Ucore操作系统在web环境上的实现与应用

uCompiler

编译(compile)，是将高级的语言程序，逐步转变为机器可识别的执行程序的过程。uCompiler是用于编译用c语言实现的ucore源码的编译器。相比较于常用的gcc编译器，其最后生成的执行程序所用的指令集是自定的v9-cpu指令集，所以不能直接被执行，需要用后面的模拟器来模拟执行。

uCompiler编译的过程分为三部分：词法分析，语法分析，指令生成。

词义分析

词义分析，是识别高级语言的原代码中所有保留字，常量（数字/字符/字符串），定义名称（结构名称/函数名称/变量名称），以及特殊符号，并将用token来替代的过程。

保留字,是在高级语言中已经定义的单词，一般只有固定的一种或者多种含义，并且不能用作别的用途，例如变量名称，宏定义等。

c语言的所有保留字，可分为如下几类。

|  |  |  |
| --- | --- | --- |
| 名称 | 特征 | 举例 |
| 基本类型 | 表示c语言中的基本类型 | int, char short float… |
| 逻辑跳转 | 根据表达式的结果进行跳转 | if while for switch goto |
| 逻辑跳出 | 跳到或跳出逻辑结构的末尾 | continue break |
| 类型定义 | 定义新类型名称 | struct enum typedef |
| switch跳转点 | 根据switch的索引表达式跳转的位置 | case default |
| 可变参数 | 可以省略或任意长度的函数参数 | va\_list va\_start va\_arg |
|  |  |  |

除了大小写单词字母以外，C语言还包含了其他的符号。

C语言中的所有特殊符号：

|  |  |
| --- | --- |
| 符号 | 用法 |
| 二元运算符，根据符号两边的表达式，计算结果 | |
| + - \* / % | 数学计算（整数/浮点） |
| & | ^ << >> | 逻辑运算 |
| && || ^^ | 布尔逻辑运算 |
| < > <= >= == != | 比较运算 |
| = += &= 等 | 赋值，包括直接赋值和计算赋值，返回赋值后的值 |
| . -> | 成员访问，左边是对象，右边是访问的成员名称 |
| 前缀运算符，在被运算的表达式左边 | |
| ! | if (expr ==0) return 1; else return 0; |
| ~ | 逻辑非 |
| \* | 地址寻对象 |
| & | 对象寻地址 |
| - | 0 - expr |
| (type) | 类型强制转换 |
| ++ -- | expr += 1/expr -= 1，返回运算后的值 |
| 后缀运算符，在被运算的表达式右边 | |
| [offset] | \*(expr+offset) |
| (args) | 函数调用 |
| ++ -- | expr += 1/expr -= 1，返回运算前的值 |
| 其他运算符 | |
| expr1?expr2:expr3 | if (expr1) return expr2; else return expr3; |
| ( ) | 括号 |
| 其他符号 | |
| {} | 大括号（函数内容/结构内容/枚举内容等） |
| () | 圆括号（函数参数等） |
| # | 宏 |
| … | 可变参数 |

uCompiler与一般编译器在词法分析上有很大的区别。主要区别是，在词法分析里，uCompiler自顶向下的构造了代码的语法树，来表示原有的逻辑结构。这样语法分析的大部分工作在词法分析中就已经完成了，后面的工作主要是检查和规范语法树。

uCompiler的语法树，是为了表达代码的逻辑结构而设计使用的一种数据结构。树中的每个结点代表代码中的一个逻辑单元。语法树的构成是自上而下的，根据当前状态来决定生成的策略。

语法树的生成在一般情况下都有通用的生成策略，大体流程分为以下几步。要注意的是，每一步之间可能有空格或者注释，一般情况下扫掉这些对于编译来说无用的信息，直到有用的逻辑代码为止。

