# 深入理解Android

## 卷2

### SystemServer

可以称之为系统进程，运行了系统的主要服务。

进程名：system\_server

线程：ServerThread

## 4 PKMS

创建

systemThread.run

PackageManagerService.main

new PackageManagerService

settings.addSharedUserLPw

new SharedUserSetting

sharedUserSetting.addUserIdLPw

pkms.readPermission

~.readPermissionsFromXml

settings.readLPw

installer.dexopt：dex化核心库

file.delete: 删除缓存文件

pkms.scanDirLI

~.scanPackageLI

new PackageParser

packageparser.parsePackage

~.parsePackage

~.parsePackageName

new Package

~.paserApplication

pkms.scanPacakgeLi

{单独处理系统Activity}

~.fixProcessName

~.getDataPathForPackage

userManager.installPacakgeForAllUsers

{处理so}

~.performDexOptLI

ServiceManager.addService

packageManagerService.isFirstBoot

~.performBootDexOpt

~.systemReady

## 6 AMS

### 6.2 创建AMS

系统进程：

ServerThread.run

AMS.main

AThread: 创建消息循环、new AMS

ActivityThread.systemMain

activityThread.attach

~.getSystemContext

activityThread.getSystemContext

ams.startRunning

AMS.setSystemProcess

activityThread.installSystemApplicationInfo

context.init

ams.newProcessRecordLocked

~.newProcessRecord

AMS.installSystemProvider

ams.generateApplicationProviderLocked

pms.queryContentProviders

activityThread.installSystemProviders

~.installContentProviders

~.installProvider

ams.publishContentProviders

AMS.systemReady

ServerThread.main

systemServer.startSystemUi

activityStack.resumeTopActivityLocked

ams.startHomeActivityLocked

### 6.3 Activity生命周期

系统进程：

ams.startActivityAndWait

activityStatck.startActivityMayWait

~.startActivityLocked

~.startActivityUncheckedLocked

~.startActivtyLocked

~.resumeTopActivityLocked

~.startPausingLocked

~.startSpecificActivityLocked

ams.startProcessLocked

~.startProcessLocked

Process.start

应用进程

ActivityThread.main

activityThread.attach

activityManager.attachApplication

系统进程

ams.attachApplictionLocked

applicationThread.bindApplication

~.handleBindApplication

activityStack.realStartActivtyLocked ->

activtyStack.completeResumeLocked

ams.realStartServiceLocked

应用进程

activityThread.scheduleLaunchActivity

~.handleLaunchActivity

~.handleResumeActivty

~.performResumeActivity

activityStack.activityIdleInternal

activityStack.startPausingLocked

handlePausingActivity

performPauseActivity

### 6.4 Broadcast

contextImpl.registerReceiver

contextImpl.registerReceiverInternal

amn.registerReceiver

ams.registerReceiver

contextImpl.sendBroadcast

amn.broadcastIntent

ams.broadcastIntentInternal: 内部处理广播；匹配；发送；

scheduleBroadcastsLocked

BROADCAST\_INTENT\_MSG

BROADCAST\_INTENT\_MSG

ams.processNextBroadcast

ams.deliverToRegisteredReceiverLocked

ams.performReceiveLocked

applicationThread.scheduleRegisteredReceiver

/iintentReceiver.performReceive

activityThread. scheduleRegisteredReceiver

LoadedApk.receiverDispatcher.performReceive

handler.post

args.sendFinished

ams.finshReceiver

args.run

broadcastReceiver.onReceive

ams.finshReceiver: 调度下一轮广播

### 6.5 startService

### 6.6 进程

ams.attachApplictionLocked

~.updateLrcProcessLocked

~. updateLrcProcessInternalLocked: 调整进程列表

~.updateOomAdjLocked

~.updateOomAdjLocked

~.computeOomAdjLocked

### 6.7 crash

RuntimeInit.commonInit

Thread.setDeafultUncaughtExceptionHandler

uncaughtHandler.uncaughtException

ams.handleApplicationCrash

~.crashApplication

~.makeAppCrashingLocked

~.startAppProblemLocked

~.handleAppCrashLocked

appDeathRecipient.binderDied

ams.appDiedLocked

~.handleAppDiedLocked

~.cleanUpApplicationRecordLocked

~.removeDyingProviderLocked

## 7 ContentProvider

context.getContentResolver

contextImpl.getContentResolver

contextImpl.init

new ApplicationContentResolver

MediaStore.Image.Media.query

contentResolver.query

~.acquireProvider

applciationContentResolver.acquireProvider

activityThread.acquireProvider

~.getProvider

ams.ContentProvider

ams.ContentProvider

~.getContentProviderImpl

~.startProcessLocked

~.incProviderCount

contentProviderRecord.wait

系统进程

ams.attachApplicationLocked

applicationThread.bindApplication

服务端进程

activityThread.handleBindApplication

~.installContentProviders

~.installProvider

new ContentProvider

contentProvider.getIContentProvider

ams.publishContentProviders

