一、GNU/Linux、GPL

GNU计划，又称革奴计划，目标是创建一套完全自由的操作系统，其内容软件完全以[GPL](https://baike.baidu.com/item/GPL)方式发布。GNU操作系统是[GNU计划](https://baike.baidu.com/item/GNU%E8%AE%A1%E5%88%92)的主要目标，名称来自GNU's Not Unix!的[递归缩写](https://baike.baidu.com/item/%E9%80%92%E5%BD%92%E7%BC%A9%E5%86%99" \t "https://baike.baidu.com/item/gnu/_blank)，因为GNU的设计类似Unix，但它不包含具著作权的Unix代码。作为操作系统，GNU的发展仍未完成，其中最大的问题是具有完备功能的内核尚未被开发成功。GNU的内核，称为[Hurd](https://baike.baidu.com/item/Hurd" \t "https://baike.baidu.com/item/gnu/_blank)，是[自由软件基金会](https://baike.baidu.com/item/%E8%87%AA%E7%94%B1%E8%BD%AF%E4%BB%B6%E5%9F%BA%E9%87%91%E4%BC%9A" \t "https://baike.baidu.com/item/gnu/_blank)发展的重点，但是其发展尚未成熟。在实际使用上，多半使用[Linux内核](https://baike.baidu.com/item/Linux%E5%86%85%E6%A0%B8)、[FreeBSD](https://baike.baidu.com/item/FreeBSD)等替代方案，作为系统核心，其中主要的操作系统是Linux的发行版。[Linux](https://baike.baidu.com/item/Linux" \t "https://baike.baidu.com/item/gnu/_blank)操作系统包涵了[Linux内核](https://baike.baidu.com/item/Linux%E5%86%85%E6%A0%B8" \t "https://baike.baidu.com/item/gnu/_blank)与其他自由软件项目中的GNU组件和软件，故被称为[GNU/Linux](https://baike.baidu.com/item/GNU/Linux" \t "https://baike.baidu.com/item/gnu/_blank)（见[GNU/Linux命名争议](https://baike.baidu.com/item/GNU/Linux%E5%91%BD%E5%90%8D%E4%BA%89%E8%AE%AE" \t "https://baike.baidu.com/item/gnu/_blank)）。

GPL:是软件通用公共许可证，为保证GNU软件可以自由地“使用、复制、修改和发布”，所有GNU软件都在一份在禁止其他人添加任何限制的情况下授权所有权利给任何人的协议条款，即GNU通用公共许可证。GPL由斯托曼撰写，用于[GNU计划](https://baike.baidu.com/item/GNU%E8%AE%A1%E5%88%92)。它以GNU Emacs、[GDB](https://baike.baidu.com/item/GDB" \t "https://baike.baidu.com/item/GPL/_blank)、[GCC](https://baike.baidu.com/item/GCC" \t "https://baike.baidu.com/item/GPL/_blank)的许可证的早期版本为蓝本。这些许可证都包含有一些GPL的版权思想，但仅只针对特定程序。斯托曼的目标就是创造出一种四海之内皆可使用的许可证，这样就能为许多源代码共享计划带来福音。GPL版本1就这样，在1989年1月诞生。到1990年时，因为一些[共享库](https://baike.baidu.com/item/%E5%85%B1%E4%BA%AB%E5%BA%93" \t "https://baike.baidu.com/item/GPL/_blank)而出现了对比GPL更宽松的许可证的需求。所以当GPL版本2在1991年6月发布时，另一许可证——库通用许可证（Library General Public License，简称LGPL）也随之发布，并记作“版本2”以示对GPL的补充。版本号在LGPL版本2.1发布时不再相同，而LGPL也被重命名为[GNU宽通用公共许可证](https://baike.baidu.com/item/GNU%E5%AE%BD%E9%80%9A%E7%94%A8%E5%85%AC%E5%85%B1%E8%AE%B8%E5%8F%AF%E8%AF%81" \t "https://baike.baidu.com/item/GPL/_blank)（Lesser General Public License）以体现GNU哲学观。

