ASM 与 Presto 动态代码生成简介

代码生成是很多计算引擎中常用的执行优化技术，比如我们熟悉的 Apache Spark 和 Presto 在表达式等地方就使用到代码生成技术。这两个计算引擎虽然都用到了代码生成技术，但是实现方式完全不一样。在 Spark 中，代码生成其实就是在 SQL 运行的时候根据相关算子动态拼接 Java 代码，然后使用 Janino 来动态编译生成相关的 Java 字节码并加载到相关 classLoader 中，这个动态编译会带来一定的使用成本，不过这个对于 Spark 来说，这个开销是完全可以承受住的，更多关于 Apache Spark 的代码生成技术可以参见 《一条 SQL 在 Apache Spark 之旅（下）》。

而 Presto 是定位于 OLAP 的，需要快速的把执行结果返回给客户端，所以每条 SQL 的执行时间比较短（比如秒级别），而如果采用 Spark 的方式，那么代码编译的时间可能会影响到 Presto 的整个查询时间；所以 Presto 使用 ASM 直接生成 Java 字节码的方式来达到代码生成的目的。

Java 代码和 Java 字节码

Java 代码其实就是我们平时在 IDE 中开发的一堆 java 文件，这些 Java 文件是需要经过编译成 class 文件才能被 Java 虚拟机执行，而这些编译后的 class 文件其实就是这里说的 Java 字节码。Java 代码和 Java 字节码还有以下的区别：

- 编译后的一个 Java 字节码文件只描述一个类，而一个 Java 文件里面可以包含多个类。例如，具有一个内部类的类源文件被编译为两个 Java 字节码文件：一个用于主类，另一个用于内部类。主类文件包含对其内部类的引用，而在方法内部定义的内部类包含对其外围方法的引用。

- 编译后的 Java 字节码文件不包含注释，但可以包含类、字段、方法和代码属性。
- 编译后的 Java 字节码文件不包含 package 和 import 部分，因此所有类名必须是带包名的。比如 String 编译成字节码后需要用 com.lang.String 表示。

Java 字节码（Java bytecode）是 Java 虚拟机执行的一种指令格式，每个使用 javac 编译后的 class 文件是遵循 Java Virtual Machine 相关规范的，具体可以参见 The class File Format 了解详情。

另外，Java 字节码的计算模型是面向堆栈结构计算机的，其和汇编有点类似，比如如果我们实现两个数值相加的话，汇编实现如下：

```
mov eax, byte [ebp-4]
mov edx, byte [ebp-8]
add eax, edx
mov ecx, eax
```

而如果用 Java 实现的话可以如下：

```
public int add(int a, int b) {
   return a + b;
}
```

使用 javac 编译成 Java 字节码后我们使用 javap 查看生成的字节码如下：

```
public int add(int, int);
   descriptor: (II)I
   flags: ACC_PUBLIC
   Code:
     stack=2, locals=3, args_size=3
        0: iload_1
        1: iload_2
        2: iadd
        3: ireturn
     LineNumberTable:
       line 9: 0
     LocalVariableTable:
       Start  Length  Slot  Name   Signature
           0       4     0  this   Lcom/iteblog/Test;
           0       4     1     a   I
```

```
0    4    2    b  I
```

其中比较重要的是第6到第9行。 iload_1、iload_2、iadd 和 ireturn 都是 Java 虚拟机支持的指令集（Instruction Set）。iload_1 是将 #1 个 int 型本地变量推送至栈顶；iload_2 是将 #2 个 int 型本地变量推送至栈顶；iadd 是将栈顶的两个 int 值相加，并把结果压入栈顶；ireturn 是返回当前的 int 值。

ASM 简介

ASM 是一个通用的 Java 字节码操作和分析框架。它可以用来修改现有的类，或者直接以二进制形式动态生成类。ASM 提供了一些常见的字节码转换和分析算法，可以基于它构建定制的复杂转换和代码分析工具。ASM 的设计非常关注性能，因此它的设计和实现尽可能的小和快，它非常适合在动态系统中使用。

ASM API 提供了两种方式来操作 Java 字节码：基于事件（event-based）和基于树节点（tree-based）。

Event-based API

这个 API 很大程度上是基于 Visitor 模式，类似于处理 XML 文档的 SAX 解析模型。它的核心由以下几个部分组成：

- ClassReader ：可以使用它来读取 java class 文件，并将读出来的字节码存放到字节数组中，它的 accept 方法接受一个 ClassVisitor 实现类，并按照顺序调用 ClassVisitor 中的方法；
- ClassVisitor ：可以对 ClassReader 读取的 java class 文件进行转换（transform），这个过程会访问类的成员信息；包括标记在类上的注解、类的构造方法、类的字段、类的方法、静态代码块等；
- ClassWriter ： ClassWriter 是一个 ClassVisitor 的子类，和 ClassReader 类正好对应，其可以将 Java 字节码输出到 class 文件。