1. 检查第一个单词，判断是否为保留字。若为保留字，则跳到新的与保留字对应的状态中。最后生成语法树为这个状态下的构成的语法树。
2. 检查是否为已经定义的类型/枚举/变量，注意类型也有可能成为变量名称，这种情况下通过前面的保留字struct来区分。
3. 若单词为空，那么通过接下来的符号来判断。例如#表示宏定义，(表示括号内的表达式/函数参数定义/函数调用参数等多种情况，根据当前状态来判断。
4. 若为非法符号，或者对当前状态不能识别的符号，通常情况下返回syntax error/expected编译错误。

|  |  |  |
| --- | --- | --- |
| 状态名称 | 定义 | 策略 |
| code | 代码的最外部 | 保留字struct（类型定义），enum（枚举定义），typedef（类型重命名），unsigned（无符号类型）  ，符号’#’（宏定义）。 |
| typedef | 类型重命名 | 一个已有的类型后面接着一个新的类型名称，注意保留字struct和unsigned |
| para | 函数内部的段落 | 分成stmt来生成，遇到}为结尾。 |
| stmt | 函数内部的语句 | 保留字包括if,while,for,switch等常用到的c语言结构，接下来判断是否为变量定义，否则跳到expr状态来检查是否为表达式。 |
| const | 通常值 | 可能是直接表示的整数，浮点数，字符，或者字符串。 |
| expr  expr类 | 表达式 | 这个比较复杂，下面会具体阐述 |
| class | 类型 | code状态中的首单词为类型时，跳转到的状态。后面根据特殊符号’{‘或者单词来判断类型定义或者变量/函数定义两种情况。 |
| enum | 枚举类的内部 | 第一个单词是枚举类中的名称，这里视为对常量的定义，类型为unsigned int。后面的符号可能为’=’,’,’或者’}’，也就是说枚举内部可能会自行赋值。 |
| type | 变量/函数定义 | 第一个单词为变量/函数名称，接下来通过判断是否为符号’(‘来辨别函数定义，’=’为变量定义的赋值，’;’为变量定义的末尾。 |
| f\_args | 函数参数 | 注意，先判断是否为可变参数…  类型名称和变量名称，后面跟着符号’,’或者’)’，前者表示后面还有参数，后者表示参数末尾。  其他符号均非法，返回syntax错误，下面遇到同样的情况也同样处理。 |
| array | 直接定义的数组 | 用{}来定义，不可修改。策略和para相同，检查’,’和’}’。 |
| fc\_args | 调用函数的参数 | 和f\_args大致相同，但是没有类型名称 |
| fp\_args | 函数指针的参数 | 同fc\_args |
| c\_code | 类型定义内部 | 定义类型内部的成员。在构造策略上是code的子集，只包括成员变量的定义。 |
| s\_code | switch内部 | 在构造策略上是para的子集，不包括变量定义。 |

这里着重强调expr和expr类的构造策略，这里占用了整个构造过程将近一半的篇幅。expr类主要是在expr状态的基础上，判断一些特殊的情况而定义的，包括exprwc(expression without comma，不包含括号运算符的表达式)，exprcase(switch语句中case后的表达式，主要是为了判断末尾符号’:’)。

expr的构造通过单词-运算符之间的交替识别进行的，这里没有考虑逻辑结构，但是识别了类型转换，常量变量等一部分功能。剩下的部分在之后的语法分析过程中进行处理。循环执行如下几步，直到遇到末尾符为止。

1. 检查是否为左括号’(‘，分为子表达式或者类型转换两种情况，检查括号内部的结构，若为(typename)或者(struct typename)这样的结构，则确定·是类型转换，重复步骤1，否则为子表达式，递归调用expr状态。
2. 检查是否为数字，’或者”。分别代表了数字常量/字符常量和字符串常量，调用const状态来生成。
3. 检查是否为单词，单词是否已经定义（变量/函数），若未定义，返回uncleared，若后面跟着’(‘符号，则为函数调用，否则为变量取值。
4. 检查是否为运算符，最长的运算符优先匹配(<<=优先,<<其次，最后才是<)。

