



鲁宏伟/luhw@hust.edu.cn

## 第三讲 理解代码结构

# 控制流和程序

- 计算机程序中绝大多数逻辑是利用分支来实现的。
- 分支是最常见的程序构造，不管是不是用高级语言。程序测试一个或多个逻辑条件，并根据测试的结果转移到程序的不同部分去执行。
- 识别出分支并其含义及目的是最基本的代码级逆向工程任务之一。讨论的控制流构造有：单分支条件语句，双分支条件语句，多分支条件语句以及循环等。

# 解密函数

- 程序中最基本的构架块是例程，或者说函数。从逆向工程的观点来看，函数是很容易辩识的，因为函数都拥有"序言(prologues )"和"尾声 ( epilogues )"。
- 编译器会给几乎所有的函数都加上标准的初始化序列。这些初始化的具体代码与你使用的编译器有关，也与其他因素比如调用约定有关。
- 在 IA-32 处理器中，函数几乎都是使用 CALL指令来调用的。
- CALL指令将当前的指令指针压入堆栈，然后跳转向该函数的地址。利用CALL 指令，我们可以很容易地把函数调用和其他无条件跳转指令区别开来。

# 解密函数

- **内部函数**
- 当被调函数的实现代码与主调函数在同一个二进制可执行文件中时，被调函数即为内部函数。
- 当编译器为一个内部函数生成调用代码时，通常直接把被调函数的地址嵌入到代码中。因此，我们可以很容易地识别出内部函数。
- 下面是一个常见的内部函数调用的例子。

Call CodeSectionAddress

# 解密函数

- **导入函数**

- 当一个模块调用了在另外一个二进制可执行模块中实现的函数的时候，就发生了导入函数调用。
- 导入函数调用非常重要，因为在编译过程中，编译器无法确定到哪里找导入函数，所以无法把导入函数的地址直接嵌入代码中（像处理内部函数那样）。
- 通常，导入函数的调用是通过使用导入目录（Import Direcory）和导入地址表（Import Address Table，IAT）来实现的。导入目录是在运行时使用的，用它来解析函数名（即在目标可执行模块中匹配函数）。IAT 则用来存放目标函数的真实地址。主调函数可以从IAT中加载函数的指针对函数进行调用。

# 解密函数

- **导入函数**

- 下面的例子是一个典型的导入函数调用：

call DWORD PTR [IAT\_Pointer]

- 注意在指针前面的"DWORD PTR"——这个修饰词很重要，因为它告诉 CPU不是跳转到IAT\_Pointer 的地址上去，而是跳转到IAT\_Pointer所指向的地址上去。另外，还要记住的是这个指针通常不会有名字，通常只是一个指向IAT的地址。
- 识别导入函数调用比较简单，这是因为除了这种类型的调用外，其他函数极少会使用一个硬编码函数指针间接调用。但是，还是建议大家在逆向的早期就确定 IAT的位置，并用它来确认函数是不是导入的。
- 定位IAT相当容易，而且可以用各种各样的工具来定位IAT。

# 单分支条件

- 在大多数程序中，逻辑的最基本形式是由一个条件和一个紧随其后的分支条件组成。
- 在高级语言中，程序员写一条带有条件的if语句和一段满足该条件时就会执行的条件代码块。
- 下面是一个简单的例子：

```
_main_0  
var_CC  
var_8  
  
#include "stdio.h"  
  
int CallSFunction();  
  
int main(int argc, char* argv[])  
{  
    int nSomeVariable = 0;  
  
    if (nSomeVariable == 0)  
        CallSFunction();  
    return 0;  
}  
  
int CallSFunction() { return 1;}
```

```
proc near ; CODE XR  
= byte ptr -0CCh  
= dword ptr -8  
  
push    ebp  
mov     ebp, esp  
sub     esp, 0CCh  
push    ebx  
push    esi  
push    edi  
lea     edi, [ebp+var_CC]  
mov     ecx, 33h  
mov     eax, 0CCCCCCCCh  
rep stosd  
mov     [ebp+var_8], 0  
cmp     [ebp+var_8], 0  
jnz     short loc_411380  
call    sub_41106E
```

## 双分支条件

- 在高级语言中，最典型的的就是使用if-else这对关键字来实现双分支条件。
- 我们简单看一下编译器是如何编译双分支条件语句的。首先，在双分支条件中，分支条件指向的是"else"块，而不是指向条件语句后边接着的代码。其次，条件本身几乎总是被取反(因此只有在条件不满足时才会跳转到"else"语句块)，而主条件块(即指if块)就放在条件跳转之后(因此当条件满足时就会执行这段条件代码)。
- 条件块总是以一个无条件跳转结束，这样就跳过了"else"语句块——这是一个很好的识别双分支条件语句的标志。
- "else"块放在条件块末尾，紧接着无条件跳转之后。



