

Fragment 的使用

实现很简单，创建一个布局，然后在 Activity 里点击时替换 Fragment。

```
1 mFragmentManager = getSupportFragmentManager();
2 mFragmentManager.beginTransaction()
3     .replace(R.id.fl_content, fragment)
4     .commitAllowingStateLoss();
```

代码很简单，核心就三步：

1. 创建 Fragment
2. 获取 FragmentManager
3. 调用事务，添加、替换

我们一步步来了解这背后的故事。

Fragment 大家应该比较熟悉，放到最后。

先来看看 `FragmentManager`。

FragmentManager

```
1 public abstract class FragmentManager {...}
```

`FragmentManager` 是一个抽象类，定义了一些和 Fragment 相关的操作和内部类/接口。

定义的操作

`FragmentManager` 中定义的方法如下：

```
1 //开启一系列对 Fragments 的操作
2 public abstract FragmentTransaction beginTransaction();
3
4 //FragmentManager.commit() 是异步执行的，如果你想立即执行，可以调用这个方法
5 public abstract boolean executePendingTransactions();
6
7 //根据 ID 找到从 XML 解析出来的或者事务中添加的 Fragment
8 //首先会找添加到 FragmentManager 中的，找不到就去回退栈里找
9 public abstract Fragment findFragmentById(@IdRes int id);
10
11 //跟上面的类似，不同的是使用 tag 进行查找
12 public abstract Fragment findFragmentByTag(String tag);
13
14 //弹出回退栈中栈顶的 Fragment，异步执行的
15 public abstract void popBackStack();
16
17 //立即弹出回退栈中栈顶的，直接执行哦
18 public abstract boolean popBackStackImmediate();
19
```

```

20 //返回栈顶符合名称的，如果传入的 name 不为空，在栈中间找到了 Fragment，那将弹出这个
    Fragment 上面的所有 Fragment
21 //有点类似启动模式的 singleTask 的感觉
22 //如果传入的 name 为 null，那就和 popBackStack() 一样了
23 //异步执行
24 public abstract void popBackStack(String name, int flags);
25
26 //同步版的上面
27 public abstract boolean popBackStackImmediate(String name, int flags);
28
29 //和使用 name 查找、弹出一样
30 //不同的是这里的 id 是 FragmentTransaction.commit() 返回的 id
31 public abstract void popBackStack(int id, int flags);
32 //你懂得
33 public abstract boolean popBackStackImmediate(int id, int flags);
34
35 //获取回退栈中的元素个数
36 public abstract int getBackStackEntryCount();
37 //根据索引获取回退栈中的某个元素
38 public abstract BackStackEntry getBackStackEntryAt(int index);
39
40 //添加或者移除一个监听器
41 public abstract void
    addOnBackStackChangeListener(OnBackStackChangeListener listener);
42 public abstract void
    removeOnBackStackChangeListener(OnBackStackChangeListener listener);
43
44 //还定义了将一个 Fragment 实例作为参数传递
45 public abstract void putFragment(Bundle bundle, String key, Fragment
    fragment);
46 public abstract Fragment getFragment(Bundle bundle, String key);
47
48 //获取 manager 中所有添加进来的 Fragment
49 public abstract List<Fragment> getFragments();

```

可以看到，定义的方法有很多是异步执行的，后面看看它究竟是如何实现的异步。

内部类/接口：

- BackStackEntry: Fragment 后退栈中的一个元素
- onBackStackChangeListener: 后退栈变动监听器
- FragmentLifecycleCallbacks: FragmentManager 中的 Fragment 生命周期监听

```

1 //后退栈中的一个元素
2 public interface BackStackEntry {
3     //栈中该元素的唯一标识
4     public int getId();
5
6     //获取 FragmentTransaction#addToBackStack(String) 设置的名称
7     public String getName();
8
9     @StringRes
10    public int getBreadCrumbTitleRes();
11    @StringRes
12    public int getBreadCrumbShortTitleRes();
13    public CharSequence getBreadCrumbTitle();

```

```

14     public CharSequence getBreadCrumbShortTitle();
15 }

```

可以看到 `BackStackEntry` 的接口比较简单，关键信息就是 ID 和 Name。

```

1  //在 Fragment 回退栈中有变化时回调
2  public interface OnBackStackChangeListener {
3      public void onBackStackChanged();
4  }
5  //FragmentManager 中的 Fragment 生命周期监听
6      public abstract static class FragmentLifecycleCallbacks {
7          public void onFragmentPreAttached(FragmentManager fm, Fragment f,
Context context) {}
8          public void onFragmentAttached(FragmentManager fm, Fragment f,
Context context) {}
9          public void onFragmentCreated(FragmentManager fm, Fragment f,
Bundle savedInstanceState) {}
10         public void onFragmentActivityCreated(FragmentManager fm, Fragment
f,
11             Bundle savedInstanceState) {}
12         public void onFragmentViewCreated(FragmentManager fm, Fragment f,
View v,
13             Bundle savedInstanceState) {}
14         public void onFragmentStarted(FragmentManager fm, Fragment f) {}
15         public void onFragmentResumed(FragmentManager fm, Fragment f) {}
16         public void onFragmentPaused(FragmentManager fm, Fragment f) {}
17         public void onFragmentStopped(FragmentManager fm, Fragment f) {}
18         public void onFragmentSaveInstanceState(FragmentManager fm,
Fragment f, Bundle outState) {}
19         public void onFragmentViewDestroyed(FragmentManager fm, Fragment f)
{}
20         public void onFragmentDestroyed(FragmentManager fm, Fragment f) {}
21         public void onFragmentDetached(FragmentManager fm, Fragment f) {}
22     }
23 }

```

熟悉 Fragment 生命周期的同学一定觉得很面熟，这个接口就是为我们提供一个 `FragmentManager` 所有 Fragment 生命周期变化的回调。

小结：

可以看到，`FragmentManager` 是一个抽象类，它定义了对一个 Activity/Fragment 中 **添加进来的 Fragment 列表**、**Fragment 回退栈** 的操作、管理。

实现类 `FragmentManagerImpl`

`FragmentManager` 定义的任务是由 `FragmentManagerImpl` 实现的。

主要成员：

```

1  final class FragmentManagerImpl extends FragmentManager implements
LayoutInflaterFactory {
2
3      ArrayList<OpGenerator> mPendingActions;
4      Runnable[] mTmpActions;
5      boolean mExecutingActions;

```

```

6
7     ArrayList<Fragment> mActive;
8     ArrayList<Fragment> mAdded;
9     ArrayList<Integer> mAvailIndices;
10    ArrayList<BackStackRecord> mBackStack;
11    ArrayList<Fragment> mCreatedMenus;
12
13    // Must be accessed while locked.
14    ArrayList<BackStackRecord> mBackStackIndices;
15    ArrayList<Integer> mAvailBackStackIndices;
16
17    ArrayList<OnBackStackChangeListener> mBackStackChangeListeners;
18    private CopyOnWriteArrayList<Pair<FragmentLifecycleCallbacks, Boolean>>
mLifecycleCallbacks;
19    //...
20 }

```

可以看到，`FragmentManagerImpl` 中定义了 添加的、活跃的。以及回退栈的列表，这和 `FragmentManager` 的要求一致。

```

1  int mCurState = Fragment.INITIALIZING;
2  FragmentHostCallback mHost;
3  FragmentContainer mContainer;
4  Fragment mParent;
5
6  static Field sAnimationListenerField = null;
7
8  boolean mNeedMenuInvalidate;
9  boolean mStateSaved;
10 boolean mDestroyed;
11 String mNoTransactionsBecause;
12 boolean mHavePendingDeferredStart;

```

接着还有当前的状态，当前 `Fragment` 的起始 `mParent`，以及 `FragmentManager` 的 `mHost` 和 `mContainer`。

`FragmentContainer` 就是一个接口，定义了关于布局的两个方法：

```

1  public abstract class FragmentContainer {
2      @Nullable
3      public abstract view onFindViewById(@IdRes int id);
4      public abstract boolean onHasView();
5  }

