



# Aviator引擎解析

Krse Lee 2016/6/23

# 目录

## Catalogue

- Aviator概述
- 编译原理
- ASM生成class
- Aviator编译和执行过程
- 修改代码

# 什么是Aviator?

- Aviator是一个高性能、轻量级的java语言实现的表达式求值引擎，主要用于各种表达式的动态求值。
- Aviator的实现思路与其他轻量级的求值器很不相同，其他求值器一般都是通过解释的方式运行，而Aviator则是直接将表达式编译成Java字节码，交给JVM去执行。
- Aviator = 编译器+执行器

# 使用Aviator

- 将表达式转为Java String；
- 调用AviatorEvaluator.execute()方法执行。

```
public class SayHello {  
    public static void main(String[] args) {  
        if (args.length < 1) {  
            System.err.print("Usage: Java SayHello yourname");  
            System.exit(1);  
        }  
        String yourname = args[0];  
        Map<String, Object> env = new HashMap<>();  
        env.put("yourname", yourname);  
        String result = (String) AviatorEvaluator.execute("hello " + yourname, env);  
        System.out.println(result);  
    }  
}
```

```
// math function  
System.out.println("test math function...");  
System.out.println(AviatorEvaluator.execute("math.abs(-3)"));  
System.out.println(AviatorEvaluator.execute("math.pow(-3, 2)"));  
System.out.println(AviatorEvaluator.execute("math.sqrt(14.0)"));  
System.out.println(AviatorEvaluator.execute("math.log(100)"));  
System.out.println(AviatorEvaluator.execute("math.log10(1000)"));  
System.out.println(AviatorEvaluator.execute("math.sin(20)"));
```

# 使用Aviator

- Aviator表达式可以做的功能包括：
  - 字符串操作
  - 数值、逻辑运算
  - List、数组操作
  - 正则表达式
  - 函数式编程
  - .....
  - 加入自定义方法

```
final List<String> list = new ArrayList<>();
list.add("hello");
list.add(" world");

final int[] array = new int[3];
array[0] = 0;
array[1] = 1;
array[2] = 3;

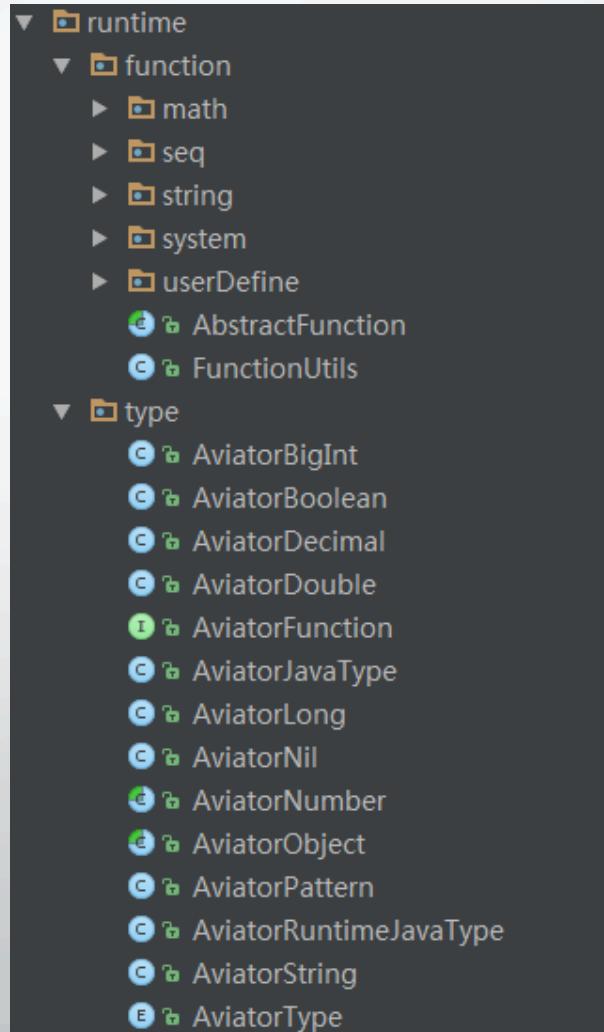
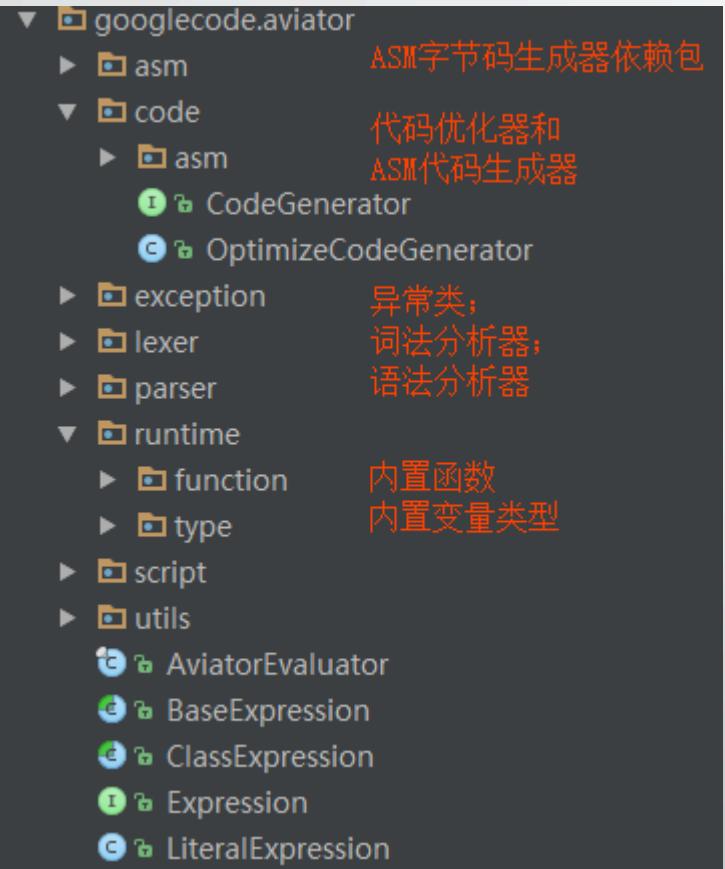
final Map<String, Date> map = new HashMap<>();
map.put("date", new Date());

Map<String, Object> env = new HashMap<>();
env.put("list", list);
env.put("array", array);
env.put("map", map);

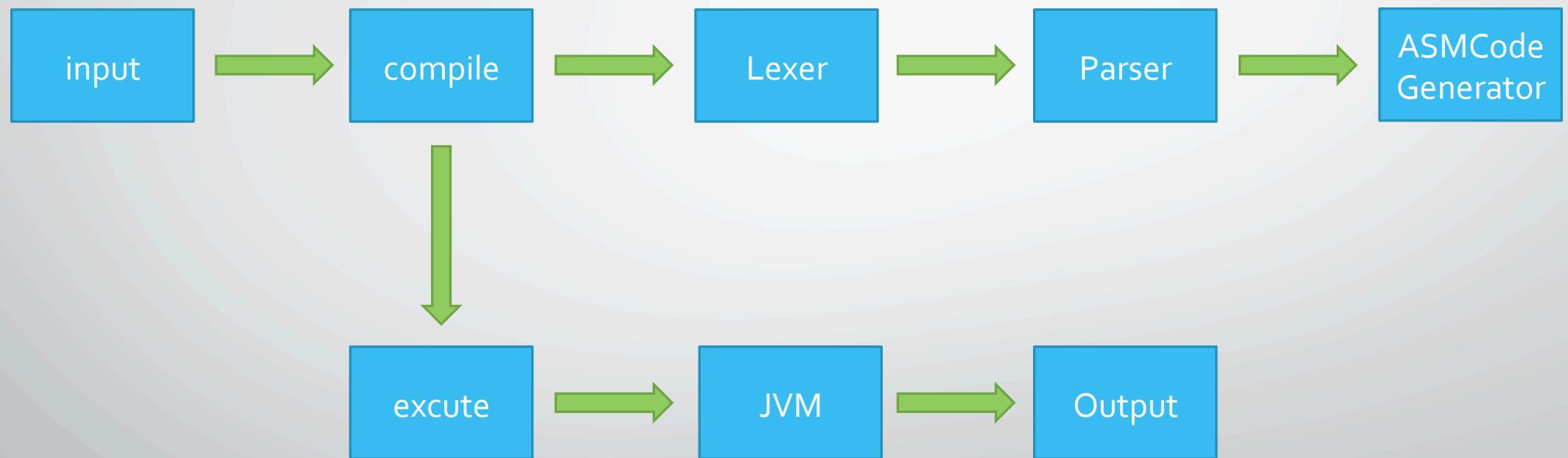
System.out.println(AviatorEvaluator.execute(
    "list[0]+list[1]+'\narray[0]+array[1]+array[2]='+(array[0]+array[1]+array[2]) +' \ntoday is '+map.date ", env));

String email = "killme2008@gmail.com";
Map<String, Object> env = new HashMap<>();
env.put("email", email);
String username = (String) AviatorEvaluator.execute("email=~/([\\w0-8]+)@\\w+[\\.]\\w+/ ? $1:'unknown'", env);
System.out.println(username);
```

