DOI:10.16184/j.cnki.comprg.2011.15.002

Linux 运行前探秘之三
——内核解压缩（二）

徐 炜

摘 要：分析了 Linux 内核解压过程和算法，并对 HSF 解码表创建过程的关键源代码作了详细解析。

关键词：Huffman 编码；HSF 解码表

1 概述

前一篇讲到 HSF 编码（一种 Huffman 编码实现）的解码表是多级表，并且不同的编码因其码长不同，级数也不同。标准的 Huffman 编码是通过二叉树构造并且解码的，每访问一个节点解出一位码长，完全解码必须遍历完编码所在叶节点的深度(即码长 n)，效率较低。HSF 解码表通过构造索引码长 L 的多级表，使得一次可以解码 L 位，并根据解码的中间结果得到需要续读的编码位数，处理次数减少为 (n+L-1) /L 次。另外，单级表除了浪费内存空间外，也无法实现对变长前缀码的解码。所以，多级表是 HSF 解码的最佳选择。

2 HSF 解码表

HSF 解码表是由 huft 结构数组构成，解码时以编码中的当前 l 位为索引找到数组中的 huft 结构来对 l 位进行解码的。解完一个编码的最后一位时到达的 huft 结构就相当于叶节点，其中包含了编码的值的信息，而其他中间表项的作用是指示下一级表的起始地址和索引位长 (即接下来读入并解码的二进制位数)。上篇展示了解码表的层级关系，有点类似页目录表、页中间目录表和页表间的关系。这里补充说明一下解码表之间的另一层关系，就是线性链表关系，这是为了在解码完毕后释放各解码表空间时能通过链指针找到各表地址。为了保存链表指针，每次创建解码表时，都会多分配一项，比如计算得出要分配 z 项，就分配 (z+1) 个 huft 结构的内存。假设分配时返回的地址 q，那么表的起始地址是 (huft *) q+1，而 *q 则用作线性表的 link 项，各解码表通过 q->v.t 指针以生成次序链接在一起。

