**Launcher启动过程**

       此前已经学习了 Android 系统启动流程的 i n it 进程、 Zygote 进程和 SystemServer 进程， 最后我们来学习 La unch er 的启动过程。这篇文章主要分析Android8.1系统的Launcher的启动流程。

    1 Launcher 概述

       系统启动的最后一步是启动一个应用程序用来显示系统中已经安装的应用程序，也就是我们手机看到的桌面，这个应用程序就叫作 Launcher。Launcher 在启动过程中会请求 PackageManagerService 返回系统中已经安装的应用程序的信息，并将这些信息封装成一个快捷图标列表显示在系统屏幕上， 这样用户可以通过点击这些快捷图标来启动相应的应用程序。

       通俗来讲 Launcher 就是 Android 系统的桌面，它的作用主要有以下两点：

       (1 ）作为 Android 系统的启动器，用于启动应用程序

       (2 ）作为 Android 系统的桌面，用于显示和管理应用程序的快捷图标或者其他桌面组件。

    2 Launcher 启动过程介绍

       SystemServer 进程在启动的过程中会启动ActivityManagerServer、PackageManagerService服务, PackageManagerService启动之后会将系统中的应用程序安装完成。AMS(ActivityManagerServer) 会将 Launcher 启动起来。

       启动Launcher的入口为AMS的sytemReady方法，它在SystemServer的startOtherServices方法中被调用，代码如下所示：

**frameworks/base/services/java/com/android/server/SystemServer.java**

private void startOtherServices() {

mActivityManagerService.systemReady(() -> {

Slog.i(TAG, "Making services ready");

traceBeginAndSlog("StartActivityManagerReadyPhase");

mSystemServiceManager.startBootPhase(

SystemService.PHASE\_ACTIVITY\_MANAGER\_READY);

...

}, BOOT\_TIMINGS\_TRACE\_LOG);

}

         在Android8.0开始在源码中引入了Java Lambda表达式，接下来分析AMS的systemReady方法做了什么，代码如下：

**frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java**

public void systemReady(final Runnable goingCallback, TimingsTraceLog traceLog) {

...

synchronized (this) {

...

startHomeActivityLocked(currentUserId, "systemReady"); // ... 1

...

mStackSupervisor.resumeFocusedStackTopActivityLocked();// ... 2

mUserController.sendUserSwitchBroadcastsLocked(-1, currentUserId);

}

}

**特别注意！！！**

**在网络上有很多博客中基于Android7.0以及更高系统版本分析Launcher启动过程时，说是在ActivityManagerService的systemReady方法中使用的是注释1处的代码调用startHomeActivityLocked方法启动Launcher的，这种说法是错误的。其实是注释2处的代码通过一系列的调用最后还是调用到AMS的startHomeActivityLocked方法启动Launcher的。**

**为了验证这个结果，我们来修改Android8.1的AMS代码，然后编译系统，查看Log日志，对AMS修改如下：**

**frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java**

boolean startHomeActivityLocked(int userId, String reason) {

Slog.wtf("ActivityManagerService", "reason = " + reason +

" ; mFactoryTest =" + mFactoryTest

+ " ; mTopAction = " + mTopAction

+ " ; userId = " + userId); // ... 1

if (mFactoryTest == FactoryTest.FACTORY\_TEST\_LOW\_LEVEL

&& mTopAction == null) {

// We are running in factory test mode, but unable to find

// the factory test app, so just sit around displaying the

// error message and don't try to start anything.

return false;

}

Intent intent = getHomeIntent();

ActivityInfo aInfo = resolveActivityInfo(intent, STOCK\_PM\_FLAGS, userId);

Slog.wtf("ActivityManagerService", "reason = " + reason

+ " ; aInfo = " + aInfo); // ... 2

if (aInfo != null) {

intent.setComponent(new ComponentName(aInfo.applicationInfo.packageName, aInfo.name));

Slog.wtf("ActivityManagerService", "reason = " + reason

+ " ; aInfo.applicationInfo.packageName = " + aInfo.applicationInfo.packageName); // ... 3

// Don't do this if the home app is currently being

// instrumented.

aInfo = new ActivityInfo(aInfo);

aInfo.applicationInfo = getAppInfoForUser(aInfo.applicationInfo, userId);

