你知道 Linux 内核是如何构建的吗？

2015-9-11 09:26 收藏: 13 分享: 1

原文：https://github.com/0xAX/linux-insides/blob/master/Misc/how_kernel_compiled.md 作者： 0xAX
译文：LCTT https://linux.cn/article-6197-1.html 译者： oska874

介绍

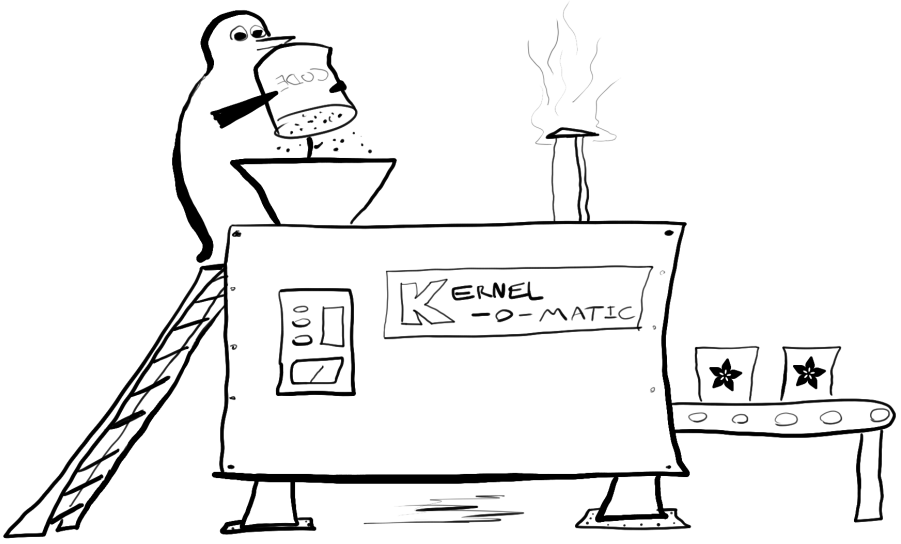
我不会告诉你怎么在自己的电脑上去构建、安装一个定制化的 Linux 内核，这样的资料太多了，它们会对你有帮助。本文会告诉你当你在内核源码路径里敲下 `make` 时会发生什么。

当我刚刚开始学习内核代码时，Makefile 是我打开的第一个文件，这个文件看起来真令人害怕 :)。那时候这个 Makefile 还只包含了 `1591` 行代码，当我开始写本文时，内核已经是4.2.0的第三个候选版本 了。

这个 makefile 是 Linux 内核代码的根 makefile，内核构建就始于此处。是的，它的内容很多，但是如果你已经读过内核源代码，你就会发现每个包含代码的目录都有一个自己的 makefile。当然了，我们不会去描述每个代码文件是怎么编译链接的，所以我们将只会挑选一些通用的例子来说明问题。而你不会在这里找到构建内核的文档、如何整洁内核代码、tags 的生成和交叉编译 相关的说明，等等。我们将从 `make` 开始，使用标准的内核配置文件，到生成了内核镜像 bzImage 结束。

如果你已经很了解 make 工具那是最好，但是我也会描述本文出现的相关代码。

让我们开始吧！



（题图来自：adafruit.com）

编译内核前的准备

在开始编译前要进行很多准备工作。最主要的就是找到并配置好配置文件，`make` 命令要使用到的参数都需要从这些配置文件获取。现在就让我们深入内核的根 `makefile` 吧

内核的根 `Makefile` 负责构建两个主要的文件：vmlinux （内核镜像可执行文件）和模块文件。内核的 Makefile 从定义如下变量开始：

```
VERSION = 4
PATCHLEVEL = 2
SUBLEVEL = 0
EXTRAVERSION = -rc3
NAME = Hurr durr I'm a sheep
```

这些变量决定了当前内核的版本，并且被使用在很多不同的地方，比如同一个 `Makefile` 中的 `KERNELVERSION` ：

```
KERNELVERSION = $(VERSION)$(if $(PATCHLEVEL),.$(PATCHLEVEL)$(if
$(SUBLEVEL),.$(SUBLEVEL)))$(EXTRAVERSION)
```

