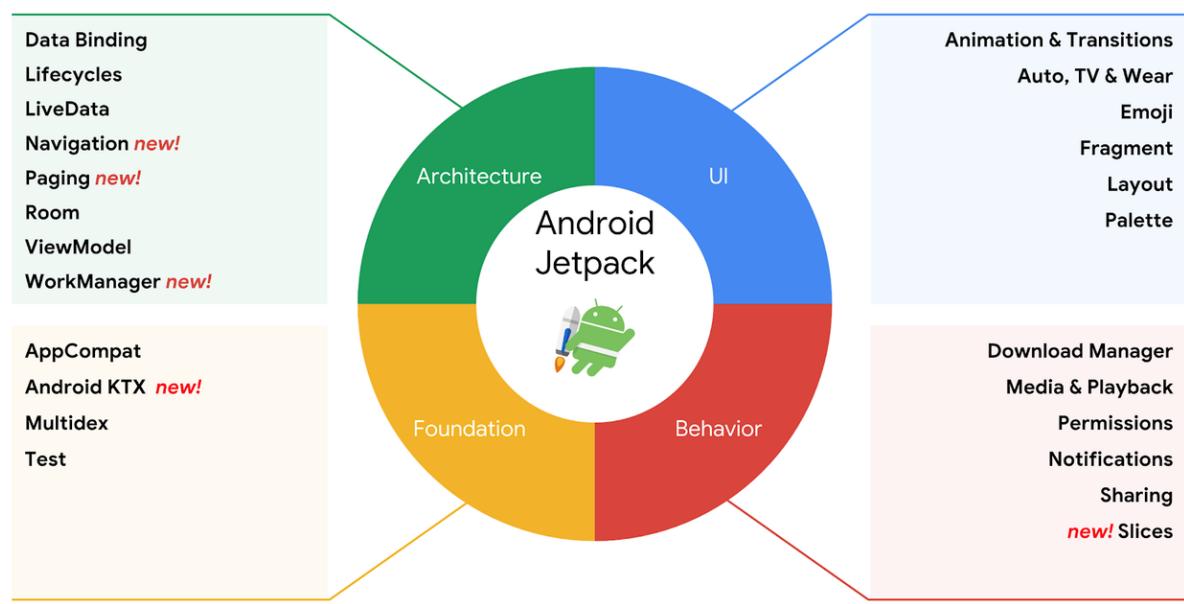


Jetpack



1. 什么是Jetpack

Jetpack就是Google官方推出的一套方便开发者的库。

Android Jetpack

Jetpack 是一个由多个库组成的套件，可帮助开发者遵循最佳做法、减少样板代码并编写可在各种 Android 版本和设备中一致运行的代码，让开发者可将精力集中于真正重要的编码工作。



其致力于

- 遵循最佳做法

iHeartRadio 使用 Android 架构组件创建更清晰、更精简的代码库

iHeartRadio 是一家总部位于纽约市的公司，该公司在一个应用中提供无限量的音乐和成千上万的电台。该公司的业务包括无线电广播、线上服务、移动、数字和社交媒体、现场音乐会和活动、整合、音乐研究服务和独立媒体代理。

自 2008 年推出以来，该应用的全球下载量已超过了 10 亿次。不过，到 2017 年底，代码库已经在持续老化，而且管理代码和集成新功能也被证明是难以完成的事情。



<https://developer.android.google.cn/stories/apps/iheartradio>

- 减少样板代码

Monzo 利用 CameraX 缩减了 9,000 多行代码并使注册流程中的访问者流失率降低了 5 倍

Monzo 是一家银行服务公司并提供了同名应用，仅在移动设备上提供数字金融服务。他们的使命是向每个人传授生财之道。为了完成新客户注册，Monzo 应用会拍摄身份证明文件（例如护照、驾照或身份证件）的图片，并拍摄自拍视频来证明身份证明文件属于申请者。

<https://developer.android.google.cn/stories/apps/monzo-camerax>

- 减少不一致

A screenshot of the Monzo app interface showing two testimonial cards. Each card has a title '赞誉' (Praise), a quote, and the name of the person who provided the testimonial.

“我们正在通过 Android 架构组件重新构造整个应用。很高兴采用一种 Google 认可的、有条理且整洁的方式构建 Android 应用，使支持配置更改变得更容易。”

Drew Hannay, [LinkedIn](#) 的资深软件工程师

“Room 可让我们非常轻松地创建数据库表格和 DAO，因此我们可以快速构建自己的产品。而且，注重可测试性对我们来说至关重要。”

Demian Insung Hwang, [KakaoTalk](#) 的 KakaoTalk 开发者

“借助 Android 架构组件，我们在新功能的开发上实现了更高的灵活性和更短的周转时间。随着越来越多的开发者开始使用它，我们的整体速度也在不断提升。”

Vishwanath Ramarao, [Hike](#) 首席技术官

[查看案例研究](#)

“我们很喜欢 ViewModel 和 LiveData！我们的代码明显变得更加简洁、稳定和易读，并且代码架构实现了完美统一。稳定性也得到了提升！”

Zheng Songyin, [BeautyPlus](#) 的高级开发经理

[查看案例研究](#)

<https://developer.android.google.cn/jetpack/testimonials>

总之不管男女老少用过都说好

2. 在项目中引入Jetpack

没想到吧，当我们创建App的时候build.gradle已经为我们添加了Jetpack的支持。

在应用中使用 Jetpack 库

所有 Jetpack 组件都可在 [Google Maven 代码库](#) 中找到。

打开项目的 `build.gradle` 文件并添加 `google()` 代码库，如下所示：

```
Groovy   Kotlin
allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

⚠ 警告：JCenter 代码库已于 2021 年 3 月 31 日变为只读代码库。如需了解详情，请参阅 [JCenter 服务更新](#)。

在引入`google()`之后便可以在`dependences`添加对应的Jetpack组件了。比如`LiveData`, `Lifecycle`, `ViewModel`, `Navigation`.....

```
dependencies {
    val lifecycle_version = "2.2.0"

    implementation("androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version")
    implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version")
    ...
}
```

.....然后就是一番感人的调包环节了。

3. 细化Jetpack组件的使用

- `LiveData`
- `ViewModel`
- `Lifecycle`
- `Room`
- `Navigation`
- `DataBinding/ViewBinding`
- `Dagger2/Hilt`

ViewModel

1. 什么是ViewModel

`ViewModel` 类旨在以注重生命周期的方式存储和管理界面相关的数据。`ViewModel` 类让数据可在发生屏幕旋转等配置更改后继续留存。

借用google的一句话就是缓存数据的，当我的Activity发生配置变化时候会重新调用`onCreate`方法创建新的一个Activity的实例。这会导致屏幕内的数据丢失。这很不符合用户预想中的使用，所以通常情况下我们会通过 `onSaveInstanceState()` 来保存并拯救丢失的数据。但是 `onSaveInstanceState()` 只可以序列化再反序列化的少量数据，而不适合数量可能较大的数据，

所以它不太适合存储整个页面的数据。所以就有了ViewModel，**但ViewModel并不是onSaveInstanceState()的替代品。**

2.ViewModel的初步使用

Code

Tips: 代码在

com/example/viewmodeldemo/MainActivity.kt,
com/example/viewmodeldemo/ui/activity/vm/MainViewModel.kt
文件中

step1 创建ViewMode

```
package com.example.viewmodeldemo

import androidx.lifecycle.ViewModel

/**
 * @author Zhiqiang Tu
 * @time 2021/7/19 11:18
 * @signature 我将追寻并获取我想要的答案
 */

class MainViewModel : ViewModel{
    var number:Long = 0
        private set
    fun plusOne(){
        number++
    }
    fun plusTwo(){
        number+=2
    }
}
```

step2 在视图组件(Activity,Fragment,...)中获取ViewModel的实例

```
package com.example.viewmodeldemo

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.lifecycle.ViewModelProvider
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
    //懒加载viewModel的实例
    private val viewModel:MainViewModel by lazy {
        ViewModelProvider(this,ViewModelProvider.NewInstanceFactory()).get(MainViewModel::class.java)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
```

```
initView()
//这种写法可能会出现空指针，建议使用viewBinding/DataBinding
plus_one.setOnClickListener {
    viewModel.plusOne()
    updateTextView()
}
plus_two.setOnClickListener {
    viewModel.plusTwo()
    updateTextView()
}
}
//更新视图
private fun updateTextView() {
    show_number.text = viewModel.number.toString()
}
//初始化视图
private fun initView() {
    updateTextView()
}

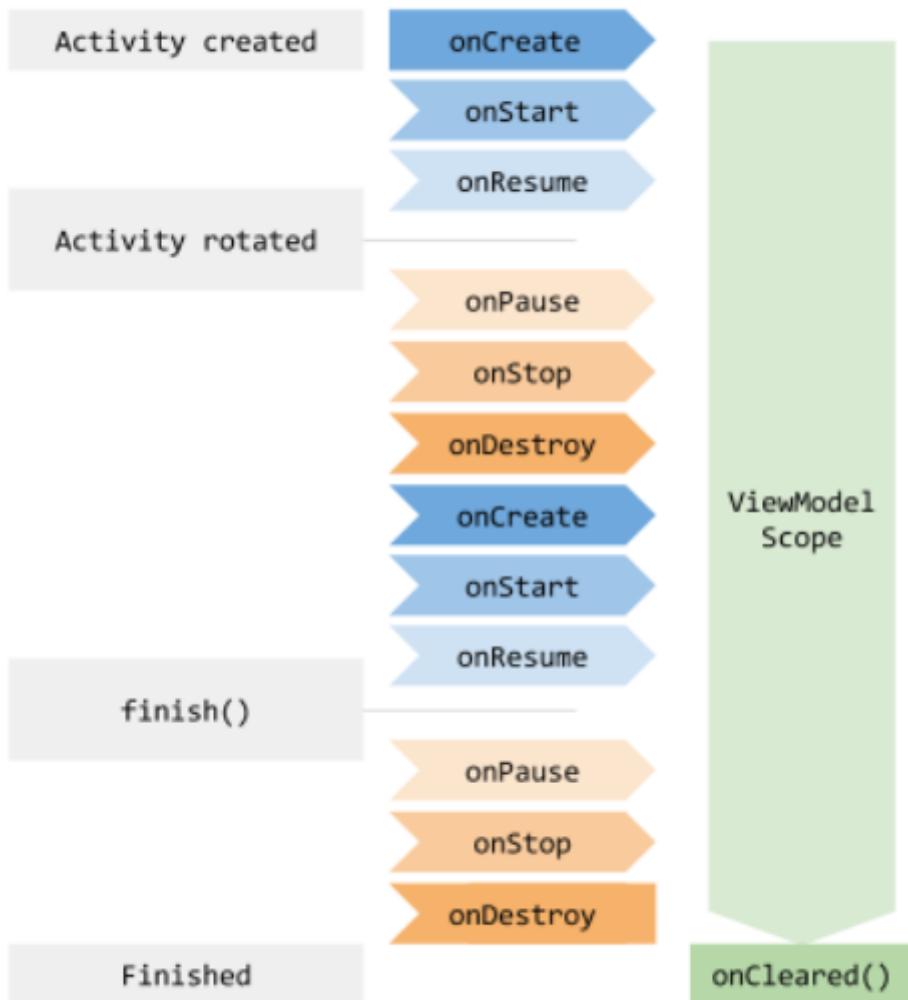
}
```

3. ViewModel的进一步探究

1. ViewModel的生命周期

`ViewModel` 对象存在的时间范围是获取 `ViewModel` 时传递给 `ViewModelProvider` 的 `Lifecycle`。`ViewModel` 将一直留在内存中，直到限定其存在时间范围的 `Lifecycle` 永久消失：对于 Activity，是在 Activity 完成时；而对于 Fragment，是在 Fragment 分离时。

图 1 说明了 Activity 经历屏幕旋转而后结束时所处的各种生命周期状态。该图还在关联的 Activity 生命周期的旁边显示了 `ViewModel` 的生命周期。此图表说明了 Activity 的各种状态。这些基本状态同样适用于 Fragment 的生命周期。



您通常在系统首次调用 Activity 对象的 `onCreate()` 方法时请求 `ViewModel`。系统可能会在 Activity 的整个生命周期内多次调用 `onCreate()`，如在旋转设备屏幕时。`ViewModel` 存在的时间范围是从您首次请求 `ViewModel` 直到 Activity 完成并销毁。

也就是说ViewModel的生命周期依托于ViewModelProvider传入的lifecycle参数，而lifecycle接口又由Activity, Fragment等实现，故ViewModel会和视图组件间接性的形成联系。并且在lifecycle组件第一次调用onCreate后初始化ViewModel（在之后的配置变化如屏幕旋转等都不会再次创建ViewModel），等到lifecycle组件调用了onDestroy方法，并且彻底凉透了，ViewModel才会调用onCleared释放内存。

总体来说

- `ViewModel`依赖于`Lifecycle`组件。在`Lifecycle`组件利用`ViewModelProvider`创建`ViewModel`实例的时候建立联系，并在`Lifecycle`组件第一次调用`onCreate`时候创建`ViewModel`，在`Lifecycle`组件彻底凉透了再释放`ViewModel`内存。
- `ViewModel`的生命周期长于`Activity`。我们不能让`ViewModel`持有`Lifecycle`组件。否者会发生内存泄漏。

2.ViewModel的种类

- 普通`ViewModel`
- 略
- `AndroidViewModel`

这是一个具有`application`的`ViewModel`除此之外与其他的`ViewModel`并没有什么不同（这个AndriodViewModel并不是说生命周期绑定的`application`, 它生命周期绑定的还是`this`（lifecycle）, 只是构造函数中被传入了一个`application`参数。）

Code

Tipes:

代码在

`com/example/viewmodeldemo/ui/activity/vm/DemoAndroidViewModel.kt`,
`com/example/viewmodeldemo/ui/activity/presentation/AndroidViewModelActivity.kt`
文件中

step1 创建`ViewModel`

```
package com.example.viewmodeldemo.vm

import android.app.Application
import androidx.lifecycle.AndroidViewModel

/**
 * @author zhiQiang Tu
 * @time 2021/7/19 21:06
 * @signature 我们不明前路，却已在路上
 */
class DemoAndroidViewModel(application: Application) :
    AndroidViewModel(application) {
    val mApplication = getApplication<Application>()

}
```

step2 实例化`AndroidViewModel`

```
class AndroidViewModelActivity : AppCompatActivity() {
    private val viewModel:DemoAndroidViewModel by lazy {
        ViewModelProvider(this,ViewModelProvider.AndroidViewModelFactory(application))
            .get(DemoAndroidViewModel::class.java)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_android_view_model)
    }
}
```

- SharedViewModel

SharedViewModel实现了同Activity上的Fragment之间的数据共享。

这就是一个以Activity为Lifecycle的Fragment的ViewModel。

有点绕，也就是说这个ViewModel是给Fragment用的但是创建ViewModelProvider用的ViewModelStoreOwner却是Fragment所Attach的activity。

因为只有将ViewModelStoreOwner变成activity才能实现fragment间数据的共享。

```
ViewModelProvider(@NonNull viewModelStoreOwner owner, @NonNull Factory
factory)
```

Code

Tips: 代码在

com/example/viewmodeldemo/ui/activity/presentation/SharedViewModelActivity.kt,

com/example/viewmodeldemo/ui/fragment/presentation

文件中

step1 创建ViewModel

```
package com.example.viewmodeldemo.ui.fragment.vm

import androidx.lifecycle.ViewModel
import com.example.viewmodeldemo.ui.fragment.model.DemoData

/**
 * @author zhiQiang Tu
 * @time 2021/7/19 21:46
 * @signature 我们不明前路，却已在路上
 */
class SharedViewModel: ViewModel() {
    //测试
    var data:DemoData = DemoData(0,"data")
}
```

step2 创建Fragment并实例化ViewModel

```
package com.example.viewmodeldemo.ui.fragment.presentation

import android.os.Bundle
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.lifecycle.ViewModelProvider
import com.example.viewmodeldemo.R
import com.example.viewmodeldemo.ui.fragment.vm.SharedViewModel
import kotlinx.android.synthetic.main.fragment_demo01.*

private const val TAG = "DemoFragment01"

class DemoFragment01 : Fragment() {
    private val viewModel by lazy {
        viewModelProvider(requireActivity()).get(SharedViewModel::class.java)
    }
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_demo01, container, false)
    }
}
```

```

    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        textView.text = "当前viewModel为$viewModel"
        textView2.text = "当前data为${viewModel.data}"
        Log.e(TAG, "当前viewModel为$viewModel" )
        Log.e(TAG, "当前data为${viewModel.data}")
    }

}

```

```

package com.example.viewmodeldemo.ui.fragment.presentation

import android.os.Bundle
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.lifecycle.ViewModelProvider
import com.example.viewmodeldemo.R
import com.example.viewmodeldemo.ui.fragment.vm.SharedViewModel
import kotlinx.android.synthetic.main.fragment_demo02.*
import kotlin.math.log

private const val TAG = "DemoFragment02"

class DemoFragment02 : Fragment() {
    private val viewModel by lazy {
        ViewModelProvider(requireActivity()).get(SharedViewModel::class.java)
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_demo02, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        textView4.text = "当前viewModel为$viewModel"
        textView5.text = "当前data为${viewModel.data}"
        Log.e(TAG, "当前viewModel为$viewModel" )
        Log.e(TAG, "当前data为${viewModel.data}")
    }

}

```

我们在两个Fragment拿到的ViewModel和ViewModel的数据的hashCode是一样的

```

E/DemoFragment01: 当前viewModel为com.example.viewmodeldemo.ui.fragment.vm.SharedViewModel@c5e387d
E/DemoFragment01: 当前data为com.example.viewmodeldemo.ui.fragment.model.DemoData@7025372
E/DemoFragment02: 当前viewModel为com.example.viewmodeldemo.ui.fragment.vm.SharedViewModel@c5e387d
E/DemoFragment02: 当前data为com.example.viewmodeldemo.ui.fragment.model.DemoData@7025372

```

- 自定义构造器的ViewModel

从之前的几种ViewModel中我们可以分成两类：

- 一类是默认构造含函数的ViewModel比如 `sharedViewModel`，最基础的ViewModel。
- 另外一就是非默认构造函数的ViewModel比如 `AndroidViewModel`。

那如何创建一个自定义的构造函数的 `ViewModel` 呢？

那不简单，这样嘛

```
//ViewModel  
class MyViewModel(val myData: Data): ViewModel()  
  
//初始化  
val viewModel = MyViewModel(myData)
```

我竟无法反驳。

值得注意的是当我们创建一个ViewModel的时候是利用的 `ViewModelProvider` 创建的，不是直接 `MyViewModel(myData)` 这样new出来，所以上述的方法貌似没什么用。

回归 `ViewModelProvider` 上看看

```
public ViewModelProvider(@NonNull ViewModelStoreOwner owner, @NonNull  
Factory factory) {  
    this(owner.getViewModelStore(), factory);  
}
```

你发现了什么，他有一个构造方法需要传入两个参数，一个 `owner` 一个 `Factory`，哦所以自定义的构造函数的参数的传递需要靠这玩意了呗。

Code

Tips：代码在

`com/example/viewmodeldemo/ui/activity/vm/CustomViewModel.kt`,

`com/example/viewmodeldemo/ui/activity/presentation/CustomFactoryViewModelActivity.kt`

文件中

step1 创建 `ViewModel`

```
package com.example.viewmodeldemo.ui.activity.vm  
  
import android.util.Log  
import androidx.lifecycle.ViewModel  
import androidx.lifecycle.ViewModelProvider  
import com.example.viewmodeldemo.ui.fragment.model.DemoData  
  
/**  
 * @author zhiQiang Tu  
 * @time 2021/7/20 7:34  
 * @signature 我们不明前路，却已在路上  
 */
```

```

private const val TAG = "CustomViewModel"
class CustomViewModel(var data: DemoData) : ViewModel(){
    fun logData(){
        //检测data是否真的被传入了
        Log.e(TAG, "$data")
    }
}

```

step2 自定义Factory

```

class CustomFactory: ViewModelProvider.Factory{
    //这个方法是ViewModel内部调用创建ViewModel实例的，所以它的任务就只是返回一个ViewModel，你
    //怎么返回它并不关心。
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        val data = DemoData(0, "data")
        val customViewModel = CustomViewModel(data)
        return customViewModel as T
    }
}

```

step3 在owner组件中初始化ViewModel实例

```

package com.example.viewmodeldemo.ui.activity.presentation

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.lifecycle.ViewModelProvider
import com.example.viewmodeldemo.R
import com.example.viewmodeldemo.ui.activity.vm.CustomFactory
import com.example.viewmodeldemo.ui.activity.vm.CustomViewModel

class CustomFactoryViewModelActivity : AppCompatActivity() {
    private val viewModel by lazy {
        ViewModelProvider(this, CustomFactory()).get(CustomViewModel::class.java)
    }
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_custom_factory_view_model)
        viewModel.logData()
    }
}

```

注意这个factory必须要传入的哦，不传入就会这样。

```

at com.example.viewmodeldemo.ui.activity.presentation.CustomFactoryViewModelActivity$viewModel$2.invoke(CustomFactoryViewModelActivity.kt:11)
at com.example.viewmodeldemo.ui.activity.presentation.CustomFactoryViewModelActivity$viewModel$2.invoke(CustomFactoryViewModelActivity.kt:11)
at kotlin.SynchronizedLazyImpl.getValue(LazyJVM.kt:74)
at com.example.viewmodeldemo.ui.activity.presentation.CustomFactoryViewModelActivity.getViewModel(CustomFactoryViewModelActivity.kt:11)
at com.example.viewmodeldemo.ui.activity.presentation.CustomFactoryViewModelActivity.onCreate(CustomFactoryViewModelActivity.kt:15)
at android.app.Activity.performCreate(Activity.java:8201)
at android.app.Activity.performCreate(Activity.java:8189)
at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1320)
at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:4033) <8 more...> <1 internal call> <2 more...>
Caused by: java.lang.InstantiationException: java.lang.Class<com.example.viewmodeldemo.ui.activity.vm.CustomViewModel> has no zero argument constructor
at java.lang.Class.newInstance(Native Method)

```

Caused by: java.lang.InstantiationException:
java.lang.Class<com.example.viewmodeldemo.ui.activity.vm.CustomViewModel> **has no zero argument constructor**

不传入默认就是无参构造函数，

```
@NonNull  
@Override  
public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {  
    //noinspection TryWithIdenticalCatches  
    try {  
        return modelClass.newInstance();  
    } catch (InstantiationException e) {  
        throw new RuntimeException("Cannot create an instance of " + modelClass, e);  
    } catch (IllegalAccessException e) {  
        throw new RuntimeException("Cannot create an instance of " + modelClass, e);  
    }  
}
```

内部通过使用get函数里面的class参数进行反射创建ViewModel。

```
private val viewModel by lazy {  
    ViewModelProvider(this).get(CustomViewModel::class.java) }
```

而ViewModel并没有无参构造，这就直接crash了。

3.ViewModel+Ktx扩展

看看下面几个ViewModel的初始化方法。

