写在前面的话

本节视频来自2018年为2440的裸机加强版视频，非常适合本教程。
要注意几点：

- 以前源码目录是`003_Makefile`，现在目录改为`04_2018_Makefile`
  - GIT仓库里：`01_all_series_quickstart\04_嵌入式Linux应用开发基础知识\source\04_2018_Makefile`
- 本节视频配套的文档，就是本文档，位于：
  - GIT仓库里：`01_all_series_quickstart\04_嵌入式Linux应用开发基础知识\doc_pic\04.2018_Makefile`

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.ob.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。