第 5 章　Android 用户界面

Android 用户界面是应用程序开发的重要组成部分，决定了应用程序是否美观、易用。通过本章的学习可以让读者掌握基于 Jetpack Compose 的用户界面开发方法，了解在界面开发过程中常见的基础 UI 组件、布局组件、高级组件和导航组件的使用方法。

本章学习目标：

- 了解 UI 开发的基本概念。
- 掌握 Modifier 修饰符的使用方法。
- 掌握基础 UI 组件、布局组件和高级组件的使用方法。

5.1　Jetpack Compose 概述

Jetpack Compose(见图 5.1)是谷歌公司推出的现代化 Android UI 构建工具包，采用声明式编程范式，极大地简化了用户界面开发流程。Compose 是 Jetpack 组件家族的一员，旨在取代传统的基于 XML 的 UI 描述方式，使开发者可以使用 Kotlin 语言，通过函数的方式直接构建界面。

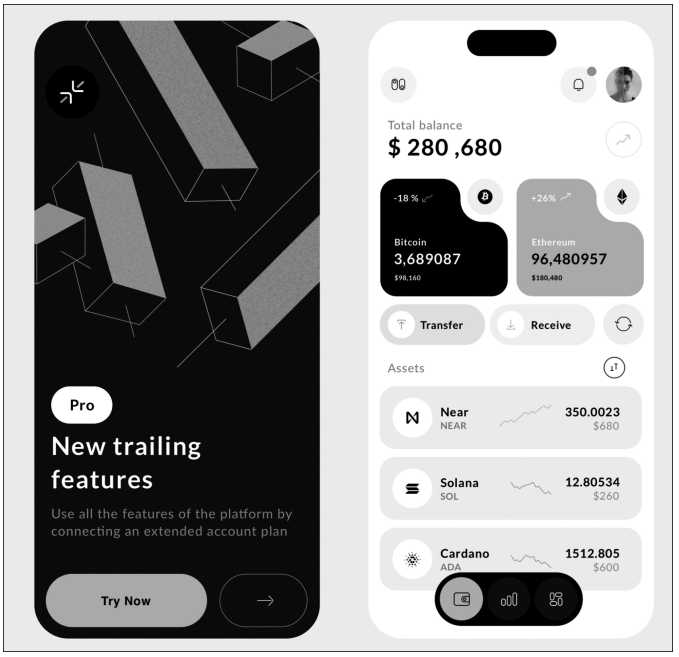

图 5.1　Jetpack Compose UI 框架

Jetpack Compose(简称 Compose)在性能方面展现出明显优势。它通过跳过不必要的 UI 重绘，大幅减少渲染开销，从而保证界面流畅。Compose 在底层充分利用 Kotlin 编译器

优化,支持按需更新界面元素,避免传统 View 层级过深带来的性能瓶颈。相比基于 XML 的方式,它在内存占用和渲染速度上更高效,同时具备更佳的响应能力。

Jetpack Compose 受到了 React、Flutter 等框架的影响,强调以状态驱动界面更新,并具备更高的可组合性和可读性。开发者可以通过 Composable 函数定义 UI 组件,界面更新逻辑更加清晰简洁。谷歌公司在 Compose 的设计中,深度集成了 Kotlin 的语言特性,如扩展函数、高阶函数和协程,从而让界面开发更加灵活高效。

Jetpack Compose 经过一系列测试版本后,于 2021 年 7 月发布了 1.0 稳定版,此后,谷歌公司持续优化 Compose 的性能与功能,陆续引入了 Material 3 支持、多平台适配(JetBrains Compose Multiplatform)、自定义布局等能力,逐渐形成了完整的生态体系。

Compose 的推出标志着 Android UI 开发从命令式向声明式的重大转变,它不仅提高了代码的可维护性,也加快了界面的开发效率。如今,Jetpack Compose 已成为官方重点推荐的 UI 框架,广泛应用于新一代 Android 应用开发中。

