**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

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：

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：

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启动后就没办法走stop()方法了，

//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。

### 2.2.2 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；
* 3、**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。观察者通过在对应的操作前执行状态判断，即可避免在错误的状态执行操作。owner.getLifecycle().getCurrentState().isAtLeast(STARTED)

Support Library 26.1.0及以上、AndroidX的 Fragment 和 Activity 已实现 LifecycleOwner 接口。

/\*\*

\* 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();

}

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：

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;

}

}

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：

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 ApplicationLifecycleObserver());

}

/\*\*

\* 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.Event.ON\_RESUME。

\* 而Lifecycle.Event.ON\_PAUSE, Lifecycle.Event.ON\_STOP，将在最后一个Activit退出后后延迟分发。如果由于配置更改而销毁并重新创建活动，则此延迟足以保证ProcessLifecycleOwner不会发送任何事件。

\*

\* 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");

}

}

}

**源码分析**

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。

## Lifecycle类

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

public boolean isAtLeast(@NonNull State state) {

return compareTo(state) >= 0;

}

}

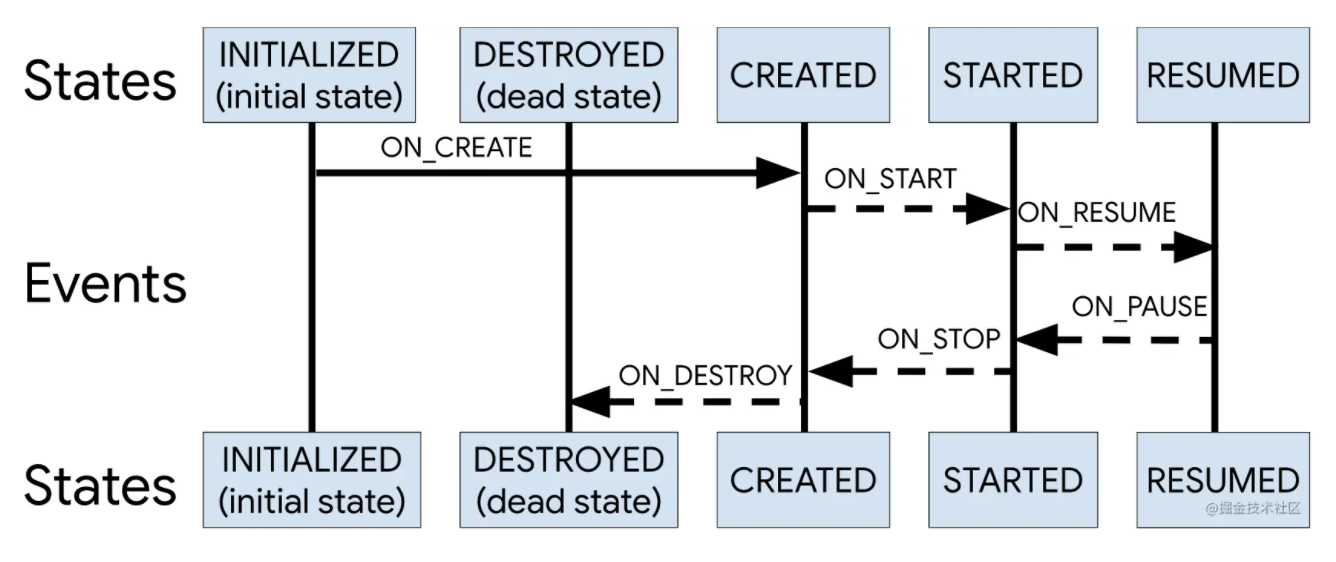
Lifecycle 使用两种主要枚举跟踪其关联组件的生命周期状态：

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。



## 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:

//androidx.activity.ComponentActivity，这里忽略了一些其他代码，我们只看Lifecycle相关

public class ComponentActivity extends androidx.core.app.ComponentActivity implements LifecycleOwner{

...

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;

}

}

## 生命周期事件分发——ReportFragment

//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).commit();

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(event);

return;

}

if (activity instanceof LifecycleOwner) {

Lifecycle lifecycle = ((LifecycleOwner) activity).getLifecycle();

if (lifecycle instanceof LifecycleRegistry) {

((LifecycleRegistry) lifecycle).handleLifecycleEvent(event);//使用LifecycleRegistry的handleLifecycleEvent方法处理事件

}

}

}

@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

private void dispatch(@NonNull Lifecycle.Event event) {

if (Build.VERSION.SDK\_INT < 29) {

dispatch(getActivity(), event);

}

}

//在API 29及以上，使用的生命周期回调

static class LifecycleCallbacks implements Application.ActivityLifecycleCallbacks {

...

