第34卷 第10期 计 算 机 工 程 2008年5月
Vol.34 No.10 Computer Engineering May 2008
·软件技术与数据库· 文章编号：1000—3428(2008)10—0089—03 文献标识码：A 中图分类号：TP391

可重用 Linux 设备驱动程序框架

袁丽慧[1]，彭 磊[2]
(1. 西华师范大学数学与信息学院，南充 637002；2. 电子科技大学计算机学院，成都 610054)

摘 要：Linux 设备驱动程序的开发工作涉及到相当多的系统内核细节，对开发人员的要求很高。由于缺乏必要的可重用性，一个新设备的驱动程序的开发速度也很缓慢。为了简化其开发流程和提高已有代码的可重用性，该文将 C++语言引入到 Linux 内核环境，以面向对象的方法设计了一个驱动程序开发框架。该框架封装了 Linux 内核对设备驱动程序的生命周期管理和行为管理，可以在保证设备驱动程序质量的基础上，降低开发难度，加快开发速度，规范开发流程。

关键词：Linux 系统；设备驱动程序；框架；C++语言

Reusable Framework of Linux Device Drivers

YUAN Li-hui[1], PENG Lei[2]
(1. School of Mathematics and Information, China West Normal University, Nanchong 637002;
2. School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 610054)

【Abstract】 To develop device drivers for Linux is a skillful work as it is twisted too much with kernel details. On the other hand, due to lack of the necessary reusability, a new driver has to be produced slowly. To simplify the development process and improve the reusability, this paper contributes one object-oriented framework for device drivers by introducing C++ language into Linux kernel. The framework has encapsulated the management of driver's life and behaviors. With aids of the framework, Linux device drivers with good quality can be developed easily, quickly, and normatively.

【Key words】 Linux system; device driver; framework; C++ language

1 概述

软件工程和面向对象的理念已渗透到软件开发领域的多个方面。在桌面和分布式程序的开发过程中，程序员们通常会依赖一些框架而非原生的 API 调用。框架一词在这样的语境下包含了至少两种潜在意义：(1)可重用的类库，该类库包含了常用的数据结构和算法以及封装了系统相关的 API。(2)将程序的启动、初始化、事件/消息传递、结束等系统相关的流程纳入规范化管理，并保持对开发者的透明。框架屏蔽了过于繁琐的系统相关细节，增强了代码复用度，降低了程序开发难度。MFC 和.NET Framework 都具备了这样的特性，成为了当今业界流行的开发框架。

设备驱动程序则有所不同，这种运行在操作系统内核，和硬件直接交互的程序，似乎很难和框架联系起来。大多数人认为框架在为开发工作带来方便、健壮等优越性的同时，也带来了程序尺寸增大，运行速度降低等缺点，这不符合人们对驱动程序的要求。但随着CPU速度和存储器容量的快速发展，这样的情况正在发生变化。事实上Java的发展也经历了相似的阶段，最终用户对驱动程序稳定性和开发者对驱动程序研发速度和质量的需求会成为选择驱动开发工具的首要因素。在Windows上，尽管微软提供了DDK，但开发者们更愿意使用Driverstudio[1]这样带有明显框架特征的工具进行驱动程序的开发。

Linux还没有形成针对设备驱动程序开发的框架，这对Linux的长期生存和发展并非有利。目前Linux的设备驱动开发相当繁琐[2]，尽管几乎所有的设备驱动程序在结构上类似，但是要重用这些代码却并非易事。此外，现有的Linux设备驱动是基于过程设计的，容易出现控制流程过于复杂，模块间耦合度高的问题。

为解决上述 Linux 驱动程序开发过程中的问题，本文设计了一个可重用的 Linux 设备驱动框架，可大大提高代码复用度，简化开发过程，并从面向对象的角度给出了一个设备驱动程序的构造视图。

2 Linux 内核中的 C++

C++和 Java 等其他面向对象的语言不同，除了保持封装、继承、多态 3 个面向对象的基本特点外，还拥有指针可针对内核编程。传统上，Linux 一直采用 C 作为内核开发语言，为了保持和内核的无缝结合，驱动程序也用 C 编码，这在某种程度上影响了驱动程序开发框架的形成。

另一方面，尽管 C++可用于内核编程，但由于驱动和普通应用的差异，以及内核缺乏对 C++运行库的支持使得并不能直接利用 C++构造驱动程序，除非能达到以下两个目标：(1)内核能够对 C++编码的驱动程序实现动态的加载和卸载。(2)C++语言重要的特性仍然得以保留，诸如继承、虚函数等。

