# 目录

# 第一章、使用 Compose 实现底部按钮和首页 banner 以及数据列表

compose 作为 Android 现在主推的 UI 框架，各种文章铺天盖地的席卷而来，作为一名 Android 开发人员也是很有必要的学习一下了，这里就使用 wanandroid 的开放 api 来编写一个 compose 版本的玩安卓客户端，全当是学习了，各位大佬轻喷~
先来看一下首页的效果图：


从图片中可以看到首页的内容主要分为三部分，头部标题栏，banner，数据列表，底部导航栏；今天就实现这几个功能。

# 1. Column、Row、ConstraintLayout 布局先知

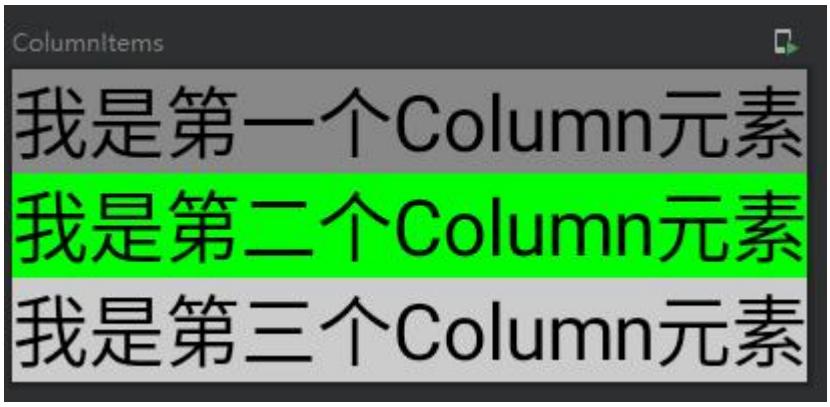
在 Compose 布局中主要常用的就是这三个布局，分别代表纵向排列布局，横向排列布局，以及约束布局;先大概了解一下用法,以及布局包裹内部元素的排列方便在项目中更好的使用。

### 1.1.1 Column 纵向排列布局

Column 主要是将布局包裹内的元素由上至下垂直排列显示，类似于 Recyclerview 的 item，简单来看一段代码：

```
@Preview
@Composable
fun ColumnItems(){
    Column {
        Text(text = "我是第一个Column元素",Modifier.background(Color.Gray))
        Text(text = "我是第二个Column元素",Modifier.background(Color.Green))
        Text(text = "我是第三个Column元素",Modifier.background(Color.LightGray))
    }
}
```

可以看到在一个 Column 里面包裹了三个 Text，那么来看一下效果：




可以看到所有元素是由上至下进行排列的。

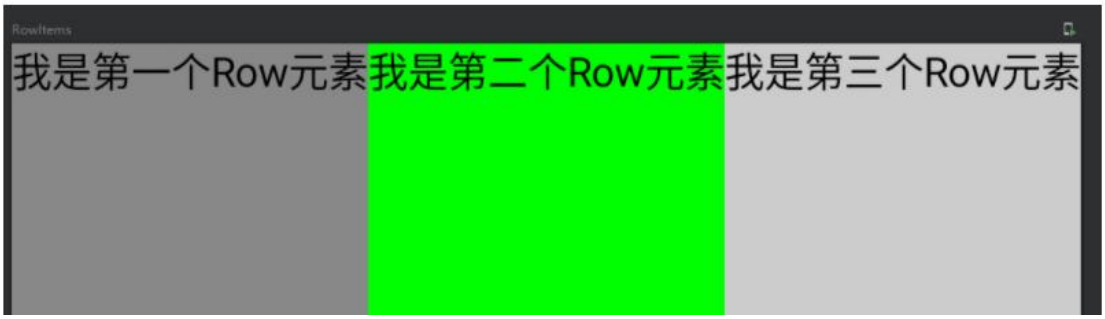
### 1.1.2 Row 横向排列布局

简而言之就是将布局里面的元素一个一个的由左到右横向排列。

再来看一段简短的代码

```
@Preview
@Composable
fun RowItems(){
    Row {
        Text(text = "我是第一个Row元素",Modifier.background(Color.Gray).height(100.dp))
        Text(text = "我是第二个Row元素",Modifier.background(Color.Green).height(100.dp))
        Text(text = "我是第三个Row元素",Modifier.background(Color.LightGray).height(100.dp))
    }
}
```

在 Row 里面同样包裹了三个 Text 文本，再来看一下效果：




可以看到 Row 里面的元素是由左到右横向进行排列的。

### 1.1.3 ConstraintLayout 约束布局

在 compose 里面同样可以使用约束布局，主要主用于一些 Column 或者 Row 或者 Box 布局无法直接实现的布局，在实现更大的布局以及有许多复杂对齐要求以及布局嵌套过深的场景下，ConstraintLayout 用起来更加顺手，在使用 ConstraintLayout 之前需要先导入相关依赖包：

```
implementation "androidx.constraintlayout:constraintlayout-compose:1.0.0-rc01"
```

这里额外提一句，在你创建项目的时候所有 compose 的相关依赖包都要和你项目当前的 compose 版本一致，或者都更新到最新版，如果 compose 的版本大于你现在导入的其他依赖库的版本，那么就会报错。

在使用 ConstraintLayout 需要注意以下几点：

1、声明元素 通过 createRefs() 或 createRef() 方法初始化声明的，并且每个子元素都会关联一个 ConstraintLayout 中的 Composable 组件；
2、关联组件 Modifier.constrainAs(text)通过 constrainAs 关联组件；
3、约束关系可以使用 linkTo 或其他约束方法实现；
4、parent 是一个默认存在的引用，代表 ConstraintLayout 父布局本身，也是用于子元素的约束关联。

来看一段代码：

```
@Preview
@Composable
fun ConstraintLayoutDemo(){
    ConstraintLayout {
        //声明元素
        val (text,text2,text3) = createRefs()

        Text(text = "我是第一个元素",Modifier.height(50.dp).constrainAs(text){
            //将第一个元素固定到父布局的右边
            end.linkTo(parent.end)
        })
        Text(text = "老二",modifier = Modifier.background(Color.Green).constrainAs(text2){
            //将第二个元素定位到第一个元素的底部
            top.linkTo(text.bottom)
            //，然后于第一个元素居中
            centerTo(text)
        })
        Text(text = "老三",modifier = Modifier.constrainAs(text3){
            //将第三个元素定位到第二个元素的底部
            top.linkTo(text2.bottom)
            //将第三个元素定位在第二个元素的右边
            start.linkTo(text2.end)
        })
    }
}
```

看一下效果：


约束布局只要习惯 linkTo 的使用就能很好的使用该布局。

## 1.2 Modifier 的简单使用

Modifier 在 compose 里面可以设置元素的宽高，大小，背景色，边框，边距等属性；这里

只介绍一些简单的用法。

先看一段代码：

modifier = Modifier

// .fillMaxSize()//横向 纵向 都铺满,设置了 fillMaxSize 就不需要设置 fillMaxHeight 和 fillMaxWidth 了

// .fillMaxHeight()//fillMaxHeight 纵向铺满

.fillMaxWidth()//fillMaxWidth()横向铺满 match

.padding(8.dp)//外边距 vertical = 8.dp 上下有 8dp 的边距; horizontal = 8.dp 水平有 8dp 的边距

.padding(8.dp)//内边距 padding(8.dp)=.padding(8.dp,8.dp,8.dp,8.dp)左上右下都有 8dp 的边距

// .width(100.dp)//宽 100dp

// .height(100.dp)//高 100dp

.size(100.dp)//宽高 100dp

// .widthIn(min: Dp = Dp.Unspecified, max: Dp = Dp.Unspecified)//设置自身的最小和最大宽度（当子级元素超过自身时，子级元素超出部分依旧可见）；

.background(Color.Green)//背景颜色

.border(1.dp, Color.Gray,shape = RoundedCornerShape(20.dp))//边框

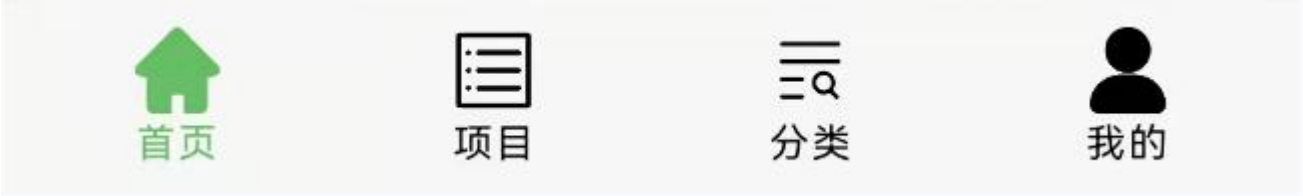
1.fillMaxSize 设置布局纵向横向都铺满。
2.fillMaxHeight 设置布局铺满纵向。
3.fillMaxWidth 设置布局铺满横向，这三个属性再使用了 fillMaxSize 就没必要在设置下面两个了。
4.padding 设置边距，方向由左上右下设置，添加了 vertical 就是设置垂直的上下边距，horizontal 设置了水平的左右边距。这里注意写了两个 padding，第一个是外边距，第二个是内边距，外边距最好是放在 Modifier 的第一个元素。
5.width 设置元素的宽。

6.height 设置元素的高。
7.size 设置元素大小，只有一个值时宽高都是一个值，.size(100.dp，200.dp)两个值前者是宽，后者是高。
8.widthIn 设置自身的最小和最大宽度（当子级元素超过自身时，子级元素超出部分依旧可见）。
9.background 设置元素的背景颜色。
10.border 设置边框，参数值：边框大小，边框颜色，shape。

更多 Modifier 的设置可以查看源码或者官方文档。

## 1.3 底部导航栏的实现

从图中可以可以出，底部导航栏主要包含四个 tab，分别是首页、项目、分类以及我的，而每个 tab 又分别包含一张图片和一个文字。




具体实现步骤：

1、编写每个 tab 的样式，这里要使用到 Column 进行布局，Column 列的意思，就是 Column 里面的元素会一个顺着一个往下排的意思，所以我们需要在里面放一个图片 Icon 和一个文本 Text。

```
Column(
    modifier.padding(vertical = 8.dp),//垂直（上下边距）8dp
    horizontalAlignment = Alignment.CenterHorizontally) {//对齐方式水平居中
    Icon(painter = painterResource(id = iconId),//图片资源
         contentDescription = tabName,//描述
         //图片大小                    //颜色
         modifier = Modifier.size(24.dp),tint = tint)
         //      文本          字体大小        字体颜色
    Text(text = tabName,fontSize = 11.sp,color = tint)
}
```

因为是四个按钮，并且有着选中和未选中的状态，所以我们需要封装成一个方法进行使用：

```
/**
 * 参数解析
 * @DrawableRes iconId: Int
 *
 * iconId  参数名称
 * Int     参数类型
 * @DrawableRes 只能填入符合当前属性的值
 * */
@Composable
private fun TabItem(@DrawableRes iconId: Int, //tab 图标资源
                    tabName: String,//tab 名称
                    tint: Color,//tab 颜色(选中或者未选中状态)
                    modifier: Modifier = Modifier
){
    Column(
        modifier.padding(vertical = 8.dp),
        horizontalAlignment = Alignment.CenterHorizontally) {
        Icon(painter = painterResource(id = iconId),
            contentDescription = tabName,
            modifier = Modifier.size(24.dp),tint = tint)
        Text(text = tabName,fontSize = 11.sp,color = tint)
    }
}
```

