> 写在前面的话：本大纲由享学课堂【路哥】编写，仅仅用于广大编程爱好者用于学习交流（为了辅助学习视频而编写的简要说明），禁止任何机构用于商业宣传，文档版权归享学课堂所有，欢迎大家来Android技术宇宙最强享学课堂进行Android技术的交流和学习。

Jetpack Compose组件编程指南

Jetpack Compose是什么?

我们来看看官网上的描述：

```
Jetpack Compose is Android's modern toolkit for building native UI. It
simplifies and accelerates UI development on Android. Quickly bring your app to
life with less code, powerful tools, and intuitive Kotlin APIs.
```

简单来说，Jetpack Compose 是Google发布的一个Android原生现代UI工具包，它完全采用Kotlin编写，可以使用Kotlin语言的全部特性，可以帮助你轻松、快速的构建高质量的Android应用程序。

- Jetpack Compose是一个声明式的UI框架。
- Compose基于Kotlin构建，因此可以与Java编程语言完全互操作，并且可以直接访问所有Android和Jetpack API。
- Compose主要的作用是为了快速和简单的绘制UI。

Jetpack Compose优点和优势

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

roadmap

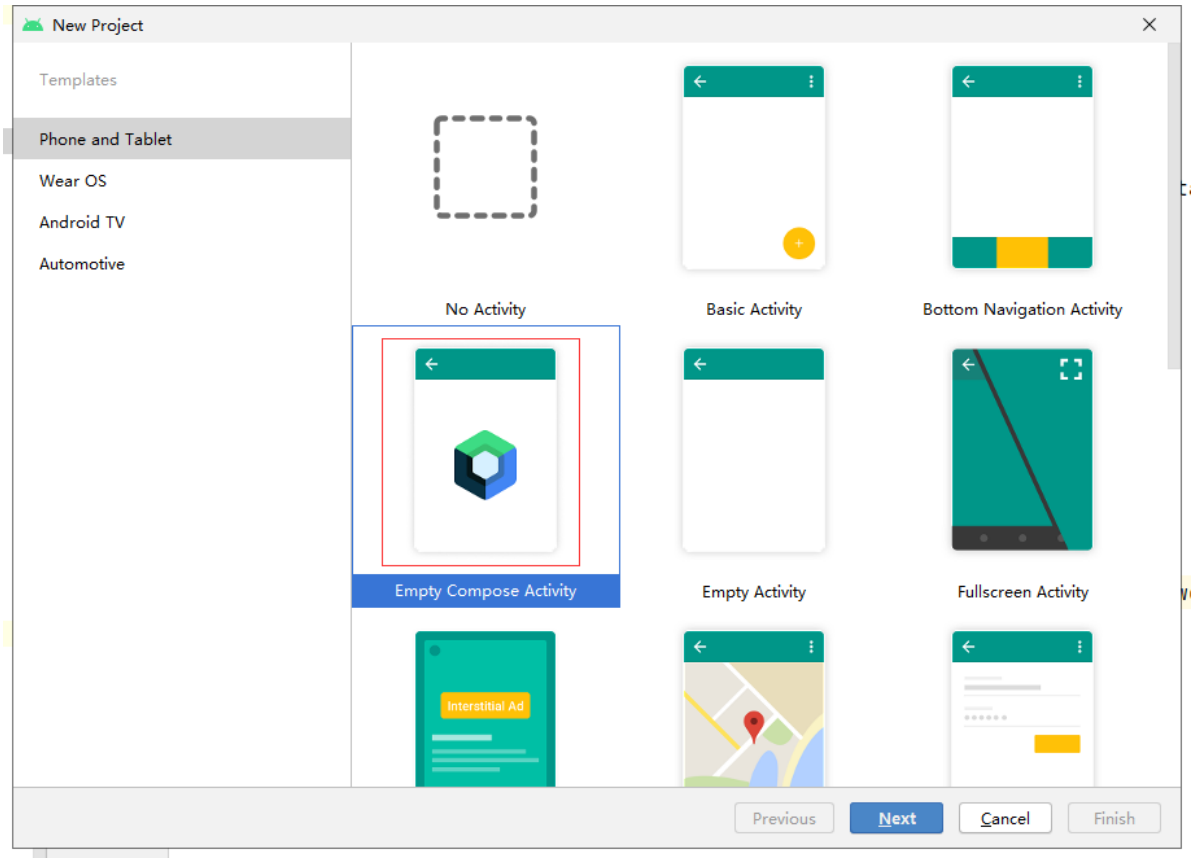
https://developer.android.com/jetpack/androidx/compose-roadmap?hl=ru

