代码中打印C++调用堆栈

AlexanderGreat 关注 IP属地: 山东

1 2019.07.22 23:30:48 字数 1,497 阅读 17,365

基本动因

有时在进行大型项目的开发时，我发现找出调用某些函数或方法的所有位置非常有用。而且，我不仅仅想要直接调用者，而是整个调用栈。这在两个场景中最有用：

- 在调试的时候
- 试图弄清楚某些代码如何工作的时候，即学习源代码的时候

一种可能的解决方案是使用调试器：在调试器中运行程序，在某个特定的地方放置断点，在停止时检查调用堆栈。虽然这种做法很有效并且有时非常有用，但我个人更喜欢代码化的方法：即编写一个专门用来打印调用堆栈的函数，以便在我感兴趣的每个地方打印出调用堆栈。然后我可以使用grep和更复杂的工具来分析调用日志，从而更好地理解某些代码的工作原理。

在这篇文章中，我想提出一个相对简单的方法来做到这一点。它主要针对Linux平台，但应该在其他Unix（包括OS X）上进行少量修改也可以达到相同的效果。

利用libunwind库来获取调用堆栈

我目前知道有三种靠谱且普遍的编程的方法来获取调用堆栈：

1. gcc编译器自带的宏：__builtin_return_address：这是一种非常粗糙，底层的方式。这个宏将获得堆栈上每个帧上函数的返回地址。 注意：只是地址，而不是函数名称。 因此需要额外的处理来获得函数名称。
2. glibc的backtrace和backtrace_symbols：可以获取调用堆栈上函数的实际符号名称。
3. 使用libunwind。

在三者之间，我非常喜欢libunwind库，因为它是最时髦，最广泛和最方便的解决方案。 它也比第二种方法的backtrace更灵活，可以够提供额外的信息，例如每个堆栈帧的CPU的寄存器值。

此外，在系统编程中，libunwind是最接近你现在可以获得的“官方词汇”。 例如，gcc可以使用libunwind实现零成本的C++异常捕捉（当实际抛出异常时需要堆栈展开）[^1]。大名鼎鼎的LLVM还

在libc++中重新实现了libunwind接口，该接口用于在基于此库的LLVM工具链中展开调用堆栈。

代码示例

这是一个使用libunwind库从代码中的任意点获取调用回溯的完整的代码示例。有关此处调用的API函数的更多详细信息，请参阅libunwind Document

```
#define UNW_LOCAL_ONLY
#include <libunwind.h>
#include <stdio.h>

// Call this function to get a backtrace.
void backtrace() {
  unw_cursor_t cursor;
  unw_context_t context;

  // Initialize cursor to current frame for local unwinding.
  unw_getcontext(&context);
  unw_init_local(&cursor, &context);

  // Unwind frames one by one, going up the frame stack.
  while (unw_step(&cursor) > 0) {
    unw_word_t offset, pc;
    unw_get_reg(&cursor, UNW_REG_IP, &pc);
    if (pc == 0) {
      break;
    }
    printf("0x%lx:", pc);

    char sym[256];
    if (unw_get_proc_name(&cursor, sym, sizeof(sym), &offset) == 0) {
      printf(" (%s+0x%lx)\n", sym, offset);
    } else {
      printf(" -- error: unable to obtain symbol name for this frame\n");
    }
  }
}

void foo() {
  backtrace(); // <-------- backtrace here!
}

void bar() {
  foo();
}

int main(int argc, char **argv) {
  bar();

  return 0;
}
```

libunwind很容易从源代码,或者直接从二进制包来安装。 我只是使用通常的configure，make和make install经典三部曲从源代码构建它并将其放入/usr/local/lib。

一旦你在编译器可以找到[2]的地方安装libunwind，则编译上面的代码：

```
gcc -o libunwind_backtrace -Wall -g libunwind_backtrace.c -lunwind
```

最后运行：

```
$ LD_LIBRARY_PATH=/usr/local/lib ./libunwind_backtrace
0x400958: (foo+0xe)
0x400968: (bar+0xe)
0x400983: (main+0x19)
0x7f6046b99ec5: (__libc_start_main+0xf5)
0x400779: (_start+0x29)
```

因此，我们在调用backtrace的位置获得完整的调用堆栈。 我们可以获得函数符号名称和调用指令的地址（更确切地说，返回地址是下一条指令）。

但是，有时我们不仅需要调用函数的名字，还需要调用函数的位置（即源文件名+行号）。 当一个函数从多个位置调用另一个函数并且我们想要确定哪一个是真正的给定调用堆栈的一部分时，这很有用。 libunwind为我们提供了调用地址，但没有任何内容。 幸运的是，它全部在二进制文件的DWARF信息中，并且给定地址我们可以通过多种方式提取确切的调用位置。 最简单的可能是调用addr2line命令：

```
$ addr2line 0x400968 -e libunwind_backtrace
libunwind_backtrace.c:37
```