2、使用 Row 放置四个 TabItem，Row 水平排列的意思。

```
@Composable
fun BottomBar(modifier: Modifier = Modifier, content: @Composable RowScope.() -> Unit) {
    Row(
        modifier
            .fillMaxWidth()
            .background(ComposeUIDemoTheme.colors.bottomBar)
            .padding(4.dp, 0.dp)
            .navigationBarsPadding(),
        content = content
    )
}
```

```
@Composable
fun BottomTabBar(selectedPosition: Int, currentChanged: (Int) -> Unit){
    //使用 Row 将四个 TabItem 包裹起来，让它们水平排列
    BottomBar() {
        TabItem(
            iconId = if (selectedPosition == 0) R.drawable.home_selected else R.drawable.home_unselected,
            tabName = "首页",
            tint = if (selectedPosition == 0) ComposeUIDemoTheme.colors.iconCurrent else ComposeUIDemoTheme.colors.icon,
            Modifier
                .clickable {
                    currentChanged(0)
                }
                .weight(1f))
        TabItem(
            iconId = if (selectedPosition == 1) R.drawable.project_selected else R.drawable.project_unselected,
            tabName = "项目",
            tint = if (selectedPosition == 1) ComposeUIDemoTheme.colors.iconCurrent else
```

```
ComposeUIDemoTheme.colors.icon,
                Modifier
                    .clickable {
                        currentChanged(1)
                    }
                    .weight(1f))
            TabItem(
                iconId = if (selectedPosition == 2) R.drawable.classic_selected else R.drawable.classic_unselected,
                tabName = "分类",
                tint = if (selectedPosition == 2) ComposeUIDemoTheme.colors.iconCurrent else ComposeUIDemoTheme.colors.icon,
                Modifier
                    .clickable {
                        currentChanged(2)
                    }
                    .weight(1f))
            TabItem(iconId = if (selectedPosition == 3) R.drawable.mine_selected else R.drawable.mine_unselected,
                tabName = "我的",
                tint = if (selectedPosition == 3) ComposeUIDemoTheme.colors.iconCurrent else ComposeUIDemoTheme.colors.icon,
                Modifier
                    .clickable {
                        currentChanged(3)
                    }
                    .weight(1f))
        }
}
```

TabItem 填充解析：

1. iconId tab 图标资源，当选中的下标等于当前 tab 的下标时显示选中的资源，否则显示非选中资源。
2. tabName tab 文本。
3. tint tab 颜色，同样分为选中和未选中。
4. Modifier 使用 Modifier 设置点击事件，以及权重。
5. currentChanged(0) tabitem 的点击事件，返回当前 item 的下标。

```
TabItem(
    iconId = if (selectedPosition == 0) R.drawable.home_selected elseR.drawable.home_unselected,
    tabName = "首页",
    tint = if (selectedPosition == 0) ComposeUIDemoTheme.colors.iconCurrent else ComposeUIDemoTheme.colors.icon,
    Modifier
        .clickable {
            currentChanged(0)
        }
        .weight(1f))
```

3、分别创建 HomePage、ProjectPage、ClassicPage 和 MinePage 四个页面，页面编写一些简单的代码铺满页面即可。

```
@Composable
fun ClassicPage(viewModel: BottomTabBarViewModel = viewModel()){
    Column(Modifier.fillMaxWidth()) {
        DemoTopBar(title = "分类")
        Box(
            Modifier
                .background(ComposeUIDemoTheme.colors.background)
                //使用Modifier将页面铺满
                .fillMaxSize()
        ) {
            Text(text = "分类")
        }
    }
}
```

4、使用 HorizontalPager 进行页面滑动，并且与 tabitem 的点击事件进行绑定，达到页面滑动切换以及点击 tabitem 进行切换的效果。

HorizontalPager 主要参数解析：

1.count 总页面数。

2.state 当前选中的页面状态。

使用 HorizontalPager 需要导入以下资源：

```
implementation "com.google.accompanist:accompanist-pager:$accompanist_pager"//0.20.2
```

具体实现步骤如下：

先通过 remember 记录住当前选中的下标，这个主要作用与 tabItem 的切换

```
//记录页面状态
val indexState = remember { mutableStateOf(0) }
```

然后通过 rememberPagerState 记录 HorizontalPager 的 currentPager 也就是当前页面下标

```
val pagerState = rememberPagerState()
```

使用 HorizontalPager 填充页面

```
HorizontalPager(count = 4,
   state = pagerState,
   modifier = Modifier.fillMaxSize().weight(1f))
    { page: Int ->
        when(page){
            0 ->{
               HomePage()
            }
            1 ->{
               ProjectPage()
            }
            2 ->{
               ClassicPage()
            }
            3 ->{
                MinePage()
            }
    }
}
```

使用 LaunchedEffect 进行页面切换

```
//页面切换
LaunchedEffect(key1 = indexState.value, block = {
      pagerState.scrollToPage(indexState.value)
})
```

最后绑定底部导航栏并绑定点击事件

```
//滑动绑定底部菜单栏
/**
selectedPosition = pagerState.currentPage
将当前的currentPager赋值给tabitem的selectPosition对底部导航栏进行绑定

indexState.value = it
将底部导航栏的点击回调下标赋值给indexState对pager进行绑定
*/
BottomTabBar(selectedPosition = pagerState.currentPage){
        indexState.value = it
}
```

到这里就能实现一个底部导航栏以及四个页面的切换了。

# 2.首页内容的实现

## 2.1.1 Banner 的实现

因为获取 Banner 数据要进行网络请求，至于网络封装就不贴代码了，这里直接从 ViewModel 开始展示，具体的网络代码可以移步到项目进行观看。

## 2.1.2 首页 ViewModel

主要用于 Banner 和首页文章列表的网络请求：

```
class HomeViewModel : ViewModel() {
    private var _bannerList = MutableLiveData(listOf<BannerEntity>())
    val bannerList:MutableLiveData<List<BannerEntity>>    = _bannerList

    fun getBannerList(){
        NetWork.service.getHomeBanner().enqueue(object                                    :
Callback<BaseResult<List<BannerEntity>>>{
            override  fun  onResponse(call:  Call<BaseResult<List<BannerEntity>>>,response:
Response<BaseResult<List<BannerEntity>>>) {
                response.body()?.let {
                    _bannerList.value = it.data
                }
            }

            override fun onFailure(call: Call<BaseResult<List<BannerEntity>>>, t: Throwable) {
            }
        })
    }

    private var _articleData = MutableLiveData<ArticleEntityPage>()
    val articleData:MutableLiveData<ArticleEntityPage> = _articleData
```

```
    fun getArticleData(){
        NetWork.service.getArticleList().enqueue(object : Callback<BaseResult<ArticleEntityPage>>{
            override fun onResponse(call: Call<BaseResult<ArticleEntityPage>>,response: Response<BaseResult<ArticleEntityPage>>) {
                response.body()?.let {
                    articleData.value = it.data
                }
            }

            override fun onFailure(call: Call<BaseResult<ArticleEntityPage>>, t: Throwable) {
            }
        })
    }
}
```

在调用 HomePage 的时候将 HomeViewModel 传入进去，不推荐直接在 compose 里面直接调用，会重复调用：

```
val bVM = HomeViewModel()
HomePage(bVM = bVM)
```

HomePage 的创建：

```
fun HomePage(viewModel: BottomTabBarViewModel = viewModel(), bVM:HomeViewModel){
}
```

数据调用进行请求，首先要创建变量通过 observeAsState 进行数据接收刷新

```
val bannerList by bVM.bannerList.observeAsState()
```

Compose 的网络请求要放到 LaunchedEffect 去执行，才不会重复请求数据

```
val requestState = remember { mutableStateOf("") }
LaunchedEffect(key1 = requestState.value, block = {
    bVM.getBannerList()
})
```

绘制 Banner 的 View，这里同样使用到 HorizontalPager，并且还使用了 coil 进行网络加载，需要导入相关依赖包

```
implementation 'io.coil-kt:coil-compose:1.3.0'
```

BannerView 的代码，实现大致和 tabitem 差不多，只是添加了一个轮播，就不做过多的极细，直接贴代码了

```
@ExperimentalCoilApi
@ExperimentalPagerApi
@Composable
fun BannerView(bannerList: List<BannerEntity>,timeMillis:Long){
    Box(
        Modifier
            .fillMaxWidth()
```

```
                .height(160.dp)) {

            val pagerState = rememberPagerState()
            var executeChangePage by remember { mutableStateOf(false) }
            var currentPageIndex = 0

            HorizontalPager(count = bannerList.size,
                state = pagerState,
                modifier = Modifier
                    .pointerInput(pagerState.currentPage) {
                        awaitPointerEventScope {
                            while (true) {
                                val event = awaitPointerEvent(PointerEventPass.Initial)
                                val dragEvent = event.changes.firstOrNull()
                                when {
                                    dragEvent!!.positionChangeConsumed() -> {
                                        return@awaitPointerEventScope
                                    }

                                    dragEvent.changedToDownIgnoreConsumed() -> {
                                        //记录下当前的页面索引值
                                        currentPageIndex = pagerState.currentPage
                                    }

                                    dragEvent.changedToUpIgnoreConsumed() -> {
                                        if (pagerState.targetPage == null) return@awaitPointerEventScope
                                        if (currentPageIndex == pagerState.currentPage && pagerState.pageCount > 1) {
                                            executeChangePage = !executeChangePage
                                        }
                                    }
                                }
                            }
                        }
                    }
                    .clickable {
                        Log.e(
                            "bannerTAG",
                            "点击的 banner item:${pagerState.currentPage}
```

```
itemUrl:${bannerList[pagerState.currentPage].imagePath}"
                            )
                    }
                    .fillMaxSize()) { page ->
                Image(
                    painter                                                    =
rememberImagePainter(bannerList[page].imagePath),
                    modifier = Modifier.fillMaxSize(),
                    contentScale = ContentScale.Crop,
                    contentDescription = null
                )
            }

            //自动轮播
            LaunchedEffect(key1 = pagerState.currentPage, block = {
                Log.e("LaunchedEffect","${pagerState.currentPage}")
                if (pagerState.currentPage >=0 && pagerState.currentPage <
bannerList.size -1){
                    delay(timeMillis = timeMillis)
                    pagerState.animateScrollToPage(pagerState.currentPage
+1)
                }else{
                    delay(timeMillis = timeMillis)
                    pagerState.animateScrollToPage(0)
                }
            })

    }
}
```

最后就行调用

```
InitBanner(bannerList= bannerList!!,2000L)
```

### 2.1.3 实现文章列表

数据请求就不做过多赘述了，和 banner 的数据请求一样，这里主要解析一下 Compose 的约束布局 ConstraintLayout。

Compose 约束布局需要导入以下相关依赖

```
implementation "androidx.constraintlayout:constraintlayout-compose:1.0.0-rc01"
```

1.val img = createRef() 创建一个依赖点，就像 xml 约束布局里面的 id 意义昂
2.Modifier 使用 Modifier 的 constrainAs 进行约束调整

```
ConstraintLayout(
        Modifier
            .fillMaxWidth()
            .padding(horizontal = 8.dp)) {
             val img = createRef()
             Text(text = "作者:${entity.audit}")

             Image(painter = painterResource(id = R.drawable.icon_un_select),
                   contentDescription = "收藏",
                   Modifier
                       .width(30.dp)
                       .height(30.dp)
                       .constrainAs(img) {
                           end.linkTo(parent.end)
                        },
                    alignment = Alignment.CenterEnd)
}
```

