Jetpack-Lifecycle 笔记

> 源码版本 **lifecycle 2.6.1**

1. 认识 Lifecycle

> 👍 Lifecycle 是多个 Jetpack 组件的基础，为 Android 的生命周期定义一套标准的行为模式。从 Android 应用开发者的角度来看，Lifecycle 框架的主要作用是简化实现生命周期感知型组件的复杂度。

假设有这么一个需求：开发一个定位组件，要求页面可见时连接定位服务来获取定位信息更新 UI，页面不可见时断开连接。在传统的实现方式中，需要手动从宿主（Activity、Fragment）中将生命周期事件分发到组件内部：

`LocationComponent.kt`

```kotlin
class LocationComponent(
    private val context: Context,
    private val callback: (Location) -> Unit
) {
    fun start() {
        // connect to system location service
    }

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

`MyActivity.kt` （宿主）

```kotlin
class MyActivity : AppCompatActivity() {
    private lateinit var locationComponent: LocationComponent

    override fun onCreate(...) {
        locationComponent = LocationComponent(this) { location ->
            // update UI
        }
```

```kotlin
    }

    public override fun onStart() {
        super.onStart()
        locationComponent.start()
    }

    public override fun onStop() {
        super.onStop()
        locationComponent.stop()
    }
}
```

在真实的项目中，往往有多个组件需要感知宿主的生命周期，同样需要手动在宿主生命周期回调中注入代码，这会导致如 `onStart()`、 `onStop()` 中包含大量代码，难以维护

此外，以定位组件为例，如果在 `locationComponent.start()` 之前需要执行耗时操作，如检查某种配置，那么就无法保证 `locationComponent.start()` 会在 `onStop()` 之前执行，比如：

```kotlin
class MyActivity : AppCompatActivity() {
    private lateinit var locationComponent: LocationComponent

    override fun onCreate(...) {
        locationComponent = LocationComponent(this) { location ->
            // update UI
        }
    }

    public override fun onStart() {
        super.onStart()
        Util.checkUserStatus { result ->
            // 如果该回调在 Activity onStop() 之后执行会怎么样
            if (result) {
                locationComponent.start()
            }
        }
    }

    public override fun onStop() {
        super.onStop()
        locationComponent.stop()
    }
}
```

回想初衷，之所以将定位组件的方法暴露给宿主调用，是因为组件自身无法感知宿主的生命周期。借助 Lifecycle 框架改进一下定位组件：

```
class LocationComponent(
    private val context: Context,
    private val callback: (Location) -> Unit
): DefaultLifecycleObserver {
    override fun onStart(owner: LifecycleOwner) {
        start()
    }

    override fun onStop(owner: LifecycleOwner) {
        stop()
    }

    private fun start() {
        // connect to system location service
    }

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

改进后，定位组件将原先暴露给宿主的方法收敛到组件内部，并且宿主不需要直接参与调整定位组件的生命周期：

```
class MyActivity : AppCompatActivity() {
    private lateinit var locationComponent: LocationComponent

    override fun onCreate(...) {
        locationComponent = LocationComponent(this) { location ->
            // update UI
        }
        lifecycle.addObserver(locationComponent)
    }
}
```

对于第二个问题，在 `locationComponent.start()` 之前需要执行耗时操作，如检查某种配置，则无法保证 `locationComponent.start()` 会在 `onStop()` 之前执行，Lifecycle 提供了查询宿主当前状态的方法：

