“终于懂了“系列：Jetpack AAC完整解析（一）Lifecycle 完全掌握！

胡飞洋 2020-11-11 22:20 19018

关注

> Jetpack AAC 系列文章:
>
> “终于懂了“系列：Jetpack AAC完整解析（一）Lifecycle 完全掌握！
>
> “终于懂了“系列：Jetpack AAC完整解析（二）LiveData 完全掌握！
>
> “终于懂了“系列：Jetpack AAC完整解析（三）ViewModel 完全掌握！
>
> “终于懂了“系列：Jetpack AAC完整解析（四）MVVM - Android架构探索！
>
> “终于懂了“系列：Jetpack AAC完整解析（五）DataBinding 重新认知！

> 欢迎关注我的 **公 众 号**，微信搜索 **胡飞洋** ，文章更新可第一时间收到。

一、Android Jetpack 介绍

1.1 Jetpack是啥

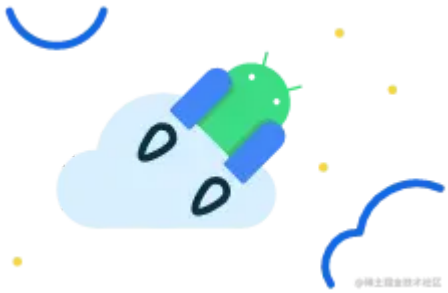
官方定义如下:

> 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，让开发者精力集中编写重要的代码。

JetPack更多是一种概念和态度，它是谷歌开发的非Android Framework SDK自带、但同时是Android开发必备的/推荐的SDK/开发规范合集。相当于Google把自己的Android生态重新整理了一番，确立了Android未来的开发大方向。

使用Jetpack有如下好处：

- **遵循最佳做法**，Android Jetpack 组件采用最新的设计方法构建，具有向后兼容性，可以减少崩溃和内存泄露。
- **消除样板代码**，Android Jetpack 可以管理各种繁琐的 Activity（如后台任务、导航和生命周期管理），以便您可以专注于打造出色的应用。
- **减少不一致**，这些库可在各种 Android 版本和设备中以一致的方式运作，助您降低复杂性。

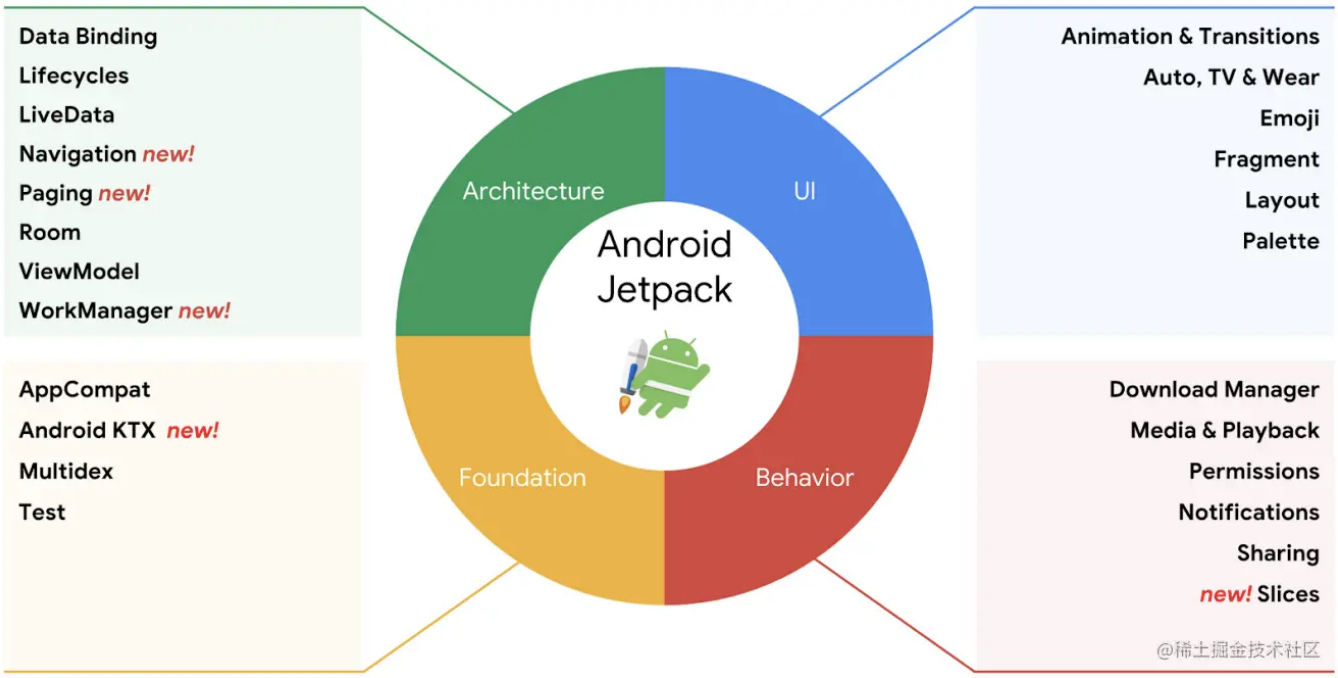


Jetpack原意为 喷气背包，Android背上Jetpack后就直冲云霄，这很形象了~

也就是，Jetpack是帮助开发者高效开发应用的工具集。那么这一工具包含了哪些内容呢?

1.2 Jetpack分类

分类如下图（现在官网已经找不到这个图了）：



Android Jetpack 组件覆盖以下 4 个方面：架构（Architecture）、基础（Foundation）、行为（Behavior）、界面（UI）。

真正的精华主要是Architecture，全称是Android Architecture Component（AAC），即**Android架构组件**。

其包括比较成功的Lifecycle、LiveData、ViewModel，同时也是我们使用MVVM模式的最好框架工具，可以组合使用，也可以单独使用。

> 以上基本都是官网的介绍，我们主要目标就是掌握AAC的组件，深入理解进而运用到MVVM架构中。

如题，我们学习Jetpack的重点就是AAC，这篇就从基础的Lifecycle讲起。

二、Lifecycle

Lifecycle，顾名思义，是用于帮助开发者管理Activity和Fragment 的生命周期，它是LiveData和ViewModel的基础。下面就先介绍为何及如何使用Lifecycle。

2.1 Lifecycle之前

官方文档有个例子 来说明使用Lifecycle之前是如何生命周期管理的：

假设我们有一个在屏幕上显示设备位置的 Activity。常见的实现可能如下所示：

java 复制代码

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

    void start() {
        // 连接系统定位服务
    }

    void stop() {
        // 断开系统定位服务
    }
}

class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

```

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

        @Override
        public void onStart() {
            super.onStart();
            myLocationListener.start();
            // 管理其他需要响应activity生命周期的组件
        }

        @Override
        public void onStop() {
            super.onStop();
            myLocationListener.stop();
            // 管理其他需要响应activity生命周期的组件
        }
    }

