Aviator

2.3.0

用户指南 使用手册

Phase-Support, Featured

版本变更

- 2013-05-18 完成 0.6 版本，添加大数和精度计算小节，适配 aviator 2.3.0
- 2010-06-28 完成 0.1 版本
- 2010-09-07 修改完成 0.2 版本
- 2011-07-13 完成 0.3 版本，适配 aviator 2.0
- 2011-09-18 完成 04.版本，适配 aviator 2.1.1
- 2011-12-27 完成 0.5 版本，适配 aviator 2.2.1

简介

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。现在已经有很多开源可用的 java 表达式求值引擎，为什么还需要 Avaitor 呢？

Aviator 的设计目标是**轻量级**和**高性能**，相比于 Groovy、JRuby 的笨重，Aviator 非常小，加上依赖包也才 450K,不算依赖包的话只有 70K；当然，Aviator 的语法是受限的，它不是一门完整的语言，而只是语言的一小部分集合。

其次，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。简单来说，Aviator 的定位是介于 Groovy 这样的重量级脚本语言和 IKExpression 这样的轻量级表达式引擎之间。

特性

Aviator 的特性

- 支持大部分运算操作符，包括算术操作符、关系运算符、逻辑操作符、位运算符、正则匹配操作符(=~)、三元表达式?:，并且支持操作符的优先级和括号强制优先级，具体请看后面的操作符列表。

- 支持函数调用和自定义函数

- 内置支持正则表达式匹配，类似 Ruby、Perl 的匹配语法，并且支持类 Ruby 的$digit 指向匹配分组。
- 自动类型转换，当执行操作的时候，会自动判断操作数类型并做相应转换，无法转换即抛异常。
- 支持传入变量，支持类似 a.b.c 的嵌套变量访问。
- 函数式风格的 seq 库，操作集合和数组
- 性能优秀

Aviator 的限制：

- 没有 if else、do while 等语句，没有赋值语句，仅支持逻辑表达式、算术表达式、三元表达式和正则匹配。
- 不支持八进制数字字面量，仅支持十进制和十六进制数字字面量。

整体结构

Aviator 的结构非常简单，一个典型的求值器的结构

依赖包

commons-beanutils 和 commons-logging

maven 依赖

使用 maven 添加下列依赖即可：

```
<dependency>
  <groupId>com.googlecode.aviator</groupId>
  <artifactId>aviator</artifactId>
  <version>2.3.0</version>
</dependency>
```

使用手册

执行表达式

Aviator 的使用都是集中通过 com.googlecode.aviator.AviatorEvaluator 这个入口类来处理，最简单的例子，执行一个计算 1+2+3 的表达式：

```
import com.googlecode.aviator.AviatorEvaluator;
public class SimpleExample {
    public static void main(String[] args) {
        Long result = (Long) AviatorEvaluator.execute("1+2+3");
        System.out.println(result);
    }
}
```

细心的朋友肯定注意到结果是 Long，而不是 Integer。这是因为 Aviator 的数值类型仅支持 Long 和 Double，任何整数都将转换成 Long，任何浮点数都将转换为 Double，包括用户传入的变量数值。这个例子的打印结果将是正确答案 6。

使用变量

想让 Aviator 对你 say hello 吗？很简单，传入你的名字，让 Aviator 负责字符串的相加：

```
import com.googlecode.aviator.AviatorEvaluator;


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

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

上面的例子演示了怎么向表达式传入变量值，表达式中的 yourname 是一个变量，默认为 null，通过传入 Map<String,Object>的变量绑定环境，将 yourname 设置为你输入的名称。env 的 key 是变量名，value 是变量的值。

上面例子中的'hello '是一个 Aviator 的 String，Aviator 的 String 是任何用单引号或者双引号括起来的字符序列，String 可以比较大小（基于 unicode 顺序），可以参与正则匹配，可以与任何对象相加，任何对象与 String 相加结果为 String。String 中也可以有转义字符，如\n、\\、\' 等。

