12 | 标准库：非本地跳转与可变参数是怎样实现的？

2022-01-07 于航

讲述：于航
时长 18:04 大小 16.55M

你好，我是于航。

我曾在第 05 讲 中介绍过，C 语言中的函数调用是在 `call` 与 `ret` 两个指令的共同协作下完成的。这个过程包括程序执行流的转移、栈帧的创建、函数代码的执行、资源的清理，一直到函数调用完毕并返回至调用点的下一条指令上。总的来看，函数在正常情况下的调用流程是稳定有序的。

但实际上，这种以函数为单位的“顺序”执行流并不能完全满足 C 语言在使用时的所有应用场景。因此，C 标准从 C90 开始，便为我们提供了名为 “setjmp.h” 的标准库头文件。通过使用该头文件提供的两个接口 setjmp 与 longjmp，我们能够在函数调用过程中，实现对执行流的跨函数作用域转变。而对于上述这种函数执行流程上的变化，我们一般称它为“**非本地跳转**（Non-local Jump）”。

除此之外，在正常的 C 语法中，函数在被实际调用时，只能接收与其函数原型和函数定义中标注的，类型及个数相同的实参。而为了进一步增强 C 函数在使用上的灵活性，同样是在 C90 之后的标准中，C 语言还为我们提供了名为 “stdarg.h” 的头文件。配合使用在该头文件中定义的宏，我们便可以在 C 代码中定义“**可变参数函数**（Variadic Function）”。而可变参数函数与普通函数的最大区别就在于，它们在被调用时可以接收任意多个实参，而无需提前在函数原型或定义中声明这些参数的信息。

那么，今天我们就来聊一聊 C 语言中的非本地跳转与可变参数函数。在接下来的内容中，我会分别介绍它们的基本用法与实现原理。

非本地跳转

非本地跳转的概念并不直观，既然如此，我们就换一个方式来理解它：在继续深入之前，我们先来看一个与它相对的简单概念，本地跳转（Local Jump）。相信了解过这个概念后，你会对非本地跳转与程序执行流变化的对应关系有更直观的理解。

本地跳转

在 C 语言中，本地跳转一般是指由 goto 语句完成的程序执行流的转移过程。这里我们来看一个简单的例子：

```c
#include <stdio.h>
int main(void) {
  int n;
head:
  scanf("%d", &n);
  if (n > 0) goto head;
  return 0;
}

```

```asm
.LC0:
        .string "%d"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
.L2:
        lea     rax, [rbp-4]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    __isoc99_scanf
        mov     eax, DWORD PTR [rbp-4]
        test    eax, eax
        jle     .L3
        jmp     .L2
.L3:
        mov     eax, 0
        leave
        ret
```

可以看到，在上图左侧的 C 代码中，我们通过设置标签 “head” 的方式，显式地指定了函数 main 内部一个可能的“执行跳入点”。随着程序的运行，当用户的输入满足 if 条件判断语句的要求时，我们便通过 goto 语句，将程序的执行流程重新转移到了标签 head 的所在位置。从右侧红框内的汇编代码中也可以看到，对应的执行流程转移过程是通过 jmp 指令来完成的。按照这样的方式，程序得以在 main 函数内部，根据外部的用户输入，动态地调整其执行流程。

