

DOCUMENT
OF
Stair JavaScript
Another Java Script Interpreter

THIS
IS
MORE THAN A TOY

BY
李其迈 海杰文 陈广翔 蔡武威

目录

1 概述	3
2 如何使用StairJS	3
2.1 第一个程序Hello World	3
2.2 两种模式	4
2.2.1 交互式模式	4
2.2.2 文件解释模式	4
3 StairJS解释流程	5
3.1 语法解析	5
3.2 解释执行	6
4 基本类型	6
5 运算符和表达式	6
6 语句	7
6.1 Empty	7
6.2 Block	7
6.3 Variable Declaration	7
6.4 Variable Declaration	8
6.5 Do-While	8
6.6 While	8
6.7 For	8
6.8 For-Each	8
6.9 Return	8
6.10 Print	8
7 函数	9
7.1 函数定义	9
7.2 函数调用	10
7.3 高阶函数	10
8 对象	10
8.1 获得对象实例	11
8.1.1 Object Literal	11

8.1.2	Array Literal	11
8.1.3	”构造”函数	11
8.2	成员	12
8.2.1	访问成员	12
8.2.2	添加成员	12
8.3	成员函数	13
8.3.1	this指向哪里?	13
8.3.2	“构造”函数只是普通函数	13
9	BNF	13

1 概述

StairJS是用python3实现的一个较为完善的JavaScript解释器，支持了JavaScript大部分基本特性。其所能够解释的语言是JavaScript的一个子集。StairJS实现了以下特性：

1. 两种模式：文件解释模式、交互式模式
2. 支持JavaScript中所有的基本类型
3. 几乎所有运算符
4. 13级运算优先级
5. 大部分语句
6. 函数的定义、调用、递归以及成员函数和高阶函数
7. 面向对象
8. 垃圾回收机制
9. 注释
10. 错误提示信息

2 如何使用StairJS

2.1 第一个程序Hello World

运行StairJS，你需要：

1. Python3的解释器
2. PLY库

因为 StairJS 是用 Python3 开发的，所以首先确保你有一个Python3的解释器。StairJS 在 Python3.4.3下做过完整的测试，保证可以运行。Python3 版本之间差异不大，应该都可以成功运行 StairJS 。有了 python 解释器后，还需要安装 PLY-3.8 这个 python 的库。访问 PLY-3.8 的官网 <http://www.dabeaz.com/ply/>，下载并安装后即可运行我们的 StairJS 。

不给任何参数运行 StairJS.py，会进入交互式模式，并显示版本号。

```
root@desktop: python StairJS.py
Stair JavaScript Interpreter V1.0
Powered by Li Qimai, Hai Jiewen, Chen Guangxiang, Cai Wuwei
>>>
```

">>>"是输入提示符，表示解释器正在等待输入代码。让我们输入`print 'Hello World'`，并按enter键。此时显示"...”，提示等待后续输入，我们直接回车输

入一个空行，就可以看到程序显示了 `Hello World`。用户可以一次输入多行，StairJS会一直读入代码，直到遇到空行为止，之后将会开始解释执行刚才的输入。`print` 语句¹是我们额外添加的一个关键字，提供一种输出的手段。程序执行结果如下：

```
>>>print 'Hello World'
...
Hello World
>>>
```

之后输入`exit`退出解释器，回到终端。

```
>>>exit
Bye!
root@desktop:
```

完整的执行过程如下：

```
root@desktop: python StairJS.py
Stair JavaScript Interpreter V1.0
Powered by Li Qimai, Hai Jiewen, Chen Guangxiang, Cai Wuwei
>>>print 'Hello World'
...
Hello World
>>>exit
Bye!
root@desktop:
```

2.2 两种模式

2.2.1 交互式模式

命令：

```
python StairJS.py
```

如果不给任何命令行参数，直接运行解释器，即可进入交互式界面。在交互式界面，由用户输入一串以空行结尾的 JavaScript 代码，之后解释器解释这段代码。显示"`>>>`"表示正在等待一段全新的代码的输入。显示"`...`"表示正在等待后续输入。

如果本次输入是一个表达式，则执行完毕后解释器会将表达式的计算结果显示出来。如果本次输入不是表达式，或者是多条表达式，则不会显示结果。

2.2.2 文件解释模式

命令：

```
python StairJS.py [-i] <JavaScriptSource> <arg>*
```

其中 `<JavaScriptSource>` 是一个 JavaScript 的源代码文件。`arg` 是要传入 JavaScript 程序的命令行参数。从源代码文件名开始的所有参数将按顺序放入全局变

¹详见 §6 中的 `print` 语句。

量`args`中。在JavaScript程序中可以通过`args`访问到这些命令行参数。
编写 JavaScript 程序如下，存入文件`PrintArgs.js`中

```
/*PrintArgs.js 打印出所有的命令行参数*/  
for(var i in args){  
    print i + ":" + args[i]  
}
```