2.1 C++与内核的兼容

第(1)个目标实质上是要求从Linux模块装载工具insmod[3-4]的角度看，用C++编码的和C编码的驱动程序在外观上必须保持一致。具体而言，一个驱动程序至少需要具备声明为:int init_module(void)和:void cleanup_module(void)的

基金项目：四川应用基础研究计划基金资助项目(2006J13－067)
作者简介：袁丽慧(1977－)，女，讲师，主研方向：嵌入式软件设计方法；彭 磊，博士研究生
收稿日期：2007-05-25 **E-mail**：yuanlihui8841@sina.com

装载/卸载函数，同时，出于保持内核版本兼容的目的，驱动程序的可执行版本还应包括一个.modeinfo节，而该节中的kernel_version字段必须与当前的内核版本一致。

C++在缺省情况下，会对程序使用的名字进行解析，以满足函数重载的需要；另外，C++的缺省优化会将程序中未曾使用的变量删除。但这两种行为不宜出现在此处，前者会重命名驱动程序的装载/卸载点函数，后者则会遗失.modeinfo中的 kernel_version 字段，都会使得 insmod 无法装载该驱动程序。

为了能够顺利装载模块，需用 extern“C”关键字修饰装载/卸载函数，抑制 C++的名字解析机制。同时对位于 module.h 中的 kernel_version 变量用 violate 关键字修饰，强制 C++保有该变量。

在 Linux 某些版本的内核中，使用 C++的一些关键字作为变量，这会造成 C++驱动程序的编译期错误，这种错误可以通过一种名为头文件包裹的技巧解决，其核心为将 Linux 的内核头文件通过#include 指令包裹在一个头文件中，并在这个头文件引用前利用#define 指令转义 C++关键字，引用完毕后，再使用#undef 重置 C++关键字，示例如下：

```
#ifdef __cplusplus
#define new          cxx_new
…//转义所有 C++特有的关键字
#endif
#include “linux_import.h” //头文件包裹
#ifdef __cplusplus
#undef new
…//恢复所有 C++特有的关键字
#endif
```

2.2 C++语言特性的保持

由g++编译器产生的C++对象的内存布局在操作系统内核和用户态一致，基于此，C++语言的封装性和继承性在内核模式下得以保存。但是，多态性必须依赖于C++对象动态创建和销毁的能力，在内核模式下，没有用户态程序的堆区域，没有C++运行库的支持，必须自己维护一个处于内核模式下的“堆区域”和分配策略。解决这一问题的唯一途径就是重载operator new及其operator delete[5]。

C++的关键字 new 在进行 C++对象动态创建时，分为两阶段：

(1)获取存储空间。在此阶段，new 会查找调用要生成对象的 operator new，如果该操作符存在，则由后者完成分配内存的实质性工作。否则将使用全局的 operator new 获得内存。这部分内存空间在 C++运行库的支持下，默认属于用户程序的堆区域。

(2)在已获得的内存空间里进行对象的初始化工作。即调用对象的构造函数。

因此，可以据此在内核中完成相似的功能。在 Linux 内核中，最为常见的获取内存的函数是 kmalloc，此函数使用了内核存储分配器 Slab 的普通高速缓存。可以用它作为全局的分配符。同样，也可以使用 kfree 作为全局的销毁符，示例如下：

```
void* operator new(size_t size)
{
   return kmalloc(size,GFP_KERNEL);
}
void operator delete(void* deadobject)
{
    kfree(deadobject);
}
```

如果某些对象需要频繁地产生或清除，则可以使用 kmem_cache 函数族来定制实现类的成员函数的动态分配。kmem_cache 函数族能保证对象拥有一个字节边界对齐的对象缓冲池。

如本节所述，由于内核缺乏对 C++的运行库支持，因此在内核中完全发挥 C++的特点比较困难。但是总是可以通过对 C++对象的了解和 Linux 内核的研究来逐步达到这样的效果，最终形成一个近似于 C++运行库的内核版本。

3 Linux 设备驱动程序框架

一个设备驱动程序实际上是由两部分组成的：装载/卸出部分和驱动程序功能体部分。装载/卸出部分使得驱动程序可以被Linux内核加载或卸载，实现形式为Linux内核模块接口；驱动程序功能体则完成具体工作。功能体还可细分硬件IO和软件策略。前者负责硬件的IO端口操作，后者依赖内核提供的服务，完成如缓冲区、定时器、并发/独占访问等驱动程序的管理策略。以UML[6]作为描述语言，该结构如图 1 所示。

