Ext2 文件系统的设计与实现

Réy Card， Laboratoire MASI--Institut Blaise Pascal， E-Mail: card@masi。ibp。fr， and
Theodore Ts'o， Massachussets Institute of Technology， E-Mail: tytso@mit。edu， and
Stephen Tweedie， University of Edinburgh， E-Mail: sct@dcs。ed。ac。uk
for you

介绍

Linux 是一个运行在 PC-386 上的类 Unix 的操作系统，它首先作为对 Minix 操作系统[Tanenbaum 1987]的扩展实现，第一个版本仅支持 Minix 文件系统。Minix 文件系统存在着两个重大的局限:块地址储存在 16 位的整型数中，因此文件系统的大小被限制在 64MB。并且目录含有条目大小固定和文件名最长是 14 个字母。

我们设计并实现了两个新的文件系统，它们被包含在标准的 Linux 内核之中。这些文件系统被叫"extended file system"(Ext fs)和"second extended file system"(Ext2 fs)，它们提升了限制，并且增加了新的属性。

在这篇文章中，我们会描述 Linux 文件系统的历史。并简短的介绍 Unix 文件系统基本原理的实现。介绍 Linux 中虚拟文件系统层的实现并且详细的介绍 Ext2 文件系统核心代码和用户级工具。最后，我们给出在 Linux 和 BSD 文件系统中的性能测量，总结当前 Ext2fs 的状态和将来的发展方向。

Linux 文件系统的历史

早期，Linux 是在 Minix OS 下交叉开发而的。在两个系统间共享磁盘比设计一个新的文件系统要容易得多，因而 Linus Torvalds 决定在 Linux 中实现对 Minix 文件系统的支持。Minix 文件系统是一个有效的并且相对比较稳定的软体。

然而，在 Minix 文件系统的设计上有太多的限制，所以人们开始想在 Linux 上实现新的文件系统。

为了容易的在 Linux 内核中添加新的文件系统，一种虚拟文件系统被开发出来，VFS 最初是由 Chris Provenzano 编写，后来在它被融合到 Linux 内核之前又被 Linus Torvalds 重写了一遍。“虚拟文件系统”中会提到这些。

在 VFS 融合到内核之后，一个新的文件系统，"extended file system"在 1992 四月被实现，并且添加到 Linux 0.96c 中。这个新的文件系统摒除了 Minix 文件系统的两大限制:最大支持 2GB，最长文件名支持 255 个字符。它是在 Minix 文件系统上改进的，但一些问题仍旧存在。不支持:分离访问，节点修改，数据修改时间戳。文件系统使用链接表去保存空闲块和空闲节点的磁道，但这产生了严重性能问题:在文件系统使用时，列表会变得很乱，文件系统容易产生碎片。

对于这些问题，两个新的文件系统在 1993 年一月的 alpha 版本发布:Xia 文件系统和 Ext2 文件系统。Xia 文件系统几乎完全基于 Minix 文件系统核心代码，并且仅仅是在 Minix 文件系统上添加了一些新的改进。基本上，它支持长文件名，支持大的分区和支持 3 个时间戳。而另一方面，Ext2fs 则是在 Extfs 上进行大量的整合和改进。它的设计体现了一种思想上的进步，并且为未来的改进预留出了空间。更多的细节在"Ext2 文件系统"中。

当这两种文件系统首次发布时，它们提供了相似的主要功能。由于微型化的设计，Xia fs 比 Ext2 fs 稳定一些。因为被广泛使用，Ext2fs 中许多 bug 被修补，一些新的特性被整合时。现在 Ext2fs 已经非常稳定而且变为标准的 Linux 文件系统。

这张表显示了不同文件系统间的性质:

