|  |
| --- |
|  |
| 海量存储技术 |
| Using Hints to Improve Inline Block-Layer Deduplication |

|  |
| --- |
| 刘瀚 2017282110235  2017/12/8 |

Using Hints to Improve Inline Block-Layer Deduplication

Sonam Mandal,1 Geoff Kuenning,3 Dongju Ok,1 Varun Shastry,1 Philip Shilane,4 Sun Zhen,1,5

Vasily Tarasov,2 and Erez Zadok1

**摘要**—块设备层的重复数据删除使得文件系统和应用程序在不用预先修改的情况下获得重复数据删除的好处。然而，数据上下文相关的重要信息会在块设备层丢失。把这种上下文传给块设备层可以提升重复数据删除的表现和可靠性。我们在一个开源块设备层重复数据删除系统dmdedup中部署了一个提示接口，可以给块层传递相关内容，并评估两个提示NODEDUP和PREFETCH。为了使上层存储能够传递基于可用内容的提示，我们修改了VFS和文件系统层，让提示接口暴露给用户程序。我们说明了传递NODEDUP提示可以给应用加速5.3倍，因为当重复数据删除没有好处的时候避免了它的开销。同时也展示了通过为即将被访问的数据缓存哈希，PREFETCH提示加速应用1.8。

**1 简介**

一些组织存储的数据量正在急剧增加。硬盘和固态硬盘价格的下降也无法弥补这种增长；其结果就是公司花在存储上的费用越来越高了。一种减少存储消耗的技术就是重复数据删除，这种技术使得站点可以存储更少的原始数据。重复数据删除的核心就是根据参照系统地删除那些重复的数据块。对于很多真实世界的数据集，重复数据删除大大减少了原始数据的存储。

重复数据删除可以被应用在存储堆栈的几个层次中。现存最多的方案是将其内置在文件系统中，因为文件系统中有足够多的信息能够在不破坏可靠性的前提下高效地进行重复数据删除。比如文件大小、元数据和磁盘布局这些对于文件系统来说都是已知的；通常来说文件系统是知道执行I/O操作的用户和过程的。这个信息可以被用来避免特定数据块的删除（比如元数据），或者用来预取去重元数据（比如为那些可能会一起被访问的快）。

还有一种选择就是在块设备层进行去重，它提供了一个简单的读/写接口。由于这种简便性，在块设备层添加特征比改变文件系统要容易一些。这一观察同样适用于直接在块设备层工作的系统，如数据库和对象存储。

然而，一个块级的重复数据删除系统它所操作的数据的内容。一个典型的I/O请求只包含操作类型（读或写），大小和偏移量，并没有像元数据和用户数据之间差异这种附加的语义信息。对元数据的去重会伤害可靠性，例如，由于许多文件系统会特意保存像超级快这样的重要数据，也会浪费计算资源，因为典型的元数据（节点表，目录条目等）的冗余度很低。特别地，行内删重的开销很大，因为分块（定长或是边长），哈希的计算和查找都是爱数据写入磁盘之前执行的；我们不希望在那些重复数据删除无法带来好处的数据上花费资源。

为了使设备块层的重复数据删除将上下文信息考虑在内，我们提出了一个接口使得文件系统和应用能够提供关于删重执行位置上上下文的简单提示。这种提示只需要很细小的文件系统改动，使得将其添加到现有成熟文件系统上变得可行。我们使用两种提示：NODEDUP和PREFETCH，我们发现他们在广泛的案例中都很有用。

为了评估提示所带来的潜在好处，我们使用一个开源的设备块层重复数据删除系统—dmdedup。Dmdedup是主要用于行内主存储器的重复数据删除，被部署为一个Linux内核中的可堆叠块设备。我们在多种工作量和唯一数据混合VS重复数据之下评估我们的提示。我们的结果阐释了通过保留那些可能唯一的数据，NODEDUP提示可以将应用的速度相比于Dmdedup提高5.3倍之多，PREFETCH可以加速超不多1.8倍。

