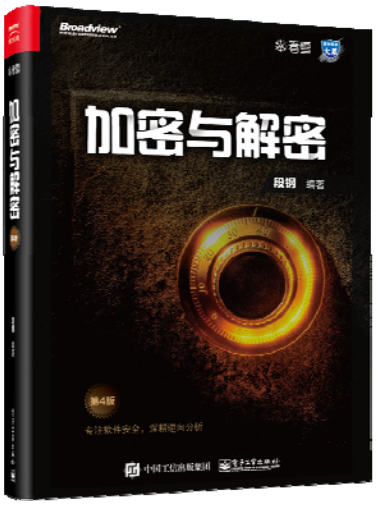
附　录



目 录

附录 A 浮点指令

一般的汇编书籍对浮点指令的介绍很少，在这里简单介绍一下浮点数。

A.1 浮点数据格式

在计算机中，表达实数时要采用浮点数格式。Intel 80x87 遵循 IEEE 浮点格式标准。IEEE 浮点数的格式由符号位 S、指数部分 E 和尾数部分 M 组成，如图 A.1 所示。

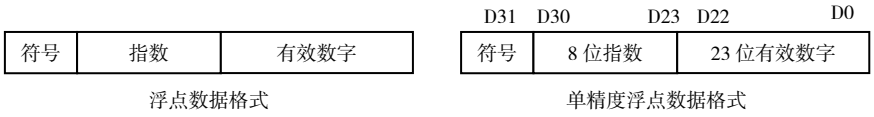


图 A.1 浮点数据格式

- 符号（Sign）：表示数据的正负，在最高有效位，负数的符号是 1，正数的符号是 0。
- 指数（Exponent）：或称“阶码”，表示数据以 2 为底的幂。指数采用偏移码表示，恒为整数。例如，单精度格式 8 位指数的偏移基数为 127，除去全 0、全 1 两个编码外，1 ~ 254 编码表示阶码数值 -126 ~ 127。
- 有效数字（Significand）：表示数据的有效位数，反映了数据的精度。
  - 单精度浮点数据格式是 32 位，符号位占 1 位，E 占 8 位，M 占 23 位。
  - 扩展单精度格式，符号位占 1 位，E 大于或等于 11 位，M 占 31 位。
  - 双精度浮点数据格式是 64 位，符号位占 1 位，E 占 11 位，M 占 52 位。
  - 扩展精度浮点数据格式是 80 位，符号位占 1 位，E 大于或等于 15 位，M 大于 63 位。

1．把浮点格式数据转换成实数表达式

设单精度浮点数的符号位是 s，指数是 e，有效数字是 x，则该浮点数的值用十进制表示为

$$(-1)^{s} \times (1+x) \times 2^{(e-127)}$$

例如，49E48E68 h = 0100 1001 1110 0100 1000 1110 0110 1000 b，将它分成符号、指数和有效数字 3 部分，结果为

49E48E68 h = 0 100 1001 1 110 0100 1000 1110 0110 1000 b

- 符号位是 0，即 $s = 0$。
- 指数部分是 100 1001 1，转换成十进制就是 147，即 $e = 147$。
- 有效数字部分是 110 0100 1000 1110 0110 1000，也就是二进制的纯小数 0.110 0100 1000 1110 0110 1000，其十进制形式为 0.78559589385986328125，即 x = 0.78559589385986328125。

这样，该浮点数的十进制表示如下。

$$(-1)^{s} \times (1+x) \times 2^{(e-127)} = (-1)^{0} \times (1+0.78559589385986328125) \times 2^{(147-127)} = 1872333$$

2．把实数转换成浮点格式

例如，100.25 = 0110 0100.01 b = 1.10010001 b×2^6。

- 符号位为 0。
- 指数部分为 127+6 = 133 = 10000101 b。
- 有效数字部分为 10010001000000000000000 b。

这样，将 100.25 表示成单精度浮点数，为

0 10000101 10010001000000000000000 b = 42C88000 h

A.2　浮点寄存器

组成浮点执行环境的寄存器主要是 8 个通用数据寄存器和几个专用寄存器，它们是状态寄存器、控制寄存器和标记寄存器。

1．浮点数据寄存器

浮点处理单元有 8 个浮点数据寄存器（FPU），编号为 ST0 ~ ST7。OllyDbg 寄存器面板显示的浮点寄存器如图 A.2 所示。每个浮点寄存器都是 80 位，并以扩展精度格式存储数据。8 个浮点寄存器组成首尾相接的堆栈，不采用随机存取，而是按照“后进先出”的堆栈原则工作，且首尾循环。所以，浮点寄存器常被称为浮点数据栈。

```
ST0 valid 4.0000000000000000000
ST1 empty 0.000000000004535328 0e-4933
ST2 empty 0.000000000000023290e-4933
ST3 empty 7.077798931489875331 0e-2505
ST4 empty -NAN FFFF 804E6760 804E3490
ST5 empty -UNORM 8A68 0000003B A820FC38
ST6 empty +UNORM 003B 0012FDC0 00000000
ST7 empty 16.000000000000000000
```

图 A.2　OllyDbg 寄存器面板显示的浮点寄存器

2．浮点状态寄存器

浮点状态寄存器表明 FPU 当前的各种操作状态，每条浮点指令都对它进行修改以反映执行结果，其作用与整数处理单元的标志寄存器 EFLAGS 相同，如图 A.3 所示。

15	14	13~11	10	9	8	7	6	5	4	3	2	1	0
B	C3	TOP	C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

图 A.3　浮点状态寄存器

C3 ~ C0 是 4 位条件码标志。其中，C1 表示数据寄存器栈出现上溢或下溢，C3、C2 和 C0 保存了浮点比较指令的结果。

A.3　浮点操作

下面介绍几个常用的浮点指令。

1．取数指令

取数指令从存储器或浮点数据寄存器中取得数据，将其压入寄存器栈顶 st(0)。压栈操作使

栈顶指针的值减 1，数据进入新的栈顶 st(0)，原来的 st(0) 成为现在的 st(1)，原来的 st(1) 为现在的 st(2)，依此类推。数据进入寄存器栈前，由浮点处理单元自动转换成扩展精度浮点数。

- FLD m32r/m64r/m80r/ st(i)：取存储器或 st(i) 中的浮点数，压入栈顶 st(0)。
- FILD m16i/m32i/m64i：取存储器的整数，压入栈顶 st(0)。

2．存数指令

存数指令除执行存数功能外，还要弹出（pop）栈顶。出栈操作是将栈顶 st(0) 清空，并使栈顶指针的值加 1，原来的 st(1) 成为现在的 st(0)，原来的 st(2) 为现在的 st(1)，依此类推。出栈指令的助记符都以“P”结尾。

- FSTP m32r/m64r/m80r/ st(i)：st(0) 按浮点格式存入存储器或 st(i)，然后出栈。
- FISTP m16i/m32i/m64i：st(0) 按整数格式存入存储器，然后出栈。

3．比较指令

浮点比较指令用于比较栈顶数据与栈顶的源操作数，比较结果通过浮点状态寄存器反映出来，如表 A.1 所示。

表 A.1 比较指令的结果

比较结果	C3	C2	C0
st(0) > 源操作数	0	0	0
st(0) < 源操作数	0	0	1
st(0) = 源操作数	1	0	0
不可排序	1	1	1

- FCOM：浮点数比较，即 st(0) 与 st(1) 比较。
- FCOMP：浮点数比较，即 st(0) 与 st(1) 比较并出栈。
- FICOM m16i/m32i：整数比较，即 st(0) 与 m16i/m32i 比较。

在比较结果中，“不可排序”是指两个浮点格式数不能按照相对值进行比较排序的一种关系。由于浮点指令没有转移指令，所以需要将比较结果 C3、C2、C0 转换到整数状态的寄存器中，然后利用整数转移指令进行跳转，具体过程如下。

①用 FSTSW AX 指令将浮点状态字存入整数寄存器 AX。

②通过 SAHF 指令将条件码传送到整数状态寄存器的低 4 位。SAHF 指令将包含 C3、C2、C0 的高字节送入整数状态字的低字节，正好分别对应于 ZF、PF、CF。

③利用 jxx 指令进行条件判断转移（使用无符号条件转移指令）。

汇编形式的代码如下。

```
FCOM            ;比较指令
FSTSW AX        ;浮点状态寄存器送 AX
SAHF            ;将 AX 高字节转送到整数标志寄存器的低字节
JB …            ;小于，转移
```

A.4 浮点指令

首先对指令做一些说明。

- st(i) 代表浮点寄存器，我们所说的出栈、入栈操作都是对 st(i) 的影响。
- src、dst、dest、op 等表示指令操作数，src 表示源操作数，dst、dest 表示目的操作数。
- mem8、mem16、mem32、mem64、mem80 等表示内存操作数，后面的数值表示该操作数的内存位数（8 位为 1 字节）；
- $x \leftarrow y$ 表示将 y 的值放入 x。例如，st(0) ← st(0)-st(1) 表示将 st(0)-st(1) 的值放入浮点寄存器 st(0)。

1．数据传递和对常量的操作指令

数据传递和对常量的操作指令如表 A.2 所示。

表 A.2　数据传递和对常量的操作指令

指令格式	指令含义	执行的操作
FLD src	装入实数到 st(0)	st(0) ← src (mem32/mem64/mem80)
FILD src	装入整数到 st(0)	st(0) ← src (mem16/mem32/mem64)
FBLD src	装入 BCD 数到 st(0)	st(0) ← src (mem80)
FLDZ	将 0.0 装入 st(0)	st(0) ← 0.0
FLD1	将 1.0 装入 st(0)	st(0) ← 1.0
FLDPI	将 π 装入 st(0)	st(0) ← π
FLDL2T	将 $\log_2 10$ 装入 st(0)	st(0) ← $\log_2 10$
FLDL2E	将 $\log_2 e$ 装入 st(0)	st(0) ← $\log_2 e$
FLDLG2	将 $\log_{10} 2$ 装入 st(0)	st(0) ← $\log_{10} 2$
FLDLN2	将 $\log_e 2$ 装入 st(0)	st(0) ← $\log_e 2$
FST dest	保存实数 st(0) 到 dest 处	dest ← st(0) (mem32/mem64)
FSTP dest		dest ← st(0) (mem32/mem64/mem80)，并出栈
FIST dest	将 st(0) 以整数保存到 dest 处	dest ← st(0) (mem32/mem64)
FISTP dest		dest ← st(0) (mem16/mem32/mem64)，并出栈
FBST dest	将 st(0) 以 BCD 保存到 dest 处	dest ← st(0) (mem80)
FBSTP dest		Dest ← st(0) (mem80)，并出栈

2．比较指令

比较指令如表 A.3 所示。

表 A.3　比较指令

指令格式	指令含义	执行的操作
FCOM	浮点数比较：st(0) 与 st(1)	将标志位设置为 st(0) - st(1) 的结果标志位
FCOM op	浮点数比较	将标志位设置为 st(0) - op (mem32/mem64) 的结果标志位
FICOM op	和整数比较	将 Flags 的值设置为 st(0) - op 的结果 op (mem16/mem32)
FICOMP op	和整数比较	将 st(0) 和 op 与 op (mem16/mem32) 比较并出栈
FTST	零检测	比较 st(0) 和 0.0

续表

指令格式	指令含义	执行的操作
FUCOM st(i)	不可排序比较指令	比较 st(0) 和 st(i)
FUCOMP st(i)	不可排序比较指令	比较 st(0) 和 st(i)，并且执行 1 次出栈操作
FUCOMPP st(i)	不可排序比较指令	比较 st(0) 和 st(i)，并且执行 2 次出栈操作
FXAM	检测栈顶数据 st(0)	条件码 C1=0 为正数，C1=1 为负数

3．运算指令

运算指令如表 A.4 所示。

表 A.4　运算指令

指令格式	指令含义	执行的操作
加法		
FADD	加实数	st(0) ← st(0) + st(1)
FADD src		st(0) ← st(0) + src (mem32/mem64)
FADD st(i),st		st(i) ← st(i) + st(0)
FADDP st(i),st		st(i) ← st(i) + st(0)，并出栈
FIADD src	加一个整数	st(0) ← st(0) + src (mem16/mem32)
减法		
FSUB	减一个实数	st(0) ← st(0) - st(1)
FSUB src		st(0) ← st(0) - src (reg/mem)
FSUB st(i),st		st(i) ← st(i) - st(0)
FSUBP st(i),st		st(i) ← st(i) - st(0)，并出栈
FSUBR st(i),st	用一个实数来减	st(i) ← st(0) - st(i)
FSUBRP st(i),st		st(i) ← st(0) - st(i)，并出栈
FISUB src	减一个整数	st(0) ← st(0) - src (mem16/mem32)
FISUBR src	用一个整数来减	st(0) ← src - st(0) (mem16/mem32)
乘法		
FMUL	乘以一个实数	st(0) ← st(0) × st(1)
FMUL st(i)		st(0) ← st(0) × st(i)
FMUL st(i),st		st(i) ← st(0) × st(i)
FMULP st(i),st		st(i) ← st(0) × st(i)，并出栈
FIMUL src	乘以一个整数	st(0) ← st(0) × src (mem16/mem32)
除法		
FDIV	除以一个实数	st(0) ← st(0) ÷ st(1)
FDIV st(i),st(0)		st(i) ← st(0) ÷ st(i)
FDIVP st(i),st(0)		st(i) ← st(i) ÷ st(0)，并出栈
FIDIV src	除以一个整数	st(0) ← st(0) ÷ src (mem16/mem32)
FDIVR st(i),st	用实数除	st(0) ← st(0) ÷ st(i)
FDIVRP st(i),st		st(i) ← st(0) ÷ st(i)，并出栈
FIDIVR src	用整数除	st(0) ← src ÷ st(0) (mem16/mem32)

续表

指令格式	指令含义	执行的操作
FSQRT	平方根	st(0) ← sqrt st(0)
FSCALE	2 的 st(0) 次方	st(0) ← st(0) × $2^{st(1)}$
FXTRACT	取指数和有效数	将 st(0) 的指数部分存于原数据寄存器，将原 st(0) 的有效数字压入栈顶
FPREM	取余数	st(0) ← st(0) MOD st(1)
FPREM1	取余数（IEEE 标准），st(0) ← st(1) ÷ st(0) 的余数，余数的符号与原来栈顶数据的符号一样	
FRNDINT	取整（四舍五入）	st(0) ← INT(st(0))
FABS	求绝对值	st(0) ← |st(0)|
FCHS	改变符号位（求负数）	st(0) ←st(0)
F2XM1	计算 2^x-1	st(0) ← $2^{st(0)}$ - 1
FYL2X	计算以 2 为基数的对数	st(1) ← st(1) × $\log_2$(st(0))，并出栈
FCOS	余弦函数（cos）	st(0) ← COS(st(0))
FPTAN	正切函数（tan）	st(0) ← TAN(st(0))，并将 1.0 压入栈顶
FPATAN	反正切函数（arctan）	st(1) ← ARCTAG(st(1) / st(0))
FSIN	正弦函数（sin）	st(0) ← SIN(st(0))
FSINCOS	sincos 函数	st(0) ← SIN(st(0))，并且压入 st(1) st(0) ← COS(st(0))
FYL2XP1	计算 st(0) 接近 0 的对数值	st(1) ← st(1) × $\log_2$(st(0)+1.0)，并出栈
处理器控制指令		
FINIT	初始化 FPU	
FSTSW AX	保存状态字的值到 AX	AX ← MSW
FSTSW dest	保存状态字的值到 dest 处	dest ← MSW (mem16)
FLDCW src	从 src 装入 FPU 的控制字	FPU CW ← src (mem16)
FSTCW dest	将 FPU 的控制字保存到 dest 处	dest ← FPU CW
FCLEX	清除异常	
FSTENV dest	保存环境到内存地址 dest 处，保存状态字、控制字、标志字和异常指针的值	
FLDENV src	从主存中取出 14/28 字节的数据，设置 FPU 的环境	
FSAVE dest	检测和处理未屏蔽的错误，将全部状态存入主存，然后初始化 FPU	
FRSTOR src	从 src 处装入由 FSAVE 保存的 FPU 状态	

续表

指令格式	指令含义	执行的操作
FINCSTP	堆栈指针 ST 的值加 1；如果 TOP=7，则增量后为 0	st(6) ← st(5); st(5) ← st(4),..., st(0) ← ?
FDECSTP	堆栈指针 ST 的值减 1；如果 TOP=0，则减量为 7	st(0) ← st(1); st(1) ← st(2),..., st(7) ← ?
FFREE st(i)	标志寄存器 st(i) 未被使用	
FNOP	空操作，等同于 CPU 的 nop 指令	st(0) ← st(0)
WAIT/FWAIT	同步 FPU 与 CPU：停止 CPU 的运行，直到 FPU 完成当前操作码	
FXCH	交换指令，交换 st(0) 和 st(1) 的值	st(0) ← st(1) st(1) ← st(0)

附录 B　在 Visual C++ 中使用内联汇编

使用内联汇编可以在 C/C++ 代码中嵌入汇编语言指令，而且不需要额外的汇编和连接步骤。在 Visual C++ 中，内联汇编是内置的编译器，因此不需要配置诸如 MASM 一类的独立汇编工具。这里以 Visual Studio .NET 2003 为背景，介绍在 Visual C++ 中使用内联汇编的相关知识（如果是早期的版本，可能会有些出入）。

内联汇编代码可以使用 C/C++ 变量和函数，因此它能非常容易地整合到 C/C++ 代码中。它能完成一些对单独使用 C/C++ 来说非常笨重或不可能完成的任务。

内联汇编的用途如下。

- 使用汇编语言编写特定的函数。
- 编写对速度要求较高的代码。
- 在设备驱动程序中直接访问硬件。
- 编写 naked 函数的初始化和结束代码。

B.1　关键字

使用内联汇编时要用到 __asm 关键字，它可以出现在任何允许 C/C++ 语句出现的地方。先来看一些例子。

- 简单的 __asm 块，示例如下。

```
__asm
{
    MOV AL, 2
    MOV DX, 0xD007
    OUT AL, DX
}
```

- 在每条汇编指令之前加 __asm 关键字，示例如下。

```
__asm MOV AL, 2
__asm MOV DX, 0xD007
__asm OUT AL, DX
```

- 因为 __asm 关键字是语句分隔符，所以可以把多条汇编指令放在同一行，示例如下。

```
__asm MOV AL, 2  __asm MOV DX, 0XD007  __asm OUT AL, DX
```

显然，第一种方法与 C/C++ 的风格一致，不仅把汇编代码和 C/C++ 代码清楚地分开了，还避免了重复输入 __asm 关键字。因此，推荐使用第 1 种方法。

不像 C/C++ 中的“{}”，__asm 块的“{}”不会影响 C/C++变量的作用范围。同时，__asm 块可以嵌套，而且嵌套不会影响变量的作用范围。

为了与低版本的 Visual C++ 兼容，_asm 和 __asm 具有相同的意义。另外，Visual C++ 支持

标准 C++ 的 asm 关键字，但不会生成任何指令，其作用仅限于使编译器不会出现编译错误。要使用内联汇编，必须使用 __asm 关键字，而不是 asm 关键字。

B.2 汇编语言

1．指令集

内联汇编支持 Intel Pentium 4 和 AMD Athlon 的所有指令。对其他处理器的指令，可以通过 _EMIT 伪指令来创建（_EMIT 伪指令的说明见下文）。

2．MASM 表达式

在内联汇编代码中，可以使用所有的 MASM 表达式（MASM 表达式是用来计算一个数值或一个地址的操作符和操作数的组合）。

3．数据指示符和操作符

虽然在 __asm 块中允许使用 C/C++ 的数据类型和对象，但它不能使用 MASM 指示符和操作符来定义数据对象。在这里要特别指出，在 __asm 块中不允许使用 MASM 中的定义指示符（DB、DW、DD、DQ、DT 和 DF），也不允许使用 DUP 和 THIS 操作符，MASM 中的结构和记录也不再有效。内联汇编不接受 STRUC、RECORD、WIDTH 或者 MASK。

4．EVEN 和 ALIGN 指示符

尽管内联汇编不支持大多数 MASM 指示符，但支持 EVEN 和 ALIGN。当需要的时候，这些指示符会在汇编代码里加入 nop 指令（空操作），使标号对齐到特定边界，从而使某些处理器在取指令时具有更高的效率。

5．MASM 宏指示符

内联汇编不是宏汇编，不能使用 MASM 宏指示符（MACRO、REPT、IRC、IRP 和 ENDM）和宏操作符（“<>”、“!”、“&”、“%”和“.TYPE”）。

6．段

必须使用寄存器而不是名称来指明段（段名称“_TEXT”是无效的），而且段跨越必须显式地说明，例如 ES:[EBX]。

7．类型和变量大小

在内联汇编中，可以用 LENGTH、SIZE 和 TYPE 来获取 C/C++ 类型和变量的大小。

- LENGTH 操作符用来获取 C/C++ 中数组的元素个数（如果不是一个数组，则结果为 1）。
- SIZE 操作符用来获取 C/C++ 变量的大小（一个变量的大小是 LENGTH 和 TYPE 的乘）。
- TYPE 操作符用来返回 C/C++ 类型和变量的大小（如果变量是一个数组，得到的是数组中单个元素的大小）。

例如，程序中定义了一个 8 维的整数型变量，代码如下。

```
int iArray[8];
```

C 和汇编表达式中得到的 iArray 及其元素的相关值如表 B.1 所示。

表 B.1　C 和汇编表达式中得到的 iArray 及其元素的相关值

__asm	C	Size
LENGTH iArray	sizeof(iArray)/sizeof(iArray[0])	8
SIZE iArray	sizeof(iArray)	32
TYPE iArray	sizeof(iArray[0])	4

8．注释

在内联汇编中可以使用汇编语言的注释，即“;”，示例如下。

```
__asm MOV EAX, OFFSET pbBuff              ;将 pbBuff 的地址载入 EAX
```

因为 C/C++ 宏将展开到一个逻辑行中，所以，为了避免在宏中使用汇编语言注释带来的混乱，内联汇编也允许使用 C/C++ 风格的注释。

9．_EMIT 伪指令

_EMIT 伪指令相当于 MASM 中的 DB，但是 _EMIT 每次只能在当前代码段（.text 段）中定义 1 字节，示例如下。

```
__asm
{
   JMP _CodeLabel

    _EMIT 0x00                    ;定义混合在代码段中的数据
    _EMIT 0x01

_CodeLabel:                       ;这里是代码
    _EMIT 0x90                    ;nop 指令
}
```

10．寄存器的使用

一般来说，不能假定某个寄存器在 __asm 块开始的时候有已知的值。寄存器的值将不能保证从一个 __asm 块保留到另外一个 __asm 块中。

如果一个函数声明为 __fastcall 调用方式，则其参数将通过寄存器而不是堆栈来传递。这将会使 __asm 块产生问题，因为函数无法得知哪个参数在哪个寄存器中。如果函数接收了 EAX 中的参数并立即储存一个值到 EAX 中，原来的参数将被丢弃。另外，在所有声明为 __fastcall 的函数中，ECX 寄存器是必须一直保留的。为了避免以上冲突，包含 __asm 块的函数不要声明为 __fastcall 调用方式。

