1. 概论

1. 1. gcc、g++之异同

1、g++和 gcc 都是 GNU 组织发布的编译器，两者存在不同，这里分成三中文件说明

- 第一种：扩展名为.c 的文件
  - gcc 会把他当成 c 程序来处理
  - 而 g++会把他当成 c++程序处理
- 第二种：扩展名为.c++的文件
  - 两者都会当成 C++程序处理
- 第三种：扩展名为.cpp 的文件
  - 在编译阶段，其实 gcc 和 g++都是相同的，都使用的是 gcc 来进行处理
  - 但是当进入链接阶段的时候
    - gcc 无法自动链接 C++的函数库，要想链接 C++函数库，必须手动操作(添加**-lstdc++参数**)，命令为 gcc hello.cpp **-lstdc++** -o hello
    - 而 g++则会自动链接 C++的函数库
  - 为了方便起见，对于.cpp 的文件就直接使用 g++来进行编译和连接，省去了使用 gcc 进行编译的阶段，从而使得有些人感觉对于.cpp 文件的处理么有 gcc 什么事儿，其实不然，**编译阶段 g++还是使用了 gcc 进行编译**

2、两个例子

- g++ -std=gnu++11 -pthread -o LTEV2X *.cpp -lfftw3 <==编译成功
  - -lfftw3：这一项不是 gcc/g++的参数，而是代表一个动态/静态库文件，不能放到前面作为参数，只能放在后面作为连接文件
- gcc -std=gnu++11 -pthread -o LTEV2X *.cpp -lfftw3 -lstdc++ <==失败，以下是错误提示
  - note: 'floor@@GLIBC_2.2.5' is defined in DSO /lib64/libm.so.6 **so try adding it to the linker command line**
- gcc -std=gnu++11 -pthread -o LTEV2X *.cpp -lfftw3 -lstdc++ **/lib64/libm.so.6**
  - -lstdc++：与-lfftw3 不同，是作为 gcc 的参数使用的，因此可以放在任意位置，该参数的意思是，在链接过程中会去调用 c++的动态链接库

1. 2. 编译过程

1、gcc/g++是经过以下几个阶段完成编译链接生成可执行文件(以 hello.*为例)

- **预处理阶段(-E)**
  - 一段程序中通常会包含宏定义和头文件包含，预处理阶段就是对这两者进行处理，同时包括了语法检查
  - 该阶段的命令为 gcc **-E** hello.c -o hello.i
  - 生成一个 hello.i 文件。文件 hello.i 文件特别大，**是因为程序将头文件进行了替换，导致文件大的现象**，所以在实际编程过程中，如果用不到的头文件就不需要包含在程序中，否则会造成时间和空间的浪费。
- **生成汇编文件(-S)**
  - 对预处理文件进行汇编生成汇编文件
  - 该命令为：gcc **-S** hello.i -o hello.s

- **由汇编文件生成目标(.o 文件)(-c)**
  - 对汇编文件进一步进行处理，使每个源程序都会生成一个目标文件，扩展名为.o
  - 该命令为 gcc -c hello.s -o hello.o
- **链接目标文件和库函数文件，生成可执行文件**
  - 在链接阶段，需要将目标文件和库函数文件相链接，这里包括静态库和动态库，生成最终的可执行文件
  - 该命令为：gcc hello.o -o hello
- **可以运行可执行文件 hello**

1. 3. 静态/动态库

1、如何使用 g++编译动态库\静态库?如何使用 g++连接非标准库和应用程序?

什么是库呢?

- 简单的说库就是一组已经写好了的函数和变量、是经过编译了的代码，为了提高开发的效率和运行的效率而设计的
- 库可以分为**静态库**和**动态库(共享库)**两类
  - 在 linux 系统中静态库的扩展名为.a，动态库的扩展名是.so
  - 静态库是在每个程序进行链接的时候将库在目标程序中进行一次拷贝当目标程序生成的时候，程序可以脱离库文件单独运行，换言之原来的文件即使删除程序还是会正常工作
  - 共享库可以被多个应用程序共享，实在程序运行的时候进行**动态的加载**，因此对于每个应用程序来说，即使不再使用某个共享库，也不应该将其删除，因为其他的引用程序可能需要这个库

1. 4. 安装

1、Linux 平台

- yum install gcc
- yum install gcc gcc-c++

2. g++编译器详解

2.1. 总格式

gcc[option|filename]…
g++[option|filename]…

2.2. 参数详解

2.2.1. 总体选项(Overall Option)

