《Android基础：Fragment，看这篇就够了》

原创 damonxia 腾讯Bugly 2017-10-19

> | 导语 Fragment作为Android最基本，最重要的基础概念之一，在开发中经常会和他打交道。本文从为什么出现Fragment开始，介绍了Fragment相关的方方面面，包括Fragment的基本定义及使用、回退栈的内部实现、Fragment通信、DialogFragment、ViewPager+Fragment的使用、嵌套Fragment、懒加载等。

基本概念

Fragment，简称碎片，是Android 3.0（API 11）提出的，为了兼容低版本，support-v4库中也开发了一套Fragment API，最低兼容Android 1.6。

过去support-v4库是一个jar包，24.2.0版本开始，将support-v4库模块化为多个jar包，包含：support-fragment, support-ui, support-media-compat等，这么做是为了减少APK包大小，你需要用哪个模块就引入哪个模块。

如果想引入整个support-v4库，则compile `'com.android.support:support-v4:24.2.1'`，如果只想引入support-fragment库，则 `com.android.support:support-fragment:24.2.1`。

> 因为support库是不断更新的，因此建议使用support库中的android.support.v4.app.Fragment，而不要用系统自带的android.app.Fragment。而如果要使用support库的Fragment，Activity必须要继承FragmentActivity（AppCompatActivity是FragmentActivity的子类）。

Fragment官方的定义是：

> A Fragment represents a behavior or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running.

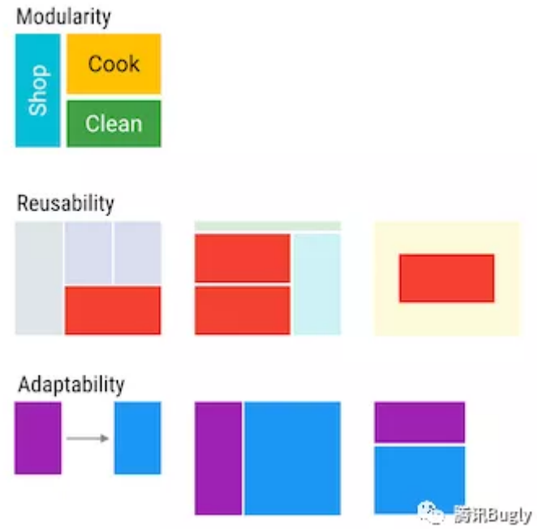
根据上面的定义可知：

- Fragment是依赖于Activity的，不能独立存在的。
- 一个Activity里可以有多个Fragment。
- 一个Fragment可以被多个Activity重用。
- Fragment有自己的生命周期，并能接收输入事件。
- 我们能在Activity运行时动态地添加或删除Fragment。

> Android 3.0系统只针对平板电脑，且闭源，那时候针对手机和针对平板是两套源代码，后来Android 4.0时整合了手机和平板的源码，因此市面上很难看到Android 3.0系统。

Fragment的优势有以下几点：

- 模块化（Modularity）：我们不必把所有代码全部写在Activity中，而是把代码写在各自的Fragment中。
- 可重用（Reusability）：多个Activity可以重用一个Fragment。
- 可适配（Adaptability）：根据硬件的屏幕尺寸、屏幕方向，能够方便地实现不同的布局，这样用户体验更好。



Fragment核心的类有：

- Fragment：Fragment的基类，任何创建的Fragment都需要继承该类。
- FragmentManager：管理和维护Fragment。他是抽象类，具体的实现类是FragmentManagerImpl。
- FragmentTransaction：对Fragment的添加、删除等操作都需要通过事务方式进行。他是抽象类，具体的实现类是BackStackRecord。

Nested Fragment（Fragment内部嵌套Fragment的能力）是Android 4.2提出的，support-fragment库可以兼容到1.6。通过 `getChildFragmentManager()` 能够获得管理子Fragment的FragmentManager，在子Fragment中可以通过 `getParentFragment()` 获得父Fragment。

基本使用

这里给出Fragment最基本的使用方式。首先，创建继承Fragment的类，名为Fragment1：

```
public class Fragment1 extends Fragment{
  private static String ARG_PARAM = "param_key";
     private String mParam;
     private Activity mActivity;
     public void onAttach(Context context) {
        mActivity = (Activity) context;
        mParam = getArguments().getString(ARG_PARAM);  //获取参数
    }
     public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        View root = inflater.inflate(R.layout.fragment_1, container, false);
        TextView view = root.findViewById(R.id.text);
        view.setText(mParam);
             return root;
    }
     public static Fragment1 newInstance(String str) {
        Fragment1 frag = new Fragment1();
        Bundle bundle = new Bundle();
        bundle.putString(ARG_PARAM, str);
        fragment.setArguments(bundle);   //设置参数
        return fragment;
```

```
    }
}
```

Fragment有很多可以复写的方法，其中最常用的就是`onCreateView()`，该方法返回Fragment的UI布局，需要注意的是`inflate()`的第三个参数是false，因为在Fragment内部实现中，会把该布局添加到container中，如果设为true，那么就会重复做两次添加，则会抛如下异常：

```
Caused by: java.lang.IllegalStateException: The specified child already has a parent. You must call removeView() on the child's parent first.
```