接下来我们会看到很多 `ifeq` 条件判断语句，它们负责检查传递给 `make` 的参数。内核的 `Makefile` 提供了一个特殊的编译选项 `make help` ，这个选项可以生成所有的可用目标和一些能传给 `make` 的有效的命令行参数。举个例子， `make V=1` 会在构建过程中输出详细的编译信息，第一个 `ifeq` 就是检查传递给 make 的 `V=n` 选项。

```
ifeq ("$(origin V)", "command line")
  KBUILD_VERBOSE = $(V)
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif

ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet = quiet_
  Q = @
endif

export quiet Q KBUILD_VERBOSE
```

如果 `V=n` 这个选项传给了 `make` ，系统就会给变量 `KBUILD_VERBOSE` 选项附上 `V` 的值，否则的话 `KBUILD_VERBOSE` 就会为 `0` 。然后系统会检查 `KBUILD_VERBOSE` 的值，以此来决定 `quiet` 和 `Q` 的值。符号 `@` 控制命令的输出，如果它被放在一个命令之前，这条命令的输出将会是 `CC scripts/mod/empty.o` ，而不是 `Compiling ... scripts/mod/empty.o` （LCTT 译注：CC 在 makefile 中一般都是编译命令）。在这段最后，系统导出了所有的变量。

下一个 `ifeq` 语句检查的是传递给 `make` 的选项 `O=/dir` ，这个选项允许在指定的目录 `dir` 输出所有的结果文件：

```
ifeq ($(KBUILD_SRC),)

ifeq ("$(origin O)", "command line")
  KBUILD_OUTPUT := $(O)
endif

ifneq ($(KBUILD_OUTPUT),)
saved-output := $(KBUILD_OUTPUT)
KBUILD_OUTPUT := $(shell mkdir -p $(KBUILD_OUTPUT) && cd $(KBUILD_OUTPUT) \
                                && /bin/pwd)
$(if $(KBUILD_OUTPUT),, \
     $(error failed to create output directory "$(saved-output)"))

sub-make: FORCE
    $(Q)$(MAKE) -C $(KBUILD_OUTPUT) KBUILD_SRC=$(CURDIR) \
     -f $(CURDIR)/Makefile $(filter-out _all sub-make,$(MAKECMDGOALS))

skip-makefile := 1
endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)
```

系统会检查变量 `KBUILD_SRC`，它代表内核代码的顶层目录，如果它是空的（第一次执行 makefile 时总是空的），我们会设置变量 `KBUILD_OUTPUT` 为传递给选项 `O` 的值（如果这个选项被传进来了）。下一步会检查变量 `KBUILD_OUTPUT`，如果已经设置好，那么接下来会做以下几件事：

- 将变量 `KBUILD_OUTPUT` 的值保存到临时变量 `saved-output`；
- 尝试创建给定的输出目录；
- 检查创建的输出目录，如果失败了就打印错误；
- 如果成功创建了输出目录，那么就在新目录重新执行 `make` 命令（参见选项 `-C`）。

下一个 `ifeq` 语句会检查传递给 make 的选项 `C` 和 `M`：

```
ifeq ("$(origin C)", "command line")
  KBUILD_CHECKSRC = $(C)
endif
ifndef KBUILD_CHECKSRC
  KBUILD_CHECKSRC = 0
endif

ifeq ("$(origin M)", "command line")
  KBUILD_EXTMOD := $(M)
endif
```

第一个选项 `C` 会告诉 `makefile` 需要使用环境变量 `$CHECK` 提供的工具来检查全部 `c` 代码，默认情况下会使用sparse。第二个选项 `M` 会用来编译外部模块（本文不做讨论）。

系统还会检查变量 `KBUILD_SRC`，如果 `KBUILD_SRC` 没有被设置，系统会设置变量 `srctree` 为 `.`：

```
ifeq ($(KBUILD_SRC),)
        srctree := .
endif

objtree := .
src     := $(srctree)
obj     := $(objtree)

export srctree objtree VPATH
```

这将会告诉 `Makefile` 内核的源码树就在执行 `make` 命令的目录，然后要设置 `objtree` 和其他变量为这个目录，并且将这些变量导出。接着就是要获取 `SUBARCH` 的值，这个变量代表了当前的系统架构（LCTT 译注：一般都指CPU 架构）：

```
SUBARCH := $(shell uname -m | sed -e s/i.86/x86/ -e s/x86_64/x86/ \
                  -e s/sun4u/sparc64/ \
                  -e s/arm.*/arm/ -e s/sa110/arm/ \
                  -e s/s390x/s390/ -e s/parisc64/parisc/ \
                  -e s/ppc.*/powerpc/ -e s/mips.*/mips/ \
                  -e s/sh[234].*/sh/ -e s/aarch64.*/arm64/)
```

