GCC 与make 知识点考试复习补充材料

复习指南

* Linux 环境下C编程：第六章 教材内容和课后习题，入6.1、6.2、6.4、6.7、6.8、6.9，吃透嚼烂，上机灵活运用。（gdb的相关内容只需要掌握6.2 6.2.1的内容，课后习题6.5、6.6 可以忽略）。
* PPT对应知识点，吃透嚼烂，上机灵活运用。
* 鸟书 21章所有内容理解；尤其是21.2、21.3和21.5节所有内容，吃透嚼烂，上机灵活运用。
* 以下所有内容，吃透嚼烂，上机灵活运用。

目录

[Linux应用编程之GCC编译器及编译过程 2](#_Toc103195600)

[1 GCC编译器简介 2](#_Toc103195601)

[2什么是编译器 3](#_Toc103195602)

[3交叉编译器 3](#_Toc103195603)

[4为何需使用交叉编译 4](#_Toc103195604)

[5编译的基本流程 4](#_Toc103195605)

[6简单体验一下GCC 5](#_Toc103195606)

[Linux应用编程之GCC常用编译选项 6](#_Toc103195607)

[**GCC编译流程图** 6](#_Toc103195608)

[2**预编译 -E 选项** 6](#_Toc103195609)

[3**编译过程 -S 选项** 7](#_Toc103195610)

[4**汇编过程的 -c 选项** 8](#_Toc103195611)

[5**链接过程** 9](#_Toc103195612)

[6**最后的话** 11](#_Toc103195613)

[静态库制作和使用简介 11](#_Toc103195614)

[2**静态链接库制作过程** 12](#_Toc103195615)

[3**小结** 16](#_Toc103195616)

[动态库制作和使用简介 16](#_Toc103195617)

[2**动态链接库制作过程** 17](#_Toc103195618)

[3**小结** 19](#_Toc103195619)

[Linux应用编程之动态链接与静态链接 20](#_Toc103195620)

[**Windows与Linux动静库对比** 20](#_Toc103195621)

[2**动态和静态链接库** 21](#_Toc103195622)

[3**小结** 22](#_Toc103195623)

[Linux应用编程之动静态链接库辅助技巧(ldd,-static) 23](#_Toc103195624)

[**查看执行文件共享库** 23](#_Toc103195625)

[2**指定链接动态或者静态库** 25](#_Toc103195626)

[3**小结** 27](#_Toc103195627)

[Linux应用编程之动静态混合链接演示 28](#_Toc103195628)

[**链接方式总结** 28](#_Toc103195629)

[2**明确指定混合链接** 29](#_Toc103195630)

[2**实例演示** 30](#_Toc103195631)

[**过程说明:** 30](#_Toc103195632)

[**过程说明:** 31](#_Toc103195633)

[过程说明: 32](#_Toc103195634)

[3**小结** 32](#_Toc103195635)

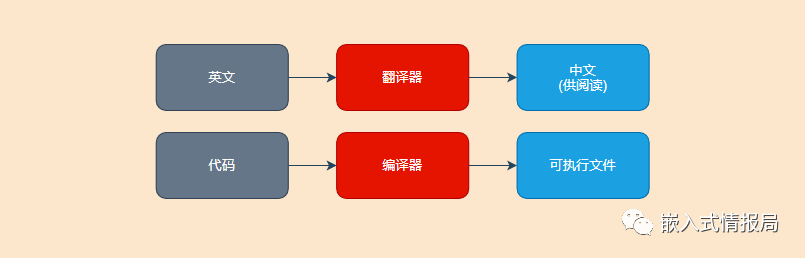
# Linux应用编程之GCC编译器及编译过程

## 1 **GCC编译器简介**

    GCC编译器(GNU C Compiler)是GNU开发的一款多平台编译器，根据名字GCC是一款编译C程序的编译器。但随着该编译器的扩展逐渐支持多平台、多语言的编译套件(GUN Compiler Colloction )。

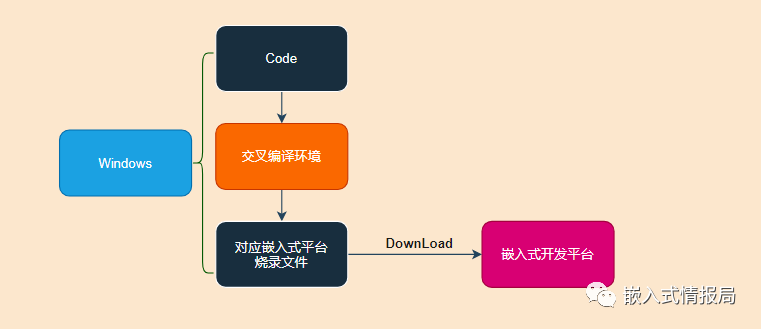
## 2****什么是编译器****

    编译器并不是什么高级的东西，仅仅只是一个根据相应平台要求把用户所编写的代码删除、修改、翻译等等处理后生成对应的可执行文件的一种软件程序，其本质也是属于软件，所以大家可以把编译器看成一个翻译软件。



## 3****交叉编译器****

    在嵌入式Linux开发过程中，比如我们开发ARM平台上的运行程序，一般会在windows平台上进行代码的编译并生成对应的可执行文件烧录到对应的平台中运行，然而这个时候使用编译器就是交叉编译器。



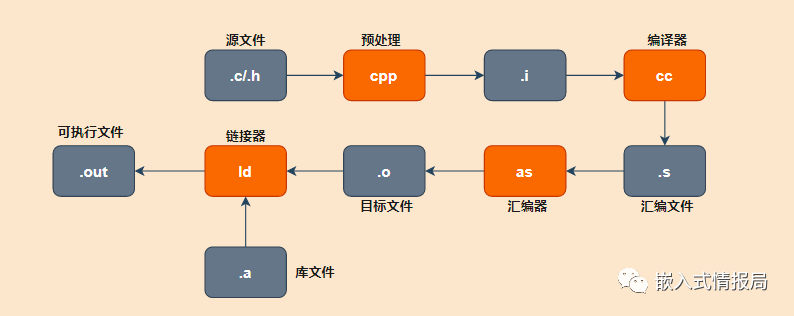
## 4****为何需使用交叉编译****

    嵌入式相比计算机最大的区别个人觉得在于资源分配上，相对而言嵌入式平台的运行速度、内存大小等等都比计算机平台小很多，既然编译器也是软件那就需要占用一定的资源，随着编译器的扩展需要的硬件平台要求也提高了，所以一般的嵌入式平台难以满足编译要求，不过对于资源丰富，性能强劲的平台当然另当别论了。

    所以为了满足在PC上就可以开发嵌入式软件就有了前面所提到的交叉编译环境，编译器可以通过被指定所要运行的目标平台来对相应程序进行编译生成目标平台的可执行文件，当然此时所生成的可执行文件不可在当前环境下运行。

## 5****编译的基本流程****

    既然学习Linux应用编程，那么必定需要将平时所编写的程序通过编译器最终生成可执行文件，下面大致看看GCC如何编译程序的:

