## gcc编程入门

written by 王保明

### 什么是gcc

|  |
| --- |
| gcc（GNU C Compiler）编译器的作者是Richard Stallman，也是GNU项目的奠基者。 |
| 什么是gcc：gcc是GNU Compiler Collection的缩写。最初是作为C语言的编译器（GNU C Compiler），现在已经支持多种语言了，如C、C++、Java、Pascal、Ada、COBOL语言等。 |
| gcc支持多种硬件平台，甚至对Don Knuth 设计的 MMIX 这类不常见的计算机都提供了完善的支持 |

### gcc主要特征

|  |
| --- |
| 1）gcc是一个可移植的编译器，支持多种硬件平台  2）gcc不仅仅是个本地编译器，它还能跨平台交叉编译。  3）gcc有多种语言前端，用于解析不同的语言。  4）gcc是按模块化设计的，可以加入新语言和新CPU架构的支持  5）gcc是自由软件 |

### gcc编译过程

|  |
| --- |
| 预处理（Pre-Processing）  编译（Compiling）  汇编（Assembling）  链接（Linking）  Gcc \*.c –o 1exe (总的编译步骤)  Gcc –E 1.c –o 1.i  Gcc –S 1.i –o 1.s  Gcc –c 1.s –o 1.o  Gcc 1.o –o 1exe  结论：gcc编译工具是一个工具链。。。。 |
|  |
| hello程序是一个高级Ｃ语言程序，这种形式容易被人读懂。为了在系统上运行hello.c程序，每条Ｃ语句都必须转化为低级机器指令。然后将这些指令打包成可执行目标文件格式，并以二进制形式存储器于磁盘中。 |

### gcc常用编译选项

|  |  |
| --- | --- |
| 选项 | 作用 |
| -o | 产生目标（.i、.s、.o、可执行文件等） |
| -c | 通知gcc取消链接步骤，即编译源码并在最后生成目标文件 |
| -E | 只运行C预编译器 |
| -S | 告诉编译器产生汇编语言文件后停止编译，产生的汇编语言文件扩展名为.s |
| -Wall | 使gcc对源文件的代码有问题的地方发出警告 |
| -Idir | 将dir目录加入搜索头文件的目录路径 |
| -Ldir | 将dir目录加入搜索库的目录路径 |
| -llib | 链接lib库 |
| -g | 在目标文件中嵌入调试信息，以便gdb之类的调试程序调试 |

### 练习

|  |
| --- |
| gcc -E hello.c -o hello.i（预处理）  gcc -S hello.i -o hello.s（编译）  gcc -c hello.s -o hello.o（汇编）  gcc hello.o -o hello（链接）  以上四个步骤，可合成一个步骤  gcc hello.c -o hello（直接编译链接成可执行目标文件）  gcc -c hello.c或gcc -c hello.c -o hello.o（编译生成可重定位目标文件） |
| 建议初学都加这个选项。下面这个例子如果不加-Wall选项编译器不报任何错误，但是得到的结果却不是预期的。  #include <stdio.h>  int main(void)  {  printf("2+1 is %f", 3);  return 0;  } |
| Gcc编译多个.c |
| hello\_1.h  hello\_1.c  main.c  一次性编译  gcc hello\_1.c main.c –o newhello  独立编译  gcc -Wall -c main.c -o main.o  gcc -Wall -c hello\_1.c -o hello\_fn.o  gcc -Wall main.o hello\_1.o -o newhello |