Linux的历史是和GNU紧密联系在一起的。从1983年开始的GNU计划致力于开发一个自由并且完整的类Unix操作系统，包括软件开发工具和各种应用程序。到1991年Linux内核发布的时候**，GNU已经几乎完成了除了系统内核之外的各种必备软件的开发。Linus Torvalds(linux之父：开发Linux内核、Git)和其他开发人员编写出了与UNIX兼容的Linux操作系统内核并在GPL条款下发布，故GNU组件可以运行于Linux内核之上。**整个linux的内核是基于GNU通用公共许可，也就是GPL(General Pubic License)的，但是Linux内核并不是GNU 计划的一部分，GNU计划自己的内核Hurd依然在开发中，目前已经发布Beta版本。现在许多UNIX系统上也安装了GNU软件，因为GNU软件的质量比之前UNIX的软件还要好。GNU工具还被广泛地移植到Windows和Mac OS上。

**Linux只是一个操作系统内核而已，而GNU提供了大量的自由软件来丰富在其之上各种应用程序。**

严格来讲，Linux这个词本身只表示Linux内核，我们常说的Linux，准确地来讲，应该是叫“GNU/Linux”操作系统。但在实际上人们已经习惯了用Linux来形容整个基于Linux内核，并且使用GNU 工程各种工具和数据库的操作系统。基于这些组件的Linux软件被称为Linux发行版。一般来讲，一个Linux发行套件包含大量的软件，比如软件开发工具，数据库，Web服务器（例如Apache)，X Window，桌面环境（比如GNOME和KDE），办公套件（比如OpenOffice.org），等等。  
绝大多数基于Linux内核的操作系统使用了大量的GNU软件，包括了一个shell程序、工具、程序库、编译器及工具，还有许多其他程序，例如Emacs。正是由于Linux使用了许多GNU程序，GNU计划的开创者Richard Stallman博士提议将Linux操作系统改名为GNU/Linux。但有些人只把操作系统叫做"Linux"。  
有部分Linux套件，包括了Debian，采用了“GNU/Linux”的称呼。但大多数商业Linux套件依然将操作系统称为Linux。有些人也认为“操作系统”一词指的应该只是系统的内核，其他程序都只能算是应用软件，这么一来，该操作系统的内核应叫Linux，而Linux套件是在Linux内核的基础上加入各种GNU工具。

二、gcc/g++

GCC（GNU Compiler Collection，**GNU编译器套件**），是由 GNU 开发的编程语言编译器。它是以[GPL](https://baike.baidu.com/item/GPL" \t "https://baike.baidu.com/item/gcc/_blank)许可证所发行的自由软件，也是 GNU计划的关键部分。GCC原本作为GNU操作系统的官方编译器，现已被大多数**类**[**Unix**](https://baike.baidu.com/item/Unix)**操作系统（如[Linux](https://baike.baidu.com/item/Linux" \t "https://baike.baidu.com/item/gcc/_blank)**、[BSD](https://baike.baidu.com/item/BSD" \t "https://baike.baidu.com/item/gcc/_blank)、[Mac OS X](https://baike.baidu.com/item/Mac OS X" \t "https://baike.baidu.com/item/gcc/_blank)等）采纳为标准的[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "https://baike.baidu.com/item/gcc/_blank)，GCC同样适用于微软的[Windows](https://baike.baidu.com/item/Windows" \t "https://baike.baidu.com/item/gcc/_blank)。**GCC 原名为 GNU C语言编译器**（GNU C Compiler），因为它原本只能处理C[语言](https://baike.baidu.com/item/C%E8%AF%AD%E8%A8%80)，GCC 很快地扩展，变得可处理 [C++](https://baike.baidu.com/item/C++)。后来又扩展能够支持更多编程语言，如[Fortran](https://baike.baidu.com/item/Fortran" \t "https://baike.baidu.com/item/gcc/_blank)、[Pascal](https://baike.baidu.com/item/Pascal" \t "https://baike.baidu.com/item/gcc/_blank)、[Objective-C](https://baike.baidu.com/item/Objective-C" \t "https://baike.baidu.com/item/gcc/_blank)、[Java](https://baike.baidu.com/item/Java" \t "https://baike.baidu.com/item/gcc/_blank)、[Ada](https://baike.baidu.com/item/Ada/5606819" \t "https://baike.baidu.com/item/gcc/_blank)、[Go](https://baike.baidu.com/item/Go/953521" \t "https://baike.baidu.com/item/gcc/_blank)以及各类[处理器架构](https://baike.baidu.com/item/%E5%A4%84%E7%90%86%E5%99%A8%E6%9E%B6%E6%9E%84" \t "https://baike.baidu.com/item/gcc/_blank)上的[汇编语言](https://baike.baidu.com/item/%E6%B1%87%E7%BC%96%E8%AF%AD%E8%A8%80/61826)等，所以改名GNU编译器套件（GNU Compiler Collection）。

