# Scoped Storage原理分析

Scoped Storage是Android Q上针对外部存储的新特性。

在以往的Android版本中，APP只要申请了存储相关的运行时权限，就对完整的外部存储持有读写权限。

在Scoped Storage中，外部存储被分为公共目录及App-specific目录。APP在自身的App-specific中，APP无需任何权限即可读写，而在公共目录中，APP无法直接读写，需要通过MediaStore接口或SAF访问。

也就是说，Android Q相当于在外部存储上为每个APP分配了App-specific目录作为它们的私有空间，APP在外部存储的私有空间上读写是被放宽要求的（实际上没有权限要求）；而在外部存储的其他空间上，APP读写是被更加严格地限制的，具体来说，APP需要申请READ\_EXTERNAL\_STORAGE的运行时权限，才能通过MediaStore接口访问公共目录下的多媒体文件（photos/videos/audio），或者通过SAF接受用户的明确授权。

此外，系统应用申请WRITE\_MEDIA\_STORAGE的要求也更加严格，只有non-user-visible的APP可以申请（《Android Bootcamp 2019 - Privacy Overview.pdf》中要求，虽然暂未对system app做限制）。

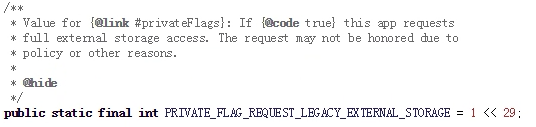
下面将从四个方面分析Scoped Storage的原理：

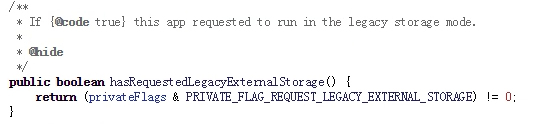
1. 包安装与进程启动中开启新特性
2. 使用路径读写文件
3. SAF原理分析
4. MediaProvider原理分析

## 1.包安装与进程启动中开启新特性

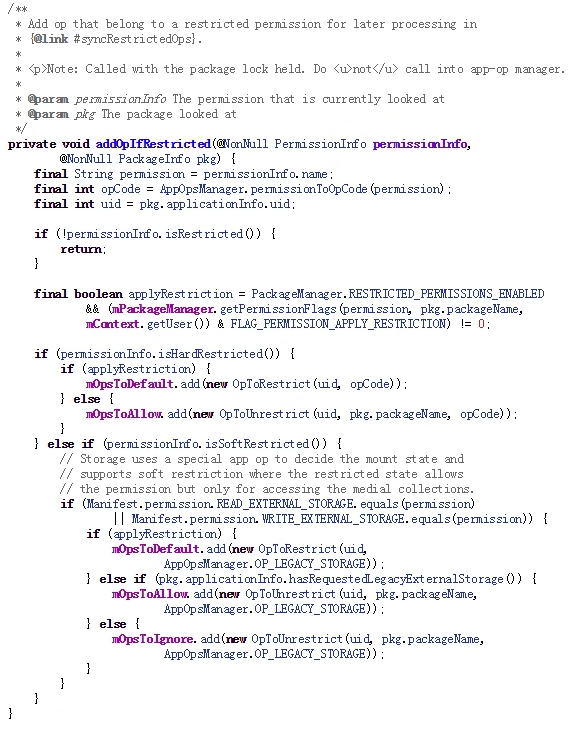
APP可以在AndroidManifest.xml中设置新属性android:requestLegacyExternalStorage=”true | false”来设定APP外部存储视图模式。新属性在PackageParser.java里被解析并添加到ApplicationInfo中。

在ApplicationInfo.java里对应的标志位和向外提供的属性查询方法：





PermissionPolicyService中用到ApplicationInfo中的新属性，用于设定APP的App Ops（AppOpsManager.OP\_LEGACY\_STORAGE）。这一过程的具体流程从PermissionPolicyService.onStartUser开始，包括grantOrUpgradeDefaultRuntimePermissionsIfNeeded/ synchronizePermissionsAndAppOpsForUser/ startWatchingRuntimePermissionChanges。



从注释里可见，AppOpsManager.OP\_LEGACY\_STORAGE是特殊的，支持允许读写权限但只限于媒体类别。注意，只有在APP申请了READ\_EXTERNAL\_STORAGE/WRITE\_EXTERNAL\_STORAGE的情况下，才会设置AppOpsManager.OP\_LEGACY\_STORAGE。

当AppOpsManager.OP\_LEGACY\_STORAGE与APP绑定后，以后可以通过该app op来判断该APP是否开启了legacy view。在APK进程启动过程中，就使用了这个app op来判断存储的mount方式。

APK启动过程：

ActivityManagerService.startProcessLocked

ProcessList.startProcessLocked

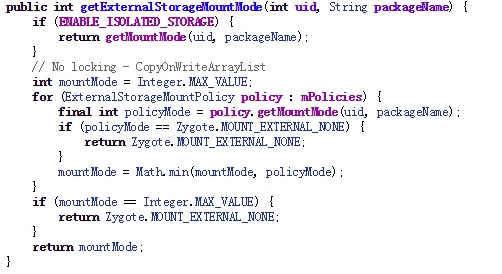
ProcessList.startProcessLocked



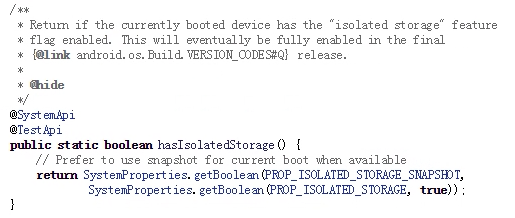
观察构建APP的group id和mount mode的过程：

gids包括从package manager获取的应用的permGids，和一些补充gids，其中UserHandle.getUserGid()获取的是固定gid 9997，对应的是/mnt/runtime/write中文件的group id。

参数mountExtStorageFull默认是false，所以mount mode由StorageManagerService的getExternalStorageMountMode决定：







ENABLE\_ISOLATED\_STORAGE判断是否开启了isolated storage，即设备上的scoped storage新特性开关是否打开了，有两个：PRO\_ISOLATED\_STORAGE\_SNAPTSHOT（高优先级）和PRO\_ISOLATED\_STORAGE。

