WMS 进阶

一、投屏

投屏，就是将一台设备上的媒体内容，通过一定的技术方案，在另外一台设备上显示。其中这个一定的技术方案，主要分为两种：

推送模式：主要用于投屏音视频。投屏之后手机可以关闭，电视（接收端）不会停止播放。核心原理就是通过指定协议，类似于蓝牙那样搜索匹配，并将音视频的播放地址传输过去，然后接收端播放这个地址的流媒体。常见的投屏协议有DLNA、Airplay。

镜像模式：所谓的镜像就是同屏，把手机（发送端）的屏幕内容同步传输到电视（接收端）上显示。核心原理就是一边录屏一遍发送给电视同步播放。常见协议有Miracast、Airplay。

一般来说我们常用的投屏是推送模式，本文主要也是介绍这个。

1.1 投屏协议——DLNA

DLNA（Digital Living Network Alliance），即数字家庭网络联盟。

DLNA不是技术，而是一种解决方案，它是多种技术的整合，并致力于构建家庭媒体共享。

DLNA包含多种网络协议，如http、https、upnp等，其中upnp是其重要组成部分。

DLNA主要包含以下四种产品：

DMS，即Digital Media Server（数字媒体服务器）的缩写，其主要作用是作为媒体内容的提供者，为DMP/DMR提供内容播放，DMS可控制提供哪些媒体内容。

DMP，即Digital Media Player（数字媒体播放器）的缩写，可搜索并播放DMS的内容，其作用相当于DMR+DMC。

DMC，即Digital Media Controller（数字媒体控制器）的缩写，可搜索并控制DMR播放DMS提供的内容，即控制DMR与DMS的交互。

DMR，即Digital Media Renderer（数字媒体渲染器）的缩写，可播放DMS提供的内容。

DLNA并不是真正的无线显示解决方案。相反，它只是一种在一个设备上获取内容并在另一台设备上播放内容的方法。也就是说他不是真正的投屏技术。

1.2 投屏协议 —— Miracast

Miracast是Wi-Fi联盟制定的Wi-Fi投屏行业标准，实质上是对Apple AirPlay的回应。Miracast支持内置在Android 4.2+和Windows 8.1、Windows 10。允许Android智能手机、Windows平板电脑和笔记本电脑以及其他设备以无线方式传输到兼容Miracast的接收器比如智能电视、平板电脑等。当前已经有很多电视盒子都支持Miracast协议，比如小米盒子、荣耀盒子等等，小米手机、华为的手机也都支持Miracast协议，配合小米盒子、荣耀盒子即可实现投屏。

1.3 投屏协议 ——Airplay

AIrplay协议是苹果的协议，主要局限在仅适用于 Apple 设备，我们在这里不做展开。

1.4 投屏协议 —— 其他第三方

最后就是很多专门投屏的投屏APP，这些APP要么是实现了上面几种协议，要么是自己实现一套私有协议。手机和智能电视都要安装这些APP，否则无法投屏。而前面几个协议都是标准协议，操作系统内置，无需安装。比较著名的投屏APP有乐播投屏、APowerMirror等，使用都很方便，一般是通过扫描智能电视显示的二维码来实现投屏到特定电视机上。这些投屏APP的另外一个好处就是：不局限在同一个局域网内，可以跨三层网络、甚至广域网。

画中画

画中画支持

Android 8.0（API 级别 26）允许以画中画模式启动 Activity。画中画是一种特殊类型的多窗口模式，最常用于视频播放。使用该模式，用户可以通过固定到屏幕一角的小窗口观看视频，同时在应用之间进行导航或浏览主屏幕上的内容。

画中画利用 Android 7.0 中的多窗口模式 API 来提供固定的视频叠加窗口。要将画中画添加到您的应用中，您需要注册支持画中画的 Activity、根据需要将 Activity 切换为画中画模式，并确保当 Activity 处于画中画模式时，界面元素处于隐藏状态且视频能够继续播放。

画中画窗口会显示在屏幕的最上层，位于系统选择的一角。您可以将画中画窗口拖动到其他位置。当您点按该窗口时，会看到两个特殊的控件：全屏切换开关（位于窗口的中心）和关闭按钮（右上角的“X”）。

您的应用会控制当前 Activity 在何时进入画中画模式：

1、Activity 可以在用户点按主屏幕或最近使用的应用按钮来选择其他应用时，进入画中画模式。（这就是 Google 地图在用户同时运行其他 Activity 时继续显示方向的方式。）

2、您的应用可以在用户从某个视频返回以前浏览其他内容时，将该视频切换到画中画模式。

3、您的应用可以在用户观看到某集内容的结束时将视频切换到画中画模式。主屏幕会显示有关这部电视剧下一集的宣传信息或剧情摘要信息。

4、您的应用可以提供一种方式，让用户可以在观看视频时将其他内容加入播放队列。当主屏幕显示内容选择 Activity 时，视频会继续以画中画模式播放。

1、如何切换到PIP模式：

- 首先声明对画中画的支持，清单文件中对对应Activity设置android:supportsPictureInPicture为true

```
<activity
    android:name=".pic.PicInPicActivity"
    android:resizeableActivity="true"
    android:supportsPictureInPicture="true"
    android:theme="@style/AppCompatTheme" />
```

