Android热修复原理分析

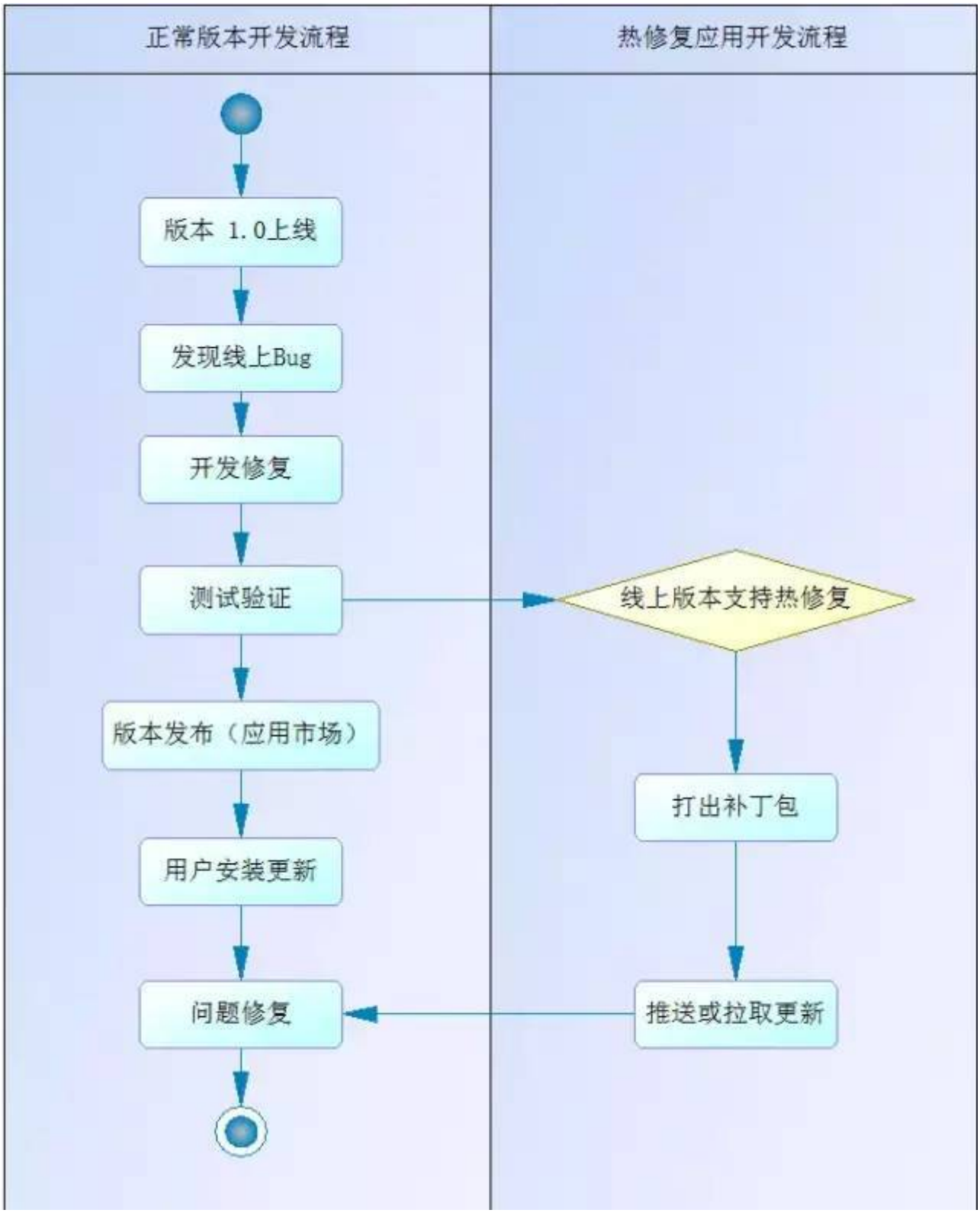
> 本文是体验课预习资料，目的是让大家了解什么是热修复，具体的实现细节，将在系列课中为大家直播演绎。大家了解了原理后，lance老师才能够顺利的带着手写代码了。请大家记住一句话：
>
> ***纸上得来终觉浅，绝知此事要躬行。***
>
> 代码需要大家自己写出来的那才是自己的代码，否则的话你看了，背了，最后工作中还是写不出来优秀的代码

