转载请注明转载自：http://blog.csdn.net/feather_wch/article/details/79462351

Fragment有这一篇就够了！汲取各路大神精华，亲自从源码剖析Fragment工作原理和底层机制。

包含:

> 1. Fragment基础-讲解Fragment基本用法和很多注意要点
> 2. Fragment相关的FragmentManager和FragmentTransaction讲解
> 3. Fragment通信
> 4. 从源码角度分析Fragment底层机制和原理

Android Fragment详解

版本: 2019/3/30-1(20:23)

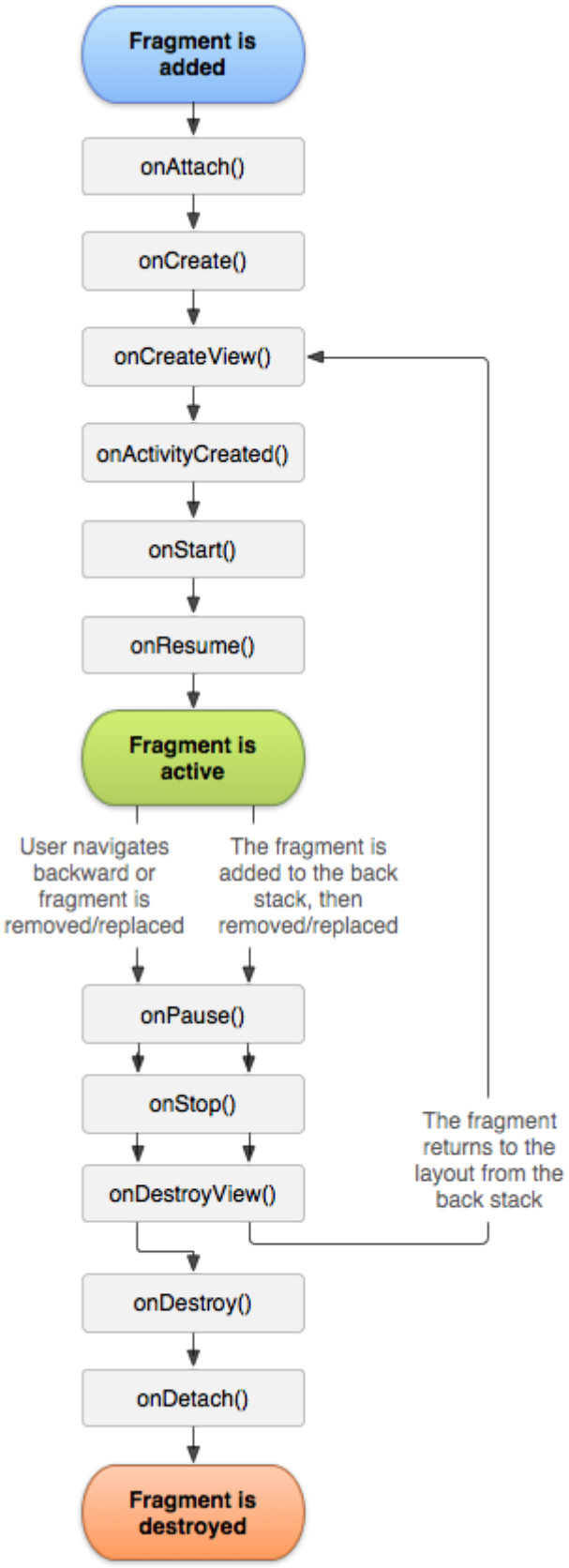
Fragment基础

1、Fragment特点

> 1. Activity界面的一部分，必须依赖Activity
> 2. `Fragment` 拥有自己的 `生命周期` ，可以在 `Activity` 内部被添加或者删除
> 3. `Fragment` 的生命周期只接受所在的 `Activity` 影响（ `Activity` 暂停，其 `Fragment` 也会暂停）
> 4. `Fragment` 在 `Android 3.0` 引入，低版本中需要采用 `android-support-v4.jar` 兼容包
> 5. `Fragment` 的可重用：多个 `Activity` 可以重用一个 `Fragment`

生命周期

2、官方生命周期图



3、Fragment生命周期方法调用场景

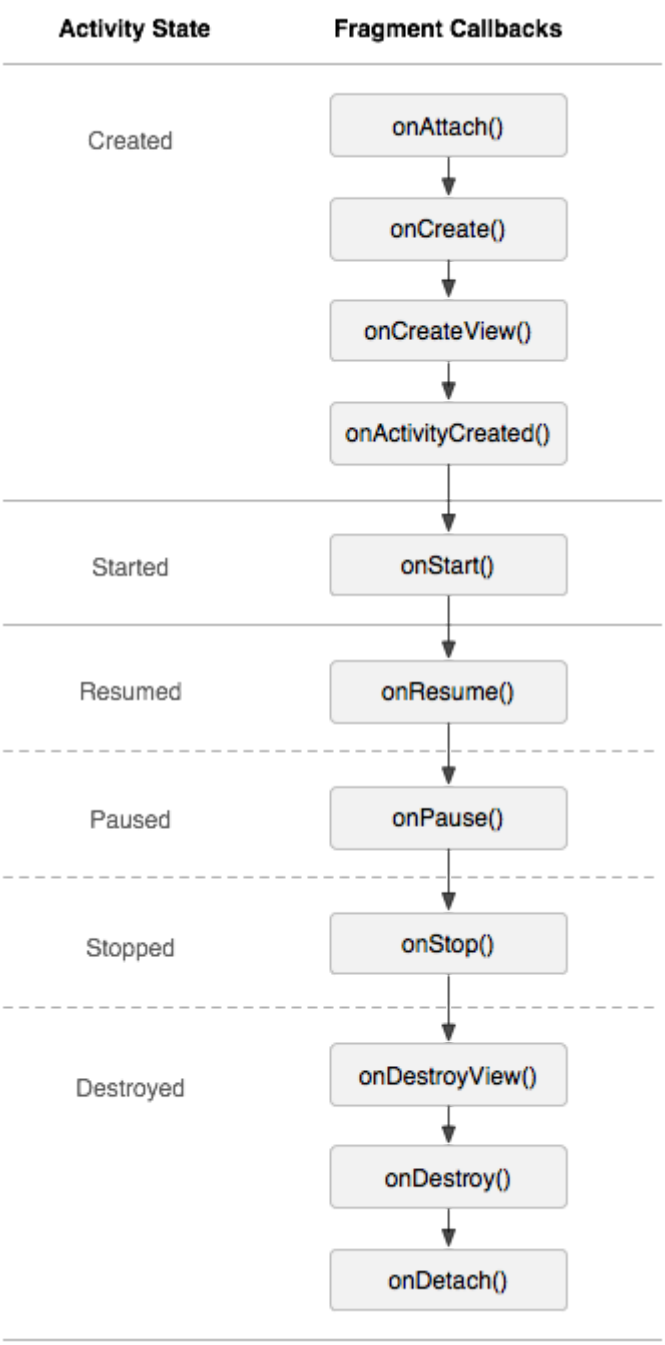
方法	解释
onAttach()	Fragment和Activity建立关联时调用(获取 `Activity` 传递的值)
onCreateView()	为 `Fragment` 创建View(加载布局)时调用

