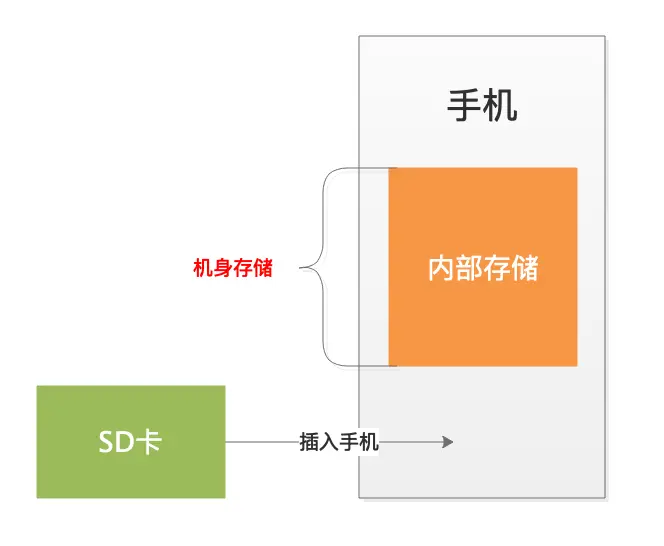
# Android 文件存储

# Android-存储基础

## 存储划分

### Android 4.4 之前

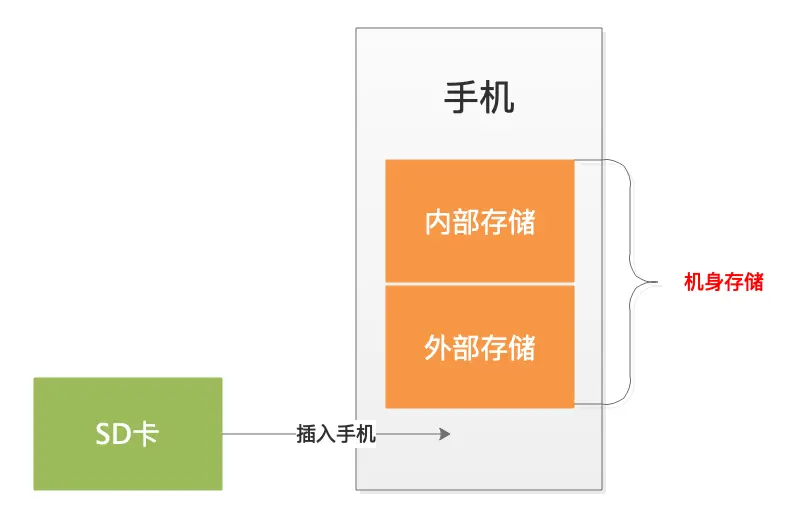
在Android 4.4 之前，由于硬件发展受限，手机自身的存储空间有限，需要通过外置SD卡来扩展存储空间。



如上图，手机自身的存储空间，称之为机身存储，在Android 4.4 之前作为内部存储使用。当然内部存储空间一般是不够用的，所以需要通过插入外置SD卡来扩充存储空间，这当做外部存储。

### Android 4.4之后

在Android 4.4 之后(含)，手机机身存储扩大了：



如上图，机身存储划分为两部分：

**1、内部存储**

**2、外部存储**

当然，依然可以插入SD卡来扩充存储空间，这部分的存储空间称为扩展的外部存储空间。只是现在机身存储都比较大，很少插入SD卡了。

接下来将以Android 4.4 之后的存储划分来分析具体的存储方案。

## 内部存储

### 存放位置

回想一下平时使用的持久化方案：

1、SharedPreferences---->适用于存储小文件

2、数据库---->存储结构比较复杂的大文件

以上这些文件都是默认放在内部存储里。

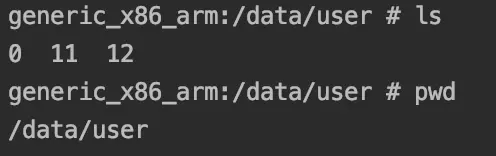
"/" 表示根目录，内部存储里给每个应用按照其包名各自划分了目录，假设App的包名为：com.fish.myapplication

那么该文件在内部存储里的目录为：

/data/user/0/com.fish.myapplication/

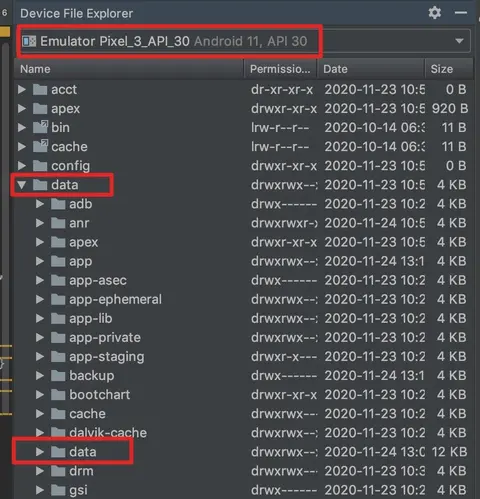
第一个"/"表示根目录，其后每个"/"表示目录分割符。

"0" 表示是第一个用户，后续添加了多用户则生成相应的用户目录：



如上图，新增了两个用户，生成的目录分别是："11"、"12"。目前来说，很少开启多用户的。

一般来说，adb shell里是没有权限查看/data目录的。若要查看内部存储，通常是通过Android Studio侧边栏Device File Explorer选择对应的目标设备查看。



同样的，如果包名为：com.fish.myapplication，则对应的内部存储目录为：

/data/data/com.fish.myapplication/

/data/user/0/com.fish.myapplication/ 会将值转换到

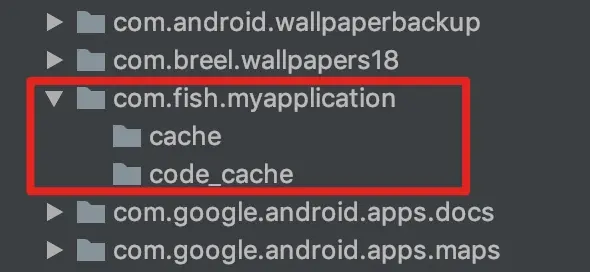
/data/data/com.fish.myapplication/ 路径下。

每个App的内部存储空间仅允许自己访问(除非有更高的权限，如root)，程序卸载后，该目录也会被删除。

### 存储内容

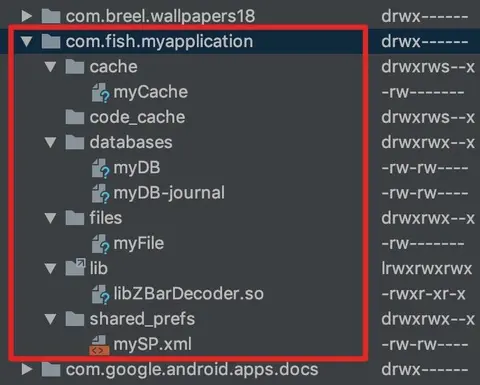
除了SharedPreferences、数据库文件，内部存储还存放了哪些文件呢？

为方便起见，只查看/data/data/目录下的。



刚开始有只有两个空目录。

当进行写入SharedPreferences，创建数据库、写入文件等操作后新增了几个目录：



大致介绍一下以上目录作用：

1、cache-->存放缓存文件

2、code\_cache-->存放运行时代码优化等产生的缓存

3、databases-->存放数据库文件

4、files-->存放一般文件

5、shared\_prefs-->存放SharedPreferences 文件

6、lib-->存放App依赖的so库 是软链接，指向/data/app/ 某个子目录下

### 访问方式

既然知道了各类文件存储的目录，那么如何读写这些文件呢？

我们知道在Java 的世界里，操作文件有两种方式：

字符流和字节流

以字节流为为例，一个简单的读取写入文件Demo:

//写入文件  
private void writeFile(String filePath) {  
 if (TextUtils.isEmpty(filePath))  
 return;  
  
 try {  
 File file = new File(filePath);  
 FileOutputStream fileOutputStream = new FileOutputStream(file);  
 BufferedOutputStream bos = new BufferedOutputStream(fileOutputStream);  
 String writeContent = "hello world\n";  
 bos.write(writeContent.getBytes());  
 bos.flush();  
 bos.close();  
  
 } catch (Exception e) {  
  
 }  
}  
  
//从文件读取  
private void readFile(String filePath) {  
 if (TextUtils.isEmpty(filePath))  
 return;  
  
 try {  
 File file = new File(filePath);  
 FileInputStream fileInputStream = new FileInputStream(file);  
 BufferedInputStream bis = new BufferedInputStream(fileInputStream);  
 byte[] readContent = new byte[1024];  
 int readLen = 0;  
 while (readLen != -1) {  
 readLen = bis.read(readContent, 0, readContent.length);  
 if (readLen > 0) {  
 String content = new String(readContent);  
 Log.*d*("test", "read content:" + content.substring(0, readLen));  
 }  
 }  
 fileInputStream.close();  
 } catch (Exception e) {  
  
 }  
}

可以看出，通过FileInputStream/FileOutputStream构造函数传入File对象即可实现文件读写，而File对象的构造依赖于文件的存放路径，因此重点在于如何获取文件的路径。

分别说明各个目录下文件的读写：

#### 读写files目录下文件

#Context.java  
public abstract File getFilesDir();

使用方式：

private String getFilePath(Context context) {  
 //获取files根目录  
 File fileDir = context.getFilesDir();  
 //获取文件  
 File myFile = new File(fileDir, "myFile");  
 return myFile.getAbsolutePath();  
}

context.getFilesDir()的结果是返回files目录：

/data/user/0/com.fish.myapplication/files/

拿到对应文件的File对象后，构造相应的输入输出流即可实现对该文件的读写。可以看出，过程虽然简单但是有点枯燥，因此Google将这些步骤封装好了，直接返回对应文件的FileOutputStream/FileInputStream：

#Context.java  
public abstract FileInputStream openFileInput(String name)  
 throws FileNotFoundException;  
  
public abstract FileOutputStream openFileOutput(String name, @FileMode int mode)  
 throws FileNotFoundException;

其中name 表示文件名，mode表示访问权限。

#### 读写cache目录下文件

与读取files目录相似：

#Context.java  
public abstract File getCacheDir();

context.getCacheDir()的结果是返回cache目录：

/data/user/0/com.fish.myapplication/cache/

#### 读写shared\_prefs目录下文件

SharedPreferences 提供了简易的快速持久化数据的方案。

private void testSP(String fileName, String key, String value) {  
 if (TextUtils.isEmpty(fileName) || TextUtils.isEmpty(key) || TextUtils.isEmpty(value))  
 return;  
  
 //构造SP文件  
 SharedPreferences sp = getSharedPreferences(fileName, *MODE\_PRIVATE*);  
  
 //写入SP  
 sp.edit().putString(key, value).commit();  
  
 //读取SP  
 String myValue = sp.getString(key, "");  
}

其内部也是使用了输入输出流，以写入SP文件为例：

#SharedPreferencesImpl.java  
private void writeToFile(MemoryCommitResult mcr, boolean isFromSyncCommit) {  
 ...  
 //构造输出流  
 FileOutputStream str = createFileOutputStream(mFile);  
 XmlUtils.writeMapXml(mcr.mapToWriteToDisk, str);  
 FileUtils.sync(str);  
 str.close();  
 ...  
}

