一种面向基于属性存取的文件系统的设计与实现*)

谢琳峰 吴 健
(中国科学院软件研究所 北京 100080)

摘 要 基于属性的存取,对于在包含不同来源的大量文件的系统中进行有效的信息管理来说,是一种非常具有吸引力的特性。然而尽管在相当长的一段时间内其价值已广为人知,该特性的真正有实用价值的实现仍然非常之少。本文探讨了实现该特性时面临的主要挑战,以及它们是怎样在一种专为此目的设计的文件系统中,通过成功的应用各种原则而被妥善处理。

关键词 基于属性的查询,栈文件系统,属性-文件映射图,i节点-链接反向映射图

Design and Implementation of a File System Oriented for Attribute-based Access

XIE Lin-Feng WU Jian
(Institute of Software, Chinese Academy of Sciences, Beijing 100080)

Abstract Attribute-based access is an amazing feature for efficient information management in those systems comprising of large quantity of files originated from various sources. While the virtue of the feature has been well recognized in relatively long term, there is rarely qualified implementation for practical usage. This paper addresses main challenges for implementation of attribute-based access, and how they get properly handled by the principles successfully applied in a file system dedicatedly designed to brace it.

Keywords Attribute-based access, Stackable file system, Attribute-file map, Inode-link reverse map

1 引言

随着现代硬件技术的飞速发展,主要信息存储设备的容量在过去的15年中平均提升了近两个数量级。存储资源的丰富,使得个人计算机和其他计算设备的使用者可以利用其长时间地保存大量的来源和种类迥异的个人信息。在这样的应用场景中,如何实现面向海量信息管理的有效机制,已成为现代系统软件设计者所必须面临的一大挑战。

早期的Unix操作系统采用层次化的文件系统(Hierarchical File System)[1]作为信息管理的基础。层次化的文件系统基于单一的视点(Viewpoint)创建目录(Directory),并以此为基础组织作为基本的信息管理单位的文件(File);对于视点的选择无统一的规范。这种机制因简单灵活而广受青睐;至今除现代Unix外,主要的商业操作系统如Windows, Linux, Mac OS等仍沿袭这一传统。但对于系统中存在大量生命周期较长、种类各异的文件的现代应用场景,该机制的固有缺陷使其很难被用于有效的文件管理,从而给系统的用户造成沉重的负担。

在上世纪90年代初,基于属性的存取(Attribute-Based Access)[2]被提出作为层次化文件系统的重要补充技术。然而尽管相关方法的意义获得了广泛的肯定,也有众多的原型系统作为研究的成果被提出,真正具有实用意义的商业实现却难得一见。在当前具有一定用户群体的操作系统发布中,仅有BeOS[3]为代表的一些非主流操作系统包含了对该特性的部分实现;而在主流商业操作系统中,唯一的代表是Mac OS的"Spotlight"[4]技术。相比之下,稍早或同时期提出的日志结构文件系统[5]等技术则早已被包含于主要的商业操作系统发布中。

在本文的以下章节中,我们将首先简述基于属性存取的基本思想,及其相比传统的文件存取机制的优势;然后将探讨作为阻碍其普及的主要原因的,设计与实现该特性必须面临的挑战。在此基础上,我们将阐述在基于标准Linux内核开发的面向基于属性存取的文件系统中,主要的设计问题如何通过成功地应用相关设计原则加以解决。最后,将说明进一步的工作方向。

2 基于属性的存取

在传统的层次化文件系统中,对文件的存取必须通过由其本身的文件名和包含其的各层的目录名组成的"路径名"进行。在此类文件系统中,目录抽象提供了一种从某种视点对不同的文件对象进行分类的机制。如在Linux系统中,所有的设备文件被要求归类于"/dev"目录之下。

作为层次化文件系统的支撑机制,基于路径名的存取存在着以下明显的不足:

(1)在文件的生命周期内,文件的存取被绑定到单一或有限的分类视点。

(2)对于分类视点的选择,基本上由用户根据个人选择决定,不存在系统级的强制或帮助对相关选择进行规范的机制。

以上不足决定了在层次化文件系统中固有的信息管理的不便之处:为快速查找所需要的文件,用户必须准确记忆文件本身或对应的链接的路径名,或者对系统的目录结构有非常清晰的了解。对于现代系统中文件数量以千万计,且随着时