如果在创建Fragment时要传入参数，必须要通过`setArguments(Bundle bundle)`方式添加，而不建议通过为Fragment添加带参数的构造函数，因为通过`setArguments()`方式添加，在由于内存紧张导致Fragment被系统杀掉并恢复（re-instantiate）时能保留这些数据。官方建议如下：

```
It is strongly recommended that subclasses do not have other constructors with parameters, since these constructors will not be called when the fragment is re-instantiated.
```

我们可以在Fragment的`onAttach()`中通过`getArguments()`获得传进来的参数，并在之后使用这些参数。如果要获取Activity对象，不建议调用`getActivity()`，而是在`onAttach()`中将Context对象强转为Activity对象。

创建完Fragment后，接下来就是把Fragment添加到Activity中。在Activity中添加Fragment的方式有两种：

- 静态添加：在xml中通过
  的方式添加，缺点是一旦添加就不能在运行时删除。
- 动态添加：运行时添加，这种方式比较灵活，因此建议使用这种方式。

虽然Fragment能在XML中添加，但是这只是一个语法糖而已，Fragment并不是一个View，而是和Activity同一层次的。

这里只给出动态添加的方式。首先Activity需要有一个容器存放Fragment，一般是FrameLayout，因此在Activity的布局文件中加入FrameLayout：

```
<FrameLayout
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

然后在`onCreate()`中，通过以下代码将Fragment添加进Activity中。

```
if (bundle == null) {
    getSupportFragmentManager().beginTransaction()
        .add(R.id.container, Fragment1.newInstance("hello world"), "f1")        //.addToBackStack("fname")
        .commit();
}
```

这里需要注意几点：

- 因为我们使用了support库的Fragment，因此需要使用`getSupportFragmentManager()`获取FragmentManager。
- `add()`是对Fragment众多操作中的一种，还有`remove()`，`replace()`等，第一个参数是根容器的id（FrameLayout的id，即"@id/container"），第二个参数是Fragment对象，第三个参数是fragment的tag名，指定tag的好处是后续我们可以通过`Fragment1 frag = getSupportFragmentManager().findFragmentByTag("f1")`从FragmentManager中查找Fragment对象。
- 在一次事务中，可以做多个操作，比如同时做`add().remove().replace()`。

- `commit()` 操作是异步的，内部通过 `mManager.enqueueAction()` 加入处理队列。对应的同步方法为 `commitNow()`， `commit()` 内部会有 `checkStateLoss()` 操作，如果开发人员使用不当（比如 `commit()` 操作在 `onSaveInstanceState()` 之后），可能会抛出异常，而 `commitAllowingStateLoss()` 方法则是不会抛出异常版本的 `commit()` 方法，但是尽量使用 `commit()`，而不要使用 `commitAllowingStateLoss()`。
- `addToBackStack("fname")` 是可选的。FragmentManager拥有回退栈（BackStack），类似于Activity的任务栈，如果添加了该语句，就把该事务加入回退栈，当用户点击返回按钮，会回退该事务（回退指的是如果事务是 `add(frag1)`，那么回退操作就是 `remove(frag1)`）；如果没添加该语句，用户点击返回按钮会直接销毁Activity。
- Fragment有一个常见的问题，即Fragment重叠问题，这是由于Fragment被系统杀掉，并重新初始化时再次将fragment加入activity，因此通过在外围加if语句能判断此时是否是被系统杀掉并重新初始化的情况。

Fragment有个常见的异常：

