我们先来回顾一下，经过前面同学的介绍，我们了解了Gradle生命周期及hook点，Task定义、依赖与挂接到构建流程的基本操作。同时，还了解了自定义gradle插件的基本流程。有了前面的基础，我们今天来讲transform相关知识点。

我们先来思考几个：

> 1.为了对app性能做一个全面的评估，我们需要做UI，内存，网络等等方面的性能监控，如何做?
> 2.发现某个第三方依赖，用起来不爽，但是不想拿它的源码修改再重新编译，有什么好的办法吗?
> 3.每次写打log时，想让TAG自动生成，让它默认就是当前类的名称，甚至你想让log里自动加上当前代码所在的行数，更方便定位日志位置,如何实现?
> ……

为了实现上面这些想法，可能我们最开始的第一反应，都是能否通过代码生成技术、APT，抑或反射、抑或动态代理来实现，但是想来想去，貌似这些方案都不能很好满足上面的需求，而且，有些问题不能从Java文件入手，而应该从class文件寻找突破。而从class文件入手，我们就不得不来近距离接触一下字节码!

JVM平台上，修改、生成字节码无处不在，从ORM框架（如Hibernate, MyBatis）到Mock框架（如Mockio），再到Java Web中的常青树Spring框架，再到新兴的JVM语言Kotlin的编译器，还有大名鼎鼎的cglib项目，都有字节码的身影。

字节码相关技术的强大之处自然不用多说，而且在Android开发中，无论是使用Java开发和Kotlin开发，都是JVM平台的语言，所以如果我们在Android开发中，使用字节码技术做一下hack，还可以天然地兼容Java和Kotlin语言。

那问题就很明了了，我们就是要通过字节码技术去解决前面提出的想法，这就回答了我们用什么做的问题。接下来，我们就开始来探索如何去做，我们从以下技术点展开。

- Transform的应用、原理、优化
- ASM的应用，开发流，以及与Android工程的适配
- 具体应用案例

一、Transform

引入Transform

Transform是Android gradle plugin 1.5开始引入的概念。

我们先从如何引入Transform依赖说起，首先我们需要编写一个自定义插件，然后在插件中注册一个自定义Transform。这其中我们需要先通过gradle引入Transform的依赖，这里有一个坑，Transform的库最开始是独立的，后来从2.0.0版本开始，被归入了Android编译系统依赖的gradle-api中，让我们看看Transform在jcenter上的历个版本。

所以，很久很久以前我引入transform依赖是这样

```
compile 'com.android.tools.build:transform-api:1.5.0'
```

现在是这样

```
//从2.0.0版本开始就是在gradle-api中了
implementation 'com.android.tools.build:gradle-api:3.1.4'
```

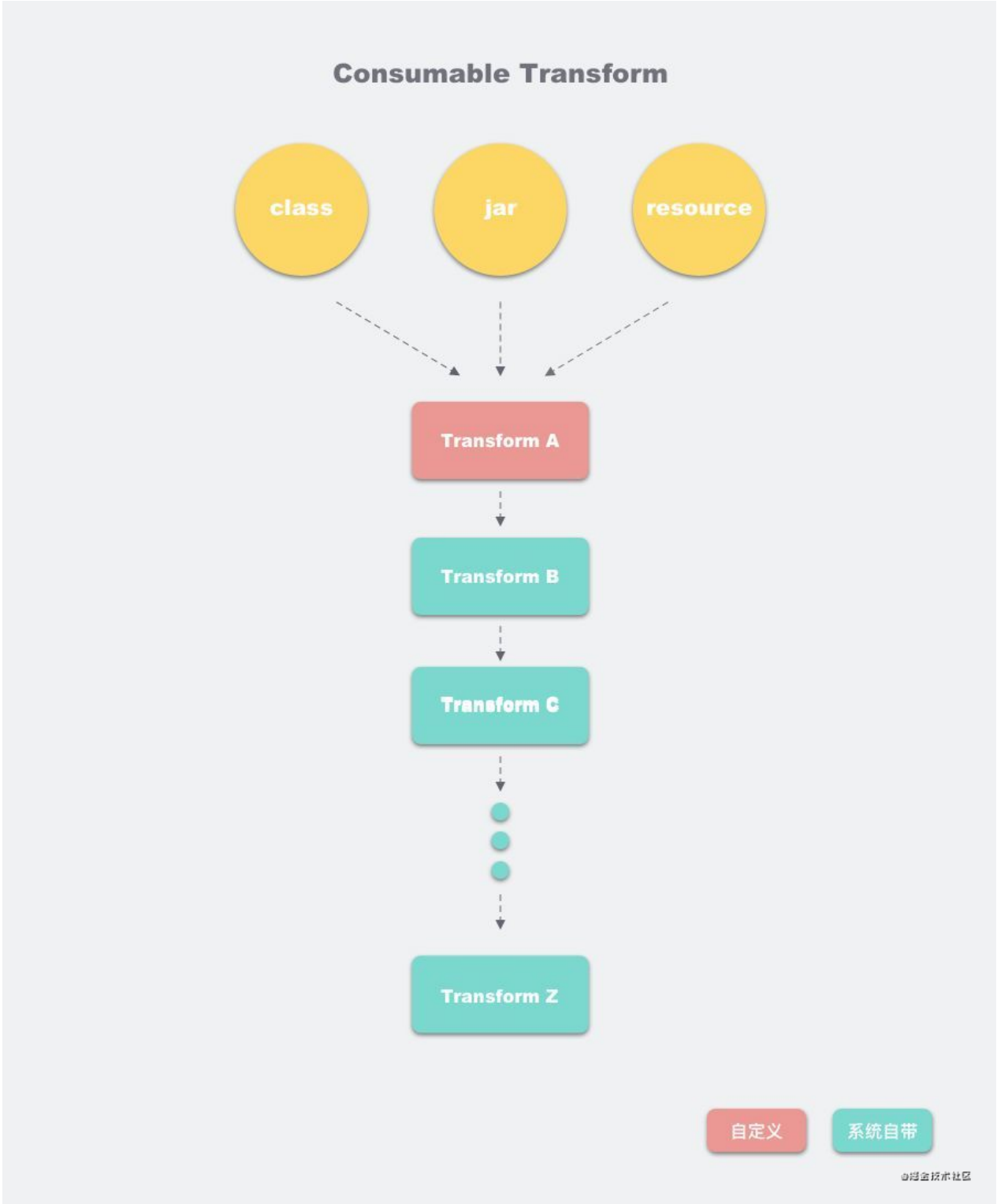
然后，让我们在自定义插件中注册一个自定义Transform，gradle插件可以使用java，groovy，kotlin编写，我这里选择使用java。

```
public class CustomPlugin implements Plugin<Project> {
    @SuppressWarnings("NullableProblems")
    @Override
    public void apply(Project project) {
        AppExtension appExtension =
(AppExtension)project.getProperties().get("android");
        appExtension.registerTransform(new CustomTransform(),
Collections.EMPTY_LIST);
    }
}
```

那么如何写一个自定义Transform呢?

Transform的原理与应用

介绍如何应用Transform之前，我们先介绍Transform的原理，一图胜千言



每个Transform其实都是一个gradle task，Android编译器中的TaskManager将每个Transform串连起来，第一个Transform接收来自javac编译的结果，以及已经拉取到在本地的第三方依赖（jar. aar），还有resource资源，注意，这里的resource并非android项目中的res资源，而是asset目录下的资源。这些编译的中间产物，在Transform组成的链条上流动，每个Transform节点可以对class进行处理再传递给下一个Transform。我们常见的混淆，Desugar等逻辑，它们的实现如今都是封装在一个个Transform中，而我们自定义的Transform，会插入到这个Transform链条的最前面。

