Androidx 下 Fragment 懒加载的新实现

原创 AndyJennifer 掘金开发者社区 2020-01-19

前言

> 年后最后一篇文章啦，在这里先祝大家新年快乐~最重要的抽中 全家福 ，明年继续修福报🐶

以前处理 Fragment 的懒加载，我们通常会在 Fragment 中处理 `setUserVisibleHint + onHiddenChanged` 这两个函数，而在 Androidx 模式下，我们可以使用 `FragmentTransaction.setMaxLifecycle()` 的方式来处理 Fragment 的懒加载。

在本文章中，我会详细介绍不同使用场景下两种方案的差异。大家快拿好小板凳。一起来学习新知识吧！

> 本篇文章涉及到的 Demo，已上传至Github---->传送门

老的懒加载处理方案

> 如果你熟悉老一套的 Fragment 懒加载机制，你可以直接查看 Androix 懒加载相关章节

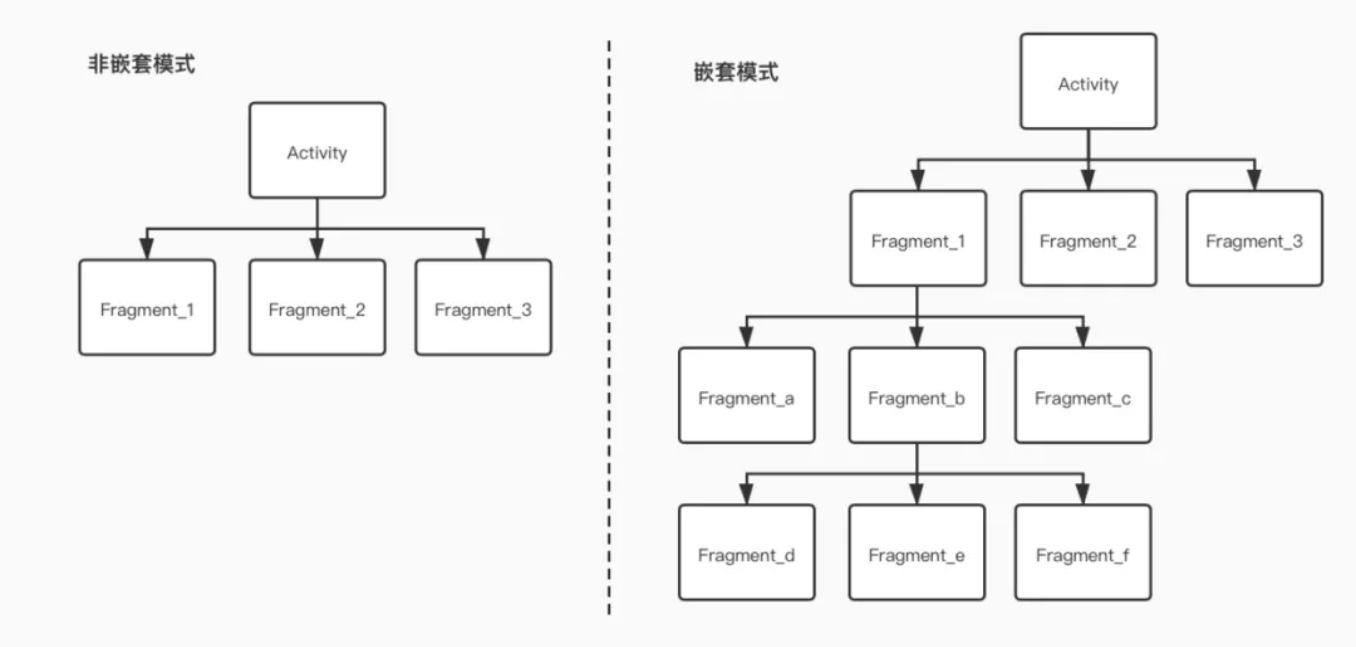
add+show+hide 模式下的老方案

在没有添加懒加载之前，只要使用 `add+show+hide` 的方式控制并显示 Fragment, 那么不管 Fragment 是否嵌套，在初始化后，如果 只调用了`add+show` ，同级下的 Fragment 的相关生命周期函数都会被调用。且调用的生命周期函数如下所示：

`onAttach -> onCreate -> onCreatedView -> onActivityCreated -> onStart -> onResume`

> Fragment 完整生命周期：onAttach -> onCreate -> onCreatedView -> onActivityCreated -> onStart -> onResume -> onPause -> onStop -> onDestroyView -> onDestroy -> onDetach

什么是同级 Frament 呢？看下图



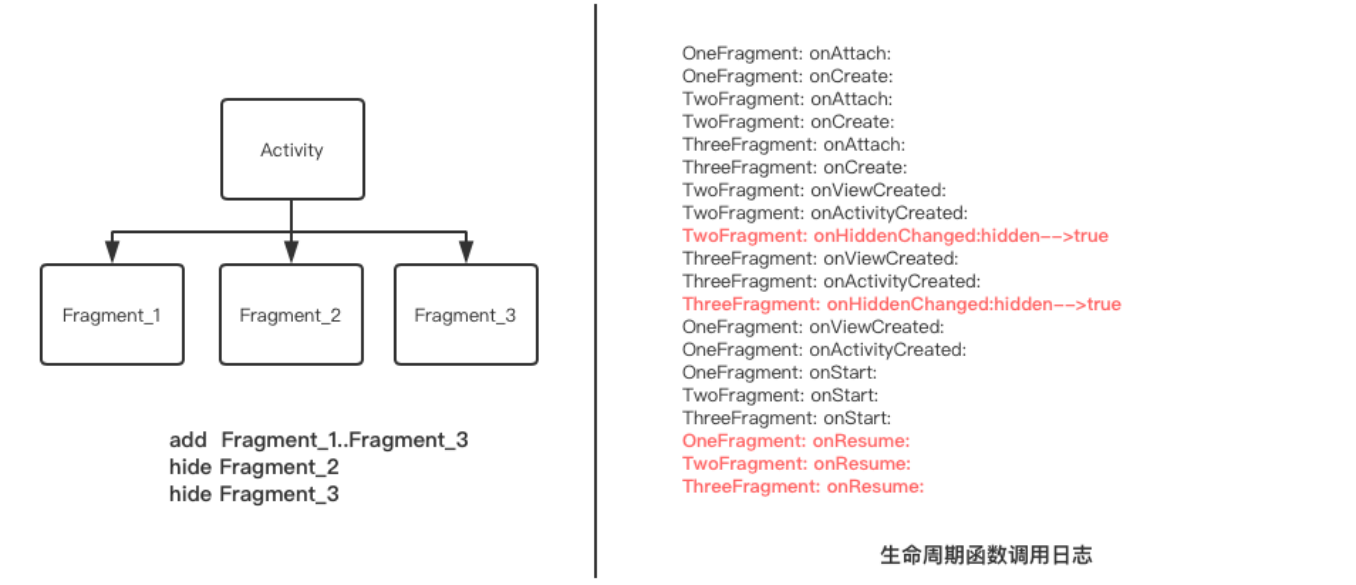
同级Fragment.jpg

> 上图中，都是使用 `add+show+hide` 的方式控制 Fragment,

在上图两种模式中:

- Fragment_1、Fragment_2、Fragment_3 属于同级 Fragment
- Fragment_a、Fragment_b、Fragment_c 属于同级 Fragment
- Fragment_d、Fragment_e、Fragment_f 属于同级 Fragment