```
java.lang.IllegalStateException: Can not perform this action after onSaveInstanceState
    at android.support.v4.app.FragmentManagerImpl.checkStateLoss(FragmentManager.java:1341)
    at android.support.v4.app.FragmentManagerImpl.enqueueAction(FragmentManager.java:1352)
    at android.support.v4.app.BackStackRecord.commitInternal(BackStackRecord.java:595)
    at android.support.v4.app.BackStackRecord.commit(BackStackRecord.java:574)
```

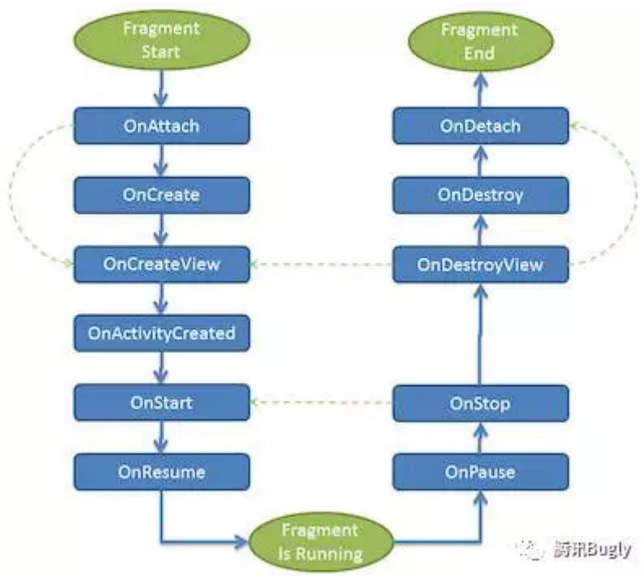
该异常出现的原因是： `commit()` 在 `onSaveInstanceState()` 后调用。首先， `onSaveInstanceState()` 在 `onPause()` 之后， `onStop()` 之前调用。 `onRestoreInstanceState()` 在 `onStart()` 之后，`onResume()` 之前。

因此避免出现该异常的方案有：

- 不要把Fragment事务放在异步线程的回调中，比如不要把Fragment事务放在AsyncTask的 `onPostExecute()`，因此 `onPostExecute()` 可能会在 `onSaveInstanceState()` 之后执行。
- 逼不得已时使用 `commitAllowingStateLoss()`。

生命周期

Fragment的生命周期和Activity类似，但比Activity的生命周期复杂一些，基本的生命周期方法如下图：



解释如下：

- onAttach()：Fragment和Activity相关联时调用。可以通过该方法获取Activity引用，还可以通过getArguments()获取参数。
- onCreate()：Fragment被创建时调用。
- onCreateView()：创建Fragment的布局。
- onActivityCreated()：当Activity完成onCreate()时调用。
- onStart()：当Fragment可见时调用。
- onResume()：当Fragment可见且可交互时调用。
- onPause()：当Fragment不可交互但可见时调用。
- onStop()：当Fragment不可见时调用。
- onDestroyView()：当Fragment的UI从视图结构中移除时调用。
- onDestroy()：销毁Fragment时调用。
- onDetach()：当Fragment和Activity解除关联时调用。

上面的方法中，只有onCreateView()在重写时不用写super方法，其他都需要。

因为Fragment是依赖Activity的，因此为了讲解Fragment的生命周期，需要和Activity的生命周期方法一起讲，即Fragment的各个生命周期方法和Activity的各个生命周期方法的关系和顺序，如图：

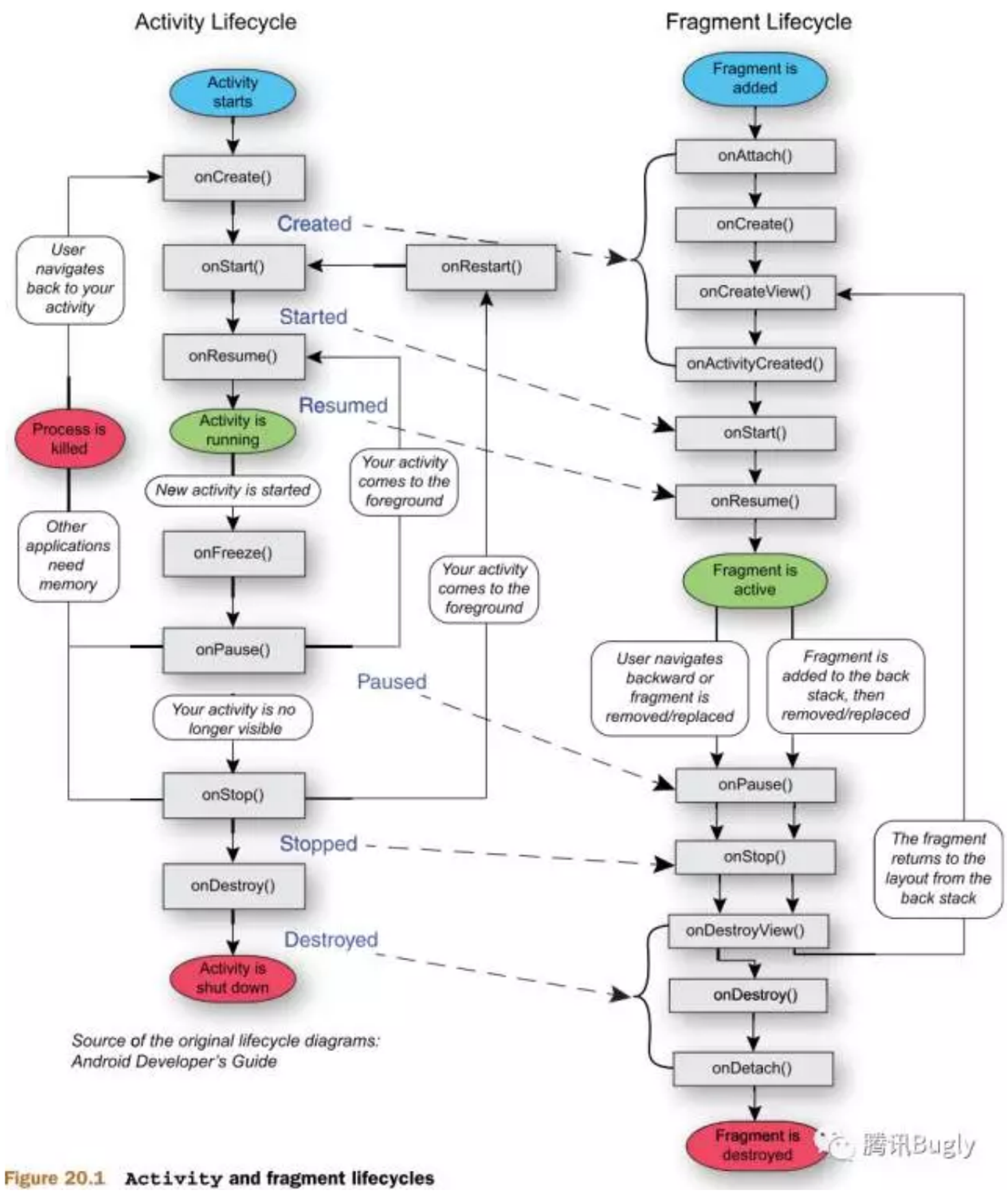


Figure 20.1 Activity and fragment lifecycles

我们这里举个例子来理解Fragment生命周期方法。功能如下：共有两个Fragment：F1和F2，F1在初始化时就加入Activity，点击F1中的按钮调用replace替换为F2。

当F1在Activity的`onCreate()`中被添加时，日志如下：

