第009课 gcc和arm-linux-gcc和Makefile

来自百问网嵌入式Linux wiki

目录

第001节_gcc编译器1_gcc常用选项__gcc编译过程详解

gcc的使用方法

```
gcc [选项] 文件名
```

gcc常用选项

选项	功能
-v	查看gcc编译器的版本，显示gcc执行时的详细过程
-o <file>	指定输出文件名为file，这个名称不能跟源文件名同名
-E	只预处理，不会编译、汇编、链接t
-S	只编译，不会汇编、链接
-c	编译和汇编，不会链接

一个c/c++文件要经过预处理、编译、汇编和链接才能变成可执行文件。

- （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。

- （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)。**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。

hello.c(预处理)->hello.i(编译)->hello.s(汇编)->hello.o(链接)->hello

详细的每一步命令如下：

```
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
```

上面一连串命令比较麻烦，gcc会对.c文件默认进行预处理操作，使用-c再来指明了编译、汇编，从而得到.o文件, 再将.o文件进行链接，得到可执行应用程序。简化如下：

```
gcc -c -o hello.o hello.c
gcc -o hello hello.o
```

第002节_gcc编译器2_深入讲解链接过程

前面编译出来的可执行文件比源代码大了很多，这是什么原因呢？

我们从链接过程来分析，链接将汇编生成的OBJ文件、系统库的OBJ文件、库文件链接起来，crt1.o、crti.o、crtbegin.o、crtend.o、crtn.o这些都是gcc加入的系统标准启动文件，它们的加入使最后出来的可执行文件相原来大了很多。

-lc：链接libc库文件，其中libc库文件中就实现了printf等函数。

gcc -v -nostdlib -o hello hello.o:

会提示因为没有链接系统标准启动文件和标准库文件，而链接失败。

这个-nostdlib选项常用于裸机bootloader、linux内核等程序，因为它们不需要启动文件、标准库文件。

一般应用程序才需要系统标准启动文件和标准库文件。 裸机/bootloader、linux内核等程序不需要启动文件、标准库文件。

- **动态链接**使用动态链接库进行链接，生成的程序在执行的时候需要加载所需的动态库才能运行。

动态链接生成的程序体积较小，但是必须依赖所需的动态库，否则无法执行。

```
gcc -c -o hello.o hello.c
gcc -o hello_shared  hello.o
```

- **静态链接**使用静态库进行链接，生成的程序包含程序运行所需要的全部库，可以直接运行，

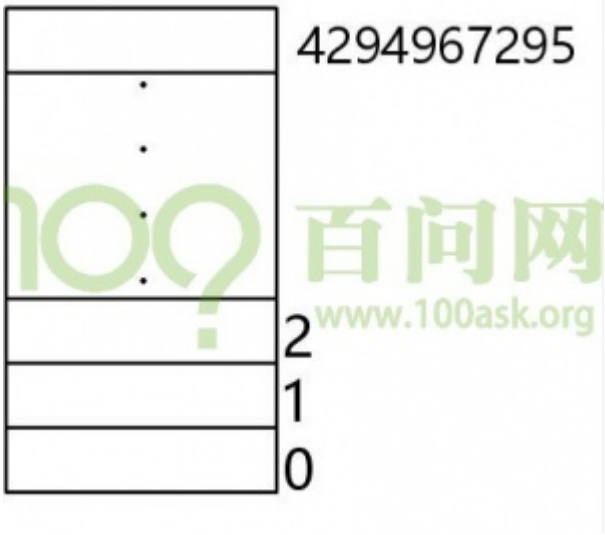
不过静态链接生成的程序体积较大。

```
gcc -c -o hello.o hello.c
gcc -static -o hello_static hello.o
```

第003节_c语言指针复习1__指向char和int的指针

日常中，我们把笔记写到记事本中，记事本就相当于一个载体（存储笔记的内容）。
C语言中有些变量，例如，char、int类型的变量，它们也需要一个载体，来存储这些变量的值，这个载体就是内存。
比如我们的电脑内存有4GB内存，也就是**4*1024*1024*1024=4294967296**字节。

我们可以把整个内存想象成一串连续格子，每个格子(字节)都可以放入一个数据，如下图所示。



每一个小格子都有一个编号，小格子的编号从0开始，我们可以通过读取格子的编号，得到格子里面的内容。同理，我们根据内存的变量的地址，来获得其中的数据。
下面写个小程序进行测试，实例：

point_test.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("sizeof(char   ) = %d\n",sizeof(char   ));
   printf("sizeof(int    ) = %d\n",sizeof(int    ));
   printf("sizeof(char  *) = %d\n",sizeof(char  *));
   printf("sizeof(char  **) = %d\n",sizeof(char **));

   return 0;
}
```

根据程序可以看出来，函数的功能是输出,char,int,char **类型所占据的字节数；

编译

```
gcc -o pointer_test pointer_test.c
```

运行应用程序：

```
./pointer_test
```

结果：（我用的是64位的编译器）

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 8
sizeof(char **) = 8
```

可以看出在64位的机器中，用8个字节表示指针，我们可以测试一下用32位的机器编译

编译：

```
gcc -m32 -o pointer_test pointer_test.c   //加上-m32:编译成32位的机器码
```

编译可能会出现下面提示错误：

```
/usr/include/features.h:374:25: fatal error: sys/cdefs.h: No such file or directory
```

解决错误，安装lib32readline-gplv2-dev，执行：

```
sudo apt-get install lib32readline-gplv2-dev
```

重新编译

```
gcc -m32 -o pointer_test pointer_test.c    //没有错误
```

运行生成的应用程序

```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
```

可以看出编译成32位的机器码，指针就是用4个字节来存储的，

总结：

1. 所用变量不论是普通变量(char,int)还是指针变量，都存在内存中。

2. 所用变量都可以保存某些值。

3. 怎么使用指针?

