Создание внешних модулей

В этом документе описывается, как создать модули, не входящие в дерево ядра.

Оглавление

1 Введение

«kbuild» является система построения, используемая ядром Linux. Модули должны использовать kbuild для того, что бы оставаться совместимыми с изменениями в инфраструктуре построения и подобрать правильные флаги «gcc.» Функциональность для создания модулей в дереве и вне дерева предоставляется. Методы для их построения аналогичные для всех модулей, изначально разработанных и построенны в и вне дерева.

В настоящем документе предлагается информация для разработчиков, заинтересованных в создании модулей вне дерева (или «внешних»). Автор внешнего модуль должен предоставить файл makefile, который скрывает большую сложность, так что только напечатав «make» можно собрать модуль. Это легко сделать, и в разделе 3 будет представлен полный пример.

2 Как построить внешние модули

Для создания внешних модулей, Вы должны иметь готовые ядра, который содержит конфигурационный файл и файлы заголовков, используемых в построении ядра. Кроме того, ядро должно были построено с поддержкой модулей. Если вы используете дистрибутив ядра, то должен быть пакет для ядра, который Вы запускаете, для получения дистрибутива.

Альтернативой является использование «make» с целью «modules_prepare.» Это позволит убедиться, что ядро содержит необходимую информацию. Наличие целевого объекта обеспечивает простой способ подготовки исходного дерева ядра для создания внешних модулей.

ПРИМЕЧАНИЕ: «modules_prepare» не будет строить Module.symvers, даже если установлена CONFIG_MODVERSIONS; Поэтому необходимо выполнить полную сборку ядра , чтобы получить работающий модуль управления версиями.

2.1 синтаксис команды

Командой для построения внешнего модуль является:

```
$ make -C <path_to_kernel_src> M=$PWD
```

Kbuild система знает, что строится внешний модуль потому, что параметр "M =", переданн в команде.

Для построения для работающего ядра, используйте:

```
$ make -C /lib/modules/`uname -r`/build M=$PWD
```

Затем для установки модуля, который только что построили, просто добавьте цель «modules_install» в команду:

```
$ make -C /lib/modules/`uname -r`/build M=$PWD modules_install
```

2.2 Опции

($KDIR ссылается на путь к каталогу с исходниками ядра.)

```
make -C $KDIR M=$PWD
```

-C $KDIR каталог, в котором находится исходный код ядра. «make» будет фактически изменить указанный каталог при выполнении и изменять обратно после завершения.

M=$PWD информирует Kbuild, что строится внешний модуль Значение, заданное для «M» является абсолютным путём к каталогу, где находится внешний модуль (и его kbuild файл).

2.3 Цели

При создании внешнего модуля, доступны только подмножество целей для «make».

```
make -C $KDIR M=$PWD [target]
```

По умолчанию, модуль будет построен в текущем каталоге, поэтому цель не нужно указать. All выходные файлы будут создаваться также в этом каталоге. Не предпринимаются попытки обновить исходники ядра, и это является условием того, что «make» будет выполнен успешно для ядра.

modules цель по умолчанию для внешних модулей. Это имеет такую же функциональность как если не задавать цели. Смотрите описание выше.

modules_install Установить внешние модули. Расположение по умолчанию — /lib/modules//extra, но префикс может быть задан с помощью INSTALL_MOD_PATH (обсуждается в разделе 5).

clean Удалить все созданные файлы только в каталоге модуля.

help Список доступных целей для внешних модулей.

2.4 Построение отдельных файлов

Можно строить отдельные файлы, которые являются частью модуля. Это работает одинаково хорошо для ядра, модуля и даже для внешних модулей.

Пример (модуль foo.ko, состоит из bar.o и baz.o):

```
make -C $KDIR M=$PWD bar.lst
make -C $KDIR M=$PWD baz.o
make -C $KDIR M=$PWD foo.ko
make -C $KDIR M=$PWD /
```

3. Создание файла Kbuild для внешнего модуля

В последнем разделе мы видели команду, которая строит модуль для работающего ядра. Однако, модуль на самом деле не построен, поскольку требуется файл сборки. В этом файле будет содержатся имя модуля, который строится, а также список необходимых исходных файлов. Файл может быть простой, всего одна строка:

```
obj-m := <module_name>.o
```

Kbuild система будет строить .o из .c и после линковки, получит модуль ядра .ko. Эту строку можно поместить в файл «Kbuild» или «Makefile». Когда модуль строится из нескольких исходных файлов, требуется

дополнительная строка со списком файлов:

```
<module_name>-y := <src1>.o <src2>.o ...
```

ПРИМЕЧАНИЕ: Дополнительная документация, описывающие синтаксис, используемый в kbuild, расположена в Documentation/kbuild/makefiles.txt.