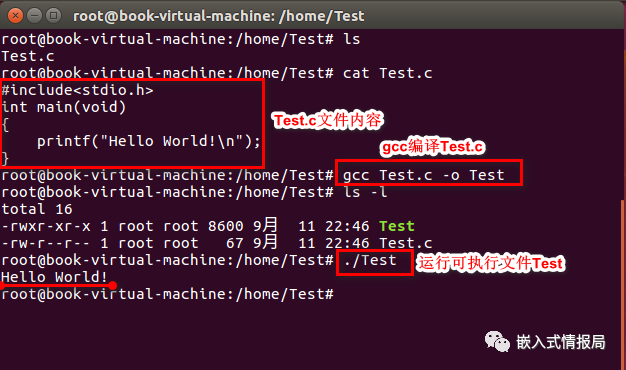


编译过程解析

* 1）预处理阶段：主要对源文件进行文件包含和预处理语句的分析处理；
* 2）编译阶段：把预处理完的.i文件进行一系列词法分析、语法分析、语义分析以及优化后生成相应的汇编代码文件；
* 3）汇编阶段：使用汇编器将汇编代码转换成机器可以执行的指令并生成.o文件；
* 4）链接阶段：使用链接器把所有的目标文件和库文件链接起来放在合适的位置生成.out可执行文件。

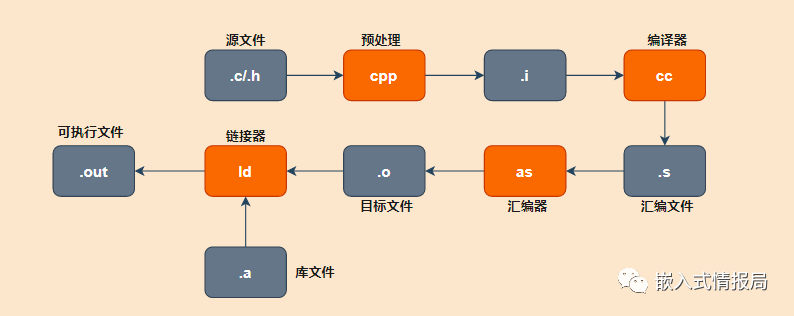
## 6****简单体验一下GCC****

    下面小哥就简单的在ubuntu下面写了一个.c程序并使用gcc编译运行，其实与我们平时在windows环境中编译运行类似，仅仅只是把windows点击按钮换成了对应的命令行罢了。



# Linux应用编程之GCC常用编译选项

**GCC编译流程图**

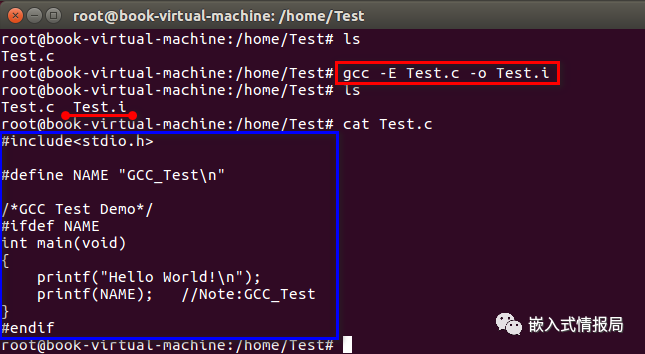


    上一小节我们简单的演示了一下gcc如何生成可执行文件，然而整个编译过程并不是很明朗，于是这里小哥就之前介绍的编译流程图通过GCC的编译选项来逐步进行分解介绍：

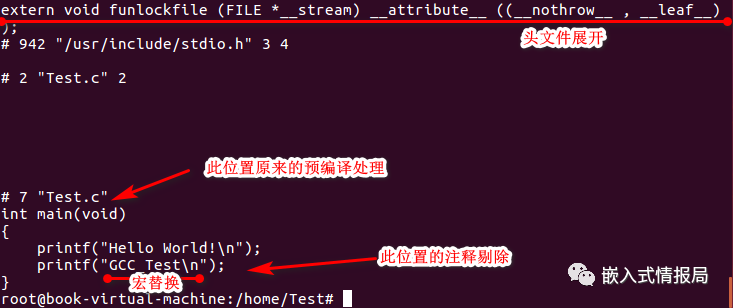
2**预编译 -E 选项**

    GCC的 -E 选项实现源文件到.i文件的预编译处理过程 : 主要用于处理一些带有#的代码等，比如宏#define、条件编译#if，#endif、#pragma；同时也包括注释的剔除，以及include展开和系统预定义宏的替换(如\_\_LINE\_\_,\_\_FILE\_\_)等处理。

    通过执行如下-E选项即可生成对应的.i中间文件:



    下面我们通过使用cat命令来看一下.i文件是否大致完成前面提到的那些预处理工作:

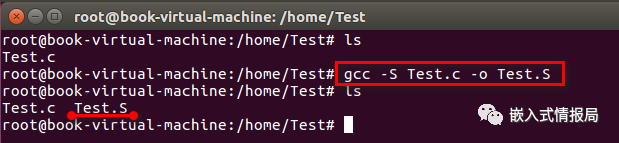


    通过上图与我们了解到的预处理过程是相符合的。

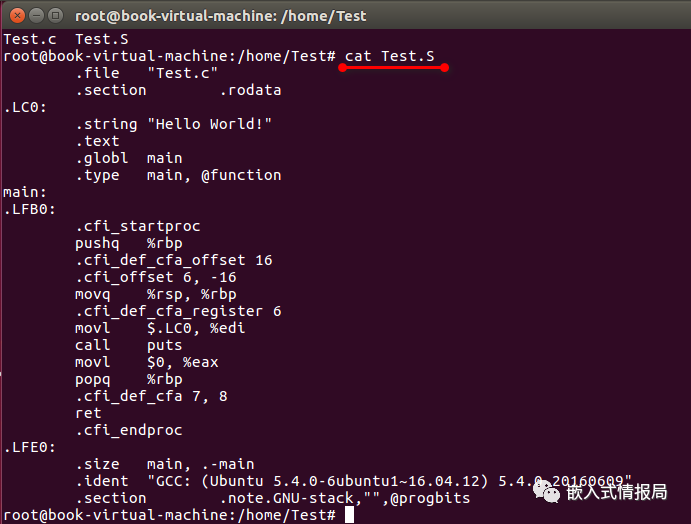
3**编译过程 -S 选项**

    GCC中的-S选项主要是编译生成对应平台的汇编文件，特别是在嵌入式领域对于一些棘手的问题都会通过分析对应的汇编文件来进行定位。

    通过执行-S选项便可以生成对应的.S汇编文件。

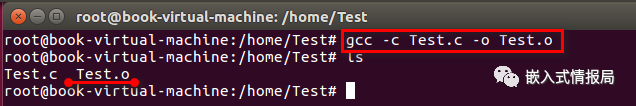


    同样下面我们通过使用cat命令来看看.S文件的大致内容:



4**汇编过程的 -c 选项**

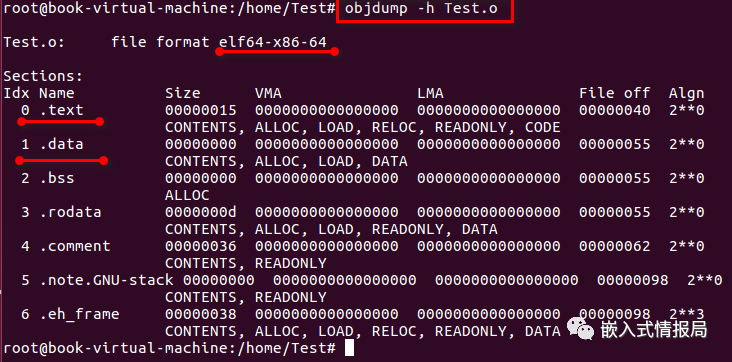
    GCC中的 -c 选项仅仅用于编译生成.o的目标文件，不进行文件的链接。