- 为了防止在画中画模式转换期间发生布局更改，设置一下android:configChanges属性

```
android:configChanges="screenSize|smallestScreenSize|screenLayout|orientation"
```

- 在点击按钮进入画中画模式之前，先判断下是否可以使用画中画模式，因为低内存设备可能无法使用画中画模式

```
boolean isSupportPipMode = getPackageManager().hasSystemFeature(PackageManager.FEATURE_PICTURE_IN_PICTURE) && Build.VERSION.SDK_INT >= Build.VERSION_CODES.N
```

- 点击按钮切换到画中画模式，要进入画中画模式，Activity 必须调用 enterPictureInPictureMode()

```
// 进入画中画模式，注意enterPictureInPictureMode是Android8.0之后新增的方法
enterPictureInPictureMode();
```

2、画中画模式的参数构建器介绍

画中画模式中，我们可通过它的参数构建器实现窗口宽高比例的设置以及添加窗口控件

- 创建画中画模式的参数构建器
- 设置宽高比例值，第一个参数表示分子，第二个参数表示分母
- setAspectRatio设置画中画窗口的宽高比例
- setActions设置控件

```
// 创建画中画模式的参数构建器
PictureInPictureParams.Builder builder = null;
builder = new PictureInPictureParams.Builder();
final PendingIntent intent = PendingIntent.getBroadcast(
                PicInPicActivity.this,
                2,
                new Intent(ACTION_MEDIA_CONTROL).putExtra(EXTRA_CONTROL_TYPE, 2),
                0);
final Icon icon = Icon.createWithResource(PicInPicActivity.this, R.drawable.btn_pause);
final ArrayList<RemoteAction> actions = new ArrayList<>();
actions.add(new RemoteAction(icon, "title", "title", intent));
actions.add(new RemoteAction(
                Icon.createWithResource(PicInPicActivity.this, R.drawable.btn_pause),
                getString(R.string.info),
                getString(R.string.info_description),
                PendingIntent.getActivity(
                        PicInPicActivity.this,
                        3,
                        new Intent(
                                Intent.ACTION_VIEW,
                                Uri.parse(getString(R.string.info_uri))),
                        0)));
builder.setActions(actions);
// 设置宽高比例值，第一个参数表示分子，第二个参数表示分母
// 下面的10/5=2，表示画中画窗口的宽度是高度的两倍
Rational aspectRatio = new Rational(10, 5);
// 设置画中画窗口的宽高比例
builder.setAspectRatio(aspectRatio);
// 进入画中画模式，注意enterPictureInPictureMode是Android8.0之后新增的方法
enterPictureInPictureMode(builder.build());
```


进入 PIP 模式的最常见流程如下：

1. 从按钮触发

*** onClicked (View)，onOptionsItemSelected (MenuItem) 等等。**

2. 有意的离开您的应用程序触发

*** onUserLeaveHint ( )**

3. 从返回触发

*** onBackPressed ( )**

3、处理画中画模式下的界面元素

当 Activity 进入或退出画中画模式时，系统会调用 onPictureInPictureModeChanged()，可以在这个回调中做一些界面元素的交互处理

```
// 在进入画中画模式/退出画中画模式时触发
public void onPictureInPictureModeChanged(boolean isInPicInPicMode, Configuration newConfig) {
    Log.e( tag: "xtt",  msg: "onPictureInPictureModeChanged isInPicInPicMode=" + isInPicInPicMode+"::Configuration::"+newConfig.orientation);
    super.onPictureInPictureModeChanged(isInPicInPicMode, newConfig);
    if (isInPicInPicMode) { // 进入画中画模式，则隐藏除视频画面之外的其它控件
        ll_btn.setVisibility(View.GONE);
        vc_play.setVisibility(View.GONE);
    } else { // 退出画中画模式，则显示除视频画面之外的其它控件
        ll_btn.setVisibility(View.VISIBLE);
        vc_play.setVisibility(View.VISIBLE);
    }
}
```


4、继续播放视频

在画中画模式下继续播放视频

当您的 activity 切换到画中画模式时，系统会将该 activity 置于暂停状态并调用 activity 的 `onPause()` 方法。当 activity 在画中画模式下暂停时，视频播放不得暂停，而应继续播放。

在 Android 7.0 及更高版本中，当系统调用 activity 的 `onStop()` 时，您应暂停视频播放；当系统调用 activity 的 `onStart()` 时，您应恢复视频播放。这样一来，您就无需在 onPause() 中检查应用是否处于画中画模式，只需继续播放视频即可。

如果您必须在 `onPause()` 实现中暂停播放，请通过调用 `isInPictureInPictureMode()` 检查是否处于画中画模式并相应地处理播放情况，例如：

