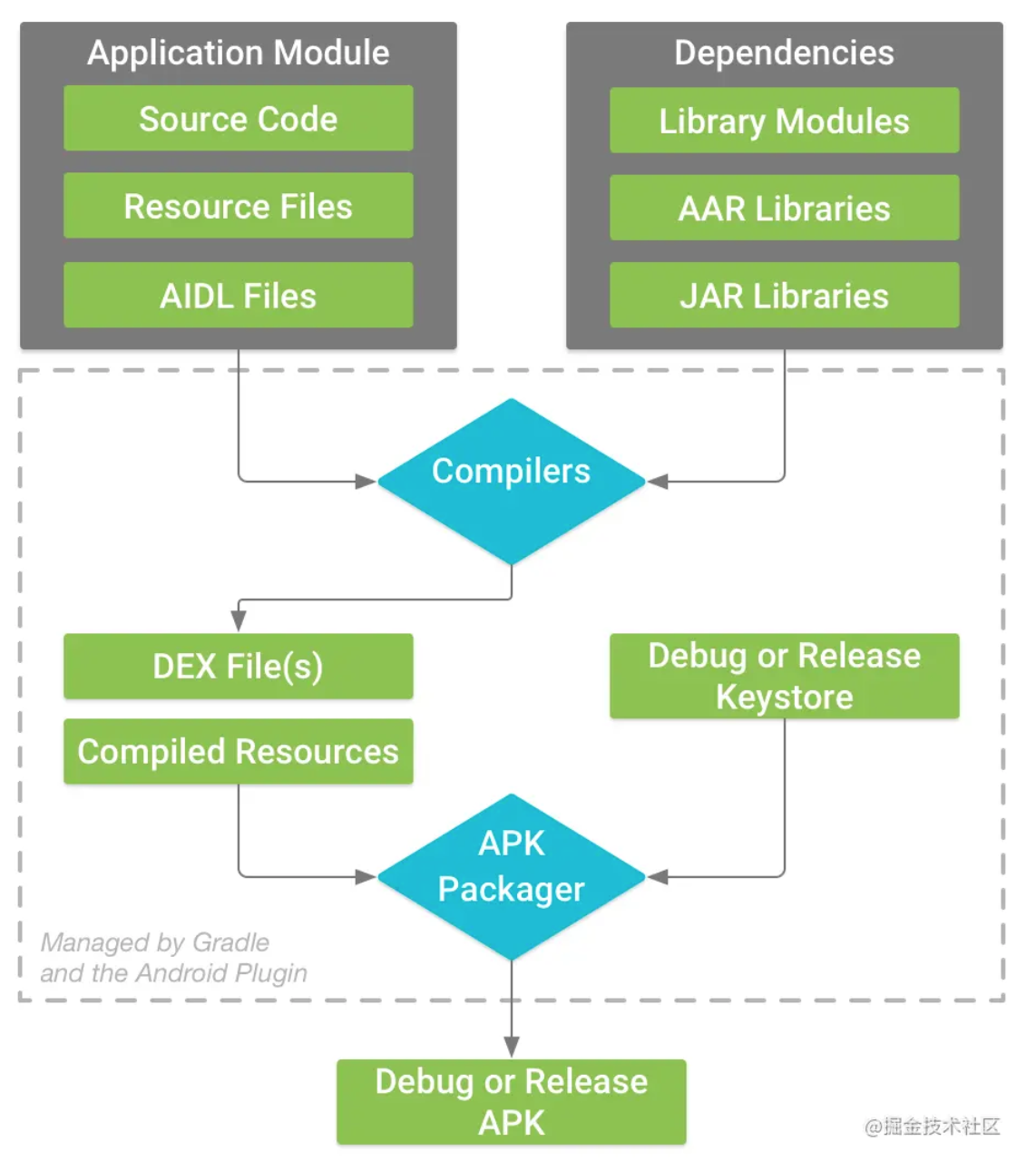
**transform简介**

在打包流程中，我们知道生成.class文件后，利用dx工具生成.dex文件，而利用Transform API可以在生成.class文件后修改.class文件，从而修改源码。我们将Transform注册到AppExtension中，在java compile Task执行后会执行Tramsform类型的task。



