如果说 javaagent 是操作字节码的入口，那么 ASM 可以看做是 Java 字节码的手术刀。
在premain函数中，我们会拿到一个class文件的二进制数组，我们可以选择自己解析这个class 文件，在符合 Java 字节码规范的前提下进行字节码改造。如果你写过 class 文件的解析程序，就会发现这个过程极其繁琐，更别说进行增加方法等操作了。

0x01 什么是 ASM

ASM 是一个 Java 字节码操控框架。它能被用来动态生成类或者增强既有类的功能。ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。

它有以下优点

- 架构设计精巧，使用方便。
- 更新速度快，支持最新的 Java 版本
- 速度非常快，在动态代理 class 的生成和 class 的转换时，尽可能确保运行中的应用不会被 ASM 拖慢
- 非常可靠、久经考验，已经有很多著名的开源框架都在使用，例如 cglib,、mybatis、fastjson

0x02 ASM 核心类介绍

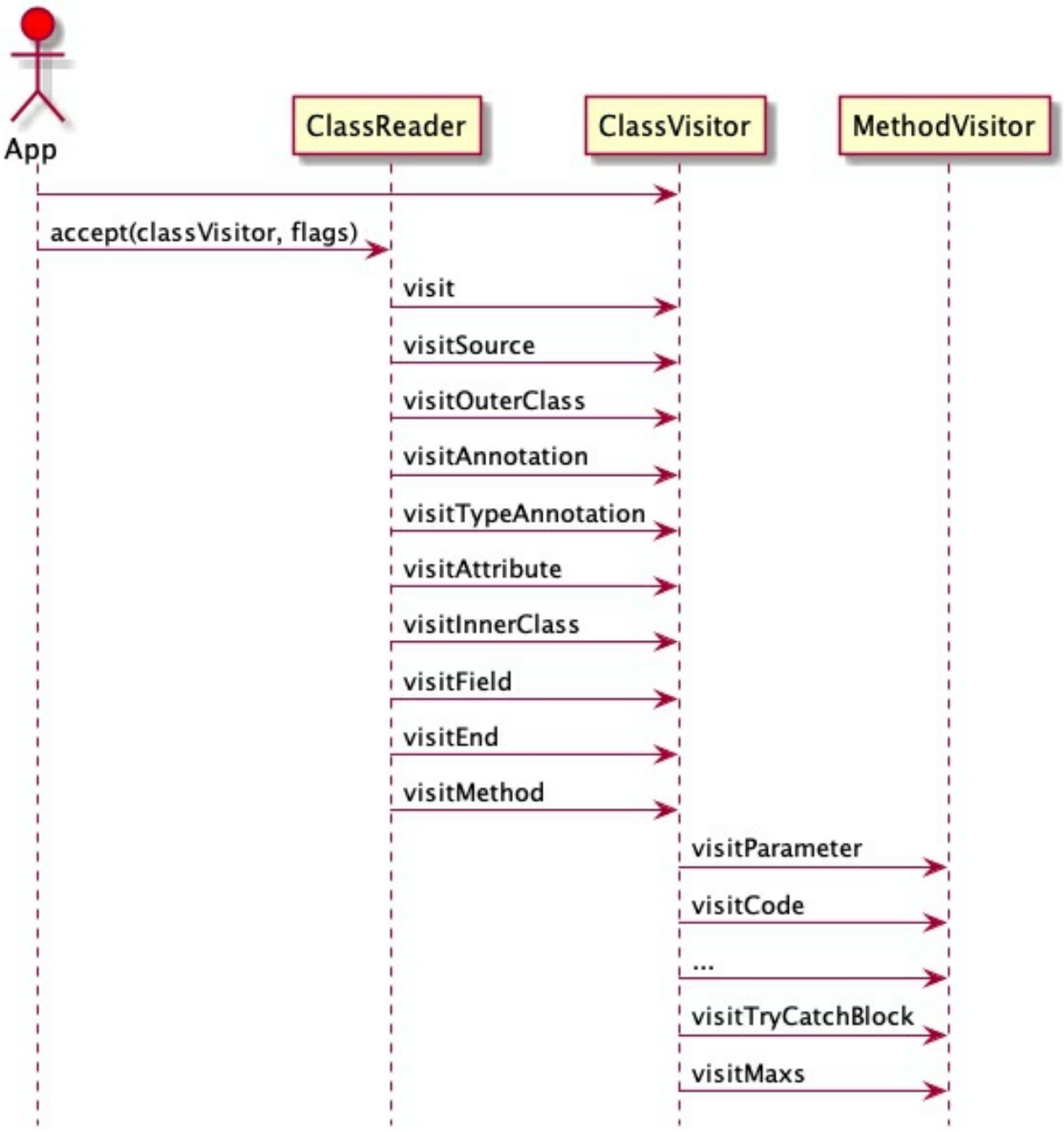
ASM 库是设计模式中访问者模式的典型应用，三大核心类 ClassReader、ClassVisitor、ClassWriter 介绍如下