Kotlin Java

```
@Override
public void onPause() {
    // If called while in PiP mode, do not pause playback
    if (isInPictureInPictureMode()) {
        // Continue playback
        ...
    } else {
        // Use existing playback logic for paused Activity behavior.
        ...
    }
}
```

当您的 activity 从画中画模式切换回全屏模式时，系统会恢复您的 activity 并调用 `onResume()` 方法。

在画中画模式中使用单个播放 activity

在您的应用中，可能会出现以下情况：有一个视频播放 activity 正处于画中画模式，用户在主屏幕上浏览内容时选择了新的视频。应以全屏模式在现有的播放 activity 中播放新的视频，而不是启动可能会令用户感到困惑的新 activity。

如要确保将单个 activity 用于视频播放请求并根据需要进入或退出画中画模式，请在清单中将 activity 的 `android:launchMode` 设置为 `singleTask`：

```
<activity android:name="VideoActivity"
    ...
    android:supportsPictureInPicture="true"
    android:launchMode="singleTask"
    ...
```

在您的 activity 中，替换 `onNewIntent()` 并处理新的视频，从而根据需要停止任何现有的视频播放。

最佳做法：

最佳做法

低 RAM 设备可能无法使用画中画模式。在应用使用画中画之前，请务必通过调用 `hasSystemFeature(PackageManager.FEATURE_PICTURE_IN_PICTURE)` 进行检查以确保可以使用画中画。

画中画旨在用于播放全屏视频的 activity。将 activity 切换到画中画模式时，请避免显示视频内容以外的任何内容。跟踪您的 activity 何时进入画中画模式并隐藏界面元素，如处理画中画模式下的界面元素中所述。

当 activity 进入画中画模式后，默认不会获得输入焦点。如需在画中画模式下接收输入事件，请使用 `MediaSession.setCallback()`。如需详细了解如何使用 `setCallback()`，请参阅显示“闻曲知音”卡片。

当您的应用处于画中画模式时，画中画窗口中的视频播放可能会对其他应用（例如，音乐播放器应用或语音搜索应用）造成音频干扰。为避免出现此问题，请在开始播放视频时请求音频焦点，并处理音频焦点更改通知，如管理音频焦点中所述。如果您在处于画中画模式时收到音频焦点丢失通知，请暂停或停止视频播放。

当您的应用即将进入画中画模式时，请注意，只有顶层 activity 才会进入画中画模式。在某些情况下（例如在多窗口设备上），此时系统可能会显示下层 activity，在画中画 activity 旁边，您可能会再次看到下层 activity。您应相应地处理这种情况，包括以下 activity 获取 `onResume()` 或 `onPause()` 回调。用户也有可能与该 activity 互动。例如，如果您的视频列表 activity 正在显示，视频播放 activity 处于画中画模式，用户可能会从列表中选择新视频，画中画 activity 应相应地进行更新。

进阶使用

1、通过MediaSession达到效果

2、自定义按钮 (推荐)

(注意,按钮不超过三个,位置不可调节)

```
void updatePictureInPictureActions(@DrawableRes int iconId, String title, int controlType, int requestCode) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        if (mPictureInPictureParamsBuilder == null) {
            mPictureInPictureParamsBuilder = new PictureInPictureParams.Builder();
        }
        final ArrayList<RemoteAction> actions = new ArrayList<>();
        // This is the PendingIntent that is invoked when a user clicks on the action item.  You need to use distinct


        //上一个
        final PendingIntent intentLast = PendingIntent.getBroadcast(this, REQUEST_TYPE_NEXT, new Intent(ACTION_MEDIA_C
        actions.add(new RemoteAction(Icon.createWithResource(this, R.drawable.gsy_play_video_icon_last), "", "", inter
        //暂停/播放
        final PendingIntent intentPause = PendingIntent.getBroadcast(this, requestCode, new Intent(ACTION_MEDIA_CONTRO
        actions.add(new RemoteAction(Icon.createWithResource(this, iconId), title, title, intentPause));
        //下一个
        final PendingIntent intentNext = PendingIntent.getBroadcast(this, REQUEST_TYPE_LAST, new Intent(ACTION_MEDIA_C
        actions.add(new RemoteAction(Icon.createWithResource(this, R.drawable.gsy_play_video_icon_next), "", "", inter

        mPictureInPictureParamsBuilder.setActions(actions);

        // This is how you can update action items (or aspect ratio) for Picture-in-Picture mode. Note this call can h
        setPictureInPictureParams(mPictureInPictureParamsBuilder.build());
    }
}
```

