一、前言

所谓的ASM，其实就是如何生成一个Class文件或者修改一个Class文件的工具，包括对Class里的成员变量或者方法进行增加或修改。相比于Javassist，ASM最大的好处就是性能方面优于Javassist，但随之带来的就是需要开发者精通 class 文件格式和 JVM 指令集。

二、用ASM生成Class文件

> **引入ASM工具库**

```
//核心库

implementation 'org.ow2.asm:asm:9.3'
//辅助库
implementation 'org.ow2.asm:asm-commons:9.3'
```

首先，先简单生成一个Class文件。运行以下代码，就可以直接生成一个Class文件。

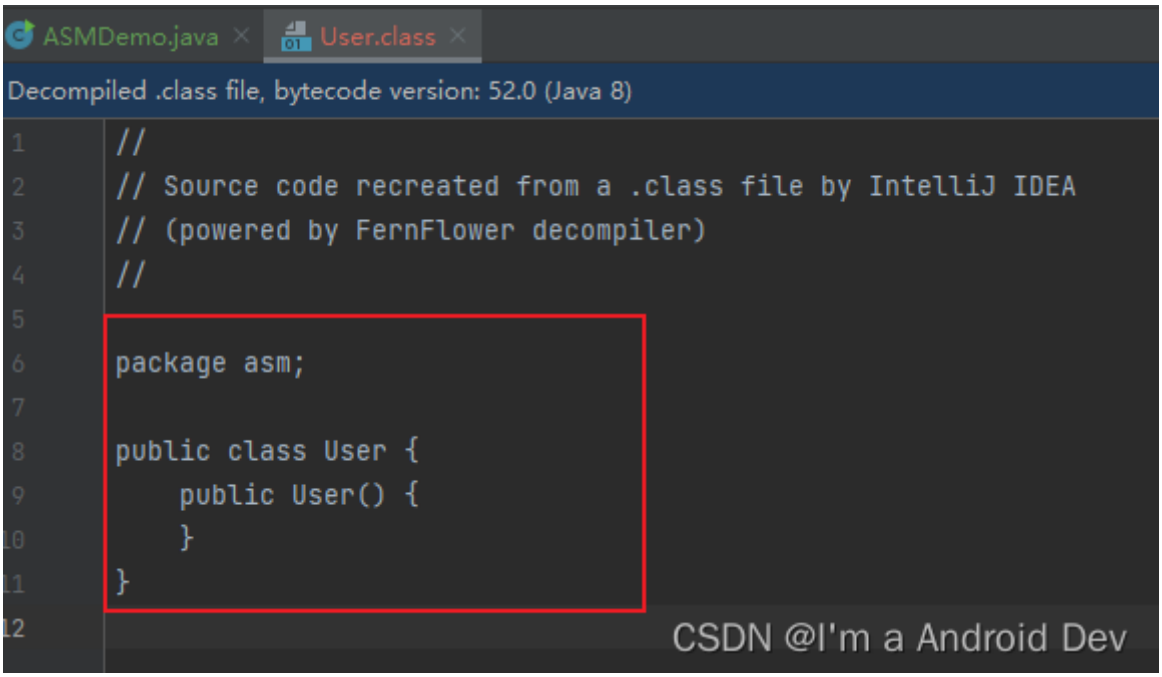
2.1 生成Class 字节码

```
    public static byte[] genClass() {
        ClassWriter classWriter = new ClassWriter(0);
        classWriter.visit(V1_8, ACC_PUBLIC | ACC_SUPER, "asm/User",
                null, "java/lang/Object", null);
        {
            MethodVisitor methodVisitor = classWriter.visitMethod(ACC_PUBLIC, "
<init>", "()V", null, null);
            methodVisitor.visitCode();
            Label label0 = new Label();
            methodVisitor.visitLabel(label0);
            methodVisitor.visitLineNumber(3, label0);
            methodVisitor.visitVarInsn(ALOAD, 0);
            methodVisitor.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "
<init>", "()V", false);
            methodVisitor.visitInsn(RETURN);
            Label label1 = new Label();
            methodVisitor.visitLabel(label1);
            methodVisitor.visitLocalVariable("this", "Lasm/User;", null, label0,
label1, 0);
            methodVisitor.visitMaxs(1, 1);
            methodVisitor.visitEnd();
        }
        classWriter.visitEnd();
        return classWriter.toByteArray();
    }
```

