在基类ComponentActivity中实现了LifecycleOwner接口：

private final LifecycleRegistry mLifecycleRegistry = new LifecycleRegistry(this);

@NonNull

@Override

public Lifecycle getLifecycle() {

return mLifecycleRegistry;

}

LifecycleRegistry继承了Lifecycle，实现了对Activity的状态标记和修改：

①标记状态

@Deprecated

@MainThread

public void markState(@NonNull State state) {

enforceMainThreadIfNeeded("markState");

setCurrentState(state);

}

②修改状态

@MainThread

public void setCurrentState(@NonNull State state) {

enforceMainThreadIfNeeded("setCurrentState");

moveToState(state);

}

public void handleLifecycleEvent(@NonNull Lifecycle.Event event) {

enforceMainThreadIfNeeded("handleLifecycleEvent");

moveToState(event.getTargetState());

}

接下来介绍moveToState的实现，代码如下：

private void moveToState(State next) {

if (mState == next) {

return;

}

mState = next;

if (mHandlingEvent || mAddingObserverCounter != 0) {

mNewEventOccurred = true;

// we will figure out what to do on upper level.

return;

}

mHandlingEvent = true;

sync();

mHandlingEvent = false;

}

其中sync方法实现如下：

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()) {

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);

}

Map.Entry<LifecycleObserver, ObserverWithState> newest = mObserverMap.newest();

if (!mNewEventOccurred && newest != null

&& mState.compareTo(newest.getValue().mState) > 0) {

forwardPass(lifecycleOwner);

}

}

mNewEventOccurred = false;

}

以及静态内部类ObserverWithState实现：

static class ObserverWithState {

State mState;

LifecycleEventObserver mLifecycleObserver;

ObserverWithState(LifecycleObserver observer, State initialState) {

//处理LifecycleEventObserver

mLifecycleObserver = Lifecycling.lifecycleEventObserver(observer);

mState = initialState;

}

void dispatchEvent(LifecycleOwner owner, Event event) {

State newState = event.getTargetState();

mState = min(mState, newState);

mLifecycleObserver.onStateChanged(owner, event);

mState = newState;

}

}

在Lifecycling的lifecycleEventObserver中对LifecycleObserver作了分类处理：

@NonNull

static LifecycleEventObserver lifecycleEventObserver(Object object) {

boolean isLifecycleEventObserver = object instanceof LifecycleEventObserver;

boolean isFullLifecycleObserver = object instanceof FullLifecycleObserver;

if (isLifecycleEventObserver && isFullLifecycleObserver) {

return new FullLifecycleObserverAdapter((FullLifecycleObserver) object,

(LifecycleEventObserver) object);

}

if (isFullLifecycleObserver) {

return new FullLifecycleObserverAdapter((FullLifecycleObserver) object, null);

}

if (isLifecycleEventObserver) {

return (LifecycleEventObserver) object;

}

final Class<?> klass = object.getClass();

int type = getObserverConstructorType(klass);

if (type == GENERATED\_CALLBACK) {

List<Constructor<? extends GeneratedAdapter>> constructors =

sClassToAdapters.get(klass);

if (constructors.size() == 1) {

GeneratedAdapter generatedAdapter = createGeneratedAdapter(

constructors.get(0), object);

return new SingleGeneratedAdapterObserver(generatedAdapter);

}

GeneratedAdapter[] adapters = new GeneratedAdapter[constructors.size()];

for (int i = 0; i < constructors.size(); i++) {

adapters[i] = createGeneratedAdapter(constructors.get(i), object);

}

return new CompositeGeneratedAdaptersObserver(adapters);

}

return new ReflectiveGenericLifecycleObserver(object);

}

其中getObserverConstructorType的处理如下：

private static int getObserverConstructorType(Class<?> klass) {

Integer callbackCache = sCallbackCache.get(klass);

if (callbackCache != null) {

return callbackCache;

}

int type = resolveObserverCallbackType(klass);

sCallbackCache.put(klass, type);

return type;