```

而 `FragmentHostCallback` 就复杂一点了，它提供了 `Fragment` 需要的信息，也定义了 `Fragment` 宿主应该做的操作：

```

1  public abstract class FragmentHostCallback<E> extends FragmentContainer {
2      private final Activity mActivity;
3      final Context mContext;
4      private final Handler mHandler;
5      final int mWindowAnimations;
6      final FragmentManagerImpl mFragmentManager = new FragmentManagerImpl();
7      //...
8  }

```

我们知道，一般来说 Fragment 的宿主就两种：

1. Activity
2. Fragment

比如 `FragmentActivity` 的内部类 `HostCallbacks` 就实现了这个抽象类：

```
1  class HostCallbacks extends FragmentHostCallback<FragmentActivity> {
2      public HostCallbacks() {
3          super(FragmentActivity.this /*fragmentActivity*/);
4      }
5      //...
6
7      @Override
8      public LayoutInflater onGetLayoutInflater() {
9          return
10         FragmentActivity.this.getLayoutInflater().cloneInContext(FragmentActivity.t
11         his);
12     }
13
14     @Override
15     public FragmentActivity onGetHost() {
16         return FragmentActivity.this;
17     }
18
19     @Override
20     public void onStartActivityFromFragment(Fragment fragment, Intent
21     intent, int requestCode) {
22         FragmentActivity.this.startActivityFromFragment(fragment, intent,
23         requestCode);
24     }
25
26     @Override
27     public void onStartActivityFromFragment(
28         Fragment fragment, Intent intent, int requestCode, @Nullable
29         Bundle options) {
30         FragmentActivity.this.startActivityFromFragment(fragment, intent,
31         requestCode, options);
32     }
33
34     @Override
35     public void onRequestPermissionsFromFragment(@NonNull Fragment
36     fragment,
37         @NonNull String[] permissions, int requestCode) {
38         FragmentActivity.this.requestPermissionsFromFragment(fragment,
39         permissions,
40         requestCode);
41     }
42
43     @Override
44     public boolean onShouldShowRequestPermissionRationale(@NonNull String
45     permission) {
46         return ActivityCompat.shouldShowRequestPermissionRationale(
47             FragmentActivity.this, permission);
48     }
49
50     @Override
51     public boolean onHasWindowAnimations() {
```

```

43         return getWindow() != null;
44     }
45
46     @Override
47     public void onAttachFragment(Fragment fragment) {
48         FragmentActivity.this.onAttachFragment(fragment);
49     }
50
51     @Nullable
52     @Override
53     public View onCreateView(int id) {
54         return FragmentActivity.this.findViewById(id);
55     }
56
57     @Override
58     public boolean onHasView() {
59         final Window w = getWindow();
60         return (w != null && w.peekDecorView() != null);
61     }
62 }

```

我们再看看他对 `FragmentManager` 定义的关键方法是如何实现的。

```

1  @Override
2  public FragmentTransaction beginTransaction() {
3      return new BackStackRecord(this);
4  }

```

`beginTransaction()` 返回一个新的 `BackStackRecord`，我们后面介绍。

前面提到了，`popBackStack()` 是一个异步操作，它是如何实现异步的呢？

```

1  @Override
2  public void popBackStack() {
3      enqueueAction(new PopBackStackState(null, -1, 0), false);
4  }
5  public void enqueueAction(OpGenerator action, boolean allowStateLoss) {
6      if (!allowStateLoss) {
7          checkStateLoss();
8      }
9      synchronized (this) {
10         if (mDestroyed || mHost == null) {
11             throw new IllegalStateException("Activity has been destroyed");
12         }
13         if (mPendingActions == null) {
14             mPendingActions = new ArrayList<>();
15         }
16         mPendingActions.add(action);
17         scheduleCommit();
18     }
19 }
20 private void scheduleCommit() {
21     synchronized (this) {
22         boolean postponeReady =
23             mPostponedTransactions != null &&
24             !mPostponedTransactions.isEmpty();

```

```

24         boolean pendingReady = mPendingActions != null &&
mPendingActions.size() == 1;
25         if (postponeReady || pendingReady) {
26             mHost.getHandler().removeCallbacks(mExecCommit);
27             mHost.getHandler().post(mExecCommit);
28         }
29     }
30 }

```

可以看到，调用到最后，是调用宿主中的 Handler 来发送任务的，so easy 嘛。其他的异步执行也是类似，就不赘述了。

后退栈相关方法：

```

1  ArrayList<BackStackRecord> mBackStack;
2  @Override
3  public int getBackStackEntryCount() {
4      return mBackStack != null ? mBackStack.size() : 0;
5  }
6
7  @Override
8  public BackStackEntry getBackStackEntryAt(int index) {
9      return mBackStack.get(index);
10 }

```

可以看到，开始事务和后退栈，返回/操作的都是 `BackStackRecord`，我们来了解了解它是何方神圣。

事务

`BackStackRecord` 继承了 `FragmentTransaction`：

```

1  final class BackStackRecord extends FragmentTransaction implements
2      FragmentManager.BackStackEntry, FragmentManagerImpl.OpGenerator {...}

```

先来看看 `FragmentTransaction`。

FragmentTransaction

`FragmentTransaction` 定义了一系列对 Fragment 的操作方法：

```

1  //它会调用 add(int, Fragment, String)，其中第一个参数传的是 0
2  public abstract FragmentTransaction add(Fragment fragment, String tag);
3
4  //它会调用 add(int, Fragment, String)，其中第三个参数是 null
5  public abstract FragmentTransaction add(@IdRes int containerViewId,
Fragment fragment);
6
7  //添加一个 Fragment 给 Activity 的最终实现
8  //第一个参数表示 Fragment 要放置的布局 id
9  //第二个参数表示要添加的 Fragment，【注意】一个 Fragment 只能添加一次
10 //第三个参数选填，可以给 Fragment 设置一个 tag，后续可以使用这个 tag 查询它
11 public abstract FragmentTransaction add(@IdRes int containerViewId,
Fragment fragment,
12     @Nullable String tag);

```

```

13
14 //调用 replace(int, Fragment, String), 第三个参数传的是 null
15 public abstract FragmentTransaction replace(@IdRes int containerViewId,
16     Fragment fragment);
17
18 //替换宿主中一个已经存在的 fragment
19 //这方法等价于先调用 remove(), 再调用 add()
20 public abstract FragmentTransaction replace(@IdRes int containerViewId,
21     Fragment fragment,
22     @Nullable String tag);
23
24 //移除一个已经存在的 fragment
25 //如果之前添加到宿主上, 那它的布局也会被移除
26 public abstract FragmentTransaction remove(Fragment fragment);
27
28 //隐藏一个已存的 fragment
29 //其实就是将添加到宿主上的布局隐藏
30 public abstract FragmentTransaction hide(Fragment fragment);
31
32 //显示前面隐藏的 fragment, 这只适用于之前添加到宿主上的 fragment
33 public abstract FragmentTransaction show(Fragment fragment);
34
35 //将指定的 fragment 将布局上解除
36 //当调用这个方法时, fragment 的布局已经销毁了
37 public abstract FragmentTransaction detach(Fragment fragment);
38
39 //当前面解除一个 fragment 的布局绑定后, 调用这个方法可以重新绑定
40 //这将导致该 fragment 的布局重建, 然后添加、展示到界面上
41 public abstract FragmentTransaction attach(Fragment fragment);

```

对 fragment 的操作基本就这几步, 我们知道, 要完成对 fragment 的操作, 最后还需要提交一下:

```

1 mFragmentManager.beginTransaction()
2     .replace(R.id.fl_child, getChildFragment())
3 //     .commit()
4     .commitAllowingStateLoss();

```

事务的四种提交方式

事务最终的提交方法有四种:

1. `commit()`
2. `commitAllowingStateLoss()`
3. `commitNow()`
4. `commitNowAllowingStateLoss()`

它们之间的特点及区别如下:

```

1 public abstract int commit();