方法	解释
onActivityCreated()	`Activity` 的 `onCreate()` 方法执行完毕后调用
onDestoryView()	`Fragment` 的布局被移除时调用
onDetach()	`Fragment` 和 `Activity` 解除关联的时候调用

4、Fragment生命周期场景

场景	生命周期调用顺序
Fragment被创建	onAttach()->onCreate()->onCreateView()->onActivityCreated()
Fragment可见	onStart()->onResume()
Fragment进入后台	onPause()->onStop()
Fragment被销毁/退出应用	onPause()->onStop()->onDestoryView()->onDestory()->onDetach()
屏幕灭掉/回到桌面	onPause()->onSaveInstanceState()->onStop()
屏幕解锁/回到应用	onStart()->onResume()
切换到其他Fragment	onPause()->onStop()->onDestoryView()
切换回本身Fragment	onCreateView()->onActivityCreated()->onStart()->onResume()

5、Fragment和Activity生命周期对比图



> 1. 创建流程中：先执行Activity生命周期回调，再执行Fragment方法。
> 2. 销毁流程中：先执行Fragment生命周期回调，再执行Activity方法。

添加

静态

6、Fragment的静态添加-在Activity的布局文件中添加

- Fragment

```
//BlankFragment.java
public class BlankFragment extends Fragment {

    public BlankFragment() {
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // 1. 加载布局，第三个参数必须为`false`，否则会加载两次布局并且抛出异常！！
        return inflater.inflate(R.layout.fragment_blank, container, false);
    }
}


//fragment的布局
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.a6005001819.androiddeveloper.Fragment.BlankFragment">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="@string/hello_blank_fragment" />

</FrameLayout>
```

- Activity

```
//如一般Activity代码-直接省略
如果`Fragment`使用的是`support-v4`中的，Activity需要继承自`FragmentActivity`
```

- Activity的布局

```
//用fragment标签加载fragment
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.example.a6005001819.androiddeveloper.MainActivity">

    <fragment
      android:id="@+id/example_fragment"
        android:name="com.example.a6005001819.androiddeveloper.Fragment.BlankFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</LinearLayout>
```

7、Fragment静态添加注意点

> 1. 如果 Fragment 属于 android.support.v4.app.Fragment， 所用的 Activity 必须继承自 FragmentActivity
> 2. 如果 Fragment 属于 android.app.Fragment， 直接使用 Activity即可

动态

8、Fragment的动态添加

- Fragment代码不变
- Activity中进行动态添加

```
public class MainJavaActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_java);

        //1. 获取FragmentManager
        FragmentManager fragmentManager = getFragmentManager();
        //2. 获取FragmentTransaction
        FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
        //3. 创建Fragment
        BlankFragment blankFragment = new BlankFragment();
        //4. 将Fragment替换到Activity的布局中(Framelayout)
        fragmentTransaction.add(R.id.main_java_activity_container, blankFragment);
        fragmentTransaction.commit();
    }
}
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.a6005001819.androiddeveloper.MainJavaActivity">

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

</android.support.constraint.ConstraintLayout>
```

重复添加

9、Fragment的重复添加

> 问题： Fragment 的动态添加是在 onCreate() 中，当手机横竖屏切换，会导致多次调用 onCreate() 最终导致创建多个 Fragment
> 解决办法：在 onCreate() 检查 参数savedInstanceState 来确定是第一次运行还是恢复后运行

```java
//java版本
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main_java);

    if(savedInstanceState == null){
        //1. 获取FragmentManager(support-v4中用`getSupportFragmentManager`)
        FragmentManager fragmentManager = getFragmentManager();
        //2. 获取FragmentTransaction
        FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
        //3. 创建Fragment
        BlankFragment blankFragment = new BlankFragment();
        //4. 将Fragment替换到Activity的布局中(Framelayout)
        fragmentTransaction.add(R.id.main_java_activity_container, blankFragment);
        fragmentTransaction.commit();

        Log.i("feather", "创建新的Fragment");
    }
}
```

```
//kotlin版本
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main_kotlin)

    if(savedInstanceState == null){
        fragmentManager.beginTransaction()
                .add(R.id.main_kotlin_activity_container, BlankFragment())
                .commit()
        Log.i("feather", "创建新的Fragment")
    }
}
```

Fragment基础要点

10、Fragment中 `onCreateView()`

> 1. 内部加载布局的 `inflate()` 的第三个参数必须是 `false` ， 否则会导致重复添加，抛出异常

11、Fragment需要传入参数该怎么做?