2.2 将生成的Class字节码输出

```
    public static void main(String[] args) throws Exception {
        //1.生成Class字节码
        byte[] genClassByte = genClass();
        try {
            //输出Class字节码文件
            FileOutputStream fos = new FileOutputStream("src/asm/User.class");
            fos.write(genClassByte);
            fos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

效果如下：



可以看到，相比与Javassist，复杂的不是一点两点。我们可以把上面的代码区分为两部分。

分别是ASM工具的使用和Java字节码。如下：

ASM工具的使用，例如ClassWriter，MethodVisitor等
Java字节码，例如: "" 代表的是初始化构造方法，"()V" 代表的是这个方法无参并且无返回参数。
接下来我们介绍ClassWriter，MethodVisitor类的用法，以及一些JVM字节码知识，高能预警！！

2.3 ClassWriter

很明显，这是一个对Class进行写入操作的类，例如对Class文件添加变量，方法。我们先看看他的构造方法。

2.3.1 构造方法

```
ClassWriter classWriter = new ClassWriter(0);
```

```
public ClassWriter(final int flags)
```

关于flags的传值如下：

1. flag == 0， 你必须自己计算栈帧和局部变量以及操作数堆栈的大小，也就是你要自己调用visitmax和visitFrame方法。
2. flag == ClassWriter. COMPUTE_MAXS，局部变量和操作数堆栈部分的大小会为你计算，还需要调用visitFrame方法设置栈帧。
3. flag == ClassWriter.COMPUTE_FRAMES，所有的内容都是自动计算的。 你不必调用visitFrame和visitmax。换句话说，COMPUTE_FRAMES包含

COMPUTE_MAXS
注意：使用ClassWriter. COMPUTE_MAXS他会使得ClassWriter的速度慢10%，ClassWriter.COMPUTE_FRAMES会慢20%。

2.3.2 visit方法

作用：用来定义类的属性

方法使用如下：

例如：生成一个User类。

```
        classWriter.visit(
                V1_8,
                ACC_PUBLIC | ACC_SUPER,
                "asm/User",
                null,
                "java/lang/Object",
                null);
```

源代码如下：

```
public final void visit(
      final int version,
      final int access,
      final String name,
      final String signature,
      final String superName,
      final String[] interfaces)
```

参数解释：

version：Java版本号，例如 V1_8 代表Java 8
access：Class访问权限，一般默认都是 ACC_PUBLIC | ACC_SUPER,部分字段解释如下

标 志 名 称	标志值	含　义
ACC_PUBLIC	0x0001	是否为 public 类型
ACC_FINAL	0x0010	是否被声明为 final，只有类可设置
ACC_SUPER	0x0020	是否允许使用 invokespecial 字节码指令的新语义，invokespecial 指令的语义在 JDK 1.0.2 发生过改变，为了区别这条指令使用哪种语义，JDK 1.0.2 之后编译出来的类的这个标志都必须为真
ACC_INTERFACE	0x0200	标识这是一个接口
ACC_ABSTRACT	0x0400	是否为 abstract 类型，对于接口或者抽象类来说，此标志值为真，其他类型值为假
ACC_SYNTHETIC	0x1000	标识这个类并非由用户代码产生的
ACC_ANNOTATION	0x2000	标识这是一个注解
ACC_ENUM	0x4000	标识这是一个枚举
ACC_MODULE	0x8000	标识这是一个模块

1. name： Class文件名，例如：asm/User，包名加类名
2. signature：类的签名，除非你是泛型类或者实现泛型接口，一般默认null。
3. superName：继承的类，很明显所有类默认继承Object。例如：java/lang/Object ，如果是继承自己写的类Animal，那就是 asm/Animal
4. interfaces：实现的接口，例如实现自己写的接口IPrint，那就是new String[]{"asm/IPrint"}

2.3.3 visitMethod方法

作用：用来定义类的方法

方法使用如下：

例如：生成User的构造方法 public User（）{}

