ELF文件——栈回溯

原创 LeoSoldOut

前言

本文以libunwind库对栈回溯流程进行描述。

libunwind栈回溯流程

libunwind包含两套使用接口，分别以前缀unw_和_Unwind标识，其中_Unwind前缀的接口是供C++异常处理的高级函数接口，unw前缀的则是更为底层通用的接口。根据libunwind代码中的configure.ac文件，在arm架构下是不使能C++的异常处理的，所以栈回溯使用的接口均为前缀位unw的函数接口。

```
AC_MSG_CHECKING([whether to enable C++ exception support])
AC_ARG_ENABLE(cxx_exceptions,
AS_HELP_STRING([--enable-cxx-exceptions],[use libunwind to handle C++ exceptions]),,
[
# C++ exception handling doesn't work too well on x86
case $target_arch in
  x86*) enable_cxx_exceptions=no;;
  aarch64*) enable_cxx_exceptions=no;;
  arm*) enable_cxx_exceptions=no;;
  mips*) enable_cxx_exceptions=no;;
  tile*) enable_cxx_exceptions=no;;
  *) enable_cxx_exceptions=yes;;
esac
])
```

栈回溯以函数unw_backtrace为起点：

```
int
unw_backtrace (void **buffer, int size)
{
  unw_cursor_t cursor;
  unw_context_t uc;
  int n = size;

  tdep_getcontext_trace (&uc); //保存当前寄存器的值

  if (unlikely (unw_init_local (&cursor, &uc) < 0))  //初始化cursor，保存当前寄存器的值
    return 0;

  if (unlikely (tdep_trace (&cursor, buffer, &n) < 0)) //快速查找，在cache中查找，如果在cache中没有找到，则需要根据elf文件逐个回
    {
      unw_getcontext (&uc);
      return slow_backtrace (buffer, size, &uc); //逐个函数进行回溯
    }

  return n;
}
```

```
libunwind_i.h
#define tdep_getcontext_trace           unw_getcontext
libunwind-common.h.in
#define unw_getcontext(uc)              unw_tdep_getcontext(uc)

libunwind-arm.h
#ifndef __thumb__
```

```
#define unw_tdep_getcontext(uc) (({                                 \
  unw_tdep_context_t *unw_ctx = (uc);                               \
  register unsigned long *unw_base __asm__ ("r0") = unw_ctx->regs;  \
  __asm__ __volatile__ (                                            \
    "stmia %[base], {r0-r15}"                                       \
    : : [base] "r" (unw_base) : "memory");                          \
  }), 0)
#else /* __thumb__ */
#define unw_tdep_getcontext(uc) (({                                 \
  unw_tdep_context_t *unw_ctx = (uc);                               \
  register unsigned long *unw_base __asm__ ("r0") = unw_ctx->regs;  \
  __asm__ __volatile__ (                                            \
    ".align 2\nbx pc\nnop\n.code 32\n"                              \
    "stmia %[base], {r0-r15}\n"                                     \
    "orr %[base], pc, #1\nbx %[base]\n"                             \
    ".code 16\n"                                                    \
    : [base] "+r" (unw_base) : : "memory", "cc");                   \
  }), 0)
#endif
//将R0-R15的值依次保存到uc->reg[0] - uc->reg[15]中
```

```
int unw_init_local (unw_cursor_t *cursor, unw_context_t *uc)
{
  return unw_init_local_common(cursor, uc, 1);
}

static int unw_init_local_common (unw_cursor_t *cursor, unw_context_t *uc, unsigned use_prev_instr)
{
  struct cursor *c = (struct cursor *) cursor;

  if (!tdep_init_done)
    tdep_init ();

  Debug (1, "(cursor=%p)\n", c);

  c->dwarf.as = unw_local_addr_space;
  c->dwarf.as_arg = uc;

  return common_init (c, use_prev_instr);
}

static inline int
common_init (struct cursor *c, unsigned use_prev_instr)
{
  int ret, i;
//将此前保存的寄存器的值的 地址 保存到struct cursor之中
  c->dwarf.loc[UNW_ARM_R0] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R0);
  c->dwarf.loc[UNW_ARM_R1] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R1);
  c->dwarf.loc[UNW_ARM_R2] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R2);
  c->dwarf.loc[UNW_ARM_R3] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R3);
  c->dwarf.loc[UNW_ARM_R4] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R4);
  c->dwarf.loc[UNW_ARM_R5] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R5);
  c->dwarf.loc[UNW_ARM_R6] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R6);
  c->dwarf.loc[UNW_ARM_R7] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R7);
  c->dwarf.loc[UNW_ARM_R8] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R8);
  c->dwarf.loc[UNW_ARM_R9] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R9);
  c->dwarf.loc[UNW_ARM_R10] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R10);
  c->dwarf.loc[UNW_ARM_R11] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R11);
  c->dwarf.loc[UNW_ARM_R12] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R12);
  c->dwarf.loc[UNW_ARM_R13] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R13);
  c->dwarf.loc[UNW_ARM_R14] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R14);
  c->dwarf.loc[UNW_ARM_R15] = DWARF_REG_LOC (&c->dwarf, UNW_ARM_R15);
  for (i = UNW_ARM_R15 + 1; i < DWARF_NUM_PRESERVED_REGS; ++i)
    c->dwarf.loc[i] = DWARF_NULL_LOC;

  ret = dwarf_get (&c->dwarf, c->dwarf.loc[UNW_ARM_R15], &c->dwarf.ip);
  if (ret < 0)
    return ret;