**gcc/g++ 两者均可编译c和c++代码**

区别： (1)后缀为.c的，gcc把它当作是C程序，而g++当作是[c++](https://www.baidu.com/s?wd=c++&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)程序； 后缀为.cpp的，两者都会认为是[c++](https://www.baidu.com/s?wd=c++&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)程序，注意，虽然[c++](https://www.baidu.com/s?wd=c++&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)是c的超集，但是两者对语法的要求是有区别的，C++的语法规则更加严谨一些。

1. 在编译阶段时，g++会自动调用gcc，故两者是等价的(仅gcc用来编译)。对于c++代码，因为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，为了统一起见，干脆编译/链接统统用g++了，这就给人一种错觉，好像cpp程序只能用g++似的。
2. 编译可用：gcc/g++(会自动调用gcc); 链接可用：g++/gcc -lstdc++;

虽然我们称GCC是[C语言](https://baike.baidu.com/item/C%E8%AF%AD%E8%A8%80" \t "https://baike.baidu.com/item/gcc/_blank)的[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)，但使用**gcc由C语言[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81" \t "https://baike.baidu.com/item/gcc/_blank)文件生成**[**可执行文件**](https://baike.baidu.com/item/%E5%8F%AF%E6%89%A7%E8%A1%8C%E6%96%87%E4%BB%B6)**的过程不仅仅是编译的过程，而是要经历四个相互关联的步骤∶**[**预处理**](https://baike.baidu.com/item/%E9%A2%84%E5%A4%84%E7%90%86)**(也称[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91" \t "https://baike.baidu.com/item/gcc/_blank)，Preprocessing)、[编译](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91" \t "https://baike.baidu.com/item/gcc/_blank)(Compilation)、[汇编](https://baike.baidu.com/item/%E6%B1%87%E7%BC%96" \t "https://baike.baidu.com/item/gcc/_blank)(Assembly)和链接(Linking)**。命令gcc首先调用cpp进行预处理，在预处理过程中，对源代码文件中的文件包含(include)、预编译语句(如[宏](https://baike.baidu.com/item/%E5%AE%8F" \t "https://baike.baidu.com/item/gcc/_blank)定义define等)进行分析。接着调用cc1进行编译，这个阶段根据输入文件生成以.i为后缀的目标文件。汇编过程是针对汇编语言的步骤，调用as进行工作，一般来讲，.S为后缀的汇编语言[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81" \t "https://baike.baidu.com/item/gcc/_blank)文件和汇编、.s为后缀的汇编语言文件经过[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91" \t "https://baike.baidu.com/item/gcc/_blank)和[汇编](https://baike.baidu.com/item/%E6%B1%87%E7%BC%96)之后都生成以.o为后缀的目标文件。当所有的[目标文件](https://baike.baidu.com/item/%E7%9B%AE%E6%A0%87%E6%96%87%E4%BB%B6" \t "https://baike.baidu.com/item/gcc/_blank)都生成之后，gcc就调用ld来完成最后的关键性工作，这个阶段就是连接。在连接阶段，所有的目标文件被安排在可执行程序中的恰当的位置，同时，该程序所调用到的[库函数](https://baike.baidu.com/item/%E5%BA%93%E5%87%BD%E6%95%B0" \t "https://baike.baidu.com/item/gcc/_blank)也从各自所在的档案库中连到合适的地方。

**预[编译](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91" \t "https://baike.baidu.com/item/gcc/_blank)过程：**这个过程处理[宏定义](https://baike.baidu.com/item/%E5%AE%8F%E5%AE%9A%E4%B9%89)和include，去除注释，不会对语法进行检查。

**编译过程：**这个阶段，检查语法，生成[汇编](https://baike.baidu.com/item/%E6%B1%87%E7%BC%96)代码。

**汇编过程：**这个阶段，生成ELF格式的目标代码。(可执行与可链接格式)

[**链接**](https://baike.baidu.com/item/%E9%93%BE%E6%8E%A5)**过程：**链接过程。生成[可执行代码](https://baike.baidu.com/item/%E5%8F%AF%E6%89%A7%E8%A1%8C%E4%BB%A3%E7%A0%81)。链接分为两种，一种是[静态链接](https://baike.baidu.com/item/%E9%9D%99%E6%80%81%E9%93%BE%E6%8E%A5)，另外一种是[动态链接](https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E9%93%BE%E6%8E%A5)。使用静态链接的好处是，依赖的动态链接库较少，对动态链接库的版本不会很敏感，具有较好的兼容性；缺点是生成的程序比较大。使用动态链接的好处是，生成的程序比较小，占用较少的内存。

gcc所遵循的部分约定规则：

.c为后缀的文件，C语言[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81" \t "https://baike.baidu.com/item/gcc/_blank)文件；