取值

移动指针

实例0

- 步骤一

```
#include <stdio.h>

void test0()
```

```
{
    char c;
    char *pc;

    /*第一步：所有变量都保存在内存中，我们打印一下变量的存储地址*/
    printf("&c  =%p\n",&c);
    printf("&pc =%p\n",&pc);

}

int main(int argc, char *argv[])
{
    printf("sizeof(char   ) = %d\n",sizeof(char   ));
    printf("sizeof(int    ) = %d\n",sizeof(int    ));
    printf("sizeof(char  *) = %d\n",sizeof(char  *));
    printf("sizeof(char **) = %d\n",sizeof(char **));
    printf("//===============\n");
    test0();

    return 0;
}
```

- 编译：

```
gcc -m32 -o pointer_test pointer_test.c
```

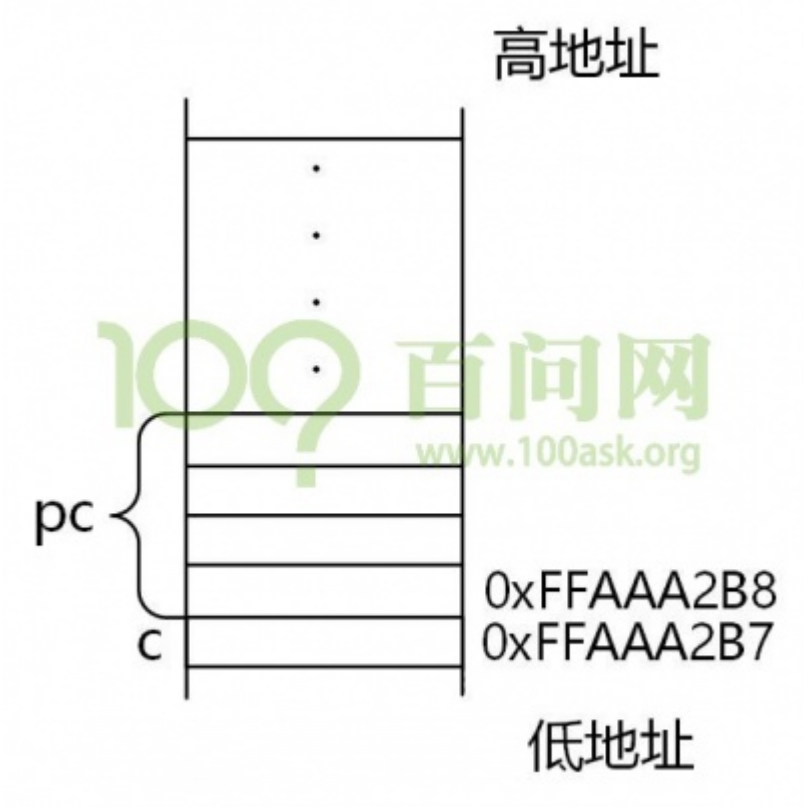
- 运行：

```
./pointer_test
```

- 结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//===============
&c  =0xffaaa2b7
&pc =0xffaaa2b8
```

从运行的结果我们可知，变量c的地址编号(即地址)是0xffaaa2b7，指针变量pc的地址编号是0xffaaa2b8，如下图所示，编译成32位的机器码，字符类型占用一个字节，指针类型就是用4个字节来存储的。



- 步骤二

我们把test0()函数里面的变量保存(赋予)一些值，假如这些变量不保存数据的话，那么存储该变量的地址空间就会白白浪费，就相当于买个房子不住，就会白白浪费掉。

我们把上面程序中的test0()函数里面的字符变量c,指针变量pc进行赋值。

```
c = ‘A’ ; //把字符 ‘A’ 赋值给字符变量c
pc = &c; //把字符变量c的地址赋值给指针变量pc
```

然后把赋值后变量的值打印出来

```
printf("c =%c\n",c);
printf("pc =%p\n",pc)
```

编译：

```
gcc -m32 -o pointer_test pointer_test.c
```

运行：

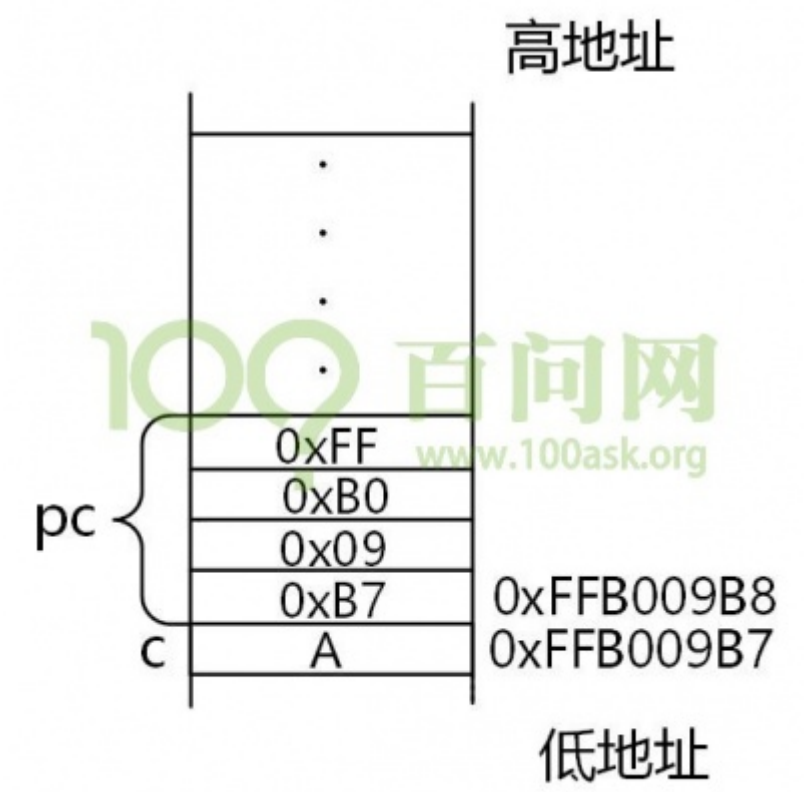
```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//===============
&c  = 0xffb009b7
&pc = 0xffb009b8
c  =  A
pc = 0xffb009b7
```

从运行的结构来看字符变量和指针变量的地址编号发成了变化，所以在程序重新运行时，变量的地址，具有不确定性，字符变量c存储的内容是字符‘A’，指针变量pc存储的内容是0xffb009b7（用四个字节来存储）。

由于内存的存储方式是，小端模式：低字节的数据放在低地址，高字节的数据放在高地址。在内存中的存储格式如下图所示。



- 步骤三

我们辛辛苦苦定义的指针类型变量，我们要把他用起来了，下面我们来分析一下，用指针来取值，‘*’：表示取指针变量存储地址的数据。

我们在test0()函数里面添加如下代码：

