账号安全方案？

AccountManagerService负责管理手机中用户的在线账户，主要的工作涉及账户的添加、删除和AuthToken的获取和更新

# TODO

http://wiki.jikexueyuan.com/project/deep-android-v2/content-account.html

# 简介

AccountManager帐号管理器，集中管理apps注册的不同类型的帐号。 不同类型的帐号服务会使用不同的帐号登录和鉴权方式，所以AccountManager为不同类型的帐号提供一个插件式authenticator模块，authenticators自己处理帐号登录/认证的具体细节，也可以自己存储帐号信息

简言之，AccountManager是一个面向应用程序开发的组件，它提供了一套对应于IAccountManager协议的应用程序接口；这组接口通过Binder机制与系统服务AccountManagerService进行通信，协作完成帐号相关的操作。同时，AccountManager接收authenticators提供的回调，以便在帐号操作完成之后向调用此帐号服务的业务返回对应的接口，同时触发这个业务对结果的处理。

- authenticators 即注册帐号服务的app；

- 业务调用方 即使用authenticators提供的帐号服务的第三方，也可以是authenticator自己

# 使用AccountManager注册帐号服务

如果应用想要注册一个新的帐号服务，必须实现AbstractAccountAuthenticator类，这是创建一个account authenticator的抽象基础类；然后新建一个authenticator service，注册action必须为”android.accounts.AccountAuthenticator”，且该service要实现onBinder(android.content.Intent)方法，返回AbstractAccountAuthenticator实现类的实例

说下必须要注册一个action为”android.accounts.AccountAuthenticator”的authenticator service：

首先，AbstractAccountAuthenticator是创建一个account authenticator必须实现的抽象基础类，接口协议定义在IAccountAuthenticator中，是一个authenticator自定义自己登录/认证等的接口协议；

那如何将authenticator的实现回调给AccountManagerService，供其调起authenticator的具体实现呢？ 就是通过action注册为”android.accounts.AccountAuthenticator”的authenticator service了：

这个action即为AccountManager#ACTION\_AUTHENTICATOR\_INTENT的常量值，系统服务AccountManagerService是通过bind到action为AccountManager#ACTION\_AUTHENTICATOR\_INTENT的intent service上来调起某个账号类型的authenticator service，然后通过调用这个service的getBinder()方法来获取AbstractAccountAuthenticator的实现实例，进而调用authenticator对帐号登录认证等服务的具体实现

至于每个帐号服务都定义一个action为”android.accounts.AccountAuthenticator”的service，那AccountManagerService是如何区分的呢？

当然是通过账号类型了，每个accountType只能对应一个authenticator

那系统是如何知道每个authenticator service对应的账号类型？

在AndroidManifest.xml中注册authenticator service时声明帐号属性的meta-data配置，声明的meta-data是一个name为 “android.accounts.AccountAuthenticator”的xml 资源（AccountManager#AUTHENTICATOR\_META\_DATA\_NAME），该XML资源文件定义了account-authenticator用到的一些属性：如accountType；系统解析authenticator service info之后，loadXmlMetaData获取authenticator 的xml属性，然后利用 Xml.asAttributeSet即

final PackageManager pm = mContext.getPackageManager();

final List<ResolveInfo> resolveInfos = pm.queryIntentServicesAsUser(new Intent("android.accounts.AccountAuthenticator", PackageManager.GET\_META\_DATA | PackageManager.MATCH\_DIRECT\_BOOT\_AWARE| PackageManager.MATCH\_DIRECT\_BOOT\_UNAWARE, userId);

for (ResolveInfo resolveInfo : resolveInfos) {

android.content.pm.ServiceInfo si = service.serviceInfo;

ComponentName componentName = new ComponentName(si.packageName, si.name);

PackageManager pm = mContext.getPackageManager();

XmlResourceParser parser = null;

try {

parser = si.loadXmlMetaData(pm, "android.accounts.AccountAuthenticator")

if (parser == null) {

throw new XmlPullParserException("No " + mMetaDataName + " meta-data");

}

AttributeSet attrs = Xml.asAttributeSet(parser);

...//解析authenticator xml的帐号属性

}

...

## 测试帐号

### TestAccountAuthenticator

Dsfg

创建一个继承自AbstractAccountAuthenticator的类TestAccountAuthenticator

public class TestAccountAuthenticator extends AbstractAccountAuthenticator {

private Context mContext;

public TestAccountAuthenticator(Context context) {

super(context);

mContext = context;

}

@Override

public Bundle editProperties(AccountAuthenticatorResponse response, String accountType) {

return null;

}

@Override

public Bundle addAccount(AccountAuthenticatorResponse response, String accountType, String authTokenType, String[] requiredFeatures, Bundle options) throws NetworkErrorException {//登录界面的定制化实现

Intent addAccountIntent = new Intent(mContext, LoginActivity.class);

addAccountIntent.putExtra("authTokenType", authTokenType);

if (options != null) {

addAccountIntent.putExtras(options);

}

addAccountIntent.putExtra(AccountManager.KEY\_ACCOUNT\_AUTHENTICATOR\_RESPONSE, response);//一定要把response传入intent的extra中，便于将登录操作的结果回调给AccountManager

Bundle bundle = new Bundle();

bundle.putParcelable(AccountManager.KEY\_INTENT, addAccountIntent);

return bundle;

}

@Override

public Bundle getAccountRemovalAllowed(AccountAuthenticatorResponse response, Account account) throws NetworkErrorException {//是否允许删除你的账号，这里是不允许删除，可自定义什么时候可以被删除，默认是true

Bundle bundle = new Bundle();

bundle.putBoolean(AccountManager.KEY\_BOOLEAN\_RESULT, false);

return bundle;

}

@Override

public Bundle confirmCredentials(AccountAuthenticatorResponse response, Account account, Bundle options)

throws NetworkErrorException {//自己实现：验证用户的密码

return null;

}

@Override

public Bundle getAuthToken(AccountAuthenticatorResponse response, Account account, String authTokenType, Bundle options)

throws NetworkErrorException {//自己完成获取鉴权token的流程

return null;