# Aviator的组成



# Aviator执行过程

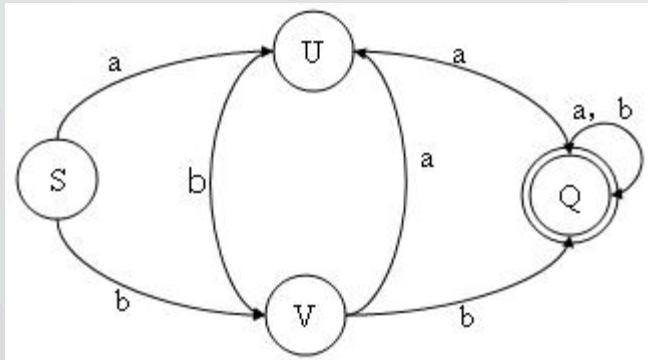


# 编译原理

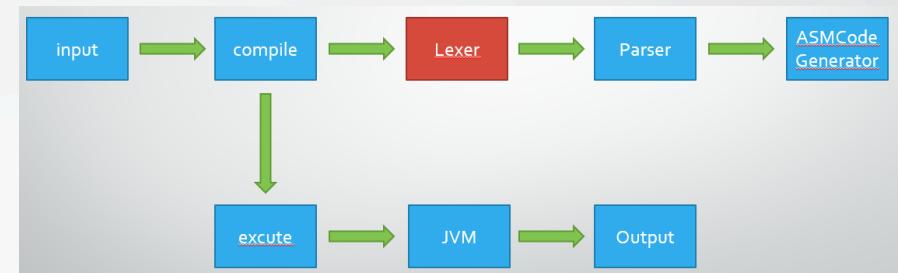
有穷状态自动机——词法分析

自顶向下的LL(1)——文法分析

生成Java字节码——目标代码生成



# 词法分析



- 有穷状态自动机：
  - 用来识别合法的token，如类型、变量名、操作符等。
  - 五元组： $M=(K, \Sigma, f, S, Z)$ ，  $K$ 是有穷状态集合，  $\Sigma$ 是输入符号表，  $f$ 是转移函数，  $S$ 是初态，  $Z$ 是终态集合。
- 图中的有穷状态自动机中，  $K=\{S, U, V, Q\}$ ，  $\Sigma=\{a, b\}$ ， 可接受的token可以有ababba, abbabbb, bb等。

# 如何实现有穷状态自动机

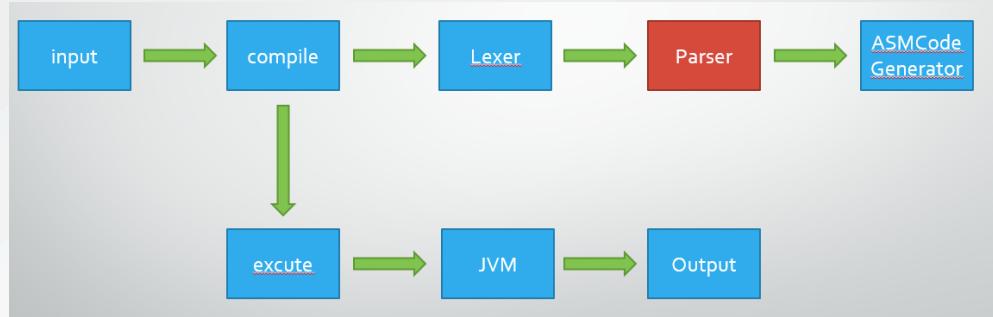
- 以识别一个十六进制数的有穷自动状态机为例：
- 0X1a, 0xff等

```
static final char[] VALID_HEX_CHAR = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A',
                                         'a', 'B', 'b', 'C', 'c', 'D', 'd', 'E', 'e', 'F', 'f' };

public boolean isValidHexChar(char ch) {
    for (char c : VALID_HEX_CHAR) {
        if (c == ch) {
            return true;
        }
    }
    return false;
}
```

```
// if it is a hex digit
if (Character.isDigit(this.peek) && this.peek == '0') {
    this.nextChar();
    if (this.peek == 'x' || this.peek == 'X') {
        this.nextChar();
        StringBuffer sb = new StringBuffer();
        int startIndex = this.iterator.getIndex() - 2;
        long value = 0L;
        do {
            sb.append(this.peek);
            value = 16 * value + Character.digit(this.peek, 16);
            this.nextChar();
        } while (this.isValidHexChar(this.peek));
        return new NumberToken(value, sb.toString(), startIndex);
    } else {
        this.prevChar();
    }
}
```

# 文法分析



- 文法分析是指对代码的语法规规范进行分析，例如判断

```
If ( i == 1){ a=3; }
```

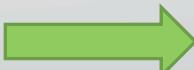
的合法性

- LL文法：自顶向下建立语法树，语法分析器由手工编码实现
- LR文法：自底向上建立语法树，语法分析器由一套算法和自动化工具生成的Action表和Goto表组成。
- Aviator是采用的LL文法分析的方式，这种方式可以处理的语法集有限，但是方便扩展，但是一旦语法集较大，扩展起来就会很麻烦。

# LL文法

- LL文法对语法集的要求比较高，因为语法集中是不能有左递归和二义性的。
- 考虑左边的文法，第一个表达式中的E和第二个表达式中的T都存在左递归，因此需要先改写成右边的LL文法，然后根据文法设计编码。
- 终结符：如+ \* id ()等不能继续展开的符号；
- 非终结符：如E、T、F等可以进一步推导成其他终结符的符号。

```
E -> E + T | T  
T -> T * F | F  
F -> ( E ) | id
```



```
E -> TE'  
E' -> +TE' | #  
T -> FT'  
T' -> *FT' | #  
F -> ( E ) | id
```

# Aviator的BNF文法集

- 例子：

- ternary -> (a==b) ? 1 : 0
- join -> ( a==b )&&( c>o )
- unary -> a | !a | -a
- method -> myFunc(a1, a2)

```
ternary -> bool | bool? ternary : ternary
bool -> join | bool | `` | join
join -> equality | join && equality
equality -> rel | equality == rel | equality =~ rel | equality!= rel
rel -> expr | rel <= expr | rel < expr | rel >= expr | rel > expr
expr -> term | expr + term | expr-term
term -> unary | term * unary | term/unary
unary -> factor | !factor | -factor
factor -> e | number | String | variable | ( ternary ) | / pattern / | method | variable[ternary]
pattern -> char list
method -> variable(paramlist)
paramlist -> ternary | paramlist,ternary
```

