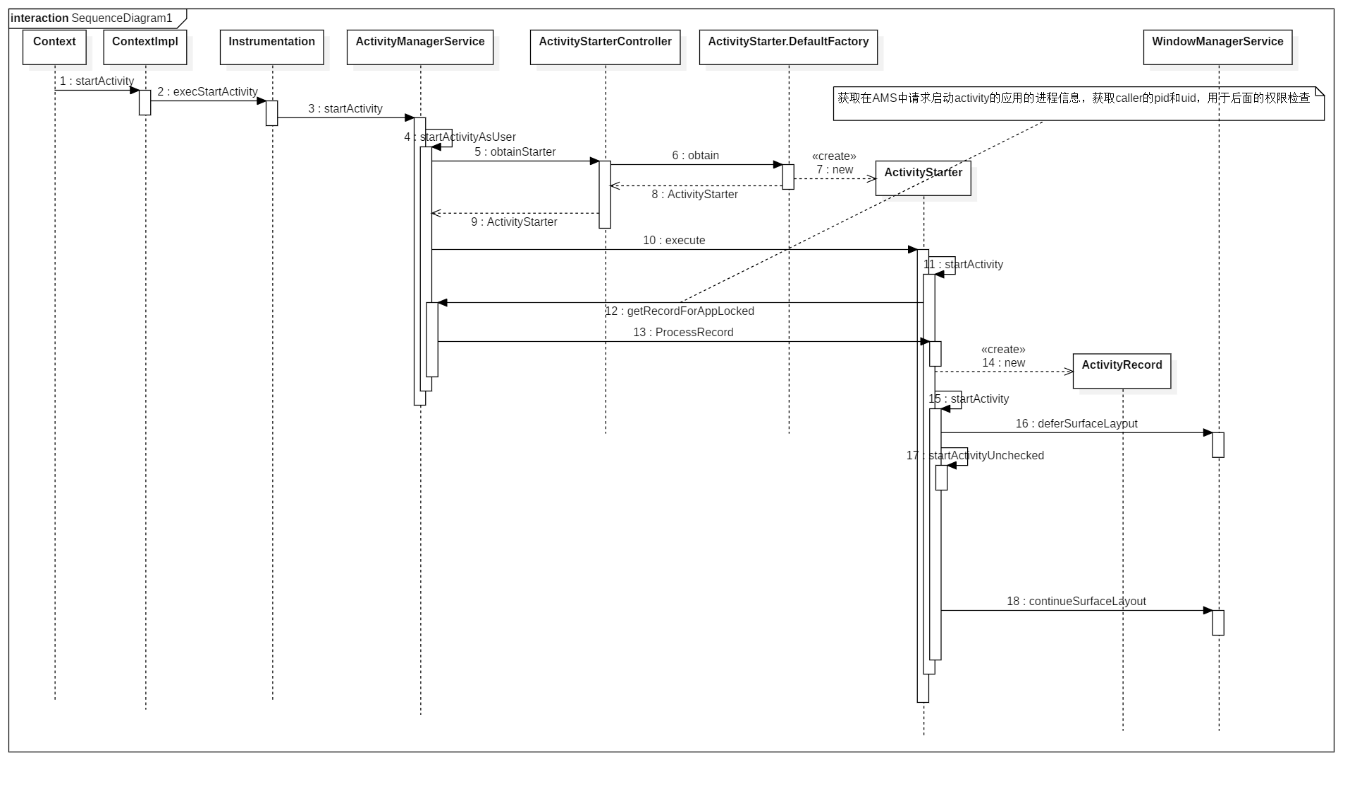
Android系统框架学习记录

# AMS学习

## StartActivity启动过程

以context触发的startActivity流程为切入点进行学习，context是应用进程内部的API接口，其触发startActivity后，将由WMS和AMS协助进行Activity的启动。



关注AMS中的几个关键类：

## ContextImpl中的startActivity：

public void startActivity(Intent intent) {

warnIfCallingFromSystemProcess();

startActivity(intent, null);

}

传入Intent时启动Activity时，实际上调用了：

public void startActivityAsUser(Intent intent, UserHandle user) {

startActivityAsUser(intent, null, user);

}

此时user传入了null；然后调用了startActivityAsUser

public void startActivityAsUser(Intent intent, Bundle options, UserHandle user) {

try {

ActivityManager.getService().startActivityAsUser(

mMainThread.getApplicationThread(), getBasePackageName(), intent,

intent.resolveTypeIfNeeded(getContentResolver()),

null, null, 0, Intent.FLAG\_ACTIVITY\_NEW\_TASK, null, options,

user.getIdentifier());

} catch (RemoteException e) {

throw e.rethrowFromSystemServer();

}

}

从ActivityManagerService中的startActivity函数开始走读：

public final int startActivityAsUser(IApplicationThread caller, String callingPackage,

Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,

int startFlags, ProfilerInfo profilerInfo, Bundle bOptions, int userId,

boolean validateIncomingUser) {

enforceNotIsolatedCaller("startActivity");

userId = mActivityStartController.checkTargetUser(userId, validateIncomingUser,

Binder.getCallingPid(), Binder.getCallingUid(), "startActivityAsUser");

// TODO: Switch to user app stacks here.

return mActivityStartController.obtainStarter(intent, "startActivityAsUser")

.setCaller(caller)

.setCallingPackage(callingPackage)

.setResolvedType(resolvedType)

.setResultTo(resultTo)

.setResultWho(resultWho)

.setRequestCode(requestCode)

.setStartFlags(startFlags)

.setProfilerInfo(profilerInfo)

.setActivityOptions(bOptions)

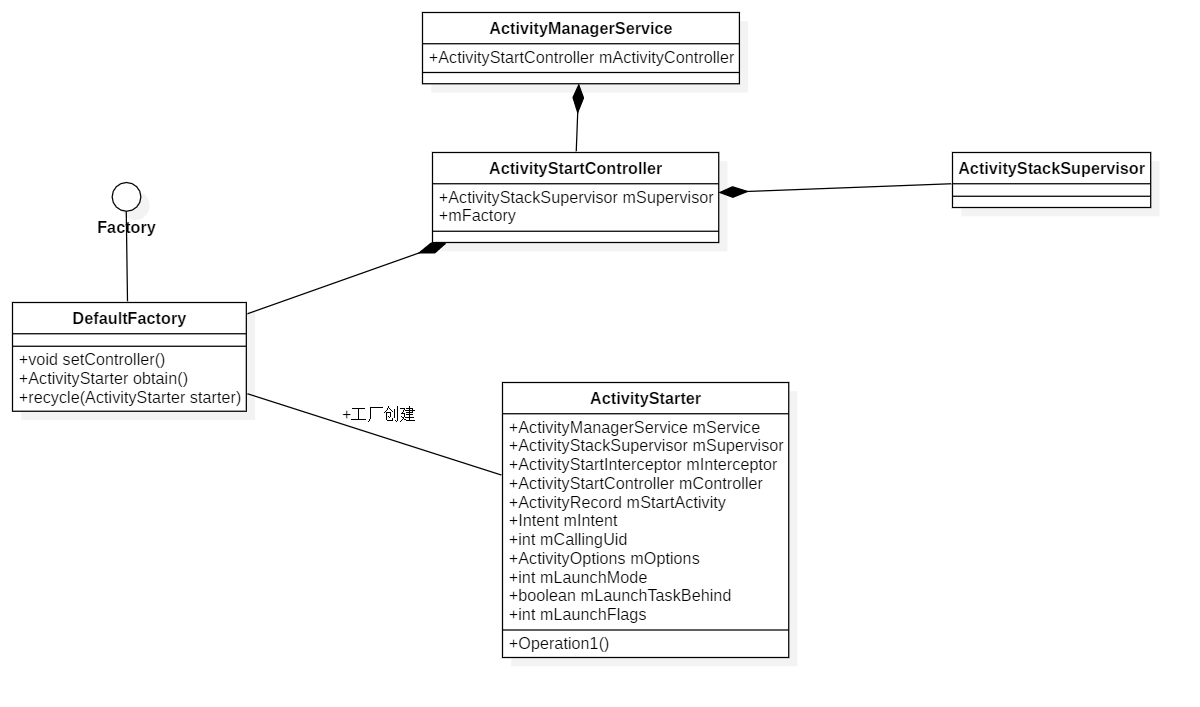
.setMayWait(userId)