> 1. 应该通过 `setArguments(Bundle bundle)` 方法添加
> 2. 之后通过 `onAttach()` 中的 `getArguments()` 获得传进来的参数。
> 3. 不建议通过给 `Fragment` 添加 带参数的构造函数 来传参数，这样 内存紧张 时被系统杀掉会导致 无法恢复数据

12、Fragment中获取 `Activity` 对象

> 1. 不应该！使用 `getActivity()` 获取
> 2. 应该在 `onAttach()` 中将 `Context`对象 强转为 `Activity`对象

13、Fragment不是一个View，而是和Activity一个层级

14、不能在 `onSaveInstanceState()` 后调用 `commit()`

> 1. onSaveInstanceState()在onPause()和onStop()之间调用。
> 2. onRestoreInstanceState()在onStart()和onResume()之间调用。
>   避免异常方法：
> 3. 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()---onPostExecute()可能会在onSaveInstanceState()之后执行。
> 4. 无可奈何时使用commitAllowingStateLoss()-一般不推荐。

15、生命周期场景：Fragment1在Activity初始化时就添加

> 1. Fragment的onAttach()->onCreate()->onCreateView()->onActivityCreated()->onStart()都是在Activity的onStart()中调用的。
> 2. Fragment的onResume()在Activity的onResume()之后调用。

16、生命周期场景：15的情况下，点击一个Button执行replace将F1替换为F2(不加addToBackStack())

1. F2.onAttach()->Activity.onAttachFragment()->F2.onCreate()->F1.onPause()$_{onDetach()->F2.onCreateView()}$onResume()
2. F1进行销毁并添加了F2

17、生命周期场景：15的情况下，点击一个Button执行replace将F1替换为F2(加addToBackStack())

1. F1被替换只会调用 `onDestoryView()` 而不会调用后续生命周期。

FragmentTransaction

18、FragmentTransaction的 `replace()` 方法

1. 作用： 等于先 `Remove` 移除容器所有的 `Fragment` ， 然后再 `add`一个`Fragment`
2. 在 `replace()` 前调用多次 `add()`，最终只有 `replace()` 的 `Fragment` 留存

```
getSupportFragmentManager().beginTransaction()
                .add(R.id.fl_main, new ContentFragment(), null)
                .add(R.id.fl_main, new ContentFragment(), null)
                .add(R.id.fl_main, new ContentFragment(), null)
                .replace(R.id.fl_main, new ContentFragment(), null)
                .commit();
```

19、FragmentTransaction的 `addToBackStack()` 方法

1. 将此次 事物 添加到后台堆栈， 按下“回退键” 不会退出该 `Activity` 而是回到上一个操作时的状态。
2. 按下“回退键” =执行了 `add()` 对应的 `remove()`
3. 加入 回退栈 ， 则用户点击返回按钮，会 回滚`Fragment`事务

```
getSupportFragmentManager().beginTransaction()
                    .add(R.id.fl_main, new ContentFragment(), null)
                    .addToBackStack(null) //1. 记录之前的add操作，回退会出现“白屏”，也就是回到`ad
                    .commit();
```

20、FragmentTransaction的 `show()`、`hide()` 方法

1. `hide(fragment)` -隐藏一个存在的fragment
2. `show(fragment)` -显示一个被隐藏过的fragment
3. `hide()`和`show()` 不会调用任何 生命周期方法
4. `Fragment` 中重写 `onHiddenChanged()` 可以对 `Fragment` 的 `hide`、`show` 状态进行监听

21、FragmentTransaction的 `attach()`、`detach()` 方法

1. `detach(fragment)` -分离fragment的UI视图

2. attach(fragment) -重新关联一个Fragment(必须在 detach 后才能执行)
3. 当Fragment被detach后，Fragment的生命周期执行完onDestroyView就终止---Fragment的实例并没有被销毁，只是UI界面被移除（和remove有区别）。
4. 当Fragment被detach后，执行attach操作，会让Fragment从onCreateView开始执行，一直执行到onResume。

22、FragmentTransaction的 setCustomAnimations()

1. 给 Fragment 的 进入/退出 设置指定的动画资源
2. getSupportFragmentManager 不支持属性动画，仅支持 补间动画 。 getFragmentManager 支持 属性动画 。
3. setCustomAnimations 一定要放在add、remove...等操作之前。

```
//1. 载入Fragment动画
getFragmentManager().beginTransaction()
  //1. 设置动画
  .setCustomAnimations(R.animator.enter_animator, R.animator.exit_animator)
    .add(R.id.main_java_activity_container,  new BlankFragment())
      .commit();
//2. 销毁Fragment时动画(support-v4中只能用)
getSupportFragmentManager().beginTransaction()
                    .setCustomAnimations(R.anim.enter_anim, R.anim.exit_anim)
                    .remove(getSupportFragmentManager().findFragmentById(R.id.fl_main))
                    .commit();


//动画:
// res/animator/enter_animator
<?xml version="1.0" encoding="utf-8"?>
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:interpolator/accelerate_decelerate"
    android:valueFrom="-1280"
    android:valueTo="0"
    android:valueType="floatType"
    android:propertyName="X"
    android:duration="2000"  />

// res/animator/exit_animator
<?xml version="1.0" encoding="utf-8"?>
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:interpolator/accelerate_decelerate"
    android:valueFrom="0"
    android:valueTo="1280"
    android:valueType="floatType"
    android:propertyName="X"
    android:duration="2000" />
```

23、FragmentTransaction的 addSharedElement(View sharedElement, String name)

1. 用于将 View 从一个 被删除的或隐藏 的 Fragment 映射到另一个 显示或添加的Fragment 上。

2. 设置共享View，第二个参数name是这个共享元素的标识。
3. 博客推荐： 使用SharedElement切换Fragment页面:https://www.jianshu.com/p/e9f63ead8bf5

24、FragmentTransaction

的 `commit()`、`commitAllowingStateLoss()`、`commitNow()`、`commitNowAllowingStateLoss()`

1. `commit()`： 安排一个事务的提交。
2. `commitAllowingStateLoss()`： 和commit一样，但是允许Activity的状态保存之后提交。
3. `commitNow()`： 同步的提交这个事务。（API_24添加）
4. `commitNowAllowingStateLoss()`： 和commitNow()一样，但是允许Activity的状态保存之后提交。（API_24添加）

FragmentManager

25、FragmentManagerImpl是什么?