提示：如果使用 EAX、EBX、ECX、EDX、ESI 和 EDI 寄存器，则不需要保存它。但如果使用 DS、SS、SP、BP 和标志寄存器，就应该用 PUSH 指令保存这些寄存器。如果程序中改变了用于 STD 和 CLD 的方向标志，则必须将其恢复为原来的值。

B.3 使用 C/C++ 元素

1．可用的 C/C++ 元素

C/C++ 与汇编语言可以混合使用，在内联汇编中可以使用 C/C++ 变量及很多其他的 C/C++ 元素，具体如下。

- 符号：包括标号、变量和函数名。
- 常量：包括符号常量和枚举型成员。
- 宏定义和预处理指示符。
- 注释：包括“/**/”和“//”。
- 类型名：包括 MASM 中所有合法的类型。
- typedef 名称：通常使用 PTR 和 TYPE 操作符，或者使用指定的结构或枚举成员。

在内联汇编中，可以使用 C/C++ 或汇编语言的基数计数法。例如，0x100 和 100H 是相等的。

2．操作符的使用

内联汇编中不能使用诸如“<<”一类的 C/C++ 操作符。但是，C/C++ 和 MASM 共有的操作符（例如“*”和“[]”操作符）都被认为是汇编语言的操作符，是可以使用的，示例如下。

```
int iArray[10];
 __asm MOV iArray[6], BX            ;把 BX 的值传送到 iArray+6 地址处
iArray[6] = 0;                      //把 iArray+12 地址处的值赋 0
```

在内联汇编中，可以使用 TYPE 操作符使其与 C/C++ 一致。例如，下面两条语句的作用是一样的。

```
__asm MOV iArray[6 * TYPE int], 0       ;把 iArray+12 地址处的值赋 0
```

```
iArray[6] = 0;                          //把 iArray+12 地址处的值赋 0
```

3．C/C++ 符号的使用

在 __asm 块中可以引用所有在作用范围内的 C/C++ 符号，包括变量名称、函数名称和标号，但不能访问 C++ 类的成员函数。

下面是在内联汇编中使用 C/C++ 符号的一些限制。

- 每条汇编语句只能包含一个 C/C++ 符号。在一条汇编指令中，多个符号只能出现在 LENGTH、TYPE 或 SIZE 表达式中。
- 在 __asm 块中引用函数时必须先声明，否则，编译器将不能区别 __asm 块中的函数名和标号。
- 在 __asm 块中不能使用对 MASM 来说是保留字的 C/C++ 符号（不区分大小写）。MASM 保留字包括指令名称（例如 PUSH）和寄存器名称（例如 ESI）等。
- 在 __asm 块中不能识别结构和联合标签。

4．访问 C/C++ 中的数据

内联汇编的一个非常大的方便之处是可以使用名称来引用 C/C++ 变量。如果 C/C++ 变量 iVar 在作用范围内，示例如下。

```
__asm MOV EAX, iVar                    ;将 iVar 的值传送到 EAX 处
```

如果 C/C++ 中的类、结构或者枚举成员具有唯一的名称，则在 __asm 块中可以只通过成员名称来访问（省略“.”操作符之前的变量名或 typedef 名称）。然而，如果成员不是唯一的，就必须在“.”操作符之前加上变量名或 typedef 名称。例如，下面两个结构都具有 SameName 这个成员变量。

```
struct FIRST_TYPE
  {
    char *pszWeasel;
    int SameName;
  };
```

```
struct SECOND_TYPE
{
    int iWonton;
    long SameName;
 };
```

如果按下面的方式声明变量：

```
struct FIRST_TYPE ftTest;
struct SECOND_TYPE stTemp;
```

那么，所有引用 SameName 成员的地方都必须使用变量名，因为 SameName 不是唯一的。另外，由于上面的 pszWeasel 变量具有唯一的名称，所以，可以仅使用它的成员名称来引用它，示例如下。

```
__asm
{
    MOV EBX, OFFSET ftTest
    MOV ECX, [EBX]ftTest.SameName       ;必须使用“ftTest”
    MOV ESI, [EBX]. pszWeasel           ;可以省略“ftTest”
}
```

提示：省略变量名仅仅是为了方便书写代码，生成的汇编指令还是一样的。

5．用内联汇编编写函数

如果用内联汇编编写函数，要传递参数和返回一个值都是非常容易的。通过下面的例子比较一下用独立汇编和内联汇编编写的函数。

```
;PowerAsm.asm
;Compute the power of an integer
 PUBLIC     GetPowerAsm
 _TEXT      SEGMENT WORD PUBLIC 'CODE'
```

```
GetPowerAsm PROC
    PUSH    EBP                         ;保存 EBP
    MOV     EBP, ESP                    ;把 ESP 的值传给 EBP，以方便我们引用堆栈参数
    MOV     EAX, [EBP+4]                ;取第 1 个参数
    MOV     ECX, [EBP+6]                ;取第 2 个参数
    SHL     EAX, CL                     ;EAX = EAX * (2 ^ CL)
    POP     EBP                         ;恢复 EBP
    RET                                 ;返回
  GetPowerAsm  ENDP
  _TEXT      ENDS
  END
```

C/C++ 函数一般用堆栈来传递参数，所以在上面的函数中需要通过堆栈位置来访问它的参数（在 MASM 或其他一些汇编工具中，也允许通过名称来访问堆栈参数和局部堆栈变量）。

下面的程序是使用内联汇编编写的。

```
//PowerC.c
#include <Stdio.h>
int GetPowerC(int iNum, int iPower);
int main()
{
   printf("3 times 2 to the power of 5 is %d\n", GetPowerC( 3, 5));
}
   int GetPowerC(int iNum, int iPower)
{
    __asm
   {
         MOV EAX, iNum              ;取第 1 个参数
         MOV ECX, iPower            ;取第 2 个参数
         SHL EAX, CL                ;EAX = EAX * (2 to the power of CL)
   }
   //返回值在 EAX 中
}
```

使用内联汇编编写的函数 GetPowerC 可以通过参数名称来引用它的参数。由于 GetPowerC 函数没有执行 C 的 return 语句，所以编译器会给出一个警告信息，可以通过“#pragma warning”语句禁止生成这个警告信息。

内联汇编的用途之一是编写 naked 函数的初始化和结束代码。对一般的函数，编译器会自动生成函数的初始化（构建参数指针和分配局部变量等）和结束代码（平衡堆栈和返回一个值等）。在使用内联汇编时，我们可以自己编写干净的函数。当然，此时必须动手做一些有关函数初始化和扫尾的工作，示例如下。

```
void __declspec(naked) MyNakedFunction()
{
  //为 naked 函数提供初始化代码
  __asm
      {
        PUSH EBP
        MOV ESP, EBP
```

```
        SUB ESP, __LOCAL_SIZE
      }
        ……
        //函数结束代码
  __asm
      {
        POP EBP
        RET
      }
}
```

6．调用 C/C++ 函数

在内联汇编中调用声明为 __cdecl 方式（默认）的 C/C++ 函数时，必须由调用者清除参数堆栈。下面是一个调用 C/C++ 函数的例子。

```
#include <Stdio.h>
char szFormat[] = "%s %s\n";
char szHello[] = "Hello";
char szWorld[] = " world";

void main()
{
   __asm
       {
         MOV    EAX, OFFSET szWorld
         PUSH   EAX
         MOV    EAX, OFFSET szHello
         PUSH   EAX
         MOV    EAX, OFFSET szFormat
         PUSH   EAX
         CALL   printf
         //在堆栈中压入了 3 个参数，调用函数之后要调整堆栈
         ADD    ESP, 12
      }
}
```

提示：参数是按从右往左的顺序压入堆栈的。

如果要调用 __stdcall 方式的函数，则不需要自己清除堆栈，因为这种函数的返回指令是 RETn，会自动清除堆栈。大多数 Windows API 函数均为 __stdcall 调用方式（仅除 wsprintf 等几个）。一个调用 MessageBox 函数的例子如下。

```
#include <Windows.h>
TCHAR g_tszAppName[] = TEXT("API Test");

void main()
{
  TCHAR tszHello[] = TEXT("Hello, world!");
   __asm
       {
```

```
            PUSH    MB_OK OR MB_ICONINFORMATION
            PUSH    OFFSET g_tszAppName            ;全局变量用 OFFSET
            LEA     EAX, tszHello                  ;局部变量用 LEA
            PUSH    EAX
            PUSH    0
            ;注意：这里不是 CALL MessageBox，而是调用重定位过的函数地址
            CALL    DWORD PTR [MessageBox]
        }
}
```

提示：参数可以不受限制地访问 C++ 成员变量，但不能访问 C++ 的成员函数。

7．定义 __asm 块为 C/C++ 宏

使用 C/C++ 宏可以方便地把汇编代码插入源代码中，但在这个过程中需要特别注意，因为宏将扩展到一个逻辑行中。为了不出现问题，请按如下规则编写宏。

- 使用花括号把 __asm 块括住。
- 把 __asm 关键字放在每条汇编指令之前。
- 使用经典的 C 风格的注释“/* comment */”，不要使用汇编风格的注释“; comment”或单行的 C/C++ 注释“// comment”。

举个例子，下面定义了一个简单的宏。

```
#define PORTIO __asm      \
/* Port output */         \
{                         \
    __asm MOV AL, 2       \
    __asm MOV DX, 0xD007  \
    __asm OUT DX, AL      \
}
```

乍一看，后面的 3 个 __asm 关键字好像是多余的，但实际上它们是有用的，因为宏将被扩展到如下单行代码中。

```
__asm /* Port output */ {__asm MOV AL, 2  __asm MOV DX, 0xD007  __asm OUT DX, AL}
```

从扩展后的代码中可以看出，第 3 个和第 4 个 __asm 关键字是必需的（作为语句分隔符）。在 __asm 块中，只有 __asm 关键字和换行符会被认为是语句分隔符，又因为定义为宏的一个语句块会被认为是一个逻辑行，所以必须在每条指令之前使用 __asm 关键字。

括号也是需要的，如果省略了它，编译器将不知道汇编代码在哪里结束，__asm 块后面的 C/C++ 语句会被认为是汇编指令。

同样是由于宏展开，汇编风格的注释“; comment”和单行的 C/C++ 注释“// commen”也可能出现错误。为了避免这些错误，在定义 __asm 块为宏时，请使用经典 C 风格的注释“/* comment */”。

和 C/C++ 宏一样，__asm 块写的宏也可以拥有参数。和 C/C++ 宏不一样的是，__asm 宏不能返回一个值，因此不能使用这种宏作为 C/C++ 表达式。

不要不加选择地调用这种类型的宏。例如，在声明为 __fastcall 的函数中调用汇编语言宏可能会导致不可预料的结果（请参考前文的说明）。

8．跳转

可以在 C/C++ 里使用 goto 指令转跳到 __asm 块中的标号处，也可以从 __asm 块中转跳到 __asm 块中或外面的标号处。__asm 块中的标号是不区分大小写的（指令、指示符等也是不区分大小写的），示例如下。

```
void MyFunction()
{
   goto C_Dest;        /* 正确 */
   goto c_dest;        /* 错误 */
   goto A_Dest;        /* 正确 */
   goto a_dest;        /* 正确 */
   __asm
   {
      JMP C_Dest       ;正确
      JMP c_dest       ;错误
      JMP A_Dest       ;正确
      JMP a_dest       ;正确
    a_dest:            ;__asm 标号
    }
   C_Dest:             /* C/C++ 标号 */
     return;
}
```

不要将函数名称作为标号，否则将转跳到函数中执行，而不是标号处。例如，由于 exit 是 C/C++ 的函数，如下转跳将不会来到 exit 标号处。

```
;错误：将函数名作为标号
     JNE exit
     ……
exit:
     ……
```

美元符号“$”用于指定当前指令的位置，常用于条件跳转中，示例如下。

```
     JNE $+5             ;下面这条指令的长度是 5 字节
     JMP _Label
     NOP                 ;$+5，转跳到了这里
     ……
  _Label:
     ……
```

B.4　在 Visual C++ 工程中使用独立汇编

内联汇编代码不易移植，如果程序打算在不同类型的机器（例如 x86 和 Alpha）上运行，可能需要在不同的模块中使用特定的机器代码。这时可以使用 MASM（Microsoft Macro Assembler），因为 MASM 支持更多方便的宏指令和数据指示符。

这里简单介绍一下在 Visual Studio .NET 2003 中调用 MASM 编译独立汇编文件的步骤。

在 Visual C++ 工程中，添加按 MASM 的要求编写的 .asm 文件。在解决方案资源管理器中，在这个文件上单击右键，在弹出的快捷菜单中选择“属性”菜单项，在“属性”对话框中单击

“自定义生成步骤”选项，设置如下项目。

- 命令行：ML.exe /nologo /c /coff "-Fo$(IntDir)\$(InputName).obj" "$(InputPath)"
- 输出：$(IntDir)\$(InputName).obj

如果要生成调试信息，可以在命令行中加入“/Zi”参数，还可以根据需要生成 .lst 和 .sbr 文件。

如果要在汇编文件中调用 Windows API，可以从网上下载 MASM32 包（包含 MASM 汇编工具、非常完整的 Windows API 头文件/库文件、实用宏及大量的 Win32 汇编例子等）。相应地，应该在命令行中加入“/I X:\MASM32\INCLUDE”参数来指定 Windows API 汇编头文件（.inc）的路径。MASM32 的主页是 http://www.masm32.com，在那里可以下载最新版本的 MASM32 包。

附录 C　Visual Basic 程序

VB3 和 VB4 是典型的解释语言，它们都有相应的反编译器存在。VB5 和 VB6 不再是单纯的解释程序，在保留 P-code 编译的同时引入了 Native 编译方式，使得生成本地二进制代码成为可能。关于 VB.NET，其生成的 EXE 程序完全可用 .NET 反编译工具完成，此处不再介绍。

本章仅简单介绍 Visual Basic 程序的一般调试方法，有关 Visual Basic 6 逆向工程的更多知识，请参考《软件加密技术内幕》一书由周文雄撰写的第 8 章“Visual Basic 6 逆向工程”。

C.1　基础知识

下面我们了解一下 Visual Basic 的相关基础知识。

C.1.1　字符编码方式

Visual Basic 32 位版本的字符串处理采用 Unicode 编码（或称宽字符，Widechars）。也就是说，字符串在 Visual Basic 内部是以 Unicode 格式存放的。在 Unicode 中，所有的字符都是 16 位的，例如 7 位 ASCII 码都被扩充为 16 位（注意：高位扩充的是 0）。

因为版本不同，Visual Basic 中的字符串格式也不同。16 位 VB 4.0 和以前版本的 VB 字符串格式很复杂，使用连续指针的方式，源指针指向一个由字符串长度和字符串首地址指针组合而成的结构，其中的字符串首地址指针再指向字符串。VB 5.0 及以后版本则是单指针方式，源指针指向一个复合字符串，从此字符串开始的 4 字节组成一个长整数，以指示此字符串的长度，其后是字符串，这个字符串除了应有的字符外，结尾还有两个“00”，而这两个“00”不计算在字符串长度之内。一个英文的 VB 字符串“pediy”在内存中（编译后）应该如图 C.1 所示。

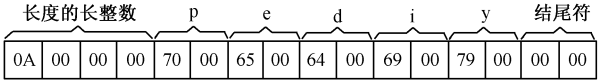


图 C.1　VB 字符串在内存中的形式

C.1.2　编译模式

编译器的编译技术可以分为 Native-Compile（自然编译）与 Pcode-Compile（伪编译）两种。自然编译是编译器将高级语言转换为汇编代码，并经链接生成 EXE 程序的过程。伪编译是编译器将高级语言转换为某种编码后，将能解释、执行此编码的一段程序一同链接，生成 EXE 程序。

所谓伪代码，其基本工作原理是编译器先把执行程序编译成比 80x86 机器码紧凑得多的中间代码形式，运行时，其基于堆栈的引擎将特殊操作码翻译成 CPU 上的操作指令，这个引擎其实就是一种代码虚拟机。依靠 P-code 编译技术，使编程语言不依赖机器或系统平台成为可能。

Visual Basic 1.0 到 4.0 都只生成 P-code。P-code 的最大问题就是执行速度比本地编译的程序慢，优点是使程序不依赖硬件或操作系统成为可能。从 1997 年 Visual Basic 5 问世起，微软

在编译过程中添加了允许生成本地机器码的选项。虽然目前 P-code 编译形式大多见于 Visual Basic，但 Java、PowerBuilder 等的编译实际上也是一样的。

因为 VB3 和 VB4 都使用伪代码编译形式，而伪代码编译的代码实际上只是“变形的源代码”，所以，只要能理解其对应机制，就能开发出反编译器，例如 Dodi's VB 3/4 Disassembler。Java、PowerBuilder 也有反编译器。

C.2 自然编译

自然编译（Native）是指 VB 源代码生成汇编代码并链接的过程。因为是生成汇编代码，所以对它的解读完全遵照一般的反汇编常规。

C.2.1 相关 VB 函数

掌握一些常用 VB 内部函数，将有助于对 VB 程序的分析。以 Visual Basic 6.0 为例，实际上的数据计算和比较是在 msvbvm60.dll 及 oleaut32.dll 中完成的。如果要深入研究 VB，不妨用 IDA 反汇编 msvbvm60.dll 及 oleaut32.dll，理解其函数的意义。msvbvm60.dll 中包含的函数名总是以“__vba”或“rtc”开头，而 oleaut32.dll 函数名以“var”开头。对于函数的作用，一般可以按照函数名从右往左理解，例如“__vbaI2Str”表示字符串转换为整数。

通过研究 OLEAUT.H、OAIDL.h 文件，可以了解 VB 函数中的各类变量，进而根据函数名称猜出大部分函数的用途，如表 C.1 所示。

表 C.1 VB 相关函数

函 数 名	含 义
MultiByteToWideChar	将 ANSI 字符串转换成 Unicode 字符（Win32 API 函数）
WideCharToMultiByte	将 Wide-Character（Unicode）字符转换成 ANSI 字符（Win32 API 函数）
rtcR8ValFromBstr	把字符串转换成浮点数
__vbaStrCmp	比较字符串
__vbaStrComp	比较字符串
__vbaStrCopy	复制字符串
__vbaStrMove	移动字符串
__vbaVarTstNe	进行变量比较
__vbaVarTstEq	进行变量比较
StrConv	转换字符
rtcMsgBox	显示对话框
rtcGetPresentDate	取得当前日期
VarBstrCmp（Oleaut32.dll）	比较字符串，例如“bp Oleaut32.dll! VarBstrCmp”
VarCyCmp（Oleaut32.dll）	比较字符

注意：这些函数前的下画线“__”是由两根短线“_”组成的。

C.2.2　VB 程序比较模式

本节从汇编的角度来分析一下 VB 字符串的比较模式，帮助读者熟悉 VB 的一些处理方式。

1．字符串（String）比较

在这种方法里，正确的密码串（例如“Correct Password”）和输入的密码串（例如“Entered Password”）做比较。字符串是由相邻的字符按顺序排列组成的。一个字符串包括字母、数字、空格和标点符号。下面是一段范例代码。

```
If "Correct Password" = "Entered Password" then   ; 直接比较两个字符串
    GoTo Correct Message
Else
    GoTo Wrong Message
End if
```

这种方式是明码比较。如果程序用这种函数比较，很容易被破解。可能用到的断点是 __vbaStrComp 或 __vbaStrCmp。

运行用 VB6 编译好的实例程序 string.exe，可以用 __vbaStrCmp 或 rtcMsgBox 设置断点。汇编形式如下。

```
0040227B   push    eax                          ;输入的假序列号
0040227C   push    00401BD8                     ;查看 00401BD8 地址即可看到真序列号
00402281   call    [<&MSVBVM60.__vbaStrCmp>]    ;比较函数
{
    660E8A03   push    dword ptr [esp+8]
    660E8A07   push    dword ptr [esp+8]
    660E8A0B   push    0                        ;0 为二进制比较（默认）；1 为文本方式比较
    660E8A0D   call    __vbaStrComp
    660E8A12   retn    8
}
00402287   mov     esi, eax                     ;相等则返回 0 到 eax
```

OllyDbg 断下后，在数据窗口查看 00401BD8 处的数据，如图 C.2 所示。数据以宽字符显示，转换过来就是“pediy”，这就是真的序列号。

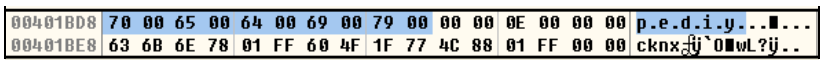


图 C.2　在数据窗口查看字符

2．变量（Variant）比较

在这种方法中，两个变量（变量数据类型）互相比较。变量数据类型是一种特殊的数据类型，包括数字、字符串、日期数据及一些用户定义的类型。这种类型存储的数字长度是 16 字节，而字符长度是 22 字节（加上字符串长度）。下面是一段范例代码。

```
Dim correct As Variant, entered As Variant      ;定义 correct 和 entered 作为变量
correct = "pediy"                               ;设置 correct 放置“pediy”
entered = Text1.Text                            ;设置 entered 为输入的密码
If correct = entered Then                       ;用变量方法比较
```

```
    GoTo Correct Message
Else
    GoTo Wrong Message
```

在此方法中，字符串比较中的两个 API 断点将不起作用，因为程序不再使用 __vbaStrCmp 等函数比较字符串。其比较的方式依赖于变量是否相等，即用 __vbaVarTstEq 函数比较变量。

用 VB 6 编译的程序为 Variant.exe。比较代码汇编形式如下。

```
004024BF   push    ecx                              ;在 OllyDbg 命令行里执行 D [ecx+8]
004024C0   push    edx                              ;在 OllyDbg 命令行里执行 D [edx+8]
004024C1   call    [<&MSVBVM60.__vbaVarTstEq>]      ;变量比较函数
004024C7   test    ax, ax                           ;相等则返回-1
```

在 OllyDbg 的命令行里，执行“D [edx+8]”命令，将显示正确的序列号，其以宽字符显示，如图 C.3 所示。

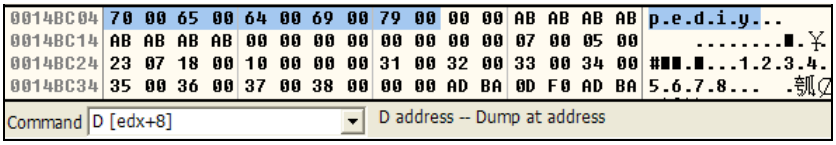


图 C.3　在数据窗口查看参数

3．字节（Byte）比较

这种方法是用两个字节数据进行比较。Byte 变量存储为单精度型、无符号整型、8 位（1 字节）的数值形式，范围在 0 至 255 之间。下面是一段范例代码。

```
Dim correct As Byte, entered As Byte          ;定义 correct 和 entered 为字节型
correct =12                                   ;设置 correct 为“12”
entered = Text1.Text                          ;设置 entered 为输入的密码
If correct = entered Then                     ;比较
     GoTo Correct Message
Else
     GoTo Wrong Message
End If
```

