

目录

写在前面

查看PathClassLoader的源码

ActivityThread类的主方法

总结

写在前面

双亲委托机制中，当加载一个Class的时候，如果当前ClassLoader有父加载器的时候用父加载器加载。为什么PathClassLoader的父加载器(parent)是BootClassLoader?

查看PathClassLoader的源码

```
public class PathClassLoader extends BaseDexClassLoader {  
    public PathClassLoader(String dexPath, ClassLoader parent) {  
        super((String)null, (File)null, (String)null, (ClassLoader)null);  
        throw new RuntimeException("Stub!");  
    }  
  
    public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader parent) {  
        super((String)null, (File)null, (String)null, (ClassLoader)null);  
        throw new RuntimeException("Stub!");  
    }  
}
```

<https://blog.csdn.net/zhangshuny>

从上图可知，PathClassLoader构造函数传入了一个ClassLoader类型的参数parent。

PathClassLoader是何时创建以及parent是什么

从哪看呢？由于ActivityThread是我们常说的UI线程，ActivityThread类中的main()方法是整个应用的入口。所以我们来看一下ActivityThread的main()。

ActivityThread类的主方法

```
06623: public static void main(String[] args) {
06624:     Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ActivityThreadMain");
06625:
06626:     // CloseGuard defaults to true and can be quite spammy. We
06627:     // disable it here, but selectively enable it later (via
06628:     // StrictMode) on debug builds, but using DropBox, not logs.
06629:     CloseGuard.setEnabled(false);
06630:
06631:     Environment.initForCurrentUser();
06632:
06633:     // Set the reporter for event logging in libcore
06634:     EventLogger.setReporter(new EventLoggingReporter());
06635:
06636:     // Make sure TrustedCertificateStore looks in the right place for CA certificates
06637:     final File configDir = Environment.getUserConfigDirectory(UserHandle.myUserId());
06638:     TrustedCertificateStore.setDefaultUserDirectory(configDir);
06639:
06640:     Process.setArgV0("<pre-initialized>");
06641:
06642:     Looper.prepareMainLooper();
06643:
06644:     // Find the value for {@link #PROC_START_SEQ_IDENT} if provided on the command line.
06645:     // It will be in the format "seq=114"
06646:     long startSeq = 0;
06647:     if (args != null) {
06648:         for (int i = args.length - 1; i >= 0; --i) {
06649:             if (args[i] != null && args[i].startsWith(PROC_START_SEQ_IDENT)) {
06650:                 startSeq = Long.parseLong(
06651:                     args[i].substring(PROC_START_SEQ_IDENT.length()));
06652:             }
06653:         }
06654:     }
06655:     ActivityThread thread = new ActivityThread();
06656:     thread.attach(false, startSeq);
06657:
06658:     if (sMainThreadHandler == null) {
06659:         sMainThreadHandler = thread.getHandler();
06660:     }
06661:
06662:     if (false) {
06663:         Looper.myLooper().setMessageLogging(new
06664:             LogPrinter(Log.DEBUG, "ActivityThread"));
06665:     }
06666:
06667:     // End of event ActivityThreadMain.
06668:     Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
06669:     Looper.loop();
06670:
06671:     throw new RuntimeException("Main thread loop unexpectedly exited");
06672: } ? end main ?
```

<https://blog.csdn.net/zhangshuny>

main方法中进行了MainLooper的初始化和主线程的创建以及主线程Handler的初始化
这里我们关注一下attach方法

```

06478: private void attach(boolean system, long startSeq) {
06479:     sCurrentActivityThread = this;
06480:     mSystemThread = system;
06481:     if (!system) {
06482:         ViewRootImpl.addFirstDrawHandler(new Runnable() {
06483:             @Override
06484:             public void run() {
06485:                 ensureJitEnabled();
06486:             }
06487:         });
06488:         android.ddm.DdmHandleAppName.setAppName("<pre-initialized>",
06489:             UserHandle.myUserId());
06490:         RuntimeInit.setApplicationObject(mAppThread.asBinder());
06491:         final IActivityManager mgr = ActivityManager.getService();
06492:         try {
06493:             mgr.attachApplication(mAppThread, startSeq);
06494:         } catch (RemoteException ex) {
06495:             throw ex.rethrowFromSystemServer();
06496:         }
06497:         // Watch for getting close to heap limit.
06498:         BinderInternal.addGcWatcher(new Runnable() {
06499:             @Override public void run() {
06500:                 if (!mSomeActivitiesChanged) {
06501:                     return;
06502:                 }
06503:                 Runtime runtime = Runtime.getRuntime();
06504:                 long dalvikMax = runtime.maxMemory();
06505:                 long dalvikUsed = runtime.totalMemory() - runtime.freeMemory();
06506:                 if (dalvikUsed > ((3*dalvikMax)/4)) {
06507:                     if (DEBUG_MEMORY_TRIM) Slog.d(TAG, "Dalvik max=" + (dalvikMax/1024)
06508:                         + " total=" + (runtime.totalMemory()/1024)
06509:                         + " used=" + (dalvikUsed/1024));
06510:                     mSomeActivitiesChanged = false;
06511:                     try {
06512:                         mgr.releaseSomeActivities(mAppThread);
06513:                     } catch (RemoteException e) {
06514:                         throw e.rethrowFromSystemServer();
06515:                     }
06516:                 }
06517:             }
06518:         });
06519:     } else {

