

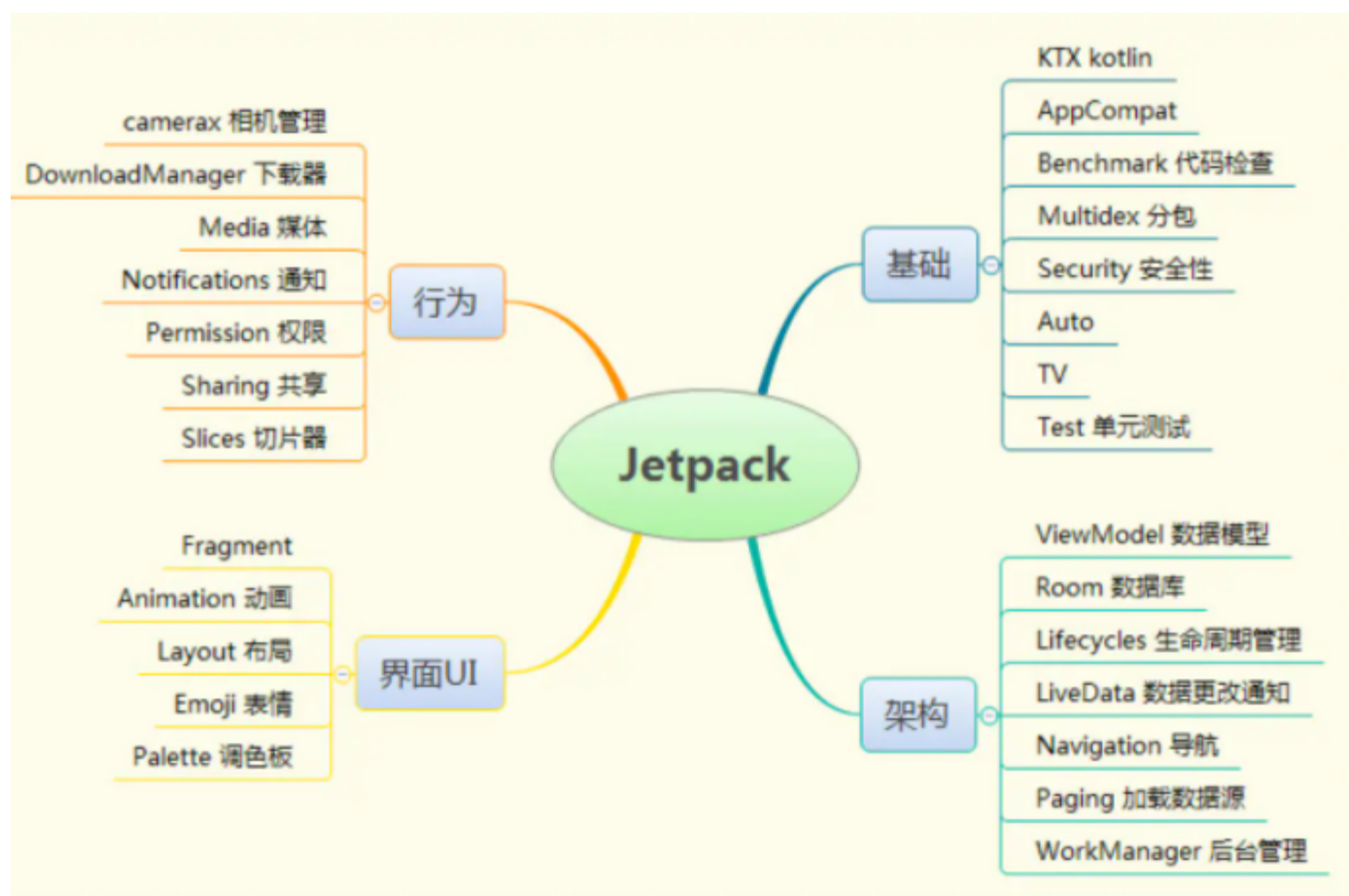
Android JetPack Compose开发应用指南

第一章 初识Jetpack

1.1 JetPack是什么

JetPack: 2018年谷歌I/O 发布了一系列辅助android开发者的实用工具，合称Jetpack。以帮助开发者构建出色的 Android 应用。Jetpack 是一套库、工具和指南，可帮助开发者更轻松地编写优质应用。这些组件可帮助你遵循最佳做法、让你摆脱编写样板代码的工作并简化复杂任务，以便你将精力集中放在所需的代码上。

JetPack分类有四种，分别是Architecture、Foundationy、Behavior、UI。



每个组件都可以单独使用，也可以配合在一起使用。每个组件都给用户提供了一个标准，能够帮助开发者遵循最佳做法，减少样板代码并编写可在各种 Android 版本和设备中一致运行的代码，让开发者能够集中精力编写重要的业务代码。

1.2 JetPack和AndroidX

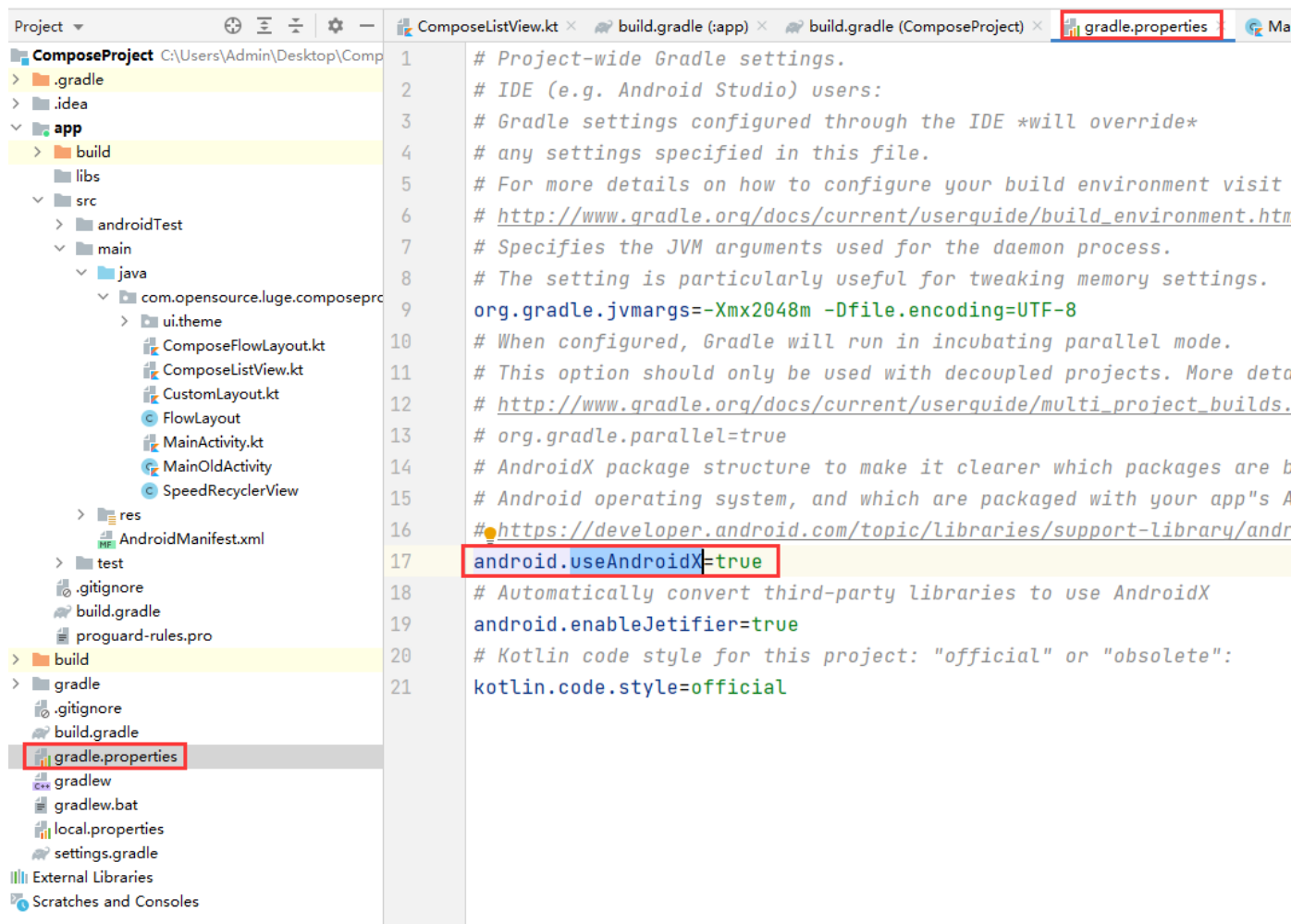
Jetpack 是各种组件库的统称，AndroidX 是这些组件的统一包名。AndroidX 对原始 Android Support Library 进行了重大改进，后者现在已不再维护。androidx 软件包完全取代了 support 包，不仅提供同等的功能，而且提供了新的库。Jetpack 组件中也是完全使用 androidx 开头的包名。与 Support Library 一样，androidx 命名空间中的库与 Android 平台分开提供，并向后兼容各个 Android 版本。

注意：

- AndroidX 中的所有软件包都使用一致的命名空间，以字符串 androidx 开头。Support Library 软件包已映射到对应的 androidx.* 软件包。
- 与 Support Library 不同，androidx 软件包会单独维护和更新。从版本 1.0.0 开始，androidx 软件包使用严格的语义版本控制。可以单独更新项目中的各个 AndroidX 库。
- 版本 28.0.0 是 Support Library 的最后一个版本。以后将不再发布 android.support 库版本。所有新功能都将在 androidx 命名空间中开发。

如果要在新项目中使用命名空间为 androidx 的库，就需要将编译 SDK 设置为 Android 9.0（API 级别 28）或更高版本，并在 [gradle.properties](#) 文件中将以下两个 Android Gradle 插件标志设置为 true。

- `android.useAndroidX`：该标志设置为 true 时，Android 插件会使用对应的 AndroidX 库，而非支持库。如果未指定，那么该标志默认为 false。
- `android.enableJetifier`：该标志设置为 true 时，Android 插件会通过重写其二进制文件来自动迁移现有的第三方库，以使用 AndroidX 依赖项。如果未指定，那么该标志默认为 false。



1.3 AndroidX的迁移

在进入迁移之前，要注意下几点：

- 请使用3.2及更高版本的Android Studio。
- 请在单独的分支中进行迁移。
- 在AndroidX迁移的时候千万不要做任何版本开发、代码重构工作，因为androidx迁移会涉及项目绝大部分的源码文件。（当然也不要过于担心影响app的原有功能，只是对support库中涉及到的包名和类名进行改动）。
- 一定要擦亮眼睛（因为就算用工具迁移，也会存在遗漏的地方，需要少量手动迁移）。

第一步：升级版本

将compileSdkVersion调整为28，将项目的support库版本升级到28.0.0。因为androidx 1.0.0版本和support库的28.0.0版本在二级制层面是等效的，也就是说这两个版本的差异之处仅限于代码包名称，一切的API都是相同的。

第二步：开启Jetifier

在项目的gradle.properties文件内添加如下代码：

```
1 android.useAndroidX=true
2 android.enableJetifier=true
```

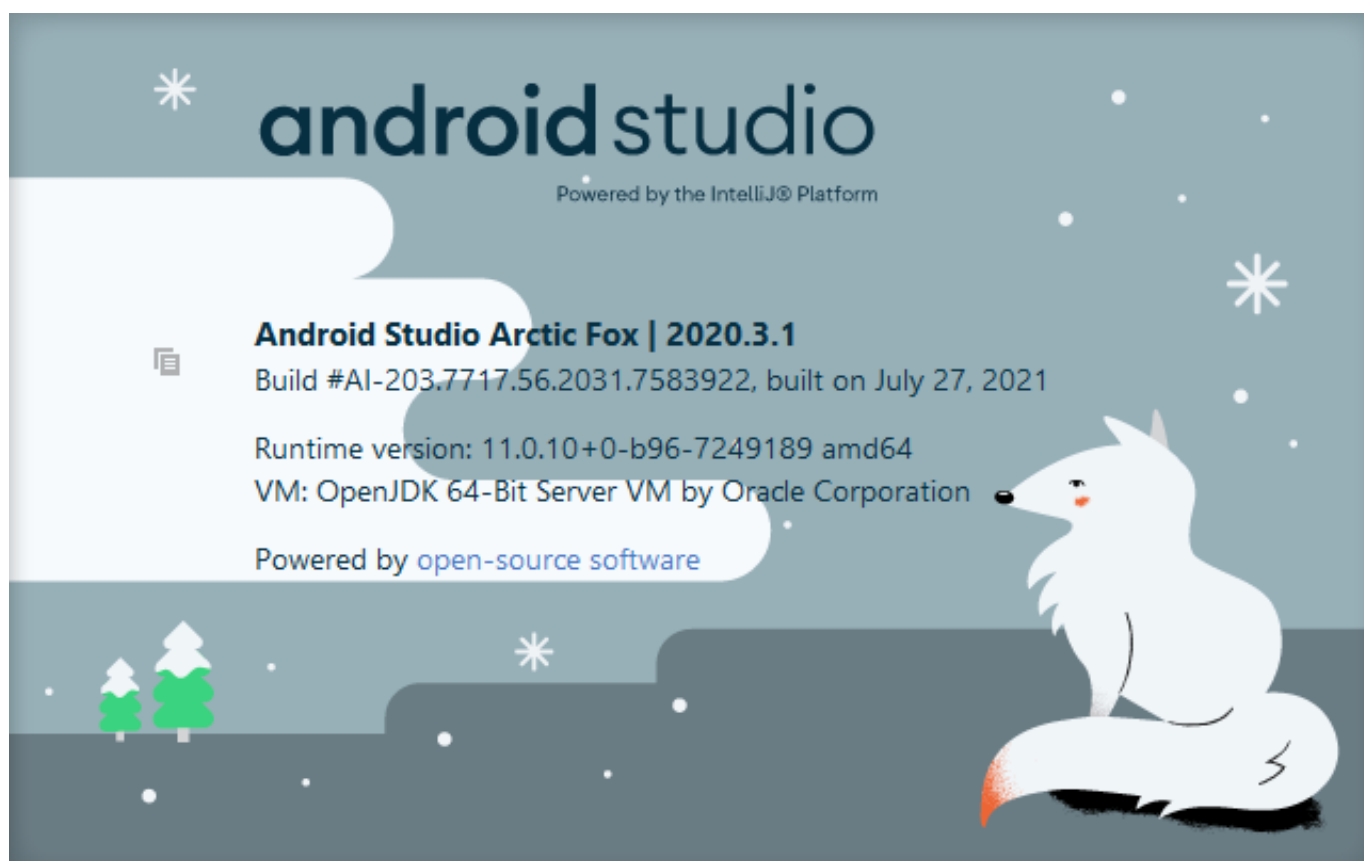
第三步：更新依赖

将app依赖的三方库尽量都升级到支持androidx的版本，这样可以避免在迁移中发生冲突。
如果你使用了kotlin，请将annotationProcessor替换为kapt