# 双分支条件

- 下展示了普通的 if-else 语句在汇编语言中的样子。
- 注意无条件JMP 正好就在函数调用之后。第一个条件就是从这个地方跳过else块转而执行后面的代码的。在反汇编程序中**辨认**简单的"if-else"条件语句的基本模式是：确定紧跟在条件语句块后面的代码是不是以一个无条件跳转语句结束。

```
int CallSFunction();
int SomeFunction();

int main(int argc, char* argv[])
{
    int nSomeVariable = 7;

    if (nSomeVariable == 7)
        CallSFunction();
    else
        SomeFunction();
    return 0;
}

int CallSFunction() { return 1;}
int SomeFunction() { return 2;}
```

```
.text:0041139C      rep stosd
.text:0041139E      mov     [ebp+var_8], 7
.text:004113A5      cmp     [ebp+var_8], 7
.text:004113A9      jnz     short loc_4113B2
.text:004113AB      call    j_CallSFunction
.text:004113B0      jmp     short loc_4113B7
.text:004113B2      ; -----
.text:004113B2      loc_4113B2:      call    j_SomeFunction
.text:004113B2
.text:004113B7      loc_4113B7:      xor     eax, eax
.text:004113B7
```

主分支 →

## 双分支条件

- 大多数高级语言还支持一种略微复杂的双分支条件语句，即两个代码块要各用一个独立的条件语句。
- 这通常是将"if"和"else if"两个关键词组合使用来实现的。
- 这样，如果不满足第一个条件，程序就跳转到第二个条件语句并测试这个条件，如果两个条件都不满足，程序就直接跳过整个条件块。
- 如果满足了其中的一个条件，程序就会执行相应的条件块，执行完后接着执行下一个程序语句。

# 双分支条件

- 下图给出了这种控制流构造的高级语言/低级语言视图。

```
int CallSFunction();
int SomeFunction();

int main(int argc, char* argv[])
{
    int nSomeVariable = 7;

    if (nSomeVariable == 7)
        CallSFunction();
    else if (nSomeVariable < 7)
        SomeFunction();
    return 0;
}

int CallSFunction() { return 1;}
int SomeFunction() { return 2;}
```

```
.text:00411397      mov     eax, 0CCCCCCCCh
.text:0041139C      rep stosd
.text:0041139E      mov     [ebp+var_8], 7
.text:004113A5      cmp     [ebp+var_8], 7
.text:004113A9      jnz     short loc_4113B2
.text:004113AB      call    j_CallSFunction
.text:004113B0      jmp     short loc_4113BD
.text:004113B2      ; -----
.text:004113B2      loc_4113B2:
.text:004113B2      cmp     [ebp+var_8], 7
.text:004113B6      jge     short loc_4113BD
.text:004113B8      call    j_SomeFunction
.text:004113BD      loc_4113BD:
.text:004113BD      xor     eax, eax
```

## 多选一条件

- 有时候，程序员会写一些很长的有多个条件的语句，满足不同的条件就会执行不同的代码块。
- 高级语言中实现多选一条件语句的其中一个方法就是使用"switch"语句块，当然，也可以使用常规的"if"语句块来实现。  
程序员有时候必须使用"if"语句块的原因是"if"语句使我们能够实现更加灵活的条件语句。
- "switch"语句块实现不了复杂的条件，只能使用硬编码的常量。
- 相对于switch语句，一连串的"else if"语句允许你对每一个块使用任意复杂的条件——的确灵活多了。

## 多选一条件

- 识别多选一条件块的基本方法与识别普通的双分支条件块的方法非常相似，区别就在于是编译器添加了额外的"可选块"，可选块由一个或多个逻辑测试、真正的条件代码块、以及可以跳转到整个多选一条件块结束处的JMP指令组成。当然只有在条件满足的情况下才会执行JMP指令。
- "switch"块几种条件可以共用同一代码块。与"switch"块不同，"else-if"块中每个条件对应一个代码块。