```

虽然此示例看起来没问题，但在真实的应用中，最终会有太多管理界面和其他组件的调用，以响应生命周期的当前状态。管理多个组件会在生命周期方法（如 onStart() 和 onStop()）中放置大量的代码，这使得它们难以维护。

此外，无法保证组件会在 Activity 或 Fragment 停止之前启动myLocationListener。在我们需要执行长时间运行的操作（如 onStart() 中的某种配置检查）时尤其如此。在这种情况下，myLocationListener的onStop() 方法会在 onStart() 之前调用，这使得组件留存的时间比所需的时间要长，从而导致内次泄漏。如下：

java 复制代码

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

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

        @Override
        public void onStart() {
            super.onStart();
```

```
            Util.checkUserStatus(result -> {
                //如果checkUserStatus耗时较长，在activity停止后才回调，那么myLocationListener
                //又因为myLocationListener持有activity，所以会造成内存泄漏。
                if (result) {
                    myLocationListener.start();
                }
            });
        }

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

```

即2个问题点：

- activity的生命周期内有大量管理组件的代码，难以维护。
- 无法保证组件会在 Activity/Fragment停止后不执行启动

Lifecycle库 则可以 以弹性和隔离的方式解决这些问题。

2.2 Lifecycle的使用

Lifecycle是一个库，也包含Lifecycle这样一个类，Lifecycle类 用于存储有关组件（如 Activity 或 Fragment）的生命周期状态的信息，并允许其他对象观察此状态。

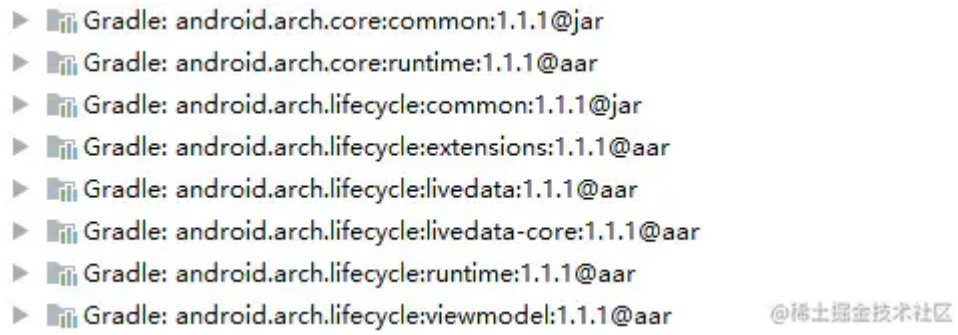
2.2.1 引入依赖

1、**非androidX项目** 引入:

java 复制代码

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

添加这一句代码就依赖了如下的库:



2、**androidX项目** 引入:

如果项目已经依赖了AndroidX:

java 复制代码

```
implementation 'androidx.appcompat:appcompat:1.2.0'
```

那么我们就可以使用Lifecycle库了，因为appcompat依赖了androidx.fragment，而androidx.fragment下依赖了ViewModel和 LiveData，LiveData内部又依赖了Lifecycle。

如果想要单独引入依赖，则如下：

在项目根目录的build.gradle添加 google() 代码库，然后app的build.gradle引入依赖，官方给出的依赖如下：

java 复制代码

```
//根目录的 build.gradle
    repositories {
        google()
        ...
    }

//app的build.gradle
    dependencies {
        def lifecycle_version = "2.2.0"
        def arch_version = "2.1.0"

        // ViewModel
        implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"
        // LiveData
        implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
        // 只有Lifecycles (不带 ViewModel or LiveData)
        implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle_version"

```

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

        // lifecycle注解处理器
        annotationProcessor "androidx.lifecycle:lifecycle-compiler:$lifecycle_version"
        // 替换 - 如果使用Java8,就用这个替换上面的lifecycle-compiler
        implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"

    //以下按需引入
        // 可选 - 帮助实现Service的LifecycleOwner
        implementation "androidx.lifecycle:lifecycle-service:$lifecycle_version"
        // 可选 - ProcessLifecycleOwner给整个 app进程 提供一个lifecycle
        implementation "androidx.lifecycle:lifecycle-process:$lifecycle_version"
        // 可选 - ReactiveStreams support for LiveData
        implementation "androidx.lifecycle:lifecycle-reactivestreams:$lifecycle_version"
        // 可选 - Test helpers for LiveData
        testImplementation "androidx.arch.core:core-testing:$arch_version"
    }

```

看着有很多，实际上如果只使用Lifecycle，只需要引入lifecycle-runtime即可。但通常都是和ViewModel、 LiveData 配套使用的，所以lifecycle-viewmodel、lifecycle-livedata 一般也会引入。

另外，lifecycle-process是给整个app进程提供一个lifecycle，会面也会提到。

2.2.2 使用方法

Lifecycle的使用很简单：

- 1、生命周期拥有者 使用getLifecycle()获取Lifecycle实例，然后代用addObserve()添加观察者；
- 2、观察者实现LifecycleObserver，方法上使用OnLifecycleEvent注解关注对应生命周期，生命周期触发时就会执行对应方法；

2.2.2.1 基本使用

在Activity（或Fragment）中 一般用法如下：

java 复制代码

```
public class LifecycleTestActivity extends AppCompatActivity {

```

```
    private String TAG = "Lifecycle_Test";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lifecycle_test);
        //Lifecycle 生命周期
        getLifecycle().addObserver(new MyObserver());
        Log.i(TAG, "onCreate: ");
    }
    @Override
    protected void onResume() {
        super.onResume();
        Log.i(TAG, "onResume: ");
    }
    @Override
    protected void onPause() {
        super.onPause();
        Log.i(TAG, "onPause: ");
    }
}
```

Activity（或Fragment）是生命周期的拥有者，通过getLifecycle()方法获取到生命周期Lifecycle对象，Lifecycle对象使用addObserver方法 给自己添加观察者，即MyObserver对象。当Lifecycle的生命周期发生变化时，MyObserver就可以感知到。

MyObserver是如何使用生命周期的呢？看下MyObserver的实现：

java 复制代码

```
public class MyObserver implements LifecycleObserver {

    private String TAG = "Lifecycle_Test";

    @OnLifecycleEvent(value = Lifecycle.Event.ON_RESUME)
    public void connect(){
        Log.i(TAG, "connect: ");
    }

    @OnLifecycleEvent(value = Lifecycle.Event.ON_PAUSE)
    public void disConnect(){
        Log.i(TAG, "disConnect: ");
    }
}
```