## gcc编程进价

### 1使用外部库

|  |
| --- |
| 在使用C语言和其他语言进行程序设计的时候，我们需要头文件来提供对常数的定义和对系统及库函数调用的声明。 |
| 库好处：模块化、可重用性高、可维护性好 |
| 库文件是一些预先编译好的函数集合，那些函数都是按照可重用原则编写的。它们通常由一组互相关联的用来完成某项常见工作的函数构成。比如用来处理屏幕显示情况的函数（ncurses库）和数据库访问例程（dbm库）等。 |
| /usr/include及其子目录底下的include文件夹  /usr/local/include及其子目录底下的include文件夹  /usr/lib  /usr/local/lib |
| 外部库时候用-l选项 |
| //calc.c  #include <math.h>  #include <stdio.h>  int main(void)  {  double x = pow(2.0, 3.0);  printf("The cubed is %f\n", x);  return 0;  }  gcc -Wall calc.c -o calc -lm  -lm表示要链接libm.so或者libm.a库文件 |

### 2动态库和静态库

1静态库和动态库概念

|  |
| --- |
| 静态库（.a）：程序在编译链接的时候把库的代码链接到可执行文件中。程序运行的时候将不再需要静态库 |
| 共享库（ .so或.sa ）：程序在运行的时候才去链接共享库的代码，多个程序共享使用库的代码。 |
| 1、一个与共享库链接的可执行文件仅仅包含它用到的函数入口地址的一个表，而不是外部函数所在目标文件的整个机器码  2、在可执行文件开始运行以前，外部函数的机器码由操作系统从磁盘上的该共享库中复制到内存中，这个过程称为动态链接（dynamic linking）  3、共享库可以在多个程序间共享，所以动态链接使得可执行文件更小，节省了磁盘空间。操作系统采用虚拟内存机制允许物理内存中的一份共享库被要用到该库的所有进程共用，节省了内存和磁盘空间。 |

2静态库生成

|  |
| --- |
|  |
| myadd.h  myadd.c  mymain.c  gcc -Wall myadd.c -o myadd.o  ar rcs libmyadd.a myadd.o  ar是gnu归档工具，rcs表示(replace and create)  gcc -Wall mymain.c libmyadd.a -o main 注意：带静态库全名  gcc -Wall -L. mymain.c -o main –lmyadd 注意：-L参数 |
| 备注： .a文件实质上.o文件的集合。打个包而已。 |
| C\_INCLUDE\_PATH、LIBRARY\_PATH  从左到右搜索-I -L指定的目录。  由环境变量指定的目录  由系统指定的目录 |

3动态库生成

|  |
| --- |
|  |
| shared: 表示生成共享库格式  fPIC：产生位置无关码(position independent code)  这个动态库，可在任何位置加载，叫做位置无关码。  这个动态库，在指定位置加载，成为与位置有关。。  库名规则：libxxx.so  示例：gcc -shared -fPIC myadd.o -o libmyadd.so  gcc -shared -fPIC myadd.c –o libmyadd.so 直接从c开始  使用共享库  /home/wbm04/lib/  编译选项  l：链接共享库，只要库名即可(去掉lib以及版本号)  L：链接库所在的路径.  示例：  gcc mymain.o -o mymain –L. -lmyadd |
| 使用动态库需要的配置 |
| 1、拷贝.so文件到系统共享库路径下  一般指/usr/lib  2、更改LD\_LIBRARY\_PATH  3、ldconfig  　配置ld.so.conf，ldconfig更新ld.so.cache  例如: 配置完整路径：/home/test/lib/libmyadd.so，然后再ldconfig命名更新缓存 |
| LD\_LIBRARY\_PATH |

### 3经典用法汇总

|  |
| --- |
| 目录结构：  [test@localhost mylib]$ ls  inc lib mycca myccso src srctest srctestso |
| 静态库生成及测试：  编码myccso脚本  gcc -c -Wall -I./inc ./src/myadd.c -o myadd.o  ar rcs libmyadd.a myadd.o  rm \*.o  mv libmyadd.a ./lib  gcc -Wall -g -I../inc -L../lib main.c -o main –lmyadd |
| 动态库生成及测试：  编写生成动态库gcc脚本  gcc -Wall -g -shared -fPIC -I./inc ./src/myadd.c -o libmyadd.so  mv libmyadd.so ./lib  编写测试静态库gcc脚本  gcc -Wall -g -I../inc -L../lib main.c -o main -lmyadd |
|  |

