F2FS文档

# Design Background and Function

F2FS (Flash Friendly File System) 是专门为基于 NAND 的存储设备设计的新型开源 flash 文件系统。特别针对NAND 闪存存储介质做了友好设计。F2FS 于2012年12月进入Linux 3.8 内核。目前，F2FS仅支持Linux操作系统。

F2FS 选择 log-structured文件系统方案，并使之更加适应新的存储介质(NAND)。同时，修复了旧式日志结构文件系统的一些已知问题，如(1) wandering tree 的滚雪球效应和(2)高清理开销。

根据内部几何结构和闪存管理机制(FTL)，闪存存储设备有很多不同的属性，所以F2FS的设计者增加了多种参数，不仅用于配置磁盘布局，还可以选择分配和清理算法，优化性能(并行IO提高性能)。

## Main Problem

### LFS(Log-structured File System)

为管理磁盘上的大的连续的空间以便快速写入数据，将 log 切分成 Segments，使用 Segment Cleaner 从重度碎片化的 Segment 中转移出有效的信息，然后将该 Segment 清理干净用于后续写入数据。

### Wandering Tree

在 LFS 中，当文件的数据块被更新的时候是写到 log 的末尾，该数据块的直接指针也因为数据位置的改变而更改，然后间接指针块也因为直接指针块的更新而更新。按照这种方式，上层的索引结构，如 inode、inode map 以及 checkpoint block 也会递归地更新。这就是所谓的 wandering tree 问题。为了提高性能，数据块更新的时候应该尽可能地消除或减少wandering tree 的更新节点传播。

### Cleaning Overhead

为了产生新的 log 空闲空间，LFS 需要回收无效的数据块，释放出空闲空间，这一过程称为 Cleaning 过程。Cleaning 过程包含如下操作：

(1)通过查找 segment 使用情况链表，选择一个 segment；

(2)通过 segment summary block 识别出选中的 segment 中的所有数据块的父索引结构并载入到内存；

(3)检查数据及其父索引结构的交叉引用；

(4)选择有效数据进行转移有效数据。

Cleaning工作可能会引起难以预料的长延时，因而 LFS 要解决的一个重要问题是对用户隐藏这种延时，并且应当减少需要转移的数据的总量以及快速转移数据。

## Characteristic and Function

### Not Support (for now)

当前F2FS不支持如下的特性，但是这些特性在以后如果必要的话也会支持：

(1)快照

(2)用户配额——最难实现

(3)不支持NFS

(4)不支持“security”方面的扩展特性(xattrs)

### Characteristic and Function

F2FS具有以下特性和功能：

(1)最大文件系统16TB；

(2)32位块寻址空间；

(3)最大文件大小3.94TB；

(4)Log-structured；

(5)Flash 感知

a) 扩大随机写区域（元数据区域，two-location）以获取更好的性能；

b) 尽量使 F2FS 的数据结构与 FTL 的运算部件对齐。

(6)解决Wandering Tree 问题

a) 用术语“node”表示 inode 以及各种索引指针块；

b) 引入 Node Address Table (NAT) 包含所有“node”块的位置，这将切断数据块更新的递归传播。

(7)最小化Cleaning 开销

a) 支持后台 Cleaning 进程；

b) 支持 greedy 和 低成本(转移数据最少)的待清理 Section 选择算法；

c) 支持 multi-head logs 用于动态/静态 hot 与 cold 数据分离；

d) 引入自适应 logging 用于有效块分配。

# Flash-friendly on-disk layout

F2FS采用三个可配置单位：segment，section和zone。 它从多个单独zone的以segment为单元中分配存储块。它以section为单位执行“cleaning”。 引入这些单位是为了与底层FTL的操作单位保持一致，以避免不必要的（但昂贵的）数据复制。

## On-Disk Layout

F2FS的磁盘数据结构经过精心布局，以配合底层NAND闪存的组织和管理。 如图1所示，F2FS将整个卷分成固定大小的段。 该段是F2FS中的基本管理单元，用于确定初始文件系统元数据布局。一个section由连续的segments组成，一个zone由一系列sections组成。 这些单元在logging和cleaning过程中非常重要。默认情况下一个 Zone 大的大小是一个 Section，而一个 Section 的大小是一个 Segment。

### 2.1.2 Blocks

(1) F2FS文件系统的所有块大小都是4KB，F2FS 代码隐式地将块大小链接到系统的页大小，因而F2FS不可能在更大的页的系统上运行，如 IA64 和 PowerPC。

(2) 块地址是32位的，最大文件系统是2^(32+12) Bytes，也就是16TB（完全大于当前的 NAND flash 设备的大小）。

### 2.1.2 Segments

(1) 连续的Blocks集合成Segments，一个Segment的大小是512个Blocks（也就是2MB）；

(2) 每个Segment都有一个Segment Summary Block元数据结构，描述了Segment 中的每个Block的所有者（该块所属的文件及块在文件内的偏移）。Segment Summary主要用于在执行Cleaning操作时识别哪些Blocks中的数据需要转移到新的位置，以及在转移之后如何更新Blocks的索引信息。一个Block就可以完全存储512个Blocks的summary信息，每个blocks都有一个1 bit的额外空间用于其它目的。

(3) 2MB是最合适的Segment大小，太大不合适，太小又会造成存储summary信息的Block空间浪费；

### 2.1.3 Sections

(1) 连续的Segments集合成Section。Section中具有的segments个数是任意的，但是要满足是2的幂；默认情况下，一个Section大小等同于一个Segment（2^0 Segments）；

(2) 一个Section对应log structuring的一个区域“region”，log在使用下一个Section之前，通常要从头到尾将当前的Section填满数据；

(3) 清理器Cleaner一次处理一个Section；

(4) 在F2FS中，任意时刻都有6个“打开的”Sections用于将各种不同种类的数据（元数据、数据）分别写入到各个Sections中，实现数据分离。这样便允许文件内容（数据）与其索引信息（节点，node）分离，允许F2FS文件系统根据各种启发式方法将Sections划分成三类：即“hot”、“warm”、“cold”。例如，目录数据被当做hot来对待，使其与文件数据分离，存放到“hot”Section中。Cold数据是指那些很长时间内都不会改变的数据，因而装满Cold数据的Section就不需要执行Clean操作。对于hot节点（索引信息节点），一般更新很快，一段时间之后，装满 hot 节点的Section中的有效数据(alive data)就会很少，因而选择这样的Section进行Clean操作开销就很小（因为要转移的数据很少）。

### 2.1.4 Zone

(1) 连续的Sections集合成Zone。一个Zone可以包含任意整数个Sections。默认是一个Zone中包含一个Section；

(2) 设置Zone的唯一目的是尽可能将6个打开的Sections位于Flash设备的不同的子设备中。理论上，Flash设备通常是由一组相互完全独立的子设备构成，每个子设备都可单独地处理I/O请求，不同子设备可并行处理I/O请求；如果Zone的大小与子设备大小对齐，6个打开的Sections可并行写入，充分利用设备的特性；