首先MyObserver实现了接口LifecycleObserver，LifecycleObserver用于标记一个类是生命周期观察者。 然后在connectListener()、disconnectListener()上 分别都加了@OnLifecycleEvent注

解，且value分别是Lifecycle.Event.ON_RESUME、Lifecycle.Event.ON_PAUSE，这个效果就是：connectListener()会在ON_RESUME时执行，disconnectListener()会在ON_PAUSE时执行。

我们打开LifecycleTestActivity 然后退出，日志打印如下：

java 复制代码

```
2020-11-09 17:25:40.601 4822-4822/com.hfy.androidlearning I/Lifecycle_Test: onCreate:

2020-11-09 17:25:40.605 4822-4822/com.hfy.androidlearning I/Lifecycle_Test: onResume:
2020-11-09 17:25:40.605 4822-4822/com.hfy.androidlearning I/Lifecycle_Test: connect:

2020-11-09 17:25:51.841 4822-4822/com.hfy.androidlearning I/Lifecycle_Test: disConnect:
2020-11-09 17:25:51.841 4822-4822/com.hfy.androidlearning I/Lifecycle_Test: onPause:
```

可见MyObserver的方法 确实是在对应关注的生命周期触发时调用。 当然注解中的value你也写成其它 你关注的任何一个生命周期，例如Lifecycle.Event.ON_DESTROY。

2.2.2.2 MVP架构中的使用

如果是 **在MVP架构中**，那么就可以把presenter作为观察者：

java 复制代码

```java
public class LifecycleTestActivity extends AppCompatActivity implements IView {
    private String TAG = "Lifecycle_Test";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lifecycle_test);
        //Lifecycle 生命周期
//        getLifecycle().addObserver(new MyObserver());

        //MVP中使用Lifecycle
        getLifecycle().addObserver(new MyPresenter(this));
        Log.i(TAG, "onCreate: ");
    }

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

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

    @Override
    public void showView() {}
    @Override
    public void hideView() {}
}

//Presenter
class MyPresenter implements LifecycleObserver {
    private static final String TAG = "Lifecycle_Test";
    private final IView mView;

    public MyPresenter(IView view) {mView = view;}

    @OnLifecycleEvent(value = Lifecycle.Event.ON_START)
    private void getDataOnStart(LifecycleOwner owner){
        Log.i(TAG, "getDataOnStart: ");

        Util.checkUserStatus(result -> {
                //checkUserStatus是耗时操作，回调后检查当前生命周期状态
                if (owner.getLifecycle().getCurrentState().isAtLeast(STARTED)) {
                   start();
                    mView.showView();
                }
            });
    }
    @OnLifecycleEvent(value = Lifecycle.Event.ON_STOP)
    private void hideDataOnStop(){
        Log.i(TAG, "hideDataOnStop: ");
        stop();
        mView.hideView();
    }
}

//IView
interface IView {
    void showView();
    void hideView();
}
```

这里是让Presenter实现LifecycleObserver接口，同样在方法上注解要触发的生命周期，最后在Activity中作为观察者添加到Lifecycle中。

这样做好处是啥呢？ 当Activity生命周期发生变化时，MyPresenter就可以感知并执行方法，不需要在MainActivity的多个生命周期方法中调用MyPresenter的方法了。

- **所有方法调用操作都由组件本身管理**：Presenter类自动感知生命周期，如果需要在其他的Activity/Fragment也使用这个Presenter，只需添加其为观察者即可。
- **让各个组件存储自己的逻辑，减轻Activity/Fragment中代码，更易于管理**；

—— 上面提到的第一个问题点就解决了。

另外，注意到 getDataOnStart()中耗时校验回调后，对当前生命周期状态进行了检查：至少处于STARTED状态才会继续执行start()方法，也就是保证了Activity停止后不会走start()方法；

—— 上面提到的第二个问题点也解决了。

2.2.3 自定义LifecycleOwner

在Activity中调用getLifecycle()能获取到Lifecycle实例，那getLifecycle()是哪里定义的方法呢？是接口LifecycleOwner，顾明来思义，生命周期拥有者：

java 复制代码

```java
/**
 * 生命周期拥有者
 * 生命周期事件可被 自定义的组件 用来 处理生命周期事件的变化，同时不会在Activity/Fragmen中写任何代码
 */
public interface LifecycleOwner {
    @NonNull
    Lifecycle getLifecycle();
}
```

Support Library 26.1.0及以上、AndroidX的 Fragment 和 Activity 已实现 LifecycleOwner 接口，所以我们在Activity中可以直接使用getLifecycle()。

如果有一个自定义类并希望使其成为LifecycleOwner，可以使用LifecycleRegistry类，它是Lifecycle的实现类，但需要将事件转发到该类：

java 复制代码

```java
    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;
        }
    }
```

MyActivity实现LifecycleOwner，getLifecycle()返回lifecycleRegistry实例。lifecycleRegistry实例则是在onCreate创建，并且在各个生命周期内调用markState()方法完成生命周期事件的传递。这就完成了LifecycleOwner的自定义，也即MyActivity变成了LifecycleOwner，然后就可以和 实现了LifecycleObserver的组件配合使用了。

补充一点，**观察者的方法可以接受一个参数LifecycleOwner**，就可以用来获取当前状态、或者继续添加观察者。 若注解的是ON_ANY还可以接收Event，用于区分是哪个事件。如下：

java 复制代码

```
    class TestObserver implements LifecycleObserver {
        @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
        void onCreated(LifecycleOwner owner) {
//            owner.getLifecycle().addObserver(anotherObserver);
//            owner.getLifecycle().getCurrentState();
        }
        @OnLifecycleEvent(Lifecycle.Event.ON_ANY)
        void onAny(LifecycleOwner owner, Lifecycle.Event event) {
//            event.name()
        }
    }
```

2.3 Application生命周期 ProcessLifecycleOwner

之前对App进入前后台的判断是通过registerActivityLifecycleCallbacks(callback)方法，然后在callback中利用一个全局变量做计数，在onActivityStarted()中计数加1，在onActivityStopped