画中画框架解析

多窗口框架的核心思想是分栈和设置栈边界

栈：

既然提到了分栈，那我们首先要了解这个栈是什么？在Android系统中，启动一个Activity之后，必定会将此Activity存放于某一个Stack，在Android N中，系统定义了5种Stack ID，系统所有Stack的ID属于这5种里面的一种。不同的Activity可能归属于不同的Stack，但是具有相同的Stack ID。StackID如下图所示：

```
/** First static stack ID. */ 第一个静态栈ID
public static final int FIRST_STATIC_STACK_ID = 0;
    主页活动堆栈ID
/** Home activity stack ID. */
public static final int HOME_STACK_ID = FIRST_STATIC_STACK_ID;
    全屏活动通常启动到的堆栈的ID
/** ID of stack where fullscreen activities are normally launched into. */
public static final int FULLSCREEN_WORKSPACE_STACK_ID = 1;
    自由形式/调整大小的活动通常启动到的堆栈的ID。
/** ID of stack where freeform/resized activities are normally launched into. */
public static final int FREEFORM_WORKSPACE_STACK_ID = FULLSCREEN_WORKSPACE_STACK_ID + 1;
    占用屏幕专用区域的堆栈的ID
/** ID of stack that occupies a dedicated region of the screen. */
public static final int DOCKED_STACK_ID = FREEFORM_WORKSPACE_STACK_ID + 1;
    存在时总是在顶部(总是可见)的堆栈的ID。
/** ID of stack that always on top (always visible) when it exist. */
public static final int PINNED_STACK_ID = DOCKED_STACK_ID + 1;
```

正常情况下，Launcher和SystemUI进程里面的Actiivty所在的Stack的id是HOME_STACK_ID(FIRST_STATIC_STACK_ID);

普通的Activity所在的Stack的id是 FULLSCREEN_WORKSPACE_STACK_ID;

自由模式下对应的栈ID是FREEFORM_WORKSPACE_STACK_ID(FULLSCREEN_WORKSPACE_STACK_ID+1);

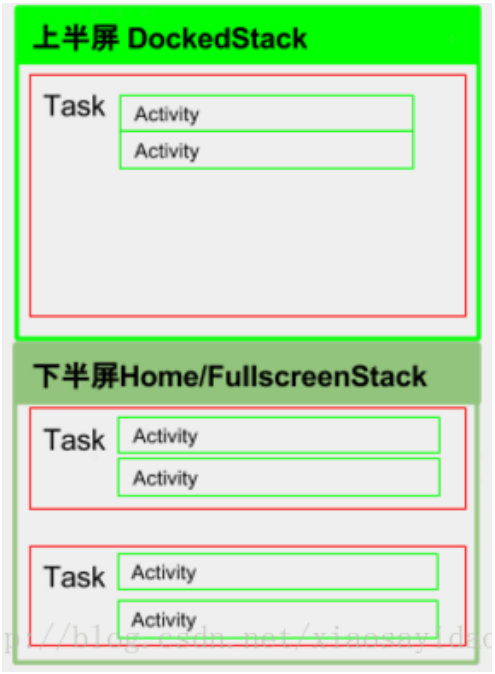
分屏模式下，上半部分窗口里面的Activity所处的栈ID是DOCKED_STACK_ID;

画中画模式中，位于小窗口里面的Activity所在的栈ID是PINNED_STACK_ID;

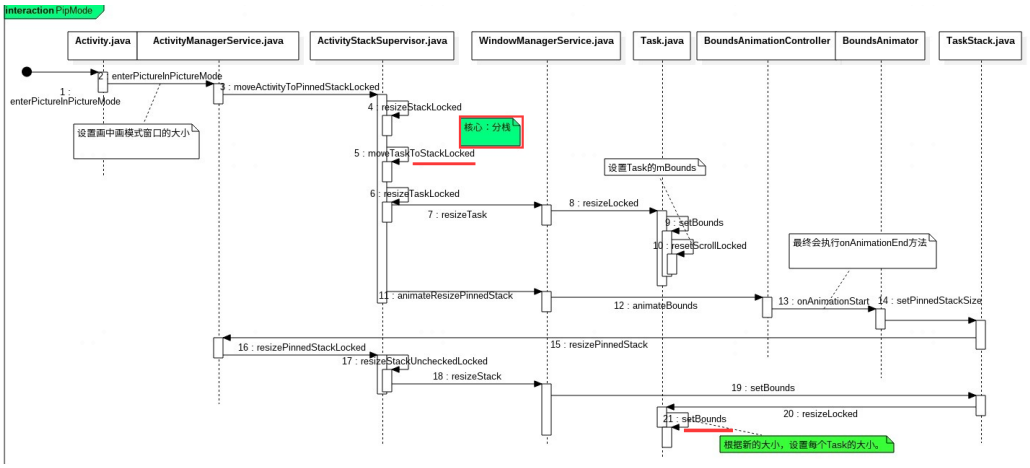
栈边界

在多窗口框架中，通过设置Stack的边界（Bounds）来控制里面每个Task的大小，最终Task的大小决定了窗口的大小。栈边界通过Rect(left,top,right,bottom)来表示，存储了四个值，分别表示矩形的4条边离坐标轴的位置，最终显示在屏幕上窗口的大小是根据Stack边界的大小来决定的。

分屏模式下的Activity的状态。整个屏幕被分成了两个Stack，一个DockedStack，一个FullScreenStack。每个Stack里面有多个Task，每个Task里面又有多个Activity。当我们设置了Stack的大小之后，Stack里面的所有的Task的大小以及Task里面所有的Activity的窗口大小都确定了。假设屏幕的大小是1440x2560，整个屏幕的栈边界就是（0,0,1440,2560):



