Fragment 的使用

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

```
mFragmentManager = getSupportFragmentManager();
mFragmentManager.beginTransaction()
    .replace(R.id.fl_content, fragment)
    .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

```
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：

```
//开启一系列对 Fragments 的操作
public abstract FragmentTransaction beginTransaction();

//FragmentTransaction.commit() 是异步执行的，如果你想立即执行，可以调用这个方法
public abstract boolean executePendingTransactions();

//根据 ID 找到从 XML 解析出来的或者事务中添加的 Fragment
//首先会找添加到 FragmentManager 中的，找不到就去回退栈里找
public abstract Fragment findFragmentById(@IdRes int id);

//跟上面的类似，不同的是使用 tag 进行查找
public abstract Fragment findFragmentByTag(String tag);

//弹出回退栈中栈顶的 Fragment，异步执行的
public abstract void popBackStack();

//立即弹出回退栈中栈顶的，直接执行哦
public abstract boolean popBackStackImmediate();

```

```
//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
//有点类似启动模式的 singleTask 的感觉
//如果传入的 name 为 null，那就和 popBackStack() 一样了
//异步执行
public abstract void popBackStack(String name, int flags);

//同步版的上面
public abstract boolean popBackStackImmediate(String name, int flags);

//和使用 name 查找、弹出一样
//不同的是这里的 id 是 FragmentTransaction.commit() 返回的 id
public abstract void popBackStack(int id, int flags);
//你懂得
public abstract boolean popBackStackImmediate(int id, int flags);

//获取回退栈中的元素个数
public abstract int getBackStackEntryCount();
//根据索引获取回退栈中的某个元素
public abstract BackStackEntry getBackStackEntryAt(int index);

//添加或者移除一个监听器
public abstract void addOnBackStackChangedListener(OnBackStackChangedListener listener);
public abstract void removeOnBackStackChangedListener(OnBackStackChangedListener listener);

//还定义了将一个 Fragment 实例作为参数传递
public abstract void putFragment(Bundle bundle, String key, Fragment fragment);
public abstract Fragment getFragment(Bundle bundle, String key);

//获取 manager 中所有添加进来的 Fragment
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
- onBackStackChangedListener：后退栈变动监听器
- FragmentLifecycleCallbacks: FragmentManager 中的 Fragment 生命周期监听

```
//后退栈中的一个元素
public interface BackStackEntry {
    //栈中该元素的唯一标识
    public int getId();

    //获取 FragmentTransaction#addToBackStack(String) 设置的名称
    public String getName();

    @StringRes
    public int getBreadCrumbTitleRes();
    @StringRes
    public int getBreadCrumbShortTitleRes();
    public CharSequence getBreadCrumbTitle();
```

```
    public CharSequence getBreadCrumbShortTitle();
}
```

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

```
//在 Fragment 回退栈中有变化时回调
public interface OnBackStackChangedListener {
    public void onBackStackChanged();
}
//FragmentManager 中的 Fragment 生命周期监听
    public abstract static class FragmentLifecycleCallbacks {
        public void onFragmentPreAttached(FragmentManager fm, Fragment f,
Context context) {}
        public void onFragmentAttached(FragmentManager fm, Fragment f,
Context context) {}
        public void onFragmentCreated(FragmentManager fm, Fragment f,
Bundle savedInstanceState) {}
        public void onFragmentActivityCreated(FragmentManager fm, Fragment
f,
                Bundle savedInstanceState) {}
        public void onFragmentViewCreated(FragmentManager fm, Fragment f,
View v,
                Bundle savedInstanceState) {}
        public void onFragmentStarted(FragmentManager fm, Fragment f) {}
        public void onFragmentResumed(FragmentManager fm, Fragment f) {}
        public void onFragmentPaused(FragmentManager fm, Fragment f) {}
        public void onFragmentStopped(FragmentManager fm, Fragment f) {}
        public void onFragmentSaveInstanceState(FragmentManager fm,
Fragment f, Bundle outState) {}
        public void onFragmentViewDestroyed(FragmentManager fm, Fragment f)
{}
        public void onFragmentDestroyed(FragmentManager fm, Fragment f) {}
        public void onFragmentDetached(FragmentManager fm, Fragment f) {}
    }
}
```

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：

```
final class FragmentManagerImpl extends FragmentManager implements
LayoutInflaterFactory {

    ArrayList<OpGenerator> mPendingActions;
    Runnable[] mTmpActions;
    boolean mExecutingActions;
```

```
    ArrayList<Fragment> mActive;
    ArrayList<Fragment> mAdded;
    ArrayList<Integer> mAvailIndices;
    ArrayList<BackStackRecord> mBackStack;
    ArrayList<Fragment> mCreatedMenus;

// Must be accessed while locked.
    ArrayList<BackStackRecord> mBackStackIndices;
    ArrayList<Integer> mAvailBackStackIndices;

    ArrayList<OnBackStackChangedListener> mBackStackChangeListeners;
    private CopyOnWriteArrayList<Pair<FragmentLifecycleCallbacks, Boolean>>
mLifecycleCallbacks;
//...
}
```

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。