方法中计数减1，从而判断前后台切换。

而使用ProcessLifecycleOwner可以直接获取应用前后台切换状态。（记得先引入lifecycle-process依赖）

使用方式和Activity中类似，只不过要使用ProcessLifecycleOwner.get()获取ProcessLifecycleOwner，代码如下：

java 复制代码

```java
public class MyApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();

   //注册App生命周期观察者
        ProcessLifecycleOwner.get().getLifecycle().addObserver(new ApplicationLifecycleO
    }

    /**
     * Application生命周期观察，提供整个应用进程的生命周期
     *
     * Lifecycle.Event.ON_CREATE只会分发一次，Lifecycle.Event.ON_DESTROY不会被分发。
     *
     * 第一个Activity进入时，ProcessLifecycleOwner将分派Lifecycle.Event.ON_START, Lifecycle
     * 而Lifecycle.Event.ON_PAUSE, Lifecycle.Event.ON_STOP，将在最后一个Activit退出后后延迟分
     *
     * 作用：监听应用程序进入前台或后台
     */
    private static class ApplicationLifecycleObserver implements LifecycleObserver {
        @OnLifecycleEvent(Lifecycle.Event.ON_START)
        private void onAppForeground() {
            Log.w(TAG, "ApplicationObserver: app moved to foreground");
        }

        @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
        private void onAppBackground() {
            Log.w(TAG, "ApplicationObserver: app moved to background");
        }
    }
}
```

看到确实很简单，和前面Activity的Lifecycle用法几乎一样，而我们使用ProcessLifecycleOwner就显得很优雅了。 生命周期分发逻辑已在注释里说明。

三、 源码分析

Lifecycle的使用很简单，接下来就是对Lifecycle原理和源码的解析了。

我们可以先猜下原理：LifecycleOwner（如Activity）在生命周期状态改变时（也就是生命周期方法执行时），遍历观察者，获取每个观察者的方法上的注解，如果注解是@OnLifecycleEvent且value是和生命周期状态一致，那么就执行这个方法。 这个猜测合理吧？下面你来看看。

3.1 Lifecycle类

先来瞅瞅Lifecycle：

java 复制代码

```java
public abstract class Lifecycle {
    //添加观察者
    @MainThread
    public abstract void addObserver(@NonNull LifecycleObserver observer);
    //移除观察者
    @MainThread
    public abstract void removeObserver(@NonNull LifecycleObserver observer);
    //获取当前状态
    public abstract State getCurrentState();

//生命周期事件，对应Activity生命周期方法
    public enum Event {
        ON_CREATE,
        ON_START,
        ON_RESUME,
        ON_PAUSE,
        ON_STOP,
        ON_DESTROY,
        ON_ANY  //可以响应任意一个事件
    }

    //生命周期状态。 (Event是进入这种状态的事件)
    public enum State {
        DESTROYED,
        INITIALIZED,
        CREATED,
        STARTED,
        RESUMED;

        //判断至少是某一状态
```

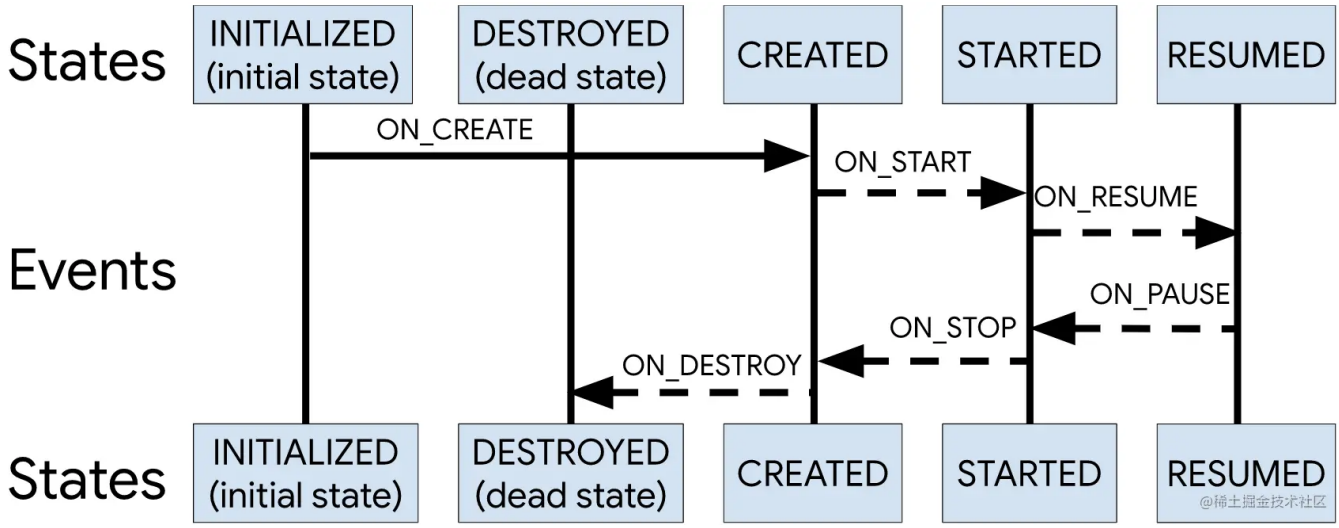
```
31          public boolean isAtLeast(@NonNull State state) {
32              return compareTo(state) >= 0;
33          }
34      }
```

Lifecycle 使用两种主要枚举跟踪其关联组件的生命周期状态：

1. Event，生命周期事件，这些事件对应Activity/Fragment生命周期方法。
2. State，生命周期状态，而Event是指进入一种状态的事件。 Event触发的时机：

- ON_CREATE、ON_START、ON_RESUME事件，是在LifecycleOwner对应的方法执行 之后分发。
- ON_PAUSE、ON_STOP、ON_DESTROY事件，是在LifecycleOwner对应的方法调用 之前分发。 这保证了LifecycleOwner是在这个状态内。

官网有个图很清晰：



3.2 Activity对LifecycleOwner的实现

前面提到Activity实现了LifecycleOwner，所以才能直接使用getLifecycle()，具体是在androidx.activity.ComponentActivity中：

java 复制代码

```
1  //androidx.activity.ComponentActivity，这里忽略了一些其他代码，我们只看Lifecycle相关
2  public class ComponentActivity extends androidx.core.app.ComponentActivity implements Li
3      ...
4
5      private final LifecycleRegistry mLifecycleRegistry = new LifecycleRegistry(this);
```

```
    ...
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mSavedStateRegistryController.performRestore(savedInstanceState);
        ReportFragment.injectIfNeededIn(this); //使用ReportFragment分发生命周期事件
        if (mContentLayoutId != 0) {
            setContentView(mContentLayoutId);
        }
    }
    @CallSuper
    @Override
    protected void onSaveInstanceState(@NonNull Bundle outState) {
        Lifecycle lifecycle = getLifecycle();
        if (lifecycle instanceof LifecycleRegistry) {
            ((LifecycleRegistry) lifecycle).setCurrentState(Lifecycle.State.CREATED);
        }
        super.onSaveInstanceState(outState);
        mSavedStateRegistryController.performSave(outState);
    }

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

这里忽略了一些其他代码，我们只看Lifecycle相关。

看到ComponentActivity实现了接口LifecycleOwner，并在getLifecycle()返回了LifecycleRegistry实例。前面提到LifecycleRegistry是Lifecycle具体实现。

然后在onSaveInstanceState()中设置mLifecycleRegistry的状态为State.CREATED，然后怎么没有了？其他生命周期方法内咋没处理？what？和猜测的不一样啊。 别急，在onCreate()中有这么一行：**ReportFragment**. `injectIfNeededIn(this);` ,这个就是关键所在。

3.3 生命周期事件分发——ReportFragment

java 复制代码