如你所见，系统执行 uname 得到机器、操作系统和架构的信息。因为我们得到的是 `uname` 的输出，所以我们需要做一些处理再赋给变量 `SUBARCH`。获得 `SUBARCH` 之后就要设置 `SRCARCH` 和 `hfr-arch`，`SRCARCH` 提供了硬件架构相关代码的目录，`hfr-arch` 提供了相关头文件的目录：

```
ifeq ($(ARCH),i386)
        SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
        SRCARCH := x86
endif

hdr-arch  := $(SRCARCH)
```

注意：`ARCH` 是 `SUBARCH` 的别名。如果没有设置过代表内核配置文件路径的变量 `KCONFIG_CONFIG`，下一步系统会设置它，默认情况下就是 `.config`：

```
KCONFIG_CONFIG  ?= config
export KCONFIG_CONFIG
```

以及编译内核过程中要用到的 shell

```
CONFIG_SHELL := $ shell if [ x "$$BASH" ]; then echo $$BASH; \
      else if [ x  bin bash ]; then echo  bin bash; \
      else echo sh; fi; fi
```

接下来就要设置一组和编译内核的编译器相关的变量。我们会设置主机的 `C` 和 `C++` 的编译器及相关配置项：

```
HOSTCC       = gcc
HOSTCXX      = g++
HOSTCFLAGS   = -Wall -Wmissing-prototypes -Wstrict-prototypes -O2 -fomit-frame-pointer -std=gnu89
HOSTCXXFLAGS = -O2
```

接下来会去适配代表编译器的变量 `CC`，那为什么还要 `HOST*` 这些变量呢？这是因为 `CC` 是编译内核过程中要使用的目标架构的编译器，但是 `HOSTCC` 是要被用来编译一组 `host` 程序的（下面我们就会看到）。

然后我们就看到变量 `KBUILD_MODULES` 和 `KBUILD_BUILTIN` 的定义，这两个变量决定了我们要编译什么东西（内核、模块或者两者）：

```
KBUILD_MODULES :=
KBUILD_BUILTIN := 1

ifeq ($(MAKECMDGOALS),modules)
  KBUILD_BUILTIN := $(if $(CONFIG_MODVERSIONS),1)
endif
```

在这我们可以看到这些变量的定义，并且，如果们仅仅传递了 `modules` 给 `make`，变量 `KBUILD_BUILTIN` 会依赖于内核配置选项 `CONFIG_MODVERSIONS`。

下一步操作是引入下面的文件：

```
include scripts/Kbuild.include
```

文件 Kbuild 或者又叫做 `Kernel Build System` 是一个用来管理构建内核及其模块的特殊框架。`kbuild` 文件的语法与 makefile 一样。文件scripts/Kbuild.include 为 `kbuild` 系统提供了一些常规的定义。因为我们包含了这个 `kbuild` 文件，我们可以看到和不同工具关联的这些变量的定义，这些工具会在内核和模块编译过程中被使用（比如链接器、编译器、来自 binutils 的二进制工具包，等等）：

```
AS      = $(CROSS_COMPILE)as
LD      = $(CROSS_COMPILE)ld
CC      = $(CROSS_COMPILE)gcc
CPP     = $(CC) -E
AR      = $(CROSS_COMPILE)ar
NM      = $(CROSS_COMPILE)nm
STRIP       = $(CROSS_COMPILE)strip
OBJCOPY     = $(CROSS_COMPILE)objcopy
OBJDUMP     = $(CROSS_COMPILE)objdump
AWK     = awk
...
...
...
```

在这些定义好的变量后面，我们又定义了两个变量：`USERINCLUDE` 和 `LINUXINCLUDE`。他们包含了头文件的路径（第一个是给用户用的，第二个是给内核用的）：

```
USERINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include/uapi \
        -Iarch/$(hdr-arch)/include/generated/uapi \
        -I$(srctree)/include/uapi \
        -Iinclude/generated/uapi \
        -include $(srctree)/include/linux/kconfig.h

LINUXINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include \
        ...
```