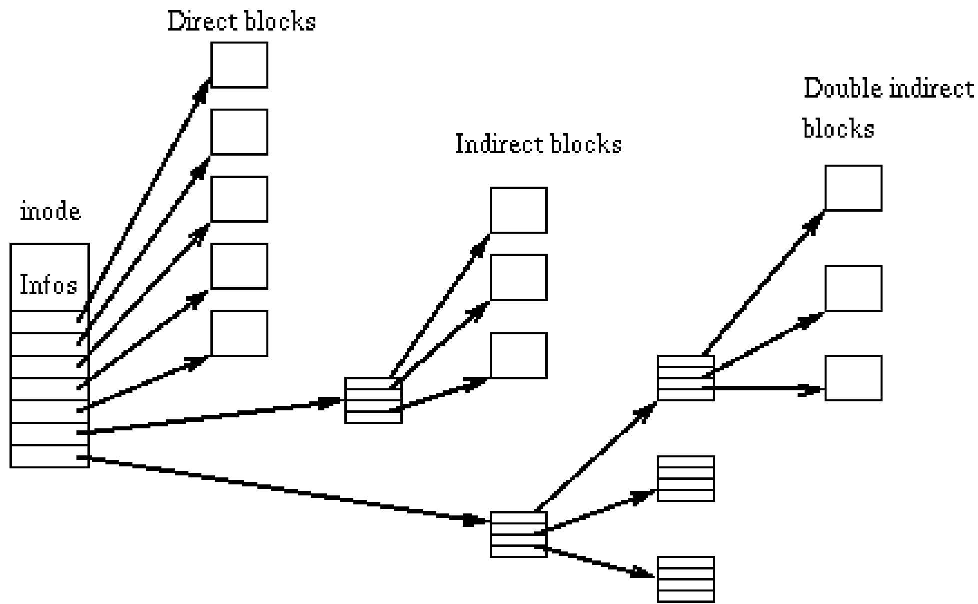
	Minix FS	Ext FS	Ext2 FS	Xia FS
Max FS size	64 MB	2 GB	4 TB	2GB
Max file size	64 MB	2 GB	2 GB	64MB
Max file name	16/30 c	255 c	255 c	248 c
3 times support	No	No	Yes	Yes
Extensible	NO	NO	Yes	NO
Var. block size	No	No	Yes	No
Maintained	Yes	No	Yes	?

基本文件系统概念

所有 Linux 文件系统的一套基本概念的实现是来自于 Unix 操作系统 [Bach 1986]:文件被节点代替，目录则是一个包含入口列表的简单文件，并且设备是能够通过 I/O 请求而访问的特殊文件。

节点

每一个文件被一种叫做节点的结构代替。每一个节点包含着文件的信息:文件类型，访问权限，所有者，时间戳，文件大小，和数据区块指针。分配给文件的数据区块的地址储存在它的节点中。当一个用户在文件上请求一个 I/O 操作时，内核把当前的偏移转换到一个区块数字，使用这个数字作为区地址表的一个索引并且对物理区块进行读或写操作。下面这幅图体现了一个节点的结构。

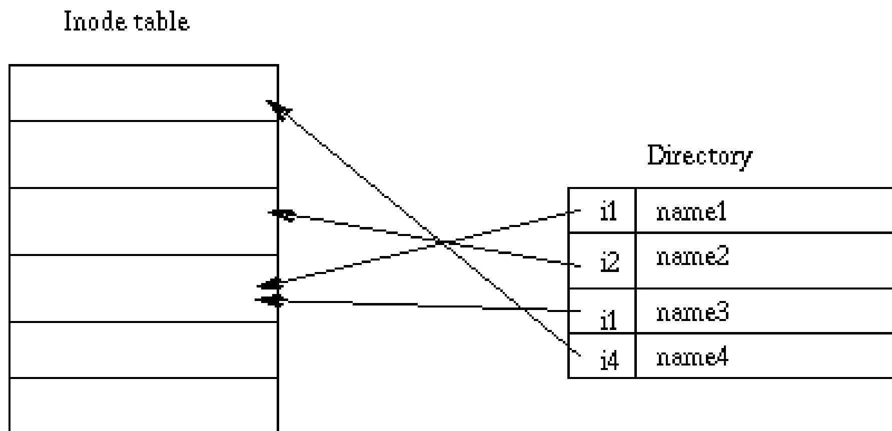


目录

目录建立在多叉树上。目录可以包含文件和子目录。

目录是作为一种特殊类型的文件而实现，实际上，目录就是一个包含入口列表文件。每一个入口包含一个节点数和一个文件名。当一个进程用到路径名时，内核在目录中搜索并找到符合条件的节点号。当路径名被转换成一个节号后，该节点便被载入内存并且用于随后的请求。

这幅图体现了目录结构:



链接

Unix 文件系统提供链接的概念，多个名字可以被一个节点所关联。节点包含了一片含有被联接的文件数的区域。增加一个链接仅在于创建一个目录项目，这个目录项目的节点数指向节点，并且在节点上不断递增链接数。当删除一个节点，比如，用 rm 命令删除一个文件名(filename)，内核会减少链接数，并且如果这个数目变成零的话，会重新定位节点。

这种链接类型叫做硬链接并且只能用于单一的文件系统内:不能在不同的文件系统间创建硬链接。此外，硬链接只能指向文件:目录硬链接不可以被创建是为了阻止在同一个目录树中出现死循环。

