学穿 Jetpack Lifecycle → 生命周期

CoderPig 郭霖 2022-11-02 08:00 发表于江苏

/ 今日科技快讯 /

10月31日晚8点，天猫双11第一波售卖正式开启。销售1小时，102个品牌成交额过亿元。在102个首小时成交额过亿的品牌中，国货品牌占比过半，成为今年天猫双11一大亮点。黑鹿、挪客、牧高笛、森宝、未及、佳奇等国货品牌首小时成交额均超过去年11月1日全天。积木品牌未及更是只用10分钟就实现成交额超去年双11全周期。

/ 作者简介 /

本篇文章转自coder_pig的博客，文章主要分享了 Lifecycle 生命周期的原理解析，相信会对大家有所帮助！

原文地址：

> https://juejin.cn/post/7071144317636575262

/ Lifecycle核心思想 /

本质上是围绕着这两个设计模式进行的：

- 模板模式 → 定义算法骨架，对外开放扩展点，基于继承关系实现，子类重写父类抽象方法；
- 观察者模式 → 对象间定义一对多的依赖，当一个对象状态发生改变，依赖对象都会自动收到通知；

对这两种模式不了解的强烈建议看下笔者之前写的专栏：《把书读薄 | 设计模式之美》。

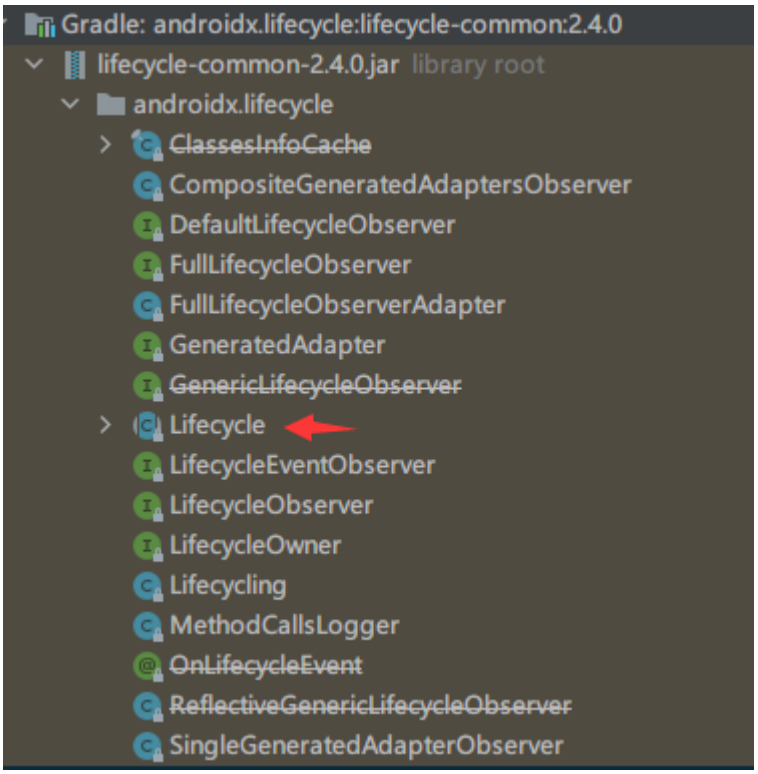
把书读薄 | 设计模式之美地址：

> https://juejin.cn/column/6969403852676136967

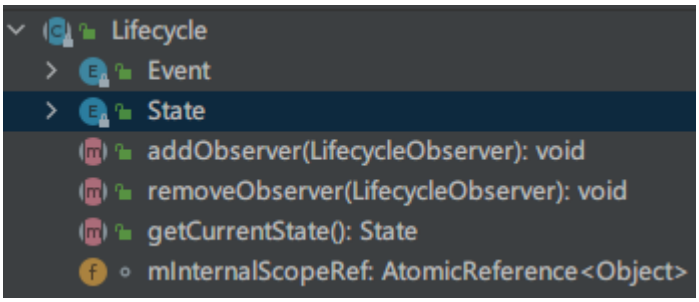
本节先肝下 Lifecycle 组件的两个库 lifecycle-common 和 lifecycle-runtime 的源码，了解实现原理，在肝 Activity、Fragment 中 Lifecycle 是如何发挥作用的。希望通过这节，能让你在实际开中能够有的放矢，放心大胆地用上 Lifecycle。

/ lifecycle-common源码解读 /

lifecycle-common 包中包含下述文件，挑着看：



① Lifecycle抽象类



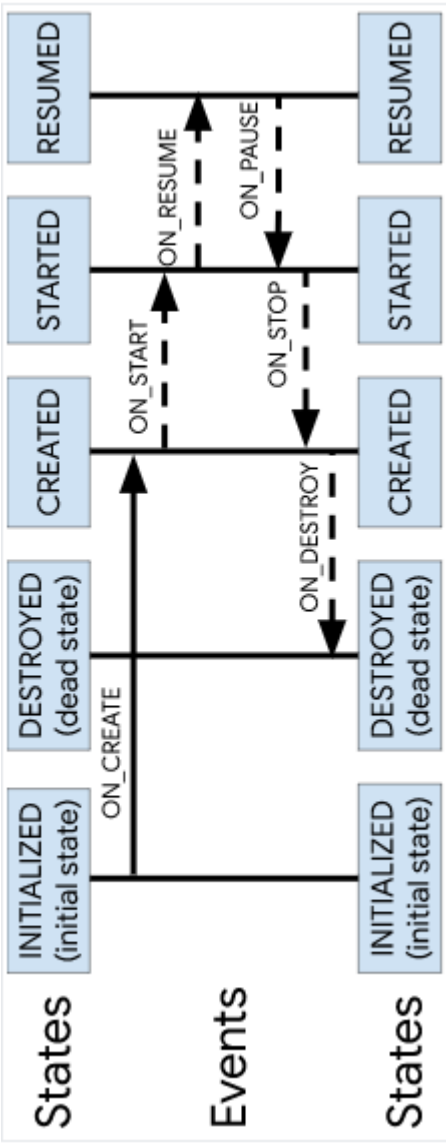
类中定义了生命周期事件和状态，先看 Event，枚举了7种生命周期事件：

ON_CREATE、ON_START、ON_RESUME、ON_PAUSE、ON_STOP、ON_DESTROY、ON_ANY

定义了状态升级、降级的四个方法：

downFrom()、downTo()、upFrom()、upTo()

