yaffs2 文件系统分析

作者：armstar

1.前言

略。

2.yaffs 文件系统简介

按理说这里应该出现一些诸如“yaffs 是一种适合于 NAND Flash 的文件系统 XXXXX”之类的字眼，不过考虑到网络上关于 yaffs/yaffs2 的介绍已经多如牛毛，所以同上，略。

3.本文内容组织

本文将模仿《linux 内核源代码情景分析》一书，以情景分析的方式对 yaffs2 文件系统的源代码进行分析。首先将分析几组底层函数，如存储空间的分配和释放等；其次分析文件逻辑地址映射；然后是垃圾收集机制；接下来……Sorry，本人还没想好。:-)

4.说明

因为 yaffs2 貌似还在持续更新中，所以本文所列代码可能和读者手中的代码不完全一致。另外，本文读者应熟悉 C 语言，熟悉 NAND Flash 的基本概念（如 block 和 page）。

Ok，步入正题。首先分析存储空间的分配。

5.NAND Flash 存储空间分配和释放

我们知道，NAND Flash 的基本擦除单位是 Block，而基本写入单位是 page。yaffs2 在分配存储空间的时候是以 page 为单位的，不过在 yaffs2 中把基本 存储单位称为 chunk，和 page 是一样的大小，在大多数情况下和 page 是一个意思。在下文中我们使用 chunk 这个词，以保持和 yaffs2 的源代 码一致。

我们先看存储空间的分配(在 yaffs_guts.c 中。这个文件也是 yaffs2 文件系统的核心部分)：

```
static int yaffs_AllocateChunk(yaffs_Device * dev, int useReserve,
yaffs_BlockInfo **blockUsedPtr)
{
int retVal;
yaffs_BlockInfo *bi;

if (dev->allocationBlock < 0) {
/* Get next block to allocate off */
dev->allocationBlock = yaffs_FindBlockForAllocation(dev);
dev->allocationPage = 0;
}
```

函数有三个参数，dev 是 yaffs_Device 结构的指针，yaffs2 用这个结构来记录一个 NAND 器件的属性（如 block 和 page 的大小）和 系统运行过程中的一些统计值（如器件中可用

chunk 的总数），还用这个结构维护着一组 NAND 操作函数（如读、写、删除）的指针。整个结构体比较大，我 们会按情景的不同分别分析。useReserve 表示是否使用保留空间。yaffs2 文件系统并不会将所有的存储空间全部用于存储文件系统数据，而要空出 部分 block 用于垃圾收集时使用。一般情况下这个参数都是 0，只有在垃圾收集时需要分配存储空间的情况下将该参数置 1。yaffs_BlockInfo 是描述 block 属性的结构，主要由一些统计变量组成，比如该 block 内还剩多少空闲 page 等。我们同样在具体情景中再分析这个结构中的字段含义。

函数首先判断 dev->allocationBlock 的值是否小于 0。yaffs_Device 结构内的 allocationBlock 字段用于 记录当前从中分配 chunk（page）的那个 block 的序号。当一个 block 内的所有 page 全部分配完毕时，就将这个字段置为-1，下次进入该函 数时就会重新挑选空闲的 block。这里我们假定需要重新挑选空闲 block，因此进入 yaffs_FindBlockForAllocation 函数：

```
[yaffs_AllocateChunk() => yaffs_FindBlockForAllocation()]
static int yaffs_FindBlockForAllocation(yaffs_Device * dev)
{
int i;
yaffs_BlockInfo *bi;
if (dev->nErasedBlocks < 1) {
/* Hoosterman we've got a problem.
* Can't get space to gc
*/
T(YAFFS_TRACE_ERROR,
(TSTR("yaffs tragedy: no more eraased blocks" TENDSTR)));
return -1;
}
```

dev->nErasedBlocks 记录着器件内所有可供分配的 block 的数量。如果该值小于 1，那显然是有问题了。不但正常的分配请求无法完成，就连垃圾收集都办不到了。

```
for (i = dev->internalStartBlock; i <= dev->internalEndBlock; i++) {
dev->allocationBlockFinder++;
if (dev->allocationBlockFinder < dev->internalStartBlock
|| dev->allocationBlockFinder > dev->internalEndBlock) {
dev->allocationBlockFinder = dev->internalStartBlock;
```

internalStartBlock 和 internalEndBlock 分别是 yaffs2 使用的 block 的起始序号和结束序号。也就是说 yaffs2 文件系统不一定要占据整个 Flash，可以只占用其中的一部分。

