转载请注明链接: https://blog.csdn.net/feather_wch/article/details/88630095

> 谷歌的架构组件 Lifecycle的基本使用

Lifecycle的基本使用

版本号:2019-03-22(11:00)

简介

1、Lifecycle组件是在其他组件的生命周期状态发生改变时，产生相应行为的一种组件。

2、Lifecycle能帮助产生更好组织且更轻量级的代码，便于维护。

3、在不使用Lifecycle时，常规的生命周期处理的缺点是什么?

> 1. 常规方法是在Activity和Fragment的生命周期方法里面去实现独立组件的行为
> 2. 这种模式会导致代码组织性差
> 3. 增生更多的错误

4、Lifecycle的好处是什么?

> 1. 通过使用可感知生命周期的组件，可以将生命周期方法中关于这些组件的代码，移动到组件的内部

> 2. 通过 android.arch.lifecycle 提供的内容，可以让 组件主动调节自身的行为 ，根据 activity/fragment当前的生命周期状态 进行调整。

5、几乎所有app组件都可以和Lifecycle关联起来，这些都是由操作系统或者运行在程序中的FrameWork层代码进行支持的。

> 使用Lifecycle能减少内存泄漏和系统崩溃的可能性

Lifecycle集成

1、Lifecycle的集成只需要在app的build.gradle中配置

```
    // 引入lifecycle
    def lifecycle_version = "2.0.0"
    // ViewModel and LiveData
    implementation "androidx.lifecycle:lifecycle-extensions:$lifecycle_version"
```

Lifecycle的使用

原Activity中生命周期处理的代码

1、现在有一个功能是读取当前设备的位置。用常规的方法处理。

> 1-定位功能，start()连接到系统定位服务，stop()断开和系统定位服务的连接

```
class MyLocationListener {
    public MyLocationListener(Context context, Callback callback) {
        // ...
    }

    void start() {
        // connect to system location service
    }

    void stop() {
        // disconnect from system location service
    }
}
```

> 2-Activity中使用该功能。Activity的生命周期和定位功能紧密相连

```
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    @Override
    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, (location) -> {
            // update UI
        });
    }

    @Override
    public void onStart() {
        super.onStart();
        myLocationListener.start();
        // manage other components that need to respond
        // to the activity lifecycle
    }

    @Override
    public void onStop() {
        super.onStop();
        myLocationListener.stop();
        // manage other components that need to respond
        // to the activity lifecycle
    }
}
```

2、常规的处理方法中onStart()和onStop()等生命周期中会有大量的代码

3、还有可能会导致 `onStart()` 中的方法在 `onStop()` 方法执行后执行

> 1-例如:

```
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    @Override
    public void onStart() {
        super.onStart();
        Util.checkUserStatus(result -> {
            /**=============================================================================
             * 如果该操作有极高的延迟，导致【定位服务】的start方法居然在stop方法中执行。会产生意想不
             *=============================================================================*/
            if (result) {
                myLocationListener.start();
            }
        });
    }

    @Override
    public void onStop() {
        super.onStop();
        myLocationListener.stop();
    }
}
```

Lifecycle的大致原理

1、Lifecycle类会持有组件(Activity、Fragment)生命周期状态的信息， 并且允许其他对象能监听该状态

2、Lifecycle使用两个主要的枚举对相关的组件的生命周期状态进行追踪

> 1. Event
>   - Lifecycle事件由framework和Lifecycle类进行分发
>   - Event会映射到activity和fragment的回调事件上
> 2. State
>   - Lifecycle对象会追踪组建的当前状态

3、Events和States

> 1. States
>   1. INITIALIZED
>   2. DESTORYED
>   3. CREATED
>   4. STARTED
>   5. RESUMED
> 2. Events
>   1. ON_CREATE
>   2. ON_START

> 3. ON_RESUME
> 4. ON_PAUSE
> 5. ON_STOP
> 6. ON_DETSORY

监听组建的生命周期

1、实例：监听组件的生命周期

> 1-自定义定位服务，具有注册和解注册

```
public class MyLocation implements LifecycleObserver {

    Lifecycle mLifecycle;

    public MyLocation(Lifecycle lifecycle){
        mLifecycle = lifecycle;
        // 将自己加入到目标组件生命周期的监听列表中
        mLifecycle.addObserver(this);
    }

    // 在onCreate时调用
    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
    public void register(){
        Log.d("wch", "注册定位服务");
    }

    // 在onDetsory时调用
    @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
    public void unregister(){
        Log.d("wch", "解注册定位服务");
    }
}
```