第四步：使用Android Studio进行迁移

第二章 Compose的设计原理和基本概念

2.1 JetPack Compose 环境搭建



建议升级到北极狐版本（2020.3.1）。

可以看到IDE已经帮我们构建了生成Compose的选项。

gradle 配置

1) 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：

```
12 kotlinOptions {
13     jvmTarget = '1.8'
14     useIR = true
15 }
16 buildFeatures {
17     compose true
18 }
19 composeOptions {
20     kotlinCompilerExtensionVersion compose_version
21     kotlinCompilerVersion '1.5.10'
22 }
23 packagingOptions {
24     resources {
25         excludes += '/META-INF/{AL2.0,LGPL2.1}'
26     }
27 }
28 }
29
30 dependencies {
31
32     implementation 'androidx.core:core-ktx:1.3.2'
33     implementation 'androidx.appcompat:appcompat:1.2.0'
34     implementation 'com.google.android.material:material:1.3.0'
35     implementation "androidx.compose.ui:ui:$compose_version"
36     implementation "androidx.compose.material:material:$compose_version"
37     implementation "androidx.compose.ui:ui-tooling-preview:$compose_version"
38     implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.3.1'
39     implementation 'androidx.activity:activity-compose:1.3.0-alpha06'
40     testImplementation 'junit:junit:4.+'
41     androidTestImplementation 'androidx.test.ext:junit:1.1.2'
42     androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
43     androidTestImplementation "androidx.compose.ui:ui-test-junit4:$compose_version"
44     debugImplementation "androidx.compose.ui:ui-tooling:$compose_version"
```

开启Compose

2) 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：

```
1 implementation "androidx.compose.ui:ui:$compose_version"
2 //Material Design
3 implementation "androidx.compose.material:material:$compose_version"
4 // Tooling support (Previews, etc.)
5 implementation "androidx.compose.ui:ui-tooling-
6 preview:$compose_version"
7 implementation 'androidx.activity:activity-compose:1.3.0-alpha06'
8 // Foundation (Border, Background, Box, Image, Scroll, shapes,
9 animations, etc.)
10 implementation("androidx.compose.foundation:foundation:1.0.1")
```


2.2 JetPack Compose 新特性和组件依赖

你有没有想过为什么Google要设计一套新的框架，我们不是有View和ViewGroup吗？Android发展有10年的时间了，之前的技术在构建新的用户需求的时候会捉襟见肘，开发人员需要有新的工具来完成UI的编程。另外之前的View的代码已经很冗余了，Google也不希望在之前的代码上继续维护（或者说是污染代码、修改代码）所以这是Compose的出来的缘由。

总结来说：**Compose的优势**

- 紧密结合Kotlin，可以利用现代化编程语言的魅力（高阶函数、各种函数新特性）
- 提高声明式UI开发效率
- 结合最新的IDE可以进行实时预览、动画执行等功能
- Jetpack Compose 为我们提供了很多开箱即用的Material 组件
- 还有很多等你体验

Android Developers > Jetpack > Libraries

☆☆☆☆☆

Compose

[User Guide](#) [Code Sample](#)

API Reference

[androidx.compose](#)

Define your UI programmatically with composable functions that describe its shape and data dependencies.

Compose is combination of 6 Maven Group Ids within `androidx`. Each Group contains a targeted subset of functionality, each with it's own set of release notes.

This table explains the groups and links to each set of release notes.

Group	Description
compose.animation	Build animations in their Jetpack Compose applications to enrich the user experience.
compose.compiler	Transform @Composable functions and enable optimizations with a Kotlin compiler plugin.
compose.foundation	Write Jetpack Compose applications with ready to use building blocks and extend foundation to build your own design system pieces.
compose.material	Build Jetpack Compose UIs with ready to use Material Design Components. This is the higher level entry point of Compose, designed to provide components that match those described at www.material.io .
compose.runtime	Fundamental building blocks of Compose's programming model and state management, and core runtime for the Compose Compiler Plugin to target.
compose.ui	Fundamental components of compose UI needed to interact with the device, including layout, drawing, and input.

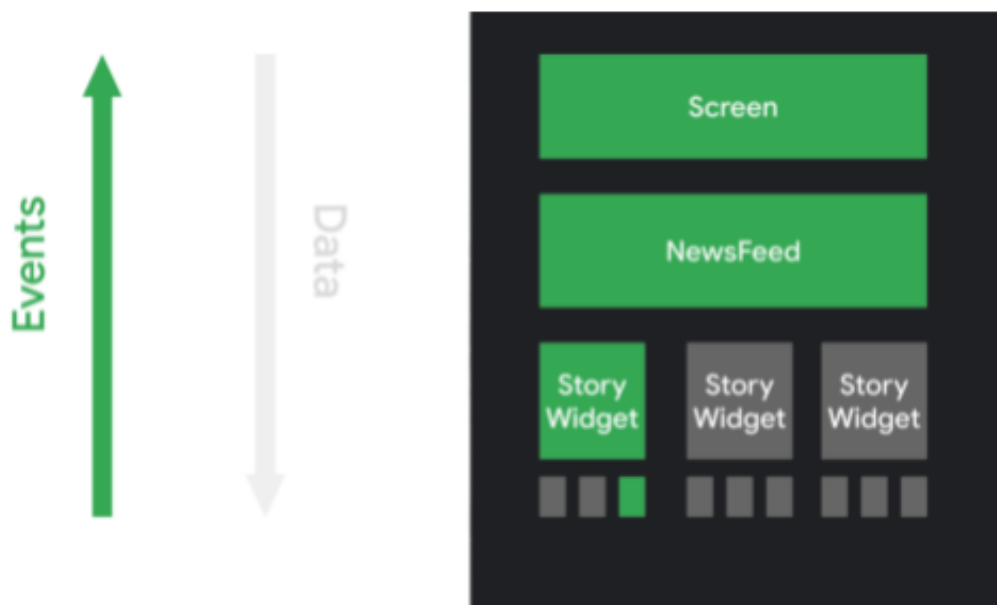
需要注意的地方，Runtime和Compile2个组件，尤其是Compiler完成Compose的编译和中间结果的输出。Ui和material完成了Compose组件最核心的界面的展示。

2.3 JetPack Compose 编程思想总结

Compose需要解决的问题是关注点分离。强内聚低耦合。设计思想来说，java语言更多的是使用继承，Kotlin推荐我们使用组合。组合可以增加自由度、解决java当中单个父类限制。

Jetpack Compose 是一个适用于 Android 的新式声明性界面工具包。Compose 提供声明性 API，让您可在不以命令方式改变前端视图的情况下呈现应用界面，从而使编写和维护应用界面变得更加容易。

- 声明性编程范式
- 可组合函数（@Composable函数）
- 声明性范式转变（在 Compose 的声明性方法中，微件相对无状态，并且不提供 setter 或 getter 函数。）



如何理解Compose中的重组

官方说明：

Recomposition skips as much as possible When portions of your UI are invalid, Compose does its best to recompose just the portions that need to be updated. This means it may skip to re-run a single Button's composable without executing any of the composables above or below it in the UI tree.

经典素材阅读：

https://dev.to/zachklipp/scoped-recomposition-jetpack-compose-what-happens-when-state-changes-l78?utm_source=dormosheio&utm_campaign=dormosheio

```
1  /**
2   * Display a list of names the user can click with a header
3   */
4  @Composable
5  fun NamePicker(
6      header: String,
7      names: List<String>,
8      onNameClicked: (String) -> Unit
9  ) {
10     Column {
11         // this will recompose when [header] changes, but not when
12         // [names] changes
13         Text(header, style = MaterialTheme.typography.h5)
14         Divider()
15
16         // LazyColumnFor is the Compose version of a RecyclerView.
17         // The lambda passed is similar to a RecyclerView.ViewHolder.
18         LazyColumnFor(names) { name ->
19             // When an item's [name] updates, the adapter for that
20             // item
21             // will recompose. This will not recompose when [header]
22             // changes
23             NamePickerItem(name, onNameClicked)
24         }
25     }
26 }
27
28 /**
29  * Display a single name the user can click.
30  */
31 @Composable
32 private fun NamePickerItem(name: String, onCliked: (String) -> Unit) {
33     Text(name, Modifier.clickable(onClick = { onCliked(name) }))
34 }
```

重组我们需要知道重组范围这个概念就可以了。

理解Compose中的mutableStateOf和remember

函数源码部分参见视频分析。

核心知识点总结：

- 代理属性Kotlin怎么用
- remember的实质含义
- MutableState
- Compose里面的State

理解Compose里面的State

```
1 interface MutableState<T> : State<T> {  
2     override var value: T  
3 }
```

使用：

```
1 val mutableState = remember { mutableStateOf(default) }  
2 var value by remember { mutableStateOf(default) }  
3 val (value, setValue) = remember { mutableStateOf(default) }
```

Jetpack Compose 像React或者Flutter一样，需要通过state变更驱动UI刷新。但是Compose没有React的ClassComponent或者Flutter的StatefulWidget，那Compose是如何更新并监视state呢？一种方式是使用state{...}方法声明初始状态，并监听状态变化。当状态变化时，会触发recomposition执行从而刷新UI。

```
@Composable  
fun MyCheckbox() {  
    // 初始state  
    var checked by state { false }  
  
    Checkbox(  
        checked = checked.value,  
        onCheckedChange = {  
            // 更新state后, recomposition执行  
            checked.value = it  
        }  
    )  
}
```

state{...}返回checked的同时，建立当前节点（MyCheckbox）对state的监听。

第三章 Compose入门

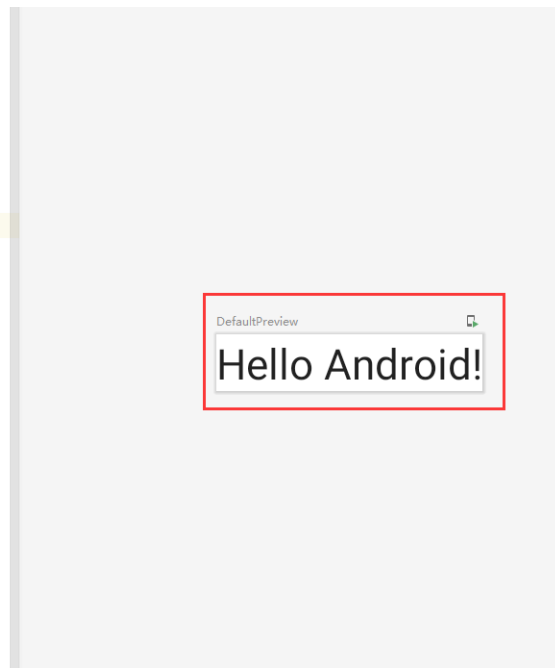
3.1 JetPack Compose 入门的基础案例

打开 MainActivity.kt 文件，如果编译顺利的话，你应该看到类似下面的样子：

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeProject2Theme {
                // A surface container using the 'background' color from the theme
                Surface(color = MaterialTheme.colors.background) {
                    Greeting(name: "Android")
                }
            }
        }
    }
}

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

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    ComposeProject2Theme {
        Greeting(name: "Android")
    }
}
```



- 刷新视图，当代码发生变化的时候，点击它可以刷新视图界面。
- 进行交互模式，比如视图上有个按钮，就可以不用运行直接点击并作出对应的响应。

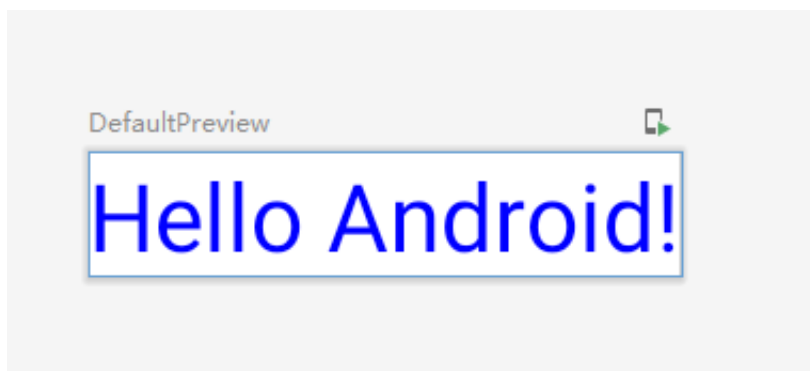
上面是第一个组件Text，让我们来看看Text的源码：

```
1  @Composable
2  fun Text(
3      text: String, //显示的内容
4      modifier: Modifier = Modifier, // 关于布局的设置
5      color: Color = Color.Unspecified, // 文字颜色
6      fontSize: TextUnit = TextUnit.Unspecified, //文字大小
7      fontStyle: FontStyle? = null, //文字风格
8      fontWeight: FontWeight? = null, //文字粗细
9      fontFamily: FontFamily? = null, //字体风格
10     letterSpacing: TextUnit = TextUnit.Unspecified, //字间距
11     textDecorations: TextDecorations? = null, //文字装饰、比如下划线
12     textAlign: TextAlign? = null, //对齐方式
13     lineHeight: TextUnit = TextUnit.Unspecified, //行高
14     overflow: TextOverflow = TextOverflow.Clip, // 文字显示不完的处理方式
15     softWrap: Boolean = true, // 文本是否应在换行符处中断,对中文没啥用
16     maxLines: Int = Int.MAX_VALUE, //最大行数
17     onTextLayout: (TextLayoutResult) -> Unit = {},
18     style: TextStyle = AmbientTextStyle.current
```

```
19 | ) {  
20 | .....  
21 | }
```

我们需要设置个性化 **Text** 组件，只要传入对应的参数就行了，和 XML 添加对应属性差不多。比如把文字改成蓝色：