**transform执行机制**

主要的逻辑代码是对jar包以及.class文件进行扫描，当文件符合修改标准的情况下会回调文件修改的方法，然后基于asm的classvisitor 对文件进行访问操作。

构建transform

abstract class DoubleTapTransform : Transform() {  
 override fun getName(): String {  
 return "DoubleTapTransform"  
 }  
  
 override fun isIncremental(): Boolean {  
 return true  
 }  
  
 @Throws(TransformException::class, InterruptedException::class, IOException::class)  
 override fun transform(transformInvocation: TransformInvocation) {  
 val injectHelper = DoubleTapClassNodeHelper()  
 val baseTransform = BaseTransform(transformInvocation, object : TransformCallBack {  
 override fun process(className: String, classBytes: ByteArray?): ByteArray? {  
 if (ClassUtils.checkClassName(className)) {  
 try {  
 return classBytes?.*let* **{** injectHelper.modifyClass(**it**) **}** } catch (e: IOException) {  
 e.printStackTrace()  
 }  
 }  
 return null  
 }  
 })  
 baseTransform.startTransform()  
 }  
  
 override fun isCacheable(): Boolean {  
 return true  
 }  
}

上述代码对transform 以及classvisitor代码进行了一次抽象封装，方便后续如果有类似的插入逻辑可以快速接入开发。

class BaseTransform(  
 transformInvocation: TransformInvocation?,  
 callBack: TransformCallBack,  
 single: Boolean = false  
) {  
 private var mCallBack: TransformCallBack? = callBack  
 var context: Context? = null  
 private var inputs: Collection<TransformInput>? = null  
 private var outputProvider: TransformOutputProvider? = null  
 private var isIncremental = false  
 private var deleteCallBack: DeleteCallBack? = null  
 private var simpleScan = false  
 var filter: ClassNameFilter? = null  
 private val executor: ExecutorService  
 private val tasks: MutableList<Callable<Void>> = ArrayList()  
 private val destFiles = *mutableListOf*<File>()  
  
 init {  
 context = transformInvocation?.*context* inputs = transformInvocation?.*inputs* outputProvider = transformInvocation?.*outputProvider* isIncremental = transformInvocation?.*isIncremental* ?: false  
 executor = if (!single) {  
 ForkJoinPool.commonPool()  
 } else {  
 Executors.newSingleThreadExecutor()  
 }  
 }  
  
 fun openSimpleScan() {  
 simpleScan = true  
 }  
  
 fun setDeleteCallBack(deleteCallBack: DeleteCallBack?) {  
 this.deleteCallBack = deleteCallBack  
 }  
  
 fun startTransform() {  
 try {  
 val startTimeUsage = System.currentTimeMillis()  
 if (!isIncremental) {  
 outputProvider?.deleteAll()  
 }  
 inputs?.*forEach* **{** input **->** for (jarInput in input.*jarInputs*) {  
 val status = jarInput.*status* var destName = jarInput.*file*.*name  
 /\* 重名名输出文件,因为可能同名,会覆盖\*/* val hexName = DigestUtils.md5Hex(jarInput.*file*.*absolutePath*).*substring*(0, 8)  
 if (destName.*endsWith*(".jar")) {  
 destName = destName.*substring*(0, destName.length - 4)  
 }  
 */\*获得输出文件\*/* val dest = outputProvider!!.getContentLocation(  
 destName + "\_" + hexName,  
 jarInput.*contentTypes*, jarInput.*scopes*, Format.*JAR* )  
 if (isIncremental) {  
 when (status) {  
 Status.*ADDED* -> foreachJar(dest, jarInput)  
 Status.*CHANGED* -> diffJar(dest, jarInput)  
 Status.*REMOVED* -> try {  
 deleteScan(dest)  
 if (dest.exists()) {  
 FileUtils.forceDelete(dest)  
 }  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 else -> {  
  
 }  
 }  
 } else {  
 foreachJar(dest, jarInput)  
 }  
 }  
 for (directoryInput in input.*directoryInputs*) {  
 foreachClass(directoryInput)  
 }  
 **}** executor.invokeAll(tasks)  
  
 destFiles.*forEach* **{  
 it**.*filterTest*("temp")?.*forEach* **{** file **->** file.*deleteAll*()  
 **}  
 }** val timeUsage = System.currentTimeMillis() - startTimeUsage  
 Log.info("transform coast time: $timeUsage ms")  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 }  
  
