对 Linux 可加载内核模块应用框架的研究

Research on Linux loadable kernel module application frameworks

周应华 （重庆邮电大学 计算机学院 重庆 400065）

摘要：可加载内核模块技术是 Linux 提供的一种在系统运行时动态添加、去除、更动操作系统内核功能的技术。Linux 内核模块应用框架使基本内核代码可以方便地调用动态加载的内核模块提供的功能代码，协同为应用提供系统服务，有效支持了内核的可伸缩性。典型的 Linux 内核模块应用框架及工作机制得到了系统的分析和总结，为操作系统内核可伸缩性研究与可加载内核模块应用打下了良好的基础。

关键词：可加载内核模块 模块应用框架 注册 系统调用 内核可伸缩性

1 引言

可加载内核模块技术从内核版本 1.2 以后出现在 Linux 操作系统中。系统使用者不用更改操作系统内核源代码、不用重新编译、安装内核并重启系统，就能实现内核功能代码的添加、更动。在系统性能方面，内核模块在需要它的时候才加载到核心空间，不需要的时候可以卸载以释放内存资源。内核模块动态加载，可以避免把许多可能会用到、但不是一定要用的内核功能代码编译到 Linux 基本内核中，基本内核可以更小，从而避免基本内核占据太多的核心内存空间。在内核功能代码开发方面，内核的开发者不必因为需要不断添加内核功能代码而频繁地更动并编译基本内核，很多内核代码可以实现在模块中，这对于众多的计算机设备驱动程序尤其如此。另外，从软件工程原理“软件模块化”的角度看，内核代码实现在模块中，也有利于这些代码的开发、测试、调试、维护。总的来说，可加载内核模块技术的引进，使得 Linux 操作系统内核更高效，有高度的可伸缩性 [1,2,3,4]。

鉴于可加载内核模块的上述优势和特点，这种技术在很多方面都得到了应用。基于 Linux 2.4 及以上版本，本文第 2 节系统地分析 Linux 内核在设备驱动、文件系统支持、可执行格式的解释、网络协议、语言与字符集、系统安全方面为可加载内核模块提供的应用框架，第 3 节对 Linux 内核模块应用框架进行了总结。

2 可加载内核模块典型应用框架

Linux 可加载内核模块的典型应用有：设备驱动程序、文件系统、可执行格式解释器、网络协议栈、本地语言支持、安全模块支持、系统调用实现；其中，通过动态加载内核模块来添加或替换系统调用的应用虽然较多，但 Linux 内核并不打算提供便利让加载的模块更改基本内核中的系统调用实现代码，所以在系统调用方面，Linux 没有提供成形的应用框架和规范的接口；因此本节只对前六种内核模块典型应用的框架进行分析。

2.1 设备驱动模块应用框架

设备驱动模块应用框架提供 register_chrdev 和 register_blkdev 注册函数，字符设备驱动模块和块设备驱动模块通过这两个函数向设备控制子系统注册；注册以后，模块提供的功能代码就能够被内核所使用 [2],[5]。设备驱动模块应用框架也提供注销函数 unregister_chrdev 和 unregister_blkdev，当字符设备驱动模块和块设备驱动模块不再被需要时，可以通过这两个函数向设备控制子系统注销。字符设备和块设备驱动模块的应用框架见图 1。

通常，应用软件通过对设备文件进行文件系统调用，来使相应的设备驱动模块中的代码被执行，从而控制、操作设备。所以设备驱动模块应用框架要提供这样一种机制，使得打开、读、写设备文件等操作被实现为调用驱动模块中的函数。具体实现机制对于字符设备和块设备有所不同；以字符设备为例：字符设备驱动模块注册后，将在 chrdev 内核数组中加入一个 device_struct 结构的元素，该元素提供对相应字符设备文件进

行打开、读、写操作的函数的指针；当应用软件通过文件系统调用打开相应字符设备文件时，文件系统将为该设备文件创建一个文件对象，并将根据设备文件的主设备号，到 chrdev 数组中找到相应的 device_struct 元素，把它提供的文件操作指针赋予被创建的文件对象；因此，对该设备文件对象的操作都被实现为对注册的设备驱动模块所提供函数的调用。