# 多选一条件

- 下图展示了一个四条分支条件代码序列，其中包括一个"if"条件块和三个备选的"else-if"路径。

```
int SomeFunction() {return 1;};
int SomeOtherFunction() {return 2;};
int AnotherFunction() {return 3;};
int YetAnotherFunction() {return 0;};

int main(int argc, char* argv[])
{
    int nSomeVariable = 7;

    if (nSomeVariable < 10)
        SomeFunction();
    else if(nSomeVariable ==345)
        SomeOtherFunction();
    else if(nSomeVariable ==346)
        AnotherFunction();
    else if(nSomeVariable ==347)
        YetAnotherFunction();
    return 0;
}
```

```
.text:00411A3E      mov     [ebp+var_8], 7
.text:00411A45      cmp     [ebp+var_8], 0Ah
.text:00411A49      jge     short loc_411A52
.text:00411A4B      call    j_SomeFunction
.text:00411A50      jmp     short loc_411A80
.text:00411A52 ; -----
.text:00411A52      loc_411A52:                                     ; CODE
.text:00411A52      cmp     [ebp+var_8], 159h
.text:00411A59      jnz     short loc_411A62
.text:00411A5B      call    j_SomeOtherFunction
.text:00411A60      jmp     short loc_411A80
.text:00411A62 ; -----
.text:00411A62      loc_411A62:                                     ; CODE
.text:00411A62      cmp     [ebp+var_8], 15Ah
.text:00411A69      jnz     short loc_411A72
.text:00411A6B      call    j_AnotherFunction
.text:00411A70      jmp     short loc_411A80
.text:00411A72 ; -----
.text:00411A72      loc_411A72:                                     ; CODE
.text:00411A72      cmp     [ebp+var_8], 15Bh
.text:00411A79      jnz     short loc_411A80
.text:00411A7B      call    j_YetAnotherFunction
.text:00411A80      loc_411A80:                                     ; CODE
.text:00411A80                                     ; main
.text:00411A80      xor     eax, eax
```

# 复合条件

- 在实际的程序中，程序员经常会在一个条件判断语句中使用多个条件。在决定是否执行某个条件代码块之前要测试两个或者更多的条件，这种情况是非常常见的。
- 对于逆向者来说，这样的代码解读起来难度就比较大了，因为为逻辑测试组合生成的低级语言代码通常都比较难解读。
- 下面我们将介绍几种典型的复合条件以及怎样解读它们。
- 我们先讨论构建复合条件的几个最常见的逻辑运算符，然后再从低级语言和高级语言的角度讲解几种不同的复合条件块。

# 复合条件

- **逻辑运算符**
- 高级语言中有专门的操作符，这种操作符允许程序员在单个条件语句中使用复合条件。当要判断条件不止一个时，代码必须指明多个条件是怎样组合在一起的。
- 组合多个逻辑语句最常用的操作符是 AND 和 OR 操作符(注意不要把按位逻辑操作符混作一谈)。
- 在汇编语言中识别这样的代码通常比较容易，因为你会看到两个连续的条件都根据条件跳转到同一个地方。



# 复合条件

- **逻辑运算符**
- 右边是一个简单的例子：
- 在这个代码片断中，两个条件跳转都指向同一个代码地址 (AfterCondition)，这一事实揭示了这是一个组合逻辑条件。
- **这段代码的思路**很简单：首先测试第一个条件，如果不满足就直接跳转到 AfterCondition 的地址；如果满足第一个条件，才会测试第二个条件，如果不满足则还是跳转到 AfterCondition 的地址。
- 条件代码块就放在第二个条件分支之后（因此如果在两个跳转都没有执行的时候就可以立即执行条件代码块）。
- 解读真正的复合条件和前边讲的单分支条件的思路是一样的，它们的测试条件也被取反了。在这个例子里，**测试 Variable1 的值"不等于100"就是原始代码中测试 Variable1 的值是否"等于100"**。基于这些信息，你就可以重构出这个代码片断的源代码了

```
cmp    [Variable1], 100
jne    AfterCondition
cmp    [Variable2], 50
jne    AfterCondition
ret
AfterCondition:
...
```

```
if (Variable1==100 && Variable2==50) return ;
```

# 复合条件

- **逻辑运算符**
- 另一个常用的逻辑操作符是 OR 操作符，OR 操作符的含义是只要其中的一个条件满足整个复合条件就成立。在C和C+语言中 OR 操作符是用"||"符号来表示的。
- 在逆向过程中，判断包含OR 操作符的条件语句要比判断 AND 操作符的条件代码略微复杂一些。
- 实现 OR操作符的直接方法是对每个条件（这里不对条件取反）使用一个条件跳转，并加一个跳过整个条件代码块（在两个条件都不满足的情况下）的跳转指令。