dev->allocationBlockFinder 记录着上次分配的块的序号。如果已经分配到系统尾部，就从头重新开始搜索可用块。

```
bi = yaffs_GetBlockInfo(dev, dev->allocationBlockFinder);
if (bi->blockState == YAFFS_BLOCK_STATE_EMPTY) {
bi->blockState = YAFFS_BLOCK_STATE_ALLOCATING;
dev->sequenceNumber++;
```

```
bi->sequenceNumber = dev->sequenceNumber;
dev->nErasedBlocks--;
T(YAFFS_TRACE_ALLOCATE,
(TSTR("Allocated block %d, seq %d, %d left" TENDSTR),
dev->allocationBlockFinder, dev->sequenceNumber,
dev->nErasedBlocks));
return dev->allocationBlockFinder;
}
```

yaffs_GetBlockInfo 函数获取指向 block 信息结构的指针，该函数比较简单，就不详细介绍了。yaffs_BlockInfo 结构中的 blockState 成员描述该 block 的状态，比如空，满，已损坏，当前分配中，等等。因为是要分配空闲块，所以块状态必须是 YAFFS_BLOCK_STATE_EMPTY，如果不是，就继续测试下一个 block。找到以后将 block 状态修改为 YAFFS_BLOCK_STATE_ALLOCATING，表示当前正从该 block 中分配存储空间。正常情况下，系统中只会有一个 block 处于该状 态。另外还要更新统计量 ErasedBlocks 和 sequenceNumber。这个 sequenceNumber 记录着各 block 被分配出去的先后 顺序，以后在垃圾收集的时候会以此作为判断该 block 是否适合回收的依据。

现在让我们返回到函数 yaffs_AllocateChunk 中。yaffs_CheckSpaceForAllocation()函数检查 Flash 上是否有足够的可用空间，通过检查后，就从当前供分配的 block 上切下一个 chunk：

```
if (dev->allocationBlock >= 0) {
bi = yaffs_GetBlockInfo(dev, dev->allocationBlock);
retVal = (dev->allocationBlock * dev->nChunksPerBlock) +
dev->allocationPage;
bi->pagesInUse++;
yaffs_SetChunkBit(dev, dev->allocationBlock,
dev->allocationPage);
dev->allocationPage++;
dev->nFreeChunks--;
/* If the block is full set the state to full */
if (dev->allocationPage >= dev->nChunksPerBlock) {
bi->blockState = YAFFS_BLOCK_STATE_FULL;
dev->allocationBlock = -1;
}
if(blockUsedPtr)
*blockUsedPtr = bi;
return retVal;
}
```

dev->allocationPage 记录着上次分配的 chunk 在 block 中的序号，每分配一次加 1。从这里我们可以看出，系统在分配 chunk 的时候是从 block 的开头到结尾按序分配的，直到一个 block 内的所有 chunk 全部分配完毕为止。retVal 是该 chunk 在整个 device 内的总序号。PagesInUse 记录着该 block 中已分配使用的 page 的数量。

系统在设备描述结构 yaffs_Device 中维护着一张位图，该位图的每一位都代表着 Flash 上的一个 chunk 的状态。yaffs_SetChunkBit()将刚分配得到的 chunk 在位图中的对应位置 1，表明该块已被使用。更新一些统计量后，就可以返回了。

看过 chunk 分配以后，我们再来 chunk 的释放。和 chunk 分配不同的是，chunk 的释放在大多数情况下并不释放对应的物理介质，这是因为 NAND 虽然可以按 page 写，但只能按 block 擦除，所以物理介质的释放要留到垃圾收集或一个 block 上的所有 page 全部变成空闲的时候才进行。 根据应用场合的不同，chunk 的释放方式并不唯一，分别由 yaffs_DeleteChunk 函数和 yaffs_SoftDeleteChunk 函数完 成。我们先看 yaffs_DeleteChunk：

```
void yaffs_DeleteChunk(yaffs_Device * dev, int chunkId, int markNAND, int lyn)
```

chunkId 就是要删除的 chunk 的序号，markNand 参数用于 yaffs 一代的代码中，yaffs2 不使用该参数。
参数 lyn 在调用该函数时置成当前行号(__LINE__)，用于调试。
首先通过 yaffs_GetBlockInfo 获得 chunk 所在 block 的信息描述结构指针，然后就跑到 else 里面去了。if 语句的判断条件中有一 条!dev->isYaffs2，所以对于 yaffs2 而言是不会执行 if 分支的。在 else 分支里面只是递增一下统计计数就出来了，我们接着往下 看。

```
if (bi->blockState == YAFFS_BLOCK_STATE_ALLOCATING ‖
bi->blockState == YAFFS_BLOCK_STATE_FULL ‖
bi->blockState == YAFFS_BLOCK_STATE_NEEDS_SCANNING ‖
bi->blockState == YAFFS_BLOCK_STATE_COLLECTING) {
dev->nFreeChunks++;
yaffs_ClearChunkBit(dev, block, page);
bi->pagesInUse--;
if (bi->pagesInUse == 0 &&
!bi->hasShrinkHeader &&
bi->blockState != YAFFS_BLOCK_STATE_ALLOCATING &&
bi->blockState != YAFFS_BLOCK_STATE_NEEDS_SCANNING) {
yaffs_BlockBecameDirty(dev, block);
}
} else {
/* T(("Bad news deleting chunk %d\n",chunkId)); */
}
```