这里的升降级，读者看了可能会有点懵，可以把之前那个图竖着看：



以 downFrom() 为例，从传入状态降级，返回对应 State：

```
@Nullable
public static Event downFrom(@NonNull State state) {
    switch (state) {
        case CREATED:
            return ON_DESTROY;
        case STARTED:
            return ON_STOP;
        case RESUMED:
            return ON_PAUSE;
        default:
            return null;
    }
}
```

从 CREATED 往下走，调用 OnDestory，从 STARTED 往下走，调用 onStop()，从 RESUME 往下走，调用 onPause()。

嘿嘿，是不是一下子就看懂了，再看下 downTo()：

```
public static Event downTo(@NonNull State state) {
    switch (state) {
        case DESTROYED:
            return ON_DESTROY;
        case CREATED:
            return ON_STOP;
        case STARTED:
            return ON_PAUSE;
        default:
            return null;
    }
}
```

往下走到 DESTORYED，调用 onDestory()，往下走到 CREATED，调用 onStop()，剩下两个也是类似~

然后还有个 getTargetState()。

```
public State getTargetState() {
    switch (this) {
        case ON_CREATE:
        case ON_STOP:
            return State.CREATED;
        case ON_START:
        case ON_PAUSE:
            return State.STARTED;
        case ON_RESUME:
            return State.RESUMED;
        case ON_DESTROY:
            return State.DESTROYED;
        case ON_ANY:
            break;
    }
    throw new IllegalArgumentException(this + " has no target state");
}
```

这个就更好理解了，调用 onCreate()、onStop() 会处于 CREATED 状态，其他同理。

说完 Event 说 State，更简单，定义了定义了5个状态的枚举值：

DESTROYED、INITIALIZED、CREATED、STARTED、RESUMED

以及一个判断状态是否相等的方法：

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

最后定义了三个抽象方法：

addObserver()、removeObserver()、getCurrentState()

可以把 Lifecycle 看作抽象被观察者，抽取出生命周期事件与状态，统一状态流转过程，并提供了增删观察者的抽象方法供具体被观察者实现。

② LifecycleObserver接口

空接口，类型标记，可看作抽象观察者。

```
public interface LifecycleObserver {

}
```

③ FullLifecycleObserver、LifecycleEventObserver接口

都继承 LifecycleObserver 接口，提供了两类不同的回调：

```
interface FullLifecycleObserver extends LifecycleObserver {

    void onCreate(LifecycleOwner owner);

    void onStart(LifecycleOwner owner);

    void onResume(LifecycleOwner owner);

    void onPause(LifecycleOwner owner);

    void onStop(LifecycleOwner owner);

    void onDestroy(LifecycleOwner owner);
}
```

定义了一些回调的抽象方法

一种是详细的生命周期回调，一种是有状态变化就回调，前者优先级大于后者。

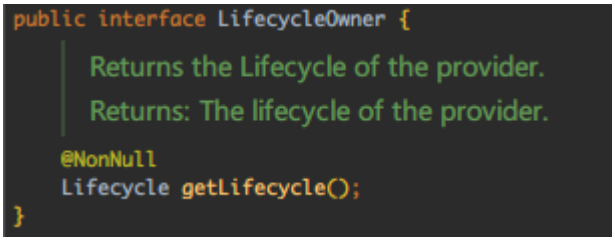
④ DefaultLifecycleObserver接口

继承 FullLifecycleObserver，这里用到了Java 8后才有的特性。接口声明默认方法，默认重写了回调方法，具体观察者按需实现关注的回调方法即可。

```
public interface DefaultLifecycleObserver extends FullLifecycleObserver {
   @Override default void onCreate(@NonNull LifecycleOwner owner) { }
   @Override default void onStart(@NonNull LifecycleOwner owner) { }
   @Override default void onResume(@NonNull LifecycleOwner owner) { }
   @Override default void onPause(@NonNull LifecycleOwner owner) { }
   @Override default void onStop(@NonNull LifecycleOwner owner) { }
   @Override default void onDestroy(@NonNull LifecycleOwner owner) { }
}
```

⑤ LifecycleOwner接口

提供一个获取 Lifecycle 的方法：



⑥ Lifecycling类

将传入的 LifecycleObserver 进行类型包装，生成一个新的 LifecycleEventObserver 实例，使得 Event 分发过程可以统一入口。直接关注 lifecycleEventObserver()：


```
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, lifecycleEventObserver: 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);
}
```

看下 FullLifecycleObserverAdapter：

```java
class FullLifecycleObserverAdapter implements LifecycleEventObserver {

    private final FullLifecycleObserver mFullLifecycleObserver;
    private final LifecycleEventObserver mLifecycleEventObserver;

    FullLifecycleObserverAdapter(FullLifecycleObserver fullLifecycleObserver,
            LifecycleEventObserver lifecycleEventObserver) {
        mFullLifecycleObserver = fullLifecycleObserver;
        mLifecycleEventObserver = lifecycleEventObserver;
    }

    @Override
    public void onStateChanged(@NonNull LifecycleOwner source, @NonNull Lifecycle.Event event) {
        switch (event) {
            case ON_CREATE:
                mFullLifecycleObserver.onCreate(source);
                break;
            case ON_START:
                mFullLifecycleObserver.onStart(source);
                break;
            case ON_RESUME:
                mFullLifecycleObserver.onResume(source);
                break;
            case ON_PAUSE:
                mFullLifecycleObserver.onPause(source);
                break;
            case ON_STOP:
                mFullLifecycleObserver.onStop(source);
                break;
            case ON_DESTROY:
                mFullLifecycleObserver.onDestroy(source);
                break;
            case ON_ANY:
                throw new IllegalArgumentException("ON_ANY must not been send by anybody");
        }
        if (mLifecycleEventObserver != null) {
            mLifecycleEventObserver.onStateChanged(source, event);
        }
    }
}
```

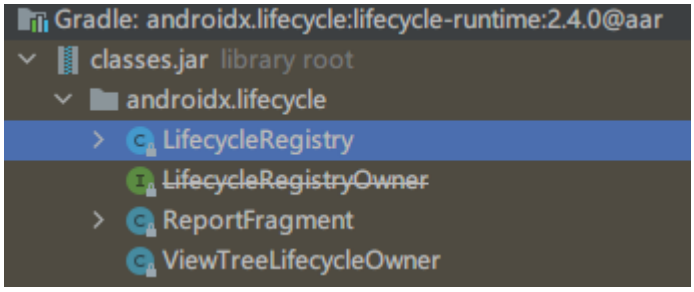
就是套了一层，保证先执行 FullLifecycleObserver 的回调，再执行 LifecycleEventObserver 的回调。