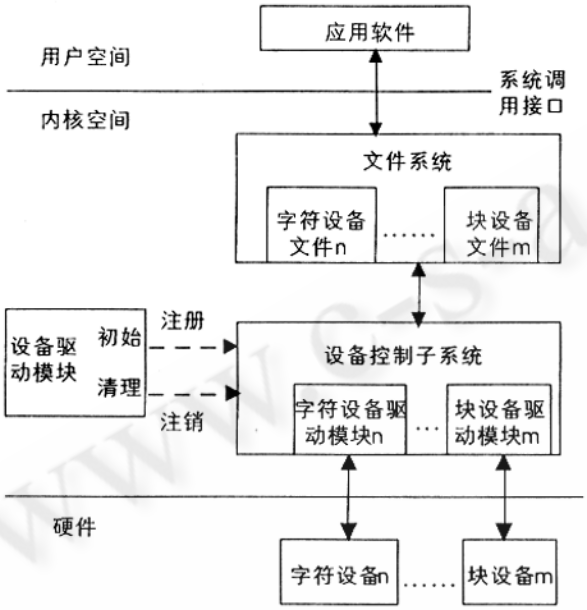

图 1　字符设备和块设备驱动模块应用框架

虽然都是设备驱动，但网络设备驱动模块应用框架和字符设备以及块设备驱动模块的应用框架差别较大。网络设备在系统中不是以设备文件的形式出现，涉及的很多操作往往是通过 socket 系统调用来触发。网络设备驱动模块应用框架提供 register_netdev 注册函数，驱动模块通过它向网络设备子系统注册后，在全局变量 dev_base 所指向的 net_device 结构的链表中加入一项，该数据结构中包含了指针指向该设备驱动模块提供的设备打开、停止、数据发送等操作函数；注册后，模块提供的功能代码就能够被内核使用[6]。网络设备驱动模块应用框架也提供了 unregister_netdev 注销函数，当一个网络设备驱动模块不再被需要时，可以通过它向网络设备子系统注销。网络设备驱动模块应用框架见图 2。

2.2 文件系统模块应用框架

在一个 Linux 系统中，多种文件系统可能同时存在，如：Ext2、MS－DOS、NTFS、NFS、ISO9660 CD－ROM 文件系统等，用内核模块来实现这些文件系统（root 文件系统除外），能充分利用本文引言中所述的内核模块机制的优点。为了给应用软件提供一个统一的文件系统界面，屏蔽不同类型具体文件系统的不同实现，Linux 内核在具体文件系统的上面提供了一个 VFS 虚拟文件系统[2]。见图 3。