1. 是 `FragmentManager` 的实现类
2. `mAdded` 是已经添加到Activity的Fragment的集合
3. `mActive` 不仅包含 `mAdded` ， 还包含 虽然不在`Activity`中，但还在回退栈 中的Fragment。

```
final class FragmentManagerImpl extends FragmentManager implements LayoutInflater.Factory2 {
    //1. 待加入的Fragment列表
    final ArrayList<Fragment> mAdded = new ArrayList<>();
    //2. 活跃的Fragment列表
    SparseArray<Fragment> mActive;
    //3. 回退栈
    ArrayList<BackStackRecord> mBackStack;
    //4. 通过ID查找Fragment
    public Fragment findFragmentById(int id) {...}
    //5. 通过tag查找Fragment
    public Fragment findFragmentByTag(String tag) {...}
    //6. 获得所有"待添加列表"中的Fragment
    public List<Fragment> getFragments() {...}
}
```

26、findFragmentById()

1. 根据ID来找到对应的Fragment实例，主要用在静态添加fragment的布局中，因为静态添加的fragment才会有ID
2. 会先从 `mAdded`(待添加列表) 中查找，没有找到就会从 `mActive`(活跃列表) 中查找

27、findFragmentByTag()

1. 根据TAG找到对应的Fragment实例，主要用于在动态添加的fragment中，根据TAG来找到fragment实例
2. 会先从 `mAdded`(待添加列表) 中查找，没有找到就会从 `mActive`(活跃列表) 中查找

28、getFragments()

> 1. 返回 mAdded(待添加列表)

29、popBackStack()的作用

> 1. 进行 回退 功能，对应于 addToBackStack()
> 2. void popBackStack(int id, int flags); ---根据Fragment的ID进行回退，ID可以通过Commit返回值获取
> 3. void popBackStack(String name, int flags); ---根据Tag进行回退
> 4. flags = 0 时， 参数指定的Fragment 以上的所有内容全部 出栈
> 5. flags = POP_BACK_STACK_INCLUSIVE 时， 参数指定的Fragment (包括该Fragment)以上的所有内容全部 出栈
> 6. popBackStackImmediate() 相关所有函数，都是立即执行。而不是将 事务 插入队列后等待执行。

DialogFragment

30、DialogFragment的作用和优缺点

> 1. DialogFragment 在 Android 3.0 提出，用于替代 Dialog
> 2. 优点： 旋转屏幕也能保留对话框状态
> 3. 自定义对话框样式：继承 DialogFragment 并重写 onCreateView()， 该方法返回 对话框UI

```
//自定义DialogFragment
public class CustomDialogFragment extends DialogFragment{

    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, Bundle sav
        //1. 消除标题
        getDialog().requestWindowFeature(Window.FEATURE_NO_TITLE);
        //2. 自定义样式
        View root = inflater.inflate(R.layout.fragment_blank, container);
        return root;
    }
}
//使用DialogFragment
CustomDialogFragment dialogFragment = new CustomDialogFragment();
dialogFragment.show(getFragmentManager(), "dialog");
```

Fragment通信

31、EventBus是解决Fragment通信的终极解决方案

> 1. Fragment 通信涉及：Fragment向Activity传递数据、Activity向Fragment传递数据、Fragment之间床底数据
> 2. EventBus：一个Android事件发布/订阅轻量级框架---能够便捷、高效解决 Fragment所有通信问题

Fragment和Activity

32、Fragment调用Activity中的方法

> 1. getActivity()或者onAttach()中Context转为Activity
> 2. 用该Activity对象，调用Activity的对象方法。

33、Activity调用Fragment中的方法

> 1. Activity中直接用该Fragment对象去调用方法。
> 2. 用过接口回调

34、Fragment向Activity传递数据的方法

> 1. 接口回调: 在 Fragment 中 定义接口 ， 并让 Activity实现该接口
> 2. FABridge: 以 注解 的形式免去了 接口回调 的各种步骤，github: (https://github.com/hongyangAndroid/FABridge)

```
//接口回调实例
//1、Fragment中定义接口
public interface onFragmentInteractionListerner{
    void onItemClick(String msg);
}
//2、Activity实现接口
public class MainActivity extends Activity
   implements BlankFragment.onFragmentInteractionListerner{

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main_java);
       //1. Activity作为参数传入
       Fragment f = new Fragment(MainActivity.this);
    }
    //2. 操作从Fragment传递进来的数据
    public void onItemClick(String msg) {
        Log.i("feather", "Activity:" + msg);
    }
}
//3、Fragment中转换Context为接口
onFragmentInteractionListerner mInterface;
public BlankFragment(Context context) {
    mInterface = (onFragmentInteractionListerner) context;
}
//4、Fragment中调用方法进行数据传递
mInterface.onItemClick("data from fragment");
```

35、Activity向Fragment传递数据

> 1. 通过 构造函数 传递数据(不推荐)
> 2. 通过 参数 传递数据，在 Fragment 中 onAttach() 中 getArguments()
> 3. 在 Activity 中获取 Fragment 对象，直接调用 Fragment的方法 ， 通过 参数传递数据

Fragment之间

36、Fragment之间的通信

> 1. 需要借助 Activity 进行数据通信
> 2. Activity中通过FragmentManager的 findFragmentById 去获取Fragment

Fragment与ViewPager

> ViewPager的基本使用：请参考该链接

37、ViewPager介绍

> 1. ViewPager 常用于实现 Fragment的左右滑动切换 的效果
> 2. 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Fragment对象，且生命周期函数运行到onResume()。
> 3. 通过 ViewPager的setOffscreenPageLimit(count) 设置离线缓存的界面个数。

