

解析器设计说明书

Rtimeman Confidential

1	文档基本信息	5
1.1	文档状态	5
1.2	修改记录	5
2	简单介绍	6
2.1	涵盖范围	6
2.2	对谁可见	6
2.3	项目信息	6
2.4	缩写和简称	6
2.5	相关文档	7
3	模块列表	7
4	结构框图	7
5	程序文件工作流程	8
6	如何跳行	9
7	正向执行和逆向执行的详细逻辑	12
7.1	执行逻辑背景	12
7.2	如何兼容逆向	13
7.3	如何兼容跳行	13
7.4	目前的实现策略	13
8	XML 文件翻译	14
9	TP 的行号信息	17
10	基于 NanoMsg 机制的解析器与控制器的交互	17
11	解析器状态机	20
12	解析程序架构	21
13	解析程序使用的 thread_control_block 结构体	21
14	表达式分析器流程图	23
15	表达式分析器对于数组的支持	24
16	数学函数支持	24
17	字符串函数和转换函数支持	25

18	token 取词器流程图	26
19	token 和 tok	28
20	符号表	28
21	组件设计：双运算符的实现	28
22	程序总体流程图	29
23	组件设计：扫描文件中的标号和代码行信息流程图	31
24	组件设计：程序命令 IF 和 ELSEIF 解析流程图	31
25	组件设计：程序命令 while 解析流程图	32
26	组件设计：程序命令 wend 解析流程图	34
27	组件设计：程序命令 break 解析流程图	35
28	组件设计：程序命令 continue 解析流程图	36
29	组件设计：程序命令 call/gosub 解析流程图	36
30	组件设计：程序命令 return 解析流程图	37
31	组件设计：程序命令 end 解析流程图	37
32	组件设计：程序命令 select 解析流程图	39
33	组件设计：程序命令 case 解析流程图	39
34	组件设计：赋值语句流程图	40
35	组件设计：MOV 相关命令解析流程图	42
36	组件设计：MOV 相关命令扩展参数解析流程图	43
37	组件设计：Timer 命令解析流程图	43
38	组件设计：UserAlarm 命令解析流程图	44
39	组件设计：Wait 命令解析流程图	44
40	组件设计：Pause 命令解析流程图	46

41	组件设计: Abort 命令解析流程图	46
42	验证策略	47
43	其它	47

Rtimeman Confidential

1 文档基本信息

本文档主要是用来说明 Rtimeman 控制技术公司的解释器的软件流程。

1.1 文档状态

目前状态：通过评审。

状态	描述
不可用	文档在制订中，无法作为参考
完成	文档完成
通过评审	文档已经通过评审

1.2 修改记录

文档修改记录			
版本号	日期 yyyy-mm-dd	作者	修改描述
FFAL2000D001	2018-02-02	卢佳明	初稿
FFAL2000D001	2018-02-07	卢佳明	添加共享内存机制，补充一些细节。
FFAL2000D001	2018-02-14	卢佳明	根据评审记录修改
FFAL2000D001	2018-11-07	卢佳明	修改控制器通信方式

2 简单介绍

2.1 涵盖范围

此文档用来规范 Rtimeman 控制技术公司产品中的控制器中的解析器的开发业务逻辑。

2.2 对谁可见

系统软件组工程师

2.3 项目信息

#	条目	信息	描述
1			
2			

2.4 缩写和简称

缩写和简称	描述
token	指的是解析器中的一个单词。是解析器解析的最小单元。

--	--

2.5 相关文档

#	文档名称	版本	文档存储地址	描述
1	Rtimeman Robot_核心系统_用户指令 (PartI 语法格式)_需求文档_(批准)	V1.1		
2				
3				
4				
5				
6				

3 模块列表

包括如下模块：

1. 数学运算 `forsight_eval_type` 模块。包括加减乘除。主要用于寄存器的运算。
2. 内嵌函数 `forsight_innerfunc` 模块。包括正弦余弦数学函数在内的各种函数。
3. 寄存器操作 `reg_manager` 模块。包括寄存器操作和 `process_comm` 初始化。
4. IO 操作 `forsight_io_controller` 模块。包括 IO 读写操作。
5. XML 翻译 `forsight_xml_reader` 模块。把 XML 文件翻译成 BAS 文件。
6. 程序属性文件 `forsight_program_property` 模块。包括读取 P 寄存器。
7. 控制器操作 `forsight_inter_control` 模块。包括各种控制器操作。
8. 内嵌命令 `forsight_innercmd` 模块。包括各种 MOV 语句在内的各种命令。
9. 语法分析 `forsight_basint` 模块。该模块实现了一个 BASIC 解释器。

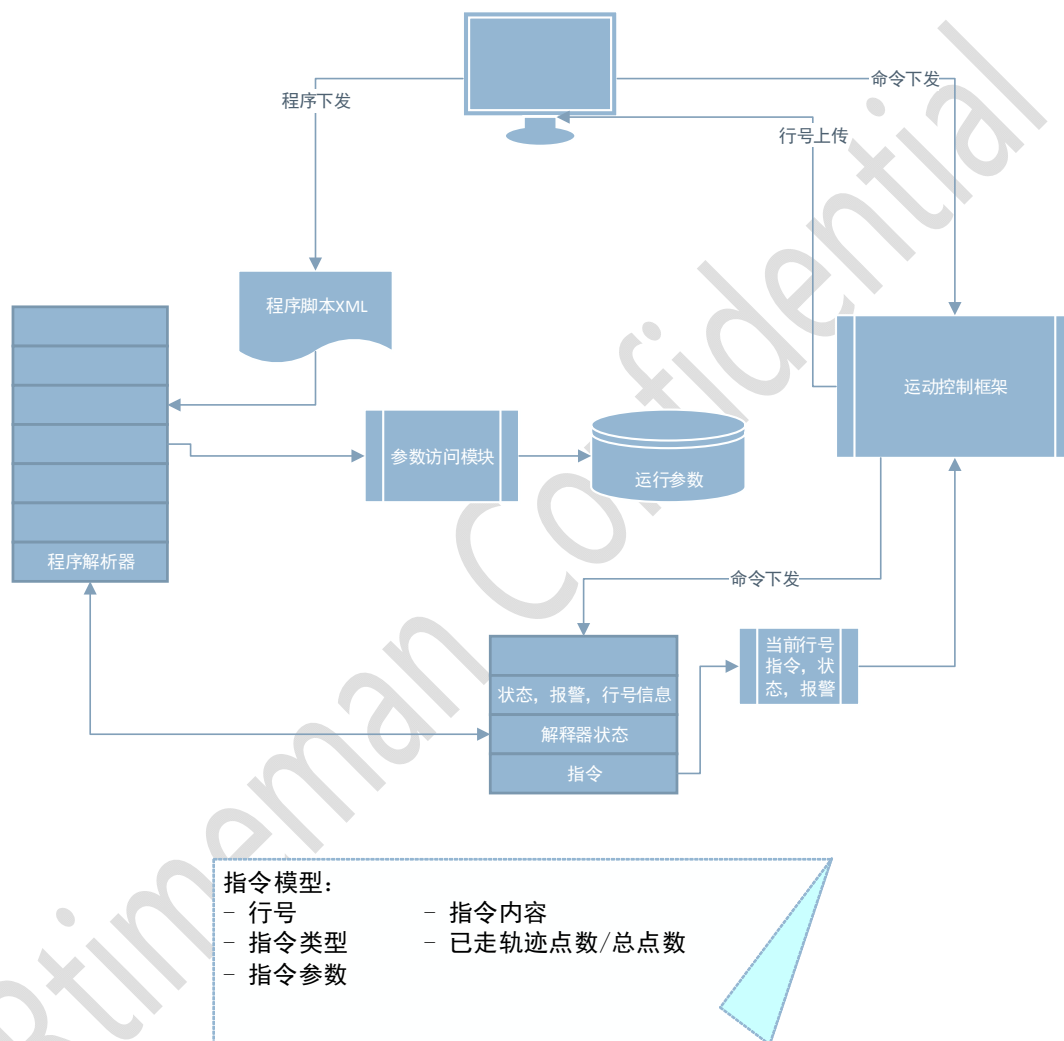
4 结构框图

控制器主控在收到 TP 的程序执行相关指令的时候，把指令发给解析器。

解析器根据指令执行保存在控制器上的程序，对于运动相关指令发送给控制器执行。

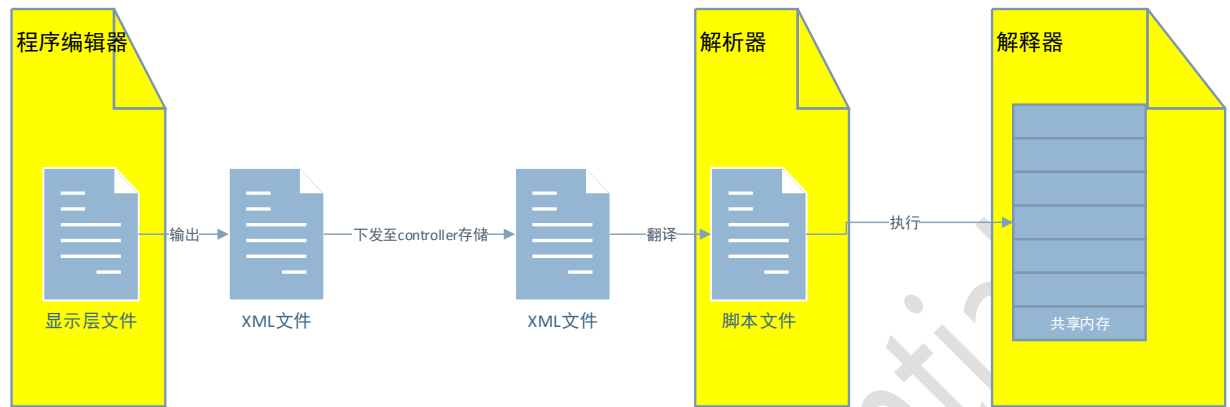
对于寄存器和 IO，从控制器获取。

在执行过程中，更新行号。和解释器的状态。

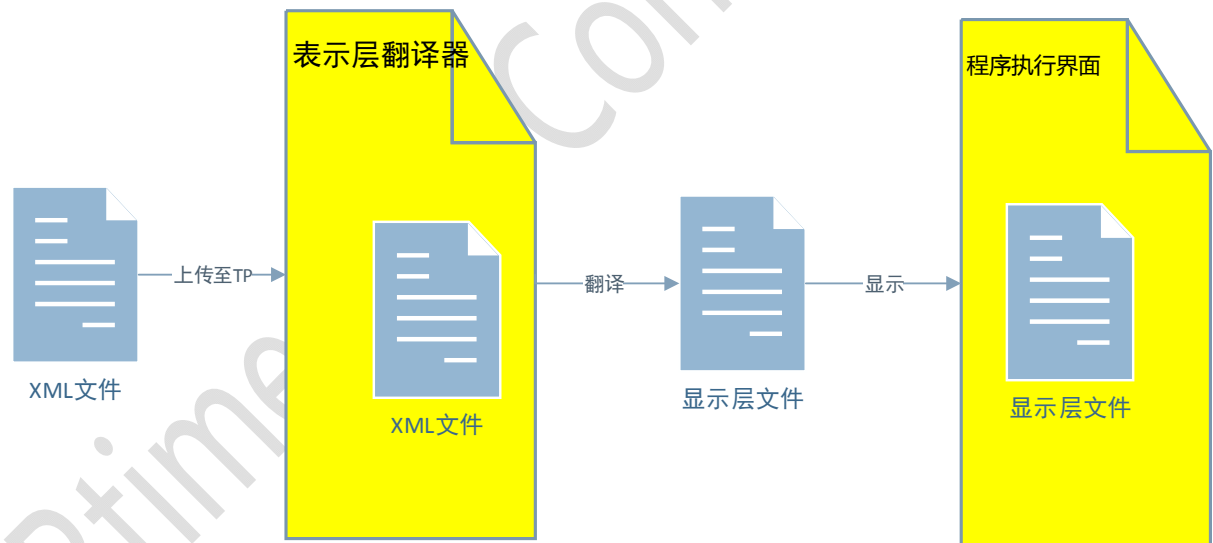


5 程序文件工作流程

程序下发解析流程:



程序上传显示流程：



6 如何跳行

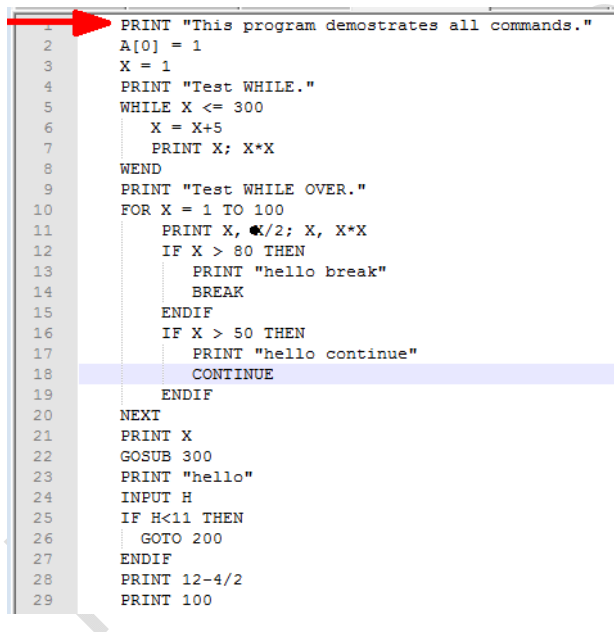
首先脚本代码样例如下：

```
1 PRINT "This program demonstrates all commands."
2 A[0] = 1
3 X = 1
4 PRINT "Test WHILE."
5 WHILE X <= 300
6   X = X+5
7   PRINT X; X*X
8 WEND
9 PRINT "Test WHILE OVER."
10 FOR X = 1 TO 100
11   PRINT X, X/2; X, X*X
12   IF X > 80 THEN
13     PRINT "hello break"
14     BREAK
15   ENDIF
16   IF X > 50 THEN
17     PRINT "hello continue"
18     CONTINUE
19   ENDIF
20 NEXT
21 PRINT X
22 GOSUB 300
23 PRINT "hello"
24 INPUT H
25 IF H<11 THEN
26   GOTO 200
27 ENDIF
28 PRINT 12-4/2
29 PRINT 100
```

可以看到这个脚本代码是一行一行的字符串。

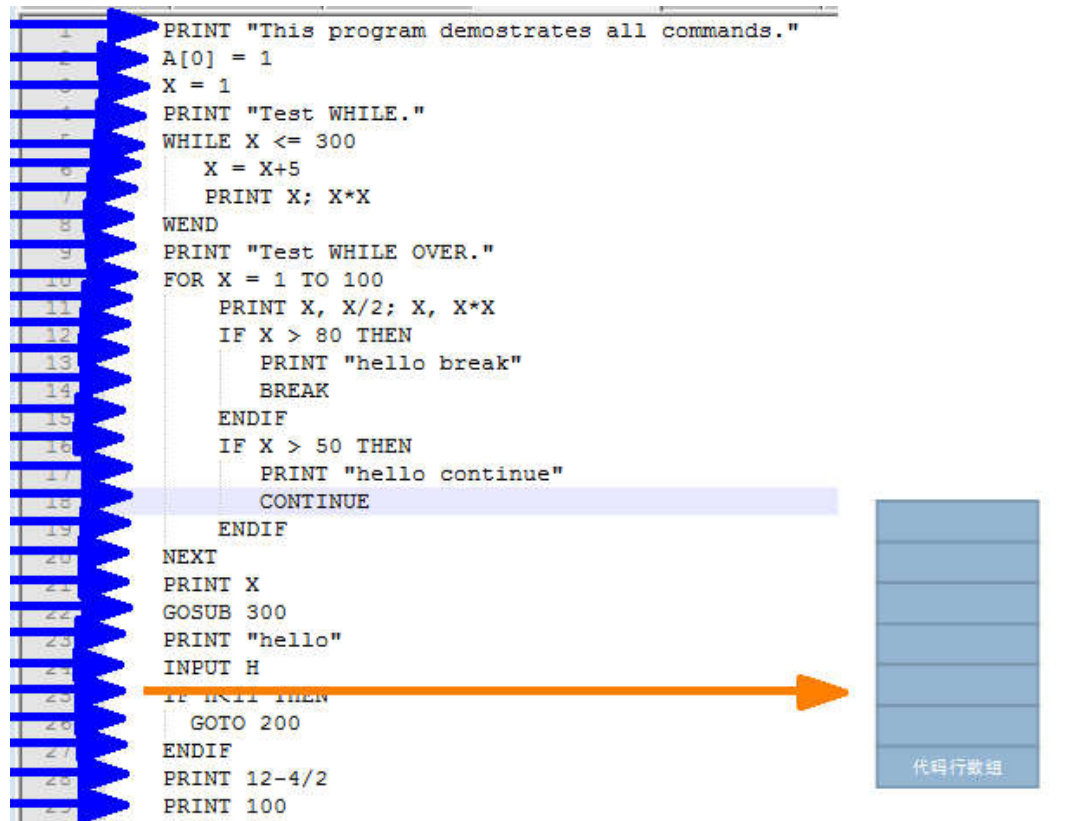
跳行的方法如下：

1. 读入脚本代码。使用一个指针指向代码字符串。表示从哪里开始解析。



```
1 PRINT "This program demonstrates all commands."
2 A[0] = 1
3 X = 1
4 PRINT "Test WHILE."
5 WHILE X <= 300
6   X = X+5
7   PRINT X; X*X
8 WEND
9 PRINT "Test WHILE OVER."
10 FOR X = 1 TO 100
11   PRINT X, X/2; X, X*X
12   IF X > 80 THEN
13     PRINT "hello break"
14     BREAK
15   ENDIF
16   IF X > 50 THEN
17     PRINT "hello continue"
18     CONTINUE
19   ENDIF
20 NEXT
21 PRINT X
22 GOSUB 300
23 PRINT "hello"
24 INPUT H
25 IF H<11 THEN
26   GOTO 200
27 ENDIF
28 PRINT 12-4/2
29 PRINT 100
```

