Iris Programming Language for v0.0.1.0

1. 介绍

Iris 是一门动态的解释性语言。它具有很多特性，你能够从它身上看到诸如 Java、C#、Ruby、Python、Perl、PHP 等语言的影子，同时它也具有一些自己的特性，这将会在您了解它的具体使用方法过后更加清楚的。Iris 借鉴最多的是 Ruby，同时又简化了一些 Ruby 并不容易使用的、比较花哨的语法，但总体而言，Iris 还是有 Ruby 有很多相似之处的。

开发 Iris 的初衷是想要打造一门易于扩展、能够跨各种平台的脚本语言，以实现对跨平台游戏引擎脚本化的支持。这里谈到的跨平台主要是指 Windows/Windows Mobile、Mac OS/IOS、Android 这些平台。由于目前并没有任何一款脚本语言能够做到这一点，因此 Iris 也就因此诞生了。

同时 Iris 的开发初衷就是作为对某一主要语言的脚本化扩展，因此比如，在 Windows 上的 Iris 能够和 C++进行非常方便的交互并且能够方便地用 C++扩展 Iris，从而把现有的 C++库引入 Iris，让 Iris 驱动——这也是开发 Iris 最终的构想。因此 Iris 是粘性很大的语言，可以说它是依赖于宿主语言而生的。

Iris 是完全面向对象的，这一点和 Ruby 很像。在 Iris 中任何一个组件都是对象，因此 Iris 能够用优雅的面向对象方法解决问题（这甚至是强制性的）。虽然完全面向对象可能会造成一些性能上的问题，但是 Iris 的设计本身就不是以效率为重的，毕竟 Iris 的设计并不如 Lua 那般轻巧；而 Iris 的设计也是以编码的优雅为主的。

Iris 是完全动态的语言，因此对元编程的支持也是和 Ruby 类似的。您甚至可以用流程控制语句来控制一个类是否被定义、您还可以在函数执行的时候动态修改某个类的方法，等等。基本上只要您能够想到的“动态”的特征，Iris 都具备，当然，Iris 的这点灵活性或许会给您带来麻烦——但您完全可以避免这一点。

Iris 是一门小巧精致的语言，它虽然并不具有 C++这样子重量级语言的特性，但是相信在一般的应用场合，Iris 还是能够满足您的需求的。

2. 基本语法

2.1. 表达式（Expression）

表达式是 Iris 的基本语法单位，Iris 的表达式包括字面量、赋值表达式、一元表达式、二元表达式、自运算赋值表达式、索引表达式、方法调用表达式、成员引用表达式、域表达式、数组表达式和 Hash 表达式。

下面进行逐一的介绍：

2.1.1. 字面量

目前版本下的 Iris 拥有 3 种字面量，分别为整数字面量、浮点数字面量以及字符串字面

量，这些字面量分别是 Iris 内置的 Integer、Float、String 类的对象。

其中整数字面量的表达形式为：[+/-]整数值，比如 10、-23、+8 等。

浮点数字面量的表达形式为：[+/-]小数值，比如 1.37、-3.14、0.0 等。

字符串字面量的表达形式为：”字符串内容”，比如”Hello,World!”。

注意由于当前版本的 Iris 并没有实现大数，因此整数的范围是您所在环境下编译 Iris 的 C++编译器一个 int 类型值的范围、浮点数的范围是您所在环境下编译 Iris 的一个 double 类型值的范围。

字面量是写出来了就会被解释器自动解释为对应类的对象的值。

2.1.2. 赋值表达式

赋值表达式是让指定变量有值的表达式。关于 Iris 的变量将会在后面介绍。

赋值表达式的形式为：变量名 = 值。比如 a = 10。

2.1.3. 一元表达式

一元表达式指的是对运算符所接受的运算对象只有一个的表达式。在 Iris 中共有四种一元表达式。

1、 逻辑非：表达形式为 !值，比如 !true;

2、 按位非：表达形式为 ~值，比如 ~1;

3、 取负：表达形式为 –值，比如 –a;

4、 取正：表达形式为 +值，比如 +10;

