16 | 日志型文件系统：写入文件的时候断电了会发生什么？

2022-01-15 黄清昊

《算法实战高手课》 课程介绍 >

讲述：黄清昊
时长 15:12 大小 13.93M

你好，我是微扰君。

今天我们就来聊一聊操作系统最常见的外存——磁盘的问题。我们知道计算机的内存一旦断电，数据就会全部丢失，所以如果需要持久化一些数据，磁盘就是必不可少的硬件，甚至在计算机上运行的整个操作系统的大部分代码逻辑，其实也是存储在磁盘中的。

计算机要和磁盘打交道，就需要用到文件系统。

文件系统，其实就是操作系统中用于管理文件的子系统，它通过建立一系列诸如文件、目录，以及许多类似于 inode 这样的元数据的数据结构，把基础的文件访问能力抽象出来，给用户提供了一套好用的文件接口。

和一般的数据结构和算法主要考虑性能不同，文件系统还需要考虑一件非常重要的事情——数据的可持久化。因为文件系统一定要保证，计算机断电之后整个文件系统还可以正常运作，只要磁盘没有损坏，上面的数据在重新开机之后都可以正常访问。

这件事听起来感觉很简单，但是真正实践起来可要难得多，在过去几十年里为了解决各种各样不同的问题，文件系统层出不穷，今天我们就来讨论其中一个问题：**写文件写到一半断电了，或者因为各种各样的原因系统崩溃了，系统重启之后文件是否还能被正常地读写呢？如果不能的话，我们应该怎么办呢**？

这个问题，我们一般叫崩溃一致性问题（crash-consistent problem）。目前最流行的解决方案是 Linux 中的 Ext3 和 Ext4 文件系统所采用的日志方案，也就是 journaling，而 Ext3 和 Ext4 自然也就是所谓的日志型文件系统。

崩溃一致性

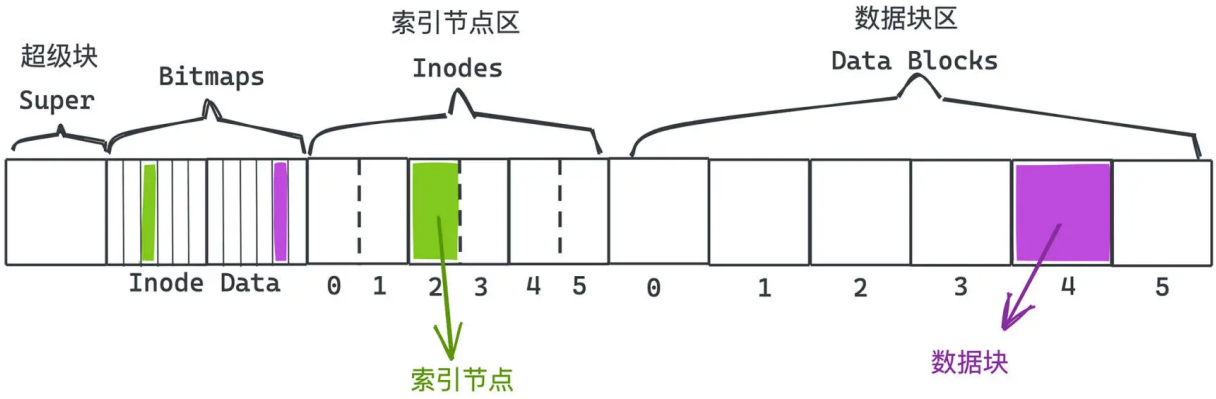
在讲解问题的具体解决方案之前，我们还是得先来认真审视一下崩溃一致性问题的本质：**写文件的时候我们具体做了哪些事情呢？崩溃后为什么可能会产生一致性问题？**

这自然也就关系到文件在系统中到底是如何存储的了。不过，不同的文件系统对文件的组织方式千差万别，我们会以 Ext4 中的存储方式为例，带你简单了解演化了许多年之后现在主流的文件系统是如何存储数据的。