```
1 @Composable  
2 fun TestText(){  
3     Text(text = "Hello Android",color= Color.Blue)  
4 }
```



我们在来看看Button组件又应该如何使用？

```
1 @Composable  
2 fun Button(  
3     onClick: () -> Unit, // 按钮的点击事件  
4     modifier: Modifier = Modifier,  
5     enabled: Boolean = true, //是否启用  
6     interactionState: InteractionState = remember {  
InteractionState() },  
7     elevation: ButtonElevation? = ButtonDefaults.elevation(), //海拔  
8     shape: Shape = MaterialTheme.shapes.small, //形状  
9     border: BorderStroke? = null, //边框  
10    colors: ButtonColors = ButtonDefaults.buttonColors(), //颜色  
11    contentPadding: PaddingValues = ButtonDefaults.ContentPadding, //内  
    边距  
12    content: @Composable RowScope.() -> Unit //内容  
13 ) { ..... }  
14
```

我们可以仔细的看一看，这个 **Button** 组件点击事件 **onClick** 和内容 **content** 是必填参数外，好像没有设置 **Button** 文字的参数。这和我们以前在 XML 构建视图不太一样。那么我们怎么让 **Button** 显示文字呢？我们在 **content** 添加一个 **Text** 组件就可以了。

```
1 @Composable
2 fun TestButton() {
3     Button(onClick = { }) {
4         Text(text = "button")
5     }
6 }
```

最后一个参数 `content` 是函数类型，为了让这个函数体里面能调用可组合函数 所以在这个参数还加个 `@Composable` 表示是一个可以组合函数类型，所以 `Button` 的 `content` 的函数体中可以调用其它可组合函数。

我们还可以修改`Button`的样式，比如修改`Shape`的样式，我们可以任意的修改大小，比如我修改边角的DP值，我修改到12DP。我们再来看看效果，

3.2 JetPack Compose 基础实战

在3.1小结中我们看到了 Jetpack Compose的单个组件的使用，我们接下来在这一小结我们看看如何在Compose中使用布局组件，进一步加强Compose的认知和学习。Jetpack Compose 是用于构建原生界面的新款 Android 工具包。它可简化并加快 Android 上的界面开发。使用更少的代码、强大的工具和直观的 Kotlin API，快速让应用生动而精彩。

Compose中可以将多个控件元素组合使用，例如下面这样：

```
1 @Preview(showBackground = true)
2 @Composable
3 fun ShowWidgetGroup() {
4     Text(text = "不为往事扰")
5     Text(text = "余生只愿笑")
6 }
```

但是我们会发现，如果仅仅是这样，两个文本控件会重叠在一起，类似于上面这种样式：我们想将多个控件垂直摆放在一起，可以在控件外层添加`Column`：组件

```
1 @Preview(showBackground = true)
2 @Composable
3 fun ShowWidgetGroup() {
4     Column { //增加Column组件
5         Text(text = "不为往事扰")
6         Text(text = "余生只愿笑")
7     }
8 }
```

如果想要水平摆放在页面上可以使用 Row 组件：

```
1 @Preview(showBackground = true)
2 @Composable
3 fun ShowWidgetGroup() {
4     Row() { //水平组件
5         Text(text = "不为往事扰")
6         Text(text = "余生只愿笑")
7     }
8 }
```

如果我们还想在文字下面添加一张图片，可以直接添加一个 Image 控件：

```
1 @Composable
2 fun WidgetGroup() {
3     val image = imageResource(id = R.drawable.header)
4     val imageModifier = Modifier
5         .preferredHeight(180.dp)
6         .fillMaxWidth()
7         .clip(RoundedCornerShape(10.dp))
8     Column {
9         Text(text = "不为往事扰")
10        Text(text = "余生只愿笑")
11        Spacer(modifier = Modifier.preferredHeight(10.dp))
12        Image(asset = image, modifier = imageModifier, contentScale =
ContentScale.Crop)
13    }
14 }
```


第四章 Compose布局

4.1 Compose State

Jetpack Compose 像React或者Flutter一样，需要通过state变更驱动UI刷新。但是Compose没有React的ClassComponent或者Flutter的StatefulWidget，那Compose是如何更新并监视state呢？

一种方式是使用state{...}方法声明初始状态，并监听状态变化。当状态变化时，会触发recomposition执行从而刷新UI。

```
1 @Composable
2 fun MyCheckbox() {
3     // 初始state
4     var checked by state { false }
5
6     Checkbox(
7         checked = checked.value,
8         onCheckedChange = {
9             // 更新state后, recomposition执行
10            checked.value = it
11        }
12    )
13 }
```

state{...}返回checked的同时，建立当前节点（MyCheckbox）对state的监听,state{...}并不直接返回bool值，而是一个代理：

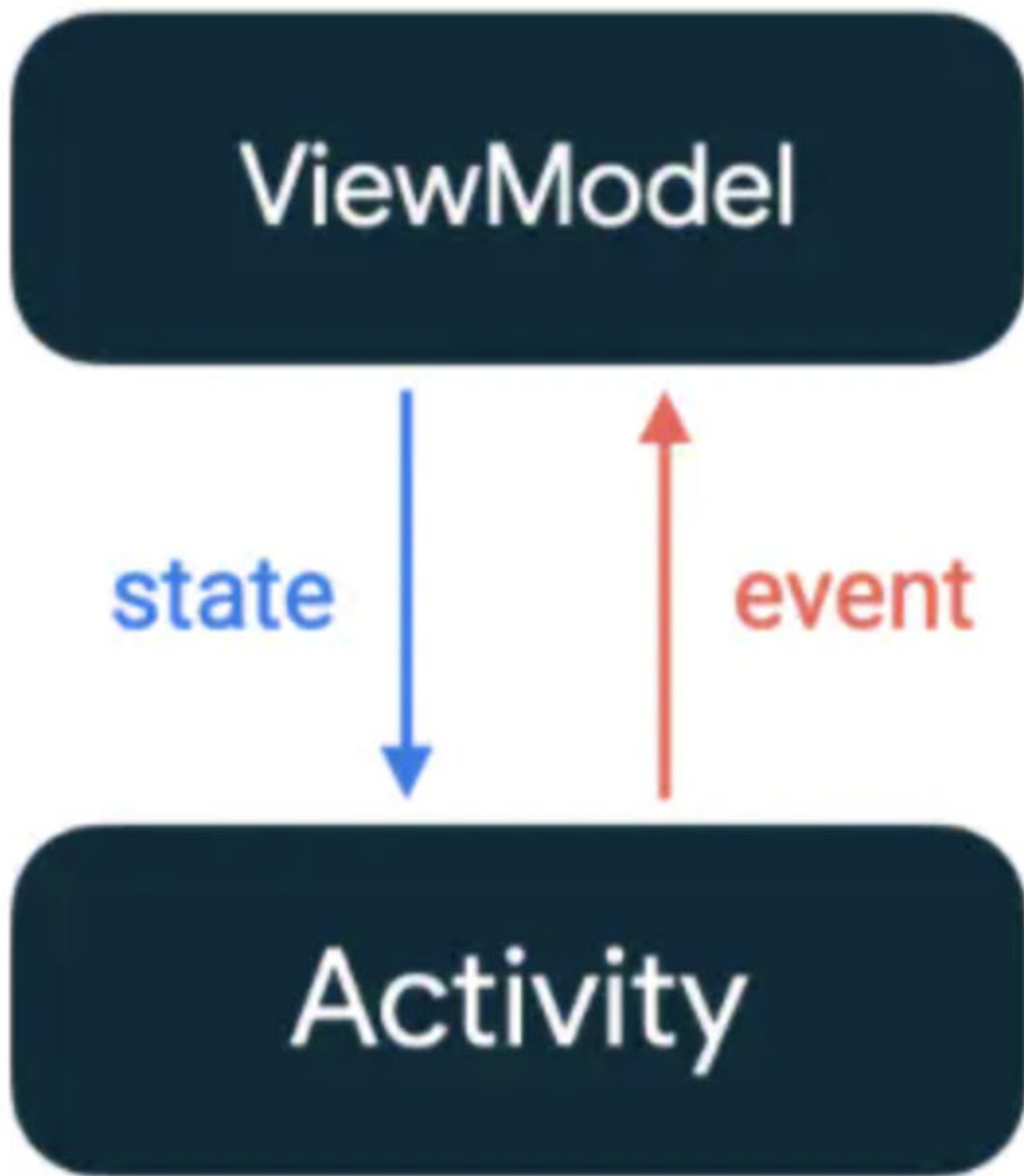
```
1 interface MutableState<T> : State<T> {
2     override var value: T
3     operator fun component1(): T
4     operator fun component2(): (T) -> Unit
5     operator fun getValue(thisObj: Any?, property: KProperty<*>): T
6     operator fun setValue(thisObj: Any?, property: KProperty<*>, next:
7     T)
8 }
```

当checked被set时，通过代理触发recomposition，找到observer的节点（MyCheckbox）并重新执行Composable方法。可以看到，Compose中吸收了类似React Hooks的机制，让函数组件依然可以像类组件一样管理和订阅state。

另外一种方式就是通过@model注解来完成Compose State的管理。使用@model添加的Class，其所有的属性的变成了可观察的状态。

```
1 @Model
2 class Count(
3     var count: Int = 0
4 )
5
6 @Composable
7 fun MyText() {
8     // 使用Model类创建实例
9     val count = Count()
10
11     Ripple(bounded = false) {
12         Clickable(onClick = {
13             // 更新Model的值, recomposition执行
14             count.count += 1
15         }) {
16             Container {
17                 // count中保存Model的最新值
18                 Text(text = count.count.toString())
19             }
20         }
21     }
22 }
```

Android 推出了ViewModel 和LiveData,就是为了解决命令式编程,想要改变UI就必须得调用更新UI的方法。



单向数据流就是指 符合事件向上传递而状态向下传递的设计模式。例如在ViewModel中，事件通过UI的调用向上传递给ViewModel，而状态通过LiveData 的 `setValue` 向下传递。就像刚才说的，单向数据流不仅仅是描述ViewModel的术语，任何属于这种设计的能被称之为单向数据流。Compose也是遵循这个模型的一个UI框架，在Compose中推荐用 `MutableState` 来管理状态，而不是LiveData。

在Compose中通常这样声明state:

```
1 | val name by mutableStateOf("Compose")
```

这里用到了Kotlin的by关键字，name的类型，取决于mutableStateOf方法传进去的类型，在这里其实就是String类型，通过by引用的对象，在取值和赋值的时候均会调用代理类的getValue和setValue方法方法，这两个方法分别在State接口和MutableState中声明。

```
1 State 中的getValue
2 inline operator fun <T> State<T>.getValue(thisObj: Any?, property:
  KProperty<*>): T = value
3
4 MutableState 中的setValue
5 inline operator fun <T> MutableState<T>.setValue(thisObj: Any?,
  property: KProperty<*>, value: T) {
6     this.value = value
7 }
```

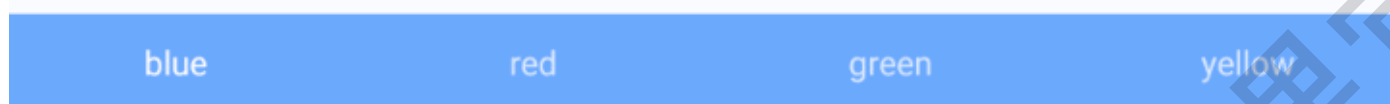
我们来看一个实际的案列：

```
1 @Composable
2 fun Counter() {
3     var count by mutableStateOf(1)
4     Button(onClick = {
5         count ++
6     }) {
7         Text(text = count.toString())
8     }
9 }
```

在这段代码中，用代理模式，将int类型的count代理给了mutableStateOf返回的state，然后对count进行set操作的时候就会触发recompose，然后对Counter进行重新绘制。

4.2 Compose 样式 (Theme)

Compose提供了系统化的方法来帮助我们自定义主题，这让我们在实现暗黑主题以及其他颜色主题的时候非常非常的方便。这一小节我们先通过一个小的案列来了解Compose的样式是什么？又该如何的设置？



这个需求很简单，我们的App需要有蓝，红，绿，黄四种主题色，在点击tab的时候分别切换不同的主题色，大致效果如上所示。

默认的主题配置

```
1  setContent {
2      ComposeComingTheme {
3
4          Surface(color = MaterialTheme.colors.background) {
5
6          }
7      }
8  }
9
10
11 private val DarkColorPalette = darkColors(
12     primary = Purple200,
13     primaryVariant = Purple700,
14     secondary = Teal200
15 )
16
17 private val LightColorPalette = lightColors(
18     primary = Purple500,
19     primaryVariant = Purple700,
20     secondary = Teal200
21 )
22
23 @Composable
24 fun ComposeComingTheme(
25     darkTheme: Boolean = isSystemInDarkTheme(),
26     content: @Composable() () -> Unit
27 ) {
28     val colors = if (darkTheme) {
29         DarkColorPalette
30     } else {
31         LightColorPalette
32     }
33
34     MaterialTheme(
35         colors = colors,
36         typography = Typography,
37         shapes = Shapes,
38         content = content
39     )
40 }
41
```

通过上述代码，我们可以了解到：ComposeComingTheme 该组合函数在 MaterialTheme 上构建而成，MaterialTheme由colors、typography、shapes属性组成。然后该组合函数默认根据系统 是否是暗黑主题 将 颜色属性 进行了不同的定义。如果是暗黑主题使用 DarkColorPalette，否则使用LightColorPalette。darkColors和lightColors都继承自 Colors，并有相关的默认颜色值。我们在来看看Color的源代码是什么？

```
1 @Stable
2 class Colors(
3     primary: Color,
4     primaryVariant: Color,
5     secondary: Color,
6     secondaryVariant: Color,
7     background: Color,
8     surface: Color,
9     error: Color,
10    onPrimary: Color,
11    onSecondary: Color,
12    onBackground: Color,
13    onSurface: Color,
14    onError: Color,
15    isLight: Boolean
16 )
```

我们现在来看看如何自定义主题？首先在ui.theme文件夹下的Color.kt文件中添加如下颜色：