需要注意的是，在 Iris 中，一元表达式的执行都会被转换为运算符操作对象调用对应的运算符名的方法的形式。因此 Iris 的类是可以重载着一元运算符的。

2.1.4. 二元表达式

二元表达式指的是运算符所接受的运算对象有两个的表达式。在 Iris 中有 21 种二元运算符，其中又分为算术运算符、逻辑运算符、关系运算符、位运算符四类。

1、 算术运算符：+（加）、-（减）、*（乘）、/（除）、**（乘方）、%（取模）;

2、 逻辑运算符：&&（逻辑与）、||（逻辑或）;

3、 关系运算符：>（大于）、>=（大于等于）、<（小于）、<=（小于等于）、==（等于）、!=（不等于）;

4、 位运算符：&（按位与）、|（按位或）、^（按位异或）、<<（算术左移）、>>（算术右移）、<<<（逻辑左移）、>>>（逻辑右移）;

二元表达式的形式是：值 操作符 值。

需要注意的是，在 Iris 中，二元表达式的执行都会被转换为运算符操作的左对象调用对应的运算符名的方法的形式，它具有一个参数，这个参数是右对象。因此 Iris 的类是可以重载二元运算符的。

2.1.5. 自运算赋值表达式

自运算赋值表达式是赋值表达式和二元表达式结合的糖衣语法，设 op 为某一二元运算符，则自运算赋值表达式等价于：

```
;temp = left op right
;left = temp
```

目前 Iris 有 12 种自运算赋值表达式，分别为：+=、-=、*=、/=、%=、&=、|=、^=、<<=、>>=、<<<=、>>>=。

2.1.6. 索引表达式

索引表达式的形式为：值[索引]。任何一个重载了[]运算符类的对象都能够使用索引表达式，索引表达式常用于数组和 Hash 对象，用于依据索引来获取数组对应下标的值或者 Hash 中键对应的值。

比如：

```
;array = [1, 2, 3, 4]
;print(array[2])
```

需要注意的是，在 Iris 中，索引表达式的执行都会被转换为索引运算符操作的主对象调用“[]”方法的形式，它具有一个参数，这个参数是索引对象。因此 Iris 的类是可以重载“[]”运算符的。同时如果是索引表达式的赋值的话，那么调用的就是“[]=”方法了，同样的“[]=”运算符也可以被重载。

2.1.7. 方法调用表达式

方法调用表达式是指对某一个方法进行调用的表达式，方法包括全局方法、类方法和实例方法。

Iris 中任何一个方法都是 Method 类的对象，使用方法调用表达式，事实上会被隐式转换为对这个 Method 类对象的 call()方法的调用——而从理论上讲 call()方法又是对 call()方法这个对象的 call()方法的调用……这样子看下去似乎永无止境了，但请放心，Iris 内部对此做了一些小的调整，因此虽然表面上看起来是这个样子，但实际上并不会无限调用下去。

方法调用表达式的形式为：方法名([参数])。

2.1.8. 成员引用表达式

成员引用表达式是指对某个对象的内部公开成员的引用，这包括对象的公开属性以及对象的公开方法。对于 Iris 类的成员访问权限将会在后面介绍。

成员引用表达式的形式为：对象值.成员公开属性/对象值.成员公开方法()。

2.1.9. 数组表达式

严格来说，Iris 的数组表达式也属于字面量，但是这里特别地分出来进行说明。Iris 的数组是对普通意义上数组的扩充，比如 Iris 的数组没有“界”的概念，您可以任意地访问 Iris 数组的任何一个位置（这个位置上如果不存在任何元素，则数组将返回 nil，同时将数组长度扩展到您访问的这个位置为止，并全部插入 nil）。关于详细的 Iris 数组内容将在后面介绍。

数组表达式也就是字面定义数组的表达式，其形式为：[元素 1,元素 2,…元素 n[,]]。

2.1.10. Hash 表达式

同样，严格来说，Iris 的 Hash 表达式也属于字面量，这里同样特别地分出来进行说明。Iris 的 Hash 由键值对构成，通过键能够获得相应的值，如果不存在则返回 nil。