对这种类型没有专门的断点函数，因为其数据比较是在主程序里，而不是在 Visual Basic 运行库中。

用 VB 6 编译的程序为 Byte.exe。代码用十六进制数据比较，汇编形式如下。

```
004023D0   cmp     bl, al                        ;al=0xc，转换成十进制是 12
……
004023E3   jnz     short 00402444
```

4．整型（Integer）比较

这种方法是用两个整型数据进行比较。Integer 变量存储为 16 位（2 字节）的数值形式，其范围为 –32768 到 32767。下面是一段范例代码。

```
Dim correct As Integer, entered As Integer    ;定义 correct 和 entered 为整型
```

```
correct = 1212                                  ;设置 correct 为“1212 ”
entered = Text1.Text                            ;设置 entered 为输入的密码
If correct = entered Then                       ;比较
     GoTo Correct Message
Else
     GoTo Wrong Message
End If
```

对这种类型没有专门的断点函数，因为其数据比较是在主程序里，而不是在 Visual Basic 运行库中。

用 VB 6 编译的程序为 Integer.exe。代码用十六进制数据比较，汇编形式如下。

```
004023B4   cmp     si, 4BC                      ;1212 的十六进制就是 04BCh
……
004023C5   jnz     short 00402426
```

5．长整型（Long）比较

这也是一种常见的方法，两个变量（长整型）互相比较。Long（长整型）变量存储为 32 位（4 字节）有符号的数值形式，其范围从-2147483648 到 2147483647。因此，该方法只能用来比较数字。下面是一段范例代码。

```
Dim correct As Long, entered As Long         ;定义 correct 和 entered 作为长整型
correct = 1872333                            ;设置“1872333”为正确密码
entered = Text1.Text                         ;设置 entered 为输入的密码
If entered = correct Then                    ;用长整型方法比较
      GoTo Correct Message
Else
      GoTo Wrong Message
End If
```

用 VB 6 编译的程序为 Long.exe。代码用十六进制数据比较，汇编形式如下。

```
0040250D   cmp     esi, 1C91CD               ;1872333 的十六进制就是 1C91CDh
……
0040251F   jnz     short 00402580            ;相等则不跳转
```

6．单精度实数（Single）比较

这种方法用两个单精度实数进行比较。Single（单精度浮点型）变量存储为 32 位 IEEE（4 字节）浮点数值的形式，它的范围在负数的时候是从 -3.402823E38 到 -1.401298E–45，在正数的时候是从 1.401298E–45 到 3.402823E38。因此，这种方法只能比较数字，但可以将姓名和序列号转换成实数进行比较。查看相关数据时用 DL 命令显示浮点型（Long/Real）。下面是一段范例代码。

```
Dim correct As Single, entered As Single     ;定义 correct 和 entered 为单精度型
correct = 1872333                            ;设置 correct 为“1872333”
entered = Text1.Text                         ;设置 entered 为输入的密码
If correct = entered Then                    ;比较
     GoTo Correct Message
```

```
Else
     GoTo Wrong Message
End If
```

用 VB 6 编译的程序为 Single.exe。代码用单精度实数数据格式比较，汇编形式如下。

```
004023B3   cmp     eax, 49E48E68              ;1872333 的单精度浮点型是 49E48E68h
……
004023C9   jnz     short 0040242A             ;相等则不跳转
```

7．双精度（Double）比较

这种方法用两个双精度数据比较。Double（双精度浮点型）变量存储为 64 位 IEEE（8 字节）浮点数值的形式，它为负数时是从 -1.79769313486231E308 到 -4.94065645841247E–324，为正数时是从 4.94065645841247E–324 到 1.79769313486232E308。双精度与单精度类似，只能比较数字。下面是一段范例代码。

```
Dim correct As Double, entered As Double     ;定义 correct 和 entered 为双精度型
correct = 12345678901234                     ;设置 correct 为 “12345678901234”
entered = Text1.Text                         ;设置 entered 为输入的密码
If correct = entered Then                    ;比较
     GoTo Correct Message
Else
     GoTo Wrong Message
End If
```

用 VB 6 编译的程序为 Double.exe。由于比较的数是 64 位的，因此分两次比较。第 1 次是比较双精度的前 32 位“9C5FE400”，第 2 次是比较双精度的后 32 位“42A674E7”。本例的汇编代码形式如下。

```
00402952  push    eax                            ;指向输入的字符串
00402953  call    [<&MSVBVM60.__vbaR8Str>]       ;转换 string 为 Integer/Real 到 st(0)
00402959  fstp    qword ptr [ebp-24]             ;[ebp-24]=st(0)，处理输入的序列号
0040295C  lea     ecx, dword ptr [ebp-28]
0040295F  call    [<&MSVBVM60.__vbaFreeStr>]
00402965  lea     ecx, dword ptr [ebp-2C]
00402968  call    [<&MSVBVM60.__vbaFreeObj>]
0040296E  cmp     dword ptr [ebp-24], 9C5FE400   ;比较双精度的前 32 位
00402975  jnz     004029FC
0040297B  cmp     dword ptr [ebp-20], 42A674E7   ;比较双精度的后 32 位
00402982  jnz     short 004029FC
```

8．货币（Currency）比较

这种方法是用两个 Currency 数据类型进行比较。Currency 变量存储为 64 位（8 字节）整型数值的形式，然后除以 10000，给出一个定点数，其小数点左边有 15 位数字，右边有 4 位数字。这种表示法的范围可以从 –922 337 203 685 477.5808 到 922 337 203 685 477.5807。Currency 数据类型在货币计算与定点计算中很有用，因为在这些场合精度特别重要。下面是一段范例代码。

```
Dim correct As Currency, entered As Currency     ;定义 correct 和 entered 为 Currency
```

```
correct = 1234567890                              ;设置 correct 为“1234567890”
entered = Text1.Text                              ;设置 entered 为输入的密码
If correct = entered Then                         ;比较
     GoTo Correct Message
Else
     GoTo Wrong Message
End If
```

用 VB 6 编译的程序为 Currency.exe。代码将 Currency 数据分成两段来比较，先比较前 32 位，再比较后 32 位。汇编形式如下。

```
004023DE   cmp    esi, dword ptr [ebp-1C]         ;数据的前 32 位 73CE2B20h
004023E1   jnz    short 00402460
004023E3   cmp    edi, dword ptr [ebp-18]         ;数据的后 32 位 00000B3Ah
004023E6   jnz    short 00402460
```

9．小结

当两个表达式都是数值数据类型（Byte、Integer、Long、Single、Double、Currency）时，进行数值比较。当一个 Single 与一个 Double 做比较时，Double 会进行舍入处理且与 Single 有相同的精确度。当一个 Currency 与一个 Single 或 Double 进行比较时，Single 或 Double 会转换成一个 Currency。在实际应用中，程序一般同时采用两种以上的方法来比较数据，例如 Currency 和 String、Variant 和 Long 等。

在分析 Visual Basic 程序时，对 Vbrun*.dll（VB 3 和 VB 4 版本）和 Msvbvm*.dll（VB 5 和 VB 6）强调得比较多。实际上，Visual Basic 程序的很多运算是在 Oleaut32.dll 中完成的，这个 DLL 提供了许多对 Visual Basic 中 Variant 类型的变量进行操作的函数，主要是一系列 VarXXX 函数。

除了使用 OllyDbg 等工具分析 VB 程序，也可以使用 SmartCheck 来辅助分析。SmartCheck 是 NuMega 公司推出的一款针对 Visual Basic 的错误检测和调试工具，有关内容请参考随书文件。它能够自动检测和诊断 VB 运行时的错误，并将一些表达不清楚的错误信息转换为确切的错误描述。

C.3　伪编译

伪编译是生成伪代码并链接的过程，在运行时依赖解释引擎将伪代码翻译为汇编代码再执行。伪代码的运行完全依赖于栈。

如果用 IDA 反汇编 vbpcode.exe 程序，会看到一些莫名其妙的代码，因为它不再是传统意义上的汇编代码，只有用 P-code 反编译器才能得到正确的代码。VB 5 和 VB 6 的 P-code 反编译器有 Exdec、WKTVBDebugger、VBDE 等版本。

C.3.1　虚拟机与伪代码

相对于 VB Native Code，P-code 编译模式要出现得更早一些。事实上，在 VB 5.0 以前的版本中，所有的 VB 程序代码都不加选择地被编译为 P-code 形式。VB P-code 的运行速度较慢，

是由它本身的运行机制决定的。

1．虚拟执行的原理

P-code，即“Pseudo Code”（伪代码），这一概念最早出现在 Pascal 编译器中，是为了提供跨平台可移植性而产生的，实现这一编译机制的 Pascal 编译器称为“Pascal P Compiler”。Sun 公司在其推出的 Java 语言中也成功地实现了这种机制。Java 程序的伪编译代码由一系列代表一定意义的字节码（byte code）组成，它们同属于一套特定的指令集。这种字节码不能由不同的 CPU 直接执行，而是要通过特殊的解释引擎翻译为 CPU 可以识别的指令才能执行，这种解释引擎就是我们常说的虚拟机。只要在不同的平台上提供虚拟机，把字节码翻译为对应的 CPU 指令集，就实现了所谓的跨平台特性。

微软推出的 VB P-code 实际上也是一组自定义的指令集，必须通过基于堆栈的虚拟机翻译为 80x86 上的指令集才能执行，担任虚拟机任务的就是系统目录下的 msvbvm50.dll 和 msvbvm60.dll 这两个动态链接库文件。由于在文件执行过程中多出了一个解释的步骤，所以自然会影响其执行速度。至少到目前为止，VB P-code 没有实现所谓跨平台运行特性，而这对于“Pseudo”这个词的起源是不恰当的。另外，采用 P-code 形式编译生成的 VB 应用程序的体积一般要小于采用 Native Code 形式编译的同样程序（这是由于 P-code 指令集的每一条指令往往对应于一组 80×86 指令所完成的任务）。

2．虚拟机实例分析

为了理解虚拟机的运作方式，这里提供一个用 P-code 编译的小程序 VBP-code-Ex.exe，代码如下。

```
Private Sub Command1_Click()
X = InputBox("Please input an integer", "Input")
If X <> "" Then
   Y = X + 2
   X = Y * 3
   Out.Text = X
End If
End Sub
```

这个程序只处理 Command_Click 事件，同时用到了 InputBox，以便用 rtcInputBox 函数设断。

用 OllyDbg 加载这个程序，在命令行插件中输入“bp rtcInputBox”，按“F9”键运行，然后单击“OK”按钮，中断如下。

```
6A360CF2  PUSH EBP                                ;rtcInputBox 函数的第 1 句指令
6A360CF3  MOV EBP,ESP
6A360CF5  SUB ESP,54
6A360CF8  MOV EAX,DWORD PTR SS:[EBP+1C]
```

按“Ctrl + F9”组合键返回，当 InputBox 弹出后输入任意整数，继续进行单步跟踪，直到下面的代码处。

```
6A37D2CD  MOVSX EAX,WORD PTR DS:[ESI]
6A37D2D0  PUSH DWORD PTR DS:[EAX+EBP]
```

```
6A37D2D3  XOR EAX,EAX
6A37D2D5  MOV AL,BYTE PTR DS:[ESI+2]
6A37D2D8  ADD ESI,3
6A37D2DB  JMP DWORD PTR DS:[EAX*4+6A37DA58]      ;注意这一句
6A37D2E2  MOVSX EAX,WORD PTR DS:[ESI]            ;停在这里
6A37D2E5  ADD EAX,EBP
6A37D2E7  PUSH EAX
6A37D2E8  XOR EAX,EAX
6A37D2EA  MOV AL,BYTE PTR DS:[ESI+2]
6A37D2ED  ADD ESI,3
6A37D2F0  JMP DWORD PTR DS:[EAX*4+6A37DA58]      ;注意这一句
```

注意：由于 msvbvm60.dll 的版本不同，读者计算机上显示的代码和地址会与本书不同。

如果不明白这些指令的意图，就会迷失在行为相似的代码之中。事实上，这几条指令正是虚拟机读取 P-code 伪代码的引擎部分。在这里，尤其值得关注的是它读取了什么，而不是执行了什么。为了说明这些指令的功能，首先了解一下 VB P-code 伪代码的格式，示例如下。

```
3A 6C FF 03 00
操作码      操作数
```

这是一句典型的 VB P-code 指令，其助记符为 LitVarStr，表示将一个 Variant 型的字符串入栈。“3A”是指令的操作码；“0003”是操作数，是以某种形式标记的字符串地址。现在我们有足够的信息来解释虚拟机所做的一切了，代码如下。

```
6A37D2E2  MOVSX EAX,WORD PTR DS:[ESI]            ;ESI 指向待解释指令的操作数
6A37D2E5  ADD EAX,EBP                            ;取得某种形式的字符串指针
6A37D2E7  PUSH EAX                               ;压栈（这是伪代码的核心操作）
6A37D2E8  XOR EAX,EAX                            ;将 EAX 清零
6A37D2EA  MOV AL,BYTE PTR DS:[ESI+2]             ;取下一条指令的操作码
6A37D2ED  ADD ESI,3                              ;移至下一条指令的操作数
6A37D2F0  JMP DWORD PTR DS:[EAX*4+6A37DA58]      ;根据跳转地址表到达下一条指令的解释单元
```

虚拟机并没有用简单的条件判断来识别操作码，而是采用了跳转地址表法，把执行流直接引导到相应的解释单元。6A37DA58h 是地址跳转表的首地址，EAX 保存了下一条指令的操作码，由于每一个跳转地址是一个 DWORD，所以用 EAX 乘以 4 的值加上跳转表的基地址来索引下一条指令的解释单元。当然，由于每一条指令的长度是不同的，所以前面提到的读取 P-code 伪代码的引擎部分并不完全相同。明白了虚拟机的解释原理以后，跟踪 P-code 程序就方便多了，一旦运行到读取 P-code 伪代码的引擎部分，就意味着虚拟机开始解释下一条指令了。尽管如此，调试 P-code 程序还是比调试本地机器码编译的普通可执行程序困难得多，因为在虚拟机的工作流程中无法随时回顾前面执行过的指令。

下面来看看加法在虚拟机中是如何被解释执行的。按“F8”键跟踪代码，来到如下代码处。

```
6A37D3B3  ADD EDI,EBP
6A37D3B5  PUSH EDI
6A37D3B6  XOR EAX,EAX
6A37D3B8  MOV AL,BYTE PTR DS:[ESI]
6A37D3BA  INC ESI
```

```
6A37D3BB  JMP DWORD PTR DS:[EAX*4+6A37DA58] ;al = FBh
```

AL 的值为“FB”，这是 Variant 型变量算术运算伪指令的操作码。ESI 指向该操作码后的一个次操作码 94，代表加法运算。

跟随跳转地址表来到如下代码处。

```
6A37D9BA  XOR EAX,EAX
6A37D9BC  MOV AL,BYTE PTR DS:[ESI]
6A37D9BE  INC ESI
6A37D9BF  JMP DWORD PTR DS:[EAX*4+6A37DE58]      ;al = 94h
```

这是一个二级跳转表，要注意，6A37DE58 这个值和前面的跳转地址表基址不同了，这说明 VB P-code 算术运算伪指令采用了二级跳转来解释执行。由于 1 字节的操作码可以表示的操作指令种类最多为 256 个，所以为了解决指令数不够的问题，微软对部分伪指令进行了二级跳转解释执行。

```
6A384628  LEA EBX,DWORD PTR DS:[__vbaVarSub]
6A38462E  JMP SHORT MSVBVM60.6A384610
6A384630  LEA EBX,DWORD PTR DS:[__vbaVarMul]
6A384636  JMP SHORT MSVBVM60.6A384610
6A384638  LEA EBX,DWORD PTR DS:[__vbaVarDiv]
6A38463E  JMP SHORT MSVBVM60.6A384610
6A384640  LEA EBX,DWORD PTR DS:[__vbaVarIdiv]
6A384646  JMP SHORT MSVBVM60.6A384610
6A384648  LEA EBX,DWORD PTR DS:[__vbaVarMod]
6A38464E  JMP SHORT MSVBVM60.6A384610
6A384650  LEA EBX,DWORD PTR DS:[__vbaVarAdd]     ;跳转到这里执行，获得加法函数地址
6A384656  JMP SHORT MSVBVM60.6A384610
6A384658  LEA EBX,DWORD PTR DS:[__vbaVarAnd]
6A38465E  JMP SHORT MSVBVM60.6A384610
6A384660  LEA EBX,DWORD PTR DS:[__vbaVarOr]
6A384666  JMP SHORT MSVBVM60.6A384610
```

很明显，Variant 变量的算术逻辑运算伪操作都分布在这一区域，待调用函数 __vbaVarAdd 的地址被装入 EBX。继续向下，来到如下代码处。

```
6A384610  MOVSX EDI,WORD PTR DS:[ESI]              ;取加法指令的操作数
6A384613  ADD EDI,EBP
6A384615  PUSH EDI                                 ;操作数入栈
6A384616  CALL EBX                                 ;这里调用函数__vbaVarAdd 执行加法操作
6A384618  PUSH EDI                                 ;运算结果保存在栈结构中
6A384619  XOR EAX,EAX
6A38461B  MOV AL,BYTE PTR DS:[ESI+2]               ;继续取下一条伪指令的操作码
6A38461E  ADD ESI,3                                ;指向下一条伪指令的操作数
6A384621  JMP DWORD PTR DS:[EAX*4+6A37DA58]        ;跳向下一条伪指令的解释单元
```

在 OllyDbg 朴素而强有力的逐行追击中，VB P-code 虚拟机的解释机理一览无余。

C.3.2　动态分析 VB P-code 程序

毋庸置疑，OllyDbg 完全可以胜任 VB P-code 程序的调试，然而在大部分情况下，它并非最好的选择。基于通常的调试习惯，人们总是希望能够直接跟踪可执行文件本身的每一条指令，以便直观地了解程序实现的功能。从这个角度来看，尽管调试工具是分析 VB P-code 虚拟机的好手，却无法将更多的注意力集中在 P-code 程序本身的功能上。

伴随这样一种想法，功能强大的 VB P-code 专用调试器 WKTVBDebugger 应运而生。这个具有划时代意义的 VB P-code 调试器翻开了解释型语言程序调试策略的崭新一页，它不仅给 VB P-code 程序的调试带来了极大的方便，其编写思想和运作机制也给人以深刻的启发。在本节中，笔者将简要介绍这款调试器的各种特性和使用方法。

1．介绍

WKTVBDebugger 以单个伪代码指令为执行单元，跟踪、设断、修改等方面都完全以伪代码为基础，彻底屏蔽了虚拟机的解释细节，如同直接调试 P-code 可执行文件的功能一样。举例来说，当程序执行一条 Variant 型变量的加法伪指令时，读者在调试器中跟踪的将不再是冗长繁复的虚拟机解释代码，而是 AddVar 这一条指令（严格地说，AddVar 只是助记符）。不仅操作码如此，每一条伪指令的操作数也可以通过栈来方便地观察。

在基于本地机器码的调试器实现中，通过设置单步跟踪标志位，可以在每执行一条指令以后产生单步调试断点，从而将控制权转移给调试器，调试器在执行下一条指令前可以观察寄存器的状态并修改程序的执行流程。此外，调试器还可以通过在指定的指令位置插入 CC（INT 3）来设置断点，捕获异常并取得控制权。特别地，在 Win32 环境下，这类断点会在 Ring 3 级形成异常，由 Windows 系统向调试器报告异常事件，调试器由此获得处理异常的优先权。然而，对于由虚拟机解释执行的伪代码，因为系统并未给程序开发人员特意留出标准的调试接口，异常的处理对用户而言也是透明的，所以传统的调试器原理便无法照搬过来。

WKTVBDebugger 的实现与 VB P-code 虚拟机运作机制及伪指令的研究密不可分。其作者 Mr. Silver 和 Mr. Snow 采用了一种类似于 Hook 的方法，把调试器代码插入虚拟机和 VB P-code 伪代码之间，从而巧妙地解决了让调试器取得控制权的问题。观察一下 VB P-code 虚拟机伪代码读取引擎，具体如下。

```
XOR EAX,EAX
MOV AL,BYTE PTR DS:[ESI+2]
ADD ESI,3
JMP DWORD PTR DS:[EAX*4+6A37DA58]
```

在这一模式中，寄存器 ESI 始终指向伪指令流，虚拟机读取下一条伪指令的操作码以后，将 ESI 指向次级操作码（对于具有多级操作码的伪指令）或者操作数（对于具有单级操作码的伪指令），每一条 jmp 指令都通过跳转地址表移向下一条伪指令的解释单元——如果让这里的跳转地址指向自定义的调试代码，不就可以在下一条伪指令执行前取得控制权了吗？调试所需的初始化工作，仅仅是把 6A37DA58 所指向的跳转地址表中的所有项替换为调试循环代码的入口地址，一旦在此截获 P-code 指令，由于 AL 中保存着伪指令的操作码，而 ESI 指向伪指令的操作数，调试器便可以把对应的伪指令助记符在反编译窗口中显示出来，并判断当前的地址是

否需要中断——无论如何，程序的执行流程已在掌握。一旦调试器的任务完成，就恢复现场的寄存器环境，根据原来的跳转地址表移至下一条指令的解释单元。这就是 WKTVBDebugger Hook 的基本原理。

调试器的基本框架有了，随之而来的是伪指令的解释问题。微软定义了 VB P-code 指令规范，却没有把它们公开。这不是一个大问题，无论如何，这套指令规范是可以获得的。在这里要感谢 Josephco，他的 VB P-code 反编译器 Exdec 在这方面做了许多先驱性的工作。另外，笔者推荐两款相当不错的国产 VB P-code 反编译器，分别是万涛编写的 VBExplorer 和 ljtt 编写的 VBParser。严格地说，VBExplorer 不是一个纯粹的反编译器，它还可以编辑和修改 VB 控件的各种属性，但反编译功能做得还不精。相比之下，VBParser 就是一个专业的反编译器了。

2．安装

执行 WKTVBDebugger 文件就能完成安装。但有几个方面要注意一下，如果不能正常装载程序，可以进行如下尝试。

- 将目标软件复制到 WKTVBDebugger 安装目录里调试，即与 Loader.exe 同一目录。
- 将安装目录的 WKTVBDE.dll 文件复制到系统目录里。
- 将 MSVBVM60.DLL 替换成 2003 年以前的版本。

3．使用

运行 WKTVBDebugger 后，打开目标程序，单击菜单“Action”→“Run”装载程序。输入用户名及序列号后，单击目标程序上的“确定”按钮就能中断在 WKTVBDebugger 里，如图 C.4 所示。