Jetpack 是 Google 提供的一套 Android 开发组件库集合,旨在简化应用开发,提升开发效率与代码质量。它涵盖了架构、UI、行为、基础功能等多个方面,包括常用的 Room、ViewModel、LiveData、Navigation 和 WorkManager 等组件。Compose 与 Jetpack 组件无缝对接,提供现代化的 Android 应用开发体验。

Jetpack Compose 改变了 Android UI 的编程方式,使界面构建更自然、响应更灵活、开发效率更高,代表了 Android UI 开发的未来方向。

5.2 基本概念

5.2.1 Composable 函数

Jetpack Compose 中的所有 UI 都是由可组合函数(Composable Functions)构成的,这些函数是 Compose 编程模型的核心,负责描述界面的结构与内容。通过组合这些函数,可以灵活地构建复杂的用户界面。

所谓 Composable 函数,是指使用@Composable 注解标记的 Kotlin 函数。它们可以直接输出 UI 元素,也可以组合其他 Composable 函数构建更复杂的 UI。Compose 会在运行时根据界面状态的变化,自动调用这些函数进行界面重组。

基本语法如下:

```
1   @Composable
2   fun Greeting(name: String) {
3       Text(text = "Hello, $name!")
4   }
```

第 1 行代码的@Composable 注解,表示函数 Greeting()是一个 Composable 函数。第 3 行代码的 Text 是一个显示文本的基础 UI 组件,显示的内容是“Hello, $name!”,其中 name 是函数 Greeting()传递的参数。

Android Studio 提供了对 Composable 组件的预览功能,需要再写一个 Composable 函数,在这个新的函数中调用 Greeting()函数。

```
1   @Preview
2   @Composable
3   fun DefaultPreview(){
4       Greeting("Android")
5   }
```

第 1 行代码增加@Preview 注解，用于在 Android Studio 中预览 Composable 函数的 UI，无须在模拟器中运行应用，就可以看到 UI 的显示效果，而且可设置如主题、设备尺寸等参数，帮助开发者快速调试和设计界面。

第 4 行代码调用了 Greeting()函数，name 的值为 Android，这样在 Android Studio 中就可以看到预览效果，如图 5.2 所示。

图 5.2　预览效果

@Preview 注解还可以指定预览时使用不同参数设置分辨率和显示效果，具体的用途说明和示例参考表 5.1。

表 5.1　@Preview 注解参数说明

用 途 说 明	示 例 代 码
设置预览名称	@Preview(name = "Light Mode Preview")
显示背景色	@Preview(showBackground = true)
自定义背景颜色	@Preview(showBackground = true, backgroundColor = 0xFFE0E0E0)
设置固定预览尺寸	@Preview(widthDp = 360, heightDp = 640)
夜间模式预览	@Preview(uiMode = Configuration.UI_MODE_NIGHT_YES, showBackground = true)
模拟特定设备(如 Pixel 4)	@Preview(device = "spec:shape=Normal,width=411,height=891,unit=dp") 或 @Preview(device = "pixel_4")

将这些注解组合使用，以覆盖更多真实设备和环境下的 UI 效果。

5.2.2　重组机制

在 Jetpack Compose 中，“重组”指当界面数据发生变化时，自动重新计算和更新 UI 的过程。它就像一个智能的“更新机制”，只会重新绘制那些变化的部分，而不是整个界面。这种方式与传统的 UI 更新不同，传统的 UI 更新通常需要重新绘制整个屏幕。

重组是由“状态”变化触发的，当界面上的数据或状态变化时，Compose 会自动检查哪些 UI 元素需要更新。只更新实际变化的部分，而保持其他部分不变。

下面是 Compose 重组示例，这个例子用来展示 Compose 如何自动进行局部重组：

```
1   @Composable
2   fun CounterExample() {
```

```
3       var count by remember { mutableStateOf(0) }
4
5       Column {
6           Text(text = "当前计数:$count")
7           Button(onClick = { count++ }) {
8               Text("增加")
9           }
10      }
11  }
12
```