在 ClassVisitor 中，我们拥有所有访问器方法，我们将使用这些方法访问给定 Java 类的不同组件(字段、方法等)。我们通过提供 ClassVisitor 的子类来实现给定类中的任何更改。由于需要确保输出的 class 文件准守 Java 虚拟机规范，这个类需要一个严格的顺序来调用它的方法以生成正确的输出。基于事件的 API 中的 ClassVisitor 方法按以下顺序调用:

```
visit
visitSource?
visitOuterClass?
( visitAnnotation | visitAttribute )*
( visitInnerClass | visitField | visitMethod )*
```

visitEnd

也就是说必须先访问 visit 方法，接着是 visitSource（最多只有一个），接着是 visitOuterClass（最多只有一个），接着是 visitAnnotation 或者 visitAttribute，接着是 visitInnerClass、visitField 或 visitMethod，最后必须以 visitEnd 结尾。

Tree-based API

这个 API 是一个更加面向对象的 API，类似于处理 XML 文档的 JAXB 模型。

Event-based API 占用更少的系统资源。从内存的角度看，Tree-based API 由于要把字节码抽象成 tree，在内存中会占用跟多的空间；不过 Event-based API 比 Tree-based API 更难用，每次只能操作一个指令，需要非常了解字节码相关规范，写起来要小心翼翼。

使用 ASM 进行代码生成

因为 Presto 里面使用的是 Event-based API ，所以下面只介绍使用 Event-based API 来进行代码生成。在介绍使用代码生成之前，有必要简单介绍一下 Java 类和编译后的字节码的一些区别。

Java 类型与 class 文件内部类型对应关系

通常我们在编写 Java 代码时定义变量类型的时候会使用 int、long、String 这些来表示，但是在 Java 字节码里面可不是这么表示的。在 JVM 中对每一种类型都有与之相对应的类型 描述符（Type descriptor），对应关系如下：

- 基本类型：
  - 'V' - void
  - 'Z' - boolean
  - 'C' - char
  - 'B' - byte
  - 'S' - short
  - 'I' - int
  - 'F' - float
  - 'J' - long
  - 'D' - double
- Class 类型：
  - Lcom/iteblog/T2; - com.iteblog.T2
  - Ljava/io/ObjectOutput; - java.io.ObjectOutput
  - Ljava/lang/String; - String

上面列表的左边是 JVM Type 描述，右边是 Java 里面的类型，也就是我们平时编程使用的。由于

ASM 是直接操纵字节码的，所以会用到 JVM Type。另外，如果你不知道 Java 类型怎么转换到 JVM Type 描述，那么可以使用 ASM 中 org.objectweb.asm.Type 类的 getDescriptor(final Class c) 方法来获取，具体如下：

```
String stringDesc = Type.getDescriptor(String.class);
String intDesc = Type.getDescriptor(int.class);
```

Java 方法声明与 class 文件内部声明的对应关系

在 Java 字节码文件中，方法的方法名和方法的描述都是存储在 Constant pool 中的，且在两个不同的单元里。因此，方法描述中不含有方法名，只含有参数类型和返回类型。Java 字节码里面的方法描述符（method descriptor）和 Java 方法声明的对应关系：

Method declaration in source file	Method descriptor
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;

从上面可知，方法描述符是一个类型描述符（type descriptors）列表，用于在单个字符串中描述方法的参数类型和返回类型。方法描述符用左括号开始，其次是每个参数的类型描述符，接着是一个右括号，接着是方法的返回类型的类型描述符，如果方法返回 void 则使用 V 表示。一旦你了解了 Java 字节码中类型描述符和方法描述符的含义，你就可以很容易理解 Java 字节码中对方法的描述。比如你看到 (I)I 方法描述符，你就知道这个函数接收一个 int 类型的参数，并返回 int 类型的结果。

JVM 中关于类、方法以及字段的访问标记

在平常写 Java 代码的时候，我们会使用 public、private 等修饰符来设置类、方法以及字段的访问情况。在 JVM 中每一种修饰符都有一个 flag 来表示（可以参见JVM 类的 Access flags、JVM 字段的 Access flags、JVM 方法的 Access flags），比如下面是类的 Access flags

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared final; no subclasses

		allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the invokespecial instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared abstract; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an enum type.
ACC_MODULE	0x8000	Is a module, not a class or interface.

