llvm-symbolizer解析logcat堆栈信息

原创 尹鹏帅 YPS技术闲谈 2023年12月23日 10:12 上海

在使用Unity引擎开发的游戏项目中，大多数时间开发者都是和C#语言打交道。在Unity 5.0开始支持IL2cpp运行时后，C#不在运行在Mono解释器上。而是以转换后的C++代码执行。另外，Unity引擎底层本身也是C++开发的，C#是对引擎C++接口的封装，更加便于游戏开发。所以说，开发Unity游戏和C++关系紧密。

C++程序crash问题是其相对于脚本语言复杂的地方。脚本语言，如python等，都可以很方便的打印出crash的堆栈信息。信息里会包含类名、方法名、文件路径和行号等。这些信息对定位问题的原因很有帮助。但是C++的crash信息则“抽象”很多。不过，通常C#中的异常都已经被Untiy很好的处理了，很难触发Crash。

在Android系统中，可以从logcat中看到crash的信息。这是Android系统提供的功能。一眼看去，crash信息提供的调用栈和Crash的原因描述。但是细看调用栈却只是模块名和地址信息。显然，我们需要一些方法来处理下这些“原始”的信息才能进一步定位问题的原因。

```
backtrace:
  #00  pc 0000000000042f89  libexample.so
  #01  pc 0000000000000640  libexample.so
  #02  pc 0000000000065a3b  /system/lib/libc.so
  #03  pc 000000000001e4fd  /system/lib/libc.so
```

C++的堆栈信息如此简陋，是因为通常执行文件不包含可读的函数符号信息。之所以这样做，有如下几个原因: - 减少执行文件大小。符号文件的文件空间占用很大，甚至远大于执行文件的大小。- 安全考虑。剥离符号文件，可以增加逆向分析的难度。

在unity打包apk时会生成独立的符号文件。我们需要使用专用的工具来使用符号文件。google文档中提到使用addr2line来翻译crash的符号信息。在NDK中有提供addr2line。

本文则会讲述，如何使用llvm-symbolizer翻译符号信息。llvm-symbolizer是llvm项目的子工程。现在android 已经转用clang编译器。clang也是llvm项目的子工程。因此，有了这篇文章的想法。

在介绍使用llvm-symbolizer之前，先思考下符号文件是如何管理符号信的。符号文件有两种标准pdb和dwarf https://dwarfstd.org 。pdb是VC++使用的格式。可以说是Windows专用。dwarf是Linux&Unix使用的格式，也适用于android和iOS。

如果让我们来设计文件的格式。朴素的想法应该是，记录每个函数的地址信息。然后根据crash堆栈中的地址信息来查找对应的函数。dwarf标准确实如此。但是有个问题需要解决，那就是同一个函数每次运行时的堆栈地址是会变化的。这是操作系统使用地址空间内存随机化技术导致的。那么，我们应该在符号文件中持久化存储什么地址呢？如果稍微了解下动态加载技术，就很容易想到方案，那就是存储符号在代码模块文件中的相对位置（称为文件地址）。这样，只要能从堆栈中的地址计算出文件地址，即可查询到对应的符号信息。

事实上，dwarf是设计用来进行源码级调试的工具。方法符号信息只是其内容的一部分。其还包含模块，变量，类型等用于调试的信息。dwarf以DIE描述块平铺的方式来描述代码的信息。其存储格式是二进制的方式。假如用文本化的方式来描述，其形式如下。

```
1:  int a;
2:  void foo()
3:  {
4:      register int b;
5:      int c;
6:  }

<1>:    DW_TAG_subprogram
          DW_AT_name = foo
<2>:    DW_TAG_variable
          DW_AT_name = b
          DW_AT_type = <4>
          DW_AT_location = (DW_OP_reg0)
<3>:    DW_TAG_variable
          DW_AT_name = c
          DW_AT_type = <4>
          DW_AT_location =
                (DW_OP_fbreg: -12)
<4>:    DW_TAG_base_type
          DW_AT_name = int
          DW_AT_byte_size = 4
          DW_AT_encoding = signed
<5>:    DW_TAG_variable
          DW_AT_name = a
          DW_AT_type = <4>
          DW_AT_external = 1
          DW_AT_location = (DW_OP_addr: 0
```