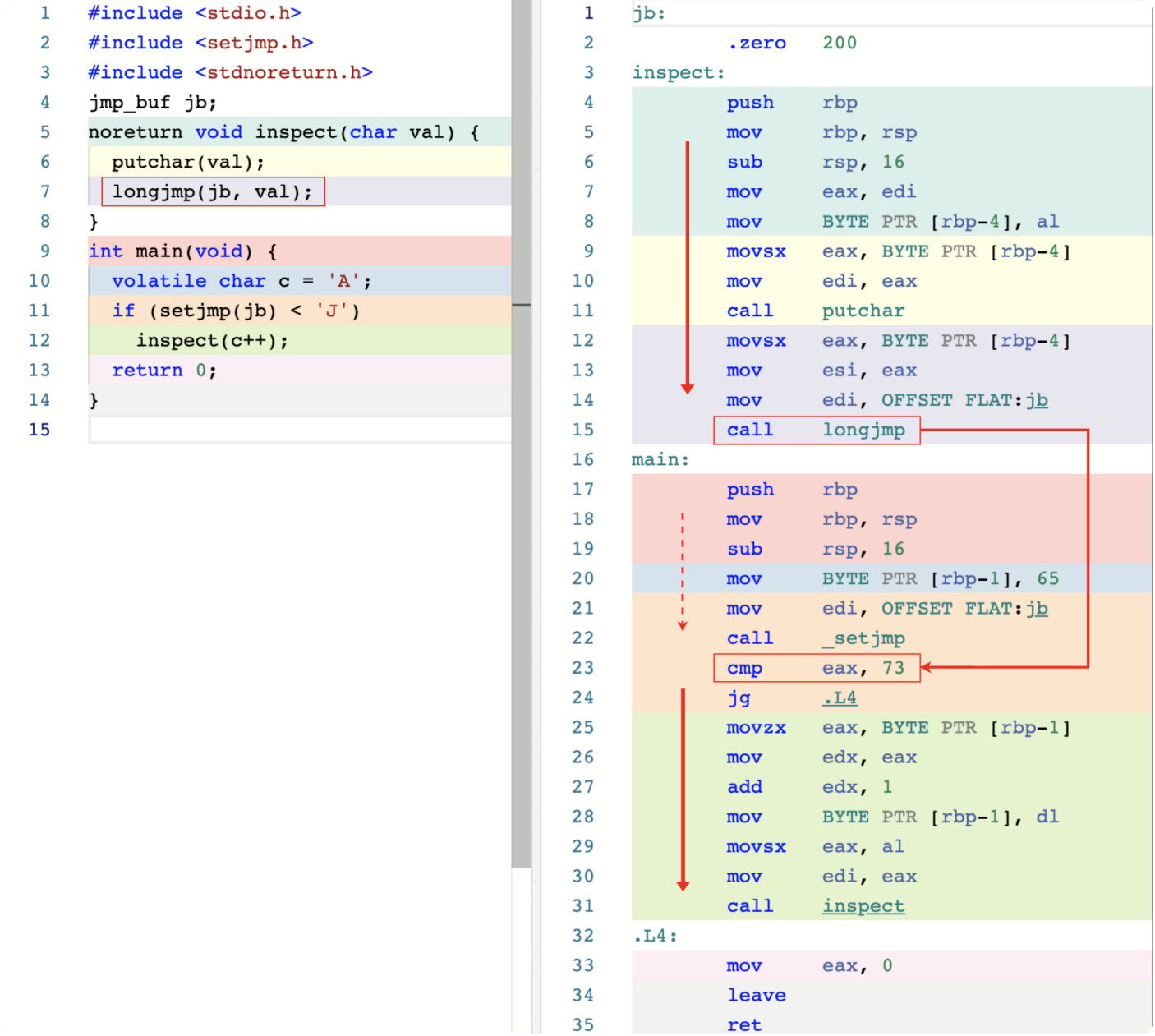
而我们之所以称这种通过 goto 语句实现的程序执行流变化为本地跳转，是因为**在这种方式下的执行流程转移仅能够发生在当前程序运行所在的某个具体函数中**。相对地，程序无法做到从某个函数体的执行中途，直接将其执行流转移到其他函数的内部。“跨函数”的调用仅能够通过常规的 call 与 ret 指令来实现。

但是，非本地跳转却可以打破这个限制。接下来，我们具体看看与它有关的 setjmp 和 longjmp 函数。

setjmp 与 longjmp 函数

在 C 语言中，非本地跳转的实现依赖于标准库头文件 setjmp.h 内的两个函数 setjmp 与 longjmp。关于它们的具体使用方式，你可以点击这个链接参考更多信息，这里我就不展开了。

接下来，我们直接使用这两个函数编写一段简单的代码，并观察程序运行时非本地跳转的实际执行方式。代码如下所示：



```c
#include <stdio.h>
#include <setjmp.h>
#include <stdnoreturn.h>
jmp_buf jb;
noreturn void inspect(char val) {
  putchar(val);
  longjmp(jb, val);
}
int main(void) {
  volatile char c = 'A';
  if (setjmp(jb) < 'J')
    inspect(c++);
  return 0;
}
```

```asm
jb:
        .zero   200
inspect:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     eax, edi
        mov     BYTE PTR [rbp-4], al
        movsx   eax, BYTE PTR [rbp-4]
        mov     edi, eax
        call    putchar
        movsx   eax, BYTE PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:jb
        call    longjmp
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     BYTE PTR [rbp-1], 65
        mov     edi, OFFSET FLAT:jb
        call    _setjmp
        cmp     eax, 73
        jg      .L4
        movzx   eax, BYTE PTR [rbp-1]
        mov     edx, eax
        add     edx, 1
        mov     BYTE PTR [rbp-1], dl
        movsx   eax, al
        mov     edi, eax
        call    inspect
.L4:
        mov     eax, 0
        leave
        ret
```