2. 之后遍历一遍这个代码字符串。记下每一行的起始位置。



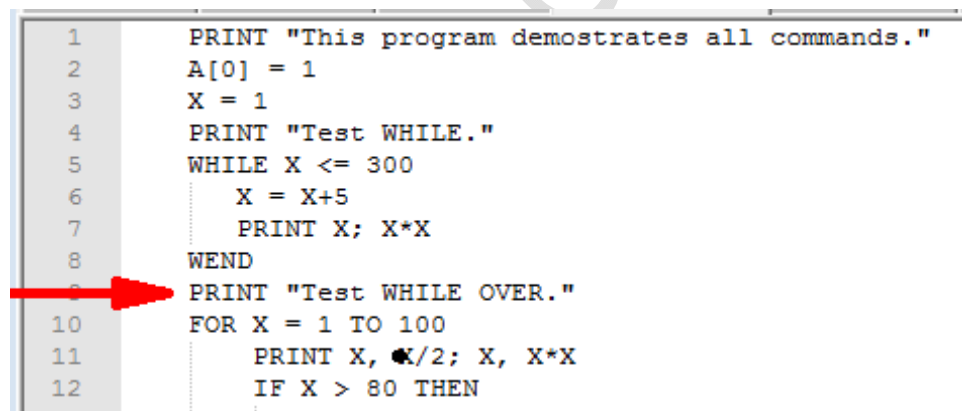
```

1 PRINT "This program demonstrates all commands."
2 A[0] = 1
3 X = 1
4 PRINT "Test WHILE."
5 WHILE X <= 300
6     X = X+5
7     PRINT X; X*X
8 WEND
9 PRINT "Test WHILE OVER."
10 FOR X = 1 TO 100
11     PRINT X, X/2; X, X*X
12     IF X > 80 THEN
13         PRINT "hello break"
14         BREAK
15     ENDIF
16     IF X > 50 THEN
17         PRINT "hello continue"
18         CONTINUE
19     ENDIF
20 NEXT
21 PRINT X
22 GOSUB 300
23 PRINT "hello"
24 INPUT H
25 IF H=11 THEN
26     GOTO 200
27 ENDIF
28 PRINT 12-4/2
29 PRINT 100

```

代码行数组

3. 在单步模式下，解释器会每次解释代码字符串指针指向的一行。



```

1 PRINT "This program demonstrates all commands."
2 A[0] = 1
3 X = 1
4 PRINT "Test WHILE."
5 WHILE X <= 300
6     X = X+5
7     PRINT X; X*X
8 WEND
9 PRINT "Test WHILE OVER."
10 FOR X = 1 TO 100
11     PRINT X, X/2; X, X*X
12     IF X > 80 THEN

```

4. 解释一行完成之后，等待用户输入行号。

5. 根据行号，取出每一行的起始位置。这里假设输入 4。

6. 代码字符串指针指向这个位置。再次调用解释器。

```
1 PRINT "This program demonstrates all commands."
2 A[0] = 1
3 X = 1
4 PRINT "Test WHILE."
5 WHILE X <= 300
6     X = X+5
7     PRINT X; X*X
8 WEND
9 PRINT "Test WHILE OVER."
10 FOR X = 1 TO 100
11     PRINT X, X/2; X, X*X
12     IF X > 80 THEN
13         PRINT "hello break"
14         BREAK
15     ENDIF
16     IF X > 50 THEN
17         PRINT "hello continue"
18         CONTINUE
19     ENDIF
20 NEXT
21 PRINT X
22 GOSUB 300
23 PRINT "hello"
24 INPUT H
25 IF H<11 THEN
26     GOTO 200
27 ENDIF
28 PRINT 12-4/2
29 PRINT 100
```

7 正向执行和逆向执行的详细逻辑

7.1 执行逻辑背景

首先解释器在每次执行完一条语句后，程序指针 `prog` 会指向下一条准备执行的指令。

- 在全速模式下，这个是正确的执行策略。
- 当处于单步调试模式下，解释器在每次执行完一条语句后，会进入 PAUSE 状态。等待 TP 的指令。此时，程序指针 `prog` 会指向下一条准备执行的指令。

如果不编写逻辑，其实是可以满足一半要求的，因为：

- 在正向单步的时候，只需要设定解释器状态为 EXECUTE，此时解释器会自动执行下一条准备执行的指令。
- 此时行号还没有更新。行号还是刚刚执行完的那一条语句对应的行号。

7.2 如何兼容逆向

但是，当需要进行逆向执行的时候，上面的策略就会出现问題。

在逆向的时候，需要实现下面的逻辑。

下面以 wait_test 程序已经执行完第三行 Wait2，程序指针指向到第 4 行 Wait3 的情况下，举例说明。

```
1 SUB main ()
2     WAIT 1
3     WAIT 2
→ 4     WAIT 3
5     WAIT 4
6     END
7 END SUB
```

此时行号为 3。

- 1 第一：当第一次从正向执行变成逆向执行，需要执行上次执行过的那一行。
也就是当第一次执行逆向命令，需执行第 3 行。相当于模拟倒车的行为。
- 2 第二：已经从正向执行变成逆向执行以后，需要执行上次执行过的那一行的前一行。
也就是第二次执行逆向命令的时候，需要执行第 2 行。

7.3 如何兼容跳行

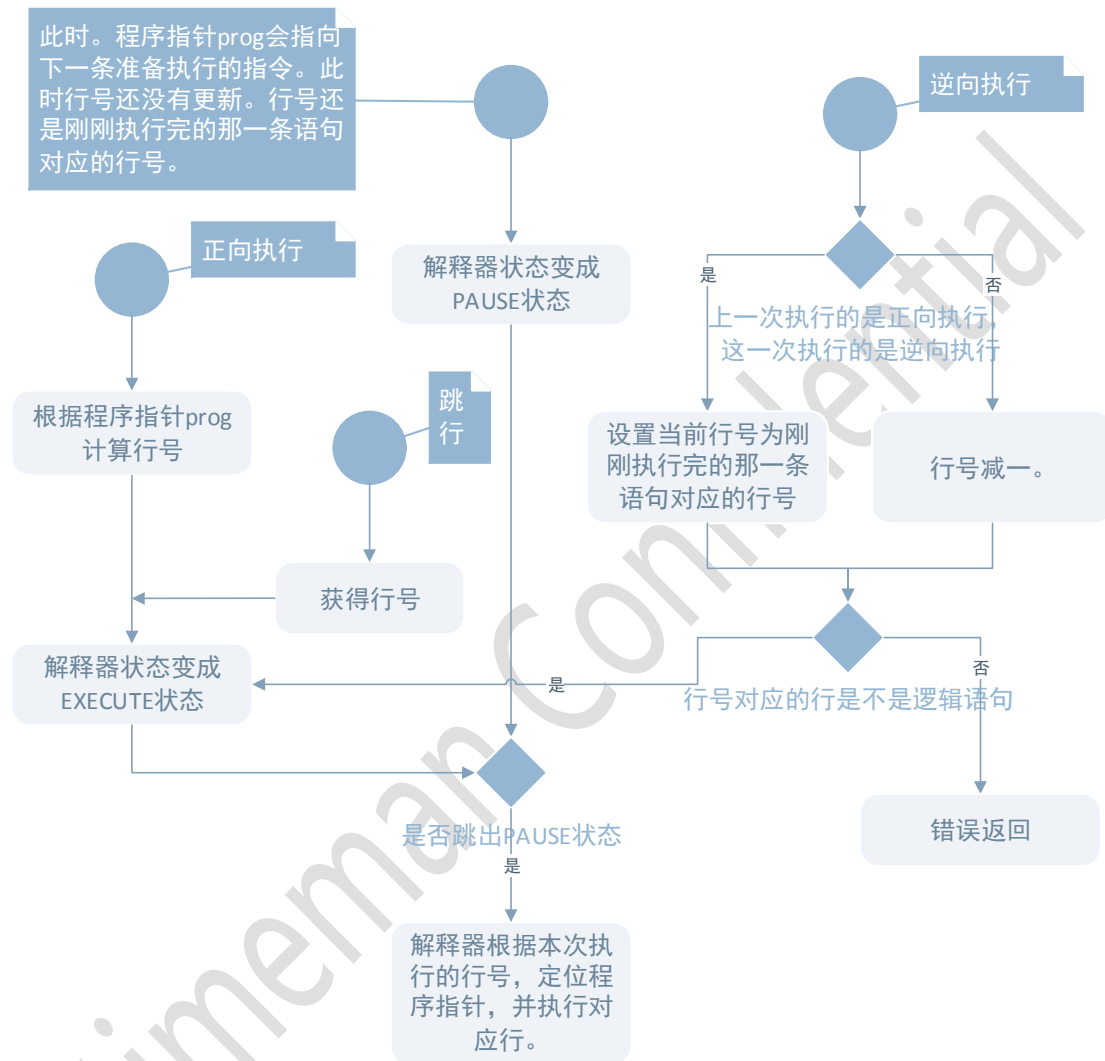
有了上面的讨论。可以看出其实逆向本质上就是等同于行号减一的跳行。

7.4 目前的实现策略

目前的实现策略是：

- 首先解释器在每次执行完一条语句后，程序指针 prog 会指向下一条准备执行的指令。此时行号还没有更新。行号还是刚刚执行完的那一条语句对应的行号。
- 之后解释器把自己变成 PAUSE 状态，等待操作命令把自己变成 EXECUTE 状态。
- 如果输入的是正向执行，正向执行操作命令处理函数处理如下：
 - 根据程序指针 prog 计算行号。此时计算出来的就是下一条准备执行的指令对应的行号。
 - 之后切换 EXECUTE 状态。
- 如果是逆向执行，逆向执行操作命令处理函数处理如下：
 - 如果上一次执行的是正向执行，这一次执行的是逆向执行，设置当前行号为刚刚执行完的那一条语句对应的行号。其实就是什么都不做。
 - 如果上一次执行的是逆向执行，这一次执行的也是逆向执行，行号减一。
 - 根据程序指针 prog 计算行号。判断该行是不是逻辑语句，如果是逻辑语句不能逆序。
 - 之后切换 EXECUTE 状态。

- 如果是跳行执行，跳行执行操作命令处理函数处理如下：
 - 获得行号。
- 当解释器从 PAUSE 被切换到 EXECUTE 状态的时候，本次执行的行号已经由外部操作命令处理函数成功生成。
- 解释器根据本次执行的行号，定位程序指针，并执行对应行。



8 XML 文件翻译

翻译程序使用 libxml2 库使用 DOM 遍历 XML 的每一个节点，生成 BASIC 格式的程序文件。

XML 文件格式参见《#1454-用户程序存储格式设计方案》

根据需求，每一个文件都是一个子程序。在设计上，实现为定义在文件中的一个名字叫 main 的函数。也就是文件支持包含多个函数。默认执行 main 函数。

因为 libxml2 库使用 DOM 将 XML 解释为一棵树。

整体流程其实就是对于这个 DOM 树的深度遍历。流程如下：

1. 调用 xmlDocGetRootElement 获取根节点，返回一个 xmlDocPtr 对象。

2. 遍历根节点的子节点。也就是根节点 `xmlNodePtr` 对象的 `children` 指针。
3. 对于得到的子节点，也是一个 `xmlNodePtr` 对象，再次遍历这个节点 `xmlNodePtr` 对象的子节点。
4. 每一个节点都有 **name 成员**。并可以根据**属性名**调用 `xmlGetProp` 获得属性值。调用 `xmlNodeGetContent` 获得**值**。

`<argument name="velocity" type="num"`

`unit="mm/s">30</argument>`

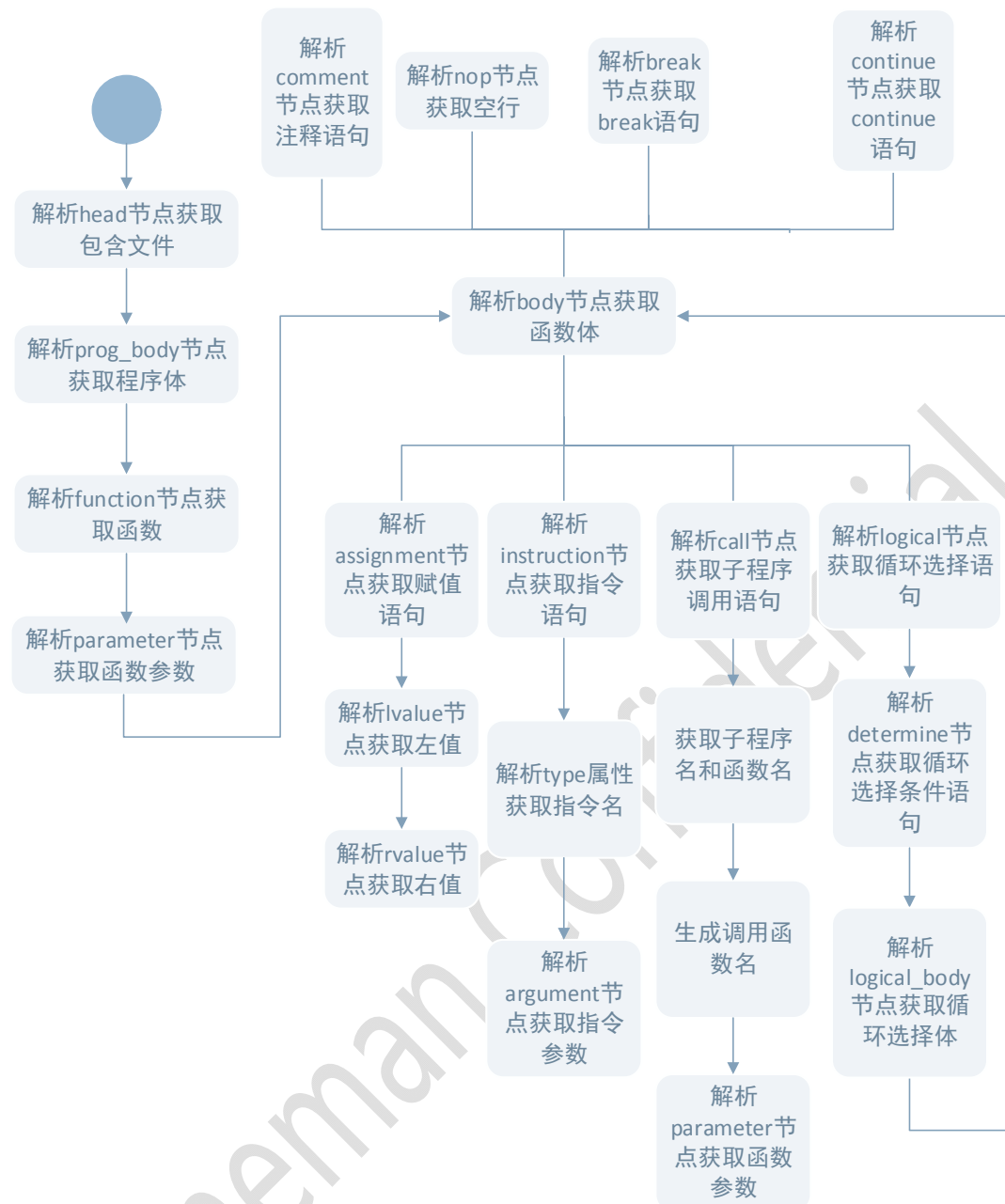
```
xmlNodePtr rootProg = xmlDocGetRootElement(doc);
```

```
for(xmlNodePtr nodeHead = rootProg->children;
```

```
    nodeHead; nodeHead = nodeHead->next){
```

```
    ....
```

```
}
```