```
printf("*pc =%c\n",*pc);
printf("//==================\n");
```

编译：

```
gcc -m32 -o pointer_test pointer_test.c
```

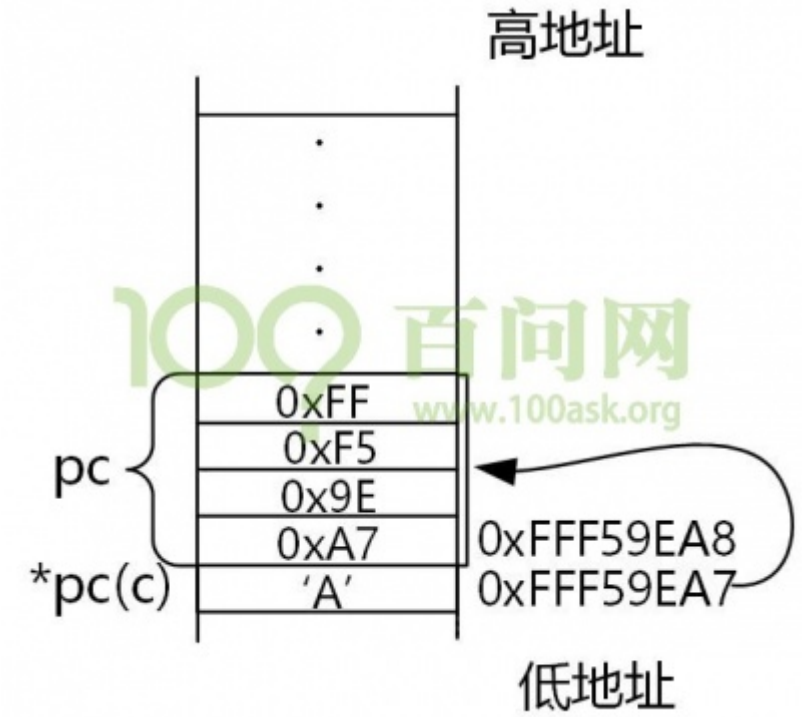
运行：

```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//===============
&c  =0xfff59ea7
&pc =0xfff59ea8
c  =A
pc =0xfff59ea7
*pc =A
//==================
```

指针变量pc存储的内容是是字符变量c的地址，所以*pc就想相当于取字符变量c的内容。如图



实例1

- 步骤一

我们在上面函数的基础上，写一个函数test1()

```
void test1()
{
   int  ia;
   int  *pi;
   char *pc;
```

```
    /*第一步：所有变量都保存在内存中，我们打印一下变量的存储地址*/
    printf("&ia =%p\n",&ia);
    printf("&pi =%p\n",&pi);
    printf("&pc =%p\n",&pc);
}
```

main.c

```
int main(int argc, char *argv[])
{
    printf("sizeof(char   ) = %d\n",sizeof(char   ));
    printf("sizeof(int    ) = %d\n",sizeof(int    ));
    printf("sizeof(char  *) = %d\n",sizeof(char  *));
    printf("sizeof(char **) = %d\n",sizeof(char **));
    printf("//===============\n");
    //test0();
    test1();
    return 0;
}
```

我们在test1()函数中定义了一个整型变量ia，定义了一个指向整型的指针变量pi，定义了一个指向字符型的指针变量pc。然后打印出这些变量的地址。

编译

```
gcc -m32 -o pointer_test pointer_test.c
```

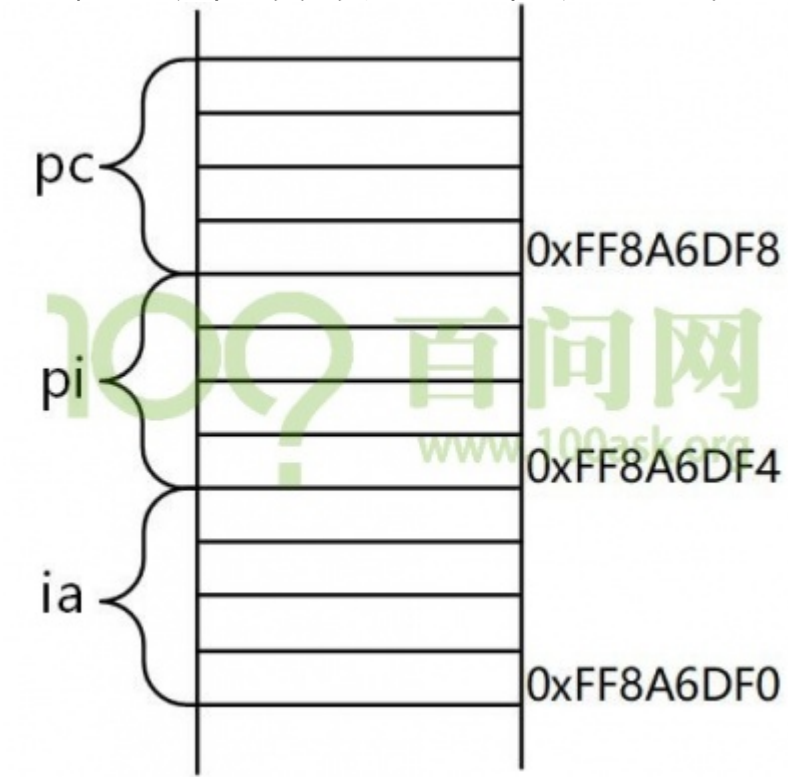
运行：

```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//===============
&ia =0xffc936e4
&pi =0xffc936e8
&pc =0xffc936ec
```

在32位的系统中int类型变量在内存中占用4个字节，指针型变量在内存中占用4个字节如图:



- 步骤二

在test1()的函数中对定义的变量进行赋值，然后把赋值的结果打印出来。

```
/*第二步：所有变量都可以保存某些值,接着赋值并打印*/
   ia = 0x12345678;
   pi = &ia;
   pc = (char *)&ia;
   printf("ia =0x%x\n",ia);
   printf("pi =%p\n",pi);
   printf("pc =%p\n",pc);
```

编译

```
gcc -m32 -o pointer_test pointer_test.c
```

运行：

```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//===============
&ia = 0xffb6f724
&pi = 0xffb6f728
&pc = 0xffb6f72c
ia = 0x12345678
```

```
pi = 0xffb6f724
pc = 0xffb6f724
```

从结果可以看出来，变量pi和pc的值都等于变量ia的地址。

- 步骤三

我们使用指针并且对其进行取值，然后移动指针，在test1中添加如下代码，完成所述要求

```
/*第三步：使用指针：1)取值  2)移动指针*/
```

printf("*pi =0x%x\n",*pi); printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc); pc=pc+1; printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc); pc=pc+1; printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc); pc=pc+1; printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc); printf("//==================\n");

编译