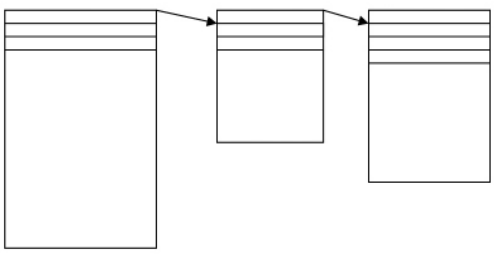

图 1 huft 表的线性链接

3 代码分析

3.1 构造 Huffman 编码表

```
STATIC int huft_build(b, n, s, d, e, t, m)
unsigned *b;      /* 待编码二进制值的编码长度数组指针(编码长度 <= BMAX) */
unsigned n;       /* 编码数 ( <= N_MAX) */
unsigned s;       /* 简单值(无附加位)编码数 (0..s-1) */
const ush *d;     /* 非简单值编码的基本值数组指针 */
const ush *e;     /* 非简单值编码的附加位长数组指针 */
struct huft **t;  /* 通过 *t 返回起始解码表(第 0 级表)指针 */
int *m;           /* *m 返回解码时访问 0 级表的索引位数 */
/* 本函数根据参数给出编码长度的数组和编码数，创建多级解码表，*t 指向 0 级表。
成功返回 0；给定编码集不完整返回 1 ；某位长无法容纳指定的编码数时返回 2，；
没有足够的内存建表返回 3。如果传入的位长都是 0，则不创建解码表，*t=null。 */
{
  unsigned a;   /* 循环中保存当前编码长度 k 的剩余编码数 */
  unsigned c[BMAX+1];     /* c[BMAX+1]:保存不同位长编码计数的数组 */
  unsigned f;       /* i repeats in table every f entries */
  int g;            /* 最大编码长度 */
  int h;            /* 表级别，创建的第一个表级别是 0 */
  register unsigned i;
/* i 是计数器，也是当前的 HSF 编码(反序二进制编码) */
  register unsigned j;        /* counter */
  register int k;             /* 当前编码 i 的位数(即码长) */
  int l;            /* 各级子表的最大解码长度(即索引位数) */
  register unsigned *p;     /* 指向 c[], b[], or v[]的临时指针 */
  register struct huft *q;      /* 指向当前创建的子表 */
  struct huft r;      /* 用于给解码表项赋值的 huft 结构变量 */
  struct huft *u[BMAX];
/* 保存当前编码解码时需访问的各级子表指针 */
  unsigned v[N_MAX];
/*v[N_MAX]:以位长顺序排列的各 HSF 编码的值 */
  register int w;
/* w=l*h 表示访问 h 级别子表前已解码的位长(h>=0)*/
  unsigned x [BMAX+1];          /* bit offsets, then code
```

```
stack */
  unsigned *xp;                 /* 指向 x[]数组的指针 */
  int y;              /* 计算要添加的 dummy codes 数量 */
  unsigned z;         /* 当前表的项数 */
DEBG("huft1 ");
//以下计算各种位长度的编码数,保存在 c[]数组,首先把数组清
//0
  memzero(c, sizeof(c));
  p = b;  i = n; //参数 b 指向编码长度数组,n 是编码数
  do {
    Tracecv(*p, (stderr, (n-i >= ' ' && n-i <= '~' ? "%c %d\n"
: "0x%x %d\n"),
            n-i, *p));
    c[*p]++;            /* 增加相应的编码长度 *p 的计数 */
    p++;                      /* p++指向下一个长度 */
  } while (--i);              /* 循环直至遍历完 n 个编码长度 */
  if (c[0] == n)                /* 所有编码的长度都是 0 的情况 */
  {
    *t = (struct huft *)NULL;  /* 此时无需创建解码表,因此 *t
返回 NULL*/
    *m = 0;                 /* 0 级解码表的索引位数 *m=0*/
    return 0;               /* 返回 0 表示成功 */
  }
DEBG("huft2 ");
/* 下面找出最小和最大的编码长度, 并保证 l=*m 不超出它们
的范围 */
  l = *m;           //l = *m
  for (j = 1; j <= BMAX; j++)  //计算最小编码长度
    if (c[j])
      break;
  k = j;                        /* k=j=最小编码长度 */
  if ((unsigned)l < j)          //如果 l<最小长度 j,令 l=j;
    l = j;
  for (i = BMAX; i; i--)        //计算最大编码长度
    if (c[i])
      break;
  g = i;                        /* g=i=最大编码长度 */
