# 内核基础‌

模块篇 1：模块加载

‌**引言**

在用户空间程序中，main()函数作为程序执行的统一入口点，是编译型语言（如C/C++）设计的核心要求，用于通知操作系统程序执行的起始位置。然而，Linux内核设计中几乎不存在main函数的身影。为什么？笔者初次接触内核时便有此疑问，本篇回答该问题。

1. ‌**内核的结构本质**‌

Linux内核是一个复杂的操作系统核心，由多个独立组件和模块（如内存管理、进程调度、设备驱动）组成。这些组件可能使用不同编程语言实现，并相互协作提供系统功能。因此，内核不需要一个全局的统一入口点（如main()），而是由多个分散的入口函数分别初始化不同子系统。

1. **内核启动过程‌**

内核启动时，内核镜像被引导加载程序（如GRUB）加载后，CPU从预设地址开始执行汇编代码，最终跳转到start\_kernel()（定义于init/main.c），这是硬件执行流的唯一物理入口（注：内核解压后执行的第一个函数，承担全局初始化职责。）。start\_kernel()**是物理启动的入口函数**‌，但‌**并非类似用户空间**main()**的集中式逻辑控制中心**‌，而是作为顶层协调者启动分散的子系统（子模块）初始化流程。各子系统（如内存管理、进程调度）通过专属初始化函数（如mm\_init()、sched\_init()）自我配置，无需start\_kernel()介入具体实现。

更详细的内核启动过程我放入“第4节[附录]内核启动关键初始化步骤” 介绍。请先阅读第三节再看第四节会有更深的体会。

1. **内核模块加载**
   1. **入点与出点‌**

内核模块（如驱动程序）具有入点（entry point）和出点（exit point）机制：

* ‌**入点（加载时触发）**‌：内核模块通过module\_init()宏声明入点函数（如xxx\_init），该函数在模块加载命令（例如insmod或modprobe）执行时被调用，负责初始化硬件、注册设备等操作。入点函数可由开发者任意命名，无需遵循特定格式，但需通过module\_init()注册以被内核识别。
* ‌**出点（卸载时触发）**‌：通过module\_exit()宏声明出点函数（如xxx\_exit），在模块卸载命令（例如rmmod或modprobe -r）执行时调用，用于释放资源、注销设备等清理操作。

这与用户空间程序的main()函数形成对比：

* 用户空间程序：依赖main()作为固定入口，返回即结束进程。
* 内核模块：入点和出点可任意命名，且生命周期由内核动态管理，无需通过函数返回退出。

**这里体现了内核和用户空间程序的设计哲学差异：**用户空间程序需通过main()为操作系统提供明确入口，以支持编译、链接和可执行文件加载。而内核作为资源管理和硬件抽象层，更强调动态扩展性和性能（如热插拔设备），因此采用分散的初始化机制而非集中式入口。

* 1. **内核模块加载方式‌**
     1. **静态编译的核心模块‌**
  + 在编译内核时，通过Makefile中的obj-y选项将模块代码直接链接进内核镜像。
  + 内核启动时，start\_kernel()函数会扫描.init.text段（存放所有静态初始化函数），依次执行内存管理、调度器等核心模块的初始化函数。
  + ‌例如‌：调度器的sched\_init()、内存管理的mm\_init()均在此阶段执行，无需用户干预。
    1. **动态加载的驱动程序‌**
  + 通过Makefile中的obj-m选项编译为独立的内核模块（.ko文件）。
  + 需用户手动执行insmod或modprobe加载模块，或通过init.rc配置insmod命令，内核模块加载器调用sys\_init\_module()系统调用，触发module\_init()注册的初始化函数（如helloworld\_init）。
  + ‌卸载时‌：rmmod命令触发module\_exit()注册的清理函数。

1. **[附录]内核启动关键初始化步骤**

start\_kernel()本身不实现核心功能，而是依次调用各子系统初始化函数：

**4.1 硬件相关初始化**

* setup\_arch()：解析硬件信息（如设备树），完成CPU、内存等底层配置；
* trap\_init()：设置系统自陷入口（如中断向量表），依赖架构相关代码（如arch/alpha/kernel/entry.S）。

‌**4.2 核心子系统启动‌**

* paging\_init()：初始化页表结构，建立虚拟内存映射。
* sched\_init()：初始化进程调度器，包括默认的Bottom-half机制。
* time\_init()：校准时钟中断频率，为定时器提供基准。

‌**4.3调试与安全机制‌**

* lockdep\_init()：建立锁依赖跟踪哈希表，检测死锁风险（如递归加锁或中断上下文冲突）。
* debug\_objects\_early\_init()：初始化调试对象追踪器，监控内核对象生命周期。

‌**4.4后续流程‌‌**

* 完成基础初始化后，内核启动init线程加载用户空间首个进程（如/sbin/init），移交控制权。
* 原start\_kernel()执行线程转为空闲进程（0号进程），进入cpu\_idle()循环。

1. **本章总结**

内核通过分散的初始化函数（如module\_init()注册的驱动入口）实现动态扩展，而用户程序依赖main()的固定生命周期。start\_kernel()的线性执行模式避免了用户空间程序的多级跳转开销，更适合底层硬件操作。此设计体现了内核“按需初始化”和“分层抽象”的核心思想。