Jetpack Compose is currently at version 1.0. The major themes that we are focused on for our next releases are:

- Performance
- Material You components
- Large Screen improvements
- Homescreen Widgets
- WearOS support

Jetpack Compose 环境准备

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：

```
    kotlinOptions {
        jvmTarget = '1.8'
        useIR = true
    }
    buildFeatures {
        compose true
    }
    composeOptions {
        kotlinCompilerExtensionVersion compose_version
        kotlinCompilerVersion '1.5.10'
    }
    packagingOptions {
        resources {
            excludes += '/META-INF/{AL2.0,LGPL2.1}'
        }
    }
}

dependencies {

    implementation 'androidx.core:core-ktx:1.3.2'
    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'com.google.android.material:material:1.3.0'
    implementation "androidx.compose.ui:ui:$compose_version"
    implementation "androidx.compose.material:material:$compose_version"
    implementation "androidx.compose.ui:ui-tooling-preview:$compose_version"
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.3.1'
    implementation 'androidx.activity:activity-compose:1.3.0-alpha06'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
    androidTestImplementation "androidx.compose.ui:ui-test-junit4:$compose_version"
    debugImplementation "androidx.compose.ui:ui-tooling:$compose_version"
```

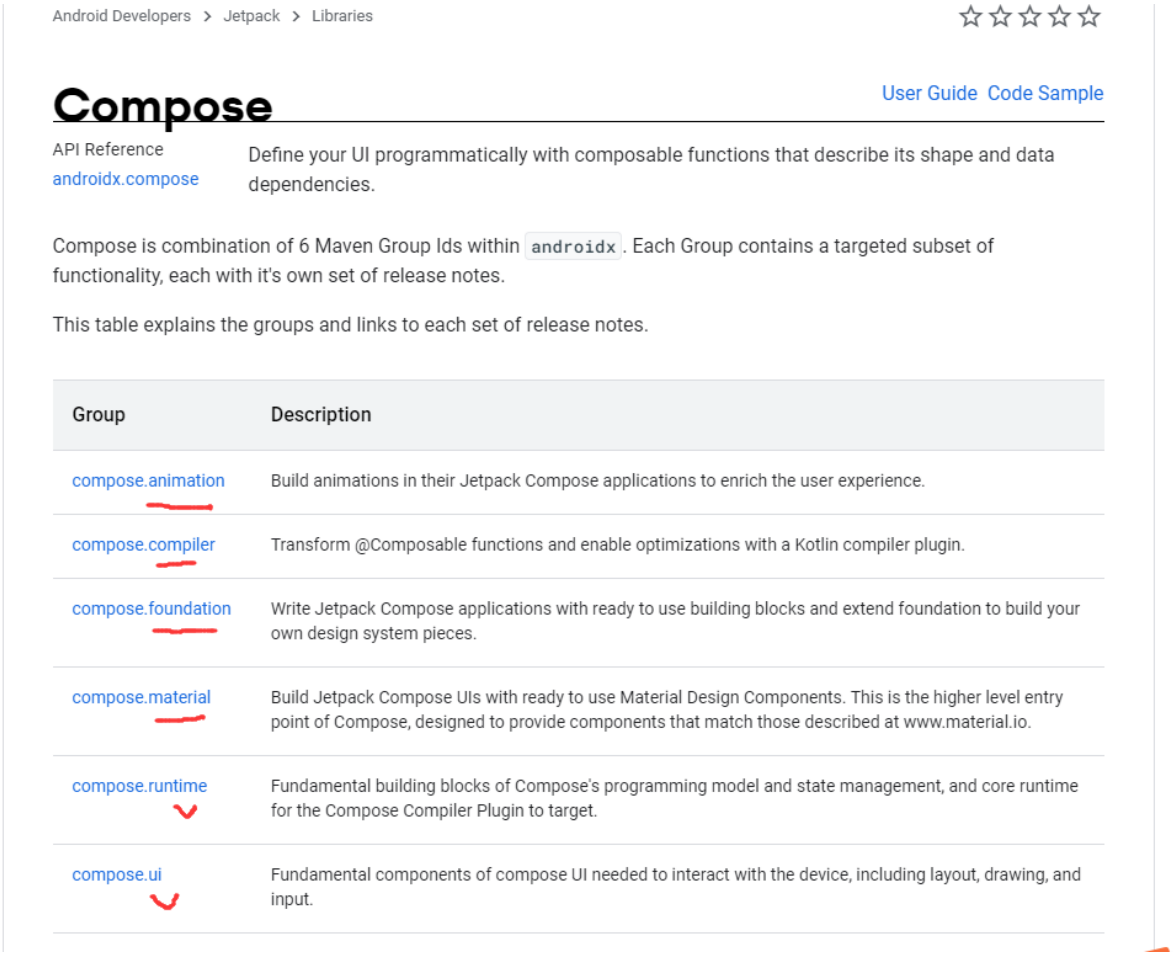
开启Compose

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：

```
implementation "androidx.compose.ui:ui:$compose_version"
//Material Design
implementation "androidx.compose.material:material:$compose_version"
// Tooling support (Previews, etc.)
implementation "androidx.compose.ui:ui-tooling-preview:$compose_version"
implementation 'androidx.activity:activity-compose:1.3.0-alpha06'
// Foundation (Border, Background, Box, Image, Scroll, shapes, animations, etc.)
