C++ 的全链路追踪方案，稍微有点高端

CPP开发者 2022-02-17 11:50

以下文章来源于程序喵大人 ，作者程序喵大人

程序喵大人

C++，Go、Java、Android、iOS、音视频、深度学习不定期更新。

背景：笔者主要在做C++ SDK的开发，需要给到业务端去集成，在集成的过程中可能会出现某些功能性bug，即没有得到想要的结果。那怎么调试？

分析：这种问题其实调试起来稍微有点困难，它不像crash，当发生crash时还能拿到堆栈信息去分析，然而功能性bug没有crash，也就没法捕捉对应到当时的堆栈信息。因为不是在本地，也没法用编译器debug。那思路就剩log了，一种方式是考虑在SDK内部的关键路径下打印详细的log，当出现问题时拿到log去分析。然而总有漏的时候，谁能保证log一定打的很全面，很有可能问题就出现在没有log的函数中。

解决：基于上面的背景和问题分析，考虑是否能做一个全链路追踪的方案，把打印出整个SDK的调用路径，从哪个函数进入，从哪个函数退出等。

想法1：可以考虑在SDK的每个接口都加一个context结构体参数，记录下来函数的调用路径，这可能是比较通用有效的方案，但是SDK接口已经固定了，更改接口要面临的困难很大，业务端基本不会同意，所以这种方案不适合我们现有情况，当然一个从0开始建设的中间件和SDK可以考虑考虑。

想法2：有没有一种不用改接口，还能追踪到函数调用路径的方案？

继续沿着这个思路继续调研，我找到了gcc和clang编译器的一个编译参数：-finstrument-functions，编译时添加此参数会在函数的入口和出口处触发一个固定的回调函数，即：

```
1 __cyg_profile_func_enter(void *callee, void *caller);
2 __cyg_profile_func_exit(void *callee, void *caller);
```

参数就是callee和caller的地址，那怎么将地址解析成对应函数名？可以使用dladdr函数：

```
int dladdr(const void *addr, Dl_info *info);
```

看下下面的代码：

```
// tracing.cc

#include <cxxabi.h>
#include <dlfcn.h>  // for dladdr
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef NO_INSTRUMENT
#define NO_INSTRUMENT __attribute__((no_instrument_function))
#endif

extern "C" __attribute__((no_instrument_function)) void __cyg_profile_fu
    Dl_info info;
    if (dladdr(callee, &info)) {
        int status;
        const char *name;
        char *demangled = abi::__cxa_demangle(info.dli_sname, NULL, 0, &
        if (status == 0) {
            name = demangled ? demangled : "[not demangled]";
        } else {
            name = info.dli_sname ? info.dli_sname : "[no dli_sname nd s
        }

        printf("enter %s (%s)\n", name, info.dli_fname);

        if (demangled) {
            free(demangled);
            demangled = NULL;
        }
    }
}

```

```
extern "C" __attribute__((no_instrument_function)) void __cyg_profile_fu
    Dl_info info;
    if (dladdr(callee, &info)) {
        int status;
        const char *name;
        char *demangled = abi::__cxa_demangle(info.dli_sname, NULL, 0, &
        if (status == 0) {
            name = demangled ? demangled : "[not demangled]";
        } else {
            name = info.dli_sname ? info.dli_sname : "[no dli_sname and
        }
        printf("exit %s (%s)\n", name, info.dli_fname);

        if (demangled) {
            free((void *)demangled);
            demangled = NULL;
        }
    }
}
```

这是测试文件：

```
// test_trace.cc
void func1() {}

void func() { func1(); }

int main() { func(); }
将test_trace.cc和tracing.cc文件同时编译链接，即可达到链路追踪的目的：
g++ test_trace.cc tracing.cc -std=c++14 -finstrument-functions -rdynamic
输出：enter main (./a.out)
enter func() (./a.out)
enter func1() (./a.out)
exit func1() (./a.out)
exit func() (./a.out)
exit main (./a.out)
```