```

```
  /* FIXME: correct for ARM?  */
  ret = dwarf_get (&c->dwarf, DWARF_REG_LOC (&c->dwarf, UNW_ARM_R13),
                   &c->dwarf.cfa);
  if (ret < 0)
    return ret;

  c->sigcontext_format = ARM_SCF_NONE;
  c->sigcontext_addr = 0;
  c->sigcontext_sp = 0;
  c->sigcontext_pc = 0;

  /* FIXME: Initialisation for other registers.  */

  c->dwarf.args_size = 0;
  c->dwarf.stash_frames = 0;
  c->dwarf.use_prev_instr = use_prev_instr;
  c->dwarf.pi_valid = 0;
  c->dwarf.pi_is_dynamic = 0;
  c->dwarf.hint = 0;
  c->dwarf.prev_rs = 0;

  return 0;
}

#include <linunwind_i.h>
#define DWARF_REG_LOC(c,r)      (DWARF_LOC((unw_word_t) tdep_uc_addr((c)->as_arg, (r)), 0))                      \
#define DWARF_LOC(r, t)         ((dwarf_loc_t) { .val = (r) })
Ginit.c
# ifdef UNW_LOCAL_ONLY

HIDDEN void *
tdep_uc_addr (unw_tdep_context_t *uc, int reg)
{
  return uc_addr (uc, reg);
}
uc_addr (unw_tdep_context_t *uc, int reg)
{
  if (reg >= UNW_ARM_R0 && reg < UNW_ARM_R0 + 16)
    return &uc->regs[reg - UNW_ARM_R0];           //保存寄存器值的地址，根据代码回溯，这个地址应该是在栈上的，
  else
    return NULL;
}
```

libunwind 默认会依次使用三种回溯方式对栈进行回溯，其中，DWARF方式是通过eh_frame section来进行回溯，只有配置了CONFIG_DEBUG_FRAME宏才会编译该段内容，在configure.ac中对于arm架构，该宏默认打开。本文暂且只对使用exidx的栈回溯方式进行说明。

```
int unw_step (unw_cursor_t *cursor)
#ifdef CONFIG_DEBUG_FRAME
    if (UNW_TRY_METHOD(UNW_ARM_METHOD_DWARF))  //DWARF方式回溯
    {
        ret = dwarf_step (&c->dwarf);
    }
#endif
    if (UNW_TRY_METHOD (UNW_ARM_METHOD_EXIDX))  //unwind table方式回溯
    {
      ret = arm_exidx_step (c);
    }
    if (UNW_TRY_METHOD(UNW_ARM_METHOD_FRAME))   //frame pointer方式回溯
    {
        ...
    }
//使用对应方式进行回溯时，还需要判断对应的回溯方式是否被使能
#define UNW_TRY_METHOD(x)   (unwi_unwind_method & x)

//Gglobal.c中，由该全局变量的定义，可以更改该全局变量的值使能对应的栈回溯方式
HIDDEN int unwi_unwind_method = UNW_ARM_METHOD_ALL;
```

```
#define UNW_ARM_METHOD_ALL          0xFF
#define UNW_ARM_METHOD_DWARF        0x01
#define UNW_ARM_METHOD_FRAME        0x02
#define UNW_ARM_METHOD_EXIDX        0x04

