# ASM 应用与实践

2023/01/13

# 引言

- ASM 是什么?
- ASM 有什么用?

# ASM 是什么

一个字节码操作库

# ASM 有什么用


Java
JDK
Lambda 表达式
反射的动态实现
Android
AGP
BuildConfig 类生成
shrinkResources
APM
Booster
ByteX
Matrix

# ASM 两类 API

```
// Core API
implementation "org.ow2.asm:asm:9.4"
// Tree API
implementation "org.ow2.asm:asm-tree:9.4"
```

# ASM 示例

不在于演示它怎样使用，而在于了解它有什么用

# 示例一：读取 ArrayList 类

```
private fun readArrayListByTreeApi() {
    // 1. 从类的全限定名、或字节数组、或二进制字节流中读取字节码
    val classReader = ClassReader(ArrayList::class.java.canonicalName)
    // 2. 以 ClassNode 形式表示字节码
    val classNode = ClassNode(Opcodes.ASM9)
    classReader.accept(classNode, ClassReader.SKIP_CODE)
    classNode.apply {
        println("name: $name\n")
        // 3. 读取属性
        fields.take(2).forEach {
            println("field: ${it.name} ${Modifier.toString(it.access)} ${it.desc} ${it.value}")
        }
        println()
        // 4. 读取方法
        methods.take(2).forEach {
            println("method: ${it.name} ${Modifier.toString(it.access)} ${it.desc}")
        }
    }
}
```

```
// 输出
name: java/util/ArrayList

field: serialVersionUID private static final J 8683452581122892189
field: DEFAULT_CAPACITY private static final I 10

method: <init> public (I)V
method: <init> public ()V
```

# 示例二：输出特定方法耗时

```java
public class MeasureMethodTime {

    @MeasureTime
    public void measure() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```java
public class MeasureMethodTimeTreeClass {
    public MeasureMethodTimeTreeClass() {
    }

    @MeasureTime
    public void measure() {
        long var3 = System.currentTimeMillis();

        long var5;
        try {
            Thread.sleep(2000L);
        } catch (InterruptedException var7) {
            RuntimeException var10000 = new RuntimeException(var7);
            var5 = System.currentTimeMillis();
            System.out.println(var5 - var3);
            throw var10000;
        }

        var5 = System.currentTimeMillis();
        System.out.println(var5 - var3);
    }
}
```

# 示例二：输出特定方法耗时

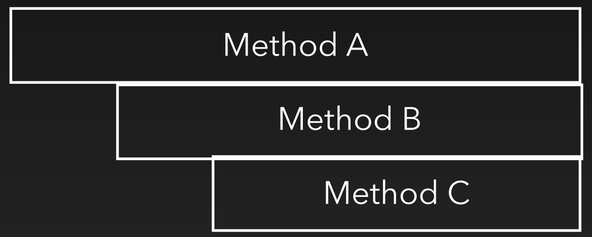
```
public measure()V
@Ltask_5/MeasureTime;() // invisible
  ------------ insert start
  TRYCATCHBLOCK L0 L1 L2 java/lang/InterruptedException
 L0
  LDC 2000
  INVOKESTATIC java/lang/Thread.sleep (J)V
 L1
  GOTO L3
 L2
  ASTORE 1
  NEW java/lang/RuntimeException
  DUP
  ALOAD 1
  INVOKESPECIAL java/lang/RuntimeException.<init> (Ljava/lang/Throwable;)V
  ------------ insert end
  ATHROW
 L3
  ------------ insert end
  RETURN
  MAXSTACK = 3
  MAXLOCALS = 2
```

```
classNode.methods.forEach { methodNode ->
    // 该方法的注解列表中包含 @MeasureTime
    if (methodNode.invisibleAnnotations?.map { it.desc }
            ?.contains(Type.getDescriptor(MeasureTime::class.java)) == true) {
        val localVariablesSize = methodNode.localVariables.size
        // 在方法的第一个指令之前插入 System.currentTimeMillis()
        val firstInsnNode = methodNode.instructions.first
        methodNode.instructions.insertBefore(firstInsnNode, InsnList().apply {
            add(MethodInsnNode(Opcodes.INVOKESTATIC, "java/lang/System", "currentTimeMillis", "()J"))
            add(VarInsnNode(Opcodes.LSTORE, localVariablesSize + 1))
        })

        // 在方法 return 指令之前插入
        methodNode.instructions.filter {
            it.opcode.isMethodReturn()
        }.forEach {
            methodNode.instructions.insertBefore(it, InsnList().apply {
                add(MethodInsnNode(Opcodes.INVOKESTATIC, "java/lang/System", "currentTimeMillis", "()J"))
                // 注意, Long 是占两个局部变量槽位的, 所以这里要较之前 +3, 而不是 +2
                add(VarInsnNode(Opcodes.LSTORE, localVariablesSize + 3))
                add(FieldInsnNode(Opcodes.GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;"))
                add(VarInsnNode(Opcodes.LLOAD, localVariablesSize + 3))
                add(VarInsnNode(Opcodes.LLOAD, localVariablesSize + 1))
                add(InsnNode(Opcodes.LSUB))
                add(MethodInsnNode(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream", "println", "(J)V"))
            })
        }
    }
}
```