# 复合条件

```
cmp [Variable1], 100
je ConditionalBlock
cmp [Variable2], 50
je ConditionalBlock
jmp AfterConditionalBlock
ConditionalBlock:
mov [Result], 1
AfterConditionalBlock:
...
```

- **逻辑运算符**
- 右边是这种复合条件的例子
- 跟前面一样，这段代码中最值得注意的是条件跳转指令都指向相同的代码。

**if (Variable1 == 100 || Variable2 == 50)**

**Result = 1 ;**

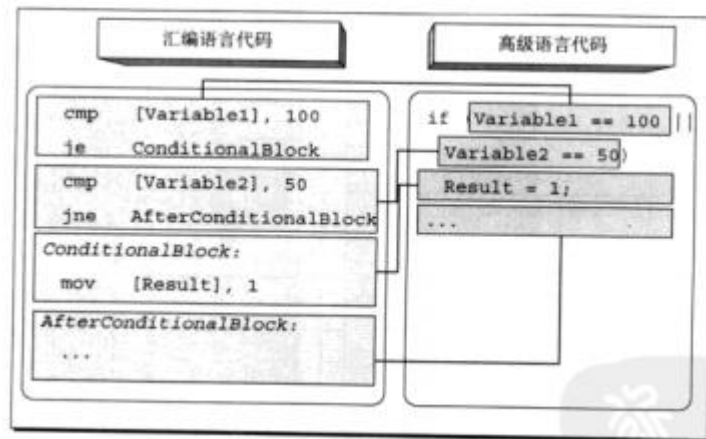
- 需要记住的是，**OR复合逻辑的条件跳转实际指向的是条件代码块（与我们前面讨论的 AND复合逻辑恰恰相反，AND复合逻辑的条件跳转指向的是条件代码块之后的代码）。**
- GCC和其他几个编译器都采用这种方法，这种方法的好处在于（至少从逆向工程的角度来看）相当易读和直观。但**它的运行性能要差一些**，因为只有所有条件都不满足才会执行到最后那条 JMP指令。

# 复合条件

- **逻辑运算符**
- 其他一些带有优化功能的编译器，如 Microsoft 的编译器采用了一种略有不同的方法来实现 OR 操作符，从而避免了这个额外的JMP指令的问题。
- Microsoft 的思路是：只将第二个测试条件取反，并让它指向条件代码块之后的代码，而第一个测试条件仍然指向条件代码块本身。

# 复合条件

- **逻辑运算符**
- 右图图解了在使用这种方法进行编译的情况下，同一个逻辑会变成什么样子。
- 第一个条件测试了Variable1 是否"等于 100"，这和它在源代码中的测试条件是一样的。
- 第二个条件测试了Variable2 是否"不等于 50"。这样做是因为我们希望第一个条件满足时就执行条件代码，第二个条件（被取反了）满足时不执行跳转。在第二个条件不满足时，程序就跳过条件代码块。

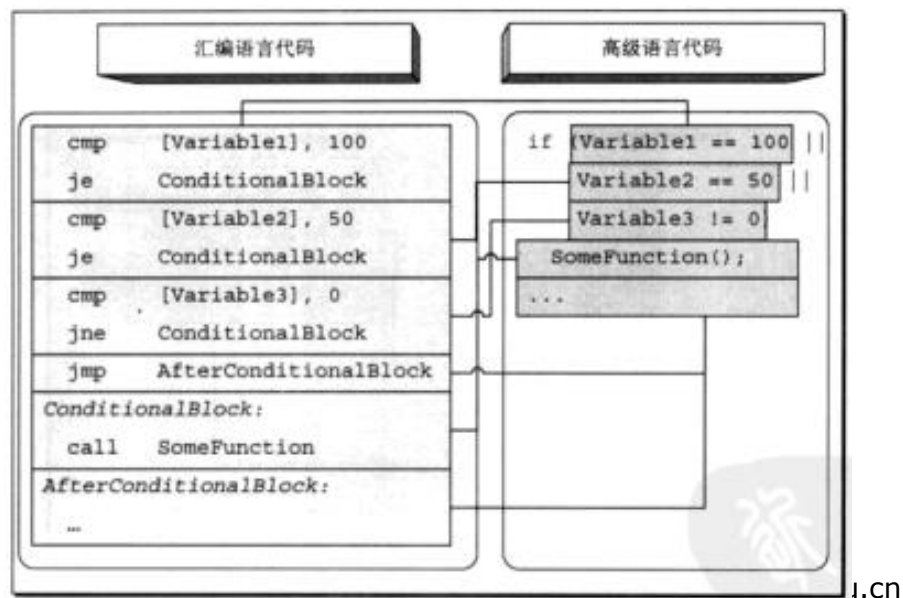


# 复合条件

- 如果碰到用逻辑操作符组合两个以上测试条件的情况下会是怎样的呢?通常,这种情况只是在处理只有两个条件的情况下所使用策略的扩展。
- 对于GCC 编译器来说,只是在那句无条件跳转指令之前再加一个条件测试罢了。

