PKMS第二节课

签名:

静默安装，**签名**，须知内容：

```
        // 如果想实现，静默安装，就需要设置好UID，只有设置这个UID后，才有安装的权限
        // 但是这个UID必须要求有系统的[签名]，而这个系统的[签名]是属于各大手机厂商的机密，也意味着非常的坑爹
        // 如果是系统厂商要做这种静默安装，那就是非常容易的事情，因为系统厂商可以轻而易举的拿到 系统的[签名]
        mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.phone", RADIO_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.log", LOG_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.nfc", NFC_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.shell", SHELL_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.se", SE_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
        mSettings.addSharedUserLPw("android.uid.networkstack", NETWORKSTACK_UID,
                ApplicationInfo.FLAG_SYSTEM,
ApplicationInfo.PRIVATE_FLAG_PRIVILEGED);
```

做系统应用开发测试时, 是 adb push 的安装方式:

```
1.adb remount

2.adb shell

3.chmod777 system/app

4.exit

5.adb push xxx/ooo.apk system/app

6.adb reboot

注：xxx/ooo.apk 是apk本地路径            system/app是系统目录
```

具体详情需要看 pkmsdemo 案例源码 ----->

requestPermissions源码流程解析

6.0 动态申请权限的前戏：

Google在 Android 6.0 开始引入了权限申请机制，将所有权限分成了**正常权限**和**危险权限**。

同学们注意：App每次在使用**危险权限**时需要动态的申请并得到用户的授权才能使用。

权限的分类：

系统权限分为两类：**正常权限 和 危险权限。**

正常权限不会直接给用户隐私权带来风险。如果您的应用在其清单中列出了正常权限，系统将自动授予该权限。

危险权限会授予应用访问用户机密数据的权限。如果您的应用在其清单中列出了正常权限，系统将自动授予该权限。如果您列出了危险权限，则用户必须明确批准您的应用使用这些权限。

那么，那些是**危险权限**呢，为什么是**危险权限**呢？ 要和同学们说清楚

```
    <!-- 权限组：CALENDAR == 日历读取的权限申请 -->
    <uses-permission android:name="android.permission.READ_CALENDAR" />
    <uses-permission android:name="android.permission.WRITE_CALENDAR" />

    <!-- 权限组：CAMERA == 相机打开的权限申请 -->
    <uses-permission android:name="android.permission.CAMERA" />

    <!-- 权限组：CONTACTS == 联系人通讯录信息获取/写入的权限申请 -->
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <uses-permission android:name="android.permission.WRITE_CONTACTS" />
```

```
    <!-- 权限组：LOCATION == 位置相关的权限申请 -->
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

    <!-- 权限组：PHONE == 拨号相关的权限申请 -->
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />

    <!-- 权限组：SMS == 短信相关的权限申请 -->
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.READ_SMS" />

    <!-- 权限组：STORAGE == 读取存储相关的权限申请 -->
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

核心函数：

`ContextCompat.checkSelfPermission`
检查应用是否具有某个危险权限。如果应用具有此权限，方法将返回 `PackageManager.PERMISSION_GRANTED`，并且应用可以继续操作。如果应用不具有此权限，方法将返回 `PackageManager.PERMISSION_DENIED`，且应用必须明确向用户要求权限。

`ActivityCompat.requestPermissions`
应用可以通过这个方法动态申请权限，调用后会弹出一个对话框提示用户授权所申请的权限。

`ActivityCompat.shouldShowRequestPermissionRationale`
如果应用之前请求过此权限但用户拒绝了请求，此方法将返回 `true`。如果用户在过去拒绝了权限请求，并在权限请求系统对话框中选择了 `Don't ask again` 选项，此方法将返回 `false`。如果设备规范禁止应用具有该权限，此方法也会返回 `false`。

`onRequestPermissionsResult`
当应用请求权限时，系统将向用户显示一个对话框。当用户响应时，系统将调用应用的 `onRequestPermissionsResult()` 方法，向其传递用户响应，处理对应的场景。

同学们注意，现在演示，上面 **“核心函数”** 实例：

```
<!-- 第一步：在AndroidManifest.xml中添加所需权限。 -->
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        requestPermission();
    }

    // 第二步：封装了一个requestPermission方法来动态检查和申请权限
```

```
    private void requestPermission() {

        Log.i(TAG,"requestPermission");

        // Here, thisActivity is the current activity
        if (ContextCompat.checkSelfPermission(this,
Manifest.permission.READ_CONTACTS) != PackageManager.PERMISSION_GRANTED) {
            Log.i(TAG,"checkSelfPermission");

            // Should we show an explanation?
            if (ActivityCompat.shouldShowRequestPermissionRationale(this,
Manifest.permission.READ_CONTACTS)) {
                Log.i(TAG,"shouldShowRequestPermissionRationale");

                // Show an expanation to the user *asynchronously* -- don't
block
                // this thread waiting for the user's response! After the user
                // sees the explanation, try again to request the permission.

