ASM字节码插桩

一、什么是插桩

QQ空间曾经发布的热修复解决方案中利用`Javaassist`库实现向类的构造函数中插入一段代码解决**CLASS_ISPREVERIFIED** 问题。包括了**Instant Run**的实现以及参照**Instant Run**实现的热修复美团Robus等等等等都利用到了插桩技术。

插桩就是将一段代码插入到另一段代码，或替换另一段代码。字节码插桩顾名思义就是在我们编写的源码编译成字节码（Class）后，在Android下生成dex之前修改Class文件，修改或者增强原有代码逻辑的操作。

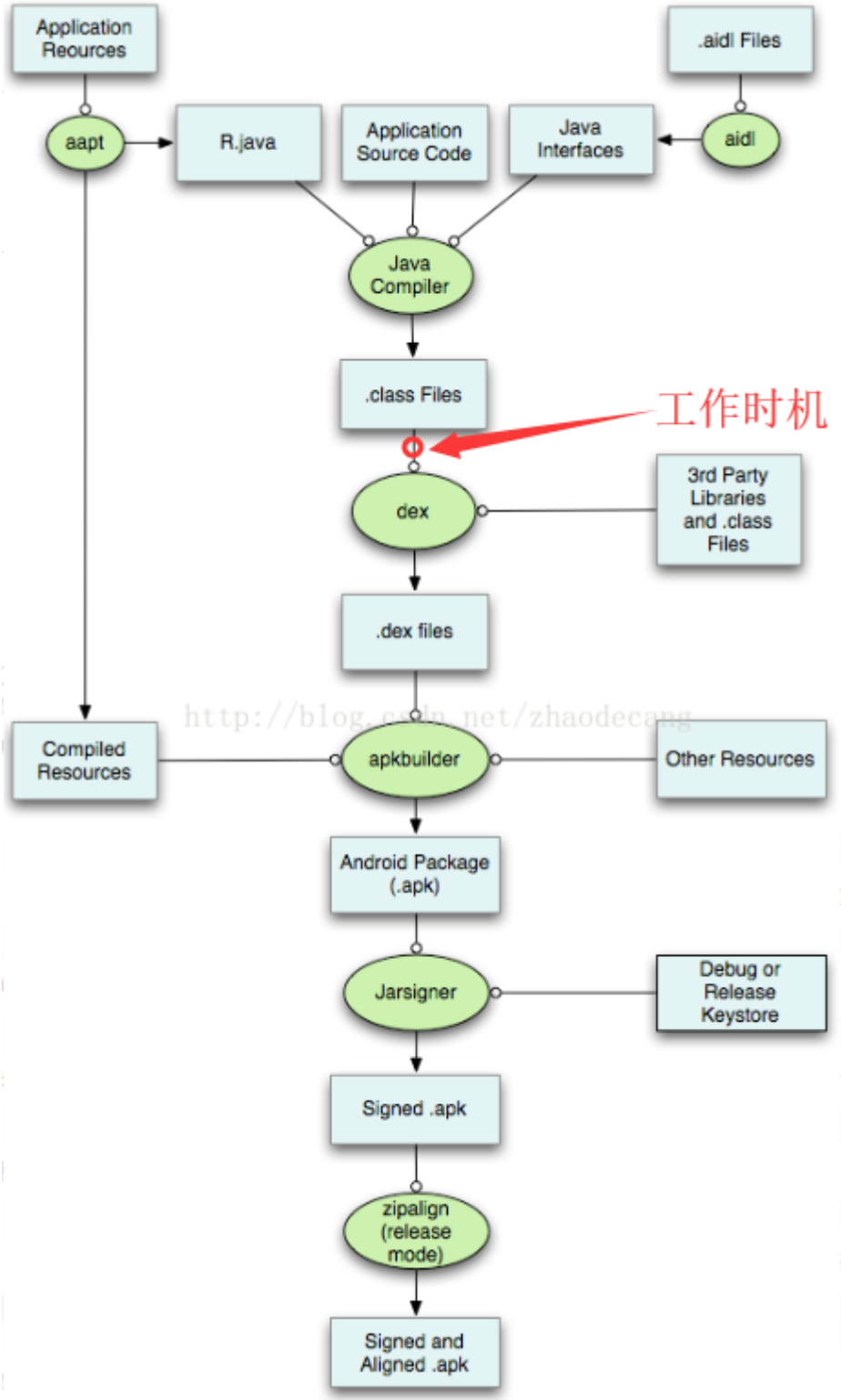
```
package com.enjoy.asminject;

import ...

public class MainActivity extends AppCompatActivity {
    public MainActivity() {
    }

    @InjectTime
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.setContentView(2131296284);
        this.a();
    }

    @InjectTime
    void a() {
        try {
            Thread.sleep( millis: 2000L);
        } catch (InterruptedException var2) {
            var2.printStackTrace();
        }

    }
}
```

```
public class MainActivity extends AppCompatActivity {
    public MainActivity() {
    }

    @InjectTime
    protected void onCreate(Bundle savedInstanceState) {
        long var2 = System.currentTimeMillis();
        super.onCreate(savedInstanceState);
        this.setContentView(2131296284);
        this.a();
        long var4 = System.currentTimeMillis();
        System.out.println("MainActivity execute onCreate: " + (var4 - var2) + "ms.");
    }

    @InjectTime
    void a() {
        long var1 = System.currentTimeMillis();

        try {
            Thread.sleep( millis: 2000L);
        } catch (InterruptedException var6) {
            var6.printStackTrace();
        }

        long var4 = System.currentTimeMillis();
        System.out.println("MainActivity execute a: " + (var4 - var1) + "ms.");
    }
}
```

