Linux 内核源码分析--zImage 出生实录（Linux-3.0 ARMv7）

此文为两年前为好友刘庆敏的书《嵌入式 Linux 开发详解--基于 AT91RM9200 和 Linux 2.6》中帮忙写的章节的重新整理。如有雷同，纯属必然。经作者同意，将我写的部分重新整理后放入 blog 中。

~~~~~~

自己移植编译过内核的朋友都知道：生成的 zImage 内核的位置在 arch/arm/boot 目录下。但是这个映像是怎么产生的？下面简要地分析一下。

内核根目录下的 vmlinux 映像文件是内核 Makefile 的默认目标。这个 vmlinux 映像的生成可以通过阅读内核 Makefile 文件得知，简单的说：Makefile 解析内核配置文件.config，递归到各目录下编译出.o 文件，最后将其链接成 vmlinux。而这个链接成的 vmlinux 文件**是一个包含内核代码的静态可执行 ELF 文件**，你可以通过 file 命令来验证这一点。**她不能通过 bootloader 引导并启动，如果想要使其可引导，必须使用编译工具链中的 objcopy 命令把这个 ELF 格式的 vmlinux 转化为二进制格式才行。**
而平常使用的 zImage 文件就是这个 vmlinux 文件经过多次的转换得到的。现在就来仔细研究一下她的生成过程。

（1）arch/$(ARCH)/Makefile

首先嵌入式中经常使用的编译目标 zImage 并不在顶层 Makefile 文件中，而在被顶层 Makefile 包含的 arch/$(ARCH)/Makefile 文件中，对于 ARM 处理器来说就是 arch/arm/Makefile 文件。其中的部分规则如下：

```
......
# Default target when executing plain make
ifeq ($(CONFIG_XIP_KERNEL),y)
KBUILD_IMAGE := xipImage
else
KBUILD_IMAGE := zImage
endif

all: $(KBUILD_IMAGE)

boot := arch/arm/boot

archprepare:
$(Q)$(MAKE) $(build)=arch/arm/tools include/generated/mach-types.h

# Convert bzImage to zImage
bzImage: zImage

```
~~~~~~

```
19. zImage Image xipImage bootpImage uImage: vmlinux
20. $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
21. ......
```

从这里可以看出，zImage 的依赖是顶层 vmlinux 文件，下面的命令展开得到：

```
1. make -f scripts/Makefile.build obj= arch/arm/boot MACHINE=arch/arm/mach-* arch/arm
   /boot/ zImage
```

可以看出 zImage 其实是 make 解析 arch/arm/boot 目录下的 Makefile 文件生成的，而参数传递了目标芯片信息和目标“arch/arm/boot/zImage”。所以 zImage 其实是在 arch/arm/boot 目录下完成编译的，这就是为什么可引导 zImage 映像会在 arch/arm/boot 目录下。

（2）arch/$(ARCH)/boot/Makefile

现在来分析一下 arch/arm/boot/Makefile 中的部分规则，看看目标 zImage 的生成：

```
1. $(obj)/Image: vmlinux FORCE
2. $(call if_changed,objcopy)
3. @echo ' Kernel: $@ is ready'
4.
5. $(obj)/compressed/vmlinux: $(obj)/Image FORCE
6. $(Q)$(MAKE) $(build)=$(obj)/compressed $@
7.
8. $(obj)/zImage: $(obj)/compressed/vmlinux FORCE
9. $(call if_changed,objcopy)
10. @echo ' Kernel: $@ is ready'
```

先看最后一行，从中可以得知 arch/arm/boot/zImage 的依赖目标是 arch/arm/boot/compressed/vmlinux，且目标 zImage 是其二进制化的产物。

而 arch/arm/boot/compressed/vmlinux 是如何得到的呢？再看上一规则，arch/arm/boot/compressed/vmlinux 的依赖目标是 arch/arm/boot/Image。这个依赖目标的生成由最上面的规则决定，显然 arch/arm/boot/Image 是由顶层 vmlinux 二进制化得到的。而中间这行规则的含义是 arch/arm/boot/compressed/vmlinux 由 make 解析 arch/arm/boot/compressed/目录下的 Makefile 文件生成的，这条命令展开得到：

```
1. make -f scripts/Makefile.build obj= arch/arm/boot/compressed arch/arm/boot/compres
   sed/vmlinux
```

