# TODO

SyntheticJoystickHandler

private void update(MotionEvent event, boolean synthesizeNewKeys) {

final long time = event.getEventTime();

final int metaState = event.getMetaState();

final int deviceId = event.getDeviceId();

final int source = event.getSource();

int xDirection = joystickAxisValueToDirection(

event.getAxisValue(MotionEvent.AXIS\_HAT\_X));

if (xDirection == 0) {

xDirection = joystickAxisValueToDirection(event.getX());

}

if (xDirection == 0 && KPadCoreHelper.getIns().enableJoystickRotation()) {

xDirection = joystickAxisValueToDirection(

event.getAxisValue(MotionEvent.AXIS\_RX));

}

# ActivityThread

ActivityThread是Android Framework中一个非常重要的类，它代表一个应用进程的主线程（对于应用进程来说，ActivityThread的main函数确实是由该进程的主线程执行），其职责就是调度及执行在该线程中运行的四大组件。

**注意**应用进程指那些运行APK的进程，它们由Zyote 派生（fork）而来，上面运行了dalvik虚拟机。与应用进程相对的就是系统进程（包括Zygote和SystemServer）。

另外，读者须将“应用进程和系统进程”与“应用APK和系统APK”的概念区分开来。APK的判别依赖其文件所在位置（如果apk文件在/data/app目录下，则为应用APK）。

## 承上

AtMS.startProcessLocked函数进一步操作

反射方式启动的

1. **private** **final** **void** startProcessLocked(ProcessRecord app,
2. String hostingType, String hostingNameStr) {
3. **try** {
4. **int** uid = app.info.uid;
5. **int**[] gids = **null**;
6. **try** {
7. gids = mContext.getPackageManager().getPackageGids(
8. app.info.packageName);
9. } **catch** (PackageManager.NameNotFoundException e) {
10. }
11. **int** debugFlags = 0;
12. **int** pid = Process.start("android.app.ActivityThread",
13. mSimpleProcessManagement ? app.processName : **null**, uid, uid,
14. gids, debugFlags, **null**);
15. } **catch** (RuntimeException e) {
16. }
17. }

这里主要是调用**Process.start**接口来创建一个新的进程，新的进程会导入android.app.ActivityThread类，并且执行它的main函数，这就是为什么我们前面说每一个应用程序都有一个ActivityThread实例来对应的原因。

## systemMain

**systemMain函数将为SystemServer进程搭建一个和应用进程一样的Android运行环境。**这句话涉及两个概念。

·  进程：来源于操作系统，是在OS中看到的运行体。我们编写的代码一定要运行在一个进程中。

·  Android运行环境：Android努力构筑了一个自己的运行环境。在这个环境中，进程的概念被模糊化了。组件的运行及它们之间的交互均在该环境中实现。

Android运行环境是构建在进程之上的。有Android开发经验的读者可能会发现，在应用程序中，一般只和Android运行环境交互。基于同样的道理，SystemServer希望它内部的那些Service也通过Android运行环境交互，因此也需为它创建一个运行环境。由于SystemServer的特殊性，此处调用了systemMain函数，而普通的应用进程将在主线程中调用ActivityThread的main函数来创建Android运行环境。

另外，ActivityThread虽然本意是代表进程的主线程，但是作为一个Java类，它的实例到底由什么线程创建，恐怕不是ActivityThread自己能做主的，所以在SystemServer中可以发现，ActivityThread对象由其他线程创建，而在应用进程中，ActivityThread将由主线程来创建。

[-->ActivityThread.java::systemMain]

public static final ActivityThread systemMain() {

   HardwareRenderer.disable(true);//禁止硬件渲染加速

   //创建一个ActivityThread对象，其构造函数非常简单

  ActivityThread thread = new ActivityThread();

  thread.attach(true);//调用它的attach函数，注意传递的参数为true

   return thread;

 }

在分析ActivityThread的attach函数之前，先提一个问题供读者思考：前面所说的ActivityThread代表应用进程（其上运行了APK）的主线程，而SystemServer并非一个应用进程，那么为什么此处也需要ActivityThread呢？

·  还记得在PackageManagerService分析中提到的framework-res.apk吗？这个APK除了包含资源文件外，还包含一些Activity（如关机对话框），这些Activity实际上运行在SystemServer进程中[②]。从这个角度看，SystemServer是一个特殊的应用进程。

·  另外，通过ActivityThread可以把Android系统提供的组件之间的交互机制和交互接口（如利用Context提供的API）也拓展到SystemServer中使用。

**提示**解答这个问题，对于理解SystemServer中各服务的交互方式是尤其重要的。

### attach函数分析

[-->ActivityThread.java::attach]

private void attach(boolean system) {

    sThreadLocal.set(this);

    mSystemThread= system;//判断是否为系统进程

    if(!system) {

        ......//应用进程的处理流程

     } else {//系统进程的处理流程，该情况只在SystemServer中处理

       //设置DDMS时看到的systemserver进程名为system\_process

       android.ddm.DdmHandleAppName.setAppName("system\_process");

       try {

            //ActivityThread的几员大将出场，见后文的分析

            mInstrumentation = new Instrumentation();

            ContextImpl context = new ContextImpl();

            //初始化context，注意第一个参数值为getSystemContext

            context.init(getSystemContext().mPackageInfo, null, this);

            Application app = //利用Instrumentation创建一个Application对象

                    Instrumentation.newApplication(Application.class,context);

             //一个进程支持多个Application，mAllApplications用于保存该进程中

            //的Application对象

            mAllApplications.add(app);

             mInitialApplication = app;//设置mInitialApplication

            app.onCreate();//调用Application的onCreate函数

           }......//try/catch结束

      }//if(!system)判断结束

     //注册Configuration变化的回调通知

     ViewRootImpl.addConfigCallback(newComponentCallbacks2() {

          publicvoid onConfigurationChanged(Configuration newConfig) {

            ......//当系统配置发生变化（如语言切换等）时，需要调用该回调

          }

           public void onLowMemory() {}

           public void onTrimMemory(int level) {}

        });

 }

attach函数中出现了几个重要成员，其类型分别是Instrumentation类、Application类及Context类，它们的作用如下

Instrumentaion是一个工具类。当它被启用时，系统先创建它，再通过它来创建其他组件。另外，系统和组件之间的交互也将通过Instrumentation来传递，这样，Instrumentation就能监测系统和这些组件的交互情况了。在实际使用中，我们可以创建Instrumentation的派生类来进行相应的处理。读者可查询Android中Junit的使用来了解Intrstrumentation的作用

Application类保存了一个全局的application状态。Application由AndroidManifest.xml中的<application>标签声明。在实际使用时需定义Application的派生类。此处的Application是Android中的一个概念，可理解为一种容器，它内部包含四大组件。另外，一个进程可以运行多个Application。

Context是一个接口，通过它可以获取并操作Application对应的资源、类，甚至包含于Application中的四大组件。

Context是一个抽象类，而由AMS创建的将是它的子类ContextImpl。如前所述，Context提供了Application的上下文信息，这些信息是如何传递给Context的呢？此问题包括两个方面：

·  Context本身是什么？

·  Context背后所包含的上下文信息又是什么？

## getSystemContext

[-->ActivityThread.java::getSystemContext]

public ContextImpl getSystemContext() {

  synchronized(this) {

   if(mSystemContext == null) {//单例模式

       ContextImplcontext =  ContextImpl.createSystemContext(this);

       //LoadedApk是2.3引入的一个新类，代表一个加载到系统中的APK

       LoadedApkinfo = new LoadedApk(this, "android", context, null,

                       CompatibilityInfo.DEFAULT\_COMPATIBILITY\_INFO);

       //初始化该ContextImpl对象

      context.init(info, null, this);

      //初始化资源信息

      context.getResources().updateConfiguration(

                        getConfiguration(),getDisplayMetricsLocked(

                       CompatibilityInfo.DEFAULT\_COMPATIBILITY\_INFO, false));

       mSystemContext = context;//保存这个特殊的ContextImpl对象

      }

   }

    Return mSystemContext;

}

以上代码无非是先创建一个ContextImpl，然后再将其初始化（调用init函数）。为什么函数名是getSystemContext呢？因为在初始化ContextImp时使用了一个LoadedApk对象。如注释中所说，LoadedApk是Android 2.3引入的一个类，该类用于保存一些和APK相关的信息（如资源文件位置、JNI库位置等）。在getSystemContext函数中初始化ContextImpl的这个LoadedApk所代表的package名为“android”，其实就是framework-res.apk，由于该APK仅供SystemServer使用，所以此处叫getSystemContext。

上面这些类的关系比较复杂，可通过图6-2展示它们之间的关系。

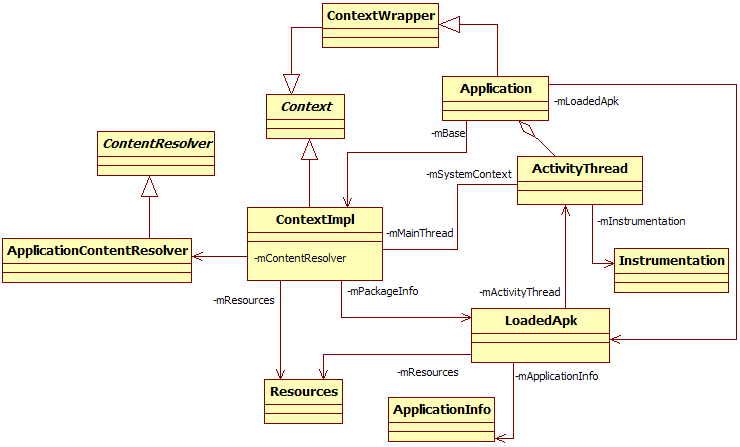


图6-2  ContextImpl和它的“兄弟”们

由图6-2可知：

·  先来看派生关系， ApplicationContentResolver从ConentResolver派生，它主要用于和ContentProvider打交道。ContextImpl和ContextWrapper均从Context继承，而Application则从ContextWrapper派生。

·  从社会关系角度看，ContextImpl交际面最广。它通过mResources指向Resources，mPackageInfo指向LoadedApk，mMainThread指向ActivityThread，mContentResolver指向ApplicationContentResolver。

·  ActivityThread代表主线程，它通过mInstrumentation指向Instrumentation。另外，它还保存多个Application对象。

**注意**在函数中有些成员变量的类型为基类类型，而在图6-2中直接指向了实际类型。

### Context家族图谱

getSystemContext函数分析

该函数在上一节已经见过了。调用该函数后，将得到一个代表系统进程的Context对象。到底什么是Context？先来看如图6-3所示的Context家族图谱。

**注意**该族谱成员并不完全。另外，Activity、Service和Application所实现的接口也未画出。

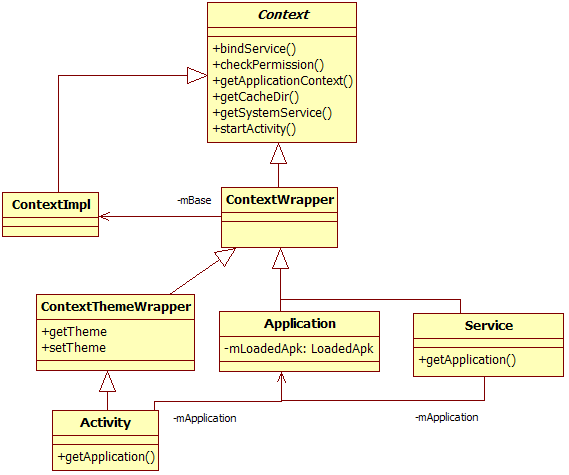


图6-3  Context家族图谱

由图6-3可知：

·  ContextWrapper比较有意思，其在SDK中的说明为“Proxying implementation ofContext that simply delegates all of its calls to another Context. Can besubclassed to modify behavior without changing the original Context.”大概意思是：ContextWrapper是一个代理类，被代理的对象是另外一个Context。在图6-3中，被代理的类其实是ContextImpl，由ContextWrapper通过mBase成员变量指定。读者可查看ContextWrapper.java，其内部函数功能的实现最终都由mBase完成。这样设计的目的是想把ContextImpl隐藏起来。

·  Application从ContextWrapper派生，并实现了ComponentCallbacks2接口。Application中有一个LoadedApk类型的成员变量mLoadedApk。LoadedApk代表一个APK文件。由于一个AndroidManifest.xml文件只能声明一个Application标签，所以一个Application必然会和一个LoadedApk绑定。

·  Service从ContextWrapper派生，其中Service内部成员变量mApplication指向Application（在AndroidManifest.xml中，Service只能作为Application的子标签，所以在代码中Service必然会和一个Application绑定）。

·  ContextThemeWrapper重载了和Theme（主题）相关的两个函数。这些和界面有关，所以Activity作为Android系统中的UI容器，必然也会从ContextThemeWrapper派生。与Service一样，Activity内部也通过mApplication成员变量指向Application。

## ActivityThread.installSystemApplicationInfo函数分析

installSystemApplicationInfo函数的参数为一个ApplicationInfo对象，该对象由AMS通过Context查询PKMS中一个名为“android”的package得来（根据前面介绍的知识，目前只有framework-res.apk声明其package名为“android”）。

