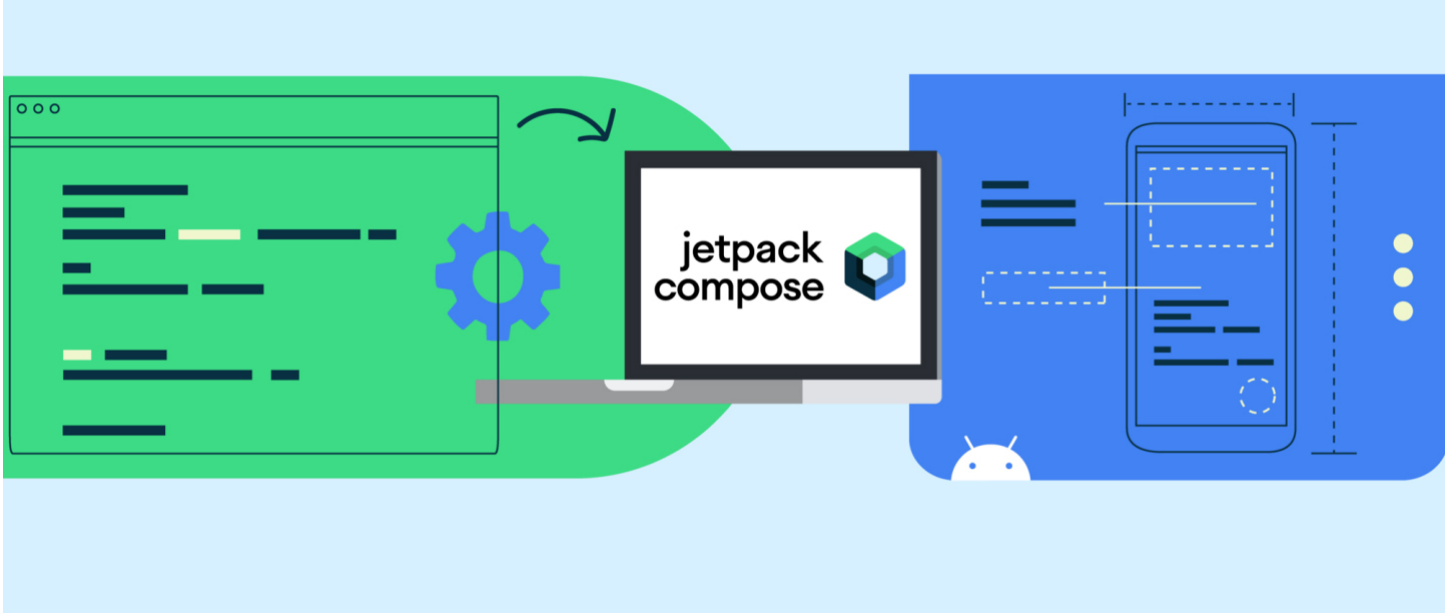


深入详解 Jetpack Compose | 优化 UI 构建

谷歌开发者
已认证的官方帐号

已关注

16 人赞同了该文章

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。

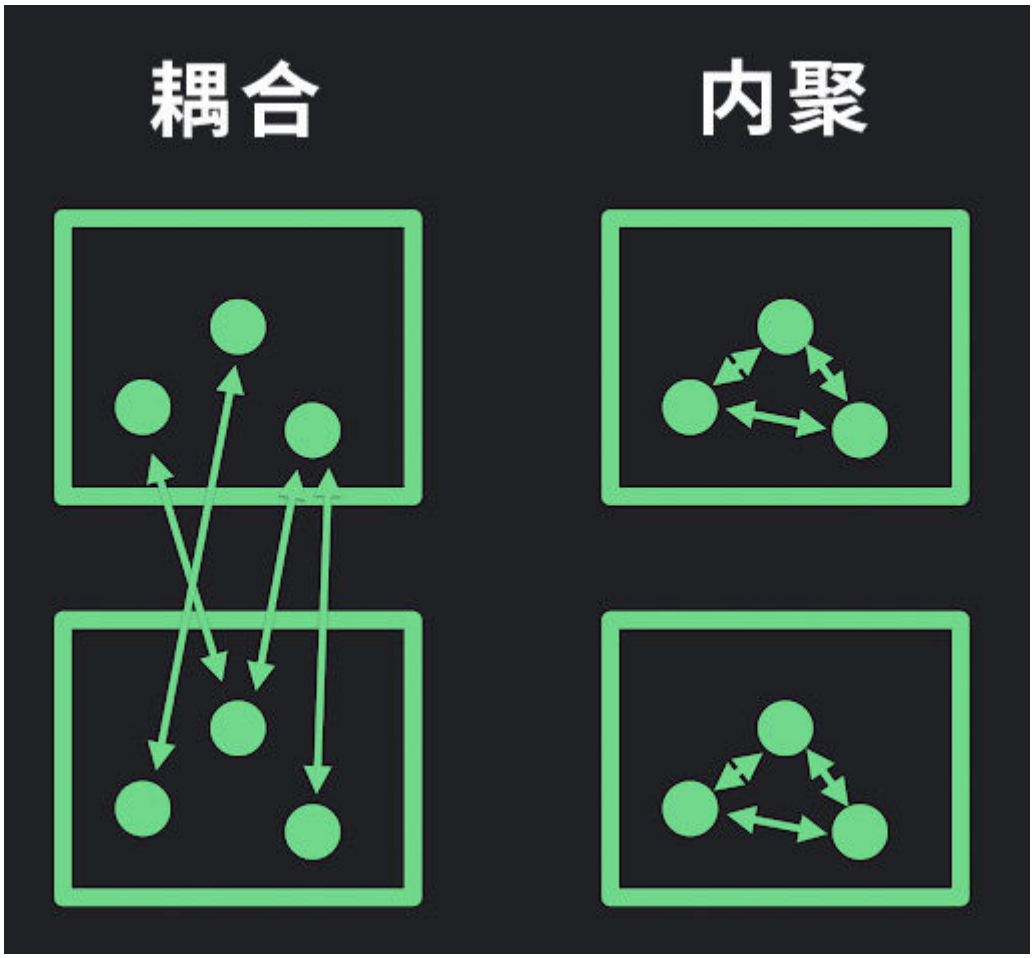
在本系列的两篇文章中，我们将阐述 Compose 的优势，并探讨它背后的工作原理。