  if ((unsigned)l > i)          //如果 l>最大长度 i,令 l=i;
    l = i;
  *m = l;
//*m = l。最后经过调整的 l 将作为第一级解码表的索引码长
DEBG("huft3 ");
/* 调整最大长度的编码数到最大值 */
  for (y = 1 << j; j < i; j++, y <<= 1)
    if ((y -= c[j]) < 0)        //当 y-=c[j]<0 时,表示预置的码长为
//j 的编码数大于该码长实际能表示的最大编码数,此时返回 2
      return 2;       /* bad input: more codes than bits */
  if ((y -= c[i]) < 0)
    return 2;
  c[i] += y;     //y 表示最大码长中剩余未用的编码数量,增加到
//最大码长的编码数量中
DEBG("huft4 ");
/* 按照编码本身的二进制数值的顺序(码长由小到大),得出各码
长编码在序列中的偏移位置,放在 x[]数组中 */
  x[1] = j = 0;
  p = c + 1;  xp = x + 2;
  while (--i) {                 /* i 由最大码长 g 开始递减 */
    *xp++ = (j += *p++);   /* *p 是当前位长编码数,当前位长
编码起始位置后移 *p 个位置是下个码长的起始位置 */
  }
DEBG("huft5 ");
/* v[]中存放按位长度顺次排列的每个编码(同一长度中编码连
续递增)代表的值(0~n-1) */
  p = b;  i = 0;
  do {
    if ((j = *p++) ! = 0)
      v[x[j]++] = i;
  } while (++i < n);
DEBG("h6 ");
/* 以下生成 huffman 编码以及每个编码的解码表项 */
  x[0] = i = 0;               /* 第一个编码是 0 开始的 */
  p = v;                    /* p 指向首个编码的值, 以后按编码位序
不断指向后续编码值 */
  h = -1;                       /* no tables yet--level -1 */
  w = -l;                       /* bits decoded == (l * h) */
  u[0] = (struct huft *)NULL;   /* just to keep compilers hap-
py */
  q = (struct huft *)NULL;      /* ditto */
  z = 0;                        /* ditto */
DEBG("h6a ");
/* 主循环遍历位长度 (k 初始是最小位长,g 是最大位长),huff-
man 编码以位长 l 被分段,开始 l 位为 0 级表的索引,接着的 1~l
位(根据实际码长)为 1 级表索引,以此类推,如果码长不超过 l
位,则没有次级表,0 级表项就是叶节点;如果码长范围 l+1~2l,
则 1 级表项是叶节点,以此类推 */
  for (; k <= g; k++)  //主 for 循环一次处理一个码长
  {
DEBG("h6b ");
    a = c[k];  //a=位长为 k 的编码数
//while 循环处理当前码长 k 的剩余 a 个编码。i 是当前的 k 位
//huffman 编码,对应的值是 *p
    while (a--) {
DEBG("h6b1 ");
/* 第三级循环,根据解码 k 位编码的需要创建足够层级的子表。
其中 w 对应上一级表的解码码长,w+l 是当前级别表的最大解
码码长,当当前编码码长 k>w+l,就需要创建子表来解码多出的
k-(w+l)位(前提是 k-w-l<=l,否则还要循环创建再下一级表,以
此类推)。 */
      while (k > w + l)
      {
DEBG1("1 ");
      h++;    /* 当前级别,0 级表,1 级表,…… 0 级表只有一张 */
        w += l;      /* w=w+l,上级表的最大解码码长 */
/* 计算创建的表的大小上限,索引位长上限 z 不大于 min(l,g-
```

```
w) */
      z = (z = g - w) > (unsigned)l ? l : z;
/* 先尝试 j=k-w 位，如果前 w 位中包含 k+1 位编码的前缀(1 << (j = k - w)) > a + 1),扩展要创建的表的位长 j,直到 l */
      if ((f = 1 << (j = k - w)) > a + 1)     /* f=1<<j,是(k-w)位索引的初始表长+1 */
      {
/* 当前码长 k 的剩余编码数 a<f-1，而扩展 1 个索引位刚好能容纳 k+1 位码长的所有编码，
也就是说当前剩余 k 位码长编码和 k+1 位码长的编码有共同的 w 位前缀，必须共存于一个 h 级子表 (h 级表由同一个 h-1 级表项指向),则扩展表索引位,如此做循环计算判断扩展,直到该条件不满足 */
DEBG1("2 ");
        f -= a + 1;   /* deduct codes from patterns left */
        xp = c + k;
        while (++j < z)       /* 循环判断扩展 */
        {
          if ((f <<= 1) <= *++xp)