```
AviatorEvaluator.execute(" 'a\"b' ");   //字符串 a'b
AviatorEvaluator.execute(" \"a\'b\" ");  //字符串 a"b
AviatorEvaluator.execute(" 'hello'+3 ");  //字符串 hello 3
AviatorEvaluator.execute(" 'hello '+ unknow ");  //字符串 hello null
```

exec 方法

Aviator 2.2 开始新增加一个 exec 方法，可以更方便地传入变量并执行，而不需要构造 env 这个 map 了：

```
String myname="dennis";
AviatorEvaluator.exec(" 'hello ' + yourname ", myname);
```

只要在 exec 中按照变量在表达式中的出现顺序传入变量值就可以执行，不需要构建 Map 了。

调用函数

Aviator 支持函数调用，函数调用的风格类似 lua，下面的例子获取字符串的长度：

```
AviatorEvaluator.execute("string.length('hello')");
```

string.length('hello')是一个函数调用，string.length 是一个函数,'hello'是调用的参数。

再用 string.substring 来截取字符串：

```
AviatorEvaluator.execute("string.contains(\"test\",string.substring('hello',1,2))");
```

通过 string.substring('hello',1,2)获取字符串'e'，然后通过函数 string.contains 判断 e 是否在'test'中。可以看到，函数可以嵌套调用。

Aviator 的内置函数列表请看后面。

自定义函数

Aviator 除了内置的函数之外，还允许用户自定义函数，只要实现 com.googlecode.aviator.runtime.type.AviatorFunction 接口，并注册到 AviatorEvaluator 即可使用

AviatorFunction 接口十分庞大，通常来说你并不需要实现所有的方法，只要根据你的方法的参数个数，继承 AbstractFunction 类并 override 相应方法即可。

可以看一个例子，我们实现一个 add 函数来做数值的相加：

```
 import com.googlecode.aviator.AviatorEvaluator;
 import com.googlecode.aviator.runtime.function.AbstractFunction;
 import com.googlecode.aviator.runtime.function.FunctionUtils;
 import com.googlecode.aviator.runtime.type.AviatorDouble;
 import com.googlecode.aviator.runtime.type.AviatorObject;
 public class AddFunction extends AbstractFunction {

       @Override
       public AviatorObject call(Map<String, Object> env, AviatorObject arg1, AviatorObject arg2) {
           Number left = FunctionUtils.getNumberValue(arg1, env);
           Number right = FunctionUtils.getNumberValue(arg2, env);
           return new AviatorDouble(left.doubleValue() + right.doubleValue());
       }


       public String getName() {
           return "add";
       }

   }
```

注册到 AviatorEvaluator 并调用如下：

```
        //注册函数
        AviatorEvaluator.addFunction(new AddFunction());
        System.out.println(AviatorEvaluator.execute("add(1,2)"));
        System.out.println(AviatorEvaluator.execute("add(add(1,2),100)"));
```

注册函数通过 AviatorEvaluator.addFunction 方法，移除可以通过 removeFunction。

编译表达式

上面提到的例子都是直接执行表达式，事实上 Aviator 背后都帮你做了编译并执行的工作。你可以自己先编译表达式，返回一个编译的结果，然后传入不同的 env 来复用编译结果，提高性能，这是更推荐的使用方式：

```
import java.util.HashMap;
import java.util.Map;

import com.googlecode.aviator.AviatorEvaluator;
import com.googlecode.aviator.Expression;

public class CompileExample {
    public static void main(String[] args) {
        String expression = "a-(b-c)>100";
        // 编译表达式
        Expression compiledExp = AviatorEvaluator.compile(expression);