解释执行`PrintArgs.js`，并传入参数`arg1`，`arg2`，`arg3`，结果如下：

```
root@desktop: python StairJS.py PrintArgs.js arg1 arg2 arg3  
PrintArgs.js  
arg1  
arg2  
arg3
```

可以在源代码文件前给参数 `-i`，则解释器在解释完源代码文件后，会进入交互式界面。此时仍然可以访问文件中定义的变量和各种函数。命令格式如下：

```
python StairJS.py -i HelloWorld.js
```

3 StairJS解释流程

StairJS 的整个运作过程可以分为两个阶段——语法解析和解释执行。

3.1 语法解析

除了额外添加的 `print` 关键字，StairJS 所能够解释执行的语言是 JavaScript 的一个子集。这个子集由§9中的 BNF 精确定义。每次拿到JavaScript的源码后，StairJS 会根据 BNF 来做词法分析和语法分析，并建立一棵 AST(Abstract Syntax Tree)。树中每个叶子节点对应一个终结符，非叶子节点对应于一个非终结符²。AST结构化的描述了输入的JavaScript程序。

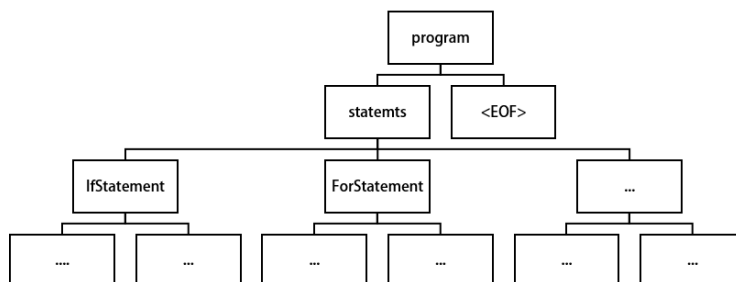


图 1: Abstract Syntax Tree示意图

²这棵树完全符合 Context Free Grammer 的 Parsing Tree 的规范。详细结构见一般的计算理论教材

3.2 解释执行

得到 AST 后, 解释器将会遍历 AST 进行解释执行的工作。几乎每个类型的节点都有对应的解释执行该节点的函数。解释某个节点时, 只需调用对应函数解释所有的子节点, 然后将所有子节点的解释结果综合处理, 即可完成该节点的解释工作。整个解释执行的过程其实是一个递归遍历的过程, 步骤清晰明了。

4 基本类型

StairJS支持JavaScript的所有基本类型Number, String, Boolean, Object, Function, Null, Undefined。因为解释器是用Python开发的, 所以JavaScript的变量最终都要映射成为Python中的变量关系。

JavaScript	Python
Number	int,float
String	str
Boolean	bool
Object	StObject 继承自 dict
Function	StFunction 继承自 StObject
Null	StNULL
Undifined	StUndifined

表 1: JavaScript变量与Python变量的映射

其中 StObject³是我们定义的一个类, 用以表示 JavaScript 中的 Object。StFunction 是用来表示 JavaScript 中函数的类。因为在JavaScript中, 函数也是对象, 所以 StFunction 继承自 StObject。

而StNULL和StUndifined虽然是一个类, StNULL和StUndifined在整个解释器中都只有一个实例。

5 运算符和表达式

StairJS 支持除了三目运算符、delete、instanceof以及前缀的++、--以外所有的运算符。并且按照 JavaScript 的运算优先级, 将所有的运算符分为了14级。具体优先级参见下一页的Table-2。

其中在StairJS中==的效果与===一样。

在"+运算时, 如果有一个操作数是字符串, 则会自动将另外一转换成字符串, 之后做字符串连接。

函数定义也将被视为一个表达式, 可以参与函数调用和取成员、void typeof以及相等比较等运算。

³St 是 StairJS 中 Stair 的简写

优先级	类型	运算符	结合性
1	函数调用和取成员	() [] .	从左到右
2	后缀运算符	++ --	从左到右
3	前缀运算符	void typeof + - ~ !	从右到左
4	乘法类型运算符	* / %	从左到右
5	加法类型运算符	+ -	从左到右
6	移位运算符	<< >> >>>	从左到右
7	关系运算符	< > <= >=	从左到右
8	等于运算符	== != === !==	从左到右
9	按位与	&	从左到右
10	按位异或	^	从左到右
11	按位或		从左到右
12	逻辑与	&&	从左到右
13	逻辑或		从左到右
14	赋值	= *= /= %= += -= <<= >>= >>>= &= ^= =	从右到左

表 2: 运算符优先级

6 语句

支持的语句有: Empty, Block, Variable Declaration, If, Do-While, While, For, For-Each, Return, Print