```
//1  
private val viewModel: DemoAndroidViewModel by lazy {  
    ViewModelProvider(this, ViewModelProvider.AndroidViewModelFactory(application)).g  
et(DemoAndroidViewModel::class.java) }  
  
//2  
private val viewModel by lazy {  
    ViewModelProvider(this).get(CustomViewModel::class.java) }  
  
//3  
private val viewModel by lazy {  
    ViewModelProvider(requireActivity()).get(SharedViewModel::class.java) }
```

太长了对吧。而且这个初始化很模板化。就是

ViewModelProvider(viewModel所联接的组件, factory).get(你所需要创建的ViewModel的class参数)

为了简化viewModel的初始化，ktx有更为简单的扩展。

```
def lifecycle_version = "2.4.0-alpha02"  
// ViewModel  
implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version")  
  
//ktx  
implementation("androidx.activity:activity-ktx:1.2.2")  
implementation("androidx.fragment:fragment-ktx:1.3.4")
```

然后之前冗长的代码就变成了这样，简直爽翻天。

```
//1  
private val viewModel:DemoAndroidViewModel by viewModels()  
  
//2  
private val viewModel:CustomViewModel by viewModels { CustomFactory() }  
  
//3  
private val viewModel:SharedViewModel by activityViewModels()
```

4.ViewModel失效了！

每当我们使用ViewModel的时候我们总是认为：ViewModel一定能帮我们在任何情况下保存好界面的数据，然而真实情况是这样的吗？

进行以下设置

- 打开设置面板



- 选择开发者选项

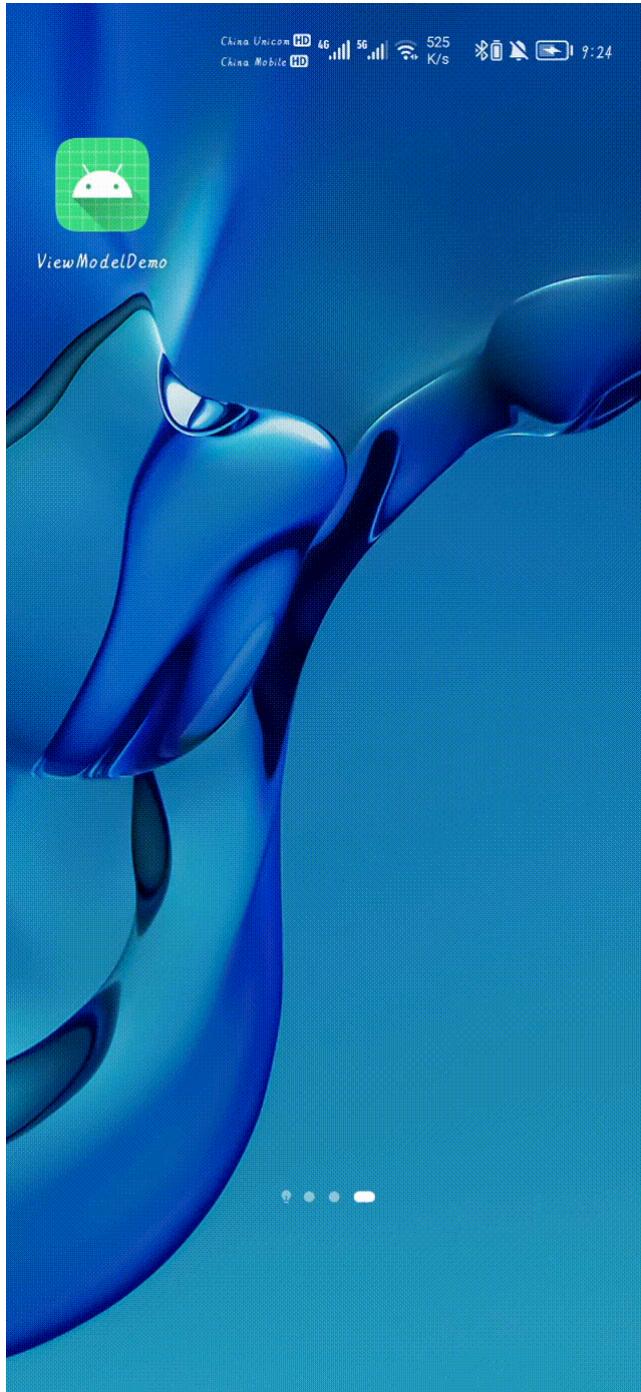


- 选择应用设置，勾选切入后台不保留activity



然后发生了很恐怖的事情，切入后台再回来，数据没了。

效果图（GIF）



这是为什么，原来是当打开开发者设置**不保留后台进程**之后，切入后台之后，Activity会直接被系统杀死了，**并且不调用任何生命周期方法**。连带着ViewModel都挂了，所以数据没法保存。还记得最早的时候说的：**ViewModel不是onSaveInstanceState的替代品吗？**

数量可能较大的数据，所以它不太适合存储整个页面的数据。所以就有了
ViewModel，但ViewModel并不是onSaveInstanceState()的替代品。

系统杀死Activity是不会调用任何什么周期方法的，那我们有什么方法能拯救那些数据呢？

其实是有的，在系统杀死Activity之前它留了一线生机。会调用onSaveInstanceState这会是你恢复数据的希望。我们可以这样写。

Code

Tip：该代码在com/example/viewmodeldemo/MainActivity.kt中

重写onSaveInstanceState把需要存放的数据保存下来

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putString("number", show_number.text.toString())
}
```

然后再onCreate方法中判断一下savedInstanceState是不是空的。若不是空的就把数据取出来。

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    savedInstanceState?.let {
        viewModel.number = (it.get("number") as String).toLong()
    }
    initView()
    setListener()
}
```

效果图 (GIF)

除此之外我们还能使用ViewModel来实现。

不过还得加入一个依赖

```
// Saved state module for ViewModel
implementation("androidx.lifecycle:lifecycle-viewmodel-
savedstate:$lifecycle_version")
```

Code

Tips: 代码位置

com/example/viewmodeldemo/MainActivity.kt,

com/example/viewmodeldemo/ui/activity/vm/MainViewModel.kt

还有一个需要提醒大家的是SavedStateHandle不适合把整个页面的数据都保存下来，它的定位是保存最为重要的一小部分数据与onSaveInstanceState的定位是一样的。

如下。

- The `SavedStateHandle` is very similar to the bundle — it's a key-value map that survives memory-constraint-related process death. For the same reasons given above, you should only store a **small** amount of data in the `SavedStateHandle`. Basically, the `SavedStateHandle` replaces the Bundle.

对之前的MainViewModel稍作修改

```
package com.example.viewmodeldemo.ui.activity.vm

import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
```

```

/**
 * @author ZhiQiang Tu
 * @time 2021/7/19 11:18
 * @signature 我将追寻并获取我想要的答案
 */
private val TAG = "MainViewModel"
class MainViewModel(private val savedStateHandle: SavedStateHandle) :
viewModel() {
    var number: Long = if (savedStateHandle.contains("number")) {
        savedStateHandle.get<Long>("number") !!
    } else {
        savedStateHandle.set("number", 0)
        0
    }

    fun plusOne(){
        number++
        updateSavedStateHandle(number)
    }

    private fun updateSavedStateHandle(number: Long) {
        savedStateHandle.set("number", number)
    }

    fun plusTwo(){
        number+=2
        updateSavedStateHandle(number)
    }
}

```

实例化ViewModel

```

private val viewModel: MainViewModel by viewModels{
    SavedStateViewModelFactory(application, this)
}

```

注意这里的 `SavedStateViewModelFactory` 构造函数需要传入两个参数，一个 `Application`，一个是 `SavedStateRegistryOwner`。

```

public SavedStateViewModelFactory(@Nullable Application application,
    @NonNull SavedStateRegistryOwner owner)

```

`ComponentActivity` 和 `Fragment` 已经实现了 `LifecycleOwner`, `ViewModelStoreOwner`, `SavedStateRegistryOwner` 接口。

其中 `AppCompatActivity` 继承自 `FragmentActivity` 继承自 `ComponentActivity`

```

public class ComponentActivity extends androidx.core.app.ComponentActivity implements
    LifecycleOwner,
    ViewModelStoreOwner,
    SavedStateRegistryOwner,
    OnBackPressedDispatcherOwner {

```

```
public class Fragment implements ComponentCallbacks, OnCreateContextMenuListener, LifecycleOwner,  
    ViewModelStoreOwner, HasDefaultViewModelProviderFactory, SavedStateRegistryOwner,  
    ActivityResultCaller {
```

5. ViewModel的使用建议

- ✗ Don't let ViewModels (and Presenters) know about Android framework classes

不要让ViewModel知晓Android的FrameWork, ViewModel只是用来写点逻辑代码和存数据的。

Ideally, ViewModels shouldn't know anything about Android. This improves testability, leak safety and modularity. A general rule of thumb is to make sure there are no `android.*` imports in your ViewModels (with exceptions like `android.arch.*`). The same applies to presenters.

我所读的文章告诉我要将ViewModel和FrameWork隔离，最好的办法就是在ViewModel中不要导入android的包，（`android.arch`.除外）（也算是个好办法。雾.....）

- Keep the logic in Activities and Fragments to a minimum

尽量不要在Activity, Fragment中写逻辑代码。

这个在刚学Android的时候估计是非常常见的问题，一个Activity 六七百行人的写傻了。

也算是设计模式的一个运用了，在MVVM中推荐将业务逻辑代码写在ViewModel中（这个其实也会存在问题的，后面有讲）。

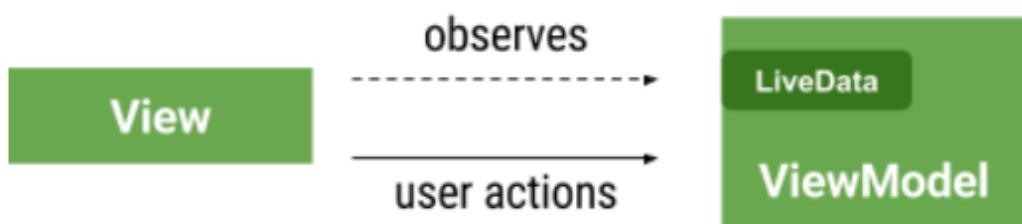
- ✗ Avoid references to Views in ViewModels.

不要将页面组件（Activity, Fragment）放在ViewModel中，这个估计懂得都懂。ViewModel生命周期比Activity等owner要长一点，如果ViewModel持有Activity/Fragment等就会造成内存泄漏嘛。

- Instead of pushing data to the UI, let the UI observe changes to it.

在页面View数据的更新上，不是让ViewModel持有View通过set把数据装载到UI上。而是让UI观察ViewModel中的数据，当数据发生改变后自己更新。（观察者模式也算的上MVVM中很重要的部分了。）

这我得跟你谈谈LiveData的含金量了。



也就是说要让ViewModel持有UI数据包裹的LiveData (不是MutableLiveData哦，LiveData更加符合面向对象的封装性，而且只要不是双向绑定UI根本不会有更改数据的行为存在的所以没必要暴露Mutable)最后在View去观察对应数据的变化。

Distribute responsibilities, add a domain layer if needed.

分配职责，在有需要的时候新加一个domain层。之前有提过在 ViewModel里面写逻辑代码是存在一定问题的。那就是----ViewModel膨胀。如果app的业务逻辑比较复杂，那么就会导致 ViewModel内代码很多。所以很有必要对业务逻辑进行合理的分离。

- Moving some logic out to a presenter, with the same scope as the ViewModel. It will communicate with other parts of your app and update the LiveData holders in the ViewModel.
- Adding a Domain layer and adopting [Clean Architecture](#). This leads to a very testable and maintainable architecture. It also facilitates getting off the main thread quickly. There's a Clean Architecture sample in [Architecture Blueprints](#).

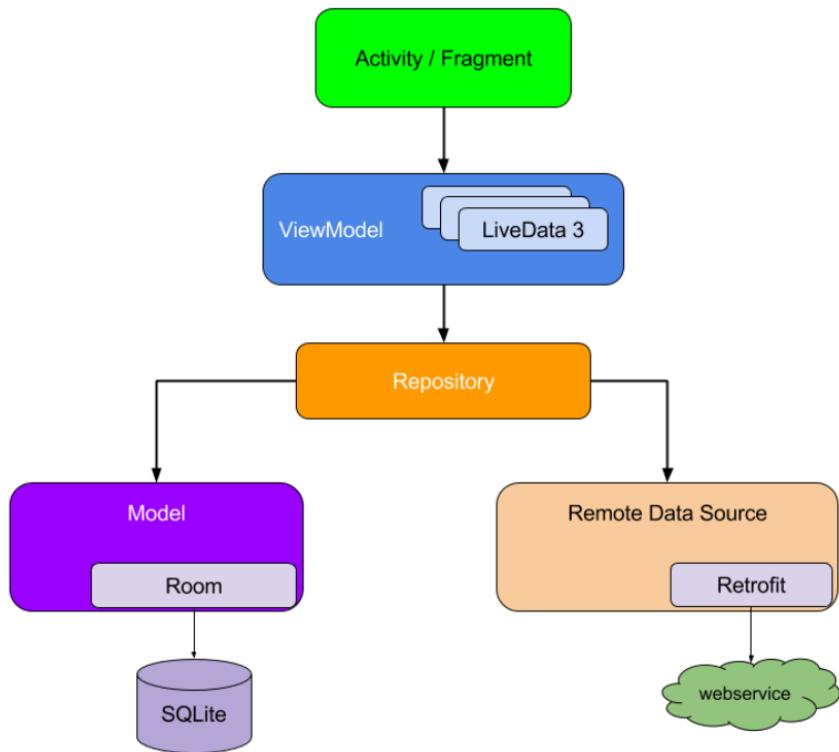
它给出了两种方法：

- 将一些业务逻辑分配到presenter中去，该presenter和ViewModel有相同的作用域。并且和app的其他板块进行交互并且更新ViewModel中的LiveData。
- 像Clean Architecture一样采取添加一个domain层。使得架构更加具有可测试性和可维护性。[\(Clean Architecture我不懂，真的，需要的自取 Clean Architecture\)](#)

Add a data repository as the single-point entry to your data

添加一个data repository，并且repository对数据的使用是单向的。

就和google推荐的架构差不多，repository对Model和Romote的联接是单向箭头。



数据的获取路径它给出了3种

1. Remote: network or the Cloud

2. Local: database or file

3. In-memory cache

Remote和Local就不必说了，增添了一种In-memory cache（内存缓存）。

最后就是如果你有很多的并且差异很明显的数据，可以选择开辟更多的Repository.

- Expose information about the state of your data using a wrapper or another LiveData.

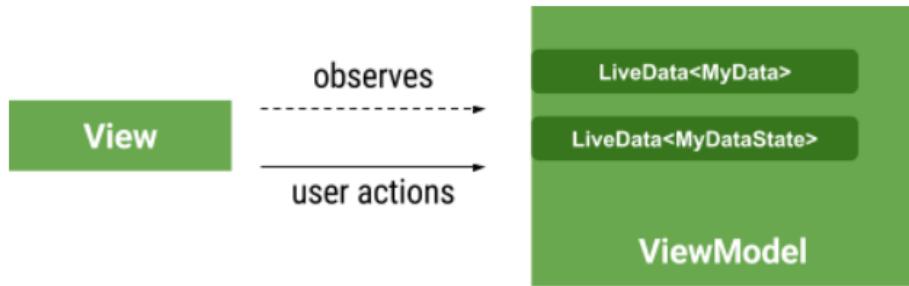
通过包装的方法，或者额外添加一个LiveData来提供数据的状态。

其中数据状态需要包含什么？

`MyDataState` could contain information about whether the data is currently loading, has loaded successfully or failed

状态包含数据是否正在加载，是否加载完成，是否失败等。

- 通过添加额外的LiveData暴露数据状态



其中MyData是数据本身， MyDataState是数据加载的情况。

- 通过装饰模式暴露数据的状态。

附录：公开网络状态

在上文的[推荐应用架构](#)部分中，我们省略了网络错误和加载状态以简化代码段。

本部分将演示如何使用可封装数据及其状态的 `Resource` 类来公开网络状态。

以下代码段提供了 `Resource` 的实现示例：

`Resource`

```
// A generic class that contains data and status about loading this data.
sealed class Resource<T>(
    val data: T? = null,
    val message: String? = null
) {
    class Success<T>(data: T) : Resource<T>(data)
    class Loading<T>(data: T? = null) : Resource<T>(data)
    class Error<T>(message: String, data: T? = null) : Resource<T>(data, message)
}
```

`data`就是数据本身， `message`是数据的状态信息（这写法不由得想到了MVI架构中的 `ViewState`）

Design events as part of your state. For more details read [LiveData with Snackbar, Navigation and other events \(the SingleLiveEvent case\)](#).

把事件当成状态的一部分。好像这样翻译起来太过绕口。

我认为它想表达的意思是将 `Event` 进行封装。

这个 `Event` 就是一些消费性事件，比如Snackbar弹窗，Toast，点击事件等这些**不具有状态的消费性事件**。

为什么要把它封装起来封装以后又放在哪？

- 我们知道 presentation 层（也就是 Activity 和 Fragment）需要和 ViewModel 交互。
- presentation 层需要发送一个事件给 ViewModel 然后 ViewModel 处理这个事件。
- 比如一个登陆场景，用户点击了 LoginActivity 的 Login Button，然后 LoginViewModel 接受到一个登陆事件，这个登陆事件里面包含了用户名，密码等配置信息。LoginViewModel 根据从这个事件里面获取的用户名密码发送网络请求比对是否与服务器上的一致，最后更改对应的 LiveData。最后 Presentation 观察到 LiveData 变化做出响应（如登陆成功，登陆失败）。
- 当我们回首去看这个登陆流程的时候后我们会发现一个问题，这个登陆事件怎么处理？

- 有些人会直接通过对button设置监听，当点击触发直接调用viewModel里面的方法把需要的参数直接传入进去。不是很推荐，主要是写法太过简洁，不能装*（虽然我找不出什么问题但总是感觉怪怪的）。
- 还有些人会在xml里面的onClick里面通过dataBinding的单项绑定直接调用viewModel对象的方法，不过好像和上面的方法并没有本质区别。也不是很推荐。
- 推荐的方法是通过将这个LoginEvent封装，因为这样Testable（虽然我根本不懂啥是Test，并且逻辑更为清晰）。

```
data class LoginEvent(val username:String, val password:String)
```

然后在需要的时候发送Event到ViewModel里，不过为了保证这个事件是1次性消费事件还得做一些处理，比如。

具体的我就不做多的演示了，这个内容得有两三篇博客那么长，况且我也没看太懂。有兴趣的可以自己看看。

SingleLiveEvent和Event确保事件为单次消费事件

[LiveData with Snackbar, Navigation and other events \(the SingleLiveEvent case\)](#)

利用协程Flow库确保事件为单次消费事件（2021最新解决方案）

[Android SingleLiveEvent Redux with Kotlin Flow](#)

思来想去还是把代码贴一下。

way1

通过继承MutableLiveData创建一个特殊的MutableLiveData。

这个SingleLiveEvent确保了事件为单次的消费性事件。

但是存在线程不安全的问题。

```
/*
 * Copyright 2017 Google Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.android.architecture.blueprints.todoapp;

import android.arch.lifecycle.LifecycleOwner;
```

```
import android.arch.lifecycle.MutableLiveData;
import android.arch.lifecycle.Observer;
import android.support.annotation.MainThread;
import android.support.annotation.Nullable;
import android.util.Log;

import java.util.concurrent.atomic.AtomicBoolean;

/**
 * A lifecycle-aware observable that sends only new updates after
subscription, used for events like
 * navigation and Snackbar messages.
 * <p>
 * This avoids a common problem with events: on configuration change (like
rotation) an update
 * can be emitted if the observer is active. This LiveData only calls the
observable if there's an
 * explicit call to setValue() or call().
 * <p>
 * Note that only one observer is going to be notified of changes.
 */
public class SingleLiveEvent<T> extends MutableLiveData<T> {

    private static final String TAG = "SingleLiveEvent";

    private final AtomicBoolean mPending = new AtomicBoolean(false);

    @MainThread
    public void observe(LifecycleOwner owner, final Observer<T> observer) {

        if (hasActiveObservers()) {
            Log.w(TAG, "Multiple observers registered but only one will be
notified of changes.");
        }

        // Observe the internal MutableLiveData
        super.observe(owner, new Observer<T>() {
            @Override
            public void onChanged(@Nullable T t) {
                if (mPending.compareAndSet(true, false)) {
                    observer.onChanged(t);
                }
            }
        });
    }

    @MainThread
    public void setValue(@Nullable T t) {
        mPending.set(true);
        super.setValue(t);
    }

    /**
     * Used for cases where T is Void, to make calls cleaner.
     */
    @MainThread
    public void call() {
        setValue(null);
    }
}
```

```
    }  
}
```

way2 使用装饰器包裹一层。

```
/**  
 * Used as a wrapper for data that is exposed via a LiveData that represents  
an event.  
 */  
open class Event<out T>(private val content: T) {  
  
    var hasBeenHandled = false  
        private set // Allow external read but not write  
  
    /**  
     * Returns the content and prevents its use again.  
     */  
    fun getContentIfNotHandled(): T? {  
        return if (hasBeenHandled) {  
            null  
        } else {  
            hasBeenHandled = true  
            content  
        }  
    }  
  