将以上过程识别的运算符元素存放在一张表中，之后的语法分析会继续整理成树形逻辑结构。

通过以上所述的构造方法，最后能够生成表达原代码的完整逻辑的语法树，节点通过js特有的无类型object来顶定义，这样可以省去很多节点类型定义的麻烦。

|  |  |  |
| --- | --- | --- |
| 名称 | 解释 | 结构 |
| 基本层/结构层/枚举层 | | |
| class | 结构定义 | text:定义内容，next:下一个结点 |
| var | 变量定义 | name:名称，s\_type:类型，value:赋值，next:下一个结点 |
| func | 函数定义 | name:名称，s\_type:类型，args:参数，text:函数内容  next:下一个结点 |
| enum | 枚举定义 | name:名称，text:内容，next:下一个结点 |
| typedef | 类型重命名 | s\_type1:类型名称，s\_type2:新名称，next:下一个结点 |
| 函数层 | | |
| expr | 表达式 | text:表达式序列，next:下一个节点（可省略） |
| CONST | 常量（大写表示表达式内容，下同） | s\_type:类型，value:值 |
| VAR | 变量 | name:变量名 |
| FUNC | 函数调用 | name:函数名，args:参数 |
| … | 运算符 | 无 |
| CONV | 类型装换 | s\_type:装换类型 |
| para | 用{}包围的代码段落 | bracket:括号名，text:段落内容，next:下一个结点 |
| if | 条件语句 | cond:条件表达式，stmt:条件正确时执行的语句，  else:条件错误时执行的语句，next:下一个结点 |
| while | while循环 | cond:条件表达式，stmt:执行语句，next:下一个结点 |
| for | for循环 | s\_expr:循环开始时执行的表达式，c\_expr:条件表达式，  e\_expr:循环末尾执行的表达式，  stmt:循环执行的语句，next:下一个结点 |
| switch | 多路条件 | cond:条件表达式，text:结构内容，next:下一个结点 |
| case | 跳转点 | value:常量,为条件坐标，next:下一个结点 |
| default | 默认跳转点 | next:下一个节点 |
| break | 跳出 | b\_point:跳出点（一般为父结构的下一条语句），  next:下一个结点 |
| continue | 跳过 | c\_point:跳过点（一般为循环末尾），  next:下一个结点 |
| asm | 内嵌汇编 | inst:指令，imme:立即数，  next:下一个结点 |
| return | 返回 | value:返回值，  next:下一个结点 |
| 宏 | | |
| include | 头文件包含 | filename:文件名，text:语法树 |
| define | 宏定义 | text1:匹配文本，text2:替换文本 |
| 可变参数相关 | | |
| va\_arg\_list | 可变参数 | 无 |
| va\_start | 指向第一个参数的指针 | ap:要赋值的指针，v:参数列表 |
| va\_arg | 下一个参数 | ap:要赋值的指针，t:参数类型 |
| va\_end | 参数指针设为无效 | ap:要赋值的指针 |

要注意的是，词法分析生成的语法树并不完善，例如break和continue的跳转点，以及将表达式的序列结构转换为逻辑结构，这些工作都是需要接下来的语法分析来完成的。

在词法分析的过程中，可能会检查并报出编译错误，并且大多数编译错误都将导致接下来的分析工作无法继续进行。例如语句结尾的’;’符号无法识别，那么无法判断下一条语句的起始位置。为了解决这个问题，在uCompiler里引入了括号匹配原则，在发生编译错误时，进行括号匹配寻找当前括号的结尾，然后继续进行词法分析。