在Android Q Beta4上，是默认打开新特性的。接着确定mount mode的流程：

getMountMode -> getMountModeInternal



在getMountModeInternal中分几种情况：

1. Isolated process（uid：99000-99999或90000-98999），返回MOUNT\_EXTERNAL\_NONE。
2. 通过PMS判断应用是instant app，返回MOUNT\_EXTERNAL\_DEFAULT。
3. 根据APP申请到的权限：
   1. READ\_EXTERNAL\_STORAGE/ WRITE\_EXTERNAL\_STORAGE分别对应于hasRead/ hasWrite。
   2. WRITE\_MEDIA\_STORAGE权限对应hasFull，从字面意义上就表示了全部权限。WRITE\_MEDIA\_STORAGE权限的等级是signature|privileged的，而且根据CDD最新要求，申请WRITE\_MEDIA\_STORAGE时不仅要求有WRITE\_EXTERNAL\_STORAGE运行时权限，而且User-launchable的应用不允许申请该权限。
   3. INSTALL\_PACKAGES权限或OP\_REQUEST\_INSTALL\_PACKAGES op对应了installer access。
   4. OP\_LEGACY\_STORAGE op对应hasLegacy。它正是之前分析的AppOpsManager.OP\_LEGACY\_STORAGE。

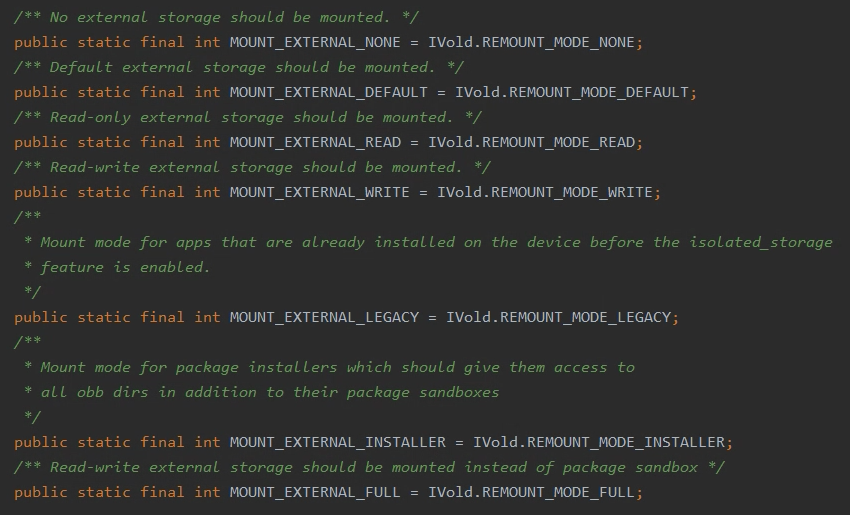
当hasFull并且hasWrite时，返回MOUNT\_EXTERNAL\_FULL。拥有所有设备的完整读写权限。

当是installer并且hasWrite时，返回MOUNT\_EXTERNAL\_WIRTE。

当hasLegacy时：hasWrite对应MOUNT\_EXTERNAL\_WRITE，hasRead对应MOUNT\_EXTERNAL\_READ。读写存储的运行时权限都没有时或者不处于legacy view模式时，返回MOUNT\_EXTERNAL\_DEFAULT。

Mount mode与挂载节点的对应关系：

1. NONE: APP不mount任何节点。
2. DEFAULT: APP mount /mnt/runtime/default下面设备，无法读写外置存储。只能访问APP-specific目录。
3. FULL: 拥有所有存储设备的完整读写权限。
4. READ: 拥有外置存储的读权限。
5. WRITE：拥有外置存储的读写权限。



在设置好gids和mount mode之后，processList会通过zygote fork进程。

ProcessList.startProcessLocked

ProcessList.startProcessLocked

ProcessList.startProcess

Process.start

ZygoteProcess.start

ZygoteProcess.startViaZygote (在这里将mount mode装载到zygote arg)

-> zygoteSendArgsAndGetResult

attemptZygoteSendArgsAndGetResult（通过zygoteWrite发送zygote arg）

-> ZygoteServer.runSelectLoop

ZygoteConnection.processOneCommand（从socketReader中读取zygote arg）

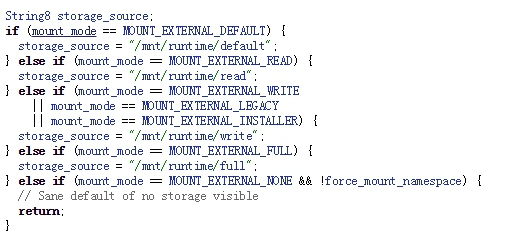
Zygote.forkAndSpecialize

Zygote.nativeForkAndSpecialize

-> com\_android\_internal\_os\_Zygote\_nativeForkAndSpecialize（通过ForkCommon创建一个进程）

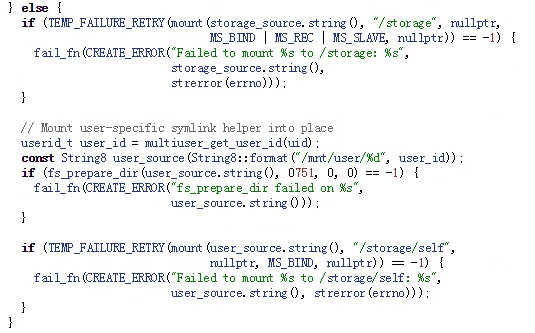
SpecializeCommon（提取参数）

MountEmulatedStorage



在MountEmulatedStorage中确定mount source：





Beta3中的mount方式已经disable了，Beta4中的mount target与Android P相同，将**/mnt/runtime/<mount\_mode> bind mount到/storage**，将**/mnt/user/<user\_id> bind-mount到/storage/self**。