.a为后缀的文件，是由[目标文件](https://baike.baidu.com/item/%E7%9B%AE%E6%A0%87%E6%96%87%E4%BB%B6" \t "https://baike.baidu.com/item/gcc/_blank)构成的档案库文件；

.C，.cc或.cxx 为后缀的文件，是C++源代码文件且必须要经过[预处理](https://baike.baidu.com/item/%E9%A2%84%E5%A4%84%E7%90%86" \t "https://baike.baidu.com/item/gcc/_blank)；

.h为后缀的文件，是程序所包含的[头文件](https://baike.baidu.com/item/%E5%A4%B4%E6%96%87%E4%BB%B6" \t "https://baike.baidu.com/item/gcc/_blank)；

.i 为后缀的文件，是C源代码文件且不应该对其执行预处理；

.ii为后缀的文件，是C++源代码文件且不应该对其执行预处理；

.m为后缀的文件，是Objective-C源代码文件；

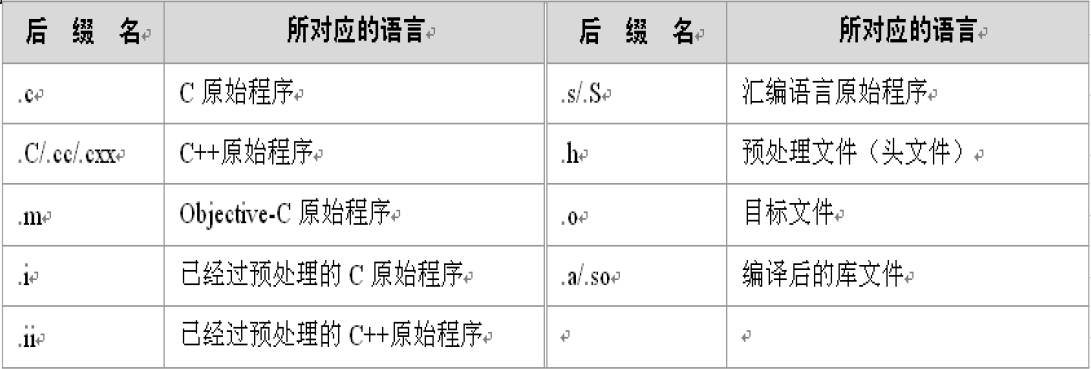
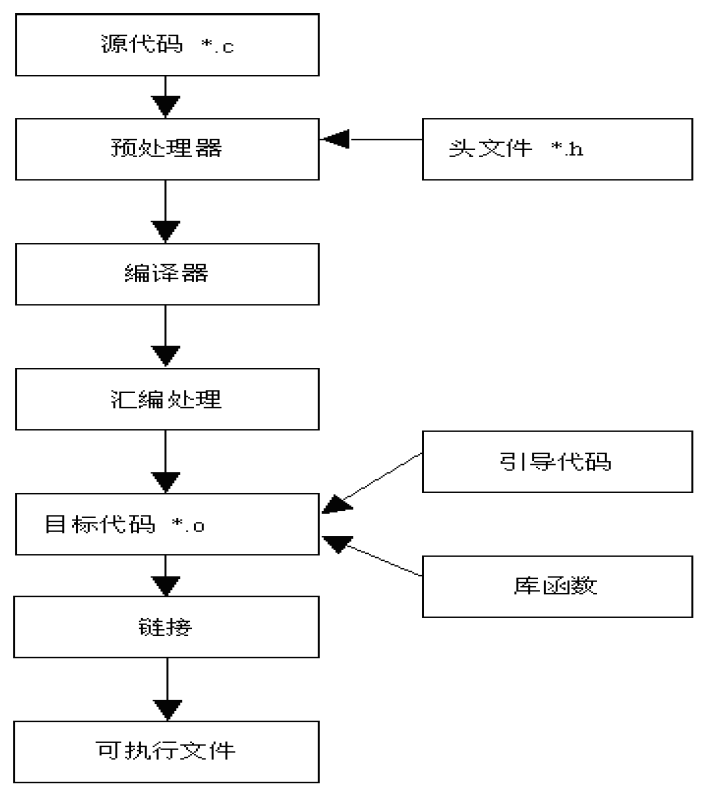
.mm为后缀的文件，是Objective-C++[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81/3587471)文件；