```
gcc -m32 -o pointer_test pointer_test.c
```

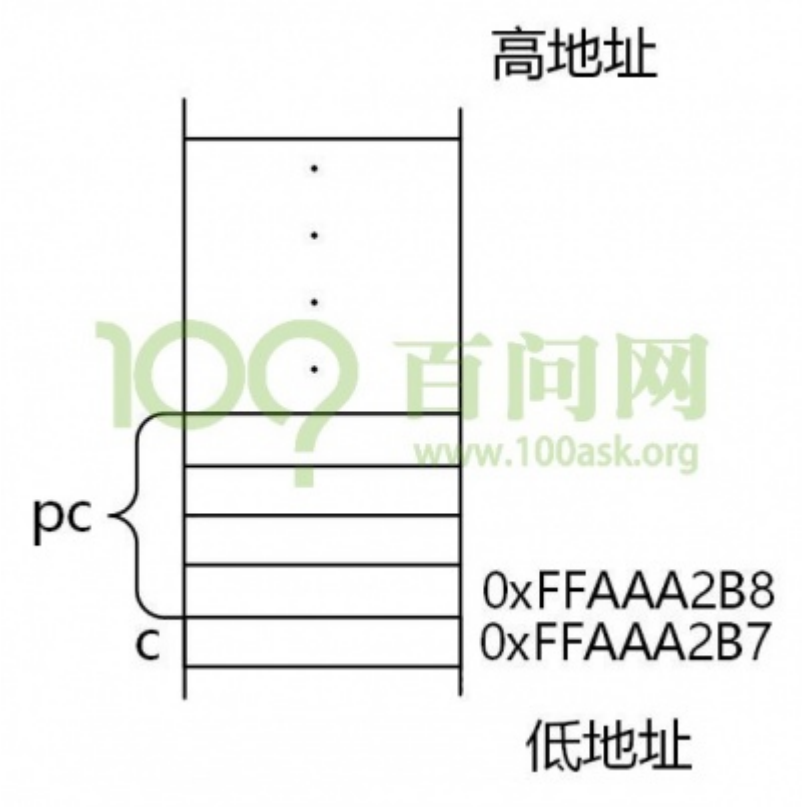
运行：

```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//==============
&ia =0xffee0930
&pi =0xffee0934
&pc =0xffee0938
ia =0x12345678
pi =0xffee0930
pc =0xffee0930
*pi =0x12345678
pc =0xffee0930  *pc =0x78
pc =0xffee0931  *pc =0x56
pc =0xffee0932  *pc =0x34
pc =0xffee0933  *pc =0x12
```

由于pi指向了ia,所以*pi的值为0x12345678。由于pc也指向了ia,但是由于pc是字符型指针变量，一次只能访问一个字节，需要四次才能访问完。如图所示：



结论：

1. 指针变量所存储的内容是所指向的变量在内存中的起始地址。

2. &变量：

目的：获得变量在内存中的地址; 返回：变量在内存中起始地址;

第004节_c语言指针复习2_指向数组和字符串的指针

实例2

我们在pointer_test.c的文件中写一个test2()函数，我们定义一个有3个元素的字符数组初始化值分别为，‘A’，‘B’，‘C’，然后定义一个字符指针pc，把数组ca的首地址复制给字符指针pc,然后通过访问指针变量pc,来读取指针变量pc所指向地址的数据，代码如下：

```
void test2()
{
    char ca[3]={'A','B','C'};
    char *pc;

    /*第一步：所有变量都保存在内存中，我们打印一下变量的存储地址*/
    printf("ca  =%p\n",ca);
    printf("&pc =%p\n",&pc);

    /*第二步：所有变量都可以保存某些值,接着赋值并打印*/
    //前面已经有ca[3]={'A','B','C'};
    pc = ca;
    printf("pc =%p\n",pc);
```

```
    /*第三步：使用指针：1)取值 2)移动指针*/
    printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc); pc=pc+1;
    printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc); pc=pc+1;
    printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc);
    printf("//==================\n");
}
```

main()函数

```
int main(int argc,char **argv)
{
    printf("sizeof(char   )=%d\n",sizeof(char   ));
    printf("sizeof(int    )=%d\n",sizeof(int    ));
    printf("sizeof(char  *)=%d\n",sizeof(char  *));
    printf("sizeof(char **)=%d\n",sizeof(char **));
    printf("//==================\n");
    //test0();
    //test1();
    test2();
    return 0;
}
```

编译

```
gcc -m32 -o pointer_test pointer_test.c
```

运行：

```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//===============
ca  =0xffb946b9
&pc =0xffb946b4
pc =0xffb946b9
pc =0xffb946b9  *pc =0x41
pc =0xffb946ba  *pc =0x42
pc =0xffb946bb  *pc =0x43
//==================
```

分析：

- 第一步：

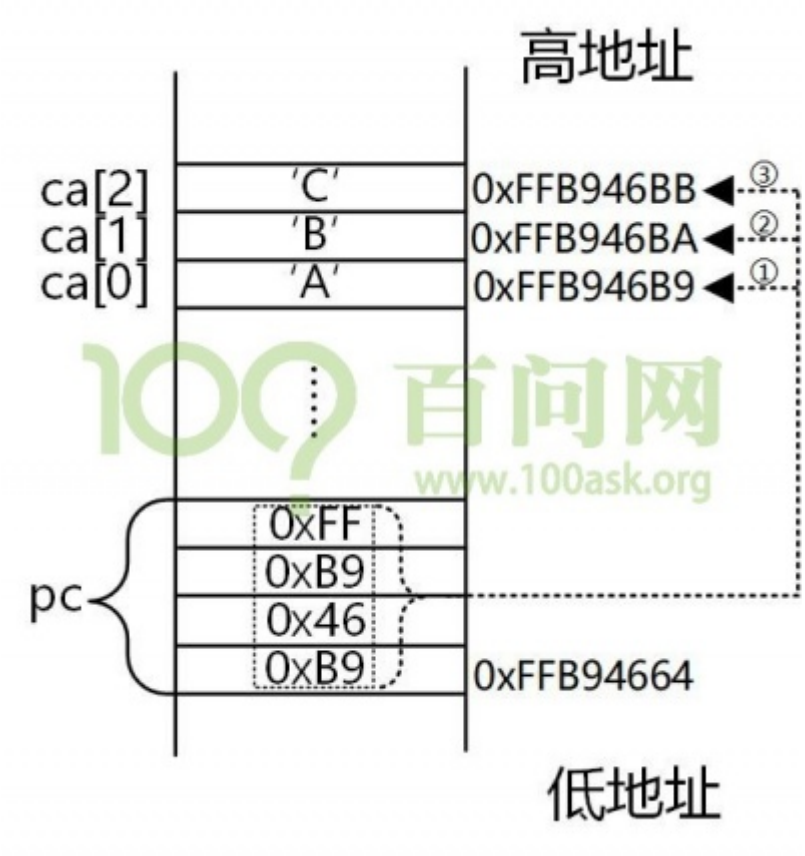
首先定义一个3个元素的字符数组ca(数组名表示该数组存储的首地址)，然后定义一个字符指针pc，然后通过printf()函数把定义这两个变量在内存中的地址打印出来。

- 第二步：

执行pc = ca;就是把数组ca的首地址复制给指针变量pc，然后通过printf()函数打印pc的值可以看出pc的值就是字符数组ca的首地址0xffb946b9。

- 第三步：

通过移动指针我们可以发现数组所占用的内存是连续的，0x41(的ascii值 ‘A ‘)，0x42(的ascii值 ‘B ‘)，0x43(的ascii值 ‘C ‘)。
如图



实例3

我们在pointer_test.c的文件中写一个test3()函数，我们定义一个有3个元素的整型数组ia,初始化值分别为，0x12345678, 0x87654321, 0x13572468，然后定义一个整型指针pi，把数组ia的首地址复制给整型指针pi,然后通过访问指针变量pi,来读取指针变量pi所指向地址的数据，代码如下：