再来看installSystemApplicationInfo的代码，如下所示：

[-->ActivityThread.java::installSystemApplicationInfo]

public voidinstallSystemApplicationInfo(ApplicationInfo info) {

 synchronized (this) {

   //返回的ContextImpl对象即之前在AMS的main函数一节中创建的那个对象

   ContextImpl context = getSystemContext();

    //又调用init初始化该Context，是不是重复调用init了？

   context.init(new LoadedApk(this, "android", context, info,

              CompatibilityInfo.DEFAULT\_COMPATIBILITY\_INFO), null, this);

     //创建一个Profiler对象，用于性能统计

     mProfiler = new Profiler();

     }

 }

在以上代码中看到调用context.init的地方，读者可能会有疑惑，getSystemContext函数将返回mSystemContext，而此mSystemContext在AMS的main函数中已经初始化过了，此处为何再次初始化呢？

仔细查看看代码便会发现：

·  第一次执行init时，在LoadedApk构造函数中第四个表示ApplicationInfo的参数为null。

·  第二次执行init时，LoadedApk构造函数的第四个参数不为空，即该参数将真正指向一个实际的ApplicationInfo，该ApplicationInfo来源于framework-res.apk。

基于上面的信息，某些读者可能马上能想到：Context第一次执行init的目的仅仅是为了创建一个Android运行环境，而该Context并没有和实际的ApplicationInfo绑定。而第二次执行init前，先利用Context和PKMS交互得到一个实际的ApplicationInfo，然后再通过init将此Context和ApplicationInfo绑定。

是否觉得前面的疑惑已豁然而解？且慢，此处又抛出了一个更难的问题：

第一次执行init后得到的Context虽然没有绑定ApplicationInfo，不是也能使用吗？此处为何非要和一个ApplicationInfo绑定？

答案很简单，因为framework-res.apk（包括后面将介绍的SettingsProvider.apk）运行在SystemServer中。和其他所有apk一样，它的运行需要一个正确初始化的Android运行环境。

长嘘一口气，这个大难题终于弄明白了！在此即基础上，AMS下一步的工作就就顺理成章了。

由于framework-res.apk是一个APK文件，和其他APK文件一样，它应该运行在一个进程中。而AMS是专门用于进程管理和调度的，所以运行APK的进程应该在AMS中有对应的管理结构。因此AMS下一步工作就是将这个运行环境和一个进程管理结构对应起来并交由AMS统一管理。

## ActivityThread.main

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** attach(**boolean** system) {
3. mSystemThread = system;
4. **if** (!system) {
5. IActivityManager mgr = ActivityManagerNative.getDefault();
6. **try** {
7. mgr.attachApplication(mAppThread);
8. } **catch** (RemoteException ex) {
9. }
10. } **else** {
11. }
12. }
13. **public** **static** **final** **void** main(String[] args) {
14. ActivityThread thread = **new** ActivityThread();
15. thread.attach(**false**);
16. Looper.loop();
17. thread.detach();
18. }
19. }

这个函数在进程中创建一个ActivityThread实例，然后调用它的attach函数，接着就进入消息循环了，直到最后进程退出。

函数attach最终调用了AtMS的远程接口ActivityManagerProxy的attachApplication函数，传入的参数是mAppThread，这是一个ApplicationThread类型的Binder对象，它的作用是用来进行进程间通信的

# Android Context

## 概述

接触过Android的小伙伴, 一定不会对Context感到陌生, 有大量的场景使用都离不开Context, 下面列举部分常见场景:

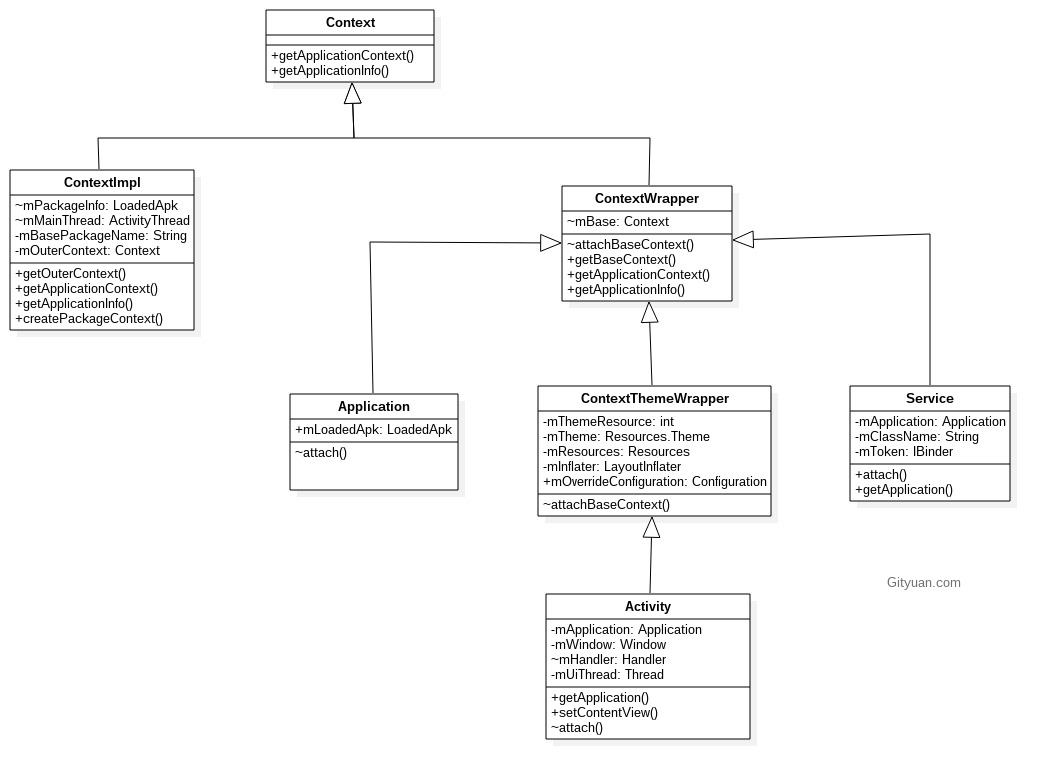
* 启动Activity (startActivity)
* 启动服务 (startService)
* 发送广播 (sendBroadcast), 注册广播接收者 (registerReceiver)
* 获取ContentResolver (getContentResolver)
* 获取类加载器 (getClassLoader)
* 打开或创建数据库 (openOrCreateDatabase)
* 获取资源 (getResources)
* …

四大组件,各种资源操作以及其他很多场景都离不开Context, 那么Context到底是何方神圣呢? 中文意思为上下文, 顾名思义就是在某一个场景中本身所包含的一些潜在信息. 举个例子来说明, 比如当下你正在看Gityuan博客 作为一个Context, 那么这个上下文就会隐藏 博客作者, 博客网址, 博客目录等信息, 其中通过Contet.getAuthor()就能返回”Gityuan”. 这就是上下文, 某一个场景背后所隐藏的信息.

### 类关系图

回到主题, Android Context本身是一个抽象类. ContextImpl, Activity, Service, Application这些都是Context的直接或间接子类, 下面通过看看这些类的关系,如下:

[点击查看大图](http://www.gityuan.com/images/context/context.jpg)



图解:

1. ContextImpl:
   * Application/Activity/Service通过attach() 调用父类ContextWrapper的attachBaseContext(), 从而设置父类成员变量mBase为ContextImpl对象;
   * ContextWrapper的核心工作都是交给mBase(即ContextImpl)来完成;
2. Application: 四大组件属于某一Application, 获取所在Application:
   * Activity/Service: 是通过调用其方法getApplication(),可主动获取当前所在mApplication;
     + mApplication是由LoadedApk.makeApplication()过程所初始化的;
   * Receiver: 是通过其方法onReceive()的第一个参数指向通当前所在Application,也就是只有接收到广播的时候才能拿到当前的Application对象;
   * provider: 目前没有提供直接获取当前所在Application的方法, 但可通过getContext()可以获取当前的ContextImpl.

## 组件初始化

要理解Context, 需要依次来看看四大组件的初始化过程.

### 2.1 performLaunchActivity

[-> ActivityThread.java]

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {

...

ActivityInfo aInfo = r.activityInfo;

if (r.packageInfo == null) {

//step 1: 创建LoadedApk对象

r.packageInfo = getPackageInfo(aInfo.applicationInfo, r.compatInfo,

Context.CONTEXT\_INCLUDE\_CODE);

}

... //component初始化过程

java.lang.ClassLoader cl = r.packageInfo.getClassLoader();

//step 2: 创建Activity对象

Activity activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);

...

//step 3: 创建Application对象

Application app = r.packageInfo.makeApplication(false, mInstrumentation);

if (activity != null) {

//step 4: 创建ContextImpl对象

Context appContext = createBaseContextForActivity(r, activity);

CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());

Configuration config = new Configuration(mCompatConfiguration);

//step5: 将Application/ContextImpl都attach到Activity对象 [见小节4.1]

activity.attach(appContext, this, getInstrumentation(), r.token,

r.ident, app, r.intent, r.activityInfo, title, r.parent,

r.embeddedID, r.lastNonConfigurationInstances, config,

r.referrer, r.voiceInteractor);

...

int theme = r.activityInfo.getThemeResource();

if (theme != 0) {

activity.setTheme(theme);

}

activity.mCalled = false;

if (r.isPersistable()) {

//step 6: 执行回调onCreate

mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);

} else {

mInstrumentation.callActivityOnCreate(activity, r.state);

}

r.activity = activity;

r.stopped = true;

if (!r.activity.mFinished) {

activity.performStart(); //执行回调onStart

r.stopped = false;

}

if (!r.activity.mFinished) {

//执行回调onRestoreInstanceState

if (r.isPersistable()) {

if (r.state != null || r.persistentState != null) {

mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state,

r.persistentState);

}

} else if (r.state != null) {

mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);

}

}

...

r.paused = true;

mActivities.put(r.token, r);

}

return activity;

}

startActivity的过程最终会在目标进程执行performLaunchActivity()方法, 该方法主要功能:

1. 创建对象LoadedApk;
2. 创建对象Activity;
3. 创建对象Application;
4. 创建对象ContextImpl;
5. Application/ContextImpl都attach到Activity对象;
6. 执行onCreate()等回调;

### 2.2 handleCreateService

[-> ActivityThread.java]

private void handleCreateService(CreateServiceData data) {

...

//step 1: 创建LoadedApk

LoadedApk packageInfo = getPackageInfoNoCheck(

data.info.applicationInfo, data.compatInfo);

java.lang.ClassLoader cl = packageInfo.getClassLoader();

//step 2: 创建Service对象

service = (Service) cl.loadClass(data.info.name).newInstance();

//step 3: 创建ContextImpl对象

ContextImpl context = ContextImpl.createAppContext(this, packageInfo);

context.setOuterContext(service);

//step 4: 创建Application对象

Application app = packageInfo.makeApplication(false, mInstrumentation);

//step 5: 将Application/ContextImpl都attach到Activity对象 [见小节4.2]

service.attach(context, this, data.info.name, data.token, app,

ActivityManagerNative.getDefault());

//step 6: 执行onCreate回调

service.onCreate();

mServices.put(data.token, service);

ActivityManagerNative.getDefault().serviceDoneExecuting(

data.token, SERVICE\_DONE\_EXECUTING\_ANON, 0, 0);

...

}

整个过程:

1. 创建对象LoadedApk;
2. 创建对象Service;
3. 创建对象ContextImpl;
4. 创建对象Application;
5. Application/ContextImpl分别attach到Service对象;
6. 执行onCreate()回调;

### 2.3 handleReceiver

[-> ActivityThread.java]

***private void handleReceiver(ReceiverData data) {***

***...***

***String component = data.intent.getComponent().getClassName();***

***//step 1: 创建LoadedApk对象***

***LoadedApk packageInfo = getPackageInfoNoCheck(***

***data.info.applicationInfo, data.compatInfo);***

***IActivityManager mgr = ActivityManagerNative.getDefault();***

***java.lang.ClassLoader cl = packageInfo.getClassLoader();***

***data.intent.setExtrasClassLoader(cl);***

***data.intent.prepareToEnterProcess();***

***data.setExtrasClassLoader(cl);***

***//step 2: 创建BroadcastReceiver对象***

***BroadcastReceiver receiver = (BroadcastReceiver)cl.loadClass(component).newInstance();***

***//step 3: 创建Application对象***

***Application app = packageInfo.makeApplication(false, mInstrumentation);***

***//step 4: 创建ContextImpl对象***

***ContextImpl context = (ContextImpl)app.getBaseContext();***

***sCurrentBroadcastIntent.set(data.intent);***

***receiver.setPendingResult(data);***

***//step 5: 执行onReceive回调 [见小节4.3]***

***receiver.onReceive(context.getReceiverRestrictedContext(), data.intent);***

***...***

***}***

整个过程:

1. 创建对象LoadedApk;
2. 创建对象BroadcastReceiver;
3. 创建对象Application;
4. 创建对象ContextImpl;
5. 执行onReceive()回调;

说明:

* 以上过程是静态广播接收者, 即通过AndroidManifest.xml的标签来申明的BroadcastReceiver;
* 如果是动态广播接收者,则不需要再创建那么多对象, 因为动态广播的注册时进程已创建, 基本对象已创建完成. 那么只需要回调BroadcastReceiver的onReceive()方法即可.