#### 读写数据库目录下文件

创建数据库：

MyDatabaseHelper myDatabaseHelper = new MyDatabaseHelper(v.getContext(), "myDB", null, 10);

myDB是数据库文件名。打开数据库的相应表，即可读写数据。

获取数据库文件路径：

#Context.java  
Context.public abstract File getDatabasePath(String name);

获取结果如下：

/data/user/0/com.fish.myapplication/databases/myDB

#### 读写code\_cache目录下文件

#Context.java API>=21  
public abstract File getCodeCacheDir();

获取结果如下：

/data/user/0/com.fish.myapplication/code\_cache/

以上是分别列举了各个子目录/文件的获取方式，如果想获取：/data/user/0/com.fish.myapplication/，可通过：

#Context.java  
public abstract File getDataDir();

该方法需要API>=24。

## 外部存储

外部存储分为两部分：自带外部存储和扩展外部存储(外置SD卡)

### 自带外部存储存储

#### 存放位置

存储的根目录是："/"。

根目录下几个需要关注的目录：

/data/

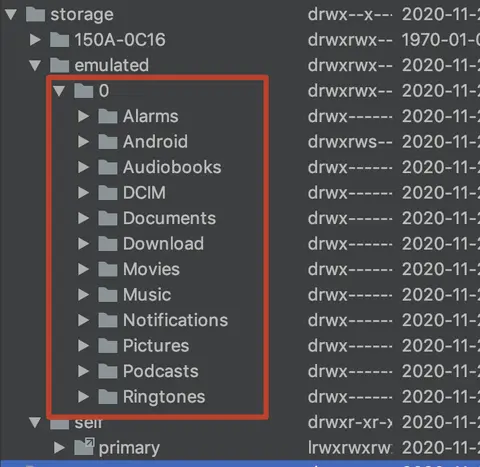
/sdcard/

/storage/

其中/data/目录前面已经分析过。

/sdcard/是软链接，指向/storage/self/primary

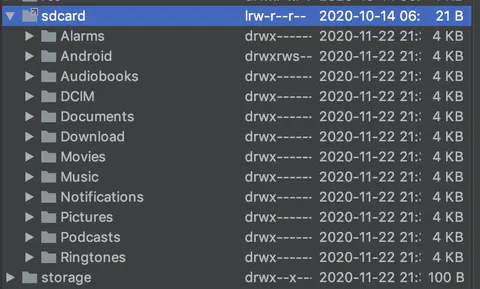
而/storage/下有几个目录：



/storage/self/primary/是软链接，指向/storage/emulated/0/

也就是说/sdcard/、/storage/self/primary/ 真正指向的是/storage/emulated/0/

#### 存储内容



如上图所示，/sdcard/目录下的子目录看起来都比较眼熟。

这些子目录分为分为三部分：

##### 第一部分：共享存储空间

也就是所有App共享的部分，比如相册、音乐、铃声、文档等。

共享存储空间按文件类型又分为两部分：

**1、媒体文件**

DCIM/ 和 Pictures/-->存储图片

DCIM/、Movies/ 和 Pictures-->存储视频

Alarms/、Audiobooks/、Music/、Notifications/、Podcasts/ 和 Ringtones/-->存储音频文件

Download/-->下载的文件

**2、文档和其它文件**

Documents-->存储如.pdf类型等文件

##### 第二部分：App外部私有目录

Android/data/--->存储各个App的外部私有目录

与内部存储类似，命名方式是：Android/data/xx------>xx指应用的包名。

如：/sdcard/Android/data/com.fish.myapplication

##### 第三部分：其它目录

比如各个App在/sdcard/目录下创建的目录，如支付宝创建的目录：alipy/，微博创建的目录：com.sina.weibo/，qq创建的目录：com.tencent.mobileqq/等。

#### 访问方式

与访问内部存储文件类似，外部存储也可以通过构造输入输出流访问文件。

#### 读写共享存储空间

视频、图片等可能分散存储在各个不同的目录里，如果想要获取所有的图片地址，那么得需要遍历不同的目录寻找，效率显而易见的低。Android 将视频、图片等信息存储在数据库里，每当某个App想要访问这些共享的媒体文件时只需要查找数据库对应的表，读取符合条件的行，找出每个媒体的文件路径等信息。

App查询共享存储空间的媒体方式是：通过ContentProvider访问。

访问媒体文件

以查询图片为例：

private void getImagePath(Context context) {  
 ContentResolver contentResolver = context.getContentResolver();  
 Cursor cursor = contentResolver.query(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, null, null, null, null);  
 while(cursor.moveToNext()) {  
 String imagePath = cursor.getString(cursor.getColumnIndex(MediaStore.Images.ImageColumns.DATA));  
 }  
}

查询到图片的地址，当然就可以展示图片了。

访问文档和其它文件

Storage Access Framework 简称SAF：存储访问框架

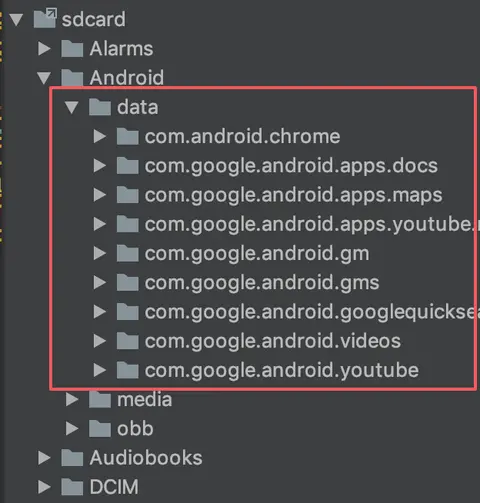
以查看.pdf文件为例：

private void startSAF() {  
 Intent intent = new Intent(Intent.*ACTION\_OPEN\_DOCUMENT*);  
 intent.addCategory(Intent.*CATEGORY\_OPENABLE*);  
 intent.setType("application/pdf");  
 startActivityForResult(intent, 100);  
}  
  
@Override  
protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {  
 super.onActivityResult(requestCode, resultCode, data);  
  
 if (requestCode == 100) {  
 Uri uri = data.getData();  
 }  
}

SAF实际上就是调用系统提供的选择器，选中后在onActivityResult(xx)里接收结果，拿到Uri后当然就可以读写对应的文件了。

#### 读写App外部私有目录

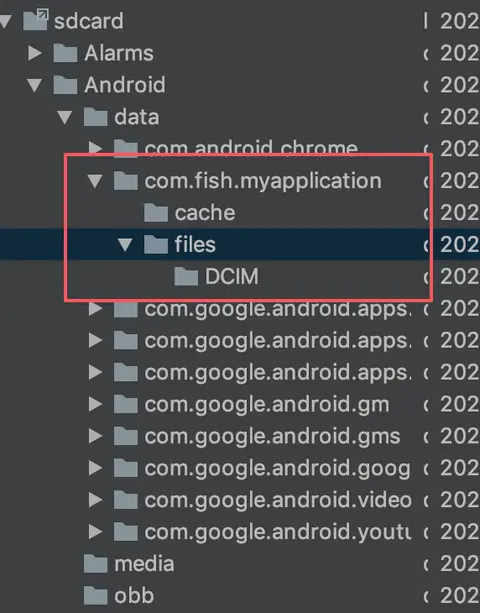
刚开始并没有自己App的包名。



调用如下方法后：

private void testAppDir(Context context) {  
 //4个基本方法  
 File fileDir = context.getExternalFilesDir(null);  
 //API>=19  
 File[] fileList = context.getExternalFilesDirs(null);  
  
 File cacheDir = context.getExternalCacheDir();  
 //API>=19  
 File[] cacheList = context.getExternalCacheDirs();  
  
 //指定目录，自动生成对应的子目录  
 File fileDir2 = context.getExternalFilesDir(Environment.DIRECTORY\_DCIM);  
}

再查看目录树：



可以看出再/sdcard/Android/data/目录下生成了com.fish.myapplication/目录，该目录下有两个子目录分别是：files/、cache/。当然也可以选择创建其它目录。

当App卸载的时候，两者都会被清除。

#### 读写其它目录

只要拿到根目录，就可以遍历寻找其它子目录/文件。

private void testOtherDir(Context context) {  
 File rootDir = Environment.getExternalStorageDirectory();  
}

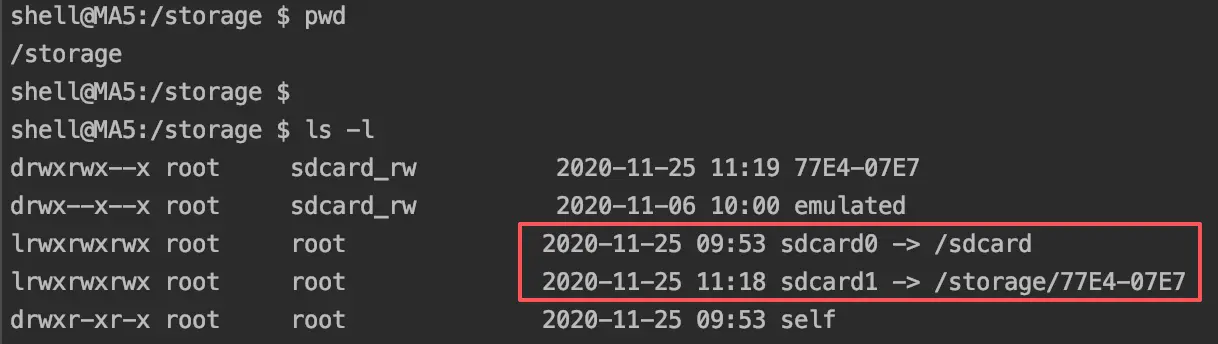
返回的rootDir路径：/storage/emulated/0/。

### 扩展外部存储(外置SD卡)

#### 存储位置

当给设备插入SD卡后，查看其目录：

/sdcard/ 依然指向/storage/self/primary，继续来看/storage/:



可以看出，多了sdcard1，软链接指向了/storage/77E4-07E7/。

#### 存储内容

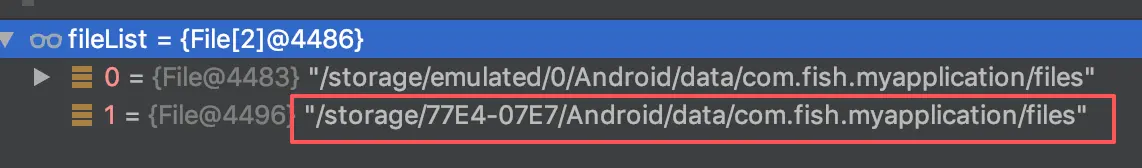
取决于SD卡上装了什么东西。

#### 访问方式

还记得上面获取外部存储-App私有目录方式吗？

File[] fileList = context.getExternalFilesDirs(null);