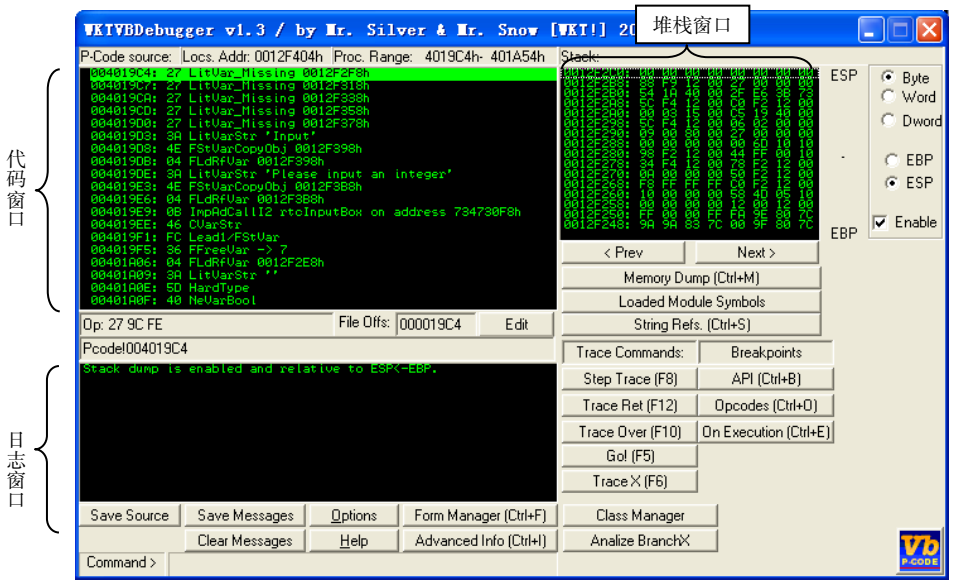


图 C.4　WKTVBDebugger 主窗口

主窗口由 3 个基本部分组成，分别是代码（Disassembly）、日志（Log）及栈（Stack）窗口。

(1) 代码窗口

这个窗口显示当前执行程序的反汇编伪代码，指令格式如下。

```
XXXXXXXX:XX 指令
```

其中“XXXXXXXX”是指令的内存虚拟地址，跟着的两个数字是以十六进制格式显示的机器码。注意，只有第 1 个机器码被显示了，后面的是 P-code 伪指令（会根据变量和自变量的不同而不同）。另外，在该窗口中单击右键将打开命令菜单。

(2) 日志窗口

当程序执行时，显示程序中所有变量的信息和提示消息。所以，跟踪调试时这个窗口是很重要的，它将提供一些有价值的信息，最重要的是，它将显示当前指令执行时所进行的操作。

(3) 栈窗口

栈窗口有几个模式，例如字节、字、双字。P-code 的运行有别于传统的 CPU 结构，传统的 CPU 执行依赖于寄存器和栈，而 P-code 只使用栈，所以栈窗口非常重要，各种指令都通过栈来交换数据。

4．机器码与助记命令

机器码与助记命令表（Opcode and Mnemonics Table）很重要，具体参考 WKTVBDebugger 的帮助文件。掌握 VB P-code 的关键就在这些助记命令上。这里只列出几个常用的助记命令。

- BranchF：机器码是 1Ch，3 字节，条件跳转指令（如果栈的值是 0 就跳转）。单击“Analize BranchX”按钮可以了解当前进程中所有条件跳转指令的位置。
- BranchT：机器码是 1Dh，3 字节，条件跳转指令（如果栈的值是 FFFFFFFh（-1）就跳转）。
- Branch：机器码是 1Eh，3 字节，无条件转移。
- EqVarBool：机器码是 33h，1 字节，比较指令（比较两个变量，根据结果将 -1 或 0 压入栈）。
- LitI2_Byte：机器码是 F4h，2 字节，将数据压入栈。
- ConcatStr：机器码是 2Ah，1 字节，字符串连接指令（相当于 C 语言中的 strcat 函数）。此指令单步跟踪（Step Trace）时，会在日志窗口留下相应字符串的连接结果。所以，可以单击“Opcodes”按钮在此伪指令处设置断点，以便了解某字符串的值。
- FLdZeroAd / CVarStr：取字符串指令，特点与 ConcatStr 相同。

C.3.3　伪代码的综合分析

尽管微软为 VB P-code 定义了一整套伪指令助记符，但并没有权威的技术文档解释这些伪指令执行的细节。既然 WKTVBDebugger 作为一个伪代码级的调试器已经屏蔽了 VB P-code 虚拟机的解释过程，为什么我们还要颇费周折地去了解这些伪指令执行的细节呢？在调试的过程中，读者一定会遇到这样的问题：假设用 WKTVBDebugger 步过了 AddVar 这条伪指令，试问如何才能获得它的操作数和操作结果？既然 VB P-code 虚拟机是基于堆栈的，那么操作数和操作

结果一定存放在堆栈中。实际情况诚然如是，然而它们究竟是以怎样的形式存在的呢？是单个的操作数，是指针，还是其他复杂的数据结构？对于不同的 P-code 伪指令，其存放形式也是迥然相异的。如果在调试过程中无法正确地获知和修改每条指令的操作数和操作结果，那么 WKTVBDebugger 的功能也就止步于静态反编译了。

这里仍然以 VBP-code-Ex.exe 为例说明如何用现有的工具来解决上述困惑。首先使用 ljtt 的 VBParser 解析 VB P-code.exe，得到如下伪代码。

```
-----=====-----=====-----=====--Pcode--=====-----=====-----=====-----
[CommandButton]
Private Sub Command1_Click()
'-=-=-=-=-=-= ProcAddr Range: [004019C4 - 00401A54] , ProcSize: 90 =-=-=-=-=-=-
004019C4: 27 9C FE         LitVar_Missing      PushVarError 80020004 (missing)
004019C7: 27 BC FE         LitVar_Missing      PushVarError 80020004 (missing)
004019CA: 27 DC FE         LitVar_Missing      PushVarError 80020004 (missing)
004019CD: 27 FC FE         LitVar_Missing      PushVarError 80020004 (missing)
004019D0: 27 1C FF         LitVar_Missing      PushVarError 80020004 (missing)
*********** Referent String: "Input" ***********
004019D3: 3A 4C FF 00 00   LitVarStr           PushVarString Ptr_00401434
004019D8: 4E 3C FF         FStVarCopyObj       [local_C4]=vbaVarDup(Pop)
004019DB: 04 3C FF         FLdRfVar            Push local_C4
*********** Referent String: "Please input an integer" ***********
004019DE: 3A 6C FF 01 00   LitVarStr           PushVarString Ptr_00401400
```

以上就是 Command_Click 事件响应代码的开头部分，由于 VB P-code 虚拟机以流的形式顺序读入每一句伪指令，然后通过一个跳转地址表找到相应的解释代码，所以，为了跟踪它解释伪指令的细节，就必须在伪指令的操作码上下内存访问断点。第 1 句伪指令 LitVar_Missing 从 004019C4（虚拟地址）开始。用 OllyDbg 加载 VBP-code-Ex.exe，在转储窗口中来到 004019C4 处，对第 1 个字节（操作码）下内存访问断点，按“F9”键执行，单击“OK”按钮，中断如下。

```
6A37D153  MOV AL,BYTE PTR DS:[ESI]              ;开始读操作码，ESI=004019C4
6A37D155  INC ESI                               ;使 ESI 指向操作数
6A37D156  JMP DWORD PTR DS:[EAX*4+6A37DA58]     ;根据跳转地址表和操作码寻址解释单元
```

按“F8”键往下走一步，来到这条伪指令的解释单元处，代码如下。

```
6A37D39F  MOVSX EDI,WORD PTR DS:[ESI]           ;把字操作数带符号扩展到 EDI
6A37D3A2  ADD ESI,2                             ;ESI 指向下一句伪指令的操作码
6A37D3A5  MOV WORD PTR DS:[EDI+EBP],0A          ;EBP 是程序栈区某处的基址，但不是栈顶的指针
                                                ;它把 0A 保存到 EDI 指向的偏移地址处
6A37D3AB  MOV DWORD PTR DS:[EDI+EBP+8],80020004 ;向下偏移 8 个字节处存入 80020004
          ;根据 VBParser 的说明，这个数字表示空参数（默认参数），事实上在源代码中没有提供这个参数
6A37D3B3  ADD EDI,EBP                           ;使 EDI 指向 0A
6A37D3B5  PUSH EDI                              ;在栈中压入这个虚拟地址
6A37D3B6  XOR EAX,EAX                           ;清空 EAX，准备读取下一条伪指令
6A37D3B8  MOV AL,BYTE PTR DS:[ESI]              ;读取下一条伪指令的操作码
6A37D3BA  INC ESI                               ;使 ESI 指向下一条伪指令的次级操作码或操作数
6A37D3BB  JMP DWORD PTR DS:[EAX*4+6A37DA58]     ;根据地址跳转表和操作码寻址解释单元
```

看一下这些指令执行完以后的栈。

```
0012F458   0012F494                    ;栈顶
0012F45C   00000000
0012F460   00000000
.....
.....
0012F488   00000000
0012F48C   00000000
0012F490   00000000
0012F494   0000000A                    ;这就是刚才压入栈顶的数据
0012F498   00000000
0012F49C   80020004
0012F4A0   00000000
```

现在这个伪指令的动作已经明确了。在 LitVar_Missing 执行时，把一个虚拟地址压入栈，这个虚拟地址指向 0000000A、00000000、80020004。实际上，这句伪指令的功能就是在栈中提供一个空参数，其栈完全没有参考价值。在下面的说明中，笔者将省略对虚拟机伪代码读取引擎的注释，因为这部分都是一样的。

现在来看看 004019D3 处的伪指令 LitVarStr。老规矩，在 004019D3 处设内存访问断点，按“F9”键中断在如下代码处。

```
6A37D3B8  MOV AL,BYTE PTR DS:[ESI]          ;esi=004019D3
6A37D3BA  INC ESI
6A37D3BB  JMP DWORD PTR DS:[EAX*4+6A37DA58]
```

跟随跳转来到如下代码处。

```
6A37D3C2  MOVSX EDI,WORD PTR DS:[ESI]   ;第 1 个字操作数 FF4C（栈区偏移量），带符号扩展到 EDI
6A37D3C5  MOVZX EAX,WORD PTR DS:[ESI+2] ;第 2 个字操作数 0000（数据区偏移量），无符号扩展到 EAX
6A37D3C9  MOV EDX,DWORD PTR SS:[EBP-54] ;取得 P-code 程序数据区的基址
6A37D3CC  MOV EAX,DWORD PTR DS:[EDX+EAX*4]  ;根据 EAX 产生偏移量，取得数据区的数据
                                            ;这是一个虚拟地址，指向 Unicode 字符串“Input”
6A37D3CF  ADD EDI,EBP                       ;使 EDI（栈区偏移量）指向栈区即将保存数据的地方
6A37D3D1  MOV WORD PTR DS:[EDI],8           ;用于识别数据类型
6A37D3D6  MOV DWORD PTR DS:[EDI+8],EAX      ;向下偏移 8 字节处存入指向 Unicode 字符串“Input”
的虚拟地址
6A37D3D9  PUSH EDI                          ;最后，栈区数据指针入栈
6A37D3DA  XOR EAX,EAX
6A37D3DC  MOV AL,BYTE PTR DS:[ESI+4]
6A37D3DF  ADD ESI,5
6A37D3E2  JMP DWORD PTR DS:[EAX*4+6A37DA58]
```

同样地，有必要观察一下栈，具体如下。

```
0012F444   0012F544                          ;栈顶
0012F448   0012F514                          ;下面是前面的其他指令形成的栈
0012F44C   0012F4F4
0012F450   0012F4D4
0012F454   0012F4B4
0012F458   0012F494
0012F45C   00000000
.....
.....
0012F540   00000000
0012F544   00000008
0012F548   00000000
0012F54C   00401434  UNICODE "Input"
0012F550   00000000
```

结合上述跟踪分析，LitVarStr 伪指令对操作数的获取方法就很清楚了：在转储窗口观察 0012F544 处的内容，向后移动 8 字节，得到虚拟地址 00401434，00401434 处的内容就是入栈的字符串参数。

相应地，笔者在 WKTVBDebugger 中演示一下操作过程。

①用 WKTVBDebugger 加载 VB P-code.exe。

②在 Form Manager 中对 Command1 控件设置断点。

③单击“OK”按钮。

④WKTVBDebugger 中断在如下代码处。

```
004019C4: 27 LitVar_Missing 0012F474h
004019C7: 27 LitVar_Missing 0012F494h
004019CA: 27 LitVar_Missing 0012F4B4h
004019CD: 27 LitVar_Missing 0012F4D4h
004019D0: 27 LitVar_Missing 0012F4F4h
004019D3: 3A LitVarStr 'Input'               ;这就是在 OllyDbg 中跟踪分析的 LitVarStr 伪指令
004019D8: 4E FStVarCopyObj 0012F514h
004019DB: 04 FLdRfVar 0012F514h
004019DE: 3A LitVarStr 'Please input an integer'
004019E3: 4E FStVarCopyObj 0012F534h
004019E6: 04 FLdRfVar 0012F534h
004019E9: 0B ImpAdCallI2 rtcInputBox on address 73472265h
```

⑤注意注释标出的这条伪指令，单步走过这条指令时，右上角栈窗口显示如下（为了便于观察，在右侧的单选框中选择“DWORD”选项）。

```
0012F424: 0012F524 0012F4F4                  ;注意这里的栈顶 0012F524
0012F41C: 0012FE3C 0000004E
.....
0012F3B4: 00000000 00000000
0012F3AC: 77E6780F 0000008C
```

⑥按“Ctrl+M”组合键打开转储窗口，在“Address to Dump”组合框中输入“0012F524”，得到如下结果。

```
0012F524:08 00 00 00 00 00 00 00
0012F52C:34 14 40 00 00 00 00 00        ;向下移动 8 字节，00401434 就是对应字符串的指针
0012F534:00 00 00 00 00 00 00 00
```

⑦记下这个指针，将其输入“Address to Dump”组合框，这个 Unicode 字符串终于露出了真面目，具体如下。

```
00401434:49 00 6E 00 70 00 75 00        I.n.p.u.
0040143C:74 00 00 00 00 00 00 00        t.......
00401444:00 00 00 00 E1 4E AD 33        ....酸?
```

当然，就该指令本身而言，WKTVBDebugger 已经在日志窗口中输出了其操作数，所以要观察这个字符串大可不必那么麻烦。但是，对于其他没有日志记录的伪指令，这套分析方法仍具有启发性。

C.3.4　VB P-code 攻击实战

本节用 CyberBlade 编写的一个中等难度的 VB P-code CrackMe 作为范例来初步介绍 VB P-code 程序的调试过程，以帮助读者熟悉一些常见的 P-code 伪指令。

由于反编译器通常会提供一些有用的信息，因此首先用 Josephco 的 Exdec 生成该 CrackMe 的反编译代码，对照 WKTVBDebugger 的代码窗口进行调试。在 WKTVBDebugger 的 Form Manager 中对“Check”按钮下断点，填写用户名“cyclotron”和序列号“131421”，单击“Check”按钮以后，首先是关于用户名和序列号的存在性校验，代码如下。

```
40E26C: 04 FLdRfVar             local_009C
40E26F: 21 FLdPrThis
40E270: 0f VCallAd              text
40E273: 19 FStAdFunc            local_0098
40E276: 08 FLdPr                local_0098
40E279: 0d VCallHresult         get__ipropTEXTEDIT    ;调用函数取得用户名
40E27E: 6c ILdRf                local_009C
40E281: 1b LitStr:
40E284: Lead0/30 EqStr                                ;用户名是否存在
40E286: 2f FFree1Str            local_009C
40E289: 1a FFree1Ad             local_0098
40E28C: 1c BranchF:             40E2C1
.................
.................
40E2CE: 0d VCallHresult         get__ipropTEXTEDIT    ;调用函数取得序列号
40E2D3: 6c ILdRf                local_009C
40E2D6: 1b LitStr:
40E2D9: Lead0/30 EqStr                                ;序列号是否存在
40E2DB: 2f FFree1Str            local_009C
40E2DE: 1a FFree1Ad             local_0098
40E2E1: 1c BranchF:             40E316
40E2E4: 27 LitVar_Missing
40E2E7: 27 LitVar_Missing
40E2EA: 3a LitVarStr:           (local_00CC) Error
40E2EF: 4e FStVarCopyObj        local_00DC
```

```
40E2F2: 04 FLdRfVar           local_00DC
40E2F5: f5 LitI4:             0x40  64  (...@)
40E2FA: 3a LitVarStr:         (local_00AC) You have to enter a key first.
```

接着是关于序列号长度合法性的校验，代码如下。

```
40E323: 0d VCallHresult       get__ipropTEXTEDIT    ;调用函数取得序列号
40E328: 6c ILdRf              local_009C
40E32B: 1b LitStr:
40E32E: Lead0/30 EqStr                              ;序列号是否大于等于 5 位
40E330: 2f FFree1Str          local_009C
40E333: 1a FFree1Ad           local_0098
40E336: 1c BranchF:           40E36B
40E339: 27 LitVar_Missing
40E33C: 27 LitVar_Missing
40E33F: 3a LitVarStr:         ( local_00CC ) Error
40E344: 4e FStVarCopyObj      local_00DC
40E347: 04 FLdRfVar           local_00DC
40E34A: f5 LitI4:             0x40  64  (...@)
40E34F: 3a LitVarStr:         ( local_00AC ) You have to enter at least 5 chars.
40E354: 4e FStVarCopyObj      local_00BC
40E357: 04 FLdRfVar           local_00BC
```

从下面开始的两个循环是这个 CrackMe 的算法核心，涉及的指令和函数对 P-code 程序的调试都具有相当重要的指导意义。

第 1 个循环（For 结构）的代码如下。

```
40E36B: 28 LitVarI2:          (local_00EC)0x1 (1)   ;立即数 1 入栈，Lit 表示立即数，作为循环的初始值
40E370: 04 FLdRfVar           local_012C            ;入栈，保存循环计数器
40E373: 04 FLdRfVar           local_009C
40E376: 21 FLdPrThis
40E377: 0f VCallAd            text
40E37A: 19 FStAdFunc          local_0098
40E37D: 08 FLdPr              local_0098
40E380: 0d VCallHresult       get__ipropTEXTEDIT    ;调用函数取得用户名
40E385: 6c ILdRf              local_009C
40E388: 4a FnLenStr                                 ;取用户名的长度
40E389: Lead2/69 CVarI4       local_00CC            ;转换（C-Convert）为变体型（Var-Variant），以用户名长度为循环次数
40E38D: 2f FFree1Str          local_009C
40E390: 1a FFree1Ad           local_0098
40E393: Lead3/68 ForVar:      (when done) 40E3F5    ;ForVar 开始循环计算
40E399: 04 FLdRfVar           local_009C
40E39C: 21 FLdPrThis
40E39D: 0f VCallAd            text
40E3A0: 19 FStAdFunc          local_0098
40E3A3: 08 FLdPr              local_0098
40E3A6: 0d VCallHresult       get__ipropTEXTEDIT    ;取用户名的长度
40E3AB: 04 FLdRfVar           local_0094            ;用户名的指针入栈
40E3AE: 28 LitVarI2:          (local_00DC)0x1 (1)   ;立即数 1 入栈
```

```
40E3B3: 04 FLdRfVar            local_012C        ;循环计数器转换为整型(I-Integer)
40E3B6: Lead1/22 CI4Var
40E3B8: 3e FLdZeroAd           local_009C
40E3BB: 46 CVarStr             local_00BC        ;转换为变体型
40E3BE: 04 FLdRfVar            local_00FC        ;用户名字符串指针入栈
40E3C1: 0a ImpAdCallFPR4:                        ;调用 rtcMidCharVar 函数，每轮循环取用户名
的 1 个字符
40E3C6: 04 FLdRfVar            local_00FC        ;取得的字符入栈
40E3C9: Lead2/fe CStrVarVal    local_0150        ;转换为字符串型(Str-String)
40E3CD: 0b ImpAdCallI2                           ;调用 rtcAnsiValueBstr 函数，转换字符为
ASCII 码
40E3D2: 44 CVarI2              local_00CC        ;转换为变体型,将每轮循环得到的十进制数作为字
符串相连接
40E3D5: Lead0/ef ConcatVar

40E3D9: Lead1/f6 FstVar                          ;保存字符串（St－Save to）
40E3DD: 2f FFree1Str           local_0150
40E3E0: 1a FFree1Ad            local_0098
40E3E3: 36 FFreeVar
40E3EC: 04 FLdRfVar            local_012C
40E3EF: Lead3/7e NextStepVar:(continue) 40E399   ;继续下一轮循环
```

这个循环的算法总结如下。

①逐位读取用户名字符："cyclotron" → 'c'，'y'，'c'，'l'，'o'，'t'，'r'，'o'，'n'。

②分别转换为十进制的数字字符串：99，121，99，108，111，116，114，111，110。

③顺序连接所有的数字字符串："9912199108111116114111110"。记为 S1（注意：这个字符串是以 Unicode 编码的形式在内存中出现的）。

下面来看第 2 个循环（While 结构），代码如下。

```
40E3F5: 04 FLdRfVar        local_0094          ;第 1 个循环生成的数字字符串入栈
40E3F8: Lead0/eb FnLenVar                      ;取其长度
40E3FC: 28 LitVarI2:       (local_00AC)0x9(9)  ;立即数 9 入栈
40E401: 5d HardType
40E402: Lead0/74 GtVarBool                     ;是否大于 9 位
40E404: 1c BranchF:        40E425              ;小于等于 9 位则终止循环
40E407: 04 FLdRfVar            local_0094
40E40A: Lead3/c4 LitVarR8                      ;浮点立即数 3.141592654000000 入栈
                                               ;R8-Real number of 8 bytes
```

在这个位置，WKTVBDebugger 会遇到一点小小的困难——它无法现场输出这个浮点立即数的标准形式。如何获得这个浮点数呢？可以利用 OllyDbg 的数据窗口以标准形式输出浮点数，代码如下。

```
40E416: Lead0/bc DivVar ;做除法，先入栈的是被除数，后入栈的是除数，其他算术运算指令也遵循这个规则
40E41A: Lead0/e1 FnFixVar                  ;对除法的结果取整
40E41E: Lead1/f6 FstVar                    ;保存结果为 Variant 型
40E422: 1e Branch:             40e3f5      ;回到 40e3f5 循环运算
40E425: 04 FLdRfVar            local_0094
40E428: Lead3/c1 LitVarI4:     (local_param_5678FF54) 0x30f85678  (821581432)
```

```
                                        ;立即数 0x30f85678 入栈
40E430: Lead0/17 XorVar                 ;Xor，异或运算
40E434: Lead1/f6 FstVar                 ;保存为变体型
40E438: 04 FLdRfVar          local_0094 ;前面的运算结果入栈
40E43B: 08 FLdPr             local_param_0008
40E43E: 8a MemLdStr                     ;调入一个内存操作数 0D8B3h
40E441: Lead2/69 CVarI4      local_00AC ;转换为变体型
40E445: Lead0/9c SubVar                 ;两数相减
40E449: Lead1/f6 FstVar                 ;保存为变体型，记为 S2
```

上述循环运算过程可以总结如下。

①变换 S1 到 9 位以下的十进制形式，代码如下。

```
while ( strlen(S1) > 9 )
S1 = Fix ( S1/3.141592654 )
```