.o为后缀的文件，是[编译](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91" \t "https://baike.baidu.com/item/gcc/_blank)后的目标文件；

.s为后缀的文件，是[汇编](https://baike.baidu.com/item/%E6%B1%87%E7%BC%96" \t "https://baike.baidu.com/item/gcc/_blank)语言源代码文件；

.S为后缀的文件，是经过[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91)的汇编语言源代码文件。

二. **gcc**编译过程：预编译、编译、汇编、链接



**(1)预处理( .i文件)：gcc –E test.c -o test.i**

预处理器对源文件中的所有宏进行替换，即处理所有包含#的文件([宏定义](https://baike.baidu.com/item/%E5%AE%8F%E5%AE%9A%E4%B9%89)和include),不做语法检查。gcc将 .c文件编译成 .i文件。

**(2)编译( .s汇编文件：汇编指令助记符)：gcc -S test.i –o test.s**

gcc将 .i文件编译成 .s汇编文件；检查语法生成汇编代码。

**(3)汇编( .o目标文件：机器码、微指令)：as test.s -o test.o**

as将汇编文件编译成二进制的机器码，可用nm查看: **nm test.o**

**(4)链接(可执行二进制文件：ELF文件)： gcc test.o –o test**

将目标文件和外部符号进行连接，得到一个可执行二进制文件。

**把函数的符号替换成所对应的地址值 (执行函数时便会根据地址跳转过去)。**

#include <stdio.h>

#define NUMBER 1+2

int main()

{

int x = NUMBER \* NUMBER;

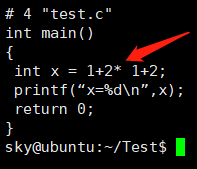
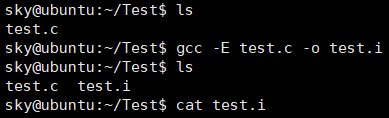
printf(“x=%d\n”,x);

return 0;