Item 代码如下

```
@Composable
private fun ArticleListItem(entity: ArticleEntity,modifier: Modifier = Modifier){
    Card(
        shape = RoundedCornerShape(10.dp),
        backgroundColor = ComposeUIDemoTheme.colors.listItem,
        elevation = 2.dp,modifier =
        Modifier.padding(0.dp,10.dp,0.dp,0.dp)
    ) {
        Row(
            Modifier
                .fillMaxWidth()
                .clickable {
                    Log.e("articleTAG", "文章点击")
                }) {
            Column(Modifier.fillMaxWidth()) {
                Text(text = "${entity.title}",
                    Modifier.padding(8.dp,8.dp,8.dp,8.dp),
                    fontSize = 16.sp,
                    color = ComposeUIDemoTheme.colors.textPrimary,
                    maxLines = 1,
                    overflow = TextOverflow.Ellipsis)

                ConstraintLayout(
                    Modifier
                        .fillMaxWidth()
                        .padding(horizontal = 8.dp)) {
                    val img = createRef()
                    Text(text = "作者:${entity.audit}")

                    Image(painter = painterResource(id = R.drawable.icon_un_select),
                        contentDescription = "收藏",
                        Modifier
                            .width(30.dp)
                            .height(30.dp)
                            .constrainAs(img) {
                                end.linkTo(parent.end)
                            },
                        alignment = Alignment.CenterEnd)
                }
```

```
                ConstraintLayout(
                    Modifier
                        .fillMaxWidth()
                        .padding(8.dp)) {
                    val parentView = createRef()

                    Text(text = "${entity.superChapterName}",
                        modifier = Modifier
                            .padding(8.dp)
                            .border(
                                1.dp, color = Color(R.color.b_666),
                                RoundedCornerShape(8.dp)
                            )
                            .padding(horizontal = 8.dp, vertical = 2.dp),
                        color = Color.Gray,
                        fontSize = 12.sp
                    )
                    Text(text = "${entity.niceShareDate}",modifier = Modifier.constrainAs(parentView){
                        end.linkTo(parent.end)
                        centerVerticallyTo(parent)
                    },color = Color.Gray,fontSize = 12.sp,textAlign = TextAlign.Center)
                }
            }
        }
    }
}
```

## 2.1 SwipeRefresh 下拉刷新

操作列表的时候下拉刷新和上拉加载更多的操作肯定是少不了的，这里先说一下 Compose SwipeRefresh 下拉刷新组件的使用。

首先导入依赖包：

```
implementation "com.google.accompanist:accompanist-swiperefresh:$accompanist_pager"//0.20.2
```

在 ViewModel 里面保存一个记录刷新状态的元素，在刷新请求数据过程中这个值要变成 true 进行刷新动画的加载，刷新完成之后要变成 false 关闭加载动画。

```
val _isRefreshing: MutableLiveData<Boolean> = MutableLiveData(false)
val isRefreshing by viewModel._isRefreshing.observeAsState(false)
SwipeRefresh(state = rememberSwipeRefreshState(isRefreshing = isRefreshing),
    onRefresh = {
          //将刷新状态的值改为true  显示加载动画
          viewModel._isRefreshing.value = true
          //将请求数据的值做改变  让它能重新请求数据
          requestState.value = "refresh"+System.currentTimeMillis()
    }) {
        //填充数据
        Box(
           Modifier
                .background(ComposeUIDemoTheme.colors.background)
                .fillMaxSize()
          ) {
              Column(Modifier.fillMaxWidth()) {
                    if (bannerList !== null){
                        InitBanner(bannerList= bannerList!!,2000L)
                    }
                    if (articleEntityPage !== null){
                        InitHomeArticleList(articleData = articleEntityPage!!)
                    }
            }
     }
}

LaunchedEffect(key1 = requestState.value, block = {
    bVM.getBannerList()
    bVM.getArticleData({
        viewModel._isRefreshing.value = false//将刷新状态的值改成false 关闭加载状态
    })
})
```

首页的内容一共就这么多了，到这里就能实现一个 app 的地步导航栏以及首页了。

## 2.2 LaunchedEffect 简介

LaunchedEffect 存在的意义是允许我们在被 Composable 标注的方法中使用协程。

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

主要参数是 key1，block；key1 是触发条件，当 key1 发生改变的时候就会执行 block 里面的方法，注意是发生改变的时候。

我们一开始在请求数据的时候创建了一个 requestState 来记录请求状态，如果这个状态不发生改变，那么就只会触发一次 LaunchedEffect 对数据进行请求。

```
//记录请求状态
val requestState = remember { mutableStateOf("") }
LaunchedEffect(key1 = requestState.value, block = {
   bVM.getBannerList()
   bVM.getArticleData({
        viewModel._isRefreshing.value = false//将刷新状态的值改成false 关闭加载状态
   })
})
```

而当我们在刷新的时候在 onRefres 里面改变了这个参数的值，那么它就会触发

LaunchedEffect 重新进行数据请求。

```
onRefresh = {
        //将刷新状态的值改为true  显示加载动画
        viewModel._isRefreshing.value = true
        //将请求数据的值做改变  让它能重新请求数据
        requestState.value = "refresh"+System.currentTimeMillis()
}
```

## 2.3 HorizontalPager 简介

HorizontalPager 类似于 AndroidView 的 ViewPager，用于页面承载和滑动切换，主要参数是 count 页面数据，以及 state 当前页面状态。

```
// Display 10 items
HorizontalPager(count = 10) { page ->
    // Our page content
    Text(
        text = "Page: $page",
        modifier = Modifier.fillMaxWidth()
    )
}
```

通过 scrollToPage 可以进行页面切换，但是只能在协程里面被执行，所以同样要使用 LaunchedEffect 进行页面切换。

```
//页面切换
LaunchedEffect(key1 = indexState.value, block = {
     pagerState.scrollToPage(indexState.value)
})
```

更多用法以及 demo 移步以下链接。

```
https://google.github.io/accompanist/pager/
```

# 第二章、导航规整并实现登录页个人中心页

在前面开发时只是注重了页面绘制，已经 compose 各种组件的使用，没有规整导航，所以页面跳转的操作很难实现；今天先规整一下页面导航，在页面跳转操作完成之后在绘制登录页以及个人中心页。

## 1. 导航规整

在前面绘制页面的时候说到，compose 打开页面的时候会在当前页面直击打开，所以就需要把要打开的页面都放在主页中进行打开，那么就要区分页面是首页，还是其他页面。

首先定义一个页面枚举，main 代表首页，其他则是其他页面：

```
/**
 * 页面类
 * */
enum class RouteKey(val route:String){
    Main("main"),
    Login("login"),
    WebView("webview")
}
```

使用 navhost 进行导航，主页所有页面都用 navigation 进行包裹：

```
@ExperimentalMaterialApi
@ExperimentalCoilApi
@ExperimentalPagerApi
@Composable
fun RouteNavigation(navHostController: NavHostController,
          onFinish: () -> Unit
){
    val context = LocalContext.current

    NavHost(navController = navHostController, startDestination = RouteKey.Main.route){
        //主页面
        navigation(
            route = RouteKey.Main.route,
            startDestination = Nav.BottomNavScreen.HomeScreen.route
        ){

            composable(Nav.BottomNavScreen.HomeScreen.route){
                HomePage(navHostController = navHostController)
            }
            ....//其他要展示在主页面的paer
        }
        //要打开的新页面
        //登录页
        composable(RouteKey.Login.route) {
            LoginPage(navHostController = navHostController)
        }
    }
}
```

首页导航页面把四个页面都封装起来：

```
object Nav {
    sealed class BottomNavScreen(val route: String, @StringRes val resourceId: Int, @DrawableRes val id: Int) {
        object HomeScreen: BottomNavScreen("home", R.string.nav_home, R.drawable.home_unselected)
        object ProjectScreen: BottomNavScreen("project",R.string.nav_project,R.drawable.project_unselected)
        object ClassicScreen: BottomNavScreen("classic",R.string.nav_classic,R.drawable.classic_unselected)
        object MineScreen: BottomNavScreen("mine", R.string.nav_mine, R.drawable.mine_unselected)
    }
    //主页点击两次返回桌面
    var onMainBackPressed = false
    val bottomNavRoute = mutableStateOf<BottomNavScreen>(BottomNavScreen.HomeScreen)
}
```

将不同页面的展示封装到一个 page，所有页面都在这个 page 打开，并加载到 MainActivity 里面去，但是要区分是主页，还是其他页面。

先判断是否是主页：

```
fun isMainScreen(route:String):Boolean = when(route){
    Nav.BottomNavScreen.HomeScreen.route,
        Nav.BottomNavScreen.ProjectScreen.route,
        Nav.BottomNavScreen.ClassicScreen.route,
        Nav.BottomNavScreen.MineScreen.route -> true

    else -> false
}
```

然后根据得到的结果加载页面：

```
@ExperimentalPagerApi
@ExperimentalMaterialApi
@Composable
fun MainPage(
    navHostController: NavHostController = rememberNavController(),
    onFinish:() -> Unit
){
    //返回back堆栈的顶部条目
    val navBackStackEntry by navHostController.currentBackStackEntryAsState()
    //返回当前route
    val currentRoute = navBackStackEntry?.destination?.route ?: Nav.BottomNavScreen.HomeScreen.route
    //加载主页内容
    if (isMainScreen(currentRoute)){
        Scaffold(
            contentColor = MaterialTheme.colors.background,
            //标题栏
            topBar = {
                Column {
                    Spacer(
                        modifier = Modifier
                            .background(MaterialTheme.colors.primary)
                            .statusBarsHeight()
                            .fillMaxWidth()
                    )
                }
            },
            //底部导航栏
            bottomBar = {
                Column {
                    BottomNavBar(Nav.bottomNavRoute.value, navHostController)
                    Spacer(
                        modifier = Modifier
                            .background(MaterialTheme.colors.primary)
                            .navigationBarsHeight()
                            .fillMaxWidth()
                    )
                }
            },
            //内容
            content = { paddingValues: PaddingValues ->
                //内容嵌套在Scaffold中
                RouteNavigation(navHostController, paddingValues, onFinish)

                OnBackClick(navHostController)
            })
    }else{
        //加载独立页面
        RouteNavigation(navHostController, onFinish = onFinish)
    }
}
```

到这里就完成了导航的规整，页面打开也没有问题，接下来就是个人中心页面以及登录页面的绘制和实现了。

# 2. 个人中心的实现

目前个人中心比较简单，就展示了一个头像，昵称，用户 id 以及用户积分，更多的东西等

到实现收藏等操作之后在添加，简单看一下效果图


布局元素比较简单，这里就不贴布局文件了。

## 2.1 MineViewmodel 获取数据

登录成功之后保存 cookie，通过 cookie 调用用户信息接口，获取用户信息。

```
class MineViewModel : ViewModel() {
    //默认头像
    val                          defaultHead                          =
"https://jusha-info.oss-cn-shenzhen.aliyuncs.com/obt/mall/upload/imag
e/store/2021/08/06/1628250153533.png"

    private val _userInfo = MutableLiveData<UserConfigModule>()
    val userInfo = _userInfo
    fun getUserInfo(){
        Log.e("intoTAG","get user info")
        NetWork.service.getUserInfo().enqueue(object                    :
Callback<BaseResult<UserConfigModule>>{
            override fun onResponse(
                call:    Call<BaseResult<UserConfigModule>>,response:
Response<BaseResult<UserConfigModule>>) {
                Log.e("intoTAG","response")
                response.body()?.let {
                    _userInfo.value = it.data
                }
            }

            override                fun                onFailure(call:
Call<BaseResult<UserConfigModule>>, t: Throwable) {
                Log.e("intoTAG","onFailure${t.message}")
            }
        })
```

```
    }
}
```

