**GCC 各工具使用简介**

(2013-06-17 21:23:56)

[*[http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif](javascript:;)转载▼*](javascript:;)

|  |  |
| --- | --- |
|  |  |

**GCC：**GNU开发的程序编译器

**GNU：**“GNU‘s Not Unix”，最初是为了实现一个类似unix的自由操作系统，感觉现在已经通常泛指遵循GPL自由软件精神的组织。

**GPL：**GNU通用公共许可证（GNU General Public License） ，简单的说就是遵循GPL的代码任意用户可以复制发布；使用或者修改了GPL的代码也必须遵循GPL精神；遵循GPL的代码已源代码发布；GPL 并不排斥对自由软件进行商业性质的包装和发行，也不限制在自由软件的基础上打包发行其他非自由软件。（所以在产品中要小心使用GPL的软件，否则将涉及到一些商业秘密）

**交叉编译：**

     在嵌入式领域，目标运行平台通常使用的是只能完成某部分特定功能的CPU，而为了提供开发和编译连接的效率，通常开发、编译、链接是发生在功能更强大的PC上。

     但是，目标板上的CPU通常和编译、链接的PC上的CPU属于不同的体系架构。PC上通常是inter或者是AMD的CPU，都属于X86的 体系架构；但是嵌入式的CPU却可能是如ARM、MIPS、PowerPC、SH、alpha等。不同的体系架构最直观的区别就是指令集的不同。

    所以，需要通过运行在PC上的工具链将目标板上的程序编译成目标板的指令集，这个工具链就叫交叉工具链；编译过程就叫交叉编译

**交叉编译工具：**

常见的误区：GCC只是linux下的编译工具。实际上我们在STB里面接触到的所有方案的编译工具，都是移植或者与GCC的功能大致相同的，其与目标平台所使用的操作系统没有关系。 比如st平台使用的操作系统是OS21，MSD平台使用的是ECOS，但是他们的编译工具集的基本功能都是一样的。

       交叉工具链GCC的任意工具在名字前都带有该体系架构的交叉编译前缀。如MSD5043平台属于MIPS的架构，其前缀是mipsisa32-elf- ；st平台是时候sh4架构的，其前缀是sh-superh-elf-;nxp24507平台属于ARM的架构，其前缀是arm-linux-uclibcgnueabi-

下面是我在pnx8473平台工具链中看到的**GCC工具集**：

arm-linux-uclibcgnueabi-addr2line          arm-linux-uclibcgnueabi-gcc-4.4.0

arm-linux-uclibcgnueabi-objcopy            arm-linux-uclibcgnueabi-ar

arm-linux-uclibcgnueabi-gccbug             arm-linux-uclibcgnueabi-objdump

arm-linux-uclibcgnueabi-as                     arm-linux-uclibcgnueabi-gcov

arm-linux-uclibcgnueabi-ranlib               arm-linux-uclibcgnueabi-c++

arm-linux-uclibcgnueabi-gdb                 arm-linux-uclibcgnueabi-readelf

arm-linux-uclibcgnueabi-cc                    arm-linux-uclibcgnueabi-gprof

arm-linux-uclibcgnueabi-size                 arm-linux-uclibcgnueabi-c++filt

arm-linux-uclibcgnueabi-ld                    arm-linux-uclibcgnueabi-strings

arm-linux-uclibcgnueabi-cpp                 arm-linux-uclibcgnueabi-ldconfig

arm-linux-uclibcgnueabi-strip                arm-linux-uclibcgnueabi-g++

arm-linux-uclibcgnueabi-ldd                  arm-linux-uclibcgnueabi-gcc

arm-linux-uclibcgnueabi-nm

**学习方法**

途径一：网络搜索

途径二：在linux系统下使用man命令查看手册。比如想知道链接命令的使用方法和链接选项，直接在linux系统的命令行man ld即可查看。

**常见文件的内幕**

Obj：由源文件编译而成的目标文件，包括**程序段**和**自身能输出的符号表**以及**未链接的符号表（即重定位符号表）**

Lib：由一个或者多个目标文件打包在一起的文件。简单的说就是**一堆.o打包而成的文件**。这样做主要目的就是提供一个手段可以让开发者**将一个单一的模块以二进制方式提供给该模块的应用人员**。

.out:需要说明的是，这里指的.out文件是我们工作中看到的.out文件。比如基于MSD平台编译时，在integration\product目录下都会生成一个**a.out**文件。这个文件实际上就是链接后可执行程序了。

但是他是**elf文件格式**。像**linux这样的系统是直接可以运行elf文件格式的可执行程序【类似windows中运行.exe】**，但是其他系统如OS21和ecos不支持该格式的直接运行，其运行需要将其**转换成.bin**。但是，.out文件里面包含了很多与调试信息相关的内容，我们后面说到的很多工具就会基于它来分析。