首先要判断一下该 block 上是否确实存在着可释放的 chunk。block 不能为空，不能是坏块。 YAFFS_BLOCK_STATE_NEEDS_SCANNING 表明正对该块进行垃圾回收，我们后面会分析；YAFFS_BLOCK_STATE_NEEDS_SCANNING 在我手上的源代码中似乎没有用到。
通过判断以后，所做的工作和 chunk 分配函数类似，只是一个递增统计值，一个递减。递减统计值以后还要判断该 block 上的 page 是否已全部释放，如 果已全部释放，并且不是

当前分配块，就通过 yaffs_BlockBecameDirty 函数删除该 block，只要能通过删除操作（不是坏块），该 block 就又可以用于分配了。
相比较来说，yaffs_SoftDeleteChunk 所做的工作就简单多了。关键的代码只有两行：

```
static void yaffs_SoftDeleteChunk(yaffs_Device * dev, int chunk)
{
……
theBlock->softDeletions++;
dev->nFreeChunks++;
……
}
```

这里递增的是 yaffs_blockInfo 结构中的另一个统计量 softDeletions，而没有修改 pagesInUse 成员，也没有修改 chunk 状态位图。那么，这两个函数的应用场合有什么区别呢？
一般来说，yaffs_DeleteChunk 用于文件内容的更新。比如我们要修改文件中的部分内容，这时候 yaffs2 会分配新的 chunk，将更改后的内容写入新 chunk 中，原 chunk 的内容自然就没有用了，所以要将 pageInUse 减 1，并修改位图；
yaffs_SoftDeleteChunk 用于文件的删除。yaffs2 在删除文件的时候只是删除该文件在内存中的一些描述结构，而被删除的文件所占用 的 chunk 不会立即释放，也就是不会删除文件内容，在后续的文件系统操作中一般也不会把这些 chunk 分配出去，直到系统进行垃圾收集的时候才有选择地 释放这些 chunk。熟悉 DOS 的朋友可能还记得，DOS 在删除的文件的时候也不会立即删除文件内容，只是将文件名的第一个字符修改为 0xA5，事后还可以恢复文件内容。yaffs2 在这点上是类似的。

1.文件地址映射
上面说到，yaffs 文件系统在更新文件数据的时候，会分配一块新的 chunk，也就是说，同样的文件偏移地址，在该地址上的数据更新前和更新后，其对应 的 flash 上的存储地址是不一样的。那么，如何根据文件内偏移地址确定 flash 存储地址呢？最容易想到的办法，就是在内存中维护一张映射表。由于 flash 基本存储单位是 chunk，因此，只要将以 chunk 描述的文件偏移量作为表索引，将 flash chunk 序号作为表内容，就可以解决该问题了。但是这个方法有几个问题，首先就是在做 seek 操作的时候，要从表项 0 开始按序搜索，对于大文件会消耗很 多时间；其次是在建立映射表的时候，无法预计文件大小的变化，于是就可能在后来的操作中频繁释放分配内存以改变表长，造成内存碎片。yaffs 的解决方法 是将这张大的映射表拆分成若干个等长的小表，并将这些小表组织成树的结构，方便管理。我们先看小表的定义：

```
union yaffs_Tnode_union {
union yaffs_Tnode_union *internal[YAFFS_NTNODES_INTERNAL];
}
```

YAFFS_NTNODES_INTERNAL 定义为(YAFFS_NTNODES_LEVEL0 / 2)，而 YAFFS_NTNODES_LEVEL0 定义为 16，所以这实际上是一个长度为 8 的指针数组。不管是叶子节点还是非叶节点，都是这个结构。当节点为非叶 节点时，数组中的每个元素都指向下一层子节点；当节点为叶子节点时，该数组拆分为 16 个 16 位长的短整数(也有例外，后