# LL文法分析的例子(1)

- 考虑表达式加减的文法，这里存在左递归，因此先改写文法：

原式：

expr → term | expr + term | expr-term

消除左递归：

expr → term expr'  
expr' → +term expr' | -term expr' | #

- 然后按照新的文法进行编码。
- 问题：代码中的expr'哪里去了？

```
public void expr() {
    this.term();
    while (true) {
        if (this.expectLexeme("+")) {
            int start = this.currToken.getStartIndex();
            this.move(true);
            this.term();
            this.codeGenerator.onAdd(this.currToken, start);
        } else if (this.expectLexeme("-")) {
            int start = this.currToken.getStartIndex();
            this.move(true);
            this.term();
            this.codeGenerator.onSub(this.currToken, start);
        } else {
            break;
        }
    }
}
```

# LL文法分析的例子 (2)

- 同理，先消除左递归

原式：

```
equality -> rel == equality | rel =~ equality | rel != equality | rel
```

消除左递归：

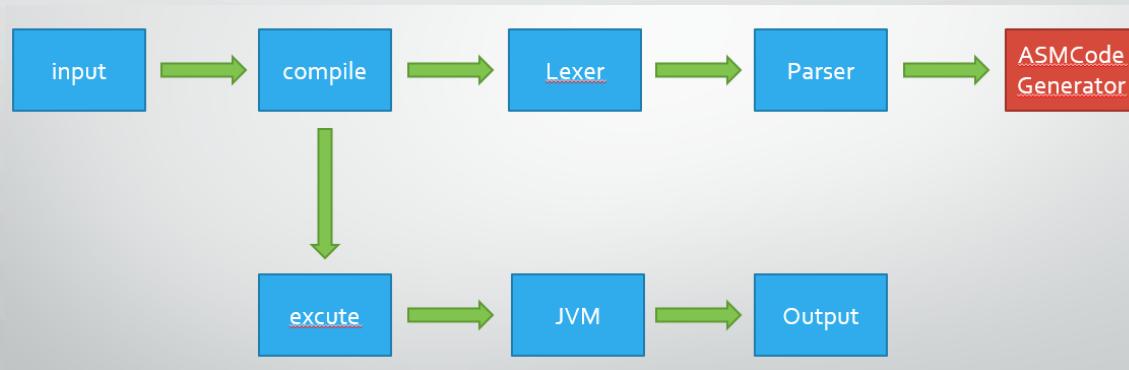
```
equality -> rel equality'  
equality' -> ==rel equality' | =~rel equality' | !=rel equality' | #
```

- 然后按照新的文法进行编码

```
public void equality() {  
    this.rel();  
    while (true) {  
        if (this.expectLexeme("==")) {  
            this.move(true);  
            if (this.expectLexeme("==")) {  
                int start = this.prevToken.getStartIndex();  
                this.move(true);  
                this.rel();  
                this.codeGenerator.onEq(this.currToken, start);  
            } else if (this.expectLexeme("~=")) {  
                // It is a regular expression  
                int start = this.currToken.getStartIndex();  
                this.move(true);  
                this.rel();  
                this.codeGenerator.onMatch(this.currToken, start);  
            } else {  
                this.reportSyntaxError("Aviator doesn't support assignment");  
            }  
        } else if (this.expectLexeme("!=")) {  
            int start = this.currToken.getStartIndex();  
            this.move(true);  
            this.eat("!=");  
            this.rel();  
            this.codeGenerator.onNeq(this.currToken, start);  
        } else {  
            break;  
        }  
    }  
}
```

# 目标代码生成

- Aviator在进行语法解析的同时用到了CodeGenerator接口进行目标代码的生成；
- ASMCodeGenerator是一个CodeGenerator的实现类，它以ASM为基础，将语法解析出的结果翻译为Java字节码，生成.class文件。



ASM生成Java class

# 一个.class文件的结构

类型	名称	数量	
u4	magic	1	魔数： class 文件标识符
u2	minor_version	1	版本号
u2	major_version	1	
u2	constant_pool_count	1	
cp_info	constant_pool	constant_pool_count-1	常量池
u2	access_flags	1	访问标识： class、 interface、 public
u2	this_class	1	
u2	super_class	1	
u2	interfaces_count	1	类索引、 父索引、 接口索引集合
u2	interfaces	interfaces_count	
u2	fields_count	1	
field_info	fields	fields_count	字段表（类成员变量）
u2	methods_count	1	
method_info	methods	methods_count	
u2	attributes_count	1	方法表集合（类成员方法， 包含属性表， 其中代码在属 性表code字段中）
attribute_info	attributes	attributes_count	

# ASM生成Java .class

- 这里是一个用ASM生成.class文件的例子，对应的.class文件反编译得到的.java文件如下：

```
public interface com.asm3.Comparable extends com.asm3.Mesurable {  
    public static final int LESS;  
  
    public static final int EQUAL;  
  
    public static final int GREATER;  
  
    public abstract int compareTo(java.lang.Object);  
}
```

```
public void test() throws IOException {  
    //生成一个类只需要ClassWriter组件即可  
    ClassWriter cw = new ClassWriter(0);  
    //通过visit方法确定类的头部信息  
    cw.visit(Opcodes.V1_5, Opcodes.ACC_PUBLIC+Opcodes.ACC_ABSTRACT+Opcodes.ACC_INTERFACE,  
            "com/asm3/Comparable", null, "java/lang/Object", new String[]{"com/asm3/Mesurable"});  
    //定义类的属性  
    cw.visitField(Opcodes.ACC_PUBLIC+Opcodes.ACC_FINAL+Opcodes.ACC_STATIC,  
                 "LESS", "I", null, new Integer(-1)).visitEnd();  
    cw.visitField(Opcodes.ACC_PUBLIC+Opcodes.ACC_FINAL+Opcodes.ACC_STATIC,  
                 "EQUAL", "I", null, new Integer(0)).visitEnd();  
    cw.visitField(Opcodes.ACC_PUBLIC+Opcodes.ACC_FINAL+Opcodes.ACC_STATIC,  
                 "GREATER", "I", null, new Integer(1)).visitEnd();  
    //定义类的方法  
    cw.visitMethod(Opcodes.ACC_PUBLIC+Opcodes.ACC_ABSTRACT, "compareTo",  
                  "(Ljava/lang/Object;)I", null, null).visitEnd();  
    cw.visitEnd(); //使cw类已经完成  
    //将cw转换成字节数组写到文件里面去  
    byte[] data = cw.toByteArray();  
    File file = new File("D://Comparable.class");  
    FileOutputStream fout = new FileOutputStream(file);  
    fout.write(data);  
    fout.close();  
}
```

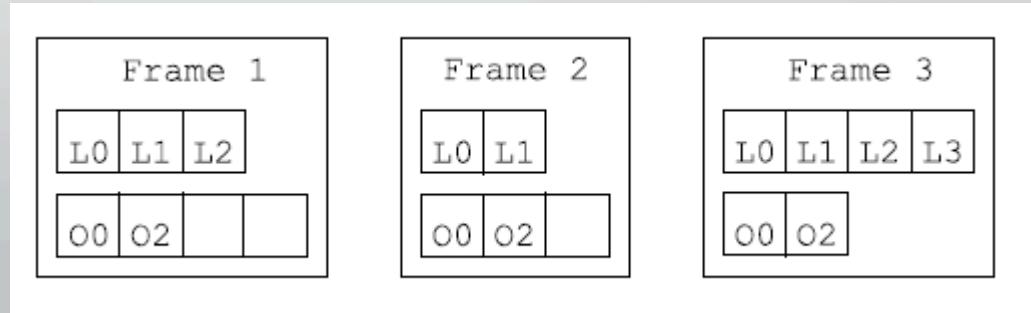
# 如何生成含可执行代码的Java类？

- 使用ClassVisitor生成类结构 (ClassVisitor是ClassWriter的接口);
- 通过cv.visitMethod方法生成方法头，并获取MethodVisitor实例，生成方法Code。
- 怎么生成想要的Code对应的Java字节码呢？

```
ClassVisitor cv = ...;
cv.visit(...);
MethodVisitor mv1 = cv.visitMethod(..., "m1", ...);
mv1.visitCode();
mv1.visitInsn(...);
...
mv1.visitMaxs(...);
mv1.visitEnd();
MethodVisitor mv2 = cv.visitMethod(..., "m2", ...);
mv2.visitCode();
mv2.visitInsn(...);
...
mv2.visitMaxs(...);
mv2.visitEnd();
cv.visitEnd();
```

