DOI:10.16184/j.cnki.comprg.2011.17.003

Linux 运行前探秘之四
——内核解压缩（三）

徐 炜

摘 要：分析了 Linux 内核解压过程和算法，并分析了关键的源代码。

关键词：inflate；Huffman 编码；stored block；fixed block；dynamic block

1 概述

当 Huffman 解码表创建完毕，则“万事俱备，只等解码”了。函数 gunzup () 主要负责文件格式验证，读取文件头，以及解码结束后验证 crc 值和文件原始长度，解码工作是通过调用 infalte () 函数完成的。inflate () 解码过程以块为单位进行(deflate 压缩数据是分块的)，循环调用 inflate_block () 解码一个块，直到最后一块。每个块看做一个长位串，inflate_block () 首先读取块的第 1 位，这是结束块的标记，为 1 表示当前解码的块是最后一个块 (这个值通过 e 返回到 inflate () 中作为循环结束的条件)；接着读入 2 位块类型码，根据其值的不同(00-stored block,01-fixed block,10-dynamic block) 分别调用 inflate_stored ()、inflate_fixed ()、inflate_dynamic () 进行块解码。

2 代码分析

2.1 解压缩主程序 gunzip ()

decompress_kernel () 调用该函数进行内核解压缩：

```
static int gunzip(void)
{
    uch flags;
    unsigned char magic[2]; /* magic header */
    char method;
    ulg orig_crc = 0;   /* 原始 crc 值 */
    ulg orig_len = 0;   /* 原始数据长度(压缩前的内核长度) */
    int res;
    //文件开头两字节是魔数(格式标志),第 3 个字节是压缩方
//法,这里应该是 8,表示 deflate
    magic[0] = (unsigned char)get_byte();
    magic[1] = (unsigned char)get_byte();
    method = (unsigned char)get_byte();
    if (magic[0] ! = 037 ||
       ((magic[1] ! = 0213) && (magic[1] ! = 0236))) {
            error("bad gzip magic numbers");
            return -1;
    }
    /* We only support method #8, DEFLATED */
    if (method ! = 8)  {
            error("internal error, invalid method");
            return -1;
    }
    //读取标志字节
    flags  = (uch)get_byte();
    //输入是加密的,报错返回
    if ((flags & ENCRYPTED) ! = 0) {
            error("Input is encrypted\n");
            return -1;
    }
    //multi-part 输入,报错返回
    if ((flags & CONTINUATION) ! = 0) {
            error("Multi part input\n");
            return -1;
    }
    //无效标志,报错返回
    if ((flags & RESERVED) ! = 0) {
            error("Input has invalid flags\n");
            return -1;
    }
    //以下 4 个字节是时间戳
    (ulg)get_byte();     /* Get timestamp */
    ((ulg)get_byte()) << 8;
    ((ulg)get_byte()) << 16;
    ((ulg)get_byte()) << 24;
    //以下两字节是 extra flag 和 os type,被忽略
    (void)get_byte();  /* Ignore extra flags for the moment */
    (void)get_byte();  /* Ignore OS type for the moment */
    //flag 有 EXTRA_FIELD 标志,读入附加字节
    if ((flags & EXTRA_FIELD) ! = 0) {
            unsigned len = (unsigned)get_byte(); //首先的两字节
//是附加域的长度
            len |= ((unsigned)get_byte())<<8;
            while (len--) (void)get_byte(); //读入 len 字节
    }
    /* 读入原始文件名并丢弃 */
    if ((flags & ORIG_NAME) ! = 0) {
            /* Discard the old name */
            while (get_byte() ! = 0) /* null */ ;
    }
    /* 如果有文件注释,读入并丢弃直到遇见 0 为止 */
    if ((flags & COMMENT) ! = 0) {
```

```
        while (get_byte() ! = 0) /* null */ ;
    }
    /* 调用 inflate()解压缩,返回 0 是成功;其它值表示出错 */
    if ((res = inflate())) {
        switch (res) {
        case 0:
            break;
        case 1:
            error("invalid compressed format (err=1)");
            break;
        case 2:
            error("invalid compressed format (err=2)");
            break;
        case 3:
            error("out of memory");
            break;
        default:
            error("invalid compressed format (other)");
        }
        return -1;
    }
    /* Get the crc and original length */
    /* crc32  (see algorithm.doc)
     * uncompressed input size modulo 2^32
     */
  //压缩数据之后是原始的 4 字节的 crc32 的值,以及 4 字节的
//原始数据长度
    orig_crc = (ulg) get_byte();
    orig_crc |= (ulg) get_byte() << 8;
    orig_crc |= (ulg) get_byte() << 16;
    orig_crc |= (ulg) get_byte() << 24;
    orig_len = (ulg) get_byte();
    orig_len |= (ulg) get_byte() << 8;
    orig_len |= (ulg) get_byte() << 16;
    orig_len |= (ulg) get_byte() << 24;
    /* 把原始的 crc32 值和解压缩过程计算出的 crc 值对比,原
始的数据长度和解压缩输出的数据长度对比, 一致的话表示解
压缩成功 */
    if (orig_crc ! = CRC_VALUE) {
        error("crc error");
        return -1;
    }
    if (orig_len ! = bytes_out) {
        error("length error");
        return -1;
    }
    return 0;
}
```