```

`commit()` 在主线程中异步执行, 其实也是 Handler 抛出任务, 等待主线程调度执行。

注意：

`commit()` 需要在宿主 Activity 保存状态之前调用，否则会报错。

这是因为如果 Activity 出现异常需要恢复状态，在保存状态之后的 `commit()` 将会丢失，这和调用的初衷不符，所以会报错。

```
1 public abstract int commitAllowingStateLoss();
```

`commitAllowingStateLoss()` 也是异步执行，但它的不同之处在于，允许在 Activity 保存状态之后调用，也就是说它遇到状态丢失不会报错。

因此我们一般在界面状态出错是可以接受的情况下使用它。

```
1 public abstract void commitNow();
```

`commitNow()` 是同步执行的，立即提交任务。

前面提到 `FragmentManager.executePendingTransactions()` 也可以实现立即提交事务。但我们一般建议使用 `commitNow()`，因为另外那位是一下子执行所有待执行的任务，可能会把当前所有的事务都一下子执行了，这有可能有副作用。

此外，这个方法提交的事务可能不会被添加到 `FragmentManager` 的后退栈，因为你这样直接提交，有可能影响其他异步执行任务在栈中的顺序。

和 `commit()` 一样，`commitNow()` 也必须在 Activity 保存状态前调用，否则会抛异常。

```
1 public abstract void commitNowAllowingStateLoss();
```

同步执行的 `commitAllowingStateLoss()`。

OK，了解了 `FragmentTransaction` 定义的操作，去看看我们真正关心的、`beginTransaction()` 中返回的 `BackStackRecord`：

```
1 @Override
2 public FragmentTransaction beginTransaction() {
3     return new BackStackRecord(this);
4 }
```

事务真正实现/回退栈 BackStackRecord

`BackStackRecord` 既是对 `Fragment` 进行操作的事务的真正实现，也是 `FragmentManager` 中的回退栈的实现：

```
1 final class BackStackRecord extends FragmentTransaction implements
2     FragmentManager.BackStackEntry, FragmentManagerImpl.OpGenerator {...}
```

它的关键成员：

```
1 final FragmentManagerImpl mManager;
2
3 //Op 可选的状态值
4 static final int OP_NULL = 0;
5 static final int OP_ADD = 1;
6 static final int OP_REPLACE = 2;
```

```

7  static final int OP_REMOVE = 3;
8  static final int OP_HIDE = 4;
9  static final int OP_SHOW = 5;
10 static final int OP_DETACH = 6;
11 static final int OP_ATTACH = 7;
12
13 ArrayList<Op> mOps = new ArrayList<>();
14 static final class Op {
15     int cmd;    //状态
16     Fragment fragment;
17     int enterAnim;
18     int exitAnim;
19     int popEnterAnim;
20     int popExitAnim;
21 }
22
23 int mIndex = -1;    //栈中最后一个元素的索引

```

可以看到 Op 就是添加了状态和动画信息的 Fragment, `mOps` 就是栈中所有的 Fragment。

事务定义的方法它是如何实现的呢。

先看添加一个 Fragment 到布局 `add()` 的实现:

```

1  @Override
2  public FragmentTransaction add(int containerViewId, Fragment fragment) {
3      doAddOp(containerViewId, fragment, null, OP_ADD);
4      return this;
5  }
6
7  @Override
8  public FragmentTransaction add(int containerViewId, Fragment fragment,
9  String tag) {
10     doAddOp(containerViewId, fragment, tag, OP_ADD);
11     return this;
12 }
13 private void doAddOp(int containerViewId, Fragment fragment, String tag,
14 int opcode) {
15     final Class fragmentClass = fragment.getClass();
16     final int modifiers = fragmentClass.getModifiers();
17     if (fragmentClass.isAnonymousClass() || !Modifier.isPublic(modifiers)
18         || (fragmentClass.isMemberClass() &&
19             !Modifier.isStatic(modifiers))) {
20         throw new IllegalStateException("Fragment " +
21             fragmentClass.getCanonicalName()
22             + " must be a public static class to be properly recreated
23             from"
24             + " instance state.");
25     }
26
27     //1.修改添加的 fragmentManager 为当前栈的 manager
28     fragment.mFragmentManager = mManager;
29
30     if (tag != null) {
31         if (fragment.mTag != null && !tag.equals(fragment.mTag)) {
32             throw new IllegalStateException("Can't change tag of fragment "
33                 + fragment + ": was " + fragment.mTag

```

```

29         + " now " + tag);
30     }
31     fragment.mTag = tag;
32 }
33
34 if (containerViewId != 0) {
35     if (containerViewId == View.NO_ID) {
36         throw new IllegalArgumentException("Can't add fragment "
37             + fragment + " with tag " + tag + " to container view
with no id");
38     }
39     if (fragment.mFragmentId != 0 && fragment.mFragmentId !=
containerViewId) {
40         throw new IllegalStateException("Can't change container ID of
fragment "
41             + fragment + ": was " + fragment.mFragmentId
42             + " now " + containerViewId);
43     }
44     //2.设置宿主 ID 为布局 ID
45     fragment.mContainerId = fragment.mFragmentId = containerViewId;
46 }
47
48 //3.构造 Op
49 Op op = new Op();
50 op.cmd = opcmd;    //状态
51 op.fragment = fragment;
52 //4.添加到数组列表中
53 addOp(op);
54 }
55 void addOp(Op op) {
56     mOps.add(op);
57     op.enterAnim = mEnterAnim;
58     op.exitAnim = mExitAnim;
59     op.popEnterAnim = mPopEnterAnim;
60     op.popExitAnim = mPopExitAnim;
61 }

```

可以看到添加一个 Fragment 到布局很简单，概况一下就是：

修改 fragmentManager 和 ID，构造成 Op，设置状态信息，然后添加到列表里。

添加完了看看替换 `replace` 的实现：

```

1  @Override
2  public FragmentTransaction replace(int containerViewId, Fragment fragment)
3  {
4      return replace(containerViewId, fragment, null);
5  }
6  @Override
7  public FragmentTransaction replace(int containerViewId, Fragment fragment,
String tag) {
8      if (containerViewId == 0) {
9          throw new IllegalArgumentException("Must use non-zero
containerViewId");
10     }
11
12     doAddOp(containerViewId, fragment, tag, OP_REPLACE);

```

```
13     return this;
14 }
```

太可怕了，也是调用上面刚提到的 `doAddOp()`，不同之处在于第四个参数为 `OP_REPLACE`，看来之前小看了这个状态值！

再看其他方法的实现就很简单了，无非就是构造一个 `Op`，设置对应的状态值。

```
1  @Override
2  public FragmentTransaction remove(Fragment fragment) {
3      Op op = new Op();
4      op.cmd = OP_REMOVE;
5      op.fragment = fragment;
6      addOp(op);
7
8      return this;
9  }
10
11 @Override
12 public FragmentTransaction hide(Fragment fragment) {
13     Op op = new Op();
14     op.cmd = OP_HIDE;
15     op.fragment = fragment;
16     addOp(op);
17
18     return this;
19 }
20
21 @Override
22 public FragmentTransaction show(Fragment fragment) {
23     Op op = new Op();
24     op.cmd = OP_SHOW;
25     op.fragment = fragment;
26     addOp(op);
27
28     return this;
29 }
```