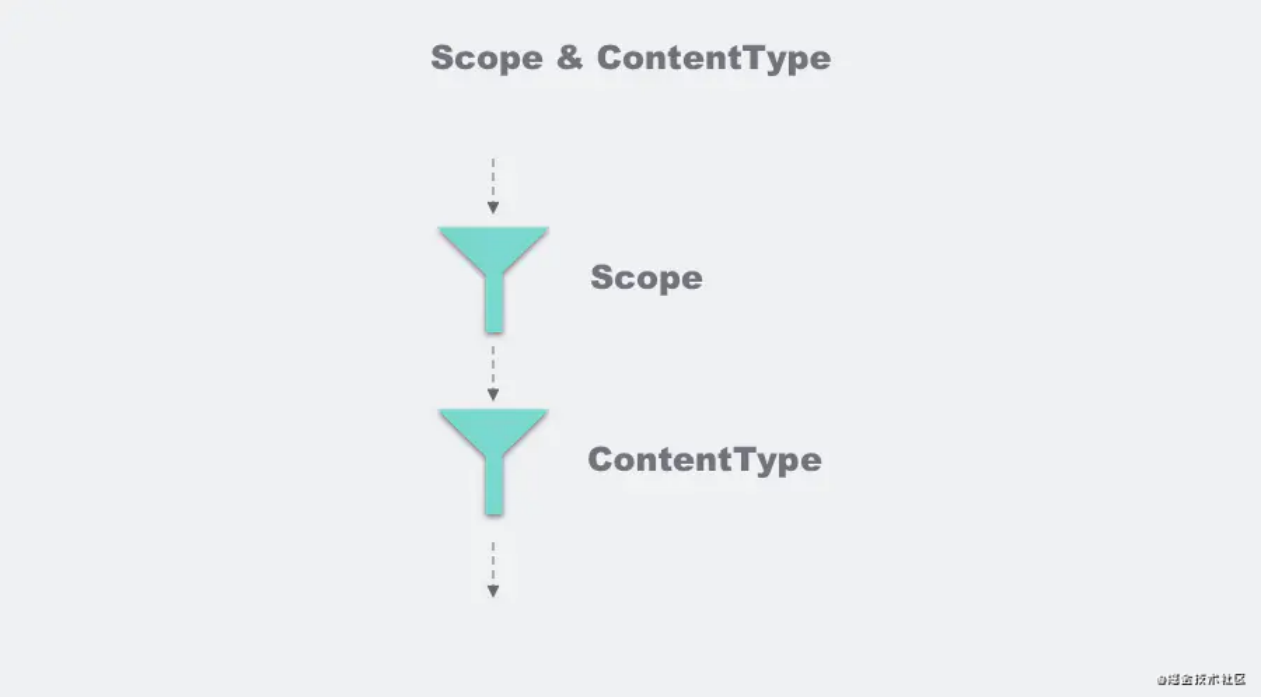
但其实，上面这幅图，只是展示Transform的其中一种情况。而Transform其实可以有两种输入，一种是消费型的，当前Transform需要将消费型型输出给下一个Transform，另一种是引用型的，当前Transform可以读取这些输入，而不需要输出给下一个Transform，比如Instant Run就是通过这种方式，检查两次编译之间的diff的。至于怎么在一个Transform中声明两种输入，以及怎么处理两种输入，后面将有示例代码。

为了印证Transform的工作原理和应用方式，我们也可以从Android gradle plugin源码入手找出证据，在TaskManager中，有一个方法`createPostCompilationTasks`.为了避免贴篇幅太长的源码，这里附上链接

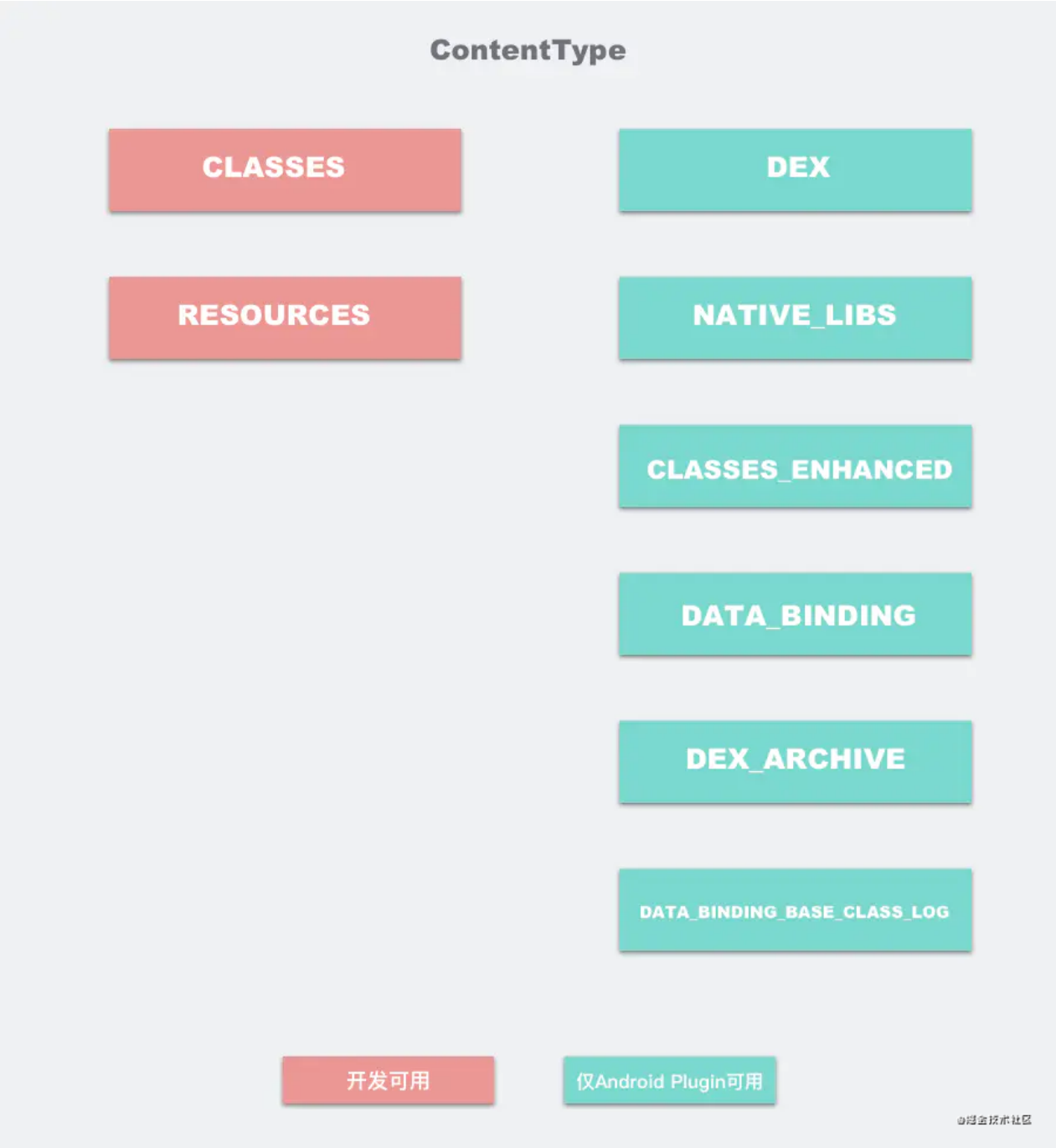
TaskManager#createPostCompilationTasks

这个方法的脉络很清晰，我们可以看到，Jacoco，Desugar，MergeJavaRes，AdvancedProfiling，Shrinker，Proguard, JarMergeTransform, MultiDex, Dex都是通过Transform的形式一个个串联起来。其中也有将我们自定义的Transform插进去。

讲完了Transform的数据流动的原理，我们再来介绍一下Transform的输入数据的过滤机制，Transform的数据输入，可以通过Scope和ContentType两个维度进行过滤。