返回File对象数组，当有多个外部存储时候，存储在数组里。

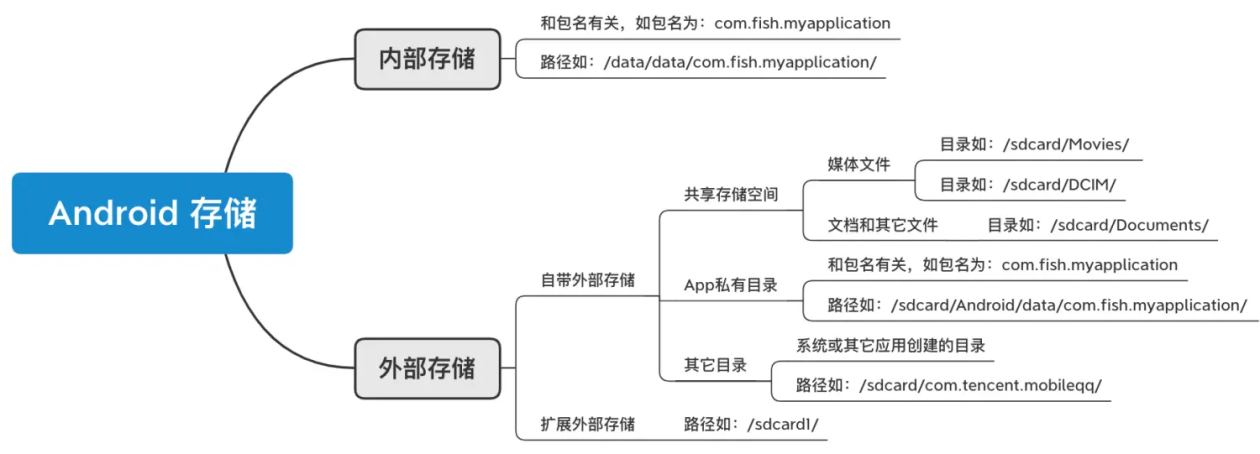


返回的数组有两个元素，一个是自带外部存储存储，另一个是刚插入的SD卡。

拿到路径后，当然就可以访问相应的文件了。

## 易混淆点说明

以上分别阐述了内部存储、自带外部存储、扩展外部存储等，这几者关系如下：



其中比较容易混淆的是：

内部存储与外部存储里的App私有目录，两者命名风格很像。

### 不同点：

/data/data/com.fish.myapplication/ 位于内部存储，一般用于存储容量较小的，私密性较强的文件。而/sdcard/Android/data/com.fish.myapplication/ 位于外部存储，作为App私有目录，一般用于存储容量较大的文件，即使删除了也不影响App正常功能。

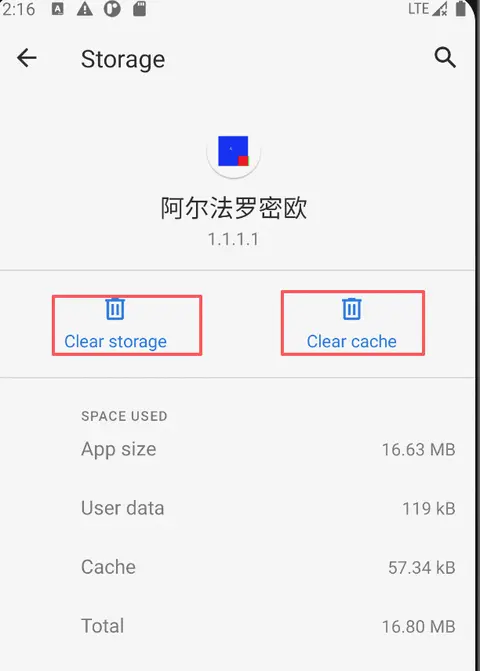
### 相同点：

1、属于App专属，App自身访问两者无需任何权限。

2、App卸载后，两者皆被删除。

3、两者目录下增加的文件最终会被统计到"设置->存储和缓存"里。

另外，常见的在设置里的"存储与缓存"项：



当点击"Clear cache" 时：

内部存储/data/data/com.fish.myapplication/cache/

/data/data/com.fish.myapplication/code\_cache/目录会被清空

外部存储/sdcard/Android/data/com.fish.myapplication/cache/ 会被清空

当点击"Clear storage" 时：

内部存储/data/data/com.fish.myapplication/下除了lib/，其余子目录皆被删除

外部存储/sdcard/Android/data/com.fish.myapplication/被清空

{注：该功能慎用，因为会删除用户数据库，SP文件等，相当于重置了App}

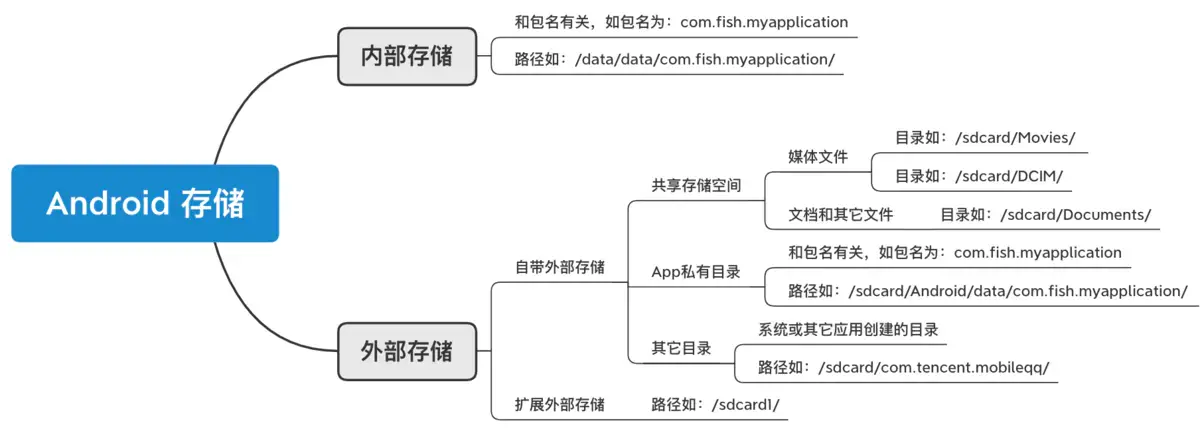
# Android 字符流和字节流读取文件

<https://www.cnblogs.com/jun9207/p/5197697.html>

# Android-10、11-存储完全适配(上)

## 存储基本知识

先来看看存储区域划分：



其中，以下目录无需存储权限即可访问：

1、App自身的内部存储

2、App自身的自带外部存储-私有目录

剩下的都需要申请存储权限，Android 10.0前后对于存储作用域访问的区别就体现在如何访问剩余这些目录内的文件。

**重点在自带外部存储之共享存储空间和其它目录**

## Android 10.0 之前访问方式

继续细分为Android 6.0 之前和之后。

### Android 6.0 之前访问方式

Android 6.0 之前是无需申请动态权限的，在AndroidManifest.xml 里声明存储权限：

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />  
<uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE" />

就可以访问共享存储空间、其它目录下的文件了。

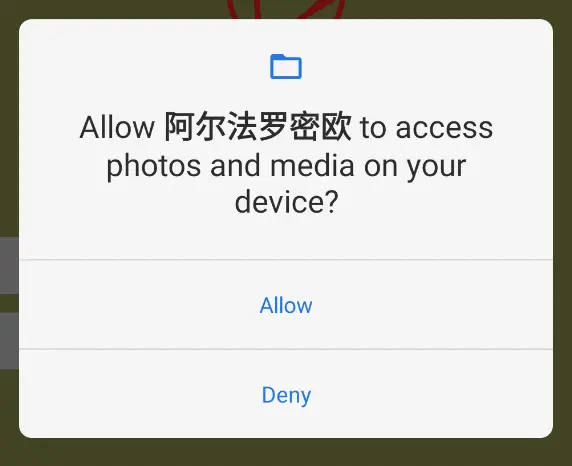
### Android 6.0 之后的访问方式

#### 动态申请权限

Android 6.0 后需要动态申请权限，除了在AndroidManifest.xml 里声明存储权限外，还需要在代码里动态申请。

//检查权限，并返回需要申请的权限列表  
private List<String> checkPermission(Context context, String[] checkList) {  
 List<String> list = new ArrayList<>();  
 for (int i = 0; i < checkList.length; i++) {  
 if (PackageManager.PERMISSION\_GRANTED != ActivityCompat.checkSelfPermission(context, checkList[i])) {  
 list.add(checkList[i]);  
 }  
 }  
 return list;  
}  
  
//申请权限  
private void requestPermission(Activity activity, String requestPermissionList[]) {  
 ActivityCompat.requestPermissions(activity, requestPermissionList, 100);  
}  
  
//用户作出选择后，返回申请的结果  
@Override  
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {  
 if (requestCode == 100) {  
 for (int i = 0; i < permissions.length; i++) {  
 if (permissions[i].equals(Manifest.permission.WRITE\_EXTERNAL\_STORAGE)) {  
 if (grantResults[i] == PackageManager.PERMISSION\_GRANTED) {  
 Toast.makeText(MainActivity.this, "存储权限申请成功", Toast.LENGTH\_SHORT).show();  
 } else {  
 Toast.makeText(MainActivity.this, "存储权限申请失败", Toast.LENGTH\_SHORT).show();  
 }  
 }  
 }  
 }  
}  
  
//测试申请存储权限  
private void testPermission(Activity activity) {  
 String[] checkList = new String[]{Manifest.permission.WRITE\_EXTERNAL\_STORAGE, Manifest.permission.READ\_EXTERNAL\_STORAGE};  
 List<String> needRequestList = checkPermission(activity, checkList);  
 if (needRequestList.isEmpty()) {  
 Toast.makeText(MainActivity.this, "无需申请权限", Toast.LENGTH\_SHORT).show();  
 } else {  
 requestPermission(activity, needRequestList.toArray(new String[needRequestList.size()]));  
 }  
}

申请权限后，提示用户作出选择：



#### 访问文件

权限申请成功后，即可对自带外部存储之共享存储空间和其它目录进行访问。

分别以共享存储空间和其它目录为例，阐述访问方式：

#### 访问共享存储空间

共享存储空间分为两类文件：媒体文件和文档/其它文件。

#### 访问媒体文件

目的是拿到媒体文件的路径，有两种方式获取路径：

##### 1、直接构造路径

以图片为例，假设图片存储在/sdcard/Pictures/目录下。

private void testShareMedia() {  
 //获取目录：/storage/emulated/0/  
 File rootFile = Environment.getExternalStorageDirectory();  
 String imagePath = rootFile.getAbsolutePath() + File.*separator* + Environment.DIRECTORY\_PICTURES + File.*separator* + "myPic.png";  
 Bitmap bitmap = BitmapFactory.*decodeFile*(imagePath);  
}

如上，myPic.png的路径：/storage/emulated/0/Pictures/myPic.png，拿到路径后就可以解析并获取Bitmap。

##### 2、通过MediaStore获取路径

沿用上篇的demo:

private void getImagePath(Context context) {  
 ContentResolver contentResolver = context.getContentResolver();  
 Cursor cursor = contentResolver.query(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, null, null, null, null);  
 while(cursor.moveToNext()) {  
 String imagePath = cursor.getString(cursor.getColumnIndex(MediaStore.Images.ImageColumns.DATA));  
 Bitmap bitmap = BitmapFactory.*decodeFile*(imagePath);  
 break;  
 }  
}

