin】关于Kotlin的高阶函数". Below the title, there's a section titled "Kotlin的高阶函数" and another titled "高阶函数介绍". Under "高阶函数介绍", there's a sub-section "1. 概念". The text discusses what high-order functions are and provides examples. At the bottom of the article, there are social sharing icons for "点赞" (Like), "评论" (Comment), and "收藏" (Save), along with a font size adjustment icon.

The screenshot shows a sign-in page with a blue header bar and a back arrow. The main content area has two input fields: one for "Email" and one for "Password" with a visibility toggle icon. Below the password field is a "Sign in" button. To the right of the sign-in form, there are links for "FORGOT PASSWORD?", "or", and "SIGN IN AS GUEST".

看着是不是也还可以？那就开始着手编写吧！

由于之前已经编写过 MVVM 版本的玩安卓了，所以说很多东西咱们就可以直接进行使用了，比如说一下图片资源，又比如说数据、网络请求等等都是现成了，咱们要做的只是将以前的 xml 布局改成 Compose 即可。

听着是不是很简单？但是写的时候有点懵，这还是我之前写过 Flutter 的情况下，如果大家没有写过 Flutter 或者 SwiftUI 的话看起来可能会更懵，因为里面好多东西都颠覆了我对安卓的看法。。。

为了区分和之前 MVVM 版本的区别，我把这次的 Compose 的版本分支改为了 main 分支，大家下载代码的时候切换下分支就可以了，或者直接下载 main 分支的代码也可以。

### 3.4.2 引入依赖

```
// `Compose`  
implementation "androidx.compose.ui:ui:$compose_version"
```

```
implementation "androidx.compose.runtime:runtime-livedata:$compose_version"
implementation "androidx.compose.material:material:$compose_version"
implementation "androidx.compose.ui:ui-tooling:$compose_version"
implementation "androidx.activity:activity-compose:1.3.0-alpha03"
implementation "androidx.lifecycle:lifecycle-viewmodel-compose:1.0.0-alpha02"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$compose_version"
implementation "androidx.compose.foundation:foundation:$compose_version"
implementation "androidx.compose.foundation:foundation-layout:$compose_version"
implementation "androidx.compose.material:material-icons-extended:$compose_version"

androidTestImplementation "androidx.compose.ui:ui-test:$compose_version"
androidTestImplementation "androidx.compose.ui:ui-test-junit4:$compose_version"

// navigation
implementation "androidx.navigation:navigation-compose:1.0.0-alpha08"
```

what？不是一个 Compose 库吗？干嘛引入这么多？我之前也是这么想的，但是在用的时候一个个又加进去的。。。如果不知道每一个包是什么意思的话可以去官方文档中查看下，不过光看依赖名称基本就知道是什么意思了。。。

如果你也想像我一样在以前的项目中使用 Compose，那么下面的这一步千万别忘了，我就是忘了添加下面这一步找了整整一天的错。

```
android {
    .....
    buildFeatures {
        `Compose` true
        viewBinding true
    }

    `Compose` Options {
        kotlinCompilerExtensionVersion compose_version
        kotlinCompilerVersion kotlin_version
    }
}
```

就是上面的，一定别忘了进行配置，不然找错误能找死。。。给我提示的是 Kotlin 内部 JVM 错误，搞得我都准备给 Kotlin 提 Bug 了。。。

大家可能看到上面的依赖中有 navigation，看名字就知道是专门为 Compose 写的，这也是 Compose 跳转的重要工具，也许有更好的，只是我没有发现吧。

上面也不止一次提到 Compose 颠覆了我之前对安卓的看法，之前的我认为安卓就是一堆 Activity 加上 Fragment，但是写了 Compose 之后我发现并不是这样的，好多官方的 Demo 只有一个 Activity。

看得我有点懵，但是后来想了想就明白了，还是类似 Flutter，在 Flutter 中不也是一个 Activity 嘛，每一个页面也都是一个 Widget！跳转也不是之前的 Intent，而是路由，。现在的 Compose 也是一样，只不过 Widget 改为了 Composable，路由改为了 navigation。

今天就写一下首页框架吧，就是一个底部导航栏加上四个页面，实现点击进行切换。

### 3.4.3 新建 Activity

先来新建一个 Activity 吧，这个项目之后就用这一个 Activity，看下 AndroidManifest：

```
<activity
    android:name=".Compose.NewMainActivity"
    android:theme="@style/AppTheme.NoActionBars">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

这没什么说的，就是把之前的首页改为了新的首页，其它的 Activity 都已经用不到了。

```
class NewMainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            PlayTheme {
                Home()
            }
        }
    }

}
```

这个 Activity 很简单，大家发现了没有，咱们熟悉的 setContentView 方法没有调用，取而代之的是 setContent 方法，不过不重要，这是 Compose 的固定写法而已，当然也可以将 Compose 写到 xml 中然后通过 findViewById 来找到再进行操作，还可以直接 new 出一个 Compose 来进行操作，这里咱们就选择 setContent 这种方法。

看下 setContent 这个方法吧：

```
public fun ComponentActivity.setContent(
    parent: CompositionContext? = null,
    content: @Composable () -> Unit
) {
    val existingComposeView = window.decorView
        .findViewById<ViewGroup>(android.R.id.content)
        .getChildAt(0) as? ComposeView

    if (existingComposeView != null) with(existingComposeView) {
        setParentCompositionContext(parent)
        setContent(content)
    } else ComposeView(this).apply {
        // Set content and parent **before** setContentView
        // to have ComposeView create the composition on attach
        setParentCompositionContext(parent)
        setContent(content)
        setContentView(this, DefaultActivityContentLayoutParams)
    }
}
```

明白了吧？这是 ComponentActivity 的一个扩展方法，里面其实还会执行 setContentView 方法的，并没有什么神器的魔法。。

### 3.4.4 创建 Compose

虽然这里选择了 setContent 这种方法，但是还是说下别的的情况下怎么使用吧。

可以将 ComposeView 放在 XML 布局中，就像放置其他任何 View 一样：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/hello_world"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Hello Android!" />

    <androidx.compose.ui.platform.ComposeView
        android:id="@+id/compose_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

上面的布局很简单，大家注意看，咱们把 Compose 直接写在了 xml 中，这样也是可以进行使用的，怎么使用呢？

```
class ExampleFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        // Inflate the layout for this fragment
        return inflater.inflate(
            R.layout.fragment_example, container, false
        ).apply {
            findViewById<ComposeView>(R.id.compose_view).setContent {
                // In Compose world
                MaterialTheme {
                    Text("Hello Compose!")
                }
            }
        }
    }
}
```

是不是很简单，当然也可以直接 new 出一个 Compose 来进行操作：

```
class ExampleFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return ComposeView(requireContext()).apply {
            setContent {
                MaterialTheme {
                    // In Compose world
                    Text("Hello Compose!")
                }
            }
        }
    }
}
```

好的，这样其实已经满足大部分的需求了，不过还有一种情况：如果同一布局中存在多个 ComposeView 元素，每个元素必须具有唯一的 ID 才能使 savedInstanceState 发挥作用：

```
class ExampleFragment : Fragment() {

    override fun onCreateView(...): View = LinearLayout(...).apply {
        addView(ComposeView(...).apply {
            id = R.id.compose_view_x
            ...
        })
        addView(TextView(...))
        addView(ComposeView(...).apply {
            id = R.id.compose_view_y
            ...
        })
    }
}
```

上面的代码也不难，里面需要注意一点，ComposeView ID 需要在 res/values/ids.xml 文件中进行定义：

```
<resources>
    <item name="compose_view_x" type="id" />
    <item name="compose_view_y" type="id" />
</resources>
```

### 3.4.5 PlayTheme

上面本来想直接写主题来着，但是想了想还是说清楚一点吧，要不使用不同场景的就不知道该如何使用了，咱们来接着看刚才定义的 Activity。

setContent 中包裹了一层 PlayTheme，顾名思义，这是一个自定义的主题，这块也是颠覆我的一个地方。在我之前对安卓的认知中，主题一般存放在 values 中的 styles 文件中，现在 Compose 中已经不再使用 xml 中的主题了，取而代之的是 Compose 自己的一套主题系统。在这里我也吃过一次亏：死活修改不了颜色。

看来 Compose 对 xml 是深恶痛绝啊，一个 xml 都不想使用，包括 color。来看下 PlayTheme 的定义：

