Android 系统框架学习记录

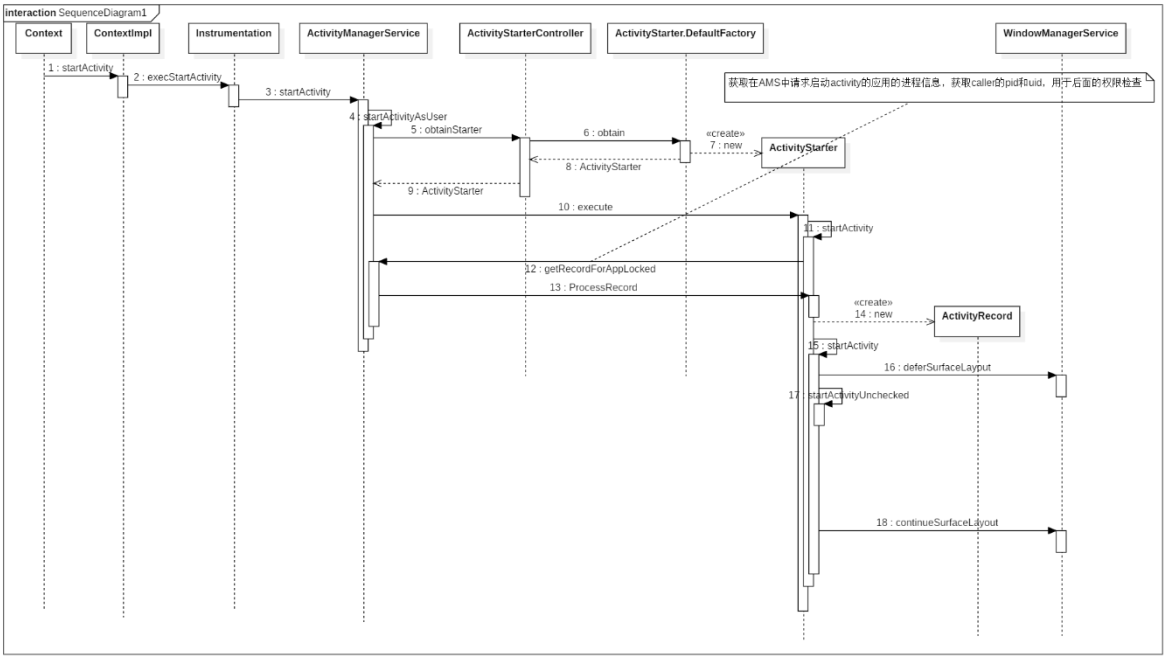
Author:applematrix
e-Mail:cshdzxjtu@163.com

目录

AMS&WMS 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();
}
}
```

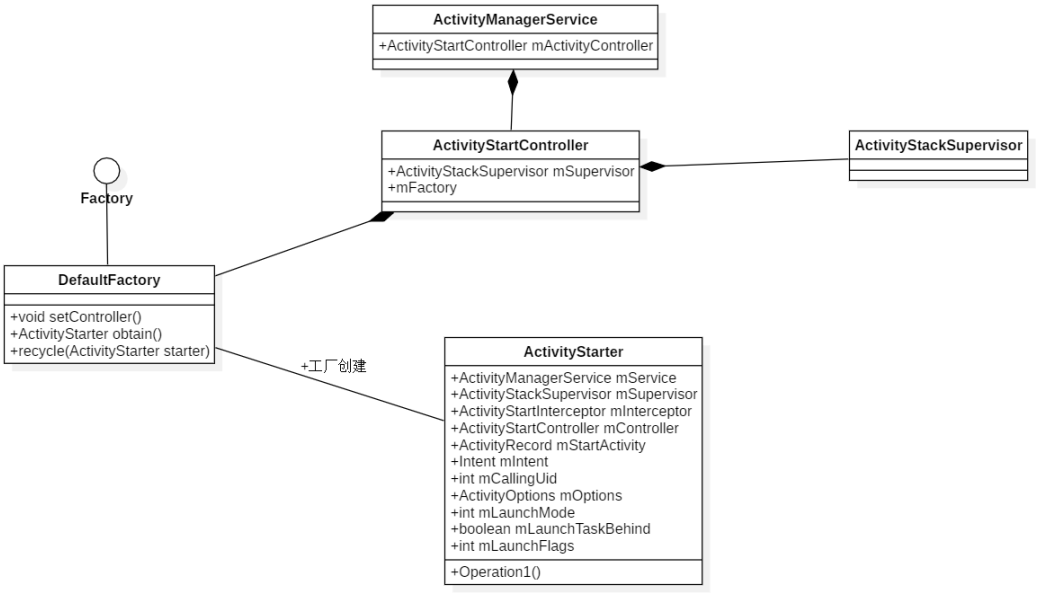
从 ActivityManagerService 中的 startActivity 函数开始走读：

```
    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;
```

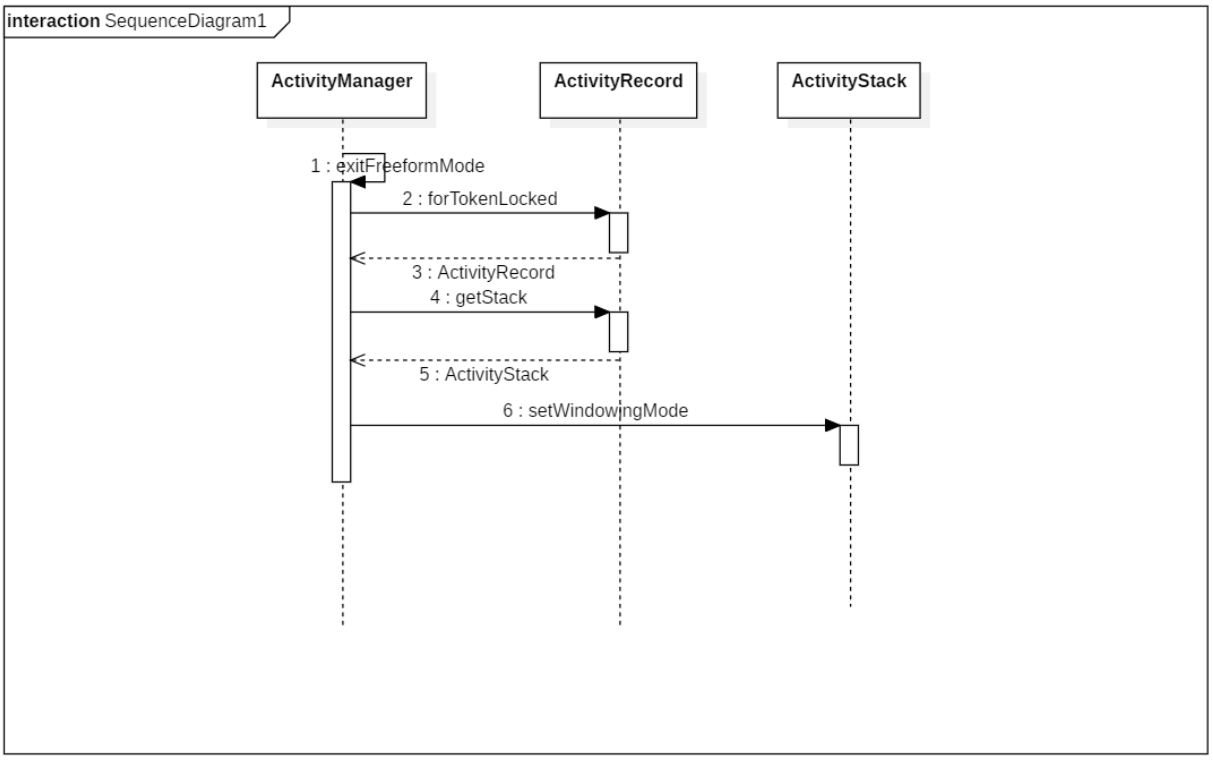
在看下 ActivityStarter 中的第二个 startActivity 函数：

