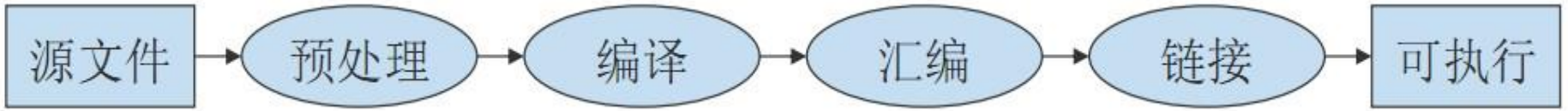
Linux操作系统编程

Linux程序编译调试方法

gcc概述

- gcc是GNU计划的一个项目。是一个自由编译器，如今的gcc已经是一个包含众多语言的编译器了（C，C++，Ada，Object C，Java及Go等）。所以，GCC也由原来的GNU C Compiler变为GNU Compiler Collection
- GCC主要包括:
  - cpp(预处理器)
  - gcc(c编译器)、g++(c++编译器)等编译器
  - binutils等二进制工具.
    - ⑩as(汇编器)
    - ⑩ ld(链接器)
    - ⑩…

gcc编译过程




```
# cpp -o hello.i hello.c

# cc1 -o hello.s hello.i

# as -o hello.o hello.s

# ld -o hello hello.o
```

gcc基础使用方法

- gcc命令格式：gcc [选项] <文件名>
  - -o filename ： 指定输出文件为filename.该选项不在乎gcc产生什么输出，无论是可执行文件，目标文件，汇编文件还是预处理后的C代码
  - 如果没有使用”-o”选项,默认的输出结果是:可执行文件为”a.out”，编译后产生的目标文件是”sourcename.o”，汇编文件是 “sourcename.s”，而预处理后的C源代码送往标准输出
- 对于源代码main.c，可以通过如下命令编译成最终可执行文件（默认包含了预处理、编译、汇编及链接四个阶段）：

  gcc main.c –o main

GCC基础使用方法

```c
#include "myprj.h"
//hello
int main(int argc, char *argv[])
{
  addRecord();
  modifyRecord();
  deleteRecord();
  exit(0);
}
```

```c
//myprj.h
extern void addRecord();
extern void modifyRecord();
extern void deleteRecord();
```

```c
//add.c
#include <stdio.h>
void addRecord()
{
 printf("record added successful\n");
}
```

```c
//delete.c
#include<stdio.h>
void deleteRecord()
{
  printf("record deleted successfully\n");
}
```

```c
//modify.c
#include <stdio.h>
void modifyRecord()
{
  printf("record modified successfully\n");
}
```

- 编译方法： 首先需要生成目标文件 example.o add.o modify.o delete.o

```
gcc –c example.c
gcc –c add.c
gcc –c modify.c
gcc –c delete.c
```

- 链接 4个目标文件，生成可执行文件 example

```
gcc example.o add. o modify.o delete.o –o example
```

gcc常用编译选项

- **-D: 宏定义选项，等同于代码中的#define MACRO 但-D定义的宏作用于所有的源文件。**
  - 范例:

    #define PI 3.14159(如果程序用到PI则用3.14159代替)

    gcc -DPI=3.14159 main.c（但如果没有定义宏的话，就可以直接在编译的时候赋值再运行）
- **-I 头文件的搜索路径:如果用户的头文件不在gcc的搜索路径中，可以用此选项指定额外搜索路径。**
  - 范例：

    gcc helloworld.c –I /usr/include –o helloworld（将/usr/inlcude加入到文件头文件的搜索路径中）

gcc常用编译选项

- 警告选项
  - 警告是针对程序结构的诊断信息,程序不一定有错误,而是存在风险,或者可能存在错误。
  - 所有以-W开头的选项基本上均可使用-Wno-option来关闭该警告信息，如-Wunused在某个局部变量除了声明就没再使用,或者声明了静态函数但是没有定义,或者某条语句的运算结果显然没有使用时, 编译器就发出警告。使用-Wno-unused可禁止该警告信息。
  - -w：禁止所有警告信息.
  - -Wall：打开所有警告选项，输出警告信息
- 通常建议打开-Wall，这样至少可以看出你的代码里有哪些地方可能存在问题。