以及给 C 编译器的标准标志：

```
KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
           -fno-strict-aliasing -fno-common \
           -Werror-implicit-function-declaration \
           -Wno-format-security \
           -std=gnu89
```

这并不是最终确定的编译器标志，它们还可以在其他 makefile 里面更新（比如 `arch/` 里面的 kbuild）。变量定义完之后，全部会被导出供其他 makefile 使用。

下面的两个变量 `RCS_FIND_IGNORE` 和 `RCS_TAR_IGNORE` 包含了被版本控制系统忽略的文件：

```
export RCS_FIND_IGNORE := \( -name SCCS -o -name BitKeeper -o -name .svn -o    \
              -name CVS -o -name .pc -o -name .hg -o -name .git \) \
              -prune -o
export RCS_TAR_IGNORE := --exclude SCCS --exclude BitKeeper --exclude .svn \
              --exclude CVS --exclude .pc --exclude .hg --exclude .git
```

这就是全部了，我们已经完成了所有的准备工作，下一个点就是如果构建 `vmlinux` 。

直面内核构建

现在我们已经完成了所有的准备工作，根 makefile（注：内核根目录下的 makefile）的下一步工作就是和编译内核相关的了。在这之前，我们不会在终端看到 `make` 命令输出的任何东西。但是现在编译的第一步开始了，这里我们需要从内核根 makefile 的 598 行开始，这里可以看到目标 `vmlinux` ：

```
all: vmlinux
    include arch/$(SRCARCH)/Makefile
```

不要操心我们略过的从 `export RCS_FIND_IGNORE.....` 到 `all: vmlinux.....` 这一部分 makefile 代码，他们只是负责根据各种配置文件（ `make *.config` ）生成不同目标内核的，因为之前我就说了这一部分我们只讨论构建内核的通用途径。

目标 `all:` 是在命令行如果不指定具体目标时默认使用的目标。你可以看到这里包含了架构相关的 makefile（在这里就指的是 arch/x86/Makefile）。从这一时刻起，我们会从这个 makefile 继续进行下去。如我们所见，目标 `all` 依赖于根 makefile 后面声明的 `vmlinux` ：

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
```

`vmlinux` 是 linux 内核的静态链接可执行文件格式。脚本 scripts/link-vmlinux.sh 把不同的编译好的子模块链接到一起形成了 vmlinux。

第二个目标是 `vmlinux-deps` ，它的定义如下：

```
vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)
```

它是由内核代码下的每个顶级目录的 `built-in.o` 组成的。之后我们还会检查内核所有的目录， `kbuild` 会编译各个目录下所有的对应 `$(obj-y)` 的源文件。接着调用 `$(LD) -r` 把这些文件合并到一个 `build-in.o` 文件里。此时我们还没有 `vmlinux-`

deps ，所以目标 vmlinux 现在还不会被构建。对我而言 vmlinux-deps 包含下面的文件：

```
arch x86 kernel vmlinux lds arch x86 kernel head_64 o
arch x86 kernel head64 o    arch x86 kernel head o
init built in o             usr built in o
arch x86 built in o         kernel built in o
mm built in o               fs built in o
ipc built in o              security built in o
crypto built in o           block built in o
lib lib a                   arch x86 lib lib a
lib built in o              arch x86 lib built in o
drivers built in o          sound built in o
firmware built in o         arch x86 pci built in o
arch x86 power built in o   arch x86 video built in o
net built in o
```

下一个可以被执行的目标如下：

```
$ sort $ vmlinux deps : $ vmlinux dirs ;
$ vmlinux dirs : prepare scripts
    $ Q $ MAKE -f $ build -f $@
```

就像我们看到的， vmlinux-dir 依赖于两部分： prepare 和 scripts 。第一个 prepare 定义在内核的根 makefile 中，准备工作分成三个阶段：