Hash 表达式也就是字面定义 Hash 的表达式，其形式为：{键 1=>值 1, 键 2=>值 2,…, 键 n=>值 n[,]}。

2.2. 语句（Statement）

Iris 的语句是对表达式的组合，从而形成具有具体含义的语句。目前 Iris 拥有 16 种语句，下面将进行介绍。

2.2.1. 常语句（Normal Statement）

常语句就是对一个表达式的执行，它的格式为：;表达式。

请注意，Iris 一个相当大的特点就是每一条常语句都是分号在前。

例：

```
;a = 10
;b = a + 20
;print(a, “,”,b)
```

2.2.2. if 语句（if Statement）

Iris 的有两种 if 语句，一种是用于控制条件执行的，另一种适用于循环的，分别称为条件 if 和循环 if。下面分别进行介绍。

2.2.2.1. 条件 if

条件 if 有四种表达形式：

1） if(条件) { … }

2） if(条件) { … } else {…}

3） if(条件) { … } elseif(条件 1) {…} elseif(条件 2) {…} … elseif(条件 n)

4） if(条件) { … } elseif(条件 1) {…} elseif(条件 2) {…} … elseif(条件 n) { … } else { … }

条件判断由上至下，当满足其中一个条件的时候（if/elseif），就会执行其后紧随的大括号内的一系列语句——两个大括号及其内部的语句被称为块（Block）。而如果条件都没有满足，同时存在 else 语句的话，那么将会执行 else 后的块。

请注意，Iris 中除了 false 和 nil 以外都被视作真。

例：

```
;a = 1
if(a == 0) {
    ;print(“a = 0”)
}
elseif(a == 1) {
    ;print(“a = 1”)
}
else {
    ;print(“a = other”)
}
```

2.2.2.2. 循环 if

循环 if 是 Iris 特有的循环语法。您可能会发现，Iris 没有 while 语句，是的，Iris 的设计就并不支持 while 语句，原因在于 while 语句的使用频率并不是很高（相对的，for 语句似乎更加受欢迎），因此 Iris 移除了 while 语句，同时加入了更加方便且操作方便的循环 if。

Iris 的 if 循环的基本语法如下：

```
If(循环基础条件, 循环次数 [, 计次变量]){
    // 循环体
}
```

首先，循环基础条件是循环得以进行的基础，Iris 第一次遇到到循环体的时候，将会检测循环基础条件，如果条件为真，那么就开始执行循环体。

而循环次数可以指定一个具体的数字来控制这个循环的执行次数，但是，请注意，如果循环次数没有达到但是循环基础条件已经为假了的话，那么并不会继续执行循环。

计次变量是一个可选的参数，它可以方便地记录下当前循环进行的次数，需要注意的是，计次变量必须是一个局部变量（不能为类变量、实例变量、全局变量等）。

循环的停止条件有如下几个：

1、 遇到 return 语句；

2、 遇到 break 语句；

3、 基础循环条件为假；

4、 循环次数达到指定值；

以上四点任何一点达到，循环都会停止。

例：

```
if(i <= 3, 4, i) {
    ;print(i)
}
```

上面这个例子将会顺序打印 1、2、3，但并不会打印 4，因为当 i=4 的时候，循环基础条件为假，结束循环。

另外，如果您设定的循环次数为小于等于零的数的话，那么 if 循环将会完全视基础循环条件是否成立而进行，因此您可以这样子实现一个无限循环：

```
if(true, 0) {
    // 循环体
}
```

而以下表达和直接使用 if 语句效果相同：

```
if(true, 1) {
    // 条件体
}
```

2.2.3. switch 语句（switch Statement）

Iris 的 switch 语句和 C++的 switch 语句非常类似，但是更加易用，并且在特性上也有所不同。

switch 语句是多路条件选择语句，它可以和条件 if 语句等价互换，但如果选择项较多的话 switch 语句还是比较方便的。

switch 语句的基本语法如下：

```
switch(待比较的表达式) {
    when(表达式 1, 表达式 2,…, 表达式 n) {
        // 语句
    }
    when(表达式 n+1, 表达式 n+2,…, 表达式 m) {
        // 语句
    }
    //…
    [else {
        // 语句
    }]
}
```

switch 语句的执行首先计算待比较的表达式，保存其值，然后对每一个 when 子语句的表达式列表中对每一个表达式进行计算，如果遇到有一个 when 子语句的表达式列表中的某个表达式计算结果和待比较的表达式的值相同的话，那么就执行这条 when 子语句的块。如果全部不满足，且存在 else 子语句的话，那么将执行 else 语句中的内容。