**【用GCC编译时不指定输出文件名，就默认生产a.out, gcc tst.c ==>a.out  在linux下直接可以运行该文件 ./a.out】**

.bin:纯粹二进制执行程序。由.out转换而来。**.bin往往比.out文件小很多，其内部只包含了程序执行所需要的代码段、数据段。**

**Readelf-读取文件头**

顾名思义，用于读取elf文件信息的工具。具体如下：

1.读取elf文件 文件头：

命令：readelf -h integration/product/a.out

  ELF Header:

  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00    //魔数

  Class:                         ELF32       //文件格式

  Data:                          2‘s complement, little endian //大小端

  Version:                       1 (current)

  OS/ABI:                        UNIX - System V //编译宿主机操作系统

  ABI Version:                   0 //ABI 版本

  Type:                          EXEC (Executable file) //表明是可执行文件

  Machine:                       MIPS R3000 //目标机体系架构

  Version:                       0x1

  Entry point address:           0x80000224

  Start of program headers:      52 (bytes into file)

  Start of section headers:      23916900 (bytes into file)

  Flags:                         0x50003001, noreorder, eabi32, mips32

  Size of this header:           52 (bytes)

  Size of program headers:       32 (bytes)

  Number of program headers:     1

  Size of section headers:       40 (bytes)

  Number of section headers:     38

  Section header string table index: 35

     典型问题：链接的时候提示库的格式错误之类的莫名错误。这时，其中一个可能原因就是用成了其他平台的库，或者该库是其他工具链编译出来的。遇 到此类问题就可以去比较报错的库的文件头里面的信息是否与其他库一样，这些信息主要包括上面红色部分标明的内容，如ABI规划和目标机类型等。

   命令：readelf -l integration/product/a.out

   Elf file type is EXEC (Executable file)

   Entry point 0x80000224

   There are 1 program headers, starting at offset 52

  Program Headers:

  Type     Offset         VirtAddr       PhysAddr     FileSiz      MemSiz     Flg     Align

  LOAD    0x000000 0x80000000 0x80000000 0xb23928 0x2035fc0 RWE 0x1000

 Section to Segment mapping:

  Segment Sections...

  00     .rom\_vectors .text .rodata .data .eh\_frame .gcc\_except\_table .ctors .dtors .devtab .sdata .sbss .bss

主要作用：

1.显示程序开始运行的内存地址，如上面显示的0x80000000。

2. 显示程序由那些段组成。

**注意：只有可执行文件才有程序头，目标文件和库文件没有该信息。**

**Readelf-读取段表**

命令：readelf -S integration/product/a.out

该命令将文件中存在的段的信息列出，通常情况下我们比较关注的就是代码段(.text),只读数据段(.rodata),数据段(.data),未初始化数据段(.bss).下面以代码段为例简要说明下主要几项的意义：

      [Nr] Name         Type               Addr          Off         Size     ES   Flg  Lk  Inf  Al

      [ 5] .text             PROGBITS    80001000  001000  77f9f8  00   AX   0    0    4

第一项(name)为段名，这里是代码段(.text)

第二项(type)为段类型，具体可以参考ELF规范文档

第三项(addr)为段在运行时的加载地址，为虚拟地址。

第四项(off)为该段起始地址在文件中的偏移。

第五项(size)为该段大小

第七项(Flg)包含了程序的控制信息。在命令的输出下面有说明：

     W (write), A (alloc), X (execute), M (merge), S (strings)

     I (info), L (link order), G (group), x (unknown)

    O (extra OS processing required) o (OS specific), p (processor specific)

**addr2line**

**作用：将程序地址转换成行号。**

     这个工具在我们日常开发中非常有用。他可以快速的定位到程序死机的位置。注意，这里说的死机是指程序因为非法地址访问，除数为0，地址未对其访问(部分平台有此限制),bus error等错误造成的程序崩溃。不包含死锁、程序死循环等造成的死机现象。

     在说明该工具的用法之前，先了解两个概念：

     1. epc：在学校我们学习汇编的时候知道pc是CPU保存当前运行指令地址的寄存器，那么这个**epc就是error pc。保存的是当程序崩溃时，造成指令异常的那条指令的地址**。也就是问题的第一现场。比如，程序因为非法地址访问造成了死机，那么epc保存的就是直接造成非法地址访问的那条指令的地址。

     2. ra：当前程序返回地址。当程序进行函数调用时更新该寄存器。当程序死机时，该地址就是第二现场。

        一般程序崩溃时都有epc、ra的地址打印出来。

**addr2line使用方法一**

命令：addr2line -e integration/product/a.out 802f07a8 –f

其中integration/product/a.out为造成死机对应的程序。 802f07a8 为地址。比如上面一张所说的epc             地址或者是ra地址。

