1.LifeCycle

MyJetpackStudy: jetpack学习 (gitee.com)

Lifecycle是什么?

lifecycle 是属于Android Jetpack（官方开发工具包）— Architecture（架构组件）中的一员。

官方介绍:构建生命周期感知型组件，这些组件可以根据 Activity 或 Fragment 的当前生命周期状态调整行为。
白话:lifecycle可以和Activity或Fragment生命周期绑定，方便我们做一些跟生命周期相关的业务逻辑。

Lifecycle的特点

1.生命周期感知：Lifecycle 可以感知应用组件的生命周期状态，包括活动状态、暂停状态、停止状态等。
2.简化生命周期回调：使用 Lifecycle，可以避免在应用组件中手动实现繁琐的生命周期回调。
3.灵活性：Lifecycle 可以与其他 Jetpack 组件（如 ViewModel、LiveData 等）结合使用，提供更灵活的生命周期管理方案。
4.生命周期安全性：Lifecycle 能够确保在正确的生命周期状态下执行相应的操作，避免了可能导致崩溃或内存泄漏的问题。

通过 Lifecycle，开发者可以更好地管理和控制应用组件的生命周期，执行与生命周期相关的操作，如初始化资源、释放资源、注册/注销观
察者等。它可以帮助开发者编写更健壮、可维护的代码，并提供更好的用户体验。

Lifecycle使用

1.先看看可以引用哪些?

```
//androidx
implementation 'androidx.appcompat:appcompat:1.3.0'
```

```
   //非androidx
  dependencies {
    def lifecycle_version = "1.1.1"
    // 包含ViewModel和LiveData
    implementation "android.arch.lifecycle:extensions:$lifecycle_version"
    // 仅仅包含ViewModel
    implementation "android.arch.lifecycle:viewmodel:$lifecycle_version" // For Kotlin use
viewmodel-ktx
    // 仅仅包含LiveData
    implementation "android.arch.lifecycle:livedata:$lifecycle_version"
    // 仅仅包含Lifecycles
    implementation "android.arch.lifecycle:runtime:$lifecycle_version"
    annotationProcessor "android.arch.lifecycle:compiler:$lifecycle_version" // For Kotlin use
kapt instead of annotationProcessor
    // 如果用Java8，用于替代compiler
    implementation "android.arch.lifecycle:common-java8:$lifecycle_version"
    // 可选，ReactiveStreams对LiveData的支持
    implementation "android.arch.lifecycle:reactivestreams:$lifecycle_version"
    // 可选，LiveData的测试
    testImplementation "android.arch.core:core-testing:$lifecycle_version"
}
```

除了Activity和Fragment外，还可以绑定Service和Application的生命周期。只要引入支持相关的可选库即可；官方提到最多的是Activity和Fragment，是因为平时主要用于这两个组件；其实只要有生命周期的组件都可以跟它绑定。而在Android中大多数的组件都是有生命周期的。

2.简单使用案例：

```
package com.example.lifecycledemo1;

import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleObserver;
import android.arch.lifecycle.OnLifecycleEvent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getLifecycle().addObserver(new MyObserver());//1
    }
    public class MyObserver implements LifecycleObserver{
        @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
        void onResume(){
            Log.d(TAG, "Lifecycle call onResume");
        }
        @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
        void onPause(){
            Log.d(TAG, "Lifecycle call onPause");
        }
    }
    @Override
    protected void onResume() {
        super.onResume();
        Log.d(TAG, "onResume");
    }
    @Override
    protected void onPause() {
        super.onPause();
        Log.d(TAG, "onPause");
    }
}
```

工具类封装：

```
//使用：被观察者继承BaseActivity或者添加getLifecycle().addObserver(new BaseLifeCycle());即可
public abstract class BaseActivity <T extends BaseLifeCycle> extends AppCompatActivity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getLifecycle().addObserver(new BaseLifeCycle());
    }
}

//观察者类
```

```
public class BaseLifeCycle implements LifecycleObserver{
    private String TAG="tyl";
    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
    void onCreate(LifecycleOwner owner) {
        Log.e(TAG, "BaseLifeCycle-onCreate:"+owner.getClass().toString());
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    void onStart(LifecycleOwner owner) {
        Log.e(TAG, "BaseLifeCycle-onStart:"+owner.getClass().toString());
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    void onStop(LifecycleOwner owner) {
        Log.e(TAG, "BaseLifeCycle-onStop:"+owner.getClass().toString());
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    void onResume(LifecycleOwner owner) {
        Log.e(TAG, "BaseLifeCycle-onResume:"+owner.getClass().toString());
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    void onPause(LifecycleOwner owner) {
        Log.e(TAG, "BaseLifeCycle-onPause:"+owner.getClass().toString());
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
    void onDestory(LifecycleOwner owner) {
        Log.e(TAG, "BaseLifeCycle-onDestory:"+owner.getClass().toString());
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_ANY)
    void onAny(LifecycleOwner owner) {//无论当前活动处于何种生命周期状态，都会触发相应的生命周期方法
//        Log.e(TAG, "BaseLifeCycle-onAny:"+owner.getClass().toString());
    }

}
```

2.LiveData

MyJetpackStudy: jetpack学习 (gitee.com)

1.什么是LiveData

LiveData是Jetpack组件的一部分，更多的时候是搭配ViewModel来使用，相对于Observable，LiveData的最大优势是其具有生命感知的，

换句话说，LiveData可以保证只有在组件（ Activity、Fragment、Service）处于活动生命周期状态的时候才会更新数据。

LiveData是一个可观察的数据持有者类，与常规的Observable不同，LiveData可感知Activity、Fragment、Service的生命周期，确保

LiveData仅更新处于活动生命周期状态的组件观察者。如果应用程序组件观察者处于started或者resumed，则LiveData认为该组件处于

活跃状态，该组件会收到LiveData的数据更新，而其他注册的组件观察者将不会收到任何数据更新。

LiveData的优点

1.确保界面符合数据状态

LiveData 遵循观察者模式。当底层数据发生变化时，LiveData 会通知 Observer 对象。您可以整合代码以在这些 Observer 对象中

更新界面。这样一来，您无需在每次应用数据发生变化时更新界面，因为观察者会替您完成更新。

2.不会发生内存泄漏

观察者会绑定到 Lifecycle 对象，并在其关联的生命周期遭到销毁后进行自我清理。

3.不会因 Activity 停止而导致崩溃

如果观察者的生命周期处于非活跃状态（如返回堆栈中的 activity），它便不会接收任何 LiveData 事件。

4.不再需要手动处理生命周期

界面组件只是观察相关数据，不会停止或恢复观察。LiveData 将自动管理所有这些操作，因为它在观察时可以感知相关的生命周期状态变化。

5.数据始终保持最新状态

如果生命周期变为非活跃状态，它会在再次变为活跃状态时接收最新的数据。例如，曾经在后台的 Activity 会在返回前台后立即接收最新的

数据。

6.适当的配置更改

如果由于配置更改（如设备旋转）而重新创建了 activity 或 fragment，它会立即接收最新的可用数据。

7.共享资源

您可以使用单例模式扩展 LiveData 对象以封装系统服务，以便在应用中共享它们。LiveData 对象连接到系统服务一次，然后需要相应资

源的任何观察者只需观察 LiveData 对象。

使用 LiveData 对象

请按照以下步骤使用LiveData对象：

1. 创建 LiveData的实例以存储某种类型的数据。这通常在ViewModel类中完成。

2. 创建可定义 onChanged()对象，该方法可以控制当LiveData对象存储的数据更改时会发生什么。通常情况下，您可以在界面控制器
（如 activity 或 fragment）中创建 Observer对象。

3. 使用 observe()方法将Observer对象附加到 LiveData对象。observe()方法会采用LifecycleOwner对象。这样会使Observer对象
订阅LiveData对象，以使其收到有关更改的通知。通常情况下，您可以在界面控制器（如 activity 或 fragment）中附加Observer对象。

当您更新存储在LiveData对象中的值时，它会触发所有已注册的观察者（只要附加的LifecycleOwner处于活跃状态）。
LiveData允许界面控制器观察者订阅更新。当LiveData对象存储的数据发生更改时，界面会自动更新以做出响应。

LiveData的使用

常用方法：

方法名	作用
setValue(T value)：	设置 LiveData 的值，并通知所有活跃的观察者进行更新。此方法应在主线程中调用
postValue(T value)	与setValue()类似，但可以在任何线程中调用。内部使用 Handler 来确保更新操作在主线程执行。
getValue()	获取当前 LiveData 实例的值
observe(LifecycleOwner owner, Observer<? super T> observer)	将观察者（Observer）添加到 LiveData。当 LiveData 的值发生变化时，会通知观察者。该方法传入一个 LifecycleOwner 对象（如 Activity 或 Fragment），LiveData 将观察者与该 LifecycleOwner 的生命周期关联起来，以自动管理观察者的注册和取消注册。
hasObservers()	检查 LiveData 是否有活跃的观察者。
removeObserver(Observer<? super T> observer)	从 LiveData 中移除观察者。
observeForever(Observer<? super T> observer)	与 observe()类似，但观察者不会自动注销。需要手动调用 removeObserver()方法来取消观察。