对于指令节点，参数节点顺序，和《Rtimeman Robot_核心系统_用户指令（Part1 语法格式）_需求文档_V1.0》中保持一致。这一点由《#1454-用户程序存储格式设计方案》保证。

XML 文件格式如下：	翻译后的程序文件如下：
 prog_demo_dec.xml	 prog_demo_dec.bas.txt

9 TP 的行号信息

在生成每一行语句的时候，调用 `xmlGetNodePath` 获取这个节点对应的 XPath 作为行号信息。

代码如下：

```
21  R[1] = 5
22  IF R[1] + sin( 2 ) == 100 THEN
23      MOVEL 1.1 2.2 3.3 4.4 5.5 6.6 100 SV 50 ;ACC 40
24  ELSEIF R[1] == 1 THEN
25      WAIT COND DI[0] = ON 10 skip
26  ENDIF
```

代码对应的行号信息如下：

```
021:/prog/prog_body
022:/prog/prog_body/function[1]/body/logical[1]/determine
023:/prog/prog_body/function[1]/body/logical[1]/logical_body/instruction
024:/prog/prog_body/function[1]/body/logical[1]/vice_logical
025:/prog/prog_body/function[1]/body/logical[1]/vice_logical/vice_logical_body/instruction
026:/prog/prog_body/function[1]/body/logical[1]
```

可以看到，这里的 026 行是 `endif`。不是运动指令。在实际执行过程中是不会上报的。因为 TP 过来的 XML 是一个类似于这样的结构。

```
<logical type="if">
  <determine> . . . </determine>
  <logical_body> . . . . </logical_body>
</logical>
```

026行的`endif`是我自己生成的。生成的节点是`if`所在的节点。而`if`所在的节点则使用`if`中的`determine`加以区分。

10 基于 NanoMsg 机制的解析器与控制器的交互

1. 控制器和解释器在启动时候，会建立基于 Nano Msg 的全双工 TCP 连接。
2. 这个全双工 TCP 连接的建立方法如下：
 - a) 首先控制器启动。建立控制器服务器。
 - b) 之后解释器启动。建立解释器服务器。
 - c) 之后解释器建立解释器客户端，并连接控制器服务器。
 - d) 控制器服务器在接收到解释器客户端的连接后，建立控制器客户端，并连接解释器服务器。

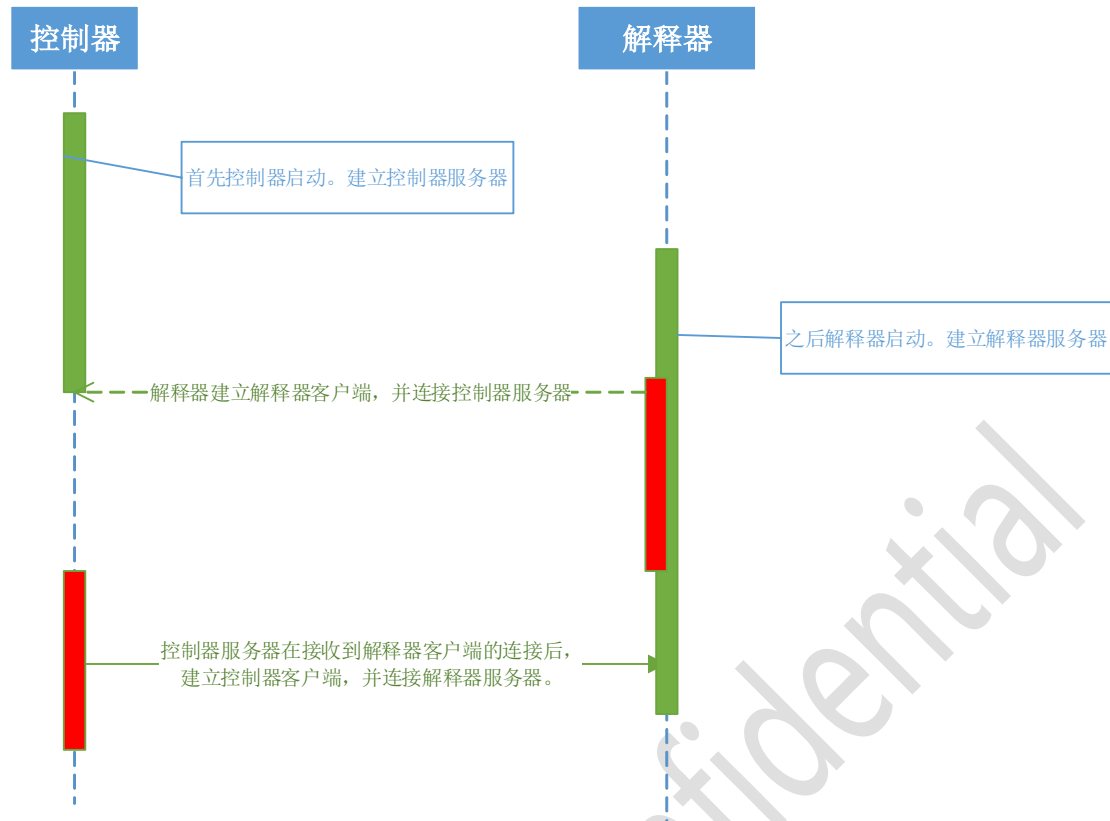
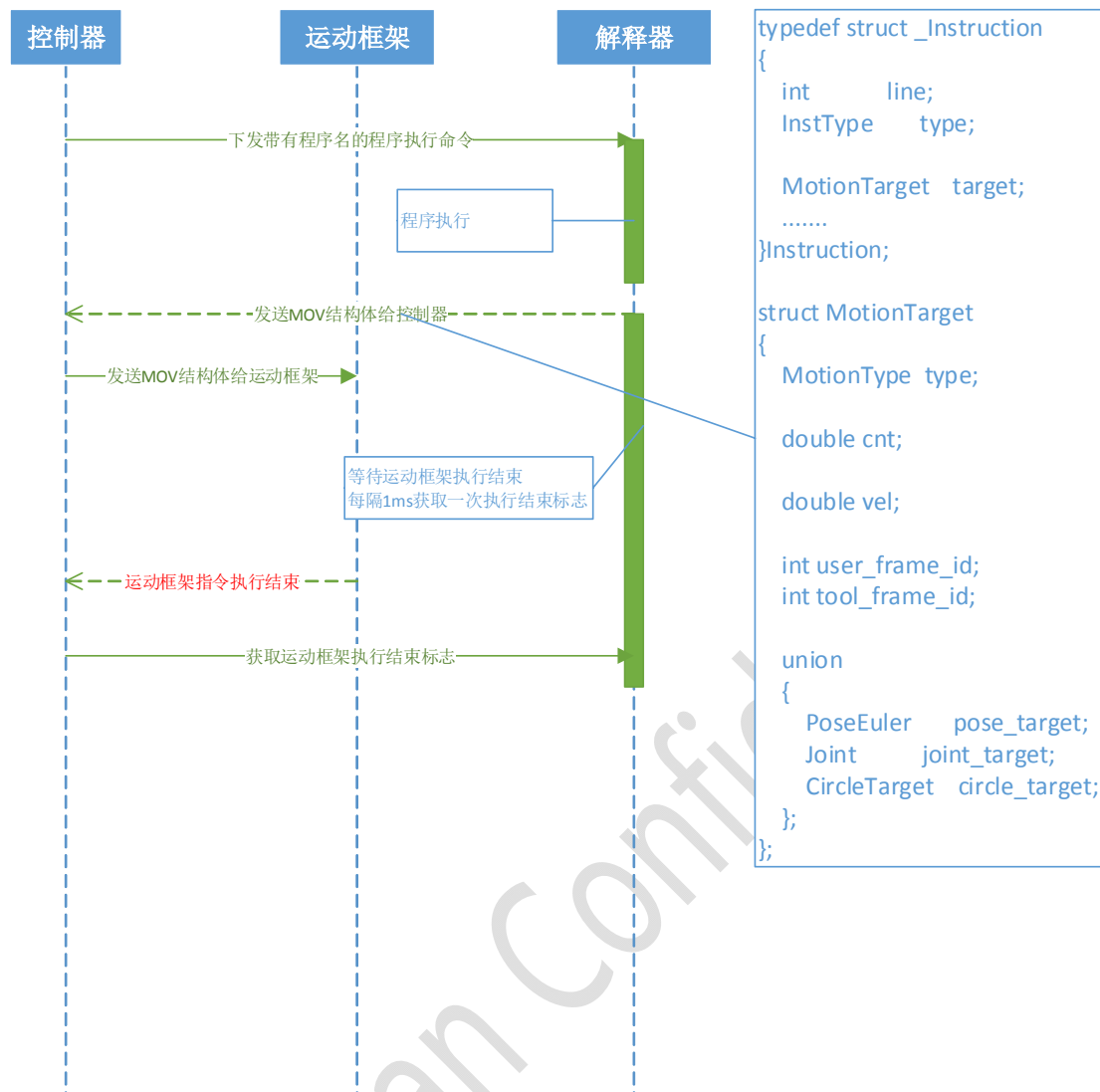


图 1 解析器和控制器之间的全双工TCP连接

3. 这个全双工 TCP 连接，包括两个 TCP 连接。
 - i. 一个 TCP 连接的客户端是控制器，服务器是解释器。用于控制器向解释器发送程序执行，单步，跳行，Pause，Abort 的控制命令。
 - ii. 一个 TCP 连接的客户端是解释器，服务器是控制器。用于解释器向控制器发送寄存器读写，IO 读写。MOV 语句，解释器状态。
4. TP 向控制器发送程序启动请求。控制器把该请求发送给解释器。
5. 解释器执行程序。解释器状态切换为执行状态
6. 当遇到 MOV 语句的时候，首先把把 MOV 语句填入 Instruction 结构体。发送给控制器。之后解释器进入一个死循环，每隔 1ms 查询一次控制器的运动下一条标志置位。
7. 控制器把 Instruction 结构体中的 MotionTarget 成员发送给运动框架。
8. 控制器每当收到一次解释器的查询，就会调用一次运动框架的 nextMovePermitted。
9. 当运动框架运动认为可以下发下一条后，函数 nextMovePermitted 会置位。
10. 解释器查询到控制器的运动下一条标志置位后，继续进行程序执行。一直到遇到下一条 MOV 的时候，又会进入到第 4 步。

序列图如下：



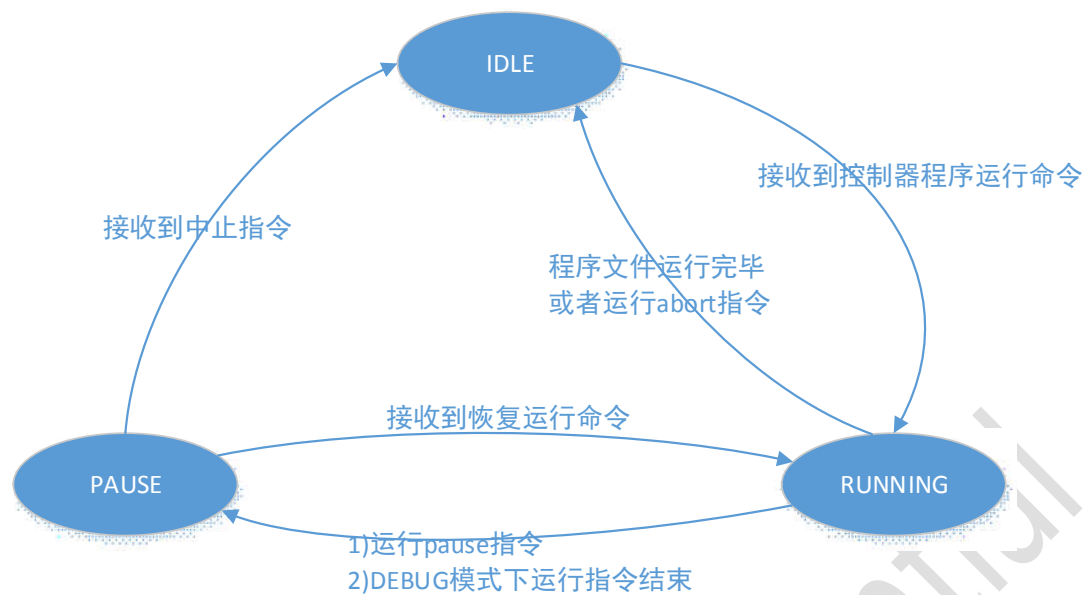
针对解析器与控制器的数据结构体定义如下：

- 1 解释器命令。定义为一个Instruction对象。
 - 1.1 PR寄存器修改SET_PR_REG
 - 1.2 HR寄存器修改SET_HR_REG
 - 1.3 MR寄存器修改SET_MR_REG
 - 1.4 SR寄存器修改SET_SR_REG
 - 1.5 R寄存器修改SET_R_REG
 - 1.6 PR寄存器读取GET_PR_REG
 - 1.7 HR寄存器读取GET_HR_REG
 - 1.8 MR寄存器读取GET_MR_REG
 - 1.9 SR寄存器读取GET_SR_REG
 - 1.10 R寄存器读取GET_R_REG
 - 1.11 发送指令SET_INSTRUCTION
 - 1.12 是否可以执行下一条指令IS_NEXT_INSTRUCTION_NEEDED
 - 1.13 校验IO是否存在CHECK_IO

- 1.14 IO修改SET_IO
- 1.15 IO读取GET_IO
- 1.16 更新解释器状态SET_INTERPRETER_SERVER_STATUS
- 2 解释器状态。如下：
 - 2.1 行号。
 - 2.2 解释器状态。
 - 2.3 程序名。
 - 2.4 行号对应的XPath。
- 3 解释器事件。如下：
 - 3.1 错误码。
- 4 控制器命令。定义了下面六个指令。
 - 4.1 程序加载LOAD
 - 4.2 程序跳转JUMP
 - 4.3 程序启动START
 - 4.4 程序正序执行FORWARD
 - 4.5 程序逆序执行BACKWARD
 - 4.6 程序跳行JUMP
 - 4.7 程序暂停PAUSE
 - 4.8 程序继续RESUME
 - 4.9 程序中止ABORT
 - 4.10 程序预读GET_NEXT_INSTRUCTION（待实现）
 - 4.11 程序启动模式设定SET_AUTO_START_MODE（已废弃）
 - 4.12 程序执行自动/单步模式切换SWITCH_STEP

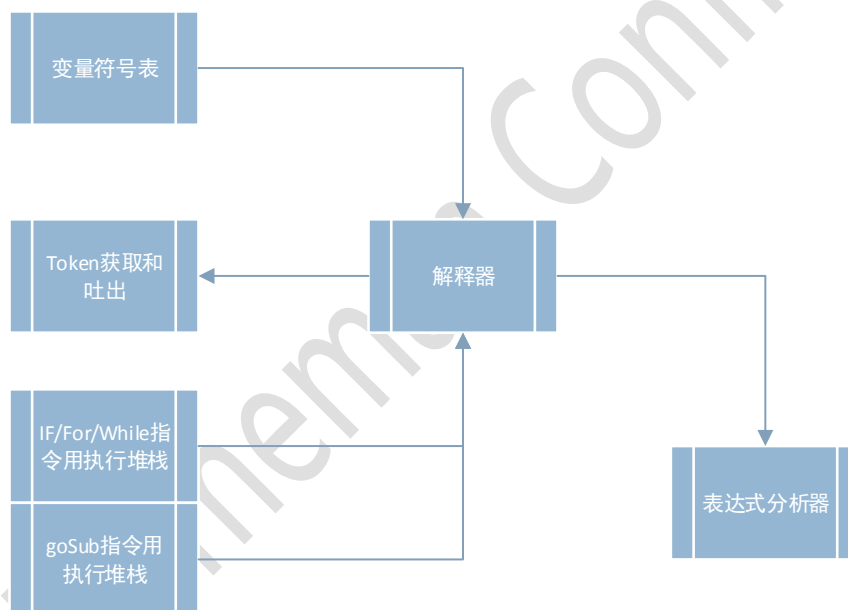
11 解析器状态机

解析器解析用户程序并向控制器发送指令，而控制器也控制解析器的运动过程，状态机如下：



12 解析程序架构

BAS-INT 是网上的一份开源代码。梁肇新的《编程高手箴言》中也引用过这套代码。



13 解析程序使用的 thread_control_block 结构体

每一个执行的程序会带一个 thread_control_block 结构体。里面包含了执行程序所需的所有信息。

```

struct thread_control_block {
    int iThreadIdx;           // 程序执行线程 ID
    char project_name[LAB_LEN]; // 程序名
    // This vector holds info for global variables.
    vector<var_type> global_vars; // 全局变量表
    Copyright, 2019   Rtimeman Motion Control Co., Ltd.   All rights reserved
    page 21 / 47
  