implementation("androidx.compose.foundation:foundation:1.0.1")
```

Compose组件依赖



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.

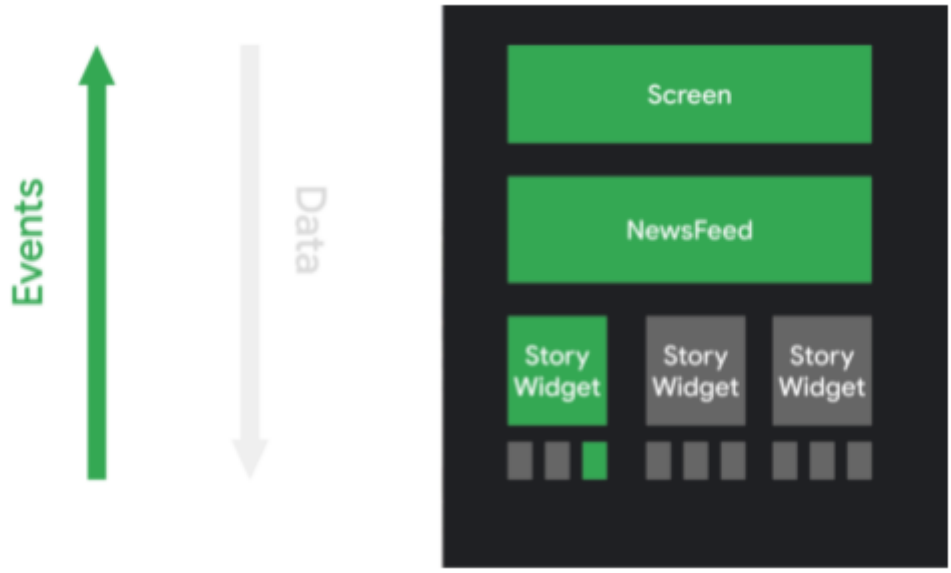
需要注意的地方，Runtime和Compile2个组件，尤其是Complier完成Compose的编译和中间结果的输出。Ui和material完成了Compose组件最核心的界面的展示。

Compose设计思想的总结

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

```
/**
 * Display a list of names the user can click with a header
 */
@Composable
fun NamePicker(
    header: String,
    names: List<String>,
    onNameClicked: (String) -> Unit
) {
    Column {
        // this will recompose when [header] changes, but not when [names]
changes
        Text(header, style = MaterialTheme.typography.h5)
        Divider()

        // LazyColumnFor is the Compose version of a RecyclerView.
        // The lambda passed is similar to a RecyclerView.ViewHolder.
        LazyColumnFor(names) { name ->
            // When an item's [name] updates, the adapter for that item
            // will recompose. This will not recompose when [header] changes
            NamePickerItem(name, onNameClicked)
        }
    }
}

/**
```

```
 * Display a single name the user can click.
 */
@Composable
private fun NamePickerItem(name: String, onClicked: (String) -> Unit) {
    Text(name, Modifier.clickable(onClick = { onClicked(name) }))
}
```

```
@Composable
fun Foo() {
    var text by remember { mutableStateOf( value: "") }
    Log.d( tag: "TAG",  msg: "Foo")
    Button(onClick = {
        text = "$text $text"
    }.also { Log.d( tag: "TAG",  msg: "Button") }) {  this: RowScope
        Wrapper {
            Log.d( tag: "TAG",  msg: "Button content lambda")
            Text(text).also { Log.d( tag: "TAG",  msg: "Text") }
        }
    }
}
//仔细注意这里的inline的函数 只是copy到调用点 这些函数没有自己的重组范围 如果没有inline 关键字
@Composable  inline fun Wrapper(content: @Composable () -> Unit) {
    println("Wrapper recomposing")
    content()
}
```