            break;            /* enough codes to use up j bits */
          f -= *xp; /* else deduct codes from patterns */
        }
      }
DEBG1("3 ");
      z = 1 << j;       /* z 是最后确定的表长,j 是索引位数(也就是本级表解码码长) */
/* 创建 huft 表,q 指向该表,注意实际创建的表长为 z+1(多分配一个 huft 表项),解码表项是从 q+1 项开始的,表项 q 是 link 项,指向下一次生成的  表(最后生成的各表会通过其表首
link 项 t->v.t 连接成链表,在 huft_free()时通过该链找到各表并释放空间) */
      if ((q = (struct huft *)malloc((z + 1)*sizeof(struct huft))) ==
          (struct huft *)NULL)
      {
/* 分配内存失败。如 h>0,u[0]指向第一级表的开始表项(它前面一项--u[0]是连接后续 huft 表的 link 项),huft_free(u[0])会找到连接指针(--u[0])->v.t 并释放所有已分配的 huft 表 */
        if (h)
          huft_free(u[0]);
        return 3;             /* 内存不足返回 3 */
      }
DEBG1("4 ");
      hufts += z + 1;         /* track memory usage */
/* t 指向上次分配的 huft 表的 link 项的连接指针,*t=q+1,将本次分配的表和上次的表连接起来 */
      *t = q + 1;
/* huft 数组线性链表指针指向本次分配的表 */
      *(t = &(q->v.t)) = (struct huft *)NULL; //t = &(q->v.t),
// t 指向本次分配表的 link 项的连接指针
/* u[0]~u[h]记录了当前编码 i 的各级解码表的地址 */
      u[h] = ++q;  /* 当前表是从 q+1 开始的,q 是 link 项 */
DEBG1("5 ");
      /* connect to last table, if there is one */
/* 当 h>0,设置编码 i 的上一级解码表项,指向刚分配的子表。先初始化 huft 结构 r,然后设置到上一级表项中 */
      if (h)
      {
/* x[h]=i;保存了本级表的起始编码,也是本级表各项编码的共同前缀(起始编码的扩展位总是从 0 开始的),作为下面判断是否回溯的依据(当当前编码 i 的前缀部分和 x[h]不等时,表明前缀变化,需要回溯到上一级表的下一个表项 */
        x[h] = i;             /* save pattern for backing up */
/* r.b 是根据上一级表项搜寻本表时需要丢弃的位数(丢弃已解码的码长,再读入下一级表的索引码长) */
        r.b = (uch)l;         /* bits to dump before this table */
/* j 是子表码长,r.e 设置为 16+j,表示上级表项有子表,子表索引码长为 j=r.e-16 */
        r.e = (uch)(16 + j);  /* bits in this table */
/* 作为非叶节点,r.v.t 指向子表 */
        r.v.t = q;            /* pointer to this table */
/* i 是当前编码,右移(w-l)位后得到的 j 是上级表项的索引 */
        j = i >> (w - l);     /* (get around Turbo C bug) */
/* 设置上一级表项。u[h-1]是上一级解码表指针,j 是索引码,u[h-1][j]指向编码 i 的上一级表项 */
        u[h-1][j] = r;        /* connect to last table */
      }
DEBG1("6 ");
    }     /* end while k>w+l */
DEBG("h6c ");

    /* set up table entry in r */
/* 下面要设置编码 i 在本级表中的表项，先在 huft 结构临时变量 r 中进行初始化,然后设置到表项中。*/
/* r.b 是访问本表项解码后要丢弃的位数，因为子表可能经过 k-w~l 的扩展，所以各表项对应的码长不一定相同,r.b 可能是小于本级表的索引码长的,多读入的位不能丢弃 */
    r.b = (uch)(k - w);
/* 超出值范围则标记为无效编码。一共有 n 个编码,对应 n 个值 v[0]~v[n-1],p 指向当前编码 i 对应的值的 v[]元素。如果 p>=v+n,则超出值的范围,把当前表项的 e 设置为 99,r.e=99 */
    if (p >= v + n)
      r.e = 99;               /* out of values--invalid code */
/* s 为 simple value(简单值)的数量,值 0~(s - 1)为简单值,这部分值没有附加位。检查当前编码的值 *p,如果是<256 的简单值,r.e = 16;如果是 256,r.e=15,这是块结束码。r 作为叶节点时,r.v.n 为编码代表的值 */
    else if (*p < s)