画中画模式的框架原理



PIP模式分栈

Step1-5

PIP模式下分栈核心代码，后面的步骤是设置栈边界的核心代码。

如前面所说，系统有5种Stack ID，PIP模式中的Activity所在的stack id是PINNED_STACK_ID。普通Activity位于id是FULLSCREEN_WORKSPACE_STACK_ID的stack里面。因此画中画模式分栈的核心工作是把activity从id是FULLSCREEN_WORKSPACE_STACK_ID的栈移动到id是PINNED_STACK_ID的stack里面。

```
/**
 * Puts the activity in picture-in-picture mode if possible in the current system state. Any
 * prior calls to {@link #setPictureInPictureParams(PictureInPictureParams)} will still apply
 * when entering picture-in-picture through this call.
 *
 * @see #enterPictureInPictureMode(PictureInPictureParams)
 * @see android.R.attr#supportsPictureInPicture 这个方法代替
 */
@Deprecated
public void enterPictureInPictureMode() {  启用
    enterPictureInPictureMode(new PictureInPictureParams.Builder().build());
}
```

但这里还是调用的enterPictureInPictureMode:

```
/**
 * Puts the activity in picture-in-picture mode if the activity supports.
 * @see android.R.attr#supportsPictureInPicture
 * @hide
 */
@Override
public void enterPictureInPictureModeIfPossible() {
    if (mActivityInfo.supportsPictureInPicture()) {
        enterPictureInPictureMode();
    }
}
```

只要设置了画中画标志即可返回true:

```
@UnsupportedAppUsage
public boolean supportsPictureInPicture() {
    return (flags & FLAG_SUPPORTS_PICTURE_IN_PICTURE) != 0;
}
```

```
public boolean enterPictureInPictureMode(@NonNull PictureInPictureParams params) {
    if (!deviceSupportsPictureInPictureMode()) {  设备是否支持PIP
        return false;
    }
    if (params == null) {
        throw new IllegalArgumentException("Expected non-null picture-in-picture params");
    }
    if (!mCanEnterPictureInPicture) {
        throw new IllegalStateException("Activity must be resumed to enter"
                + " picture-in-picture");
    }
    // Set mIsInPictureInPictureMode earlier and don't wait for
    // onPictureInPictureModeChanged callback here. This is to ensure that
    // isInPictureInPictureMode returns true in the following onPause callback.
    // See https://developer.android.com/guide/topics/ui/picture-in-picture for guidance.
    mIsInPictureInPictureMode = ActivityClient.getInstance().enterPictureInPictureMode(
            mToken, params);
    return mIsInPictureInPictureMode;
}
```

```
boolean enterPictureInPictureMode(IBinder token, PictureInPictureParams params) {
    try {
        return getActivityClientController().enterPictureInPictureMode(token, params);
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

```
@Override
public boolean enterPictureInPictureMode(IBinder token, final PictureInPictureParams params) {
    final long origId = Binder.clearCallingIdentity();
    try {
        synchronized (mGlobalLock) {
            final ActivityRecord r = ensureValidPictureInPictureActivityParams(
                    "enterPictureInPictureMode", token, params);
            return mService.enterPictureInPictureMode(r, params);
        }
    } finally {
        Binder.restoreCallingIdentity(origId);
    }
}
```

ATMS

检查系统状态和与给定 {@param token} 关联的活动，以验证该活动是否支持画中画，如果所有检查都通过，则返回给定 {@param token} 的活动记录：

```
     */
    private ActivityRecord ensureValidPictureInPictureActivityParams(String caller,
            IBinder token, PictureInPictureParams params) {
        if (!mService.mSupportsPictureInPicture) {
            throw new IllegalStateException(caller
                    + ": Device doesn't support picture-in-picture mode.");
        }

        final ActivityRecord r = ActivityRecord.forTokenLocked(token);
        if (r == null) {
            throw new IllegalStateException(caller
                    + ": Can't find activity for token=" + token);
        }

        return r;
    }
```

根据token拿到ActiivtyRecord

```
    static @Nullable ActivityRecord forTokenLocked(IBinder token) {
        if (token == null) return null;
        final Token activityToken;
        try {
            activityToken = (Token) token;
        } catch (ClassCastException e) {
            Slog.w(TAG, "Bad activity token: " + token, e);
            return null;
        }
        final ActivityRecord r = activityToken.mActivityRef.get();
        return r == null || r.getRootTask() == null ? null : r;
    }
```

ActivityRecord.java （已生成）