ContentProvider.transport.query

contentProvider.query

### 7.3 SQLite

mediaProvider.attach

new DatabaseHelper

databaseHelper.getWritableDatabase

contentImple.openOrCreateDatabase

SQLiteDatabase.openOrCreateDatabase

~.openDatabase

New SQLiteDatabase

sqliteDatabase.open

~.onCreate

/~.onUpgrade

7.4 Cursor

客户端进程

contentResolver.query

~.acquireProvider

contentProviderProxy.query

new BulkCursorToCursorAdapter

adaptor.getObserver

BulkCursorNative.asInterface

adaptor.initailize

contentResolver.releaseProvider

return new CursorWrapperInner

服务端进程

contentProviderNative.onTransact

IContentObserver.Stub.asInterface

mediaProvider.query

sqliteQueryBuilder.query

~.buildQuery

sqliteDatabase.rawQueryWithFactory

sqliteCursorDriver.query

new SQLiteQuery

new SQLiteCursor

cusor.setNotificationUri

new CursorToBulkCursorAdapter

adaptor.count

sqliteCursor.count

~.fillWindow

~.clearOrCreateLocalWindow: 创建共享内存

sqliteQuery.fillWindow

客户端进程

cursorWrapper.moveToFirst

abstractCursor.moveToFirst

~.moveToPosition

bulkCursorToCursorAdaptor.onMove

bulkCursorProxy.getWindow

服务端

cursorToBulkCursorAdaptor.getWindow

sqliteCursor.getWindow

~.moveToPosition

~.onMove

~.fillWindow

### 7.5 cusor close

cursorWrapperInner.close

cursorWrapper.close

bulkCursorToCursorAdapter.close

abstarctCursor.close

abstarctWindowedCursor.onDeactivateOrClose

~.closeWindow

cursorWindow.close

sqliteClosable.releaseReference

cursorWindow.onAllReferenceReleased

~.navtiveDispose

### 7.6 openAssetFileDescriptor

客户端

contentResolver.openAssetFileDescriptor

~.openTypedAssetFileDescriptor

contentProviderProxy.openTypedAssetFile

服务端

contentProvider.openTypedAssetFile

~.openAssetFile

mediaProvider.openFile

contentProvider.openFileHelper

ParcelFileDescriptor.open

## ContentService

创建

ServerThread.run

ContentService.main

new ContentService

getSyncManager

new SyncManager

注册

contentResovler.registerContentObserver

contentObserver.getContentObserver

contentService.registerContentObserver

observerNode.addObserverLocked

通知

mediaProvider.update

contentResolver.notifyChange

~.~

contentService.~

contentObserver.onChange

## AccountManagerService

serverThread.run

new AccountManagerService

new AccountAuthenticatorCache

new RegisteredServicesCache

generateServicesMap

parseServiceInfo

new AccountManagerService

## SyncManager

new SyncManager

SyncStorageEngine.init

new SyncStorageEngine

~.getSingleton

new SyncAdaptersCache

registeredServicesCache

new SyncQueue

syncStorageEngine.addStatusChangeListener

accountManager.addOnAccountsUpatedListener

contentService.setSyncAutomatically

syncStorageEngine.~

发起同步请求：

contentResolver.requestSync

contentService.~

syncManager.scheduleSync

new SyncOperation

scheduleSyncOperation

syncHandler.handleMessage

syncManager.maybeStartNextSyncLocked

syncManager.dispatchSyncOperation

activeSyncContext.bindToSyncAdapter

context.bindService

activeSyncContext.onServiceConnected

MESSAGE\_SERVICE\_CONNECTED

syncHandler.handleMessage

syncManager.runBoundToSyncAdapter

syncAdapter.startSync

同步服务进程？

emailSyncAdapterService.onBind

sSyncAdapter.getSyncAdapterBinder

iSyncAdapter.startSync

new SyncThread

iSyncAdapterImpl.run

abstractThreadedSyncAdapter.onPerformdSync

emailSyncAdapterService.~

syncContext.onFinished

# 动画常用类

# 事件

# Webview

# 进程和线程

当某个应用组件启动且该应用没有运行其他任何组件时，Android 系统会使用单个执行线程为应用启动新的 Linux 进程。默认情况下，同一应用的所有组件在相同的进程和线程（称为“主”线程）中运行。 如果某个应用组件启动且该应用已存在进程（因为存在该应用的其他组件），则该组件会在此进程内启动并使用相同的执行线程。 但是，您可以安排应用中的其他组件在单独的进程中运行，并为任何进程创建额外的线程。

本文档介绍进程和线程在 Android 应用中的工作方式。

进程

默认情况下，同一应用的所有组件均在相同的进程中运行。如果您发现需要控制某个组件所属的进程，则可在清单文件中执行此操作。

各类组件元素的清单文件条目—[<activity>](https://developer.android.google.cn/guide/topics/manifest/activity-element.html)、[<service>](https://developer.android.google.cn/guide/topics/manifest/service-element.html)、[<receiver>](https://developer.android.google.cn/guide/topics/manifest/receiver-element.html) 和 [<provider>](https://developer.android.google.cn/guide/topics/manifest/provider-element.html)—均支持 android:process 属性，此属性可以指定该组件应在哪个进程运行。您可以设置此属性，使每个组件均在各自的进程中运行，或者使一些组件共享一个进程，而其他组件则不共享。 此外，您还可以设置 android:process，使不同应用的组件在相同的进程中运行，但前提是这些应用共享相同的 Linux 用户 ID 并使用相同的证书进行签署。