```
prepare: prepare0
prepare0: archprepare FORCE
    $ Q $ MAKE -f $ build/Makefile
archprepare: archheaders archscripts prepare1 scripts_basic

prepare1: prepare2 $ version_h $ include/generated/utsrelease h \
                   include/config/auto conf
    $ cmd_crmodverdir
prepare2: prepare3 outputmakefile asm generic
```

第一个 prepare0 展开到 archprepare ，后者又展开到 archheader 和 archscripts ，这两个变量定义在 x86_64 相关的 Makefile。让我们看看这个文件。 x86_64 特定的 makefile 从变量定义开始，这些变量都是和特定架构的配置文件(defconfig，等等)有关联。在定义了编译 16-bit 代码的编译选项之后，根据变量 BITS 的值，如果是 32 ， 汇编代码、链接器、以及其它很多东西（全部的定义都可以在arch/x86/Makefile找到）对应的参数就是 i386 ，而 64 就对应的是 x86_84 。

第一个目标是 makefile 生成的系统调用列表（syscall table）中的 archheaders ：

```
archheaders:
    $ Q $ MAKE -f $ build/Makefile arch x86 entry syscalls all
```

第二个目标是 makefile 里的 archscripts ：

```
archscripts: scripts_basic
    $ Q $ MAKE -f $ build/Makefile arch x86 tools relocs
```

我们可以看到 archscripts 是依赖于根 Makefile里的 scripts_basic 。首先我们可以看出 scripts_basic 是按照 scripts/basic 的 makefile 执行 make 的：

```
scripts_basic:
    $ Q $ MAKE -f $ build/Makefile scripts basic
```

scripts/basic/Makefile 包含了编译两个主机程序 fixdep 和 bin2 的目标：

```
hostprogs-y := fixdep
hostprogs-$(CONFIG_BUILD_BIN2C) := bin2c
always := $(hostprogs-y)

$(addprefix $(obj)/, $(filter-out fixdep, $(always))): $(obj)/fixdep
```

第一个工具是 `fixdep`：用来优化 gcc 生成的依赖列表，然后在重新编译源文件的时候告诉make。第二个工具是 `bin2c`，它依赖于内核配置选项 `CONFIG_BUILD_BIN2C`，并且它是一个用来将标准输入接口（LCTT 译注：即 stdin）收到的二进制流通过标准输出接口（即：stdout）转换成 C 头文件的非常小的 C 程序。你可能注意到这里有些奇怪的标志，如 `hostprogs-y` 等。这个标志用于所有的 `kbuild` 文件，更多的信息你可以从documentation 获得。在我们这里， `hostprogs-y` 告诉 `kbuild` 这里有个名为 `fixed` 的程序，这个程序会通过和 `Makefile` 相同目录的 `fixdep.c` 编译而来。

执行 make 之后，终端的第一个输出就是 `kbuild` 的结果：

```
$ make
  HOSTCC  scripts/basic/fixdep
```

当目标 `script_basic` 被执行，目标 `archscripts` 就会 make arch/x86/tools 下的 makefile 和目标 `relocs`：

```
$(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

包含了重定位 的信息的代码 `relocs_32.c` 和 `relocs_64.c` 将会被编译，这可以在 `make` 的输出中看到：

```
  HOSTCC  arch/x86/tools/relocs_32.o
  HOSTCC  arch/x86/tools/relocs_64.o
  HOSTCC  arch/x86/tools/relocs_common.o
  HOSTLD  arch/x86/tools/relocs
```

在编译完 `relocs.c` 之后会检查 `version.h`：

```
$(version_h): $(srctree)/Makefile FORCE
    $(call filechk,version.h)
    $(Q)rm -f $(old_version_h)
```

我们可以在输出看到它：

```
CHK     include/config/kernel.release
```

以及在内核的根 Makefiel 使用 `arch/x86/include/generated/asm` 的目标 `asm-generic` 来构建 `generic` 汇编头文件。在目标 `asm-generic` 之后， `archprepare` 就完成了，所以目标 `prepare0` 会接着被执行，如我上面所写：

```
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
```

注意 `build`，它是定义在文件 scripts/Kbuild.include，内容是这样的：

```
build := -f $(srctree)/scripts/Makefile.build obj
```

或者在我们的例子中，它就是当前源码目录路径： `.`：

```
$(Q)$(MAKE) -f $(srctree)/scripts/Makefile.build obj=.
```

脚本 scripts/Makefile.build 通过参数 `obj` 给定的目录找到 `Kbuild` 文件，然后引入 `kbuild` 文件：

```
include $(kbuild-file)
```