*)本文工作得到863重大专项(编号:2003AA1Z2110)和国家自然科学基金(编号:60373054)的资助。谢琳峰 博士生,研究方向为操作系统;吴 健 副研究员,研究方向为操作系统和中文信息处理。

间而动态增长的情况下，对路径名的记忆已日益成为用户的沉重负担。

作为基于路径名存取的补充和替代方案，以下三种文件的存取机制先后被提出：

(1)基于文件名的存取，即在全局的名字空间中通过文件的文件名进行直接或间接(将文件名转换为其路径名)的存取。商业 Linux 发布普遍安装的 GNU 的 findtutils[6]是这种机制的一种典型实现。

(2)基于文件内容的存取，即通过指定的文件内容中存在的模式对其进行直接或间接的存取。现有的相关实现主要提供基于文件内的关键字/关键字符串序列的查询，如 Unix/Linux 系统中传统的 grep[7]，和 Google 的桌面搜索[8]。

(3)基于属性的存取，即通过指定代表文件某方面特性的属性(Attribute)值组合来对文件进行存取。

相对基于文件名的存取和基于文件内容的存取，基于属性的存取有其无可替代的独特优势。

基于文件名的存取本质上仍是以单一视点对文件对象进行标识，对重名的文件无法作进一步的有效区分。基于文件内容的存取是更具革命性的方法。但合理选择作为文件标识的内容模式在很多情况下比较困难；同时这种方法对于区分内容近似而在其他的方面有明显差异的文件对象(如同一文件面向不同权限用户的各个版本)有明显的局限。而基于属性的存取允许对文件对象进行多视点的标识，在分类的灵活性和文件标识管理的规范性上，胜过以上两种机制。

尽管在实现上的细节略有不同，一般认为完整的基于属性存取的机制应提供以下主要功能：

(1)对文件对象进行动态的属性指派、属性赋值和属性撤消的操作。

(2)在一定的上下文内以指定的属性/值组合查询符合条件的文件对象集合。

(3)存储、操纵和利用作为查询结果的文件对象集合进行文件的存取。

3 实现的主要挑战

尽管基于属性存取的机制有明显的优点，但是以下实现的主要挑战在很大程度上妨碍了该特性的普及和商用化：

(1)性能和开销。和其他的文件存取机制不同，基于属性的存取需要有效地保存大量的与属性相关的元数据(Meta data)，并支持对其的动态操作。

传统的 Unix 操作系统为所有的文件对象保留一组包括文件的属主、创建时间、最近访问时间等在内的基本属性。而出于实现某些安全操作系统特性如存取控制列表(ACL)的需要，现代操作系统如 Linux 还支持用户自行定义的扩展属性(Extended Attributes)[9]。在当前的相关实现中，不论是基本属性或扩展属性，都是以如图 1 的文件-属性映射的逻辑结构组织的：文件的属性必须通过相应的文件结构进行存取。以 Linux 的 Ext3 文件系统为例，文件的基本属性直接存放在文件的控制结构 i 节点中，而文件的扩展属性存放在由其 i 节点指向的磁盘块中。

文件-属性映射是一种很方便于进行基于文件的属性操作(定义/撤消文件的属性，或指定/改变的文件属性值等)的信息结构，但对于基于属性的文件查询是非常不方便的。类似“属性 A 的值为 B 的所有文件”的简单查询，需要遍历在查询范围内的所有文件才能完成；这在性能上是根本无法接受的。任何以实用为目标的系统，必须提供解决此类问题的有效方案。