```
//专门用于分发生命周期事件的Fragment
public class ReportFragment extends Fragment {

    public static void injectIfNeededIn(Activity activity) {
```

```
        if (Build.VERSION.SDK_INT >= 29) {
            //在API 29及以上，可以直接注册回调 获取生命周期
            activity.registerActivityLifecycleCallbacks(
                    new LifecycleCallbacks());
        }
        //API29以前，使用fragment 获取生命周期
        if (manager.findFragmentByTag(REPORT_FRAGMENT_TAG) == null) {
            manager.beginTransaction().add(new ReportFragment(), REPORT_FRAGMENT_TAG).co
            manager.executePendingTransactions();
        }
    }

    @SuppressWarnings("deprecation")
    static void dispatch(@NonNull Activity activity, @NonNull Lifecycle.Event event) {
        if (activity instanceof LifecycleRegistryOwner) {//这里废弃了，不用看
            ((LifecycleRegistryOwner) activity).getLifecycle().handleLifecycleEvent(even
            return;
        }

        if (activity instanceof LifecycleOwner) {
            Lifecycle lifecycle = ((LifecycleOwner) activity).getLifecycle();
            if (lifecycle instanceof LifecycleRegistry) {
                ((LifecycleRegistry) lifecycle).handleLifecycleEvent(event);//使用Lifecyc
            }
        }
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        dispatch(Lifecycle.Event.ON_CREATE);
    }
    @Override
    public void onStart() {
        super.onStart();
        dispatch(Lifecycle.Event.ON_START);
    }
    @Override
    public void onResume() {
        super.onResume();
        dispatch(Lifecycle.Event.ON_RESUME);
    }
    @Override
    public void onPause() {
        super.onPause();
        dispatch(Lifecycle.Event.ON_PAUSE);
    }
    ...省略onStop、onDestroy

```

```
54      private void dispatch(@NonNull Lifecycle.Event event) {
55          if (Build.VERSION.SDK_INT < 29) {
56              dispatch(getActivity(), event);
57          }
58      }
59
60      //在API 29及以上，使用的生命周期回调
61      static class LifecycleCallbacks implements Application.ActivityLifecycleCallbacks {
62          ...
63          @Override
64          public void onActivityPostCreated(@NonNull Activity activity,@Nullable Bundle sa
65              dispatch(activity, Lifecycle.Event.ON_CREATE);
66          }
67          @Override
68          public void onActivityPostStarted(@NonNull Activity activity) {
69              dispatch(activity, Lifecycle.Event.ON_START);
70          }
71          @Override
72          public void onActivityPostResumed(@NonNull Activity activity) {
73              dispatch(activity, Lifecycle.Event.ON_RESUME);
74          }
75          @Override
76          public void onActivityPrePaused(@NonNull Activity activity) {
77              dispatch(activity, Lifecycle.Event.ON_PAUSE);
78          }
79          ...省略onStop、onDestroy
80      }
81  }
```

首先injectIfNeededIn()内进行了版本区分：在API 29及以上 直接使用activity的registerActivityLifecycleCallbacks 直接注册了生命周期回调，然后给当前activity添加了ReportFragment，注意这个fragment是没有布局的。

然后， 无论LifecycleCallbacks、还是fragment的生命周期方法 最后都走到了 dispatch(Activity activity, Lifecycle.Event event)方法，其内部使用LifecycleRegistry的handleLifecycleEvent方法处理事件。

而ReportFragment的作用就是获取生命周期而已，因为fragment生命周期是依附Activity的。好处就是把这部分逻辑抽离出来，实现activity的无侵入。如果你对图片加载库Glide比较熟，就会知道它也是使用透明Fragment获取生命周期的。

3.4 生命周期事件处理——LifecycleRegistry

到这里，生命中周期事件的处理有转移到了 **LifecycleRegistry** 中：

java 复制代码

```java
//LifecycleRegistry.java
   //系统自定义的保存Observer的map，可在遍历中增删
    private FastSafeIterableMap<LifecycleObserver, ObserverWithState> mObserverMap = new

    public void handleLifecycleEvent(@NonNull Lifecycle.Event event) {
        State next = getStateAfter(event);//获取event发生之后的将要处于的状态
        moveToState(next);//移动到这个状态
    }

    private void moveToState(State next) {
        if (mState == next) {
            return;//如果和当前状态一致，不处理
        }
        mState = next; //赋值新状态
        if (mHandlingEvent || mAddingObserverCounter != 0) {
            mNewEventOccurred = true;
            return;
        }
        mHandlingEvent = true;
        sync(); //把生命周期状态同步给所有观察者
        mHandlingEvent = false;
    }

        private void sync() {
        LifecycleOwner lifecycleOwner = mLifecycleOwner.get();
        if (lifecycleOwner == null) {
            throw new IllegalStateException("LifecycleOwner of this LifecycleRegistry is
                    + "garbage collected. It is too late to change lifecycle state.");
        }
        while (!isSynced()) {  //isSynced()意思是 所有观察者都同步完了
            mNewEventOccurred = false;
            //mObserverMap就是 在activity中添加observer后 用于存放observer的map
            if (mState.compareTo(mObserverMap.eldest().getValue().mState) < 0) {
                backwardPass(lifecycleOwner);
            }
            Entry<LifecycleObserver, ObserverWithState> newest = mObserverMap.newest();
            if (!mNewEventOccurred && newest != null
                    && mState.compareTo(newest.getValue().mState) > 0) {
                forwardPass(lifecycleOwner);
            }
        }
        mNewEventOccurred = false;
    }
    ...

```

```
46      static State getStateAfter(Event event) {
47          switch (event) {
48              case ON_CREATE:
49              case ON_STOP:
50                  return CREATED;
51              case ON_START:
52              case ON_PAUSE:
53                  return STARTED;
54              case ON_RESUME:
55                  return RESUMED;
56              case ON_DESTROY:
57                  return DESTROYED;
58              case ON_ANY:
59                  break;
60          }
61          throw new IllegalArgumentException("Unexpected event value " + event);
62      }
```

逻辑很清晰：使用getStateAfter()获取event发生之后的将要处于的状态（看前面那张图很好理解），moveToState()是移动到新状态，最后使用sync()把生命周期状态同步给所有观察者。

注意到sync()中有个while循环，很显然是在遍历观察者。并且很显然观察者是存放在mObserverMap中的，而mObserverMap对观察者的添加 很显然 就是 Activity中使用getLifecycle().addObserver()这里：

java 复制代码