```
void test3()
{
    int ia[3]={0x12345678,0x87654321,0x13572468};
    int *pi;

    /*第一步：所有变量都保存在内存中，我们打印一下变量的存储地址*/
    printf("ia  =%p\n",i);
    printf("&pi =%p\n",&pi);

    /*第二步：所有变量都可以保存某些值,接着赋值并打印*/
    //前面已经有ia[3]={0x12345678,0x87654321,0x13572468};
    pi = ia;
    printf("pi =%p\n",pi);

    /*第三步：使用指针：1)取值  2)移动指针*/
```

```
    printf("pi =%p\t",pi);   printf("*pi =0x%x\n",*pi); pi=pi+1;
    printf("pi =%p\t",pi);   printf("*pi =0x%x\n",*pi); pi=pi+1;
    printf("pi =%p\t",pi);   printf("*pi =0x%x\n",*pi);
    printf("//==================\n");
}
```

把main()函数test2()修改为test3().

编译

```
gcc -m32 -o pointer_test pointer_test.c
```

运行：

```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//===============
ia  =0xff91c060
&pi =0xff91c05c
pi =0xff91c060
pi =0xff91c060  *pi =0x12345678
pi =0xff91c064  *pi =0x87654321
pi =0xff91c068  *pi =0x13572468
```

分析：

- 第一步：

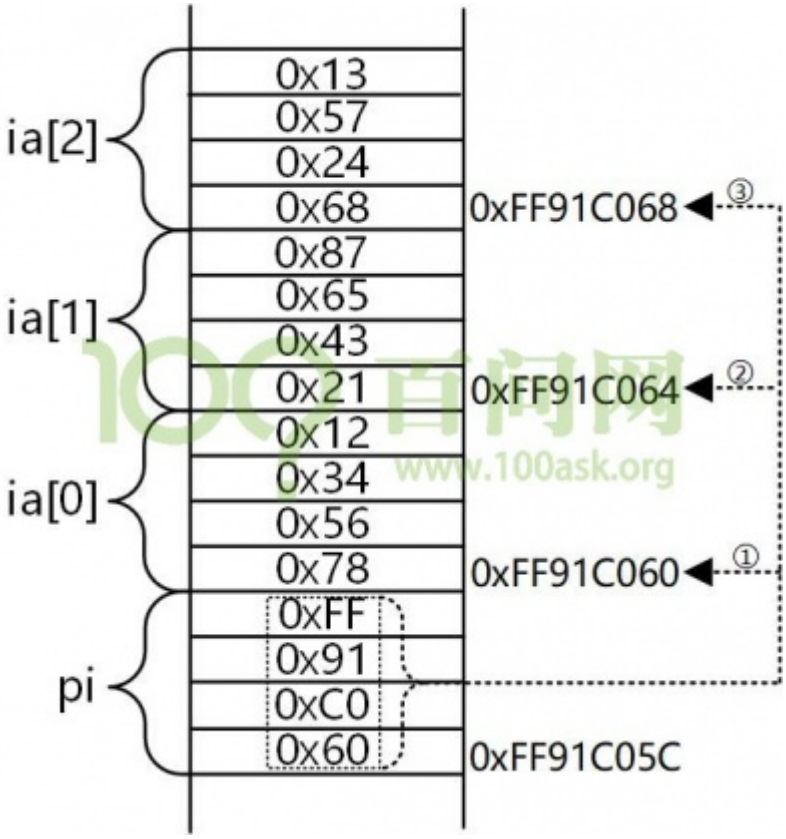
我们定义一个有3个元素的整型数组ia数组名表示该数组存储的首地址),初始化值分别为，0x12345678, 0x87654321, 0x13572468, 然后定义一个整型指针pi，然后通过printf()函数把定义这两个变量在内存中的地址打印出来。

- 第二步：

执行pi = ia; 就是把数组ia的首地址复制给指针变量pi，然后通过printf()函数打印pi的值可以看出pi的值就是整型数组ia的首地址0xff91c060。

- 第三步：

我们知道 pi是整型指针变量，并且整型变量占用四个字节，所以整型指针变量pi是以四字节为单元进行访问的，所以pi和pi+1之间的差是一个整型变量的大小（4个字节）。



实例4

定义一个指向字符串的指针pc，然后对字符串指针进行初始化设置为**abc**，代码如下：

```
void test4()
{
   char *pc="abc";
   /*第一步：所有变量都保存在内存中，我们打印一下变量的存储地址*/
   printf("&pc =%p\n",&pc);

   /*第二步：所有变量都可以保存某些值,接着赋值并打印*/
   //前面已经有pc="abc";

   /*第三步：使用指针：1)取值 2)移动指针*/
   printf("pc    =%p\n", pc);
   printf("*pc   =%c\n",*pc);
   printf("pc str=%s\n", pc);
}
```

把main()函数test3()修改为test4(). 编译

```
gcc -m32 -o pointer_test pointer_test.c
```

运行：

```
./pointer_test
```

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//==============
&pc   =0xfff49a68
pc    =0x08048b4b
*pc   =a
pc str=abc
```