 @Throws(IOException::class)  
 private fun foreachClass(directoryInput: DirectoryInput) {  
 val dest = outputProvider!!.getContentLocation(  
 directoryInput.*name*, directoryInput.*contentTypes*,  
 directoryInput.*scopes*, Format.*DIRECTORY* )  
 destFiles.add(dest)  
 val map = directoryInput.*changedFiles* val dir = directoryInput.*file* if (isIncremental) {  
 for ((file, status) in map) {  
 val destFilePath = file.*absolutePath*.*replace*(dir.*absolutePath*, dest.*absolutePath*)  
 val destFile = File(destFilePath)  
 when (status) {  
 Status.*ADDED*, Status.*CHANGED* -> {  
 val callable = *Callable*<Void> **{** try {  
 FileUtils.touch(destFile)  
 } catch (ignored: Exception) {  
 *// Files.createParentDirs(destFile)* }  
 modifySingleFile(dir, file, destFile)  
 null  
 **}** tasks.add(callable)  
 }  
 Status.*REMOVED* -> deleteDirectory(destFile, dest)  
 else -> {  
 }  
 }  
 }  
 } else {  
 changeFile(dir, dest)  
 }  
 }  
  
 private fun deleteDirectory(destFile: File, dest: File) {  
 try {  
 if (destFile.*isDirectory*) {  
 destFile.*walkTopDown*().*forEach* **{** classFile **->** deleteSingle(classFile, dest)  
 **}** } else {  
 deleteSingle(destFile, dest)  
 }  
 } catch (ignored: Exception) {  
 }  
 try {  
 if (destFile.exists()) {  
 FileUtils.forceDelete(destFile)  
 }  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 }  
  
 private fun deleteSingle(classFile: File, dest: File) {  
 try {  
 if (classFile.*name*.*endsWith*(".class")) {  
 val absolutePath = classFile.*absolutePath*.*replace*(  
 dest.*absolutePath* +  
 File.*separator*, ""  
 )  
 val className = ClassUtils.path2Classname(absolutePath)  
 val bytes = IOUtils.toByteArray(FileInputStream(classFile))  
 deleteCallBack?.delete(className, bytes)  
  
 }  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 }  
  
 @Throws(IOException::class)  
 private fun modifySingleFile(dir: File, file: File, dest: File) {  
 try {  
 val absolutePath = file.*absolutePath*.*replace*(  
 dir.*absolutePath* +  
 File.*separator*, ""  
 )  
 val className = ClassUtils.path2Classname(absolutePath)  
 if (absolutePath.*endsWith*(".class")) {  
 var modifiedBytes: ByteArray?  
 val bytes = IOUtils.toByteArray(FileInputStream(file))  
 modifiedBytes = if (!simpleScan) {  
 process(className, bytes)  
 } else {  
 process(className, null)  
 }  
 if (modifiedBytes == null) {  
 modifiedBytes = bytes  
 }  
 ClassUtils.saveFile(dest, modifiedBytes)  
 } else {  
 if (!file.*isDirectory*) {  
 *copyIfLegal*(file, dest)  
 }  
 }  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 }  
  
 fun process(className: String, classBytes: ByteArray?): ByteArray? {  
 try {  
 if (filter == null) {  
 filter = DefaultClassNameFilter()  
 }  
 if (filter?.filter(className) == false) {  
 return mCallBack?.process(className, classBytes)  
 }  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 return null  
 }  
  
 @Throws(IOException::class)  
 private fun changeFile(dir: File, dest: File) {  
 if (dir.*isDirectory*) {  
 FileUtils.copyDirectory(dir, dest)  
 dir.*walkTopDown*().*filter* **{ it**.*isFile* **}** .*forEach* **{** classFile **->** if (classFile.*name*.*endsWith*(".class")) {  
 val task = *Callable*<Void> **{** val absolutePath = classFile.*absolutePath*.*replace*(  
 dir.*absolutePath* + File.*separator*, ""  
 )  
 val className = ClassUtils.path2Classname(absolutePath)  
 if (!simpleScan) {  
 val bytes = IOUtils.toByteArray(FileInputStream(classFile))  
 val modifiedBytes = process(className, bytes)  
 modifiedBytes?.*let* **{** saveClassFile(**it**, dest, absolutePath) **}** } else {  
 process(className, null)  
 }  
 null  
 **}** tasks.add(task)  
 }  
 **}** }  
 }  
  
