Android 类加载机制

一．**Java中的ClassLoader**

Java中的类加载器主要有两种类型，系统类加载和自定义类加载器。其中系统类加载器包括3种，分别是Bootstrap ClassLoader、 Extensions ClassLoader和 App ClassLoader。

1. Bootstrap ClassLoader

用C/C++代码实现的加载器，用于加载Java虚拟机运行时所需要的系统类，如java.lang.\*、java.uti.\*等这些系统类，它们默认在$JAVA\_HOME/jre/lib目录中，也可以通过启动Java虚拟机时指定-Xbootclasspath选项，来改变Bootstrap ClassLoader的加载目录。   
Java虚拟机的启动就是通过 Bootstrap ClassLoader创建一个初始类来完成的。由于Bootstrap ClassLoader是使用C/C++语言实现的， 所以该加载器不能被Java代码访问到。需要注意的是Bootstrap ClassLoader并不继承java.lang.ClassLoader。 Bootstrap ClassLoader的加载目录都是$JAVA\_HOME/jre/lib目录中的jar包，包括rt.jar、resources.jar和charsets.jar等等。

1. Extensions ClassLoader

用于加载 Java 的拓展类 ，拓展类的jar包一般会放在$JAVA\_HOME/jre/lib/ext目录下，用来提供除了系统类之外的额外功能。也可以通过-Djava.ext.dirs选项添加和修改Extensions ClassLoader加载的路径。

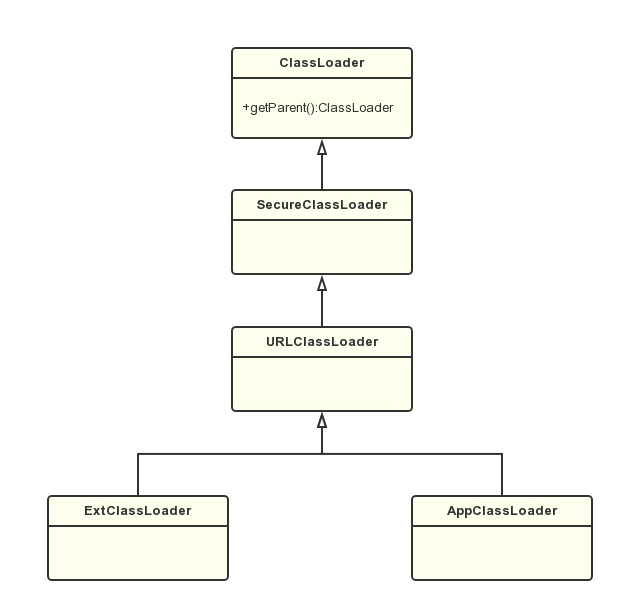
1. App ClassLoader

负责加载当前应用程序Classpath目录下的所有jar和Class文件。也可以加载通过-Djava.class.path选项所指定的目录下的jar和Class文件。

1. Custom ClassLoader

除了系统提供的类加载器，还可以自定义类加载器，自定义类加载器通过继承java.lang.ClassLoader类的方式来实现自己的类加载器，除了 Bootstrap ClassLoader，Extensions ClassLoader和App ClassLoader也继承了java.lang.ClassLoader类。