列举出所有在词法分析时可能发生的错误：

|  |  |
| --- | --- |
| 错误 | 注释 |
| expected char (after stmt) | （对于对应的语句）找不到匹配的符号 |
| syntax at char | 不合法的符号 |
| word uncleard | 未定义的单词 |
| expr is not a constant | 在需要常量的地方使用非常量 |
| unknown operator at char | 表达式内部使用未定义的运算符 |
| a label can only be part of a statement and a declaration is not a statement | 在switch内部不能声明变量 |
| include : ‘file.h’ not found | 未找到头文件 |

遭遇到编译错误时，编译器会显示发生所在的行号和列号，并将源代码整行和错误信息显示在浏览器上。否则，在未发生词法分析的错误那么词法分析就通过了，得到的是一个基本完善的表示程序逻辑的树结构。

语法分析

语法分析，是将获得的单词token进行逻辑组合，检查是否合法，并生成表达式，语句，函数等逻辑结构。目前流行的两种语法分析策略分别为自上而下（LL）与自下而上（LR）。

uCompiler的语法分析中，除了表达式以外，其他结构的逻辑结构已经分析完毕，所以语法分析所要做的主要工作如下：

1. 检查表达式的合法性，从最高优先级运算符开始，自下而上的检查表达式是否正确，并且构造表达式的语法树。
2. 类型运算和类型转换的合法性。浮点数不能位运算和取模，指针地址不能除了加减以外的其他运算。这些需要对表达式自下而上的判断计算结果的类型。
3. 对于break和continue未查询的跳转点，在语法分析里找到并设置。
4. 检查函数调用的参数是否匹配，可能会出现参数数量不一致和类型不一致等情况。

上面重点实现的表达式识别，这里继承了上面的工作，将表达式序列重新组合成一个表达式的语法树。但是在构造表达式的语法树之前，需要进行各种预处理，包括之间因为运算符的多重含义，所要做的运算符重新识别。包括’\*’的两个含义(乘法，指针地址的实例)，’&’的两个含义（与运算，取地址），’-’号的两个含义（减法，取反）等。但幸运的是，这些运算符的两个用法其中一个是左右二元的，另一个是前缀一元的的。这样判断前面的是否是’)’以外的运算符就可以解决了。

通过预处理之后，我们确定了每个运算符和子表达式都是有效正确的，就可以开始表达式树的生成了。运算符按照优先权（运算符优先，从左到右顺序优先）来进行扩展，二元运算符扩展左右两个节点，一元运算符只扩展一个节点，三目条件运算符需要扩展条件和两个分支，数组运算符需要扩展[]内的内容。

自下而上的检查并完善每个语法树的结点，最后保证能够生成一个正确有效的语法树。这里可能出现的编译错误如下：

|  |  |
| --- | --- |
| 错误 | 解释 |
| expected expression after op token | 运算符后需要表达式 |
| expected expression before op token | 运算符前需要表达式 |
| expected ‘?’/’:’ | 条件运算符缺省符号 |
| could not convert from type1 to type2 | 类型转换不合法 |
| fixed-point types not supported for this target | 访问非结构类型的成员 |
| expected struct number after variable | 结构类型不包含该成员 |
| invalid type argument of '->' | 非指针使用->符号 |
| request for member in something not a structure or union | 指针指向的类型非结构类型 |
| invalid operands to binary op  (have type1 and type2) | 对type1和type2的运算op是非法的 |
| wrong type argument to increment | 对结构类型使用++,--运算符 |
| array subscript is not an integer | 数组运算符的索引值不是一个整数 |
| subscripted value is neither array nor pointer nor vector | 数组运算符的基地址不是一个指针 |
| used struct type value where scalar is required | 结构类型不支持数字运算 |
| wrong type argument to bit-complement | 非整数类型不支持位运算 |
| too few arguments to function | 调用参数比函数参数要少 |
| too many arguments to function | 调用参数比函数参数要多 |
| case label not within a switch statement | case语句不在switch结构里 |
| default label not within a switch statement | default语句不在switch结构里 |
| break statement not within loop or switch | break语句不在循环结构或switch结构里 |
| undifined instruction | 内嵌汇编未定义指令 |