                ActivityCompat.requestPermissions(this,
                        new String[]{Manifest.permission.READ_CONTACTS},
                        MY_PERMISSIONS_REQUEST_READ_CONTACTS);

            } else {
                Log.i(TAG,"requestPermissions");
                // No explanation needed, we can request the permission.
                ActivityCompat.requestPermissions(this,
                        new String[]{Manifest.permission.READ_CONTACTS},
                        MY_PERMISSIONS_REQUEST_READ_CONTACTS);
                // MY_PERMISSIONS_REQUEST_READ_CONTACTS is an
                // app-defined int constant. The callback method gets the
                // result of the request.
            }
        }
    }
```

```
    // 第三步：重写onRequestPermissionsResult方法根据用户的不同选择做出响应。
    @Override
    public void onRequestPermissionsResult(int requestCode,
                                           String permissions[], int[]
grantResults) {
        switch (requestCode) {
            case MY_PERMISSIONS_REQUEST_READ_CONTACTS: {
                // If request is cancelled, the result arrays are empty.
                if (grantResults.length > 0
                        && grantResults[0] == PackageManager.PERMISSION_GRANTED)
{
                    Log.i(TAG,"onRequestPermissionsResult granted");
                    // permission was granted, yay! Do the
                    // contacts-related task you need to do.

                } else {
                    Log.i(TAG,"onRequestPermissionsResult denied");
                    // permission denied, boo! Disable the
                    // functionality that depends on this permission.
                    showWaringDialog();
                }
```

```
                return;
            }

            // other 'case' lines to check for other
            // permissions this app might request
        }
    }

    // 如果点击 拒绝，就会弹出这个
    private void showWaringDialog() {
        new AlertDialog.Builder(this)
                .setTitle("警告！")
                .setMessage("请前往设置->应用->PermissionDemo->权限中打开相关权限，否则功能无法正常运行！")
                .setPositiveButton("确定", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        // 一般情况下如果用户不授权的话，功能是无法运行的，做退出处理
                        finish();
                    }
                }).show();
    }
```

运行结果：

requestPermissions源码整体

总结上面的几个 ”核心函数“

```
检查权限
checkSelfPermission(@NonNull String permission)

申请权限
requestPermissions(@NonNull String[] permissions, int requestCode)

处理结果回调
onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
@NonNull int[] grantResults)

是否需要显示UI界面提示用户为什么需要这个权限
shouldShowRequestPermissionRationale(@NonNull String permission)
```

权限申请源码流程总结:

第一步：MainActivity 调用 requestPermissions 进行动态权限申请；

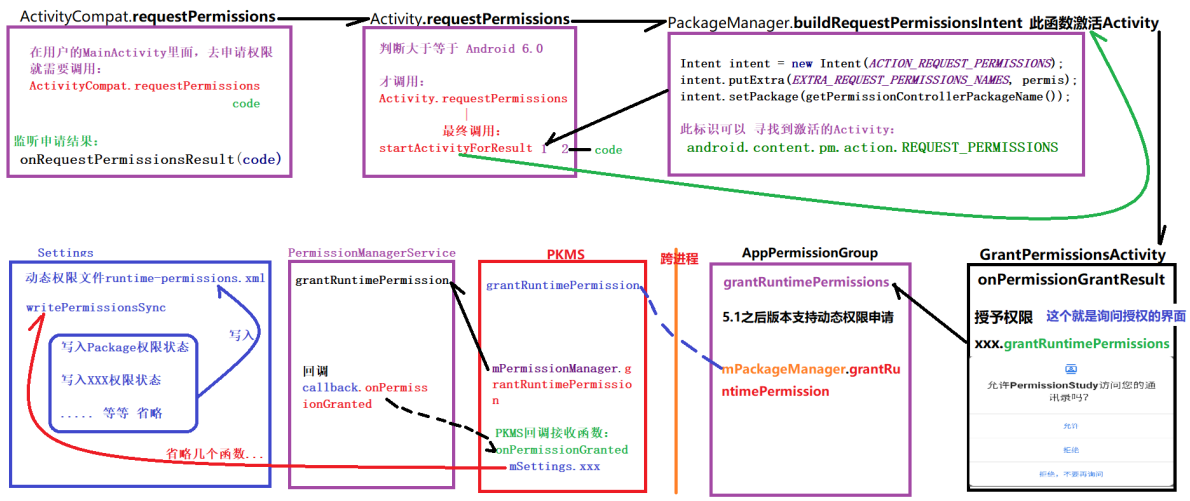
第二步：requestPermissions函数通过隐士意图，激活PackageInstaller的GrantPermissionsActivity界面，让用户选择是否授权；

第三步：经过PKMS把相关信息传递给PermissionManagerService处理；

第四步：PermissionManagerService处理结束后回调给---->PKMS中的onPermissionGranted方法把处理结果返回；

第五步：PKMS通知过程中权限变化，并调用writeRuntimePermissionsForUserLPr函数让PackageManager的settings记录下相关授 权信息；

权限申请整体流程图:



MainActiivty:

```
ActivityCompat.requestPermissions(this, new String[]
{Manifest.permission.READ_CONTACTS},
                                  MY_PERMISSIONS_REQUEST_READ_CONTACTS);
```

ActivityCompat.**requestPermissions**:

```
public static void requestPermissions(final @NonNull Activity activity,
            final @NonNull String[] permissions, final @IntRange(from = 0) int
requestCode) {
        if (sDelegate != null
                && sDelegate.requestPermissions(activity, permissions,
requestCode)) {
            // Delegate has handled the permission request.
            return;
        }

        if (Build.VERSION.SDK_INT >= 23) {
            if (activity instanceof RequestPermissionsRequestCodeValidator) {
                ((RequestPermissionsRequestCodeValidator) activity)
                        .validateRequestPermissionsRequestCode(requestCode);
            }
            // 【同学们注意】，重点是看这句代码，就是权限申请， 下面代码就会分析这句代码
            activity.requestPermissions(permissions, requestCode);
        } else if (activity instanceof OnRequestPermissionsResultCallback) {
           ....
        }
    }
```

Activity.**requestPermissions**:

位置：frameworks/base/core/java/android/app/Activity.java

```
public final void requestPermissions(@NonNull String[] permissions, int
requestCode) {
    if (requestCode < 0) {
        throw new IllegalArgumentException("requestCode should be >= 0");
    }
    if (mHasCurrentPermissionsRequest) {
        Log.w(TAG, "Can request only one set of permissions at a time");
        // Dispatch the callback with empty arrays which means a cancellation.
        onRequestPermissionsResult(requestCode, new String[0], new int[0]);
        return;
    }
    // 【同学们注意】 关注 buildRequestPermissionsIntent，下面会分析这个函数
    Intent intent =
getPackageManager().buildRequestPermissionsIntent(permissions);
    startActivityForResult(REQUEST_PERMISSIONS_WHO_PREFIX, intent, requestCode,
null);
    mHasCurrentPermissionsRequest = true;
}
```

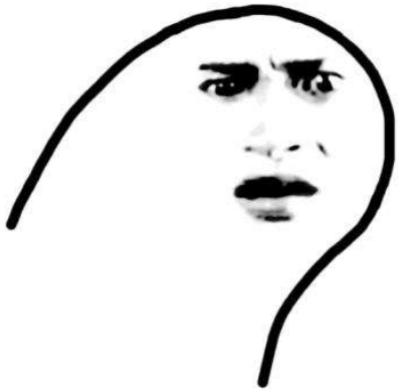
PackageManager.**buildRequestPermissionsIntent**:

位置：frameworks/base/core/java/android/content/pm/PackageManager.java

```
    public Intent buildRequestPermissionsIntent(@NonNull String[] permissions) {
        if (ArrayUtils.isEmpty(permissions)) {
           throw new IllegalArgumentException("permission cannot be null or
empty");
        }
        Intent intent = new Intent(ACTION_REQUEST_PERMISSIONS);
        intent.putExtra(EXTRA_REQUEST_PERMISSIONS_NAMES, permissions);
        intent.setPackage(getPermissionControllerPackageName());
        return intent;
    }
```

总结：这个**buildRequestPermissionsIntent**函数的目的，就是去 **激活某个Activity**，就这么简单，哈哈哈。

啊..... 一脸懵逼，Derry老师你说了什么东西啊？，Derry你在这么调皮把你吊起来打

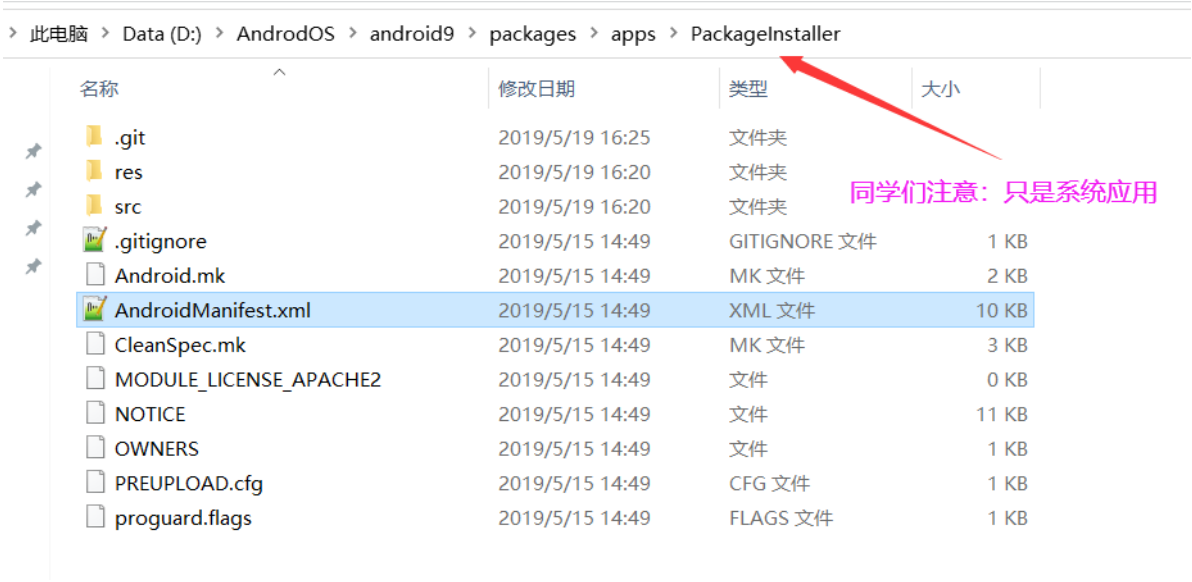


慢慢来分析：

同学们注意：**灵感来自与此** (既然**buildRequestPermissionsIntent**函数是为了拼接一个Intent，那么想都不用想，一定是想搞隐士意图来**激活某个Activity**，这个灵感非常关键，同学们以后遇到Intent的封装，就应该想到是为了什么了吧，是不是为了激活某个Activity)