}

@Override

public String getAuthTokenLabel(String authTokenType) {

return null;

}

@Override

public Bundle updateCredentials(AccountAuthenticatorResponse response, Account account, String authTokenType, Bundle options)

throws NetworkErrorException {

return null;

}

@Override

public Bundle hasFeatures(AccountAuthenticatorResponse response, Account account, String[] features) throws NetworkErrorException {

return null;

}

}

### TestAuthenticatiorService

Dfg创建一个authenticator service—TestAuthenticatiorService，实现onBinder()方法，在onBinder方法里返回TestAccountAuthentcator的实例

public class TestAuthenticatiorService extends Service {

private static final String TAG = "XmAuthenticationService";

private TestAccountAuthenticator mAuthenticator;

@Override

public void onCreate() {

super.onCreate();

mAuthenticator = new TestAccountAuthenticator(this);

}

@Nullable

@Override

public IBinder onBind(Intent intent) {

return mAuthenticator.getIBinder();

}

}

#### AndroidManifest.xml

Ds‘

在AndroidManifest.xml文件中注册该TestAuthenticatorService

<service

android:name=".TestAuthenticatiorService"

android:exported="true">

<intent-filter>

<action android:name="android.accounts.AccountAuthenticator" />

</intent-filter>

<meta-data

android:name="android.accounts.AccountAuthenticator"

android:resource="@xml/authenticator" />

</service>

#### Authenticator

Fdg

其中，authenticator是一个xml的资源文件，定义了account的一些属性

<?xml version="1.0" encoding="utf-8"?>

<account-authenticator

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

android:accountType="com.test"//账号类型

android:icon="@drawable/icon"//设置-同步-添加 账号类型的icon

android:smallIcon="@drawable/miniIcon"//小icon

android:label="@string/label"//设置-同步-添加 账号类型的名称

android:accountPreferences="@xml/account\_preferences"//在设置中展示的一些偏好

android:customTokens="false"//authenticator是否要自己处理auth token的存储和获取权限

/>

ps：说下customTokens属性

如设置为true,就需要在TestAccountAuthenticator类的getAuthToken方法的实现中自己进行caller app的权限检查和token存储问题

如不设置（默认为false）或设置为false，则是使用AccountManager的权限检查和存储机制，**默认只有签名相同的app才可调用getAuthToken()方法**，存储在系统数据库中，但要app判断是否有效，失效要调用invalidate才可清除系统的存储

# Api介绍

## AccountManagerFuture

AccountManagerFuture类似于java的Future，提供了对AccountManager任务的执行进行取消，查询是否已经被取消，是否完成以及获取执行结果，通过getResult方法获取执行结果，该方法会阻塞直到任务返回结果

public interface AccountManager<V> {

/\*\*取消任务的执行

如果任务已经完成/已经取消/不能被取消，则返回false

\*/

boolean cancel(boolean mayInterruptIfRunning);

/\*\*任务是否被取消

如果任务在完成之前被取消，则返回true

\*/

boolean isCancelled();

/\*\*任务是否完成

\*/

boolean isDone();

/\*\*获取任务的执行结果

这个接口可能会阻塞当前线程，直到任务完成/任务被cancel，故不要在主线调用此接口

\*/

V getResult() throws OperationCanceledException, IOException, AuthenticatorException;

/\*\*同上

参数规定了等待的时间timeout，如果超时，则会直接被cancel

\*/

V getResult(long timeout, TimeUnit unit) throws OperationCanceledException, IOException, AuthenticatorException;

}

## AccountManagerCallback

fAccountManagerCallback，即AccountManager接口的回调，可在主线程执行

public interface AccountManagerCallback<V> {

void run(AccountManagerFuture<V> future);

}

## AccountManager

### 获取AccountManager实例

Sd

\*\*

获取AccountManager实例

备注：context最好传递application context，避免可能的不必要的内存泄漏

\*/

public static AccountManager get(Context context)；

### authenticator app

下面这些即将列出的方法主要是for authenticator app，其他需要app主要是调用authToken相关方法

所以下面下列方法的调用是有权限求的：调用下面接口的app，要与对应帐号的authenticator app签名一致

还有一点：下面方法均可在主线程调用

/\*\*

将一个帐号直接存储到AccountManager

\*/

public boolean addAccountExplicitly(Account account, String password, Bundle userdata)

/\*\*

获取account的password（不一定是真实的用户密码，看authenticator app的存储）

\*/

public String getPassword(final Account account)；

/\*\*

重置该account的password

注意：调用这个方法会清除该账号所有authToken在AccountManager中的缓存

\*/

public void setPassword(final Account account, final String password) ;

/\*\*

清除account的password

注意： 调用这个方法会清除该账号的password及authToken在AccountManager中的缓存

\*/

public void clearPassword(final Account account) ;

/\*\*

将account对应key-value存储在AccountManager中

\*/

public void setUserData(final Account account, final String key, final String value);

/\*\*

获取和该帐号相关的对应key值的value

\*/

public String getUserData(final Account account, final String key)；

/\*\*

从AccountManager中删除account，由authenticator app决定是否可以删除该帐号

注意：这个方法可以在主线程调用，但是返回的AccountManagerFuture不能在主线程中使用

\*/

public AccountManagerFuture<Bundle> removeAccount(final Account account, final Activity activity, AccountManagerCallback<Bundle> callback, Handler handler);

/\*\*

直接从AccountManager数据库中删除account

\*/

public boolean removeAccountExplicitly(Account account)

### 使用帐号服务的apps调用

当然authenticator app也可调用

/\*\*

获取和该帐号相关的对应key值的value

备注：这个方法可在主线程调用

权限要求：和注册account的authenticator app签名一致

\*/

public String getUserData(final Account account, final String key)；

/\*\*

获取当前注册到AccountManager中的所有authenticator app的属性（即在authenticator xml中定义的account属性，再加上authenticator app的包名）