 @Throws(Exception::class)  
 private fun saveClassFile(modifiedBytes: ByteArray, dest: File, absolutePath: String) {  
 val tempDir = File(dest, "/temp")  
 val tempFile = File(tempDir, absolutePath)  
 tempFile.mkdirs()  
 val modified = ClassUtils.saveFile(tempFile, modifiedBytes)  
 *//key为相对路径* val target = File(dest, absolutePath)  
 if (target.exists()) {  
 target.delete()  
 }  
 *copyIfLegal*(modified, target)  
 tempFile.delete()  
 }  
  
 private fun foreachJar(dest: File, jarInput: JarInput) {  
 val task = *Callable*<Void> **{** try {  
 if (!simpleScan) {  
 val modifiedJar = JarUtils.modifyJarFile(jarInput.*file*, context?.*temporaryDir*, this)  
 *copyIfLegal*(modifiedJar, dest)  
 } else {  
 val jarFile = jarInput.*file* val classNames = JarUtils.scanJarFile(jarFile)  
 for (className in classNames) {  
 process(className, null)  
 }  
 *copyIfLegal*(jarFile, dest)  
 }  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 null  
 **}** tasks.add(task)  
 }  
  
 private fun diffJar(dest: File, jarInput: JarInput) {  
 try {  
 val oldJarFileName = JarUtils.scanJarFile(dest)  
 val newJarFileName = JarUtils.scanJarFile(jarInput.*file*)  
 val diff = SetDiff(oldJarFileName, newJarFileName)  
 val removeList = diff.*removedList* if (removeList.size > 0) {  
 JarUtils.deleteJarScan(dest, removeList, deleteCallBack)  
 }  
 foreachJar(dest, jarInput)  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 }  
  
 private fun deleteScan(dest: File) {  
 try {  
 JarUtils.deleteJarScan(dest, deleteCallBack)  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 }  
  
}

object ClassUtils {  
 fun path2Classname(entryName: String): String {  
 *// 感谢大佬 dingshaoran  
 //ClassUtils.path2Classname(className); File.separator donot match jar entryName on windows* return entryName.*replace*(".class", "")  
 .*replace*('\\', '.')  
 .*replace*('/', '.')  
 }  
  
 fun checkClassName(className: String): Boolean {  
 if (className.*contains*("R\$")) {  
 return false  
 }  
 if (className.*endsWith*("R.class")) {  
 return false  
 }  
 return (!className.*contains*("R\\$") && !className.*endsWith*("R")  
 && !className.*endsWith*("BuildConfig"))  
 }  
  
 fun saveFile(mTempDir: File?, modifiedClassBytes: ByteArray?): File? {  
 val modified: File? = mTempDir  
 modifiedClassBytes?.*apply* **{** if (mTempDir!!.exists()) {  
 mTempDir.delete()  
 }  
 mTempDir.createNewFile()  
 val stream = FileOutputStream(mTempDir)  
 stream.*use* **{** stream.write(modifiedClassBytes)  
 **}  
 }** return modified  
 }  
}

class DoubleTapClassNodeHelper : AsmHelper {  
  
 private val classNodeMap = *hashMapOf*<String, ClassNode>()  
  