作为开篇，在本文中，我会分享 Compose 所解决的问题、一些设计决策背后的原因，以及这些决策如何帮助开发者。此外，我还会分享 Compose 的思维模型，您应如何考虑在 Compose 中编写代码，以及如何创建您自己的 API。

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。

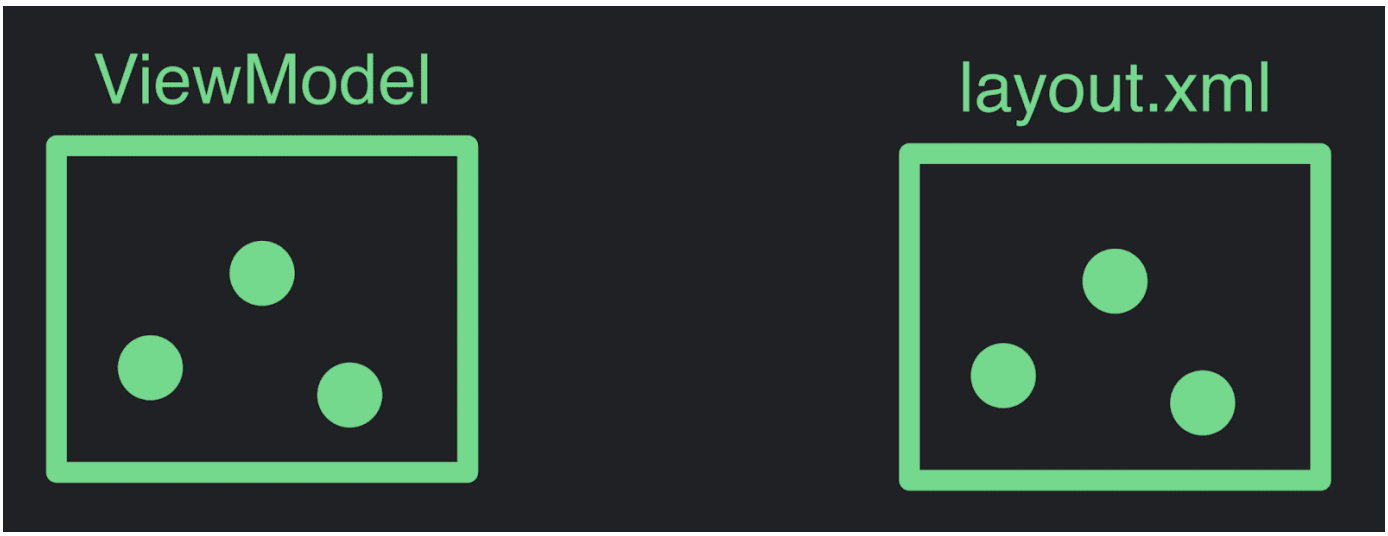
在编写可维护的软件时，我们的目标是最大程度地**减少耦合**并**增加内聚**。