    /**  
     * Returns the content, even if it's already been handled.  
     */  
    fun peekContent(): T = content  
}
```

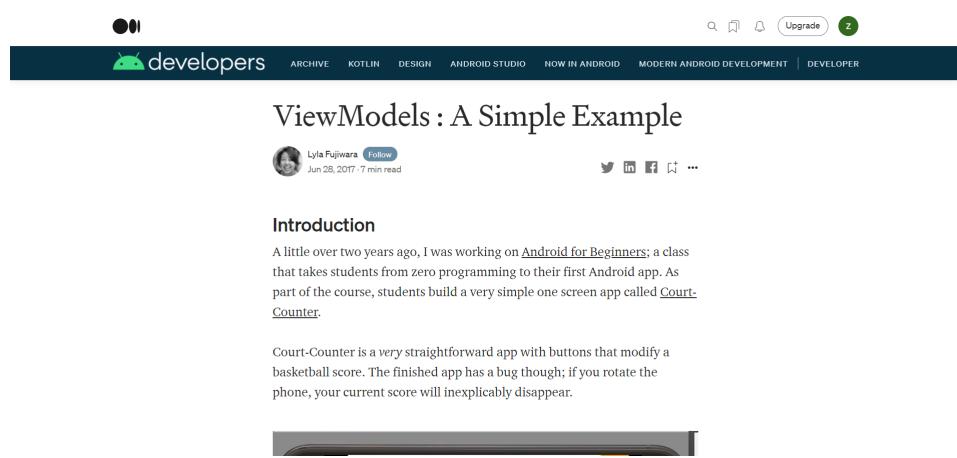
way3 利用协程Flow

这个我没看。

4. 内容来源

什么是ViewModel, ViewModel的初步使用 ---- [Google官方文档](#), [Google官方文档提供的博客](#)

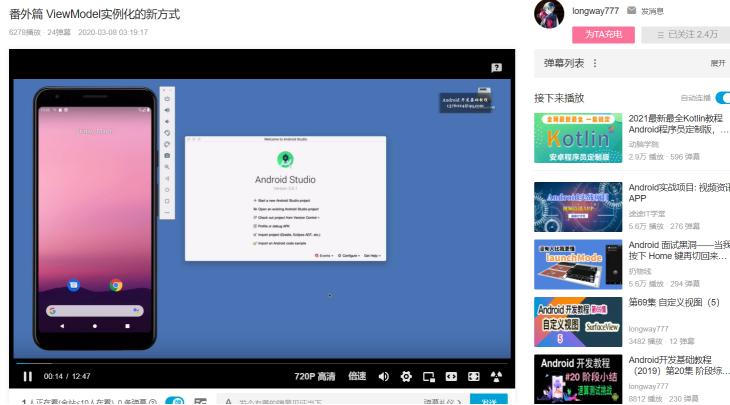
- [ViewModel 和 LiveData：模式 + 反模式](#)



The screenshot shows a blog post titled "ViewModels : A Simple Example" by Lyle Fujiwara. The post discusses the introduction of ViewModels in Android, mentioning their use in the Court Counter app. It includes a brief history of the course and a note about a bug in the app.

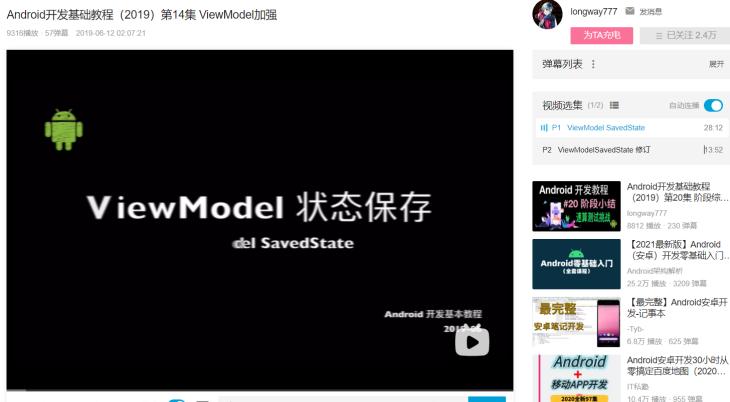
Introduction
A little over two years ago, I was working on [Android for Beginners](#); a class that takes students from zero programming to their first Android app. As part of the course, students build a very simple one screen app called [CourtCounter](#).
CourtCounter is a very straightforward app with buttons that modify a basketball score. The finished app has a bug though; if you rotate the phone, your current score will inexplicably disappear.

ViewModel + Ktx 扩展 ---- [Bilibili](#)



ViewModel的生命周期 ---- [Google官方文档](#)

ViewModel失效了 ---- [Bilibili](#), [Google官方文档提供的博客](#)



• [ViewModel 和 LiveData: 模式 + 反模式](#)

ViewModels: Persistence, `onSaveInstanceState()`, Restoring UI State and Loaders

Introduction

In the [last blog post](#) I explored a simple use case with the new `ViewModel` class for saving basketball score data during a configuration change.

ViewModels are designed to hold and manage UI-related data in a life-cycle conscious way. ViewModels allow data to survive configuration changes such as screen rotations.

At this point, you might have a few questions about the breadth of what

• [ViewModel 和 LiveData：模式 + 反模式](#)

The screenshot shows a blog post on the Android Developers website. The title is "ViewModels and LiveData: Patterns + AntiPatterns". Below the title is a small profile picture of Jose Alcarreca and the date "Sep 13, 2017 - 8 min read". There are social sharing icons and a "..." button. The main content starts with a section titled "Views and ViewModels" and "Distributing responsibilities". It includes a diagram illustrating the architecture layers: "Android OS" (bottom left), "Presentation layer" (center), and "Other layers (domain, data)" (right). In the center, there's a "View" box, a "LiveData/ViewModel" box, and an "Interactors/Repository" box. Arrows show the "View" observing the "LiveData/ViewModel" via "user actions" and the "LiveData/ViewModel" interacting with the "Interactors/Repository".

Navigation

1.什么是Navigation

导航是指支持用户导航、进入和退出应用中不同内容片段的交互。Android Jetpack 的导航组件可帮助您实现导航，无论是简单的按钮点击，还是应用栏和抽屉式导航栏等更为复杂的模式，该组件均可应对。导航组件还通过遵循一套既定原则来确保一致且可预测的用户体验。

也就是说Navigation是用来处理页面之间的跳转的（不知道是否准确，但应该不至于错的离谱）。

在日常开发中我们经常会遇见页面跳转的，比如购物的时候点击商品列表之后会跳转到商品详细信息。点击底部的导航栏会在不同的Fragment页面间进行来回切换，点击抽屉式菜单在页面中来回切换。等等一系列。除此之外我们在进行页面跳转的时候可能还会出现其他的需要加入考虑的事情，**比如参数的传递，比如跳转动画的添加，比如Fragment的回退栈等.....**所以页面跳转一直都不仅仅是 `startActivity(Intent(this, SecondActivity::class.java))` 这么简单。

哪里有困难哪里就有Jetpack，所以就有了Navigation的出现。

2.Navigation一览

Navigation的组成

Navigation由3个部分组成

- **NavGraph**

这是一个XML资源，它描述了页面间的**跳转关系**，从哪里跳转到哪里，需要传入什么参数，跳转过程有什么动画等等。

- **NavController**

NavController是一个特殊的**Fragment**。它是Fragment的容器，我们可以将NavGraph通过XML的形式引入到NavController中，然后NavController会**呈现相应的页面**。

- **NavController**

从Controller可以看出它是一个用于管理的类，管理什么？管理**页面的切换**，管理NavController的视图呈现。

Navigation的优势

- 处理 Fragment 事务。
- 默认情况下，正确处理往返操作。
- 为动画和转换提供标准化资源。
- 实现和处理深层链接。
- 包括导航界面模式（例如抽屉式导航栏和底部导航），用户只需完成极少的额外工作。
- Safe Args - 可在目标之间导航和传递数据时提供**类型安全**的 Gradle 插件。
- `viewModel` 支持 - 您可以将 `viewModel` 的范围限定为导航图，以在图表的目标之间共享与界面相关的信息。

就是navGraphViewModels

```
//就像这样 这样的viewModel的生命周期与传入的navGraph一种
val viewModel:TestViewModel by navGraphViewModels(R.id.nav_demo)
```

此外，您还可以使用 Android Studio 的 [Navigation Editor](#) 来查看和编辑导航图。（应该没有人手打Navigation吧.haha）

3.摸一摸Navigation

也就是Navigation的基本使用吧。

第一步加入依赖

有两种办法

- way 1 去Google官方文档上查看。

[地址](#)

```
dependencies {
    val nav_version = "2.3.5"

    // Java language implementation
    implementation("androidx.navigation:navigation-fragment:$nav_version")
    implementation("androidx.navigation:navigation-ui:$nav_version")

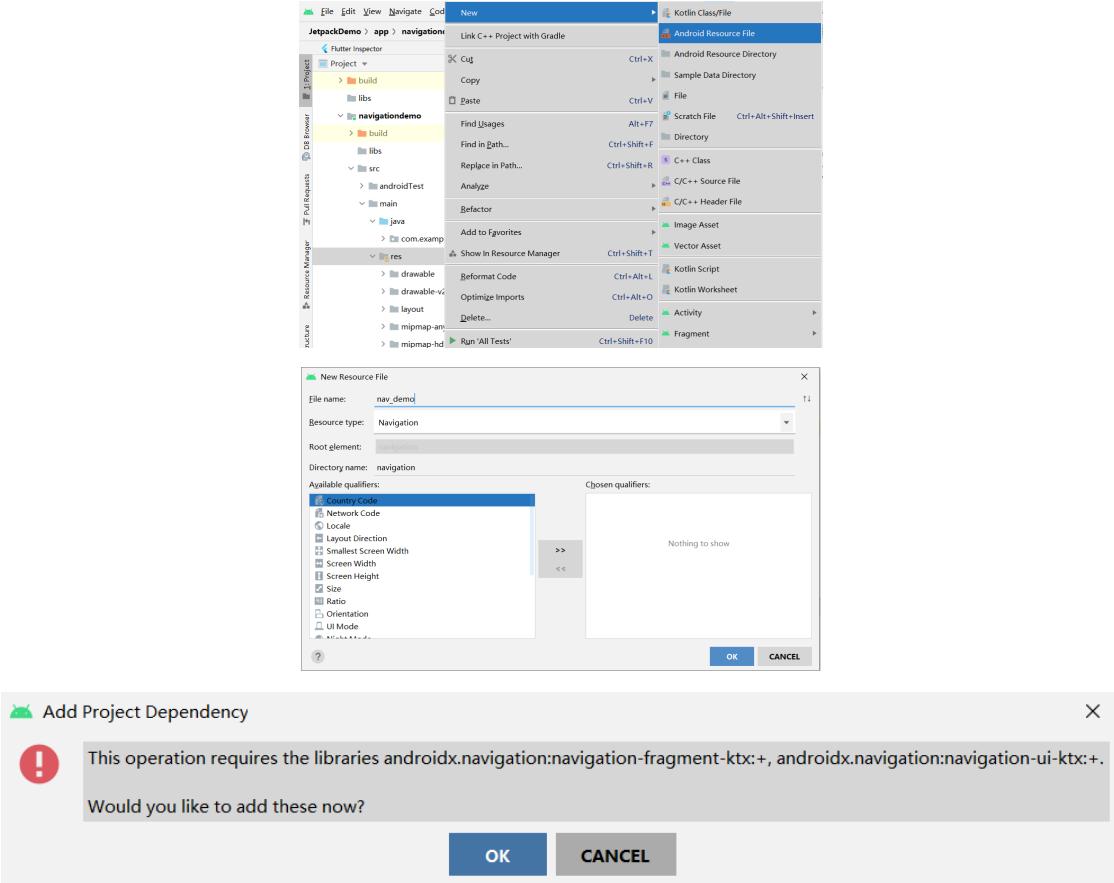
    // Kotlin
    implementation("androidx.navigation:navigation-fragment-ktx:$nav_version")
    implementation("androidx.navigation:navigation-ui-ktx:$nav_version")

    // Feature module Support
    implementation("androidx.navigation:navigation-dynamic-features-
fragment:$nav_version")

    // Testing Navigation
    androidTestImplementation("androidx.navigation:navigation-
testing:$nav_version")

    // Jetpack Compose Integration
    implementation("androidx.navigation:navigation-compose:2.4.0-alpha04")
}
```

- way 2 让Android Studio自己帮我们导入



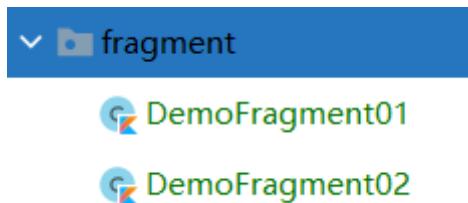
看自动导入了两个依赖

```
dependencies {

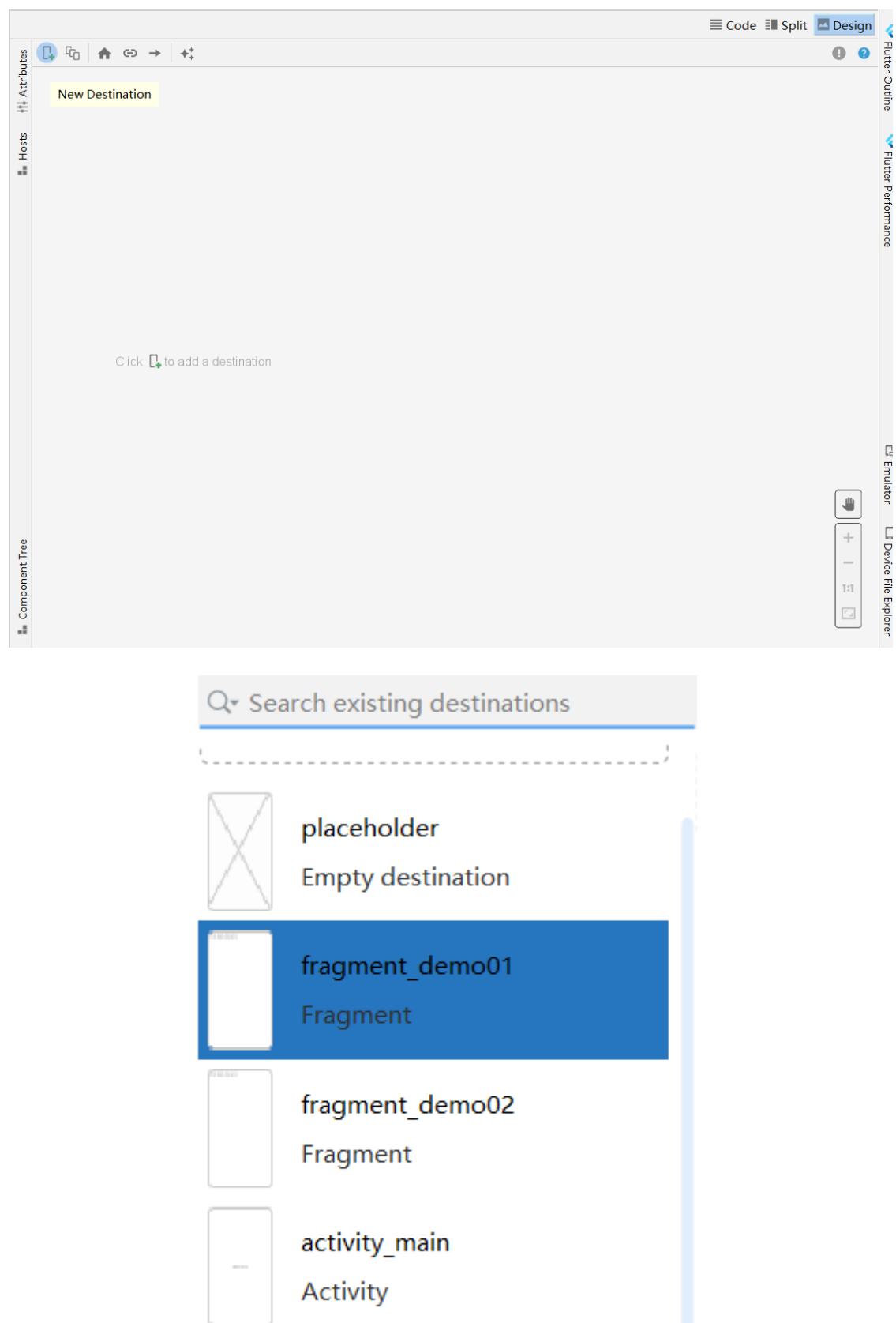
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    implementation 'androidx.core:core-ktx:1.3.1'
    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'com.google.android.material:material:1.2.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.1'
    implementation 'androidx.navigation:navigation-fragment-ktx:2.3.0'
    implementation 'androidx.navigation:navigation-ui-ktx:2.3.0'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
}
```

第二步创建Fragment或者需要跳转的Activity

我这里创建了两个Fragment



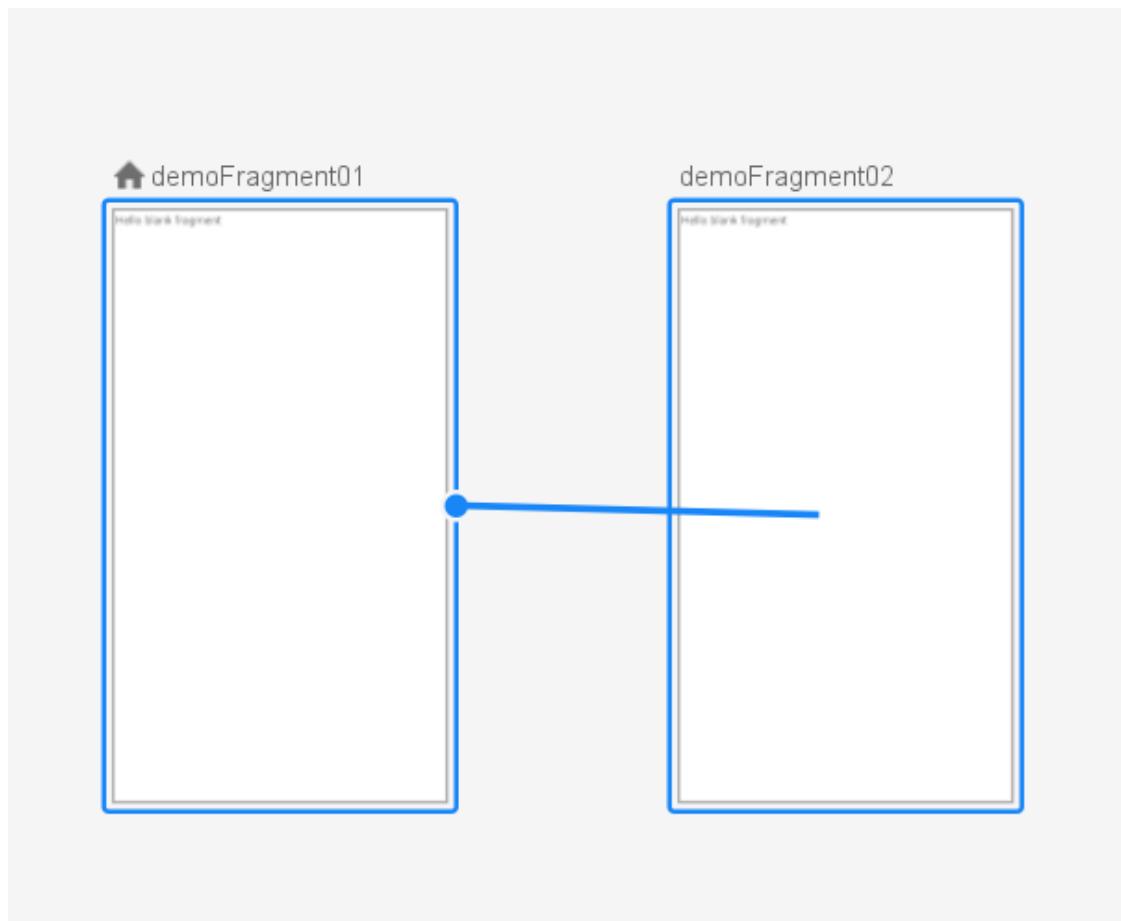
第三步将Fragment或者需要跳转的Activity添加到NavGraph中





第四步通过Navagation Editor建立跳转关系

拖动demoFragment01的小圆点与demoFragment02相连



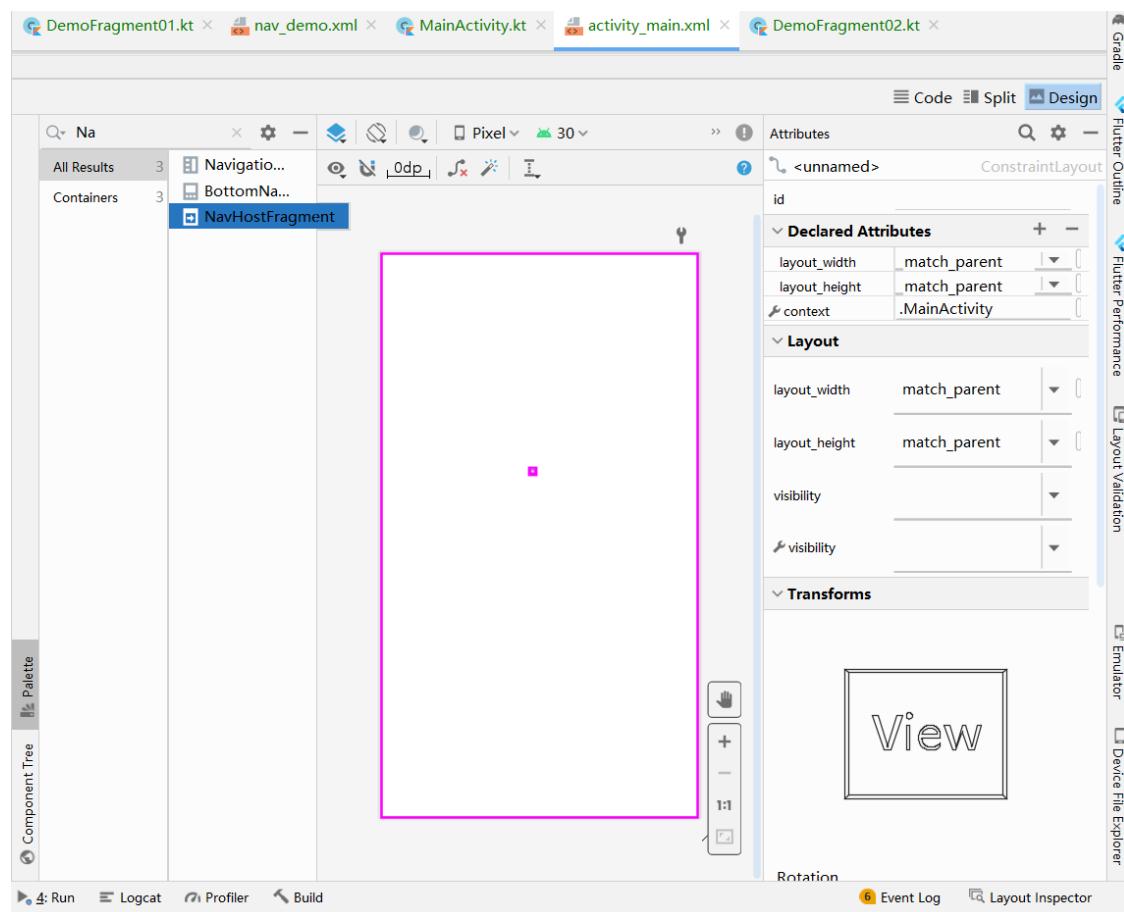
这样一个跳转关系就建立了。



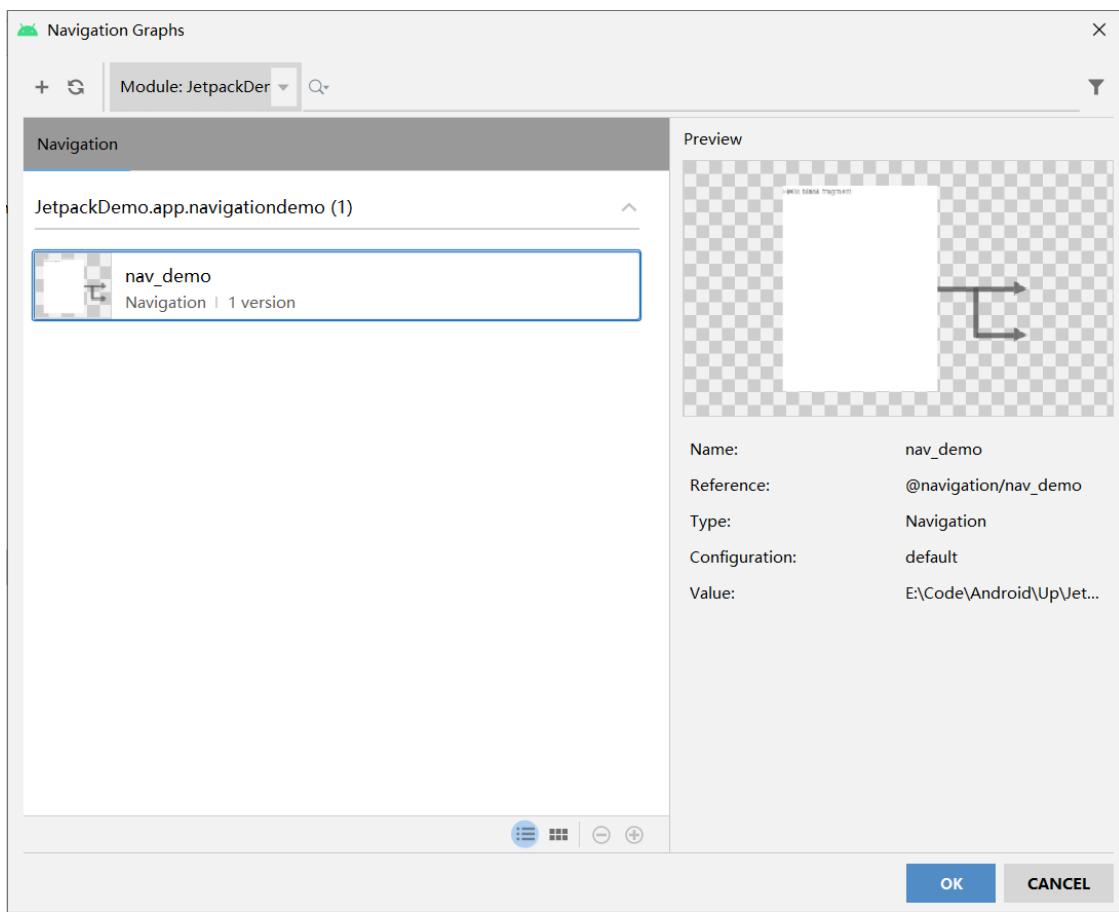
第五步使用NavHostFragment呈现Fragment

之前说过**NavHostFragment**是**Fragment**的容器，它可以展示**Fragment**。前几步我们完成了对跳转关系的配置，**Fragment**的建立，但是并没有将**Fragment**展示出来。也就是说现在的app还是一片空白。

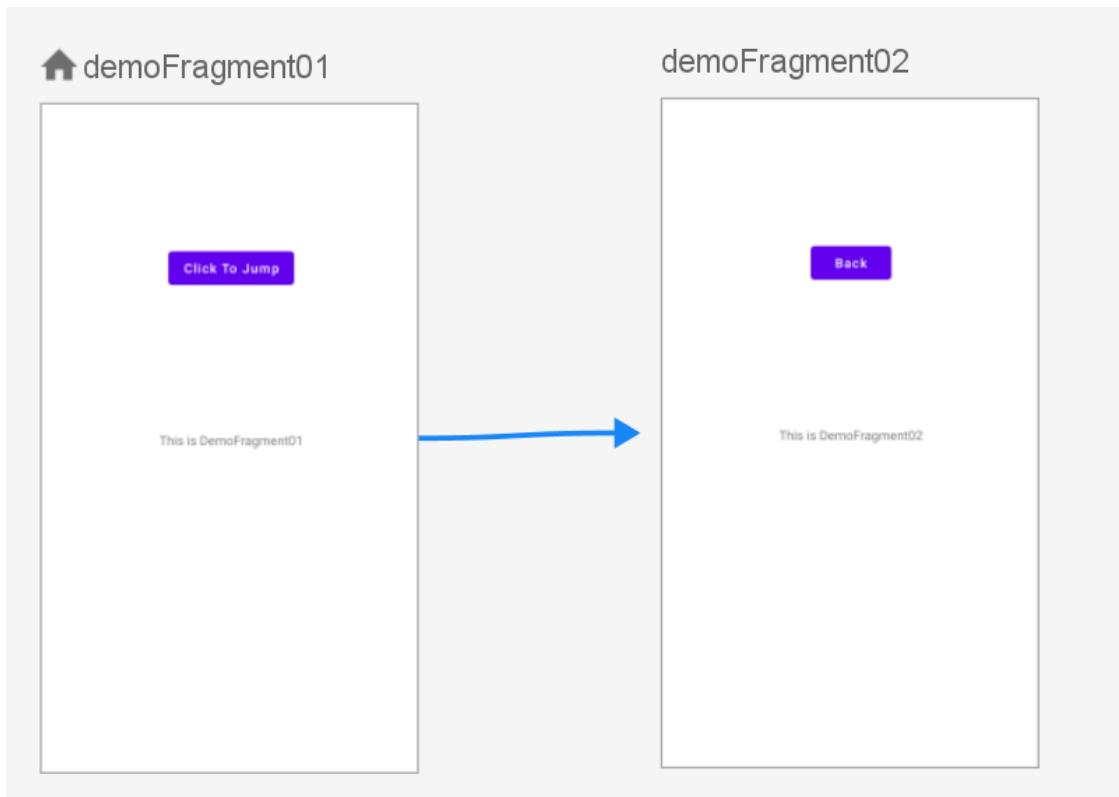
在>MainActivity中添加**NavHostFragment**



选择对应的**NavGraph**(由于**NavHostFragment**需要进行页面的呈现，所以它必须知道页面的跳转配置，这样它才知道什么时候应该呈现什么样的布局。)



稍微写了一下Fragment的界面



现在Fragment已近可以呈现到Activity上了，但是跳转的逻辑还没有写。



Click To Jump

This is DemoFragment01

第六步实现Fragment之间的跳转

对主Fragment的按钮进行监听，点击事件触发以后直接调用navigation的api。先通过Fragment的扩展方法findNavController寻找到当下的NavController实例，在通过调用NavController的navigate方法实现跳转。

也就是说管理跳转的是NavController对象