```
    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()
```

2、 获取当前 stack 中处于 topTask

```
TaskRecord topTask = topTask()
```

3、 获取当前的 stack 对应的 display

```
ActivityDisplay display = getDisplay()
```

4、 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，

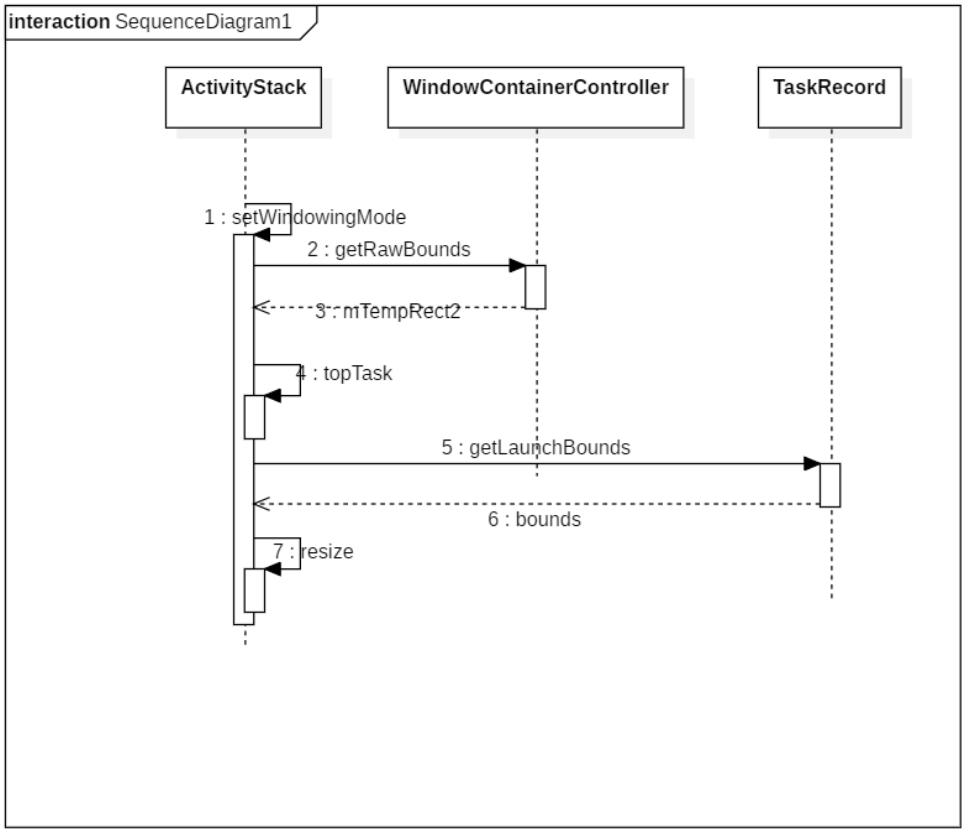
```
int windowingMode = creating
```

```
                ? 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
```

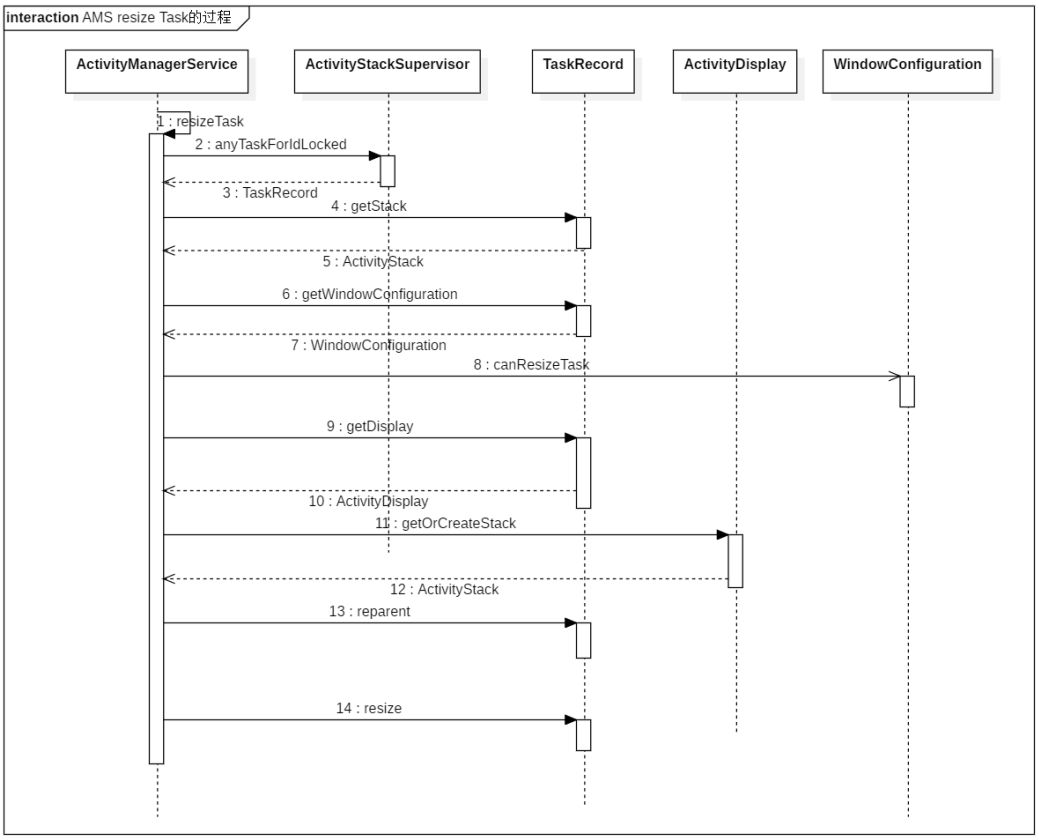
```
                    // of the stack, but we do try to make sure the tasks are
still contained
                    // with the bounds of the stack.
                    mTmpRect2.set(task.getOverrideBounds());
                    fitWithinBounds(mTmpRect2, bounds);
                    task.updateOverrideConfiguration(mTmpRect2);
                } 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);
```

2、 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);
```

3、 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;
      }
```

2、 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);
        }
```

