Linux下的C编程基础

GCC的编译流程

编译器的类别：

编译器可以生成用来在与编译器本身所在的计算机和操作系统（平台）相同的环境下运行的目标代码，这种编译器又叫做“本地”编译器。

编译器也可以生成用来在其它平台上运行的目标代码，这种编译器又叫做交叉编译器。

GCC：

是一套由 GNU 开发的编程语言编译器。它是一套 GNU编译器套装以 GPL 及 LGPL 许可证所发行的自由软件，也是 GNU计划的关键部分，亦是自由的类Unix及苹果电脑 Mac OS X 操作系统的标准编译器。

Linux 上最常用的 C 语言编译系统是 gcc (GNU Compiler Collection), 它是GNU项目中符合ANSI C标准的编译系统。

在shell的提示符下键入gcc -v，屏幕上就会显示出目前正在使用的gcc的版本。

编译流程：

预处理（Pre-Processing）

编译 （Compiling）

汇编 （Assembling）

链接 （Linking）

gcc编译系统文件名后缀

		文件名后	

文件名后缀	文 件 类 型	缀	文 件 类 型
.c	C源文件	.F .fpp .FPP	FORTRAN源文件
.i	预处理后的C源文件	.s	汇编程序文件
.ii	预处理后的C++源文件	.S	必须预处理的汇编程序文件
.m	Objective-C源文件	.o	目标文件
.mi	预处理后的Objective-C源文件	.a	静态链接库
.h	头文件	.so	动态链接库
.C .cc .cp .cpp .c++ .cxx	C++源文件		

预处理阶段

预处理是常规编译之前预先进行的工作.它读取c语言源文件,对其中以“#”开头的指令 - “伪指令”和特殊符号进行处理.

伪指令主要包括

- 头文件包含(#include <my.h>, #include “my.h”)
- 宏定义
- 条件编译指令

特殊符号(__FILE__, __LINE__, __DATE__)

编译阶段

本阶段对预处理之后的输出文件进行词法分析和语法分析,试图找出所有不符合语法

规则的部分.并根据问题大小给出错误消息,终止编译,或者给出警告,继续做下去.在确定各成分都符合语法规则后,将其”翻译”为功能等价的中间代码表示或者汇编代码.

用户可以用“-S”选项进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

```
gcc –S hello.i –o hello.s
```

汇编阶段

该阶段是汇编程序把汇编语言源代码翻译成目标机器代码的过程.
目标文件由机器码构成.通常它至少有代码段和数据段两部分.前者包含程序的指令,后者存放程序中用到的各种全局或静态的数据.
用户可以用“-c”选项。该选项只激活预处理,编译,和汇编,也就是他只把程序做成obj文件

```
gcc –c hello.s –o hello.o
```

链接阶段

链接程序要解决外部符号访问地址问题,也就是将文件中引用的符号(如符号或函数调用)与该符号在另外一个文件中的定义连接起来,从而使有关的目标文件连成一个整体,最终成为可被操作系统执行的可执行文件。

> *函数库分为**静态库**和**动态库***
> *静态库在编译链接时，把库文件的代码全部加入到可执行文件中，生成的文件比较大，运行时不需要库文件。后缀名一般为“.a”*
> *动态库编译链接时未把库文件的代码加入到可执行文件中，在程序执行时由运行时链接文件加载库。后缀名一般为“.so”。*

GCC编译选项

选项格式	功能
-c	只生成目标文件，不进行连接。用于对源文件的分别编译
-S	只进行编译，不做汇编，生成汇编代码文件格式，其名与源文件相同，但扩展名为.s
-o file	将输出放在文件file中。如果未使用该选项，则可执行文件放在a.out中
-g	指示编译程序在目标代码中加入供调试程序gdb使用的附加信息
-v	在标准出错输出上显示编译阶段所执行的命令，即编译驱动程序及预处理程序的版本号
-O -O1	试图减少代码大小和执行时间，但并不执行需要花费大量编译时间的任何优化
-O2	在-O1级别的优化之上，还进行一些额外调整工作——除不做循环展开、函数内联和寄存器重新命名外，几乎进行所有可选优化
-O3	除了完成所有-O2级别的优化之外，还进行包括循环展开和其他一些与处理器特性相关的优化工作
-O0	不执行优化
-Os	具有-O2级别的优化，同时并不特别增加代码大小
object-file-name	不以专用后缀结尾的文件名就认为是目标文件名或库名。连接程序可以根据文件内容来区分目标文件和库
-c -S -E	如果使用其中任何一个选项，那么都不运行连接程序.
-l library	连接时搜索由library命名的库。连接程序按照在命令行上给定的顺序搜索和处理库及目标文件。实际的库名是liblibrary，但按默认规则，开头的lib和后缀（.a或.so）可以被省略
-static	在支持动态连接的系统中，它强制使用静态链接库，而阻止连接动态库。而在其他系统中无起作用

	库；而在其他系统中不起作用
-L dir	把指定的目录dir加到连接程序搜索库文件的路径表中，即在搜索-l后面列举的库文件时，首先到dir下搜索，找不到再到标准位置下搜索
-B prefix	该选项规定在什么地方查找可执行文件、库文件、包含文件和编译程序本身数据文件
-o file	指定连接程序最后生成的可执行文件名称为file，不是默认的a.out

GDB调试器的使用

Linux 包含了一个叫 gdb 的 调试程序. gdb 是一个用来调试 C 和 C++ 程序的强力调试器. 它使你能在程序运行时观察程序的内部结构和内存的使用情况.

