# 第4篇 嵌入式Linux应用开发基础知识

## 第一章 HelloWorld背后没那么简单

### 1.1 交叉编译hello.c

使用GIT下载所有源码后，本节源码位于如下目录：

01\_all\_series\_quickstart\

04\_嵌入式Linux应用开发基础知识\source\01\_hello

hello.c的源码如下：

01 #include <stdio.h>

02

03 /\* 执行命令: ./hello weidongshan

04 \* argc = 2

05 \* argv[0] = ./hello

06 \* argv[1] = weidongshan

07 \*/

08

09 int main(int argc, char \*\*argv)

10 {

11 if (argc >= 2)

12 printf("Hello, %s!\n", argv[1]);

13 else

14 printf("Hello, world!\n");

15 return 0;

16 }

17

在Ubuntu中可以执行以下命令编译、执行：

$ gcc -o hello hello.c

$ ./hello

Hello, world!

$ ./hello weidongshan

Hello, weidongshan!

上述命令编译得到的可执行程序hello可以在Ubuntu中运行，但是如果把它放到ARM板子上去，它是无法执行的。因为它是使用gcc编译的，是给PC机编译的，里面的机器指令是x86的。

我们要想给ARM板编译出hello程序，需要使用交叉编译工具链，比如：

$ arm-linux-gnueabihf-gcc -o hello hello.c

这样编译出来的hello程序才可以在ARM板子上运行。

### 1.2 请回答这几个问题



① 怎么确定交叉编译器中头文件的默认路径？

进入交叉编译器的目录里，执行：find -name “stdio.h”，它位于一个“include”目录下的根目录里。

这个“include”目录，就是要找的路径。

② 怎么自己指定头文件目录？

编译时，加上“-I <头文件目录>”这样的选项。

③ 怎么确定交叉编译器中库文件的默认路径？

进入交叉编译器的目录里，执行：find -name lib，可以得到xxxx/lib、xxxx/usr/lib，一般来说这2个目录就是要找的路径。

如果有很多类似的lib，进去看看，有很多so文件的目录一般就是要找的路径。

④ 怎么自己指定库文件目录、指定要用的库文件？

编译时，加上“-L <库文件目录>”这样的选项，用来指定库目录；

编译时，加上“-labc”这样的选项，用来指定库文件libabc.so。

### 1.3 演示

## 第二章 GCC编译器的使用

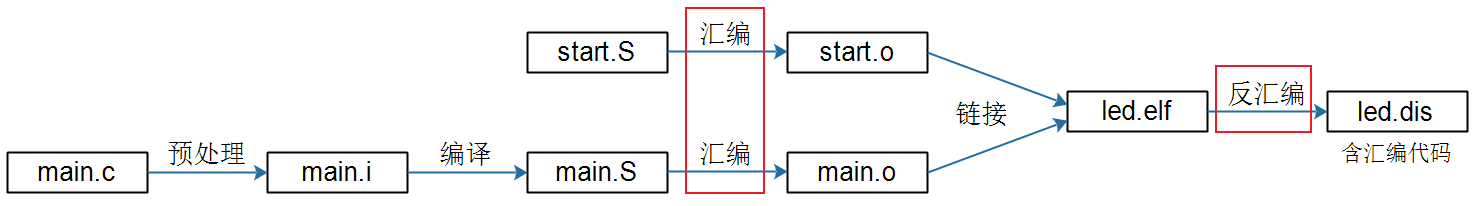
源文件需要经过编译才能生成可执行文件。在Windows下进行开发时，只需要点几个按钮即可编译，集成开发环境(比如Visual studio)已经将各种编译工具的使用封装好了。Linux下也有很优秀的集成开发工具，但是更多的时候是直接使用编译工具；即使使用集成开发工具，也需要掌握一些编译选项。

PC机上的编译工具链为gcc、ld、objcopy、objdump等，它们编译出来的程序在x86平台上运行。要编译出能在ARM平台上运行的程序，必须使用交叉编译工具xxx-gcc、xxx-ld等(不同版本的编译器的前缀不一样，比如arm-linux-gcc)，下面分别介绍。

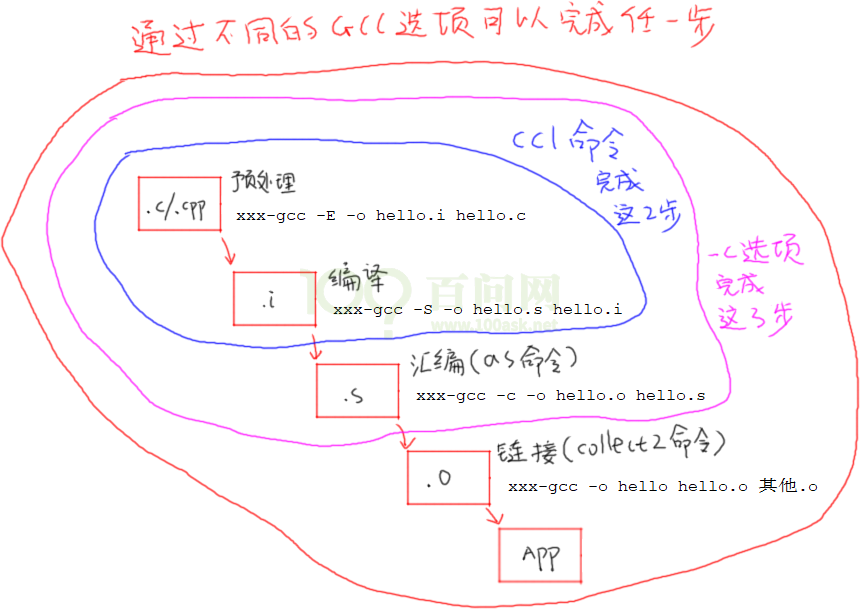
### 2.1 配套视频内容大纲

#### 2.1.1 GCC编译过程(精简版)

一个C/C++文件要经过预处理(preprocessing)、编译(compilation)、汇编(assembly)和链接(linking)等4步才能变成可执行文件。



通过不同的gcc选项可以控制这些过程：



在日常交流中通常使用“编译”统称这4个步骤，如果不是特指这4个步骤中的某一个，本教程也依惯例使用“编译”这个统称。

gcc使用示例：

gcc hello.c // 输出一个名为a.out的可执行程序，然后可以执行./a.out

gcc -o hello hello.c // 输出名为hello的可执行程序，然后可以执行./hello

gcc -o hello hello.c -static // 静态链接

gcc -c -o hello.o hello.c // 先编译(不链接)

gcc -o hello hello.o // 再链接

执行“gcc -o hello hello.c -v”时，可以查看到这些步骤：

cc1 main.c -o /tmp/ccXCx1YG.s

as -o /tmp/ccZfdaDo.o /tmp/ccXCx1YG.s

cc1 sub.c -o /tmp/ccXCx1YG.s

as -o /tmp/ccn8Cjq6.o /tmp/ccXCx1YG.s

collect2 -o test /tmp/ccZfdaDo.o /tmp/ccn8Cjq6.o ....

可以手工执行以下命令体验一下：

gcc -E -o hello.i hello.c

gcc -S -o hello.s hello.i

gcc -c -o hello.o hello.s

gcc -o hello hello.o

#### 2.1.2 常用编译选项

在学习时，我们暂时只需要了解下表中的选项。

|  |  |
| --- | --- |
| 常用选项 | 描述 |
| -E | 预处理，开发过程中想快速确定某个宏可以使用“-E -dM” |
| -c | 把预处理、编译、汇编都做了，但是不链接 |
| -o | 指定输出文件 |
| -I | 指定头文件目录 |
| -L | 指定链接时库文件目录 |
| -l | 指定链接哪一个库文件 |

#### 2.1.3 怎么编译多个文件

① 一起编译、链接：

gcc -o test main.c sub.c

② 分开编译，统一链接：

gcc -c -o main.o main.c

gcc -c -o sub.o sub.c

gcc -o test main.o sub.o

#### 2.1.4 制作、使用动态库

制作、编译：

gcc -c -o main.o main.c

gcc -c -o sub.o sub.c

gcc -shared -o libsub.so sub.o ~~sub2.o sub3.o(~~可以使用多个.o生成动态库~~)~~

gcc -o test main.o -lsub -L /libsub.so/所在目录/

运行：

① 先把libsub.so放到Ubuntu的/lib目录，然后就可以运行test程序。

② 如果不想把libsub.so放到/lib，也可以放在某个目录比如/a，然后如下执行：

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/a

./test

#### 2.1.5 制作、使用静态库

gcc -c -o main.o main.c

gcc -c -o sub.o sub.c

ar crs libsub.a sub.o ~~sub2.o sub3.o(~~可以使用多个.o生成静态库~~)~~

gcc -o test main.o libsub.a (如果.a不在当前目录下，需要指定它的绝对或相对路径)