那这些状态值的不同是什么时候起作用的呢？

别忘了我们操作 `Fragment` 还有最后一步，提交。

看看这两个是怎么实现的：

```
1  @Override
2  public int commit() {
3      return commitInternal(false);
4  }
5
6  @Override
7  public int commitAllowingStateLoss() {
8      return commitInternal(true);
9  }
10 int commitInternal(boolean allowStateLoss) {
11     if (mCommitted) throw new IllegalStateException("commit already
12     called");
13     //...
```

```

13     mCommitted = true;
14     if (mAddToBackStack) {
15         mIndex = mManager.allocBackStackIndex(this);    //更新 index 信息
16     } else {
17         mIndex = -1;
18     }
19     mManager.enqueueAction(this, allowStateLoss);    //异步任务入队
20     return mIndex;
21 }
22 public void enqueueAction(OpGenerator action, boolean allowStateLoss) {
23     if (!allowStateLoss) {
24         checkStateLoss();
25     }
26     synchronized (this) {
27         if (mDestroyed || mHost == null) {
28             throw new IllegalStateException("Activity has been destroyed");
29         }
30         if (mPendingActions == null) {
31             mPendingActions = new ArrayList<>();
32         }
33         mPendingActions.add(action);
34         scheduleCommit();    //发送任务
35     }
36 }
37 private void scheduleCommit() {
38     synchronized (this) {
39         boolean postponeReady =
40             mPostponedTransactions != null &&
41             !mPostponedTransactions.isEmpty();
42         boolean pendingReady = mPendingActions != null &&
43             mPendingActions.size() == 1;
44         if (postponeReady || pendingReady) {
45             mHost.getHandler().removeCallbacks(mExecCommit);
46             mHost.getHandler().post(mExecCommit);
47         }
48     }
49 }

```

前面已经介绍过了，`FragmentManager.enqueueAction()` 最终是使用 Handler 实现的异步执行。

现在的问题是执行的任务是啥？

答案就是 Handler 发送的任务 `mExecCommit`：

```

1  Runnable mExecCommit = new Runnable() {
2      @Override
3      public void run() {
4          execPendingActions();
5      }
6  };
7  /**
8   * Only call from main thread!
9   * 更新 UI 嘛，肯定得在主线程
10  */
11  public boolean execPendingActions() {
12      ensureExecReady(true);
13  }

```

```

14     boolean didSomething = false;
15     while (generateOpsForPendingActions(mTmpRecords, mTmpIsPop)) {
16         mExecutingActions = true;
17         try {
18             optimizeAndExecuteOps(mTmpRecords, mTmpIsPop);    //这里是入口
19         } finally {
20             cleanupExec();
21         }
22         didSomething = true;
23     }
24
25     doPendingDeferredStart();
26
27     return didSomething;
28 }
29 private void optimizeAndExecuteOps(ArrayList<BackStackRecord> records,
30     ArrayList<Boolean> isRecordPop) {
31     if (records == null || records.isEmpty()) {
32         return;
33     }
34
35     if (isRecordPop == null || records.size() != isRecordPop.size()) {
36         throw new IllegalStateException("Internal error with the back
37 stack records");
38     }
39
40     // Force start of any postponed transactions that interact with
41     // scheduled transactions:
42     executePostponedTransaction(records, isRecordPop);
43
44     final int numRecords = records.size();
45     int startIndex = 0;
46     for (int recordNum = 0; recordNum < numRecords; recordNum++) {
47         final boolean canOptimize =
48 records.get(recordNum).mAllowOptimization;
49         if (!canOptimize) {
50             // execute all previous transactions
51             if (startIndex != recordNum) {
52                 //这里将 Ops 过滤一遍
53                 executeOpsTogether(records, isRecordPop, startIndex,
54 recordNum);
55             }
56             // execute all unoptimized pop operations together or one add
57             // operation
58             //...
59         }
60         if (startIndex != numRecords) {
61             executeOpsTogether(records, isRecordPop, startIndex, numRecords);
62         }
63     }
64     private void executeOpsTogether(ArrayList<BackStackRecord> records,
65         ArrayList<Boolean> isRecordPop, int startIndex, int endIndex) {
66         final boolean allowOptimization =
67 records.get(startIndex).mAllowOptimization;
68         boolean addToBackStack = false;
69         if (mTmpAddedFragments == null) {
70             mTmpAddedFragments = new ArrayList<>();
71         } else {

```

```

66         mTmpAddedFragments.clear();
67     }
68     if (mAdded != null) {
69         mTmpAddedFragments.addAll(mAdded);
70     }
71     for (int recordNum = startIndex; recordNum < endIndex; recordNum++) {
72         final BackStackRecord record = records.get(recordNum);
73         final boolean isPop = isRecordPop.get(recordNum);
74         if (!isPop) {
75             record.expandReplaceOps(mTmpAddedFragments);    //修改状态
76         } else {
77             record.trackAddedFragmentsInPop(mTmpAddedFragments);
78         }
79         addToBackStack = addToBackStack || record.mAddToBackStack;
80     }
81     mTmpAddedFragments.clear();
82
83     if (!allowOptimization) {
84         FragmentTransition.startTransitions(this, records, isRecordPop,
85             startIndex, endIndex,
86             false);
87     }
88     //真正处理的入口
89     executeOps(records, isRecordPop, startIndex, endIndex);
90
91     int postponeIndex = endIndex;
92     if (allowOptimization) {
93         ArraySet<Fragment> addedFragments = new ArraySet<>();
94         addAddedFragments(addedFragments);
95         postponeIndex = postponePostponableTransactions(records,
96             isRecordPop,
97             startIndex, endIndex, addedFragments);
98         makeRemovedFragmentsInvisible(addedFragments);    //名字就能看出来作
99         用
100     }
101
102     if (postponeIndex != startIndex && allowOptimization) {
103         // need to run something now
104         FragmentTransition.startTransitions(this, records, isRecordPop,
105             startIndex,
106             postponeIndex, true);
107         moveToState(mCurState, true);
108     }
109     //...
110 }
111 //修改 ops 状态，这一步还没有真正处理状态
112 expandReplaceOps(ArrayList<Fragment> added) {
113     for (int opNum = 0; opNum < mOps.size(); opNum++) {
114         final Op op = mOps.get(opNum);
115         switch (op.cmd) {
116             case OP_ADD:
117             case OP_ATTACH:
118                 added.add(op.fragment);
119                 break;
120             case OP_REMOVE:
121             case OP_DETACH:
122                 added.remove(op.fragment);
123                 break;

```

```

120         case OP_REPLACE: {
121             Fragment f = op.fragment;
122             int containerId = f.mContainerId;
123             boolean alreadyAdded = false;
124             for (int i = added.size() - 1; i >= 0; i--) {
125                 Fragment old = added.get(i);
126                 if (old.mContainerId == containerId) {
127                     if (old == f) {
128                         alreadyAdded = true;
129                     } else {
130                         op.removeOp = new Op();
131                         removeOp.cmd = OP_REMOVE; //可以看到，替换也是
132
133                         removeOp.fragment = old;
134                         removeOp.enterAnim = op.enterAnim;
135                         removeOp.popEnterAnim = op.popEnterAnim;
136                         removeOp.exitAnim = op.exitAnim;
137                         removeOp.popExitAnim = op.popExitAnim;
138                         mOps.add(opNum, removeOp);
139                         added.remove(old);
140                         opNum++;
141                     }
142                 }
143             }
144             if (alreadyAdded) {
145                 mOps.remove(opNum);
146                 opNum--;
147             } else {
148                 op.cmd = OP_ADD;
149                 added.add(f);
150             }
151             break;
152         }
153     }
154 }
155 //设置将要被移除的 Fragment 为不可见的最终实现
156 private void makeRemovedFragmentsInvisible(HashSet<Fragment> fragments) {
157     final int numAdded = fragments.size();
158     for (int i = 0; i < numAdded; i++) {
159         final Fragment fragment = fragments.valueAt(i);
160         if (!fragment.mAdded) {
161             final View view = fragment.getView(); //获取 Fragment 的布
162             if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
163                 fragment.getView().setVisibility(View.INVISIBLE);
164             } else { //高版本设置透明度
165                 fragment.mPostponedAlpha = view.getAlpha();
166                 view.setAlpha(0f);
167             }
168         }
169     }
170 }

```

代码多了一点，但我们终于找到了最终的实现：Handler 异步发到主线，调度执行后，聚合、修改 Ops 的状态，然后遍历、修改 Fragment 栈中的 View 的状态。