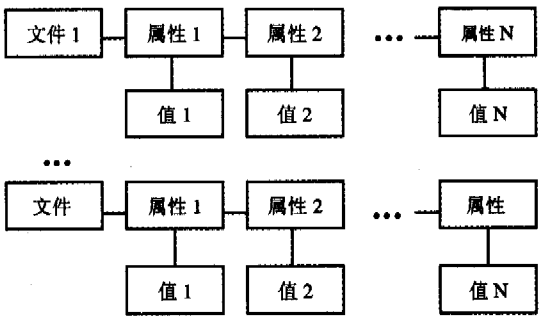

图 1 文件-属性映射

(2)与现有系统的兼容性。基于属性的存取和传统的基于路径名的存取从原理到操作方式都存在很大的不同，完全的非兼容式跃迁无法为现有系统庞大的用户群体所接受。因此实现者必须考虑实现与现有系统的完全或部分兼容。

应当指出，实现与现有系统的兼容实际上是一件非常困难的工作。其范围既包括保证传统功能相关操作的正确性，也包括使新增的操作能以用户熟悉的方式提供；后者尤其重要。现有操作系统业已提供了对文件属性的基本操作(如 Linux 中的 getfattr/setfattr)，但没有基于用户定义的属性进行的文件查询和其它的高级功能。能否为这些功能提供方便用户掌握的操作形式，很大程度上决定了实现的可用性。

(3)实现的非侵入性。由于基于属性的存取目标在于实现一种有效的文件管理机制，因此一般须通过对操作系统中基本文件系统部分(如 Unix/Linux 系统中 VFS)的增强来实现。而主流操作系统的开发者由于担心对现有机制的可用性与性能的潜在影响，往往对涉及到其基本部分的改动持非常消极的态度。因此能否以独立的模块，而非对基本文件系统的直接修改的形式实现基于属性的存取，直接决定了相关实现能否被主流操作系统所接纳。

综合以上因素，到目前为止，在基于属性的存取的众多实现中，仅有 Apple 公司在其全封闭架构、以功能新奇著称的 Mac OS 中包括的“Spotlight”取得有限的商业成功，实有其必然性。

4 基于 Linux 的设计与实现

在本节，我们将描述一种新的基于 Linux 系统的面向基于属性存取的文件系统。该系统基于 Linux 2.6.13 内核实现，目的在于在提供参考实现的基础上，促成基于属性存取机制成为 Linux 文件系统的基本组成部分。在以下的各节中，我们将描述该实现在用户界面、元数据管理机制、与现有系统的互操作、性能优化等方面，如何通过应用相关原则成功应对在第 3 节描述的各种挑战。

4.1 用户界面

在我们的参考实现中，基本沿袭了在 Inamura 和 Moriai[10]中提到的用户界面，但用和 Linux 中现有的 EA 机制对应的 getfattr/setfattr 命令取代了原有的对属性的相应操作。

此用户界面中最独特的特性，也是主要促使我们选择它的原因，是其对基于属性查询的语义的表达。在支持该界面的文件系统中，每个目录下都有名为“…”的特殊目录；而所有在相应目录下进行基于属性查询的操作，都被映射为对“…”

目录下的某个其名称为查询条件表达式的子目录的操作。

以下为查询条件表达式的文法形式：

〈查询条件表达式〉→〈简单式〉|〈查询条件表达式〉〈简单式〉

〈简单式〉→〈属性〉〈演算子〉〈值〉'%'

〈演算子〉→':' | '~'

这种特殊的子目录实际上并不像普通的目录一样是真实存在的文件对象，而只是在逻辑上包含符合查询条件的所有文件。但用户可以对其实施常见的操作如"ls"、"cat"等获取查询结果，以及更高级的操作如建立对其的符号链接，和将其设为缺省的执行路径等。

为保持与现有的层次文件系统的完全兼容，基于属性的查询结果以文件的路径名的形式表达。这样从应用的角度，包含查询结果的特殊目录与普通的文件目录几乎没有区别。

同时为了方便用户管理在系统范围内定义的属性，支持此界面的文件系统在其根目录下会缺省地提供名为"…attrib"的特殊子目录。同样这个目录并不是真实的文件对象，对其进行的查询操作将返回所有在系统中定义的属性和值。

相对众多更为激进的设计，本界面的主旨在于尽量尊重现有用户群体的使用习惯。同时以实现者的角度来说，由于所有的操作可以用现有的基本命令实现来完成，支持该界面也不需要在操作系统一级增加任何特殊的系统调用，实现者可免于开发和维护新的工具。这使得本界面在使用性上有非常明显的优势。

