ClassLoader类加载

ART 和 Dalvik

DVM也是实现了**JVM**规范的一个虚拟器，默认使用CMS垃圾回收器，但是与JVM运行 Class 字节码不同，DVM 执行 Dex(Dalvik Executable Format)** ——专为 Dalvik 设计的一种压缩格式。Dex 文件是很多 .class 文件处理压缩后的产物，最终可以在 Android 运行时环境执行。

而**ART（Android Runtime）** 是在 Android 4.4 中引入的一个开发者选项，也是 Android 5.0 及更高版本的默认 Android 运行时。ART 和 Dalvik 都是运行 Dex 字节码的兼容运行时，因此针对 Dalvik 开发的应用也能在 ART 环境中运作。

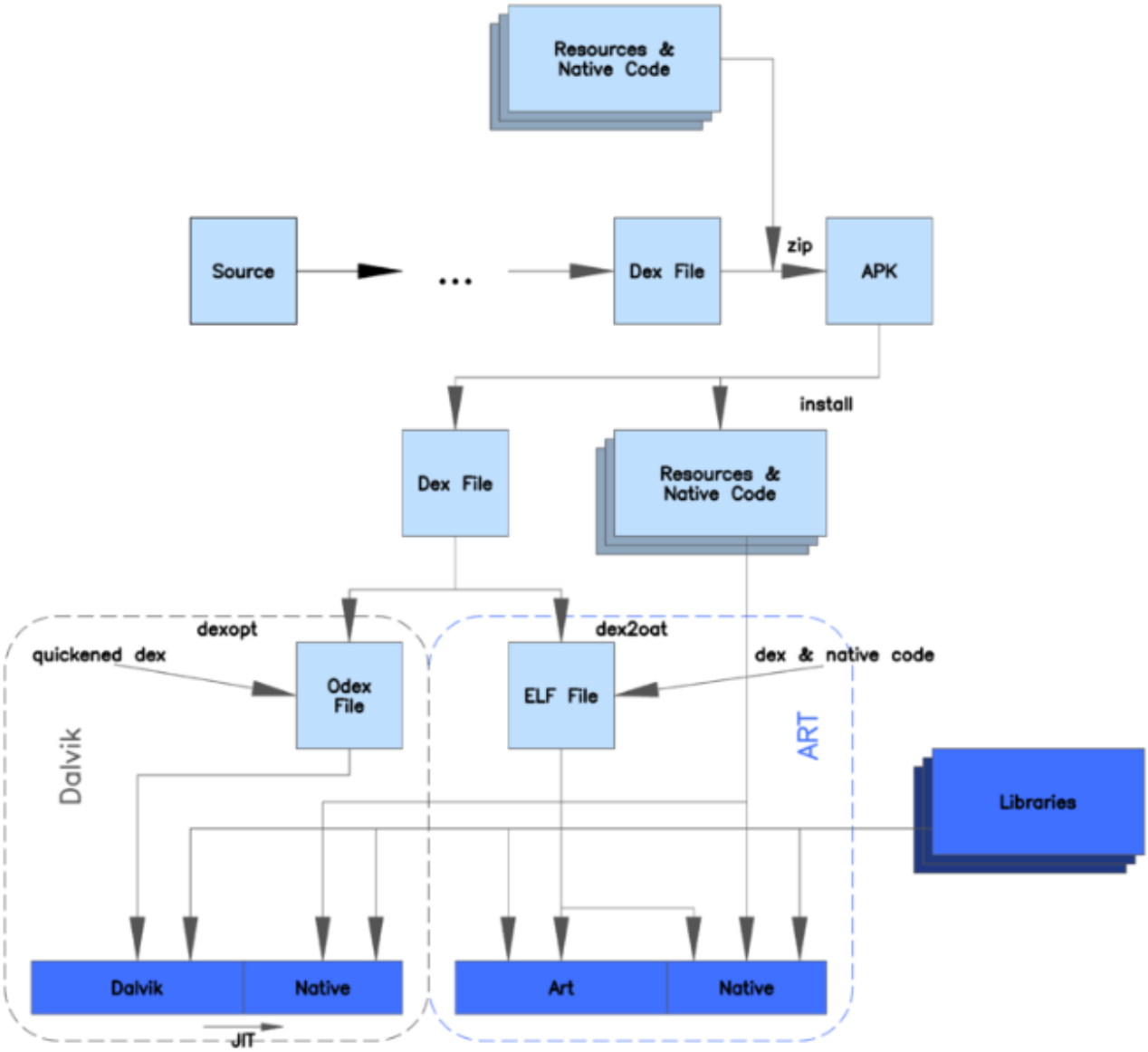
dexopt与dexaot

- **dexopt**

  在**Dalvik**中虚拟机在加载一个dex文件时，对 dex 文件 进行 验证 和 优化的操作，其对 dex 文件的优化结果变成了 odex(Optimized dex) 文件，这个文件和 dex 文件很像，只是使用了一些优化操作码。

- **dex2oat**

  ART 预先编译机制，在安装时对 dex 文件执行AOT 提前编译操作，编译为OAT（实际上是ELF文件）可执行文件（机器码）。



ClassLoader介绍

任何一个 Java 程序都是由一个或多个 class 文件组成，在程序运行时，需要将 class 文件加载到 JVM 中才可以使用，负责加载这些 class 文件的就是 Java 的类加载机制。ClassLoader 的作用简单来说就是加载 class 文件，提供给程序运行时使用。每个 Class 对象的内部都有一个 classLoader 字段来标识自己是由哪个 ClassLoader 加载的。

```
class Class<T> {
  ...
  private transient ClassLoader classLoader;
  ...
}
```