```
@Composable
fun PlayTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit
) {
    val colors = if (darkTheme) {
        PlayThemeDark
    } else {
        PlayThemeLight
    }
    MaterialTheme(
        colors = colors,
        typography = typography,
        content = content
    )
}

private val PlayThemeLight = lightColors(
    primary = blue,
    onPrimary = Color.white,
    primaryVariant = blue,
    secondary = blue
)

private val PlayThemeDark = darkColors(
    primary = blueDark,
    onPrimary = Color.white,
    secondary = blueDark,
    surface = blueDark
)
```

定义很简单，但是根据上面描述的内容发现了一些什么没有，连主题都是 Composable，真的像 Flutter 中的 Widget。再来看下里面的颜色值：

```
val blue = Color(0xFF2772F3)
val blueDark = Color(0xFF0B182E)

val Purple300 = Color(0xFFCD52FC)
val Purple700 = Color(0xFF8100EF)
```

### 3.4.6 画页面

准备工作做得差不多了，来开始画页面吧！

先想想咱们最终需要做的样子，忘记的可以滑到上面再看看。

其实很简单，今天咱们只是初探嘛！只需要画出下面的底部导航栏和上面几个空页面就行了！

说干就干！先来创建一个新的 Composable：

```
@Composable
fun Home() {
}
```

很简单，一个方法加上 @Composable 的注解就是一个新的 Composable 了，咱们需要在这里画咱们的首页了。

### 3.4.7 底部导航栏

查了一下官方文档，Compose 中和 Flutter 一样有现成底部导航栏，完全够咱们使用了：

```
@Composable
fun Home() {
    ComposeDemoTheme {
        val (selectedTab, setSelectedTab) = remember { mutableStateOf(CourseTabs.HOME_PAGE) }
        val tabs = CourseTabs.values()
        Scaffold(
            backgroundColor = MaterialTheme.colors.primarySurface,
            bottomBar = {
                BottomNavigation(
                    Modifier.navigationBarsHeight(additional = 56.dp)
                ) {
                    tabs.forEach { tab ->
                        BottomNavigationItem(
                            icon = { Icon(painterResource(tab.icon), contentDescription =
null) },
                            label = { Text(stringResource(tab.title).toUpperCase()) },
                            selected = tab == selectedTab,
                            onClick = { setSelectedTab(tab) },
                            alwaysShowLabel = false,
                            selectedContentColor = MaterialTheme.colors.secondary,
                            unselectedContentColor = LocalContentColor.current,
                            modifier = Modifier.navigationBarsPadding()
                        )
                    }
                }
            }
        ) { innerPadding ->
            val modifier = Modifier.padding(innerPadding)
            when (selectedTab) {
                CourseTabs.HOME_PAGE -> One(modifier)
                CourseTabs.PROJECT -> Two(modifier)
                CourseTabs.OFFICIAL_ACCOUNT -> Three(modifier)
                CourseTabs.MINE -> Four(modifier)
            }
        }
    }
}
```

上面代码有点长，但是意思很简单，稍微给大家说一下吧，Scaffold 在 Flutter 中也有，意思也是差不多的，来看一下 Scaffold 的源码吧：

```
@Composable
fun Scaffold(
    modifier: Modifier = Modifier,
    scaffoldState: ScaffoldState = rememberScaffoldState(),
    topBar: @Composable () -> Unit = {},
    bottomBar: @Composable () -> Unit = {},
    snackbarHost: @Composable (SnackbarHostState) -> Unit = { SnackbarHost(it) },
    floatingActionButton: @Composable () -> Unit = {},
    floatingActionButtonPosition: FabPosition = FabPosition.End,
    isFloatingActionButtonDocked: Boolean = false,
    drawerContent: @Composable (ColumnScope.() -> Unit)? = null,
    drawerGesturesEnabled: Boolean = true,
    drawerShape: Shape = MaterialTheme.shapes.large,
    drawerElevation: Dp = DrawerDefaults.Elevation,
    drawerBackgroundColor: Color = MaterialTheme.colors.surface,
    drawerContentColor: Color = contentColorFor(drawerBackgroundColor),
    drawerScrimColor: Color = DrawerDefaults.scrimColor,
    backgroundColor: Color = MaterialTheme.colors.background,
    contentColor: Color = contentColorFor(backgroundColor),
    content: @Composable (PaddingValues) -> Unit
)
```

它就是一个脚手架，官方是这样进行描述的：

Material 支持的最高级别的可组合项是 Scaffold。Scaffold 可让您实现具有基本 Material Design 布局结构的界面。Scaffold 可以为最常见的顶级 Material 组件（如 TopAppBar、BottomAppBar、FloatingActionButton 和 Drawer）提供插槽。通过使用 Scaffold，很容易确保这些组件得到适当放置且正确地协同工作。

意思很明确，如果不是要求自定义度特别高的页面，使用 Scaffold 就完全能满足需求了，这里咱们使用的就是。

上面代码中的 CourseTabs 还没有写，它是一个枚举类，用来表示首页的几个页面的：

```
enum class CourseTabs(
    @StringRes val title: Int,
    @DrawableRes val icon: Int
) {
    HOME_PAGE(R.string.home_page, R.drawable.ic_nav_news_normal),
    PROJECT(R.string.project, R.drawable.ic_nav_tweet_normal),
    OFFICIAL_ACCOUNT(R.string.official_account, R.drawable.ic_nav_discover_normal),
    MINE(R.string.mine, R.drawable.ic_nav_my_normal)
}
```

这里其实有的地方大家还是看不太懂的，比如上面的 remember 是个什么东西？

### 3.4.8 管理状态

上面所说的的 remember 其实是用来管理 Compose 的状态的，大家就先记着 remember 可以记录咱们点击的按钮数据，从而驱使页面发生改变吧。

mutableStateOf(CourseTabs.HOME\_PAGE) 其实是 MutableState

[mutableStateOf] :

[https://developer.android.google.cn/reference/kotlin/androidx/compose/runtime/package-summary#mutableStateOf\(androidx.compose.runtime.mutableStateOf.T](https://developer.android.google.cn/reference/kotlin/androidx/compose/runtime/package-summary#mutableStateOf(androidx.compose.runtime.mutableStateOf.T)

androidx.compose.runtime.SnapshotMutationPolicy)) 会创建可观察的 MutableState , MutableState 是与 Compose 运行时集成的可观察类型。

这块一时半会说不清，需要后面的文章慢慢来和大家讲。

### 3.4.9 添加页面

上面一共创建了四个字页面：one、two、three、four，四个页面非常简单：

```
@Composable
fun One(modifier: Modifier) {
    Text(modifier = modifier
        .fillMaxSize()
        .padding(top = 100.dp), text = "One", color = Teal200)
}

@Composable
fun Two(modifier: Modifier) {
    Text(modifier = modifier
        .fillMaxSize()
        .padding(top = 100.dp), text = "Two", color = Teal200)
}

@Composable
fun Three(modifier: Modifier) {
    Text(modifier = modifier
        .fillMaxSize()
        .padding(top = 100.dp), text = "Three", color = Teal200)
}