3、 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

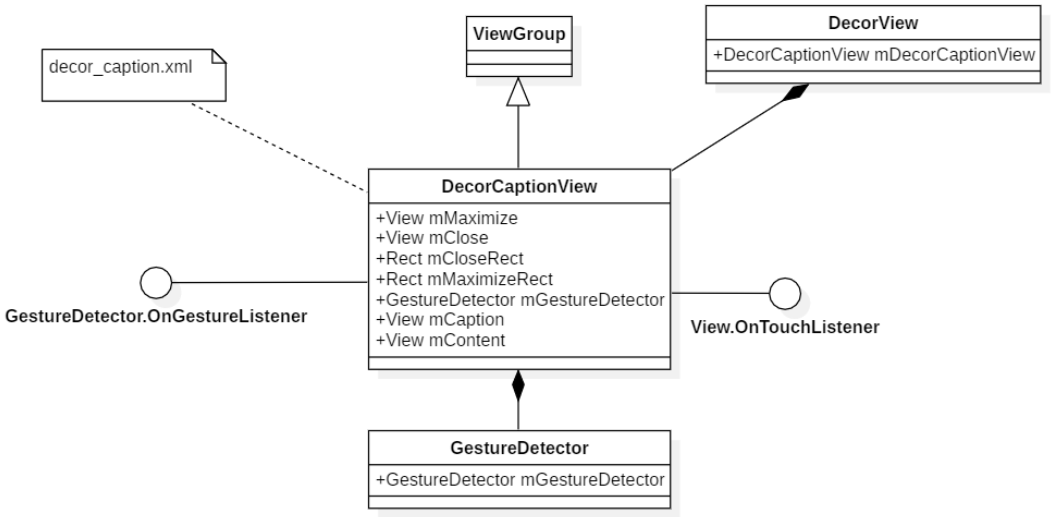
```
    if (stack.mDisplayId != DEFAULT_DISPLAY && !mService.mSupportsMultiDisplay) {
            throw new IllegalArgumentException("Device doesn't support multi-display,
can not"
                    + " reparent task=" + task + " to stackId=" + stackId);
        }
```

4、 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);
        }
```

5、 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。

```
    <LinearLayout
            android:id="@+id/caption"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="end"
            android:background="@drawable/decor_caption_title"
            android:focusable="false"
            android:descendantFocusability="blocksDescendants" >
        <Button
                android:id="@+id/maximize_window"
                android:layout_width="32dp"
                android:layout_height="32dp"
                android:layout_margin="5dp"
                android:padding="4dp"
                android:layout_gravity="center_vertical|end"
                android:contentDescription="@string/maximize_button_text"
                android:background="@drawable/decor_maximize_button_dark" />
        <Button
                android:id="@+id/close_window"
                android:layout_width="32dp"
                android:layout_height="32dp"
                android:layout_margin="5dp"
                android:padding="4dp"
                android:layout_gravity="center_vertical|end"
                android:contentDescription="@string/close_button_text"
                android:background="@drawable/decor_close_button_dark" />
    </LinearLayout>
</com.android.internal.widget.DecorCaptionView>
```

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

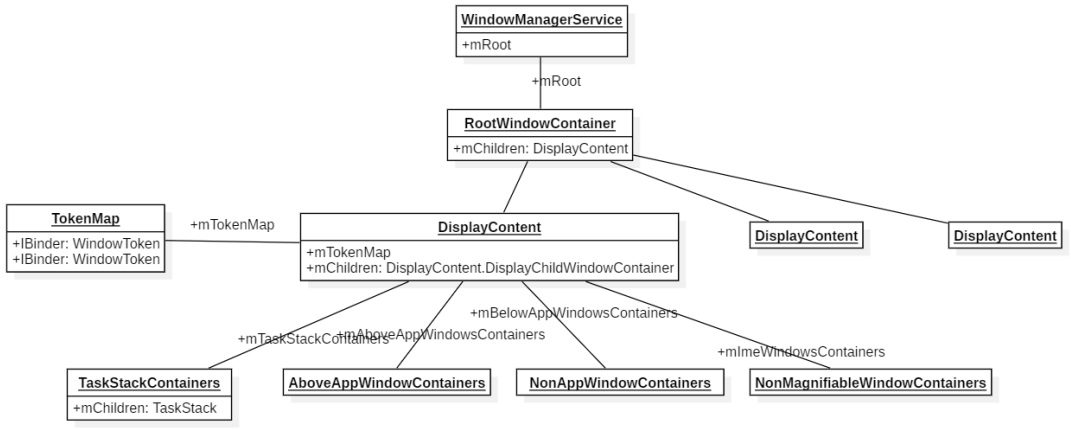
4、 mCaption。mCaption 对应是 DecorCaptionView 这个 ViewGroup 的第一个 child，DecorCaptionView 后面还会将应用的窗口内容加载进来，所以其实就是对应 decor_caption.xml inflate 出来的 view。

```
@Override
protected void onFinishInflate() {
    super.onFinishInflate();
    mCaption = getChildAt(0);
}
```

5、 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：