另一种链接类型存在于大部分 Unix 文件系统中。符号链接是一种包含文件名的简单文件。当内核在从路径名到节点转换过程中读取一个符号链接时，它会用符号链接的内容代替符号连接名，举个例子，就像重新启动路径名解读目标文件的名字。由于符号链接并不是指向节点，所以可以在交叉文件系统中创建符号链接。符号链接可以指向任何文件，甚至可以是不存在文件。符号链接是非常有用的，因为它们并没有像硬链接的那些限制。但无论如何，它们都会使用一些硬盘空间，因为要分配给它们节点和数据区块，并且造成经常性的路径名到节点的转换-因为内核需要重启名称解读-当它遇到一个符号链接时。

设备文件

类 Unix 系统中，设备可以通过特殊的文件访问。一个设备文件并不使用文件系统上的任何空间。它仅仅是使一个设备驱动器的入口。

特殊文件有两种：字符和字块特殊文件。以前允许在字符特殊模式中的 I/O 操作，直到稍后的在字块模式中通过缓冲器缓存写入数据的请求。当 I/O 请求作用在一个特殊文件时，它首先转发给一个(虚拟)的设备驱动器。专署文件通过一个主参考数字来确定设备类型，通过一个辅助数字来确定设备(unit)。

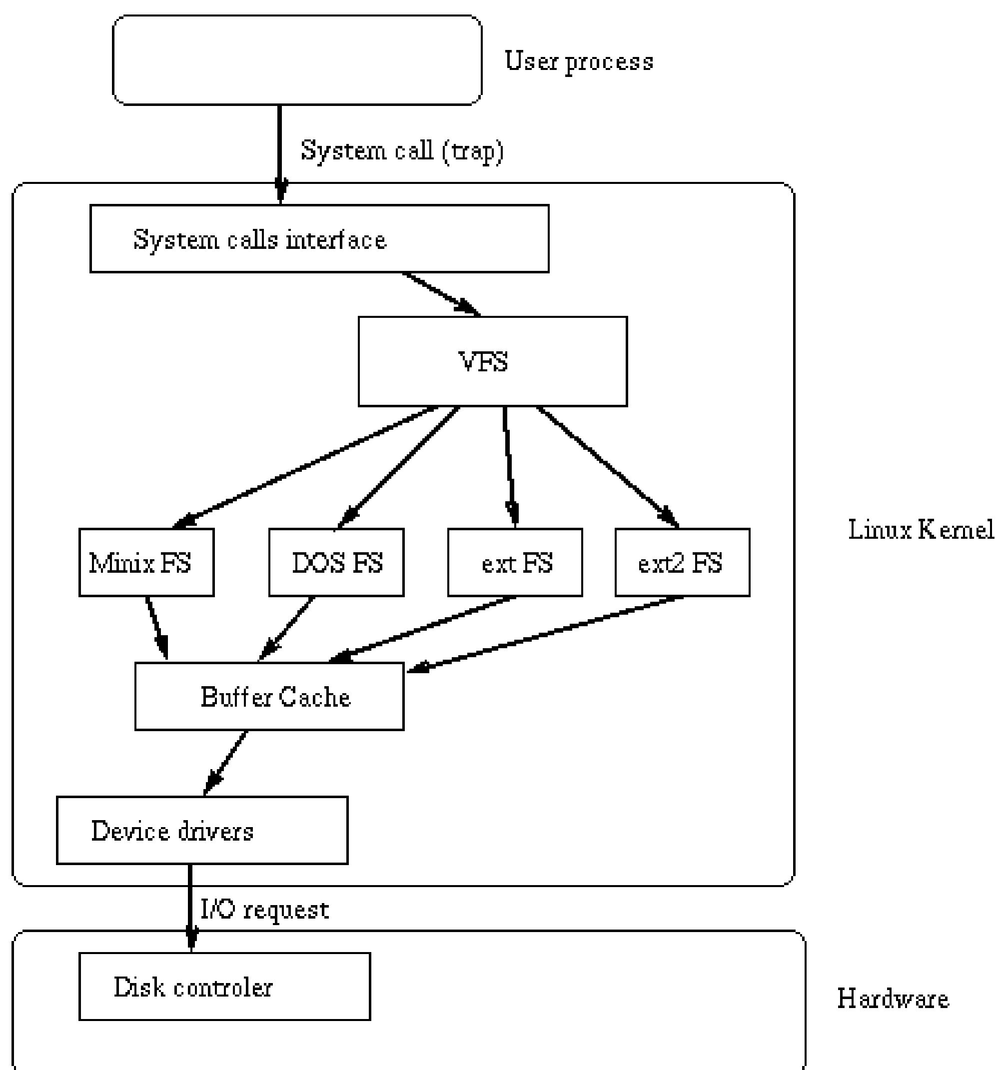
虚拟文件系统

原理

Linux 内核包含了一个虚拟文件系统层次，用于文件上的系统调用。VFS 是系统调用间接对文件进行操作，和为完成 I/O 而在物理文件系统上调用必须的操作编码。

这种间接的机制经常在类 Unix 系统中用于调整兼容与使用多种文件系统。 [Kleiman 1986， Seltzer et al. 1993].