38、Fragment的FragmentPagerAdapter的实现方法

```
public class MyFragmentStatePagerAdapter extends FragmentStatePagerAdapter{
    //1. Fragment链表
    List<Fragment> fragmentList;
    //2. 构造
    public MyFragmentStatePagerAdapter(android.support.v4.app.FragmentManager fm, List<Fragment
        super(fm);
        fragmentList = list;
    }
    //3. 返回当前Position对应的Fragment
    @Override
    public Fragment getItem(int position) {
        return fragmentList.get(position);
    }
    //4. Fragment的总数量
    @Override
    public int getCount() {
        return fragmentList.size();
    }
}
```

懒加载

39、懒加载是什么

> 1. ViewPager 默认会 预加载 左右相邻的 Fragment ，但是在一些有耗时操作的情况下，需要 懒加载 -在打开相应 Fragment 时才加载数据
> 2. 懒加载的实现思路 ： 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。

40、懒加载之Fragment的setUserVisibleHint()方法

> 1. 懒加载需要依赖Fragment的setUserVisibleHint()方法
> 2. 当Fragment变为 可见 时，需要调用 setUserVisibleHint(true)
> 3. 当Fragment变为 不可见 时，会调用 setUserVisibleHint(false)

31、setUserVisibleHint()的调用时机

> 1. onAttach() 之前，调用 setUserVisibleHint(false) 。
> 2. onCreateView() 之前，如果该界面为当前页，则调用 setUserVisibleHint(true) ，否则调用 setUserVisibleHint(false) 。
> 3. 界面变为可见时，调用setUserVisibleHint(true)。界面变为不可见时，调用setUserVisibleHint(false)。

42、懒加载Fragment的实现

> 1、网络请求不涉及UI更新时直接重写 Fragment 的 setUserVisibleHint() 方法

```
public void setUserVisibleHint(boolean isVisibleToUser) {
    Log.d("TAG", mTagName + " setUserVisibleHint() --> isVisibleToUser = " + isVisibleToUser);

    if (isVisibleToUser) {
        pullData();
    }
    super.setUserVisibleHint(isVisibleToUser);
}
```

> 2、如果耗时请求涉及UI更新(请求好数据后结果UI还没有绑定好)，这种情况适合在 onStart() 通过 getUserVisibleHint() 判断.

```
@Override
public void onStart() {
    super.onStart();
    Log.d("TAG", mTagName + " onStart()");

    ...

    if(getUserVisibleHint()) {
        pullData();
    }

}
```

FragmentPagerAdapter

43、FragmentPagerAdapter的特点

> 1. Item销毁时，只是将其detach()和界面分离
> 2. 适合Fragment较少的情况，更占内存，但是反应速度会更快
> 3. destroyItem()源码

```
public void destroyItem(@NonNull ViewGroup container, int position, @NonNull Object object) {
        //...
        this.mCurTransaction.detach((Fragment)object);
}
```

FragmentStatePagerAdapter

44、FragmentStatePagerAdapter的特点

> 1. Item销毁时，会直接remove。
> 2. 不占内存，适合大量Fragment的情况。
> 3. destroyItem()源码

```
public void destroyItem(@NonNull ViewGroup container, int position, @NonNull Object object) {
        //...
        this.mCurTransaction.remove(fragment);
}
```

Fragment进阶

Fragment源码分析

45、getFragmentManager()源码和逻辑分析

```
//Activty.java
FragmentController mFragments = FragmentController.createController(new HostCallbacks());
public FragmentManager getFragmentManager() {
    //1. 调用FragmentController的方法
    return mFragments.getFragmentManager();
}
//FragmentController.java
public static final FragmentController createController(FragmentHostCallback<?> callbacks) {
    //2. 用callback(也就是HostCallbacks)创建对象
    return new FragmentController(callbacks);
}
//FragmentController.java
public FragmentManager getFragmentManager() {
    //3. 调用callback的方法
    return mHost.getFragmentManagerImpl();
}
//FragmentHostCallback.java-HostCallBacks的父类
final FragmentManagerImpl mFragmentManager = new FragmentManagerImpl();
FragmentManagerImpl getFragmentManagerImpl() {
    //4. 返回FragmentManagerImpl()
    return mFragmentManager;
}
```

1. FragmentManager 是一个 抽象类 ， 最终实现类是 FragmentManagerImpl

46、beginTransaction()源码解析

```
//FragmentManager.java的内部类FragmentManagerImpl的方法:
public FragmentTransaction beginTransaction() {
    //1. 返回BackStackRecord---保存了全部操作轨迹
    return new BackStackRecord(this);
}
```

47、 BackStackRecord 类是什么?

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

    //1. Op用于表示`某个操作`
    static final class Op {
        int cmd; //记录操作：add()或者remove()或者replace()等等
        Fragment fragment; //操作的Fragment对象
        int enterAnim;
        int exitAnim;
        int popEnterAnim;
        int popExitAnim;

        Op() {
        }

        Op(int cmd, Fragment fragment) {
            this.cmd = cmd;
            this.fragment = fragment;
        }
    }
    //2. BackStackRecord内部拥有Op类的链表
    ArrayList<Op> mOps = new ArrayList<>();
}
```

1. 继承FragmentTransaction(事务) ---保存了整个事务的 全部操作轨迹
2. 实现BackStackEntry ---Fragment回退栈中的实体，因此在 popBackStack() 时能回退整个事务。
3. 实现OpGenerator ---用于 UI线程 调度一个 add或pop 事务。
4. 内部拥有表示 add 等操作的 Op 类的链表。

48、add()源码解析

```
//BackStackRecord.java
public FragmentTransaction add(int containerViewId, Fragment fragment) {
    //1. OP_ADD作为参数传入
    doAddOp(containerViewId, fragment, null, OP_ADD);
    return this;
}
private void doAddOp(int containerViewId, Fragment fragment, String tag, int opcmd) {
    ...
    //2. 保存`容器ViewId`等信息到Fragment中
    fragment.mFragmentManager = mManager;
    fragment.mTag = tag;
    fragment.mContainerId = fragment.mFragmentId = containerViewId;
    ...
    //3. 创建Op并保存到链表中
    addOp(new Op(opcmd, fragment));
}
void addOp(Op op) {
    //3. 添加到链表中
    mOps.add(op);
    //4. Fragment的相关动画
    op.enterAnim = mEnterAnim;
    op.exitAnim = mExitAnim;
    op.popEnterAnim = mPopEnterAnim;
    op.popExitAnim = mPopExitAnim;
}
```

49、addToBackStack(“”)源码