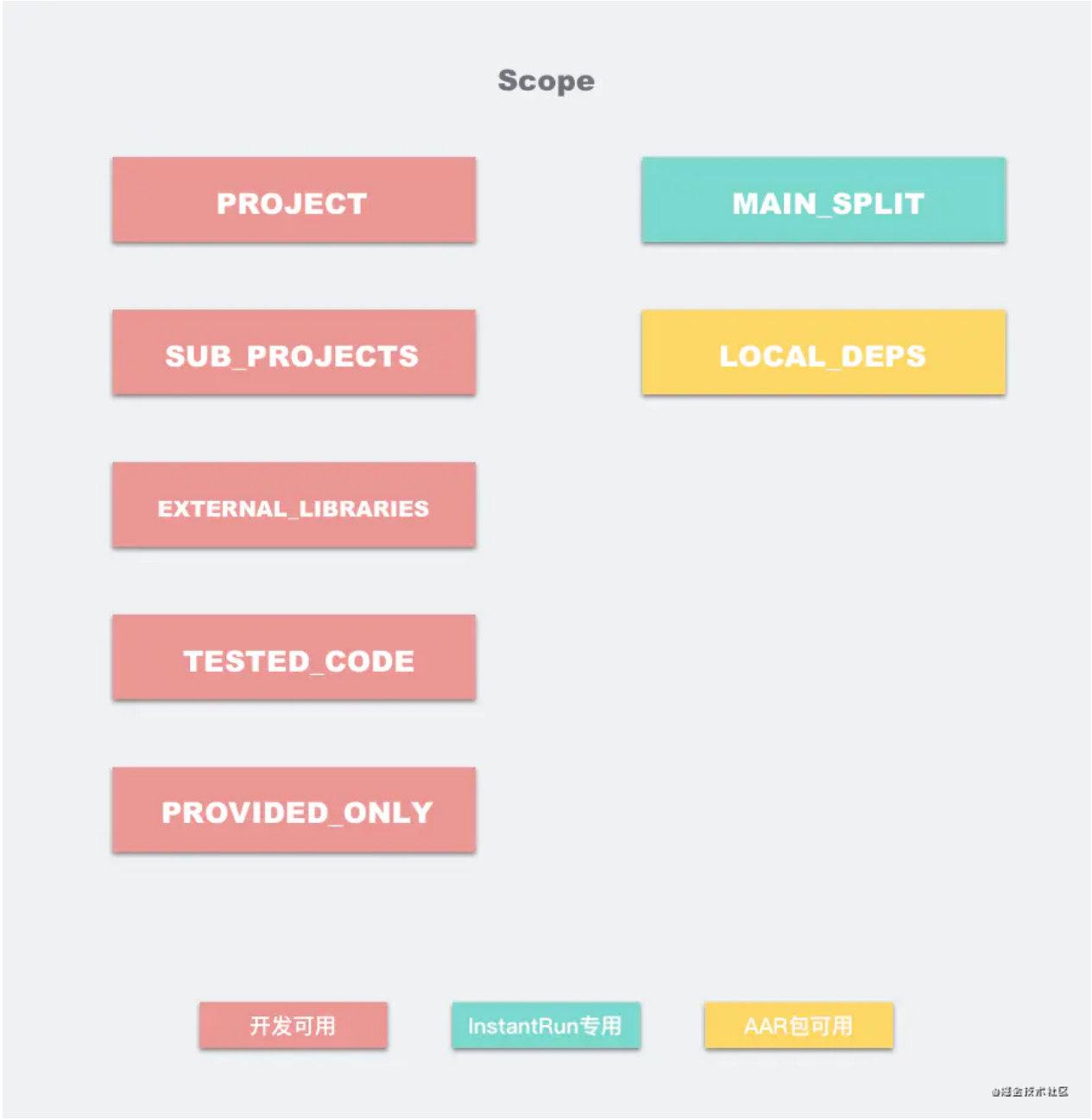


ContentType，顾名思义，就是数据类型，在插件开发中，我们一般只能使用CLASSES和RESOURCES两种类型，注意，其中的CLASSES已经包含了class文件和jar文件



从图中可以看到，除了CLASSES和RESOURCES，还有一些我们开发过程无法使用的类型，比如DEX文件，这些隐藏类型在一个独立的枚举类ExtendedContentType中，这些类型只能给Android编译器使用。另外，我们一般使用 TransformManager中提供的几个常用的ContentType集合和Scope集合，如果是要处理所有class和jar的字节码，ContentType我们一般使用`TransformManager.CONTENT_CLASS`。

Scope相比ContentType则是另一个维度的过滤规则，



我们可以发现，左边几个类型可供我们使用，而我们一般都是组合使用这几个类型，TransformManager有几个常用的Scope集合方便开发者使用。如果是要处理所有class字节码，Scope我们一般使用`TransformManager.SCOPE_FULL_PROJECT`。

好，目前为止，我们介绍了Transform的数据流动的原理，输入的类型和过滤机制，我们再写一个简单的自定义Transform，让我们对Transform可以有一个更具体的认识.

```
public class CustomTransform extends Transform {
    public static final String TAG = "CustomTransform";
    public CustomTransform() {
        super();
    }
    @Override
    public String getName() {
        return "CustomTransform";
    }
    @Override
```

```
    public void transform(TransformInvocation transformInvocation) throws
TransformException, InterruptedException, IOException {
        super.transform(transformInvocation);
        //当前是否是增量编译
        boolean isIncremental = transformInvocation.isIncremental();
        //消费型输入，可以从中获取jar包和class文件夹路径。需要输出给下一个任务
        Collection<TransformInput> inputs = transformInvocation.getInputs();
        //引用型输入，无需输出。
        Collection<TransformInput> referencedInputs =
transformInvocation.getReferencedInputs();
        //OutputProvider管理输出路径，如果消费型输入为空，你会发现OutputProvider ==
null
        TransformOutputProvider outputProvider =
transformInvocation.getOutputProvider();
        for(TransformInput input : inputs) {
            for(JarInput jarInput : input.getJarInputs()) {
                File dest = outputProvider.getContentLocation(
                        jarInput.getFile().getAbsolutePath(),
                        jarInput.getContentTypes(),
                        jarInput.getScopes(),
                        Format.JAR);
                //将修改过的字节码copy到dest，就可以实现编译期间干预字节码的目的了

                FileUtils.copyFile(jarInput.getFile(), dest);
            }
            for(DirectoryInput directoryInput : input.getDirectoryInputs()) {
                File dest =
outputProvider.getContentLocation(directoryInput.getName(),
                        directoryInput.getContentTypes(),
directoryInput.getScopes(),
                        Format.DIRECTORY);
                //将修改过的字节码copy到dest，就可以实现编译期间干预字节码的目的了

                FileUtils.copyDirectory(directoryInput.getFile(), dest);
            }
        }
    }
    @Override
    public Set<QualifiedContent.ContentType> getInputTypes() {
        return TransformManager.CONTENT_CLASS;
    }
    @Override
    public Set<? super QualifiedContent.Scope> getScopes() {
        return TransformManager.SCOPE_FULL_PROJECT;
    }
    @Override
    public Set<QualifiedContent.ContentType> getOutputTypes() {
        return super.getOutputTypes();
    }
```

```
    @Override
    public Set<? super QualifiedContent.Scope> getReferencedScopes() {
        return TransformManager.EMPTY_SCOPES;
    }
    @Override
    public Map<String, Object> getParameterInputs() {
        return super.getParameterInputs();
    }
    @Override
    public boolean isCacheable() {
        return true;
    }

    @Override
    public boolean isIncremental() {
        return true; //是否开启增量编译
    }
}
```

可以看到，在transform方法中，我们将每个jar包和class文件复制到dest路径，这个dest路径就是下一个Transform的输入数据，而在复制时，我们就可以做一些狸猫换太子，偷天换日的事情了，先将jar包和class文件的字节码做一些修改，再进行复制即可，至于怎么修改字节码，就要借助我们后面介绍的ASM了。而如果开发过程要看你当前transform处理之后的class/jar包，可以到\/build/intermediates/transforms/CustomTransform/下查看，你会发现所有jar包命名都是123456递增，这是正常的，这里的命名规则可以在OutputProvider.getContentLocation的具体实现中找到

```
public synchronized File getContentLocation(
        @NonNull String name,
        @NonNull Set<ContentType> types,
        @NonNull Set<? super Scope> scopes,
        @NonNull Format format) {
    // runtime check these since it's (indirectly) called by 3rd party
transforms.
    checkNotNull(name);
    checkNotNull(types);
    checkNotNull(scopes);
    checkNotNull(format);
    checkState(!name.isEmpty());
    checkState(!types.isEmpty());
    checkState(!scopes.isEmpty());
    // search for an existing matching substream.
    for (SubStream subStream : subStreams) {
        // look for an existing match. This means same name, types, scopes,
and format.
        if (name.equals(subStream.getName())
                && types.equals(subStream.getTypes())
                && scopes.equals(subStream.getScopes())
```

```
                && format == subStream.getFormat()) {
            return new File(rootFolder, subStream.getFilename());
        }
    }
    //按位置递增！！
    // didn't find a matching output. create the new output
    SubStream newSubStream = new SubStream(name, nextIndex++, scopes, types,
format, true);
    subStreams.add(newSubStream);
    return new File(rootFolder, newSubStream.getFilename());
}
```