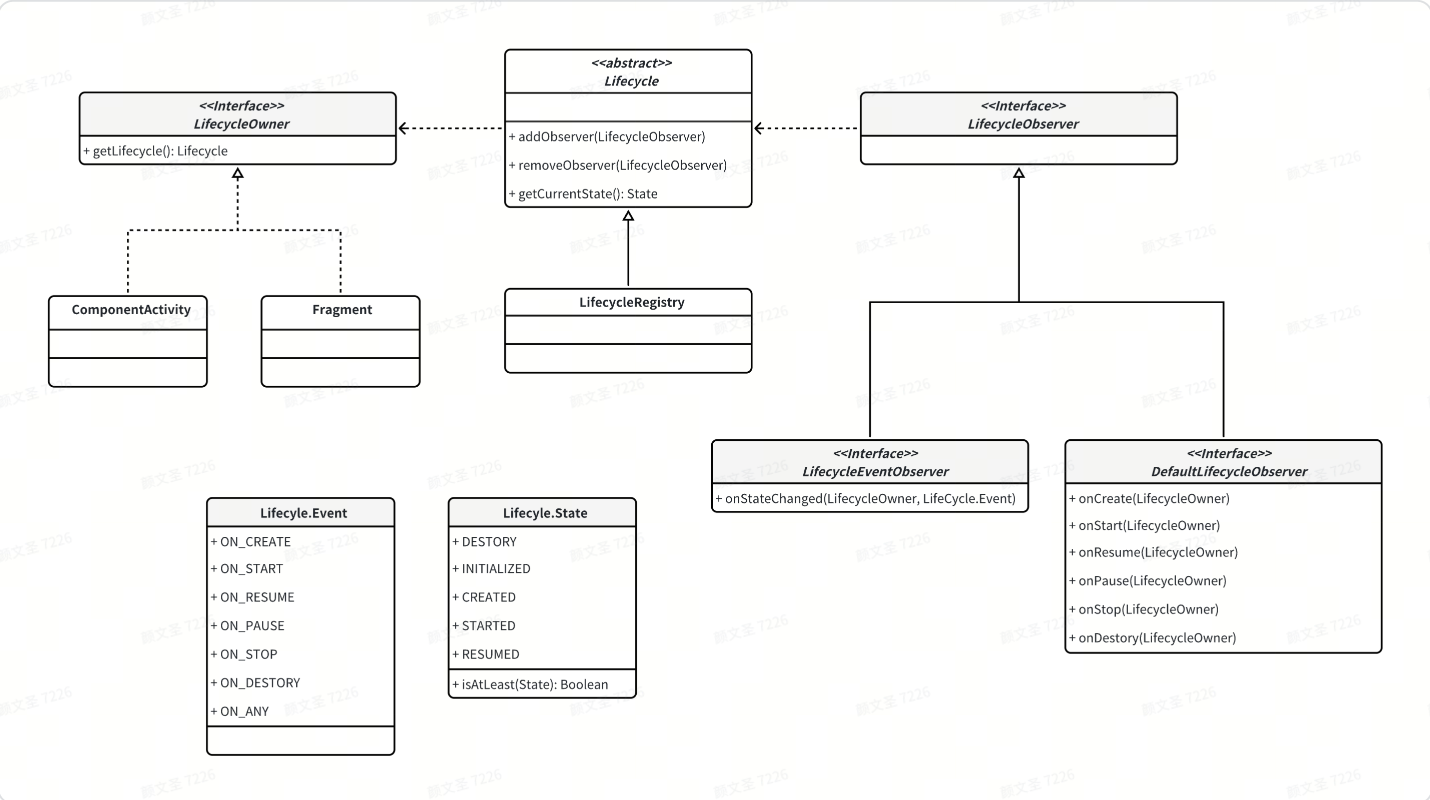
```kotlin
public abstract class Lifecycle {
    // 宿主当前状态
    public abstract val currentState: State
}
```

> 🤔 啥时候取消注册观察者?

