Android 10/11分区存储适配

现状：在Android 10之前的版本上，我们在做文件的操作时都会申请存储空间的读写权限。但是这些权限完全被滥用，造成的问题就是手机的存储空间中充斥着大量不明作用的文件，并且应用卸载后它也没有删除掉。另外，这些应用还可能会读取其他应用的一些敏感文件数据。

分区存储:

为了解决外部存储空间文件读写混乱，权限过于宽泛的这个问题，Android 10 中引入了`Scoped Storage`的概念。分区存储改变了应用程序在外部存储空间读写文件的方式。因为外部存储的特定目录下的文件不需要权限也能访问了，而其它目录下的文件则必须使用特定API才能访问。并且分区存储在 Android 11 中得到了进一步的加强。

为什么要适配

1. 在Android10+分区存储模型下，即便获取了外部存储空间读写权限，共享目录页是不能访问的，如果强行访问，会在创建或读写文件的API上报错。
2. 市场上Android10+ 的机型占比达到90%+，应用市场强制要求targetSDKVersion >=29
3. 隐私合规政策强监管下，不合规的APP会被强制整改或下架

下表总结了分区存储如何影响文件访问：

文件位置		文件路径	适用版本	是否需要存储权限	文件访问方式	应用卸载后文件自动删除
内部存储	**应用私有目录**	data/data/packagename	所有版本	否	getFileDir() getCacheDir()	是
外部存储	**应用私有目录**	Android/data/packagename	Android 4.4+	否	getEexternalFileDir() getExternalCacheDir()	是
外部存储	**共享目录**	媒体目录： Pictures、Music、Movies 其它目录： Download、Documents	Android 9-	是	Environment.getExternalSotrageDirectory()	否
				否	Storage Access FrameWork(SAF)	
			Android 10+	否	**MediaStore API** **只能访问自己应用创建的媒体文件**	
				是	**MediaStore API** **能访问其它应用创建的媒体文件**	
			Android 10+	否	**Storage Access FrameWork(SAF)** **访问其它目录非媒体文件(.pdf、.excel)**	

适配的难度

1. 需要先想明白需要存的数据是属于app私有的还是需要共享的
    - 如果是App私有的，可以通过getExternalFilesDir()获取外部空间的应用私有目下。或通过getFileDir()获取内部存储的应用私有目录下
    - 如果是需要共享的，需要通过MediaStore API 存储在共享目录下(dcim, movies,pictures,download...)
2. 对绝对路径相关接口依赖比较深的 APP 适配改动还是挺多的
    - 需要将File API 替换成MediaStore API。MediaStore API , SAF API 这类接口以前就设计好了，现在更加推荐这种方式去操作共享目录下的文件。
3. 适配分为两部分，新数据的存储、老数据的迁移。

新数据的存储

把App所有需要存的数据梳理一遍，对于私有数据我们存到SD卡app私有目录下，对于需要共享的媒体数据我们通过MediaStore的方式。数据放到私有目录很简单我们不讲，主要讲怎么共享媒体数据，以视频为例，看下面的代码：

```
/**
     * 保存共享媒体资源，必须使用先在MediaStore创建表示视频保存信息的Uri，然后通过Uri写入视频数据的方式。
     * 在"分区存储"模型中，这是官方推荐的，因为在Android 10禁止通过File的方式访问媒体资源，Android 11又允许了
     * 从Android 10开始默认是分区存储模型
     *
     *
     * 说明：
     * 此方法中MediaStore默认的保存目录是/storage/emulated/0/video
     * 而Environment.DIRECTORY_MOVIES的目录是/storage/emulated/0/Movies
     * @param context
     * @return
     */
    static Uri getSaveToGalleryVideoUri(Context context, String videoName, String mineType, String subDir) {
        ContentValues values = new ContentValues();
        values.put(MediaStore.Video.Media.DISPLAY_NAME, videoName);
        values.put(MediaStore.Video.Media.MIME_TYPE, mineType);
        values.put(MediaStore.Video.Media.DATE_MODIFIED, System.currentTimeMillis() / 1000);
```

```
        if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.Q) {

 values.put(MediaStore.Video.Media.RELATIVE_PATH,
Environment.DIRECTORY_MOVIES + subDir);
        }

        Uri uri =
context.getContentResolver().insert(MediaStore.Video.Media
.EXTERNAL_CONTENT_URI, values);

        return uri;
    }
```