## 2.2 MinePage

获取信息并展示。

```
@Composable
fun MinePage(navHostController: NavHostController){
    val mineViewModel:MineViewModel = viewModel()
    val userInfo by mineViewModel.userInfo.observeAsState()
    mineViewModel.getUserInfo()
    Column(
        Modifier
            .fillMaxSize()
            .verticalScroll(rememberScrollState())) {
        com.yangchoi.composeuidemo.ui.bar.TopAppBar(title = "我的")
        Box(
            Modifier
                .background(Color.White)
                .fillMaxSize()
        ) {
            Column(Modifier.fillMaxSize()) {
                if (userInfo !== null){
                    //头像昵称
```

```
ConstraintLayout {

    val (headImg,userName,userId)    = createRefs()


    Image(painter = rememberImagePainter(mineViewModel.defaultHead),
        contentDescription = "用户头像",
        modifier = Modifier
            .size(80.dp)
            .padding(16.dp, 20.dp, 0.dp, 0.dp)
            .clip(shape = RoundedCornerShape(50))
            .constrainAs(headImg) {})


    Text(text = "${userInfo!!.userInfo.nickname}",fontSize = 14.sp,color = Color.Black,modifier = Modifier
        .padding(10.dp, 20.dp, 0.dp, 0.dp)
        .constrainAs(userName) {
            start.linkTo(headImg.end)
            top.linkTo(headImg.top)
        })


    Text(text = "${userInfo!!.userInfo.id}",fontSize = 12.sp,color = Color.Gray,modifier = Modifier
        .padding(10.dp, 20.dp, 0.dp, 0.dp)
```

```
                .constrainAs(userId) {
                    start.linkTo(headImg.end)
                    bottom.linkTo(headImg.bottom)
                })
            }

            ConstraintLayout(modifier = Modifier
                .fillMaxWidth()
                .padding(vertical = 40.dp)
                .height(50.dp)) {
                val (icons,title,integral,btmLine) = createRefs()

                Row(Modifier
                    .constrainAs(icons) {}
                    .fillMaxHeight()
                    .padding(16.dp, 0.dp, 0.dp, 0.dp),
                    verticalAlignment = Alignment.CenterVertically) {
                    Image(painter = painterResource(id
=R.drawable.icon_integral),
                        contentDescription = "积分", modifier =
Modifier
                            .height(20.dp)
                            .width(20.dp))
```

```
                }


                Row(modifier = Modifier
                    .constrainAs(title) {
                        start.linkTo(icons.end)
                    }
                    .fillMaxHeight()
                    .padding(horizontal = 10.dp),
                    verticalAlignment = Alignment.CenterVertically) {
                    Text(text = "积分",fontSize = 12.sp,color =
Color.Black,textAlign = TextAlign.Center)
                }


                Row(modifier = Modifier
                    .fillMaxHeight()
                    .constrainAs(integral) {
                        end.linkTo(parent.end)
                    }
                    .padding(horizontal = 16.dp),
                    verticalAlignment = Alignment.CenterVertically) {
                    Text(text =
"${userInfo!!.coinInfo.coinCount}",fontSize = 12.sp,color = Color.Gray,textAlign =
TextAlign.Center,)
```

```
                }


                Divider(
                    modifier = Modifier
                        .padding(0.dp, 0.dp, 16.dp, 0.dp,)
                        .constrainAs(btmLine) {
                            bottom.linkTo(parent.bottom)
                        },
                    color = Color(229,224,227),
                    thickness = 1.dp,
                    startIndent = 16.dp)
            }
        }else{
            Row(modifier = Modifier
                .padding(horizontal = 16.dp, vertical = 200.dp)
                .fillMaxWidth()
                .height(50.dp)
                .border(
                    1.dp,
                    color = Color(114, 160, 240),
                    shape = RoundedCornerShape(20.dp)
                ),verticalAlignment = Alignment.CenterVertically) {
```

```
                Text(text = "登　录",
                    fontSize = 16.sp,
                    modifier = Modifier
                        .fillMaxWidth()
                        .clickable {
navHostController.navigate("${RouteKey.Login.route}")
                        },
                    color = Color(114, 160, 240),
                    textAlign = TextAlign.Center)

                }
            }


        }
    }
}
```

## 2.3 请求头添加 cookie

登录成功之后会返回一个 cookie 在请求头里面，只需要将 cookie 拦截并保存下来，就可以通过 cookie 去获取用户信息。

```kotlin
//创建OKhttp
private val client: OkHttpClient.Builder = OkHttpClient.Builder()
            .addInterceptor(LogInterceptor())
            .addInterceptor {
                val request = it.request()
                val response = it.proceed(request)
                val requestUrl = request.url.toString()
                val domain = request.url.host
                //cookie可能有多个，都保存下来
                if ((requestUrl.contains(SAVE_USER_LOGIN_KEY) || requestUrl.contains(SAVE_USER_REGISTER_KEY))) {
                    val cookies = response.headers(SET_COOKIE_KEY)
                    val cookie = encodeCookie(cookies)
                    saveCookie(requestUrl, domain, cookie)
                }
                response
            }
            //请求时设置cookie
            .addInterceptor {
                val request = it.request()
                val builder = request.newBuilder()
                val domain = request.url.host
                //获取domain内的cookie
                if (domain.isNotEmpty()) {
                    val sqDomain: String = DataStoreUtil.readStringData(domain, "")
                    val cookie: String = if (sqDomain.isNotEmpty()) sqDomain else ""
                    if (cookie.isNotEmpty()) {
                        builder.addHeader(COOKIE_NAME, cookie)
                    }
                }
                it.proceed(builder.build())
            }
            .connectTimeout(10, TimeUnit.SECONDS)
            .readTimeout(20, TimeUnit.SECONDS)
            .retryOnConnectionFailure(false)
```

在网络请求的位置添加以上两个拦截器就行了。

# 3. 登录页面的实现

首先来看效果图。


简单的绘制了一个登录页面，UI 就不要纠结了，丑是真的丑~

可以看到在输入框左边有一个图标，然后是提示内容，以及密码框右边的显示和隐藏密码的图标；选中的时候颜色发生改变，并且在左上角显示提示用户输入的内容。

## 3.1 OutlinedTextField 属性解析

在实现以上效果前，先要了解 OutlinedTextField 的属性，才能加以运用 ；先看一下属性列表。

```
@Composable
fun OutlinedTextField(
    value: String,
    onValueChange: (String) -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    readOnly: Boolean = false,
    textStyle: TextStyle = LocalTextStyle.current,
    label: @Composable (() -> Unit)? = null,
    placeholder: @Composable (() -> Unit)? = null,
    leadingIcon: @Composable (() -> Unit)? = null,
    trailingIcon: @Composable (() -> Unit)? = null,
    isError: Boolean = false,
    visualTransformation: VisualTransformation = VisualTransformation.None,
    keyboardOptions: KeyboardOptions = KeyboardOptions.Default,
    keyboardActions: KeyboardActions = KeyboardActions.Default,
    singleLine: Boolean = false,
    maxLines: Int = Int.MAX_VALUE,
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },
    shape: Shape = MaterialTheme.shapes.small,
    colors: TextFieldColors = TextFieldDefaults.outlinedTextFieldColors()
)
```

- value: String 输入框显示的文本
- onValueChange: (String) -> Unit 值发生改变之后触发的回调
- modifier: Modifier = Modifier 修饰
- enabled: Boolean = true 可用
- readOnly: Boolean = false 是否只读
- textStyle: TextStyle = LocalTextStyle.current
- label: @Composable (() -> Unit)? = null 输入框获取焦点时左上角提示的内容
- placeholder: @Composable (() -> Unit)? = null 输入框提示的内容
- leadingIcon: @Composable (() -> Unit)? = null 输入框左侧的图标
- trailingIcon: @Composable (() -> Unit)? = null 输入框右侧的图标
- isError: Boolean = false 是否处于错误状态
- visualTransformation: VisualTransformation = VisualTransformation.None, 转换输入值的视觉表示
- keyboardOptions: KeyboardOptions = KeyboardOptions.Default 输入框输入类型
- singleLine: Boolean = false, 是否单行显示
- maxLines: Int = Int.MAX_VALUE 最大行数
- colors: TextFieldColors = TextFieldDefaults.outlinedTextFieldColors() 颜色集合，设置获取焦点，失去焦点以及光标等颜色

大致就是这些属性了，知道使用之后就可以封装输入框了。

## 3.2 封装输入框

定义状态枚举 PwdShowState，通过状态设置密码是否可见。

通过 value:String 设置输入框显示的值。

通过 placeholder:String 设置输入框提示的值。

通过 color:TextFieldColors 设置对应状态下的颜色，获得焦点、失去焦点、以及光标时候的颜色。

通过 leadingIcon:ImageVector 设置左侧图标。

通过 trailingIcon:ImageVector 设置右侧图标,通过 trailingtintIcon:Color 设置图标颜色。

通过 keyboardOptions: KeyboardOptions 设置输入框输入类型。
通过 visualTransformation: VisualTransformation = VisualTransformation.None 改变设置密码是否可见。

通过 onValueChange:(String) -> Unit 获取输入框发生改变时值的回调。

```
//输入框
enum class PwdShowState{
    Show,Hide
}
@Composable
fun MyTextField(value:String,
                label:String,
                placeholder:String,
                color:TextFieldColors,
                leadingIcon:ImageVector,
                trailingIcon:ImageVector,
                trailingtintIcon:Color,
                modifier: Modifier,
                modifierTrailing: Modifier,
                keyboardOptions: KeyboardOptions,
                visualTransformation: VisualTransformation = VisualTransformation.None,
                onValueChange:(String) -> Unit){

    val showState = remember {
        mutableStateOf(PwdShowState.Hide)
    }

    val icon =  if (showState.value === PwdShowState.Hide){
        painterResource(id = R.drawable.pwd_look)
    }else{
        painterResource(id = R.drawable.pwd_hide)
    }

    OutlinedTextField(value = value,
        colors = color,
        label = {
            Text(text = label)
        },
        placeholder = {
            Text(text = placeholder)
        },
        modifier = modifier,
        keyboardOptions = keyboardOptions,
        leadingIcon = {
            Icon(leadingIcon,"左边图标",modifierTrailing,trailingtintIcon)
        },
        trailingIcon = {
            if (label.equals("密码")){
                IconButton(onClick = {
                    if (showState.value === PwdShowState.Hide){
                        showState.value = PwdShowState.Show
                    }else{
                        showState.value = PwdShowState.Hide
                    }
                }) {
                    if (showState.value === PwdShowState.Hide){
                        Icon(icon, contentDescription = "点击密码可见",modifier = Modifier.size(30.dp))
                    }else{
                        Icon(icon, contentDescription = "点击密码隐藏",modifier = Modifier.size(30.dp))
                    }
                }
            }
        },
        visualTransformation = if (label.equals("密码")){
            if (showState.value === PwdShowState.Hide){ PasswordVisualTransformation()} else visualTransformation
        }else{
            visualTransformation
        },
        singleLine = true,
        onValueChange = onValueChange)
}
```

## 3.3 输入框的使用

根据不同的使用场景，设置不同的参数。

账号：

```
val userName = remember {
        mutableStateOf("")
}
val colors = TextFieldDefaults.outlinedTextFieldColors(
          focusedBorderColor = Color(68,84,246),
          unfocusedBorderColor = Color.Gray,
          cursorColor = Color(68,84,246)
)
MyTextField(
          value = userName.value,
          label = "账号",
          placeholder = "请输入账号",
          color = colors,
          leadingIcon = Icons.Default.Phone,
          trailingIcon = Icons.Default.Phone,
          trailingtintIcon = Color(68,84,246),
          modifier = Modifier
                .padding(12.dp, 0.dp, 12.dp, 0.dp)
                .fillMaxWidth(),
          modifierTrailing = Modifier,
          keyboardOptions = KeyboardOptions(
               keyboardType = KeyboardType.Text
          ),
          onValueChange = {
                userName.value = it
         }
)
```

密码：

```
val password = remember {
       mutableStateOf("")
}
val colors = TextFieldDefaults.outlinedTextFieldColors(
          focusedBorderColor = Color(68,84,246),
          unfocusedBorderColor = Color.Gray,
          cursorColor = Color(68,84,246)
)
MyTextField(
      value = password.value,
      label = "密码",
      placeholder = "请输入密码",
      color = colors,
      leadingIcon = Icons.Default.Lock,
      trailingIcon = Icons.Default.Lock,
      trailingtintIcon = Color(68,84,246),
      modifier = Modifier
              .padding(12.dp, 0.dp, 12.dp, 0.dp)
              .fillMaxWidth(),
      modifierTrailing = Modifier,
      keyboardOptions = KeyboardOptions(
              keyboardType = KeyboardType.Password
      ),
      onValueChange = {
        password.value = it
      }
)
```