为APP准备sandbox（specific dirs）目录：

以系统应用扫描为例：

PackageManagerService.PackageManagerService()

prepareAppDataAndMigrateLIF

prepareAppDataLIF

prepareAppDataLeafLIF

-> StorageManagerService.prepareSandboxForApp

（定义sandboxId -> shared-<sharedUserId>或packageName）

IVold.prepareSandboxForApp

-> VoldNativeSerivce.prepareSandboxForApp

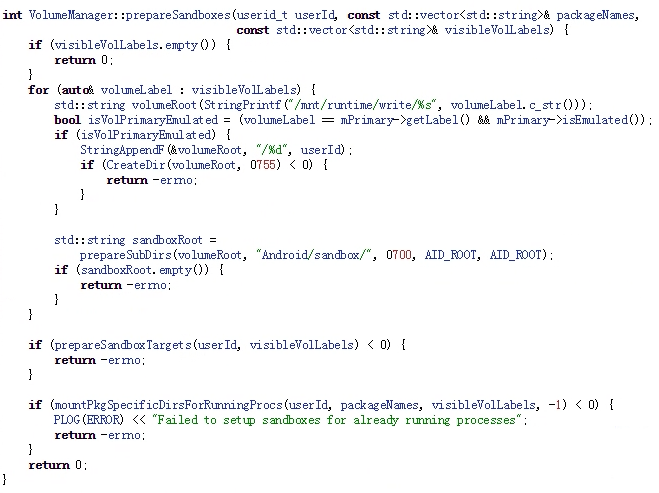
-> VolumeManager.prepareSandboxForApp

1. 检查是否开启了isolated storage模式，user是否已经是start状态，检查APP是否已经在user的包集合中。

2. 将APP包名加入user的包集合中，记录APP的appId和sandboxId。

3. 遍历所有可见volume，把对当前用户可见的volume添加到visibleVolLabels。

-> VolumeManager.prepareSandboxes



1.对所有当前用户可见的volume循环：

volumeRoot = /mnt/runtime/write/<volumeLabel>

（内置卡的外部存储名是emulated）

对于primary volume要加userId：volumeRoot = /mnt/runtime/write/<volumeLabel>/<userId>

（primary volume一般是内置卡的外部存储，所以对主用户来说是/mnt/runtime/write/emulated/0）

然后新建文件夹：sandboxRoot = volumeRoot/Android/sandbox/

（/mnt/runtime/write/emulated/0/Android/sandbox/）

2.新建sandbox targets：

2.1新建mntTargetRoot = /mnt/user/<userId>/package

2.2在所有当前用户可见的volume上：

新建/mnt/user/<userId>/package/<volumeLabel>

对于primary volume还要加userId：/mnt/user/<userId>/package/<volumeLabel>/<userId>

2.3新建mntTargetRoot = /mnt/user/<userId>/package/self

对于primary volume还要加primary：/mnt/user/<userId>/package/self/primary，并symlink到<primaryVolume->getPath()>/<userId>，一般来说是/storage/emulated/0

1. mountPkgSpecificDirsForRunningProcs

rootName = /proc/1/ns/mnt

mntFullSb = /mnt/runtime/full

mntWriteSb = /mnt/runtime/write

obbMountDir = /mnt/user/<userId>/obb\_mount

循环/proc下的子文件夹，找到package对应的pid：

pidName = /proc/<pid>/ns/mnt

调用getMountModeForRunningProc获取mountMode。对于mountMode为FULL/LEGACY/NONE，不做处理。对于mountMode为WRITE，遍历所有可见volume：

/mnt/runtime/write/<volumeLabel>[/userId]/Android/sandbox/<sandboxId> bind mount到 /storage/<volumeLabel>[/userId]

/mnt/runtime/write/<volumeLabel>[/userId]/Android/obb/package bind mount到/storage/<volumeLabel>[/userId]/Android/obb/package（对于mount mode为INSTALLER）

/mnt/runtime/write/<volumeLabel>[/userId]/Android/data/package bind mount到/storage/<volumeLabel>[/userId]/Android/data/package

/mnt/runtime/write/<volumeLabel>[/userId]/Android/media/package bind mount到/storage/<volumeLabel>[/userId]/Android/media/package

/mnt/runtime/write/<volumeLabel>[/userId]/Android/obb bind mount到/storage/<volumeLabel>[/userId]/Android/obb（对于mount mode为INSTALLER）

至此apk sandbox（specific dirs）已经准备好了。

其他情况下为应用准备sandbox（specific dirs），如apk安装、apk enable同样也是最后使用到VolumeManager::prepareSandboxes。

多用户添加的情况，同样也是最后使用到了VolumeManager::prepareSandboxes，主要区别是这时对应的packages是所有已安装的apk，而不是单个apk。

外置sd卡的情况：

StorageManagerService::mount

VoldNativeService::mount

VolumeBase::mount

PublicVolume:doMount

VolumeManager::onVolumeMounted

VolumeManager::prepareSandboxes

同样也是使用VolumeManager为所有应用在新增存储设备上增加sandbox（specific dirs）。

## 2. 使用路径读写文件

使用file path创建一个File类对象，使用File类对象构造OutputStream/InputStream。

FileOutputStream.FileOutputStream(File file, boolean append)（构造函数）

IoBridge.open(String path, int flags)（返回指定路径的文件的fd，flag表示打开文件的方式，如O\_WRONLY | O\_CREAT | O\_APPEND，底层代码将通过flag来确定需要检查的权限）

BlockGuardOs.open(String path, int flags, int mode)（mode仅在创建文件时使用）

Linux.open(String path, int flags, int mode)（native方法）

-> libcore\_io\_Linux::Linux\_open()

open(path.c\_str(), flags, mode)

最后使用了系统调用函数open()来获取文件的fd。

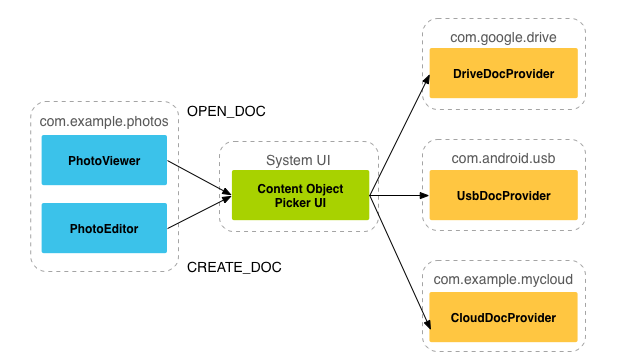
从为APP准备specifc dirs的过程中，可以看到将/mnt/runtime/write/<volumeLabel>/[/userId]/Android/(data/media/obb)/package目录bind mount到/storage/<volumeLabel>/[/userId]/Android/ (data/media/obb)/package目录。所以APP可以直接通过文件路径读写这些specific dirs（包括外置T卡）下的文件。