这段代码现在看起来还是有些难懂,读者可以只关注第 6 行和第 7 行代码,分别创建了文本组件和按钮组件,如图 5.3 所示。

图 5.3　创建文本组件和按钮组件

当单击按钮时,count 发生变化,因为文本组件显示的内容依赖于 count,因此文本组件会被更新。而按钮组件的显示与 count 无关,因此不会参与 UI 更新。

重组的意义在于提升界面更新的效率与性能。传统 UI 框架在状态变化时常常需要手动刷新整个界面或控件,容易导致冗余渲染和性能浪费。而 Compose 通过重组机制,能够根据状态的变化,仅更新界面中受影响的部分,避免不必要的计算和重绘,这样既能保持 UI 的一致性,又能提升应用的响应速度和流畅性。对于开发者来说,重组还简化了状态管理逻辑,让界面编写更加直观,是现代 UI 编程的重要特性之一。

5.2.3　状态管理

在 Jetpack Compose 中,状态(State)是指那些会影响 UI 显示的数据,并且这些数据一旦发生变化,系统就会自动触发对应 Composable 函数的重组,以更新界面显示,可以简单理解为:状态是驱动 Compose UI 的源头。

举个例子说明什么是“状态”:

```
1   @Composable
2   fun CounterRemember() {
3       var count by remember { mutableStateOf(0) }
4       Button(onClick = { count++ }) {
5           Text(text = "单击次数: $count")
6       }
7   }
```

在这个代码中,第 3 行代码的 count 就是一个状态变量。在第 4 行代码中,当用户单击按钮时,count 会累加 1,因此改变了 count 状态。因为第 5 行代码的 Text 的显示内容依赖于 count 这个状态,所以 count 状态的改变会导致 Compose 重组,更新 UI 中的数值,如图 5.4 所示。

图 5.4　预览 CounterRemember()

仔细看这个代码:

```
1  var count by remember { mutableStateOf(0) }
```

mutableStateOf(0)创建一个可变的状态对象，初始值为 0。remember { ... }用于在 Composable 中“记住”这个状态，只在第一次进入组合时创建，避免在每次重组时重新初始化。by 是 Kotlin 的属性委托语法，可以直接通过 count 来访问和修改状态值，而不需要显式调用 get()或 set()方法。

1. mutableStateOf

mutableStateOf 是 Compose 中统一的状态包装器，可以广泛地用于多种类型的状态管理。不仅可以用于 Int，还可以创建任何类型的可变状态，只要这个类型在 Compose 中是可观察的，mutableStateOf 常见用法如表 5.2 所示。

表 5.2　mutableStateOf 常见用法

状 态 类 型	示 例 代 码
整数状态	val count = mutableStateOf(0)
字符串状态	val text = mutableStateOf("Hello")
布尔状态	val isVisible = mutableStateOf(true)
列表状态	val items = mutableStateOf(listOf("A", "B", "C"))
自定义数据类	val user = mutableStateOf(User("Tom", 18))
可空类型	val name = mutableStateOf<String?>(null)

2. remember { ... }

remember{ ... }是用于在组合过程中存储状态的关键 API，它的作用是在 Composable 函数中记住某个值，防止在每次重组时被重新计算或创建。

Compose 是一个响应式、声明式的 UI 框架，它的 Composable 函数可能会因为状态变化被频繁调用。为了避免每次重组都重新执行某些初始化操作，remember{...}会在首次执行时保存值，并在后续重组中复用先前保存的结果，且它在组合期间的生命周期中保持稳定。

需要注意的是，remember{...}只能在 Composable 函数内部调用，remember{...}的值只会在当前组合期间保存，如果 Composable 离开组合（如页面切换、旋转屏幕）就会被销毁。