ClassReader

它是字节码读取和分析引擎，帮我们做了最苦最累的解析二进制的 class 文件字节码的活。采用类似于 SAX 的事件读取机制，每当有事件发生时，触发相应的 ClassVisitor、MethodVisitor 等做相应的处理。

ClassVisitor

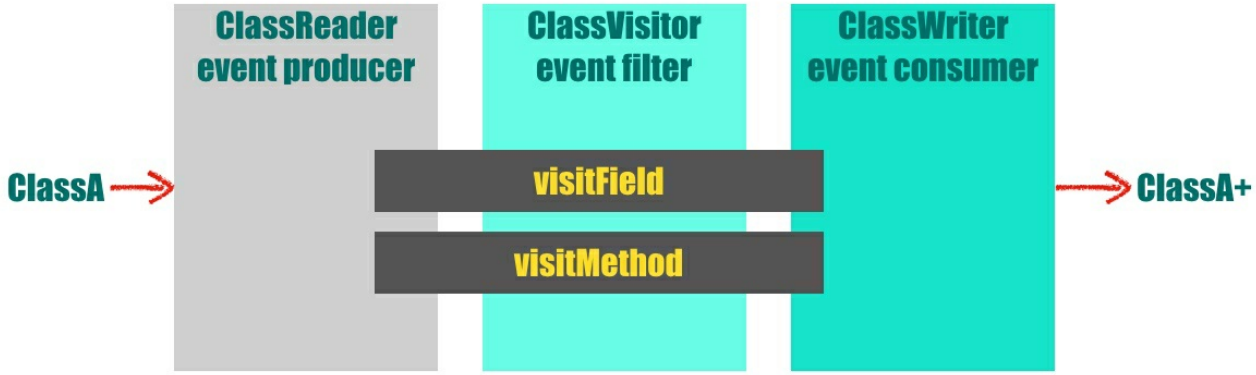
它是一个抽象类，ClassReader 对象创建之后，调用 ClassReader.accept() 方法，传入一个 ClassVisitor 对象。
ClassVisitor 在解析字节码的过程中遇到不同的节点时会调用不同的 visit() 方法，比如 visitSource, visitOuterClass, visitAnnotation, visitAttribute, visitInnerClass, visitField, visitMethod 和 visitEnd方法。
在上述 visit 的过程中还会产生一些子过程，比如 visitAnnotation 会触发 AnnotationVisitor 的调用、visitMethod 会触发 MethodVisitor 的调用。
正是在这些 visit 的过程中，我们得以有机会去修改各个子节点的字节码。
整个过程时序图如下：



ClassWriter

这个类是 ClassVisitor 抽象类的一个实现类，其之前的每个 ClassVisitor 都可能对原始的字节码做修改，ClassWriter 的 toByteArray 方法则把最终修改的字节码以 byte 数组的形式返回

这三个核心类的关系如下图



0x03 用 ASM 实现简单的调用链跟踪

同样，我们来看一个最简单的 demo，读取一个 class 文件，并对指定的方法进行注入，在方法执行前和执行后分别加一句打印
原始的 main 函数如下，step1() 和 step2() 函数是我们要注入的函数

```
public class Test01 {
    public static void main(String[] args) {
        System.out.println("in test01 main");
        new Test01().process();
    }
    public void process() {
        // 注入打印 "Call step1", 也即
System.out.println("Call " + methodName);
        step1();
        // 注入打印 "Return step1",也即
System.out.println("Return " + methodName);

        // 注入打印 "Call step2"
        step2();
        // 注入打印 "Return step2"
    }

    public void step1() {
        System.out.println("in step1");
    }

    public void step2() {
        System.out.println("in step2");
    }
}

//  执行 javac 把源文件编译成 class 文件
javac Test01
```

下面这段代码是把上面的Test01类文件改写并存储到一个新的文件中

```
FileInputStream in = new
FileInputStream("/path/to/Test01.class");
ClassReader cr = new ClassReader(in);
ClassWriter cw = new ClassWriter(cr,
ClassWriter.COMPUTE_FRAMES);
ClassVisitor cv = new TraceClassVisitor(cw);
cr.accept(cv, ClassReader.SKIP_FRAMES |
ClassReader.SKIP_DEBUG);
byte[] bytes =  cw.toByteArray();
// 把改写以后的类文件字节数组写入到新的文件中
FileUtils.writeByteArrayToFile(new
File("/new/path/to/Test01.class"), bytes, false);
```

