Navigation作为一系列Fragment的导航机制，我们在布局中使用FragmentContainerView来实现它。

FragmentContainerView作为Fragment的承载容器，它本身是一个FrameLayout，而它承载的第一个Fragment是设置的NavHostFragment，因为我们并没有显示它的需求，所以很明显它是一个站位Fragment。

我们在显示首个展示的Fragment的时候肯定会替换它，并调用Fragment的replace方法进行替换。

使用Navigation

通过id 定位到布局中设置的NavHostFragment：

NavHostFragment fragment = (NavHostFragment) getSupportFragmentManager()

.findFragmentById();

获取NavHostFragment的导航控制器：

NavController controller = fragment.getNavController();

通过NavController 导航到指定id的Fragment：

controller.navigate()

可以看到导航功能使用的主要是NavHostFragment类中的NavController 。我们看看NavHostFragment是怎么初始化它的。

加载首个展示的Fragment：

在NavHostFragment的onInflate()初始化方法中，会获取我们在布局中定义的NavHostFragment的属性，包括navGraph和defaultNavHost属性。navGraph属性用于初始化graphId字段，它用于解析我们定义的Fragment之间的跳转规则，它是在xml文件中定义的；defaultNavHost用于初始化mDefaultNavHost字段，它代表是否要拦截系统的返回键。

获取到graphId字段之后，我们通过Resources进行xml文件的解析，又是IO！！！通过文件的解析，将里面定义的fragment标签一个个转换成NavDestination对象，这些对象的管理者就是NavGraph。也就是我们定义的xml文件最后会被抽象成内存中的NavGraph对象。解析完成后通过FragmentNavigator#navigate跳转到第一个Fragment。通过反射的方式加载对应的Fragment类。然后通过FragmentManager通过replace方法替换掉NavHostFragment。

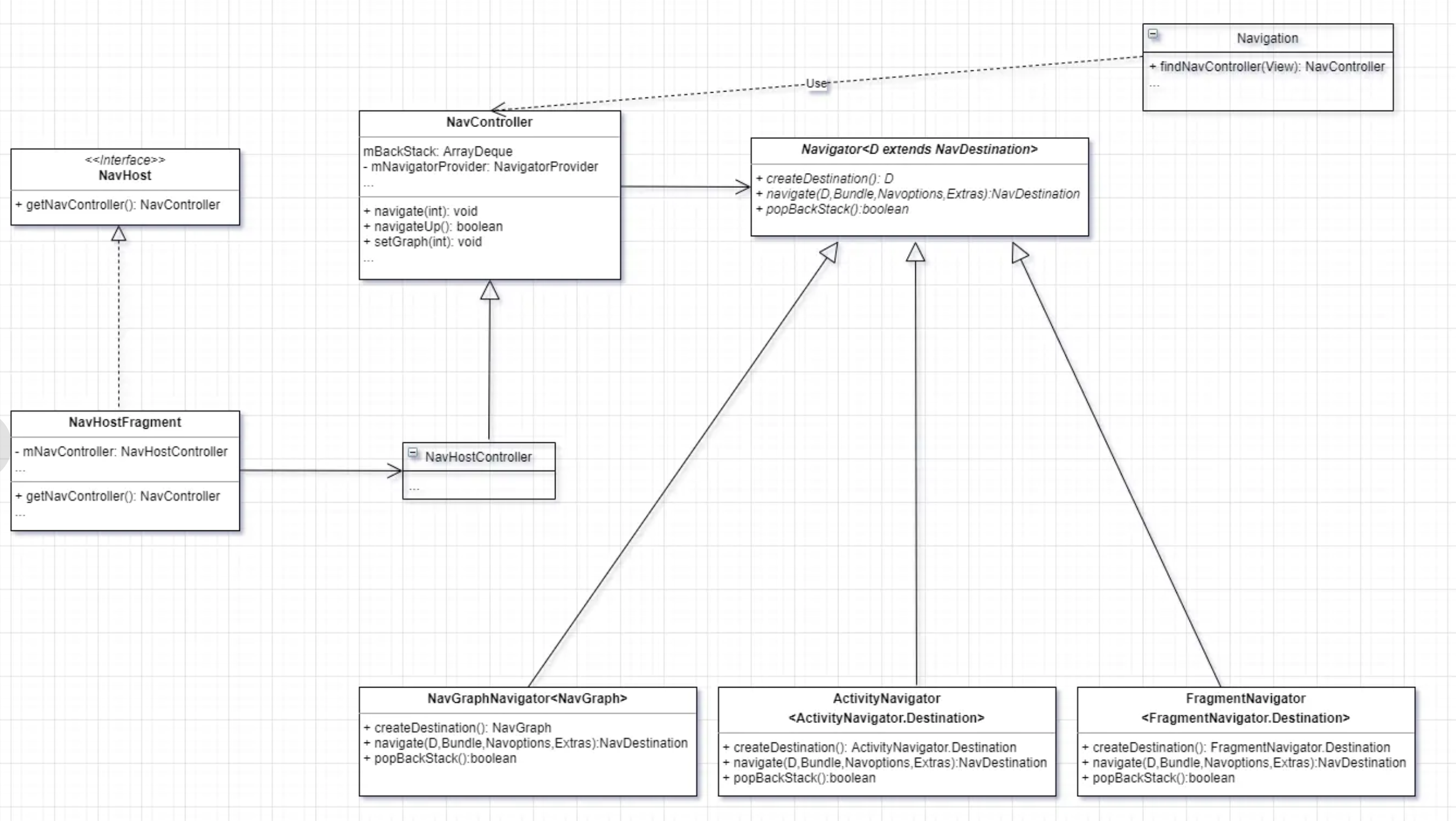
在NavHostFragment的onCreate()生命周期方法中初始化了NavHostController实例，它继承自NavController 。在实例中添加了四个Navigator，Navigator是用来控制界面导航的基类。在这里添加了ActivityNavigator、NavGraphNavigator、DialogFragmentNavigator、FragmentNavigator。