备注：这个方法可在主线程调用

\*/

public AuthenticatorDescription[] getAuthenticatorTypes()；

/\*\*

获取当前存储到AccountManager的所有帐号列表，和getAccountsByType（null）相同

备注：这个方法可在主线程调用

权限要求：需要GET\_ACCOUNTS的权限（这个权限在android 6.0及以上是运行时权限，需要动态申请）

\*/

public Account[] getAccounts() ；

public Account[] getAccountsByTypeForPackage(String type, String packageName) ；

/\*\*

获取特定account type

备注：这个方法可在主线程调用

权限要求：需要GET\_ACCOUNTS的权限（这个权限在android 6.0及以上是运行时权限，需要动态申请）

\*/

public Account[] getAccountsByType(String type)；

/\*\*

删除存储AccountManager中此账号类型对应的authToken缓存，应用必须调用这个方法将缓存的authToken置为过期，否则getAuthToken获取到的一直是缓存的token

备注：这个方法可以在主线程中调用

\*/

public void invalidateAuthToken(final String accountType, final String authToken) ;

/\*\*

获取AccountManager缓存的某个账号对应authTokenType的authToken

如果缓存中没有对应authTokenType的token值，并不会重新生成，而是直接返回false

备注：这个方法可以在主线程中调用

权限要求：和注册account的authenticator app签名相同

\*/

public String peekAuthToken(final Account account, final String authTokenType) ;

/\*\*

将一个account对应autheTokenType的authToken缓存在AccountManager中

备注：这个方法可以在主线程中调用

权限要求：和注册account的authenticator app签名相同

\*/

public void setAuthToken(Account account, final String authTokenType, final String authToken);

/\*\*

同步获取account对应authTokenType的authToken

notifyAuthFailure，如果authenticator app返回出错，要不要弹通知提醒

备注：这个方法可以可能涉及到网络请求，不要在主线程调用

权限要求：和注册account的authenticator app签名相同

\*/

public String blockingGetAuthToken(Account account, String authTokenType, boolean notifyAuthFailure);

/\*\*

权限要求：和注册account的authenticator app签名相同

\*/

public AccountManagerFuture<Bundle> getAuthToken(final Account account, final String authTokenType, final Bundle options, final Activity activity, AccountManagerCallback<Bundle> callback, Handler handler);

/\*\*

权限要求：和注册account的authenticator app签名相同

\*/

public AccountManagerFuture<Bundle> getAuthToken(final Account account, final String authTokenType, final Bundle options, final boolean notifyAuthFailure, AccountManagerCallback<Bundle> callback, Handler handler);

/\*\*

要求用户添加某个特定accountType的帐号，如没有则会引导注册（authenticator app自己的处理）

注意：调用这个接口会调起authenticator app帐号登录的页面。传的activity参数就是用来启动这个intent的：如果传的activity参数不为null，则AccountManager会自动帮你start 登录的intent，否则你自己调用future.getResult()，返回的结果中会有AccountManager#KEY\_INTENT对应的帐号登录页面的intent

\*/

public AccountManagerFuture<Bundle> addAccount(final String accountType, final String authTokenType, final String[] requiredFeatures, final Bundle addAccountOptions, final Activity activity, AccountManagerCallback<Bundle> callback, Handler handler);

/\*\*

认证用户的身份（一般是验证帐号的密码）

注意：验证的行为由authenticator app来定义，同addAccount会调起authenticator app认证的页面

\*/

public AccountManagerFuture<Bundle> confirmCredentials(final Account account, final Bundle options, final Activity activity, final AccountManagerCallback<Bundle> callback, final Handler handler);

### 获取帐号属性

D

/\*\*

获取当前注册到AccountManager中的所有authenticator app的属性（即在authenticator xml中定义的account属性，再加上authenticator app的包名）

备注：这个方法可在主线程调用

\*/

public AuthenticatorDescription[] getAuthenticatorTypes()；

# 启动流程

## startService

看下AccountManagerService的初始化，在SystemServer中的startOtherServices方法看到如下代码：

mSystemServiceManager.startService(ACCOUNT\_SERVICE\_CLASS);

其中ACCOUNT\_SERVICE\_CLASS为

com.android.server.accounts.AccountManagerService$Lifecycle，SystemServiceManager的startService里面会通过反射创建AccountManagerService.Lifecycle对象，并调用其onStart方法

## Lifecycle.onStart

public static class Lifecycle extends SystemService {

public void onStart() {

mService = new AccountManagerService(getContext());

publishBinderService(Context.ACCOUNT\_SERVICE, mService);

}

## New AccountMS

可以看到创建AccountManagerService对象，并发布到服务总管里面。下面看下AccountManagerService构造函数：

public AccountManagerService(Context context) {

this(context, context.getPackageManager(), new AccountAuthenticatorCache(context));

}

public AccountManagerService(Context context, PackageManager packageManager,

IAccountAuthenticatorCache authenticatorCache) {

mContext = context;

mPackageManager = packageManager;

mAppOpsManager = mContext.getSystemService(AppOpsManager.class);

synchronized (mCacheLock) {

//此数据库文件对应为/data/system/accounts.db

mOpenHelper = new DatabaseHelper(mContext);

}

mMessageHandler = new MessageHandler(FgThread.get().getLooper());

mAuthenticatorCache = authenticatorCache;

mAuthenticatorCache.setListener(this, null /\* Handler \*/);

### accounts.db

sdf

/\*

accounts.db数据库中有一个grants表，用于存储授权信息，该信息用于保存哪些Package

有权限获取账户信息。下面的函数将根据grants表中的数据查询PKMS，以判断这些

Package是否还存在。如果系统中已经不存在这些Package，则grants表需要更新

\*/

purgeOldGrants();