### 2.4 installProvider

[-> ActivityThread.java]

private IActivityManager.ContentProviderHolder installProvider(Context context, IActivityManager.ContentProviderHolder holder, ProviderInfo info, boolean noisy, boolean noReleaseNeeded, boolean stable) {

ContentProvider localProvider = null;

IContentProvider provider;

if (holder == null || holder.provider == null) {

Context c = null;

ApplicationInfo ai = info.applicationInfo;

if (context.getPackageName().equals(ai.packageName)) {

c = context;

} else if (mInitialApplication != null &&

mInitialApplication.getPackageName().equals(ai.packageName)) {

c = mInitialApplication;

} else {

//step 1 && 2: 创建LoadedApk和ContextImpl对象

c = context.createPackageContext(ai.packageName,Context.CONTEXT\_INCLUDE\_CODE);

}

final java.lang.ClassLoader cl = c.getClassLoader();

//step 3: 创建ContentProvider对象

localProvider = (ContentProvider)cl.loadClass(info.name).newInstance();

provider = localProvider.getIContentProvider();

//step 4: ContextImpl都attach到ContentProvider对象 [见小节4.4]

//step 5: 并执行回调onCreate

localProvider.attachInfo(c, info);

} else {

...

}

...

return retHolder;

}

该方法主要功能:

1. 创建对象LoadedApk;
2. 创建对象ContextImpl;
3. 创建对象ContentProvider;
4. ContextImpl都attach到ContentProvider对象;
5. 执行onCreate回调;

### 2.5 handleBindApplication

[-> ActivityThread.java]

private void handleBindApplication(AppBindData data) {

//step 1: 创建LoadedApk对象

data.info = getPackageInfoNoCheck(data.appInfo, data.compatInfo);

...

//step 2: 创建ContextImpl对象;

final ContextImpl appContext = ContextImpl.createAppContext(this, data.info);

//step 3: 创建Instrumentation

mInstrumentation = new Instrumentation();

//step 4: 创建Application对象; [见小节3.2.1]

Application app = data.info.makeApplication(data.restrictedBackupMode, null);

mInitialApplication = app;

//step 5: 安装providers

List<ProviderInfo> providers = data.providers;

installContentProviders(app, providers);

//step 6: 执行Application.Create回调

mInstrumentation.callApplicationOnCreate(app);

该过程主要功能:

1. 创建对象LoadedApk
2. 创建对象ContextImpl;
3. 创建对象Instrumentation;
4. 创建对象Application;
5. 安装providers;
6. 执行Create回调;

## 核心对象

上面介绍了4大组件以及Application的初始化过程, 接下来再进一步说明其中LoadedApk, ContextImpl, Application的初始化过程.

### 3.1 创建LoadedApk

#### 3.1.1 AT.getPackageInfo

[-> ActivityThread.java]

public final LoadedApk getPackageInfo(ApplicationInfo ai, CompatibilityInfo compatInfo, int flags) {

boolean includeCode = (flags&Context.CONTEXT\_INCLUDE\_CODE) != 0;

//是否违反隐私问题

boolean securityViolation = includeCode && ai.uid != 0

&& ai.uid != Process.SYSTEM\_UID && (mBoundApplication != null

? !UserHandle.isSameApp(ai.uid, mBoundApplication.appInfo.uid)

: true);

boolean registerPackage = includeCode && (flags&Context.CONTEXT\_REGISTER\_PACKAGE) != 0;

...

return getPackageInfo(ai, compatInfo, null, securityViolation, includeCode,

registerPackage);

}

当securityViolation=true,则代表违反隐私问题, 会抛出SecurityException异常.

#### 3.1.2 AT.getPackageInfo

[-> ActivityThread.java]

***private LoadedApk getPackageInfo(ApplicationInfo aInfo, CompatibilityInfo compatInfo, ClassLoader baseLoader, boolean securityViolation, boolean includeCode, boolean registerPackage) {***

***final boolean differentUser = (UserHandle.myUserId() != UserHandle.getUserId(aInfo.uid));***

***synchronized (mResourcesManager) {***

***WeakReference<LoadedApk> ref;***

***if (differentUser) {***

***ref = null;***

***} else if (includeCode) {***

***ref = mPackages.get(aInfo.packageName); //从mPackages查询***

***} else {***

***...***

***}***

***LoadedApk packageInfo = ref != null ? ref.get() : null;***

***if (packageInfo == null || (packageInfo.mResources != null***

***&& !packageInfo.mResources.getAssets().isUpToDate())) {***

***//创建LoadedApk对象, 此时baseLoader为null***

***packageInfo = new LoadedApk(this, aInfo, compatInfo, baseLoader,***

***securityViolation, includeCode &&***

***(aInfo.flags&ApplicationInfo.FLAG\_HAS\_CODE) != 0, registerPackage);***

***...***

***if (differentUser) {***

***...***

***} else if (includeCode) {***

***//将新创建的LoadedApk加入到mPackages***

***mPackages.put(aInfo.packageName, new WeakReference<LoadedApk>(packageInfo));***

***} else {***

***...***

***}***

***}***

***return packageInfo;***

***}***

***}***

该方法主要功能:

* mPackages的数据类型为ArrayMap<String, WeakReference>,记录着每一个包名所对应的LoadedApk对象的弱引用;
* 当mPackages没有找到相应的LoadedApk对象, 则创建该对象并加入到mPackages.

#### 3.1.3 AT.getPackageInfoNoCheck

public final LoadedApk getPackageInfoNoCheck(ApplicationInfo ai, CompatibilityInfo compatInfo) {

return getPackageInfo(ai, compatInfo, null, false, true, false);

}

除了Activity的初始化, 其他组件初始化都是采用该方法,有默认参数值, 主要功能还是一致的.

* securityViolation=false,则不进行是否违反隐私的监测;
* registerPackage=false, 则在获取类加载器(getClassLoader)时,不会将该package添加到当前所在进程的成员变量pkgDeps.

### 3.2 创建Application

有了LoadedApk对象, 接下来可以创建Application对象, 该对象一个Apk只会创建一次.

#### 3.2.1 LoadedApk.makeApplication

[-> LoadedApk.java]

public Application makeApplication(boolean forceDefaultAppClass, Instrumentation instrumentation) {

//保证一个LoadedApk对象只创建一个对应的Application对象

if (mApplication != null) {

return mApplication;

}

String appClass = mApplicationInfo.className;

if (forceDefaultAppClass || (appClass == null)) {

appClass = "android.app.Application"; //设置应用类名

}

java.lang.ClassLoader cl = getClassLoader();

if (!mPackageName.equals("android")) {

initializeJavaContextClassLoader();

}

//创建ContextImpl对象

ContextImpl appContext = ContextImpl.createAppContext(mActivityThread, this);

//创建Application对象, 并将appContext attach到新创建的Application[见3.2.2]

Application app = mActivityThread.mInstrumentation.newApplication(cl, appClass, appContext);

appContext.setOuterContext(app);

...

mActivityThread.mAllApplications.add(app);

mApplication = app; //将刚创建的app赋值给mApplication

...

return app;

}

该方法主要功能:

1. 获取当前应用的ClassLoader对象,根据是否为”android”来决定调用initializeJavaContextClassLoader()的过程;
2. 根据当前ActivityThread对象来创建相应的ContextImpl对象
3. 创建Application对象, 初始化其成员变量:
   * mBase指向新创建ContextImpl;
   * mLoadedApk指向当前所在的LoadedApk对象;
4. 将新创建的Application对象保存到ContextImpl的成员变量mOuterContext.

关于initializeJavaContextClassLoader()的过程, 见文章[理解Application初始化](http://gityuan.com/2017/04/02/android-application/)的[小节2.9].

关于应用类名采用的是Apk中声明的应用类名,即Manifest.xml中定义的类名. 有两种特殊情况会强制 设置应用类名为”android.app.Application”:

* 当forceDefaultAppClass=true, 目前只有system\_server进程初始化包名为”android”的apk过程会调用;
* Apk没有自定义应用类名的情况.

#### newApplication

[-> Instrumentation.java]

static public Application newApplication(Class<?> clazz, Context context) throws InstantiationException, IllegalAccessException, ClassNotFoundException {

Application app = (Application)clazz.newInstance(); //创建Application

app.attach(context); //执行attach操作[见小节4.3.5]

return app;

}

### 3.3 创建ContextImpl

创建ContextImpl的方式有多种, 不同的组件初始化调用不同的方法,如下:

* Activity: 调用createBaseContextForActivity初始化;
* Service/Application: 调用createAppContext初始化;
* Provider: 调用createPackageContext初始化;
* BroadcastReceiver: 直接从Application.getBaseContext()来获取ContextImpl对象;

#### 3.3.1 createBaseContextForActivity

[-> ActivityThread.java]

private Context createBaseContextForActivity(ActivityClientRecord r, final Activity activity) {

int displayId = Display.DEFAULT\_DISPLAY;

try {

displayId = ActivityManagerNative.getDefault().getActivityDisplayId(r.token);

} catch (RemoteException e) {

}

//创建ContextImpl对象

ContextImpl appContext = ContextImpl.createActivityContext(

this, r.packageInfo, displayId, r.overrideConfig);

appContext.setOuterContext(activity);

Context baseContext = appContext;

...

return baseContext;

}

[-> ContextImpl.java]

static ContextImpl createActivityContext(ActivityThread mainThread, LoadedApk packageInfo, int displayId, Configuration overrideConfiguration) {

return new ContextImpl(null, mainThread, packageInfo,

null, null, false,

null, overrideConfiguration, displayId);

}

Activity采用该方法来初始化ContextImpl对象.

#### 3.3.2 createAppContext

[-> ContextImpl.java]

static ContextImpl createAppContext(ActivityThread mainThread, LoadedApk packageInfo) {

if (packageInfo == null) throw new IllegalArgumentException("packageInfo");

return new ContextImpl(null, mainThread, packageInfo,

null, null, false,

null, null, Display.INVALID\_DISPLAY);

}

Service/Application采用该方法来初始化ContextImpl对象.

#### 3.3.3 createPackageContext

[-> ContextImpl.java]

public Context createPackageContext(String packageName, int flags) throws NameNotFoundException {

return createPackageContextAsUser(packageName, flags,

mUser != null ? mUser : Process.myUserHandle());

}

public Context createPackageContextAsUser(String packageName, int flags, UserHandle user) throws NameNotFoundException {

final boolean restricted = (flags & CONTEXT\_RESTRICTED) == CONTEXT\_RESTRICTED;

if (packageName.equals("system") || packageName.equals("android")) {

return new ContextImpl(this, mMainThread, mPackageInfo, mActivityToken,

user, restricted, mDisplay, null, Display.INVALID\_DISPLAY);

}

//创建LoadedApk

LoadedApk pi = mMainThread.getPackageInfo(packageName, mResources.getCompatibilityInfo(),

flags | CONTEXT\_REGISTER\_PACKAGE, user.getIdentifier());

if (pi != null) {

//创建ContextImpl

ContextImpl c = new ContextImpl(this, mMainThread, pi,

mActivityToken,user, restricted,

mDisplay, null, Display.INVALID\_DISPLAY);

if (c.mResources != null) {

return c;

}

}

}

provider采用该方法来初始化ContextImpl对象.

#### 3.3.4 ContextImpl初始化

[-> ContextImpl.java]

class ContextImpl extends Context {

final ActivityThread mMainThread;

final LoadedApk mPackageInfo;

private final IBinder mActivityToken;

private final String mBasePackageName;

private Context mOuterContext;

//缓存Binder服务

final Object[] mServiceCache = SystemServiceRegistry.createServiceCache();

private ContextImpl(ContextImpl container, ActivityThread mainThread, LoadedApk packageInfo, IBinder activityToken, UserHandle user, boolean restricted, Display display, Configuration overrideConfiguration, int createDisplayWithId) {

mOuterContext = this; //ContextImpl对象

mMainThread = mainThread; // ActivityThread赋值

mPackageInfo = packageInfo; // LoadedApk赋值

mBasePackageName = packageInfo.mPackageName; //mBasePackageName等于“android”

...

}

}

## Context attach过程

### 4.1 Activity

[-> Activity.java]

final void attach(Context context, ActivityThread aThread, Instrumentation instr, IBinder token, int ident, Application application, Intent intent, ActivityInfo info, CharSequence title, Activity parent, String id, NonConfigurationInstances lastNonConfigurationInstances, Configuration config, String referrer, IVoiceInteractor voiceInteractor) {

attachBaseContext(context); //调用父类方法设置mBase.

mUiThread = Thread.currentThread();

mMainThread = aThread;

mApplication = application;

mIntent = intent;

mComponent = intent.getComponent();

mActivityInfo = info;

...