上图中左侧为 C 代码，右侧为对应的汇编代码。此时，你可以先停下来，观察整个程序的大致实现方式。程序的实际运行结果是连续打印出了从 A 到 J 共 10 个字符。这里建议你动手实践下，并验证我们的结论。

接下来，回到左侧的 main 函数，让我们来看一下程序的执行细节。

首先，我们定义了名为 c 的字符变量，并将它的值初始化为字符 A。通过标注 `volatile` 关键字，编译器不会对该变量的使用过程进行任何优化。然后，我们在 if 条件语句中调用了函数 setjmp，并将声明为 jmp_buf 类型的全局变量 jb 传递给了它。该函数的调用返回值会与字符 J 进行比较，若满足小于关系，则会调用另外的函数 inspect 来打印变量 c 的内容，并同时递增 c 的值。

接下来把目光移到 C 代码的第 5 行，inspect 函数的定义部分。这里，该函数接收一个字符变量，并使用 putchar 函数将它的值打印了出来。可以看到，函数在定义时被标注了 `_Noreturn` 关键字，也就是说，inspect 函数在调用结束后不会通过正常的 `ret` 指令退出。而之所以会发生这样的情况，便是由于代码在第 7 行调用的 longjmp 函数。

实际上，随着 longjmp 函数执行完毕，程序的执行流程将会直接跳转到 main 函数中 `call setjmp` 指令（也就是调用 setjmp 函数的那条指令）的下一条指令上。在上图右侧的汇编代码中，我使用红色箭头标注出了程序在机器代码层面的具体执行顺序，其中，实线部分对应的机器代码将会在程序运行过程中执行多次。

可以看到，当程序执行流由 inspect 函数转移回 main 函数后，借由 `cmp` 指令，程序得以再次检查寄存器 rax 中的值是否小于字符 J 对应的 ASCII 码值 73。若条件成立，则重复之前的程序执行逻辑；否则，程序直接退出。此时，rax 寄存器中存放的值便对应于 longjmp 函数在被调用时传入的第二个参数值。

不同于常规的函数调用过程，**非本地跳转为我们提供了一种可以暂存函数调用状态，并在未来某个时刻再恢复的能力**。借助这种能力，我们便能够实现跨函数的、精确到某条具体语句的程序执行流程跳转。那这种能力是如何实现的呢？

运作原理

通过上面的例子我们得知，非本地跳转实际上是由 setjmp 与 longjmp 这两个函数共同协作完成的。我们来看看它们究竟都做了哪些事情。

setjmp 函数在执行时，会将程序此刻的函数调用环境信息，存储在由其第一个参数指定的 jmp_buf 类型的对象中，并同时将数值 0 作为函数调用结果返回。而当程序执行到 longjmp 函数时，该函数便会从同一个 jmp_buf 对象中再次恢复之前保存的函数调用上下文。通过这种方式，程序的执行流程得到了“重置”。

那么，与函数调用环境相关的信息有哪些呢？我曾在 05 讲 中介绍过，以 x86-64 为例，SysV 调用约定中规定，属于 Callee-saved 类型的寄存器信息需要在 `call` 指令调用时，由被调用函数负责保存和恢复。这也就意味着，这类寄存器中实际上存放着与当前调用函数（caller）有关的上下文状态信息。因此，当被调用函数通过 `ret` 指令返回时，这些寄存器中的“旧值”仍需要被调用函数继续使用。所以，对于 setjmp 函数的实现，是不是只要把所有 Callee-saved 寄存器中的值进行保存就可以了呢？让我们来做个实验吧！

自定义实现

在这个简单的实验中，我们将构建自己的 setjmp 与 longjmp 函数，并重新编译之前的 C 示例程序，来让它使用这两个函数的自定义实现。这里，我们将直接编写汇编代码，并在最后以对象文件的形式将它们链接到程序中使用。需要注意的是，下面我将要介绍的实现方式仅适用于 x86-64 平台。对于其他平台，使用的汇编指令以及需要保存的寄存器可能有所区别，但整体思路基本一致。