```

Date:

```

// This vector holds info for local variables
// and parameters.
vector<var_type> local_var_stack;           // 函数内局部变量表

// Stack for managing function scope.
std::stack<int> func_call_stack;           // 函数调用堆栈

int g_variable_error ; // = 0 ;
char *p_buf;           // program buffer           // 程序文本缓存区
char *prog;            /* prog pointer holds expression to be analyzed */
char *prog_end;        /* prog end position */      // 当前程序执行位置和程序终点

int    iSubProgNum ;           // sub programs number 子函数个数
char * sub_prog[NUM_SUBROUTINE]; // sub programs buffer 子函数缓存区

vector<prog_line_info> prog_jump_line; // jmp_line info 行号信息
ProgMode prog_mode ; // = 0; /* 0 - run to end, 1 - step */ 单步或全速
ExecuteDirection execute_direction ; /* 0 - FORWARD, 1 - BACKWARD */
                                     正向执行或逆向执行
bool is_abort , is_paused, is_in_macro;      是否处于 abort, paused 或宏指令
int  is_main_thread ; // = 0; /* 0 - Monitor, 1 - Main */
                                     是否为带有运动的主程序

char token[80];           // Current token 当前 token
char token_type, tok;     // 当前 token 类型

vector<sub_label> sub_label_table; // [NUM_LAB]; 函数跳转表
float ret_value ; // retrun values of function 函数返回值

/* stack for FOR/NEXT loop */           循环语句用结构体
struct select_and_cycle_stack selcyclstack[SELECT_AND_CYCLE_NEST];

char *gosub_stack[SUB_NEST];           /* stack for gosub */ 函数调用堆栈

int select_and_cycle_tos; /* index to top of FOR/WHILE/IF/SELECT stack */
                                     循环语句嵌套计数器
int gosub_tos; /* index to top of GOSUB stack */
                                     函数调用嵌套计数器

Instruction * instrSet ; // used by MOV* 控制器用运动结构体
// LineNum and Update flag
int iLineNum ; // Current LineNum 行号
//LineNumState stateLineNum ;
// MotionTarget currentMotionTarget ;
map<int, MoveCommandDestination> start_mov_position ; 每一句运动
// iLineNum :: movCmdDst 语句的起始点

```

```
vector<string> vector_XPath ;
// Home Pose
char home_pose_exp[LAB_LEN];
Joint currentJoint ;
PoseEuler currentCart ;
};
```

行号和 XPath 的对应表

HOME 点数据

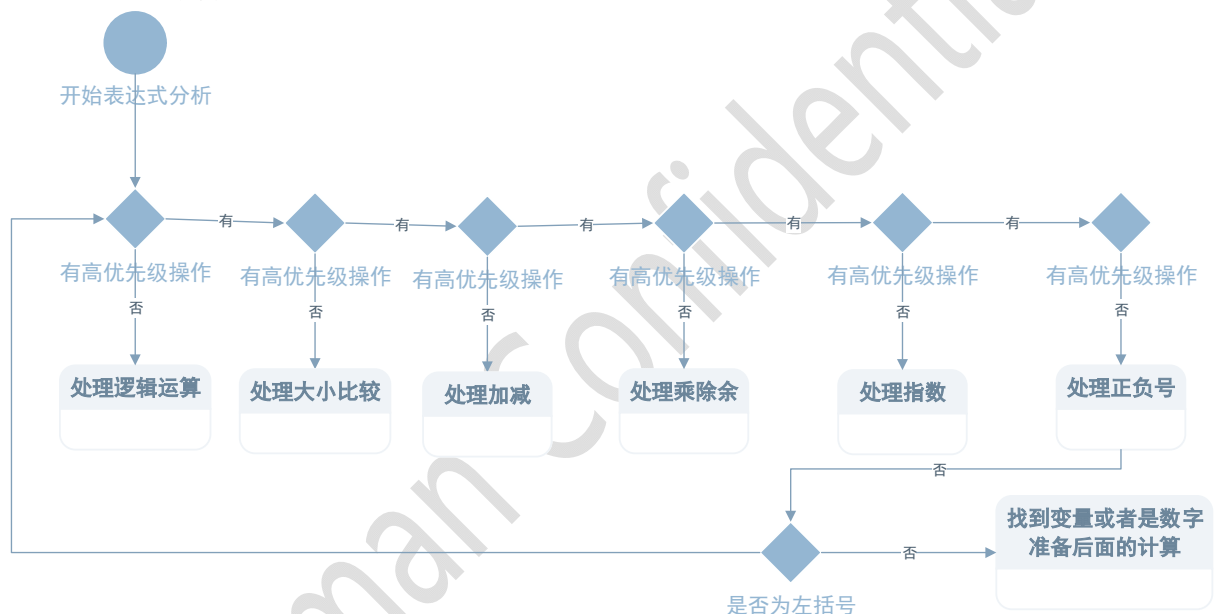
程序当前关节位置

程序当前笛卡尔位置

14 表达式分析器流程图

根据《Rtimeman Robot_核心系统_用户指令（PartI 语法格式）_需求文档_V1.0》中的要求。表达式分析器需要支持下面的运算操作符：

1. 条件运算如 AND 和 OR。
2. 比较运算如，大于，等于，小于以及组合。
3. 加减乘除。

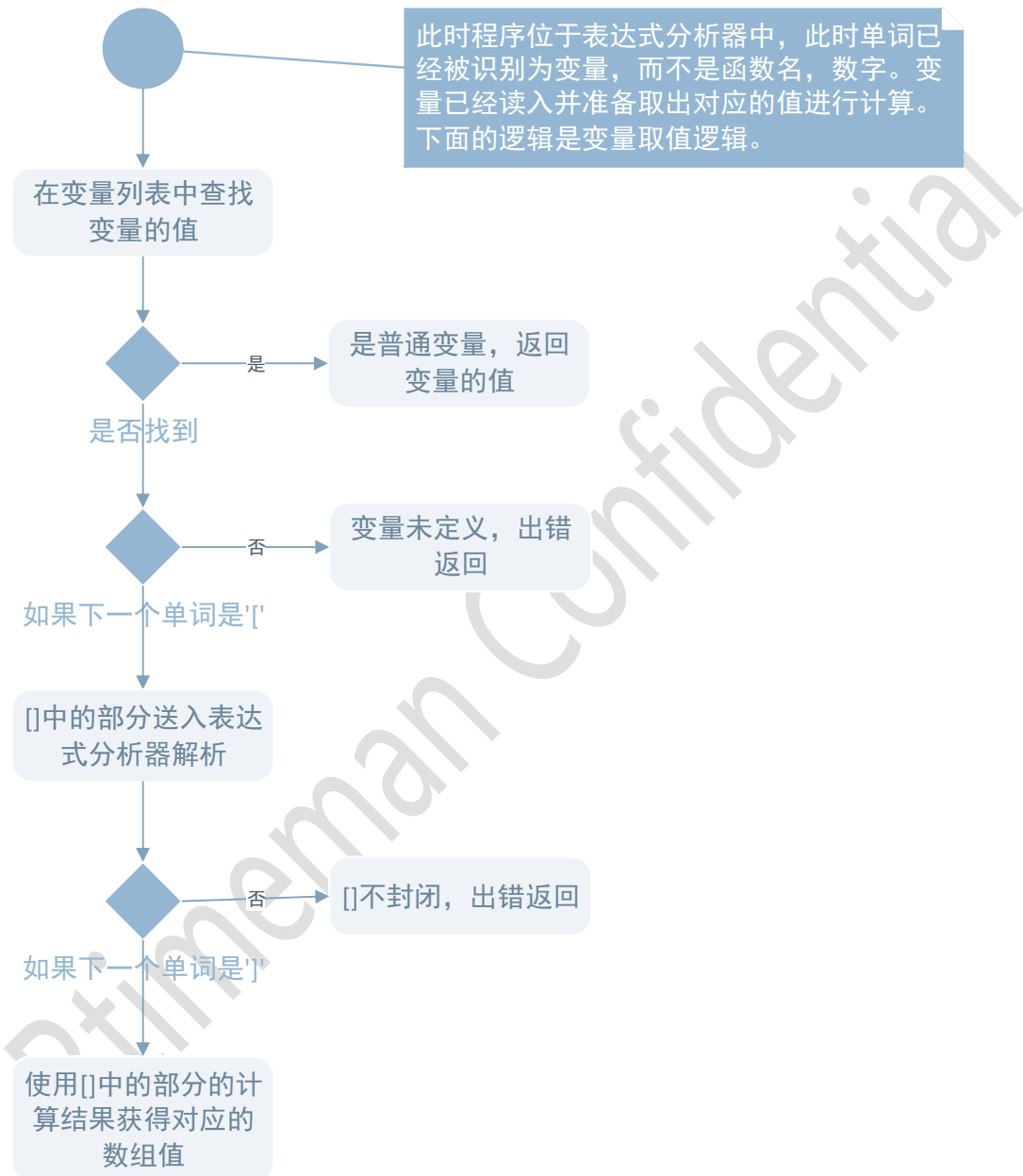


15 表达式分析器对于数组的支持

因为所有的寄存器都是数组格式的。因此上表达式分析器需要支持数组格式的变量。

这里利用了一个规定：

也就是对于形如 `PR[x]` 这样的寄存器。`PR` 是一个关键字，不能作为普通变量名。



16 数学函数支持

解释器支持 C 语言的大部分数学函数。如下。

函数	解释	函数	解释
fabs	求整型 x 的绝对值，返回计算结果。	fmod	求整除 x/y 的余数，返回该余数的双精度。
acos	计算 $\cos^{-1}(x)$ 的值，返回计算结果， x 应在-1 到 1 范围内。	frexp	把双精度数 val 分解为数字部分(尾数) x 和以 2 为底的指数 n，即 $val = x * 2^n$，n 存放在 $eptr$ 指向的变量中。返回数字部分 x $0.5 \leq x < 1$。
asin	计算 $\sin^{-1}(x)$ 的值，返回计算结果， x 应在-1 到 1 范围内。	log	$\log_e x$ ， $\ln x$ 。返回计算结果。
atan	计算 $\tan^{-1}(x)$ 的值，返回计算结果。	log10	求 $\log_{10} x$ 。返回计算结果。
atan2	计算 $\tan^{-1}(x/y)$ 的值，返回计算结果。	pow	计算 x^y 的值，返回计算结果。
cos	计算 $\cos(x)$ 的值，返回计算结果， x 的单位为弧度。	rand	产生-90 到 32767 间的随机整数。返回随机整数。
cosh	计算 x 的双曲余弦 $\cosh(x)$ 的值，返回计算结果。	sin	计算 $\sin x$ 的值。返回计算结果。 x 单位为弧度。
exp	求 E^x 的值，返回计算结果。	sinh	计算 x 的双曲正弦函数 $\sinh(x)$ 的值，返回计算结果。
fabs	求 x 的绝对值，返回计算结果。	sqrt	计算根号 x 。返回计算结果。 x 应 ≥ 0 。
floor	求出不大于 x 的最大整数，返回该整数的双精度实数。	tan	计算 $\tan(x)$ 的值，返回计算结果。 x 单位为弧度。
		tanh	计算 x 的双曲正切函数 $\tanh(x)$ 的值。返回计算结果。

17 字符串函数和转换函数支持

解释器支持很多常用的字符串函数和转换函数。如下。

函数	解释	函数	解释
strlen	求整型 x 的绝对值，返回计算结果。	lower	字符串小写。
findstr	字符串查找，调用 <code>strstr</code> 函数。	upper	字符串大写。
substr	获取子字符串。调用 <code>string</code> 的 <code>substr(start, length)</code> 。	revert	字符串翻转。
replace	字符串单次正向替换。	atoi	字符串转整数。
replaceall	字符串迭代反复替换。	atof	字符串转浮点。

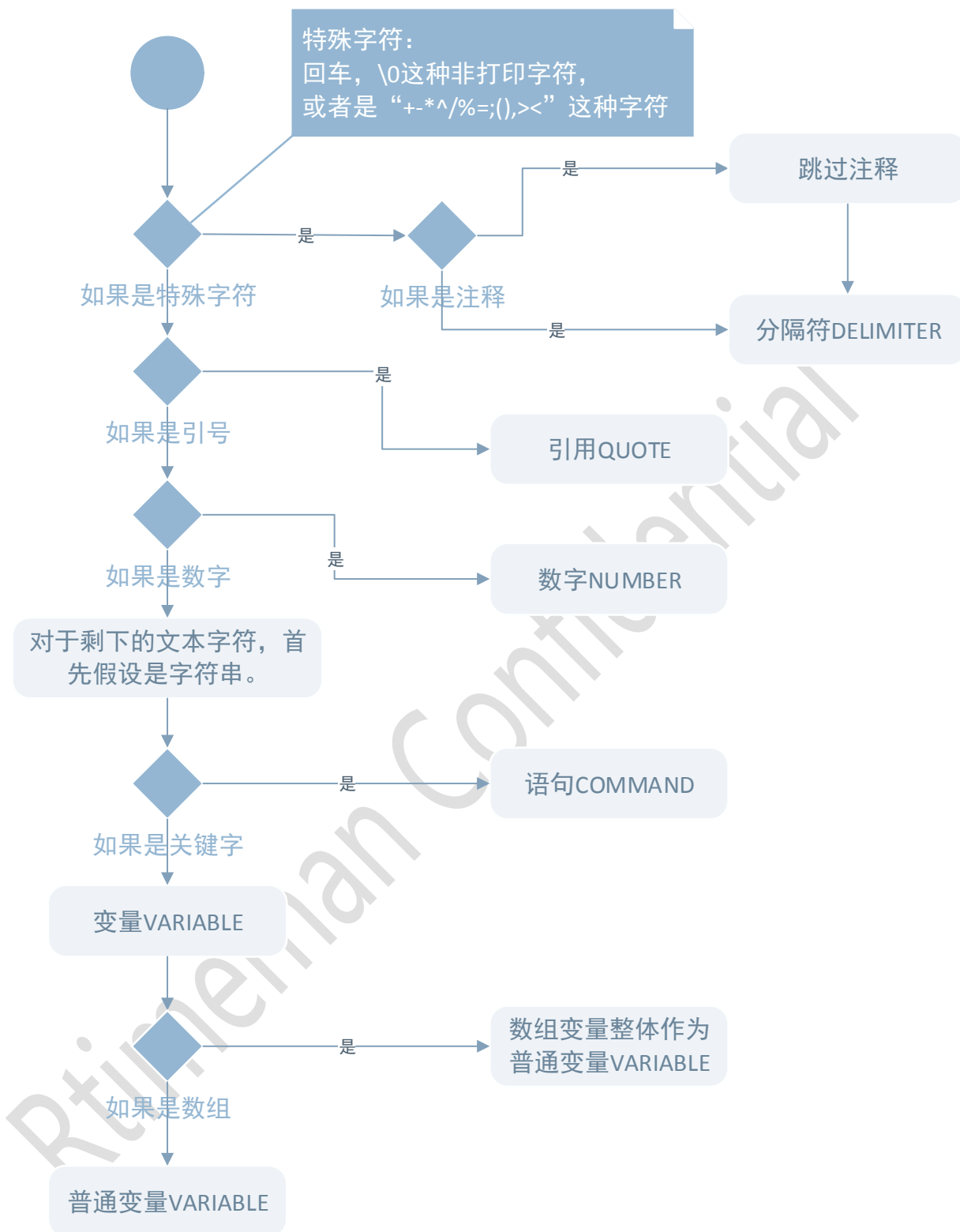
Replaceall iteration	字符串非迭代反复替换。	itoa	整数转字符串。
degrees	弧度转角度	ftoa	浮点转字符串。
radians	角度转弧度		

18 token 取词器流程图

首先是获取 token 类型的 `get_token` 函数返回的 `token_type` 如下。

```
#define DELIMITER 1    分隔符
#define VARIABLE  2    变量
#define NUMBER    3    数字
#define COMMAND   4    命令，也就是语句
#define STRING     5    字符串，中间类型。
                        最后要么是命令，要么是变量。
#define QUOTE     6    引用，其实就是用双引号包起来的字符串。
```