        Map<String, Object> env = new HashMap<String, Object>();
        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);
    }
}
```

通过 compile 方法可以将表达式编译成 Expression 的中间对象，当要执行表达式的时候传入 env 并调用 Expression 的 execute 方法即可。表达式中使用了括号来强制优先级，这个例子还使用了>用于比较数值大小，比较运算符!=、==、>、>=、<、<=不仅可以用于数值，也可以用于 String、Pattern、Boolean 等等，甚至是任何用户传入的两个都实现了 java.lang。Comparable 接口的对象之间。

编译后的结果你可以自己缓存，也可以交给 Aviator 帮你缓存，AviatorEvaluator 内部有一个全局的缓存池，如果你决定缓存编译结果，可以通过：

```
   public static Expression compile(String expression, boolean cached)
```

将 cached 设置为 true 即可，那么下次编译同一个表达式的时候将直接返回上一次编译的结果。使缓存失效通过：

```
public static void invalidateCache(String expression)
```

方法。

访问数组和集合

可以通过中括号去访问数组和 java.util.List 对象，可以通过 map.key 访问 java.util.Map 中 key 对应的 value，一个例子：

```
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.googlecode.aviator.AviatorEvaluator;


public class CollectionExample {
    public static void main(String[] args) {
        final List<String> list = new ArrayList<String>();
        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<String, Date>();
        map.put("date", new Date());

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

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

Aviator 同样支持对多维数组的访问（从 2.2 版本开始）：

```
    int [][] a=......
    AviatorEvaluator.exec("a[0][1]+a[0][0]",a);
```

三元操作符

Aviator 不提供 if else 语句，但是提供了三元操作符?:用于条件判断，使用上与 java 没有什么不同：

```
import java.util.HashMap;
import java.util.Map;

import com.googlecode.aviator.AviatorEvaluator;


public class TernaryOperatorExample {
    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: java TernaryOperatorExample
[number]");
            System.exit(1);
        }
        int num = Integer.parseInt(args[0]);
        Map<String, Object> env = new HashMap<String, Object>();
        env.put("a", num);
        String result = (String) AviatorEvaluator.execute("a>0? 'yes':'no'",
env);
        System.out.println(result);
    }
}
```

这个例子用来判断用户传入的数字是否是正整数，是的话打印 yes。

Aviator 的三元表达式对于两个分支的结果类型并不要求一致，可以是任何类型，这一点与 java 不同。

正则表达式匹配

Aviator 支持类 Ruby 和 Perl 风格的表达式匹配运算，通过=~操作符，如下面这个例子匹配 email 并提取用户名返回：

```
import java.util.HashMap;
import java.util.Map;

import com.googlecode.aviator.AviatorEvaluator;

public class RegularExpressionExample {
    public static void main(String[] args) {
        String email = "killme2008@gmail.com";
        Map<String, Object> env = new HashMap<String, Object>();
        env.put("email", email);
        String username = (String) AviatorEvaluator.execute("email=~/([\\w0-
8]+)@\\w+[\\.\\w+]+/ ? $1:'unknow'", env);
        System.out.println(username);
    }
}
```

email 与正则表达式//([\\w0-8]+@\\w+[\\.\\w+]+)/通过=~操作符来匹配，结果为一个 Boolean 类型，因此可以用于三元表达式判断，匹配成功的时候返回$1，指代正则表达式的分组 1，也就是用户名，否则返回 unknown。这个例子将打印 killme2008 这个用户名。

Aviator 在表达式级别支持正则表达式，通过//括起来的字符序列构成一个正则表达式，正则表达式可以用于匹配（作为=~的右操作数)、比较大小，匹配仅能与字符串进行匹配。匹配成功后，Aviator 会自动将匹配成功的分组放入$num 的变量中，其中$0 指代整个匹配的字符串，而 $1 表示第一个分组，以此类推。