面会说到)，该短整数就是文件内容 在 flash 上的存储位置（即 chunk 序号）。至于如何通过文件内偏移找到对应的 flash 存储位置，源代码所附文档（Development/yaffs/Documentation/yaffs-notes2.html）已经有说明，俺就不在此处饶舌了。下面看具体 函数。

为了行文方便，后文中将 yaffs_Tnode 这个指针数组称为“一组”Tnode，而将数组中的每个元素称为“一个”Tnode。树中的每个节点，都是“一组”Tnode。

先看映射树的节点的分配。

```
static yaffs_Tnode *yaffs_GetTnode(yaffs_Device * dev)
{
yaffs_Tnode *tn = yaffs_GetTnodeRaw(dev);

if(tn)
memset(tn, 0, (dev->tnodeWidth * YAFFS_NTNODES_LEVEL0)/8);

return tn;
}
```

调用 yaffs_GetTnodeRaw 分配节点，然后将得到的节点初始化为零。

```
static yaffs_Tnode *yaffs_GetTnodeRaw(yaffs_Device * dev)
{
yaffs_Tnode *tn = NULL;

/* If there are none left make more */
if (!dev->freeTnodes) {
yaffs_CreateTnodes(dev, YAFFS_ALLOCATION_NTNODES);
}
```

当前所有空闲节点组成一个链表，dev->freeTnodes 是这个链表的表头。我们假定已经没有空闲节点可用，需通过 yaffs_CreateTnodes 创建一批新的节点。

```
static int yaffs_CreateTnodes(yaffs_Device * dev, int nTnodes)
{
……
tnodeSize = (dev->tnodeWidth * YAFFS_NTNODES_LEVEL0)/8;
newTnodes = YMALLOC(nTnodes * tnodeSize);
mem = (__u8 *)newTnodes;
```

上面说过，叶节点中一个 Tnode 的位宽默认为 16 位，也就是可以表示 65536 个 chunk。对于时下的大容量 flash，chunk 的大小为 2K，因 此在默认情况下 yaffs2 所能寻址的最大 flash 空间就是 128M。为了能将 yaffs2 用于大容量 flash 上，代码作者试图通过两种手段解决这个 问题。第一种手段就是这里的 dev->tnodeWidth，通过增加单个 Tnode 的位宽，就可以增加其所能表示的最大 chunk Id；另一种手段是我们后面将看到的 chunk group，通过将若干个 chunk 合成一组用同一个 id 来表示，也可以增加系统所能寻址的 chunk 范围。

俺为了简单，分析的时候不考虑这两种情况，因 此tnodeWidth取默认值16，也不考虑将多个chunk合成一组的情况，只在遇到跟这两种情况有关的代码时作简单说明。

在32位的系统中，指针的宽度为32位，而chunk id的宽度为16位，因此相同大小的Tnode组，可以用来表示N个非叶Tnode（作为指针使用），也可以用来表示N * 2个叶子Tnode（作为chunk id使用）。代码中分别用YAFFS_NTNODES_INTERNAL和YAFFS_NTNODES_LEVEL0来表示。前者取值为8，后者取值为16。从这里我们也可以看出若将yaffs2用于64位系统需要作哪些修改。 针对上一段叙述的问题，俺以为在内存不紧张的情况下，不如将叶节点Tnode和非叶节点Tnode都设为一个指针的长度。

分配得到所需的内存后，就将这些空闲空间组成Tnode链表：

```
for(i = 0; i < nTnodes -1; i++) {
curr = (yaffs_Tnode *) &mem[i * tnodeSize];
next = (yaffs_Tnode *) &mem[(i+1) * tnodeSize];
curr->internal[0] = next;
}
```

每组Tnode的第一个元素作为指针指向下一组Tnode。完成链表构造后，还要递增统计量，并将新得到的Tnodes挂入一个全局管理链表yaffs_TnodeList：

```
dev->nFreeTnodes += nTnodes;
dev->nTnodesCreated += nTnodes;
tnl = YMALLOC(sizeof(yaffs_TnodeList));
if (!tnl) {
T(YAFFS_TRACE_ERROR,
(TSTR
("yaffs: Could not add tnodes to management list" TENDSTR)));
} else {
tnl->tnodes = newTnodes;
tnl->next = dev->allocatedTnodeList;
dev->allocatedTnodeList = tnl;
}
```

回到yaffs_GetTnodeRaw，创建了若干组新的Tnode以后，从中切下所需的Tnode，并修改空闲链表表头指针：

```
if (dev->freeTnodes) {
tn = dev->freeTnodes;
dev->freeTnodes = dev->freeTnodes->internal[0];
dev->nFreeTnodes--;
}
```