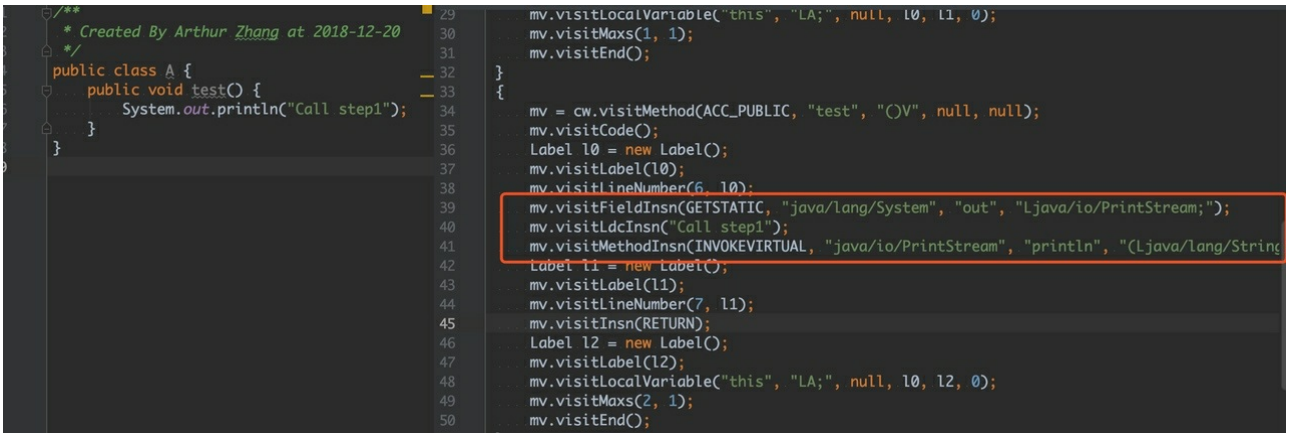
核心的改写类是TraceClassVisitor。我们只需要覆盖visitMethod，这个方法的返回值是一个MethodVisitor,这个对象会被用来处理方法体，可以插入额外的指令来完成我们打印调用链的功能。
我们来看一下核心的注入行System.out.println("Call step1");对应的字节码是什么

```
0: getstatic     #2 // Field
java/lang/System.out:Ljava/io/PrintStream;
3: ldc           #3 // String Call step1
5: invokevirtual #4 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
```

翻译成 ASM 的代码就是

```
mv.visitFieldInsn(Opcodes.GETSTATIC,
"java/lang/System", "out",
"Ljava/io/PrintStream;");
mv.visitLdcInsn("Call " + name);
mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
"java/io/PrintStream", "println", "
(Ljava/lang/String;)V", false);
```

当然这个过程，Intellij 有插件（[ASM Bytecode Outline (https://plugins.jetbrains.com/plugin/5918-asm-bytecode-outline)](https://plugins.jetbrains.com/plugin/5918-asm-bytecode-outline)）可以直接生成，不用自己去手写。



完整的 TraceClassVisitor 代码如下

```
public class TraceClassVisitor extends
ClassVisitor {
    public TraceClassVisitor(ClassVisitor cv) {
        super(ASM5, cv);
    }

    @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 TraceMethodVisitor(mv);
    }

    public static class TraceMethodVisitor
extends MethodVisitor {

        public TraceMethodVisitor(MethodVisitor
mv) {
            super(ASM5, mv);
        }

        @Override
        public void visitMethodInsn(int opcode,
String owner, String name, String desc, boolean
itf) {
            if (!name.startsWith("step")) {
                mv.visitMethodInsn(opcode, owner,
name, desc, itf);
                return;
            }
            // 增加 System.out.println("Call " +
name);
            mv.visitFieldInsn(Opcodes.GETSTATIC,
"java/lang/System", "out",
"Ljava/io/PrintStream;");
            mv.visitLdcInsn("Call " + name);

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

            // 调用原始的 call
```

```
            mv.visitMethodInsn(opcode, owner,
name, desc, itf);

            // 增加 System.out.println("Return " +
name);
            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);
        }
    }
}
```

执行一下 main 函数，生成改写的 Test01.class，然后执行

```
in test01 main
Call step1
in step1
Return step1
Call step2
in step2
Return step2
```

生成的 class 文件用反编译工具（jd-gui）如下

```
public class Test01
{
  public static void main(String[] paramArrayOfString)
  {
    System.out.println("in test01 main");
    new Test01().process();
  }

  public void process()
  {
    System.out.println("Call step1");
    step1();
    System.out.println("Return step1");
    System.out.println("Call step2");
    step2();
    System.out.println("Return step2");
  }

  public void step1()
  {
    System.out.println("in step1");
  }

  public void step2()
  {
    System.out.println("in step2");
  }
}
```

0x04 小结

这篇文章我们主要讲解了 ASM 字节码操作框架，一起来回顾一下要点：

- 第一，ASM 是一个久经考验的工业级字节码操作框架。
- 第二，ASM 的三个核心类 ClassReader、ClassVisitor、ClassWriter。ClassReader 对象创建之后，调用 ClassReader.accept() 方法，传入一个 ClassVisitor 对象。ClassVisitor 在解析字节码的过程中遇到不同的节点时会调用

不同的 visit() 方法。ClassWriter 负责把最终修改的字节码以 byte 数组的形式返回

- 第三，介绍完原理，用 ASM 实现了一个简单的调用链跟踪。

0x05 思考

给你留一道作业题：除了文中介绍的 ASM 的字节码应用，你知道还有哪些库或者框架使用了 ASM 吗？ASM 在其中承担的作用是什么？

欢迎你在留言区留言，和我一起讨论。