为了处理这六种情况。处理的顺序非常重要。因此上逻辑如下：



读取的单词被保存在一个全局变量中，因为存在读取过后，需要再次吐出来的情况。

19 token 和 tok

这里还有个 tok 的概念。

#define PRINT	1	#define INPUT	2
#define IF	3	#define THEN	4
#define ELSE	5	#define FOR	6
#define NEXT	7	#define TO	8
#define GOTO	9	#define EOL	10
#define FINISHED	11	#define GOSUB	12
#define RETURN	13	#define BREAK	14
#define CONTINUE	15	#define SELECT	16
#define CASE	17	#define WHILE	18
#define WEND	19	#define ENDIF	20
#define ENDIF	21	#define LOOP	22
#define ENDLOOP	23	#define SUB	24
#define CALL	25	#define END	26
#define IMPORT	27	#define DEFAULT	28
#define WAIT	29		

这里大部分都是命令。但是有一个特殊的命令 **FINISHED**。该命令在取词器读到 \0 时，被设置。表明代码解析结束。当然一般来说，主程序的最后一句正常来说是 **END**。

20 符号表

符号表定义如下：

```
// This structure encapsulates the info
// associated with variables.
struct var_type {
    char var_name[80]; // name
    float fValue ;
};
```

之后这里定义的是一个 vector 变量。

全局变量： `vector<var_type> global_vars;`

局部变量： `vector<var_type> local_var_stack;`

21 组件设计：双运算符的实现

获取 token 类型的 `get_token` 函数。会设置下面三个全局变量：

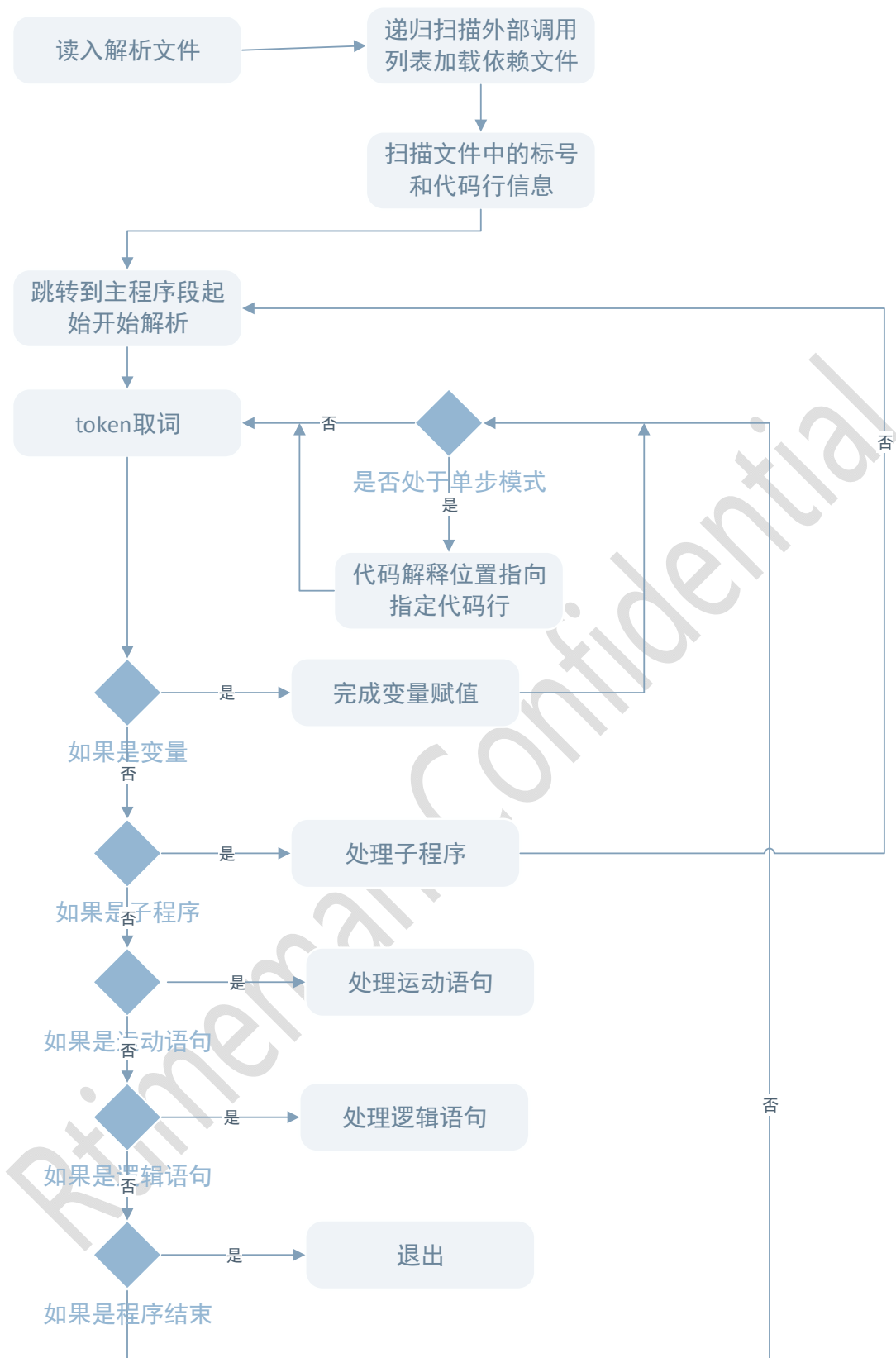
- char token[80];
- char token_type;
- char tok;

函数 `get_token` 执行过后，读取的单词作为字符串保存在 `token` 中。
这里如果需要实现双运算符，需要在读词的时候，进行翻译，变成特殊字符。

```
enum double_ops {  
  
    LT=1,    // value < partial_value  
  
    LE,      // value <= partial_value  
  
    GT,      // value > partial_value  
  
    GE,      // value >= partial_value  
  
    EQ,      // value == partial_value  
  
    NE,      // value <> partial_value  
  
    AND,     // value AND partial_value  
  
    OR,      // value OR partial_value  
  
};
```

22 程序总体流程图

该流程支持多线程。



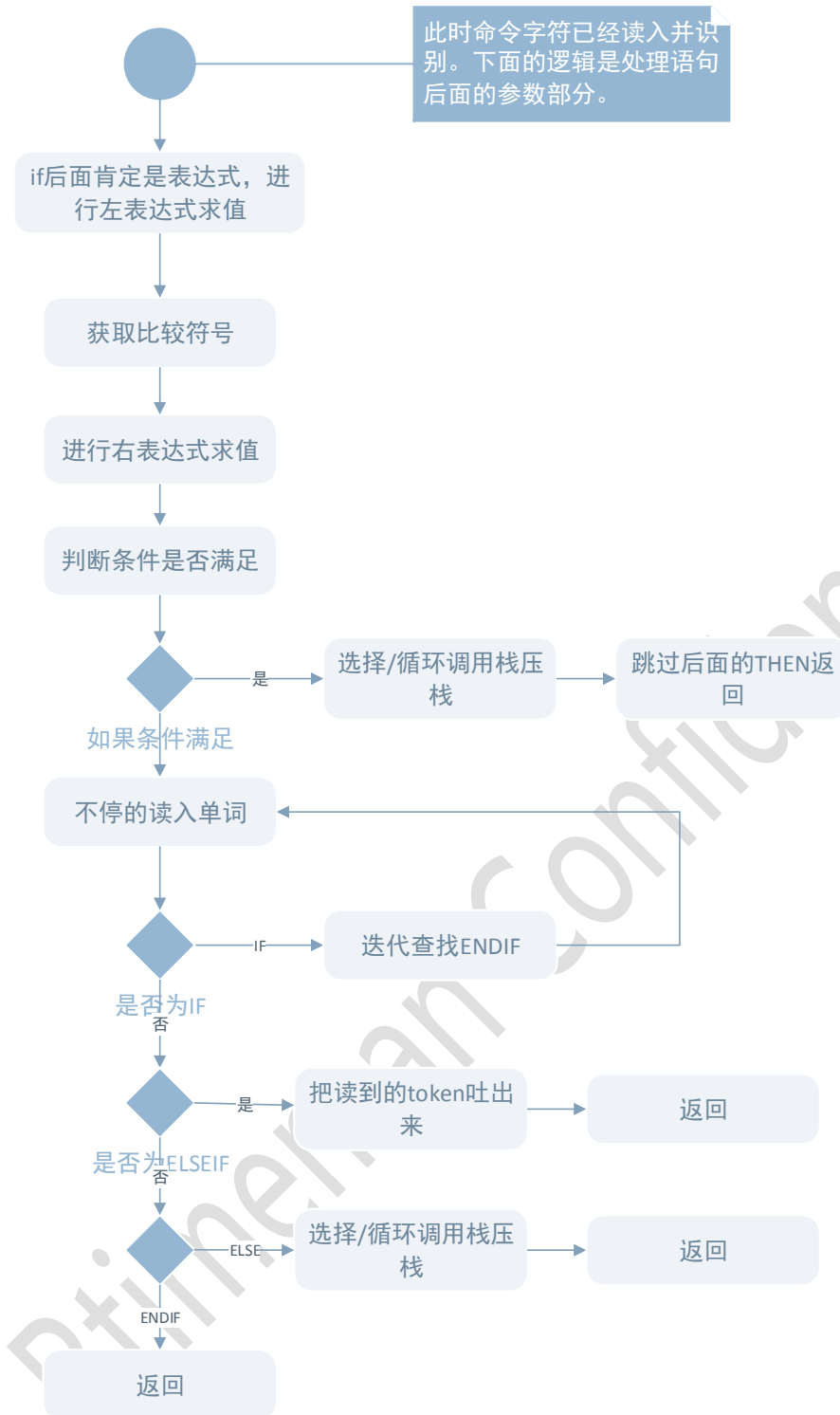
23 组件设计：扫描文件中的标号和代码行信息流程图

在 BASIC 语法中，函数的开头单词为 SUB。



24 组件设计：程序命令 IF 和 ELSEIF 解析流程图

判断条件后，满足条件就继续解析。不满足条件，就选择跳过这个代码块。



25 组件设计：程序命令 **while** 解析流程图

首先 **WHILE** 语句是需要和 **WEND** 语句配对的。因此上需要定义一个用于配对栈的 **fstack** 栈变量。定义如下：

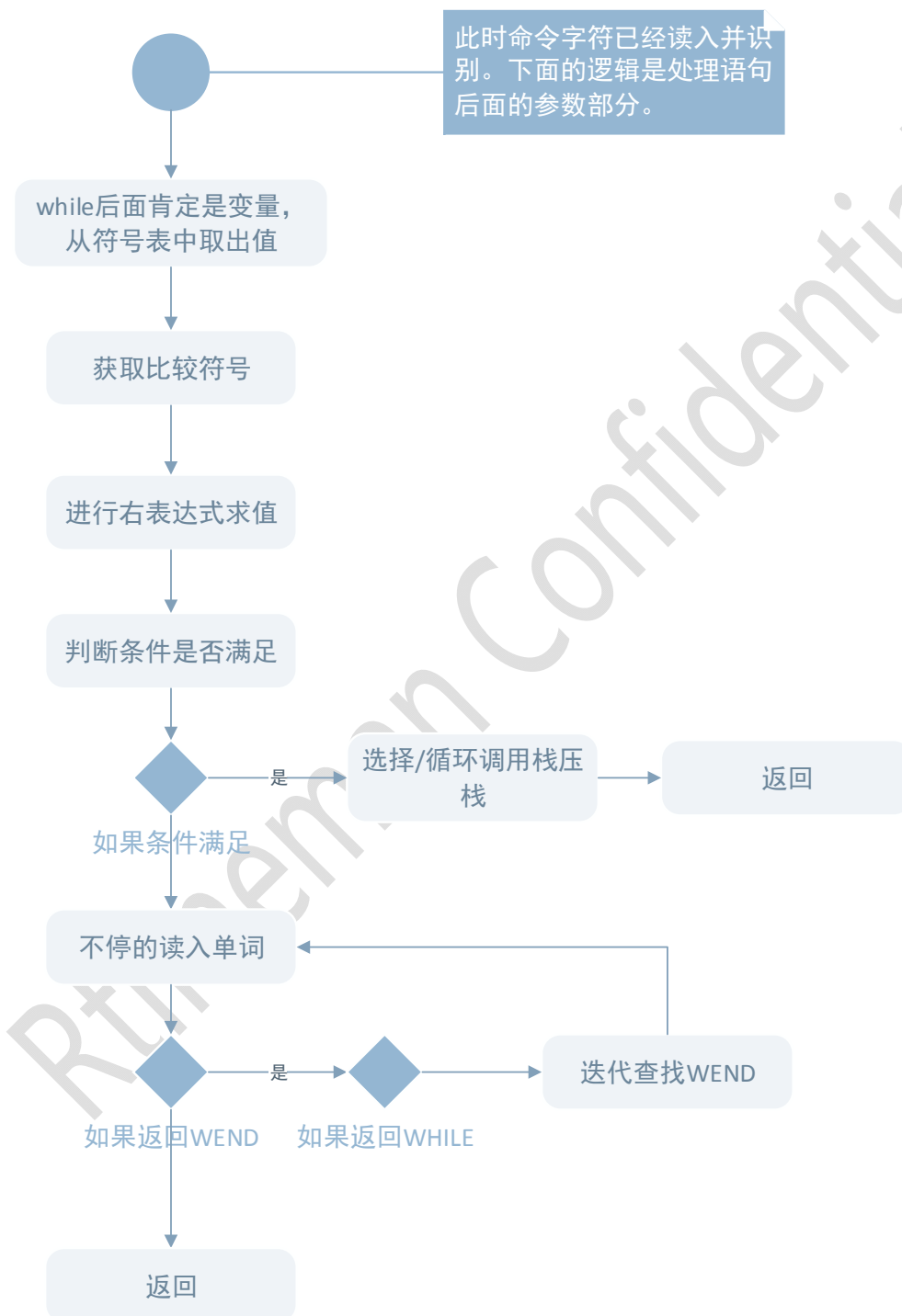
```

typedef struct select_and_cycle_stack {
    int itokentype ;
    // For and While with now

```

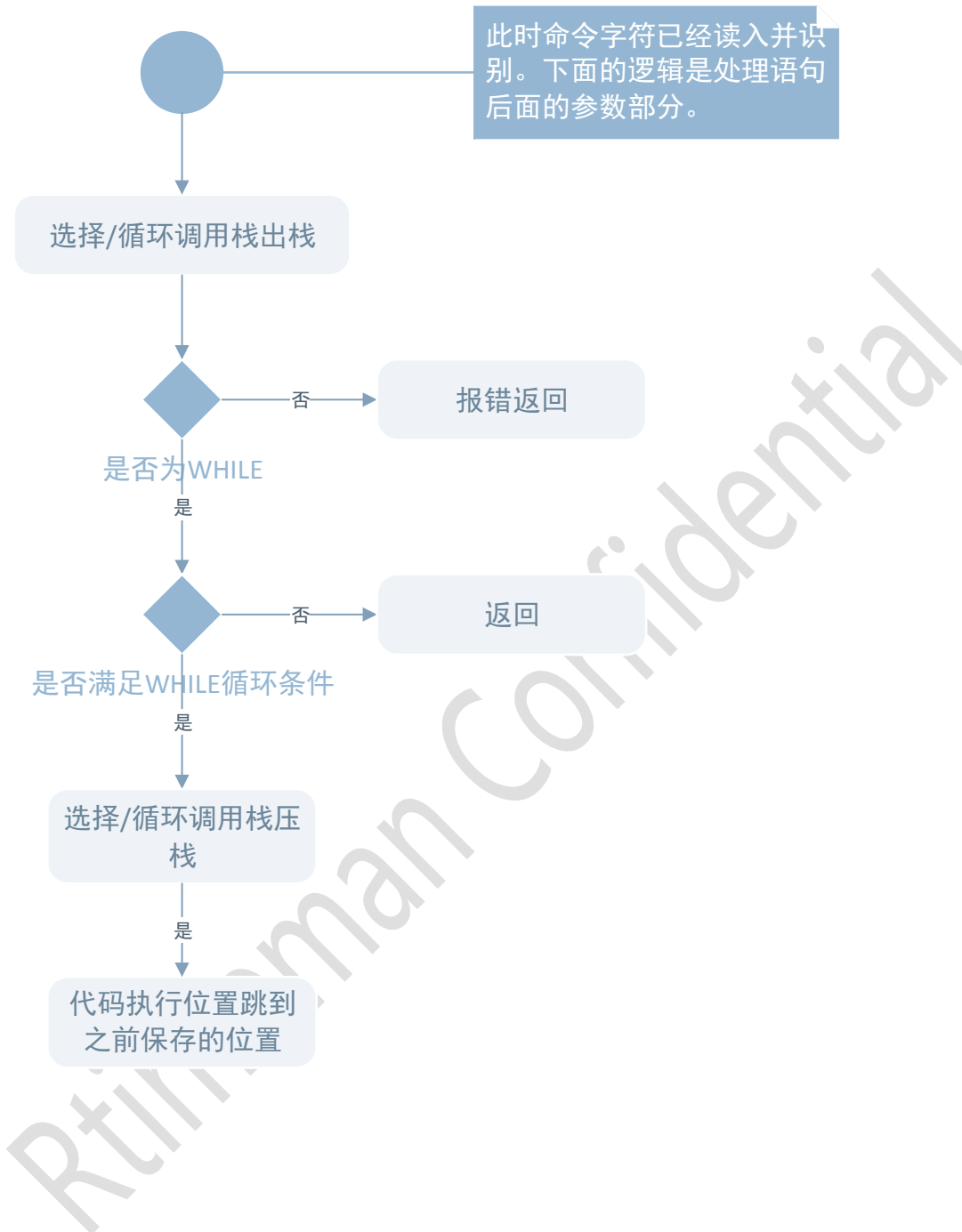
配对类型：IF/ FOR/ WHILE

float target; /* target value */	FOR 和 WHILE 用目标值
// For And While	
int var; /* counter or selector variable */	循环变量名。
char *loc;	代码循环起始位置。
}select_and_cycle_stack_struct ;	