```
int mCurState = Fragment.INITIALIZING;
FragmentHostCallback mHost;
FragmentContainer mContainer;
Fragment mParent;

static Field sAnimationListenerField = null;

boolean mNeedMenuInvalidate;
boolean mStateSaved;
boolean mDestroyed;
String mNoTransactionsBecause;
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：

```
public abstract class FragmentContainer {
    @Nullable
    public abstract View onFindViewById(@IdRes int id);
    public abstract boolean onHasView();
}
```

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：

```
public abstract class FragmentHostCallback<E> extends FragmentContainer {
    private final Activity mActivity;
    final Context mContext;
    private final Handler mHandler;
    final int mWindowAnimations;
    final FragmentManagerImpl mFragmentManager = new FragmentManagerImpl();
    //...
}
```

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：

```
class HostCallbacks extends FragmentHostCallback<FragmentActivity> {
    public HostCallbacks() {
        super(FragmentActivity.this /*fragmentActivity*/);
    }
    //...

    @Override
    public LayoutInflater onGetLayoutInflater() {
        return FragmentActivity.this.getLayoutInflater().cloneInContext(FragmentActivity.this);
    }

    @Override
    public FragmentActivity onGetHost() {
        return FragmentActivity.this;
    }

    @Override
    public void onStartActivityFromFragment(Fragment fragment, Intent intent, int requestCode) {
        FragmentActivity.this.startActivityFromFragment(fragment, intent, requestCode);
    }

    @Override
    public void onStartActivityFromFragment(
            Fragment fragment, Intent intent, int requestCode, @Nullable Bundle options) {
        FragmentActivity.this.startActivityFromFragment(fragment, intent, requestCode, options);
    }

    @Override
    public void onRequestPermissionsFromFragment(@NonNull Fragment fragment,
            @NonNull String[] permissions, int requestCode) {
        FragmentActivity.this.requestPermissionsFromFragment(fragment, permissions,
                requestCode);
    }

    @Override
    public boolean onShouldShowRequestPermissionRationale(@NonNull String permission) {
        return ActivityCompat.shouldShowRequestPermissionRationale(
                FragmentActivity.this, permission);
    }

    @Override
    public boolean onHasWindowAnimations() {
```

```
        return getWindow() != null;
    }

    @Override
    public void onAttachFragment(Fragment fragment) {
        FragmentActivity.this.onAttachFragment(fragment);
    }

    @Nullable
    @Override
    public View onFindViewById(int id) {
        return FragmentActivity.this.findViewById(id);
    }

    @Override
    public boolean onHasView() {
        final Window w = getWindow();
        return (w != null && w.peekDecorView() != null);
    }
}
```

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

```
@Override
public FragmentTransaction beginTransaction() {
    return new BackStackRecord(this);
}
```

`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?

```
@Override
public void popBackStack() {
    enqueueAction(new PopBackStackState(null, -1, 0), false);
}
public void enqueueAction(OpGenerator action, boolean allowStateLoss) {
    if (!allowStateLoss) {
        checkStateLoss();
    }
    synchronized (this) {
        if (mDestroyed || mHost == null) {
            throw new IllegalStateException("Activity has been destroyed");
        }
        if (mPendingActions == null) {
            mPendingActions = new ArrayList<>();
        }
        mPendingActions.add(action);
        scheduleCommit();
    }
}
private void scheduleCommit() {
    synchronized (this) {
        boolean postponeReady =
                mPostponedTransactions != null &&
!mPostponedTransactions.isEmpty();
```

```
        boolean pendingReady = mPendingActions != null &&
mPendingActions.size() == 1;
        if (postponeReady || pendingReady) {
            mHost.getHandler().removeCallbacks(mExecCommit);
            mHost.getHandler().post(mExecCommit);
        }
    }
}
```

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：

```
ArrayList<BackStackRecord> mBackStack;
@Override
public int getBackStackEntryCount() {
    return mBackStack != null ? mBackStack.size() : 0;
}

@Override
public BackStackEntry getBackStackEntryAt(int index) {
    return mBackStack.get(index);
}
```

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`：

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

先来看看 `FragmentTransaction`。

FragmentTransaction

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

```
//它会调用 add(int, Fragment, String)，其中第一个参数传的是 0
public abstract FragmentTransaction add(Fragment fragment, String tag);

//它会调用 add(int, Fragment, String)，其中第三个参数是 null
public abstract FragmentTransaction add(@IdRes int containerViewId,
Fragment fragment);

//添加一个 Fragment 给 Activity 的最终实现
//第一个参数表示 Fragment 要放置的布局 id
//第二个参数表示要添加的 Fragment，【注意】一个 Fragment 只能添加一次
//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
public abstract FragmentTransaction add(@IdRes int containerViewId,
Fragment fragment,
        @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,
    Fragment fragment);
16
17  //替换宿主中一个已经存在的 fragment
18  //这一个方法等价于先调用 remove()，再调用 add()
19  public abstract FragmentTransaction replace(@IdRes int containerViewId,
    Fragment fragment,
20          @Nullable String tag);
21
22  //移除一个已经存在的 fragment
23  //如果之前添加到宿主上，那它的布局也会被移除
24  public abstract FragmentTransaction remove(Fragment fragment);
25
26  //隐藏一个已存的 fragment
27  //其实就是将添加到宿主上的布局隐藏
28  public abstract FragmentTransaction hide(Fragment fragment);
29
30  //显示前面隐藏的 fragment，这只适用于之前添加到宿主上的 fragment
31  public abstract FragmentTransaction show(Fragment fragment);
32
33  //将指定的 fragment 将布局上解除
34  //当调用这个方法时，fragment 的布局已经销毁了
35  public abstract FragmentTransaction detach(Fragment fragment);
36
37  //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
38  //这将导致该 fragment 的布局重建，然后添加、展示到界面上
39  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 FragmentManger 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;
```

```
static final int OP_REMOVE = 3;
static final int OP_HIDE = 4;
static final int OP_SHOW = 5;
static final int OP_DETACH = 6;
static final int OP_ATTACH = 7;

ArrayList<Op> mOps = new ArrayList<>();
static final class Op {
    int cmd;    //状态
    Fragment fragment;
    int enterAnim;
    int exitAnim;
    int popEnterAnim;
    int popExitAnim;
}

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**：