```
1 val blue = Color(0xFF579DFD)
2 val red = Color(0xFFEB685E)
3 val green = Color(0xFF27BA6A)
4 val yellow = Color(0xFFFFCB1F)
```

然后在ui.theme文件夹下的Theme.kt文件中自定义主题，在这里我们统一使用了lightColors，然后根据传递进来的主题的名称将primary属性设置为了不同的颜色，如下所示：

```
1 @Composable
2 fun MyAppTheme(
3     themeName: String = "blue",
4     content: @Composable() () -> Unit
5 ) {
6
7     val colors = when (themeName) {
8         "red" -> lightColors(primary = red)
```



```
9         "green" -> lightColors(primary = green)
10        "yellow" -> lightColors(primary = yellow)
11        else -> lightColors(primary = blue)
12    }
13
14    MaterialTheme(
15        colors = colors,
16        typography = Typography,
17        shapes = Shapes,
18        content = content
19    )
20 }
```

接下来是TabRow的实现，我们将TabRow的背景颜色 **backgroundColor** 设置为刚定义的 **primary** 的颜色，点击不同tab的时候设置不同的主题，并将点击事件提升到上层处理，代码如下所示：

```
1  @Composable
2  fun MyTabRow(
3      onTabItemClick: (name: String) -> Unit,
4      indexDefault: Int = 0
5  ) {
6
7      val indexState = remember {
8          mutableStateOf(indexDefault)
9      }
10
11     val colorsTab = arrayOf("blue", "red", "green", "yellow")
12
13     TabRow(
14         selectedTabIndex = indexState.value,
15         modifier = Modifier
16             .fillMaxWidth()
17             .height(46.dp),
18         backgroundColor = MaterialTheme.colors.primary
19     ) {
20         for ((index, name) in colorsTab.withIndex()) {
21             Tab(
22                 selected = index == indexState.value,
23                 onClick = {
24                     indexState.value = index
25                     onTabItemClick(colorsTab[index])
26                 },
27                 modifier = Modifier.fillMaxHeight(),
```

```
28
29         ) {
30         Text(
31             text = name,
32             modifier = Modifier
33                 .fillMaxWidth()
34                 .wrapContentHeight(),
35             textAlign = TextAlign.Center
36         )
37     }
38 }
39 }
40 }
```

通过上面的代码设置我们就完成了Compose的自定义主题需求。

4.3 Compose布局核心控件

Jetpack Compose 提供了 [Material Design](#) 的实现，后者是一个用于创建数字化界面的综合设计系统。Material 组件（按钮、卡片、开关等）和布局（如 **Scaffold**）可作为可组合函数提供。

Button

```
1 Button(
2     onClick = { /* ... */ },
3     // Uses ButtonDefaults.ContentPadding by default
4     contentPadding = PaddingValues(
5         start = 20.dp,
6         top = 12.dp,
7         end = 20.dp,
8         bottom = 12.dp
9     )
10 ) {
11     // Inner content including an icon and a text label
12     Icon(
13         Icons.Filled.Favorite,
14         contentDescription = "Favorite",
15         modifier = Modifier.size(ButtonDefaults.IconSize)
16     )
17     Spacer(Modifier.size(ButtonDefaults.IconSpacing))
18     Text("Like")
19 }
```

Scaffold

```
1  @Composable
2  fun ScaffoldDemo() {
3      val context = LocalContext.current
4      val scaffoldState = rememberScaffoldState()
5      Scaffold(
6          scaffoldState = scaffoldState,
7          //抽屉组件区域
8          drawerContent = {
9
10             Box(
11                 modifier = Modifier.fillMaxSize(),
12                 contentAlignment = Alignment.Center
13             ) {
14                 Text(text = "抽屉组件中内容")
15             }
16
17         },
18         //标题栏区域
19         topBar = {
20             TopAppBar(
21                 title = { Text(text = "脚手架示例") },
22                 navigationIcon = {
23                     IconButton(
24                         onClick = {
25
26                             Toast.makeText(context, "click! ! ", Toast.LENGTH_LONG).show()
27                         }
28                     ) {
29                         Icon(
30                             imageVector = Icons.Filled.Menu,
31                             contentDescription = null
32                         )
33                     }
34                 }
35             ),
36             //悬浮按钮
37             floatingActionButton = {
38                 ExtendedFloatingActionButton(
39                     text = { Text("悬浮按钮") },
40                     onClick = { }
41                 )
42             }
43         }
44     )
45 }
```

```
42     },
43     floatingActionButtonPosition = FabPosition.End,
44     //屏幕内容区域
45     content= {
46         Box(
47             modifier = Modifier.fillMaxSize(),
48             contentAlignment = Alignment.Center
49         ) {
50             Text(text = "屏幕内容区域")
51         }
52     },
53 )
54 }
```

BottomNavigation

```
1  @Composable
2  fun ShowBottomNavigation() {
3      var selectIndex by remember {
4          mutableStateOf(0)
5      }
6      val navList = listOf("首页", "发现", "我的")
7
8      Scaffold(bottomBar = {
9          BottomNavigation() {
10             navList.forEachIndexed { index, str ->
11                 BottomNavigationItem(
12                     selected = index == selectIndex, onClick = {
13                         selectIndex = index },
14                     icon = {
15                         Icon(imageVector = Icons.Default.Favorite,
16                             contentDescription = null )
17                     }, label = {Text(str)}
18                 )
19             }
20         }) {
21             Text(text = "这是${navList[selectIndex]}")
22         }
23     }
```

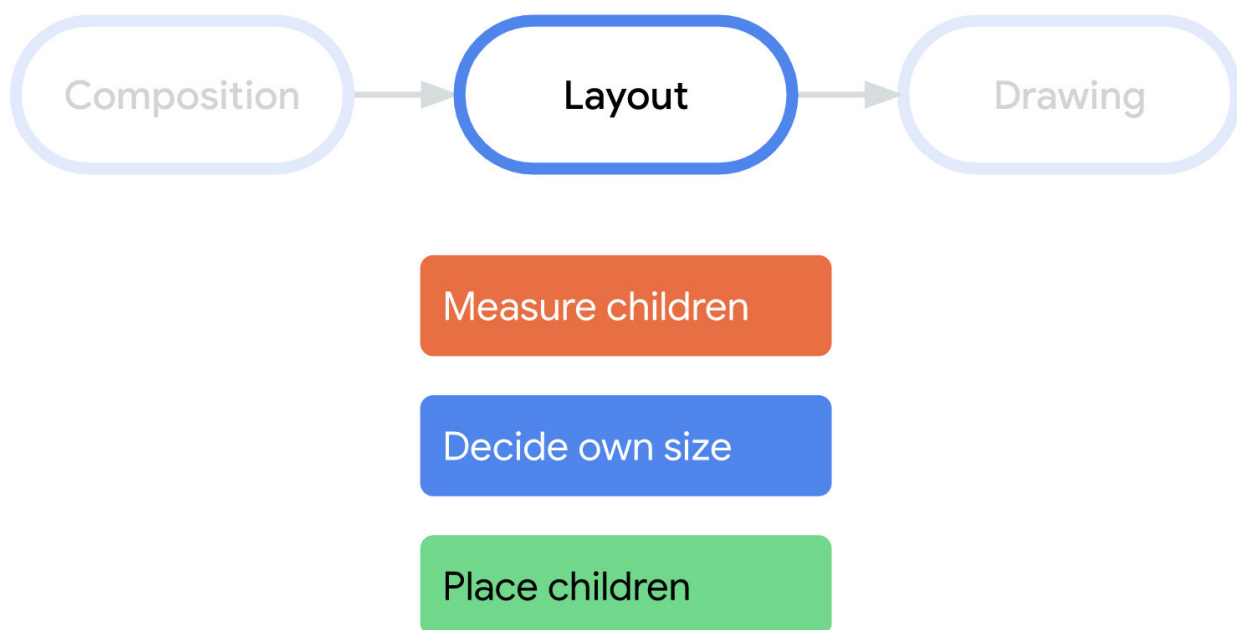
4.4 自定义布局

在 Compose 中，界面元素由可组合函数表示，此类函数在被调用后会发出一部分界面，这部分界面随后会被添加到呈现在屏幕上的界面树中。每个界面元素都有一个父元素，还可能有多个子元素。此外，每个元素在其父元素中都有一个位置，指定为 (x, y) 位置；也都有一个尺寸，指定为 `width` 和 `height`。

父元素定义其子元素的约束条件。元素需要在这些约束条件内定义尺寸。约束条件可限制元素的最小和最大 `width` 和 `height`。如果某个元素有子元素，它可能会测量每个子元素，以帮助确定其尺寸。一旦某个元素确定并报告了它自己的尺寸，就有机会定义如何相对于自身放置它的子元素。

在界面树中布置每个节点的过程分为三个步骤。每个节点必须：

1. 测量所有子项
2. 确定自己的尺寸
3. 放置其子项



我们来看一下如何创建自定义布局，`layout` 修饰符仅更改调用可组合项。如需测量和布置多个可组合项，请改用 `Layout` 可组合项。此可组合项允许您手动测量和布置子项。`Column` 和 `Row` 等所有较高级别的布局都使用 `Layout` 可组合项构建而成。在 `View` 系统中，创建自定义布局必须扩展 `ViewGroup` 并实现测量和布局函数。在 `Compose` 中，您只需使用 `Layout` 可组合项编写一个函数即可。

```
1 @Composable
2 fun MyBasicColumn(
3     modifier: Modifier = Modifier,
4     content: @Composable () -> Unit
5 ) {
6     Layout(
7         modifier = modifier,
8         content = content
9     ) { measurables, constraints ->
10         // measure and position children given constraints logic here
11     }
12 }
```

与 `layout` 修饰符类似，`measurables` 是需要测量的子项的列表，而 `constraints` 是来自父项的约束条件。完整的代码如下：

```
1 @Composable
2 fun MyBasicColumn(
3     modifier: Modifier = Modifier,
4     content: @Composable () -> Unit
5 ) {
6     Layout(
7         modifier = modifier,
8         content = content
9     ) { measurables, constraints ->
10         // Don't constrain child views further, measure them with
11         // given constraints
12         // List of measured children
13         val placeables = measurables.map { measurable ->
14             // Measure each children
15             measurable.measure(constraints)
16         }
17
18         // Set the size of the layout as big as it can
19         layout(constraints.maxWidth, constraints.maxHeight) {
20             // Track the y co-ord we have placed children up to
21             var yPosition = 0
22
23             // Place children in the parent layout
24             placeables.forEach { placeable ->
25                 // Position item on the screen
26                 placeable.placeRelative(x = 0, y = yPosition)
27
28                 // Record the y co-ord placed up to
```

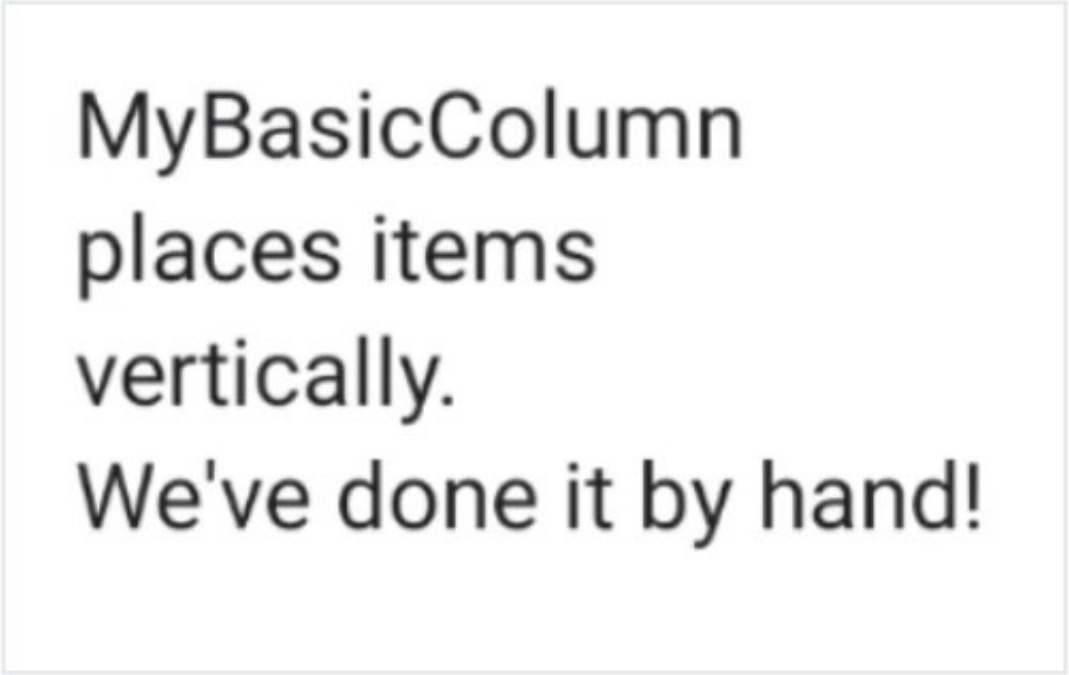


```
28         yPosPosition += placeable.height
29     }
30 }
31 }
32 }
```

该自定义可组合项的使用方式如下：

```
1 @Composable
2 fun CallingComposable(modifier: Modifier = Modifier) {
3     MyBasicColumn(modifier.padding(8.dp)) {
4         Text("MyBasicColumn")
5         Text("places items")
6         Text("vertically.")
7         Text("We've done it by hand!")
8     }
9 }
```

具体的效果如下：



MyBasicColumn
places items
vertically.
We've done it by hand!

4.5 Compose中的ConstraintLayout

第一步，如需使用 Compose 中的 ConstraintLayout，您需要在 build.gradle 中添加以下依赖项：