此外，[<application>](https://developer.android.google.cn/guide/topics/manifest/application-element.html) 元素还支持 android:process 属性，以设置适用于所有组件的默认值。

进程生命周期

如果内存不足，而其他为用户提供更紧急服务的进程又需要内存时，Android 可能会决定在某一时刻关闭某一进程。在被终止进程中运行的应用组件也会随之销毁。 当这些组件需要再次运行时，系统将为它们重启进程。

决定终止哪个进程时，Android 系统将权衡它们对用户的相对重要程度。例如，相对于托管可见 Activity 的进程而言，它更有可能关闭托管屏幕上不再可见的 Activity 的进程。 因此，是否终止某个进程的决定取决于该进程中所运行组件的状态。 下面，我们介绍决定终止进程所用的规则。

Android 系统将尽量长时间地保持应用进程，但为了新建进程或运行更重要的进程，最终需要移除旧进程来回收内存。 为了确定保留或终止哪些进程，系统会根据进程中正在运行的组件以及这些组件的状态，将每个进程放入“重要性层次结构”中。 必要时，系统会首先消除重要性最低的进程，然后是重要性略逊的进程，依此类推，以回收系统资源。

重要性层次结构一共有 5 级：

1. **前台进程**

用户当前操作所必需的进程。如果一个进程满足以下任一条件，即视为前台进程：

* + 托管用户正在交互的 [Activity](https://developer.android.google.cn/reference/android/app/Activity.html)（已调用 [Activity](https://developer.android.google.cn/reference/android/app/Activity.html) 的 [onResume()](https://developer.android.google.cn/reference/android/app/Activity.html#onResume()) 方法）
  + 托管某个 [Service](https://developer.android.google.cn/reference/android/app/Service.html)，后者绑定到用户正在交互的 Activity
  + **托管正在“前台”运行的**[**Service**](https://developer.android.google.cn/reference/android/app/Service.html)**（服务已调用**[**startForeground()**](https://developer.android.google.cn/reference/android/app/Service.html#startForeground(int,%20android.app.Notification))**）**
  + 托管正执行一个生命周期回调的 [Service](https://developer.android.google.cn/reference/android/app/Service.html)（[onCreate()](https://developer.android.google.cn/reference/android/app/Service.html#onCreate())、[onStart()](https://developer.android.google.cn/reference/android/app/Service.html#onStart(android.content.Intent,%20int)) 或 [onDestroy()](https://developer.android.google.cn/reference/android/app/Service.html#onDestroy())）
  + 托管正执行其 [onReceive()](https://developer.android.google.cn/reference/android/content/BroadcastReceiver.html#onReceive(android.content.Context,%20android.content.Intent)) 方法的 [BroadcastReceiver](https://developer.android.google.cn/reference/android/content/BroadcastReceiver.html)

通常，在任意给定时间前台进程都为数不多。只有在内存不足以支持它们同时继续运行这一万不得已的情况下，系统才会终止它们。 此时，设备往往已达到内存分页状态，因此需要终止一些前台进程来确保用户界面正常响应。

1. **可见进程**

没有任何前台组件、但仍会影响用户在屏幕上所见内容的进程。 如果一个进程满足以下任一条件，即视为可见进程：

* + 托管不在前台、但仍对用户可见的 [Activity](https://developer.android.google.cn/reference/android/app/Activity.html)（已调用其 [onPause()](https://developer.android.google.cn/reference/android/app/Activity.html#onPause()) 方法）。
  + 托管绑定到可见（或前台）Activity 的 [Service](https://developer.android.google.cn/reference/android/app/Service.html)。

除非为了维持所有前台进程同时运行而必须终止，否则系统不会终止这些进程。

1. **服务进程**

正在运行已使用 [startService()](https://developer.android.google.cn/reference/android/content/Context.html#startService(android.content.Intent)) 方法启动的服务且不属于上述两个更高类别进程的进程。尽管服务进程与用户所见内容没有直接关联，但是它们通常在执行一些用户关心的操作（例如，在后台播放音乐或从网络下载数据）。因此，除非内存不足以维持所有前台进程和可见进程同时运行，否则系统会让服务进程保持运行状态。

1. **后台进程**

包含目前对用户不可见的 Activity 的进程（已调用 Activity 的 [onStop()](https://developer.android.google.cn/reference/android/app/Activity.html#onStop()) 方法）。这些进程对用户体验没有直接影响，系统可能随时终止它们，以回收内存供前台进程、可见进程或服务进程使用。 通常会有很多后台进程在运行，因此它们会保存在 LRU （最近最少使用）列表中，以确保包含用户最近查看的 Activity 的进程最后一个被终止。如果某个 Activity 正确实现了生命周期方法，并保存了其当前状态，则终止其进程不会对用户体验产生明显影响，因为当用户导航回该 Activity 时，Activity 会恢复其所有可见状态。

1. **空进程**

不含任何活动应用组件的进程。保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。 为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。

根据进程中当前活动组件的重要程度，Android 会将进程评定为它可能达到的最高级别。

此外，一个进程的级别可能会因其他进程对它的依赖而有所提高，即服务于另一进程的进程其级别永远不会低于其所服务的进程。 例如，如果进程 A 中的内容提供程序为进程 B 中的客户端提供服务，或者如果进程 A 中的服务绑定到进程 B 中的组件，则进程 A 始终被视为至少与进程 B 同样重要。