    最终生成的.o文件是一种中间文件，在Linux中其格式也是ELF，仅仅只是没有进行链接，所以其内容与最终的目标文件几乎是一样的。

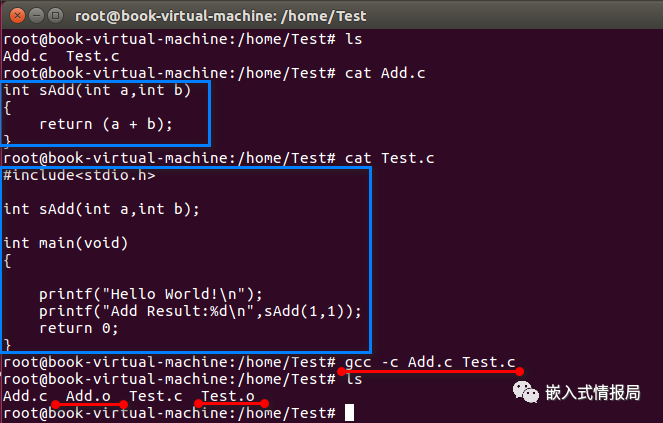
    那么什么是可执行文件呢?当然就是我们常说的由.bss段、.text段、.data段等组成的文件了，当然该文件就不能用常规的文本打开阅读了，而要借助一些命令行分析，如objdump、readelf等。

    下面小哥就通过objdump -h来简单的打印各个段的信息(objdump和readelf命令行还有很多其他选项功能，后面小哥再一起讲解一下):

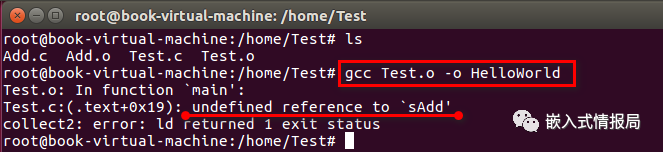


5**链接过程**

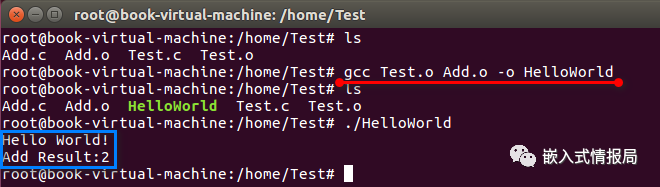
    根据之前的编译流程图，链接过程算是生成可执行文件的最后阶段，限于篇幅，这里小哥暂时不举例与库的链接，以后会专门出一则文章详细说明，这里仅仅用两个.o文件完成最后的链接阶段:



    上图编写了两个.c文件，并分别生成了目标文件.o,假如我们直接用Test.c来完成最后的链接会出现如下结果:



    链接结果报了一个错误，没有定义sAdd函数，正确的做法可执行如下命令:



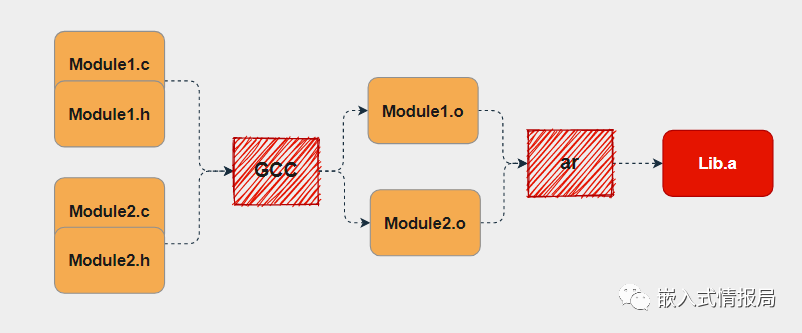
    这样就生成了最终的可执行文件。

6**最后的话**

    大家应该也发现了，每个编译过程的编译选项可以仅仅编译当前过程，也可以从源文件直接编译到当前选项所规定的编译结束阶段。

# 静态库制作和使用简介

    在Linux中制作静态链接库并不复杂，首先小哥画了一张制作简单描述一下制作过程:



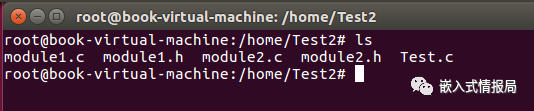
过程简要说明:

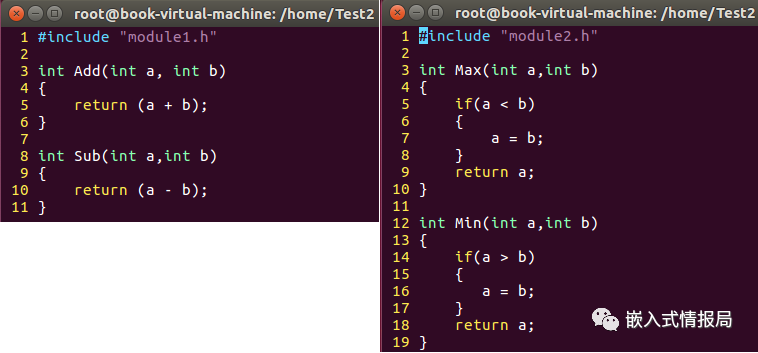
    首先通过GCC把源文件编译生成对应的.o目标文件，然后通过ar命令打包成对应的静态链接文件，最终发布.a和.h文件即可。

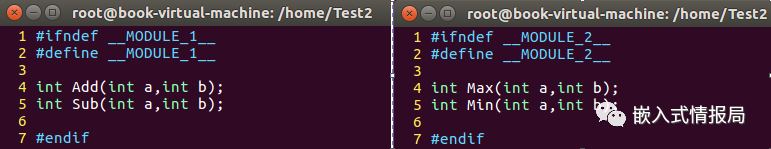
2**静态链接库制作过程**

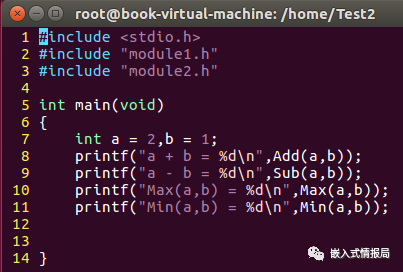
01

**准备源文件**



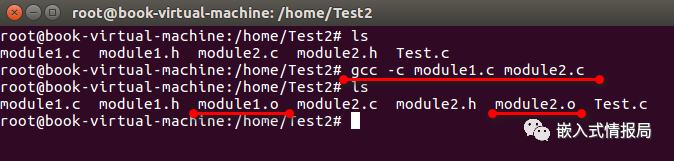






02

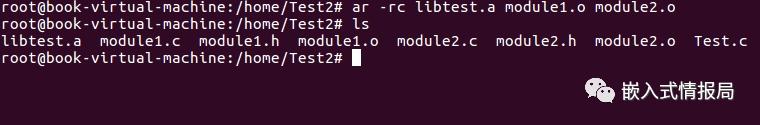
**编译生成目标文件**



    记得这里使用gcc -c选项生成.o目标文件，然后进行下面的ar打包。

03

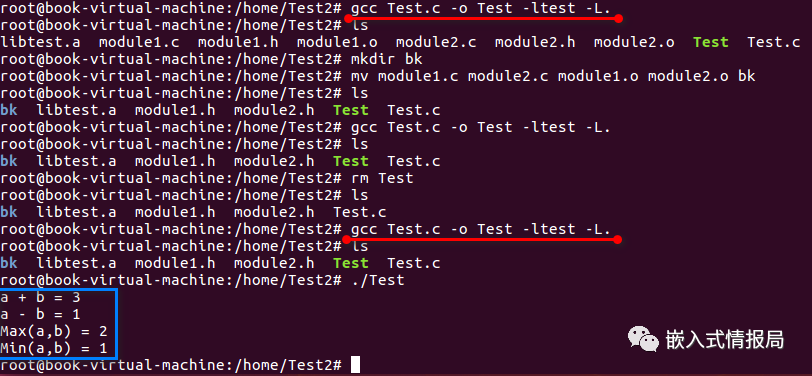
**打包生成对应的.a文件**



        通过使用ar命令行来生成.a文件，其中.a文件的命名形式为:libxxx.a，方便后面进行gcc的链接编译查找。

ar命令行常用选项Linux ar命令用于建立或修改备存文件，或是从备存文件中抽取文件。  
-r ：用于将文件插入库文件中-c ：用于建立库文件-t ：显示库文中所包含文件-d ：删除库文件中的成员文件......其他选项就不过多介绍了