(3) Zone构成了F2FS的“主要(main)”区域。

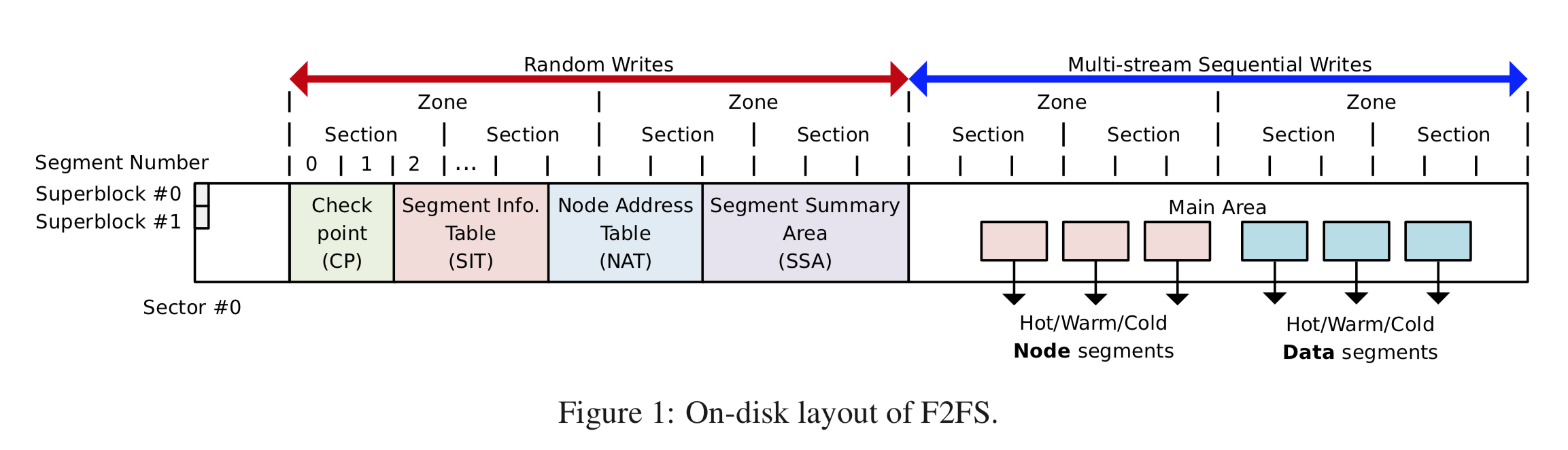
### 2.1.5 Meta Area

F2FS有一个“meta”区域，包含了各种不同的元数据（如之前提到的segment summary），这一部分不是采用标准的log-structured流水线方式管理，因而更多的工作留给了FTL去做。有三种方法管理对“meta”区域的写操作：

a) 第一，有少量的只读数据（超级块）从来都不是不是在文件系统创建的时候立即写入；

b) 第二，对Segment Summary Block 简单采取本地更新的方法。这种本地更新可能导致文件系统奔溃后数据块“修正”内容的不确定性，但对segment summary来说这都不是问题，segment summary blocks中的数据在使用前要进行有效性验证，如果有任何信息丢失的可能，它都将会在恢复进程中从其他source中恢复。

c) 第三种方法，分配需求空间两倍大小的空间，使得每个block都有两个不同的位置：一个Primary，一个Secondary，任意时刻，两个位置的block仅有一个是live状态。因而LFS的Copy-On-Write需求就可以通过向non-live位置的block写入更新后的block内容并且更新记录哪个位置的block是live状态的方式简单实现。对于元数据来说，这种技术是实现快照功能的主要实现方法。当创建一个Checkpoint的时候，F2FS执行少量的Journaling更新到最后的组(last group)，这在一定程度上减轻了FTL的工作。



F2FS将整个卷分成六个区域：除了超级块(Superblock，SB)外，其余每个区域都包含多个 Segments

•**超级块（SB）** 具有基本分区信息和F2FS的默认参数，这些参数在格式化时间给出并且不可更改。

与其他文件系统不同，F2FS 清晰地区分出传统超级块中的只读部分(Superblock，SB)和可修改部分(Checkpoint，CP)，存放在两个单独的数据结构(BP 和 CP)中。

F2FS 的 f2fs\_super\_block 存储在设备的第二个块中，仅包含只读数据，称为超级块 Superblock (SB) 。一旦文件系统创建，SB 的信息就不会再改变，SB 描述了文件系统有多大、Segment 有多大、Section 有多大、Zone 有多大以及分配了多少空间给各个部分的“元数据”区域以及其他少量的细节信息。

SB 位于文件系统分区的开头，有两个备份以避免文件系统 crash 无法恢复的情况发生。它包含基本的分区信息和默认的 F2FS 参数。

•**检查点（CP）** 保留文件系统状态，有效NAT / SIT集的位图（请参见下文），当前活动段的孤立索引节点列表和摘要条目。一个成功的“检查点包”应该在给定的时间点存储一致的F2FS状态 - 在突然断电事件后的恢复点（第2.7节）。 CP区域在两个段（＃0和＃1）上存储两个检查点包：一个用于上一个稳定版本，另一个用于中间（已过时）版本。

传统文件系统超级块中的可写信息，如空闲空间总量、下一个将要写入数据的Segment的地址以及其他可更改信息存储在 f2fs\_checkpoint 中，称为 Checkpoint (CP)。“Checkpoint”是一种元数据类型，允许使用两个位置(two-location)方法实现 copy-on-write——有两个相邻的 Segments，每个都存储一个 Checkpoint，但仅有一个是当前有效使用的。Checkpoint 包含一个版本号，因而当文件系统挂载的时候，两个 Checkpoint 都被读取，但是使用的是仅有较高的版本号的Checkpoint 作为有效使用的Checkpoint。

Checkpoint 包含文件系统信息，有效NAT/SIT集的位图(bitmaps)，Orphan inodes 的列表，以及当前有效 Segments 的 Summary 项。

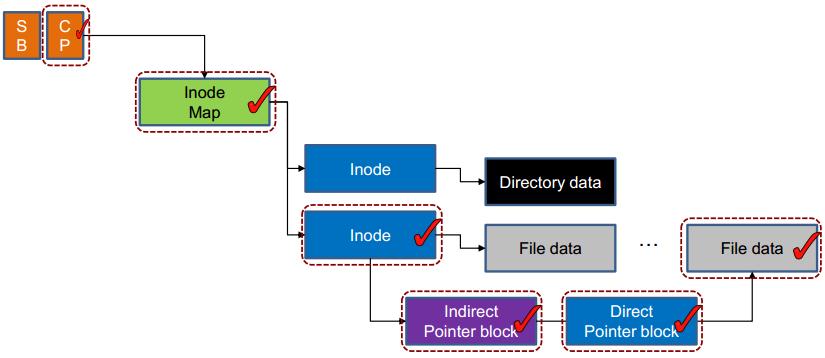
•**段信息表（SIT）** 包含每段信息，例如有效块的数量和位于“Main”区域（见下文）中所有块有效性的位图。检索SIT信息以选择受害人群并在清洁过程中识别其中的有效区块。

SIT 为每个 Segment 存储64字节的信息且与 Segment Summaries 分离，因为它修改的频率更高。它主要用来跟踪哪些数据块仍然是有效的(有效块个数以及数据块有效性bitmap)，因而当 Segment 中无有效块时，就可以回收该 Segment，或者当该 Segment 中有效数据块很少的时候进行 clean 操作。

•**节点地址表（NAT）** 是一个块地址表，用于查找存储在主区域中的所有“节点块”。

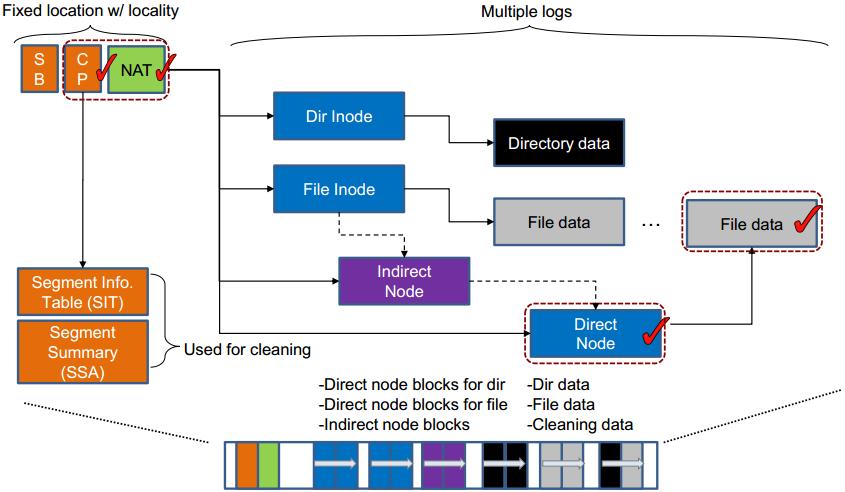
存储在 Main 区域的所有节点（inode 和 索引节点）数据块的块地址表。

所有的节点(node)块由 NAT 映射，这意味着每个 node 的位置由 NAT 表来转换。考虑 wandering tree 问题，F2FS 使用这种节点映射索引方式可以切断由于叶子节点修改操作引起的节点更新传播问题，如下图两个所示，其中下图是传统 LFS 文件系统 wandering tree 结构叶子节点修改所引起的所有需要修改的节点(红勾表示)