ProcessRecord app = getProcessRecordLocked(aInfo.processName,

aInfo.applicationInfo.uid, true);

if (app == null || app.instrumentationClass == null) {

intent.setFlags(intent.getFlags() | Intent.FLAG\_ACTIVITY\_NEW\_TASK);

mActivityStarter.startHomeActivityLocked(intent, aInfo, reason);

}

} else {

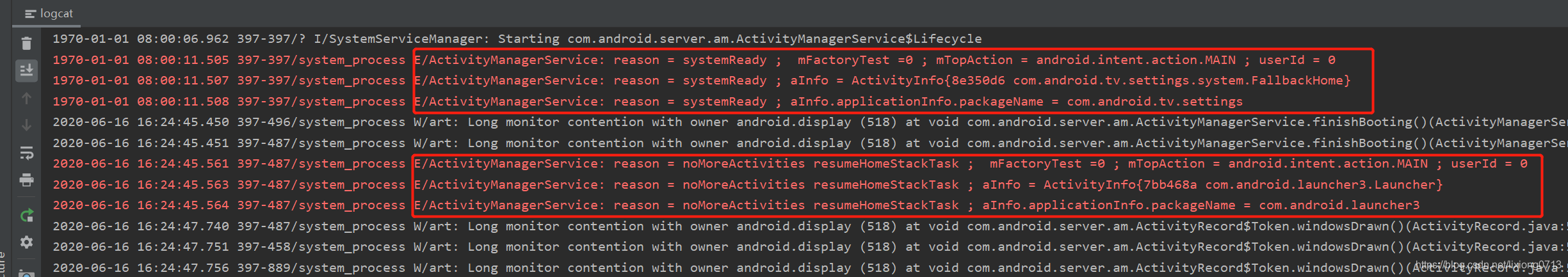
Slog.wtf(TAG, "No home screen found for " + intent, new Throwable());

}

return true;

}

**我们在AMS的** **startHomeActivityLocked方法中增加了注释1 、2 、3 处的打印log日志的代码，然后再编译系统（不会编译系统的请自行百度研究，这里就不做讲解），启动模拟器查看日志输出，我们通过日志分析工具查看到如下log日志信息：**

  
        **在AMS的systemReady方法中，注释1处调用startHomeActivityLocked方法是传入的reason参数的值是“systemReady”，在Log日志中我们发现reason为“systemReady”时，所启动的并不是我们的Launcher程序，而是Settings程序的FallbackHome这Activity。注释2处代码通过一系列调用之后最终调用到startHomeActivityLocked方法是传入的reason参数的值是** **“noMoreActivities resumeHomeStackTask” （这个值我们后面具体分析是怎么传入的）时，在Log日志中我们发现reason为“noMoreActivities resumeHomeStackTask”时，所启动的才是我们的Launcher程序。**

**由此可以证明我们所说的结果是正确的。**

       那么为什么AMS的systemReady方法中在注释1处要调用 startHomeActivityLocked方法启动Setting程序的FallbackHome呢？

       其实是Android 7.0引入了Direct Boot模式，当手机已经通电开机但是用户并有解锁锁屏的时候，Android 7.0以后系统会运行于一个安全的模式，也就是Dierect Boot模式。

       Direct Boot模式下，仅限于运行一些关键的、紧急的APP，比如：

* Apps that have scheduled noti cations, such as alarm clock apps.
* Apps that provide important user noti cations, like SMS apps.
* Apps that provide accessibility services, like Talkback.

       在Android 7.0以后，在启动Launcher之前会先启动一个FallbackHome；  FallbackHome是Settings里的一个activity，Settings的android:directBootAware为true，而且FallbackHome在category中配置了Home属性；而Launcher的android:directBootAware为false，所以在DirectBoot模式下，只有FallbackHome可以启动。即先启动com.android.tv.settings/.FallbackHome（或者是com.android.settings/.FallbackHome ） ，待用户解锁后再启动com.android.launcher3/.Launcher。想更深入了解Direct Boot模式和FallbackHome所做的工作的同学可以去阅读源码研究，这里就不做深入说明，下面我们继续分享Lancher的启动过程。

       再回到 ActivityManagerService的 systernReady方法，在注释2处调用了 ActivityStackSupervisor的 resumeFocusedStackTopActivityLocked 方法，代码如下：

**frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java**

boolean resumeFocusedStackTopActivityLocked() {

return resumeFocusedStackTopActivityLocked(null, null, null);

}

boolean resumeFocusedStackTopActivityLocked(

ActivityStack targetStack, ActivityRecord target, ActivityOptions targetOptions) {

if (!readyToResume()) {

return false;

}

if (targetStack != null && isFocusedStack(targetStack)) {

return targetStack.resumeTopActivityUncheckedLocked(target, targetOptions); // ... 1

}

final ActivityRecord r = mFocusedStack.topRunningActivityLocked();

if (r == null || r.state != RESUMED) {

mFocusedStack.resumeTopActivityUncheckedLocked(null, null);

} else if (r.state == RESUMED) {

// Kick off any lingering app transitions form the MoveTaskToFront operation.

mFocusedStack.executeAppTransition(targetOptions);

}

return false;

}

       在注释1 处调用 ActivityStack的 resumeTopActivityUncheckedLocked方法 ， ActivityStack 对象是用来描述 Activity 堆栈 的， resumeTopActivityUncheckedLocked 方法如下所示：

**frameworks/base/services/core/java/com/android/server/am/ActivityStack.java.java**

boolean resumeTopActivityUncheckedLocked(ActivityRecord prev, ActivityOptions options) {

if (mStackSupervisor.inResumeTopActivity) {

// Don't even start recursing.

return false;

}

boolean result = false;

try {

// Protect against recursion.

mStackSupervisor.inResumeTopActivity = true;

result = resumeTopActivityInnerLocked(prev, options); // ... 1

} finally {

mStackSupervisor.inResumeTopActivity = false;

}

// When resuming the top activity, it may be necessary to pause the top activity (for

// example, returning to the lock screen. We suppress the normal pause logic in

// {@link #resumeTopActivityUncheckedLocked}, since the top activity is resumed at the end.

// We call the {@link ActivityStackSupervisor#checkReadyForSleepLocked} again here to ensure

// any necessary pause logic occurs. In the case where the Activity will be shown regardless

// of the lock screen, the call to {@link ActivityStackSupervisor#checkReadyForSleepLocked}

// is skipped.

final ActivityRecord next = topRunningActivityLocked(true /\* focusableOnly \*/);

if (next == null || !next.canTurnScreenOn()) {

checkReadyForSleep();

}

return result;

}

       在注释1调 用了 resume TopActivitylnnerLocked 方法，代码如下所示：

**frameworks/base/services/core/java/com/android/server/am/ActivityStack.java**

private boolean resumeTopActivityInnerLocked(ActivityRecord prev, ActivityOptions options) {

...

if (!hasRunningActivity) {

// There are no activities left in the stack, let's look somewhere else.

return resumeTopActivityInNextFocusableStack(prev, options, "noMoreActivities"); // ... 1

}

...

}

       resumeTopActivitylnnerLocked 方法的 代码很长，这里 仅 截取我 们要分析的关键的部分，注释1处调用resumeTopActivityInNextFocusableStack方法并且传入的reason参数为"noMoreActivities"，代码如下：

**frameworks/base/services/core/java/com/android/server/am/ActivityStack.java**

private boolean resumeTopActivityInNextFocusableStack(ActivityRecord prev,

ActivityOptions options, String reason) {

...

return isOnHomeDisplay() &&

mStackSupervisor.resumeHomeStackTask(prev, reason); // ... 1

}

       在注释1处调用 ActivityStackSupervisor 的resumeHomeStackTask 方法并且传入的reason参数就是"noMoreActivities"，代码如下所示：

**frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java**

boolean resumeHomeStackTask(ActivityRecord prev, String reason) {

...

// reason 的值为noMoreActivities

final String myReason = reason + " resumeHomeStackTask"; // ... 1

// Only resume home activity if isn't finishing.

if (r != null && !r.finishing) {

moveFocusableActivityStackToFrontLocked(r, myReason);

return resumeFocusedStackTopActivityLocked(mHomeStack, prev, null);

}

return mService.startHomeActivityLocked(mCurrentUser, myReason); // ... 2

}

       在注释1处myReason的值被赋值为" noMoreActivities resumeHomeStackTask "，在注释2处mService就是ActivityManagerService，因此 调用了 AMS的 startHomeAc tivityLocke方法并将传入reason参数的值为myReason的值（验证上面Log日志输出的reason值） ，如下所示：

**frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java**

boolean startHomeActivityLocked(int userId, String reason) {

// 判断工厂模式和mTopAction的值

if (mFactoryTest == FactoryTest.FACTORY\_TEST\_LOW\_LEVEL

&& mTopAction == null) { // ... 1

// We are running in factory test mode, but unable to find

// the factory test app, so just sit around displaying the

// error message and don't try to start anything.

return false;

}

// 创建Launcher启动所需的Intent

Intent intent = getHomeIntent(); // ... 2

ActivityInfo aInfo = resolveActivityInfo(intent, STOCK\_PM\_FLAGS, userId); // ... 3

if (aInfo != null) {

intent.setComponent(new ComponentName(aInfo.applicationInfo.packageName, aInfo.name));

// Don't do this if the home app is currently being

// instrumented.

aInfo = new ActivityInfo(aInfo);

aInfo.applicationInfo = getAppInfoForUser(aInfo.applicationInfo, userId);

ProcessRecord app = getProcessRecordLocked(aInfo.processName,

aInfo.applicationInfo.uid, true);

if (app == null || app.instr == null) { // ... 4

intent.setFlags(intent.getFlags() | Intent.FLAG\_ACTIVITY\_NEW\_TASK);

final int resolvedUserId = UserHandle.getUserId(aInfo.applicationInfo.uid);

// For ANR debugging to verify if the user activity is the one that actually

// launched.

final String myReason = reason + ":" + userId + ":" + resolvedUserId;

mActivityStarter.startHomeActivityLocked(intent, aInfo, myReason); // ... 5

}

} else {

Slog.wtf(TAG, "No home screen found for " + intent, new Throwable());

}

return true;

}

       注释1处的 mFactoryTest 代表系统的运行模式，系统的运行模式分为三 种，分别是非工厂模式、低级工厂模式和高级工厂模式， mTopAction 则用来描述第一 个被启动 Activity 组件的 Action ，它的默认值为 Intent.ACTION\_ MAIN ， 因此注释1 处的代码的意思就是 mFactoryTest为FactoryTest.FACTORY \_TEST LOW\_LEVEL （低级工厂模式）并且 mTopAction 等于 null 时，直接返回 false 。 注释2 处的 getHomelntent 方法如下所示：

**frameworks/base/services/core/java/com/android/server/am/ActivityManageService.java**

Intent getHomeIntent() {

Intent intent = new Intent(mTopAction, mTopData != null ? Uri.parse(mTopData) : null);

intent.setComponent(mTopComponent);

intent.addFlags(Intent.FLAG\_DEBUG\_TRIAGED\_MISSING);

if (mFactoryTest != FactoryTest.FACTORY\_TEST\_LOW\_LEVEL) {

intent.addCategory(Intent.CATEGORY\_HOME);

}

return intent;

}

        在getHomelntent 方法 中创建了 Intent ，并将 mTopAction和 mTopData 传入。 mTopAction 的值为 Intent.ACTION\_ MAIN ，并且如果系统运行模式不是低级工厂模式，则将 intent的 Category 设置为 Intent.CATEGORY HOME 后返回 Intent 。

       再回到 AMS的startHomeActivityLocked 方法，在注释3处通过resolveActivityInfo方法解析处ActivityInfo信息，代码如下：

**frameworks/base/services/core/java/com/android/server/am/ActivityManageService.jav**

private ActivityInfo resolveActivityInfo(Intent intent, int flags, int userId) {

ActivityInfo ai = null;

ComponentName comp = intent.getComponent();

try {

if (comp != null) {

// Factory test.

ai = AppGlobals.getPackageManager().getActivityInfo(comp, flags, userId);

} else {

ResolveInfo info = AppGlobals.getPackageManager().resolveIntent(

intent,

intent.resolveTypeIfNeeded(mContext.getContentResolver()),

flags, userId);

if (info != null) {

ai = info.activityInfo;

}

}

} catch (RemoteException e) {

// ignore

}

return ai;

}

        再回到 AMS的startHomeActivityLocked 方法，假设系统的运行模式不是低级工厂模式，在注释4 处判断符合 Action Intent.ACTION MAIN 、 Category 为 Intent.CATEGORY\_ HOME 的应用程序是否已经启动，如 果没 启动则调 用注释4 处的方法启 动该应用程序。  这个被启 动的应用程 序就 Launcher ，因为 Launcher的 AndroidManifest 文 件中的 intent- filter 表情 匹配了 Action 为Intent.ACTION\_MAIN, Category为 Intent.CATEGORY \_HOME。

Launcher AndroidManifest 文件如下所示：

**packages/apps/Launche3/AndroidManifest.xml**

<manifest

xmlns:android="http://schemas.android.com/apk/res/android"

package="com.android.launcher3">

<uses-sdk android:targetSdkVersion="23" android:minSdkVersion="21"/>

...