```
1 implementation "androidx.constraintlayout:constraintlayout-  
compose:1.0.0-beta02"
```

是否将 `ConstraintLayout` 用于 `Compose` 中的特定界面取决于开发者的偏好。在 `Android View` 系统中，使用 `ConstraintLayout` 作为构建更高性能布局的一种方法，但这在 `Compose` 中并不是问题。在需要进行选择时，请考虑 `ConstraintLayout` 是否有助于提高可组合项的可读性和可维护性。

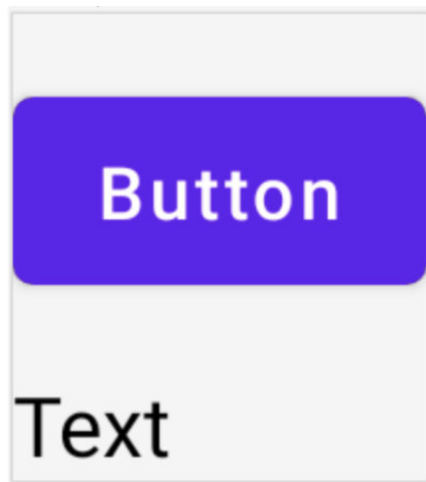
`Compose` 中的 `ConstraintLayout` 支持 [DSL](#):

- 引用是使用 `createRefs()` 或 `createRefFor()` 创建的，`ConstraintLayout` 中的每个可组合项都需要有与之关联的引用。
- 约束条件是使用 `constrainAs()` 修饰符提供的，该修饰符将引用作为参数，可让您在主体 `lambda` 中指定其约束条件。
- 约束条件是使用 `linkTo()` 或其他有用的方法指定的。
- `parent` 是一个现有的引用，可用于指定对 `ConstraintLayout` 可组合项本身的约束条件。

```
1 @Composable  
2 fun ConstraintLayoutContent() {  
3     ConstraintLayout {  
4         // Create references for the composables to constrain  
5         val (button, text) = createRefs()  
6  
7         Button(  
8             onClick = { /* Do something */ },  
9             // Assign reference "button" to the Button composable  
10            // and constrain it to the top of the ConstraintLayout  
11            modifier = Modifier.constrainAs(button) {  
12                top.linkTo(parent.top, margin = 16.dp)  
13            }  
14        ) {  
15            Text("Button")  
16        }  
17  
18        // Assign reference "text" to the Text composable  
19        // and constrain it to the bottom of the Button composable  
20        Text("Text", Modifier.constrainAs(text) {  
21            top.linkTo(button.bottom, margin = 16.dp)  
22        })  
23    }  
24 }
```

此代码使用 `16.dp` 的外边距来约束 `Button` 顶部到父项的距离，同样使用 `16.dp` 的外边距来约束 `Text` 到 `Button` 底部的距离。

效果如下：



我们可以在官网的代码的基础上自由拓展：

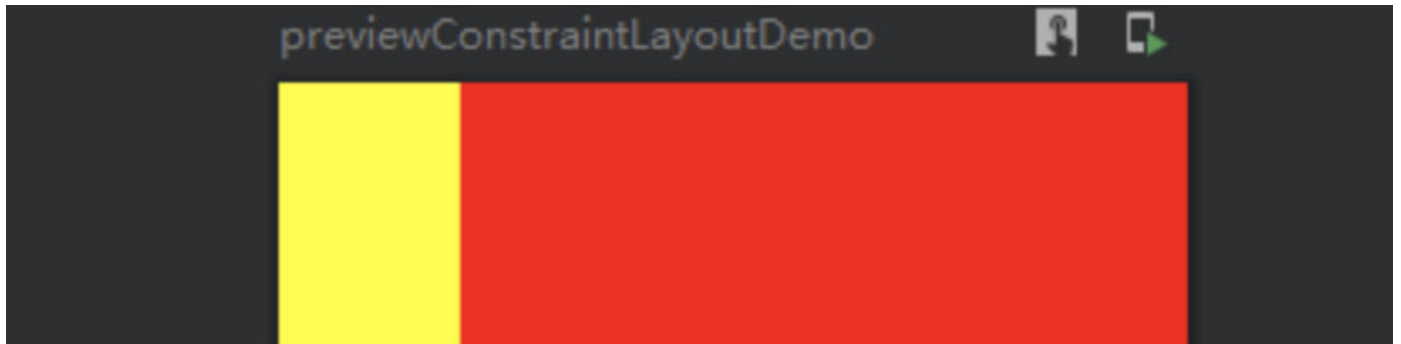
```
1 @Composable
2 fun ConstraintLayoutContent() {
3     ConstraintLayout {
4         // 创建references
5         val (button1, button2, text) = createRefs()
6
7         Button(
8             onClick = {},
9             modifier = Modifier.constrainAs(button1) {
10                 top.linkTo(parent.top, margin = 16.dp)
11             }
12         ) {
13             Text("Button1")
14         }
15         // Text显示在Button1的下方，Text的中线和Button1的end对齐
16         Text("Text", Modifier.constrainAs(text) {
17             top.linkTo(button1.bottom, margin = 16.dp)
18             centerAround(button1.end)
19         })
20         // 根据button1和text的end（取其长）创建barrier
21         val barrier = createEndBarrier(button1, text)
22         // Button2显示在barrier开始处
23         Button(
24             onClick = {},
25             modifier = Modifier.constrainAs(button2) {
```

```
26         top.linkTo(parent.top, margin = 16.dp)
27         start.linkTo(barrier)
28     }
29     ) {
30         Text("Button2")
31     }
32 }
33 }
```



```
1  @Composable
2  fun ConstraintLayoutDemo() {
3      ConstraintLayout(
4          modifier = Modifier.fillMaxSize()
5      ) {
6
7          //在该布局中从开始（左侧）的百分之20的位置，创建一条竖向引导线。
8          val guideline = createGuidelineFromStart(0.2f)
9          val (box1, box2) = createRefs()
10
11         Box( // 创建了两个Box布局，分别是结束部分链接到引导线，开始部分链接到引
            导线。
12             modifier = Modifier.fillMaxSize()
13                 .background(color = Color.Yellow)
14                 .constrainAs(box1) {
15                     end.linkTo(guideline)
16                 }
17         )
18
19         Box(
20             modifier = Modifier.fillMaxSize()
```

```
21         .background(color = Color.Red)
22         .constrainAs(box2) {
23             start.linkTo(guideline)
24         }
25     )
26 }
```



第五章 Compose动画

5.1 Compose SideEffect

简单的可组合函数，使用 Compose，可以通过定义一组接受数据而发出界面元素的可组合函数来构建界面。一个简单的示例是 `Greeting` 微件，它接受 `String` 而发出一个显示问候消息的 `Text` 微件。



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

关于此函数，有几点值得注意：

- 此函数带有 `@Composable` 注释。所有可组合函数都必须带有此注释；此注释可告知 `Compose` 编译器：此函数旨在将数据转换为界面。
- 此函数接受数据。可组合函数可以接受一些参数，这些参数可让应用逻辑描述界面。在本例中，我们的微件接受一个 `String`，因此它可以按名称问候用户。
- 此函数可以在界面中显示文本。为此，它会调用 `Text()` 可组合函数，该函数实际上会创建文本界面元素。可组合函数通过调用其他可组合函数来发出界面层次结构。
- 此函数不会返回任何内容。发出界面的 `Compose` 函数不需要返回任何内容，因为它们描述所需的屏幕状态，而不是构造界面微件。
- 此函数快速、幂等且没有副作用。
 - 使用同一参数多次调用此函数时，它的行为方式相同，并且它不使用其他值，如全局变量或对 `random()` 的调用。
 - 此函数描述界面而没有任何副作用，如修改属性或全局变量。

一般来说，出于重组部分所述的原因，所有可组合函数都应使用这些属性来编写。

在计算机学术的概念中，我们需要关注函数的副作用。函数副作用是指当调用函数时，除了返回函数值之外，还对主调用函数产生附加的影响。副作用的函数不仅仅只是返回了一个值，而且还做了其他的事情，比如：

- 1、修改了一个变量
- 2、直接修改数据结构
- 3、设置一个对象的成员
- 4、抛出一个异常或以一个错误终止
- 5、打印到终端或读取用户输入
- 6、读取或写入一个文件
- 7、在屏幕上画图

函数副作用会给程序设计带来不必要的麻烦，给程序带来十分难以查找的错误，并且降低程序的可读性，严格的函数式语言要求函数必须无副作用。

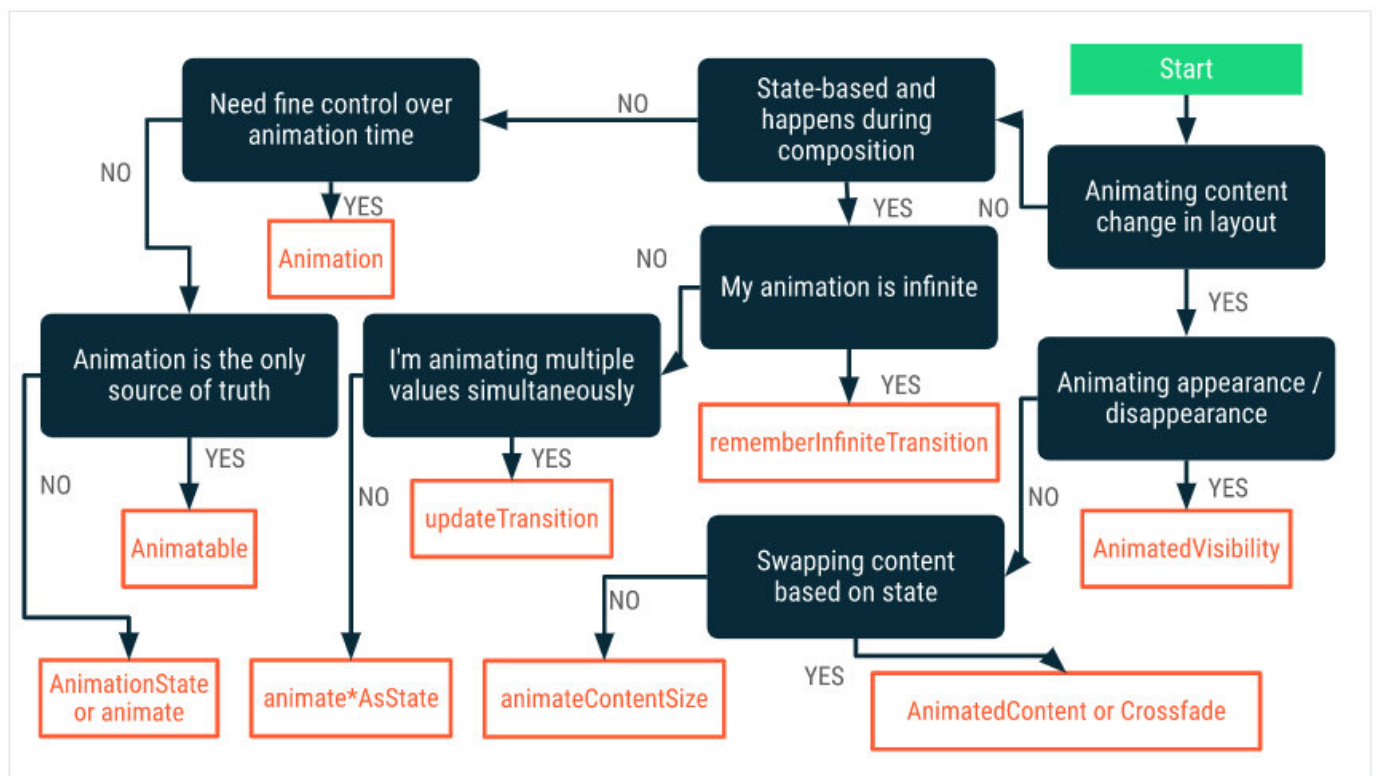
纯函数 (Pure Function)

- | | |
|---|---|
| 1 | 纯函数是这样一种函数，即相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用。相同输入必定能得出相同输出。 |
| 2 | 函数执行过程中没有任何副作用。 |

我们来看一下**SideEffect**的主要作用：将 Compose 状态发布为非 Compose 代码。**SideEffect**相当于**DisposableEffect**的简化版，当不需要onDispose、不需要参数控制（即每次onCommit都执行）时使用，等价于

```
1 DisposableEffect() {  
2     // TODO: handle some side effects  
3     onDispose{/*do nothing*/}  
4 }
```

5.2 Compose 动画概述



我们来看几个核心重点的函数：

AnimatedVisibility: `AnimatedVisibility` 可组合项可为内容的出现和消失添加动画效果。

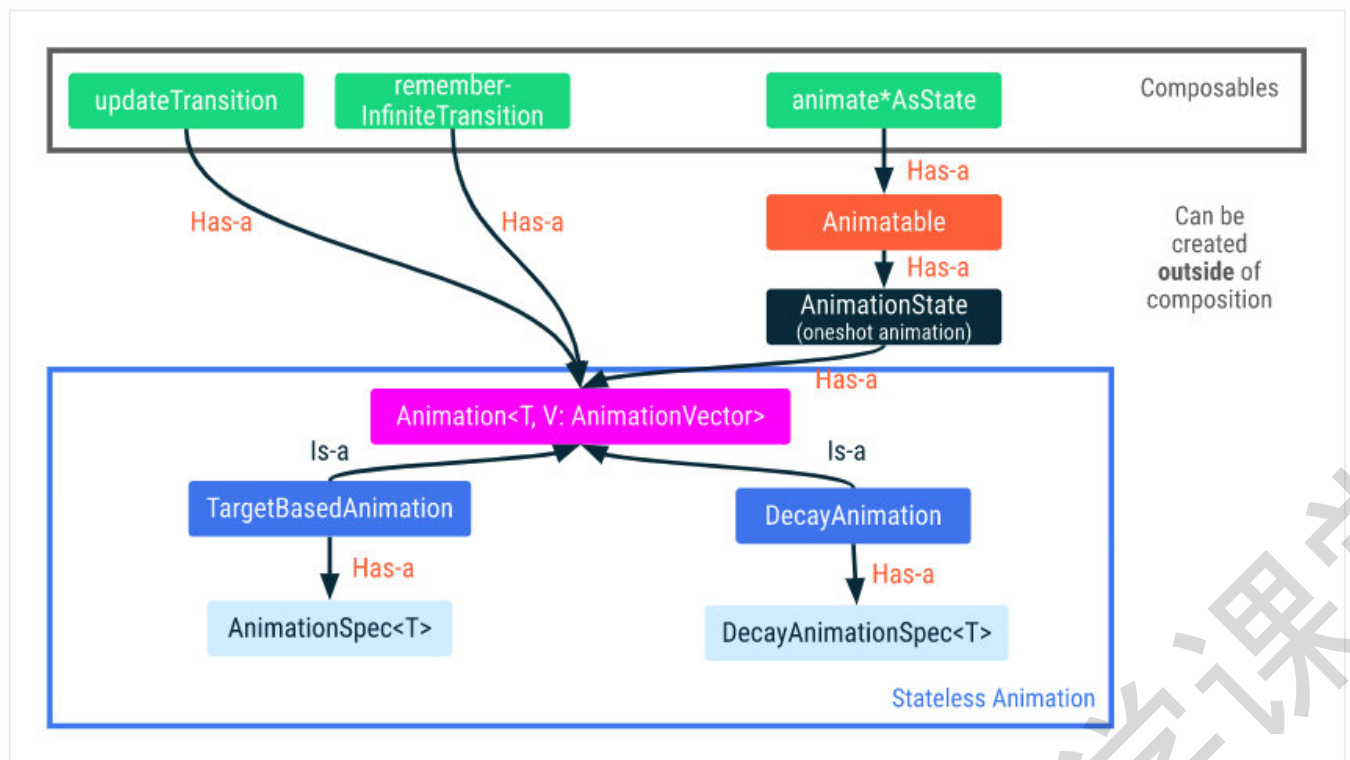
```
1 var editable by remember { mutableStateOf(true) }  
2 AnimatedVisibility(visible = editable) {  
3     Text(text = "Edit")  
4 }
```