目前最主流的持久化存储介质还是磁盘。限于磁盘的物理结构，它读写的最小单位是扇区，大小是 512B，但是每次都只读一个扇区，不利于读写效率的提升；所以文件系统普遍会把多个扇区组成一个块，也就是 block。在 Ext4 中，逻辑块的大小是 4KB，包含 8 个扇区。

我们的数据自然也就是放在这一个个数据块上的。不过和内存一样，**磁盘空间大小虽然大的多，但仍然是有限的，我们需要为不同的文件划分出自己的区域**，也就是数据具体要存储在哪些块上的。

为了更灵活地存储文件、更高效地利用磁盘空间、更快速地访问到每个文件的数据存储在哪些块上，Linux 的做法是把文件分成几块区域：至少包括超级块、索引节点区、数据块区。



极客时间

超级块，是文件系统中的第一个块，用来存放文件系统本身的信息，比如可以用于记录每块区域的大小；

索引节点区，每个文件对应索引节点区中的一个块，我们称为索引节点，也就是 Inode，存放每个文件中所用到的数据块的地址，Inode 也是元数据主要存储的地方；

数据块区，也就是 Data Blocks，这里是真实数据存放的区域，一个文件的 inode 可能存有多个指向数据块的指针。

另外，为了标记哪些 Inodes 和 Data Blocks 可以被使用，操作系统还建立了两块存放 Bitmap 的区域。

这样我们就可以非连续地表示各种大小的文件了，因为在索引节点上就会有很多指针链到数据区的不同区块；我们也就可以快速地获取文件所需要的数据内容了，只要在访问文件的时候根据索引节点中的指针和操作系统的磁盘调度算法，读取文件中数据块中的内容即可。

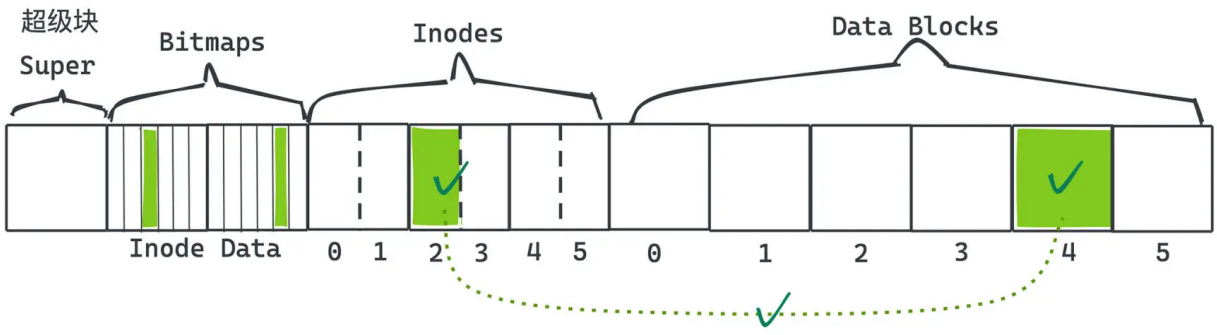
引入元数据的问题

但是在文件系统中引入了元数据带来了灵活性的同时，也带来了问题。

现在每个文件都对应一个 Inode，它记录了所有数据块的位置，可以预想到，之后在修改、创建文件的时候，除了修改数据块区的内容，也需要修改文件的元数据和 Bitmaps。比如，当我们往某个文件里追加数据，很可能就需要创建新的数据块把数据写入其中，并且在 Inode 上追加一个指向这个数据块的指针。

总之这么看，**每次写文件的操作其实都是一个操作序列，而不是一个单一的操作**。但是，磁盘在同一时间里肯定只能接受一次读写的请求，因此文件系统就引入了崩溃一致性问题。

