

make menuconfig

阅读本文及后续文章的 flowchart 的前提和 tips :

- 假设编译发生在源码目录, 未定义KBUILD_SRC, 即没有 make O=
- 假设基于 x86_64
- 不是编译单个存在于单独文件夹下的 module, 即未定义KBUILD_EXTMOD
- flowchart 图中, 以 # 开始的部分为注释
- 假设命令行执行 make 时, 除了 target, 没有其他选项
- **Makefile 执行流程的规则是深度优先**

本系列文章遵循小白的思路进行分析, i.e.:

```
make menuconfig/[all]/modules_install/install/clean(or mrproper,disclean)
```

内核编译的第一步是配置, 即: make *config。有很多 config target 可以用:

config	- Update current config utilising a line-oriented program
nconfig	- Update current config utilising a ncurses menu based program
menuconfig	- Update current config utilising a menu based program
xconfig	- Update current config utilising a Qt based front-end
gconfig	- Update current config utilising a GTK+ based front-end
oldconfig	- Update current config utilising a provided .config as base
localmodconfig	- Update current config disabling modules not loaded
localyesconfig	- Update current config converting local mods to core
defconfig	- New config with default from ARCH supplied defconfig
savedefconfig	- Save current config as ./defconfig (minimal config)
allnoconfig	- New config where all options are answered with no
allyesconfig	- New config where all options are accepted with yes
allmodconfig	- New config selecting modules when possible
alldefconfig	- New config with all symbols set to default
randconfig	- New config with random answer to all options
listnewconfig	- List new options
olddefconfig	- Same as oldconfig but sets new symbols to their default value without prompting
kvmconfig	- Enable additional options for kvm guest kernel support
xenconfig	- Enable additional options for xen dom0 and guest kernel support
tinyconfig	- Configure the tiniest possible kernel

- 注: 曾有 silentoldconfig, 现在被 rename 为 synconfig, 且不会在 make help 中展示出来, 因为名不符实。下文的更新有解释, 下文当提到 "silentoldconfig" 时也即 "synconfig"(2018/5/11)

他们 match 了 top makefile 中的这条 rule :

```
config: scripts_basic outputmakefile FORCE
    $(Q)$(MAKE) $(build)=scripts/kconfig $@

%config: scripts_basic outputmakefile FORCE
    $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

变量 build 定义在 scripts/Kbuild.include 中 :

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

所以当执行 make *config 时, 其实是执行下面这条命令

```
make -f $(srctree)/scripts/Makefile.build obj=scripts/kconfig *config
```

由于 target “*config” 还有很多 prerequisites, 所以我们以 menuconfig 为例, 画出他们的流程图。

在 Top makefile 中 :

```

config, %config<--|--scripts_basic
|                   #_build fixdep & bin2c under scripts/basic
|                   $(Q)$(MAKE) $(build)=scripts/basic
|                   $(Q)rm -f .tmp_quiet_recordmcount
|
|                   # This rule take effect on `make 0=`, so, skip it.
|--outputmakefile
|
|--FORCE
|
|
|                   $(Q)$(MAKE) $(build)=scripts/kconfig $@

```

Target "scripts_basic" 被很多 target 依赖，它的作用是生成整个内核编译过程中所需要的 basic program: fixdep & bin2c。其实 target "config" 最终也是执行一个 host program，用来生成配置文件 .config 等，后面将以它为例介绍编译 host program 的详细过程。

所以，make menuconfig 的过程就剩下执行 config 的 recipe，也就是进入 scripts/kconfig/Makefile 中寻找真正的 target：menuconfig，在此 makefile 中有：

```

menuconfig: $(obj)/mconf
    $< $(silent) $(Kconfig)

hostprogs-y := conf nconf mconf kxgettext qconf gconf
mconf-objs  := mconf.o zconf.tab.o $(lxdialog)

lxdialog := lxdialog/checklist.o lxdialog/util.o lxdialog/inputbox.o
lxdialog += lxdialog/textbox.o lxdialog/yesno.o lxdialog/menubox.o

```

看起来 menuconfig 的 target-prerequisite 关系流程很简单，仅仅是使用一个叫做 mconf 的 host program。那么剩下的问题就分为了 2 个：

- i. host program "mconf" 是如何生成的？
- ii. mconf 如何生成 .config 等配置文件？

关于第一个问题，首先需要了解 kernel build 的 recursive make 的框架。由上面 config 的 rule 可以看出，recursive make 是通过这条 recipe:

```
$(Q)$(MAKE) $(build)=scripts/kconfig $@
```

也就是说，make 进入了 scripts/Makefile.build 中，这是绝大多数 recursive make 的发生入口。分析此 makefile 可以发现，它依次包含了几个文件

```

-include include/config/auto.conf # 由于是 clean build，还没有 make menuconfig，auto.conf 此时是没有的，这
也是为什么前面有特殊前缀字符 "-"
include scripts/Kbuild.include
include $(kbuild-file) # kbuild makefile，目的编译目录的makefile
include scripts/Makefile.lib # 对 kbuild makefile 中定义的通用变量(如 obj-y, obj-m) 进行处理
include scripts/Makefile.host # 如果 kbuild makefile 中有定义 host program，才会包含此 makefile，处理 host
program 相关的变量

```

每一次 recursive make 的目标都定义在 kbuild makefile 中，所以从某种意义上说，它是上述几个 makefile 中最主要的，其他的作用作为 kbuild 的 framework。

host program "mconf" 是如何生成的？

Documentation/kbuild/makefiles.txt 的 "4 Host Program support" 对此有全面的介绍，必读。

由上面 scripts/kconfig/Makefile 中的片段代码可知，make menuconfig 时使用的 host program 是 mconf，且 mconf 只由 .c 文件编译而来，所以下面的代码级分析仅针对此例进行分析。但需要知道，此例只是很多场景中的一种，本文仅作抛砖引玉，若想了解全部，必须通读 Makefile.host。

上面说了，Makefile.host 会处理 kbuild makefile 中定义的 host program 相关的变量。

```

# 本段代码块来自 scripts/Makefile.host
__hostprogs := $(sort $(hostprogs-y) $(hostprogs-m))

# C code. Executables compiled from a single .c file
# 从 kbuild makefile 中定义的 host program 中挑出仅由一个.c文件编译的program

```

```

host-csingle := $(foreach m,$(__hostprogs), \
                $(if $($m)-objs)$($m)-cxxobjs),,$(m)))

# C executables linked based on several .o files
# 从 kbuild makefile 中定义的 host program 中挑出由若干 .c 文件编译的 program, 也叫做 Composite Host Program.
# 本例中, mconf 就是一个 composite host program
host-cmulti := $(foreach m,$(__hostprogs),\
                $(if $($m)-cxxobjs),,$(if $($m)-objs),$(m)))

# Object (.o) files compiled from .c files
# 得到编译所有 host programs 所需的所有 .o 文件名字。其实也就是得到所有需要编译的 .c 文件(没有 cpp 文件)
host-cobjs := $(sort $(foreach m,$(__hostprogs),$(m)-objs))

# 仅仅加上 $(obj) 前缀。结果是下面变量中的文件名都是相对于 kernel source 根目录的相对路径名
__hostprogs := $(addprefix $(obj)/,$(__hostprogs))
host-csingle := $(addprefix $(obj)/,$(host-csingle))
host-cmulti := $(addprefix $(obj)/,$(host-cmulti))
host-cobjs := $(addprefix $(obj)/,$(host-cobjs))

# 编译 host program 所需要的 flag。关于 flag 的处理, 可以另成一文。
_hostc_flags = $(HOSTCFLAGS) $(HOST_EXTRACFLAGS) \
                $(HOSTCFLAGS_$$(basetarget).o)
ifeq ($(KBUILD_SRC),)
    _hostc_flags = $(_hostc_flags)
else
    _hostc_flags = -I$(obj) $(call flags,_hostc_flags)
endif
hostc_flags = -Wp,-MD,$(depfile) $(_hostc_flags)