```
public static final String ACTION_REQUEST_PERMISSIONS =
"android.content.pm.action.REQUEST_PERMISSIONS";
```

那么我们就根据 ”android.content.pm.action.REQUEST_PERMISSIONS“ 表示动作来找到 需要激活的某个Activity不就行了，同学们是不是很简单

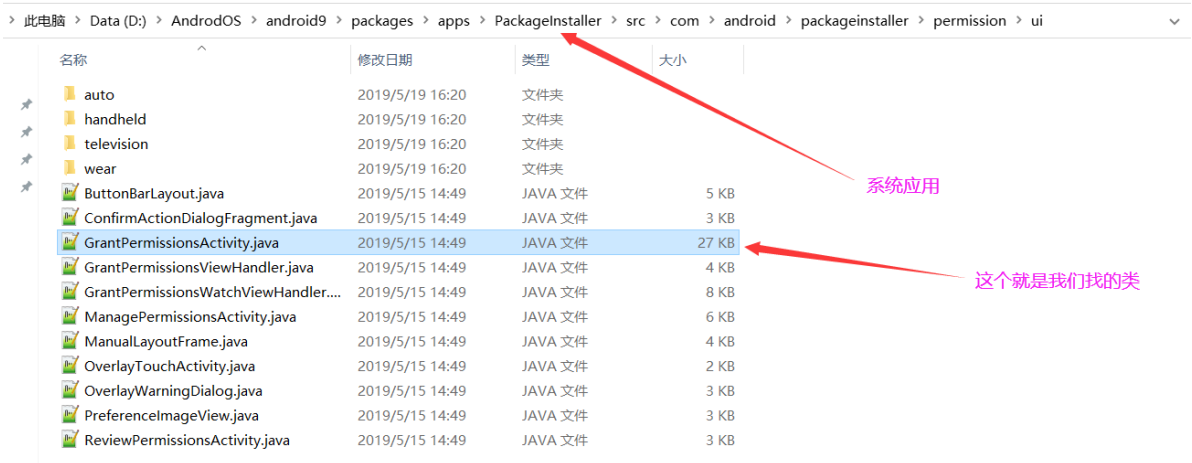


打开AndroidManfiest.xml

```
        同学们注意：下面我们就分析 GrantPermissionsActivity
        <activity android:name=".permission.ui.GrantPermissionsActivity"
                android:configChanges="orientation|keyboardHidden|screenSize"
                android:excludeFromRecents="true"
                android:theme="@style/GrantPermissions"
                android:visibleToInstantApps="true">

                <!-- 那么我们就根据
"android.content.pm.action.REQUEST_PERMISSIONS“ 表示动作来找到
                     需要激活的某个Activity不就行了，同学们是不是很简单 -->
                <intent-filter android:priority="1">
                    <action
android:name="android.content.pm.action.REQUEST_PERMISSIONS" />
                    <category android:name="android.intent.category.DEFAULT" />
                </intent-filter>
        </activity>
```

打开**GrantPermissionsActivity**



onPermissionGrantResult 函数：

同学们注意，原来GrantPermissionsActivity也就是我们常见的权限申请界面，用户可以根据提示选择是否授权给应用相应的权限。用户操作后的结果会通过回调GrantPermissionsActivity的onPermissionGrantResult方法返回。在onPermissionGrantResult方法中会根据返回结果去决定是走授予权限还是撤销权限流程，然后会更新授权结果，最后返回结果并结束自己：


```
    @Override
    public void onPermissionGrantResult(String name, boolean granted, boolean
doNotAskAgain) {
        GroupState groupState = mRequestGrantPermissionGroups.get(name);
        if (groupState.mGroup != null) {
```

```
            if (granted) {
                // 【同学们注意】重点是这个 授予权限， 下面会分析这个函数
grantRuntimePermissions
                // 授予权限
                groupState.mGroup.grantRuntimePermissions(doNotAskAgain,
                        groupState.affectedPermissions);
                groupState.mState = GroupState.STATE_ALLOWED;
            } else {
                // 撤销权限
                groupState.mGroup.revokeRuntimePermissions(doNotAskAgain,
                        groupState.affectedPermissions);
                groupState.mState = GroupState.STATE_DENIED;

                int numRequestedPermissions = mRequestedPermissions.length;
                for (int i = 0; i < numRequestedPermissions; i++) {
                    String permission = mRequestedPermissions[i];