对程序的语义检查完毕后，就认定这是一个合法的语言程序，也就可以转换为可执行的指令程序了。

指令生成

编译的最后一步，需要用到中间得到的信息，生成最后可执行的目标代码。在这之前，通常编译器需要生成中间代码和代码优化，uCompiler省略了这一步，直接生成了执行程序。通过自上而下的访问语法树，生成每个结构结点所对应的机器语言。

v9-cpu指令集是在简化了的内核系统v9-cpu上执行的一套机器指令集，虽然在寄存器数量和功能上处于劣势，但换来的是指令的具体化和多样化（因为寄存器少，所以一条指令不需占用太大空间表示寄存器标号，这样寄存器信息可直接包含在指令类型里）。

由于v9-cpu指令集中指令较多，在此只做分类说明。所有指令请查看同目录下的cpu.md.

|  |  |  |
| --- | --- | --- |
| 类别 | 特征 | 举例 |
| 系统指令 | 系统中断，对sp和pc 的直接操作，对内存连续一段的操作 | HALT,ENT,JSR, MCPY |
| Load指令 | 从内存中读取数据，可以是任意基本类型，可以读取到ra,rb,rf,rg中。以及将立即数存到寄存器里。 | LL, LBL，LLS(short),LLB(byte),LG, LX, LI, |
| Store指令 | 将ra,rf中的值，储存到内存中。可以以任意基本类型的形式存储。 | SL, SLS, SLB, SG, SX |
| Branch指令 | 根据寄存器值来跳转 | BZ, BNZ, BLT, BLE, BGT,BLTF |
| 类型转换指令 | 32位整数与64位浮点之间的转换 | CID,CUD,CDI,CDU |
| Misc指令 | 涉及到特权级的切换，以及栈的操作 | CLI,PSHA,POPA |
| Math指令 | 浮点运算，包括指数对数，三角函数，取整等 | LOG,FLOR,SIN |

执行程序的生成遵循一定的方法，根据程序结构的不同分为：

|  |  |  |
| --- | --- | --- |
| 结构 | 例子 | 方法 |
| 常量 | 1, 0.1, ‘c’, “string” | 直接用（立即数/数据段地址）来表示 |
| 变量 | int I, j, char\* c | 根据变量地址从内存读写对应数据 |
| 运算 | i+j, ~i, expr1+expr2 | 1. 常量参与的运算，用立即数指令来完成。 2. 先计算子表达式的值，放在寄存器里，再用指令运算。 3. 二元运算若有必要，将左表达式的值暂存于栈中，计算右表达式后用栈读取运算指令。 4. 布尔与或运算的断短路问题，通过b指令跳过之后的运算。 |
| 赋值 | var = expr | 将表达式的值计算于寄存器，用s指令写入内存。 |
| 成员 | struct.number  pointer->number | 结构地址+成员偏移量  指针地址+成员偏移量 |
| 寻址 | \*pointer | 计算pointer的表达式，在根据结构读出数据 |
| 取值 | &var | 根据编译信息直接得到 |
| 调用 | func(args) | 将参数压栈，JSR跳转至所调用函数，返回出栈 |
| 条件 | if (expr) stmt  if (expr) stmt else stmt  while (expr) stmt  for (e1;e2;e3) stmt | 1. if：计算expr，betz跳转至下一条语句。 2. if-else：计算expr，betz跳转至else语句，then语句末尾跳转至下一条语句。 3. while：跳转至循环末尾，末尾计算expr，bnez跳转至循环头部。 4. for：同while，循环前执行e1,循环末尾前执行e3。 |
| switch | switch(expr) {  case 1:  case 2:  …  default:  } | 1. 将所有case的值进行排序。 2. 得到最大值，若不超过一定阈值，则可以用entrylist来实现，否者只能用通过二分法的策略来查找入口。 |
| 跳转 | break, continue, goto | 直接用跳转指令 |
| 返回 | return value | 计算value于寄存器ra，LEV 指令返回 |
| 汇编 | asm(inst,imme) | 直接用对应的指令 |
|  |  |  |
|  |  |  |