# 编译 host program 所需要的所有 rule 都定义在本 makefile 中。(与 kernel 本身的编译略有不同,
# 编译 Kernel 和 module 所需要的所有 rule 基本定义在 Makefile.build 中)
# 首先将 .c 编译成 .o。if_changed_dep 函数来自 Kbuild.include 文件, 将在下一个代码块中介绍。
# 一个 .c 文件生成对应的 .o 文件。host-cobjs -> .o
quiet_cmd_host-cobjs = HOSTCC $@
cmd_host-cobjs = $(HOSTCC) $(hostc_flags) -c -o $@ $<
$(host-cobjs): $(obj)/%.o: $(src)/%.c FORCE
    $(call if_changed_dep,host-cobjs)

# 通过上面的 rule 生成了所有的 .o 文件, composite host program 由它依赖的所有 .o 文件链接而来。
# 这里用了一个小技巧, 下面3行代码包含了 2 条 rule, 第一条很容易辨别出, 仅定义了 recipe ;
# 第二条使用了自定义的函数 multi_depend, 它会调用 eval 函数生成一条规则来描述依赖关系。
# 最终在 makefile 的数据库中, 会将这两条 rule 合并成一条。
# multi_depend 函数来自 Makefile.lib, 将在下面的代码块中介绍。
$(host-cmulti): FORCE
    $(call if_changed,host-cmulti)
$(call multi_depend, $(host-cmulti), , -objs)

# 将所依赖的 .o 文件链接成可执行文件时所使用的命令行。
quiet_cmd_host-cmulti = HOSTLD $@
cmd_host-cmulti = $(HOSTCC) $(HOSTLDFLAGS) -o $@ \
                  $(addprefix $(obj)/,$($(@F)-objs)) \
                  $(HOST_LOADLIBES) $(HOSTLOADLIBES_$(@F))

```

上面使用了多个 kbuild 自己定义的函数：if_changed/if_changed_dep, multi_depend. 尤其 if_changed 函数系列, 在各种 makefile 中曝光率颇高。下面详细分析他们的实现

```

# 下面的代码来自 scripts/Kbuild.include

# if_changed      - execute command if any prerequisite is newer than
#                  target, or command line has changed
# if_changed_dep  - as if_changed, but uses fixdep to reveal dependencies
#                  including used config symbols
# if_changed_rule - as if_changed but execute rule instead
# See Documentation/kbuild/makefiles.txt for more info

# Execute command if command has changed or prerequisite(s) are updated.
if_changed = $(if $(strip $(any-prereq) $(arg-check)), \
                @set -e; \
                $(echo-cmd) $(cmd_$(1)); \
                printf '%s\n' 'cmd_$(@) := $(make-cmd)' > $(dot-target).cmd, @:)
```

```
# Execute the command and also postprocess generated .d dependencies file.
if_changed_dep = $(if $(strip $(any-prereq) $(arg-check) ), \
    @set -e; \
    $(cmd_and_fixdep), @:)

# Find any prerequisites that is newer than target or that does not exist.
# PHONY targets skipped in both cases.
# 原注释已经表达的很清楚了，无需再进行注释。自动变量的定义参考 GNU make 文档 10.5.3 节。
any-prereq = $(filter-out $(PHONY),$?) $(filter-out $(PHONY) $(wildcard $^),$^

# Check if both arguments are the same including their order. Result is empty string if equal.
# 此处涉及到另外一大波姿势。目前只需知道，在编译时通过给 gcc 传递参数，可以让它帮助生成某个 .c 文件的
# 依赖关系描述文件(*.d)，参考 GNU make 文档的“4.14 Generating Prerequisites Automatically”。
# 然后，kbuild 会 post process 此文件，并命名为 .<target_name>.cmd，使其包含更丰富的内容，其中一条是
# 编译该 target 的详细命令行参数。
# 所以，此语句的意思是：针对某 target，比较上次和本次编译的命令行参数是否发生了变化，若命令行参数发生了
# 变化，则重新编译该 target。cmd_@ 是上次的命令行，cmd_$1 是本次的命令行。
arg-check = $(filter-out $(subst $(space),$(space_escape),$(strip $(cmd_@))), \
    $(subst $(space),$(space_escape),$(strip $(cmd_$1))))