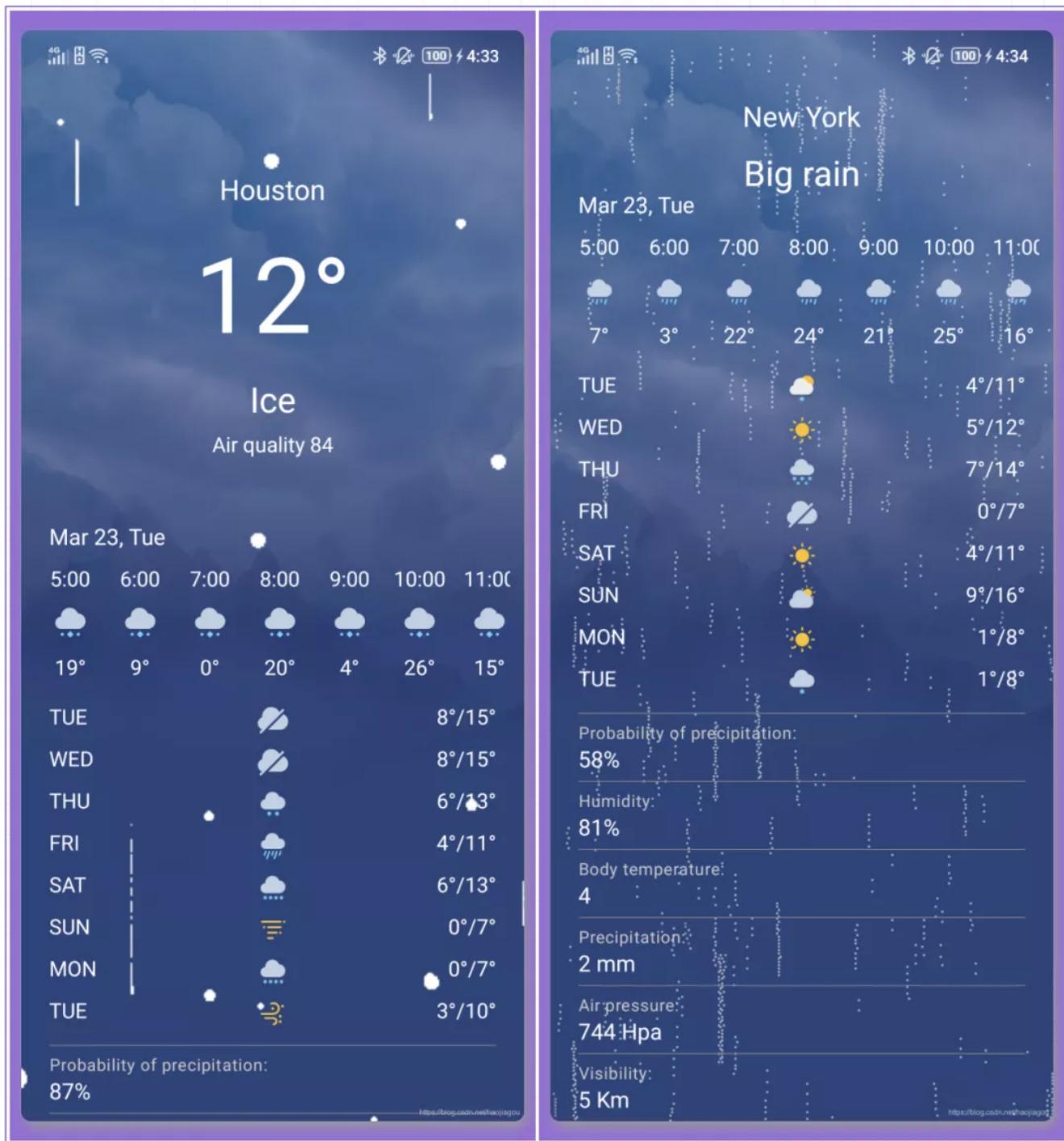
@Composable
fun Four(modifier: Modifier) {
    Text(modifier = modifier
        .fillMaxSize()
        .padding(top = 100.dp), text = "Four", color = Teal200)
}
```

ok，这就差不多了。

## 3.5 用Compose Android 写一个天气应用

### 3.5.1 开篇

在开搞之前还是先来看看最终的实现效果吧，上面 Gif 图有点卡顿，来看看静态的效果：



数据什么的都是模拟的假数据，其它就没啥了，都是 Compose 的简单使用。下面就来和大家唠唠实现过程吧。

## 模拟数据

第一步咱们先把数据给模拟出来吧，没有数据页面画起来也不好画。看看上面的图，咱们来总结下需要哪些数据：

- 地址**：这必须有吧，显示在第一行的，光有个天气谁知道是哪的天气，虽然是模拟的，也得像真的是不？
- 天气**：这更得有，天气预报没有天气哪能行！
- 当前温度**：这也是必要的，天气预报应用基本都有这个功能。
- 空气质量**：人们都非常关心的东西，加上吧。
- 24小时天气**：每个小时具体的天气预报，这也得有
- 未来一周天气**：预报嘛，肯定得预报啊

- **天气基本信息**：比如降水概率啊，湿度啊，紫外线啊什么的

嗯，上面列举的差不多了，想要数据肯定得先有实体类来存放数据吧，咱们来看看实体类的写法吧：

```
data class Weather(
    val weather: Int = R.string.weather_sunny,
    val address: Int = R.string.city_new_york,
    val currentTemperature: Int = 0,
    val quality: Int = 0,
    @DrawableRes val background: Int = R.drawable.home_bg_1,
    @DrawableRes val backgroundGif: Int = R.drawable.bg_topgif_2,
    val twentyFourHours: List<TwentyFourHour> = arrayListOf(),
    val weekWeathers: List<WeekWeather> = arrayListOf(),
    val basicWeathers: List<BasicWeather> = arrayListOf()
)
```

是不是发现上面代码中多了几个东西，没事，别着急，这就说是啥意思。就算我不说大家肯定也都知道，不就是背景图片和背景 gif 图嘛！没错，就是！

接下来看看 TwentyFourHour、WeekWeather 和 BasicWeather 这三个类吧：

```
data class TwentyFourHour(
    val time: String = "",
    @DrawableRes val icon: Int,
    val temperature: String
)

data class WeekWeather(
    val weekStr: String = "",
    @DrawableRes val icon: Int,
    val temperature: String = ""
)

data class BasicWeather(
    val name: Int,
    val value: String = ""
)
```

是不是很简单，就不细说了。

下面就来看看数据的定义吧。我简单定义了下平时可能遇到的天气状况：

```
<string name="weather_sunny">晴</string>
<string name="weather_cloudy">多云</string>
<string name="weather_overcast">阴</string>
<string name="weather_small_rain">小雨</string>
<string name="weather_mid_rain">中雨</string>
<string name="weather_big_rain">大雨</string>
<string name="weather_rainstorm">暴雨</string>
<string name="weather_small_snow">小雪</string>
<string name="weather_mid_snow">中雪</string>
<string name="weather_big_snow">大雪</string>
```

```
<string name="weather_snowstorm">暴风雪</string>
<string name="weather_foggy">雾</string>
<string name="weather_ice">结冰</string>
<string name="weather_haze">阴霾</string>
```

嗯，就写这些吧，肯定还有很多种天气，这里就不写那么细了，大家如果想加就下载代码自己加吧。

下面就需要一些审美了，我找了一些现在应用商店的天气预报的应用，看了看那个好看，“下载”点资源图去，要不自己怎么搞。。。这块详细步骤就不写了，大家自行百度。

数据差不多都有了，那么该怎么一一对应呢？可能我没说明白，比如说你模拟的天气是下雨，你的 gif 图总不能是在下雪吧？背景图片也不能是冰天雪地啊，对不？这块我使用的方法是枚举，将不同天气及资源进行一一对应：

```
enum class WeatherEnum(
    @StringRes val weather: Int,
    @DrawableRes val icon: Int,
    @DrawableRes val background: Int,
    @DrawableRes val backgroundGif: Int,
) {
    SUNNY(
        R.string.weather_sunny,
        R.drawable.n_weather_icon_sunny,
        R.drawable.home_bg_1,
        R.drawable.bg_topgif_10
    ),
    CLOUDY(
        R.string.weather_cloudy,
        R.drawable.n_weather_icon_cloud,
        R.drawable.home_bg_4,
        R.drawable.bg_topgif_10
    ),
    OVERCAST(
        R.string.weather_overcast,
        R.drawable.n_weather_icon_overcast,
        R.drawable.home_bg_6,
        R.drawable.bg_topgif_10
    ),
}
```

这块由于篇幅原因就不写全了，写三个作为演示吧，后面的天气状况也是这么写。

## 编写ViewModel

数据实体类和资源都准备好了，就差个 ViewModel 来提供数据了，说干就干，整一个 ViewModel。

```

class weatherPageViewModel : viewModel() {

    private val _weatherLiveData = MutableLiveData<Weather>()
    val weatherLiveData: LiveData<Weather> = _weatherLiveData

    private fun onWeatherChanged(weather: Weather) {
        _weatherLiveData.value = weather
    }

}

```

先简单定义一个 ViewModel , 什么 ? 没看明白 ? 回去重新看 MVVM 去。再来写一个方法 , 提供给外部获取天气的方法 :

```

fun getWeather() {
    val random = Random()
    val city = cityArray[random.nextInt(5)]
    val weatherEnums = WeatherEnum.values()
    val weatherEnum = weatherEnums[random.nextInt(14)]
    val calendar = Calendar.getInstance()
    val hours: Int = calendar.get(Calendar.HOUR)
    val twentyFourHours = arrayListOf<TwentyFourHour>()
    val weekWeathers = arrayListOf<WeekWeather>()
    for (index in hours + 1..24) {
        twentyFourHours.add(
            TwentyFourHour(
                "$index:00",
                weatherEnum.icon,
                "${random.nextInt(29)}°"
            )
        )
    }
    val week = calendar.get(Calendar.DAY_OF_WEEK)
    val weekListString = DateUtils.getWeekListString(week = week)
    for (index in weekListString.indices) {
        val small = random.nextInt(10)
        weekWeathers.add(
            WeekWeather(
                weekListString[index],
                getWeatherIcon(random.nextInt(35)), "$small°/${small + 7}°"
            )
        )
    }
}

val basicWeathers = arrayListOf<BasicWeather>()
basicWeathers.add(BasicWeather(R.string.basic_rain, "${random.nextInt(100)}%"))
basicWeathers.add(BasicWeather(R.string.basic_humidity, "${random.nextInt(100)}%"))

val weather = Weather(
    weatherEnum.weather,
    address = city,
    currentTemperature = random.nextInt(30),

```