.execute();

}

首先调用ActivityStartController的obtainStarter函数，获取对应的Starter，然后将starter的caller设置为调用者的信息，设置requestCode和startFlags等信息，最后调用execute函数开始执行。

## ActivityStartController是什么东西



ActivityStartController是AMS系统中的一个控制器，用于控制ActivityStarter的生成，当有应用通过AMS请求启动Activity时，AMS通过其中的ActivityStarter的工厂类，生成一个ActivityStarter返回出去，设置Activity启动的对应参数在ActivityStarter中，由AMS后续启动。

ActivityStarter中包含启动Activity的Intent，调用者的Uid，启动选项，启动模式，启动的标志位，对应所在的task，期望启动的display，是否需要启动到前台，是否需要包含动画等信息，等等

## ActivityStarter中startActivity

private int startActivityMayWait(IApplicationThread caller, int callingUid,

String callingPackage, Intent intent, String resolvedType,

IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,

IBinder resultTo, String resultWho, int requestCode, int startFlags,

ProfilerInfo profilerInfo, WaitResult outResult,

Configuration globalConfig, SafeActivityOptions options, boolean ignoreTargetSecurity,

int userId, TaskRecord inTask, String reason,

boolean allowPendingRemoteAnimationRegistryLookup,

PendingIntentRecord originatingPendingIntent) {

// Refuse possible leaked file descriptors

//启动activity时，intent中不能传递FileDescritor

if (intent != null && intent.hasFileDescriptors()) {

throw new IllegalArgumentException("File descriptors passed in Intent");

}

mSupervisor.getActivityMetricsLogger().notifyActivityLaunching();

//检查当前的intent中是否包含component的信息

boolean componentSpecified = intent.getComponent() != null;

//获取binder调用者的pid和uid的信息

final int realCallingPid = Binder.getCallingPid();

final int realCallingUid = Binder.getCallingUid();

//如果调用者的Uid>=0,

int callingPid;

if (callingUid >= 0) {

callingPid = -1;

} else if (caller == null) {

callingPid = realCallingPid;

callingUid = realCallingUid;

} else {

callingPid = callingUid = -1;

}

// Save a copy in case ephemeral needs it

final Intent ephemeralIntent = new Intent(intent);

// Don't modify the client's object!

intent = new Intent(intent);

//如果指定了componet，component指定的组件为instantApp安装器组件，将其清除

if (componentSpecified

&& !(Intent.ACTION\_VIEW.equals(intent.getAction()) && intent.getData() == null)

&& !Intent.ACTION\_INSTALL\_INSTANT\_APP\_PACKAGE.equals(intent.getAction())

&& !Intent.ACTION\_RESOLVE\_INSTANT\_APP\_PACKAGE.equals(intent.getAction())

&& mService.getPackageManagerInternalLocked()

.isInstantAppInstallerComponent(intent.getComponent())) {

// intercept intents targeted directly to the ephemeral installer the

// ephemeral installer should never be started with a raw Intent; instead

// adjust the intent so it looks like a "normal" instant app launch

intent.setComponent(null /\*component\*/);

componentSpecified = false;

}

//根据intent来resolveIntent，解析出ResolveInfo，resolveInfo通过系统中的PMS组件查询符合intent的ResolveInfo信息，解析出来的ResolveInfo，resolveInfo需要根据是否多用户场景进行处理

ResolveInfo rInfo = mSupervisor.resolveIntent(intent, resolvedType, userId,

0 /\* matchFlags \*/,

computeResolveFilterUid(

callingUid, realCallingUid, mRequest.filterCallingUid));

if (rInfo == null) {

UserInfo userInfo = mSupervisor.getUserInfo(userId);

if (userInfo != null && userInfo.isManagedProfile()) {

// Special case for managed profiles, if attempting to launch non-cryto aware

// app in a locked managed profile from an unlocked parent allow it to resolve

// as user will be sent via confirm credentials to unlock the profile.

UserManager userManager = UserManager.get(mService.mContext);

boolean profileLockedAndParentUnlockingOrUnlocked = false;

long token = Binder.clearCallingIdentity();

try {

UserInfo parent = userManager.getProfileParent(userId);

profileLockedAndParentUnlockingOrUnlocked = (parent != null)

&& userManager.isUserUnlockingOrUnlocked(parent.id)

&& !userManager.isUserUnlockingOrUnlocked(userId);

} finally {

Binder.restoreCallingIdentity(token);

}

if (profileLockedAndParentUnlockingOrUnlocked) {

rInfo = mSupervisor.resolveIntent(intent, resolvedType, userId,

PackageManager.MATCH\_DIRECT\_BOOT\_AWARE

| PackageManager.MATCH\_DIRECT\_BOOT\_UNAWARE,

computeResolveFilterUid(

callingUid, realCallingUid, mRequest.filterCallingUid));

}

}

}

//从resolveInfo中解析出ActivityInfo信息

// Collect information about the target of the Intent.

ActivityInfo aInfo = mSupervisor.resolveActivity(intent, rInfo, startFlags, profilerInfo);

synchronized (mService) {

//获取当前在Supervisor中的焦点Stack

final ActivityStack stack = mSupervisor.mFocusedStack;

stack.mConfigWillChange = globalConfig != null

&& mService.getGlobalConfiguration().diff(globalConfig) != 0;

if (DEBUG\_CONFIGURATION) Slog.v(TAG\_CONFIGURATION,

"Starting activity when config will change = " + stack.mConfigWillChange);

final long origId = Binder.clearCallingIdentity();

if (aInfo != null &&

(aInfo.applicationInfo.privateFlags

& ApplicationInfo.PRIVATE\_FLAG\_CANT\_SAVE\_STATE) != 0 &&

mService.mHasHeavyWeightFeature) {

// This may be a heavy-weight process! Check to see if we already

// have another, different heavy-weight process running.

//如果当前的ActivityInfo中的ApplicationInfo中指定了ApplicationInfo.PRIVATE\_FLAG\_CANT\_SAVE\_STATE，并且AMS支持HeavyWeight的特性，那么检查一下是否有重量级的进程存在运行

if (aInfo.processName.equals(aInfo.applicationInfo.packageName)) {

//如果当前系统中存在一个重量级的进程，并且重量级的进程和当前要启动的进程不一样（通过uid或者进程名判断）

final ProcessRecord heavy = mService.mHeavyWeightProcess;

if (heavy != null && (heavy.info.uid != aInfo.applicationInfo.uid

|| !heavy.processName.equals(aInfo.processName))) {

int appCallingUid = callingUid;

if (caller != null) {

ProcessRecord callerApp = mService.getRecordForAppLocked(caller);

if (callerApp != null) {

appCallingUid = callerApp.info.uid;

} else {

Slog.w(TAG, "Unable to find app for caller " + caller

+ " (pid=" + callingPid + ") when starting: "

+ intent.toString());

SafeActivityOptions.abort(options);

return ActivityManager.START\_PERMISSION\_DENIED;

}

}

IIntentSender target = mService.getIntentSenderLocked(

ActivityManager.INTENT\_SENDER\_ACTIVITY, "android",

appCallingUid, userId, null, null, 0, new Intent[] { intent },

new String[] { resolvedType }, PendingIntent.FLAG\_CANCEL\_CURRENT

| PendingIntent.FLAG\_ONE\_SHOT, null);

Intent newIntent = new Intent();

if (requestCode >= 0) {

// Caller is requesting a result.

newIntent.putExtra(HeavyWeightSwitcherActivity.KEY\_HAS\_RESULT, true);

}

newIntent.putExtra(HeavyWeightSwitcherActivity.KEY\_INTENT,

new IntentSender(target));

//设置新的intent，将当前的重量进程中的最顶端的Activity对应的packgeName和taskId设置到新的intent中，

if (heavy.activities.size() > 0) {

ActivityRecord hist = heavy.activities.get(0);

newIntent.putExtra(HeavyWeightSwitcherActivity.KEY\_CUR\_APP,

hist.packageName);

newIntent.putExtra(HeavyWeightSwitcherActivity.KEY\_CUR\_TASK,

hist.getTask().taskId);