 @Throws(IOException::class)  
 override fun modifyClass(srcClass: ByteArray): ByteArray {  
 val classNode = ClassNode(*ASM5*)  
 val classReader = ClassReader(srcClass)  
 *//1 将读入的字节转为classNode* classReader.accept(classNode, 0)  
 classNodeMap[classNode.name] = classNode  
 *// 判断当前类是否实现了OnClickListener接口* val hasAnnotation = classNode.*hasAnnotation*()  
 val className = classNode.outerClass  
 val parentNode = classNodeMap[className]  
 val hasKeepAnnotation = if (hasAnnotation) {  
 true  
 } else {  
 parentNode?.*hasAnnotation*() ?: false  
 }  
 if (!hasKeepAnnotation) {  
 classNode.interfaces?.*forEach* **{** if (**it** == "android/view/View\$OnClickListener") {  
 classNode.methods?.*forEach* **{** method **->** *// 找到onClick 方法* if (method.name == "<init>") {  
 initFunction(classNode, method)  
 }  
 if (method.name == "onClick" && method.desc == "(Landroid/view/View;)V") {  
 insertTrack(classNode, method)  
 }  
 **}** }  
 **}** classNode.*lambdaHelper* **{** (**it**.name == "onClick" && **it**.desc.*contains*(")Landroid/view/View\$OnClickListener;"))  
 **}**.*apply* **{** if (*isNotEmpty*()) {  
 classNode.methods?.*forEach* **{** method **->** if (method.name == "<init>") {  
 initFunction(classNode, method)  
 return@forEach  
 }  
 **}** }  
 **}**.*forEach* **{** method **->** insertTrack(classNode, method)  
 **}** }  
 *//调用Fragment的onHiddenChange方法* val classWriter = ClassWriter(0)  
 *//3 将classNode转为字节数组* classNode.accept(classWriter)  
 return classWriter.toByteArray()  
 }  
  
  
 private fun insertLambda(node: ClassNode, method: MethodNode) {  
 *// 根据outClassName 获取到外部类的Node* }  
  
 private fun initFunction(node: ClassNode, method: MethodNode) {  
 var hasDoubleTap = false  
 node.fields?.*forEach* **{** if (**it**.name == "doubleTap") {  
 hasDoubleTap = true  
 }  
 **}** if (!hasDoubleTap) {  
 node.visitField(*ACC\_PRIVATE* + *ACC\_FINAL*, "doubleTap", String.*format*("L%s;",  
 DoubleTabConfig.ByteCodeInjectClassName), node.signature, null)  
 val instructions = method.instructions  
 method.instructions?.iterator()?.*forEach* **{** if ((**it**.*opcode* >= Opcodes.*IRETURN* && **it**.*opcode* <= Opcodes.*RETURN*) || **it**.*opcode* == Opcodes.*ATHROW*) {  
 instructions.insertBefore(**it**, VarInsnNode(*ALOAD*, 0))  
 instructions.insertBefore(**it**, TypeInsnNode(*NEW*, DoubleTabConfig.ByteCodeInjectClassName))  
 instructions.insertBefore(**it**, InsnNode(*DUP*))  
 instructions.insertBefore(**it**, MethodInsnNode(*INVOKESPECIAL*, DoubleTabConfig.ByteCodeInjectClassName,  
 "<init>", "()V", false))  
 instructions.insertBefore(**it**, FieldInsnNode(*PUTFIELD*, node.name, "doubleTap",  
 String.*format*("L%s;", DoubleTabConfig.ByteCodeInjectClassName)))  
 }  
 **}** }  
 }  
  
  
 private fun insertTrack(node: ClassNode, method: MethodNode) {  
 *// 判断方法名和方法描述* val instructions = method.instructions  
 val firstNode = instructions.*first* instructions?.insertBefore(firstNode, LabelNode(Label()))  
 instructions?.insertBefore(firstNode, VarInsnNode(*ALOAD*, 0))  
 instructions?.insertBefore(firstNode, FieldInsnNode(*GETFIELD*, node.name,  
 "doubleTap", String.*format*("L%s;", DoubleTabConfig.ByteCodeInjectClassName)))  
 instructions?.insertBefore(firstNode, MethodInsnNode(*INVOKEVIRTUAL*, DoubleTabConfig.ByteCodeInjectClassName,  
 DoubleTabConfig.ByteCodeInjectFunctionName, "()Z", false))  
 val labelNode = LabelNode(Label())  
 instructions?.insertBefore(firstNode, JumpInsnNode(*IFNE*, labelNode))  
 instructions?.insertBefore(firstNode, InsnNode(*RETURN*))  
 instructions?.insertBefore(firstNode, labelNode)  
 }  
  