简单示例：

```
public class NameViewModel extends ViewModel {
    // 使用字符串创建LiveData
    private MutableLiveData<String> currentName;
    public MutableLiveData<String> getCurrentName() {
        if (currentName == null) {
            currentName = new MutableLiveData<String>();
        }
        return currentName;
    }
}
```

```
public class LiveDataActiviry extends BaseActivity {
    private NameViewModel model;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button=findViewById(R.id.button);
        Button button2=findViewById(R.id.button2);
        model = new ViewModelProvider(this).get(NameViewModel.class);
        // 创建用于更新UI的观察者
        Observer<String> nameObserver = new Observer<String>() {
            @Override
            public void onChanged(@Nullable String newName) {
                // 更新UI
                button.setText(newName);
            }
        };
        // 观察LiveData，将此活动作为LifecycleOwner和observer传入
        model.getCurrentName().observe(this, nameObserver);
        //更新LiveData对象的值
        button2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String anotherName = "John Doe";
                //主线程传值
                model.getCurrentName().setValue(anotherName);
                //子线程传值
//                new Thread(new Runnable() {
//                    @Override
//                    public void run() {
//                        model.getCurrentName().postValue("anotherName");
//                    }
//                }).start();
            }
        });
    }
}
```

3.DataBinding

什么是DataBinding

> DataBinding 是谷歌官方发布的一个框架，顾名思义即为数据绑定，是 MVVM 模式在 Android 上的一种实现，用于降低布局和逻辑的
> 耦合性，使代码逻辑更加清晰。
> DataBinding 能够省去我们一直以来的 findViewById() 步骤，大量减少 Activity 内的代码，数据能够单向或双向绑定到 layout 文件
> 中，有助于防止内存泄漏，而且能自动进行空检测以避免空指针异常。

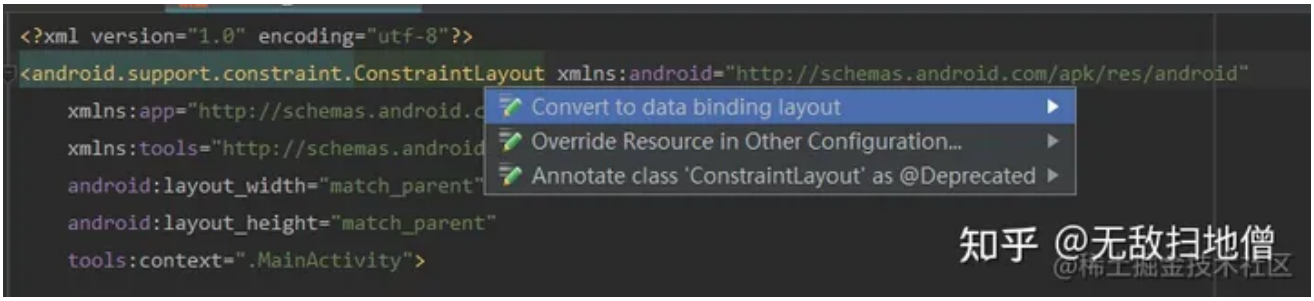
启用 DataBinding 的方法是在对应 Model 的 **build.gradle** 文件里加入以下代码，同步后就能引入对 DataBinding 的支持

```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
```

简单使用

启用 DataBinding 后，这里先来看下如何在布局文件中绑定指定的变量

打开布局文件，选中根布局的 **ViewGroup**，按住 **Alt + 回车键**，点击 “**Convert to data binding layout**”，就可以生成 DataBinding 需要的布局规则



这里先来声明一个 Modle

```
public class User {
    private String name;
    private String pwd;
    public User(String name, String pwd) {
        this.name = name;
        this.pwd = pwd;
    }
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    public String getPwd() {
        return pwd;
```

```
    }

    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

和原始布局的区别在于多出了一个 **layout** 标签将原布局包裹了起来，**data** 标签用于声明要用到的变量以及变量类型，要实现 MVVM 的 ViewModel 就需要把数据（Model）与 UI（View）进行绑定，**data** 标签的作用就像一个桥梁搭建了 View 和 Model 之间的通道

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="user"
            type= "com.tyl.mystudy2.User"/>
    </data>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <TextView
            android:textSize="50dp"
            android:id="@+id/tv1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@{user.name}"
            />
        <TextView
            android:textSize="50dp"
            android:id="@+id/tv2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@{user.pwd}"
            />
    </LinearLayout>
</layout>
```

通过 `@{viewModel.name}` 使 TextView 引用到相关的变量，DataBinding 会将之映射到相应的 **getter** 方法 之后可以在 Activity 中通过 `DataBindingUtil` 设置布局文件，省略原先 Activity 的 `setContentView()` 方法，并为变量 **userInfo** 赋值

如果 User 类型要多处用到，也可以直接将之 **import** 进来，这样就不用每次都指明整个包名路径了，而 `java.lang.*` 包中的类会被自动导入，所以可以直接使用

```
 <data>
        <import type="com.tyl.mystudy2.User"/>
        <variable
            name="user"
            type= "User"/>
    </data>
```

由于 `@{user.name}`在布局文件中并没有明确的值，所以在预览视图中什么都不会显示，不便于观察文本的大小和字体颜色等属性，此时可以为之设定默认值（文本内容或者是字体大小等属性都适用），默认值将只在预览视图中显示，且默认值不能包含引号

```
android:text="@{user.name,default=defaultValue}"
```

在 Activity 中通过 `DataBindingUtil` 设置布局文件，省略原先 Activity 的 `setContentView()` 方法，并为变量 **userInfo** 赋值

```
public class DataBindingActivity extends AppCompatActivity {
    private ActivityDatabindingBinding binding;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //注意DataBindingUtil.setContentView()的返回值是DataBindingUtil工具生成的Binding类，
        //布局文件名来生成，将之改为首字母大写的驼峰命名法来命名，并省略布局文件名包含的下划线
        binding = DataBindingUtil.setContentView(this, activity_databinding);
        //数据是从网络或是数据库拿来的
        User user=new User("jett","123");
        binding.setUser(user);//view.setText(text);
        binding.setVariable(BR.name,"jett");
    }
}
```

此外，也可以通过 ActivityMain2Binding 直接获取到指定 ID 的控件

```
  binding.tv1.setText("tyl");
```

Databinding 同样是支持在 **Fragment** 和 **RecyclerView** 中使用 。例如，可以看 Databinding 在 Fragment 中的使用

```
@Override
    public View onCreateView(@NonNull LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
        FragmentBlankBinding fragmentBlankBinding = DataBindingUtil.inflate(inflater,
R.layout.fragment_blank, container, false);
        fragmentBlankBinding.setHint("Hello");
        return fragmentBlankBinding.getRoot();
    }
```

以上实现数据绑定的方式，每当绑定的变量发生变化的时候，都需要重新向 ViewDataBinding 传递新的变量值才能刷新 UI 。接下来看如何实现自动刷新 UI

驱动 UI 刷新

实现数据变化自动驱动 UI 刷新的方式有三种：`BaseObservable`、`ObservableField`、`ObservableCollection`

BaseObservable

一个纯净的 ViewModel 类被更新后，并不会让 UI 自动更新。而数据绑定后，我们自然会希望数据变更后 UI 会即时刷新，Observable 就是为此而生的概念

BaseObservable 提供了 **notifyChange()** 和 **notifyPropertyChanged()** 两个方法，前者会刷新所有的值域，后者则只更新对应 **BR** 的 **flag**，该 BR 的生成通过注释 **@Bindable** 生成，可以通过 **BR notify** 特定属性关联的视图

```
public class User extends BaseObservable {
```

```
    //如果是 public 修饰符，则可以直接在成员变量上方加上 @Bindable 注解
    @Bindable
    public String name;
    //如果是 private 修饰符，则在成员变量的 get 方法上添加 @Bindable 注解
    private String pwd;
    public User(String name, String pwd) {
        this.name = name;
        this.pwd = pwd;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    @Bindable
    public String getPwd() {
        return pwd;
    }

    public void setPwd(String pwd) {
        //更新所有字段
        this.pwd = pwd;
        notifyChange();
    }
}
```

