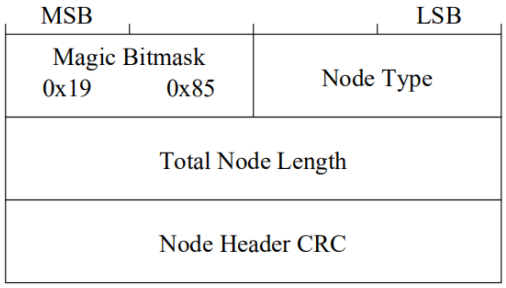
Jffs2 文件系统综合分析

1190201215
冯开来

大致简单来说，JFFS 是一个专门用在嵌入式系统闪存设备的日志结构的文件系统。因为现在的闪存技术，许多文件系统不能直接作用于闪存芯片，因此会导致两个日志记录文件系统相互连接影响效率，而 JFFS 避免了这一点。

节点类型和兼容性

最初的 JFFS 在介质上只有一种类型的节点，但 JFFS2 更灵活，它允许定义新的节点类型，同时通过使用受 ext2 文件系统的兼容性位掩码启发的方案来保持向后兼容性。每种类型的节点都以一个公共标头开始，其中包含完整的节点长度、节点类型和一个循环冗余校验和(CRC)。常见的头部结构如下图所示：



除了有一个唯一的数值标识节点结构和意义，节点类型字段还包含最重要的两个位的位掩码，它表示一个内核要求做出的行为但是这个行为并不被节点类型支持。比如：

JFFS2_FEATURE_INCOMPAT-在找到具有不受显式支持的此特性掩码的节点时，JFFS2 实现必须拒绝挂载该文件系统。

JFFS2_FEATURE_ROCOMPAT-具有此特性掩码的节点可以被不支持它的实现安全地忽略，但文件系统不能被写入。

JFFS2 FEATURE RWCOMPAT DELETE-具有此掩码的不支持的节点可能被安全忽略，并可能写入文件系统。在垃圾收集找到它的扇区时，应该删除该节点。JFFS2 FEATURE RWCOMPAT COPY-具有此掩码的不支持的节点可能被安全忽略，并可能写入文件系统。在垃圾收集找到它的扇区时，节点应该被完整地复制到一个新的位置。

不幸的是，这种兼容性位掩码实际上是导致旧 JFFS2 文件系统的兼容性被破坏的原因。最初，CRC 被省略了公共节点头，并发现因为完整的特性掩码比其他位设置，相对容易，通过中断擦除，意外地生成一个结构媒体看起来像一个未知的节点的特性位集。因此，除了节点内容和数据或名称字段上的现有 CRC 外，还在节点头上添加了一个 CRC。

日志的布局和块列表

除了单个节点的差异外，JFFS2 的高级布局也改变了原来单一的循环日志格式，这是由于严格的垃圾收集造成的问题。在 JFFS2 中，每个擦除块都是单独处理的，节点可能不会像在原始 JFFS 中那样重叠擦除块边界。

这意味着垃圾收集代码可以通过一次从一个块收集垃圾，并智能地决定从下一个块收集垃圾来提高工作效率。

每个擦除块可能处于许多状态之一，主要取决于它的内容。JFFS2 代码保留了许多表示单个擦除块的结构链表。在 JFFS2 文件系统的正常操作过程中，大多数擦除块将在干净列表或脏列表上，它们分别表示充满有效节点的块和包含至少一个过时节点的块。在一个新的文件系统中，许多擦除块可能在自由列表中，并且将只包含一个有效的节点——该标记显示块已正确和完全擦除。

垃圾收集代码使用这些列表来为垃圾收集选择一个扇区。为了确定哪个块应该被选择，有一种非常简单的概率方法——基于 jiffies 的计数器。如果 jiffies100%非零，那么则从脏列表中获取一个块。否则，在公式为零的百分之一的情况下，将从干净列表中抽取一个块。通过这种方式，我们优化了垃圾收集，以重用已经部分过时的块，但随着时间的推移，我们仍然足够好地在媒体上移动数据，以确保没有一个擦除块会在其他块之前磨损。