需要注意的是 Iris 的 switch 语句执行完一个 when 或者 else 子语句之后，将会直接结束，而不会像 C++的 switch 语句一样顺序往下继续执行。

例：

```
;c = “Hello”
switch (c) {
    when(1, 2) {
        ;print(“Number”)
    }
```

```
        when(“Hello”) {
            ;print(“String”)
        }
        else {
            ;print(“Other”)
        }
}
```

2.2.4. for 语句（for Statement）

for 语句是对 Ruby 中有范围概念的对象的遍历的一种糖衣循环，它可以用 if 循环替代。
基本语法为：
1、对数组：

```
for(迭代变量 in  数组) {
    //  语句
}
```

例：

```
;array = [1, 2, “array”]
for(I in array){
    ;print(i, “\n”)
}
```

上面这个例子将会顺序打印出数组中的所有元素。

2、对 Hash：

```
for((键,值) in Hash) {
    //  语句
}
```

例：

```
;hash = {1 => ”Kuroneko”, 2=> “Kurumi”, 3=>10}
for((key, value) in hash) {
    ;print(key, “=>”, value)
}
```

上面这个例子将会打印出该 Hash 的所有键值对。

3、对 Range 对象（该特性暂未添加）

```
for(值  in Range 对象) {
    //  语句
}
```

例：

```
for(i in (0 -> 10]) {
    ;print(I, “\n”)
}
```

上面这个例子将会顺序打印出 1-10 的所有整数。

这里提一下 Range 对象，Range 对象是 Iris 中对范围的抽象，比如”a”-”z”,0-9 等都可以被称为范围。Iris 内置了一些范围，当然，您也可以继承 Range 类自定义一些自己的 Range。

Iris 的 Range 是基于数学的区间概念（离散的区间），比如您可以方便地字面指定一个拥有开闭的区间：

(0 -> 100)：1-99 的整数区间；（左开右开区间）

(“a” -> “z”]：”b”-“z”的字符串区间；（左开右闭区间）

[“Z” -> “A”)：”Z”-“B”的字符串区间；（左闭右开区间）

[-100 -> 20]：-100-20 的整数区间；（左闭右闭区间）。

2.2.5. break 语句（break Statement）

break 语句用于从循环中跳出。

注意，break 语句仅能跳出包含该语句的所有循环的最内层的循环。而如果 break 语句出现在非循环语句的块中的话，Iris 将会报错。

例：

```
for(i in [0 -> 10]) {
    if(I == 8) {
        ;break
    } else {
        ;print(i, “\n”)
    }
}
```

上面这个例子将会顺序打印 0-7 的所有整数。

2.2.6. continue 语句（continue Statement）

continue 语句用于立即结束本次循环进入下一次循环。

同样的，continue 语句仅能结束包含该语句的所有循环的最内层的循环。而如果 continue 语句出现在非循环语句的块中的话，Iris 将会报错。

例：

```
for(I in [0->10]) {
    if(I % 2 == 0) {
        ;continue
    }
    ;print(i)
}
```

上面这个例子将会顺序打印 0-10 中所有的奇数。

2.2.7. 方法定义语句（Method-Define Statement）

方法定义语句用于在 Iris 中定义方法，而定义的同时将会产生相应的方法（Method）对象。至于方法定义在何处需要看定义该方法处于的上下文，在 Iris 中定义方法有三种上下文：