在 **setName()** 方法中更新的只是本字段，而 **setPwd()** 方法中更新的是所有字段

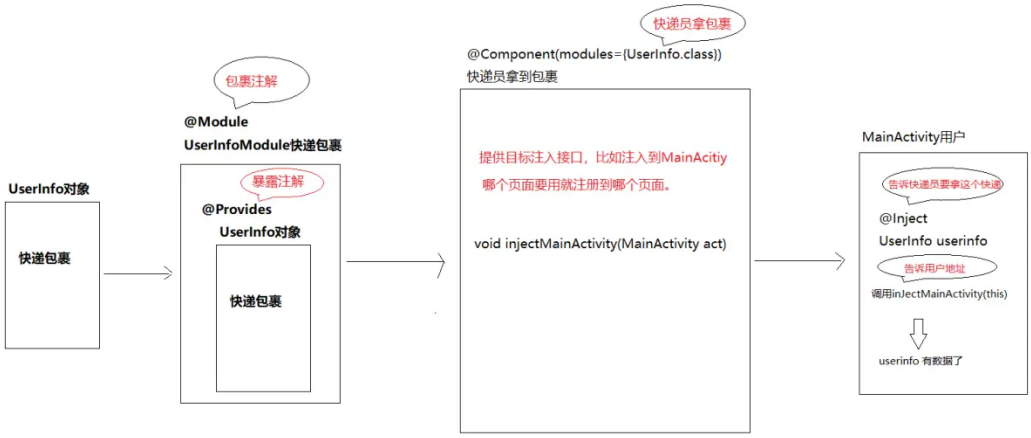
4.Dagger2

什么是Dagger2

> Dagger 2是由Google开发的 依赖注入框架，它利用Java和Java注解处理器的强大功能，提供了一种优雅的方式来进行依赖注入。
> Dagger 2基于一组注解和代码生成器，可以在编译时自动生成依赖注入的代码，从而提高性能和类型安全性。

dagger2 简单理解

> 1.一般我们使用一个实体类或者工具或者请求，比如在MainActivity中使用UerInfo.class，我们会在new Userinfo()，去使用
> 而dagger帮我们省略了这一步，dagger去管理new UserInfo()，我们直接在Activity中使用。
> 2.dagger 理解其实就是相当于买了快递后，快递送货流程
> 实现此流程需要四个注解 @Module @Component @Provides @Inject
> 如下图：



dagger2 的使用

1.在build.gradle中引入插件

```
// dagger2 的功能支持
implementation 'com.google.dagger:dagger:2.4'
// dagger2 自己的注解处理器
annotationProcessor 'com.google.dagger:dagger-compiler:2.4'
```

2.简单使用：

```
//1.HttpObject 对象-->快递
public class HttpObject {
    private String HttpClient="client";
    public String getHttpClient() {
        return HttpClient;
    }
    public void setHttpClient(String httpClient) {
        HttpClient = httpClient;
```

```
    }
}
//2.HttpModule-->包裹
@Module// 包裹注解
public class HttpModule {
    @Provides // 暴露出去注解
    public HttpObject providerHttpObject(){
        //........
        return new HttpObject();
    }
}
//3.Component 类似于快递员，拿到所有包裹->根据地址配给用户
//存放module的组件
@Component(modules = {HttpModule.class, DataModule.class})
public interface MyComponent {
    //注入的位置就写在参数上         不能用多态
    void injectMainActivity(Dagger2Activity mainActivity);
}
//4.用户使用-->接收快递
public class Dagger2Activity extends AppCompatActivity {
    @Inject
    HttpObject httpObject;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dagger);
        //rebuild项目后才有DaggerMyComponent文件,Dagger+自定义的Component名字
        // 方式1： 注入
//        DaggerMyComponent.create().injectMainActivity(this);
        // 方式2:
        DaggerMyComponent.builder().httpModule(new
HttpModule()).build().injectMainActivity(this);
        Log.e("tyl",httpObject.getHttpClient());
    }
}
```

3.dagger2 单例使用

```
1.局域单例：存在于当前类中，只有一个实例，需要增加一个单例注解 @Singleton。
2.全局单例：存在于整个项目，只有一个实例，需要增加一个单例注解 @Singleton，全局单例需要配合Application
使用，否则只能是
    当前activity单例。
```

局域单例示例：(在moudle和Component类中添加@Singleton注解)

```
//1.moudle
@Module
public class HttpModule {
    @Singleton
    @Provides // 暴露出去注解
    public HttpObject providerHttpObject(){
        //........
        return new HttpObject();
    }
}
//2.Compinent类
@Singleton
```

```
@Component(modules = {HttpModule.class})
public interface MyComponent {
    void injectMainActivity(Dagger2Activity dagger2Activity);
}
//3.使用示例:
public class Dagger2Activity extends AppCompatActivity {
    @Inject
    HttpObject httpObject;
    @Inject
    HttpObject httpObject2;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dagger);
        //rebuild项目后才有DaggerMyComponent文件,Dagger+自定义的Component名字
        // 方式1:  注入
//        DaggerMyComponent.create().injectMainActivity(this);
        // 方式2:
        DaggerMyComponent.builder().httpModule(new
HttpModule()).build().injectMainActivity(this);
        Log.e("tyl",httpObject.hashCode()+"");
        Log.e("tyl",httpObject2.hashCode()+"");
        //得出结果值一样
    }
}
```

全局单例示例：(在moudle和Component类中添加@Singleton注解)

```
//1.Component中添加次级要调用的类
@Singleton
@Component(modules = {HttpModule.class})
public interface MyComponent {
    //多少个类就自定义多少个方法
    void injectMainActivity(Dagger2Activity dagger2Activity);
    void injectSecActivity(Dagger2Activity2 dagger2Activity2);
}
//2.application中注入
public class MyApplication extends Application {
    private MyComponent myComponent;
    private static MyApplication context;
    @Override
    public void onCreate() {
        super.onCreate();
        context = this;
         myComponent = DaggerMyComponent.builder()
                .httpModule(new HttpModule())
                .build();
    }
    public static MyApplication getInstance() {
        return context;
    }
    public MyComponent getMyComponent() {
        return myComponent;
    }
}
//3.主activity中执行调用
public class Dagger2Activity extends AppCompatActivity {
```

```
    @Inject
    HttpObject httpObject;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dagger);
        MyApplication.getInstance().getMyComponent().injectMainActivity(this);
        Log.e("tyl",httpObject.hashCode()+"");
    }
    public void jump(View view) {
        startActivity(new Intent(this,Dagger2Activity2.class));
    }
}
//4.次activity中执行调用
public class Dagger2Activity2 extends AppCompatActivity {
    @Inject
    HttpObject httpObject;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        MyApplication.getInstance().getMyComponent().injectSecActivity(this);
        Log.e("tyl",httpObject.hashCode()+"");
    }
}
```

4.多个Component组合依赖

dagger2不能使用多个Component同时注入同一个类中 这种情况需要进行Component的组合;

先确定使用哪个 Component 作为主 Component，确定后，主 Component 仍然执行注入操作，而其他 Component 作为依赖项，不再执行注入，转而提供 Module 提供的对象。一般我们会选择 ApplicationComponent 作为主 Component。

```
//1.新建一个对象DataObject
public class DataObject {
    private String data="data";
    public String getData() {
        return data;
    }
    public void setData(String data) {
        this.data = data;
    }
}
//2.新建一个moudle
@Module
public class DataModule {
    @Provides
    public DataObject providerDatabaseObject(){
        //........
        return new DataObject();
    }
}
//3.新建一个Component
@Component(modules = {DataModule.class})
public interface SecondComponent {
    //使用依赖关系，就不再使用这种语法
    //void injectMainActivity(Dagger2Activity dagger2Activity);
```

```
    //直接提供object对象及自定义方法名
    DataObject providerDatabaseObject();
}
//4.在ApplicationComponent 的 @Component 注解值中增加 dependencies，指定依赖的 Component：
@Singleton
@Component(modules = {HttpModule.class},dependencies = {SecondComponent.class})
public interface MyComponent {
    void injectMainActivity(Dagger2Activity dagger2Activity);
    void injectSecActivity(Dagger2Activity2 dagger2Activity2);
}
//5.application中新增指定的实现类
public class MyApplication extends Application {
    private MyComponent myComponent;
    private static MyApplication context;
    @Override
    public void onCreate() {
        super.onCreate();
        context = this;
         myComponent = DaggerMyComponent.builder()
                .httpModule(new HttpModule())
                 //指定 PresenterComponent 的实现类
                .secondComponent(DaggerSecondComponent.create())
                .build();
    }
    public static MyApplication getInstance() {
        return context;
    }
    public MyComponent getMyComponent() {
        return myComponent;
    }
}
//6.使用
public class Dagger2Activity extends AppCompatActivity {
    @Inject
    HttpObject httpObject;
    @Inject
    DataObject dataObject;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dagger);
        MyApplication.getInstance().getMyComponent().injectMainActivity(this);
        Log.e("tyl",httpObject.hashCode()+"");
        Log.e("tyl2",dataObject.getData()+"");
    }
}
```

5. @Scope

假如现在有个需求，想让 DataObject 也变成个单例对象，按照我们之前的做法，给 PresenterModule 中的 Provides 方法和 PresenterComponent 加上 @Singleton 之后，会发现编译报错了：

```
com.demo.dagger.component.ApplicationComponent also has @Singleton
```