由于运行服务的进程其级别高于托管后台 Activity 的进程，因此启动长时间运行操作的 Activity 最好为该操作启动[服务](https://developer.android.google.cn/guide/components/services.html)，而不是简单地创建工作线程，当操作有可能比 Activity 更加持久时尤要如此。例如，正在将图片上传到网站的 Activity 应该启动服务来执行上传，这样一来，即使用户退出 Activity，仍可在后台继续执行上传操作。使用服务可以保证，无论 Activity 发生什么情况，该操作至少具备“服务进程”优先级。广播接收器也应使用服务，而不是简单地将耗时冗长的操作放入线程中。

线程

应用启动时，系统会为应用创建一个名为“主线程”的执行线程。 此线程非常重要，因为它负责将事件分派给相应的用户界面小部件，其中包括绘图事件。 此外，它也是应用与 Android UI 工具包组件（来自 [android.widget](https://developer.android.google.cn/reference/android/widget/package-summary.html) 和 [android.view](https://developer.android.google.cn/reference/android/view/package-summary.html)软件包的组件）进行交互的线程。因此，主线程有时也称为 UI 线程。

系统*不会*为每个组件实例创建单独的线程。运行于同一进程的所有组件均在 UI 线程中实例化，并且对每个组件的系统调用均由该线程进行分派。 因此，响应系统回调的方法（例如，报告用户操作的 [onKeyDown()](https://developer.android.google.cn/reference/android/view/View.html#onKeyDown(int,%20android.view.KeyEvent)) 或生命周期回调方法）始终在进程的 UI 线程中运行。

例如，当用户触摸屏幕上的按钮时，应用的 UI 线程会将触摸事件分派给小部件，而小部件反过来又设置其按下状态，并将失效请求发布到事件队列中。 UI 线程从队列中取消该请求并通知小部件应该重绘自身。

在应用执行繁重的任务以响应用户交互时，除非正确实现应用，否则这种单线程模式可能会导致性能低下。 具体地讲，如果 UI 线程需要处理所有任务，则执行耗时很长的操作（例如，网络访问或数据库查询）将会阻塞整个 UI。 一旦线程被阻塞，将无法分派任何事件，包括绘图事件。 从用户的角度来看，应用显示为挂起。 更糟糕的是，如果 UI 线程被阻塞超过几秒钟时间（目前大约是 5 秒钟），用户就会看到一个让人厌烦的“[应用无响应](http://developer.android.google.cn/guide/practices/responsiveness.html)”(ANR) 对话框。如果引起用户不满，他们可能就会决定退出并卸载此应用。

此外，Android UI 工具包*并非*线程安全工具包。因此，您不得通过工作线程操纵 UI，而只能通过 UI 线程操纵用户界面。 因此，Android 的单线程模式必须遵守两条规则：

1. 不要阻塞 UI 线程
2. 不要在 UI 线程之外访问 Android UI 工具包

工作线程

根据上述单线程模式，要保证应用 UI 的响应能力，关键是不能阻塞 UI 线程。 如果执行的操作不能很快完成，则应确保它们在单独的线程（“后台”或“工作”线程）中运行。

例如，以下代码演示了一个点击侦听器从单独的线程下载图像并将其显示在 [ImageView](https://developer.android.google.cn/reference/android/widget/ImageView.html) 中：

public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");  
            mImageView.setImageBitmap(b);  
        }  
    }).start();  
}

乍看起来，这段代码似乎运行良好，因为它创建了一个新线程来处理网络操作。 但是，它违反了单线程模式的第二条规则：*不要在 UI 线程之外访问 Android UI 工具包* — 此示例从工作线程（而不是 UI 线程）修改了 [ImageView](https://developer.android.google.cn/reference/android/widget/ImageView.html)。 这可能导致出现不明确、不可预见的行为，但要跟踪此行为困难而又费时。

为解决此问题，Android 提供了几种途径来从其他线程访问 UI 线程。 以下列出了几种有用的方法：

* [Activity.runOnUiThread(Runnable)](https://developer.android.google.cn/reference/android/app/Activity.html#runOnUiThread(java.lang.Runnable))
* [View.post(Runnable)](https://developer.android.google.cn/reference/android/view/View.html#post(java.lang.Runnable))
* [View.postDelayed(Runnable, long)](https://developer.android.google.cn/reference/android/view/View.html#postDelayed(java.lang.Runnable,%20long))

例如，您可以通过使用 [View.post(Runnable)](https://developer.android.google.cn/reference/android/view/View.html#post(java.lang.Runnable)) 方法修复上述代码：

public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            final Bitmap bitmap =  
                    loadImageFromNetwork("http://example.com/image.png");  
            mImageView.post(new Runnable() {  
                public void run() {  
                    mImageView.setImageBitmap(bitmap);  
                }  
            });  
        }  
    }).start();  
}

现在，上述实现属于线程安全型：在单独的线程中完成网络操作，而在 UI 线程中操纵 [ImageView](https://developer.android.google.cn/reference/android/widget/ImageView.html)。