Aviator 的正则表达式规则跟 Java 完全一样，因为内部其实就是使用 java.util.regex.Pattern 做编译的。

变量的语法糖衣

Aviator 有个方便用户使用变量的语法糖衣，当你要访问变量 a 中的某个属性 b，那么你可以通过 a.b 访问到，更进一步，a.b.c 将访问变量 a 的 b 属性中的 c 属性值，推广开来也就是说 Aviator 可以将变量声明为嵌套访问的形式，一个例子，Foo 类有属性 i、f、date：

```
public class Foo {
        int i;
        float f;
        Date date = new Date();
```

```
        public Foo(int i, float f, Date date) {
            super();
            this.i = i;
            this.f = f;
            this.date = date;
        }

        public int getI() {
            return i;
        }

        public void setI(int i) {
            this.i = i;
        }

        public float getF() {
            return f;
        }

        public void setF(float f) {
            this.f = f;
        }

        public Date getDate() {
            return date;
        }

        public void setDate(Date date) {
            this.date = date;
        }
    }
```

Foo 类符合 JavaBean 规范，并且是 public 的，我们执行一个表达式来描述 Foo:

```
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import com.googlecode.aviator.AviatorEvaluator;

public class VariableExample {
    public static void main(String[] args) {
        Foo foo = new Foo(100, 3.14f, new Date());
        Map<String, Object> env = new HashMap<String, Object>();
```

```
        env.put("foo", foo);

        String result =
                (String) AviatorEvaluator.execute(
                    " '[foo i='+ foo.i + ' f='+foo.f+'
year='+(foo.date.year+1900)+ ' month='+foo.date.month +']' ",
                    env);
        System.out.println(result);

    }
}
```

细看下表达式：

```
'[foo i='+ foo.i + ' f='+foo.f+' year='+(foo.date.year+1900)+ '
month='+foo.date.month +']'
```

可以看到我们通过 foo.i 和 foo.f 的方式来访问 foo 变量中的 i 和 f 属性，并且通过 foo.date.year 的方式来访问 foo 中 date 对象的 year 属性（其实是 getYear 方法）。

nil 对象

nil 是 Aviator 内置的常量，类似 java 中的 null，表示空的值。nil 跟 null 不同的在于，在 java 中 null 只能使用在==、!=的比较运算符，而 nil 还可以使用>、>=、<、<=等比较运算符。Aviator 规定，任何对象都比 nil 大除了 nil 本身。用户传入的变量如果为 null，将自动以 nil 替代。

```
        AviatorEvaluator.execute("nil == nil");  //true
        AviatorEvaluator.execute(" 3> nil");    //true
        AviatorEvaluator.execute(" true!= nil");    //true
        AviatorEvaluator.execute(" ' '>nil ");  //true
        AviatorEvaluator.execute(" a==nil ");   //true,a is null
```

nil 与 String 相加的时候，跟 java 一样显示为 null

日期比较

Aviator 并不支持日期类型，如果要比较日期，你需要将日期写字符串的形式，并且要求是形如"yyyy-MM-dd HH:mm:ss:SS"的字符串，否则都将报错。 字符串跟 java.util.Date 比较的时候将自动转换为 Date 对象进行比较：

```
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
import java.util.HashMap;
import java.util.Map;

import com.googlecode.aviator.AviatorEvaluator;


public class DateExample {
    public static void main(String[] args) throws Exception {
        Map<String, Object> env = new HashMap<String, Object>();
        final Date date = new Date();
        String dateStr = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss:SS").format(date);
        env.put("date", date);
        env.put("dateStr", dateStr);

        Boolean result = (Boolean) AviatorEvaluator.execute("date==dateStr",
env);
        System.out.println(result);

        result = (Boolean) AviatorEvaluator.execute("date > '2009-12-20
00:00:00:00' ", env);
        System.out.println(result);

        result = (Boolean) AviatorEvaluator.execute("date < '2200-12-20
00:00:00:00' ", env);
        System.out.println(result);

        result = (Boolean) AviatorEvaluator.execute("date ==date ", env);
        System.out.println(result);
    }

}
```