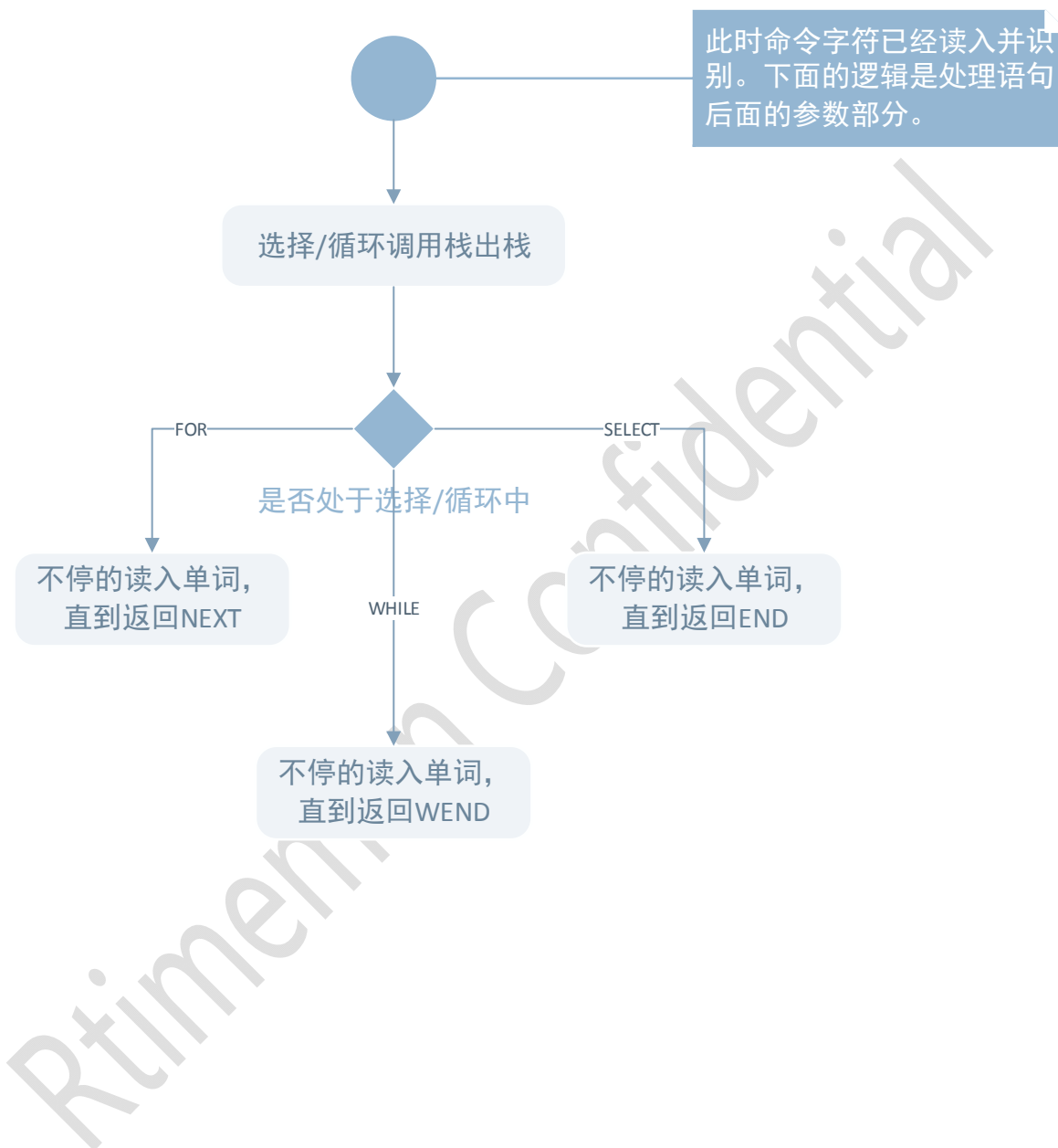
26 组件设计：程序命令 **wend** 解析流程图

wend 的功能是判断循环变量是否满足循环条件。



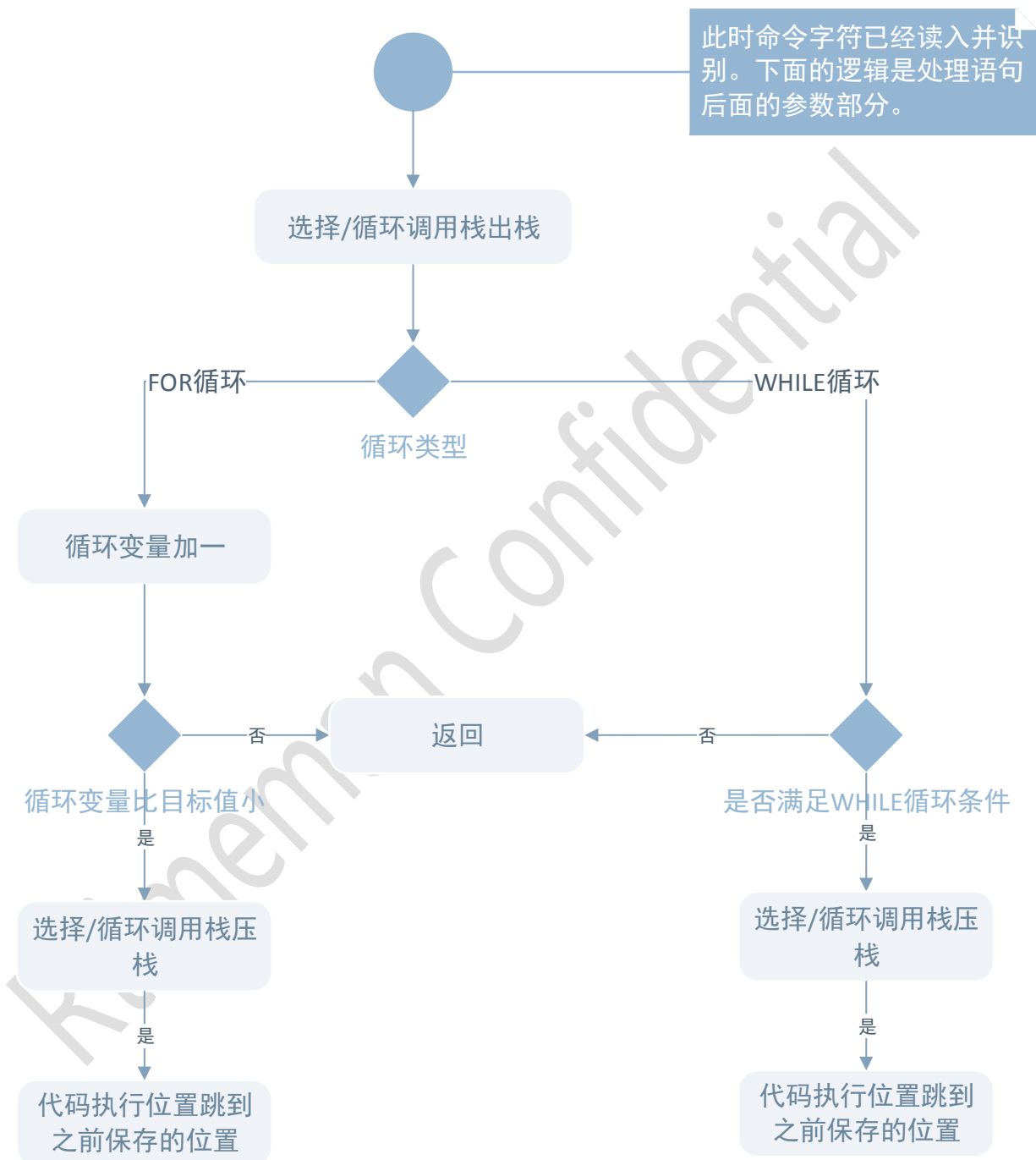
27 组件设计：程序命令 **break** 解析流程图

Break 的功能是跳出循环。



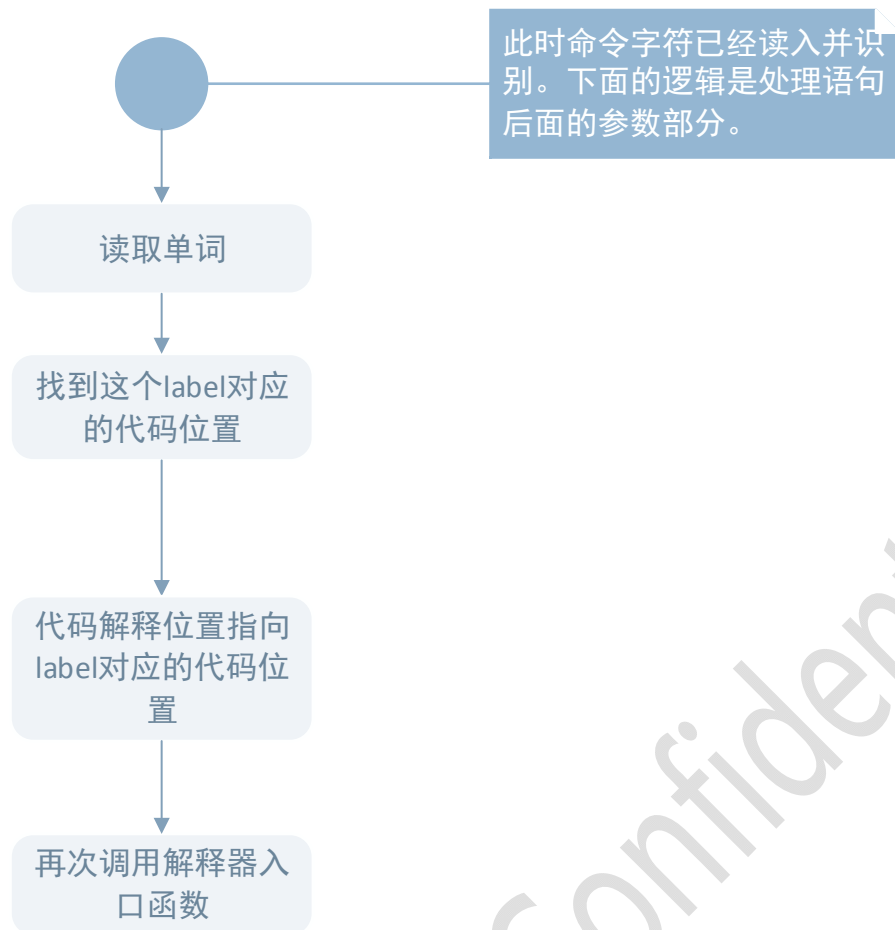
28 组件设计：程序命令 **continue** 解析流程图

Continue 的功能同 Wend 和 NEXT 的功能相同。



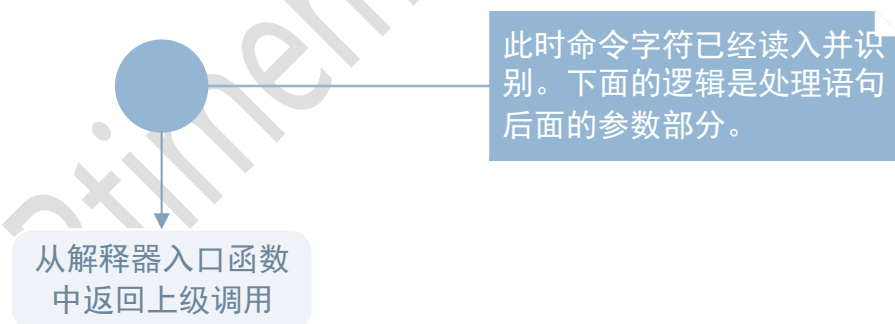
29 组件设计：程序命令 **call/gosub** 解析流程图

直接调用解释器入口函数实现该功能。



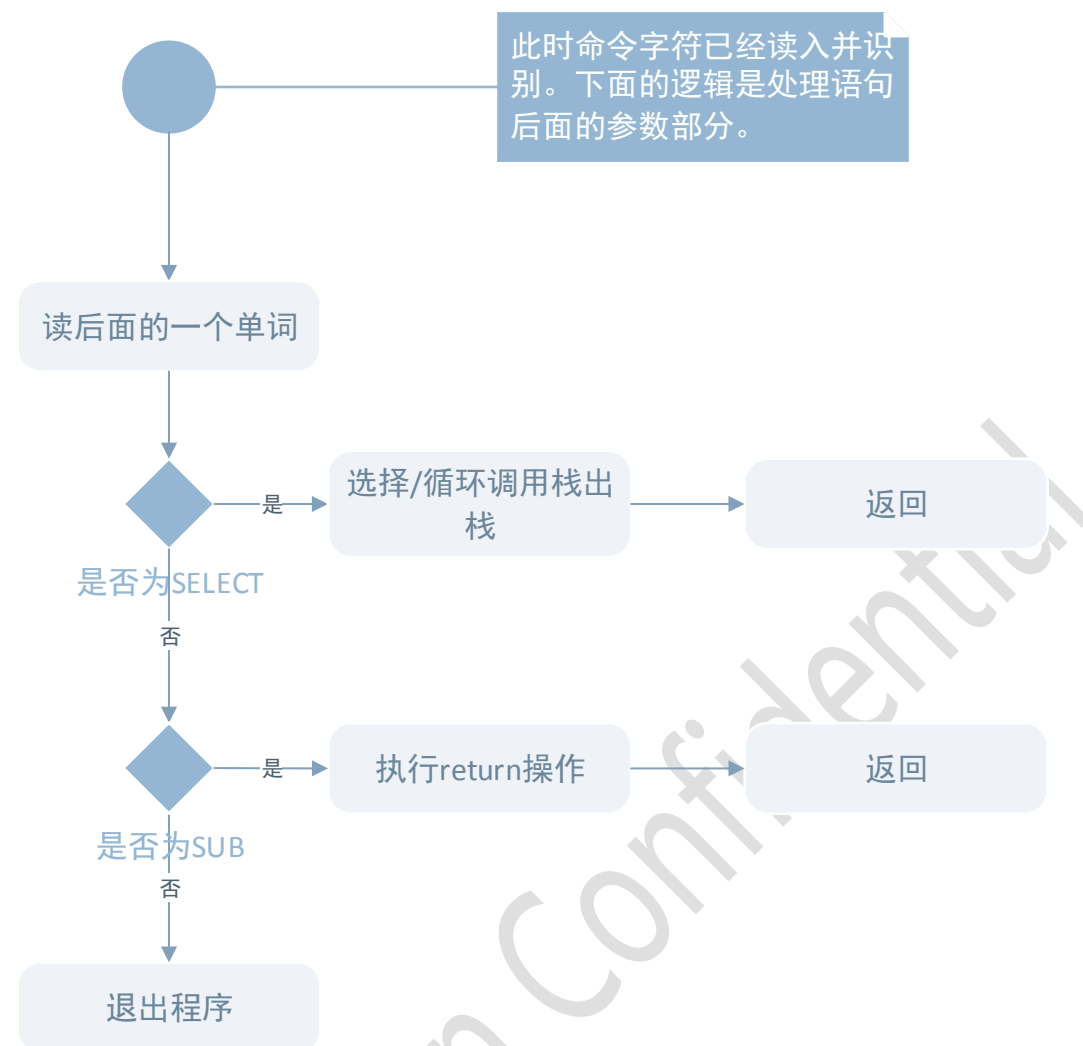
30 组件设计：程序命令 **return** 解析流程图

直接使用 C 语言的函数功能返回实现。



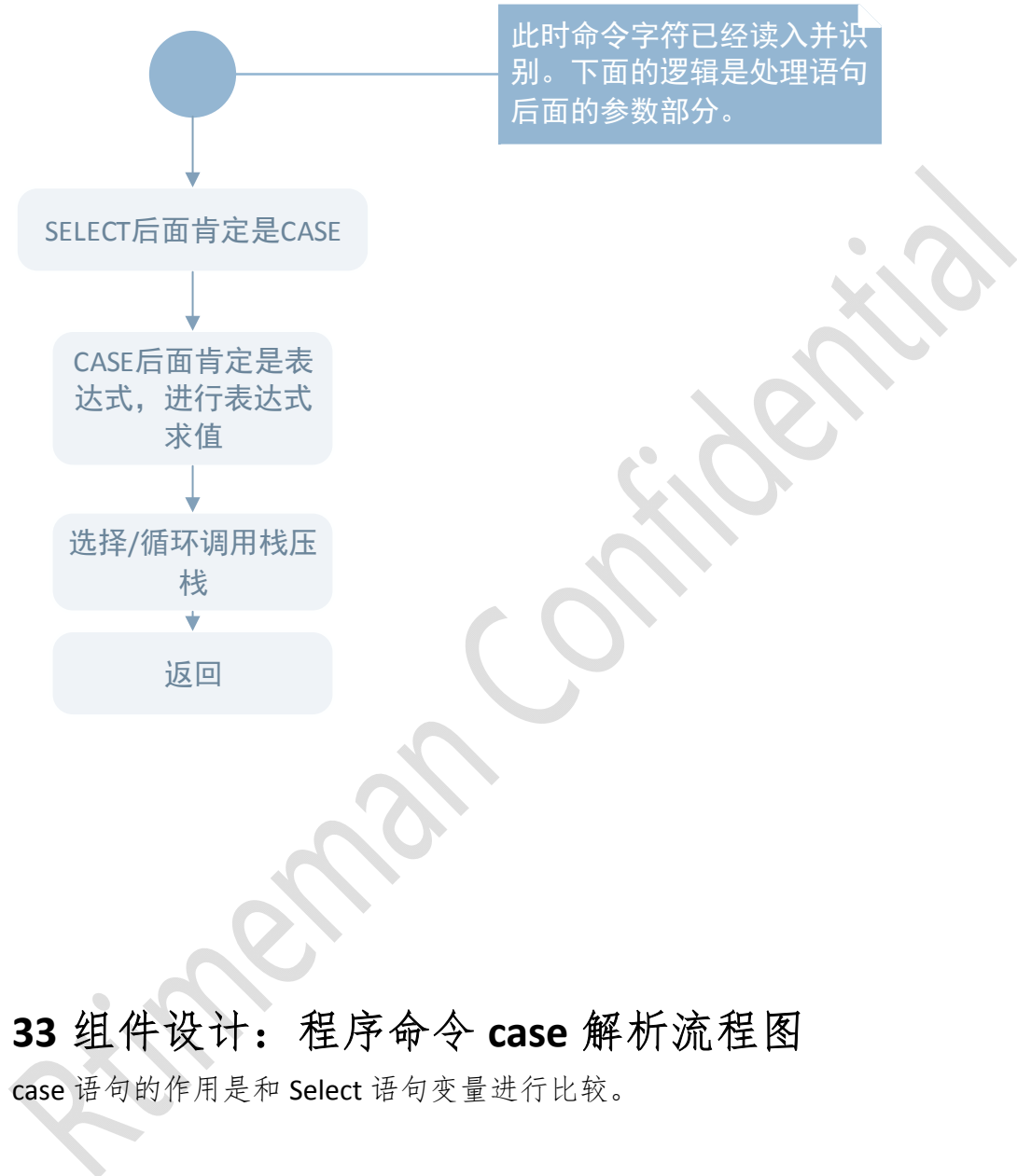
31 组件设计：程序命令 **end** 解析流程图

这里处理了三种情况，分别是 END SELECT 语句，END SUB 语句和程序结束 END 语句。