                    if (groupState.mGroup.hasPermission(permission)) {
                        EventLogger.logPermissionDenied(this, permission,
                                mAppPermissions.getPackageInfo().packageName);
                    }
                }
            }
            // 更新授权结果
            updateGrantResults(groupState.mGroup);
        }
        if (!showNextPermissionGroupGrantRequest()) {
            // 返回授权结果并结束自己
            setResultAndFinish();
        }
    }
```

接下来继续跟踪AppPermissionGroup.grantRuntimePermissions方法分析授权流程。
AppPermissionGroup.grantRuntimePermissions方法中会判断targetSdkVersion是否大于LOLLIPOP_MR1（22），如果大于则做动态权限申请处理

位置：
packages/apps/PackageInstaller/src/com/android/packageinstaller/permission/model/AppPermissionGroup.java

Platform Version	API Level	VERSION_CODE
Android 6.0	23	M
Android 5.1	22	LOLLIPOP_MR1
Android 5.0	21	LOLLIPOP
Android 4.4W	20	KITKAT_WATCH
Android 4.4	19	KITKAT
Android 4.3	18	JELLY_BEAN_MR2
Android 4.2, 4.2.2	17	JELLY_BEAN_MR1
Android 4.1, 4.1.1	16	JELLY_BEAN
Android 4.0.3, 4.0.4	15	ICE_CREAM_SANDWICH_MR1
Android 3.2	13	HONEYCOMB_MR2
Android 3.1.x	12	HONEYCOMB_MR1
Android 3.0.x	11	HONEYCOMB

```
    public boolean grantRuntimePermissions(boolean fixedByTheUser, String[]
filterPermissions) {
        final int uid = mPackageInfo.applicationInfo.uid;

        // We toggle permissions only to apps that support runtime
        // permissions, otherwise we toggle the app op corresponding
        // to the permission if the permission is granted to the app.
        for (Permission permission : mPermissions.values()) {
            ...
            if (mAppSupportsRuntimePermissions) {
                // 【同学们注意】 在Android 5.1后，就需要支持动态申请权限啦
                // LOLLIPOP_MR1之后版本，支持动态权限申请
                // Do not touch permissions fixed by the system.
                if (permission.isSystemFixed()) {
                    return false;
                }

                // Ensure the permission app op enabled before the permission
grant.
                if (permission.hasAppOp() && !permission.isAppOpAllowed()) {
                    permission.setAppOpAllowed(true);
                    mAppOps.setUidMode(permission.getAppOp(), uid,
AppOpsManager.MODE_ALLOWED);
                }
```

```
                // Grant the permission if needed.
                if (!permission.isGranted()) {
                    permission.setGranted(true);

                    // 【同学们注意】 这里很关键，通过
mPackageManager.grantRuntimePermission 跨进程到 PKMS
                    // 下面我们就分析这个操作了哦，注意哦
                    // 熟悉Android源码的同学都知道XXXManager只是一个辅助类，其真正提供服
务的都是XXXManagerService，所以                                直接跳转PackageManagerService
中的grantRuntimePermission方法。

 mPackageManager.grantRuntimePermission(mPackageInfo.packageName,
                            permission.getName(), mUserHandle);
                }

                // Update the permission flags.
                if (!fixedByTheUser) {
                    // Now the apps can ask for the permission as the user
                    // no longer has it fixed in a denied state.
                    if (permission.isUserFixed() || permission.isUserSet()) {
                        permission.setUserFixed(false);
                        permission.setUserSet(false);

 mPackageManager.updatePermissionFlags(permission.getName(),
                                mPackageInfo.packageName,
                                PackageManager.FLAG_PERMISSION_USER_FIXED
                                        |
PackageManager.FLAG_PERMISSION_USER_SET,
                                0, mUserHandle);
                    }
                }
            } else {
                // LOLLIPOP_MR1之前版本，不支持动态权限申请
                // Legacy apps cannot have a not granted permission but just in
case.
                ....
            }
        }

        return true;
    }
```

终于到了 **PKMS**了，是不是很开心了：

位置：frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
    @Override
    public void grantRuntimePermission(String packageName, String permName,
final int userId) {
        mPermissionManager.grantRuntimePermission(permName, packageName, false
/*overridePolicy*/,
                getCallingUid(), userId, mPermissionCallback);
    }
```

PermissionManagerInternal---- 接口到实现 ---
PermissionManagerService.**grantRuntimePermission**:

```
    private void grantRuntimePermission(String permName, String packageName,
boolean overridePolicy,
            int callingUid, final int userId, PermissionCallback callback) {
        // 检查用户是否存在
        if (!mUserManagerInt.exists(userId)) {
            Log.e(TAG, "No such user:" + userId);
            return;
        }
        // 检查PackageInstaller是否有动态权限授权权限
        mContext.enforceCallingOrSelfPermission(
                android.Manifest.permission.GRANT_RUNTIME_PERMISSIONS,
                "grantRuntimePermission");
        ...

        // 【同学们注意】 下面会分析这个回调
        // 回调PermissionCallback的onPermissionGranted方法通知授予权限
        if (callback != null) {
            callback.onPermissionGranted(uid, userId);
        }
        ...
    }
```

又回到 **PKMS**，位置:
frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
        // 回调的是PackageManagerService中的PermissionCallback，在其实现的
onPermissionGranted方法中会去通知观察者权                限发生变化，并调用PackageManager的
Settings记录动态权限授权状态。
        @Override
        public void onPermissionGranted(int uid, int userId) {
            mOnPermissionChangeListeners.onPermissionsChanged(uid);

