动态加载技术（也叫插件化技术）在技术驱动型的公司中扮演着相当重要的角色，当项目越来越庞大的时候，需要通过插件化来减轻应用的内存和CPU占用，还可以实现热插拔，即在不发布新版本的情况下更新某些模块。动态加载是一项很复杂的技术，这里主要介绍动态加载技术中的三个基础性问题，至于完整的动态加载技术的实现请参考笔者发起的开源插件化框架DL

不同的插件化方案各有各的特色，但是它们都必须要解决三个基础性问题：资源访问、Activity生命周期的管理和ClassLoader的管理。 在介绍它们之前，首先要明白宿主和插件的概念，宿主是指普通的apk，而插件一般是指经过处理的dex或者apk，在主流的插件化框架中多采用经过特殊处 理的apk来作为插件，处理方式往往和编译以及打包环节有关，另外很多插件化框架都需要用到代理Activity的概念，插件Activity的启动大多 数是借助一个代理Activity来实现的。

1．资源访问

我们知道，宿主程序调起未安装的插件apk，一个很大的问题就是资源如何访问，具体来说就是插件中凡是以R开头的资源都不能访问了。这是因为宿主程序中并没有插件的资源，所以通过R来加载插件的资源是行不通的，程序会抛出异常：无法找到某某id所对应的资源。

针对这个问题，有人提出了将插件中的资源在宿主程序中也预置一份，这虽然能解决问题，但是这样就会产生一些弊端。首先，这样就需要宿主和插件同时持 有一份相同的资源，增加了宿主apk的大小；其次，在这种模式下，每次发布一个插件都需要将资源复制到宿主程序中，这意味着每发布一个插件都要更新一下宿 主程序，这就和插件化的思想相违背了。

因为插件化的目的就是要减小宿主程序apk包的大小，同时降低宿主程序的更新频率并做到自由装载模块，所以这种方法不可取，它限制了插件的线上更新 这一重要特性。还有人提供了另一种方式，首先将插件中的资源解压出来，然后通过文件流去读取资源，这样做理论上是可行的，但是实际操作起来还是有很大难度 的。首先不同资源有不同的文件流格式，比如图片、XML等，其次针对不同设备加载的资源可能是不一样的，如何选择合适的资源也是一个需要解决的问题，基于这两点，这种方法也不建议使用，因为它实现起来有较大难度。为了方便地对插件进行资源管理，下面给出一种合理的方式。

我们知道，Activity的工作主要是通过ContextImpl来完成的， Activity中有一个叫mBase的成员变量，它的类型就是ContextImpl。注意到Context中有如下两个抽象方法，看起来是和资源有关 的，实际上Context就是通过它们来获取资源的。这两个抽象方法的真正实现在ContextImpl中，也就是说，只要实现这两个方法，就可以解决资源问题了。

```
/** Return an AssetManager instance for your application's package. */
public abstract AssetManager getAssets();
/** Return a Resources instance for your application's package. */
public abstract Resources getResources();
```

下面给出具体的实现方式，首先要加载apk中的资源，如下所示

```
protected void loadResources() {
    try {
        AssetManager assetManager = AssetManager.class.newInstance();
        Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", String.class);
        addAssetPath.invoke(assetManager, mDexPath);
        mAssetManager = assetManager;
    } catch (Exception e) {
        e.printStackTrace();
    }
    Resources superRes = super.getResources();
    mResources = new Resources(mAssetManager, superRes.getDisplayMetrics(),
            superRes.getConfiguration());
    mTheme = mResources.newTheme();
    mTheme.setTo(super.getTheme());
}
```

从loadResources()的实现可以看出，加载资源的方法是通过反射，通过调用AssetManager中的addAssetPath方 法，我们可以将一个apk中的资源加载到Resources对象中，由于addAssetPath是隐藏API我们无法直接调用，所以只能通过反射。下面 是它的声明，通过注释我们可以看出，传递的路径可以是zip文件也可以是一个资源目录，而apk就是一个zip，所以直接将apk的路径传给它，资源就加 载到AssetManager中了。然后再通过AssetManager来创建一个新的Resources对象，通过这个对象我们就可以访问插件apk中 的资源了，这样一来问题就解决了。

```
/**
 * Add an additional set of assets to the asset manager.  This can be
 * either a directory or ZIP file.  Not for use by applications.  Returns
 * the cookie of the added asset, or 0 on failure.
 * {@hide}
 */

public final int addAssetPath(String path) {
    synchronized (this) {
        int res = addAssetPathNative(path);
        makeStringBlocks(mStringBlocks);
        return res;
    }
}
```

接着在代理Activity中实现getAssets()和getResources()，如下所示。关于代理Activity的含义请参看DL开源插件化框架的实现细节，这里不再详细描述了。

```
@Override
public AssetManager getAssets() {
    return mAssetManager == null ? super.getAssets() : mAssetManager;
}

@Override
public Resources getResources() {
    return mResources == null ? super.getResources() : mResources;
}
```