传统 LFS 文件系统 wandering tree 滚雪球效应

下图是传统F2FS 文件系统采用 NAT 数据结构后叶子节点修改所引起的所有需要修改的节点(红勾表示):



F2FS 文件系统引入NAT消除 wandering tree 滚雪球效应

•**段汇总区（SSA）** 存储表示主区中所有块的所有者信息的汇总表，例如父节点编号及其节点/数据偏移量。在清理期间迁移有效块之前，SSA条目标识父节点块。

SSA 中包含许多 summary 项， summary 项中包含存储在 Main 区域的所有数据和节点块( node blocks)的所有者信息。

•**主区域（数据区域/Log区域）填充4KB区块** 每个块被分配并输入为节点或数据。 节点块包含inode或数据块索引，而数据块包含目录或用户文件数据。 请注意，一个section不会同时存储数据和节点块。

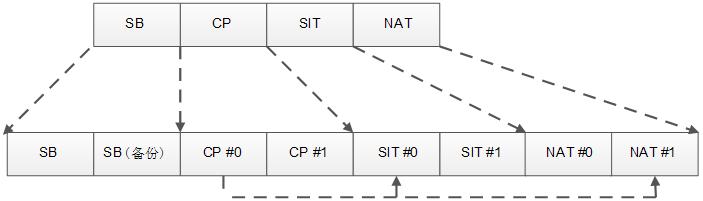
Main Area 包含文件和目录结构以及它们的索引数据。

•**元数据更新** 当需要对 NAT 或 SIT 更新的时候，F2FS 并不是立即执行更新，而是将其存储(缓存)在内存中，直到下一个 Checkpoint 写入的时候才会真正更新NAT 或 SIT。如果更新相对很少，那这些更新就不会写到它们最终的位置，而是以日志的方式记录到Segment Summary Blocks 中的一些备用空间中，通常与Segment Summary Blocks 的更新同时写入。如果Segment Summary Blocks 需要更新的总量足够少，甚至没有写请求，SIT、NAT 和 SSA都与Checkpoint 一起以日志方式记录更新。因而当 F2FS 留下一些工作给FTL 处理的时候，F2FS 试图友好地并且仅执行它不得不做的随机块更新。当 F2FS 需要做随机块更新的时候，它同时执行多个随机块的更新，这样可以减轻 FTL 的负担。

•**地址对齐** 为了避免文件系统和 flash 存储之间不对齐，F2FS 使 CP 的起始块地址与 Segment 大小对齐。同时，F2FS 使 Main Area 的起始块地址与 Zone 的大小对齐，这通过在SSA 中预留一些 Segments 来实现对齐。

•**F2FS文件系统元数据管理** F2FS 采用 Checkpoint 机制维护文件系统的一致性。在挂载文件系统的时候，F2FS 首先尝试通过扫描 CP 区域找到最新的有效的 Checkpoint。为减少扫描时间， CP仅使用两个备份，其中一个备份中包含的是最新的有效数据。这种两个备份，且仅有一个包含最新版本有效数据的技术也称为 shadow copy 机制。除了 CP，NAT 和 SIT 也采用shadow copy 机制保证数据一致性。

为保持文件系统一致性，每个 CP 指向有效的 NAT 和 SIT 备份，如下图所示：



考虑到上面的磁盘数据结构，让我们来说明如何完成文件查找操作。 假设文件“/ dir / file”，F2FS执行以下步骤：（1）通过读取一个从NAT获得位置的块来获得root inode; （2）在根索引节点块中，它从其数据块中搜索名为dir的目录条目并获取其索引节点号; （3）通过NAT将检索到的inode号码转换为物理位置; （4）通过读取相应的块得到名为dir的inode; （5）在目录索引节点中，它识别名为file的目录条目，最后通过对文件重复步骤（3）和（4）来获得文件索引节点。 实际数据可以从主区域检索，索引通过相应的文件结构获得。

# Cost-effective index structure

LFS将数据和索引块写入新分配的空闲空间。 如果叶数据块更新（并写入某处），则其直接索引块也应更新。一旦直接索引块被写入，其间接索引块应该被更新。 这种递归的更新导致了一连串的写作，造成了“wandering tree”问题[4]。 为了解决这个问题，我们提出了一个新的索引表，称为节点地址表。

## File Structure

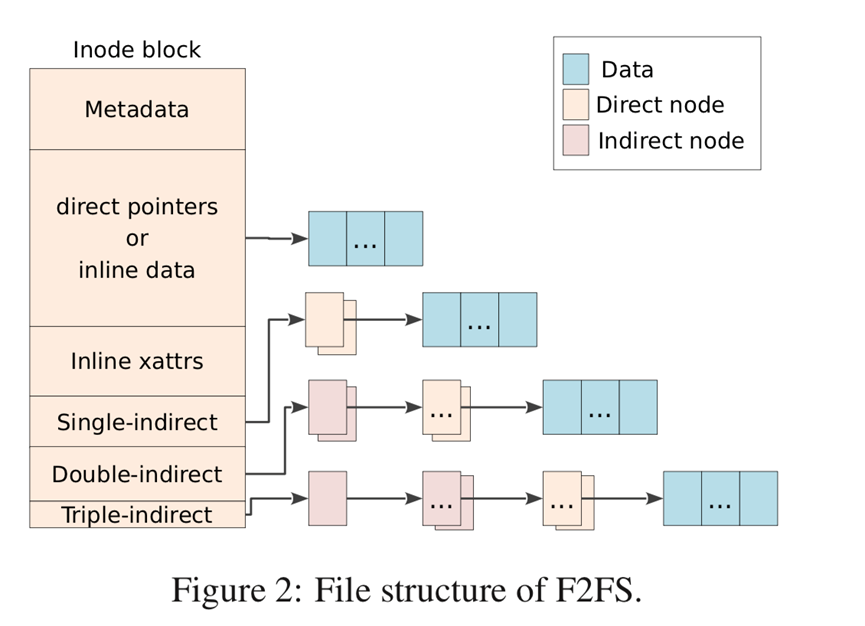
### 3.1.1 Inode

原始LFS引入的inode映射将inode编号转换为磁盘上的位置。相比之下，F2FS利用扩展索引节点映射的“节点”结构来定位更多索引块。每个节点块都有一个唯一的标识号“节点ID”。通过使用节点ID作为索引，NAT服务于所有节点块的物理位置。节点块代表三种类型之一：inode，直接节点和间接节点。 inode块包含文件的元数据，如文件名，inode编号，文件大小，atime和dtime。直接节点块包含数据的块地址，而间接节点块具有定位另一节点块的节点ID。

F2FS 分配4 KB 的空间给一个 inode，其中包括929个数据块索引指针，两个一级索引块(直接 node )指针，两个二级索引块(间接 node )指针，以及一个三级索引块(二级间接 node)指针。一个一级索引块(直接 node )包含1018个数据块指针，一个二级索引块(间接 node )包含1018个一级索引块(直接 node )指针，一个三级索引块(二级间接 node)包含1018个二级索引块(间接 node )指针。因此一个文件的最大大小是：

4 KB \* (929 + 2\*1018 + 2\*1018\*1018 + 1018\*1018\*1018) := 3.94 TB

如图2所示，F2FS使用基于指针的文件索引与直接和间接节点块消除更新传播（即“wandering tree”问题[27]）。在传统的LFS设计中，如果更新叶数据，则其递归地更新其直接和间接指针块。然而，F2FS只更新一个直接节点块和它的NAT条目，有效地解决了wanderring树问题。例如，当一个4KB数据附加到8MB到4GB的文件中时，LFS会递归更新两个指针块，而F2FS只更新一个直接节点块（不考虑缓存效果）。对于大于4GB的文件，LFS会更新多于一个指针块（总共三个），而F2FS仍然只更新一个。



inode块包含指向文件数据块的直接指针，两个单独间接指针，两个双重间接指针和一个三重间接指针。 F2FS支持嵌入式数据和嵌入式扩展属性，这些嵌入式数据和嵌入式扩展属性嵌入了inode块自身的小型数据或扩展属性。 内联减少了空间需求并提高了I / O性能。 请注意，许多系统都有小文件和少量的扩展属性。 默认情况下，如果文件大小小于3,692字节，F2FS将激活数据的内联。 F2FS在inode块中保留200个字节用于存储扩展属性。