也就是说 String 除了能跟 String 比较之外，还能跟 nil 和 java.util.Date 对象比较。

大数计算和精度

从 2.3.0 版本开始，aviator 开始支持大数字计算和特定精度的计算，本质上就是支持 `java.math.BigInteger` 和 `java.math.BigDecimal` 两种类型，这两种类型在 aviator 中简称为 big int 和 decimal 类型。

类似“99999999999999999999999999999999”这样的数字在 Java 语言里是没办法编译通过的，因为它超过了 Long 类型的范围，只能用 `BigInteger` 来封装。但是 aviator 通过包装，可以直接支持这种大整数的计算，例如：

```
 System.out.println(AviatorEvaluator.exec("99999999999999999999999999999999+
99999999999999999999999999999999"));
```

结果为类型 big int 的：

```
1999999999999999999999999999999998
```

字面量表示

big int 和 decimal 的表示与其他数字不同，两条规则：

- 以大写字母 N 为后缀的整数都被认为是 big int，如 1N,2N,9999999999999999999N 等，都是 big int 类型。
- 超过 long 范围的整数字面量都将自动转换为 big int 类型。
- 以大写字母 M 为后缀的数字都被认为是 decimal，如 1M,2.222M, 100000.9999M 等等，都是 decimal 类型。

用户也可以通过变量传入这两种类型来参与计算。

ps.熟悉 Clojure 语言的朋友应该会注意到这个表示法是来自该语言。

运算

big int 和 decimal 的运算，跟其他数字类型 long,double 没有什么区别，操作符仍然是一样的。aviator 重载了基本算术操作符来支持这两种新类型：

```
 Object rt = AviatorEvaluator.exec("92233720368547758071OO.356M * 2");
        System.out.println(rt + "  " + rt.getClass());

        rt = AviatorEvaluator.exec("9223372036854775807+1000");
        System.out.println(rt + "  " + rt.getClass());

        BigInteger a = new BigInteger(String.valueOf(Long.MAX_VALUE) +
String.valueOf(Long.MAX_VALUE));
        BigDecimal b = new BigDecimal("3.2");
        BigDecimal c = new BigDecimal("9999.99999");

        rt = AviatorEvaluator.exec("a+100000000000000000", a);
        System.out.println(rt + "  " + rt.getClass());

        rt = AviatorEvaluator.exec("b+c*2", b, c);
        System.out.println(rt + "  " + rt.getClass());
```

```
        rt = AviatorEvaluator.exec("a*b/c", a, b, c);
        System.out.println(rt + "  " + rt.getClass());
```

打印结果为：

```
184467440737095516142 00.712  class java.math.BigDecimal
92233720368547759074  class java.math.BigInteger
922337203685477580892233720368547758 07  class java.math.BigInteger
20003.19998  class java.math.BigDecimal
2.95147905474500731328015521845950 8E+34  class java.math.BigDecimal
```

big int 类型同时支持各种位运算。

math 库的函数也都相应更新来支持 big int 和 decimal 类型。

类型转换和提升

当 big int 或者 decimal 和其他类型的数字做运算的时候，按照 long < big int < decimal < double 的规则做提升，也就是说运算的数字如果类型不一致，结果的类型为两者之间更“高”的类型。

例如:

- 1 + 3N， 结果为 big int 的 4N
- 1 + 3.1M，结果为 decimal 的 4.1M
- 1N + 3.1M，结果为 decimal 的 4.1M
- 1.0 + 3N，结果为 double 的 4.0
- 1.0 + 3.1M，结果为 double 的 4.1

decimal 的计算精度

Java 的 `java.math.BigDecimal` 通过 `java.math.MathContext` 支持特定精度的计算，任何涉及到金额的计算都应该使用 decimal 类型。

默认 Aviator 的计算精度为 `MathContext.DECIMAL128`，你可以自定义精度，通过:

```
AviatorEvaluator.setMathContext(MathContext.DECIMAL64);
```