@Override

public void onActivityPostCreated(@NonNull Activity activity,@Nullable Bundle savedInstanceState) {

dispatch(activity, Lifecycle.Event.ON\_CREATE);

}

@Override

public void onActivityPostStarted(@NonNull Activity activity) {

dispatch(activity, Lifecycle.Event.ON\_START);

}

@Override

public void onActivityPostResumed(@NonNull Activity activity) {

dispatch(activity, Lifecycle.Event.ON\_RESUME);

}

@Override

public void onActivityPrePaused(@NonNull Activity activity) {

dispatch(activity, Lifecycle.Event.ON\_PAUSE);

}

...省略onStop、onDestroy

}

}

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。

## 生命周期事件处理——LifecycleRegistry

//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 FastSafeIterableMap<>();

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 already"

+ "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;

}

...

static State getStateAfter(Event event) {

switch (event) {

case ON\_CREATE:

case ON\_STOP:

return CREATED;

case ON\_START:

case ON\_PAUSE:

return STARTED;

case ON\_RESUME:

return RESUMED;

case ON\_DESTROY:

return DESTROYED;

case ON\_ANY:

break;

}

throw new IllegalArgumentException("Unexpected event value " + event);

}

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：

//LifecycleRegistry.java

@Override

public void addObserver(@NonNull LifecycleObserver observer) {

State initialState = mState == DESTROYED ? DESTROYED : INITIALIZED;

//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者了

ObserverWithState statefulObserver = new ObserverWithState(observer, initialState);

ObserverWithState previous = mObserverMap.putIfAbsent(observer, statefulObserver);

//observer作为key，ObserverWithState作为value，存到mObserverMap

if (previous != null) {

return;//已经添加过，不处理

}

LifecycleOwner lifecycleOwner = mLifecycleOwner.get();

if (lifecycleOwner == null) {

return;//lifecycleOwner退出了，不处理

}

//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。

//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

boolean isReentrance = mAddingObserverCounter != 0 || mHandlingEvent;

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.mState));

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：

private void sync() {

LifecycleOwner lifecycleOwner = mLifecycleOwner.get();

if (lifecycleOwner == null) {

Log.w(LOG\_TAG, "LifecycleOwner is garbage collected, you shouldn't try dispatch "

+ "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;

}

private boolean isSynced() {

if (mObserverMap.size() == 0) {

return true;

}//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

State eldestObserverState = mObserverMap.eldest().getValue().mState;

State newestObserverState = mObserverMap.newest().getValue().mState;

return eldestObserverState == newestObserverState && mState == newestObserverState;

}

private void forwardPass(LifecycleOwner lifecycleOwner) {

Iterator<Entry<LifecycleObserver, ObserverWithState>> ascendingIterator = mObserverMap.iteratorWithAdditions();

while (ascendingIterator.hasNext() && !mNewEventOccurred) {//正向遍历，从老到新

Entry<LifecycleObserver, ObserverWithState> entry = ascendingIterator.next();

ObserverWithState observer = entry.getValue();

while ((observer.mState.compareTo(mState) < 0 && !mNewEventOccurred && mObserverMap.contains(entry.getKey()))) {

pushParentState(observer.mState);

observer.dispatchEvent(lifecycleOwner, upEvent(observer.mState));//observer获取事件

popParentState();

}

}

}

private void backwardPass(LifecycleOwner lifecycleOwner) {

Iterator<Entry<LifecycleObserver, ObserverWithState>> descendingIterator = mObserverMap.descendingIterator();

while (descendingIterator.hasNext() && !mNewEventOccurred) {//反向遍历，从新到老

Entry<LifecycleObserver, ObserverWithState> entry = descendingIterator.next();

ObserverWithState observer = entry.getValue();

while ((observer.mState.compareTo(mState) > 0 && !mNewEventOccurred && mObserverMap.contains(entry.getKey()))) {

Event event = downEvent(observer.mState);

pushParentState(getStateAfter(event));

observer.dispatchEvent(lifecycleOwner, event);//observer获取事件

popParentState();

}

}

}

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：

private void sync() {

LifecycleOwner lifecycleOwner = mLifecycleOwner.get();

if (lifecycleOwner == null) {

Log.w(LOG\_TAG, "LifecycleOwner is garbage collected, you shouldn't try dispatch "

+ "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;

}

private boolean isSynced() {

if (mObserverMap.size() == 0) {

return true;

}//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

State eldestObserverState = mObserverMap.eldest().getValue().mState;

State newestObserverState = mObserverMap.newest().getValue().mState;

return eldestObserverState == newestObserverState && mState == newestObserverState;

}

private void forwardPass(LifecycleOwner lifecycleOwner) {

Iterator<Entry<LifecycleObserver, ObserverWithState>> ascendingIterator = mObserverMap.iteratorWithAdditions();

while (ascendingIterator.hasNext() && !mNewEventOccurred) {//正向遍历，从老到新

Entry<LifecycleObserver, ObserverWithState> entry = ascendingIterator.next();

ObserverWithState observer = entry.getValue();

while ((observer.mState.compareTo(mState) < 0 && !mNewEventOccurred && mObserverMap.contains(entry.getKey()))) {

pushParentState(observer.mState);

observer.dispatchEvent(lifecycleOwner, upEvent(observer.mState));//observer获取事件

popParentState();