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

```
@Composable
    fun HelloContent() {
        Column(modifier = Modifier.padding(16.dp)) {
            Text(
                text = "Hello",
                modifier = Modifier.padding(bottom = 8.dp),
                style = MaterialTheme.typography.h5,
            )
            OutlinedTextField(
                value = "",
                onValueChange = {},
                label = { Text(text = "Name") },
            )
        }
    }
```

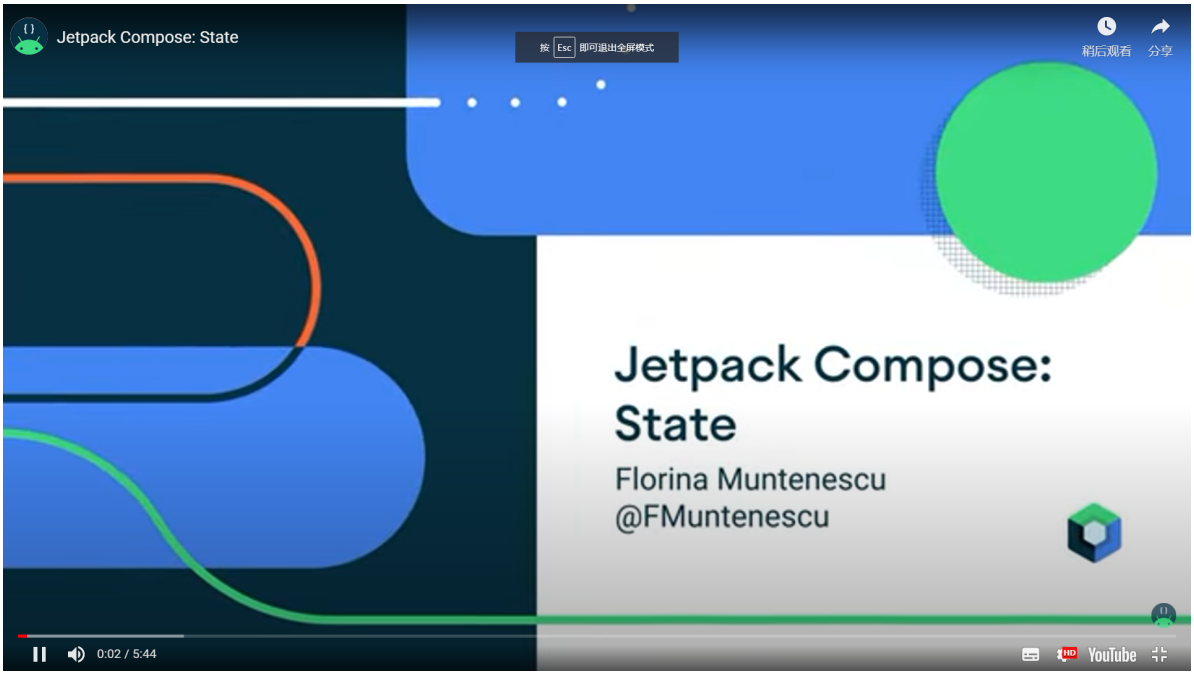
```
        }
    }
}

@SuppressLint( ...value: "UnrememberedMutableState")
@Composable
fun HelloContent() {
    //注意这里的函数需要提取到class里面  HelloContent() 依赖了class的变量 ？？这底层？
//    var text by remember { mutableStateOf("") }
    Column(modifier = Modifier.padding(16.dp)) { this: ColumnScope
        Text(
            text = "Hello",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.h5,
        )
        OutlinedTextField(
            value = inputText.value,
            onValueChange = {inputText.value = it},
            label = { Text(text = "Name") },
        )
    }
}
}
```

请大家自己敲一下代码，仔细体会下里面的意思，重点掌握remember的意思。Jetpack Compose的响应式编程是什么意思?