1、-x language

- 明确指出后面输入文件的语言为 language(而不是从文件名后缀得到的默认选择)
- 这个选项应用于后面所有的输入文件，直到遇着下一个-x'选项
- language 的可选值有：
  - "c"
  - "objective-c"
  - "c-header"
  - "c++"
  - "cpp-output"
  - "assembler"
  - "assembler-with-cpp"

2、-x none

- 关闭任何对语种的明确说明，因此依据文件名后缀处理后面的文件(就象是从未使用过"-x"选项)
- 如果只操作四个阶段(预处理、编译、汇编、连接)中的一部分，可以使用"-x"选项(或文件名后缀)告诉 gcc 从哪里开始，用"-c"，"-S"，"-E"选项告诉 gcc 到哪里结束

3、-c

- 编译或汇编源文件，但是不作连接，编译器输出对应于源文件的目标文件。
- GCC 忽略"-c"选项后面任何无法识别的输入文件(他们不需要编译或汇编)

4、-S

- 编译后即停止，不进行汇编
- 对于每个输入的非汇编语言文件，输出文件是汇编语言文件
- GCC 忽略任何不需要编译的输入文件

5、-E

- 预处理后即停止，不进行编译。**预处理后的代码送往标准输出(除非用"-o"参数指定输出文件或者重定向)**
- GCC 忽略任何不需要预处理的输入文件

6、-o file

- 指定输出文件为 file
- 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
- **由于只能指定一个输出文件**，因此编译多个输入文件时，使用"-o"选项没有意义，除非输出一个可执行文件
- 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"
- 链接文件的目标文件是"source.o"
- 汇编文件是"source.s"
- 预处理后的 C 源代码送往标准输出

7、-v：(在标准错误)显示执行编译阶段的命令，同时显示编译器驱动程序、预处理器、编译器的版本号

2.2.2. 与 gcc 编译器版本有关的选项

1、-std=gnu++11：支持 C++11 新特性需要的参数

2、-pthread：支持 C++11 的多线程参数，否则会被 Linux 内核拒绝

2.3. 静态链接库的生成与使用

1、假设需要生成一个矩阵运算的静态链接库

- 头文件如下：Complex.h Matrix.h Function.h Exception.h
- 源文件如下：Complex.cpp Matrix.cpp

2、生成步骤

- g++ -std=gnu++11 -c Matrix.cpp Complex.cpp
- ar -crv libmatrix.a Matrix.o Complex.o
- 于是会生成一个名为 libmatrix.a 的静态链接库

3、使用步骤

- 将该类库的头文件(Complex.h Matrix.h Function.h Exception.h)以及静态链接库文件(libmatrix.a)放到需要使用的目录下
- g++ -std=gnu++11 -o test main.cpp libmatrix.a
- 直接生成一个可执行文件 test

3. 更新 gcc

1、步骤

- 下载 gcc 源码文件(tar.bz2、tar.gz)等，这里**以"gcc-5.3.0.tar.bz2"为例**
- tar -jxv -f gcc-5.3.0.tar.bz2 <==在任意路径，解压缩
- cd gcc-5.3.0/ <==进入该路径
- ./contrib/download_prerequisites <==下载依赖项
- cd .. <==退回上层路径
- mkdir gcc-build-5.3.0
- cd gcc-build-5.3.0 <==进入输出目录，之后的命令就在该路径下执行
- ../gcc-5.3.0/configure -enable-checking=release -enable-languages=c,c++ \
  > -disable-multilib
- make -j 15 <==编译，可能要 1-3 个小时，CPU 占用率可能 100%，等吧
- make install <==安装

2、重新建立连接文件

- 安装完后，动态库文件：
  - /usr/local/lib64/libstdc++.so.6.0.21
  - 反正找到序号最大的
  - 如果不在此路径
    - updatedb
    - locate libstdc++.so.6 <==查找路径
- 将找到的 libstdc++.so.6.0.21 拷贝到/usr/lib64 当中
  - cp /usr/local/lib64/libstdc++.so.6.0.21 /usr/lib64
- 删除原来的软连接
  - cd /usr/lib64
  - rm libstdc++.so.6
- 将默认库软连接指向最新动态库
  - ln -s libstdc++.so.6.0.21 libstdc++.so.6

