# 内核驱动程序基础‌

模块篇 2：模块依赖

‌**引言**

在Linux内核开发中，模块依赖是一个重要的概念，它涉及到内核模块之间的相互关系和构建顺序。理解了模块依赖的概念，有助于解决实际开发中遇到的问题。本章介绍模块依赖的基本概念，以及笔者最近在开发中遇到的一个模块依赖在多平台表现差异的实际问题、可能原因、问题总结。

1. **内核模块依赖基础概念**

ko模块（即内核模块）依赖是指一个内核模块在执行其功能时对其他模块或内核功能的依赖。这种依赖性通常表现为一个模块引用了另一个模块导出的符号，如函数、变量或结构体。一个模块可以依赖于另一个模块的功能或符号，这意味着在编译和加载时，被依赖的模块需要先被编译和加载。

* 1. **模块组织结构**

### 树内ko (Intree)

指那些位于Linux内核源码树中的模块（如drivers/char/下的驱动），它们随着内核源码一起编译和构建。典型特征包括：

* + 通过顶层的Kconfig和Makefile参与编译流程
  + 使用obj-y/m直接声明编译目标
  + 依赖关系通过内核构建系统自动解析

### 树外ko (Out-of-tree)

指那些独立于内核源码树之外的模块（如厂商私有驱动），通常位于用户自定义的目录中。需要通过特定的构建机制（如Makefile）来编译和构建。

* + 关键挑战在于符号版本兼容性和依赖链完整性维护。
  1. **依赖方式**

### 低版本内核（传统方式）

模块依赖通常通过Kconfig文件和Makefile来定义。Kconfig文件用于配置选项和依赖关系，而Makefile则用于具体的构建规则。

* ‌**Kconfig文件‌**
  + 定义模块的配置选项（如bool/tristate类型）和依赖关系（depends on语法），生成.config文件指导编译‌。
  + 例如：config XXX\_DRIVER tristate "driver module" depends on ARCH\_X86 表示该模块依赖x86架构‌。
* ‌**Makefile**‌
  + 通过obj-y（内建）、obj-m（模块）等指令实现编译规则，依赖.config中的CONFIG\_XXX变量‌。
  + 多模块编译时可通过形如以下形式组织‌：

obj-$(CONFIG\_XXX1) := A.o B.o

obj-$(CONFIG\_XXX2) += C.o D.o

### 高版本内核（Bazel引入）

* ‌**Bazel工具链**‌
  + 部分高版本内核开始采用Bazel替代传统Makefile/Kconfig，通过声明式构建规则（如cc\_library）管理依赖，提升构建速度和跨平台一致性。
  1. **构建系统配置对比**

### Kbuild传统方式

树内模块条件编译示例：

obj-$(CONFIG\_XXX\_DRIVER) += A.o

A-objs := main.o helper.o

A-$(CONFIG\_MY\_MODULE) += feature.o # 动态编译单元扩展

**局限性**：

* + 依赖解析基于文件时间戳机制
  + 跨平台编译时易出现顺序错乱（如MTK/QCOM的Makefile差异）

### Bazel现代方式

声明式依赖图构建（Starlark语法）：

cc\_binary(

name = "module\_b ",

srcs = ["b.c"],

exported\_hdrs = ["b.h"], # 显式头文件暴露

)

cc\_binary(

name = "module\_a",

srcs = ["a.c"],

deps = [":module\_b"], # 显式跨模块依赖

target\_compatible\_with = ["@platforms//linux:x86\_64"],

)

**技术优势：**

* + 沙盒化构建实现依赖隔离
  + 平台无关的确定性编译顺序

1. **实际问题**
   1. **问题描述**

最近在QCOM芯片平台和MTK芯片平台去增加自研的相同模块代码时，出现了一个差异表现问题：QCOM芯片平台编译和模块功能均正常，MTK芯片平台编译可以编译过但模块功能不正常。

具体情况是，在QCOM和MTK原生树内ko模块A中侵入式修改增加了一个自研函数F1，该自研函数F1调用另一个自研的树外ko模块B中的函数F2。F2在A中被用自定义的宏CONFIG\_MY\_MODULE包裹，CONFIG\_MY\_MODULE宏的定义与赋值均在自研的ko模块B目录下的kconfig文件和perf文件中。由于存在A.ko对B.ko的依赖，因此在控制ko模块A编译的CONFIG\_XXX\_DRIVER宏定义处（模块A目录下的kconfig文件）增加了对CONFIG\_MY\_MODULE宏的依赖。

代码示例：

ko模块A所在目录下的kconfig文件：

# QCOM/MTK芯片平台原生驱动模块，它依赖于MY\_MODULE

config XXX\_DRIVER

tristate "kernel (qcom/mtk) driver module"

depends on MY\_MODULE

default m

help

This is a kernel driver module that depends on MY\_MODULE.

It is built-in, loaded as a module, or disabled.

ko模块B所在目录下的kconfig文件：

# 自研模块MY\_MODULE

config MY\_MODULE

tristate "My kernel module"

default m

help

This is a self-developed kernel driver module.