- 响应式编程就是数据发生了变化，对应的界面就重新绘制，和DataBinding有点类似（不完全一样）
- 传统的界面写法是命令式编程，都是我们主动告诉界面view说，你改变吧，界面view才改变。但是Compose颠覆了，数据驱动UI。

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

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

使用:

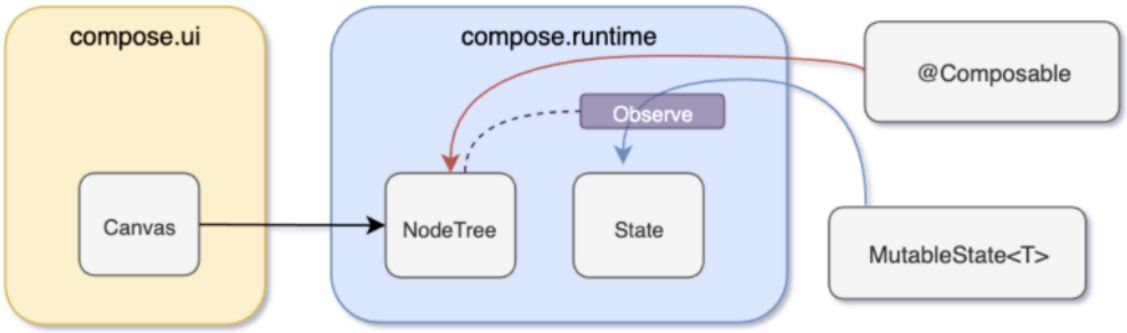
```
val mutableState = remember { mutableStateOf(default) }
var value by remember { mutableStateOf(default) }
val (value, setValue) = remember { mutableStateOf(default) }
```

Jetpack Compose Runtime 与 NodeTree 管理

参考资料:

https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cdd

`compose.runtime` 提供 NodeTree管理、State管理等，声明式UI的基础运行时。此部分与平台无关，在此基础上各平台只需实现UI的渲染就是一套完整的声明式UI框架。而 `compose.compiler` 通过编译期的优化，帮助开发者书写更简单的代码调用 runtime 的能力。



Compose核心实践

> 基础的实践参见视频，这里我将列举一些实战开发中常见的业务来展示。

BottomNavigation

自定义布局

自定义布局可以通过两种方式去处理，一种是使用布局修饰符Modifier.layout，一种是使用Layout去创建自定义布局。我们先来讲下Modifier.layout的方式。Modifier.layout修饰符仅更改调用的可组合项。如需测量和布置多个可组合项，请改用 Layout。

```
/***
 * 绘制圆在想要的位置,Modifier.layout
 */
fun Modifier.customCornerPosLayout(pos:CornerPosition) = layout { measurable,
constraints ->
    //measure
    val placeable = measurable.measure(constraints)
    Log.e("placeable.height", placeable.height.toString())
    Log.e("placeable.height", placeable.height.toString())
```