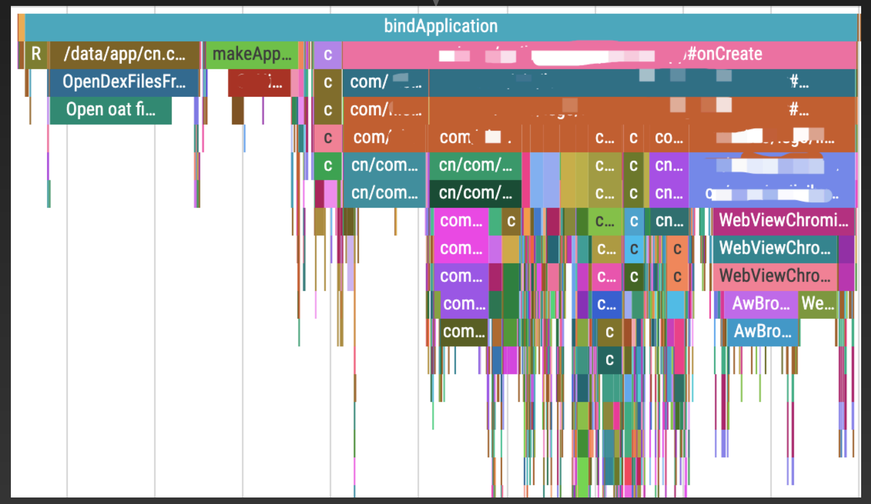
# 示例二：输出特定方法耗时

```
public inline fun measureTimeMillis(block: () -> Unit): Long {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    val start = System.currentTimeMillis()
    block()
    return System.currentTimeMillis() - start
}
```



# 示例二：输出特定方法耗时

```
private fun trace() {
    Trace.beginSection("className#methodName")
    // ...
    Trace.endSection()
}
```



# 示例三：删除日志语句

```java
public class DeleteLogInvoke {

    public String print(String name, int age) {
        System.out.println(name);
        String result = name + ": " + age;
        System.out.println(result);
        System.out.println("Delete current line.");
        System.out.println("name = " + name + ", age = " + age);
        System.out.printf("name: %s%n", name);
        System.out.println(String.format("age: %d", age));
        return result;
    }
}
```

```java
public class DeleteLogInvokeCoreClass {
    public DeleteLogInvokeCoreClass() {
    }

    public String print(String var1, int var2) {
        String var3 = var1 + ": " + var2;
        return var3;
    }
}
```

```
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;  // start line
    ALOAD 1
    ILOAD 2
    INVOKEDYNAMIC makeConcatWithConstants(Ljava/lang/String;I)Ljava/lang/String; [
      // handle kind 0x6 : INVOKESTATIC

 java/lang/invoke/StringConcatFactory.makeConcatWithConstants(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/
String;Ljava/lang/invoke/MethodType;Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/invoke/CallSite;
      // arguments:
      "name = \u0001, age = \u0001"
    ]
    INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V // end line
```

# 示例三：删除日志语句

## Why Proguard?

```
-assumenosideeffects class android.util.Log {
    public static boolean isLoggable(java.lang.String, int);
    public static int v(...);
    public static int i(...);
    public static int w(...);
    public static int d(...);
    public static int e(...);
}
```

# 示例三：删除日志语句

```
Log.i("MainActivity", "onCreate: $packageName")
```

Dex Byte Code

```
    invoke-virtual {p0}, Landroid/content/Context;->getPackageName()Ljava/lang/String;

    move-result-object p1

    const-string v0, "onCreate: "

    invoke-static {v0, p1},
kotlin.jvm.internal.Intrinsics.stringPlus(Ljava/lang/String;Ljava/lang/Object;)Ljava/lang/String;

    move-result-object p1

    const-string v0, "MainActivity"

    invoke-static {v0, p1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I
```