一般来说，GDB主要帮忙你完成下面四个方面的功能：

1、启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。

2、可让被调试的程序在你所指定的调置的断点处停住。（断点可以是条件表达式）

3、当程序被停住时，可以检查此时程序中发生的事。

4、动态的改变程序的执行环境。

1.加入调试信息

首先 在编译时，我们必须要把调试信息加到可执行文件中。使用编译器（ cc/gcc/g++）的 -g 参数可以做到这一点。

如： `gcc -g hello.c -o hello`

如果没有-g，你将看不见程序的函数名、变量名，所代替的全是运行 时的内存地址。

2.启动Gdb

`gdb <program>` (program是要调试的可执行文件)

Gdb进行调试的是可执行文件，而不是源代码，因此要先通过Gcc编译生成可执行

文件才能用Gdb进行调试

3.Gdb命令种类：

Gdb中的命令主要分为以下几类：

- 工作环境相关命令
- 设置断点与恢复命令
- 源代码查看命令
- 查看运行数据相关命令
- 修改运行参数命令

暂停/恢复程序运行

在gdb中，我们可以有以下几种暂停方式：断点（BreakPoint）、观察点（Watch Point）、捕捉点（Catch Point）、信号（Signals）、线程停止（Thread S–>s）。如果要恢复程序运行，可以使用c或是continue命令。

设置断点

用break命令来设置断点。设置断点的方法：

break <function>

break <linenum>

break filename:linenum

break *address

break +offset

break -offset

查看断点时，可使用info命令

设置观察点

观察点一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停住程序。我们有下面的几种方法来设置观察点：

watch <expr>

rwatch <expr>

awatch <expr>

info watchpoints

设置捕捉点（**CatchPoint**）

可设置捕捉点来补捉程序运行时的一些事件。如：载入共享库（动态链接库）或是C++的异常。设置捕捉点的格式为：catch <event>

当event发生时，停住程序。event可以是下面的内容：

1、throw 一个C++抛出的异常。（throw为关键字）

2、catch 一个C++捕捉到的异常。（catch为关键字）

3、exec 调用系统调用exec时。（exec为关键字，目前此功能只在HP-UX下有用）

4、fork 调用系统调用fork时。（fork为关键字，目前此功能只在HP-UX下有用）

5、vfork 调用系统调用vfork时。（vfork为关键字，目前此功能只在HP-UX下有用）

6、load 或 load <libname> 载入共享库（动态链接库）时。（load为关键字，目前此功能只在HP-UX下有用）

7、unload 或 unload <libname> 卸载共享库（动态链接库）时。（unload为关键字，目前此功能只在HP-UX下有用）

tcatch <event>

维护停止点

GDB中的停止点也就是上述的三类。在GDB中，如果你觉得已定义好的停止点没有

用了，你可以使用delete、clear、disable、enable这几个命令来进行维护。

查看文件

键入"l"即可查看所载入文件

运行代码

键入"r"默认从首行执行（若想从指定位置开始运行在r后加上行号）

变量的检查和赋值

whatis:识别数组或变量的类型

ptype:比whatis的功能更强，他可以提供一个结构的定义

set variable:将值赋予变量

print 除了显示一个变量的值外，还可以用来赋值

单步运行

单步运行可以使用命令“n”(next)或“s”（step）

两者区别在于：若有函数调用的时候，“s”会进入函数而“n”不会进入该函数。

“s”类似“step in”

“n”类似“step over”

恢复程序运行

在查看完所需变量及堆栈情况后，可以使用命令“c”（continue）恢复程序的正常运行。

函数的调用

call name 调用和执行一个函数

```
(gdb) call gen_and_sork( 1234,1,0 )
(gdb) call printf(“abcd”)
```

finish 结束执行当前函数，显示其返回值（如果有的话）

修改运行参数

在单步执行的过程中，键入命令“set 变量＝设定值”。这样，在此之后，程序就会按照该设定的值运行了

机器语言工具

有一组专用的gdb变量可以用来检查和修改计算机的通用寄存器，gdb提供了目前每一台计算机中实际使用的4个寄存器的标准名字：

$pc ： 程序计数器

$fp ： 帧指针（当前堆栈帧）

$sp ： 栈指针

$ps ： 处理器状态

> *Gdb 使用要点:*
> *Gcc编译选项中一定要加入“-g”*
> *只有在代码处于“运行”或“暂停”状态时才能查看变量值*
> *设置断点后程序在指定行之前停止*