最后生成的执行程序，就可以直接交给模拟器uEmulator来运行了。

uEmulator

uEmulator是模拟v9-cpu计算机模型运行，并且能够支持调试功能的仿真软件。若要使用uEmulator的调试模式，需要让uCompiler生成支持调试的DML文件（Debug Markup Language），DML文件包含每行代码的内容，以及所对应的机器指令的相对地址。

uEmulator读取程序和DML文件，经过二步流程来准备调试工作。

1. 分析DML文件，标记出源代码每行对应的指令位置。
2. 用状态机统计所有的全局变量，所有函数的局部变量和参数。

状态机的状态转移机制比较繁琐，故只列出每个状态的含义。

|  |  |
| --- | --- |
| 状态(hex表示) | 解释 |
| 0xXXDD | DD表示当前括号匹配的层数，括号包括圆括号和大括号，视为同一种括号来匹配。 |
| 0x01XX | 表示当前在字符/字符串内部 |
| 0x02XX | 表示当前在函数参数声明的内部 |
| 0x04XX | 表示当前在函数代码的内部 |
| 0x0800 | 表示当前在类型构造的内部 |
| 0x1000 | 表示当前在枚举构造的内部 |
|  |  |

有了状态机之后，就可以围绕这个状态机来识别原代码中的各类结构了。

|  |  |  |
| --- | --- | --- |
| 结构 | 所在状态 | 识别方法 |
| 函数参数 | 0x0201 | 检查’,’符号，获取前面两个单词，第一个是参数类型，第二个是参数名称。  （遇到指针’\*’时，可以跳过’\*’前面的空格。） |
| 局部变量 | 0x04XX（非0） | 检查’,’,’=’,’;’符号，获取前面两个单词，若能匹配第一个单词的类型，第二个单词是合法的变量名称（数字字符下划线，第一个字符是字符或下划线），那么这就是个变量定义。  ‘,’之后只声明了变量名称，所以只需检查一个单词。 |
| 结构内容 | 0x0801 | 检查’,’,’;’符号，之后和检查局部变量相同。 |
| 全局变量 | 0x0000 | 与检查结构内容相同。 |
| 常量 | 0x0000 | 检查’const’单词，并匹配之后的两个单词（类型和名称），一个’=’符号，一个数字，字符或字符串。 |
| 枚举内容 | 0x1000 | 视为常量，和检查变量的方式相同。 |
|  |  |  |

通过这种策略，在状态机工作完毕之后，就可以得到与源代码结构相对应的数据结构了。

|  |  |  |
| --- | --- | --- |
| 名称 | 意义 | 结构 |
| var\_struct | 变量结构 | type:类型，name:名称 |
| func\_struct | 函数结构 | name:名称, addr:指令地址，  start:代码起始行，end:代码末尾行  args:参数列表，locs:局部变量列表 |
| const\_struct | 常量结构 | name:名称，value:数值 |
| stu\_struct | 类型结构 | Name:名称，size:空间大小，  nums:成员列表（var\_struct） |
| stmt\_struct | 原代码语句结构 | start:语句起始行，end:语句末尾行，  text:语句内容 |
| Backtrace\_struct | 函数调用路径 | Func:函数编号，stmt:调用所在行，  Sp:栈帧 |
| Watchpoint\_struct | 视点结构 | Ltype:左类型，lvalue:左数值，  Rtype:右类型，rvalue:右数值  Optype:运算类型 |

其中watchpoint中的两个数值可以是整数，可以是指针地址。根据类型的不同有不同的含义：

type=0，value表示一个有符号的32位整数。

type=1，value指向一个watchpoint\_struct,用来表示全局变量，lvalue为变量var\_struct的地址，rvalue为变量在内存中的地址。