二、字节码操作框架

上面我们提到QQ空间使用了`Javaassist`来进行字节码插桩，除了`Javaassist`之外还有一个应用更为广泛的`ASM`框架同样也是字节码操作框架，Instant Run包括`AspectJ`就是借助`ASM`来实现各自的功能。

我们非常熟悉的JSON格式数据是基于文本的，我们只需要知道它的规则就能够轻松的生成、修改JSON数据。同样的Class字节码也有其自己的规则(格式)。操作JSON可以借助GSON来非常方便的生成、修改JSON数据。而字节码Class，同样可以借助Javassist/ASM来实现对其修改。可能`Javassist`更加简单，而`ASM`性能更好。



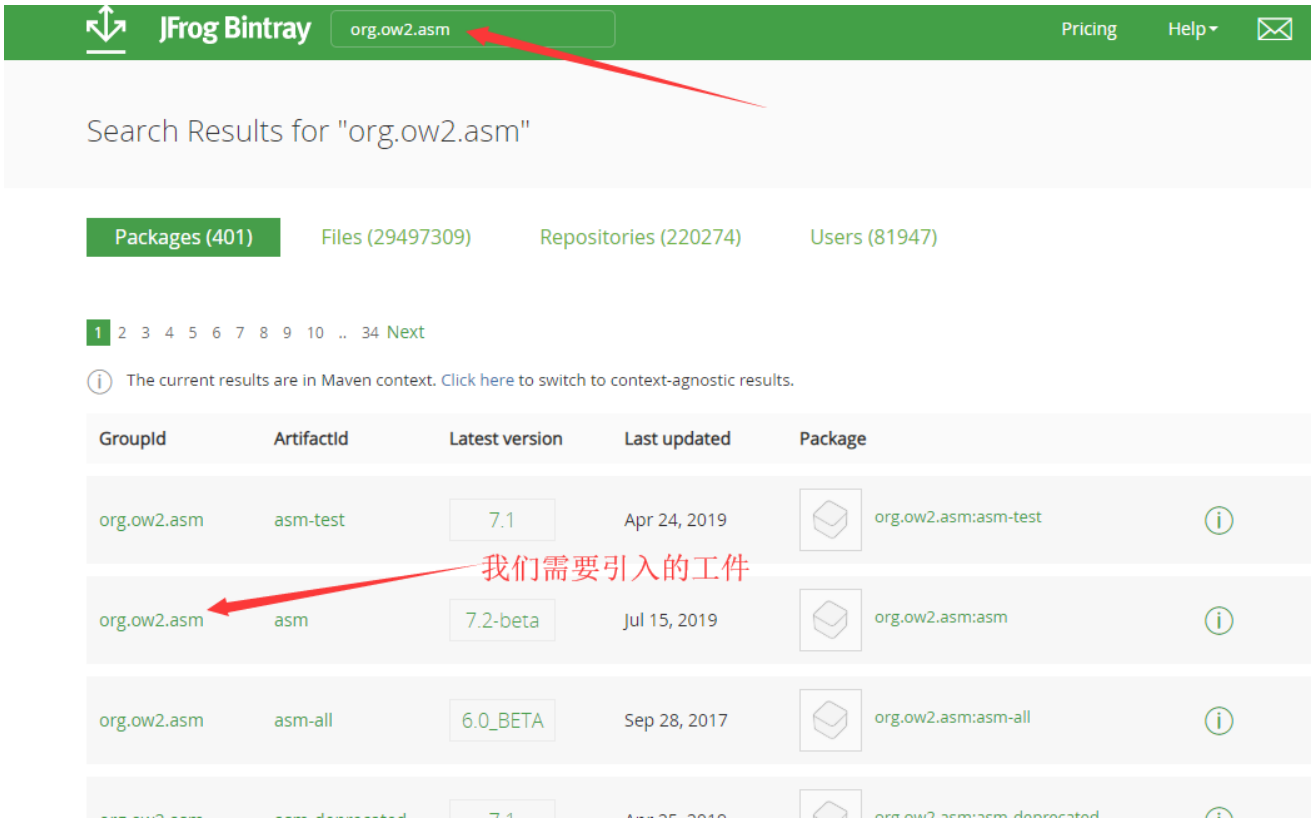
字节码操作框架的作用在于生成或者修改Class文件，因此在Android中字节码框架本身是不需要打包进入APK的，只有其生成/修改之后的Class才需要打包进入APK中。它的工作时机在上图Android打包流程中的生成Class之后，打包dex之前。