4.2 元数据管理机制

对任何基于属性存取机制的实现来说，对文件属性等元数据的管理都是其核心功能。根据本文第 3 节的相关说明，传统的文件-属性映射的信息结构仅适合于对属性的存取操作；为有效地支持基于属性的文件查询，有必要引入其它形式的信息结构。

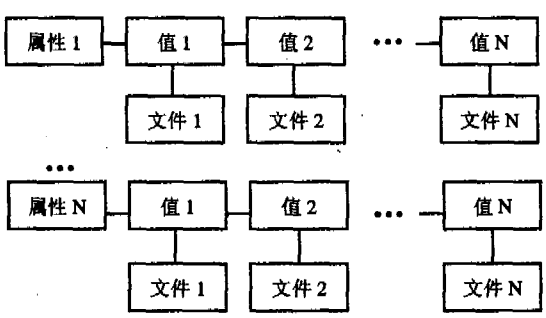

图 2 属性-文件映射

图 2 给出了在我们的参考实现中引入的新的属性-文件映射结构。

属性-文件映射本质上是以属性角度组织的满足特定的属性/值的文件信息，与对应的文件-属性映射等价且可相互转换。由于现有的文件系统的元数据均以文件为中心组织，属性-文件映射的信息必须被独立地存储。

在我们的当前的参考实现中，属性-文件映射被存储在每局部文件系统的根目录下名为". attrdb"的元数据数据库文件中。考虑到属性可被动态的定义/赋值/撤消，整个属性-文件映射被组织成以单个属性/值对为索引的 B+树。

在属性-文件映射中，怎样存储文件的标识信息是另外的一个问题。由于基于属性的文件查询的结果以路径名的形式表达，存储文件的路径名是最简单的选择。然而由于现代文件系统普遍支持长路径名，而多属性的文件信息将会被存储多次(每属性一次)，按路径名存储将会导致巨大的空间浪费。

作为替代的方案，存储文件的 i 节点号是更为可取的选择。现代文件系统的 i 节点号和路径名同样可作为对文件对象的唯一标识；i 节点号一般为 4 至 8 字节的整数，与动辄达数十至数百字节的路径名相比，在空间开销上优势明显。但存储 i 节点号也会带来新的性能问题：在进行基于属性的文件查询中，获得的 i 节点号必须转换为对应的路径名，但这并非易事。在层次化文件系统中，由文件的路径名到其对应的 i 节点的逻辑链接是完全单向的，完成从 i 节点到路径名的反向转换从理论上需要重新扫描整个目录树；由此带来的性能开销可以让属性-文件映射引入完全失去意义。

有鉴于此，我们引入了另外一种信息结构 i 节点-链接反向映射作为属性-文件映射的支撑机制。本质上这种结构是在". attrdb"文件中独立存储的、按所有文件的 i 节点号存放的文件对象和所有链接的路径名。由于 i 节点本身也是可重用的资源，i 节点-链接反向映射同样被组织成以 i 节点号为索引的 B+树。图 3 显示了 i 节点-链接反向映射的基本结构。