```
1873    boolean isFullyTransparentBarAllowed(Rect rect) {
1874        return mLetterboxUiController.isFullyTransparentBarAllowed(rect);
1875    }
1876
1877    private static class Token extends Binder {
1878        @NonNull WeakReference<ActivityRecord> mActivityRef;
1879
1880        @Override
1881        public String toString() {
1882            return "Token{" + Integer.toHexString(System.identityHashCode(this)) + " "
1883                    + mActivityRef.get() + "}";
1884        }
1885    }
```

该引用指向的该类的ActiivtyRecord

ensureValidPictureInPictureActivityParams 方法作用就是通过传入的token，去获取到token对应的ActivityRecord对象，判断该对象是否支持PIP，支持就返回这个

token对应的ActivityRecord

拿到了ActivityRecord之后，调用了ATMS.enterPictureInPictureMode(r,params):

```
boolean enterPictureInPictureMode(@NonNull ActivityRecord r, PictureInPictureParams params) {
    // If the activity is already in picture in picture mode, then just return early
    if (r.inPinnedWindowingMode()) {   如果这个页面本身就是PIP，那尽早返回
        return true;
    }

    // Activity supports picture-in-picture, now check that we can enter PiP at this
    // point, if it is      页面支持PIP，检查线是否可以进入PIP
    if (!r.checkEnterPictureInPictureState("enterPictureInPictureMode",
            false /* beforeStopping */)) {
        return false;
    }
                                        Thread
    final Runnable enterPipRunnable = () -> {
        synchronized (mGlobalLock) {
            if (r.getParent() == null) {没有父窗口就直接返回
                Slog.e(TAG, "Skip enterPictureInPictureMode, destroyed " + r);
                return;
            }                   设置PIP模式参数
            r.setPictureInPictureParams(params);          核心方法，移动栈到PINNED_STACK中
            mRootWindowContainer.moveActivityToPinnedRootTask(r,
                    null /* launchIntoPipHostActivity */, "enterPictureInPictureMode");
            final Task task = r.getTask();
            // Continue the pausing process after entering pip.
            if (task.getPausingActivity() == r) {  进入pip模式过程中继续暂停过程
                task.schedulePauseActivity(r, false /* userLeaving */,
                        false /* pauseImmediately */, "auto-pip");
            }
        }
    };

    if (r.isKeyguardLocked()) {
        // If the keyguard is showing or occluded, then try and dismiss it before
        // entering picture-in-picture (this will prompt the user to authenticate if the
        // device is currently locked).
        mActivityClientController.dismissKeyguard(r.token, new KeyguardDismissCallback() {
            @Override
            public void onDismissSucceeded() {
                mH.post(enterPipRunnable);   如果锁屏状态，尝试解锁进入PIP，如果有密码等，需要用户解锁
            }
        }, null /* message */);
    } else {
        // Enter picture in picture immediately otherwise
        enterPipRunnable.run();   执行线程
    }
    return true;
}
```

多窗口的核心是分stack，以上方法的最后一句话会把当前Activity移动到系统为PIP分配的stack。接下来到moveActivityToPinnedStackLocked里面，默认情况下PinnedStack不存在，系统会创建这个stack，然后会根据当前Activity（正常窗口）所在的task的边界来设置PinnedStack的边界，注意此时还没有用到我们默认为PIP指定的bounds，当前activity的边界就是屏幕的可视区域，最终在WindowManagerService.java里面我们会把当前的task添加到PIP模式所在的Stack里面。

```
void moveActivityToPinnedStackLocked(ActivityRecord r, String reason, Rect bounds) {
    mWindowManager.deferSurfaceLayout();
    try {
        final TaskRecord task = r.task;

        if (r == task.stack.getVisibleBehindActivity()) {
            // An activity can't be pinned and visible behind at the same time. Go ahead and
            // release it from been visible behind before pinning.
            requestVisibleBehindLocked(r, false);
        }

        // Need to make sure the pinned stack exist so we can resize it below...
        final ActivityStack stack = getStack(PINNED_STACK_ID, CREATE_IF_NEEDED, ON_TOP);

        // Resize the pinned stack to match the current size of the task the activity we are
        // going to be moving is currently contained in. We do this to have the right starting
        // animation bounds for the pinned stack to the desired bounds the caller wants.
        resizeStackLocked(PINNED_STACK_ID, task.mBounds, null /* tempTaskBounds */,
                null /* tempTaskInsetBounds */, !PRESERVE_WINDOWS,
                true /* allowResizeInDockedMode */, !DEFER_RESUME);

        if (task.mActivities.size() == 1) {
            // There is only one activity in the task. So, we can just move the task over to
            // the stack without re-parenting the activity in a different task.
            if (task.getTaskToReturnTo() == HOME_ACTIVITY_TYPE) {
                // Move the home stack forward if the task we just moved to the pinned stack
                // was launched from home so home should be visible behind it.
                moveHomeStackToFront(reason);
            }
            moveTaskToStackLocked(
                    task.taskId, PINNED_STACK_ID, ON_TOP, FORCE_FOCUS, reason, !ANIMATE);
        } else {
            stack.moveActivityToStack(r);
        }
    } finally {
        mWindowManager.continueSurfaceLayout();
    }