Transform的优化：增量与并发

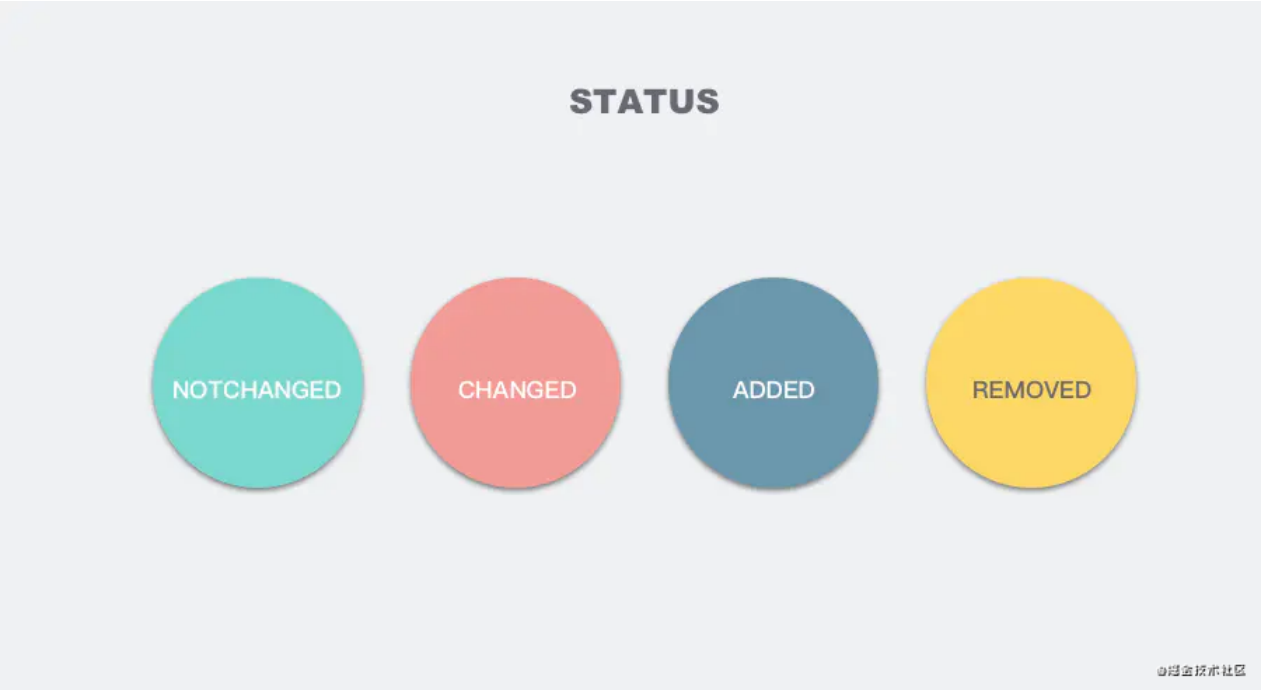
到此为止，看起来Transform用起来也不难，但是，如果直接这样使用，会大大拖慢编译时间，为了解决这个问题，摸索了一段时间后，也借鉴了Android编译器中Desugar等几个Transform的实现，发现我们可以使用增量编译，并且上面transform方法遍历处理每个jar/class的流程，其实可以并发处理，加上一般编译流程都是在PC上，所以我们可以尽量敲诈机器的资源。

想要开启增量编译，我们需要重写Transform的这个接口，返回true。

```
@Override
public boolean isIncremental() {
    return true;
}
```

虽然开启了增量编译，但也并非每次编译过程都是支持增量的，毕竟一次clean build完全没有增量的基础，所以，我们需要检查当前编译是否是增量编译。

如果不是增量编译，则清空output目录，然后按照前面的方式，逐个class/jar处理\
如果是增量编译，则要检查每个文件的Status，Status分四种，并且对这四种文件的操作也不尽相同



- NOTCHANGED: 当前文件不需处理，甚至复制操作都不用；
- ADDED、CHANGED: 正常处理，输出给下一个任务；
- REMOVED: 移除outputProvider获取路径对应的文件。

大概实现可以一起看看下面的代码

```
@Override
public void transform(TransformInvocation transformInvocation){
    Collection<TransformInput> inputs = transformInvocation.getInputs();
    TransformOutputProvider outputProvider =
transformInvocation.getOutputProvider();
    boolean isIncremental = transformInvocation.isIncremental();
    //如果非增量，则清空旧的输出内容
    if(!isIncremental) {
        outputProvider.deleteAll();
    }
    for(TransformInput input : inputs) {
        for(JarInput jarInput : input.getJarInputs()) {
            Status status = jarInput.getStatus();
            File dest = outputProvider.getContentLocation(
                    jarInput.getName(),
                    jarInput.getContentTypes(),
                    jarInput.getScopes(),
                    Format.JAR);
            if(isIncremental && !emptyRun) {
                switch(status) {
                    case NOTCHANGED:
                        continue;
                    case ADDED:
                    case CHANGED:
                        transformJar(jarInput.getFile(), dest, status);
```

```
                        break;
                    case REMOVED:
                        if (dest.exists()) {
                            FileUtils.forceDelete(dest);
                        }
                        break;
                }
            } else {
                transformJar(jarInput.getFile(), dest, status);
            }
        }
        for(DirectoryInput directoryInput : input.getDirectoryInputs()) {
            File dest =
outputProvider.getContentLocation(directoryInput.getName(),
                    directoryInput.getContentTypes(),
directoryInput.getScopes(),
                    Format.DIRECTORY);
            FileUtils.forceMkdir(dest);
            if(isIncremental && !emptyRun) {
                String srcDirPath =
directoryInput.getFile().getAbsolutePath();
                String destDirPath = dest.getAbsolutePath();
                Map<File, Status> fileStatusMap =
directoryInput.getChangedFiles();
                for (Map.Entry<File, Status> changedFile :
fileStatusMap.entrySet()) {
                    Status status = changedFile.getValue();
                    File inputFile = changedFile.getKey();
                    String destFilePath =
inputFile.getAbsolutePath().replace(srcDirPath, destDirPath);
                    File destFile = new File(destFilePath);
                    switch (status) {
                        case NOTCHANGED:
                            break;
                        case REMOVED:
                            if(destFile.exists()) {
                                FileUtils.forceDelete(destFile);
                            }
                            break;
                        case ADDED:
                        case CHANGED:
                            FileUtils.touch(destFile);
                            transformSingleFile(inputFile, destFile,
srcDirPath);
                            break;
                    }
                }
            } else {
                transformDir(directoryInput.getFile(), dest);
```

```
            }
        }
    }
}
```

这就能为我们的编译插件提供增量的特性。

实现了增量编译后，我们最好也支持并发编译，并发编译的实现并不复杂，只需要将上面处理单个jar/class的逻辑，并发处理，最后阻塞等待所有任务结束即可。

```
private WaitableExecutor waitableExecutor =
WaitableExecutor.useGlobalSharedThreadPool();
//异步并发处理jar/class
waitableExecutor.execute(() -> {
    bytecodeWeaver.weaveJar(srcJar, destJar);
    return null;
});
waitableExecutor.execute(() -> {
    bytecodeWeaver.weaveSingleClassToFile(file, outputFile, inputDirPath);
    return null;
});
//等待所有任务结束
waitableExecutor.waitForTasksWithQuickFail(true);
```

上面我们介绍了Transform，以及如何高效地在编译期间处理所有字节码，那么具体怎么处理字节码呢？接下来让我们一起看看JVM平台上的处理字节码神兵利器，ASM!

二、ASM

ASM的官网在这里asm.ow2.io/，贴一下它的主页介绍，一起感受下它的强大

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form. ASM provides some common bytecode transformations and analysis algorithms from which custom complex transformations and code analysis tools can be built. ASM offers similar functionality as other Java bytecode frameworks, but is focused on performance. Because it was designed and implemented to be as small and as fast as possible, it is well suited for use in dynamic systems (but can of course be used in a static way too, e.g. in compilers).

ASM is used in many projects, including:

- the **OpenJDK**, to generate the lambda call sites, and also in the Nashorn compiler,
- the **Groovy** compiler and the **Kotlin** compiler,
- **Cobertura** and **Jacoco**, to instrument classes in order to measure code coverage,
- **CGLIB**, to dynamically generate proxy classes (which are used in other projects such as **Mockito** and **EasyMock**),
- **Gradle**, to generate some classes at runtime.

@掘金技术社区

JVM平台上，处理字节码的框架最常见的就三个，ASM，Javasist，AspectJ。关于以上几个框架的优劣可以参考前面同事的文档。毫无疑问，ASM是最优的，因为使用它可以更底层地处理字节码的每条命令，处理速度、内存占用，也优于其他两个框架。

ASM的引入

下面是一份完整的gradle自定义plugin + transform + asm所需依赖，注意一下，此处两个gradleApi的区别

```
dependencies {
    //使用项目中指定的gradle wrapper版本，插件中使用的Project对象等等就来自这里
    implementation gradleApi()
    //使用本地的groovy
    implementation localGroovy()
    //Android编译的大部分gradle源码，比如上面讲到的TaskManager
    implementation 'com.android.tools.build:gradle:4.1.0'
    //这个依赖里其实主要存了transform的依赖，注意，这个依赖不同于上面的gradleApi()
    implementation 'com.android.tools.build:gradle-api:4.1.0'
    //ASM相关
    implementation 'org.ow2.asm:asm:7.1'
    implementation 'org.ow2.asm:asm-util:7.1'
    implementation 'org.ow2.asm:asm-commons:7.1'
}
```

ASM的应用

ASM设计了两种API类型，一种是Tree API, 一种是基于Visitor API(visitor pattern),

Tree API将class的结构读取到内存，构建一个树形结构，然后需要处理Method、Field等元素时，到树形结构中定位到某个元素，进行操作，然后把操作再写入新的class文件。

