理解EXT4，第三部分 extent树

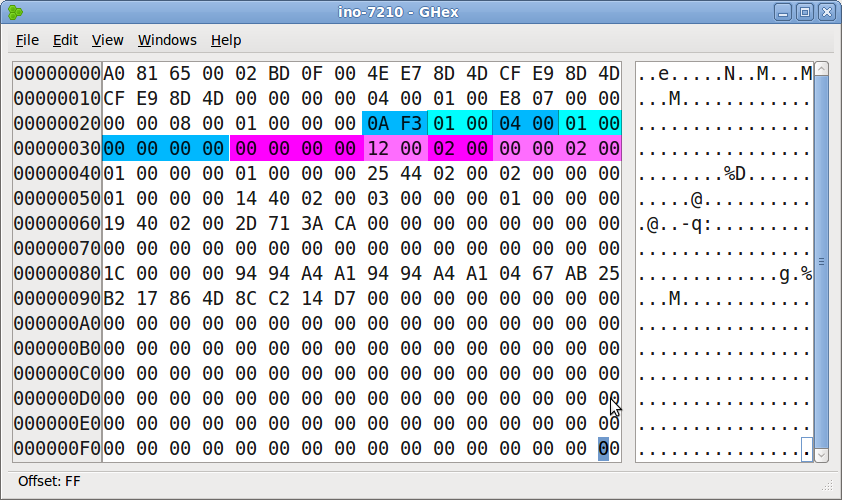
在我们解读EXT4文件系统之前，有一个很重要的概念需要讲解。在第一个部分中，提到了过在一个inode中，最多只能存4个extent结构，更进一步说，每个extent结构中只有16个bit用来保存块号，实际上，最高位被保存了下来（最高位用来表示这个externt是“保存？预分？”还是已经初始化， 部分用于EXT4的预分配功能）。这意味着，一个extent最大只能包含2的15次方个块，当一个block为4k大小的时候，即128MB。

128MB看起来足够大了，但是当你的文件大于0.5G的时候，这个文件就需要大于4个extent来保存整个block的索引。或者，当你的文件很小，但是由很多不连续的片段组成。这些情况下，就需要用大于4个extent来组织文件。

在这篇文章里，我将展示这些情况下EXT4的细节。我将使用/var/log/message文件作为范例。这个文件已经被反复追加写入了一个星期以上，并且相当的不连续。当日志文件在一个共享的目录中，并且与其他文件同时更新，就会表现出这种状态。

**分析这个inode**

使用第一部分中介绍的方法，将/var/log/message文件的inode导出，并用16进制编辑器打开。在下图中，用蓝色标出了extent的头，紫色的是第一个extent：

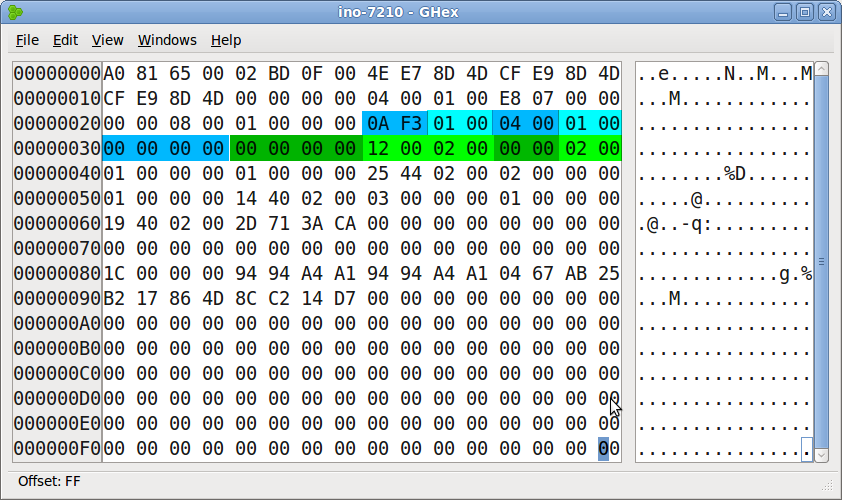


仔细观察extent的头，以magic数0xF30A起始。下一个区域告诉我们，这是一个inode中的extent。这之后的数字4，说明inode中最多只能有4个extent。之后，我们会发现与第一部分的例子中，有所区别的地方：“树深度”区为1，而不是0。最后，我们会发现generation ID一项，同样也是0。