```
1 var visible by remember { mutableStateOf(true) }  
2 val density = LocalDensity.current  
3 AnimatedVisibility(  
4     visible = visible,
```

```
5   enter = slideInVertically(  
6       // Slide in from 40 dp from the top.  
7       initialOffsetY = { with(density) { -40.dp.roundToPx() } }  
8   ) + expandVertically(  
9       // Expand from the top.  
10      expandFrom = Alignment.Top  
11  ) + fadeIn(  
12      // Fade in with the initial alpha of 0.3f.  
13      initialAlpha = 0.3f  
14  ),  
15  exit = slideOutVertically() + shrinkVertically() + fadeOut()  
16  ) {  
17      Text("Hello", Modifier.fillMaxWidth().height(200.dp))  
18  }
```

animateContentSize:修饰符可为大小变化添加动画效果。

```
1  var message by remember { mutableStateOf("Hello") }  
2  Box(  
3      modifier = Modifier.background(Color.Blue).animateContentSize()  
4  ) {  
5      Text(text = message)  
6  }
```



所有这些 API 都基于更基础的 **Animation** API。虽然大多数应用不会直接与 **Animation** 互动，但 **Animation** 的某些自定义功能可以通过更高级别的 API 获得。

animate*AsState: 是 Compose 中最简单的动画 API，用于为单个值添加动画效果。您只需提供结束值（或目标值），该 API 就会从当前值开始向指定值播放动画。

```
1 val alpha: Float by animateFloatAsState(if (enabled) 1f else 0.5f)
2 Box(
3     Modifier.fillMaxSize()
4         .graphicsLayer(alpha = alpha)
5         .background(Color.Red)
6 )
```

Compose 为 **Float**、**Color**、**Dp**、**Size**、**Offset**、**Rect**、**Int**、**IntOffset** 和 **IntSize** 提供开箱即用的 **animate*AsState** 函数。通过为接受通用类型的 **animateValueAsState** 提供 **TwoWayConverter**，您可以轻松添加对其他数据类型的支持。

Animatable: 是一个值容器，它可以在通过 **animateTo** 更改值时为值添加动画效果。该 API 支持 **animate*AsState** 的实现。它可确保一致的连续性和互斥性，这意味着值变化始终是连续的，并且会取消任何正在播放的动画。

```
1 // Start out gray and animate to green/red based on `ok`
2 val color = remember { Animatable(Color.Gray) }
3 LaunchedEffect(ok) {
4     color.animateTo(if (ok) Color.Green else Color.Red)
5 }
6 Box(Modifier.fillMaxSize().background(color.value))
```

5.3 Compose Crossfade

Crossfade 可以通过监听状态值的变化，使用淡入淡出的动画在两个布局之间添加动画效果，函数自身就是一个 **Composable**，代码如下：

```
1 //Crossfade 淡入淡出动画
2 var fadeStatus by remember { mutableStateOf(true) }
3
4 Column(
5     Modifier
6         .fillMaxWidth()
7         .fillMaxHeight(),
8     Arrangement.Top,
9     Alignment.CenterHorizontally
```

```
10     ) {
11         Button(
12             onClick = { fadeStatus = !fadeStatus }
13         ) {
14             Text(text = if (fadeStatus) "Fade In" else "Fade Out")
15         }
16
17         Spacer(Modifier.height(16.dp))
18
19
20         //布局的切换
21         Crossfade(targetState = fadeStatus, animationSpec =
tween(3000)) { screen ->
22             when (screen) {
23                 true -> Image(
24                     painter = painterResource(id = R.drawable.test),
25                     contentDescription = null,
26                     Modifier
27                         .animateContentSize()
28                         .size(300.dp)
29                 )
30                 false -> Image(
31                     painter = painterResource(id = R.drawable.test2),
32                     contentDescription = null,
33                     Modifier
34                         .animateContentSize()
35                         .size(300.dp)
36                 )
37             }
38         }
39     }
```

效果可以看出来，2个图片来回的切换，形成场景切换的动画感。

5.4 Compose animateContentSize

先看下函数的定义：

```
1 fun Modifier.animateContentSize(
2     animationSpec: FiniteAnimationSpec<IntSize> = spring(),
3     finishedListener: ((initialValue: IntSize, targetValue: IntSize) -
> Unit)? = null
4 )
```

可以为布局大小动画设置动画速度和监听值。由函数的定义可以看出这个函数本质上就 **Modifier** 的一个扩展函数。可以通过变量 **size** 监听状态变化实现布局大小的动画效果，代码如下：

```
1 //放大缩小动画 animateContentSize
2 var size by remember { mutableStateOf(Size(300F, 300F)) }
3
4 Column(
5     Modifier
6         .fillMaxWidth()
7         .fillMaxHeight(),
8     Arrangement.Top,
9     Alignment.CenterHorizontally
10 ) {
11     Spacer(Modifier.height(16.dp))
12
13     Button(
14         onClick = {
15             size = if (size.height == 300F) {
16                 Size(500F, 500F)
17             } else {
18                 Size(300F, 300F)
19             }
20         }
21     ) {
22         Text(if (size.height == 300F) "Shrink" else "Expand")
23     }
24     Spacer(Modifier.height(16.dp))
25
26     Box(
27         Modifier
28             .animateContentSize()
29     ) {
30         Image(
31             painter = painterResource(id = R.drawable.pikaqiu),
32             contentDescription = null,
33             Modifier
34                 .animateContentSize()
35                 .size(size = size.height.dp)
36         )
37     }
38 }
```

5.5 Animatable

Animatable 是一个值容器，它可以在通过 **animateTo** 更改值时为值添加动画效果。该 API 支持 **animate*AsState** 的实现。它可确保一致的连续性和互斥性，这意味着值变化始终是连续的，并且会取消任何正在播放的动画。

```
1 // Start out gray and animate to green/red based on `ok`
2 val color = remember { Animatable(Color.Gray) }
3 LaunchedEffect(ok) {
4     color.animateTo(if (ok) Color.Green else Color.Red)
5 }
6 Box(Modifier.fillMaxSize().background(color.value))
```

This **Animatable** function creates a **Color** value holder that automatically animates its value when the value is changed via **animateTo**. **Animatable** supports value change during an ongoing value change animation. When that happens, a new animation will transition **Animatable** from its current value (i.e. value at the point of interruption) to the new target. This ensures that the value change is always continuous using **animateTo**. If **spring** animation (i.e. default animation) is used with **animateTo**, the velocity change will be guaranteed to be continuous as well.

Unlike **AnimationState**, **Animatable** ensures mutual exclusiveness on its animation. To do so, when a new animation is started via **animateTo** (or **animateDecay**), any ongoing animation job will be cancelled via a **CancellationException**.

Animatable also supports animating data types other than **Color**, such as **Floats** and generic types. See [androidx.compose.animation.core.Animatable](#) for other variants.

Params: **initialValue** – initial value of the **Animatable**

Samples: [androidx.compose.animation.samples.AnimatableColor](#)
// Unresolved

```
fun Animatable(initialValue: Color): Animatable<Color, AnimationVector4D> =
    Animatable(initialValue, (Color.VectorConverter)(initialValue.colorSpace))
```

Compose的动画API，基本上相对比较清晰，可以通过值的变化来，达到动画的加载。

5.6 Compose自定义动画

AnimationSpec

AnimationSpec 可以自定义动画的行为，效果类似于原生动画中的估值器。

```
1 @Composable
2 fun animSpring() {
3     val state = remember {
4         mutableStateOf(true)
5     }
6     var value = animateIntAsState(
7         targetValue = if (state.value) 300 else 100,
8         animationSpec = spring(
9             dampingRatio = Spring.DampingRatioHighBouncy,
```

```
10         stiffness = Spring.StiffnessVeryLow
11     )
12 )
13
14 Box(
15     Modifier
16         .fillMaxSize(1f)
17         .padding(start = 30.dp), contentAlignment =
Alignment.CenterStart
18 ) {
19     Box(
20         Modifier
21             .width(value.value.dp)
22             .height(80.dp)
23             .background(Color.Red, RoundedCornerShape(topEnd =
30.dp, bottomEnd = 30.dp))
24             .clickable {
25                 state.value = !state.value
26             }, contentAlignment = Alignment.CenterStart
27     ) {
28         Text(text = "测试AnimationSpec", style = TextStyle(color =
Color.White, fontSize = 20.sp))
29     }
30 }
31 }
```

```
1 @Composable
2 fun animTweenSpec() {
3     val state = remember {
4         mutableStateOf(true)
5     }
6     val value = animateIntAsState(
7         targetValue = if (state.value) 300 else 100,
8         animationSpec = tween(
9             durationMillis = 1500,
10            delayMillis = 200,
11            easing = LinearEasing
12        )
13    )
14
15    Box(
16        Modifier
17            .fillMaxSize(1f)
```



```
18         .padding(start = 50.dp), contentAlignment =
Alignment.CenterStart
19     ) {
20         Box(
21             Modifier
22                 .width(value.value.dp)
23                 .height(100.dp)
24                 .background(Color.Red, RoundedCornerShape(topEnd =
30.dp, bottomEnd = 30.dp))
25                 .clickable {
26                     state.value = !state.value
27                 }
28         ) {
29
30     }
31 }
32 }
```

[AnimationSpec](#) stores the specification of an animation, including 1) the data type to be animated, and 2) the animation configuration (i.e. [VectorizedAnimationSpec](#)) that will be used once the data (of type [T](#)) has been converted to [AnimationVector](#).

Any type [T](#) can be animated by the system as long as a [TwoWayConverter](#) is supplied to convert the data type [T](#) from and to an [AnimationVector](#). There are a number of converters available out of the box. For example, to animate [androidx.compose.ui.unit.IntOffset](#) the system uses [IntOffset.VectorConverter](#) to convert the object to [AnimationVector2D](#), so that both x and y dimensions are animated independently with separate velocity tracking. This enables multidimensional objects to be animated in a true multi-dimensional way. It is particularly useful for smoothly handling animation interruptions (such as when the target changes during the animation).

```
interface AnimationSpec<T> {
```

Creates a [VectorizedAnimationSpec](#) with the given [TwoWayConverter](#).

The underlying animation system operates on [AnimationVectors](#). [T](#) will be converted to [AnimationVector](#) to animate. [VectorizedAnimationSpec](#) describes how the converted [AnimationVector](#) should be animated. E.g. The animation could simply interpolate between the start and end values (i.e. [TweenSpec](#)), or apply spring physics to produce the motion (i.e. [SpringSpec](#)), etc)

Params: converter – converts the type [T](#) from and to [AnimationVector](#) type

```
fun <V : AnimationVector> vectorize(
    converter: TwoWayConverter<T, V>
): VectorizedAnimationSpec<V>
```

```
}
```

大多数 Compose 动画 API 都支持将 Float、Color、Dp 以及其他基本数据类型作为开箱即用的动画值，但有时我们需要为其他数据类型（包括我们的自定义类型）添加动画效果。

```
1 @Composable
2 fun animAnimationVector() {
3     var state by remember {
4         mutableStateOf(true)
5     }
6     val value by animateValueAsState(
7         targetValue = if (state) AnimSize(0xffff5500, 100f) else
AnimSize(0xff00ff00, 300f),
8         typeConverter = TwoWayConverter(
9             convertToVector = {
10 //                 AnimationVector2D(target.color.toFloat(),
target.size)
11                 AnimationVector2D(it.color.toFloat(), it.size)
12             },
13             convertFromVector = {
14                 AnimSize(it.v1.toLong(), it.v2)
15             }
16         )
17     )
18     println("颜色:${value.color}")
19     Box(modifier = Modifier.fillMaxSize(1f).padding(30.dp),
contentAlignment = Alignment.Center) {
20         Box(
21             modifier = Modifier
22                 .size(value.size.dp)
23 //                 .size(300.dp)
24                 .background(Color(value.color),
RoundedCornerShape(30.dp))
25                 .clickable {
26                     state = !state
27                 }
28         ) {
29             }
30         }
31     }
32 }
33 data class AnimSize(val color: Long, val size: Float)
```

第六章 Compose图形

6.1Compose Canvas

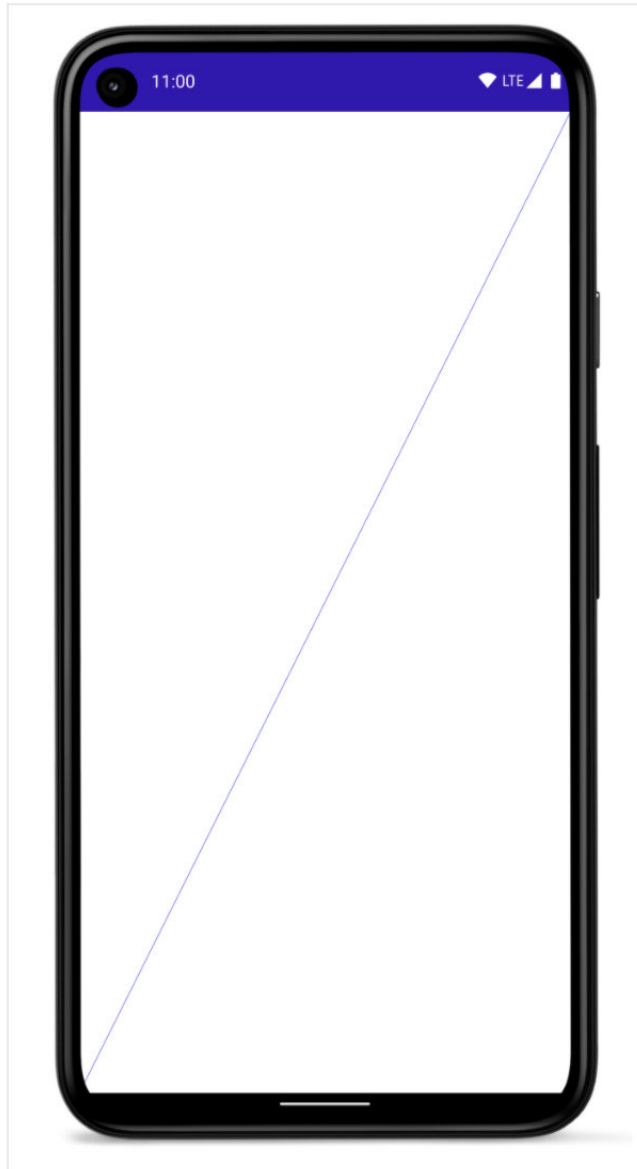
自定义图形的核心可组合项是 Canvas。在布局中放置 **Canvas** 的方式与放置其他 Compose 界面元素相同。在 **Canvas** 中，您可以通过精确控制元素的样式和位置来绘制元素。

以下代码将创建一个 **Canvas** 可组合项，用于填充其父元素中的所有可用空间：