```

        quality = random.nextInt(100),
        background = weatherEnum.background,
        backgroundGif = weatherEnum.backgroundGif,
        twentyFourHours = twentyFourHours,
        weekweathers = weekweathers,
        basicweathers = basicweathers
    )
    onweatherChanged(weather)
}

```

数据很简单，大部分是直接通过 Random 来随机生成的，第一次进入或刷新的时候就可以生成了。

### 3.5.2 画页面

数据都准备好了，就差页面了，画一画吧，不管什么时候，页面都是比较简单的，相对于数据逻辑来说，可能我这话说的有点绝对，也可能因为我太年轻。。。不多说了，开始画吧！

咱们就从 Activity 开始吧：

```

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        BarUtils.transparentStatusBar(this)
        setContent {
            MyTheme {
                WeatherPage()
            }
        }
    }
}

```

上面代码很简单，为什么这么写在这里就不说了，大家可以去看我之前的文章。对了，里面有一行代码，顾名思义，就是设置状态栏为透明的，代码很容易找到，就不贴了。

接着来看 WeatherPage：

```

@Composable
fun WeatherPage() {
    val refreshingState = remember { mutableStateOf(REFRESH_STOP) }
    val weatherPageViewModel: WeatherPageViewModel = viewModel()
    val weather by weatherPageViewModel.weatherLiveData.observeAsState(weather())
    var loadState by remember { mutableStateOf(false) }
    if (!loadState) {
        loadState = true
        weatherPageViewModel.getWeather()
    }

    Surface(color = MaterialTheme.colors.background) {
        SwipeToRefreshLayout(
            refreshingState = refreshingState.value,
            onRefresh = {
                refreshingState.value = REFRESH_START
                weatherPageViewModel.getWeather()
            }
        )
    }
}

```

```

        loadState = true
        refreshingState.value = REFRESH_STOP
    },
    progressIndicator = {
        ProgressIndicator()
    }
) {
    weatherBackground(weather)
    weatherContent(weather)
}
}
}
}

```

这块代码就得稍微说一说了，先从 Surface 看起，里面包裹了一个 SwipeRefreshLayout，看过上一篇文章的应该知道，这是下拉刷新的控件，如果没看过上一篇文章的，可以先去看看：Compose 实现下拉刷新和上拉加载。

再来看上面的内容：

- refreshingState 就是是否正在刷新的状态，在 onRefresh 开始时设置为 REFRESH\_START，刷新完成之后设置为 REFRESH\_STOP，默认状态也是 REFRESH\_STOP。
- weatherPageViewModel 就是上面咱们写的，不过多解释
- weather 这个就是将 ViewModel 中的 LiveData 转为 Compose 中支持观察的 State。
- loadState 是记住是否加载过，避免重复加载数据。

onRefresh中的刷新内容就是直接调一下weatherPageViewModel中的getWeather()方法。

然后直接开始看大括号中的内容，看着应该就知道是啥意思了，WeatherBackground 是背景，WeatherContent 是内容，那为什么要传入 Weather 呢？当然是为了展示了。。。

### 3.5.3 画背景

接下来先来看看 WeatherBackground 吧：

```

@Composable
fun WeatherBackground(weather: Weather) {
    Box {
        Image(
            modifier = Modifier.fillMaxSize(),
            painter = painterResource(weather.background),
            contentDescription = stringResource(id = weather.weather),
            contentScale = ContentScale.Crop
        )
        val context = LocalContext.current
        val glide = Glide.with(context)
        CompositionLocalProvider(LocalRequestManager provides glide) {
            GlideImage(
                modifier = Modifier.fillMaxSize(),
                data = weather.backgroundGif,
                contentDescription = stringResource(id = weather.weather),
                contentScale = ContentScale.Crop
            )
        }
    }
}

```

很简单，一张背景图，一张 gif 动态图，用来展示下雨或者下雪等特效。这块的动态图使用了 Glide 的 gif 展示功能。使用方法上面已经贴出来了，下面贴下依赖吧：

```
implementation "dev.chrisbanes.accompanist:accompanist-glide:0.6.0"
```

### 3.5.4 画内容

WeatherContent 是内容，这里有好几块，咱们慢慢看：

```
@Composable
fun WeatherContent(weather: Weather) {
    val scrollState = rememberScrollState()
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(horizontal = 10.dp)
            .verticalScroll(scrollState),
    ) {
        weatherBasic(weather, scrollState)
        WeatherDetails(weather)
        weatherWeek(weather)
        weatherOther(weather)
    }
}
```

可以看到分为好几块，分别对应上面图中的几块。上面还写的有 scrollState，保存着滚动的状态，由于咱们想要竖着滑动，所以设置 verticalScroll(scrollState)。

#### WeatherBasic

这块是天气的基本信息，也就是城市啊、天气状况啊、当前温度啊啥的，上面的代码中还将滚动状态传入了这里，下面咱们就能看到作用了：

```
@Composable
fun WeatherBasic(weather: Weather, scrollState: ScrollState) {
    val offset = (scrollState.value / 2)
    val fontSize = (100f / offset * 70).coerceAtLeast(30f).coerceAtMost(75f).sp
    val modifier = Modifier
        .fillMaxWidth()
        .wrapContentWidth(Alignment.CenterHorizontally)
        .graphicsLayer { translationY = offset.toFloat() }
    val context = LocalContext.current
    Text(
        modifier = modifier.padding(top = 100.dp, bottom = 5.dp),
        text = stringResource(id = weather.address), fontSize = 20.sp,
        color = Color.White,
    )
    AnimatedVisibility(visible = fontSize == 75f.sp) {
        Text(
            modifier = modifier.padding(top = 5.dp, bottom = 5.dp),
            text = "${weather.currentTemperature}°",
        )
    }
}
```

```

        fontSize = fontsize,
        color = Color.White
    )
}
Text(
    modifier = modifier.padding(top = 5.dp, bottom = 2.5.dp),
    text = stringResource(id = weather.weather), fontSize = 25.sp,
    color = Color.White
)
AnimatedVisibility(visible = fontsize == 75f.sp) {
    Text(
        modifier = modifier.padding(top = 2.5.dp),
        text = stringResource(id = R.string.weather_air_quality) + " " +
weather.quality,
        fontSize = 15.sp,
        color = Color.White
    )
}
Text(
    modifier = Modifier.padding(top = 45.dp, start = 10.dp),
    text = DateUtils.getDefaultDate(context, System.currentTimeMillis()),
    fontSize = 16.sp,
    color = Color.White
)
}
}

```

是不是很简单？使用 AnimatedVisibility 来控制是否显示当前温度和空气质量，嗯，就没了。。。还有啥？大家可以试试 Modifier 的各种功能，越用越发现这个东西的强大。。。

## WeatherDetails

这个吧，就是 24 小时天气详情，上面数据已经写好了，直接展示即可：

```

@Composable
fun WeatherDetails(weather: Weather) {
    val twentyFourHours = weather.twentyFourHours
    LazyRow(modifier = Modifier.fillMaxWidth()) {
        items(twentyFourHours) { twentyFourHour ->
            WeatherHour(twentyFourHour)
        }
    }
}

@Composable
fun WeatherHour(twentyFourHour: TwentyFourHour) {
    val modifier = Modifier.padding(top = 9.dp)
    Column(modifier = Modifier.width(50.dp), horizontalAlignment =
    Alignment.CenterHorizontally) {
        Text(modifier = modifier, text = twentyFourHour.time, color = Color.White, fontSize
        = 15.sp)
        Image(
            modifier = modifier.size(25.dp),
            painter = painterResource(id = twentyFourHour.icon),
        )
    }
}

```

```

        contentDescription = twentyFourHour.temperature
    )
    Text(
        modifier = modifier,
        text = twentyFourHour.temperature,
        color = Color.White,
        fontSize = 15.sp
    )
}
}
}

```

一个 LazyRow 就搞定了，是不是很省事，比之前的 RecyclerView 还简单。。。

### WeatherWeek

这是下面的未来一周的天气，和上面 24 小时类似：

```

@Composable
fun weatherweek(weather: Weather) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(top = 10.dp)
            .padding(horizontal = 10.dp)
    ) {
        for (weekweather in weather.weekweathers) {
            weatherweekDetails(weekweather)
        }
    }
}

```

里面具体的 WeatherWeekDetails 由于篇幅原因就不贴代码了，大家可以去下载代码看。

### WeatherOther

这个是当天天气的一些值，比如降水概率啊、湿度啊啥的：