**使用库文件编译**



    第一条命令中有两点需要讲解一下:

1、-ltest 命令选项，用于确定链接库名称。

    因为前面生成的静态链接库为libxxx.a,那么gcc编译在连接静态链接库的时候会自动在-lxxx选项中提取xxx，然后在前添加lib并在后添加.a来搜索链接库。

    比如需要链接的静态库为libTest.a，那么使用gcc链接的选项为-lTest即可。

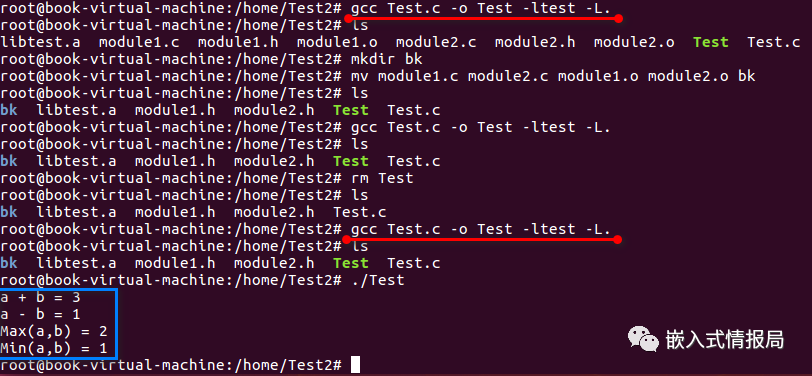
2、-L.命令选项，用于确定库的位置(注：L后面有一个点号)

    默认gcc会链接到系统目录，比如lib目录，而不会在当前目录中查找，这样就会导致编译失败，所以我们使用-L选项进行指定，其中-L.表示在当前目录。

        如果指定其他目录可以采用 -L/home/... 的形式即可。

05

**链接库的使用**



    还是使用该图，小哥第一个没有把源文件和目标文件删除，所以也不能证明是直接链接的静态库文件。

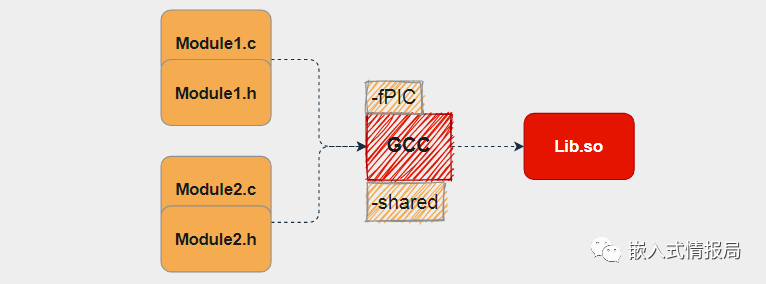
    所以第二步小哥把无关文件转移到了bk文件夹中，然后继续编译链接，成功生成Test可执行文件，并输出正确的结果，同时也说明静态链接库发布.a和对应的.h即可。

3**小结**

    今天主要是跟大家介绍了静态链接库的制作和使用，下一节跟大家演示一下动态链接库的使用。

# 动态库制作和使用简介

    在Linux中制作动态链接库只需要使用好GCC这两个编译选项即可搞定，下面小哥画了一张图简单的说明了下:



过程简要说明:  
    其实链接库的生成过程也可以先分别生成位置无关的目标文件.o文件，然后再创建生成最终的.so动态链接文件。不过gcc可以直接使用这两个编译选项一次性生成。

2**动态链接库制作过程**

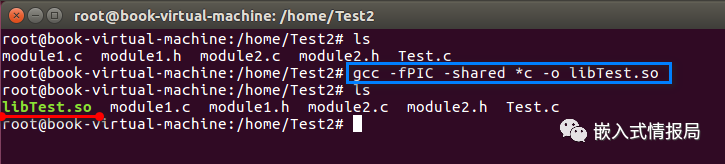
01

**准备源文件**

    这里小哥还是使用昨天的两个module作为例子来讲解动态链接过程，这里就不板书了，可以到上一节静态链接中查看源文件。

02

**编译生成动态链接库**



    这里通过-fPIC和-shared两个选项即可把所有的.c文件生成libxxx.so文件。那下面看一下这两个选项是什么意思:

**>>>>**

**-fPIC选项**

    PIC - Position Independent Code,意思是位置无关码，即加载器把其加载到内存任何位置，其代码均可以正常运行，所以代码中都是使用的是相对地址，这样的特点也就非常满足动态库加载位置变化的特性。  
    同时这样的特点也让动态库被多个应用程序连接的时候不需要copy形成多个副本，大大节省内存。

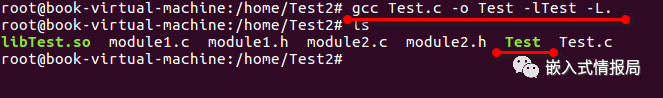
**>>>>**

**-shared选项**

   该选项主要是GCC把目标文件链接成为一个共享库。

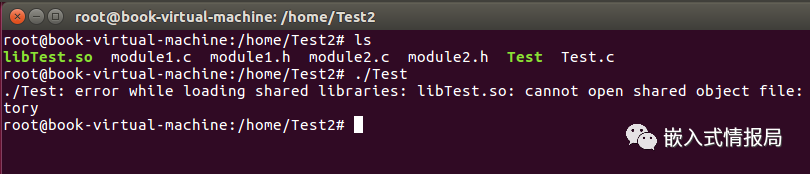
03

**使用库文件编译**



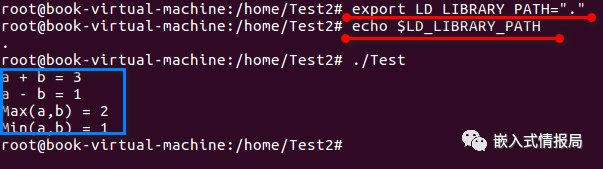
    这里使用共享库来生成可执行文件的用法和前面的静态链接库是类似的，-lxxx（其中的xxx即.so文件去掉lib和后缀的名称），-L.即表示在当前文件中链接共享库。

**动态库的使用**



        如果直接运行可执行文件Test，系统会提示无法打开共享文件，那一般都是由于共享库没有在系统加载器所在指定的路径中。

   一般加载器都会默认在"/lib"、"/usr/lib"后者LD\_LIBRARY\_PATH环境变量指定的路径中搜索动态库，所以如果把.so文件复制到前面两个目录便可执行，不过为了不破坏系统文件，一般都会使用环境变量定义路径。



   上面小哥直接使用export设置了环境变量定义为了当前路径，通过"echo $xxx"便可以查看所设置的路径，当然你也可以使用unset来删除环境变量，然后再次运行程序即可获得跟上次静态链接库一样的结果了。

    当然这里采用了最直接的环境变量设置方法，一般系统注销，该环境变量就会消失，后面小哥会再整理一下环境变量的设置包括如何长久保留环境变量等，比如在用户主目录下的.profile或.bashrc文件配置。