```
ActivityRecord(ActivityManagerService _service, ProcessRecord _caller, int _launchedFromPid,
        int _launchedFromUid, String _launchedFromPackage, Intent _intent, String _resolvedType,
        ActivityInfo aInfo, Configuration _configuration,
        ActivityRecord _resultTo, String _resultWho, int _reqCode,
        boolean _componentSpecified, boolean _rootVoiceInteraction,
        ActivityStackSupervisor supervisor, ActivityOptions options,
        ActivityRecord sourceRecord) {
    service = _service;
    appToken = new Token(this, _intent);
    info = aInfo;
    launchedFromPid = _launchedFromPid;
    launchedFromUid = _launchedFromUid;
```

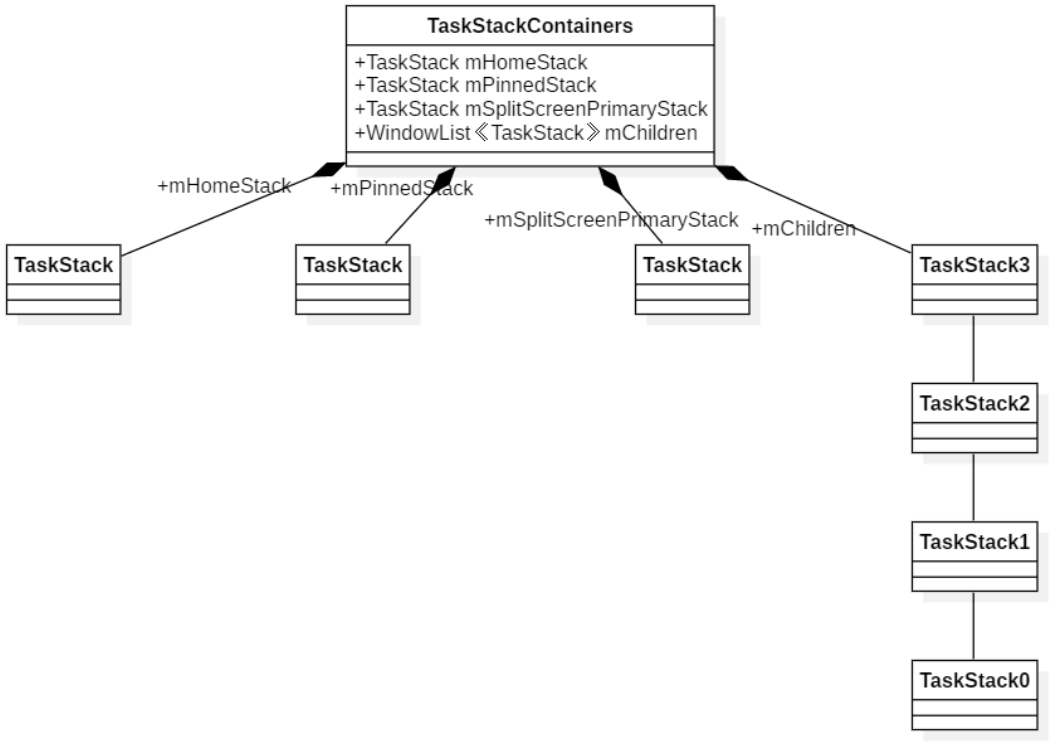
该 Token 简单的存储了创建 ActivityRecord 时的 intent 等信息，Token 用于在 displayContent 中找到对应的 WindowToken 对象。

每个 ActivityRecord 都对应一个 AppWindowContainerController

每一个 TaskRecord 也对应着一个 TaskWindowContainerController

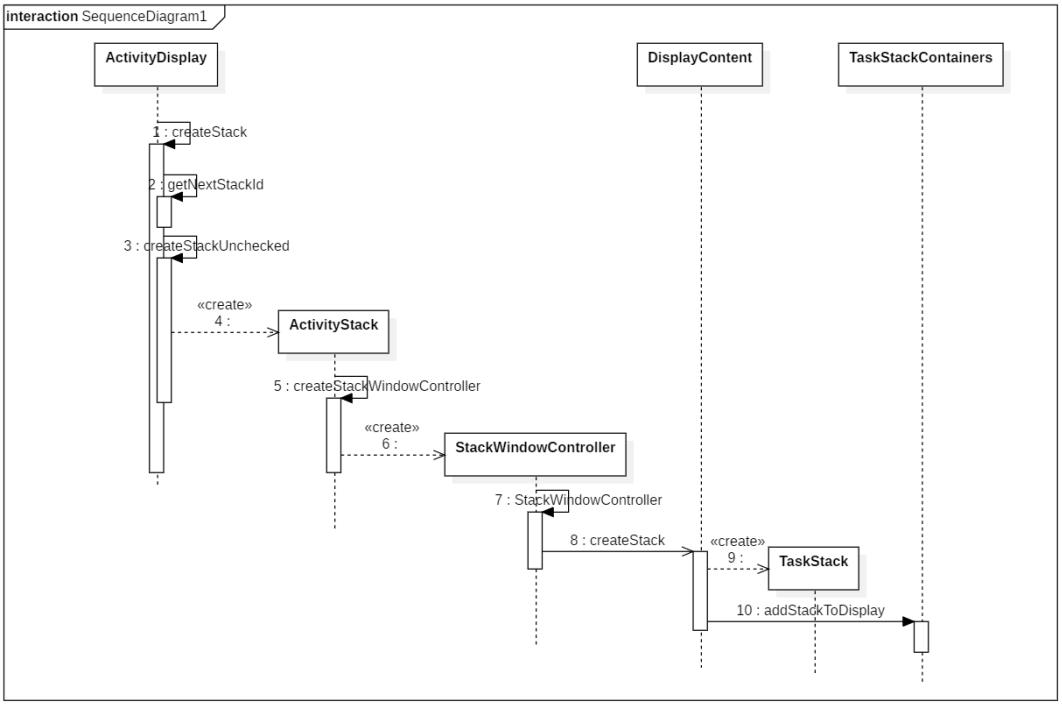
TaskStackContainer

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。

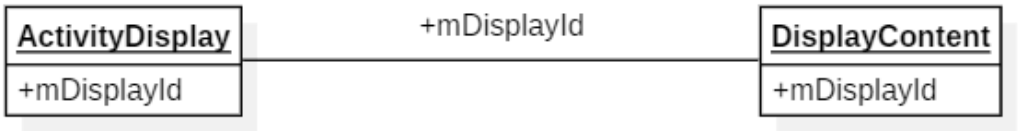
TaskStackContainer 中的 Stack 创建过程与 ActivityStack 的关系