按照之前的方案，我们可以直接写出 setjmp 函数的汇编实现，代码如下所示：

复制代码

```
1  .global setjmp
2  .intel_syntax noprefix
3  setjmp:
4    mov  QWORD PTR [rdi], rbx
5    mov  QWORD PTR [rdi+0x8], rbp
6    mov  QWORD PTR [rdi+0x10], r12
7    mov  QWORD PTR [rdi+0x18], r13
8    mov  QWORD PTR [rdi+0x20], r14
9    mov  QWORD PTR [rdi+0x28], r15
10   lea  rdx, [rsp+0x8]
11   mov  QWORD PTR [rdi+0x30], rdx
12   mov  rdx, QWORD PTR [rsp]
13   mov  QWORD PTR [rdi+0x38], rdx
14   xor  eax, eax
15   ret
```

其中，前两行以 “.” 开头的语句为汇编器指令，它们用来指示汇编器应该如何处理接下来的汇编代码。第一行的 “.global” 指令指示汇编器可以将符号 setjmp 暴露给链接器使用；第二行指令指示汇编器，接下来的汇编代码将采用 Intel 语法格式。

在 setjmp 的函数体实现中，由于 SysV 调用规范的约束，作为接收函数第一个实参的 rdi 寄存器，其内部将保存有传入的 jmp_buf 对象的首地址。你可以直接将该对象看成是一个具有足够大小的字节数组，而这里我们要做的，就是把各个 Callee-saved 寄存器中的值按顺序放入这个数组中进行暂存。

紧接着，第 4 ~ 9 行的代码将寄存器 rbx、rbp、r12、r13、r14，以及 r15 的值进行了暂存；第 10 ~ 11 行的代码将 setjmp 函数调用之前的 rsp 寄存器的值进行了暂存；第 12 ~ 13 行的代码将 setjmp 函数调用后的返回地址进行了暂存，这个地址将由 longjmp 函数进行使用。最

后，第 14 行代码将寄存器 rax 的值置零，以作为该函数的返回值。至此，setjmp 函数的实现便完成了。

按照相同的思路，我们也可以直接得到 longjmp 函数实现的汇编代码，如下所示：

```
.global longjmp
.intel_syntax noprefix
longjmp:
  xor  eax, eax
  cmp  esi, 0x1
  adc  eax, esi
  mov  rbx, QWORD PTR [rdi]
  mov  rbp, QWORD PTR [rdi+0x8]
  mov  r12, QWORD PTR [rdi+0x10]
  mov  r13, QWORD PTR [rdi+0x18]
  mov  r14, QWORD PTR [rdi+0x20]
  mov  r15, QWORD PTR [rdi+0x28]
  mov  rsp, QWORD PTR [rdi+0x30]
  jmp  QWORD PTR [rdi+0x38]
```

这里需要注意的是代码的第 5~6 行。longjmp 函数在调用时需要遵守的一个规则是：如果传递给它的第二个实参为 0，则使用数值 1 对其进行替换。这样做是为了能够通过 rax 寄存器中的不同“返回值”，区分当前代码是在 setjmp 函数调用后首次被执行的，还是由 longjmp 恢复函数调用环境后再次被执行的。这里我给你留个思考题：这两行汇编代码是如何实现上述的替换逻辑的？欢迎在评论区告诉我你的想法。

至此，setjmp 与 longjmp 这两个函数的自定义实现便准备完毕了。我们将这些汇编代码分别存放到名为 setjmp.s 与 longjmp.s 的文本文件中。然后，通过下面这两行命令，我们能够将它们编译成各自对应的 .o 对象文件。

```
gcc -c setjmp.s -o setjmp.o
gcc -c longjmp.s -o longjmp.o
```


接下来，为了在 C 代码中正确调用这两个函数，我们还需要为它们提供相应的函数原型，以及 jmp_buf 类型的详细定义。包含有上述这些内容以及原始示例程序的完整 C 代码如下所示：

```c
#include <stdio.h>
#include <stdnoreturn.h>
// 定义 jmp_buf 类型：
typedef long jmp_buf[8];
// 提供函数原型：
int setjmp(jmp_buf);
noreturn void longjmp(jmp_buf, int);
// 原始 C 示例程序代码：
jmp_buf jb;
noreturn void inspect(char val) {
  putchar(val);
  longjmp(jb, val);
}
int main(void) {
  volatile char count = 'A';
  if (setjmp(jb) < 'J')
    inspect(count++);
  return 0;
}
```