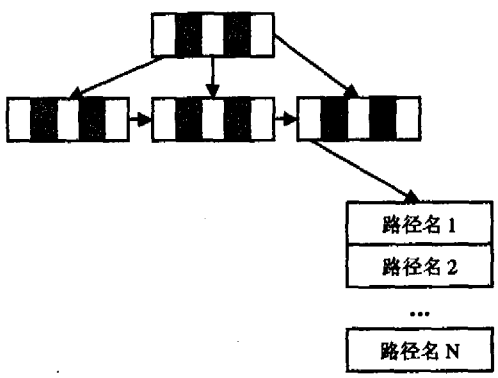

图 3 i 节点-链接反向映射

由于属性-文件映射和 i 节点-链接反向映射的引入，在元数据数据库(文件. attrdb)中需要存储大量动态分配/回收的变长对象，如在属性-文件映射的 B+树中作为查询索引的属性/值对，作为查询结果的 i 节点号，和在 i 节点-链接反向映射作为查询结果的路径名等。为有效处理此类对象，我们采用源自 BSD、被现代操作系统广为采用作为磁盘空间管理的块-组机制[11]来管理数据库内的空间：每类变长对象被存储在一系列大小固定的称为组(group)的连续区域内；组的偏移地址被保存在数据库文件的首部的组地址表内。每个对象组被进一步划分为定长的块(block)，块内有特殊的字段允许分散的块链接成逻辑的整体。块是基本的分配单位；根据待存储对象的大小，一次可分配一个或多个块。组内的块的分配信息以位图的形式存放在该组的起始位置。图 4 给出了在元数据数据库中采用的块-组机制的基本形式。

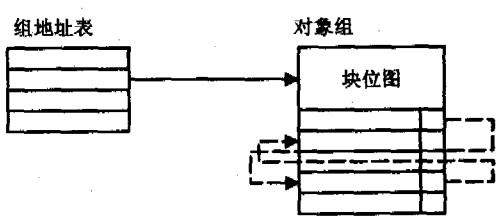

图 4 变长对象的块-组机制

4.3 与现有系统的互操作

在本文第3节中，我们已指出，实现的非侵入性对于其能否为主流操作系统所接纳具有非同一般的意义。由于采用4.1节描述的用户界面，我们的参考实现并不需要添加特殊的系统调用，决定实现的非侵入性的关键在于如何截获和正确处理被映射为对特殊目录的操作的基于属性的查询。

通常对文件系统进行非侵入式增强的有效方法是栈文件系统[12]，但Erez等人[13]指出，Linux的VFS与局部文件系统的强耦合使得很难对现有的Linux实现应用这种方法。基于这样的现实，我们采用了一种特别的办法来规避相应的障碍。

在我们的参考实现作为内核模块被插入时，将为每一种支持的局部文件系统类型注册相应的"影子类型"(如对应ext3的attrfs. ext3)；而用户在需要基于属性的存取机制时，在加载对应的磁盘分区时，应指定分区类型对应的影子类型。以下为当磁盘分区为ext3文件系统时，不激活和激活基于属性存取机制的加载命令形式：

mount -t ext3 〈设备名〉〈加载路径名〉

mount -t attrfs. ext3 〈设备名〉〈加载路径名〉

当采用后一种命令形式时，在现有的Linux实现中，影子类型的get_sb方法将被调用以注册该类型超级块/文件/i节点操作集合。对于以上操作集合的大部分方法来说，其实现会直接调用原类型的相应实现。但对于部分涉及到属性存取和基于属性的查询的操作，相应方法将调用参考实现中的相关方法进行处理。

通过以上方法，我们在不对现有Linux的实现进行直接修改的基础上，动态地在Linux VFS和局部文件系统之间以栈文件系统的方式建立了所需要的封装层，有效地保证了实现的非侵入性。

4.4 性能优化

和任何其他的文件系统一样，作为主要性能指标的相关操作的执行时间对于系统的可用性有非常重要的意义。

在4.2节中，我们已经描述了在数据库的组织中一些有利于提高性能的特性。但如果所有的操作都需要直接访问元数据数据库，大量的磁盘访问必然导致性能的降低。在我们的参考实现中，以下3种方法被用于减少访问数据库的开销：

(1)在加载时动态装入数据库的部分内容。

(2)创建和维护对象块存取缓存。

(3)创建和维护多条件缓存。

在我们的实现中，缺省在加载时被装入的数据库对象包括数据库文件的头部和各种对象组的地址表；同时根据用户在加载文件系统时指定的装入选项和系统内存的使用状况，部分或全部属性-文件映射的B+树，部分或全部i节点-链接反向映射的B+树，以及一个或多个对象组可以被装入内存。

为了加速对没有被装入内存的对象组的访问，我们的实现为每种对象提供了对象块存取缓存。这种缓存本质上是以块号为索引的哈希表，用以在内存中保存较近访问过的、但其所属对象组未被装入内存的对象块。