/\*

accounts.db中有一个accounts表，该表中存储了账户类型和账户名。其中，账户类型

就是AuthenticatorDescription中的accountType，它和具体应用有关。下面这个

函数将比较accounts表中的内容与AccountAuthenticatorCache中服务的信息，如果

AccountAuthenticatorCache已经不存在对应账户类型的服务，则需要删除accounts表

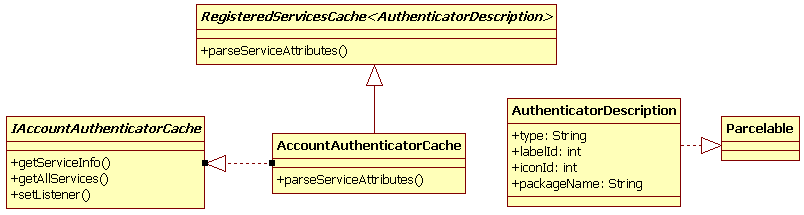
中的对应项

\*/

## AccountAuthenticatorCache

这里又创建了AccountAuthenticatorCache对象并设置监听器，AccountAuthenticatorCache是android平台中账户验证服务（AAS）的管理中心，AAS是应用程序中定义的服务，对它的定义有一定的要求，后面会看到。接着看下AccountAuthenticatorCache的构造函数

先来看AccountAuthenticatorCache的派生关系，如图8-3所示



### AccountAuthenticatorCache

/\* package private \*/ class AccountAuthenticatorCache

extends RegisteredServicesCache<AuthenticatorDescription>

public AccountAuthenticatorCache(Context context) {

super(context, AccountManager.ACTION\_AUTHENTICATOR\_INTENT,

AccountManager.AUTHENTICATOR\_META\_DATA\_NAME,

AccountManager.AUTHENTICATOR\_ATTRIBUTES\_NAME, sSerializer);

}

这里又调用父类RegisteredServicesCache构造函数，传递的参数如下定义：

public static final String ACTION\_AUTHENTICATOR\_INTENT =

"android.accounts.AccountAuthenticator";

public static final String AUTHENTICATOR\_META\_DATA\_NAME =

"android.accounts.AccountAuthenticator";

public static final String AUTHENTICATOR\_ATTRIBUTES\_NAME = "account-authenticator";

后面解析XML要用到，所以AAS的配置文件也需要符合这个要求。

### RegisteredServicesCache

public RegisteredServicesCache(Context context, String interfaceName, String metaDataName,

String attributeName, XmlSerializerAndParser<V> serializerAndParser) {

mContext = context;

mInterfaceName = interfaceName;

mMetaDataName = metaDataName;

mAttributesName = attributeName;

mSerializerAndParser = serializerAndParser;

FilesystemDir = new File(dataDir, "system");

//syncDir指向/data/system/registered\_service目录

FilesyncDir = new File(systemDir, "registered\_services");

//下面这个文件指向syncDir目录下的android.accounts.AccountAuthenticator.xml

mPersistentServicesFile = new AtomicFile(new File(syncDir,

interfaceName+ ".xml"));

migrateIfNecessaryLocked();

//注册Package安装、卸载和更新等广播监听者

IntentFilter intentFilter = new IntentFilter();

intentFilter.addAction(Intent.ACTION\_PACKAGE\_ADDED);

intentFilter.addAction(Intent.ACTION\_PACKAGE\_CHANGED);

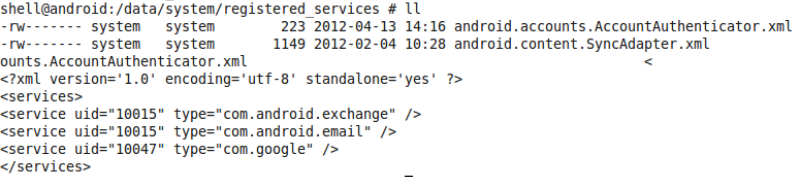
intentFilter.addAction(Intent.ACTION\_PACKAGE\_REMOVED);

intentFilter.addDataScheme("package");

mContext.registerReceiverAsUser(mPackageReceiver, UserHandle.ALL, intentFilter, null, null)

由以上代码可知：

· 成员变量mPersistentServicesFile指向/data/system/registered\_service/目录下的一个文件，该文件保存了以前获取的对应Service的信息。就AccountAuthenticator而言，mPersistentServicesFile指向该目录的android.accounts.AccountAuthenticator.xml文件

。

· 由于RegisteredServicesCache管理的是系统中指定Service的信息，当系统中有Package安装、卸载或更新时，RegisteredServicesCache也需要对应更新自己的信息，因为有些Service可能会随着APK被删除而不复存在。

#### migrateIfNecessaryLocked

其中migrateIfNecessaryLocked用户在系统目录下创建文件：

File systemDir = new File(getDataDirectory(), "system");

File syncDir = new File(systemDir, REGISTERED\_SERVICES\_DIR);

AtomicFile oldFile = new AtomicFile(new File(syncDir, mInterfaceName + ".xml"))

#### mPackageReceiver

接着注册了包添加删除广播。猜测应用安装或卸载会在这里进行账户信息添加和删除。下面看下这个广播接收器：

private final BroadcastReceiver mPackageReceiver = new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

final int uid = intent.getIntExtra(Intent.EXTRA\_UID, -1);

if (uid != -1) {

handlePackageEvent(intent, UserHandle.getUserId(uid));

}

}

};

##### handlePackageEvent

接着进入handlePackageEvent：

private final void handlePackageEvent(Intent intent, int userId) {

// Don't regenerate the services map when the package is removed or its

// ASEC container unmounted as a step in replacement. The subsequent

// \_ADDED / \_AVAILABLE call will regenerate the map in the final state.

final String action = intent.getAction();

// it's a new-component action if it isn't some sort of removal

final boolean isRemoval = Intent.ACTION\_PACKAGE\_REMOVED.equals(action)

|| Intent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE.equals(action);

// if it's a removal, is it part of an update-in-place step?

final boolean replacing = intent.getBooleanExtra(Intent.EXTRA\_REPLACING, false);

if (isRemoval && replacing) {

// package is going away, but it's the middle of an upgrade: keep the current

// state and do nothing here. This clause is intentionally empty.

