一、概述

> Starting with 1.5.0-beta1, the Gradle plugin includes a Transform API allowing 3rd party plugins to manipulate compiled class files before they are converted to dex files.(The API existed in 1.4.0-beta2 but it's been completely revamped in 1.5.0-beta1)

Android Gradle 工具在 1.5.0 版本后提供了 Transfrom API, 允许第三方 Plugin 在打包 dex 文件之前的编译过程中操作 .class 文件。目前 jarMerge、proguard、multi-dex、Instant-Run 都已经换成 Transform 实现。

二、分析

从官方的描述中得知：

1. Transform API 是新引进的操作 class 的方式
2. Transform API 在编译之后，生成 dex 之前起作用

在翻查文档以及结合之前自己实现 Plugin 的经验，想到的几个问题：

1. Transform 是如何拿到 class 文件的?
2. Transform 与 Gradle Task 之间的关系?
3. 为什么 Transform 的作用域在编译之后， 生成 Dex 之前，Gradle 是如何控制的?
4. 既然 Instant-Run 使用 Transform 实现，那 Transform 是如何得到变更的内容的?
5. Transform 之间的依赖关系是怎样的?

1. Transform

在解答问题之前，先看下 Transform 长什么样：

```
public class TestTransform extends Transform {
    @Override
    public String getName() {
        return null;
    }

    @Override
    public Set<QualifiedContent.ContentType> getInputTypes() {
        return null;
    }

    @Override
    public Set<? super QualifiedContent.Scope> getScopes() {
        return null;
    }

    @Override
    public boolean isIncremental() {
        return false;
    }

    @Override
```

```
    public void transform(TransformInvocation transformInvocation) throws
TransformException, InterruptedException, IOException {
        super.transform(transformInvocation);
    }
}
```

name: 给 transform 起个名字。 这个 name 并不是最终的名字， 在 TransformManager 中会对名字再处理：

```
    static String getTaskNamePrefix(Transform transform) {
        StringBuilder sb = new StringBuilder(100);
        sb.append("transform");
        sb.append((String)transform.getInputTypes().stream().map((inputType) ->
{
            return CaseFormat.UPPER_UNDERSCORE.to(CaseFormat.UPPER_CAMEL,
inputType.name());

}).sorted().collect(Collectors.joining("And"))).append("With").append(StringHelp
er.capitalize(transform.getName())).append("For");
        return sb.toString();
    }
```

inputTypes: transform 要处理的数据类型。

- CLASSES 表示要处理编译后的字节码，可能是 jar 包也可能是目录
- RESOURCES 表示处理标准的 java 资源

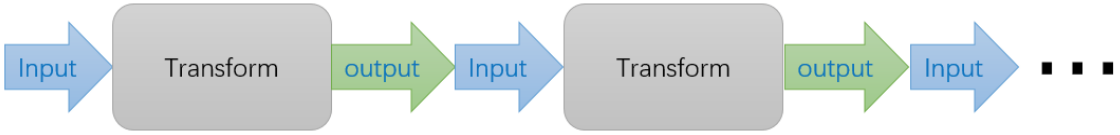
scopes：transform 的作用域

type	Des
PROJECT	只处理当前项目
SUB_PROJECTS	只处理子项目
PROJECT_LOCAL_DEPS	只处理当前项目的本地依赖,例如jar, aar
EXTERNAL_LIBRARIES	只处理外部的依赖库
PROVIDED_ONLY	只处理本地或远程以provided形式引入的依赖库
TESTED_CODE	测试代码

> ContentType 和 Scopes 都返回集合，TransformManager 中封装了默认的几种集中类型

isIncremental : 当前 Transform 是否支持增量编译

Transform 的工作流程



Transform 将输入进行处理，然后写入到指定的目录下作为下一个 Transform 的输入源。

获取输出路径：

```
destDir = transformInvocation.outputProvider.getContentLocation(dirInput.name,
dirInput.contentTypes, dirInput.scopes, Format.DIRECTORY)
```