1. ClassLoader的继承关系

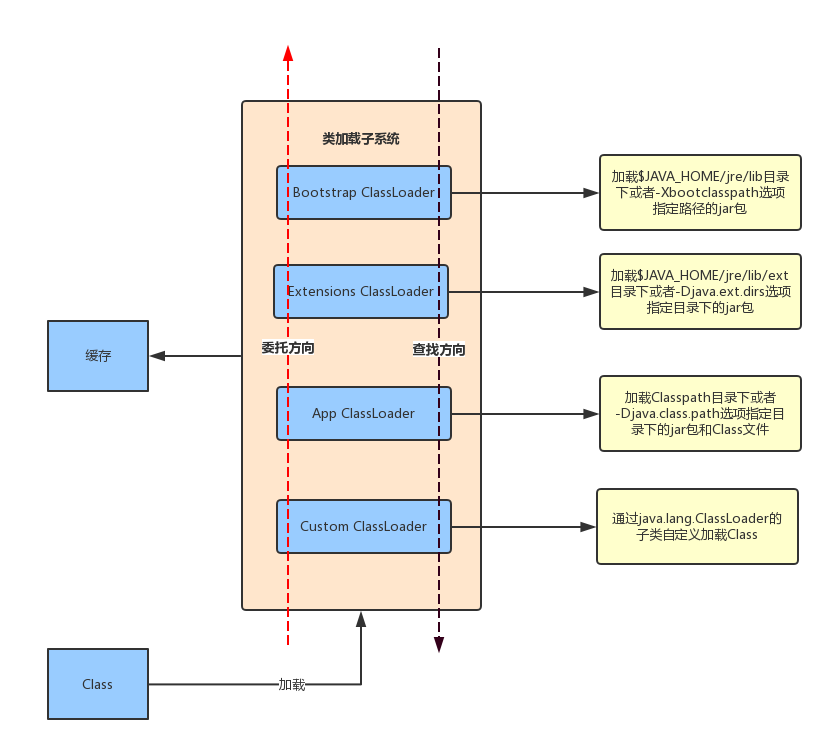


可以看到上图中共有5个ClassLoader相关类，下面简单对它们进行介绍：

* [ClassLoader](http://www.grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/lang/ClassLoader.java#ClassLoader)是一个抽象类，其中定义了ClassLoader的主要功能。
* SecureClassLoader继承了抽象类ClassLoader，但SecureClassLoader并不是ClassLoader的实现类，而是拓展了ClassLoader类加入了权限方面的功能，加强了ClassLoader的安全性。
* URLClassLoader继承自SecureClassLoader，用来通过URl路径从jar文件和文件夹中加载类和资源。
* ExtClassLoader和AppClassLoader都继承自URLClassLoader，它们都是[Launcher](http://www.grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/sun/misc/Launcher.java" \l "Launcher.AppClassLoader" \t "_blank)的内部类，Launcher 是Java虚拟机的入口应用，ExtClassLoader和AppClassLoader都是在Launcher中进行初始化的。

1. 双亲委派模式

类加载器查找Class所采用的是双亲委托模式，所谓双亲委托模式就是首先判断该Class是否已经加载，如果没有则不是自身去查找而是委托给父加载器进行查找，这样依次的进行递归，直到委托到最顶层的Bootstrap ClassLoader，如果Bootstrap ClassLoader找到了该Class，就会直接返回，如果没找到，则继续依次向下查找，如果还没找到则最后会交由自身去查找。



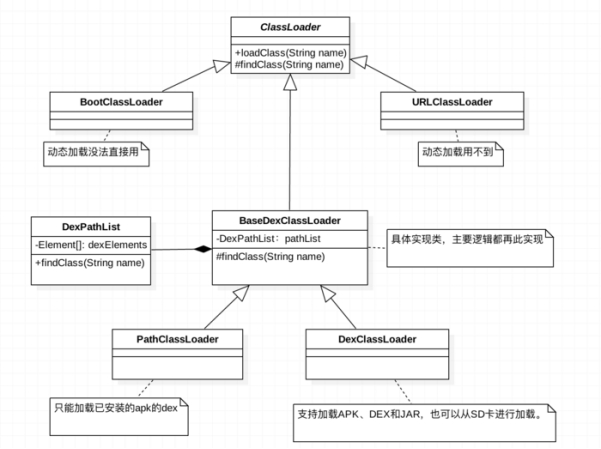
假设我们要加载一个位于D盘的Class文件，这时系统所提供的类加载器不能满足条件，这时就需要我们自定义类加载器继承自java.lang.ClassLoader，并复写它的findClass方法。加载D盘的Class文件步骤如下：