# 这是编译 target: *.o 最终执行的所有命令：
# 1. 第一行显示执行的命令并执行。
# 2. 后面三行处理生成依赖关系描述文件(post process the generated .d file)。
cmd_and_fixdep = \
    $(echo-cmd) $(cmd_$1)); \
    scripts/basic/fixdep $(depfile) @$ $(make-cmd)' > $(dot-target).tmp;\
    rm -f $(depfile); \
    mv -f $(dot-target).tmp $(dot-target).cmd;
```

上面详述了 `if_changed_def` 的过程，`if_changed` 大致一样，省略其过程分析。
`multi_depend` 函数定义在 `scripts/Makefile.lib`，如下：

```
# Useful for describing the dependency of composite objects
# Usage:
# $(call multi_depend, multi_used_targets, suffix_to_remove, suffix_to_add)
define multi_depend
$(foreach m, $(notdir $1), \
    $(eval $(obj)/$m: \
        $(addprefix $(obj)/, $(foreach s, $3, $(strip $(subst $(s),$(s),$(strip $(obj)/$m))))))
    endef
```

以 `mconf` 举例，生成的描述依赖关系的 rule 长这样：

```
scripts/kconfig/mconf: $(mconf-objs)
```

由上面的 makefile 可知，`mconf` 依赖 `mconf.o` `zconf.tab.o` 和其他 `.o` 文件。细心的读者会发现，在干净的源码目录 `scripts/kconfig/` 中，`mconf.c` 可以找到，但 `zconf.tab.c` 却没有，why？

我们找一下 `zconf.tab.o` match 了哪儿些 rule。在 `kbuild` makefile 中有：

```
$(obj)/zconf.tab.o: $(obj)/zconf.lex.c
```

同时它也 match 了 `Makefile.build` 中最普遍使用的一条 rule:

```
$(obj)/%.o: $(src)/%.c $(recordmcount_source) $(objtool_obj) FORCE
```

所以，`zconf.tab.o` 依赖 `zconf.lex.c` 和 `zconf.tab.c`，但这里两个文件都不在，如何生成他们？在 `Makefile.lib` 中刚好有这条看起来很万能的 rule 可以 match:

```
$(obj)/%: $(src)/%_shipped
    $(call cmd,shipped)
```

目录中果然有 `*_shipped` 文件，Finally Got it! 它的 recipe 也很简单，只是做了文件名的转换。
 细心的读者还会发现，为什么编译结果中没有 `zconf.lex.o`？这里的 trick 是：在 `zconf.tab.c` 的最下面有：