### 3.1.2 Meaning of File Index Tree

但是这种索引树机制有一定的代价——这也是其他文件系统抛弃这种索引方式的主要原因——但是这种方式对 LFS 却有好处。因为F2FS不使用 extents ，任意给定文件的索引树是固定的而且大小已知。这意味着，当数据块由于 cleaning 操作需要转移数据时，不可能因为可用 extens 发生改变而导致索引树变得更大（原来文件中每个 extent 很大，一个文件的索引树仅仅用少数几个 extents 就可完全表示，而新的存储部分每个可用 extent 都很小，要存储原来的文件就需要更多的 extents，导致文件的索引树变大）的情况产生——这可能会导致一种尴尬的情况产生就是 cleaning 的目的本来是要释放空间，而现在索引树变大却导致空间变小了。Logfs 文件系统也因为相同的原因，使用了许多相同的编排处理方式。

显然，所有这些需求都要求 F2FS 的 inode 比 Ext3 的inode要大许多。因为Copy-on-write 不适用于小于块大小的对象，因而 F2FS 中每个 inode 占用的空间大小是一个 4 KB 的块，该 4 KB 的空间中提供了大量的空间用于索引。甚至提供空间存储（基本的）文件名，或者名字之一，以及其父节点的 inode 号，这简化了在文件系统 crash 恢复过程中近期创建的文件的恢复操作，减少了为保证文件安全需要写入的数据块的数量。

### 3.1.3 Extent to speed up file lookup

Inode 中包含了一个 extent，它总结了索引树的某些部分，它说明该文件的某些范围的blocks在存储上是空间连续的，并给出该范围的地址。F2FS 文件系统尝试在此处表示文件中最大的 extent，使用其加速地址查找。例如，通常情况下文件是连续写入且无任何明显的暂停，这样可能的结果是整个文件构成一个 extent，使得没有必要进入文件的索引树查找。

### 3.1.3 NAT address wandering Tree

任何基于 Copy-on-write 技术的文件系统的一个缺点在于，无论何时要改写一个数据块，该数据块的地址就会因为数据重写到其他位置而发生改变。然后，在索引树中该块的父节点也因此必须重写到其他位置，如此往复下去直到重写操作传播到索引树的根。LFS 的logging 特性意味着在恢复过程中向前回滚可以重建近期对索引树的修改，因而所有这些修改不必立即写到磁盘（只是写到 log 中），但是这些修改最终还是会写到磁盘的，这就将更多的工作丢给 cleaner 去做。

这是 F2FS 为了方便而使用其底层 FTL 去处理的另一个区域。“元数据”区域("meta" area)中的内容是一个 NAT——Node Address Table（节点地址表）。这里的节点是指inode 和间接索引块，以及用于存储扩展属性的数据块。当 inode 的地址存储在目录中，或者索引块存储在一个 inode 中或者其他的索引块中的时候，它存储的不是（存储的内容所在的）地址，而是其在 NAT 中的偏移。实际的（存储内容的）块地址是存储在 NAT 中对应的偏移处。这意味着当一个数据块被写入，仍然需要更新和修改指向该数据块的节点，但是修改该节点仅需更新对应的 NAT 项。NAT 是元数据的一部分，NAT 使用两个位置(two-location)的 Journaling（依靠FTL实现写聚合），因而不需要进一步的索引。

### 3.1.4 Inline Technology support situation

F2FS 目前不使用小文件内嵌（Inline）技术。针对这一技术的实际情况是：暂时还不支持 Inline 技术，但是很容易实现加入这一技术，并且这一技术迟早会加入到 F2FS中。

### 3.1. Time resolution

很不幸，F2FS 没有足够空间存储64位的时间戳，也就是最小时间分辨率不是纳秒(ns)，而是以秒(s)为时间分辨率。这可能导致2038问题，但时间戳分辨率问题应该会在未来的版本中解决。

## Directory

LFS(Log-structured File System) 实际上对目录布局没有强加任何特殊的要求，因而访问 F2FS 文件系统的目录等同于访问其他文件系统的目录。目录的主要目的是提供更快速的文件名查找，为每个可使用 telldir() 报告的文件名提供稳定的地址。

原始的 Unix 文件系统支持256字节的文件名长度，使用与 Ext2 相同的目录机制——在布满目录项的文件中顺序遍历查找文件名。这种方法简单有效，但是对于大目录的扩展性不好。

多数现在主流文件系统如 Ext3，XFS 和 Btrfs 使用多种机制（包括 B-tree），有时通过哈希索引文件名。B-tree 的一个问题是节点（nodes）需要切分，这导致一些目录项在文件中频繁移动，这给 telldir() 提供文件名对应的稳定地址带来了额外的挑战。这大概就是 telldir() 经常被认为是不太好的接口的主要原因。

### Directory Structure

一个目录项占据11 Bytes，也就是包含以下属性：

(1) – hash 文件名的哈希值

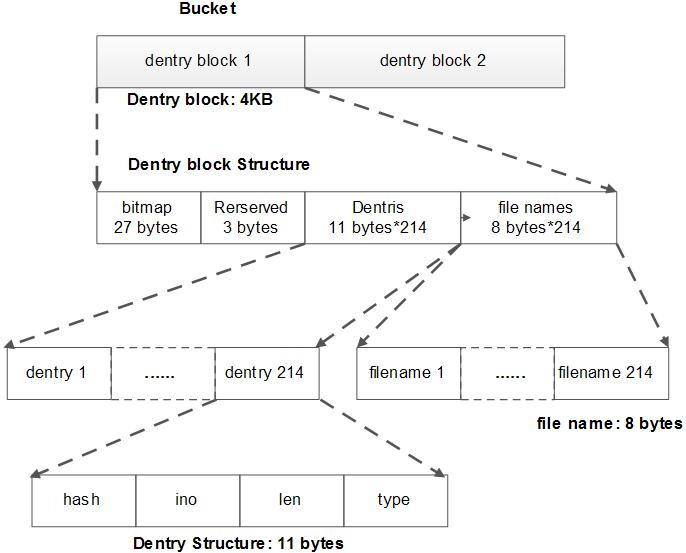
(2) – ino inode 号

(3) – len 文件名长度

(4) – type 文件类型，如目录，符号链接，等等

一个目录项 block 包含214个目录项槽位(slot)和文件名，其中用一个 bitmap 表示每个目录项是否有效，1个目录项块占用 4 KB，结构如下：

Dentry block(4 Kb) = bitmap(27bytes) + Reserved(3bytes) + Dentries(11\*214bytes) + filename(8\*214bytes)



### Directory Hash

F2FS 使用一些连续搜索和哈希方法提供一个简单的搜索机制，合理高效，简单实现提供稳定的 telldir() 的地址。F2FS 文件系统的大多数的哈希算法代码都是参考 Ext3，但是删除（忽略）了使用每个目录作为seed。F2FS 的 seed 使用一个秘密的随机数，以确保满足使用的哈希值在每个目录中都不同，因而是不可预测的。使用这样的 seed 可以给哈希冲突提供保护。虽然哈希冲突在现实中是不可避免的，但使用这种方式可以很容易避免这种冲突，这种删除(忽略) 使用每个目录作为seed 带来的效果很意外。

很容易想到目录结构是一系列的哈希表连续地存储在一个文件中，每个哈希表具有一组相当大的 buckets。查找过程从第一个哈希表到下一个哈希表，在每一个阶段对合适的bucket 执行线性查找，直到找到文件名或者查找到最后的那个哈希表。在查找过程中，在合适的 bucket 中的任意空闲空间都会被记录，以防需要创建文件名的时候用到。

F2FS 对目录结构实现了多级哈希表，每一级都有一个使用专用数字的哈希 bucket 的哈希表，如下所示。注意，“A(2B)”表示一个哈希 bucket 包含两个数据块。

----------------------

A : bucket 哈希bucket

B : block 数据块

N : MAX\_DIR\_HASH\_DEPTH 最大目录哈希层级

----------------------

level #0: A(2B)

level #1: A(2B) - A(2B)