什么是热修复

> 热修复：让应用能够在无需重新安装的情况实现更新，帮助应用快速建立动态修复能力。

早期遇到Bug我们一般会紧急发布了一个版本。然而这个Bug可能就是简简单单的一行代码，为了这一行代码，进行全量或者增量更新迭代一个版本，未免有点大材小用了。而且新版本的普及需要时间，以Android用户的升级习惯，即使是相对活跃的微信也需要10天以上的时间去覆盖50%的用户。使用热修复技术，能做到1天覆盖70%以上。这也是基于补丁体积较小，可以直接使用移动网络下载更新。

热修复开发流程



目前Android业内，热修复技术百花齐放，各大厂都推出了自己的热修复方案，使用的技术方案也各有所异。其中**QZone超级补丁**基于的是dex分包方案，而dex分包是基于Java的类加载机制`ClassLoader`。

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 文件加载到虚拟机 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，而它的主要实现类主要有：

- `BootClassLoader`

  用于加载Android Framework层class文件。
- `PathClassLoader`

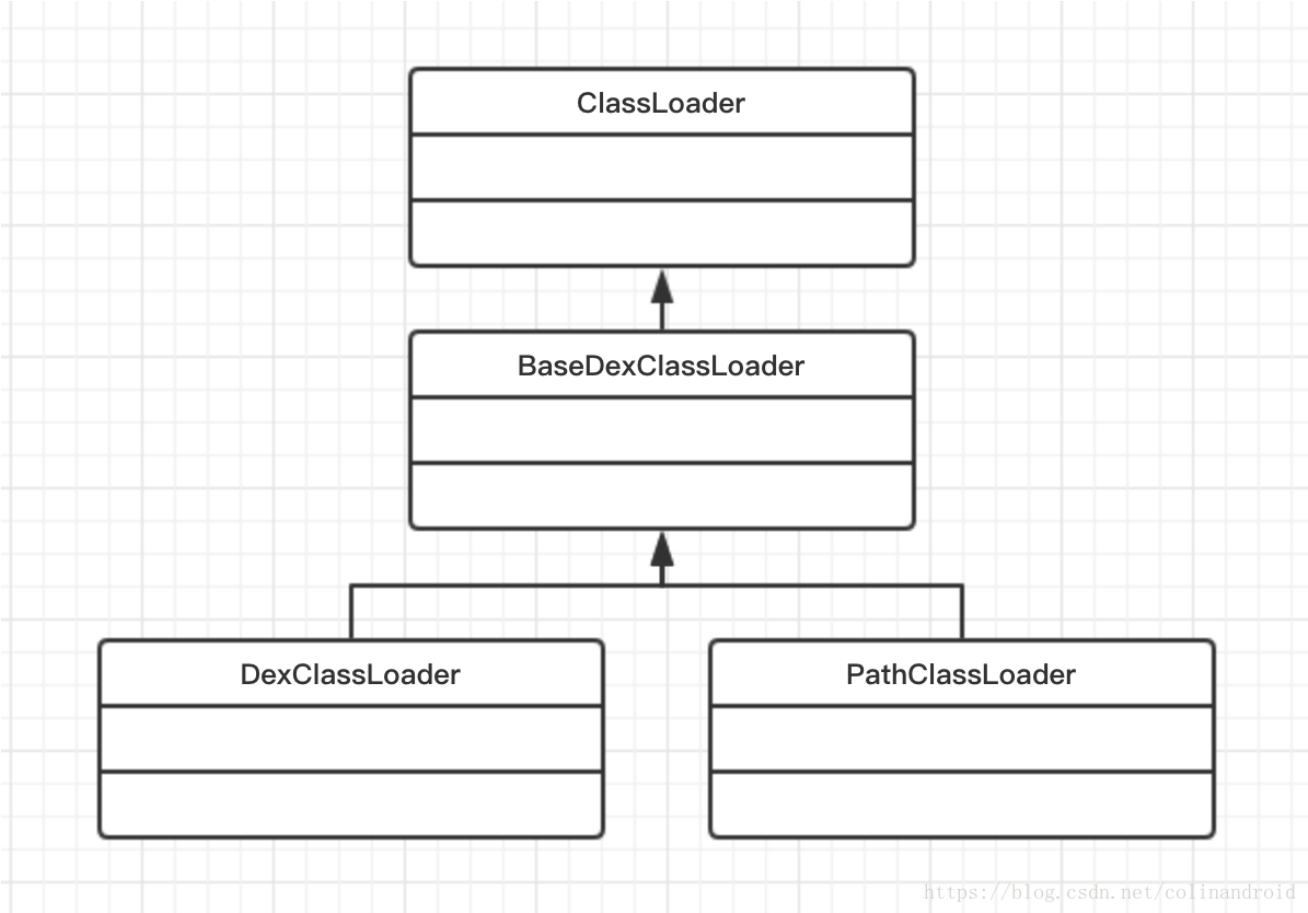
  用于Android应用程序类加载器。可以加载指定的dex，以及jar、zip、apk中的classes.dex
- `DexClassLoader`