全局上下文、类上下文、模块上下文。

全局上下文表示该方法定义在全局，可以直接全局调用。全局定义的方法无论是实例方法还是类方法，如果名称相同那么会视为同一方法，后定义的将会覆盖前定义的。

类上下文表示该方法定义在某个类中，此时实例方法和类方法将会区分对待。

模块上下文和内上下文类似，表示该方法定义在某个模块中，此时实例方法和类方法也会区分对待。

定义实例方法的语句为：

```
fun 方法名([参数 1, 参数 2, … 参数 n] [, *可变参数]) {
    // 方法体
}
```

定义类方法的语句为：

```
fun self.方法名([参数 1, 参数 2, … 参数 n] [, *可变参数]) {
    // 方法体
}
```

例：

```
fun fib(a, b, from, to) {
    ;print(b)
    if(from == to) {
        ;return
    }
    ;return fib(b, a + b, from + 1, to)
}
;print(1)
;fib(1, 1, 1, 10)
```

上面这个例子将会递归地顺序打印斐波那契数列的前 10 项。

类方法的定义将会在介绍类的时候说明。

注意如果定义了可变参数，那么调用方法的时候，属于可变参数的那些实参将会被打包成为一个数组传给定义的那个可变参数形式参数变量。

2.2.8. return 语句（return Statement）

return 语句用于从块中返回值。在 Iris 中 return 除了可以在方法中返回值以外，还可以从块中返回值，关于块是 Iris 中比较高级的概念，它与 Iris 的闭包息息相关，将会放到后面进行说明。

return 将会立即结束当前方法的执行并返回值，如果 return 是在循环中，那么无论循环深度有多大，所有的循环都会被跳出然后返回值。

例：

```
;array = [0, 1, 2, 3, nil, 4, 5, 6]
fun elem_count_before_nil(array) {
    ;count = 0
    for(i in [0, array.size())) {
```

```
            if(array[i] != nil){
                ;count += 1
            }
            else {
                ;return count
            }
        }
    }
    ;print(elem_count_before_nil(array))
```

上面这个例子将会计算数组中 nil 元素之前的所有元素的个数并打印出来。

请注意，return 仅能出现在方法或者块中，如果在其他情况下使用 return 语句那么 Iris 将会报错。

2.2.9. 类定义语句（Class-Define Statement）

类是 Iris 这种面向对象语言中非常重要的组件，Iris 的一切设计都是基于类的，因此可以说没有类，Iris 是不可能正常运作的。

一个类就类似于一件产品的设计图纸，有了这张图纸就可以批量生产出无穷多个实实在在的产品，这一般被称为类的实例或者对象，在 Iris 中一般习惯称其为对象（Object）。

定义一个类非常简单：

```
class 类名 [extends 父类]
            [involves 模块名 1, 模块名 2, …, 模块名 n]
            [joints 接口名 1, 接口名 2, …, 接口名 n] {
    // 类体
}
```

Iris 的继承是单继承，也就是说一个 Iris 类只允许有一个父类——Iris 的设计屏蔽掉了多继承造成的继承混乱问题，如果您没有指定具体的父类，那么默认任何您创建的类的父类都是 Object 类，Object 类是所有类的父类。

这里简单介绍一下 Iris 的继承体系，Iris 底层的几个类分别为 Object、Class、Module、Method、Interface 类，它们的继承关系为 Object 类为其他类的父类，而包括 Object 类在内的所有类的类对象本身又是 Class 类的对象。——这里可能有些不好理解，简单说明一下。在 Iris 中，类本身也是一个对象，称为类类的对象（Object of class Class），下文如果没有做特殊说明，类对象，都是指某个类的类对象。。而且类对象本身是可以被引用的，不过类对象永远被保存在常量中（Iris 中要求类名必须大写字母开头，而以大写字母开头的标识符都是常量），因此保存类对象的容器是不可修改的。

类中允许定义实例方法、类方法、类变量，以及设定访问器等。但是请注意，Iris 类不允许定义类中类，类中类这种类似命名空间的效用在 Iris 中可以用模块来代替。

类中定义的实例方法，可以为类的对象所调用；而类中定义的类方法，则仅能够由类本身调用。请注意，Iris 在这里和 C++的特性是不一样的，在 C++中，类的对象可以调用类的静态成员函数，但是 Iris 中对象是不允许调用类的静态成员函数的。