先忽略“树深度”这一项，只看第一个extent的结构，这个extent的头告诉我们它正在被使用。对这个extent解码之后，我们可以知道它从0起始，0x0012 = 18个块的长度，这些块的起始位置是0x000200020000 = 8590065664。但是，这个块号肯定是错误的，因为这已经超过了32TB的EXT4文件系统容量的限制了，并且这台机器上的/var分区只有2GB大小。那这是为什么呢？

实际上，inode中的第一个“extent”并非标准的extent结构，上面解读extent的方法是错误的。当EXT4需要大于4个extent时，它会创建一个在磁盘上创建一个树（b树）用来保存必须的extent数据，这就是extent头上的“树深度”一项表达的含义。在树最底层的叶子节点上，放置的是规则的extent结构，就像第一部分里展示的那样。但是在树的中间节点上，是不同的结构，称为extent索引。我们已经知道，“树深度”一项为1，说明这里的并不是树的叶子。

这里有一张不同的图片，其中extent索引被标记为绿色：



这里是偏移和他们对应的区域功能描述：

Bytes 52-55: Logical block number (0x0000) 逻辑块号

56-59: Lower 32 bits of physical block address (0x00020012) 低32位的逻辑块地址

60-61: Upper 16 bits of physical block address (0x0000) 高16位的逻辑块地址

62-63: Not used

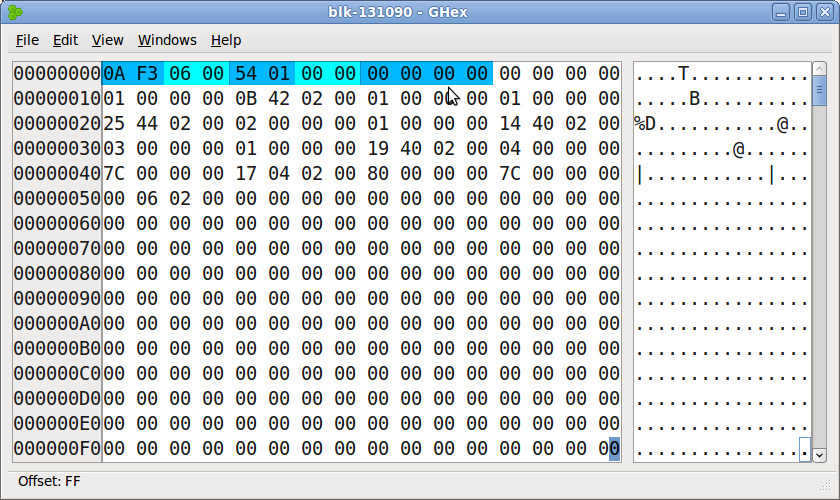
实质上，extent头包含了两个值。第一个部分是，这个extent在tree中的位置（个人理解，即以块号为单位，该extent在树起始位置的偏移）——这是extent索引结构的前4个位。在这个例子中，这个extent位于字数的0块，即意味着处于文件的开始位置。

extent索引中的剩余值是拥有下一层树信息的数据块的物理块号。就像EXT4中的块地址，这个48位的值被拆成两部分：32位的低位和16位的高位。这样，0x000000020012 = 131090，这是一个正常的块偏移。

还有16位的值在extent索引中没有被使用。你可能期望它们是0，但是在这里我们看到它被设置为0x0002。实际上，即使inode中的extent头结构声明，只有使用了第一个extent索引，你也会发现另外一个extent结构域在64~69bit处，也是非0。具体原因将会稍后一点给出，这里先看一下131090这个块。

**解读数据块**

每个用来保存extent树信息的数据块的起始位置都它自身的extent头结构，就像inode内部的extent一样。这里，16进制编辑器显示了第131090块的前256，其中extent头域被标成了高亮：