```
demo01_jump_button.setOnClickListener {
    findNavController().navigate(R.id.action_demoFragment01_to_demoFragment02)
}
```

不算总结的总结

完成一个Navigation的跳转需要完成：

- 添加依赖
- 创建Fragment或者需要跳转的Activity
- 将Fragment或者需要跳转的Activity添加到NavGraph中
- 通过Navagation Editor建立跳转关系
- 将NavHostFragment添加到Activity的XML中

- 通过NavController实现跳转

4.Navigation初级探究

1. NavHostFragment XML布局参数解析

参考自

如果你打开XML布局去寻找NavHostFragment的时候你或许会惊奇。因为并没有NavHostFragment的标签。



有的只是一个 FragmentContainerView

```
<androidx.fragment.app.FragmentContainerView  
    android:id="@+id/fragmentContainerView"  
    android:name="androidx.navigation.fragment.NavHostFragment"  
    android:layout_width="0dp"  
    android:layout_height="0dp"  
    app:defaultNavHost="true"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    app:navGraph="@navigation/nav_demo" />
```

然鹅确是有NavHostFragment这个类的

```
public class NavHostFragment extends Fragment implements NavHost
```

我们可以通过 `NavHostFragment` 的 `name` 参数指定 `NavHostFragment`。

于此同时使用 `fragment` 标签指定 `name` 参数的效果也是一样的。

所以就把 `NavHostFragment` 看成是一个特殊的 `Fragment` 吧。

- `android:name` 属性包含 `NavHost` 实现的类名称。只要你使用的是 `NavHostFragment`, 就把 `NavHostFragment` 的包路径抄下来吧。`androidx.navigation.fragment.NavHostFragment`
- `app:navGraph` 属性将 `NavHostFragment` 与导航图相关联。导航图会在此 `NavHostFragment` 中指定用户可以导航到的所有目的地。也就是说通过这个将 `NavGraph` 资源引入
- `app:defaultNavHost="true"` 属性确保您的 `NavHostFragment` 会拦截系统返回按钮。请注意, 只能有一个默认 `NavHost`。如果同一布局 (例如, 双窗格布局) 中有多个宿主, 请务必仅指定一个默认 `NavHost`。`true` 表示你的返回操作会被提交到 `NavHostFragment` 中处理, 值得注意的是只能有一个 `NavHost`, 如果一个界面有多个 `NavHostFragment` 务必只选取一个将 `app:defaultNavHost` 设置为 `true` 其余设置为 `false`。

2.目的地XML解析

参考自

什么是目的地?

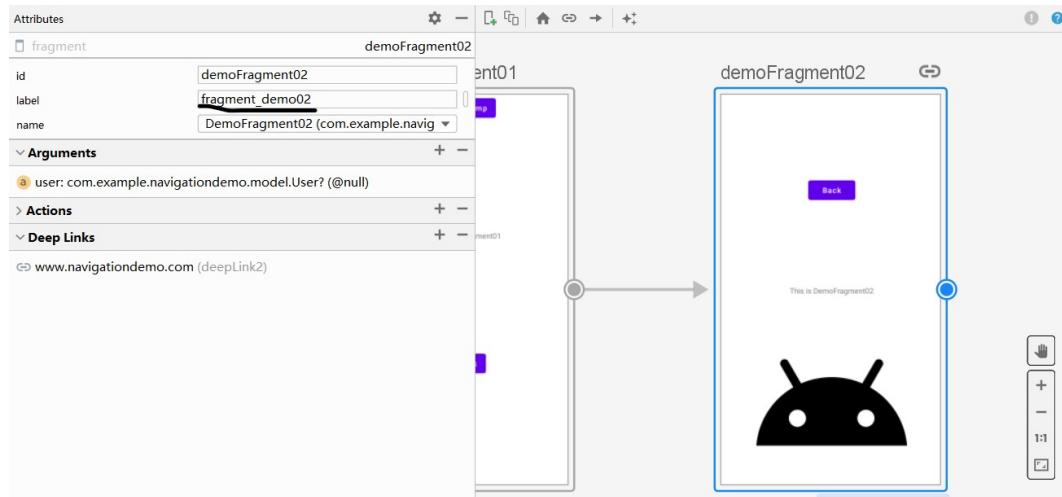
这就是。



Navigation Editor里面的所有页面都是目的地。

XML参数解析

- id 由于目的地与目的地之间会存在跳转关系，需要描述从哪里到哪里，id就是区分不同 destination的参数
- name 也就是当前组件的引用地址，好让NavGraph知道这是什么Fragment或者Activity.....
- label 标签，在和顶部的Toolbar或者Actionbar进行联动的时候，Navigation会使用label的值作为其标题。





3.action标签

- android:id 用于区分不同的action
- app:destination 与名称想表达的意思一样，就是跳转的目的地的id

4. 导航到目的地

参考自

Tips: Navigation还可以导航到Activity，和导航Fragment是类似的，下面实例就不多讲Navigation在Activity之间的跳转，不懂可以看看参考文档。

导航到目的地是使用 [NavController](#) 完成的，它是一个在 [NavHost](#) 中管理应用导航的对象。每个 [NavHost](#) 均有自己的相应 [NavController](#)。您可以使用以下方法之一检索 [NavController](#)：

在Kotlin中可以直接使用 `findNavController()` 获取NavController

Kotlin:

- `Fragment.findNavController()`
- `View.findNavController()`
- `Activity.findNavController(viewId: Int)`

其中 `Activity` 的 `.findNavController` 其实是存在一定问题的。

如果直接传入“`NavHostFragment`”的 id

```
findNavController(R.id.fragmentContainerView)
```

他会报错

```
Activity com.example.navigationdemo.MainActivity@48f9cbc does not have a NavController set on 2131231096
NavController(Navigation.java:61)
NavController(Activity.kt:30)
Activity.onCreate(MainActivity.kt:12)
```

说在Activity中找不到NavController

但是如果传入的id是这个

```
<fragment
    android:id="@+id/m_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>

    findNavController(R.id.m_fragment)
```

很奇怪.....又可以了。

先不纠结了。

所以在Activity中获取NavController最好改成这样

```
val navHostFragment =
    supportFragmentManager.findFragmentById(R.id.fragmentContainerView) as
    NavHostFragment
val navController = navHostFragment.navController
```

之后就是调用 `navigate` 方法。

NavController有13个重载的navigate()方法。在此不做多的解释了。可以自行翻阅。

5. 使用 Safe Args插件进行传递参数

safe args如其名称一样是安全的，它确保了类型的安全性。

Tips：这里也阉割了Activity的传参。

[参考自](#)

[参考自](#)

依赖

```
buildscript {  
    repositories {  
        google()  
    }  
    dependencies {  
        val nav_version = "2.3.5"  
        classpath("androidx.navigation:navigation-safe-args-gradle-  
plugin:$nav_version")  
    }  
}
```

```
plugins {  
    id("androidx.navigation.safeargs")  
}  
  
plugins {  
    id("androidx.navigation.safeargs.kotlin")  
}
```

Safe Args会帮我们生成一些代码，然后确保传递参数时的类型安全。

我们可以试着对比一下

原生Bundle传参

先创建了一个User类

```
package com.example.navigationdemo.model  
  
import java.io.Serializable  
  
/**  
 * @author zhiQiang Tu  
 * @time 2021/7/21 16:53  
 * @signature 我们不明前路，却已在路上  
 */  
data class User(val username:String, val age:Int):Serializable
```

然后在navigate之前new一个User，放进bundle里面，再通过navigate传入。

```

val bundle = Bundle().also {
    it.putSerializable("user", User("tr", 19))
}
findNavController().navigate(R.id.demoFragment02, bundle)

```

再在目的地获取bundle强转

```

val user = arguments?.get("user") as User

```

这里就出问题了。

第一argument需要判空，如果你是用的java，写的还可能存在空指针.....

第二通过argument.get到的是一个Object。也就是说你强转成String编译器在编译阶段也不会报错。

```

package com.example.navigationdemo.ui.fragment

import ...
private const val TAG = "DemoFragment02"
class DemoFragment02 : Fragment() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val user = arguments?.get("user") as String
        Log.e(TAG, user.toString() )
    }

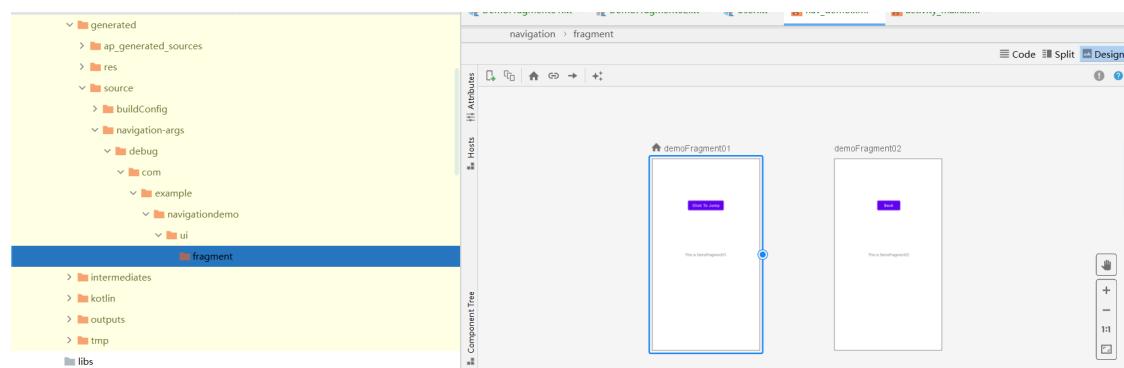
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_demo02, container, attachToRoot: false)
    }
}

```

这样就埋下了隐患。程序一运行就crash了，`java.lang.ClassCastException: com.example.navigationdemo.model.User cannot be cast to java.lang.String`

使用Safe Args传参

使用Safe Args传参的时候需要在NavGraph中建立跳转关系也就是action。否者就会是这样的，什么都没有生成毫无作用。



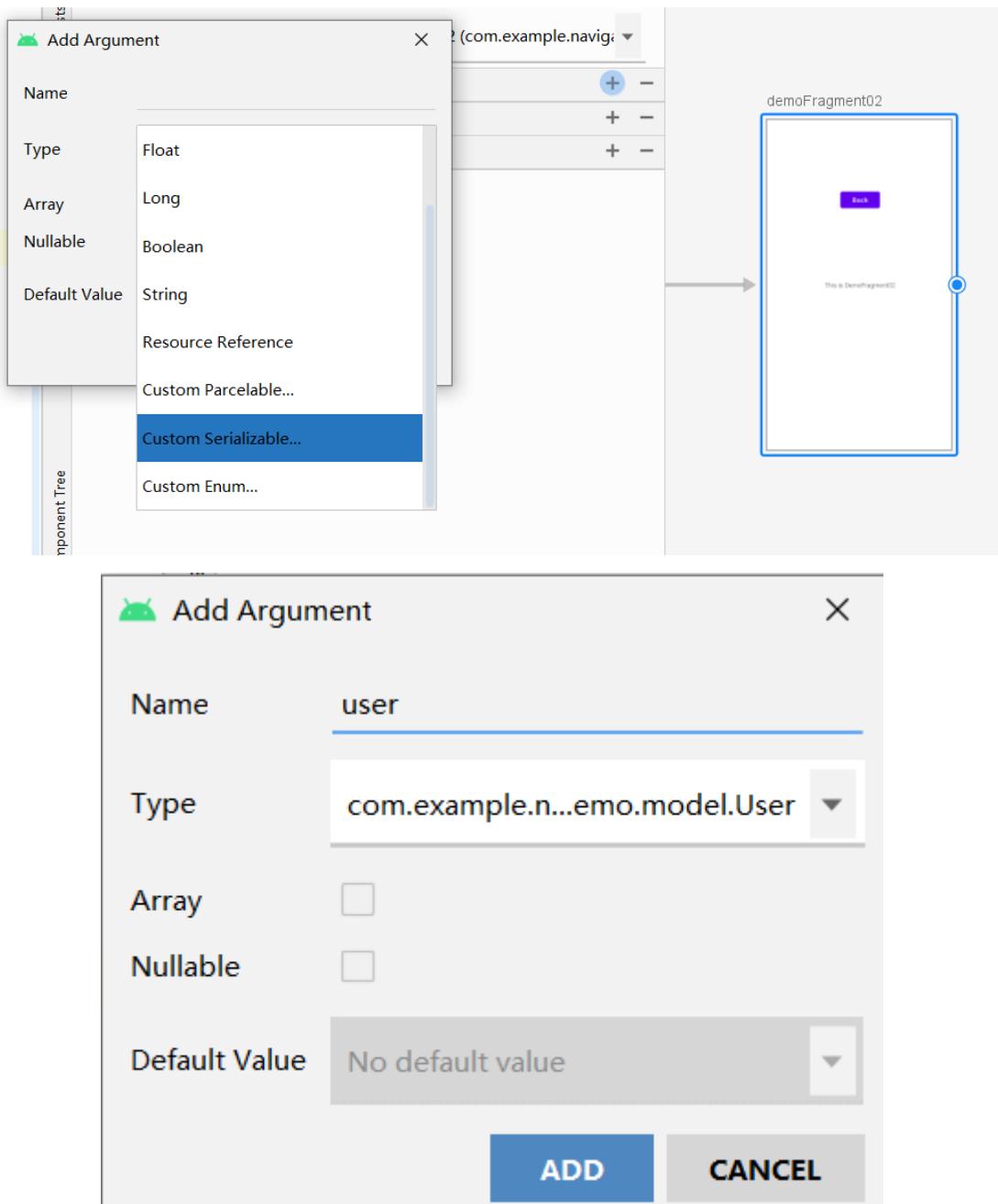
连接以后只生成了这么一个类

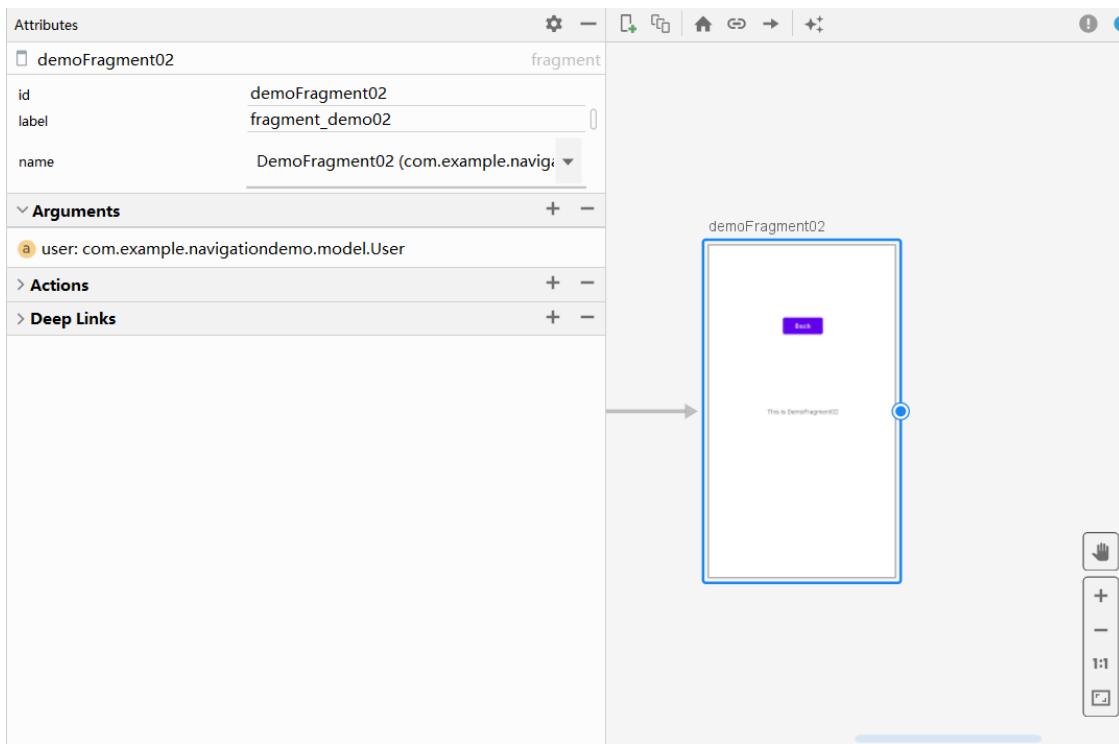
```
package com.example.navigationdemo.ui.fragment

import ...

public class DemoFragment01Directions private constructor() {
    public companion object {
        public fun actionDemoFragment01ToDemoFragment02(): NavDirections =
            ActionOnlyNavDirections(R.id.action_demoFragment01_to_demoFragment02)
    }
}
```

接着我们在跳转的终点加入所需要的参数。





在rebuild一下又生成了一个类

```
package com.example.navigationdemo.ui.fragment

import android.os.Bundle
import android.os.Parcelable
import androidx.navigation.NavArgs
import com.example.navigationdemo.model.User
import java.io.Serializable
import java.lang.IllegalArgumentException
import java.lang.UnsupportedOperationException
import kotlin.Suppress
import kotlin.jvm.JvmStatic

public data class DemoFragment02Args(
    public val user: User
) : NavArgs {
    @Suppress("CAST_NEVER_SUCCEEDS")
    public fun toBundle(): Bundle {
        val result = Bundle()
        if (Parcelable::class.java.isAssignableFrom(User::class.java)) {
            result.putParcelable("user", this.user as Parcelable)
        } else if (Serializable::class.java.isAssignableFrom(User::class.java)) {
            result.putSerializable("user", this.user as Serializable)
        } else {
            throw UnsupportedOperationException(User::class.java.name +
                " must implement Parcelable or Serializable or must be an Enum.")
        }
        return result
    }

    public companion object {
        @JvmStatic
        public fun fromBundle(bundle: Bundle): DemoFragment02Args {
            bundle.setClassLoader(DemoFragment02Args::class.java.classLoader)
            val __user : User?
```