Приведенные ниже примеры демонстрируют, как создать файл сборки для модуля 8123.ko, который построен из следующих файлов:

```
8123_if.c
8123_if.h
8123_pci.c
8123_bin.o_shipped  <= Binary blob
```

3.1 Общий файл Makefile

Внешний модуль всегда включает оболочку для makefile, который поддерживает создание модуля, используя «make» без аргументов. Эта цель не используется kbuild; Это только для удобства. Дополнительные функции, такие как цели тестирования, могут быть включены, но из kbuild должны быть отфильтрованы из-за конфликта имён.

Пример 1:

```
--> filename: Makefile
ifneq ($(KERNELRELEASE),)
    # kbuild part of makefile
    obj-m  := 8123.o
    8123-y := 8123_if.o 8123_pci.o 8123_bin.o
else
    # normal makefile
    KDIR ?= /lib/modules/`uname -r`/build
default:
    $(MAKE) -C $(KDIR) M=$$PWD
    # Module specific targets
    genbin:
    echo "X" > 8123_bin.o_shipped
endif
```

Проверка KERNELRELEASE используется для разделения двух частей файла makefile. В этом примере kbuild будет видеть только два присвоения, в то время как «make» будет видеть все, кроме этих двух присвоений. Это происходит из-за двух проходов по файлу: первый проход делает «make», запущенный из командной строки; а второй проход выполняется kbuild системой, которая инициирует параметризованной «make» в целевой объект по умолчанию.

3.2 Отдельные файлы Kbuild и Makefile

В новых версиях ядра kbuild сначала будет искать файл с именем «Kbuild», и только если, не найдёт, затем будет искать makefile. Использование файла «Kbuild» позволяет нам разделить файл makefile из примера 1 в два файла:

Пример 1:

```
--> filename: Kbuild
obj-m  := 8123.o
8123-y := 8123_if.o 8123_pci.o 8123_bin.o

--> filename: Makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
    $(MAKE) -C $(KDIR) M=$$PWD

# Module specific targets
genbin:
    echo "X" > 8123_bin.o_shipped
```

Разбиение Makefile в примере 2 кажется сомнительным из-за простоты каждого файла; Однако некоторые внешние модули используют Makefile-ы, состоящий из нескольких сотен строк, и здесь действительно стоит отделить часть kbuild от остального.

Следующий пример показывает версию обратной совместимости.

Пример 3:

```
--> filename: Kbuild
obj-m  := 8123.o
8123-y := 8123_if.o 8123_pci.o 8123_bin.o

--> filename: Makefile
ifneq ($(KERNELRELEASE),)
    # kbuild part of makefile
    include Kbuild
else
    # normal makefile
    KDIR ?= /lib/modules/`uname -r`/build
default:
    $(MAKE) -C $(KDIR) M=$$PWD
    # Module specific targets
    genbin:
    echo "X" > 8123_bin.o_shipped
endif
```

Здесь «Kbuild» файл вставляется в makefile. Это позволяет использовать более старые версии kbuild, которые знают только Makefile, когда make

и kbuild части разделены на отдельные файлы.

3.3 Двоичные объекты BLOB

Некоторые внешние модули должны включать объектные файлы, такие как blob. kbuild имеет поддержку для этого, но требует что бы blob-файл будет называться \<filename>_shipped. Когда kbuild обрабатывает правила, копия \<filename>_shipped создаётся без " _shipped", выдавая \<filename>. Это сокращенное имя файла может использоваться в присвоении на модуль.

В этом разделе 8123_bin.o_shipped используется для построения модуля ядра 8123.ko; Файл был включен как 8123_bin.o.

```
8123-y := 8123_if.o 8123_pci.o 8123_bin.o
```

Хотя не существует различия между обычными исходными файлами и двоичными файлами, kbuild будет подбирать различные правила при создании объектного файла модуля.