即可设置，更多关于 decimal 的精度问题请看 `java.math.BigDecimal` 的 javadoc 文档。

强大的 seq 库

aviator 拥有强大的操作集合和数组的 seq 库。整个库风格类似函数式编程中的高阶函数。在 aviator 中，数组以及 java.util.Collection 下的子类都称为 seq，可以直接利用 seq 库进行遍历、过滤和聚合等操作。

例如，假设我有个 list：

```
        Map<String, Object> env = new HashMap<String, Object>();
       ArrayList<Integer> list = new ArrayList<Integer>();
       list.add(3);
       list.add(100);
       list.add(-100);
       env.put("list", list);
```

我可以求长度：

```
   count(list)
```

可以求和：

```
reduce(list,+,0)
```

resuce 函数接收三个参数，第一个是 seq，第二个是聚合的函数，如+、**等，第三个是聚合的初始值。**

还可以过滤：

```
filter(list,seq.gt(0))
```

这个例子过滤出 list 中所有大于 0 的元素并返回集合。seq.gt 函数用于生成一个谓词，表示大于某个值。

我们还可以判断元素在不在集合里：

```
include(list,100)
```

还可以排序:

```
sort(list)
```

最后，我们可以遍历整个集合:

```
map(list,println)
```

map 接受的第二个函数将作用于集合中的每个元素，这里简单地调用 println 打印每个元素。

两种运行模式

默认 AviatorEvaluator 以执行速度优先：

```
AviatorEvaluator.setOptimize(AviatorEvaluator.EVAL);
```

你可以修改为编译速度优先，这样不会做编译优化：

```
AviatorEvaluator.setOptimize(AviatorEvaluator.COMPILE);
```

调试信息

从 2.1.1.版本开始，Aviator 允许设置输出每个表达式生成的字节码，只要设置 trace 为 true 即可：

```
 AviatorEvaluator.setTrace(true);
```

方便用户做跟踪和调试。默认是输出到标准输出，你可以改变输出指向：

```
AviatorEvaluator.setTraceOutputStream(new FileOutputStream(new
File("aviator.log")));
```

语法手册

下面是 Aviator 详细的语法规则定义。

数据类型

- Number 类型：数字类型，支持四种类型，分别是 long,double,`java.math.BigInteger`（简称 big int）和 `java.math.BigDecimal`（简称 decimal），规则如下：

- 任何以大写字母 N 结尾的整数都被认为是 big int；
- 任何以大写字母 M 结尾的数字都被认为是 decimal；
- 其他的任何整数都将被转换为 Long，
- 其他任何浮点数都将被转换为 Double。
- 超过 long 范围的整数字面量都将自动转换为 big int 类型。

其中 big int 和 decimal 是 2.3.0 版本开始引入的。数字还支持十六进制（以 0x 或者 0X 开头的数字），以及科学计数法，如 1e-3 等。不支持其他进制。

- String 类型： 字符串类型，单引号或者双引号括起来的文本串，如'hello world'，变量如果传入的是 String 或者 Character 也将转为 String 类型。

- Bool 类型： 常量 true 和 false，表示真值和假值，与 java 的 Boolean.TRUE 和 Boolean.False 对应。

- Pattern 类型： 类似 Ruby、perl 的正则表达式，以//括起来的字符串，如/\d+/，内部实现为 java.util.Pattern。

- 变量类型： 与 Java 的变量命名规则相同，变量的值由用户传入，如"a"、*"b"*等

- *nil 类型: 常量 nil,类似 java 中的 null，但是 nil 比较特殊，nil 不仅可以参与==、!=的比较，也可以参与>、>=、<、<=的比较，Aviator 规定任何类型都 n 大于 nil 除了 nil 本身，nil==nil 返回 true。用户传入的变量值如果为 null，那么也将作为 nil 处理，nil 打印为 null。*

操作符

算术运算符

*Aviator 支持常见的算术运算符，包括+ - * / % 五个二元运算符，和一元运算符"-"。其中 - * / %和一元的"-"仅能作用于 Number 类型。*