}

resolveObserverCallbackType(此方法处理Observer的返回值类型)中的处理如下：

private static int resolveObserverCallbackType(Class<?> klass) {

// anonymous class bug:35073837

if (klass.getCanonicalName() == null) {

return REFLECTIVE\_CALLBACK;

}

Constructor<? extends GeneratedAdapter> constructor = generatedConstructor(klass);

if (constructor != null) {

sClassToAdapters.put(klass, Collections

.<Constructor<? extends GeneratedAdapter>>singletonList(constructor));

return GENERATED\_CALLBACK;

}

boolean hasLifecycleMethods = ClassesInfoCache.sInstance.hasLifecycleMethods(klass);

if (hasLifecycleMethods) {

return REFLECTIVE\_CALLBACK;

}

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

List<Constructor<? extends GeneratedAdapter>> adapterConstructors = null;

if (isLifecycleParent(superclass)) {

if (getObserverConstructorType(superclass) == REFLECTIVE\_CALLBACK) {

return REFLECTIVE\_CALLBACK;

}

adapterConstructors = new ArrayList<>(sClassToAdapters.get(superclass));

}

for (Class<?> intrface : klass.getInterfaces()) {

if (!isLifecycleParent(intrface)) {

continue;

}

if (getObserverConstructorType(intrface) == REFLECTIVE\_CALLBACK) {

return REFLECTIVE\_CALLBACK;

}

if (adapterConstructors == null) {

adapterConstructors = new ArrayList<>();

}

adapterConstructors.addAll(sClassToAdapters.get(intrface));

}

if (adapterConstructors != null) {

sClassToAdapters.put(klass, adapterConstructors);

return GENERATED\_CALLBACK;

}

return REFLECTIVE\_CALLBACK;

}

其中对是否包含Lifecycle作了判断处理并返回REFLECTIVE\_CALLBACK类型，以及对实现类是否包含Lifecycle也作了相应处理。

/\*\*

\* 当前klass类是否包含Lifecycle的方法判断

\*/

boolean hasLifecycleMethods = ClassesInfoCache.sInstance.hasLifecycleMethods(klass);

if (hasLifecycleMethods) {

return REFLECTIVE\_CALLBACK;

}

/\*\*

\*对klacc父类是否包含Lifecycle的判断

\*/

if (isLifecycleParent(superclass)) {

if (getObserverConstructorType(superclass) == REFLECTIVE\_CALLBACK) {

return REFLECTIVE\_CALLBACK;

}

adapterConstructors = new ArrayList<>(sClassToAdapters.get(superclass));

}

//对klass实现的所有接口进行遍历

for (Class<?> intrface : klass.getInterfaces()) {

if (!isLifecycleParent(intrface)) {

continue;

}

if (getObserverConstructorType(intrface) == REFLECTIVE\_CALLBACK) {

return REFLECTIVE\_CALLBACK;

}

if (adapterConstructors == null) {

adapterConstructors = new ArrayList<>();

}

adapterConstructors.addAll(sClassToAdapters.get(intrface));

}

/\*\*

\*对adapterConstructors是否为空作了判断，如果不为空，则返回GENERATED\_CALLBACK类型

\*/

if (adapterConstructors != null) {

sClassToAdapters.put(klass, adapterConstructors);

return GENERATED\_CALLBACK;

}

判断是否包含Lifecycle总结：

1.是否包含基本的Lifecycle构造器判断构造器是否为空来实现

Constructor<? extends GeneratedAdapter> constructor = generatedConstructor(klass);

2.当前类klass是包含Lifecycle的方法

boolean hasLifecycleMethods = ClassesInfoCache.sInstance.hasLifecycleMethods(klass);

3.判断klass父类是否为Lifecycle

if (isLifecycleParent(superclass)) {

if (getObserverConstructorType(superclass) == REFLECTIVE\_CALLBACK) {

return REFLECTIVE\_CALLBACK;

}

adapterConstructors = new ArrayList<>(sClassToAdapters.get(superclass));

}

//获得构造器类型

private static int getObserverConstructorType(Class<?> klass) {

Integer callbackCache = sCallbackCache.get(klass);

if (callbackCache != null) {

return callbackCache;