2.2 inflate () 解码函数

压缩数据是分块的，inflate () 函数通过一个主循环，每次循环调用 inflate_block () 解码一个块。块解码前通过 gzup_mark () 保存堆指针，解码完一个块后调用 gzip_release () 恢复堆指针，这样就达到了释放解码过程中分配的堆内存的目的。inflate_block () 返回一个标志 e，如果非 0 表示刚解完的块是最后一个块，循环结束。

```
STATIC int inflate()
/* decompress an inflated entry */
{
  int e;               /* e 保存最后块标记 */
  int r;               /* r 保存返回值 */
  unsigned h;          /* maximum struct huft's malloc'ed */
  void *ptr;
 /* 初始化滑动窗口指针和位缓存。每次解码一个块时,这些值
先被拷贝到块解码函数的局部变量中,当一个块解码结束后,再
把当前的值重新保存回这 3 个变量中,以便在块间传递。因此,
在解码一个块时, 是可以引用上一个块的滑动窗口剩余数据的
*/
  wp = 0;              //wp 是宏定义,即 outcnt
  bk = 0;              //bk 是位缓存的数据位数
  bb = 0;              //bb 是位缓存的数据部分
  /* 循环,每次解压一个块,直到最后一个块 */
  h = 0;
  do {
    hufts = 0;
    gzip_mark  (&ptr); //解压块以前 ptr 保存堆指针
free_mem_ptr
    if ((r = inflate_block(&e)) ! = 0) { //调用 inflate_block(&e)解
//压一个块,e 保存返回的标志
      gzip_release(&ptr);
      return r;              //解压块出错,返回非 0 值
    }
    gzip_release(&ptr);   //解压完一个块以后(或者解压出错返
//回时)恢复 free_mem_ptr,相当于释放已使用的堆内存
    if (hufts > h)
      h = hufts;
  } while (! e);  //e=1 时表示已解完最后一个块,循环结束;否则
//(e=0)继续解压下一个块。
  /* Undo too much lookahead. The next read will be byte
aligned so we
   * can discard unused bits in the last meaningful byte.
   */
  while (bk >= 8) {
    bk -= 8;
    inptr--;
  }
  flush_output(wp);  //把滑动窗口中剩余的数据输出掉
  /* return success */
#ifdef DEBUG
  fprintf(stderr, "<%u> ", h);
#endif /* DEBUG */
  return 0;          //解压成功,返回 0
}
```

2.3 解码一个块 inflate_block ()