```
BasicActivity: [onCreate] BEGIN
BasicActivity: [onCreate] END
BasicActivity: [onStart] BEGIN
Fragment1: [onAttach] BEGIN
Fragment1: [onAttach] END
BasicActivity: [onAttachFragment] BEGIN
BasicActivity: [onAttachFragment] END
Fragment1: [onCreate] BEGIN
Fragment1: [onCreate] END
Fragment1: [onCreateView]
```

```
Fragment1: [onViewCreated] BEGIN
Fragment1: [onViewCreated] END
Fragment1: [onActivityCreated] BEGIN
Fragment1: [onActivityCreated] END
Fragment1: [onStart] BEGIN
Fragment1: [onStart] END
BasicActivity: [onStart] END
BasicActivity: [onPostCreate] BEGIN
BasicActivity: [onPostCreate] END
BasicActivity: [onResume] BEGIN
BasicActivity: [onResume] END
BasicActivity: [onPostResume] BEGIN
Fragment1: [onResume] BEGIN
Fragment1: [onResume] END
BasicActivity: [onPostResume] END
BasicActivity: [onAttachedToWindow] BEGIN
BasicActivity: [onAttachedToWindow] END
```

可以看出：

- Fragment的onAttach()->onCreate()->onCreateView()->onActivityCreated()->onStart()都是在Activity的onStart()中调用的。
- Fragment的onResume()在Activity的onResume()之后调用。

接下去分两种情况，分别是不加addToBackStack()和加addToBackStack()。
1、当点击F1的按钮，调用replace()替换为F2，且不加addToBackStack()时，日志如下：

```
Fragment2: [onAttach] BEGIN
Fragment2: [onAttach] END
BasicActivity: [onAttachFragment] BEGIN
BasicActivity: [onAttachFragment] END
Fragment2: [onCreate] BEGIN
Fragment2: [onCreate] END
Fragment1: [onPause] BEGIN
Fragment1: [onPause] END
Fragment1: [onStop] BEGIN
Fragment1: [onStop] END
Fragment1: [onDestroyView] BEGIN
Fragment1: [onDestroyView] END
Fragment1: [onDestroy] BEGIN
Fragment1: [onDestroy] END
Fragment1: [onDetach] BEGIN
Fragment1: [onDetach] END
Fragment2: [onCreateView]
Fragment2: [onViewCreated] BEGIN
Fragment2: [onViewCreated] END
Fragment2: [onActivityCreated] BEGIN
Fragment2: [onActivityCreated] END
Fragment2: [onStart] BEGIN
Fragment2: [onStart] END
Fragment2: [onResume] BEGIN
Fragment2: [onResume] END
```

可以看到，F1最后调用了onDestroy()和onDetach()。

2、当点击F1的按钮，调用replace()替换为F2，且加addToBackStack()时，日志如下：

```
Fragment2: [onAttach] BEGIN
Fragment2: [onAttach] END
BasicActivity: [onAttachFragment] BEGIN
BasicActivity: [onAttachFragment] END
Fragment2: [onCreate] BEGIN
Fragment2: [onCreate] END
Fragment1: [onPause] BEGIN
Fragment1: [onPause] END
Fragment1: [onStop] BEGIN
Fragment1: [onStop] END
Fragment1: [onDestroyView] BEGIN
Fragment1: [onDestroyView] END
Fragment2: [onCreateView]
Fragment2: [onViewCreated] BEGIN
Fragment2: [onViewCreated] END
Fragment2: [onActivityCreated] BEGIN
Fragment2: [onActivityCreated] END
Fragment2: [onStart] BEGIN
Fragment2: [onStart] END
Fragment2: [onResume] BEGIN
Fragment2: [onResume] END
```

可以看到，F1被替换时，最后只调到了 `onDestroyView()` ，并没有调用 `onDestroy()` 和 `onDetach()` 。当用户点返回按钮回退事务时，F1会调onCreateView()->onStart()->onResume()，因此在Fragment事务中加不加 `addToBackStack()` 会影响Fragment的生命周期。

FragmentTransaction有一些基本方法，下面给出调用这些方法时，Fragment生命周期的变化：

- add(): onAttach()->...->onResume()。
- remove(): onPause()->...->onDetach()。
- replace(): 相当于旧Fragment调用remove()，新Fragment调用add()。
- show(): 不调用任何生命周期方法，调用该方法的前提是要显示的Fragment已经被添加到容器，只是纯粹把Fragment UI的setVisibility为true。
- hide(): 不调用任何生命周期方法，调用该方法的前提是要显示的Fragment已经被添加到容器，只是纯粹把Fragment UI的setVisibility为false。
- detach(): onPause()->onStop()->onDestroyView()。UI从布局中移除，但是仍然被FragmentManager管理。
- attach(): onCreateView()->onStart()->onResume()。

Fragment实现原理和Back Stack

我们知道Activity有任务栈，用户通过startActivity将Activity加入栈，点击返回按钮将Activity出栈。Fragment也有类似的栈，称为回退栈（Back Stack），回退栈是由FragmentManager管理的。默认情况下，Fragment事务是不会加入回退栈的，如果想将Fragment事务加入回退栈，则可以加入 `addToBackStack("")` 。如果没有加入回退栈，则用户点击返回按钮会直接将Activity出栈；如果加入了回退栈，则用户点击返回按钮会回滚Fragment事务。

我们将通过最常见的Fragment用法，讲解Back Stack的实现原理：