```

@Composable
fun weatherother(weather: Weather) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(top = 10.dp)
            .padding(horizontal = 10.dp)
    ) {
        for (weekweather in weather.basicweathers) {
            weatherotherDetails(weekweather)
        }
    }
}

```

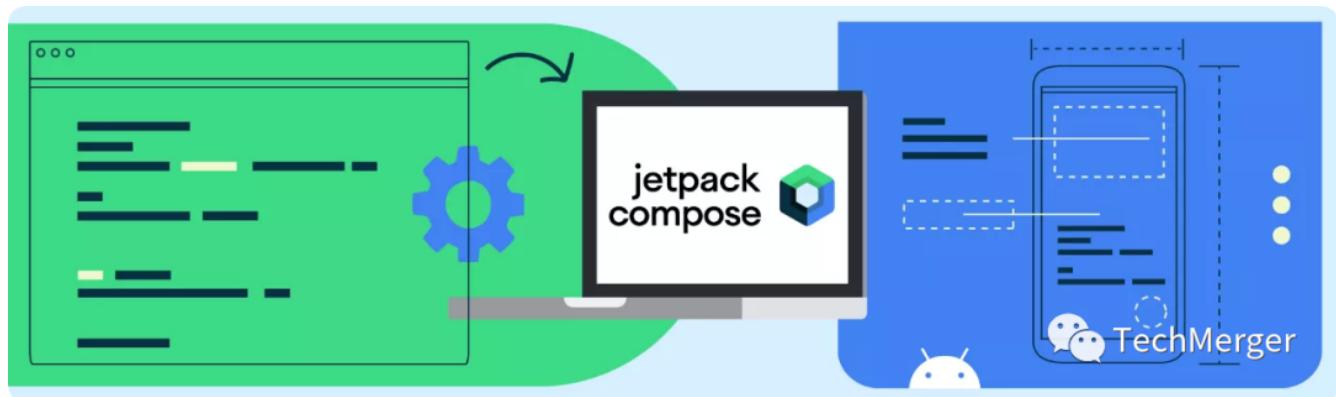
## 3.5.5 小结

到这里基本上就结束了，就这么点内容，就写出了我自认为挺好看的一个天气应用。大家如果想要代码的话直接去Github中看：

<https://github.com/zhujiang521/Weather>

## 3.6 用Compose快速打造一个“电影App”

关于Compose，后续会推送一系列的Compose文章，今天先来个实战，有兴趣的可以跑下代码，实际感受一下~



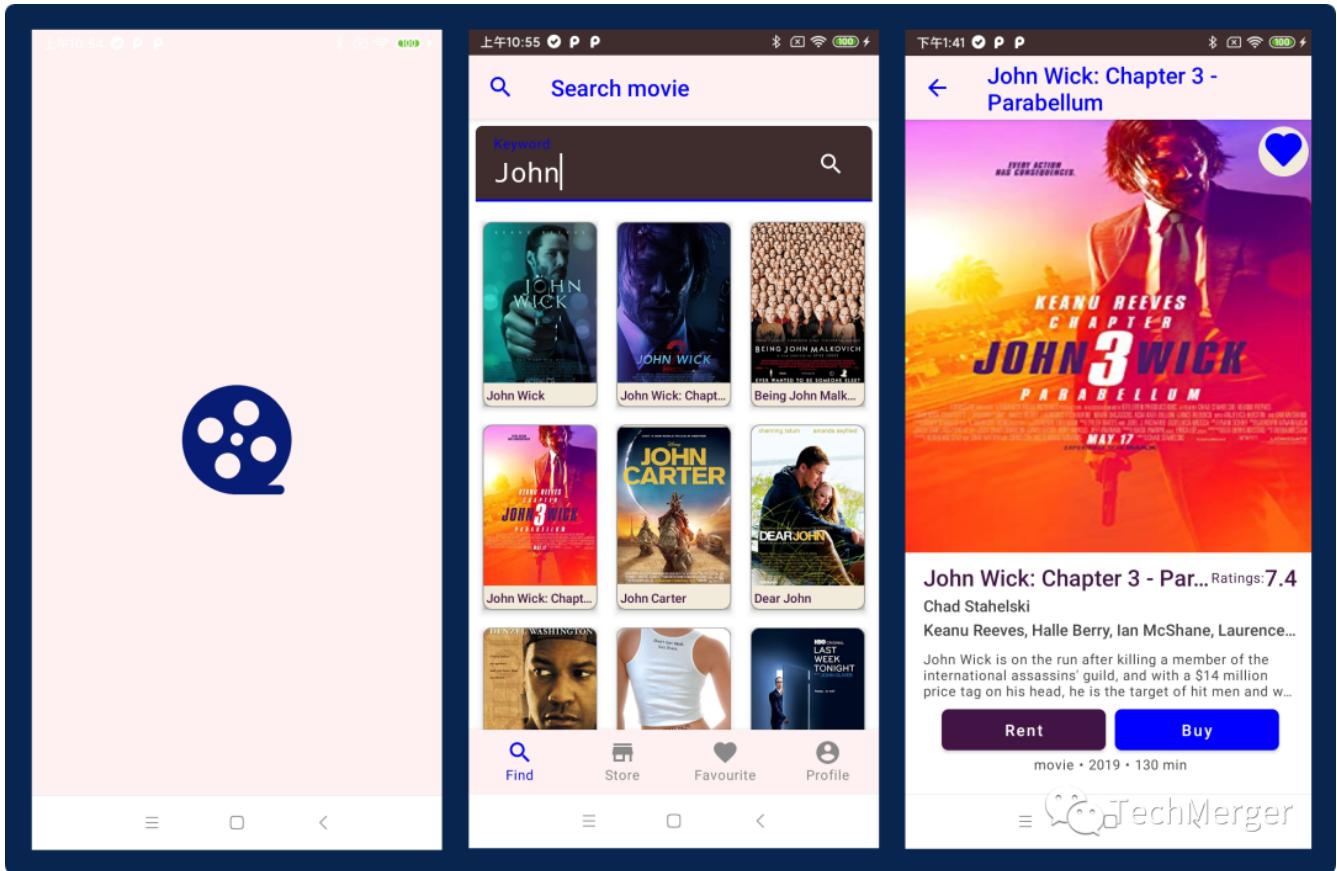
去年开源了一个电影App，其采用的是成熟的(过时的)MVP架构。而今Jetpack框架愈发火热，便萌生了完全使用Jetpack框架重新开发的想法。加上Compose Beta版的正式公开，这个时机再合适不过了。

整体上采用Compose去实现UI。数据请求则依赖Coroutines调用Retrofit接口，最后通过LiveData反映结果。

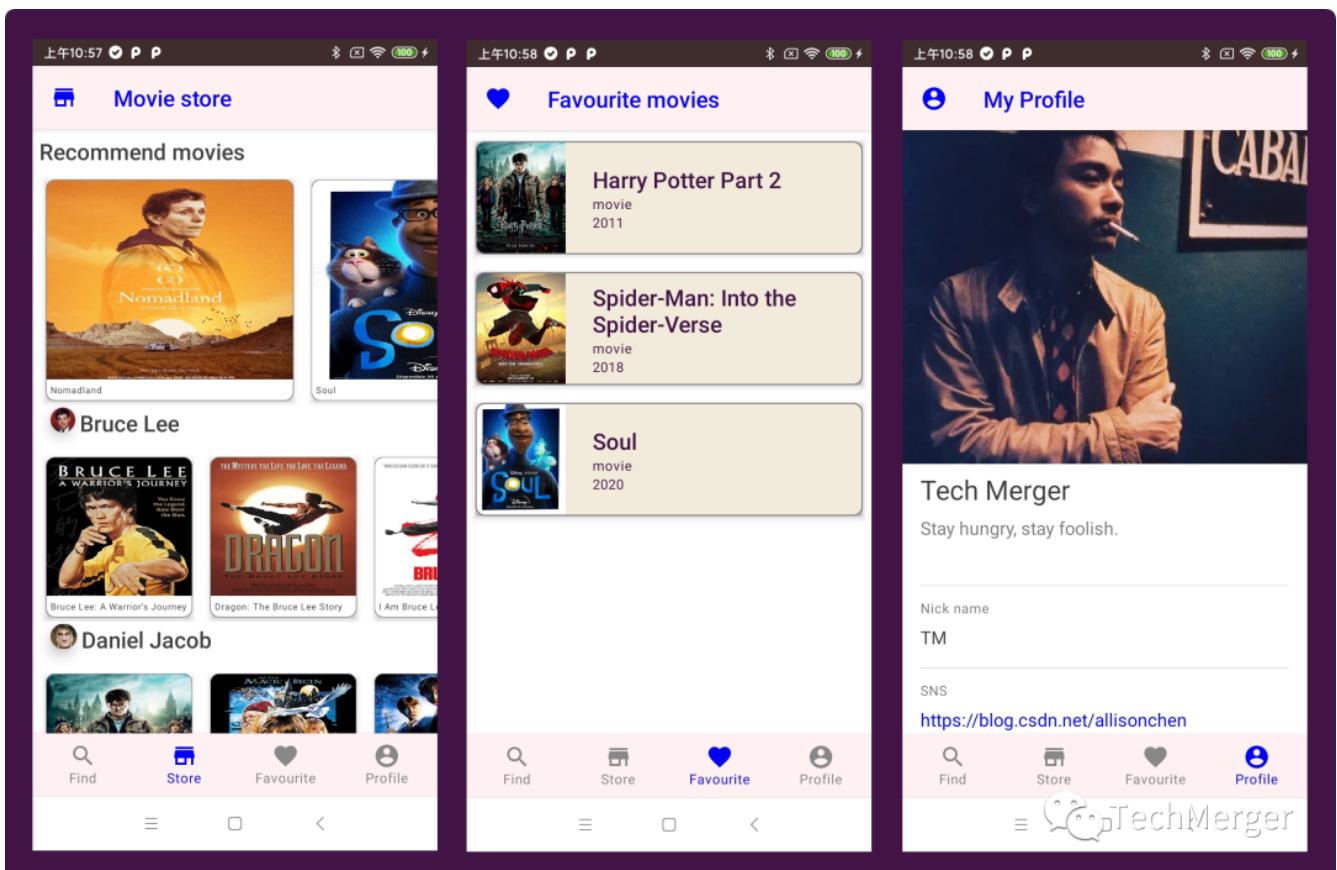
### 3.6.1 成品

话不多说，先看下效果。

启动页面，搜索页面和电影详情页面。



店铺页面，收藏页面以及和个人资料页面。



Github地址如下，欢迎参考，不吝STAR。

<https://github.com/ellisonchan/ComposeMovie>

### 3.6.2 实现方案

讲述本次的实现方案前先来回顾下之前的MVP版本是怎么做的。

功能点	技术方案
整体架构	MVP
UI	ViewPager + Fragment
View注入	ButterKnife
异步处理	RxJava
数据请求	Retrofit
图片处理	Glide

之前的做法可以说是比较成熟、比较传统的（轻喷）。

那如果采用Jetpack的Compose作为UI基盘，我会给出什么样的方案？

功能点	技术方案
整体架构	MVVM
UI	Compose
View注入	不需要
异步处理	Coroutines + LiveData
数据请求	Retrofit
图片处理	coil

### 3.6.3 实战

跟拍电影一样，脚本和选角都定了。接下来就让各单位按部就班地动起来。

#### 3.6.3.1 UI导航

整体UI采用BottomNavigation组件作为底部导航栏，将预设的几个TAB页面Compose进来。同时提供TopAppBar作为TITLE栏展示页面标题和返回导航。