Visitor API则将通过接口的方式，分离读class和写class的逻辑，一般通过一个ClassReader负责读取class字节码，然后ClassReader通过一个ClassVisitor接口，将字节码的每个细节按顺序通过接口的方式，传递给ClassVisitor（你会发现ClassVisitor中有多个visitXXXX接口），这个过程就像ClassReader带着ClassVisitor游览了class字节码的每一个指令。

上面这两种解析文件结构的方式在很多处理结构化数据时都常见，一般得看需求背景选择合适的方案，而我们的需求是这样的，出于某个目的，寻找class文件中的一个hook点，进行字节码修改，这种背景下，我们选择Visitor API的方式比较合适。

让我们来写一个简单的demo，这段代码很简单，通过Visitor API读取一个class的内容，保存到另一个文件

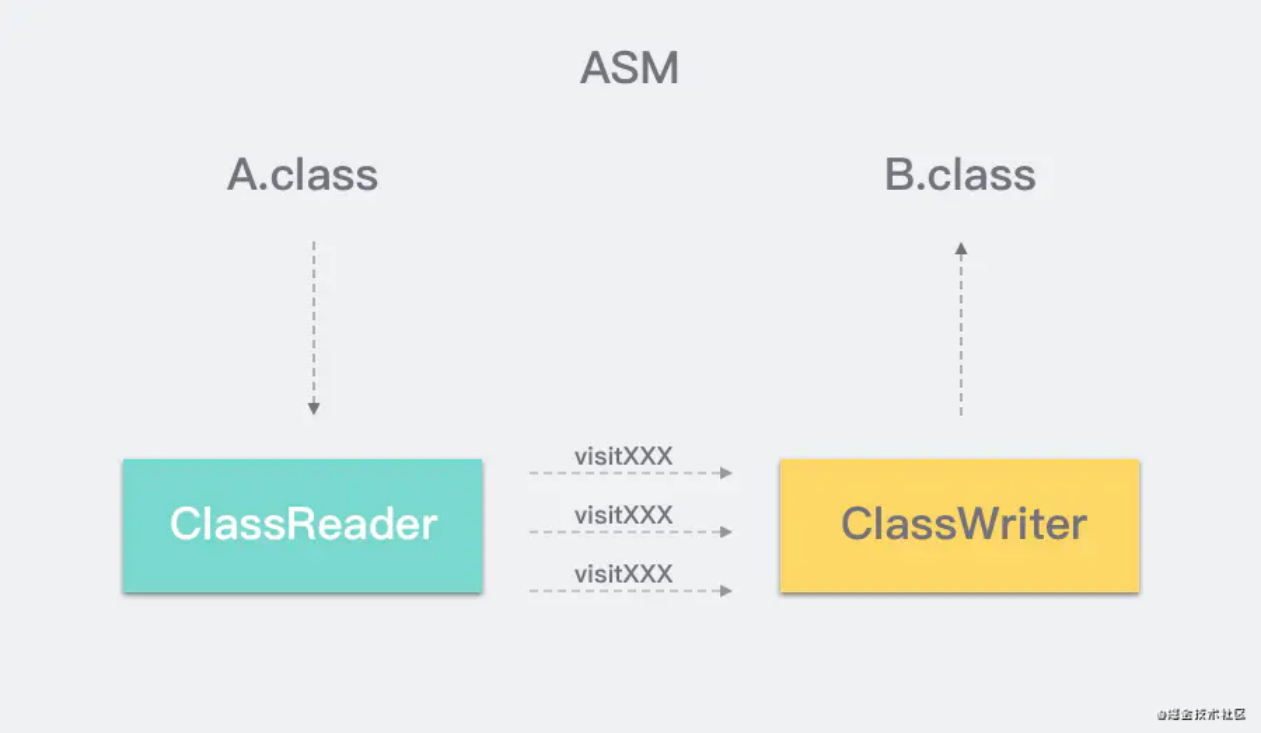
```
private void copy(String inputPath, String outputPath) {
    try {
        FileInputStream is = new FileInputStream(inputPath);
        ClassReader cr = new ClassReader(is);
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        cr.accept(cw, 0);
        FileOutputStream fos = new FileOutputStream(outputPath);
        fos.write(cw.toByteArray());
        fos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

首先，我们通过ClassReader读取某个class文件，然后定义一个ClassWriter，这个ClassWriter我们可以看它源码，其实就是一个ClassVisitor的实现，负责将ClassReader传递过来的数据写到一个字节流中，而真正触发这个逻辑就是通过ClassWriter的accept方式。

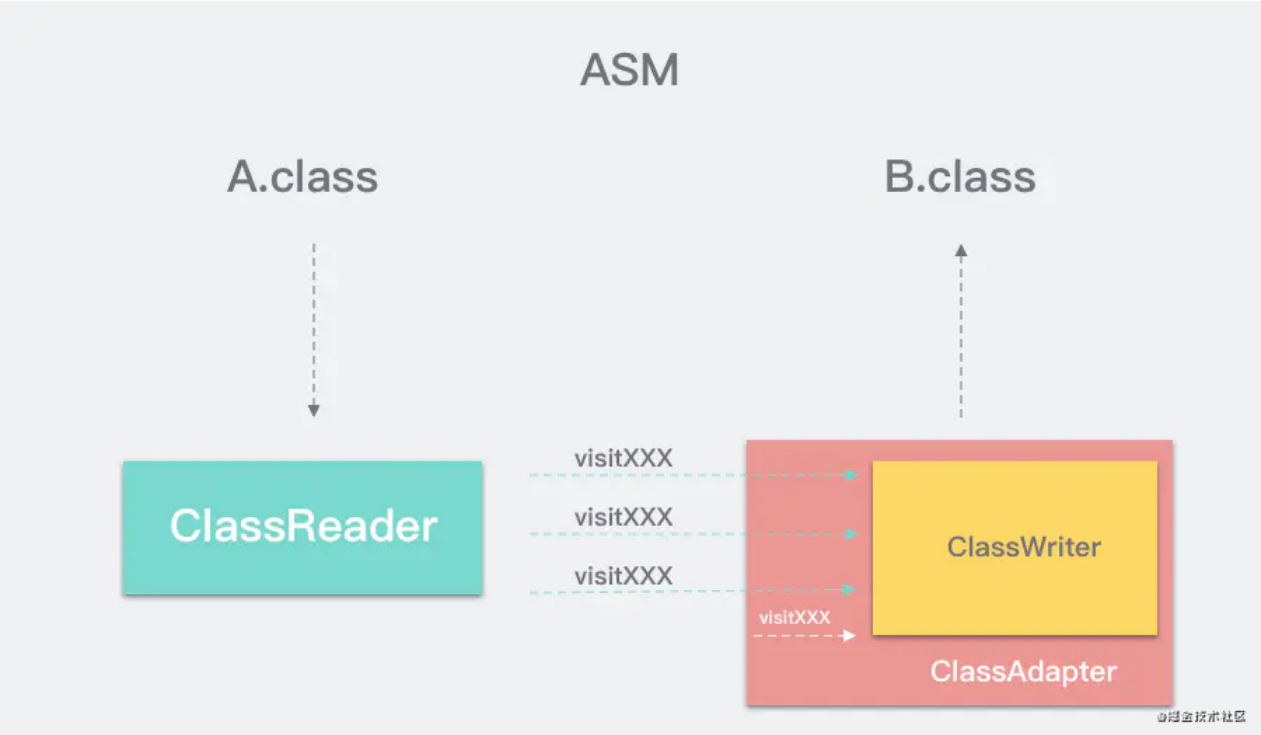
```
public void accept(ClassVisitor classVisitor, Attribute[] attributePrototypes,
int parsingOptions) {

    // 读取当前class的字节码信息
    int accessFlags = this.readUnsignedShort(currentOffset);
    String thisClass = this.readClass(currentOffset + 2, charBuffer);
    String superClass = this.readClass(currentOffset + 4, charBuffer);
    String[] interfaces = new String[this.readUnsignedShort(currentOffset +
6)];


    //classVisitor就是刚才accept方法传进来的ClassWriter，每次visitXXX都负责将字节码的
信息存储起来
    classVisitor.visit(this.readInt(this.cpInfoOffsets[1] - 7), accessFlags,
thisClass, signature, superClass, interfaces);