它说 ApplicationComponent 中已经有 @Singleton 了。显然，Dagger2 要求 @Singleton 不能用在多个组件上。

```
使用 @Scope 的原则：

1.多个 Component 上面的 Scope 不能相同
2.没有 Scope 的 Component 不能依赖有 Scope 的组件
3.使用作用域注解的模块只能在带有相同作用域注解的组件中使用
4.使用构造方法注入（通过 @Inject）时，应在类中添加作用域注解；使用 Dagger 模块时，应在 @Provides 方法中添加作用域注解
```

@Scope 注解表示作用域，被其修饰的类或提供对象的方法会被做成单例，所以我们可以用 @Scope 自定义一个作用域注解：

```
@Scope
@Retention(RetentionPolicy.RUNTIME)
public @interface MyScope {}//自定义名字
```

实际上它和 @Singleton 一样只是名字不同：

```
@Scope
@Documented
@Retention(RUNTIME)
public @interface Singleton {}
```

以下示例编译报错，未查明原因

Component使用

```
@MyScope
@Component(modules = {DataObject.class})
public interface SecondComponent {
    //使用依赖关系，就不再使用这种语法
//    void inject(MainActivity activity);
    DataObject providerDatabaseObject();
}
```

moudle使用

```
@Module
@MyScope
public class DataModule {
    @MyScope
    @Provides
    public DataObject providerDatabaseObject(){
        //........
        return new DataObject();
    }
}
```

5.ViewModle

什么是ViewModel

它是介于View(视图)和Model(数据模型)之间的一个东西。它起到了桥梁的作用，使视图和数据既能分离，也能保持通信。即ViewModel
是以生命周期的方式存储与管理UI相关数据。
用于分离 UI 逻辑与 UI 数据。在发生 Configuration Changes 时，它不会被销毁。在界面重建后，方便开发者呈现界面销毁前
的 UI 状态。

ViewModel的特性

1.数据持久化
2.异步回调问题，不会造成内存泄漏
3.隔离Model层与View层
4.Fragments间共享数据

ViewModel出现所解决的问题：
1.Activity的具体生命周期管理，每一次反转都是去重新创建了一个Activity
2.Activity的使用，正常onDestory都会清空数据，每次onCreate都会重新加载进来
（手机旋转时，activity会销毁后新建1个activity,viewmodel相当于在做了数据恢复，onCreate数据恢复）

业务目的：
是为了达成数据持久化，去规避Activity在设计当中对于旋转处理造成的实例Activity重置下的数据保存问题
1.ViewModel = Boudle使用！
2.只不过是作了上层的封装！
3.利用了jetpack组件套当中的应用！
4. 在上一个被销毁的Activity与新建的这个Activity之间建立一个Boudle数据传递！

1.依赖库

```
implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
```

2.使用

```
//1.model类
public class DataModel extends ViewModel {
    private int age=0;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
//2.xml
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
    </data>
```

```
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:id="@+id/tv_display"
            android:gravity="center"
            android:textColor="@color/black"
            android:layout_width="match_parent"
            android:layout_height="40dp"/>
            <Button
                android:id="@+id/bn_change"
                android:onClick="change"
                android:text="change"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"/>
        <Button
            android:id="@+id/bn_rote"
            android:onClick="rote"
            android:text="rote"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>
    </LinearLayout>
</layout>
//3.使用类
public class ViewModelActivity extends AppCompatActivity {
    private ActivityViewModelBinding binding;
    private DataModel dataModel;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = DataBindingUtil.setContentView(this, R.layout.activity_view_model);
        //不能直接 new DataModel()，这种方式无法自动恢复数据
        dataModel = new ViewModelProvider(this).get(DataModel.class);
        Log.e("tyl","age="+ dataModel.getAge());
    }

    public void change(View view) {
       dataModel.setAge(dataModel.getAge()+1);
       binding.tvDisplay.setText(dataModel.getAge()+"");
    }

    //旋转后activity销毁后重新onCreate，但是会恢复age在销毁前的值
    private Boolean isLand=false;
    public void rote(View view) {
        setScreenOrientation(isLand);
    }
    public void setScreenOrientation(boolean isLandScape){
        if (isLandScape){
            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);// 横屏
        }else {
            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);// 横屏
        }
        isLand=!isLand;
    }
}
```

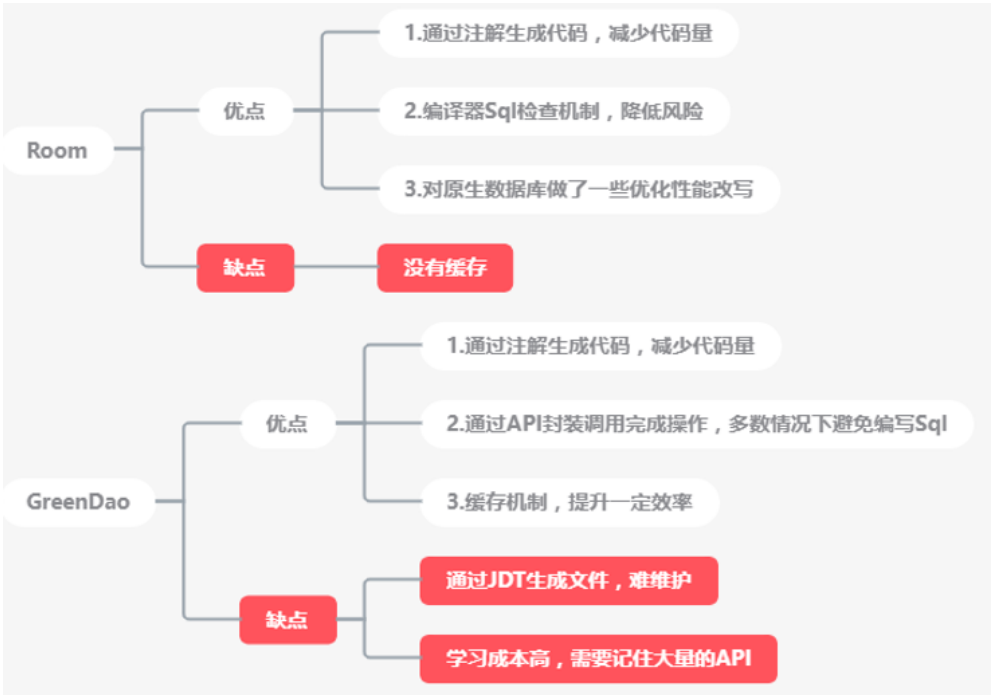
6.Room

Room是什么?

Room 是一个轻量级 orm 数据库，本质上是一个SQLite抽象层，但是使用起来会更加简单，类似于Retrofit库。Room在开发阶段通过注解
的方式标记相关功能，编译时自动生成响应的 impl 实现类

ORM映射关系设计与详解

对象关系映射(英语:Object Relational Mapping，简称ORM，或0/RM，或0/R mapping),是一种程序设计技术，用于实现面向对象编程
语言里不同类型系统的数据之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的“虚拟对象数据库”。如今已有很多免费和
付费的ORM产品，而有些程序员更倾向于创建自己的ORM工具。



Room使用:

```
implementation "androidx.room:room-runtime:2.2.3"
annotationProcessor "androidx.room:room-compiler:2.2.3"
```

官网的文档也指出，我们需要创建这三个文件，才能正式的使用Room:
1.Entity：普通Java Bean类
2.DAO：定义一个接口类
3.Room Database：定义一个抽象类，并继承Room Database

1.定义Entity:

```
/**
 * 1.Entity类需要使用@Entity注解标注，并且其中的tableName即数据表名，可以改为自己想取的名字
 * 2.构造函数：构造函数可以允许多个，但只允许只有一个不加@Ignore注解，其它的都得加上@Ignore注解，实际运行发现，若多个构造函数不
```

```
 * 加@Ignore，将无法通过编译。至于多个构造函数的存在是因为CRUD的特性决定的，例如删除，只需要id主键就行了。因此提供了多个构造
 * 函数
 * 3.Getter和Setter方法：这些方法是必须要的，否则无法对对象属性进行读取或修改
 */

//Entity类需要使用@Entity注解标注，并且其中的tableName即数据表名，可以改为自己想取的名字
@Entity(tableName = "StudentDao")
public class StudentEntity {
    /**
     * 使用@PrimaryKey声明为主键，并且允许自动生成
     * 使用@ColumnInfo表明这个属性是表中的一列列名，并可以指明列的名称
     */
    @PrimaryKey(autoGenerate = true)//autoGenerate = true 使得主键的值自动生成
    @ColumnInfo(name = "id")
    private int id;
    @ColumnInfo
    private String name;
    @ColumnInfo
    private String pwd;
    @ColumnInfo
    private int age;


    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPwd() {
        return pwd;
    }

    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

2.定义 Dao：

DAO类主要是提供对数据库的访问方法，是一个接口类。通过将SQL语句与方法结合的方式，降低结构的复杂程度。