```
1  //LifecycleRegistry.java
2      @Override
3      public void addObserver(@NonNull LifecycleObserver observer) {
4          State initialState = mState == DESTROYED ? DESTROYED : INITIALIZED;
5          //带状态的观察者，这个状态的作用：新的事件触发后 遍历通知所有观察者时，判断是否已经通知这个观察
6          ObserverWithState statefulObserver = new ObserverWithState(observer, initialStat
7          ObserverWithState previous = mObserverMap.putIfAbsent(observer, statefulObserver
8          //observer作为key，ObserverWithState作为value，存到mObserverMap
9
10         if (previous != null) {
11             return;//已经添加过，不处理
12         }
13         LifecycleOwner lifecycleOwner = mLifecycleOwner.get();
14         if (lifecycleOwner == null) {
15             return;//lifecycleOwner退出了，不处理
16         }
17     //下面代码的逻辑：通过while循环，把新的观察者的状态 连续地 同步到最新状态mState。
18      //意思就是：虽然可能添加的晚，但把之前的事件一个个分发给你(upEvent方法)，即粘性
19         boolean isReentrance = mAddingObserverCounter != 0 || mHandlingEvent;
20         State targetState = calculateTargetState(observer);//计算目标状态
```

```
        mAddingObserverCounter++;
        while ((statefulObserver.mState.compareTo(targetState) < 0
                && mObserverMap.contains(observer))) {
            pushParentState(statefulObserver.mState);
            statefulObserver.dispatchEvent(lifecycleOwner, upEvent(statefulObserver.mSta
            popParentState();
            // mState / subling may have been changed recalculate
            targetState = calculateTargetState(observer);
        }

        if (!isReentrance) {
            sync();
        }
        mAddingObserverCounter--;
    }
```

用observer创建带状态的观察者ObserverWithState，observer作为key、ObserverWithState作为value，存到mObserverMap。 接着做了安全判断，最后把新的观察者的状态 连续地 同步到最新状态mState，意思就是：虽然可能添加的晚，但会把之前的事件一个个分发给你，即粘性。

回到刚刚sync()的while循环，看看如何处理分发事件：

java 复制代码

```
    private void sync() {
        LifecycleOwner lifecycleOwner = mLifecycleOwner.get();
        if (lifecycleOwner == null) {
            Log.w(LOG_TAG, "LifecycleOwner is garbage collected, you shouldn't try dispa
                    + "new events from it.");
            return;
        }
        while (!isSynced()) {
            mNewEventOccurred = false;
            // no need to check eldest for nullability, because isSynced does it for us.
            if (mState.compareTo(mObserverMap.eldest().getValue().mState) < 0) {
                backwardPass(lifecycleOwner);
            }
            Entry<LifecycleObserver, ObserverWithState> newest = mObserverMap.newest();
            if (!mNewEventOccurred && newest != null
                    && mState.compareTo(newest.getValue().mState) > 0) {
                forwardPass(lifecycleOwner);
            }
        }
        mNewEventOccurred = false;
    }
```

```
23      private boolean isSynced() {
24          if (mObserverMap.size() == 0) {
25              return true;
26          }//最老的和最新的观察者的状态一致，都是ower的当前状态，说明已经同步完了
27          State eldestObserverState = mObserverMap.eldest().getValue().mState;
28          State newestObserverState = mObserverMap.newest().getValue().mState;
29          return eldestObserverState == newestObserverState && mState == newestObserverSta
30      }
31
32      private void forwardPass(LifecycleOwner lifecycleOwner) {
33          Iterator<Entry<LifecycleObserver, ObserverWithState>> ascendingIterator = mObser
34          while (ascendingIterator.hasNext() && !mNewEventOccurred) {//正向遍历，从老到新
35              Entry<LifecycleObserver, ObserverWithState> entry = ascendingIterator.next()
36              ObserverWithState observer = entry.getValue();
37              while ((observer.mState.compareTo(mState) < 0 && !mNewEventOccurred && mObse
38                  pushParentState(observer.mState);
39                  observer.dispatchEvent(lifecycleOwner, upEvent(observer.mState));//obser
40                  popParentState();
41              }
42          }
43      }
44
45      private void backwardPass(LifecycleOwner lifecycleOwner) {
46          Iterator<Entry<LifecycleObserver, ObserverWithState>> descendingIterator = mObse
47          while (descendingIterator.hasNext() && !mNewEventOccurred) {//反向遍历，从新到老
48              Entry<LifecycleObserver, ObserverWithState> entry = descendingIterator.next(
49              ObserverWithState observer = entry.getValue();
50              while ((observer.mState.compareTo(mState) > 0 && !mNewEventOccurred && mObse
51                  Event event = downEvent(observer.mState);
52                  pushParentState(getStateAfter(event));
53                  observer.dispatchEvent(lifecycleOwner, event);//observer获取事件
54                  popParentState();
55              }
56          }
57      }
```

循环条件是!isSynced()，若最老的和最新的观察者的状态一致，且都是ower的当前状态，说明已经同步完了。

没有同步完就进入循环体：

- mState比最老观察者状态小，走backwardPass(lifecycleOwner)：从新到老分发，循环使用downEvent()和observer.dispatchEvent()，连续分发事件；
- mState比最新观察者状态大，走forwardPass(lifecycleOwner)：从老到新分发，循环使用upEvent()和observer.dispatchEvent()，连续分发事件。

接着ObserverWithState类型的observer就获取到了事件，即observer.dispatchEvent(lifecycleOwner, event)，下面来看看它是如何让加了对应注解的方法执行的。

3.5 事件回调后 方法执行

我们继续看下 **ObserverWithState**:

java 复制代码

```java
    static class ObserverWithState {
        State mState;
        GenericLifecycleObserver mLifecycleObserver;

        ObserverWithState(LifecycleObserver observer, State initialState) {
            mLifecycleObserver = Lifecycling.getCallback(observer);
            mState = initialState;
        }

        void dispatchEvent(LifecycleOwner owner, Event event) {
            State newState = getStateAfter(event);
            mState = min(mState, newState);
            mLifecycleObserver.onStateChanged(owner, event);
            mState = newState;
        }
    }
```

mState的作用是：新的事件触发后 遍历通知所有观察者时，判断是否已经通知这个观察者了，即防止重复通知。

mLifecycleObserver是使用Lifecycling.getCallback(observer)获取的GenericLifecycleObserver实例。GenericLifecycleObserver是接口，继承自LifecycleObserver：

java 复制代码