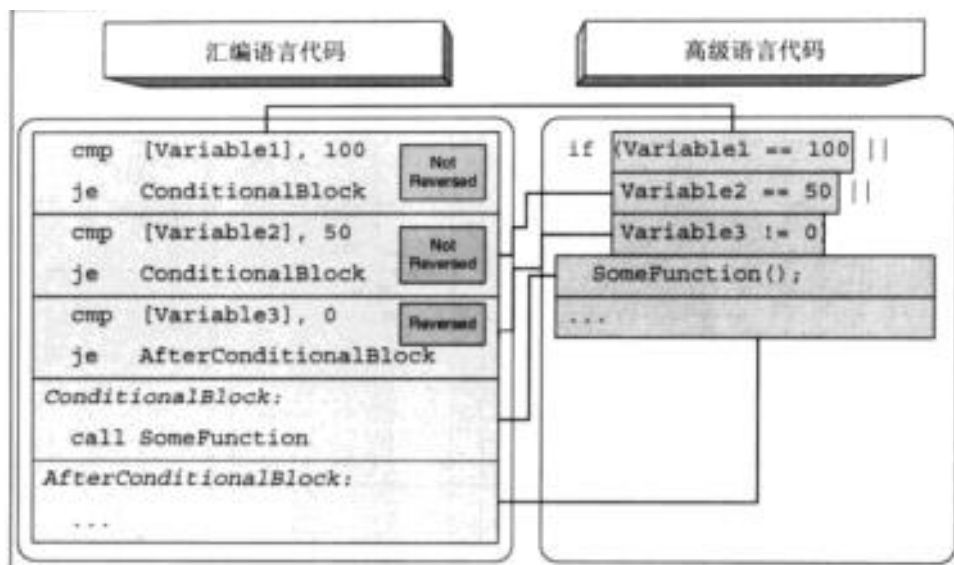
## 复合条件

- 在下图中所示的代码片断中，Variable1、Variable2 与前边例子中相同的数值进行了比较，Variable3 和0进行比较。只要所有的测试条件都是用 OR操作符连接的，编译器就会简单地再加一条指向条件块的条件跳转。与前边的情况一样，编译器总是在紧接着最后的条件分支指令之后放置一条无条件跳转指令。这条无条件跳转指令会在所有条件都不满足的情况下跳过条件块、直接到达条件块之后的代码的。



# 复合条件

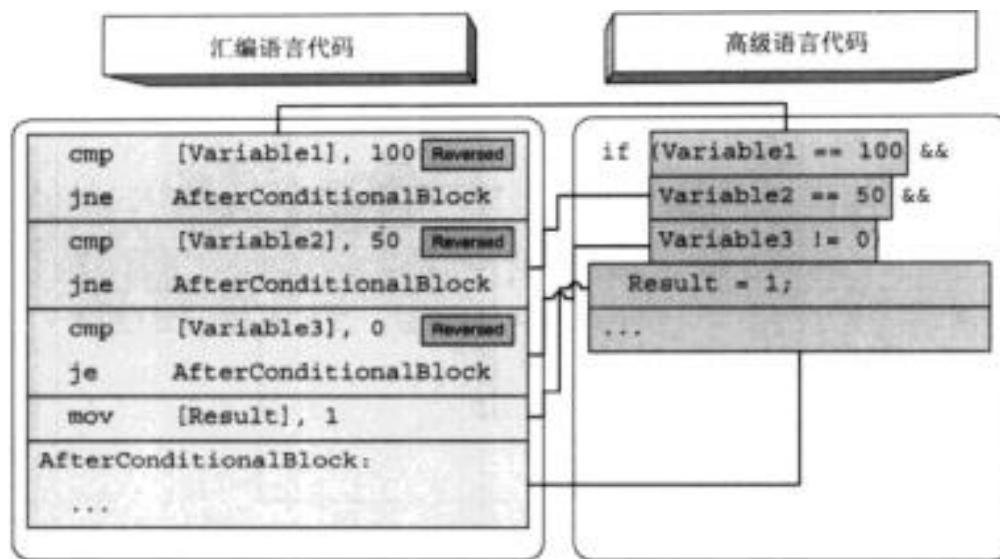
- 对于一些编译优化技术来说（对 Microsof 等编译器而言），方法是一样的，只是不使用无条件跳转，只将最后一个测试条件取反。其余的测试条件都直接以指向条件代码块的条件跳转来实现。
- 下图给出了采用第二种技术编译的代码的结果。