ClassLoader是一个抽象类，而它的具体实现类主要有：

- `BootClassLoader`

  用于加载Android Framework层class文件。

- `PathClassLoader`

  用于Android应用程序类加载器。可以加载指定的dex，以及jar、zip、apk中的classes.dex
- `DexClassLoader`

  用于加载指定的dex，以及jar、zip、apk中的classes.dex

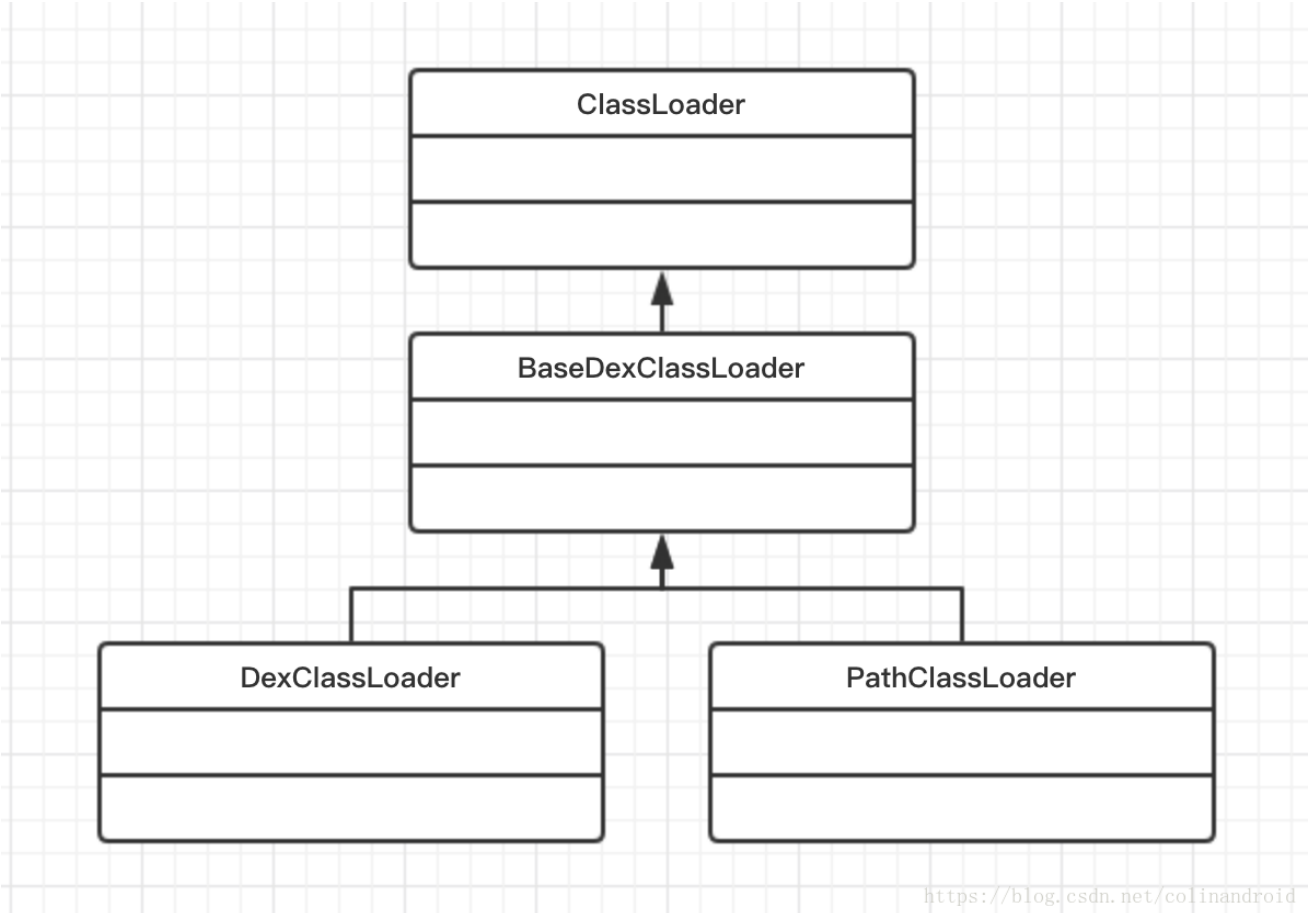
> 很多博客里说PathClassLoader只能加载已安装的apk的dex，其实这说的应该是在dalvik虚拟机上。但现在一般不用关心dalvik了。

```
Log.e(TAG, "Activity.class 由: " + Activity.class.getClassLoader() +" 加载");
Log.e(TAG, "MainActivity.class 由: " + getClassLoader() +" 加载");


//输出:
Activity.class 由: java.lang.BootClassLoader@d3052a9 加载

MainActivity.class 由: dalvik.system.PathClassLoader[DexPathList[[zip file
"/data/app/com.enjoy.enjoyfix-1/base.apk"],nativeLibraryDirectories=
[/data/app/com.enjoy.enjoyfix-1/lib/x86, /system/lib, /vendor/lib]]] 加载
```

它们之间的关系如下：



`PathClassLoader`与`DexClassLoader`的共同父类是`BaseDexClassLoader`。

```
public class DexClassLoader extends BaseDexClassLoader {

    public DexClassLoader(String dexPath, String optimizedDirectory,
        String librarySearchPath, ClassLoader parent) {
        super(dexPath, new File(optimizedDirectory), librarySearchPath, parent);
    }
}

public class PathClassLoader extends BaseDexClassLoader {

    public PathClassLoader(String dexPath, ClassLoader parent) {
        super(dexPath, null, null, parent);
    }

    public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader parent){
         super(dexPath, null, librarySearchPath, parent);
    }
}
```