    /**
```

```
        略去很多visit逻辑
    */
    //visit Attribute
    while(attributes != null) {
        Attribute nextAttribute = attributes.nextAttribute;
        attributes.nextAttribute = null;
        classVisitor.visitAttribute(attributes);
        attributes = nextAttribute;
    }
    /**
        略去很多visit逻辑
    */
    classVisitor.visitEnd();
}
```

最后，我们通过ClassWriter的toByteArray()，将从ClassReader传递到ClassWriter的字节码导出，写入新的文件即可。这就完成了class文件的复制，这个demo虽然很简单，但是涵盖了ASM使用Visitor API修改字节码最底层的原理，大致流程如图



我们来分析一下，不难发现，如果我们要修改字节码，就是要从ClassWriter入手，上面我们提到ClassWriter中每个visitXXX（这些接口实现自ClassVisitor）都会保存字节码信息并最终可以导出，那么如果我们可以代理ClassWriter的接口，就可以干预最终字节码的生成了。



我们只要稍微看一下ClassVisitor的代码，发现它的构造函数，是可以接收另一个ClassVisitor的，从而通过这个ClassVisitor代理所有的方法.让我们来看一个例子，为class中的每个方法调用语句的开头和结尾插入一行代码

修改前的方法是这样

```
private static void printTwo() {
    printOne();
    printOne();
}
```

被修改后的方法是这样

```
private static void printTwo() {
    System.out.println("CALL printOne");
    printOne();
    System.out.println("RETURN printOne");
    System.out.println("CALL printOne");
    printOne();
    System.out.println("RETURN printOne");
}
```

让我们来看一下如何用ASM实现

```
private static void weave(String inputPath, String outputPath) {
    try {
        FileInputStream is = new FileInputStream(inputPath);
        ClassReader cr = new ClassReader(is);
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        CallClassAdapter adapter = new CallClassAdapter(cw);
```

```
        cr.accept(adapter, 0);
        FileOutputStream fos = new FileOutputStream(outputPath);
        fos.write(cw.toByteArray());
        fos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

这段代码和上面的实现复制class的代码唯一区别就是，使用了CallClassAdapter，它是一个自定义的ClassVisitor，我们将ClassWriter传递给CallClassAdapter的构造函数。来看看它的实现

```
//CallClassAdapter.java
public class CallClassAdapter extends ClassVisitor implements Opcodes {
    public CallClassAdapter(final ClassVisitor cv) {
        super(ASM5, cv);
    }
    @Override
    public void visit(int version, int access, String name, String signature,
String superName, String[] interfaces) {
        super.visit(version, access, name, signature, superName, interfaces);
    }
    @Override
    public MethodVisitor visitMethod(final int access, final String name,
                                     final String desc, final String
signature, final String[] exceptions) {
        MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
exceptions);
        return mv == null ? null : new CallMethodAdapter(name, mv);
    }
}
//CallMethodAdapter.java
class CallMethodAdapter extends MethodVisitor implements Opcodes {
    public CallMethodAdapter(final MethodVisitor mv) {
        super(ASM5, mv);
    }
    @Override
    public void visitMethodInsn(int opcode, String owner, String name, String
desc, boolean itf) {
        mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
        mv.visitLdcInsn("CALL " + name);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "
(Ljava/lang/String;)V", false);
        mv.visitMethodInsn(opcode, owner, name, desc, itf);
        mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
        mv.visitLdcInsn("RETURN " + name);
```

```
        mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream",
"println", "(Ljava/lang/String;)V", false);
    }
}
```

CallClassAdapter中的visitMethod使用了一个自定义的MethodVisitor——CallMethodAdapter，它也是代理了原来的MethodVisitor，原理和ClassVisitor的代理一样。

看到这里，貌似使用ASM修改字节码的大概套路都走完了，那么如何写出上面`visitMethodInsn`方法中插入打印方法名的逻辑，这就需要一些字节码的基础知识了，我们说过这里不会展开介绍字节码，但是我们可以介绍一些快速学习字节码的方式，同时也是开发字节码相关工程一些实用的工具。

在这之前，我们先讲讲行号的问题

如何验证行号

上面我们给每一句方法调用的前后都插入了一行日志打印，那么有没有想过，这样岂不是打乱了代码的行数，这样，万一crash了，定位堆栈岂不是乱套了。其实并不然，在上面visitMethodInsn中做的东西，其实都是在同一行中插入的代码，上面我们贴出来的代码是这样

```
private static void printTwo() {
    System.out.println("CALL printOne");
    printOne();
    System.out.println("RETURN printOne");
    System.out.println("CALL printOne");
    printOne();
    System.out.println("RETURN printOne");
}
```

无论你用idea还是eclipse打开上面的class文件，都是一行行展示的，但是其实class内部真实的行数应该是这样

```

private static void printTwo() {
    System.out.println("CALL printOne"); printOne();
System.out.println("RETURN printOne");
    System.out.println("CALL printOne"); printOne();
System.out.println("RETURN printOne");
}
```

ASM code

解析来介绍，如何写出上面生成代码的逻辑。首先，我们设想一下，如果要对某个class进行修改，那需要对字节码具体做什么修改呢？最直观的方法就是，先编译生成目标class，然后看它的字节码和原来class的字节码有什么区别（查看字节码可以使用javap工具），但是这样还不够，其实我们最终并不是读写字节码，而是使用ASM来修改，我们这里先做一个区别，bytecode vs ASM code，前者就是JVM意义的字节码，而后者是用ASM描述的bytecode，其实二者非常的接近，只是ASM code用Java代码来描

述。所以，我们应该是对比ASM code，而不是对比bytecode。对比ASM code的diff，基本就是我们要做的修改。

而ASM也提供了一个这样的类：ASMifier，它可以生成ASM code，但是，其实还有更快捷的工具，Intellij IDEA有一个插件\
Asm Bytecode Outline，可以查看一个class文件的bytecode和ASM code。

到此为止，貌似使用对比ASM code的方式，来实现字节码修改也不难，但是，这种方式只是可以实现一些修改字节码的基础场景，还有很多场景是需要对字节码有一些基础知识才能做到，而且，要阅读懂ASM code，也是需要一定字节码的的知识。所以，如果要开发字节码工程，还是需要学习一番字节码。

ClassWriter在Android上的坑

如果我们直接按上面的套路，将ASM应用到Android编译插件中，会踩到一个坑，这个坑来自于ClassWriter，具体是因为ClassWriter其中的一个逻辑，寻找两个类的共同父类。可以看看ClassWriter中的这个方法getCommonSuperClass，

```
/**
 * Returns the common super type of the two given types. The default
 * implementation of this method <i>loads</i> the two given classes and uses
 * the java.lang.Class methods to find the common super class. It can be
 * overridden to compute this common super type in other ways, in particular
 * without actually loading any class, or to take into account the class
 * that is currently being generated by this ClassWriter, which can of
 * course not be loaded since it is under construction.
 *
 * @param type1
 *            the internal name of a class.
 * @param type2
 *            the internal name of another class.
 * @return the internal name of the common super class of the two given
 *         classes.
 */
protected String getCommonSuperClass(final String type1, final String type2) {
    Class<?> c, d;
    ClassLoader classLoader = getClass().getClassLoader();
    try {
        c = Class.forName(type1.replace('/', '.'), false, classLoader);
        d = Class.forName(type2.replace('/', '.'), false, classLoader);
    } catch (Exception e) {
        throw new RuntimeException(e.toString());
    }
    if (c.isAssignableFrom(d)) {
        return type1;
    }
    if (d.isAssignableFrom(c)) {
        return type2;
    }
```

```
    if (c.isInterface() || d.isInterface()) {
        return "java/lang/Object";
    } else {
        do {
            c = c.getSuperclass();
        } while (!c.isAssignableFrom(d));
        return c.getName().replace('.', '/');
    }
}
```

这个方法用于寻找两个类的共同父类，我们可以看到它是获取当前class的classLoader加载两个输入的类型，而编译期间使用的classloader并没有加载Android项目中的代码，所以我们需要一个自定义的ClassLoader，将前面提到的Transform中接收到的所有jar以及class，还有android.jar都添加到自定义ClassLoader中。（其实上面这个方法注释中已经暗示了这个方法存在的一些问题）

```
public static URLClassLoader getClassLoader(Collection<TransformInput> inputs,
                                            Collection<TransformInput>
referencedInputs,
                                            Project project) throws
MalformedURLException {
    ImmutableList.Builder<URL> urls = new ImmutableList.Builder<>();
    String androidJarPath  = getAndroidJarPath(project);
    File file = new File(androidJarPath);
    URL androidJarURL = file.toURI().toURL();
    urls.add(androidJarURL);
    for (TransformInput totalInputs : Iterables.concat(inputs,
referencedInputs)) {
        for (DirectoryInput directoryInput : totalInputs.getDirectoryInputs())
{
            if (directoryInput.getFile().isDirectory()) {
                urls.add(directoryInput.getFile().toURI().toURL());
            }
        }
        for (JarInput jarInput : totalInputs.getJarInputs()) {
            if (jarInput.getFile().isFile()) {
                urls.add(jarInput.getFile().toURI().toURL());
            }
        }
    }
    ImmutableList<URL> allUrls = urls.build();
    URL[] classLoaderUrls = allUrls.toArray(new URL[allUrls.size()]);
    return new URLClassLoader(classLoaderUrls);
}
```