3.4 Создание нескольких модулей

kbuild поддерживает создание нескольких модулей с помощью одного файла построения. Например если вы хотите построить два модуля, foo.ko и bar.ko, строки для kbuild будут выглядеть так:

```
obj-m := foo.o bar.o
foo-y := <foo_srcs>
bar-y := <bar_srcs>
```

Это просто!

4. Include файлы

В ядре заголовочные файлы хранятся в стандартных местах согласно следующему правилу:

- Если файл заголовка описывает только внутренний интерфейс модуля, файл помещается в том же каталоге, что и исходные файлы.
- Если файл заголовка описывает интерфейс, используемый другими частями ядра, которые находятся в разных каталогах, то файл помещается в include/linux /.

ПРИМЕЧАНИЕ: Есть еще два заметных исключения из этого правила: более крупные подсистемы имеют свой собственный каталог include/, например, include/scsi; и заголовочные файла специальные для конкретной архитектуры расположены в arch / $(ARCH) /include/.

4.1 Include файлы ядра

Чтобы включить файл заголовка, расположенный под include/linux /,

просто используйте:

```
#include <linux/module.h>
```

kbuild добавит опции для «gcc», для поиска соответствующих каталогов.

4.2 Один подкаталог

Внешние модули, как правило, размещают файлы заголовков в отдельной директории include/, где находится их исходный текст, хотя это не обычный стиль ядра. Чтобы сообщить kbuild о каталоге, используйте ccflags-y или CFLAGS_\<filename>. o.

Используя пример из раздела 3, если мы переместим файл 8123_if.h в подкаталог include, то в результате, kbuild файл будет выглядеть так:

--> filename: Kbuild obj-m := 8123.o

ccflags-y := -Iinclude 8123-y := 8123_if.o 8123_pci.o 8123_bin.o

Обратите внимание, что в присвоении нет пробела между - I и путём. Это ограничение kbuild: не должно быть пробелов.

4.3 Несколько подкаталогов

kbuild может обрабатывать файлы, которые распределены по нескольким каталогам. Рассмотрим следующий пример:

```
.
|__ src
|    |__ complex_main.c
|    |__ hal
|    |__ hardwareif.c
|    |__ include
|        |__ hardwareif.h
|__ include
|__ complex.h
```

Чтобы построить модуль complex.ko, нам нужен следующий файл kbuild:

```
--> filename: Kbuild
obj-m := complex.o
complex-y := src/complex_main.o
complex-y += src/hal/hardwareif.o

ccflags-y := -I$(src)/include
ccflags-y += -I$(src)/src/hal/include
```

Как вы можете видеть, kbuild знает, как обрабатывать объектные файлы, расположенных в других каталогах. Хитрость заключается в

том, чтобы указать каталог относительно местоположения файла kbuild. Это, как говорится, не рекомендуемая практика.

Для файлов заголовков, необходимо явным образом объявлять kbuild, где их искать. При выполнении kbuild текущий каталог всегда является корнем дерева ядра (аргумент «-C»), и поэтому требуется абсолютный путь. $(src) предоставляет абсолютный путь, указывая каталог, где находится текущий выполняемый файл kbuild.

5. Установка модуля

Модули, которые включены в ядро установлены в каталоге:

```
/lib/modules/$(KERNELRELEASE)/kernel/
```

И внешние модули установлены в:

```
/lib/modules/$(KERNELRELEASE)/extra/
```

5.1 INSTALL_MOD_PATH

Выше указывались каталоги по умолчанию, но как всегда, некоторый уровень настройки возможен. Префикс может быть добавлен к пути установки, используя переменную INSTALL_MOD_PATH:

```
$ make INSTALL_MOD_PATH=/frodo modules_install
=> Install dir: /frodo/lib/modules/$(KERNELRELEASE)/kernel/
```

INSTALL_MOD_PATH может быть установлены в качестве обычной shell-переменной или, как показано выше, можно указать в командной строке при вызове «make». Это имеет эффект при установке как в дереве, так и вне дерева модулей.

5.2 INSTALL_MOD_DIR

Внешние модули по умолчанию установливаются в каталог /lib/modules/ $ (KERNELRELEASE)/extra/, но вы можете найти модули для выполнения определенных функций в отдельном каталоге. Для этого используйте INSTALL_MOD_DIR, чтобы указать альтернативное имя «extra».

```
$ make INSTALL_MOD_DIR=gandalf -C $KDIR \
M=$PWD modules_install
=> Install dir: /lib/modules/$(KERNELRELEASE)/gandalf/
```

6. Версии модуля

Модуль управления версиями включается тегом CONFIG_MODVERSIONS и используется в качестве простой проверки согласованности ABI. Создаются значения CRC полного прототипа для каждого

экспортируемого символа. Когда модуль загружается/используется, значения CRC, содержащиеся в ядре сравниваются с аналогичными значениями в модуле; Если они не равны, ядро отказывается от загрузки модуля.