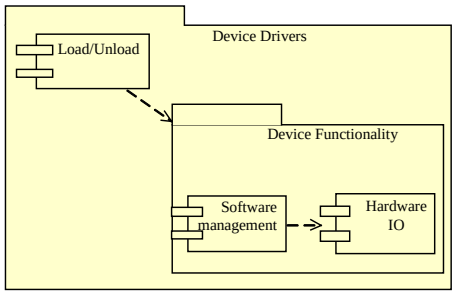

图 1 设备驱动程序构成组件

从驱动开发者的角度看，开发一个设备驱动程序首要工作是完成和硬件的通信，其次是怎样更有效和更好地利用硬件。而其他系统内核的相关细节则可能成为对他们主要目标的干扰——这正是框架被开发者们需要的原因。框架可以屏蔽繁琐的系统细节，提供设计优良的算法，辅助开发人员更专注他们的工作目标。

3.1 框架类库结构

采用 C++作为内核开发语言，使得框架的类库设计和实现工作的难度降低。根据上文对设备驱动程序的结构分析，设计出的类库应该包括两个部分的抽象，分别对应装载/卸载机制和设备功能体。同时，类库提供内核服务和硬件访问机制的封装，以及定义良好的数据结构和算法用于辅助设备进行自身状态和资源管理。

图 2 是对 Linux 设备驱动框架的简要描述。类库中，Kdriver, KDevice 及其子类被定义为抽象类，类中除了声明满足 Linux 设备驱动规范的接口外，同时封装了系统请求分发等规程。KStream 代表了字符设备驱动，KBlock 表示块设备驱动。KDispatchObject 及其子类封装了系统内核对象，如互斥同步、内核定时器等。STL for kernel 完成了 C++标准模板库，在内核中可以使用如 set, vector 等高级容器和相应算法。此外还包括了部分 IO 接口标准。框架中，IO 接口、STL、内核对象部分属于简单封装，目的在于提供一个面向对象的外观，可供开发人员直接使用。这部分的类不具备主动活动的能力，受 KDriver 及 KDevice 驱动进行工作。

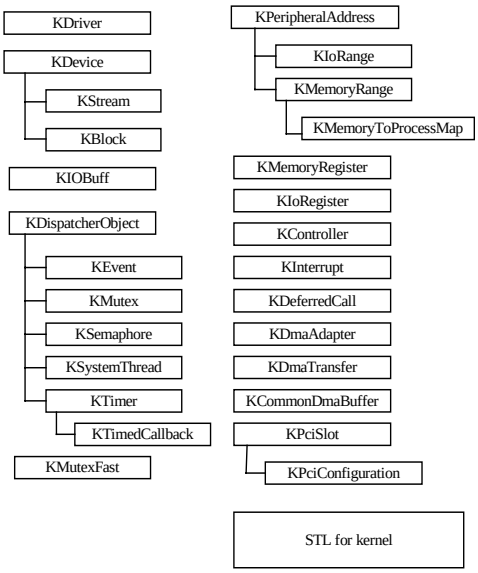

图 2　Linux 设备驱动框架结构

图 3 描述了框架中两个核心类 KDriver 和 KDevice 及其子类的关系，开发人员不允许直接使用这几个类，而应该从这些类中派生出对应于特定设备的子类。

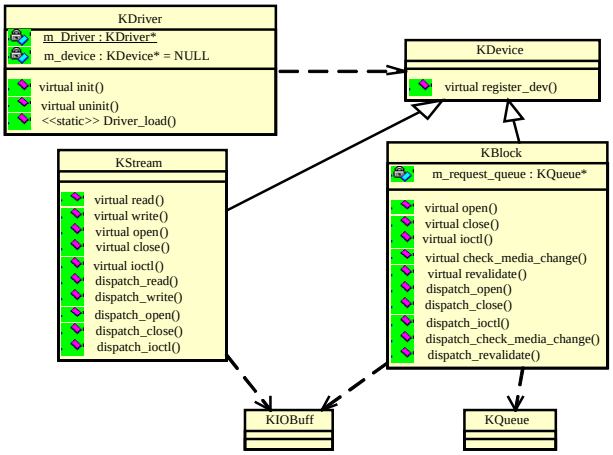

图 3　框架中核心类之间的关系

3.2　框架行为设计

Linux 设备驱动程序框架除了提供易于使用的基础库外，更重要的是将系统繁琐的细节向开发人员屏蔽，让他们专注设备功能部分，如 read, write 等方法的实现。框架通过对象间关系和某些巧妙设计的代码完成这项屏蔽工作。