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。

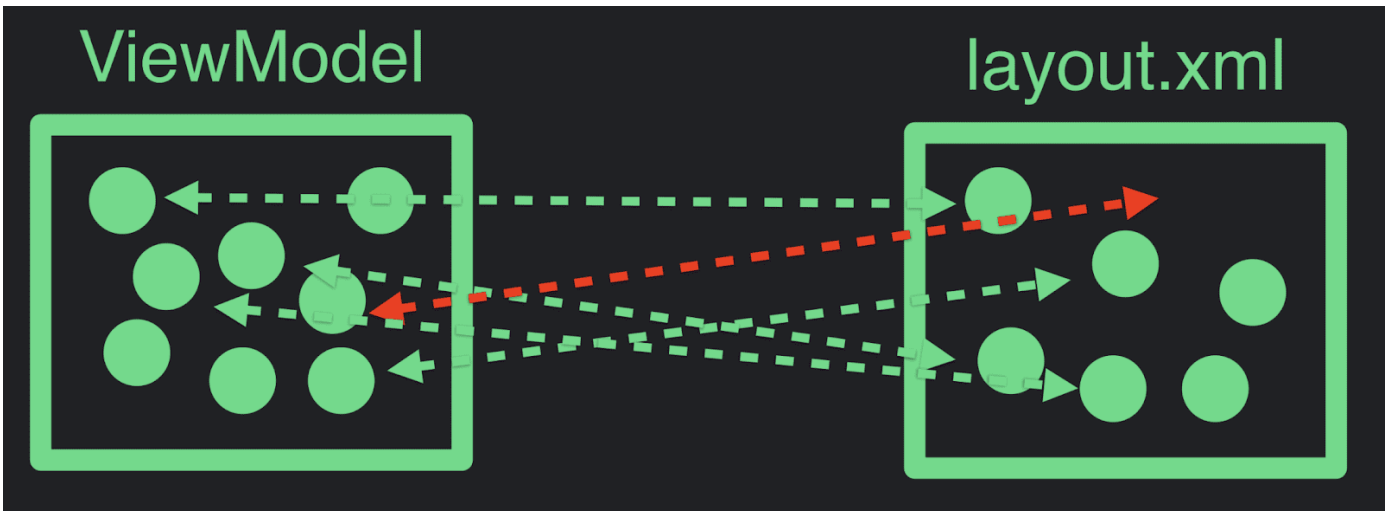
让我们在当前 Android 开发的上下文中进行更为实际的操作，并以视图模型 (view model) 和 XML 布局为例:



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。

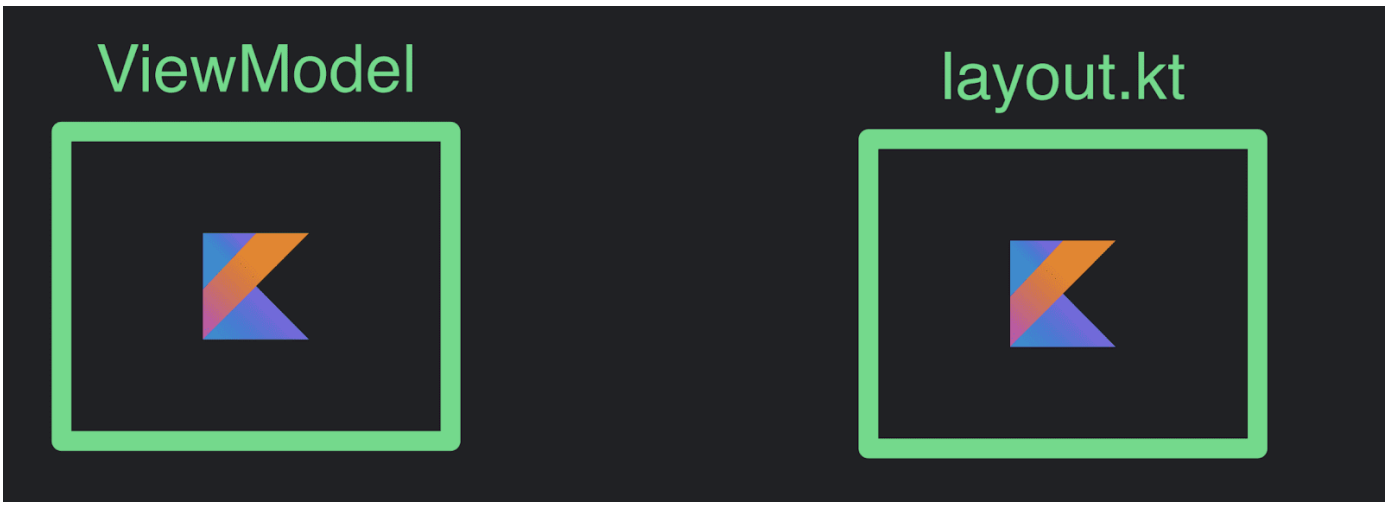
使用这些 API 需要了解 XML 布局是如何定义并与视图模型产生耦合的。由于应用规模会随着时间增长，我们还必须保证这些依赖不会过时。