至此，分配工作就完成了。相比较来说，释放Tnodes的工作就简单多了，简单的链表和

统计值操作：

```
static void yaffs_FreeTnode(yaffs_Device * dev, yaffs_Tnode * tn)
{
if (tn) {
tn->internal[0] = dev->freeTnodes;
dev->freeTnodes = tn;
dev->nFreeTnodes++;
}
}
```

看过 Tnode 的分配和释放，我们再来看看这些 Tnode 是如何使用的。在后文中，我们把以 chunk 为单位的文件内偏移称作逻辑 chunk id，文件内容在 flash 上的实际存储位置称作物理 chunk id。先看一个比较简单的函数。

```
void yaffs_PutLevel0Tnode(yaffs_Device *dev, yaffs_Tnode *tn, unsigned pos,
unsigned val)
```

这个函数将某个 Tnode 设置为指定的值。tn 是指向一组 Tnode 的指针；pos 是所要设置的那个 Tnode 在该组 Tnode 中的索引；val 就是所要设置的值，也就是物理 chunk id。函数名中的 Level0 指映射树的叶节点。函数开头几行如下：

```
pos &= YAFFS_TNODES_LEVEL0_MASK;
val >>= dev->chunkGroupBits;
bitInMap = pos * dev->tnodeWidth;
wordInMap = bitInMap /32;
bitInWord = bitInMap & (32 -1);
mask = dev->tnodeMask << bitInWord;
```

上面说过，一组 Tnode 中的 8 个指针在叶节点这一层转换成 16 个 16 位宽的 chunk Id，因此需要 4 位二进制码对其进行索引，这就是 YAFFS_TNODES_LEVEL0_MASK 的值。我们还说过这个 16 位值就是 chunk 在 flash 上的序号，当 flash 容量比较大， chunk 数量多时，16 位可能无法给 flash 上的所有 chunk 编号，这种情况下可以增加 chunk id 的位宽，具体位宽的值记录在 dev->tnodeWidth 中。yaffs2 允许使用非字节对齐的 tnodeWidth，因此可能出现某个 chunk id 跨 32 位边界存储的情况。所以在下面的代码中，需要分边界前和边界后两部分处理：

```
map[wordInMap] &= ~mask;
map[wordInMap] |= (mask & (val << bitInWord));

if(dev->tnodeWidth > (32-bitInWord)) {
bitInWord = (32 - bitInWord);
wordInMap++;;
mask = dev->tnodeMask >> (/*dev->tnodeWidth -*/ bitInWord);
```

```
map[wordInMap] &= ~mask;
map[wordInMap] |= (mask & (val >> bitInWord));
}
```

if 语句判断当前 chunk 序号是否跨越当前 32 位边界。整个代码初看起来比较难理解，其实只要将 dev->tnodeWidth 以 16 或 32 代入， 就很好懂了。还有一个类似的函数 yaffs_GetChunkGroupBase，返回由 tn 和 pos 确定的一组 chunk 的起始序号，就不详细分析了。

现在我们假设有这样一个情景：已知文件偏移地址，要找到 flash 上对应的存储地址，该怎么做呢？这项工作的主体是由函数 yaffs_FindLeve10Tnode 完成的。

```
static yaffs_Tnode *yaffs_FindLeve10Tnode(yaffs_Device * dev,
yaffs_FileStructure * fStruct,
__u32 chunkId)
{

yaffs_Tnode *tn = fStruct->top;
__u32 i;
int requiredTallness;
int level = fStruct->topLevel;
```

函数参数中，fStruct 是指向文件描述结构的指针，该结构保存着文件大小、映射树层高、映射树顶层节点指针等信息。chunkId 是逻辑 chunk id。
fStruct->top 是映射树顶层节点指针，fStruct->topLevel 是映射树层高。注意：当只有一层时，层高为 0。

```
/* First check we're tall enough (ie enough topLevel) */
i = chunkId >> YAFFS_TNODES_LEVEL0_BITS;
requiredTallness = 0;
while (i) {
i >>= YAFFS_TNODES_INTERNAL_BITS;
requiredTallness++;
}
if (requiredTallness > fStruct->topLevel) {
/* Not tall enough, so we can't find it, return NULL. */
return NULL;
}
```

在 看这段代码之前，我们先用一个例子来回顾一下映射树的组成。假定我们有一个大小为 128K 的文件，flash 的 page 大小为 2K，那么我们就需要 64 个 page（或者说 chunk）来存储该文件。一组 Tnode 的 size 是 8 个指针，或者 16 个 16 位整数，所以我们需要 64 / 16 = 4 组 Tnode 来存储物理 chunk 序号。这 4 组 Tnode 就是映射树的叶节点，也就是 Leve10 节点。由于这 4 组 Tnode 在内存中不一定连续，所以 我们需要另外一组 Tnode，将其作为指针数组使用，这个指针数组的前 4 个元素分别指向 4 组 Leve10 节点，而 fStruct->top