真正处理的部分

前面主要是对 Fragment 的包装类 Ops 进行一些状态修改，真正根据 Ops 状态进行操作在这个部分：

```
1  /**
2   * Executes the operations contained within this transaction. The Fragment
3   * states will only
4   * be modified if optimizations are not allowed.
5   */
6  void executeOps() {
7      final int numOps = mOps.size();
8      for (int opNum = 0; opNum < numOps; opNum++) {
9          final Op op = mOps.get(opNum);
10         final Fragment f = op.fragment;
11         f.setNextTransition(mTransition, mTransitionStyle);
12         switch (op.cmd) {
13             case OP_ADD:
14                 f.setNextAnim(op.enterAnim);
15                 mManager.addFragment(f, false);
16                 break;
17             case OP_REMOVE:
18                 f.setNextAnim(op.exitAnim);
19                 mManager.removeFragment(f);
20                 break;
21             case OP_HIDE:
22                 f.setNextAnim(op.exitAnim);
23                 mManager.hideFragment(f);
24                 break;
25             case OP_SHOW:
26                 f.setNextAnim(op.enterAnim);
27                 mManager.showFragment(f);
28                 break;
29             case OP_DETACH:
30                 f.setNextAnim(op.exitAnim);
31                 mManager.detachFragment(f);
32                 break;
33             case OP_ATTACH:
34                 f.setNextAnim(op.enterAnim);
35                 mManager.attachFragment(f);
36                 break;
37             default:
38                 throw new IllegalArgumentException("Unknown cmd: " +
39 op.cmd);
40         }
41         if (!mAllowOptimization && op.cmd != OP_ADD) {
42             mManager.moveFragmentToExpectedState(f);
43         }
44         if (!mAllowOptimization) {
45             // Added fragments are added at the end to comply with prior
46             behavior.
47             mManager.moveToState(mManager.mCurState, true);
48         }
49     }
50 }
```

FragmentManager 对这些方法的实现也很简单，修改 Fragment 的状态值，比如

`remove(Fragment)`：

```
1 public void removeFragment(Fragment fragment) {
2     if (DEBUG) Log.v(TAG, "remove: " + fragment + " nesting=" +
fragment.mBackStackNesting);
3     final boolean inactive = !fragment.isInBackStack();
4     if (!fragment.mDetached || inactive) {
5         if (mAdded != null) {
6             mAdded.remove(fragment);
7         }
8         if (fragment.mHasMenu && fragment.mMenuVisible) {
9             mNeedMenuInvalidate = true;
10        }
11        fragment.mAdded = false;    //设置属性值
12        fragment.mRemoving = true;
13    }
14 }
```

最终会调用 `moveToState()`，我们直接来看它的实现：

```
1 void moveToState(Fragment f, int newState, int transit, int
transitionStyle,
2     boolean keepActive) {
3     //还没有添加的 Fragment 处于 onCreate() 状态
4     if ((!f.mAdded || f.mDetached) && newState > Fragment.CREATED) {
5         newState = Fragment.CREATED;
6     }
7     if (f.mRemoving && newState > f.mState) {
8         // while removing a fragment, we can't change it to a higher
state.
9         newState = f.mState;
10    }
11    //推迟启动的设置为 stop
12    if (f.mDeferStart && f.mState < Fragment.STARTED && newState >
Fragment.STOPPED) {
13        newState = Fragment.STOPPED;
14    }
15    if (f.mState < newState) {
16        // For fragments that are created from a layout, when restoring
from
17        // state we don't want to allow them to be created until they are
18        // being reloaded from the layout.
19        if (f.mFromLayout && !f.mInLayout) {
20            return;
21        }
22        if (f.getAnimatingAway() != null) {
23            // The fragment is currently being animated... but! Now we
24            // want to move our state back up. Give up on waiting for the
25            // animation, move to whatever the final state should be once
26            // the animation is done, and then we can proceed from there.
27            f.setAnimatingAway(null);
28            //如果当前 Fragment 正有动画，直接修改为最终状态
29            moveToState(f, f.getStateAfterAnimating(), 0, 0, true);
30        }
31        switch (f.mState) {
```

```

32         case Fragment.INITIALIZING:
33             if (DEBUG) Log.v(TAG, "move to CREATED: " + f);
34             if (f.mSavedFragmentState != null) {
35
36                 f.mSavedViewState =
37                 f.mSavedFragmentState.getSparseParcelableArray(
38                     FragmentManagerImpl.VIEW_STATE_TAG);
39                 f.mTarget = getFragment(f.mSavedFragmentState,
40                     FragmentManagerImpl.TARGET_STATE_TAG);
41                 if (f.mTarget != null) {
42                     f.mTargetRequestCode =
43                     f.mSavedFragmentState.getInt(
44                         FragmentManagerImpl.TARGET_REQUEST_CODE_STATE_TAG, 0);
45                 }
46                 f.mUserVisibleHint = f.mSavedFragmentState.getBoolean(
47                     FragmentManagerImpl.USER_VISIBLE_HINT_TAG,
48                     true);
49                 if (!f.mUserVisibleHint) {
50                     f.mDeferStart = true;
51                     if (newState > Fragment.STOPPED) {
52                         newState = Fragment.STOPPED;
53                     }
54                 }
55                 f.mHost = mHost;
56                 f.mParentFragment = mParent;
57                 f.mFragmentManager = mParent != null
58                     ? mParent.mChildFragmentManager :
59                     mHost.getFragmentManagerImpl();
60                 dispatchOnFragmentPreAttached(f, mHost.getContext(),
61                     false);
62                 f.mCalled = false;
63                 f.onAttach(mHost.getContext()); //调用 Fragment 生命周期
64                 方法
65                 if (!f.mCalled) {
66                     throw new SuperNotCalledException("Fragment " + f
67                         + " did not call through to
68                         super.onAttach()");
69                 }
70                 if (f.mParentFragment == null) {
71                     mHost.onAttachFragment(f);
72                 } else {
73                     f.mParentFragment.onAttachFragment(f);
74                 }
75                 dispatchOnFragmentAttached(f, mHost.getContext(), false);
76                 if (!f.mRetaining) {
77                     f.performCreate(f.mSavedFragmentState); //调用 Fragment
78                     生命周期方法
79                     dispatchOnFragmentCreated(f, f.mSavedFragmentState,
80                         false);
81                 } else {
82                     f.restoreChildFragmentState(f.mSavedFragmentState);
83                     f.mState = Fragment.CREATED;

```

```

78         }
79         f.mRetaining = false;
80         if (f.mFromLayout) { //从布局解析来的
81             // For fragments that are part of the content view
82             // layout, we need to instantiate the view immediately
83             // and the inflater will take care of adding it.
84             f.mView = f.performCreateView(f.getLayoutInflater(
//调用 Fragment 生命周期方法
85                 f.mSavedFragmentState), null,
f.mSavedFragmentState);
86             if (f.mView != null) {
87                 f.mInnerView = f.mView;
88                 if (Build.VERSION.SDK_INT >= 11) {
89                     ViewCompat.setSaveFromParentEnabled(f.mView,
false);
90                 } else {
91                     f.mView =
NoSaveStateFrameLayout.wrap(f.mView);
92                 }
93                 if (f.mHidden) f.mView.setVisibility(View.GONE);
94                 f.onViewCreated(f.mView, f.mSavedFragmentState);
//调用 Fragment 生命周期方法
95                 dispatchOnFragmentViewCreated(f, f.mView,
f.mSavedFragmentState, false);
96             } else {
97                 f.mInnerView = null;
98             }
99         }
100         case Fragment.CREATED:
101             if (newState > Fragment.CREATED) {
102                 if (DEBUG) Log.v(TAG, "moveto ACTIVITY_CREATED: " +
f);
103                 if (!f.mFromLayout) {
104                     ViewGroup container = null;
105                     if (f.mContainerId != 0) {
106                         if (f.mContainerId == View.NO_ID) {
107                             throwException(new
IllegalArgumentException(
108                                 "Cannot create fragment "
109                                     + f
110                                     + " for a container view
with no id"));
111                         }
112                         container = (ViewGroup)
mContainer.onFindViewById(f.mContainerId);
113                         if (container == null && !f.mRestored) {
114                             String resName;
115                             try {
116                                 resName =
f.getResources().getResourceName(f.mContainerId);
117                             } catch (NotFoundException e) {
118                                 resName = "unknown";
119                             }
120                             throwException(new
IllegalArgumentException(
121                                 "No view found for id 0x"
122                                     +
Integer.toHexString(f.mContainerId) + " ("