从开启APP进程的过程中，可以看到将/mnt/runtime/<mount mode> bind mount到/storage。所以mount mode决定了外置存储设备对APP的可见性。如果mount mode是write，那么APP可以直接使用路径对文件进行读写，例如legacy视图模式且申请了WRITE\_EXTERNAL\_STORAGE的情况下；如果mount mode是full，那么APP对所有外部存储设备有读写权限，例如申请了WRITE\_EXTERNAL\_STORAGE和WRITE\_MEDIA\_STORAGE的情况下。具体如何获取APP的mount mode在第一节中已经介绍。

## 3. SAF原理分析

SAF是System Access Framework。APP调用SAF选择文件、选择目录、创建文件，SAF都返回一个Uri给APP。APP利用这个Uri即可读写文件、目录，无需申请任何权限。

SAF包括文档提供程序、文档选取器、客户端程序三部分。



1. 文档提供程序的实体是DocumentsProvider，它继承ContentProvider。想要向外提供文件的APP可以继承实现DocumentsProvider。Android系统已经实现了几个DocumentsProvider，如MtpDocumentsProvider、MediaDocumentsProvider、ExternalStorageProvider、BugreportStorageProvider等。

2. 文档选取器是客户端程序与文档提供程序之间的桥梁。因为文档需要受到保护，DocumentsProvider的子类是必须声明read/write permission为MANAGE\_DOCUMETS，而这个权限的等级是signature，所以只有有系统签名的程序才能直接访问这些DocumentsProvider，从而获取文件的Uri。所以系统提供了文档选择器，即DocumentUI的picker部分。APP通过intent调起DocumentUI的PickActivity，DocumentUI向用户展示有层次的文件系统，用户选择想要授权的文件，DocumentUI返回文件Uri给APP，并加上read/write uri的permission。

在Android 4.3及更低的版本，需要使用ACTION\_PICK或ACTION\_GET\_CONTENT来访问其他应用的文件，用户选择一个提供文件的应用，所选的应用需要自己提供一个UI界面。而在Android 4.4及更高版本，使用ACTION\_OPEN\_DOCUMENT可以调起系统提供的选择器UI（PickActivity）。

注意：CTS要求MANAGE\_DOCUMENTS权限只能有一个package拥有，所以vendor的系统APP即使拥有系统签名，也不能申请这一权限。

1. 客户端程序，即需要访问文件的APP。APP使用的Intent取决于应用的需要：如果APP只是想读取/导入数据，请使用ACTION\_GET\_CONTENT，使用此方法时，APP会导入数据的副本；如果APP想要获取对文件的长期、持久性访问权限，请使用ACTION\_OPEN\_DOCUMENT，例如APP需要编辑DocumentsProvider提供的照片。

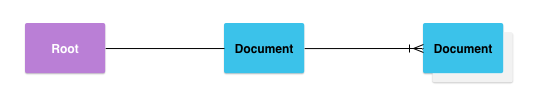
### 3.1 DocumentsProvider

DocumentsProvider提供文件的数据结构：传统的文件层次结构。

1. 一个DocumentsProvider可以提供多棵文件树，每棵树都定义了一个root，用唯一的COLUMN\_ROOT\_ID标记。root表征的目录用一个COLUMN\_DOCUMENT\_ID标记。root表征的目录扇出它下面的文档或目录，这些文档或目录都有COLUMN\_DOCUMENT\_ID标记，而其中的每个目录又可以扇出它下面的文档或目录。这些COLUMN\_DOCUMENT\_ID必须具有唯一性，一旦发放不能更改，因为它们用于所有设备重启过程中的永久性URI授权。

2. 每个文档都可以具有不同的功能，在COLUMN\_FLAGS中标记。如FLAG\_SUPPORTS\_WRITE/ FLAG\_SUPPORTS\_DELETE/ FLAG\_SUPPORTS\_THUMBNAIL。APP获取到文档Uri后，需要查询一下文档的元数据，其中的COLUMN\_FLAGS就展示了这个Uri支持的操作，例如有FLAG\_SUPPORTS\_DELETE才能删除该文档。

3. 不同的目录中可以包含有相同COLUMN\_DOCUMENT\_ID的文档。



4. 定义DocumentsProvider:



exported和grantUriPermissions必须是true，否则在install provider的时候attach info会抛出异常。

permission必须是MANAGE\_DOCUMENT，否则系统文档选择器无法使用定义的DocumentsProvider。

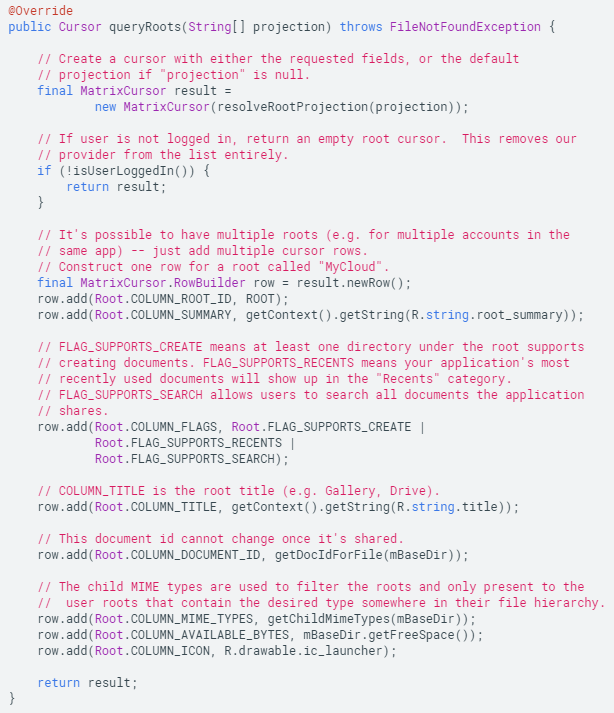
enable根据当前运行设备的API决定true/false，在Android4.3或更低的情况下，为false。同理，如果APP有为ACTION\_GET\_CONTENT定义一个picker activity，那么需要在运行设备为Android4.4或更高时，禁用ACTION\_GET\_CONTENT这个过滤器。否则系统文档选择器打开时，APP会有两个入口。