Type=2，value指向一个watchpoint\_struct用来表示局部变量，lvalue为变量var\_struct的地址，rvalue为变量相对于当前栈帧的偏移地址(offset)。

Type=3，value指向一个

当uEmulator开始调试程序时，用户可以用类似于gdb的方式，通过输入指令来实现各种调试功能。

|  |  |
| --- | --- |
| 命令 | 功能 |
| c | 继续运行，直到到达第一个断点停止。 |
| s | 单步运行，运行一行代码 |
| si | 单步运行，运行一条指令 |
| sc | 单步执行到下一行，跳过调用 |
| b addr  b name  b name+line | 按照指令的相对地址，设置断点  按照函数名称设置断点  按照函数名称+行号，设置到所在函数的行。 |
| b if expr  b if expr! | 设置当expr值为1时暂停的条件断点  设置当expr的值改变时的监视断点 |
| bl  bd index  bdall | 查看所有断点  删除对应序号的断点  删除所有断点 |
| bil  bid index  bidall | 查看所有条件断点  删除对应序号的条件断点  删除所有条件断点 |
| bt | 查看函数调用栈 |
| ir  il  ig | 显示所有寄存器的信息  显示函数内局部变量和参数的信息  显示所有全集变量的信息 |
| x /48i 0x0080  x /32i pc  x /64x 0x0100  x /32x sp | 显示0x0080后48bit的命令  显示pc后32bit的命令  显示0x0100后64bit的数据(hex显示，下同)  显示栈顶后32bit的数据 |
| h | 显示帮助文档 |
| q | 退出uEmulator |
|  |  |
|  |  |

模拟器执行一条指令的周期分为多步，在此期间会进行检查调试信息，检查断点条件等调试工作。

1. 根据PC地址，读取一条机器指令。
2. 检查PC地址是否到达某一个断点，若存在到达的断点，则打开调试开关。
3. 检查所有条件断点是否符合条件，若存在符合的断点，则打开调试开关。
4. 当调试开关打开时，进入调试模式，读取用户输入的命令，根据命令执行相应的操作。
5. c命令（继续运行）或s命令（单步运行）会继续运行模拟器，其他命令到最后会进入循环。
6. 根据读取的机器指令，进行相应的操作。不涉及PC跳转的情况下会跳转到步骤（1），否则跳转到步骤（7）。
7. 对PC做调整，包括调整为4的倍数等。同时更新函数调用栈的栈顶。最后循环整个步骤。

支持调试功能并没有用到高级的数据结构，两类断点均用列表来存储和修改，还有在外部维护的函数调用栈。所有数据结构的用途和实现方法会在接下来的篇幅中去介绍。

|  |  |
| --- | --- |
| 名称 | 功能 |
| 结构树 | 表示c语言中struct内部的内容。  内部设有成员列表，每个元素包含成员的类型和名称。  通过自上而下访问可以实现对结构的访问和显示。 |
| 断点列表 | 保存和维护所有的地址断点和条件断点。  支持末尾添加功能和通过整体位移一格的定点删除功能。 |
| 函数调用栈 | 维护函数调用的Backtrace. 主要包括前面调用的函数的id,代码位置和栈帧顶部。  检测机器指令JSR和LEV，出现这两个指令时，说明执行了函数调用和跳出，对相应的Backtrace进行push和pop操作。  特别是在函数调用（JSR）中，通过调用的入口在函数表中查找对应的函数id，并加入函数栈。 |
| 函数表 | 其中包括名称，行首，行末，参数列表，局部变量列表。  可以先通过指令地址在语句表里查找对应的行数，然后再通过函数表查找函数和函数所在行。 |
| 语句表 | 解析DML文件后生成的每条语句的相关信息。  包含了每条语句的指令起始点和末尾。 |
|  |  |
|  |  |