```
#include "zconf.lex.c"
#include "util.c"
#include "confdata.c"
```

```
#include "expr.c"
#include "symbol.c"
#include "menu.c"
```

好吧，这就是 trick，在 .c 文件中包含 .c 文件~为什么它的处理这么绕弯？可以参考：[special handle of scripts/kconfig/zconf.tab.o](#)

参考了上面的链接就会发现，这里所涉及的知识还不仅这一点，还包括 flex, bison, gperf 等，另一个大千世界。

这就是 mconf 的编译过程，本节只介绍了框架，其中隐藏了无数细节姿势等你去探索

.config 文件是如何生成的？How Kconfig work？

(2018/5/11更新)

.config 的生成过程基本上 cover 了 Kconfig 大部分工作内容，和 kbuild 就没有关系了。Kconfig 定义了一套 language，本节不涉及代码如何解析 kconfig language，因为这需要更多的基础知识，同时对于本节的主旨帮助不大。

上一节中我们已经了解，`make menuconfig` 的过程其实是一个叫做 mconf 的 host program 来做配置。用来做配置的 host program 不止 mconf，从 scripts/kconfig/Makefile 中可以看出还有 conf, gconf, nconf, qconf，除了 conf 是用命令行的方式，其他的都是使用各种不同图形界面。

因为需要解析 kconfig language，所以这些 host program 的生成需要一套语言解析的特殊文件，也即那些 *_shipped 文件，他们分别是用 gperf, flex, bison 生成的，源文件和 .c 文件的对应关系如下

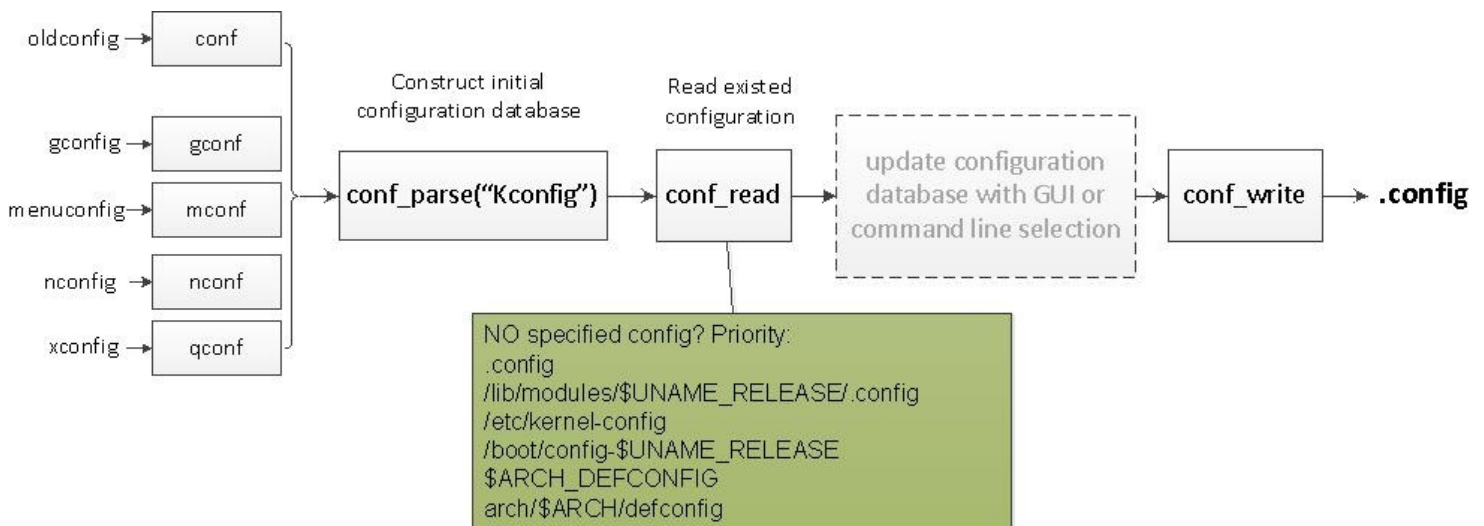
- `.l -- flex --> .lex.c`
- `.y -- bison --> .tab.c`
- `zconf.gperf -- gperf --> zconf.hash.c`

_shipped 文件是上面这三个 .c 文件的 off-the-shelf(现成的) 版，由上一节可知，仅仅是做了文件名的转换。为什么提供这些 _shipped 文件呢，目的是为了减少 kernel 编译对于 host system 的依赖(依赖恰当版本的相应工具)，同时加速 make *config 的过程，这也是因为多年前机器的性能低下导致那些工具的处理相对今天慢很多。但到了2018年，性能问题已经不是瓶颈。

我在 2017/8/15 问了关于 *_shipped 文件的[问题](#)，也许是受到我的启发，maintainer 于 2017/8/19 发了 [RFC patch](#)，来删除这些 shipped 文件，而直接使用相应工具生成，得到了 linux 本人的赞许。在交流中，Linus 本人发现了 gperf 的一些兼容性问题，遂将 gperf 的使用在 kernel 中完全删除(commit: bb3290d91695bb1ae78ab86f18fb4d7ad8e5ebcc)。最后在 2017/12 月，maintainer 发了 [patch](#)，将 shipped 文件全部删除。

scripts/kconfig/Makefile 中定义了很多 config 的 target，这些 target 分别由不同的 host program 处理，且他们的目的并不完全相同。绝大部分 target 是外部接口(用户使用)，作用是生成 .config 文件，唯独 silentoldconfig(现在叫synconfig) 不是供用户使用，它不仅会生成 .config，还进一步生成 include/config/auto.conf，include/config/tristate.conf，include/generated/autoconf.h，以及 include/config/ 下的所有空白头文件(由函数 conf_write_autoconf 实现，并只在 silentoldconfig 时调用)，这些文件才是编译时实际使用的配置文件，而不是 .config 文件。auto.conf 和 tristate.conf 在 Makefile 的处理过程中被使用，include/config/ 下的空白头文件存在的原因详见 scripts/basic/fixdep.c 开头的描述。简而言之：GCC 的编译选项 -MD 帮助生成文件依赖关系 .d，但是在 kernel 编译环境下，这个依赖关系中缺少了该源码文件对配置项开关的依赖关系，所以 fixdep 耍了一个 trick，将源码对配置项开关的依赖转变成文件依赖关系，并将其添加到 .cmd 文件中。也就是说，如果配置项发生改变，它对应的 include/config/xxx 也会改变，而这个依赖关系存在于 fixdep 生成的 .cmd 文件中。通过这个 trick 便可以在下次编译时侦测到配置项的变化。