```
@Override
public FragmentTransaction add(int containerViewId, Fragment fragment) {
    doAddOp(containerViewId, fragment, null, OP_ADD);
    return this;
}

@Override
public FragmentTransaction add(int containerViewId, Fragment fragment,
String tag) {
    doAddOp(containerViewId, fragment, tag, OP_ADD);
    return this;
}
private void doAddOp(int containerViewId, Fragment fragment, String tag,
int opcmd) {
    final Class fragmentClass = fragment.getClass();
    final int modifiers = fragmentClass.getModifiers();
    if (fragmentClass.isAnonymousClass() || !Modifier.isPublic(modifiers)
            || (fragmentClass.isMemberClass() &&
!Modifier.isStatic(modifiers))) {
        throw new IllegalStateException("Fragment " +
fragmentClass.getCanonicalName()
                + " must be a public static class to be  properly recreated
from"
                + " instance state.");
    }

    //1.修改添加的 fragmentManager 为当前栈的 manager
    fragment.mFragmentManager = mManager;

    if (tag != null) {
        if (fragment.mTag != null && !tag.equals(fragment.mTag)) {
            throw new IllegalStateException("Can't change tag of fragment "
                    + fragment + ": was " + fragment.mTag
```

```
                    + " now " + tag);
        }
        fragment.mTag = tag;
    }

    if (containerViewId != 0) {
        if (containerViewId == View.NO_ID) {
            throw new IllegalArgumentException("Can't add fragment "
                    + fragment + " with tag " + tag + " to container view
with no id");
        }
        if (fragment.mFragmentId != 0 && fragment.mFragmentId !=
containerViewId) {
            throw new IllegalStateException("Can't change container ID of
fragment "
                    + fragment + ": was " + fragment.mFragmentId
                    + " now " + containerViewId);
        }
        //2.设置宿主 ID 为布局 ID
        fragment.mContainerId = fragment.mFragmentId = containerViewId;
    }

    //3.构造 Op
    Op op = new Op();
    op.cmd = opcmd;    //状态
    op.fragment = fragment;
    //4.添加到数组列表中
    addOp(op);
}
void addOp(Op op) {
    mOps.add(op);
    op.enterAnim = mEnterAnim;
    op.exitAnim = mExitAnim;
    op.popEnterAnim = mPopEnterAnim;
    op.popExitAnim = mPopExitAnim;
}
```

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**：

```
@Override
public FragmentTransaction replace(int containerViewId, Fragment fragment)
{
    return replace(containerViewId, fragment, null);
}

@Override
public FragmentTransaction replace(int containerViewId, Fragment fragment,
String tag) {
    if (containerViewId == 0) {
        throw new IllegalArgumentException("Must use non-zero
containerViewId");
    }

    doAddOp(containerViewId, fragment, tag, OP_REPLACE);
```

```
    return this;
}
```

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。

```
@Override
public FragmentTransaction remove(Fragment fragment) {
    Op op = new Op();
    op.cmd = OP_REMOVE;
    op.fragment = fragment;
    addOp(op);

    return this;
}

@Override
public FragmentTransaction hide(Fragment fragment) {
    Op op = new Op();
    op.cmd = OP_HIDE;
    op.fragment = fragment;
    addOp(op);

    return this;
}

@Override
public FragmentTransaction show(Fragment fragment) {
    Op op = new Op();
    op.cmd = OP_SHOW;
    op.fragment = fragment;
    addOp(op);

    return this;
}
```

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：

```
@Override
public int commit() {
    return commitInternal(false);
}

@Override
public int commitAllowingStateLoss() {
    return commitInternal(true);
}
int commitInternal(boolean allowStateLoss) {
    if (mCommitted) throw new IllegalStateException("commit already
called");
    //...
```

```
    mCommitted = true;
    if (mAddToBackStack) {
        mIndex = mManager.allocBackStackIndex(this);    //更新 index 信息
    } else {
        mIndex = -1;
    }
    mManager.enqueueAction(this, allowStateLoss);    //异步任务入队
    return mIndex;
}
public void enqueueAction(OpGenerator action, boolean allowStateLoss) {
    if (!allowStateLoss) {
        checkStateLoss();
    }
    synchronized (this) {
        if (mDestroyed || mHost == null) {
            throw new IllegalStateException("Activity has been destroyed");
        }
        if (mPendingActions == null) {
            mPendingActions = new ArrayList<>();
        }
        mPendingActions.add(action);
        scheduleCommit();    //发送任务
    }
}
private void scheduleCommit() {
    synchronized (this) {
        boolean postponeReady =
                mPostponedTransactions != null &&
!mPostponedTransactions.isEmpty();
        boolean pendingReady = mPendingActions != null &&
mPendingActions.size() == 1;
        if (postponeReady || pendingReady) {
            mHost.getHandler().removeCallbacks(mExecCommit);
            mHost.getHandler().post(mExecCommit);
        }
    }
}
```

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`:

```
Runnable mExecCommit = new Runnable() {
    @Override
    public void run() {
        execPendingActions();
    }
};
/**
 * Only call from main thread!
 * 更新 UI 嘛，肯定得在主线程
 */
public boolean execPendingActions() {
    ensureExecReady(true);

```

```
    boolean didSomething = false;
    while (generateOpsForPendingActions(mTmpRecords, mTmpIsPop)) {
        mExecutingActions = true;
        try {
            optimizeAndExecuteOps(mTmpRecords, mTmpIsPop);    //这里是入口
        } finally {
            cleanupExec();
        }
        didSomething = true;
    }

    doPendingDeferredStart();

    return didSomething;
}
private void optimizeAndExecuteOps(ArrayList<BackStackRecord> records,
        ArrayList<Boolean> isRecordPop) {
    if (records == null || records.isEmpty()) {
        return;
    }

    if (isRecordPop == null || records.size() != isRecordPop.size()) {
        throw new IllegalStateException("Internal error with the back
stack records");
    }

    // Force start of any postponed transactions that interact with
scheduled transactions:
    executePostponedTransaction(records, isRecordPop);

    final int numRecords = records.size();
    int startIndex = 0;
    for (int recordNum = 0; recordNum < numRecords; recordNum++) {
        final boolean canOptimize =
records.get(recordNum).mAllowOptimization;
        if (!canOptimize) {
            // execute all previous transactions
            if (startIndex != recordNum) {
                //这里将 Ops 过滤一遍
                executeOpsTogether(records, isRecordPop, startIndex,
recordNum);
            }
            // execute all unoptimized pop operations together or one add
operation
              //...
    }
    if (startIndex != numRecords) {
        executeOpsTogether(records, isRecordPop, startIndex, numRecords);
    }
}
private void executeOpsTogether(ArrayList<BackStackRecord> records,
        ArrayList<Boolean> isRecordPop, int startIndex, int endIndex) {
    final boolean allowOptimization =
records.get(startIndex).mAllowOptimization;
    boolean addToBackStack = false;
    if (mTmpAddedFragments == null) {
        mTmpAddedFragments = new ArrayList<>();
    } else {
```

```
        mTmpAddedFragments.clear();
    }
    if (mAdded != null) {
        mTmpAddedFragments.addAll(mAdded);
    }
    for (int recordNum = startIndex; recordNum < endIndex; recordNum++) {
        final BackStackRecord record = records.get(recordNum);
        final boolean isPop = isRecordPop.get(recordNum);
        if (!isPop) {
            record.expandReplaceOps(mTmpAddedFragments);    //修改状态
        } else {
            record.trackAddedFragmentsInPop(mTmpAddedFragments);
        }
        addToBackStack = addToBackStack || record.mAddToBackStack;
    }
    mTmpAddedFragments.clear();

    if (!allowOptimization) {
        FragmentTransition.startTransitions(this, records, isRecordPop,
startIndex, endIndex,
                false);
    }
    //真正处理的入口
    executeOps(records, isRecordPop, startIndex, endIndex);

    int postponeIndex = endIndex;
    if (allowOptimization) {
        ArraySet<Fragment> addedFragments = new ArraySet<>();
        addAddedFragments(addedFragments);
        postponeIndex = postponePostponableTransactions(records,
isRecordPop,
                startIndex, endIndex, addedFragments);
        makeRemovedFragmentsInvisible(addedFragments);    //名字就能看出来作
用
    }

    if (postponeIndex != startIndex && allowOptimization) {
        // need to run something now
        FragmentTransition.startTransitions(this, records, isRecordPop,
startIndex,
                postponeIndex, true);
        moveToState(mCurState, true);
    }
    //...
}
//修改 Ops 状态，这一步还没有真正处理状态
expandReplaceOps(ArrayList<Fragment> added) {
    for (int opNum = 0; opNum < mOps.size(); opNum++) {
        final Op op = mOps.get(opNum);
        switch (op.cmd) {
            case OP_ADD:
            case OP_ATTACH:
                added.add(op.fragment);
                break;
            case OP_REMOVE:
            case OP_DETACH:
                added.remove(op.fragment);
                break;
```