并根据这个构建目标。我们这里 `.` 包含了生成 `kernel/bounds.s` 和 `arch/x86/kernel/asm-offsets.s` 的 Kbuild 文件。在此之后，目标 `prepare` 就完成了它的工作。 `vmlinux-dirs` 也依赖于第二个目标 `scripts` ，它会编译接下来的几个程序： `filealias` ， `mk_elfconfig` ， `modpost` 等等。之后， `scripts/host-programs` 就可以开始编译我们的目标 `vmlinux-dirs` 了。

首先，我们先来理解一下 `vmlinux-dirs` 都包含了那些东西。在我们的例子中它包含了下列内核目录的路径：

```
init usr arch/x86 kernel mm fs ipc security crypto block
drivers sound firmware arch/x86/pci arch/x86/power
arch/x86/video net lib arch/x86/lib
```

我们可以在内核的根 Makefile 里找到 `vmlinux-dirs` 的定义：

```
vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
             $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
             $(net-y) $(net-m) $(libs-y) $(libs-m)))

init-y       := init/
drivers-y    := drivers/ sound/ firmware/
net-y        := net/
libs-y       := lib/
...
...
...
```

这里我们借助函数 `patsubst` 和 `filter` 去掉了每个目录路径里的符号 `/` ，并且把结果放到 `vmlinux-dirs` 里。所以我们就有了 `vmlinux-dirs` 里的目录列表，以及下面的代码：

```
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

符号 `$@` 在这里代表了 `vmlinux-dirs` ，这就表明程序会递归遍历从 `vmlinux-dirs` 以及它内部的全部目录（依赖于配置），并且在对应的目录下执行 `make` 命令。我们可以在输出看到结果：

```
  CC      init/main.o
  CHK     include/generated/compile.h
  CC      init/version.o
  CC      init/do_mounts.o
  ...
  CC      arch/x86/crypto/glue_helper.o
  AS      arch/x86/crypto/aes-x86_64-asm_64.o
  CC      arch/x86/crypto/aes_glue.o
  ...
  AS      arch/x86/entry/entry_64.o
  AS      arch/x86/entry/thunk_64.o
  CC      arch/x86/entry/syscall_64.o
```

每个目录下的源代码将会被编译并且链接到 `built-in.o` 里：

```
$ find . -name built-in.o
./arch/x86/crypto/built-in.o
./arch/x86/crypto/sha-mb/built-in.o
./arch/x86/net/built-in.o
./init/built-in.o
./usr/built-in.o
```

```
8.  ...
```

好了，所有的 `built-in.o` 都构建完了，现在我们回到目标 `vmlinux` 上。你应该还记得，目标 `vmlinux` 是在内核的根 makefile 里。在链接 `vmlinux` 之前，系统会构建 samples, Documentation 等等，但是如上文所述，我不会在本文描述这些。

```
1.  vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
2.      +$(call if_changed,link-vmlinux)
```

你可以看到，调用脚本 scripts/link-vmlinux.sh 的主要目的是把所有的 `built-in.o` 链接成一个静态可执行文件，和生成 System.map。 最后我们来看看下面的输出：

```
1.    LINK    vmlinux
2.    LD      vmlinux.o
3.    MODPOST vmlinux.o
4.    GEN     .version
5.    CHK     include/generated/compile.h
6.    UPD     include/generated/compile.h
7.    CC      init/version.o
8.    LD      init/built-in.o
9.    KSYM    .tmp_kallsyms1.o
10.   KSYM    .tmp_kallsyms2.o
11.   LD      vmlinux
12.   SORTEX  vmlinux
13.   SYSMAP  System.map
```

`vmlinux` 和 `System.map` 生成在内核源码树根目录下。

```
1.  $ ls vmlinux System.map
2.  System.map  vmlinux
```

这就是全部了， `vmlinux` 构建好了，下一步就是创建 bzImage.

制作bzImage

`bzImage` 就是压缩了的 linux 内核镜像。我们可以在构建了 `vmlinux` 之后通过执行 `make bzImage` 获得 `bzImage` 。同时我们可以仅仅执行 `make` 而不带任何参数也可以生成 `bzImage` ，因为它是在 arch/x86/kernel/Makefile 里预定义的、默认生成的镜像：

```
1.  all: bzImage
```

让我们看看这个目标，它能帮助我们理解这个镜像是怎么构建的。我已经说过了 `bzImage` 是被定义在 arch/x86/kernel/Makefile，定义如下：