同样的，也是拿到图片路径后获取Bitmap。

还有一种不直接通过路径访问的方法：

##### 3、通过MediaStore获取Uri

1. private void getImagePath(Context context) {  
    ContentResolver contentResolver = context.getContentResolver();  
    Cursor cursor = contentResolver.query(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, null, null, null, null);  
    while(cursor.moveToNext()) {  
    //获取唯一的id  
    long id = cursor.getLong(cursor.getColumnIndexOrThrow(MediaStore.MediaColumns.\_ID));  
    //通过id构造Uri  
    Uri uri = ContentUris.withAppendedId(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, id);  
    openUri(uri);  
    break;  
    }  
   }

与直接拿到路径不同的是，此处拿到的是Uri。图片的信息封装在Uri里，通过Uri构造出InputStream，再进行图片解码拿到Bitmap

#### 访问文档和其它文件

##### 直接构造路径

与媒体文件一样，可以直接构造路径访问。

##### 通过SAF访问

Storage Access Framework 简称SAF：存储访问框架。相当于系统内置了文件选择器，通过它可以拿到想要访问的文件信息。

同样的以获取图片为例：

private void startSAF() {  
 Intent intent = new Intent(Intent.*ACTION\_OPEN\_DOCUMENT*);  
 intent.addCategory(Intent.*CATEGORY\_OPENABLE*);  
 //选择图片  
 intent.setType("image/jpeg");  
 startActivityForResult(intent, 100);  
}  
  
@Override  
protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {  
 super.onActivityResult(requestCode, resultCode, data);  
  
 if (requestCode == 100) {  
 //选中返回的图片封装在uri里  
 Uri uri = data.getData();  
 openUri(uri);  
 }  
}  
  
private void openUri(Uri uri) {  
 try {  
 //从uri构造输入流  
 InputStream fis = getContentResolver().openInputStream(uri);  
 Bitmap bitmap = BitmapFactory.*decodeStream*(fis);  
 } catch (Exception e) {  
  
 }  
}

可以看出，通过SAF并不能直接拿到图片的路径，图片的信息封装在Uri里，通过Uri构造出InputStream，再进行图片解码拿到Bitmap。

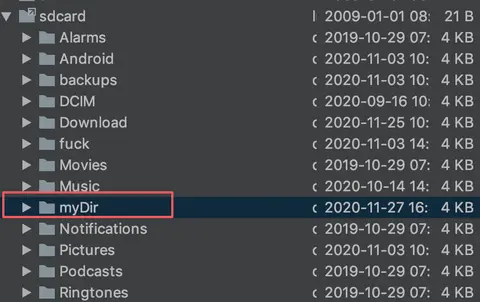
#### 访问其它目录

有两种方式：

##### 直接构造路径

在/sdcard/目录下直接创建目录：

private void testPublicFile() {  
 File rootFile = Environment.getExternalStorageDirectory();  
 String imagePath = rootFile.getAbsolutePath() + File.*separator* + "myDir";  
 File myDir = new File(imagePath);  
 if (!myDir.exists()) {  
 myDir.mkdir();  
 }  
}



可以看出，/sdcard/myDir/目录创建成功。

##### 2、通过SAF访问

与共享存储空间SAF访问方式一致。

### Android 10.0 之前访问方式总结

#### 为什么要变更

由上面分析的共享存储空间/其它目录访问方式可知，访问目录/文件可通过如下两个方法：

1、通过路径访问。路径可以直接构造也可以通过MediaStore获取。

2、通过Uri访问。Uri可以通过MediaStore或者SAF获取。

Android 6.0 以下访问共享存储空间/其它目录步骤：

1、AndroidManifest.xml里声明存储权限

2、通过路径或者Uri访问文件

Android 6.0(含)~Android 10.0(不含)访问共享存储空间/其它目录步骤：

1、AndroidManifest.xml里声明存储权限

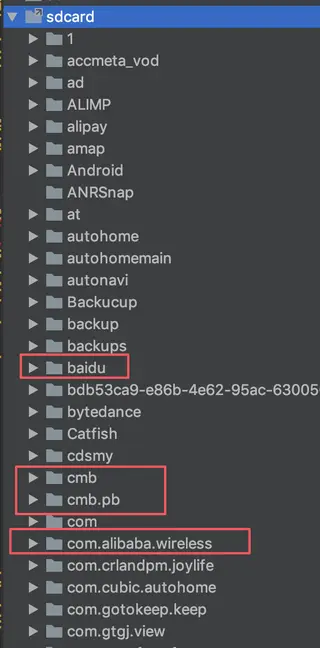
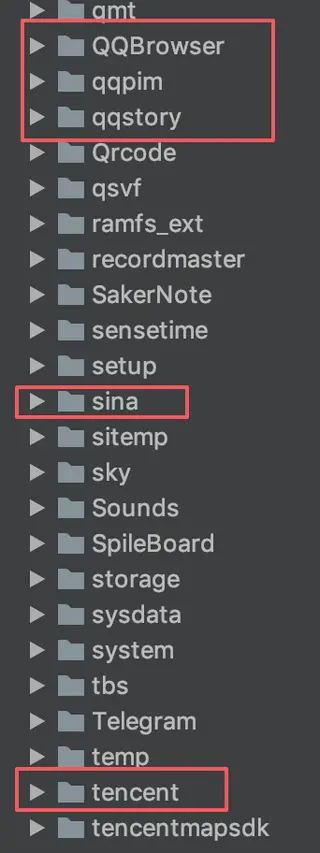
2、动态申请存储权限

3、通过路径或者Uri访问文件

3、Android 10.0 访问方式变更

为什么要变更

你可能已经发现了上面访问方式的弊端，比如我们能够直接在/sdcard/目录下创建目录/文件。事实上，很多App就是这么干的，看图说话：

可以看出/sdcard/目录下，如淘宝、qq、qq浏览器、微博、支付宝等都自己建了目录。

这么看来，导致目录结构很乱，而且App卸载后，对应的目录并没有删除，于是就是遗留了很多"垃圾"文件，久而久之不处理，用户的存储空间越来越小。

总结弊端如下：

1、在设置里"Clear storage"或者"Clear cache"并不能删除该目录下的文件

2、卸载App也不能删除该目录下的文件

3、App可以随意修改其它目录下的文件，如修改别的App创建的文件等，不安全

你也许会问，为什么要在/sdcard/目录下新建自己的目录呢？

大体有以下两个原因：

1、此处新建的目录不会被设置里的App存储用量统计，让用户"看起来"自己的App占用的存储空间很小

2、方便操作文件

#### 如何变更

面对众多App不讲"码德"随意新建目录/文件的现象，Google在Android 10.0上重拳出击了。

#### 引入Scoped Storage

翻译成中文有好几个版本：作用域存储、分区存储、沙盒存储。

具体中文翻译不重要，下面以分区存储指代。

分区存储原理：

1、App访问自身内部存储空间、访问外部存储空间-App私有目录不需要任何权限(这个与Android 10.0之前一致)

2、外部存储空间-共享存储空间、外部存储空间-其它目录 App无法通过路径直接访问，不能新建、删除、修改目录/文件等

3、外部存储空间-共享存储空间、外部存储空间-其它目录 需要通过Uri访问

分区存储的变更在于第二点、第三点。

#### 为什么Uri能够访问

先来看为什么通过路径无法直接访问。

我们知道访问文件最终是通过构造InputStream/OutputStream来实现的，以InputStream为例，看看其构造方法：

#FileInputStream.java  
//文件描述符  
private final FileDescriptor fd;  
public FileInputStream(File file) throws FileNotFoundException {  
 String name = (file != null ? file.getPath() : null);  
 ...  
 //传入name，构造FileDescriptor  
 //没有权限访问，则此处抛出异常  
 fd = IoBridge.open(name, O\_RDONLY);  
 ...  
}

可以看出，要想FileInputStream 能读入文件，核心是需要构造FileDescriptor，而对于Android 10.0，直接通过路径构造FileDescriptor 会抛出异常。

那么我们自然会想到，有没有通过构造好的FileDescriptor 来生成FileInputStream对象，进而使用read(xx)方法读取数据。

还真有，请看：通过Uri构造InputStream。

InputStream fis = getContentResolver().openInputStream(uri);

进入看其源码：

#ContentResolver.java  
public final @Nullable  
 InputStream openInputStream(@NonNull Uri uri)  
 throws FileNotFoundException {  
 ...  
 if (SCHEME\_ANDROID\_RESOURCE.equals(scheme)) {  
 ...  
 } else if (SCHEME\_FILE.equals(scheme)) {  
 ...  
 } else {  
 //通过Uri构造fd是被允许的  
 AssetFileDescriptor fd = openAssetFileDescriptor(uri, "r", null);  
 try {  
 //反过来创建InputStream  
 return fd != null ? fd.createInputStream() : null;  
 } catch (IOException e) {  
 throw new FileNotFoundException("Unable to create stream");  
 }  
 }  
}

AssetFileDescriptor 持有ParcelFileDescriptor 引用，而ParcelFileDescriptor 持有FileDescriptor 引用。

同理也适用于FileOutputStream。因此，通过Uri能够访问文件。

### 如何不适配Android 10.0

从以上分析可知，适配Android 10.0 有点麻烦，问题来了有没有简单的方法绕过检测。

第一种方法

1、Android 10.0 及其以后才会有分区存储功能，只要Android 设备不升级系统到Android 10.0以后，就不会有问题。

2、可能觉得这是句废话，其实不然，有些定制的设备系统一般都不会升级的。

如果不能使用第一种方法，还可以采用第二种方法。

第二种方法

1、Android 一般升级功能的时候都会配合targetSdkVersion使用。只要targetSdkVersion<=28，分区存储功能就不会开启。

有关targetSdkVersion 作用请移步：targetSdkVersion、compileSdkVersion、minSdkVersion作用与区别

如果第二种方法也不能使用，则还有第三种方法。

第三种方法

在AndroidManifest.xml 里application标签下添加：

android:requestLegacyExternalStorage="true" 可禁用分区存储

从长远的角度看，以上三个方法都不是一劳永逸的方法，其中第二种、第三种方法是Google 留给App开发者适配的缓冲时间。

对于第二种方法：

Google 在App上架App Store 时候可能会强制要求升级targetSdkVersion，因此该方法不保险。

对于第三种方法：

在Android 11会忽略该字段，强制开启分区存储，该字段也不怎么靠谱。

因此，最终还是需要老老实实按照Google 的要求适配Android 10.0，下篇将重点分析Android 10.0/11 该如何来适配。

# Android-10、11-存储完全适配(下)

## MediaStore 基本知识

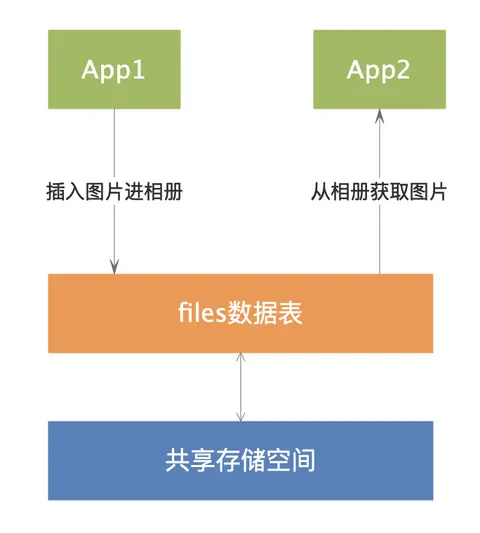
Android 10.0 存储访问方式变更地方在于：

自带外部存储-共享存储空间和自带外部存储-其它目录

以上两个地方不能通过路径直接访问文件，而是需要通过Uri访问。

### 共享存储空间

共享存储空间存放的是图片、视频、音频等文件，这些资源是公用的，所有App都能够访问它们。



系统里有external.db数据库，该数据库里有files表，该表里存放着共享文件的诸多信息，如图片有宽高，经纬度、存放路径等，视频宽高、时长、存放路径等。而文件真正存放的地方在于共享存储空间。