可以看到，示例代码的信息，几乎都有对应的描述。更专业的介绍，可以查看官方提供的文档。

sevaa在github上提供了可视化查看dwarf符号文件的工具dwex sevaa/dwex: DWARF Explorer - a GUI utility for navigating the DWARF debug information (github.com)，可用于学习和排查问题。

本文将关注其中的函数符号地址相关的细节。

我们构造一个crash的例子。这个例子里，构造了一个非法的指针，在尝试使用该指针时会调用Char.ToString方法导致App Crash。这里使用unsafe代码，更方便让Unity崩溃。

```
void Crash()
{
    var testStr = "TestString";
    unsafe
    {
        char* c = (char*)(IntPtr)1;
        testStr += *c;
    }
}
```

启动之后，app会crash。从logcat中提取堆栈信息，简化后如下：

```
backtrace:
#00 pc 000000000034f584  libil2cpp.so
#01 pc 0000000000277a04  libil2cpp.so
#02 pc 0000000000163b54  libil2cpp.so
#03 pc 00000000001639b4  libil2cpp.so
#04 pc 00000000001f3078  libunity.so
#05 pc 00000000001ff080  libunity.so
#06 pc 000000000020c4a4  libunity.so
```

那么第#01帧，应该就是Crash函数内的调用Char.ToString处的地址。地址是0x0000000000277a04。

使用dwex查看Crash函数的符号信息：

Attribute	Form	Value
low_pc	addr	0x2779bc
high_pc	data4	104
frame_base	exprloc	sp
name	strp	TestDemo_Crash_mF1EBDDA46E000EAFDABAD2763CEBE217D13457D3
decl_file	data1	1: Assembly-CSharp.cpp
decl_line	data1	209
external	flag_present	True

low_pc表示函数的起始地址。high_pc表示函数的结束地址（这里用的相对low_pc的偏移表示）。计算一下，可以得到high_pc地址为0x277a24。那么函数Crash的范围就是0x2779bc - 0x277a24。回头看logcat中得到的地址是0x0000000000277a04，正好对应起来。因此通过计算查询dwarf符号文件中的函数信息，就能确定堆栈对应的函数。

不过要说明的是，logcat中的地址信息并不是运行时虚拟地址，而是logcat自动处理后的文件地址，不再需要手动处理了。对于动态库来说，会有加载基地址的概念，动态库内的地址引用都是相对于基地址的。使用llvm-objdump.exe -h libil2cpp.so查看模块的VMA地址。

```
Sections:
Idx Name                 Size     VMA              Type
    il2cpp               002aae38 00000000002779b8 TEXT
```

在操作系统最初始的设计中，VMA就是模块被加载后的虚拟内存地址。但是，现代操作系统为了提高安全性，采用了地址空间内存随机化（ASLR）技术。就会导致模块实际加载的地址，并不是符号文件中查看到的VMA地址。要想知道实际的运行时加载地址，必须在进程执行状态下才能获取。幸运的是，在crash时logcat帮我们处理地址转换，直接显示的是文件地址。

现在可以看下如何使用llvm-symbolizer.exe了。阅读官方文档llvm-symbolizer - convert addresses into source code locations — LLVM 18.0.0git documentation,可以看出其基本的用法是

```
llvm-symbolizer [options] [addresses…]
```

但是并没有说明addresses是什么地址。查找资料后确认，这里的地址参数要求的是文件地址。因此，可以直接使用logcat中的地址。使用如下命令：

```
llvm-symbolizer.exe -e libil2cpp.so 0x000000000034f584 0x0000000000277a04
```

得到如下结果：

```
Char_ToString_m2A308731F9577C06AF3C0901234E2EAC8327410C
UnityDemo/Library/Bee/artifacts/Android/il2cppOutput/cpp\mscorlib1.cpp:22175:17

TestDemo_Crash_mF1EBDDA46E000EAFDABAD2763CEBE217D13457D3
UnityDemo/Library/Bee/artifacts/Android/il2cppOutput/cpp\Assembly-CSharp.cpp:229:9
```

这样我们就可以定位到问题的原因了。