参考下图感受 .config 的产生过程：



所有做 kconfig 的 host program 的代码开始时读取所有目录下的 Kconfig 文件，将所有配置项存储在内部的数据结构中(这个过程是由函数 `conf_parse` 完成)，我把它叫做 initial configuration database(注意，Kconfig 文件中的每一个配置项都有其默认值)。不同的 config target 根据不同的输入条件来调整 initial configuration database，最后写入 .config 文件。这些不同的输入条件大致有下面两类：

1. 基于已有的配置文件和 GUI/Command line 选择，更新 initial configuration database；
2. 前缀为 all* 的 config target 和 randconfig，顾名思义，他们仅仅是按需更新 initial configuration database。

`conf_read` 函数用于读取一个已有配置文件。当使用 `defconfig` 时，arch Makefile 中会指定 default config 文件(更详细解释见下文)；当使用 `config`, `nconfig`, `menuconfig`, `xconfig`, `gconfig`, `oldconfig`, `olddefconfig` 时，则按照图示中的优先级查找可用的。`conf_write` 函数将 configuration database 写入到 .config 文件。

除了生成额外的文件，`silentoldconfig` 和 `oldconfig` 一样，只是基本不可能(不是做不到) drop you to the command line，因为 `silentoldconfig` 仅在编译过程中供内部使用，这时有几种情况：

1. .config 不存在，将报错 *Configuration file ".config" not found! Please run some configurator (e.g. "make oldconfig" or "make menuconfig" or "make xconfig")*。
2. .config 已存在，且其内容(配置项名字和数量)与当前 kernel 的配置项(所有 Kconfig 的内容)一致，这意味着不存在未配置的新 configuration item，说明已经做过 `make config`。
3. .config 已存在，但其内容和当前 kernel 的配置项不一致，如果不进行 `make config`，直接 `make`，会被 drop 到命令行进行交互式配置。这种情况出现的场景是：更新后的代码引入了新的配置项，而用户直接执行 `make`，这时会被 drop 到命令行进行配置。

`silentoldconfig(synconfig)` 生成的文件中，`auto.conf` 的内容是配置项 value = "n" 之外所有的项；`tristate.conf` 的内容是配置项 type = S_TRISTATE && value != "n" 的所有项；`autoconf.h` 的内容跟 `auto.conf` 的内容一致，只是用 C 语言的格式，这两个文件的实际有效行数是一样的。.config 中那些标记为“# CONFIG_xxx is not set”的意思是该配置项的值是 "no"。

`defconfig` 表示使用当前 ARCH 的默认配置。每个 ARCH 在 `arch/$(ARCH)/configs/` 目录下提供了一些默认配置文件，当前 ARCH 使用哪一个由变量 `KBUILD_DEFCONFIG` 表示，定义在 `arch/$(ARCH)/Makefile` 中。如果 arch Makefile 没有提供 `KBUILD_DEFCONFIG` 的定义，代码中会通过函数 `conf_get_default_confname` 来获取一个默认配置文件名：`arch/$(ARCH)/defconfig`。在众多 ARCH 中，只有有下面两个没有定义 `KBUILD_DEFCONFIG` 变量：