```

<https://blog.csdn.net/zhangshuny>

在ActivityThread的attach方法中，ActivityManagerService通过attachApplication方法,将ApplicationThread对象绑定到ActivityManagerService上，mAppThread对象所对应的类是ApplicatinThread，是ActivityThread的内部类，实现了IBinder接口，用于ActivityThread和ActivityManagerService的进程间通信

我们接着看attachApplication方法

```

07931: @Override
07932: public final void attachApplication(IApplicationThread thread, long startSeq) {
07933:     synchronized (this) {
07934:         int callingPid = Binder.getCallingPid();
07935:         final int callingUid = Binder.getCallingUid();
07936:         final long origId = Binder.clearCallingIdentity();
07937:         attachApplicationLocked(thread, callingPid, callingUid, startSeq);
07938:         Binder.restoreCallingIdentity(origId);
07939:     }
07940: }
07941:

```

方法中调用了attachApplicationLocked方法

```

07808: getProcessInfo
07809: killApplicationProcess
07810: forceStopPackageLocked
07811: finishForceStopPackageLocked
07812: killPackageProcessesLocked
07813: cleanupDisabledPackageComponentsLocked
07814: clearBroadcastQueueForUserLocked
07815: forceStopPackageLocked
07816: removeProcessNameLocked
07817: removeProcessNameLocked
07818: addProcessNameLocked
07819: removeProcessLocked
07820: processContentProviderPublishTime
07821: processStartTimeoutLocked
07822: attachApplicationLocked
07823: attachApplication
07824: activityIdle
07825: postFinishBooting
07826: enableScreenAfterBoot
07827: showBootMessage
07828: keyguardGoingAway
07829: isKeyguardLocked
07830: finishBooting
07831: bootAnimationComplete
07832: ensureBootCompleted
07833: activityResumed
07834: activityPaused
07835: activityStopped
07836: activityDestroyed
07837: activityRelaunched
07838: reportSizeConfigurations
07839: notifyLaunchTaskBehindComplete
07840: notifyEnterAnimationComplete
07841: getCallingPackage

```

```

07808: checkTime(startTime, "attachApplicationLocked: immediately before bindApplication");
07809: mStackSupervisor.getActivityMetricsLogger().notifyBindApplication(app);
07810: if (app.isolatedEntryPoint != null) {
07811:     // This is an isolated process which should just call an entry point instead of
07812:     // being bound to an application.
07813:     thread.runIsolatedEntryPoint(app.isolatedEntryPoint, app.isolatedEntryPointArgs);
07814: } else if (app.instr != null) {
07815:     thread.bindApplication(processName, appInfo, providers,
07816:         app.instr.mClass,
07817:         profilerInfo, app.instr.mArguments,
07818:         app.instr.mWatcher,
07819:         app.instr.mUiAutomationConnection, testMode,
07820:         mBinderTransactionTrackingEnabled, enableTrackAllocation,
07821:         isRestrictedBackupMode || !normalMode, app.persistent,
07822:         new Configuration(getGlobalConfiguration()), app.compat,
07823:         getCommonServicesLocked(app.isolated),
07824:         mCoreSettingsObserver.getCoreSettingsLocked(),
07825:         buildSerial, isAutoFillCompatEnabled);
07826: } else {
07827:     thread.bindApplication(processName, appInfo, providers, null, profilerInfo,
07828:         null, null, null, testMode,
07829:         mBinderTransactionTrackingEnabled, enableTrackAllocation,
07830:         isRestrictedBackupMode || !normalMode, app.persistent,
07831:         new Configuration(getGlobalConfiguration()), app.compat,
07832:         getCommonServicesLocked(app.isolated),
07833:         mCoreSettingsObserver.getCoreSettingsLocked(),
07834:         buildSerial, isAutoFillCompatEnabled);
07835: }
07836:

```

<https://blog.csdn.net/zhangshuny>

attachApplicationLocked方法中调用了mAppThread的bindApplication方法

ctivityThread.java

```

ApplicationThread
IApplicationThread.Stub
DB_INFO_FORMAT
mLastProcessState
updatePendingConfiguration
scheduleSleeping
scheduleReceiver
scheduleCreateBackupAgent
scheduleDestroyBackupAgent
scheduleCreateService
scheduleBindService
scheduleUnbindService
scheduleServiceArgs
scheduleStopService
bindApplication
runIsolatedEntryPoint
scheduleExit
scheduleSuicide
scheduleApplicationInfoChanged

```

```

00895: AppBindData data = new AppBindData();
00896: data.processName = processName;
00897: data.appInfo = appInfo;
00898: data.providers = providers;
00899: data.instrumentationName = instrumentationName;
00900: data.instrumentationArgs = instrumentationArgs;
00901: data.instrumentationWatcher = instrumentationWatcher;
00902: data.instrumentationUiAutomationConnection = instrumentationUiConnection;
00903: data.debugMode = debugMode;
00904: data.enableBinderTracking = enableBinderTracking;
00905: data.trackAllocation = trackAllocation;
00906: data.restrictedBackupMode = isRestrictedBackupMode;
00907: data.persistent = persistent;
00908: data.config = config;
00909: data.compatInfo = compatInfo;
00910: data.initProfilerInfo = profilerInfo;
00911: data.buildSerial = buildSerial;
00912: data.autoFillCompatibilityEnabled = autoFillCompatibilityEnabled;
00913: sendMessage(H.BIND_APPLICATION, data);
00914:
00915:

```

<https://blog.csdn.net/zhangshuny>

bindApplication方法中创建了AppBindData对象，对象中设置了线程信息、application信息等
然后调用了sendMessage发消息，what = BIND_APPLICATION

```

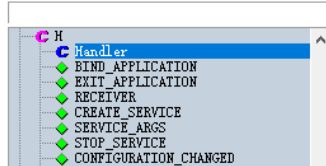
02756: void sendMessage(int what, Object obj) {
02757:     sendMessage(what, obj, 0, 0, false);
02758: }
02759:
02760: private void sendMessage(int what, Object obj, int arg1) {
02761:     sendMessage(what, obj, arg1, 0, false);
02762: }
02763:
02764: private void sendMessage(int what, Object obj, int arg1, int arg2) {
02765:     sendMessage(what, obj, arg1, arg2, false);
02766: }
02767:
02768: private void sendMessage(int what, Object obj, int arg1, int arg2, boolean async) {
02769:     if (DEBUG_MESSAGES) Slog.v(
02770:         TAG, "SCHEDULE " + what + " " + mH.codeToString(what)
02771:         + ": " + arg1 + " / " + arg2);
02772:     Message msg = Message.obtain();
02773:     msg.what = what;
02774:     msg.obj = obj;
02775:     msg.arg1 = arg1;
02776:     msg.arg2 = arg2;
02777:     if (async) {
02778:         msg.setAsynchronous(true);
02779:     }
02780:     mH.sendMessage(msg);
02781: }
02782:
02783: private void sendMessage(int what, Object obj, int arg1, int arg2, int seq) {
02784:     if (DEBUG_MESSAGES) Slog.v(
02785:         TAG, "SCHEDULE " + mH.codeToString(what) + " arg1=" + arg1 + " arg2=" + arg2 +
02786:         "seq=" + seq);
02787:     Message msg = Message.obtain();
02788:     msg.what = what;
02789:     SomeArgs args = SomeArgs.obtain();
02790:     args.arg1 = obj;
02791:     args.argi1 = arg1;
02792:     args.argi2 = arg2;
02793:     args.argi3 = seq;
02794:     msg.obj = args;
02795:     mH.sendMessage(msg);
02796: }
02797:

```

<https://blog.csdn.net/zhangshuny>

sendMessage中封装了一下给主线程handler: mH 发消息

ActivityThread.java

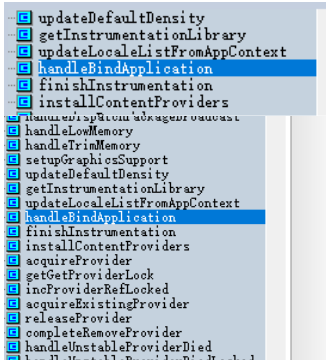


```

01643:     } ? end codeToString ?
01644:     public void handleMessage(Message msg) {
01645:         if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
01646:         switch (msg.what) {
01647:             case BIND_APPLICATION:
01648:                 Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "bindApplication");
01649:                 AppBindData data = (AppBindData) msg.obj;
01650:                 handleBindApplication(data);
01651:                 Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
01652:                 break;
01653:             case EXIT_APPLICATION:

```

主线程handler的handleMessage方法中调用handleBindApplication方法



```

05748:         final ContextImpl appContext = ContextImpl.createAppContext(this, data.info);
05749:         updateLocaleListFromAppContext(appContext,
05750:             mResourceManager.getConfiguration().getLocales());
05751:
05752:         instrApp.initForUser(UserHandle.myUserId());
05753:
05794:         final LoadedApk pi = getPackageInfo(instrApp, data.compatInfo,
05795:             appContext.getClassLoader(), false, true, false);
05796:         final ContextImpl instrContext = ContextImpl.createAppContext(this, pi);
05797:
05798:         try {
05799:             final ClassLoader cl = instrContext.getClassLoader();
05800:             mInstrumentation = (Instrumentation)
05801:                 cl.loadClass(data.instrumentationName.getClassName()).newInstance();
05802:         } catch (Exception e) {
05803:             throw new RuntimeException(
05804:                 "Unable to instantiate instrumentation "
05805:                 + data.instrumentationName + ": " + e.toString(), e);
05806:         }
05807:
05808:

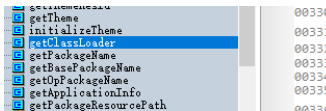
```

<https://blog.csdn.net/zhangshuny>

handleBindApplication方法中代码比较多，先关注两点：

1. 创建了ContextImpl
2. 调用了ContextImpl的getClassLoader()方法

看一下如何getClassLoader的



```

00330:     @Override
00331:     public ClassLoader getClassLoader() {
00332:         return mClassLoader != null ? mClassLoader : (mPackageInfo != null ? mPackageInfo.getClassLoader() : ClassLoader.getSystemClassLoader())
00333:     }
00334:
00335:     @Override
00336:     public String getPackageName() {

```

走这个方法

mClassLoader和mPackageInfo是构造函数中传入的，mClassLoader传入的是null，mPackageInfo

传入的是data.info所以调用mPackageInfo.getClassLoader()

```

00807: public ClassLoader getClassLoader() {
00808:     synchronized (this) {
00809:         if (mClassLoader == null) {
00810:             createOrUpdateClassLoaderLocked(null /*addedPaths*/);
00811:         }
00812:         return mClassLoader;
00813:     }
00814: }
00865:

00604: private void createOrUpdateClassLoaderLocked(List<String> addedPaths) {
00605:     if (mPackageName.equals("android")) {
00606:         // Note: This branch is taken for system server and we don't need to setup
00607:         // jit profiling support.
00608:         if (mClassLoader != null) {
00609:             // nothing to update
00610:             return;
00611:         }
00612:
00613:         if (mBaseClassLoader != null) {
00614:             mClassLoader = mBaseClassLoader;
00615:         } else {
00616:             mClassLoader = ClassLoader.getSystemClassLoader();
00617:         }
00618:         mAppComponentFactory = createAppFactory(mApplicationInfo, mClassLoader);
00619:
00620:         return;
00621:     }
00622: }
00677:

01094: * @revised 1.4
01095: */
01096: @CallerSensitive
01097: public static ClassLoader getSystemClassLoader() {
01098:     return SystemClassLoader.loader;
01099: }
01100:

00179: */
00180: public abstract class ClassLoader {
00181:
00182:     static private class SystemClassLoader {
00183:         public static ClassLoader loader = ClassLoader.createSystemClassLoader();
00184:     }
00185:
00186:     /**
00201:     private final ClassLoader parent;
00202:
00203:     /**
00204:     * Encapsulates the set of parallel capable loader types.
00205:     */
00206:     private static ClassLoader createSystemClassLoader() {
00207:         String classPath = System.getProperty("java.class.path", ".");
00208:         String librarySearchPath = System.getProperty("java.library.path", "");
00209:
00210:         // String[] paths = classPath.split(":");
00211:         // URL[] urls = new URL[paths.length];
00212:         // for (int i = 0; i < paths.length; i++) {
00213:         //     try {
00214:         //         urls[i] = new URL("file://" + paths[i]);
00215:         //     }
00216:         //     catch (Exception ex) {
00217:         //         ex.printStackTrace();
00218:         //     }
00219:         // }
00220:         // return new java.net.URLClassLoader(urls, null);
00221:
00222:         // TODO: Make this a java.net.URLClassLoader once we have those?
00223:         return new PathClassLoader(classPath, librarySearchPath, BootClassLoader.getInstance());
00224:     }
00225: }
00226:

```

如果是系统api类
调用ClassLoader.getSystemClassLoader()

创建PathClassLoader,传入BootClassLoader

总结

我们写的应用程序的Class使用PathClassLoader加载，PathClassLoader的父加载器(parent)是BootClassLoader