```java
//接受生命周期改变并分发给真正的观察者
public interface LifecycleEventObserver extends LifecycleObserver {
    //生命周期状态变化
    void onStateChanged(@NonNull LifecycleOwner source, @NonNull Lifecycle.Event event);
}
```

也就说，LifecycleEventObserver 给 LifecycleObserver 增加了感知生命周期状态变化的能力。

看看Lifecycling.getCallback(observer)：

```java
    @NonNull
    static LifecycleEventObserver lifecycleEventObserver(Object object) {
        ...省略很多类型判断的代码
        return new ReflectiveGenericLifecycleObserver(object);
    }
```

方法内有很多对observer进行类型判断的代码，我们这里关注的是ComponentActivity，所以LifecycleEventObserver的实现类就是ReflectiveGenericLifecycleObserver了：

```java
class ReflectiveGenericLifecycleObserver implements LifecycleEventObserver {
    private final Object mWrapped;
    private final CallbackInfo mInfo;

    ReflectiveGenericLifecycleObserver(Object wrapped) {
        mWrapped = wrapped;
        mInfo = ClassesInfoCache.sInstance.getInfo(mWrapped.getClass());//存放了event与加
    }

    @Override
    public void onStateChanged(@NonNull LifecycleOwner source, @NonNull Event event) {
        mInfo.invokeCallbacks(source, event, mWrapped);//执行对应event的观察者的方法
    }
}
```

它的onStateChanged()方法内部使用CallbackInfo的invokeCallbacks方法，这里应该就是执行观察者的方法了。

ClassesInfoCache内部用Map存了 所有观察者的回调信息，CallbackInfo是当前观察者的回调信息。

先看下CallbackInfo实例的创建，ClassesInfoCache.sInstance.getInfo(mWrapped.getClass())：

```java
//ClassesInfoCache.java
    private final Map<Class, CallbackInfo> mCallbackMap = new HashMap<>();//所有观察者的回
    private final Map<Class, Boolean> mHasLifecycleMethods = new HashMap<>();//观察者是否
```

```
4
5       CallbackInfo getInfo(Class<?> klass) {
6           CallbackInfo existing = mCallbackMap.get(klass);//如果已经存在当前观察者回调信息 直接
7           if (existing != null) {
8               return existing;
9           }
10          existing = createInfo(klass, null);//没有就去收集信息并创建
11          return existing;
12      }
13
14      private CallbackInfo createInfo(Class<?> klass, @Nullable Method[] declaredMethods)
15          Class<?> superclass = klass.getSuperclass();
16          Map<MethodReference, Lifecycle.Event> handlerToEvent = new HashMap<>();//生命周期
17          ...
18          Method[] methods = declaredMethods != null ? declaredMethods : getDeclaredMethod
19          boolean hasLifecycleMethods = false;
20          for (Method method : methods) {//遍历方法 找到注解OnLifecycleEvent
21              OnLifecycleEvent annotation = method.getAnnotation(OnLifecycleEvent.class);
22              if (annotation == null) {
23                  continue; //没有注解OnLifecycleEvent 就return
24              }
25              hasLifecycleMethods = true;//有注解OnLifecycleEvent
26              Class<?>[] params = method.getParameterTypes(); //获取方法参数
27              int callType = CALL_TYPE_NO_ARG;
28              if (params.length > 0) { //有参数
29                  callType = CALL_TYPE_PROVIDER;
30                  if (!params[0].isAssignableFrom(LifecycleOwner.class)) {
31                      throw new IllegalArgumentException(//第一个参数必须是LifecycleOwner
32                              "invalid parameter type. Must be one and instanceof Lifecycl
33                  }
34              }
35              Lifecycle.Event event = annotation.value();
36
37              if (params.length > 1) {
38                  callType = CALL_TYPE_PROVIDER_WITH_EVENT;
39                  if (!params[1].isAssignableFrom(Lifecycle.Event.class)) {
40                      throw new IllegalArgumentException(//第二个参数必须是Event
41                              "invalid parameter type. second arg must be an event");
42                  }
43                  if (event != Lifecycle.Event.ON_ANY) {
44                      throw new IllegalArgumentException(//有两个参数 注解值只能是ON_ANY
45                              "Second arg is supported only for ON_ANY value");
46                  }
47              }
48              if (params.length > 2) { //参数不能超过两个
49                  throw new IllegalArgumentException("cannot have more than 2 params");
50              }
51              MethodReference methodReference = new MethodReference(callType, method);
52              verifyAndPutHandler(handlerToEvent, methodReference, event, klass);//校验方法
```

```
        }
        CallbackInfo info = new CallbackInfo(handlerToEvent);//获取的 所有注解生命周期的方法
        mCallbackMap.put(klass, info);//把当前观察者的回调信息存到ClassesInfoCache中
        mHasLifecycleMethods.put(klass, hasLifecycleMethods);//记录 观察者是否有注解了生命周
        return info;
    }
```

- 如果不存在当前观察者回调信息，就使用createInfo()方法收集创建
- 先反射获取观察者的方法，遍历方法 找到注解了OnLifecycleEvent的方法，先对方法的参数进行了校验。
- 第一个参数必须是LifecycleOwner；第二个参数必须是Event；有两个参数 注解值只能是ON_ANY；参数不能超过两个
- 校验方法并加入到map，key是方法，value是Event。map handlerToEvent是所有的注解了生命周期的方法。
- 遍历完，然后用 handlerToEvent来构造 当前观察者回调信息CallbackInfo，存到ClassesInfoCache的mCallbackMap中，并记录 观察者是否有注解了生命周期的方法。

整体思路还是很清晰的，继续看CallbackInfo的invokeCallbacks方法：

java 复制代码

```java
    static class CallbackInfo {
        final Map<Lifecycle.Event, List<MethodReference>> mEventToHandlers;//Event对应的
        final Map<MethodReference, Lifecycle.Event> mHandlerToEvent;//要回调的方法

        CallbackInfo(Map<MethodReference, Lifecycle.Event> handlerToEvent) {
            mHandlerToEvent = handlerToEvent;
            mEventToHandlers = new HashMap<>();
            //这里遍历mHandlerToEvent来获取mEventToHandlers
            for (Map.Entry<MethodReference, Lifecycle.Event> entry : handlerToEvent.entr
                Lifecycle.Event event = entry.getValue();
                List<MethodReference> methodReferences = mEventToHandlers.get(event);
                if (methodReferences == null) {
                    methodReferences = new ArrayList<>();
                    mEventToHandlers.put(event, methodReferences);
                }
                methodReferences.add(entry.getKey());
            }
        }

        @SuppressWarnings("ConstantConditions")
        void invokeCallbacks(LifecycleOwner source, Lifecycle.Event event, Object target
            invokeMethodsForEvent(mEventToHandlers.get(event), source, event, target);//
            invokeMethodsForEvent(mEventToHandlers.get(Lifecycle.Event.ON_ANY), source,
        }
```