6.1 Empty

格式:

```
","
```

Empty语句为空语句,仅由一个“,”组成,什么也不做。有这条语句存在,语句结束可以有多个“,”,而不影响程序执行。

6.2 Block

格式:

```
{ Statement* }
```

Block中可以有零条或多条语句。Block的作用在于可以把多条语句结合成一条语句。

6.3 Variable Declaration

格式:

```
"var" <Identifier> ("=" Expression)? ";"?
```

变量声明,同时可以用一个表达式来初始化这个变量。

6.4 Variable Declaration

格式:

```
"if" "(" Expression ")" Statement ("else" Statement)? ";"?
```

if语句，根据Expression的返回值执行不同的语句。如果需要多条语句放在if内，请使用Block语句。

6.5 Do-While

格式:

```
"do" Statement "while" "(" Expression ")" ";"?
```

重复Statement，直到Expression值为false。

6.6 While

格式:

```
"while" "(" Expression ")" Statement
```

重复Statement，直到Expression值为false。

6.7 For

格式:

```
"for" "(" Expression ";" Expression ";" Expression ";" ")" Statement
```

- 1)第一个Expression在开始时会被执行一次。
- 2)执行中间的Expression，若返回false则语句执行结束。返回true则执行Statement，然后执行第三个Expression，之后重复2)

6.8 For-Each

格式:

```
"for" "(" "var" <Identifier> "in" Expression ")" Statement
```

对Expression返回值的所有key值依次赋给变量Identifier，并执行Statement。

6.9 Return

格式:

```
"return" ( Expression )? ";"?
```

将Expression的值作为返回值从函数中返回。如果没有 Expression，则返回undefined。

6.10 Print

格式:

```
"print" Expression ";"?
```

将Expression的值打印出来。

7 函数

在JavaScript中，函数具有和其它变量相同的地位。函数也是一等公民。可以被赋值，可以被传入另外一个函数，也可以从另外一个函数中传回。所以我们定义了一个继承自 StObject⁴的类StFunction。

```
class StFunction(StObject):
    def __init__(self):
        super(StFunction, self).__init__()
        self.name = ""
        self.ast = None # code
        self.argument_list = []
        self.outFunction = None
```

name 是函数定义时给的名字，在将函数转化为字符串时会使用到。

ast 指向该函数的抽象语法树，在调用该函数时会遍历这棵树。

argument_list 函数的参数名字表，传参时需要使用。

outFunction 定义该函数的函数的Active Record。

7.1 函数定义

函数定义是动态进行的，也就是说，只有当解释器解释到了函数的定义，才会生成这个函数，而未解释到的函数是不存在的。而对于函数内的函数定义,如:

```
function f(){
    function g(a,b){}
    ...
}
```

每次调用函数f，函数g都会被重新定义一遍。这就是函数的动态定义。

每次定义函数时，都会生成一个StFunction的实例。之后初始化StFunction的成员变量。比如上面的函数g，其**name**="g"，**ast** 指向g对应的抽象语法树，**argument_list**=["a","b"]即g的参数名称，**outFunction**指向此时f的Active Record。

⁴StObject是表达JavaScript中Object的类

7.2 函数调用

函数执行之前需要为函数分配一个空间存放它的本地变量。这个存放函数本地变量的空间叫做Active Record(AR)。我们定义类StActiveRecord来表达Active Record。

```
class StActiveRecord(dict):
    def __init__(self):
        super(StActiveRecord, self).__init__()
        self.return_value = None
        self.this = None
        self.outFunction = None
```

每次调用函数，就会实例化一个StActiveRecord。并根据 StFunction 的 argument_list 将参数填入 StActiveRecord 中。outFunction 和 this 同样会被赋值。outFunction 的详细介绍详见7.3高阶函数。this的赋值规则见8.3成员函数。完成 StActiveRecord 的初始化后，解释器会切换上下文到这个函数的AR，然后遍历这个函数的抽象语法树，解释执行这个函数。

7.3 高阶函数

高阶函数高阶函数的实现有两个方面，一是函数可以作为参数传入另外一个函数，二是函数可以作为函数返回值，且返回的函数可以访问到定义它的函数的本地变量。接收函数或返回函数的函数，因为它将另外一个函数视为一般的变量，所以被称为高阶函数(High-Order Function)。

我们的 StFunction 是继承自 StObject 的，所以这就保证了它可以被当作参数，也可以被当作函数返回值。

outFunction为了让返回的函数可以访问到定义它的函数的本地变量，我们在 StFunction 和 StActiveRecord 中加入了 outFunction 字段。在函数定义时，会把当前正在执行的这个函数的AR赋给 StFunctiond 的 outFunction。函数调用时，又将StFunction 的 outFunction 赋给 StActiveRecord 的 outFunction。同一个 StFunction 的实例在多次调用时，outFunction 都指向同一个 AR，所以访问的高阶函数的本地变量是相同的。全局变量也是放在一个AR中的。所以定义在全局的函数的 outFunction 就指向这个全局的AR，也就可以访问全局变量了。我们并没有为全局变量做特殊处理，这符合正交的原则。而全局的AR的 outFunction 值为 None。