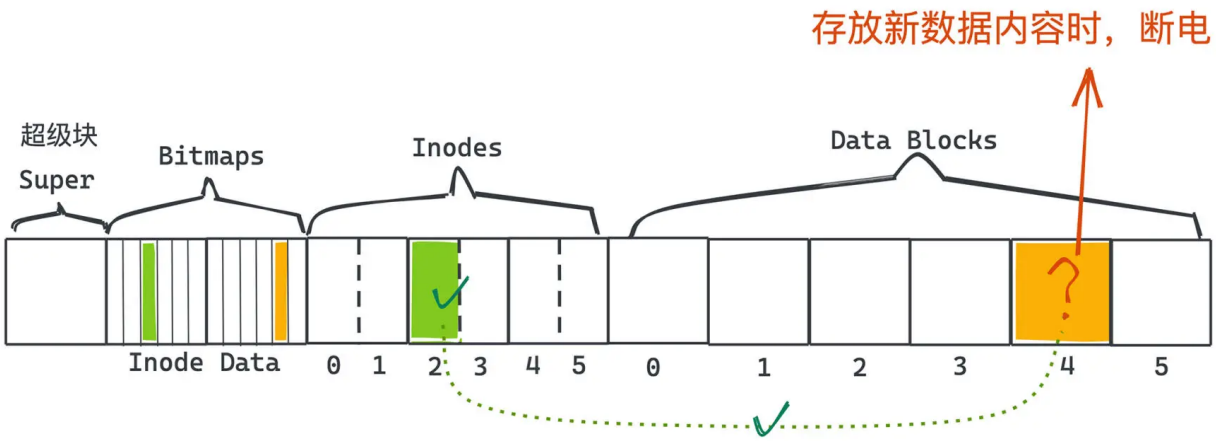
这很好理解，我们看一个具体的例子。



极客时间

已知我们写一个文件至少会碰到 Bitmaps、Inodes 和 Data Blocks 三块数据的修改，在不会遇到崩溃的时候，我们可能就像这张图一样顺利修改了每个部分，此时文件系统没有任何问题。

但假设，修改完 Inodes 了，我们把 Inodes 中某个指针指向了一段即将要存放数据内容的数据块，此时如果遭遇了断电，Data Blocks 上的内容是未知的，很可能是很早之前别的程序写过的数据，我们可以认为此时上面的数据是脏的垃圾数据。等系统恢复，我们重新去读取文件数据的时候，会发现也没有什么有效的依据供我们检验文件是否正常。



极客时间

在这种情况下，Inodes 指向的文件内容和我们的预期就会不一致，从而产生文件损坏的情况。其实和我们平时说的事务性、原子性是类似的场景。

这也只是不一致的一种情况。事实上，因为操作系统会对磁盘读写的顺序做调度，以提高读写的效率，我们其实不能知道这几个写磁盘的独立步骤确切的执行顺序。

不过可以想见，在其他几种写 Inodes、Data Blocks 和 Bitmaps 的顺序下，**如果没有执行完全部步骤就遭遇了断电等情况，文件系统在大部分时候仍然都会进入不正确的状态，这就是崩溃一致性问题**，如果你感兴趣可以模拟其他几种情况。

如何解决

现在，我们搞清楚了崩溃一致性问题的本质，自然就要尝试解决它。历史上比较流行的解决方案有两种：一种是早期操作系统普遍采用的 FSCK 机制（file system check），另一种就是我们今天主要学习的日志机制（journaling file system）。

我们先从早期的 FSCK 机制学起。

解决方案 1：FSCK

FSCK 机制的策略很简单：错误会发生，没关系，我们挂载磁盘的时候检查这些错误并修复就行。

比如，检查发现 Inodes 和 Bitmaps 不一致的时候，我们选择相信 Inodes，而更新 Bitmaps 的状态；或者当两个 Inodes 都指向同一个 block 时，我们会把其中明显是异常的一个 Inodes 移除。

但因为崩溃，毕竟有一部分信息是丢失了的，所以很多时候我们也没有办法智能地解决所有问题，尤其是前面说的 Inodes 指向脏数据的情况，事实上这种情况下，FSCK 没有办法做任何事情，因为它本质上只是让文件系统元数据的状态内部保持一致而已。但是，这还不是 FSCK 最大的问题。

FSCK 真正的问题是，每次出现问题需要执行 FSCK 的时候的时间非常久。

因为需要扫描全部磁盘空间，并对每种损坏的情况都做校验，才能让磁盘恢复到一个合法的状态，对普通的家用电脑来说，很可能需要长达几十分钟甚至几小时的时间。所以，现在 FSCK 基本上已经不再流行，取而代之的就是日志型文件系统。