```

exidx 结构

ELF文件中与栈回溯相关的setion包括exidx和extab section，信息如下例：

```
  [14] .ARM.extab        PROGBITS        000063e0 0063e0 00030c 00   A  0   0  4
  [15] .ARM.exidx        ARM_EXIDX       000066ec 0066ec 0001a0 00  AL 11   0  4
```

exidx section为一个表格，表格中每一项由如下结构体进行描述：

```
struct EHEntry {
  uint32_t Offset; //函数起始偏移，需要结合当前地址计算函数
  uint32_t World1; //字节码/待解释数据
};
```

EHEntry.Offset并不直接指向其作用的函数地址（ELF文件的物理偏移），而是间接的利用该值根据重定位规则计算得到，以保证exidx section在各个平台上的通用性。重定位类型为R_ARM_PREL31，在此重定位类型中函数地址的计算规则为当前地址+Offset，其中Offset为一个31位的有符号数，其最高位固定为0。在对函数地址计算时，首先需要将这个31位的有符号数转变为32位的有符号数，转变的实现过程如下：

```
offset = ((long)offset << 1) >> 1;
```

例如，0x7FFFFFFF在31位有符号数中为-1，但是在32位有符号数中为214748364，为保证其在32位有符号数中仍保持-1的值，此时需要对32的数据先左移一位，去掉在31位数中没有使用到的最高位（0x7FFFFFFF << 1 = 0xFFFFFFFE），再将其右移一位，使32位数的最高位保持和31位数的最高位一致（符号位相同），即（0xFFFFFFFE >> 1 = 0xFFFFFFFF = -1），此时32位有符号数解析出来的值才是31位有符号数的真实值。以readelf工具读取的unwind table来验证计算的正确性：

```
Unwind section '.ARM.exidx' at offset 0x66ec contains 52 entries:

0x2178 <deregister_tm_clones>: 0x80b0b0b0
  Compact model index: 0
  0xb0      finish
  0xb0      finish
  0xb0      finish

0x21d8 <__do_global_dtors_aux>: @0x63e0
  Compact model index: 1
  0xb1 0x08 pop {r3}
  0x84 0x00 pop {r14}
  0xb0      finish
  0xb0      finish
```

以unwind table中的第二项为例，其在ELF文件中的HEX值为：0x66ec(exidx section addr) + 8(entry size) = 0x66f4，如下：

```
000066e0  8101b2da  1fafb0b0  00000000  7fffba8c
000066f0  80b0b0b0  7fffbae4  7ffffce8  7fffbb1c
00006700  80b0b0b0  7fffbb18  7ffffce4  7fffbb24
```

Offset的值为0x7fffbae4，其地址为0x66f4，函数的地址即为：

```
0x66f4 + ((long)0x7fffbae4 << 1) >> 1 = 0x66f4 + (0xffff75c8 >> 1) = 0x66f4 + 0xffffbae4 = 0x21d8
```

计算出的函数地址与readelf工具读取出来unwind table中的函数地址一致。

EHEntry.Word1记录了如何对对应函数进行栈回溯的字节码，该元素的解析分类如下：

- EHEntry.Word1 = 1，表示Cannot Unwind
- EHEntry.Word1 最高位为0，低31位组成一个prel31类型的值，指向extab（exception-handling table）中的一项，该项内容记录了如何对该函数进行栈回溯的字节码