```
//BackStackRecord.java
public FragmentTransaction addToBackStack(String name) {
    ...
    //1. 设置为true，该变量在cmmoit中被使用
    mAddToBackStack = true;
    mName = name;
    return this;
}
```

50、commit()源码解析

```
/**==============================*
 * 思路流程:
 *    1. 将“事务”添加到“回退栈”中
 *    2. 将“事务”添加到队列中
 *    3. 发起异步调度commit操作
 * //BackStackRecord.java
 *==============================*/
public int commit() {
    return commitInternal(false);
}

//BackStackRecord.java
int commitInternal(boolean allowStateLoss) {
    //1. 提交过会导致异常
    if (mCommitted) {
        throw new IllegalStateException("commit already called");
    }
    ...
    mCommitted = true;
    //2. `addToBackStack`设置的变量, 为true时将当前`事务`加入到`回退栈`中
    if (mAddToBackStack) {
        mIndex = mManager.allocBackStackIndex(this);
    } else {
        mIndex = -1;
    }
    //3. 将该`事务`加入到队列中
    mManager.enqueueAction(this, allowStateLoss);
    return mIndex;
}

//BackStackRecord.java
public int allocBackStackIndex(BackStackRecord bse) {
    synchronized (this) {
        ...
        //1. 将"事务"加入到"回退栈中"
        int index = mBackStackIndices.size();
        mBackStackIndices.add(bse);
        return index;
        ...
    }
}

//FragmentManager.java: 将`action`加入到 待定操作 的队列中
public void enqueueAction(OpGenerator action, boolean allowStateLoss) {
    synchronized (this) {
        ...
        if (mPendingActions == null) {
            mPendingActions = new ArrayList<>();
        }
        //1. 加入到待定操作的链表中
        mPendingActions.add(action);
        //2. 异步调度commit操作
        scheduleCommit();
    }
```

```
}

//FragmentManager.java
private void scheduleCommit() {
    synchronized (this) {
        //1. 准备"延期缓办"---延期事务不为null 且 延期事务不为空
        boolean postponeReady =
                mPostponedTransactions != null && !mPostponedTransactions.isEmpty();
        //2. 准备"即将发生"---待定操作不为null 且 待定操作数量为1
        boolean pendingReady = mPendingActions != null && mPendingActions.size() == 1;
        //3. 两者满足其中之一, 主线程中执行“待定操作”
        if (postponeReady || pendingReady) {
            mHost.getHandler().removeCallbacks(mExecCommit);
            mHost.getHandler().post(mExecCommit);
        }
    }
}

/**==============================
 * 异步调度commit流程:
 *   1. 遍历所有需要执行的事务
 *   2. 最终执行BackStackRecord的executeOps()执行所有操作
 * //FragmentManager.java
 *==============================*/
Runnable mExecCommit = new Runnable() {
    @Override
    public void run() {
        //1. 执行“待定操作”
        execPendingActions();
    }
};

//FragmentManager.java: 只能在“主线程”中调用
public boolean execPendingActions() {
    ...
    while (generateOpsForPendingActions(mTmpRecords, mTmpIsPop)) {
        //1. 将pendingActions中所有积压的"事务"全部执行
        removeRedundantOperationsAndExecute(mTmpRecords, mTmpIsPop);
    }
    ...
}
//FragmentManager.java
private void removeRedundantOperationsAndExecute(ArrayList<BackStackRecord> records, ArrayList<
    ...
    executePostponedTransaction(records, isRecordPop);
    ...
    //1. 执行"事务列表"中范围(startIndex~endIndex)的所有事务
    executeOpsTogether(records, isRecordPop, startIndex, endIndex);
    ...
}

//FragmentManager.java
private void executeOpsTogether(ArrayList<BackStackRecord> records, ArrayList<Boolean> isRecord
    //1. 执行"事务列表"中所有事务(范围为startIndex~endIndex)
    executeOps(records, isRecordPop, startIndex, endIndex);
```

```
    ...
}
//FragmentManager.java
private static void executeOps(ArrayList<BackStackRecord> records, ArrayList<Boolean> isRecordF
    //1. 遍历"事务"
    for (int i = startIndex; i < endIndex; i++) {
        final BackStackRecord record = records.get(i);
        final boolean isPop = isRecordPop.get(i);
        if (isPop) {
            record.bumpBackStackNesting(-1);
            //2. 执行Pop操作: 仅仅在所有事务最后执行add操作
            boolean moveToState = i == (endIndex - 1);
            record.executePopOps(moveToState);
        } else {
            //3. 执行操作
            record.bumpBackStackNesting(1);
            record.executeOps();
        }
    }
}
//BackStackRecord.java
void executeOps() {
    /**==============================================
     * 遍历事务的全部操作:
     *  1.add  2.remove 3.hide 4.show 5.detach 6.attach
     *==============================================*/
    final int numOps = mOps.size();
    for (int opNum = 0; opNum < numOps; opNum++) {
        final Op op = mOps.get(opNum);
        final Fragment f = op.fragment;
        if (f != null) {
            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;
            case OP_SET_PRIMARY_NAV:
                mManager.setPrimaryNavigationFragment(f);
                break;
            case OP_UNSET_PRIMARY_NAV:
                mManager.setPrimaryNavigationFragment(null);
                break;
            default:
                throw new IllegalArgumentException("Unknown cmd: " + op.cmd);
        }
        if (!mReorderingAllowed && op.cmd != OP_ADD && f != null) {
            mManager.moveFragmentToExpectedState(f);
        }
    }
    if (!mReorderingAllowed) {
        // Added fragments are added at the end to comply with prior behavior.
        mManager.moveToState(mManager.mCurState, true);
    }
}
```

51、Fragment的7种状态