```
if (bundle.containsKey("user")) {
    if (Parcelable::class.java.isAssignableFrom(User::class.java) ||
        Serializable::class.java.isAssignableFrom(User::class.java)) {
        __user = bundle.get("user") as User?
    } else {
        throw UnsupportedOperationException(User::class.java.name +
            " must implement Parcelable or Serializable or must be an Enum.")
    }
    if (__user == null) {
        throw IllegalArgumentException("Argument \"user\" is marked as non-null
but was passed a null value.")
    }
} else {
    throw IllegalArgumentException("Required argument \"user\" is missing and
does not have an android:defaultValue")
}
return DemoFragment02Args(__user)
}
}
}
```

然后在跳转的时候初始化一个NavDirections作为navigate的参数传递过去。完事。

```
val action =
DemoFragment01Directions.actionDemoFragment01ToDemoFragment02(user("tr",
19))
findNavController().navigate(action)
```

使用的时候利用DemoFragment02Args就可以了。

```
arguments?.let {
val user = DemoFragment02Args.fromBundle(it).user
Log.e(TAG, "${user.username} ${user.age}")
}
```

但是好像还是有点复杂

那试试kotlin的委托。这样简单多了吧，而且还确保了类型安全。

```
val args:DemoFragment02Args by navArgs()
val user = args.user
Log.e(TAG, "${user.username} ${user.age}")
```

相较之下显然Safe Args要好不少

6.跳转动画

Tips:

还是阉割了Activity的跳转动画。

Code Place:

[com/example/navigationdemo/ui/fragment/DemoFragment01.kt](https://github.com/tao19910615/com.example.navigationdemo/blob/main/app/src/main/java/com/example/navigationdemo/ui/fragment/DemoFragment01.kt),

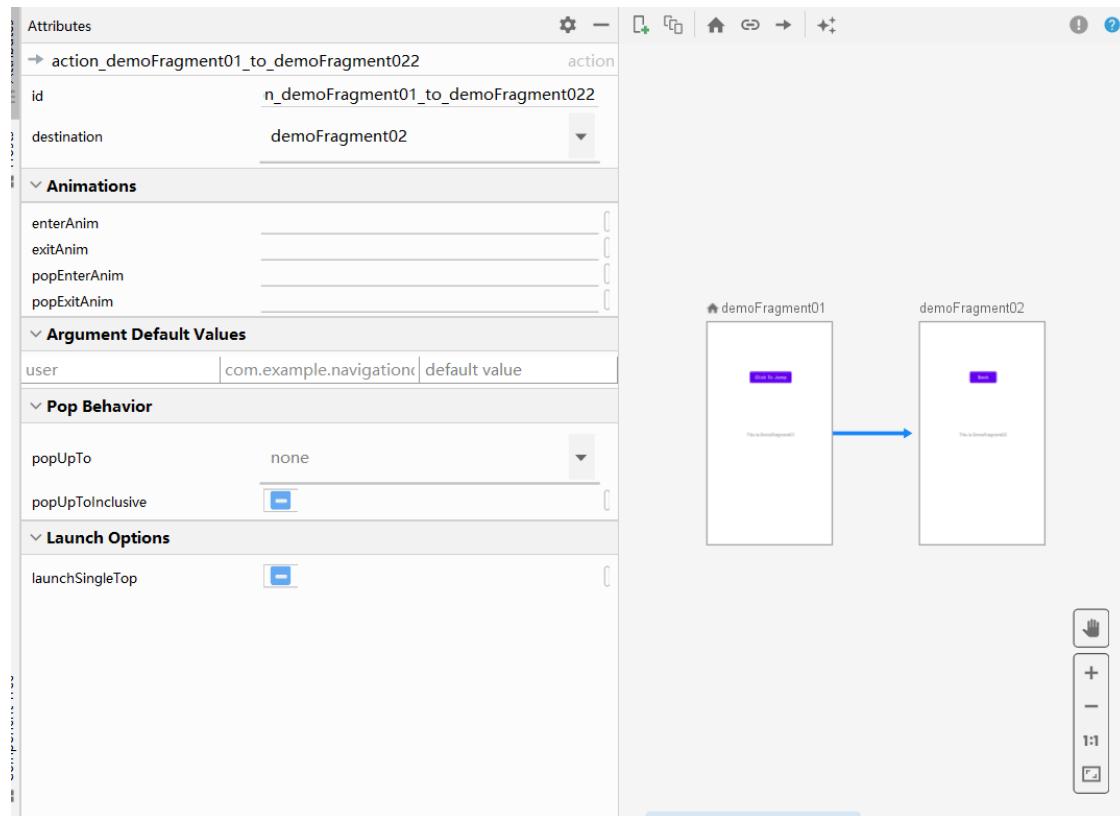
[com/example/navigationdemo/ui/fragment/DemoFragment02.kt](https://github.com/tao19910615/com.example/navigationdemo/blob/main/app/src/main/java/com/example/navigationdemo/ui/fragment/DemoFragment02.kt)

参考自

普通跳转动画

很多时候跳转需要和动画结合，在没有Navigation前，跳转的实现需要加上一些代码，但有了Navigation之后可以在**Navigation Editor**里面直接加入。简单不少。

首先点击Action，动画是添加到Action里面的



然后添加了几个系统自带的动画。

| | |
|--------------|----------------------------|
| enterAnim | @anim/fragment_open_enter |
| exitAnim | @anim/fragment_open_exit |
| popEnterAnim | @anim/fragment_close_enter |
| popExitAnim | @anim/fragment_close_exit |

效果图(GIF)



分析一下XML中参数的意义

- app:enterAnim 入场动画（可以在这里引入入场动画的XML资源）。A到B的一个跳转A退场，B入场，如果对这个action使用enterAnim的话配置的就是B入场的动画。
- app:exitAnim 退场动画。同上
- app:popEnterAnim 也就是弹栈时候的入场动画。之前举了A到B跳转的例子，现在在此基础上在B按下返回键，在Navigation回退栈中B被弹出A出现。这可以理解为B到A，其中A入场，B退场。在这个例子中添加popEnterAnim会对A添加一个动画效果。
- app:popExitAnim 也就是弹栈时候的退场动画。同上。

共享元素动画

这玩意不太好解释，还是上图

效果图(GIF)



共享元素动画也就是，跳转的起点和终点存在共同的组件，上图中的共享元素是机器人头像。

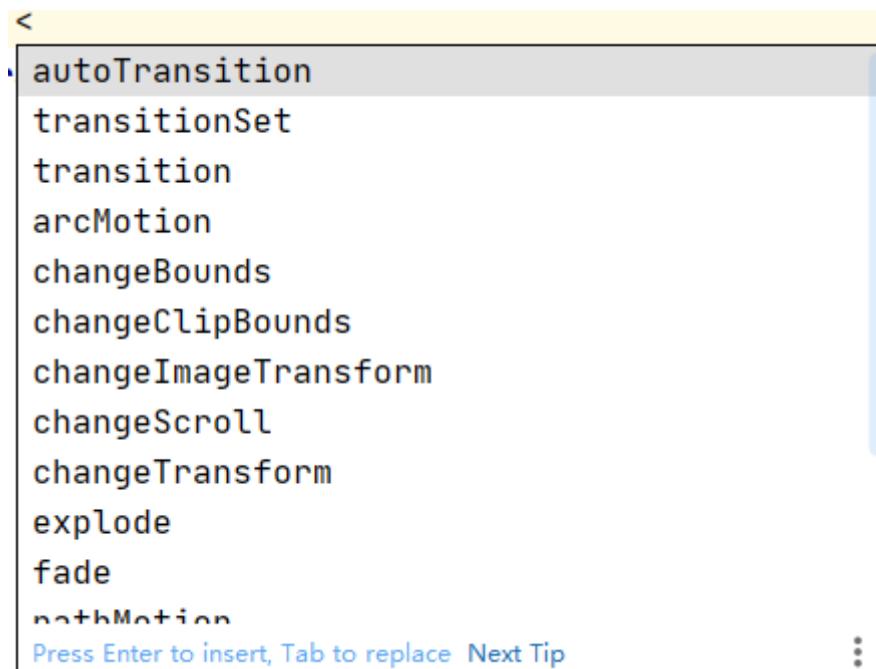
如何加入共享元素动画？

首先创建一个Transition资源

```
<?xml version="1.0" encoding="utf-8"?>
<transitionSet>
    <autoTransition/>
</transitionSet>
```

这个比较通用就用这个了，还有其他的[Transition资源](#)

这里就不做多的阐述。有兴趣的可以下去自行看[文档](#)。



然后在跳转的目的地的onCreate方法中设置sharedElementEnterTransition

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    sharedElementEnterTransition = TransitionInflater.from(requireContext())
        .inflateTransition(R.transition.demo_transition)
}
```

最后在需要跳转的地方创建Navigator.Extras并传入到navigate中去。

```
Navigator.Extrasdemo01_jump_button.setOnClickListener {

    //创建Navigator.Extras
    val imageTransaction = Pair<View, String>(imageView, "demoImage")
    val extras = FragmentNavigatorExtras(imageTransaction)
    val action =
        DemoFragment01Directions.actionDemoFragment01ToDemoFragment022(user("tr",
19))
    findNavController().navigate(action, extras)
}
```

最最后。

不建议将NavGraph的action动画和共享元素动画一起使用（不是我说的，Google说的）。

★ 注意：使用共享元素过渡时，不得使用动画框架（上一部分中的 `enterAnim`、`exitAnim` 等），而只能使用过渡框架来设置进入和退出过渡。

也即是这两个择一即可。

Animations

`enterAnim`

`exitAnim`

`popEnterAnim`

`popExitAnim`

```
sharedElementEnterTransition = TransitionInflater.from(requireContext())
    .inflateTransition(R.transition.demo_transition)
```

7. DeepLink

参考自

在 Android 中，深层链接是指将用户直接转到应用内特定目的地的链接。

借助 Navigation 组件，您可以创建两种不同类型的深层链接：显式深层链接和隐式深层链接。

Tip:

Code Place :

[com/example/navigationdemo/ui/fragment/DemoFragment01.kt](https://github.com/taoyuan1993/com.example.navigationdemo/blob/main/app/src/main/java/com/example/navigationdemo/ui/fragment/DemoFragment01.kt)

- 显式的深层链接

显式深层链接是深层链接的一个实例，该实例使用 `PendingIntent` 将用户转到应用内的特定位置。例如，您可以在通知或应用微件中显示显式深层链接。

比如在Fragment中发送一条Notification，Notifaction承载一个由Navigation创建的 PendingIntent

```
deep_link_button.setOnClickListener {
    val manager =
        NotificationManagerCompat.from(requireContext())
    manager.notify(notificationId++, createNotification())

}

//创建Notification
private fun createNotification(): Notification {
    val notificationName = requireActivity().packageName
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val channel = NotificationChannel(
            notificationName, "DeepLinkChanner",
            NotificationManager.IMPORTANCE_DEFAULT
        )

        val notificationManager =
```

```
    requireActivity().getSystemService(NotificationManager::class.java)
        notificationManager.createNotificationChannel(channel)

    }
    return NotificationCompat.Builder(requireContext(),
notificationName)
    .setSmallIcon(R.drawable.ic_launcher_foreground)
    .setContentTitle("这是DeepLink Demo")
    .setContentText("DeepLink")
    .setContentIntent(getPendingIntent())
    .setAutoCancel(true)
    .build()
}
```

```
//创建一个PendingIntent
private fun getPendingIntent(): PendingIntent {
    return NavDeepLinkBuilder(requireActivity())
        .setGraph(R.navigation.nav_demo)
//.setComponentName感觉很鸡肋，又没太懂他是干啥的。
//.setComponentName(DeepLinkActivity::class.java)
        .setDestination(R.id.deepLinkActivity)
        .createPendingIntent()
}
```

Activity中进行DeepLink与此相似

代码在com/example/navigationdemo/MainActivity.kt中不做过多解释。

总的来说DeepLink并没甚过人之处，**它只是提供了PendingIntent**。实际DeepLink的使用也就只有这点代码

```
NavDeepLinkBuilder(requireActivity())
    .setGraph(R.navigation.nav_demo)
    .setDestination(R.id.deepLinkActivity)
    .createPendingIntent()
```

对了deepLink还可以这样创建PendingIntent，跟上面的代码时等价的(通过NavController)

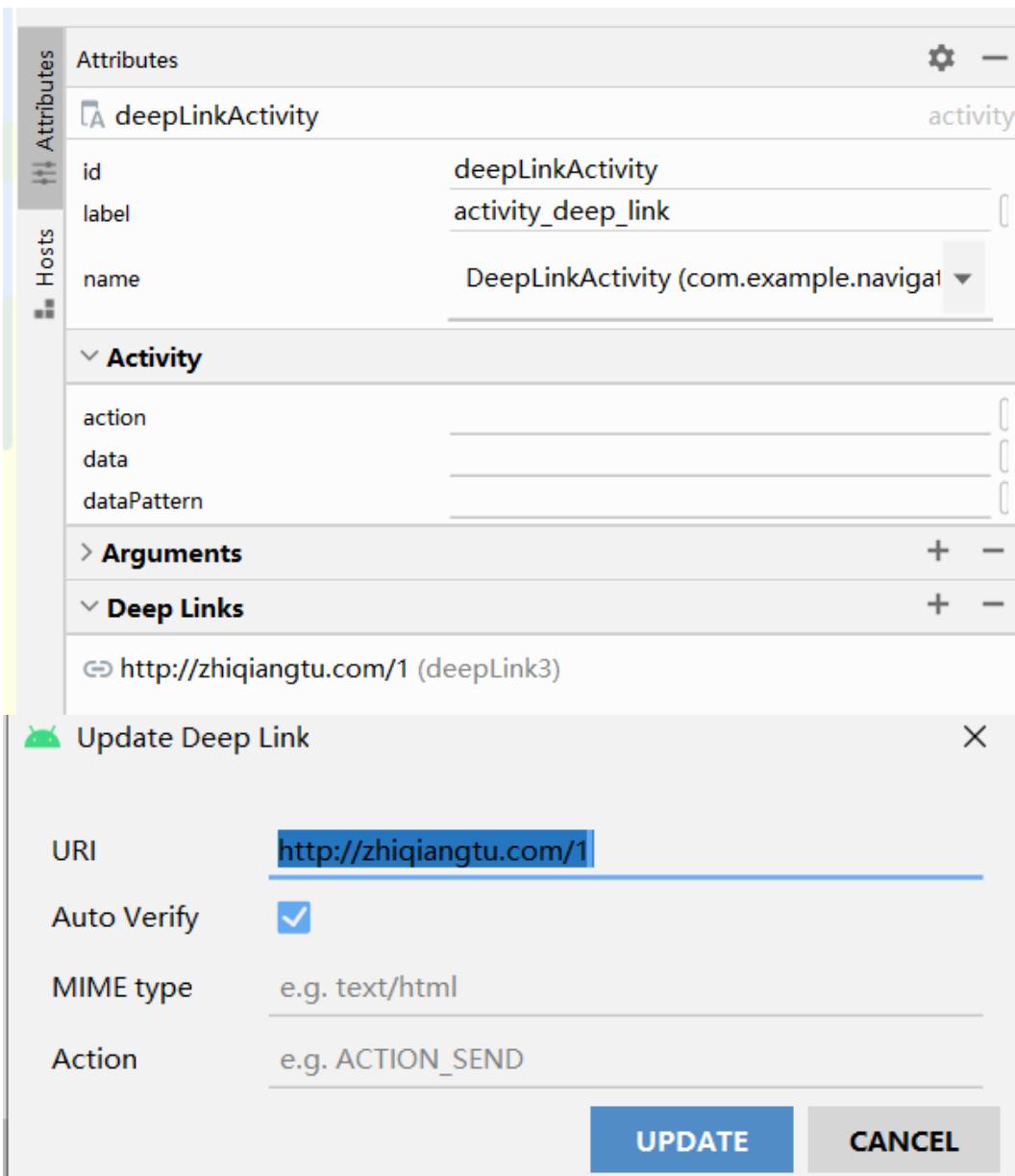
```
findNavController()
    .createDeepLink()
    .setDestination(R.id.deepLinkActivity)
    .createPendingIntent()
```

- 隐式的深层链接

我们在学习startActivity的时候学了显式启动和隐式启动，DeepLink也即是类似的。

我们只需要配置两步即可完成。

在对应的链接位置配置好deeplink



然后就是在对应的activity的manifest文件中加一个nav-graph标签（不是必须要配置到MainActivity中只是我的NavGraph在MainActivity中所以这样配置）

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <nav-graph android:value="@navigation/nav_demo" />
</activity>
```

然后就配置完成了。

就可以通过这个URI指向这个app了。

注意不能直接在浏览器中搜索这个URI这样是无法跳转的。

在浏览器中会是这样的



This site can't be reached

Make sure the web address you entered is correct,
or try again later.

NAME_NOT_RESOLVED -105 SEARCH

Address not found. Showing search results for "zhi
qiangtu"

综合 视频 资讯 小视频 图片 音乐 月

任志强_财经百科_网易财经



任志强,地产商,北京市政协委员,北
京市华远地产股份有限公司董事长,
同时兼任北京市商业银行监事、...

zhiqiangtu_jiran1147_新浪博客

her voiceSome joke. stuffed tight into the pockets



为了测试这个跳转我网上抄了一段HTML代码

```
<!DOCTYPE html>
<!DOCTYPE html>
<html>

<head>
    <title>跳转测试</title>
    <meta http-equiv="content-type" content="text/html">
</head>

<body>
<a href="http://zhiqiangtu.com/1">点击跳转到app</a>
</body>

</html>
```

传入到手机里面，通过浏览器打开。

China Unicom 4G 4G 5G 5G 140 K/S 10:46
China Mobile 4G

← 跳转测试

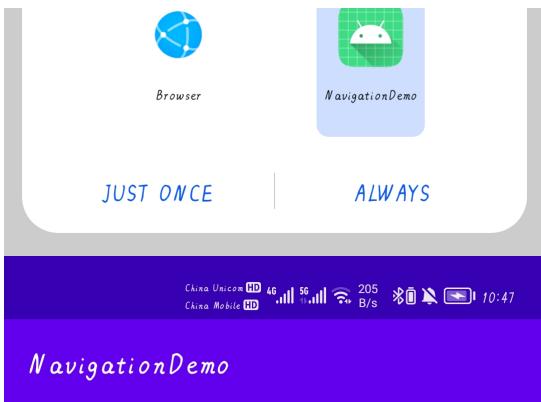
[点击跳转](#)

China Unicom 4G 4G 5G 5G 64.2 K/S 10:47
China Mobile 4G

← 跳转测试

[点击跳转](#)

Open with



This is DeepLink Activity

DeepLink讲的有些次，有兴趣可以在文档上查查。

8.Navigation UI

Navigation 组件包含 `NavigationUI` 类。此类包含多种静态方法，可帮助您使用**顶部应用栏、抽屉式导航栏和底部导航栏**来管理导航。

顶部导航栏

参考自

顶部应用栏在应用顶部提供了一个固定位置，用于显示当前屏幕的信息和操作。

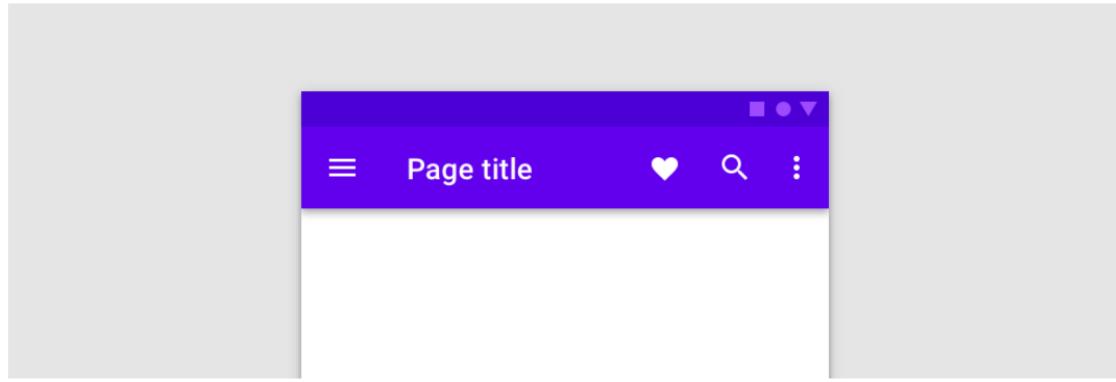


图 1. 显示顶部应用栏的屏幕。

利用 `NavigationUI` 包含的方法，您可以在用户浏览应用的过程中自动更新顶部应用栏中的内容。例如，`NavigationUI` 可使用导航图中的目的地标签及时更新顶部应用栏的标题。

```
<navigation>
    <fragment ...>
        android:label="Page title"
    ...
</fragment>
</navigation>
```

总结一句话就是顶部导航栏上的`TextView`的`text`属性是`NavGraph`中`fragment`的`label`标签的值。

`NavigationUI` 支持以下顶部应用栏类型：

- `Toolbar`
- `CollapsingToolbarLayout`
- `ActionBar`

Tip:

Code Place: com/example/navigationdemo/MainActivity.kt

Navigation如何管理顶部应用栏呢？通过 `AppBarConfiguration`

导航按钮的行为会根据用户是否位于顶层目的地而变化。

顶层目的地是一组存在层次关系的目的地中的根级或**最高级目的地**。顶层目的地不会在顶部应用栏中显示“向上”按钮，因为不存在更高等级的目的地。默认情况下，应用的起始目的地是唯一的顶层目的地。

当用户位于**顶层目的地**时，如果目的地使用了 `DrawerLayout`，导航按钮会变为抽屉式导航栏图标 。如果目的地没有使用 `DrawerLayout`，导航按钮处于隐藏状态。当用户位于**任何其他目的地**时，导航按钮会显示为向上按钮 。在配置导航按钮时，如需将起始目的地用作唯一顶层目的地，请创建 `AppBarConfiguration` 对象并传入相应的导航图，如下所示

```
val appBarConfiguration = AppBarConfiguration(navController.graph)
```

在某些情况下，您可能需要定义多个顶层目的地，而不是使用默认的起始目的地。这种情况的一种常见用例是 `BottomNavigationView`，在此场景中，同级屏幕可能彼此之间并不存在层次关系，并且可能各自有一组相关的目的地。对于这样的情况，您可以改为将一组目的地 ID 传递给构造函数，如下所示：

```
val appBarConfiguration = AppBarConfiguration(setOf(R.id.main, R.id.profile))
```

开始敲代码了

theme改成NoActionBar XML中写入Toolbar

```
val appBarConfiguration = AppBarConfiguration(navController.graph)
toolbar.setupWithNavController(navController, appBarConfiguration)
```

Navigation Menu

参考自

`NavigationUI` 提供了对 `Menu` 驱动跳转的支持。`NavigationUI` 包含一个方法 `onNavDestinationSelected()`。它将 `MenuItem` 和 `Destination` 进行关联。如果 `Menu` 的 `Id` 与 `Destination` 的 `Id` 是一致的那么 `NavController` 就会直接帮我们导航到 `Destination` 去。

上代码

Code Place:

[com/example/navigationdemo/MMainActivity.kt](#)