那这种方式会带来什么问题呢？结合下图我们来分别分析。



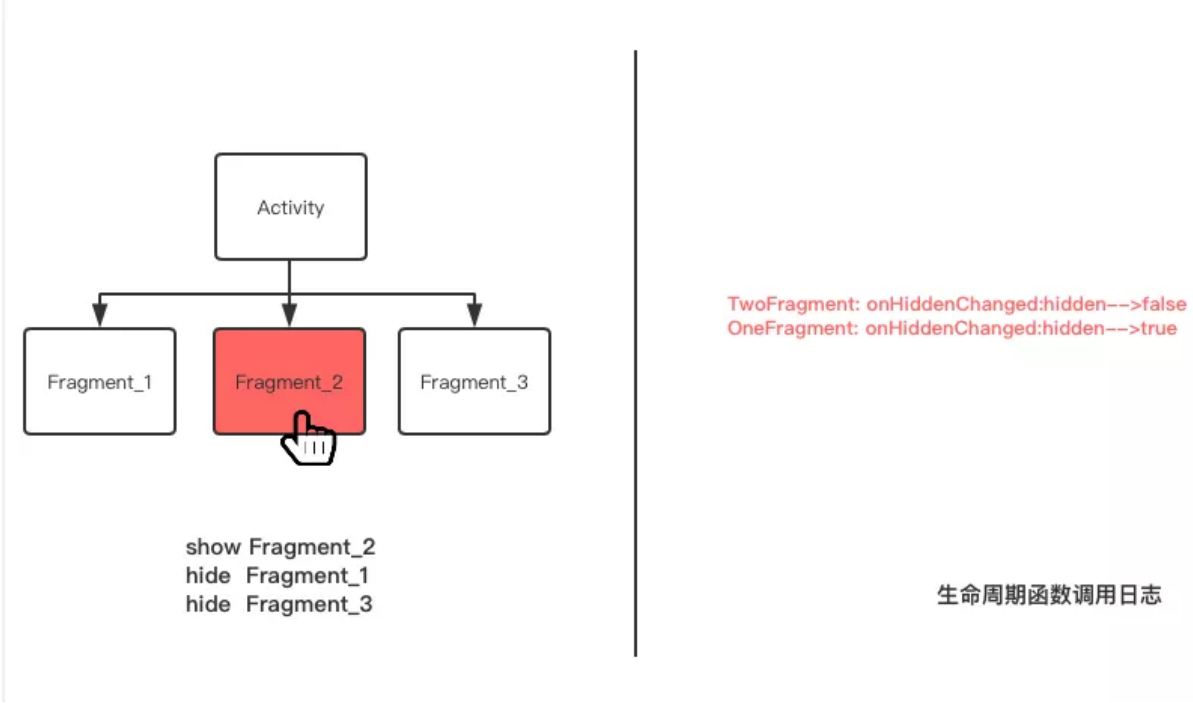
show1.png

观察上图我们可以发现，同级的Fragment_1、Fragment_2、Fragment_3都调用了 `onAttach...onResume` 系列方法，也就是说，如果我们没有对 Fragment 进行懒加载处理，那么我们就会无缘无故的加载一些并 不可见 的 Fragment，也就会造成用户流量的无故消耗（我们会在 Fragment 相关生命周期函数中，请求网络或其他数据操作）。

> 这里 "不可见的Fragment" 是指，实际不可见但是相关可见生命周期函数(如 `onResume` 方法）被调用的 Fragment

如果使用嵌套 Fragment，这种浪费流量的行为就更明显了。以本节的图一为例，当 Fragment_1 加载时，如果你在 Fragment_1 生命周期函数中使用 `show+add+hide` 的方式添加 `Fragment_a`、`Fragment_b`、`Fragment_c` ，那么 Fragment_b 又会在其生命周期函数中继续加载 `Fragment_d`、`Fragment_e`、`Fragment_f` 。

那如何解决这种问题呢？我们继续接着上面的例子走，当我们 `show Fragment_2`，并 hide其他 Fragment 时，对应 Fragment 的生命周期调用如下：



show2.png

从上图中，我们可以看出 Fragment_2 与 Fragment_3 都调用了 `onHiddenChanged` 函数，该函数的官方 API 声明如下：

```
/**
 * Called when the hidden state (as returned by {@link #isHidden()} of
 * the fragment has changed.  Fragments start out not hidden; this will
 * be called whenever the fragment changes state from that.
 * @param hidden True if the fragment is now hidden, false otherwise.
 */
public void onHiddenChanged(boolean hidden) {
}
```

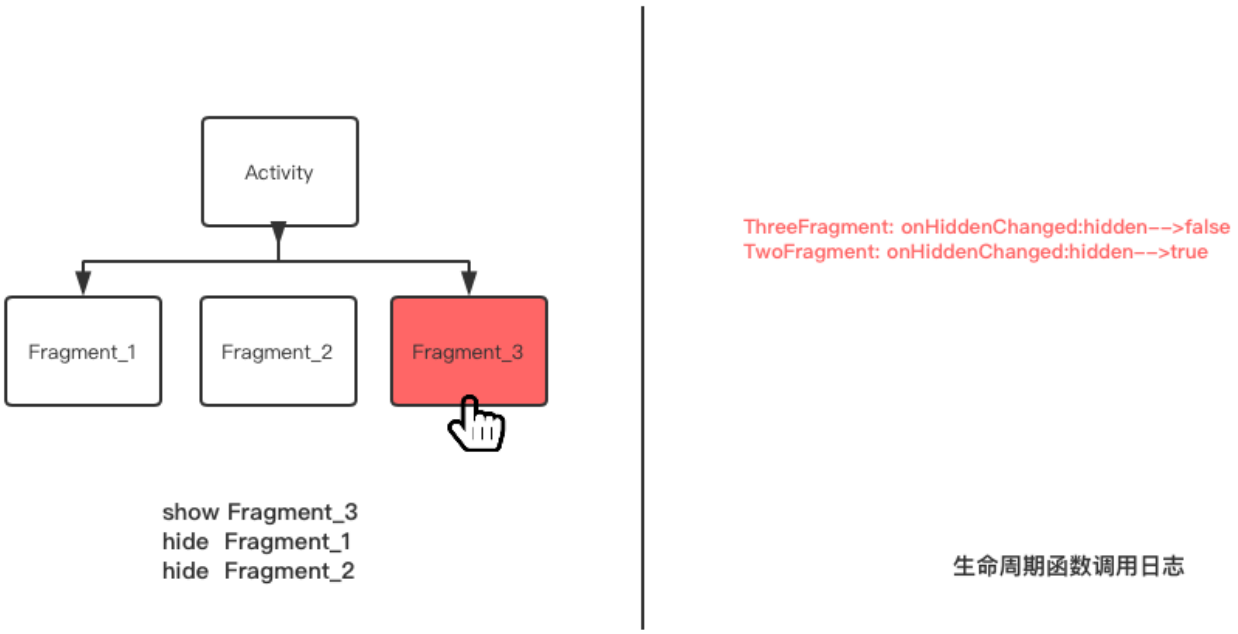
根据官方 API 的注释，我们大概能知道，当 Fragment 隐藏的状态发生改变时，该函数将会被调用，如果当前 Fragment 隐藏， `hidden` 的值为 true, 反之为 false。最为重要的是 `hidden` 的值，可以通过调用 `isHidden()` 函数获取。

那么结合上述知识点，我们能推导出：

- 因为 Fragment_1 的 隐藏状态 从 可见转为了不可见 ，所以其 `onHiddenChanged` 函数被调用，同时 `hidden` 的值为 true。
- 同理对于 Fragment_2 ，因为其 隐藏状态 从 不可见转为了可见 ，所以其 hidden 值为 false。
- 对于 Fragment_3 ,因为其隐藏状态从始至终都没有发生变化，所以其 onHiddenChanged 函数并不会调用。

嗯，好像有点眉目了。不急，我们继续看下面的例子。

show Fragment_3 并 hide 其他 Fragment ， 对应生命周期函数调用如下所示：



show3.png

从图中，我们可以看出，确实只有 隐藏状态 发生了改变的 Fragment 其 `onHiddenChanged` 函数才会调用，那么结合以上知识点，我们能得出如下重要结论：

只要通过 `show+hide` 方式控制 Fragment 的显隐，那么在第一次初始化后，Fragment 任何的生命周期方法都不会调用，只有 `onHiddenChanged` 方法会被调用。

那么，假如我们要在 `add+show+hide` 模式下控制 Fragment 的懒加载，我们只需要做这两步：

- 我们需要在 `onResume()` 函数中调用 `isHidden()` 函数，来处理默认显示的 Fragment
- 在 `onHiddenChanged` 函数中控制其他不可见的Fragment，

也就是这样处理：

```
abstract class LazyFragment:Fragment(){

    private var isLoaded = false //控制是否执行懒加载

    override fun onResume() {
        super.onResume()
        judgeLazyInit()

    }
    override fun onHiddenChanged(hidden: Boolean) {
        super.onHiddenChanged(hidden)
        isVisibleToUser = !hidden
        judgeLazyInit()
    }

    private fun judgeLazyInit() {
        if (!isLoaded && !isHidden) {
            lazyInit()
            isLoaded = true
        }
    }

     override fun onDestroyView() {
        super.onDestroyView()
        isLoaded = false
    }

    //懒加载方法
    abstract fun lazyInit()
}
```

> 该懒加载的实现，是在 `onResume` 方法中操作，当然你可以在其他生命周期函数中控制。但是建议在该方法中执行懒加载。

ViewPager+Fragment 模式下的老方案

使用传统方式处理 ViewPager 中 Fragment 的懒加载，我们需要控制 `setUserVisibleHint(boolean isVisibleToUser)` 函数，该函数的声明如下所示：

```
    public void setUserVisibleHint(boolean isVisibleToUser) {}
```