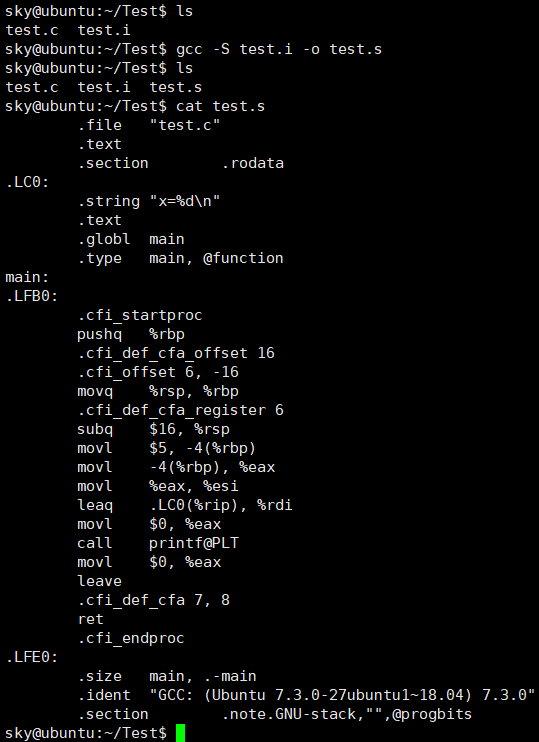
}

(1) 预处理：gcc –E test.c -o test.i 我们用cat查看test.i的内容如下：

int x=1+2\*1+2;我们可以看到，文件中宏定义NUMBER出现的位置被1+2替换掉了，其它的内容保持不变。



(2) 编译：gcc -S test.i –o test.s 通过cat test.s查看test.s的内容为助记符。

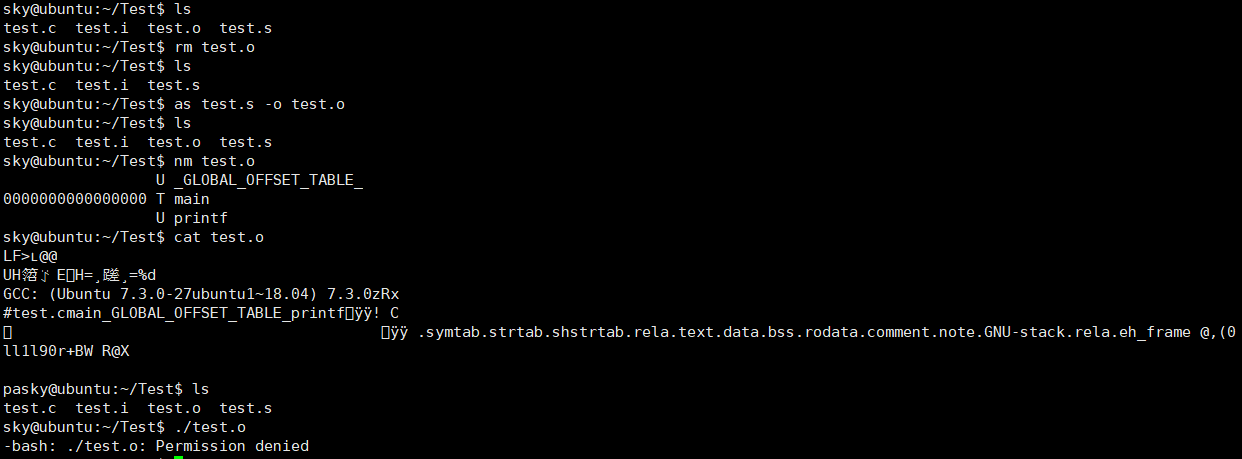


1. 汇编：as test.s -o test.o 利用as将汇编文件编译成机器码。得到输出文件为test.o. test.o中为目标机器上的二进制文件.

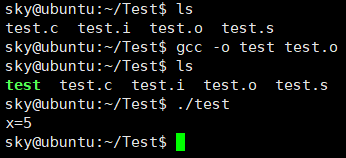
用**nm查看文件中的符号： nm test.o**输出如下：00000000 T main。有的编译器上会显示：00000000 b .bss 00000000 d .data 00000000

t .text U \_\_\_main U \_\_alloca 00000000 T \_main

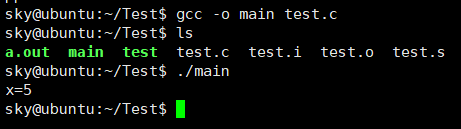
既然已经是 二进制目标文件了，能不能执行呢？试一下./test.o,提示 cannotexecute binary file。 原来 **\_\_\_main前面的U表示这个符号的地址还没有定下来，T表示这个符号属于代码。**



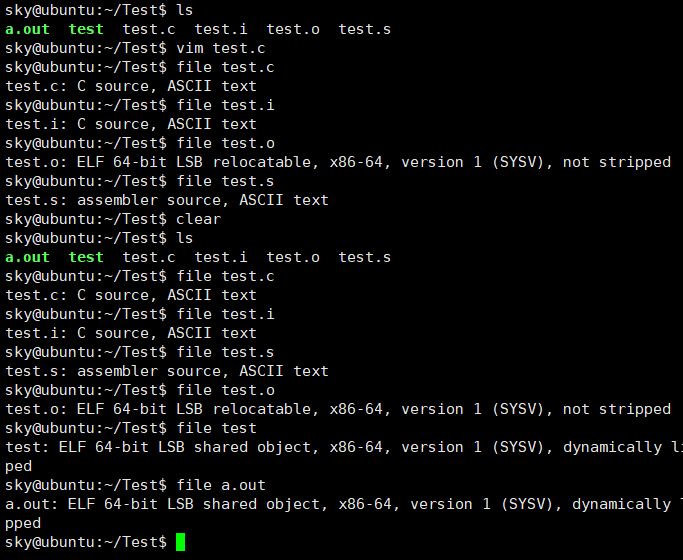
(4)链接：gcc test.o–o test ,将所有的.o文件链接起来生产可执行程序。