2. 案例解读

Metis 是一个 Android 的 SPI 实现，解决运行时获取指定的服务类型。
主要原理是用注解标记指定的类型，插件在编译过程中扫描所有的 class；对被注解标记过的类动态生成一个 java 源文件，再将 java 文件编译之后会被打包进 dex； 运行时只要调用工具类的方法执行查询操作即可。
动态生成的源文件：

```
final class MetisRegistry {
  private static final Map<Class<?>, HashSet<Class<?>>> sServices = new
LinkedHashMap<Class<?>, HashSet<Class<?>>>();

  static {
    register(io.github.yangxiaolei.sub.TestAction.class,
io.github.yangxiaolei.sub.TestAction1.class);
    register(io.github.yangxiaolei.sub.TestAction.class,
io.github.yangxlei.TestAction3.class);
    register(io.github.yangxlei.TestAction3.class,
io.github.yangxlei.MainActivity.class);
  }

  static final Set<Class<?>> get(Class<?> key) {
    Set<Class<?>> result = sServices.get(key);
    return null == result ? Collections.<Class<?>>emptySet() :
Collections.unmodifiableSet(result);
  }

  private static final void register(Class key, Class<?> value) {
    HashSet<Class<?>> result = sServices.get(key);
    if (result == null) {
      result = new HashSet<Class<?>>();
      sServices.put(key, result);
    }
    result.add(value);
  }
}
```

①. 如何获取 class 文件

```
    @Override
    public Set<QualifiedContent.ContentType> getInputTypes() {
        return TransformManager.CONTENT_CLASS;
    }

    @Override
    public Set<? super QualifiedContent.Scope> getScopes() {
        return TransformManager.SCOPE_FULL_PROJECT;
    }

    @Override
    public boolean isIncremental() {
```

```
        return true;
    }
```

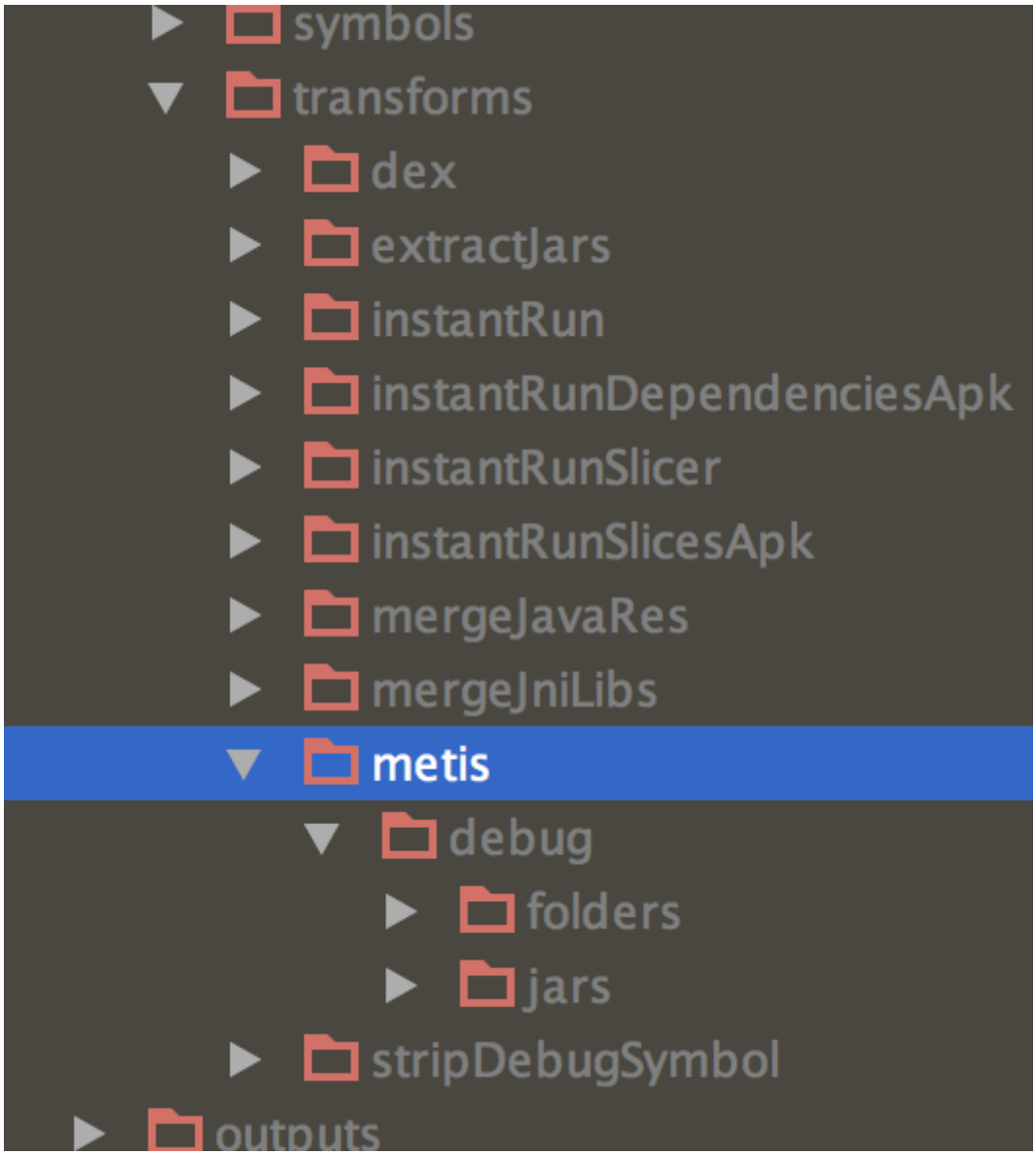
配置 Transform 的输入类型为 Class， 作用域为全工程。 这样在 transform(TransformInvocation transformInvocation) 方法中， transformInvocation.inputs 会传入工程内所有的 class 文件。