}

//复制旧的inten中的所有flag，将重量进程的switcherActvity的切换信息放到intent中

newIntent.putExtra(HeavyWeightSwitcherActivity.KEY\_NEW\_APP,

aInfo.packageName);

newIntent.setFlags(intent.getFlags());

newIntent.setClassName("android",

HeavyWeightSwitcherActivity.class.getName());

//原有的intent使用新的intent替换

intent = newIntent;

resolvedType = null;

caller = null;

callingUid = Binder.getCallingUid();

callingPid = Binder.getCallingPid();

componentSpecified = true;

//重新使用新的intent来进行resolve info的信息，再从Resolve中获取到最新的ActivityInfo

rInfo = mSupervisor.resolveIntent(intent, null /\*resolvedType\*/, userId,

0 /\* matchFlags \*/, computeResolveFilterUid(

callingUid, realCallingUid, mRequest.filterCallingUid));

aInfo = rInfo != null ? rInfo.activityInfo : null;

if (aInfo != null) {

aInfo = mService.getActivityInfoForUser(aInfo, userId);

}

}

}

}

//新建了一个数组作为传出参数

final ActivityRecord[] outRecord = new ActivityRecord[1];

//再次调用startActivity函数，重载的startActivity，这次调用的参数中包含了caller，调用者的进程信息，aInfo:启动的Activity的Info，rInfo，resolve的Info信息，其他的callingPid，callingUid，callingPackage等信息。

int res = startActivity(caller, intent, ephemeralIntent, resolvedType, aInfo, rInfo,

voiceSession, voiceInteractor, resultTo, resultWho, requestCode, callingPid,

callingUid, callingPackage, realCallingPid, realCallingUid, startFlags, options,

ignoreTargetSecurity, componentSpecified, outRecord, inTask, reason,

allowPendingRemoteAnimationRegistryLookup, originatingPendingIntent);

//如果stack的config发生变化，则触发AMS进行updateConfiguration

if (stack.mConfigWillChange) {

// If the caller also wants to switch to a new configuration,

// do so now. This allows a clean switch, as we are waiting

// for the current activity to pause (so we will not destroy

// it), and have not yet started the next activity.

mService.enforceCallingPermission(android.Manifest.permission.CHANGE\_CONFIGURATION,

"updateConfiguration()");

stack.mConfigWillChange = false;

if (DEBUG\_CONFIGURATION) Slog.v(TAG\_CONFIGURATION,

"Updating to new configuration after starting activity.");

mService.updateConfigurationLocked(globalConfig, null, false);

}

// Notify ActivityMetricsLogger that the activity has launched. ActivityMetricsLogger

// will then wait for the windows to be drawn and populate WaitResult.

//打log信息的逻辑，记录Activity被启动起来了

mSupervisor.getActivityMetricsLogger().notifyActivityLaunched(res, outRecord[0]);

//如果指定来要输出结果

if (outResult != null) {

outResult.result = res;

final ActivityRecord r = outRecord[0];

//去除res的结果

switch(res) {

case START\_SUCCESS: {

//当前的状态为启动成功，把outResult加入到Supervisor的等待启动列表里面，直到状态被切换为START\_TASK\_TO\_FRONT或者超时

mSupervisor.mWaitingActivityLaunched.add(outResult);

do {

try {

mService.wait();

} catch (InterruptedException e) {

}

} while (outResult.result != START\_TASK\_TO\_FRONT

&& !outResult.timeout && outResult.who == null);

if (outResult.result == START\_TASK\_TO\_FRONT) {

res = START\_TASK\_TO\_FRONT;

}

break;

}

case START\_DELIVERED\_TO\_TOP: {

//当前的状态为START\_DELIVERED\_TO\_TOP，表明已经启动到前台

outResult.timeout = false;

outResult.who = r.realActivity;

outResult.totalTime = 0;

break;

}

case START\_TASK\_TO\_FRONT: {

// ActivityRecord may represent a different activity, but it should not be

// in the resumed state.

//当前的状态为START\_TASK\_TO\_FRONT，检查是否课件并且状态是RESUMED的状态，如果不是，那么需要持续等待Activity切换到前台可见，直到超时

if (r.nowVisible && r.isState(RESUMED)) {

outResult.timeout = false;

outResult.who = r.realActivity;

outResult.totalTime = 0;

} else {

final long startTimeMs = SystemClock.uptimeMillis();

mSupervisor.waitActivityVisible(r.realActivity, outResult, startTimeMs);

// Note: the timeout variable is not currently not ever set.

do {

try {

mService.wait();

} catch (InterruptedException e) {

}

} while (!outResult.timeout && outResult.who == null);

}

break;

}

}

}

return res;

在看下ActivityStarter中的第二个startActivity函数：

private int startActivity(final ActivityRecord r, ActivityRecord sourceRecord,

IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,

int startFlags, boolean doResume, ActivityOptions options, TaskRecord inTask,

ActivityRecord[] outActivity) {

int result = START\_CANCELED;

try {

//停止surface布局

mService.mWindowManager.deferSurfaceLayout();

result = startActivityUnchecked(r, sourceRecord, voiceSession, voiceInteractor,

startFlags, doResume, options, inTask, outActivity);

} finally {

// If we are not able to proceed, disassociate the activity from the task. Leaving an

// activity in an incomplete state can lead to issues, such as performing operations

// without a window container.

//获取mStartActivity的stack，如果startActivityUnchecked的结果是不成功的，那么需要出发stack将mStartActivity finish掉。mStartActivity就是需要新启动的Activity

final ActivityStack stack = mStartActivity.getStack();

if (!ActivityManager.isStartResultSuccessful(result) && stack != null) {

stack.finishActivityLocked(mStartActivity, RESULT\_CANCELED,

null /\* intentResultData \*/, "startActivity", true /\* oomAdj \*/);

}

//恢复surface布局

mService.mWindowManager.continueSurfaceLayout();

}

postStartActivityProcessing(r, result, mTargetStack);

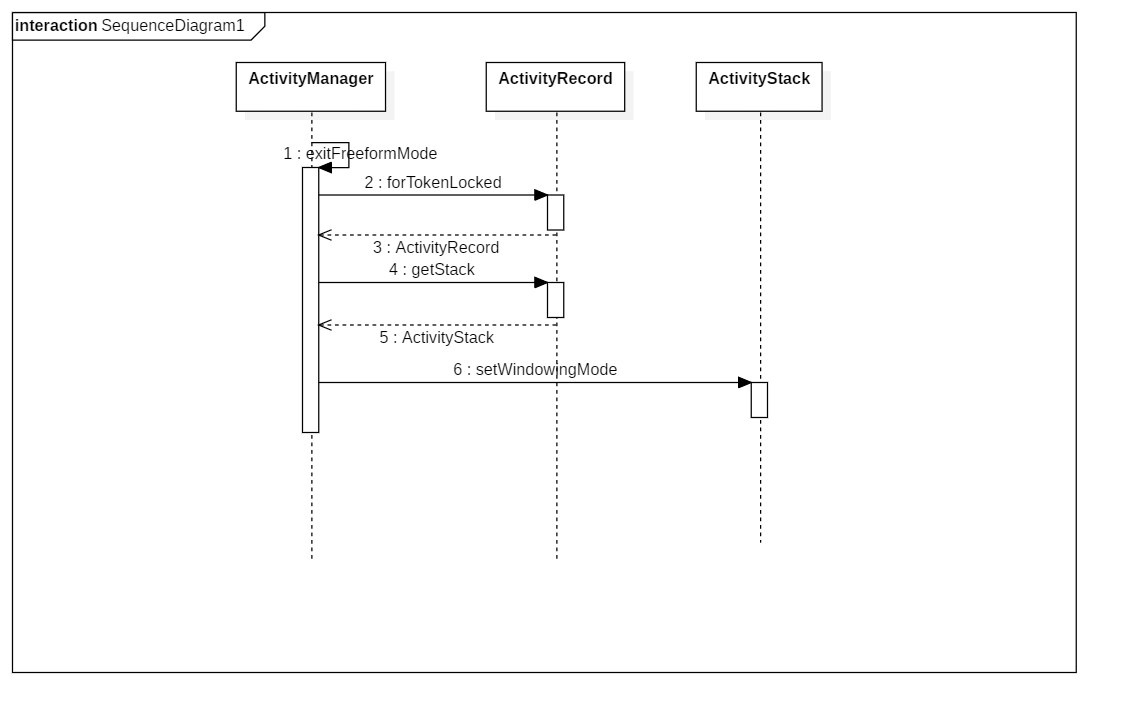
return result;