It is built-out, loaded as a module, or disabled.

然而编译后发现，在QCOM芯片平台，函数F1中CONFIG\_MY\_MODULE宏生效，也就是说，函数F1中被宏CONFIG\_MY\_MODULE包裹的函数F2生效（例如打印了一句调试代码）。而在MTK芯片平台，函数F1中的CONFIG\_MY\_MODULE宏未生效，也就是说，函数F1中被宏CONFIG\_MY\_MODULE包裹的函数F2未生效（例如未打印调试代码）。

* 1. **可能原因【根因待分析】**

分析发现原因可能是跟编译顺序有关，在QCOM芯片平台，编译到原生树内ko模块A时，自研的树外ko模块B已经被编译，因此CONFIG\_MY\_MODULE宏的赋值代码已生效。在MTK芯片平台，编译到原生树内ko模块A时，自研的树外ko模块B还未被编译，因此CONFIG\_MY\_MODULE宏的赋值代码还未生效，但奇怪的是在MTK芯片平台上原生树内ko模块A成功被编译出来了，只是被CONFIG\_MY\_MODULE宏包裹的代码未生效。理论上，如果kconfig中一个控制模块ko编译的宏所依赖的另一个宏未被定义，那么该ko不应该被编译出来。所以在MTK芯片平台上原生树内ko模块A在编译时已经能够看到CONFIG\_MY\_MODULE宏的定义，只是该宏的赋值语句尚未执行。

由于笔者尚未对编译框架有深入了解，根因有待未来分析补充完善。结合第一节的基础概念，上述平台差异表现可能由以下几个关键因素导致：

* ‌**Kconfig依赖关系处理差异‌**

QCOM平台可能通过depends on MY\_MODULE正确建立了模块间的编译顺序依赖，确保B模块先于A模块编译，使得CONFIG\_MY\_MODULE宏在A模块编译时已定义‌。而MTK平台可能未严格遵循此依赖关系，导致宏未生效时A模块已编译完成‌。

* ‌**Makefile/bazel/Kbuild系统配置差异‌**

QCOM平台的Makefile(最近内核版本已更改为bazel)可能显式指定了模块B的编译优先级（如obj-$(CONFIG\_MY\_MODULE)的声明顺序或子目录包含顺序），确保宏定义先被处理‌。MTK平台可能未正确处理树外模块的编译顺序，导致模块B的Kconfig定义未被及时加载‌。

* ‌**编译顺序与预处理时机问题‌**

QCOM平台的Makefile可能通过obj-$(CONFIG\_MY\_MODULE)的声明顺序或子目录包含顺序，强制模块B先编译，使宏在预处理阶段生效‌。MTK平台可能因树外模块的编译顺序未强制依赖，导致模块A编译时CONFIG\_MY\_MODULE宏虽存在但值为空（如未定义y/m）。

* 1. **解决方式**

笔者在定位出问题跟芯片平台kconfig配置或编译顺序差异有关后，就更改了ko间依赖的方式。不再依靠在kconfig文件中让A宏依赖B宏的方式来配置ko间的依赖，而是通过在bazel文件中明确配置ko间的依赖关系。

# Bazel配置模块依赖示意

cc\_binary(

name = "driver",

srcs = ["driver.c"],

deps = ["//modules/B:my\_module"], *# 显式跨模块依赖*

target\_compatible\_with = ["@platforms//linux"],

)

为什么通过bazel配置依赖关系可解决上述实际问题，大体可能原因是：

* Bazel通过声明式的deps属性显式定义模块间依赖，构建时自动生成有向无环图(DAG)，确保编译顺序与依赖链严格匹配‌。而Makefile依赖depends on声明，实际编译顺序由目录遍历顺序决定，在异构平台(如MTK/QCOM)易出现时序错乱，导致宏未生效但编译通过的异常现象‌。
* Bazel为每个目标(target)创建独立沙盒环境，隔离式解析依赖关系，避免全局宏污染。传统Makefile的CONFIG\_XXX宏在跨模块传递时，可能因头文件包含顺序产生作用域穿透问题‌。
* Bazel采用内容哈希校验，仅重编译受影响的模块。Makefile依赖时间戳检查，在并发编译时可能因文件修改时序导致依赖失效，如MTK平台模块B未编译时模块A已开始构建‌。
* Bazel在编译前执行依赖闭包验证，确保所有依赖目标可用。Makefile仅在链接阶段检查符号，导致MTK平台出现“编译通过但运行时失效”的边界情况‌。

总结，Makefile隐式时序依赖与Bazel显式声明依赖的本质区别，导致在不同平台出现模块编译顺序敏感性问题‌。

1. **本章总结**

内核版本在不断迭代演进。在做内核开发时，要深入理解模块间的依赖关系，以及对应的内核版本的模块依赖配置规则。在适配多平台时，需要在每个平台验证，必要时对模块依赖做平台差异化处理。