节点类型

相比于 JFFS1，JFFS2 中的第三个主要变化是目录条目和内部节点之间的分离，这允许 JFFS2 支持硬链接，也消除了重复名称信息的问题。在编写时，JFFS2 定义和实现了三种类型的节点。具体情况如下：

JFFS2 NODETYPE INODE - 这个节点最类似于 JFFS v1 中的结构体 jffs_raw_inode。它包含所有的 inode 元数据，以及可能属于 inode 的一系列数据。但是，它不再包含文件名或父 inode 的编号。与传统的类似 unix 的文件系统一样，innode 现在是与目录条目完全不同的实体。当引用它的最后一个目录条目被取消链接时，inode 将被删除，并且它没有打开的文件描述符。
附加到这些节点上的数据可以进行压缩。最简单的类型是“无”和“零”，这意味着数据是未压缩的，或者数据都为零。两种压缩算法是专门为 JFFS2 开发的，JFFS2 代码还可以包含 zlib 压缩库的另一个副本，它已经在 Linux 内核源代码的至少三个其他地方存在。

为了便于在执行 readpage()请求时快速解压缩数据，节点包含不超过一页数据。这意味着在某些情况下，JFFS2 文件系统映像不能在主机之间移植。但这并不是一个严重的问题，因为闪存介质的性质使得设备之间的传输不太可能。JFFS2 在存储大于单个字节的数字时也完全是主机环境化的。

JFFS2 NODETYPE DIRENT-此节点表示一个目录条目，或一个到 inode 的链接。它包含要找到该链接的目录的 inode 号、该链接的名称和该链接所引用的该目录的 inode 号。每个节点中的版本号是父节点的序列。通过写入一个名称相同但目标编号为 0 的不同节点来删除链接。

POSIX 要求在重命名时满足原子性，例如“passwd.new”变为”passwd”，任何时候对该名称的查找都不能返回旧目标或新目标。JFFS2 满足这一要求。但是整个重命名操作却和其他许多文件系统一样，不是原子性的。

重命名分两个阶段执行。首先编写一个新的不同节点，重命名 inode 的新名称和 inode 编号。这在原子上用新的链接替换了到原始 inode 的链接，并且与创建硬链接的方式相同。接下来，通过编写一个包含原始名称和目标编号为零的不同节点，取消链接原始名称。

这个两个阶段的过程意味着，在重命名操作期间的某个时候，可以通过旧名称和新名称访问被重命名到位的 inode。POSIX 的原子性要求保证了仅针对目标

链路的行为。

JFFS2 NODETYPE CLEANMARKER-此节点被写入一个新擦除的块，以显示擦除操作已成功完成，并且该块可以安全地用于存储。

最初的 JFFS 简单地认为在第一次扫描时出现的每个字节中包含 0xFF 的块都是自由的，并且将在挂载时选择最长的明显空闲空间运行作为日志的头和尾部之间的空间。不幸的是，在 JFFS 上进行的广泛的功率失败测试证明了这并不好。对于许多类型的闪存芯片，如果在擦除操作过程中电源丢失，一些位可能会处于不稳定状态，而大多数位则会被重置为逻辑位。如果初始扫描读取所有和处理块包含等不稳定位可用，那么数据可能会丢失数据丢失。

经验结果表明，即使多次重读块的整个内容以试图检测不稳定的位，也不够可靠地避免数据丢失，因此需要另一种方法。可接受的解决方案是，在成功完成擦除操作后，立即将标记节点写入闪存块。

当遇到似乎不包含任何有效节点的闪存块时，JFFS2 将触发一个擦除操作，并随后将适当的标记节点写入已擦除的块。

该节点类型是在 JFFS2 开始在实际应用程序中使用后引入的，并使用 RWCOMPAT_DELETE 删除特性位掩码来表示较旧的 JFFS2 实现可以安全地忽略该节点。