}

## AMS中的重量级进程

在AMS中有一个

## 退出freeform模式



AMS提供exitFreeformMode的接口，传入token后，通过token找到对应的ActivityRecord，再通过ActivityRecord获取到对应的stack，最后将stack的WindowMode设置为WINDOWING\_MODE\_FULLSCREEN即可退出freeform状态。

## ActivityStack中设置WindowMode的过程

首先要

1. 获取当前的stack对应的windowingMode

int currentMode = getWindowingMode()

1. 获取当前stack中处于topTask

TaskRecord topTask = topTask()

1. 获取当前的stack对应的display

ActivityDisplay display = getDisplay()

1. 判断当前是否是处于创建阶段

boolean creating = mWindowContainerController == null;

如果当前的WindowContainerController为空，则表明当前的stack处于创建过程中。

需要先检查当前的stack是否支持需要设定的WindowingMode,如果在创建过程中，那么设定的windowingMode是肯定支持的，否则需要通过Display的resolve判断是否支持，注意此时已经在mTmpOptions中设定了想要设定的windowingMode，

int windowingMode = creating

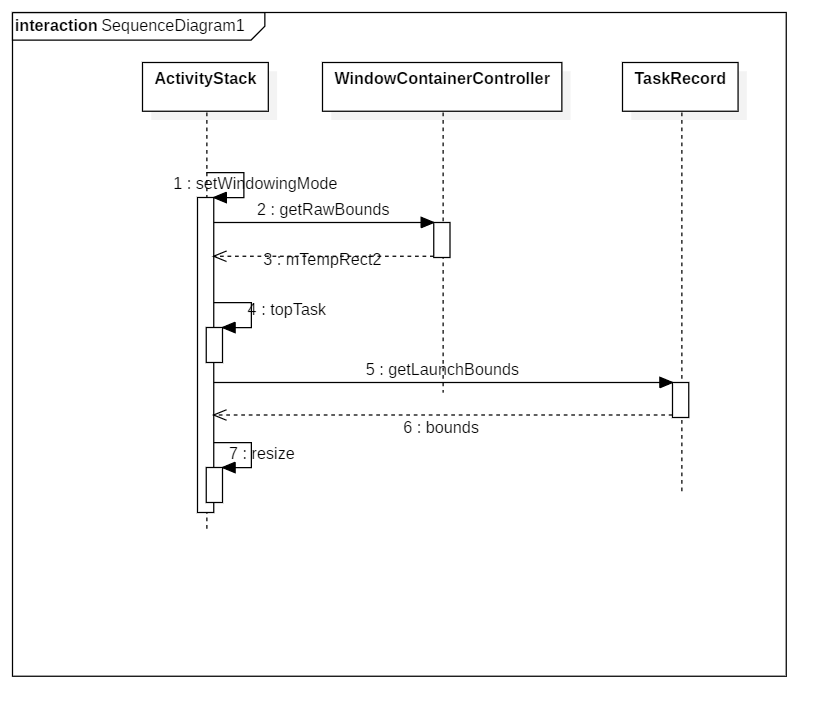
? preferredWindowingMode

: display.resolveWindowingMode(

null /\* ActivityRecord \*/, mTmpOptions, topTask, getActivityType());

ActivityDisplay解析支持的WindowingMode的过程如下，activitydisplay中判断windowingMode的方式，主要根据：

1. activityrecord是否支持分屏、是否支持resizeable
2. topTask是否支持分屏、是否支持resizeable



设置windowingMode时，首先通过WindowContaninerController获取到原始的bounds，设置为需要resize的bounds，然后获取当前Stack中的顶端的task，获取其对应的LaunchBounds，使用获取到的bounds来进行resize。

## ActivityStack的resize主要的逻辑

for (int i = mTaskHistory.size() - 1; i >= 0; i--) {

final TaskRecord task = mTaskHistory.get(i);

if (task.isResizeable()) {

if (inFreeformWindowingMode()) {

// TODO: Can be removed now since each freeform task is in its own stack.

// For freeform stack we don't adjust the size of the tasks to match that

// of the stack, but we do try to make sure the tasks are still contained

// with the bounds of the stack.

mTmpRect2.set(task.getOverrideBounds());

fitWithinBounds(mTmpRect2, bounds);

task.updateOverrideConfiguration(mTmpRect2);

} else {

task.updateOverrideConfiguration(taskBounds, insetBounds);

}

}

mTmpBounds.put(task.taskId, task.getOverrideBounds());

if (tempTaskInsetBounds != null) {

mTmpInsetBounds.put(task.taskId, tempTaskInsetBounds);

}

}

mWindowContainerController.resize(bounds, mTmpBounds, mTmpInsetBounds);

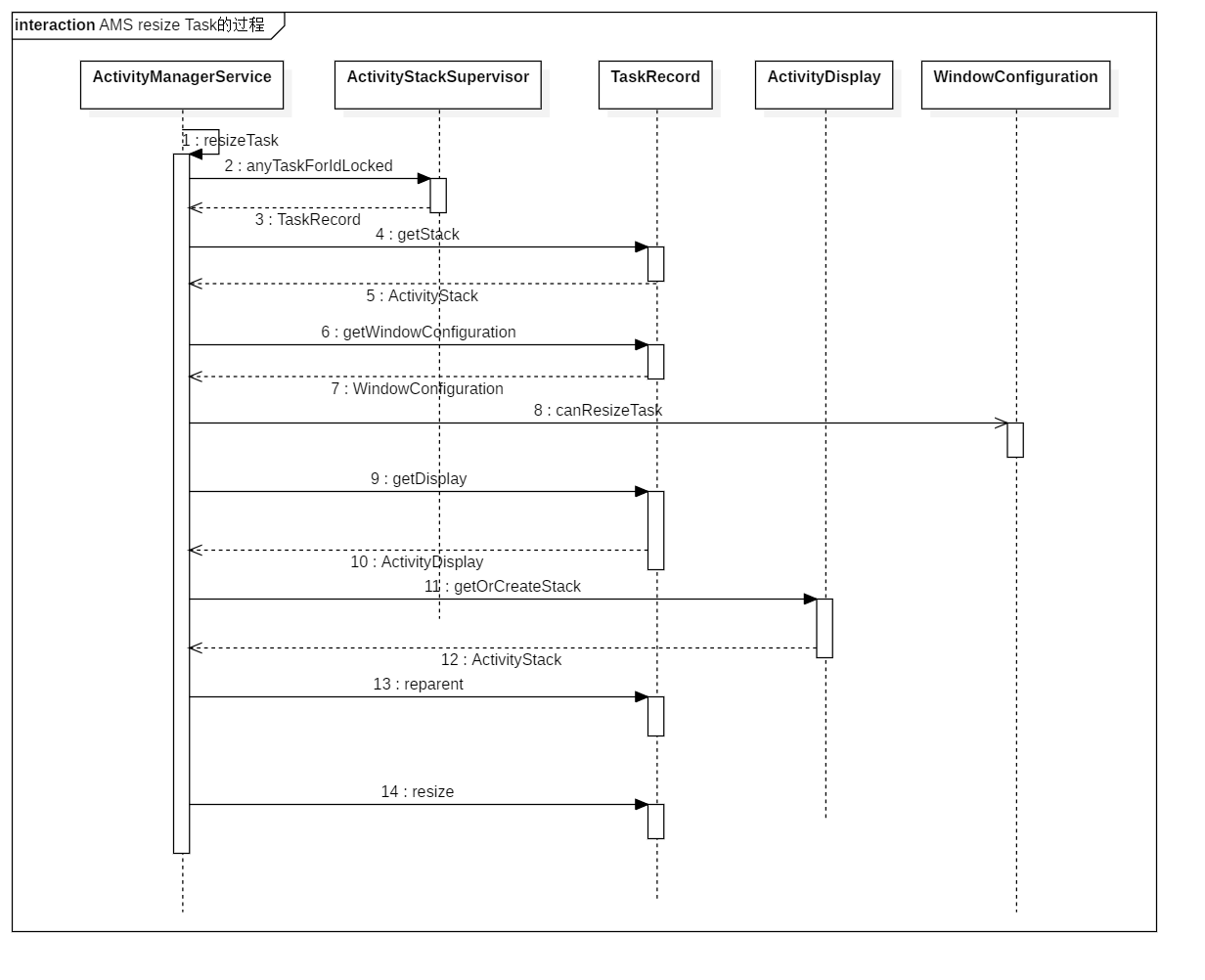
setBounds(bounds);