Sample

- 使用 MediaStore 增删改查媒体集
- 使用 Storage Access Framework 访问文件集

1） 查询媒体集（需要 READ_EXTERNAL_STORAGE 权限）

实际上 MediaStore 是以前就有的 API，不同的是过去主要通过 MediaStore.Video.Media.*DATA这个 colum 请求原始数据，可以得到绝对 Uri，现在需要请求MediaStore.Video.Media.*ID来得到相对Uri再进行处理。

```
/ Need the READ_EXTERNAL_STORAGE permission if accessing
video files that your
// app didn't create.

// Container for information about each video.
```

```
data class Video(
    val uri: Uri,
    val name: String,
    val duration: Int,
    val size: Int
)
val videoList = mutableListOf<Video>()

val projection = arrayOf(
    MediaStore.Video.Media._ID,
    MediaStore.Video.Media.DISPLAY_NAME,
    MediaStore.Video.Media.DURATION,
    MediaStore.Video.Media.SIZE
)

// Show only videos that are at least 5 minutes in
duration.
val selection = "${MediaStore.Video.Media.DURATION} >= ?"
val selectionArgs = arrayOf(
    TimeUnit.MILLISECONDS.convert(5,
TimeUnit.MINUTES).toString()
)

// Display videos in alphabetical order based on their
display name.
val sortOrder = "${MediaStore.Video.Media.DISPLAY_NAME}
ASC"

val query = ContentResolver.query(
    MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
    projection,
    selection,
    selectionArgs,
```

```
    sortOrder
)
query?.use { cursor ->
    // Cache column indices.
    val idColumn =
cursor.getColumnIndexOrThrow(MediaStore.Video.Media._ID)
    val nameColumn =

 cursor.getColumnIndexOrThrow(MediaStore.Video.Media.DISPL
AY_NAME)
    val durationColumn =

 cursor.getColumnIndexOrThrow(MediaStore.Video.Media.DURAT
ION)
    val sizeColumn =
cursor.getColumnIndexOrThrow(MediaStore.Video.Media.SIZE)

    while (cursor.moveToNext()) {
        // Get values of columns for a given video.
        val id = cursor.getLong(idColumn)
        val name = cursor.getString(nameColumn)
        val duration = cursor.getInt(durationColumn)
        val size = cursor.getInt(sizeColumn)

        val contentUri: Uri = ContentUris.withAppendedId(
            MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
            id
        )

        // Stores column values and the contentUri in a
local object
        // that represents the media file.
```

```
            videoList += Video(contentUri, name, duration,
size)
        }
    }
```

2）插入媒体集（无需权限）

```
// Add a media item that other apps shouldn't see until
the item is
// fully written to the media store.
val resolver = applicationContext.contentResolver

// Find all audio files on the primary external storage
device.
// On API <= 28, use VOLUME_EXTERNAL instead.
val audioCollection = MediaStore.Audio.Media
        .getContentUri(MediaStore.VOLUME_EXTERNAL_PRIMARY)

val songDetails = ContentValues().apply {
    put(MediaStore.Audio.Media.DISPLAY_NAME, "My Workout
Playlist.mp3")
    put(MediaStore.Audio.Media.IS_PENDING, 1)
}

val songContentUri = resolver.insert(audioCollection,
songDetails)

resolver.openFileDescriptor(songContentUri, "w", null).use
{ pfd ->
    // Write data into the pending audio file.
}
```

```
// Now that we're finished, release the "pending" status,
and allow other apps
// to play the audio track.
songDetails.clear()
songDetails.put(MediaStore.Audio.Media.IS_PENDING, 0)
resolver.update(songContentUri, songDetails, null, null)
```

3）更新自己创建的媒体集（无需权限）

```
// Updates an existing media item.
val mediaId = // MediaStore.Audio.Media._ID of item to
update.
val resolver = applicationContext.contentResolver

// When performing a single item update, prefer using the
ID
val selection = "${MediaStore.Audio.Media._ID} = ?"

// By using selection + args we protect against improper
escaping of // values.
val selectionArgs = arrayOf(mediaId.toString())

// Update an existing song.
val updatedSongDetails = ContentValues().apply {
    put(MediaStore.Audio.Media.DISPLAY_NAME, "My Favorite
Song.mp3")
}

// Use the individual song's URI to represent the
collection that's
// updated.
val numSongsUpdated = resolver.update(
        myFavoriteSongUri,
```

```
        updatedSongDetails,
        selection,
        selectionArgs)
```

4）更新/删除其它应用创建的媒体集

若已经开启分区存储则会抛出 RecoverableSecurityException，捕获并通过SAF请求权限

```
// Apply a grayscale filter to the image at the given
content URI.
try {
    contentResolver.openFileDescriptor(image-content-uri,
"w")?.use {
        setGrayscaleFilter(it)
    }
} catch (securityException: SecurityException) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
        val recoverableSecurityException =
securityException as?
            RecoverableSecurityException ?:
            throw
RuntimeException(securityException.message,
securityException)

        val intentSender =

recoverableSecurityException.userAction.actionIntent.inten
tSender
        intentSender?.let {
            startIntentSenderForResult(intentSender,
image-request-code,
                    null, 0, 0, 0, null)
        }
```

```
    } else {
        throw RuntimeException(securityException.message,
securityException)
    }
}
```

2. 文件集 （通过 SAF）

1）创建文档

注：创建操作若重名的话不会覆盖原文档，会添加 (1) 最为后缀，如 document.pdf -> document(1).pdf

```
/ Request code for creating a PDF document.
const val CREATE_FILE = 1

private fun createFile(pickerInitialUri: Uri) {
    val intent =
Intent(Intent.ACTION_CREATE_DOCUMENT).apply {
        addCategory(Intent.CATEGORY_OPENABLE)
        type = "application/pdf"
        putExtra(Intent.EXTRA_TITLE, "invoice.pdf")

        // Optionally, specify a URI for the directory
that should be opened in
        // the system file picker before your app creates
the document.
        putExtra(DocumentsContract.EXTRA_INITIAL_URI,
pickerInitialUri)
    }
    startActivityForResult(intent, CREATE_FILE)
}
```