Delete

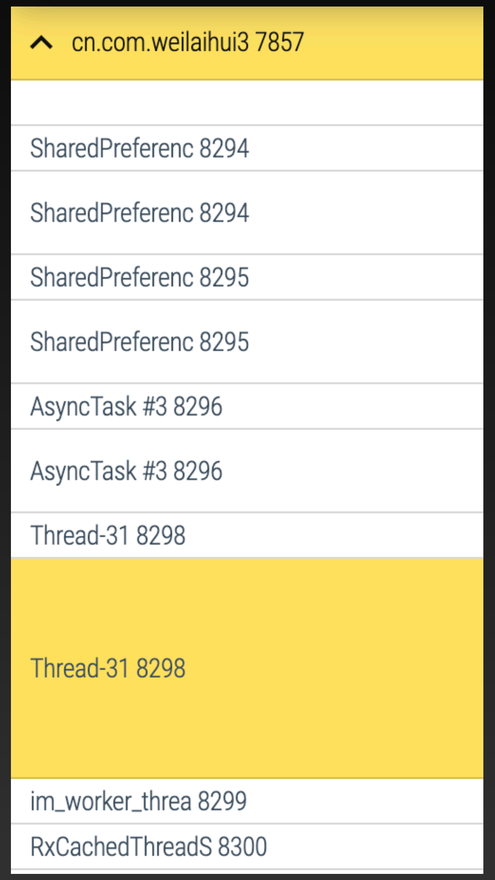
```
    invoke-virtual {p0}, Landroid/content/Context;->getPackageName()Ljava/lang/String;

    move-result-object p1

    const-string v0, "onCreate: "

    invoke-static {v0, p1}, Landroidx/constraintlayout/widget/R$id;-
>kotlin.jvm.internal.Intrinsics.stringPlus(Ljava/lang/String;Ljava/lang/Object;)Ljava/lang/String;
```

# 示例四：线程重命名




线程管理面临的挑战：

- 可能存在低优先级的子线程抢占 CPU，导致主线程 UI 响应能力降低
- 不可控的线程创建可能导致 OOM
- 线程命名默认是以 Thread-{N} 形式命名，不利于问题排查

# 示例四：线程重命名

```java
public class ThreadReName {

    public static void main(String[] args) {
        // 不带线程名称
        new Thread(new InternalRunnable()).start();

        // 带线程名称
        Thread thread0 = new Thread(new InternalRunnable(), "thread0");
        System.out.println("thread0: " + thread0.getName());
        thread0.start();

        Thread thread1 = new Thread(new InternalRunnable());
        // 设置线程名字
        thread1.setName("thread1");
        System.out.println("thread1: " + thread1.getName());
        thread1.start();
    }
}
```

```java
public class ThreadReNameTreeClass {
    public ThreadReNameTreeClass() {
    }

    public static void main(String[] var0) {
        (new ShadowThread(new InternalRunnable(), "sample/ThreadReNameTreeClass#main-Thread-0")).start();
        ShadowThread var1 = new ShadowThread(new InternalRunnable(), "thread0",
"sample/ThreadReNameTreeClass#main-Thread-1");
        System.out.println("thread0: " + var1.getName());
        var1.start();
        ShadowThread var2 = new ShadowThread(new InternalRunnable(), "sample/ThreadReNameTreeClass#main-Thread-
2");
        var2.setName(ShadowThread.makeThreadName("thread1", "sample/ThreadReNameTreeClass#main-Thread-3"));
        System.out.println("thread1: " + var2.getName());
        var2.start();
    }
}
```