就指向这 组作为指针数组使用的Tnode。随着文件长度的增大，所需的叶节点越多，非叶节点也越多，树也就越长越高。
回过头来看代码，首先是检查函数参数chunkId是否超过文件长度。作为非叶节点使用的Tnode每组有8个指针，需要3位二进制码对其进行索引，因此 树每长高一层，逻辑chunkId就多出3位。相反，每3位非零chunkId就代表一层非叶节点。while循环根据这个原则计算参数chunkId所 对应的树高。如果树高超过了文件结构中保存的树高，那就说明该逻辑chunkId已经超出文件长度了。通过文件长度检查之后，同样根据上面的原则，就可以 找到逻辑chunkId对应的物理chunkId了。具体的操作通过一个while循环完成：

```
/* Traverse down to level 0 */
while (level > 0 && tn) {
tn = tn->
internal[(chunkId >>
( YAFFS_TNODES_LEVEL0_BITS +
(level - 1) *
YAFFS_TNODES_INTERNAL_BITS)
) &
YAFFS_TNODES_INTERNAL_MASK];
level--;

}

return tn;
```

将返回值和逻辑chunk id作为参数调用yaffs_GetChunkGroupBase，就可以得到物理chunk id了。

下面我们看另一个情景，看看当文件长度增加的时候，映射树是如何扩展的。主要函数为

```
static yaffs_Tnode *yaffs_AddOrFindLevel0Tnode(yaffs_Device * dev,
yaffs_FileStructure * fStruct,
__u32 chunkId,
yaffs_Tnode *passedTn)
```

函数的前几行和yaffs_FindLevel0Tnode一样，对函数参数作一些检查。通过检查之后，首先看原映射树是否有足够的高度，如果高度不够，就先将其“拔高”：

```
if (requiredTallness > fStruct->topLevel) {
/* Not tall enough,gotta make the tree taller */
for (i = fStruct->topLevel; i < requiredTallness; i++) {
tn = yaffs_GetTnode(dev);
if (tn) {
tn->internal[0] = fStruct->top;
```

```
fStruct->top = tn;
} else {
T(YAFFS_TRACE_ERROR,
(TSTR("yaffs: no more tnodes" TENDSTR)));
}
}
fStruct->topLevel = requiredTallness;
}
```

for 循环完成增加新层的功能。新增的每一层都只有一个节点（即一组 Tnode），fStruct->top 始终指向最新分配的节点。将映射树扩展到所需的高度之后，再根据需要将其“增肥”,扩展其“宽度”：

```
l = fStruct->topLevel;
tn = fStruct->top;
if(l > 0) {
while (l > 0 && tn) {
x = (chunkId >>
( YAFFS_TNODES_LEVEL0_BITS +
(l - 1) * YAFFS_TNODES_INTERNAL_BITS)) &
YAFFS_TNODES_INTERNAL_MASK;
if((l>1) && !tn->internal[x]){
/* Add missing non-level-zero tnode */
tn->internal[x] = yaffs_GetTnode(dev);
} else if(l == 1) {
/* Looking from level 1 at level 0 */
if (passedTn) {
/* If we already have one, then release it.*/
if(tn->internal[x])
yaffs_FreeTnode(dev,tn->internal[x]);
tn->internal[x] = passedTn;
} else if(!tn->internal[x]) {
/* Don't have one, none passed in */
tn->internal[x] = yaffs_GetTnode(dev);
}
}
tn = tn->internal[x];
l--;
}
}
```

上面“拔高”的时候是从下往上“盖楼”，这里“增肥”的时候是从上往下“扩展”。tn->internal[x]为空表示下层节点尚未创建，需要通过 yaffs_GetTnode 分配之，就是“增肥”了。如果函数参数 passedTn 有效，就用该组 Tnode 代替 level0 上原先的那组

Tnode；否则按需分配新的 Tnode 组。所以这里的函数名似乎应该取作 yaffs_AddOrFindOrReplaceLevel0Tnode 更加恰当。不过这个新名字也太长了些……

树的创建、搜索和扩展说完了，下面该说什么？……对了，收缩和删除。不过看过创建搜索扩展之后，收缩和删除已经没什么味道了。主要函数有：

yaffs_DeleteWorker()

yaffs_SoftDeleteWorker()

yaffs_PruneWorker()