4. Makefile

1、makefile 关系到了整个工程的编译规则。一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile 定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 makefile 就像一个 Shell 脚本一样，其中也可以执行操作系统的命令

4.1. 关于程序的编译和连接

1、一般来说，无论是 C、C++、还是 pas

- 首先要把源文件编译成中间代码文件，在 Windows 下也就是.obj 文件，UNIX 下是.o 文件，即 Object File，这个动作叫做编译(compile)
- 然后再把大量的 Object File 合成执行文件，这个动作叫作链接(link)

2、**编译时**

- 编译器需要的是语法的正确，函数与变量的声明的正确
- 对于后者，通常是你需要告诉编译器头文件的所在位置(头文件中应该只是声明，而定义应该放在 C/C++文件中)
- 只要所有的语法正确，编译器就可以编译出中间目标文件
- 一般来说，每个源文件都应该对应于一个中间目标文件(.o 文件或是.obj 文件)

3、**链接时**

- 主要是链接函数和全局变量，所以，我们可以使用这些中间目标文件(.o 文件或是.obj 文件)来链接我们的应用程序
- 链接器并不管函数所在的源文件，只管函数的中间目标文件(Object File)
- 在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链接时需要明显地指出中间目标文件名，这对于编译很不方便
- 所以，我们要给中间目标文件打个包，在 Windows 下这种包叫"库文件(Library File)"，也就是.lib 文件，在 UNIX 下，是 Archive File，也就是.a 文件

4、**总结**

- 源文件首先会生成中间目标文件
- 再由中间目标文件生成执行文件
- 在编译时，编译器只检测程序语法，和函数、变量是否被声明
- 如果函数未被声明，编译器会给出一个警告，但可以生成 Object File
- 而在链接程序时，链接器会在所有的 Object File 中找寻函数的实现，如果找不到，那到就会报链接错误码(Linker Error)
  - 在 VC 下，这种错误一般是：Link 2001 错误，意思说是说，链接器未能找到函数的实现。你需要指定函数的 Object File

4.2. Makefile 介绍

4.2.1. Makefile 的规则

1、Makefile 的规则大致如下

```
target ... : prerequisites ...
command
```

...
...

- target 也就是一个目标文件，可以是 Object File，也可以是执行文件。还可以是一个标签(Label)
- prerequisites 就是，要生成那个 target 所需要的文件或是目标
- command 也就是 make 需要执行的命令(任意的 Shell 命令)
- 这是一个文件的依赖关系，**也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中**
- **说白一点就是说，prerequisites 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容**

2、以一个例子介绍

```
edit : main.o kbd.o command.o display.o /
insert.o search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o /
insert.o search.o files.o utils.o

main.o : main.c defs.h
cc -c main.c

kbd.o : kbd.c defs.h command.h
cc -c kbd.c

command.o : command.c defs.h command.h
cc -c command.c

display.o : display.c defs.h buffer.h
cc -c display.c

insert.o : insert.c defs.h buffer.h
cc -c insert.c

search.o : search.c defs.h buffer.h
cc -c search.c

files.o : files.c defs.h buffer.h command.h
cc -c files.c

utils.o : utils.c defs.h
cc -c utils.c

clean :
rm edit main.o kbd.o command.o display.o /
insert.o search.o files.o utils.o
```

- 该目录下直接输入命令"make"就可以生成执行文件 edit。如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下"make clean"就

可以了

- 在这个 makefile 中，目标文件(target)包含：执行文件 edit 和中间目标文件(*.o)，依赖文件(prerequisites)就是冒号后面的那些.c 文件和.h 文件。每一个.o 文件都有一组依赖文件，而这些.o 文件又是执行文件 edit 的依赖文件。依赖关系的实质上就是说明了目标文件是由哪些文件生成的，换言之，目标文件是哪些文件更新的
- **在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个 Tab 键作为开头**
- **make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期要比 targets 文件的日期要新，或者 target 不存在的话，那么，make 就会执行后续定义的命令**
  - 不会递归地比较 **prerequisites**
  - 假设有如下一段
    main.o:main.cpp
    [tab] gcc -o main.o main.cpp
  - 如果 main.cpp 包含了头文件 a.h，且 main.o 已经存在，那么你改变 a.h 并不会使得执行 make 时重新生成 main.o
  - 要想根据 a.h 的新旧程度来决定是否重新执行 command，必须将 a.h 写在依赖项之中
- clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 lable 一样，其冒号后什么也没有，那么，make 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后明显得指出这个 lable 的名字。这样的方法非常有用，我们可以在一个 makefile 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等