如果需要在配置变更（如旋转屏幕）后依然能恢复数据，可以使用 rememberSaveable。

```
1   @Composable
2   fun CounterSave() {
3       var count by rememberSaveable { mutableStateOf(0) }
4       Button(onClick = { count++ }) {
5           Text(text = "单击次数: $count")
6       }
7   }
```

这个新的 Counter 只有第 3 行代码不同，将原来的 remember 修改成 rememberSaveable。示例需要启动 Android 模拟器进行测试。先单击按钮增加单击次数，然后旋转模拟器屏幕方向，查看“单击次数”是否保持不变。

3. By

by 是一个委托关键字，用于将某个属性的读写操作委托给另一个对象来实现，这种机

制称为“属性委托”(Property Delegation)。

by 关键字在 Compose 中的常见用途就是简化状态变量的读写语法,让代码更自然、接近普通变量的使用方式。

```
1   val state = remember { mutableStateOf(0) }
2   var count = state.value
3   state.value = 1                                    //这样写要手动处理
```

上面的代码和下面的代码等效:

```
1   var count by remember { mutableStateOf(0) }
2   count = 1
```

使用 by 后,就可以像操作普通变量一样使用 count,其实背后会调用 state.value,使代码更简洁、清晰。

5.2.4 状态提升

在 Compose 中,为了保持组件的可组合性与可重用性,通常不推荐在 Composable 函数内部直接管理状态,而是将状态作为参数传入,通过回调函数通知外部更新,这一模式称为状态提升(State Hoisting)。

通俗地讲就是:把状态管理从子组件“提升”到父组件中,由父组件来控制状态,子组件只负责展示和通知变化。

首先看一下没有进行状态提升的代码:

```
1   @Composable
2   fun Counter() {
3       var count by remember { mutableStateOf(0) }
4       Button(onClick = { count++ }) {
5           Text("Count: $count")
6       }
7   }
8
9   @Composable
10  fun CounterScreen() {
11      Counter()
12  }
13
```

这里的子组件 Counter 管理状态 count,父组件 CounterScreen 没有管理任何状态。如果将 count 的状态管理从子组件 Counter“提升”到父组件 CounterScreen 中,代码如下:

```
1   @Composable
2   fun Counter(count: Int, onIncrement: () -> Unit) {
3       Button(onClick = onIncrement) {
4           Text("Count: $count")
5       }
6   }
7
8   @Composable
9   fun CounterScreen() {
```

```
    var count by remember { mutableStateOf(0) }
    Counter(count = count, onIncrement = { count++ })
}
```

Counter 不再自己管理状态，而是通过参数 count 和 onIncrement 实现展示与事件处理，CounterScreen 管理状态并传递给 Counter。

之所以要做状态提升，就是把状态交给更高层的组件管理，让子组件专注于展示与反馈，写出更清晰、解耦、可重用的 UI。

这里举个通俗易懂的例子：孩子（子组件）玩玩具（显示 UI），原来是自己保管玩具（自己管理状态），但现在玩具太贵了（状态要被多个组件共享或保存），于是家长（父组件）来保管玩具（状态），孩子只管玩和告诉家长“我要换玩具”（通过回调通知改变状态）。

这种方式遵循了单向数据流（Unidirectional Data Flow，UDF）的设计理念。单向数据流是一种设计模式，指的是数据在应用中沿着单一方向流动：状态（State）→驱动 UI→用户操作（Event）→更新状态→再次驱动 UI。

数据流动路径如下。

（1）State（状态）。由上层组件提供。

（2）UI（界面）。由状态生成，Compose 会根据状态变化自动重组 UI。

（3）Event（事件）。用户与 UI 交互，触发事件（如单击按钮）。

（4）State 更新。事件处理函数更新状态。

（5）新的 UI 渲染。状态变化，Compose 重新绘制 UI。

单向数据流的核心优势在于数据流动路径清晰、可控，极大地简化了界面状态的管理。当状态始终从上层传递给下层组件，并且所有状态更新通过统一的事件回传处理，就避免了“状态不同步”或“状态来源不明”的问题。