遍历当前stack中的task列表，如果当前的task是resizeable的，在freeform时将task的overridebounds

boolean creating = mWindowContainerController == null

## ActivityDisplay解析支持的WindowingMode的过程

## AMS resize Task的调用过程



public void resizeTask(int taskId, Rect bounds, int resizeMode)

AMS的resizeTask传入taskId，和bounds，可以将指定的task进行resize，bounds是需要resize的目标大小。

首先AMS通过ActivityStackSupervisor查询到taskId对应的task，然后在通过task获取到对应的task所在的stack栈，获取stack栈的目的是为了后面判断，是否需要进行reparent。

在resize之前，先通过task获取到对应的WindowConfiguration，通过WindowConfiguration判断当前的task是否支持resize，在当前的android平台，只有windowingMode为FreeForm的状态下，对应的task才可能resize。

接下来需要确定resize时的目标stack，需要判断当前的resize过程的stack是否需要变化：

1. 如果当前没有指定bounds(bounds == null),并且当前stack的WindowingMode是FreeForm模式。表明当前的resize，可能是需要切换到全屏态，在当前stack所在的ActivityDisplay获取FullScreen所在的栈。

stack = stack.getDisplay().getOrCreateStack(

WINDOWING\_MODE\_FULLSCREEN, stack.getActivityType(), ON\_TOP);

1. 如果当前指定了bounds，而且当前task所在的stack的WindowingMode不是FreeForm。表明需要切换到freeform状态，再在freeform状态下resize。在当前stack所在的ActivityDisplay获取FreeForm所在的栈。

stack = stack.getDisplay().getOrCreateStack(

WINDOWING\_MODE\_FREEFORM, stack.getActivityType(), ON\_TOP);

1. 其他场景则无需变化。比如当前已经在freeform状态下进行resize，那么无需再重新查找stack

如果当前的task所在的stack和查询到的目标stack不是同一个，那么需要将当前的task重新进行parent的操作，链到目标stack下：

// Reparent the task to the right stack if necessary

boolean preserveWindow = (resizeMode & RESIZE\_MODE\_PRESERVE\_WINDOW) != 0;

if (stack != task.getStack()) {

// Defer resume until the task is resized below

task.reparent(stack, ON\_TOP, REPARENT\_KEEP\_STACK\_AT\_FRONT, ANIMATE,

DEFER\_RESUME, "resizeTask");

preserveWindow = false;

}

最后在执行整个Task的resize过程：

// After reparenting (which only resizes the task to the stack bounds), resize the

// task to the actual bounds provided

task.resize(bounds, resizeMode, preserveWindow, !DEFER\_RESUME);

## TaskRecord的Reparent过程

## ActivityStackSupervisor中reparent时获取task对应的targetStack的过程

TaskRecord需要reparent其所在的stack时，首先要通过ActivityStackSupervisor的getReparentTargetStack来对target的stack进行检查。

1. 如果想要reparent的stack和当前的task所在的stack相同的（可通过stackId判断），那么没有必要进行改变，直接返回当前task所在的stack

// Check that we aren't reparenting to the same stack that the task is already in

if (prevStack != null && prevStack.mStackId == stackId) {

Slog.w(TAG, "Can not reparent to same stack, task=" + task

+ " already in stackId=" + stackId);

return prevStack;

}

1. 如果想要reparent的stack时多窗口模式，但是AMS不支持多窗口，会直接抛出异常

if (inMultiWindowMode && !mService.mSupportsMultiWindow) {

throw new IllegalArgumentException("Device doesn't support multi-window, can not"

+ " reparent task=" + task + " to stack=" + stack);

}

1. 如果想要reparent的stack对应的display不是默认的display，但是AMS不支持多display，则抛出异常

if (stack.mDisplayId != DEFAULT\_DISPLAY && !mService.mSupportsMultiDisplay) {

throw new IllegalArgumentException("Device doesn't support multi-display, can not"

+ " reparent task=" + task + " to stackId=" + stackId);

}

1. 如果想要reparent的stack是freeform的状态，但是AMS不支持freeform，则抛出异常

if (stack.getWindowingMode() == WINDOWING\_MODE\_FREEFORM

&& !mService.mSupportsFreeformWindowManagement) {

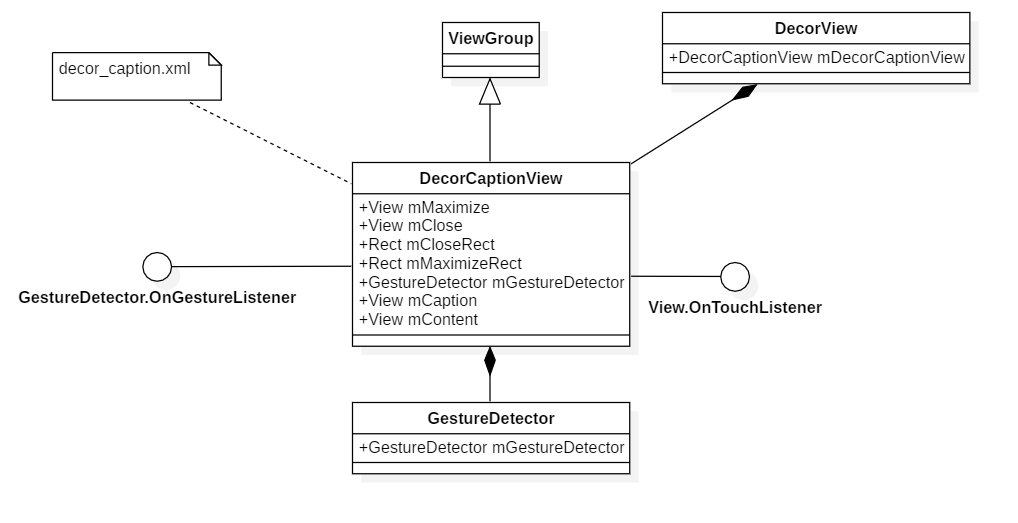
throw new IllegalArgumentException("Device doesn't support freeform, can not reparent"

+ " task=" + task);

}

1. 如果当前的窗口不是resizeable的，但是想要reparent的stack时多窗口模式，那么重新创建一个新全屏的stack返回

## Freeform的标题栏

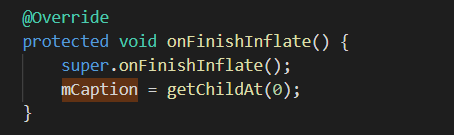


FreeForm窗口显示时，窗口标题栏上会有一个title栏，标题栏通过DecorCaptionView实现，DecorCaptionView本身是一个ViewGroup，其通过inflate décor\_caption.xml布局文件实例化，在布局文件中，布局文件是一个高为32dp的长条。主要就是定义了两个按钮：最大化和关闭按钮，最大化按钮将把整个窗口最大化，而关闭按钮则将推出freeform状态。