  
 *// 判断Field是否包含注解* private fun ClassNode.hasAnnotation(): Boolean {  
 var hasAnnotation = false  
 this.visibleAnnotations?.*forEach* **{** annotation **->** *// Log.info("name:$name visibleAnnotations:${annotation.desc} ")* if (annotation.desc == "Lcom/wallstreetcn/sample/adapter/Test;") {  
 hasAnnotation = true  
 }  
 **}** this.invisibleAnnotations?.*forEach* **{** annotation **->** *// Log.info("name:$name visibleAnnotations:${annotation.desc} ")* if (annotation.desc == "Lcom/wallstreetcn/sample/adapter/Test;") {  
 hasAnnotation = true  
 }  
 **}** return hasAnnotation  
 }  
}

ClassVisitor机制

构造了一个类访问器，然后顺序的读取类的所以属性，方法，以及方法的每一行。

class ClassFilterVisitor extends ClassVisitor {  
  
 private String[] interfaces;  
 boolean visitedStaticBlock = false;  
 private String owner;  
  
 ClassFilterVisitor(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);  
 this.interfaces = interfaces;  
 if (interfaces != null && interfaces.length > 0) {  
 for (Map.Entry<String, MethodCell> entry : MethodHelper.*sInterfaceMethods*.entrySet()) {  
 MethodCell cell = entry.getValue();  
 for (String anInterface : interfaces) {  
 if (anInterface.equals(cell.parent)) {  
 visitedStaticBlock = true;  
 this.owner = name;  
 cv.visitField(Opcodes.*ACC\_PRIVATE* + Opcodes.*ACC\_FINAL*, "doubleTap",  
 String.*format*("L%s;", DoubleTabConfig.*ByteCodeInjectClassName*),  
 signature, null);  
 }  
 }  
 }  
 }  
 }  
  
  
 @Override  
 public MethodVisitor visitMethod(int access, String name,  
 String desc, String signature, String[] exceptions) {  
 if (interfaces != null && interfaces.length > 0) {  
 try {  
 if (visitedStaticBlock && name.equals("<init>")) {  
 MethodVisitor methodVisitor = cv.visitMethod(access, name, desc, signature, exceptions);  
 return new InitBlockVisitor(methodVisitor, owner);  
 }  
 MethodCell cell = MethodHelper.*sInterfaceMethods*.get(name + desc);  
 if (cell != null) {  
 for (String anInterface : interfaces) {  
 if (anInterface.equals(cell.parent)) {  
 MethodVisitor methodVisitor = cv.visitMethod(access, name, desc, signature, exceptions);  
 return new CheckVisitor(methodVisitor, owner);  
 }  
 }  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 return super.visitMethod(access, name, desc, signature, exceptions);  
 }

}

其中visit方法代表类被访问了，会返回这个类继承的接口等等基础参数。

然后就是classvistior的visitMethod，这个是我们主要要调整的地方，其中一个关键点是我们需要修改两个地方，一个类的初始化，另外一个onClick方法。

上述代码只完成了一件事情，就是在init 之后执行new DoubleTapCheck();这个操作。

# 增量编译定义

编译过程中会去遍历所有的jar .class文件，然后对文件进行io操作，以及asm插入代码，这个过程耗时一般都会很长。  
  
这里需要注意一点：不是每次的编译都是可以增量编译的，毕竟一次clean build完全没有增量的基础，所以，我们需要检查当前的编译是否增量编译。  
需要做区分：  
  
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

增量编译，则要检查每个文件的Status，Status分为四种，并且对四种文件的操作不尽相同  
  
**NOTCHANGED** 当前文件不需要处理，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  
  
上述是对增量的一些定义，可以看出来在transfrom过程中，应该是对文件打了一些tag标签。  
那么我们在开发阶段首先要先区分当前这次是不是增量编译，然后再编译当前变更的文件，对变更的文件进行处理。

fun startTransform() {  
 try {  
 val startTimeUsage = System.currentTimeMillis()  
 if (!isIncremental) {  
 outputProvider?.deleteAll()  
 }  
 inputs?.*forEach* **{** input **->** for (jarInput in input.*jarInputs*) {  
 val status = jarInput.*status* var destName = jarInput.*file*.*name  
 /\* 重名名输出文件,因为可能同名,会覆盖\*/* val hexName = DigestUtils.md5Hex(jarInput.*file*.*absolutePath*).*substring*(0, 8)  
 if (destName.*endsWith*(".jar")) {  
 destName = destName.*substring*(0, destName.length - 4)  
 }  
 */\*获得输出文件\*/* val dest = outputProvider!!.getContentLocation(  
 destName + "\_" + hexName,  
 jarInput.*contentTypes*, jarInput.*scopes*, Format.*JAR* )  
 if (isIncremental) {  
 when (status) {  
 Status.*ADDED* -> foreachJar(dest, jarInput)  
 Status.*CHANGED* -> diffJar(dest, jarInput)  
 Status.*REMOVED* -> try {  
 deleteScan(dest)  
 if (dest.exists()) {  
 FileUtils.forceDelete(dest)  
 }  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 else -> {  
  