```
            case OP_REPLACE: {
                Fragment f = op.fragment;
                int containerId = f.mContainerId;
                boolean alreadyAdded = false;
                for (int i = added.size() - 1; i >= 0; i--) {
                    Fragment old = added.get(i);
                    if (old.mContainerId == containerId) {
                        if (old == f) {
                            alreadyAdded = true;
                        } else {
                            Op removeOp = new Op();
                            removeOp.cmd = OP_REMOVE;    //可以看到，替换也是通过删除实现的
                            removeOp.fragment = old;
                            removeOp.enterAnim = op.enterAnim;
                            removeOp.popEnterAnim = op.popEnterAnim;
                            removeOp.exitAnim = op.exitAnim;
                            removeOp.popExitAnim = op.popExitAnim;
                            mOps.add(opNum, removeOp);
                            added.remove(old);
                            opNum++;
                        }
                    }
                }
                if (alreadyAdded) {
                    mOps.remove(opNum);
                    opNum--;
                } else {
                    op.cmd = OP_ADD;
                    added.add(f);
                }
            }
            break;
        }
    }
}
//设置将要被移除的 Fragment 为不可见的最终实现
private void makeRemovedFragmentsInvisible(ArraySet<Fragment> fragments) {
    final int numAdded = fragments.size();
    for (int i = 0; i < numAdded; i++) {
        final Fragment fragment = fragments.valueAt(i);
        if (!fragment.mAdded) {
            final View view = fragment.getView();    //获取 Fragment 的布局，设置状态
            if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
                fragment.getView().setVisibility(View.INVISIBLE);
            } else {        //高版本设置透明度
                fragment.mPostponedAlpha = view.getAlpha();
                view.setAlpha(0f);
            }
        }
    }
}
```

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：

```
/**
 * Executes the operations contained within this transaction. The Fragment states will only
 * be modified if optimizations are not allowed.
 */
void executeOps() {
    final int numOps = mOps.size();
    for (int opNum = 0; opNum < numOps; opNum++) {
        final Op op = mOps.get(opNum);
        final Fragment f = op.fragment;
        f.setNextTransition(mTransition, mTransitionStyle);
        switch (op.cmd) {
            case OP_ADD:
                f.setNextAnim(op.enterAnim);
                mManager.addFragment(f, false);
                break;
            case OP_REMOVE:
                f.setNextAnim(op.exitAnim);
                mManager.removeFragment(f);
                break;
            case OP_HIDE:
                f.setNextAnim(op.exitAnim);
                mManager.hideFragment(f);
                break;
            case OP_SHOW:
                f.setNextAnim(op.enterAnim);
                mManager.showFragment(f);
                break;
            case OP_DETACH:
                f.setNextAnim(op.exitAnim);
                mManager.detachFragment(f);
                break;
            case OP_ATTACH:
                f.setNextAnim(op.enterAnim);
                mManager.attachFragment(f);
                break;
            default:
                throw new IllegalArgumentException("Unknown cmd: " + op.cmd);
        }
        if (!mAllowOptimization && op.cmd != OP_ADD) {
            mManager.moveFragmentToExpectedState(f);
        }
    }
    if (!mAllowOptimization) {
        // Added fragments are added at the end to comply with prior behavior.
        mManager.moveToState(mManager.mCurState, true);
    }
}
```

FragmentManager 对这些方法的实现也很简单，修改 Fragment 的状态值，比如 `remove(Fragment)`：

```
public void removeFragment(Fragment fragment) {
    if (DEBUG) Log.v(TAG, "remove: " + fragment + " nesting=" +
fragment.mBackStackNesting);
    final boolean inactive = !fragment.isInBackStack();
    if (!fragment.mDetached || inactive) {
        if (mAdded != null) {
            mAdded.remove(fragment);
        }
        if (fragment.mHasMenu && fragment.mMenuVisible) {
            mNeedMenuInvalidate = true;
        }
        fragment.mAdded = false;    //设置属性值
        fragment.mRemoving = true;
    }
}
```

最终会调用 `moveToState()`，我们直接来看它的实现：

```
void moveToState(Fragment f, int newState, int transit, int
transitionStyle,
        boolean keepActive) {
    //还没有添加的 Fragment 处于 onCreate() 状态
    if ((!f.mAdded || f.mDetached) && newState > Fragment.CREATED) {
        newState = Fragment.CREATED;
    }
    if (f.mRemoving && newState > f.mState) {
        // While removing a fragment, we can't change it to a higher
state.
        newState = f.mState;
    }
    //推迟启动的设置为 stop
    if (f.mDeferStart && f.mState < Fragment.STARTED && newState >
Fragment.STOPPED) {
        newState = Fragment.STOPPED;
    }
    if (f.mState < newState) {
        // For fragments that are created from a layout, when restoring
from
        // state we don't want to allow them to be created until they are
        // being reloaded from the layout.
        if (f.mFromLayout && !f.mInLayout) {
            return;
        }
        if (f.getAnimatingAway() != null) {
            // The fragment is currently being animated...  but!  Now we
            // want to move our state back up.  Give up on waiting for the
            // animation, move to whatever the final state should be once
            // the animation is done, and then we can proceed from there.
            f.setAnimatingAway(null);
            //如果当前 Fragment 正有动画，直接修改为最终状态
            moveToState(f, f.getStateAfterAnimating(), 0, 0, true);
        }
        switch (f.mState) {
```

```
            case Fragment.INITIALIZING:
                if (DEBUG) Log.v(TAG, "moveto CREATED: " + f);
                if (f.mSavedFragmentState != null) {

 f.mSavedFragmentState.setClassLoader(mHost.getContext().getClassLoader())
;
                    f.mSavedViewState =
f.mSavedFragmentState.getSparseParcelableArray(
                            FragmentManagerImpl.VIEW_STATE_TAG);
                    f.mTarget = getFragment(f.mSavedFragmentState,
                            FragmentManagerImpl.TARGET_STATE_TAG);
                    if (f.mTarget != null) {
                        f.mTargetRequestCode =
f.mSavedFragmentState.getInt(