运行：

不需要把静态库libsub.a放到板子上。

**注意**:执行arm-linux-gnueabihf-gcc -c -o sub.o sub.c交叉编译需要在最后面加上 -fPIC参数。

#### 2.1.6 很有用的选项

gcc -E main.c // 查看预处理结果，比如头文件是哪个

gcc -E -dM main.c > 1.txt // 把所有的宏展开，存在1.txt里

gcc -Wp,-MD,abc.dep -c -o main.o main.c // 生成依赖文件abc.dep，后面Makefile会用

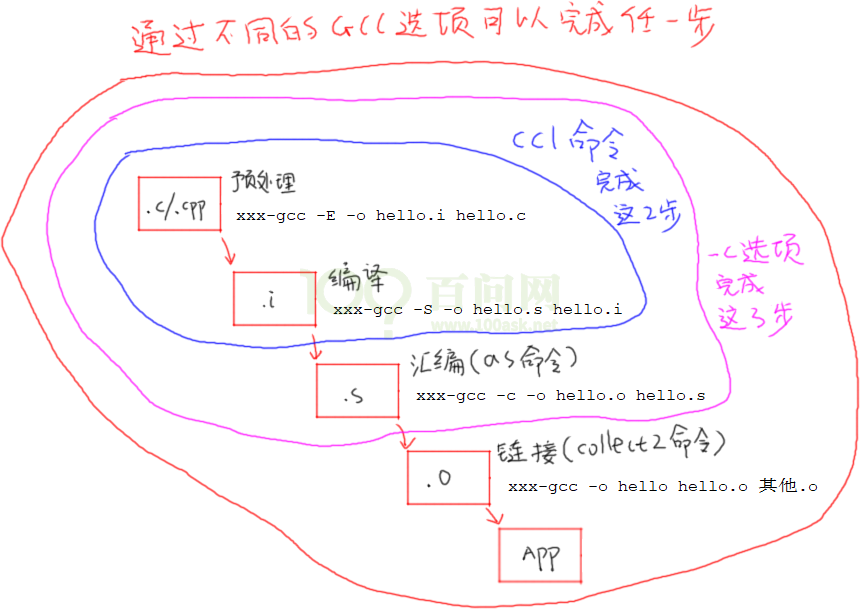
echo 'main(){}'| gcc -E -v - // 它会列出头文件目录、库目录(LIBRARY\_PATH)

我们的视频会配套写成书：《嵌入式Linux应用开发完全手册 升级版》菜单。下面的资料会写进书里，我会写得详细一点。

下面的资料来自GCC官方文档及一些中文资料，没必要逐一研读，用到时再翻翻。

### 2.2 GCC编译过程

一个C/C++文件要经过预处理(preprocessing)、编译(compilation)、汇编(assembly)和链接(linking)等4步才能变成可执行文件。



在日常交流中通常使用“编译”统称这4个步骤，如果不是特指这4个步骤中的某一个，本教程也依惯例使用“编译”这个统称。

本节文档使用x86上的gcc来试验，使用ARM板的交叉编译工具链做实验时效果也是类似的。不同的交叉编译器工具链前缀可能不同，比如arm-linux-gcc。

（1）预处理

C/C++源文件中，以“#”开头的命令被称为预处理命令，如包含命令“#include”、宏定义命令“#define”、条件编译命令“#if”、“#ifdef”等。预处理就是将要包含(include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些东西输出到一个“.i”文件中等待进一步处理。

（2）编译

编译就是把C/C++代码(比如上述的“.i”文件)“翻译”成汇编代码，所用到的工具为cc1(它的名字就是cc1，x86有自己的cc1命令，ARM板也有自己的cc1命令)。

（3）汇编

汇编就是将第二步输出的汇编代码翻译成符合一定格式的机器代码，在Linux系统上一般表现为ELF目标文件(OBJ文件)，用到的工具为as。x86有自己的as命令，ARM版也有自己的as命令，也可能是xxxx-as（比如arm-linux-as）。

“反汇编”是指将机器代码转换为汇编代码，这在调试程序时常常用到。

（4）链接

链接就是将上步生成的OBJ文件和系统库的OBJ文件、库文件链接起来，最终生成了可以在特定平台运行的可执行文件，用到的工具为ld或collect2。

编译程序时，加上-v选项就可以看到这几个步骤。比如：

gcc -o hello hello.c -v

可以看到很多输出结果，我们把其中的主要信息摘出来：

cc1 hello.c -o /tmp/cctETob7.s

as -o /tmp/ccvv2KbL.o /tmp/cctETob7.s

collect2 -o hello crt1.o crti.o crtbegin.o /tmp/ccvv2KbL.o crtend.o crtn.o

以上3个命令分别对应于编译步骤中的预处理+编译、汇编和链接，ld被collect2调用来链接程序。预处理和编译被放在了一个命令（cc1）中进行的，可以把它再次拆分为以下两步：

cpp -o hello.i hello.c

cc1 hello.i -o /tmp/cctETob7.s

我们不需要手工去执行cpp、cc1、collect2等命令，我们直接执行gcc并指定不同的参数就可以了。

可以通过各种选项来控制gcc的动作，下面介绍一些常用的选项。

|  |  |
| --- | --- |
| 常用选项 | 描述 |
| -E | 预处理，开发过程中想快速确定某个宏可以使用“-E -dM” |
| -c | 把预处理、编译、汇编都做了，但是不链接 |
| -o | 指定输出文件 |
| -I | 指定头文件目录 |
| -L | 指定链接时库文件目录 |
| -l | 指定链接哪一个库文件 |

### 2.3 GCC总体选项(Overall Option)

（1）-c

预处理、编译和汇编源文件，但是不作链接，编译器根据源文件生成OBJ文件。缺省情况下，GCC通过用`.o'替换源文件名的后缀`.c'，`.i'，`.s'等，产生OBJ文件名。可以使用-o选项选择其他名字。GCC忽略-c选项后面任何无法识别的输入文件。

（2）-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，输出结果是汇编语言文件。缺省情况下，GCC通过用`.s'替换源文件名后缀`.c'，`.i'等等，产生汇编文件名。可以使用-o选项选择其他名字。GCC忽略任何不需要汇编的输入文件。

（3）-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。

（4）-o file

指定输出文件为file。无论是预处理、编译、汇编还是链接，这个选项都可以使用。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'；修改输入文件的名称是`source.suffix'，则它的OBJ文件是`source.o'，汇编文件是 `source.s'，而预处理后的C源代码送往标准输出。

（5）-v

显示制作GCC工具自身时的配置命令；同时显示编译器驱动程序、预处理器、编译器的版本号。

以一个程序为例，它包含三个文件，代码在02\_options目录下。下面列出源码：

File: main.c

01 #include <stdio.h>

02 #include "sub.h"

03

04 int main(int argc, char \*argv[])

05 {

06 int i;

07 printf("Main fun!\n");

08 sub\_fun();

09 return 0;

10 }

11

File: sub.h

01 void sub\_fun(void);

02

File: sub.c

01 void sub\_fun(void)

02 {

03 printf("Sub fun!\n");

04 }

05

ARM版本的编译工具与gcc、ld等工具的使用方法相似，很多选项是一样的。本节使用gcc、ld等工具进行编译、链接，这样可以在PC上直接看到运行结果。使用上面介绍的选项进行编译，命令如下：

$ gcc -c -o main.o main.c

$ gcc -c -o sub.o sub.c

$ gcc -o test main.o sub.o

其中，main.o、sub.o是经过了预处理、编译、汇编后生成的OBJ文件，它们还没有被链接成可执行文件；最后一步将它们链接成可执行文件test，可以直接运行以下命令：

$ ./test

Main fun!

Sub fun!