 }  
 }  
 } else {  
 foreachJar(dest, jarInput)  
 }  
 }  
 for (directoryInput in input.*directoryInputs*) {  
 foreachClass(directoryInput)  
 }  
 **}** executor.invokeAll(tasks)  
  
 destFiles.*forEach* **{  
 it**.*filterTest*("temp")?.*forEach* **{** file **->** file.*deleteAll*()  
 **}  
 }** val timeUsage = System.currentTimeMillis() - startTimeUsage  
 Log.info("transform coast time: $timeUsage ms")  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
}

开始的时候我们先判断当前这次是不是增量编译，如果不是增量则开始遍历所有jar，如果是增量编译，会去获取当前jar的状态，如果状态是删除则先扫描jar之后把output 中的文件删除。如果状态是ADD的情况下，则扫描修改这个jar文件。最后如果是CHANGE状态，则先扫描新旧两个jar，比较获取删除的文件，然后重复ADD操作。

@Throws(IOException::class)  
private fun foreachClass(directoryInput: DirectoryInput) {  
 val dest = outputProvider!!.getContentLocation(  
 directoryInput.*name*, directoryInput.*contentTypes*,  
 directoryInput.*scopes*, Format.*DIRECTORY* )  
 destFiles.add(dest)  
 val map = directoryInput.*changedFiles* val dir = directoryInput.*file* if (isIncremental) {  
 for ((file, status) in map) {  
 val destFilePath = file.*absolutePath*.*replace*(dir.*absolutePath*, dest.*absolutePath*)  
 val destFile = File(destFilePath)  
 when (status) {  
 Status.*ADDED*, Status.*CHANGED* -> {  
 val callable = *Callable*<Void> **{** try {  
 FileUtils.touch(destFile)  
 } catch (ignored: Exception) {  
 *// Files.createParentDirs(destFile)* }  
 modifySingleFile(dir, file, destFile)  
 null  
 **}** tasks.add(callable)  
 }  
 Status.*REMOVED* -> deleteDirectory(destFile, dest)  
 else -> {  
 }  
 }  
 }  
 } else {  
 changeFile(dir, dest)  
 }  
}

这个是修改.class文件的操作 , 和修改jar包的逻辑基本一样，但是又一个区别，如果是增量编译的情况下，我们获取的对象是一个Map，而非增量编译的情况下，我们使用的是整个文件夹路径。

# 使用场景

## Apm 监控方法耗时

1. 滴滴的哆啦A梦调试组件，里面有个工具叫性能监控，他可以调试出项目内耗时的方法，之后将耗时方法打印出来。
2. 腾讯的matrix，这个项目是拿来做项目的性能监控的，其中也有一个监控方法耗时的。

简单的说就是在项目的方法头和方法尾插入了一段代码块，然后通过这个代码块去监控函数耗时。

## 动态注册

通过transform，然后在编译过程中，搜集好所有实现了特定接口的class，然后将他们插入到一个注册的类上去。

## 无痕埋点

将所有实现了View.OnClickListner接口的实现类全部进行一次修改，在onClick(View v)的方法头插入一段埋点的代码，然后内部通过反射的机制，获取到当前class的一些字段内容然后上报。

## Dex包体积优化

对于dex来说，.class文件越小自然包体积越小，这个是大家都知道的一个现象。

 shrink-r-plugin 先将R文件内的id值全部提取出来，然后搜索所有.class，当发现到R.xxx.xxx的引用直接替换为值，最后删除多余的R

 getter-setter-inline-plugin get set 函数内联

 access-inline-plugin 内联access$方法

 const-inline-plugin 常量值内联

## 类替换

项目内有时候会直接使用一些系统提供的api，但是其实内部是可能出现一些异常情况的。举个例子intent取值的时候如果类型转换出错是有可能直接崩溃的。特别是在一些路由跳转的情况下，将url params的参数反序列化之后。

