探索Linux内核：Kconfig/kbuild的秘密

深入了解Linux配置/构建系统的工作原理。

自从Linux内核代码迁移到Git以来，Linux内核配置/构建系统（也称为Kconfig /kbuild）已经存在了很长时间。然而，作为支撑基础设施，它很少成为人们关注的焦点；甚至在日常工作中使用它的内核开发人员也从未真正过它。

为了探索如何编译Linux内核，本文将深入研究Kconfig/kbuild内部过程，解释如何生成.config文件和vmlinux/bzImage文件，并介绍依赖性跟踪的智能技巧。

Kconfig

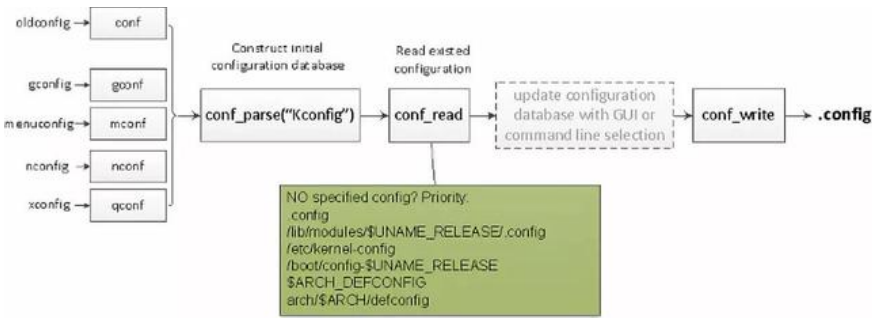
构建内核的第一步始终是Kconfig，Kconfig有助于使Linux内核高度模块化和可定制。 Kconfig为用户提供了许多配置目标：

config使用基于行的程序更新当前配置nconfig使用ncurses基于菜单的程序更新当前配置menuconfig使用基于菜单的程序更新当前配置xconfig使用基于Qt的前端更新当前配置gconfig使用基于GTK +的前端更新当前配置oldconfig使用提供的.config作为基础更新当前配置localmodconfig更新当前配置禁用未加载的模块localyesconfig更新当前配置，将本地mod转换到内核defconfig默认来自Arch提供的defconfig的新配置savedefconfig将当前配置保存为./defconfig（最小配置）allnoconfig新配置，其中所有选项均以“否”回答allyesconfig新配置，其中所有选项都被'yes'接受allmodconfig尽可能新配置选择模块alldefconfig新配置，所有符号都设置为默认值randconfig新配置，随机回答所有选项listnewconfig列出新选项olddefconfig与oldconfig相同，但在没有提示的情况下将新符号设置为其默认值kvmconfig为KVM虚拟机内核支持启用其他选项xenconfig为xen dom0和虚拟机内核支持启用其他选项tinyconfig配置最小的内核

我认为menuconfig是这些选项中最受欢迎的。目标由不同的主程序处理，这些程序由内核提供并在内核构建期间构建。一些目标有一个GUI（为了方便用户），而大多数没有。与Kconfig相关的工具和源代码主要位于内核源代码中的scripts/kconfig /下。正如我们从scripts/kconfig/Makefile中看到的，有几个主机程序，包括conf，mconf和nconf。除了conf之外，它们中的每一个都负责基于GUI的配置目标之一，因此，conf处理它们中的大多数。

从逻辑上讲，Kconfig的基础结构有两部分：一部分实现一种新语言来定义配置项（参见内核源代码下的Kconfig文件），另一部分解析Kconfig语言并处理配置操作。

大多数配置目标具有大致相同的内部过程（如下所示）：



请注意，所有配置项都具有默认值。

第一步是读取源根目录下的Kconfig文件，构建初始配置数据库; 然后它通过根据此优先级读取现有配置文件来更新初始数据库：

.config

/lib/modules/$(shell,uname -r)/.config

/etc/kernel-config

/boot/config-$(shell,uname -r)