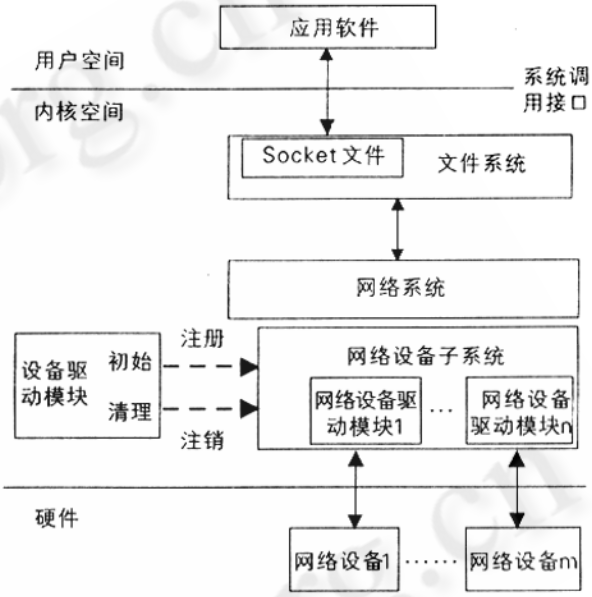

图 2　网络设备驱动模块应用框架

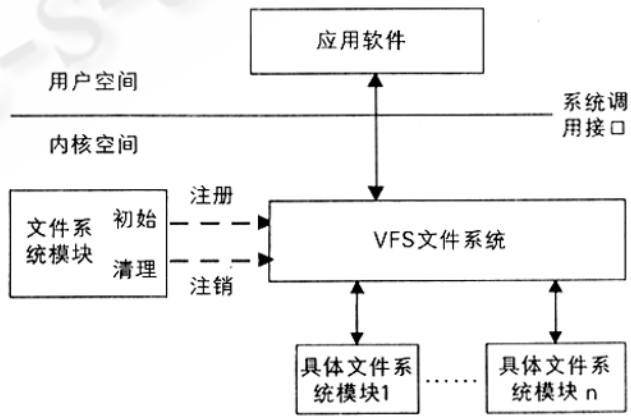

图 3　文件系统模块应用框架

如图 3 所示，当应用软件通过 open 系统调用向 VFS 文件系统要求打开某个挂载的文件系统中的文件时，内核经过复杂的路径查找定位操作，通过挂载的文

件系统，调用文件系统模块提供的函数读取该文件的属性信息以创建 VFS 索引节点，并赋予它指向文件系统模块提供的文件操作函数的指针；需要指出的是，对于特殊文件（如上面提到的设备文件），处理有所不同；然后创建 VFS 文件对象，文件对象中包含的文件操作指针被设置为 VFS 索引节点提供的文件操作指针。因此，后续对该 VFS 文件对象进行的操作能够被实现为对文件系统模块所提供的文件操作函数的调用。

2.3 可执行格式解释模块应用框架

Linux 系统中存在多种格式的可执行文件，如：a.out、ELF、Java 字节码文件、脚本文件等，其格式解释器可以实现为内核模块，当需要执行文件时，解释器才加载到内核中。可执行格式解释模块应用框架提供 register_binfmt 注册函数，一个解释器模块通过该函数向进程管理系统注册后，将向 formats 变量所指向的 linux_binfmt 结构的链表中加入一项[2]。可执行格式解释模块应用框架也提供 unregister_binfmt 注销函数，当一个解释器模块不再被需要时，可通过该函数从内核注销。可执行格式解释模块应用框架见图 4。

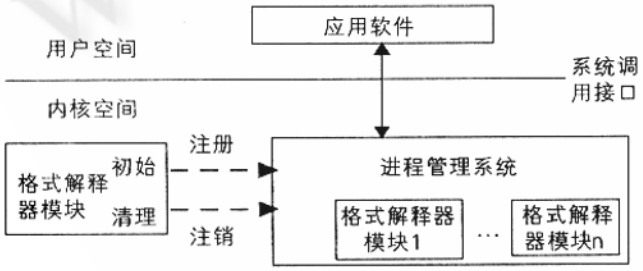

图 4　可执行格式解释模块应用框架

通常，当应用软件发出 exec 系统调用要求执行程序时，进程管理系统扫描 formats 链表，试着用这些 linux_binfmt 节点的 load_binary 方法去加载该可执行文件，load_binary 方法是由格式解释器提供函数来实现的。如果有一个 linux_binfmt 节点的 load_binary 方法执行成功，该程序文件的代码、数据、堆栈段等将被加载，要用到的共享库也将由运行时刻链接器加载到进程地址空间，最后跳到程序的入口点函数处去开始执行。如果在 formats 链表中找不到相应的可执行格式解释器，则通过程序文件最前面的魔数（magic number）序列生成一个模块名，并要求内核动态加载该格式解释器模块。

2.4 网络协议模块应用框架

Linux 操作系统支持多种网络协议族或协议家庭，如：IPv4、IPv6、IPX/SPX、DECnet、Appletalk，协议族的功能代码可以在可加载内核模块中实现；当然，网络分层协议的处理代码也可以通过内核模块实现。网络协议模块应用框架提供 sock_register 函数，协议族模块通过这个函数向网络系统注册后，将在 net_families 数组中加入一个 net_proto_family 结构的指针；网络协议模块应用框架也提供了 sock_unregister 函数，当协议族不再被需要时，可通过该函数从内核注销。网络协议模块应用框架见图 5。