```
classWriter.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
```

源代码如下：

```
  public final MethodVisitor visitMethod(
      final int access,
      final String name,
      final String descriptor,
      final String signature,
      final String[] exceptions)
```

参数解释：

access：方法的访问权限，也就是public，private等
name： 方法名
　　在Class中，有两个特殊方法名。和，其他的方法都是正常对应Java方法名。

<init>代表的是类的实例构造方法初始化，也就是new 一个类时，肯定会调用的方法。

<clinit>代表的类的初始化方法，不同于<init>，它不是显示调用的。因为Java虚拟机会自动调用<clinit>，并且保证在子类的<clinit>前调用父类的<clinit>。也就是说，Java虚拟机中，第一个被执行<clinit>方法的肯定是java.lang.Object。

注意：<init>和<clinit>执行的时机不一样，<clinit>的时机早于<init>，<clinit>是在类被加载阶段的初始化过程中调用<clinit>方法，而<init>方法的调用是在new一个类的实例的时候。

descriptor：方法的描述符

所谓方法的描述符，就是字节码对代码中方法的形参和返回值的一个描述。其实就是一个一一对应的模板。如下：

```
(IF)V =（表示方法的形参类型描述符)方法的返回值
```

关于形参的类型描述符如下：

Java类型	类型描述符
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object; （L + 包名+类名 + ; ）
int [ ]	[I （[+ I ）
Object [ ] [ ]	[[Ljava/lang/Object; （[+ [+ L + 包名 + 类名 + ; ）
User（自定义User类）	Lasm/User; （L + 包名 + 类名 + ; ）

方法描述符如下：

代码中方法的声明	方法描述符
void m（int i，float f)	(IF)V
int m（Object o)	(Ljava/lang/Object;)I
int[] m（int i，String s)	(ILjava/lang/String;)[I
Object m（int [ ] i)	([I)Ljava/lang/Object;

signature：方法签名，除非方法的参数、返回类型和异常使用了泛型，否则一般为 null。
exceptions：方法上的异常。这里我们没有抛出任何异常，所以为null。如果throws Exception，exceptions的值为：new String[]{"java/lang/Exception"}

2.3.4 visitField方法

作用：用来定义一个变量

方法使用如下：

例如：生成一个 private int a = 10;

```
classWriter.visitField(ACC_PRIVATE, "a", "I", null, null);
```

源代码如下：

```
  public final FieldVisitor visitField(
      final int access,
      final String name,
      final String descriptor,
      final String signature,
      final Object value)
```

参数解释：

access：变量的访问权限，，也就是public，private等
name：变量名
descriptor：变量的描述符，可以参考上面的Java类型对应的描述符
signature：变量的签名，如果没有使用泛型则为null
value：变量的初始值。这个字段仅作用于被final修饰的字段，或者接口中声明的变量。其他默认为null，变量的赋值是通过MethodVisitor 的 visitFieldInsn方法。

2.3.5 visitEnd方法

作用：用来通知Class已经使用完。

2.3.6 toByteArray方法

作用：返回一个字节数组

2.4 MethodVisitor

这是一个用来生成方法的类。

2.4.1 visitCode方法

作用：表示开始生成字节码

通常第一个调用，固定格式

2.4.2 visitLabel方法

作用：设置Label，Label的作用相当于表示方法在字节码中的位置

每一个方法都需要一个Label，用来保证方法调用顺序。

2.4.3 visitLineNumber方法

作用：定义源代码中的行号与对应的指令

源代码如下：