3.2.1　驱动程序生命周期行为设计

使用传统方法开发 Linux 设备驱动时，开发者总会定义好装载/卸载函数后，然后在装载函数中进行设备注册，再开始初始化设备。这是个公式化的行为，框架应该将这个操作序列封装在自身内部，然后通过一个定义良好的函数允许开发者进行一些自定义的初始化工作。对开发者而言，不再考虑装载和设备注册等过程，此函数成为驱动程序的生命起点。

为了达到这个效果，可以将 Linux 的内核模块装载/卸载函数在框架中实现，届时通过链接器与驱动程序实体绑定，达到对开发者屏蔽的效果。在装载函数内，框架调用 KDriver 的静态函数 Driver_Load，由后者产生 KDriver 子类的实例，再通过虚函数 init 进行子类的初始化。

图 4 是这个过程的图解，整个过程的关键在于需要框架正确产生一个对它而言类型未知的对象实例。参照设计模式中的工厂模式[7]，利用宏DECLARE_DRIVER(class)为一个特定的类指定创建工厂。驱动程序实体通过语句DECLARE_DRIVER(MyDriver)的展开，生成函数create_Driver_instance()，后者返回一个指向MyDriver实体的指针。此函数将被框架在Driver_load中调用，从而获得MyDriver的正确实体。值得注意的是，一个驱动程序中有且仅能有一个KDriver子类的实例，即只能使用一次DECLARE_DRIVER宏，否则将造成符号重名的链接错误。

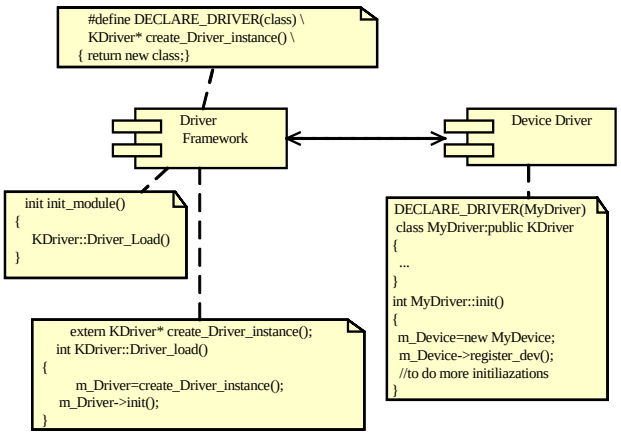

图 4　框架进行设备初始化的内幕工作

驱动程序对象卸载相对容易，卸载点函数被系统调用时，框架通过持有的 m_Driver 调用虚函数 uninit 进行清除工作。

3.2.2　驱动程序实体行为设计

以传统方式进行驱动程序开发，必须对设备驱动的每个接口函数和系统交流的细节信息非常熟悉，例如在 open 接口中记得增加引用计数，并且将驱动程序的状态变量保存到接口要求的 filp->private_data。框架应该在内部封装和系统交互的细节，给出开发者友好的接口用于进行设备特定的处理。

图 5 描述的是框架屏蔽一个字符设备 open 系统调用的细节。设备注册时，Linux 要求设备必须为系统提过的 file_operations 结构绑定设备操作函数。框架将自身的静态成员函数 dispatch_xxx 族填充该结构，成为系统调用的代理响应函数。

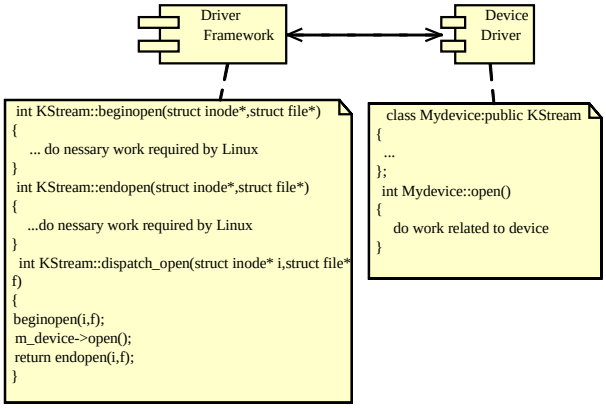

图 5　框架简化设备对系统调用 open 的内幕工作

在图 5 中，当用户通过系统调用 open 打开设备操作时，dispatch_open 被首先调用，该函数完成了和系统内核交互的必要工作后，再调用设备实际的 open 方法，后者能以一个简单的接口面向开发者。最后，dispatch_open 通过内部方法

(下转第 94 页)