3**小结**

    今天主要是跟大家介绍了动态链接库的制作和使用，下期精彩见。

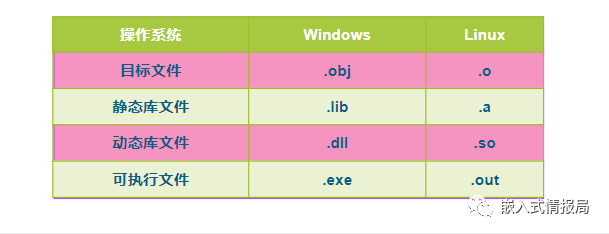
# Linux应用编程之动态链接与静态链接

**Windows与Linux动静库对比**

    小哥觉得学习Linux应该多和Windows对比，对于操作系统其实很多技术和定义都是相似的，所以很多概念和实现都可以对比去理解。

    毕竟一般在平时的生活和开发中接触Windows的时间相对比较多，而Windows的图形化已经让很多人对一些概念形成了一定思维定式，转而进入Linux的命令行使用就会有点抽象。

    下面就Windows和Linux与链接库有关文件进行对比:



**值得注意:**

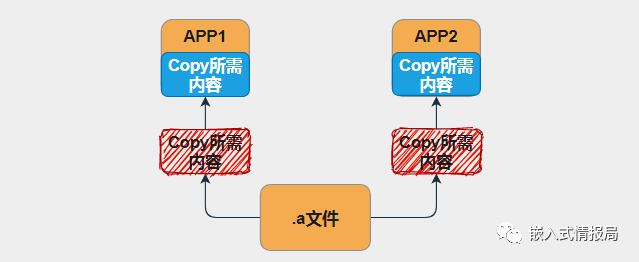
文件后缀对于Windows系统而言代表着文件的具体格式，而对于大部分Linux系统而言是没有实际意义了，Linux系统中主要关注文件的属性，然而在Linux中一般会带有后缀主要是方便阅读和识别文件。

2**动态和静态链接库**

    库文件说得简单一点就是一些通用函数方法的打包集合-Library，平时一些通用的方法可以直接从库文件中调用即可，而无需自己再进行编写，也是为了避免重复"造轮子"。

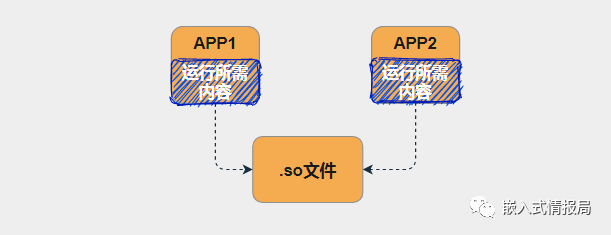
    那一些人该问了:为何不直接使用源文件进行打包呢？还可以更好的调试程序。前面也说了是一些通用的方法程序打包，一方面可以隐藏一些算法的实现细节发布给客户进行二次开发；另外一方面减少代码编译的时间，同时使用动态链接库能够更好的节约内存。

01  
图片静态链接库



    .a文件可以简单的看成.o目标文件的一个集合，当程序处于最后的链接阶段每个APP文件会分别从.a静态库文件中找到所需的函数等并复制到对应位置形成一个整体，如果有多个APP就需要复制多份。

02  
图片动态链接库



    .so文件属于动态链接文件，从上面静态链接的介绍了解到库文件在程序编译的过程中把相应的部分copy到执行程序中形成一个新的执行程序。

    而动态链接仅仅只是在APP文件中留下一个标记，当程序在内存中执行时发现缺少该动态链接文件，便会加载到内存中，如果其他APP也需要使用该库中的内容直接调用就行，无需再复制了，这样也就大大节省了内存。

    所以对于GCC默认链接操作是动态链接。

3**小结**

    动态链接库优点非常多，所以现在也非常受大家的喜爱，当需要升级对应的库文件只需要直接替换即可，不需要再进行编译过程。

    但虽然静态链接库最终生成的程序占用内存较多，不过其兼容性相比动态链接更好，同时编译生成即可直接运行，不需要再依赖动态链接库。

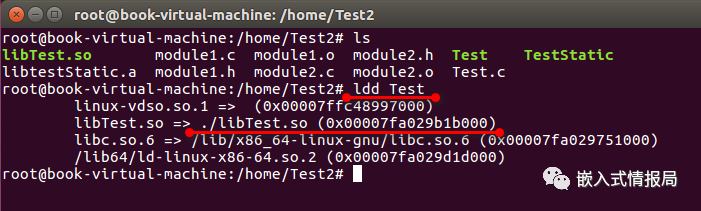
# Linux应用编程之动静态链接库辅助技巧(ldd,-static)

**查看执行文件共享库**

    在平时的开发中可能会存在这样的需求，想确定所提供的可执行文件需要哪些动态共享库，这样方便我们进行相关路径的设置和准备，防止在编译过程中造成的编译错误。

    这样Linux提供了ldd命令，通过使用该命令行不仅仅能够知道该可执行文件需要哪些动态共享库文件，同时还能够测试这些共享库是否能够被加载。

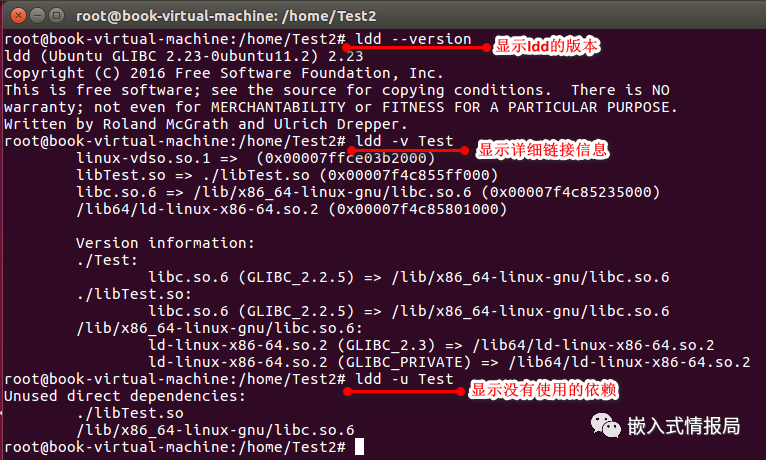
    下面小哥就使用ldd命令查看之前的Test可执行文件的链接状态，大家可以发现在连接过程中不仅仅只包括我们自己编写的libTest.so共享库文件，还包括libc其他动态库文件。



**ldd命令**

    ldd不是一个可执行文件，仅仅只是一个shell脚本，通过该脚本设置相应的环境变量的值，从而使得ld-linux.so（elf动态库的装载器）只显示该执行程序的依赖。

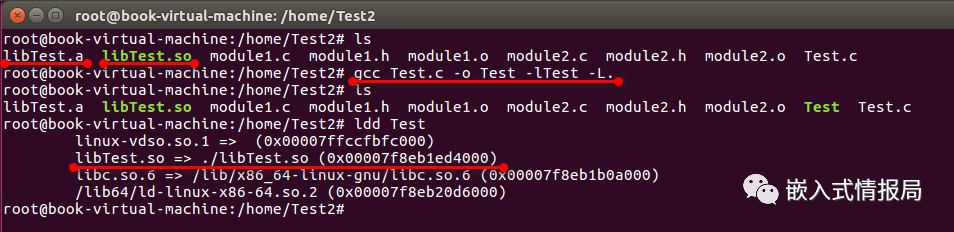
    ldd命令还有一些可选的选项，下面小哥简单的例举了几个，其他的可以通过man命令来查找:



2**指定链接动态或者静态库**

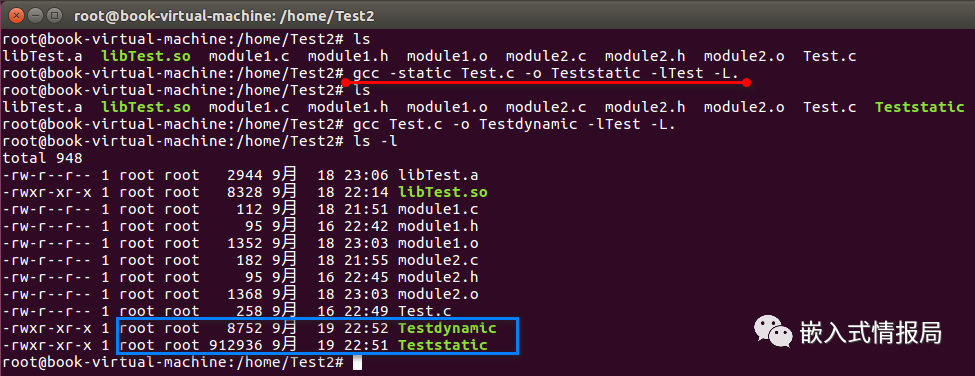
    前面的知识中不管是动态链接库还是静态链接库都会把最终的库文件定义为libxxx.so或者libxxx.a,这样方便GCC进行链接查找并进行相应的程序链接。

    然而如果其中的xxx名字是一样的，GCC该进行如何的链接呢?下面小哥就前面的例子实验一下:



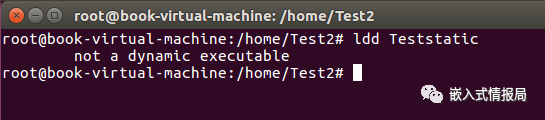
     把两个链接库名字中的xxx都改成了Test，然后执行可执行文件的生成，最终使用ldd命令发现其进行的还是动态链接方式，所以gcc默认是采用的动态链接，如果找不到对应的动态链接就会采用静态链接方式。

    那么小哥再介绍一下gcc的 -static选项，通过使用该选项，最终编译器会以静态链接的方式进行库的链接，如下:



    通过使用-static选项，最终生成的可执行文件大小相比动态链接库不是一个数量等级上的。

    从上面的结果也可知道该选项使用的是完全的静态链接方式，如果使用ldd会提示其没有任何依赖的共享库:



    然而在平时的开发中还会有动静态混合链接的需求，这样就需要使用到GCC的-Wl,-Bstatic ，-Wl,-Bdynamic等选项，本文就暂时不展开了，后面的文章小哥再跟大家演示。

3**小结**

    今天主要是跟大家介绍了ldd命令和-static完全的静态链接，同时在平时的开发中还会有动静态混合链接的需求，下期精彩见。

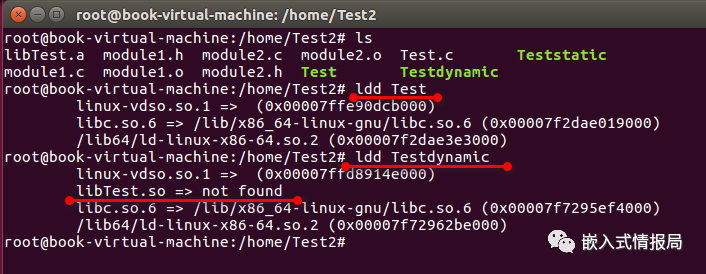
# Linux应用编程之动静态混合链接演示

**链接方式总结**

    前面的文章为大家介绍了动态链接和静态链接，如果使用gcc的-static选项所采用的链接是全静态链接方式，也就是说不会使用到动态链接库。

    不过有时候我们有着动态链接和静态链接混合的需求，那该如何进行编译呢?

     当没有libxxx.so和libxxx.a同名的时候，且仅仅只有.a静态库时，采用gcc Test.c -o Test -lTest -L. 命令生成的可执行文件就只会链接静态链接库，而其他的仍然会采用动态链接方式,如下:



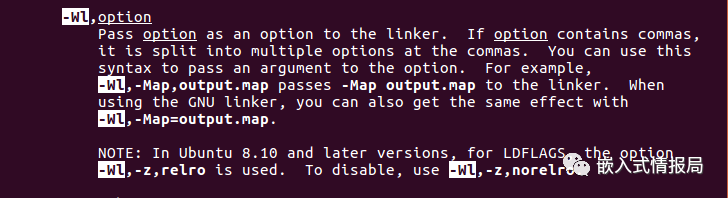
    采用ldd命令查看Test和Testdynamic中发现，其中的Test可执行文件没有去链接libTest.so，因为前面采用了直接链接libTest.a的方式来生成，然而可执行文件其他的链接均为动态链接，从而形成了一种混合链接方式。

    其中最后一个句中的not found，由于小哥把之前生成的libtest.so动态库进行了删除，所以使用ldd命令无法找到该库，不过不影响实验现象。

2**明确指定混合链接**

    前面的这种方式对于不重名的库链接还是比较好用的，不过我们还是想更加明确的指定哪些库进行动态链接，哪些库进行静态链接，这样就需要借助gcc编译器的-Wl选项了。

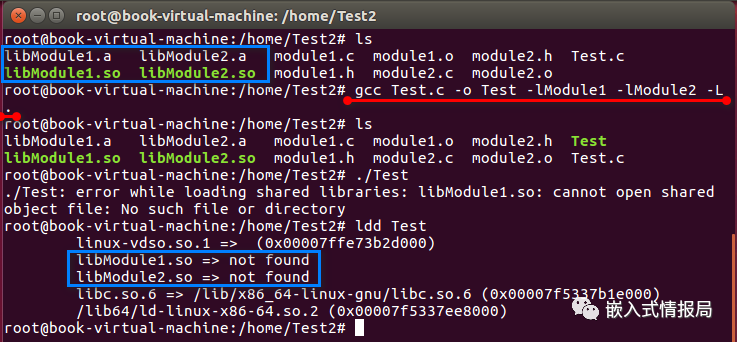
    可以通过man gcc或者info gcc查找gcc的对应帮助文档了解该选项的相关说明:



    通过上面的解释大致可以了解到该选项是用来引导链接方式的，其中-Wl,-Bstatic指示跟在后面的xxx链接的都是静态库，-Wl,-Bdynamic指示跟在后面的xxx选项都是动态库。

2**实例演示**

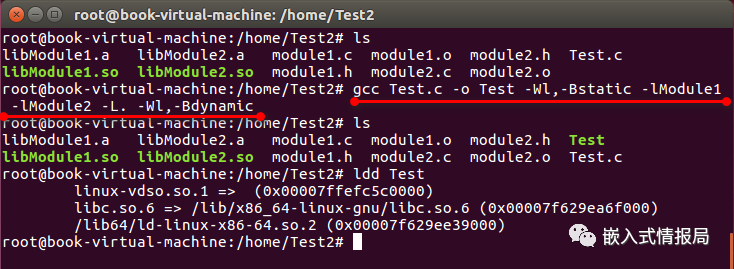
    如下面的实现过程所示:



**过程说明:**

    首先根据之前的介绍把module1和module2分别编译生成对应的静态库文件和动态库文件。

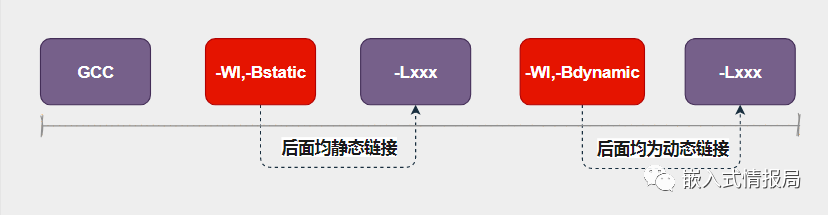
    然后我们选择没有任何指定的-lxxx编译选项，最后的结果是所有的编译链接方式都是动态链接，因为小哥这里没有设置环境变量，所以加载器找不到共享库文件。