可以看到两者唯一的区别在于：创建`DexClassLoader`需要传递一个`optimizedDirectory`参数，并且会将其创建为`File`对象传给`super`，而`PathClassLoader`则直接给到null。因此两者都可以加载**指定的dex，以及jar、zip、apk中的classes.dex**

```
PathClassLoader pathClassLoader = new PathClassLoader("/sdcard/xx.dex", getClassLoader());

File dexOutputDir = context.getCodeCacheDir();
DexClassLoader dexClassLoader = new
DexClassLoader("/sdcard/xx.dex",dexOutputDir.getAbsolutePath(), null,getClassLoader());
```

其实，`optimizedDirectory`参数就是dexopt的产出目录(odex)。那`PathClassLoader`创建时，这个目录为null，就意味着不进行dexopt？并不是，`optimizedDirectory`为null时的默认路径为：***/data/dalvik-cache***。

> 在API 26源码中，将DexClassLoader的optimizedDirectory标记为了 deprecated 弃用，实现也变为了：
>
> ```
>  public DexClassLoader(String dexPath, String optimizedDirectory,
>                        String librarySearchPath, ClassLoader parent) {
>     super(dexPath, null, librarySearchPath, parent);
>  }
> ```
>
> ……和PathClassLoader一摸一样了！

双亲委托机制

可以看到创建`ClassLoader`需要接收一个`ClassLoader parent`参数。这个`parent`的目的就在于实现类加载的双亲委托。即：

某个类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

```
protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException{

    // 检查class是否有被加载
    Class c = findLoadedClass(name);
    if (c == null) {
        long t0 = System.nanoTime();
        try {
            if (parent != null) {
                //如果parent不为null，则调用parent的loadClass进行加载
                c = parent.loadClass(name, false);
            } else {
                //parent为null，则调用BootClassLoader进行加载
                c = findBootstrapClassOrNull(name);
            }
        } catch (ClassNotFoundException e) {

        }

        if (c == null) {
            // 如果都找不到就自己查找
            long t1 = System.nanoTime();
            c = findClass(name);
        }
    }
    return c;
}
```

> 因此我们自己创建的ClassLoader: `new PathClassLoader("/sdcard/xx.dex", getClassLoader());`并不仅仅只能加载 xx.dex中的class。

> 值得注意的是：`c = findBootstrapClassOrNull(name);`
>
> 按照方法名理解，应该是当parent为null时候，也能够加载`BootClassLoader`加载的类。
>
> `new PathClassLoader("/sdcard/xx.dex", null)`，能否加载Activity.class?
>
> 但是实际上，Android当中的实现为：（Java不同）
>
> ```
> private Class findBootstrapClassOrNull(String name)
>  {
>      return null;
>  }
> ```

findClass

可以看到在所有父ClassLoader无法加载Class时，则会调用自己的`findClass`方法。`findClass`在ClassLoader中的定义为:

```
protected Class<?> findClass(String name) throws ClassNotFoundException {
    throw new ClassNotFoundException(name);
}
```

其实任何ClassLoader子类，都可以重写`loadClass`与`findClass`。一般如果你不想使用双亲委托，则重写`loadClass`修改其实现。而重写`findClass`则表示在双亲委托下，父ClassLoader都找不到Class的情况下，定义自己如何去查找一个Class。而我们的`PathClassLoader`会自己负责加载`MainActivity`这样的程序中自己编写的类，利用双亲委托父ClassLoader加载Framework中的`Activity`。说明`PathClassLoader`并没有重写`loadClass`，因此我们可以来看看PathClassLoader中的`findClass`是如何实现的。

```
public BaseDexClassLoader(String dexPath, File optimizedDirectory,String
                        librarySearchPath, ClassLoader parent) {
    super(parent);
    this.pathList = new DexPathList(this, dexPath, librarySearchPath,
                                    optimizedDirectory);
}

@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
    //查找指定的class
    Class c = pathList.findClass(name, suppressedExceptions);
    if (c == null) {
        ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find class \"" +
name + "\" on path: " + pathList);
        for (Throwable t : suppressedExceptions) {
            cnfe.addSuppressed(t);
        }
            throw cnfe;
    }
    return c;
}
```