} else {

int[] uids = null;

// either we're adding/changing, or it's a removal without replacement, so

// we need to update the set of available services

if (Intent.ACTION\_EXTERNAL\_APPLICATIONS\_AVAILABLE.equals(action)

|| Intent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE.equals(action)) {

uids = intent.getIntArrayExtra(Intent.EXTRA\_CHANGED\_UID\_LIST);

} else {

int uid = intent.getIntExtra(Intent.EXTRA\_UID, -1);

if (uid > 0) {

uids = new int[] { uid };

}

}

generateServicesMap(uids, userId);

}

##### generateServicesMap

当包正在删除且部分在更新情况不做任何处理，接着进入generateServicesMap：

private void generateServicesMap(int[] changedUids, int userId) {

if (DEBUG) {

Slog.d(TAG, "generateServicesMap() for " + userId + ", changed UIDs = " + changedUids);

}

final ArrayList<ServiceInfo<V>> serviceInfos = new ArrayList<ServiceInfo<V>>();

final List<ResolveInfo> resolveInfos = queryIntentServices(userId);

for (ResolveInfo resolveInfo : resolveInfos) {

try {

ServiceInfo<V> info = parseServiceInfo(resolveInfo);

if (info == null) {

Log.w(TAG, "Unable to load service info " + resolveInfo.toString());

continue;

}

serviceInfos.add(info);

} catch (XmlPullParserException|IOException e) {

Log.w(TAG, "Unable to load service info " + resolveInfo.toString(), e);

}

}

首先传递userid通过过PMS查询应用包服务信息。接着通过parseServiceInfo解析服务信息：

##### parseServiceInfo

protected ServiceInfo<V> parseServiceInfo(ResolveInfo service)

throws XmlPullParserException, IOException {

android.content.pm.ServiceInfo si = service.serviceInfo;

ComponentName componentName = new ComponentName(si.packageName, si.name);

PackageManager pm = mContext.getPackageManager();

XmlResourceParser parser = null;

parser = si.loadXmlMetaData(pm, mMetaDataName);

AttributeSet attrs = Xml.asAttributeSet(parser);

V v = parseServiceAttributes(pm.getResourcesForApplication(si.applicationInfo),

si.packageName, attrs);

if (v == null) {

return null;

}

final android.content.pm.ServiceInfo serviceInfo = service.serviceInfo;

return new ServiceInfo<V>(v, serviceInfo, componentName);

解析MetaData信息，接着调用子类parseServiceAttributes来解析MetaData中的resource信息。

### parseServiceAttributes

public AuthenticatorDescription parseServiceAttributes(Resources res,

String packageName, AttributeSet attrs) {

TypedArray sa = res.obtainAttributes(attrs,

com.android.internal.R.styleable.AccountAuthenticator);

try {

final String accountType =

sa.getString(com.android.internal.R.styleable.AccountAuthenticator\_accountType);

final int labelId = sa.getResourceId(

com.android.internal.R.styleable.AccountAuthenticator\_label, 0);

final int iconId = sa.getResourceId(

com.android.internal.R.styleable.AccountAuthenticator\_icon, 0);

final int smallIconId = sa.getResourceId(

com.android.internal.R.styleable.AccountAuthenticator\_smallIcon, 0);

final int prefId = sa.getResourceId(

com.android.internal.R.styleable.AccountAuthenticator\_accountPreferences, 0);

final boolean customTokens = sa.getBoolean(

com.android.internal.R.styleable.AccountAuthenticator\_customTokens, false);

if (TextUtils.isEmpty(accountType)) {

return null;

}

return new AuthenticatorDescription(accountType, packageName, labelId, iconId,

smallIconId, prefId, customTokens);

} finally {

sa.recycle();

}

## MetaData

MetaData的resource一般是xml文件，下面举例SimContact模块中的AAS（SimAuthenticateService），看下它的配置文件和MetaDta的resource：



上面xml文件中的accountType标签用于指定账户类型，icon、smallIcon、label等用于界面显示。最终将sim\_authenticator文件解析封装到AuthenticatorDescription对象中返回。

上面xml文件中的accountType标签用于指定账户类型，icon、smallIcon、label等用于界面显示。最终将sim\_authenticator文件解析封装到AuthenticatorDescription对象中返回。

# 添加账户

一般默认添加账户都是在**设置**里面添加，应用程序也可以自己添加。添加账户入口在AccountManager类的addAccount：

## addAccount

public AccountManagerFuture<Bundle> addAccount(final String accountType,

final String authTokenType, final String[] requiredFeatures,

final Bundle addAccountOptions,

final Activity activity, AccountManagerCallback<Bundle> callback, Handler handler) {

return new AmsTask(activity, handler, callback) {

@Override

public void doWork() throws RemoteException {

mService.addAccount(mResponse, accountType, authTokenType,

requiredFeatures, activity != null, optionsIn);

}

}.start();

}

### 参数

Sf

在以上代码中：

· addAccount的返回值类型是AccountManagerFuture<Bundle>。其中，AccountManagerFuture是一个模板Interface，其真实类型只有在分析addAccount的实现时才能知道。现在可以告诉读者的是，它和Java并发库（concurrent库）中的FutureTask有关，是对异步函数调用的一种封装[①]。调用者在后期只要调用它的getResult函数即可取得addAccount的调用结果。由于addAccount可能涉及网络操作（例如，AAS需要把账户添加到网络服务器上），所以这里采用了异步调用的方法以避免长时间的阻塞。这也是AccountManagerFuture的getResult不能在主线程中调用的原因。

· addAccount的第一个参数accountType代表账户类型。该参数不能为空。就本例而言，它的值为“com.android.email”。

· authTokenType、requiredFeatures和addAccountOptions与具体的AAS服务有关。如果想添加指定账户类型的Account，则须对其背后的AAS有所了解。

· activity：此参数和界面有关。例如有些AAS需要用户输入用户名和密码，故需启动一Activity。在这种情况下，AAS会返回一个Intent，客户端将通过这个activity启动Intent所标示的Activity。读者将通过下文的分析了解这一点。

· callback和handler：这两个参数与如何获取addAccount返回结果有关。如这两个参数为空，客户端则须单独启动一个线程去调用AccountManagerFuture的getResult函数。