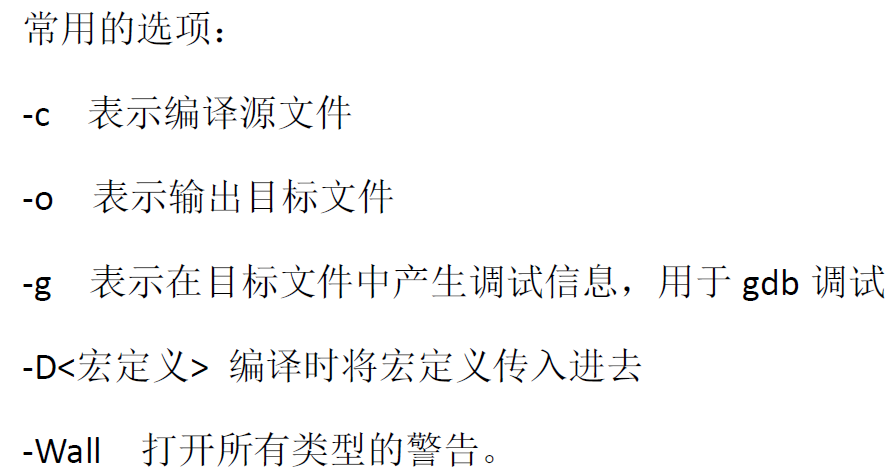
(5) gcc test.c -o test //编译并链接



(6)查看各个文件的文件类型：file命令



**(7) gcc常用选项: 有以下两部分**

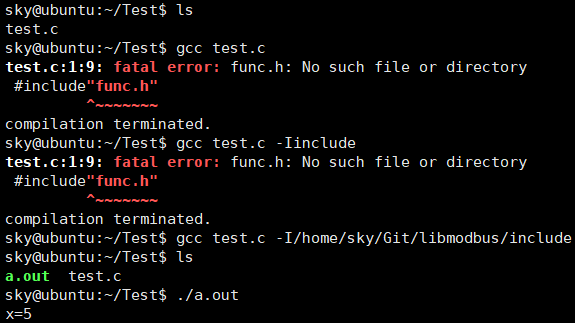




**其中两个重点：**

**1)gcc加 -c :只编译不链接，生成.o目标文件： 用于增量编译！！！**

**实现了预处理、编译、汇编。常用于增量编译，因为预处理、编译、汇编是最耗时的，而链接并不耗时。这样有多个.c文件时，若修改了其中一个.c文件,便只用加 -c选项编译这一个.c文件即可，其他 .c文件不用动。**



**2)gcc加 -Idir(路径) -Idir(路径)... ：(I为大写i)在头文件的搜索路径列表中添加dir目录。 注意：系统默认搜索路径在： /usr/include/目录下**

**如当前.c的头文件不在当前目录下，而在其他路径的include目录中：**

**gcc test.c -I(路径)include (如上图)**

**3)gcc -c main.c //直接生成 .o目标文件**

历史知识： Linux是一套免费使用和自由传播的[类Unix](https://www.baidu.com/s?wd=%E7%B1%BBUnix&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)，是一个**基于POSIX和UNIX的多用户、多任务、支持多线程和多CPU**的[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)。而严格来讲，Linux这个词本身只表示[Linux内核](https://www.baidu.com/s?wd=Linux%E5%86%85%E6%A0%B8&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)，但实际上人们已经习惯了用Linux来形容整个基于[Linux内核](https://www.baidu.com/s?wd=Linux%E5%86%85%E6%A0%B8&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)，并且使用GNU 工程各种工具和数据库的[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)。

Linux是开放源代码的，所以网上会出现各种各样的发行版本，Ubuntu Linux就是其中一种。Ubuntu采用[Linux内核](https://www.baidu.com/s?wd=Linux%E5%86%85%E6%A0%B8&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)，图形界面采用GNOME（Kubuntu使用KDE）。简而言之，**Linux系统是个统称，它有Red Hat、Debian、Suse、Ubuntu等发行版本，它们都是用的Linux内核，都是Linux系统。**

对于ubuntu而言，就是linux操作系统的**具体**，而linux对于ubuntu来说就是他的**抽象**；在linux操作系统中，因为应用程序和管理策略的不同，有多个版本，例如：ubuntu,fedora,redhat,centos等。

四、数据库

**MySQL是关系型数据库**，是持久化存储的，查询检索的话，会涉及到磁盘IO操作，为了提高性能，可以使用缓存技术，而memcached就是内存数据库，数据存储在内存中（当然也可以进行持久化存储），可以用作缓存数据库。用户首先去memcached查询数据，如果未查询到（即缓存未命中），才去MySQL中查询数据，查询到的数据会更新到缓存数据库中，提供给下次可能进行的查询。提高了数据查询方面的性能。

**Redis和memcached都是缓存数据库，可以大大提升高数据量的web访问速度**。但是memcached只是提供了简单的数据结构string，而Redis的value可以是string、list、set、hash、sorted set这些，功能更加强大。web应用中一般采用MySQL+Redis的方式，web应用每次先访问Redis，如果没有找到数据，才去访问MySQL。