```
public void visitLineNumber(final int line, final Label start)
```

- line：源代码中对应的行号
- start：行号对应的字节码指令

2.4.4 visitVarInsn方法

作用：用来对变量进行加载和存储的指令操作

源代码如下：

```
public void visitVarInsn(final int opcode, final int varIndex)
```

参数解释:

- opcode:对应的变量字节码指令

例如：获取一个int数值类型的指令对应 iload 0

字节码	助记符	指令含义
0x15	iload	将指定的 int 型本地变量推送至栈顶
0x16	lload	将指定的 long 型本地变量推送至栈顶
0x17	fload	将指定的 float 型本地变量推送至栈顶
0x18	dload	将指定的 double 型本地变量推送至栈顶
0x19	aload	将指定的引用类型本地变量推送至栈顶
0x1a	iload_0	将第一个 int 型本地变量推送至栈顶
0x1b	iload_1	将第二个 int 型本地变量推送至栈顶

有获取就肯定会有存储

字节码	助记符	指令含义
0x38	fstore	将栈顶 float 型数值存入指定本地变量
0x39	dstore	将栈顶 double 型数值存入指定本地变量
0x3a	astore	将栈顶引用型数值存入指定本地变量
0x3b	istore_0	将栈顶 int 型数值存入第一个本地变量
0x3c	istore_1	将栈顶 int 型数值存入第二个本地变量

- varIndex：变量对应在局部变量表的下标

例如下列代码：

```
        int a = 1;
        int b = 2;
        int d = a + b;
```

上面代码对应的字节码指令就是

```
    L5
     LINENUMBER 13 L5
     ICONST_1
     ISTORE 1
    L6
     LINENUMBER 14 L6
     ICONST_2
     ISTORE 2
    L7
     LINENUMBER 15 L7
     ILOAD 1
     ILOAD 2
     IADD
     ISTORE 3
```

解释下对应的字节码含义：

1. 将1变量加载到操作数栈，对应的指令就是ICONST_1
2. 将栈顶的值保存到局部变量表第一个位置，对应的指令就是ISTORE_1
3. 将2变量加载到操作数栈，对应的指令就是ICONST_2
4. 将栈顶的值保存到局部变量表第二个位置，对应的指令就是ISTORE_2
5. 取出局部变量表第一个元素到操作数栈（也就是变量a）,对应的指令就是 ILOAD_1
6. 取出局部变量表第二个元素到操作数栈（也就是变量b），对应的指令就是 ILOAD_2
7. 此时操作数栈的栈顶就有a和b两个元素，执行指令IADD，就会把栈顶的两个元素相加并将结果压入栈顶
8. 将栈顶的值保存到局部变量表第三个位置

2.4.5 visitMethodInsn方法

作用：用来对一个方法执行指令操作

他可以执行的指令如下：

0xb6	invokevirtual	调用实例方法
0xb7	invokespecial	调用超类构造方法，实例初始化方法，私有方法
0xb8	invokestatic	调用静态方法
0xb9	invokeinterface	调用接口方法

2.4.6 visitInsn方法

作用：用来执行对操作数栈的指令

可以执行的指令如下：

NOP, ACONST_NULL, ICONST_M1, ICONST_0, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5, LCONST_0, LCONST_1, FCONST_0, FCONST_1, FCONST_2, DCONST_0, DCONST_1, IALOAD, LALOAD, FALOAD, DALOAD, AALOAD, BALOAD, CALOAD, SALOAD, IASTORE, LASTORE, FASTORE, DASTORE, AASTORE, BASTORE, CASTORE, SASTORE, POP, POP2, DUP, DUP_X1, DUP_X2, DUP2, DUP2_X1, DUP2_X2, SWAP, IADD, LADD, FADD, DADD, ISUB, LSUB, FSUB, DSUB, IMUL, LMUL, FMUL, DMUL, IDIV, LDIV, FDIV, DDIV, IREM, LREM, FREM, DREM, INEG, LNEG, FNEG, DNEG, ISHL, LSHL, ISHR, LSHR, IUSHR, LUSHR, IAND, LAND, IOR, LOR, IXOR, LXOR, I2L, I2F, I2D, L2I, L2F, L2D, F2I, F2L, F2D, D2I, D2L, D2F, I2B, I2C, I2S, LCMP, FCMPL, FCMPG, DCMPL, DCMPG, IRETURN, LRETURN, FRETURN, DRETURN, ARETURN, RETURN, ARRAYLENGTH, ATHROW, MONITORENTER, or MONITOREXIT.

部分指令说明如下：

0x5f	swap	将栈最顶端的两个数值互换（数值不能是 long 或 double 类型）
0x60	iadd	将栈顶两 int 型数值相加并将结果压入栈顶
0x61	ladd	将栈顶两 long 型数值相加并将结果压入栈顶
0x62	fadd	将栈顶两 float 型数值相加并将结果压入栈顶
0x63	dadd	将栈顶两 double 型数值相加并将结果压入栈顶
0x64	isub	将栈顶两 int 型数值相减并将结果压入栈顶
0x65	lsub	将栈顶两 long 型数值相减并将结果压入栈顶
0x66	fsub	将栈顶两 float 型数值相减并将结果压入栈顶
0x67	dsub	将栈顶两 double 型数值相减并将结果压入栈顶
0x68	imul	将栈顶两 int 型数值相乘并将结果压入栈顶
0x69	lmul	将栈顶两 long 型数值相乘并将结果压入栈顶
0x6a	fmul	将栈顶两 float 型数值相乘并将结果压入栈顶
0x6b	dmul	将栈顶两 double 型数值相乘并将结果压入栈顶
0x6c	idiv	将栈顶两 int 型数值相除并将结果压入栈顶
0x6d	ldiv	将栈顶两 long 型数值相除并将结果压入栈顶
0x6e	fdiv	将栈顶两 float 型数值相除并将结果压入栈顶
0x6f	ddiv	将栈顶两 double 型数值相除并将结果压入栈顶

2.4.7 visitLocalVariable方法

作用：对局部变量设置

源代码如下：