  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，但是实际上`PathClassLoader`和`DexClassLoader`一样都能够加载sdcard中的dex。

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

> `optimizedDirectory`参数为odex的目录。实际上Android中的ClassLoader在加载dex时，会首先经过dexopt对dex执行优化，产生odex文件。`optimizedDirectory`为null时的默认路径为：***/data/dalvik-cache***。并且处于安全考虑，此目录需要使用app私有目录，如：`getCodeCacheDir()`
>
> 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

创建`ClassLoader`需要接收一个`ClassLoader parent`参数。这个`parent`为父类加载。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。这就是**双亲委托机制**！

```
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());` 并不仅仅只能获得 xx.dex中的Class，还能够获得其父ClassLoader中加载的Class。

findClass

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`

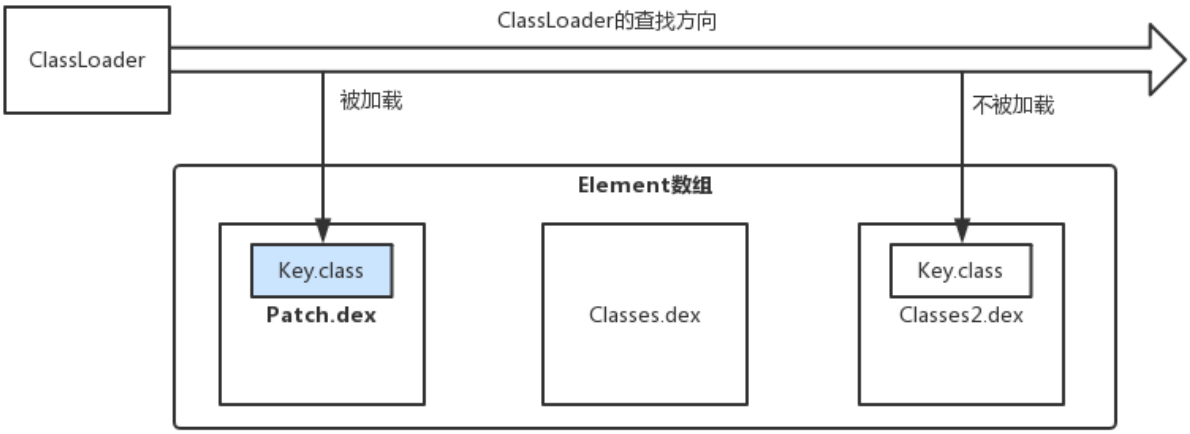
```
public DexPathList(ClassLoader definingContext, String dexPath,
            String librarySearchPath, File optimizedDirectory) {
```

```
    //.........
    // 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。

而对于类的查找，由代码`for (Element element : dexElements)`得知，会由数组从前往后进行查找。



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，可以将出现Bug的class单独的制作一份patch.dex文件(补丁包)，然后在程序启动时，从服务器下载patch.dex保存到某个路径，再通过patch.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patch.dex中的修复类，从而解决Bug。QQ空间热修复的原理就是这样，利用反射Hook了PathClassLoader中pathList的dexElements数组。

总结

在实现热修复的过程中，必须掌握的技术包括了反射、类加载机制并且掌握Framewrok层源码中关于`ClassLoader`部分的内容。同时如果需要继续自动化补丁生成还需要掌握gradle编程等内容。

文章中每一个部分都包含一系列BAT面试的面试点，这些点构建了一个完整的知识体系，热修复。后面，lance老师会细化里面的知识，如果 大家觉得有问题，请联系我哦，微信号：q2260035406，QQ同号。

享学课堂-Lance老师

扫一扫上面的二维码图案，加我微信