}

将新创建的ContextImpl赋值到父类ContextWrapper.mBase变量.

### 4.2 Service

[-> Service.java]

public final void attach( Context context, ActivityThread thread, String className, IBinder token, Application application, Object activityManager) {

attachBaseContext(context); //调用父类方法设置mBase.

mClassName = className;

mToken = token;

mApplication = application;

...

}

将新创建的ContextImpl赋值到父类ContextWrapper.mBase变量.

### 4.3 BroadcastReceiver

[-> ContextImpl.java]

final Context getReceiverRestrictedContext() {

if (mReceiverRestrictedContext != null) {

return mReceiverRestrictedContext;

}

return mReceiverRestrictedContext = new ReceiverRestrictedContext(getOuterContext());

}

对于广播来说Context的传递过程, 跟其他组件完全不同. 广播是在onCreate过程通过参数将ReceiverRestrictedContext传递过去的. 此处getOuterContext()返回的是ContextImpl对象.

### 4.4 ContentProvider

[-> ContentProvider.java]

public void attachInfo(Context context, ProviderInfo info) {

attachInfo(context, info, false);

}

private void attachInfo(Context context, ProviderInfo info, boolean testing) {

mNoPerms = testing;

if (mContext == null) {

//将新创建ContextImpl对象保存到ContentProvider对象的成员变量mContext

mContext = context;

...

if (info != null) {

setReadPermission(info.readPermission);

setWritePermission(info.writePermission);

setPathPermissions(info.pathPermissions);

mExported = info.exported;

mSingleUser = (info.flags & ProviderInfo.FLAG\_SINGLE\_USER) != 0;

setAuthorities(info.authority);

}

// 执行onCreate回调;

ContentProvider.this.onCreate();

}

}

该方法主要功能:

* 将新创建ContextImpl对象保存到ContentProvider对象的成员变量mContext;
  + 可通过getContext()获取该ContextImpl;
* 执行onCreate回调;

### 4.5 Application

[-> Application.java]

final void attach(Context context) {

attachBaseContext(context); //Application的mBase

mLoadedApk = ContextImpl.getImpl(context).mPackageInfo;

}

该方法主要功能:

1. 将新创建的ContextImpl对象保存到Application的父类成员变量mBase;
2. 将当前所在的LoadedApk对象保存到Application的父员变量mLoadedApk;

### 4.6 Context核心方法

再来说说Context相关的几个核心方法:

| **对象** | **方法** | **返回值类型** | **含义** |
| --- | --- | --- | --- |
| Activity | getApplication() | Application | 获取Activity所属的mApplication |
| Service | getApplication() | Application | 获取Service所属的mApplication |
| ContextWrapper | getBaseContext | ContextImpl | 获取mBase,即ContextImpl |
| ContextWrapper | getApplicationContext | Application | 见小节4.6.1 |
| ContextImpl | getApplicationContext | Application | 见小节4.6.1 |
| ContextImpl | getOuterContext | ContextImpl | 获取mOuterContext |
| ContextImpl | getApplicationInfo | ApplicationInfo | mPackageInfo.mApplicationInfo |

**关于Application:**

* Activity的mApplication是由 [小节3.2.1]makeApplication() 过程创建, 由 [小节4.1]赋值;
* Service的mApplication是由 [小节3.2.1]makeApplication() 过程创建, 由 [小节4.2]赋值;
* Receiver是通过其方法onReceive()的第一个参数指向当前所在Application;
* provider无法获取application,因为其所在application不一定初始化;

**关于mOuterContext:** ContextImpl的mOuterContext,默认值是由[小节3.3.4]ContextImpl初始化过程创建. 但往往通过调用setOuterContext()使其指向外部的Context;

* makeApplication过程, mOuterContext指向Application;
* handleCreateService()过程, mOuterContext指向Service;
* performLaunchActivity的createBaseContextForActivity过程, mOuterContext指向Activity;
* BroadcastReceiver/Provider则采用默认值ContextImpl;

#### 4.6.1 CI.getApplicationContext

class ContextImpl extends Context {

public Context getApplicationContext() {

return (mPackageInfo != null) ?

mPackageInfo.getApplication() : mMainThread.getApplication();

}

}

//上述mPackageInfo的数据类型为LoadedApk

public final class LoadedApk {

Application getApplication() {

return mApplication;

}

}

//上述mMainThread为ActivityThread

public final class ActivityThread {

public Application getApplication() {

return mInitialApplication;

}

}

1. mPackageInfo.getApplication(): 返回的是LoadedApk.mApplication
   * Activity/Servie/BroadcastReceiver/Application初始化, 调用[小节3.2.1]makeApplication()完成; 但对于同一个apk只会执行一次;
2. mMainThread.getApplication(): 返回的是ActivityThread.mInitialApplication
   * ActivityThread.handleBindApplication()赋值;
   * system\_server进程的AT.attach()赋值;

## 总结

### 5.1 组件初始化

下面用一幅图来看看核心组件的初始化过程会创建哪些对象:

| **类型** | **LoadedApk** | **ContextImpl** | **Application** | **创建相应对象** | **回调方法** |
| --- | --- | --- | --- | --- | --- |
| Activity | √ | √ | √ | Activity | onCreate |
| Service | √ | √ | √ | Service | onCreate |
| Receiver | √ | √ | √ | BroadcastReceiver | onReceive |
| Provider | √ | √ | × | ContentProvider | onCreate |
| Application | √ | √ | √ | Application | onCreate |

每个Apk都对应唯一的application对象和LoadedApk对象, 当Apk中任意组件的创建过程中, 当其所对应的的LoadedApk和Application没有初始化则会创建, 且只会创建一次.

另外大家会注意到唯有Provider在初始化过程并不会去创建所相应的Application对象.也就意味着当有多个Apk运行在同一个进程的情况下, 第二个apk通过Provider初始化过程再调用getContext().getApplicationContext()返回的并非Application对象, 而是NULL. 这里要注意会抛出空指针异常.

### 5.2 Context attach过程

1. Application:
   * 调用attachBaseContext()将新创建ContextImpl赋值到父类ContextWrapper.mBase变量;
   * 可通过getBaseContext()获取该ContextImpl;
2. Activity/Service:
   * 调用attachBaseContext() 将新创建ContextImpl赋值到父类ContextWrapper.mBase变量;
   * 可通过getBaseContext()获取该ContextImpl;
   * 可通过getApplication()获取其所在的Application对象;
3. ContentProvider:
   * 调用attachInfo()将新创建ContextImpl保存到ContentProvider.mContext变量;
   * 可通过getContext()获取该ContextImpl;
4. BroadcastReceiver:
   * 在onCreate过程通过参数将ReceiverRestrictedContext传递过去的.
5. ContextImpl:
   * 可通过getApplicationContext()获取Application;

### 5.3 Context使用场景

| **类型** | **startActivity** | **startService** | **bindService** | **sendBroadcast** | **registerReceiver** | **getContentResolver** |
| --- | --- | --- | --- | --- | --- | --- |
| Activity | √ | √ | √ | √ | √ | √ |
| Service | - | √ | √ | √ | √ | √ |
| Receiver | - | √ | × | √ | - | √ |
| Provider | - | √ | √ | √ | √ | √ |
| Application | - | √ | √ | √ | √ | √ |

说明: (图中第一列代表不同的Context, √代表允许在该Context执行相应的操作; ×代表不允许; -代表分情况讨论)

1. 当Context为Receiver的情况下:
   * 不允许执行bindService()操作, 由于限制性上下文(ReceiverRestrictedContext)所决定的,会直接抛出异常.
   * registerReceiver是否允许取决于receiver;
     + 当receiver == null用于获取sticky广播, 允许使用;
     + 否则不允许使用registerReceiver;
2. 纵向来看startActivity操作
   * 当为Activity Context则可直接使用;
   * 当为其他Context, 则必须带上FLAG\_ACTIVITY\_NEW\_TASK flags才能使用;
   * 另外UI相关要Activity中使用.
3. 除了以上情况, 其他的操作都是被允许执行.

### 5.4 getApplicationContext

绝大多数情况下, getApplication()和getApplicationContext()这两个方法完全一致, 返回值也相同; 那么两者到底有什么区别呢? 真正理解这个问题的人非常少. 接下来彻底地回答下这个问题:

getApplicationContext()这个的存在是Android历史原因. 我们都知道getApplication()只存在于Activity和Service对象; 那么对于BroadcastReceiver和ContentProvider却无法获取Application, 这时就需要一个能在Context上下文直接使用的方法, 那便是getApplicationContext().

两者对比:

1. 对于Activity/Service来说, getApplication()和getApplicationContext()的返回值完全相同; 除非厂商修改过接口;
2. BroadcastReceiver在onReceive的过程, 能使用getBaseContext().getApplicationContext获取所在Application, 而无法使用getApplication;
3. ContentProvider能使用getContext().getApplicationContext()获取所在Application. 绝大多数情况下没有问题, 但是有可能会出现空指针的问题, 情况如下:

当同一个进程有多个apk的情况下, 对于第二个apk是由provider方式拉起的, 前面介绍过provider创建过程并不会初始化所在application, 此时执行 getContext().getApplicationContext()返回的结果便是NULL. 所以对于这种情况要做好判空.

**Tips:** 如果对于Application理解不够深刻, 建议getApplicationContext()方法谨慎使用, 做好是否为空的判定,防止出现空指针异常.

## 参考

理解Android Context

http://gityuan.com/2017/04/09/android\_context/

全面剖析SharedPreferences

# Application创建过程

## 一. 概述

system进程和app进程都运行着一个或多个app，每个app都会有一个对应的Application对象(该对象 跟LoadedApk一一对应)。下面分别以下两种进程创建Application的过程：

* system\_server进程；
* app进程；

## 二. system\_server进程

### 2.1 SystemServer.run

[-> SystemServer.java]

public final class SystemServer {

private void run() {

...

createSystemContext(); //[见2.2]

startBootstrapServices(); //开始启动服务

...

}

}

### 2.2 createSystemContext

[-> SystemServer.java]

private void createSystemContext() {

ActivityThread activityThread = ActivityThread.systemMain(); //[见2.3]

mSystemContext = activityThread.getSystemContext(); //[见2.6.1]

mSystemContext.setTheme(android.R.style.Theme\_DeviceDefault\_Light\_DarkActionBar);

}

### 2.3 AT.systemMain

[-> ActivityThread.java]

public static ActivityThread systemMain() {

...

ActivityThread thread = new ActivityThread(); //[见2.4]

thread.attach(true); //[见2.5]

return thread;

}

### 2.4 AT初始化

[-> ActivityThread.java]

public final class ActivityThread {

//创建ApplicationThread对象

final ApplicationThread mAppThread = new ApplicationThread();

final Looper mLooper = Looper.myLooper();

final H mH = new H();

//当前进程中首次初始化的app对象

Application mInitialApplication;

final ArrayList<Application> mAllApplications;

//标记当前进程是否为system进程

boolean mSystemThread = false;

//记录system进程的ContextImpl对象

private ContextImpl mSystemContext;

final ArrayMap<String, WeakReference<LoadedApk>> mPackages;

static Handler sMainThreadHandler;

private static ActivityThread sCurrentActivityThread;

ActivityThread() {

mResourcesManager = ResourcesManager.getInstance();

}

}

其中mInitialApplication的赋值过程分两种场景:

* system\_server进程是由ActivityThread.attach()过程赋值;
* 普通app进程是由是由ActivityThread.handleBindApplication()过程赋值;这是进程刚创建后attach到system\_server后, 便会binder call到app进程来执行该方法.

AT.currentApplication返回的便是mInitialApplication对象。创建完ActivityThread对象，接下来执行attach()操作。

### 2.5 AT.attach

[-> ActivityThread.java]

private void attach(boolean system) {

sCurrentActivityThread = this;

mSystemThread = system; //设置mSystemThread为true

if (!system) {

...

} else { //system进程才执行该流程

//创建Instrumentation

mInstrumentation = new Instrumentation();

//[见小节2.6]

ContextImpl context = ContextImpl.createAppContext(

this, getSystemContext().mPackageInfo);

//[见小节2.7]

mInitialApplication = context.mPackageInfo.makeApplication(true, null);

//回调onCreate方法[见小节2.5.1]

mInitialApplication.onCreate();

...