该函数读取块开头的 2 位值判断块类型，根据不同类型分

别调用 inflate_stored ()、inflate_fixed ()、inflate_dynamic () 进行解码，并通过参数 *e 返回当前解码的块是否为最后一个块的标志。

```
STATIC int inflate_block(e)
int *e;                 /* last block flag */
/* decompress an inflated block */
{
  unsigned t;           /* block type */
  register ulg b;       /* bit buffer */
  register unsigned k;  /* number of bits in bit buffer */
  DEBG("<blk");
  /* make local bit buffer */
  b = bb;
  k = bk;
  /* 块的最开始 1 位表示是否为最后一个块(e=1 时表示是末尾块) */
  NEEDBITS(1)
  *e = (int)b & 1;
  DUMPBITS(1)
  /* 接下来两位指示块的类型(3 种) */
  NEEDBITS(2)
  t = (unsigned)b & 3;
  DUMPBITS(2)
  /* restore the global bit buffer */
  bb = b;
  bk = k;
  /* inflate that block type */
  if (t == 2) //t==2 表示 dynamic 块,调用 inflate_dynamic()
    return inflate_dynamic();
  if (t == 0) //t==0 表示 stored 块,调用 inflate_stored()
    return inflate_stored();
  if (t == 1) //t==1 表示 fixed 块,调用 inflate_fixed()
    return inflate_fixed();
  DEBG(">");
  /* bad block type */
  return 2;
}
```

2.4 处理 stored block (非压缩块)

stored 块中是未编码的数据，无需解码。首先，读取开头 2 字节的块长度 n 和 2 字节的块长度反码进行验证，然后读取后续的 n 字节并直接拷贝到滑动窗口中。slide 是滑动窗口缓存 Windows 的宏定义。

```
STATIC int inflate_stored()
/* "decompress" an inflated type 0 (stored) block. */
{
  unsigned n;           /* number of bytes in block */
  unsigned w;           /* current window position */
  register ulg b;       /* bit buffer */
  register unsigned k;  /* number of bits in bit buffer */
DEBG("<stor");
  /* 块解码前,全局量中的位缓存、滑动窗口指针数据保存到局部变量中 */
  b = bb;                       /* 位缓存:b-位数据;k-位长度 */
  k = bk;
  w = wp;                       /* 窗口的当前指针 */
  /* 非压缩块是字节边界开始的(其它块的开始位可能是跨字节的),所以要丢弃多余位,直到字节边界为止 */
  n = k & 7;
  DUMPBITS(n);
  /* 开始两字节是块长度 */
  NEEDBITS(16)
  n = ((unsigned)b & 0xffff);
  DUMPBITS(16)
  /* 紧接着的两字节是块长度的反码(one's complement),这里先验证一下 */
  NEEDBITS(16)
  if (n != (unsigned)((~b) & 0xffff))
    return 1;                   /* 验证不通过返回 1*/
  DUMPBITS(16)
  /* 主循环实现数据的直接拷贝(无需解码),每次拷贝 1 字节到滑动窗口,拷贝完 WSIZE,就把窗口数据输出,窗口位置 w=0 */
  while (n--)
  {
    NEEDBITS(8)
    slide[w++] = (uch)b;  //slide[w++]即 window[w++]
    if (w == WSIZE)   //滑动窗口数据满则输出
    {
      flush_output(w);
      w = 0;
    }
    DUMPBITS(8)
  }
  /* 块解码结束后局部变量值保存到全局变量中 */
  wp = w;                       /* 保存窗口指针 */
  bb = b;                       /* 保存位缓存 */
  bk = k;
  DEBG(">");
  return 0;
}
```

2.5 解压 fixed block (固定 Huffman 编码块)