当一个进程发起一个面向文件的系统调用时，内核会从 VFS 中调用一个函数。这个函数会建立一个独立的处理操作，并且重定向去调用一个包含在物理文件系统编码里的功能项，它负责处理和建立相关的操作。文件系统编码利用缓冲器缓存功能在设备上实现 I/O 请求。这个流程在图中显示。



VFS 结构

VFS 定义了一套所有文件系统需要实现的接口函数，这种接口制定了一套关于三种目标类型的操作:文件系统，节点，还有打开文件。

VFS 知道内核支持的文件系统。它在内核配置中使用定义列表。每一个项目在列表中描述了一种文件系统类型:它包含了文件系统类型的名称还有当挂载操作时的一个函数调用的指针。文件系统被挂载时，相应的挂载函数被调用。这个函数负责从硬盘上面读取 superblock，初始内部变量，在 VFS 上返回一个文件系统描述符。当文件系统被挂载后，VFS 会利用这个操作符去访问物理文件系统例程。

一个文件系统挂载操作符包含一些数据类型:大体上所有文件系统的信息，指向由物理文件系统内核代码函数的指针，由物理文件系统内核代码维护的私有数据。包含在文件系统描述符的功能指针允许 VFS 访问文件系统内部例程。

另两种被 VFS 用到的描述符类型是:节点描述符和打开文件描述符。每一种描述符都包含

了使用文件的信息和由物理文件系统编码提供的一套操作。节点描述符包含在任何文件上进行操作的指针，而文件描述符仅包含打开文件操作的指针。

Ext2 文件系统

动机

Ext2 文件系统的设计与实现是为了修正存在于 ext 文件系统上的一些问题。我们的目标是提供一个强大的文件系统，并且实现 Unix 文件的结构(semantics)和提供高级的特性。

当然，我们想让 Ext2 文件系统拥有更加强大的性能。我们同样想提供在强化使用上为了减少数据丢失的稳定的文件系统。最后，但也是最重要的，Ext2 文件系统包含了允许用户不用重新格式化文件系统而从新的性能上受益的扩展的准备。

"标准"的 Ext2 文件系统特征

etx2fs 支持标准的 Unix 文件系统类型:正规文件，目录，设备特殊文件和符号链接。

Ext2fs 可以在超大分区上管理文件系统的创建，虽然最初的内核代码限制最大文件系统到 2gb，但当前 VFS 层把限制提高到 4TB。这样，我们现在可以使用大的硬盘而不用格式化出许多的分区。

Ext2fs 提供长文件名。它使用可变长度的目录项。最大的文件名可达 255 个字符。如果需要的话，这个限制可以扩展到 1012。

Ext2fs 为超级用户(root)保留了一些 block(区块)。通常，5%的区块是保留的。这样，即使用户的程序将文件系统填满，管理员也可以很容易的从这种状况下恢复回来。

Ext2fs 的"高级"特性

除了 Unix 的那些性质，Ext2fs 也支持一些通常在 Unix 文件系统内不存在的扩展。

文件属性允许用户去改变内核对文件操作的行为。你可以在文件或目录上面设置文件属性。而且，当新的文件在这个目录创建时，目录的属性也继承给了这个文件。

BSD 或者 System V Release4 的语法习惯在挂载时是可以选择的。挂载操作允许管理者选择文件创建方式。以 BSD 的习惯，当一个文件系统被挂载时，文件文件创建的 id 是和它们上层文件夹的 group id 相同的。而 System V 的方式却稍显复杂:如果一个目录有 setgid bit 设置，新的文件则继承目录组 id，并且子目录继承组 id 和 setgid bit;另一方面，文件和子目录是依据调用程序的主组创建的。

类 BSD 系统可以同步更新使用 Ext2fs。挂载操作允许管理员在元数据(节点，位图块，间接访问块，直接访问块)改变时向磁盘请求同步写入。这样可以有效地去维持一个元数据

的稳定性，但性能上却有所下降。实际上，这种特性并不是被广泛的使用，除了在使用元数据同步更新而产生的性能下降外，它造成的用户数据错误会使文件系统 checker 变得不容易标示。

Ext2fs 允许管理者在创建文件系统时选择逻辑区块的大小。区块大小可以分为 1024，2048 和 4096bytes。使用大的区块能够在访问文件时提升 I/O 请求和寻道的速度。但，大区块会浪费很多磁盘空间:平均下来，一个文件分配给最后一个 block 仅占了一半空间。如果 block 很大，那么很多空间就浪费在每一个文件最后的那个 block 里面。但是，大区块文件优势被 Ext2 文件系统的预分配技术所体现。 见"性能优化"章节