```
// Navigation.kt
@Composable
fun Navigation() {
    ...
    scaffold(
        topBar = {
```

```

        TopAppBar(
            ...
        )
    },
    bottomBar = {
        if (!isCurrentMovieDetail.value) {
            BottomNavigation {
                ...
            }
        }
    }
) {
    NavHost(navController, startDestination = Screen.Find.route) {
        composable(Screen.Find.route) {
            FindScreen(navController, setTitle, movieModel)
        }
        composable(
            route = Constants.ROUTE_DETAIL,
            arguments = listOf(navArgument(Constants.ROUTE_DETAIL_KEY) {
                type = NavType.StringType
            })
        ) {
            backStackEntry ->
            DetailScreen(
                backStackEntry.arguments?.getString(Constants.ROUTE_DETAIL_KEY) !!,
                setTitle,
                movieModel
            )
        }
        composable(Screen.Store.route) {
            StoreScreen(setTitle)
        }
        composable(Screen.Favourite.route) {
            FavouriteScreen(setTitle)
        }
        composable(Screen.Profile.route) {
            ProfileScreen(setTitle)
        }
    }
}
}

```

这里有两点需要注意一下。

1、电影详情页面是从搜索页面跳转过去的，展示底部导航栏比较奇怪。所以需要声明State来控制这个页面不展示导航栏。

2、底部导航栏导航到店铺等其他页面的话会被记录在栈里，这将导致TITLE栏展示了返回按钮。对于独立的TAB页面来说没有必要提供返回操作。那同样声明State去确保这些页面不展示返回按钮。

### 3.6.3.2 搜索页面

搜索页面需要先确保网络能正常使用，并在网络不畅的情况下给出AlertDialog提醒。

UI上采用TextField提供输入区域，LaunchedEffect观察输入内容更新，自动执行搜索请求的协程。

在数据成功取得后通过LiveData反映到提供GRID列表的LazyVerticalGrid。LazyVerticalGrid组件仍然是实验性的API，随时可能删除，使用的话需要添加的@ExperimentalFoundationApi注解。

```
// Find.kt
@ExperimentalFoundationApi
@Composable
fun Find(movieModel: MovieModel, onClick: (Movie) -> Unit) {
    ...
    if (!utils.ensureNetworkAvailable(context, false))
        ShowDialog(R.string.search_dialog_tip, R.string.search_failure)

    Column {
        Row() {
            TextField(
                value = textFieldValue,
                ...
                trailingIcon = {
                    IconButton(
                        onClick = {
                            if (textFieldValue.text.length > 1) {
                                searchQuery = textFieldValue.text
                            } else Toast.makeText(
                                context,
                                warningTip,
                                Toast.LENGTH_SHORT
                            ).show()
                        }
                    )
                } {
                    Icon(Icons.Outlined.Search, "search", tint = Color.White)
                }
            ),
            ...
        }
    }
}

LaunchedEffect(searchQuery) {
    if (searchQuery.length > 0) {
        movieModel.searchMoviesComposeCoroutines(searchQuery)
    }
}
val moviesData: State<List<Movie>> = movieModel.movies.observeAsState(emptyList())
val movies = moviesData.value
val scrollState = rememberLazyListState()

LazyVerticalGrid(
    ...
) {
    items(movies) { movie ->
        MovieThumbnail(movie, onClick = { onclick(movie) })
    }
}
```

```
    }
}
```

另外Compose里的UI展示与否都依赖State的更新，网络不畅的AlertDialog亦是如此。在点击取消后仍需要依赖State触发Dialog的消失，不然它永远会在那的。

```
// Dialog.kt
@Composable
fun ShowDialog(
    title: Int,
    message: Int
) {
    val openDialog = remember { mutableStateOf(true) }

    if (openDialog.value)
        AlertDialog(
            onDismissRequest = { openDialog.value = false },
            title = {
                ...
            },
            text = {
                ...
            },
            confirmButton = {
                TextButton(onClick = { openDialog.value = false }) {
                    ...
                }
            },
            shape = shapes.large,
        )
}
```

电影海报的加载则依赖我们Compose的coil加载函数。

```
// LoadImage.kt
@Composable
fun LoadImage(
    url: String,
    contentDescription: String?,
    modifier: Modifier = Modifier,
    contentScale: ContentScale = ContentScale.Crop,
    placeholderColor: Color? = MaterialTheme.colors.compositeOnSurface(0.2f)
) {
    coilImage(
        data = url,
        modifier = modifier,
        contentDescription = contentDescription,
        contentScale = contentScale,
        fadeIn = true,
    )
}
```

```

onRequestCompleted = {
    when (it) {
        is ImageLoadState.Success -> ...
        is ImageLoadState.Error -> ...
        ImageLoadState.Loading -> Utils.logDebug(Utils.TAG_NETWORK, "Image
loading")
        ImageLoadState.Empty -> Utils.logDebug(Utils.TAG_NETWORK, "Image empty")
    }
},
loading = {
    if (placeholderColor != null) {
        Spacer(
            modifier = Modifier
                .fillMaxSize()
                .background(placeholderColor)
        )
    }
}
)
}

```

### 3.6.3.3 详情页面

电影详情页面的布局相对来说较为复杂，主要是想要展示的内容很多，简单的布局会显得臃肿，缺乏层次。

所以灵活采用了Box、Card、Column、Row及IconToggleButton这些组件来实现横纵嵌套的多层次布局。

用作展示收藏按钮的IconToggleButton和之前的AlertDialog一样，依赖State更新Toggle状态。在Compose工具包里State的概念可谓是无处不在啊。