> 2-Activity中进行创建

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

        // 创建定位服务
        MyLocation location = new MyLocation(getLifecycle());
    }
```

LifecycleOwner

1、LifecycleOwner的作用?

> 1. 一个接口
> 2. 具有方法 `getLifecycle()` ， 例如Activity就可以通过该方法，获取到Lifecycle
> 3. `getLifecycle()` 的内部返回的是 `LifecycleRegistry` ， 通过 `LifecycleRegistry` 处理不同的状态。

2、如何监控整个app的生命周期?

> 1. 使用 `ProcessLifecycleOwner`

3、Fragment、AppCompatActivity都实现了LifecycleOwner接口

4、自定义Application需要自己实现LifecycleOwner接口

5、如果下面的MyLocation的注册方法中还进行了Fragment transaction，在Activity状态保存后，会导致崩溃

> 1-MyLocation.java

```
public class MyLocation implements LifecycleObserver {

    // xxx

    // 在onCreate时调用
    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
    public void register(){
        Log.d("wch", "注册定位服务");
        // 延时操作后，进行了可能会因为Activity和Fragment生命周期不同步，出现崩溃的操作
        enable();
    }
}
```

> 2-进行保护操作，判断当前的状态

```
    public void enable() {
        enabled = true;
        if (lifecycle.getCurrentState().isAtLeast(STARTED)) {
            // 其他操作
        }
    }
```

自定义LifecycleOwner

6、要自定义LifecycleOwner需要借助LifecycleRegistry

> 就是通过 `LifecycleRegistry`的`markState()` 进行状态的切换
> 自定义Activity

```
public class MyActivity extends Activity implements LifecycleOwner {
    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;
    }
}
```

LifecycleRegistry

7、LifecycleRegistry是什么?

> 1. Lifecycle的实现类
> 2. 通过 `makeState()` 方法进行状态的切换

实践要点

1、UI控制器如Activity、Fragment都不要去获取自身的数据，应该使用 `ViewModel` ,来观察一个 `LiveData` 对象，来将数据的改变反应到仕途上。

2、要编写 数据驱动 的UI，让UI控制器负责在数据改变时去更新视图，或者将用户的行为反馈给 `ViewModel`

3、应该将数据逻辑放置到 `ViewModel` 中。

> 1. ViewModel，是 `UI控制器` 和 其他部分 之间的桥梁
> 2. 注意，ViewModel的职责不是去获取数据，比如从网络请求中获取数据。
> 3. ViewModel负责的是通知合适的组件去获取数据，然后将结果 反馈给`UI控制器`

4、使用Data Binding在视图和Activity/Fragment之间维护一个清晰的接口

> 1. 能让视图拥有自说明的能力

2. 让Activity/Fragment中用于更新数据的代码尽可能的少
3. 如果想要在Java中拥有这些效果，建议使用 `Butter Knife` 来避免 样板式的代码 ， 也能拥有更好的抽象能力

5、如果UI过于复杂，建议考虑创建一个 `Presenter(主持人)` 来处理UI的改变。

1. MVP模式，开始构建的时候会比较费劲，但是最终能让UI组件更容易进行测试

6、避免在ViewModel中引用 `View或者Activity` 的Context

1. 如果ViewModel比Activity活的更长久，会导致内存泄漏

使用案例

1、定位app，在前台时提供精细的定位并更新位置，在后台时提供粗糙的定位并更新位置的服务。

1. 可以使用 `LiveData` ， 作为一种生命周期感知组件，允许在位置改变时自动更新UI

2、开始和停止视频的缓冲

1. 使用生命周期感知组件，尽可能快的开始视频缓冲
2. 推迟重放操作，直到app完全启动
3. 在app销毁后，自动停止app缓冲

3、开始和停止网络的连接

1. 在前台时，开启实时更新网络数据
2. 在后台时，自动暂停人物

4、暂停和恢复Animated Drawables

1. 后台时暂停动画，前台时恢复动画

Stop事件的处理

1、当Fragment或者AppCompatActivity的状态通过 `onSaveInstanceState()` 保存后，直到 `ON_START` 被调用，UI都是不可变的

1. 如果尝试修改UI都会导致app导航状态出现矛盾
2. 这就是为什么在 状态保存后 触发 `FragmentTransaction` 会导致 `FragmentManager`抛出异常

2、LiveData如果发现观察者对应的Lifecycle还未到达 `STARTED` ,就不会通知观察者，用于防止出现上例的问题。

通过 `isAtLeast()` 进行判断

参考资料

1. 官方文档: Lifecycle