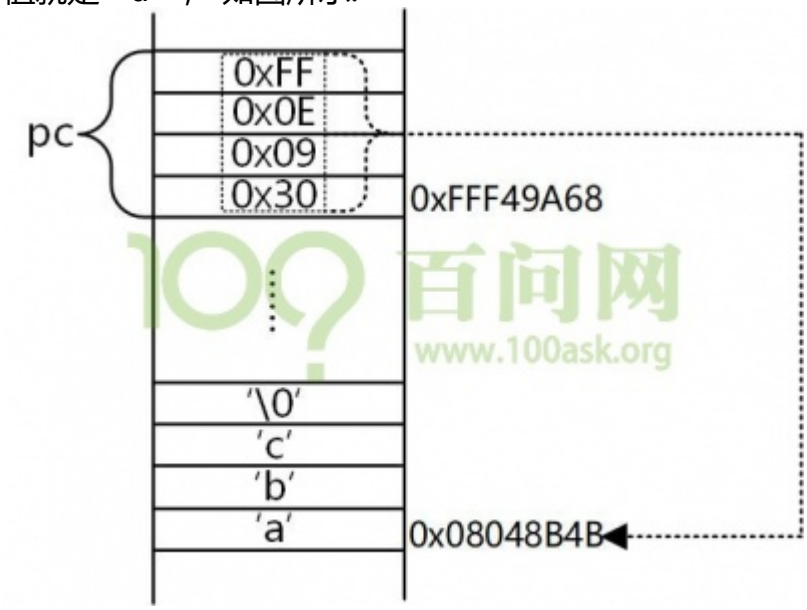
分析：

- 第一步：

定义一个指向字符串的指针pc，然后对字符串指针进行初始化设置为**abc**，此时，指针变量pc的值就是字符串**abc**的首地址，然后通过printf()函数把指针pc的地址打印出来为0xfff49a68

- 第二步：

首先通过printf()函数打印出指针变量pc的值(字符串**abc**的首地址)，pc的值为0x08048b4b，然后通过pc指针访问第一个字符(pc的就是字符串的首地址)，所以pc的值就是字符 ‘a ‘的地址，所以*pc的值就是’ a ‘， 如图所示:



下面分析一下指向数组的指针和指向字符串的指针:

```
char ca[3]={'A','B','C'};
char *pc0 = ca;
```

pc0是指向字符数组的字符指针,pc0就是数组首元素的地址,pc0=&a[0]

```
char *pc11="abc";
```

pc是指向字符串的字符指针,pc1就是字符串"abc"的首字符'a'的地址。

第005节_Makefile的引入及规则

使用keil, mdk, avr等工具开发程序时点击鼠标就可以编译了，它的内部机制是什么？它怎么组织管理程序？怎么决定编译哪一个文件？

答：实际上windows工具管理程序的内部机制，也是Makefile，我们在linux下来开发裸板程序的时候，使用Makefile组织管理这些程序，本节我们来讲解Makefile最基本的规则。Makefile要做什么事情呢？ 组织管理程序，组织管理文件，我们写一个程序来实验一下：

文件a.c

```
02   #include <stdio.h>
03
04   int main()
05   {
06   func_b();
07   return 0;
08}
```

文件b.c

```
2  #include <stdio.h>
3
4  void func_b()
5  {
6      printf("This is B\n");
7  }
```

编译：

```
gcc -o test a.c b.c
```

运行：

```
./test
```

结果：

```
This is B
```

gcc -o test a.c b.c这条命令虽然简单，但是它完成的功能不简单。 我们来看看它做了哪些事情。

我们知道.c程序 --> 得到可执行程序

它们之间要经过四个步骤：

1.预处理

2.编译

3.汇编

4.链接

我们经常把前三个步骤统称为编译了。我们具体分析：gcc -o test a.c b.c这条命令 它们要经过下面几个步骤：

1）.对于a.c执行：预处理 编译 汇编 的过程，a.c -->xxx.s -->xxx.o 文件。

2）.对于b.c执行：预处理 编译 汇编 的过程，b.c -->yyy.s -->yyy.o 文件。

3）.最后：xxx.o和yyy.o链接在一起得到一个test应用程序。

提示：gcc -o test a.c b.c -v ： 加上一个 ‘-v’ 选项可以看到它们的处理过程，

第一次编译a.c得到xxx.o文件，这是很合乎情理的， 执行完第一次之后，如果修改a.c 又再次执行：gcc -o test a.c b.c，对于a.c应该重新生成xxx.o，但是对于b.c又会重新编译一次，这完全没有必要，b.c根本没有修改，直接使用第一次生成的yyy.o文件就可以了。

缺点：对所有的文件都会再处理一次，即使b.c没有经过修改，b.c也会重新编译一次， 当文件比较少时，这没有没有什么问题，当文件非常多的时候，就会带来非常多的效率问题。

如果文件非常多的时候，我们，只是修改了一个文件，所用的文件就会重新处理一次，编译的时候就会等待很长时间。

对于这些源文件，我们应该分别处理，执行：预处理 编译 汇编 ，先分别编译它们，最后再把它们链接在一次，比如：

编译：

```
gcc -o a.o a.c
gcc -o b.o b.c
```

链接：

```
gcc -o test a.o b.o
```

比如：上面的例子，当我们修改a.c之后,a.c会重现编译然后再把它们链接在一起就可以了。，b.c 就不需要重新编译。

那么问题又来了，怎么知道哪些文件被更新了/被修改了？

比较时间：比较a.o和a.c的时间，如果a.c的时间比a.o的时间更加新的话，就表明a.c被修改了，同理b.o和b.c也会进行同样的比较。比较test和a.o, b.o的时间，如果a.o或者b.o的时间比test更加新的话，就表明应该重新生成test。Makefile 就是这样做的。

我们现在来写出一个简单的Makefile: makefie最基本的语法是规则，规则：

```
目标： 依赖1  依赖2  ...
[TAB]命令
```

当“依赖”比“目标”新，执行它们下面的命令。我们要把上面三个命令写成makefile规则，如下：

```
test： a.o b.o  //test是目标，它依赖于a.o b.o文件，一旦a.o或者b.o比test新的时候，
```

就需要执行下面的命令，重新生成test可执行程序。

```
gcc -o test a.o b.o
a.o : a.c  //a.o依赖于a.c，当a.c更加新的话，执行下面的命令来生成a.o
gcc -c -o a.o a.c
b.o : b.c  //b.o依赖于b.c,当b.c更加新的话，执行下面的命令，来生成b.o
gcc -c -o b.o b.c
```

我们来作一下实验：

在改目录下我们写一个Makefile文件：

文件：Makefile

```
1  test:a.o b.o
2      gcc -o test a.o b.o
3
4  a.o : a.c
5      gcc -c -o a.o a.c
6
7  b.o : b.c
8      gcc -c -o b.o b.c
```