当应用程序想通过 socket 系统调用来进行网络连接、收发网络数据时，首先要通过系统调用建立 socket，内核代码将创建一个 socket 对象，并根据指定的协议家庭编号从 net_families 数组中找到对应的 net_proto_family 结构的指针，通过 net_proto_family 结构包含的 create 函数指针，调用协议族模块提供的函数，把 socket 对象的 proto_ops 函数表设置为由协议族模块提供的 proto_ops 函数表。接下来应用软件对创建的 socket 发出系统调用要求进行网络连接、数据传输、断开连接等操作，就将通过 socket 对象的 proto_ops 函数表中的 connect、sendmsg、recvmsg、shutdown 等函数指针调用由协议族模块提供的功能函数。

2.5 本地语言支持模块应用框架

系统对字符的处理涉及到本地语言的支持，譬如简体中文的 GBK 字符集及支持函数。本地语言支持模块应用框架提供 register_nls 注册函数，一个语言支持模块通过该函数向内核注册后，将向 tables 全局变量所指向的 nls_table 结构的链表中加入一项。本地语言支持模块应用框架也提供 unregister_nls 函数，当一个语言支持模块不再被需要时，可以通过该函数注销。目前 Linux 内核对本地语言支持的应用，主要是对具体文件系统（如：VFAT、NTFS、ISO 9660 文件系统等）的字符处理，对用户而言涉及文件、目录的相关操作。本地语言支持模块应用框架见图 6。

当应用软件向 VFS 发出标准文件系统调用时，上面第 2.2 节已经分析过，很多操作将由具体文件系统模块提供的函数来实现；而挂载的具体文件系统如果需要某种语言支持，那么创建它在内存中的超级块结

构时，超级块结构中就将有指针指向相应的语言支持模块所注册的 nls_table 结构对象，而 nls_table 结构对象包含指针指向语言模块提供的字符处理函数。所以，当具体文件系统进行类似路径查找、读目录、文件更名这样的操作时，能够通过内存中的超级块结构找到对应的 nls_table 结构对象，从而能够调用语言模块所提供的字符处理函数。