Ext2fs 实现了快速的符号链接。快速符号链接并不是可以用到文件系统的任何文件上。目标文件名不能储存在数据区块中而只能在节点本身。这种方式可以节省一部分空间(没有数据区块需要分配)并且可以提升链接操作的速度(因为当访问一个链接是没有必要去读取数据区块的内容)。当然，在一个节点内可用空间的限制使得并不是每一个链接都能实现快速链接。在一个快速符号连接里，目标文件名最长可以达到 60 个字符。我们打算在不久的将来把这种安排扩充到小型文件上。

Ext2fs 保持辊奶踪文件系统状态。在 superblock 上有一块特殊的区域被用于内核指明文件系统的状态。当文件系统被挂载为读/写的模式时，它的状态被设置为"not clean"。当文件系统被卸载或者重新挂载为只读属性时，它的状态被重新设置到"clean"。启动时，文件系统 checker 利用这里的信息去决定文件系统是否需要检查。内核也同样把错误记录在这片区域内。当内核探测到文件系统出现非一致性时，文件系统被标识为"erroneous"。文件系统 checker 对文件系统进行测试和强行的检查并且不论文件系统的状态是否显示为"clean"。

总是跳过文件系统检查在有些时候也许会是不安全的，所以 Ext2fs 提供了在每隔一段时间进行两种强行检查的方式。一种是挂载计数器被驻留在 superblock 里。每一次文件系统被挂载为 read/write 模式时，这个计数器便会自加一次。当达到最大值时(同样在 superblock 里记载)，文件系统 checker 便会强行的对文件系统进行检查--甚至文件系统是"clean"的。最后一次检查时间和全面检查的间隔同样也记录在 superblock 里面，这两个区域允许管理者进行定期检查的请求。当全面检查间隔被发现，checker 便护绿文件系统状态并且对文件系统进行强行的检查。Ext2fs 提供了调试文件系统状态的工具。tunne2fs 程序可以用于修改:

错误行为。当内核探测到非一致性，文件系统被标识为"erroneous"并且下面三种行为中的一种将要被实施:继续的执行，重新挂载文件系统为只读模式去避开损坏的文件系统，进入内核应急模式并且重新启动计算机去运行文件系统检查器。
最大挂载数。
全面检查间隔。
为超级用户所保留的逻辑区块数。

挂载操作同样用于改变内核错误行为。

文件属性允许用户请求对文件的安全删除。当一个文件被删除，一些随机的数据会写入到从前分配给这个文件的空间内。这样就可以防止一些"有心人"利用一些硬盘编辑器从从前的文件中获取信息。

最后，一些从 4.4BSD 文件系统中受启发而形成的性质被加入到当前的 Ext2fs 中。immutable 文件。这种文件只可以进行读操作:任何人不可一写入或者删除。这可以用于保护敏感的配置文件。append-only 文件。它们可以在写入模式下打开但数据往往是附加到文件的末尾。就像 immutable 文件，它们不能被删除或者重命名。这种特殊的形式可以用于 log 文件，因为它们只能被生成。

物理结构

Ext2fs 文件系统的物理结构很大程度上受到了 BSD 文件系统 [McKusick et al。 1984]布局的影响。文件系统是由 block 组构成。Block 块组类似于 BSD 的 FFS 磁柱群。然而，在磁盘上区块组并不是同区块物理布局连接在一起，因为现代的驱动器倾向于顺序存取法的优化而向操作系统隐藏它们的物理结构排列方式。

一个文件系统物理结构的例子:

Boot	Block	Block	...	Block
Sector	Group 1	Group 2	...	Group N

每一个 block 组都包含着一份系统文件关键控制信息的冗余(superblock 和文件系统描述符)和同样包含了文件系统的一部分(block 位图，节点位图，和一张节点列表，和数据区块)。区块组结构的一个例子: Super

Super Block	FS descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

使用 block 组就可靠性而言是一个很大的成功:因为控制结构被复制到每一个 block 组中，它能够很容易的从一个 superblock 损坏的文件系统中恢复。这种结构同样也有助于优化文件系统:这种结构可以减少数据区块和节点列表之间的距离，它也使得文件在输入输出过程中的磁头寻道频率减少成为可能。

Ext2 文件系统中，目录被作为一种可变长度项的链接列表所管理。每个项目包含着节点数，项目长度，文件名和它的长度。之所以使用可变长度项目，是因为它在目录中可以使用长的文件名而不必浪费更多的磁盘空间。下面这张表给出了目录项的结构:

Inode number	Entry length	Name length	filename

举个例子，下面这张表就体现了一个包含三个文件的目录结构:

	16	05	File1
	40	14	long_file_name
	12	02	f2

性能优化

Ext2fs 的核心代码包含了许多的性能优化，它趋向于在读写文件时提升 I/O 的速度。

Ext2fs 利用执行预读对缓冲器缓存管理:当读到一个 block，内核就在周边几个 block 进行 I/O 请求。这种方式尝试确保将要读取的下一个 block 载入到缓冲器缓存。预读普遍作用在顺序读取的文件中并且 Ext2fs 把它们扩展到直接读取，同样的也可以 explicit read(readdir(2) calls)或者 implicit 读取(namei 内核目录)

Ext2fs 同时也包含着许多分配优化方式。block 组用于对相关节点和数据结合在一起:内核通常在相同的组中为一个文件分配数据区块作为它的节点。这种做法可以让内核在读取一个节点和它的数据区块的时候减少硬盘寻道操作。

当写入数据到文件时，在分配一个新的区块时 Ext2fs 会同时预分配它附近的 8 个区块。预分配在一个完整的文件系统上平均使用率大概 75%。在重负载的情况下预处理仍旧可以获得很好的写入性能。它同样允许周围的区块分配给文件，这样的话，在将来的顺序读取情况下也会有一个速度上的提升。

下面两种分配优化有很好的局部化效应:

  文件通过 block 组关联。
  区块通过区块分配的 8bit 簇关联。

Ext2 文件系统库

为了允许用户模式程序去操作 Ext2 文件系统的控制结构，libExt2fs 库因此而生。Ext2 库提供了允许用户程序利用它去对 Ext2 文件系统的数据进行检验和修改的例程，可以利用它通过物理驱动器直接对文件系统进行访问。

Ext2 文件系统的设计通过软件抽象技术使得大量的代码可以重用。举个例子，提供了一些不同机制的迭代器。程序可以直接向 Ext2fs_block_interate()中传递一个函数。使用这个函数可以在节点中调用任何一个区块。另一个迭代器允许用户提供的函数去调用目录中的每一个文件。

Ext2fs 中的许多程序(mke2fs，e2fsck，tune2fs，dumpe2fs，和 debugfs)都使用 Ext2fs 库。这

极大的简化了这些程序的维护。如果文件系统格式有什么新的性质那么仅仅需要在一个地方产生改变--那就是 Ext2 文件系统库。由于 Ext2 文件系统的库可以作为一个共享库镜像构建，所以它编译产生的是较小的二进制文件。

因为 Ext2fs 库的接口是比较灵活和通用的，所以新的程序要想直接做到访问 Ext2fs 文件系统是比较容易实现的。举个例子，Ext2fs 库被用作 4.4BSD 端口的转储和恢复备份程序中。如果把他们应用到 Linux 中的话只需要很小的改变:只有一小部分的文件系统依赖函数需要到 Ext2fs 库中替换调用。

Ext2fs 库提供了几类访问操作。第一类是文件系统定向操作。一个程序可以打开和关闭一个文件系统，读写位图，并且在磁盘上面创建一个新的文件系统。这个功能同样可以操作文件系统的坏区块列表。

第二种是伪目录操作。Ext2fs 库调用器可创建和扩展一个目录，当然也可以删除或者增加一个目录项。这个函数同样提供了分析路径名到节点数字，并且确定节点的路径名并给出他的节点数。

最后一类是围绕节点的操作。它可以扫描节点列表，读写节点，并且在一个节点中通过所有的 block 扫描。可以分配和收回例程，并且允许用户模式程序去分配和释放区块和节点。

Ext2fs 工具

Ext2fs 中有一些强大的管理工具。这些工具被用于创建，修改，和修正 Ext2 文件系统中的非一致性问题。mke2fs 就是一个用于初始化分区到一个空的 Ext2 文件系统的工具。

tune2fs 用于改变文件系统参数。在 Ext2 文件系统的"高级"特性章节有相关的介绍。 它可以改变错误行为，最大挂载数，全面检查间隔，和为超级用户保留的区块数。

最有趣的工具可能就是文件系统检查器了。e2fsck 是用于修复文件系统在系统非正常关机的情况下而产生的非一致性问题。最初的版本是建立在 LinusTorvald 为 Minix 文件系统所写的检查程序上的。然而，当前的 e2fsck 程序是 Scratch 重新构造的，利用了 Ext2fs 库，并且它可以非常迅速的纠正文件系统的非一致性问题。

e2fsck 的运行被设计的尽可能的迅速。由于文件系统检查器受到了磁盘的限制，e2fsck 利用算法的优化而完成，所以文件系统并不是从磁盘反复访问的。另一方面，这种方式可以按区块数排列来检查节点和目录，减少磁盘寻道时间。许多的方式最初都是由 [Bina and Emrath 1989] 开发的，虽然在那之后还有其他人对此进行过改进。