```
public void visitLocalVariable(
    final String name,
    final String descriptor,
    final String signature,
    final Label start,
    final Label end,
    final int index)
```

参数解释：

name：局部变量名
descriptor：局部变量名的类型描述符
signature：局部变量名的签名，如果没有使用到泛型，则为null
start：第一个指令对应于这个局部变量的作用域(包括)。
end：最后一条指令对应于这个局部变量的作用域(排他)。
index：局部变量名的下标，也就是局部变量名的行号顺序（从1开始）。
例如代码：

```
public void test(){
    int a = 1;
    int b = 2;
    int d = a + b;
}
```

对应的使用方法如下：

```
            methodVisitor.visitLocalVariable("this", "Lasm/User;",
                    "Lasm/User<TT;>;", label0, label4, 0);
            methodVisitor.visitLocalVariable("a", "I", null, label1, label4, 1);
            methodVisitor.visitLocalVariable("b", "I", null, label2, label4, 2);
            methodVisitor.visitLocalVariable("d", "I", null, label3, label4, 3);
```

注意：每个方法都会默认有一个this的引用

2.4.8 visitMaxs方法

作用：设置这个本地方法最大操作数栈和最大本地变量表

源代码如下：

```
  public void visitMaxs(final int maxStack, final int maxLocals)
```

例如代码：

```
    public void test(){
        int a = 1;
        int b = 2;
        int d = a + b;
    }
```

对应的使用方法如下：

```
methodVisitor.visitMaxs(2, 4);
```

其中

- maxStack == 2，分别是，**ICONST_1， IAdd**操作

注意：maxStack == 2，不是代表只有两个对操作数栈的指令，而是操作数栈容量大小为2，可以满足上面代码，例如下面代码，操作数栈大小为2也可以满足。