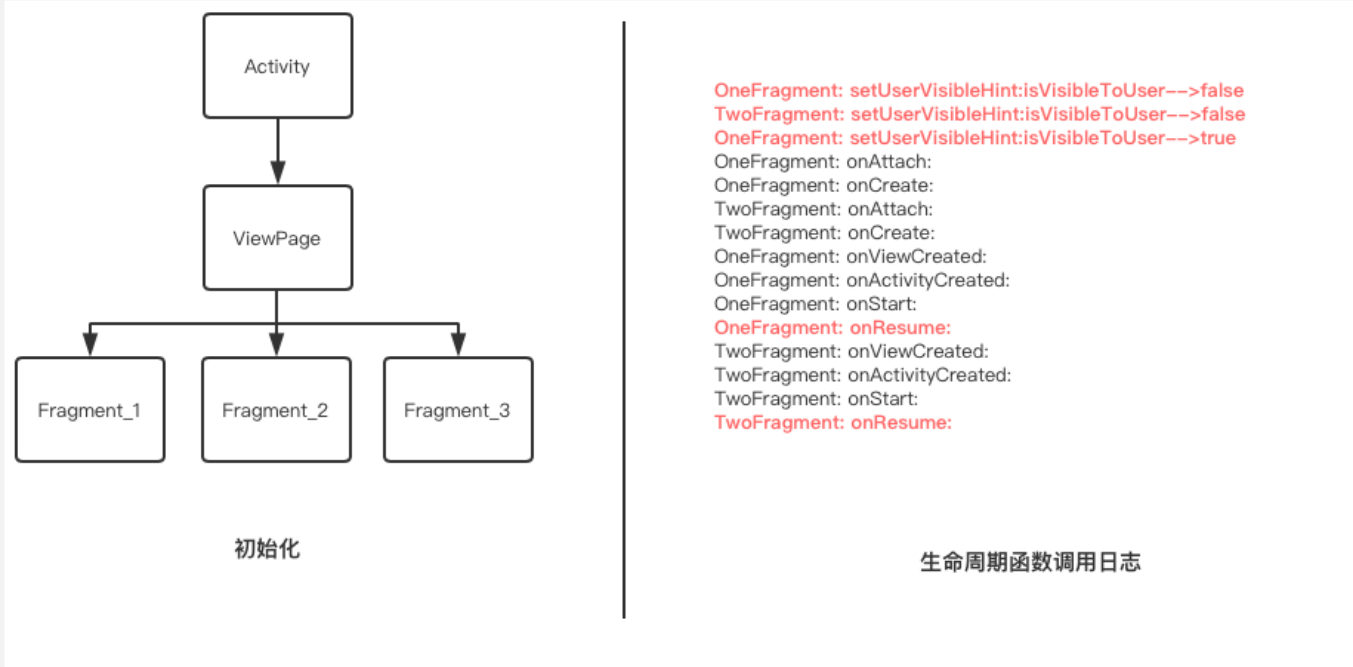
该函数与之前我们介绍的 `onHiddenChanged()` 作用非常相似，都是通过传入的参数值来判断当前 Fragment 是否对用户可见，只是 `onHiddenChanged()` 是在 `add+show+hide` 模式下使

用，而 `setUserVisibleHint` 是在 ViewPager+Fragment 模式下使用。

在本节中，我们用 `FragmentPagerAdapter + ViewPager` 为例，向大家讲解如何实现 Fragment 的懒加载。

> 注意：在本例中没有调用 `setOffscreenPageLimit` 方法去设置 ViewPager 预缓存的 Fragment 个数。默认情况下 ViewPager 预缓存 Fragment 的个数为 `1` 。

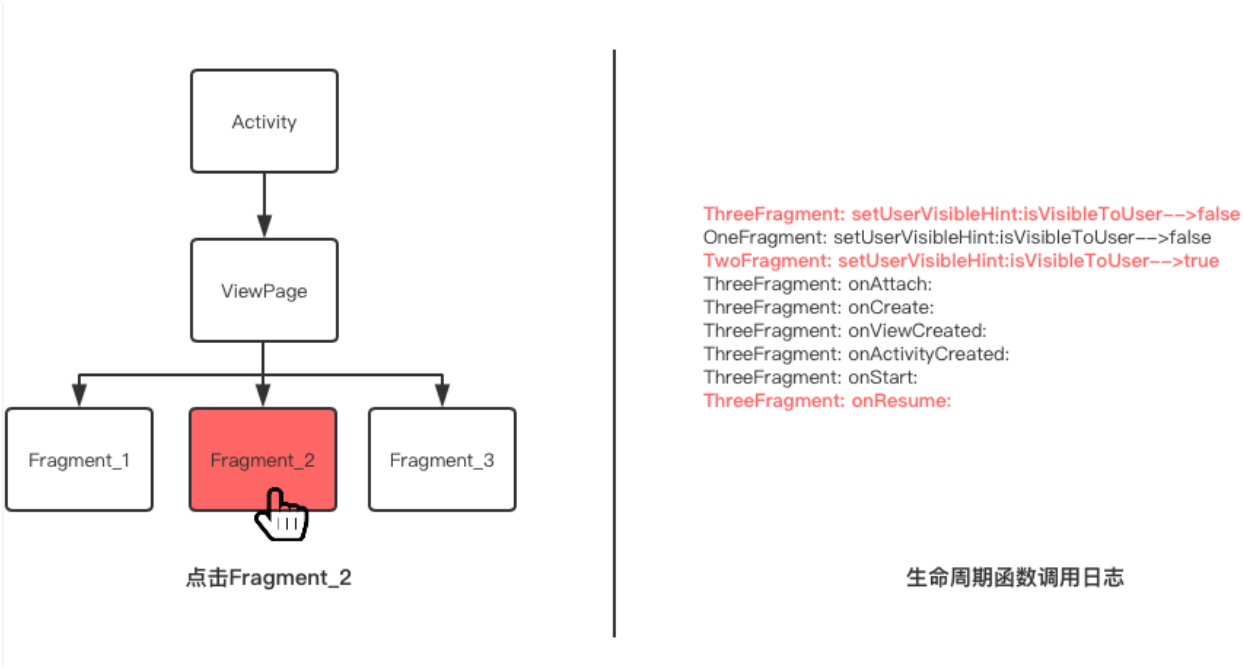
初始化 ViewPager 查看内部 Fragment 生命周期函数调用情况：



viewpager1.png

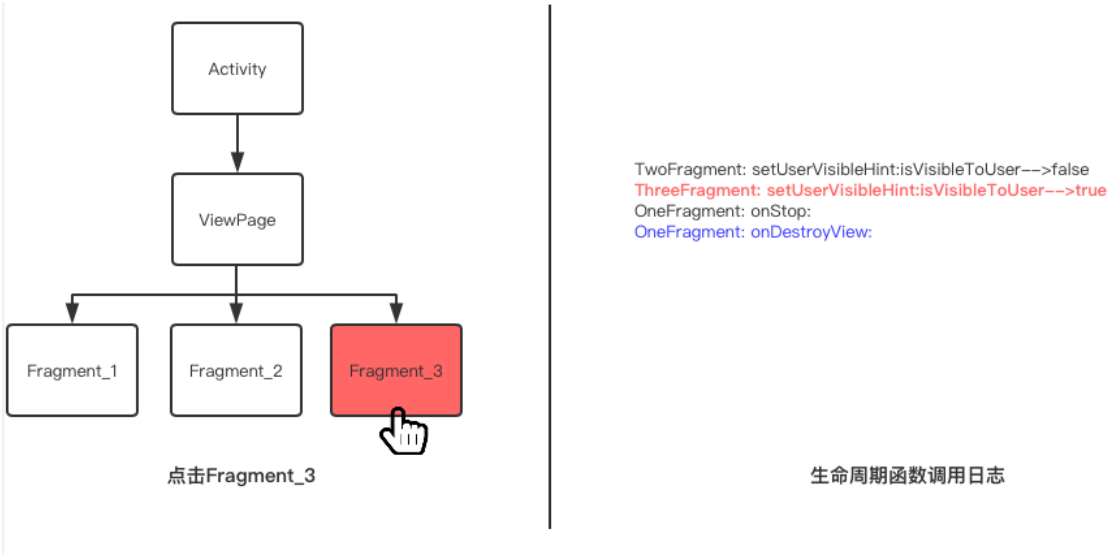
观察上图，我们能发现 ViePager 初始化时，默认会调用其内部 Fragment 的 setUserVisibleHint 方法，因为其预缓存 Fragment 个数为 `1` 的原因，所以只有 Fragment_1 与 Fragment_2 的生命周期函数被调用。