```

```

123         + resName
124         + ") for fragment " + f));
125     }
126 }
127 f.mContainer = container;
128 f.mView = f.performCreateView(f.getLayoutInflater(
//调用 Fragment 生命周期方法
129     f.mSavedFragmentState), container,
f.mSavedFragmentState);
130     if (f.mView != null) {
131         f.mInnerView = f.mView;
132         if (Build.VERSION.SDK_INT >= 11) {
133
134             ViewCompat.setSaveFromParentEnabled(f.mView, false);
135             } else {
136                 f.mView =
137                 NoSaveStateFrameLayout.wrap(f.mView);
138             }
139             if (container != null) {
140                 container.addView(f.mView); //将
141                 Fragment 的布局添加到父布局中
142             }
143             if (f.mHidden) {
144                 f.mView.setVisibility(View.GONE);
145             }
146             f.onViewCreated(f.mView,
147                 f.mSavedFragmentState); //调用 Fragment 生命周期方法
148             dispatchOnFragmentViewCreated(f, f.mView,
149                 f.mSavedFragmentState,
150                 false);
151             // Only animate the view if it is visible.
152             This is done after
153             // dispatchOnFragmentViewCreated in case
154             visibility is changed
155             f.mIsNewlyAdded = (f.mView.getVisibility() ==
156                 View.VISIBLE)
157                 && f.mContainer != null;
158             } else {
159                 f.mInnerView = null;
160             }
161         }
162     }
163     f.performActivityCreated(f.mSavedFragmentState); //调用
164     Fragment 生命周期方法
165     dispatchOnFragmentActivityCreated(f,
166     f.mSavedFragmentState, false);
167     if (f.mView != null) {
168         f.restoreViewState(f.mSavedFragmentState);
169     }
170     f.mSavedFragmentState = null;
171 }
172 case Fragment.ACTIVITY_CREATED:
173     if (newState > Fragment.ACTIVITY_CREATED) {
174         f.mState = Fragment.STOPPED;
175     }
176 case Fragment.STOPPED:
177     if (newState > Fragment.STOPPED) {

```

```

169         if (DEBUG) Log.v(TAG, "moveto STARTED: " + f);
170         f.performStart();
171         dispatchOnFragmentStarted(f, false);
172     }
173     case Fragment.STARTED:
174         if (newState > Fragment.STARTED) {
175             if (DEBUG) Log.v(TAG, "moveto RESUMED: " + f);
176             f.performResume();
177             dispatchOnFragmentResumed(f, false);
178             f.mSavedFragmentState = null;
179             f.mSavedViewState = null;
180         }
181     }
182 } else if (f.mState > newState) {
183     switch (f.mState) {
184         case Fragment.RESUMED:
185             if (newState < Fragment.RESUMED) {
186                 if (DEBUG) Log.v(TAG, "movefrom RESUMED: " + f);
187                 f.performPause();
188                 dispatchOnFragmentPaused(f, false);
189             }
190         case Fragment.STARTED:
191             if (newState < Fragment.STARTED) {
192                 if (DEBUG) Log.v(TAG, "movefrom STARTED: " + f);
193                 f.performStop();
194                 dispatchOnFragmentStopped(f, false);
195             }
196         case Fragment.STOPPED:
197             if (newState < Fragment.STOPPED) {
198                 if (DEBUG) Log.v(TAG, "movefrom STOPPED: " + f);
199                 f.performReallyStop();
200             }
201         case Fragment.ACTIVITY_CREATED:
202             if (newState < Fragment.ACTIVITY_CREATED) {
203                 if (DEBUG) Log.v(TAG, "movefrom ACTIVITY_CREATED: " +
204 f);
205                 if (f.mView != null) {
206                     // Need to save the current view state if not
207                     // done already.
208                     if (mHost.onShouldSaveFragmentState(f) &&
209 f.mSavedViewState == null) {
210                         saveFragmentViewState(f);
211                     }
212                 }
213                 f.performDestroyView();
214                 dispatchOnFragmentViewDestroyed(f, false);
215                 if (f.mView != null && f.mContainer != null) {
216                     Animation anim = null;
217                     if (mCurState > Fragment.INITIALIZING &&
218 !mDestroyed
219                         && f.mView.getVisibility() == View.VISIBLE
220                         && f.mPostponedAlpha >= 0) {
221                         anim = loadAnimation(f, transit, false,
222 transitionStyle);
223                     }
224                     f.mPostponedAlpha = 0;
225                     if (anim != null) {
226                         final Fragment fragment = f;

```

```

224         f.setAnimatingAway(f.mView);
225         f.setStateAfterAnimating(newState);
226         final View viewToAnimate = f.mView;
227         anim.setAnimationListener(new
AnimateOnHWLayerIfNeededListener(
228             viewToAnimate, anim) {
229             @Override
230             public void onAnimationEnd(Animation
animation) {
231                 super.onAnimationEnd(animation);
232                 if (fragment.getAnimatingAway() !=
null) {
233                     fragment.setAnimatingAway(null);
234                     moveToState(fragment,
fragment.getStateAfterAnimating(),
235                                     0, 0, false);
236                 }
237             }
238         });
239         f.mView.startAnimation(anim);
240     }
241     f.mContainer.removeView(f.mView);
242 }
243 f.mContainer = null;
244 f.mView = null;
245 f.mInnerView = null;
246 }
247 case Fragment.CREATED:
248     if (newState < Fragment.CREATED) {
249         if (mDestroyed) {
250             if (f.getAnimatingAway() != null) {
251                 // The fragment's containing activity is
252                 // being destroyed, but this fragment is
253                 // currently animating away. Stop the
254                 // animation right now -- it is not needed,
255                 // and we can't wait any more on destroying
256                 // the fragment.
257                 View v = f.getAnimatingAway();
258                 f.setAnimatingAway(null);
259                 v.clearAnimation();
260             }
261         }
262         if (f.getAnimatingAway() != null) {
263             // We are waiting for the fragment's view to
finish
264             // animating away. Just make a note of the state
265             // the fragment now should move to once the
animation
266             // is done.
267             f.setStateAfterAnimating(newState);
268             newState = Fragment.CREATED;
269         } else {
270             if (DEBUG) Log.v(TAG, "movefrom CREATED: " + f);
271             if (!f.mRetaining) {
272                 f.performDestroy();
273                 dispatchOnFragmentDestroyed(f, false);
274             } else {
275                 f.mState = Fragment.INITIALIZING;

```

```

276         }
277
278         f.performDetach();
279         dispatchOnFragmentDetached(f, false);
280         if (!keepActive) {
281             if (!f.mRetaining) {
282                 makeInactive(f);
283             } else {
284                 f.mHost = null;
285                 f.mParentFragment = null;
286                 f.mFragmentManager = null;
287             }
288         }
289     }
290 }
291 }
292 }
293
294 if (f.mState != newState) {
295     Log.w(TAG, "moveToState: Fragment state for " + f + " not updated
inline; "
296         + "expected state " + newState + " found " + f.mState);
297     f.mState = newState;
298 }
299 }

```

代码很长，但做的事情很简单：

1. 根据状态调用对应的生命周期方法
2. 如果是新创建的，就把布局添加到 ViewGroup 中

Fragment 是什么

Fragment 是什么，从官网、别人博客上看到的都是他人之言，我们还是得去看源码才能得到答案。

```

1 public class Fragment implements ComponentCallbacks,
  onCreateContextMenuListener {...}

```

可以看到，Fragment 没有继承任何类，只是实现了这两个接口，第二个不太重要，第一个是在内存不足时可以收到回调。

没有什么特别信息，我们还是去看看它的主要成员。

Fragment 的主要成员

```

1 static final int INITIALIZING = 0;    // Not yet created.
2 static final int CREATED = 1;        // Created.
3 static final int ACTIVITY_CREATED = 2; // The activity has finished its
creation.
4 static final int STOPPED = 3;         // Fully created, not started.
5 static final int STARTED = 4;         // Created and started, not resumed.
6 static final int RESUMED = 5;         // Created started and resumed.
7
8 //当前 Fragment 的状态值
9 int mState = INITIALIZING;