ASM 中的 org.objectweb.asm.Opcodes 类里面定义了这些 Access flags：

```
int ACC_PUBLIC = 0x0001; // class, field, method
int ACC_PRIVATE = 0x0002; // class, field, method
int ACC_PROTECTED = 0x0004; // class, field, method
int ACC_STATIC = 0x0008; // field, method
int ACC_FINAL = 0x0010; // class, field, method, parameter
int ACC_SUPER = 0x0020; // class
int ACC_SYNCHRONIZED = 0x0020; // method
int ACC_OPEN = 0x0020; // module
int ACC_TRANSITIVE = 0x0020; // module requires
int ACC_VOLATILE = 0x0040; // field
int ACC_BRIDGE = 0x0040; // method
int ACC_STATIC_PHASE = 0x0040; // module requires
int ACC_VARARGS = 0x0080; // method
int ACC_TRANSIENT = 0x0080; // field
int ACC_NATIVE = 0x0100; // method
int ACC_INTERFACE = 0x0200; // class
int ACC_ABSTRACT = 0x0400; // class, method
int ACC_STRICT = 0x0800; // method
int ACC_SYNTHETIC = 0x1000; // class, field, method, parameter, module *
int ACC_ANNOTATION = 0x2000; // class
int ACC_ENUM = 0x4000; // class(?) field inner
int ACC_MANDATED = 0x8000; // parameter, module, module *
int ACC_MODULE = 0x8000; // class
```

好了，前面已经简单介绍了一下 JVM 关于字节码的一些规范以及 ASM 的一些基础知识，现在我们使用 ASM 实现一个简单的加法类。我们的 Java 代码如下：

```
package com.iteblog;

/**
 * @author iteblog
 * @version 9/27/21 11:52 PM
 */
public class Test {
 public int add(int a, int b) {
  return a + b;
 }
}
```

逻辑很简单，类名是 Test，包名是 com.iteblog，里面定义了一个 int add(int a, int b) 方法，那如果我们使用 asm 实现的话具体如下：

```
public byte[] addByAsm() {
 ClassWriter writer = new ClassWriter(0);
 MethodVisitor mv;
 writer.visit(Opcodes.V1_8, Opcodes.ACC_PUBLIC + Opcodes.ACC_SUPER, "com/iteblog/Test", n
ull, "java/lang/Object", null);

 writer.visitSource("Test.java", null);
 mv = writer.visitMethod(Opcodes.ACC_PUBLIC + Opcodes.ACC_STATIC, "add", "(II)I", null, null)
;
 mv.visitCode();
 Label l0 = new Label();
 mv.visitLabel(l0);
 mv.visitLineNumber(9, l0);
 mv.visitVarInsn(Opcodes.ILOAD, 0);
 mv.visitVarInsn(Opcodes.ILOAD, 1);
 mv.visitInsn(Opcodes.IADD);
 mv.visitInsn(Opcodes.IRETURN);
 Label l1 = new Label();
 mv.visitLabel(l1);
 mv.visitLocalVariable("a", "I", null, l0, l1, 0);
 mv.visitLocalVariable("b", "I", null, l0, l1, 1);
```

```
 mv.visitMaxs(2, 2);
 mv.visitEnd();

 writer.visitEnd();

 return writer.toByteArray();
}
```

因为我们是要生成 Java 字节码，所以直接使用 ClassWriter 就可以。Opcodes.V1_8 代表的是 JVM 1.8，我们这里定义的类为 com/iteblog/Test，它的父类是 java/lang/Object，访问修饰符是 Opcodes.ACC_PUBLIC + Opcodes.ACC_SUPER。然后使用 visitMethod 定义了一个方法，方法的修饰符是 Opcodes.ACC_PUBLIC + Opcodes.ACC_STATIC，方法名是 add，方法描述符是 (II)I，根据前面的知识就是 add 方法输入有两个参数，都是 int 类型，返回值也是 int。add 方法的具体内容实现是下面四行代码：

```
mv.visitVarInsn(Opcodes.ILOAD, 0);
mv.visitVarInsn(Opcodes.ILOAD, 1);
mv.visitInsn(Opcodes.IADD);
mv.visitInsn(Opcodes.IRETURN);
```