### 4相关工具

|  |
| --- |
|  |
| ar选项  option c  Create a file  option d  Delete a file from the archive  option p  Print a list of files in the archive  option q  Append files to the archive  option r  Insert new files in the archive by replacing if a file already exists in the archive  option t  Displaying contents of an archive  option x  Extracting files from an archive |
| objdump Utility  -f  Displaying Header Information  -h  Displaying Section Headers  -d  Disassembling a File  -a  Displaying Information about Library Files |
| nm Utility  used to list symbols used in an object file.  示例：  $ nm -s libcommon.a  $ nm -s a.out  另外：  –a option with the nm command also shows the debug symbols. |
| Size utility  The size utility displays sizes of each section in  an object file.  示例：  [root@boota]# size a.out  text　 　data 　bss　 dec 　hex　 filename  1015 　232 　　24 　1271 　4f7 　a.out |
| Strings utility  The strings utility displays printable strings in an object file. By default it displays strings only in initialized and loaded sections of the object file. |
| Ldd utility  The ldd utility is very useful in finding out the dependencies of an executable on shared libraries.  示例：  [root@boota]# ldd a.out  libc.so.6 => /lib/i686/libc.so.6 (0x4002c000)  /lib/ld-linux.so.2 =>/lib/ld-linux.so.2 (0x40000000)  [root@boota]# |
| Ranlib utility  The “ranlib” command is used to create index  entries inside an archive file.  示例：  $ranlib static-lib-name  等价于 $ar –s static-lib-name |

### 5开放主题

|  |
| --- |
| 应用程序主动加载动态库，win Pk Linux  可以在自己的程序中使用 dlopen()。该函数将打  开一个新库，并把它装入内存。  dlopen() 在 dlfcn.h 中定义，并在 dl 库中实现。  handle = dlopen( "/usr/lib/libalg.so", RTLD\_LAZY );  dl\_add = dlsym( handle, "add" );  dl\_mod = dlsym( handle, "mod" );  printf ("x + y = %3d\n", dl\_add( x, y ) );  printf ("x % y = %3d\n", dl\_mod( x, y ) );  dlclose( handle );  return 1;  } |

# make工具

## make基础

### make基本工作原理

|  |
| --- |
| Make工具 |
| 利用 make 工具可以自动完成编译工作。这些工作包括：如果仅修改了某几个源文件，则只重新编译这几个源文件；如果某个头文件被修改了，则重新编译所有包含该头文件的源文件。利用这种自动编译可大大简化开发工作，避免不必要的重新编译。 |
| make 工具通过一个称为 Makefile 的文件来完成并自动维护编译工作。Makefile文件描述了整个工程的编译、连接等规则。 |
|  |

### Makefile基本规则

|  |
| --- |
| TARGET … : DEPENDENCIES …  COMMAND  …  目标（TARGET）程序产生的文件，如可执行文件和目标文件；目标也可以是要执行的动作，如clean，也称为伪目标。  依赖（DEPENDENCIES）是用来产生目标的输入文件列表，一个目标通常依赖于多个文件。  命令（COMMAND）是make执行的动作（命令是shell命令或是可在shell下执行的程序）。注意：每个命令行的起始字符必须为TAB字符！  如果DEPENDENCIES中有一个或多个文件更新的话，COMMAND就要执行，这就是Makefile最核心的内容 |
| Make工具的最核心思想：如果DEPENDENCIES有所更新，需要重新生成TARGET；进而执行COMMAND命令 |
|  |

### 简单的makefile

|  |
| --- |
| main:main.o add.o sub.o  gcc main.o add.o sub.o -o main  main.o:main.c add.h sub.h  gcc -c main.c -o main.o  add.o:add.c add.h  gcc -c add.c -o add.o  sub.o:sub.c sub.h  gcc -c sub.c -o sub.o  clean:  rm -f main main.o add.o sub.o |
|  |