示例一：

MSD5043 UNE项目时移进出老化死机，死机打印：

!!!CPU exception=4 epc=802f07a8

通过命令查找：

[root@localhost trunk]# addr2line -e integration/product/a.out 802f07a8 -f

GetShowStateFromCSWND

??:0

这里行数没有找到，但是找到了死机函数：GetShowStateFromCSWND，进一步查找发现该函数是graphic库里面的。进一步分析，该 graphic库在很多项目上使用，所以该库直接出问题的概率较小，再结合死机时的操作是在PVR的时移playbar上，再结合现象是要操作一段时间后 才出现，所以初步判断为playerbar的显示有内存泄露，根据这个线索去查最后找到问题原因。

**addr2line使用方法二**

示例二：

MSD5043 CTH项目测试部随机按键老化死机，死机打印：

!!!CPU exception=4 epc=801e6f0c

通过命令查找：

[root@localhost trunk]# addr2line -e integration/product/a.out 802f07a8 –f

OnCommandWeeklyEPGView

integration/ui/c/hd\_v10/EPG/WeeklyEpgView.c:912

这个示例中直接查找到了死机的文件已经具体行数。再根据该信息在代码中发现是由于对某个变量取余数时，发现除数为0所导致。

**Addr2line的优势与不足**

优势：

可以不用重新跑程序就可以快速的定位到死机的位置。对于解决非必现的问题非常有帮助。可以做到随时发现随时定位。

不足：

**只能看到死机当前地址和返回地址，不能显示整个堆栈函数调用的信息**。而某些问题，虽然是某个地方死机了，但是问题的根本原因不是死机的地方有问题，而可能是上几层的调用逻辑关系错误。对于这类问题，通过addr2line只能看到问题的表象，问题的根本原因无法快速定位。

Addr2line的限制与工作中的改进

限制：

1. **Addr2line要能准确找到死机位置要求代码编译时使用-g选项**。所以，在实际使用过程中有时死在某个库函数里面，然后通过该工具只能找到死机的函数，无法准确定位到行，其原因就是该库的编译

没有使用-g选项。

2. **addr2line使用时的程序文件必须是未裁剪过的.out文件，而不能是.bin文件**。

工作中的改进：

基于上面的限制，所以，我们在提交测试时可以将编译出来的中间文件：.out和map文件都放到测试包中。这样一旦测试部测试时出现死机问题，可以根据打印和.out文件，用addr2line工具快速定位到问题点，提高解决问题的效率。

**nm**

Nm主要是用于查找**目标文件**、**库文件**（**库文件实际上就是目标文件的打包**）、**elf格式可执行文件中的特定符号**，这些符号**主要是函数名和全局变量名**。这个工具通常可以帮快速定位一些链接问题。

比如一个工程链接的时候提示某个函数找不到。则通过该工具在所有库中去查找看未定义的函数到底在那个库里面，从而将该库添加到makefile中。比如假设提示无法找到符号：CSUDIPlusOSTimerStop，则可以通过命令：

**nm -A lib/MSD7853/release/\*.a | grep CSUDIPlusOSTimerStop**

获得打印信息如下：

libkernel.a:event.o:         U    CSUDIPlusOSTimerStop

libkernel.a:dsm\_sg.o:         U    CSUDIPlusOSTimerStop

libos\_udi2\_to\_udi1.a:udi1\_os.o:         U    CSUDIPlusOSTimerStop

libUDIPlus.a:udiplus\_ostimer.o:00000308 T   CSUDIPlusOSTimerStop

    上面的打印中，可以看到那些库里面和这个函数有关系。

    上面的CSUDIPlusOSTimerStop前面的“U”表示undifined，即该库里面调用该函数但是该库里面并未定义该函数。而”T”就表示 函数的定义。所以，从这个例子里面可以看到CSUDIPlusOSTimerStop是在库libUDIPlus.a的 udiplus\_ostimer.c文件中实现的，需要添加该库的链接。

注意：对于符号前的各个字母的说明，即上例中红色部分标出的内容，请用man nm查看手册

**Nm示例-查找重定义**

有些项目工程为了makefile写的简单，在编译的时候往往通过shell命令找出该目录下的所有.c文件，并将其编译。而我们调试的时候有时为了备份 一个改动，往往会将改动的文件重命名一下，然后，从服务器再取一个新的。这样，编译链接时就报错，说有重定义。或者有些平台因为makefile的原因不 会报重定义，但是一运行，发现总是没有按照自己预想的路径运行。这样我们也可以通过nm 来确认下是否我们的链接库里面某个函数有多个定义。比如假设是PVRLite播放入口重定义，或者没有按照我们的预想运行，我们找到入口函数 CSPVRLitePlayerStart，运行命令：