<application

android:backupAgent="com.android.launcher3.LauncherBackupAgent"

android:fullBackupOnly="true"

android:fullBackupContent="@xml/backupscheme"

android:hardwareAccelerated="true"

android:icon="@drawable/ic\_launcher\_home"

android:label="@string/derived\_app\_name"

android:theme="@style/LauncherTheme"

android:largeHeap="@bool/config\_largeHeap"

android:restoreAnyVersion="true"

android:supportsRtl="true" >

<!--

Main launcher activity. When extending only change the name, and keep all the

attributes and intent filters the same

-->

<activity

android:name="com.android.launcher3.Launcher"

android:launchMode="singleTask"

android:clearTaskOnLaunch="true"

android:stateNotNeeded="true"

android:windowSoftInputMode="adjustPan"

android:screenOrientation="nosensor"

android:configChanges="keyboard|keyboardHidden|navigation"

android:resizeableActivity="true"

android:resumeWhilePausing="true"

android:taskAffinity=""

android:enabled="true">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.HOME" />

<category android:name="android.intent.category.DEFAULT" />

<category android:name="android.intent.category.MONKEY"/>

<category android:name="android.intent.category.LAUNCHER\_APP" />

</intent-filter>

</activity>

...

</application>

</manifest>

       可以看到intent-filter设置了<category android:name="android.intent.category.HOME" /> 属性，这样名称为com.android.launcher3.Launcher的Activity 就成为了主 Activity。 从前面 AMS的startHomeActivityLocked方法的注释5处，我们知如 Launcher 没有启动就会调用ActivityStarter的startHomeActivityLocked 方法来启动Launcher ，如下所示：

**frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java**

void startHomeActivityLocked(Intent intent, ActivityInfo aInfo, String reason) {

// 将Launcher放入HomeStack中

mSupervisor.moveHomeStackTaskToTop(reason); // ... 1

mLastHomeActivityStartResult = startActivityLocked(null /\*caller\*/, intent,

null /\*ephemeralIntent\*/, null /\*resolvedType\*/, aInfo, null /\*rInfo\*/,

null /\*voiceSession\*/, null /\*voiceInteractor\*/, null /\*resultTo\*/,

null /\*resultWho\*/, 0 /\*requestCode\*/, 0 /\*callingPid\*/, 0 /\*callingUid\*/,

null /\*callingPackage\*/, 0 /\*realCallingPid\*/, 0 /\*realCallingUid\*/,

0 /\*startFlags\*/, null /\*options\*/, false /\*ignoreTargetSecurity\*/,

false /\*componentSpecified\*/, mLastHomeActivityStartRecord /\*outActivity\*/,

null /\*inTask\*/, "startHomeActivity: " + reason); // ... 2

if (mSupervisor.inResumeTopActivity) {

// If we are in resume section already, home activity will be initialized, but not

// resumed (to avoid recursive resume) and will stay that way until something pokes it

// again. We need to schedule another resume.

mSupervisor.scheduleResumeTopActivities();

}

}

       在注释1处将 Launcher放入HomeStack 中，HomeStack 是在 ActivityStackSupervisor 中定义的用于存储Launcher的变量 。接着调用注释2处的startActivityLocked方法来启动Launcher, 剩余的过程会和普通Activity 启动过程类似，后面会出Activity启动过程相关文章，这里就先不做讲解。最终进Launcher的onCreate 方注中，到这里 Launcher就启动完成了。

Advertisement

    3 Launcher 中应用图标显示过程

       Launcher 完成启动后会做很多的工作，作为桌面它会显示应用程序图标， 这与应用程序开发有所关联，应用程序图标是用户进入应用程的入口，因此我们有必要了解 Launcher 是如何显示应用程序图标的。

       我们先从 Launcher的 on Create 方法入手，代码如下所：

**packages/apps/Launcher3/src/com/android/launcher3/Launcher.java**

@Override

protected void onCreate(Bundle savedInstanceState) {

...