}

}

attach的主要功能：

* 根据LoadedApk对象来创建ContextImpl，对于system进程LoadedApk对象取值为mSystemContext；
* 初始化Application信息。

#### 2.5.1 onCreate

[-> Application.java]

public void onCreate() {

... //该方法为空, 一般地都是由其子类所覆写该方法

}

### 2.6 CI.createAppContext

[-> ContextImpl.java]

static ContextImpl createAppContext(ActivityThread mainThread, LoadedApk packageInfo) {

//[见小节2.6.4]

return new ContextImpl(null, mainThread,

packageInfo, null, null, false, null, null, Display.INVALID\_DISPLAY);

}

创建ContextImpl对象有多种方法，常见的有：

createSystemContext(ActivityThread mainThread)

createAppContext(ActivityThread mainThread, LoadedApk packageInfo)

createApplicationContext(ApplicationInfo application, int flags)

createPackageContext(String packageName, int flags)

此处，packageInfo是getSystemContext().mPackageInfo，getSystemContext()获取的ContextImpl对象， 其成员变量mPackageInfo便是LoadedApk对象。所以先来看看getSystemContext()过程。

#### 2.6.1 AT.getSystemContext

public ContextImpl getSystemContext() {

synchronized (this) {

if (mSystemContext == null) {

mSystemContext = ContextImpl.createSystemContext(this);

}

return mSystemContext;

}

}

单例模式创建mSystemContext对象。

#### 2.6.2 CI.createSystemContext

static ContextImpl createSystemContext(ActivityThread mainThread) {

//创建LoadedApk对象 【见小节2.6.3】

LoadedApk packageInfo = new LoadedApk(mainThread);

// 创建ContextImpl【见小节2.6.4】

ContextImpl context = new ContextImpl(null, mainThread,

packageInfo, null, null, false, null, null, Display.INVALID\_DISPLAY);

...

return context;

}

#### 2.6.3 LoadedApk初始化

public final class LoadedApk {

private final ActivityThread mActivityThread;

private ApplicationInfo mApplicationInfo;

private Application mApplication;

final String mPackageName;

private final ClassLoader mBaseClassLoader;

private ClassLoader mClassLoader;

LoadedApk(ActivityThread activityThread) {

mActivityThread = activityThread; //ActivityThread对象

mApplicationInfo = new ApplicationInfo(); //创建ApplicationInfo对象

mApplicationInfo.packageName = "android";

mPackageName = "android"; //默认包名为"android"

...

mBaseClassLoader = null;

mClassLoader = ClassLoader.getSystemClassLoader(); //创建ClassLoader

...

}

}

只有一个参数的LoadedApk构造方法只有createSystemContext()过程才会创建， 其中LoadedApk初始化过程会创建ApplicationInfo对象，且包名为“android”。 创建完LoadedApk对象，接下里创建ContextImpl对象。

#### 2.6.4 ContextImpl初始化

class ContextImpl extends Context {

final ActivityThread mMainThread;

final LoadedApk mPackageInfo;

private final IBinder mActivityToken;

private final String mBasePackageName;

private Context mOuterContext;

//缓存Binder服务

final Object[] mServiceCache = SystemServiceRegistry.createServiceCache();

private ContextImpl(ContextImpl container, ActivityThread mainThread, LoadedApk packageInfo, IBinder activityToken, UserHandle user, boolean restricted, Display display, Configuration overrideConfiguration, int createDisplayWithId) {

mOuterContext = this; //ContextImpl对象

mMainThread = mainThread; // ActivityThread赋值

mPackageInfo = packageInfo; // LoadedApk赋值

mBasePackageName = packageInfo.mPackageName; //mBasePackageName等于“android”

...

}

}

首次执行getSystemContext，会创建LoadedApk和contextImpl对象，接下来利用刚创建的LoadedApk对象来创建新的ContextImpl对象。

### 2.7 LoadedApk.makeApplication

[-> LoadedApk.java]

public Application makeApplication(boolean forceDefaultAppClass, Instrumentation instrumentation) {

//保证一个LoadedApk对象只创建一个对应的Application对象

if (mApplication != null) {

return mApplication;

}

String appClass = mApplicationInfo.className;

if (forceDefaultAppClass || (appClass == null)) {

appClass = "android.app.Application"; //system\_server进程, 则进入该分支

}

//创建ClassLoader对象【见小节2.8】

java.lang.ClassLoader cl = getClassLoader();

if (!mPackageName.equals("android")) {

initializeJavaContextClassLoader(); //[见小节2.9]

}

ContextImpl appContext = ContextImpl.createAppContext(mActivityThread, this);

//创建Application对象[见2.10]

Application app = mActivityThread.mInstrumentation.newApplication(cl, appClass, appContext);

appContext.setOuterContext(app);

...

mActivityThread.mAllApplications.add(app);

mApplication = app; //将刚创建的app赋值给mApplication

...

return app;

}

#### 2.8 getClassLoader

[-> LoadedApk.java]

public ClassLoader getClassLoader() {

synchronized (this) {

if (mClassLoader != null) {

return mClassLoader;

}

if (mPackageName.equals("android")) {

if (mBaseClassLoader == null) {

//创建Classloader对象

mClassLoader = ClassLoader.getSystemClassLoader();

} else {

mClassLoader = mBaseClassLoader;

}

return mClassLoader;

}

// 当包名不为"android"的情况

if (mRegisterPackage) {

//【见小节2.8.1】

ActivityManagerNative.getDefault().addPackageDependency(mPackageName);

}

zipPaths.add(mAppDir);

libPaths.add(mLibDir);

apkPaths.addAll(zipPaths);

...

if (mApplicationInfo.isSystemApp()) {

isBundledApp = true;

//对于系统app，则添加vendor/lib, system/lib库

libPaths.add(System.getProperty("java.library.path"));

...

}

final String zip = TextUtils.join(File.pathSeparator, zipPaths);

//获取ClassLoader对象【见小节2.8.2】

mClassLoader = ApplicationLoaders.getDefault().getClassLoader(zip,

mApplicationInfo.targetSdkVersion, isBundledApp, librarySearchPath,

libraryPermittedPath, mBaseClassLoader);

return mClassLoader;

}

}

##### 2.8.1 AMS.addPackageDependency

public void addPackageDependency(String packageName) {

synchronized (this) {

int callingPid = Binder.getCallingPid();

if (callingPid == Process.myPid()) {

return;

}

ProcessRecord proc;

synchronized (mPidsSelfLocked) {

//查询的进程

proc = mPidsSelfLocked.get(Binder.getCallingPid());

}

if (proc != null) {

if (proc.pkgDeps == null) {

proc.pkgDeps = new ArraySet<String>(1);

}

//将目标包名加入到调用者进程的pkgDeps

proc.pkgDeps.add(packageName);

}

}

}

##### 2.8.2 AL.getClassLoader

[-> ApplicationLoaders.java]

public ClassLoader getClassLoader(String zip, int targetSdkVersion, boolean isBundled,

String librarySearchPath, String libraryPermittedPath,

ClassLoader parent) {

//获取父类的类加载器

ClassLoader baseParent = ClassLoader.getSystemClassLoader().getParent();

synchronized (mLoaders) {

if (parent == null) {

parent = baseParent;

}

if (parent == baseParent) {

ClassLoader loader = mLoaders.get(zip);

if (loader != null) {

return loader;

}

//创建PathClassLoader对象

PathClassLoader pathClassloader = PathClassLoaderFactory.createClassLoader(

zip,

librarySearchPath,

libraryPermittedPath,

parent,

targetSdkVersion,

isBundled);

mLoaders.put(zip, pathClassloader);

return pathClassloader;

}

PathClassLoader pathClassloader = new PathClassLoader(zip, parent);

return pathClassloader;

}

}

### 2.9 initializeJavaContextClassLoader

[-> LoadedApk.java]

private void initializeJavaContextClassLoader() {

IPackageManager pm = ActivityThread.getPackageManager();

android.content.pm.PackageInfo pi;

pi = pm.getPackageInfo(mPackageName, 0, UserHandle.myUserId());

boolean sharedUserIdSet = (pi.sharedUserId != null);

boolean processNameNotDefault =

(pi.applicationInfo != null &&

!mPackageName.equals(pi.applicationInfo.processName));

boolean sharable = (sharedUserIdSet || processNameNotDefault);

ClassLoader contextClassLoader =

(sharable)

? new WarningContextClassLoader()

: mClassLoader;

//设置当前线程的Context ClassLoader

Thread.currentThread().setContextClassLoader(contextClassLoader);

}

### 2.10 newApplication

[-> Instrumentation.java]

public Application newApplication(ClassLoader cl, String className, Context context) throws InstantiationException, IllegalAccessException, ClassNotFoundException {

return newApplication(cl.loadClass(className), context);

}

此处cl便是前面getClassLoader所获取的PathClassLoader对象。通过其方法loadClass()来加载目标Application对象；

#### 2.10.1 newApplication

[-> Instrumentation.java]

static public Application newApplication(Class<?> clazz, Context context) throws InstantiationException, IllegalAccessException, ClassNotFoundException {

Application app = (Application)clazz.newInstance(); //【见小节2.10.2】

app.attach(context); //【见小节2.10.3】

return app;

}

#### 2.10.2 Application初始化

[-> Application.java]

public class Application extends ContextWrapper implements ComponentCallbacks2 {

public LoadedApk mLoadedApk;

public Application() {

super(null);

}

}

#### 2.10.3 App.attach

[-> Application.java]

final void attach(Context context) {

attachBaseContext(context); //Application的mBase

mLoadedApk = ContextImpl.getImpl(context).mPackageInfo;

}

该方法主要功能:

1. 将新创建的ContextImpl对象保存到Application的父类成员变量mBase;
2. 将新创建的LoadedApk对象保存到Application的父员变量mLoadedApk;

## 三. App进程

### 3.1 ActivityThread.main

[-> ActivityThread.java]

public static void main(String[] args) {

,,,

ActivityThread thread = new ActivityThread();

thread.attach(false);

,,,

}

这是运行在app进程，当进程由zygote fork后执行ActivityThread的main方法。

### 3.2 AT.attach

[-> ActivityThread.java]

private void attach(boolean system) {

sCurrentActivityThread = this;

mSystemThread = system;

if (!system) {

//初始化RuntimeInit.mApplicationObject值

RuntimeInit.setApplicationObject(mAppThread.asBinder());

final IActivityManager mgr = ActivityManagerNative.getDefault();

mgr.attachApplication(mAppThread); //[见小节3.3]

} else {

...

}

}

经过binder调用，进入system\_server进程，执行如下操作。

### 3.3 AMS.attachApplication

[-> ActivityManagerService.java]

public final void attachApplication(IApplicationThread thread) {

synchronized (this) {

int callingPid = Binder.getCallingPid();

final long origId = Binder.clearCallingIdentity();

attachApplicationLocked(thread, callingPid);

Binder.restoreCallingIdentity(origId);

}

}

private final boolean attachApplicationLocked(IApplicationThread thread, int pid) {

ProcessRecord app;

if (pid != MY\_PID && pid >= 0) {

synchronized (mPidsSelfLocked) {

app = mPidsSelfLocked.get(pid); // 根据pid获取ProcessRecord

}

}

...

ApplicationInfo appInfo = app.instrumentationInfo != null

? app.instrumentationInfo : app.info;

//[见流程3.4]

thread.bindApplication(processName, appInfo, providers, app.instrumentationClass,

profilerInfo, app.instrumentationArguments, app.instrumentationWatcher,

app.instrumentationUiAutomationConnection, testMode, enableOpenGlTrace,

isRestrictedBackupMode || !normalMode, app.persistent,

new Configuration(mConfiguration), app.compat,

getCommonServicesLocked(app.isolated),

mCoreSettingsObserver.getCoreSettingsLocked());

...

return true;

}

system\_server收到attach操作, 然后再向新创建的进程执行handleBindApplication()过程:

### 3.4 AT.handleBindApplication

[-> ActivityThread.java ::H]

当主线程收到H.BIND\_APPLICATION,则调用handleBindApplication

private void handleBindApplication(AppBindData data) {

mBoundApplication = data;

Process.setArgV0(data.processName);//设置进程名

...

//获取LoadedApk对象[见小节3.5]

data.info = getPackageInfoNoCheck(data.appInfo, data.compatInfo);

...

// 创建ContextImpl上下文[2.6.4]

final ContextImpl appContext = ContextImpl.createAppContext(this, data.info);

...

try {

// 此处data.info是指LoadedApk, 通过反射创建目标应用Application对象[见小节2.7]

Application app = data.info.makeApplication(data.restrictedBackupMode, null);

mInitialApplication = app;

...

mInstrumentation.onCreate(data.instrumentationArgs);

//回调onCreate [见小节3.4.1]

mInstrumentation.callApplicationOnCreate(app);

} finally {

StrictMode.setThreadPolicy(savedPolicy);

}

}

在handleBindApplication()的过程中,会同时设置以下两个值:

* LoadedApk.mApplication
* ActivityThread.mInitialApplication

#### 3.4.1 onCreate

[-> Instrumentation.java]

public void callApplicationOnCreate(Application app) {

app.onCreate();

}

### 3.5 getPackageInfoNoCheck

[-> ActivityThread.java]

public final LoadedApk getPackageInfoNoCheck(ApplicationInfo ai, CompatibilityInfo compatInfo) {

return getPackageInfo(ai, compatInfo, null, false, true, false);

}

private LoadedApk getPackageInfo(ApplicationInfo aInfo, CompatibilityInfo compatInfo, ClassLoader baseLoader, boolean securityViolation, boolean includeCode, boolean registerPackage) {

final boolean differentUser = (UserHandle.myUserId() != UserHandle.getUserId(aInfo.uid));

synchronized (mResourcesManager) {

WeakReference<LoadedApk> ref;

if (differentUser) {

ref = null;

} else if (includeCode) {

ref = mPackages.get(aInfo.packageName); //从mPackages查询

} else {

...