```
static final int INVALID_STATE = -1;   // 作为null值的非法状态
static final int INITIALIZING = 0;     // 没有被create
static final int CREATED = 1;          // 已经create
static final int ACTIVITY_CREATED = 2; // Activity已经完成了create
static final int STOPPED = 3;          // 完全创建，还没start
static final int STARTED = 4;          // 已经create和start，还没有resume
static final int RESUMED = 5;          // 已经完成create,start和resume
```

> 1. Fragment的生命周期 就是 Fragment状态切换 的过程
> 2. 若Fragment的 当前状态 小于 新状态 ， 就会进行 创建 、 唤醒 等过程
> 3. 若Fragment的 当前状态 大于 新状态 ， 就会进行 暂停 、 彻底销毁 等过程

52、FragmentManager的addFragment()源码解析

```
//BackStackRecord.java
void executeOps() {
    //1. 通过"FragmentManager"的addFragment()将Fragment添加到"待添加的列表"中
    switch (op.cmd) {
        case OP_ADD:
            f.setNextAnim(op.enterAnim);
            mManager.addFragment(f, false);
            break;
    }
    ...
    //2. 执行FragmentManager的方法，对Fragment进行状态切换
    mManager.moveToState(mManager.mCurState, true);
}

//FragmentManager.java
public void addFragment(Fragment fragment, boolean moveToStateNow) {
    //1. Fragment放置到"活跃的"Fragment列表中
    makeActive(fragment);
    //2. 将Fragment添加到待添加的Fragment列表(mAdded)中
    synchronized (mAdded) {
        mAdded.add(fragment);
    }
    ...
}

//FragmentManager.java
void moveToState(int newState, boolean always) {
    //1. FragmentManager的当前状态
    mCurState = newState;

    if (mActive != null) {
        //2. 遍历取出"待添加的Fragment"--执行onAttach()~onResume()的全部生命周期
        if (mAdded != null) {
            final int numAdded = mAdded.size();
            for (int i = 0; i < numAdded; i++) {
                Fragment f = mAdded.get(i);
                moveFragmentToExpectedState(f); //将Fragment跳转到预期状态
                ...
            }
        }
        //3. 遍历取出"活跃列表"中的Fragment -根据Fragment的状态决定是唤醒、终止还是彻底销毁
        final int numActive = mActive.size();
        for (int i = 0; i < numActive; i++) {
            Fragment f = mActive.valueAt(i);
            moveFragmentToExpectedState(f); //将Fragment跳转到预期状态
            ...
        }
        ...
    }
}

//FragmentManager.java
void moveFragmentToExpectedState(final Fragment f) {
    //1. 根据Fragment的旧状态和新状态，进行操作，最终Fragment达到预期状态
```

```
    moveToState(f, nextState, f.getNextTransition(), f.getNextTransitionStyle(), false);
    //2. 此前Fragment已经完成onCreate()~onResume()所有周期, mView(Fragment视图)一定不为空
    if (f.mView != null) {
        ...
        //3. Fragment的动画
        Animator anim = loadAnimator(f, f.getNextTransition(), true, f.getNextTransitionStyle()
        anim.setTarget(f.mView);
        anim.start();
    }
    //3. 根据mHiddenChanged,完成Fragment的show和hide
    if (f.mHiddenChanged) {
        completeShowHideFragment(f);
    }
}

//FragmentManager.java
void moveToState(Fragment f, int newState, int transit, int transitionStyle, boolean keepActive
    /**==========================================*
     * 1. Fragment状态 < 新状态 :  表明Fragment需要去创建或者唤醒
     *==========================================*/
    if (f.mState <= newState) {
        ...
        /**================================
         *  switch中没有break，直接顺序执行
         *================================*/
        switch (f.mState) {
            /**======================================*
             *  2. Fragment初始化: onAttach、onCreate()
             *======================================*/
            case Fragment.INITIALIZING:
                if (newState > Fragment.INITIALIZING) {
                    //1. Fragment执行onAttach()方法
                    f.onAttach(mHost.getContext());
                    //2. 没有父Fragment: 包含Fragment的Activity执行onAttachFragment
                    if (f.mParentFragment == null) {
                        mHost.onAttachFragment(f);
                    } else {
                        //3. 有父Fragment: 父Fragment执行onAttachFragment
                        f.mParentFragment.onAttachFragment(f);
                    }
                    ...
                    //4. 执行Fragment的onCreate()方法
                    f.performCreate(f.mSavedFragmentState);
                }
                /**======================================*
                 *  3. Fragment创建: onCreateView()、onViewCreated()
                 *======================================*/
            case Fragment.CREATED:
                if (newState > Fragment.CREATED) {
                    //1. Fragment的onCreateView()-【创建了mView】
                    f.mView = f.performCreateView(f.performGetLayoutInflater(f.mSavedFragmentSt
                    //2. 将mView添加到父容器内, 根据mHidden隐藏
                    if (f.mView != null) {
                        f.mView.setSaveFromParentEnabled(false);
                        if (container != null) {
```