注解反射相关的暂且不看，基于 OnLifecycleEvent 注解方式进行回调，是面向基于 Java 7作为编译版本的平台，现在基本都 Java 8了，甚至有些玩 Compose 的都已经用上 Java 11了。注释也有说，只是为了兼容保留，后续会逐步废弃。

lifecycle-common 包看得差不多了，接下来看另一个~

/ lifecycle-runtime源码解读 /

lifecycle-runtime 包含下述四个文件：



① LifecycleRegistry类

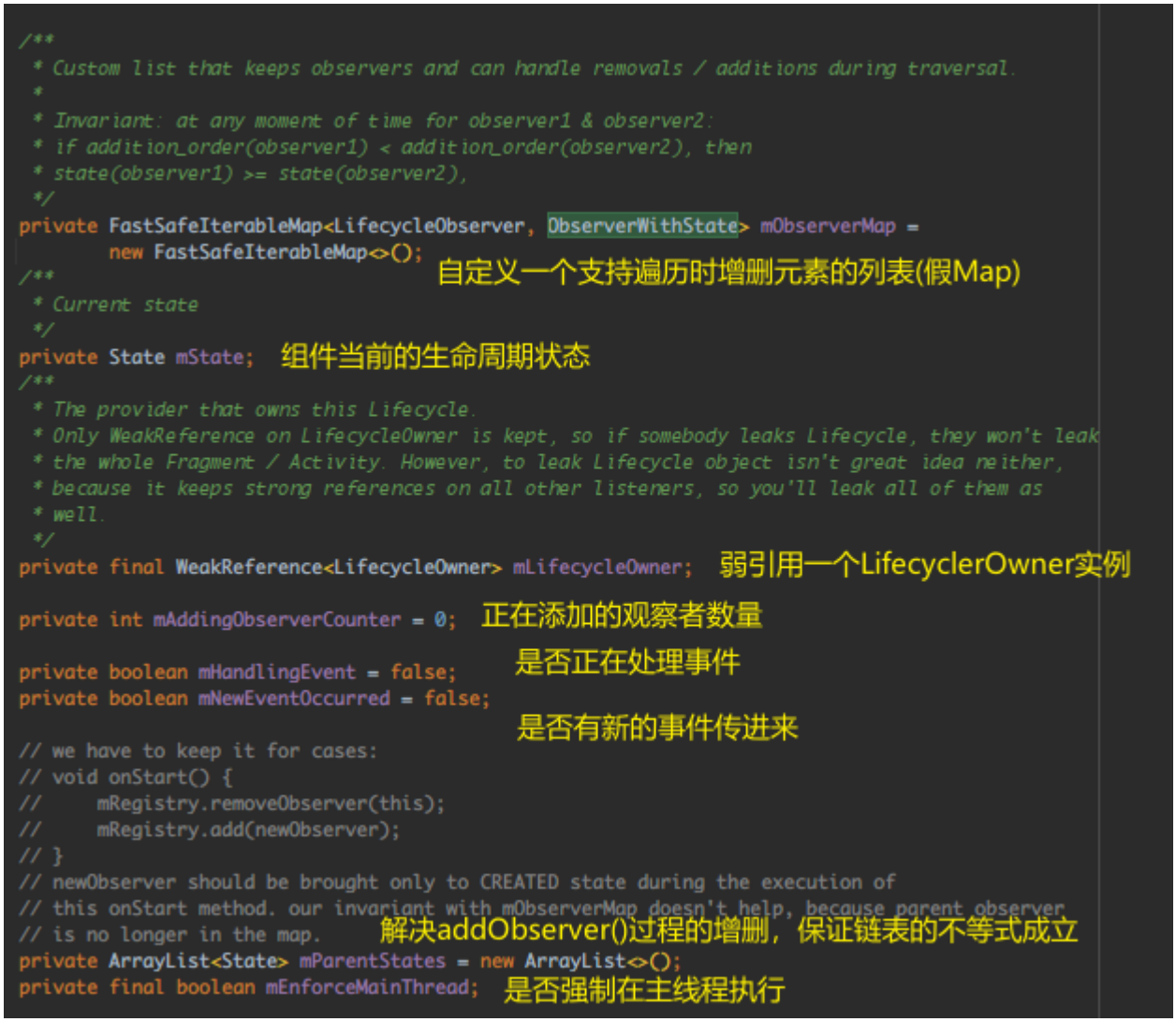
整个包里最重要的一个类，可看作具体被观察者，常规玩法都是：

定义一个集合，存所有观察者，事件产生时，迭代集合，调用观察者对应的回调方法。

但在这里，逻辑变得更复杂了，因为还涉及到了状态管理，还得考虑这些问题：

- ① 有事件产生迭代观察者集合时，可能增删集合中的观察者 → 集合需要支持迭代时增删元素 → 你像 ArrayList 就不行，for 迭代时移除元素会报 ConcurrentModificationException；
- ② 处理事件回调时，新加入的观察者该如何处理？该设置为什么状态？要不要也进行回调？
- ③ 如果移除观察者呢？状态要更新吗？还是直接忽略？等等...

心中埋下这些问题的种子，然后开始跟源码，先是这个支持迭代时增删元素的集合：




```
/**
 * Custom list that keeps observers and can handle removals / additions during traversal.
 *
 * Invariant: at any moment of time for observer1 & observer2:
 * if addition_order(observer1) < addition_order(observer2), then
 * state(observer1) >= state(observer2),
 */
private FastSafeIterableMap<LifecycleObserver, ObserverWithState> mObserverMap =
        new FastSafeIterableMap<>();
/**
 * Current state
 */
private State mState;
/**
 * The provider that owns this Lifecycle.
 * Only WeakReference on LifecycleOwner is kept, so if somebody leaks Lifecycle, they won't leak
 * the whole Fragment / Activity. However, to leak Lifecycle object isn't great idea neither,
 * because it keeps strong references on all other listeners, so you'll leak all of them as
 * well.
 */
private final WeakReference<LifecycleOwner> mLifecycleOwner;

private int mAddingObserverCounter = 0;

private boolean mHandlingEvent = false;
private boolean mNewEventOccurred = false;

// we have to keep it for cases:
// void onStart() {
//     mRegistry.removeObserver(this);
//     mRegistry.add(newObserver);
// }
// newObserver should be brought only to CREATED state during the execution of
// this onStart method. our invariant with mObserverMap doesn't help, because parent observer
// is no longer in the map.
private ArrayList<State> mParentStates = new ArrayList<>();
private final boolean mEnforceMainThread;
```