1. 自定义类加载器首先从缓存中要查找Class文件是否已经加载，如果已经加载就返回该Class，如果没加载则委托给父加载器也就是App ClassLoader。
2. 按照上图中红色虚线的方向递归步骤1。
3. 一直委托到Bootstrap ClassLoader，如果Bootstrap ClassLoader在缓存中还没有查找到Class文件，则在自己的规定路径$JAVA\_HOME/jre/libr中或者-Xbootclasspath选项指定路径的jar包中进行查找，如果找到则返回该Class，如果没有则交给子加载器Extensions ClassLoader。
4. Extensions ClassLoader查找$JAVA\_HOME/jre/lib/ext目录下或者-Djava.ext.dirs选项指定目录下的jar包，如果找到就返回，找不到则交给App ClassLoader。
5. App ClassLoader找Classpath目录下或者-Djava.ext.dirs选项所指定的目录下的jar包和Class文件，如果找到就返回，找不到交给我们自定义的类加载器，如果还找不到则抛出异常。

总的来说就是Class文件加载到类加载子系统后，先沿着图中红色虚线的方向自下而上进行委托，再沿着黑色虚线的方向自上而下进行查找，整个过程就是先上后下。

二． **Android中的ClassLoader**

Dalvik虚拟机如同其他Java虚拟机一样，在运行程序时首先需要将对应的类加载到内存中。而在Java标准的虚拟机中，类加载可以从class文件中读取，只不过Android平台上虚拟机运行的是Dex字节码,传统Class文件是一个Java源码文件会生成一个.class文件，而Android是把所有Class文件进行合并，优化，然后生成一个最终的class.dex,目的是把不同class文件重复的东西只需保留一份,如果我们的Android应用不进行分dex处理,最后一个应用的apk只会有一个dex文件。

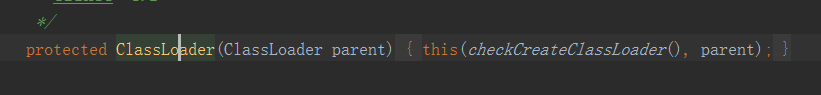


Android中类加载器有BootClassLoader, URLClassLoader, PathClassLoader, DexClassLoader, BaseDexClassLoader,等都最终继承自java.lang.ClassLoader。

1. ClassLoader

java.lang.ClassLoader是所有ClassLoader的最终父类。构造方法主要以下两种:

(1)传入一个父类构造器



(2)无参默认构造法



可以看出ClassLoader主要就是传入一个父构造器，而且一般父构造器不能为空，不像java虚拟机里父构造器为空时默认的父构造器为Bootstrap ClassLoader。Android中默认无父构造器传入的情况下，默认父构造器为一个PathClassLoader且此PathClassLoader父构造器为BootClassLoader。

ClassLoader中重要的方法是loadClass(String name)，其他的子类都继承了此方法且没有进行复写。

加载类时首先判断这个类是否之前被加载过，如果有则直接返回，如果没有则首先尝试让parent ClassLoader进行加载,加载不成功才在自己的findClass中进行加载。这和java虚拟机中常见的双亲委派模型一致的，这种模型并不是一个强制性的约束模型，比如你可以继承ClassLoader复写loadCalss方法来破坏这种模型，只不过双亲委派模是一种被推荐的实现类加载器的方式，而且jdk1.2以后已经不提倡用户在覆盖loadClass方法，而应该把自己的类加载逻辑写到findClass中。

2. BootClassLoader

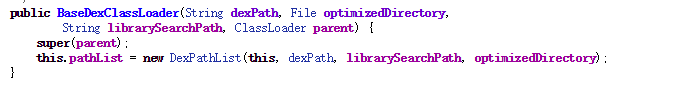
Android系统启动时会使用BootClassLoader来预加载常用类，和java虚拟机中不同的是BootClassLoader是ClassLoader内部类,由java代码实现而不是c++实现,是Android平台上所有ClassLoader的最终parent,这个内部类是包内可见,所以我们没法使用。

3. URLClassLoader

只能用于加载jar文件，但是由于 dalvik 不能直接识别jar，所以在 Android 中无法使用这个加载器。

4. BaseDexClassLoader

PathClassLoader和DexClassLoader都继承自BaseDexClassLoader,其中的主要逻辑都是在BaseDexClassLoader完成的。