}

LoadedApk packageInfo = ref != null ? ref.get() : null;

if (packageInfo == null || (packageInfo.mResources != null

&& !packageInfo.mResources.getAssets().isUpToDate())) {

//创建LoadedApk对象

packageInfo = new LoadedApk(this, aInfo, compatInfo, baseLoader,

securityViolation, includeCode &&

(aInfo.flags&ApplicationInfo.FLAG\_HAS\_CODE) != 0, registerPackage);

if (mSystemThread && "android".equals(aInfo.packageName)) {

...

}

if (differentUser) {

...

} else if (includeCode) {

//将新创建的LoadedApk加入到mPackages

mPackages.put(aInfo.packageName,

new WeakReference<LoadedApk>(packageInfo));

} else {

...

}

}

return packageInfo;

}

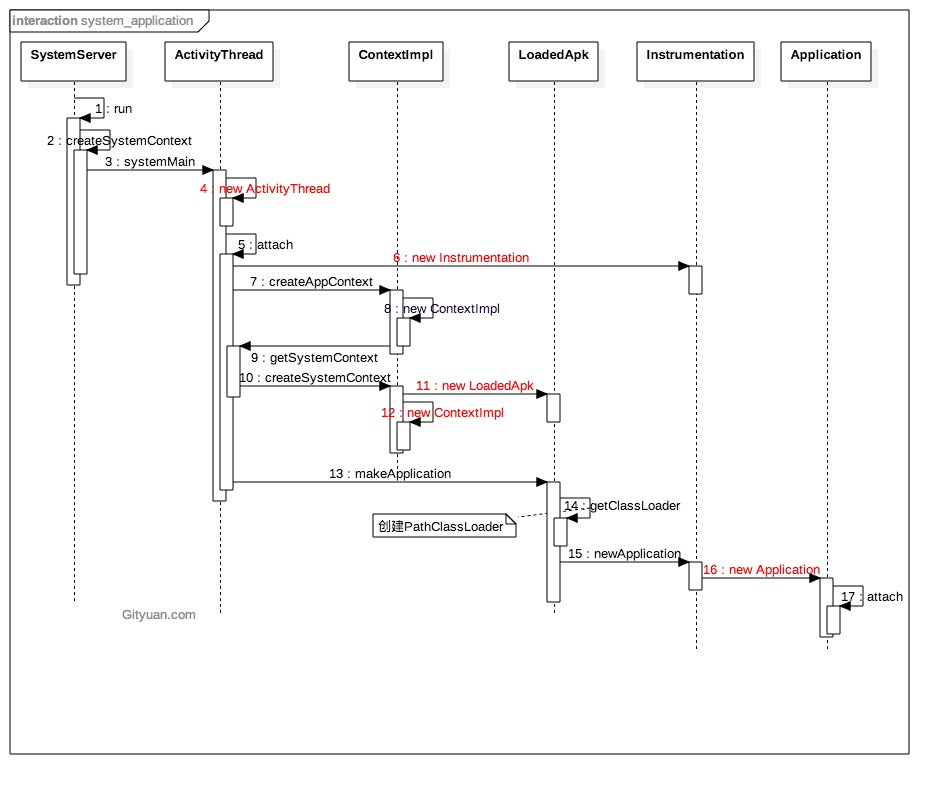
}

创建LoadedApk对象,并将将新创建的LoadedApk加入到mPackages. 也就是说每个app都会创建唯一的LoadedApk对象. 此处aInfo来源于ProcessRecord.info变量, 也就是进程中的第一个app.

## 四. 总结

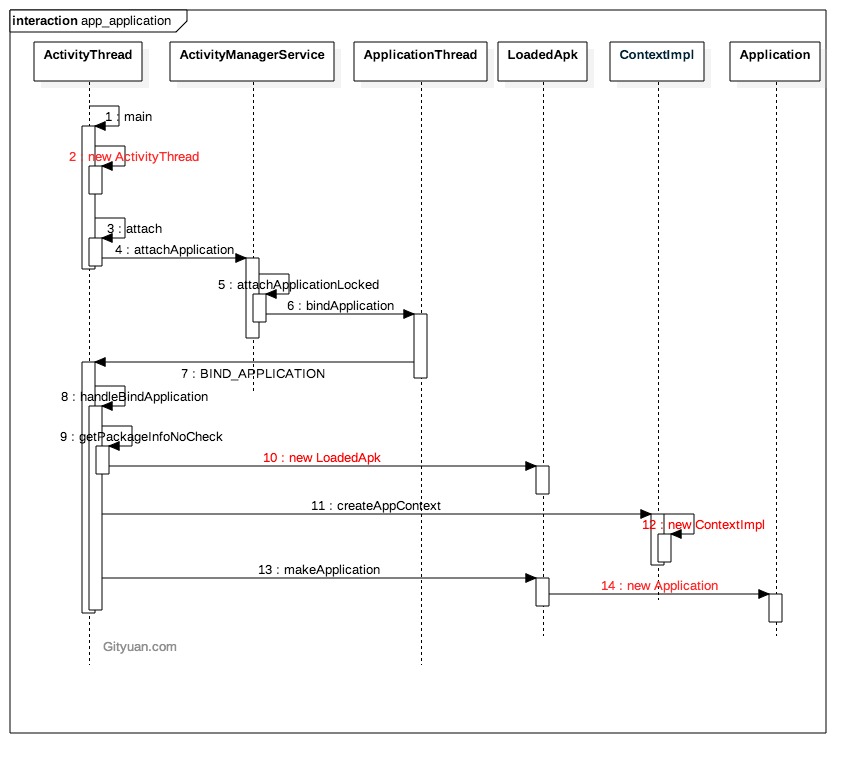
(一)system\_server进程 [查看大图](http://www.gityuan.com/images/application/system_application.jpg)

其application创建过程都创建对象有ActivityThread，Instrumentation, ContextImpl，LoadedApk，Application。 流程图如下：



(二) app进程 [查看大图](http://www.gityuan.com/images/application/app_application.jpg)

其application创建过程都创建对象有ActivityThread，ContextImpl，LoadedApk，Application。 流程图如下：



App进程的Application创建过程，跟system进程的核心逻辑都差不多。只是app进程多了两次binder调用。

## 参考

理解Application创建过程

http://gityuan.com/2017/04/02/android-application/

# Instrumentation

Android instrumentation是Android系统里面的一套控制方法或者”钩子“。这些钩子可以在正常的生命周期（正常是由操作系统控制的)之外控制Android控件的运行，其实指的就是Instrumentation类提供的各种流程控制方法，下表展示了部分方法的对应关系

Method Control by User(Instrumentation) Control by OS

onCreate callActivityOnCreate onCreate

onDestroy callActivityOnDestroy onDestroy

onStart

## 模拟点击屏幕

.使用前需要申明权限

<uses-permission android:name="android.permission.INJECT\_EVENTS" />

2.使用Instrumentation模拟点击有一定限制，只能点击自己的APP，或者UID相同的APP，如果有root权限或者系统签名则可以点击所有APP

try{

getInst().sendPointerSync(MotionEvent.obtain(SystemClock.uptimeMillis(),

SystemClock.uptimeMillis(), MotionEvent.ACTION\_DOWN, x, y, 0)); //x,y 即是事件的坐标

getInst().sendPointerSync(MotionEvent.obtain(SystemClock.uptimeMillis(),

SystemClock.uptimeMillis(), MotionEvent.ACTION\_UP, x, y, 0));

}catch (Exception e){

ToastUtils.showShort("点击出错");

## Hook应用生命周期的最好的地方

所以这里就好比Instrumentation勾住了本应该系统调用的onCreate方法，然后由用户自己来控制勾住的这个方法什么时候执行

**public void** callApplicationOnCreate(Application app) {  
 app.onCreate();  
}

Activity. getInstrumentation()其实都是ActivityThread中的

## runOnMainSync

getInstrumentation().runOnMainSync(new PerformClick(button));

...

/\*

\* 模拟按钮点击

\*/

private class PerformClick implements Runnable {

Button btn;

public PerformClick(Button button) {

btn = button;

}

public void run() {

btn.performClick();

}

}

---------------------

作者：木灵木灵

来源：CSDN

原文：https://blog.csdn.net/a19891024/article/details/54342799

版权声明：本文为博主原创文章，转载请附上博文链接！

**public void** runOnMainSync(Runnable runner) {  
 validateNotAppThread();  
 SyncRunnable sr = **new** SyncRunnable(runner);  
 mThread.getHandler().post(sr);  
 sr.waitForComplete();  
}

## Robotium

Robotium是通过对instrumentation的注入事件（sendKeyDownUpSync、sendPointerSync）的封装、以及通过调用instrumentation的runOnMainSync、runOnUiThread等来完成对控件的操作，让用户可以直接调用一个solo.clickOnText等方法，就能够完成操作，而不用自己去写一堆找到控件，然后再点击触发等方法

# SharedPreferences

## 概述

SharedPreferences(简称SP)是Android中很常用的数据存储方式，SP采用key-value（键值对）形式, 主要用于轻量级的数据存储, 尤其适合保存应用的配置参数, 但不建议使用SP 来存储大规模的数据, 可能会降低性能.

SP采用xml文件格式来保存数据, 该文件所在目录位于/data/data/package\_name/shared\_prefs/

### 使用示例

SharedPreferences sharedPreferences = getSharedPreferences("gityuan", Context.MODE\_PRIVATE);

Editor editor = sharedPreferences.edit();

editor.putString("blog", "www.gityuan.com");

editor.putInt("years", 3);

editor.commit();

生成的gityuan.xml文件内容如下：

**<?xml version='1.0' encoding='utf-8' standalone='yes' ?>**

<map>

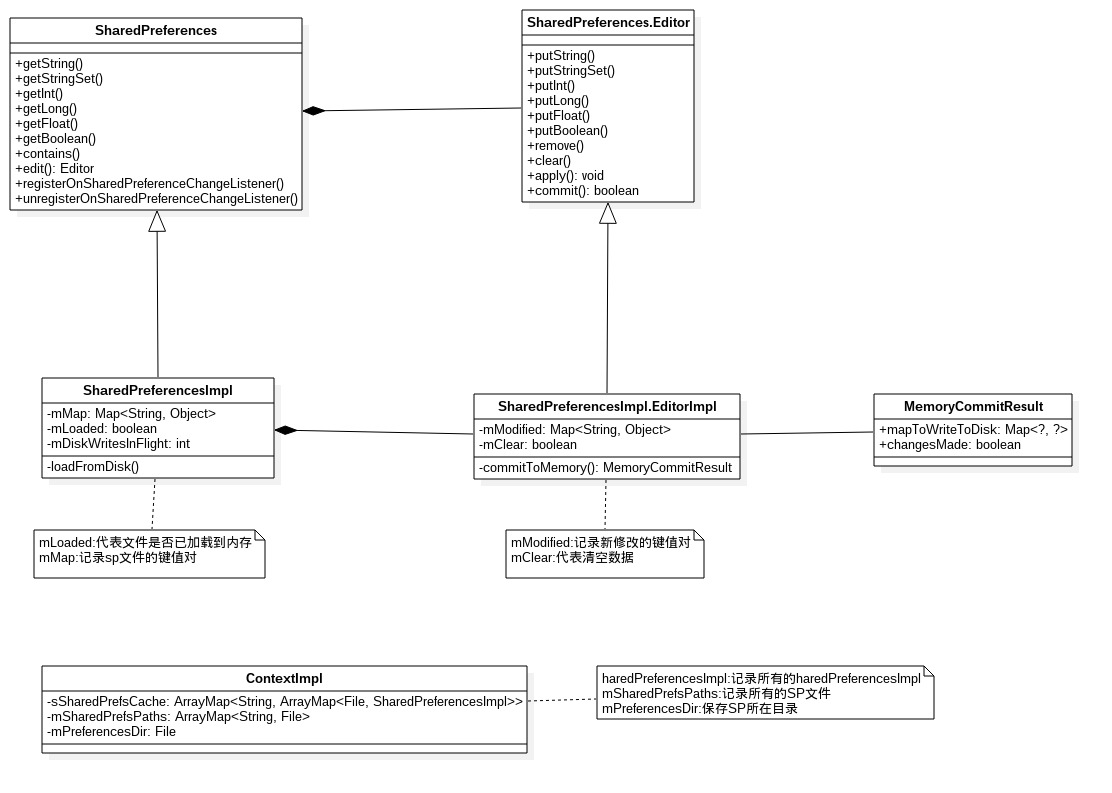
<string name="blog">"www.gityuan.com</string>