但是，随着操作日趋复杂，这类代码也会变得复杂且难以维护。 要通过工作线程处理更复杂的交互，可以考虑在工作线程中使用 [Handler](https://developer.android.google.cn/reference/android/os/Handler.html) 处理来自 UI 线程的消息。当然，最好的解决方案或许是扩展 [AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 类，此类简化了与 UI 进行交互所需执行的工作线程任务。

使用 AsyncTask

[AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 允许对用户界面执行异步操作。 它会先阻塞工作线程中的操作，然后在 UI 线程中发布结果，而无需您亲自处理线程和/或处理程序。

要使用它，必须创建 [AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 的子类并实现 [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 回调方法，该方法将在后台线程池中运行。 要更新 UI，应该实现 [onPostExecute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onPostExecute(Result)) 以传递 [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 返回的结果并在 UI 线程中运行，以便您安全地更新 UI。 稍后，您可以通过从 UI 线程调用 [execute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#execute(Params...)) 来运行任务。

例如，您可以通过以下方式使用 [AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 来实现上述示例：

public void onClick(View v) {  
    new DownloadImageTask().execute("http://example.com/image.png");  
}  
  
private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {  
    /\*\* The system calls this to perform work in a worker thread and  
      \* delivers it the parameters given to AsyncTask.execute() \*/  
    protected Bitmap doInBackground(String... urls) {  
        return loadImageFromNetwork(urls[0]);  
    }  
  
    /\*\* The system calls this to perform work in the UI thread and delivers  
      \* the result from doInBackground() \*/  
    protected void onPostExecute(Bitmap result) {  
        mImageView.setImageBitmap(result);  
    }  
}

现在 UI 是安全的，代码也得到简化，因为任务分解成了两部分：一部分应在工作线程内完成，另一部分应在 UI 线程内完成。

下面简要概述了 AsyncTask 的工作方法，但要全面了解如何使用此类，您应阅读 [AsyncTask](https://developer.android.google.cn/reference/android/os/AsyncTask.html) 参考文档：

* 可以使用泛型指定参数类型、进度值和任务最终值
* 方法 [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 会在工作线程上自动执行
* [onPreExecute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onPreExecute())、[onPostExecute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onPostExecute(Result)) 和 [onProgressUpdate()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onProgressUpdate(Progress...)) 均在 UI 线程中调用
* [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 返回的值将发送到 [onPostExecute()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onPostExecute(Result))
* 您可以随时在 [doInBackground()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#doInBackground(Params...)) 中调用[publishProgress()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#publishProgress(Progress...))，以在 UI 线程中执行 [onProgressUpdate()](https://developer.android.google.cn/reference/android/os/AsyncTask.html#onProgressUpdate(Progress...))
* 您可以随时取消任何线程中的任务

**注意**：使用工作线程时可能会遇到另一个问题，即：[运行时配置变更](https://developer.android.google.cn/guide/topics/resources/runtime-changes.html)（例如，用户更改了屏幕方向）导致 Activity 意外重启，这可能会销毁工作线程。 要了解如何在这种重启情况下坚持执行任务，以及如何在 Activity 被销毁时正确地取消任务，请参阅[书架](http://code.google.com/p/shelves/)示例应用的源代码。

线程安全方法

在某些情况下，您实现的方法可能会从多个线程调用，因此编写这些方法时必须确保其满足线程安全的要求。

这一点主要适用于可以远程调用的方法，如[绑定服务](https://developer.android.google.cn/guide/components/bound-services.html)中的方法。如果对 [IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 中所实现方法的调用源自运行 [IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 的同一进程，则该方法在调用方的线程中执行。但是，如果调用源自其他进程，则该方法将在从线程池选择的某个线程中执行（而不是在进程的 UI 线程中执行），线程池由系统在与 [IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 相同的进程中维护。 例如，即使服务的 [onBind()](https://developer.android.google.cn/reference/android/app/Service.html#onBind(android.content.Intent)) 方法将从服务进程的 UI 线程调用，在 [onBind()](https://developer.android.google.cn/reference/android/app/Service.html#onBind(android.content.Intent)) 返回的对象中实现的方法（例如，实现 RPC 方法的子类）仍会从线程池中的线程调用。 由于一个服务可以有多个客户端，因此可能会有多个池线程在同一时间使用同一 [IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 方法。因此，[IBinder](https://developer.android.google.cn/reference/android/os/IBinder.html) 方法必须实现为线程安全方法。

同样，内容提供程序也可接收来自其他进程的数据请求。尽管 [ContentResolver](https://developer.android.google.cn/reference/android/content/ContentResolver.html) 和 [ContentProvider](https://developer.android.google.cn/reference/android/content/ContentProvider.html) 类隐藏了如何管理进程间通信的细节，但响应这些请求的 [ContentProvider](https://developer.android.google.cn/reference/android/content/ContentProvider.html) 方法（[query()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#query(android.net.Uri,%20java.lang.String[],%20android.os.Bundle,%20android.os.CancellationSignal))、[insert()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#insert(android.net.Uri,%20android.content.ContentValues))、[delete()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#delete(android.net.Uri,%20java.lang.String,%20java.lang.String[]))、[update()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#update(android.net.Uri,%20android.content.ContentValues,%20java.lang.String,%20java.lang.String[])) 和 [getType()](https://developer.android.google.cn/reference/android/content/ContentProvider.html#getType(android.net.Uri)) 方法）将从内容提供程序所在进程的线程池中调用，而不是从进程的 UI 线程调用。 由于这些方法可能会同时从任意数量的线程调用，因此它们也必须实现为线程安全方法。