### AmsTask

Sd

先来看AmsTask的继承关系，如图8-7所示。

image

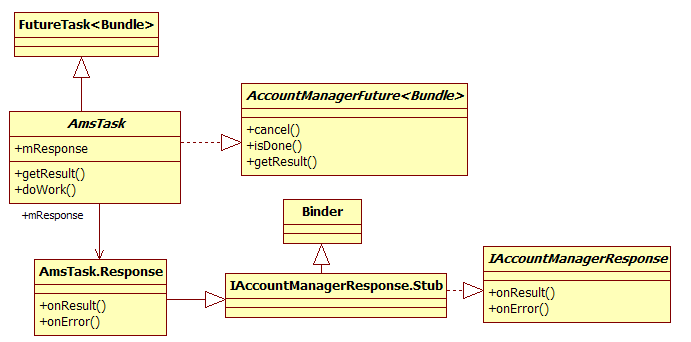


图8-7 AmsTask继承关系

由图8-7可知：

· AmsTask继承自FutureTask，并实现了AccountManagerFuture接口。FutureTask是Java concurrent库中一个常用的类。AmsTask定义了一个doWork虚函数，该函数必须由子类来实现。

· 一个AmsTask对象中有一个mResponse成员，该成员的类型是AmsTask中的内部类Response。从Response的派生关系可知，Response将参与Binder通信，并且它是Binder通信的Bn端。而AccountManagerService的addAccount将得到它的Bp端对象。当添加完账户后，AccountManagerService会通过这个Bp端对象的onResult或onError函数向Response通知处理结果。

这里返回AmsTask对象，这是什么鬼？

private abstract class AmsTask extends FutureTask<Bundle> implements AccountManagerFuture<Bundle> {

final IAccountManagerResponse mResponse;

final Handler mHandler;

final AccountManagerCallback<Bundle> mCallback;

final Activity mActivity;

public AmsTask(Activity activity, Handler handler, AccountManagerCallback<Bundle> callback) {

super(new Callable<Bundle>() {

@Override

public Bundle call() throws Exception {

throw new IllegalStateException("this should never be called");

}

});

mHandler = handler;

mCallback = callback;

mActivity = activity;

mResponse = new Response();

}

public final AccountManagerFuture<Bundle> start() {

try {

doWork();

} catch (RemoteException e) {

setException(e);

}

return this;

}

public abstract void doWork() throws RemoteException;

private class Response extends IAccountManagerResponse.Stub {

@Override

public void onResult(Bundle bundle) {

Intent intent = bundle.getParcelable(KEY\_INTENT);

mActivity.startActivity(intent);

}

@Override

public void onError(int code, String message) {

}

}

它继承FutureTask实现AccountMana Future，关于FutureTask介绍可查看链接 FutureTask 深度解析 ，构造函数里面还创建了Response对象，它继承IAccountManagerResponse.Stub，根据经验它是服务的实现者，用于响应AccountManagerService的回调。接着调用start-》dowork（）进入到AccountManagerService中的addAccount：

## AccountManagerService.addAccount

//检查客户端进程是否有“android.permission.MANAGE\_ACCOUNTS”的权限

public void addAccount(final IAccountManagerResponse response, final String accountType,

final String authTokenType, final String[] requiredFeatures,

final boolean expectActivityLaunch, final Bundle optionsIn) {

UserAccounts accounts = getUserAccounts(usrId);

logRecordWithUid(

accounts, DebugDbHelper.ACTION\_CALLED\_ACCOUNT\_ADD, TABLE\_ACCOUNTS, uid);

new Session(accounts, response, accountType, expectActivityLaunch,

true /\* stripAuthTokenFromResult \*/, null /\* accountName \*/,

false /\* authDetailsRequired \*/, true /\* updateLastAuthenticationTime \*/) {

@Override

public void run() throws RemoteException {

mAuthenticator.addAccount(this, mAccountType, authTokenType, requiredFeatures,

options);

}

@Override

protected String toDebugString(long now) {

return super.toDebugString(now) + ", addAccount"

+ ", accountType " + accountType

+ ", requiredFeatures "

+ (requiredFeatures != null

? TextUtils.join(",", requiredFeatures)

: null);

}

}.bind();

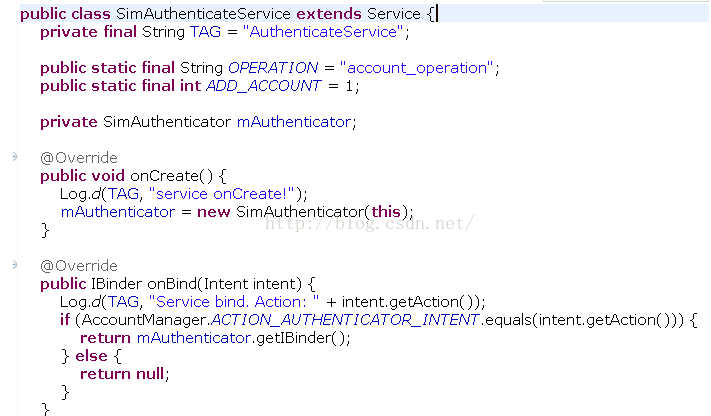
这里创建了一个Session对象，这又是什么鬼？

### Session

Sd

1. private abstract class Session extends IAccountAuthenticatorResponse.Stub
2. implements IBinder.DeathRecipient, ServiceConnection {
3. IAccountManagerResponse mResponse;
4. IAccountAuthenticator mAuthenticator = null;
5. public Session(UserAccounts accounts, IAccountManagerResponse response, String accountType,
6. boolean expectActivityLaunch, boolean stripAuthTokenFromResult, String accountName,
7. boolean authDetailsRequired, boolean updateLastAuthenticatedTime) {
8. super();
9. mAccounts = accounts;
10. mResponse = response;
11. mAccountType = accountType;
12. mAccountName = accountName;
13. synchronized (mSessions) {
14. mSessions.put(toString(), this);
15. }
16. }
17. void bind() {
18. if (!bindToAuthenticator(mAccountType)) {
19. }
20. }
21. public void onServiceConnected(ComponentName name, IBinder service) {
22. mAuthenticator = IAccountAuthenticator.Stub.asInterface(service);
23. run();
24. }
25. public abstract void run() throws RemoteException;
26. public void onResult(Bundle result) {
27. }
28. private boolean bindToAuthenticator(String authenticatorType) {
29. final AccountAuthenticatorCache.ServiceInfo<AuthenticatorDescription> authenticatorInfo;
30. authenticatorInfo = mAuthenticatorCache.getServiceInfo(
31. AuthenticatorDescription.newKey(authenticatorType), mAccounts.userId);
32. Intent intent = new Intent();
33. intent.setAction(AccountManager.ACTION\_AUTHENTICATOR\_INTENT);
34. intent.setComponent(authenticatorInfo.componentName);
35. if (Log.isLoggable(TAG, Log.VERBOSE)) {
36. Log.v(TAG, "performing bindService to " + authenticatorInfo.componentName);
37. }
38. if (!mContext.bindServiceAsUser(intent, this, Context.BIND\_AUTO\_CREATE,
39. UserHandle.of(mAccounts.userId))) {
40. }
41. return true;
42. }