```
/**
 * DAO层接口,需要添加@Dao注解声明
 * 所有的操作都以主键为依托进行
 */
@Dao
public interface  StudentDao {

    /**
     * 查询所有的数据，返回List集合
     * @return
     */
    @Query("Select * from StudentDao")
    List<StudentEntity> getAllStudentList();

    /**
     * 传递参数的集合，注意 Room通过参数名称进行匹配，若不匹配，则编译出现错误
     * @param personId
     * @return
     */
    @Query("select * from StudentDao where id in (:personId)")
    List<StudentEntity> getStudentById(int[] personId);

    /**
     * 返回一定条件约束下的数据，注意参数在查询语句中的写法
     * @param minAge
     * @param maxAge
     */
    @Query("select * from StudentDao where age between :minAge and :maxAge")
    List<StudentEntity> getStudentByAgeRange(int minAge, int maxAge);

    /**
     * 插入数据，onConflict = OnConflictStrategy.REPLACE表明若存在主键相同的情况则直接覆盖
     * 返回的long表示的是插入项新的id
     */
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    long insertStudent(StudentEntity studentEntity);

    /**
     * 更新数据，这意味着可以指定id然后传入新的person对象进行更新
     * 返回的long表示更新的行数
     */
    @Update
    int updatePerson(StudentEntity studentEntity);


    /**
     * 删除数据，根据传入实体的主键进行数据的删除。
     * 也可以返回long型数据，表明从数据库中删除的行数
     */
    @Delete
    int deletePerson(StudentEntity studentEntity);
}
```

3.Room Database类构建

得益于Room的良好封装，这个类的构建步骤只有两步，非常的简单。

- 创建一个StudentDatabase抽象类并继承自RoomDatabase类
- 声明一个StudentDao()的抽象方法并返回StudentDao的对象引用

```
/**
 * 指明是需要从那个class文件中创建数据库，并必须指明版本号
 */
@Database(entities = {StudentEntity.class}, version = 1)
public abstract class StudentDataBase extends RoomDatabase {
    public abstract StudentDao studentDao();
}
```

这样就完成了对RoomDatabase的构建，这里要注意引入@Database注解，它表明需要与哪一个Entity类产生联系，生成对应的数据库表。并且version版本号是一个必填字段，在这里，我们定义为1即可。

4.在Activity中操作Room

这里要特别注意的是：**虽然Room可以通过调用.allowMainThreadQueries()方法允许在主线程中进行查询，但是实际操作中禁止在主线程操作数据库，避免出现ANR问题。需要开启子线程进行数据的操作！**

```
public class RoomActivity extends AppCompatActivity {
    private StudentDataBase studentDataBase;
    private StudentDao studentDao;
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_room);
        studentDataBase = Room.databaseBuilder(
                getApplicationContext(),
                StudentDataBase.class,
                "StudentDao"
            ).build();
        studentDao = studentDataBase.studentDao();
    }

    private int defaultClick=1;
    public void addData(View view) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                StudentEntity studentEntity = new StudentEntity();
                studentEntity.setName("name:"+defaultClick);
                studentEntity.setAge(defaultClick);
                studentEntity.setPwd("pwd:"+defaultClick);
                long l = studentDao.insertStudent(studentEntity);
                defaultClick++;
                if (l!=0){
                    Log.e("tyl","添加成功");
                }else {
                    Log.e("tyl","添加失败");
                }
            }
        }).start();
```

```
    }

    public void deleteData(View view) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                List<StudentEntity> allStudentList = studentDao.getAllStudentList();
                if (allStudentList!=null&&allStudentList.size()!=0){
                    studentDao.deletePerson(allStudentList.get(allStudentList.size()-1));
                }
            }
        }).start();
    }

    public void changeData(View view) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                List<StudentEntity> allStudentList = studentDao.getAllStudentList();
                if (allStudentList!=null&&allStudentList.size()!=0){
                    StudentEntity studentEntity = allStudentList.get(allStudentList.size() - 1);
                    studentEntity.setName("upData");
                    studentEntity.setAge(0);
                    studentDao.updatePerson(studentEntity);
                }
            }
        }).start();
    }

    public void queryData(View view) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                List<StudentEntity> allStudentList = studentDao.getAllStudentList();
                if (allStudentList!=null&&allStudentList.size()!=0){
                    for (int i = 0; i <allStudentList.size() ; i++) {
                        StudentEntity studentEntity = allStudentList.get(i);
                        Log.e("tyl","id:"+studentEntity.getId());
                        Log.e("tyl","name:"+studentEntity.getName());
                        Log.e("tyl","pwd:"+studentEntity.getPwd());
                        Log.e("tyl","age:"+studentEntity.getAge());
                    }
                }
            }
        }).start();
    }
}
```

7.Hilt

HILT 是什么

Hilt 提供了一种合并 Dagger 的标准方法 依赖项注入到 Android 应用程序中。

Hilt 的目标是:
1.简化 Android 应用的 Dagger 相关基础架构。
2.要创建一组标准的组件和示波器以简化设置， 可读性/理解力，以及应用程序之间的代码共享。
3.提供一种简单的方法来为各种生成提供不同的绑定 类型（例如测试、调试或发布）。

Hilt 通过代码为您生成 Dagger 设置代码。这带走了 大多数使用 Dagger 的样板，实际上只留下了 定义如何创建对象以及注入对象的位
置。Hilt 将生成 Dagger 组件和用于自动注入 Android 类的代码

添加依赖项

```
//1. 配置Hilt的插件路径(project->build.gradle)
buildscript {
    ...
    dependencies {
        ...
        classpath 'com.google.dagger:hilt-android-gradle-plugin:2.28-alpha'
    }
}
或
buildscript {//需要放在plugins之前
    dependencies {
        classpath 'com.google.dagger:hilt-android-gradle-plugin:2.28-alpha'
    }
}
plugins {
    id 'com.android.application' version '7.1.3' apply false
    id 'com.android.library' version '7.1.3' apply false

}
//2. 引入Hilt的插件(app->build.gradle )
apply plugin: 'com.android.application'
apply plugin: 'dagger.hilt.android.plugin'
或
plugins {
    id 'com.android.application'
    id 'dagger.hilt.android.plugin'
}
//3. 添加Hilt的依赖库及java8
android{
  ...
   compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
dependencies {
    ...
    implementation "com.google.dagger:hilt-android:2.28-alpha"
    annotationProcessor "com.google.dagger:hilt-android-compiler:2.28-alpha"
```

```
}
```

Hilt 目前支持以下 Android 类：

```
 1.Application
 2.Activity
 3.Fragment
 4.View
 5.Service
 6.BroadcastReceiver
```

最简单的使用

```
1.提供一个对象
  public class HttpObject {
}
2.编写Module
@InstallIn(ActivityComponent.class)
@Module
public class HttpModule {
    @Provides
    public HttpObject getHttpObject(){
        return new HttpObject();
    }
}
3.注入到Activity
@AndroidEntryPoint
public class MainActivity extends AppCompatActivity {
    @Inject
    HttpObject httpObject;
    @Inject
    HttpObject httpObject2;

4.Application中注册
  @HiltAndroidApp
public class MyApplication extends Application {
}
5.全局单例使用
@InstallIn(ApplicationComponent.class)
@Module
public class HttpModule {

    @Provides
    @Singleton
    public HttpObject getHttpObject(){
        return new HttpObject();
    }
}
注意：Hilt注入的字段是不可以声明成private
```

-为接口注入实例

```
//1
public interface AnalyticsService {
  void analyticsMethods();
}
```

```
//2
public class AnalyticsServiceImpl implements AnalyticsService {
  ...
  @Inject
  AnalyticsServiceImpl(...) {
    ...
  }
}
//3
@Module
@InstallIn(ActivityComponent.class)
public abstract class AnalyticsModule {

  @Binds
  public abstract AnalyticsService bindAnalyticsService(
    AnalyticsServiceImpl analyticsServiceImpl
  );
}
```

HILT 常用的注解的含义