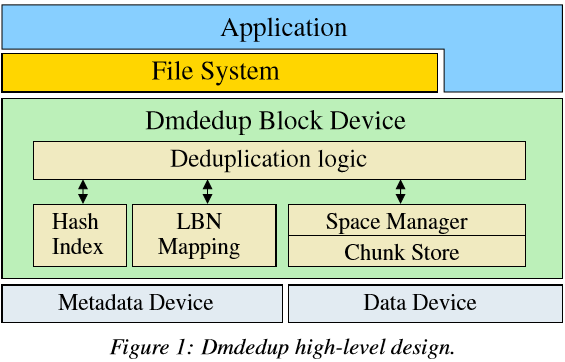
**2 背景**

**上下文恢复。**此前的研究已经说明了块设备层与文件系统之间的语义鸿沟，也阐述了恢复全部或者部分上下文信息可以有效提高块级的性能和可靠性。我们在这个观察之上去构建，通过恢复部分文件系统和应用上下文信息来改进块级重复数据删除。

上下文信息恢复可以通过自我检查（introspection）或提示（hinting）来实现。自省依靠块设备层的情报去推断文件系统或应用的操作。自省的好处是不需要任何文件系统的改动；缺点是成功的实施太过困难。相反，提示（hinting）需要更高层次为重复数据删除系统来提供少量额外信息。尽管文件系统或是应用一定要被改变，必要的调整跟好处相比起来就很小了。更重要的是，可以通过插入一个能从诸如文件或程序名称，文件格式等信息中推断提示的库来最小化应用程序更改。本文中，我们在块设备层用提示来恢复上下文信息。

**Dmdedup。**我们使用一个开源的块设备级重复数据删除系统Dmdedup去评估提示hint的作用。Dmdedup使用固定大小的块，依赖于Linux的加密API去散列。他可以用两个元数据尾端inram和cowbtree之一；前者只在RAM中存储元数据（如果RAM是有电池后备的），后者将元数据持久写入磁盘。也有人提出了基于软更新的元数据后端。

图1描绘了dmdedup的主要组成部分和在一个典型安装中的位置。



Dmdedup部署在物理块设备（如磁盘驱动器，磁盘阵列RAID，固态硬盘）的上层，或者其他逻辑设备（如加密用的dm-crypt）。他通常需要两个块设备去运行：一个存储真是用户数据的数据设备，一个保存组织信息（如哈希索引）的元数据设备。在我们的实验中，用一个HDD来保存数据，SSD保存元数据。把元数据存在SSD中是合理的，因为它比数据本身要小很多，通常都不到数据大小的1%，但是元数据却很重要，要求低延迟的访问。在上层，dmdedup提供了一个惯用的读写块接口。通常来说，对dmdedup的每一次写操作都会对所有现存数据进行哈希查找和检查；如果检测到一个重复数据，相应的元数据就会更新，然后也不会有数据被写入。新的非重复内容会被传到数据设备上然后在元数据中追踪。由于只存储给定块的一个实例，如果文件损坏，多个文件可能会收到影响。因此，dmdedup可以在磁盘阵列或者复制系统上运行进而最小化数据丢失的风险。

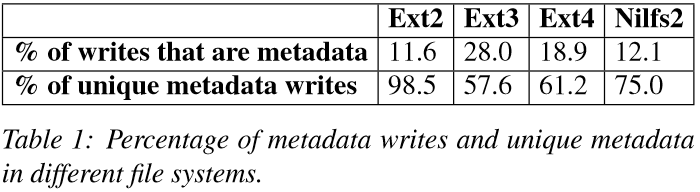
在其内部，dmdedup有五个组成部分：（1）给数据分块，计算哈希，定位其他组件的重复数据删除逻辑单元；（2）跟踪所有现存数据块的哈希和地址的哈希索引；（3）对上层可见的逻辑块号（LBN）和数据真实存储位置的物理块号（PBN）之间的映射；（4）一个在数据设备上跟踪空间，维护参考计数，分配新块，回收未被引用的数据的空间管理器；（5）一个存储用户数据到数据设备上的块仓库。

**3 潜在提示（Potential Hints）**

**跳过去重。** 一些已知为先验的写操作很有可能是唯一的。应用程序可能会生成那些不该或者不能被删除的数据。例如，有些应用程序会写入随机、压缩或者加密的数据；有的会写一些格式复杂的（比如虚拟磁盘图像）带有内部元数据并且很有可能是唯一的数据，以及一些高分辨率传感器会产生独一无二的数据流。

尝试去删除唯一的写操作会在计算哈希索引时浪费CPU时间，保存哈希索引的时候浪费I/O带宽。非重复的哈希也会增加索引的大小，就需要更多的RAM空间和带宽去用于查找，插入以及垃圾回收。