```



```

10 //...
11 // True if the fragment is in the list of added fragments.
12 boolean mAdded;
13
14 // If set this fragment is being removed from its activity.
15 boolean mRemoving;
16
17 // Set to true if this fragment was instantiated from a layout file.
18 boolean mFromLayout;
19
20 // Set to true when the view has actually been inflated in its layout.
21 boolean mInLayout;
22
23 // True if this fragment has been restored from previously saved state.
24 boolean mRestored;
25
26 // Number of active back stack entries this fragment is in.
27 int mBackStackNesting;
28
29 // Set to true when the app has requested that this fragment be hidden
30 // from the user.
31 boolean mHidden;
32
33 // Set to true when the app has requested that this fragment be
34 // deactivated.
35 boolean mDetached;
36
37 // If set this fragment would like its instance retained across
38 // configuration changes.
39 boolean mRetainInstance;
40
41 // If set this fragment is being retained across the current config change.
42 boolean mRetaining;
43
44 // If set this fragment has menu items to contribute.
45 boolean mHasMenu;
46
47 // Set to true to allow the fragment's menu to be shown.
48 boolean mMenuVisible = true;
49
50 // Used to verify that subclasses call through to super class.
51 boolean mCalled;

```

一堆标志位和状态值。然后就是关键的成员了：

```

1 // The fragment manager we are associated with. Set as soon as the
2 // fragment is used in a transaction; cleared after it has been removed
3 // from all transactions.
4 FragmentManagerImpl mFragmentManager;
5
6 //FragmentManager 绑定的对象，一半就是 Activity 和 Fragment
7 FragmentHostCallback mHost;
8 //管理子 Fragment
9 FragmentManagerImpl mChildFragmentManager;
10
11 // For use when restoring fragment state and descendant fragments are
12 // retained.

```

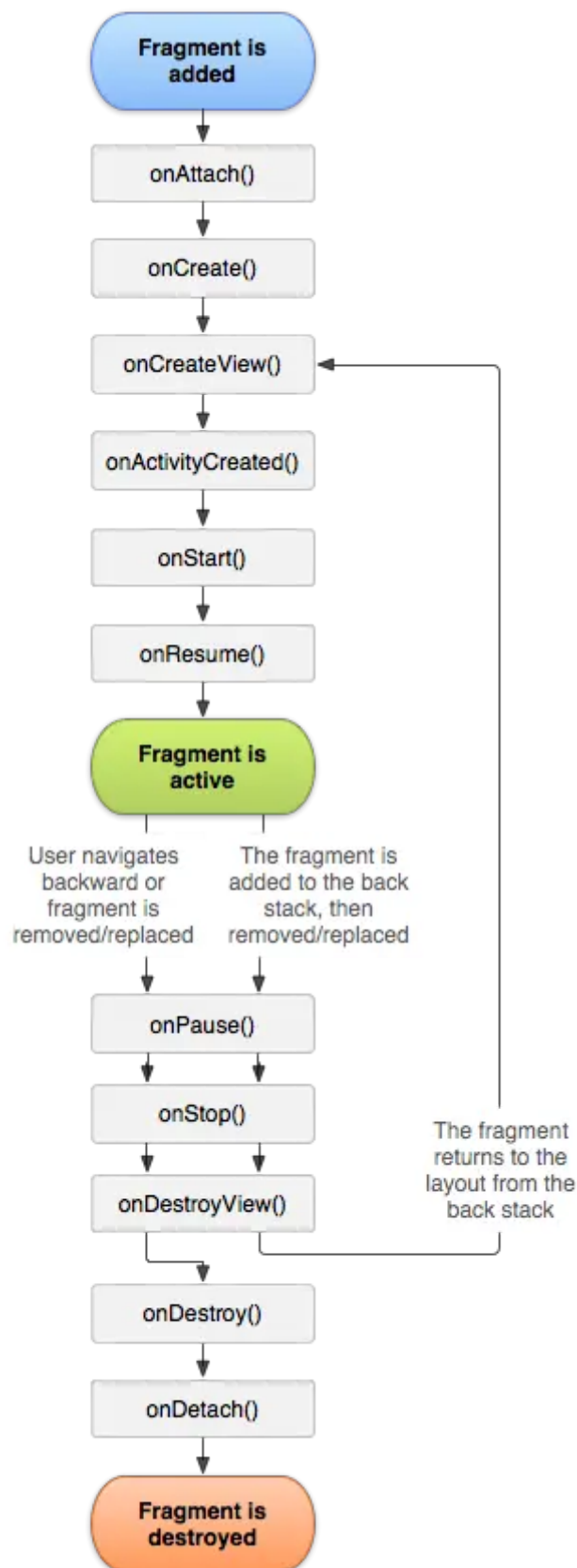
```
12 // This state is set by FragmentState.instantiate and cleared in onCreate.
13 FragmentManagerNonConfig mChildNonConfig;
14 //如果这个 Fragment 绑定的是另一个 Fragment，就需要设置这个值
15 Fragment mParentFragment;
16 //容器 Fragment 的ID
17 int mFragmentId;
18 //容器 View 的ID
19 int mContainerId;
20
21 //父布局
22 ViewGroup mContainer;
23
24 //当前 Fragment 的布局
25 View mView;
26
27 //真正保存状态的内部布局
28 View mInnerview;
```

看到这里，结合前面的，我们就清晰了一个 Fragment 的创建、添加过程：

在 `onCreateView()` 中返回一个 布局，然后在 `FragmentManager` 中拿到这个布局，添加到要绑定容器（`Activity/Fragment`）的 `ViewGroup` 中，然后设置相应的状态值。

生命周期方法

Fragment 的生命周期大家都清楚，官方提供了一张很清晰的图：



总共 11 个方法，这里我们看一下各个方法的具体源码。

1. `onAttach(Context)`

```

1  @CallSuper
2  public void onAttach(Context context) {
3      mCalled = true;
4      final Activity hostActivity = mHost == null ? null :
mHost.getActivity();
5      if (hostActivity != null) {

```

```

6         mCalled = false;
7         onAttach(hostActivity);
8     }
9 }
10
11 @Deprecated
12 @CallSuper
13 public void onAttach(Activity activity) {
14     mCalled = true;
15 }

```

`onAttach()` 是一个 `Fragment` 和它的 `Context` 关联时第一个调用的方法，这里我们可以获得对应的 `Context` 或者 `Activity`，可以看到这里拿到的 `Activity` 是 `mHost.getActivity()`，后面我们介绍 `FragmentManager` 时介绍这个方法。

2. onCreate(Bundle)

```

1 public void onCreate(@Nullable Bundle savedInstanceState) {
2     mCalled = true;
3     restoreChildFragmentState(savedInstanceState);
4     if (mChildFragmentManager != null
5         && !mChildFragmentManager.isStateAtLeast(Fragment.CREATED)) {
6         mChildFragmentManager.dispatchCreate();
7     }
8 }
9 void restoreChildFragmentState(@Nullable Bundle savedInstanceState) {
10     if (savedInstanceState != null) {
11         Parcelable p = savedInstanceState.getParcelable(
12             FragmentActivity.FRAGMENTS_TAG);
13         if (p != null) {
14             if (mChildFragmentManager == null) {
15                 instantiateChildFragmentManager();
16             }
17             mChildFragmentManager.restoreAllState(p, mChildNonConfig);
18             mChildNonConfig = null;
19             mChildFragmentManager.dispatchCreate();
20         }
21     }
22 }

```

`onCreate()` 在 `onAttach()` 后调用，用于做一些初始化操作。

需要注意的是，`Fragment` 的 `onCreate()` 调用时关联的 `Activity` 可能还没创建好，所以这里不要有依赖外部 `Activity` 布局的操作。如果有依赖 `Activity` 的操作，可以放在 `onActivityCreated()` 中。