Lifecycle 的合理使用可赋予我们的代码显著的优点:

- 无需在生命周期提供者（Activity、Fragment）生命周期回调里写大量代码，即可实现对生命周期的监听处理，即解耦又可维护
- Lifecycle 的实现类 LifecycleRegister 不会直接持有生命周期提供者（Activity、Fragment）的强引用，而是以弱引用的形式存在，避免内存泄漏问题

2. Lifecycle 设计思想



Lifecycle 整体上采用了观察者模式，核心 API 有 `LifecycleObserver`、`LifecycleOwner`、`Lifecycle`

- `LifecycleObserver`：观察者，自定义生命周期感知型组件需要间接实现该接口以观察宿主的生命周期

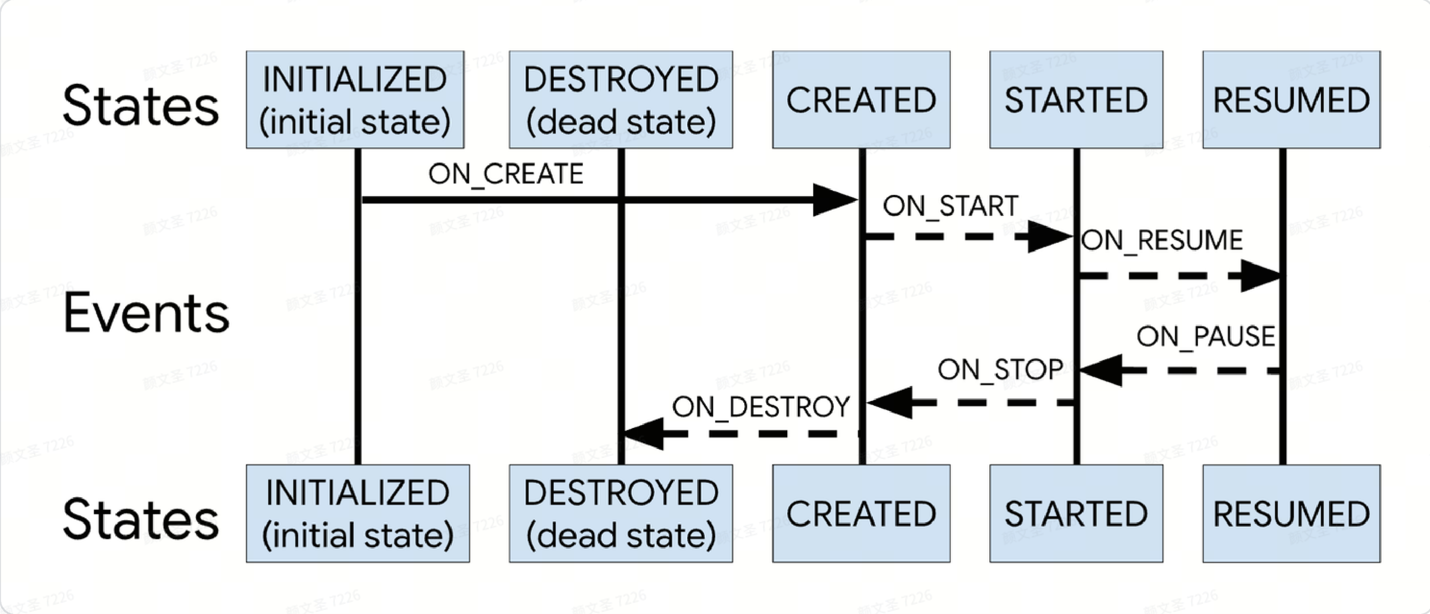
- `LifecycleOwner`：被观察者，宿主需要实现该接口，并将生命周期分发给 `Lifecycle`，从而间接分发给观察者
- `Lifecyle`：定义了生命周期的标准行为模式，属于 Lifecycle 框架的核心类，实现类 `LifecycleRegister`

3. Lifecycle 实现分析

3.1 Lifecycle

```
public abstract class Lifecycle {
    // 添加观察者
    @MainThread
    public abstract fun addObserver(observer: LifecycleObserver)

    // 移除观察者
    @MainThread
    public abstract fun removeObserver(observer: LifecycleObserver)

    // 获取宿主当前生命周期状态
    @get:MainThread
    public abstract val currentState: State

    // 生命周期事件
    public enum class Event {
        ON_CREATE,
        ON_START,
        ON_RESUME,
        ON_PAUSE,
        ON_STOP,
        ON_DESTROY,
        ON_ANY;
    }

    // 生命周期状态
    public enum class State {
        DESTROYED,
        INITIALIZED,
        CREATED,
        STARTED,
        RESUMED;
    }
}
```