大部分的文件系统元数据都是唯一的—比如索引节点（有不同的时间戳和块指针），目录节点，间接块。表1展示了几个文件系统的4KB元数据写入的占比（唯一的和整体），我们用Filebench的文件服务器，将工作量从1KB的块调整到4KB的块（为了与重复数据删除系统的块大小匹配）。在所有通过文件系统的写入操作中大约有12-28%写的是元数据；在所有情况中至少有57%的元数据是唯一的。Ext3和Ext4比Ext2和Nilfs2有更多的元数据重复删除，一个由日志造成的现象是：Ext4先是在日志中写入元数据然后再讲同样的块写到磁盘上合适的位置。



元数据写入对于整个系统性能来说比数据写入要更为重要，因为前者通常是同步的。增加额外重复数据删除的开销可能会增加那些重要的元数据写入的延迟。避免过多的元数据去重删除也有助于可靠性因为许多文件系统会存储元数据的冗余的副本（比如Ext2/3/4会保存多个超级块；ZFS利用重复元数据去避免腐败）。对这些副本进行重复数据删除会避免这些特性。同样的，文件系统日志加强了可靠性，所以对他们的块进行重删可能会适得其反。一个被删重的日志同样也会失去顺序性，这样会伤害到性能。

总的来说，如果一个块级重复数据删除系统能够知道什么时候不该去删除一次写入，他就可以优化它的想能和可靠性。我们用一个NODEDUP提示去告知我们的系统一个相应的请求不应该被重删。

**预取哈希。** 当一个重复数据删除系统知道什么数据将要被写入的时候，它就可以从索引中预取哈希值，从而通过减少查找延迟来加入数据写入。例如，一个复制操作首先需要读取元数据，然后再将其写回。如果重删系统可以在读的时候识别这个行为，它就可以从索引中预取相应的哈希节点从而加速写的过程。我们实施这个提示并把它称之为预取（PREFETCH）。这个提示hint的另一个有趣的用例是在垃圾回收期间段之间迁移数据的日志结构文件系统（如Nilfs2）中的段清理。

**跳过压缩。** 有一些重复数据删除系统会压缩数据块来节省更多的空间。然而，如果一个文件以及被压缩过了，额外的压缩会消耗CUP时间，却并没有什么好处。

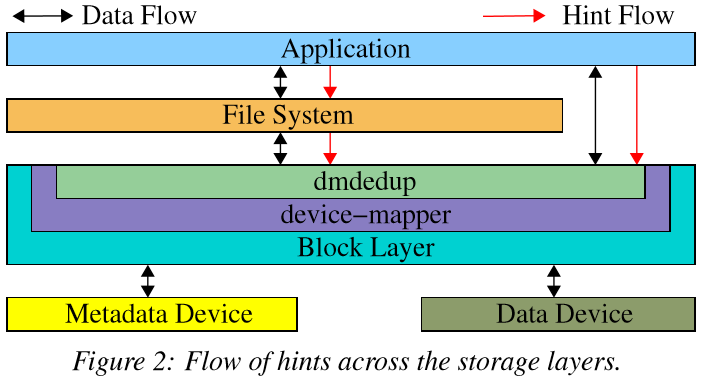
**聚集哈希。** 处于相同目录下的文件会倾向于同时被访问。在一个多用户的环境中，一个特定用户的工作集通常来说要比整个文件系统树要小很多。基于文件所有权或者哪些目录包含文件，重复数据删除系统可以在索引中对哈希进行分组，并更有效地预加载缓存。

**分区的哈希索引。** 基于传入块的属性对哈希索引进行分区是一种用来提升重复数据删除的流行的技术。在相同类型的文件中找到重复数据的几率要比在所有文件中找到的几率大很多，所以可以对文件进行分区，比如根据扩展名。

**智能分块。** 了解文件边界使得一个重复数据删除系统高效地对数据分块。某些大文件（比如压缩包）包含许多小文件。将有关内容边界的信息传递给设备块层可以带来更高的重删率。

**4 设计和实施**

为了使块设备层了解上下文，我们设计了一个让提示在存储栈中从上层流向下层的系统。应用程序和文件系统然后就可以向下层传递关于他们中的数据的重要信息。图2中的红色箭头展示了提示是怎么传送到块设备层的。我们部署了两个重要的提示hint：NODEDUP和PREFETCH。