现在试试其他选项，以下命令生成的main.s是main.c的汇编语言文件：

$ gcc -S -o main.s main.c

以下命令对main.c进行预处理，并将得到的结果打印出来。里面扩展了所有包含的文件、所有定义的宏。在编写程序时，有时候查找某个宏定义是非常繁琐的事，可以使用`-dM –E’选项来查看。命令如下：

$ gcc -E main.c

### 2.4 警告选项(Warning Option)

（1）-Wall

这个选项基本打开了所有需要注意的警告信息，比如没有指定类型的声明、在声明之前就使用的函数、局部变量除了声明就没再使用等。

上面的main.c文件中，第6行定义的变量i没有被使用，但是使用“gcc –c –o main.o main.c”进行编译时并没有出现提示。

可以加上-Wall选项，例子如下：

$ gcc -Wall -c main.c

执行上述命令后，得到如下警告信息：

main.c: In function `main':

main.c:6: warning: unused variable `i'

这个警告虽然对程序没有坏的影响，但是有些警告需要加以关注，比如类型匹配的警告等。

### 2.5 调试选项(Debugging Option)

（1）-g

以操作系统的本地格式(stabs，COFF，XCOFF，或DWARF)产生调试信息，GDB能够使用这些调试信息。在大多数使用stabs格式的系统上，`-g'选项加入只有GDB才使用的额外调试信息。可以使用下面的选项来生成额外的信息：`-gstabs+'，`-gstabs'，`-gxcoff+'，`-gxcoff'，`-gdwarf+'或`-gdwarf'，具体用法请读者参考GCC手册。

### 2.6 优化选项(Optimization Option)

（1）-O或-O1

优化：对于大函数，优化编译的过程将占用稍微多的时间和相当大的内存。不使用`-O'或`-O1'选项的目的是减少编译的开销，使编译结果能够调试、语句是独立的：如果在两条语句之间用断点中止程序，可以对任何变量重新赋值，或者在函数体内把程序计数器指到其他语句，以及从源程序中精确地获取你所期待的结果。

不使用`-O'或`-O1'选项时，只有声明了register的变量才分配使用寄存器。

使用了`-O'或`-O1'选项，编译器会试图减少目标码的大小和执行时间。如果指定了`-O'或`-O1'选项,，`-fthread-jumps'和`-fdefer-pop'选项将被打开。在有delay slot的机器上，`-fdelayed-branch'选项将被打开。在即使没有帧指针 (frame pointer)也支持调试的机器上，`-fomit-frame-pointer'选项将被打开。某些机器上还可能会打开其他选项。

（2）-O2

多优化一些。除了涉及空间和速度交换的优化选项，执行几乎所有的优化工作。例如不进行循环展开(loop unrolling)和函数内嵌(inlining)。和`-O'或`-O1'选项比较，这个选项既增加了编译时间，也提高了生成代码的运行效果。

（3）-O3

优化的更多。除了打开-O2所做的一切，它还打开了-finline-functions选项。

（4）-O0

不优化。

如果指定了多个-O选项，不管带不带数字，生效的是最后一个选项。

在一般应用中，经常使用-O2选项，比如对于options程序：

$ gcc -O2 -c -o main.o main.c

$ gcc -O2 -c -o sub.o sub.c

$ gcc -o test main.o sub.o

### 2.7 链接器选项(Linker Option)

下面的选项用于链接OBJ文件，输出可执行文件或库文件。

（1）object-file-name

如果某些文件没有特别明确的后缀(a special recognized suffix)，GCC就认为他们是OBJ文件或库文件(根据文件内容,链接器能够区分OBJ文件和库文件)。如果GCC执行链接操作，这些OBJ文件将成为链接器的输入文件。

比如上面的“gcc -o test main.o sub.o”中，main.o、sub.o就是输入的文件。

（2）-llibrary

链接名为library的库文件。

链接器在标准搜索目录中寻找这个库文件，库文件的真正名字是`liblibrary.a'。搜索目录除了一些系统标准目录外，还包括用户以`-L'选项指定的路径。一般说来用这个方法找到的文件是库文件──即由OBJ文件组成的归档文件(archive file)。链接器处理归档文件的方法是：扫描归档文件，寻找某些成员，这些成员的符号目前已被引用，不过还没有被定义。但是，如果链接器找到普通的OBJ文件，而不是库文件，就把这个OBJ文件按平常方式链接进来。指定`-l'选项和指定文件名的唯一区别是，`-l’选项用`lib'和`.a'把library包裹起来，而且搜索一些目录。

即使不明显地使用-llibrary选项，一些默认的库也被链接进去，可以使用-v选项看到这点：

$ gcc -v -o test main.o sub.o

输出的信息如下：

/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf\_i386 -dynamic-linker /lib/ld-linux.so.2

-o test

/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../../crt1.o /usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../../crti.o

/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/crtbegin.o

-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2

-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..

main.o

sub.o

-lgcc -lgcc\_eh -lc -lgcc -lgcc\_eh

/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/crtend.o

/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../../crtn.o

可以看见，除了main.o、sub.o两个文件外，还链接了启动文件crt1.o、crti.o、crtend.o 、crtn.o，还有一些库文件(-lgcc -lgcc\_eh -lc -lgcc -lgcc\_eh)。

（3）-nostartfiles

不链接系统标准启动文件，而标准库文件仍然正常使用：

$ gcc -v -nostartfiles -o test main.o sub.o

输出的信息如下：

/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf\_i386 -dynamic-linker

/lib/ld-linux.so.2

-o test

-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2

-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..

main.o

sub.o

-lgcc -lgcc\_eh -lc -lgcc -lgcc\_eh

/usr/bin/ld: warning: cannot find entry symbol \_start; defaulting to 08048184

可以看见启动文件crt1.o、crti.o、crtend.o 、crtn.o没有被链接进去。需要说明的是，对于一般应用程序，这些启动文件是必需的，这里仅是作为例子(这样编译出来的test文件无法执行)。在编译bootloader、内核时，将用到这个选项。

（4）-nostdlib

不链接系统标准启动文件和标准库文件，只把指定的文件传递给链接器。这个选项常用于编译内核、bootloader等程序，它们不需要启动文件、标准库文件。

仍以options程序作为例子：

$ gcc -v -nostdlib -o test main.o sub.o

输出的信息如下：

/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf\_i386 -dynamic-linker /lib/ld-linux.so.2

-o test

-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2

-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..

main.o

sub.o

/usr/bin/ld: warning: cannot find entry symbol \_start; defaulting to 08048074

main.o(.text+0x19): In function `main':

: undefined reference to `printf'

sub.o(.text+0xf): In function `sub\_fun':

: undefined reference to `printf'

collect2: ld returned 1 exit status

出现了一大堆错误，因为printf等函数是在库文件中实现的。在编译bootloader、内核时，用到这个选项──它们用到的很多函数是自包含的。

（5）-static

在支持动态链接(dynamic linking)的系统上，阻止链接共享库。

仍以options程序为例，是否使用-static选项编译出来的可执行程序大小相差巨大：

$ gcc -c -o main.c

$ gcc -c -o sub.c

$ gcc -o test main.o sub.o

$ gcc -o test\_static main.o sub.o –static

$ ls -l test test\_static

-rwxr-xr-x 1 book book 6591 Jan 16 23:51 test

-rwxr-xr-x 1 book book 546479 Jan 16 23:51 test\_static

其中test文件为6591字节，test\_static文件为546479字节。当不使用-static编译文件时，程序执行前要链接共享库文件，所以还需要将共享库文件放入文件系统中。

（6）-shared

生成一个共享OBJ文件，它可以和其他OBJ文件链接产生可执行文件。只有部分系统支持该选项。

当不想以源代码发布程序时，可以使用-shared选项生成库文件，比如对于options程序，可以如下制作库文件：

$ gcc -c -o sub.o sub.c

$ gcc -shared -o libsub.so sub.o

以后要使用sub.c中的函数sub\_fun时，在链接程序时，指定引脚libsub.so即可，比如：

$ gcc -o test main.o -lsub -L /libsub.so/所在的目录/

可以将多个文件制作为一个库文件，比如：

$ gcc -shared -o libsub.so sub.o sub2.o sub3.o

（7）-Xlinker option

把选项option传递给链接器。可以用来传递系统特定的链接选项，GCC无法识别这些选项。如果需要传递携带参数的选项，必须使用两次`-Xlinker'，一次传递选项，另一次传递其参数。例如，如果传递`-assert definitions'，要成`-Xlinker -assert -Xlinker definitions'，而不能写成`-Xlinker "-assert definitions"'，因为这样会把整个字符串当做一个参数传递，显然这不是链接器期待的。