Intent-filter中的DOCUMENT\_PROVIDER action是用于表征该content provider是用于SAF的document provider，系统的文档选择器选择文档数据源时，就是用这个action来匹配document provider的。

自定义的DocumentsProvider至少需要实现：

1. queryRoots(String[] projection)

返回当前能提供的所有root。当至少有一个root返回时，这个自定义的document provider才会出现在系统文档选择器的第三方栏目中。root的metadata被定义在合约类DocumentsContract.Root中，metadata中的COLUMN\_DOCUMENT\_ID就是表征这个root目录的document id。参数projection表示调用方想要返回的字段集合。



如果这些root发生了变化，需要主动调用notifyChange方法来通知root uri的监听者。

单个root Uri的合约方法：

DocumentsContract.buildRootUri(String authority, String rootId)

总的root Uri的合约方法：

DocumentsContract.buildRootsUri(String authority)

1. queryChildDocuments(String parentDocumentId, String[] projection, String sortOrder)

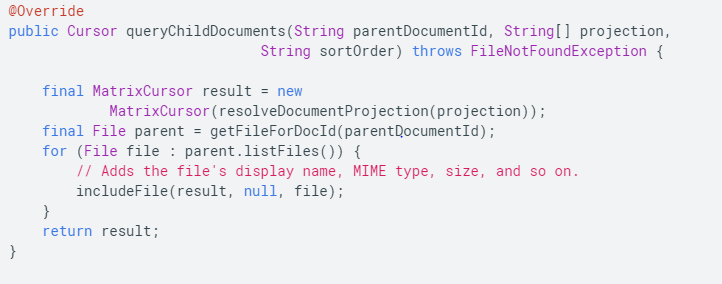
返回指定目录下的子目录和文档，不递归。parentDocumentId是指定目录document id，projection是想要返回的字段集合，sortOrder是想要返回的数据的排序方式。

Target SDK在26（O）及以上的APP应该重写queryChildDocuments(String, String[], Bundle)。

APP如果想要监听某个目录下所有子目录/文档变化的对应的Uri，应该使用的合约方法是：

DocumentsContract.buildChildDocumentsUri(String authority, String parentDocumentId)

DocumentsProvider在notifyChange时也使用这个Uri。



1. queryDocument(String documentId, String[] projection)

返回指定文件的metadata。Document的metadata字段定义在合约DocumentsContract.Document中。



1. openDocument(String documentId, String mode, CancellationSignal signal)

返回指定文件的ParcelFileDescriptor。mode是打开文件的模式，”r”应该总是被支持。如果provider无法支持传入的mode，需要抛出UnsupportedOperationException。CancellationSignal是取消标志，如果打开文件的过程被例如下载文件等事件阻塞，provider需要周期性地检查CancellationSignal.isCanceled()来取消已被抛弃的文件打开请求。



除了以上4个必须实现的方法外，自定义的DocumentsProvider还可以重写其他的方法，例如queryRecentDocuments（如果DocumentsProvider返回的root定义了FLAG\_SUPPORTS\_RECENTS，那么必须实现这个方法，返回最近修改的文件集合，因为默认是抛出异常的）、querySearchDocuments（FLAG\_SUPPORTS\_SEARCH标记某个root是否支持文件搜索，如果支持就需要实现这个方法，queryArgs与文件如何匹配由这个方法的具体实现决定）等等。

### 3.2 系统文档选择器

系统文档选择器位于DocumentUI APP中。DocumentUI的功能package主要有两个：files和picker。files对应了文件管理器，提供了文件管理的UI，用户可以查看、操作所有DocumentsProvider提供的文档；picker就对应了系统文档选择器，APP可以通过Intent调起pickActivity，然后用户选择root以及root下具体的文件/目录。

下面介绍几个比较重要的类。

1. DocumentsApplication

是DocumentUI的Application类。在它里面创建了几个全局的类。ThumbnailCache是一个缓冲缩略图的LruCache；ClipStorage是一个缓存剪切的文件列表到cache目录的类，用于一次性复制大量文件的情况；DocumentClipper则是管理APP内复制剪切操作的类。其中最重要的是ProvidersCache，它缓存了当前系统中所有的DocumentsProvider。



2. ProvidersCache



在DocumentsApplication的onCreate方法中，就会调用一次ProvidersCache的updateAsync方法，进一步就会执行一次UpdateTask，查询所有活跃的DocumentsProvider并缓存从它们中查询到的root，同时也会记录所有Stopped package里的DocumentsProvider。在DocumentUI APP的UI界面起来时，或是真正使用ProvidersCache时，总是会先loadStoppedAuthorities，保证拿到所有的root。

1. PickActivity与它的注入容器

PickActivity是系统文档选择器的UI界面。它的组件从容器Injector<T extends ActionHandler>中获取。



这些组件中，features定义了APP的一些细节属性，如isArchiveCreationEnabled等等；config是当前Activity的一些细节属性，例如isDocumentEnabled等等；prefs是一个Preferences，记录了internal storage root是否要展示；fileTypeLookup是mimeType与label的匹配表；appsRowManager用于保存RootFragment要展示的数据，并控制每个item view的构建与显隐，实际上RootFragment的RootLoader是从ProvidersCache里直接获取了root数据然后加以整理，分隔成多个类别；pickResult记录了选择结果(root/fileCount/fileUri)，其中也定义了一些选择过程中的数据，例如mRepeatedPickTimes，这个数据会被保存到PickCountRecordProvider中；actions，即ActionHandler，定义了各种操作接口，如openRoot、getRootDocument等。