一眼就看到这个 FastSafeIterableMap，点开类，注释说到：

简陋版的 LinkedHashMap，支持遍历时的元素删除，比 SafeIterableMap占用更多内存，非线程安全。

他继承 SafeIterableMap（链表实现）类，并重写了这四个方法。

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

    private HashMap<K, Entry<K, V>> mHashMap = new HashMap<>();
                                另外定义一个HashMap存键值对
    @Override
    protected Entry<K, V> get(K k) { return mHashMap.get(k); }
                                方便快速根据Key索引值
    @Override
    public V putIfAbsent(@NonNull K key, @NonNull V v) {
        Entry<K, V> current = get(key);
        if (current != null) {                  不用链表慢慢找
            return current.mValue;
        }
        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;
    }

    Returns true if this map contains a mapping for the specified key.
    public boolean contains(K key) { return mHashMap.containsKey(key); }
```

空间换时间，就是套了一层 HashMap，使得查找起来会更快而已，接着看 SafeIterableMap。

```
@RestrictTo(RestrictTo.Scope.LIBRARY_GROUP_PREFIX)
public class SafeIterableMap<K, V> implements Iterable<Map.Entry<K, V>> {

    /WeakerAccess/ /* synthetic access */
    Entry<K, V> mStart;         链表起点
    private Entry<K, V> mEnd;链表终点
    // using WeakHashMap over List<WeakReference>, so we don't have to manually remove
    // WeakReferences that have null in them.  弱引用HashMap，键和值都可以为null
    private WeakHashMap<SupportRemove<K, V>, Boolean> mIterators = new WeakHashMap<>();
    private int mSize = 0;
```

定义了头、尾 Entry，存迭代器的 WeakHashMap，节点计数器，看下 Entry。

就是每个独立的节点，里面除了key，value外，还有前后节点的引用，继续看get() 和 put()：

单链表的常规操作了，而 putIfAbsent() 更简单，就是调 get()，拿到元素直接返回，拿不到 put() 插入元素。

回到关注点：遍历时删除元素，跟下 remove()：

```
public V remove(@NonNull K key) {
    Entry<K, V> toRemove = get(key);
    if (toRemove == null) {
        return null;
    }
    mSize--; 删除元素时，通知所有正在遍历的iterator
    if (!mIterators.isEmpty()) {
        for (SupportRemove<K, V> iter : mIterators.keySet()) {
            iter.supportRemove(toRemove);
        }
    }

    if (toRemove.mPrevious != null) {
        toRemove.mPrevious.mNext = toRemove.mNext;
    } else {
        mStart = toRemove.mNext;
    }

    if (toRemove.mNext != null) {
        toRemove.mNext.mPrevious = toRemove.mPrevious;
    } else {
        mEnd = toRemove.mPrevious;
    }
                                    修改链表的前后元素指向
    toRemove.mNext = null;
    toRemove.mPrevious = null;
    return toRemove.mValue;
}
```

跟下 supportRemove()：

```
@Override
public void supportRemove(@NonNull Entry<K, V> entry) {
    if (mExpectedEnd == entry && entry == mNext) {
        mNext = null;
        mExpectedEnd = null;          只有一个节点直接干掉
    }
最后节点等于移除节点，最后节点往前挪一个节点
    if (mExpectedEnd == entry) {
        mExpectedEnd = backward(mExpectedEnd);
    }
开始节点等于移除节点，开始节点往后挪一个节点
    if (mNext == entry) {
        mNext = nextNode();
    }
}
```

关于迭代器具体的添加和删除，提供三个具体实现类：

- AscendingIterator(升序)
- DescendingIterator(降序)
- IteratorWithAdditions(还支持添加元素)

就不去抠了，在调用 iterator() 获得迭代器时，都会把迭代器添加到集合中：

```
@NonNull
@Override
public Iterator<Map.Entry<K, V>> iterator() {
    ListIterator<K, V> iterator = new AscendingIterator<>(mStart, mEnd);
    mIterators.put(iterator, false);
    return iterator;
}
```

移除的话就不牢费心了，因为是弱引用，GC 会自动回收，关于这个迭代时可增删元素的集合就了解到这。看回 LifecycleRegistry：

```
private FastSafeIterableMap<LifecycleObserver, ObserverWithState> mObserverMap =
        new FastSafeIterableMap<>();
```

key 是 LifecycleObserver，Value 是 ObserverWithState，看看定义：

```
static State min(@NonNull State state1, @Nullable State state2) {
    return state2 != null && state2.compareTo(state1) < 0 ? state2 : state1;
}

static class ObserverWithState {
    State mState;
    LifecycleEventObserver mLifecycleObserver;

    ObserverWithState(LifecycleObserver observer, State initialState) {
        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;
    }
}
```

状态和观察者关联

转换成FullLifecycleObserverAdapter

事件分发

获得当前状态

比较添加观察者和当前状态

取更小的那个

回调对应方法

更新状态

就是状态与观察者进行关联，并提供统一的事件分发入口，接着看下啥时候加集合里了，搜下 putIfAbsent 定位到了 addObserver()。