输入框实现完成~

## 3.4 登录按钮实现

在点击登录按钮的时候，登录接口请求过程中加载一个简单的动画，在登录成功或者失败之后结束动画。

## 3.5 创建按钮状态枚举

Normal 正常情况下的状态
Pressed 按下时的状态
remember 记录状态的值

```
//按钮添加动画
enum class ButtonState{
    Normal,Pressed
}
//记录状态值
val buttonState = remember {
        mutableStateOf(ButtonState.Normal)
}
```

## 3.6 定义 transition

定义一个 transition，以及后面通过该元素设置颜色、大小等参数。

```
val transition = updateTransition(targetState = buttonState, label = "ButtonTransition")
```

## 3.7 设置按钮颜色、大小以及 shape

```
    val buttonBackgroundColor: Color by transition.animateColor(
        transitionSpec = { tween(duration)}
    ) { buttonState ->
        when(buttonState.value){
            ButtonState.Normal -> Color(68,84,246)
            ButtonState.Pressed -> Color(68,84,246)
        }
    }

    val buttonWidth: Dp by transition.animateDp(transitionSpec = {
        tween(duration)}
    ) {buttonState ->
        when(buttonState.value){
            ButtonState.Normal -> 300.dp
            ButtonState.Pressed -> 60.dp
        }
    }

    val buttonShape: Dp by transition.animateDp(transitionSpec = {
        tween(duration)}
    ) {buttonState ->
        when(buttonState.value){
            ButtonState.Normal -> 4.dp
            ButtonState.Pressed -> 100.dp
        }
    }
```

## 3.8 使用 Button 并配置样式

属性列表

```
@OptIn(ExperimentalMaterialApi::class)
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },
    elevation: ButtonElevation? = ButtonDefaults.elevation(),
    shape: Shape = MaterialTheme.shapes.small,
    border: BorderStroke? = null,
    colors: ButtonColors = ButtonDefaults.buttonColors(),
    contentPadding: PaddingValues = ButtonDefaults.ContentPadding,
    content: @Composable RowScope.() -> Unit
)
```

• onClick: () -> Unit 点击事件回调

• enabled: Boolean = true 是否可用，是否可以点击，这里可以加上判断，当用户名和密码都不为空的时候可以使用 enabled = !userName.isNullOrBlank() && !password.isNullOrBlank()

• colors: ButtonColors = ButtonDefaults.buttonColors() 设置背景颜色以及点击时候的背景颜色等

• content: @Composable RowScope.() -> Unit compose 函数，实现逻辑

```
Button(modifier = Modifier
        .width(buttonWidth)
        .height(50.dp),shape = RoundedCornerShape(buttonShape),
        onClick = {
        //todo
        },colors = ButtonDefaults.buttonColors(
            backgroundColor = buttonBackgroundColor,
            disabledBackgroundColor = Color(68,84,246).copy(0.5f)
        ),enabled = !userName.isNullOrBlank() && !password.isNullOrBlank()) {
            if (buttonState.value == ButtonState.Normal){
                Text(text = "登录")
            }else{
                CircularProgressIndicator(
                    color = Color.White,
                    strokeWidth = 2.dp,
                    modifier = Modifier.size(24.dp)
                )
            }
        }
```

点击逻辑，将按钮状态设置成 Pressed

```
buttonState.value = ButtonState.Pressed
```

并请求登录接口

```
loginViewModel.toLogin(userName,password,{
    //回调  状态重置
      buttonState.value = ButtonState.Normal
      navHostController.navigateUp()
})
```

通过 CircularProgressIndicator 实现动画

```
if (buttonState.value == ButtonState.Normal){
            Text(text = "登录")
}else{
            CircularProgressIndicator(
                color = Color.White,
                strokeWidth = 2.dp,
                modifier = Modifier.size(24.dp)
            )
}
```

## 3.9 按钮全部代码

按钮封装的代码：

```
//按钮添加动画
enum class ButtonState{
    Normal,Pressed
}

@Composable
fun MyButton(userName:String,password:String,loginViewModel:LoginViewModel,navHostController: NavHostController){
    val buttonState = remember {
        mutableStateOf(ButtonState.Normal)
    }

    val transition = updateTransition(targetState = buttonState, label = "ButtonTransition")

    val duration = 600

    val buttonBackgroundColor: Color by transition.animateColor(
        transitionSpec = { tween(duration)}
    ) { buttonState ->
        when(buttonState.value){
            ButtonState.Normal -> Color(68,84,246)
            ButtonState.Pressed -> Color(68,84,246)
        }
    }

    val buttonWidth: Dp by transition.animateDp(transitionSpec = {
        tween(duration)}
    ) {buttonState ->
        when(buttonState.value){
            ButtonState.Normal -> 300.dp
            ButtonState.Pressed -> 60.dp
        }
    }

    val buttonShape: Dp by transition.animateDp(transitionSpec = {
        tween(duration)}
    ) {buttonState ->
        when(buttonState.value){
            ButtonState.Normal -> 4.dp
            ButtonState.Pressed -> 100.dp
        }
```

```
    }

    Button(modifier = Modifier
        .width(buttonWidth)
        .height(50.dp),shape = RoundedCornerShape(buttonShape),onClick = {
        buttonState.value = ButtonState.Pressed
        loginViewModel.toLogin(userName,password,{
            buttonState.value = ButtonState.Normal
            navHostController.navigateUp()
        })
    },colors = ButtonDefaults.buttonColors(
        backgroundColor = buttonBackgroundColor,
        disabledBackgroundColor = Color(68,84,246).copy(0.5f)
    ),enabled = !userName.isNullOrBlank() && !password.isNullOrBlank()) {
        if (buttonState.value == ButtonState.Normal){
            Text(text = "登录")
        }else{
            CircularProgressIndicator(
                color = Color.White,
                strokeWidth = 2.dp,
                modifier = Modifier.size(24.dp)
            )
        }
    }
}
```

调用：

```
MyButton(userName.value,password.value,loginViewModel,navHostController)
```

到这里呢整个登录页面所有的元素都构建好了，剩下的就是 viewmodel 实现登录请求以及结果回调了。

### 3.10 LoginViewModel

```
class LoginViewModel : ViewModel() {

    private val _loginInfo = MutableLiveData<Any>()
    val loginInfo = _loginInfo

    fun toLogin(username:String,password:String,callback:()->Unit){
        NetWork.service.login(username,password).enqueue(object : Callback<BaseResult<Any>>{
            override fun onResponse(call: Call<BaseResult<Any>>,response: Response<BaseResult<Any>>) {
                response.body()?.let {
                    _loginInfo.value = it
                }
                callback.invoke()
            }

            override fun onFailure(call: Call<BaseResult<Any>>, t: Throwable) {
                callback.invoke()
            }
        })
    }

}
```

登录页面的绘制以及实现就完成了，因为不能放置视频，登录按钮点击时的动画也没有弄成 GIF，这里就不放效果图了，代码很简单，效果跑起来就能看到。

## 第三章、实现分类页面

之前实现了底部导航栏以及滑动切换，这里根据官方推荐的底部导航栏的使用方式重新实现了底部导航栏，并实现分类页面，通过 API 获取导航数据，实现左边菜单栏，右边内容显示的效果，效果图如下：


# 1. Scaffold 简单使用

使用 Scaffold 可以实现 Compose 的基槽位布局，比如 topBar 顶部菜单栏，bottomBar 底部导航栏，floatingActionButtonPosition 悬浮按钮等等；这里就不做过多的介绍了，详情可以查阅 Scaffold 的属性进行设置，这里主要看 bottomBar 的实现。

先看一下 bottomBar 在 Scaffold 的表现形式：

```
bottomBar: @Composable () -> Unit = {},
```

从参数类型可以看出来，我们需要在里面放置一个被@Composable 标记的函数，那么就先创建一个函数，并使用@Composable 注解：

```
@Composable
fun BottomTab(){
    //实现逻辑
}
```

然后使用 Scaffold，参数实现一个 bottomBar 就可以了：

```
Scaffold(
  bottomBar = {
      BottomTab()
    }) {
        //逻辑实现
    }
  }
```

接下来就是使用 BottomNavigation 和 NavHost 实现底部导航的操作了。

# 2. BottomNavigation 和 NavHost 实现底部导航

官方推荐使用 BottomNavigation 实现导航栏，先来看一下 BottomNavigation 的属性，根据自己的需求设置即可：

```
@Composable
fun BottomNavigation(
    modifier: Modifier = Modifier,
    backgroundColor: Color = MaterialTheme.colors.primarySurface,
    contentColor: Color = contentColorFor(backgroundColor),
    elevation: Dp = BottomNavigationDefaults.Elevation,
    content: @Composable RowScope.() -> Unit
)
```

在正式使用之前我们还需要设置一些变量，比如底部菜单的文字，选中和未选中的图片资源：

```
private val tabs = arrayOf("首页","项目","分类","我的")
private val defImg = arrayOf(R.drawable.home_unselected,R.drawable.project_unselected,R.drawable.classic_unselected,R.drawable.mine_unselected)
private val selectImg = arrayOf(R.drawable.home_selected,R.drawable.project_selected,R.drawable.classic_selected,R.drawable.mine_selected)
```

然后将 BottomTab 的方法补齐，如下：

```
@Composable
fun BottomTab(navController:NavController,viewModel:BottomTabBarViewModel,labels:Array<String>,selectImages:Array<Int>,defImages:Array<Int>){
        BottomNavigation(backgroundColor = Color.White, elevation = 6.dp,modifier = Modifier.navigationBarsPadding()//要设置这个属性，不然你会发现你的底部导航栏不见了
        ) {
            for (i in labels.indices) {
                BottomNavigationItem(selected = viewModel.bottomBarIndex == i, onClick = {
                        viewModel.bottomBarIndex = i
                        navController.navigate(labels[i])
                    }, icon = {
                        Image(
                            painter = painterResource(id = if (viewModel.bottomBarIndex == i) selectImages[i] else defImages[i]),
                            contentDescription = labels[i],
                            modifier = Modifier.size(25.dp)
                        )
```

```
                    }, label = {
                        Text(text = labels[i], color = if
(viewModel.bottomBarIndex == i) Color(114,160,240) else Color.Gray)
                    })
                }
            }
}
```

参数分析：

1.navController 导航控制器，主要用于设置 NavHost 的路由，代码里面表现为 navController.navigate(labels[i])

2.labels 文字资源集合

3.selectImages 选中图片集合

4.defImages 未选中图片集合

5.modifier = Modifier.navigationBarsPadding 应用与内容底部边缘的导航栏高度相匹配的附加空间，以及与相应开始边缘和结束边缘上的导航栏宽度相匹配的附加空间。简单来说就是不设置该属性你的导航栏会被挤出屏幕外。

## 2.1 BottomNavigationItem

底部导航栏的 item，和 Recyclerview 的 item 差不多，这里通过 for 循环去添加，一共四个 item

```
for (i in labels.indices) {
    BottomNavigationIte(){}
}
```

参数解析:

1.selected: Boolean, 是否选中，代码表现为：selected = viewModel.bottomBarIndex == i

2.onClick: () -> Unit 点击事件，点击后要保存选中的下标，并且通知 NavHost 切换路由，代码表现为：

```
onClick = {
    viewModel.bottomBarIndex = i//保存选中下标
    navController.navigate(labels[i])//切换路由
}
```