（3）arch/$(ARCH)/boot/compressed/Makefile

最后就来分析一下 arch/arm/boot/compressed/Makefile 中的部分规则，看看 arch/arm/boot/compressed/vmlinux 的生成：

```
1.
2. ......
3. suffix_$(CONFIG_KERNEL_GZIP) = gzip
4. suffix_$(CONFIG_KERNEL_LZO)  = lzo
```

```
suffix_$(CONFIG_KERNEL_LZMA) = lzma
......
$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/$(HEAD) $(obj)/piggy.$(suffix_y).o \
$(addprefix $(obj)/, $(OBJS)) $(lib1funcs) FORCE
$(call if_changed,ld)
@$(check_for_bad_syms)

$(obj)/piggy.$(suffix_y): $(obj)/../Image FORCE
$(call if_changed,$(suffix_y))

$(obj)/piggy.$(suffix_y).o: $(obj)/piggy.$(suffix_y) FORCE
......
```

上面的第一条规则就说明了：其实 arch/arm/boot/compressed/vmlinux 是由几个部分根据 arch/arm/boot/compressed/vmlinux.lds 脚本链接而成的：

$(obj)/$(HEAD)：arch/arm/boot/compressed/head.o，在链接时处于 vmlinux 的最前面，其主要作用就是做一些必要的初始化工作，如初始化 CPU、中断描述符表 IDT 和内存页目录表 GDT 等等，最后跳到 misc.c 中的 decompress_kernel 函数进行内核的自解压工作。

$(addprefix $(obj)/, $(OBJS))：arch/arm/boot/compressed/ misc.o，位于 head.o 之后，是内核自解压的实现代码。

以下假定是 gzip 模式压缩：

$(obj)/piggy.$(suffix_y).o：arch/arm/boot/compressed/ piggy.gzip.o，其实是 arch/arm/boot/Image 经过 gzip 压缩后（piggy.gzip），再借助 piggy.gzip.S 一起编译出的 ELF 可链接文件。其中的原理可以看看 piggy.gzip.S 源码：

```
.section .piggydata,#alloc
.globl input_data
input_data:
.incbin "arch/arm/boot/compressed/piggy.gzip"
.globl input_data_end
input_data_end:
```

这里我还是要额外的提一下 gzip 压缩，也就是**$(call if_changed,$(suffix_y))**这个过程。这个命令认真解析起来比较麻烦，这里如果有兴趣的读者可以自行分析。这里介绍两篇经典的分析文档：《kbuild 实现分析》、《Kbuild 系统原理分析》，读者可自行上网下载学习。这里我直接给出了结果，这条命令执行了在 Makefile.lib(scripts)中定义的：