这种模式让 UI 渲染成为状态的纯函数，提高了组件的可预测性与可测试性。同时，它也促进了业务逻辑与界面显示的分离，降低了组件之间的耦合，增强了代码的复用性和维护性。

5.2.5　Composable 的生命周期

Jetpack Compose 是响应式 UI 框架，是以“状态变化”为核心的生命周期来驱动 UI 渲染。生命周期是状态驱动、事件触发的“循环过程”，而非一次性的线性流程。

Composable 在生命周期中有三种状态：组合（Composition）、重组（Recomposition）和卸载（Disposal）。

1. 组合（可以多次执行）

每次组件重新出现在界面中都会重新组合。例如，用户从首页进入详情页再返回首页，这时候首页的 Composables 又会重新被“组合”一次。

2. 重组（频繁被执行）

只要状态变了，就会触发重组，次数可能很多。每次状态变动（如单击按钮、输入文字等）都会触发对应 Composable 的“重组”，这是 Compose 最核心的机制。

每点一次按钮，就会触发 Text 重组示例：

```
1   val count = remember { mutableStateOf(0) }
2   Button(onClick = { count.value++ }) {
3       Text("单击次数:${count.value}")
4   }
```

3. 卸载(每次消失就会被执行)

当组件被从界面中移除(或替换)时就会执行卸载,可能发生多次。

多次显示/隐藏 MyDialog 组件,就会多次触发卸载→组合→卸载→...示例:

```
1   if (showDialog) {
2       MyDialog()                          //当 showDialog 变为 false,这个组件会被卸载
3   }
```

5.2.6 副作用函数

在 Jetpack Compose 中,界面是通过函数方式"声明"出来的。这意味着 Compose 会频繁地调用@Composable 函数来"重组"界面。当状态变化时,Compose 会重新执行 UI 构建逻辑,以更新界面内容。

非 UI 操作之所以不能直接写在 UI 逻辑中,是因为 Compose 的重组机制会导致它们被多次执行,从而破坏预期的行为或引发性能问题。

为什么非 UI 操作不能直接放在 UI 逻辑中示例:

```
1   @Composable
2   fun MyScreen() {
3       //这是危险的做法:协程会在每次重组时都重新启动
4       CoroutineScope(Dispatchers.IO).launch {
5           fetchData()
6       }
7   }
```

在 UI 逻辑中直接启动协程是非常危险的做法,因为重组会导致多次执行这部分代码,从而导致代码异常或性能问题。

应该使用以下代码:

```
1   @Composable
2   fun MyScreen() {
3       LaunchedEffect(Unit) {                    //这样是正确的,只启动一次
4           fetchData()
5       }
6   }
```

这里的 LaunchedEffect 是副作用函数的一种,可以保证在一个 Composable 生命周期中只执行一次,只有这样才是安全的做法。

副作用函数(Side-effect APIs)是用来执行那些不能直接放进 UI 构建逻辑里的"额外操作",例如启动协程、注册监听器或者更新外部状态等。这些函数会跟随 Composable 的生命周期自动启动和清理,不用手动控制,常见的副作用函数如表 5.3 所示。

副作用函数依赖于组合函数的生命周期,用于确保某些只能执行一次或需在特定时机执行的操作不会在重组时被重复触发。它们根据生命周期的变化自动启动、更新或清理副

作用，从而避免重复执行、内存泄漏或逻辑错误，使界面行为与状态保持一致，副作用函数和生命周期的关系如表 5.4 所示。

表 5.3　常见的副作用函数

副作用函数	用　途	生命周期关系
LaunchedEffect	启动协程，在组合时或键变化时执行	绑定组合，自动取消
rememberCoroutineScope	获取可复用的协程作用域，配合事件用	不会自动取消，需要手动管理
SideEffect	每次成功重组后执行	依附于组合，重组时触发
DisposableEffect	注册/解绑资源，如监听器、回调	组合时执行，卸载时清理
derivedStateOf	派生状态，防止无效重组	与 remember 一起用，随组合
rememberUpdatedState	保存更新值，避免闭包引用旧值	组合时更新，用于协程或监听器
snapshotFlow	将 Compose 状态转换成 Flow	用于协程，跟随状态更新