进程间通信

Android 利用远程过程调用 (RPC) 提供了一种进程间通信 (IPC) 机制，通过这种机制，由 Activity 或其他应用组件调用的方法将（在其他进程中）远程执行，而所有结果将返回给调用方。 这就要求把方法调用及其数据分解至操作系统可以识别的程度，并将其从本地进程和地址空间传输至远程进程和地址空间，然后在远程进程中重新组装并执行该调用。 然后，返回值将沿相反方向传输回来。 Android 提供了执行这些 IPC 事务所需的全部代码，因此您只需集中精力定义和实现 RPC 编程接口即可。

要执行 IPC，必须使用 [bindService()](https://developer.android.google.cn/reference/android/content/Context.html#bindService(android.content.Intent,%20android.content.ServiceConnection,%20int)) 将应用绑定到服务上。

后台执行限制

每次在后台运行时，应用都会消耗一部分有限的设备资源，例如 RAM。 这可能会影响用户体验，如果用户正在使用占用大量资源的应用（例如玩游戏或观看视频），影响尤为明显。

为了提升用户体验，Android 8.0 对应用在后台运行时可以执行的操作施加了限制。 本文档说明了操作系统的一些变更，以及如何更新应用以便在新限制下正常运行。

概览

多个 Android 应用和服务可以同时运行。 例如，用户可以在一个窗口中玩游戏，同时在另一个窗口中浏览网页，并使用第三个应用播放音乐。

同时运行的应用越多，对系统造成的负担越大。 如果还有应用或服务在后台运行，这会对系统造成更大负担，进而可能导致用户体验下降；例如，音乐应用可能会突然关闭。

为了降低发生这些问题的几率，Android 8.0 对应用在用户不与其直接交互时可以执行的操作施加了限制。

应用在两个方面受到限制：

* [后台服务限制](https://developer.android.google.cn/about/versions/oreo/background#services)：处于空闲状态时，应用可以使用的后台服务存在限制。 这些限制不适用于前台服务，因为前台服务更容易引起用户注意。
* [广播限制](https://developer.android.google.cn/about/versions/oreo/background#broadcasts)：除了有限的例外情况，应用无法使用清单注册隐式广播。 它们仍然可以在运行时注册这些广播，并且可以使用清单注册专门针对它们的显式广播。

注：默认情况下，这些限制仅适用于针对 O 的应用。 不过，用户可以从 **Settings** 屏幕为任意应用启用这些限制，即使应用并不是以 O 为目标平台。

在大多数情况下，应用都可以使用 [JobScheduler](https://developer.android.google.cn/reference/android/app/job/JobScheduler.html) 作业克服这些限制。 这种方式让应用安排为在未活跃运行时执行工作，不过仍能够使系统可以在不影响用户体验的情况下安排这些作业。

Android 8.0 提供针对 [JobScheduler](https://developer.android.google.cn/reference/android/app/job/JobScheduler.html) 的多个改进，让您可以更轻松地使用计划作业取代服务和广播接收器；如需了解详细信息，请参阅 [JobScheduler 改进](https://developer.android.google.cn/preview/api-overview.html#jobscheduler)。

后台服务限制

在后台中运行的服务会消耗设备资源，这可能降低用户体验。 为了缓解这一问题，系统对这些服务施加了一些限制。

系统可以区分 *前台* 和 *后台* 应用。 （用于服务限制目的的后台定义与[内存管理](https://developer.android.google.cn/topic/performance/memory-overview.html)使用的定义不同；一个应用按照内存管理的定义可能处于后台，但按照能够启动服务的定义又处于前台。）如果满足以下任意条件，应用将被视为处于前台：

* 具有可见 Activity（不管该 Activity 已启动还是已暂停）。
* 具有前台服务。
* 另一个前台应用已关联到该应用（不管是通过绑定到其中一个服务，还是通过使用其中一个内容提供程序）。 例如，如果另一个应用绑定到该应用的服务，那么该应用处于前台：
  + [IME](https://developer.android.google.cn/guide/topics/text/creating-input-method.html)
  + 壁纸服务
  + 通知侦听器
  + 语音或文本服务

如果以上条件均不满足，应用将被视为处于后台。

绑定服务不受影响

这些规则不会对绑定服务产生任何影响。 如果您的应用定义了绑定服务，则不管应用是否处于前台，其他组件都可以绑定到该服务。

处于前台时，应用可以自由创建和运行前台服务与后台服务。 进入后台时，在一个持续数分钟的时间窗内，应用仍可以创建和使用服务。

在该时间窗结束后，应用将被视为处于 *空闲* 状态。 此时，系统将停止应用的后台服务，就像应用已经调用服务的“[Service.stopSelf()](https://developer.android.google.cn/reference/android/app/Service.html#stopSelf())”方法。