②变换 S1 到 S2：

$$S2 = (S1 \text{ xor } 30F85678h) - 0D8B3h = 667574632$$

接下来，进入另一个循环，代码如下。

```
40E44D: 28 LitVarI2:         (local_00EC)0x1(1)   ;立即数 1 入栈，作为循环的初始值
40E452: 04 FLdRfVar          local_012C           ;循环计数器
40E455: 28 LitVarI2:         (local_00CC)0xa(10)  ;循环终止值 10
40E45A: Lead3/68 ForVar:     (when done) 40E495   ;进入循环
40E460: 04 FLdRfVar          local_009C
40E463: 21 FLdPrThis
40E464: 0f VCallAd           text
40E467: 19 FStAdFunc         local_0098
40E46A: 08 FLdPr             local_0098
40E46D: 0d VCallHresult      get__ipropTEXTEDIT   ;调用函数取得序列号
40E472: 6c ILdRf             local_009C
40E475: 04 FLdRfVar          local_012C
40E478: Lead1/22 CI4Var                           ;转换为整型
40E47A: 08 FLdPr             local_param_0008
40E47D: 06 MemLdRfVar        local_param_0034
40E480: 9e Ary1LdI4                               ;从字符串数组中依次取出一系列字符串
    UNICODE "373703670"
    UNICODE "684708686"
    UNICODE "698673531"
    UNICODE "391184533"
    UNICODE "329528230"
    UNICODE "654824169"
    UNICODE "557168731"
    UNICODE "387375850"
    UNICODE "212298498"
    UNICODE "851143730"
40E481: Lead0/30 EqStr                            ;上面这些字符串依次同序列号比较
40E483: 2f FFree1Str         local_009C
40E486: 1a FFree1Ad          local_0098
40E489: 1c BranchF:          40E48C    ;奇怪的跳转，比较结果为“False”就转移到下一句执行
40E48C: 04 FLdRfVar          local_012C
```

```
40E48F: Lead3/7e NextStepVar:   (continue) 40E460     ;继续下一轮循环
```

令人意外的是，这个循环对程序的运行路线没有任何影响，可鉴定为程序作者布下的迷魂阵。尽管如此，距离最后的验证也已经不远了，代码如下。

```
40E4A2: 0d VCallHresult        get__ipropTEXTEDIT     ;调用函数取得序列号
40E4A7: 3e FLdZeroAd           local_009C
40E4AA: 46 CVarStr             local_00BC             ;转换为变体型
40E4AD: 04 FLdRfVar            local_0094             ;用户名的计算结果 S2 入栈
40E4B0: Lead0/9c SubVar                               ;序列号减去 S2 得到 S3
40E4B4: 04 FLdRfVar            local_0150
40E4B7: 21 FLdPrThis
40E4B8: 0f VCallAd             text
40E4BB: 19 FStAdFunc           local_0174
40E4BE: 08 FLdPr               local_0174
40E4C1: 0d VCallHresult        get__ipropTEXTEDIT     ;调用函数取得用户名
40E4C6: 6c ILdRf               local_0150
40E4C9: 4a FnLenStr                                   ;取用户名长度
40E4CA: Lead2/69 CVarI4        local_00AC             ;转换为变体型
40E4CE: 5d HardType
40E4CF: Lead0/33 EqVarBool                            ;是否与 S3 相等
40E4D1: 2f FFree1Str           local_0150
40E4D4: 29 FFreeAd:
40E4DB: 35 FFree1Var           local_00BC
40E4DE: 1c BranchF:            40E55B                 ;真正的关键跳转
40E4E1: 27 LitVar_Missing
40E4E4: 27 LitVar_Missing
40E4E7: 3a LitVarStr:          ( local_00CC ) Correct key
40E4EC: 4e FStVarCopyObj       local_00DC
40E4EF: 04 FLdRfVar            local_00DC
40E4F2: f5 LitI4:              0x40  64  (...@)
40E4F7: 3a LitVarStr:          ( local_00AC ) Wow, you have found a correct key!
40E4FC: 4e FStVarCopyObj       local_00BC
40E4FF: 04 FLdRfVar            local_00BC
40E502: 0a ImpAdCallFPR4:        □
40E507: 36 FFreeVar
40E512: 27 LitVar_Missing
40E515: 27 LitVar_Missing
40E518: 3a LitVarStr:          ( local_00CC ) Correct key!
40E51D: 4e FStVarCopyObj       local_00DC
40E520: 04 FLdRfVar            local_00DC
40E523: f5 LitI4:              0x40  64  (...@)
40E528: 3a LitVarStr:          ( local_00AC ) Mail me, how you got it
40E52D: 4e FStVarCopyObj       local_00BC
```

这部分的算法总结如下。

$$\text{strlen(用户名)} = \text{序列号} - \text{S2}$$

据此得到正确的注册信息：

$$\text{注册码} = \text{strlen (" cyclotron ")} + 667574632 = 667574641$$

附录 D　加密算法变形引擎[①]

本章主要对算法变形引擎的技术原理进行了说明，但其中有很多不完善的地方，更多的时候是在讲解一个自定义虚拟机及汇编器的设计与实现。在本章的最后提出了在自定义汇编下自动实现“加密算法”多态的思路，不过这个实现也不是非常完善，还有很大的提升空间。

D.1　算法变形引擎

变形技术是通过变形引擎实现的，在不影响代码逻辑功能的前提下，通过改变代码流程、指令等，使代码呈现出不同的形式。

D.1.1　变形引擎原理

变形引擎最先在病毒中使用，其目的是提取对抗特征码。变形引擎的基础模式如下，其中“aaa”等表示一些汇编指令，这里就不列出具体指令了。

- 解密代码（用来解密加密后的病毒体），示例如下。

```
aaa
bbb
ccc
```

- 加密后的病毒体，示例如下。

```
;调用(call)感染模块
aaa
bbb
ccc
```

- 感染模块，示例如下。

```
aaa
bbb
ccc
;调用(call)算法变形引擎
aaa
bbb
ccc
```

- 算法变形引擎（生成新的解密代码并写入新的病毒体中），示例如下。

```
aaa
bbb
ccc
```

以上模式的原理是：利用算法变形引擎，将解密头算法进行变形。其实，与其叫作“解密

① 本章由看雪论坛“编程技术”版主玩命（阎文斌）编写。

算法”，不如叫作“模式处理”。例如，$y = x + 19 - 93h$ 这样的式子，其中 x 是要加密的字节，而 y 是加密后的字节，那么解密公式就变成 $x = y + 93h - 19h$，而其中的 19h、93h 是一个使用随机数控制的所谓密钥。在数学意义上，其实它并没有所谓的强度，但是在生成的汇编代码中却达到了特征变换的效果，在实际编程上的意义也可以得到实现（因为它的实现并不困难）。

笔者于 2008 年写了一篇关于 Vulcano 病毒分析的文章，其中详细分析了一个完整的算法变形引擎（https://bbs.pediy.com/thread-78862.htm）。

为什么称作“算法变形引擎”，而不直接叫“变形引擎”呢？就笔者的理解，病毒体并没有实现真正的变形。本节介绍的并非以上这种引擎，而是其中的一个子集，其作用也并非是对抗特征码扫描，而是在软件保护机制中放置一个可变换的“加解密算法”的模块，这样每次加密时就使用新的算法进行解密（毕竟在很多情况下，破解者对加解密算法的强度不是很在意，他们只关心解密的整体流程）本节的这个小程序是用来对抗静态脱壳机的，它是一个基本框架。

D.1.2　代码变形引擎 polycypt 简介

我们可以从 https://git.coding.net/devilogic/polycypt.git 中迁出一套完整的 polycypt 源代码。

一边看代码一边完成一个简单的算法变形的框架。这里利用了虚拟解释语言的一个原理：先编写一个虚拟机，只负责解释我们自己定义的指令；然后，只处理我们自己规定的一个可执行文件格式；随后，写一个对应的汇编器负责汇编代码；最后，编写一个变形引擎，每次由这个引擎生成加解密代码。在加密程序中使用虚拟机执行随机加密算法字节码，在解密体中使用虚拟机执行随机解密算法字节码。

D.1.3　polycypt 的编译与使用

目前的 polycypt 只是一个基础版本，算法的变形力度不够。其子模块如下。

- pcvm 虚拟机。
- pcasm 汇编器。
- polycypt 算法变形引擎。

使用 xxx_main.cpp 进行外部测试（可单独编译成库使用）。使用 polycypt 随机生成加解密算法，执行“polycypt -t template_path result_dir”命令。在 result_dir 中生成 4 个文件，分别是 encrypt.asm、decrypt.asm、encrypt.pbc、decrypt.pbc，其中 .asm 文件是生成的汇编源代码，用于查看；.pbc 是编译后的字节码文件，用 pcvm 执行；template_path 是模版路径，用于支持算法。

1．pcvm 的编译与使用方法

编译 pcvm 时需要导入的文件包括 pcfile.h、pcvm.h、pcvm.cpp、pcvm_main.cpp。开启调试器，代码如下。

```
if (vm.run((unsigned char*)codes, codesize, entry, false) == false) {
  printf("error : %d\n", vm.error());
}
```

虚拟机 run 接口的最后一个参数如果设置为“true”，则启动调试器模式。运行后会出现一个简单的调试器，如图 D.1 所示。

调试指令只有如下 3 条。

- c：继续运行。
- q：退出。
- h：帮助。

使用命令如下。

```
usage: pcvm [options] <bytecode file>
-d              disasm
-i <port> <file>  bind input io to file
-o <port> <file>  bind output io to file
```

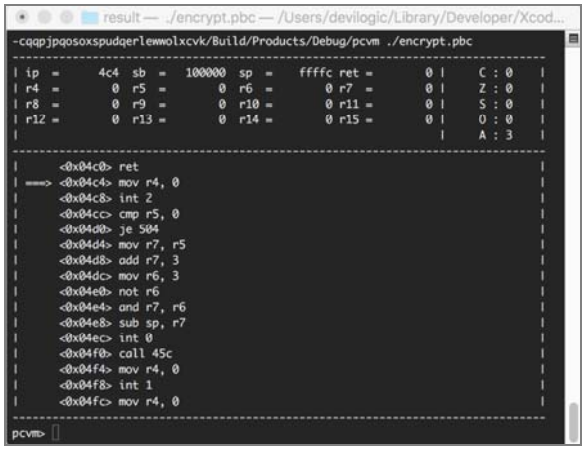


图 D.1　简单的调试器

2．pcasm 的编译与使用方法

编译 pcasm 时需要导入的文件包括 pcfile.h、pcvm.h、pcasm.h、pcasm.cpp、pcvm_main.cpp，使用命令如下。

```
usage: pcasm <asm file> [out file]
```

3．polycypt 的编译与使用方法

编译 polycryt 时需要导入的文件包括 pcfile.h、pcasm.h、pcasm.cpp、pcasm.cpp、polycypt_alg0.cpp、polycypt_alg0.h、polycypt_error.h、polycypt_factory.cpp、polycypt_factory.h、polycypt.cpp、polycypt.h、polycypt_main.cpp，使用命令如下。

```
usage: polycypt [options] <output path>
-t <template dir path>
```

D.1.4　虚拟机的设计与实现

这里的虚拟机设定不必太过复杂，因为我们要达到的目的就是进行基本的算术运算，从而控制一些基本流程。但是，对一些辅助操作还是要考虑，例如虚拟机对外的数据读入与输出。核心的源文件只有一个，即 pcvm.cpp。要想将虚拟机嵌入自己的代码中，只要使用这个文件即可。

1．pcvm 类的结构说明

我们首先按照重要程度来分析一下 pcvm 这个类，示例如下。

```
class pcvm {
public:
  pcvm();
  virtual ~pcvm();

public:
  /* 字节序的确定，但在这个版本中没有用（估计以后也不会支持） */
  static bool is_big_endian();

  /* 调试支持，用于调试 PCVM */
#ifdef SUPPORT_DEBUGER
  /* 反汇编指定的缓存代码 */
  bool disasm_all(unsigned char *buf, size_t bufsize);
#endif
  /* 从偏移 entry_offset 处运行 codes 缓存中 cs 字节长的代码 */
  /* 这里支持一个调试选项：如果将 debug 设置为“true”，运行后会启动一个简单的调试器控制台 */
  bool run(const unsigned char *codes,
          size_t cs,
          unsigned entry_offset=0
#ifdef SUPPORT_DEBUGER
          ,bool debug=false
#endif
         );

 /* 以下 8 个函数用于设定与获取外部数据的流 */
 bool set_input_io(int io, unsigned char *stream);
 bool set_output_io(int io, unsigned char *stream);
 bool set_input_io_size(int io, size_t size);
 bool set_output_io_size(int io, size_t size);
 unsigned char *get_input_io(int io);
 unsigned char *get_output_io(int io);
 size_t get_input_io_size(int io);
 size_t get_output_io_size(int io);

 /* 获取出错函数 */
 int error();

private:
  /* 重置整个虚拟机 */
  void reset();

  /* 执行 ins 指令 */
  bool call(unsigned ins);

  /* 以下 3 个函数都是与获取寄存器相关的 */
  bool invalid_register(int i);
  bool registers(int i, unsigned &v, bool four=false);
```

```
  bool set_registers(int i, unsigned r, bool four=false);

  /* 读取和写入数据到指定的内存地址 */
  bool read_memory(unsigned char *address, unsigned char *v);
  bool write_memory(unsigned char *address, unsigned v);

  /* 读取一条指令 */
  bool readi(unsigned &i);
  /* 判定偏移是否合法 */
  bool invalid_offset(unsigned off);
  /* 通过偏移计算地址 */
  bool calc_address(unsigned off, unsigned **addr);

  /* 与设置及获取字节序相关 */
  unsigned short get_te16(unsigned char *address);
  unsigned int get_te32(unsigned char *address);
  void set_te16(unsigned char *address, unsigned short v);
  void set_te32(unsigned char *address, unsigned v);

  /* 当虚拟机解释指令时，会将指令拆解成 8 种模式。指令再多，模式也只有这 8 种。这些函数会在指令执
行函数中调用*/
  bool handle_ins_mode_op(unsigned ins, unsigned &op);
  bool handle_ins_mode_op_imm(unsigned ins, unsigned &op, unsigned &imm);
  bool handle_ins_mode_op_reg(unsigned ins, unsigned &op, unsigned &reg);
  bool handle_ins_mode_op_reg_imm(unsigned ins, unsigned &op,unsigned &reg, \
  unsigned &imm);
  bool handle_ins_mode_op_reg_reg(unsigned ins, unsigned &op,unsigned &reg1, \
  unsigned &reg2);
  bool handle_ins_mode_op_mem_imm(unsigned ins, unsigned &op,unsigned **address,\
  unsigned &imm);
  bool handle_ins_mode_op_mem_reg(unsigned ins, unsigned &op,unsigned **address,\
  unsigned &reg);
  bool handle_ins_mode_op_mem_mem(unsigned ins, unsigned &op,unsigned **address1,\
  unsigned **address2);

  /* 将 4 字节的整数指令转换成对应的模式结构 */
  bool ins_2_opocde_mode(unsigned ins,
                    unsigned &opcode,
                    unsigned &mode);
  bool ins_2_mode_op(unsigned ins, pcvm_ins_mode_op &mode);
  bool ins_2_mode_op_imm(unsigned ins, pcvm_ins_mode_op_imm &mode);
  bool ins_2_mode_op_reg(unsigned ins, pcvm_ins_mode_op_reg &mode);
  bool ins_2_mode_op_reg_imm(unsigned ins, pcvm_ins_mode_op_reg_imm &mode);
  bool ins_2_mode_op_reg_reg(unsigned ins, pcvm_ins_mode_op_reg_reg &mode);
  bool ins_2_mode_op_mem_imm(unsigned ins, pcvm_ins_mode_op_mem_imm &mode);
  bool ins_2_mode_op_mem_reg(unsigned ins, pcvm_ins_mode_op_mem_reg &mode);
  bool ins_2_mode_op_mem_mem(unsigned ins, pcvm_ins_mode_op_mem_mem &mode);

  /* 指令执行 */
  bool iMOV(unsigned ins, unsigned mode);
```

```
  bool iPUSH(unsigned ins, unsigned mode);
  bool iPOP(unsigned ins, unsigned mode);
  ……

  /* 下面是支持调试器的辅助函数 */
#ifdef SUPPORT_DEBUGER
  void show_dbg_info();
  bool disasm(unsigned ins, std::string &out);
  void debugger(unsigned curr_ip);
#endif

private:
  /* 寄存器组 */
  unsigned _registers[PCVM_REG_NUMBER];
  /* 标志寄存器 */
  pcvm_flags_register _flags;
  /* 整个内存空间 */
  unsigned char *_space;
  /* 字节码在内存中的位置 */
  unsigned char *_code;
  /* 栈在内存中的位置 */
  unsigned char *_stack;
  /* 代码长度 */
  size_t _code_size;
  /* 栈长度 */
  size_t _stack_size;
  /* 指令执行函数队列 */
  ins_handle_fptr _handles[PCVM_OP_NUMBER];
  /* 输入与输出流设定 */
  unsigned char *_io_input[PCVM_IO_INPUT_NUMBER];
  unsigned char *_io_output[PCVM_IO_OUTPUT_NUMBER];
  size_t _io_input_size[PCVM_IO_INPUT_NUMBER];
  size_t _io_output_size[PCVM_IO_OUTPUT_NUMBER];

  /* 关机标志 */
  bool _shutdown;
  /* 字节序判定 */
  bool _is_big_endian;
  /* 出错代码 */
  int _error;
};
```

上面的代码注释比较清楚地描述了这个虚拟机的样子，后面会针对这个虚拟机的一些核心功能进行详细介绍。

2．I/O 设定

在上述代码中有如下函数。

```
/* 以下 8 个函数用于设定与获取外部数据的流 */
 bool set_input_io(int io, unsigned char *stream);
```

```
bool set_output_io(int io, unsigned char *stream);
bool set_input_io_size(int io, size_t size);
bool set_output_io_size(int io, size_t size);
unsigned char *get_input_io(int io);
unsigned char *get_output_io(int io);
size_t get_input_io_size(int io);
size_t get_output_io_size(int io);
```

这 8 个函数用于设定虚拟机对外数据的输入与输出。在运行虚拟代码之前，可以通过上述函数要输入或者输出的数据，使用 get*_io 函数获取输入或输出的内容，如果要获取其对应的长度，则使用 get*_io_size 函数。在后面的指令解析中，由一组中断指令“int 0,int 1,int 2,int 3”来完成以上设定。

3．寄存器说明

pcvm.h 文件定义了如下枚举体，用于描述虚拟机的寄存器设定。

```
enum PCVM_REG {
  PCVM_REG_IP,            /* 地址寄存器 */
  PCVM_REG_SB,            /* 栈基寄存器 */
  PCVM_REG_SP,            /* 栈指针寄存器 */
  PCVM_REG_RET,           /* 返回地址寄存器 */
  PCVM_REG_R4,            /* 通用寄存器 */
  PCVM_REG_R5,            /* 通用寄存器 */
  PCVM_REG_R6,            /* 通用寄存器 */
  PCVM_REG_R7,            /* 通用寄存器 */
  PCVM_REG_R8,            /* 通用寄存器 */
  PCVM_REG_R9,            /* 通用寄存器 */
  PCVM_REG_R10,           /* 通用寄存器 */
  PCVM_REG_R11,           /* 通用寄存器 */
  PCVM_REG_R12,           /* 通用寄存器 */
  PCVM_REG_R13,           /* 通用寄存器 */
  PCVM_REG_R14,           /* 通用寄存器 */
  PCVM_REG_R15,           /* 通用寄存器 */
  PCVM_REG_NUMBER
};
```

共有 16 个寄存器，前 4 个寄存器有特殊的功能，但算法变形引擎并不需要这么多寄存器。笔者起初直接设定了 256 个寄存器，后来发现实在没有必要这样做。

4．标志寄存器说明

pcvm.h 文件中定义了一个结构，代码如下。

```
typedef struct {
  unsigned int C : 1;
  unsigned int Z : 1;
  unsigned int S : 1;
  unsigned int O : 1;
  unsigned int A : 3;
  unsigned int reserve : 25;
```

```
} pcvm_flags_register;
```

其实在虚拟机的实际执行过程中没有全部使用这些标志，因为此虚拟机毕竟不是为了支持复杂的语言而设计的。在这个引擎中，这个标志位的设计有些画蛇添足了。我们在自己设计虚拟机时不一定要完全模拟现实中机器的特性设定。

标志寄存器是为了使程序顺利运行而做的设计，此外就是受真实机器的影响在编写程序时自然地编写了一个结构，如表 D.1 所示。

表 D.1　标志寄存器

标　志	意　义
C	运算结果是否产生进位
Z	运算结果是否为 0
S	为 1，则操作数值为有符号运算；为 0，则无符号运算
O	溢出位，但似乎没有使用
A	取值为 1、2、3，分别对应于 1 字节访问、2 字节访问、4 字节访问

5．内存的读取与写入

首先看两个函数。

- 读内存，示例如下。

```
bool pcvm::read_memory(unsigned char *address, unsigned char *v) {
  /* 访问寄存器的值如果为 1，则使用 1 字节读取 */
  if (_flags.A == 1) {
    *v = *address;
  }
  /* 访问寄存器的值如果为 2，则使用 2 字节读取 */
  else if (_flags.A == 2) {
    *(unsigned short*)v = get_te16(address);
  }
  /* 访问寄存器的值如果为 3，则使用 4 字节读取 */
  else if (_flags.A == 3) {
    *(unsigned int *)v = get_te32(address);
  }
  else {
    _error = PCVM_ERROR_INVALID_FLAG_A;
    return false;
  }
  return true;
}
```

- 写内存，示例如下。

```
bool pcvm::write_memory(unsigned char *address, unsigned v) {
  /* 访问寄存器的值如果为 1，则使用 1 字节写入 */
  if (_flags.A == 1) {
    *address = (unsigned char)(v & 0xFF);
  }
  /* 访问寄存器的值如果为 2，则使用 2 字节写入 */
  else if (_flags.A == 2) {
```

```
    set_te16(address, (unsigned short)(v & 0xFFFF));
  }
  /* 访问寄存器的值如果为 3，则使用 4 字节写入 */
  else if (_flags.A == 3) {
    set_te32(address, v);
  }
  else {
    _error = PCVM_ERROR_INVALID_FLAG_A;
    return false;
  }
  return true;
}
```

因为这是一个简单的虚拟机，并未涉及更多的指令，所以只是简单地使用了标志寄存器来控制内存的访问粒度。

6．内存地址计算

通过以下函数可知 pcvm 的地址管理并不是十分严谨，但是已经够用了。因为当指令访问内存时只能通过寄存器中保存的偏移来计算内存地址，例如“mov [r1], 5”，所以“[]”中只能是寄存器，不能是其他内容，代码如下。造成这种问题的原因是笔者想把指令放到 4 字节内而不想做更复杂的处理。

```
bool pcvm::calc_address(unsigned off, unsigned **addr)
{
  /* 检测偏移是否正确 */
  if (invalid_offset(off)) {
    _error = PCVM_ERROR_INVALID_ADDRESS;
    return false;
  }
  /* _space 就是虚拟机执行的内存空间的起始地址 */
  *addr = reinterpret_cast<unsigned*>(_space + off);
  return true;
}
```