从上面的代码还可以看到，如果是从旧状态中恢复，会执行子 `Fragment` 状态的恢复，此外还在 `onCreate()` 中调用了子 `Fragment` 管理者的创建。

3. onCreateView(LayoutInflater, ViewGroup, Bundle)

```

1 @Nullable
2 public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup
3     container,
4     @Nullable Bundle savedInstanceState) {
5     return null;
6 }

```

在 `onCreate()` 后就会执行 `onCreatView()`，这个方法返回一个 `View`，默认返回为 `null`。

当我们需要在 `Fragment` 中显示布局时，需要重写这个方法，返回要显示的布局。

后面布局销毁时就会调用 `onDestroyView()`。

3.1. onViewCreated

```
1 public void onViewCreated(View view, @Nullable Bundle savedInstanceState) {  
2 }
```

`onViewCreate()` 不是生命周期中的方法，但是却很有用。

它会在 `onCreateView()` 返回后立即执行，参数中的 `view` 就是之前创建的 `View`，因此我们可以在 `onViewCreate()` 中进行布局的初始化，比如这样：

```
1 @Override  
2 public void onViewCreated(final View view, @Nullable final Bundle  
   savedInstanceState) {  
3     if (view == null) {  
4         return;  
5     }  
6     mTextView = (TextView) view.findViewById(R.id.tv_content);  
7     mBtnSwitchChild = (Button) view.findViewById(R.id.btn_switch_child);  
8  
9     Bundle arguments = getArguments();  
10    if (arguments != null && mTextView != null &&  
       !TextUtils.isEmpty(arguments.getString(KEY_TITLE))) {  
11        mTextView.setText(arguments.getString(KEY_TITLE));  
12    }  
13    mBtnSwitchChild.setOnClickListener(new View.OnClickListener() {  
14        @Override  
15        public void onClick(final View v) {  
16            //...  
17        }  
18    });  
19 }
```

4. onActivityCreated(Bundle)

```
1 @CallSuper  
2 public void onActivityCreated(@Nullable Bundle savedInstanceState) {  
3     mCalled = true;  
4 }
```

`onActivityCreated()` 会在 `Fragment` 关联的 `Activity` 创建好、`Fragment` 的布局结构初始化完成后调用。

可以在这个方法里做些和布局、状态恢复有关的操作。

4.1 onViewStateRestored(Bundle)

```
1 @CallSuper  
2 public void onViewStateRestored(@Nullable Bundle savedInstanceState) {  
3     mCalled = true;  
4 }
```

`onViewStateRestored()` 方法会在 `onActivityCreated()` 结束后调用，用于一个 Fragment 在从旧的状态恢复时，获取状态 `saveInstanceState` 恢复状态，比如恢复一个 check box 的状态。

经过这四步，Fragment 创建完成，同步于 Activity 的创建过程。

5. onStart()

```
1  @CallSuper
2  public void onStart() {
3      mCalled = true;
4
5      if (!mLoadersStarted) {
6          mLoadersStarted = true;
7          if (!mCheckedForLoaderManager) {
8              mCheckedForLoaderManager = true;
9              mLoaderManager = mHost.getLoaderManager(mwho, mLoadersStarted,
false);
10         }
11         if (mLoaderManager != null) {
12             mLoaderManager.doStart();
13         }
14     }
15 }
```

`onStart()` 当 Fragment 可见时调用，同步于 Activity 的 `onStart()`。

6. onResume()

```
1  @CallSuper
2  public void onResume() {
3      mCalled = true;
4  }
```

`onResume()` 当 Fragment 可见并且可以与用户交互时调用。

它和 Activity 的 `onResume()` 同步。

7. onPause()

```
1  @CallSuper
2  public void onPause() {
3      mCalled = true;
4  }
```

`onPause()` 当 Fragment 不再可见时调用。

也和 Activity 的 `onPause()` 同步。

8. onStop()

```
1  @CallSuper
2  public void onStop() {
3      mCalled = true;
4  }
```

`onStop()` 当 Fragment 不再启动时调用，和 `Activity.onStop()` 一致。

9. onDestroyView()

```
1 @CallSuper
2 public void onDestroyView() {
3     mCalled = true;
4 }
```

当 `onCreateView()` 返回的布局（不论是不是 null）从 Fragment 中解除绑定时调用 `onDestroyView()`。

下次 Fragment 展示时，会重新创建布局。

10. onDestroy()

```
1 @CallSuper
2 public void onDestroy() {
3     mCalled = true;
4     //Log.v("foo", "onDestroy: mCheckedForLoaderManager=" +
mCheckedForLoaderManager
5     //      + " mLoaderManager=" + mLoaderManager);
6     if (!mCheckedForLoaderManager) {
7         mCheckedForLoaderManager = true;
8         mLoaderManager = mHost.getLoaderManager(mWho, mLoadersStarted,
false);
9     }
10    if (mLoaderManager != null) {
11        mLoaderManager.doDestroy();
12    }
13 }
```

当 Fragment 不再使用时会调用 `onDestroy()`，它是一个 Fragment 生命周期的倒数第二步。

可以看到这里，调用了 `mLoaderManager.doDestroy()`，后面介绍它。

11. onDetach()

```
1 @CallSuper
2 public void onDetach() {
3     mCalled = true;
4 }
```

Fragment 生命周期的最后一个方法，当 Fragment 不再和一个 Activity 绑定时调用。

Fragment 的 `onDestroyView()`，`onDestroy()`，`onDetach()` 三个对应 Activity 的 `onDestroyed()` 方法。

总结

OK，看完这篇文章，相信对开头提出的问题你已经有了答案，这里再总结一下。

Fragment、FragmentManager、FragmentTransaction 关系

- Fragment

- 其实是对 View 的封装，它持有 view, containerView, fragmentManager, childFragmentManager 等信息
- fragmentManager
 - 是一个抽象类，它定义了对一个 Activity/Fragment 中 **添加进来的 Fragment 列表、Fragment 回退栈**的操作、管理方法
 - 还定义了获取事务对象的方法
 - 具体实现在 fragmentManager 中
- fragmentManager
 - 定义了对 Fragment 添加、替换、隐藏等操作，还有四种提交方法
 - 具体实现是在 BackStackRecord 中

Fragment 如何实现布局的添加替换

通过获得当前 Activity/Fragment 的 fragmentManager/ChildFragmentManager，进而拿到事务的实现类 BackStackRecord，它将目标 Fragment 构造成 Ops（包装 Fragment 和状态信息），然后提交给 fragmentManager 处理。

如果是异步提交，就通过 Handler 发送 Runnable 任务，FragmentManager 拿到任务后，先处理 Ops 状态，然后调用 `moveToState()` 方法根据状态调用 Fragment 对应的生命周期方法，从而达到 Fragment 的添加、布局的替换隐藏等。

下面这张图从下往上看就是一个 Fragment 创建经历的方法：

```
at android.support.v4.app.Fragment.performCreateView(Fragment.java:2192)
at android.support.v4.app.FragmentManagerImpl.moveToState(FragmentManager.java:1299)
at android.support.v4.app.FragmentManagerImpl.moveFragmentToExpectedState(FragmentManager.java:1528)
at android.support.v4.app.FragmentManagerImpl.moveToState(FragmentManager.java:1595)
at android.support.v4.app.BackStackRecord.executeOps(BackStackRecord.java:758)
at android.support.v4.app.FragmentManagerImpl.executeOps(FragmentManager.java:2363)
at android.support.v4.app.FragmentManagerImpl.executeOpsTogether(FragmentManager.java:2149)
at android.support.v4.app.FragmentManagerImpl.optimizeAndExecuteOps(FragmentManager.java:2103)
at android.support.v4.app.FragmentManagerImpl.execSingleAction(FragmentManager.java:1984)
at android.support.v4.app.BackStackRecord.commitNowAllowingStateLoss(BackStackRecord.java:626)
```

嵌套 Fragment 的原理

也比较简单，Fragment 内部有一个 childFragmentManager，通过它管理子 Fragment。

在添加子 Fragment 时，把子 Fragment 的布局 add 到父 Fragment 即可