### make的自定义变量、自动化变量、模式规则

|  |
| --- |
| $@ 规则的目标文件名  $< 规则的第一个依赖文件名  $^ 规则的所有依赖文件列表 |
| 利用makefile可以编译多个可执行文件 |
| 模式规则  %.o:%.c  后缀规则  .c.o: |

## make进价

make常用内嵌函数

|  |
| --- |
| 函数调用  $(function arguments) |
| $(wildcard PATTERN)  当前目录下匹配模式的文件  例如：src=$(wildcard \*.c) |
| $(patsubst PATTERN,REPLACEMENT,TEXT)  模式替换函数  例如：$(patsubst %.c,%.o,$src)  等价于$(src:.c=.o) |
| shell函数  执行shell命令  例如：$(shell ls –d \*/) |
| 应用1：多级目录makefile编写 |
|  |
| 应用2：多目录makefile执行 |
|  |

## make工具在项目开发中的应用

# gdb工具

## gdb简介

|  |
| --- |
| 什么是gdb： |
| gdb是GNU debugger的缩写，是编程调试工具。 |
| Gdb功能 |
| 启动程序，可以按照用户自定义的要求随心所欲的运行程序。  可让被调试的程序在用户所指定的调试的断点处停住 (断点可以是条件表达式)。  当程序停住时，可以检查此时程序中所发生的事。  动态改变量程序的执行环境。 |
| 运行程序 |
| run(r)运行程序  run arg1 arg2 … |
|  |
|  |

## gdb调试功能

|  |
| --- |
| 查看源码 |
| list(l) - 查看最近10行源码  list fun - 查看fun函数源码  list file:fun - 查看file文件中的fun函数源码 |
| **设置断点与观察点**  break 行号  break fun  break file:行号  break file:fun  break if <condition> - 条件成立时程序停住。  info break(i b) - 查看断点  watch expr - 一量expr值发生改变，程序停住。  delete n - 删除断点 |
| **单步调试**  continue(c) - 运行至下一个断点  step(s) - 单步跟踪，进入函数，类似于VC中的step in  next(n) - 单步跟踪，不进入函数，类似于VC中的step out  finish - 运行程序，直到当前函数完成返回。并打印函数返回时的堆栈地址和返回值及参数值等信息。  until - 当厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。 |
| **gdb常用命令总结**  run(r) 运行  list(l) 列出源代码  break(b) 设置断点  info break(i b) 查看断点信息  continue(c) 继续程序运行，直到下一个断点  watch 设置观察点  step(s) 单步跟踪，类似于VC中的step in  next(n) 单步跟踪，类似于VC中的step out  finish 运行程序，直到当前函数返回  until(u) 退出循环  print(p) 查看运行时的变量以及表达式 |
|  |

gdb实战

|  |
| --- |
| **查看运行时数据**  print - 查看变量值  ptype - 查看类型  print array - 查看数组  print \*array@len - 查看动态内存  print x=5 - 改变运行时数据 |
| **程序错误**  编译错误：编写程序的时候没有符合语言规范导致编译错误。  运行时错误：编译器检查不出这种错误，但在运行的时候可能会导致程序崩溃。  逻辑错误：编译和运行都很顺利，但是程序没有干它该干的事情。 |
| **gdb调试段错误**  段错误是由于访问非法地址而产生的错误。  访问系统数据区，尤其是往系统保护的内存地址写数据。最常见就是给一个指针以0地址  内存越界(数组越界，变量类型不一致等) 访问到不属于你的内存区域 |
| **1、core文件**  **在程序崩溃时，一般会生成一个文件叫core文件。core文件记录的是程序崩溃时的内存映像，并加入调试信息。core文件生成的过程叫做core dump**  **2、设置生成core文件**  **ulimit -c 查看core-dump状态**  **ulimit -c 数字 （如：ulimit -c 1024）**  **ulimit -c unlimited**  **3、gdb利用core文件调试**  **gdb 文件名 core文件**  **bt** |
|  |
|  |