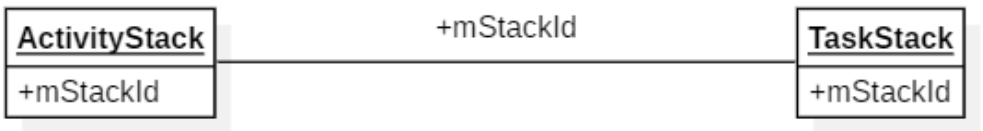


ActivityDisplay 中创建 ActivityStack 时，会同事创建 StackWindowController，在 StackWindowController 中调用 DisplayContent 的 createStack 方法创建出 TaskStack，创建出来后通过 addStackToDisplay 将 stack 加入到 TaskStackContainer 的列表中进行管理。

每个 Stack 都有一个 stackId，在上图中可以看到 stackId 时在 ActivityStack 中通过 getNextStackId 分配的，该 stackId 同样作为 taskStack 的 Id 来使用。

可以看出在框架层有两个关于 Display 的数据结构，两个关于 Stack 的数据结构，他们之间是一一对应的，通过 id 映射：

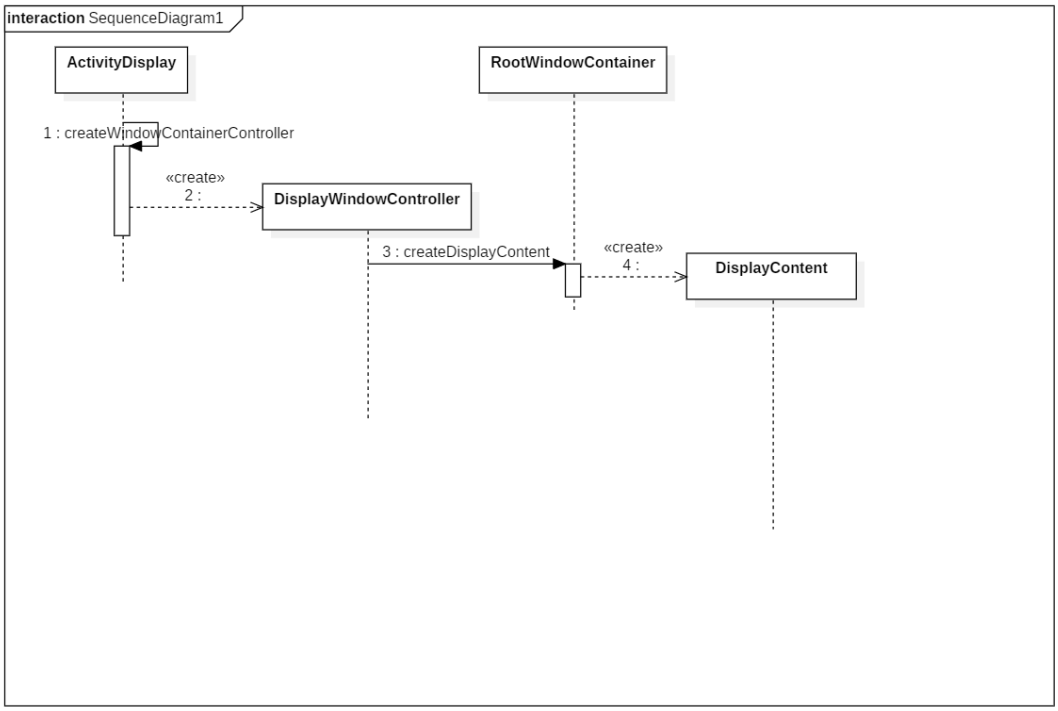




区别在与 ActivityDisplay 是 AMS 中管理 display 的数据结构，而 DisplayContent 是 WMS 中管理 display 的数据结构，同样 ActivityStack 是 AMS 中管理 stack 的数据结构，而 TaskStack 则

是 WMS 中管理 stack 的数据结构。

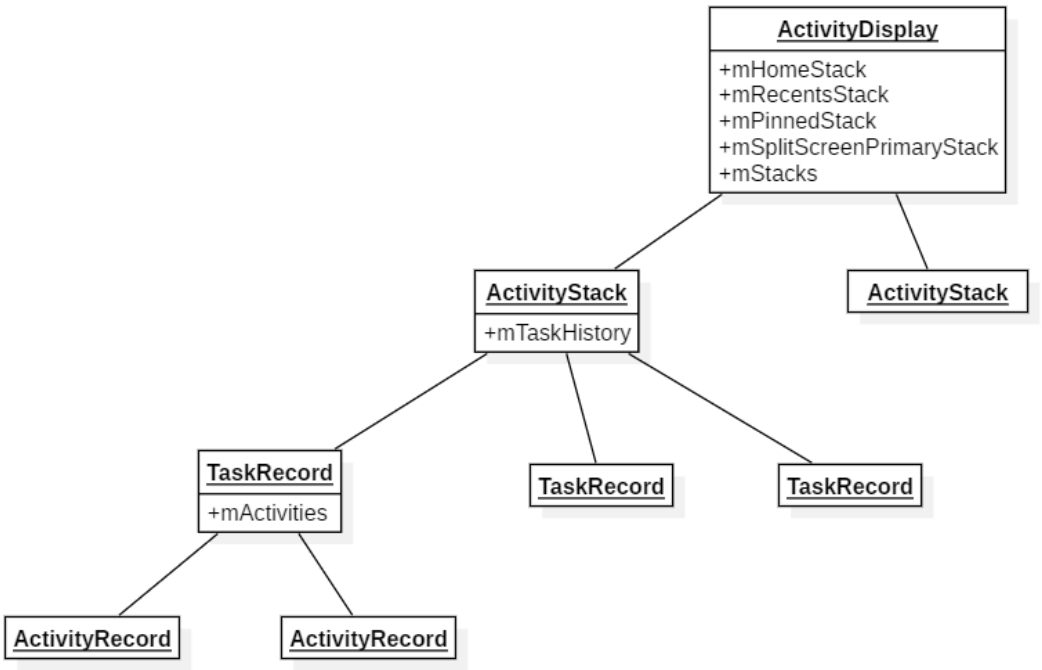
DisplayContent 的创建过程



前面提到 ActivityDisplay 是 AMS 中管理 Display 的数据结构，在 WMS 中管理 display 相关的信息由 displaycontent 承载。当 AMS 中创建出 ActivityDisplay 的时候，会通过 DisplayWindowController 链接到 WMS 的 RootWindowContainer 中，在 RootWindowContainer 中创建出对应的 DisplayContent 出来。

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

ActivityStack 的管理



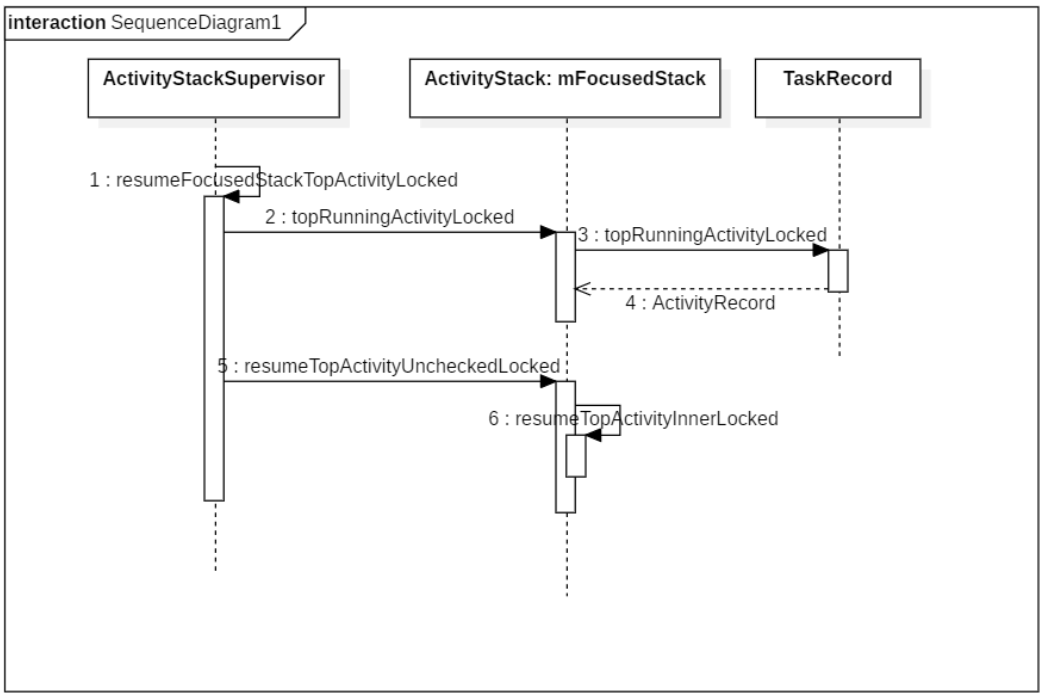