这个时候也可以通过transform 对当前类进行方法调用进行一次修改，替换成我们包装过的一个异常处理过的类，这样就可以保证一些奇特的异常场景，无法处理的情况。

还有就是如果你要删除你项目内的Log输出，你的历史代码又特别多，这个时候咋办。自己定义一个lint规则搜索，删除工作量太大。你也可以用transform去改，在项目打包时检测是否存在特定的类调用，如果调用则不去写入。

## 资源文件体积压缩

[AndResGuard](https://link.juejin.cn/?target=https%3A%2F%2Fgithub.com%2Fshwenzhang%2FAndResGuard)这个项目应该有些人都听说过，是一个资源文件混淆的库，微信团队写的。也是做包体积优化的。

但是这个仓库和上面的还是有很大差异的，上面的大部分是基于Transform去写的，而这个仓库则是添加了个task 任务，去做资源文件的混淆。

# Transform和Task的关系

每一个Project在构建的时候都包含一系列的Task。比如一个Android APK的编译可能包含：Java源码编译Task、资源编译Task、JNI编译Task、lint检查Task、打包生成APK的Task、签名Task等。

AppExtension appExtension = project.getExtensions().getByType(AppExtension.class);

appExtension.registerTransform(new DoubleTabTransform(project));

当我们在编写一个含有Transform的plugin插件的时候，只是对安卓的AppExtension注册了一个Transform。

### 自定义Transform和其他系统Transform执行的顺序

任务还是会根据DAG（有向无环图）的拓扑排序将Task顺序执行。其中会包含一些系统的Transform，其顺序有可能会被插入到自定义的Transform之前，而有一些则会被放置在所有的Tranform执行之后。

拓扑排序(Topological Order)是指，将一个有向无环图(Directed Acyclic Graph简称DAG)进行排序进而得到一个有序的线性序列。

例如，一个项目包括A、B、C、D四个子部分来完成，并且A依赖于B和D，C依赖于D。现在要制定一个计划，写出A、B、C、D的执行顺序。这时，就可以利用到拓扑排序，它就是用来确定事物发生的顺序的。在拓扑排序中，如果存在一条从顶点A到顶点B的路径，那么在排序结果中B出现在A的后面。

### Transform和Task的关系

我们注册到AppExtension里面的Transform,之后所有的Transform会createTasksForVariantScope方法调用到，然后会把taskManger的addTransform方法。

然后我们看transformManager.addTransform,直接观察最后一行代码，发现我们会往taskFactory中直接注册一个TransformTask，那么回答第一个问题，一个Transform就是gradle plugin中的一个原子的Task，所以Transform==Task 。

### Transform是如何被执行的

当我们知道Transform的本质是Task之后就知道了，他会和一个普通的Task一样被执行，而且会按照特定的依赖顺序执行，而每个Task又是一个Gradle Project中的最基础的原子操作，所以Transform的执行顺序就是Task的执行顺序。

首先我们构造一个Task之后要先找到这个Task的前置任务依赖，然后我们可以把实际的Task内容执行在线程池内，充分的利用多线程的优势去构造一个项目，然后我们需要做的就是在后置的依赖任务执行之前，获取到任务的Task，之后在Task 的doFirst方法做一个await操作，这样就可以把充分利用cpu和多线程的优势，去优化一个Task任务了。

val manager = scope.transformManager

val field = manager.javaClass.getDeclaredField("transforms").apply {

isAccessible = true

}

val list = field.get(manager) as List<Transform>

list.forEach {

if (it is ShrinkResourcesTransform) {

val taskName = scope.getTaskName(getTaskNamePrefix(it))

val task = project.tasks.getByName(taskName)

task.doFirst {

val testTask = project.tasks.getByName(resGuardTaskName)

as TestTask

val taskLogger = TimeUsageHelper()

taskLogger.info("AwaitTask start ")

resProguardTask.await()

taskLogger.logTotalInfo("AwaitTask finish ")

Log.writeToFile(project)

}

task.dependsOn(testTask)

}

}

上述代码就是先获取一个系统的Transform任务，然后在任务之前await上一个任务完成。