解决方案 2：Journaling

日志型文件系统这个方案，其实是从 DBMS 也就是数据库系统中借鉴而来的。

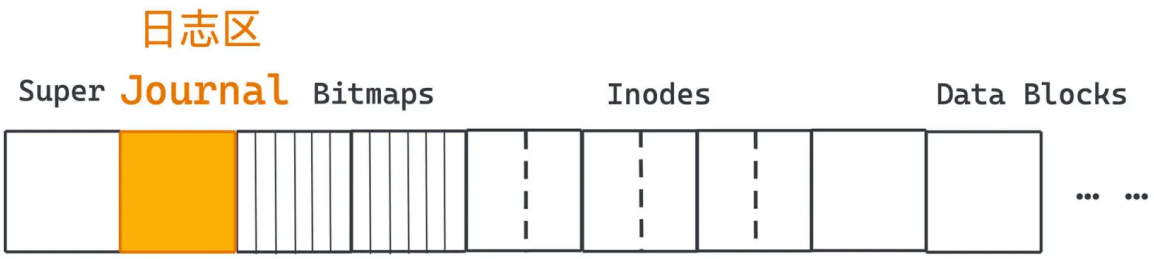
journaling file system 的核心思想是鼎鼎大名的预写日志 WAL 也就是 write-ahead logging，这个也正是数据库系统中用于实现原子事务的主要机制，也很好理解，毕竟事务和文件系统一样，都需要保证一致性。

那具体是怎么做的呢？

思想也很简单，就是**每次在真正更新磁盘中的数据结构之前，我们把要做的操作先记录下来，然后再执行真正的操作，这也就是先写日志，WAL 中 write-ahead 的意思**。这样做的好处在于，如果真的发生错误的时候，没有关系，我们回去查阅一下日志，按照日志记录的操作从头到尾重新做一遍就可以了。