4.2.2. make 是如何工作的

1、在默认的方式下，也就是我们只输入 make 命令，那么

- make 会在当前目录下找名字叫"Makefile"或"makefile"的文件
- 如果找到，它会找文件中的第一个目标文件(target)，在上面的例子中，他会找到"edit"这个文件，并把这个文件作为最终的目标文件
- 如果 edit 文件不存在，或是 edit 所依赖的后面的.o 文件的文件修改时间要比 edit 这个文件新，那么，他就会执行后面所定义的命令来生成 edit 这个文件
- 如果 edit 所依赖的.o 文件也不存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件(这有点像一个堆栈的过程)
- 当然，你的 C 文件和 H 文件是存在的啦，于是 make 会生成.o 文件，然后再用.o 文件生命 make 的终极任务，也就是执行文件 edit 了

2、这就是整个 make 的依赖性，make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件

- 在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么 make 就会直接退出，并报错

- 而对于所定义的命令的错误，或是编译不成功，make 根本不理。make 只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦
- 像 clean 这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要 make 执行。即命令"make clean"，以此来清除所有的目标文件，以便重编译

4.2.3. makefile 中使用变量

1、还是以上述例子来介绍，如下是片段

```
edit : main.o kbd.o command.o display.o /
insert.o search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o /
insert.o search.o files.o utils.o
```

- 我们可以看到.o 文件的字符串被重复了两次
- 如果我们的工程需要加入一个新的[.o]文件，那么我们需要在两个地方加(应该是三个地方，还有一个地方在 clean 中)
- 当然，我们的 makefile 并不复杂，所以在两个地方加也不累，但如果 makefile 变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败
- **所以，为了 makefile 的易维护，在 makefile 中我们可以使用变量**。makefile 的变量也就是一个字符串，理解成 C 语言中的宏可能会更好

2、例如，我们在 makefile 一开始就这样定义

```
objects = main.o kbd.o command.o display.o /
insert.o search.o files.o utils.o
```

- 于是，我们就可以很方便地在我们的 makefile 中以"$(objects)"的方式来使用这个变量了
- 于是我们的改良版 makefile 就变成下面这个样子

```
objects = main.o kbd.o command.o display.o /
insert.o search.o files.o utils.o

edit : $(objects)
cc -o edit $(objects)
main.o : main.c defs.h
cc -c main.c
kbd.o : kbd.c defs.h command.h
cc -c kbd.c
command.o : command.c defs.h command.h
cc -c command.c
display.o : display.c defs.h buffer.h
cc -c display.c
insert.o : insert.c defs.h buffer.h
cc -c insert.c
search.o : search.c defs.h buffer.h
cc -c search.c
files.o : files.c defs.h buffer.h command.h
```

```
cc -c files.c
utils.o : utils.c defs.h
cc -c utils.c
clean :
rm edit $(objects)
```

- 如果有新的.o 文件加入，我们只需简单地修改一下 objects 变量就可以了

4.2.4. 让 maike 自动推导

1、GNU 的 make 很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的 make 会自动识别，并自己推导命令

2、只要 make 看到一个.o 文件，它就会自动的把.c 文件加在依赖关系中

- 如果 make 找到一个 whatever.o，那么 whatever.c，就会是 whatever.o 的依赖文件
- 并且 cc -c whatever.c 也会被推导出来，于是，我们的 makefile 再也不用写得这么复杂
- 我们的是新的 makefile 又出炉了

```
objects = main.o kbd.o command.o display.o /
insert.o search.o files.o utils.o

edit : $(objects)
cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
rm edit $(objects)
```

- ".PHONY"表示：clean 是个伪目标文件

4.2.5. 清空目标文件的规则

1、每个 Makefile 中都应该写一个清空目标文件(.o 和执行文件)的规则，这不仅便于重编译，也很利于保持文件的清洁。这是一个"修养"

- 一般的风格都是

```
clean:
rm edit $(objects)
```

- 更为稳健的做法是

```
.PHONY : clean
clean :
```

-rm edit $(objects)

- .PHONY 意思表示 clean 是一个"伪目标"
- 而在 rm 命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事

➢ 不成文的规矩是：clean 从来都是放在文件的最后