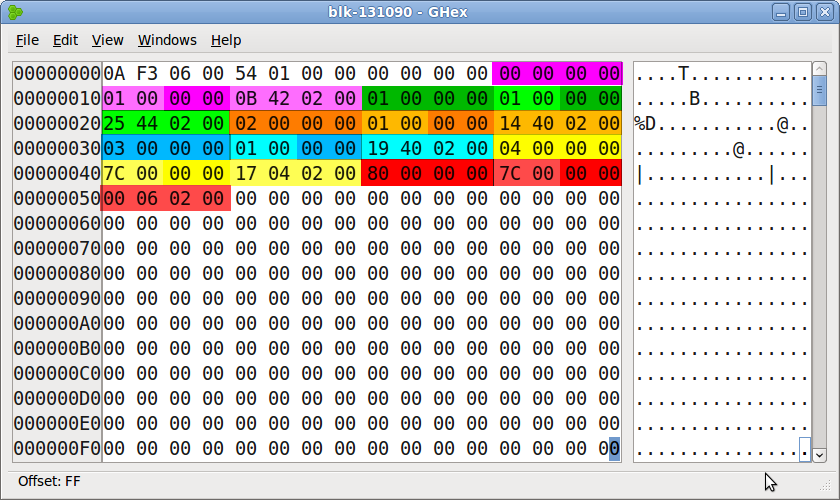


我们可以看到象征extent的magic数0xF30A，如果我们观察“树深度”域，就会发现值为0。说明我们已经在树的结构中下移了一层，这里我们看到的是规则的extent，而不是extent索引。

第2~3位告诉我们，实际上这个文件包括6个extent。但是检查第4~5位的“最大extent数量”域，其值为0x0154 = 340!这是因为我们用了一整个4k的数据块来保存extent信息。位于数据块起始部分的extent头消耗了12位，但是仍然有4084位可以用了保存extent结构。结果就是340个12位的extent结构一共占用了4080位的数据。

这看起来很不容易发生，对一个文件进行不常规的操作，导致它不连续，所以需要大于340个extent来对其进行描述。并且，每个extent最大能表示128MB的数据，340个extent会使你的表示的文件达到42.5GB。如果文件比这个还要大，我们就需要增大extent树。

但是现在，让我们对这个extent进行解读。在下图中，我们标记出6个extent：



通过上图可以解读出如下结果：

1. 逻辑块号: 0, 块数量: 1, 起始块号: 147979
2. 逻辑块号: 1, 块数量: 1, 起始块号: 148517
3. 逻辑块号: 2, 块数量: 1, 起始块号: 147476
4. 逻辑块号: 3, 块数量: 1, 起始块号: 147481
5. 逻辑块号: 4, 块数量: 124, 起始块号: 132119
6. 逻辑块号: 128, 块数量: 124, 起始块号: 132608

检查一下实际文件的情况：

# **blkcat /dev/mapper/RD-var 147979 >ext1-blks**

# **blkcat /dev/mapper/RD-var 148517 >ext2-blks**

# **blkcat /dev/mapper/RD-var 147476 >ext3-blks**

# **blkcat /dev/mapper/RD-var 147481 >ext4-blks**

# **blkcat /dev/mapper/RD-var 132119 124 >ext5-blks**

# **blkcat /dev/mapper/RD-var 132608 124 >ext6-blks**

# **cat ext\* | tr -d 0 >newmess**

# **md5sum newmess /var/log/messages**

8e8c9445d8ff3e17a22ef5a3034422a9 newmess

8e8c9445d8ff3e17a22ef5a3034422a9 /var/log/messages

首先，使用blkcat将解读出的块导出.最后一个extent将会有一定数量的空字符，这是因为文件不一定占用整个块，所以使用tr命令去掉文件最后的空字符。md5sum校验以后，显示通过dump将extent手动拷贝出来创建的文件，跟/var/log/message下的文件相符合。

**最后**

在前面，我指出了在inode中那个没有使用的extent结构看起来表示了某些数据。如果你仔细观察会发现，在inode中第64~99比特位处，保存了第2~4个extent的信息，跟第131090中的第2~4个extent相同。

我也指出了，在inode结构中的extent索引结构的高2位，通常是不用的，但是看起来有一些其中有一些数据。如果你对比一下，就会发现在extent树的索引中的最后两个字节（原文是bytes，但是我觉得有可能是bits）“02 00”，与131090块中的第一个extent结构体中的最后两个字节一致。

那其中有何奥秘？看起来ext4的程序有些懒惰。/var/log/messages这个文件不断地变大，并且别分成多个段落进行保存，所以文件系统会不断的在inode中增加extent结构。当需要使用第15个extent时，在extent头中的“树的深度”值就会被改写，inode中的第一个extent也会被extent树的索引结构取代。然而，ext4程序没有同时将inode中没有用到的extent树填0清除掉。实际上，它也没有将extent树的index同步清空。但愿这样做可以提高性能，但是看上去确实比较诡异。