DecorCaptionView由DecorView实例化，其中包含几个重要的数据：

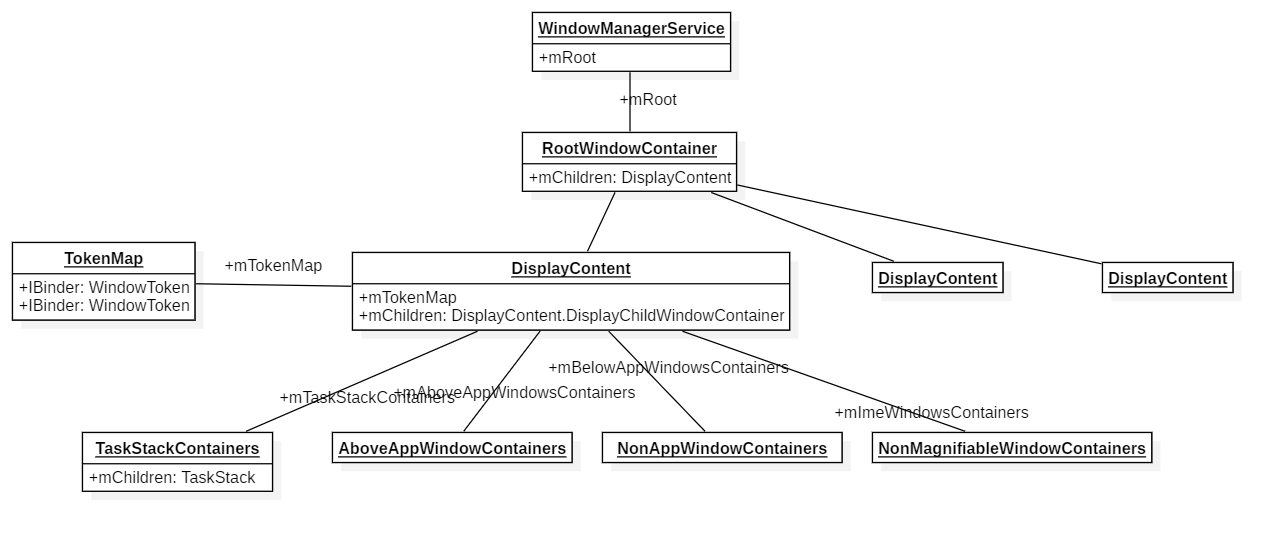
1. mMaximize和mClose。最大化和最小化的按钮
2. mMaximizeRect和mCloseRect。最大化和最小化按钮的Rect，用于判断后面的点击事件，系统中并没有使用button的click等事件处理
3. mGestureDector。手势解析处理器，用于判断手势的点击，双击滑动等手势，最终的手势结果将发回给DecorCaptionView的回调进行处理
4. mCaption。mCaption对应是DecorCaptionView这个ViewGroup的第一个child，DecorCaptionView后面还会将应用的窗口内容加载进来，所以其实就是对应decor\_caption.xml inflate出来的view。



1. mContent。mContent指下方的窗口。

## RootWindowContainer

rootwindowContainer继承自WindowContainer，其mChildren中包含的是DisplayContent，结构如下所示：

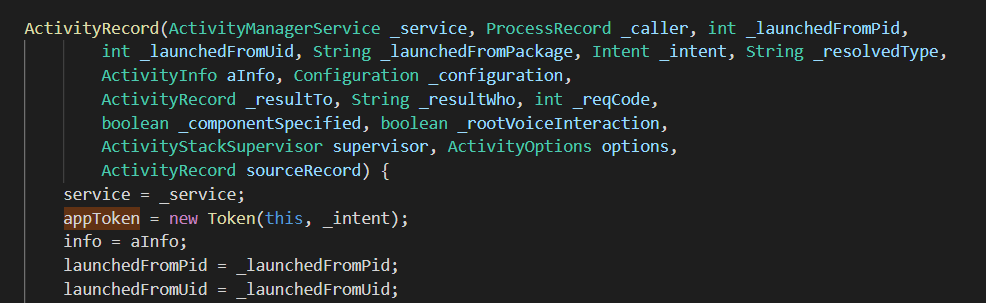


在WMS中包含一个RootWindowContainer，包含所有的window，其中包含多个DisplayContent，每个DisplayContent中都包含：

1. TaskStackContainers，
2. AboveAppWindowContainers，
3. NonAppWindowContainers。
4. NonMagnifiableWindowContainers。

在DisplayContent中有一个mTokenMap映射表，保存了windowToken的映射关系，键为IBinder对象，值为WindowToken，此处的WindowToken可能是WindowToken也有可能是AppWindowToken。IBinder对象为：

ActivityRecord创建时创建的一个Token：



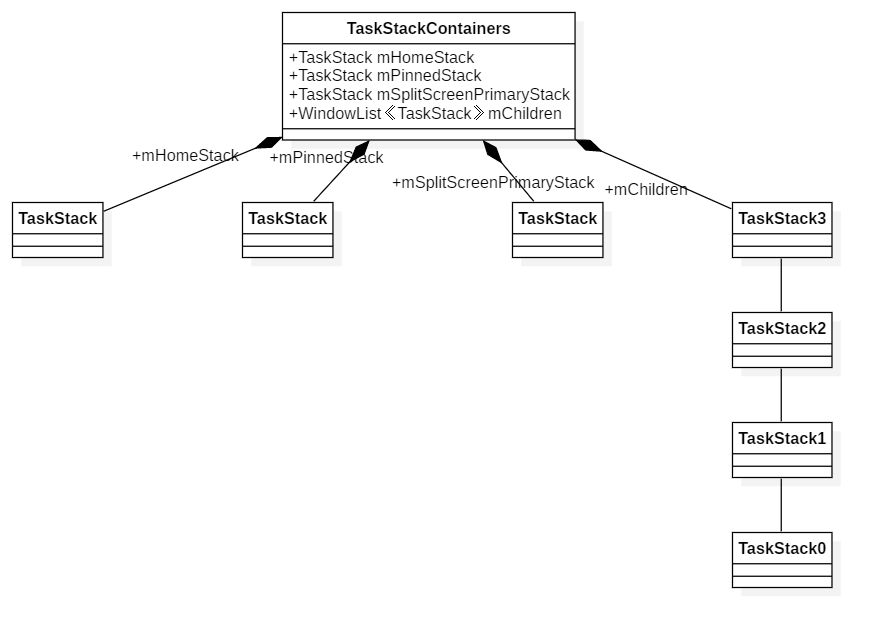
该Token简单的存储了创建ActivityRecord时的intent等信息，Token用于在displayContent中找到对应的WindowToken对象。

每个ActivityRecord都对应一个AppWindowContainerController

每一个TaskRecord也对应着一个TaskWindowContainerController

## TaskStackContainer

系统中普通的窗口的task将在DisplayContent中的TaskStackContainer中进行管理。



TaskStackContainer中包括几个特殊的TaskStack：

mHomeStack：Home的stack

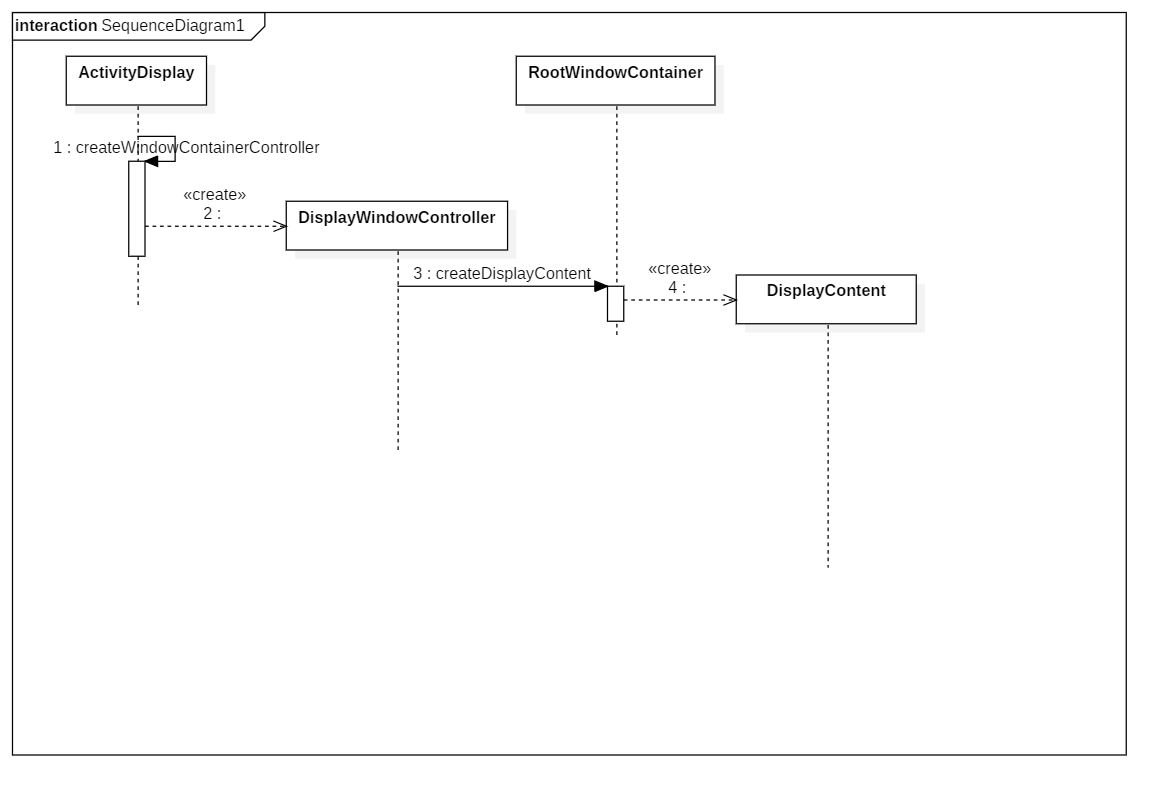
mPinnedStack：画中画的stack

mSplitScreenPrimaryStack:分屏的stack

此外在TaskStackContainer中还管理了通用的TaskStack，存储在其中的mChildren列表中，存储列表按顺序存储，下标大的stack显示在顶端。