它继承IAccountAuthenticatorResponse.Stub，是服务的实现者，跟谁通信？后面再看，又实现ServiceConnection，感觉在绑定服务的时候见过。接着调用bind()-》bindToAuthenticator，这里通过mAuthenticatorCache获取服务信息，mAuthenticatorCache在前面认识AccountManagerService介绍过。获取到ServiceInfo信息就开始绑定服务。这里以SimAuthenticateService为例，bindServiceAsUser调用后就会调用它的onBind方法：



这里的SimAuthenticator继承AbstractAccountAuthenticator，返回它的getIBinder()，其实是AbstractAccountAuthenticator中的内部类Transport对象的binder对象：

private Transport mTransport = new Transport();

/\*\*

\* @return the IBinder for the AccountAuthenticator

\*/

public final IBinder getIBinder() {

return mTransport.asBinder();

}

private class Transport extends IAccountAuthenticator.Stub {

@Override

public void addAccount(IAccountAuthenticatorResponse response, String accountType,

可见Transport又是服务的实现者，猜测在AccountManagerService会调用。

回到上面AccountManagerService中的添加服务流程中，刚才走到绑定服务，绑定成功会调用onServiceConnected函数，该函数中获取服务代理对象：

mAuthenticator = IAccountAuthenticator.Stub.asInterface(service);

接着run：

public void run() throws RemoteException {

mAuthenticator.addAccount(this, mAccountType, authTokenType, requiredFeatures,

options);

}

mAuthenticator为本地代理对象，调用addAccount会根据binder机制进入服务的实现者，也就是AbstractAccountAuthenticator中的内部类Transport：

final Bundle result = AbstractAccountAuthenticator.this.addAccount(

new AccountAuthenticatorResponse(response)

if (result != null) {

response.onResult(result);

}

这里调用AbstractAccountAuthenticator的子类，也就是上面举例的SimAuthenticator，注意上面传递的IAccountAuthenticatorResponse 的binder对象response，它的真正实现者在AccountManagerService内部类Session（private abstract class Session extends IAccountAuthenticatorResponse.Stub），也就是说这里调用response.onResult(result);又回到了Session中：

public void onResult(Bundle result) {

IAccountManagerResponse response;

response = mResponse;

if (response != null) {

response.onResult(result);

}

}

这里的response又是个binder对象。额，已经晕了。仔细回忆，它是在AccountManager的addAccount传递进来的，它是AmsTask的内部类Response：

private class Response extends IAccountManagerResponse.Stub {

@Override

public void onResult(Bundle bundle) {

Intent intent = bundle.getParcelable(KEY\_INTENT);

if (intent != null && mActivity != null) {

// since the user provided an Activity we will silently start intents

// that we see

mActivity.startActivity(intent);

整个流程差不多介绍完了

最后看下SimAuthenticator是如何添加服务的，它调用AccountManager的addAccountExplicitl

## addAccountExplicitl

public boolean addAccountExplicitly(Account account, String password, Bundle userdata) {

if (account == null) throw new IllegalArgumentException("account is null");

try {

return mService.addAccountExplicitly(account, password, userdata);

} catch (RemoteException e) {

throw e.rethrowFromSystemServer();

}

}

直接进入AccountManagerService的addAccountExplicitly：

public boolean addAccountExplicitly(Account account, String password, Bundle extras) {

UserAccounts accounts = getUserAccounts(userId);

return addAccountInternal(accounts, account, password, extras, callingUid);

## addAccountInternal

private boolean addAccountInternal(UserAccounts accounts, Account account, String password,

Bundle extras, int callingUid) {

final SQLiteDatabase db = accounts.openHelper.getWritableDatabaseUserIsUnlocked();

db.beginTransaction();

ContentValues values = new ContentValues();

values.put(ACCOUNTS\_NAME, account.name);

values.put(ACCOUNTS\_TYPE, account.type);

values.put(ACCOUNTS\_PASSWORD, password);

long accountId = db.insert(CE\_TABLE\_ACCOUNTS, ACCOUNTS\_NAME, values);

values = new ContentValues();

values.put(ACCOUNTS\_ID, accountId);

values.put(ACCOUNTS\_NAME, account.name);

values.put(ACCOUNTS\_TYPE, account.type);

values.put(ACCOUNTS\_LAST\_AUTHENTICATE\_TIME\_EPOCH\_MILLIS,

System.currentTimeMillis());

if (db.insert(TABLE\_ACCOUNTS, ACCOUNTS\_NAME, values) < 0)

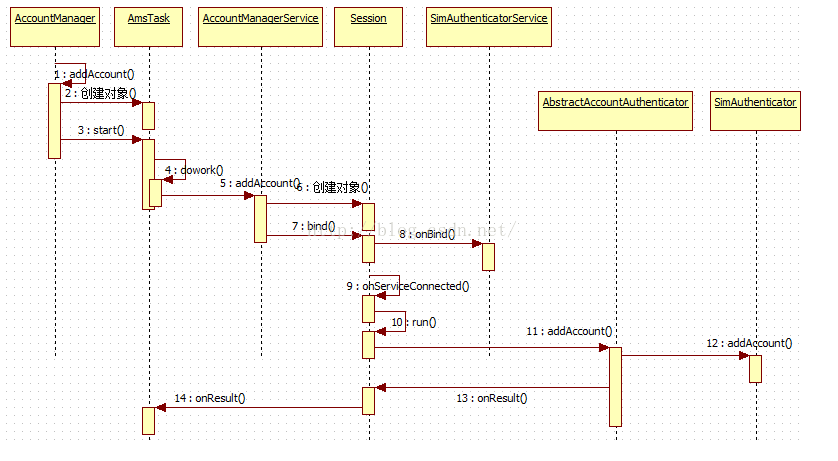
db.endTransaction();