# JVM 执行模型

- 线程：
  - 执行java code的单位，每条线程包含一个执行帧栈（frame stack）。
- 帧：
  - 一个帧代表一个函数调用，调用时往栈压入一帧，返回时（包括异常返回）弹出一帧。
- 帧结构：
  - 局部变量 + 操作对象栈。
  - 当调用方法时，操作对象栈初始化为空，局部变量初始化为目标对象和方法参数。例如调用 `a.equals(b)` 时，局部变量中会初始化 `a` 和 `b`。



# 执行指令

- Java字节码中的执行指令是byteCode指令，为了方便我们理解，引入了相应的助记符，类似于汇编指令，例如：  
`IADD x1, x2, y`
- 所有的执行指令可以分为两类：
  - 转移型：将局部变量表中的值转移到操作栈，或者反过来；
  - 操作型：只作用于操作对象栈。
- 前者是一些加载、存储指令，如xLOAD、xSTORE；后者比较复杂，种类较多。

# 一些指令的例子

- Stack操作：
  - POP：从操作栈弹出一个元素
  - DUP：压入一个栈顶元素的副本
  - SWAP：交换栈顶的两个元素的位置
- 算术运算：
  - xADD：从操作栈中取出两个元素，相加合并，结果压入栈中。  
• X可以是I、L、F、D。
  - ADD可以替换成SUB、MUL、DIV、REM表示-、×、÷、%。
- 方法调用：
  - INVOKEVIRTUAL owner name desc：调用类名为owner中的name方法，其方法描述符为desc。
  - 调用方法时，从栈中弹出与参数数目等量个对象和一个目标对象，然后压入函数调用的执行结果。
- 跳转：
  - 在满足一定条件时可以跳转到任意带label的指令继续执行，用于编译if、while、for、break等语法。
  - IFEQ label：满足相等条件时弹出一个int值，跳转到label标签对应的指令。

# Java方法代码和其对应的字节码

```
public void checkAndSetF(int f) {  
    if (f >= 0) {  
        this.f = f;  
    } else {  
        throw new IllegalArgumentException();  
    }  
}
```



```
ILOAD 1  
IFLT label  
ALOAD 0  
ILOAD 1  
PUTFIELD pkg/Bean f I  
GOTO end  
label:  
NEW java/lang/IllegalArgumentException  
DUP  
INVOKESPECIAL java/lang/IllegalArgumentException <init> ()V  
ATHROW  
  
end:  
RETURN
```

# 执行过程中的帧状态

```
public void checkAndSetF(int f) {  
    if (f >= 0) {  
        this.f = f;  
    } else {  
        throw new IllegalArgumentException();  
    }  
}
```

因此，我们在编译生成Java字节码时，主要是生成对操作对象栈和局部变量表操作的可执行指令。

State of the execution frame <i>before</i>	Instruction
[pkg/Bean I] []	ILOAD 1
[pkg/Bean I] [I]	IFLT <i>label</i>
[pkg/Bean I] []	ALOAD 0
[pkg/Bean I] [pkg/Bean]	ILOAD 1
[pkg/Bean I] [pkg/Bean I]	PUTFIELD
[pkg/Bean I] []	GOTO <i>end</i>
[pkg/Bean I] []	<i>label</i> :
[pkg/Bean I] []	NEW
[pkg/Bean I] [Uninitialized( <i>label</i> )]	DUP
[pkg/Bean I] [Uninitialized( <i>label</i> ) Uninitialized( <i>label</i> )]	INVOKESPECIAL
[pkg/Bean I] [java/lang/IllegalArgumentException]	ATHROW
[pkg/Bean I] []	<i>end</i> :
[pkg/Bean I] []	RETURN

# Aviator的编译和执行过程

# 编译——程序入口

- 主方法中调用compile或者execute方法最终会进入右图的compile方法中。
- 调用 AviatorEvaluate 中的 compile 方法，根据表达式是否做了缓存，调用 innerCompile 方法。

```
public static Expression compile(final String expression, final boolean cached) {  
    if (expression == null || expression.trim().length() == 0) {  
        throw new CompileExpressionErrorException("Blank expression");  
    }  
  
    if (cached) {  
        FutureTask<Expression> task = cacheExpressions.get(expression);  
        if (task != null) {  
            return getCompiledExpression(expression, task);  
        }  
        task = new FutureTask<Expression>(() -> { return innerCompile(expression, cached); });  
        FutureTask<Expression> existedTask = cacheExpressions.putIfAbsent(expression, task);  
        if (existedTask == null) {  
            existedTask = task;  
            existedTask.run();  
        }  
        return getCompiledExpression(expression, existedTask);  
    }  
    else {  
        return innerCompile(expression, cached);  
    }  
}
```

# 编译——步骤

- innerCompile中构建词法分析器ExpressionLexer、语法分析器ExpressionParser和目标代码生成器CodeGenerator。
- newCodeGenerator方法会根据设置的optimize值的不同来构建ASM类字节码生成器或者代码优化器。
- optimize初始值为EVAL

```
private static Expression innerCompile(final String expression, boolean cached) {  
    ExpressionLexer lexer = new ExpressionLexer(expression);  
    CodeGenerator codeGenerator = newCodeGenerator(cached);  
    ExpressionParser parser = new ExpressionParser(lexer, codeGenerator);  
    return parser.parse();  
}
```

```
private static CodeGenerator newCodeGenerator(boolean cached) {  
    switch (optimize) {  
        case COMPILE:  
            ASMCodeGenerator asmCodeGenerator =  
                new ASMCodeGenerator(getAviatorClassLoader(cached), traceOutputStream, trace);  
            asmCodeGenerator.start();  
            return asmCodeGenerator;  
        case EVAL:  
            return new OptimizeCodeGenerator(getAviatorClassLoader(cached), traceOutputStream, trace);  
        default:  
            throw new IllegalArgumentException("Unknown option " + optimize);  
    }  
}
```

# 编译——文法分析

```
public Expression parse() {
    this.ternary();
    if (this.parenDepth > 0) {
        this.reportSyntaxError("insert ')' to complete Expression");
    }
    if (this.bracketDepth > 0) {
        this.reportSyntaxError("insert ']' to complete Expression");
    }
    return this.codeGenerator.getResult();
}
```

- `ternary`是BNF文法的入口非终结符，从该方法开始将每一个非终结符开始的产生式定义为一个方法。方法中实现的文法是消除了左递归和左公因子的。

```
public void ternary() {
    this.join();
    if (this.lookhead == null || this.expectLexeme(":") || this.expectLexeme(",") {
        return;
    }
    if (this.expectLexeme("?")) {
        this.move(true);
        this.codeGenerator.onTernaryBoolean(this.lookhead);
        this.ternary();
        if (this.expectLexeme(":")) {
            this.move(true);
            this.codeGenerator.onTernaryLeft(this.lookhead);
            this.ternary();
            this.codeGenerator.onTernaryRight(this.lookhead);
        }
        else {
            this.reportSyntaxError("expect ':'");
        }
    }
    else {
        if (this.expectLexeme(")")) {
            if (this.parenDepth > 0) {
                return;
            }
        }
    }
}
```

# 编译——代码优化

```
public Expression getResult() {  
    // execute literal expression  
    while (this.execute() > 0) {  
        ;  
    }  
}
```