    {
      r.e =  (uch)(*p < 256 ? 16 : 15);    /* 256 is end-of-block code */
      r.v.n = (ush)(*p);    /* simple code is just the value */
      p++;                  /* one compiler does not like *p++ */
    }
```

```
/* 当前编码的值不是简单值(基本值+附加值),r.e 是解码时需要读入的附加位长度,r.v.n 是基本值,解码的值=r.v.n+(r.e 位附加位的值),参考 RFC1951 */
    else
    {
   r.e = (uch)e[*p - s];   /* non-simple--look up in lists */
     r.v.n = d[*p++ - s];
    }
DEBG("h6d ");

/* 当本表索引位数>k-w 时,当前位索引的低(k-w)位相同的表项(间距 f)都置为相同的 r 值 */
    f = 1 << (k - w);
/* 设置当前编码 i 在本级子表中的表项。j=i>>w 是编码 i 在当前子表中的索引,q [j]=r 把 i 对应的表项设置为上面初始化的 r 的值。另外,因为子表可能经过 k-w~l 的扩展,比如 4 位编码 0000,扩展为 5 位索引后,读入 5 位进行解码,出现的可能是 00000,也可能是 10000,最高位本是下次解码用的,因索引位数的关系也被带进来了,因为要顺利解码必须所有低位为 0000 的索引项都置为相同的 r,f 是相同 r 的表项的间距,解码后要丢弃的位长 r.b=4,保留多读的 1 位下次解码用 */
    for (j = i >> w; j < z; j += f)
      q[j] = r;
/* 递增 Huffman 编码的值。这里的编码是反序的,不是通常那样递增右边的位。当前编码长度是 k,下面的语句使 i 在低 k 位上实现从左到右的递增操作,左边是 LSB 位。比如 0000000 首先递增为 1000000,然后是 0100000,然后是 1100000;当码长增加 1 位时,在 k+1 位上继续递增,1100000->0010000->10010000。i 是按照编码位长顺次变化的,和 v[]中的值顺序一致,因此 *p 就是编码 i 表示的值 */
//低 k 位,从左向右检测各位,碰到 1 就翻转变 0,直到碰到 0
    for (j = 1 << (k - 1); i & j; j >>= 1)        i ^= j;
//上面碰到的 0 位翻转为 1,如此实现了反位序(左->右,高位->//低位)的加 1 操作。
    i ^= j;
/* 这里检查是否需要回溯到上一级解码表。这一般发生于当前编码的前缀发生变化时,要回溯到上一级表的下一个表项,建立新的子表,必要时可能要多级回溯,所以用循环判断。这里也可以看出 x[0]、x[1]……x[h]的作用,就是保存了各级表的上一级表项的编码(也等于本级表的起始编码),其中 x[0]=0。这里循环检测编码前缀,不同就向上一级回溯,直到相同为止,最多回溯到 x[0]的时候肯定会循环终止的,此时已到达了 0 级表。 */
    /* backup over finished tables */
    while ((i & ((1 << w) - 1)) ! = x[h])
    {
/* 回溯:当前层级 h 减 1,解码长度 w 减 l,当前级的表指针 q 不必修改,因为返回到上面的 while a--循环中执行 while k>w+l 循环(因 w-=1 后循环条件已符合)时会重新生成当前编码的子表的(q 会指向新的表) */
      h--;                    /* don't need to update q */
      w -= l;
    }
DEBG("h6e ");
   }        /* end while a-- */
DEBG("h6f ");
 }         /* end for ;k<=g;k++ */

DEBG("huft7 ");

 /* Return true (1) if we were given an incomplete table */
 return y ! = 0 && g ! = 1;
}
```

3.2 Huffman 编码表空间释放

参数 t 指向第一次分配的 huft 表，也是 huft 表的链表的表首。

```
STATIC int huft_free(t)
struct huft *t;         /* table to free */
{
 register struct huft *p, *q;


/* 循环遍历 huft 表的链表,释放各表占用的内存空间。t[-1]是表的 link 项,t[-1]->v.t 指向下一张 huft 表 */
 p = t;
 while (p ! = (struct huft *)NULL)
 {
   q = (--p)->v.t;
   free((char*)p);
   p = q;
 }
 return 0;
}
```

4 结语

编码是一项工程，读码又何尝不是，在此分析的代码是 HSF 解码表的创建过程，也是 Linux 内核解压的准备阶段。理解它，才能理解下篇介绍的 Linux 内核解压过程代码。

参考文献

[1] Linux 内核 2.4.0 源代码. 可以从 www.kernel.org 下载.
[2] RFC1951 (Request For Comment 1951) .
[3] 程序员实用算法.

（收稿日期：2011-04-01）