跳转到指定Fragment：

跳转的动作是通过NavController对象实现的。跳转之前获取当前Fragment的NavDestination对象，传入指定的跳转动作的id，获取跳转动作的封装类NavAction，里面包含了跳转目标Fragment的id。我们就可以通过这个id去寻找NavGraph中对应的Fragment的封装类NavDestination。

获取指定NavController：

在NavHostFragment初始化的时候，由于它的容器是一个FrameLayout，在初始化的时候，会将NavController设置给FrameLayout的tag。通过tag进行获取NavController。



从类的UML图可以看出，Navigation框架使用了设计模式中的策略模式，Navigator定义了一个导航规则，具体怎么导航由子类来实现。这里的子类就是图中的ActivityNavigator、FragmentNavigator等。

自定义FragmentNavigator：

原因：在Fragment切换时，由于用的是replace方法，所以再次回到某个Fragment界面时又会执行onCreateView方法导致界面重绘。

Fragment的加载都是在FragmentNavigator#navigate方法内执行，所以重写这个方法相对来说改动量是最小的。

先将当前正在显示的Fragment隐藏，因为每一次新的Fragment都会调用setPrimaryNavigationFragment这个方法，我们可以通过getPrimaryNavigationFragment获取当前现实的Fragment，只不过不是在FragmentTransaction对象上，而是在mFragmentManager对象上。

根据tag查找这个目标Fragment是否有加载过，有就直接show，没有就反射创建。

mBackStack是一个ArrayDeque类型的对象，用来记录Fragment的回退栈。它在父类中是私有的，只能通过反射来拿这个对象。

@Navigator.Name("customFragment")

public class CustomFragmentNavigator extends FragmentNavigator {

private static final String TAG = "CustomFragmentNavigator";

@NonNull

private final Context mContext;