 FragmentManagerImpl.TARGET_REQUEST_CODE_STATE_TAG, 0);
                    }
                    f.mUserVisibleHint = f.mSavedFragmentState.getBoolean(
                            FragmentManagerImpl.USER_VISIBLE_HINT_TAG,
true);
                    if (!f.mUserVisibleHint) {
                        f.mDeferStart = true;
                        if (newState > Fragment.STOPPED) {
                            newState = Fragment.STOPPED;
                        }
                    }
                }
                f.mHost = mHost;
                f.mParentFragment = mParent;
                f.mFragmentManager = mParent != null
                        ? mParent.mChildFragmentManager :
mHost.getFragmentManagerImpl();
                dispatchOnFragmentPreAttached(f, mHost.getContext(),
false);
                f.mCalled = false;
                f.onAttach(mHost.getContext());    //调用 Fragment 生命周期
方法
                if (!f.mCalled) {
                    throw new SuperNotCalledException("Fragment " + f
                            + " did not call through to
super.onAttach()");
                }
                if (f.mParentFragment == null) {
                    mHost.onAttachFragment(f);
                } else {
                    f.mParentFragment.onAttachFragment(f);
                }
                dispatchOnFragmentAttached(f, mHost.getContext(), false);

                if (!f.mRetaining) {
                    f.performCreate(f.mSavedFragmentState); //调用 Fragment
生命周期方法

                    dispatchOnFragmentCreated(f, f.mSavedFragmentState,
false);
                } else {
                    f.restoreChildFragmentState(f.mSavedFragmentState);
                    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) + " ("
```

```
                                        + resName
                                        + ") for fragment " + f));
                            }
                        }
                        f.mContainer = container;
                        f.mView = f.performCreateView(f.getLayoutInflater(
//调用 Fragment 生命周期方法
                                f.mSavedFragmentState), container,
f.mSavedFragmentState);
                        if (f.mView != null) {
                            f.mInnerView = f.mView;
                            if (Build.VERSION.SDK_INT >= 11) {

 ViewCompat.setSaveFromParentEnabled(f.mView, false);
                            } else {
                                f.mView =
NoSaveStateFrameLayout.wrap(f.mView);
                            }
                            if (container != null) {
                                container.addView(f.mView);       //将
Fragment 的布局添加到父布局中
                            }
                            if (f.mHidden) {
                                f.mView.setVisibility(View.GONE);
                            }
                            f.onViewCreated(f.mView,
f.mSavedFragmentState);//调用 Fragment 生命周期方法
                            dispatchOnFragmentViewCreated(f, f.mView,
f.mSavedFragmentState,
                                    false);
                            // Only animate the view if it is visible.
This is done after
                            // dispatchOnFragmentViewCreated in case
visibility is changed
                            f.mIsNewlyAdded = (f.mView.getVisibility() ==
View.VISIBLE)
                                    && f.mContainer != null;
                        } else {
                            f.mInnerView = null;
                        }
                    }

                    f.performActivityCreated(f.mSavedFragmentState); //调用
Fragment 生命周期方法

                    dispatchOnFragmentActivityCreated(f,
f.mSavedFragmentState, false);
                    if (f.mView != null) {
                        f.restoreViewState(f.mSavedFragmentState);
                    }
                    f.mSavedFragmentState = null;
                }
            case Fragment.ACTIVITY_CREATED:
                if (newState > Fragment.ACTIVITY_CREATED) {
                    f.mState = Fragment.STOPPED;
                }
            case Fragment.STOPPED:
                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: " +
f);
204                     if (f.mView != null) {
205                         // Need to save the current view state if not
206                         // done already.
207                         if (mHost.onShouldSaveFragmentState(f) &&
f.mSavedViewState == null) {
208                             saveFragmentViewState(f);
209                         }
210                     }
211                     f.performDestroyView();
212                     dispatchOnFragmentViewDestroyed(f, false);
213                     if (f.mView != null && f.mContainer != null) {
214                         Animation anim = null;
215                         if (mCurState > Fragment.INITIALIZING &&
!mDestroyed
216                                 && f.mView.getVisibility() == View.VISIBLE
217                                 && f.mPostponedAlpha >= 0) {
218                             anim = loadAnimation(f, transit, false,
219                                     transitionStyle);
220                         }
221                         f.mPostponedAlpha = 0;
222                         if (anim != null) {
223                             final Fragment fragment = f;
```