ActivityStack 的管理如上所示，所有的 ActivityStack 都在 ActivityDisplay 中来管理，在 ActivityDisplay 中有多个 ActivityStack，其中有几个特殊的 stack：

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



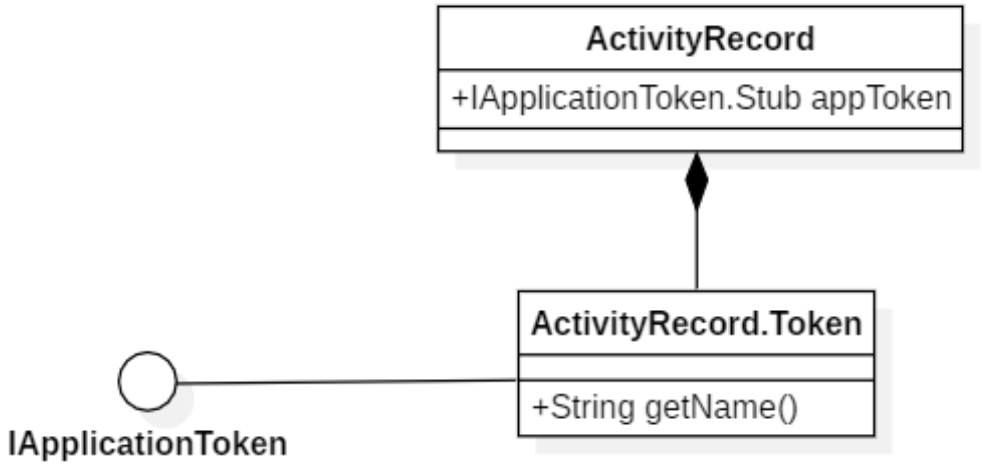
Activity 的 resume 由 ActivityStackSupervisor 发起，当调用 resumeFocusedStackTopActivity 时，首先到 ActivityStackSupervisor 的 focused 栈的 TaskRecord 列表（mTaskHistory）中从后往前找，在对应的 TaskRecord 管理的 Activities 列表中，找到一个没有被 finish，并且可以获得焦点的 ActivityRecord，将其返回给 ActivityStackSupervisor 后，由 ActivityStackSupervisor 调用 ActivityStack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

AMS 和 WMS 中的各种 Token

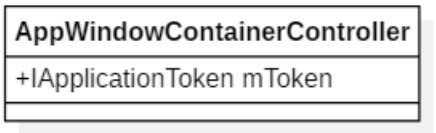
ActivityRecord 中的 Token



每个 ActivityRecord 中有一个 appToken，该 Token 在 ActivityRecord 创建时创建，所以可以认为可以用于关联 ActivityRecord。对外的接口方法在 IApplicationToken 中体现。IApplicationToken 只提供了一个 getName 的函数。实现为返回 ActivityRecord 中 intent 的 componentName。

AppWindowContainerController 中的 Token

在 AppWindowContainerController 中也有一个 token，AppWindowContainerController 是 AMS 到 WMS 的桥梁。



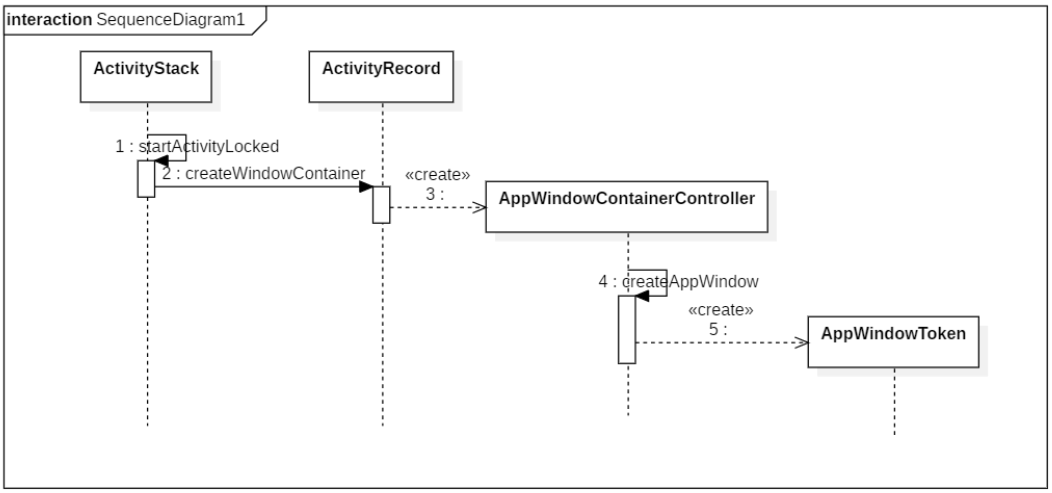
通过代码可以看到 AppWindowContainerController 中的 token 就是 ActivityRecord 的 Token，是同一个对象：