下面的例子定义了一个分数类用于展示 Iris 的类定义：

```
class Fraction {
    ;gset [@numerator]
```

```
;gset [@denominator]

// 构造方法
fun __format(numerator, denominator) {
    // 约分处理
    ;gcd = Fraction.gcd(numerator, denominator)
    ;@numerator = numerator / gcd
    ;@denominator = denominator / gcd
}

// 显示
fun display() {
    ;print(numerator, "/", denominator)
}

// 重载+方法
fun +(obj) {
    if(obj.instance_of(Fraction) {
        ;gcd = Fraction.gcd(@denominator, obj.denominator)
        ;lcm = Fraction.lcm(@denominator, obj.denominator)
        ;return Fraction.new(
            @numerator * lcm / @denominator
            + obj.numerator * lcm / @denominator, lcm)
    }
}

// 最大公约数
fun self.gcd(a, b) {
    if(a < b) {
        ;t = a
        ;a = b
        ;b = t
    }
    if(t = a%b != 0; 0) {
        ;a = b
        ;b = t
    }
    ;return b
}

// 最小公倍数
fun self.lcm(a, b) {
    ;return a * b / gcd(a, b)
}
```

```
}
;fra1 = Fraction.new(1, 2)
;fra2 = Fraction.new(6, 8)
;(fra1 + fra2).show()
```

这个例子将会输出 5/4。

Iris 一个类的对象通过调用 new 类方法来通过类对象生成。

上面的例子中涉及到了访问器，关于访问器将会在后面介绍。另外，在上面的例子中，还出现了构造方法__format，构造方法是在生成对象之后对象首先会隐式调用的方法，它是 Iris 完成的调用。这里需要说明的是，Iris 只有构造方法而没有析构方法，因为 Iris 拥有垃圾回收机制，因此会在恰当的时刻回收所有托管到 Iris 上的垃圾内存，因此不需要析构方法。

2.2.10. 模块定义语句（Module-Define Statement）

Iris 的模块在 Iris 中起到了命名空间的作用，一个模块能够定义模块中的模块，以及模块中的类，但模块的功能不仅仅是如此，Iris 学习了 Ruby 的经验，模块实现了 mix-in，弥补了因为 Iris 没有采用多继承而造成的一些缺点——模块中可以定义实例方法、类方法和类变量、常量。一旦一个类包括了一个模块，那么当一个类的对象调用实例方法的时候，那么 Iris 还会在这个模块中搜索是否有这个方法——类变量和类方法同理。

模块需要被包括才能生效，而类和模块都能包括模块，包括的关键字是 involve。定义一个模块非常简单：

```
module 模块名 [involves 模块名 1, 模块名 2, …, 模块名 n] {
    // 模块体
}
```

下面定义了一个简单的 Math 模块。

```
module Math {
    fun self.sqrt(n) {
        // 平方根倒数速算法
        ;threehalfs = 1.5
        ;x2 = n * 0.5
        ;y = n
        ;i = y
        ;i = 0x5f3759df – (i >> 1);
        ;y = i.to_float()
        ;y = y * (threehalfs – x2 * y * y)
        ;return 1.0 / y
    }
}
;print(Math.sqrt(2))
```

这个例子将会返回 2 的平方根并打印。

2.2.11. 接口定义语句（Interface-Define Statement）

Iris 的接口和静态预言的接口不同，Iris 的接口仅仅作为一种规范的工具存在的。事实上

您完全可以不使用接口，但如果您需要一种类扩展时候的规范的话，那么接口也是一种比较好的选择。

Iris 的接口的功能非常单纯：一个类如果接入了一个接口的话，那么它必须实现接口中声明了的实例方法——并且实现的实例方法不仅要同名，参数个数和类型也要一样（这里的类型一样指的是如果有可变参数那么在实现里面也必须要有可变参数）。