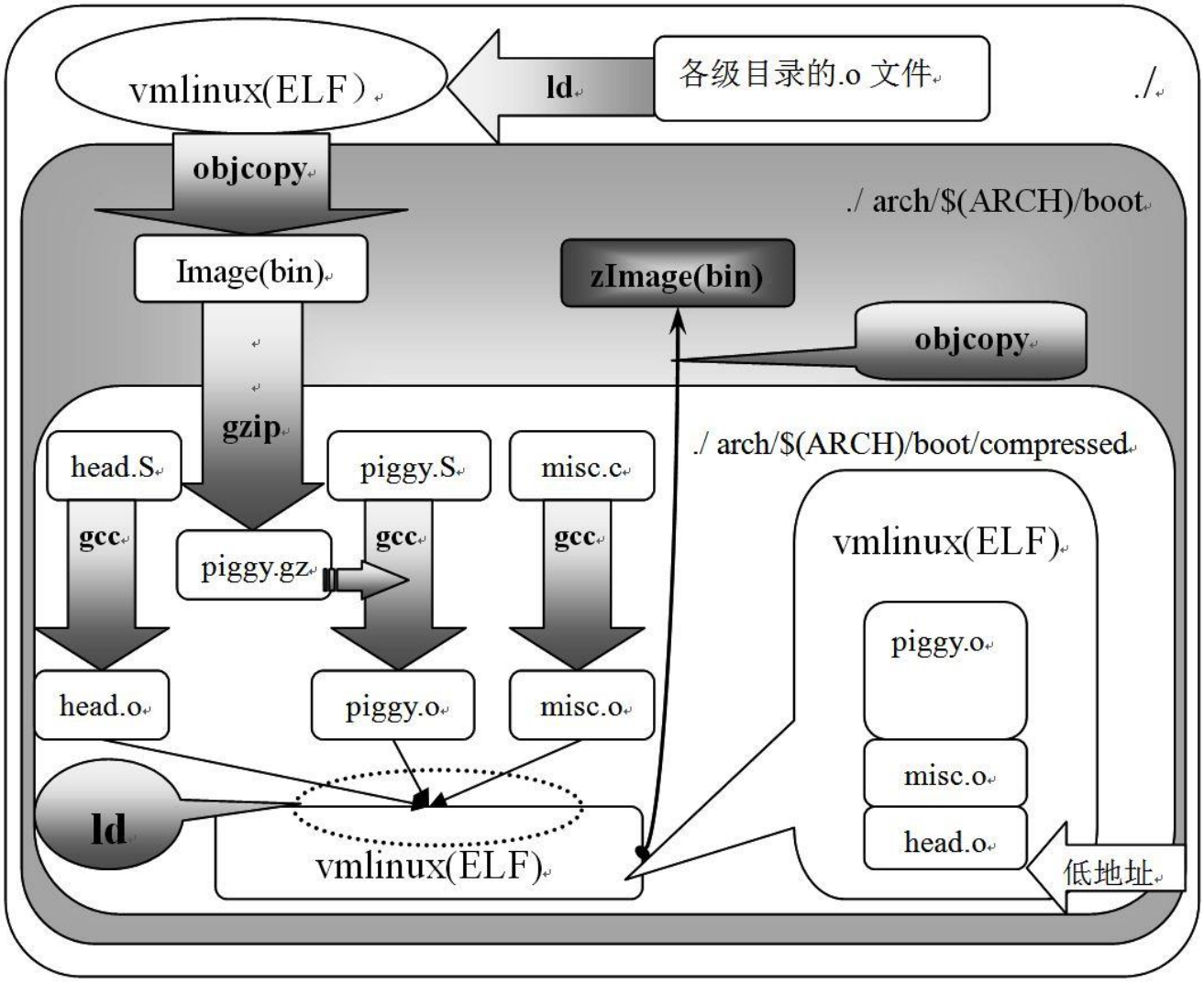
```
cmd_gzip = (cat $(filter-out FORCE,$^) | gzip -n -f -9 > $@) || \
(rm -f $@ ; false)
```

也就是说，piggy.gzip 是将 arch/arm/boot/Image 文件 cat 到标准输出，并通过管道传入 gzip 命令（gzip -n -f -9 ）的标准输入，最后将 gzip 的输出重定向到目标 piggy.gzip

而这个 piggy.gzip 文件有一个重要的特性：最后的四个字节，是文件压缩前的大小数据，存放格式是小端模式。这个数据在 zImage 自解压时会被用于程序得到内核解压后所需要的空间！！！

感兴趣的朋友可以自己随便用“gzip -n -f -9”压缩一个文件试试，验证一下，我已亲自验证过了。

这样跟踪下来，zImage 的产生过程已经看完了，但是读者可能会被这有点复杂的关系绕晕了，所以现在可以结合一下的流程图简单地总结一下：



首先顶层 vmlinux 是 ELF 格式的可执行文件，必须将其二进制化生成 Image 后才可以被 bootloader 引导。为了实现压缩的内核映像，arch/arm/boot/compressed/Makefile 又将这个非压缩映像 Image 做 gzip 压缩，生成了 piggy.gzip。但要实现在启动时自解压，必须将这个 piggy.gzip 转化为.o 文件，并同初始化程序 head.o 和自解压程序 misc.o 一同链接，生成

arch/arm/boot/compressed/vmlinux。最后 arch/arm/boot/Makefile 将这个 ELF 格式的 arch/arm/boot/compressed/vmlinux 二进制化得到可被 bootloader 引导的映像文件 **zImage**。