我们继续切换到 Fragment_2，查看各个Fragment的生命周期函数的调用变化。如下图所示：



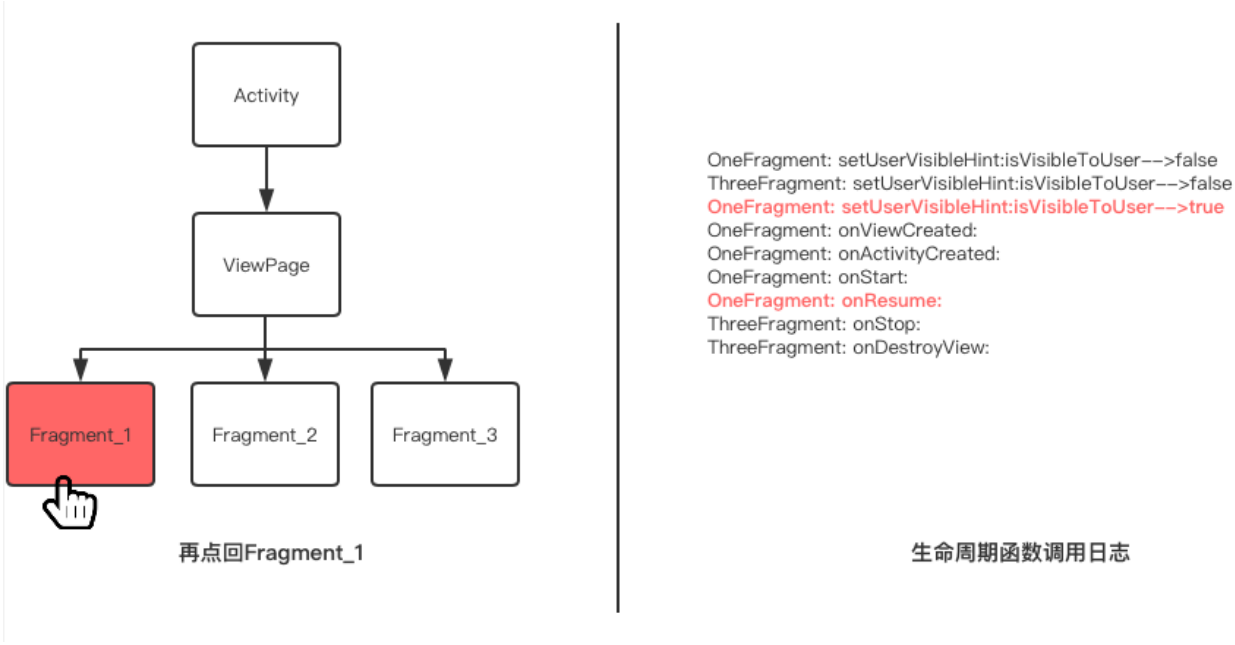
viewpage2.png

观察上图，我们同样发现 Fragment 的 setUserVisibleHint 方法被调用了，并且 Fragment_3 的一系列生命周期函数被调用了。继续切换到 Fragment_3:



viewpager_3.png

观察上图可以发现，Fragment_3 调用了 setUserVisibleHint 方法，继续又切换到 Fragment_1，查看调用函数的变化：



viewpager4.png

> 因为之前在切换到 Fragment_3 时，Frafgment_1 已经走了 onDestoryView(图二，蓝色标记处) 方法，所以 Fragment_1 需要重新走一次生命周期。

那么结合本节的三幅图，我们能得出以下结论：

- 使用 ViewPager，切换回上一个 Fragment 页面时（已经初始化完毕），不会回调任何生命周期方法以及onHiddenChanged()，只有 setUserVisibleHint(boolean isVisibleToUser) 会被回调。
- setUserVisibleHint(boolean isVisibleToUser) 方法总是会优先于 Fragment 生命周期函数的调用。

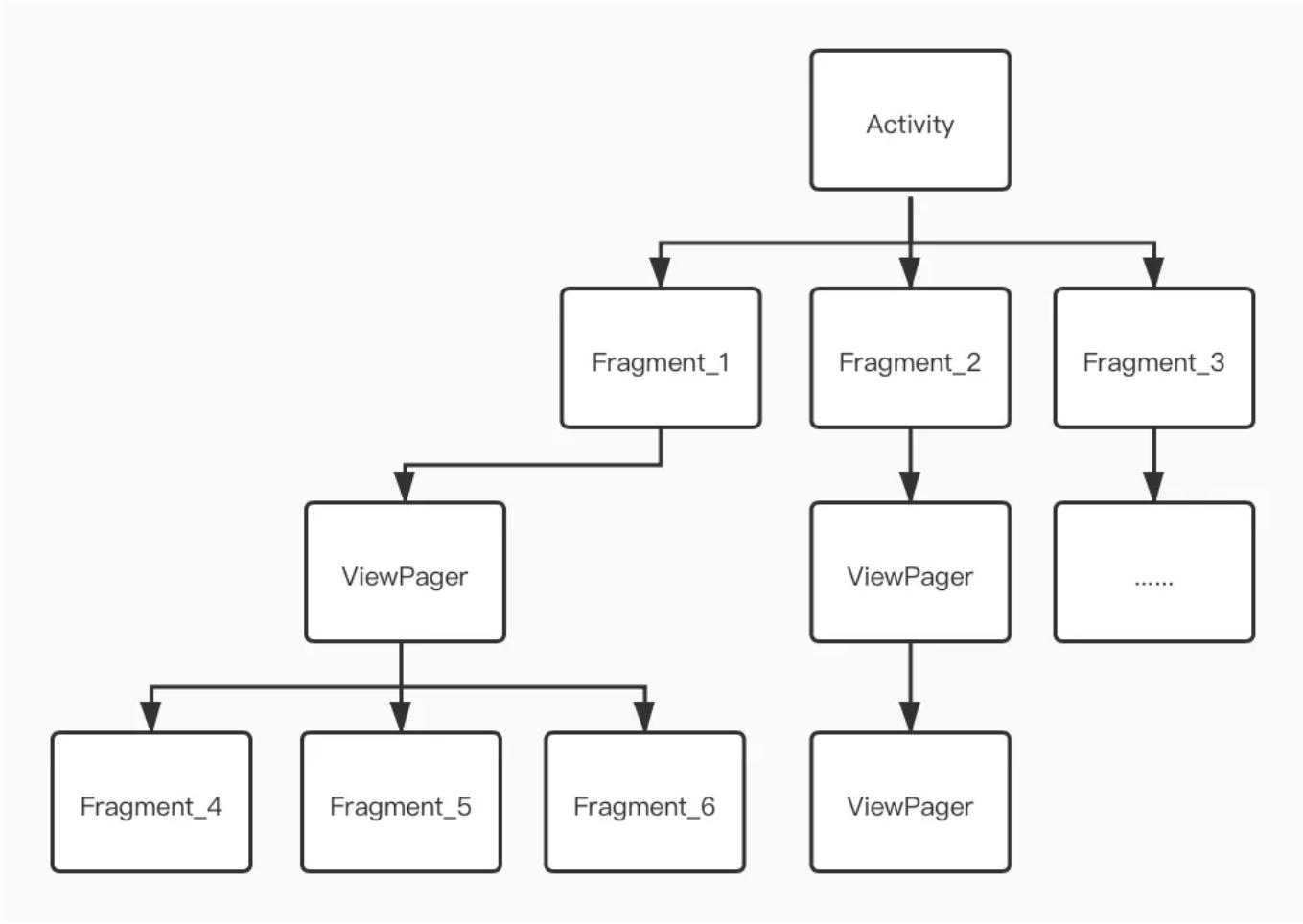
所以如果我们想对 ViewPager 中的 Fragment 懒加载，我们需要这样处理：

```
abstract class LazyFragment : Fragment() {

    /**
     * 是否执行懒加载
     */
    private var isLoaded = false

    /**
     * 当前Fragment是否对用户可见
     */
    private var isVisibleToUser = false

    /**
     * 当使用ViewPager+Fragment形式会调用该方法时，setUserVisibleHint会优先Fragment生命周期
     * 所以这个时候就,会导致在setUserVisibleHint方法执行时就执行了懒加载，
     * 而不是在onResume方法实际调用的时候执行懒加载。所以需要这个变量
     */
    private var isCallResume = false
```

```
    override fun onResume() {
        super.onResume()
        isCallResume = true
        judgeLazyInit()
    }


    private fun judgeLazyInit() {
        if (!isLoaded && isVisibleToUser && isCallResume) {
            lazyInit()
            Log.d(TAG, "lazyInit:!!!!!!!")
            isLoaded = true
        }
    }

    override fun onHiddenChanged(hidden: Boolean) {
        super.onHiddenChanged(hidden)
        isVisibleToUser = !hidden
        judgeLazyInit()
    }

    //在Fragment销毁View的时候，重置状态
    override fun onDestroyView() {
        super.onDestroyView()
        isLoaded = false
        isVisibleToUser = false
        isCallResume = false
    }

    override fun setUserVisibleHint(isVisibleToUser: Boolean) {
        super.setUserVisibleHint(isVisibleToUser)
        this.isVisibleToUser = isVisibleToUser
        judgeLazyInit()
    }

    abstract fun lazyInit()
}
```

复杂 Fragment 嵌套的情况

当然，在实际项目中，我们可能会遇到更为复杂的 Fragment 嵌套组合。比如 Fragment+Fragment、Fragment+ViewPager、ViewPager+ViewPager….等等。
如下图所示：