三、ASM的使用

由于`ASM`具有相对于`Javassist`更好的性能以及更高的灵活行，我们这篇文章以使用ASM为主。在真正利用到Android中之前，我们可以先在`Java`程序中完成对字节码的修改测试。

3.1、在AS中引入ASM

`ASM`可以直接从jcenter()`仓库中引入，所以我们可以进入:https://bintray.com/进行搜索



点击图中标注的工件进入，可以看到最新的正式版本为：7.1。

因此，我们可以在AS中加入：

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:


    testImplementation 'org.ow2.asm:asm:7.1'
    testImplementation 'org.ow2.asm:asm-commons:7.1'
}
```

同时，需要注意的是：我们使用`testImplementation`引入，这表示我们只能在Java的单元测试中使用这个框架，对我们Android中的依赖关系没有任何影响。

> AS中使用gradle的Android工程会自动创建Java单元测试与Android单元测试。测试代码分别在test与androidTest。

3.2、准备待插桩Class

在`test/java`下面创建一个Java类：

```
public class InjectTest {

    public static void main(String[] args) {

    }
}
```

由于我们操作的是字节码插桩，所以可以进入`test/java`下面使用`javac`对这个类进行编译生成对应的class文件。

```
javac InjectTest.java
```

3.3、执行插桩

因为`main`方法中没有任何输出代码，我们输入命令：`java InjectTest`执行这个Class不会有任何输出。那么我们接下来利用`ASM`，向`main`方法中插入一开始图中的记录函数执行时间的日志输出。

在单元测试中写入测试方法

```
        /**
         * 1、准备待分析的class
         */
        FileInputStream fis = new FileInputStream
                ("xxxxx/test/java/InjectTest.class");
```

```
        /**
         * 2、执行分析与插桩
         */
        //class字节码的读取与分析引擎
        ClassReader cr = new ClassReader(fis);
        // 写出器 COMPUTE_FRAMES 自动计算所有的内容，后续操作更简单
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        //分析，处理结果写入cw EXPAND_FRAMES：android必须用它(栈帧格式)
        cr.accept(new ClassAdapterVisitor(cw), ClassReader.EXPAND_FRAMES);


        /**
         * 3、获得结果并输出
         */
        byte[] newClassBytes = cw.toByteArray();
        File file = new File("xxx/test/java2/");
        file.mkdirs();

        FileOutputStream fos = new FileOutputStream
                ("xxx/test/java2/InjectTest.class");
        fos.write(newClassBytes);

        fos.close();
```

上面代码的会获取上一步生成的class，然后由ASM执行完插桩之后，将结果输出到`test/java2`目录下。其中关键点就在于第2步中，如何进行插桩。关于ASM框架本身的设计，我们这里先不讨论。