通过上述这两个步骤，就可以通过R来访问插件中的资源了。

2．Activity生命周期的管理

管理Activity生命周期的方式各种各样，这里只介绍两种：反射方式和接口方式。反射的方式很好理解，首先通过Java的反射去获取 Activity的各种生命周期方法，比如onCreate、onStart、onResume等，然后在代理Activity中去调用插件 Activity对应的生命周期方法即可，如下所示。

```
@Override
protected void onResume() {
    super.onResume();
    Method onResume = mActivityLifecircleMethods.get("onResume");
    if (onResume != null) {
        try {
            onResume.invoke(mRemoteActivity, new Object[] { });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

@Override
protected void onPause() {
    Method onPause = mActivityLifecircleMethods.get("onPause");
    if (onPause != null) {
        try {
            onPause.invoke(mRemoteActivity, new Object[] { });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    super.onPause();
}
```

使用反射来管理插件Activity的生命周期是有缺点的，一方面是反射代码写起来比较复杂，另一方面是过多使用反射会有一定的性能开销。下面介绍 接口方式，接口方式很好地解决了反射方式的不足之处，这种方式将Activity的生命周期方法提取出来作为一个接口（比如叫DLPlugin），然后通 过代理Activity去调用插件Activity的生命周期方法，这样就完成了插件Activity的生命周期管理，并且没有采用反射，这就解决了性能 问题。同时接口的声明也比较简单，下面是DLPlugin的声明：

```
public interface DLPlugin {
    public void onStart();
    public void onRestart();
    public void onActivityResult(int requestCode, int resultCode, Intent data);
    public void onResume();
    public void onPause();
    public void onStop();
    public void onDestroy();
    public void onCreate(Bundle savedInstanceState);
    public void setProxy(Activity proxyActivity, String dexPath);
    public void onSaveInstanceState(Bundle outState);
    public void onNewIntent(Intent intent);
    public void onRestoreInstanceState(Bundle savedInstanceState);
    public boolean onTouchEvent(MotionEvent event);
    public boolean onKeyUp(int keyCode, KeyEvent event);
    public void onWindowAttributesChanged(LayoutParams params);
    public void onWindowFocusChanged(boolean hasFocus);
    public void onBackPressed();
    …
}
```

在代理Activity中只需要按如下方式即可调用插件Activity的生命周期方法，这就完成了插件Activity的生命周期的管理。

```
...
@Override
protected void onStart() {
    mRemoteActivity.onStart();
    super.onStart();
}

@Override
protected void onRestart() {
    mRemoteActivity.onRestart();
    super.onRestart();
}

@Override
protected void onResume() {
    mRemoteActivity.onResume();
    super.onResume();
}
...
```

通过上述代码应该不难理解接口方式对插件Activity生命周期的管理思想，其中mRemoteActivity就是DLPlugin的实现。

3．插件ClassLoader的管理

为了更好地对多插件进行支持，需要合理地去管理各个插件的DexClassLoader，这样同一个插件就可以采用同一个ClassLoader去 加载类，从而避免了多个ClassLoader加载同一个类时所引发的类型转换错误。在下面的代码中，通过将不同插件的ClassLoader存储在一个 HashMap中，这样就可以保证不同插件中的类彼此互不干扰。

```java
public class DLClassLoader extends DexClassLoader {
    private static final String TAG = "DLClassLoader";

    private static final HashMap<String, DLClassLoader> mPluginClassLoaders
    = new HashMap<String, DLClassLoader>();

    protected DLClassLoader(String dexPath, String optimizedDirectory, String libraryPath,
    ClassLoader parent) {
        super(dexPath, optimizedDirectory, libraryPath, parent);
    }

    /**
     * return a available classloader which belongs to different apk
     */
    public static DLClassLoader getClassLoader(String dexPath, Context
    context, ClassLoader parentLoader) {
        DLClassLoader dLClassLoader = mPluginClassLoaders.get(dexPath);
        if (dLClassLoader != null)
            return dLClassLoader;

        File dexOutputDir = context.getDir("dex", Context.MODE_PRIVATE);
        final String dexOutputPath = dexOutputDir.getAbsolutePath();
        dLClassLoader = new DLClassLoader(dexPath, dexOutputPath, null,
        parentLoader);
        mPluginClassLoaders.put(dexPath, dLClassLoader);

        return dLClassLoader;
    }
}
```

事实上插件化的技术细节非常多，这绝非一个章节的内容所能描述清楚的，另外插件化作为一种核心技术，需要开发者有较深的开发功底才能够很好地理解， 因此本节的内容更多是让读者对插件化开发有一个感性的了解，细节上还需要读者自己去钻研，也可以通过DL插件化框架去深入地学习。