    // The task might have already been running and its visibility needs to be synchronized
    // with the visibility of the stack / windows.
    ensureActivitiesVisibleLocked(null, 0, !PRESERVE_WINDOWS);
    resumeFocusedStackTopActivityLocked();

    mWindowManager.animateResizePinnedStack(bounds, -1);
    mService.notifyActivityPinnedLocked();
}
```

至此分栈的过程就完成了。

PIP模式设置栈边界

接着分栈的分析，最后会调用WindowManager的animateResizePinnedStack(bounds, -1)方法，根据当前Stack的大小和指定的PIP窗口的边界，通过动画慢慢更改当前窗口的大小，直到最后显示画中画模式的窗口。

```
    public void animateResizePinnedStack(final Rect bounds, final int animationDuration) {
        synchronized (mWindowMap) {
...
            UiThread.getHandler().post(new Runnable() {
                @Override
                public void run() {
                    mBoundsAnimationController.animateBounds(
                            stack, originalBounds, bounds, animationDuration);
                }
            });
        }
    }
```

mBoundsAnimationController.animateBounds的from和to参数，分别表示在全屏stack id下的栈边界和指定的PIP模式的栈边界。

```
    void animateBounds(final AnimateBoundsUser target, Rect from, Rect to, int animationDuration) {
...
        final BoundsAnimator animator =
                new BoundsAnimator(target, from, to, moveToFullscreen, replacing);
        mRunningAnimations.put(target, animator);
        animator.setFloatValues(0f, 1f);
        animator.setDuration((animationDuration != -1 ? animationDuration
                : DEFAULT_APP_TRANSITION_DURATION) * DEBUG_ANIMATION_SLOW_DOWN_FACTOR);
        animator.setInterpolator(new LinearInterpolator());
        animator.start();
    }
```

在动画的执行过程中，不断的去更改当前stack的大小。

```
    @Override
    public void onAnimationUpdate(ValueAnimator animation) {
        // ...
        if (!mTarget.setPinnedStackSize(mTmpRect, mTmpTaskBounds)) {
        // ...
        }
    }
```

省略掉中间的一些步骤。直接到ActivityStackSupervisor.java的resizeStackUncheckedLocked。由于我们的Stack将要发生变化，所以会更新当前stack里面的所有task的相关配置。且会通知应用当前的多窗口状态发生了变化，此时会更新Task对应的最小宽度和最小高度等config信息。

```
void resizeStackUncheckedLocked(ActivityStack stack, Rect bounds, Rect tempTaskBounds,
        Rect tempTaskInsetBounds) {
    bounds = TaskRecord.validateBounds(bounds);

    if (!stack.updateBoundsAllowed(bounds, tempTaskBounds, tempTaskInsetBounds)) {
        return;
    }

    mTmpBounds.clear();
    mTmpConfigs.clear();
    mTmpInsetBounds.clear();
    final ArrayList<TaskRecord> tasks = stack.getAllTasks();
    final Rect taskBounds = tempTaskBounds != null ? tempTaskBounds : bounds;
    final Rect insetBounds = tempTaskInsetBounds != null ? tempTaskInsetBounds : taskBounds;
    for (int i = tasks.size() - 1; i >= 0; i--) {
        final TaskRecord task = tasks.get(i);
        if (task.isResizeable()) {
            if (stack.mStackId == FREEFORM_WORKSPACE_STACK_ID) {
                // 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.
                tempRect2.set(task.mBounds);
                fitWithinBounds(tempRect2, bounds);
                task.updateOverrideConfiguration(tempRect2);
            } else {
                task.updateOverrideConfiguration(taskBounds, insetBounds);
            }
        }

        mTmpConfigs.put(task.taskId, task.mOverrideConfig);
        mTmpBounds.put(task.taskId, task.mBounds);
        if (tempTaskInsetBounds != null) {
            mTmpInsetBounds.put(task.taskId, tempTaskInsetBounds);
        }
    }

    // We might trigger a configuration change. Save the current task bounds for freezing.
    mWindowManager.prepareFreezingTaskBounds(stack.mStackId);
    stack.mFullscreen = mWindowManager.resizeStack(stack.mStackId, bounds, mTmpConfigs,
            mTmpBounds, mTmpInsetBounds);
    stack.setBounds(bounds);
}
```

接下来我们会进入到设置Stack大小变化的最后一步。设置当前Stack的大小

```
public boolean resizeStack(int stackId, Rect bounds,
        SparseArray<Configuration> configs, SparseArray<Rect> taskBounds,
        SparseArray<Rect> taskTempInsetBounds) {
    synchronized (mWindowMap) {
        final TaskStack stack = mStackIdToStack.get(stackId);
        if (stack == null) {
            throw new IllegalArgumentException("resizeStack: stackId " + stackId
                    + " not found.");
        }
        if (stack.setBounds(bounds, configs, taskBounds, taskTempInsetBounds)
                && stack.isVisibleLocked()) {
            stack.getDisplayContent().layoutNeeded = true;
            mWindowPlacerLocked.performSurfacePlacement();
        }
        return stack.getRawFullscreen();
    }
}
```

可以看到当前设置的是TaskStack的边界。