3. icon: @Composable () -> Unit 图片资源

4.label: @Composable (() -> Unit)? = null, 文字资源

## 2.2 NavHost 切换路由

使用 NavHost 切换路由，先来看一下 NavHost 的属性：

```
@Composable
public fun NavHost(
    navController: NavHostController,
    startDestination: String,
    modifier: Modifier = Modifier,
    route: String? = null,
    builder: NavGraphBuilder.() -> Unit
)
```

1. navController 导航控制器

2startDestination 设置目的地

3.builder: NavGraphBuilder.() -> Unit 实现逻辑

看代码实现：

```
val navController = rememberNavController()
Scaffold(
        bottomBar = {
                BottomTab(navController = navController,viewModel = viewModel, labels = tabs, selectImages = selectImg, defImages = defImg)
        }) {
                NavHost(navController = navController, startDestination = tabs[viewModel.bottomBarIndex]){
                        //startDestination 的值等于 tabs[0]的值切换到 HomePage
                        composable(tabs[0]) {
                                HomePage(bVM = bVM)
                        }
                        //startDestination 的值等于 tabs[1]的值切换到 ProjectPage
                        composable(tabs[1]) {
                                ProjectPage()
                        }
                        composable(tabs[2]) {
                                ClassicPage(cVM = cVM)
                        }
                        composable(tabs[3]) {
                                MinePage()
                        }
        }
}
```

通过以上代码就可以实现官方推荐的导航使用方法了。

# 3.分类页面的实现

前面说的都是导航栏的使用，属于前面的内容了；进入今天的主题，分类页面的实现；从效果图可以看出分类页面主要分为两部分，左边的菜单栏和右边的内容显示栏，点击左边的菜单，右边显示对应的内容。

## 3.1 获取数据

在实现功能之前肯定要先获取数据，那么创建 ClassicViewModel 进行数据获取：

```
class ClassicViewModel : ViewModel() {
    private var _naviList = MutableLiveData(listOf<DataEntity>())
    val naviList:MutableLiveData<List<DataEntity>> = _naviList

    fun getNaviList(){
        NetWork.service.getNaviJson().enqueue(object : Callback<NaviEntity>{


            override fun onResponse(call: Call<NaviEntity>, response: Response<NaviEntity>) {
                response.body()?.let {
                    _naviList.value = it.data
                }
            }

            override fun onFailure(call: Call<NaviEntity>, t: Throwable) {
            }

        })
    }
    val selectIndex: MutableLiveData<Int> = MutableLiveData(0)
    init {
        getNaviList()
    }
}
```

在 ClassicPage 页面获取到数据，并且获取选中的下标：

```
val naviList by cVM.naviList.observeAsState()
val selectIndex by cVM.selectIndex.observeAsState(0)
```

## 3.2 左边布局的实现

因为左边布局比较简单，就一个列表然后设置选中和未选中的样式，这里就不做过多的赘述了，直接贴代码：

```
@Composable
private fun ClassicLeftList(naviList: List<DataEntity>,selectIndex: Int,clickCallBack:((Int)->Unit)){
    LazyColumn{
        itemsIndexed(naviList){ index: Int, item: DataEntity ->
            Box(modifier = Modifier
                .width(120.dp)
                .background(if (index == selectIndex) Color(150,180,233) else ComposeUIDemoTheme.colors.listItem)
                .height(48.dp)
                .clickable {
                    clickCallBack.invoke(index)
                }) {
                ClassicLeftItem(title = naviList[index].name,index = index, selectIndex = selectIndex)
            }
        }
    }
}
@Composable
private fun ClassicLeftItem(title:String,index:Int,selectIndex:Int){
    Row(verticalAlignment = Alignment.CenterVertically,modifier = Modifier.height(48.dp)) {
        Text(text = title,
            modifier = Modifier.fillMaxWidth(),
            fontSize = 12.sp,
            color = if (index == selectIndex) Color(248,249,249) else ComposeUIDemoTheme.colors.icon,
            textAlign = TextAlign.Center)
    }
}
```

## 3.3 右边布局的实现

从图中可以看到，右边不是列表，而是一个流式布局，但是要内容总有超出屏幕显示区域的时候，所以这里先设置一下右边布局的基本属性：

```
@Composable
private fun ClassicRightList(dataList:List<Article>){
    //verticalScroll(rememberScrollState()设置内容可以上下滑动
    Column(modifier = Modifier
        .padding(16.dp,0.dp,0.dp,0.dp)
        .fillMaxSize()
        .background(color = Color.White)
        .verticalScroll(rememberScrollState())) {
        ClassicRightLayout{
            for (index in dataList.indices) {
                Child(text = dataList[index].title)
            }
        }
    }
}
@Composable
private fun Child(modifier: Modifier = Modifier, text: String) {
    Card(
        modifier = modifier,
        border = BorderStroke(color = Color.Black, width = Dp.Hairline),
        shape = RoundedCornerShape(8.dp)
    ) {
        Row(
            modifier = Modifier.padding(start = 8.dp, top = 4.dp, end = 8.dp, bottom = 4.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            Spacer(Modifier.width(4.dp))
            Text(text = text)
        }
    }
}
```

## 3.4 填充 ClassicPage 内容

通过 Row 来填充页面内容

```
@Composable
fun ClassicPage(cVM:ClassicViewModel){
    val naviList by cVM.naviList.observeAsState()
    val selectIndex by cVM.selectIndex.observeAsState(0)
    Column(Modifier.fillMaxWidth()) {
        DemoTopBar(title = "分类")
        Row(modifier = Modifier.fillMaxSize()) {
            if (naviList != null && naviList?.size !== 0){
                ClassicLeftList(naviList = naviList!!,selectIndex){
                    cVM.selectIndex.value = it
                }
                Box(modifier = Modifier
                    .fillMaxHeight()
                    .width(10.dp)
                    .background(color = Color(234,233,234))) {

                }
                ClassicRightList(dataList = naviList!![selectIndex].articles)
            }
        }
    }
}
```

# 4. Compose 自定义布局实现流式布局

Compose 的自定义 view 和 Android 传统的自定义 view 步骤差不多，一般分为以下几个步骤：

1. 获取父 view 的总宽度
2. 测量每一个子 view 所占用的宽度
3. 根据不同需求摆放子 view 的位置

在 Compose 使用 Layout 来测量和布置子 view，如下：

```
fun ClassicRightLayout(
    modifier: Modifier = Modifier,
    content: @Composable () -> Unit
) {
    Layout(
        modifier = modifier,
        content = content
    ) { measurables, constraints ->

    }
}
```

参数解析：

1. measurables 需要测量的子项列表

2. constraints 父布局的约束条件

## 4.1 遍历所有子项，测量宽高

```
//获取父控件最大宽度
val parentWidth = constraints.maxWidth

//当前行宽(超出屏幕要换行)
var lineWidth = 0
//当前行高
var lineHeight = 0
//总高度(每换行一次记录一次)
var totalHeight = 0
//所有可放置的内容
val placeableList = mutableListOf<MutableList<Placeable>>()
//每行的最高高度
val mLineHeight = mutableListOf<Int>()
//每行放置的内容
var lineViews = mutableListOf<Placeable>()

/**
 * 需要测量的子项 测量子View，获取FlowLayout的宽高
 * 遍历子项测量宽高
 * */
measurables.mapIndexed { i, measurable ->
      // 测量子view
      val placeable = measurable.measure(constraints)
      // 设置子view宽高
      val childWidth = placeable.width
      val childHeight = placeable.height
      //如果当前行宽度超出父Layout则换行
      if (lineWidth + childWidth > parentWidth) {
            mLineHeight.add(lineHeight)//添加行高
            placeableList.add(lineViews)//将当前子布局放到所有的内容集合里面去

            //将当前行的子view清空，然后换行添加新的view
            lineViews = mutableListOf()
            lineViews.add(placeable)
            //记录总高度
            totalHeight += lineHeight
            //重置行高与行宽
            lineWidth = childWidth
            lineHeight = childHeight
            totalHeight += 10.dp.toPx().toInt()
```

```
      } else {
            //记录每行宽度
            lineWidth += childWidth + if (i == 0) 0 else 10.dp.toPx().toInt()
            //记录每行最大高度
            lineHeight = maxOf(lineHeight, childHeight)
            //将当前子view添加到当前行内容里面去
            lineViews.add(placeable)
      }
}
```

## 4.2 定位子项

```
layout(parentWidth, totalHeight) {
        //从左上角开始定位 top 0  Left 0
      var topOffset = 0
      var leftOffset = 0
      //循环定位
      for (i in placeableList.indices) {
           lineViews = placeableList[i]
           lineHeight = mLineHeight[i]
           for (j in lineViews.indices) {
               val child = lineViews[j]
               val childWidth = child.width
               val childHeight = child.height
               // 根据Gravity获取子项y坐标
               val childTop = topOffset + (lineHeight - childHeight) / 2
               child.placeRelative(leftOffset, childTop)
               // 更新子项x坐标
               leftOffset += childWidth + 10.dp.toPx().toInt()
           }
           //重置子项x坐标
           leftOffset = 0
           //子项y坐标更新
           topOffset += lineHeight + 10.dp.toPx().toInt()
      }
}
```

以上代码就是本章的全部内容了~

# 第四章、实现搜索页面

今天来实现一下搜索页面，使用 ROOM 数据库保存搜索的历史记录，根据不同加载状态展示不同布局，并使用官方的 Flow layout 来展示数据等操作。

先来看一下效果图，很丑~将就看一下


## 1. ROOM 数据库

Room 是一个数据持久化库，它是 Architecture Component 的一部分。它让 SQLiteDatabase 的使用变得简单，大大减少了重复的代码，并且把 SQL 查询的检查放在了编译时。

Room 数据库主要由 Dao、Entity、DataBase 三部分组成。

使用 Room 之前要导入依赖：

```
//jetpack room
implementation 'androidx.room:room-runtime:2.2.6'
kapt 'androidx.room:room-compiler:2.2.6'
```

并在 build.gradle 的 plugins 中添加以下代码：

```
plugins {
    ...
    id 'kotlin-kapt'
}
```

## 1.1 Entity

在声明数据库实体类的时候需要使用@Entity 来标注改实体类，主要属性如下：

- tableName 设置表名
- indices 设置索引
- primaryKeys 设置主键
- foreignKeys 设置外键
- inheritSuperIndices 父类索引是否被当前类继承

创建一个 tableName 为 search_table 的 Entity，同时设置主键，注意设置主键的时候的初始值要为 0：

```
@Entity(tableName = "search_table")
data class SearchModule(
    @PrimaryKey(autoGenerate = true)//主键自增
    val id:Int = 0,//主键初始值为0

    @ColumnInfo(name = "time_stamp")//列名
    @SerializedName("time_stamp")
    val time_stamp:String,

    @ColumnInfo(name = "search_content")
    @SerializedName("search_content")
    val search_content:String,

    @ColumnInfo(name = "is_delete")
    @SerializedName("is_delete")
    val is_delete:Boolean = false,

    @ColumnInfo(name = "create_time")
    @SerializedName("create_time")
    val create_time:String
)
```

## 1.2 Dao

数据访问对象，全称 Data Access Objects，是 Room 的主要组件，负责定义访问数据库的方法，Room 在编译时创建每个 DAO 实例。DAO 抽象地以一种干净的方式去访问数据库，它可以是一个接口也可以是一个抽象类。如果它是一个抽象类，它可以有一个构造函数，它将

RoomDatabase 作为其唯一参数。

定义一个 Dao，并实现增加，查询，删除三个方法；同时该类也要被@Dao 进行注解：

```
@Dao
interface Dao {

    @Insert
    fun insert(searchModule: SearchModule):Long

    @Delete
    fun delete(searchList:List<SearchModule>):Int

    @Query("SELECT * FROM SEARCH_TABLE WHERE is_delete ==:isDelete")
    fun queryDataList(isDelete:Boolean):List<SearchModule>
}
```