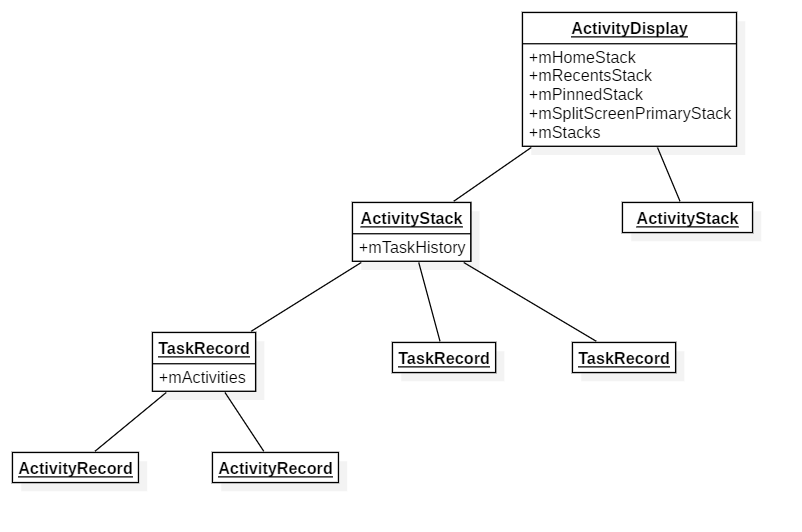
TaskStackContainer管理taskStack的方式与ActivityStack的管理方式类似，应该是一一对应的

## DisplayContent的创建过程



## AMS新启动Activity为什么会导致其他Activity pause

## ActivityStack的管理



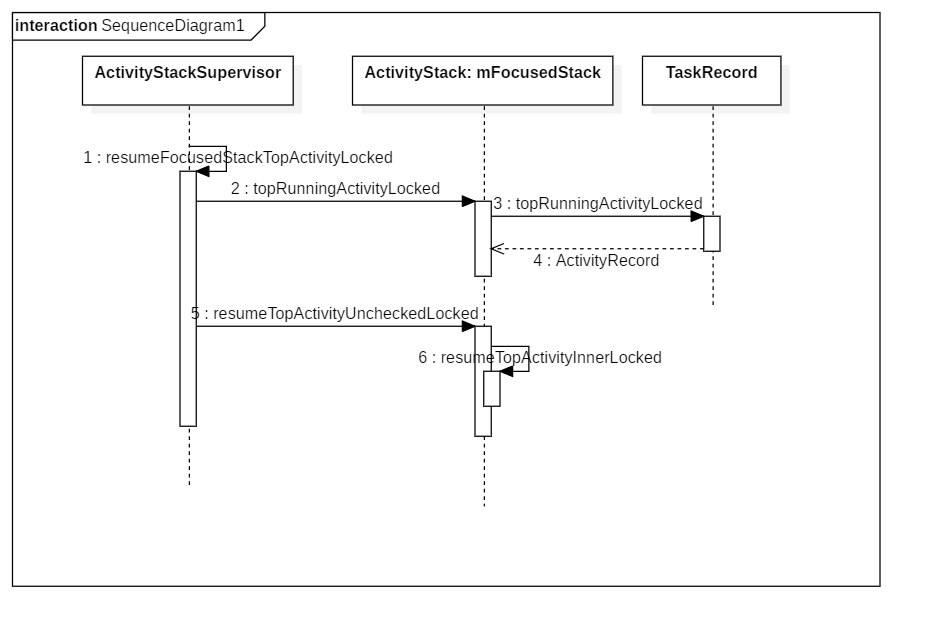
ActivityStack的管理如上所示，所有的ActivityStack都在ActivityDisplay中来管理，在ActivityDisplay中有多个ActivityStack，其中有几个特殊的stack：

1. mHomeStack。在此ActivityDisplay中的HomeActivity所在的Stack
2. mRecentsStack。在此ActivityDisplay中recents所在的stack
3. mPinnedStack。画中画专用的stack
4. mSplitScreenPrimaryStack。分屏模式的主stack
5. 其他的ActivityStack都保留在mStacks中。mStacks是其他的所有ActivityStack的列表，mStacks是一个数组，靠后的stacks就在对应ActivityDisplay的顶端。

每个ActivityStack中都具有多个Task，每个Task用一个TaskRecord指代，存储在其内部的mTaskHistory中，ActivityStack中的Task也有层级顺序。在mTaskHistory列表中，越靠后，对应的Task就越在顶端。

每个Task中包含多个ActivityRecord，对应就是在应用侧的Activity，在Task中通过TaskRecord中的mActivities列表中管理，该列表也是一个数组，在mActivities中存在最尾端的ActivityRecord就是该Task中最顶端的Activity

## Activity的resume调用逻辑



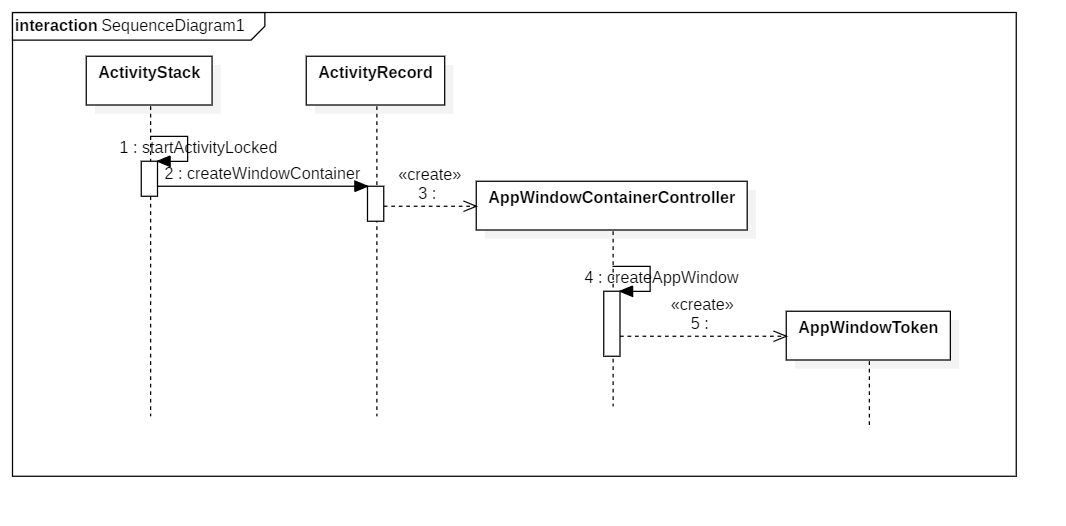
Activity的resume由ActivityStackSupervisor发起，当调用resumeFocusedStackTopActivity时，首先到ActivityStackSupervisor的focused栈的TaskRecord列表（mTaskHistory）中从后往前找，在对应的 TaskRecord管理的Activities列表中，找到一个没有被finish，并且可以获得焦点的ActivityRecord，将其返回给ActivityStackSupervisor后，由ActivityStackSupervisor调用ActivityStack进行resume处理。

