GCC寫個庫給你玩，就這?

原創 DeRoy 編程學習基地 2020-12-21

收錄於話題
#Linux
1個 >

點擊藍字**關注我們**

前言

什麼是GCC

GCC原名為 GNU C語言編譯器

「GCC」 (GNU Compiler Collection,GNU編譯套件)

是由GNU開發的編程語言編譯器。

正文

安裝命令

```
sudo apt-get insatll gcc g++
```

注意安裝版本要大於4.8.5因為4.8.5以後的版本才支持c++11標準

查看版本

```
gcc -v
gcc --version
g++ -v
g++ --version
```

gcc和g++的區別

> gcc和g++ 都是GNU (組織)的一個編譯器。

- 「誤區一」：gcc只能編譯c代碼，g++只能編譯c++代碼。

「兩者都可以」，請注意:

1. 「後綴為.c」的，gcc把它當作是「C程序」，而g++當作是c++程序
2. 「後綴為.cpp」的，兩者都會認為是「C++程序」，C++的語法規則更加嚴謹一些
3. 編譯階段，g++會調用gcc，對於C++代碼，兩者是等價的，但是因為 gcc命令不能自动和C++程序使用的库联接，所以通常用g++来完成链接 ，為了統一起見，乾脆編譯/鏈接統

統用g++了，這就給人一種錯覺，好像cpp程序只能用g++似的.

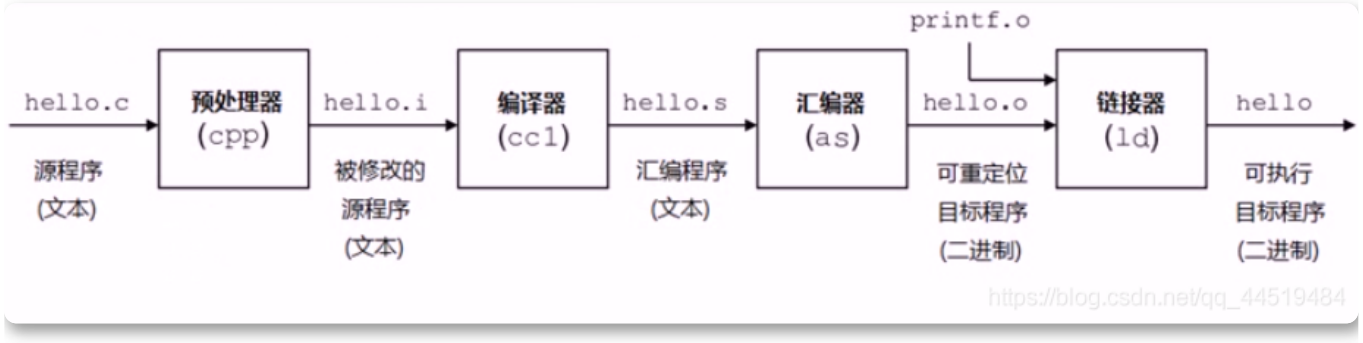
■誤區二: gcc不會定義 _cplusplus 宏，而g++會

1. 實際上，這個宏只是 标志着编译器将会把代码按C还是C++语法来解释
2. 如上所述，如果「後綴為.c」，並且採用gcc編譯器，則該宏就是未定義的，否則，就是已定義

■ 誤區三:編譯只能用gcc,鏈接只能用g++

1. 嚴格來說，這句話不算錯誤，但是它混淆了概念，應該這樣說: 编译可以用gcc/g++，而链接可以用g++ 或者gcc -lstdc++ 。
2. gcc命令不能自动和C++程序使用的库联接 ，所以通常使用g++來完成聯接，但在編譯階段，g++會自動調用gcc,二者等價.

gcc編譯過程



在這裡插入圖片描述

gcc常用參數

選項名	作用
-o	產生目標（.i、.s、.o、可執行文件等）
-E	只運行C預編譯器
-S	告訴編譯器產生彙編程序文件後停止編譯，產生的彙編語言文件拓展名為.s
-c	通知gcc取消連接步驟，即編譯源碼，並在最後生成目標文件
-w	不產生任何警告信息
-Wall	使gcc對源文件的代碼有問題的地方發出警告
-Idir	指定include 包含文件的搜索目錄
-Ldir	指定編譯的時候，搜索的庫的路徑。
-lLib	在程序編譯的時候，指定使用的庫
-g	在目標文件中嵌入調試信息，以便gdb之類的調試程序調試
-D	允許從編譯程序命令行進行宏定義符號

gcc的使用示例：

```
gcc -E hello.c -o hello.i   #对hello.c文件进行预处理，生成了hello.i 文件
gcc -S hello.i -o hello.s   #对预处理文件进行编译，生成了汇编文件
gcc -c hello.s -o hello.o   #对汇编文件进行编译，生成了目标文件
gcc hello.o -o hello   #对目标文件进行链接，生成可执行文件
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
```