## 1.3 DataBase

数据库持有者，并作为与应用持久关联数据的底层连接的主要访问点。在运行时，通 databaseBuilder() 或者 inMemoryDatabaseBuilder()获取 Database 实例。

使用 DataBase 要注意以下几个地方：

- 该类必须是 abstract 的
- 该类必须继承 RoomDatabase
- 该类必须包含至少一个@Entity 标注的类
- 该类必须包含一个被@Dao 标注的类

```
//至少包含一个@Entity标注的类
@Database(entities = [SearchModule::class],exportSchema = false,version = 1)
abstract class SearchDataBase : RoomDatabase() {
    //至少包含一个@Dao标注的类
    abstract val dao:Dao
}

@Volatile
private var dbInstance: SearchDataBase? = null

val Context.db: SearchDataBase
    get() {
        if (dbInstance == null){
            synchronized(SearchDataBase::class) {
                if (dbInstance == null) {
                    val ctx = MyApplication.context
                    dbInstance = Room
                        .databaseBuilder(ctx, SearchDataBase::class.java, "search")
                        .build()
                }
            }
        }
        return dbInstance!!
    }
```

## 1.4 viewmodel 定义操作方法

创建好 room 相对应的三个类之后，就可以执行数据增删改查操作了；创建一个 SearchViewModel 来实现数据操作，在 viewmodel 中操作数据。

在 viewmodel 中创建插入方法：

```
fun insetData(context: Context,searchModule: SearchModule):Long{
    val db = context.db
    val insert = db.dao.insert(searchModule)
    return insert
}
```

在 viewmodel 中创建查询方法：

```
fun queryDataList(context: Context,isDelete:Boolean){
    var dataList:List<SearchModule>? = null
    val db = context.db
    dataList = db.dao.queryDataList(isDelete)
    _dataList.postValue(dataList)
}
```

这里给数据源设置值的时候要注意，因为操作数据库要在子线程中操作，所以不能直接使用 _dataList.value 的方法设置值，而是要改为使用 postValue 的方法设置值。

在 viewmodel 中创建删除方法：

```
fun clearDataBase(context: Context,dataList:List<SearchModule>):Int{
    val db = context.db
    val int = db.dao.delete(dataList)
    if (int != 0){
        _dataList.postValue(null)
     }
    return int
}
```

## 1.5 page 实现数据操作

在 page 页面调用 viewmodel 的方法，进行输入的新增查询等操作。

在 page 页面点击搜索的时候往数据库插入数据，在插入数据之前需要先定义插入数据的相关类：

```
val searchModule = SearchModule(
time_stamp = "${System.currentTimeMillis()}",
search_content = searchTv.value,
is_delete = false,
create_time = formatTime())
```

新开一个子线程进行输入插入：

```
thread {
    val insertLong = searchViewModel.insetData(MyApplication.context,searchModule)
}
```

insertLong 返回的是插入成功的条数。
在 page 页面查询数据：

```
val searchViewModel:SearchViewModel = viewModel()
val dataList by searchViewModel.dataList.observeAsState()
thread {
    searchViewModel.queryDataList(MyApplication.context,false)
}
```

# 2. 官方 Flow Layout

前面在绘制分类页面的时候，自定义了一个 FlowLayout，当时只是为了尝试一下 compose 的自定义 view；在搜索这里同样要实现内容的 Flow 排列，这里使用官方的 Flow Layout。要使用官方的 Flow Layout 需要导入以下依赖：

```
implementation "com.google.accompanist:accompanist-flowlayout:$accompanist_pager"
```

## 2.1 Flow Layout 属性

使用 FlowLayout 实现历史记录的展示，如下图：


在使用组件之前还是要先知道组件的属性，才能更好的使用该组件。

```
@Composable
public fun FlowRow(
    modifier: Modifier = Modifier,
    mainAxisSize: SizeMode = SizeMode.Wrap,
    mainAxisAlignment: FlowMainAxisAlignment = FlowMainAxisAlignment.Start,
    mainAxisSpacing: Dp = 0.dp,
    crossAxisAlignment: FlowCrossAxisAlignment = FlowCrossAxisAlignment.Start,
    crossAxisSpacing: Dp = 0.dp,
    lastLineMainAxisAlignment: FlowMainAxisAlignment = mainAxisAlignment,
    content: @Composable () -> Unit
)
```

- modifier 修饰
- mainAxisSize 布局在主轴方向上的大小
- mainAxisAlignment 每行的子对象在主轴方向上的对齐。
- mainAxisSpacing 每行子对象之间的主轴间距。
- crossAxisAlignment 每行的子项在横轴方向上的对齐。
- crossAxisSpacing 布局行之间的横轴间距。
- lastLineMainAxisAlignment 替代最后一行的主轴对齐方式。
- content flowlayout 的内容，可以理解为放置的子元素

同时还要注意 Flow Layout 分为 Flow Row 和 Flow Column 两种，我们这里使用的是 Flow Row。使用示例如下：

```
FlowRow(modifier = Modifier
                .fillMaxWidth()
                .padding(12.dp, 20.dp, 0.dp, 0.dp)) {
                /***/
}
```

### 2.2 FlowRow 添加数据

直接在 FlowRow 的 content 里面循环添加子元素

# 3. 状态布局

简单实现一个数据加载状态布局，分别对应数据加载的加载中，加载失败，加载成功，数据为空等状态时显示不同的布局。

### 3.1 定义状态枚举

创建四个枚举，分别对应不同状态：

```
enum class LoadingState{
    Loading,
    Error,
    Empty,
    Success
}
```

### 3.2 展示布局

根据不同状态显示不同布局，首先要定义四个布局回调：

```
@Composable
fun LoadingStateLayout(modifier: Modifier,
                       loadingState: LoadingState,
                       loading:@Composable () ->Unit,//加载中
                       error:@Composable () ->Unit,//加载出错
                       empty:@Composable () ->Unit,//数据为空
                       success:@Composable () ->Unit){//加载成功
                       /***/

}
```

根据不同状态显示不同布局：

```
Box(modifier = modifier){
        val state = loadingState
        when(state){
            LoadingState.Loading->  loading()
            LoadingState.Error ->   error()
            LoadingState.Empty ->   empty()
            LoadingState.Success -> success()
        }
}
```

### 3.3 记录数据状态

在 viewmodel 中记录请求状态

```
val searchLoadingState:MutableLiveData<LoadingState> = MutableLiveData(LoadingState
```

在 page 页面同步请求状态：

```
//搜索内容加载状态loading
val searchLoadingState by searchViewModel.searchLoadingState.observeAsState()
```

### 3.4 viewmodel 获取和 page 展示数据

在 viewmodel 中创建数据集合

```
//搜索结果
private val _searchList = MutableLiveData<SearchModule2>()
val searchList = _searchList
```

在 viewmodel 中创建请求数据的方法：

```
fun getSearchContent(page:Int,keyWord:String,pageSize:Int){
        NetWork.service.searchContent(page,keyWord,pageSize).enqueue(object : Callback<BaseResult<SearchModule2>>{
            override fun onResponse(
                call: Call<BaseResult<SearchModule2>>,
                response: Response<BaseResult<SearchModule2>>) {
                response.body()?.let {
                    _searchList.value = it.data
                }
            }

            override fun onFailure(call: Call<BaseResult<SearchModule2>>, t: Throwable) {
                searchLoadingState.value = LoadingState.Error
            }

        })
}
```

根据返回的数据来判断是否为空，如果为空则将保存的状态设置为 empty，否则为 success：

```
if (it.data?.datas != null){
       searchLoadingState.value = LoadingState.Success
}else{
        searchLoadingState.value = LoadingState.Empty
}
```

在 page 中获取数据：

```
val searchList by searchViewModel.searchList.observeAsState()
searchViewModel.getSearchContent(0,searchTv.value,3)
```

在 page 中使用状态布局展示数据：

```
Spacer(modifier = Modifier.height(20.dp))
Text(text = "搜索结果",fontSize = 14.sp,color = Color.Black,modifier = Modifier.padding(horizontal = 16.dp))
LoadingStateLayout(
            modifier = Modifier.fillMaxWidth(),
            loadingState = searchLoadingState!!,
            loading = {
                Column(modifier = Modifier
                    .padding(vertical = 20.dp)
                    .fillMaxWidth()
                    .height(100.dp),
                    horizontalAlignment = Alignment.CenterHorizontally,
                    verticalArrangement = Arrangement.Center) {
                    CircularProgressIndicator()
                    Text(text = "请先搜索",modifier = Modifier.padding(vertical = 10.dp))
                }
            },
            error = {
                Column(modifier = Modifier
                    .fillMaxWidth()
                    .height(100.dp),
                    horizontalAlignment = Alignment.CenterHorizontally,
                    verticalArrangement = Arrangement.Center) {
                    Text(text = "暂无数据")
                }
            },
            empty = {
                Column(modifier = Modifier
                    .fillMaxWidth()
                    .height(100.dp),
                    horizontalAlignment = Alignment.CenterHorizontally,
                    verticalArrangement = Arrangement.Center) {
                    Text(text = "暂无数据")
                }
            }) {
            LazyColumn(modifier = Modifier.fillMaxWidth()){
                itemsIndexed(searchList!!.datas){ index: Int, item: SearchList ->
                    Column(modifier = Modifier
                        .fillMaxWidth()
                        .height(45.dp),
                        horizontalAlignment = Alignment.CenterHorizontally,
                        verticalArrangement = Arrangement.Center) {
                        Text(text = item.title,fontSize = 14.sp)
                    }
                    Divider(
                        modifier = Modifier
                            .padding(0.dp, 0.dp, 12.dp, 0.dp,),
                        color = Color(229,224,227),
                        thickness = 1.dp,
                        startIndent = 16.dp)
                }
            }
}
```

# 第五章、项目页面的实现

之前已经实现首页和分类页面，今天来实现项目页面，主要是一个顶部的滑动菜单，和下面滑动的 pager，以及点击 item 跳转 webview 打开链接；先来看一下效果图，因为手机崩掉了，所以用一张小程序的效果图代替，大致一样。


## 1. 获取数据

在 viewmodel 中获取数据，顶部的 tab 菜单栏直接获取，底部数据栏在选中的 tab 发生变化时要重新获取。

```
class ProjectViewModel : ViewModel() {
    private var _treeList = MutableLiveData(listOf<TreeEntity>())
    val treeList = _treeList

    fun getProjectTreeList(){
        NetWork.service.getProjectTreeList().enqueue(object : Callback<BaseResult<List<TreeEntity>>>{
            override fun onResponse(
                call: Call<BaseResult<List<TreeEntity>>>,
                response: Response<BaseResult<List<TreeEntity>>>) {
                response.body()?.let {
                    _treeList.value = it.data
                }
            }

            override fun onFailure(call: Call<BaseResult<List<TreeEntity>>>, t: Throwable) {
            }

        })
    }

    private var _treeChild = MutableLiveData<TreeChildEntity>()
    val treeChild:MutableLiveData<TreeChildEntity> = _treeChild

    fun getTreeChildList(cid:Int){
        NetWork.service.getProjectTreeChildList(cid).enqueue(object : Callback<BaseResult<TreeChildEntity>>{
            override fun onResponse(
                call: Call<BaseResult<TreeChildEntity>>,
                response: Response<BaseResult<TreeChildEntity>>) {
                response.body()?.let {
                    _treeChild.value = it.data
                }
            }

            override fun onFailure(call: Call<BaseResult<TreeChildEntity>>, t: Throwable) {
            }

        })
    }

    val _isToWebview: MutableLiveData<Boolean> = MutableLiveData(false)
    val webviewUrl: MutableLiveData<String> = MutableLiveData("")
    init {
        getProjectTreeList()
    }
}
```