```
1 Canvas(modifier = Modifier.fillMaxSize()) {  
2 }
```

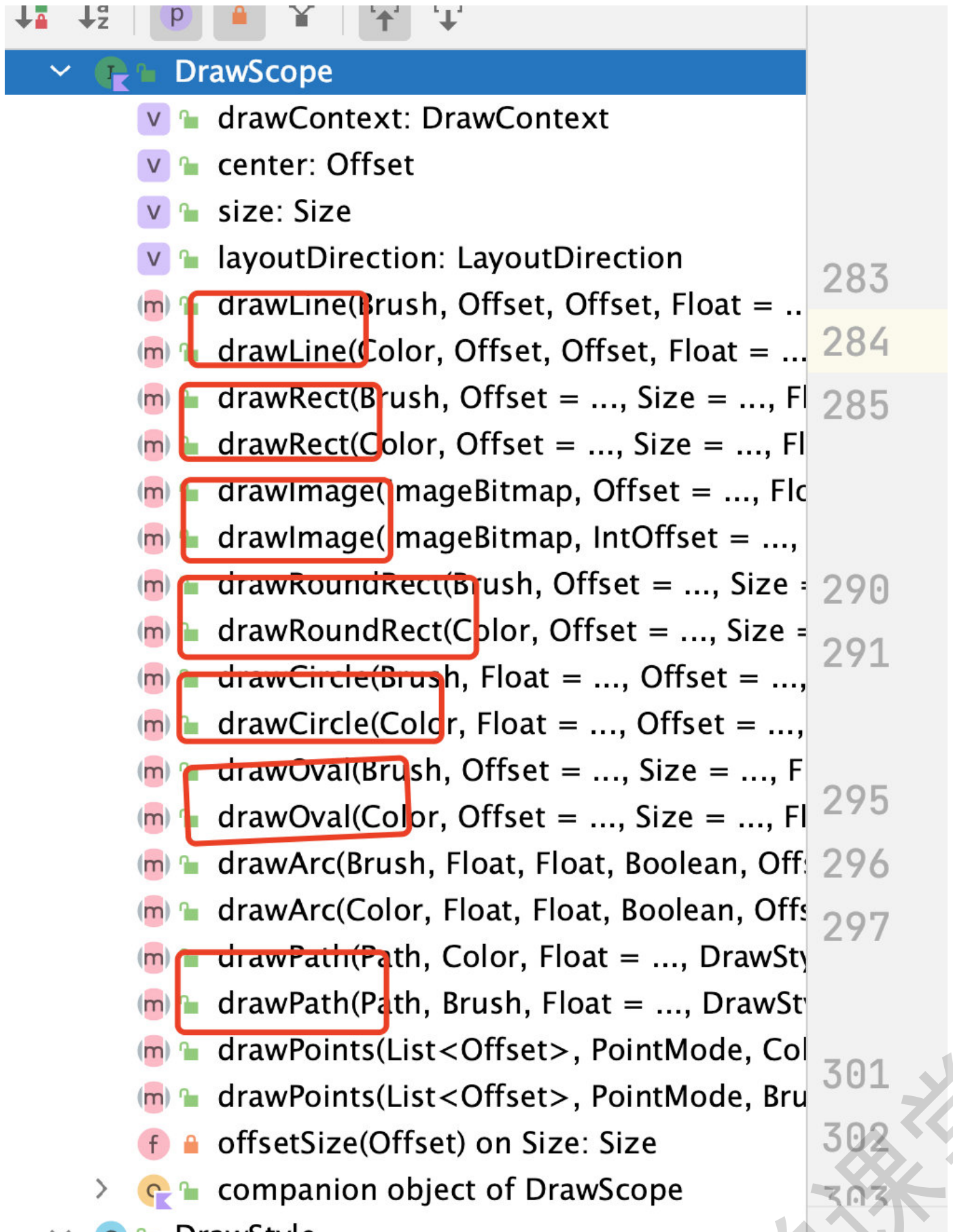
Canvas 会自动提供 DrawScope（一个维护自身状态且限定了作用域的绘图环境）。这让您可以为一组图形元素设置参数。**DrawScope** 提供了几个有用的字段，例如 **size**，一个指定 **DrawScope** 的当前维度和最大维度的 **Size** 对象。例如，假设您想绘制一条从画布右上角到左下角的对角线。这可以通过添加 drawLine 可组合项来实现：

```
1 Canvas(modifier = Modifier.fillMaxSize()) {  
2     val canvasWidth = size.width  
3     val canvasHeight = size.height  
4  
5     drawLine(  
6         start = Offset(x = canvasWidth, y = 0f),  
7         end = Offset(x = 0f, y = canvasHeight),  
8         color = Color.Blue  
9     )  
10 }
```

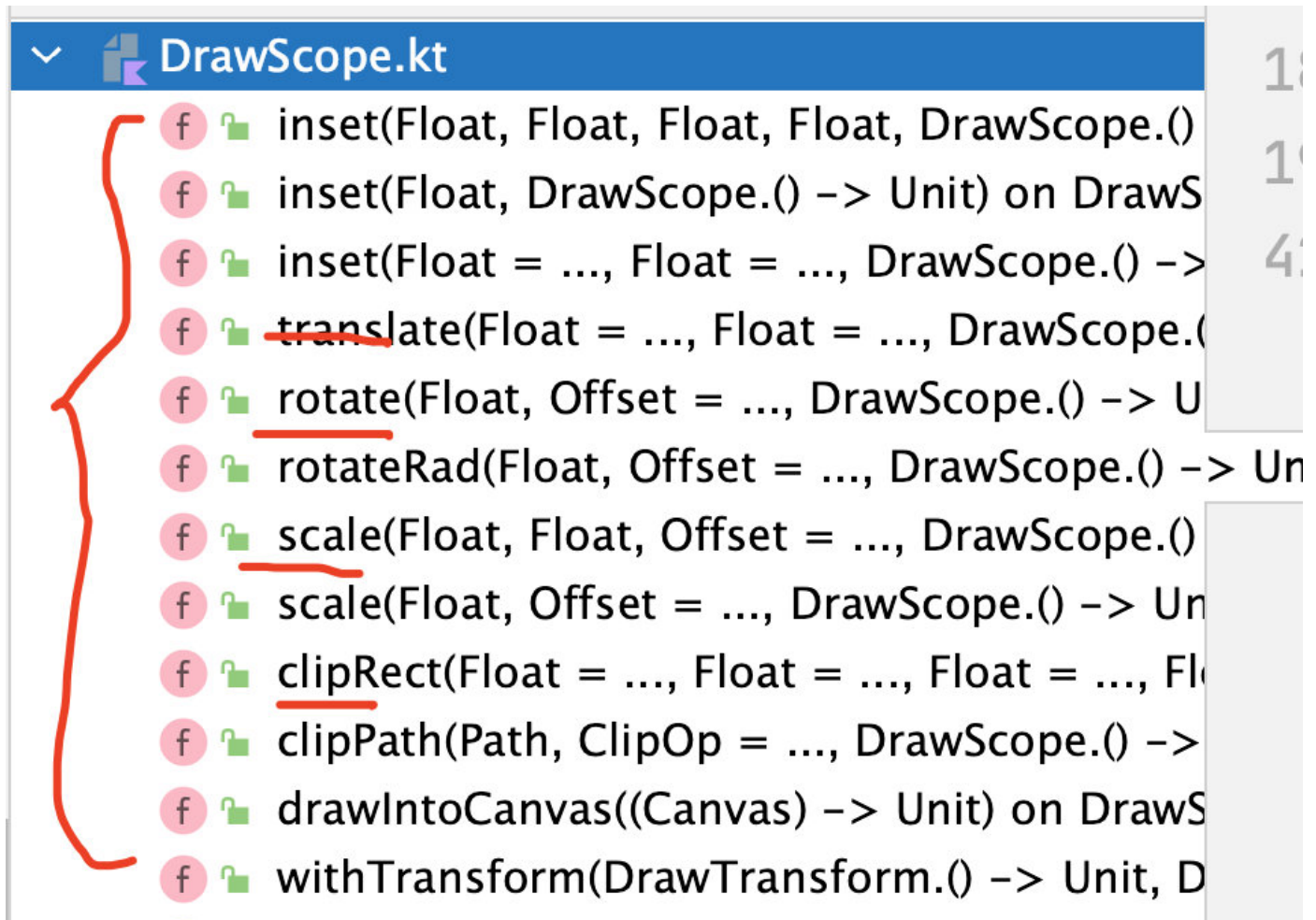


现在来看看相关的API:

享学课堂



可以看到每个 Compose Canvas 提供一个 DrawScope（限定了作用域的绘图环境），您可以在其中实际发出绘图命令。我们在来看看：



```
1 withTransform({
2     translate(left = canvasWidth / 5F)
3     rotate(degrees = 45F)
4 }) {
5     drawRect(
6         color = Color.Gray,
7         topLeft = Offset(x = canvasWidth / 3F, y = canvasHeight /
8 3F),
9         size = canvasSize / 3F
10    )
11 }
```

所以上面的代码用于向矩形同时应用平移和旋转，使用 `withTransform()` 函数，该函数用于创建并应用整合了您所需的所有更改的单一转换。与对各个转换进行嵌套调用相比，使用 `withTransform()` 更有效，因为这种情况下所有转换都在单个操作中一起执行，Compose 不必计算并保存每个嵌套转换。

6.2 Compose 绘制API的分析

Canvas Composable 是官方提供的一个专门来自定义绘制的独立组件，这个组件不包含任何子元素，类似于传统View系统中的一个独立View（不是ViewGroup，不包含子View）。Canvas参数有两个参数，类型分别是 Modifier 与 DrawScope() -> Unit。Modifier 作为该组件的修饰符不难理解， DrawScope() -> Unit 是一个 reciever 为 DrawScope 类型的 lambda。那么我们就可以在 lambda 中任意使用 DrawScope 为我们所提供的 API 了。

```
1 fun Canvas(modifier: Modifier, onDraw: DrawScope() -> Unit)
```

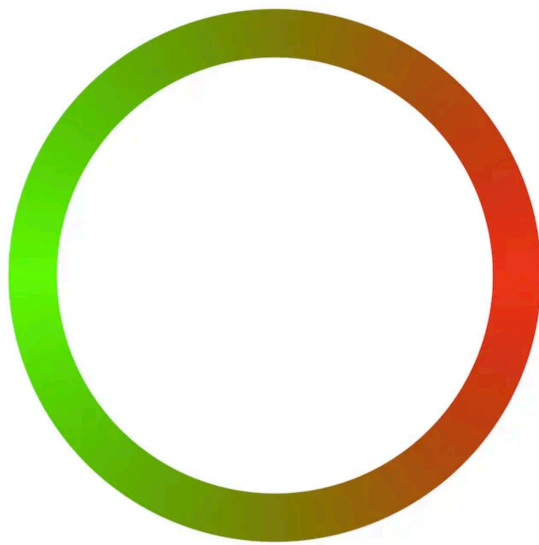
我们来看看 DrawScope 为我们限定了哪些 API。可以参见下表：

API	描述
drawLine	绘制一条线
drawRect	绘制一个矩形
drawImage	绘制一张图片
drawRoundRect	绘制一个圆角矩形
drawCircle	绘制一个圆
drawOval	绘制一个椭圆
drawArc	绘制一条弧线
drawPath	绘制一条路径
drawPoints	绘制一些点

```
1 @Preview
2 @Composable
3 fun DrawColorRing() {
4     Box(
5         modifier = Modifier.fillMaxSize(),
6         contentAlignment = Alignment.Center
7     ) {
8         var radius = 300.dp
```



```
9      var ringWidth = 30.dp
10      Canvas(modifier = Modifier.size(radius)) {
11          this.drawCircle(
12              brush = Brush.sweepGradient(listOf(Color.Red,
13              Color.Green, Color.Red), Offset(radius.toPx() / 2f, radius.toPx() /
14              2f)),
15              radius = radius.toPx() / 2f,
16              style = Stroke(
17                  width = ringWidth.toPx()
18              )
19          )
20      }
```



6.3 Compose自定义绘制

对于自定义绘制，官方为我们提供了三个 Modifier API，分别是 drawWithContent、drawBehind、drawWithCache。

drawWithContent:

drawWithContent 需要一个Receiver为 ContentDrawScope 类型的lambda，而这个ContentDrawScope 拓展了 DrawScope 的能力，多了个 drawContent API。这个 API 是提供给开发者来控制绘制层级的。

```
1 fun Modifier.drawWithContent(  
2     onDraw: ContentDrawScope.() -> Unit  
3 )  
4  
5 interface ContentDrawScope : DrawScope {  
6     /**  
7      * Causes child drawing operations to run during the `onPaint`  
8      lambda.  
9      */  
10    fun drawContent()  
11 }
```

drawBehind:

drawBehind, 画在后面。具体画在谁后面呢, 具体画在他所修饰的UI组件后面。