```
25
26          private static void invokeMethodsForEvent(List<MethodReference> handlers,
27                  LifecycleOwner source, Lifecycle.Event event, Object mWrapped) {
28              if (handlers != null) {
29                  for (int i = handlers.size() - 1; i >= 0; i--) {//执行Event对应的多个方法
30                      handlers.get(i).invokeCallback(source, event, mWrapped);
31                  }
32              }
33          }
34      }
```

很好理解，执行对应event的方法、执行注解了ON_ANY的方法。其中mEventToHandlers是在创建CallbackInfo时由遍历mHandlerToEvent来获取，存放了每个Event对应的多个方法。

最后看看handlers.get(i).invokeCallback，即MethodReference中：

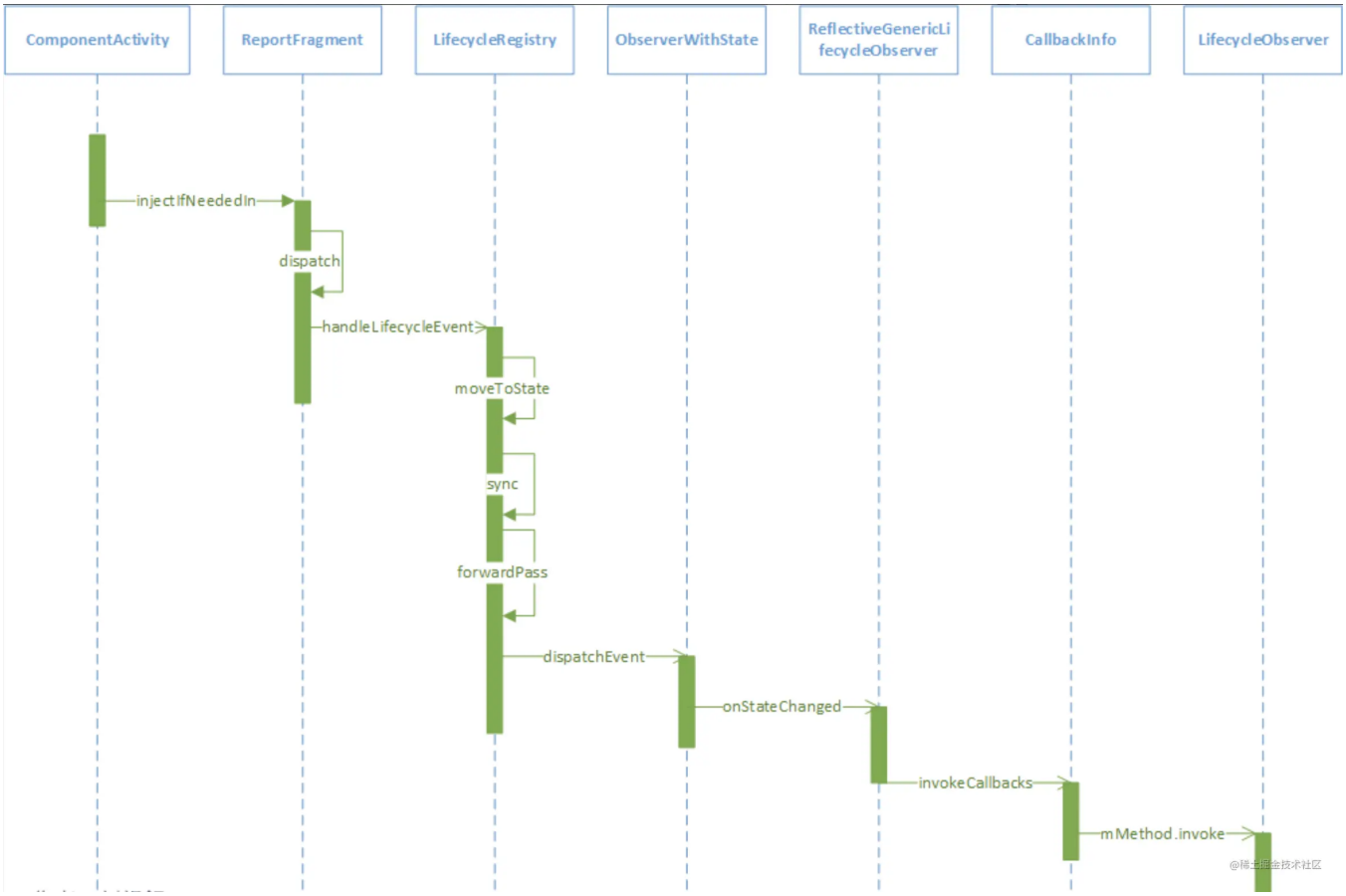
java 复制代码

```
1       static class MethodReference {
2           ...
3
4           void invokeCallback(LifecycleOwner source, Lifecycle.Event event, Object target)
5               try {
6                   switch (mCallType) {
7                       case CALL_TYPE_NO_ARG:
8                           mMethod.invoke(target);//没有参数的
9                           break;
10                      case CALL_TYPE_PROVIDER:
11                          mMethod.invoke(target, source);//一个参数的：LifecycleOwner
12                          break;
13                      case CALL_TYPE_PROVIDER_WITH_EVENT:
14                          mMethod.invoke(target, source, event);//两个参数的：LifecycleOwner
15                          break;
16                  }
17              }
18             ...
19          }
20  ...
21      }
```

根据不同参数类型，执行对应方法。

到这里，整个流程就完整了。实际看了这么一大圈，基本思路和我们的猜想是一致的。

这里借Android Jetpack架构组件（三）一文带你了解Lifecycle（原理篇）的图总结下：



四、总结

本文先介绍了Jetpack和AAC的概念，这是Android官方推荐的通用开发工具集。其中AAC是架构组件，是本系列文章的介绍内容。接着介绍了AAC的基础组件Lifecycle，它能让开发者更好的管理Activity/Fragment生命周期。最后详细分析了Lifecycle源码及原理。

Jetpack的AAC是我们后续开发Android必备知识，也是完成MVVM架构的基础。Lifecycle更是AAC中的基础，所以完整掌握本篇内容十分必要。

.

感谢与参考：

Lifecycle官方文档

Android Jetpack架构组件（三）一文带你了解Lifecycle（原理篇）

Android架构组件（2）LifecycleRegistry 源码分析

.

> **你的 点赞、评论，是对我的巨大鼓励！**

> 欢迎关注我的 **公 众 号**：胡飞洋

标签： Android Jetpack

文章被收录于专栏：

JetPack 架构组件系列

JetPack 架构组件 全面解析~

订阅专栏