# 复合条件

- 下面我们来看看用AND操作符组合多于两个的测试条件的情况下会怎样。
- 在这种情况下，编译器只是增加了更多的取反了的测试条件，当它们满足的时候就跳过条件代码块（**记住测试条件被取反了**），如果不满足就继续测试下一个条件（或者如果是最后一个测试条件的话就转向条件代码块本身执行）。



## n-路分支条件(switch块)

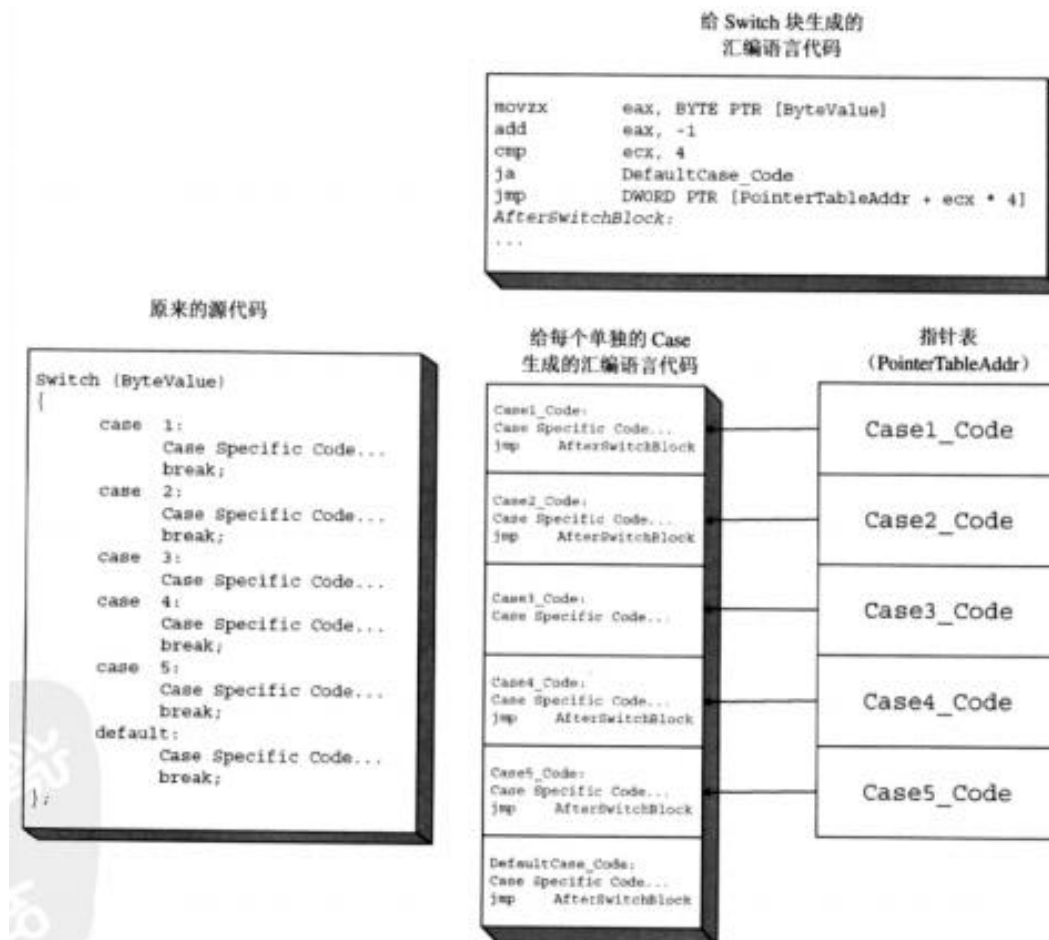
- switch块（或者称为n-路分支条件块）常常用于同一个操作数取得不同值的时候需要采用不同的操作的情况。
- switch 语句块实质上是让程序员创建有关所有可能的取值以及相应的响应代码的表。需要注意的是，一段响应代码可以对应多个取值。
- 编译器有几种处理switch 语句的方法，具体使用哪一种方法取决于switch块的大小以及接收取值的范围。
- 下面我们将介绍两种最常见的n-路分支的实现：**表实现**和**树实现**。