```
arch/s390/defconfig
arch/alpha/defconfig
```

还有一些 ARCH 提供的默认配置文件名是 `arch/$(ARCH)/configs/defconfig`，容易使读者 confusing，其实也在 Makefile 中定义了 `KBUILD_DEFCONFIG`：

```
arch/riscv/configs/defconfig
arch/arm64/configs/defconfig
arch/nds32/configs/defconfig
```

当你想保存当前的配置为一份默认配置时，正确的做法是通过：

```
make savedefconfig
```

在源码根目录下生成 `defconfig` 文件，然后

```
cp defconfig arch/$(ARCH)/my_cool_defconfig
```

注意，这个名字必须以 "_defconfig" 结尾。使用的时候则：

```
make my_cool_defconfig
```

即可。值得注意的是，无论是生成的 `defconfig`，还是 `arch/$(ARCH)/configs/` 下的默认配置文件，他们的内容都是只保存和默认值不一样的配置项(还记得上面说过每一个配置项都是有默认值的吗)，所以他们的内容相对 .config 来说很短。换句话说，他们保存的是一个增量。

`localyesconfig` 和 `localmodconfig` 的处理基于 `scripts/kconfig/streamline_config.pl` 生成仅包含当前已加载 module 信息的 .config 文件(临时文件)，然后再执行 `oldconfig`，基于刚才的临时 .config，生成最终的 .config 文件。`streamline_config.pl` 的作用在它的文件头部解释的很清楚：

What it does?

If you have installed a Linux kernel from a distribution that turns on way too many modules than you need, and you only want the modules you use, then this program is perfect for you.

It gives you the ability to turn off all the modules that are not loaded on your system.

xenconfig 和 kvmconfig 的处理过程一致，通过 scripts/kconfig/merge_config.sh 将 kernel/configs/ 下相应的 config 文件 merge 到 .config，然后执行 `make oldconfig`

An introduction to scripts/basic/fixdep

从 fixdep 所在的路径也可以看出，它是 kernel build 的一个基础工具，它仅有一个 fixdep.c 生成。fixdep.c 的文件头部注释对它的作用解释的很清楚。

fixdep 的使用方式定义在 scripts/Kbuild.include

```
cmd_and_fixdep = \
$(echo-cmd) $(cmd_$(1)); \
scripts/basic/fixdep $(depfile) @$ '$(make-cmd)' > $(dot-target).tmp;\
rm -f $(depfile); \
mv -f $(dot-target).tmp $(dot-target).cmd;
```

它接受 3 个参数：gcc 编译选项 -MD 生成的 .d 文件，target name，以及编译当前 target 的命令行。它的作用是基于 .d 依赖关系文件，生成包含更多内容的 .cmd 文件。其中比较重要的一项工作是，将当前 target 对 configuration 的依赖关系写入 .cmd 文件。通过扫描 .d 文件，获得依赖文件的列表，依次打开并扫描每一个依赖文件，查找 "CONFIG_" 开头的配置项名，并将该配置项名字按如下样式写入 .cmd 文件：

```
$(wildcard include/config/init/env/arg/limit.h)
```

这文本对应了 CONFIG_INIT_ENV_ARG_LIMIT。每当该配置项发生变化时，kconfig 就会更新对应的 empty header file 的时间戳。这样，下次编译的时候，make 就可以 catch 到依赖的更新，从而重新编译 target。

Host program 编译选项的处理

上面的代码中已知，host program 编译 flags 的处理如下：

```
_hostc_flags = $(HOSTCFLAGS) $(HOST_EXTRACFLAGS) \
$(HOSTCFLAGS_$(basetarget).o)
ifeq ($(KBUILD_SRC),)
__hostc_flags = $(_hostc_flags)
else
__hostc_flags = -I$(obj) $(call flags,_hostc_flags)
endif
hostc_flags = -Wp,-MD,$(depfile) $(__hostc_flags)
```

“-Wp,-MD” 用来生成 .d 依赖关系文件。HOSTCFLAGS 定义在 top Makefile 中，是全局的 host program 编译选项；某目录下所有的 host program 如果要使用特定的编译选项，应在其目录下的 Makefile 中使用 HOST_EXTRACFLAGS；如果某个 host program 要使用特定的编译选项，应使用 \$(HOSTCFLAGS_\$(basetarget).o)。

详细且权威的介绍在：[4.4 Controlling compiler options for host programs](#) of Documentation/kbuild/makefiles.txt

参考

[Kernel build command line reference](#)