Linux 内核的模块加载机制

马浩琨

Aug 25, 2026

Linux 内核之所以能在几十年间保持「小巧却无限扩展」的特性，核心原因之一就在于它提供了一套完整的动态代码装载机制，让开发者不必重新编译整个内核，就能把新的驱动、文件系统或协议栈以模块形式热插拔地加入运行中的系统。本文将沿着从用户空间命令到内核内部的完整路径，逐一剖析模块加载、符号解析、依赖管理与卸载的全过程。

1　什么是 LKM

LKM（Loadable Kernel Module）是编译后以 `.ko` 文件形式存在的一段可执行代码，它在加载时被内核动态链接进地址空间，卸载后则彻底释放占用的内存。与内建（built-in）到 `vmlinux` 的代码不同，LKM 在运行时可被按需载入或移除，从而让内核体积保持最小，同时具备按需扩展的能力。从 ELF 文件结构来看，`.ko` 本质上是一个可重定位的目标文件，包含代码段、数据段、符号表和重定位表；此外，它还带有内核专用的 `.modinfo` 段，用于存放模块作者、许可证、依赖关系等元信息。

2　用户空间工具链

当用户在命令行执行 `insmod` 或 `modprobe` 时，实际上是在调用 `finit_module` 或 `init_module` 系统调用。busybox 中的简易实现与 `kmod` 库的完整实现存在差异：前者仅做最基本的文件映射，后者则会解析 `modules.dep` 依赖数据库、自动加载前置模块，并支持模块签名验证。模块签名机制通过 `CONFIG_MODULE_SIG_FORCE` 选项强制要求所有模块必须由受信任的 X.509 证书签名，否则拒绝加载，从而在 Secure Boot 环境中保证内核完整性。

3　系统调用入口

进入内核后，`load_module()` 函数首先读取 ELF 文件头，验证魔数与架构兼容性，然后分配一个 `struct module` 对象，用于记录模块在内核中的全部状态。接下来，内核通过 `module_alloc()` 在专用虚拟地址区间分配可执行内存，并把各个段按权限映射到相应位置。此时模块状态被置为 `COMING`，表示正在初始化，尚未对外部可见。

4 核心装载步骤

布局阶段完成后，内核调用 apply_relocate_add() 对模块中的重定位表进行处理，把所有外部符号地址回填为内核真实地址。符号解析依赖于内核维护的两张表：__ksymtab 保存符号地址，__kcrctab 保存 CRC32 校验值，用于在 CONFIG_MODVERSIONS 开启时检测模块与内核版本是否匹配。开发者通过 EXPORT_SYMBOL 或 EXPORT_SYMBOL_GPL 宏导出的符号会被自动加入上述两张表，供其他模块引用。
模块参数的解析则由 parse_args() 完成：内核在模块加载时扫描 __param 段，把用户通过 modprobe 传递的 key=value 键值对写入对应变量。参数解析成功后，do_init_module() 调用模块通过 module_init() 宏注册的初始化回调；若回调返回非零值，内核会自动回滚已分配的资源，并将模块状态回退到 COMING 之前的阶段，避免半初始化状态残留。

5 依赖与引用计数

模块之间的符号依赖形成了有向图，内核用 drivers/base/module.c 中的引用计数机制来维护拓扑关系。当模块 A 导出符号被模块 B 使用时，B 在加载时会调用 try_module_get() 增加 A 的引用计数；卸载时则通过 module_put() 递减。只有当引用计数为零，且没有设备文件或文件系统仍持有该模块时，delete_module() 系统调用才能真正执行卸载流程。

6 卸载流程

rmmod 命令最终映射到 sys_delete_module 系统调用。内核首先检查引用计数与设备占用情况，确认安全后调用 free_module()。该函数依次执行模块通过 module_exit() 注册的清理回调，释放所有 section 占用的内存，最后调用 vfree() 归还虚拟地址空间。整个过程结束后，模块状态被置为 GOING，随后从内核模块链表中移除。

7 安全与调试

为防止恶意模块加载，内核提供了 lockdown 模式与 SELinux/AppArmor 的细粒度控制。lockdown 模式下，即使拥有 root 权限也无法加载未签名的模块；SELinux 则通过 module_request 钩子对特定域发起的模块加载请求进行审计与拦截。调试时，dmesg 中「Loading module」与「Freeing module」日志、/proc/kallsyms 中的符号表以及 /sys/module/ 下的模块目录，都是定位问题的第一手资料。crash 工具可通过 mod 命令列出当前已加载模块，并用 lx-symbols 自动加载符号文件，极大简化了现场分析。

8 进阶主题

Livepatch 技术允许在不重启系统的情况下，用新函数替换内核既有函数，其实现同样建立在模块加载框架之上。eBPF 则提供了更轻量的内核扩展方式，通过字节码即时编译执行，避免了传统 LKM 的编译与签名流程。Rust for Linux 项目正在把类型安全的 Rust 代码编译为 .ko 文件，并通过稳定的 FFI 边界导出符号，为内核模块开发引入内存安全的新范式。

9 实践示例

编写最小的 hello 模块只需几行代码：

```
#include <linux/init.h>
#include <linux/module.h>

static int __init hello_init(void)
{
   pr_info("Hello, kernel!\n");
   return 0;
}

static void __exit hello_exit(void)
{
   pr_info("Goodbye, kernel!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

这段代码中，module_init 与 module_exit 宏分别把初始化与清理函数注册到内核的回调链表；MODULE_LICENSE 宏则把许可证字符串写入 .modinfo 段，供 modinfo 命令读取。

若要为模块添加参数，可使用：

```
static int debug = 0;
module_param(debug, int, 0644);
MODULE_PARM_DESC(debug, "Enable debug messages");
```

module_param 宏在编译时生成参数描述符，加载时由内核解析并赋值给 debug 变量。

从用户空间的 modprobe 到内核的 load_module()，再到符号重定位、初始化回调与引用计数管理，Linux 模块加载机制形成了一套严密且可扩展的动态链接体系。理解这一体系，不仅能帮助开发者编写可靠的内核模块，也为探索 livepatch、eBPF 等前沿技术奠定了基础。