```
@HiltAndroidApp
@HiltAndroidApp 将会触发 Hilt 的代码生成，作为程序依赖项容器的基类
生成的 Hilt 依附于 Application 的生命周期，他是 App 的父组件，提供访问其他组件的依赖
在 Application 中配置好后，就可以使用 Hilt 提供的组件了；组件包含 Application，Activity，Fragment，View，Service 等。
@HiltAndroidApp
创建一个依赖容器，该容器遵循 Android 的生命周期类，目前支持的类型是: Activity, Fragment, View, Service, BroadcastReceiver.
@Inject
使用 @Inject 来告诉 Hilt 如何提供该类的实例，常用于构造方法，非私有字段，方法中。
Hilt 有关如何提供不同类型的实例信息也称之为绑定
@Module
module 是用来提供一些无法用 构造@Inject 的依赖，如第三方库，接口，build 模式的构造等。
使用 @Module 注解的类，需要使用 @InstallIn 注解指定 module 的范围
增加了 @Module 注解的类，其实代表的就是一个模块，并通过指定的组件来告诉在那个容器中可以使用绑定安装。
@InstallIn
使用 @Module 注入的类，需要使用 @InstallIn 注解指定 module 的范围。
例如使用 @InstallIn(ActivityComponent::class) 注解的 module 会绑定到 activity 的生命周期上。
@Provides
常用于被 @Module 注解标记类的内部方法上。并提供依赖项对象。
@EntryPoint
Hilt 支持最常见的 Android 类 Application、Activity、Fragment、View、Service、BroadcastReceiver 等等，但是您可能需要
在Hilt 不支持的类中执行依赖注入，在这种情况下可以使用 @EntryPoint 注解进行创建，Hilt 会提供相应的依赖。
```

Hilt 组件	注入器面向的对象
`ApplicationComponent`	`Application`
`ActivityRetainedComponent`	`ViewModel`
`ActivityComponent`	`Activity`
`FragmentComponent`	`Fragment`
`ViewComponent`	`View`
`ViewWithFragmentComponent`	带有 `@WithFragmentBindings` 注释的 `View`
`ServiceComponent`	`Service`

★ **注意**：Hilt 不会为广播接收器生成组件，因为 Hilt 直接从 **`ApplicationComponent`** 注入广播接收器。

生成的组件	创建时机	销毁时机
`ApplicationComponent`	`Application#onCreate()`	`Application#onDestroy()`
`ActivityRetainedComponent`	`Activity#onCreate()`	`Activity#onDestroy()`
`ActivityComponent`	`Activity#onCreate()`	`Activity#onDestroy()`
`FragmentComponent`	`Fragment#onAttach()`	`Fragment#onDestroy()`
`ViewComponent`	`View#super()`	视图销毁时
`ViewWithFragmentComponent`	`View#super()`	视图销毁时
`ServiceComponent`	`Service#onCreate()`	`Service#onDestroy()`

★ **注意**：**`ActivityRetainedComponent`** 在配置更改后仍然存在，因此它在第一次调用 **`Activity#onCreate()`** 时创建，在最后一次调用 **`Activity#onDestroy()`** 时销毁。

Android 类	生成的组件	作用域
`Application`	`ApplicationComponent`	`@Singleton`
`View Model`	`ActivityRetainedComponent`	`@ActivityRetainedScope`
`Activity`	`ActivityComponent`	`@ActivityScoped`
`Fragment`	`FragmentComponent`	`@FragmentScoped`
`View`	`ViewComponent`	`@ViewScoped`
带有 `@WithFragmentBindings` 注释的 `View`	`ViewWithFragmentComponent`	`@ViewScoped`
`Service`	`ServiceComponent`	`@ServiceScoped`

8.Paging

Paging概述

Paging 库可帮助您加载和显示来自本地存储或网络中更大的数据集中的数据页面。此方法可让您的应用更高效地利用网络带宽和系统资源;

使用 Paging 库的优势

- 分页数据的内存中缓存。该功能有助于确保您的应用在处理分页数据时高效使用系统资源。
- 内置的请求去重功能，可确保您的应用高效利用网络带宽和系统资源。
- 可配置的RecyclerView适配器，会在用户滚动到已加载数据的末尾时自动请求数据。
- 对 Kotlin 协程和数据流以及LiveData和 RxJava 的一流支持。
- 内置对错误处理功能的支持，包括刷新和重试功能。

```
  implementation 'androidx.paging:paging-runtime:2.1.0'
```

//没有认真研究，写真实项目时再认真学习下，以下是简单的使用案例，真实项目可能不一样

```
/**
 * PagedList: 数据源获取的数据最终靠PagedList来承载。
 * 对于PagedList,我们可以这样来理解，它就是一页数据的集合。
 * 每请求一页，就是新的一个PagedList对象。
 */
public class StudentViewModel extends ViewModel {

    // 看源码: @1 listLiveData 数据怎么来的
    private final LiveData<PagedList<Student>> listLiveData;

    public StudentViewModel() {
        StudentDataSourceFactory factory = new StudentDataSourceFactory();

        // 初始化 ViewModel,如何生产一页数据出来给我!
        this.listLiveData = new LivePagedListBuilder<Integer, Student>(factory, Flag.SIZE)
                .setBoundaryCallback(new MyBoundaryCallback())
                .build();
    }
    // TODO 暴露数据出去
    public LiveData<PagedList<Student>> getListLiveData() {
        return listLiveData;
    }
}
```

```
public class RecyclerPagingAdapter extends PagedListAdapter<Student,
RecyclerPagingAdapter.MyRecyclerViewHolder> {
    // TODO 比较的行为
    private static DiffUtil.ItemCallback<Student> DIFF_STUDNET = new
            DiffUtil.ItemCallback<Student>() {
                // 一般是比较 唯一性的内容， ID
                @Override
                public boolean areItemsTheSame(@NonNull Student oldItem, @NonNull Student
newItem) {
```

```
                    return oldItem.getId().equals(newItem.getId());
                }

                // 对象本身的比较
                @Override
                public boolean areContentsTheSame(@NonNull Student oldItem, @NonNull Student
newItem) {
                    return oldItem.equals(newItem);
                }
            };

    protected RecyclerPagingAdapter() {
        super(DIFF_STUDNET);
    }

    @NonNull
    @Override
    public MyRecyclerViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.item, null);
        return new MyRecyclerViewHolder(view);
    }

    @Override
    public void onBindViewHolder(@NonNull MyRecyclerViewHolder holder, int position) {
        Student student = getItem(position);

        // item view 出来了， 分页库还在加载数据中，我就显示 Id加载中
        if (null == student) {
            holder.tvId.setText("Id加载中");
            holder.tvName.setText("Name加载中");
            holder.tvSex.setText("Sex加载中");
        } else {
            holder.tvId.setText(student.getId());
            holder.tvName.setText(student.getName());
            holder.tvSex.setText(student.getSex());
        }
    }

    // Item 优化的 ViewHolder
    public static class MyRecyclerViewHolder extends RecyclerView.ViewHolder {

        TextView tvId;
        TextView tvName;
        TextView tvSex;

        public MyRecyclerViewHolder(View itemView) {
            super(itemView);
            tvId = itemView.findViewById(R.id.tv_id); // ID
            tvName = itemView.findViewById(R.id.tv_name); // 名称
            tvSex = itemView.findViewById(R.id.tv_sex); // 性别
        }
    }

}
```

```
public class MainActivity extends AppCompatActivity {
```

```
    private RecyclerView recyclerView;
    RecyclerPagingAdapter recyclerPagingAdapter;
    StudentViewModel viewModel;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        recyclerView =  findViewById(R.id.recycle_view);
        recyclerPagingAdapter = new RecyclerPagingAdapter();

        // 最新版本初始化 viewModel
        viewModel = new ViewModelProvider(this, new ViewModelProvider.NewInstanceFactory())
                .get(StudentViewModel.class);

        // LiveData 观察者 感应更新
        viewModel.getListLiveData().observe(this, new Observer<PagedList<Student>>() {
            @Override
            public void onChanged(PagedList<Student> students) {
                // 再这里更新适配器数据
                recyclerPagingAdapter.submitList(students);
            }
        });
        recyclerView.setAdapter(recyclerPagingAdapter);
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
    }
}
```

9.WorkManager

WorkManager概述

> WorkManager 是适合用于持久性工作的推荐解决方案。如果工作始终要通过应用重启和系统重新启动来调度，便是持久性的工作。由于大多
> 数后台处理操作都是通过持久性工作完成的，因此 WorkManager 是适用于后台处理操作的主要推荐 API。
> 不依赖于当前用户进程:当前用户进行killed，任务能够继续执行;

> WorkManager 适用于需要可靠运行的工作，即使用户导航离开屏幕、退出应用或重启设备也不影响工作的执行。例如：
>
> 向后端服务发送日志或分析数据。
> 定期将应用数据与服务器同步。
> WorkManager 不适用于那些可在应用进程结束时安全终止的进程内后台工作。它也并非对所有需要立即执行的工作都适用的通用解决方案。

3个主要的核心类

Worker

我们要执行的具体任务是需要放在继承了 Worker 的类里面的，所以 Worker 里面定义了做什么，继承了 Worker 后我们需要重写 doWork 方法，具体要执行的任务应该放置在重写的 doWork方法中。

WorkRequest

上面的 Worker 只是定义了我们要在后台做什么，但是怎么做，做几次之类的需要 WorkRequest 来定义。所以一个 Worker 需要经过 WorkRequest 的包装。

- → OneTimeWorkRequest：定义一次性任务
- → PeriodicWorkRequest：定义可重复执行任务

WorkManager

如果说 Worker 定义了做什么任务，WorkRequest 定义了如何做，那 WorkManager 就是对 WorkRequest 的管理。

基础使用