BaseDexClassLoader的构造函数包含四个参数，分别为：

· dexPath,指目标类所在的APK或jar文件的路径,类装载器将从该路径中寻找指定的目标类,该类必须是APK或jar的全路径.如果要包含多个路径,路径之间必须使用特定的分割符分隔,特定的分割符可以使用System.getProperty(“path.separtor”)获得。上面"支持加载APK、DEX和JAR，也可以从SD卡进行加载"指的就是这个路径，最终做的是将dexPath路径上的文件ODEX优化到内部位置optimizedDirectory，然后，再进行加载的。

· File optimizedDirectory,由于dex文件被包含在APK或者Jar文件中,因此在装载目标类之前需要先从APK或Jar文件中解压出dex文件,该参数就是制定解压出的dex 文件存放的路径。这也是对apk中dex根据平台进行ODEX优化的过程。其实APK是一个程序压缩包，里面包含dex文件，ODEX优化就是把包里面的执行程序提取出来，就变成ODEX文件，因为你提取出来了，系统第一次启动的时候就不用去解压程序压缩包的程序，少了一个解压的过程。这样的话系统启动就加快了。为什么说是第一次呢？是因为DEX版本的也只有第一次会解压执行程序到 /data/dalvik-cache（针对PathClassLoader）或者optimizedDirectory(针对DexClassLoader）目录，之后也是直接读取目录下的的dex文件，所以第二次启动就和正常的差不多了。当然这只是简单的理解，实际生成的ODEX还有一定的优化作用。ClassLoader只能加载内部存储路径中的dex文件，所以这个路径必须为内部路径。

· libPath,指目标类中所使用的C/C++库存放的路径

· classload,是指该装载器的父装载器,一般为当前执行类的装载器，例如在Android中以context.getClassLoader()作为父装载器。

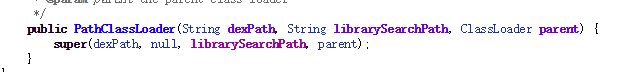
5. DexClassLoader



对于DexClassLoader来说，参数optimizedDirectory是至关重要的，我们知道应用程序第一次被加载的时候，为了提高以后的启动速度和执行效率，Android系统会对dex相关文件做一定程度的优化，并生成一个ODEX文件，此后再运行这个应用程序的时候，只要加载优化过的ODEX文件就行了，省去了每次都要优化的时间，而参数optimizedDirectory就是代表存储ODEX文件的路径，这个路径必须是一个内部存储路径。

DexClassLoader支持加载APK、DEX和JAR，也可以从SD卡进行加载。  
上面说dalvik不能直接识别jar,DexClassLoader却可以加载jar文件,这难道不矛盾吗?其实在BaseDexClassLoader里对".jar",".zip",".apk",".dex"后缀的文件最后都会生成一个对应的dex文件,所以最终处理的还是dex文件,而URLClassLoader并没有做类似的处理。  
一般我们都是用这个DexClassLoader来作为动态加载的加载器。它是热修复和插件化技术的基础。

6. PathClassLoader



Android系统使用PathClassLoader来加载系统类和应用程序的类，如果是加载非系统应用程序类，则会加载data/app/目录下的dex文件以及包含dex的apk文件或jar文件。

很简单明了，可以看出PathClassLoader没有将optimizedDirectory置为Null,也就是没设置优化后的存放路径。其实optimizedDirectory为null时的默认路径就是/data/dalvik-cache 目录。  
PathClassLoader继承自BaseDexClassLoader，很明显PathClassLoader的方法实现都在BaseDexClassLoader中。从PathClassLoader的构造方法也可以看出它遵循了双亲委托模式

总结：首先android系统启动时，使用BootClassLoader来预加载一些常用的系统类，然后再使用PathClassLoader去加载安装的应用生成的位于/data/dalvik-cache目录下得dex文件来启动应用。BootClassLoader是在Zygote进程的入口方法中创建的，PathClassLoader则是在Zygote进程创建SystemServer进程时创建的。另外，DexClassLoader可以加载指定位置的dex相关文件，所以一般用来进行热修复。