## ActivityStack中如何找到下一个需要启动的Activity

ActivityStack中有多个Activity，Activity存在ActivityStack内部的mTaskHistory中

## AppWindowToken

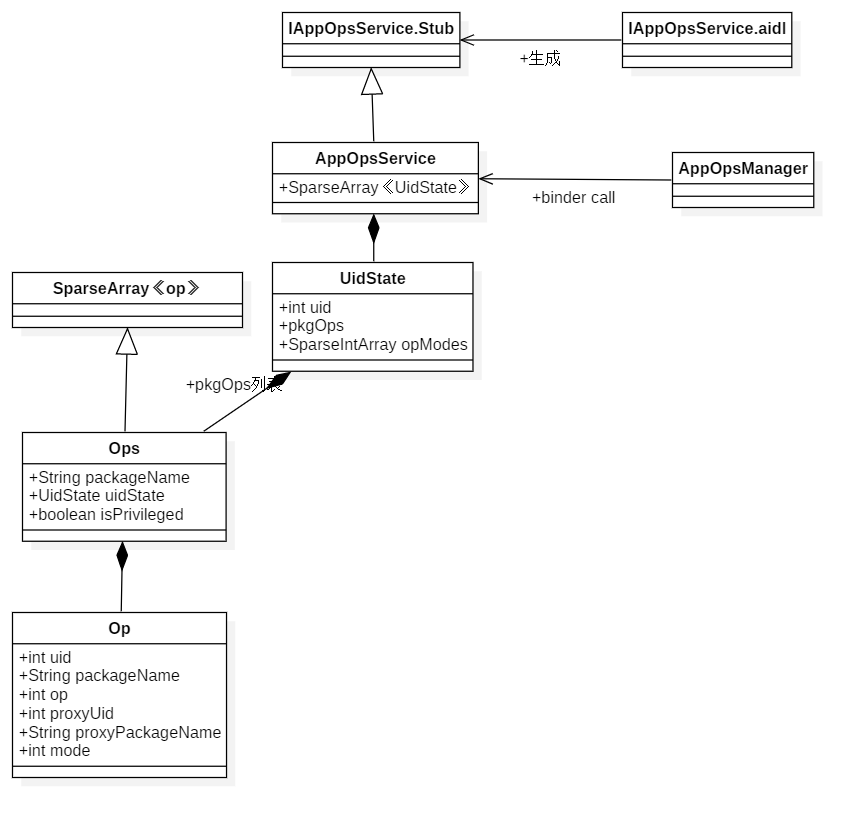
appwindowToken的创建过程



WindowToken

# AppOpsService结构

## AppOpsService的静态结构



AppOps在系统中由AppOpsService服务提供实现，AppOpsService实现了IAppOpsService的Stub方法，也就是说其支持对外通过binder的方式来调用功能。

在AppOpsService中，通过Uid的方式来管理系统中所有的应用的Op项，因为系统中可能存在sharedUserId的应用，对于sharedUserId的应用，其Op项是一样的。AppOpsService中通过UidState的列表来存储当前系统中所有的Uid状态。

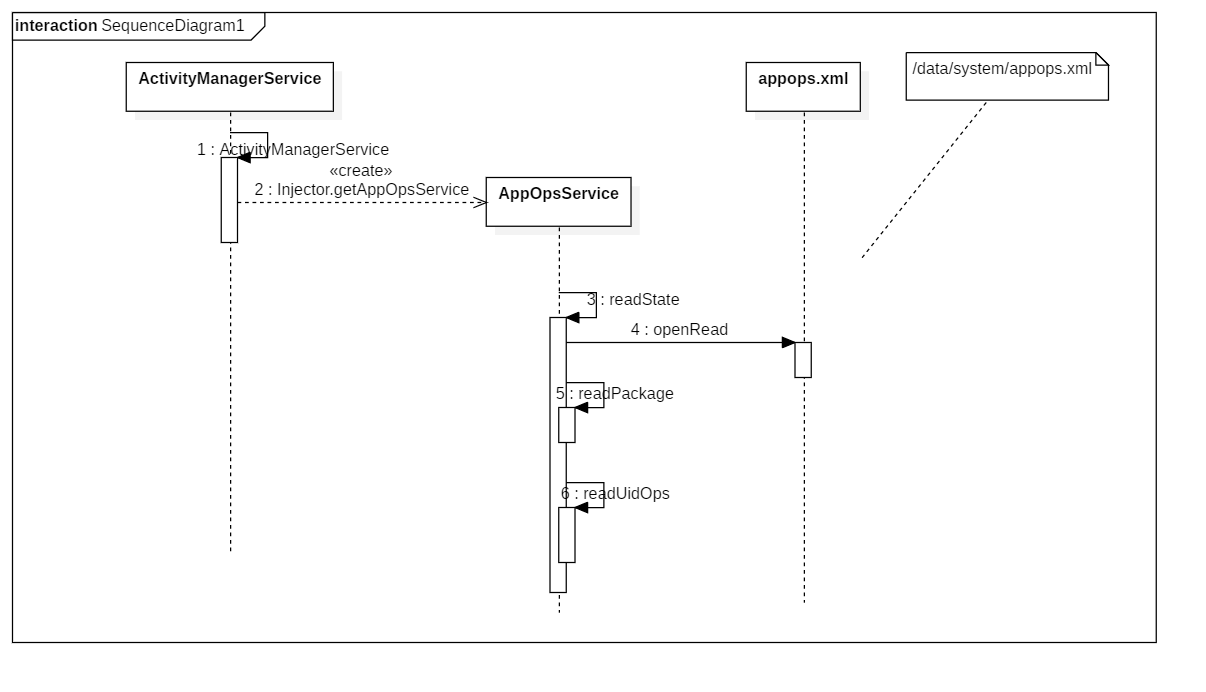
每个UidState结构中又存储了一个Ops对应某个应用，Ops的packgeName即指明了对应的应用报名，此外在Android中Ops设计为SpraseArray的子类，所以其中会包含多个Op项。

每个Op项中具体就指明了Op的信息，op字段指明了对应的OP操作，mode指明了当前AppOpsService对该应用的授权情况。此外其中还有op的reject时间等信息，在上图中没有体现出来。

ProxyUid和proxyPackageName的使用场景不明。在Q版本重新进行了重构，可以支持多个proxy。

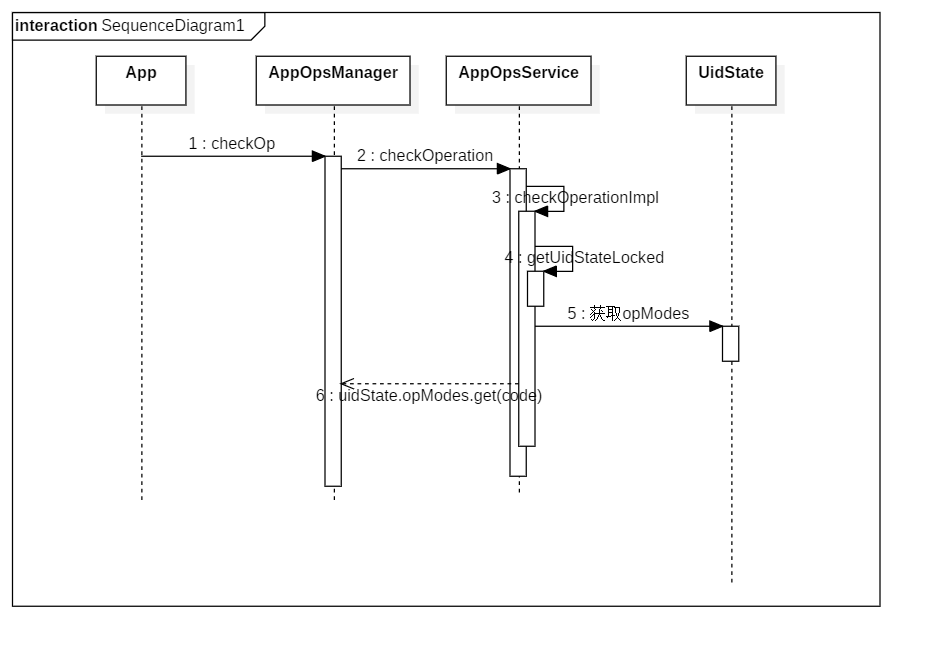
应用侧可以通过AppOpsManager SDK类接口调用AppOpsService的功能方法，AppOpsManager在应用进程的SystemServiceRegistry中启动时注册，绑定到AppOpsService服务。

## AppOpsService的启动过程



AppOpsService由AMS启动，启动后，AppOpsService读取当前系统中持久化的appops的Op状态，通过readPackage和readUidOps等解析函数解析出来，状态存储到前面的UidState等数据结构中。

## AppOpsService checkOp过程



CheckOp需要传入需要检查的Op项，uid和包名：

public int checkOp(int op, int uid, String packageName)

如果返回