@NonNull

private final FragmentManager mFragmentManager;

private final int mContainerId;

public CustomFragmentNavigator(@NonNull Context context, @NonNull FragmentManager manager, int containerId) {

super(context, manager, containerId);

mContext = context;

mFragmentManager = manager;

mContainerId = containerId;

}

@Nullable

@Override

public NavDestination navigate(@NonNull Destination destination, @Nullable Bundle args, @Nullable NavOptions navOptions, @Nullable Navigator.Extras navigatorExtras) {

if (mFragmentManager.isStateSaved()) {

Log.i(TAG, "Ignoring navigate() call: FragmentManager has already"

+ " saved its state");

return null;

}

String className = destination.getClassName();

if (className.charAt(0) == '.') {

className = mContext.getPackageName() + className;

}

// final Fragment frag = instantiateFragment(mContext, mFragmentManager,

// className, args);

// frag.setArguments(args);

final FragmentTransaction ft = mFragmentManager.beginTransaction();

// 先把当前正在显示的Fragment隐藏

Fragment current = mFragmentManager.getPrimaryNavigationFragment();

if (current != null) {

Log.i(TAG, "当前需要隐藏的fragment: " + current.getClass().getCanonicalName());

ft.hide(current);

}

int enterAnim = navOptions != null ? navOptions.getEnterAnim() : -1;

int exitAnim = navOptions != null ? navOptions.getExitAnim() : -1;

int popEnterAnim = navOptions != null ? navOptions.getPopEnterAnim() : -1;

int popExitAnim = navOptions != null ? navOptions.getPopExitAnim() : -1;

if (enterAnim != -1 || exitAnim != -1 || popEnterAnim != -1 || popExitAnim != -1) {

enterAnim = enterAnim != -1 ? enterAnim : 0;

exitAnim = exitAnim != -1 ? exitAnim : 0;

popEnterAnim = popEnterAnim != -1 ? popEnterAnim : 0;

popExitAnim = popExitAnim != -1 ? popExitAnim : 0;

ft.setCustomAnimations(enterAnim, exitAnim, popEnterAnim, popExitAnim);

}

// 查找需要显示的Fragment是否加载过

Fragment frag = mFragmentManager.findFragmentByTag(className);

if (frag == null) {

// 说明从未加载过，那就需要通过反射创建对象

frag = instantiateFragment(mContext, mFragmentManager, className, args);

frag.setArguments(args);

ft.add(mContainerId, frag, className);

Log.d(TAG, "反射创建fragment: " + className);

} else {

ft.show(frag);

}

// ft.replace(mContainerId, frag);

ft.setPrimaryNavigationFragment(frag);

final @IdRes int destId = destination.getId();

ArrayDeque<Integer> mBackStack = null;

try {

Field mBackStackField = FragmentNavigator.class.getDeclaredField("mBackStack");

mBackStackField.setAccessible(true);

//noinspection unchecked

mBackStack = (ArrayDeque<Integer>) mBackStackField.get(this);

} catch (Exception e) {

Log.e(TAG, "反射获取mBackStack对象异常: " + e.getMessage());

}

final boolean initialNavigation = mBackStack.isEmpty();

// TODO Build first class singleTop behavior for fragments

final boolean isSingleTopReplacement = navOptions != null && !initialNavigation

&& navOptions.shouldLaunchSingleTop()

&& mBackStack.peekLast() == destId;

boolean isAdded;

if (initialNavigation) {

isAdded = true;

} else if (isSingleTopReplacement) {

// Single Top means we only want one instance on the back stack

if (mBackStack.size() > 1) {

// If the Fragment to be replaced is on the FragmentManager's

// back stack, a simple replace() isn't enough so we

// remove it from the back stack and put our replacement

// on the back stack in its place

mFragmentManager.popBackStack(

generateBackStackName(mBackStack.size(), mBackStack.peekLast()),

FragmentManager.POP\_BACK\_STACK\_INCLUSIVE);

ft.addToBackStack(generateBackStackName(mBackStack.size(), destId));