上面是makefile中的三条规则。makefile,就是名字为“makefile”的文件。当我们想编译程序时，直接执行make命令就可以了，一执行make命令它想生成第一个目标test可执行程序, 如果发现a.o 或者b.o没有，就要先生成a.o或者b.o，发现a.o依赖a.c，有a.c

但是没有a.o,他就会认为a.c比a.o新，就会执行它们下面的命令来生成a.o，同理b.o和b.c的处理关系也是这样的。

如果修改a.c ，我们再次执行make，它的本意是想生成第一个目标test应用程序,

它需要先生成a.o, 发现a.o依赖a.c(执行我们修改了a.c)发现a.c比a.o更加新，就会执行gcc -c -o a.o a.c命令来生成a.o文件。b.o依赖b.c，发现b.c并没有修改，就不会执行gcc -c -o b.o b.c来重新生成b.o文件。现在a.o b.o都有了，其中的a.o比test更加新，就会执行gcc -o test a.o b.o来重新链接得到test可执行程序。所以当执行make命令时候就会执行下面两条执行：

```
gcc -c -o a.o a.c
gcc -o test a.o b.o
```

我们第一次执行make的时候，会执行下面三条命令(三条命令都执行)： gcc -c -o a.o a.c gcc -c -o b.o b.c gcc -o test a.o b.o

再次执行make 就会显示下面的提示：

```
make: `test' is up to date.
```

我们再次执行make 就会判断Makefile文件中的依赖，发现依赖没有更新，所以目标文件就不会重现生成，就会有上面的提示。当我们修改a.c后，重新执行make,

就会执行下面两条指令：

```
gcc -c -o a.o a.c
gcc -o test a.o b.o
```

我们同时修改a.c b.c，执行make就会执行下面三条指令。

```
gcc -c -o a.o a.c
gcc -c -o b.o b.c
gcc -o test a.o b.o
```

a.c文件修改了，重新编译生成a.o, b.c修改了重新编译生成b.o，a.o, b.o都更新了重新链接生成test可执行程序，makefile的规则其实还是比较简单的。

规则是Makefie的核心，执行make命令的时候，就会在当前目录下面找到名字为：Makefile的文件，根据里面的内容来执行里面的判断/命令。

第006节_Makefile的语法

本节我们只是简单的讲解Makefile的语法，如果想比较深入学习Makefile的话可以：

a. 百度搜 "gnu make 于凤昌"。

b. 查看官方文档: http://www.gnu.org/software/make/manual/

通配符

假如一个目标文件所依赖的依赖文件很多，那样岂不是我们要写很多规则，这显然是不合乎常理的。

我们可以使用通配符，来解决这些问题。

我们对上节程序进行修改代码如下：

```
test: a.o b.o
    gcc -o test $^

%.o : %.c
    gcc -c -o $@ $<
```

```
%.o：表示所用的.o文件
%.c：表示所有的.c文件
$@：表示目标
$<：表示第1个依赖文件
$^：表示所有依赖文件
```

我们来在该目录下增加一个c.c文件，代码如下：

```
#include <stdio.h>

void func_c()
{
    printf("This is C\n");
}
```

然后在main函数中调用修改Makefile，修改后的代码如下：

```
test: a.o b.o c.o
	gcc -o test $^

%.o : %.c
	gcc -c -o $@ $<
```

执行：

```
make
```

结果：

```
gcc -c -o a.o a.c
gcc -c -o b.o b.c
gcc -c -o c.o c.c
gcc -o test a.o b.o c.o
```

运行：

```
./test
```

结果：

```
This is B
This is C
```

假想目标: .PHONY

1.我们想清除文件，我们在Makefile的结尾添加如下代码就可以了：

```
clean:
	rm *.o test
```

1）.执行make：生成第一个可执行文件。

2）.执行make clean: 清除所有文件，即执行：rm *.o test。

make后面可以带上目标名，也可以不带，如果不带目标名的话它就想生成第一个规则里面的第一个目标。

2.使用Makefile

执行：make [目标]

也可以不跟目标名，若无目标默认第一个目标。我们直接执行make的时候，会在makefile里面找到第一个目标然后执行下面的指令生成第一个目标。当我们执行make clean的时候，就会在Makefile里面找到clean这个目标，然后执行里面的命令，这个写法有些问题，原因是我们的目录里面没有clean这

个文件，这个规则执行的条件成立，他就会执行下面的命令来删除文件。

如果：该目录下面有名为clean文件怎么办呢?

我们在该目录下创建一个名为“clean”的文件，然后重新执行：make然后make clean，结果(会有下面的提示：)：

```
make: `clean' is up to date.
```

它根本没有执行我们的删除操作，这是为什么呢?

我们之前说，一个规则能过执行的条件：

1).目标文件不存在

2).依赖文件比目标新。

现在我们的目录里面有名为“clean”的文件，目标文件是有的，并且没有依赖文件，没有办法判断依赖文件的时间。这种写法会导致：有同名的"clean"文件时，就没有办法执行make clean操作。解决办法：我们需要把目标定义为假想目标，用关键字PHONY。

```
.PHONY: clean  //把clean定义为假想目标。他就不会判断名为“clean”的文件是否存在，
```

然后在Makfile结尾添加.PHONY: clean语句，重新执行：make clean，就会执行删除操作。

变量

在makefile中有两种变量：

- 1)简单变量(即时变量)：

```
A := xxx  # A的值即刻确定，在定义时即确定
```

对于即时变量使用“:=”表示，它的值在定义的时候已经被确定了

- 2）延时变量

```
B = xxx   # B的值使用到时才确定
```

对于延时变量使用“=”表示。它只有在使用到的时候才确定，在定义/等于时并没有 确定下来。

想使用变量的时候使用“$”来引用，如果不想看到命令是，可以在命令的前面加上"@"符号，就不会显示命令本身。当我们执行make命令的时候，make这个指令本身，会把整个Makefile读进去，进行全部分析，然后解析里面的变量。常用的变量的定义如下：

```
:=  # 即时变量
=   # 延时变量
?=  # 延时变量, 如果是第1次定义才起效, 如果在前面该变量已定义则忽略这句
```

```
+=   # 附加, 它是即时变量还是延时变量取决于前面的定义
?=：如果这个变量在前面已经被定义了，这句话就不会起效果，
```

实例：

```
A := $(C)
B = $(C)
C = abc

#D = 100ask
D ?= weidongshan

all:
    @echo A = $(A)
    @echo B = $(B)
    @echo D = $(D)

C += 123
```