```
getSupportFragmentManager().beginTransaction()
    .add(R.id.container, f1, "f1")
    .addToBackStack("")
    .commit();
```

上面这个代码的功能就是将Fragment加入Activity中，内部实现为：创建一个BackStackRecord对象，该对象记录了这个事务的全部操作轨迹（这里只做了一次add操作，并且加入回退栈），随后将该对象提交到FragmentManager的执行队列中，等待执行。

BackStackRecord类的定义如下

```
class BackStackRecord extends FragmentTransaction implements FragmentManager.BackStackEntry, Runnable {}
```

从定义可以看出，BackStackRecord有三重含义：

- 继承了FragmentTransaction，即是事务，保存了整个事务的全部操作轨迹。
- 实现了BackStackEntry，作为回退栈的元素，正是因为该类拥有事务全部的操作轨迹，因此在popBackStack()时能回退整个事务。
- 继承了Runnable，即被放入FragmentManager执行队列，等待被执行。

先看第一层含义，`getSupportFragmentManager.beginTransaction()`返回的就是BackStackRecord对象，代码如下：

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

BackStackRecord类包含了一次事务的整个操作轨迹，是以链表形式存在的，链表的元素是Op类，表示其中某个操作，定义如下：

```
static final class Op {
    Op next; //链表后一个节点
    Op prev; //链表前一个节点
    int cmd;  //操作是add或remove或replace或hide或show等
    Fragment fragment; //对哪个Fragment对象做操作}
```

我们来看下具体场景下这些类是怎么被使用的，比如我们的事务做add操作。add函数的定义：

```
public FragmentTransaction add(int containerViewId, Fragment fragment, String tag) {
    doAddOp(containerViewId, fragment, tag, OP_ADD);
    return this;
}
```

doAddOp()方法就是创建Op对象，并加入链表，定义如下：

```
private void doAddOp(int containerViewId, Fragment fragment, String tag, int opcmd) {
    fragment.mTag = tag;  //设置fragment的tag
    fragment.mContainerId = fragment.mFragmentId = containerViewId;  //设置fragment的容器id
    Op op = new Op();
    op.cmd = opcmd;
    op.fragment = fragment;
```

```
    addOp(op);
}
```

addOp()是将创建好的Op对象加入链表，定义如下：

```
void addOp(Op op) {
    if (mHead == null) {
        mHead = mTail = op;
    } else {
        op.prev = mTail;
        mTail.next = op;
        mTail = op;
    }
    mNumOp++;
}
```

`addToBackStack("")`是将mAddToBackStack变量记为true，在`commit()`中会用到该变量。`commit()`是异步的，即不是立即生效的，但是后面会看到整个过程还是在主线程完成，只是把事务的执行扔给主线程的Handler，`commit()`内部是`commitInternal()`，实现如下：

```
int commitInternal(boolean allowStateLoss) {
    mCommitted = true;
    if (mAddToBackStack) {
        mIndex = mManager.allocBackStackIndex(this);
    } else {
        mIndex = -1;
    }
    mManager.enqueueAction(this, allowStateLoss); //将事务添加进待执行队列中
    return mIndex;
}
```

如果mAddToBackStack为true，则调用`allocBackStackIndex(this)`将事务添加进回退栈，FragmentManager类的变量ArrayListmBackStackIndices;就是回退栈。实现如下：

```
public int allocBackStackIndex(BackStackRecord bse) {
    if (mBackStackIndices == null) {
        mBackStackIndices = new ArrayList<BackStackRecord>();
    }
    int index = mBackStackIndices.size();
    mBackStackIndices.add(bse);
        return index;
}
```

在`commitInternal()`中，`mManager.enqueueAction(this, allowStateLoss)`;是将BackStackRecord加入待执行队列中，定义如下：