```
    public void test(){
        int a = 1;
        int b = a + 2;
        int c = b * 2;
        int d = b * 2;
        int e = b * 2;
        int f = b * 2;
    }
```

再来一个例子， 它的操作数栈大小必须为3。

分别是，**DUP， INVOKESPECIAL，ASTORE**操作

```
public void test(){
    Object user = new Object();
}
```

- maxLocals == 4，分别是局部变量this,a,b,d。

这些值的作用其实是用来决定操作数栈和本地变量表的大小，内存优化小知识😀

注意：visitMaxs方法的调用必须在所有的MethodVisitor指令结束后调用。

2.4.9 visitEnd方法

作用：通知MethodVisitor生成方法结束，表示结束生成字节码。

通常是作为MethodVisitor最后一个调用，固定格式。与visitCode，一个最前一个最后。

三、如何使用生成的代码

以下代码可以直接复制，简单修改就可以测试自己的生成的Class文件。

```
public class ASMDemo {

    public static void main(String[] args) throws Exception {
        //1.生成Class字节码
        byte[] genClassByte = genClass();
        try {
            //输出Class字节码文件
            FileOutputStream fos = new FileOutputStream("src/asm/User.class");
            fos.write(genClassByte);
            fos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        testClass("asm.User",genClassByte);
    }

    public static byte[] genClass() {
        ClassWriter classWriter = new ClassWriter(0);
        classWriter.visit(
                V1_8,
                ACC_PUBLIC | ACC_SUPER,
                "asm/User",
                null,
```

```
                "java/lang/Object",
                null);
        {
            MethodVisitor methodVisitor = classWriter.visitMethod(
                    ACC_PUBLIC,
                    "<init>",
                    "()V",
                    null,
                    null);
            methodVisitor.visitCode();
            Label label0 = new Label();
            methodVisitor.visitLabel(label0);
            methodVisitor.visitLineNumber(3, label0);
            methodVisitor.visitVarInsn(ALOAD, 0);
            methodVisitor.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "
<init>", "()V", false);
            methodVisitor.visitInsn(RETURN);
            Label label1 = new Label();
            methodVisitor.visitLabel(label1);
            methodVisitor.visitLocalVariable("this", "Lasm/User;", null, label0,
label1, 0);
            methodVisitor.visitMaxs(1, 1);
            methodVisitor.visitEnd();
        }
        classWriter.visitEnd();
        return classWriter.toByteArray();
    }


    /**
     * 测试运行Class
     *
     * @param name Class完整包名路径
     * @param b    Class字节码
     * @throws Exception
     */
    public static void testClass(String name, byte[] b) throws Exception {
        MyClassLoader myClassLoader = new MyClassLoader();
        Class clazz = myClassLoader.defineClass(name, b);
        Object obj = clazz.newInstance();

        //仅为了测试
        for (java.lang.reflect.Method method : clazz.getMethods()) {
            System.out.println("Method Name:" + method.getName());
            if (method.getName().equals("main")) {
                method.invoke(obj, new String[1]);
                break;
            }
            if (method.getName().equals("hashCode")) {
                Integer code = (Integer) method.invoke(obj, null);
                System.out.println("hashCode code:"+code);
            }
        }
    }

    public static class MyClassLoader extends ClassLoader{
        public Class defineClass(String name, byte[] b) {
            return defineClass(name, b, 0, b.length);
```