```
                            f.setAnimatingAway(f.mView);
                            f.setStateAfterAnimating(newState);
                            final View viewToAnimate = f.mView;
                            anim.setAnimationListener(new
AnimateOnHWLayerIfNeededListener(
                                    viewToAnimate, anim) {
                                @Override
                                public void onAnimationEnd(Animation
animation) {
                                    super.onAnimationEnd(animation);
                                    if (fragment.getAnimatingAway() !=
null) {
                                        fragment.setAnimatingAway(null);
                                        moveToState(fragment,
fragment.getStateAfterAnimating(),
                                                0, 0, false);
                                    }
                                }
                            });
                            f.mView.startAnimation(anim);
                        }
                        f.mContainer.removeView(f.mView);
                    }
                    f.mContainer = null;
                    f.mView = null;
                    f.mInnerView = null;
                }
            case Fragment.CREATED:
                if (newState < Fragment.CREATED) {
                    if (mDestroyed) {
                        if (f.getAnimatingAway() != null) {
                            // The fragment's containing activity is
                            // being destroyed, but this fragment is
                            // currently animating away.  Stop the
                            // animation right now -- it is not needed,
                            // and we can't wait any more on destroying
                            // the fragment.
                            View v = f.getAnimatingAway();
                            f.setAnimatingAway(null);
                            v.clearAnimation();
                        }
                    }
                    if (f.getAnimatingAway() != null) {
                        // We are waiting for the fragment's view to
finish
                        // animating away.  Just make a note of the state
                        // the fragment now should move to once the
animation
                        // is done.
                        f.setStateAfterAnimating(newState);
                        newState = Fragment.CREATED;
                    } else {
                        if (DEBUG) Log.v(TAG, "movefrom CREATED: " + f);
                        if (!f.mRetaining) {
                            f.performDestroy();
                            dispatchOnFragmentDestroyed(f, false);
                        } else {
                            f.mState = Fragment.INITIALIZING;
```

```
                        }

                        f.performDetach();
                        dispatchOnFragmentDetached(f, false);
                        if (!keepActive) {
                            if (!f.mRetaining) {
                                makeInactive(f);
                            } else {
                                f.mHost = null;
                                f.mParentFragment = null;
                                f.mFragmentManager = null;
                            }
                        }
                    }
                }
            }
        }

        if (f.mState != newState) {
            Log.w(TAG, "moveToState: Fragment state for " + f + " not updated inline; "
                    + "expected state " + newState + " found " + f.mState);
            f.mState = newState;
        }
    }
```

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。

```
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

```
static final int INITIALIZING = 0;     // Not yet created.
static final int CREATED = 1;          // Created.
static final int ACTIVITY_CREATED = 2; // The activity has finished its creation.
static final int STOPPED = 3;          // Fully created, not started.
static final int STARTED = 4;          // Created and started, not resumed.
static final int RESUMED = 5;          // Created started and resumed.

//当前 Fragment 的状态值
int mState = INITIALIZING;
```

```
//...
// True if the fragment is in the list of added fragments.
boolean mAdded;

// If set this fragment is being removed from its activity.
boolean mRemoving;

// Set to true if this fragment was instantiated from a layout file.
boolean mFromLayout;

// Set to true when the view has actually been inflated in its layout.
boolean mInLayout;

// True if this fragment has been restored from previously saved state.
boolean mRestored;

// Number of active back stack entries this fragment is in.
int mBackStackNesting;

// Set to true when the app has requested that this fragment be hidden
// from the user.
boolean mHidden;

// Set to true when the app has requested that this fragment be
deactivated.
boolean mDetached;

// If set this fragment would like its instance retained across
// configuration changes.
boolean mRetainInstance;

// If set this fragment is being retained across the current config change.
boolean mRetaining;

// If set this fragment has menu items to contribute.
boolean mHasMenu;

// Set to true to allow the fragment's menu to be shown.
boolean mMenuVisible = true;

// Used to verify that subclasses call through to super class.
boolean mCalled;
```

一堆标志位和状态值。然后就是关键的成员了：

```
// The fragment manager we are associated with.  Set as soon as the
// fragment is used in a transaction; cleared after it has been removed
// from all transactions.
FragmentManagerImpl mFragmentManager;

//Fragmemt 绑定的对象，一半就是 Activity 和 Fragment
FragmentHostCallback mHost;
//管理子 Fragment
FragmentManagerImpl mChildFragmentManager;

// For use when restoring fragment state and descendant fragments are
retained.
```