# 示例四：线程重命名

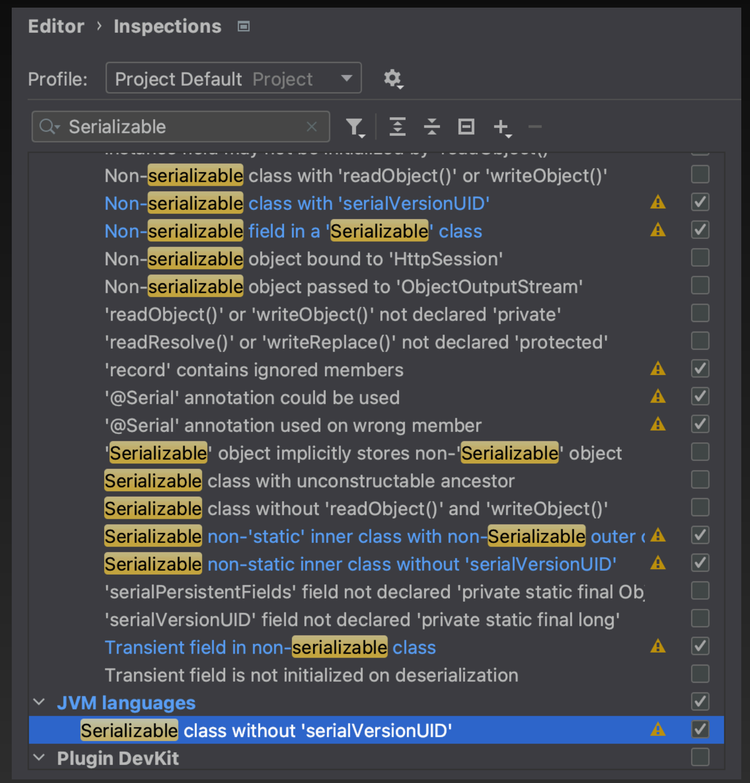
## How to fix System bug！？

```java
public class ReplaceMethodInvoke {
    public static void main(String[] args) {
        // throw NPE
        new Toast().show();
    }
}

public class Toast {
    private String msg = null;

    public void show() {
        System.out.println("Toast: " + msg + ", msg.length: " + msg.length());
    }
}
```

# 示例五：序列化检查




IDEA Inspections：

- 实现了 Serializable 的类未提供 serialVersionUID 字段
- 实现了 Serializable 的类包含了非 transient、static 的字段，这些字段并未实现 Serializable
- 未实现 Serializable 接口的类，包含了 transient、serialVersionUID 字段
- 实现了 Serializable 的非静态内部类，它的外层类并未实现 Serializable 接口

```
Attention: Non-serializable field 'itemBean1' in a Serializable class
[sample/SerializationCheckCoreClass]
Attention: This [sample/SerializationCheckCoreClass] class is serializable, but
does not define a 'serialVersionUID' field.
```

# ASM 不能做什么

不支持动态分析或者说是运行时分析

# 示例一：无用 assets 资源监测

思路：

- 收集项目中所有的 AAR 包含的 assets 资源名
- 在 Transform 阶段，检测 AssetManager#open 调用，收集所有已使用的 assets 资源名 ✗
- 两者一对比，就可得知哪些 assets 资源未被使用类

```
context.getAssets().open("fileName.json")
```

✓

```
private fun openAsset(context:Context, fileName:String) {
    context.assets.open(fileName)
}
```

✗

# 总结

- ASM 的应用范畴?
- 不在 ASM 的应用范畴?

# ASM 的应用范畴


Static Analysis
序列化检查
字段、方法引用检查
隐私合规
AOP
Systrace 插桩
代码覆盖率
Hook
线程重命名
点击防手抖
修复系统 bug

# 不在 ASM 的应用范畴

- 运行时分析

  AssetManager#open、Resources#getIdentifier 参数分析

- 目标非字节码

  Android SDK、APT

# 更多资源

- 1. https://github.com/Omooo/ASM-Task
- 2. Chapter 4. The class File Format

# ASM - TASK

Omooo / ASM-Task

master

- gradle/wrapper
- library1
- libs
- src/main/kotlin
  - common
  - task_1
    - ReadArrayListCoreApi.kt
    - ReadArrayListTreeApi.kt
  - task_10
    - ReferCheck.kt
    - ReferCheckCoreApi.kt
    - ReferCheckTreeApi.kt
  - task_2
    - GenerateMainClassCoreApi.kt
    - GenerateMainClassTreeApi.kt
    - Main.java
  - task_3
    - PrintMethodParams.java
    - PrintMethodParamsCoreApi.kt
    - PrintMethodParamsTreeApi.kt
  - task_4
    - DeleteLogInvoke.java
    - DeleteLogInvokeCoreApi.kt
    - DeleteLogInvokeTreeApi.kt
  - task_5
  - task_6
  - task_7

README.md

## ASM-Task

**目录**



**任务一：读取 ArrayList 类**

掌握最基础的 class 表现形式。

**任务二：生成 Main 类**

使用 ASM Api 生成以下 Main 类：

```
public class Main {

    private static final String TAG = "Main";

    public static void main(String[] args) {
```

# The Class File Format

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

# Q&A