表 5.4　副作用函数和生命周期的关系

生命周期阶段	可用副作用函数	说　明
组合	LaunchedEffect，DisposableEffect，remember，SideEffect	初始化操作、监听器注册、状态派生
重组	SideEffect，rememberUpdatedState	重组后同步外部状态、更新闭包引用值
卸载	DisposableEffect 的 onDispose	在移除时自动释放资源（如注销回调）

1. LaunchedEffect

LaunchedEffect 用于启动协程并绑定“组合”生命周期的副作用函数，它只会在指定的 key 变化时或第一次进入“组合”时执行一次，即使组合函数多次重组也不会重复执行协程逻辑，适合用来执行一次性的逻辑，例如加载数据、发起网络请求、播放动画、延时跳转等。

启动协程加载数据示例：

```
1   @Composable
2   fun UserProfile(userId: String) {
3       var userName by remember { mutableStateOf("加载中...") }
4
5       LaunchedEffect(userId) {
6           //当 userId 变化时重新执行此协程
7           userName = loadUserName(userId)
8       }
9
10      Text(text = "用户名:$userName")
11  }
12
13  //模拟网络请求
14  suspend fun loadUserName(id: String): String {
15      delay(1000)
16      return "用户$id"
17  }
```

第 14 行代码的 suspend 关键字用于声明挂起函数，表示这个函数可以在协程中挂起执行，不会阻塞主线程。

第 15 行代码使用 delay(1000)模拟网络延迟，不会阻塞线程，只是“挂起”1s。

第 5 行代码的 LaunchedEffect(userId)用于启动协程,并在第一次进入组合时执行协程逻辑。同时,因为设定了 key 的值 userId,因此在 userId 改变后会再次执行协程逻辑。

在 Compose 中更新界面必须运行在主线程,但网络请求、读取文件等耗时任务不能阻塞主线程,否则界面会卡顿甚至失去响应。使用 suspend 关键字配合 LaunchedEffect 可以在后台执行这些操作,并在完成后安全地更新界面状态。

定时器示例:

```
1   @Composable
2   fun TimerLoggerComposable() {
3       LaunchedEffect(Unit) {
4           while (true) {
5               delay(1000)
6               Log.d("TimerLogger", "定时任务触发")
7           }
8       }
9
10      Text(text = "定时任务已启动,每秒触发一次")
11  }
```

第 3 行代码用于启动协程,因为 key 设置为 Unit,则只在第一次进入组合时执行一次。第 4～6 行代码是无限循环,每 1000ms,在 Logcat 中打印一次数据。

LaunchedEffect 用于自动绑定到 Composable 生命周期,退出界面就会自动停止任务,不会造成内存泄漏。

2. DisposableEffect

DisposableEffect 可以用于有初始化和清理需求的副作用操作,它会在指定的 key 第一次出现时执行副作用逻辑,并在组合退出或 key 变化时自动调用清理操作,适合用于注册监听器、订阅回调、启动计时器、添加生命周期观察者等场景,这些操作通常需要在不使用时取消或释放资源。

进入页面时打印“注册资源”,退出页面时打印“释放资源”示例:

```
1   @Composable
2   fun DisposableScreen(){
3       var show by remember { mutableStateOf(true) }
4
5       Column {
6           Button(onClick = { show = !show }) {
7               Text("切换组件")
8           }
9
10          if (show) {
11              SimpleDisposableEffect()
12          }
13      }
14
15  }
16
17  @Composable
18  fun SimpleDisposableEffect() {
19      DisposableEffect(Unit) {
```