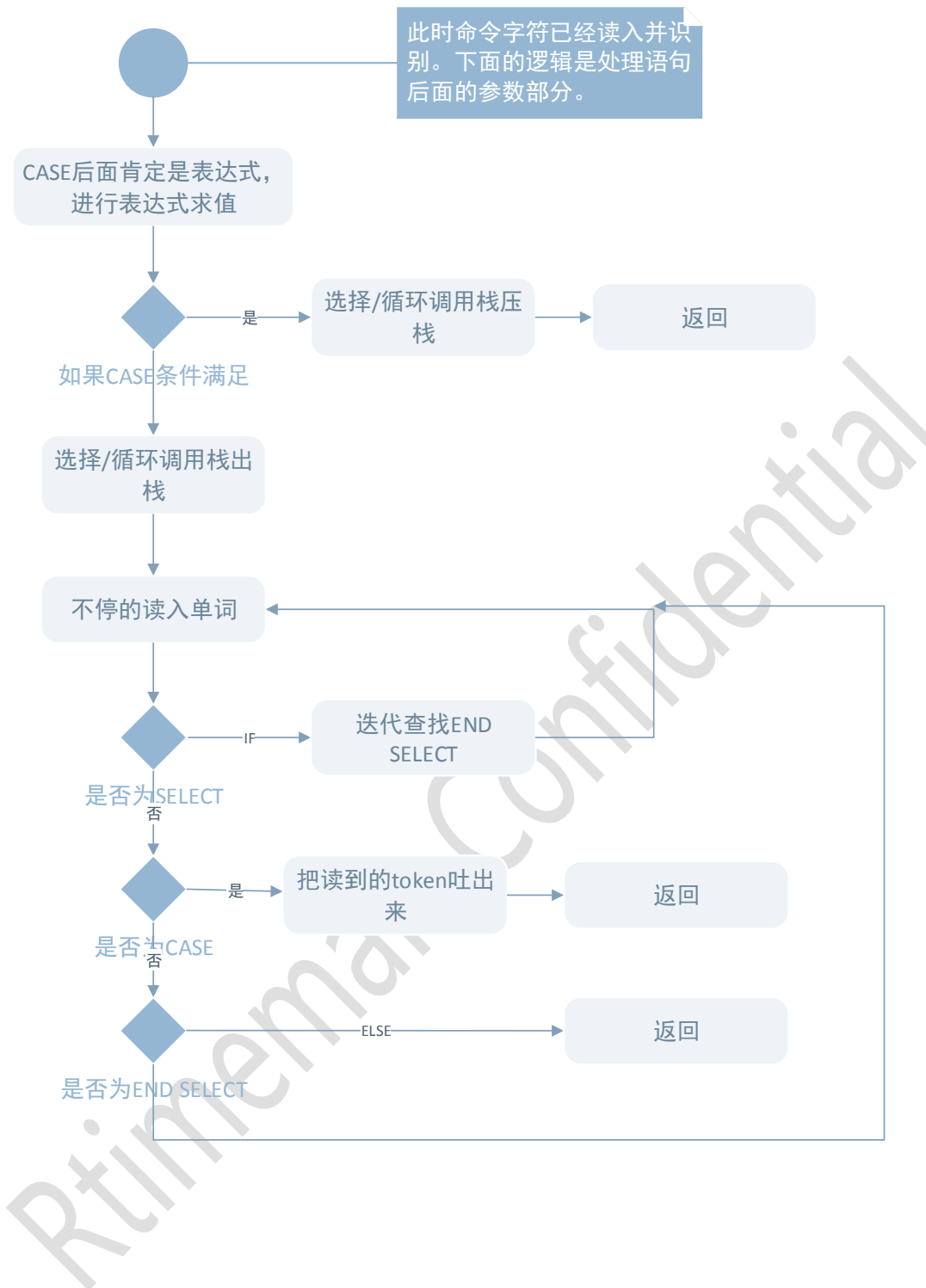
32 组件设计：程序命令 **select** 解析流程图

Select 语句的作用是记录用于 case 语句比较的变量。



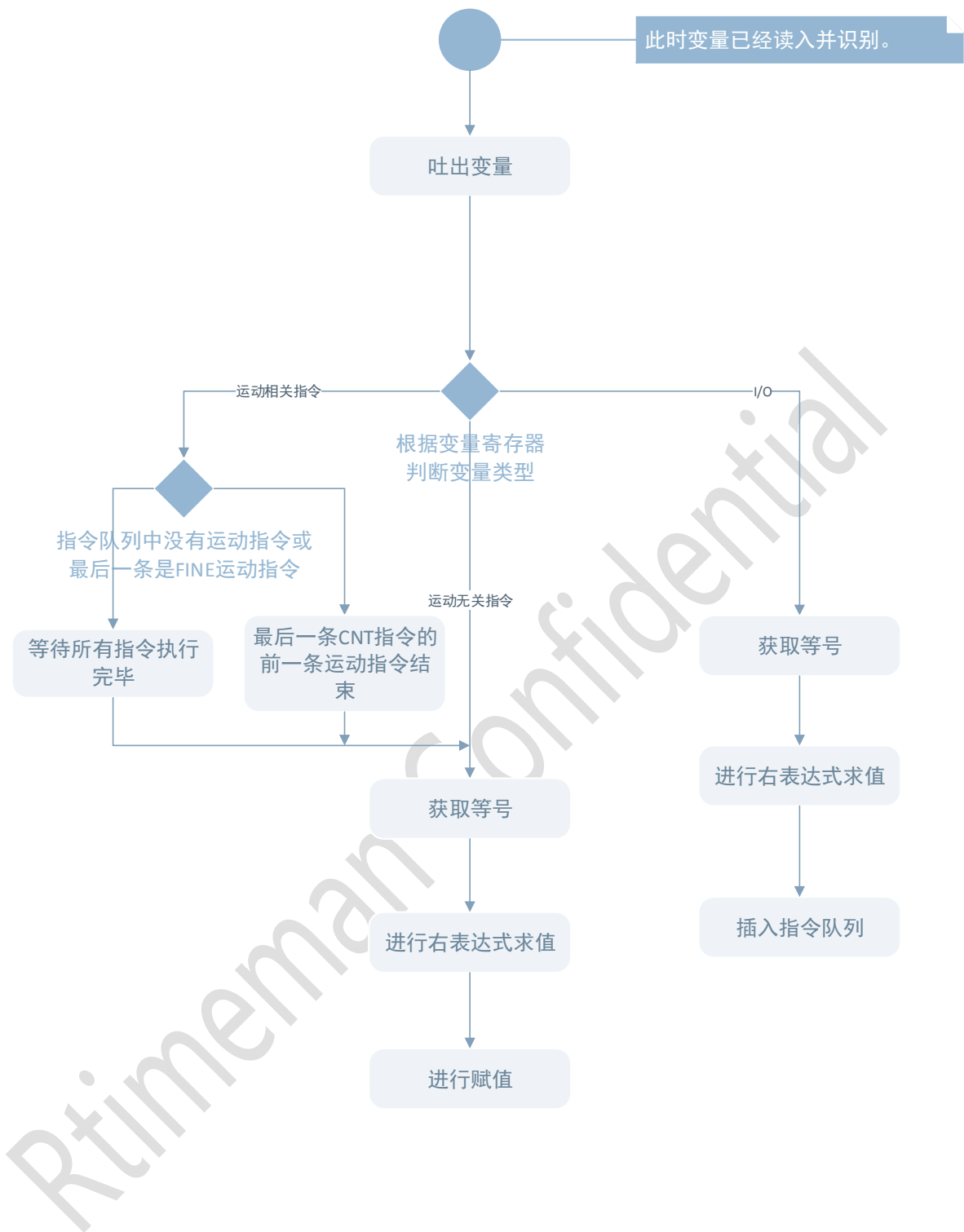
33 组件设计：程序命令 **case** 解析流程图

case 语句的作用是和 Select 语句变量进行比较。



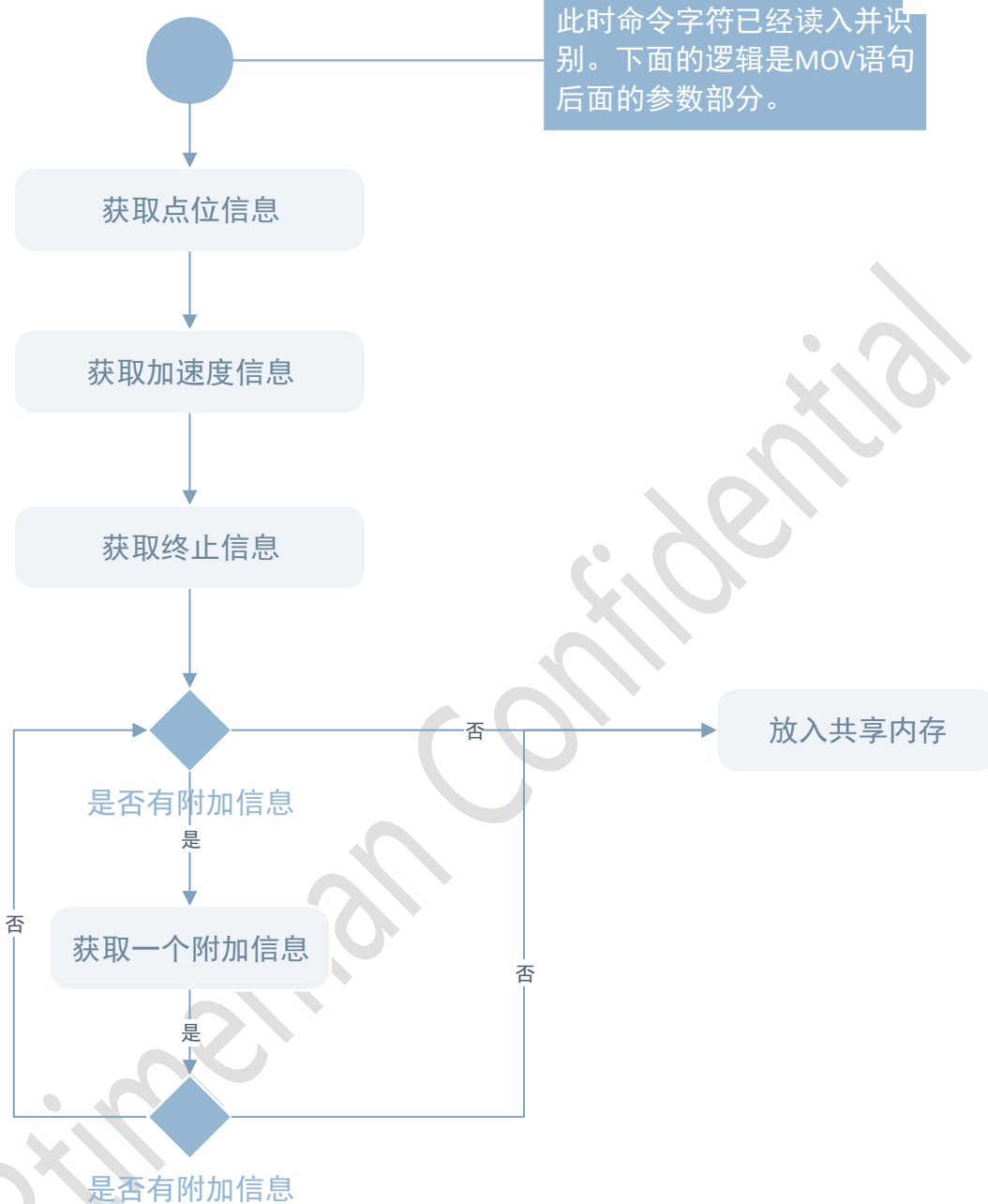
34 组件设计：赋值语句流程图

脚本解析器在读到变量的时候，会进入赋值流程。



35 组件设计：MOV 相关命令解析流程图

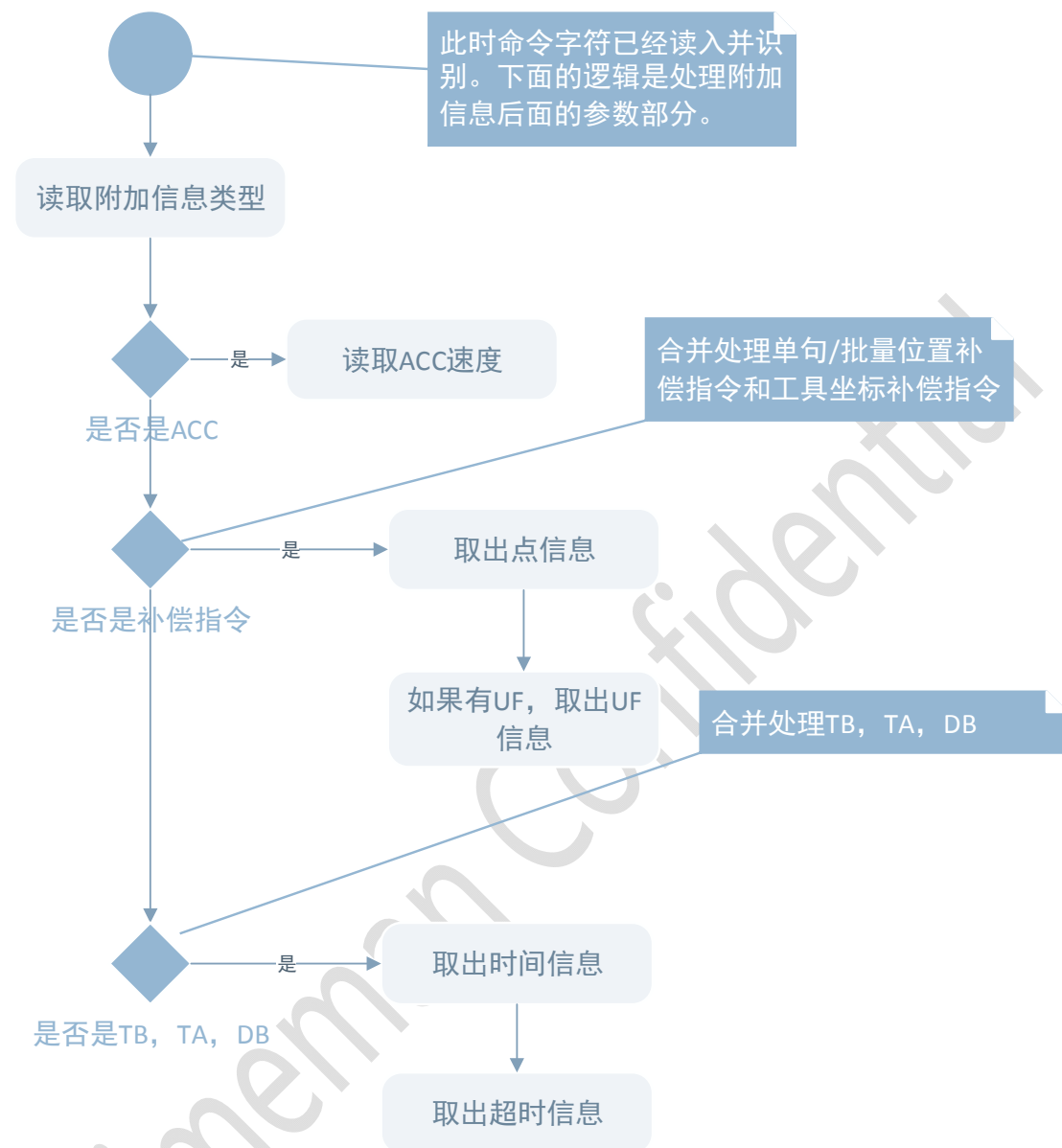
MOV 命令的参数依次为：点位，加速度，终止信息和扩展参数。



圆滑类型	圆滑参数	结果
CNT	大于零	速度圆滑
CNT	小于零	不圆滑
SV	大于零	速度圆滑
SV	小于零	不圆滑
SD	大于零	距离圆滑
SD	小于零	不圆滑

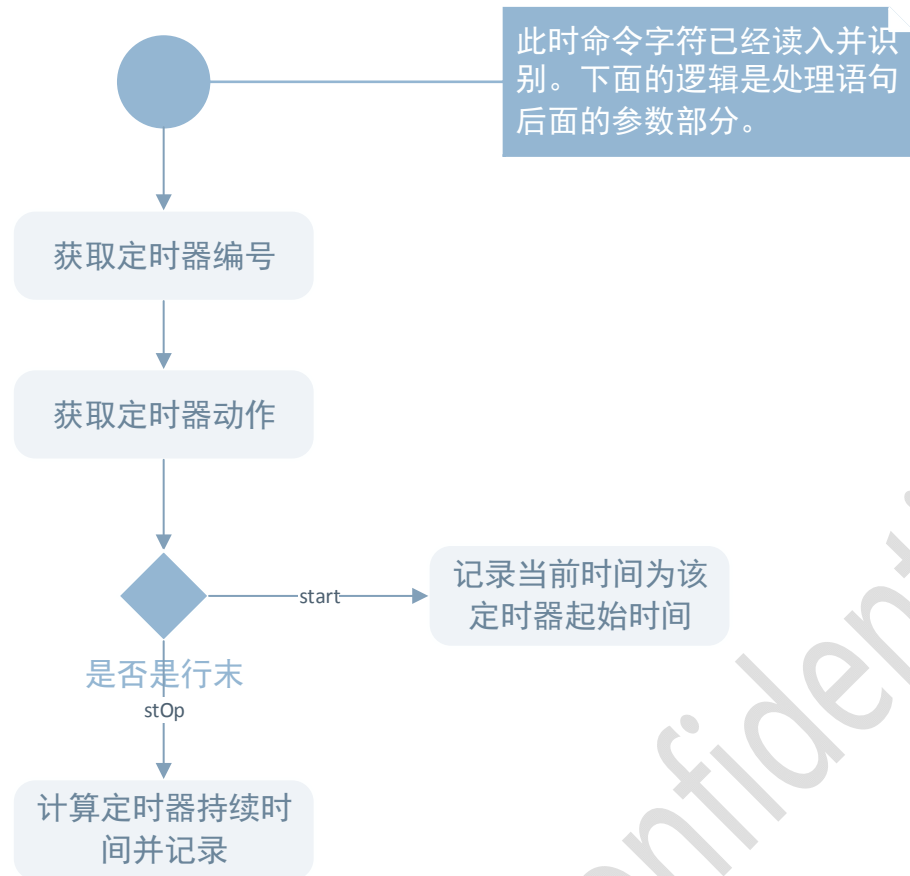
36 组件设计：MOV 相关命令扩展参数解析流程图

支持 ACC，位置补偿，TB/TA/DB 三种扩展参数。