如果在func()中调用了一些其他的函数呢？

```cpp
#include <iostream>
#include <vector>

void func1() {}

void func() {
    std::vector<int> v{1, 2, 3};
    std::cout << v.size();
    func1();
}

int main() { func(); }
```

再重新编译后输出会是这样：

```
enter [no dli_sname nd std] (./a.out)
enter [no dli_sname nd std] (./a.out)
exit [no dli_sname and std] (./a.out)
exit [no dli_sname and std] (./a.out)
enter main (./a.out)
enter func() (./a.out)
enter std::allocator<int>::allocator() (./a.out)
enter __gnu_cxx::new_allocator<int>::new_allocator() (./a.out)
exit __gnu_cxx::new_allocator<int>::new_allocator() (./a.out)
exit std::allocator<int>::allocator() (./a.out)
enter std::vector<int, std::allocator<int> >::vector(std::initializer_li
enter std::_Vector_base<int, std::allocator<int> >::_Vector_base(std::al
enter std::_Vector_base<int, std::allocator<int> >::_Vector_impl::_Vecto
enter std::allocator<int>::allocator(std::allocator<int> const&) (./a.ou
enter __gnu_cxx::new_allocator<int>::new_allocator(__gnu_cxx::new_alloca
exit __gnu_cxx::new_allocator<int>::new_allocator(__gnu_cxx::new_allocat
exit std::allocator<int>::allocator(std::allocator<int> const&) (./a.out
exit std::_Vector_base<int, std::allocator<int> >::_Vector_impl::_Vector
exit std::_Vector_base<int, std::allocator<int> >::_Vector_base(std::all
```

上面我只贴出了部分信息，这显然不是我们想要的，我们只想要显示自定义的函数调用路径，其他的都想要过滤掉，怎么办？

这里可以将自定义的函数都加一个统一的前缀，在打印时只打印含有前缀的符号，这种个人认为是比较通用的方案。

下面是我过滤掉std和gnu子串的代码：

```
if (!strcasestr(name, "std") && !strcasestr(name, "gnu")) {
    printf("enter %s (%s)\n", name, info.dli_fname);
}

if (!strcasestr(name, "std") && !strcasestr(name, "gnu")) {
    printf("exit %s (%s)\n", name, info.dli_fname);
}
```

重新编译后就会输出我想要的结果：

```
g++ test_trace.cc tracing.cc -std=c++14 -finstrument-functions -rdynamic
输出：enter main (./a.out)
enter func() (./a.out)
enter func1() (./a.out)
exit func1() (./a.out)
exit func() (./a.out)
exit main (./a.out)
```

还有一种方式是在编译时使用下面的参数：

```
-finstrument-functions-exclude-file-list
```

它可以排除不想要做trace的文件，但是这个参数只在gcc中可用，在clang中却不支持 ，所以上面的字符串过滤方式更通用一些。

上面只能拿到函数的名字，不能定位到具体的文件和行号，如果想要获得更多信息，需要结合bfd系列参数(bfd_find_nearest_line)和libunwind一起使用，大家可以继续研究。。。

tips1：这是一篇抛砖引玉的文章，笔者不是后端开发，据我所知后端C++中有很多成熟的trace方案，大家有更好的方案可以留言，分享一波。

tips2：上面的方案可以达到链路追踪的目的，但最后没有应用到项目中，因为在做的项目对性能要求较高，使用此种方案会使整个SDK性能下降严重，无法满足需求正常运行。于是暂时放弃了链路追踪的这个想法。

本文的知识点还是值得了解一下的，大家或许会用得到。在研究的过程中我也发现了一个基于此种方案的开源项目（call-stack-logger），感兴趣的也可以去了解了解。

- EOF -

推荐阅读 — 点击标题可跳转

1、深度思考：TCP 协议存在那些缺陷？

2、2022 技术趋势：C++、Go、Rust 大放异彩

3、C++ 反射 TS 初探