Android的Proxy/Delegate Application框架

Posted on 2013 年 11 月 25 日 by yammaz

有的时候，为了实现一些特殊需求，如界面换肤、插件化等，我们希望改变应用的运行环境(surrounding)。例如，我们希望某个应用在运行时，所有Class（包括自定义Application，下面假设它叫MyApplication）都被一个自定义的ClassLoader加载。

要实现这个需求，需要在MyApplication被加载之前，先替换掉API层的默认ClassLoader，否则MyApplication就会被默认ClassLoader加载。但这会产生一个悖论，MyApplication被加载之前，没有任何应用代码可以运行，替换ClassLoader无法办到。Proxy/Delegate Application框架就是用来解决这类问题的。

Proxy/Delegate Application简介

在Proxy/Delegate Application框架里，应用一共有两个Application对象，一个称为ProxyApplication，另一个称为DelegateApplication：

(1) ProxyApplication：框架会提供一个ProxyApplication抽象基类（abstract class），使用者需要继承这个类，并重载其initProxyApplication()方法，在其中改变surrounding，如替换ClassLoader等。

(2) DelegateApplication：即应用原有的Application，应用从getApplicationContext()等方法中取到的都是DelegateApplication。注意DelegateApplication只是一个称谓，并没有一个叫DelegateApplication的基类存在。

使用Proxy/Delegate Application框架，使用者可以在对原有Application类不做任何修改的情况下，改变整个应用的运行环境。所需要做的只是添加一个新的Application类，并相应的修改AndroidManifest.xml。

老的AndroidManifest.xml：

```
<application
android:name=".MyApplication"
android:icon="@drawable/icon"
android:label="@string/app_name" >
```

添加的Application类：

```
public class MyProxyApplication extends ProxyApplication {
    @Override
    protected void initProxyApplication() {
        // 在这里替换运行环境，如将ClassLoader替换为自定义的
        // ......
    }
}
```

新的AndroidManifest.xml：

```
<application
android:name=".MyProxyApplication"
android:icon="@drawable/icon"
android:label="@string/app_name" >
    <meta-data
    android:name="DELEGATE_APPLICATION_CLASS_NAME"
    android:value=".MyApplication" >
    </meta-data>
```

MyProxyApplication（ProxyApplication）对象对应用是不可见的，应用看到的Application是

MyApplication（DelegateApplication），也就是以前的Application对象。这样对于应用而已，似乎一切都没有改变；但它的运行环境已经改变，例如所有的类已经被新的ClassLoader加载了。整个实现是非侵入式的，已有代码无须任何修改，只有AndroidManifest.xml略有改动。

下面开始探讨ProxyApplication本身如何实现。核心问题是两个，一是什么时机调用子类的initProxyApplication()方法，让子类改变surrounding；二是如何加载DelegateApplication并让应用认为它就是真实的Application。另外Android四大组件之一的ContentProvider会给我们带来不少麻烦，需要妥善处理。

ProxyApplication实现：时机

理论上ProxyApplication对任何能够访问到的变量，包括Java层和Native层，都是可以替换（或者HOOK，类似的含义）的；比较有意义的除了ClassLoader外，还有Resources和各路Binder对象。通过这些手段可以实现非常多有意思的功能。具体如何替换ClassLoader、Resources等这里不深入讨论，如有兴趣，在网上可以找到很多相关资料。本文的重点是介绍框架本身，替换ClassLoader仅作为一个例子。

现在的问题是改变surrounding的时机必须足够早，特别是对于ClassLoader来说尤为重要。是否可以在Application:onCreate()里做？我们通常认为，Application是一个Android应用最早被加载的组件；但当应用注册有ContentProvider的时候，这并不正确的。ContentProvider:onCreate()调用优先于Application:onCreate()。

幸好，我们还有另一个方法：attachBaseContext()。Android的几个主要顶级组件（Application、Activity、Service）都是ContextWrapper的子类。ContextWrapper一方面继承(inherit)了Context，一方面又包含(composite)了一个Context对象（称为mBase），对Context的实现为转发给mBase对象处理。这一个听起来很绕的设计，是为了对这些顶级组件中的Context功能做延迟初始化(delay init)的处理。这里不展开讨论了，仅贴一些Android源代码片段做参考。

```
01 // android.app.Application
02 public class Application extends ContextWrapper {
03     // ...
04     public application() {
05         super(null);
06     }
07     // ...
08 }
09 // android.content.ContextWrapper
10 public class ContextWrapper extends Context {
11     Context mBase;
12     // ...
13     public ContextWrapper(Context base) {
14         mBase = base;
15     }
16     protected void attachBaseContext(Context base) {
17         if (mBase != null) {
18             throw newIllegalStateException("Base context already set");
19         }
20         mBase = base;
21     }
22     // ...
23     @Override
24     public AssetManager getAssets() {
25         return mBase.getAssets();
26     }
27     @Override
28     public Resources getResources()
29     {
30         return mBase.getResources();
31     }
32     // ...
33 }
```

ContextWrapper完成这个delay init语义的方法就是attachBaseContext()。可以这样说，Application对象在刚刚构

造完成时是“残废”的，访问所有Context的方法都会抛出NullPointerException。只有attachBaseContext()执行完后，它的功能才完整。

在ContentProvider:onCreate()中，我们知道Application:onCreate()还没有运行，但已经可以使用getContext().getApplicationContext()函数获取Application对象，并访问其Context方法。显然，Android的API设计者不能允许此时获取的Application是“残废”的。结论是Application:attachBaseContext()必须要发生在ContentProvider:onCreate()之前，否则API将出现BUG；无论Android的系统版本如何变化，这一点也不能改变。