ARCH_DEFCONFIG

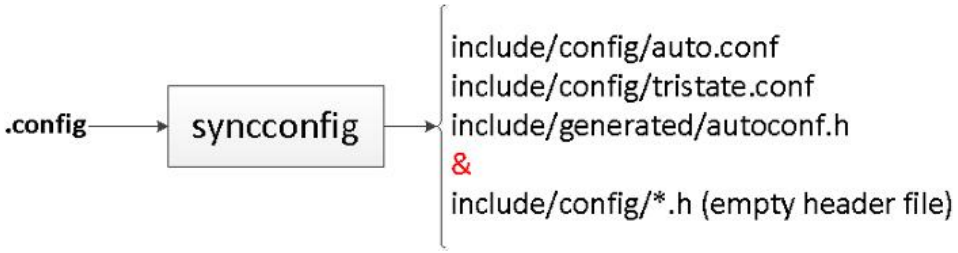
arch/$(ARCH)/defconfig

如果你通过menuconfig进行基于GUI的配置或通过oldconfig进行基于命令行的配置，则会根据你的自定义更新数据库。最后，将配置

数据库转储到.config文件中。

但.config文件不是内核构建的最终配置；这就是syncconfig目标存在的原因。syncconfig曾经是一个名为silentoldconfig的配置选项，但它不会执行旧名称所说的内容，因此它已重命名。此外，因为它是供内部使用（不适用于用户），所以它已从列表中删除。

以下是syncconfig的作用：



syncconfig将.config作为输入并输出许多其他文件，这些文件分为三类：

- auto.conf和tristate.conf用于makefile文本处理。 例如，可以在组件的makefile中看到这样的语句：

obj-$(CONFIG_GENERIC_CALIBRATE_DELAY) += calibrate.o

- autoconf.h用于C语言源文件。
- include / config /下的空头文件用于kbuild期间的配置依赖性跟踪，如下所述。

配置完成后，我们将知道哪些文件和代码片段未编译。

kbuild

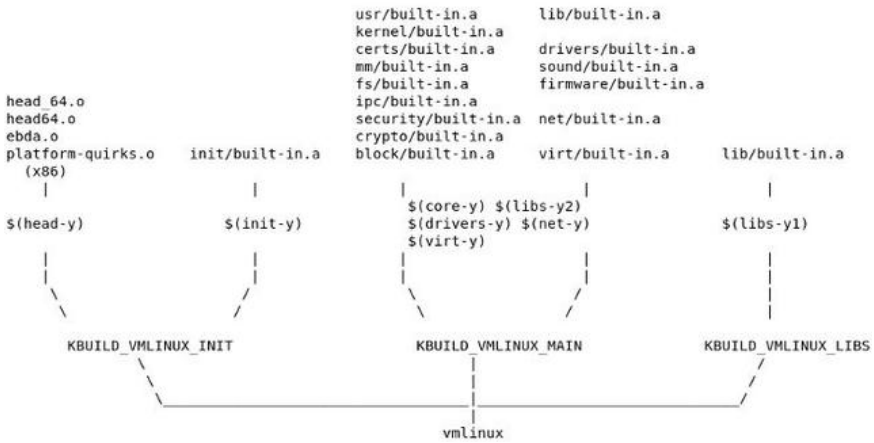
组件式构建（称为递归make）是GNU make管理大型项目的常用方法。Kbuild是递归make的一个很好的例子。通过将源文件划分为不同的模块/组件，每个组件都由其自己的makefile管理。当开始构建时，顶级makefile以正确的顺序调用每个组件的makefile，构建组件，并将它们收集到最终的执行程序中。

Kbuild指的是不同类型的makefile：

- Makefile是位于源根目录中的顶级makefile。
- .config是内核配置文件。
- arch / $（ARCH）/Makefile是arch makefile，它是top makefile的补充。
- scripts/Makefile。*描述了所有kbuild makefile的通用规则。
- 最后，大约有500个kbuild makefile。

top makefile包含arch makefile，读取.config文件，下载到子目录，在scripts/Makefile。*中定义的例程的帮助下，在每个组件的makefile上调用make，构建每个中间对象，并将所有中间对象链接到vmlinux中。内核文档Documentation/kbuild/makefiles.txt描述了这些makefile的所有方面。