（8）-Wl,option

把选项option传递给链接器。如果option中含有逗号，就在逗号处分割成多个选项。链接器通常是通过gcc、arm-linux-gcc等命令间接启动的，要向它传入参数时，参数前面加上`-Wl,’。

（9）-u symbol

使链接器认为取消了symbol的符号定义，从而链接库模块以取得定义。可以使用多个 `-u'选项，各自跟上不同的符号，使得链接器调入附加的库模块。

### 2.8 目录选项(Directory Option)

下列选项指定搜索路径，用于查找头文件，库文件，或编译器的某些成员。

（1）-Idir

在头文件的搜索路径列表中添加dir 目录。

头文件的搜索方法为：如果以“#include < >”包含文件，则只在标准库目录开始搜索(包括使用-Idir选项定义的目录)；如果以“#include “ ””包含文件，则先从用户的工作目录开始搜索，再搜索标准库目录。

（2）-I-

任何在`-I-'前面用`-I'选项指定的搜索路径只适用于`#include "file"'这种情况；它们不能用来搜索`#include <file>'包含的头文件。如果用`-I'选项指定的搜索路径位于`-I-'选项后面，就可以在这些路径中搜索所有的`#include'指令(一般说来-I选项就是这么用的)。还有，`-I-'选项能够阻止当前目录(存放当前输入文件的地方)成为搜索`#include "file"'的第一选择。

`-I-'不影响使用系统标准目录，因此，`-I-'和`-nostdinc'是不同的选项。

（3）-Ldir

在`-l'选项的搜索路径列表中添加dir目录。

仍使用options程序进行说明，先制作库文件libsub.a：

$ gcc -c -o sub.o sub.c

$ gcc -shared -o libsub.a sub.o

编译main.c：

$ gcc -c -o main.o main.c

链接程序，下面的指令将出错，提示找不到库文件：

$ gcc -o test main.o -lsub

/usr/bin/ld: cannot find -lsub

collect2: ld returned 1 exit status

可以使用-Ldir选项将当前目录加入搜索路径，如下则链接成功：

$ gcc -L. -o test main.o -lsub

（4）-Bprefix

这个选项指出在何处寻找可执行文件，库文件，以及编译器自己的数据文件。编译器驱动程序需要使用某些工具，比如：`cpp'，`cc1' (或C++的`cc1plus')，`as'和`ld'。它把prefix当作欲执行的工具的前缀，这个前缀可以用来指定目录，也可以用来修改工具名字。

对于要运行的工具，编译器驱动程序首先试着加上`-B'前缀(如果存在)，如果没有找到文件，或没有指定`-B'选项，编译器接着会试验两个标准前缀`/usr/lib/gcc/'和`/usr/local/lib/gcc-lib/'。如果仍然没能够找到所需文件，编译器就在`PATH'环境变量指定的路径中寻找没加任何前缀的文件名。如果有需要，运行时(run-time)支持文件`libgcc.a'也在`-B'前缀的搜索范围之内。如果这里没有找到，就在上面提到的两个标准前缀中寻找，仅此而已。如果上述方法没有找到这个文件，就不链接它了。多数情况的多数机器上，`libgcc.a'并非必不可少。

可以通过环境变量GCC\_EXEC\_PREFIX获得近似的效果；如果定义了这个变量，其值就和上面说的一样被用作前缀。如果同时指定了`-B'选项和GCC\_EXEC\_PREFIX变量，编译器首先使用`-B'选项，然后才尝试环境变量值。

### 2.9 ld/objdump/objcopy选项

我们在开发APP时，一般不需要直接调用这3个命令；在开发裸机、bootloader时，或是调试APP时会涉及，到时再讲。

## 第三章 Makefile的使用

在Linux中使用make命令来编译程序，特别是大程序；而make命令所执行的动作依赖于Makefile文件。最简单的Makefile文件如下：

hello: hello.c

gcc -o hello hello.c

clean:

rm -f hello

将上述4行存为Makefile文件(注意必须以Tab键缩进第2、4行，不能以空格键缩进)，放入01\_hello目录下，然后直接执行make命令即可编译程序，执行“make clean”即可清除编译出来的结果。

make命令根据文件更新的时间戳来决定哪些文件需要重新编译，这使得可以避免编译已经编译过的、没有变化的程序，可以大大提高编译效率。

要想完整地了解Makefile的规则，请参考《GNU Make 使用手册》，以下仅粗略介绍。

### 3.1 配套视频内容大纲

#### 3.1.1 Makefile规则与示例

参考文档：gunmake.htm

① 为什么需要Makefile

怎么高效地编译程序？

想达到什么样的效果？请参考Visual Studio：修改源文件或头文件，只需要重新编译牵涉到的文件，就可以重新生成APP

② Makefile其实挺简单

一个简单的Makefile文件包含一系列的“规则”，其样式如下：

目标(target)…: 依赖(prerequiries)…

<tab>命令(command)

如果“依赖文件”比“目标文件”更加新，那么执行“命令”来重新生成“目标文件”。

命令被执行的2个条件：依赖文件比目标文件新，或是 目标文件还没生成。

③ 先介绍Makefile的2个函数

A. $(foreach var,list,text)

简单地说，就是 for each var in list, change it to text。

对list中的每一个元素，取出来赋给var，然后把var改为text所描述的形式。

例子：

objs := a.o b.o

dep\_files := $(foreach f, $(objs), .$(f).d) // 最终 dep\_files := .a.o.d .b.o.d

B. $(wildcard pattern)

pattern所列出的文件是否存在，把存在的文件都列出来。

例子：

src\_files := $( wildcard \*.c) // 最终 src\_files中列出了当前目录下的所有.c文件

④ 一步一步完善Makefile

第1个Makefile，简单粗暴，效率低：

test : main.c sub.c sub.h

gcc -o test main.c sub.c

第2个Makefile，效率高，相似规则太多太啰嗦，不支持检测头文件：

test : main.o sub.o

gcc -o test main.o sub.o

main.o : main.c

gcc -c -o main.o main.c

sub.o : sub.c

gcc -c -o sub.o sub.c

clean:

rm \*.o test -f

第3个Makefile，效率高，精炼，不支持检测头文件：

test : main.o sub.o

gcc -o test main.o sub.o

%.o : %.c

gcc -c -o $@ $<

clean:

rm \*.o test -f

第4个Makefile，效率高，精炼，支持检测头文件(但是需要手工添加头文件规则)：

test : main.o sub.o

gcc -o test main.o sub.o

%.o : %.c

gcc -c -o $@ $<

sub.o : sub.h

clean:

rm \*.o test -f

第5个Makefile，效率高，精炼，支持自动检测头文件：

objs := main.o sub.o

test : $(objs)

gcc -o test $^

# 需要判断是否存在依赖文件

# .main.o.d .sub.o.d

dep\_files := $(foreach f, $(objs), .$(f).d)

dep\_files := $(wildcard $(dep\_files))

# 把依赖文件包含进来

ifneq ($(dep\_files),)

include $(dep\_files)

endif

%.o : %.c

gcc -Wp,-MD,.$@.d -c -o $@ $<

clean:

rm \*.o test -f

distclean:

rm $(dep\_files) \*.o test -f

#### 3.1.2 通用Makefile的使用

我参考Linux内核的Makefile编写了一个通用的Makefile，它可以用来编译应用程序：

① 支持多个目录、多层目录、多个文件；

② 支持给所有文件设置编译选项；

③ 支持给某个目录设置编译选项；

④ 支持给某个文件单独设置编译选项；

⑤ 简单、好用。

使用git下载本教程的文档后，下列目录中就有说明和示例：

01\_all\_series\_quickstart\

04\_嵌入式Linux应用开发基础知识\source\05\_general\_Makefile

#### 3.1.3 通用Makefile的解析

① 零星知识点

A. make命令的使用：

执行make命令时，它会去当前目录下查找名为“Makefile”的文件，并根据它的指示去执行操作，生成第一个目标。

我们可以使用“-f”选项指定文件，不再使用名为“Makefile”的文件，比如：

make -f Makefile.build

我们可以使用“-C”选项指定目录，切换到其他目录里去，比如：

make -C a/ -f Makefile.build

我们可以指定目标，不再默认生成第一个目标：

make -C a/ -f Makefile.build other\_target

B. 即时变量、延时变量：

变量的定义语法形式如下：

A = xxx // 延时变量

B ?= xxx // 延时变量，只有第一次定义时赋值才成功；如果曾定义过，此赋值无效

C := xxx // 立即变量

D += yyy // 如果D在前面是延时变量，那么现在它还是延时变量；

// 如果D在前面是立即变量，那么现在它还是立即变量

在GNU make中对变量的赋值有两种方式：延时变量、立即变量。

上图中，变量A是延时变量，它的值在使用时才展开、才确定。比如：

A = $@

test:

@echo $A

上述Makefile中，变量A的值在执行时才确定，它等于test，是延时变量。

如果使用“A := $@”，这是立即变量，这时$@为空，所以A的值就是空。

C. 变量的导出(export)：

在编译程序时，我们会不断地使用“make -C dir”切换到其他目录，执行其他目录里的Makefile。如果想让某个变量的值在所有目录中都可见，要把它export出来。

比如“CC = $(CROSS\_COMPILE)gcc”，这个CC变量表示编译器，在整个过程中都是一样的。定义它之后，要使用“export CC”把它导出来。

D. Makefile中可以使用shell命令：

比如：

TOPDIR := $(shell pwd)

这是个立即变量，TOPDIR等于shell命令pwd的结果。

E. 在Makefile中怎么放置第1个目标：

执行make命令时如果不指定目标，那么它默认是去生成第1个目标。

所以“第1个目标”，位置很重要。有时候不太方便把第1个目标完整地放在文件前面，这时可以在文件的前面直接放置目标，在后面再完善它的依赖与命令。比如：

First\_target: // 这句话放在前面

．．．． // 其他代码，比如include其他文件得到后面的xxx变量

First\_target : $(xxx) $(yyy) // 在文件的后面再来完善

command

F. 假想目标：

我们的Makefile中有这样的目标：

clean:

rm -f $(shell find -name "\*.o")

rm -f $(TARGET)

如果当前目录下恰好有名为“clean”的文件，那么执行“make clean”时它就不会执行那些删除命令。

这时我们需要把“clean”这个目标，设置为“假想目标”，这样可以确保执行“make clean”时那些删除命令肯定可以得到执行。

使用下面的语句把“clean”设置为假想目标：

.PHONY : clean

G. 常用的函数：

i. $(foreach var,list,text)

简单地说，就是 for each var in list, change it to text。

对list中的每一个元素，取出来赋给var，然后把var改为text所描述的形式。

例子：

objs := a.o b.o

dep\_files := $(foreach f, $(objs), .$(f).d) // 最终 dep\_files := .a.o.d .b.o.d

ii. $(wildcard pattern)

pattern所列出的文件是否存在，把存在的文件都列出来。

例子：

src\_files := $( wildcard \*.c) // 最终 src\_files中列出了当前目录下的所有.c文件

iii. $(filter pattern...,text)

把text中符合pattern格式的内容，filter(过滤)出来、留下来。

例子：

obj-y := a.o b.o c/ d/

DIR := $(filter %/, $(obj-y)) //结果为：c/ d/

iv. $(filter-out pattern...,text)

把text中符合pattern格式的内容，filter-out(过滤)出来、扔掉。

例子：

obj-y := a.o b.o c/ d/

DIR := $(filter-out %/, $(obj-y)) //结果为：a.o b.o

vi. $(patsubst *pattern*,*replacement*,*text*)

寻找`text’中符合格式`pattern’的字，用`replacement’替换它们。`pattern’和`replacement’中可以使用通配符。

比如：

subdir-y := c/ d/

subdir-y := $(patsubst %/, %, $(subdir-y)) // 结果为：c d

② 通用Makefile的设计思想

A. 在Makefile文件中确定要编译的文件、目录，比如：

obj-y += main.o

obj-y += a/

“Makefile”文件总是被“Makefile.build”包含的。

B. 在Makefile.build中设置编译规则，有3条编译规则：

i. 怎么编译子目录？ 进入子目录编译：

$(subdir-y):

make -C $@ -f $(TOPDIR)/Makefile.build

ii. 怎么编译当前目录中的文件？

%.o : %.c

$(CC) $(CFLAGS) $(EXTRA\_CFLAGS) $(CFLAGS\_$@) -Wp,-MD,$(dep\_file) -c -o $@ $<

iii. 当前目录下的.o和子目录下的built-in.o要打包起来：

built-in.o : $(cur\_objs) $(subdir\_objs)

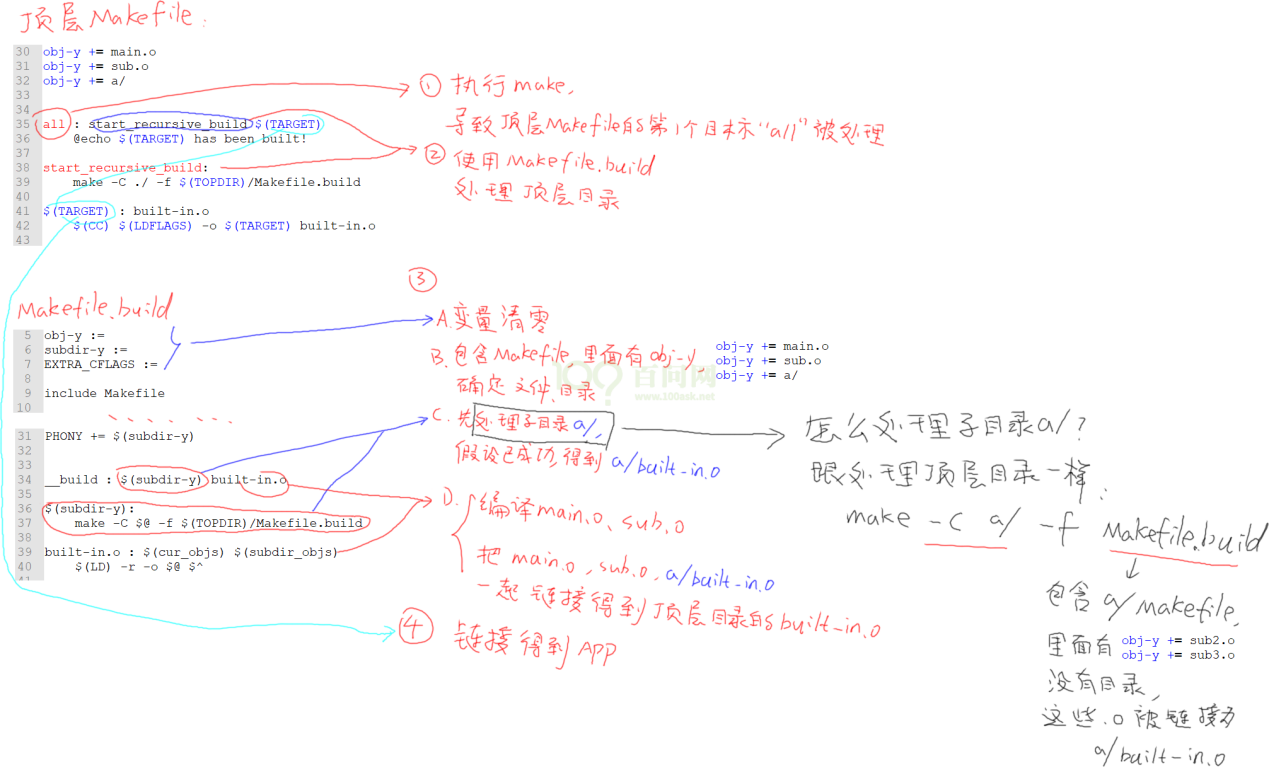
$(LD) -r -o $@ $^

C. 顶层Makefile中把顶层目录的built-in.o链接成APP：

$(TARGET) : built-in.o

$(CC) $(LDFLAGS) -o $(TARGET) built-in.o

③ 情景演绎



本节下面的内容中不需要看，这是为写书《嵌入式Linux应用开发完全手册 升级版》而准备的。结合3.0节看视频就可以了。

### 3.2 Makefile规则

一个简单的Makefile文件包含一系列的“规则”，其样式如下：

目标(target)…: 依赖(prerequiries)…

<tab>命令(command)

