**make bzImage的过程**

2012年07月11日 19:14:22 [pingrui1234](https://me.csdn.net/pingrui1234) 阅读数：5005 标签： [makefile](http://so.csdn.net/so/search/s.do?q=makefile&t=blog" \t "_blank)[include](http://so.csdn.net/so/search/s.do?q=include&t=blog)[build](http://so.csdn.net/so/search/s.do?q=build&t=blog)[wildcard](http://so.csdn.net/so/search/s.do?q=wildcard&t=blog)[image](http://so.csdn.net/so/search/s.do?q=image&t=blog)[扩展](http://so.csdn.net/so/search/s.do?q=%E6%89%A9%E5%B1%95&t=blog) 更多

个人分类： [linux](https://blog.csdn.net/pingrui1234/article/category/1166850" \t "_blank)

从以上例子中可以看到，内核的编译系统kbuild是个很庞大的系统。但是，它所使用的make和我们平时用的make是一模一样的。kbuild只是通过预定义一些变量（obj-m,obj-y等等）和目标(bzImage ,menuconfig等等)，使内核的编译和扩展变得十分方便。我们不妨yy一下kbuild的一些功能：  
1.考虑到**Linux**能够方便地移植到各个硬件平台，kbuild也必须很容易添加对某个新的平台的支持，同时上层的Makefile不需要做大的改动。  
2.Linux下有众多驱动设备。它们的Makefile希望能够尽可能简洁。简洁到只要指定要编译的.o文件就行。（这方面kbuild定义了很多有用的变量如obj-m obj-y,-objs等等，用户只要为这些变量赋值，kbuild会自动把代码编译到内核或者编译成模块）  
3.要有方便的可定制性。很多参数可以让用户指定。这方面kbuild也提供了大量的变量如EXTRA\_CFLAGS，用户如果想include自己的头文件或者加其它编译参数，只要设置一下EXTRA\_CFLAGS就可以。  
4.有能力递归地调用Makefile。因为内核是一个庞大的软件。它的源代码的目录层次很深。要提供一种简洁的机制，使上层的Makefile能方便地调用下层的Makefile。在这过程中，面向对象的思想也许值得借鉴。  
5.在配置内核时，要提供友好的用户界面。这方面kbuild也提供了不少工具，如常用的make menuconfig等等。  
我们完全可以把kbuild想象成一个类库，它为普通的内核开发人员提供了接口（obj-m obj-y EXTRA\_CFLAGS等等），为用户提供了定制工具（make menuconfig）  
如果想了解kbuild的使用方法，可以参阅源代码自带的文档：  
Documentation/kbuild/makefiles.txt  
Documentation/kbuild/modules.txt  
一般情况下是不需要知道具体的编译顺序的。除了在个别情况下，如do\_initcalls()中就和函数在.initcall.init section中的顺序有关。不过喜欢寻根究底的我，还是想理一下编译内核时几个常用的命令，如make bzImage,make menuconfig等等，进而了解kbuild的架构。先看make bzImage吧。  
**make bzImage**大概脉络是怎样的呢？可以用以下命令查看。  
make -n bzImage   
如果嫌内容太多，可以过滤掉多余的信息：  
make -n bzImage | grep “make -f”  
可以猜到：  
先作一些准备工作  
make -f scripts/Makefile.build obj=scripts/basic  
然后依次递归地调用源代码中的Makefile  
make -f scripts/Makefile.build obj=init  
make -f scripts/Makefile.build obj=usr  
make -f scripts/Makefile.build obj=arch/i386/kernel  
make -f scripts/Makefile.build obj=arch/i386/kernel/acpi  
make -f scripts/Makefile.build obj=arch/i386/kernel/cpu  
make -f scripts/Makefile.build obj=arch/i386/kernel/cpu/cpufreq  
make -f scripts/Makefile.build obj=arch/i386/kernel/cpu/mcheck  
make -f scripts/Makefile.build obj=arch/i386/kernel/cpu/mtrr  
make -f scripts/Makefile.build obj=arch/i386/kernel/timers  
。。。  
最后压缩内核，生成bzImage  
make -f scripts/Makefile.build obj=arch/i386/boot arch/i386/boot/bzImage  
make -f scripts/Makefile.build obj=arch/i386/boot/compressed IMAGE\_OFFSET=0x100000 arch/i386/boot/compressed/vmlinux  
好，我们从头开始。找make bzImage的入口：  
第一反应，自然是在/usr/src/linux/Makefile中找  
bzImage:  
...  
可惜没找到。  
不过没关系，用lxr搜索一下，可知bzImage定义在arch/i386/Makefile，所以可以猜测，该makefile一定是被include了。 果然，在/usr/src/linux/Makefile中有：  
447 include $(srctree)/arch/$(ARCH)/Makefile  
又因为在arch/i386/Makefile中定义有  
141 zImage bzImage: vmlinux  
142         $(Q)$(MAKE) $(build)=$(boot) $(KBUILD\_IMAGE)  