```

// Detail.kt
@Composable
fun Detail(moviePro: MoviePro) {
    Box(
        modifier = Modifier
            .fillMaxHeight(),
    ) {
        Column(
            ...
        ) {
            Box(
                modifier = Modifier
                    .fillMaxHeight(),
                contentAlignment = Alignment.TopEnd
            ) {
                LoadImage(
                    url = moviePro.Poster,
                    modifier = Modifier
                        .fillMaxWidth()
                        .height(380.dp),
                    contentScale = ContentScale.FillBounds,
                    contentDescription = moviePro.Title
                )
            }
        }
    }
}

```

```
val checkedState = remember { mutableStateOf(false) }
Card(
    modifier = Modifier.padding(6.dp),
    shape = RoundedCornerShape(50),
    backgroundColor = likeColorBg
) {
    IconToggleButton(
        modifier = Modifier
            .padding(6.dp)
            .size(32.dp),
        checked = checkedState.value,
        onCheckedChange = {
            checkedState.value = it
        }
    ) {
        ...
    }
}

Column(modifier = Modifier.padding(horizontal = 16.dp, vertical = 8.dp)) {
    Row(
        modifier = Modifier.fillMaxWidth(),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(
            modifier = Modifier
                .weight(0.9f)
                .align(Alignment.CenterVertically),
            text = moviePro.Title,
            style = MaterialTheme.typography.h6,
            color = nameColor,
            overflow = TextOverflow.Ellipsis,
            maxLines = 1
        )
        ...
    }
    ...
}
}
```

### 3.6.3.4 店铺页面

这个页面目前是展示了推荐的电影列表和以演员分类的电影列表，称之为Store似乎不妥，暂且这样吧。

UI上采用垂直布局的Column和横向滚动的LazyRow展示嵌套的布局。需要推荐的一点是如果需要展示圆形图片，使用RoundedCornerShape可以做到。

```

// Store.kt
@Composable
fun Store() {
    Column(Modifier.verticalScroll(rememberScrollState())) {
        Spacer(Modifier.sizeIn(16.dp))
        Text(
            modifier = Modifier.padding(6.dp),
            style = MaterialTheme.typography.h6,
            text = stringResource(id = R.string.tab_store_recommend)
        )

        Spacer(Modifier.sizeIn(16.dp))
        MovieGallery(recommendedMovies, width = 220.dp, height = 190.dp)

        CastGroup(cast = testCast1)
        CastGroup(cast = testCast2)
    }
}

@Composable
fun CastGroup(cast: Cast) {
    Column {
        Spacer(Modifier.sizeIn(32.dp))
        CastCategory(cast)
        Spacer(Modifier.sizeIn(6.dp))
        MovieGallery(cast.movies)
    }
}

@Composable
fun CastCategory(cast: Cast) {
    Row(
        modifier = Modifier
            .height(40.dp)
            .padding(16.dp, 2.dp, 2.dp, 16.dp)
    ) {
        Card(
            modifier = Modifier.wrapContentSize(),
            shape = RoundedCornerShape(50),
            elevation = 8.dp
        ) {
            ...
        }
        ...
    }
}

@Composable
fun MovieGallery(movies: List<Movie>, width: Dp = 130.dp, height: Dp = 136.dp) {
    LazyRow(modifier = Modifier.padding(top = 2.dp)) {
        items(movies.size) {
            RowItem(
                ...
            )
        }
    }
}

```

```

        )
    }
}

@Composable
fun RowItem(modifier: Modifier, width: Dp = 130.dp, height: Dp = 1306.dp, movie: Movie) {
    Card(
        ...
    ) {

        Box {
            LoadImage(
                url = movie.Poster,
                modifier = Modifier
                    .width(width)
                    .height(height),
                contentScale = ContentScale.FillBounds,
                contentDescription = movie.Title
            )
            Text(
                ...
            )
        }
    }
}

```

这个页面使用Column嵌套了三个横向滚动视图，屏幕高度不够的情况下会存在显示不全的问题。自然想到了类似ScrollView的组件，一开始查到了ScrollableColumn，可是AS反复提示不存在该组件。

去官网一查，发现出于性能方面的考虑，这个组件和ScrollableRow在之前的版本被移除了。还好，官方提示可以使用Modifier.verticalScroll或LazyColumn可以达到滚动的目的。

### 3.6.3.5 收藏页面

收藏页面只展示了收藏的电影列表，最为简单。使用LazyColumn即可cover。

```

// Favourite.kt
@Composable
fun Favourite(moviePros: List<MoviePro>, onClick: () -> Unit) {
    LazyColumn(modifier = Modifier.padding(top = 2.dp)) {
        items(moviePros.size) {
            LikeItem(
                moviePro = moviePros[it],
                onClick
            )
        }
    }
}

@Composable

```

```

fun LikeItem(moviePro: MoviePro, onClick: () -> Unit) {
    Box(
        modifier = Modifier
            .wrapContentSize()
            .padding(8.dp)
    ) {
        Card(
            modifier = Modifier
                .border(1.dp, Color.Gray, shape = MaterialTheme.shapes.small)
                .shadow(4.dp),
            shape = shapes.small,
            elevation = 8.dp,
            backgroundColor = itemCardColor
        ) {
            Row(
                modifier = Modifier
                    .clickable(onClick = onClick)
                    .fillMaxWidth()
                    .height(100.dp),
                verticalAlignment = Alignment.CenterVertically
            ) {
                LoadImage(
                    url = moviePro.Poster,
                    modifier = Modifier
                        .width(80.dp)
                        .height(100.dp),
                    contentScale = ContentScale.FillBounds,
                    contentDescription = moviePro.Title
                )
                ...
            }
        }
    }
}

```

### 3.6.4.5 个人资料页面

个人资料页面需要提供封面图、名称、简介、昵称以及社交账号等信息，稍微花了些功夫。

鄙人设计天赋匮乏，部分参考了Compose示例项目Jetchat的资料页面。

需要推荐的是BoxWithConstraints组件，其可以提供类似ConstraintsLayout的效果，在指定约束规则或方向后可以动态更改其尺寸大小。

```

// Profile.kt
@Composable
fun Profile(account: Account) {
    val scrollState = rememberScrollState()

    Column(modifier = Modifier.fillMaxSize()) {
        BoxWithConstraints(modifier = Modifier.weight(1f)) {
            Surface {
                Column(

```

```

        modifier = Modifier
            .fillMaxSize()
            .verticalScroll(scrollState),
    ) {
    ProfileHeader(
        scrollState,
        this@BoxWithConstraints.maxHeight,
        account.Post
    )

    NameAndPosition(
        stringResource(id = account.FullName),
        stringResource(id = account.About)
    )

    ProfileProperty(
        stringResource(R.string.display_name),
        stringResource(id = account.NickName)
    )
    ...
    EditProfile()
}
}
}
}