```
@Override
public void addObserver(@NonNull LifecycleObserver observer) {
    enforceMainThreadIfNeeded( methodName: "addObserver");          判断当前生命周期状态，赋初值
    State initialState = mState == DESTROYED ? DESTROYED : INITIALIZED;
    ObserverWithState statefulObserver = new ObserverWithState(observer, initialState);
    ObserverWithState previous = mObserverMap.putIfAbsent(observer, statefulObserver);
                                                  初始化ObserverWithState实例，尝试添加到集合中
    if (previous != null) {
        return;      返回不为空，说明已添加过此Observer，直接返回
    }
    LifecycleOwner lifecycleOwner = mLifecycleOwner.get();
    if (lifecycleOwner == null) {  判断生命周期组件是否为null，空说明已经被销毁了，直接返回
        // it is null we should be destroyed. Fallback quickly
        return;
    }
            可重入标记 = 正在添加的Observer数量不为0 或 正在处理事件
    boolean isReentrance = mAddingObserverCounter != 0 || mHandlingEvent;
    State targetState = calculateTargetState(observer);     初始化同步到一个最小状态
    mAddingObserverCounter++; 添加Observer数量自增
    while ((statefulObserver.mState.compareTo(targetState) < 0
            && mObserverMap.contains(observer))) {   使Observer升级到目标状态，同时判断observer是否
        pushParentState(statefulObserver.mState);    不在集合中，避免因为用户在生命周期回调中移除
        final Event event = Event.upFrom(statefulObserver.mState);   了observer，立即结束迁移操作
        if (event == null) {
            throw new IllegalStateException("no event up from " + statefulObserver.mState);
        }
        statefulObserver.dispatchEvent(lifecycleOwner, event);   分发事件
        popParentState();
        // mState / subling may have been changed recalculate
        targetState = calculateTargetState(observer);   可能存在变更，获取当前目标需要迁移的目标状态
    }
                                Tips：就是粘性，添加得晚，但把之前的事件一一分发给你
    if (!isReentrance) {
        // we do sync only on the top level.
        sync();              非重入状态执行sync同步
    }
    mAddingObserverCounter--;   添加Observer数量自减
}
```

大体了解流程，有疑惑的应该是这个可重入标记和 sync() 同步，看看都同步的啥吧：

```
// happens only on the top of stack (never in reentrance),
// so it doesn't have to take in account parents
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()) {   判断是否完成同步。首尾节点状态相同说明已同步
        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);   降级同步
        }
        Map.Entry<LifecycleObserver, ObserverWithState> newest = mObserverMap.newest();
        if (!mNewEventOccurred && newest != null
                && mState.compareTo(newest.getValue().mState) > 0) {
            forwardPass(lifecycleOwner);   升级同步
        }
    }
    mNewEventOccurred = false;   新事件发生标记设置为false
}
```

看下降级、升级同步对应的两个方法：

```
private void forwardPass(LifecycleOwner lifecycleOwner) {
    Iterator<Map.Entry<LifecycleObserver, ObserverWithState>> ascendingIterator =
            mObserverMap.iteratorWithAdditions();                    正向迭代，从老到新
    while (ascendingIterator.hasNext() && !mNewEventOccurred) {
        Map.Entry<LifecycleObserver, ObserverWithState> entry = ascendingIterator.next();
        ObserverWithState observer = entry.getValue();     遇到新事件标识时中断
        while ((observer.mState.compareTo(mState) < 0 && !mNewEventOccurred
                && mObserverMap.contains(entry.getKey()))) {
            pushParentState(observer.mState);                循环对比单个观察者状态
            final Event event = Event.upFrom(observer.mState);  直到单个观察者同步到目标状态
            if (event == null) {
                throw new IllegalStateException("no event up from " + observer.mState);
            }
            observer.dispatchEvent(lifecycleOwner, event);  事件分发
            popParentState();
        }
    }
}

private void backwardPass(LifecycleOwner lifecycleOwner) {   逆向迭代，从新到老
    Iterator<Map.Entry<LifecycleObserver, ObserverWithState>> descendingIterator =
            mObserverMap.descendingIterator();
    while (descendingIterator.hasNext() && !mNewEventOccurred) {
        Map.Entry<LifecycleObserver, ObserverWithState> entry = descendingIterator.next();
        ObserverWithState observer = entry.getValue();
        while ((observer.mState.compareTo(mState) > 0 && !mNewEventOccurred
                && mObserverMap.contains(entry.getKey()))) {
            Event event = Event.downFrom(observer.mState);
            if (event == null) {
                throw new IllegalStateException("no event down from " + observer.mState);
            }
            pushParentState(event.getTargetState());
            observer.dispatchEvent(lifecycleOwner, event);
            popParentState();
        }
    }
}
```

所以 sync() 做的事情就是让所有观察者完成状态迁移，并完成相应的事件分发，而同步完成的判断依据就是首尾节点是否相等。

除了 addObserver() 添加新 Observer 时会同步外，在生命周期事件迁移时也会同步，定位到：


```
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;
}
```

看到这里应该能 feel 到为什么需要可重入的标记了，如果没有的话，可能产生 sync() 嵌套：

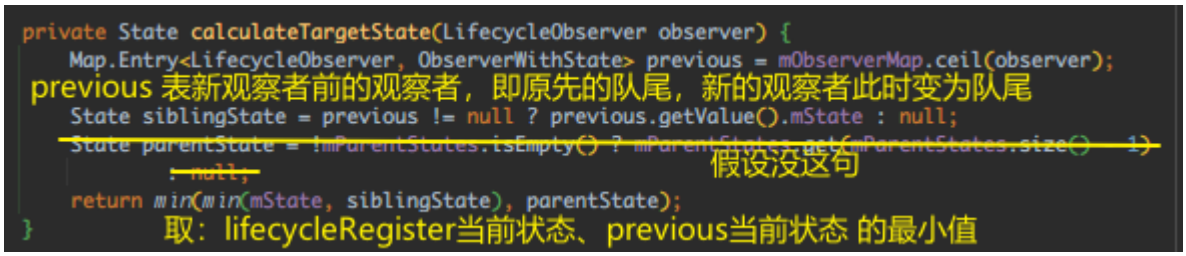
```
moveToState(state1)
→ sync()
→ moveToState(state1)
  → sync()

addObserver()
→ addObserver()
  → sync()
→ sync()

addObserver()
→ moveToState()
  → sync()
→ sync()
```

最后都会走最外层（顶层）的sync()，中间发生的 sync() 完全没必要执行，减少不必要的迭代或错误，通过这三个字段配合来完成：mHandlingEvent、mNewEventOccurred、mAddingObserverCounter。

然后还有个属性还没弄清楚是干嘛的：