这样我们通过每次写操作时增加一点额外的操作，就可以做到任何时候遭遇崩溃，都可以有一个修复系统状态的依据，从而永远能恢复到一个正确的状态。

对于文件系统的布局，我们也只是需要增加一块区域用于存放日志就行，改动不是很大，这块区域我们叫做日志区（Journal）。



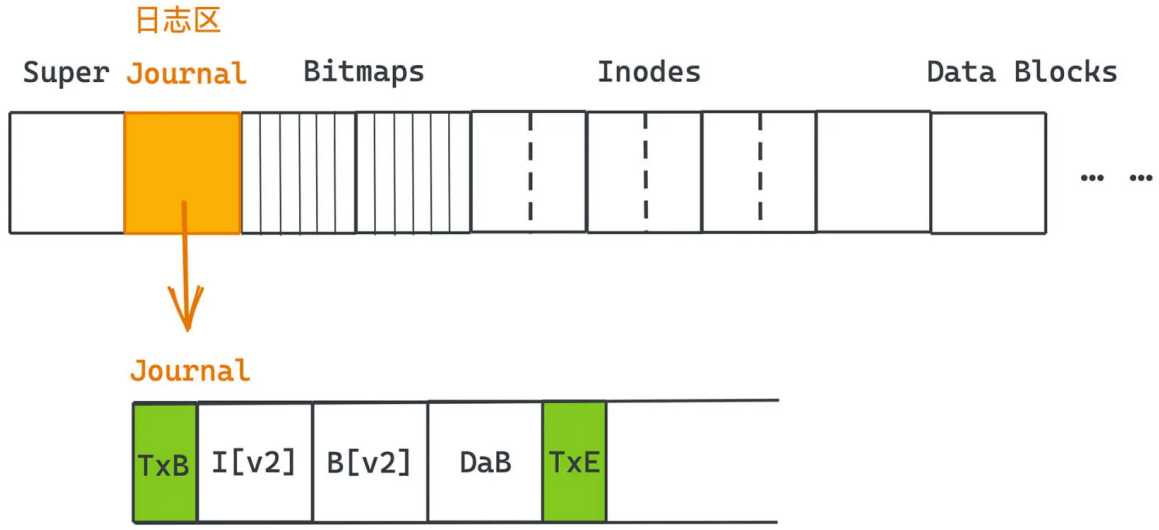
极客时间

那 Journal 区里到底要存放点什么样的内容呢？

这里会涉及一次完整的写文件对磁盘的一系列操作，你不熟悉的话也没有关系，就当这是几个独立的操作就行。**我们核心就是要实现：希望可以通过某种记录日志的方式，让这些操作一旦决定被提交，即使后续对磁盘上元数据和数据块上数据结构的改动进行到一半，系统断电了，仍然可以根据这个日志恢复出来。**

那要怎么做呢？和数据库一样，我们为了让一系列操作看起来具有原子性，需要引入“事务”的概念。

我们每次进行一次对文件的写操作，除了会先在预写日志中，记录对 Inodes 的修改记录、对 Bitmaps 的修改记录，以及对具体数据块的修改记录之外，还会同时在这几条记录的前后，分别引入一个事务开始记录和一个事务结束记录。

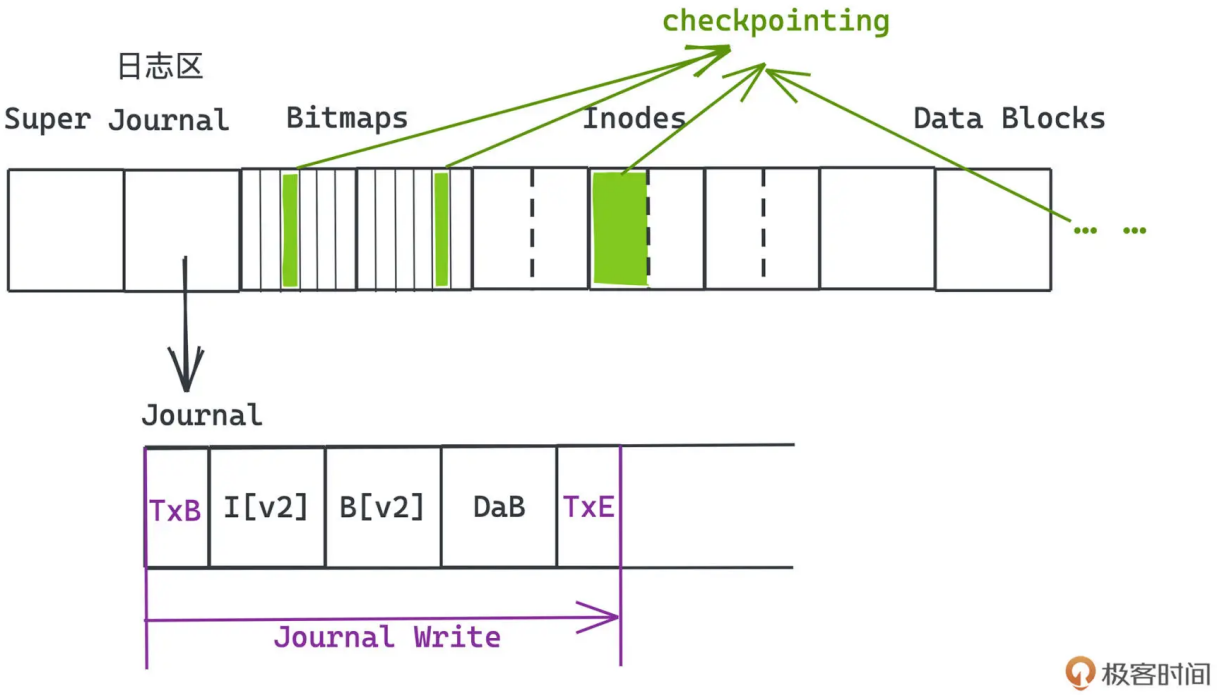


第一个写入的记录是 TxB，也就是 Transaction Begin 记录，最后一个写入的记录就是 TxE 也就是 Transaction End 记录，**在 TxE 记录完成后，就意味着整个写文件的操作全过程都被记录在案了，我们把这个步骤叫做 Journal Write**。

