|  |
| --- |
|  |
| CoffeeScript编译器文档 |
|  |
|  |
| 王思伦 江林楠 洪宇 |
| 2014/6/9 |

|  |
| --- |
| 本文档是《计算机与网络体系结构（2）》课程编译部分的大作业文档。 |

**目录**

[综述 2](#_Toc390176836)

[编译器支持的语言特性 2](#_Toc390176837)

[编译器不支持的语言特性 4](#_Toc390176838)

[编译器实现 4](#_Toc390176839)

[1、 对缩进的预处理 4](#_Toc390176840)

[（1）预处理原因 4](#_Toc390176841)

[（2）具体实现 5](#_Toc390176842)

[2、 词法分析 5](#_Toc390176843)

[3、 语法和语义分析 5](#_Toc390176844)

[4、 后期处理 7](#_Toc390176845)

# 综述

CoffeeScript是一门小巧的语言，它构建于JavaScript，并改善了JavaScript的不少语法，使之更简洁、高效。通过CoffeeScript编译器，CoffeeScript程序可以直接转化成JavaScript程序并在浏览器中执行。关于这门语言的更多介绍，可参看官方网站：<http://coffeescript.org/>。

我们实现了从CoffeeScript到JavaScript的编译器，支持部分CoffeeScript语法，其中既含有这门语言引入的新特性，也包括它与JavaScript相同的部分。通过该编译器，用户可以使用CoffeeScript进行简单的编程工作。

您可以打开example文件夹下的index.html来尝试用CoffeeScript进行编程。该网页提供了二分查找和快速排序这两个示例程序。有关编译器的环境配置，请参看配置文档。

# 编译器支持的语言特性

|  |  |  |  |
| --- | --- | --- | --- |
| 特性描述 | CoffeeScript代码 | JavaScript代码 | 备注 |
| 变量赋值与自动声明 | a = 1 | var a;  a = 1; | 如果一个变量未定义，在对其进行赋值时会自动声明。 |
| 函数定义1  return语句 | (a, b)->  return a + b | function(a, b) {  return a + b;  }; | 函数内所有语句请用1个Tab来缩进。 |
| 函数定义2 | ->  return false | function() {  return false;  }; | 同上 |
| 对象定义1 | {a:1, b:2} | {a : 1, b : 2} |  |
| 对象定义2 | x =  a:1,  b:2 | var x;  x = {a : 1, b : 2}; | “X=”这行需空出，后边两行需要1个Tab来缩进。 |
| 对象成员引用 | a.b = 1 | a.b = 1; |  |
| 函数调用 | a = fun(1);  b = Math.random(2); | var a;  a = fun(1);  var b;  b = Math.random(2); | 可以递归调用 |
| 数字、布尔值、字符串等支持 | 1  false  undefined  null  "hello world" | 1;  false;  undefined;  null;  "hello world"; | 可以使用yes、no  yes = true  no = false |
| 表达式四则运算 | a = fun1(1) + fun2(2) \* fun3(false) | var a;  a = fun1(1) + fun2(2) \* fun3(false); | 不支持四则运算的括号 |
| 数组定义与使用 | Arr = [1, "3", false];  Arr[0] = 1; | var Arr;  Arr = [1, "3", false];  Arr[0] = 1; | 支持a['b']形式 |
| \*\*运算符 | a\*\*b | Math.pow(a, b); |  |
| 比较运算符 | a is b | var a;  a === b; |  |
| 逻辑连接运算符 | a && b | a && b |  |
| for循环1：数组遍历 | for a in b  print(a) | var \_a;  var \_len;  for (\_a = 0, \_len = b.length; \_a < \_len; \_a++) {  var a;  a = b[\_a];  print(a);  } | 至少有一条语句  循环体内需要1个Tab缩进 |
| for循环2：  数组遍历 | for a in [1,2]  print(a) | var \_ref;  \_ref = [1, 2]  var \_a;  var \_len;  for (\_a = 0, \_len = \_ref.length; \_a < \_len; \_a++) {  var a;  a = \_ref[\_a];  print(a);  } | 同上 |
| for循环3：  对象遍历 | for a of {x:1, y:2}  print(a) | for (a in {x : 1, y : 2}) {  print(a);  } | 同上 |
| for循环4：  对象遍历 | for a,b of {x:1, y:2}  print(a + " " + b) | var \_ref;  \_ref = {x : 1, y : 2}  for (a in \_ref) {  var b;  b = \_ref[a];  print(a + " " + b);  } | 同上 |
| while循环  break、  continue | while(1)  break; | while (1) {  break;  } | 同上 |
| 选择控制流 | x = 0  if a > 1  x = 1  else if (a > 0)  x = 2  else  x = 3 | var x;  x = 0;  if (a > 1) {  x = 1;  }  else if (a > 0) {  x = 2;  }  else {  x = 3;  } | If、else if后边的表达式可以不加括号，也可以加。  If、else等语句体内至少有一条语句  需要1个Tab缩进 |