```
public void enqueueAction(Runnable action, boolean allowStateLoss) {
    if (mPendingActions == null) {
        mPendingActions = new ArrayList<Runnable>();
    }
    mPendingActions.add(action);
    if (mPendingActions.size() == 1) {
        mHost.getHandler().removeCallbacks(mExecCommit);
```

```
        mHost.getHandler().post(mExecCommit); //调用execPendingActions()执行待执行队列的事务
    }
}
```

mPendingActions就是前面说的待执行队列，`mHost.getHandler()`就是主线程的Handler，因此Runnable是在主线程执行的，mExecCommit的内部就是调用了`execPendingActions()`，即把mPendingActions中所有积压的没被执行的事务全部执行。执行队列中的事务会怎样被执行呢？就是调用`BackStackRecord的run()`方法，`run()`方法就是执行Fragment的生命周期函数，还有将视图添加进container中。

与`addToBackStack()`对应的是`popBackStack()`，有以下几种变种：

- popBackStack()：将回退栈的栈顶弹出，并回退该事务。
- popBackStack(String name, int flag)：name为addToBackStack(String name)的参数，通过name能找到回退栈的特定元素，flag可以为0或者FragmentManager.POP_BACK_STACK_INCLUSIVE，0表示只弹出该元素以上的所有元素，POP_BACK_STACK_INCLUSIVE表示弹出包含该元素及以上的所有元素。这里说的弹出所有元素包含回退这些事务。
- popBackStack()是异步执行的，是丢到主线程的MessageQueue执行，popBackStackImmediate()是同步版本。

我们通过讲解Demo1来更清晰地了解回退栈的使用。功能如下：共有三个Fragment：F1, F2, F3，F1在初始化时就加入Activity，点击F1中的按钮跳转到F2，点击F2的按钮跳转到F3，点击F3的按钮回退到F1。

在Activity的onCreate()中，将F1加入Activity中：

```
getSupportFragmentManager().beginTransaction()
    .add(R.id.container, f1, "f1")
    .addToBackStack(Fragment1.class.getSimpleName())
    .commit();
```

F1按钮的onClick()内容如下：

```
getFragmentManager().beginTransaction()
    .replace(R.id.container, f2, "f2")
    .addToBackStack(Fragment2.class.getSimpleName())
    .commit();
```

F2按钮的onClick()如下：

```
getFragmentManager().beginTransaction()
    .replace(R.id.container, f3, "f3")
    .addToBackStack(Fragment3.class.getSimpleName())
    .commit();
```

F3按钮的onClick()如下：

```
getFragmentManager().popBackStack(Fragment2.class.getSimpleName(), FragmentManager.POP_BACK_STACK_INCLUSIVE);
```

这样就完成了整个界面的跳转逻辑。

这里补充一个点

`getSupportFragmentManager().findFragmentByTag()`是经常用到的方法，他是FragmentManager的方法，FragmentManager是抽象类，FragmentManagerImpl是继承FragmentManager的实现类，他的内部实现是：

```
class FragmentManagerImpl extends FragmentManager {
    ArrayList<Fragment> mActive;
    ArrayList<Fragment> mAdded;
    public Fragment findFragmentByTag(String tag) {
           if (mAdded != null && tag != null) {
               for (int i=mAdded.size()-1; i>=0; i--) {
                Fragment f = mAdded.get(i);
                if (f != null && tag.equals(f.mTag)) {
                        return f;
                }
             }
        }
          if (mActive != null && tag != null) {
               for (int i=mActive.size()-1; i>=0; i--) {
                    Fragment f = mActive.get(i);
                    if (f != null && tag.equals(f.mTag)) {
                          return f;
                }
             }
        }
          return null;
    }
}
```

从上面看到，先从mAdded中查找是否有该Fragment，如果没找到，再从mActive中查找是否有该Fragment。mAdded是已经添加到Activity的Fragment的集合，mActive不仅包含mAdded，还包含虽然不在Activity中，但还在回退栈中的Fragment。

Fragment通信

Fragment向Activity传递数据

首先，在Fragment中定义接口，并让Activity实现该接口（具体实现省略）：

```
public interface OnFragmentInteractionListener {    void onItemClick(String str);  //将str从Fragment传递给Activity}
```

在Fragment的onAttach()中，将参数Context强转为OnFragmentInteractionListener对象：

```
public void onAttach(Context context) {
    super.onAttach(context);
        if (context instanceof OnFragmentInteractionListener) {
        mListener = (OnFragmentInteractionListener) context;
    } else {
                throw new RuntimeException(context.toString()
                + " must implement OnFragmentInteractionListener");
```

```
    }
}
```

并在Fragment合适的地方调用 `mListener.onItemClick("hello")` 将”hello”从Fragment传递给Activity。

FABridge

由于通过接口的方式从Fragment向Activity进行数据传递比较麻烦，需要在Fragment中定义interface，并让Activity实现该interface，FABridge(https://github.com/hongyangAndroid/FABridge)通过注解的形式免去了这些定义。

在build.gradle中添加依赖：

```
annotationProcessor 'com.zhy.fabridge:fabridge-compiler:1.0.0'compile 'com.zhy.fabridge:fabridge-api:1.0.0'
```

首先定义方法ID，这里为FAB_ITEM_CLICK，接着在Activity中定义接口：