实际的基于属性的查询，其查询条件(属性/值对)往往超过一个；而且部分多条件查询的结果(如其对应的查询目录被链接或被设为执行路径)会频繁地被使用。为了节省这种查询导致的多次查询属性-文件映射和合并查询结果的开销，我们在属性-文件映射之上实现了另外一种多条件缓存，以固定项数组的形式保存较近访问的多条件查询结果。

以上机制已被证实对提高基于属性的查询的性能具有明显的效果。在我们建立的包含约20万个文件对象，和约1万个属性/值对的实验系统中，在使用约8M的内存用于保存数据库的内存拷贝和各种缓存目的的前提下，平均约32.7%的多条件查询和79.2%的对象块访问在内存内命中；而同一系统中对于返回同样结果的各种查询，基于属性的查询的平均速度约为基于文件名(采用find命令)查询的3.61倍，和基于内容查询(采用grep命令)的17.13倍。

5 进一步的工作方向

作为进一步的工作方向，我们期待当前的实现能在以下几个方面得到提高，以使基于属性的存取机制的实用意义能够得到更明显的体现，并最终促成其成为主流Linux发布或其它Unix类商业操作系统所吸纳。

(1)更好的对用户的透明性和语义一致性。当前的实现基于对局部文件系统的封装；虽然很好地保证了非侵入性，但对于多个局部文件系统被加载的情况，基于属性的存取操作无法以对用户透明的全局方式进行。这破坏了Unix/Linux系统的传统风格，有必要通过调整现有实现的体系结构加以改进。

(2)更友好的用户界面。当前实现的用户界面在保证与传统系统的兼容性方面有明显的优势，但在查询表达式的语法形式方面仍有改进的空间。另外为新增的操作开发图形用户界面(GUI)，有助于提高其用户友好度。

(3)进一步的性能优化。当前实现的元数据数据库以文件的形式存在，这在很大程度上是一种简化的设计选择。为进一步提升系统的性能，有必要对数据库的磁盘空间组织进行更精细的控制，开发与局部文件系统的磁盘管理机制耦合更紧密的数据库管理功能。

结束语 基于属性的存取是一种非常有用的文件管理机制，能在很大程度上弥补传统层次化文件系统的不足。但由于缺乏有实用价值的实现，这种机制尚未对现有计算模式形成革命性的影响。本文详细剖析了实现该机制所必须面临的各种挑战，并通过从各个方面描述我们专为此机制设计的一种文件系统，说明相关挑战如何通过组合应用一些既有的原理和独创的方法加以成功地解决。在对当前实现进行总结评价的基础上，我们将进一步完善我们的参考实现，以期促进该机制最终为主流商业操作系统所吸纳。

参 考 文 献

1 Maurice J B. The Design of the Unix Operating System. Prentice Hall, 1986

2 Stuart S, Michael M. Blending Hierarchical and Attribute-Based File Naming. In: Proc. of the 12th International Conference on Distributed Computer Systems, May 1992. 572～580

3 Dominic Giampaolo. Practical File System Design with the Be File System. Morgan Kaufmann, 1999

4 http://www.apple.com/macosx/features/spotlight/

5 Rosenblum M, Ousterhout J. The Design and Implementation of a Log-Structured Filesystem. ACM Transactions on Computer Systems, 1992, 10(1): 26～52

6 http://www.gnu.org/software/findutils/

7 http://www.gnu.org/software/grep/

8 http://desktop.google.com/

9 http://acl.bestbits.at/

10 Inamura H, Moriai S. Integration of Hierarchical and Attribute-base Naming schemes in a Distributed File Systems. In: Proc. of the 7th Joint Workshop on Computer Communications, 1992. 227～236

11 Marshall K M, Keith B, Michael J K, et al. The Design and Implementation of the 4.4 BSD Operating System. Addison-Wesley, 1996

12 Heidemann J S, Popek G J. File System Development with Stackable Layers. Transactions on Computing Systems, 1994, 12(1): 58～89

13 Erez Z, Rakesh I, Nikolai J, et al. On Incremental File System Development. To be appeared in: ACM Transactions on Storage, May 2006. http://www.filesystems.org/docs/zen/zen.html