我们将函数bar这一帧左侧的PC地址传递给addr2line并获取文件名和行号。

或者，我们可以使用pyelftools中的dwarf_decode_address示例来获取相同的信息：

```
$ python <path>/dwarf_decode_address.py 0x400968 libunwind_backtrace
Processing file: libunwind_backtrace
Function: bar
File: libunwind_backtrace.c
Line: 37
```

如果在backtrace调用期间打印出确切位置对你来说很重要，你还可以通过使用libdwarf打开可执行文件并在backtrace调用中从中读取此信息来达到目的。

C++ 和 mangled function names

上面的代码示例运行良好，但是现在C++比C代码使用得更广泛，因此这个方案存在一些问题。 在C++中，函数和方法的名称被mangled了，也就是被混淆了。 这个功能对于函数重载，命名空间和模板等C++功能起到至关重要。 假设实际的调用顺序是：

```
namespace ns {

template <typename T, typename U>
void foo(T t, U u) {
  backtrace(); // <-------- backtrace here!
}

}  // namespace ns

template <typename T>
struct Klass {
  T t;
  void bar() {
    ns::foo(t, true);
  }
};

int main(int argc, char** argv) {
  Klass<double> k;
  k.bar();

  return 0;
}
```

实际的调用堆栈是：

```
0x400b3d: (_ZN2ns3fooIdbEEvT_T0_+0x17)
0x400b24: (_ZN5KlassIdE3barEv+0x26)
0x400af6: (main+0x1b)
0x7fc02c0c4ec5: (__libc_start_main+0xf5)
0x4008b9: (_start+0x29)
```

看起来好像不是太美好。当然C++老鸟还是能够从轻微混乱的名字中找到蛛丝马迹（就像系统程序员能够从ascill十六进制码认出文本），当代码大面积使用模板，函数名称将变得更加不堪入目。

有一个方案是使用命令行工具c++filt：

```
$ c++filt _ZN2ns3fooIdbEEvT_T0_
void ns::foo<double, bool>(double, bool)
```

但是，如果我们的调用堆栈转储器直接打印出去未混淆的名称会更好。 幸运的是，这很容易做到，使用cxxabi.h API函数，它是libstdc++的一部分（更确切地说，libsupc++）。 libc++还在底层libc++ abi中提供它。 我们需要做的就是调用abi::__cxa_demangle。 下面一个完整的例子：

```
#define UNW_LOCAL_ONLY
#include <cxxabi.h>
#include <libunwind.h>
#include <cstdio>
#include <cstdlib>

void backtrace() {
  unw_cursor_t cursor;
  unw_context_t context;

  // Initialize cursor to current frame for local unwinding.
  unw_getcontext(&context);
  unw_init_local(&cursor, &context);

  // Unwind frames one by one, going up the frame stack.
  while (unw_step(&cursor) > 0) {
    unw_word_t offset, pc;
    unw_get_reg(&cursor, UNW_REG_IP, &pc);
    if (pc == 0) {
      break;
    }
    std::printf("0x%lx:", pc);

    char sym[256];
    if (unw_get_proc_name(&cursor, sym, sizeof(sym), &offset) == 0) {
      char* nameptr = sym;
      int status;
      char* demangled = abi::__cxa_demangle(sym, nullptr, nullptr, &status);
      if (status == 0) {
        nameptr = demangled;
      }
      std::printf(" (%s+0x%lx)\n", nameptr, offset);
      std::free(demangled);
    } else {
      std::printf(" -- error: unable to obtain symbol name for this frame\n");
    }
  }
}

namespace ns {

template <typename T, typename U>
void foo(T t, U u) {
  backtrace(); // <-------- backtrace here!
}
```

```
} // namespace ns

template <typename T>
struct Klass {
  T t;
  void bar() {
    ns::foo(t, true);
  }
};

int main(int argc, char** argv) {
  Klass<double> k;
  k.bar();

  return 0;
}
```

这一次，所有未混淆的函数名称全被打印出来了：

```
$ LD_LIBRARY_PATH=/usr/local/lib ./libunwind_backtrace_demangle
0x400b59: (void ns::foo<double, bool>(double, bool)+0x17)
0x400b40: (Klass<double>::bar()+0x26)
0x400b12: (main+0x1b)
0x7f6337475ec5: (__libc_start_main+0xf5)
0x4008b9: (_start+0x29)
```

参考

Programmatic access to the call stack in C++