```
// we have to keep it for cases:
// void onStart() {
//     mRegistry.removeObserver(this);
//     mRegistry.add(newObserver);
// }
// newObserver should be brought only to CREATED state during the execution of
// this onStart method. our invariant with mObserverMap doesn't help, because parent observer
// is no longer in the map.
private ArrayList<State> mParentStates = new ArrayList<>();
```

解释下：它为了解决事件嵌套中增加新观察者对观察者队列有序性的破坏。怎么说，看代码：



假如没有 mParentStates，好像还正常，然后注释给了一个反例：

- Observer1 在 onStart() 回调中把自己从集合中移除，然后添加了新的Observer2；
- 假如集合中只有 Observer1 这个观察者，移除后集合就是空的，会导致 Observer2 直接更新到 LifecycleRegister 的 STARTED 状态；
- 但此时 Observer1 的 onStart() 回调还未执行完，而 Observer2 的 ON_START 就回调执行完了，显然就违背了 LifecycleRegistry 的设计 → 观察者的同步是按照顺序执行的；

添加了这个属性，在执行观察者回调前 pushParentState() 暂存当前观察者，回调完后 popParentState() 移除观察者，然后执行 calculateTargetState() 时判断是否为空，不为空取出最后一个缓存的观察者，然后取 LifecycleRegister 当前状态、previous 当前状态、缓存观察者状态中的最小值，作为当前观察者的状态。

关于 LifecycleRegistry 关键代码的的解析就这些，它还对外暴露了几个改变状态的方法：

```
public void markState(@NonNull State state) {
    enforceMainThreadIfNeeded( methodName: "markState");
    setCurrentState(state);
}
```

Moves the Lifecycle to the given state and dispatches necessary events to the observers.

Params: state – new state

```
@MainThread
public void setCurrentState(@NonNull State state) {
    enforceMainThreadIfNeeded( methodName: "setCurrentState");
    moveToState(state);
}
```

Sets the current state and notifies the observers.

Note that if the currentState is the same state as the last call to this method, calling this method has no effect.

Params: event – The event that was received

```
public void handleLifecycleEvent(@NonNull Lifecycle.Event event) {
    enforceMainThreadIfNeeded( methodName: "handleLifecycleEvent");
    moveToState(event.getTargetState());
}
```

Activiyt 和 Fragment 中就有用到，等下会碰到，先继续往下走~

② ReportFragment类

一个专门用于分发生命周期事件的无UI界面的 Fragment，入口方法 injectIfNeededIn()。

```
public static void injectIfNeededIn(Activity activity) {
    if (Build.VERSION.SDK_INT >= 29) {
        // On API 29+, we can register for the correct Lifecycle callbacks directly
        LifecycleCallbacks.registerIn(activity);
    }
    // Prior to API 29 and to maintain compatibility with older versions of
    // ProcessLifecycleOwner (which may not be updated when lifecycle-runtime is updated and
    // need to support activities that don't extend from FragmentActivity from support lib),
    // use a framework fragment to get the correct timing of Lifecycle events
    android.app.FragmentManager manager = activity.getFragmentManager();
    if (manager.findFragmentByTag(REPORT_FRAGMENT_TAG) == null) {
        manager.beginTransaction().add(new ReportFragment(), REPORT_FRAGMENT_TAG).commit();
        // Hopefully, we are the first to make a transaction.
        manager.executePendingTransactions();
    }
}
```

很清晰明了，API版本大于等于29，直接使用 activity.registerActivityLifecycleCallbacks()，重写生命周期回调方法进行事件分发。

```
@RequiresApi(29)
static class LifecycleCallbacks implements Application.ActivityLifecycleCallbacks {

    static void registerIn(Activity activity) {
        activity.registerActivityLifecycleCallbacks(new LifecycleCallbacks());
    }

    @Override
    public void onActivityCreated(@NonNull Activity activity,
            @Nullable Bundle bundle) {
    }

    @Override
    public void onActivityPostCreated(@NonNull Activity activity,
            @Nullable Bundle savedInstanceState) {
        dispatch(activity, Lifecycle.Event.ON_CREATE);
    }
```

API 版本小于29，直接开启一个事务，新建一个 ReportFragment 实例，添加到 activity 上，在 ReportFragment 中已对生命周期回调方法进行了重写，完成事件分发：

```
@Override
public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    dispatchCreate(mProcessListener);
    dispatch(Lifecycle.Event.ON_CREATE);
}

@Override
public void onStart() {
    super.onStart();
    dispatchStart(mProcessListener);
    dispatch(Lifecycle.Event.ON_START);
}
```

然后 dispatch() 就是调下 LifecycleRegistry.handleLifecycleEvent() 而已。

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

还定义了一个 ActivityInitializationListener 接口：

```
interface ActivityInitializationListener {
    void onCreate();

    void onStart();

    void onResume();
}
```

支持外部通过 setProcessListener() 传入自定义实现，在 lifecycle-process 源码中看到过：

```
@Override
public void onActivityCreated(Activity activity, Bundle savedInstanceState) {
    // Only use ReportFragment pre API 29 - after that, we can use the
    // onActivityPostStarted and onActivityPostResumed callbacks registered in
    // onActivityPreCreated()
    if (Build.VERSION.SDK_INT < 29) {
        ReportFragment.get(activity).setProcessListener(mInitializationListener);
    }
}
```

```
ActivityInitializationListener mInitializationListener =
        new ActivityInitializationListener() {
            @Override
            public void onCreate() {
            }

            @Override
            public void onStart() {
                activityStarted();
            }

            @Override
            public void onResume() { activityResumed(); }
        };
```

留了个后门，让 ProcessLifecycleOwner 的 onStart() 和 onResume()，先于第一个 Activity 执行。

③ ViewTreeLifecycleOwner类

```java
public class ViewTreeLifecycleOwner {
    private ViewTreeLifecycleOwner() {
        // No instances
    }

    /** Set the {@link LifecycleOwner} responsible for managing the given {@link View}. ...*/
    public static void set(@NonNull View view, @Nullable LifecycleOwner lifecycleOwner) {
        view.setTag(R.id.view_tree_lifecycle_owner, lifecycleOwner);
    }

    Retrieve the LifecycleOwner responsible for managing the given View. This may be
    work or heavyweight resources associated with the view that may span cycles of the
    detached and reattached from a window.
    Params: view – View to fetch a LifecycleOwner for
    Returns: The LifecycleOwner responsible for managing this view and/or some subs
             ancestors
    @Nullable
    public static LifecycleOwner get(@NonNull View view) {
        LifecycleOwner found = (LifecycleOwner) view.getTag(R.id.view_tree_lifecycle_owner);
        if (found != null) return found;
        ViewParent parent = view.getParent();
        while (found == null && parent instanceof View) {
            final View parentView = (View) parent;
            found = (LifecycleOwner) parentView.getTag(R.id.view_tree_lifecycle_owner);
            parent = parentView.getParent();
        }
        return found;
    }
}
```

看着有点蒙？看下 set() 调用处的代码就知道了：

ComponentActivity 实现了 LifecycleOwner 接口，所以这里传入了根视图 + ComponentActivity 这个 LifecycleOwner。

而在调用 get() 传入 view 时，通过 getParent() 一层层往上拿，直到获取到这个 LifecycleOwner 为止(也可能没有返回空)。

那这有什么用呢？简化代码！

View 内部需要基于 lifecycle 进行某些操作时，可以避免 Lifecycle 的层层传递，比如 LiveData 订阅。

/ Activity中的lifecycle相关 /

实现了 Lifecycle 接口，没干啥活，毕竟生命周期事件分发的活都交给 ReportFragment 了，直接贴相关代码~