- 在之前的代码中CodeGenerator指向的是一个OptimizeCodeGenerator，因此上一步getResult调用的也是优化器的方法。
- getResult首先对分析后的代码进行优化，这里主要是运算符优化，包括合并已知量、删除多余运算、删除无用赋值等。

```
private int execute() {  
    int exeCount = 0;  
    final int size = this.tokenList.size();  
    this.printTokenList();  
    for (int i = 0; i < size; i++) {  
        Token<?> token = this.tokenList.get(i);  
        if (token.getType() == TokenType.Operator) {  
            final OperatorToken op = (OperatorToken) token;  
            final OperatorType operatorType = op.getOperatorType();  
            final int operandCount = operatorType.getOperandCount();  
            switch (operatorType) {  
                case FUNC:  
                case INDEX:  
                    // Could not optimize function and index call  
                    break;  
                default:  
                    Map<Integer, DelegateTokenType> index2DelegateType = this.getIndex2DelegateTypeMap(operatorType);  
                    final int result = this.executeOperator(i, operatorType, operandCount, index2DelegateType);  
                    if (result < 0) {  
                        this.compactTokenList();  
                        return exeCount;  
                    }  
                    exeCount += result;  
                    break;  
            }  
        }  
    }  
    this.compactTokenList();  
    return exeCount;  
}
```

# 编译——目标代码生成（一）

```
Map<String, Integer/* counter */> variables = new LinkedHashMap<String, Integer>();
Map<String, Integer/* counter */> methods = new HashMap<String, Integer>();
for (Token<?> token : this.tokenList) {
    switch (token.getType()) {
        case Variable:
            String varName = token.getLexeme();
            if (!variables.containsKey(varName)) {
                variables.put(varName, 1);
            } else {
                variables.put(varName, variables.get(varName) + 1);
            }
            break;
        case Delegate:
            DelegateToken delegateToken = (DelegateToken) token;
            if (delegateToken.getDelegateTokenType() == DelegateTokenType.Method_Name) {
                Token<?> realToken = delegateToken.getToken();
                if (realToken.getType() == TokenType.Variable) {
                    String methodName = token.getLexeme();
                    if (!methods.containsKey(methodName)) {
                        methods.put(methodName, 1);
                    } else {
                        methods.put(methodName, methods.get(methodName) + 1);
                    }
                }
            } else if (delegateToken.getDelegateTokenType() == DelegateTokenType.Array) {
            }
    }
}
```

```
// call asm to generate byte codes
this.callASM(variables, methods);

// Last token is a literal token, then return a LiteralExpression
if (this.tokenList.size() <= 1) {
    if (this.tokenList.isEmpty()) {
        return new LiteralExpression(null, new ArrayList<String>(variables.keySet()));
    }
    final Token<?> lastToken = this.tokenList.get(0);
    if (this.isLiteralToken(lastToken)) {
        return new LiteralExpression(this.getAviatorObjectFromToken(lastToken).getValue(null),
            new ArrayList<String>(variables.keySet()));
    }
}

// get result from asm
return this.asmCodeGenerator.getResult();
}
```

- getResult方法对代码进行优化之后开始使用ASM生成目标代码。
- 首先，它将代码中的变量和方法分别放到两个Map中：variables和methods。
- 接着调用callASM方法，最后调用asm的getResult方法。

# 编译——目标代码生成（二）

- callASM方法中会调用asmCodeGenerator相关方法进行初始化。
- initVariables生成成员变量；
- initMethods把函数对象编译为成员变量；
- Start主要有两个步骤：
  - 生成构造函数，初始化成员变量和成员函数；
  - 生成该类被ClassLoader调用方法execute0的代码。
- 初始化完成后遍历表达式代码，识别每个操作符和指代式（如方法名、数组），使用ASM生成frame局部变量和操作对象栈的执行指令，加入到方法execute0中。

```
private void callASM(Map<String, Integer/* counter */> variables,
    this.asmCodeGenerator.initVariables(variables);
    this.asmCodeGenerator.initMethods(methods);
    this.asmCodeGenerator.start();

    for (int i = 0; i < this.tokenList.size(); i++) {
        Token<?> token = this.tokenList.get(i);
        switch (token.getType()) {
            case Operator:
                OperatorToken op = (OperatorToken) token;
                switch (op.getOperatorType()) {
                    case ADD:
                        this.asmCodeGenerator.onAdd(token);
                        break;
                    case SUB:
                        this.asmCodeGenerator.onSub(token);
                        break;
                }
            }
        }
    }
}
```

# 目标代码生成

## ——类的成员变量

- `initVariables`和`initMethod`使用之前生成的变量表和自定义方法表生成构造类的成员变量和成员方法。
- 生成的成员变量类型都是`private final`。

```
public void initVariables(Map<String, Integer> varTokens) {  
    this.varTokens = varTokens;  
    for (String outerVarName : varTokens.keySet()) {  
        // Use inner variable name instead of outer variable name  
        String innerVarName = this.getInnerName();  
        this.innerVarMap.put(outerVarName, innerVarName);  
        this.checkClassAdapter.visitField(ACC_PRIVATE + ACC_FINAL, innerVarName,  
            "Lcom/alipay/aviator/runtime/type/AviatorJavaType;", null, null).visitEnd();  
    }  
    // 执行跟踪开关打开时, 才进行跟踪  
    if (StepTraceThreadContext.get()) {  
        this.checkClassAdapter.visitField(ACC_PUBLIC + Opcodes.ACC_STATIC, "tracer",  
            "Ljava/lang/String;", null, null).visitEnd();  
    }  
}  
  
public void initMethods(Map<String, Integer> methods) {  
    this.methodTokens = methods;  
    for (String outerMethodName : methods.keySet()) {  
        // Use inner method name instead of outer method name  
        String innerMethodName = this.getInnerName();  
        this.innerMethodMap.put(outerMethodName, innerMethodName);  
        this.checkClassAdapter.visitField(ACC_PRIVATE + ACC_FINAL, innerMethodName,  
            "Lcom/alipay/aviator/runtime/type/AviatorFunction;", null, null).visitEnd();  
    }  
}
```

# 目标代码生成 ——类的构造函数

```
//生成构造函数头 public XX( List<Long> list )
this.mv = this.checkClassAdapter.visitMethod(ACC_PUBLIC, "<init>", "(Ljava/util/List;)V", null, null);
this.mv.visitCode();
//此时局部变量表中为 [ this , list ]，将它们压栈
this.mv.visitVarInsn(ALOAD, 0);
this.mv.visitVarInsn(ALOAD, 1);
//调用父类 ClassExpression 的构造方法，弹出 list和this
this.mv.visitMethodInsn(INVOKESTATIC, "com/googlecode/aviator/ClassExpression", "<init>",
    "(Ljava/util/List;)V");
//现在栈是空的 []
if (!this.innerVarMap.isEmpty()) {
    //初始化所有变量型的成员变量
    for (Map.Entry<String, String> entry : this.innerVarMap.entrySet()) {
        String outerName = entry.getKey();
        String innerName = entry.getValue();
        //变量值压栈 [ v ]
        this.mv.visitVarInsn(ALOAD, 0);
        //new 一个类型对象，变量类型压栈，并创建副本[ v , type , type ]
        this.mv.visitTypeInsn(NEW, "com/googlecode/aviator/runtime/type/AviatorJavaType");
        this.mv.visitInsn(DUP);
        //压入类型名称outerName [ v , type , type , outerName]
        this.mv.visitLdcInsn(outerName);
        //访问类型对象的构造方法，弹出outerName和type [ v , type ]
        this.mv.visitMethodInsn(INVOKESTATIC, "com/googlecode/aviator/runtime/type/AviatorJavaType",
            "<init>", "(Ljava/lang/String;)V");
        //赋值，弹出值和对象引用 []
        this.mv.visitFieldInsn(PUTFIELD, this.className, innerName,
            "Lcom/googlecode/aviator/runtime/type/AviatorJavaType;");
    }
}
```

```
if (!this.innerMethodMap.isEmpty()) {
    //初始化所有函数型成员变量
    for (Map.Entry<String, String> entry : this.innerMethodMap.entrySet()) {
        String outerName = entry.getKey();
        String innerName = entry.getValue();
        //函数值压栈 [ f ]
        this.mv.visitVarInsn(ALOAD, 0);
        //函数名压栈 [ f , outerName ]
        this.mv.visitLdcInsn(outerName);
        //调用AviatorEvaluator.getFunction( String s )，返回一个AviatorFunction对象 [ AviatorFunction ]
        this.mv.visitMethodInsn(INVOKESTATIC, "com/googlecode/aviator/AviatorEvaluator", "getFunction",
            "(Ljava/lang/String;)Lcom/googlecode/aviator/runtime/type/AviatorFunction;");
        //赋值 []
        this.mv.visitFieldInsn(PUTFIELD, this.className, innerName,
            "Lcom/googlecode/aviator/runtime/type/AviatorFunction;");
    }
}
```