```
//创建Menu
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.main_menu,menu)
    return true
}

//利用Navigation对MenuItem进行导航
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return item.onNavDestinationSelected(navController) ||
super.onOptionsItemSelected(item)
}
//如果你是用的Toolbar记得调用 setSupportActionBar(toolbar) 否者，Menu是显示不出来的。
```

Navigation Drawer

参考自

抽屉式导航栏是显示应用主导航菜单的界面面板。当用户触摸应用栏中的抽屉式导航栏图标 或用户从屏幕的左边缘滑动手指时，就会显示抽屉式导航栏。

- 抽屉式导航栏图标会显示在使用 `DrawerLayout` 的所有 [顶层目的地](#) 上。
- 如需添加抽屉式导航栏，请先声明 `DrawerLayout` 为 [根视图](#)。在 `DrawerLayout` 内，为主界面内容以及包含抽屉式导航栏内容的其他视图添加布局。

例如，以下布局使用含有两个子视图的 `DrawerLayout`：包含主内容的 `NavHostFragment` 和适用于抽屉式导航栏内容的 `NavigationView`

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Use DrawerLayout as root container for activity -->
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

    <!-- Layout to contain contents of main body of screen (drawer will
    slide over this) -->
    <androidx.fragment.app.FragmentContainerView
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:id="@+id/nav_host_fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />

    <!-- Container for contents of drawer - use NavigationView to make
    configuration easier -->
    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true" />

</androidx.drawerlayout.widget.DrawerLayout>
```

也即是说如果使用`DrawerLayout`抽屉视图，那么根节点必须是`DrawerLayout`，`DrawerLayout`内包含两个View一个是主界面，一个是侧滑菜单。由于需要将侧滑菜单和`NavigationView`关联，所以就使用了`NavigationView`。

但是这样的`NavigationView`貌似还是一个空白什么也没有，如何将抽屉菜单的内容加入到`NavigationView`中呢？

```
<com.google.android.material.navigation.NavigationView
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:id="@+id/navigation_view"
    android:layout_gravity="start"
    />
```

其实很简单，引入`app:menu`的标签，将menu资源引入即可。

注意menu的id和destination的id必须一致否者无法跳转。

先创建一个menu资源

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/drawerFragment02"
        android:title="drawerFragment02"
        android:icon="@drawable/ic_baseline_looks_5_24"
        android:orderInCategory="2"/>
    <item
        android:id="@+id/drawerFragment01"
        android:title="drawerFragment01"
        android:icon="@drawable/ic_baseline_looks_4_24"
        android:orderInCategory="1"/>
    <item
        android:id="@+id/drawerFragment03"
        android:title="drawerFragment03"
        android:icon="@drawable/ic_baseline_looks_6_24"
        android:orderInCategory="3"/>
</menu>
```

然后再创建几个Fragment加入到NavGraph中

最后就是一点点配置代码

```
navigation_view.setupWithNavController(navController)
```

最最后还有将drawerLayout传入AppBarConfiguration的构造函数中 (不然左上角Toolbar是没有这个图标)

```
inline fun AppBarConfiguration(
    navGraph: NavGraph,
    drawerLayout: Openable? = null,
    noinline fallbackOnNavigateUpListener: () -> Boolean = { false }
)
```

BottomNavigation

参考自

`NavigationUI` 也可以处理底部导航。当用户选择某个菜单项时，`NavController` 会调用 `onNavDestinationSelected()` 并自动更新底部导航栏中的所选项目。

这个嘛还是那么简单。

先创建一个Menu

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

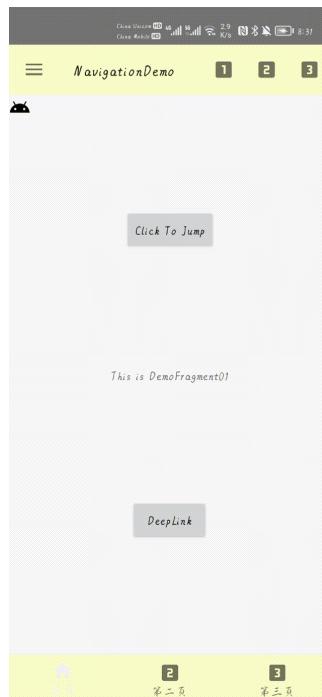
    <item android:title="首页"
        android:id="@+id/demoFragment01"
        android:orderInCategory="1"
```

```
        android:icon="@drawable/ic_baseline_home_24"/>
<item android:title="第二页"
      android:id="@+id/secondPageFragment"
      android:orderInCategory="2"
      android:icon="@drawable/ic_baseline_looks_two_24"/>
<item android:title="第三页"
      android:id="@+id/thirdPageFragment"
      android:orderInCategory="3"
      android:icon="@drawable/ic_baseline_looks_3_24"/>
</menu>
```

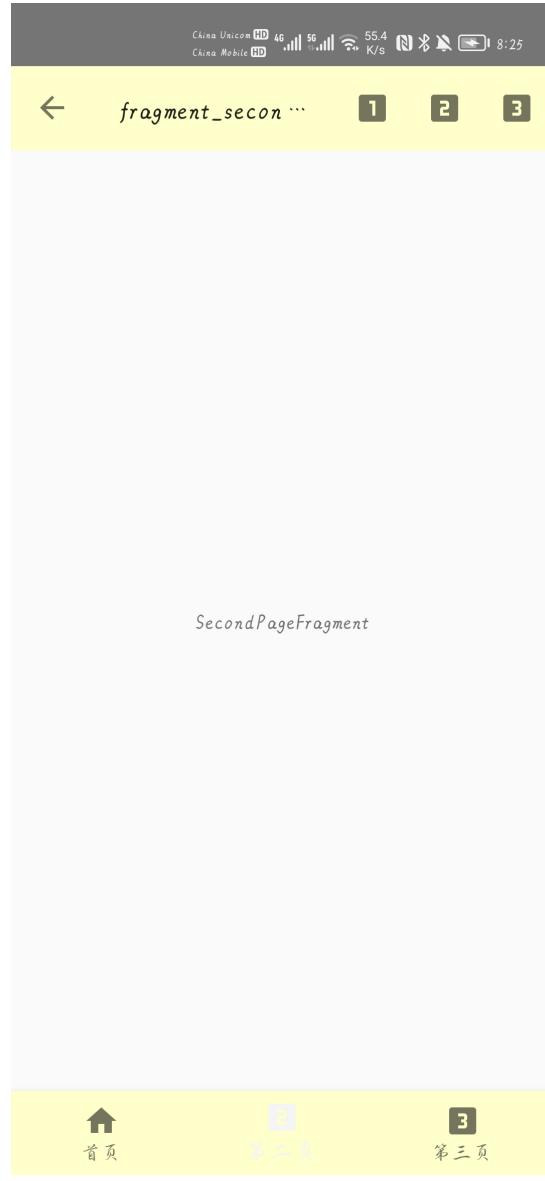
然后添加一点配置

```
bottomNavigationView.setupWithNavController(navController)
```

效果图(GIF)

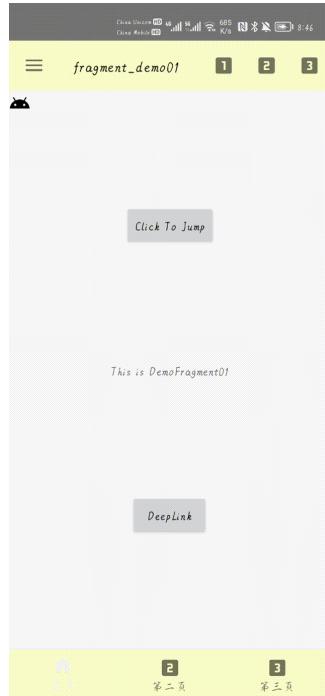


是不是有点奇怪。按道理“第二页”，“第三页”的Fragment是和“首页”一样的**顶层视图**。也就是说它是不可以返回的，然鹅Toolbar确显示了回退按钮，这很奇怪。所以得将“第二页”，“第三页”加入到AppbarConfiguration的顶层视图的集合中。也就稍微改改AppbarConfiguration。



```
//之前
AppBarConfiguration(navGraph = navController.graph
    ,drawerLayout = drawer_layout
    ,fallbackOnNavigateUpListener = ::onSupportNavigateUp)
//修改后
AppBarConfiguration(setOf(R.id.demoFragment01,R.id.secondPageFragment,R.id.thirdPageFragment)
    ,drawer_layout
    ,::onSupportNavigateUp)
```

最终效果图



虽然已近写了这么多了，但是还有很多都没讲。比如Navigation的DSL，Navigation返回栈，Navigation进行模块间的导航.....

Room

1.什么是Room

Room 在 **SQLite** 上提供了一个抽象层，以便在充分利用 **SQLite** 的强大功能的同时，能够流畅地访问数据库。

2.Room一览

Room 包含 3 个主要组件：

- **DataBase**：包含数据库持有者，并作为应用已保留的持久关系型数据的底层连接的主要接入点。

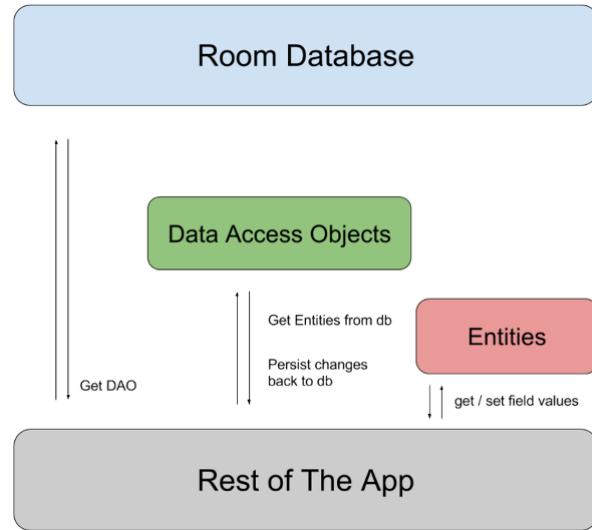
使用 `@Database` 注释的类应满足以下条件：

- 是扩展 `RoomDatabase` 的抽象类。
- 在注释中添加与数据库关联的**实体列表**。
- 包含具有 0 个参数且返回使用 `@Dao` 注释的类的抽象方法。

在运行时，您可以通过调用 `Room.databaseBuilder()` 或 `Room.inMemoryDatabaseBuilder()` 获取 `Database` 的实例。

- **Entity**：表示数据库中的表。
- **DAO**：包含用于访问数据库的方法。

关于与数据库的访问。



- 应用使用 Room 数据库来获取与该数据库关联的数据访问对象 (DAO)。
- 然后，应用使用每个 DAO 从数据库中获取实体，然后再将对这些实体的所有更改保存回数据库中。
- 最后，应用使用实体来获取和设置与数据库中的表列相对应的值。

也就是说Room提供了DAO (Data Access Objects)作为App和DataBase的中间人。

3.Room的基本使用

参考自

Code Place:

com/example/roomdemo/db,
com/example/roomdemo/MainActivity.kt

- 添加依赖

```

dependencies {
    def room_version = "2.3.0"
    //运行时的依赖
    implementation("androidx.room:room-runtime:$room_version")
    //注解处理器
    annotationProcessor "androidx.room:room-compiler:$room_version"
    // To use Kotlin annotation processing tool (kapt)
    //这个也是注解处理器只不过时kotlin-kapt
    kapt("androidx.room:room-compiler:$room_version")
    // To use Kotlin Symbolic Processing (KSP) 类似于kapt速度快一些，不过暂时处于测试版
    ksp("androidx.room:room-compiler:$room_version")

    // optional - Kotlin Extensions and Coroutines support for Room
    implementation("androidx.room:room-ktx:$room_version")

    // optional - RxJava2 support for Room
    implementation "androidx.room:room-rxjava2:$room_version"

    // optional - RxJava3 support for Room
    implementation "androidx.room:room-rxjava3:$room_version"
}

```

```
// optional - Guava support for Room, including Optional and
ListenableFuture
implementation "androidx.room:room-guava:$room_version"

// optional - Test helpers
testImplementation("androidx.room:room-testing:$room_version")
}
```

这里我就使用了两个必要的一个是在运行时的一个是注解处理器。

```
//如果使用kapt一定要加上，kotlin的注解处理插件都没kapt个锤锤。
id 'kotlin-kapt'

//room
def room_version = "2.3.0"
implementation("androidx.room:room-runtime:$room_version")
// To use Kotlin annotation processing tool (kapt)
kapt("androidx.room:room-compiler:$room_version")
```

- 创建实体类Entity

```
@Entity(tableName = "user_table")
data class User(
    @PrimaryKey(autoGenerate = true) val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

其中PrimaryKey是主键，我们可以通过这个主键直接从数据库中查询到该数据。一个表单中必须要有主键（没有为什么）。

其实一个Entity表单其实就相当于一个Excel表格，比如上面的user_table就可以这样写

| | A | B | C | D |
|---|------------|------------|-----------|---|
| 1 | user_table | | | |
| 2 | uid | first_name | last_name | |
| 3 | 1 | a | a | |
| 4 | 2 | b | b | |
| 5 | 3 | c | c | |

所以uid , firstName, lastName每个变量都占据一列。ColumnInfo就是设置每列的属性。

- 创建Dao

```

@Dao
interface UserDao {
    @Query("SELECT * FROM user_table")
    fun getAll(): List<User>

    @Query("SELECT * FROM user_table WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user_table WHERE first_name LIKE :first AND
    last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}

```

这个大家因该都能懂就不解释了。

- 创建数据库DataBase

```

@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    //单例。
    companion object{
        var instance:AppDatabase? = null
        @Synchronized

```

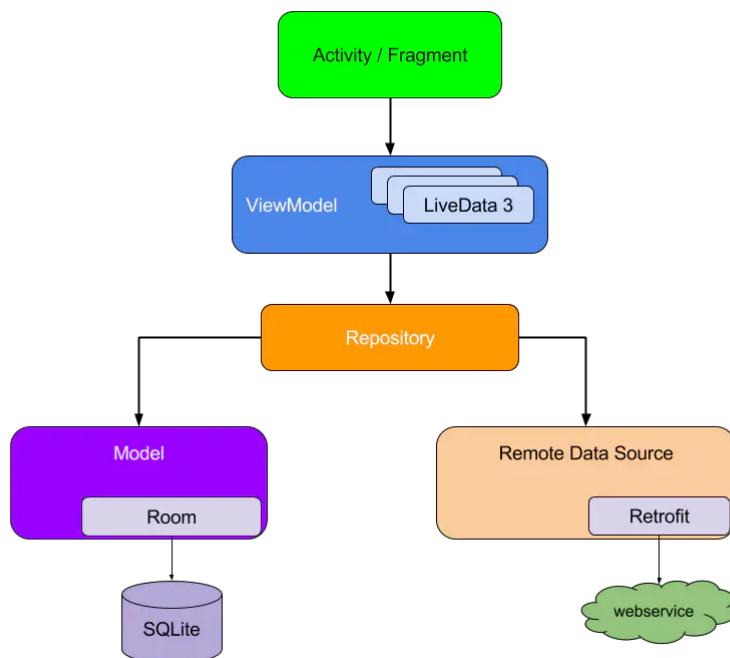
```

        fun getInstance(applicationContext: Context): AppDatabase {
            instance?.let {
                return it
            }
            return
        Room.databaseBuilder(applicationContext, AppDatabase::class.java,
                APP_DATABASE_NAME).build().apply {
                instance = this
            }
        }
    }
}

```

有一点需要注意这个Database是抽象类。除此之外我们在创建过程中有两中选择，一种是持久化的数据库，一种是缓存数据库。

然后就是在activity中使用（这代码写的有亿点烂。想必大家能懂意思。实际开发得用Google官方推荐的标准架构



```

package com.example.roomdemo

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import androidx.lifecycle.lifecycleScope
import com.example.roomdemo.db.AppDatabase
import com.example.roomdemo.db.dao.UserDao
import com.example.roomdemo.model.User
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlin.math.log
private const val TAG = "MainActivity"

```

```
class MainActivity : AppCompatActivity() {
    lateinit var userDao:UserDao
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        userDao = AppDatabase.getInstance(applicationContext).userDao()
        setContentView(R.layout.activity_main)
        setListeners()
    }

    private fun setListeners() {
        button_get_all.setOnClickListener {
            lifecycleScope.launch(Dispatchers.IO) {
                val all = userDao.getAll()
                all.forEach{
                    Log.e(TAG, "$it" )
                }
            }
        }

        button_find_by_name.setOnClickListener {
            lifecycleScope.launch(Dispatchers.IO) {
                val result = userDao.findByName("a","a")
                Log.e(TAG, "$result" )
            }
        }

        button_load_all_by_ids.setOnClickListener {
            lifecycleScope.launch (Dispatchers.IO){
                val result = userDao.loadAllByIds(intArrayOf(1,2,3,4))
                result.forEach{
                    Log.e(TAG, "$it" )
                }
            }
        }

        button_delete.setOnClickListener {
            lifecycleScope.launch(Dispatchers.IO) {
                userDao.delete(User("a","a"))
            }
        }

        button_insert_all.setOnClickListener {
            lifecycleScope.launch(Dispatchers.IO){
                userDao.insertAll(User("a", "a"),
                    User("b", "b"),
                    User("c", "c"),
                    User("d", "d"))
            }
        }
    }
}
```

注解详解

对于基础的@DAO, @Database, @Entity我们已近有所了解。但是其实还存在一些比较常用的。（注意：Room的注解其实不算少，但是有很多的注解不是很常规，所以就暂时没必要花时间去学，以下只会对常用的注解进行较为详细的描述，不常用的就一笔带过。）

@Entity

```
@Entity  
data class User(  
    @PrimaryKey var id: Int,  
    var firstName: String?,  
    var lastName: String?  
)
```

这个大家都熟吧。

其中有一点需要注意，如果你想将某个变量添加到数据库的表单中一定要满足以下两个条件中的一种

- 1.要么该数据为一个public变量
- 2.如果该数据是一个private变量，你得提供get, set方法。

- 主键PrimaryKey的使用

每个实体类至少要有一个主键,主键可以通过对变量使用@PrimaryKey，于此同时还可以在中进行申明 @Entity(primaryKeys = arrayOf()) (二选一即可)

比如这样

```
@Entity(primaryKeys = arrayOf("firstName", "lastName"))  
data class User(  
    val firstName: String?,  
    val lastName: String?  
)
```

有的时候我们可能懒得自己去生成主键，但是我们可以让Room自动帮我们生成，使用@PrimaryKey(autoGenerate=true)即可。

- 指定表单名称

Entity是一个表单，我们可以自己定义表单的名称。只需要这样就行了（默认表单名称和实体类名是一致的）

```
@Entity(tableName = "users")  
data class User (  
    // ...  
)
```

- 指定变量名称

这里需要引入一个新的注解@ColumnInfo，这个注解是作用在实体类的成员属性上的，可以指定成员属性的一些信息。关于指定变量名称，与前面的指定表名称类似

```
@Entity(tableName = "users")
data class User (
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

- 忽略变量

默认情况下，Room 会为实体中定义的每个字段创建一个列。如果某个实体中有您不想保留的字段，则可以使用 `@Ignore` 为这些字段添加注释，如以下代码段所示：

```
@Entity
data class User(
    @PrimaryKey val id: Int,
    val firstName: String?,
    val lastName: String?,
    @Ignore val picture: Bitmap?
)
```

如果实体继承了父实体的字段，则使用`@Entity`属性的 `ignoredColumns` 属性通常会更容易：

```
open class User {
    var picture: Bitmap? = null
}

@Entity(ignoredColumns = arrayOf("picture"))
data class RemoteUser(
    @PrimaryKey val id: Int,
    val hasVpn: Boolean
) : User()
```

- 支持全文搜索

如果您的应用需要通过全文搜索 (FTS) 快速访问数据库信息，请使用虚拟表（使用 FTS3 或 FTS4 为您的实体提供支持）。

如果在2.1.0以及更高版本中Room提供了 `@Fts3` 或 `@Fts4`注解，按理用处不大。这玩意因该是用于快速查找的，一般情况下手机上的数据应该不是很多的，所以用处不大。就跳过了。

需要了解的兄的[看这里](#)

- 提供对Java AutoValue的支持

对Java的支持跟我Kotlin有啥关系。ha

稍微说一下，AutoValue是用于Java的实体类创建的，有的时候我们在创建对象的时候在构造函数会传入null值，这会为空指针埋下隐患，所以在初始化bean的成员变量的时候保险的方案是这样的。

```

AutoValue_User(String name, int age, String address) {
    if (name == null) {
        throw new NullPointerException("Null name");
    } else {
        this.name = name;
        this.age = age;
        if (address == null) {
            throw new NullPointerException("Null address");
        } else {
            this.address = address;
        }
    }
}

```

但是你有没有发现代码有点长，而且都是一些模板化的判空，所以Google的几个工程师就写了一个小的java实体类生成工具。也就是AutoValue。在kt中无疑data class是更好的解决方案。

附上官方的代码

```

@AutoValue
@Entity
public abstract class User {
    // Supported annotations must include `@CopyAnnotations`.
    @CopyAnnotations
    @PrimaryKey
    public abstract Long getId();

    public abstract String getFirstName();
    public abstract String getLastName();

    // Room uses this factory method to create User objects.
    public static User create(Long id, String firstName, String
lastName) {
        return new AutoValue_User(id, firstName, lastName);
    }
}

```

其实还有很多东西都没讲，有时间可以自己下来研究。没时间就算了，这些也差不多够用了。/狗头

@DAO

...
data access objects, or DAOs. Each DAO includes methods that offer abstract access to your app's database. At compile time, Room automatically generates implementations of the DAOs that you define.