点击 tab 重新获取列表数据

```
val dataList by pVM.treeChild.observeAsState()

LaunchedEffect(key1 = pageIndex, block = {
    pVM.getTreeChildList(cid = cid)
})
```

数据已经请求到了，接下来就是页面的绘制。

## 2. Controllable 实现顶部滑动菜单

使用 TabRow 可以实现一个菜单栏，但是不能滑动，这里官方推出了 ScrollableTabRow 组件来实现可滑动的菜单栏，在菜单栏有很多的情况下可以使用此组件来实现。

### 2.1 ScrollableTabRow 属性解析

先来了解一下 ScrollableTabRow 的属性，方便在项目中使用：

```
@Composable
fun ScrollableTabRow(
    selectedTabIndex: Int,
    modifier: Modifier = Modifier,
    backgroundColor: Color = MaterialTheme.colors.primarySurface,
    contentColor: Color = contentColorFor(backgroundColor),
    edgePadding: Dp = TabRowDefaults.ScrollableTabRowPadding,
    indicator: @Composable (tabPositions: List<TabPosition>) -> Unit = @Composable { tabPositions ->
        TabRowDefaults.Indicator(
            Modifier.tabIndicatorOffset(tabPositions[selectedTabIndex])
        )
    },
    divider: @Composable () -> Unit = @Composable {
        TabRowDefaults.Divider()
    },
    tabs: @Composable () -> Unit
)
```

1.selectedTabIndex 当前选中的 tabitem 下标

2.modifier

3.backgroundColor 背景颜色

4.contentColor 内容颜色

5.edgePadding 左边边距，默认是有 52.dp 的边距，所以左边和右边看起来就会离屏幕两侧很远

6.indicator 设置滑动条的属性，默认是白色

7.divider 底部分割线

8.tabs tab 元素集合

### 2.1.1 将 ScrollableTabRow 和 HorizontalPager 进行绑定

在代码中使用 ScrollableTabRow，在使用之前我们需要获取 HorizontalPager 的页面状态，毕竟菜单栏和页面是要进行绑定的，所以用 HorizontalPager 的页面状态来进行绑定：

```
val pagerState =  rememberPagerState()//记录页面状态
selectedTabIndex = pagerState.currentPage,//将页面和tab进行绑定
pagerState.scrollToPage(index)//点击tab滑动到指定pager
```

### 2.1.2 根据各种属性设置样式

```
ScrollableTabRow(
    selectedTabIndex = pagerState.currentPage,
    modifier = Modifier
        .fillMaxWidth()
        .height(50.dp),
    backgroundColor = Color.White,
    indicator = { positions ->//设置滑动条的属性，默认是白色的
        TabRowDefaults.Indicator(
            color = Color(114,160,240),
            modifier = Modifier
                .tabIndicatorOffset(positions[pagerState.currentPage])
                .width(10.dp)
        )
    },
    divider = {//设置底部的分割线
        Box(Modifier
            .fillMaxWidth()
            .height(2.dp)
            .background(Color.White)) {

        }
    },edgePadding = 0.dp) {
    //tasb
}
```

### 2.1.3 循环添加 tabs 子项元素

```
if (treeList !== null && treeList?.size !== 0){
        treeList!!.forEachIndexed { index, treeEntity ->
            Tab(
                text = {
                    Text(
                            text = treeEntity.name,
                            color = if (index == pagerState.currentPage) Color(114,160,240) else Color.Black)
                },                          //标签名
                selected = pagerState.currentPage == index,      //是否选中
                onClick = {                                      //点击事件
                        CoroutineScope(Dispatchers.Main).launch {
                            pagerState.scrollToPage(index)
                        }
                 },
                 modifier=Modifier.alpha(0.9f),            //透明度
                 enabled=true,                                   //是否启用
                 selectedContentColor= Color(114,160,240),                  //选中的颜色
                 unselectedContentColor= Color.Black,                //未选中的颜色
            )
    }

}
```

## 3. HorizontalPager 实现页面数据列表

HorizontalPager 和 ConstraintLayout 已经在前面介绍过了，这里就不再介绍了，直接贴列表代码了。

### 3.1.1 列表样式

列表样式没什么特殊的，就用到了 compose 的 ConstraintLayout 布局，在使用的时候多注意控件 id 就可以了。

```
@ExperimentalMaterialApi
@Composable
private fun TreeChildList(viewModel: BottomTabBarViewModel,pageIndex:Int,cid:Int,pVM: ProjectViewModel){

    val dataList by pVM.treeChild.observeAsState()

    LaunchedEffect(key1 = pageIndex, block = {
        pVM.getTreeChildList(cid = cid)
    })

    if (dataList !== null && dataList?.size !== 0){
        initTreeChild(viewModel,dataList!!.datas!!,pVM)
    }
}

@ExperimentalMaterialApi
@Composable
private fun initTreeChild(viewModel: BottomTabBarViewModel,dataList:List<TreeChildDetailEntity>,pVM: ProjectViewModel){
    LazyColumn(modifier = Modifier.fillMaxSize()){
        itemsIndexed(dataList){ index: Int, item: TreeChildDetailEntity ->
            TreeChildItem(viewModel,entity = item,pVM)
        }
    }
}

@SuppressLint("UnrememberedMutableState")
@ExperimentalMaterialApi
@Composable
private fun TreeChildItem(viewModel: BottomTabBarViewModel,entity: TreeChildDetailEntity,pVM: ProjectViewModel){
    val openWebView by pVM._isToWebview.observeAsState()
    val url by pVM.webviewUrl.observeAsState()
    if (openWebView == true && !url.isNullOrBlank()){
        pVM._isToWebview.value = false
        viewModel.openWebViewPage = url
    }
    Card(
        modifier = Modifier
            .padding(10.dp, 10.dp, 10.dp, 0.dp)
            .clickable {
                pVM._isToWebview.value = true
                pVM.webviewUrl.value = entity.link
```

```
        },
    elevation = 2.dp,
    shape = RoundedCornerShape(10.dp)) {
    ConstraintLayout(modifier = Modifier.fillMaxWidth()) {
        val (img,title,desc,time,author) = createRefs()
        Image(
            painter = rememberImagePainter(entity.envelopePic),
            modifier = Modifier
                .width(120.dp)
                .height(150.dp)
                .padding(10.dp, 10.dp, 10.dp, 10.dp)
                .constrainAs(img) {},
            contentScale = ContentScale.Crop,
            contentDescription = null
        )

        Text(modifier = Modifier
            .padding(0.dp, 10.dp, 32.dp, 0.dp)
            .constrainAs(title) {
                top.linkTo(img.top)
                start.linkTo(img.end)
            },
            text = entity.title,
            fontSize = 14.sp,
            color = Color.Black,
            maxLines = 2,
            overflow = TextOverflow.Ellipsis)

        Text(modifier = Modifier
            .padding(0.dp, 10.dp, 32.dp, 0.dp)
            .constrainAs(desc) {
                top.linkTo(title.bottom)
                start.linkTo(img.end)
            },
            text = entity.desc,
            fontSize = 12.sp,
            color = Color.Gray,
            maxLines = 2,
            overflow = TextOverflow.Ellipsis)


        Text(text = entity.niceShareDate,
            fontSize = 12.sp,
            color = Color.Gray,modifier = Modifier
```

```
                .constrainAs(time) {
                    bottom.linkTo(img.bottom)
                    start.linkTo(img.end)
                }
                .padding(vertical = 10.dp))

        Text(text = "${entity.author}",
            modifier = Modifier
                .padding(8.dp)
                .border(
                    1.dp, color = Color(R.color.b_666),
                    RoundedCornerShape(8.dp)
                )
                .width(100.dp)
                .padding(horizontal = 8.dp, vertical = 2.dp)
                .constrainAs(author) {
                    bottom.linkTo(img.bottom)
                    end.linkTo(parent.end)
                },
            color = Color.Gray,
            fontSize = 12.sp,
            maxLines = 1,
            textAlign = TextAlign.Center,
            overflow = TextOverflow.Ellipsis
        )
    }
  }
}
```

### 3.1.2 使用 HorizontalPager 加载页面

在使用 HorizontalPager 的时候要注意 count 和 state 参数就可以了。

```
HorizontalPager(
        count = treeList!!.size,
        state = pagerState,                                    //用于
控制或观察 viewpage 状态的状态对象。
        modifier=Modifier.padding(top = 4.dp),                 //修饰符
        reverseLayout=false,                                   //反转
滚动和布局的方向,为 true 时第一个页面在最后
        itemSpacing=2.dp,                                      //水
平间距
        verticalAlignment= Alignment.CenterVertically          //垂直对
齐
```

) { page ->                                                    //
页面内容描述
TreeChildList(viewModel,pageIndex = pagerState.currentPage,cid
= treeList!![pagerState.currentPage].id,pVM = pVM)
}

## 4. Compose 中 Webview 的使用

在 android 中我们要记在一个网页链接都是使用 webview 进行加载，但是在 compose 里面没有对应的组件，那么还是要使用 Webview 进行加载；这里就需要使用到 AndroidView 这个组件了，让我们在 compose 中可以使用原生 android 的控件，这里就来使用一下 Webview。

## 4.2 AndroidView 的属性

先来看一下 AndroidView 的属性

```
@Composable
fun <T : View> AndroidView(
    factory: (Context) -> T,
    modifier: Modifier = Modifier,
    update: (T) -> Unit = NoOpUpdate
)
```

1.modifier

2.factory 主要就是这个参数，实现我们使用 android 原生 view 的代码块

### 4.1.1 使用 Webview

既然是在 compose 里面使用，所以方法还是要被@Composable 标注起来，再传入一个 String 类型的 url 就可以了；然后 webview 的配置可以和在 android 中使用时的配置一样。

```
fun WebViewPage(modifier: Modifier = Modifier,linkUrl:String){
    Column(modifier = Modifier.fillMaxSize()) {
        AndroidView(modifier = Modifier.fillMaxSize(),
            factory = {
                val webView = WebView(it)
                webView.settings.javaScriptEnabled = true
                webView.settings.javaScriptCanOpenWindowsAutomatically = true
                webView.settings.domStorageEnabled = true
                webView.settings.loadsImagesAutomatically = true
                webView.settings.mediaPlaybackRequiresUserGesture = false
                webView.webViewClient = WebViewClient()
                webView.loadUrl(linkUrl)
                webView
            })
    }
}
```

配置完成了再点击 item 的时候打开当前 pager 就行了；但是有一个问题，知道 compose 是在当前页面打开，所以如果在点击 item 的时候加载这个 webview，那么他就会直接在 item 那里进行加载，而不是跳转到新页面，这里就需要将 Webview 的 pager 放置到首页，然后通

过 viewmodel 记录状态进行打开。

在 viewmodel 中记录打开状态：

```
var openModule: String? by mutableStateOf(null)
var openWebViewPage: String? by mutableStateOf(null)
```

如果符合条件 viewModel.openWebViewPage !== null，则加载 webview 的页面：

```
val openOffset by animateFloatAsState(
        if (viewModel.openModule == null) {
            1f
        } else {
            0f
        }
)
fun Modifier.percentOffsetX(percent: Float) = this.layout { measurable, constraints ->
        val placeable = measurable.measure(constraints)
        layout(placeable.width, placeable.height) {
            placeable.placeRelative(IntOffset((placeable.width * percent).roundToInt(), 0))
        }
}
if (viewModel.openWebViewPage !== null){
         WebViewPage(Modifier.percentOffsetX(openOffset),linkUrl = viewModel.openWebViewPage!!)
}
```

在点击 item 的时候设置 viewmodel 中打开状态的值：

```
viewModel.openWebViewPage = url//这里是设置为要打开的链接，具体可以按照自己需求设置
```

以上就是在 compose 中使用 android view 的方法了。