组件中最重要的是Model类：它存储了当前加载的目录的数据模型。在update时通过DirectoryLoader去加载当前目录的子文件/目录信息。DirectoryLoader实际上通过目标目录的root信息找到对应的DocumentsProvider的authority，然后利用DocumentsContract.buildChildDocumentsUri()或者buildSearchDocumentsUri来构建查询Uri，最终会调用到对应的DocumentsProvider的queryChildDocuments或者querySearchDocuments方法。在获取到目标目录的子目录/文件信息后，提取出\_display\_name、document id等字段信息，可以设置到对应的item view中。





1. DocumentStack



mRoot表示当前处于的文档树；mList，维护了从文档树root到当前目录的路径上的各个document的信息，是一个栈。

1. DocumentAccess



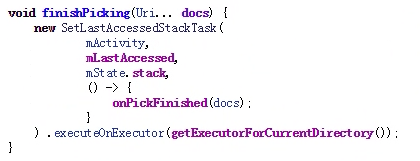
工具类，主要是再次封装了DocumentsContract中的一些合约方法，实现从Uri到DocumentInfo/RootInfo的正向/逆向转换。

1. LastAccessedStorage



LastAccessedProvider中存储了上次选择文档完成时的DocumentStack。在适当的条件下，LastAccessedStorage帮助文档选择器恢复上次的DocumentStack。

在用户选择完成后，会调用到picker包下ActionHandler的finishPicking方法，它首先将当前的document stack整理保存到provider中，然后回调callback的onPickFinished方法。



在onPickFinished方法中，首先将选择的文件Uri保存到Intent中，然后更新pick result（这过程中最重要的是将单选的文件的选择次数更新到provider中），然后根据action不同，将不同的permission flag设置到intent中。最后，通过setResult将结果返回到调用方，关闭文档选择器UI。



### 3.3 SAF的使用者

SAF的使用者，简单来说就是那些需要访问文件的上层APP。由于Android对于APP访问存储设备的限制越来越严格，现在很多情况下APP都不得不使用SAF，让用户清楚地知道APP在干什么。

关于SAF，APP最常用的ACTION是：

1. Intent.ACTION\_OPEN\_DOCUMENT（打开文件，返回文件uri）

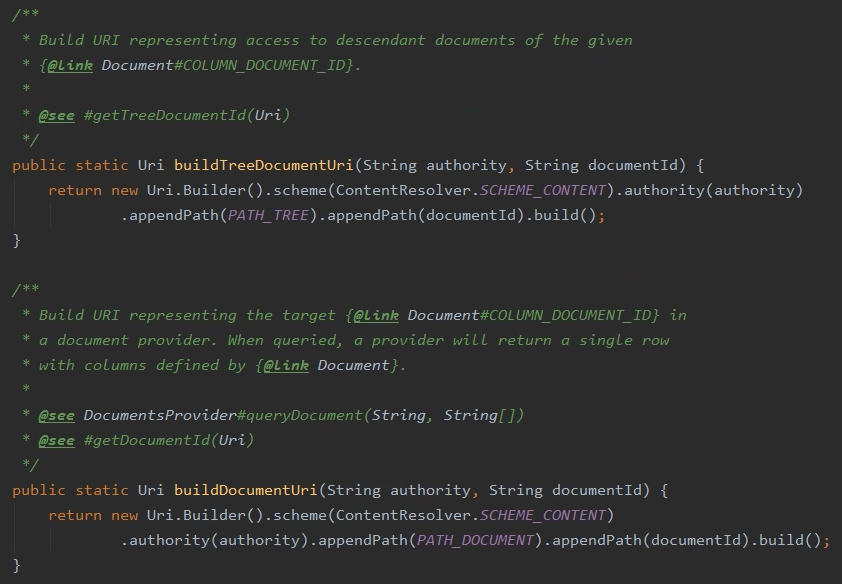
2. Intent.ACTION\_OPEN\_DOCUMENT\_TREE（打开目录，返回目录uriTree）

3. Intent.ACTION\_CREATE\_DOCUMENT（创建文件，返回文件uri）（在SUPPORTS\_CREATE时可用，因为需要DocumentsProvider实现createDocument方法）

常用的接口：

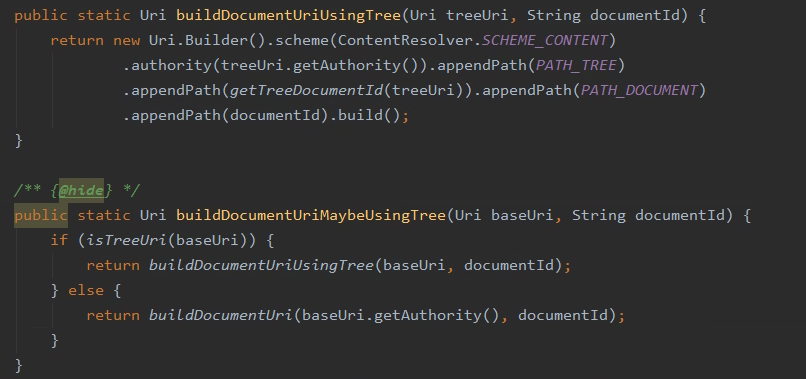
1. DocumentsContract.deleteDocument(getContentResolver(), uri)（在uri对应的文档的FLAG中包含SUPPORTS\_DELETE时可用，因为需要DocumentsProvider实现deleteDocument方法）
2. ContentResolver.takePersistableUriPermission(uri, takeFlags)（用于保留系统授予APP的权限，即获取了永久URI权限。但是这些URI可能会失效，例如其他应用已经修改或删除了文件，所以在使用URI之前，应该始终调用getContentResolver().takePersistableUriPermission()以检查有无最新数据）
3. 更多SAF的使用方法和接口，可以参考DocumentsContract的源码。