「-D參數演示」

測試代碼如下：

```
#include<stdio.h>

int main()
{
#ifdef DEBUG
 printf("DEBUG:\n");
#else
 printf("Normal:\n");
#endif
 for(int i=0;i<3;i++)
  printf("work\n");
 return 0;
}
```

測試命令

```
gcc -o Debug Debug.c
./Debug
Normal:
work
work
work
```

```
gcc -o Debug Debug.c -DDEBUG
./Debug
```

```
DEBUG:
work
work
work
```

庫的介紹

「什麼是庫？」

> 庫文件是計算機上的一類文件，可以簡單的把庫文件看成一種代碼倉庫，它提供給使用者一些可以直接拿來用的變量、函數或類。

庫是特殊的一種程序，編寫庫的程序和編寫一般的程序區別不大，只是庫不能單獨運行。

库文件有两种，静态库和动态库(共享库)

「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。** 静态库比较占用磁盘空间，而且程序不可以共享静态库。运行时也是比较占内存的，因为每个程序都包含了一份静态库。

「动态库（.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，这样就减少了程序的体积。**

「库的好处」：

1.代码保密

2.方便部署和分发

生成静态库

「静态库命名规则：」

- **「Linux」**：libxxx.a

lib ： 前缀（固定）

xxx ： 库的名字，自己起 .

a ： 后缀（固定）

- **「Windows」**：libxxx.lib

Linux生成静态库

首先准备几个文件和文件夹,文件树形结构

```
[root@deroy]# tree
.
├── calc
│   ├── add.c
│   ├── div.c
│   ├── head.h
│   ├── main.c
│   ├── mult.c
```

```
|   └── sub.c
└── library
    ├── include
    ├── lib
    └── src
```

「add.c」

```c
#include<stdio.h>
#include"head.h"

int add(int a,int b)
{
 return a+b;
}
```

「div.c」

```c
#include<stdio.h>
#include"head.h"

double divide(int a,int b)
{
 if(b==0)
  return (double)a/b;
 else
  return 0;
}
```

「mult.c」

```c
#include<stdio.h>
#include"head.h"

int multiply(int a,int b)
{
 return a*b;
}
```

「sub.c」

```c
#include<stdio.h>
#include"head.h"

int subtract(int a,int b)
{
 return a-b;
}
```

「head.h」

```c
#ifndef _HEAD_H_
#define _HEAD_H_
```

```
int add(int a,int b);
double divide(int a,int b);
int multiply(int a,int b);
int subtract(int a,int b);

#endif
```

为了生成 「.a 文件」，我们需要先生成 「.o文件」。

```
[root@deroy]# cd calc/
[root@calc]# gcc -c add.c div.c mult.c sub.c
```

打包生成静态库

```
[root@calc]# ar rcs libcalc.a add.o sub.o mult.o div.o
```

ar 是 gun 归档工具，rcs 表示 replace and create ，如果 libhello 之前存在，将创建新的 libhello.a 并将其替换。

> r - 将文件插入备存文件中
>
> c - 建立备存文件
>
> s - 索引

「将库放到指定位置」

```
[root@calc]# cp libcalc.a ../library/lib/
[root@calc]# cp head.h ../library/include/
[root@calc]# cp add.c div.c mult.c sub.c ../library/src/
```

库目录结构如下(这个库目录就是发给被人用的)

```
[root@ecs-x-medium-2-linux-20200312093025 library]# tree
.
├── include
│   └── head.h
├── lib
│   └── libcalc.a
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c
```

使用静态库

编辑 main.c 文件

```
#include<stdio.h>
#include"head.h"
```

```c
int main()
{
 int a = 20;
 int b = 12;
 printf("a = %d,b = %d\n",a,b);

 printf("a + b = %d\n",add(a,b));
 printf("a - b = %d\n",subtract(a,b));
 printf("a * b = %d\n",multiply(a,b));
 printf("a / b = %d\n",divide(a,b));
 return 0;
}
```

然后就可以这样来使用静态库 `libcalc.a`

```
[root@library]# gcc main.c -o app -I ./include/ -lcalc -L./lib
[root@library]# ./app
a = 20,b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 12
```

```
其中:
-I  directory 指定include包含文件的搜索目录
-l 在程序编译的时候，指定使用的库
-L directory 指定编译的时候，搜索的库的路径
```

生成动态库(共享库)

动态库命名规则：

「Linux：libxxx.so」

「lib：前缀（固定）」

「xxx：库的名字，自己起」.