Module.symvers содержит список всех экспортируемых из ядра символов.

6.1 Символы из ядра (vmlinux + модули)

Во время построения ядра будет создан файл с именем Module.symvers. Module.symvers содержит все экспортируемые символы из ядра и скомпилированных модулей. Для каждого символа также сохраняется соответствующее значение CRC .

Синтаксис файла Module.symvers:

```
<CRC>       <Symbol>            <module>
0x2d036834  scsi_remove_host    drivers/scsi/scsi_mod
```

Для построения ядра без включения CONFIG_MODVERSIONS, CRC будет считываться как 0x00000000.

Module.symvers служит двум целям:

1. В нем перечислены все экспортируемые символы из vmlinux и всех модулей.
2. В нем перечислены CRC, если включен CONFIG_MODVERSIONS.

6.2 Символы и внешние модули

При построении внешнего модуля, системе построения требуется доступ к символам из ядра, чтобы проверить, все ли внешние символы определены. Это делается в шаге MODPOST. modpost получает символы читая Module.symvers из исходного дерева ядра. Если в каталоге, где строится внешний модуль есть файл Module.symvers, этот файл будет прочитан тоже. На этапе MODPOST будет записан новый файл Module.symvers, содержащий все экспортированные символы, которые не были определены в ядре.

6.3 Символы из другого внешнего модуля

Иногда внешний модуль использует экспортируемые символы из другого внешнего модуля. kbuild должен иметь полное знание всех символов, чтобы избежать появления предупреждений о неопределенных символах. Существуют три решения для этой проблемы.

ПРИМЕЧАНИЕ: Метод с помощью файла kbuild верхнего уровня рекомендуется, но может быть непрактичным в определенных ситуациях.

Используйте файл верхнего уровня kbuild

Если у вас есть два модуля, foo.ko и bar.ko, где foo.ko нуждается в символах из bar.ko, можно использовать общий файл верхнего уровня kbuild поэтому оба модуля будут компилироваться в одном построении. Рассмотрим каталог следующего вида:

```
./foo/ <= contains foo.ko
./bar/ <= contains bar.ko
```

Файл верхнего уровня kbuild будет выглядеть так:

```
#./Kbuild (or ./Makefile):
obj-y := foo/ bar/
```

И выполнение

```
$ make -C $KDIR M=$PWD
```

будут ожидаемым и компилировать оба модуля с полным знанием символов из любого модуля.

Используйте дополнительный файл Module.symvers

Когда строится внешний модуль, создаётся файл Module.symvers, содержащий все экспортированные символы, которые не определены в ядре. Чтобы получить доступ к символам из bar.ko, скопируйте файл Module.symvers из компиляции bar.ko каталог, в котором находится foo.ko. Во время построения модуля kbuild будет читать Module.symvers файл в каталоге внешнего модуля, и по завершении построения, будет создан новый файл Module.symvers содержащий сумму всех символов, определенных и не являющихся частью ядра.

Используйте переменную «make» KBUILD_EXTRA_SYMBOLS

Если непрактично копировать файл Module.symvers из другого модуля, можно присвоить KBUILD_EXTRA_SYMBOLS список файлов, разделённых пробелами, файле сборки. Эти файлы будут загружены modpost во время инициализации таблицы символов.

7. Тонкости и подсказки

7.1 Тестирование CONFIG_FOO_BAR

Модулям часто требуется проверить некоторые опции CONFIG_, чтобы определить, включена ли конкретная функциональная возможность в модуль. В kbuild это делается путем ссылки на переменную CONFIG_ непосредственно.

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS) += ext2.o

ext2-y := balloc.o bitmap.o dir.o
```

```
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

Внешние модули традиционно использовали «grep» для проверки определенных параметров CONFIG_ непосредственно в конфигурации. Это не работает. Как раньше, внешние модули следует использовать kbuild для построения и таким образом можно использовать такие же методы как и внутри дерева модулей при тестировании CONFIG_ определений.