复杂嵌套Fragment.jpg

对于以上场景，我们就需要重写我们的懒加载，以支持不同嵌套组合模式下 Fragment 正确懒加载。我们需要将 LazyFragment 修改成如下这样：

```
abstract class LazyFragment : Fragment() {

    /**
     * 是否执行懒加载
     */
    private var isLoaded = false

    /**
     * 当前Fragment是否对用户可见
     */
    private var isVisibleToUser = false

    /**
     * 当使用ViewPager+Fragment形式会调用该方法时，setUserVisibleHint会优先Fragment生命周期
     * 所以这个时候就,会导致在setUserVisibleHint方法执行时就执行了懒加载，
     * 而不是在onResume方法实际调用的时候执行懒加载。所以需要这个变量
     */
    private var isCallResume = false

    /**
     * 是否调用了setUserVisibleHint方法。处理show+add+hide模式下，默认可见 Fragment 不调用
     * onHiddenChanged 方法，进而不执行懒加载方法的问题。
     */
    private var isCallUserVisibleHint = false
```

```
    override fun onResume() {
        super.onResume()
        isCallResume = true
        if (!isCallUserVisibleHint) isVisibleToUser = !isHidden
        judgeLazyInit()
    }


    private fun judgeLazyInit() {
        if (!isLoaded && isVisibleToUser && isCallResume) {
            lazyInit()
            Log.d(TAG, "lazyInit:!!!!!!!")
            isLoaded = true
        }
    }

    override fun onHiddenChanged(hidden: Boolean) {
        super.onHiddenChanged(hidden)
        isVisibleToUser = !hidden
        judgeLazyInit()
    }

    override fun onDestroyView() {
        super.onDestroyView()
        isLoaded = false
        isVisibleToUser = false
        isCallUserVisibleHint = false
        isCallResume = false
    }

    override fun setUserVisibleHint(isVisibleToUser: Boolean) {
        super.setUserVisibleHint(isVisibleToUser)
        this.isVisibleToUser = isVisibleToUser
        isCallUserVisibleHint = true
        judgeLazyInit()
    }

    abstract fun lazyInit()
}
```

Androidx 下的懒加载

虽然之前的方案就能解决轻松的解决 Fragment 的懒加载，但这套方案有一个最大的弊端， 就是不可见的 `Fragment` 执行了 `onResume()` 方法 。onResume 方法设计的初衷，难道不是当前 Fragment 可以和用户进行交互吗？你他妈既不可见，又不能和用户进行交互，你执行 onResume 方法干嘛?

基于此问题，Google 在 Androidx 在 `FragmentTransaction` 中增加了 `setMaxLifecycle` 方法来控制 Fragment 所能调用的最大的生命周期函数。如下所示:

```
/**
 * Set a ceiling for the state of an active fragment in this FragmentManager. If fra
 * already above the received state, it will be forced down to the correct state.
 *
 * <p>The fragment provided must currently be added to the FragmentManager to have i
 * Lifecycle state capped, or previously added as part of this transaction. The
 * {@link Lifecycle.State} passed in must at least be {@link Lifecycle.State#CREATED
 * an {@link IllegalArgumentException} will be thrown.</p>
 *
 * @param fragment the fragment to have it's state capped.
 * @param state the ceiling state for the fragment.
 * @return the same FragmentTransaction instance
 */
@NonNull
public FragmentTransaction setMaxLifecycle(@NonNull Fragment fragment,
        @NonNull Lifecycle.State state) {
    addOp(new Op(OP_SET_MAX_LIFECYCLE, fragment, state));
    return this;
}
```

根据官方的注释，我们能知道，该方法可以设置活跃状态下 Fragment 最大的状态，如果该 Fragment 超过了设置的最大状态，那么会强制将 Fragment 降级到正确的状态。

那如何使用该方法呢？我们先看该方法在 Androidx 模式下 ViewPager+Fragment 模式下的使用例子。

ViewPager+Fragment 模式下的方案

在 FragmentPagerAdapter 与 FragmentStatePagerAdapter 新增了含有 `behavior` 字段的构造函数，如下所示：

```
public FragmentPagerAdapter(@NonNull FragmentManager fm,
        @Behavior int behavior) {
    mFragmentManager = fm;
    mBehavior = behavior;
}

public FragmentStatePagerAdapter(@NonNull FragmentManager fm,
        @Behavior int behavior) {
    mFragmentManager = fm;
    mBehavior = behavior;
}
```

其中 Behavior 的声明如下：

```
@Retention(RetentionPolicy.SOURCE)
@IntDef({BEHAVIOR_SET_USER_VISIBLE_HINT, BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT})
private @interface Behavior { }
```

```
    /**
     * Indicates that {@link Fragment#setUserVisibleHint(boolean)} will be called when t
     * fragment changes.
     *
     * @deprecated This behavior relies on the deprecated
     * {@link Fragment#setUserVisibleHint(boolean)} API. Use
     * {@link #BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT} to switch to its replacement,
     * {@link FragmentTransaction#setMaxLifecycle}.
     * @see #FragmentPagerAdapter(FragmentManager, int)
     */
    @Deprecated
    public static final int BEHAVIOR_SET_USER_VISIBLE_HINT = 0;

    /**
     * Indicates that only the current fragment will be in the {@link Lifecycle.State#RE
     * state. All other Fragments are capped at {@link Lifecycle.State#STARTED}.
     *
     * @see #FragmentPagerAdapter(FragmentManager, int)
     */
    public static final int BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT = 1;
```

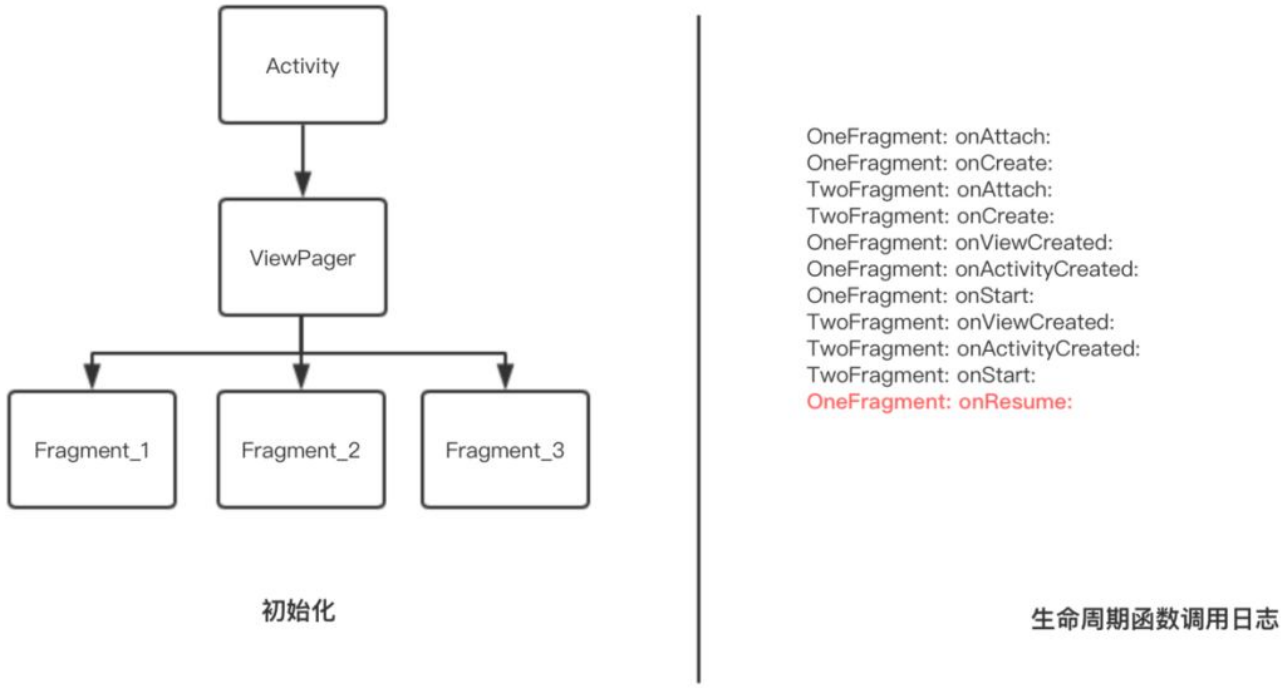
从官方的注释声明中，我们能得到如下两条结论：

- 如果 behavior 的值为 `BEHAVIOR_SET_USER_VISIBLE_HINT`，那么当 Fragment 对用户的可见状态发生改变时，`setUserVisibleHint` 方法会被调用。
- 如果 behavior 的值为 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT`，那么当前选中的 Fragment 在 `Lifecycle.State#RESUMED` 状态，其他不可见的 Fragment 会被限制在 `Lifecycle.State#STARTED` 状态。