静态库编译和使用

- 把 ‘.c’编译成 ‘.o’

  gcc –c increase.c –o increase.o

- 把 ‘.o’归档成 静态库 ‘.a’

  ar –r libincrease.a increase.o

- 静态库和其它源文件链接成可执行文件

  gcc main.c –L –static –o main

动态库编译

- 生成动态链接库

  gcc -shared -fPIC -o libinc.so increase.c

  - 动态链接库的名字必须以lib开头 .so结束，这是linux系统上的强制约束，否则无法使用该共享库
  - **-shared** 生成共享文件
  - **-fPIC** 生成位置独立的代码，此类代码可以在不同进程间共享。

动态库的使用

- **-l*library*** 名字为**library**的动态链接库。事实上此动态链接库在文件系统中的名字为**lib*library* .so**。连接器会自动加上**lib*.so**。
- **-L*dir*** *共享库搜索目录*。**gcc**除了会在自定义的目录中搜索共享库外，用户也自定义目录让**gcc**搜索。

  gcc main.c -o main -linc -L./

gdb概述

- gdb是GNU计划开发的程序调试工具
- gdb可以完成以下四个方面的功能：
  - 启动程序，可以按照自定义的要求随心所欲的运行程序
  - 可让被调试的程序在所指定的调置的断点处停住（断点可以是条件表达式）
  - 当程序被停住时，可以检查此时程序中所发生的情况
  - 动态的改变程序的执行环境。

gdb的启动

- 直接在**shell**中运行gdb命令，进入**gdb**界面后用**file program** 装载程序。
- 在**shell**中启动**gdb**并加载可执行文件

  gdb <program>
- 用**gdb**同时调试一个运行程序和**core**文件（**core**是程序非法执行后**core dump**后产生的文件）

  gdb program core
- 调试正在运行的进程

  gdb program <processid>

  进入**gdb**后用 **attach <processid>** 调试正在运行的进程。

gdb常用命令：break

- 功能：断点设置命令break(缩写 b)，当gdb执行到该断点时会让程序暂停运行。此时程序员可以查看运行中程序的情况。
- 格式：break [LOCATION] [thread THREADNUM] [if CONDITION]
- [LOCATION]：
  - linenum（行号）
  - function name(函数名)
  - filename:linenum
  - filename:function
  - class:function（c++）

  b 123　b main　b increase:main　b increase:123
- [thread THREADNUM] 调试多线程程序时，切换到哪个线程或者在那个线程中设置断点。 break frik.c:13 thread 28
- [if CONDITION]： 当条件满足时，断点才生效。一般称为条件断点。CONDITION跟C语言一样

  b 123 if index==2

  当index为2时，程序在123行停下。

gdb常用命令：watchpoint

- watchpoint称为观察点，当观察对象的值有变化时，程序立即停止执行。
- watch <expr>：为表达式（变量）expr设置一个观察点。一旦表达式值有变化时，马上停住程序。
- rwatch <expr>：当表达式（变量）expr被读时，停住程序。
- awatch <expr>：当表达式（变量）的值被读或被写时，停住程序。
- info watchpoints：列出当前所设置了的所有观察点。

清除禁止断点或观察点

- clear [linenum] [function name] 清除所有断点,不会清除watchpoints。
- delete <num> 清除编号为num的断点或者watchpoint。
- disable <num> 禁止某个断点。
- enable <num> 开启某个断点。

gdb调试命令

- step 单步调试命令，一次执行一行程序。
- next 单步调试命令，但跳过函数调用。
- finish 单步调试时直接从一个函数中返回
- disassemble 显示汇编代码。

gdb调试命令

- backtrace或者bt 查看目前程序的堆栈情况。
- where查看当前位置。
- up/down 向上或者向下移动一个堆栈。
- frame<num>或者f 移动到第num个堆栈。
- 当移动到某个堆栈时，便可以用gdb命令查看在那个堆栈中的局部变量。