coredump 调试

整体思路

- coredump 产生原因：

分类	原因	备注
机器	硬件故障(SIGBUG、SIGEMT、SIGIOT)	根据退出信号量进行排查
资源	内存超限	云部署容器内存限制
	栈空间超过设置大小	pthread(ulimit -s) bthread(--stack_size_normal)
	线程超限	检查系统的线程数限制
程序BUG	Assert问题	常见于初始化代码、服务异常断言等
	内存问题	数组越界、空指针、类型强转等
	栈问题	内存越界写坏栈状态信息；栈空间超限等
	并发问题	多线程操作同一段内存空间，非线程安全
	程序指令错乱	一般由堆栈写坏触发，如返回地址错误、虚函数表错误等
	退出问题	析构顺序依赖；线程未主动join等

1.明确 core 的大致触发原因。机器问题？自身程序问题？
2.定位代码行。哪一行代码出现了问题 (bt)
3.定位执行指令。哪一行指令干了什么事
4.定位异常变量。指令不会有问题，是指令操作的变量不符合

- 常见误区：

1. 通过空指针调用对象方法一定崩溃吗？

不一定崩溃。如果成员函数是实函数，又没有直接或间接访问成员变量，则不会发生崩溃。这种情况下，普通成员函数与静态成员函数类似

```
class D
{
public:
    void printA()
    {
        cout<<"printA"<<endl;
    }
    virtual void printB()
    {
        cout<<"printB"<<endl;
    }
};
int main(void)
{
    D *d=NULL;
    d->printA();
    d->printB();
}
```

输出 “printA” 后，程序崩溃。为什么呢？printA 是成员函数，存放在代码段（.text），所以没有实例化类的时候仍然可以调用。printB 是虚函数，关系到虚函数表和虚函数指针，虚函数指针存放在实例化的对象中，所以，未实例化对象时，不存在虚函数指针，所以调用虚函数会报错。一般找虚函数指针都是通过 this 指针地址+偏移来计算的，this 本身是空，算出来虚拟函数指针肯定有问题，只要访问就会挂。

2. 通过野指针调用对象方法一定崩溃吗？

不一定崩溃。取决于对象的内存是否被重新分配、是否被覆写、是否访问成员变量、是否为虚函数等。可能不立即崩溃但误操作内存数据，导致程序后续运行逻辑异常或 crash，即埋下一颗地雷

3. 内存不足 malloc 一定返回空指针吗？

不一定。涉及内存分配的 overcommit 问题：https://www.kernel.org/doc/Documentation/vm/overcommit-accounting。（linux 操作系统有特定的内存管理方式。其中一项策略是 Overcommit，它允许应用程序提前预订所需的内存。然而，承诺的内存在实际使用时可能无法使用）

C++ new 更复杂一些。开启 exception 的情况下，内存分配失败可能 throw std::bad_alloc，不返回空指针。可以通过 new(std::nothrow) 让 new 不抛出异常，例如：

```
void test() {
      try {
          int *p = new int[1ULL << 50U];
          std::cout << p << '\n';
          delete[] p;
      } catch (const std::bad_alloc &e) {
          std::cout << e.what() << '\n';
      }

      int *p = new(std::nothrow) int[1ULL << 50U];
      if (p == nullptr) {
          std::cout << "Allocation returned nullptr\n";
      }
      delete[] p;
  }
```

4. free(NULL) 和 delete nullptr 都是安全的，是否判断非空指针再 delete 是代码风格问题。

辅助工具：

1. asan （应对无规律 core）

   https://github.com/google/sanitizers

编译选项：

QMAKE_CFLAGS += -fsanitize=address -fno-omit-frame-pointer
QMAKE_CXXFLAGS += -fsanitize=address -fno-omit-frame-pointer
QMAKE_LFLAGS += -fsanitize=address

运行时环境变量设置：

export ASAN_OPTIONS=symbolize=1:abort_on_error=1:disable_coredump=0:unmap_shadow_on_exit=1:detect_container_overflow=0:log_path=/mnt/mtd/configInfo/Config/asan.log

需要加载运行时 so ： libasan.so

asan 原理：(Google ASan 内存诊断工具简单讨论与分析 - 知乎)

1. 内存操作进行插桩:对 new,malloc,delete,free,memcpy,其它内存访问等操作进行**编译时替换与代码插入**，是编译器完成的；

2. 内存映射与诊断：按照一定的算法对原始内存进行一分影子内存的拷贝生成，目前不是 1：1 的拷贝，而是巧妙的按 1/8 大小进行处理，并进行一定的下毒与标记，减少内存的浪费。正常访问内存前，先对影子内存进行检查访问，如果发现数据不对，就进行诊断报错处理。