}

isAdded = false;

} else {

ft.addToBackStack(generateBackStackName(mBackStack.size() + 1, destId));

isAdded = true;

}

if (navigatorExtras instanceof Extras) {

Extras extras = (Extras) navigatorExtras;

for (Map.Entry<View, String> sharedElement : extras.getSharedElements().entrySet()) {

ft.addSharedElement(sharedElement.getKey(), sharedElement.getValue());

}

}

ft.setReorderingAllowed(true);

ft.commit();

// The commit succeeded, update our view of the world

if (isAdded) {

mBackStack.add(destId);

return destination;

} else {

return null;

}

}

private String generateBackStackName(int backStackIndex, int destId) {

return backStackIndex + "-" + destId;

}

}

类上面有一个注解，注解的值被用来作为key保存到NavigatorProvider中。

所有的Navigator都被保存在NavigatorProvider，可以通过NavController对象获取,NavController被保存在NavHostFragment来获取，直接用FragmentManager就可以得到NavHostFragment。

// 此处代码放在Activity的onCreate方法内

Fragment fragment = getSupportFragmentManager().findFragmentById(R.id.nav\_host\_fragment\_activity\_main);

assert fragment != null;

NavController navController = NavHostFragment.findNavController(fragment);

NavigatorProvider navigatorProvider = navController.getNavigatorProvider();

CustomFragmentNavigator customNavigator = new CustomFragmentNavigator(this, getSupportFragmentManager(), R.id.nav\_host\_fragment\_activity\_main);

navigatorProvider.addNavigator(customNavigator);

整个navigation目录下的xml文件都是由NavGraph来管理，里面的fragment标签都会被封装成一个个NavDestination。这部分加载是由NavGraphNavigator负责。所以我们要把写在xml文件的逻辑用代码实现。

NavGraph navGraph = new NavGraph(new NavGraphNavigator(navigatorProvider));

FragmentNavigator.Destination hfd = customNavigator.createDestination();

hfd.setId(R.id.id\_home\_fragment);

hfd.setClassName(HomeFragment.class.getCanonicalName());

hfd.setLabel(getResources().getString(R.string.title\_home));

navGraph.addDestination(hfd);

FragmentNavigator.Destination dfd = customNavigator.createDestination();

dfd.setId(R.id.id\_dashboard\_fragment);

dfd.setClassName(DashboardFragment.class.getCanonicalName());

dfd.setLabel(getResources().getString(R.string.title\_dashboard));

navGraph.addDestination(dfd);

FragmentNavigator.Destination nfd = customNavigator.createDestination();

nfd.setId(R.id.id\_notifications\_fragment);

nfd.setClassName(NotificationsFragment.class.getCanonicalName());

nfd.setLabel(getResources().getString(R.string.title\_notifications));

navGraph.addDestination(nfd);

// 设置首页

navGraph.setStartDestination(R.id.id\_home\_fragment);

// 加载我们代码生成的导航图

navController.setGraph(navGraph);

binding.navView.setOnNavigationItemSelectedListener(item -> {

// 监听tab的点击事件，同时变更与之对应的NavDestination

navController.navigate(item.getItemId());

return true;

});

布局文件中就不需要再去加载navigation目录下的文件了。

由于使用了自定义的Navigator，引入了一个新的问题，那就是回退栈的问题。从第一个tab依次点第二个、第三个，此时再按返回键，先返回到第二个tab再到第一个tab，怎么进来就怎么回去。而Navigation框架是直接回到第一个tab，再退出界面。但我觉得这也是一个坑，从产品角度来看，应该是直接退出界面，而不是回到第一个tab。那么重写Activity的onBackPressed方法可以达到与原框架一样的效果。

@Override

public void onBackPressed() {

int curId = navController.getCurrentDestination().getId();

int startDestination = navController.getGraph().getStartDestination();

if (curId != startDestination) {

binding.navView.setSelectedItemId(startDestination);

} else {

finish();

}

}