接口中不定义具体方法，它只是一种对类定义的约束手段，以满足“某一大类必须要实现某某方法以保证统一调用”这一要求。另外接口是可以继承的，而且接口是多继承。

定义一个接口：

```
interface 接口名 [joints 接口名 1, 接口名 2, …, 接口名 n] {
    // 接口体
}
```

关于接口中的方法的声明在下一节中介绍。

2.2.12. 接口方法声明语句（Interface Function Declare Statement）

接口方法声明语句用于在接口中声明形式上的方法，类似于 C 语言中的函数原型。但接口中是不允许定义具体的方法的，前面已经说过了，接口就是一种约束规范。

接口方法声明语句的格式为：

```
;fun 方法名([参数 1, 参数 2, …, 参数 n] [, *可变参数])
```

下面定义一个简单的接口来展示简单的插件系统的一个模拟：

```
interface Plugin {
    ;fun install(env)
    ;fun run(env)
    ;fun uninstall(env)
}

class MyPlugin joints Plugin {
    fun install(env) {
        ;print(env, “, “, “installed!”)
    }

    fun run(env) {
        ;print(env, “, “, “run!”)
    }

    fun uninstall(env) {
        ;print(env, “, “, “uninstalled!”)
    }
}
```

2.2.13. Groan 语句（Groan Statement）

Iris 具有异常处理机制，异常处理是替代传统错误处理的比较先进的方式。Iris 的异常处

理机制是这样的：Iris 在执行的时候抛出一个异常，那么解释器将会带着这个异常往上回溯，一直到顶层为止，如果遇到异常处理语句，那么 Iris 将这个异常交付给异常处理语句进行相应的处理，否则如果到顶层环境 Iris 将会报错。

Groan 语句用于抛出一个异常对象，这个异常对象可以是任何 Iris 的对象。调用方法如下：

;groan(表达式)

表达式产生一个对象，Iris 解释器将会停止接下来的语句的运行，而带着这个对象沿着调用链往上走，直至遇到一个异常处理语句为止，并将这个对象交给异常处理语句的 serve 子语句；如果一直到调用链顶层（即 Main 环境）都没有遇到异常处理语句，那么 Iris 将会报错。

2.2.14. 异常处理语句（Exception Handling Statement）

Iris 的异常处理语句用于处理由 Groan 语句产生的异常对象，并截获这个对象进行指定的处理。异常处理语句的结构如下：

```
order {
    // 可能出现异常的语句
}
serve(接受异常对象的变量) {
    // 异常处理
}
[ignore {
    // 无论是否有异常都必须做的处理
}]
```

下面结合 Groan 语句和异常处理语句处理一个只能够接受 Ingeter 参数的方法。

```
fun print_integer(i) {
    if(!i.instance_of(Integer)) {
        ;groan("Error parameter : it must be an integer !\n")
    }
    ;print("Integer is ", I, "\n")
}

order {
    ;print_integer("Hello, Iris!")
}
serve(e) {
    ;print(e)
}
ignore {
    ;print("must do.")
}
```

上述语句将会打印出：

Error parameter : it must be an integer !
must do.

2.2.15. 访问器定义语句（Accessor-Define Statement）

访问器定义语句用于公开化 Iris 类的实例变量，使其能够通过对象在外部被访问、设置，因为 Iris 的实例变量是强制私有的。这是类似 C#的 get/set 语句的语法。

Iris 的访问器定义语句有三种：get、set、gset，其中 get/set 语句又有默认和自定义两种。默认的访问器语句，是 Iris 将会自动为您定义一般行为的访问器，而自定义的访问器将会运行您所自定义的访问器逻辑。gset 语句将会为您设定默认的 get 和 set 访问器。

get 访问器：
默认：;get [实例变量] ()
自定义：get [实例变量] () { // 您自己的逻辑 }

set 访问器：
默认：;set [实例变量](临时变量)
自定义：set [实例变量](临时变量) { //您自己的逻辑 }

gset 访问器只有默认形式：
;gset [实例变量](临时变量)

访问器定义语句的实质，是 Iris 自动地在类内部定义“__get_+实例变量名”和“__set_+实例变量名”的方法，而用户在调用成员访问表达式的时候，Iris 将会调用这两个方法。

下面是一个例子，类 A 的对象可以通过成员访问表达式来访问到它的实例变量@interger，并且只能够为@interger 赋 Interger 类型的值，否则会抛出异常。

```
class A {
    ;get [@integer] ()
    set [@interger] (value) {
        if(!value.instance_of(Integer) {
            ;groan(“Error parameter : it must be an Integer !”)
        }
        ;@integer = value
    }

    fun __format() {
        ;@interger = 0
    }
}

;obj = A.new()
;print(obj.integer, “\n”)
;obj.interger = 10
```

```
;print(obj.interger, “\n”)
order {
    ;obj.integer = “Hello, Iris!”
}
serve(e) {
    ;print(e)
}
```