}

int type = resolveObserverCallbackType(klass);

sCallbackCache.put(klass, type);

return type;

}

//处理返回类型

private static int resolveObserverCallbackType(Class<?> klass) {

// anonymous class bug:35073837

if (klass.getCanonicalName() == null) {

return REFLECTIVE\_CALLBACK;

}

Constructor<? extends GeneratedAdapter> constructor = generatedConstructor(klass);

if (constructor != null) {

sClassToAdapters.put(klass, Collections

.<Constructor<? extends GeneratedAdapter>>singletonList(constructor));

return GENERATED\_CALLBACK;

}

boolean hasLifecycleMethods = ClassesInfoCache.sInstance.hasLifecycleMethods(klass);

if (hasLifecycleMethods) {

return REFLECTIVE\_CALLBACK;

}

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

List<Constructor<? extends GeneratedAdapter>> adapterConstructors = null;

if (isLifecycleParent(superclass)) {

if (getObserverConstructorType(superclass) == REFLECTIVE\_CALLBACK) {

return REFLECTIVE\_CALLBACK;

}

adapterConstructors = new ArrayList<>(sClassToAdapters.get(superclass));

}

for (Class<?> intrface : klass.getInterfaces()) {

if (!isLifecycleParent(intrface)) {

continue;

}

if (getObserverConstructorType(intrface) == REFLECTIVE\_CALLBACK) {

return REFLECTIVE\_CALLBACK;

}

if (adapterConstructors == null) {

adapterConstructors = new ArrayList<>();

}

adapterConstructors.addAll(sClassToAdapters.get(intrface));

}

if (adapterConstructors != null) {

sClassToAdapters.put(klass, adapterConstructors);

return GENERATED\_CALLBACK;

}

return REFLECTIVE\_CALLBACK;

}

private static boolean isLifecycleParent(Class<?> klass) {

return klass != null && LifecycleObserver.class.isAssignableFrom(klass);

}

Lifecycling的lifecycleEventObserver的处理总结如下：

1.对是否为FullLifecycleObserver的处理。

2.对是否为LifecycleEventObserver的处理。此处在应用端可通过实现addObserver接口来监听生命周期反馈，最终来实现项目中的基本业务需求。

3.对于Observer构造方法类型作了区分，如果为GENERATED\_CALLBACK类型，则进一步区分构造方法数量是否为1，如果等于1，则返回SingleGeneratedAdapterObserver对象，不为1，则返回CompositeGeneratedAdaptersObserver。

4.如果以上条件均不满足则返回ReflectiveGenericLifecycleObserver，此处通过查找方法注解，类实现如下：

class ReflectiveGenericLifecycleObserver implements LifecycleEventObserver {

private final Object mWrapped;

private final CallbackInfo mInfo;

ReflectiveGenericLifecycleObserver(Object wrapped) {

mWrapped = wrapped;

mInfo = ClassesInfoCache.sInstance.getInfo(mWrapped.getClass());

}

@Override

public void onStateChanged(@NonNull LifecycleOwner source, @NonNull Event event) {

mInfo.invokeCallbacks(source, event, mWrapped);

}

}

其中调用ClassInfoCache的invokeCallbacks来实现，代码如下：

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

invokeMethodsForEvent(mEventToHandlers.get(event), source, event, target);

invokeMethodsForEvent(mEventToHandlers.get(Lifecycle.Event.ON\_ANY), source, event,

target);

}

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--) {

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

}

}

}

最终调用了MethodReference的invokeCallback方法，具体实现如下：

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

//noinspection TryWithIdenticalCatches

try {

switch (mCallType) {

case CALL\_TYPE\_NO\_ARG:

mMethod.invoke(target);

break;

case CALL\_TYPE\_PROVIDER:

mMethod.invoke(target, source);

break;

case CALL\_TYPE\_PROVIDER\_WITH\_EVENT:

mMethod.invoke(target, source, event);

break;

}

} catch (InvocationTargetException e) {

throw new RuntimeException("Failed to call observer method", e.getCause());

} catch (IllegalAccessException e) {

throw new RuntimeException(e);

}

}