前两者用于删除，第三个用于收缩。都是从 level0 开始，以递归的方式从叶节点向上删，并释放被删除 Tnode 所对应的物理 chunk。递归，伟大的递 归啊……俺不想把这篇文章做成递归算法教程，除了递归这三个函数也就不剩啥了，所以一概从略。唯一要说的就是 yaffs_DeleteWorker 和 yaffs_SoftDeleteWorker 的区别，这两个函数非常类似，只是在释放物理 chunk 的时候分别调用 yaffs_DeleteChunk 和 yaffs_SoftDeleteChunk。其中函数 yaffs_DeleteWorker 在 yaffs2 中似乎是不用的，而 yaffs_SoftDeleteWorker 主要用于在删除文件时资源的释放。

7.文件系统对象

在 yaffs2 中，不管是文件还是目录或者是链接，在内存都用一个结构体 yaffs_ObjectStruct 来描述。我们先简要介绍一下这个结构体中的 几个关键字段，然后再来看代码。在后文中提到“文件”或“文件对象”，若不加特别说明，都指广义的“文件”，既可以是文件，也可以是目录。

```
__u8 deleted:1; /* This should only apply to unlinked files. */
__u8 softDeleted:1; /* it has also been soft deleted */
__u8 unlinked:1; /* An unlinked file. The file should be in the unlinked directory.*/
```

这三个字段用于描述该文件对象在删除过程中所处的阶段。在删除文件时，首先要将文件从原目录移至一个特殊的系统目录/unlinked，以此拒绝应用程序 对该文件的访问，此时将 unlinked 置 1；然后判断该文件长度是否为 0，如果为 0，该文件就可以直接删除，此时将 deleted 置 1；如果不为 0，就 将 deleted 和 softDelted 都置 1，表明该文件数据所占据的 chunk 还没有释放，要留待后继处理。

```
struct yaffs_ObjectStruct *parent;
```

看名字就知道，该指针指向上层目录。

```
int chunkId;
```

每个文件在 flash 上都有一个文件头，存储着该文件的大小、所有者、创建修改时间等信息。chunkId 就是该文件头在 flash 上的 chunk 序号。

```
__u32 objectId; /* the object id value */
```

每一个文件系统对象都被赋予一个唯一的编号，作为对象标识，也用于将该对象挂入一个散列表，加快对象的搜索速度。

```
yaffs_ObjectType variantType;
yaffs_ObjectVariant variant;
```

前者表示该对象的类型，是目录、普通文件还是链接文件。后者是一个联合体，根据对象类型的不同有不同的解释。

其余的成员变量，我们在后面结合函数一起分析。

下面我们来看相关的函数。先看一个简单的：

```
static yaffs_Object *yaffs_CreateFakeDirectory(yaffs_Device * dev, int number,
__u32 mode)
```

所谓 Fake Directory，就是仅存在于内存中，用于管理目的的目录对象，比如我们上面提到的 unlinked 目录。这种类型的目录有一些特别的地方，如禁止改 名、禁止删除等。由于对象仅存在于内存中，因此不涉及对硬件的操作，所以函数体很简单。首先通过 yaffs_CreateNewObject 获得一个新对 象，然后对其中的一些字段初始化。先把字段初始化看一下，顺便再介绍一些字段：

renameAllowed 表示是否允许改名，对于 fake 对象为 0；

unlinkAllowed 表示是否允许删除，对于 fake 对象同样为 0；

yst_mode 就是 linux 中的访问权限位；

chunkId 是对象头所在 chunk，由于 fake 对象不占 flash 存储空间，所以置 0。

回过头来看 yaffs_CreateNewObject：

[yaffs_CreateFakeDirectory --> yaffs_CreateNewObject]

```
yaffs_Object *yaffs_CreateNewObject(yaffs_Device * dev, int number,
yaffs_ObjectType type)
{

yaffs_Object *theObject;

if (number < 0) {
number = yaffs_CreateNewObjectNumber(dev);
}

theObject = yaffs_AllocateEmptyObject(dev);
```

前面说过，每个 yaffs_Object 都有一个唯一的序列号，这个序号既可以在创建对象的时候由上层函数指定，也可以由系统分配。如果 number < 0，那就表示由系统分配。序列号分配函数是 yaffs_CreateNewObjectNumber。我们就不深入到这个函数内部了，只说明一下该函数做了些什么：

系统为了方便根据对象 id 找到对象本身，将每个对象都通过指针 hashLink 挂入了一个散列表，散列函数是 number % 256，所以这个散列表有 256 个表项。

yaffs_CreateNewObjectNumber 函数每次搜索 10 个表项，从中选取挂接链表长度最短的那一项，再根据表索引试图计算出一个和该索引上挂接对象的 id 号不重复的 id。