在这些情况下，后台应用将被置于一个临时白名单中并持续数分钟。 位于白名单中时，应用可以无限制地启动服务，并且其后台服务也可以运行。

处理对用户可见的任务时，应用将被置于白名单中，例如：

* 处理一条高优先级 [Firebase 云消息传递 (FCM)](https://firebase.google.cn/docs/cloud-messaging/) 消息。
* 接收广播，例如短信/彩信消息。
* 从通知执行 [PendingIntent](https://developer.android.google.cn/reference/android/app/PendingIntent.html)。

在很多情况下，您的应用都可以使用 [JobScheduler](https://developer.android.google.cn/reference/android/app/job/JobScheduler.html) 作业替换后台服务。 例如，CoolPhotoApp 需要检查用户是否已经从朋友那里收到共享的照片，即使该应用未在前台运行。

之前，应用使用一种会检查其云存储的后台服务。 为了迁移到 Android 8.0，开发者使用一个计划作业替换了这种后台服务，该作业将按一定周期启动，查询服务器，然后退出。

在 Android 8.0 之前，创建前台服务的方式通常是先创建一个后台服务，然后将该服务推到前台。

Android 8.0 有一项复杂功能；系统不允许后台应用创建后台服务。 因此，Android 8.0 引入了一种全新的方法，即 Context.startForegroundService()，以在前台启动新服务。

在系统创建服务后，应用有五秒的时间来调用该服务的 [startForeground()](https://developer.android.google.cn/reference/android/app/Service.html#startForeground(int, android.app.Notification)) 方法以显示新服务的用户可见通知。

如果应用在此时间限制内*未*调用 [startForeground()](https://developer.android.google.cn/reference/android/app/Service.html#startForeground(int, android.app.Notification))，则系统将停止服务并声明此应用为 [ANR](https://developer.android.google.cn/training/articles/perf-anr.html)。

广播限制

如果应用注册为接收广播，则在每次发送广播时，应用的接收器都会消耗资源。 如果多个应用注册为接收基于系统事件的广播，这会引发问题；触发广播的系统事件会导致所有应用快速地连续消耗资源，从而降低用户体验。

为了缓解这一问题，Android 7.0（API 级别 25）对广播施加了一些限制，如[后台优化](https://developer.android.google.cn/topic/performance/background-optimization.html)中所述。

Android 8.0 让这些限制更为严格。

* 针对 Android 8.0 的应用无法继续在其清单中为隐式广播注册广播接收器。 *隐式广播*是一种不专门针对该应用的广播。 例如，[ACTION\_PACKAGE\_REPLACED](https://developer.android.google.cn/reference/android/content/Intent.html#ACTION_PACKAGE_REPLACED) 就是一种隐式广播，因为它将发送到注册的所有侦听器，让后者知道设备上的某些软件包已被替换。

不过，[ACTION\_MY\_PACKAGE\_REPLACED](https://developer.android.google.cn/reference/android/content/Intent.html#ACTION_MY_PACKAGE_REPLACED) 不是隐式广播，因为不管已为该广播注册侦听器的其他应用有多少，它都会只发送到软件包已被替换的应用。

* 应用可以继续在它们的清单中注册显式广播。
* 应用可以在运行时使用 [Context.registerReceiver()](https://developer.android.google.cn/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)) 为任意广播（不管是隐式还是显式）注册接收器。
* 需要[签名权限](https://developer.android.google.cn/guide/topics/manifest/permission-element.html#plevel)的广播不受此限制所限，因为这些广播只会发送到使用相同证书签名的应用，而不是发送到设备上的所有应用。

在许多情况下，之前注册隐式广播的应用使用 [JobScheduler](https://developer.android.google.cn/reference/android/app/job/JobScheduler.html) 作业可以获得类似的功能。

例如，一款社交照片应用可能需要不时地执行数据清理，并且倾向于在设备连接到充电器时执行此操作。

之前，应用已经在清单中为 [ACTION\_POWER\_CONNECTED](https://developer.android.google.cn/reference/android/content/Intent.html#ACTION_POWER_CONNECTED) 注册了一个接收器；当应用接收到该广播时，它会检查清理是否必要。 为了迁移到 Android 8.0，应用将该接收器从其清单中移除。

应用将清理作业安排在设备处于空闲状态和充电时运行。

**注**：很多隐式广播当前均已不受此限制所限。 应用可以继续在其清单中为这些广播注册接收器，不管应用针对哪个 API 级别。 有关已豁免广播的列表，请参阅[隐式广播例外](https://developer.android.google.cn/about/versions/oreo/background-broadcasts.html)。

迁移指南

默认情况下，这些更改仅影响针对 O 的应用。 不过，用户可以从 **Settings** 屏幕为任意应用启用这些限制，即使应用并不是以 O 为目标平台。

您可能需要更新应用，使其符合新限制。

了解您的应用如何使用服务。 如果您的应用依赖某些在它处于空闲时于后台运行的服务，您需要替换这些服务。

可能的解决方法包括：

* 如果处于后台时您的应用需要创建一个前台服务，请使用新的 NotificationManager.startServiceInForeground()

方法，而不是创建一个后台服务，然后尝试将其推到前台。

* 如果服务容易被用户注意，请将其设为前台服务。 例如，播放音频的服务始终应为前台服务。

使用 NotificationManager.startServiceInForeground()

