

本文首发于微信公众号「后厂技术官」

前言

在上一篇文章中，我们基本了解了什么是Android Jetpack，这一篇文章来介绍Android Jetpack架构组件的Lifecycle，Lifecycle用于帮助开发者管理Activity和Fragment 的生命周期，由于Lifecycle是LiveData和ViewModel的基础，所以需要先学习它。

1.为什么需要Lifecycle

在应用开发中，处理Activity或者Fragment组件的生命周期相关代码是不可避免的，官方文档中举了一个例子，这里简化一下，在Activity中写一个监听，在Activity的不同生命周期方法中调用这个监听。

```
● ● JAVA

public class MainActivity extends AppCompatActivity {
    private MyListener myListener;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myListener = new MyListener(MainActivity.this);
    }

    @Override
    protected void onStart() {
        super.onStart();
        myListener.start();
    }

    @Override
    protected void onStop() {
        super.onStop();
        myListener.stop();
    }
}

class MyListener {
```

```
public MyListener(Context context) {  
    ...  
}  
void start() {  
    ...  
}  
void stop() {  
    ...  
}  
}
```

再举个MVP中常见的情况，如下所示。

```
● ● JAVA  
public class MainActivity extends AppCompatActivity {  
    private MyPresenter myPresenter;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        myPresenter = new MyPresenter();  
    }  
  
    @Override  
    protected void onResume() {  
        super.onResume();  
        myPresenter.onResume();  
    }  
  
    @Override  
    protected void onPause() {  
        super.onPause();  
        myPresenter.onPause();  
    }  
}  
  
class MyPresenter{  
    void onResume() {  
        ...  
    }  
    void onPause() {  
        ...  
    }  
}
```

这两个例子的写法已经很普遍了，实现起来也不难，但实际开发中，可能会有多个组件在Activity的生命周期中进行回调，这样Activity的生命周期的方法中可能就需要放大量的代码，这就使得它们难以维护。

还有一个问题是，如果我们在组件中做了耗时操作（比如在onStart方法），这种写法就无法保证组件在Activity或者Fragment停止之前完成启动。

因此我们需要一个能管理Activity和Fragment的生命周期的库，这个库就是Lifecycle。

2.如何使用Lifecycle

分别来介绍下依赖Lifecycle库和Lifecycle基本用法。

2.1 依赖Lifecycle库

官网给出的依赖代码如下所示：

```
● ● JAVA
dependencies {
    def lifecycle_version = "2.0.0"

    // ViewModel and LiveData
    implementation "androidx.lifecycle:lifecycle-extensions:$lifecycle_version"
    // alternatively - just ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"
    // alternatively - just LiveData
    implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
    // alternatively - Lifecycles only (no ViewModel or LiveData). Some
    // AndroidX libraries use this lightweight import for Lifecycle
    implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle_version"

    annotationProcessor "androidx.lifecycle:lifecycle-compiler:$lifecycle_version"
    // alternately - if using Java8, use the following instead of lifecycle-compiler
    implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"

    // optional - ReactiveStreams support for LiveData
    implementation "androidx.lifecycle:lifecycle-reactivestreams:$lifecycle_version"

    // optional - Test helpers for LiveData
    testImplementation "androidx.arch.core:core-testing:$lifecycle_version"
}
```

官网用的是AndroidX，因为使用AndroidX，可能会产生一些迁移的问题，这里的举例就不使用AndroidX，而是使用lifecycleandroid.arch.lifecycle库，如下所示。

JAVA

```
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"
    // 仅仅包含LiveData
    implementation "android.arch.lifecycle:livedata:$lifecycle_version"
    // 仅仅包含Lifecycle
    implementation "android.arch.lifecycle:runtime:$lifecycle_version"

    annotationProcessor "android.arch.lifecycle:compiler:$lifecycle_version"
    // 如果用Java8，用于替代compiler
    implementation "android.arch.lifecycle:common-java8:$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"
}
```









实际上我们不需要全部把这些代码全写进build.gradle进去（当然全写进去也不会有什么错），因为Gradle默认是支持依赖传递的，不知道什么是依赖传递的看[Android Gradle（二）签名配置和依赖管理](#)这篇文章。

我们直接添加如下依赖就可以满足日常的工作，如果缺少哪个库，再去单独添加就好了。

 JAVA


```
implementation "android.arch.lifecycle:extensions:1.1.1"
```

添加这一句代码就依赖了如下的库。

- ▶  Gradle: android.arch.core:common:1.1.1@jar
- ▶  Gradle: android.arch.core:runtime:1.1.1@aar
- ▶  Gradle: android.arch.lifecycle:common:1.1.1@jar
- ▶  Gradle: android.arch.lifecycle:extensions:1.1.1@aar
- ▶  Gradle: android.arch.lifecycle:livedata:1.1.1@aar
- ▶  Gradle: android.arch.lifecycle:livedata-core:1.1.1@aar
- ▶  Gradle: android.arch.lifecycle:runtime:1.1.1@aar
- ▶  Gradle: android.arch.lifecycle:viewmodel:1.1.1@aar