```
                        container.addView(f.mView);
                    }
                    if (f.mHidden) {
                        f.mView.setVisibility(View.GONE);
                    }
                    //3. Fragment的onViewCreated
                    f.onViewCreated(f.mView, f.mSavedFragmentState);
                    ...
                }
            }
            //4. Fragment的onActivityCreated()
            f.performActivityCreated(f.mSavedFragmentState);
            ...
        case Fragment.ACTIVITY_CREATED:
            ...
        case Fragment.STOPPED:
            //1. onStart()和分发
            f.performStart();
        case Fragment.STARTED:
            //2. onResume()和分发
            f.performResume();
    }
}
/**================================================*
 * 4. Fragment状态 > 新状态 : 表明Fragment需要停止或者彻底销毁
 *================================================*/
else if (f.mState > newState) {
    switch (f.mState) {
        case Fragment.RESUMED:
            //1. onPause()和分发
            f.performPause();
            dispatchOnFragmentPaused(f, false);
        case Fragment.STARTED:
            //2. onStop()和分发
            f.performStop();
            dispatchOnFragmentStopped(f, false);
        case Fragment.STOPPED:
        case Fragment.ACTIVITY_CREATED:
            if (newState < Fragment.ACTIVITY_CREATED) {
                //1. onDestoryView()和分发
                f.performDestroyView();
                dispatchOnFragmentViewDestroyed(f, false);
                //3. Fragment销毁动画
                anim = loadAnimator(f, transit, false, transitionStyle);
                anim.setTarget(f.mView);
                anim.start();
                //4. 移除Fragment视图
                f.mContainer.removeView(f.mView);
            }
        case Fragment.CREATED:
            if (newState < Fragment.CREATED) {
                //1. 执行Fragment的onDestory(), 并分发
                f.performDestroy();
                dispatchOnFragmentDestroyed(f, false);
                //2. 执行Fragment的onDetach(), 并分发
```

```
                    f.performDetach();
                    dispatchOnFragmentDetached(f, false);
                    //3. Fragment从"活跃列表"中移除
                    makeInactive(f);
                }
            }
        }
        ...
    }
```

实战场景

预加载首页的所有页面

1、 预加载首页的所有页面

```
// 预加载所有fragment，且只展示第3个Store的Fragment
loadMultipleFragment(R.id.fl_container, 2, tabsFragment);

// 加载多个Fragment
private void loadMultipleFragment(int containerId, int showPosition, ArrayList<SupportFragment>
    // 1. 获取到Fragment的事物
    FragmentTransaction fragmentTransaction = getChildFragmentManager().beginTransaction();
    // 2. add所有需要加载的fragment，并将除了需要展示的fragment都进行hide
    for (int i = 0; i < targetFragments.size(); i++)
    {
        SupportFragment fragment = targetFragments.get(i);
        fragmentTransaction.add(containerId, fragment);
        if(i != showPosition){
            fragmentTransaction.hide(fragment);

        }
    }
    fragmentTransaction.commit();
}
```

点击底部Button，加载不同的Fragment

2、点击底部Button，加载不同的Fragment

```
//如果已加载就用show hide方式切换，如果没加载就直接加载新Fragment
if (findStackFragment(tabsFragment.get(id).getClass(), getChildFragmentManager(), true) != null
{
    // 如果show和hide的Fragment不是同一个
    getChildFragmentManager().beginTransaction().show(tabsFragment.get(id)).hide(tabsFragment.g
}else{
    getChildFragmentManager().beginTransaction().add(tabsFragment.get(id))..hide(tabsFragment.g
}
```

> 判断目标Fragment是否已经加载

```
// 正常方式：利用Tag找到该Fragment，但是如果是fragment存在于FragmentPagerAdapter中，这种方式就不准
<T extends SupportFragment> T findStackFragment(Class<T> fragmentClass, FragmentManager fragmer
{
    Fragment fragment = null;
    if (isChild)
    {
        // 如果是 查找子Fragment,则有可能是在FragmentPagerAdapter/FragmentStatePagerAdapter中,这
        // 它们的Tag是以android:switcher开头,所以这里我们使用下面的方式
        List<Fragment> childFragmentList = fragmentManager.getFragments();
        if (childFragmentList == null)
            return null;

        for (int i = childFragmentList.size() - 1; i >= 0; i--)
        {
            Fragment childFragment = childFragmentList.get(i);
            if (childFragment instanceof SupportFragment
                && childFragment.getClass().getName().equals(fragmentClass.getName()))
            {
                fragment = childFragment;
                break;
            }
        }
    }
    else
    {
        fragment = fragmentManager.findFragmentByTag(fragmentClass.getName());
    }
    if (fragment == null)
    {
        return null;
    }
    return (T) fragment;
}
```

补充题

1、Fragment的常见问题，以及如何处理?

> 1. `getActivity()`空指针 ： 常见于进行异步操作的时候，此时如果 `Fragment`已经`onDetach()`,就会遇到。解决办法：在 `Fragment` 里面使用一个 全局变量`mActivity` ， 可能会导致内存泄露。但是比 崩溃 更好。
> 2. 视图重叠 ： 主要是因为 `Fragment` 的 `onCreate()` 中没有判断 `saveInstanceSate == null` ， 导致重复加载了同一个 `Fragment`

2、Fragment的切换方式有多少种?

> 1. add()、replace()、remove(): 会将Fragment进行移除(就是回收了实例)，每次都会新建一个实例。
> 2. hide()、show()
> 3. detach()、attach()

3、add和replace的区别

> 1. replace会将栈中的fragment全部remove，再add进行添加。
> 2. add是在原有基础上进行添加。

4、hide和show的作用

> 隐藏和显示：将Fragment的显示和隐藏，占一点内存，

5、detach和attach的作用

> 1. 不会回收Fragment，detach()会将Fragment中的View销毁掉。
> 2. attach()会重新构建View。
> 3. 极少使用

6、detach()和attach()为什么没啥用?

> 1. 并不能节约多少内存
> 2. 导致每次都会去重建这个View。

7、FragmentTransaction报错：commit already called

> 1. beginTransaction()和commit()要配套使用
> 2. 多次commit会出现该问题。

8、Fragment show之前需要已经被add到container中了

> 要注意重叠显示的问题

额外收获

- SparseArray：

> 当使用HashMap(K, V),如果K为整数类型时,使用SparseArray的效率更高

- 进度条动画库-Lottie:

> 进度条动画库:Lottie(https://github.com/airbnb/lottie-android)实现
> Lottie动画:(https://www.lottiefiles.com/)。
> 优点： 使用非常方便，只需要下载JSON动画文件，然后在XML中写入。

参考资料

1. Frgament基本使用方法
2. FragmentTransaction详解
3. Fragment详解
4. Android基础：Fragment看这篇就够了
5. ViewPager的使用
6. Android优化方案之--Fragment的懒加载实现