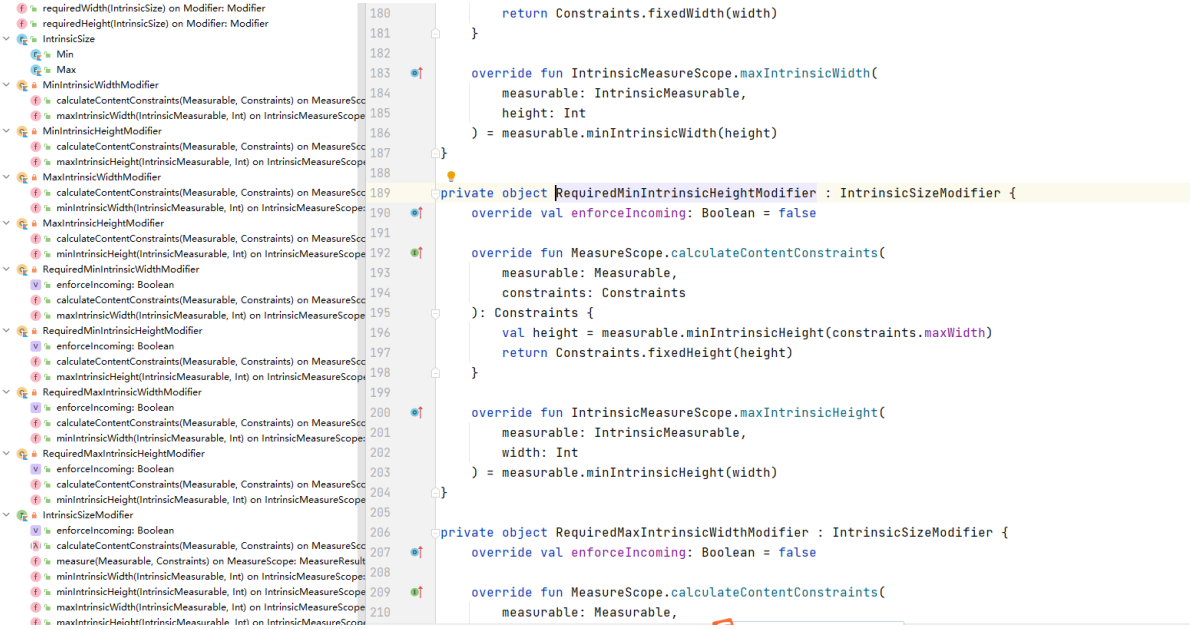
```
    layout(constraints.maxWidth,constraints.maxHeight){
    //计算 pos的 位置
        when(pos){
            CornerPosition.TopLeft->{
                placeable.placeRelative(0, 0)
            }
            CornerPosition.TopRight->{
                placeable.placeRelative(constraints.maxWidth-placeable.width, 0)
            }
            CornerPosition.BottomLeft->{
                placeable.placeRelative(0, constraints.maxHeight-
placeable.height)
            }
            CornerPosition.BottomRight->{
                placeable.placeRelative(constraints.maxWidth-placeable.width,
constraints.maxHeight-placeable.height)
            }
        }
    }
}
```

```
/***
  自定义布局 Layout() 注意和上面的写法区别
***/
@Composable
fun CustomMyColumn(
    layoutDirection: androidx.compose.ui.unit.LayoutDirection, modifier:
Modifier, content:@Composable() ()->Unit
){
    Layout(modifier = modifier,content = content){
            measurables, constraints ->
        var totalHeight = 0
        var maxWidth = 0
        val placeables = measurables.map {
            val placeable = it.measure(constraints)
            totalHeight+=placeable.height
            if(placeable.width>maxWidth){
                maxWidth = placeable.width
            }
            placeable
        }
        layout(maxWidth,totalHeight){
            if(layoutDirection == androidx.compose.ui.unit.LayoutDirection.Ltr){
                var y = 0
                placeables.forEach {
                    it.place(0,y)
                    y+=it.height
                }
            }else{
                var y = totalHeight
                placeables.forEach {
                    y-=it.height
                    it.place(0,y)
                }
            }
        }
    }
```

```
}
```

固有特性测量

以前在Android的View系统中，存在多次测量。所以在View系统中我们为了优化性能，要求减少布局层次的嵌套。而在Compose中，子项只能测量一次，测量两次就会引发运行时异常，所以在Compose中各种嵌套子项是不会影响性能的。Compose中为啥能做到不需要测量多次呢，就是因为有了这个Intrinsic Measurement(固有特性测量)。Intrinsic Measurement 是允许父项对子项测量之前，先让子项测量下自己的最大最小的尺寸。



协程相关

LaunchedEffect就是能让你在Composable中使用协程。

```
@Composable
@NonRestartableComposable
@OptIn(InternalComposeApi::class)
fun LaunchedEffect(
    key1: Any?,
    block: suspend CoroutineScope.() -> Unit
) {
    val applyContext = currentComposer.applyCoroutineContext
    remember(key1) { LaunchedEffectImpl(applyContext, block) }
}

internal class LaunchedEffectImpl(
    parentCoroutineContext: CoroutineContext,
    private val task: suspend CoroutineScope.() -> Unit
) : RememberObserver {
    private val scope = CoroutineScope(parentCoroutineContext)
    private var job: Job? = null

    override fun onRemembered() {
        job?.cancel("Old job was still running!")
        job = scope.launch(block = task)
    }

    override fun onForgotten() {
        job?.cancel()
```

```
            job = null
        }

        override fun onAbandoned() {
            job?.cancel()
            job = null
        }
    }
```

本质上就是将key进行remember之后，然后每次改变就会调用LaunchedEffectImpl的onRemembered方法，启动协程的逻辑（block）。