```
1.  bzImage: vmlinux
2.      $(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
3.      $(Q)mkdir -p $(objtree)/arch/$(UTS_MACHINE)/boot
4.      $(Q)ln -fsn ../../x86/boot/bzImage $(objtree)/arch/$(UTS_MACHINE)/boot/$@
```

在这里我们可以看到第一次为 boot 目录执行 `make` ，在我们的例子里是这样的：

```
1.  boot := arch/x86/boot
```

现在的主要目标是编译目录 `arch/x86/boot` 和 `arch/x86/boot/compressed` 的代码，构建 `setup.bin` 和 `vmlinux.bin` ，最后用这两个文件生成 `bzImage` 。第一个目标是定义在 arch/x86/boot/Makefile 的 `$(obj)/setup.elf` ：

```
$(obj)/setup.elf: $(src)/setup.ld $(SETUP_OBJS) FORCE
    $(call if_changed,ld)
```

我们已经在目录 `arch/x86/boot` 有了链接脚本 `setup.ld`，和扩展到 `boot` 目录下全部源代码的变量 `SETUP_OBJS` 。我们可以看看第一个输出：

```
  AS      arch/x86/boot/bioscall.o
  CC      arch/x86/boot/cmdline.o
  AS      arch/x86/boot/copy.o
  HOSTCC  arch/x86/boot/mkcpustr
  CPUSTR  arch/x86/boot/cpustr.h
  CC      arch/x86/boot/cpu.o
  CC      arch/x86/boot/cpuflags.o
  CC      arch/x86/boot/cpucheck.o
  CC      arch/x86/boot/early_serial_console.o
  CC      arch/x86/boot/edd.o
```

下一个源码文件是 arch/x86/boot/header.S，但是我们不能现在就编译它，因为这个目标依赖于下面两个头文件：

```
$(obj)/header.o: $(obj)/voffset.h $(obj)/zoffset.h
```

第一个头文件 `voffset.h` 是使用 `sed` 脚本生成的，包含用 `nm` 工具从 `vmlinux` 获取的两个地址：

```
#define VO__end 0xffffffff82a60000
#define VO__text 0xffffffff81000000
```

这两个地址是内核的起始和结束地址。第二个头文件 `zoffset.h` 在 arch/x86/boot/compressed/Makefile 可以看出是依赖于目标 `vmlinux` 的：

```
$(obj)/zoffset.h: $(obj)/compressed/vmlinux FORCE
    $(call if_changed,zoffset)
```

目标 `$(obj)/compressed/vmlinux` 依赖于 `vmlinux-objs-y` —— 说明需要编译目录 arch/x86/boot/compressed 下的源代码，然后生成 `vmlinux.bin` 、 `vmlinux.bin.bz2` ，和编译工具 `mkpiggy` 。我们可以在下面的输出看出来：

```
  LDS     arch/x86/boot/compressed/vmlinux.lds
  AS      arch/x86/boot/compressed/head_64.o
  CC      arch/x86/boot/compressed/misc.o
  CC      arch/x86/boot/compressed/string.o
  CC      arch/x86/boot/compressed/cmdline.o
  OBJCOPY arch/x86/boot/compressed/vmlinux.bin
  BZIP2   arch/x86/boot/compressed/vmlinux.bin.bz2
  HOSTCC  arch/x86/boot/compressed/mkpiggy
```

`vmlinux.bin` 是去掉了调试信息和注释的 `vmlinux` 二进制文件，加上了占用了 `u32` （LCTT 译注：即4-Byte）的长度信息的 `vmlinux.bin.all` 压缩后就是 `vmlinux.bin.bz2` 。其中 `vmlinux.bin.all` 包含了 `vmlinux.bin` 和 `vmlinux.relocs` （LCTT 译注：vmlinux 的重定位信息），其中 `vmlinux.relocs` 是 `vmlinux` 经过程序 `relocs` 处理之后的 `vmlinux` 镜像（见上文所述）。我们现在已经获取到了这些文件，汇编文件 `piggy.S` 将会被 `mkpiggy` 生成、然后编译：

```
  MKPIGGY arch/x86/boot/compressed/piggy.S
  AS      arch/x86/boot/compressed/piggy.o
```

这个汇编文件会包含经过计算得来的、压缩内核的偏移信息。处理完这个汇编文件，我们就可以看到 `zoffset` 生成了：