执行：

```
make
```

结果：

```
A =
B = abc 123
D = weidongshan
```

分析：

1) A := $(C)： A为即时变量，在定义时即确定，由于刚开始C的值为空，所以A的值也为空。

2) B = $(C)： B为延时变量，只有使用到时它的值才确定，当执行make时，会解析Makefile里面的所用变量，所以先解析C = abc,然后解析C += 123，此时，C = abc 123，当执行：@echo B = $(B) B的值为 abc 123。

3) D ?= weidongshan： D变量在前面没有定义，所以D的值为weidongshan，如果在前面添加D = 100ask，最后D的值为100ask。

我们还可以通过命令行存入变量的值 例如：

执行：

```
make D=123456
```

里面的D ?= weidongshan这句话就不起作用了。

结果：

```
A =
B = abc 123
D = 123456
```

第007节_Makefile函数

makefile 里面可以包含很多函数，这些函数都是make本身实现的，下面我们来几个常用的函数。

引用一个函数用“$”。

函数foreach

函数foreach语法如下：

```
$(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，对每一个‘list’扩展产生的字，将用来为‘var’扩展后命名的变量赋值；然后‘text’引用该变量扩展；因此它每次扩展都不相同。结果是由空格隔开的‘text’在‘list’中多次扩展的字组成的新的‘list’。‘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。

实例：

```
A = a b c
B = $(foreach f, &(A), $(f).o)

all:
	@echo B = $(B)
```

结果：

```
B = a.o b.o c.o
```

函数filter/filter-out

函数filter/filter-out语法如下：

```
$(filter pattern...,text)      # 在text中取出符合patten格式的值
$(filter-out pattern...,text)  # 在text中取出不符合patten格式的值
```

实例：

```
C = a b c d/

D = $(filter %/, $(C))
E = $(filter-out %/, $(C))

all:
	@echo D = $(D)
	@echo E = $(E)
```

结果：

```
D = d/
E = a b c
```

Wildcard

函数Wildcard语法如下：

```
$(wildcard pattern)   # pattern定义了文件名的格式, wildcard取出其中存在的文件
```

这个函数wildcard会以pattern这个格式，去寻找存在的文件，返回存在文件的名字。

实例：

在该目录下创建三个文件：a.c b.c c.c

```
files = $(wildcard *.c)

all:
	@echo files = $(files)
```

结果：

```
files = a.c b.c c.c
```

我们也可以用wildcard函数来判断，真实存在的文件 实例：

```
files2 = a.c b.c c.c d.c e.c  abc
files3 = $(wildcard $(files2))

all:
	@echo files3 = $(files3)
```

结果：

```
files3 = a.c b.c c.c
```

patsubst函数

函数patsubst语法如下：

```
$(patsubst pattern,replacement,$(var))
```

patsubst函数是从var变量里面取出每一个值，如果这个符合pattern格式，把它替换成replacement格式。

实例：

```
files2  = a.c b.c c.c d.c e.c abc

dep_files = $(patsubst %.c,%.d,$(files2))

all:
	@echo dep_files = $(dep_files)
```

结果：

```
dep_files = a.d b.d c.d d.d e.d abc
```

第008节_Makefile实例

前面讲了那么多Makefile的知识，现在开始做一个实例。

之前编译的程序002_syntax，有个缺陷，将其复制出来，新建一个003_example文件夹，放在里面。在c.c里面，包含一个头文件c.h，在c.h里面定义一个宏，把这个宏打印出来。

c.c:

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

void func_c()
{
   printf("This is C = %d\n", C);
}
```

c.h:

```
#define C 1
```

然后上传编译，执行./test,打印出：

```
This is B
This is C =1
```

测试没有问题，然后修改c.h：

```
#define C 2
```

重新编译，发现没有更新程序，运行，结果不变，说明现在的Makefile存在问题。

为什么会出现这个问题呢， 首先我们test依赖c.o，c.o依赖c.c，如果我们更新c.c，会重新更新整个程序。 但c.o也依赖c.h，我们更新了c.h，并没有在Makefile上体现出来，导致c.h的更新，Makefile无法检测到。 因此需要添加:

```
c.o : c.c c.h
```

现在每次修改c.h，Makefile都能识别到更新操作，从而更新最后输出文件。

这样又冒出了一个新的问题，我们怎么为每个.c文件添加.h文件呢？对于内核，有几万个文件，不可能为每个文件依次写出其头文件。 因此需要做出改进，让其自动生成头文件依赖，可以参考这篇文章：http://blog.csdn.net/qq1452008/article/details/50855810

```
gcc -M c.c // 打印出依赖

gcc -M -MF c.d c.c  // 把依赖写入文件c.d

gcc -c -o c.o c.c -MD -MF c.d  // 编译c.o, 把依赖写入文件c.d
```

修改Makefile如下：

```
objs = a.o b.o c.o

dep_files := $(patsubst %,.%.d, $(objs))
dep_files := $(wildcard $(dep_files))

test: $(objs)
    gcc -o test $^

ifneq ($(dep_files),)
include $(dep_files)
endif

%.o : %.c
    gcc -c -o $@ $< -MD -MF .$@.d

clean:
    rm *.o test

distclean:
    rm $(dep_files)

.PHONY: clean
```

首先用obj变量将.o文件放在一块。 利用前面讲到的函数，把obj里所有文件都变为.%.d格式，并用变量dep_files表示。 利用前面介绍的wildcard函数，判断dep_files是否存在。 然后是目标文件test依赖所有的.o文件。 如果dep_files变量不为空，就将其包含进来。 然后就是所有的.o文件都依赖.c文件，且通过-MD -MF生成.d依赖文件。 清理所有的.o文件和目标文件 清理依赖.d文件。

现在我门修改了任何.h文件，最终都会影响最后生成的文件，也没任何手工添加.h、.c、.o文件，完成了支持头文件依赖。

下面再添加CFLAGS，即编译参数。比如加上编译参数-Werror，把所有的警告当成错误。

```
CFLAGS = -Werror -Iinclude

.....


%.o : %.c
    gcc $(CFLAGS) -c -o $@ $< -MD -MF .$@.d
```

现在重新make，发现以前的警告就变成了错误，必须要解决这些错误编译才能进行。在a.c里面声明一下函数：

```
void func_b();
void func_c();
```

重新make，错误就没有了。

除了编译参数-Werror，还可以加上-I参数，指定头文件路径，-Iinclude表示当前的inclue文件夹下。此时就可以把c.c文件里的#include ".h"改为#include <c.h>，前者表示当前目录，后者表示编译器指定的路径和GCC路径。

《《所有章节目录》》

分类： ARM裸机加强版