Redis是内存数据库，数据保存在内存中，访问速度快。MySQL是关系型数据库，功能强大，存储在磁盘中，数据访问速度慢。像memcached，MongoDB，Redis等，都属于No sql(即非关系型的数据库)系列。

五、Git

**1.工作流**

你的 **本地仓库** 由 git 维护的三棵"树"组成。

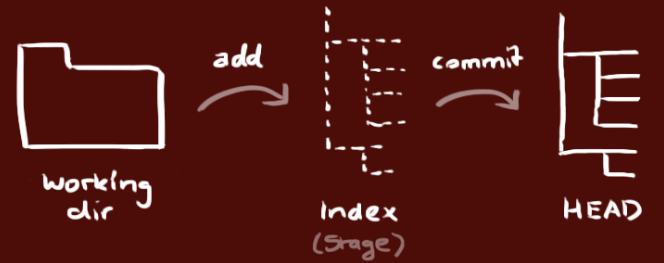
(1)第一个是你的**工作目录**，它持有实际文件；

(2)第二个是 **暂存区（Index）**，它像个缓存区域，临时保存你的改动；

(3)最后是 **HEAD**，它指向你最后一次提交的结果。

你可以提出更改（把它们**添加到暂存区**），使用如下命令：  
git add <filename>  
git add \*  
这是 git 基本工作流程的第一步；

使用如下命令以实际提交改动：  
git commit -m "代码提交信息"  
现在，已经把 **改动提交到HEAD** 了，但是还没到你的远端仓库。



### 2.推送改动

你的改动现在已经在**本地仓库的HEAD** 中了，再执行命令将这些改动提交到远端仓库：

git push origin master //**将这些改动提交到远端仓库**可以把 ***master* 可以换** 成其他想要推送的任何分支。 

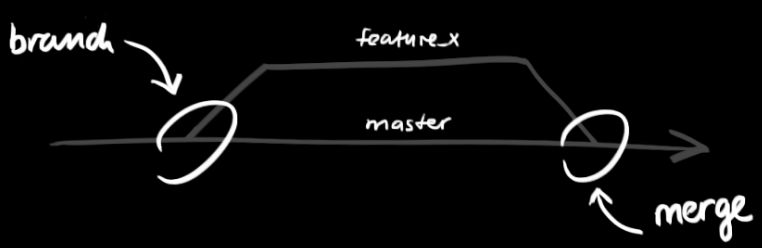
如果你还没有克隆现有仓库，并欲 **将你的仓库连接到某个远程服务器**，你可以使用如下命令添加：  
git remote add origin <server> //如此你就能够将改动推送到所添加的服务器上去了

若关联时出现错误：fatal:remote origin already exists,则执行下列语句后再去关联：

git remote rm origin

### 3.分支

分支是用来将特性开发绝缘开来的。 在你创建仓库的时候，*master* 是"默认的"分支。可以在其他分支上进行开发，完成后再将它们合并到主分支上。



**创建**一个叫做"feature\_x"的**分支，并切换**过去：git checkout -b feature\_x

切换回主分支：git checkout master

再把新建的分支删掉：git branch -d feature\_x

除非你将分支推送到远端仓库，不然该分支就是*不为他人所见的*：  
git push origin <branch>

### 4.更新与合并

要**更新你的本地仓库至最新** 改动，执行：  
**git pull**  //在你的工作目录中 *获取（fetch）* 并 *合并（merge）* 远端的改动。

要**合并其他分支到你的当前分支**（例如 master），执行：  
git merge <branch>

在这两种情况下，git 都会尝试去自动合并改动。遗憾的是，这可能并非每次都成功，并可能出现*冲突（conflicts）*。 这时候就需要你修改这些文件来手动合并这些*冲突（conflicts）*。改完之后，你需要执行如下命令以将它们标记为合并成功：  
git add <filename>  
在合并改动之前，你可以使用如下命令**预览差异**：  
git diff <source\_branch> <target\_branch>

### 5.替换本地改动

假如你操作失误， 你可以使用如下命令替换掉本地改动：  
git checkout -- <filename>

此命令会 **用 HEAD 中的最新内容替换掉你的工作目录中的文件**。 已添加到暂存区

的改动以及新文件都不会受到影响。

假如你想 **丢弃本地的所有改动与提交， 可以到服务器上获取最新的版本历史，并将你本地主分支指向它**：  
git fetch origin  
git reset --hard origin/master