```
  ZOFFSET arch/x86/boot/zoffset.h
```

现在 `zoffset.h` 和 `voffset.h` 已经生成了，arch/x86/boot 里的源文件可以继续编译：

```
  AS      arch x86 boot header o
  CC      arch x86 boot main o
  CC      arch x86 boot mca o
  CC      arch x86 boot memory o
  CC      arch x86 boot pm o
  AS      arch x86 boot pmjump o
  CC      arch x86 boot printf o
  CC      arch x86 boot regs o
  CC      arch x86 boot string o
  CC      arch x86 boot tty o
  CC      arch x86 boot video o
  CC      arch x86 boot video mode o
  CC      arch x86 boot video vga o
  CC      arch x86 boot video vesa o
  CC      arch x86 boot video bios o
```

所有的源代码会被编译，他们最终会被链接到 `setup.elf` ：

```
  LD      arch x86 boot setup elf
```


或者：

```
ld  m elf_x86_64    T arch x86 boot setup ld arch x86 boot a20 o arch x86 boot bioscall o
arch x86 boot cmdline o arch x86 boot copy o arch x86 boot cpu o arch x86 boot cpuflags o
arch x86 boot cpucheck o arch x86 boot early_serial_console o arch x86 boot edd o
arch x86 boot header o arch x86 boot main o arch x86 boot mca o arch x86 boot memory o
arch x86 boot pm o arch x86 boot pmjump o arch x86 boot printf o arch x86 boot regs o
arch x86 boot string o arch x86 boot tty o arch x86 boot video o arch x86 boot video mode o
arch x86 boot version o arch x86 boot video vga o arch x86 boot video vesa o arch x86 boot video
bios o  o arch x86 boot setup elf
```

最后的两件事是创建包含目录 `arch/x86/boot/*` 下的编译过的代码的 `setup.bin` ：

```
objcopy  O binary arch x86 boot setup elf arch x86 boot setup bin
```

以及从 `vmlinux` 生成 `vmlinux.bin` ：

```
objcopy  O binary  R  note  R  comment  S arch x86 boot compressed vmlinux
arch x86 boot vmlinux bin
```

最最后，我们编译主机程序 arch/x86/boot/tools/build.c，它将会用来把 `setup.bin` 和 `vmlinux.bin` 打包成 `bzImage` ：

```
arch x86 boot tools build arch x86 boot setup bin arch x86 boot vmlinux bin
arch x86 boot zoffset h arch x86 boot bzImage
```

实际上 `bzImage` 就是把 `setup.bin` 和 `vmlinux.bin` 连接到一起。我们会看到输出结果，就和那些用源码编译过内核的同行的结果一样：

```
Setup is 16268 bytes  padded to 16384 bytes
System is 4704 kB
CRC 9468898
Kernel  arch x86 boot bzImage is ready  4
```

全部结束。

相关阅读

Linux 内核

- 你的 Linux 启动时有几只小企鹅？
- Linux 4.0 发布——我是一只羊
- Torvalds: 写linux内核的人更加容易找工
- 如何在Ubuntu/CentOS上安装Linux内核
- Linux Kernel 4.1将是下一个长期支持版
- Linux 内核 4.2 RC 获史上最大更新量

结论

这就是本文的结尾部分。本文我们了解了编译内核的全部步骤：从执行 `make` 命令开始，到最后生成 `bzImage` 。我知道，linux 内核的 makefile 和构建 linux 的过程第一眼看起来可能比较迷惑，但是这并不是很难。希望本文可以帮助你理解构建 linux 内核的整个流程。

链接

- GNU make util
- Linux kernel top Makefile
- cross-compilation
- Ctags
- sparse
- bzImage
- uname
- shell
- Kbuild
- binutils
- gcc
- Documentation
- System.map
- Relocation

---

via: https://github.com/0xAX/linux-insides/blob/master/Misc/how_kernel_compiled.md

译者：oska874 校对：wxy

本文由 LCTT 原创翻译，Linux中国 荣誉推出

原文：https://github.com/0xAX/linux-insides/blob/master/Misc/how_kernel_compiled.md 作者： 0xAX
译文：LCTT https://linux.cn/article-6197-1.html 译者： oska874

上一篇：RHCSA 系列（三）：如何管理 RHEL7 的用户和组
下一篇：Python 开发者的 Docker 之旅