LauncherAppState app = LauncherAppState.getInstance(this); // ... 1

// Load configuration-specific DeviceProfile

mDeviceProfile = app.getInvariantDeviceProfile().getDeviceProfile(this);

if (isInMultiWindowModeCompat()) {

Display display = getWindowManager().getDefaultDisplay();

Point mwSize = new Point();

display.getSize(mwSize);

mDeviceProfile = mDeviceProfile.getMultiWindowProfile(this, mwSize);

}

mOrientation = getResources().getConfiguration().orientation;

mSharedPrefs = Utilities.getPrefs(this);

mIsSafeModeEnabled = getPackageManager().isSafeMode();

mModel = app.setLauncher(this); // ...2

...

if (!mModel.startLoader(currentScreen)) { // ... 3

// If we are not binding synchronously, show a fade in animation when

// the first page bind completes.

mDragLayer.setAlpha(0);

} else {

// Pages bound synchronously.

mWorkspace.setCurrentPage(currentScreen);

setWorkspaceLoading(true);

}

...

}

      在注释1处获取 La uncherAppState 的实 例， 在注释2 处调用它 setLauncher方法并将 Launcher对象传入 ， LauncherAppState 的 setLauncher 方法如下所示：

**packages/apps/Launcher3/src/com/android/launcher3/LauncherAppState.java**

LauncherModel setLauncher(Launcher launcher) {

getLocalProvider(mContext).setLauncherProviderChangeListener(launcher);

mModel.initialize(launcher); // ... 1

return mModel;

}

       在注释1 处会调用 Launch erModel的 initialize 方法，代码如下 ：

**packages/apps/Launcher3/src/com/android/launche3/LauncherModel.java**

public void initialize(Callbacks callbacks) {

synchronized (mLock) {

Preconditions.assertUIThread();

mCallbacks = new WeakReference<>(callbacks);

}

}

       在initialize方 法中会将 Ca llbacks 也就是传人的 Launch er ，封装成一个弱 引用对 象。 因此  mCallbacks 变量指的就是封装成弱引用对象的 Launc her ，这个 mCallbacks 后面会用到它。

       再回到 Launcher的 on Create 方法 ，在注释3处调 用了 Launcher Model的 startLoader方 法，代码如下所示：

**packages/apps/Launcher3/src/com/android/launche3/LauncherModel.java**

...

// 创建了具有消息循环的线程HandlerThread对象

@Thunk static final HandlerThread sWorkerThread = new HandlerThread("launcher-loader"); // ... 1

static {

sWorkerThread.start();

}

@Thunk static final Handler sWorker = new Handler(sWorkerThread.getLooper()); // ... 2

...

public boolean startLoader(int synchronousBindPage) {

// Enable queue before starting loader. It will get disabled in Launcher#finishBindingItems

InstallShortcutReceiver.enableInstallQueue(InstallShortcutReceiver.FLAG\_LOADER\_RUNNING);

synchronized (mLock) {

// Don't bother to start the thread if we know it's not going to do anything

if (mCallbacks != null && mCallbacks.get() != null) {

final Callbacks oldCallbacks = mCallbacks.get();

// Clear any pending bind-runnables from the synchronized load process.

mUiExecutor.execute(new Runnable() {

public void run() {

oldCallbacks.clearPendingBinds();

}

});

// If there is already one running, tell it to stop.

stopLoader();

LoaderResults loaderResults = new LoaderResults(mApp, sBgDataModel,

mBgAllAppsList, synchronousBindPage, mCallbacks); // ... 3

if (mModelLoaded && !mIsLoaderTaskRunning) {

// Divide the set of loaded items into those that we are binding synchronously,

// and everything else that is to be bound normally (asynchronously).

loaderResults.bindWorkspace();

// For now, continue posting the binding of AllApps as there are other

// issues that arise from that.

loaderResults.bindAllApps();

loaderResults.bindDeepShortcuts();

loaderResults.bindWidgets();

return true;

} else {

startLoaderForResults(loaderResults); // ... 4

}

}

}

return false;