`Lifecycle` 将生命周期信息抽象为 `Event` 与 `State`。从有限状态机的角度理解，`State` 代表着生命周期的状态，`Event` 代表着不同状态之间流转所需的事件，对应关系如下图。



与上图对应，`Lifecycle.Event` 定义了状态升级、降级的四个方法：

```
public companion object {
    // 从状态 state 降级所需的事件
    @JvmStatic
    public fun downFrom(state: State): Event? {
        return when (state) {
            State.CREATED -> ON_DESTROY
            State.STARTED -> ON_STOP
            State.RESUMED -> ON_PAUSE
            else -> null
        }
    }

    // 降级至状态 state 所需的事件
    @JvmStatic
    public fun downTo(state: State): Event? {
        return when (state) {
            State.DESTROYED -> ON_DESTROY
            State.CREATED -> ON_STOP
            State.STARTED -> ON_PAUSE
            else -> null
        }
    }

    // 从状态 state 升级所需的事件
    @JvmStatic
```

```
    public fun upFrom(state: State): Event? {
        return when (state) {
            State.INITIALIZED -> ON_CREATE
            State.CREATED -> ON_START
            State.STARTED -> ON_RESUME
            else -> null
        }
    }

    // 升级至状态 state 所需的事件
    @JvmStatic
    public fun upTo(state: State): Event? {
        return when (state) {
            State.CREATED -> ON_CREATE
            State.STARTED -> ON_START
            State.RESUMED -> ON_RESUME
            else -> null
        }
    }
}
```

3.2 LifecycleObserver

```
public interface LifecycleObserver
```

`LifecycleObserver` 是一个空接口，不建议直接实现该接口，而是通过实现 `DefaultLifecycleObserver` 或 `LifecycleEventObserver` 来观察宿主的生命周期。

`DefaultLifecycleObserver.kt`

```
public interface DefaultLifecycleObserver : LifecycleObserver {
    public fun onCreate(owner: LifecycleOwner) {}

    public fun onStart(owner: LifecycleOwner) {}

    public fun onResume(owner: LifecycleOwner) {}

    public fun onPause(owner: LifecycleOwner) {}

    public fun onStop(owner: LifecycleOwner) {}

```

```
    public fun onDestroy(owner: LifecycleOwner) {}
}
```

`LifecycleEventObserver.kt`

```
public fun interface LifecycleEventObserver : LifecycleObserver {
    public fun onStateChanged(source: LifecycleOwner, event: Lifecycle.Event)
}
```

从 `Lifecycle` 声明的 `addObserver(LifecyleObserver)` 可知，添加的观察者都是 `LifycyleObserver` 类型的，且 `DefaultLifecycleObserver` 和 `LifecycleEventObserver` 的实现是不同的，那么 `Lifecycle` 通知生命周期事件时是如何区分这些观察者的具体类型呢？关于这一点，会在 `LifecycleRegister` 中分析。

3.3 LifecycleOwner

顾名思义，Lifecycle 的持有者。LifecycleOwner 接口中只有一个方法，由于对外提供 Lifecycle

```
public interface LifecycleOwner {
    public val lifecycle: Lifecycle
}
```

ComponentActivity 和 Fragment 均实现了该接口：

```
// ComponentActivity.java
public class ComponentActivity extends Activity implements LifecycleOwner{
    private LifecycleRegistry mLifecycleRegistry = new LifecycleRegistry(this);

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

// Fragment.java
public class Fragment implements LifecycleOwner{
    LifecycleRegistry mLifecycleRegistry;

    @Override
```

```
    public Lifecycle getLifecycle() {
        return mLifecycleRegistry;
    }
}
```

3.4 LifecycleRegistry

对于被观察者，常规玩法是：定义一个集合来存放所有观察者，事件产生时，迭代集合调用观察者对应的回调方法。但是在这里，状态管理涉及到的逻辑会更复杂，还需要考虑这些问题：