函数首先设置字符/长度码 (literal/length codes) 的 Huffman 编码长度数组 (即对应 0~287 的各编码长度)，并根据此长度数组创建解码表 tl []，bl 是 tl [] 表的索引位长；然后以同样的方法创建距离码 (distance codes) 的 Huffman 解码表 td []，bd 是 td [] 的索引位长。最后使用两张解码表调用 inflate_code () 进行解码。这里的 fixed 的意思是各值的编码长度，是固定的，从而编码也是固定的。

```
STATIC int inflate_fixed()
{
  int i;                /* temporary variable */
  struct huft *tl;      /* literal/length code table */
  struct huft *td;      /* distance code table */
```

```
  int bl;              /* tl[]表的索引位长 */
  int bd;              /* td[]表的索引位长 */
  unsigned l[288];     /* l[]是编码长度表 */
DEBG("<fix");
  /* 设置 literal/length 码的编码长度表,使用 literal/length 码的值的范围是 0~287，其中 0~255 表示字符,256 表示块结束符,257~287 表示长度码(需要组合附加位) 。以下设置 0~143 的编码长度是 8,144~255 的编码长度是 9,256~279 是 7,280~287 是 8。所谓 fixed 也就是指编码长度预先固定,从而 Huffman 编码也固定的意思 */
  for (i = 0; i < 144; i++)
    l[i] = 8;
  for (; i < 256; i++)
    l[i] = 9;
  for (; i < 280; i++)
    l[i] = 7;
  for (; i < 288; i++)   /* make a complete, but wrong code set */
    l[i] = 8;
  bl = 7;
//构建 literal/length 码的 huffman 解码表,tl 指向生成的表,bl 是
//解码时开始读入的位数(一级表的码长)
  if ((i = huft_build(l, 288, 257, cplens, cplext, &tl, &bl)) ! = 0)
    return i;
  /* 设置 distance 码的编码长度表,distance 码采用 0~29 的数字,huffman 编码长度固定为 5 */
  for (i = 0; i < 30; i++)   /* make an incomplete code set */
    l[i] = 5;
  bd = 5;
//构建 distance 码的 huffman 解码表,td 指向生成的表,bd 是解
//码时开始读入的位数(一级表的码长)
  if ((i = huft_build(l, 30, 0, cpdist, cpdext, &td, &bd)) > 1)
  {
    huft_free(tl);

    DEBG(">");
    return i;
  }
  /* 利用两张 huft 表开始解码,直到块结束 */
  if (inflate_codes(tl, td, bl, bd))
    return 1;
  /* 释放解码表的空间，实际上是空操作。堆空间是在 inflate_block()后调用 gzip_release(&ptr);释放的 */
  huft_free(tl);
  huft_free(td);
  return 0;
}
```

2.6 解压 dynamic block (动态 Huffman 编码块)