- EHEntry.Word1 最高位为1，低31位组成字节码，相当于extab中的一项

extab entry有两种模式，两种模式通过最高位的值来区分：

1. The generic model ：最高位为0，低31位组成一个prel31类型的值，指向personality routine
2. The Arm-defined compact model：最高位固定为1，其他位组成如下：

31	30-28	27-24	23——0
1	0	index	data for personalityRoutine[index]

index包含0、1、2三个值，分别代表不同的回溯路径(personality routine)：

- 0，Su16 - short frame unwinding description followed by descriptors with 16-bit scope.对应函数__aeabi_unwind_cpp_pr0
- 1，Lu16 - Long frame unwinding description followed by descriptors with 16-bit scope.对应函数__aeabi_unwind_cpp_pr1
- 2，Lu32 - Long frame unwinding description followed by descriptors with 32-bit scope.对应函数__aeabi_unwind_cpp_pr2

对于short frame unwinding（index 为0）来说，32位数据的0-23位包含有三条指令，每条指令占据8位；对于long frame unwinding(index 为1或2)，32位数据的16-23位指示了除了本字以外，后续还包括多少个字（4个字节）用于存储该函数的回溯指令，32位数据的0-7，8-15则包含两条指令。仍以上面unwind table中的第二项为例，其EHEntry.Word1为0x7ffffce8，最高位为0，则0x7ffffce8为一条extab entry地址的prel31值，通过计算可得：

```
0x66f8 + ((long)0x7ffffce8 << 1) >> 1 = 0x66f8 + 0xfffffce8 = 0x63e0
```

查看0x63e0处的内容，如下：

```
000063e0  8101b108  8400b0b0  00000000  7fffbca8
000063f0  01b10884  00b0b0b0  0001ffff  7fffbc98
00006400  00a8b0b0  0001ffff  7fffbc8c  00a8b0b0
```

31	30-28	27-24	23-16	—0
1	0	1	1	b108

该值最高位为1，index为1，此时16-23位中的数据表示后续还有1个字保存有指令，即0x8400b0b0，0-7，8-15位保存有指令0xb108，extab地址0x63e0和extab的内容0xb108，0x8400b0b0与readelf读取出来的unwind table entry项一致。至于两个字节0xb1 0x08代表的是一条指令还是两条指令，则需要对照字节码进行解析。

实际上，arm unwind table只是根据函数的入栈出栈回溯sp的值，然后根据压入栈中的lr的值进行栈回溯，只能回溯出来函数但是并不能回溯出来每一个栈帧中寄存器的值，即调用该函数时的寄存器的上下文。

根据arm unwind table进行栈回溯时，通过二分法将unwind table中函数的地址与当前PC进行比较，直到该PC值大于其中一项的函数地址，而小于它下一项的函数地址，最终使用该项中的指令码进行回溯。

exidx栈回溯流程

```
tatic inline int
arm_exidx_step (struct cursor *c)
{
  unw_word_t old_ip, old_cfa;
  uint8_t buf[32];
  int ret;

  old_ip = c->dwarf.ip;
  old_cfa = c->dwarf.cfa;

  /* mark PC unsaved */
  c->dwarf.loc[UNW_ARM_R15] = DWARF_NULL_LOC;
  unw_word_t ip = c->dwarf.ip;
  if (c->dwarf.use_prev_instr)

```

```
    --ip;

  /* check dynamic info first --- it overrides everything else */
  ret = unwi_find_dynamic_proc_info (c->dwarf.as, ip, &c->dwarf.pi, 1,
                                     c->dwarf.as_arg);   //查找pc值对应的exidx entry
  if (ret == -UNW_ENOINFO)
    {
      if ((ret = tdep_find_proc_info (&c->dwarf, ip, 1)) < 0)
        return ret;
    }

  if (c->dwarf.pi.format != UNW_INFO_FORMAT_ARM_EXIDX)
    return -UNW_ENOINFO;

  ret = arm_exidx_extract (&c->dwarf, buf);  //获取exidx entry对应的字节码
  if (ret == -UNW_ESTOPUNWIND)
    return 0;
  else if (ret < 0)
    return ret;

  ret = arm_exidx_decode (buf, ret, &c->dwarf); //对字节码进行解析并进行回溯
  if (ret < 0)
    return ret;

  if (c->dwarf.ip == old_ip && c->dwarf.cfa == old_cfa)
    {
      Dprintf ("%s: ip and cfa unchanged; stopping here (ip=0x%lx)\n",
               __FUNCTION__, (long) c->dwarf.ip);
      return -UNW_EBADFRAME;
    }

  c->dwarf.pi_valid = 0;

  return (c->dwarf.ip == 0) ? 0 : 1;
}
```