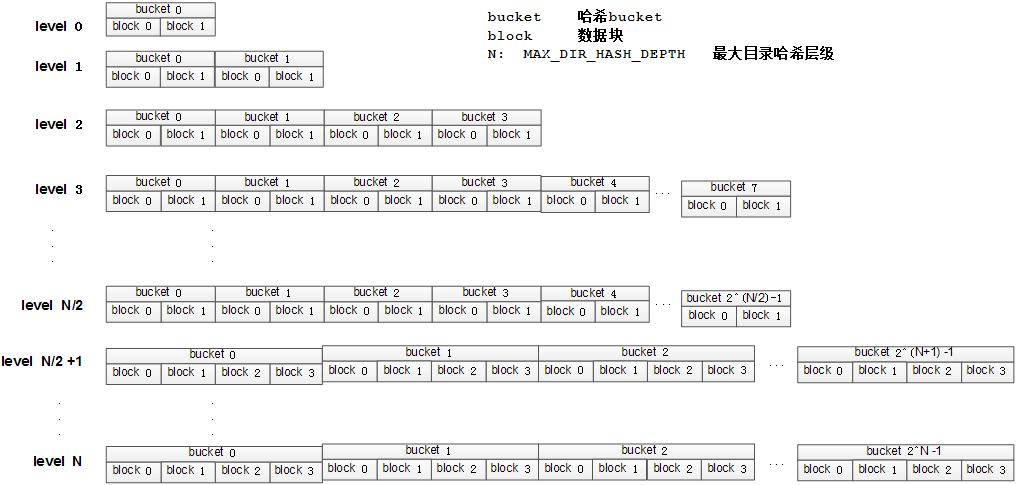
level #2: A(2B) - A(2B) - A(2B) - A(2B)

. . . . .

level #N/2: A(2B) - A(2B) - A(2B) - A(2B) - A(2B) - ... - A(2B)

. . . . .

level #N: A(4B) - A(4B) - A(4B) - A(4B) - A(4B) - ... - A(4B)



所有的哈希表按如下方式组织：第一个哈希表恰好有1个 bucket，大小是两个数据块，因而对于最开始的少数几百个目录项，使用简单的线性搜索。第二个哈希表有2个 bucket，第三个哈希表有4个 bucket，第四个哈希表有8个 bucket，然后是16 个 bucket，……，直到第31个哈希表有大约10亿（2^30）个 bucket，每个 bucket 都是两个数据块大小。从第32个哈希表开始，都与第31个哈希表一样，但是不同的是每个 bucket 的大小是四个数据块。哈希表中 block 个数与 buckets 个数的确定方法是：



F2FS 在目录中查找文件名时，首先计算文件名的哈希值，然后在 level #0 中扫描哈希值查找包含文件名和文件的 inode 的目录项。如果没有找到，F2FS 在 level #1 中扫描下一个哈希表，用这种方式，F2FS 以递增的方式扫描每个 level 的哈希表（如果上一 level 中没有查找到结果），在每个 level 中，F2FS 仅需要扫描一个bucket，而该 bucket 的号是由文件名的哈希值与该 level 中的 buckets 个数的相除取余得到的。也就是每个 level 中需要扫描的一个 bucket 由下式确定的，查找复杂度是 0：

bucket number to scan in level #n = (hash value) % (# of buckets in level #)

所以哈希表的最终结果就是需要线性搜索几百个目录项，如果目录项很大的话，需要通过几个数据块的搜索。搜索长度仅随着目录文件中目录项的个数对数级增长，所以容易扩展。这种搜索当然比完全的线性搜索算法要好，但是看起来好像比实际需要做的工作更多。但是它保证了每次加入或删除一个文件名的时候仅需要更新一个数据块，因为目录项没有移动，目录项在文件中的偏移对于 telldir() 来说就是一个稳定的地址，这是非常有意义的特性。

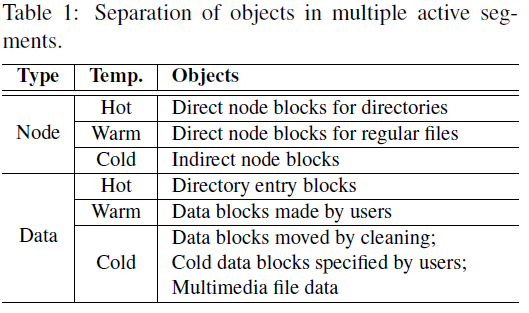
在创建文件的情况下，F2FS 在哈希表的buckets 中找到该创建文件名对应的空的连续的槽位(slots)（这一句的原文是：F2FS finds empty consecutive slots that cover the file name）。F2FS 自哈希表的 level #0 到level #N 从哈希表中给搜索该空闲的槽位，与查找文件系统中已有文件的查找操作一样。

# Multi-head logging

我们设计了在logging时间（即块分配时间）期间应用的有效热/冷数据分离方案。 它同时运行多个活动日志段，并根据其预期的更新频率将数据和元数据附加到单独的日志段。由于闪存存储设备利用介质并行性，因此可以同时运行多个活动段而无需频繁管理操作，由多次logging（vs. single-segment logging）而导致的性能下降无足轻重。

## Multi-head logging

与具有一个大log区域的LFS不同，F2FS拥有六个主要log区域，以最大限度地提高冷热数据分离的效果。 如表1所示，F2FS为节点和数据块静态定义了三个温度级别：热、温、和冷。



直接节点块被认为比间接节点块热，因为它们更频繁地更新。 间接节点块包含节点ID，并且仅在添加或删除专用节点块时写入。 目录的直接节点块和数据块被认为是很热的，因为与普通文件的块相比，它们有明显不同的写入模式。 满足以下三个条件之一的数据块被认为是冷的：

•**通过清洁移动数据块** 由于它们在很长一段时间内仍然有效，我们预计它们在不久的将来仍会如此。

•**用户标记为“冷”的数据块** 为此，F2FS支持扩展属性操作。

•**多媒体文件数据** 他们可能会显示一次写入和只读模式。 F2FS通过将文件的扩展名与注册的文件扩展名相匹配来标识它们。

默认情况下，F2FS激活六个日志写入。 用户可以在安装时将写入数据流的数量调整为两个或四个，如果这样做被认为可以在给定的存储设备和平台上获得更好的结果。 如果使用六个日志，则每个日志记录段直接对应于表1中列出的温度级别。在四个日志的情况下，F2FS在每个节点和数据类型中组合冷日志和热日志。 只有两个日志，F2FS分配一个节点，另一个分配数据类型。

F2FS引入了可配置区域以便与FTL兼容，以减轻垃圾收集（GC）开销。根据数据和“日志闪存块”之间的协同性，FTL算法大致分为三个组（块关联，集合关联和完全关联）[24]。一旦数据闪存块被分配用于存储初始数据，日志闪存块就尽可能地吸收数据更新，就像EXT4 [18]中的日志一样。对于所有数据闪存块（完全关联的）[17]，或者对于一组连续的数据闪存块（set-associative），该日志闪存块可专门用于单个数据闪存块（块关联）[13] ）[24]。现代FTL采用完全关联或集合关联的方法，以便能够正确处理随机写入。请注意，F2FS使用多头记录并行写入节点和数据块，并且关联FTL会将分离的块（在文件系统级别中）混合到同一个闪存块中。为了避免这种错位，F2FS将活动日志映射到不同的区域，以在FTL中分离它们。预计该策略对于集合关联FTL是有效的。多头记录也是最近提出的“多流”接口[10]。

# Cleaning

清理是回收分散和无效块的过程，并确保可用段进一步记录。 因为一旦潜在的储存容量已经填满，清洁就会不断发生，限制与清洁相关的成本对于维持F2FS（以及一般的任何LFS）的性能极为重要。 在F2FS中，清洁是以一个section为单位完成的。

F2FS以两种截然不同的方式进行清洁，即前台和后台。 前台清理仅在没有足够的空闲部分时触发，而内核线程定期唤醒以在后台进行清理。 清洁过程需要三个步骤：

（1）**受害者选择** 清洁过程首先开始识别非空白部分中的受害者部分。在LFS清洁期间有两种众所周知的受害者选择政策 - 贪婪和成本效益[11,27]。贪婪策略选择具有最少有效块数的section。直观地说，该策略控制迁移有效块的开销。 F2FS采用贪婪策略进行前台清理，以尽量减少应用程序可见的延迟。此外，F2FS还保留了一小部分未使用的容量（默认为存储空间的5％），以便清洁过程在高存储利用率水平下有充足的操作空间。第3.2.4节研究了使用水平对清洁成本的影响。