普通内存读写代码

```
*address = ...; // 写内存
... = *address; // 读内存
```

插桩后伪码：

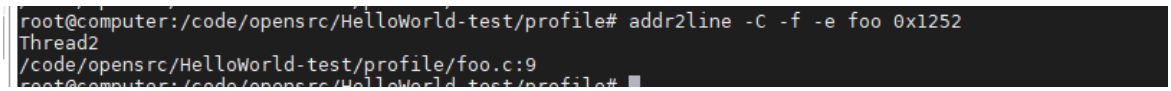
```
if (IsPoisoned(address))
{
  ReportError(address, kAccessSize, kIsWrite);
}
*address = ...; // 写内存
... = *address; // 读内存
```

tsan：

```
root@computer:/code/opensrc/HelloWorld-test/profile#
root@computer:/code/opensrc/HelloWorld-test/profile# gcc -fsanitize=thread -o foo foo.c -g -O0
root@computer:/code/opensrc/HelloWorld-test/profile#
root@computer:/code/opensrc/HelloWorld-test/profile# ./foo
==================
WARNING: ThreadSanitizer: data race (pid=641238)
  Read of size 4 at 0x5556fc819014 by thread T2:
    #0 Thread2 /code/opensrc/HelloWorld-test/profile/foo.c:9 (foo+0x1252)

  Previous write of size 4 at 0x5556fc819014 by thread T1:
    #0 Thread1 /code/opensrc/HelloWorld-test/profile/foo.c:5 (foo+0x1213)

  Location is global 'Global' of size 4 at 0x5556fc819014 (foo+0x000000004014)

  Thread T2 (tid=641241, running) created by main thread at:
    #0 pthread_create ../../../../src/libsanitizer/tsan/tsan_interceptors_posix.cpp:962 (libtsan.so.0+0x5ea79)
    #1 main /code/opensrc/HelloWorld-test/profile/foo.c:15 (foo+0x12e3)

  Thread T1 (tid=641240, finished) created by main thread at:
    #0 pthread_create ../../../../src/libsanitizer/tsan/tsan_interceptors_posix.cpp:962 (libtsan.so.0+0x5ea79)
    #1 main /code/opensrc/HelloWorld-test/profile/foo.c:14 (foo+0x12c2)

SUMMARY: ThreadSanitizer: data race /code/opensrc/HelloWorld-test/profile/foo.c:9 in Thread2
==================
ThreadSanitizer: reported 1 warnings
```

2. addr2line

初步查看 崩溃代码点



3. 打印 STL 容器

   stl-views.gdb（链接：attachment:stl-views-1.0.3.gdb of STLSupport - GDB Wiki

4. python 脚本交互

   你还在用 GDB 调试程序吗？ - 知乎

5. Core-Analyzer

   Core Analyzer Integrated with Debuggers

调试小技巧

1. 栈回溯

   ARM 汇编入门指南 - 知乎

   消失的调用栈帧-基于 fp 的栈回溯原理解析

2. 导出内存数据

```
dump binary memory file address address + length
```

3. **打印指令（x、print、display）**

```
x /<n/f/u>  <addr>
n:是正整数，表示需要显示的内存单元的个数，即从当前地址向后显示n个内存单元的内容，
一个内存单元的大小由第三个参数u定义。

f:表示addr指向的内存内容的输出格式，s对应输出字符串，此处需特别注意输出整型数据的
格式：
  x 按十六进制格式显示变量.
  d 按十进制格式显示变量。
  u 按十进制格式显示无符号整型。
  o 按八进制格式显示变量。
  t 按二进制格式显示变量。
  a 按十六进制格式显示变量。
  c 按字符格式显示变量。
  f 按浮点数格式显示变量。

u:就是指以多少个字节作为一个内存单元-unit,默认为4。u还可以用被一些字符表示：
  如b=1 byte, h=2 bytes,w=4 bytes,g=8 bytes.

<addr>:表示内存地址。
```

4. set print object on

```
(gdb) set print object off
(gdb) p CmdProcInfo.pCmdProcObject
$3 = (CCmdProcObject *) 0x37be4930
(gdb) set print object on
(gdb) p CmdProcInfo.pCmdProcObject
$4 = (CCmdProcUpdateCfgClient *) 0x37be4930
(gdb) echo is a shell builtin
```

5. 锁调试 分析

6. 导出整个进程当前的内存快照

   gdb -p $1 -batch -eval-command='gcore /mnt/usb/memory_dump'

更进一步的方向（待探索）

1. 使用 hwasan

   asan 内存消耗还是比较高，在实际的生产环境无法使用，只能作为定位问题的手段。

   [深入分析 HWASAN 检测内存错误原理]

2. 使用 rust

新工程可以使用 rust 来写，rust 可以自动化导出 C 语言接口，跟其他编程语言进行交互

总结

coredump 调试是经验性的积累，需要长久使用 gdb，善于利用**汇编指令**可以更有效的定位 Core