操作

JFFS2 的操作和 JFFS 非常相似。尽管现在有各种类型的节点，但会按顺序写入，直到填充一个块，此时从自由列表中提取一个新块，并从新块的开始继续写入。

当空闲列表的大小达到启发式阈值时，垃圾收集就开始了，将节点从旧块移动到新块中，直到可以通过删除旧块来回收空间。

但是，JFFS2 并没有始终将核心内存中的所有 inode 信息都保存在核心内存中。在装载过程中，完整的映射会像以前一样构建——但是保存在内存中的结构被严格限制在不能按需快速重新创建的信息中。对于介质上的每个 inode，都有一个结构体 jffs2inode 缓存，它存储其 inode 编号、当前到 inode 的当前链接数，以及一个指向属于该 inode 的物理节点的链接列表的开始的指针。这些结构存储在一个哈希表中，每个哈希桶都包含一个链接列表。哈希函数是一个非常原始的函数——仅仅是调制哈希表的大小的 inode 数。inode 数的分布意味着这应该是具有良好分布的。

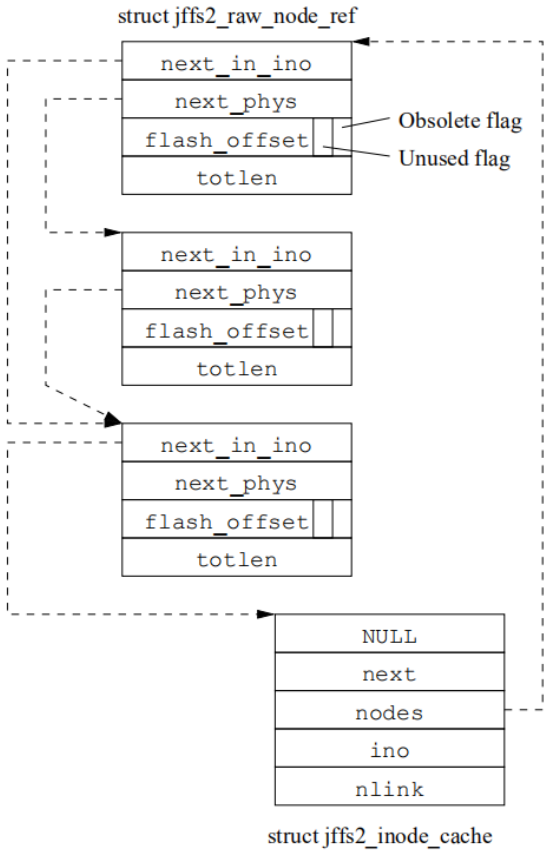
每个物理节点由一个较小的结构 jffs2_raw_node_ref，原始节点参考表示，如图 2 所示，其中包含两个指向其他原始节点引用的指针——物理擦除块中的下一个，以及每个节点列表中的下一个，以及节点的物理偏移量和总长度。由于这种结构的数量和许多嵌入式系统上可用的 RAM 数量有限，这种结构的大小非常有限。

因为介质上的所有节点都以四个字节的粒度对齐，所以闪光偏移字段中最不显著的两位是冗余的。因此，它们可以作为额外的标志使用。设置最小显著位表示所表示的节点是过时的节点，另一个节点尚未使用。

对于垃圾收集，如果给定一个原始节点引用，则必须找到它所属的 inode。最好不要在每个这样的结构中添加包含这些信息的四个字节，所以我们会用指针玩更。最后一个原始节点引用实际上包含了一个指向相关 jffs2_inode_cache 缓存的结构体的指针。因为这种结构在 struct jffs2_rao_node_ref 成为下一个节

点指针有一个偏移。inode 缓存结构中显示的 NULL 字段仅在用于临时存储的初始扫描时使用，因此可以保证在正常操作时为 NULL。

在正常操作期间，文件系统的 read_inode()方法将传递一个 inode 编号，并期望使用适当的信息填充一个结构 inode。JFFS2 使用 inode 数字查找适当的结构 jffs2inode 缓存哈希表，然后使用节点列表直接读取每个节点属于所需的 inode，从而建立一个完整的映射到每个范围的数据，类似于 JFFS 的信息会保持在内存中即使是未使用的。下图是节点的参考列表：



装入文件系统

载入 JFFS2 文件系统涉及一个四个阶段的操作。首先，扫描物理介质，检查所有节点上的 crc 的有效性，并分配原始节点引用。在此阶段，还将为每个找到有效节点的 inode 缓存结构为 inode 分配并插入到哈希表中。