**Nodedup。**由于重复数据删除会使用计算资源，可能提高延迟，它只应该在有潜在好处的时候才被执行。NODEDUP提示指示块设备层不要在写入时去删除一个特定的块。它有两个用例：（1）独一无二的数据：浪费资源去删除那些不会有重复的数据比如传感器数据或者加密的数据是没有意义的；（2）可靠性：有时候很有必要去保存某些数据块的副本，比如在很多文件系统中都有超块副本。

**Prefetch。** 在一个重复数据删除系统中最耗时的操作之一就是哈希查找，因为它经常需要额外的I/O操作。更糟糕的是，哈希的一个特性就是随机分布。因此，查找哈希通常需要随机的I/O，这在大多数存储系统中都是最慢的操作。同时，先前的研究表明，由于哈希索引的规模太大了，将他们全部存在内存中是不切实际的。

PREFETCH提示用于通知重复数据删除系统可能产生更多重复的I / O操作（例如，在文件拷贝期间），以便可以预取和缓存它们的哈希值以最小化随机访问。这个提示可以设置在会再次访问相同数据的应用的读取路径上。（需要知道的是读取操作通常只需要访问逻辑块号到物理块号的索引，跳过哈希索引。）

**4.1 部署**

为了给提示hints增加援助，我们对存储栈的几个部分进行了修改。为了支持hints从块设备层的上层传播到下层的，修改了大概77层内核代码。我们同时修改了OPEN系统调用，增加了两个标志位O\_NODEDUP和O\_PREFETCH。用户空间程序可以根据这两个标志位将hints传递到底层删重块设备。如果块设备层不支持这两个标志位，他们就会被忽略。需要冗余或者副本数很少的应用程序可以在调用写的时候传递O\_NODEDUP。相似地，知道哪些常用块，或者知道有些数据会被再次访问，就可以在读的时候传入O\_PREFETCH。被读块的哈希就可以被预取，所以在稍后的写操作中他们就可以在预取缓存中被找到。

通过修改和添加741行代码，我们将dmdedup改成支持NODEDUP和PREFETCH提示的形式。在dmdedup中，如果一个请求含有NODEDUP标志位，我们就跳过在哈希索引中的查找和更新。相反，我们仅仅在逻辑块号到物理块号的映射中增加一个条目。并不需要改变读取路径去支持NODEDUP。

在dmdedup的读取路径中，逻辑块号到物理块号的映射会被询问一个给定账号是否已知，但是这里并不需要哈希计算，因为之前的一个写操作已经把这个块添加到哈希到物理块号的映射中了。如果一个请求在读取路径上设置了PREFETCH提示，则dmdedup在读取数据之后对数据进行散列，并将相应的哈希→物理块号PBN放入预取高速缓存。在写入之后，我们的代码通过在搜索元数据后端之前检查缓存来节省执行时间。当在预取缓存中发现哈希时，它会被删除，因为在拷贝之后没有理由相信它将会很快被再次使用。

我们还修改了一些特定的文件系统，为其元数据传递NODEDUP提示，并且如果经过设置，还将OPEN标志传递给块层。对于Linux上的Nilfs2，我们修改了大概371行内核代码，用提示标记它的元数据并传播他们，以及从上层到块层的OPEN标志。相似的改变在Ext4中修改了16行代码；在Ext3中修改了6行。在Ext2/3/4中的改动很小，因为我们能够利用设置在文件系统元数据中的REQ\_META标志去决定是否要基于数据类型去进行重复数据删除。其余元数据相关的提示可以被推断出来，我们从进程名称jbd2中识别日志的写入。

**5 评估**

**实验装置。**实验中使用戴尔PowerEdge R710，搭载了英热尔Xeon4核CPU，24GB内存。机器运行Ubuntu Linux14.04版本，升级到Linux3.17.0内核。我们用英特尔DC S3700系列的200GB固态硬盘作为dmdedup元数据存储设备，Seagate Savvio 15K.2 146GB磁盘作为数据存储设备。两个硬盘都与主机控制器相连。尽管SSD很大，在我们整个系统中我们只用到1.5GB或者更少的空间来存储dmdedup元数据。