另一方面，成本效益政策是在F2FS的后台清理过程中实施的。该政策不仅基于其使用情况而且还根据其“年龄”选择受害者部分。 F2FS通过section中segments的年龄的平均来推断section的年龄，这又可以从他们在SIT中记录的最后修改时间获得。通过成本收益政策，F2FS有机会分离冷热数据。

（2）**有效的块识别和移植** 选择受害者section后，F2FS必须快速识别该section中的有效块。 为此，F2FS在SIT中为每个段维护一个有效位图。 通过扫描位图识别出所有有效的块后，F2FS将从SSA信息中检索包含其索引的父节点块。 如果块有效，F2FS将它们迁移到其他空闲日志。

对于后台清理，F2FS不会发出实际的I / O来迁移有效的块。 相反，F2FS将块加载到页面缓存中并将其标记为脏。 然后，F2FS将它们留在内核工作线程的页面缓存中，以便稍后将其刷新到存储。 这种懒惰的迁移不仅缓解了对前台I / O活动的性能影响，而且还允许组合小写入。 当正常的I / O或前台清洁过程正在进行时，后台清洁不起作用。

（3）**后期清洁过程** 在所有有效的块被扫描后，受害者section被注册为候选人，成为新的空闲部分（在F2FS中称为“预空闲”部分）。 检查点完成后，该部分最终成为空闲部分，以便重新分配。 我们这样做是因为如果在检查点之前重新使用了预先空闲部分，则文件系统可能会在发生意外断电时丢失之前检查点引用的数据。

# Adaptive logging

F2FS基本上建立在追加记录上，以将随机写入转换为顺序写入。 然而，在高存储利用率的情况下，它将日志策略改变为线程化日志[23]以避免长时间的写入延迟。 本质上，线程化日志记录将新数据写入脏段中的空闲空间，而不在前台进行清理。 这种策略适用于现代闪存设备，但在HDD上可能不这样做。

## Adaptive Logging

原来的LFS引入了两种日志记录策略，即正常日志记录和线程日志 在正常的日志记录中，块被写入干净的段，产生严格的顺序写入。 即使用户提交很多随机写入请求，只要存在足够的可用日志记录空间，此过程就会将它们转换为顺序写入。 然而，随着可用空间缩小到零，这项政策开始遭受高清洁开销，导致严重的性能下降（在苛刻的条件下量化为90％以上，参见第3.2.5节）。 另一方面，线程化日志记录将块写入现有脏段中的空洞（失效，废弃空间）。 此策略不需要清理操作，但会触发随机写入，因此可能会降低性能。

F2FS根据文件系统状态动态地实现策略并在它们之间切换。 具体来说，如果有超过k个清洁部分，其中k是预先定义的阈值，则会启动正常日志记录。 否则，线程日志被激活。 k默认设置为总节数的5％，并且可以进行配置。

当存在散布的漏洞时，线程化记录有可能导致不希望的随机写入。 但是，这种随机写入通常会显示比原生更新文件系统更好的空间局部性，因为在F2FS在其他脏段中搜索更多内容之前，脏段中的所有空洞都会先填充。 Lee等人 [16]表明闪存设备显示出更强的空间局部性，随机写入性能。

# Block Allocation and Space Management

## Default Block Alloction

在运行时，F2FS在“Main Area”区域管理6个活跃的log，分别是Hot/Warm/Cold node log和Hot/Warm/Cold data logs：

(1) Hot node contains direct node blocks of directories.

(2) Warm node contains direct node blocks except hot node blocks.

(3) Cold node contains indirect node blocks

(4) Hot data contains dentry blocks

(5) Warm data contains data blocks except hot and cold data blocks

(6) Cold data contains multimedia data or migrated data blocks

根据数据所属的类型，利用对应的log分配的空间将数据写入该空间。

## Spare Space Management

LFS有两种机制用于空闲空间管理：Threaded log和copy-and-compaction。Copy-and-compaction机制就是log-structured文件系统中常常提到的cleaning操作，Copy-and-compaction机制非常适用于具有非常好的连续写性能的设备，但是该机制的cleaning操作开销对性能影响很大。相反，Threaded log机制的弊端是要承受随机写，但是不需要cleaning过程。F2FS采用混合机制，其中Copy-and-compaction机制是默认采取的机制。但是F2FS会根据文件系统的状态(在没有足够的clean的segments )动态地更改为Threaded log机制。

为了使F2FS与底层的flash存储设备对齐，F2FS在以section为单位的存储单元中分配一个segment，F2FS期望section的大小与FTL中的垃圾回收单元的大小一致。此外，至于FTL中的映射粒度，F2FS尽可能从不同的Zone中分配每个（上述6个）活跃的log的section，因为FTL可以将活跃的log中的数据根据其映射粒度（并行）写到分配的单元中。

## Cleaning Process

F2FS可以在需要的时候(on-demand)或者空闲的时候以后台处理的方式进行clean操作。On-demand cleaning操作是由于没有足够的空闲segments服务VFS调用而触发开启的cleaning操作，后台cleaner由内核线程操作，当系统I/O Idle的时候触发cleaning操作。

F2FS支持两种选择待清理segments的策略：greedy和cost-benefit算法。在greedy算法中，F2FS选择具有最少有效数据块个数的segment；在cost-benefit算法中，F2FS根据segment的年龄以及有效数据个数选择segment，以解决greedy算法中的log block颠簸问题(频繁的cleaning操作，频繁的数据块迁移，极端情况，选中的segment可能在很长一段时间内一直是greedy算法的选择结果)。F2FS对on-demand cleaner采用greedy算法选择待清理的segment，而对于后台cleaner采用cost-benefit算法。

为识别出选中的segment中哪些数据有效，F2FS管理一个bitmap，每一位表示一个数据块的有效性，该bitmap描述了Main Area中所有数据块的数据有效性。

## 文件系统空间写满的处理

传统文件系统中如果没有剩余空间，直接返回错误就行。而对于Log-structured文件系统则没有这么潇洒地简单返回错误的处理机制，因为没有空闲块并不表示已使用的块上的数据都有效，在clean操作后又会释放出空闲块。通常一个比较有意义的做法是为Log-structured文件系统提供过量的空间，使得总是有空闲的Section用于cleaning操作中的有效数据的拷贝存储。

FTL恰好利用这种方法，提供过量空间用于“cleaning”以及用于替换高度损耗造成的坏块。FTL在内部处理过量空间，因而当F2FS开始用完空间的时候，它本质上放弃了整个log-structured的理念，只是随机地在其可以写入的地方写入数据。但此时，Inode和索引块仍然会仔细认真对待（不是随意写），有少量的过量空间用于它们的写入。但是对于数据只能采取本地更新，或是写入任意能找到的空闲块中。这样，可以预见的是当F2FS变满的时候，性能会下降，这与大多数的文件系统表现一样。

## Fully Utilize F2FS

F2FS的一个难处在于F2FS的shapes (即Sections和Zones)需要针对特定的flash设备及其FTL进行设置，而设备的厂商一般都对其FTL如何工作是保密的。F2FS也要求flash设备舒适地具有6个或更多可并行写入的“打开的”区域，这对三星的flash设备来说没有问题，但是对于其他厂商的设备就难说了。如果参数调整的不好(“打开的”区域不能实现并行写入)，F2FS的性能可能就会大大下降。

# Fsync acceleration with roll-forward recovery

通过最小化所需的元数据写入并通过高效的前滚机制恢复同步的数据，F2FS优化了小型同步写入以减少fsync请求的延迟。

## Checkpoint and Recovery

F2FS实现检查点功能，以便在突然断电或系统崩溃时提供一致的恢复点。 无论何时需要在同步，卸载和前台清理等事件中保持一致状态，F2FS将按如下方式触发检查点过程：（1）刷新页面缓存中的所有脏节点和dentry块; （2）暂停一般写活动，包括创建，取消链接和mkdir等系统调用; （3）将文件系统元数据，NAT，SIT和SSA写入其磁盘上的专用区域; （4）最后，F2FS将包含以下信息的检查点包写入CP区域：