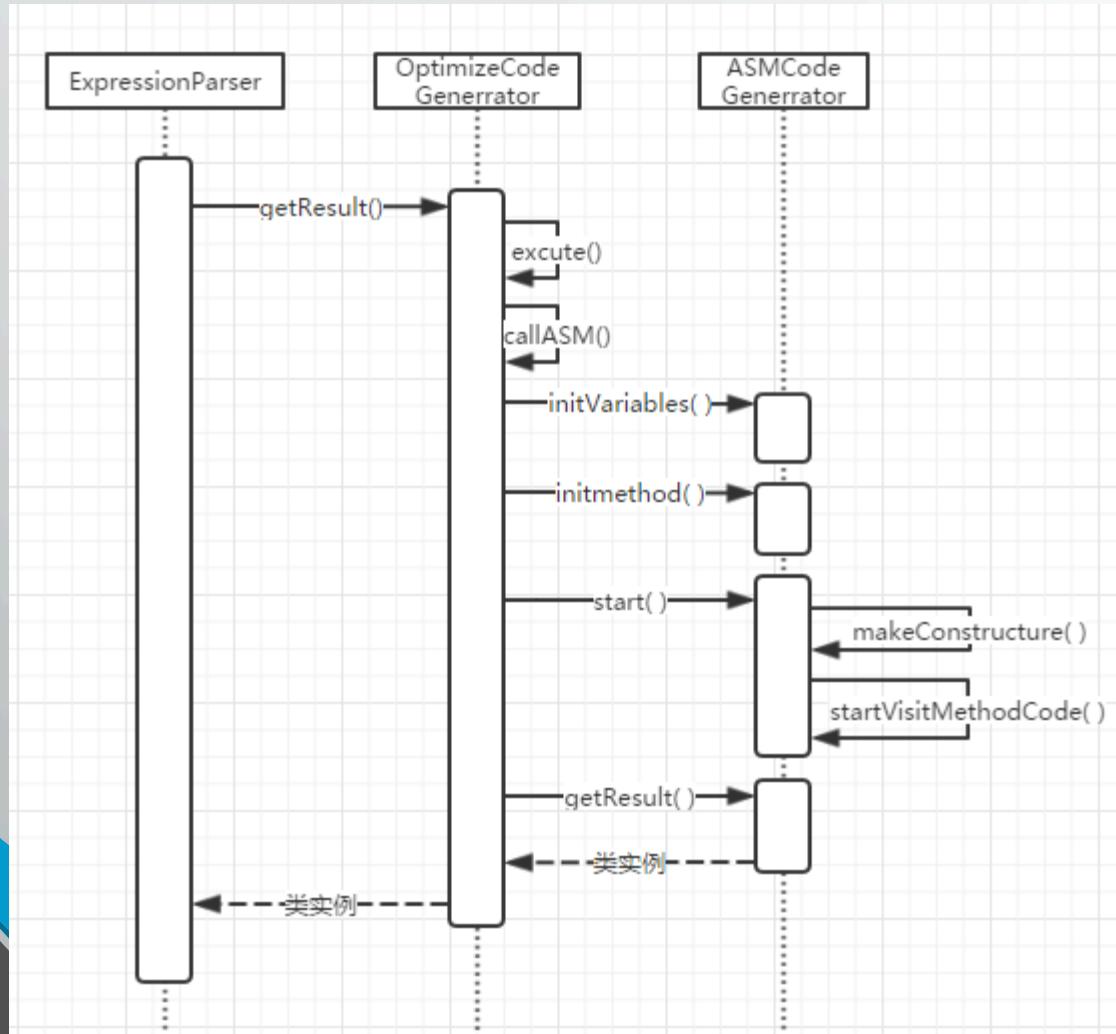
# 编译——目标代码生成（四）

- getResult方法会生成Java字节码，并将其载入到指定的类加载器中，并构造成Java可执行类实例。

```
public Expression getResult() {
    this.end();

    byte[] bytes = this.classWriter.toByteArray();
    try {
        Class<?> defineClass = this.classLoader.defineClass(this.className, bytes);
        Constructor<?> constructor = defineClass.getConstructor(List.class);
        return (Expression) constructor.newInstance(new ArrayList<String>(this.varTokens.keySet()));
    }
    catch (Exception e) {
        throw new CompileExpressionErrorException("define class error", e);
    }
}
```

# 目标代码生成过程总结



编译生成类头

生成表达式中变量对应的成员变量

生成表达式中自定义函数对应的成员变量

生成构造函数并初始化成员变量

生成被调用的execute0方法代码

# 执行

- 经过编译得到的表达式调用execute方法，传入变量表env到执行入口。
- execute0是一个抽象方法，它执行具体的表达式生成的class方法。
- 结果返回是一个Object，经过我们自己的强制转换成为最后的结果。

```
protected void startVisitMethodCode() {
    this.mv = this.checkClassAdapter.visitMethod(ACC_PUBLIC + ACC_FINAL, "execute0",
        "(Ljava/util/Map;)Ljava/lang/Object;",
        "(Ljava/util/Map;Ljava/lang/String;Ljava/lang/Object;)Ljava/lang/Object;", null);

    // 开始生成execute0方法的具体内容
    this.mv.visitCode();
}
```

```
public static Object execute(String expression, Map<String, Object> env, boolean cached) {
    Expression compiledExpression = compile(expression, cached);
    if (compiledExpression != null) {
        return compiledExpression.execute(env);
    }
    else {
        throw new ExpressionRuntimeException("Null compiled expression for " + expression);
    }
}

public Object execute(Map<String, Object> env) {
    if (env == null) {
        env = Collections.emptyMap();
    }

    try {
        return execute0(env);
    }
    catch (ExpressionRuntimeException e) {
        throw e;
    }
    catch (Throwable e) {
        throw new ExpressionRuntimeException("Execute expression error", e);
    }
}
```

# Tips：打印Aviator的执行记录

```
public static void main(String[] args) {
    try {
        AviatorEvaluator.setTrace(true);
        AviatorEvaluator.setTraceOutputStream(new FileOutputStream(new File("d:\\aviator.log")));
    } catch (IOException e) {
        e.printStackTrace();
    }

    String expression = "a-(b-c)>100";
    // 编译表达式
    Expression compiledExp = AviatorEvaluator.compile(expression);

    Map<String, Object> env = new HashMap<>();
    env.put("a", 100.3);
    env.put("b", 45);
    env.put("c", -199.100);

    // 执行表达式
    Boolean result = (Boolean) compiledExp.execute(env);
    System.out.println(result);
}
```