# 编译器不支持的语言特性

|  |  |  |  |
| --- | --- | --- | --- |
| 特性描述 | CoffeeScript代码 | JavaScript代码 | 备注 |
| 表达式加括号 | (a+b) / 2 | (a+b)/2 | 对于函数调用  c(1)来说，如果允许表达式加括号，则产生歧义 |
| +=之类的运算 | A += b | A += b |  |
| 简写if | a = 1 if b | var a;  if (b) {  a = 1;  } |  |
| 自动return最后一个表达式 | square = (x) -> x \* x | square = function(x) {  return x \* x;  }; |  |
| 其它不是很常用的语法 |  |  |  |

# 编译器实现

## 对缩进的预处理

### （1）预处理原因

coffee语句块的判定依赖于缩进。然而，诸如Python, Coffee这一类依靠缩进的对齐文法，是上下文有关文法，不能在jison中实现。因此需要对输入的Coffee语言串进行一些预处理，输出能被下推自动机处理的串形式。

### （2）具体实现

在coffee的实际实现中，采用了任意个数空格对齐的方式。偶数个空格为一个语句块，奇数个空格与小于它的偶数空格属于同一语句块，如2空格和3空格属于第二级语句块，4空格属于第三级语句块。

考虑到这样的缩进方式太过自由，造成代码可读性下降，因此我们统一采用水平制表符（\t）进行对齐，使得代码归一整齐。

我们设计了一个类IndentLexer，来处理每行的缩进。该类位于Indent.js中。其思路是：

* 类的初始化：读取coffee输入串初始化自身，指定类成员count(上一行缩进个数) = 0。
* 对输入coffee语言串根据正则表达式/\n+/进行划分。返回得到一个字符串数组，其中存有代码的每一行。
* 遍历每一行，读取每行缩进个数并消去。当该行缩进数大于count，则在该行语句前添加{。若该行缩进数小于count，则在该行语句前添加}。
* 对代码列表，使用\n进行join操作，得到处理缩进后的代码并返回。返回的代码中{}为语句块闭合的标识，能很方便地被下推自动机处理，这一点与C语言类似。

## 词法分析

我们使用jison工具来辅助完成词法分析和语法分析的工作。Jison工具类似于bison，其输入是一个包含词法、语法、语义的定义文件，输出为词法分析和语法语义分析的代码。

在词法分析中，我们需要解析出关键词、变量、符号等，并定义符号的结合性。下面，具体说明一些典型的部分。

'{' return 'LEFT\_BRACE'

这里没有使用return '{'，是考虑到该符号是jison文件的关键字。

[a-zA-Z\_$][a-zA-Z0-9\_]\* return 'VARIABLE'

这是变量定义的正则表达式，变量应该满足：首字母是a-z或A-Z或\_或$ （考虑到jQuery），剩下的字母允许大小写字母、数字和下划线。

[0-9]+(\.[0-9]+)?\b return 'NUMBER'

这是数字的定义，支持整数和小数。

'is' return '==='

对于is，我们将其直接解释为判断是否等于的运算符'==='。

## 语法和语义分析

经过对缩进的预处理，CoffeeScript的语法就属于上下文无关文法了。对于一份代码，我们认为它是由一个个接连的语句块构成，于是有语法和语义：

S

: Block S

{ $$ = $1 + $2; }

|

{ $$ = ''; }

;