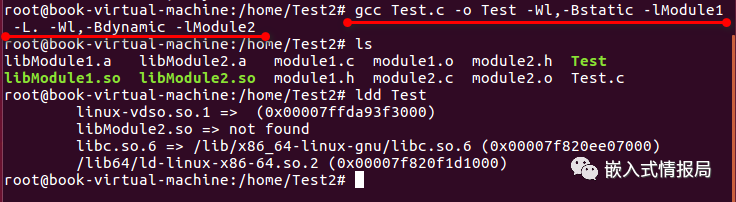


**过程说明:**

    上面通过在-lxxx前后添加-Wl选项，表示module1和module2均采用静态链接，其余采用动态链接。

    可能一些小伙伴还是觉得迷糊，小哥这里画一个图理解一下这两个命令的使用:

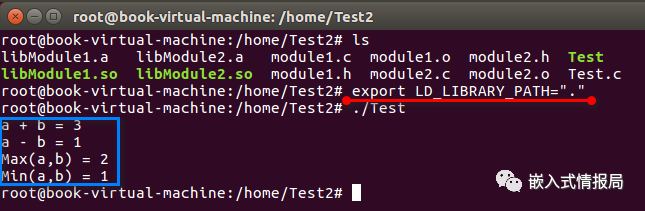




过程说明:

    上面就仅仅只让module1进行静态链接，而对于module2仍然采用的静态库，使用ldd就可以发现其链接状态。

    最后小哥设置了一下环境变量，运行了动静态混合链接生成的可执行文件，一切OK。



3**小结**

    今天主要是跟大家介绍了gcc进行动静态库混合链接的例子，gcc基本上就介绍到这里了，下一期跟大家讲解GDB调试，下期精彩见。

# Linux应用编程之"一文搞定Makefile"(上)

## ****什么是Makefile****

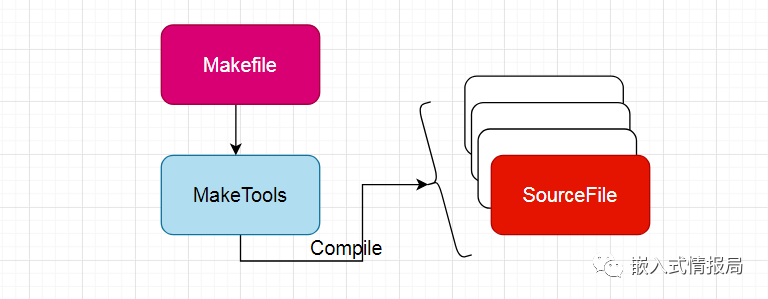
    在之前学习C程序的编译过程中，我们知道编译一个程序还比较简单， 如果要编译多个文件，或者不同文件夹中的文件，需要生成不同的库文件，以及确定这些文件的编译先后顺序，往往所需要的命令行特别多，而且比较复杂，甚至对于以后项目的维护也比较麻烦。

    再想想如果我们需要编译Linux内核这样好几W个文件的项目难道需要我们一个命令的输入吗 ? 那估计是一场噩梦。



    那么这个时候如果我们能够把所有的编译规则全部规范到文件中，然后通过解析该文件去执行对应的编译指令，这样就大大简化指令的复杂度，同时降低了编译程序过程中所带来的错误。

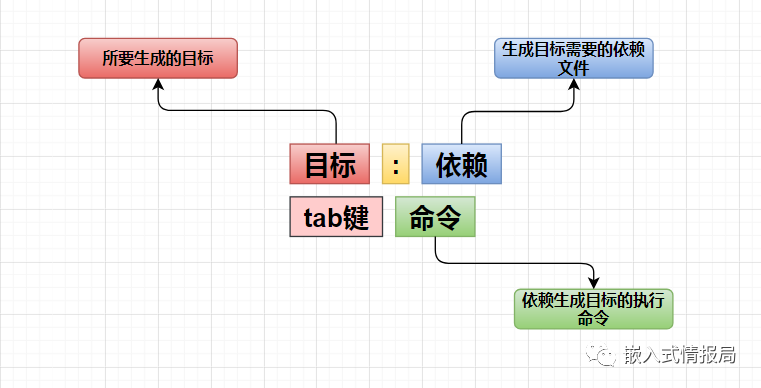
    根据上面的需求就产生了Makefile,我们的编译和处理规则就放在Makefile文件中，通过Makefile工具解析Makefile文件中的命令来指导整个工程的编译过程。



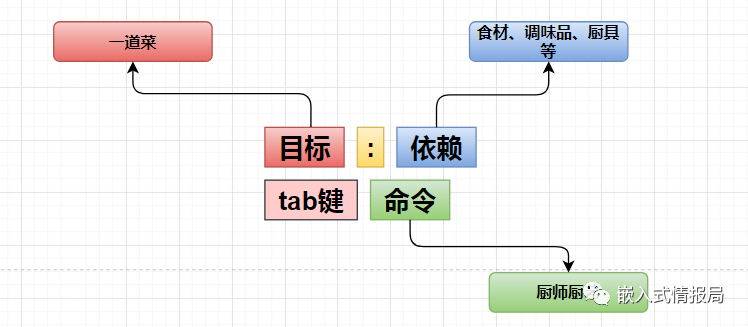
    当然Makefile文件中的命令书写是有一定的规范的，这也是今天我们所要讲到的重点，一旦该文件编写好以后在Linux命令行中执行一条make命令即可自动编译整个工程，不但提高了开发效率也便于后期维护。

    当然不同厂家的make稍有不同，并且语法上也有点区别，不过基本思想都差不多，主要还是落在目标依赖上来，这里以最广泛的GNU Make跟大家讲解。

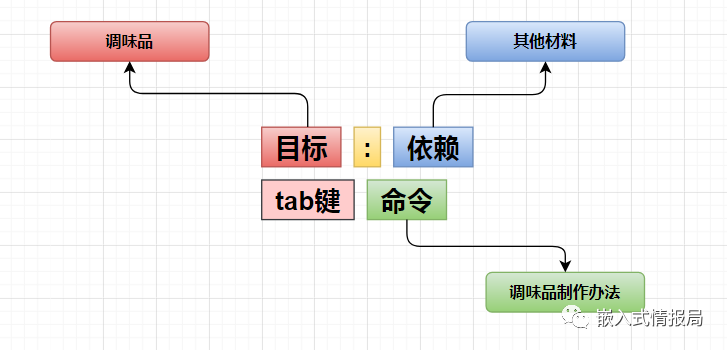
## 2****Makefile文件基本规则****



    上面的为大家展示了Makefile的核心规则，有点类似于一位厨师做菜，目标就是做一碗好菜，那么所谓的依赖就是各种食材、各种厨具等等，然而有了这些依赖还不够需要厨师有着非常好的厨艺才能做出一道好菜。



    同时这些依赖也有可能此时此刻并不存在，需要现场制作，或者由其他厨师做好，那么这个依赖就成为了其他规则的目标，该目标也会有它自己的依赖和命令，这样形成一层一层递归依赖组成了Makefile文件。

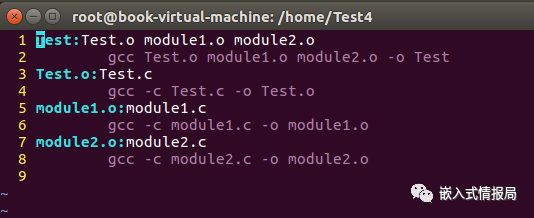


总结

* Makefile并不会关心命令是如何执行的，仅仅只是会去执行所定义的命令，和我们平时直接输入命令行是一样的效果。
* Makefile就相当于一个依赖关系文件，在执行该文件的以后会递归的查找依赖文件并执行对应的命令，最终生成第一个目标。