但是，如果只是替换了getCommonSuperClass中的Classloader，依然还有一个更深的坑，我们可以看看前面getCommonSuperClass的实现，它是如何寻找父类的呢？它是通过Class.forName加载某个类，然后再去寻找父类，但是，但是，android.jar中的类可不能随随便便加载的呀，android.jar对于Android工程来说只是编译时依赖，运行时是用Android机器上自己的android.jar。而且android.jar所有方法包括构造函数都是空实现，其中都只有一行代码

```
throw new RuntimeException("Stub!");
```

这样加载某个类时，它的静态域就会被触发，而如果有一个static的变量刚好在声明时被初始化，而初始化中只有一个RuntimeException，此时就会抛异常。

所以，我们不能通过这种方式来获取父类，能否通过不需要加载class就能获取它的父类的方式呢？谜底就在眼前，父类其实也是一个class的字节码中的一项数据，那么我们就从字节码中查询父类即可。最终实现是这样。

```
public class ExtendClassWriter extends ClassWriter {
    public static final String TAG = "ExtendClassWriter";
    private static final String OBJECT = "java/lang/Object";
    private ClassLoader urlClassLoader;
    public ExtendClassWriter(ClassLoader urlClassLoader, int flags) {
        super(flags);
        this.urlClassLoader = urlClassLoader;
    }
    @Override
    protected String getCommonSuperClass(final String type1, final String
type2) {
        if (type1 == null || type1.equals(OBJECT) || type2 == null ||
type2.equals(OBJECT)) {
            return OBJECT;
        }
        if (type1.equals(type2)) {
            return type1;
        }
        ClassReader type1ClassReader = getClassReader(type1);
        ClassReader type2ClassReader = getClassReader(type2);
        if (type1ClassReader == null || type2ClassReader == null) {
            return OBJECT;
        }
        if (isInterface(type1ClassReader)) {
            String interfaceName = type1;
            if (isImplements(interfaceName, type2ClassReader)) {
                return interfaceName;
            }
            if (isInterface(type2ClassReader)) {
                interfaceName = type2;
                if (isImplements(interfaceName, type1ClassReader)) {
                    return interfaceName;
```

```
                }
            }
            return OBJECT;
        }
        if (isInterface(type2ClassReader)) {
            String interfaceName = type2;
            if (isImplements(interfaceName, type1ClassReader)) {
                return interfaceName;
            }
            return OBJECT;
        }
        final Set<String> superClassNames = new HashSet<String>();
        superClassNames.add(type1);
        superClassNames.add(type2);
        String type1SuperClassName = type1ClassReader.getSuperName();
        if (!superClassNames.add(type1SuperClassName)) {
            return type1SuperClassName;
        }
        String type2SuperClassName = type2ClassReader.getSuperName();
        if (!superClassNames.add(type2SuperClassName)) {
            return type2SuperClassName;
        }
        while (type1SuperClassName != null || type2SuperClassName != null) {
            if (type1SuperClassName != null) {
                type1SuperClassName = getSuperClassName(type1SuperClassName);
                if (type1SuperClassName != null) {
                    if (!superClassNames.add(type1SuperClassName)) {
                        return type1SuperClassName;
                    }
                }
            }
            if (type2SuperClassName != null) {
                type2SuperClassName = getSuperClassName(type2SuperClassName);
                if (type2SuperClassName != null) {
                    if (!superClassNames.add(type2SuperClassName)) {
                        return type2SuperClassName;
                    }
                }
            }
        }
        return OBJECT;
    }
    private boolean isImplements(final String interfaceName, final ClassReader
classReader) {
        ClassReader classInfo = classReader;
        while (classInfo != null) {
            final String[] interfaceNames = classInfo.getInterfaces();
            for (String name : interfaceNames) {
                if (name != null && name.equals(interfaceName)) {
```

```
                    return true;
                }
            }
            for (String name : interfaceNames) {
                if(name != null) {
                    final ClassReader interfaceInfo = getClassReader(name);
                    if (interfaceInfo != null) {
                        if (isImplements(interfaceName, interfaceInfo)) {
                            return true;
                        }
                    }
                }
            }
            final String superClassName = classInfo.getSuperName();
            if (superClassName == null || superClassName.equals(OBJECT)) {
                break;
            }
            classInfo = getClassReader(superClassName);
        }
        return false;
    }
    private boolean isInterface(final ClassReader classReader) {
        return (classReader.getAccess() & Opcodes.ACC_INTERFACE) != 0;
    }
    private String getSuperClassName(final String className) {
        final ClassReader classReader = getClassReader(className);
        if (classReader == null) {
            return null;
        }
        return classReader.getSuperName();
    }
    private ClassReader getClassReader(final String className) {
        InputStream inputStream = urlClassLoader.getResourceAsStream(className
+ ".class");
        try {
            if (inputStream != null) {
                return new ClassReader(inputStream);
            }
        } catch (IOException ignored) {
        } finally {
            if (inputStream != null) {
                try {
                    inputStream.close();
                } catch (IOException ignored) {
                }
            }
        }
        return null;
    }
```

```
}
```

到此为止，我们介绍了在Android上实现修改字节码的两个基础技术Transform+ASM，介绍了其原理和应用，分析了性能优化以及在Android平台上的适配等。

万事俱备，只欠写一个插件来玩玩了，让我们来看看个应用案例。

应用案例：统计方法耗时

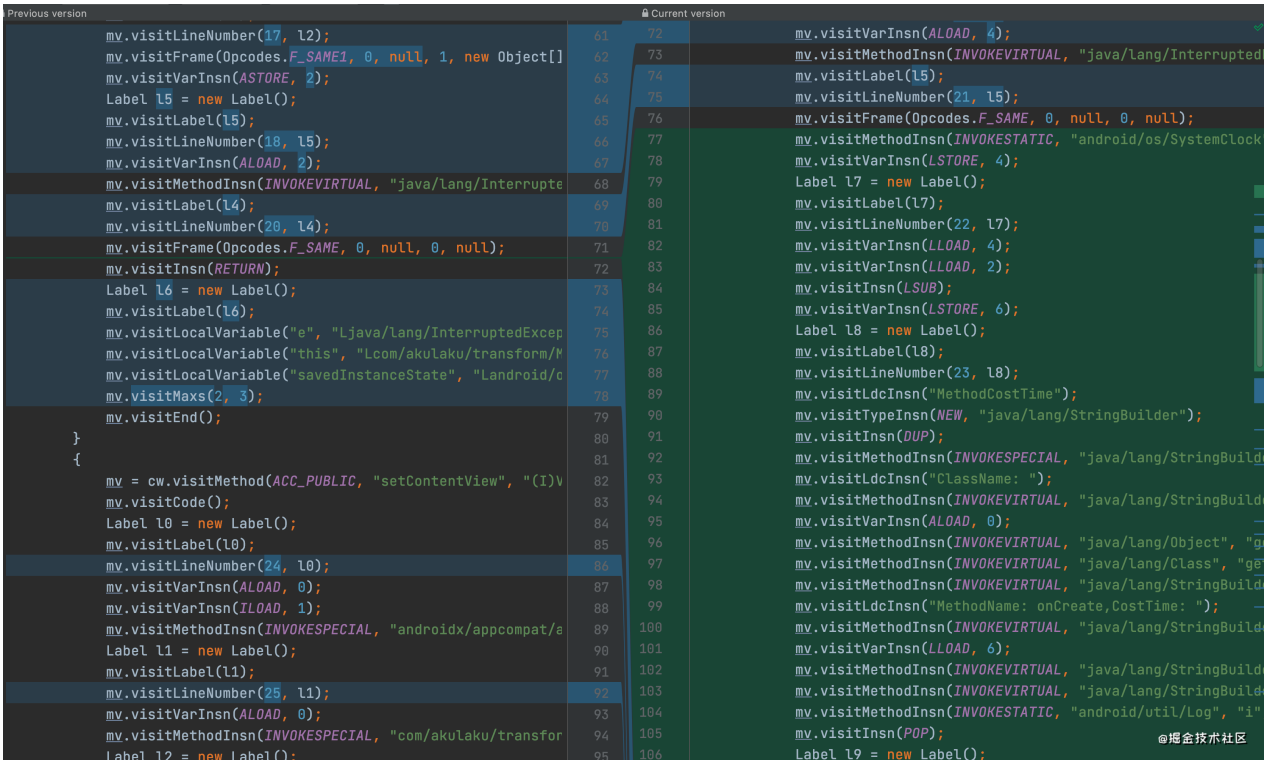
使用 ASM 编译插桩统计方法耗时主要可以细分为如下三个步骤：

- (1）**首先，我们需要通过自定义 gradle plugin 的形式来干预编译过程**。
- (2）**然后，在编译过程中获取到所有的 class 文件和 jar 包，然后遍历他们**。
- (3）**最后，利用 ASM 来修改字节码，达到插桩的目的**。