「so：后缀（固定）」

「在Linux下是一个可执行文件」

Windows : libxxx.dll

使用静态库的测试代码，库目录结构还是一样

```
[root@deroy]# tree
.
├── calc
│   ├── add.c
│   ├── div.c
│   ├── head.h
│   ├── main.c
│   ├── mult.c
│   └── sub.c
└── library
    ├── include
    ├── lib
```

```
└── src
```

为了生成 「.so」 文件，我们需要先生成 「.o」 文件得到和位置无关的代码。

```
[root@calc]# gcc -c -fpic add.c div.c mult.c sub.c
```

打包生成「动态库」

```
[root@calc]# gcc -shared add.o sub.o mult.o div.o -o libcalc.so
```

「将库放到指定位置」

```
[root@calc]# cp libcalc.so ../library/lib/
[root@calc]# cp head.h ../library/include/
[root@calc]# cp add.c div.c mult.c sub.c ../library/src/
```

库目录结构如下(这个库目录就是发给被人用的)

```
[root@library]# tree
.
├── include
│   └── head.h
├── lib
│   └── libcalc.so
```

```
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c
```

使用动态库

```
[root@calc]# gcc main.c -o app -I include/ -L lib/ -l calc
[root@calc]# ldd app
        linux-vdso.so.1 =>  (0x00007ffc75cb0000)
        libcalc.so => not found
        libc.so.6 => /lib64/libc.so.6 (0x00007f3605ddb000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f36061a9000)
```

> 通过ldd命令查看程序动态库依赖关系

ldd是 `list dynamic dependencies` 的缩写，意思是列出动态库依赖关系。

结果发现 `libcalc.so => not found` 找不到了

「那么如何让程序找到依赖库呢？这里提供四种方法」

方法一(不推荐)

```
#拷贝.so文件到系统共享库路径下，一般指/usr/lib或者/lib/目录
```

```
sudo cp ./lib/libcalc.so /usr/lib/
```

方法二(临时环境变量)

```
[root@library]# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/root/deroy/library/lib
[root@library]# echo $LD_LIBRARY_PATH
:/root/deroy/library/lib
[root@library]# ldd app
        linux-vdso.so.1 =>  (0x00007fffe1570000)
        libcalc.so => /root/deroy/library/lib/libcalc.so (0x00007f007020f000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f006fe41000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f0070411000)
[root@library]# ./app
a = 20,b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 12
```

运行成功

> 缺陷：只在当前终端有效，关闭中端后就没用了

方法三(配置用户环境变量)

> 将环境变量写入到 `~/.bashrc` 即可,即将下面内容添加到末尾

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/root/deroy/library/lib
```

```
[root@library]# cd ~
[root@~]# vim .bashrc
[root@~]# . .bashrc   #相当于source .bashrc
[root@~]# source .bashrc
[root@~]# cd deroy/library/
[root@library]# ldd app
        linux-vdso.so.1 =>  (0x00007ffe0d2db000)
        libcalc.so => /root/deroy/library/lib/libcalc.so (0x00007f937669c000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f93762ce000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f937689e000)
[root@library]# ./app
a = 20,b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 12
```

方法四(配置系统环境变量)

将环境变量写入到 `~/etc/profile` 即可,即将下面内容添加到末尾，需要root权限

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/root/deroy/library/lib
```

```
[root@library]# vim /etc/profile
[root@library]# source /etc/profile
```

```
[root@library]# ldd app
        linux-vdso.so.1 =>  (0x00007ffe08dc3000)
        libcalc.so => /root/deroy/library/lib/libcalc.so (0x00007f0eada81000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f0ead6b3000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f0eadc83000)
[root@library]# ./app
a = 20,b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 12
```

总结

静态库的优缺点

「优点」

- 静态库被打包到应用程序中加载速度快
- 发布程序无需提供静态库，移植方便

「缺点」

- 消耗系统资源，浪费内存 ◆ 更新、部署、发布麻烦

动态库的优缺点

「优点」

- 可以实现进程间资源共享（共享库）
- 更新、部署、发布简单
- 可以控制何时加载动态库

「缺点」

- 加载速度比静态库慢
- 发布程序时需要提供依赖的动态库

发送**「关键字」**获取**「Linux安装配置视频」**

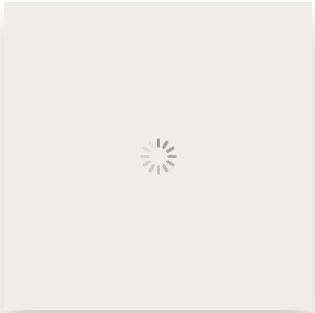
和**「GCC详细使用视频」**

VMware下Ubuntu16.04镜像完整安装配置教程

关键字【GCC】

End

你们的在看就是对我最大的肯定，
点个在看好吗~



编程学习基地

常回基地看看

喜欢此内容的人还喜欢

嵌入式系统C编程之错误处理实战篇（附代码）

技术让梦想更伟大

深度好文|面試官：進程和線程，我只問這19個問題

程序喵大人

Qt信號與槽簡單了解

C語言Plus