在这段代码的第 4 行，我们为类型 jmp_buf 提供了相应的定义。由于该对象内部仅需要连续存放 8 个 64 位的寄存器值，因此我们将它定义为具有 8 个元素的长整型数组。紧接着，我们也分别为函数 setjmp 与 longjmp 提供了与标准库中实现完全一致的函数原型。最后，将这段代码保存在文件 main.c 中，然后通过下面的命令，我们便可以编译并运行整个程序。

```c
gcc main.c setjmp.o longjmp.o -o main && ./main
```

请你动手实践下，并观察程序的输出。在正常情况下你将会得到同示例程序完全一致的运行结果。

当然，为了便于你理解，这里我简化了 setjmp 与 longjmp 函数的实现细节，但它们的“核心思想”却可以通过这短短的几行汇编代码完全体现出来。

通常来说，在 C 语言中，非本地跳转主要用来实现异常处理、协程等功能。标准中仅规定了 setjmp 函数可以正常使用的几种特定上下文情况，因此在使用它时，你需要十分小心，以防出现未定义行为。

可变参数函数

下面，我们来看另一个话题，可变参数函数。不知道你有没有察觉，这门课从开篇词开始，就一直在各个示例程序中使用这类函数。我们在初学 C 语言时，编写的第一个 “Hello, world” 程序中使用到的 printf 函数，就是其中一种。而仔细观察你会发现，printf 函数所能接收的参数个数，实际上完全取决于我们在它第一个参数，即格式控制字符串中指定的“格式控制符”的个数。

基本使用

可变参数函数能够接收不定数量的实参，而为了定义这样一个函数，我们需要配合使用由标准库头文件 stdarg.h 提供的类型与宏函数。让我们来看一个简单的例子，代码如下所示：

复制代码

```
#include <stdio.h>
#include <stdarg.h>
void print_sum(int count, ...) {
  int sum = 0;
  va_list ap;
  va_start(ap, count);
  for (int i = 0; i < count; ++i)
    sum += va_arg(ap, int);
  va_end(ap);
  printf("%d\n", sum);
}
int main(void) {
  print_sum(4, 1, 2, 3, 4);
  return 0;
}
```

这里我们定义了一个名为 print_sum 的可变参数函数，该函数会计算传递给它的所有实参（第一个实参除外）之和。而它的第一个参数，将被用来统计函数在调用时所传入的其余参数的个数。

到这里，你可以先暂缓脚步，仔细看看在上面的代码中，我们是如何使用类型 va_list，以及宏函数 va_start、va_arg，以及 va_end 的。如果你对它们还不太了解，可以先点击这个链接来查看对它们用法的更多说明。

运作原理

接下来，让我们把目光放到上面所说的几种类型与宏函数的底层实现上，来探究它们在内部是如何对传入函数的多个实参进行管理的。

对于 Clang 和 GCC 来说，这些宏函数在展开后，会调用由编译器在内部实现的 builtin 函数。因此，如果想要了解它们的实现细节，便需要深入到编译器代码的内部。但由于编译器实现较为庞杂，这个方法并不适合我们快速了解相关内容。那有没有更方便的办法呢？答案是有的。

实际上，在遵循 System V AMD64 ABI 的 x86-64 计算机上，ABI 已经详细规定了编译器应该如何实现函数的可变参数列表。因此，通过阅读其对应的文档，我们便能够了解到上述这些宏函数和类型的一种实现方式。而对于其他平台，虽然实现细节上可能稍有不同，但整体思路不会有太大的差异。那么接下来，我就带你看看标准中的可变参数函数是如何定义的。

在我们继续之前，你可能会有这样的疑问：这里我提到的 System V AMD64 ABI 与在 05 讲中介绍的同名调用约定有什么关系呢？

这是一个很好的问题。实际上，System V AMD64 ABI 的全称为 “System V AMD64 **Application Binary Interface**”。它是一种描述了应用程序应该如何在机器代码中进行某种操作的标准规范，而我们之前介绍的“函数调用约定”便是其重要内容之一。关于 ABI 的更多内容，我将在这门课的“C 程序运行原理篇”中再为你详细介绍。

SysV ABI 中详细规定了可变参数列表的实现要求，但信息较多，规则较为复杂。这里为了方便你理解，我会**按照上述代码中函数 print_sum 的执行顺序，从整体的视角为你讲述它在机器指令层面对传入实参的处理方式**。如果你想了解更多的细节性信息，可以通过这个链接下载 ABI 文档，并参考从第 54 页开始的内容。