目标(target)通常是要生成的文件的名称，可以是可执行文件或OBJ文件，也可以是一个执行的动作名称，诸如`clean’。

依赖是用来产生目标的材料(比如源文件)，一个目标经常有几个依赖。

命令是生成目标时执行的动作，一个规则可以含有几个命令，每个命令占一行。

**注意**：每个命令行前面必须是一个Tab字符，即命令行第一个字符是Tab。这是容易出错的地方。

通常，如果一个依赖发生了变化，就需要规则调用命令以更新或创建目标。但是并非所有的目标都有依赖，例如，目标“clean”的作用是清除文件，它没有依赖。

规则一般是用于解释怎样和何时重建目标。make首先调用命令处理依赖，进而才能创建或更新目标。当然，一个规则也可以是用于解释怎样和何时执行一个动作，即打印提示信息。

一个Makefile文件可以包含规则以外的其他文本，但一个简单的Makefile文件仅仅需要包含规则。虽然真正的规则比这里展示的例子复杂，但格式是完全一样的。

对于上面的Makefile，执行“make”命令时，仅当hello.c文件比hello文件新，才会执行命令“arm-linux-gcc –o hello hello.c”生成可执行文件hello；如果还没有hello文件，这个命令也会执行。

运行“make clean”时，由于目标clean没有依赖，它的命令“rm -f hello”将被强制执行。

### 3.3 Makefile文件里的赋值方法

变量的定义语法形式如下：

immediate = deferred

immediate ?= deferred

immediate := immediate

immediate += deferred or immediate

define immediate

deferred

endef

在GNU make中对变量的赋值有两种方式：延时变量、立即变量。区别在于它们的定义方式和扩展时的方式不同，前者在这个变量使用时才扩展开，意即当真正使用时这个变量的值才确定；后者在定义时它的值就已经确定了。使用`=’，`?=’定义或使用define指令定义的变量是延时变量；使用`:=’定义的变量是立即变量。需要注意的一点是，`?=’仅仅在变量还没有定义的情况下有效，即`?=’被用来定义第一次出现的延时变量。

对于附加操作符`+=’，右边变量如果在前面使用（:=）定义为立即变量则它也是立即变量，否则均为延时变量。

### 3.4 Makefile常用函数

函数调用的格式如下：

$(function arguments)

这里`function’是函数名，`arguments’是该函数的参数。参数和函数名之间是用空格或Tab隔开，如果有多个参数，它们之间用逗号隔开。这些空格和逗号不是参数值的一部分。

内核的Makefile中用到大量的函数，现在介绍一些常用的。

#### 3.4.1 字符串替换和分析函数

（1）$(subst from,to,text)

在文本`text’中使用`to’替换每一处`from’。

比如：

$(subst ee,EE,feet on the street)

结果为‘fEEt on the strEEt’。

（2）$(patsubst pattern,replacement,text)

寻找`text’中符合格式`pattern’的字，用`replacement’替换它们。`pattern’和`replacement’中可以使用通配符。

比如：

$(patsubst %.c,%.o,x.c.c bar.c)

结果为：`x.c.o bar.o’。

（3）$(strip string)

去掉前导和结尾空格，并将中间的多个空格压缩为单个空格。

比如：

$(strip a b c )

结果为`a b c’。

（4）$(findstring find,in)

在字符串`in’中搜寻`find’，如果找到，则返回值是`find’，否则返回值为空。

比如：

$(findstring a,a b c)

$(findstring a,b c)

将分别产生值`a’和`’(空字符串)。

（5）$(filter pattern...,text)

返回在`text’中由空格隔开且匹配格式`pattern...’的字，去除不符合格式`pattern...’的字。

比如：

$(filter %.c %.s,foo.c bar.c baz.s ugh.h)

结果为`foo.c bar.c baz.s’。

（6）$(filter-out pattern...,text)

返回在`text’中由空格隔开且不匹配格式`pattern...’的字，去除符合格式`pattern...’的字。它是函数filter的反函数。

比如：

$(filter %.c %.s,foo.c bar.c baz.s ugh.h)

结果为`ugh.h’。

（7）$(sort list)

将‘list’中的字按字母顺序排序，并去掉重复的字。输出由单个空格隔开的字的列表。

比如：

$(sort foo bar lose)

返回值是‘bar foo lose’。

#### 3.4.2 文件名函数

（1）$(dir names...)

抽取‘names...’中每一个文件名的路径部分，文件名的路径部分包括从文件名的首字符到最后一个斜杠(含斜杠)之前的一切字符。

比如：

$(dir src/foo.c hacks)

结果为‘src/ ./’。

（2）$(notdir names...)

抽取‘names...’中每一个文件名中除路径部分外一切字符（真正的文件名）。

比如：

$(notdir src/foo.c hacks)

结果为‘foo.c hacks’。

（3）$(suffix names...)

抽取‘names...’中每一个文件名的后缀。

比如：

$(suffix src/foo.c src-1.0/bar.c hacks)

结果为‘.c .c’。

（4）$(basename names...)

抽取‘names...’中每一个文件名中除后缀外一切字符。

比如：

$(basename src/foo.c src-1.0/bar hacks)

结果为‘src/foo src-1.0/bar hacks’。

（5）$(addsuffix suffix,names...)

参数‘names...’是一系列的文件名，文件名之间用空格隔开；suffix是一个后缀名。将suffix(后缀)的值附加在每一个独立文件名的后面，完成后将文件名串联起来，它们之间用单个空格隔开。

比如：

$(addsuffix .c,foo bar)

结果为‘foo.c bar.c’。

（6）$(addprefix prefix,names...)

参数‘names’是一系列的文件名，文件名之间用空格隔开；prefix是一个前缀名。将preffix(前缀)的值附加在每一个独立文件名的前面，完成后将文件名串联起来，它们之间用单个空格隔开。

比如：

$(addprefix src/,foo bar)

结果为‘src/foo src/bar’。

（7）$(wildcard pattern)

参数‘pattern’是一个文件名格式，包含有通配符(通配符和shell中的用法一样)。函数wildcard的结果是一列和格式匹配的且真实存在的文件的名称，文件名之间用一个空格隔开。

比如若当前目录下有文件1.c、2.c、1.h、2.h，则：

c\_src := $(wildcard \*.c)

结果为‘1.c 2.c’。

#### 3.4.3 其他函数

（1）$(foreach var,list,text)

前两个参数，‘var’和‘list’将首先扩展，注意最后一个参数‘text’此时不扩展；接着，‘list’扩展所得的每个字，都赋给‘var’变量；然后‘text’引用该变量进行扩展，因此‘text’每次扩展都不相同。

函数的结果是由空格隔开的‘text’ 在‘list’中多次扩展后，得到的新‘list’，就是说：‘text’多次扩展的字串联起来，字与字之间由空格隔开，如此就产生了函数foreach的返回值。

下面是一个简单的例子，将变量‘files’的值设置为 ‘dirs’中的所有目录下的所有文件的列表：

dirs := a b c d

files := $(foreach dir,$(dirs),$(wildcard $(dir)/\*))

这里‘text’是‘$(wildcard $(dir)/\*)’，它的扩展过程如下：

① 第一个赋给变量dir的值是`a’，扩展结果为‘$(wildcard a/\*)’；

② 第二个赋给变量dir的值是`b’，扩展结果为‘$(wildcard b/\*)’；

③ 第三个赋给变量dir的值是`c’，扩展结果为‘$(wildcard c/\*)’；

④ 如此继续扩展。

这个例子和下面的例有共同的结果：

files := $(wildcard a/\* b/\* c/\* d/\*)

（2）$(if condition,then-part[,else-part])

首先把第一个参数‘condition’的前导空格、结尾空格去掉，然后扩展。如果扩展为非空字符串，则条件‘condition’为‘真’；如果扩展为空字符串，则条件‘condition’为‘假’。

如果条件‘condition’为‘真’,那么计算第二个参数‘then-part’的值，并将该值作为整个函数if的值。

如果条件‘condition’为‘假’,并且第三个参数存在，则计算第三个参数‘else-part’的值，并将该值作为整个函数if的值；如果第三个参数不存在，函数if将什么也不计算，返回空值。