```
public AppWindowContainerController(TaskWindowContainerController taskController,
        IApplicationToken token, AppWindowContainerListener listener, int index,
        int requestedOrientation, boolean fullscreen, boolean showForAllUsers, int configChanges,
        boolean voiceInteraction, boolean launchTaskBehind, boolean alwaysFocusable,
        int targetSdkVersion, int rotationAnimationHint, long inputDispatchingTimeoutNanos,
        WindowManagerService service) {
    super(listener, service);
    mHandler = new H(service.mH.getLooper());
    mToken = token;
    synchronized(mWindowMap) {
        AppWindowToken atoken = mRoot.getAppWindowToken(mToken.asBinder());
        if (atoken != null) {
            // TODO: Should this throw an exception instead?
            Slog.w(TAG_WM, "Attempted to add existing app token: " + mToken);
            return;
        }
```

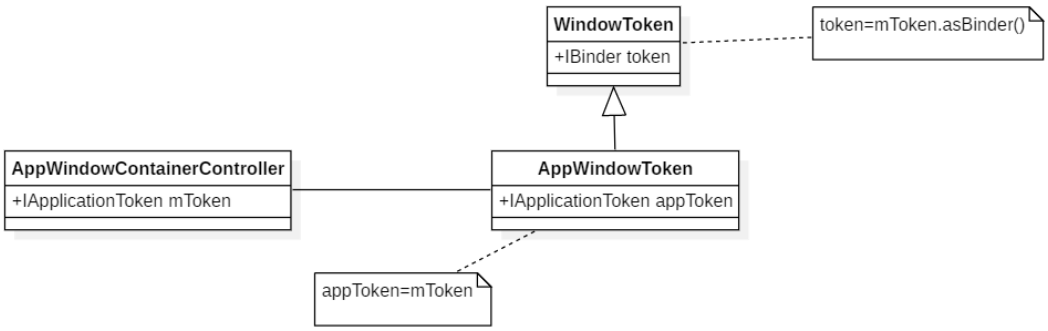
ActivityRecord 创建 AppWindowContainerController 时直接将 token 传递给新建的对象，所以在 AppWindowContainerController 保存的就是 ActivityRecord 中的 token。

AppWindowToken

appwindowToken 的创建过程



AppWindowToken 由 AppWindowContainerController 创建，其中包含了比较多的关于窗口的信息，与其他的 token 的关系如下：



AppWindowToken 是 WindowToken 的子类，在 WindowToken 中具有一个 token，对于

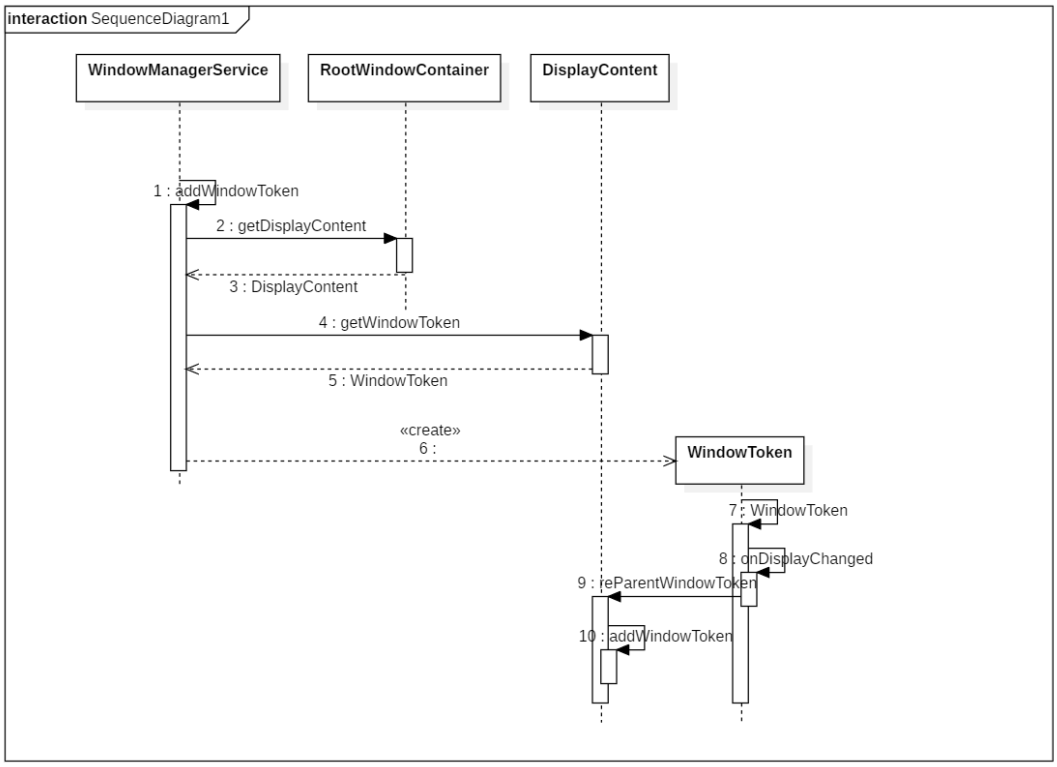
AppWindowToken，其 token 实际上也是来自于对应的 appWindowContainerController 的 mToken 作为 binder 的引用。而其内部的 appToken 实际上就是 appWindowContainerController 的 mToken。所以 AppWindowToken 是在 WMS 中又做了一次封装，其内部可以通过 AppWindowContainerController 传递过来的 token 引用到在 AMS 中 ActivityRecord 中对应的 Token，从而可以找到对应的 ActivityRecord。

AppWindowToken 的创建是通过 AMS 的 startActivity 来触发的。

WindowToken

WindowToken 是另一种 token，创建 WindowToken 的场景有以下几种

通过 WMS 的 addWindowToken 接口创建 WindowToken



在 WMS 中，提供了一个 addWindowToken 的接口，外部可以通过此接口调用来直接创建 WindowToken：

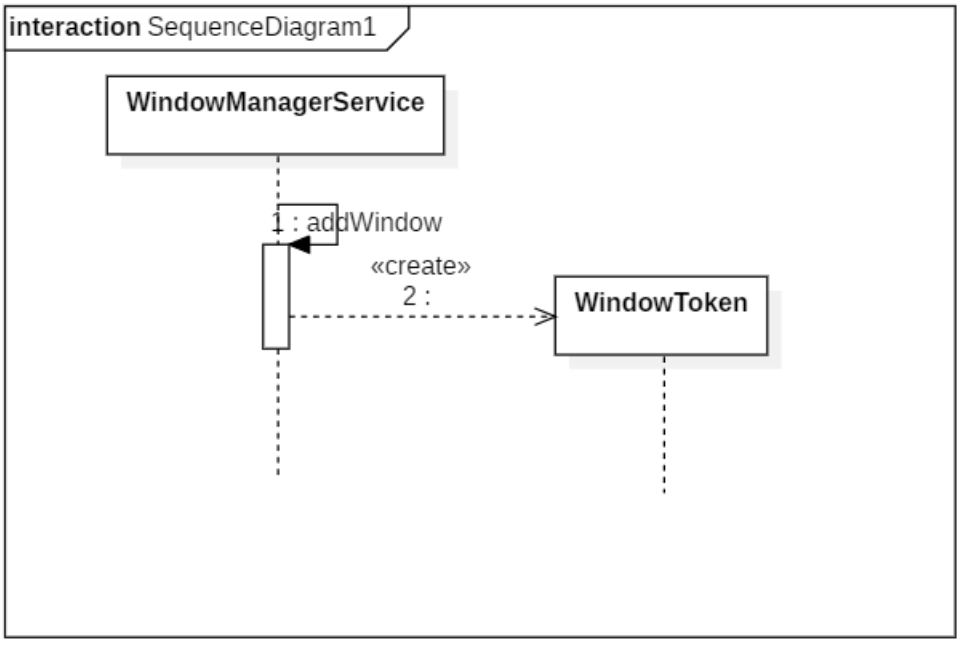
```
@Override
public void addWindowToken(IBinder binder, int type, int displayId) {
    if (!checkCallingPermission(MANAGE_APP_TOKENS, "addWindowToken()")) {
        throw new SecurityException("Requires MANAGE_APP_TOKENS permission");
    }