而对于语句块，我们考虑它几种典型的形式：表达式、If语句、For语句等，于是有语法和语义：

Block

: ExprBlock

{ $$ = $1 + ';\n'; }

| ForBlock

{ $$ = $1; }

| IfBlock

{ $$ = $1; }

……

而对于表达式ExprBlock，我们再将其分成赋值语句、数组、对象、函数声明、函数调用，以及用二元运算符将两个表达式连接等，于是有语法和语义：

ExprBlock

: ObjBlock //对象

{ $$ = $1; }

| ArrayBlock //数组

{ $$ = $1; }

| Const //常量

{ $$ = $1; }

| EXT\_VARIABLE '=' ExprBlock //赋值语句

{ $$ = $1 + ' ' + $2 + ' ' + $3; }

| EXT\_VARIABLE //变量：包含普通变量，对象成员变量，数组成员变量

{ $$ = $1; }

| FUNCTION //函数定义

{ $$ = $1; }

| 'VARIABLE' '(' ExprBlocks ')' //全局函数调用

{ $$ = $1 + $2 + $3 + $4; }

| 'OBJ\_ELEMENT' '(' ExprBlocks ')' //对象成员函数调用

{ $$ = $1 + $2 + $3 + $4; }

| ExprBlock BINARY\_RELATION ExprBlock //表达式连接

{ $$ = $1 + ' ' + $2 + ' ' + $3; }

……

二元运算符BINARY\_RELATION包括典型的比较运算符，例如>=，逻辑连接运算符，例如&&，算数运算符，例如+，具体语法不再贴出。

函数调用是CoffeeScript不同于JavaScript的地方。经过预处理后，其格式为“(参数)->{语句}”或者“->{语句}”，其中参数之间以空格隔开。语句部分由于仍旧是0个或多个语句块，因此可以采用变量S来定义。于是有如下定义：

FUNCTION

: '(' VARIABLES ')' '->' LEFT\_BRACE S RIGHT\_BRACE

{ $$ = 'function(' + $2 + ') {' + '\n' + $6 + '}'; }

| '->' LEFT\_BRACE S RIGHT\_BRACE

{ $$ = 'function() {' + '\n' + $3 + '}'; }

;

另外一个典型的地方是for循环，特别是对数组的遍历，下表是CoffeeScript和JavaScript的对比，左边为CoffeeScript语句。

|  |  |
| --- | --- |
| for a in [1,2]  print(a) | var \_ref;  \_ref = [1, 2]  var \_a;  var \_len;  for (\_a = 0, \_len = \_ref.length; \_a < \_len; \_a++) {  var a;  a = \_ref[\_a];  print(a);  } |

暂且先不考虑右边变量的自动声明（这部分在后期处理中完成），观察两边的代码，可以得知，在语义分析时，如果in之后是一个数组而不是一个变量，则需要用\_ref变量来暂存这个数组。另一方面，需要定义辅助变量\_a，\_len，并使用a=\_ref[\_a]来获取数组的元素。最后，注意一下每个语句的位置即可。语法和语义定义如下：

ForCondition

……

| 'for' 'VARIABLE' 'in' ArrayBlock

{

$$ = '\_ref = ' + $4 + '\n' + //用\_ref变量暂存

'for (\_' + $2 + ' = 0, \_len = \_ref.length; \_' +

$2 + ' < \_len; \_' + $2 + '++) {' + '\n' + //for循环本身

$2 + ' = \_ref[\_' + $2 + ']' + ';\n'; //获取数组元素

}

……

其余的语法和语义基本上看着jison文件定义就可以理解，在此不再赘述。由于jison使用的是LALR分析，我们写的语法有一些移进归约冲突，不过通过默认的解决方式解决就可以了。

## 后期处理

后期处理包含变量声明和格式化代码。对于变量声明，我们首先通过正则表达式，将代码中的变量分离出来。然后根据大括号的匹配情况，来进行层次的处理。举个例子：

|  |  |
| --- | --- |
| 处理前 | 处理后 |
| //层次1  A = 1;  Function () {  //层次2  B = 1;  }  //层次1  A = 1;  Function() {  //层次2  B = 1;  A = 2;  } | //层次1  var A;  A = 1; //第一次给A赋值，且未声明  Function () {  //扫描到大括号后，把层次1压栈  //层次2  var B ;  B = 1; //第一次给B赋值，且未声明  }  //退出层次2，里边声明的B无效  //层次1  A = 1; //A已经声明  Function() {  //层次2  var B;  B = 1; // 层次1中没有B，当前层也没有，补充声明  A = 2;// 层次1中包含A，无需声明  } |

从上个例子中，我们可以看出，为了实现该效果，需要一个变量记录当前层次已经声明的变量，我们用existsFrame来记录。此外，还需要记录之前几层已经声明的变量，我们用existsAll来表示。

每当遇到左大括号，即进入新的一层时，需要把existsFrame压入到existsAll中，清空existsFrame。而当退出这一层时，让existsFrame弹出一个元素，并把它赋值给existsFrame。另外，我们只处理“变量=表达式”这种形式，数组、对象内部的变量不做处理。

关于格式化代码就比较简单了，只需要记录当前的层次，对于该层次的每一条语句，在之前空出（层次数-1）\*4个空格即可。需要注意的是，该层次的最后一个右大括号，需要按低一层次来处理，即少空出4个空格。