那 BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT 这个值到底有什么作用呢？我们看下面的例子：

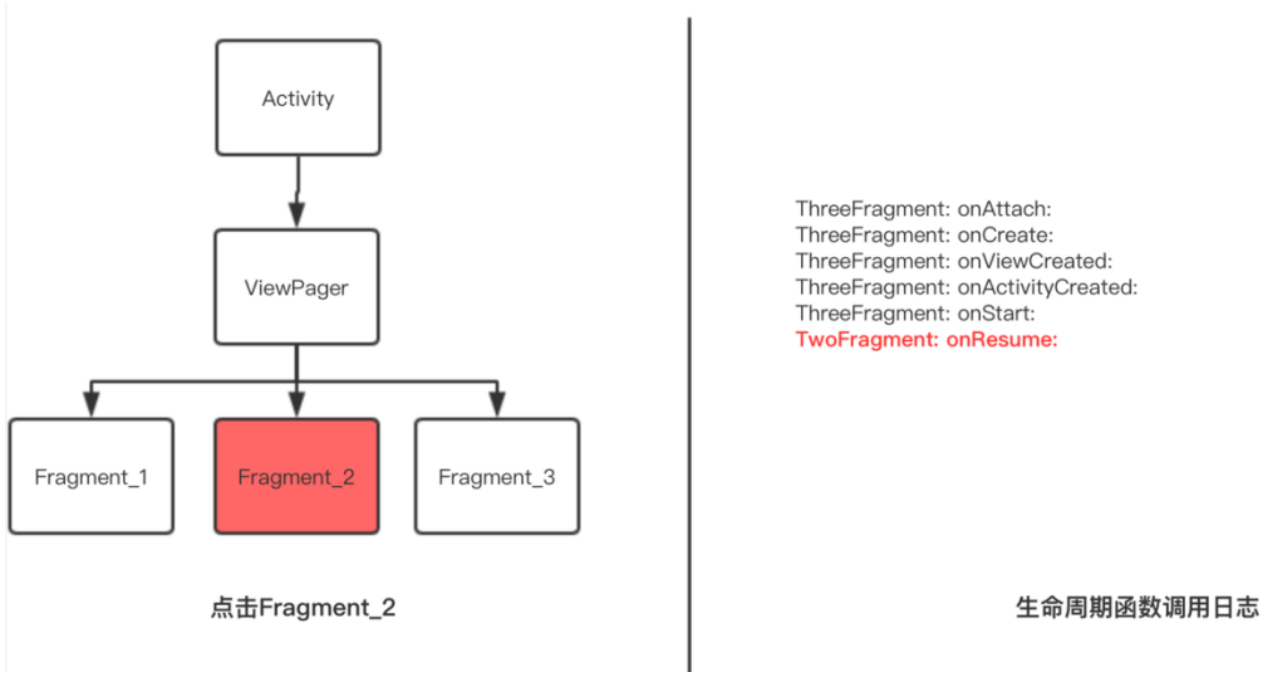
> 在该例子中设置了 ViewPager 的适配器为 FragmentPagerAdapter 且 behavior 值为 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT`。

默认初始化ViewPager，Fragment 生命周期如下所示：



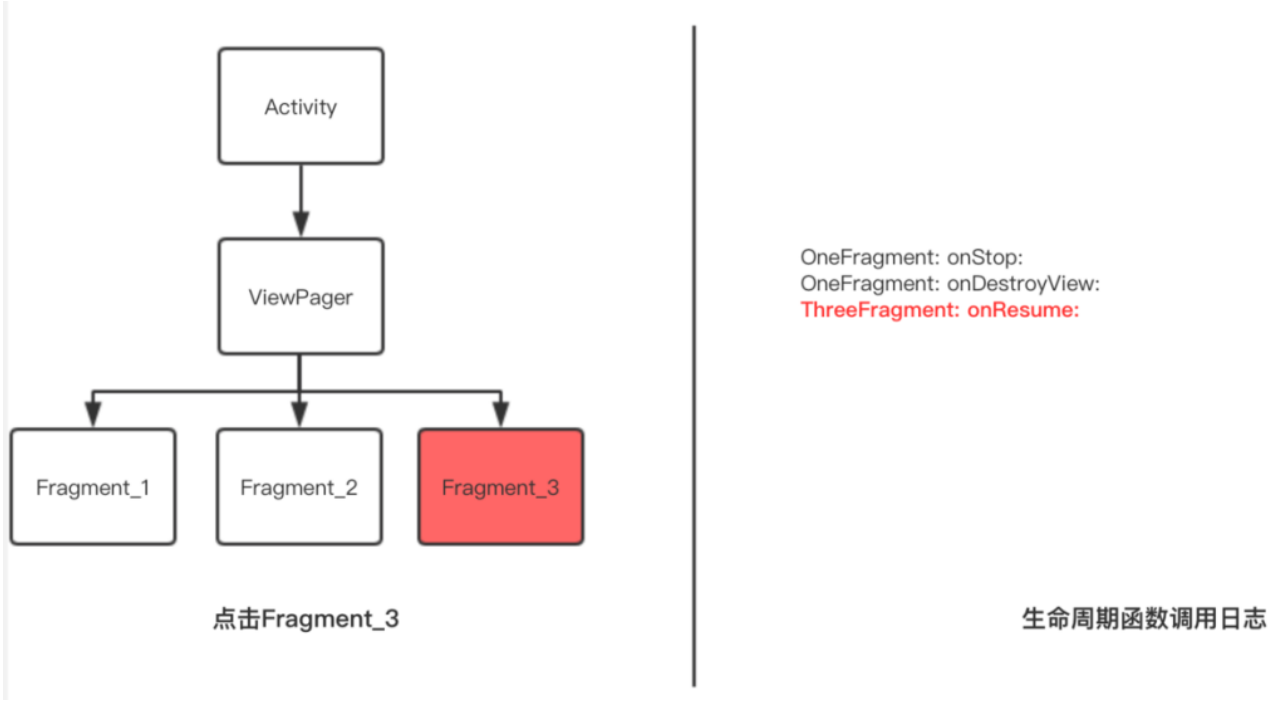
androix1.png

切换到 Fragment_2 时，日志情况如下所示：



androix2.png

切换到 Fragment_3 时，日志情况如下所示:



androidx3.png

> 因为篇幅的原因，本文没有在讲解 FragmentStatePagerAdapter 设置 behavior 下的使用情况，但是原理以及生命周期函数调用情况一样，感兴趣的小伙伴，可以根据 AndroidxLazyLoad 项目自行测试。

观察上述例子，我们可以发现，使用了 `BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT` 后，确实只有当前可见的 Fragment 调用了 onResume 方法。而导致产生这种改变的原因，是因为 FragmentPagerAdapter 在其 `setPrimaryItem` 方法中调用了 `setMaxLifecycle` 方法，如下所示：

```
public void setPrimaryItem(@NonNull ViewGroup container, int position, @NonNull Obje
    Fragment fragment = (Fragment)object;
    //如果当前的fragment不是当前选中并可见的Fragment,那么就会调用
    // setMaxLifecycle 设置其最大生命周期为 Lifecycle.State.STARTED
    if (fragment != mCurrentPrimaryItem) {
        if (mCurrentPrimaryItem != null) {
            mCurrentPrimaryItem.setMenuVisibility(false);
            if (mBehavior == BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT) {
                if (mCurTransaction == null) {
                    mCurTransaction = mFragmentManager.beginTransaction();
                }
                mCurTransaction.setMaxLifecycle(mCurrentPrimaryItem, Lifecycle.State
            } else {
                mCurrentPrimaryItem.setUserVisibleHint(false);
            }
        }
    //对于其他非可见的Fragment,则设置其最大生命周期为
    //Lifecycle.State.RESUMED
        fragment.setMenuVisibility(true);
        if (mBehavior == BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT) {
            if (mCurTransaction == null) {
                mCurTransaction = mFragmentManager.beginTransaction();
            }
```

```
                mCurTransaction.setMaxLifecycle(fragment, Lifecycle.State.RESUMED);
            } else {
                fragment.setUserVisibleHint(true);
            }

            mCurrentPrimaryItem = fragment;
        }
    }
```

既然在上述条件下，只有实际可见的 Fragment 会调用 onResume 方法， 那是不是为我们提供了 ViewPager 下实现懒加载的新思路呢？也就是我们可以这样实现 Fragment 的懒加载：

```
abstract class LazyFragment : Fragment() {

    private var isLoaded = false

    override fun onResume() {
        super.onResume()
        if (!isLoaded) {
            lazyInit()
            Log.d(TAG, "lazyInit:!!!!!!!")
            isLoaded = true
        }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        isLoaded = false
    }

    abstract fun lazyInit()
}
```

add+show+hide 模式下的新方案

虽然我们实现了Androidx 包下 ViewPager下的懒加载，但是我们仍然要考虑 add+show+hide 模式下的 Fragment 懒加载的情况，基于 ViewPager 在 `setPrimaryItem` 方法中的思路，我们可以在调用 add+show+hide 时，这样处理：