7．OPCODE 编码

pcvm.h 文件中定义了一个枚举体，代码如下。

```
enum PCVM_OP {
  PCVM_OP_MOV,      /* 数据移动指令 */
  PCVM_OP_PUSH,     /* 压栈指令 */
  PCVM_OP_POP,      /* 弹栈指令 */
  PCVM_OP_CMP,      /* 对比指令 */
  PCVM_OP_CALL,     /* 调用函数指令 */
  PCVM_OP_RET,      /* 返回指令 */
  PCVM_OP_JMP,      /* 跳转指令 */
  PCVM_OP_JE,       /* 结果为 0 时的指令 */
  PCVM_OP_JNE,      /* 结果不为 0 时的指令 */
  PCVM_OP_JB,       /* 小于则跳转指令 */
  PCVM_OP_JA,       /* 大于则跳转指令 */
```

```
  PCVM_OP_JBE,      /* 小于等于则跳转指令 */
  PCVM_OP_JAE,      /* 大于等于则跳转指令 */
  PCVM_OP_AND,      /* 与指令 */
  PCVM_OP_OR,       /* 或指令 */
  PCVM_OP_NOT,      /* 非指令 */
  PCVM_OP_ADD,      /* 加法指令 */
  PCVM_OP_SUB,      /* 减法指令 */
  PCVM_OP_MUL,      /* 乘法指令 */
  PCVM_OP_DIV,      /* 除法指令 */
  PCVM_OP_MOD,      /* 取模指令 */
  PCVM_OP_SHL,      /* 左移指令 */
  PCVM_OP_SHR,      /* 右移指令 */
  PCVM_OP_INT,      /* 中断指令 */
  PCVM_OP_NOP,      /* 空指令 */
  PCVM_OP_NUMBER
};
```

这里列出了这个虚拟机的所有指令，只有 25 条，都是基本的流程控制运算指令。其中有一个 INT 指令，用于对不同的参数进行不同的操作。这些指令基本上可以实现变形引擎的基础需求。

D.2　指令编码详解

本节主要探讨指令的具体执行流程。首先让我们探讨一下 pcvm 指令编码的 8 种模式。

在 pcvm.h 文件中定义了如下枚举体用来解释指令的编码。

```
enum PCVM_INS_MODE {
  PCVM_INS_MODE_OP,              /* opcode */
  PCVM_INS_MODE_OP_IMM,          /* opcode imm */
  PCVM_INS_MODE_OP_REG,          /* opcode reg */
  PCVM_INS_MODE_OP_REG_IMM,      /* opcode reg, imm */
  PCVM_INS_MODE_OP_REG_REG,      /* opcode reg, reg */
  PCVM_INS_MODE_OP_MEM_IMM,      /* opcode mem, imm */
  PCVM_INS_MODE_OP_MEM_REG,      /* opcode mem, reg */
  PCVM_INS_MODE_OP_MEM_MEM,      /* opcode mem, mem */
  PCVM_INS_MODE_NUMBER
};
```

此虚拟机的一条指令固定为 4 字节、32 位。以上 8 种模式的字节编码如表 D.2 所示。

表 D.2　字节编码

模式名称	编　码	示　例
PCVM_INS_MODE_OP	5:opcode,3:mode,24:–	nop
PCVM_INS_MODE_OP_IMM	5:opcode,3:mode,4:–,20:imm	push 5
PCVM_INS_MODE_OP_REG	5:opcode,3:mode,4:reg,20:–	pop r6
PCVM_INS_MODE_OP_REG_IMM	5:opcode,3:mode,4:reg,20:imm	mov r4, 1
PCVM_INS_MODE_OP_REG_REG	5:opcode,3:mode,4:reg1,4:reg2,16:–	mov r4,r5
PCVM_INS_MODE_OP_MEM_IMM	5:opcode,3:mode,4:reg,20:imm	mov [r2], 3

PCVM_INS_MODE_OP_MEM_REG	5:opcode,3:mode,4:reg1,4:reg2,16:–	mov [r3], r2
PCVM_INS_MODE_OP_MEM_MEM	5:opcode,3:mode,4:reg1,4:reg2,16:–	mov [r3], [r2]

在 pcvm.h 文件中还定义了一组结构体用来记录解码后的信息，具体如下。

```
typedef struct {
  unsigned int opcode : 5;
  unsigned int mode : 3;
  unsigned int reserve : 24;
} pcvm_ins_mode_op;
typedef struct {
  unsigned int opcode : 5;
  unsigned int mode : 3;
  unsigned int reserve : 4;
  unsigned int imm : 20;
} pcvm_ins_mode_op_imm;
……
```

这里定义了一组函数来将一个 4 字节的整数转换成上述结构体，代码如下。

```
bool ins_2_opocde_mode(unsigned ins, unsigned &opcode, unsigned &mode);
bool ins_2_mode_op(unsigned ins, pcvm_ins_mode_op &mode);
bool ins_2_mode_op_imm(unsigned ins, pcvm_ins_mode_op_imm &mode);
……
```

因为每条指令的模式都存在两种字段，即 opcode 与 mode，所以后面的函数在执行后第一个调用的函数就是 ins_2_opocde_mode，由它来解析指令，代码如下。

```
bool pcvm::ins_2_opocde_mode(unsigned ins,
                         unsigned & opcode,
                         unsigned & mode) {
  /* 通过位移来获取拆解整数 */
  opcode = ins >> 27;
  mode = (ins >> 24) & 0x07;
  /* 验证 opcode 的合法性 */
  if (opcode >= PCVM_OP_NUMBER) {
    _error = PCVM_ERROR_INVALID_OPCODE;
    return false;
  }
  /* 验证 mode 的合法性 */
  if (mode >= PCVM_INS_MODE_NUMBER) {
    _error = PCVM_ERROR_INVALID_MODE;
    return false;
  }
  return true;
}
```

其他的解码函数都要先调用以上函数并填充解码结构。这里用 ins_2_mode_op_reg_imm 来讲解，其他函数都与之类似，代码如下。

```
bool pcvm::ins_2_mode_op_reg_imm(unsigned ins,pcvm_ins_mode_op_reg_imm & mode) {
  unsigned op = 0, mod = 0;
```

```
  /* 解析出 opcode 与 mode */
  if (ins_2_opocde_mode(ins, op, mod) == false) return false;
  /* 填充解码结构 */
  mode.opcode = op;
  mode.mode = mod;
  mode.reg = (ins >> 20) & 0x0F;
  mode.imm = ins & 0xFFFFF;
  return true;
}
```

D.2.1　解码处理函数

我们定义了 8 个函数，具体的指令执行函数就是执行这些函数来获取自己所需的信息。命名规范为“handle_ins_mode_xxx”。它们是一组外层函数，其内还是调用 ins_2_mode_xxx 来实现，这里就不一一说明了，示例如下。

```
bool handle_ins_mode_op(unsigned ins, unsigned &op);
bool handle_ins_mode_op_imm(unsigned ins, unsigned &op,
                        unsigned &imm);
.....
```

有几个与内存相关的函数，在调用 ins_2_mode_xxx 时进一步对内存地址进行核算。

D.2.2　算术指令

算术指令的处理模式一共有 5 种，具体如下。iNOT 是个特例，因为它只有“not reg”一种模式，后面会对它进行详细分析。

- 算术指令“reg, imm”。
- 算术指令“reg, reg”。
- 算术指令“mem, imm”。
- 算术指令“mem, reg”。
- 算术指令“mem, mem”。

算术指令有 iAND、iOR、iNOT、iADD、iSUB、iMUL、iDIV、iMOD、iSHL、iSHR。以 iADD 为例进行解释，示例如下。

```
bool pcvm::iADD(unsigned ins, unsigned mode) {
  unsigned v1 = 0, v2 = 0;
  unsigned opcode = 0;
  /* 判断模式是否是 add reg, imm */
  if (mode == PCVM_INS_MODE_OP_REG_IMM) {
    unsigned reg = 0, imm = 0;
    /* 从中取出 reg 与 imm 的具体值 */
    if (handle_ins_mode_op_reg_imm(ins, opcode, reg, imm) == false)
      return false;
    /* 将 reg 中的值读取出来，写入 v1 */
    if (registers(reg, v1) == false) return false;
    /* 将 imm 写入 v2 */
    if (write_memory(reinterpret_cast<unsigned char*>(&v2), imm) == false)
      return false;
```

```
  /* 如果 S 标志被设置，那么作为有符号数计算 */
  if (_flags.S) {
    int v1_ = (int)v1;
    int v2_ = (int)v2;
    v1_ += v2_;
    v1 = (unsigned)v1_;
  }
  /* 作为无符号数直接计算 */
  else {
    v1 += v2;
  }
  /* 将结果设置到 reg 处 */
  if (set_registers(reg, v1) == false) return false;
}
/* 处理 add reg, reg 模式 */
else if (mode == PCVM_INS_MODE_OP_REG_REG) {
  unsigned reg1 = 0, reg2 = 0;
  /* 从 ins 中取出 opcode 及两个寄存器的索引 */
  if (handle_ins_mode_op_reg_reg(ins, opcode, reg1, reg2) == false)
    return false;
  /* 读取两个寄存器的值 */
  if (registers(reg1, v1) == false) return false;
  if (registers(reg2, v2) == false) return false;

  /* 查看符号标志位，如果是有符号位，则进行类型转换 */
  if (_flags.S) {
    int v1_ = (int)v1;
    int v2_ = (int)v2;
    v1_ += v2_;
    v1 = (unsigned)v1_;
  }
  else {
    v1 += v2;
  }

  /* 设置寄存器的值 */
  if (set_registers(reg1, v1) == false) return false;
}
/* 处理 add [mem], imm 模式 */
else if (mode == PCVM_INS_MODE_OP_MEM_IMM) {
  unsigned *address = nullptr, imm = 0;
  /* 取出 opcode、要访问的地址及立即数 */
  if (handle_ins_mode_op_mem_imm(ins, opcode, &address, imm) == false)
    return false;
  /* 取出立即数并写入 v2 临时变量 */
  if (write_memory(reinterpret_cast<unsigned char*>(&v2), imm) == false)
    return false;
  /* 从 address 的地址中取出数值 */
  if (write_memory(reinterpret_cast<unsigned char*>(&v1), *address) == false)
```

```
    return false;
  /* 按照符号进行处理 */
  if (_flags.S) {
    int v1_ = (int)v1;
    int v2_ = (int)v2;
    v1_ += v2_;
    v1 = (unsigned)v1_;
  }
  else {
    v1 += v2;
  }

  /* 最终把结果写入 address 指向的地址 */
  if (write_memory(reinterpret_cast<unsigned char*>(address), v1) == false)
    return false;
}
/* 处理 add [mem], reg 模式 */
else if (mode == PCVM_INS_MODE_OP_MEM_REG) {
  unsigned *address = nullptr, reg = 0;
  /* 从 ins 中取出 opcode、address 及要操作的寄存器 */
  if (handle_ins_mode_op_mem_reg(ins, opcode, &address, reg) == false)
    return false;
  /* 从 address 指向的内存中取出数值 */
  if (write_memory(reinterpret_cast<unsigned char*>(&v1), *address) == false)
    return false;
  /* 从寄存器中读取值 */
  if (registers(reg, v2) == false)
    return false;

  /* 按照符号寄存器进行运算 */
  if (_flags.S) {
    int v1_ = (int)v1;
    int v2_ = (int)v2;
    v1_ += v2_;
    v1 = (unsigned)v1_;
  }
  else {
    v1 += v2;
  }

  /* 将结果写入 address 指向的内存 */
  if (write_memory(reinterpret_cast<unsigned char*>(address), v1) == false)
    return false;
}
/* 处理 add [mem1], [mem2]模式 */
else if (mode == PCVM_INS_MODE_OP_MEM_MEM) {
  unsigned *address1 = nullptr, *address2 = nullptr;
  /* 从指令中取出两个要操作的地址 */
  if (handle_ins_mode_op_mem_mem(ins, opcode, &address1, &address2) == false)
    return false;
```

```
    /* 从两个地址中取出数值 */
    if (write_memory(reinterpret_cast<unsigned char*>(&v1), *address1) == false)
      return false;
    if (write_memory(reinterpret_cast<unsigned char*>(&v2), *address2) == false)
      return false;

    /* 按照符号寄存器进行计算 */
    if (_flags.S) {
      int v1_ = (int)v1;
      int v2_ = (int)v2;
      v1_ += v2_;
      v1 = (unsigned)v1_;
    }
    else {
      v1 += v2;
    }

    /* 将结果写入 address1 指向的内存 */
    if (write_memory(reinterpret_cast<unsigned char*>(address1), v1) == false)
      return false;
  }
  else {
    _error = PCVM_ERROR_INVALID_MODE;
    return false;
  }
  return true;
}
```

基本算术指令都是按照这个流程来运行的，只是具体运算不同而已。唯一不同的是 NOT 指令，它只有一种模式，就是对寄存器中的值进行非运算，代码如下。

```
bool pcvm::iNOT(unsigned ins, unsigned mode)
{
  unsigned opcode = 0;
  /* 处理 not reg */
  if (mode == PCVM_INS_MODE_OP_REG) {
    unsigned reg = 0, value = 0;
    /* 从 ins 中取出 opcode 及 reg */
    if (handle_ins_mode_op_reg(ins, opcode, reg) == false)
      return false;

    /* 进行 NOT 操作 */
    if (registers(reg, value) == false) return false;
    if (set_registers(reg, ~value) == false) return false;
  }
  else {
    _error = PCVM_ERROR_INVALID_MODE;
    return false;
  }
  return true;
}
```

D.2.3　数据转移指令

数据转移指令有 MOV、PUSH、POP。首先看一下著名的 MOV 指令，代码如下。

```
bool pcvm::iMOV(unsigned ins, unsigned mode) {
  unsigned opcode = 0;
  /* 处理 mov reg, imm */
  if (mode == PCVM_INS_MODE_OP_REG_IMM) {
    unsigned reg = 0, imm = 0;
    if (handle_ins_mode_op_reg_imm(ins, opcode, reg, imm) == false)
      return false;
    /* 直接设置立即数到指定的寄存器 */
    return set_registers(reg, imm);
  }
  /* mov reg,reg */
  else if (mode == PCVM_INS_MODE_OP_REG_REG) {
    unsigned reg1 = 0, reg2 = 0;
    if (handle_ins_mode_op_reg_reg(ins, opcode, reg1, reg2) == false)
      return false;
    unsigned v = 0;
    /* 从源寄存器中读取设置到目的寄存器中 */
    if (registers(reg2, v) == false) return false;
    return set_registers(reg1, v);
  }
/* 剩下的就是其他处理 */
}
```

压栈操作只能处理两种模式，分别是“push imm”和“push reg”，示例如下。

```
bool pcvm::iPUSH(unsigned ins, unsigned mode) {
  unsigned opcode = 0;
  /* 压栈，先读取当前栈寄存器的值，然后减去一个 4 字节（32 位系统） */
  unsigned offset = _registers[PCVM_REG_SP] - sizeof(unsigned);
  /* push imm */
  if (mode == PCVM_INS_MODE_OP_IMM) {
    unsigned imm = 0;
    if (handle_ins_mode_op_imm(ins, opcode, imm) == false)
      return false;

    /* 校验偏移是否有效 */
    if (invalid_offset(offset)) {
      _error = PCVM_ERROR_INVALID_ADDRESS;
      return false;
    }

    /* 通过 offset 计算虚拟内存的地址 */
    unsigned *address = nullptr;
    if (calc_address(offset, &address) == false) return false;
    /* 写入内存 */
    if (write_memory(reinterpret_cast<unsigned char*>(address),    imm) == false)
      return false;
  }
```

```
  /* push reg */
  else if (mode == PCVM_INS_MODE_OP_REG) {
    /* 与之前大同小异，故省略 */
  }
  else {
    _error = PCVM_ERROR_INVALID_MODE;
    return false;
  }
  /* 重新设置栈寄存器的指针 */
  _registers[PCVM_REG_SP] -= sizeof(unsigned);
  return true;
}
```

弹栈操作只能处理一种模式，即“pop reg”，示例如下。

```
bool pcvm::iPOP(unsigned ins, unsigned mode) {
  unsigned opcode = 0;
  /* pop reg */
  if (mode == PCVM_INS_MODE_OP_REG) {
    unsigned reg = 0;
    if (handle_ins_mode_op_reg(ins, opcode, reg) == false)
      return false;
    /* 取出当前栈指向的偏移并转换成地址 */
    unsigned offset = _registers[PCVM_REG_SP];
    if (invalid_offset(offset)) {
      _error = PCVM_ERROR_INVALID_ADDRESS;
      return false;
    }
    unsigned *address = nullptr;
    if (calc_address(offset, &address) == false) return false;
    /* 从栈地址中读取值并设置给寄存器 */
    if (set_registers(reg, *address) == false) return false;
  }
  else {
    _error = PCVM_ERROR_INVALID_MODE;
    return false;
  }

  /* 最后移动栈指针 */
  _registers[PCVM_REG_SP] += sizeof(unsigned);
  return true;
}
```

D.2.4 流程控制指令

流程指令有 CALL、RET、JMP、JE、JNE、JB、JA、JBE、JAE。除了跳转指令，可以使用 cmp 指令来判定流程的走向。

1．cmp 指令

cmp 指令可以处理的模式有“cmp reg”、“imm”、“cmp reg”、“reg”、“cmp mem”、“imm”、

“cmp mem”、“reg”、“cmp mem”、“mem”，示例如下。

```
bool pcvm::iCMP(unsigned ins, unsigned mode) {
  /* 以下代码忽略，其意思就是从各种模式中取出数据 */
  /* 最后的值相减 */
  int res = static_cast<int>(v1) - static_cast<int>(v2);
  /* 设置一些标志位 */
  if (res == 0) _flags.Z = 1; else _flags.Z = 0;
  if (res < 0) _flags.C = 1; else _flags.C = 0;

  return true;
}
```

2．各种跳转指令

这里只列出一些有代表意义的跳转指令，例如 call、ret、jmp。其他的判定跳转只是在 jmp 的基础上增加了标志位判定而已。

(1) call 指令

这个指令只能处理两种模式，分别是“call imm”与“call reg”，代码分析如下。

```
bool pcvm::iCALL(unsigned ins, unsigned mode) {
  /* 得到返回地址栈的偏移 */
  unsigned offset = _registers[PCVM_REG_SP] - sizeof(unsigned);
  if (invalid_offset(offset)) {
    _error = PCVM_ERROR_INVALID_ADDRESS;
    return false;
  }
  /* 通过偏移得到地址 */
  unsigned *address = nullptr;
  if (calc_address(offset, &address) == false) return false;
  /* 将当前的地址，也就是 call 指令的后一条指令的地址，写入返回地址栈 */
  unsigned next_address = _registers[PCVM_REG_IP];
  if (write_memory(reinterpret_cast<unsigned char*>(address), next_address) == false)
    return false;
  unsigned opcode = 0, jmpto = 0;
  /* call imm 模式 */
  if (mode == PCVM_INS_MODE_OP_IMM) {
    unsigned imm = 0;
    if (handle_ins_mode_op_imm(ins, opcode, imm) == false)
      return false;
    jmpto = imm;
  }
  /* call reg 模式 */
  else if (mode == PCVM_INS_MODE_OP_REG) {
    unsigned reg = 0, imm;
    if (handle_ins_mode_op_reg(ins, opcode, reg) == false)
      return false;
    /* 从寄存器中读取偏移 */
    if (registers(reg, imm, true) == false) return false;
```

```
    jmpto = imm & 0xFFFFF;
  }
  else {
    _error = PCVM_ERROR_INVALID_MODE;
    return false;
  }

  /* 设定栈与调用地址 */
  _registers[PCVM_REG_IP] = jmpto;
  _registers[PCVM_REG_SP] = offset;

  return true;
}
```

（2）ret 指令

返回指令也很简单，从栈中取出返回地址，设定 IP 寄存器即可，示例如下。

```
bool pcvm::iRET(unsigned ins, unsigned mode) {
  /* 把偏移从栈中弹出 */
  unsigned offset = _registers[PCVM_REG_SP];
  if (invalid_offset(offset)) {
    _error = PCVM_ERROR_INVALID_ADDRESS;
    return false;
  }
  unsigned *address = nullptr;
  /* 通过偏移计算地址 */
  if (calc_address(offset, &address) == false) return false;

  /* 取出最后的返回地址 */
  _registers[PCVM_REG_IP] = *address;

  /* 将栈恢复 */
  _registers[PCVM_REG_SP] = offset + sizeof(unsigned);

  return true;
}
```

（3）jmp 指令

该指令处理两种模式，分别是“jmp imm”与“jmp reg”。其中寄存器与立即数保存的都是直接地址，取出后直接设置给 IP 寄存器即可。这里之所以没有遵循现实计算机的偏移原则，是因为笔者觉得实在没有必要弄得那么复杂，而虚拟机的内存已经限定在 1MB 以内，所以直接使用地址即可，示例如下。

```
bool pcvm::iJMP(unsigned ins, unsigned mode)
{
  unsigned opcode = 0, address = 0;
  /* jmp imm */
  if (mode == PCVM_INS_MODE_OP_IMM) {
```

```
    unsigned imm = 0;
    if (handle_ins_mode_op_imm(ins, opcode, imm) == false)
      return false;
    address = imm;
  }
  /* jmp reg */
  else if (mode == PCVM_INS_MODE_OP_REG) {
    unsigned reg = 0, imm;
    if (handle_ins_mode_op_reg(ins, opcode, reg) == false)
      return false;
    if (registers(reg, imm, true) == false) return false;
    address = imm & 0xFFFFF;
  }
  else {
    _error = PCVM_ERROR_INVALID_MODE;
    return false;
  }
  /* 设定跳转地址 */
  _registers[PCVM_REG_IP] = address;
  return true;
}
```

其他条件跳转指令只不过是增加了标志位判定而已，示例如下。

```
bool pcvm::iJE(unsigned ins, unsigned mode)
{

  /* 判定标志位是否为 0 */
  if (_flags.Z == 0) {
    return true;
  }
  return iJMP(ins, mode);
}
```

D.2.5　中断指令

各个中断的说明如表 D.3 所示。其实所谓“中断”，就是借用这样一个名字而已。这种设计减少了指令的个数，而且可以方便地扩展功能，示例代码如下。

表 D.3　对各个中断的说明

中 断 号	说　　明
0	从 R4 寄存器中取出输入端口号；从 R5 寄存器中取出要读取的数据的长度，然后将数据写入当前栈指针所指向的内存，最后将长度写入 R4 寄存器
1	从 R4 寄存器中取出输出端口号；从 R5 寄存器中取出要写入的数据的长度，然后将当前栈指针所指向的内存的数据写入端口的缓存，最后将长度写入 R4 寄存器
2	从 R4 寄存器中读取出输入端口；将指定的端口中存在的数据的长度放入 R5 寄存器
3	从 R4 寄存器中读取出输出端口，将要写入的数据的长度放入指定输出端口保存长度中。上层可以通过 get_output_io_size(port) 函数获取这个长度