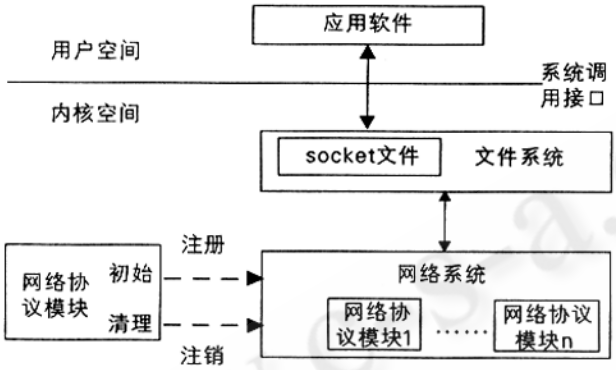

图 5　网络协议模块应用框架

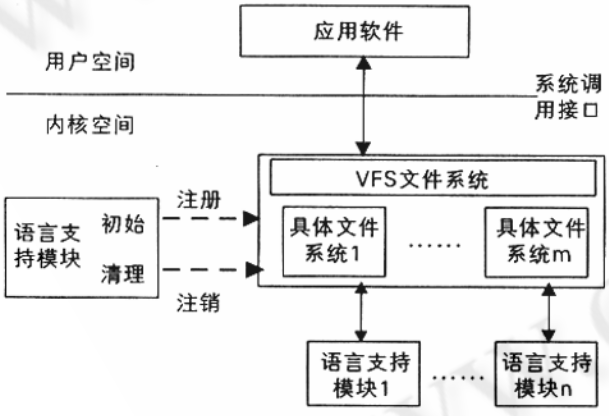

图 6　本地语言支持模块应用框架

2.6 Linux 安全模块应用框架

Linux 内核从 2.6.0 版开始提供安全模块应用框架，使得内核中的多项和安全相关的操作在实施前都必须得到安全模块提供的安全检查函数的认可[7]，[8]。安全模块可以通过 register_security 函数向安全子系统注册，即把模块提供的各安全检查函数设置为 security_operations 结构的全局变量 security_ops 的方法；当一个安全模块不再被需要时，可以通过 unregister_security 函数向安全子系统注销。另外，Linux 安全模块应用框架也支持安全模块堆叠：如果已经有模块向安全子系统注了册，新加载的模块可以通过 mod_reg_security 函数向已加载的安全模块注册，注册是否成功以及注册的结果取决于那个处于控制地位的模块；安全模块堆叠使得一个模块不能提供的安全检查函数，可以由另外的模块提供；当以堆叠方式注册的模块不再被需要时，可以通过 mod_unreg_security 函数注销。Linux 安全模块应用框架见图 7。

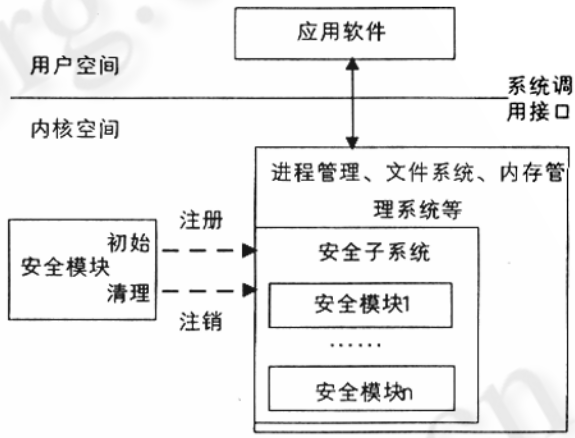

图 7　Linux 安全模块应用框架

应用软件进行系统调用会引发很多内核操作，如：挂载文件系统或卸载文件系统、创建文件或创建目录、对文件加锁、向进程发出信号、等，如果有安全模块向内核注册，很多诸如此类的内核操作就可能受到安全模块提供的函数的安全、访问许可相关的检查，因为这些内核操作在实施前会调用全局变量 security_ops 的相关方法，而这些方法是由注册的安全模块提供函数来实现的。[9]描述了 Linux 安全模块的一个应用。

3　结论

Linux 内核模块应用框架有利于内核模块的动态加载并和原有内核代码形成一个整体，通过统一的系统调用接口，向用户软件提供功能服务。总的来说，可加载内核模块框架有效地支持了 Linux 内核的动态可伸缩性。

（下转第 76 页）

(上接第 72 页)

参考文献

1 Juan – Mariano de Goyeneche, Elena Apolinario Fernandez de Sousa. Loadable kernel modules [J], IEEE Software, 1999, 16(1): 65 – 71.

2 Daniel P Bovet, Marco Cesati. Understanding the Linux kernel. 2nd Edition [M], O'Reilly, 2002.

3 Evi Nemeth, Trent R. Hein, Garth Snyder. Linux administration handbook [M], Prentice Hall, 2002.

4 Bryan Henderson. Linux loadable kernel module howto, [EB/OL] [2006 – 11]. http://tldp.org/howto/module – howto/index.html.

5 Jonathan Corbet, Alessandro Rubini, Greg Kroah – Hartman. Linux device drivers. Third Edition [M], O'Reilly, 2005.

6 Christian Benvenuti. Understanding Linux network internals [M], O'Reilly, 2005: 136 – 168.

7 Chris Wright, Crispin Cowan, Stephen Smalley, et al. Linux security modules: general security support for the Linux kernel [C], In Proceedings of the 11th USENIX Security Symposium. San Francisco: USENIX Association, 2002: 17 – 31.

8 Greg Kroah – Hartman. Using the kernel security module interface [J], Linux Journal, 2002, Issue 103.

9 Miroslaw Zakrezewski, Ibrahim Haddad. Linux distributed security module [J], Linux Journal, 2002, Issue 102.