其中这个$(build)定义在/usr/src/linux/Makefile中  
1335 build := -f $(if $(KBUILD\_SRC),$(srctree)/)scripts/Makefile.build obj  
我们在之前查看make -n bzImage信息和之后会经常看到。我们会发现kbuild通常不会直接去调用某个目录下的Makefile，而是让该目录作为scripts/Makefile.build 的参数。scripts/Makefile.build 会对该目录下的Makefile中的内容(主要是obj-m和obj-y等等)进行处理。由此看来 scripts/Makefile.build这个文件很重要。看看它做了什么：  
由于scripts/Makefile.build后面没跟目标，所以默认为第一个目标：  
007 .PHONY: \_\_build  
008 \_\_build:  
009   
010 # Read .config if it exist, otherwise ignore  
011 -include .config  
012   
013 include $(if $(wildcard $(obj)/Kbuild), $(obj)/Kbuild, $(obj)/Makefile)  
014   
015 include scripts/Makefile.lib  
这里可以看到，scripts/Makefile.build执行时会include .config文件。.config是make menuconfig后生成的内核配置文件。  
里面有如下语句：  
CONFIG\_ACPI\_THERMAL=y  
CONFIG\_ACPI\_ASUS=m  
CONFIG\_ACPI\_IBM=m  
。。。  
以前我一直对它的格式表示奇怪，现在很清楚了，它们是作为makefile的一部分，通过读取CONFIG\_XXX的值就可以知道他们是作为模块还是作为内核的一部分而编译的。  
此外，还包含了$(obj)/Makefile。这就是通过在make时传递目录名$(obj)间接调用Makefile的手法。对于内核普通代码所对应的Makefile而言，里面只是对obj-m obj-y,-objs等变量进行赋值操作。  
接下去是include scripts/Makefile.lib  
。正如它的文件名所示，这类似于一个库文件。它负责对obj-m obj-y,-objs等变量进行加工处理。从中提取出subdir-ym等变量，这是个很重要的变量，记录了需要递归调用的子目录。以后递归调用Makefile全靠它了。这里也充分体现了GNU make对字符串进行操作的强大功能。  
回到Makefile.build。这时，重要变量$(builtin-target)，$(subdir-ym)等都已经计算完毕。开始列依赖关系和具体操作了。  
079 \_\_build: $(if $(KBUILD\_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \  
080          $(if $(KBUILD\_MODULES),$(obj-m)) \  
081          $(subdir-ym) $(always)  
082         @:  
$(builtin-target)是指当前目录下的目标文件，即$(obj)/built-in.o  
如前文所说，$(subdir-ym)用来递归调用子目录的Makefile  
306 # Descending  
307 # ---------------------------------------------------------------------------  
308   
309 .PHONY: $(subdir-ym)  
310 $(subdir-ym):  
311         $(Q)$(MAKE) $(build)=$@  
通过这种方式，实现了对某个目录及其子目录的编译。  
分析完Makefile.build，回过头来再看bzImage.从arch/i386/Makefile中可以看到，bzImage是在vmlinux基础上加以压缩拼接而成。从vmlinux到bzImage的过程在《读核感悟-[**Linux内核**](http://www.top-e.org/jiaoshi/class/)启动-内核的生成》中已经有介绍。现在看看vmlinux是如何生成的。见/usr/src/linux/Makefile   
728 vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) FORCE  
729         $(call if\_changed\_rule,vmlinux\_\_)  
611 vmlinux-init := $(head-y) $(init-y)  
612 vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)  
613 vmlinux-all  := $(vmlinux-init) $(vmlinux-main)  
614 vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds  
vmlinux所依赖的目标$(vmlinux-lds) 是对arch/i386/kernel/vmlinux.lds.S进行预处理的结果：arch/i386/kernel/vmlinux.lds ,其它的依赖关系也都可以在/usr/src/linux/Makefile中查到。  
所以，当用户在源代码目录下执行make bzImage。make会检查bzImage的依赖目标，然后不停地递归调用各个Makefile，最终生成一个bzImage文件。  
如果我们换个角度，还可以归纳出不少有趣的东西。如果把make看成是一种脚本语言，那么Makefile就是代码。make就是解释器。make里也有函数，也有变量。通过定义目标，可以实现类似于函数的效果。而目标之间的依赖关系则类似于函数内部再调用其它函数。  
如果我们考虑变量的作用域，还可以归纳出以下几点：  
1.Makefile内部定义的变量作用域只限于那个Makefile中，如obj-m。  
2.要使变量的作用域扩展到整个make命令的执行过程（包括递归调用的其它Makefile），需要使用export命令。  
调用Makefile的方式也有很多种：  
1.一种是隐式调用，如运行make，它会自动在当前目录寻找Makefile等。  
2.一种是显式调用，如用make -f指定。  
3.一种是用include 来调用。