•**页眉和页脚分别写在包的开头和结尾** F2FS在页眉和页脚中维护一个在创建检查点时增加的版本号。 版本号在安装时间内区分两个录制包之间的最新稳定包;

•**NAT和SIT位图** 指示包含当前包的一组NAT和SIT块;

•**NAT和SIT日志** 包含少量最近修改的NAT和SIT条目，以避免频繁的NAT和SIT更新;

•**活动段的摘要块** 包含内存SSA块，将来会刷新到SSA区域; 和

•**孤立块保留“孤立inode”信息** 如果inode在关闭之前被删除（例如，当两个进程打开一个公共文件并且一个进程删除它时会发生这种情况），它应该被注册为孤立inode，以便F2FS在突然关闭电源后可以恢复它。

### 8.1.1 Roll-Back Recovery

突然关机后，F2FS回滚到最新的一致检查点。为了在创建新包时保持至少一个稳定的检查点包，F2FS维护两个检查点包。如果检查点数据包在页眉和页脚中具有相同的内容，则F2FS认为该数据包有效。否则，它被丢弃。

同样，F2FS也管理两套NAT和SIT块，由每个检查点包中的NAT和SIT位图区分。当在检查点中写入更新的NAT或SIT块时，F2FS将它们写入两个集合中的一个集合，然后将该位图标记为指向它的新集合。

如果少量NAT或SIT条目频繁更新，F2FS将编写多个4KB大小的NAT或SIT块。为了减轻这种开销，F2FS在检查点包内实施NAT和SIT日志。该技术减少了I / O数量，并相应地减少了检查点延迟。

在安装时的恢复过程中，F2FS通过检查页眉和页脚来搜索有效的检查点包。如果两个检查点包都有效，则F2FS通过比较其版本号来选择最新的检查点包。一旦选择了最新的有效检查点包，它将检查是否存在孤立inode块。如果是这样，它会截断它们所引用的所有数据块，并最终释放孤立的inode。然后，在前滚恢复过程成功完成后，F2FS将使用其位图引用的一组一致的NAT和SIT块启动文件系统服务，如下所述。

### 8.1.2 Roll-Forward Recovery

像数据库（例如SQLite）这样的应用程序经常将小数据写入文件并执行fsync以保证持久性。支持fsync的最佳方法是触发检查点并使用回滚模型恢复数据。但是，这种方法导致性能较差，因为检查点包含编写与数据库文件无关的所有节点和分区块。

F2FS实现了一个高效的前滚恢复机制来增强fsync的性能。关键的想法是仅写入数据块及其直接节点块，不包括其他节点或F2FS元数据块。为了在回滚到稳定检查点之后选择性地查找数据块，F2FS保留直接节点块内的特殊标志。

F2FS按照以下方式执行前滚恢复。如果我们将最后一个稳定检查点的日志位置表示为N，（1）F2FS收集具有位于N + n中的特殊标志的直接节点块，同时构建它们的节点信息的列表。n是指自上一个检查点以来更新的块数。（2）通过使用列表中的节点信息，它将名为N-n的最近写入的节点块加载到页面缓存中。 （3）然后，它比较N-n和N + n之间的数据索引。（4）如果检测到不同的数据索引，则刷新存储在N + n中的新索引的高速缓存节点块，最后将其标记为脏。一旦完成前滚恢复，F2FS将执行检查点操作以将整个内存中的更改存储到磁盘。

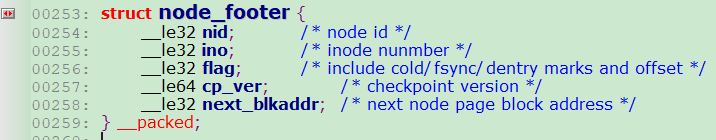
# Summary

简而言之，F2FS建立在LFS的概念基础之上，从新的设计因素考虑，f2fs与LFS提案有很大的不同。 我们已经将F2FS作为Linux文件系统实施，并将其与两个最先进的Linux文件系统EXT4和BTRFS进行比较。 我们还评估了NILFS2，它是Linux中LFS的替代实现。 我们的评估考虑了两个通常分类的目标系统：移动系统和服务器系统。 在服务器系统的情况下，我们研究SATA SSD和PCIe SSD上的文件系统。 我们在这项工作中获得的结果突出了F2FS的整体理想性能特征。

# NAT数据结构

## 10.1 NAT如何区分inode和其他dnod

首先，NAT中的每个表项都对应着MAIN AREA区域中NODE段的一个block，NODE block很特别，block末尾会有一个node\_footer结构。

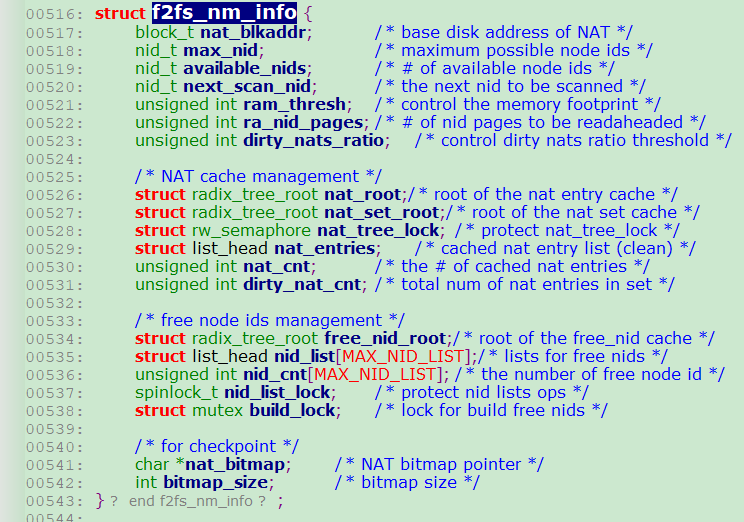


区分一个NODE block里是inode还是普通的dnode，只要比较footer中的nid和nio两个域就可以，二者如果相等，那么block里面就是个inode，否则就是普通的dnode。

还要说一点，新建一个文件的时候，会从VFS层分配一个inode，但是这个inode结构的 ino 是不确定的，需要f2fs给它一个值，这个值f2fs通过alloc\_nid函数获得，从可用的nid的选择一个，把这个值作为新申请的文件的ino，这样一来，inode的ino和nid就成一样的了，但是对于普通dnode来说，nid还是分配的这个nid，但是ino必须要赋值成文件的ino了。

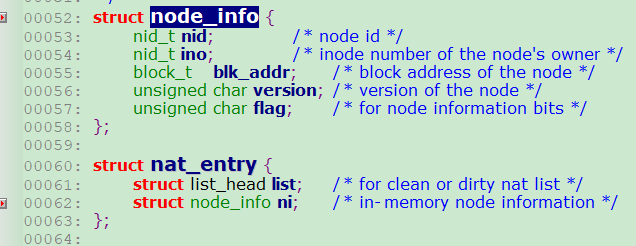
## 10.2 Node管理结构梳理

node是f2fs重要的管理结构, 它非常重要! 系统挂载完毕后, 会有一个f2fs\_nm\_info结构的node管理器来管理node的分配. f2fs\_nm\_info中最让人疑惑的是几颗基数树:

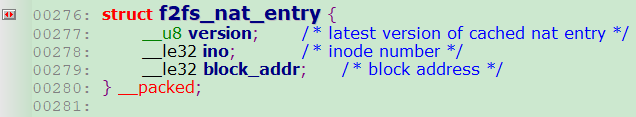


三棵基数树分别是: nat\_root, nat\_set\_root, free\_nid\_root;

首先, 对于整个node管理器来说, 一个核心的结构体是node\_info:



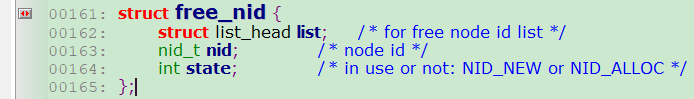
发现node\_info和f2fs\_nat\_entry【磁盘上的存储】长得太像了, 甚至比 struct f2fs\_nat\_entry 还要多出来一个flag 位来表示这个节点的一些属性！