2.2 Lifecycle基本用法

先不谈Activity和Fragment中如何使用，先举一个Lifecycle的简单例子。

● ● JAVA

```
public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void connectListener() {
        ...
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void disconnectListener() {
        ...
    }
}

myLifecycleOwner.getLifecycle().addObserver(new MyObserver()); //1
```

新建一个MyObserver类，它实现了LifecycleObserver接口，说明MyObserver成为了一个Lifecycle的观察者。

然后在注释1处将MyObserver添加到LifecycleOwner中。LifecycleOwner是一个接口，其内部只有一个方法getLifecycle()，getLifecycle方法用于获取Lifecycle，这样就可以将MyObserver添加到Lifecycle中，当Lifecycle的生命周期发生变化时，MyObserver就会观察到，或者说是感知到。

如果使用是Java8 ,那么可以使用DefaultLifecycleObserver来替代LifecycleObserver：

● ● JAVA

```
class MyObserver implements DefaultLifecycleObserver {
    @Override
    public void onCreate(LifecycleOwner owner) {
        ...
    }
}
```

除此之外，不要忘了在build.gradle添加 `"androidx.lifecycle:common-java8:<version>"`

3.Lifecycle应用举例

应用举例准备两个示例，一个是在Activity中使用，一个是在第一小节的MVP例子上进行改进。

3.1 Activity中使用

● ● JAVA

```
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");
    }

}
```

先实现MyObserver，对ON_CREATE和ON_RESUME事件进行监听。因为在Android Support Library 26.1.0 及其之后的版本，Activity和Fragment已经默认实现了LifecycleOwner接口，所以在注释1处可以直接使用getLifecycle方法获取Lifecycle对象，这样MyObserver就可以观察MainActivity的生命周期变化了，LifecycleOwner可以理解为被观察者，MainActivity默认实现了LifecycleOwner接口，也就是说MainActivity是被观察者。

运行程序，打印的log如下所示。

● ● JAVA

```
D/MainActivity: onResume
D/MainActivity: Lifecycle call onResume
D/MainActivity: Lifecycle call onPause
D/MainActivity: onPause
```

只要在MainActivity的onCreate方法中添加MyObserver，那么MyObserver就可以观察到MainActivity的各个生命周期的变化。

3.2 MVP中使用

改写第一节MVP的例子，先实现MyPresenter，如下所示。

● ● JAVA

```
public class MyPresenter implements IPresenter {
    private static final String TAG = "test";

    @Override
    public void onResume() {
        Log.d(TAG, "Lifecycle call onResume");
    }

    @Override
    public void onPause() {
        Log.d(TAG, "Lifecycle call onPause");
    }
}

interface IPresenter extends LifecycleObserver {

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    void onResume();

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    void onPause();
}
```

IPresenter接口继承自LifecycleObserver接口，MyPresenter又实现了IPresenter接口，这样MyPresenter成为了一个观察者。

接在在MainActivity中加入MyPresenter：

```
● ● JAVA

public class MainActivity extends AppCompatActivity {

    private static final String TAG = "test";
    private IPresenter mPresenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mPresenter = new MyPresenter();
        getLifecycle().addObserver(mPresenter);
    }

    @Override
    protected void onResume() {
        super.onResume();
        Log.d(TAG, "onResume");
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.d(TAG, "onPause");
    }

}
```

MainActivity成为了被观察者，当它的生命周期发生变化时，MyPresenter就可以观察到，这样就不需要在MainActivity的多个生命周期方法中调用MyPresenter的方法了。

打印的日志如下：

```
● ● JAVA

D/test: onResume
D/test: Lifecycle call onResume
D/test: Lifecycle call onPause
D/test: onPause
```

4. 自定义LifecycleOwner

如果想实现自定义LifecycleOwner，可以使用LifecycleRegistry，它是Lifecycle的实现类。Android Support Library 26.1.0及其之后的版本，Activity和Fragment已经默认实现了LifecycleOwner接口，因此我们可以这么写：

```
● ● JAVA

import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleRegistry;
import android.support.annotation.NonNull;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MyActivity extends AppCompatActivity {
    private LifecycleRegistry lifecycleRegistry;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        lifecycleRegistry = new LifecycleRegistry(this);
        lifecycleRegistry.markState(Lifecycle.State.CREATED);
    }

    @Override
    public void onStart() {
        super.onStart();
        lifecycleRegistry.markState(Lifecycle.State.STARTED);
    }

    @NonNull
    @Override
    public Lifecycle getLifecycle() {
        return lifecycleRegistry;
    }
}
```

通过新建LifecycleRegistry，为LifecycleRegistry设置Lifecycle的各种状态，并通过getLifecycle方法返回该LifecycleRegistry。

总结

这一篇介绍了Lifecycle的基本用法，并通过两个小例子来帮助大家消化理解，具体在项目中的使用也不难，唯一还算难点的是Lifecycle的原理，下一篇我们就来学习Lifecycle的原理。