## 3****简单使用Makefile****

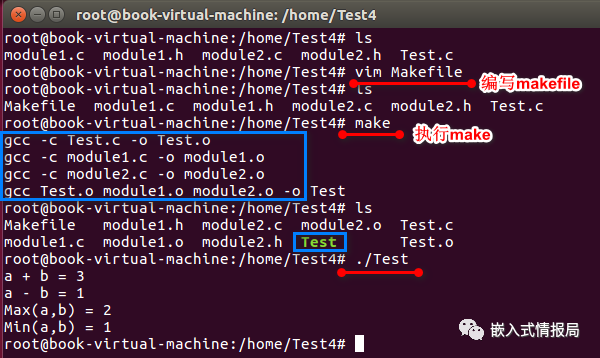
    下面小哥就之前代码简单演示一下Makefile:



    上图是makefile文件，最终为了生成Test文件，需要三个\*.o文件，然后三个.o文件又分别依赖于对应的\*.c文件，这样加上对应的gcc命令就构成了一个基本的makefile。

**注意 : 2/4/6/8行均为tab键。**

    下面使用make命令来编译生成test目标文件



    这样我们执行**make命令一键就搞定**了所有的编译任务，如果下次需要改变编译过程就直接修改对应的makefile文件中的规则即可。

**有类比过windows里面IDE环境中编译器使用的小伙伴都知道在windows里面的开发工具编译通过一键即可搞定所有编译内容。难道windows中的开发工具更强大一些？**

    其实不然，我们在windows中构建对应的工程文件的过程中通过图形界面就悄悄的把类似的makefile文件给生成了，那么当点击编译的时候就相当于在命令行里面执行make，其实都是类似的，仅仅只是我们平时使用windows更多一点，更加容易接受。

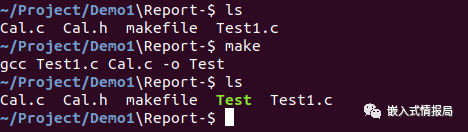
   当然这里仅仅只是演示了一些最简单的操作，比较麻烦的还是命令部分可以使用到通配符和shell指令等等，这样会增加一定的复杂度，其实makefile本身并没有太大的难度。

    下一节小哥将为大家补充一些makefile更多便捷的操作，当然现在也出现了许多类似于cmake这样的便捷工具，不过其核心还是makefile！

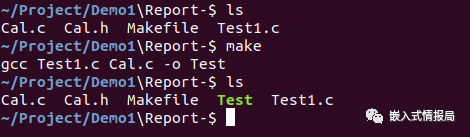
# Linux应用编程之"一文搞定Makefile"(下)

## ****指定使用Makefile文件****

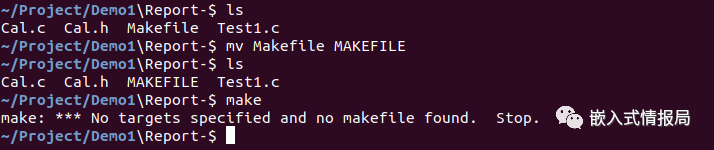
    在命令行中使用make命令，makefile会在默认路径中查找对应的makefile文件来进行工程管理，我们一般把对应的makefile文件名命名为Makefile和makefile等，如果名字不匹配可能无法找到对应的makefile文件,比如:



makefile文件名



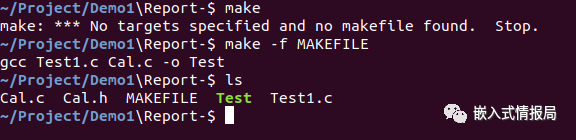
Makefile文件名



MAKEFILE文件名－编译失败

  　从上面的实验现象可以了解到make会查找默认的makefile文件名，如果没有找到规定的文件名就会报相应的故障。

    不过如果平时在一个目录下存在多个makefile文件的时候，一般我们都会通过不同命名来进行区分，那么该如何指定对应的makefile文件进行识别解析呢　?可以使用make -f选项,如下图所示:

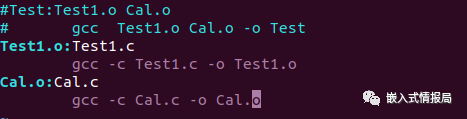


## 2****注释文本****

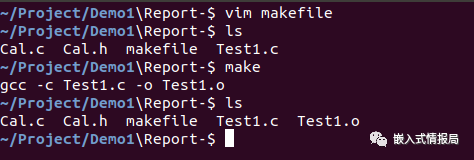
    在开发过程中存在需要注释掉makefile文件相应文本行可以在文件的行首使用＃号:



    上面是正常进行ｍakefile编译目标文件的完整实例，并且输出了正确的结果,然后小哥使用#号屏蔽掉前面两行,如下图所示:



    继续执行make命令，根据makefile的规则，会默认编译Test1.o目标文件:

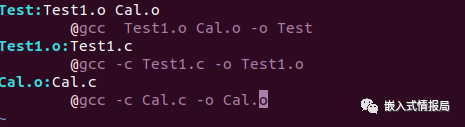


    这样我们获得了正确的结果，说明注释生效。

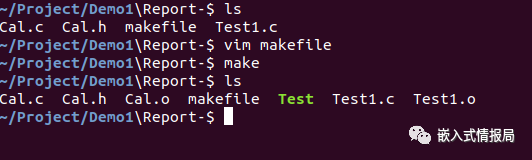
## 3****取消回显文本****

    在makefile使用过程中默认是会把相应的处理过程信息进行回显，这样能够让开发者更好的了解makefile的处理过程.

    不过，如果回显信息非常的庞大也是不利于开发者分析的，所以编辑人员可以使用＠来取消相关信息的回显，如下图所示:



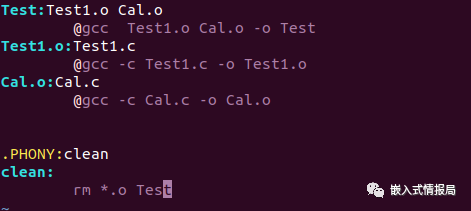
在命令tab键后加入＠



    这里我们可以看到使用make以后没有像之前使用的makefile文件那样存在回显的命令行信息了。

## 4****伪目标的使用****

    在makefile文件中有时候执行一些不需要有依赖命令的目标，称为伪目标。



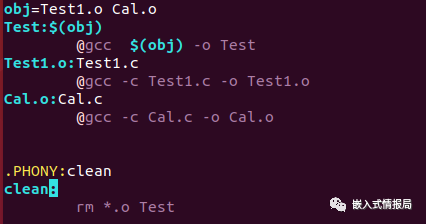
    上面是对应的makefile文件，其中里面加入伪目标clean，通过使用.PHONY修饰，这样可以直接使用make clean 来执行对应的命令。



    上面执行make clean确实执行了清除操作。

## 5****变量的使用****

    在makefile中为了减少代码的重复，使用变量的概念来简化编写，如下面的makefile进行如下改造:



    使用$(obj)来代替所有的目标文件，以后需要添加其他目标文件的话就只需要在变量obj处进行相应修改即可。

    同时系统还存在其他默认的自动化变量，这样可以大大简化makefile文件，便于设计和后期维护，如:

* $^    表示所有的依赖文件
* $@    表示生成的目标文件
* $<    代表第一个依赖文件
* 等等

    makefile基础知识点小哥暂时就讲到这里，其实大部分工程项目自己完完全全编写的并不是很多，更多的修修改改，所以遇到了具体的疑问在进行查找或许效率更高。