函数unwi_find_dynamic_proc_info和函数tdep_find_proc_info最终都会调用到arm_search_unwind_table，该函数在exidx section中以二分法的方式找到一个与PC（ip）值相对应的项，作为后续栈回溯的项。

```
int
arm_search_unwind_table (unw_addr_space_t as, unw_word_t ip,
                         unw_dyn_info_t *di, unw_proc_info_t *pi,
                         int need_unwind_info, void *arg)
{
  /* The .ARM.exidx section contains a sorted list of key-value pairs -
     the unwind entries.  The 'key' is a prel31 offset to the start of a
     function.  We binary search this section in order to find the
     appropriate unwind entry.  */
  unw_word_t first = di->u.rti.table_data;
  unw_word_t last = di->u.rti.table_data + di->u.rti.table_len - 8;
  unw_word_t entry, val;

  if (prel31_to_addr (as, arg, first, &val) < 0 || ip < val)
    return -UNW_ENOINFO;

  if (prel31_to_addr (as, arg, last, &val) < 0)
    return -UNW_EINVAL;

  if (ip >= val)
    {
      entry = last;

      if (prel31_to_addr (as, arg, last, &pi->start_ip) < 0)
        return -UNW_EINVAL;

      pi->end_ip = di->end_ip -1;
    }
  else
```

```
    {
      while (first < last - 8)
        {
          entry = first + (((last - first) / 8 + 1) >> 1) * 8;

          if (prel31_to_addr (as, arg, entry, &val) < 0)
            return -UNW_EINVAL;

          if (ip < val)
            last = entry;
          else
            first = entry;
        }

      entry = first;

      if (prel31_to_addr (as, arg, entry, &pi->start_ip) < 0)
        return -UNW_EINVAL;

      if (prel31_to_addr (as, arg, entry + 8, &pi->end_ip) < 0)
        return -UNW_EINVAL;

      pi->end_ip--;
    }

  if (need_unwind_info)
    {
      pi->unwind_info_size = 8;
      pi->unwind_info = (void *) entry;
      pi->format = UNW_INFO_FORMAT_ARM_EXIDX;
    }
  return 0;
}
```

对于一个特定的平台，在对各个部件进行编译时最好使用同一个栈回溯方式，否则可以造成意想不到的错误。比如：当一个调用栈中包含有使用exidx和frame pointer两种栈回溯方式，当使用frame pointer回溯了几级函数之后，下一级函数使用的是exidx栈回溯方式，exidx栈回溯方式并不会对上一级的栈帧地址进行保存，当再次使用frame pointer方式进行回溯时，由于此时栈帧结构已经发生了断裂，并不能回溯到完整的调用栈还有可能访问到异常的地址造成abort。

此外，在通过如gdb等工具对程序进行跟踪或分析时，不仅会用到栈回溯功能还会使用到调试信息，调试信息存储在.debug_frame、.debug_info等section中，由编译参数-g产生，这些section提供更为具体的以供分析的信息，例如函数所在的文件、行号等信息。这些信息对于正常程序的运行并无作用，只是供调试使用，一般在release版本中不包含而在debug版本中包含。当在gdb中调用不包含有debug信息的函数时，有可能出现如下错误：

```
" xxx" has unknown return type; cast the call to its declared return type
```

这是由于gdb不在默认不包含debug信息的函数的返回参数为void型，而需要显示的指定调用的函数类型。官网解释如下：

```
GDB no longer assumes functions with no debug information return
  'int'.

  This means that GDB now refuses to call such functions unless you
  tell it the function's type, by either casting the call to the
  declared return type, or by casting the function to a function
  pointer of the right type, and calling that:
```

参考

- https://developer.arm.com/documentation/ihi0038/c/?lang=en

文章知识点与官方知识档案匹配，可进一步学习相关知识