第一步，e2fsck 把所有文件系统内的节点迭代(iterates)，并且把文件系统内的节点看作非链接对象进行检查。因此，不需要对其它的文件系统对象进行重复的检查。这种检查确定文

件模式是合法的，并且确定节点内的所有区块是拥有有效的区块数。在第一步里，位图确定了那些区块和节点是经过统编的。

如果 e2fsck 发现数据区块被多个节点所关联，那么他便通过 1D 调用 1B 步骤去解决这些问题:把共享的的区块拷贝使得每一个节点都能够拥有一份共享节点的复本，或者，解除所有节点的分配。

第一步是耗时最长的步骤，因为要把所有的节点读入内存，并且进行检查。今后的步骤里减少 I/O 时间是必然的，所以将关键的系统文件信息都缓存于内存。最终要的技术实例便是在文件系统上面定位硬盘上所有的目录区块。这样可以直接获得信息而派出了在第二步期间的重读目录节点结构。

第二步把目录作为无链接对象进行检查。由于目录项并非整体的磁盘区块，每一个目录项区块必须在排除其它目录项的基础上进行单独的检查。这允许 e2fsck 利用区块数去排列所有的目录区块，并且以增序列进行检查，因此可以减少磁盘寻道时间。目录项的检查是确保目录项是有效的，并且包含了对使用的节点数的参考。(作为对步骤 1 的确定)

每一个目录节点中的第一个目录区块是"."和".."，他们是必须被确认存在的，并且"."项目就是指代当前的文件夹。(".."的节点数是在第三步中检查的)

第二步同样也把各个目录之间的链接情况缓存到内存中。(如果一个目录被多个目录所引用，而且其中有一个目录是一个不合法的硬链接，那么这个硬链接就会被删除)。

值得注意的应该是第二步的最后部分，几乎所有的磁盘 I/O 都需要完全去执行。文件系统信息在第三四五步被载入内存;因此，剩下的的步骤很大程度上受到了 CPU 的限制，这大概减少 e2fsck 运行总时间的 5-10%。

第三步检查链接的目录。e2fsck 会沿着每一个目录进行检查，最后到达根目录，使用在第二步载入内存的信息。在这次检查中，"。。"项也同样检查并且确认是有效的。所有不能退回根目录的目录都会被链接到/lost+found 目录中。

第四步，e2fsck 检查所有节点的引用数，迭代(interting)所有的节点并且链接数(在第一步载入内存的)和在第二三步载入内存的内内部记数器进行比较。在这一步中，将会把所有空链接投入到/lost+found 里。

最后，在第五步，e2fsck 文件系统汇总信息的有效值。他在文件系统上把之前所有区块、节点位图的信息和实际的位图进行比较，必要的话会纠正磁盘上的复本。

文件系统调试器是另一个很有用的工具。debugfs 是一个强大的工具，它用来测试和改变文件系统的状态信息。基本上，它提供了与 Ext2fs 库的交互式接口:用户输入的命令会被处理到库例程里进行调用。

debugfs 可以用于测试文件系统内部结构。手工修复一个损坏的文件系统，或者为 e2fsck 穿件一个检验实例。不幸的是，此程序对一个不知到自己在做什么的人是比较危险的:他能够非常容易的将文件系统毁掉。处于这个原因，debugfs 在默认的情况下是以只读模式打开文件系统的。用户必须指定-w 选项才能够使 debugfs 已读/写模式打开文件系统。

性能测试

标准的描述

我们做出了一个基准去测试文件系统的性能。这些基准已经在建立在一个 middle-end 电脑上的，基于 i486DX2 处理器，16mb 内存和 420mb ide 硬盘。这些测试是在一步和同步模式下的 Ext2fs、xiafs(Linux1.1.62)和 bsd 快速文件系统进行的(freebsd2。0 alpha-based 4。4bsd 微发行版)。

我们有两套不同的基准。bonnie 基准是在测试大文件的 I/O 速度--文件尺寸 60mb。他使用字符模式进行数据写入，重写整个文件的内容，利用区块模式写入数据，用字符和区块模式读文件，并且进行文件内部定位。andrew 基准是在 carneggie mellon 大学开发的，并且被 berkeley 大学进行对 bsd ffs 和 lfs 测试。他进行五个方面的测试:创建一个目录阶(directory hierarchy)，制作一份文件复本，递归的检查每个文件的状态，检查每一个文件的每一个数据，和整合几个文件。

Bonnie 标准的测试结果

Bonnie 标准的测试结果:

	Char Write (KB/s)	Block Write (KB/s)	Rewrite (KB/s)	Char Read (KB/s)	Block Read (KB/s)
BSD Async	710	684	401	721	888
BSD Sync	699	677	400	710	878
Ext2 fs	452	1237	536	397	1033
Xia fs	440	704	380	366	895

面向区块 I/O 的结果是很不错的:Ext2fs 高于其它文件系统。这显然是分配例程优化的优势。写速度快是因为数据是在簇模式下写入的。读速度快是因为区块的预分配造成的。在两次读操作中磁头并没有寻道，并且预读优化完全发挥了作用。

另一方面，FreeBSD 操作系统中的面向字符 I/O 的性能很不错。这可能是 FreeBSD 和 Linux 在他们各自 c 库中并没有使用相同的基本输入输出例程。看来 FreeBSD 有一个更好的字符 I/O 优化库。

Andrew 标准测试结果

	P1 Create (ms)	P2 Copy (ms)	P3 Stat (ms)	P4 Grep (ms)	P5 Compile (ms)
BSD Async	2203	7391	6319	17466	75314

BSD Sync	2330	7732	6317	17499	75681
Ext2 fs	790	4791	7235	11685	63210
Xia fs	934	5402	8400	12912	66997

测试的前两个结果可以看到 Linux 使用异步元数据(metadata)I/O 的好处。在第一步和第二步中，创建目录和文件，并且 BSD 同步写入节点和目录项。这是一种异常现象，尽管:BSD 的性能在异步传输上很差。我们怀疑 FreeBSD 下的异步传输支持并没有完全实现

第三步中，Linux 和 BSD 中所用的时间差不多。这同 6 个月前相比已经有一个很大的进步了。尽管 BSD 做的比 Linux 更好一些，但在 VFS 中添加文件名到缓存中会修复这个问题。

第四和第五步中，Linux 快过 FreeBSD，主要是因为他使用统一的缓冲器管理缓存。缓冲器缓存空间可以按需增长。而 FreeBSD 是使用固定大小的缓冲器缓存。Ext2fs 和 xiafs 的比较结果可以看到，包含在 Ext2fs 中的优化真的是很有用:性能上大概提升 5-10%。

总结

Ext2 文件系统可能已经是 Linux 使用最广泛的文件系统了。它提供了标准 Unix 文件系统的功能和高级属性。此外，由于内核中包含了拿破仑，使得它具有了更强大和优秀的性能。

因为 Ext2fs 是在一种先进的思想上开发而生的，所以他容易使用和容易添加新的属性。一些人已经在为当前的文件系统扩展而工作:兼容存取控制列表到 Posix 格式。 [IEEE 1992] 拒绝删除(undelete)，运行中文件压缩(on-the-fly file)。

Ext2fs 是第一个整合到 Linux 内核中的文件系统。并且，现在 Ext2fs 已经航向其他的操作系统中。一个 Ext2fs 的服务程序已经在 GNU hurd 中实现。人们同样把它移植到 LITES 服务中，运行在 Mach 微内核上。 [Accetta et al。 1986]同时也在 VSTa 系统上。最后，同样也是最重要的，Ext2fs 也是 masix 系统中重要的一部分， [Card et al。 1993]现在正处在作者的开发下。

致谢

Ext2fs 内核和工具大多数是这篇文档的作者所写。一些人也同样的为 Ext2fs 的开发作者贡献，或者是提议一种新的功能又或者发布一些 bug 的补丁。我们真心的感谢他们所做的贡献。

引用

[Accetta et al. 1986] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In Proceedings of the USENIX 1986 Summer Conference, June 1986.

[Bach 1986] M. Bach. The Design of the UNIX Operating System. Prentice Hall, 1986.

[Bina and Emrath 1989] E. Bina and P. Emrath. A Faster fsck for BSD Unix. In Proceedings of the USENIX Winter Conference, January 1989.

[Card et al. 1993] R. Card, E. Commelin, S. Dayras, and F. M 関 el. The MASIX Multi-Server Operating System. In OSF Workshop on Microkernel Technology for Distributed Systems, June 1993.

[IEEE 1992] SECURITY INTERFACE for the Portable Operating System Interface for Computer Environments - Draft 13. Institute of Electrical and Electronics Engineers, Inc, 1992.

[Kleiman 1986] S. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In Proceedings of the Summer USENIX Conference, pages 260--269, June 1986.

[McKusick et al. 1984] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. ACM Transactions on Computer Systems, 2(3):181--197, August 1984.

[Seltzer et al. 1993] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In Proceedings of the USENIX Winter Conference, January 1993.

[Tanenbaum 1987] A. Tanenbaum. Operating Systems: Design and Implementation. Prentice Hall, 1987.