构造document Uri和tree document Uri：

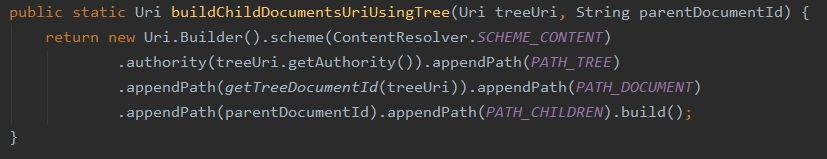


已授权的document uri可以直接打开输入/输出流，或者获取FD进行读写。

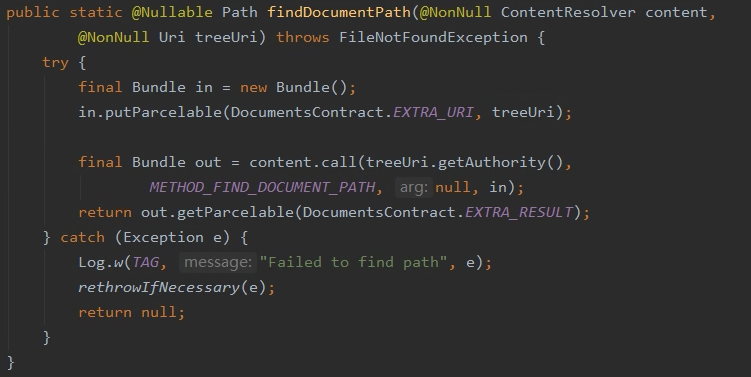
已授权的tree Uri持有对应目录的子目录/文档的权限。查询到子目录/文档的document id后，可以使用DocumentsContract.buildDocumentUriUsingTree()接口来构建文件uri，这个表征文件的uri不用单独授权。而如果直接使用buildDocumentUri来构建uri，得到的uri是需要单独授权的。



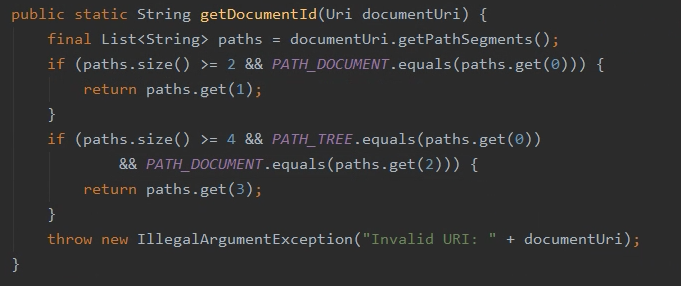
同理，对于buildChildDocumentsUri，也有对应的接口buildChildDocumentsUriUsingTree。

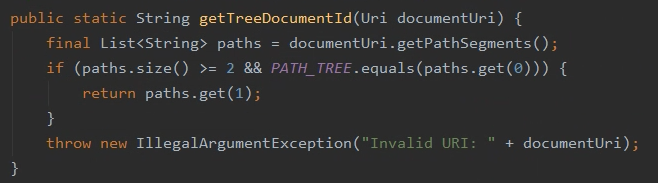


使用DocumentsContract.findDocumentPath获取从文件树根到当前目录uri的路径，返回的Path类对象中包含了从树根到当前目录路径上所有的document id。



从document uri中获取document id时，需要注意tree uri的情况。





使用content://类型Uri读写文件的流程：

1. ContentResolver.openFileDescriptor(Uri uri, String mode)

ContentResolver.openFileDescriptor

ContentResolver.openAssetFileDescriptor

ContentResolver.acquireUnstableProvider

ApplicationContentResolver.acquireUnstableProvider

ActivityThread.acquireProvider

ActivityManagerService.getContentProvider

ActivityManagerService.getContentProviderImpl

ActivityManagerService.checkContentProviderPermissionLocked

UriGrantsManagerService.checkAuthorityGrants

(UriGrantsManagerService是管理Uri授权的服务，它持有一个数据结构 SparseArray<ArrayMap<GrantUri, UriPermission>>保存每个Uid中的uri授权 情况)

UriGrantManagerService.matchesProvider

ContentProvider.Transport.openAssetFile

ContentProvider.enforceFilePermission

ContentProvider.enforceWritePermission

ContentProvider.enforceWritePermissionInner

ContextImpl.checkUriPermission

ActivityManagerService.checkUriPermission

UriGrantsManagerService.checkUriPermission

(由UriGrantsManagerService检查调用者是否有该uri的写权限，在启动 activity、send activity result等情况下会通过grantUriPermissionFromIntent 接口加入数据缓存)

DocumentsProvider.openAssetFile

DocumentProvider.openDocument

1. ContentResolver.openInputStream(Uri uri)

ContentResolver.openInputStream

ContentResolver.openAssetFileDescriptor

......

AssetFileDescriptor.createInputStream

## MediaProvider原理分析

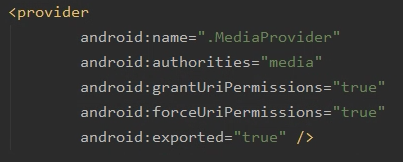
MediaProvider是系统的媒体库应用，它拥有WRITE\_MEDIA\_STORAGE的权限，可以读写所有的存储设备。媒体库应用的核心是MediaProvider.java，即一个内容提供器。

Scoped Storage新特性引入后，APP使用MediaStore提供的接口访问媒体库，媒体库的新表现有：

1. 无需权限，APP可以通过MediaStore接口创建文件，可以读写自己创建的文件。
2. 持有READ\_EXTERNAL\_STORAGE权限，APP可以读其他APP创建的媒体类文件，但删改操作需要用户授权。
3. APP不可读写其他APP创建的非媒体类文件。使用MediaStore.Files接口查询时，只返回媒体类文件。
4. MediaScanner扫描App-specific目录时，只有media路径下的文件被加入到媒体库中。