}

sendAccountsChangedBroadcast(accounts.userId);

这里做了添加数据库的操作和发送账户修改广播。

## 最后贴一张添加服务的流程图：



# 数据库存储位置

数据库存储位置

由于android7.0引入了FBE(File-Based Encryption)：可以使用不同的密钥对不同的文件进行加密，并且可以对这些文件进行单独解密（具体参考文档：https://source.android.com/security/encryption/file-based）

在启用了FBE的设备，每个用户都有两个可供应用使用的存储位置：

CE(credential encryption)凭据加密存储空间：默认存储空间，只有在用户解锁设备后才可用

DE(device encryption)设备加密存储空间：在direct root模式下以及用户解锁设备后均可用

注：direct root表示当前设备已经开机但用户尚未解锁设备，android N版本引入此安全模式;如果您的应用需要在direct root模式下运行时需要访问数据，则使用DE存储。DE存储包含使用密钥加密数据，仅在设备已成功执行验证启动后密钥才可用。

直接启动（direct root）模式官方参考文档:https://developer.android.com/training/articles/direct-boot.html?hl=zh-cn

android 7.0上AccountManager的存储也引入了此特性，对应存储位置为：

/data/system\_ce/0/accounts\_ce.db

/data/system\_de/0/accounts\_de.db

另：android 7.0之前版本的存储位置： /data/system/users/0/accounts.db

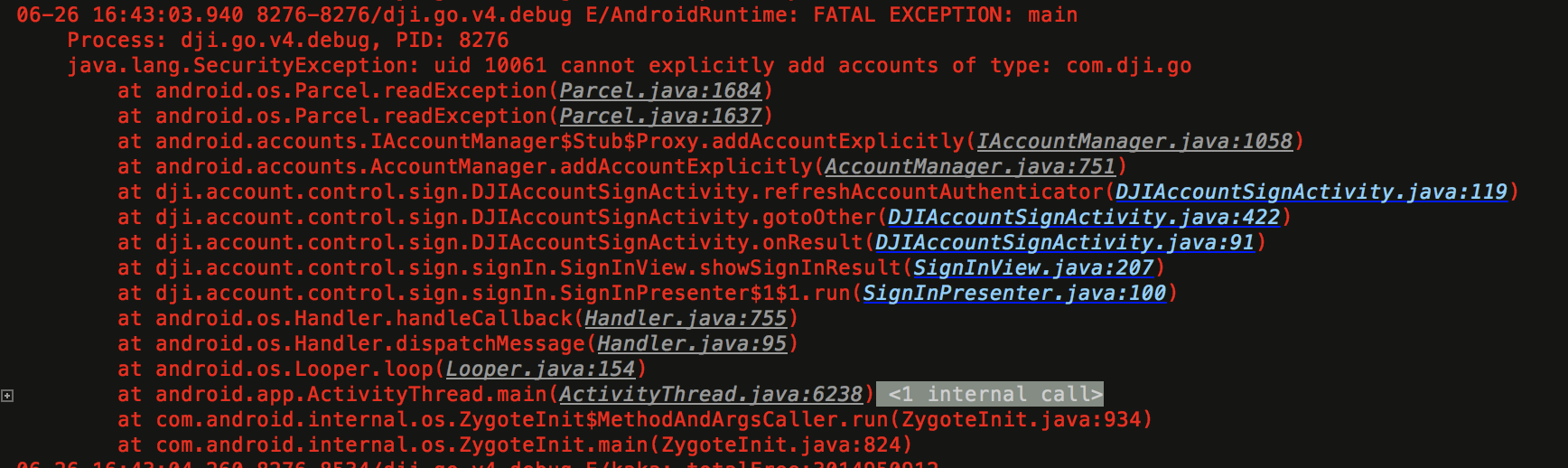
# REF

[基于N源码的AccountManagerService简单认识和账户添加流程分析](https://blog.csdn.net/yq6073025/article/details/52808094)

[Android AccountManager帐号管理（一）](https://blog.csdn.net/dzkdxyx/article/details/78569867)

[Android AccountManager帐号管理（二）](https://blog.csdn.net/dzkdxyx/article/details/78632945)

# 账号共享



uid

**callingUid=10061**

**account.type = com.dji.go**

String msg = String.format(

"uid %s cannot explicitly add accounts of type: %s",

callingUid,

account.type);

isAccountManagedByCaller(account.type, callingUid, userId)

isAccountManagedByCaller

return getTypesManagedByCaller(callingUid, userId).contains(accountType);

private List<String> getTypesManagedByCaller(int callingUid, int userId) {

return getTypesForCaller(callingUid, userId, false);

}

**private List<String> getTypesForCaller(**

**int callingUid, int userId, boolean isOtherwisePermitted) {**

**List<String> managedAccountTypes = new ArrayList<>();**

**long identityToken = Binder.clearCallingIdentity();**

**Collection<RegisteredServicesCache.ServiceInfo<AuthenticatorDescription>> serviceInfos;**

**try {**

**serviceInfos = mAuthenticatorCache.getAllServices(userId);**

**} finally {**

**Binder.restoreCallingIdentity(identityToken);**

**}**

**for (RegisteredServicesCache.ServiceInfo<AuthenticatorDescription> serviceInfo :**

**serviceInfos) {**

**final int sigChk = mPackageManager.checkSignatures(serviceInfo.uid, callingUid);**

**if (isOtherwisePermitted || sigChk == PackageManager.SIGNATURE\_MATCH) {**

**managedAccountTypes.add(serviceInfo.type.type);**

**}**

**}**

**return managedAccountTypes;**

**}**