            // Not critical; if this is lost, the application has to request
again.
            synchronized (mPackages) {
                // [同学们注意] 下面会分钟分析这个函数
                mSettings.writeRuntimePermissionsForUserLPr(userId, false);
            }
        }
```

调用流程: mSettings.writeRuntimePermissionsForUserLPr ---> writePermissionsForUserSyncLPr ----> **writePermissionsSync**:

位置: frameworks/base/services/core/java/com/android/server/pm/Settings.java

```
        // 同学们注意:
        // writePermissionsSync方法来完成最后的记录工作。
        // writePermissionsSync方法的代码很长，但是逻辑很清晰，就是先查询与应用相关的所有权限状态，
        // 然后创建 runtime-permissions.xml 文件把这些信息记录进去。
        private void writePermissionsSync(int userId) {
            //  动态权限文件（runtime-permissions.xml）
            AtomicFile destination = new AtomicFile(getUserRuntimePermissionsFile(userId),
                    "package-perms-" + userId);

            ArrayMap<String, List<PermissionState>> permissionsForPackage = new ArrayMap<>();
            ArrayMap<String, List<PermissionState>> permissionsForSharedUser = new ArrayMap<>();

            synchronized (mPersistenceLock) {
                mWriteScheduled.delete(userId);
                // 获得Package权限状态
                final int packageCount = mPackages.size();
                for (int i = 0; i < packageCount; i++) {
                    String packageName = mPackages.keyAt(i);
                    PackageSetting packageSetting = mPackages.valueAt(i);
                    if (packageSetting.sharedUser == null) {
                        PermissionsState permissionsState = packageSetting.getPermissionsState();
                        List<PermissionState> permissionsStates = permissionsState
                                .getRuntimePermissionStates(userId);
                        if (!permissionsStates.isEmpty()) {
                            permissionsForPackage.put(packageName, permissionsStates);
                        }
                    }
                }
                // 获得SharedUser权限状态
                final int sharedUserCount = mSharedUsers.size();
                for (int i = 0; i < sharedUserCount; i++) {
                    String sharedUserName = mSharedUsers.keyAt(i);
                    SharedUserSetting sharedUser = mSharedUsers.valueAt(i);
                    PermissionsState permissionsState = sharedUser.getPermissionsState();
                    List<PermissionState> permissionsStates = permissionsState
                            .getRuntimePermissionStates(userId);
                    if (!permissionsStates.isEmpty()) {
                        permissionsForSharedUser.put(sharedUserName, permissionsStates);
                    }
                }
            }

            FileOutputStream out = null;
```

```
            try {
                out = destination.startWrite();
                // 创建xml文件用于记录权限状态
                XmlSerializer serializer = Xml.newSerializer();
                serializer.setOutput(out, StandardCharsets.UTF_8.name());
                serializer.setFeature(
                        "http://xmlpull.org/v1/doc/features.html#indent-output",
true);
                serializer.startDocument(null, true);

                serializer.startTag(null, TAG_RUNTIME_PERMISSIONS);

                String fingerprint = mFingerprints.get(userId);
                if (fingerprint != null) {
                    serializer.attribute(null, ATTR_FINGERPRINT, fingerprint);
                }
                // 写入Package权限状态
                final int packageCount = permissionsForPackage.size();
                for (int i = 0; i < packageCount; i++) {
                    String packageName = permissionsForPackage.keyAt(i);
                    List<PermissionState> permissionStates =
permissionsForPackage.valueAt(i);
                    serializer.startTag(null, TAG_PACKAGE);
                    serializer.attribute(null, ATTR_NAME, packageName);
                    writePermissions(serializer, permissionStates);
                    serializer.endTag(null, TAG_PACKAGE);
                }
                 // 写入SharedUser权限状态
                final int sharedUserCount = permissionsForSharedUser.size();
                for (int i = 0; i < sharedUserCount; i++) {
                    String packageName = permissionsForSharedUser.keyAt(i);
                    List<PermissionState> permissionStates =
permissionsForSharedUser.valueAt(i);
                    serializer.startTag(null, TAG_SHARED_USER);
                    serializer.attribute(null, ATTR_NAME, packageName);
                    writePermissions(serializer, permissionStates);
                    serializer.endTag(null, TAG_SHARED_USER);
                }

                serializer.endTag(null, TAG_RUNTIME_PERMISSIONS);
                // 写入结束
                serializer.endDocument();
                destination.finishWrite(out);