"+"不仅能用于 Number 类型，还可以用于 String 的相加，或者字符串与其他对象的相加。Aviator 规定，任何类型与 String 相加，结果为 String。

逻辑运算符

Avaitor 的支持的逻辑运算符包括，一元否定运算符"!"，以及逻辑与的"&&"，逻辑或的"||"。逻辑运算符的操作数只能为 Boolean。

关系运算符

Aviator 支持的关系运算符包括"<" "<=" ">" ">=" 以及"=="和"!=" 。

&&和||都执行短路规则。

关系运算符可以作用于 Number 之间、String 之间、Pattern 之间、Boolean 之间、变量之间以及其他类型与 nil 之间的关系比较，不同类型除了 nil 之外不能相互比较。

Aviator 规定任何对象都比 nil 大除了 nil 之外。

位运算符

Aviator 支持所有的 Java 位运算符，包括"&" "|" "^" "~" ">>" "<<" ">>>"。

匹配运算符

匹配运算符"=~"用于 String 和 Pattern 的匹配，它的左操作数必须为 String，右操作数必须为 Pattern。匹配成功后，Pattern 的分组将存于变量$num，num 为分组索引。

三元运算符

Aviator 没有提供 if else 语句，但是提供了三元运算符 "?:"，形式为 bool ? exp1: exp2。 其中 bool 必须为结果为 Boolean 类型的表达式，而 exp1 和 exp2 可以为任何合法的 Aviator 表达式，并且不要求 exp1 和 exp2 返回的结果类型一致。

操作符列表

Aviator 支持操作符的优先级，并且允许通过括号来强制优先级，下面是完整的操作符列表，按照优先级从高到低的顺序排列：

序号	操作符	结合性	操作数限制
0	() [ ]	从左到右	()用于函数调用，[ ]用于数组和 java.util.List 的元素访问，要求[indx]中的 index 必须为整型
1	! - ~	从右到左	! 能用于 Boolean,- 仅能用于 Number,~仅能用于整数
2	* / %	从左到右	Number 之间
3	+ -	从左到右	+ - 都能用于 Number 之间, + 还能用于 String 之间，或者 String 和其他对象
4	<< >> >>>	从左到右	仅能用于整数
5	< <= > >=	从左到右	Number 之间、String 之间、Pattern 之间、变量之间、其他类型与 nil 之间
6	== != =~	从左到右	==和!=作用于 Number 之间、String 之间、Pattern 之间、变量之间、其他类型与 nil 之间以及 String 和 java.util.Date 之间，=~仅能作用于 String 和 Pattern 之间
7	&	从左到右	整数之间
8	^	从左到右	整数之间

9	|	从左到右	整数之间
10	&&	从左到右	Boolean 之间，短路
11	||	从左到右	Boolean 之间，短路
12	? :	从右到左	第一个操作数的结果必须为 Boolean，第二和第三操作数结果无限制

类型转换规则

- Java 的 byte short int long 都转化为 Long 类型，Java 的 float,double 都将转化为 Double 类型。Java 的 char String 都将转化为 String。Java 的 null 都将转为 nil。Java 的 `java.math.BigInteger` 和 `java.math.BigDecimal` 保持不变。
- 当两个操作数都是 Double 或者都是 Long 的时候，各自按照 Double 或者 Long 的类型执行，数字运算溢出不会做自动类型提升。
- 当两个操作数中某一个是 Double 的时候，无论另一个操作数字是什么类型都将被转换成 Double，按照 Double 类型执行。
- 当两个操作数分别是 big int 和 long 的时候，结果为 big int。
- 当两个操作数分别为 decimal 和 long，结果为 decimal。
- 当两个操作数分别为 decimal 和 big int，结果为 decimal。
- 简单来说，可以认为 long < big int < decimal < double，当两个操作数类型不同的时候，结果的类型为两个类型中更高的一个。
- 任何类型与 String 相加，结果为 String
- 任何类型都比 nil 大，除了 nil 本身。
- nil 在打印或者与字符串相加的时候，显示为 null
- 形如"yyyy-MM-dd HH:mm:ss:SS"的字符串，在与 java.util.Date 做比较的时候将尝试转换成 java.util.Date 对象比较。
- 没有规定的类型转换操作，除了未知的变量类型之间，都将抛出异常。

常量和变量

true	真值
false	假值
nil	空值
$digit	正则表达式匹配成功后的分组，$0 表示匹配的字符串，$1 表示第一个分组 etc.