    synchronized(mWindowMap) {
        final DisplayContent dc = mRoot.getDisplayContent(displayId);
        WindowToken token = dc.getWindowToken(binder);
```

该接口指定了创建的窗口类型和 displayId 用于指定在哪个 display 上创建出对应的

windowToken 出来。
创建过程：
1、 首先通过 RootWindowContainer 获取到对应 displayId 的 DisplayContent 对象
2、 通过 DisplayContent 的 getWindowToken 接口查询当前的 binder 是否在 DisplayContent 中已经有了对应的 windowToken
3、 如果没有创建了 windowToken，那么会创建出新的 WindowToken，创建完成后会将新的 WindowToken 加入到对应的 DisplayContent 中进行管理。
4、 在 DisplayContent 中通过 binder 即可以查询到对应的 WindowToken。
前面已经提到 WindowToken 中具有一个 token 对象，此处创建的新的 WindowToken 中的 token 对象即为 addWindowToken 时传入的 binder 对象。

那么什么场景会调用 addWindowToken 接口呢？
1、 输入法加入输入法的窗口
2、 Dream 添加 dream 的窗口
3、 通知管理服务添加 toast 窗口
4、 壁纸管理服务添加窗口
5、 Voiceinteraction 中添加 voice 交互相关的窗口
以上几种场景都是添加非应用的窗口的特殊场景。所以主要的应用场景，还是通过 AppWindowToken 的方式来管理。

通过 WMS 的 addWindow 接口创建 WindowToken

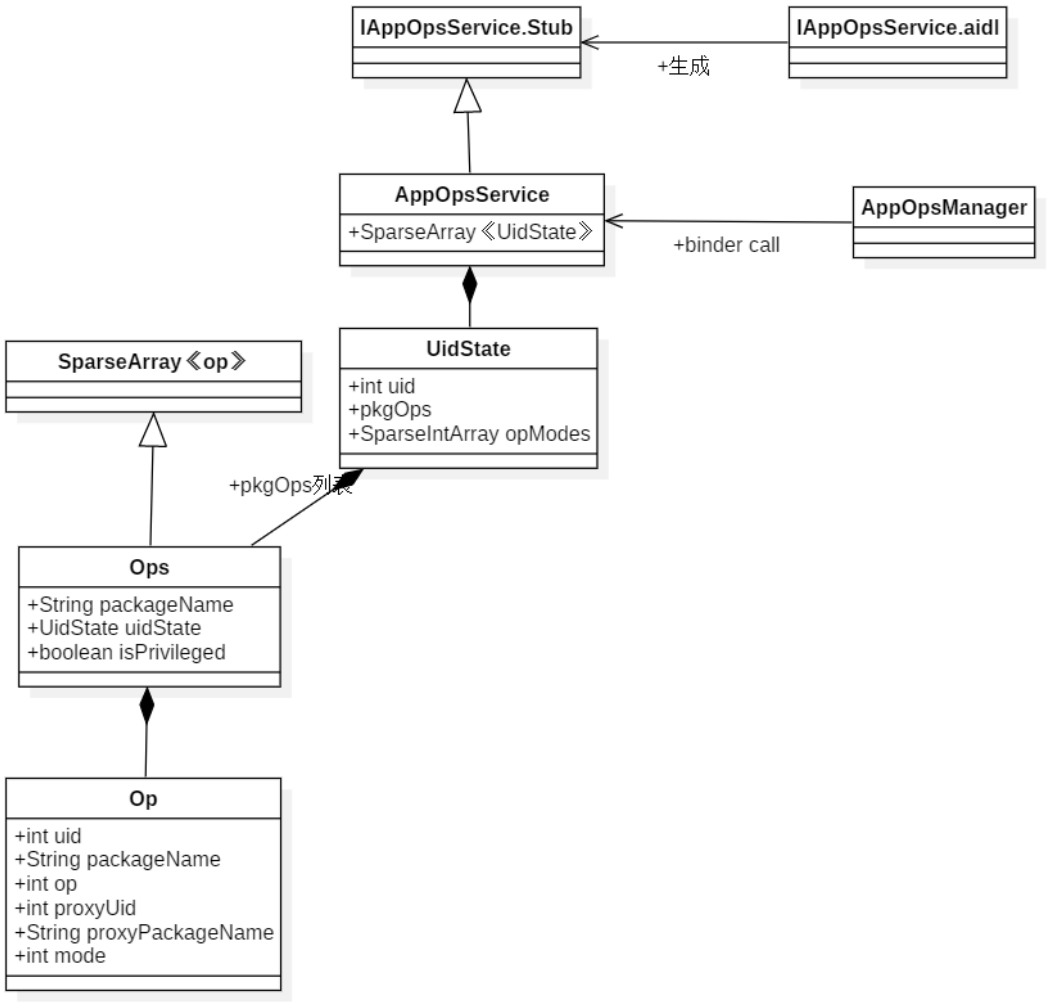


WMS 的 addWindow 接口同样可以创建出 WindowToken，创建 WindowToken 后的处理与 addWindowToken 一样。
在 WMS 中的 addWindow 接口中创建的 WindowToken 的流程主要是在特殊场景下的

addWindow，对于应用级别的 window，不会执行到。此处不做过多分析。

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。

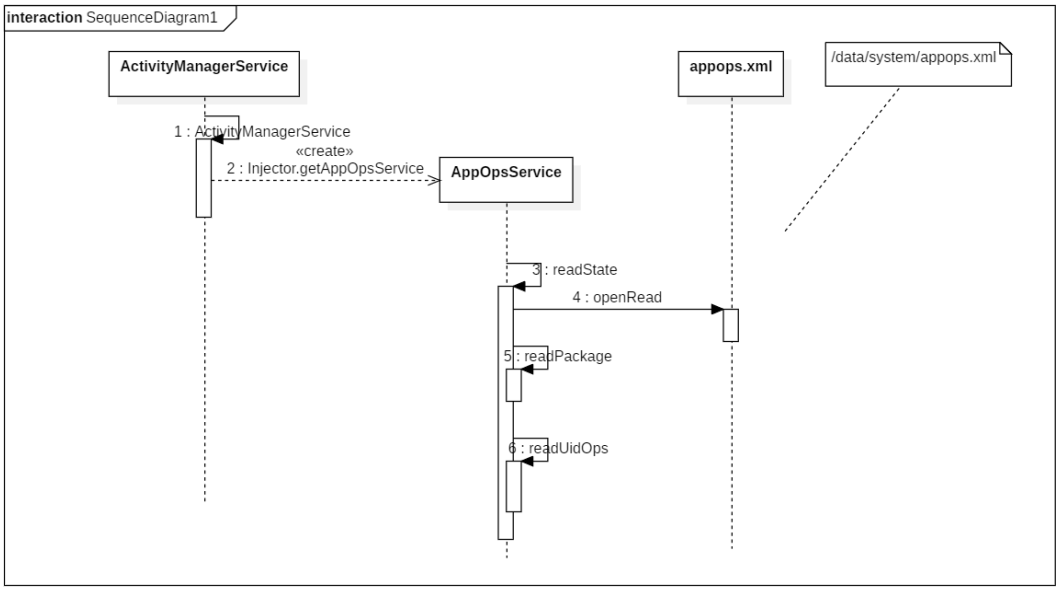
每个 Op 项中具体就指明了 Op 的信息，op 字段指明了对应的 OP 操作，mode 指明了当前 AppOpsService 对该应用的授权情况。此外其中还有 op 的 reject 时间等信息，在上图中没

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。

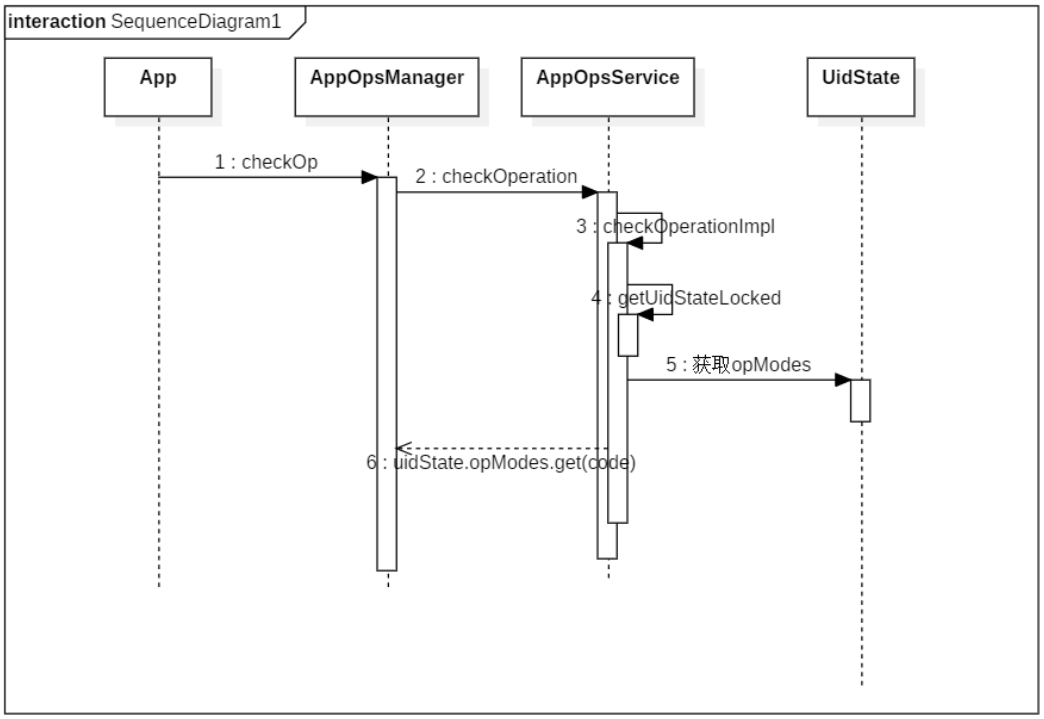
应用侧可以通过 AppOpsManager SDK 类接口调用 AppOpsService 的功能方法，AppOpsManager 在应用进程的 SystemServiceRegistry 中启动时注册，绑定到 AppOpsService 服务。

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 ALLOW 则表明对应的 Op 操作被允许