```
// 定义一个LifecycleRegistry
private final LifecycleRegistry mLifecycleRegistry = new LifecycleRegistry(this);

@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
   ...
   ReportFragment.injectIfNeededIn(this); // 使用Report分发生命周期事件
   ...
}

@Override
protected void onSaveInstanceState(@NonNull Bundle outState) {
   Lifecycle lifecycle = getLifecycle();
   if (lifecycle instanceof LifecycleRegistry) {
      // 设置mLifecycleRegistry当前状态为CREATED
      ((LifecycleRegistry) lifecycle).setCurrentState(Lifecycle.State.CREATED);
   }
   ...
}

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

/ fragment中的lifecycle相关 /

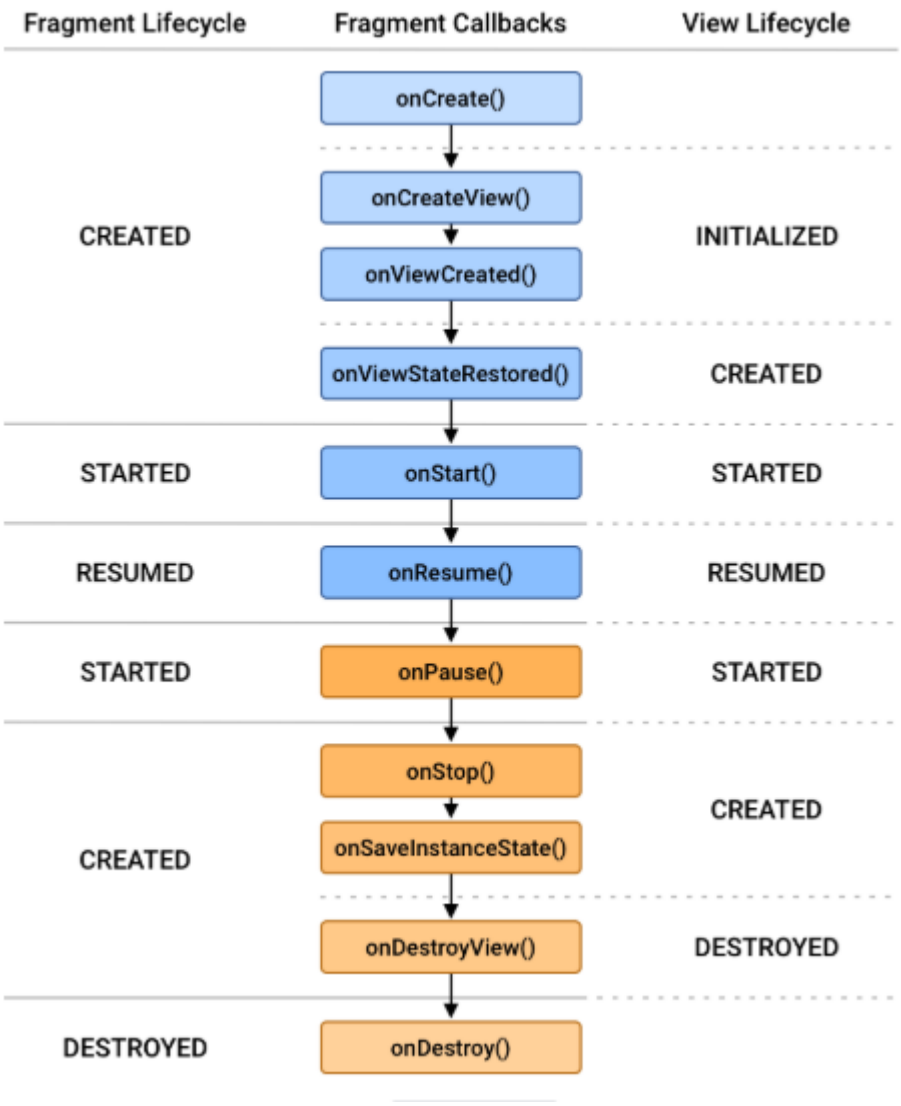
同样实现了 LifecycleOwner 接口，实例化了一个 LifecycleRegistry 用于生命周期事件转发。

有一点要注意，在大多数情况下 Fragment 与其管理的 View（视图）的生命周期是一致的，但存在特例：

Fragment 被 replace() 时 → FragmentTransaction.detach() → 回调onDestroyView() 销毁视图 → 不走onDestory() → 使得 Fragment 状态得以保留，当前内存空间得以释放，

下次加载直接 onCreateView()，速度更快。

这样的操作也导致了 Fragment 的生命周期比 View 长，所以 Fragment 还定义了一个 FragmentViewLifecycleOwner 来单独处理 View 的生命周期，官方文档有给出《Fragment lifecycle》给出了这样一张图：



源码注释中也有给出 Fragment 中管理的 View，生命周期与 Fragment 自身回调间的对应关系。

```
* Namely, the lifecycle of the Fragment's View is:
* <ol>
* <li>{@link Lifecycle.Event#ON_CREATE created} after {@link #onViewStateRestored(Bundle)}</li>
* <li>{@link Lifecycle.Event#ON_START started} after {@link #onStart()}</li>
* <li>{@link Lifecycle.Event#ON_RESUME resumed} after {@link #onResume()}</li>
* <li>{@link Lifecycle.Event#ON_PAUSE paused} before {@link #onPause()}</li>
* <li>{@link Lifecycle.Event#ON_STOP stopped} before {@link #onStop()}</li>
* <li>{@link Lifecycle.Event#ON_DESTROY destroyed} before {@link #onDestroyView()}</li>
* </ol>
```

直接抠出来，方便看：

- onViewStateRestored() 后 → ON_CREATE
- onStart() 后 → ON_START
- onResume() 后 → ON_RESUME
- onPause() 前 → ON_PAUSE
- onStop() 前 → ON_STOP
- onDestroyView() 前 → ON_DESTROY