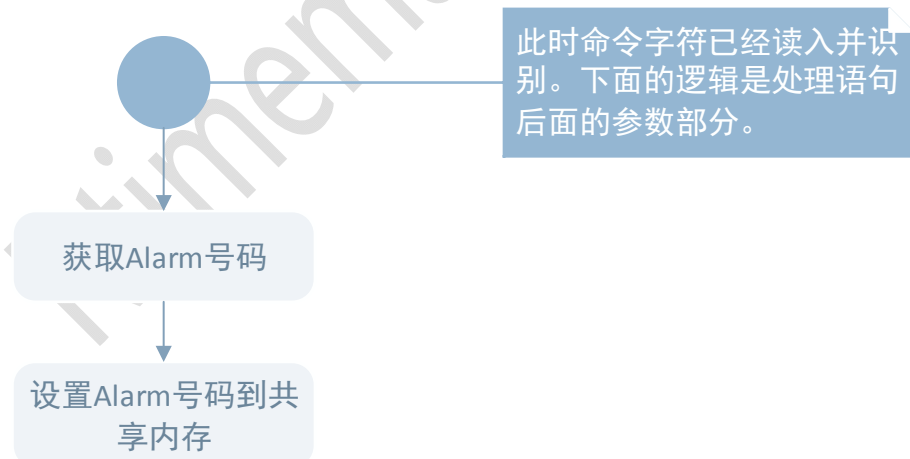
37 组件设计：Timer 命令解析流程图

实现方式就是实现一个秒表的功能。记录开始时间和结束时间的的时间差。



38 组件设计：UserAlarm 命令解析流程图

实现方式就是写入用户警告到控制器。



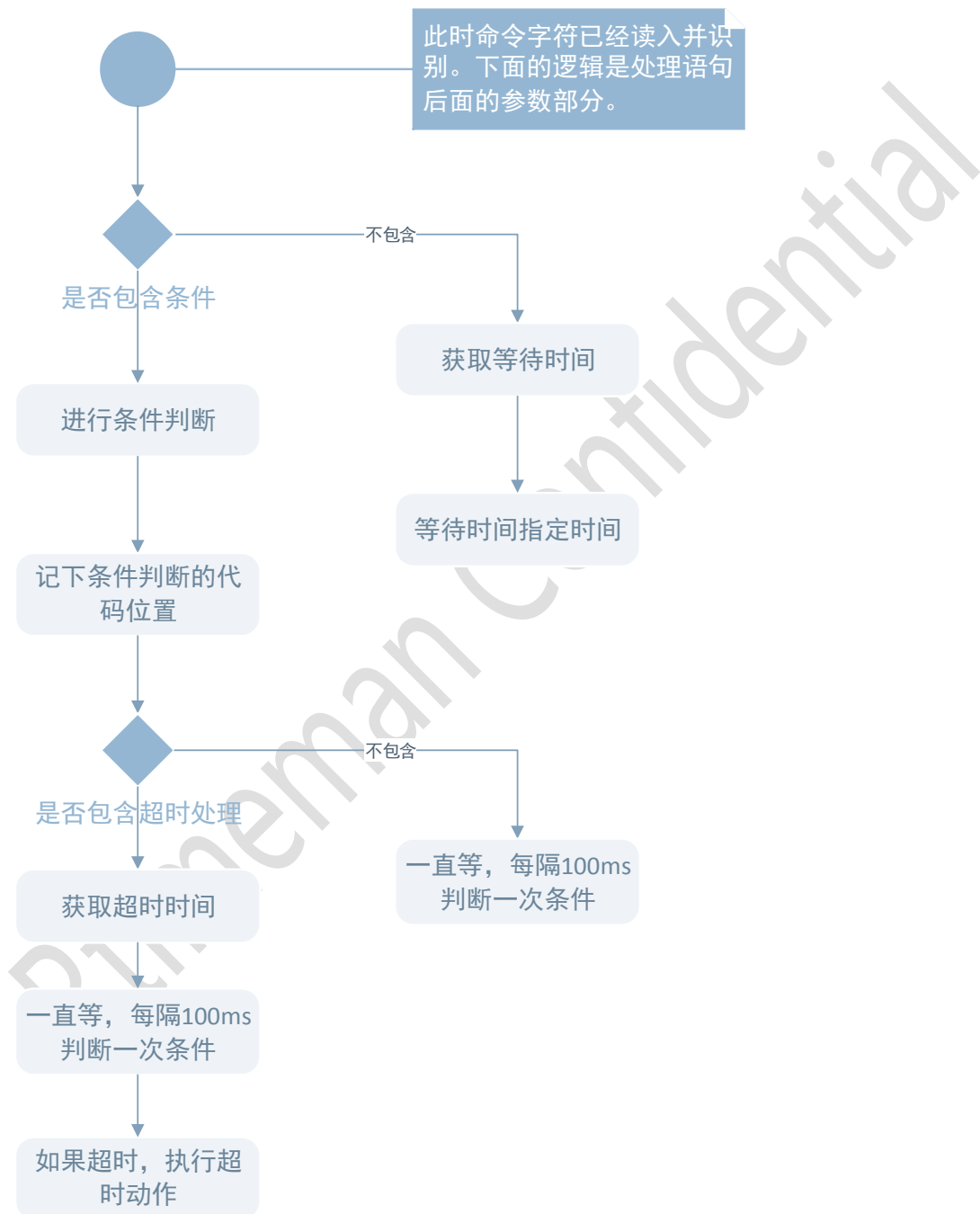
39 组件设计：Wait 命令解析流程图

在之前进行 XML 翻译的时候，会针对是否包含条件，决定是否添加关键字 COND。

WAIT 3

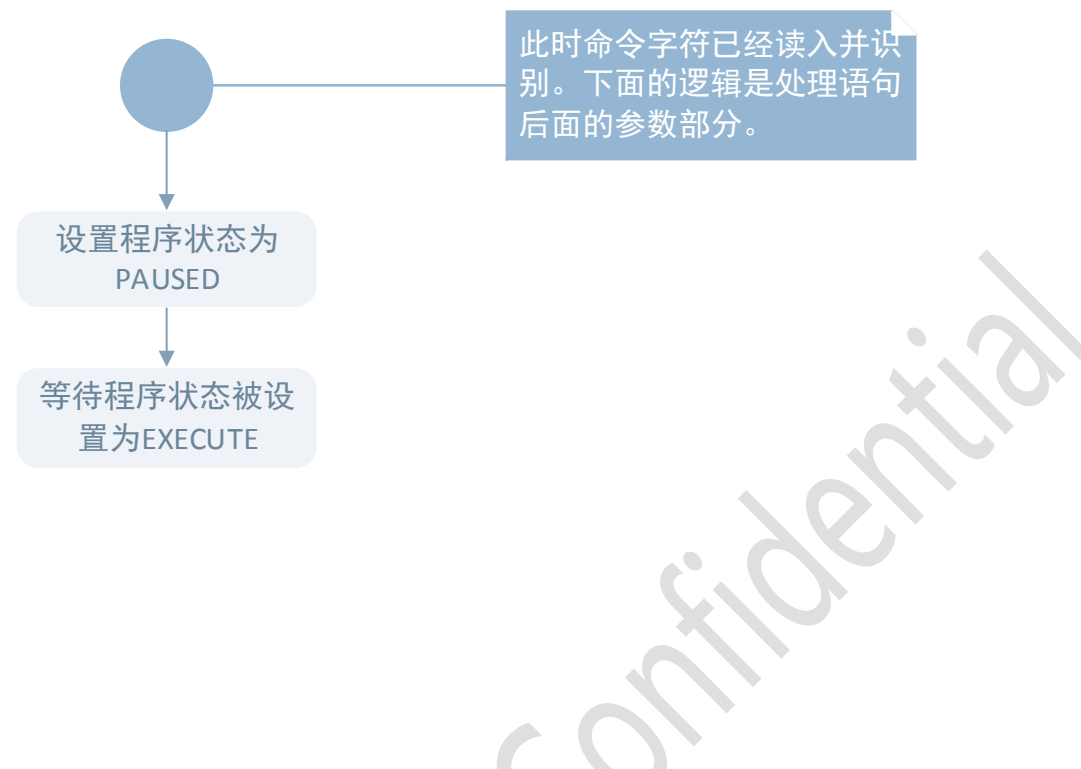
WAIT COND DI[1] = PE

WAIT COND R[1] > 10 3 skip



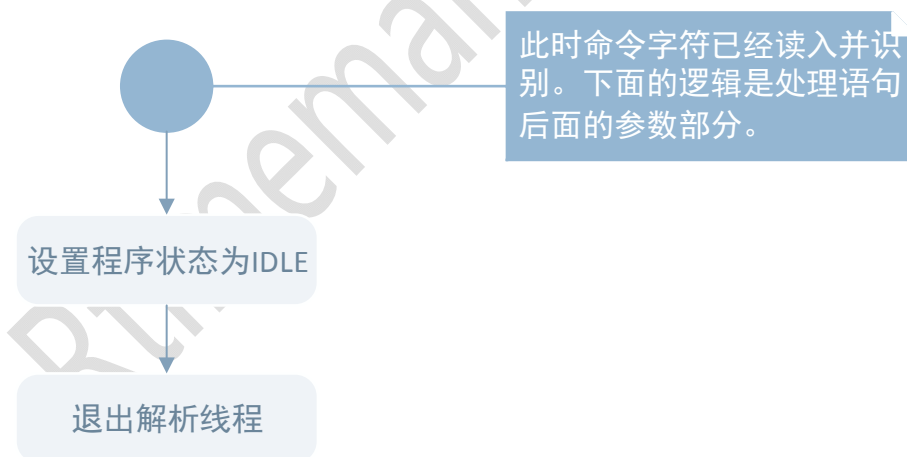
40 组件设计：Pause 命令解析流程图

实现方式就是设置程序状态为 PAUSED，暂停解析，等待控制器设置为 EXECUTE 后继续执行。



41 组件设计：Abort 命令解析流程图

实现方式就是停止解析。设置程序状态为 IDLE。



42 验证策略

目前的验证策略包括两种：一种是白盒测试，代码和解释器代码放在一起。另外一种黑盒测试：通过编写各种程序，执行后查看测试结果。

43 其它