首先，当 print_sum 函数被 `call` 指令调用前，由于所传入的参数均为整型，因此它们可以被直接存放在由 SysV ABI 函数调用约定规定的参数寄存器中。这里，对应的 5 个数字值实参将被依次存放到寄存器 rdi、rsi、rdx、rcx，以及 r8 中。

接着，函数被调用。寄存器 al 中将会存放有传入函数的浮点参数的个数，该寄存器的值将会被编译器使用，以进行相应优化。同时，一块名为 “Register Save Area”（后简称 RSA）的栈内存区域将会被构建。而每一个通过寄存器传入函数的实参值，都会按照 rdi、rsi、rdx、rcx、r8、r9、xmm0~xmm15 的寄存器先后顺序，被拷贝并存放在这段内存中。

函数继续执行，在代码的第 5 行，va_list 类型的变量 ap 被定义，而其中存放有用于支持 va_arg 宏函数正常运作的必要信息。通常来说，该类型可以被定义为如下所示的 C 数据结构。可以看到，va_list 为一个指针，指向了包含有 4 个字段的结构对象。

```
typedef struct {
  unsigned int gp_offset;  // 下一个整型数据相较于 RSA 的偏移；
  unsigned int fp_offset;  // 下一个浮点数据相较于 RSA 的偏移；
  void *overflow_arg_area;  // 指向使用栈进行传递的数据；
  void *reg_save_area;  //  指向 RSA 的指针；
} va_list[1];
```

这里我将 va_list 所指向结构对象内部各个字段的具体功能，以注释的方式标注了出来。你可以先浏览一遍，有个大致的印象，下面我就向你介绍它们的用途。

在 print_sum 函数实现的第 3 行（即代码第 6 行），va_start 宏函数对 va_list 所指向的结构对象进行了初始化。在这个过程中，字段 gp_offset 将会被设置为下一次将要从 RSA 中读取的整型实参，其值距离 RSA 开始位置的偏移。类似地，fp_offset 则用于浮点实参值。除此之外，指针 overflow_arg_area 将会指向每一个使用栈进行传递的实参值；最后的 reg_save_area 将指向 RSA 的起始地址。

代码的第 8 行，va_arg 宏函数将会根据传入的 va_list 指针以及想要提取的实参类型，从 RSA 中取出相应的数据值。在这个过程中，ap 结构体内字段 gp_offset、fp_offset，以及 overflow_arg_area 的值会随着数据的不断提取而得到更新。在实现细节上，va_arg 在提取实参的过程中，还需要考虑对由多个寄存器存放的实参（大于 64 位）的处理，以及栈上数据指针对齐等问题。

最后，当传入的实参被提取完毕后，通过代码第 9 行的宏函数 va_end，ap 结构体得到了清理。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

今天我主要介绍了 C 语言中的非本地跳转与可变参数函数这两方面内容。通过使用由 C 标准库头文件 setjmp.h 与 stdarg.h 提供的一系列接口，我们能够在程序中轻松地使用这两种特性。

非本地跳转为我们提供了一种可以临时保存函数执行上下文，并在未来某时刻再重新恢复的能力。通过这种方式，我们可以在 C 语言中实现异常捕获、协程等特殊功能。非本地跳转的基

本实现方式是在执行 setjmp 函数时，将此刻所有 Callee-saved 寄存器的值进行暂存。而在未来某一时刻 longjmp 函数调用时，再将这些寄存器的值进行复原。通过这种方式，Caller 函数的原始执行状态便会得到恢复。

可变参数函数则让我们使用函数的方式变得更加灵活。配合使用类型 va_list，以及宏函数 va_start、va_arg、va_end，我们可以在函数体内，依次获取由外部调用者传入的若干个不定参数。而为了保证兼容性，变长参数列表的实现细节需要遵循目标平台和操作系统对应的 ABI 规范。

思考题

你知道怎样使用非本地跳转来实现 try…catch 语句吗？可以尝试编写几个宏，来在 C 代码中实现类似的异常抛出和捕获效果，并在评论区分享你的实践经验。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇　11 | 标准库：深入理解标准 IO

下一篇　13 | 标准库：你需要了解的 C 并发编程基础知识有哪些？

精选留言 (9)