```
        }
    }
}
```

四、如何快速方便生成Class文件

看到这里，相信你会有一个疑惑？如果不懂这些字节码那要如何编写，不用担心，有一个ASM Bytecode Viewer可以帮助我们解决这个问题。只要完成下面几步，你也能快速生成一个Class文件。

4.1 下载 ASM Bytecode Viewer 插件

这个就不过多描述了，在Plugins中自己搜索吧

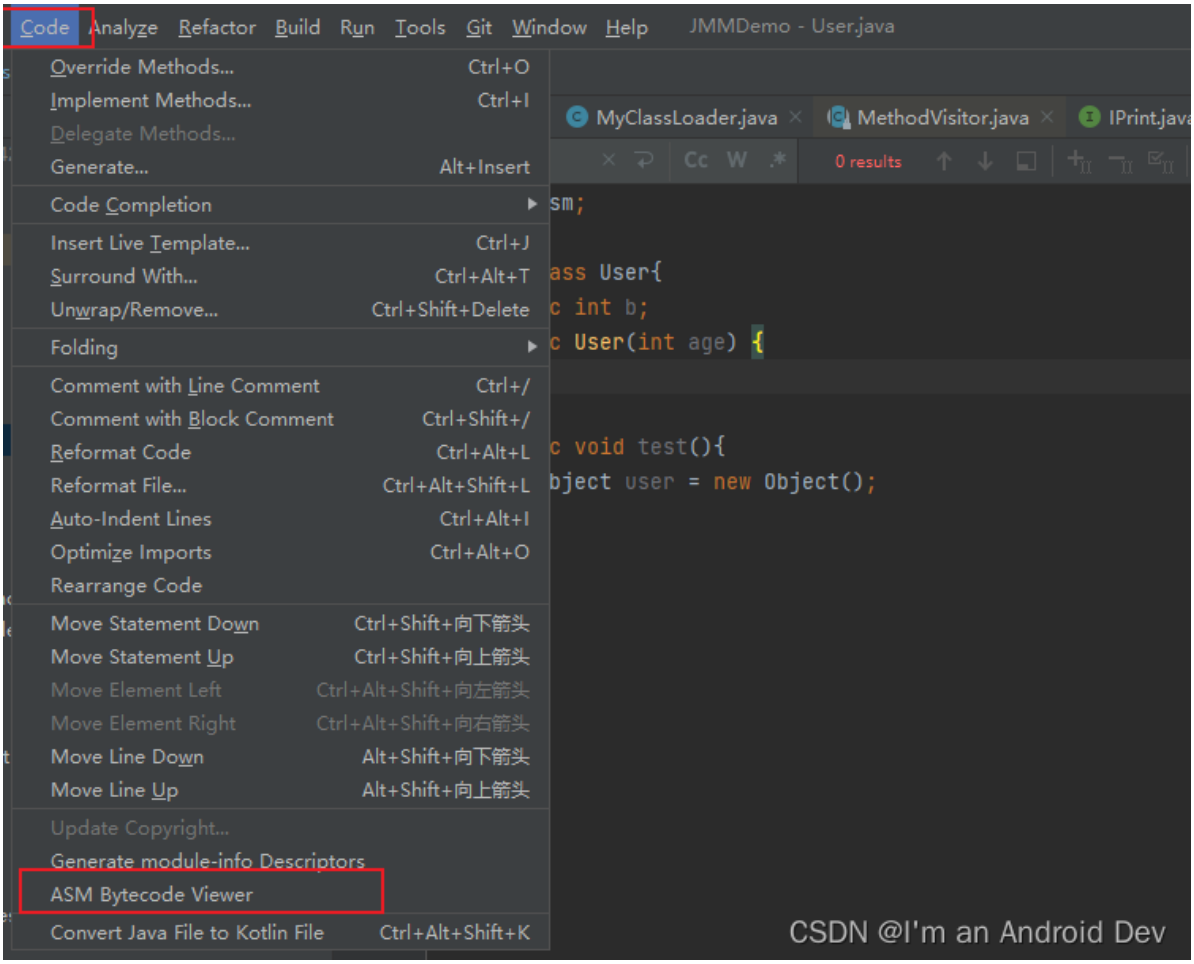
4.2 准备好你要写的Java文件

例如你想要生成如下代码：

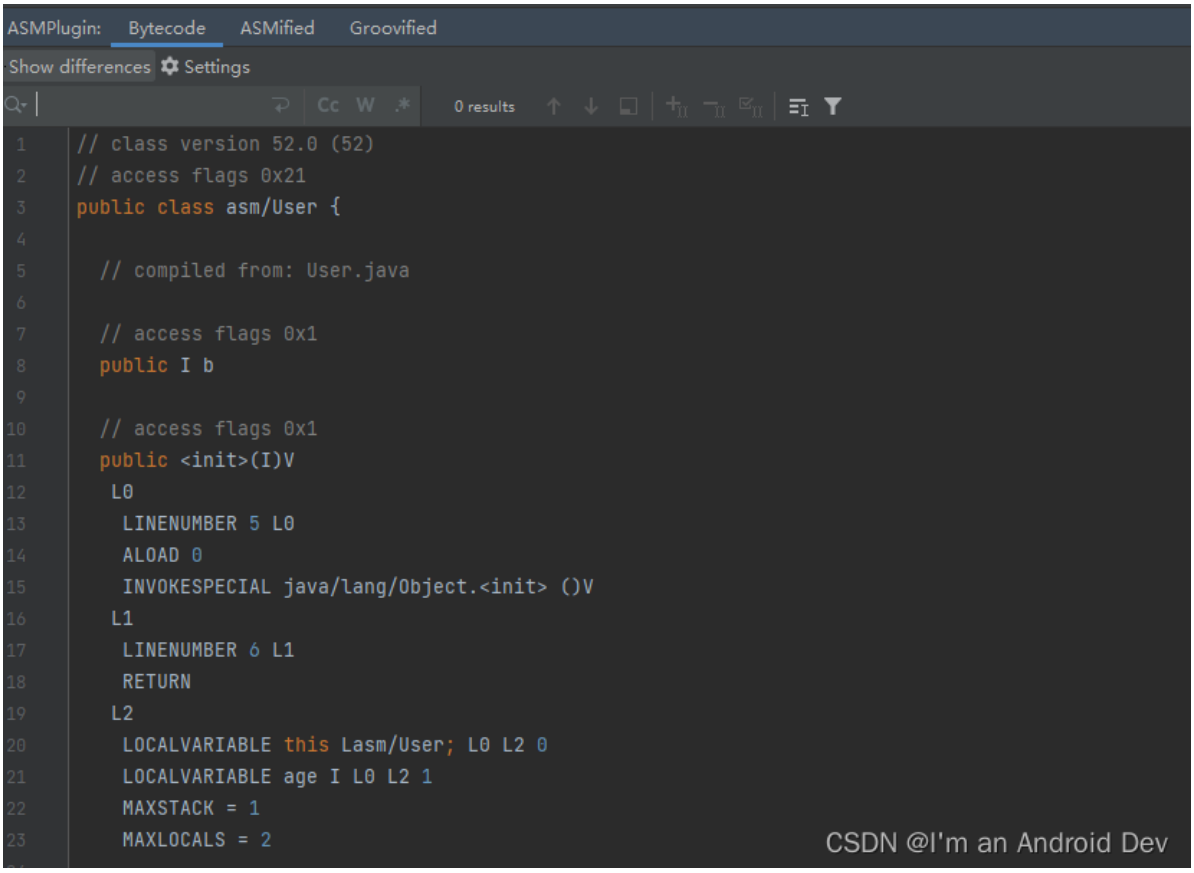
```
public class User{
    public int b;
    public User(int age) {
    }

    public void test(){
        Object user = new Object();
    }
}
```

接下来就是使用 **ASM Bytecode Viewer，步骤如下**



就可以看到如下界面，下面是我们Java代码真正的字节码内容，但是这不是我们需要的。继续往下看



看到下面，这才是我们需要的。注意是**ASMified，然后直接复制dump方法，调用就能生成一个Class的字节码数组，然后后面就是把字节码数组保存成本地，代码上面有。**