```
1 fun Modifier.drawBehind(  
2     onDraw: DrawScope.() -> Unit  
3 ) = this.then(  
4     DrawBackgroundModifier(  
5         onDraw = onDraw, // onDraw 为我们定制的绘制控制逻辑  
6         ...  
7     )  
8 )  
9  
10 private class DrawBackgroundModifier(  
11     val onDraw: DrawScope.() -> Unit,  
12     ...  
13 ) : DrawModifier, InspectorValueInfo(inspectorInfo) {  
14     override fun ContentDrawScope.draw() {  
15         onDraw() // 先画我们定制的绘制控制逻辑  
16         drawContent() // 后画UI组件本身  
17     }  
18     ...  
19 }
```

```
1 @Preview  
2 @Composable  
3 fun DrawBehind() {  
4     Box(  
5         modifier = Modifier.fillMaxSize(),  
6         contentAlignment = Alignment.Center  
7     ) {
```

```
8         Card(  
9             shape = RoundedCornerShape(8.dp)  
10            ,modifier = Modifier  
11                .size(100.dp)  
12                .drawBehind {  
13                    drawCircle(Color(0xffe7614e), 18.dp.toPx() / 2,  
center = Offset(drawContext.size.width, 0f))  
14                }  
15            ) {  
16                Image(painter = painterResource(id = R.drawable.diana),  
contentDescription = "")  
17            }  
18        }  
19    }
```

drawWithCache:

有些时候我们绘制一些比较复杂的UI效果时，不希望当 **Recompose** 发生时所有绘画所用的所有实例都重新构建一次（类似Path），这可能会产生内存抖动。在 **Compose** 中我们一般能够想到使用 **remember** 进行缓存，然而我们所绘制的作用域是 **DrawScope** 并不是 **Composable**，所以无法使用 **remember**，那我们该怎么办呢？**drawWithCache** 提供了这个能力。

```
1 fun Modifier.drawWithCache(  
2     onBuildDrawCache: CacheDrawScope.() -> DrawResult  
3 )  
4  
5 class CacheDrawScope internal constructor() : Density {  
6     ...  
7     fun onDrawBehind(block: DrawScope.() -> Unit): DrawResult  
8     fun onDrawWithContent(block: ContentDrawScope.() -> Unit):  
DrawResult  
9     ...  
10 }
```

案例实现自定义view:

```
1 //水波纹的演示  
2 @Composable  
3 fun WaveLoadingView(  
4     modifier: Modifier,  
5     text: String,  
6     textSize: TextUnit,
```

```
7     waveColor: Color,
8     downTextColor: Color,
9 ) {
10     BoxWithConstraints(modifier = modifier) {
11         val circleSizeDp = minOf(maxWidth, maxHeight)
12         val density = LocalDensity.current.density
13         val circleSizePx = circleSizeDp.value * density
14         val waveWidth = circleSizePx / 3f
15         val waveHeight = circleSizePx / 26f
16         val textPaint by remember {
17             mutableStateOf(Paint().asFrameworkPaint().apply {
18                 isAntiAlias = true
19                 isDither = true
20                 typeface = Typeface.create(Typeface.SANS_SERIF,
Typeface.BOLD)
21                 textAlign = android.graphics.Paint.Align.CENTER
22             })
23         }
24         val wavePath by remember {
25             mutableStateOf(Path())
26         }
27         val circlePath by remember {
28             mutableStateOf(Path().apply {
29                 addArc(
30                     Rect(0f, 0f, circleSizePx, circleSizePx),
31                     0f, 360f
32                 )
33             })
34         }
35         val animateValue by
rememberInfiniteTransition().animateFloat(
36             initialValue = 0f, targetValue = waveWidth,
37             animationSpec = infiniteRepeatable(
38                 animation = tween(durationMillis = 500, easing =
LinearEasing),
39                 repeatMode = RepeatMode.Restart,
40             ),
41         )
42
43         Canvas(modifier = modifier.requiredSize(size =
circleSizeDp)) {
44             drawIntoCanvas {
45                 textPaint.textSize = textSize.toPx()
46                 drawText(
```

```
47         canvas = it.nativeCanvas,
48         size = circleSizePx,
49         textPaint = textPaint,
50         textColor = waveColor.toArgb(),
51         text = text
52     )
53 }
54 wavePath.reset()
55 wavePath.moveTo(-waveWidth + animateValue, circleSizePx
/ 2.2f)
56     var i = -waveWidth
57     while (i < circleSizePx + waveWidth) {
58         wavePath.relativeQuadraticBezierTo(waveWidth / 4f, -
waveHeight, waveWidth / 2f, 0f)
59         wavePath.relativeQuadraticBezierTo(waveWidth / 4f,
waveHeight, waveWidth / 2f, 0f)
60         i += waveWidth
61     }
62     wavePath.lineTo(circleSizePx, circleSizePx)
63     wavePath.lineTo(0f, circleSizePx)
64     wavePath.close()
65
66     val resultPath = Path.combine(
67         path1 = circlePath,
68         path2 = wavePath,
69         operation = PathOperation.Intersect
70     )
71     drawPath(path = resultPath, color = waveColor)
72
73     clipPath(path = resultPath, clipOp = ClipOp.Intersect) {
74         drawIntoCanvas {
75             drawText(
76                 canvas = it.nativeCanvas,
77                 size = circleSizePx,
78                 textPaint = textPaint,
79                 textColor = downTextColor.toArgb(),
80                 text = text
81             )
82         }
83     }
84 }
85 }
86 }
87 }
```

```
88 private fun drawText(  
89     canvas: NativeCanvas,  
90     size: Float,  
91     textPaint: android.graphics.Paint,  
92     textColor: Int,  
93     text: String  
94 ) {  
95     textPaint.color = textColor  
96     val fontMetrics = textPaint.fontMetrics  
97     val top = fontMetrics.top  
98     val bottom = fontMetrics.bottom  
99     val centerX = size / 2f  
100    val centerY = size / 2f - top / 2f - bottom / 2f  
101    canvas.drawText(text, centerX, centerY, textPaint)  
102 }
```

第七章 Compose核心控件总结

7.1 Scaffold

在Compose中提供了一种脚手架Scaffold的控件帮助开发者快速开发。在Scaffold中提供了很多配件，比如顶部菜单栏、侧滑菜单、底部菜单栏等。并且除了默认的Scaffold外，还有一些类似的控件，比如BackdropScaffold、BottomSheetScaffold等等。

```
1  @Composable  
2  fun Scaffold(  
3      modifier: Modifier = Modifier,  
4      scaffoldState: ScaffoldState = rememberScaffoldState(),  
5      topBar: () -> Unit = {},  
6      bottomBar: () -> Unit = {},  
7      snackbarHost: (SnackbarHostState) -> Unit = { SnackbarHost(it) },  
8      floatingActionButton: () -> Unit = {},  
9      floatingActionButtonPosition: FabPosition = FabPosition.End,  
10     isFloatingActionButtonDocked: Boolean = false,  
11     drawerContent: ColumnScope.() -> Unit = null,  
12     drawerGesturesEnabled: Boolean = true,  
13     drawerShape: Shape = MaterialTheme.shapes.large,  
14     drawerElevation: Dp = DrawerDefaults.Elevation,  
15     drawerBackgroundColor: Color = MaterialTheme.colors.surface,  
16     drawerContentColor: Color =  
17     contentColorFor(drawerBackgroundColor),  
18     drawerScrimColor: Color = DrawerDefaults.scrimColor,  
19     backgroundColor: Color = MaterialTheme.colors.background,  
20     contentColor: Color = contentColorFor(backgroundColor),
```

```
20     content: (PaddingValues) -> Unit
21 ): @Composable Unit
```

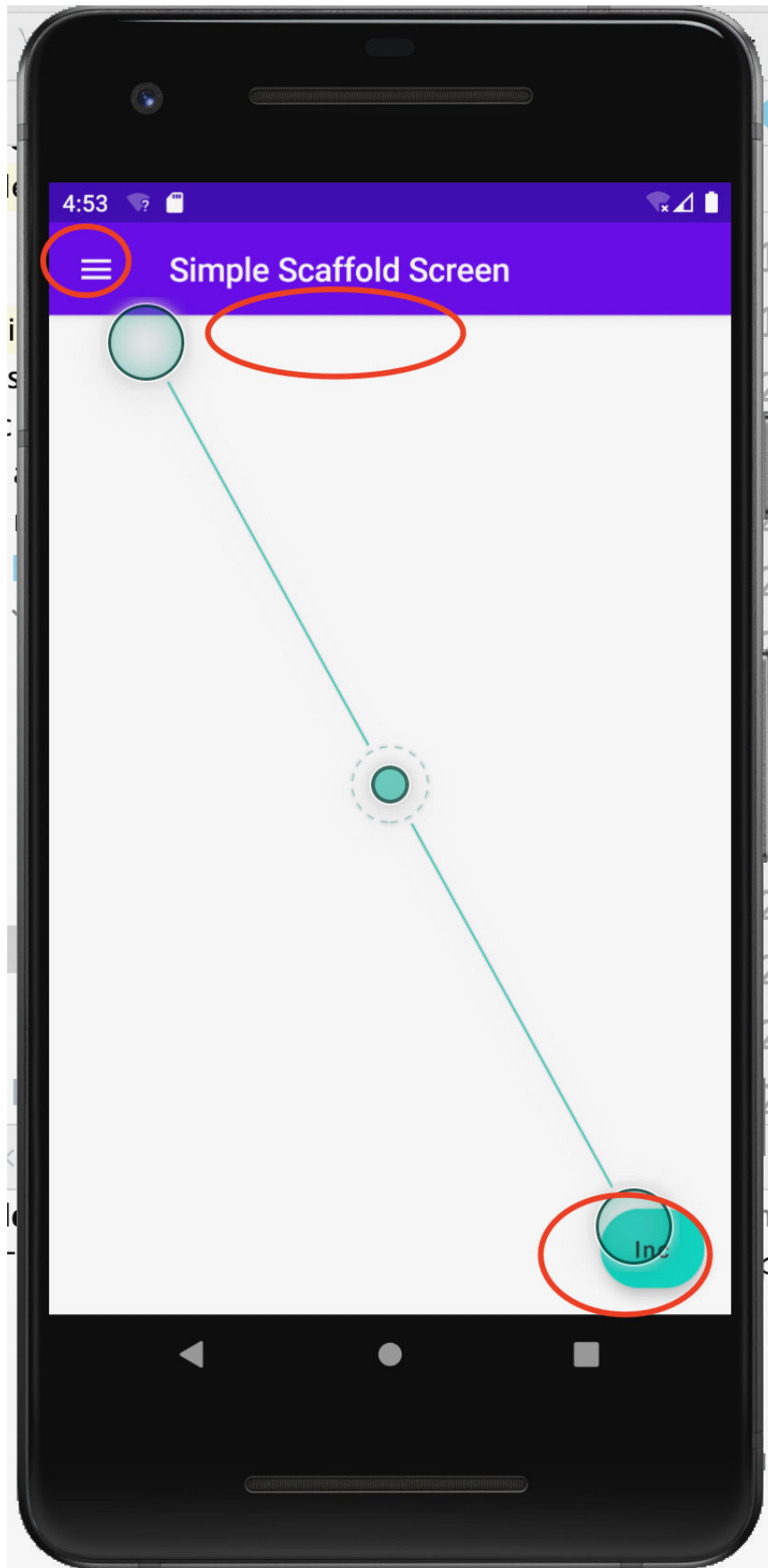
简单的代码如下：

```
1  @Composable
2  private fun scaffold(){
3      val scaffoldState = rememberScaffoldState()
4      val scope = rememberCoroutineScope()
5      Scaffold(
6          scaffoldState = scaffoldState,
7          drawerContent = { Text("Drawer content") },
8          topBar = {
9              TopAppBar(
10                 title = { Text("Simple Scaffold Screen") },
11                 navigationIcon = {
12                     IconButton(
13                         onClick = {
14                             scope.launch {
15                                 scaffoldState.drawerState.open() }
16                             }
17                             ) {
18                                 Icon(Icons.Filled.Menu, contentDescription =
19                                     "Localized description")
20                             }
21                         },
22                 floatingActionButtonPosition = FabPosition.End,
23                 floatingActionButton = {
24                     ExtendedFloatingActionButton(
25                         text = { Text("Inc") },
26                         onClick = { /* fab click handler */ }
27                     )
28                 },
29                 content = { innerPadding ->
30                     LazyColumn(contentPadding = innerPadding) {
31                         items(count = 100) {
32                             Box(
33                                 Modifier
34                                     .fillMaxWidth()
35                                     .height(50.dp)
36                                     .background(colors.background)
37                             )

```



```
38         }  
39     }  
40 }  
41 )  
42 }
```



7.2 LazyColumn

```
1 @Composable fun LazyColumn(  
2     modifier: Modifier = Modifier,  
3     state: LazyListState = rememberLazyListState(),  
4     contentPadding: PaddingValues = PaddingValues(0.dp),  
5     reverseLayout: Boolean = false,  
6     verticalArrangement: Arrangement.Vertical = if (!reverseLayout)  
Arrangement.Top else Arrangement.Bottom,  
7     horizontalAlignment: Alignment.Horizontal = Alignment.Start,  
8     content: LazyListScope.() -> Unit  
9 ): Unit
```

```
1 @ExperimentalMaterialApi  
2 @Composable  
3 fun ComposeListView() {  
4     val context: Context = LocalContext.current  
5     LazyColumn(){  
6         items(ImageDatas.size){  
7             index ->  
8             Card(onClick = {  
9                 Toast.makeText(context, "${index}",  
Toast.LENGTH_SHORT).show()},  
10                 shape = RoundedCornerShape(10.dp),  
11                 elevation = 10.dp, modifier =  
Modifier.padding(10.dp, 0.dp)  
12             ) {  
13                 //支持图片加载框架的控件coil  
14                 Image(  
15                     painter =  
rememberImagePainter(ImageDatas[index]),  
16                     contentDescription = null,  
17                     modifier =  
Modifier.height(220.dp).fillParentMaxWidth()  
18                 )  
19             }  
20         }  
21     }  
22 }
```