```
@FCallbackId(id = FAB_ITEM_CLICK)public void onItemClick(String str) {  //方法名任意
    Toast.makeText(this, str, Toast.LENGTH_SHORT).show();
}
```

最后，在Fragment中，通过以下形式调用”ID=FAB_ITEM_CLICK”的方法（该方法可能在Activity中，也可能在任何类中）：

```
Fabridge.call(mActivity,FAB_ITEM_CLICK,"data");  //调用ID对应的方法，"data"为参数值
```

Activity向Fragment传递数据

Activity向Fragment传递数据比较简单，获取Fragment对象，并调用Fragment的方法即可，比如要将一个字符串传递给Fragment，则在Fragment中定义方法：

```
public void setString(String str) {
    this.str = str;
}
```

并在Activity中调用 `fragment.setString("hello")` 即可。

Fragment之间通信

由于Fragment之间是没有任何依赖关系的，因此如果要进行Fragment之间的通信，建议通过Activity作为中介，不要Fragment之间直接通信。

DialogFragment

DialogFragment是Android 3.0提出的，代替了Dialog，用于实现对话框。他的优点是：即使旋转屏幕，也能保留对话框状态。

如果要自定义对话框样式，只需要继承DialogFragment，并重写 `onCreateView()` ，该方法返回对话框UI。这里我们举个例子，实现进度条样式的圆角对话框。

```
public class ProgressDialogFragment extends DialogFragment {    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        getDialog().requestWindowFeature(Window.FEATURE_NO_TITLE); //消除Title区域
        getDialog().getWindow().setBackgroundDrawable(new ColorDrawable(Color.TRANSPARENT));  //将背景变为透明
        setCancelable(false);  //点击外部不可取消
        View root = inflater.inflate(R.layout.fragment_progress_dialog, container);
        return root;
    }
     public static ProgressDialogFragment newInstance() {
        return new ProgressDialogFragment();
    }
}
```

进度条动画我们使用Lottie(https://github.com/airbnb/lottie-android)实现，Lottie动画从这里(https://www.lottiefiles.com/)找到。使用非常方便，只需要下载JSON动画文件，然后在XML中写入：

```
<com.airbnb.lottie.LottieAnimationView
    android:layout_width="wrap_content"  //大小根据JSON文件确定
    android:layout_height="wrap_content"
    app:lottie_fileName="loader_ring.json"    //JSON文件
    app:lottie_loop="true"     //循环播放
    app:lottie_autoPlay="true" />  //自动播放
```

然后通过下面代码显示对话框：

```
ProgressDialogFragment fragment = ProgressDialogFragment.newInstance();
fragment.show(getSupportFragmentManager(), "tag");//fragment.dismiss();
```

为了实现圆角，除了在onCreateView()中把背景设为透明，还需要对UI加入背景：

```
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <solid android:color="#ffffff"/>
    <corners
        android:radius="20dp"/>
</shape>
```

ViewPager+Fragment相关

基本使用

ViewPager是support v4库中提供界面滑动的类，继承自ViewGroup。PagerAdapter是ViewPager的适配器类，为ViewPager提供界面。但是一般来说，通常都会使用PagerAdapter的两个子类：FragmentPagerAdapter和FragmentStatePagerAdapter作为ViewPager的适配器，他们的特点是界面是Fragment。

> 在support v13和support v4中都提供了FragmentPagerAdapter和FragmentStatePagerAdapter，区别在于：support v13中使用android.app.Fragment，而support v4使用android.support.v4.app.Fragment。一般都使用support v4中的FragmentPagerAdapter和FragmentStatePagerAdapter。

默认，ViewPager会缓存当前页相邻的界面，比如当滑动到第2页时，会初始化第1页和第3页的界面（即Fragment对象，且生命周期函数运行到`onResume()`），可以通过`setOffscreenPageLimit(count)`设置离线缓存的界面个数。

FragmentPagerAdapter和FragmentStatePagerAdapter需要重写的方法都一样，常见的重写方法如下：

- public FragmentPagerAdapter(FragmentManager fm): 构造函数，参数为FragmentManager。如果是嵌套Fragment场景，子 PagerAdapter的参数传入getChildFragmentManager()。
- Fragment getItem(int position): 返回第position位置的Fragment，必须重写。
- int getCount(): 返回ViewPager的页数，必须重写。
- Object instantiateItem(ViewGroup container, int position): container是ViewPager对象，返回第position位置的Fragment。
- void destroyItem(ViewGroup container, int position, Object object): container是ViewPager对象，object是Fragment对象。
- getItemPosition(Object object): object是Fragment对象，如果返回POSITION_UNCHANGED，则表示当前Fragment不刷新，如果返回POSITION_NONE，则表示当前Fragment需要调用`destroyItem()`和`instantiateItem()`进行销毁和重建。 默认情况下返回POSITION_UNCHANGED。

懒加载

懒加载主要用于ViewPager且每页是Fragment的情况，场景为微信主界面，底部有4个tab，当滑到另一个tab时，先显示"正在加载"，过一会才会显示正常界面。

默认情况，ViewPager会缓存当前页和左右相邻的界面。实现懒加载的主要原因是：用户没进入的界面需要有一系列的网络、数据库等耗资源、耗时的操作，预先做这些数据加载是不必要的。

这里懒加载的实现思路是：用户不可见的界面，只初始化UI，但是不会做任何数据加载。等滑到该页，才会异步做数据加载并更新UI。

这里就实现类似微信那种效果，整个UI布局为：底部用PagerBottomTabStrip(https://github.com/tyzlmjj/PagerBottomTabStrip)项目实现，上面是ViewPager，使用FragmentPagerAdapter。

逻辑为：当用户滑到另一个界面，首先会显示正在加载，等数据加载完毕后（这里用睡眠1秒钟代替）显示正常界面。

ViewPager默认缓存左右相邻界面，为了避免不必要的重新数据加载（重复调用`onCreateView()`），因为有4个tab，因此将离线缓存的半径设置为3，即`setOffscreenPageLimit(3)`。

懒加载主要依赖Fragment的`setUserVisibleHint(boolean isVisible)`方法，当Fragment变为可见时，会调用`setUserVisibleHint(true)`；当Fragment变为不可见时，会调用`setUserVisibleHint(false)`，且该方法调用时机：

- onAttach()之前，调用`setUserVisibleHint(false)`。
- onCreateView()之前，如果该界面为当前页，则调用`setUserVisibleHint(true)`，否则调用`setUserVisibleHint(false)`。
- 界面变为可见时，调用`setUserVisibleHint(true)`。
  *界面变为不可见时，调用`setUserVisibleHint(false)`。

懒加载Fragment的实现：