于是，Application与ContentProvider的初始化次序是这样的：Application:attachBaseContext()最早执行，然后是ContentProvider:onCreate()，然后是Application:onCreate()。我们的解决方案也就很简单了：

```
public abstract class ProxyApplication extendsApplication {
    protected abstract void initProxyApplication();
    @Override
    protected void attachBaseContext (Context context) {
        super.attachBaseContext(context);
        initProxyApplication();
    }
    // ……
}
```

ProxyApplication实现：加载**DelegateApplication**

当子类的initProxyApplication()返回后，ProxyApplication就要加载DelegateApplication，完成自己的历史使命。这一部分在onCreate()中完成，基本是些体力活，但也有些需要注意的地方，下面分步骤简述一下。

(1) 获取**DelegateApplication**的**Class Name**

即从AndroidManifest.xml中获取DELEGATE_APPLICAION的metadata值，若不存在，则使用android.app.Application作为默认。这一步比较简单。

```
String className = "android.app.Application";
String key = "DELEGATE_APPLICATION_CLASS_NAME";
ApplicationInfo appInfo = getPackageManager().getApplicationInfo(
    super.getPackageName(), PackageManager.GET_META_DATA);
Bundle bundle = appInfo.metaData;
if (bundle != null && bundle.containsKey(key)) {
    className = bundle.getString(key);
    if (className.startsWith("."))
        className = super.getPackageName() + className;
}
```

(2) 加载**DelegateApplication**并生成对象

这里要注意的是使用哪个ClassLoader？答案是应该用getClassLoader()（即Context:getClassLoader()），而不是getClass().getClassLoader()。要仔细揣摩这两者之间的差别。

```
Class delegateClass = Class.forName(className, true, getClassLoader());
Application delegate = (Application) delegateClass.newInstance();
```

(3) 替换**API**层的所有**Application**引用

即把API层所有保存的ProxyApplication对象，都替换为新生成的DelegateApplication对象。以ProxyApplication的baseContext作为起点顺藤摸瓜，可以找到所有的位置，使用反射一一换掉。注意最后一个mAllApplications是List，要换掉其内部的内容。

```
baseContext.mOuterContext
baseContext.mPackageInfo.mApplication
baseContext.mPackageInfo.mActivityThread.mInitialApplication
baseContext.mPackageInfo.mActivityThread.mAllApplications
```

(4) 设置baseContext并调用onCreate

将控制权交给DelegateApplication。当然，后者会认为自己就是“正牌”的Application，后续的其它组件也都会这么认为。这正是我们要的效果。

```
Method attach = Application.class.getDeclaredMethod("attach", Context.class);
attach.setAccessible(true);
attach.invoke(delegate, base);
delegate.onCreate();
```

再次对付**ContentProvider**

前面提到过，Android的顶级组件Application、Activity、Service都是ContextWrapper，这个列表中并没有ContentProvider。ContentProvider不是ContextWrapper，甚至不是Context，而是内部有一个mContext变量，通过getContext()函数获取这个Context。

那么，ContentProvider:getContext()获取到的是哪一个Context？实验证明，ContentProvider:getContext()获取的Context是Application；准确的说，在Proxy/Delegate Application框架里，是ProxyApplication。这就不符合框架的语义了。那么，我们需要像其它处理其它ProxyApplication引用一样，把它换成DelegateApplication吗？这是可行的：遍历API层的ContentProvider列表，将每一个ContentProvider中的mContext都替换为DelegateApplication。

但这种处理方式，会进一步增加对Android API层源代码依赖，是否必要？毕竟Android的API文档中，并没有规定ContentProvider:getContext()返回的必须是Application；如果要取得Application，正确的方式是getContext().getApplicationContext()。那么为什么getContext()就直接返回了Application对象？我们可以从源代码中找到答案：

```
// in ActivityThread:installProvider()
if (context.getPackageName().equals(ai.packageName)) {
    c = context;
} else if (mInitialApplication != null &&
        mInitialApplication.getPackageName().equals(ai.packageName)) {
    c = mInitialApplication;
} else {
    try {
        c = context.createPackageContext(ai.packageName,
                Context.CONTEXT_INCLUDE_CODE);
    } catch (PackageManager.NameNotFoundException e) {
        // Ignore
    }
}
```

容易看出，因为ProxyApplication对象的getPackageName()函数与ContentProvider对应的包名相同，就会复用ProxyApplication对象作为Context，而不会再创建一个新的packageContext。于是解决方案也很简单了：

```
@Override
public String getPackageName() {
    return "";
}
```

由于ProxyApplication不是最终的Application，这并不会产生什么副作用。

使用注意事项

不要保留ProxyApplication子类对象的引用，也不要在任何系统回调（包括onCreate）中做事情。onCreate()被基类用于加载DelegateApplication，而其它回调都不会再收到。

在ProxyApplication:onCreate()执行完成之后，虚拟机中所有的线程栈和所有的JAVA对象，都不会再有ProxyApplication对象的引用。ProxyApplication对象将在下一次GC运行时被回收，这也意味着从

ProxyApplication到DelegateApplication的替换进行得非常彻底。自然地，ProxyApplication也收不到其它回调了。DelegateApplication会正常的接收所有的回调。

另外，在ProxyApplication子类中，如果需要获取当前APK的包名，需要使用getBaseContext().getPackageName()，而不能简单调用getPackageName()。原因在上面“再次对付ContentProvider”中有说明。

你也许会喜欢：

- Android APP通用型拒绝服务漏洞分析报告
- FakeID签名漏洞分析及利用(Google Bug 13678484)
- launchAnyWhere: Activity组件权限绕过漏洞解析(Google Bug 7699048)
- Hacking Team攻击代码分析Part5: Adobe Font Driver内核权限提升漏洞第二弹+Win32k KALSR绕过漏洞
- 浅谈Android应用性能之内存