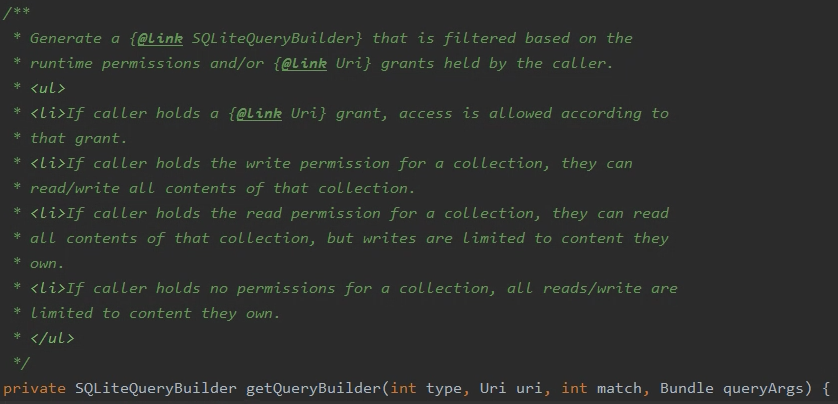
原理分析：

1. MediaProvider不再在AndroidManifest.xml中设定read/write permissions，CRUD操作从ContentProvider.Transport发射时根据操作类型检查enforceRead(Write)Permission一定能通过。从这一层面上，MediaProvider向外完全开放了CRUD操作。实际上，MediaProvider将对查询、修改、删除的限制设置在内部实现中，包括限制查询结果、抛出安全异常等。



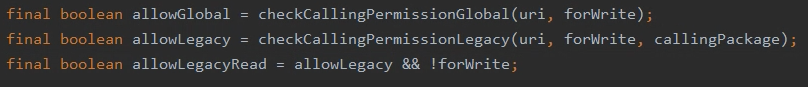
1. 查改删的限制

1) 在query/update/delete操作中，构建SQLiteQueryBuilder时，会根据调用方持有的运行时权限和Uri权限，进行范围过滤。



从注释可知，如果调用方对该uri有权限，则以uri grant为准；如果调用方有运行时写权限，则可以读写；如果有运行时读权限，则可以读，但写的范围只能是自己创建的内容；如果没有任何读写权限，那么只能读写自己创建的内容。

从代码上，getQueryBuilder方法中定义了三个变量：



allowGlobal的含义是本次操作对本uri无限制；allowLegacy的含义是调用方处于legacy模式，且有运行时写权限；allowLegacyRead的含义是在legacy模式下有运行时写权限，但本次操作只需要读权限。

如果allowGlobal为false，在大多数case中会检查调用方是否有本次操作需要的读权限或写权限，如果没有，那么操作的范围会被限制在OWNER\_PACKAGE\_NAME = callingPackage，即调用方自身。在一些case中，则直接返回空的结果。

在FILES/FILES\_DIRECTORY/MTP\_OBJECTS case中，如果非allowGlobal且非allLegacyRead，则会将非媒体类的文件范围限制在package自身，媒体类文件如果有需要的读权限或写权限，则能将范围扩展到全局。如果调用方处于legacy模式下，且有运行时写权限，那么uri对应范围是全局的。对于DOWNLOADS case，如果非allowGlobal且非allLegacyRead，操作范围被限定在package自身。

1. 在insert/update/delete操作中增加了权限检查。

在部分insert case、update media item、部分delete操作中，增加了强制权限检查。

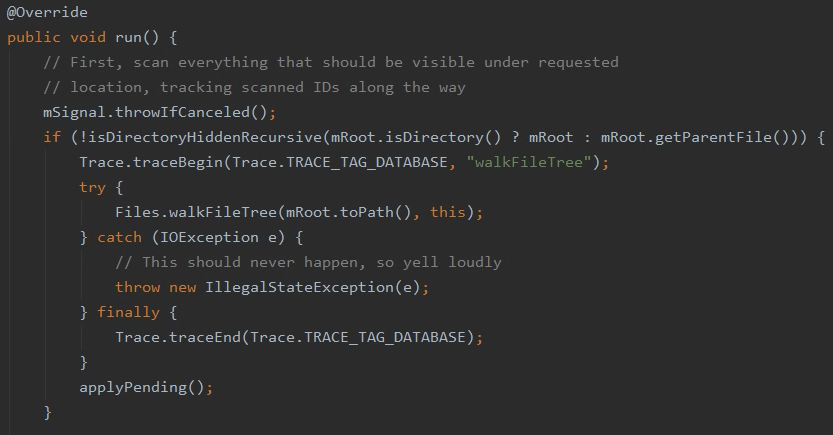
private void enforceCallingPermission(Uri uri, boolean forWrite)

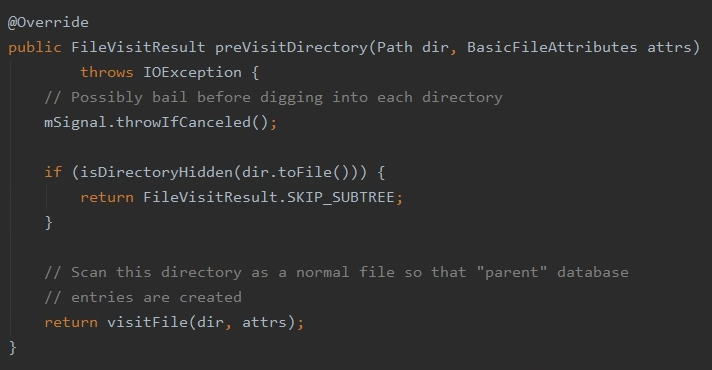
该方法会检查调用方是否有读或写指定uri的权限，如果没有则抛出SecurityException异常。如果调用方想要删写单项媒体文件，但被检测出没有权限，则改为抛出RecoverableSecurityException，该异常中包含了一个用于启动PermissionActivity的PendingIntent。调用方可以catch这个异常，然后取出该PendingIntent，用它来启动PermissionActivity来向用户请求操作该单项媒体文件的权限。

1. 在打开文件时增加了权限检查

从openFile接口或openTypedAssetFile接口打开文件，除了打开媒体文件thumbnail和其他一些legacy的情况，最后都调用到openFileAndEnforcePathPermissionsHelper方法，在其中进行了读写权限检查。

1. ModernMediaScanner扫描文件或目录时，新建并启动Scan。在Scan启动时或walkFileTree进入目录时，都会检查该目录是否是hidden的，如果是，则跳过扫描该目录。





在Android Q中，判断hidden目录新增了App-specific目录的正则匹配判断。判断到App-specific目录中的media目录则强制为可见，其他则强制为不可见。所以MediaScanner扫描时，只有App-specific media目录下的文件被加入到媒体库中。