#### 保存图片到相册

当App1保存图片到相册时，简单流程如下：

1、将路径信息写入数据库里，并获取Uri

2、通过Uri构造输出流

3、将该图片保存在/sdcard/Pictures/目录下

#### 从相册获取图片

当App2从相册获取图片时，简单流程如下：

1、先查询数据库，找到对应的图片Cursor

2、从Cursor里构造Uri

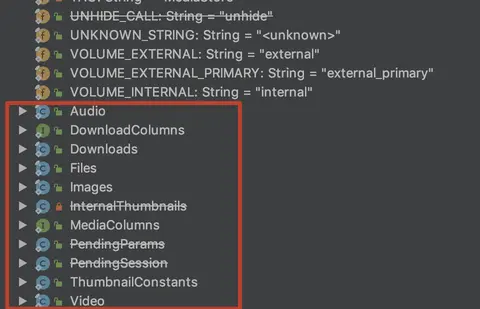
3、从Uri构造输入流读取图片

以上以图片为例简单分析了共享存储空间文件的写入与读取，实际上对于视频、音频步骤亦是如此。

### MediaStore作用

共享存储空间里存放着图片、视频、音频、下载的文件，App获取或者插入文件的时候怎么区分这些类型呢？

这个时候就需要MediaStore，来看看MediaStore.java

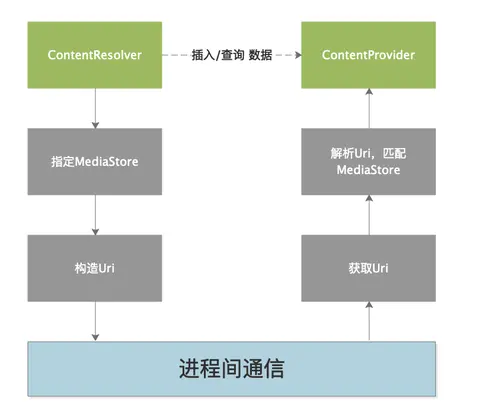


可以看出其内部有Audio、Images等内部类，这些内部类里记录着files表的各个字段名，通过构造这些参数就可以插入相应的字段值以及获取对应的字段值。

MediaStore 实际上就是相当于给各个字段起了别名，我们编码的时候更容易记住与使用：

//列举一些字段：  
//图片类型  
MediaStore.Images.Media.MIME\_TYPE  
//音频时长  
MediaStore.Audio.Media.DURATION  
//视频时长  
MediaStore.Video.Media.DURATION  
//等等，还有很多

### MediaStore和Uri联系



比如想要查询共享存储空间里的图片文件：

Cursor cursor = contentResolver.query(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, null, null, null, null);

MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI 意思是指定查询文件的类型是图片，并构造成Uri对象，Uri实现了Parcelable，能够在进程间传递。

接收方(另一个进程收到后)，匹配Uri，解析出对应的字段，进行具体的操作。

当然，MediaStore是系统提供的方便操作共享存储空间的类，若是自己写ContentProvider，则也可以自定义类似MediaStore的类用来标记自己的数据库表的字段。

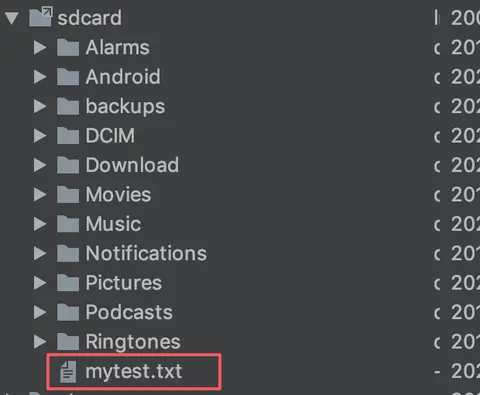
## 通过Uri读取和写入文件

既然不能通过路径直接访问文件，那么来看看如何通过Uri访问文件。在上篇文章里提到过：Uri可以通过MediaStore或者SAF获取。(此处需要注意的是：虽然也可以通过文件路径直接构造Uri，但是此种方式构造的Uri是没有权限访问文件的)

先来看看通过SAF获取Uri。

### 从Uri读取文件

现在/sdcard/目录下存在一个文件名为：mytest.txt。



传统的直接读取mytest.txt方法：

//从文件读取  
private void readFile(String filePath) {  
 if (TextUtils.isEmpty(filePath))  
 return;  
  
 try {  
 File file = new File(filePath);  
 FileInputStream fileInputStream = new FileInputStream(file);  
 BufferedInputStream bis = new BufferedInputStream(fileInputStream);  
 byte[] readContent = new byte[1024];  
 int readLen = 0;  
 while (readLen != -1) {  
 readLen = bis.read(readContent, 0, readContent.length);  
 if (readLen > 0) {  
 String content = new String(readContent);  
 Log.*d*("test", "read content:" + content.substring(0, readLen));  
 }  
 }  
 fileInputStream.close();  
 } catch (Exception e) {  
  
 }  
}

开启分区存储功能后，这种方法是不可取的，会报权限错误。

而mytest.txt不属于共享存储空间的文件，是属于其它目录的，因此不能通过MediaStore获取，只能通过SAF获取，如下：

private void startSAF() {  
 Intent intent = new Intent(Intent.*ACTION\_OPEN\_DOCUMENT*);  
 intent.addCategory(Intent.*CATEGORY\_OPENABLE*);  
 //指定选择文本类型的文件  
 intent.setType("text/plain");  
 startActivityForResult(intent, 100);  
}  
  
@Override  
protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {  
 super.onActivityResult(requestCode, resultCode, data);  
  
 if (requestCode == 100) {  
 //选中返回的文件信息封装在Uri里  
 Uri uri = data.getData();  
 openUriForRead(uri);  
 }  
}

拿到Uri后，用来构造输入流读取文件。

private void openUriForRead(Uri uri) {  
 if (uri == null)  
 return;  
  
 try {  
 //获取输入流  
 InputStream inputStream = getContentResolver().openInputStream(uri);  
 byte[] readContent = new byte[1024];  
 int len = 0;  
 do {  
 //读文件  
 len = inputStream.read(readContent);  
 if (len != -1) {  
 Log.*d*("test", "read content:" + new String(readContent).substring(0, len));  
 }  
 } while (len != -1);  
 inputStream.close();  
 } catch (Exception e) {  
 Log.*d*("test", e.getLocalizedMessage());  
 }  
}

最终输出：

C:\Users\Administrator\AppData\Roaming\Tencent\Users\282330332\QQ\WinTemp\RichOle\OU@3A2C}8%2PD`AIOIO4AMW.png

由此可以看出，mytest.txt属于"其它目录"下的文件，因此需要通过SAF访问，SAF返回Uri，通过Uri构造InputStream即可读取文件。

### 从Uri写入文件

继续来看看写的过程，现在需要往mytest.txt写入内容。

同样的，还是需要通过SAF拿到Uri，拿到Uri后构造输出流：

private void openUriForWrite(Uri uri) {  
 if (uri == null) {  
 return;  
 }  
  
 try {  
 //从uri构造输出流  
 OutputStream outputStream = getContentResolver().openOutputStream(uri);  
 //待写入的内容  
 String content = "hello world I'm from SAF\n";  
 //写入文件  
 outputStream.write(content.getBytes());  
 outputStream.flush();  
 outputStream.close();  
 } catch (Exception e) {  
 Log.*d*("test", e.getLocalizedMessage());  
 }  
}

最后来看看文件是否写入成功，通过SAF再次读取mytest.txt，发现正好是之前写入的内容，说明写入成功。

## 通过Uri 获取图片和插入相册

上面列举出了其它目录下文件的读写，方法是通过SAF拿到Uri。

SAF好处是：

系统提供了文件选择器，调用者只需要指定想要读写的文件类型，比如文本类型、图片类型、视频类型等，选择器就会过滤出相应文件以供选择。接入方便，选择简单。

想想另一种场景：

想要自己实现相册选择器，那么就需要获得共享存储空间下的文件信息。此种场景下使用SAF是无法做到的。

因此问题的关键是：如何批量获得共享存储空间下图片/视频的信息？

答案是：ContentResolver+ContentProvider+MediaStore(ContentProvider对于调用者是透明的)。

以图片为例，分析插入与查询方式。

### 插入相册

来看看图片的插入过程：

//fileName为需要保存到相册的图片名  
private void insert2Album(InputStream inputStream, String fileName) {  
 if (inputStream == null)  
 return;  
  
 ContentValues contentValues = new ContentValues();  
 contentValues.put(MediaStore.Images.ImageColumns.DISPLAY\_NAME, fileName);  
 if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.Q) {  
 //RELATIVE\_PATH 字段表示相对路径-------->(1)  
 contentValues.put(MediaStore.Images.ImageColumns.RELATIVE\_PATH, Environment.DIRECTORY\_PICTURES);  
 } else {  
 String dstPath = Environment.getExternalStorageDirectory() + File.*separator* + Environment.DIRECTORY\_PICTURES  
 + File.*separator* + fileName;  
 //DATA字段在Android 10.0 之后已经废弃  
 contentValues.put(MediaStore.Images.ImageColumns.DATA, dstPath);  
 }  
  
 //插入相册------->(2)  
 Uri uri = getContentResolver().insert(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, contentValues);  
  
 //写入文件------->(3)  
 write2File(uri, inputStream);  
}

重点说明三个点：

(1)

Android 10.0之前，MediaStore.Images.ImageColumns.DATA 字段记录的是图片的绝对路径，而Android 10.0(含)之后，DATA 被废弃，取而代之的是使用MediaStore.Images.ImageColumns.RELATIVE\_PATH，表示相对路径。比如指定RELATIVE\_PATH为Environment.DIRECTORY\_PICTURES，表示之后的图片将会放到Environment.DIRECTORY\_PICTURES目录下。

(2)

调用ContentResolver里的方法插入相册。

MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI 指的是插入图片表。

ContentValues 以Map的形式记录了待写入的字段值。

插入后返回Uri。

(3)