上面的例子会打印

```
0
10
Error parameter : it must be an Integer !
```

2.2.16. 方法权限定义语句（Method Authority Statement）

方法权限定义语句用于定义类方法、实例方法的访问权限，请注意，在 Iris 中权限定义仅仅是对于方法而言的，因为 Iris 不同于 C++没有成员变量的说法。

方法权限定义的关键字有三个：everyone/native/personal，分别对应公开/继承/私有方法，与 C++中的 public/protected/private 类似。

定义权限的语句格式为：

;everyone/native/personal [方法名]

其中 everyone 为方法定义了这样的访问行为：对于实例方法，类内部实例方法、对象、子类实例方法内部都能够调用该方法；对于类方法，类内部类方法、类对象、子类类方法内部都能进行调用。

native 为方法定义了这样的访问行为：对于实例方法，类内部实例方法、子类实例方法内部都能够调用该方法；对于类方法，类内部类方法、子类类方法内部都能进行调用。

personal 为方法定义了这样的访问行为：对于实例方法，仅类内部实例方法可以调用；对于类方法，仅类内部类方法可以调用。

请注意，必须在定义了方法之后才能定义其权限，否则 Iris 将因为找不到指定名称的方法而报错。

下面是一个例子，展示了方法的权限定义：

```
class A {
    fun __personal_method() {
        ;print(“Personal Method!”, “\n”)
    }

    fun _native_method() {
        ;print(“Native Method!” , “\n”)
    }

    fun everyone_method() {
        ;print(“Everyone Method!” , “\n”)
```

```
        }

        ;personal [__personal_method]
        ;native [_native_method]
        ;everyone [everyone_method]
    }

    class B extends A {
        fun call_method() {
            ;_native_method()
            ;everyone_method()
        }
    }

    ;obj= B.new()
    ;obj.call_method()
```

这个例子的结果将会打印出

```
Native Method!
Everyone Method!
```

而如果您使用 obj 对象调用__personal_method()或者_native_method()的时候，Iris 将会报错。

2.2.17. block 语句和 cast 语句（block Statement and cast Statement）

block 语句和 cast 语句是配合使用的语句，其中 block 用于判断调用方法的时候是否存在隐式传入的 Block，如果有进入 with 语句块否则进入 without 语句块；而 cast 则是在 with 语句块中真正调用 block 的语句。

以下例子根据是否有 Block 产生两种行为：

```
fun call_fun() {
    ;print(“Before”, “\n”)
    ;block
    ;print(“After”, “\n”)
}
with {
    ;cast(“With Block”)
}
without {
    ;print(“Without Block”, “\n”)
}
```

```
;call_fun() {iterator => [msg] : ;print(msg, “\n”)}
;call_fun()
```

上面的例子将会分别打印出：

```
Before
With Block
After
Before
Without Block
After
```

关于 Block 的用法将会在后面进行介绍。

2.2.18. super 语句（super Statement）

super 语句用于子类的实例方法调用同名的父类实例方法，如果父类不存在这个方法那么将会报错。值得一提的是，由于 Iris 的语言性质，所以子类在覆盖父类方法的时候，只需要子类方法的名字和父类方法相同即可，并不要求参数个数要相同。

super 语句的格式为：

```
;super(参数列表)
```

注意，这里的参数是传给父类的同名方法的。同时 Iris 中 super 关键字只能用于实例方法的调用中。

以下例子展示了 super 语句的用法。

```
class A {
	fun hello(msg) {
		;print(msg, “from class A !”, ”\n”)
	}
}

class B {
	fun hello(msg) {
		;super(msg)
		;print(msg, “from class B !” , ”\n”)
	}
}

;obj = B.new()
;obj.hello(“Hello”)
```

以上例子将会打印如下信息：

```
Hello from class A!
```

Hello from class B!

3. 高级用法

3.1. Block 闭包

3.2. 元编程