这个其实就是本文中最前面Java 代码和 Java 字节码章节介绍的，其中 ILOAD、IADD 以及 IRETURN 都是 JVM 的指令集，这些指令集都是定义在 ASM 中 org.objectweb.asm.Opcodes 类里面，JVM 指令集规范可以参见 The Java Virtual Machine Instruction Set。

ClassWriter 最后生成的就是字节码，我们可以如下方式把它加载到当前类路径里面：

```
public class Iteblog extends ClassLoader {
 public Iteblog(ClassLoader classLoader) {
  super(classLoader);
 }

 public byte[] addByAsm() {
         // 实现看上面
 }

 public static void main(String[] args) throws NoSuchMethodException, InvocationTargetExcept
ion, IllegalAccessException {
  Iteblog loader = new Iteblog(Thread.currentThread().getContextClassLoader());

  byte[] code = loader.addByAsm();
```

```
  Class<?> clazz = loader.defineClass("com.iteblog.Test", code, 0, code.length);
  Method add = clazz.getMethod("add", int.class, int.class);
  System.out.println(add.invoke(null, 1, 2));
 }
}
```

编译运行上面的程序就可以输出 3.

Presto 代码生成

Presto 在表达式计算方面用到了代码生成技术，它是基于 ASM 类库实现的，从前面的 a + b 的例子中可以看到直接操作 ASM 类库会比较麻烦，所以 Presto 对 ASM 进行了更高层次的封装，使用起来会更加方便。相关的代码可以到 presto-bytecode 模块里面看，presto 为我们抽象出 ClassDefinition、MethodDefinition 以及 FieldDefinition 等类来操纵类、方法以及字段字节码。如果我们使用我们使用 Presto 来生成前面的 a+b 的代码，实现如下：

```
package com.iteblog.presto.bytecode;

import com.google.common.collect.ImmutableList;
import org.testng.annotations.Test;

import java.lang.reflect.Method;

import static com.facebook.presto.bytecode.Access.FINAL;
import static com.facebook.presto.bytecode.Access.PUBLIC;
import static com.facebook.presto.bytecode.Access.STATIC;
import static com.facebook.presto.bytecode.Access.a;
import static com.facebook.presto.bytecode.ClassGenerator.classGenerator;
import static com.facebook.presto.bytecode.Parameter.arg;
import static com.facebook.presto.bytecode.ParameterizedType.type;
import static com.facebook.presto.bytecode.expression.BytecodeExpressions.add;
import static org.testng.Assert.assertEquals;

public class Iteblog {
   public void addGenerator()
         throws Exception {
      ClassDefinition classDefinition = new ClassDefinition(
            a(PUBLIC, FINAL),
            "com/iteblog/Test",
            type(Object.class));
```

```
        Parameter argA = arg("a", int.class);
        Parameter argB = arg("b", int.class);

        MethodDefinition method = classDefinition.declareMethod(
                a(PUBLIC, STATIC),
                "add",
                type(int.class),
                ImmutableList.of(argA, argB));

        method.getBody()
                .append(add(argA, argB))
                .retInt();

        Class<?> clazz = classGenerator(getClass().getClassLoader())
                .fakeLineNumbers(true)
                .runAsmVerifier(true)
                .dumpRawBytecode(true)
                .defineClass(classDefinition, Object.class);

        Method add = clazz.getMethod("add", int.class, int.class);
        System.out.println(add.invoke(null, 1, 2));
    }
}
```

可以看到，相比 ASM 代码生成，presto 屏蔽了很多操作指令的细节，操作起来更加方便。

Presto 查询查询使用代码生成的例子

比如我们下面的查询就会用到代码生成技术，

```
select o_orderstatus, count(*) from orders_1x group by o_orderstatus;
```

其中 count 计算内部在创建 HashAggregationOperator 的时候会调用 com.facebook.presto.operator.aggregation.AccumulatorCompiler 类进行代码生成，定义的类名是 com.facebook.presto.$gen.BigintCountAccumulator_20210928_032653_6，实现了 com.facebook.presto.operator.aggregation.Accumulator 接口，并通过 com.facebook.presto.bytecode.DynamicClassLoader 动态加载到 ClassLoader 里面。

总结

本文主要简单介绍了一下 ASM 的基础知识，简单介绍了一下使用 ASM 和 Presto 动态生成代码的方法。限于篇幅和个人能力，本文知识简单介绍了 Presto 的代码生成技术，后期有时间可以考虑更体系的介绍一下 Presto 的代码生成。

参考

asm guide：https://asm.ow2.io/asm4-guide.pdf
The Java Virtual Machine Instruction Set：https://docs.oracle.com/javase/specs/jvms/se9/html/jvms-6.html