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 NullReferenceExceptions 一类的问题。

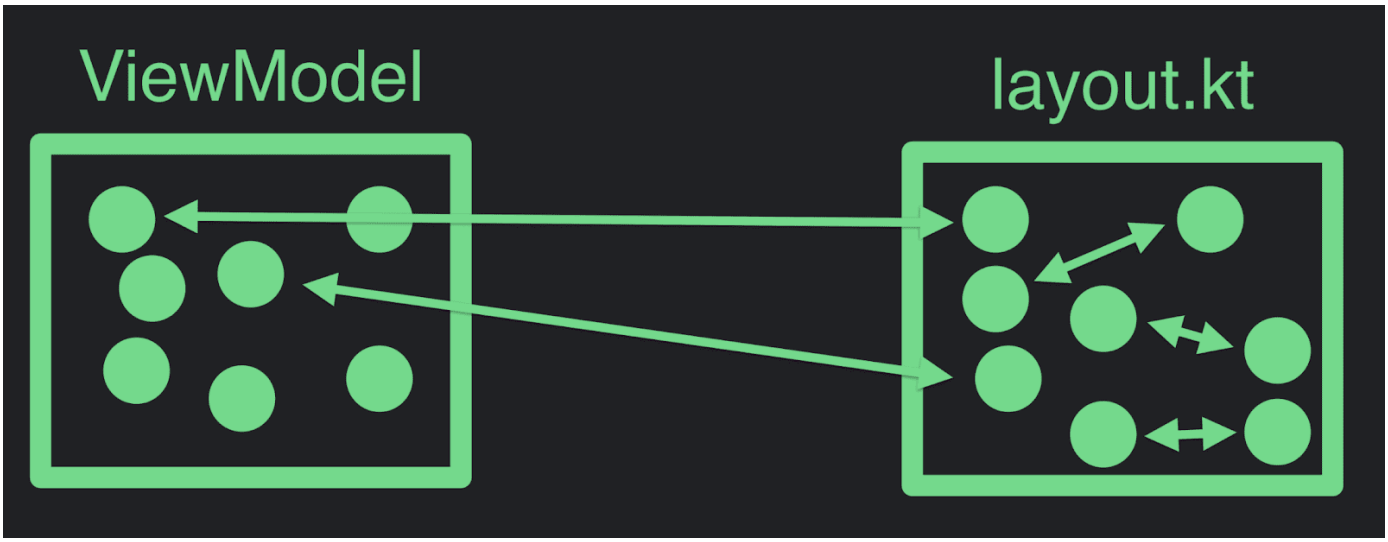


通常，视图模型会使用像 Kotlin 这样的编程语言进行定义，而布局则使用 XML。由于这两种语言的差异，使得它们之间存在一条强制的分隔线。然而即使存在这种情况，视图模型与布局 XML 还是可以关联得十分紧密。换句话说，它们二者紧密耦合。

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。

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。

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量的函数:

```
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。

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 DataInput，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对装饰类型的抽象。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。

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 函数通过回调告知当前改变已备份。

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。

总结

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。

编辑于 10-22

Android　Android 开发　Android Jetpack