以上两步仅仅只是往数据库里增加一条记录，该记录指向的新文件是空的，需要将图片写入到新文件。

而新文件位于/sdcard/Pictures/目录下，该目录是不能直接通过路径访问的，因此需要通过第二步返回的Uri进行访问。

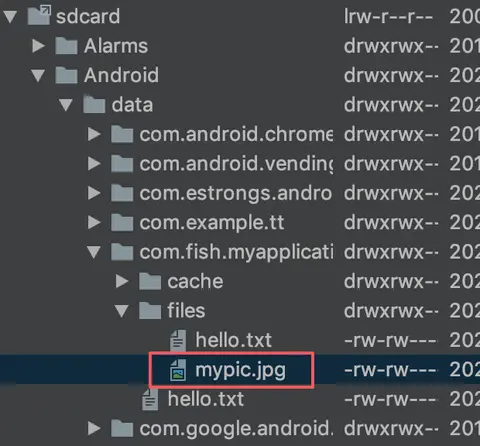
//uri 关联着待写入的文件  
//inputStream 表示原始的文件流  
private void write2File(Uri uri, InputStream inputStream) {  
 if (uri == null || inputStream == null)  
 return;  
  
 try {  
 //从Uri构造输出流  
 OutputStream outputStream = getContentResolver().openOutputStream(uri);  
  
 byte[] in = new byte[1024];  
 int len = 0;  
  
 do {  
 //从输入流里读取数据  
 len = inputStream.read(in);  
 if (len != -1) {  
 outputStream.write(in, 0, len);  
 outputStream.flush();  
 }  
 } while (len != -1);  
  
 inputStream.close();  
 outputStream.close();  
  
 } catch (Exception e) {  
 Log.*d*("test", e.getLocalizedMessage());  
 }  
}

可以看出，目标文件关联的Uri有了，还需要原始的输入文件。

测试上述的插入方法：

private void testInsert() {  
  
 String picName = "mypic.jpg";  
 try {  
 File externalFilesDir = getExternalFilesDir(null);  
 File file = new File(externalFilesDir, picName);  
 FileInputStream fis = new FileInputStream(file);  
 insert2Album(fis, picName);  
 } catch (Exception e) {  
 Log.*d*("test", e.getLocalizedMessage());  
 }  
}

其中，原始文件(图片)存放于自带外部存储-App私有目录，如下：



需要注意的是：

1、读取原始文件需要权限，上述例子里的原始文件存放在自带外部存储-App私有目录，因此本App可以使用路径直接读取

2、对于其他目录则依然需要构造Uri读取，如通过SAF获取Uri

### 获取图片

同样的，想要从系统相册中获取图片，也需要通过Uri访问。

private void queryImageFromAlbum() {  
 Cursor cursor = getContentResolver().query(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, null,  
 null, null, null);  
  
 if (cursor != null) {  
 while (cursor.moveToNext()) {  
 //获取唯一的id  
 long id = cursor.getLong(cursor.getColumnIndexOrThrow(MediaStore.MediaColumns.\_ID));  
 //通过id构造Uri  
 Uri uri = ContentUris.withAppendedId(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, id);  
 //解析uri  
 decodeUriForBitmap(uri);  
 }  
 }  
}  
  
private void decodeUriForBitmap(Uri uri) {  
 if (uri == null)  
 return;  
  
 try {  
 //构造输入流  
 InputStream inputStream = getContentResolver().openInputStream(uri);  
 //解析Bitmap  
 Bitmap bitmap = BitmapFactory.*decodeStream*(inputStream);  
 if (bitmap != null)  
 Log.*d*("test", "bitmap width-width:" + bitmap.getWidth() + "-" + bitmap.getHeight());  
 } catch (Exception e) {  
 Log.*d*("test", e.getLocalizedMessage());  
 }  
}

与插入相册过程类似，同样需要拿到Uri，再构造输入流，从输入流读取文件(图片内容)。

以上，通过Uri 获取图片和插入相册分析完毕，共享存储空间的其他文件类型如视频、音频、下载文件也是同样的流程。

需要说明的是上述的ContentResolver .insert(xx)/ContentResolver.query(xx) 的参数取值还可以更丰富，但不是本篇重点，因此忽略了，实际使用过程中具体情况具体分析。

## Android 11.0 权限申请

通过Uri访问文件似乎已经满足了Android 10.0适配要求，但是仔细想想还是有不足之处：

1、共享存储空间只能通过MediaStore访问，以前流行的访问方式是直接通过路径访问。比如自己做的相册管理器，先遍历相册拿到图片/视频的路径，然后再解析成Bitmap展示，现在需要先拿到Uri，再解析成Bitmap，多少有些不方便。此外，也许你依赖的第三方库是直接通过路径访问文件的，而三方库又没有及时更新适配分区存储，可能就会导致用不了相应的功能。

2、SAF虽然能够访问其它目录的文件，但是每次都需要跳转到新的页面去选择，当想要批量展示文件的时候，比如自己做的文件管理器，就需要列出当前目录下有哪些目录/文件，这个时候需要有权限遍历/sdcard/目录。显然，SAF并不能胜任此工作。

Android 11.0考虑到上面的问题，因此做了新的优化。

### 共享存储空间-媒体文件访问变更

媒体文件可以通过路径直接访问：

private void getImagePath(Context context) {  
 ContentResolver contentResolver = context.getContentResolver();  
 Cursor cursor = contentResolver.query(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, null, null, null, null);  
 while (cursor.moveToNext()) {  
  
 try {  
 //取出路径  
 String path = cursor.getString(cursor.getColumnIndex(MediaStore.Images.ImageColumns.DATA));  
 Bitmap bitmap = BitmapFactory.*decodeFile*(path);  
 } catch (Exception e) {  
 Log.*d*("test", e.getLocalizedMessage());  
 }  
 break;  
 }  
}

可以看出，之前在Android 10.0上被禁用的访问方式，在Android 11.0上又被允许了，这就解决了上面的第一个问题。

需要注意的是：此种方式只允许读文件，写文件依然不行

Google 官方指导意见是：

虽然可以通过路径直接访问媒体文件，但是这些操作最终是被重定向到MediaStore API的，重定向过程可能会损耗一些性能，并且直接通过路径访问不一定比MediaStore API 访问快。

总之建议非必要的话不要直接使用路径访问。

### 访问所有文件

假若App开启了分区存储功能，当App运行在Android 10.0的设备上时，是没法遍历/sdcard/目录的。而在Android 11.0上运行时是可以遍历的，需要进行如下几个步骤。

#### 声明管理权限

在AndroidManifest.xml添加权限声明

<uses-permission android:name="android.permission.MANAGE\_EXTERNAL\_STORAGE" />

#### 动态申请所有文件访问权限

private void testAllFiles() {  
 //运行设备>=Android 11.0  
 if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.R) {  
 //检查是否已经有权限  
 if (!Environment.isExternalStorageManager()) {  
 //跳转新页面申请权限  
 startActivityForResult(new Intent(Settings.ACTION\_MANAGE\_ALL\_FILES\_ACCESS\_PERMISSION), 101);  
 }  
 }  
}  
  
@Override  
protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {  
 super.onActivityResult(requestCode, resultCode, data);  
 //申请权限结果  
 if (requestCode == 101) {  
 if (Environment.isExternalStorageManager()) {  
 Toast.makeText(MainActivity.this, "访问所有文件权限申请成功", Toast.LENGTH\_SHORT).show();  
  
 //遍历目录  
 showAllFiles();  
 }  
 }  
}

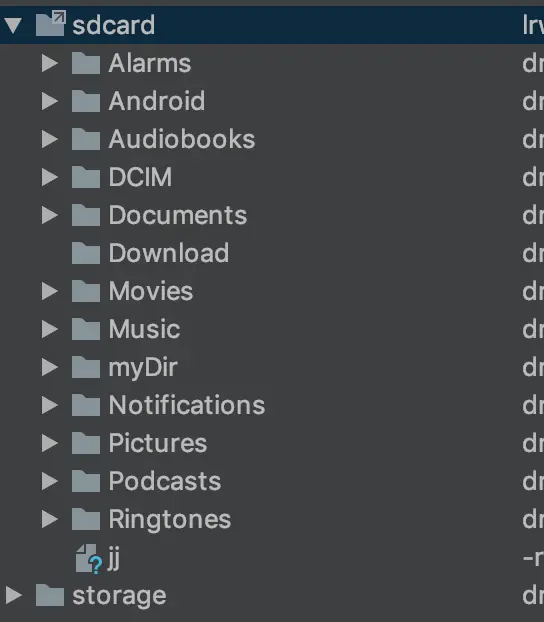
此处申请权限不是以对话框的形式提示用户，而是跳转到新的页面，说明该权限的管理更严格。

#### 遍历目录、读写文件

拥有权限后，就可以进行相应的操作了。

private void showAllFiles() {  
 File file = Environment.getExternalStorageDirectory();  
 File[] list = file.listFiles();  
 for (int i = 0; i < list.length; i++) {  
 String name = list[i].getName();  
 Log.*d*("test", "fileName:" + name);  
 }  
}

文件管理器效果图类似如下：



当然读写文件也不在话下了，比如往/sdcard/目录下写入文件：

private void testPublicFile() {  
 File rootFile = Environment.getExternalStorageDirectory();  
 try {  
 File file = new File(rootFile, "mytest.txt");  
 FileOutputStream fos = new FileOutputStream(file);  
 String content = "hello world\n";  
 fos.write(content.getBytes());  
 fos.flush();  
 fos.close();  
 } catch (Exception e) {  
 Log.*d*("test", e.getLocalizedMessage());  
 }  
}

ACTION\_MANAGE\_ALL\_FILES\_ACCESS\_PERMISSION 这个权限的名字看起来很唬人，感觉就像是能够操作所有文件的样子，这不就是打破了分区存储的规则了吗？其实不然：

即使拥有了该权限，依然不能访问内部存储和外部存储-App私有目录

需要说明的是：

1、Environment.isExternalStorageManager()、Build.VERSION\_CODES.R 等需要编译版本>=30才能编译通过。

2、Google 提示当使用MANAGE\_EXTERNAL\_STORAGE 申请权限时，并且targetSdkVersion>=30，此种情况下App被禁止上架Google Play的，限制时间最早到2021年。因此，在此时间之前若是申请了MANAGE\_EXTERNAL\_STORAGE权限，最好不要升级targetSdkVersion到30以上。

## 5、Android 10/11 存储适配建议

好了，通过分析Android 10/11存储适配方式，了解到了不同的系统需要如何进行适配，此时就需要一个统一的适配方案了。

### 适配核心

分区存储是核心，App自身产生的文件应该存放在自己的目录下：

/sdcard/Android/data/packagename/ 和/data/data/packagename/

这两个目录本App无需申请访问权限即可申请，其它App无法访问本App的目录。

### 适配共享存储

共享存储空间里的文件需要通过Uri构造输入输出流访问，Uri获取方式有两种：MediaStore和SAF。

### 适配其它目录

在Android 11上需要申请访问所有文件的权限。

具体做法

#### 第一步

在AndroidManifest.xml里添加如下字段：

权限声明：

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

<uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE" />

<uses-permission android:name="android.permission.MANAGE\_EXTERNAL\_STORAGE" />

在<application/>标签下添加如下字段：

android:requestLegacyExternalStorage="true"