这个core数据结构被两个结构控制：一个是基数树【负责索引】，一个是链表。

## 10.3 Node管理器中的free\_nid结构体

除了node\_info之外， node管理器中还有还有个重要的数据结构：



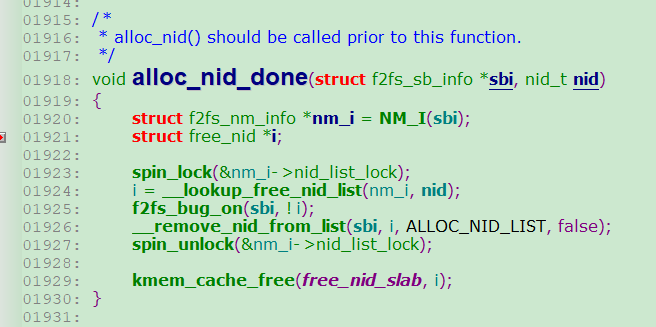
这个结构体体很简单，比刚才的node\_info轻量级多了，仅仅是标识了当前可以使用的nid，以及这个nid的状态，一个指针将其与别的free\_nid连成一串。

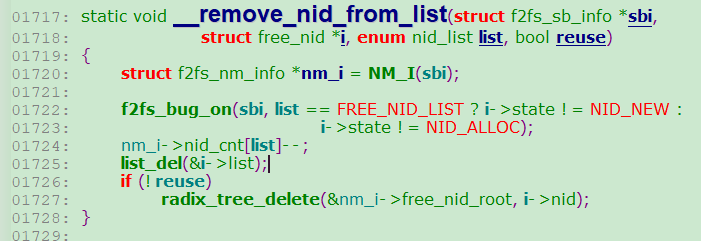
别看这个结构简单，也被node管理器中两个 “索引” 管理着呢：1）基数树 free\_nid\_root； 2）链表 free\_nid\_list；【参见函数add\_free\_nid】

但是咱这个free\_nid 也总得有个出栈的时候吧，你得为国家建设做贡献，不然在占个位置干嘛呢！

alloc\_nid 函数会从 free\_nid\_list 链表中选出一个free\_nid出来，这个nid算是被选中了，但是此时并不会把这个节点从list删除，为什么呢？因为选中nid只是万里长征的第一步，后面还有许多事情要做呢，这些事情没做完我们就不能删除，如果中途失败，那么咱这个nid算是分配出去了，到哪儿找呢！所以要给自己回旋的余地啊，f2fs的代码巧妙地把此时free\_nid的状态从NID\_NEW变成了NID\_ALLOC，什么时候完全从链表中删除呢？

函数：alloc\_nid\_done ! 其中，狠心的\_\_del\_from\_free\_nid\_list，会将这个节点从链表中删除，并删除基数树中的索引。然后其在slab中的位置也被释放了！





到这里我们基本了解了free\_nid的一生。并且也基本猜到了他的一个作用：为node\_info的出场暖场呢！

## 10.4 Node管理器中的node\_info

free\_info 功成身退，node\_info顺利接班。

// 这里还是蛮复杂的一件事，如果不搞清除的话，这个历史性的接班工作我们就接不上

上面说到 alloc\_nid 和 alloc\_nid\_done 之后，这个free\_nid算是被完全清除了。我们知道free\_info 和 node\_info 结构体的桥梁就是nid了，那么我们看下这个nid干什么用了！搜alloc\_nid，主要有两处：1）创建一个新的inode【函数f2fs\_mkdir, f2fs\_create】；2）创建文件的索引的时候：【函数 get\_dnode\_of\_data】。

我们发现这两条线到最后都不约而同地调用了接口new\_node\_page

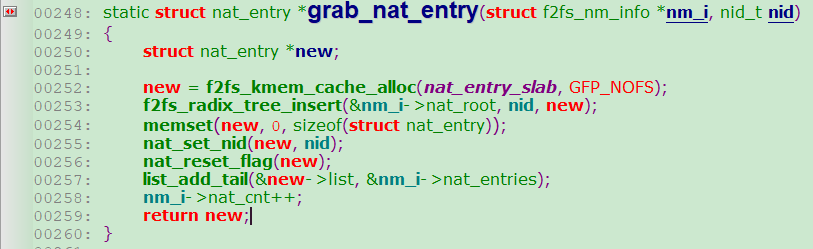
1）\_\_f2fs\_add\_link --> init\_inode\_metadata --> new\_inode\_page --> new\_node\_page --> grab\_cache\_page

2）get\_dnode\_of\_data --> new\_node\_page -->grab\_cache\_page

到这里，我们知道page中这个nid的页就建立起来了，这个得到这个页alloc\_nid的工作也算是完成了，那么node\_info中blk\_addr是怎么设置的呢？

我们自动想到了address\_spacing中的写回的钩子：f2fs\_write\_node\_page；

在f2fs\_write\_node\_page函数中，nid的信息首先会通过get\_node\_info 函数得到，但是由于node管理器中的nat\_root树中并没有相关的nid的信息，所以此时就会在最后的cache\_nat\_entry中建立啦！可见此时打配合的两个管理结构是：1）基数树，nat\_root；2）链表，nat\_entries。



但是根据日志文件系统的原理，我此时此刻还是不能确定最后的这个nid的节点要落到何处。

但是函数f2fs\_write\_node\_page，继续向下执行，就会有set\_node\_addr函数，将nat\_root中的地址填充完整，于是node\_info完整了。

好了，到这里我们明白了node\_info 与 free\_nid 这个历史性交接过程：一个nid如何从free\_list 到 nat\_entries中去了。

## 10.5 Nid如何从nat\_root中删除

上面我们谈到了一个nid如何从free\_nid中转移到node\_info中去【分别有一个链表和一棵基数树搭伙做事】，讲free\_nid时，详细说明了free\_nid中是如何进如何出的，上一篇说了nid是如何进入nat\_root的，当然一个nid进入nat\_root的情况不止如此呢，还包括如果读磁盘中已经存在的一个nid，这个nid的缓存也是存在这里的。

那么现在说说nat\_root中的nid是如何刷回去的！

涉及一个函数：\_\_del\_from\_nat\_cache

两个地方调用：

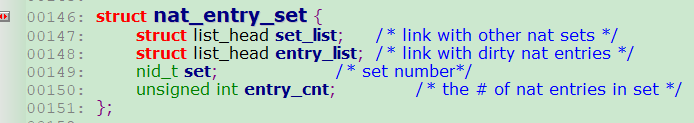
1）f2fs\_balance\_fs\_bg --> try\_to\_free\_nats --> \_\_del\_from\_nat\_cache 【内存紧张的时候，直接丢弃】

2）destroy\_node\_manager --> \_\_del\_from\_nat\_cache 【写checkpoint】

对于第一种情况，有个疑问：如果这个nat\_entry并没有写回磁盘那怎么办？！

这个情况不会发生的，你会发现在函数 f2fs\_write\_node\_page 中当nat中设置一个新地址时，会调用\_set\_nat\_cache\_dirty，将其从本来待的nat\_entries链表中删除，然后放入nat\_set\_root中这个block对应的那个list链表中去。是不是很拗口？

解释一下：nat\_set\_root是我们今天要看的第三棵基数树了，也是最后的一棵树了，杀千刀的node管理器终于要见底了，这里是一个nid的最后归宿啦！nat\_set\_root是棵什么树呢？这棵基数树的索引是这个nat\_entry所在的nat块，这个块中所有的所有的nat\_entry都链到在他的entry\_list中的！都是脏的块哇！有没有！



上面标蓝的第一条线，在丢弃这些cache之前，要用段管理器中的锁的：nat\_tree\_lock。 down\_write(&nm\_i->nat\_tree\_lock) 和 up\_write(&nm\_i->nat\_tree\_lock) 要套起来，set\_node\_addr中也要有这个锁啊，果然，这个函数一上来，这个锁就套上了！

至此，我们知道了nid去哪儿了！原来，nat\_root，只是nid暂时的一站，真的是暂时的一站，立马，你就去了就到了nat\_set\_root中自己的那个nat\_entry\_set中去了！

好了，就这样了！nat\_entry\_set中的东西到write\_checkpoint的时候清除了！

down\_write(&nm\_i->nat\_tree\_lock);