}

       在注释1 处创建了具有消息循环的线程 HandlerThread 象。在注释2 处创建了 Handler 并且传入 Hand lerThr ead Looper ，这里 Hander 的作用就是向 HandlerThread 发送消息。 在注释3处创建LoaderResults类的对象loaderResults，并传入mCallbacks，

在注释4处调用startLoaderForResults方法并传入刚才创建的loaderResults的对象，代码如下所示：

**packages/apps/Launcher3/src/com/android/launche3/LauncherModel.java**

public void startLoaderForResults(LoaderResults results) {

synchronized (mLock) {

stopLoader();

mLoaderTask = new LoaderTask(mApp, mBgAllAppsList, sBgDataModel, results); // ... 1

runOnWorkerThread(mLoaderTask); // ... 2

}

}

       在注释1处创建LoaderTask的对象mLoaderTask并将LoaderResult对象传入构造器，后面会用到这个LoaderResult的对象，在注释2处调用runOnWorkerThread方法，代码如下：

**packages/apps/Launcher3/src/com/android/launche3/LauncherModel.java**

private static void runOnWorkerThread(Runnable r) {

if (sWorkerThread.getThreadId() == Process.myTid()) {

r.run(); // 如果当前线程是工作线程，直接执行run方法

} else {

// If we are not on the worker thread, then post to the worker handler

// 如果当前线程不是工作现在，那么post到工作线程处理。

sWorker.post(r);

}

}

       LoaderTask类实现了Runnable接口，当LoaderTask所描述的消息被处理时，则会调用它的run方法，LoaderTask的run方法代码如下所示：

**packages/apps/Launcher3/src/com/android/launche3/model/LoaderTask.java**

public void run() {

synchronized (this) {

// Skip fast if we are already stopped.

if (mStopped) {

return;

}

}

try (LauncherModel.LoaderTransaction transaction = mApp.getModel().beginLoader(this)) {

long now = 0;

if (DEBUG\_LOADERS) Log.d(TAG, "step 1.1: loading workspace");

// 加载工作区信息

loadWorkspace(); // ... 1

verifyNotStopped();

if (DEBUG\_LOADERS) Log.d(TAG, "step 1.2: bind workspace workspace");

// 绑定工作区信息

mResults.bindWorkspace(); // ... 2

// Take a break

if (DEBUG\_LOADERS) {

Log.d(TAG, "step 1 completed, wait for idle");

now = SystemClock.uptimeMillis();

}

waitForIdle();

if (DEBUG\_LOADERS) Log.d(TAG, "Waited " + (SystemClock.uptimeMillis() - now) + "ms");

verifyNotStopped();

// second step

if (DEBUG\_LOADERS) Log.d(TAG, "step 2.1: loading all apps");

// 加载系统已经安装的应用程序信息

loadAllApps(); // ... 3

if (DEBUG\_LOADERS) Log.d(TAG, "step 2.2: Binding all apps");

verifyNotStopped();

// 绑定Appinfo

mResults.bindAllApps(); // ... 4

...

} catch (CancellationException e) {

// Loader stopped, ignore

if (DEBUG\_LOADERS) {

Log.d(TAG, "Loader cancelled", e);

}

}

}

       Launcher是用 工作区的形式来显示系统安装的应用程序的快捷图标的，每一 个工作区都是用来描述一个抽 象桌 面的，它由n个 屏幕 组成， 每个 屏幕又分n 个单元格，每个单元格用来显示一 个应用程序 的快捷图 标。 在注释1处和注释2 处分别调用 load Workspa ce

方法和 bindWorksp ace方法 来加 载和绑定工作区信息 。注释3 处的 l oadA llApp方法 用来加载系统已经安装的应用程序信息，代码如下：

**packages/apps/Launcher3/src/com/android/launche3/model/LoaderTask.java**

private void loadAllApps() {

...

// Clear the list of apps

mBgAllAppsList.clear();

for (UserHandle user : profiles) {

...

final List<LauncherActivityInfo> apps = mLauncherApps.getActivityList(null, user);

...

for (int i = 0; i < apps.size(); i++) {

LauncherActivityInfo app = apps.get(i);

// This builds the icon bitmaps.

mBgAllAppsList.add(new AppInfo(app, user, quietMode), app); // ... 1

}

ManagedProfileHeuristic.onAllAppsLoaded(mApp.getContext(), apps, user);

}

...