内置函数

函数名称	说明
sysdate()	返回当前日期对象 java.util.Date
rand()	返回一个介于 0-1 的随机数，double 类型
print([out],obj)	打印对象，如果指定 out，向 out 打印，否则输出到控制台
println([out],obj)	与 print 类似，但是在输出后换行
now()	返回 System.currentTimeMillis
long(v)	将值的类型转为 long
double(v)	将值的类型转为 double
str(v)	将值的类型转为 string
date_to_string(date,format)	将 Date 对象转化化特定格式的字符串,2.1.1 新增
string_to_date(source,format)	将特定格式的字符串转化为 Date 对象,2.1.1 新增
string.contains(s1,s2)	判断 s1 是否包含 s2，返回 Boolean
string.length(s)	求字符串长度,返回 Long
string.startsWith(s1,s2)	s1 是否以 s2 开始，返回 Boolean
string.endsWith(s1,s2)	s1 是否以 s2 结尾,返回 Boolean
string.substring(s,begin[,end])	截取字符串 s，从 begin 到 end，end 如果忽略的话，将从 begin 到结尾，与 java.util.String.substring 一样。
string.indexOf(s1,s2)	java 中的 s1.indexOf(s2)，求 s2 在 s1 中的起始索引位置，如果不存在为-1

string.split(target,regex,[limit])	Java 里的 String.split 方法一致,2.1.1 新增函数
string.join(seq,seperator)	将集合 seq 里的元素以 seperator 为间隔连接起来形成字符串,2.1.1 新增函数
string.replace_first(s,regex,replacement)	Java 里的 String.replaceFirst 方法，2.1.1 新增
string.replace_all(s,regex,replacement)	Java 里的 String.replaceAll 方法 ，2.1.1 新增
math.abs(d)	求 d 的绝对值
math.sqrt(d)	求 d 的平方根
math.pow(d1,d2)	求 d1 的 d2 次方
math.log(d)	求 d 的自然对数
math.log10(d)	求 d 以 10 为底的对数
math.sin(d)	正弦函数
math.cos(d)	余弦函数
math.tan(d)	正切函数
map(seq,fun)	将函数 fun 作用到集合 seq 每个元素上，返回新元素组成的集合
filter(seq,predicate)	将谓词 predicate 作用在集合的每个元素上，返回谓词为 true 的元素组成的集合
count(seq)	返回集合大小
include(seq,element)	判断 element 是否在集合 seq 中，返回 boolean 值
sort(seq)	排序集合，仅对数组和 List 有效，返回排序后的新集合
reduce(seq,fun,init)	fun 接收两个参数，第一个是集合元素，第二个是累积的函数，本函数用于将 fun 作用在集合每个元素和初始值上面，返回最终的 init 值
seq.eq(value)	返回一个谓词，用来判断传入的参数是否跟 value 相等,用于 filter 函数，如

	filter(seq,seq.eq(3)) 过滤返回等于 3 的元素组成的集合
seq.neq(value)	与 seq.eq 类似，返回判断不等于的谓词
seq.gt(value)	返回判断大于 value 的谓词
seq.ge(value)	返回判断大于等于 value 的谓词
seq.lt(value)	返回判断小于 value 的谓词
seq.le(value)	返回判断小于等于 value 的谓词
seq.nil()	返回判断是否为 nil 的谓词
seq.exists()	返回判断不为 nil 的谓词