<int name="years" value="3" />

</map>

### 架构图

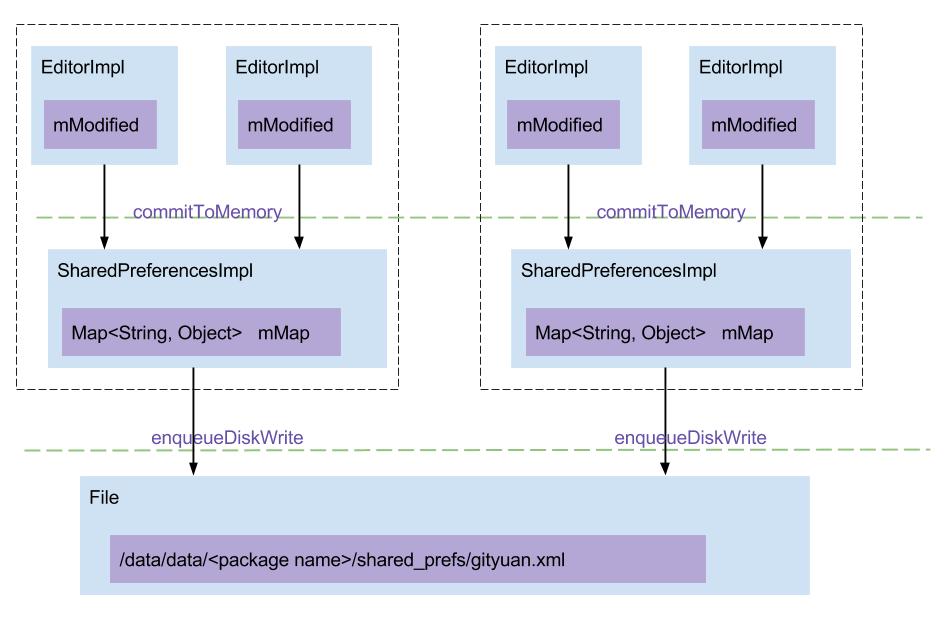
[点击查看大图](http://www.gityuan.com/images/sp/shared_preference.jpg)



SharedPreferences与Editor只是两个接口. SharedPreferencesImpl和EditorImpl分别实现了对应接口. 另外, ContextImpl记录着SharedPreferences的重要数据, 如下:

* sSharedPrefsCache:以包名为key, 二级key是以SP文件, 以SharedPreferencesImpl为value的嵌套map结构. 这里需要sSharedPrefsCache是静态类成员变量, 每个进程是保存唯一一份, 且由ContextImpl.class锁保护.
* mSharedPrefsPaths:记录所有的SP文件, 以文件名为key, 具体文件为value的map结构;
* mPreferencesDir:是指SP所在目录, 是指/data/data//shared\_prefs/

[点击查看大图](http://www.gityuan.com/images/sp/shared_preferences_arch.jpg)



图解:

1. putxxx()操作: 把数据写入到EditorImpl.mModified;
2. apply()或者commit()操作:
   * 先调用commitToMemory(), 将数据同步到SharedPreferencesImpl的mMap, 并保存到MemoryCommitResult的mapToWriteToDisk,
   * 再调用enqueueDiskWrite(), 写入到磁盘文件; 先之前把原有数据保存到.bak为后缀的文件,用于在写磁盘的过程出现任何异常可恢复数据;
3. getxxx()操作: 从SharedPreferencesImpl.mMap读取数据.

Sd

## SharedPreferences

### 2.1 获取方式

#### 2.1.1 getPreferences

[-> Activity.java]

public SharedPreferences getPreferences(int mode) {

//[见下文]

return getSharedPreferences(getLocalClassName(), mode);

}

Activity.getPreferences(mode): 以当前Activity的类名作为SP的文件名. 即xxxActivity.xml.

#### 2.1.2 getDefaultSharedPreferences

[-> PreferenceManager.java]

public static SharedPreferences getDefaultSharedPreferences(Context context) {

//[见下文]

return context.getSharedPreferences(getDefaultSharedPreferencesName(context),

getDefaultSharedPreferencesMode());

}

PreferenceManager.getDefaultSharedPreferences(Context): 以包名加上\_preferences作为文件名, 以MODE\_PRIVATE模式创建SP文件. 即packgeName\_preferences.xml.

#### 2.1.3 getSharedPreferences

当然也可以直接调用Context.getSharedPreferences(name, mode), 以上所有的方法最终都是调用到如下方法:

[-> ContextImpl.java]

class ContextImpl extends Context {

private ArrayMap<String, File> mSharedPrefsPaths;

public SharedPreferences getSharedPreferences(String name, int mode) {

File file;

synchronized (ContextImpl.class) {

if (mSharedPrefsPaths == null) {

mSharedPrefsPaths = new ArrayMap<>();

}

//先从mSharedPrefsPaths查询是否存在相应文件

file = mSharedPrefsPaths.get(name);

if (file == null) {

//如果文件不存在, 则创建新的文件 [见小节2.1.4]

file = getSharedPreferencesPath(name);

mSharedPrefsPaths.put(name, file);

}

}

//[见小节2.2]

return getSharedPreferences(file, mode);

}

}

#### 2.1.4 getSharedPreferencesPath

[-> ContextImpl.java]

public File getSharedPreferencesPath(String name) {

return makeFilename(getPreferencesDir(), name + ".xml");

}

private File getPreferencesDir() {

synchronized (mSync) {

if (mPreferencesDir == null) {

//创建目录/data/data/package name/shared\_prefs/

mPreferencesDir = new File(getDataDir(), "shared\_prefs");

}

return ensurePrivateDirExists(mPreferencesDir);

}

} 流程说明:

1. 先从mSharedPrefsPaths查询是否存在相应文件;
2. 如果文件不存在, 则创建新的xml文件; 如果目录也不存在, 则先创建目录创建目录/data/data/package name/shared\_prefs/
3. 其中mSharedPrefsPaths用于记录所有的SP文件, 是以文件名为key的Map数据结构.

### 2.2 getSharedPreferences

[-> ContextImpl.java]

public SharedPreferences getSharedPreferences(File file, int mode) {

checkMode(mode); //[见小节2.2.1]

SharedPreferencesImpl sp;

synchronized (ContextImpl.class) {

//[见小节2.2.2]

final ArrayMap<File, SharedPreferencesImpl> cache = getSharedPreferencesCacheLocked();

sp = cache.get(file);

if (sp == null) {

//创建SharedPreferencesImpl[见小节2.3]

sp = new SharedPreferencesImpl(file, mode);

cache.put(file, sp);

return sp;

}

}

//指定多进程模式, 则当文件被其他进程改变时,则会重新加载

if ((mode & Context.MODE\_MULTI\_PROCESS) != 0 ||

getApplicationInfo().targetSdkVersion < android.os.Build.VERSION\_CODES.HONEYCOMB) {

sp.startReloadIfChangedUnexpectedly();

}

return sp;

}

#### 2.2.1 checkMode

[-> ContextImpl.java]

private void checkMode(int mode) {

if (getApplicationInfo().targetSdkVersion >= Build.VERSION\_CODES.N) {

if ((mode & MODE\_WORLD\_READABLE) != 0) {

throw new SecurityException("MODE\_WORLD\_READABLE no longer supported");

}

if ((mode & MODE\_WORLD\_WRITEABLE) != 0) {

throw new SecurityException("MODE\_WORLD\_WRITEABLE no longer supported");

}

}

}

从Android N开始, 创建的SP文件模式, 不允许MODE\_WORLD\_READABLE和MODE\_WORLD\_WRITEABLE模块, 否则会直接抛出异常SecurityException. 另外, 顺带说一下MODE\_MULTI\_PROCESS这种多进程的方式也是Google不推荐的方式, 后续同样会不再支持, 强烈建议App不用使用该方式来实现多个进程实现 同一个SP文件.

当设置MODE\_MULTI\_PROCESS模式, 则每次getSharedPreferences过程, 会检查SP文件上次修改时间和文件大小, 一旦所有修改则会重新从磁盘加载文件.

#### 2.2.2 getSharedPreferencesCacheLocked

[-> ContextImpl.java]

private ArrayMap<File, SharedPreferencesImpl> getSharedPreferencesCacheLocked() {

if (sSharedPrefsCache == null) {

sSharedPrefsCache = new ArrayMap<>();

}

final String packageName = getPackageName();

ArrayMap<File, SharedPreferencesImpl> packagePrefs = sSharedPrefsCache.get(packageName);

if (packagePrefs == null) {

packagePrefs = new ArrayMap<>();

sSharedPrefsCache.put(packageName, packagePrefs);

}

return packagePrefs;

}

### 2.3 SharedPreferencesImpl初始化

[-> SharedPreferencesImpl.java]

SharedPreferencesImpl(File file, int mode) {

mFile = file;

//创建为.bak为后缀的备份文件

mBackupFile = makeBackupFile(file);

mMode = mode;

mLoaded = false;

mMap = null;

startLoadFromDisk(); //[见小节2.3.1]

}

同名的.bak备份文件用于发生异常时, 可通过备份文件来恢复数据.

#### 2.3.1 startLoadFromDisk

[-> SharedPreferencesImpl.java]

private void startLoadFromDisk() {

synchronized (this) {

mLoaded = false;

}

new Thread("SharedPreferencesImpl-load") {

public void run() {

loadFromDisk(); //[见小节2.3.2]

}

}.start();

}

mLoaded用于标记SP文件已加载到内存. 创建线程去实现从磁盘加载sp文件的工作.

#### 2.3.2 loadFromDisk

[-> SharedPreferencesImpl.java]

private void loadFromDisk() {

synchronized (SharedPreferencesImpl.this) {

if (mLoaded) {

return;

}

if (mBackupFile.exists()) {

mFile.delete();

mBackupFile.renameTo(mFile);

}

}

Map map = null;

StructStat stat = null;

try {

stat = Os.stat(mFile.getPath());

if (mFile.canRead()) {

BufferedInputStream str = null;

try {

str = new BufferedInputStream(new FileInputStream(mFile), 16\*1024);

map = XmlUtils.readMapXml(str);

} catch (XmlPullParserException | IOException e) {

...

} finally {

IoUtils.closeQuietly(str);

}

}

} catch (ErrnoException e) {

...

}

synchronized (SharedPreferencesImpl.this) {

mLoaded = true;

if (map != null) {

mMap = map; //从文件读取的信息保存到mMap

mStatTimestamp = stat.st\_mtime; //更新修改时间

mStatSize = stat.st\_size; //更新文件大小

} else {

mMap = new HashMap<>();

}

notifyAll(); //唤醒处于等待状态的线程

}

}

整个获取SharedPreferences简单总结:

* 首次使用则创建相应xml文件;
* 异步加载文件内容到内存; 此时执行getXXX()和setxxx()以及edit()方法都是阻塞等待的, 直到文件数据全部加载到内存;
* 一旦完全加载到内存, 后续的getXXX()则是直接访问内存.

### 2.4 查询数据

#### 2.4.1 getString

[-> SharedPreferencesImpl.java]

public String getString(String key, @Nullable String defValue) {

synchronized (this) {

//检查是否加载完成[见小节2.4.2]

awaitLoadedLocked();

String v = (String)mMap.get(key);

return v != null ? v : defValue;

}

}

* 当loadFromDisk没有执行完成, 则会阻塞查询操作;
* 当数据加载完成, 则直接从mMap来查询相应数据;

#### 2.4.2 awaitLoadedLocked

[-> SharedPreferencesImpl.java]

private void awaitLoadedLocked() {

if (!mLoaded) {

BlockGuard.getThreadPolicy().onReadFromDisk();

}

while (!mLoaded) {

try {

wait(); //当没有加载完成,则进入等待状态

} catch (InterruptedException unused) {

}

}

}

## 三. Editor

###　3.1　edit [-> SharedPreferencesImpl.java]

public Editor edit() {

synchronized (this) {

awaitLoadedLocked(); //[见小节2.4.2]

}

return new EditorImpl(); //创建EditorImpl

}

该过程同样要等待awaitLoadedLocked完成, 然后创建EditorImpl对象. 而EditorImpl作为SharedPreferencesImpl的内部类,其继承于Editor类.

### 3.2 EditorImpl

[-> SharedPreferencesImpl.java::　EditorImpl]

public final class EditorImpl implements Editor {

private final Map<String, Object> mModified = Maps.newHashMap();

private boolean mClear = false;

//插入数据

public Editor putString(String key, @Nullable String value) {

synchronized (this) {

//插入数据, 先暂存到mModified对象

mModified.put(key, value);

return this;

}

}

//移除数据

public Editor remove(String key) {

synchronized (this) {

mModified.put(key, this);

return this;

}

}

//清空全部数据

public Editor clear() {

synchronized (this) {

mClear = true;

return this;

}

}

}

从这里可以看出, 这些数据修改操作仅仅是修改mModified和mClear. 直到数据提交commit或许apply过程, 才会真正的把数据更新到SharedPreferencesImpl(简称SPI). 比如设置mClear=true则会情况SPI的mMap数据.

## 四. 数据提交

这里重点来说说数据提交的两个重要方法commit()和apply().