从而，日志的组织形式就是由这样一个个事务拼接而成，日志的首尾是 TxB 和 TxE 块，中间是具体对元数据和数据块修改的记录块。

当我们完成 Journal Write 操作之后，就可以放心大胆地把这些实际的元数据或文件数据覆写到磁盘上对应的数据结构中了，这个步骤我们叫做 checkpointing。

当这个步骤也成功完成，我们就可以说整个写文件的操作被完成了，在全部成功的情况下当然没什么特别的，**但整个设计的关键之处就在于，面对任意时刻崩溃的情况，我们也能把文件系统恢复到某一合法状态的能力**。



我们具体看看崩溃出现的时候，引入了 journaling 的文件系统会有什么不同吧。

如果崩溃出现在 journal write 步骤中

假设崩溃是出现在 TxE 块完成写操作之前，那其实对系统也没有任何影响。因为相当于事务没有被成功提交，而我们写的是日志，对文件本身也没有任何实际影响。

当系统断电又恢复之后，只要发现某个事务 ID 没有对应的 TxE 块，说明这个事务没有提交成功，不可能进入 checkpointing 阶段，丢弃它们对文件系统没有任何不良影响，只是相当于上次写文件的操作失败了而已。

如果崩溃刚好发生在 journal write 结束之后

不管是刚刚写完 TxE，还是已经进入了 checkpointing 的某一步，我们的处理也都是一样的。既然事务已经被提交，系统断电恢复之后，我们也不用关心之前到底 checkpointing 执行到了哪一步，比如是已经更新了 inodes？还是 bitmaps？都没有任何影响，直接按照日志重做一遍就可以，最坏的下场也不过就是重新执行了一遍执行过的操作。

顾名思义，这种重做一遍的日志我们也把它称为 redo logging，它也是一种最基本、最常见的日志记录方式。不只在文件系统中，在数据库等场景下也使用非常广泛。

总结

我们就通过引入预写日志的手段，完美解决了操作系统文件状态在崩溃后可能不一致的问题，相比于从头到尾扫描检查的 FSCK 机制。预写日志，在每次写操作的时候引入一些额外的写成本，让文件系统始终得以始终处于一种可以恢复到一致的状态，如果崩溃，只需要按照日志重放即可。

当然我们其实还有很多优化可以做。比如，可以进行批量日志的更新，把多个独立的文件写操作放到一个事务里提交，提高吞吐量；或者记录日志的时候只记录元数据，而不记录文件写操作的大头数据块等等。如果你仔细看 Linux 文件系统的实现就会发现，做的优化非常多，集结了许多前人的智慧，我们可以从中领略到很多思想，也许有一天在你的工作中，这些想法就会成为你解决一些问题的关键。

以我们今天学习的“日志”思想为例。我曾经在一家做安全硬件的公司实习，当时就有个需求要写一段代码用单片机往一个类似闪存的芯片里写一些状态，包含好几个独立的字段，每个字段都需要顺序地写，也就是说要分好几次独立的操作去写。

但是，单片机是可能随时可以掉电的，我们如何保证这个闪存中的状态是正确的，而不是状态的某几个字段是上次写的，某几个字段是这次写的呢？其实这个问题我们就可以用类似日志的思想去解决。

一种可行的方式就是，在闪存中开辟一段额外的空间，先预写日志，用 TxB 和 TxE 来标记一次完整的状态，每次启动时候先检查日志，把最新的状态覆写到指定的地址区域即可。当然事实上解决的办法要更简单一些，也更省空间，欢迎在留言区和我一起讨论。

课后练习

最后布置一个思考题，文件系统为了提高读写吞吐，实际在写日志的时候也可以通过调度，调整写不同块的先后顺序。那我们在记录日志的时候是不是也能利用这个特性，进一步提高写文件的性能呢？如果可以的话，需要做什么限制吗？

欢迎你留言和我一起讨论。如果你觉得这篇文章对你有帮助的话，也欢迎你转发给你的好朋友一起学习。我们下节课见～

上一篇 15 | LRU：在虚拟内存中页面是如何置换的？

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。