注意：仅能计算‘then-part’和‘else-part’二者之一，不能同时计算。这样有可能产生副作用（例如函数shell的调用）。

（3）$(origin variable)

变量‘variable’是一个查询变量的名称，不是对该变量的引用。所以，不能采用‘$’和圆括号的格式书写该变量，当然，如果需要使用非常量的文件名，可以在文件名中使用变量引用。

函数origin的结果是一个字符串，该字符串变量是这样定义的：

‘undefined' ：如果变量‘variable’从没有定义；

‘default' ：变量‘variable’是缺省定义；

‘environment' ：变量‘variable’作为环境变量定义，选项‘-e’没有打开；

‘environment override' ：变量‘variable’作为环境变量定义，选项‘-e’已打开；

‘file' ：变量‘variable’在Makefile中定义；

‘command line' ：变量‘variable’在命令行中定义；

‘override' ：变量‘variable’在Makefile中用override指令定义；

‘automatic' ：变量‘variable’是自动变量

（4）$(shell command arguments)

函数shell是make与外部环境的通讯工具。函数shell的执行结果和在控制台上执行‘command arguments’的结果相似。不过如果‘command arguments’的结果含有换行符（和回车符），则在函数shell的返回结果中将把它们处理为单个空格，若返回结果最后是换行符（和回车符）则被去掉。

比如当前目录下有文件1.c、2.c、1.h、2.h，则：

c\_src := $(shell ls \*.c)

结果为‘1.c 2.c’。

《Makefile介绍》这小节可以在阅读内核、bootloader、应用程序的Makefile文件时，作为手册来查询。下面以options程序的Makefile作为例子进行演示，Makefile的内容如下：

File: Makefile

01 src := $(shell ls \*.c)

02 objs := $(patsubst %.c,%.o,$(src))

03

04 test: $(objs)

05 gcc -o $@ $^

06

07 %.o:%.c

08 gcc -c -o $@ $<

09

10 clean:

11 rm -f test \*.o

上述Makefile中$@、$^、$<称为自动变量。$@表示规则的目标文件名；$^表示所有依赖的名字，名字之间用空格隔开；$<表示第一个依赖的文件名。‘%’是通配符，它和一个字符串中任意个数的字符相匹配。

options目录下所有的文件为main.c，Makefile，sub.c和sub.h，下面一行行地分析：

① 第1行src变量的值为‘main.c sub.c’。

② 第2行objs变量的值为‘main.o sub.o’，是src变量经过patsubst函数处理后得到的。

③ 第4行实际上就是：

test : main.o sub.o

目标test的依赖有二：main.o和sub.o。开始时这两个文件还没有生成，在执行生成test的命令之前先将main.o、sub.o作为目标查找到合适的规则，以生成main.o、sub.o。

④ 第7、8行就是用来生成main.o、sub.o的规则：

对于main.o这个规则就是：

main.o:main.c

gcc -c -o main.o main.c

对于sub.o这个规则就是：

sub.o:sub.c

gcc -c -o sub.o sub.c

这样，test的依赖main.o和sub.o就生成了。

⑤ 第5行的命令在生成main.o、sub.o后得以执行。

在options目录下第一次执行make命令可以看到如下信息：

gcc -c -o main.o main.c

gcc -c -o sub.o sub.c

gcc -o test main.o sub.o

然后修改sub.c文件，再次执行make命令，可以看到如下信息：

gcc -c -o sub.o sub.c

gcc -o test main.o sub.o

可见，只编译了更新过的sub.c文件，对main.c文件不用再次编译，节省了编译的时间。

## 第四章 文件IO

参考书：

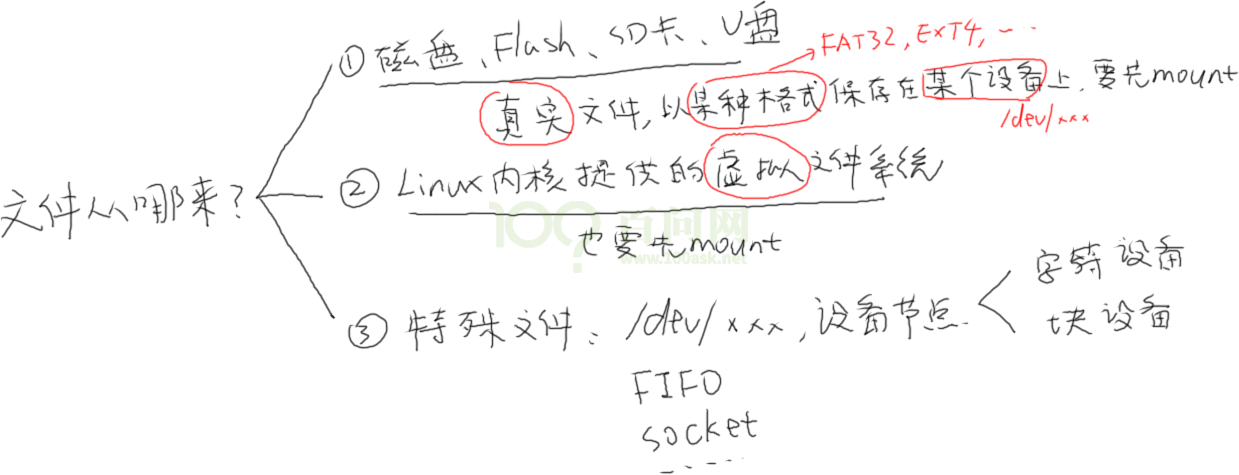


这2本书的内容类似，第一本对知识点有更细致的描述，适合初学者；第二本比较直接，一上来就是各种函数的介绍，适合当作字典，不懂时就去翻看一下。

做纯Linux应用的入门，看这2本书就可以了，不需要学习我们的视频。我们的侧重于“嵌入式Linux”。

在Linux系统中，一切都是“文件”：普通文件、驱动程序、网络通信等等。所有的操作，都是通过“文件IO”来操作的。所以，很有必要掌握文件操作的常用接口。

### 4.1 文件从哪来？



### 4.2 怎么访问文件？

#### 4.2.1 通用的IO模型：open/read/write/lseek/close

使用GIT下载所有源码后，本节源码位于如下目录：

01\_all\_series\_quickstart\

04\_嵌入式Linux应用开发基础知识\source\06\_fileio\copy.c

copy.c源码如下：

01

02 #include <sys/types.h>

03 #include <sys/stat.h>

04 #include <fcntl.h>

05 #include <unistd.h>

06 #include <stdio.h>

07

08 /\*

09 \* ./copy 1.txt 2.txt

10 \* argc = 3

11 \* argv[0] = "./copy"

12 \* argv[1] = "1.txt"

13 \* argv[2] = "2.txt"

14 \*/

15 int main(int argc, char \*\*argv)

16 {

17 int fd\_old, fd\_new;

18 char buf[1024];

19 int len;

20

21 /\* 1. 判断参数 \*/

22 if (argc != 3)

23 {

24 printf("Usage: %s <old-file> <new-file>\n", argv[0]);

25 return -1;

26 }

27

28 /\* 2. 打开老文件 \*/

29 fd\_old = open(argv[1], O\_RDONLY);

30 if (fd\_old == -1)

31 {

32 printf("can not open file %s\n", argv[1]);

33 return -1;

34 }

35

36 /\* 3. 创建新文件 \*/

37 fd\_new = open(argv[2], O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH);

38 if (fd\_new == -1)

39 {

40 printf("can not creat file %s\n", argv[2]);

41 return -1;

42 }

43

44 /\* 4. 循环： 读老文件-写新文件 \*/

45 while ((len = read(fd\_old, buf, 1024)) > 0)

46 {

47 if (write(fd\_new, buf, len) != len)

48 {

49 printf("can not write %s\n", argv[2]);

50 return -1;

51 }

52 }

53

54 /\* 5. 关闭文件 \*/

55 close(fd\_old);

56 close(fd\_new);

57

58 return 0;