实现非常简单，从`pathList`中查找class。继续查看`DexPathList`

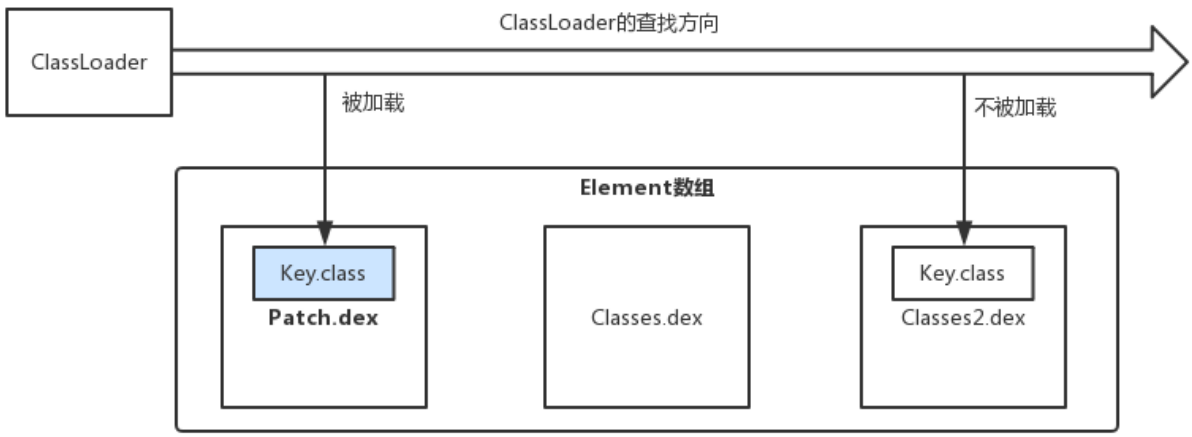
```
public DexPathList(ClassLoader definingContext, String dexPath,
            String librarySearchPath, File optimizedDirectory) {
    //.........
    // splitDexPath 实现为返回 List<File>.add(dexPath)
    // makeDexElements 会去 List<File>.add(dexPath) 中使用DexFile加载dex文件返回 Element数组
    this.dexElements = makeDexElements(splitDexPath(dexPath), optimizedDirectory,
                                       suppressedExceptions, definingContext);
    //.........

}
```

```
public Class findClass(String name, List<Throwable> suppressed) {
     //从element中获得代表Dex的 DexFile
    for (Element element : dexElements) {
        DexFile dex = element.dexFile;
        if (dex != null) {
            //查找class
            Class clazz = dex.loadClassBinaryName(name, definingContext, suppressed);
            if (clazz != null) {
                return clazz;
            }
        }
    }
    if (dexElementsSuppressedExceptions != null) {
        suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
    }
    return null;
}
```

热修复

`PathClassLoader`中存在一个Element数组，Element类中存在一个dexFile成员表示dex文件，即：APK中有X个dex，则Element数组就有X个元素。



在`PathClassLoader`中的Element数组为：[patch.dex , classes.dex , classes2.dex]。如果存在**Key.class**位于patch.dex与classes2.dex中都存在一份，当进行类查找时，循环获得`dexElements`中的DexFile，查找到了**Key.class**则立即返回，不会再管后续的element中的DexFile是否能加载到**Key.class**了。

因此实际上，一种热修复实现可以将出现Bug的class单独的制作一份fix.dex文件(补丁包)，然后在程序启动时，从服务器下载fix.dex保存到某个路径，再通过fix.dex的文件路径，用其创建`Element`对象，然后将这个`Element`对象插入到我们程序的类加载器`PathClassLoader`的`pathList`中的`dexElements`数组头部。这样在加载出现Bug的class时会优先加载fix.dex中的修复类，从而解决Bug。