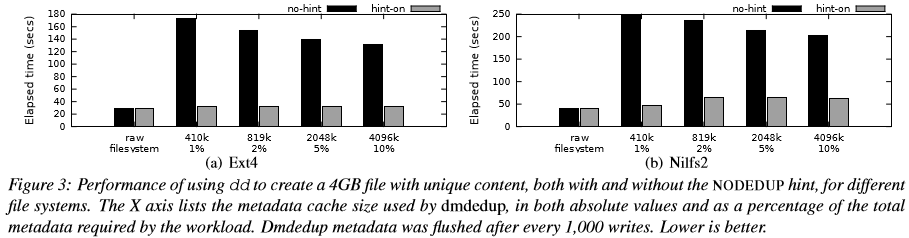
我们的所有实验都至少跑3次以确保标准偏差小于均值的5%。为了确保在微工作量中的所有脏数据都到达稳定的媒体，我们在每次运行的最后调用了sync然后卸载了文件系统；我们的时间度量也包括这两步。

对所有实验我们都使用dmdedup的cowbtree相互作用的元数据后端，因为它能够在崩溃是避免元数据状态的易变性。Cowbtree允许用户指定不同元数据缓存的大小；我们在每各个实验中使用重复数据删除的元数据1%，2%，5%和10%。这些比率在现实的重复数据删除系统中都是很典型的。Dmdedup也允许用户指定他们想要刷新元数据的粒度。我们在两种设置下跑所有的实验：每次写操作都刷新元数据，或者每1000次写操作刷新一次。我们的实验结果主要关注后一种情况因为这是个更加实际的设置。每次写操作都刷新就像是对每次操作使用O\_SYNC标志位，这在实际的系统中是不正常的；我们用这种设置去实现最差情况的评估。Dmdedup在收到来自上层刷新请求时也会刷新元数据。因此，dmdedup的数据持久性语义与常规块设备的相同。

**5.1 实验**

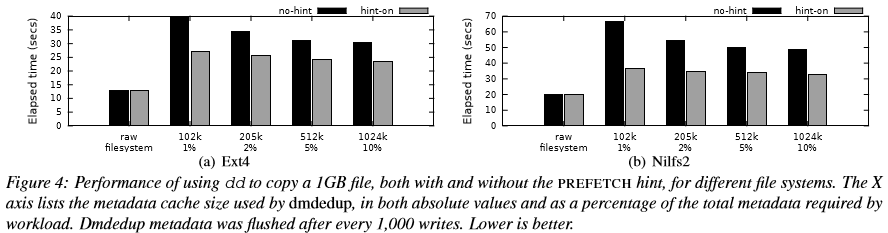
我们在4个文件系统上测试了NODEDUP和PREFETCH提示：Ext2，Ext3，Ext4和Nilfs2。Ext2是一个传统的在正确的位置更新元数据的类FFS文件系统；Ext3在Ext2的基础上增加了日志，Ext4进一步增加了扩展支持。Nilfs2是一个日志结构的文件系统：它将所有的写入操作序列化，并具有垃圾回收阶段来删除冗余的块。在所有案例中我们发现Nilfs2的性能要低于Ext4；其他的大都差不多。

**NODEDUP提示。**为了显示应用层提示的效用，我们在dd的写路径中添加NODEDUP作为一个open标志位。然后创建一个4GB的文件，文件中都是无重复的数据，在提示开和关的情况下。这个实验在正在写入唯一数据的系统上显示了NODEDUP提示的好处（例如在重复数据删除系统没有用的地方）。这个提示可能在产生许多副本的工作量是不你们有用。图3显示了每1000次写入元数据时，对Ext4和Nilfs2使用NODEDUP提示的好处；其他文件系统上的结果相似。我们发现NODEDUP提示减少了唯一数据的写次数2.2-5.3倍。由于I/O的开销很高，每次写吼都刷新dmdedup的元数据会消减NODEDUP的好处，但是我们仍然观察到1.3-1.6倍的提升。