续表

中 断 号	说　明
4	读取 R4 寄存器的值（只可以是 1、2、4，1 表示 1 字节，2 表示 2 字节，4 表示 4 字节），用来控制对数据的操作粒度
5	R4 大于 0，表示以后采用无符号计算；R4 小于等于 0，则采用有符号计算
9	关闭虚拟机

```
bool pcvm::iINT(unsigned ins, unsigned mode)
{
  unsigned opcode = 0;
  if (mode == PCVM_INS_MODE_OP_IMM) {
    unsigned imm = 0;
    if (handle_ins_mode_op_imm(ins, opcode, imm) == false)
      return false;

    /* int 0
     * 从 R4 寄存器中取出输入端口号
     * 从 R5 寄存器中取出要读取数据的长度，然后将数据写入当前栈指针所指向的内存，
     * 最后将长度写入 R4 寄存器
     */
    if (imm == 0) {
      unsigned port = 0, size = 0;
      if (registers(PCVM_REG_R4, port) == false) return false;
      if (port >= PCVM_IO_INPUT_NUMBER) {
        _error = PCVM_ERROR_INVALID_IO_ACCESS;
        return false;
      }

      if (registers(PCVM_REG_R5, size) == false) return false;
      /* fixme: check size */

      unsigned offset = _registers[PCVM_REG_SP];
      if (invalid_offset(offset)) {
        _error = PCVM_ERROR_INVALID_ADDRESS;
        return false;
      }
      unsigned *address = nullptr;
      if (calc_address(offset, &address) == false) return false;

      if (_io_input[port] == nullptr) {
        _error = PCVM_ERROR_IO_NOT_BOUNAD;
        return false;
      }
      /* 写入数据 */
      memcpy(address, _io_input[port], size);

      if (set_registers(PCVM_REG_R4, size) == false) return false;
    }
    /* int 1
```

```
 * 从 R4 寄存器中取出输出端口号
 * 从 R5 寄存器中取出要写入数据的长度，然后将当前栈指针所指向的内存的数据写入端口的缓存中，
 *最后将长度写入 R4 寄存器
 */
else if (imm == 1) {
  unsigned port = 0, size = 0;
  if (registers(PCVM_REG_R4, port) == false) return false;
  if (port >= PCVM_IO_INPUT_NUMBER) {
    _error = PCVM_ERROR_INVALID_IO_ACCESS;
    return false;
  }

  if (registers(PCVM_REG_R5, size) == false) return false;
  /* fixme: check size */

  unsigned offset = _registers[PCVM_REG_SP];
  if (invalid_offset(offset)) {
    _error = PCVM_ERROR_INVALID_ADDRESS;
    return false;
  }
  unsigned *address = nullptr;
  if (calc_address(offset, &address) == false) return false;

  if (_io_output[port] == nullptr) {
    _error = PCVM_ERROR_IO_NOT_BOUNAD;
    return false;
  }
  memcpy(_io_output[port], address, size);

  if (set_registers(PCVM_REG_R4, size) == false) return false;
}
/* int 2
 * 从 R4 寄存器中读取输入端口
 * 将指定的端口中存在的数据个数放入 R5 寄存器
 */
else if (imm == 2) {
  unsigned port = 0;
  if (registers(PCVM_REG_R4, port) == false) return false;
    if (port >= PCVM_IO_INPUT_NUMBER) {
      _error = PCVM_ERROR_INVALID_INT_PARAM;
      return false;
    }
  set_registers(PCVM_REG_R5, _io_input_size[port]);
}
/* int 3
 * 从 R4 寄存器中读取输出端口
 * 将要写入的数据的长度放入指定输出端口保存的长度中
 * 上层可以通过 get_output_io_size(port)来获取这个长度
 */
else if (imm == 3) {
  unsigned port = 0, size = 0;
```

```
      if (registers(PCVM_REG_R4, port) == false) return false;
      if (port >= PCVM_IO_OUTPUT_NUMBER) {
        _error = PCVM_ERROR_INVALID_INT_PARAM;
        return false;
      }
      if (registers(PCVM_REG_R5, size) == false) return false;
      _io_output_size[port] = size;
    }
    /* int 4
     * 读取 R4 寄存器的值只可以是 1、2、4，用于控制对数据的操作粒度
     * 1 表示 1 字节，2 表示 2 字节，4 表示 4 字节
     */
    else if (imm == 4) {
      unsigned unit = 0;
      if (registers(PCVM_REG_R4, unit) == false) return false;
      if ((unit != 1) || (unit != 2) || (unit != 4)) {
        _error = PCVM_ERROR_INVALID_INT_PARAM;
        return false;
      }
      _flags.A = unit;
    }
    /* int 5
     * R4 大于 0 表示以后采用无符号计算
     * R4 小于等于 0 则采用有符号计算
     */
    else if (imm == 5) {
      unsigned sign = 0;
      if (registers(PCVM_REG_R4, sign) == false) return false;
      _flags.S = !(sign > 0);
    }
    /* int 9
     * 直接关机
     */
    else if (imm == 9) {
      _shutdown = true;
    }
    else {
      _error = PCVM_ERROR_INVALID_INT_NUMBER;
      return false;
    }
  }
  else {
    _error = PCVM_ERROR_INVALID_MODE;
    return false;
  }
  return true;
}
```

D.2.6 虚拟机调试器

如果 SUPPORT_DEBUGER 宏开始运行，则虚拟机的整个代码会将调试器模块编译到其中。

其主要入口函数为 debugger，并在 run 函数的循环中调用，示例如下。

```
while (_shutdown == false) {
  /* 如果调试模式开启，则在执行每一条指令前进入调试器 */
#ifdef SUPPORT_DEBUGER
  if (debug) {
    debugger(_registers[PCVM_REG_IP]);
  }
#endif
  if (readi(ins) == false) {
    if (_error == PCVM_ERROR_OVER_CODE_SIZE_LIMIT) {
      _error = PCVM_ERROR_SUCCESS;
      return true;
    }
    return false;
  }
  if (call(ins) == false) return false;
}
return true;
```

每执行一次会弹出一个如图 D.2 所示的简单界面。

调试指令只有如下 3 条。

- c：继续运。
- q：退出。
- h：帮助。

其实，有简单的排错功能就够了。对此感兴趣的读者直接阅读代码即可，其实现很简单。

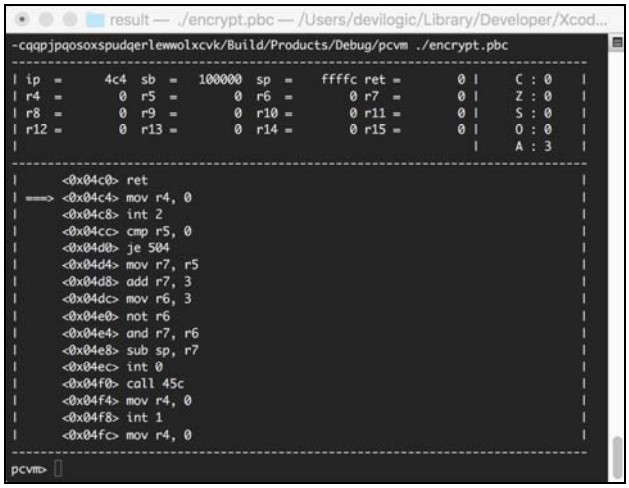


图 D.2　虚拟机调试器

D.3　虚拟机汇编器的设计与实现

现在进入汇编器的代码分析。本节给出的这个汇编器比较简单（不过，汇编器大都不复杂）。因为笔者没有设计宏的概念，所以不涉及语法的处理。这个汇编器就是使用词法处理将字符串

转换成标记，按照每条指令可以处理的模式进行汇编。其中稍微麻烦一点的是需要将标签记录下来，供汇编完成后的重定位处理使用。最后定义了一个很简单的文件格式并将代码写入。汇编器只有 3 个文件，分别是 pcasm.h、pcasm.cpp、pcasm_main.cpp。

D.3.1 汇编器结构介绍

在 pcasm.h 文件中可以看到完整的汇编器所需结构的定义。

标签定义有两项，address 是全局文件的地址，mode_address 是当前模块的地址，示例如下。

```
typedef struct {
   unsigned address;
   unsigned mode_address;
} pcasm_label;
```

重定位项的类型只有一个，就是基地址重定位，示例如下，在 D.3.2 节中会详细说明。

```
enum {
   PCASM_REL_FIXLST20B,
   PCASM_REL_NUMBER
};
```

重定位项结构示例如下。type 其实只有一个值，是由上面的枚举体定义的；address 是当前要重定位的地址；symbol 是符号字符串，用于表明要重定位的是哪个标签。可以通过符号字符串到符号表中进行查询，从而确定符号的地址。

```
typedef struct {
   int type;
   unsigned address;
   std::string symbol;
} pcasm_relocate;
```

词法分析中所用到的标记结构示例如下。token 表示标记的整型定义；text 是标记的字面值结构，用于保存这个结构当前的值，例如一个字符串、一个整数、一个寄存器等，在后面的汇编中将会使用它。

```
typedef struct {
   int token;
   pcasm_text text;
} pcasm_token;

enum {
   TOKEN_OP,                    /* OPCODE */
   TOKEN_REG,                   /* 寄存器 */
   TOKEN_IMM,                   /* 立即数 */
   TOKEN_DEF_LABEL,             /* 定义标记 */
   TOKEN_REF_LABEL,             /* 引用标记 */
   TOKEN_COLON,                 /* 冒号 */
   TOKEN_LPARAM,                /* 左括号 */
   TOKEN_RPARAM,                /* 右括号 */
   TOKEN_COMMA,                 /* 逗号 */
```

```
    TOKEN_INCLUDE,                /* include 文件包含 */
    TOKEN_STRING,                 /* 字符串 */
    TOKEN_EOF,                    /* 到达末尾 */
    TOKEN_NUMBER                  /* 标记数量 */
};
```

在这里需要了解一点编译原理。其实往深里讲，做软件保护壳和做编译器没有区别，如果想做精品，需要按照如图 D.3 所示的流程工作。把这个流程扩展，就可以完成一个二进制编译器。

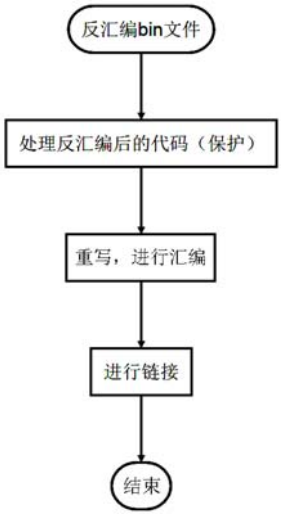


图 19.3　编译器的工作流程

标记字面值是 token 的子结构，根据 token 的不同提供不同的值为对应，通过如下字段的名称就可以看出它们对应于哪些标记了。

```
typedef struct {
    unsigned opcode;
    std::string str;
    unsigned reg;
    unsigned imm;
    std::vector<unsigned char> data;
} pcasm_text;
```

汇编器的主类定义如下。这里因为篇幅所限，将一些不重要的及重复性过高的函数去掉了。

```
class pcasm {
public:
    pcasm();
    virtual ~pcasm();
    /* 主流程入口 */
    bool compile(const std::string &asm_file, std::vector<unsigned char> \
    &bytecodes, bool is_file=true);
    /* 汇编并做重定位处理 */
    bool make(const std::string &asm_file, std::vector<unsigned char> &bytecodes);
    /* 将代码写入 pc 文件格式中 */
    bool make_pcfile(unsigned char *bytecodes, size_t bcsize,\
    unsigned char **pcfile, size_t &pcfile_size);
    /* 获取当前出错代码 */
    int error();

public:
    /* 初始化 */
    static void init();
    /* 链接操作，其实就是负责重定位 */
    static int pclink(std::vector<unsigned char> &bytecodes);
    /* 获得汇编后代码的入口偏移 */
    static unsigned entry();

private:
    /* 重新设定此类的所有数据 */
```

```
    void reset();

    /* 汇编成一条指令函数集合 */
    unsigned write_mode_op(unsigned opcode);
    unsigned write_mode_op_imm(unsigned opcode, unsigned imm);
    /* 省略 */

    /* 具体指令编译函数 */
    bool MOV_REG_LAB(unsigned opcode, std::vector<unsigned char> &bytecodes);
    bool MOV_MEM_LAB(unsigned opcode, std::vector<unsigned char> &bytecodes);
    /* 省略 */
    bool cMOV(std::vector<unsigned char> &bytecodes);

    /* 省略其他指令*/

    /* 处理 include 文件的标记 */
    bool cInclude(std::vector<unsigned char> &bytecodes);
    /* 定义一个标记 */
    bool cDefLabel(pcasm_token &token);

    /* 第 1 遍处理 */
    bool pass1();
    /* 第 2 遍处理 */
    bool pass2(std::vector<unsigned char> &bytecodes);

    /* 进行词法扫描 */
    bool scanner(pcasm_token &token);
    /* 进行汇编的主入口 */
    bool parser(std::vector<unsigned char> &bytecodes);
    /* 设置出错代码 */
    void set_error(int err);
    /* 判断当前一组标记是否匹配 */
    bool match(std::vector<int> tokens);

    /* 增加地址 */
    bool plus_address(unsigned plus=sizeof(unsigned));

private:
    /* 操作当前的字符串流 */
    bool teste();
    int readc();
    bool plusp(int plus=1);
    bool decp(int dec=1);

    /* 汇编操作集合 */
    int write_ins(int ins, std::vector<unsigned char> &bytecodes);
    int write_datas(const std::vector<unsigned char> &datas, \
    std::vector<unsigned char> &bytecodes);
    int write_imm(unsigned data, std::vector<unsigned char> &bytecodes);
    int write_string(const std::string &str, std::vector<unsigned char>\
```

```
    &bytecodes);

    /* 对词法标记的操作集合 */
    pcasm_token next_token();
    void rollback_token(int num = 1);

    /* 省略 */
};
```

让我们从 compile 函数开始，示例如下。

```
bool pcasm::compile(const std::string & asm_file, \
    std::vector<unsigned char> &bytecodes, bool is_file) {
    reset();
    /* 如果是文件，则读取文件 */
    if (is_file) {
        std::string source = s_read_file(asm_file);
        if (source == "") return false;
        _source = source;
    }
    else {
        /* 直接使用字符串作为源代码 */
        _source = asm_file;
    }
    /* 第 1 遍处理 */
    if (pass1() == false) return false;
    /* 第 2 遍处理 */
    if (pass2(bytecodes) == false) return false;

    return true;
}
```

make 与 make_pcfile 只是对上面这个函数进行了进一步的封装而已。下面我们就来看一下这两遍处理。

D.3.2　第 1 遍处理

第 1 遍处理的代码如下。

```
bool pcasm::pass1()
{
    pcasm_token token;
    _token_source.clear();
    /* 循环扫描源代码文本并将标记填充到_token_source 向量中 */
    do {
        if (scanner(token) == false) {
            return false;
        }
        _token_source.push_back(token);
    } while (token.token != TOKEN_EOF);
    return true;
}
```

第 1 遍处理的功能很明确，就是填充一个由标记组成的队列，scanner 则用于返回词法标记，部分代码如下。

```
bool pcasm::scanner(pcasm_token &token)
{
    int c = 0;
    std::string str;
    /* 测试是否到达源代码末尾 */
    while (teste() == false) {
        /* 读取 1 个字符 */
        c = readc();
        /* 跳过空白符 */
        if ((c == ' ') || (c == '\t') || (c == '\r') || (c == '\n')) {
            continue;
        }
        /* 处理注释 */
        else if (c == ';') {
            do {
                c = readc();
                if (c == -1) {
                    token.token = TOKEN_EOF;
                    return true;
                }
            } while (c != '\n');
            continue;
        }

        switch (c) {
        /* 十六进制数的读取 */
        case 'x':
            str.clear();
            do {
                c = readc();
                if (s_is_hexchar(c) == false) {
                    decp();
                    break;
                }
                str.push_back(c);
            } while (true);

            if (str.empty() == false) {
                char *hexptr = nullptr;
                token.text.imm = static_cast<unsigned>(strtol(str.c_str(), &hexptr, 16));
                token.token = TOKEN_IMM;
                return true;
            }
            else {
                _error = PCASM_ERROR_SCAN_INVALID_CHAR;
                return false;
            }
            break;
```

```
        /* 省略中间部分 */

        /* 关于标签的操作，这个是最复杂的一个了 */
        case '@':
            token.text.str.clear();
            /* 读取 1 个字符 */
            c = readc();
            if (c == -1) {
                _error = PCASM_ERROR_SCAN_QUOTATION_NOT_CLOSE;
                return false;
            }
            /* 如果当前字符是字母或者下画线，则继续读取
             * 标签只允许为字母、数字、下画线，并且只能以下画线和字母开头
             */
            if ((isalpha(c)) || (c == '_')) {
                int len = 1;
                while ((c == '_') || isalnum(c)) {
                    if (len >= PCASM_MAX_LABEL) {
                        _error =
                        PCASM_ERROR_SCAN_LABNAME_OVER_LIMIT;
                        _errstr = token.text.str;
                        return false;
                    }
                    token.text.str.push_back(c);
                    c = readc();
                    len++;
                }
                /* 如果标号名后接一个冒号，则视为标号的定义 */
                if (c == ':') {
                    token.token = TOKEN_DEF_LABEL;
                    return true;
                }
                /* 如果仅是标号名，则视为引用标号 */
                else {
                    token.token = TOKEN_REF_LABEL;
                    return true;
                }
            }
            break;
        default:
            _error = PCASM_ERROR_SCAN_NOT_MATCH_TOKEN_START_CHAR;
            return false;
        }
    }

    token.token = TOKEN_EOF;
    return true;
}
```

D.3.3 第 2 遍处理

pass2 函数只是一个外包装而已，所以我们忽略它。pass2 函数里面调用了 parser 函数。parser 函数主要从 pass1 函数压入的标记队列中依次取出标记，并按照既定的规范开始进行汇编，示例如下。

```
bool pcasm::parser(std::vector<unsigned char> &bytecodes)
{
    pcasm_token token;
    do {
        /* 取得下一个标记 */
        token = next_token();
        /* 遇到末尾则退出 */
        if (token.token == TOKEN_EOF) {
            break;
        }
        /* 遇到指令标记 */
        else if (token.token == TOKEN_OP) {
            /* 执行对应的指令汇编函数
             * _handles 是一个私有变量，保存了各条指令的汇编函数指针
             * 直接从标记字面值中获取索引进行寻址
             */
            if ((this->*_handles[token.text.opcode])(bytecodes) == false)
        return false;
            /* 汇编一条指令，则增加地址 */
            if (plus_address() == false) return false;
        }
        /* 遇到 include 标记，则调用 cInclude 函数 */
        else if (token.token == TOKEN_INCLUDE) {
            if (cInclude(bytecodes) == false)
        return false;
        }
        /* 遇到标记定义 */
        else if (token.token == TOKEN_DEF_LABEL) {
            if (cDefLabel(token) == false)
        return false;
        }
        /* 遇到立即数 */
        else if (token.token == TOKEN_IMM) {
            if (plus_address(write_imm(token.text.imm, bytecodes)) == false)
        return false;
        }
        /* 遇到字符串 */
        else if (token.token == TOKEN_STRING) {
            if (plus_address(write_string(token.text.str, bytecodes)) == false)
        return false;
        }
        else {
            _error = PCASM_ERROR_SYNTAX_INCONFORMITY_TOKEN;
            return false;
```

```
        }
    } while (true);
    return true;
}
```

1．指令地址

这里设计了两个地址：_address 其实是模块内的地址，就是一个单独的文件拥有的地址；s_address 是全局地址，是当链接成一套完整的字节码程序时的真实地址。笔者本想将 _address 作为唯一地址，然后在链接阶段重定位并合成最后的地址（这样做更符合一个真实的链接器的工作过程），但由于时间有限，就“偷工减料”了，代码如下。

```
bool pcasm::plus_address(unsigned plus)
{
    /* 任意单一模块不能超过 1MB */
    if (s_address >= (1024 * 1024)) {
        _error = PCASM_ERROR_CODE_OVER_LIMIT;
        return false;
}

    _address += plus;          /* 全局地址增加 */
    s_address += plus;
    return true;
}
```

2．指令汇编

在汇编器类中定义一个变量“ins_compile_fptr _handles[PCVM_OP_NUMBER];”。它是一个函数指针队列，其中保存了每条指令的汇编函数。这个变量在 pcasm::pcasm 的构造函数中进行填充，示例如下。

```
pcasm::pcasm()
{
    reset();
    _handles[PCVM_OP_MOV] = &pcasm::cMOV;
    _handles[PCVM_OP_PUSH] = &pcasm::cPUSH;
    _handles[PCVM_OP_POP] = &pcasm::cPOP;
    _handles[PCVM_OP_CMP] = &pcasm::cCMP;
    _handles[PCVM_OP_CALL] = &pcasm::cCALL;
    /* 省略 */
```

在“cXXX”函数中，“c”表示“compile”，是“编译”的意思，“XXX”是指令的名称。以 cMOV 函数为例，代码如下，其他指令与之基本相同。

```
bool pcasm::cMOV(std::vector<unsigned char> &bytecodes)
{
    if (MOV_REG_IMM(PCVM_OP_MOV, bytecodes) ||
        MOV_REG_REG(PCVM_OP_MOV, bytecodes) ||
        MOV_MEM_IMM(PCVM_OP_MOV, bytecodes) ||
        MOV_MEM_REG(PCVM_OP_MOV, bytecodes) ||
        MOV_MEM_MEM(PCVM_OP_MOV, bytecodes) ||
```

```
        MOV_REG_LAB(PCVM_OP_MOV, bytecodes) ||
        MOV_MEM_LAB(PCVM_OP_MOV, bytecodes)) {
        return true;
    }
    return false;
}
```

以上函数的参数是汇编输出字节码的缓存队列，在函数中调用了一组 MOV_XXX_YYY 函数，从函数名可以看出 mov 指令的语法结构。函数会依次进行调用，如果前者不为“true”则进入下一个，直到其中一个成功为止。让我们到 MOV_REG_IMM 中看看，代码如下。

```
bool pcasm::MOV_REG_IMM(unsigned opcode, std::vector<unsigned char> &bytecodes)
{
    /* 语法: mov reg, 12345 */
    /* 以下语句将此指令的模式压入一个 tokens 的队列 */
    std::vector<int> tokens;
    tokens.push_back(TOKEN_REG);
    tokens.push_back(TOKEN_COMMA);
    tokens.push_back(TOKEN_IMM);
    /* 以 tokens 为目标来检查当前的代码是否匹配 */
    if (match(tokens) == false) {
        _error = PCASM_ERROR_SYNTAX_NOT_MATCH_TOKEN;
        return false;
}

    /* 如果语法匹配，则调用对应的指令汇编函数合并成一个 4 字节 */
    int ins = write_mode_op_reg_imm(opcode, _text_stack[0].reg,\
     _text_stack[2].imm);
    /* 将汇编完成的指令 ins 写入输出缓存 bytecodes */
    write_ins(ins, bytecodes);
    return true;
}
```