> 完整的代码请点击--->ShowHideExt

```
/**
 * 使用add+show+hide模式加载fragment
 *
 * 默认显示位置[showPosition]的Fragment，最大Lifecycle为Lifecycle.State.RESUMED
 * 其他隐藏的Fragment，最大Lifecycle为Lifecycle.State.STARTED
 *
 *@param containerViewId 容器id
```

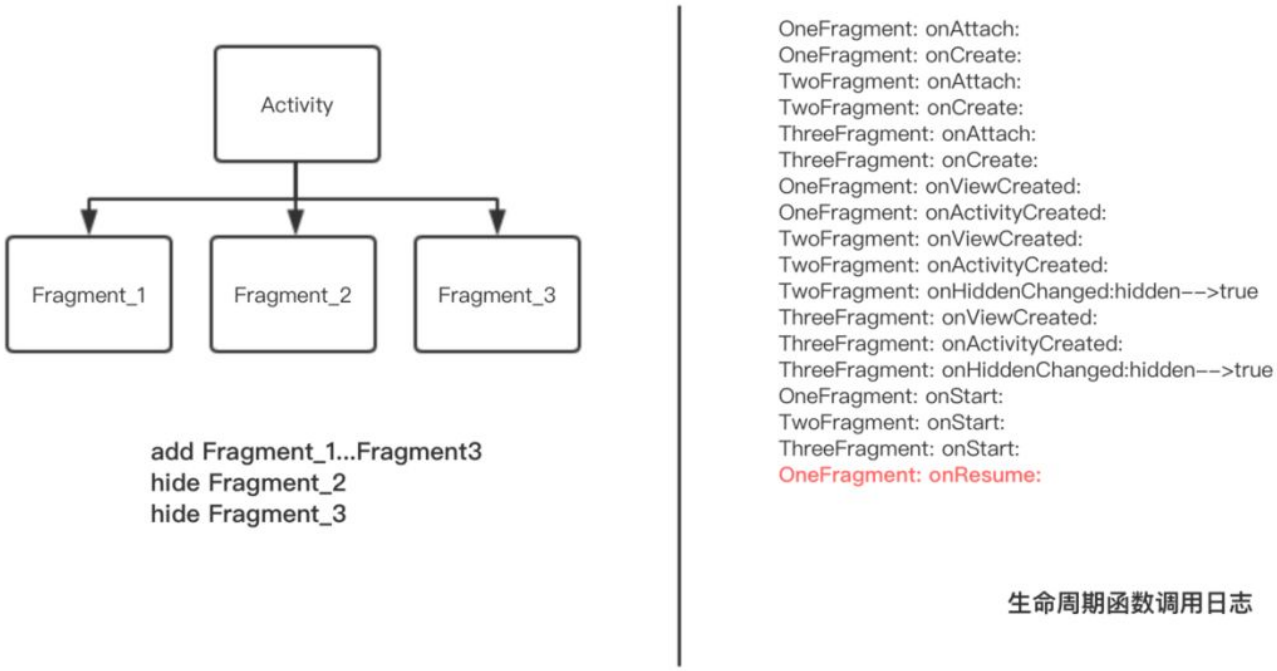
```
     *@param showPosition  fragments
     *@param fragmentManager FragmentManager
     *@param fragments  控制显示的Fragments
     */
    private fun loadFragmentsTransaction(
        @IdRes containerViewId: Int,
        showPosition: Int,
        fragmentManager: FragmentManager,
        vararg fragments: Fragment
    ) {
        if (fragments.isNotEmpty()) {
            fragmentManager.beginTransaction().apply {
                for (index in fragments.indices) {
                    val fragment = fragments[index]
                    add(containerViewId, fragment, fragment.javaClass.name)
                    if (showPosition == index) {
                        setMaxLifecycle(fragment, Lifecycle.State.RESUMED)
                    } else {
                        hide(fragment)
                        setMaxLifecycle(fragment, Lifecycle.State.STARTED)
                    }
                }

            }.commit()
        } else {
            throw IllegalStateException(
                "fragments must not empty"
            )
        }
    }

    /** 显示需要显示的Fragment[showFragment]，并设置其最大Lifecycle为Lifecycle.State.RESUMED。
     *  同时隐藏其他Fragment,并设置最大Lifecycle为Lifecycle.State.STARTED
     * @param fragmentManager
     * @param showFragment
     */
    private fun showHideFragmentTransaction(fragmentManager: FragmentManager, showFragment:
        fragmentManager.beginTransaction().apply {
            show(showFragment)
            setMaxLifecycle(showFragment, Lifecycle.State.RESUMED)

            //获取其中所有的fragment,其他的fragment进行隐藏
            val fragments = fragmentManager.fragments
            for (fragment in fragments) {
                if (fragment != showFragment) {
                    hide(fragment)
                    setMaxLifecycle(fragment, Lifecycle.State.STARTED)
                }
            }
        }.commit()
    }
```

上述代码的实现也非常简单：

- 将需要显示的 Fragment，在调用 add 或 show 方法后，`setMaxLifecycle(showFragment, Lifecycle.State.RESUMED)`
- 将需要隐藏的 Fragment，在调用 hide 方法后，`setMaxLifecycle(fragment, Lifecycle.State.STARTED)`

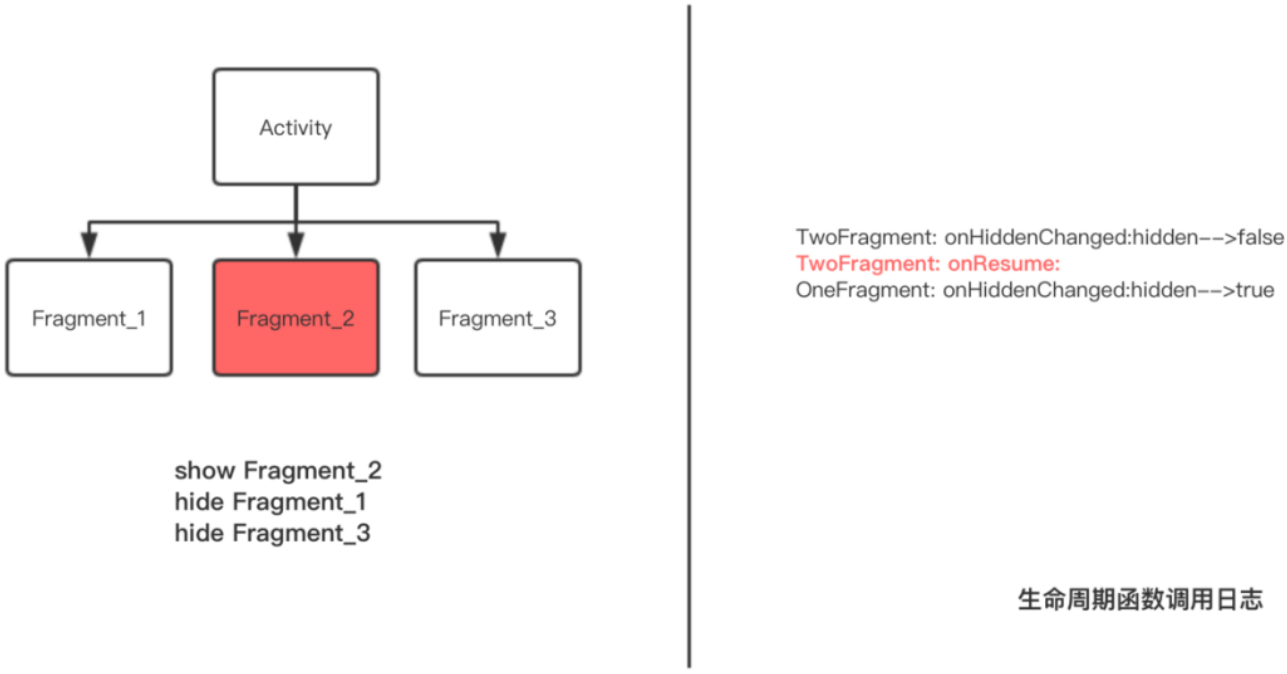
结合上述操作模式，查看使用 setMaxLifecycle 后，Fragment 生命周期函数调用的情况。

add Fragment_1、Fragment_2、Fragment_3，并 hide Fragment_2,Fragment_3
:



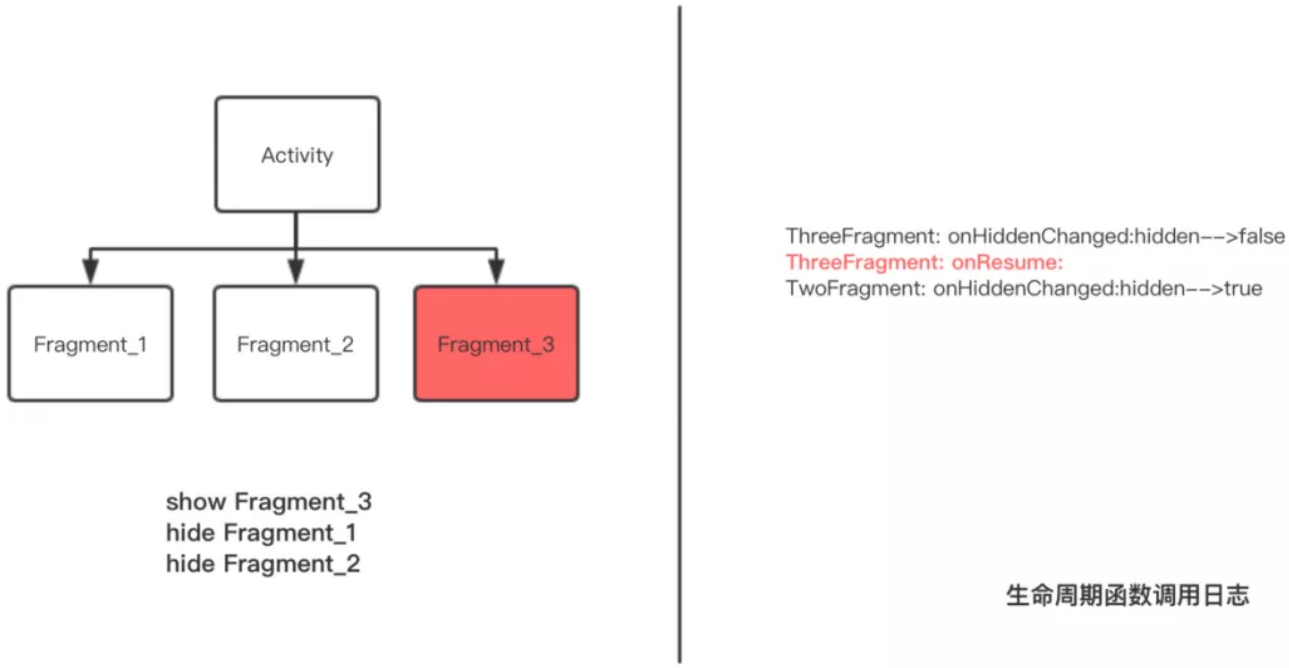
show_new1.png

show Fragment_2，hide 其他 Fragment:



show_new2.png

show Fragment_3 hide 其他 Fragment:

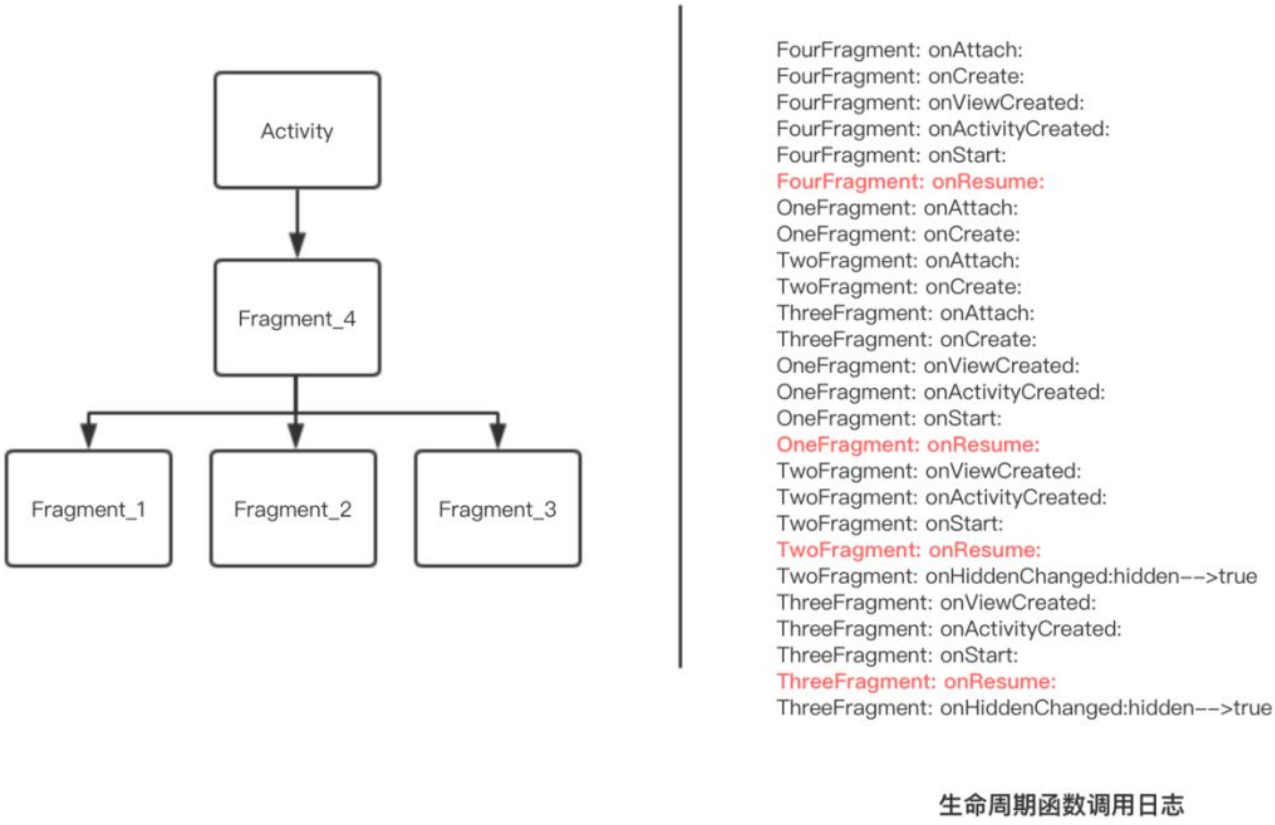


show_new3.png

参考上图，好像真的也能处理懒加载！！！！！美滋滋

并不完美的 setMaxLifecycle

当我第一次使用 setMaxLifycycle 方法时，我也和大家一样觉得万事大吉。但这套方案仍然有点点瑕疵，当 Fragment 的嵌套时，即使使用了 setMaxLifycycle 方法，第一次初始化时，同级不可见的Fragment，仍然 TMD 要调用可见生命周期方法。看下面的例子：



瑕疵.png

不知道是否是谷歌大大没有考虑到 Fragment 嵌套的情况，所以这里我们要对之前的方案就行修改，也就是如下所示：

```
abstract class LazyFragment : Fragment() {

    private var isLoaded = false

    override fun onResume() {
        super.onResume()
        //增加了Fragment是否可见的判断
        if (!isLoaded && !isHidden) {
            lazyInit()
            Log.d(TAG, "lazyInit:!!!!!!!")
            isLoaded = true
        }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        isLoaded = false
    }

    abstract fun lazyInit()

}
```

在上述代码中，因为同级的 Fragment 在嵌套模式下，仍然要调用 onResume 方法，所以我们增加了 Fragment 可见性的判断，这样就能保证嵌套模式下，新方案也能完美的支持 Fragment 的懒

加载。

ViewPager2 的处理方案

ViewPager2 本身就支持对实际可见的 Fragment 才调用 onResume 方法。关于 ViewPager2 的内部机制。感兴趣的小伙伴可以自行查看源码。

关于 ViewPager2 的懒加载测试，已上传至 AndroidxLazyLoad，大家可以结合项目查看Log日志。

两种方式的对比与总结

老一套的懒加载

- 优点：不用去控制 FragmentManager的 add+show+hide 方法，所有的懒加载都是在 Fragment 内部控制，也就是控制 `setUserVisibleHint + onHiddenChanged` 这两个函数。
- 缺点：实际不可见的 Fragment，其 `onResume` 方法任然会被调用，这种反常规的逻辑，无法容忍。

新一套的懒加载（Androidx下setMaxLifecycle）

- 优点： 在非特殊的情况下(缺点1)，只有实际的可见 Fragment，其 `onResume` 方法才会被调用，这样才符合方法设计的初衷。
- 缺点：

1. 对于 Fragment 的嵌套，及时使用了 `setMaxLifecycle` 方法。同级不可见的Fragment，仍然要调用 `onResume` 方法。
2. 需要在原有的 add+show+hide 方法中，继续调用 setMaxLifecycle 方法来控制Fragment 的最大生命状态。

最后

这两种方案的优缺点已经非常明显了，到底该选择何种懒加载模式，还是要基于大家的意愿，作者我更倾向于使用新的方案。关于 Fragment 的懒加载实现，非常愿意听到大家不同的声音，如果你有更好的方案，可以在评论区留下您的 idea，期待您的回复。如果您觉得本篇文章对你有所帮助，请不要吝啬你的关注与点赞。ლ(´･ᴗ･`)比心

阅读原文

喜欢此内容的人还喜欢

前端都是手写ECharts？

掘金开发者社区

没什么想说的，就希望她过好日子

新浪娱乐

“简历上有这种经历的大学生，我们抢着要！”

朱伟老师