- Aviator提供了一个接口将生成的java字节码打印到日志中。
- 这样我们可以清楚地知道最后一个表达式被编译成了什么样子，也可以方便我们排错。

# Aviator生成的日志

- $a - (b - c) > 100$

```
public class Script_1467286617776_0 extends com.googlecode.aviator/ClassExpression {  
  
    // access flags 18  
    private final Lcom/googlecode/aviator/runtime/type/AviatorJavaType; var_0  
  
    // access flags 18  
    private final Lcom/googlecode/aviator/runtime/type/AviatorJavaType; var_1  
  
    // access flags 18  
    private final Lcom/googlecode/aviator/runtime/type/AviatorJavaType; var_2  
  
    // access flags 1  
    public <init>(Ljava/util/List;)V  
        ALOAD 0  
        ALOAD 1  
        INVOKESTATIC com.googlecode.aviator/ClassExpression.<init> (Ljava/util/List;)V  
        ALOAD 0  
        NEW com.googlecode.aviator/runtime/type/AviatorJavaType  
        DUP  
        LDC "a"  
        INVOKESTATIC com.googlecode.aviator/runtime/type/AviatorJavaType.<init> (Ljava/lang/String;)V  
        PUTFIELD Script_1467286617776_0.var_0 : Lcom/googlecode/aviator/runtime/type/AviatorJavaType;  
        ALOAD 0  
        NEW com.googlecode.aviator/runtime/type/AviatorJavaType  
        DUP  
        LDC "b"  
        INVOKESTATIC com.googlecode.aviator/runtime/type/AviatorJavaType.<init> (Ljava/lang/String;)V  
        PUTFIELD Script_1467286617776_0.var_1 : Lcom/googlecode/aviator/runtime/type/AviatorJavaType;  
        ALOAD 0  
        NEW com.googlecode.aviator/runtime/type/AviatorJavaType  
        DUP  
        LDC "c"  
        INVOKESTATIC com.googlecode.aviator/runtime/type/AviatorJavaType.<init> (Ljava/lang/String;)V  
        PUTFIELD Script_1467286617776_0.var_2 : Lcom/googlecode/aviator/runtime/type/AviatorJavaType;  
        RETURN  
        MAXSTACK = 4  
        MAXLOCALS = 1  
}  
  
变量a、b、c  
构造函数，初始化变量
```

# Aviator生成的日志（续）

```
public final execute0(Ljava/util/Map;)Ljava/lang/Object;
    ALOAD 0
    GETFIELD Script_1467286617776_0.var_0 : Lcom/googlecode/aviator/runtime/type/AviatorJavaType;
    ALOAD 0
    GETFIELD Script_1467286617776_0.var_1 : Lcom/googlecode/aviator/runtime/type/AviatorJavaType;
    ALOAD 0
    GETFIELD Script_1467286617776_0.var_2 : Lcom/googlecode/aviator/runtime/type/AviatorJavaType;
    ALOAD 1
    INVOKEVIRTUAL com/googlecode/aviator/runtime/type/AviatorObject.sub (Lcom/googlecode/aviator/runtime/type/
    AviatorObject;Ljava/util/Map;)Lcom/googlecode/aviator/runtime/type/AviatorObject;           从局部变量表载入变量a、b、c

    ALOAD 1
    INVOKEVIRTUAL com/googlecode/aviator/runtime/type/AviatorObject.sub (Lcom/googlecode/aviator/runtime/type/
    AviatorObject;Ljava/util/Map;)Lcom/googlecode/aviator/runtime/type/AviatorObject;           调用SUB方法，计算b-c

    ALOAD 1
    INVOKEVIRTUAL com/googlecode/aviator/runtime/type/AviatorObject.sub (Lcom/googlecode/aviator/runtime/type/
    AviatorObject;Ljava/util/Map;)Lcom/googlecode/aviator/runtime/type/AviatorObject;           调用SUB方法计算a-result

    LDC 100
    INVOKESTATIC com/googlecode/aviator/runtime/type/AviatorLong.valueOf (J)Lcom/googlecode/aviator/runtime/type/AviatorLong;   压栈常量100，比较其与
    ALOAD 1
    INVOKEVIRTUAL com/googlecode/aviator/runtime/type/AviatorObject.compare (Lcom/googlecode/aviator/runtime/type/AviatorObject;Ljava/util/Map;)I   上一步计算结果大小

    IFLE L0
    GETSTATIC com/googlecode/aviator/runtime/type/AviatorBoolean.TRUE : Lcom/googlecode/aviator/runtime/type/AviatorBoolean;   如果大于等于，跳转到L0；
    GOTO L1
    L0
    GETSTATIC com/googlecode/aviator/runtime/type/AviatorBoolean.FALSE : Lcom/googlecode/aviator/runtime/type/AviatorBoolean;   否则压栈true，跳转到L1。
    L1
    ALOAD 1
    INVOKEVIRTUAL com/googlecode/aviator/runtime/type/AviatorObject.getValue (Ljava/util/Map;)Ljava/lang/Object;           调用getValue方法，压栈返回值
    ARETURN
    MAXSTACK = 4
    MAXLOCALS = 2
    返回返回值
```

修改代码  
增加条件判断语句  
增加lambda语法  
增加自定义函数

# 增加条件判断文法

- Aviator的原生文法是不包含条件判断语法的，为了满足项目的需要，扩展了这一部分文法。
- 原来的文法入口是ternary，而现在将入口改为了stmts，但是新的文法只支持if else形式的条件判断，不支持连续的if elseif else的文法。
- 另外增加了谓词文法，去掉了原来的bool推导。

```
ternary -> bool | bool? ternary : ternary  
bool -> join | bool | `` | join  
join -> equality | join && equality
```



```
stmts -> stmt stmts | #  
stmt -> if (ternary) block else block | variable = ternary | ternary  
block -> {stmts} | stmt  
ternary -> join: | join?ternary:ternary | join) | join] | join( | join[  
join -> and||join | and  
and -> bitOr&&and | bitOr  
bitOr -> xor|bitOr | xor  
xor -> bitAnd^xor | bitAnd  
bitAnd -> equality&bitAnd | equality
```

# Lambda表达式

- “Lambda 表达式”(lambda expression)是一个匿名函数，Lambda表达式基于数学中的 $\lambda$ 演算得名，直接对应于其中的lambda抽象(lambda abstraction)，是一个匿名函数，即没有函数名的函数。
- 其一般形式是  $x \rightarrow \text{func}(x)$ ，表示调用方法func()，参数为x。例如：
  - $x \rightarrow x+1$  对x加上1
  - $(x, y) \rightarrow (y, x)$  交换二元组中两个元素的位置
  - $x \rightarrow \text{abs}(x)$  对x取绝对值
  - $\text{map}(\text{list}, x \rightarrow x+1)$  对list中的每个元素加1
  - $\text{filter}(\text{list}, x > 10)$  过滤掉list中大于10的元素

# 扩展lambda文法

- 原来的文法中，对于方法的文法定义如下：

```
method -> variable(paramlist)
```

```
paramlist -> ternary | paramlist,ternary
```

- 现在加入了lambda表达式，这一部分文法扩展为：

```
method -> lambda | variable(paramlist)
```

```
lambda -> (variable)=>{expr} ternary
```

```
paramlist -> ternary | paramlist,ternary
```

- 修改的部分有：

- 按照文法规则修改语法分析代码；

- 修改CodeGenerator以支持lambda目标代码生成；

- 修改内置的seq方法map、reduce、filter等以支持lambda中调用。

# 修改seqMapFuntion以支持lambda

```
public class SeqMapFunction extends AbstractFunction {  
  
    @Override  
    @SuppressWarnings("unchecked")  
    public AviatorObject call(Map<String, Object> env, AviatorObject arg1, AviatorObject arg2)  
  
        Object first = arg1.getValue(env);  
        AviatorFunction fun = FunctionUtils.getFunction(arg2, env, 1);  
        if (fun == null) {  
            throw new ExpressionRuntimeException("There is no function named " + ((AviatorJava  
        }  
        if (first == null) {  
            throw new NullPointerException("null seq");  
        }  
        Class<?> clazz = first.getClass();  
  
        if (Collection.class.isAssignableFrom(clazz)) {  
            Collection<Object> result = null;  
            try {  
                result = (Collection<Object>) clazz.newInstance();  
            }  
            catch (Throwable t) {  
                // ignore  
                result = new ArrayList<Object>();  
            }  
            for (Object obj : (Collection<?>) first) {  
                result.add(fun.call(env, new AviatorRuntimeJavaType(obj)).getValue(env));  
            }  
            return new AviatorRuntimeJavaType(result);  
        }  
    }  
}
```



```
public AviatorObject call(Map<String, Object> env, AviatorObject arg1, AviatorObject arg2)  
  
    Object first = arg1.getValue(env);  
    AviatorFunction fun;  
    if (arg2 instanceof AviatorLambda) {  
        AviatorLambda lambda = (AviatorLambda) arg2;  
        if (lambda.getParams().size() != 1) {  
            throw new IllegalArgumentException("Wrong number of args ("  
                + lambda.getParams().size()  
                + ") passed to lambda expression");  
        }  
        fun = new LambdaFunction("_lambda_tmp_", lambda.getExpression(), lambda.getParams());  
    } else {  
        fun = FunctionUtils.getFunction(arg2, env, 1);  
    }  
    if (fun == null) {  
        throw new ExpressionRuntimeException("There is no function named " + ((AviatorJava  
    }  
    if (first == null) {  
        if(FakeThreadContext.get()) {  
            return new AviatorRuntimeJavaType(null);  
        }  
        throw new NullPointerException("null seq");  
    }  
    Class<?> clazz = first.getClass();  
  
    if (Collection.class.isAssignableFrom(clazz)) {  
        Collection<Object> result = null;  
        try {  
            result = (Collection<Object>) clazz.newInstance();  
        }
```

