Core，又称之为Core Dump文件，是Unix/Linux操作系统的一种机制，对于线上服务而言，Core令人闻之色变，因为出Core的过程意味着服务暂时不能正常响应，需要恢复，并且随着吐Core进程的内存空间越大，此过程可能持续很长一段时间（例如当进程占用60G+以上内存时，完整Core文件需要15分钟才能完全写到磁盘上），这期间产生的流量损失，不可估量。

凡事皆有两面性，OS在出Core的同时，虽然会终止掉当前进程，但是也会保留下第一手的现场数据，OS仿佛是一架被按下快门的相机，而照片就是产出的Core文件。里面含有当进程被终止时内存、CPU寄存器等信息，可以供后续开发人员进行调试。

coredump是程序崩溃时的内存快照。

操作系统在程序发生异常而异常在进程内部又没有被捕获的情况下，会把进程此刻内存、寄存器状态、运行堆栈等信息转储保存在一个文件里。下coredmp包含了程序运行时的内存，寄存器状态，堆栈指针，内存管理信息等。可以理解为把程序工作的当前状态存储成一个文件。许多程序和操作系统出错时会自动生成一个core文件。

关于Core产生的原因很多，比如过去一些Unix的版本不支持现代Linux上这种GDB直接附着到进程上进行调试的机制，需要先向进程发送终止信号，然后用工具阅读core文件。在Linux上，我们就可以使用kill向一个指定的进程发送信号或者使用gcore命令来使其主动出Core并退出。如果从浅层次的原因上来讲，出Core意味着当前进程存在BUG，需要程序员修复。从深层次的原因上讲，是当前进程触犯了某些OS层级的保护机制，逼迫OS向当前进程发送诸如SIGSEGV(即signal 11)之类的信号, 例如访问空指针或数组越界出Core，实际上是触犯了OS的内存管理，访问了非当前进程的内存空间，OS需要通过出Core来进行警示，这就好像一个人身体内存在病毒，免疫系统就会通过发热来警示，并导致人体发烧是一个道理（有意思的是，并不是每次数组越界都会出Core，这和OS的内存管理中虚拟页面分配大小和边界有关，即使不出Core，也很有可能读到脏数据，引起后续程序行为紊乱，这是一种很难追查的BUG）。

了解了这些之后，我们来看看如何阅读Core文件，并从中追查BUG。在Linux下，一般读取Core的命令为：

gdb exec\_file core\_file

使用GDB，先从可执行文件中读取符号表信息，然后读取Core文件。如果不与可执行文件搅合在一起可以吗？答案是不行，因为Core文件中没有符号表信息，无法进行调试，可以使用如下命令来验证：

Objdump –x core\_file | tail

我们看到如下两行信息：

SYMBOL TABLE:

no symbols

表明当前的ELF格式文件中没有符号表信息。

<https://blog.51cto.com/laodou/2431858>

$ulimit -c  可以查看是否打开此选项，若为0则为关闭；

　　ulimit -c 0可手动关闭

打开core文件生成：

　　$ulimit -c unlimited设置core文件大小为不限制大小；

检查core文件的选项是否打开：

　　$ulimit -a

对 core 文件的操作仅对当前生效，若需要永久生效，则要将相应操作写入 /etc/profile

生成路径

core 文件默认生成在程序的工作目录，可以对生成路径进行设置，需要保证对对应目录有足够空间并具有写权限

1

echo /MyCoreDumpDir/core.%e.%p > /proc/sys/kernel/core\_pattern

其中命名使用的参数列

%p - insert pid into filename # 添加 pid

%u - insert current uid into filename # 添加当前 uid

%g - insert current gid into filename # 添加当前 gid

%s - insert signal that caused the coredump into the filename # 添加导致产生 core 的信号

%t - insert UNIX time that the coredump occurred into filename # 添加 core 文件生成时的 unix 时间

%h - insert hostname where the coredump happened into filename # 添加主机名

%e - insert coredumping executable name into filename # 添加命令名

/proc/sys/kernel/core\_uses\_pid 这个文件的值若为１，则无论是否配置 %p ,最后生成的 core 文件都会添加 pid

可以使用 gdb 对 core 文件进行调试，编译是需要带上 -g 选项

如需要在 PC 上调试嵌入式设备产生的 core 文件，则需要选取相应平台的 gdb 工具，并在进入 gdb 后设置符号文件的位置

$使用GDB调试

第一种方法（推荐）：

    1）启动gdb，进入core文件，命令格式：gdb [exec file] [core file]，用法示例：

Gdb ./test core.xxxxx

    2）在进入gdb后，查找段错误位置：where或者bt，用法示例：

bt

#0 0x00007f205b7afde5 in \_IO\_vfscanf\_internal (s=<optimized out>, format=<optimized out>, argptr=argptr@entry=0x7ffdf417be88,

errp=errp@entry=0x0) at vfscanf.c:1902

#1 0x00007f205b7ba87b in \_\_scanf (format=<optimized out>) at scanf.c:33

#2 0x0000000000400589 in core\_test1 () at core.cpp:5

#3 0x00000000004005bf in main () at core.cpp:15

第二种方法：

    1）启动gdb，进入core文件，命令格式：

gdb -c [core file] //或 gdb --core=[core file]

（2）在进入gdb后，指定core文件对应的符号表，命令格式：

(gdb) file [exec file]

（3）查找段错误位置：where或者bt。用法示例：

 bt

**（2）永久性设置：**

如果你想持久化这些操作，可以在/etc/sysctl.conf文件中增加： kernel.core\_pattern=/corefiles/core-e%-%t

加好后，如果你想不重启看看效果的话，则用下面的命令：sysctl -p /etc/sysctl.conf

在下列条件下不产生core文件：

( a )进程是设置-用户-ID，而且当前用户并非程序文件的所有者；

( b )进程是设置-组-ID，而且当前用户并非该程序文件的组所有者；

( c )用户没有写当前工作目录的许可权；

( d )文件太大。core文件的许可权(假定该文件在此之前并不存在)通常是用户读/写，组读和其他读。

ulimit -c 可以设置core文件的大小，如果这个值为0.则不会产生core文件，这个值太小，则core文件也不会产生，因为core文件一般都比较大。

存储位置：

    core文件默认的存储位置与对应的可执行程序在同一目录下，文件名是core，可以通过下面的命令看到core文件的存在位置：

cat /proc/sys/kernel/core\_pattern # 缺省值是|/usr/share/apport/apport %p %s %c %P

注意：这里是指在进程当前工作目录的下创建。通常与程序在相同的路径下。但如果程序中调用了chdir函数，则有可能改变了当前工作目录。这时core文件创建在chdir指定的路径下。有好多程序崩溃了，我们却找不到core文件放在什么位置。和chdir函数就有关系。当然程序崩溃了不一定都产生 core文件。

为什么生成的core 文件大小为0字节

1. ulimit -c 是否为0

2. 执行的程序是否在linux和windows的共享目录里