}

       在注释1处创建AppInfo信息并添加到mBgAllAppsList列表中。

       在回到LoaderTask的run方法，在注释4处的mResult就是构造LoaderTask时传入的LoaderResult对象，调用LoaderResult的bindAllApps方法绑定App信息，代码如下：

**packages/apps/Launcher3/src/com/android/launche3/model/LoaderResult.java**

public void bindAllApps() {

// shallow copy

// 将mBgAllAppList中的data属性克隆一份

@SuppressWarnings("unchecked")

final ArrayList<AppInfo> list = (ArrayList<AppInfo>) mBgAllAppsList.data.clone();

Runnable r = new Runnable() {

public void run() {

Callbacks callbacks = mCallbacks.get();

if (callbacks != null) {

// 这里的callbacks就是Launcher的对象

callbacks.bindAllApplications(list); // ... 1

}

}

};

mUiExecutor.execute(r);

}

       在注释1处调用callbacks的bindAllApplications方法实质上就是调用Launcher的bindAllApplications方法，接下里分析Launcher中的bindAllApplications方法，代码如下：

**packages/apps/Launcher3/src/com/android/launcher3/Launcher.java**

public void bindAllApplications(final ArrayList<AppInfo> apps) {

...

if (mAppsView != null) {

Executor pendingExecutor = getPendingExecutor();

if (pendingExecutor != null && mState != State.APPS) {

// Wait until the fade in animation has finished before setting all apps list.

pendingExecutor.execute(r);

return;

}

mAppsView.setApps(apps); // ... 1

}

...

}

      在注释1处调 用 AllA pps Conta ine rView 类型的 mApps View的 setApp方法 ，并将包含应用信息的 列表 a pps传 进去， AllA pps Conta ine rView 的setApps方法如 下所示：

**packages/apps/Launcher3/src/com/android/launcher3/allapps/AllAppsContainerView.java**

public void setApps(List<AppInfo> apps) {

mApps.setApps(apps);

}

       se tApps方法会将包 含应用 信息 的列表a pps设置给 mApps ，这个 mApps是 AlphabeticalAppsList  类型 。接 着查 AllAppsContainerView的onFinishInflate 方法，代码如下 所示：

@Override

protected void onFinishInflate() {

super.onFinishInflate();

...

mAppsRecyclerView = findViewById(R.id.apps\_list\_view); // ... 1

mAppsRecyclerView.setApps(mApps); // ... 2

mAppsRecyclerView.setLayoutManager(mLayoutManager);

mAppsRecyclerView.setAdapter(mAdapter); // ... 3

...

}

       onFinishInflate方法会在AllAppsContainerView加 载完 X ML 时调用， 在 注释1处得到AllApps RecyclerV iew （RecycleView的子类）用 来显示 a pp 列 表，并在注释2 处将此前的 mApps设置进去， 在注释3处为 All AppsRecycler View 设置 Adapter 。这样应用程序快捷图标就会显示在屏幕上。

       到这里 Launcher 中应用图标显示过程以及 Launcher 启动流程就讲解完了，接下来介绍Android 系统启动流程。

 Android 系统启动流程

     1．启动电源以及系统启动

       当电源按下时引导芯片代码从预定义的地方（固化在 ROM ）开始执行。加载引导程序 BootLoader到RAM ，然后执行。

     2. 引导程序 Bootloader

       引导程序 BootLoader 是在 Android 操作系统开始运行前的一 个小程序，它的主要作用是把操作系统 拉起来并运行。

     3. Linux 内核启动

       当内核启动时  ，设置缓存、被保护存储器、计划列表、加载驱动。当内核完成系统设置时，它首先在系统文件中寻找 init.rc 文件，并启动 init 进程。

     4. init 进程启动

       初始化和启动属性服务，并且启动 Zygote 进程。

     5. Zygote 进程启动

       创建Java 虚拟机并为 Java 虚拟机注册 JNI 方法 ，创建服务器端 Socket ，启动SystemServer 进程。

     6. System Server 进程启动

       启动 Binder 线 程池和 SystemServiceManager ，并且启动各种系统服务。

     7. Launcher 启动

       SystemServer 进程启动的 AMS 会启动 Launcher, Launcher 启动 后会将己安装应用的快捷图标显示到界面上。