2）打开文档

```
/ Request code for selecting a PDF document.
const val PICK_PDF_FILE = 2

fun openFile(pickerInitialUri: uri) {
    val intent = Intent(Intent.ACTION_OPEN_DOCUMENT).apply
{
        addCategory(Intent.CATEGORY_OPENABLE)
        type = "application/pdf"

        // Optionally, specify a URI for the file that
should appear in the
        // system file picker when it loads.
        putExtra(DocumentsContract.EXTRA_INITIAL_URI,
pickerInitialUri)
    }

    startActivityForResult(intent, PICK_PDF_FILE)
}
```

3）授予对目录内容的访问权限

用户选择目录后，可访问该目录下的所有内容

Android 11 中无法访问 Downloads

```
fun openDirectory(pickerInitialUri: Uri) {
    // Choose a directory using the system's file picker.
    val intent =
Intent(Intent.ACTION_OPEN_DOCUMENT_TREE).apply {
        // Provide read access to files and sub-
directories in the user-selected
        // directory.
```

```
        flags = Intent.FLAG_GRANT_READ_URI_PERMISSION

        // Optionally, specify a URI for the directory
that should be opened in
        // the system file picker when it loads.
        putExtra(DocumentsContract.EXTRA_INITIAL_URI,
pickerInitialUri)
    }

    startActivityForResult(intent, your-request-code)
}
```

4）SAF API 响应

SAF API 调用后都是通过 onActivityResult来相应动作

```
override fun onActivityResult(
        requestCode: Int, resultCode: Int, resultData:
Intent?) {
    if (requestCode == your-request-code
            && resultCode == Activity.RESULT_OK) {
        // The result data contains a URI for the document
or directory that
        // the user selected.
        resultData?.data?.also { uri ->
            // Perform operations on the document using
its URI.
        }
    }
}
```

MediaStore 新增API

系统在调用以上任何一个方法后，会构建一个 PendingIntent 对象。应用调用此 intent 后，用户会看到一个对话框，请求用户同意应用更新或删除指定的媒体文件。

方法	方法说明
createWriteRequest	请求用户授权应用指定媒体文件的写入权限请求
createFavoriteRequest	请求用户授予应用将设备上指定的媒体文件标记为'收藏'的权限
createTrashRequest	请求用户授予映红将指定媒体文件放入设备的临时垃圾箱的权限(垃圾箱中的内容7天后自动永久删除)
createDeleteRequest	请求用户授权应用删除指定的媒体文件的权限

构建一个媒体集的写入操作 createWriteRequest()

createFavoriteRequest() createTrashRequest() createDeleteRequest() 同理

```
val urisToModify = /* A collection of content URIs to
modify. */
val editPendingIntent =
MediaStore.createWriteRequest(contentResolver,
        urisToModify)

// Launch a system prompt requesting user permission for
the operation.
startIntentSenderForResult(editPendingIntent.intentSender,
EDIT_REQUEST_CODE,
    null, 0, 0, 0)

//响应
```

```
override fun onActivityResult(requestCode: Int,
resultCode: Int,
                 data: Intent?) {
    ...
    when (requestCode) {
        EDIT_REQUEST_CODE ->
            if (resultCode == Activity.RESULT_OK) {
                /* Edit request granted; proceed. */
            } else {
                /* Edit request not granted; explain to
the user. */
            }
    }
}
```

适配和兼容

在 targetSDK = 29 APP 中，在 AndroidManifes 设置 requestLegacyExternalStorage="true" 启用兼容模式，以传统分区模式运行。

```
<manifest ... >
  <application android:requestLegacyExternalStorage="true" ... >
```

> 注意：如果某个应用在安装时启用了传统外部存储，则该应用会保持此模式，直到卸载为止。无论设备后续是否升级为搭载 Android 10 或更高版本，或者应用后续是否更新为以 Android 10 或更高版本为目标平台，此兼容性行为均适用。

意思就是在新系统新安装的应用才会启用，覆盖安装会保持传统分区模式，例如：

系统通过 OTA 升级到 Android 10/11
应用通过更新升级到 targetSdkVersion >= 29

新增权限

MANAGE_EXTERNAL_STORAGE : 类似以前的 READ_EXTERNAL_STORAGE + WRITE_EXTERNAL_STORAGE，除了应用专有目录都可以访问。

应用可通过执行以下操作向用户请求名为所有文件访问权限的特殊应用访问权限：

在清单中声明 MANAGE_EXTERNAL_STORAGE 权限。 使用 ACTION_MANAGE_ALL_FILES_ACCESS_PERMISSION intent 操作将用户引导至一个系统设置页面，在该页面上，用户可以为您的应用启用以下选项：授予所有文件的管理权限。 在 Google Play 上架的话，需要提交使用此权限的说明，只有指定的几种类型的 APP 才能使用。