#### 第二步

如果需要访问共享存储空间，则判断运行设备版本是否大于等于Android6.0，若是则需要申请WRITE\_EXTERNAL\_STORAGE 权限。拿到权限后，通过Uri访问共享存储空间里的文件。

如果需要访问其它目录，则通过SAF访问

#### 第三步

如果想要做文件管理器、病毒扫描管理器等功能。则判断运行设备版本是否大于等于Android 6.0，若是先需要申请普通的存储权。若运行设备版本为Android 10.0，则可以直接通过路径访问/sdcard/目录下文件(因为禁用了分区存储)；若运行设备版本为Android 11.0，则需要申请MANAGE\_EXTERNAL\_STORAGE 权限。

以上是Android 存储权限适配的全部内容。

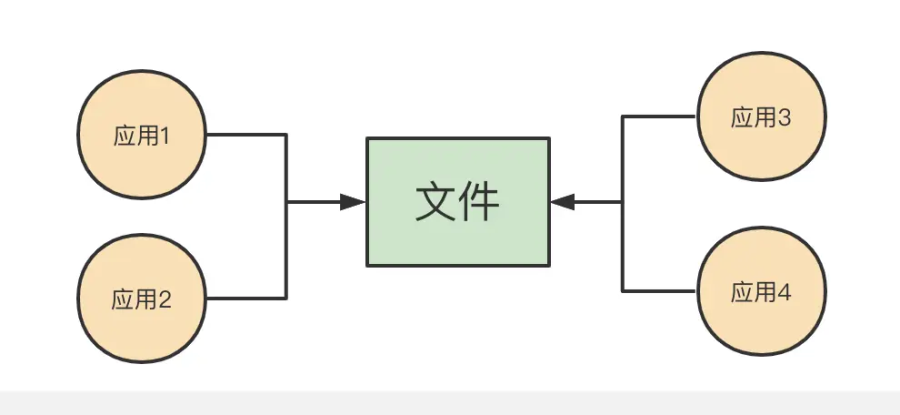
# Android-FileProvider-轻松掌握

本次，将着重分析Android 应用之间如何使用第三方应用打开文件，如何分享文件给第三方应用。

## Android 应用间共享文件

### 共享基础

提到文件共享，首先想到就是在本地磁盘上存放一个文件，多个应用都可以访问它，如下：



理想状态下只要知道了文件的存放路径，那么各个应用都可以读写它。

比如相册里的图片存放目录：/sdcard/DCIM/、/sdcard/Pictures/ 。

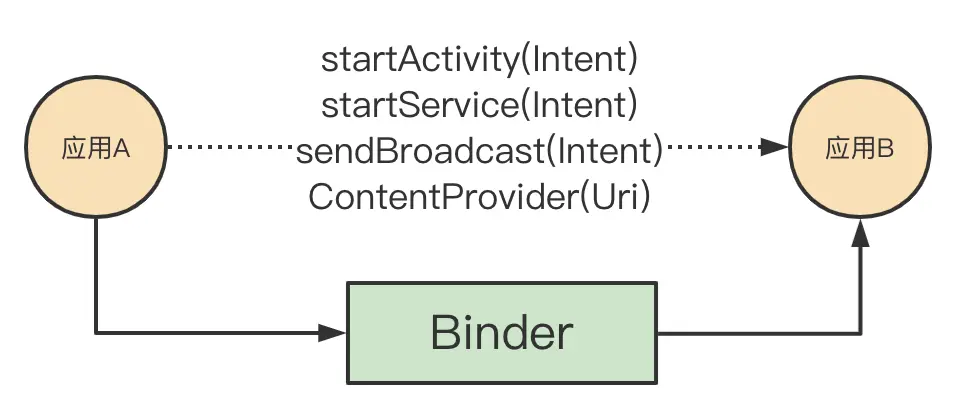
再比如相册里的视频存放目录：/sdcard/DCIM/、/sdcard/Movies/。

### 共享方式

一个常见的应用场景：

应用A里检索到一个文件my.txt，它无法打开，于是想借助其它应用打开，这个时候它需要把待打开的文件路径告诉其它应用。

假设应用B可以打开my.txt，那么应用A如何把路径传递给应用B呢，这就涉及到了进程间通信。我们知道Android进程间通信主要手段是Binder，而四大组件的通信也是依靠Binder，因此我们应用间传递路径可以依靠四大组件。



可以看出，Activity/Service/Broadcast 可以传递Intent，而ContentProvider传递Uri，实际上Intent 里携带了Uri变量，因此四大组件之间可以传递Uri，而路径就可以存放在Uri里。

## FileProvider 应用与原理

以使用其它应用打开文件为例，分别阐述Android 7.0 前后的不同点。

### Android 7.0 之前使用

上面说到了传递路径可以通过Uri，来看看如何使用：

private void openByOtherForN() {  
 Intent intent = new Intent();  
 //指定Action，使用其它应用打开  
 intent.setAction(Intent.*ACTION\_VIEW*);  
 //通过路径，构造Uri  
 Uri uri = Uri.fromFile(new File(external\_filePath));  
 //设置Intent，附带Uri  
 intent.setData(uri);  
 //跨进程传递Intent  
 startActivity(intent);  
}

其中

external\_filePath="/storage/emulated/0/fish/myTxt.txt"

构造为uri 后uriString="file:///storage/emulated/0/fish/myTxt.txt"

可以看出，文件路径前多了"file:///"字符串。

而接收方在收到Intent后，拿出Uri，通过：

filePath = uri.getEncodedPath() 拿到发送方发送的原始路径后，即可读写文件。

然而此种构造Uri方式在Android7.0(含)之后被禁止了，若是使用则抛出异常：

C:\Users\Administrator\AppData\Roaming\Tencent\Users\282330332\QQ\WinTemp\RichOle\(JM(CX$BHRHWL4GE3M2C6NU.png

可以看出，Uri.fromFile 构造方式的缺点：

1、发送方传递的文件路径接收方完全知晓，一目了然，没有安全保障。

2、发送方传递的文件路径接收方可能没有读取权限，导致接收异常。

### Android 7.0(含)之后的使用

先想想，若是我们自己操刀，如何规避以上两个问题呢？

针对第一个问题：

可以将具体路径替换为另一个字符串，类似以前密码本的感觉，比如：

"/storage/emulated/0/fish/myTxt.txt" 替换为"myfile/Txt.txt"，这样接收方收到文件路径完全不知道原始文件路径是咋样的。

不过这也引入了另一个额外的问题：接收方不知道真实路径，如何读取文件呢?

针对第二个问题

既然不确定接收方是否有打开文件权限，那么是否由发送方打开，然后将流传递给接收方就可以了呢？

Android 7.0(含)之后引入了FileProvider，可以解决上述两个问题。

### FileProvider 应用

先来看看如何使用FileProvider 来传递路径。

细分为四个步骤：

#### 定义FileProvider 子类

public class MyFileProvider extends FileProvider {  
  
}

定义一个空的类，继承自FileProvider，而FileProvider 继承自ContentProvider。

注：FileProvider 需要引入AndroidX

#### AndroidManifest 里声明FileProvider

既然是ContentProvider，那么需要像Activity一样在AndroidManifest.xml里声明：

<provider  
 android:authorities="com.fish.fileprovider"  
 android:name=".fileprovider.MyFileProvider"  
 android:exported="false"  
 android:grantUriPermissions="true">  
 <meta-data  
 android:name="android.support.FILE\_PROVIDER\_PATHS"  
 android:resource="@xml/file\_path">  
 </meta-data>  
</provider>

字段解释如下：

1、android:authorities 标识ContentProvider的唯一性，可以自己任意定义，最好是全局唯一的。

2、android:name 是指之前定义的FileProvider 子类。

3、android:exported="false" 限制其他应用获取Provider。

4、android:grantUriPermissions="true" 授予其它应用访问Uri权限。

5、meta-data 囊括了别名应用表。

5.1、android:name 这个值是固定的，表示要解析file\_path。

5.2、android:resource 自己定义实现的映射表

#### 路径映射表

可以看出，FileProvider需要读取映射表。

在/res/ 下建立xml 文件夹，然后再创建对应的映射表(xml)，最终路径如下：/res/xml/file\_path.xml。

内容如下：

<?xml version="1.0" encoding="utf-8"?>  
<paths>  
 <root-path name="myroot" path="." />  
 <external-path name="external\_file" path="fish" />  
 <external-files-path name="external\_app\_file" path="myfile" />  
 <external-cache-path name="external\_app\_cache" path="mycache/doc/" />  
 <files-path name="inner\_app\_file" path="." />  
 <cache-path name="inner\_app\_cache" path="." />  
</paths>

字段解释如下：

1、root-path 标签表示要给根目录下的子目录取别名(包括内部存储、自带外部存储、扩展外部存储，统称用"/"表示)，path 属性表示需要被更改的目录名，其值为:"."，表示不区分目录，name 属性表示将path 目录更改后的别名。

2、假若有个文件路径：/storage/emulated/0/fish/myTxt.txt，而我们只配置了root-path 标签，那么最终该文件路径被替换为：/myroot/storage/emulated/0/fish/myTxt.txt。

可以看出，因为path="."，因此任何目录前都被追加了myroot。

剩下的external-path等标签对应的目录如下：

1、external-path--->Environment.getExternalStorageDirectory()，如

/storage/emulated/0/fish

2、external-files-path--->ContextCompat.getExternalFilesDirs(context, null)。

3、external-cache-path--->ContextCompat.getExternalCacheDirs(context)。

4、files-path--->context.getFilesDir()。

5、cache-path--->context.getCacheDir()。

你可能已经发现了，这些标签所代表的目录有重叠的部分，在替换别名的时候如何选择呢？答案是：选择最长匹配的。

假设我们映射表里只定义了root-path与external-path，分别对应的目录为：

root-path--->/

external-path--->/storage/emulated/0/

现在要传递的文件路径为：/storage/emulated/0/fish/myTxt.txt。需要给这个文件所在目录取别名，因此会遍历映射表找到最长匹配该目录的标签，显然external-path 所表示的/storage/emulated/0/ 与文件目录最为匹配，因此最后文件路径被替换为：/external\_file/myTxt.txt

#### 使用FileProvider 构造路径

映射表建立好之后，接着就需要构造路径。

private void openByOther() {  
 //取得文件扩展名  
 String extension = external\_filePath.substring(external\_filePath.lastIndexOf(".") + 1);  
 //通过扩展名找到mimeType  
 String mimeType = MimeTypeMap.getSingleton().getMimeTypeFromExtension(extension);  
 //构造Intent  
 Intent intent = new Intent();  
 //赋予读写权限  
 intent.addFlags(Intent.*FLAG\_GRANT\_READ\_URI\_PERMISSION* | Intent.*FLAG\_GRANT\_WRITE\_URI\_PERMISSION*);  
 //表示用其它应用打开  
 intent.setAction(Intent.*ACTION\_VIEW*);  
 File file = new File(external\_filePath);  
 //第二个参数表示要用哪个ContentProvider，这个唯一值在AndroidManifest.xml里定义了  
 //若是没有定义MyFileProvider，可直接使用FileProvider替代  
 Uri uri = MyFileProvider.getUriForFile(this, "com.fish.fileprovider", file);  
 //给Intent 赋值  
 intent.setDataAndType(uri, mimeType);  
 try {  
 //交由系统处理  
 startActivity(intent);  
 } catch (Exception e) {  
 //若是没有其它应用能够接收打开此种mimeType，则抛出异常  
 Toast.makeText(this, e.getLocalizedMessage(),Toast.*LENGTH\_SHORT*).show();  
 }  
}