- 迭代集合调用观察者对应的回调方法时，可能需要增删集合中的观察者对象，所以集合需要支持**迭代时增删元素**
- 迭代集合调用观察者对应的回调方法时，新加入的观察者该如何处理
- 迭代集合调用观察者对应的回调方法时，删除观察者又该如何处理

3.4.1 存储观察者

`LifecycleRegister` 中，存放观察者的集合：

```
private var observerMap = FastSafeIterableMap<LifecycleObserver,
ObserverWithState>(
```

3.4.1.1 FastSafeIterableMap

`FastSafeIterableMap` 继承 `SafeIterableMap`，重写了三个方法

```
public class FastSafeIterableMap<K, V> extends SafeIterableMap<K, V> {

    // 套了一层 HashMap 空间换时间，使得查找操作更快而
    private final HashMap<K, Entry<K, V>> mHashMap = new HashMap<>();

    @Override
    protected Entry<K, V> get(K k) {
        return mHashMap.get(k);
    }

    @Override
    public V putIfAbsent(@NonNull K key, @NonNull V v) {
        Entry<K, V> current = get(key);
        if (current != null) {
            return current.mValue;
```

```java
        }
        mHashMap.put(key, put(key, v));
        return null;
    }

    @Override
    public V remove(@NonNull K key) {
        V removed = super.remove(key);
        mHashMap.remove(key);
        return removed;
    }

    public boolean contains(K key) {
        return mHashMap.containsKey(key);
    }
}
```

对于 `SafeIterableMap`，只需简单了解即可：

- 内部是双向链表实现，尾插法
- 能提供正序、逆序的迭代器
- 迭代器支持迭代时增删元素

再回到 `observerMap`，它有一个特性：依次添加 `observer0`、`observer1`、`observer2`，它可以保证在任意时刻 `observer0.state` >= `observer1.state` >= `observer2.state`，理解这一点非常重要，严格要求这种关系，可以在 `O(1)` 的时间内判断所有的观察者们是否同步完成：

```kotlin
private val isSynced: Boolean
    get() {
        if (observerMap.size() == 0) {
            return true
        }
        val eldestObserverState = observerMap.eldest()!!.value.state
        val newestObserverState = observerMap.newest()!!.value.state
        return eldestObserverState == newestObserverState && state == newestObserverState
    }
```

同样的，在同步所有观察者状态时，为升级、降级提供了两个方法，`forwardPass()`、`backwardPass()` 分别采用不用的迭代顺序同样是为了保证观察者集合中 State 的单调性

3.4.1.2 ObserverWithState

`ObserverWithState.kt`

```kotlin
internal class ObserverWithState(observer: LifecycleObserver?, initialState: State) {
    var state: State // 标记观察者的状态
    var lifecycleObserver: LifecycleEventObserver

    init {
        lifecycleObserver = Lifecycling.lifecycleEventObserver(observer!!)
        state = initialState
    }

    // 统一的回调方法
    fun dispatchEvent(owner: LifecycleOwner?, event: Event) {
        val newState = event.targetState
        state = min(state, newState)
        lifecycleObserver.onStateChanged(owner!!, event)
        state = newState
    }
}
```