# n-路分支条件(switch块)

- **表实现**
- 对大型switch块来说最高效的实现方法（从运行时的观点来看）是生成指针表。
- 表实现的思路是：编译switch语句中的每个代码块，并把指向每一个代码块的指针记录到指针表中。之后，在switch块执行的时候，switch块所操作的操作数就被用作指针表的索引，处理器就根据操作数的取值跳到正确的代码块。要注意的是，这个过程并不是函数调用，而是一个经过指针表的无条件跳转。
- 指针表通常放在包含该switch块的函数之后，但并非总是如此——这取决于所采用的编译器。当有一个函数表放在代码段的中间时，你基本可以确信这就是一张 switch 块的指针表。实际上，代码段内的硬编码指针表并不常见。

# n-路分支条件(switch块)

- **表实现**
- 下图演示了怎样用指针表实现n-路分支条件。



# n-路分支条件(switch块)

- **树实现**
- 当switch块的分支条件并不适合用表实现的话，编译器会采用二叉树搜索策略来尽可能快地找到要找的单元。
- **二叉树实现的基本思路**是：根据要查找项的取值大小将它们分成个数基本相同的两组，并记录每一组的取值范围。对每一组再重复这个分组过程，直至分到单独的项为止。
- 在查找的时候，你首先确定你要找的值在两个大组中的哪一个（表明那个组中会包含你要找的项）。然后，你再在前一步确定的组内确定哪个子组中会包含你要找的项，就这样不断重复直到找到你要找的项。

# n-路分支条件(switch块)

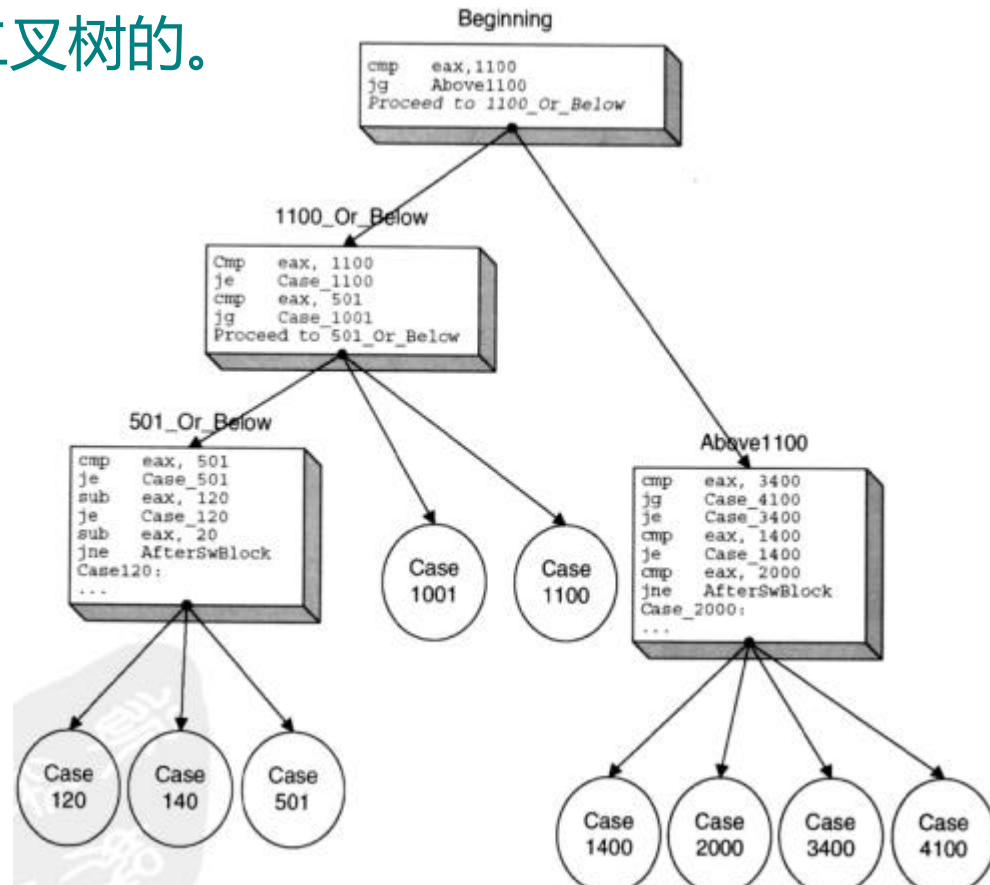
- **树实现**
- 要用二叉树实现switch块，编译器必须首先把switch块表示成二叉树。
- 它的基本思路是：由编译器会在生成代码的时候首先测试给定值是在第一组中还是第二组中，而不是在运行时用这个给定值与每一个可能的case进行比较。然后编译器跳转到另一个代码区，与这个稍小一点的子组里的值进行测试比较。这一过程将一直重复进行，直至找到正确的项或者退出了条件块（如果没有找到和要找的值匹配的case 的话）。