**PREFETCH提示。** 为了评估PREFETCH们修改了dd在读取路径上使用O\_PREFETCH开放标记以便于写操作可以从缓存哈希中得到好处。然后我们用修改过的dd去重复复制一个1GB的文件，文件中的数据都是在同一个文件系统中的副重复数据。我们用无重复的内容，以便衡量可能发生的最坏的情况，然后确保预取缓存被最大程度地使用。我们还对一个小型研究组的主目录的哈希值进行了本地收集的数据集的研究。我们分析了哈希，以了解使用4KB块大小的文件中看到了多少重复块，然后发现有99%的文件都有唯一的块。因此在唯一内容上测试PREFETCH提示是有意义的。对于四种文件系统，结果都是相似的，因为大多数文件系统对单文件数据密集型工作的管理都是类似的。图4是Ext4和Nilfs2的结果。当我们每1000次写操作吼刷新一次dmdedup的元数据，相比较与没有提示配置的复制时间减少了1.2-1.8倍。当每次写操作后都刷新元数据，复制时间的变化范围从变差16%到变好16%。有提示带来的改善在这里并不显著因为刷新的开销相比与预取哈希带来的好处更高。

不出所料，PREFETCH提示的的效用依赖于重复数据删除率。比如，我们在以上实验里把重复数据删除率改为8:1，复制耗时就从变差9%到变好55%，这取决于文件系统类型和dmdedup的设置。



**宏工作量。**我们修改了Filebench，以给定的重复分布的形式生成数据，而不是任意的数据。然后，我们运行了Filebench的Fileserverworkload，修改了4KBblocks，以评估设置NODEDUP提示的好处：（1）文件元数据写入，在这里我们标记元数据块和日志写入这个提示，（2）文件数据写入与元数据写入。我们使用独特的写入工作负载来显示将NODEDUP提示应用于编写独特数据的好处。图5显示了单独为文件元数据写入和数据和元数据写入设置NODEDUP提示的最大好处。我们运行了Filebench，在1000次写入操作后刷新了所有的全唯一写入。当仅为元数据写入设置NODEDUP提示时，我们看到吞吐量增加了1-10％。当我们为数据和元数据写入设置提示时，我们看到Ext4的吞吐量提高了1.1-1.2倍，Nilfs2提高了3.5-4.5倍。当我们仅为元数据设置NODEDUP提示时，我们观察到性能提高但重复数据删除减少。从表1计算得到，在Ext4中所有写入的约7.3％和在Nilfs2中所有写入的3.0％都是重复的文件系统元数据写入。如果未设置NODEDUP提示，则Dmdedup将通过对这些写入进行重复数据删除来节省额外的空间。换句话说，这个提示为更低的重复数据删除率提供了更高的吞吐量和可靠性。

我们还运行了一个类似的实验（为了简洁起见没有显示），其中Filebench生成的数据的重复数据删除率为4：1（每个唯一的重复块为3个）。我们只为元数据写入设置了NODEDUP提示（因为Filebench在每个写入的基础上生成了唯一的数据，而您的操作基于全局），并将其与NODEDUP提示关闭的情况进行比较。我们看到吞吐量有小幅度的提升，从Ext4的4-7％到Nilfs2的6-10％不等。

6 结论和未来工作

在块层的重复数据删除有两个主要优势：（1）允许任何文件系统和应用程序从重复数据删除中受益，（2）易于实现。不幸的是，应用程序和文件系统上下文在块层丢失，这会损害重复数据删除的有效性。但是，通过添加简单但功能强大的提示，我们能够提供缺少的屏蔽层的语义，使得系统可以提高性能，并可能提高可靠性。我们的实验显示，在dd等应用程序中添加NODEDUP提示可以在复制唯一数据时将性能提高5.3倍，因为我们可以避免重复数据消除的重复数据消除开销。这个提示可以扩展到其他应用程序，如压缩或加密的应用程序。 在像dd这样的应用程序中添加PREFETCH提示，可以将复制时间缩短1.8倍，因为我们可以缓存哈希，而不需要访问元数据设备来在写入路径上获取它们。为诸如Filebench的Fileserver工作负载之类的宏工作负载添加提示，将吞吐量提高了4.5倍。另一个重要的提示是，提示的有效性取决于重复数据删除系统所增加的开销，所写入数据的性质（例如重复数据删除率）和工作负载，因此在选择使用提示时需要考虑所有因素。

**未来工作。**由于我们最初的实验取得了成功，我们打算为其他文件系统（如Btrfs和XFS）添加提示支持。我们还计划实施其他提示（在第3节中讨论），为块层提供更丰富的上下文以及支持 传递可用于增强提示的附加信息（例如，inode编号）。 为了节段的清理，我们把PREFETCHhint加到Nilfs2上。