其他来自 flash 上节点的缓存信息，如节点所覆盖的版本和数据范围，以防止挂载过程的后续阶段不得不再次从物理介质中读取任何内容。

物理扫描完成后，首先通过所有物理节点，为每个链接节点构建一个完整的数据映射映射，这样就可以检测到过时的节点，并增加每个有效的目录输入节点的链接节点缓存中的 nlink 字段。

然后进行第二次传递，以查找在文件系统上没有剩余链接的内部节点并删除它们。每次删除目录节点时，传递都会重新启动，因为其他节点可能是孤立的。将来，可能会修改这种行为，将孤立的内节点存储在 lost+dind 目录中，而不是仅仅删除它们。

最后，第三次释放为每个 inode 缓存的临时信息；只留下通常保存在结构 jffs2inode 缓存中的信息。在此过程中，将 inode 缓存中对应于原始节点引用

的 inino 字段的字段被设置为 NULL，从而达到前面提到的非法侵入。
垃圾回收

在 JFFS2 中，垃圾收集通过确定要被垃圾收集的节点所属的 inode，并为该 inode 调用 Linux 内核的 iget()函数来移动数据节点。通常，inode 将在内核的 inode 缓存中——但有时，这将导致调用 JFFS2 的 read_inode()函数。JFFS2 非常需要的特性之一是垃圾收集算法正确性的正式证明。目前的经验方法是不够的。然而，压缩给这个证明造成了一个严重的潜在问题。如果一个完整的页面写压缩非常好，之后一个字节写在页面的中间减少了页面的可压缩性，然后当垃圾收集原始页面我们可能会发现新节点写出比原来的大。因此，将没有办法对垃圾收集装满数据的擦除块所需的空间量设置一个上限。

提出的解决方案是允许版本字段的总排序被放宽到一个标准排序。我们允许两个节点具有相同的版本字段，只要它们具有相同的数据。因此，当垃圾收集发现一个将扩展的节点，但没有足够的松弛空间允许它这样做，它可能完整地复制原始节点，服务原始版本，以便覆盖其中包含的数据的节点将继续这样做下去。

截断和 file holes

在 JFFS2 的设计阶段出现的一个问题，对于原始版本尚未得到解决，涉及到文件的截断。可能出现问题的事件序列是一个截断，然后对大于截断点的偏移量进行写入——在文件中留下一个“孔”，在读取时应该返回所有的 0。在截断时，原始的 JFFS 只是写出一个提供新长度的新节点，并将（在内存中）包含在截断点之外的数据的旧节点标记为过时。以后的写入操作也会正常进行。

在重新装入时扫描文件系统时，垃圾收集的顺序性质确保了所有包含这些范围的实际数据的旧节点在截断节点之前被垃圾收集。由于节点在物理扫描后按照版本顺序进行解释，因此可以保证正确的行为，因为在旧数据被删除之前，截断的证据一直存在

对于 JFFS2，块可以被垃圾收集，有必要确保旧数据永远不会“显示”由文件的截断和后续扩展造成的漏洞。

出于这个原因，我们决定不应该有适当意义上的漏洞——完全没有有关字节范围的信息。相反，在接收到写入大于文件当前大小的偏差或截断到更大大小的请求时，JFFS2 插入具有前面提到的压缩类型 JFFS2compr 零的数据节点，这意味着节点不包含实际数据，并且读取时，节点表示的整个范围应设置为零。

如果文件包含一个非常大的孔，最好只用介质上的一个物理节点来表示该孔，而不是为受影响范围内的每个页面表示一个“孔”节点。因此，这种孔节点是数据节点的一种特殊情况；唯一可以覆盖一页以上范围的数据节点类型。这种特殊情况本身是进一步复杂的原因，因为人们担心在垃圾收集过程中进行扩展。如果将单个字节写入以前属于孔的页面，则必须确保原始孔节点或包含数据新字节的节点的垃圾收集不需要比原始节点占用更多的空间。
这个问题的解决方案与压缩节点在合并时相同，压缩节点可能会扩展——如果垃圾收集会导致扩展，并且没有足够的松弛空间来容纳这种增长，那么原始节点将被完全复制，并保留原始版本号。