寻找本地变量用到某一个本地变量时，会先在当前函数的AR里寻找，找不到的话，就在outFunction指向的AR中寻找，并且递归的找下去。这样就实现了函数闭包。

8 对象

我们对面向对象的支持主要体现在this这个关键字上。我们并没有实现基于原型的继承。下面讲解一下StairJS支持哪些与对象有关的语法。

8.1 获得对象实例

8.1.1 Object Literal

最简单的获得一个对象实例的方法莫过于使用写 JSON 风格的 Object Literal。

```
var a = {
  1   : "adf",
  i   : 4*6,
  "f" : function (){
            return i
        },
  3.4 : true
}
```

JavaScript中的所有对象都是一个字典，对象中存放多条 key-value 项。其中 number、string 和 identifier 可以做 key，任意表达式⁵都可以做 value。JavaScript不区分整数和浮点数，它们都是 number 类型。所以浮点数也可以做 key。

8.1.2 Array Literal

除了 Object Literal，还可以直接写Array Literal。

```
var a = ['zero','one',,, 'four']
}
```

Array Literal 实质上也是个字典，并不是数组。只不过这个字典所有的key值都是number类型，而且是整数的。上面的写法和下面的写法等价：

```
var a = {
  0 : 'zero',
  1 : 'one',
  2 : void 0,    \\void 0的值为Undefined
  3 : void 0,
  4 : 'four'
}
}
```

在Array Literal中，两个逗号之间可以不指定值，此时默认赋为Undefined。

8.1.3 “构造”函数

JavaScript可以通过“构造”函数，制造对象。

⁵函数定义是表达式，表达式返回值为函数本身。所以value可以直接是一个函数。

```
function A(){
    this.i = 3
}
var a = new A()
}
```

这就制造出来了一个“A类”对象a。a有一个成员i值为3。其实这个貌似是”构造”函数的函数，并不具有C++/Java里构造函数的特殊地位。详见§8.3成员函数。

8.2 成员

8.2.1 访问成员

通过运算符.和[]可以访问一个成员。不同的是"."后面只可以跟 Identifier, 而[]中则必须是number或string。

```
var a = {
    i    : 3
    "j" : 5
}
var i = "j"
a.i    //值为3
a["j"] //值为5
a[i]   //因为i的值为"j", 所以和a["j"]等价
```

注意a[i]访问到的值由i的值决定，不一定是a.i。因为i的值为"j", a[i]所以和a["j"]等价。

8.2.2 添加成员

JavaScript中随时可以通过赋值语句为对象添加成员。

```
var a = {} //获得一个空的object
a.i = 3
a[4] = "four"
a.f = function () {return i}
}
```

对成员赋值时，如果成员不存在，则会在对象中添加该成员。这就可以很方便的在对象中添加成员变量和成员函数。

8.3 成员函数

8.3.1 this指向哪里？

函数的this究竟指向哪里？在StairJS中，一共有三种情况。

1. new var a = new A()

此时会实例化一个空的 Object，并令A中的 this 指向这个 Object。

2. 作为成员函数 a.f()

此时f被视为a的成员函数，毫无疑问f的 this 指向a。

3. 其它情况 如f()这样直接调用。

那f的 this 应该和此时 AR 的 this 相同。假如当前 AR 的 this 指向变量 a，也就是现在的函数被视为 a 的成员函数，那么不指明时 f 也应该默认被视为 a 的成员函数。

8.3.2 “构造”函数只是普通函数

注意到我们在定义一个函数时，并不需要加入任何特殊的关键字，就可以作为new时需要的函数。也就是说，任何函数都可以被视作构造函数。完全可以写出如下的代码：

```
var a = {  
  f : function () {  
    this.i = 3  
  }  
}  
b = new a.f() //成员函数作为构造函数
```

f这里明显是a的成员函数，但是也可以用来构造对象b。其实new语句，不过是制造出一个空对象，然后令函数f的this指向a，之后运行一遍这个函数而已。上述代码中的b = new a.f()和下面的等价：

```
b = {}  
b.f = a.f  
b.f()  
delete b.f
```

这充分说明了构造函数不具有特殊地位。

9 BNF

本解释器支持的语言是 JavaScript 的一个子集。精确定义该语言的BNF见<http://github.com/Nebula1084/StairJS> 中的 *BNF for StairScript.html*。本文档同目录下也附有一份副本。