需要注意的是DAO是抽象的东西，它可以用接口写，其实用抽象类写也是可以的，他的实现是Room通过注解处理器自动生成的。（只不过通常都是用的接口写，可能代码稍微少一点，方法可以不写abstract）

```

@Dao
abstract class UserDao {
    @Query("SELECT * FROM user_table")
}

```

```

abstract fun getAll(): List<User>

@Query("SELECT * FROM user_table WHERE uid IN (:userIds)")
abstract fun loadAllByIds(userIds: IntArray): List<User>

@Query("SELECT * FROM user_table WHERE first_name LIKE :first AND last_name
LIKE :last LIMIT 1")
abstract fun findByName(first: String, last: String): User

@Insert
abstract fun insertAll(vararg users: User)

@Delete
abstract fun delete(user: User)
}

```

下面的是Room为我们生成的DAO的实现类

可以在: build/generated/source/kapt/debug/....下查找代码



```

package com.example.roomdemo.db.dao;

import ...

@unchecked, deprecation/
public final class UserDao_Impl implements UserDao {
    private final RoomDatabase __db;

    private final EntityInsertionAdapter<User> __insertionAdapterOfUser;
    private final EntityDeletionOrUpdateAdapter<User> __deletionAdapterOfUser;

    public UserDao_Impl(RoomDatabase __db) {
        this.__db = __db;
        this.__insertionAdapterOfUser = new EntityInsertionAdapter<User>(__db) {
            @Override
            public String createQuery() {
                return "INSERT OR ABORT INTO `user_table` (`first_name`, `last_name`, `uid`) VALUES (?, ?, nullif(?, 0))";
            }

            @Override
            public void bind(SupportSQLiteStatement stmt, User value) {
                if (value.getFirstName() == null) {
                    stmt.bindNull(index: 1);
                } else {
                    stmt.bindString(index: 1, value.getFirstName());
                }
                if (value.getLastName() == null) {
                    stmt.bindNull(index: 2);
                } else {
                    ...
                }
            }
        };
    }
}

```

我们知道数据库的操作分为4种：

增，删，改，查

分别对应DAO的注解**@Insert, @Delete, @Update, @Query.**

其中增，删，改封装的比较好，我们可以不需要写任何的sql语句。

但是**查就不一样了**，因为**查询的方法是多样的**，你可以给出一个范围查询，也可能只是一个确切的值进行查询，这个无法很好的封装。没办法，只好和sql打交道了。你也能从下面的代码中发现。（于此同时**@Query**注解的能力是非常强的它能完成查询，但是其他的增，删，改其实也能。但这需要sql的基础了。）

- DAO注解的使用

@Insert标注的方法内传入的参数必须是@Entity标注的实体类。 (除此之外别忘了还得在Database中声明)

- @Insert

```
@Dao
interface MyDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUsers(vararg users: User)

    @Insert
    fun insertBothUsers(user1: User, user2: User)

    @Insert
    fun insertUsersAndFriends(user: User, friends: List<User>)
}
```

上述方法了解就够了。通常情况下我们很少整那么多花样。一般要么插入一个实体类，要么插入一个集合。

@Insert标记的注解其实是可以有返回值的

If the `@Insert` method receives a single parameter, it can return a `long` value, which is the new `rowId` for the inserted item. If the parameter is an array or a collection, then the method should return an array or a collection of `long` values instead, with each value as the `rowId` for one of the inserted items. To learn more about returning

如果插入的参数是一个实体类放回值要么没有，要么就是Long，这个long的含义是SQL中的rawid

而SQL里面的rawid好像是INTEGER类型的PrimaryKey(因为PrimaryKey可以是SQL的TEXT类也就是String类)

2.0 Quirks

The `PRIMARY KEY` of a rowid table (if there is one) is usually not the true primary key for the table, in the sense that it is not the unique key used by the underlying B-tree storage engine. The exception to this rule is when the rowid table declares an `INTEGER PRIMARY KEY`. In the exception, the `INTEGER PRIMARY KEY` becomes an alias for the `rowid`.

如果@Insert方法传入的是一个集合，那么返回的值可以是List，这List也就是rawid的集合。

@Insert注解里面有两个值一个是 entity，一个是onConflict。

- entity

先看看大概长什么样吧

```
@Entity
public class Playlist {
    @PrimaryKey(autoGenerate = true)
    long playlistId;

    String name;
    @Nullable
    String description

    @ColumnInfo(defaultValue = "normal")
    String category;
    @ColumnInfo(defaultValue = "CURRENT_TIMESTAMP")
    String createdTime;
    @ColumnInfo(defaultValue = "CURRENT_TIMESTAMP")
    String lastModifiedTime;
```

```

}

public class NameAndDescription {
    String name;
    String description
}

@Dao
public interface PlaylistDao {
    @Insert(entity = Playlist.class)
    public void insertNewPlaylist(NameAndDescription
nameDescription);
}

```

我们可以看出Playlist有3个变量是默认生成的，一个变量是primaryKey并设置了自动生成，也就是说如果我们需要插入一个Playlist变量到数据库，只需要给出name和description变量即可。

为了方便我们插入Playlist，我们可以把name和description声明为一个新的类，然后把新的类作为参数传入到@Insert标记的方法中。最后声明entity = Playlist.class这个声明的意思是传入的参数是Playlist的一部分。这样实际插入的是Playlist。**总感觉有点画蛇添足。**

- onConflict

看看长什么样。

```

@Insert(onConflict = OnConflictStrategy.ABORT)
abstract fun insertAll(vararg users: User):List<Long>

```

在了解任何处理插入冲突之前先了解什么是插入冲突。

我们之前讲过PrimaryKey，一个Entity必须要有一个PrimaryKey。因为PrimaryKey有特殊的用处。**PrimaryKey是区分不同行的重要标准。**

回归到插入冲突，前面说了PrimaryKey是判断不同行的标准。试想一个情景。

如果我取消了PrimaryKey的autoGenerate，当我插入数据的时候，两个数据PrimaryKey都是默认指定的2就像这样。

```

@Entity(tableName = "user_table")
data class User(
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
){
    @PrimaryKey(/*autoGenerate = true*/) var uid:Int = 2
    override fun toString(): String {
        return "$uid-$firstName-$lastName"
    }
}

```

没错这样就发生了插入冲突。

现在我们知道了插入冲突，那你认为Room会怎么做？答案是app Crash了

```

database.sqlite.SQLiteConstraintException: UNIQUE constraint failed: user_table.uid (SQLite code 1555 SQLITE_CONSTRAINT_PRIMARYKEY), (OS error - 2:No such file or directory)
    at android.database.sqlite.SQLiteConnection.nativeExecuteForLastInsertedRowId(Native Method)
    at android.database.sqlite.SQLiteConnection.executeForLastInsertedRowId(SQLiteConnection.java:923)
    at android.database.sqlite.SQLiteSession.executeForLastInsertedRowId(SQLiteSession.java:818)
    at android.database.sqlite.SQLiteStatement.executeInsert(SQLiteStatement.java:88)
    at androidx.sqlite.db.framework.FrameworkSQLiteStatement.executeInsert(FrameworkSQLiteStatement.java:51)
    at androidx.room.EntityInsertionAdapter.insertAndGetRowId(EntityInsertionAdapter.java:221)
    at androidx.room.RoomDatabase$1.insertAll(RoomDatabase.java:72)
    at androidx.room.RoomDatabase.insertAll(RoomDatabase.java:75)
    at androidx.room.RoomDatabase.insertAll(RoomDatabase.java:75)
    at com.example.roomdb.UserDao_Impl.insertAll(UserDao_Impl.java:72)
    at com.example.roomdb.MainActivity$setOnClickListener$5$1.invokeSuspend(MainActivity.kt:48)
    at kotlin.coroutines.jvm.internal.ContinuationImpl.resumeWith(ContinuationImpl.kt:104)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:106)
    at kotlinx.coroutines.safeargs房间里

```

UNIQUE constraint failed。 (也就是Primarykey是唯一的)
Room给出处理插入冲突的默认方式就是OnConflictStrategy.ABORT也即是回滚，
抛异常。

除此之外还有两种方式

- OnConflictStrategy.IGNORE 忽略掉，就当什么都没发生。
- OnConflictStrategy.REPLACE 将新的值替换掉旧的值。

比如我先插入了 User(PrimaryKey = 1,data = "1")

然后又插入了User(PrimaryKey = 1,data = "2")

那么Room会将User(PrimaryKey = 1,data = "2")替换掉User(PrimaryKey = 1,data = "1")所对应的位置。

◦ @Updata

从Updata这个单词就能看出这个是用于更改某一行数据的，和@Insert基本上是一致的。。连文档的例子都是一样的。我属实有些懵了。

@Update标记的方法接受一个实体对象，或者是一个实体集合。代码如下

```
@Dao
interface UserDao {
    @Update
    fun updateUsers(vararg users: User)
}
```

@Updata依然是靠的PrimaryKey匹配行。

不过多阐述了。就当是@Insert得了ha。

◦ @Delete

@Delete依然是通过PrimaryKey索引行（也就是说其他成员变量是否一致并不重要）的，依然与@Insert有点类似，不过不一样的是索引到了对应行是删除，而不是更新插入啥的了。方法参数也是一样的实体类，实体集合

```
@Dao
interface UserDao {
    @Delete
    fun deleteUsers(vararg users: User)
}
```

还有@Delete标记的方法的返回值可以是Unit也可以是Int。

Int表示成功删除的数据的个数。注解内的有个entity变量，前面其实以及讲过。

通常情况下如果使用@Delete得先通过@Query查询匹配结果，然后再删除。当然也可以直接使用@Query删除。反正@Query啥都能干。

◦ @Query

Query的参数很简单，就一个String，而这个String是用于与SQL交互的。可能下列的代码看不太懂，不给知道能这么写就好了。

- 简单查询

```

@Dao
interface MyDao {
    //查询加载所有User
    @Query("SELECT * FROM user")
    fun loadAllUsers(): Array<User>
}

```

■ 传递参数给Query

```

@Dao
interface MyDao {
    //查询满足条件age大于传入的minAge的所有User
    @Query("SELECT * FROM user WHERE age > :minAge")
    fun loadAllUsersOlderThan(minAge: Int): Array<User>
}

```

上面的例子只传入了单个参数，其实还可以传入多个参数。

```

@Dao
interface MyDao {
    @Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
    fun loadAllUsersBetweenAges(minAge: Int, maxAge: Int): Array<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :search " +
           "OR last_name LIKE :search")
    fun findUserWithName(search: String): List<User>
}

```

■ 返回列的子集

```

data class NameTuple(
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)

//这个表明我们只返回所有user的first_name和last_name
@Dao
interface MyDao {
    @Query("SELECT first_name, last_name FROM user")
    fun loadFullName(): List<NameTuple>
}

```

■ 传递参数的集合

```

@Dao
interface MyDao {
    @Query("SELECT first_name, last_name FROM user WHERE region
IN (:regions)")
    fun loadUsersFromRegions(regions: List<String>): List<NameTuple>
}

```

- 直接光标访问

```
@Dao
interface MyDao {
    @Query("SELECT * FROM user WHERE age > :minAge LIMIT 5")
    fun loadRawUsersOlderThan(minAge: Int): Cursor
}
```

对于Cursor我不是很清楚，网上找了相关描述

- Cursor 是每行的集合。
- 使用 moveToFirst() 定位第一行。
- 你必须知道每一列的名称。
- 你必须知道每一列的数据类型。
- Cursor 是一个随机的数据源。
- 所有的数据都是通过下标取得。

Google不是很推荐使用Cursor，除非你认为你的需求只有使用Cursor才能很好满足的时候。**使用前一定要三思。**

- 查询多个表格

以下代码段展示了如何执行表格联接以整合以下两个表格的信息：**一个表格包含当前借阅图书的用户，另一个表格包含当前处于已被借阅状态的图书的数据。**

```
@Dao
interface MyDao {
    @Query(
        "SELECT * FROM book " +
        "INNER JOIN loan ON loan.book_id = book.id " +
        "INNER JOIN user ON user.id = loan.user_id " +
        "WHERE user.name LIKE :userName"
    )
    fun findBooksBorrowedByNameSync(userName: String): List<Book>
}
```

我也看不太懂，知道能在多个表查询即可。有需要的可以看看[这个](#)（在此之前还是先学SQL吧）

@Database

RoomDatabase的标签。

@Database中有5个值：

entities, views, version, exportSchema, autoMigrations

- entities

这个我们接触的也算比较多的，主要用于在数据库声明实体。

- views

不是很懂，这个好像和一个视图数据库有些关系。

- version

数据库当前的版本号

- exportSchema

导出schema文件也就是Room结构文件。

- autoMigrations

这个自动迁移是依靠schema文件的结构，所以exportSchema一定得是true

Room 2.3.0暂时用不了。

```
@SuppressLint("RestrictedApi")
@Database(entities = {User.class}
        , version = 1
        , autoMigrations = {
            @AutoMigration(
                from = 1,
                to = 2
            ),
            @AutoMigration(
                from = 2,
                to = 3
            )
        })

```

得升级2.4.0 alpha才行

```
@SuppressLint("RestrictedApi")
@Database(entities = {User.class}
        , version = 1
        , autoMigrations = {
            @AutoMigration(
                from = 1,
                to = 2
            ),
            @AutoMigration(
                from = 2,
                to = 3
            )
        })
abstract class Test extends RoomDatabase{}
```

4.Room进一步探究

1.Room 数据库的迁移

参考自

当您在应用中添加和更改功能（版本改变）时，需要修改 Room 实体类。但是，如果应用更新更改了数据库架构，我们如何将之前版本的用户数据保存下来就很重要。而这也是数据库迁移需要解决的问题。

Room是通过Migration类进行数据库版本的迁移的，通过重写Migration的migrate方法实现数据库的迁移。以在运行时将数据库迁移到合适的版本。

在此之前介绍一个东西，schema文件。schema文件是一个json文件，它包含了数据库的结构图。当进行版本迁移后它能很好的反映版本的变化情况。

导出schema的设置默认是开着的，但是在build的时候程序并不知道schema文件放在哪，所以我们只需要将schema文件的存放地址生命就好了。（如果不打算导出schema记得给在@Database中加入exportSchema = false，不加这个也可以，程序可以运行。只不过在gradle build的过程中可能爆红，有点碍眼。）

```
defaultConfig{  
    ....  
    javaCompileOptions {  
        annotationProcessorOptions {  
            arguments = ["room.schemaLocation":  
"$projectDir/schemas".toString()]  
        }  
    }  
    ....  
}
```

- 手动迁移

Room 会从一个或多个 Migration 子类运行 migrate() 方法，以在运行时将数据库迁移到最新版本：

比如这样

有些烦人的是手动迁移需要和sql语句打交道，不太懂，欸。

```
val MIGRATION_1_2 = object : Migration(1, 2) {  
    override fun migrate(database: SupportsSQLiteDatabase) {  
        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, "  
+  
        "PRIMARY KEY(`id`))")  
    }  
}  
  
val MIGRATION_2_3 = object : Migration(2, 3) {  
    override fun migrate(database: SupportsSQLiteDatabase) {  
        database.execSQL("ALTER TABLE Book ADD COLUMN pub_year INTEGER")  
    }  
}  
  
Room.databaseBuilder(applicationContext, MyDb::class.java, "database-name")
```

```
.addMigrations(MIGRATION_1_2, MIGRATION_2_3)  
.build()
```

- 自动迁移

我在看文档的时候我发现了一个很...的事情，[数据库迁移的文档](#)中文和英文版竟然不一样。
中文少了一个自动迁移。

图片为证

目录

[测试迁移](#)

[导出架构](#)

[测试单次迁移](#)

[测试所有迁移](#)

[妥善处理缺失的迁移路径](#)

[升级到 Room 2.2.0 时处理列默认值](#)

Table of contents

[Automated migrations](#)

Automatic migration specifications

[Manual migrations](#)

[Test migrations](#)

Export schemas

Test a single migration

Test all migrations

Gracefully handle missing migration paths

Handle column default value when upgrading to Room 2.2.0

这么说是不是之前看的文档是不是都可能少了点东西（雾。

淦中文文档更新滞留了.....（暗示学英文。

Content and code samples on this page are subject to the licenses described in the [Content License](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated 2020-07-10 UTC.

这告诉我们以后看文档还是稍微注意一下文档更新时间。如果是前几年更新的说不定少了点什么。.

不扯了。其实自动迁移还只在测试版所以还可能出现一些变化，暂时只有英文文档上有相关介绍，有需要的兄弟可以去看看。由于是测试版我就不多讲了。（学了万一变更了呢 hh）

★ Note: Room supports automated migrations in `version 2.4.0-alpha01` and higher. If your app uses a lower version of Room, you must [define your migrations manually](#).

在此稍微提一下，一般库的版本要经历三个阶段

alph, beta, stable(正式版)

alph -- 预览版，bug最多，可能存在比较大的变更。

beta -- 测试版，虽然经历了alph的阶段，但是仍然存在一些bug，但是通常情况下很少出现变更了，这时候就可以去学了。

stable -- 正式版，这个基本上bug就比较少了，内容就更小几率发生变更了。

- 破坏性迁移

我们知道如果数据库版本变化**但是程序又没有找到对应的迁移策略**，那么就会抛出一个`IllegalStateException`。有的时候我们软件版本变化太大了，以至于数据库的结构发生了翻天覆地的变化，保留数据已近很困难了。那么就可以选择直接丢弃掉当前数据库里面的数据，让数据库版本进行升级。

让Room采用这种迁移方式很简单，只需要让它在Build的时候加入`fallbackToDestructiveMigration()`即可。

如下

```
Room.databaseBuilder(applicationContext, MyDb::class.java, "database-name")
    .fallbackToDestructiveMigration()
    .build()
```

注意：这个方法是用于**没有定义迁移策略**的时候调用，如果定义了就不会调用。

如果您只想让 Room 在特定情况下回退到破坏性重新创建，可以使用

`fallbackToDestructiveMigration()` 的一些替代选项：

- 如果你想在某些版本的迁移中使用破坏性迁移，可以选用`fallbackToDestructiveMigrationFrom()`，此方法接受多个int参数，每个int表示进行破坏性迁移的版本值。比如某app在版本4到版本5变更巨大，采用破坏性迁移，那么只需往`fallbackToDestructiveMigrationFrom()`传入4即可。
- 如果只有在高版本到低版本的时候进行破坏性迁移，那么就可以使用这个`fallbackToDestructiveMigrationOnDowngrade()`。

- 特殊的迁移

这种迁移是为了解决一个bug。

在很多时候给参数加入默认值这是很常见的一需求。但是在Room 2.2.0以前加入默认值的方式只有一种，那就是利用sql语句迁移的过程中添加一个。

不像在2.2.0以后可以直接使用`@ColumnInfo(defaultValue = "...")`

看下面一个实例。

如果用户在版本1到版本2迁移过程中在数据表单中添加了一列并设置了默认值。

```
//版本1下的实体类 Room版本为2.1.0
// Song Entity, DB Version 1, Room 2.1.0
@Entity
data class Song(
    @PrimaryKey
    val id: Long,
    val title: String
)

//版本2下的实体类 Room版本为2.1.0
// Song Entity, DB Version 2, Room 2.1.0
@Entity
data class Song(
    @PrimaryKey
    val id: Long,
    val title: String,
    val tag: String // Added in version 2.
)

//从版本1迁移到版本2的策略
// Migration from 1 to 2, Room 2.1.0
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        //创建了新的一列‘tag’并设置默认值为‘
        database.execSQL(
            "ALTER TABLE Song ADD COLUMN tag TEXT NOT NULL DEFAULT ''")
    }
}
```

乍一看这代码是没有问题的。如果这样想：这个默认值是在数据库迁移的过程中进行设置的，但是如果不进行迁移呢？也就是说直接安装数据库版本号对应为2的软件。这样躲过了数据库的迁移，你会发现直接安装版本2的数据库没有设置默认值。而迁移的有默认值。这造成了数据库版本2的数据库结构不一致。但在2.1.0版本这并不会造成什么问题。

但但是，如果你在这个时候将Room升级到了2.2.0以及以上并使用了@ColumnInfo设置默认值就会导致架构验证错误。（可能会直接crash，不清楚没试过）

所以为了让Room升级到2.2.0时的数据库结构一致。可以在之前的版本2上进行一次特殊的迁移。

迁移需要完成一下3步

1. 使用 @ColumnInfo 注释在各自的实体类中声明列默认值。
2. 将数据库版本号增加 1。
3. 定义实现了删除并重新创建策略的新版本迁移路径，将必要的默认值添加到现有列。

第一步是为了让直接安装版本3的数据库具有默认值。

第三步是为了保证迁移过程中将没有默认值的数据库转化成有默认值的数据库。

第三步操作的代码如下

```
//迁移过程中先创建一个new_Song的数据表单，在创建过程中设置默认值。
//然后将Song数据表单复制到new_Song中去。
//最后删除Song表单将new_Song重命名为Song表单。
```

```
// Migration from 2 to 3, Room 2.2.0
val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("""
            CREATE TABLE new_Song (
                id INTEGER PRIMARY KEY NOT NULL,
                name TEXT,
                tag TEXT NOT NULL DEFAULT ''
            )
        """.trimIndent())
        database.execSQL("""
            INSERT INTO new_Song (id, name, tag)
            SELECT id, name, tag FROM Song
        """.trimIndent())
        database.execSQL("DROP TABLE Song")
        database.execSQL("ALTER TABLE new_Song RENAME TO Song")
    }
}
```