```
 implementation "androidx.work:work-runtime:$work_version"
```

work类定义 (extends Worker)

```
public class DemoWork extends Worker {
    public static final String TAG = DemoWork.class.getSimpleName();
    private Context context;
    private WorkerParameters workerParams;

    public DemoWork(@NonNull Context context, @NonNull WorkerParameters workerParams) {
        super(context, workerParams);
        this.context = context;
        this.workerParams = workerParams;
    }
    @NonNull
    @Override
    public Result doWork() {
        Log.e(TAG, "doWork----------**-----------: 后台任务执行了");
```

```
        // 接收Activity传递过来的数据
        final String dataString = workerParams.getInputData().getString("data");
        Log.e(TAG, "doWork: 接收Activity传递过来的数据:" + dataString);
        // 反馈数据 给 Activity
        // 把任务中的数据回传到activity中
//        Data outputData = new Data.Builder().putString("data", "执行完任务将结果和数据回传给
Activity").build();
        @SuppressLint("RestrictedApi")
        Result.Success success = new Result.Success();
        // return new Result.Failure(); // 本地执行 doWork 任务时 失败
        // return new Result.Retry(); // 本地执行 doWork 任务时 重试
        // return new Result.Success(); // 本地执行 doWork 任务时 成功 执行任务完毕
        return success;
    }
}
```

单次执行及延迟调用的示例

```
  //工作通过把WorkRequest传入WorkManager中进行定义。使WorkManager可以调度任何工作;
  OneTimeWorkRequest oneTimeWorkRequest =
                new OneTimeWorkRequest.Builder(DemoWork.class)
                        .setInitialDelay(10,TimeUnit.SECONDS)
                        .build();

WorkManager.getInstance(this).enqueue(oneTimeWorkRequest);
```

WorkRequest 对象包含 WorkManager 调度和运行工作所需的所有信息

WorkRequest 本身是抽象基类。该类有两个派生实现，可用于创建 OneTimeWorkRequest和 PeriodicWorkRequest 请求。顾名思义，OneTimeWorkRequest 适用于调度非重复性工作，而PeriodicWorkRequest则更适合调度以一定间隔重复执行的工作。

调度一次性工作

```
WorkRequest myWorkRequest = OneTimeWorkRequest.from(MyWork.class);
//对于更复杂的工作，可以使用构建器：
WorkRequest myWorkRequest =
   new OneTimeWorkRequest.Builder(MyWork.class)
       //添加自定义规则
       .build();
```

调度定期工作

```
有时可能需要定期运行某些工作。例如，您可能要定期备份数据、定期下载应用中的新鲜内容或者定期上传日志到服务
器。

使用 PeriodicWorkRequest 创建定期执行的 WorkRequest 对象的方法如下：
PeriodicWorkRequest saveRequest =
       new PeriodicWorkRequest.Builder(SaveImageToFileWorker.class, 1, TimeUnit.HOURS)
           // Constraints
           .build();
//在此示例中，工作的运行时间间隔定为一小时
```

```
//执行加急工作 (从 WorkManager 2.7 开始支持)
.setExpedited(OutOfQuotaPolicy.RUN_AS_NON_EXPEDITED_WORK_REQUEST)
//延迟工作*
.setInitialDelay(10, TimeUnit.MINUTES)
```

工作约束

为了让工作在指定的环境下运行，我们可以给WorkRequest添加约束条件，常见的约束条件如下所示。 - **NetworkType**：约束运行工作
所需的网络类型，例如 Wi-Fi (UNMETERED)。 - **BatteryNotLow** ：如果设置为 true，那么当设备处于“电量不足模式”时，工作不
会运行。 - **RequiresCharging**：如果设置为 true，那么工作只能在设备充电时运行。 - **DeviceIdle**：如果设置为 true，则
要求用户的设备必须处于空闲状态才能运行工作。 - **StorageNotLow**：如果设置为 true，那么当用户设备上的存储空间不足时，工作
不会运行。

例如，以下代码会构建了一个工作请求，该工作请求仅在用户设备正在充电且连接到 Wi-Fi 网络时才会运行。

```
Constraints constraints = new Constraints.Builder()
       .setRequiredNetworkType(NetworkType.UNMETERED)
       .setRequiresCharging(true)
       .build();

WorkRequest myWorkRequest =
       new OneTimeWorkRequest.Builder(MyWork.class)
               .setConstraints(constraints)
               .build();
```

10.Navigation

Navigation概述

Navigation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。用于处理Fragment事务，使fragment之间可以自由切换和跳转，
同时还包括导航界面模式（例如抽屉式导航栏和底部导航），可以降低用户工作量;

Navigation组成(核心三件套)

1.导航图：在一个集中位置包含所有导航相关信息的 XML 资源。包含用户可以跳转的所有路径，对Navigation来说就像是地图。
2.NavHost：用来显示导航图中目标所要展示的内容。
3.NavController：在 NavHost 中管理应用导航的对象。负责NavHost里内容的改变
如果要在应用中导航，则通过NavController，沿导航图中的特定路径导航至特定目标，或直接导航至特定目标。NavController 就可以
在NavHost里进行跳转。

```
 // 指定Navigation的版本
     def nav_version = "2.5.3"

     // 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"
```

导航图使用方法

1.创建导航图

导航图是一种资源文件，其中包含Navigation所有目的地和操作。会显示应用的所有导航路径。

1.1. 具体操作

1.在“Project”窗口中，点击 res 目录，然后依次选择 New > Android Resource File。此时系统会显示 New Resource File 对话框。
2.在 File name 字段中输入Navigation的名称，例如“graph”。
3.从 Resource type 下拉列表中选择 Navigation，然后点击 OK。
这样就完成了空白导航图的创建，这时来到res文档下就会看到navigation文件夹还有你创建的导航图

2.向Activity添加NavHost

分成两种方法：