inputs 包含两个部分：

```
public interface TransformInput {
    Collection<JarInput> getJarInputs();

    Collection<DirectoryInput> getDirectoryInputs();
}
```

看接口方法可知，包含了 jar 包和目录。子 module 的 java 文件在编译过程中也会生成一个 jar 包然后编译到主工程中。app/build 的目录下可以看到 class 分别在 folders 和 jars 两个目录下：



②. Transform 与 Gradle Task 之间的关系?

Gradle 包中有一个 TransformManager 的类，用来管理所有的 Transform。 在里面找到了这样的代码：

```
    public <T extends Transform> Optional<AndroidTask<TransformTask>>
addTransform(TaskFactory taskFactory, TransformVariantScope scope, T transform,
ConfigActionCallback<T> callback) {
                ...
                this.transforms.add(transform);
                AndroidTask task1 = this.taskRegistry.create(taskFactory, new
ConfigAction(scope.getFullVariantName(), taskName, transform, inputStreams,
referencedStreams, outputStream, this.recorder, callback));
                ...
                return Optional.ofNullable(task1);
            }
        }
    }
```

addTransform 方法在执行过程中，会将 Transform 包装成一个 AndroidTask 对象。
所以可以理解为**一个 Transform 就是一个 Task**

③. Gradle 是如何控制 Transform 的作用域的?

还是在 Gradle 的包中有一个 TaskManager 类，管理所有的 Task 执行。 其中有一个方法：

```
    public void createPostCompilationTasks(TaskFactory tasks, VariantScope
variantScope) {
        ...
        List customTransforms = extension.getTransforms();
        List customTransformsDependencies =
extension.getTransformsDependencies();
        int preColdSwapTask = 0;

        for(int multiDexClassListTask = customTransforms.size(); preColdSwapTask
< multiDexClassListTask; ++preColdSwapTask) {
            Transform dexOptions =
(Transform)customTransforms.get(preColdSwapTask);
            List dexTransform =
(List)customTransformsDependencies.get(preColdSwapTask);
            transformManager.addTransform(tasks, variantScope,
dexOptions).ifPresent((t) -> {
                if(!dexTransform.isEmpty()) {
                    t.dependsOn(tasks, dexTransform);
                }

                if(dexOptions.getScopes().isEmpty()) {
                    variantScope.getAssembleTask().dependsOn(tasks, t);
                }

            });
        }
        ...
    }
```

该方法在 javaCompile 之后调用， 会遍历所有的 transform，然后一一添加进 TransformManager。加完自定义的 Transform 之后，再添加 Proguard, JarMergeTransform, MultiDex, Dex 等 Transform。

postCompilation 的调用：

```
    if(jackOptions1.isEnabled().booleanValue()) {
            javacTask = this.createJackTask(tasks, variantScope, true);
            setJavaCompilerTask(javacTask, tasks, variantScope);
        } else {
            javacTask = this.createJavacTask(tasks, variantScope);
            addJavacClassesStream(variantScope);
            setJavaCompilerTask(javacTask, tasks, variantScope);
            this.createPostCompilationTasks(tasks, variantScope);
        }
```

调用时判断是使用 jack 编译还是 javac 编译。 javac 编译完之后再组装 Transform。看了源码之后，也可以回答 Transform 之间的依赖关系：

- 因为是遍历 List 顺序添加的，所以可以在 Plugin 中通过先后顺序一一添加
- registerTransform 方法第二个参数是 dependsOn， 可以手动设置依赖关系

④. 如何得到文件的增量

再回到 TransformInput 这个接口，输入源分为 JarInput 和 DirectoryInput

```
public interface JarInput extends QualifiedContent {
    Status getStatus();
}
```