Make工程管理器的使用

Make 工程管理器是个“自动编译管理器”，它能够根据文件时间戳自动发现更新过

的文件而减少编译的工作量，同时，他通过读入makefile文件的内容来执行大量的编译工作。

GNU make工作时的执行步骤

1、读入所有的Makefile。

2、读入被include的其它Makefile。

3、初始化文件中的变量。

4、推导隐晦规则，并分析所有规则。

5、为所有的目标文件创建依赖关系链。

6、根据依赖关系，决定哪些目标要重新生成。

7、执行生成命令。

Makefile 基本结构

在一个Makefile中通常包含如下内容：

需要由make工具创建的**目标体**（target），通常是目标文件或可执行文件

要创建目标体**所依赖的文件**(dependency_file)

创建每个目标体时需要运行的**命令**（command）

格式为：

```
target：dependency_file
    command
```

> *注：在Makefiel中每个command前必须有“Tab”符，否则在运行make 命令时会出错*

Make 的工作流程

Make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。

找文件中的第一个**目标文件**（target）,并把这个文件作为最终的目标文件

如果目标文件不存在，或者目标文件所依赖的后面的.o文件修改时间比该文件新，就会执行后面的定义来生成目标文件（这是执行Makefile的条件）

如果目标文件所依赖的.o文件也存在，make会在当前文件中找目标为.o文件的依赖性，如果找到根据规则生成.o文件

当然.c和.h文件是存在的，于是make会生成.o文件，然后用.o文件生成make的终极任务即可执行任务。

示例:

```
test.o: test.c test.h
        gcc -c test.c
clean:
        rm -f *.o
```

自动变量

由于常见的Gcc编译语句中通常包含了目标文件和依赖文件，而这些文件在Makefile文件中目标体的一行已经有所体现，因此，为了进一步简化Makefile的编写，就引入了自动变量。

命令格式	含义
$*	不包含扩展名的目标文件名称
$+	所有的依赖文件，以空格分开，并以出现的先 后为序，可能包含重复

	的依赖文件
$<	第一个依赖文件的名称
$?	所有时间戳比目标文件晚的依赖文件，并以空格分开命令格式
$@	目标文件的完整名称
$^	所有不重复的依赖文件，以空格分开

环境变量

Make在启动时自动读取系统已经定义的环境变量，并且会创建与之相同名称和数值的变量。但是用户在makefile 中定义了相同名称的变量，那么用户自定义的变量会覆盖同名的环境变量。

隐式规则

隐含规则能够告诉make怎样使用传统的技术完成任务，这样，当用户使用它们时就不必详细指定编译的具体细节，而只需把目标文件列出即可。Make会自动搜索隐式规则目录来确定如何生成目标文件。

模式规则

模式规则是用来定义相同处理规则的多个文件的。它不同于隐式规则，隐式规则仅仅能够用make默认的变量来进行操作，而模式规则还能引入用户自定义变量，为多个文件建立相同的规则，从而简化Makefile的编写。
模式规则的格式类似于普通规则，这个规则中的相关文件前必须用“%”标明。

```
%.o : %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

伪目标

make允许指定伪目标。称其为伪目标是因为它们并不对应于实际的文件。伪目标体规定了make应该执行的命令。但是，因为clean没有依赖体，所以它的命令不会被自动执行。 如前例中的clean。

注释

注释用“#”字符（“\”用于转义）

文件引用

在Makefile使用include关键字可以把别的Makefile包含进来。

示例:
有这样几个Makefile：`a.mk`、`b.mk`、`c.mk`，还有一个文件叫`foo.make`，以及一个变量$(bar)，其包含了`e.mk`和`f.mk`，那么

```
include foo.make *.mk $(bar)
```

等价于：

```
include foo.make a.mk b.mk c.mk e.mk f.mk
```

AutoTools工具

Autotools系列工具它只需用户输入简单的目标文件、依赖文件、文件目录等就可以轻松地生成Makefile了。

GNU autotools主要包括下面三个工具：

- Autoconf ：这个工具用来生成configure脚本。就像前面提到的，这个脚

本主要用来分析你的系统以找到合适的工具和库。譬如:你的系统的C编译器是“cc”还是“gcc”？

- Automake ：这个工具用来生成Makefiles。它需要使用到Autoconf提供的信息。譬如，如果Autoconf检测到你的系统使用“gcc”，那Makefile就使用gcc作为C编译器。反之，如果找到“cc”，那就使用“cc”。
- Libtools ：这个工具创建共享库。它是平台无关的。