### 4.1 commit

[-> SharedPreferencesImpl.java::　EditorImpl]

public boolean commit() {

//将数据更新到内存[见小节4.2]

MemoryCommitResult mcr = commitToMemory();

//将内存数据同步到文件[见小节4.3]

SharedPreferencesImpl.this.enqueueDiskWrite(mcr, null);

try {

//进入等待状态, 直到写入文件的操作完成

mcr.writtenToDiskLatch.await();

} catch (InterruptedException e) {

return false;

}

//通知监听则, 并在主线程回调onSharedPreferenceChanged()方法

notifyListeners(mcr);

// 返回文件操作的结果数据

return mcr.writeToDiskResult;

}

#### 4.2 commitToMemory

[-> SharedPreferencesImpl.java::　EditorImpl]

private MemoryCommitResult commitToMemory() {

MemoryCommitResult mcr = new MemoryCommitResult();

synchronized (SharedPreferencesImpl.this) {

if (mDiskWritesInFlight > 0) {

mMap = new HashMap<String, Object>(mMap);

}

mcr.mapToWriteToDisk = mMap;

mDiskWritesInFlight++;

//是否有监听key改变的监听者

boolean hasListeners = mListeners.size() > 0;

if (hasListeners) {

mcr.keysModified = new ArrayList<String>();

mcr.listeners = new HashSet<OnSharedPreferenceChangeListener>(mListeners.keySet());

}

synchronized (this) {

//当mClear为true, 则直接清空mMap

if (mClear) {

if (!mMap.isEmpty()) {

mcr.changesMade = true;

mMap.clear();

}

mClear = false;

}

for (Map.Entry<String, Object> e : mModified.entrySet()) {

String k = e.getKey();

Object v = e.getValue();

//注意此处的this是个特殊值, 用于移除相应的key操作.

if (v == this || v == null) {

if (!mMap.containsKey(k)) {

continue;

}

mMap.remove(k);

} else {

if (mMap.containsKey(k)) {

Object existingValue = mMap.get(k);

if (existingValue != null && existingValue.equals(v)) {

continue;

}

}

mMap.put(k, v);

}

mcr.changesMade = true; // changesMade代表数据是否有改变

if (hasListeners) {

mcr.keysModified.add(k); //记录发生改变的key

}

}

mModified.clear(); //清空EditorImpl中的mModified数据

}

}

return mcr;

}

该方法的主要功能: 把EditorImpl数据更新到SPI.

* 将mMap信息赋值给mapToWriteToDisk, 并mDiskWritesInFlight加1;
* 当mClear为true, 则直接清空mMap;
* 当value值为this或null, 则移除相应的key;
* 当value值发生改变, 则会更新到mMap;

只要有key/value发生改变(新增, 删除), 则设置mcr.changesMade = true. 最后会清空EditorImpl中的mModified数据.

#### 4.3 enqueueDiskWrite

[-> SharedPreferencesImpl.java]

private void enqueueDiskWrite(final MemoryCommitResult mcr, final Runnable postWriteRunnable) {

final Runnable writeToDiskRunnable = new Runnable() {

public void run() {

synchronized (mWritingToDiskLock) {

//执行文件写入操作[见小节4.3.1]

writeToFile(mcr);

}

synchronized (SharedPreferencesImpl.this) {

mDiskWritesInFlight--;

}

//此时postWriteRunnable为null不执行该方法

if (postWriteRunnable != null) {

postWriteRunnable.run();

}

}

};

final boolean isFromSyncCommit = (postWriteRunnable == null);

if (isFromSyncCommit) { //commit方法会进入该分支

boolean wasEmpty = false;

synchronized (SharedPreferencesImpl.this) {

//commitToMemory过程会加1,则wasEmpty=true

wasEmpty = mDiskWritesInFlight == 1;

}

if (wasEmpty) {

//跳转到上面

writeToDiskRunnable.run();

return;

}

}

//不执行该方法

QueuedWork.singleThreadExecutor().execute(writeToDiskRunnable);

}

##### 4.3.1 writeToFile

private void writeToFile(MemoryCommitResult mcr) {

if (mFile.exists()) {

if (!mcr.changesMade) { //没有key发生改变, 则直接返回

mcr.setDiskWriteResult(true);

return;

}

if (!mBackupFile.exists()) {

//当备份文件不存在, 则把mFile重命名为备份文件

if (!mFile.renameTo(mBackupFile)) {

mcr.setDiskWriteResult(false);

return;

}

} else {

mFile.delete(); //否则,直接删除mFile

}

}

try {

FileOutputStream str = createFileOutputStream(mFile);

if (str == null) {

mcr.setDiskWriteResult(false);

return;

}

//将mMap全部信息写入文件

XmlUtils.writeMapXml(mcr.mapToWriteToDisk, str);

FileUtils.sync(str);

str.close();

ContextImpl.setFilePermissionsFromMode(mFile.getPath(), mMode, 0);

try {

final StructStat stat = Os.stat(mFile.getPath());

synchronized (this) {

mStatTimestamp = stat.st\_mtime;

mStatSize = stat.st\_size;

}

} catch (ErrnoException e) {

...

}

//写入成功, 则删除备份文件

mBackupFile.delete();

//返回写入成功, 唤醒等待线程

mcr.setDiskWriteResult(true);

return;

} catch (XmlPullParserException e) {

...

} catch (IOException e) {

...

}

//如果写入文件的操作失败, 则删除未成功写入的文件

if (mFile.exists()) {

if (!mFile.delete()) {

...

}

}

//返回写入失败, 唤醒等待线程

mcr.setDiskWriteResult(false);

}

该方法的主要功能:

1. 当没有key发生改变, 则直接返回; 否则执行step2;
2. 将mMap全部信息写入文件, 如果写入成功则删除备份文件,如果写入失败则删除mFile.

可见, 每次commit是把全部数据更新到文件, 所以每个文件的数据量必须保证足够精简. 再来看看apply过程.

### 4.4 apply

[-> SharedPreferencesImpl.java::　EditorImpl]

public void apply() {

//把数据更新到内存[见小节4.2]

final MemoryCommitResult mcr = commitToMemory();

final Runnable awaitCommit = new Runnable() {

public void run() {

try {

//进入等待状态

mcr.writtenToDiskLatch.await();

} catch (InterruptedException ignored) {

}

}

};

//将awaitCommit添加到QueuedWork

QueuedWork.add(awaitCommit);

Runnable postWriteRunnable = new Runnable() {

public void run() {

awaitCommit.run();

//从QueuedWork移除

QueuedWork.remove(awaitCommit);

}

};

//[见小节4.4.1]

SharedPreferencesImpl.this.enqueueDiskWrite(mcr, postWriteRunnable);

notifyListeners(mcr);

}

#### 4.4.1 enqueueDiskWrite

[-> SharedPreferencesImpl.java]

private void enqueueDiskWrite(final MemoryCommitResult mcr, final Runnable postWriteRunnable) {

final Runnable writeToDiskRunnable = new Runnable() {

public void run() {

synchronized (mWritingToDiskLock) {

//执行文件写入操作[见小节4.3.1]

writeToFile(mcr);

}

synchronized (SharedPreferencesImpl.this) {

mDiskWritesInFlight--;

}

if (postWriteRunnable != null) {

postWriteRunnable.run();

}

}

};

final boolean isFromSyncCommit = (postWriteRunnable == null);

if (isFromSyncCommit) {

... //postWriteRunnable不为空

}

//将任务放入单线程的线程池来执行

QueuedWork.singleThreadExecutor().execute(writeToDiskRunnable);

}

#### 4.4.2 QueuedWork

[-> QueuedWork.java]

public class QueuedWork {

private static final ConcurrentLinkedQueue<Runnable> sPendingWorkFinishers =

new ConcurrentLinkedQueue<Runnable>();

public static void add(Runnable finisher) {

sPendingWorkFinishers.add(finisher);

}

public static void remove(Runnable finisher) {

sPendingWorkFinishers.remove(finisher);

}

public static void waitToFinish() {

Runnable toFinish;

while ((toFinish = sPendingWorkFinishers.poll()) != null) {

toFinish.run();

}

}

public static boolean hasPendingWork() {

return !sPendingWorkFinishers.isEmpty();

}

}

可见, apply跟commit的最大区别 在于apply的写入文件操作是在单线程的线程池来完成.

* apply方法开始的时候, 会把awaitCommit放入QueuedWork;
* 文件写入操作完成, 则会把相应的awaitCommit从QueuedWork中移除.

QueuedWork在这里存在的价值主要是用于在Stop Service, finish BroadcastReceiver过程用于 判定是否处理完所有的异步SP操作.

## 五. 总结

apply 与commit的对比

* apply没有返回值, commit有返回值能知道修改是否提交成功
* apply是将修改提交到内存，再异步提交到磁盘文件; commit是同步的提交到磁盘文件;
* 多并发的提交commit时，需等待正在处理的commit数据更新到磁盘文件后才会继续往下执行，从而降低效率; 而apply只是原子更新到内存，后调用apply函数会直接覆盖前面内存数据，从一定程度上提高很多效率。

获取SP与Editor:

* getSharedPreferences()是从ContextImpl.sSharedPrefsCache唯一的SPI对象;
* edit()每次都是创建新的EditorImpl对象.

### 优化建议:

* 强烈建议不要在sp里面存储特别大的key/value, 有助于减少卡顿/anr
* 请不要高频地使用apply, 尽可能地批量提交;commit直接在主线程操作, 更要注意了
* 不要使用MODE\_MULTI\_PROCESS;
* 高频写操作的key与高频读操作的key可以适当地拆分文件, 由于减少同步锁竞争;
* 不要一上来就执行getSharedPreferences().edit(), 应该分成两大步骤来做, 中间可以执行其他代码.
* 不要连续多次edit(), 应该获取一次获取edit(),然后多次执行putxxx(), 减少内存波动; 经常看到大家喜欢封装方法, 结果就导致这种情况的出现.
* 每次commit时会把全部的数据更新的文件, 所以整个文件是不应该过大的, 影响整体性

## 案例

谷歌输入法hook

# 参考

1. 全面剖析SharedPreferences

http://gityuan.com/2017/06/18/SharedPreferences/

# 主题Theme

主要关注themes.xml,themes\_device\_defaults.xml两个文件。

themes.xml定义了android低版本的theme和Holo theme，themes\_device\_defaults.xml定义了DeviceDefault主题（继承自Holo主题），实际上就是在Holo主题上定制主题（For厂商）。

系统如何去选择默认的主题呢？

/\*\*

\* Returns the most appropriate default theme for the specified target SDK version.

\* <ul>

\* <li>Below API 11: Gingerbread

\* <li>APIs 12 thru 14: Holo

\* <li>APIs 15 thru 23: Device default dark

\* <li>APIs 24 and above: Device default light with dark action bar

\* </ul>

\*

\* @param curTheme The current theme, or 0 if not specified.

\* @param targetSdkVersion The target SDK version.

\* @return A theme resource identifier

\* @hide

\*/

public static int selectDefaultTheme(int curTheme, int targetSdkVersion) {

return selectSystemTheme(curTheme, targetSdkVersion,

com.android.internal.R.style.Theme,

com.android.internal.R.style.Theme\_Holo,

com.android.internal.R.style.Theme\_DeviceDefault,

com.android.internal.R.style.Theme\_DeviceDefault\_Light\_DarkActionBar);

}

/\*\* @hide \*/

public static int selectSystemTheme(int curTheme, int targetSdkVersion, int orig, int holo,

int dark, int deviceDefault) {

if (curTheme != 0) {

return curTheme;

}

if (targetSdkVersion < Build.VERSION\_CODES.HONEYCOMB) {

return orig;

}

if (targetSdkVersion < Build.VERSION\_CODES.ICE\_CREAM\_SANDWICH) {

return holo;

}

if (targetSdkVersion < Build.VERSION\_CODES.N) {

return dark;

}

return deviceDefault;

}

# AlertActivity

在开发过程中，如果在一个Recevier里面收到一条广播后需要进行弹出Dialog的处理，那么这时就需要使用到AlertActivity了，就是用Activity来模拟出Dialog，因为在Android中，Recevier里面是不能弹出Dialog的。

<http://007hiccup.blog.163.com/blog/static/444976192014721348761/>

**mAdvanceKey** = KeyEvent.***KEYCODE\_DPAD\_RIGHT***;  
**mRetreatKey** = KeyEvent.***KEYCODE\_DPAD\_LEFT***;

# APP

## DatePickerDialog

**final** Context themeContext = getContext();  
**final** LayoutInflater inflater = LayoutInflater.*from*(themeContext);  
**final** View view = inflater.inflate(R.layout.date\_picker\_dialog, **null**);  
setView(view);  
  
setButton(***BUTTON\_POSITIVE***, themeContext.getString(R.string.ok), **this**);  
setButton(***BUTTON\_NEGATIVE***, themeContext.getString(R.string.cancel), **this**);

android-7.1.1\_r1/frameworks/base/core/res/res/layout/date\_picker\_dialog.xml

<**DatePicker xmlns:android="http://schemas.android.com/apk/res/android"  
 android:id="@+id/datePicker"  
 android:layout\_gravity="center\_horizontal"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:spinnersShown="true"  
 android:calendarViewShown="false"  
 android:dialogMode="true"** />

## TimePickerDialog

为何最终进入的是android.widget.TimePicker.而不是**com.android.contacts.datepicker.DatePicker呢**

## Notification