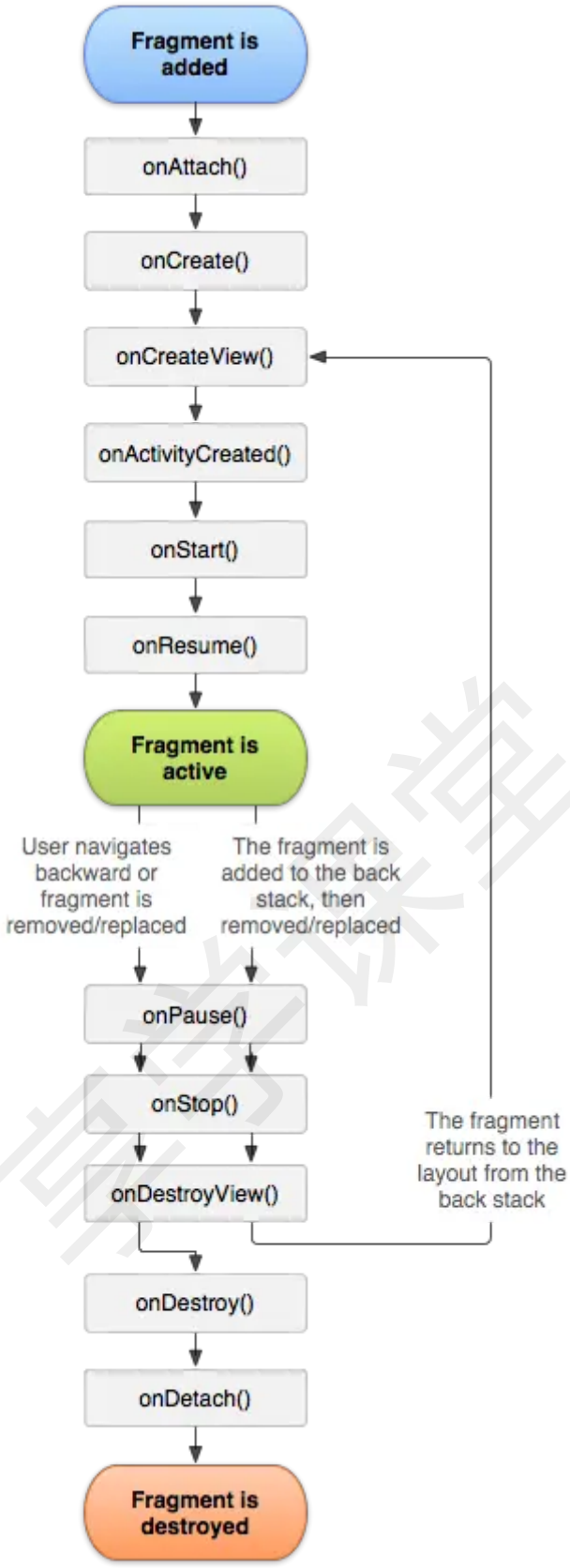
```
// This state is set by FragmentState.instantiate and cleared in onCreate.
FragmentManagerNonConfig mChildNonConfig;
//如果这个 Fragment 绑定的是另一个 Fragment，就需要设置这个值
Fragment mParentFragment;
//容器 Fragment 的ID
int mFragmentId;
//容器 View 的ID
int mContainerId;

//父布局
ViewGroup mContainer;

//当前 Fragment 的布局
View mView;

//真正保存状态的内部布局
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)

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

```
        mCalled = false;
        onAttach(hostActivity);
    }
}

@Deprecated
@CallSuper
public void onAttach(Activity activity) {
    mCalled = true;
}
```

`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)

```
public void onCreate(@Nullable Bundle savedInstanceState) {
    mCalled = true;
    restoreChildFragmentState(savedInstanceState);
    if (mChildFragmentManager != null
            && !mChildFragmentManager.isStateAtLeast(Fragment.CREATED)) {
        mChildFragmentManager.dispatchCreate();
    }
}
void restoreChildFragmentState(@Nullable Bundle savedInstanceState) {
    if (savedInstanceState != null) {
        Parcelable p = savedInstanceState.getParcelable(
                FragmentActivity.FRAGMENTS_TAG);
        if (p != null) {
            if (mChildFragmentManager == null) {
                instantiateChildFragmentManager();
            }
            mChildFragmentManager.restoreAllState(p, mChildNonConfig);
            mChildNonConfig = null;
            mChildFragmentManager.dispatchCreate();
        }
    }
}
```

`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 `onActivityCreate()` 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)

```
@Nullable
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup
container,
        @Nullable Bundle savedInstanceState) {
    return null;
}
```

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

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

`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：

```
@Override
public void onViewCreated(final View view, @Nullable final Bundle
savedInstanceState) {
    if (view == null) {
        return;
    }
    mTextView = (TextView) view.findViewById(R.id.tv_content);
    mBtnSwitchChild = (Button) view.findViewById(R.id.btn_switch_child);

    Bundle arguments = getArguments();
    if (arguments != null && mTextView != null &&
!TextUtils.isEmpty(arguments.getString(KEY_TITLE))) {
        mTextView.setText(arguments.getString(KEY_TITLE));
    }
    mBtnSwitchChild.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(final View v) {
            //...
    });
}
```

4. onActivityCreated(Bundle)

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

`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)

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

`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()

```
@CallSuper
public void onStart() {
    mCalled = true;

    if (!mLoadersStarted) {
        mLoadersStarted = true;
        if (!mCheckedForLoaderManager) {
            mCheckedForLoaderManager = true;
            mLoaderManager = mHost.getLoaderManager(mWho, mLoadersStarted,
false);
        }
        if (mLoaderManager != null) {
            mLoaderManager.doStart();
        }
    }
}
```

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

6. onResume()

```
@CallSuper
public void onResume() {
    mCalled = true;
}
```

`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()

```
@CallSuper
public void onPause() {
    mCalled = true;
}
```

`onPause()` 当 Fragment 不再可见时调用。

也和 Activity 的 `onPause()` 同步。

8. onStop()

```
@CallSuper
public void onStop() {
    mCalled = true;
}
```

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

9. onDestroyView()

```
@CallSuper
public void onDestroyView() {
    mCalled = true;
}
```

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()

```
@CallSuper
public void onDestroy() {
    mCalled = true;
    //Log.v("foo", "onDestroy: mCheckedForLoaderManager=" +
mCheckedForLoaderManager
    //        + " mLoaderManager=" + mLoaderManager);
    if (!mCheckedForLoaderManager) {
        mCheckedForLoaderManager = true;
        mLoaderManager = mHost.getLoaderManager(mWho, mLoadersStarted,
false);
    }
    if (mLoaderManager != null) {
        mLoaderManager.doDestroy();
    }
}
```

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()

```
@CallSuper
public void onDetach() {
    mCalled = true;
}
```

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 FragmentImpl 中
- FragmentTransaction
    - 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