作为示例，让我们看看如何在x86-64上生成vmlinux：



（该插图基于Richard Y. Steven的博客。它已更新，并在作者允许的情况下使用。）

进入vmlinux的所有.o文件首先进入他们自己的内置.a，通过变量
KBUILD_VMLINUX_INIT，KBUILD_VMLINUX_MAIN，KBUILD_VMLINUX_LIBS指示，然后收集到vmlinux文件中。

在简化的makefile代码的帮助下，了解如何在Linux内核中实现递归make：

```
# In top Makefile
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps)
 +$(call if_changed,link-vmlinux)
# Variable assignments
vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN) $(KBUILD_VMLINUX_LIBS)
export KBUILD_VMLINUX_INIT := $(head-y) $(init-y)
export KBUILD_VMLINUX_MAIN := $(core-y) $(libs-y2) $(drivers-y) $(net-y) $(virt-y)
export KBUILD_VMLINUX_LIBS := $(libs-y1)
export KBUILD_LDS := arch/$(SRCARCH)/kernel/vmlinux.lds
init-y := init/
drivers-y := drivers/ sound/ firmware/
net-y := net/
libs-y := lib/
core-y := usr/
virt-y := virt/
# Transform to corresponding built-in.a
init-y := $(patsubst %/, %/built-in.a, $(init-y))
core-y := $(patsubst %/, %/built-in.a, $(core-y))
drivers-y := $(patsubst %/, %/built-in.a, $(drivers-y))
net-y := $(patsubst %/, %/built-in.a, $(net-y))
libs-y1 := $(patsubst %/, %/lib.a, $(libs-y))
libs-y2 := $(patsubst %/, %/built-in.a, $(filter-out %.a, $(libs-y)))
virt-y := $(patsubst %/, %/built-in.a, $(virt-y))
# Setup the dependency. vmlinux-deps are all intermediate objects, vmlinux-dirs
# are phony targets, so every time comes to this rule, the recipe of vmlinux-dirs
# will be executed. Refer "4.6 Phony Targets" of `info make`
$(sort $(vmlinux-deps)): $(vmlinux-dirs) ;
# Variable vmlinux-dirs is the directory part of each built-in.a
vmlinux-dirs := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m)
 $(core-y) $(core-m) $(drivers-y) $(drivers-m)
 $(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))
# The entry of recursive make
$(vmlinux-dirs):
 $(Q)$(MAKE) $(build)=$@ need-builtin=1
```

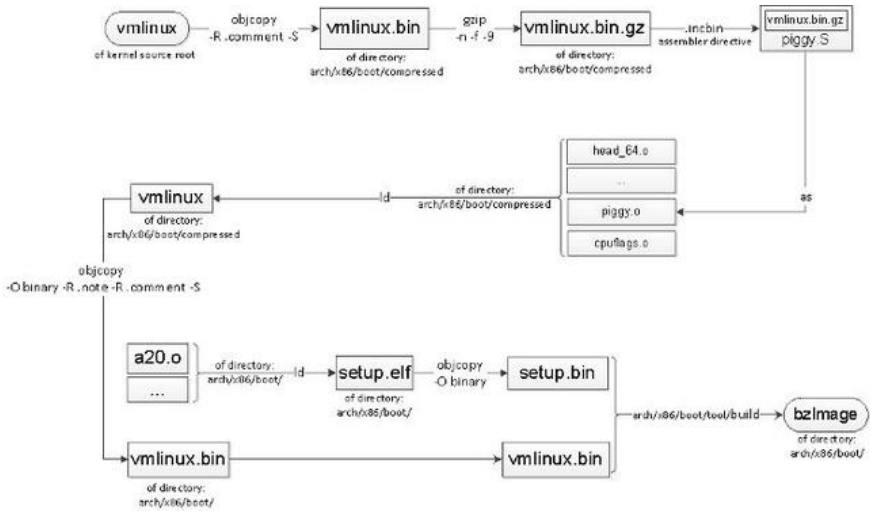
例如，扩展了递归制作配方

make -f scripts/Makefile.build obj=init need-builtin=1

这意味着make将进入scripts/Makefile.build继续构建每个内置的工作.a。 在scripts/link-vmlinux.sh的帮助下，vmlinux文件最终位于源根目录下。