```
1. 通过 XML 添加
2. 使用布局编辑器添加(暂不介绍)
```

2.1. 通过 XML 添加

在activity中加入如下代码

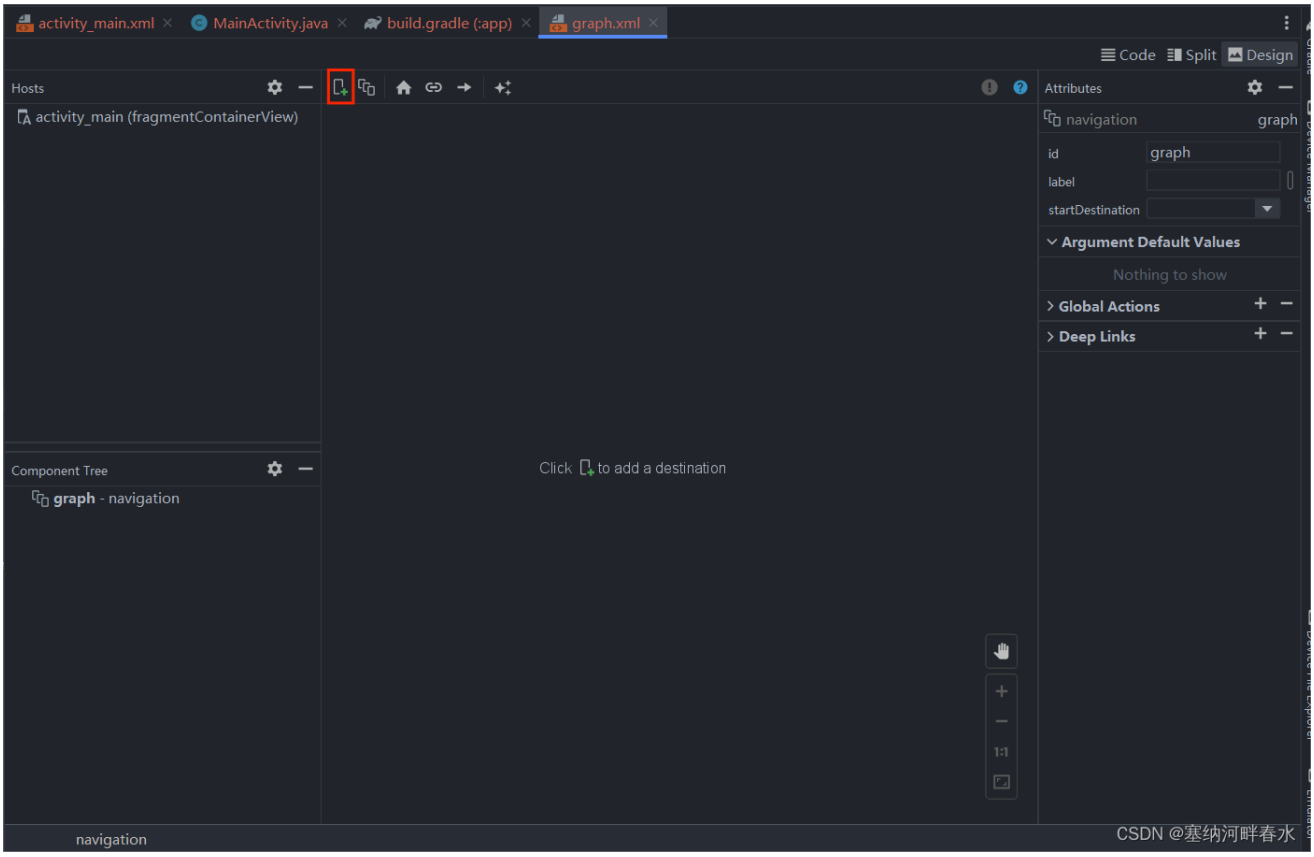
```
    <!-- 这里navGraph的值要改为自己导航图的名字 -->
    <!-- app.defaultNavHost="true"表示回退栈由fragment管理 -->
    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragmentContainerView"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="409dp"
        android:layout_height="729dp"
        app:defaultNavHost="true"
        app:navGraph="@navigation/navigation_map"
        />
```

3.在导航图中创建目的地

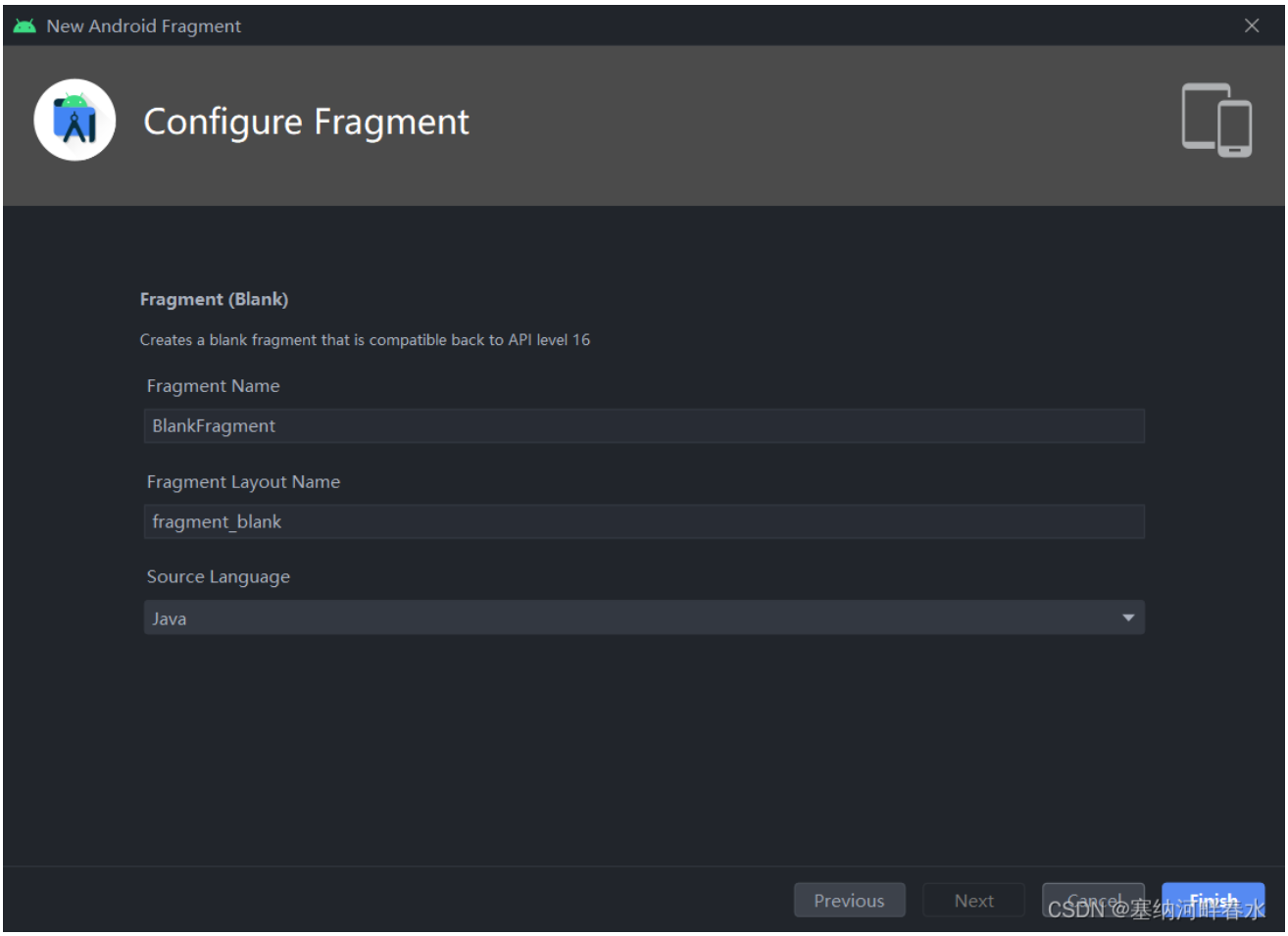
目的地相当于是导航图中的一个个地点，展示各种界面内容

3.1. 具体操作

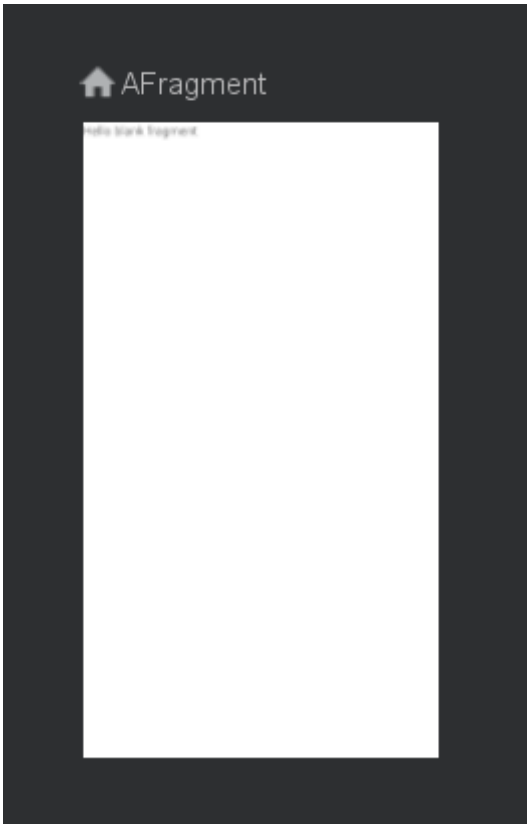
1.双击导航图，点击右上角的Design，来到下图的 Navigation Editor 界面，点击图中标红图标，然后点击 Create new destination。



2.在接下来的对话框中，创建 Fragment，Android Studio会按照如下配置创建BlankFragment类和fragment_layout布局（fragment_layout中默认采用FrameLayout的布局，可以改成ConstraintLayout）



回到 Navigation Editor 界面就可以看到导航图中已经有了一个目的地



3.连接目的地

操作会将一个目的地连接到另一个目的地，即一个界面是否可以跳转到另一个界面

为了演示，我在上面的基础上再建了一个BFragment

3.1. 具体操作

```
1.在系统生成的AFragment类里的onCreateView方法里进行改动
2.通过 Navigation.findNavController(view) 方法得到对应 NavController
3.使用 NavController 里的 navigate(int) 方法进行导航，该方法的参数为两目的地之间连接的id或者要导向的目的地id

public class AFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View inflate = inflater.inflate(R.layout.fragment_a, container, false);
        inflate.findViewById(R.id.tv_text).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // action_AFragment_to_BFragment为两目的地之间连接的id或者要导航向的目的地id，这个id在navigation_map
                //的右上角自动生成的
                Navigation.findNavController(v).navigate(R.id.action_AFragment_to_BFragment);
            }
        });
        return inflate;
    }
}
```

3.2Navigation的返回

Navigation支持多个返回堆栈可让用户在各个页面之间自由切换，同时不会在任何页面中丢失所处的位置，不需要导航图中有对应的连接就可
以进行返回操作

具体操作:
1.在系统生成的BlankFragment类里的onCreateView方法里进行改动
2.通过 Navigation.findNavController(view) 方法得到对应 NavController
3.使用 NavController 里的 popBackStack() 方法即可完成返回

```
public class BFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View inflate = inflater.inflate(R.layout.fragment_b, container, false);
        inflate.findViewById(R.id.tvback).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                //返回
                Navigation.findNavController(v).popBackStack();
            }
        });
        return inflate;
    }
}
```

4.在目的地间传递数据(不研究导航地图的传参了，麻烦无用)

```
Bundle bundle=new Bundle();
bundle.putString("name", "tyl");
Navigation.findNavController(v).navigate(R.id.action_AFragment_to_BFragment,bundle);
```

接收方通过 getArguments() 方法得到 Bundle 对象并可以使用里面的内容

```
String name = getArguments().getString("name");
```

//BottomNavigationView看了下可扩展性差，不适合正式项目使用，直接不采用这个框架