除此之外还有一种迁移，只不过和预填充数据库有些关系，就放在了预填充数据库哪里去了。

2.预填充数据库

参考自

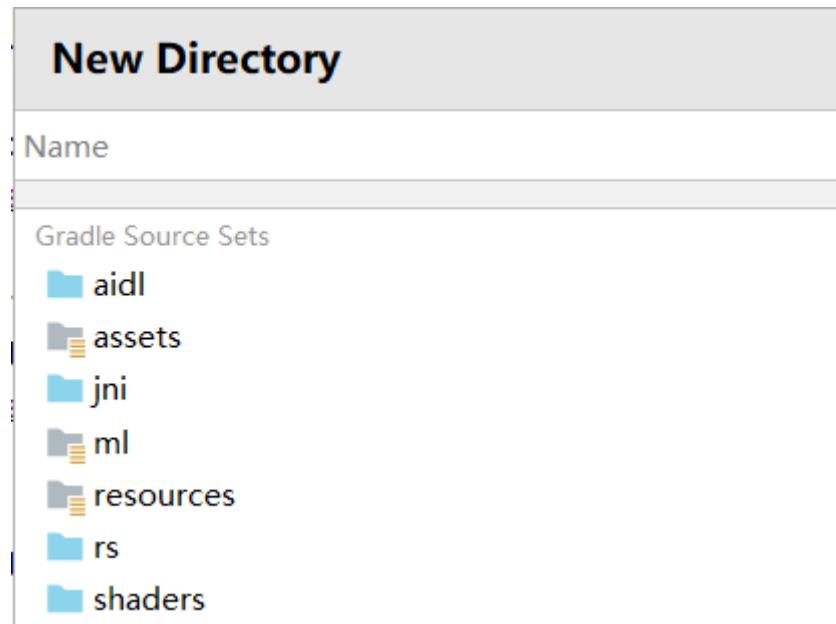
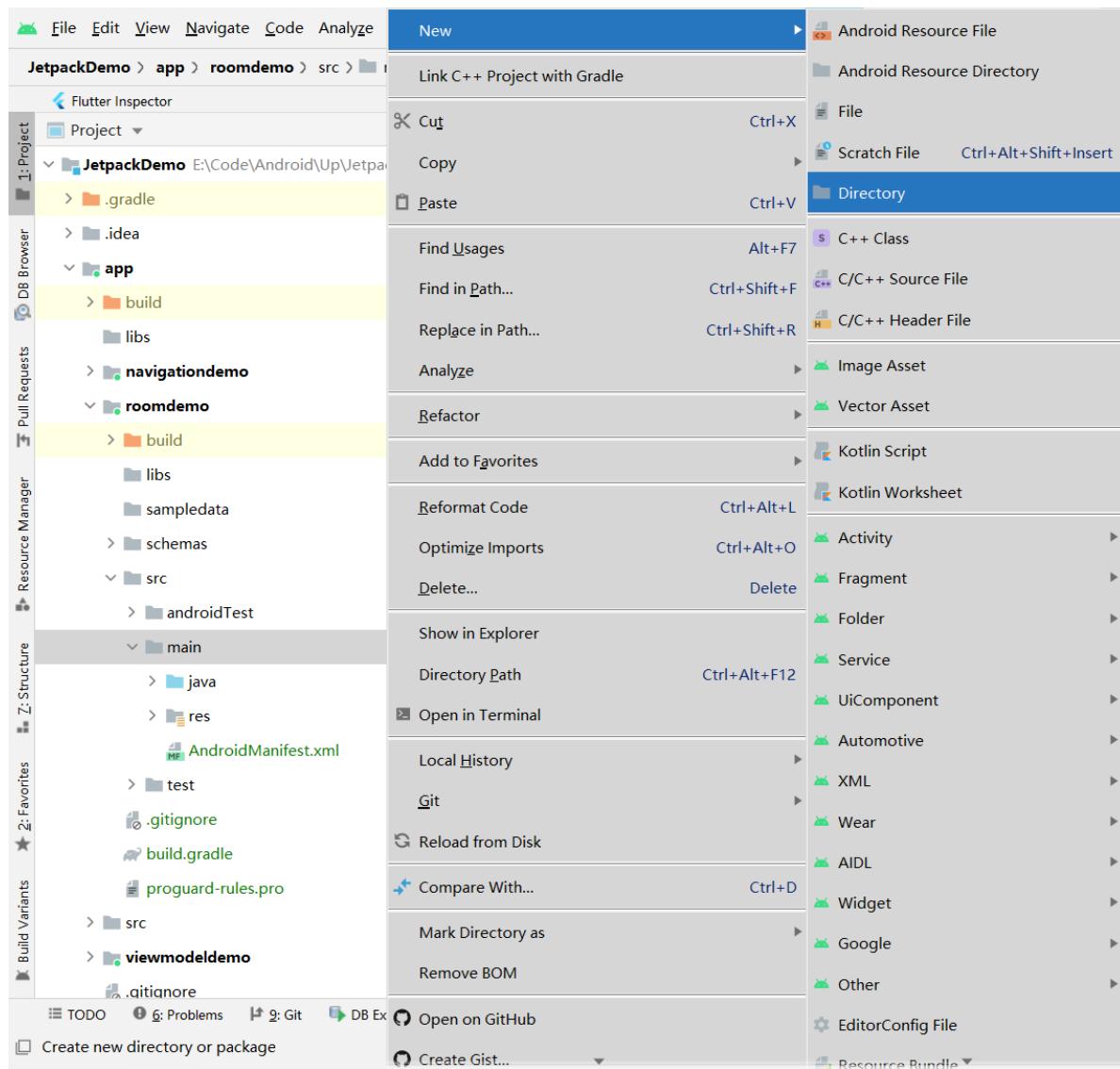
有时，您可能希望应用启动时数据库中就已经加载了一组特定的数据。这称为预填充数据库。在 **Room 2.2.0 及更高版本** 中，您可以使用 API 方法在初始化时用设备文件系统中预封装的数据库文件中的内容预填充 Room 数据库。

注意：缓存Room 数据库不支持使用 `createFromAsset()` 或 `createFromFile()` 预填充数据库。

缓存数据库就是创建在内存中的数据库，当程序退出数据库的资源全部回收。创建方法很简单，使用 `Room.inMemoryDatabaseBuilder()` 进行创建即可（创建方式与 `Room.databaseBuilder()` 基本上一致）

从应用资源预填充

`assets/` 文件某种意义上来说也算是一个数据库的，这个文件夹是默认不创建的，需要我们自己创建。
创建方式如下



所以预填充数据库与他确实有亿点关系。

如果你想从assets目录下读取文件并预填充数据库，那么可以在Room.databaseBuilder()调用build()前使用createFromAsset()，该方法接收一个String，也就是预填充的文件在assets中的位置。

比如这样

```
Room.databaseBuilder(applicationContext, AppDatabase::class.java,  
    APP_DATABASE_NAME)  
    .createFromAsset("database/myapp.db")  
    .build()
```

注意：从某个资源预填充时，Room 会验证数据库，以便确保其架构与预封装数据库的架构相匹配。在创建预封装数据库文件时，您应[导出数据库的架构](#)以作为参考。

从文件系统预填充

如需从位于设备文件系统任意位置（应用的 assets/ 目录除外）的预封装数据库文件预填充 Room 数据库，请先从 RoomDatabase.Builder 对象调用 createFromFile() 方法，然后再调用 build()：

```
Room.databaseBuilder(appContext, AppDatabase.class, "Sample.db")  
    .createFromFile(File("mypath"))  
    .build()
```

与前一个是类似的。

根据文档描述：预填充数据库是通过将预填充文件复制进自己app定义的数据库文件中，而不是直接使用预填充数据库的文件。所以是需要预填充文件的读取权限的。

createFromAsset

```
@NonNull open fun createFromAsset(@NonNull databaseFilePath: String): RoomDatabase.Builder<T>
```

Configures Room to create and open the database using a pre-packaged database located in the application 'assets/' folder.

Room does not open the pre-packaged database, instead it copies it into the internal app database folder and then opens it. The pre-packaged database file must be located in the "assets/" folder of your application. For example, the path for a file located in "assets/databases/products.db" would be "databases/products.db".

createFromFile

```
@NonNull open fun createFromFile(@NonNull databaseFile: File): RoomDatabase.Builder<T>
```

Configures Room to create and open the database using a pre-packaged database file.

Room does not open the pre-packaged database, instead it copies it into the internal app database folder and then opens it. The given file must be accessible and the right permissions must be granted for Room to copy the file.

处理包含预封装数据库的迁移

我们知道fallbackToDestructiveMigration()会直接销毁掉所有的数据。**但是在破坏性迁移的同时我们还可以加上预填充**，这样破坏性迁移以后会默认使用预填充填充数据库。

代码如下

这种情况下，在破坏性迁移以后会自动预填充。

```

// Database class definition declaring version 3.
@Database(version = 3)
abstract class AppDatabase : RoomDatabase() {
    ...
}

// Destructive migrations are enabled and a prepackaged database
// is provided.
Room.databaseBuilder(appContext, AppDatabase::class, "Sample.db")
    .createFromAsset("database/myapp.db")
    .fallbackToDestructiveMigration()
    .build()

```

但是这种情况下并不会。因为并不是破坏性迁移。

```

// Database class definition declaring version 3.
@Database(version = 3)
abstract class AppDatabase : RoomDatabase() {
    ...
}

// Migration path definition from version 2 to version 3.
val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        ...
    }
}

// A prepackaged database is provided.
Room.databaseBuilder(appContext, AppDatabase::class, "Sample.db")
    .createFromAsset("database/myapp.db")
    .addMigrations(MIGRATION_2_3)
    .build()

```

这个数据库的迁移会经历这样的步骤

- 由于没有定义2_3的迁移方式，会启动破坏性迁移。又由于加入了预填充数据库，所以在破坏性迁移以后会启用预填充。
- 又由于加入了3_4的迁移，所以在预填充以后会加载3_4的迁移。
- 最后由于预填充会将预填充文件复制到app的数据库所以预填充文件得以保留。数据库版本变更到4

```

//Tips:当前数据库版本为2
// Database class definition declaring version 4.
@Database(version = 4)
abstract class AppDatabase : RoomDatabase() {
    ...
}

// Migration path definition from version 3 to version 4.
val MIGRATION_3_4 = object : Migration(3, 4) {
    override fun migrate(database: SupportSQLiteDatabase) {
        ...
    }
}

```

```
}

// Destructive migrations are enabled and a prepackaged database is
// provided.
Room.databaseBuilder(appContext, AppDatabase.class, "Sample.db")
    .createFromAsset("database/myapp.db")
    .addMigrations(MIGRATION_3_4)
    .fallbackToDestructiveMigration()
    .build()
```

3. 定义对象之间的关系

参考自：

[Google 文档](#)

[博客地址](#)

由于 SQLite 是关系型数据库，因此您可以指定各个实体之间的关系。尽管大多数对象关系映射库都允许实体对象互相引用，但 Room 明确禁止这样做。如需了解此决策背后的技术原因，请参阅[了解 Room 为何不允许对象引用](#)。（主要原因还是性能问题。）

创建嵌套对象

有时，我们存在一种需求就是：将某个实体或数据对象在数据库逻辑中表示为一个紧密的整体。我们可以使用@Embedded实现。代码如下

```
data class Address(
    val street: String?,
    val state: String?,
    val city: String?,
    @ColumnInfo(name = "post_code") val postCode: Int
)

@Entity
data class User(
    @PrimaryKey val id: Int,
    val firstName: String?,
    @Embedded val address: Address?
)
```

这样 User 对象表中就包含 id、firstName、street、state、city 和 post_code。

简单来讲就是：如果Room表单实体类和实体类之间如果存在这种嵌套的关系就得利用@Embedded，这样Room才知道这里存在嵌套关系，它才知道这里需要将Address展开。否者他就认为Address只是一个变量。

注意：嵌套字段还可以包含其他嵌套字段。

为了避免@Embedded修饰的变量重复名，提供了@Embedded提供了一个参数prefix，prefix是前缀。上代码

```
@Embedded(prefix = "loc_")
Coordinates coordinates;
```

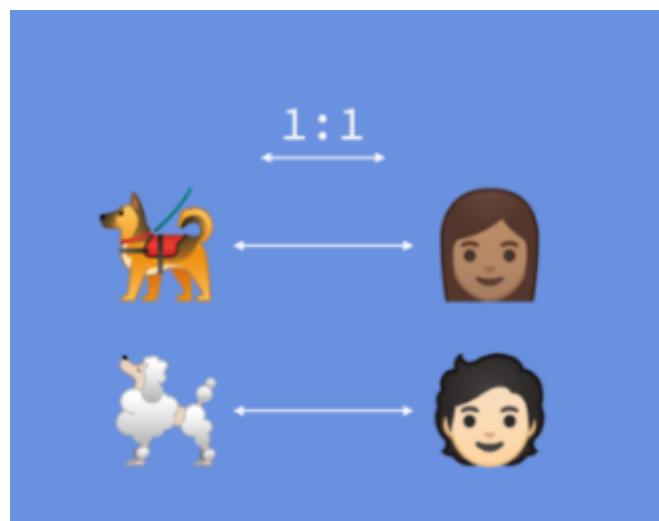
这样Coordinate变量在数据库里的实际名称就变成了loc_coordinates.

定义一对关系

Code Place

```
com/example/roomdemo/model/entity,
com/example/roomdemo/db
```

两个实体之间的一对一关系是指这样一种关系：父实体的每个实例都恰好对应于子实体的一个实例，反之亦然。



假如我们生活在一个(悲伤的)世界，每个人只能拥有一条狗，并且每条狗也只能有一个主人。这就是一对关系。为了在关系型数据库中表示这一关系，我们创建了两张表，Dog 和 Owner。在 Room 中，我们创建两个表

```
@Entity
data class Dog(
    @PrimaryKey val dogId: Long,
    val dogOwnerId: Long,
    val name: String,
    val cuteness: Int,
    val barkvolume: Int,
    val breed: String
)

@Entity
data class Owner(@PrimaryKey val ownerId: Long, val name: String)
```

上述只是建立了实体了，但是还没有建立实体关系。

而建立实体关系需要在建立一个data class代码如下

```
data class DogAndOwnerOneToOne(
    @Embedded val owner: Owner,
    @Relation(
        parentColumn = "ownerId",
        entityColumn = "dogOwnerId"
    )
    val dog: Dog
)
```

我们可以看出DogAndOwnerOneToOne中有两个实体表对象的实例。

并通过@Relation建立了表单与表单的关系。

在实体类中由于Dog具有dogOwnerId也即是说可以通过Dog在Sql中索引到Owner，但是Dog和Owner在对象引用的角度上来看是不存在引用关系的。我们称Dog和Owner具有逻辑关系。这种逻辑关系就是一对关系，其中通过Dog可以索引到Owner故又定义Dog为子实体，Owner为父实体。

在回到一对关系的建立，

- @Relation是作用于子实体的，也即是Dog。
- parentColum是父实体的primaryKey对应的列的名称。
- entityColum是子实体中与父实体PrimaryKey相对的列的名称。

最后我们还需要在Dao中的方法加上一个注解。

```
@Transaction
```

这个注解是为了确保数据库操作的原子性。

```
@Transaction
@Query("SELECT * FROM Owner")
fun getDogAndOwnerOneToOne(): List<DogAndOwnerOneToOne>
```

如果利用SQL来获取DogAndOwnerOneToOne则需要经历以下步骤

- SELECT * FROM Owner
匹配数据库中所有的Owner
- SELECT * FROM Dog
WHERE dogOwnerId IN (ownerId1, ownerId2, ...)
将第一步搜寻的Owner的id与Dog中的dogOwnerId 进行匹配
- 最后映射成DogAndOwnerOneToOne对象返回

定义一对多关系



两个实体之间的一对多关系是指这样一种关系：父实体的每个实例对应于子实体的零个或多个实例，但子实体的每个实例只能恰好对应于父实体的一个实例。

也就是说一个父实体对应多个子实体。

一对多和一对一关系是类似的，主要的差别是关系的建立上。

建立新的Relation

```
@Entity
data class Dog(
    @PrimaryKey val dogId: Long,
    val dogOwnerId: Long,
    val name: String,
    val cuteness: Int,
    val barkvolume: Int,
    val breed: String
)

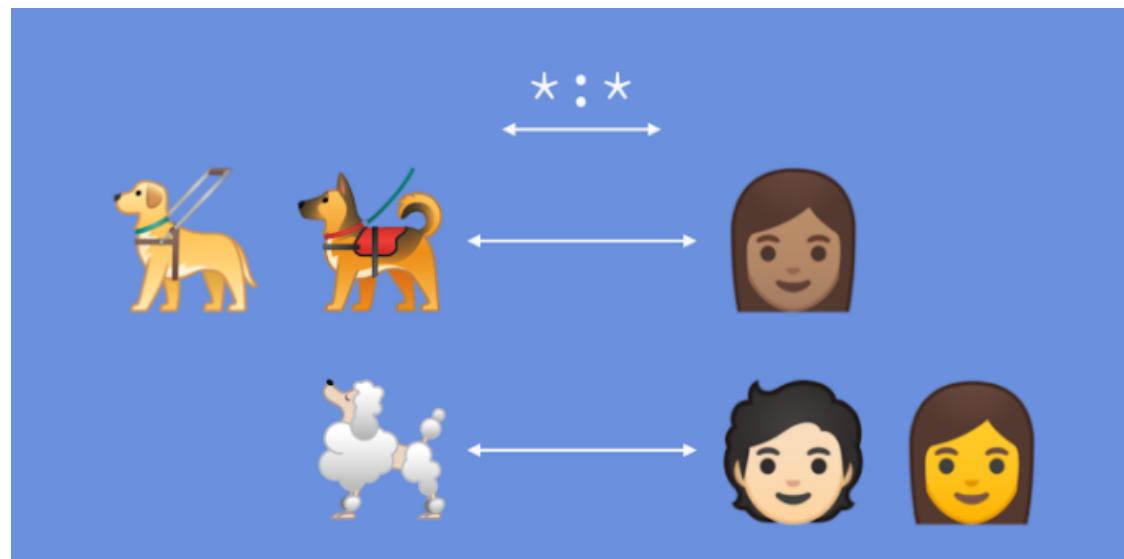
@Entity
data class Owner(@PrimaryKey val ownerId: Long, val name: String)

data class DogAndOwnerOneToMany(
    @Embedded
    val owner: Owner,

    @Relation(
        parentColumn = "ownerId",
        entityColumn = "dogOwnerId"
    )
    val dogs: List<Dog>
)
```

差别也不是很大，dog变成dogs了其余好像都没变化。

定义多对多关系



现在假设我们生活在一个完美的世界，每个主人可以拥有多条狗，每条狗也可以有多个主人。要对此关系进行建模，仅仅通过 Dog 表和 Owner 表是不够的。由于一条狗可能有多个主人，所以同一个 dogId 可能需要多条数据，以匹配不同的主人。但是在 Dog 表中，dogId 是主键，我们不能插入多个 id 相同，主人不同的狗狗。为了解决这一问题，我们需要额外创建一个存储 (dogId, ownerId) 的 **关联表**（也称为 **交叉引用表**）。

主要差异还是关系建立上。

那不简单。这样？

```
data class OwnersWithDogs(
    @Embedded val owners: List<Owner>,
    @Relation(
        parentColumn = "ownerId",
        entityColumn = "dogOwnerId"
    )
    val dogs: List<Dog>
)
```

错的，这样创建没有任何意义。

你会发现 owners 和 dogs 都是独立的。

这样确认从表面上看是 Owners to Dogs，但是这样的关系相互间无法引用，没有意义。所以多对多我们不采用这样的描述方式。

而是通过两个单多描述。

```
//一个Dog多个Owner
data class DogWithOwners(
    @Embedded
    val dog: Dog,
    @Relation(
        parentColumn = "dogId",
        entityColumn = "ownerId",
        associateBy = Junction(DogOwnerCrossRef::class)
    )
    val owner: List<Owner>
)

//一个Owner多个Dogs
data class OwnerWithDogs(
    @Embedded val owner: Owner,
    @Relation(
        parentColumn = "ownerId",
        entityColumn = "dogId",
        associateBy = Junction(DogOwnerCrossRef::class)
    )
    val dogs: List<Dog>
)
```

除此之外还差一个关系表，关系表是用来存储这两个对象的逻辑关系的。（注意两个一对多表内都要通过 associateBy 引入关系表）

```

@Entity(primaryKeys = ["dogId", "ownerId"])
data class DogOwnerCrossRef(
    val dogId: Long,
    val ownerId: Long
)

```

最后在Dao里面声明两个查询方法

```

//many to many
@Transaction
@Query("select * from Owner")
fun getOwnerwithDogs(): List<OwnerwithDogs>

@Transaction
@Query("select * from Dog")
fun getDogwithowners(): List<Dogwithowners>

```

如果我需要建立这样的关系

| ownerId | dogId |
|---------|------------|
| 1 | 2, 4 |
| 2 | 2, 3, 5 |
| 3 | 2, 3, 4, 5 |

ownerId为1的人，持有dogId为2, 4两条狗。

ownerId为.....

```

//将以下关系表插入即可。
dogeAndOwnerDao.insertRelationMap(
    DogOwnerCrossRef(4, 1),
    DogOwnerCrossRef(2, 2),
    DogOwnerCrossRef(3, 2),
    DogOwnerCrossRef(5, 2),
    DogOwnerCrossRef(2, 3),
    DogOwnerCrossRef(3, 3),
    DogOwnerCrossRef(4, 3),
    DogOwnerCrossRef(5, 3)
)

```

然后在监听点击后查询

```

get_dog_and_owner.setOnClickListener {
    lifecyclescope.launch (Dispatchers.IO) {

        val ownerwithDogs = dogeAndOwnerDao.getOwnerwithDogs()
        ownerwithDogs.forEach {
            Log.e(TAG, "getOwnerwithDogs $it" )
        }

        val dogwithOwners = dogeAndOwnerDao.getDogwithOwners()
        dogwithOwners.forEach{
            Log.e(TAG, "getDogwithOwners $it")
        }
    }
}

```

多对多与单对多综合实例

比如我们在做音乐播放器的时候，通常有这样的需求，查询用户的的所有歌单以及每个用户的歌单中包含的所有歌曲。

实体类如下

```

@Entity
data class User(
    @PrimaryKey val userId: Long,
    val name: String,
    val age: Int
)

@Entity
data class Playlist(
    @PrimaryKey val playlistId: Long,
    val userCreatorId: Long,
    val playlistName: String
)

@Entity
data class Song(
    @PrimaryKey val songId: Long,
    val songName: String,
    val artist: String
)

@Entity(primaryKeys = ["playlistId", "songId"])
data class PlaylistSongCrossRef(
    val playlistId: Long,
    val songId: Long
)

```

我们可以得知：

- User和Playlist是一对多的关系。
- Playlist和Song是多对多的关系。

建立User和Playlist的关系

```

data class PlaylistwithSongs(
    @Embedded val playlist: Playlist,
    @Relation(
        parentColumn = "playlistId",
        entityColumn = "songId",
        associateBy = @Junction(PlaylistSongCrossRef::class)
    )
    val songs: List<Song>
)

```

建立User和Playlist的关系。

```

data class UserwithPlaylistsAndSongs {
    @Embedded val user: User
    @Relation(
        entity = Playlist::class,
        parentColumn = "userId",
        entityColumn = "userCreatorId"
    )
    val playlists: List<PlaylistwithSongs>
}

```

总结

- 上述的方法皆是解决的Room表单中的**实体对象**关系建立的问题。

我们通常想到的对象关系就是引用，但是由于引用关系会多Room数据库造成性能问题，所以Room禁止，Room提倡使用注解的方式建立对象间的逻辑关系从而提高效率。

- 建立关系一般有以下几步

- 在保持原有的**实体对象**不变的情况下，新创建一个用于描述实体类之间的关系的类。
- 在通过对新创建的关系类加入@Relation的注解描述关系。（如果是多对多关系还得再创建一个交叉引用表）
- 最后在Dao里面添加对应的查询语句。（别忘了@Transaction注解确保原子性）

4.对于复杂数据的处理

之前的所有操作都是对简单的对象进行处理，比如Int, String, Long, Double, Float...这种。如果遇上复杂的对象类型（除基本数据类型和数组外的类型）Room其实是不认识的。

这就引入了另一个注解@TypeConverter

如果我们的Entity是这样的

```

@Entity
data class ConverterEntity(
    val data: Date
)

```

当我们build的时候就会爆这样的错误。

因为Room不知道Date是个什么类型。它推荐我们使用@TypeConverter

: Cannot figure out how to save this field into database. You can consider adding a type converter for it. :

首先创建一个一个class，对方法加入@TypeConverter注解

由于Room只知道基本数据类型，如果我们传入复杂的类型也只能通过将其转化为基本数据类型进行存储。

加入@TypeConverter后Room会判断方法传入的变量和返回值。

比如dateToTimestamp Room存储Date的时候就会自动调用**把Date转化为Long然后再存储**。

相似的fromTimestamp，Room会在取出过程中需要将**Long转化为Date**的时候**自动调用**。

```
class Converters {  
  
    @TypeConverter  
    fun fromTimestamp(value: Long?): Date? {  
        return value?.let { Date(it) }  
    }  
  
    @TypeConverter  
    fun dateToTimestamp(date: Date?): Long? {  
        return date?.time?.toLong()  
    }  
}
```

光这样还是不够的，还得把converter转载到Database中去。

```
@Database(version = 1,entities = [ConverterEntity::class])  
@TypeConverters(Converters::class)  
abstract class ConverterDatabase : RoomDatabase() {  
    abstract fun getConverterDao():ConverterDao  
  
    companion object{  
        var instance:ConverterDatabase? = null  
        @Synchronized  
        fun getInstance(applicationContext:Context): ConverterDatabase {  
            instance?.let {  
                return it  
            }  
            return  
        }  
        Room.databaseBuilder(applicationContext,ConverterDatabase::class.java,  
            CONVERTER_DATA_BASE_NAME)  
            .build()  
    }  
}
```