}

}

}

private void backwardPass(LifecycleOwner lifecycleOwner) {

Iterator<Entry<LifecycleObserver, ObserverWithState>> descendingIterator = mObserverMap.descendingIterator();

while (descendingIterator.hasNext() && !mNewEventOccurred) {//反向遍历，从新到老

Entry<LifecycleObserver, ObserverWithState> entry = descendingIterator.next();

ObserverWithState observer = entry.getValue();

while ((observer.mState.compareTo(mState) > 0 && !mNewEventOccurred && mObserverMap.contains(entry.getKey()))) {

Event event = downEvent(observer.mState);

pushParentState(getStateAfter(event));

observer.dispatchEvent(lifecycleOwner, event);//observer获取事件

popParentState();

}

}

}

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。

## 事件回调后 方法执行

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：

//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)：

@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：

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与加了注解方法的信息

}

@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())：

//ClassesInfoCache.java

private final Map<Class, CallbackInfo> mCallbackMap = new HashMap<>();//所有观察者的回调信息

private final Map<Class, Boolean> mHasLifecycleMethods = new HashMap<>();//观察者是否有注解了生命周期的方法

CallbackInfo getInfo(Class<?> klass) {

CallbackInfo existing = mCallbackMap.get(klass);//如果已经存在当前观察者回调信息 直接取

if (existing != null) {

return existing;

}

existing = createInfo(klass, null);//没有就去收集信息并创建

return existing;

}

private CallbackInfo createInfo(Class<?> klass, @Nullable Method[] declaredMethods) {

Class<?> superclass = klass.getSuperclass();

Map<MethodReference, Lifecycle.Event> handlerToEvent = new HashMap<>();//生命周期事件到来 对应的方法

...

Method[] methods = declaredMethods != null ? declaredMethods : getDeclaredMethods(klass);//反射获取观察者的方法

boolean hasLifecycleMethods = false;

for (Method method : methods) {//遍历方法 找到注解OnLifecycleEvent

OnLifecycleEvent annotation = method.getAnnotation(OnLifecycleEvent.class);

if (annotation == null) {

continue; //没有注解OnLifecycleEvent 就return

}

hasLifecycleMethods = true;//有注解OnLifecycleEvent

Class<?>[] params = method.getParameterTypes(); //获取方法参数

int callType = CALL\_TYPE\_NO\_ARG;

if (params.length > 0) { //有参数

callType = CALL\_TYPE\_PROVIDER;

if (!params[0].isAssignableFrom(LifecycleOwner.class)) {

throw new IllegalArgumentException(//第一个参数必须是LifecycleOwner

"invalid parameter type. Must be one and instanceof LifecycleOwner");

}

}

Lifecycle.Event event = annotation.value();

if (params.length > 1) {

callType = CALL\_TYPE\_PROVIDER\_WITH\_EVENT;

if (!params[1].isAssignableFrom(Lifecycle.Event.class)) {

throw new IllegalArgumentException(//第二个参数必须是Event

"invalid parameter type. second arg must be an event");

}

if (event != Lifecycle.Event.ON\_ANY) {

throw new IllegalArgumentException(//有两个参数 注解值只能是ON\_ANY

"Second arg is supported only for ON\_ANY value");

}

}

if (params.length > 2) { //参数不能超过两个

throw new IllegalArgumentException("cannot have more than 2 params");

}

MethodReference methodReference = new MethodReference(callType, method);

verifyAndPutHandler(handlerToEvent, methodReference, event, klass);//校验方法并加入到map handlerToEvent 中

}

CallbackInfo info = new CallbackInfo(handlerToEvent);//获取的 所有注解生命周期的方法handlerToEvent，构造回调信息实例

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期的方法

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：

static class CallbackInfo {

final Map<Lifecycle.Event, List<MethodReference>> mEventToHandlers;//Event对应的多个方法

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.entrySet()) {

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);//执行对应event的方法

invokeMethodsForEvent(mEventToHandlers.get(Lifecycle.Event.ON\_ANY), source, event,target);//执行注解了ON\_ANY的方法

}

private static void invokeMethodsForEvent(List<MethodReference> handlers,

LifecycleOwner source, Lifecycle.Event event, Object mWrapped) {

if (handlers != null) {

for (int i = handlers.size() - 1; i >= 0; i--) {//执行Event对应的多个方法

handlers.get(i).invokeCallback(source, event, mWrapped);

}

}

}

}

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：

static class MethodReference {

...

void invokeCallback(LifecycleOwner source, Lifecycle.Event event, Object target) {

try {

switch (mCallType) {

case CALL\_TYPE\_NO\_ARG:

mMethod.invoke(target);//没有参数的

break;

case CALL\_TYPE\_PROVIDER:

mMethod.invoke(target, source);//一个参数的：LifecycleOwner

break;

case CALL\_TYPE\_PROVIDER\_WITH\_EVENT:

mMethod.invoke(target, source, event);//两个参数的：LifecycleOwner，Event

break;

}

}

...

}

...

}