以上代码中有一个 _text_stack 变量用来获取寄存器及立即数的具体值。我们先看一下 match 函数的实现，代码如下。

```
bool pcasm::match(std::vector<int> tokens)
{
    pcasm_token token;
    _text_stack.clear();
    _err_on_token = TOKEN_NUMBER;
    if (tokens.empty()) return false;
    int count = 0;
    /* 遍历我们的目标标记队列 */
    for (auto t : tokens) {
        count++;   /* 计数标记 */
        /* 获取下一个标记 */
        token = next_token();
        /* 如果当前读取的标记与目标队列中取出的标记相同，则将它的字面值压入_text_stack */
        if (token.token == t) {
            _text_stack.push_back(token.text);
```

```
        }
        /* 如果没有匹配，则调用 rollback_token 函数回退 */
        else {
            rollback_token(count);
            _text_stack.clear();
            _error = PCASM_ERROR_SYNTAX_NOT_MATCH_TOKEN;
            _err_on_token = t;
            return false;
        }
}

    return true;
}
```

以上函数在标记不匹配时会将读取的标记再次压入标记流，代码如下。

```
void pcasm::rollback_token(int num)
{
    if (_token_pos - num < 0) _token_pos = 0;
    else _token_pos -= num;
}
```

以上函数的实现也比较简单，就是将当前标记的位置变量 _token_pos 减少参数 num 个来完成标记流的回退。与之对应的是 next_token 函数，用于从标记流中获取标记，代码如下。

```
pcasm_token pcasm::next_token()
{
    if (_token_source.empty()) {
        pcasm_token token;
        token.token = TOKEN_NUMBER;
        return token;
    }
    /* 这里最重要的就是控制_token_pos 的值，以达到控制标记流的作用 */
    return _token_source[_token_pos++];
}
```

匹配完成后，就要调用类似名为“write_xxx”的函数来汇编一条 4 字节的指令了。我们来看以下例子。

```
/* | 5 : opcode | 3 : mode | 24 : -| */
unsigned pcasm::write_mode_op(unsigned opcode) {
    unsigned ins = 0;;
    ins = opcode << 27;
    ins |= (PCVM_INS_MODE_OP << 24);
    return ins;
}
```

以上代码就是按照编码规则做了一些位操作而已。在真实的汇编器中，这部分无疑就是按照体系结构的编码规则操作。汇编器就是这么简单。

如果一条指令没有引用标记，则大体流程就是这样，差别无非就是调用不同模式的写入函数及填充不同的编码模式。而如果一条指令引用了标记，就会引出重定位（大多出现在跳转指

令及内存访问模式中）。

3．重定位信息

先说说为什么需要重定位。以“push label”指令为例，代码如下。

```
bool pcasm::PUSH_LAB(unsigned opcode, std::vector<unsigned char> &bytecodes)
{
    std::vector<int> tokens;
    /* 压入一个"引用标号"的标记 */
    tokens.push_back(TOKEN_REF_LABEL);
    /* 如果不匹配，则退出 */
    if (match(tokens) == false) {
        _error = PCASM_ERROR_SYNTAX_NOT_MATCH_TOKEN;
        return false;
}

    /* 建立一个重定位项目 */
    pcasm_relocate rel;
    rel.type = PCASM_REL_FIXLST20B;
    /* 当前指令的地址，以后重定位时需要 */
    rel.address = s_address;
    /* 符号的名称 */
    rel.symbol = _text_stack[0].str;
    /* 压入 s_relocates，重定位列表 */
    s_relocates.push_back(rel);

    /* 一条指令写入 opocde imm，后面的 imm 值为 0
     * 这个 imm 就是后面重定位时要修正的偏移
     */
    int ins = write_mode_op_imm(opcode, 0);
    write_ins(ins, bytecodes);
    return true;
}
```

以上代码就是一个普通的“push imm”操作，“push”指令后的立即数可以由一个标号表示。

重定位分为静态重定位与动态重定位两种。动态重定位在操作系统加载到内存中时使用。静态重定位用于在编译中将各个文件合并到一起时规划地址。而我们现在所做的就是静态重定位。下面这段代码可以解释静态重定位。

```
test:
;; 一些指令
push test
;; 其他指令
```

这个 test 是标号，如果 test 在 push 之前，那么可以很容易地计算出地址，在指令 push 后直接写入地址就好。如果 test 在 push 后面，那么在 push 时就不能直接写入地址了。因为此时还不知道 test 的地址，所以，要先遍历代码，统计所有的标记，然后将其转换成地址，在后面进行链接时再将地址填上。

4．标记定义

在 scanner 中，如果发现了标号的定义，则会返回 TOKEN_DEF_LABEL 标记，然后调用 cDefLabel 函数，代码如下。

```
bool pcasm::cDefLabel(pcasm_token &token)
{
    /* 如果没有发现符号，则创建符号 */
    if (s_symbols.find(token.text.str) == s_symbols.end()) {
        s_symbols[token.text.str] = std::shared_ptr<pcasm_label>(new pcasm_label);
        if (s_symbols[token.text.str] == nullptr) {
            _error = PCASM_ERROR_ALLOC_MEMORY;
            _errstr = token.text.str;
            return false;
        }
        /* 写入模块地址与当前地址 */
        s_symbols[token.text.str]->mode_address = _address;
        s_symbols[token.text.str]->address = s_address;
    }
    /* 符号存在 */
    else {
        _error = PCASM_ERROR_SYNTAX_SAME_LABEL;
        return false;
    }
    return true;
}
```

5．包含文件

在 parser 中，如果发现了 TOKEN_INCLUDE 标记，则调用 cInclude 函数，代码如下。

```
bool pcasm::cInclude(std::vector<unsigned char> &bytecodes)
{
    pcasm_token token;
    token = next_token();
    /* 如果为非字符串，则直接退出 */
    if (token.token != TOKEN_STRING) {
        _error = PCASM_ERROR_SYNTAX_NOT_MATCH_TOKEN;
        return false;
}

    /* 如果在符号表中没有找到这个字符串，则创建一个 pcasm 的类 */
    if (s_sources.find(token.text.str) == s_sources.end()) {
        s_sources[token.text.str] = std::shared_ptr<pcasm>(new pcasm());
        if (s_sources[token.text.str] == nullptr) {
            _error = PCASM_ERROR_ALLOC_MEMORY;
            return false;
        }
        /* 在标记中存在源文件的字符串处使用 make 生成对应的字节码 */
        if (s_sources[token.text.str]->make(token.text.str, bytecodes) == false) {
            _error = s_sources[token.text.str]->error();
            _errstr = token.text.str;
```

```
            return false;
        }
    }

    return true;
}
```

当遇到包含标签时，就进入指定的源文件进行汇编操作，并且在同一个字节码缓存内一起作为输出。

6．汇编立即数

在 parser 中，如果发现了 TOKEN_IMM 标记，则调用 write_imm 函数。此函数会将立即数标记中的数写入字节码队列，并返回写入的字节数，将最后返回的字节数增加到当前地址以修正这些地址。

```
int pcasm::write_imm(unsigned data, std::vector<unsigned char> &bytecodes)
{
    /* 按字节逐一写入字节码队列 */
    unsigned char *ptr = reinterpret_cast<unsigned char*>(&data);
    for (int i = 0; i < sizeof(int); i++) {
        bytecodes.push_back(*ptr++);
    }
    return sizeof(unsigned);
}
```

7．汇编字符串

在 parser 中，如果发现了 TOKEN_STRING 标记，则调用 write_string 函数，代码如下。

```
int pcasm::write_string(const std::string &str, std::vector<unsigned char> &bytecodes) {
    /* 遍历字符串并压入字节码队列 */
    for (auto c : str) {
        bytecodes.push_back(c);
}

    /* 字符串如果不够 4 字节对齐，则使用 0 字节补齐 */
    int s1 = str.size();
    int s2 = s_up4(s1);
    if (s2 > s1) {
        s1 = s2 - s1;
        while (s1--) {
            bytecodes.push_back(0);
        }
    }
    /* 将返回内容写入字节数（包含补齐的 0 字节） */
    return s2;
}
```

8．链接处理

链接过程是独立的，由外部进行调用，代码如下。

```
int pcasm::pclink(std::vector<unsigned char>& bytecodes)
{
    /* 如果字节码队列为空，则直接退出 */
    if (bytecodes.empty()) return false;
    /* 分配一块内存空间 */
    size_t size = bytecodes.size();
    /* 分配链接之后的空间 */
    unsigned char *ptr = new unsigned char[size + 1];
    if (ptr == nullptr) {
        return PCASM_ERROR_ALLOC_MEMORY;
}

    /* 遍历字节代码并写入 */
    size_t i = 0;
    for (auto b : bytecodes) {
        ptr[i++] = b;
}

    /* 遍历重定位项目 */
    for (auto r : s_relocates) {
        /* 从重定位表中找到符号，如果找不到，则返回链接失败 */
        if (s_symbols.find(r.symbol) == s_symbols.end()) {
            return PCASM_ERROR_LINK_NOT_FOUND_LABEL;
        }
        /* 从重定位项中取出要重定位的地址，然后取出要重定位的指令 */
        unsigned ins = *reinterpret_cast<unsigned*>(ptr + r.address);
        /* 将符号的地址设置给指令 */
        ins |= (s_symbols[r.symbol]->address & 0xFFFFF);
        *reinterpret_cast<unsigned*>(ptr + r.address) = ins;
}

    bytecodes.clear();
    /* 将重定位的代码再次写回字节码队列 */
    for (size_t i = 0; i < size; i++) {
        bytecodes.push_back(ptr[i]);
}

    if (ptr) delete[] ptr;
    return PCASM_ERROR_SUCCESS;
}
```

s_relocates 里保存了所有标号的重定位信息。填充这个结构，在进行指令汇编时，如果遇到形如“XXX_YYY_LAB”的函数，则填充这个变量。

D.4　让程序自己写程序：随机算法生成器的设计与实现

好了，有了以上基础就可以实现我们的真正目的了——让程序自己写程序。

D.4.1 框架设计

polycypt.h 文件中定义了如下框架，这个框架会随机选取一个算法工厂类。

```
/* 变形引擎配置 */
typedef struct {
    /* 算法引擎工厂索引，-1 表示随机 */
    int factory;
} polycypt_config;

class polycypt {
public:
    polycypt();
    polycypt(const polycypt_config &config);
    virtual ~polycypt();

    /* 运行
     * output_dir : 结果输出目录
     * template_dir : 算法模块目录
     */
    bool run(const std::string &output_dir, const std::string &template_dir);

    /* 出错代码 */
    int error();

private:
    void reset();                  /* 重设 */
    void load_algs();              /* 加载算法 */
    int random(size_t n = 100);    /* 生成一个随机值 */

private:
    /* 配置 */
    polycypt_config _config;
    /* 工厂类队列 */
    std::vector<std::shared_ptr<polycypt_factory> > _factories;
    /* 出错代码 */
    int _error;
};
```

在运行后，调用 make 函数进行加解密算法的对称生成，本引擎的目的就在于此。运行一次后就可以依照算法模板随机生成一套加解密对应的算法。此外，这只是一个框架引擎，并没有实现很强大的生成算法。

当前的工作原理是：先随机选取一个算法模板，然后调用加解密算法生成工厂类。当前引擎自带一个 DEMO，在 polycypt_alg0.cpp 文件中实现。如下函数会按照参数配置随机生成加解密算法。

```
/* output_dir : 结果输出目录
 * template_dir : 模板目录
 * generate_source : 如果为 TRUE，则输出生成的源代码
```

```
 */
bool polycypt_factory::make(const std::string &output_dir, \
     const std::string &template_dir, bool generate_source) {
  /* 获取临时模板目录 */
  std::string template_file = template_dir;
  if (*template_file.end() != '/') {
    template_file.append("/");
  }
  /* 默认的汇编源代码文件 */
  template_file.append("startup.asm");
  /* 读取模板文件 */
  _startup_template = s_read_file(template_file);
  if (_startup_template == "") {
    _error = POLYCYPT_ERROR_READ_FILE;
    return false;
  }

  std::string local_dir = output_dir;
  if (*local_dir.end() != '/') {
    local_dir.append("/");
  }

  /* 产生对应的加解密算法 */
  std::string encrypt_source, decrypt_source;
  if (generate(encrypt_source, decrypt_source) == false) {
    return false;
  }

  /* 如果 generate_source 为 TRUE，则将加解密算法的源代码输出 */
  if (generate_source) {
    /* 生成加解密源代码的路径 */
    std::string encrypt_source_path, decrypt_source_path;
    encrypt_source_path = local_dir + "encrypt.asm";
    decrypt_source_path = local_dir + "decrypt.asm";
    /* 生成对应的加解密源文件，其实这个文件没有用，就是让开发者看看生成的是什么 */
    if (s_write_text_file(encrypt_source_path, encrypt_source) == false) {
      _error = POLYCYPT_ERROR_WRITE_FILE;
      return false;
    }
    if (s_write_text_file(decrypt_source_path, decrypt_source) == false) {
      _error = POLYCYPT_ERROR_WRITE_FILE;
      return false;
    }
  }

  /* 将加解密算法编译成对应的字节码 */
  if (compile(encrypt_source, decrypt_source) == false) {
    return false;
  }
```

```
  /* 对解密算法生成的字节码进行链接 */
  if (pcasm::pclink(_encrypt_bytecodes) != PCASM_ERROR_SUCCESS) {
    _error = POLYCYPT_ERROR_LINK;
    return false;
  }

  /* 对加密算法生成的字节码进行链接 */
  if (pcasm::pclink(_decrypt_bytecodes) != PCASM_ERROR_SUCCESS) {
    _error = POLYCYPT_ERROR_LINK;
    return false;
  }

  /* 将生成的文件写入算法变形引擎固有的文件格式中 */
  std::string en_file, de_file;
  en_file = local_dir + "encrypt.pbc";
  de_file = local_dir + "decrypt.pbc";

  if (make_pcfile(en_file, _encrypt_bytecodes) == false) {
    return false;
  }

  if (make_pcfile(de_file, _decrypt_bytecodes) == false) {
    return false;
  }

  return true;
}
```

我们可以自行编写这个算法工厂类，然后由后台引擎来调用它。

D.4.2 算法工厂类

所有的插桩算法都继承自算法工厂类，所有算法实现的基础模型都是基于这个类实现的，示例如下。

```
class polycypt_factory {
public:
    polycypt_factory();
    virtual ~polycypt_factory();
    /* 产生加解密算法 */
    virtual bool generate(std::string &encrypt, std::string  &decrypt);
    /* 编译 */
    virtual bool compile(const std::string &encrypt, const std::string &decrypt);
    /* 产生编译后的字节码并输出 */
  virtual bool make(const std::string &output_dir, const std::string\
   &template_dir, bool generate_source=false);

protected:
    /* 这里都是内置的一些功能函数 */
    virtual void reset();
    /* 产生一个随机的符号名称 */
    virtual bool make_symbol(std::string &symbol);
    /* 产生随机数 */
    virtual int random(size_t n = 100);
```

```
    /* 产生 pc 格式的文件 */
    virtual bool make_pcfile(const std::string &path, \
    std::vector<unsigned char> &bytecodes);

protected:
    /* 产生加密算法 */
    virtual bool generate_encrypt(std::string &encrypt);
    /* 产生解密算法 */
    virtual bool generate_decrypt(std::string &decrypt);

protected:
    /*高级的算法生成支持 */
    unsigned _ip;                                    /* IP 地址寄存器 */
    unsigned _registers[PCVM_REG_NUMBER];            /* 寄存器队列 */
    bool _idle_registers[PCVM_REG_NUMBER];           /* 空闲寄存器 */
    int _error;                                      /* 错误代码 */

    /* startup.asm 的模板字符串 */
    std::string _startup_template;
    /* 汇编器 */
    pcasm _asmer;
    /* 加密算法字节码 */
    std::vector<unsigned char> _encrypt_bytecodes;
    /* 解密算法字节码 */
    std::vector<unsigned char> _decrypt_bytecodes;
};
```

以后的算法类主要重载 2 个接口，代码如下。

```
  virtual bool generate_encrypt(std::string &encrypt);
  virtual bool generate_decrypt(std::string &decrypt);
```

其他接口可以不实现。polycypt 类会调用选定的工厂类的 generate_encrypt 和 generate_decrypt 函数，产生 2 对字符串，随后调用 compile 进行遍历，这里就不具体展开了。

D.4.3　DEMO 介绍

本节程序 DEMO 的实现文件是 polycypt_alg0.cpp，继承自 polycypt_factory 工厂类。这个算法只是一个例子，并没有达到笔者想象中的程度。这个算法每次随机生成一个密码表，并且随机使用其中的一个，然后循环异或目标，代码如下。

```
class polycypt_alg0 : public polycypt_factory {
public:
    polycypt_alg0();
    virtual ~polycypt_alg0();

protected:
  /* 产生加密算法 */
  virtual bool generate_encrypt(std::string &encrypt);
  /* 产生解密算法 */
  virtual bool generate_decrypt(std::string &decrypt);

protected:
```

```
  /* 产生 xor 的汇编代码 */
  virtual bool generate_xor(std::ostringstream &oss);
  /* 产生随机的密钥 */
  virtual bool generate_keytab(std::ostringstream &oss);
  /* 产生算法起始 */
  virtual bool generate_algorithm_start(std::ostringstream &oss);
  /* 产生算法结束 */
  virtual bool generate_algorithm_end(std::ostringstream &oss);

private:
  std::ostringstream _keytab;      /* 密钥表 */
  int _keyidx;                     /* 密钥索引 */
};
```

1．xor 指令

因为虚拟机没有 xor 指令，所以只能通过模拟函数来完成。这个函数依次将汇编字符串写入字符串缓存，代码如下。

```
/* (~a & b) | (a & ~b) */
bool polycypt_alg0::generate_xor(std::ostringstream &oss) {
  /*
   * r10 = 密钥
   * r11 = 数据指针
   * r12 = 临时变量
   * r13 = 临时变量
   */
  oss << "@xor:\n";
  oss << "push r4\n";
  oss << "sub sp, 4\n";
  oss << "mov [sp], [r11]\n";
  oss << "pop r4\n";
  oss << "not r4\n";                  /* ~a */
  oss << "mov [r12], r4\n";
  oss << "and [r12], [r10]\n";        /* ~a & b */
  oss << "sub sp, 4\n";
  oss << "mov [sp], [r10]\n";
  oss << "pop r4\n";
  oss << "not r4\n";                  /* ~b */
  oss << "mov [r13], [r11]\n";
  oss << "and [r13], r4\n";           /* a & ~b */
  oss << "or [r13], [r12]\n";         /* (~a & b) | (a & ~b) */
  oss << "sub sp, 4\n";
  oss << "mov [sp], [r13]\n";
  oss << "pop r3\n";
  oss << "pop r4\n";
  oss << "ret\n";
  return true;
}
```

2．产生随机密钥表

随机产生一组密钥表，代码如下。

```
bool polycypt_alg0::generate_keytab(std::ostringstream &oss) {
  /* 密钥表 */
  oss << "@keytab:\n";
  /* 两个循环，16 * 16 = 256 */
  for (int i = 0; i < 16; i++) {
    for (int j = 0; j < 16; j++) {
      char buf[64] = {0};
      /* 随机产生一个 4 字节的数 */
      sprintf(buf, "x%x", random(0xFFFFFFFF));
      oss << buf;
      oss << " ";
    }
    oss << "\n";
  }
  oss << "\n";
  return true;
}
```

3．算法的起始与结束

产生算法的起始代码及结束代码，具体如下。

```
bool polycypt_alg0::generate_algorithm_start(std::ostringstream &oss) {
  /*
   * r10 = 密钥
   * r11 = 数据指针
   * r12 = 临时变量 1
   * r13 = 临时变量 2
   */
  generate_xor(oss);
  oss << _keytab.str();
  oss << "@key: 0\n";
  oss << "@data: 0\n";
  oss << "@tmp1: 0\n";
  oss << "@tmp2: 0\n";
  oss << "@algorithm:\n";
  oss << "push r5\n";
  oss << "push r4\n";
  oss << "mov r4, sp\n";
  oss << "add r4, 12\n";
  oss << "div r5, 4\n";

  /* 获取密钥表 */
  oss << "push r6\n";
  char buf[64] = {0};
  sprintf(buf, "%d", _keyidx);
  oss << "mov r6, " << buf << "\n";
  oss << "mul r6, 4\n";
```

```
  oss << "add r6, @keytab\n";
  oss << "mov r10, @key\n";
  oss << "mov [r10], [r6]\n";
  oss << "pop r6\n";

  /* 设置临时变量 */
  oss << "mov r12, @tmp1\n";
  oss << "mov r13, @tmp2\n";

  /* 设置数据指针 */
  oss << "mov r11, @data\n";

  /* 循环处理加解密功能 */
  oss << "@loop:\n";
  oss << "cmp r5, 0\n";
  oss << "je @algorithm_end\n";
  oss << "mov [r11], [r4]\n";
  oss << "call @xor\n";

  /* 保存结果 */
  oss << "mov [r4], r3\n";
  oss << "add r4, 4\n";
  oss << "sub r5, 1\n";
  oss << "jmp @loop\n";
  return true;
}
```

产生算法末尾块用于跳出函数，代码如下。

```
bool polycypt_alg0::generate_algorithm_end(std::ostringstream &oss) {
  oss << "@algorithm_end:\n";
  oss << "pop r4\n";
  oss << "pop r5\n";
  oss << "ret\n";
  return true;
}
```

4．_startup.asm

这套汇编代码就是一个启动函数，用于链接加解密代码，具体如下。

```
;;
;;  startup.asm
;;  polycypt
;;
;;  Created by logic.yan on 16/3/28.
;;  Copyright © 2016年 nagain. All rights reserved.
;;

@_start:
;; 从I/O端口0读取数据的长度
mov r4, 0
```

```
int 2
;; 如果 r5 为 0，则退出
cmp r5, 0
je @exit
;; r5 是数据的长度，按 4 字节对齐
;; ~3u & (3 + v)
mov r7, r5
add r7, 3
mov r6, 3
not r6
and r7, r6
;; 分配内存空间
sub sp, r7
int 0

;; 调用对应的算法
call @algorithm

@output2io:
;; 写数据到 I/O 端口 0，然后解密到 sp 指针处，r5 是加解密数据的长度
mov r4, 0
int 1
;; 设置输出 I/O 端口 0 的数据长度
mov r4, 0
int 3
;; exit
@exit:
;; 是否为内存空间
add sp, r7
int 9
```

D.4.4　小结

其实，上述内容没有实现笔者的目标，但框架算是有了。从工厂类的类变量中可以看出，其实笔者想做的是自扩散的生成算法，而不仅仅是这个框架。汇编语言其实还是不够方便，就算是简单的四则混合运算，汇编代码考虑的东西也非常多。如果有足够的时间，笔者会增加一个简单的语言编译器，然后随机生成对应的源代码，这样就不用考虑寄存器的选取、堆栈等问题了，生成的算法也更强大。如果读者对此感兴趣，也可以自行尝试。