可调用 getViewLifecycleOwner() 获得 View 的 LifecycleOwner 哈~

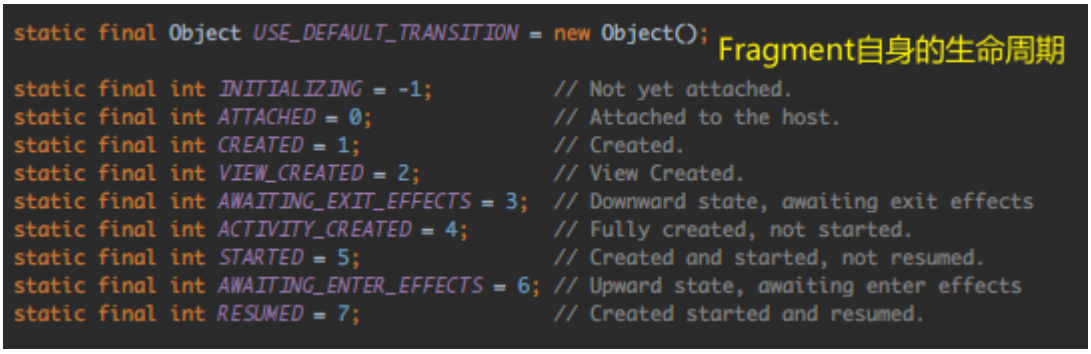
```
@MainThread
@NonNull
public LifecycleOwner getViewLifecycleOwner() {
    if (mViewLifecycleOwner == null) {
        throw new IllegalStateException("Can't access the Fragment View's LifecycleOwner when "
                + "getView() is null i.e., before onCreateView() or after onDestroyView()");
    }
    return mViewLifecycleOwner;
}
```

Fragment 中涉及到状态流转的核心代码如下：

```
void performCreate(Bundle savedInstanceState) {
  if (Build.VERSION.SDK_INT >= 19) {
    mLifecycleRegistry.addObserver(new LifecycleEventObserver() {
      @Override
      public void onStateChanged(@NonNull LifecycleOwner source,
          @NonNull Lifecycle.Event event) {
        if (event == Lifecycle.Event.ON_STOP) {
          if (mView != null) {
            mView.cancelPendingInputEvents();
          }
        }
      }
    });
  }
```

```
    mLifecycleRegistry.handleLifecycleEvent(Lifecycle.Event.ON_CREATE);
}

void performStart() {
    mState = STARTED;
    mLifecycleRegistry.handleLifecycleEvent(Lifecycle.Event.ON_START);
    if (mView != null) {
        mViewLifecycleOwner.handleLifecycleEvent(Lifecycle.Event.ON_START);
    }
}

void performResume() {
    mState = RESUMED;
    mLifecycleRegistry.handleLifecycleEvent(Lifecycle.Event.ON_RESUME);
    if (mView != null) {
        mViewLifecycleOwner.handleLifecycleEvent(Lifecycle.Event.ON_RESUME);
    }
}

void performPause() {
    if (mView != null) {
        mViewLifecycleOwner.handleLifecycleEvent(Lifecycle.Event.ON_PAUSE);
    }
    mLifecycleRegistry.handleLifecycleEvent(Lifecycle.Event.ON_PAUSE);
    mState = AWAITING_ENTER_EFFECTS;
}

void performStop() {
    if (mView != null) {
        mViewLifecycleOwner.handleLifecycleEvent(Lifecycle.Event.ON_STOP);
    }
    mLifecycleRegistry.handleLifecycleEvent(Lifecycle.Event.ON_STOP);
    mState = ACTIVITY_CREATED;
}

void performDestroy() {
    mLifecycleRegistry.handleLifecycleEvent(Lifecycle.Event.ON_DESTROY);
    mState = ATTACHED;
}
```

既有 Fragment Lifecycle 的 State，又有 ViewLifecycleOwner 的 State，还有 Fragment 自身的 State，这个要区分哈：



另外，相比以前的源码，Fragment 发生了较大的改动，比如状态管理相关的剥离到 FragmentStateManager 中了，笔者目前还不太了解，就不先不往下挖了，后续专门研究 Fragment 再说吧~

/ 写在最后 /

关于源码的解析暂且到这里吧，Lifecycle 的路数基本摸清了，小结下要点方便回顾：

- ① Lifecycle 的核心思想是：模板模式 + 观察者模式；
- ② Lifecycle 抽象类：抽象被观察者，定义了两个生命周期相关的枚举 Event 和 State，统一了 State 升降级对应触发的 Event(关联关系)，提供了添加、移除观察者，获取当前 State 的三个抽象方法；
- ③ LifecycleObserver：空接口，类型标记，抽象观察者；
- ④ FullLifecycleObserver、LifecycleEventObserver：继承 LifecycleObserver 接口，提供两种不同的回调方式；
- ⑤ DefaultLifecycleObserver：继承 FullLifecycleObserver，利用 Java 8特性接口声明默认方法，默认重写了回调方法，具体观察者；
- ⑥ LifecycleOwner：提供一个获取 Lifecycle 的方法；
- ⑦ Lifecycling → 对传入 LifecycleOwner 进行统一的类型包装，使得 Event 分发过程得以统一入口；
- ⑧ LifecycleRegistry → 具体观察者，组件状态维护，使用自定义支持迭代时增删元素的 FastSafeIterableMap (HashMap套链表) 保存观察者。键为LifecycleObserver，值为 ObserverWithState，其中包含观察者与状态关联，并提供事件分发方法 dispatchEvent()；
- ⑨ 通过三个变量：mHandlingEvent、mNewEventOccurred、mAddingObserverCounter 的配合来解决 事件嵌套引起的 sync() 多次执行；
- ⑩ 额外定义了一个 mParentStates 来解决事件嵌套中增加新观察者对观察者队列有序性的破坏。

推荐阅读：

我的新书，《第一行代码 第3版》已出版！

Kotlin Flow响应式编程，基础知识入门

Jetpack WorkManager 看这一篇就够了

欢迎关注我的公众号

学习技术或投稿

▼

长按上图，识别图中二维码即可关注

阅读原文

喜欢此内容的人还喜欢

你还在用for循环遍历list吗？

猿大侠

万物皆可金拱门？麦当劳的设计卷出新高度！

卓尔谟工业设计小站

详情页排版设计没想法，这个很值得参考！

三个小美工