nm -A ./\*.a|grep CSPVRLitePlayerStart

输出：

libcbb.a:CSPVRLitePlayer.o:00000534 T CSPVRLitePlayerStart

libcbb.a:CSPVRLitePlayer-bak.o:00000534 T CSPVRLitePlayerStart

libuihd\_v10.a:CSPlayer.o:         U CSPVRLitePlayerStart

可以看到CSPVRLitePlayerStart有两个”T”,即有两个定义的地方，并且在不同文件，而且很容易看到是我们一个备份文件被编译进去导致。

**Nm示例-程序转map文件**

**nm还有作用就是直接通过可执行文件生成相应的map文件。当然，这个前提是改可执行文件没有被strip命令裁剪过。命令如下：**

nm  integration/product/a.out > flash.map

**strings**

**Strings主要用来输出目标文件、库文件、程序文件中的字符串。**比如下面中的字符串：

1.字符串变量的赋值：

Char \*pProductName = “N8770C”;

此处的”N8770C”

2.打印：

Printf(“error:parameter error\n”);

此处的” error:parameter error”

使用命令:strings –f lib/MSD7853/release/libcbb.a

示例：

之前有位同事问我，他在一个必然会走到的函数入口处用printf添加了一个打印。但是，不管怎么弄该打印就是没有打印出来。于是他怀疑是否该平台的printf打印不出来。

我们假设他加的打印是打印一句：”PvrEntry\_start”,那么，他打印不出来的一种原因是该修改根本就没有被编译到，或者使用的库根本就不是他加过打印的库。这样我们可以用nm来证实下我们的推测

：

**strings –f integration/product/a.out |grep PvrEntry\_start**

如果找到则表示修改的代码已经编译并链接到，则应该找其他没有打印出来的原因；如果没有找到，则要去查找编译链接的原因。

**ar**

Ar的主要功能大家应该都很熟悉：**将目标文件打包成库文件**。这在工程中的makefile基本都能找到他的使用场景。

这里主要介绍一个大家不太了解但是有时有需要用到的功能。

示例一：几个.o文件打包成一个.a文件

命令：ar -rcu libmain.a A.o B.o C.o

示例二：将两个(libtest1.a和libtest2.a)库文件合并成一个库文件libfinal.a

步骤1，用ar命令将每个库文件还原成.o文件:

ar –x libtest1.a

ar –x libtest2.a

步骤2，再将还原出来的所有.o文件打包成新的库文件:

ar -rcu libfinal.a ./\*.o

注意：在具体某个平台下使用时，请在工具前加上交叉工具链前缀,比如MSD平台则为:mipsisa32-elf-ar

**objdump**

**Objdump可以将目标文件或者程序文件中的段内容显示或者反汇编**。在日常工作中，我们可能用到的基本就是**他的反汇编功能**了。

有时候我们对死机地址用addr2line无法定位到时那行时，我们还可以用objdump就程序反汇编，然后在反汇编文件中查找死机地址在那个函数范围内，这样也将问题缩小在了很小的范围，一定程度上提高解决问题的效率。

**objcopy**

**Objcopy主要作用是完成目标文件或者程序文件的格式转化**。其功能比较强大，但是很底层，一般的应用开发并不需要使用到他。在我们的开发中，有两种场景下用到了objcopy。

1. **在编译链接完成后，需要将elf格式的.out文件转化成bin文件**。如MSD平台大家留意下链接时有这样一条命令打印：

mipsisa32-elf-objcopy -O  binary  a.out   k1\_ecos.bin

这就是将链接输出的.out文件转化成2进制的bin文件。因为，MSD平台使用的ECOS不支持ELF文件的执行。如果是linux系统，则.out文件可以直接运行。

2. 在linux系统下，将.out程序文件裁剪掉程序正常运行不需要的段，比如符号表、重定位表、debug信息等。这样裁剪后的程序相比原来的.out文件小很多。所以，如果在linux下，当flash放不下需要

裁剪应用程序的时候，我们首先就要确认烧录的可执行程序是否已裁剪掉程序运行不需要的东西。下面是裁剪的一个命令例子：

arm-linux-uclibcgnueabi-objcopy -gS flash -O elf32-littlearm flash.bin

**strip**

Strip用于裁剪elf格式程序。其功能与上一章的objcopy的场景2一致。Strip就是objcopy工具的一个子集。所以，不再详述。下面是命令使用示例：

mipsisa32-elf-strip -s integration/product/a.out

      大家有兴趣可以用readelf工具将裁剪前.out的段表读出和裁剪后的flash.bin的段表读出对比，看下究竟哪些段被裁掉了。

**其他命令**

ranlib : 将目标文件打包成库文件工具，通常在makefile中使用

Cpp 预编译工具

As 汇编工具

Gcc 编译工具，当然也可以用于链接

c++ C++代码编译工具

g++ java代码编译工具

Ld 链接工具

Gdb 软件调试工具，具体使用方法请参见123上的gdb培训文档