其中 `Lifecycling.lifecycleEventObserver(observer!!)` 解决了观察者类型不一致的问题

```kotlin
public fun lifecycleEventObserver(`object`: Any): LifecycleEventObserver {
    val isLifecycleEventObserver = `object` is LifecycleEventObserver
    val isDefaultLifecycleObserver = `object` is DefaultLifecycleObserver
    // 同时实现了 LifecycleEventObserver 和 DefaultLifecycleObserver
    // 返回包装者 DefaultLifecycleObserverAdapter
    if (isLifecycleEventObserver && isDefaultLifecycleObserver) {
        return DefaultLifecycleObserverAdapter(
            `object` as DefaultLifecycleObserver,
            `object` as LifecycleEventObserver
        )
    }
    // 只实现了 DefaultLifecycleObserver，返回包装者 DefaultLifecycleObserverAdapter
    if (isDefaultLifecycleObserver) {
        return DefaultLifecycleObserverAdapter(`object` as DefaultLifecycleObserver, null)
    }
    // 只实现了 LifecycleEventObserver，直接返回
    if (isLifecycleEventObserver) {
        return `object` as LifecycleEventObserver
```

```
    }
    // 通过反射处理自定义的回调实现
    val klass: Class<*> = `object`.javaClass
    val type = getObserverConstructorType(klass)
    if (type == GENERATED_CALLBACK) {
        val constructors = classToAdapters[klass]!!
        if (constructors.size == 1) {
            val generatedAdapter = createGeneratedAdapter(
                constructors[0], `object`
            )
            return SingleGeneratedAdapterObserver(generatedAdapter)
        }
        val adapters: Array<GeneratedAdapter> = Array(constructors.size) { i ->
            createGeneratedAdapter(constructors[i], `object`)
        }
        return CompositeGeneratedAdaptersObserver(adapters)
    }
    return ReflectiveGenericLifecycleObserver(`object`)
}
```

看下 `DefaultLifecycleObserverAdapter` 做了什么

```
internal class DefaultLifecycleObserverAdapter(
    private val defaultLifecycleObserver: DefaultLifecycleObserver,
    private val lifecycleEventObserver: LifecycleEventObserver?
) : LifecycleEventObserver {
    override fun onStateChanged(source: LifecycleOwner, event: Lifecycle.Event) {
        when (event) {
            Lifecycle.Event.ON_CREATE -> defaultLifecycleObserver.onCreate(source)
            Lifecycle.Event.ON_START -> defaultLifecycleObserver.onStart(source)
            Lifecycle.Event.ON_RESUME -> defaultLifecycleObserver.onResume(source)
            Lifecycle.Event.ON_PAUSE -> defaultLifecycleObserver.onPause(source)
            Lifecycle.Event.ON_STOP -> defaultLifecycleObserver.onStop(source)
            Lifecycle.Event.ON_DESTROY -> defaultLifecycleObserver.onDestroy(source)
            Lifecycle.Event.ON_ANY ->
                throw IllegalArgumentException("ON_ANY must not been send by anybody")
        }
        lifecycleEventObserver?.onStateChanged(source, event)
```

```
    }
}
```

3.4.2 同步生命周期

总的来说，调用观察者回调方法的时机有两个：

- 添加观察者时，需要将观察者的生命周期状态同步至宿主当前状态
- 宿主生命周期变化时，需要将所有观察者的生命周期状态同步至宿主当前状态

如果只有以上两种场景，那么 `LifecycleRegistry` 的实现将会很简单，实际上还需要考虑一下场景：

> 调用观察者回调方法时，又添加了新的观察者，如何避免不必要的同步？如何保证观察者集合中 State 的单调性？

```kotlin
// 以下三个变量，用于避免不必要的同步
private var addingObserverCounter = 0 // 正在添加观察者的数量
private var handlingEvent = false // 是否正在处理宿主生命周期变化事件
private var newEventOccurred = false // 是否是宿主新生命周期事件到达

// 用于保证观察者集合中 State 的单调性
private var parentStates = ArrayList<State>()
```

对于 `parentStates`，只提供了入栈、出栈操作：

```kotlin
private fun popParentState() {
    parentStates.removeAt(parentStates.size - 1)
}

private fun pushParentState(state: State) {
    parentStates.add(state)
}
```

关于 `parentStates` 中的 parent 是谁，这里解释一下：如果在调用观察者 `Observer1` 的 `onStart()` 时，`onStart()` 内部又添加了新的观察者 `NewObserver`，此时相对于 `NewObserver` 来说，parent 就是 `Observer1`。保存 `parentStates` 的作用在于，此时 `NewObserver` 的状态必须同步至 `parentState` 而不是宿主的状态，才能保证观察者集合中 State 的单调性。