/storage/emulated/0/fish/myTxt.txt 最终构造为：

content://com.fish.fileprovider/external\_file/myTxt.txt

对于私有目录：/data/user/0/com.example.androiddemo/files/myTxt.txt 最终构造为：

content://com.fish.fileprovider/inner\_app\_file/myTxt.txt

可以看出添加了：

content 作为scheme；

com.fish.fileprovider 即为我们定义的 authorities，作为host；

如此构造后，第三方应用收到此Uri后，并不能从路径看出我们传递的真实路径，这就解决了第一个问题：

发送方传递的文件路径接收方完全知晓，一目了然，没有安全保障。

## FileProvider Uri构造与解析

### Uri 构造输入流

发送方将Uri交给系统，系统找到有能力处理该Uri的应用。发送方A需要别的应用打开myTxt.txt 文件，假设应用B具有能够打开文本文件的能力，并且也愿意接收别人传递过来的路径，那么它需要在AndroidManifest里做如下声明：

<activity android:name="com.fish.fileprovider.ReceiveActivity">  
 <intent-filter>  
 <action android:name="android.intent.action.VIEW"></action>  
 <category android:name="android.intent.category.DEFAULT"/>  
 <category android:name="android.intent.category.BROWSABLE"/>  
 <data android:scheme="content"/>  
 <data android:scheme="file"/>  
 <data android:scheme="http"/>  
 <data android:mimeType="text/\*"></data>  
 </intent-filter>  
</activity>

android.intent.action.VIEW 表示接收别的应用打开文件的请求。

android:mimeType 表示其具有打开某种文件的能力，text/\* 表示只接收文本类型的打开请求。

当声明了上述内容后，该应用就会出现在系统的选择弹框里，当用户点击弹框里的该应用时，ReceiveActivity 将会被调用。我们知道，传递过来的Uri被包装在Intent里，因此ReceiveActivity 需要处理Intent。

private void handleIntent() {  
 Intent intent = getIntent();  
 if (intent != null) {  
 if (intent.getAction().equals(Intent.*ACTION\_VIEW*)) {  
 //从Intent里获取uri  
 uri = intent.getData();  
 String content = handleUri(uri);  
 if (!TextUtils.isEmpty(content)) {  
 tvContent.setText("打开文件内容：" + content);  
 }  
 }  
 }  
}  
  
private String handleUri(Uri uri) {  
 if (uri == null)  
 return null;  
  
 String scheme = uri.getScheme();  
 if (!TextUtils.isEmpty(scheme)) {  
 if (scheme.equals("content")) {  
 try {  
 //从uri构造流  
 InputStream inputStream = getContentResolver().openInputStream(uri);  
 try {  
 //有流之后即可读取内容  
 byte[] content = new byte[inputStream.available()];  
 inputStream.read(content);  
 return new String(content);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 return null;  
}

从Intent里拿到Uri，再通过Uri构造输入流，最终从输入流里读取文件内容。

至此，应用A通过FileProvider可将其能够访问的任意路径的文件传递给应用B，应用B能够读取文件并展示。

看到这里，你可能已经发现了：还没有解决第二个问题呢：发送方传递的文件路径接收方可能没有读取权限，导致接收异常。

这就需要从getContentResolver().openInputStream(uri)说起：

#ContentResolver.java  
public final @Nullable InputStream openInputStream(@NonNull Uri uri)  
 throws FileNotFoundException {  
 Preconditions.checkNotNull(uri, "uri");  
 String scheme = uri.getScheme();  
 if (SCHEME\_ANDROID\_RESOURCE.equals(scheme)) {  
 ...  
 } else if (SCHEME\_FILE.equals(scheme)) {  
 //file开头  
 } else {  
 //content开头 走这  
 AssetFileDescriptor fd = openAssetFileDescriptor(uri, "r", null);  
 try {  
 //从文件描述符获取输入流  
 return fd != null ? fd.createInputStream() : null;  
 } catch (IOException e) {  
 throw new FileNotFoundException("Unable to create stream");  
 }  
 }  
}  
  
public final @Nullable AssetFileDescriptor openAssetFileDescriptor(@NonNull Uri uri,  
 @NonNull String mode, @Nullable CancellationSignal cancellationSignal)  
 throws FileNotFoundException {  
 ...  
  
 //根据scheme 区分不同的协议  
 String scheme = uri.getScheme();  
 if (SCHEME\_ANDROID\_RESOURCE.equals(scheme)) {  
 //资源文件  
 } else if (SCHEME\_FILE.equals(scheme)) {  
 //file 开头  
 } else {  
 //content 开头  
 if ("r".equals(mode)) {  
 return openTypedAssetFileDescriptor(uri, "\*/\*", null, cancellationSignal);  
 } else {  
 ...  
 }  
 }  
}  
  
public final @Nullable AssetFileDescriptor openTypedAssetFileDescriptor(@NonNull Uri uri,  
 @NonNull String mimeType, @Nullable Bundle opts,  
 @Nullable CancellationSignal cancellationSignal) throws FileNotFoundException {  
  
 ...  
 //找到FileProvider IPC 调用  
 IContentProvider unstableProvider = acquireUnstableProvider(uri);  
  
 try {  
 try {  
 //IPC 调用，返回文件描述符  
 fd = unstableProvider.openTypedAssetFile(  
 mPackageName, uri, mimeType, opts, remoteCancellationSignal);  
 if (fd == null) {  
 // The provider will be released by the finally{} clause  
 return null;  
 }  
 } catch (DeadObjectException e) {  
 ...  
 }  
 ...  
 //构造AssetFileDescriptor  
 return new AssetFileDescriptor(pfd, fd.getStartOffset(),  
 fd.getDeclaredLength());  
  
 } catch (RemoteException e) {  
 ...  
 }  
}

以上是应用B的调用流程，最终拿到应用A的FileProvider，拿到FileProvider 后即可进行IPC调用。

应用B发起了IPC，来看看应用A如何响应这动作的：

#ContentProviderNative.java  
//Binder调用此方法  
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)  
 throws RemoteException {  
 case OPEN\_TYPED\_ASSET\_FILE\_TRANSACTION:  
 {  
 ...  
 fd = openTypedAssetFile(callingPkg, url, mimeType, opts, signal);  
 }  
}  
  
 #ContentProvider.java  
@Override  
public AssetFileDescriptor openTypedAssetFile(String callingPkg, Uri uri, String mimeType,  
 Bundle opts, ICancellationSignal cancellationSignal) throws FileNotFoundException {  
 ...  
 try {  
 return mInterface.openTypedAssetFile(  
 uri, mimeType, opts, CancellationSignal.fromTransport(cancellationSignal));  
 } catch (RemoteException e) {  
 ...  
 } finally {  
 ...  
 }  
}  
  
public @Nullable AssetFileDescriptor openAssetFile(@NonNull Uri uri, @NonNull String mode)  
 throws FileNotFoundException {  
 ParcelFileDescriptor fd = openFile(uri, mode);  
 return fd != null ? new AssetFileDescriptor(fd, 0, -1) : null;  
}

可以看出，最后调用了openFile()方法，而FileProvider重写了该方法：

#ParcelFileDescriptor.java  
@Override  
public ParcelFileDescriptor openFile(@NonNull Uri uri, @NonNull String mode)  
 throws FileNotFoundException {  
 //解析uri，从里面拿出对应的路径  
 final File file = mStrategy.getFileForUri(uri);  
 final int fileMode = modeToMode(mode);  
 //构造ParcelFileDescriptor  
 return ParcelFileDescriptor.*open*(file, fileMode);  
}

ParcelFileDescriptor 持有FileDescriptor，可以跨进程传输。

重点是mStrategy.getFileForUri(uri)，如何通过Uri找到path，代码很简单，就不贴了，仅用图展示。

### Uri与Path互转

#### Path 转Uri

回到最初应用A如何将path构造为Uri：

应用A在启动的时候，会扫描AndroidManifest.xml 里的FileProvider，并读取映射表构造为一个Map:

image.png

这个Map的Key 为映射表里的别名，而Value对应需要替换的目录。

还是以/storage/emulated/0/fish/myTxt.txt 为例：

当调用MyFileProvider.getUriForFile(xx)时，遍历Map，找到最匹配条目，最匹配的即为external\_file。因此会用external\_file 代替/storage/emulated/0/fish/，最终形成的Uri为：content://com.fish.fileprovider/external\_file/myTxt.txt

#### Uri 转Path

构造了Uri传递给应用B，应用B又通过Uri构造输入流，构造输入流的过程由应用A完成，因此A需要将Uri转为Path：

A先将Uri分离出external\_file/myTxt.txt，然后通过external\_file 从Map里找到对应Value 为：/storage/emulated/0/fish/，最后将myTxt.txt拼接，形成的路径为：

/storage/emulated/0/fish/myTxt.txt

可以看出，Uri成功转为了Path。

现在来梳理整个流程：

1、应用A使用FileProvider通过Map(映射表)将Path转为Uri，通过IPC 传递给应用B。

2、应用B使用Uri通过IPC获取应用A的FileProvider。

3、应用A使用FileProvider通过映射表将Uri转为Path，并构造出文件描述符。

4、应用A将文件描述符返回给应用B，应用B就可以读取应用A发送的文件了。

image.png

由以上可知，不管应用B是否有存储权限，只要应用A有权限就行，因为对文件的访问都是通过应用A完成的，这就回答了第二个问题：发送方传递的文件路径接收方可能没有读取权限，导致接收异常。

以上以打开文件为例阐述了FileProvider的应用，实际上分享文件也是类似的过程。

当然，从上面可以看出FileProvider构造需要好几个步骤，还需要区分不同Android版本的差异，因此将这几个步骤抽象为一个简单的库，外部直接调用对应的方法即可。

引入库步骤：

1、project build.gradle 里加入：

allprojects {

repositories {

...

//库是发布在jitpack上，因此需要指定位置

maven { url 'https://jitpack.io' }

}

}

2、在module build.gradle 里加入：

dependencies {

...

//引入EasyStorage库

implementation 'com.github.fishforest:EasyStorage:1.0.1'

}

3、使用方式：

EasyFileProvider.fillIntent(this, new File(filePath), intent, true);

如上一行代码搞定。

效果如下：

gif.jj.gif

本文基于Android 10.0

演示代码与库源码 若是有帮助，给github 点个赞呗～

您若喜欢，请点赞、关注，您的鼓励是我前进的动力

持续更新中，和我一起步步为营系统、深入学习Android/Java

式样

微软雅黑 Light 5号

段落 固定值 23磅

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5