- Aviator原生seq函数并没有设计用来支持lambda表达式，因此直接使用编译器无法将lambda解析为参数，只能解析类似 map(list, abs)这样的方法。
- 修改之后表达式调用函数时进行了一次判断，如果传入的是lambda对象，就将seq函数注册为一个调用lambda的函数。

# 修改CodeGenerator ——支持lambda对象的目标代码生成

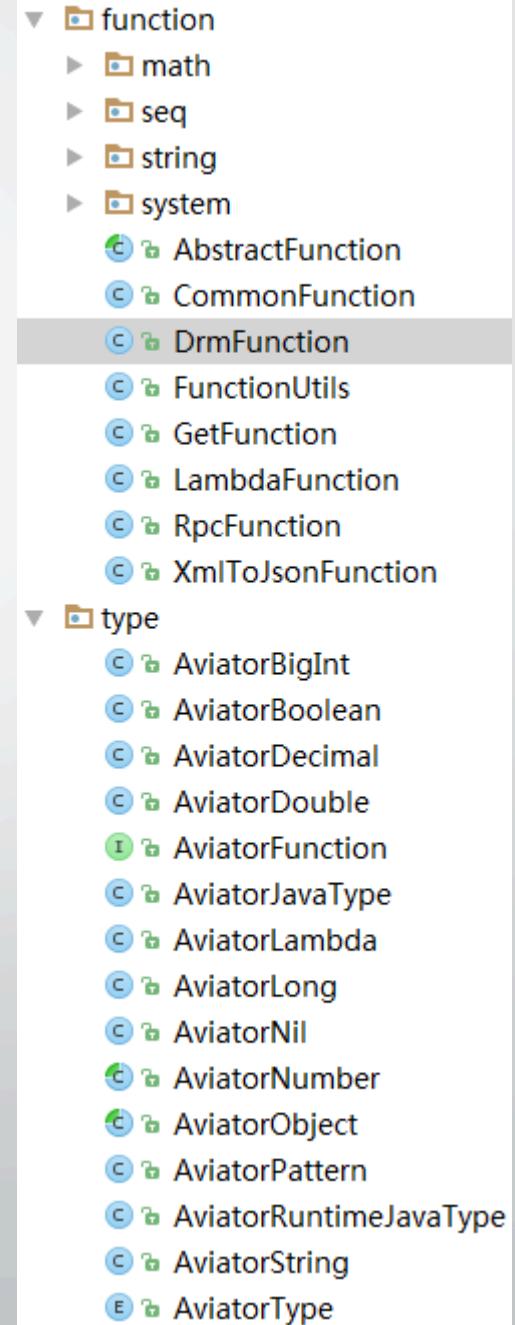
```
public void onLambdaObject(Token<?> lookhead, int start) {  
    LambdaToken lambda = (LambdaToken) lookhead;  
    String expr = lambda.getLexeme();  
    List<String> params = lambda.getParams();  
    //压栈lambda对象类型AviatorLambda [ AviatorLambda ]  
    this.s.pushOperand(0);  
    this.mv.visitTypeInsn(NEW, "com/alipay/aviator/runtime/type/AviatorLambda");  
    //创建lambda副本并压栈 [ AviatorLambda , AviatorLambda ]  
    this.s.pushOperand(0);  
    this.mv.visitInsn(DUP);  
    //将lambda表达式内容压栈 [ AviatorLambda , AviatorLambda , expr ]  
    this.s.pushOperand(0);  
    this.mv.visitLdcInsn(expr);  
    //参数对象数组size和数组对象压栈 [ AviatorLambda , AviatorLambda , expr , size ]  
    this.s.pushOperand(0);  
    mv.visitInsn(Opcodes.ICONST_0 + params.size());  
    //弹出数组size, 压入数组对象 [ AviatorLambda , AviatorLambda , expr , ArrayObject ]  
    mv.visitTypeInsn(Opcodes.ANEWARRAY, "java/lang/Object");  
    this.s.popOperand();  
    //将数组对象放入局部变量表  
    int localIndex = this.getLocalIndex();  
    mv.visitVarInsn(ASTORE, localIndex);
```

```
//数组中的每一个元素  
for (int i = 0; i < params.size(); i++) {  
    mv.visitVarInsn(ALOAD, localIndex);  
    mv.visitInsn(Opcodes.ICONST_0 + i);  
    mv.visitLdcInsn(params.get(i));  
    mv.visitInsn(Opcodes.AASTORE);  
}  
//当前对象this压栈 [ AviatorLambda , AviatorLambda , expr , ArrayObject , this ]  
this.pushOperand(0);  
this.mv.visitVarInsn(ALOAD, localIndex);  
//数组转为List的方法, 申明为 List method(Object[] o) { return Array.asList(o) }  
//弹出this和数组对象, 压入一个List对象 [ AviatorLambda , AviatorLambda , expr , List ]  
this.popOperand();  
this.popOperand();  
this.pushOperand(0);  
this.mv.visitMethodInsn(INVOKESTATIC, "java/util/Arrays", "asList",  
    "([Ljava/lang/Object;)Ljava/util/List;");  
//lambda编译后对应的类构造方法, 申明为 void method(String s, List l)  
//弹出List对象、表达式的内容、Lambda对象 [ AviatorLambda ]  
this.popOperand();  
this.popOperand();  
this.popOperand();  
this.pushOperand(0);  
this.mv.visitMethodInsn(INVOKESTATIC, "com/alipay/aviator/runtime/type/AviatorLambda",  
    "<init>", "(Ljava/lang/String;Ljava/util/List;)V");
```

- 主要是对AsmCodeGenerator的修改。

# 加入新的Function

- 为了满足功能需求，修改代码时还加入了一些自定义函数，这些函数需要继承自 `AbstractFunction`，并重写父类中提供的一些方法，加入自己的功能元素。
- 其中，必须要重写的方法有：
  - `getName()`
  - `call()`



# 自定义函数 XmlToJsonFunction

- 覆盖call方法，调用 JSON.org的 XML.toJSONObject方法将XML转为JSON。
- 覆盖getName方法，将该自定义方法的aviator 调用名设为xmlToJson。

```
public class XmlToJsonFunction extends AbstractFunction {  
  
    @Override  
    public AviatorObject call(Map<String, Object> env, AviatorObject arg1) {  
        if (FakeThreadContext.get()) {  
            arg1.getValue(env);  
            return new AviatorRuntimeJavaType(null);  
        }  
        String xml = (String) arg1.getValue(env);  
        try {  
            return new AviatorRuntimeJavaType(JSON.parseObject(XML.toJSONObject(xml).toString()));  
        } catch (JSONException e) {  
            return new AviatorRuntimeJavaType(null);  
        }  
    }  
  
    @Override  
    public String getName() { return "xmlToJson"; }  
}
```

谢谢！

“

”