而不是 [startService()](https://developer.android.google.cn/reference/android/content/Context.html#startService(android.content.Intent)) 创建服务。

* 寻找一种使用计划作业实现服务功能的方式。 如果服务未在执行容易立即被用户注意到的操作，一般情况下，您都能够使用计划作业。
* 发生网络事件时，请使用 [FCM](https://firebase.google.cn/docs/cloud-messaging/) 选择性地唤醒您的应用，而不是在后台轮询。
* 在应用正常处于前台之前，请推迟后台工作。

检查在您应用的清单中定义的广播接收器。 如果您的清单为显式广播声明了接收器，您必须予以替换。 可能的解决方法包括：

* 通过调用 [Context.registerReceiver()](https://developer.android.google.cn/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)) 而不是在清单中声明接收器的方式在运行时创建接收器。
* 使用计划作业检查条件是否会触发隐式广播。

后台位置限制

为降低功耗，**无论应用的目标 SDK 版本为何**，Android 8.0 都会对后台应用检索用户当前位置的频率进行限制。

如果您的应用在后台运行时依赖实时提醒或运动检测，这一位置检索行为就显得特别重要，必须紧记。

**重要说明**：作为起点，我们只允许后台应用每小时接收几次位置更新。我们将在整个预览版阶段继续根据系统影响和开发者的反馈优化位置更新间隔。

系统会对前台应用和后台应用进行区分。应用满足以下任一条件即视为前台应用：

* 它具有可见的 Activity，无论 Activity 处于启动还是暂停状态。
* 它具有前台服务。
* 另一个前台应用通过绑定到应用的其中一个服务或使用应用的其中一个内容提供程序与应用相连。

如果以上所有条件均不满足，应用即视为后台应用。

前台应用行为得到保留

如果应用在运行 Android 8.0 的设备上处于前台，其位置更新行为将与 Android 7.1.1（API 级别 25）及更低版本上相同。

**警告**：如果您的应用长时间进行近乎实时的位置更新检索，将大幅度缩短设备的电池寿命。

优化应用的位置行为

考虑在您的应用接收位置更新不频繁的情况下其后台运行用例是否根本无法成功。如果属于这种情况，您可以通过执行下列操作之一提高位置更新的检索频率：

* 将您的应用转至前台。
* 使用应用中的某个[前台服务](https://developer.android.google.cn/guide/components/services.html#Foreground)。激活此服务时，您的应用必须在[通知区](https://developer.android.google.cn/guide/topics/ui/notifiers/notifications.html)显示一个持续性的通知。
* 使用 Geofencing API 的元素（例如 [GeofencingApi](https://developers.google.cn/android/reference/com/google/android/gms/location/GeofencingApi) 接口），这些元素针对最大限度减少耗电进行了专门优化。
* 使用被动位置侦听器，它可以在后台应用加快位置请求频率时提高位置更新的接收频率。

**注**：如果您的应用需要访问的位置历史记录包含时间频繁更新，请使用批处理版本的 Fused Location Provider API 元素，例如 [**FusedLocationProviderApi**](https://developers.google.cn/android/reference/com/google/android/gms/location/FusedLocationProviderApi) 接口。当您的应用运行于后台时，此 API 会以高于非批处理版本 API 的频率接收用户的位置。但切记，您的应用批量接收更新的频率仍仅为每小时几次。

受影响的 API

对后台应用位置检索行为的更改影响下列 API：

[**Fused Location Provider (FLP)**](https://developers.google.cn/android/reference/com/google/android/gms/location/FusedLocationProviderApi)

* 如果您的应用运行在后台，位置系统服务只会根据 Android 8.0 行为变更中定义的间隔，按每小时几次的频率为其计算新位置。即使您的应用请求进行更频繁的位置更新，也仍是如此。
* 如果您的应用运行在前台，与 Android 7.1.1（API 级别 25）相比，在位置采样率上不会有任何变化。

**Geofencing**

* 后台应用可以高于接收 Fused Location Provider 更新的频率接收地理围栏转换事件。
* 地理围栏事件的平均响应时间是大约每两分钟一次。

**GNSS Measurements 和 GNSS Navigation Messages**

* 当您的应用位于后台时，注册用于接收 [GnssMeasurement](https://developer.android.google.cn/reference/android/location/GnssMeasurement.html) 和 [GnssNavigationMessage](https://developer.android.google.cn/reference/android/location/GnssNavigationMessage.html) 输出的回调会停止执行。

**Location Manager**

* 提供给后台应用的位置更新只会根据 Android 8.0 行为变更中定义的间隔，按每小时几次的频率提供。

**注**：如果运行您的应用的设备安装了 Google Play 服务，强烈建议您改用 [Fused Location Provider (FLP)](https://developers.google.cn/android/reference/com/google/android/gms/location/FusedLocationProviderApi)。

**WLAN 管理器**

[startScan()](https://developer.android.google.cn/reference/android/net/wifi/WifiManager.html#startScan()) 方法对后台应用执行完整扫描的频率仅为每小时数次。如果不久之后后台应用再次调用此方法， [WifiManager](https://developer.android.google.cn/reference/android/net/wifi/WifiManager.html) 类将提供上次扫描所缓存的结果。