@Composable
fun ProfileProperty(label: String, value: String, isLink: Boolean = false) {
    Column(modifier = Modifier.padding(start = 16.dp, end = 16.dp, bottom = 16.dp)) {
        Divider()
        CompositionLocalProvider(LocalContentAlpha provides ContentAlpha.medium) {
            Text(
                text = label,
                modifier = Modifier.paddingFromBaseline(24.dp),
                style = MaterialTheme.typography.caption
            )
        }
        val style = if (isLink) {
            MaterialTheme.typography.body1.copy(color = color.Blue)
        } else {
            MaterialTheme.typography.body1
        }
        ...
    }
}
}

```

App大部分的实现细节都讲完了，代码量很小。除了本身功能相对轻量以外，Compose工具包的简洁易用绝对功不可没。

### 3.6.4 不足

我们再来谈谈这个App还存在什么可以改善的地方，包括UI交互上的、功能上的等等。

#### 1.不支持中文关键字搜索

App采用的数据来源是国外的OMDB，它的电影库还是健全的，提供的电影相关内容也足够丰富。可其出生地也决定了它只擅长英文关键字的查询，使用其他语言比如中文、日文，几乎是查不到任何电影的。

为了完善中文方面的功能，亟需导入华语电影的接口。奈何没有找到，之前使用良好的豆瓣API已经废弃了。

了解的朋友可以教育一下我，感谢。

#### 2.UI设计风格需要强化

目前整体UI的设计采用米色做背景，蓝色做高亮，辅助以浅灰色、白色以及紫色作其他内容的展示。给人感觉还是有点设计的，但总有种说不出的乱，无法沉浸进去。不知道屏幕面前的你有没有一样的感受？

后面计划针对Material设计语言做个深度地学习和理解，并能将其计理念完美地融入到Compose中来。（好的，说人话。过段日子我将观摩几个不错的电影App，比如Netflix、Disney+啥的，好好地模仿一番。）

#### 3.搜索页面TITLE栏有点多余

为使搜索页面和其他页面提供风格一致，提供了展示搜索图标的TITLE栏。对于用户来说，这和下面输入框的功能有些重叠，而且会占用电影列表的显示区域。

所以完全可以将这个页面的TITLE栏删除，直接提供输入框即可。

上午10:55 ✓ P P

Bluetooth Wi-Fi 100%



Search movie

Keyword

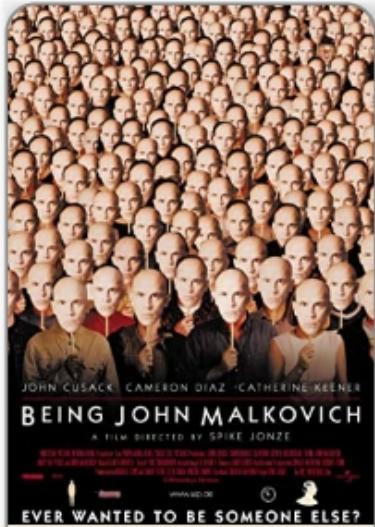
John



John Wick



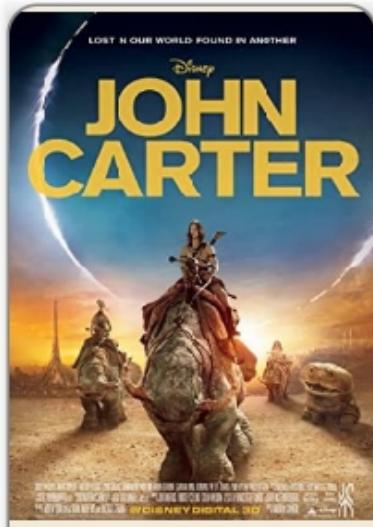
John Wick: Chapt...



Being John Malk...



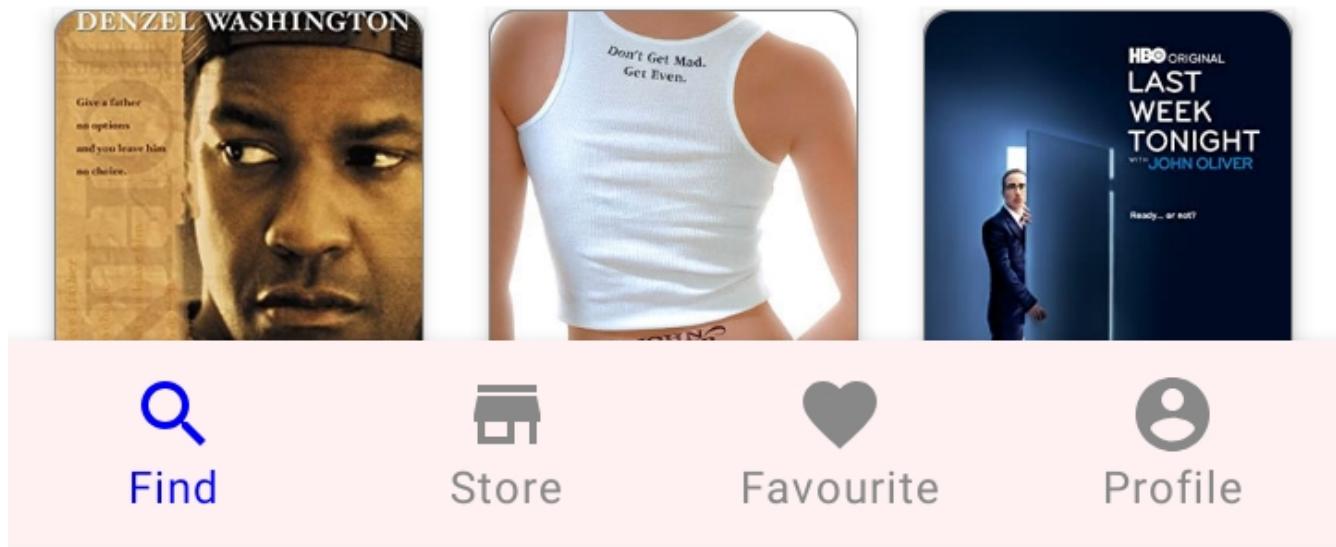
John Wick: Chapt...



John Carter



Dear John



#### 4. 搜索之后IME可以自动隐藏

点击搜索按钮之后IME面板不会自动隐藏，体验不是太好。点击搜索或搜索完毕之后自动将IME隐藏可能体验更佳。

简单查了下资料，似乎是利用TextInputService去实现，捣鼓了半小时还没实现，暂时搁置了。知道的朋友可以告知下，比心。

#### 5. 店铺页面需要强化推荐

首先啊，这个页面名称可能需要更改，改为Home主页是不是更好些。“家”才比较懂你，给你一些精准的建议。

OMDB没有提供推荐电影的接口，所以目前的推荐列表的数据是模拟的。后面可能需要记录并分析用户搜索的关键字、点击的电影类型、关注的电影导演及演员等数据，得出一套智能的推荐结果。最终按照类型、导演、演员等维度呈现出来。

到时候使用Room框架配合一套算法开干。

#### 6. 收藏和资料数据需持久化

目前收藏的电影数据没有持久化到本地，资料页面也没提供编辑入口。后面需要通过Room或DataStore框架提供数据的支撑。

当然，屏幕前的你觉得还有什么其他不足可以不吝赐教，我必洗耳恭听。

### 3.6.5 结语

文思如泉涌，一口气码了这么多字，最后还想再分享些切实感受。

#### 1. Compose版本和MVP版本的对比？

Compose版本的代码精简得多，声明式UI的编程方式也饶有新意，其侧重于声明和状态的编程思想无处不在。其与Jetpack框架、Material主题的无缝衔接让习惯了XML布局方式的开发者亦能快速入门。

Compose工具包也并非完美，其在性能方面的表现也令我有些怀疑。而且各大公司、各个产品对于这个新生技术的态度眼下也无从保证。

MVP架构庞杂的接口令人诟病，也并非一无是处。结合产品的定位和需求，辩证地看待这两种方式。

## 2、Compose在使用上有无痛点？

日志匮乏：看不到debug和error级别的任何日志，很难把控流程和定位问题。

原理学习困难：UI和逻辑的包众多、讲解原理的文章极度匮乏（希望日后我能贡献一份力）

## 3、面对Android新技术的层出不穷到底要采取什么姿态？

把头埋进土里无视是肯定不行的，需时刻保持关注并做一定的尝试。

不要把简单便捷的编码当成全部，需认识到背后的框架和编译器默默地做了很多工作。

不要执迷于框架、依赖于框架，了解并掌握其原理，在坑来临时游刃有余。

## 3.6.6 DEMO

上面只阐述了些关键的细节，需要的话还得参考完整代码。

<https://github.com/ellisonchan/ComposeMovie>

## 致谢

书稿终告断落，掩卷思量，饮水思源，在此谨表达自身的殷切期许与拳拳谢意。非常感谢各位读者耐心的看到本身的最后部分。

首先，第三章实战部分几乎已经包含全网Jetpack compose的大部分应用内容，本手册的第三章通过多个Android技术开发分享项目实例，从不同的角度引领读者亲身实战，真正地掌握Jetpack compose编程的核心开发技巧。但是，实例的数量终究有限，希望读者更多地关注于实战中的开发思想，而不是具体的代码逻辑，代码总会不断地更迭，解决问题的思维却历久弥新。本手册中的实例更多的是以点带面，读者可以一边阅读和思考，一边编写代码，相信读完本手册，一定受益匪浅；同时，通过本手册的实例可以解决一些常见的开发需求。

衷心希望每位读者在阅读完本手册之后，都“不虚此行”！在手册籍撰写过程中，引用了Google开发者、依然范特稀西、郭霖、HeartCircle、Zhujiang、小虾米君、等博主倾情无私分享的技术讲解和项目实战案例。同时也感谢其他博主的在Jetpack compose分享的学习心得笔记。在本手册引用的文章和作者这里都一一表示感谢。

最后，得感谢本手册得以付梓的幕后英雄腾讯享学技术团队，包括关键技术资料和项目实战案例搜集整理，谢谢您们！对所引用的书籍表示感谢。

## 参考资料

作者：依然范特稀西

原文链接：<https://juejin.cn/post/6844904165408243725>

作者：HeartCircle 原文链接：<https://blog.csdn.net/HeartCircle/article/details/115110096>

作者：Zhujiang

原文链接：<https://juejin.cn/user/3913917127985240/posts>

作者：**小虾米君**

公众号：**TechMerger**。

腾讯享学课堂 <https://ke.qq.com/course/341933?taid=11279327148980141&tuin=9f978cfe>