所以，在每次调用观察者回调方法之前，都会进行入栈操作；观察者回调方法调用结束，进行出栈操作：

```
pushParentState(event.targetState)
observer.dispatchEvent(lifecycleOwner, event)
popParentState()
```

计算目标状态：

```
private fun calculateTargetState(observer: LifecycleObserver): State {
    // 获取前一个观察者的状态
    val map = observerMap.ceil(observer)
    val siblingState = map?.value?.state
    // parentState
    val parentState = if (parentStates.isNotEmpty()) parentStates[parentStates.size - 1] else null
    return min(min(state, siblingState), parentState)
}
```

3.4.2.1 添加观察者时同步生命周期

```
override fun addObserver(observer: LifecycleObserver) {
    enforceMainThreadIfNeeded("addObserver")
    val initialState = if (state == State.DESTROYED) State.DESTROYED else State.INITIALIZED
    val statefulObserver = ObserverWithState(observer, initialState) // 封装为 ObserverWithState
    val previous = observerMap.putIfAbsent(observer, statefulObserver)
    if (previous != null) { // 重复添加，直接返回
        return
    }
    val lifecycleOwner = lifecycleOwner.get()?: return // 宿主已销毁，直接返回
    val isReentrance = addingObserverCounter != 0 || handlingEvent // 是否重入
    var targetState = calculateTargetState(observer)
    addingObserverCounter++
    // 逐步将观察者状态同步至目标状态
    while (statefulObserver.state < targetState && observerMap.contains(observer)) {
        pushParentState(statefulObserver.state)
        val event = Event.upFrom(statefulObserver.state)
        statefulObserver.dispatchEvent(lifecycleOwner, event) // 调用观察者的回调方法
        popParentState()
        targetState = calculateTargetState(observer)
    }
```

```kotlin
    if (!isReentrance) {
        // 只在最外层同步所有观察者状态
        sync()
    }
    addingObserverCounter--
}
```

3.4.2.2 宿主生命周期变化时同步生命周期

```kotlin
private fun moveToState(next: State) {
    // 如果当前状态和接收到的状态一致，那么直接返回不做任何处理
    if (state == next) {
        return
    }
    state = next
    // 如果正在分发状态或者有观察者正在添加，则标记有新事件发生
    if (handlingEvent || addingObserverCounter != 0) {
        newEventOccurred = true
        return
    }
    // 标记正在处理事件
    handlingEvent = true
    // 同步所有观察者状态
    sync()
    // 还原标记
    handlingEvent = false
}

private fun sync() {
    val lifecycleOwner = lifecycleOwner.get()
    while (!isSynced) {
        newEventOccurred = false
        if (state < observerMap.eldest()!!.value.state) {
            backwardPass(lifecycleOwner) // 状态降级
        }
        val newest = observerMap.newest()
        if (!newEventOccurred && newest != null && state > newest.value.state) {
            forwardPass(lifecycleOwner) // 状态升级
        }
    }
    newEventOccurred = false
}
```

```

private fun backwardPass(lifecycleOwner: LifecycleOwner) {
    val descendingIterator = observerMap.descendingIterator() // 降序迭代器
    // 迭代所有观察者，如果有新事件发生，直接取消迭代，在sync()中重新决定如何迭代
    while (descendingIterator.hasNext() && !newEventOccurred) {
        val (key, observer) = descendingIterator.next()
        while (observer.state > state && !newEventOccurred && observerMap.contains(key)
        ) {
            val event = Event.downFrom(observer.state)
            pushParentState(event.targetState)
            observer.dispatchEvent(lifecycleOwner, event)
            popParentState()
        }
    }
}
```

参考资料

- 使用生命周期感知型组件处理生命周期
- Android Jetpack 开发套件 #1 Lifecycle：生命周期感知型组件的基础 - 掘金
- 【Jetpack】学穿:Lifecycle → 生命周期 (原理篇) - 掘金
- 理解 Lifecycle · @DaVinci42’s Blog