                if (Build.FINGERPRINT.equals(fingerprint)) {
                    mDefaultPermissionsGranted.put(userId, true);
                }
            // Any error while writing is fatal.
            } catch (Throwable t) {
                Slog.wtf(PackageManagerService.TAG,
                        "Failed to write settings, restoring backup", t);
                destination.failWrite(out);
            } finally {
                IoUtils.closeQuietly(out);
            }
        }
```

权限申请源码流程总结:

第一步：MainActivity 调用 requestPermissions 进行动态权限申请；

第二步：requestPermissions函数通过隐士意图，激活PackageInstaller的GrantPermissionsActivity界面，让用户选择是否授权；

第三步：经过PKMS把相关信息传递给PermissionManagerService处理；

第四步：PermissionManagerService处理结束后回调给---->PKMS中的onPermissionGranted方法把处理结果返回；

第五步：PKMS通知过程中权限变化，并调用writeRuntimePermissionsForUserLPr函数让PackageManager的settings记录下相关授 权信息；

checkPermission：

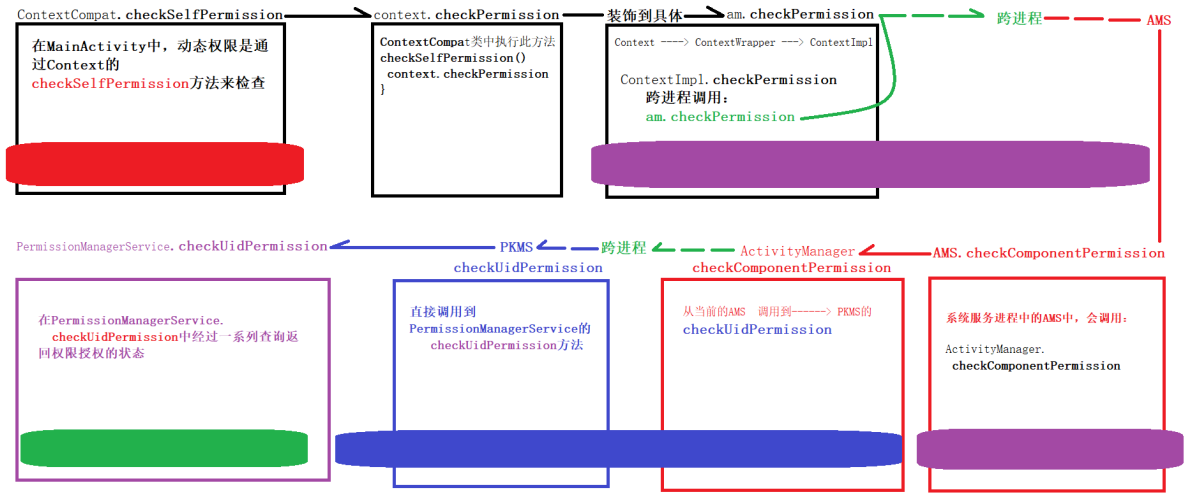
第一步：MainActivity会调用checkSelfPermission方法检测是否具有权限（红色区域）

第二步：通过实现类ContextImpl的checkPermission方法经由ActivityManager和ActivityManagerService处理（紫色区域）

第三步：经过ActivityManager处理后会调用PKMS的checkUidPermission方法把数据传递给PermissionManagerService处理（蓝色）

第四步：在PermissionManagerService.checkUidPermission中经过一系列查询返回权限授权的状态（绿色区域）

整体流程图：



MainActivity：

简单来说：动态权限是通过Context的checkSelfPermission方法来检查的，其实现是在ContextImpl中，直接跳转到相应的方法

```
ContextCompat.checkSelfPermission
```

ContextCompat:

```
    public static int checkSelfPermission(@NonNull Context context, @NonNull
String permission) {
        if (permission == null) {
            throw new IllegalArgumentException("permission is null");
        }

        return context.checkPermission(permission, android.os.Process.myPid(),
Process.myUid());
    }
```

Context（抽象） ----------> **ContextImpl**（实现）.**checkPermission**

位置：frameworks/base/core/java/android/app/ContextImpl.java

```
    @Override
    public int checkPermission(String permission, int pid, int uid) {
        if (permission == null) {
            throw new IllegalArgumentException("permission is null");
        }

       final IActivityManager am = ActivityManager.getService();
       ...
        try {
            // 【同学们注意】 这里要开始跨进程通信啦，从应用进程 ---- Binder ----->系统服
务进程AMS
            // 从ContextImpl.checkPermission方法源码中，我们很快找到了老熟人
ActivityManager的身影，还是老规矩直接跳转到                    提供对应服务的
ActivityManagerService.checkPermission方法中
            return am.checkPermission(permission, pid, uid);
        } catch (RemoteException e) {
            throw e.rethrowFromSystemServer();
        }
    }
```

AMS这边的分析：

位置：frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
    @Override
    public int checkPermission(String permission, int pid, int uid) {
        if (permission == null) {
            return PackageManager.PERMISSION_DENIED;
        }
        return checkComponentPermission(permission, pid, uid, -1, true);
    }

    int checkComponentPermission(String permission, int pid, int uid,
```

```
            int owningUid, boolean exported) {
        if (pid == MY_PID) {
            return PackageManager.PERMISSION_GRANTED;
        }
        // 【同学们注意】下面会重点分析这个方法
        // ActivityManager中的checkComponentPermission方法。
checkComponentPermission方法中先是对一些固定case作了判                断，如果都不满足，最后
会调用PackageManager的checkUidPermission方法来查询授权状态
        return ActivityManager.checkComponentPermission(permission, uid,
                owningUid, exported);
    }
```

ActivityManager.**checkComponentPermission**:

位置：frameworks/base/core/java/android/app/ActivityManager.java

```
    public static int checkComponentPermission(String permission, int uid,
            int owningUid, boolean exported) {
        // Root, system server get to do everything.
        final int appId = UserHandle.getAppId(uid);
        if (appId == Process.ROOT_UID || appId == Process.SYSTEM_UID) {
            return PackageManager.PERMISSION_GRANTED;
        }
        // Isolated processes don't get any permissions.
        if (UserHandle.isIsolated(uid)) {
            return PackageManager.PERMISSION_DENIED;
        }
        // If there is a uid that owns whatever is being accessed, it has
        // blanket access to it regardless of the permissions it requires.
        if (owningUid >= 0 && UserHandle.isSameApp(uid, owningUid)) {
            return PackageManager.PERMISSION_GRANTED;
        }
        // If the target is not exported, then nobody else can get to it.
        if (!exported) {
            /*
            RuntimeException here = new RuntimeException("here");
            here.fillInStackTrace();
            Slog.w(TAG, "Permission denied: checkComponentPermission()
owningUid=" + owningUid,
                    here);
            */
            return PackageManager.PERMISSION_DENIED;
        }
        if (permission == null) {
            return PackageManager.PERMISSION_GRANTED;
        }
        try {
            // 【同学们注意】从这里开始就要，  从当前的AMS   调用到------> PKMS的
checkUidPermission
            return AppGlobals.getPackageManager()
                    .checkUidPermission(permission, uid);
        } catch (RemoteException e) {
            throw e.rethrowFromSystemServer();
        }
    }
```

同学们注意：终于到了，**PKMS了**：

位置：frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java

```
    @Override
    public int checkUidPermission(String permName, int uid) {
        synchronized (mPackages) {
            final String[] packageNames = getPackagesForUid(uid);
            final PackageParser.Package pkg = (packageNames != null &&
packageNames.length > 0)
                    ? mPackages.get(packageNames[0])
                    : null;
            // 【同学们注意】 这里会调用到
PermissionManagerService.checkUidPermission 下面会分析
            // PackageManagerService中的checkUidPermission似乎也没干什么，直接跳转到
PermissionManagerService的                  checkUidPermission方法
            return mPermissionManager.checkUidPermission(permName, pkg, uid,
getCallingUid());
        }
    }
```

PermissionManagerService.**checkUidPermission** 最终在这个方法中会根据各种条件判断返回授权状态。

位置：
frameworks/base/services/core/java/com/android/server/pm/permission/PermissionManagerService.java

```
    private int checkUidPermission(String permName, PackageParser.Package pkg,
int uid,
            int callingUid) {
        final int callingUserId = UserHandle.getUserId(callingUid);
        final boolean isCallerInstantApp =
                mPackageManagerInt.getInstantAppPackageName(callingUid) != null;
        final boolean isUidInstantApp =
                mPackageManagerInt.getInstantAppPackageName(uid) != null;
        final int userId = UserHandle.getUserId(uid);
        if (!mUserManagerInt.exists(userId)) {
            return PackageManager.PERMISSION_DENIED;
        }

        if (pkg != null) {
            if (pkg.mSharedUserId != null) {
                if (isCallerInstantApp) {
                    return PackageManager.PERMISSION_DENIED;
                }
            } else if (mPackageManagerInt.filterAppAccess(pkg, callingUid,
callingUserId)) {
                return PackageManager.PERMISSION_DENIED;
            }
            final PermissionsState permissionsState =
                    ((PackageSetting) pkg.mExtras).getPermissionsState();
            if (permissionsState.hasPermission(permName, userId)) {
```

```
                if (isUidInstantApp) {
                    if (mSettings.isPermissionInstant(permName)) {
                        return PackageManager.PERMISSION_GRANTED;
                    }
                } else {
                    return PackageManager.PERMISSION_GRANTED;
                }
            }
            // Special case: ACCESS_FINE_LOCATION permission includes
ACCESS_COARSE_LOCATION
            if (Manifest.permission.ACCESS_COARSE_LOCATION.equals(permName) &&
permissionsState
                    .hasPermission(Manifest.permission.ACCESS_FINE_LOCATION,
userId)) {
                return PackageManager.PERMISSION_GRANTED;
            }
        } else {
            ArraySet<String> perms = mSystemPermissions.get(uid);
            if (perms != null) {
                if (perms.contains(permName)) {
                    return PackageManager.PERMISSION_GRANTED;
                }
                if (Manifest.permission.ACCESS_COARSE_LOCATION.equals(permName)
&& perms
                        .contains(Manifest.permission.ACCESS_FINE_LOCATION)) {
                    return PackageManager.PERMISSION_GRANTED;
                }
            }
        }
        return PackageManager.PERMISSION_DENIED;
    }
```

自我总结：

第一步：MainActivity会调用checkSelfPermission方法检测是否具有权限（红色区域）

第二步：通过实现类ContextImpl的checkPermission方法经由ActivityManager和ActivityManagerService处理（紫色区域）

第三步：经过ActivityManager处理后会调用PKMS的checkUidPermission方法把数据传递给PermissionManagerService处理（蓝色）

第四步：在PermissionManagerService.checkUidPermission中经过一系列查询返回权限授权的状态（绿色区域）