```
public class LazyFragment extends Fragment {    private View mRootView;
    private boolean mIsInited;
    private boolean mIsPrepared;    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        mRootView = inflater.inflate(R.layout.fragment_lazy, container, false);
        mIsPrepared = true;
        lazyLoad();
            return mRootView;
```

```
}

 public void lazyLoad() {
    if (getUserVisibleHint() && mIsPrepared && !mIsInited) {
           //异步初始化，在初始化后显示正常UI
        loadData();
    }
}

 private void loadData() {
     new Thread() {
     public void run() {
           //1. 加载数据
           //2. 更新UI
           //3. mIsInited = true
        }
    }.start();
}

@Override
public void setUserVisibleHint(boolean isVisibleToUser) {
       super.setUserVisibleHint(isVisibleToUser);
        if (isVisibleToUser) {
        lazyLoad();
    }
}

public static LazyFragment newInstance() {
       return new LazyFragment();
    }
}
```

注意点：

- 在Fragment中有两个变量控制是否需要做数据加载：
  - mIsPrepared：表示UI是否准备好，因为数据加载后需要更新UI，如果UI还没有inflate，就不需要做数据加载，因为setUserVisibleHint()会在onCreateView()之前调用一次，如果此时调用，UI还没有inflate，因此不能加载数据。
  - mIsInited：表示是否已经做过数据加载，如果做过了就不需要做了。因为setUserVisibleHint(true)在界面可见时都会调用，如果滑到该界面做过数据加载后，滑走，再滑回来，还是会调用setUserVisibleHint(true)，此时由于mIsInited=true，因此不会再做一遍数据加载。
- lazyLoad()：懒加载的核心类，在该方法中，只有界面可见（getUserVisibleHint()==true）、UI准备好（mIsPrepared==true）、过去没做过数据加载（mIsInited==false）时，才需要调loadData()做数据加载，数据加载做完后把mIsInited置为true。

布局XML主要分两个container，一个是初始显示的状态，即R.id.container_empty，当数据加载完成，就显示R.id.container：

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <RelativeLayout
        android:id="@+id/container_empty"
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerInParent="true"
            android:text="正在加载"
            />

    </RelativeLayout>
    <RelativeLayout
        android:id="@+id/container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:visibility="gone"
        >
        ...
    </RelativeLayout>
</FrameLayout>
```

参考文献

- 入门(https://www.raywenderlich.com/169885/android-fragments-tutorial-introduction-2)
- 教程1
  (http://assets.en.oreilly.com/1/event/68/Fragments%20for%20All%20Presentation.pdf)
- 教程2
  (http://vinsol.com/blog/2014/09/15/advocating-fragment-oriented-applications-in-android/)
- 生命周期(https://github.com/xxv/android-lifecycle)
- detach vs remove
  (https://stackoverflow.com/questions/9156406/whats-the-difference-between-detaching-a-fragment-and-removing-it)
- Google I/O 2016: What the Fragment?(https://www.youtube.com/watch?v=k3IT-IJ0J98)
- Google I/O 2017: Fragment Tricks
  (https://www.youtube.com/watch?v=eUG3VWnXFtg)
- mAdded和mActive的区别(https://stackoverflow.com/questions/25695960/difference-between-madded-mactive-in-source-code-of-support-fragmentmanager)
- 如何避免IllegalStateException异常(http://www.androiddesignpatterns.com/2013/08/fragment-transaction-commit-state-loss.html)