写留言

liu_liu
2022-01-07

对 setjmp 和 longjmp 的理解：

setjmp 将寄存器的值保存到全局变量中，同时保存 rsp 和返回地址的值，设置 eax = 0。

在上面的例子中，返回地址就是 cmp eax, 73 这句指令的地址。

longjmp 的作用：

1. 将保存的值从全局变量中全部恢复，其中最重要的是返回地址和 rsp。因为之后的 jmp 指令会跳转到返回地址去继续执行，也就是重新回到 cmp eax, 73。

2. 根据第二个参数更新 eax 的值。

那么当回到返回地址继续执行时，数据也都准备好，这样就达到了目的。

另外有个问题，对于 rsp 的保存，不知理解的对不对？

lea rdx, [rsp+0x8]
mov QWORD PTR [rdi+0x30], rdx

在 call _setjmp 时，会将返回地址压栈，此时 rsp -= 8。所以需要加上 8 得到之前的 rsp，也就是 rdx = rsp + 8。

作者回复: 没错的哈。

2

白凤凰
2022-01-10

setjmp和longjmp的应用场景是什么呢？实际开发过程中什么时候可以用这两个函数，是为了解决什么问题呢？

作者回复: 除了我们在文章中提到的可以用来实现 try...catch，协程以外，也可以用在信号处理程序中。比如参考这边的例子：https://www.gnu.org/software/libc/manual/html_node/Longjmp-in-Handler.html。除此之外，我好像没有再见到过其他用例。其他同学如果有另外的使用场景也可以补充哈。

共 2 条评论 >

1

pedro
2022-01-07

try catch 实现参考：https://zhuanlan.zhihu.com/p/86547911

1

shk1230
2022-02-26

那么if (setimp(jb) < 'J')，不就是恒等式？setjmp(jb)返回0时

作者回复: setjmp 初次调用会返回 0，而后续 longjmp 在每次返回到 setjmp 时都会携带更新后的值。

ZR2021
2022-01-10

对了，还有个感觉比较怪异的点，就是setjmp 函数的参数是直接传递的值，而非指针，然后setjmp 函数里还真修改了传入的值到jb 变量中！ 按照我之前的理解，值传递的时候调用函数先将变量传给rdi 寄存器，然后被调函数里将rdi 里的内容拷贝至本函数的传参变量所在的栈，再使用这个变量的值。 但是setjmp函数貌似没有新开辟传参变量的栈，而是直接找到调用函数传参的地址，然后对其进行修改，这样，调用函数里的变量就得到了修改，不知道我理解的对不对。还有，这种通过值传递的而不是指针，但最后将传入的变量内容修改的形式，哪些地方用到的比较多呢？

作者回复: setjmp 调用时传递的 jb 是一个指针哈。jb 对应的类型 jmp_buf 是这样定义的：typedef long jmp_buf[8];

共 2 条评论 >

ZR2021
2022-01-09

老师，可变参数里面va_start初始化ap的时候，那个count参数需不需要的？理论上也被存到RSA里面了吧，看起来好像有点多余，但是看到很多可变参数都会用到这个参数，很是奇怪

作者回复: va_start 在调用时的第二个参数，需要传入函数参数列表中显式定义的最后一个带名字的参数，这个是必须要传的。

共 2 条评论 >

Ping
2022-01-09

老师，yield函数是不是用这里的setjmp和longjmp实现的？

作者回复: setjmp 和 longjmp 是在 C 语言中实现协程这种语言特性的一种方式，但其他语言比如 Python、JavaScript 中的 yield 关键字是怎样实现的，这个就要具体情况具体分析了。当然，在 C 中也可以实现 yield，比如参考这个 Post：https://stackoverflow.com/questions/17478264/implementing-yield-in-c

liu_liu
2022-01-07

https://gist.github.com/silan-liu/47cf4d65e5f49b84c7f499a7d4fd24f2

尝试写了一个版本，不知思路对不对？

作者回复: 思路是对的，不过可以用宏封装一下，让使用方式更自然一些。可以参考这篇文章：http://groups.di.unipi.it/~nids/docs/longjump_try_trow_catch.html


liu_liu
2022-01-07

cmp esi, 0x1
adc eax, esi

如果 longjmp 传入的第二个参数为 0，那么此时 esi = 0。

cmp 之后的结果，会让进位 CF = 1。

adc 表示进位加和，eax = eax + esi + CF = 1。

这样就达到了令 eax = 1。



1