分配到了 id 号和空闲对象后，再根据对象类型的不同作不同的处理。我们主要关心两种情况，就是对象类型分别为文件和目录的时候：

```
case YAFFS_OBJECT_TYPE_FILE:
theObject->variant.fileVariant.fileSize = 0;
theObject->variant.fileVariant.scannedFileSize = 0;
```

```
theObject->variant.fileVariant.shrinkSize = 0xFFFFFFFF; /* max __u32 */
theObject->variant.fileVariant.topLevel = 0;
theObject->variant.fileVariant.top = yaffs_GetTnode(dev);
break;
case YAFFS_OBJECT_TYPE_DIRECTORY:
INIT_LIST_HEAD(&theObject->variant.directoryVariant.children);
break;
```

fileSize 很好理解；topLevel 就是映射树层高，新建的文件层高为 0。还要预先分配一组 Tnode 供该对象使用。 scannedFileSize 和 shrinkSize 用于 yaffs2 初始化时的 flash 扫描阶段，这里先跳过。如果该对象是目录，那么所做的工作只 是初始化子对象（就是该目录下的文件或子目录）双向链表指针，前后指针都指向链表头自身。

看过 Fake 对象创建，我们再看看普通对象的创建。按对象类型的不同，有四个函数分别用于创建普通文件、目录、设备文件、符号链接和硬链接，它们分别是：

```
yaffs_MknodFile;
yaffs_MknodDirectory;
yaffs_MknodSpecial;
yaffs_MknodSymLink;
yaffs_Link
```

这四个函数最终都调用 yaffs_MknodObject 来完成创建对象的工作，只是调用参数不一样。

```
static yaffs_Object *yaffs_MknodObject(yaffs_ObjectType type,
yaffs_Object * parent,
const YCHAR * name,
__u32 mode,
__u32 uid,
__u32 gid,
yaffs_Object * equivalentObject,
const YCHAR * aliasString, __u32 rdev)
```

函数参数中，前面几个都很好理解，分别是对象类型，上级目录对象，文件名，访问权限，文件所属 user id 和 group id； equivalentObject 是创建硬链接时的原始文件对象；aliasString 是 symLink 名称；rdev 是设备文件的设备号。

函数首先检查在父目录中是否已存在同名文件，然后同样调用 yaffs_CreateNewObject 创建新对象。参数-1 表示由系统自行选择对象 id。

```
if (in) {
in->chunkId = -1;
in->valid = 1;
in->variantType = type;
in->yst_mode = mode;
in->yst_atime = in->yst_mtime = in->yst_ctime = Y_CURRENT_TIME;
in->yst_rdev = rdev;
```

```
in->yst_uid = uid;
in->yst_gid = gid;
in->nDataChunks = 0;
yaffs_SetObjectName(in, name);
in->dirty = 1;
yaffs_AddObjectToDirectory(parent, in);
in->myDev = parent->myDev;
```

这里列出的代码省略了和 wince 相关的条件编译部分。chunkId 是对象头所在 chunk，现在还没有将对象写入 flash，所以置为-1；该新对象 暂时还没有数据，所以 nDataChunks 是 0。in->dirty = 1 表示该新对象信息还没有写入 flash。然后通过 yaffs_AddObjectToDirectory 将新对象挂入父对象的子对象链表。接下来根据对 象类型作不同处理：

```
switch (type) {
case YAFFS_OBJECT_TYPE_SYMLINK:
in->variant.symLinkVariant.alias =
yaffs_CloneString(aliasString);
break;
case YAFFS_OBJECT_TYPE_HARDLINK:
in->variant.hardLinkVariant.equivalentObject =
equivalentObject;
in->variant.hardLinkVariant.equivalentObjectId =
equivalentObject->objectId;
list_add(&in->hardLinks, &equivalentObject->hardLinks);
break;
case YAFFS_OBJECT_TYPE_FILE:
case YAFFS_OBJECT_TYPE_DIRECTORY:
case YAFFS_OBJECT_TYPE_SPECIAL:
case YAFFS_OBJECT_TYPE_UNKNOWN:
/* do nothing */
break;
}
```

对于最常用的文件对象和目录对象不做任何处理；如果是 hardlink，就将新对象挂入原对象的 hardLinks 链表。从这里我们可以看出，yaffs2 在内存中是以链表的形式处理 hardlink 的。在将 hardlink 存储到 flash 上的时 候，则是通过 objectId 将两者关联起来。Hardlink 本身占用一个 chunk 存储对象头。

最后，通过 yaffs_UpdateObjectHeader 将新对象头写入 flash。