Status 是一个枚举：

```
public enum Status {
    NOTCHANGED,
    ADDED,
    CHANGED,
    REMOVED;
}
```

所以在输入源中， 获取了 JarInput 的对象时，可以同时得到每个 jar 的变更状态。需要注意的是：比如先 clean 再编译时， jar 的状态是 NOTCHANGED

再看看 DirectoryInput：

```
public interface DirectoryInput extends QualifiedContent {
    Map<File, Status> getChangedFiles();
}
```

changedFiles 是一个 Map，其中会包含所有变更后的文件，以及每个文件对应的状态。同样需要注意的是：先 clean 再编译时， changedFiles 是空的。

所以在处理增量时，只需要根据每个文件的状态进行相应的处理即可，不需要每次所有流程都重新来一遍。

3. 踩了的坑

Transform 是用来处理 class 文件的， 但是在 Metis 的实现时，需要生成 java 源文件，再将 java 文件编译一下。之前的实现方式是：

- 创建一个 generateSourceCode 的 task，依赖 JavaCompile, 这样可以在整体编译完成之后拿到所有的 class 文件

- 再创建一个 compileSourceCode 的 task，在 generateSourceCode 执行完成后编译动态生成的 java 源码

但是现在 Transform 并不是原生的 task， 没有找到合适的办法让 task 依赖 transfrom(谁要是有好办法告诉我~~)。

现在的解决办法是在 MetisTransform 生成完 java 源文件之后，主动调用 javac 来编译文件。

然后开始了踩坑之旅。。

①. 怎么得到 sourceCompatibility & targetCompatibility 版本

调用 javac 需要兼容指定的版本，sourceCompatibility 和 targetCompatibility 有时候会配置，有时候不会配置会有默认值。但是在 Transform 如何得到这两个值呢？翻查源码时找到了 JavaCompile 包含这两个属性，所以只要能找到 JavaCompile 这个 task，就能得到这两个值：

```
        def sourceCompatibility
        def targetCompatibility
        def bootClasspath
        mProject.tasks.each { task ->
            if (AbstractCompile.isAssignableFrom(task.class)) {
                sourceCompatibility = task.sourceCompatibility
                targetCompatibility = task.targetCompatibility
            }

            if (JavaCompile.isAssignableFrom(task.class)) {
                bootClasspath = task.options.bootClasspath
            }
        }
```

bootClassPath 的值获取采用同样的方法。

②. javac 在哪?

不同的系统 javac 的配置是不一样的。在 bash 环境下可以通过

```
which javac
```

获取到 javac 的路径。在 Project 类中找到一个 exec 的方法，用来执行命令

```
    def getJavac() {
        def stdOut = new ByteArrayOutputStream()
        mProject.exec {
            commandLine 'which'
            args 'javac'
            standardOutput = stdOut
        }

        return stdOut.toString().trim()
    }
```

一定要 trim() !!!

③. commandLine 的坑

到这正常应该已经没有问题了，只需要再调用 exec 执行 javac 命令就可以了。但是...javac 的命令在程序中是一个变量, 正常代码会是这样：

```
def javac = getJavac()
 mProject.exec {
            commandLine javac
            args "xxx", "xxx", "xxx"
        }
```

然后就报异常： command property is null!但是 commandLine 后面直接配置 '/usr/bin/javac' 能编译成功。我也不知道为什么。。 谁要是知道一定要告诉我！！

最后通过曲线救国， 将 javac 命令写入到一个 shell 文件中，然后再 exec 中执行一个 shell 脚本。

```
   def generateCompileShell(tempDir, javac, sourceCompatibility,
targetCompatibility, sourceFile, destDir, bootClasspath, classpaths) {
        def shellFile = new File(tempDir, "compileMetisShell.sh")
        if (shellFile.exists()) shellFile.delete()

        shellFile.append("#!/bin/sh")

        shellFile.append("\n")

        shellFile.append("${javac} -source ${sourceCompatibility} -target
${targetCompatibility} ${sourceFile} -d ${destDir}")

        shellFile.append(" -bootclasspath ${bootClasspath}")

        shellFile.append(" -classpath ")

        classpaths.each { classpath ->
            shellFile.append("${classpath}:")
        }

        return shellFile
    }
 ExecResult result = mProject.exec {
            executable 'sh'
            args shell.absolutePath
        }
```

写在最后

这次使用 Transform api 重新实现 Metis 的插件工具，翻查了很多文档，但是很少有对 transform 讲的很详细。一步一步摸索出来感觉收获良多。