了解vmlinux与bzImage

许多Linux内核开发人员可能不清楚vmlinux和bzImage之间的关系。例如，这是他们在x86-64中的关系：



根vmlinux被剥离，压缩，放入piggy.S，然后与其他对等对象链接到arch/x86/boot/compressed/vmlinux。同时，在arch / x86 / boot下生成一个名为setup.bin的文件。可能存在具有重定位信息的可选第三文件，具体取决于CONFIG_X86_NEED_RELOCS的配置。

由内核提供的称为build的宿主程序将这两个（或三个）部分构建到最终的bzImage文件中。

依赖性跟踪

Kbuild跟踪三种依赖关系：

1. 所有必备文件（* .c和* .h）
2. 所有必备文件中使用的CONFIG_选项
3. 用于编译目标的命令行依赖项

第一个很容易理解，但第二个和第三个呢？ 内核开发人员经常会看到如下代码：

#ifdef CONFIG_SMP

__boot_cpu_id = cpu;

#endif

当CONFIG_SMP更改时，应该重新编译这段代码。 编译源文件的命令行也很重要，因为不同的命令行可能会导致不同的目标文件。

当.c文件通过#include指令使用头文件时，您需要编写如下规则：

main.o: defs.h

recipe...

管理大型项目时，需要大量的这些规则；把它们全部写下来会很乏味和乏味。幸运的是，大多数现代C编译器都可以通过查看源文件中的#include行来编写这些规则。对于GNU编译器集合（GCC），只需添加命令行参数：-MD depfile

In scripts/Makefile.lib

c_flags = -Wp,-MD,$(depfile) $(NOSTDINC_FLAGS) $(LINUXINCLUDE)

-include $(srctree)/include/linux/compiler_types.h

$(__c_flags) $(modkern_cflags)

$(basename_flags) $(modname_flags)

这将生成一个.d文件，内容如下：

init_task.o: init/init_task.c include/linux/kconfig.h

include/generated/autoconf.h include/linux/init_task.h

include/linux/rcupdate.h include/linux/types.h

...

然后主机程序fixdep通过将depfile和命令行作为输入来处理其他两个依赖项，然后以makefile语法输出。<target> .cmd文件，该文件记录命令行和所有先决条件（包括配置） 为目标。 它看起来像这样：

The command line used to compile the target

cmd_init/init_task.o := gcc -Wp,-MD,init/.init_task.o.d -nostdinc ...

...

The dependency files

deps_init/init_task.o :=

$(wildcard include/config/posix/timers.h)

$(wildcard include/config/arch/task/struct/on/stack.h)

$(wildcard include/config/thread/info/in/task.h)

...

include/uapi/linux/types.h

arch/x86/include/uapi/asm/types.h

include/uapi/asm-generic/types.h

...

在递归make中将包含一个。<target> .cmd文件，提供所有依赖关系信息并帮助决定是否重建目标。

这背后的秘密是fixdep将解析depfile（.d文件），然后解析内部的所有依赖文件，搜索所有CONFIG_字符串的文本，将它们转换为相应的空头文件，并将它们添加到目标的先决条件。每次配置更改时，相应的空头文件也将更新，因此kbuild可以检测到该更改并重建依赖于它的目标。 因为还记录了命令行，所以很容易比较最后和当前的编译参数。

展望未来

Kconfig/kbuild在很长一段时间内保持不变，直到新的维护者Masahiro Yamada于2017年初加入，现在kbuild再次正在积极开发。 如果你很快就会看到与本文中的内容不同的内容，请不要感到惊讶。