刚开始的时候，我们可以在 Activity 的 onCreate 方法 **先写下要插桩之后的代码**，如下所示：

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    long startTime = SystemClock.elapsedRealtime();
    super.onCreate(savedInstanceState);
    try {
        Thread.sleep(203);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    long endTime = SystemClock.elapsedRealtime();
    long duration = endTime - startTime;
    Log.i("MethodCostTime", "ClassName: " + getClass().getSimpleName() +
",MethodName: "+ "onCreate," + "CostTime: " + duration);
}
```

这样便于 **之后能使用 ASM Bytecode Outline 的 ASMified 下的 Show differences 去展示相邻两次修改的代码差异**，其修改之后 ASM 代码对比图如下所示：



在右图中所示的差异代码就是我们需要添加的 ASM 代码。这里我们直接使用 ASM 的事件模式，即 ASM 的 Core API 来进行字节码的读取与修改，代码如下所示：

```
ClassReader classReader = new ClassReader(inputStream);
ClassWriter classWriter = new ExtendClassWriter(classLoader,
ClassWriter.COMPUTE_MAXS);
//1
ClassVisitor classWriterWrapper = wrapClassWriter(classWriter);
classReader.accept(classWriterWrapper, ClassReader.EXPAND_FRAMES);
```

上面的实现代码我们在上面已经详细分析过了，当 classReader 调用 accept 方法时就会对类文件进行读取和被 classVisitor 访问。那么，我们是如何对方法中的字节码进行操作的呢?

在注释1处，我们通过 **wrapClassWriter()** 方法委托至具体的实现类 MethodCostTimeClassVisitor，其实现代码如下所示

```
public class MethodCostTimeClassVisitor extends ClassVisitor implements
Opcodes {

    public MethodCostTimeClassVisitor(ClassVisitor classVisitor) {
        super(Opcodes.ASM5, classVisitor);
    }

    @Override
    public void visit(int version, int access, String name, String signature,
String superName,
```

```
                      String[] interfaces) {
        super.visit(version, access, name, signature, superName, interfaces);
    }

    @Override
    public MethodVisitor visitMethod(int access, String name, String desc,
String signature,
                                     String[] exceptions) {
        MethodVisitor mv = super.visitMethod(access, name, desc, signature,
exceptions);
        return new MethodCostTimeMethodVisitor(api, mv, access, name, desc);
    }

}
```

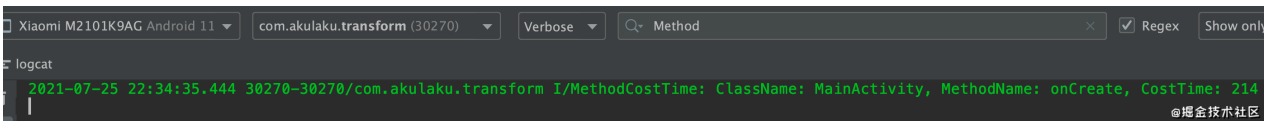
由于我们只需要对方法的字节码进行操作，直接处理 `visitMethod` 这个方法即可。**在这里我们直接将类观察者 ClassVisitor 通过访问得到的 MethodVisitor 进行了封装，使用了自定义的 AdviceAdapter 的方式来实现，而 AdviceAdapter 也是 MethodVisitor 的子类，不同于 MethodVisitor的是，它自身提供了 onMethodEnter 与 onMethodExit 方法，非常便于我们去实现方法的前后插桩**。其实现代码如下所示：

```
public class MethodCostTimeMethodVisitor extends AdviceAdapter {

    /** 是否统计方法的耗时 */
    private boolean mIsCountCostTime;

    /** 方法起始时间位于局部变量表的下标 */
    private int mStartTimeLocalIndex;

    /** 方法名称 */
    private String mMethodName;

    public MethodCostTimeMethodVisitor(int api, MethodVisitor mv, int access,
                                       String name, String desc) {
        super(api, mv, access, name, desc);
        mMethodName = name;
    }

    @Override
    public AnnotationVisitor visitAnnotation(String desc, boolean visible) {
        if ("Lcom/ffg/annotation/CostTime;".equals(desc)) {
            // 带有注解统计方法耗时
            mIsCountCostTime = true;
        }
        return super.visitAnnotation(desc, visible);
    }

    /**
     * 方法执行前
```

```
     */
    @Override
    protected void onMethodEnter() {
        super.onMethodEnter();
        if (!mIsCountCostTime) {
            // 没有注解不处理
            return;
        }
        // 调用的SystemClock.elapsedRealtime()方法的返回值位于操作数栈顶
        mv.visitMethodInsn(INVOKESTATIC, "android/os/SystemClock",
"elapsedRealtime", "()J", false);
        // 开辟一个局部标量的存储位置，并获取下标
        mStartTimeLocalIndex = newLocal(Type.LONG_TYPE);
        // 将栈顶的数存入指定下标的局部标量表
        mv.visitVarInsn(LSTORE, mStartTimeLocalIndex);
    }

    /**
     * 方法执行后
     */
    @Override
    protected void onMethodExit(int opcode) {
        super.onMethodExit(opcode);
        if (!mIsCountCostTime) {
            // 没有注解不处理
            return;
        }

        // 调用的SystemClock.elapsedRealtime()方法的返回值位于操作数栈顶
        mv.visitMethodInsn(INVOKESTATIC, "android/os/SystemClock",
"elapsedRealtime", "()J", false);
        // 将局部变量表对应下标的值放到栈顶
        mv.visitVarInsn(LLOAD, mStartTimeLocalIndex);
        // 栈顶的两个数执行减法操作，结果值位于栈顶
        mv.visitInsn(LSUB);
        // 将栈顶的数存入指定下标的局部标量表
        mv.visitVarInsn(LSTORE, mStartTimeLocalIndex);

        // 执行代码 Log.i("MethodCostTime", "ClassName: " +
getClass().getSimpleName() + "MethodName: "+ 方法名 + "CostTime: " + 方法耗时);

        // 访问常量池
        mv.visitLdcInsn("MethodCostTime");
        mv.visitTypeInsn(NEW, "java/lang/StringBuilder");
        mv.visitInsn(DUP);
        mv.visitMethodInsn(INVOKESPECIAL, "java/lang/StringBuilder", "<init>",
"()V", false);
        mv.visitLdcInsn("ClassName: ");
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/StringBuilder", "append",
```

```
                "(Ljava/lang/String;)Ljava/lang/StringBuilder;", false);
        // this压倒操作数栈顶
        mv.visitVarInsn(ALOAD, 0);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Object", "getClass", "
()Ljava/lang/Class;",
                false);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Class", "getSimpleName",
                "()Ljava/lang/String;", false);
        // 拼接getSimpleName的返回值
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/StringBuilder", "append",
                "(Ljava/lang/String;)Ljava/lang/StringBuilder;", false);
        // 方法名
        mv.visitLdcInsn(", MethodName: " + mMethodName + ", CostTime: ");
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/StringBuilder", "append",
                "(Ljava/lang/String;)Ljava/lang/StringBuilder;", false);
        // 拼接耗时
        mv.visitVarInsn(LLOAD, mStartTimeLocalIndex);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/StringBuilder", "append",
                "(J)Ljava/lang/StringBuilder;", false);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/StringBuilder",
"toString",
                "()Ljava/lang/String;", false);
        mv.visitMethodInsn(INVOKESTATIC, "android/util/Log", "i",
                "(Ljava/lang/String;Ljava/lang/String;)I", false);
        // Log.i有返回值要出栈
        mv.visitInsn(POP);
    }
}
```

然后在app的gradle脚本中添加脚本，运行app，验证结果：



三、总结

在 ASM Bytecode Outline 工具的帮助下，我们能够完成很多场景下的 ASM 插桩的需求，但是，当我们使用其处理字节码的时候还是需要考虑很多种可能出现的情况。如果想要具备这方面的深度思考能力，我们就 **必须对每一个操作码的特征都有较深的了解**，如果还不了解的同学可以去看看 《深入探索编译插桩技术（三、JVM字节码）》。因此，**要具备实现一个复杂 ASM 插桩的能力，我们需要对 JVM 字节码、ASM 字节码以及 ASM 源码中的核心工具类的实现 做到了然于心，并且在不断地实践与试错之后，我们才能够成为一个真正的 ASM 插桩高手**。