```
boolean setBounds(
        Rect stackBounds, SparseArray<Configuration> configs, SparseArray<Rect> taskBounds,
        SparseArray<Rect> taskTempInsetBounds) {
    setBounds(stackBounds);

    // Update bounds of containing tasks.
    for (int taskNdx = mTasks.size() - 1; taskNdx >= 0; --taskNdx) {
        final Task task = mTasks.get(taskNdx);
        Configuration config = configs.get(task.mTaskId);
        if (config != null) {
            Rect bounds = taskBounds.get(task.mTaskId);
            if (task.isTwoFingerScrollMode()) {
                // This is a non-resizeable task th[illegible]ked (or side-by-side to the docked
                // stack). It might have been scrolled previously, and after the stack resizing,
                // it might no longer fully cover the stack area.
                // Save the old bounds and re-apply the scroll. This adjusts the bounds to
                // fit the new stack bounds.
                task.resizeLocked(bounds, config, false /* forced */);
                task.getBounds(mTmpRect);
                task.scrollLocked(mTmpRect);
            } else {
                task.resizeLocked(bounds, config, false /* forced */);
                task.setTempInsetBounds(
                        taskTempInsetBounds != null ? taskTempInsetBounds.get(task.mTaskId)
                                : null);
            }
        } else {
            Slog.wtf(TAG_WM, "No config for task: " + task + ", is there a mismatch with AM?");
        }
    }
    return true;
}
```

在setBounds里面会更新当前TaskStack的bounds，接下来会更新TaskStack里面所有的Task的边界。

在task.resizeLocked里面，会最终设置Task的mBounds变量。也就是我们本文介绍的Task边界。至此，Task的边界bounds已经设置完毕。

Z~Order

WMS分几步完成Z-Order的排序：

1、建立窗口的时候为每个窗口分配BaseLayer和SubLayer(addWindow)。

2、按Z-Order的顺序将窗口加入到所在屏幕的窗口列表中（DisplayContent.windows)

3、在显示的时候，动态计算窗口的Layer，最终决定显示Z-Order。

当添加 Window 的时候已经确定好了 Window 的层级，显示的时候才会根据当前的层级确定 Window 应该在哪一层显示

1. BaseLayer 和SubLayer

WMS通过Layer来确定window的Z-Order,WMS为每个Window 分配一个Layer值，值越高，显示在越上面。

每个WindowState在创建的时候WMS时，都会分配一个baseLayer, 记录在WindowState.mBaseLayer中。子窗口还会对应有一个WindowState.mSubLayer值。

主窗口计算方式：

PhoneWindowManager.windowTypeToLayerLw（win.type）* TYPE_LAYER_MULTIPLIER(10000) + TYPE_LAYER_OFFSET(1000)。

所有相同类型的窗口具有相同的baseLayer值。例如：application类型的窗口对用的layer为2，那么BaseLayer值为2*10000+1000=21000。

windowTypeToLayerLw 这个方法是针对不同的窗口类型做了简单的映射。具体的映射则取决于的该设备所采用的WindowManagerPolicy。

子窗口计算方式：

PhoneWindowManager.subWindowTypeToLayerLw(a.type)。

主窗口该值为0。

可以看到上述两个PhoneWindowManager函数都会要求win.type作为参数。不同的窗口类型layer的定义看@PhoneWindowManager.java:

```
static final int APPLICATION_LAYER = 2;
static final int PHONE_LAYER = 3;
static final int SEARCH_BAR_LAYER = 4;
static final int SYSTEM_DIALOG_LAYER = 5;
// toasts and the plugged-in battery thing
static final int TOAST_LAYER = 6;
```

对于子窗口类型layer定义如下，在Z-Order排序时正值的子窗口在主窗口上面，负值的子窗口在主窗口下面。

```
static final int APPLICATION_MEDIA_SUBLAYER = -2;
static final int APPLICATION_MEDIA_OVERLAY_SUBLAYER = -1;
static final int APPLICATION_PANEL_SUBLAYER = 1;
static final int APPLICATION_SUB_PANEL_SUBLAYER = 2;
```

2. 按Z-Order的顺序将窗口加入到所在屏幕的窗口列表中

对于新加入App的窗口，WMS会通过一定的顺序规则来添加窗口到相应的DisplayContent.mWindow中，前面说到DisplayContent.mWindow中的窗口是以z-Order排序的，最后面的显示在最前面（值越大，越靠近用户）。

添加窗口到相应的*DisplayContent*.mWindow的函数为
addWindowToListInOrderLocked

参考连接：
https://developer.android.google.cn/guide/topics/ui/picture-in-picture#handling_ui
https://www.jianshu.com/p/1765480fb759
https://blog.csdn.net/XAVI_2010/article/details/104637720
https://www.jianshu.com/p/19934892a235
https://blog.csdn.net/u011200604/article/details/104701266
https://blog.csdn.net/sunmmer123/article/details/118358879

https://blog.csdn.net/xiaosayidao/article/details/75045087

深入理解Andriod内核思想