59 }

60

本节源码完全可以在Ubuntu上测试，跟在ARM板上没什么不同。

执行以下命令编译、运行：

$ gcc -o copy copy.c

$ ./copy copy.c new.c

#### 4.2.2 不是通用的函数：ioctl/mmap

使用GIT下载所有源码后，本节源码位于如下目录：

01\_all\_series\_quickstart\

04\_嵌入式Linux应用开发基础知识\source\06\_fileio\copy\_mmap.c

在Linux中，还可以把一个文件的所有内容映射到内存，然后直接读写内存即可读写文件。

copy\_mmap.c源码如下：

01

02 #include <sys/types.h>

03 #include <sys/stat.h>

04 #include <fcntl.h>

05 #include <unistd.h>

06 #include <stdio.h>

07 #include <sys/mman.h>

08

09 /\*

10 \* ./copy 1.txt 2.txt

11 \* argc = 3

12 \* argv[0] = "./copy"

13 \* argv[1] = "1.txt"

14 \* argv[2] = "2.txt"

15 \*/

16 int main(int argc, char \*\*argv)

17 {

18 int fd\_old, fd\_new;

19 struct stat stat;

20 char \*buf;

21

22 /\* 1. 判断参数 \*/

23 if (argc != 3)

24 {

25 printf("Usage: %s <old-file> <new-file>\n", argv[0]);

26 return -1;

27 }

28

29 /\* 2. 打开老文件 \*/

30 fd\_old = open(argv[1], O\_RDONLY);

31 if (fd\_old == -1)

32 {

33 printf("can not open file %s\n", argv[1]);

34 return -1;

35 }

36

37 /\* 3. 确定老文件的大小 \*/

38 if (fstat(fd\_old, &stat) == -1)

39 {

40 printf("can not get stat of file %s\n", argv[1]);

41 return -1;

42 }

43

44 /\* 4. 映射老文件 \*/

45 buf = mmap(NULL, stat.st\_size, PROT\_READ, MAP\_SHARED, fd\_old, 0);

46 if (buf == MAP\_FAILED)

47 {

48 printf("can not mmap file %s\n", argv[1]);

49 return -1;

50 }

51

52 /\* 5. 创建新文件 \*/

53 fd\_new = open(argv[2], O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH);

54 if (fd\_new == -1)

55 {

56 printf("can not creat file %s\n", argv[2]);

57 return -1;

58 }

59

60 /\* 6. 写新文件 \*/

61 if (write(fd\_new, buf, stat.st\_size) != stat.st\_size)

62 {

63 printf("can not write %s\n", argv[2]);

64 return -1;

65 }

66

67 /\* 5. 关闭文件 \*/

68 close(fd\_old);

69 close(fd\_new);

70

71 return 0;

72 }

73

本节源码完全可以在Ubuntu上测试，跟在ARM板上没什么不同。

执行以下命令编译、运行：

$ gcc -o copy\_mmap copy\_mmap.c

$ ./copy\_mmap copy\_mmap.c new2.c

### 4.3 怎么知道这些函数的用法？

Linux下有3大帮助方法：help、man、info。

想查看某个命令的用法时，比如查看ls命令的用法，可以执行：

ls --help

help只能用于查看某个命令的用法，而man手册既可以查看命令的用法，还可以查看函数的详细介绍等等。它含有9大分类，如下：

1 Executable programs or shell commands // 命令

2 System calls (functions provided by the kernel) // 系统调用，比如 man 2 open

3 Library calls (functions within program libraries) // 函数库调用

4 Special files (usually found in /dev) // 特殊文件, 比如 man 4 tty

5 File formats and conventions eg /etc/passwd // 文件格式和约定, 比如man 5 passwd

6 Games // 游戏

7 Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7) //杂项

8 System administration commands (usually only for root) // 系统管理命令

9 Kernel routines [Non standard] // 内核例程

比如想查看open函数的用法时，可以直接执行“man open”，发现这不是想要内容时再执行“man 2 open”。

在man命令中可以及时按“h”查看帮助信息了解快捷键。常用的快捷键是：

f 往前翻一页

b 往后翻一页

/patten 往前搜

?patten 往后搜

就内容来说，info手册比man手册编写得要更全面，但man手册使用起来更容易些。

以书来形容info手册和man手册的话，info手册相当于一章，里面含有若干节，阅读时你需要掌握如果从这一节跳到下一节；而man手册只相当于一节，阅读起来当然更容易。

就个人而言，我很少使用info命令。

可以直接执行“info”命令后，输入“H”查看它的快捷键，在info手册中，某一节被称为“node”，常用的快捷键如下：

Up Move up one line.

Down Move down one line.

PgUp Scroll backward one screenful.

PgDn Scroll forward one screenful.

Home Go to the beginning of this node.

End Go to the end of this node.

TAB Skip to the next hypertext link.

RET Follow the hypertext link under the cursor.

l Go back to the last node seen in this window.

[ Go to the previous node in the document.

] Go to the next node in the document.

p Go to the previous node on this level.

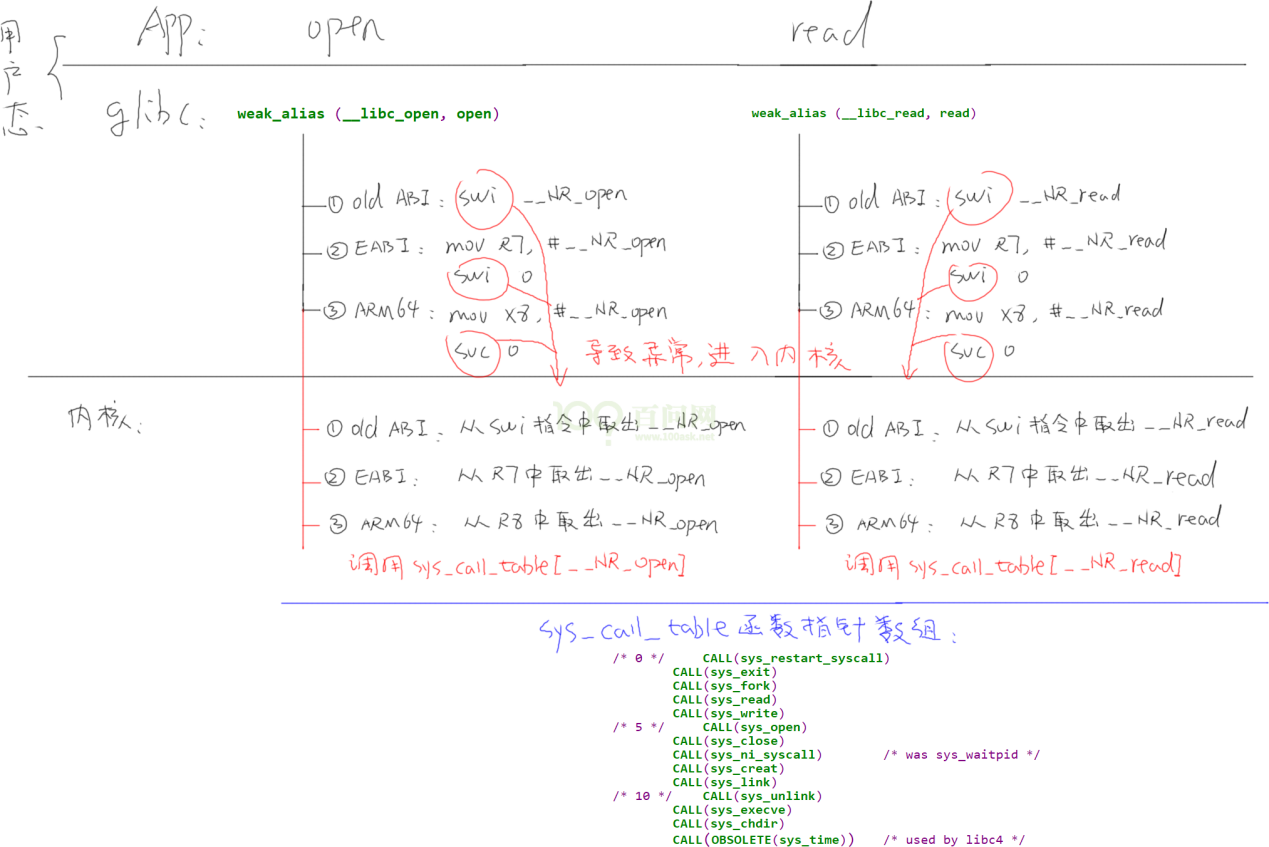
n Go to the next node on this level.

u Go up one level.

t Go to the top node of this document.

d Go to the main 'directory' node.

### 4.4 系统调用函数怎么进入内核？



### 4.5 内核的sys\_open、sys\_read会做什么？