# n-路分支条件(switch块)

- **树实现**
- 我们来看一下这段用C语言实现的switch块，并考察一下编译器是怎样将它变换成二叉树的。

```
switch (Value)
```

```
{  
  case 120:  
    Code...  
    break;  
  case 140:  
    Code...  
    break;  
  case 501:  
    Code...  
    break;  
  case 1001:  
    Code...  
    break;  
  case 1100:  
    Code...  
    break;  
  case 1400:  
    Code...  
    break;  
  case 2000:  
    Code...  
    break;  
  case 3400:  
    Code...  
    break;  
  case 4100:  
    Code...  
    break;  
};
```



# n-路分支条件(switch块)

- **树实现**
- 基于树实现的 n-路分支条件有一个独特的特点——要对于一个寄存器执行多次减法，这一点使得我们在阅读反汇编代码时比较容易识别出它。这些减法的后面通常跟着的是跳向特定 case 块的条件跳转。
- 编译器通常从传给条件块的原始值开始，逐渐从原始值中减去某些值(这些值通常都是case块的值)，并不断测试结果是不是为0。
- 这是用最少的代码来确定要跳转到哪一个 case 块的一种高效的方法。



# 循环

- 我们可以把循环看作是前边讨论过的条件代码块，只不过它是不断地重复执行直到条件不再满足为止。
- 循环通常（而不是总是）会包含一个某种类型的计数器，用以控制离循环结束还剩多少次迭代。
- 基本上任何一种高级语言的循环都可以分为两类：**先测试循环**（其逻辑测试在循环体之前，“循环体”的代码会被重复执行）和**后测试循环**（逻辑测试在循环体之后）。

# 循环

- **先测试循环**
- 尽管先测试循环在效率上要比后测试循环稍稍差一些，但是它可能是使用得最广泛的循环构造。
- **先测试循环的效率稍差的原因**是：汇编语言代码必须用两个跳转指令来表示一个先测试循环，一个是在循环一开始的条件分支指令（当该条件不再满足时就终止循环），另一个是在循环结尾处的无条件跳转指令——用来跳转回循环的开始处。

# 循环

- **先测试循环**
- 我们来看一个简单的先测试循环的例子，看看编译器是怎样实现它的：

```
int c, array[1000];
c = 0;
while (c < 1000)
{
    array[c] = c;
    c++;
}
return 0;};
```


```
.text:00412F69      mov     [ebp+var_4], eax
.text:00412F6C      mov     [ebp+var_C], 0
.text:00412F73      loc_412F73:                                     ; CODE XREF
.text:00412F73      cmp     [ebp+var_C], 3E8h
.text:00412F7A      jge     short loc_412F94
.text:00412F7C      mov     eax, [ebp+var_C]
.text:00412F7F      mov     ecx, [ebp+var_C]
.text:00412F82      mov     [ebp+eax*4+var_FB4], ecx
.text:00412F89      mov     eax, [ebp+var_C]
.text:00412F8C      add     eax, 1
.text:00412F8F      mov     [ebp+var_C], eax
.text:00412F92      jmp     short loc_412F73
.text:00412F94      ; -----
.text:00412F94      loc_412F94:                                     ; CODE XREF
.text:00412F94      xor     eax, eax
```

# 循环

- 后测试循环
- 我们来看一个简单的后测试循环的例子，看看编译器是怎样实现它的：

```
int c, array[1000];
c = 0;
do
{
    array[c] = c;
    c++;
}while (c < 1000);
return 0;
```

```
.text:00412F69      mov     [ebp+var_4], eax
.text:00412F6C      mov     [ebp+var_C], 0
.text:00412F73
.text:00412F73  loc_412F73:                                ; CODE XREF
.text:00412F73      mov     eax, [ebp+var_C]
.text:00412F76      mov     ecx, [ebp+var_C]
.text:00412F79      mov     [ebp+eax*4+var_FB4], ecx
.text:00412F80      mov     eax, [ebp+var_C]
.text:00412F83      add     eax, 1
.text:00412F86      mov     [ebp+var_C], eax
.text:00412F89      cmp     [ebp+var_C], 1000
.text:00412F90      jnl     short loc_412F73
.text:00412F92      xor     eax, eax
```



luhw@hust.edu.cn

---