动态块和静态块不同的是，literal/length 码和 distance 码的编码长度不是固定的，需要从块中读入，并且读入的编码长度本身是用 Huffman 编码的。因此流程是这样的：先读入长度码个数 nb,然后循环读入 nb 个长度码的长度值，构建长度码的解码表，然后读入 n = nl + nd (分别为 literal/length 码和 distance 码的数量) 个长度码,解码获取 n 个长度值存入 ll [] 数组，根据长度值分别构建 literal/length 码和 distance 码的解码表，最后使用解码表调用 inflate_codes () 进行解压缩。这个流程和 fixed 块不同之处就在于 fixed 块的编码长度是静态设置的，而 dynamic 是读入长度编码的长度数据后构建长度码解码表，再读入长度码，解码成长度值获取的。一旦编码长度确定后，下面的流程就相同了。

```
STATIC int inflate_dynamic()
/* decompress an inflated type 2  (dynamic Huffman codes) block. */
{
  int i;             /* temporary variables */
  unsigned j;
  unsigned l;     /* last length */
  unsigned m;     /* mask for bit lengths table */
  unsigned n;     /* number of lengths to get */
  struct huft *tl;      /* literal/length code table */
  struct huft *td;      /* distance code table */
  int bl;               /* lookup bits for tl */
  int bd;               /* lookup bits for td */
  unsigned nb;          /* number of bit length codes */
  unsigned nl;          /* number of literal/length codes */
  unsigned nd;          /* number of distance codes */
#ifdef PKZIP_BUG_WORKAROUND
  unsigned ll [288+32]; /* literal/length and distance code lengths */
#else
  unsigned ll [286+30]; /* literal/length and distance code lengths
#endif
  register ulg b;       /* bit buffer */
  register unsigned k;  /* number of bits in bit buffer */
DEBG("<dyn");
  /* make local bit buffer */
  b = bb;
  k = bk;
  /* 首先读入的 5 位二进制表示 (literal/length 编码数-257)的值 */
  NEEDBITS(5)
  nl = 257 + ((unsigned)b & 0x1f);      /* 得到 literal/length 编码的数目 257~286 */
  DUMPBITS(5)
/* 接着读入的 5 位二进制表示(distance 编码数-1)的值 */
  NEEDBITS(5)
  nd = 1 + ((unsigned)b & 0x1f);        /* 得到 distance 码的数目 1~32 */
  DUMPBITS(5)
/* 再读入 4 位是编码长度码的数目-4*/
  NEEDBITS(4)
  nb = 4 +  ((unsigned)b & 0xf);    /* 得到位长度编码的数目 4~19 */
```

```
  DUMPBITS(4)
#ifdef PKZIP_BUG_WORKAROUND
  if (nl > 288 || nd > 32)
#else
  if (nl > 286 || nd > 30)
#endif
    return 1;                   /* bad lengths */
DEBG("dyn1 ");
  /* read in bit-length-code lengths */
  /* 读入 nb 个位长度编码的长度。位长度编码的值范围是 0~
18,border[]数组指示了读入的编码顺序,border[j]表示第 j 个编
码的值 */
  for (j = 0; j < nb; j++)
  {
    NEEDBITS(3)  //读入 3 位编码长度(因此最大长度不超过 7)
    ll[border[j]] = (unsigned)b & 7;  //设置 ll[border[j]]为值 bor
//der[j]对应的编码的长度
    DUMPBITS(3)
  }
  for (; j < 19; j++)
    ll[border[j]] = 0; //剩余没用到的编码长度=0
DEBG("dyn2 ");
  /* build decoding table for trees--single level, 7 bit lookup
*/
/* 利用上述得到的编码长度数组 ll[]构造位长度编码的解码表 tl
(长度为 0 的不生成编码),只有一级,索引位长度为 bl=7,没有
附加位 */
  bl = 7;
  if ((i = huft_build(ll, 19, 19, NULL, NULL, &tl, &bl)) ! = 0)
  {
    if (i == 1)
      huft_free(tl);
    return i;                   /* incomplete code set */
  }
DEBG("dyn3 ");
  /* read in literal and distance code lengths */
/* 利用上面生成的解码表, 读入 literal 和 distance 码的长度的
编码,一共是 n=nl+nd 个长度码 */
  n = nl + nd; //n 是值的数量(literal 码和 distance 码的数量和)
  m = mask_bits[bl]; //bl 位的位掩码
  i = l = 0;
  while ((unsigned)i < n) //主循环 n 次
  {
    NEEDBITS((unsigned)bl) //读入 bl 位码长
    j = (td = tl + ((unsigned)b & m))->b;//td 为解码表的 huft
//表项,j=td->b 为该表项对应的编码位数
    DUMPBITS(j) //丢弃此 j 位(j 一般就等于 bl)
    j = td->v.n; //获取编码所表示的值
    if (j < 16)    /* j:0~15 表示 i 的编码长度值,填入 ll[i] */
      ll[i++] = l = j;          /* 最近的长度值保存在 l 中 */
    else if  (j == 16)          /* j=16 表示上一个长度值 l 重复
3~6 次,重复次数=3+(后面的 2 位附加位值) */
    {
      NEEDBITS(2)          //读入 2 位
      j = 3 + ((unsigned)b & 3); //j=重复次数
      DUMPBITS(2)
      if ((unsigned)i + j > n) //i+j 不能大于 n
        return 1;
      while (j--)       //l 中保存的前一个长度值重复 j 次(i 以及 i
//后面的 j-1 个值的编码长度都是 l)
        ll[i++] = l;
    }
    else if  (j == 17)          /* j=17 表示长度值 0 重复 3~10
次,重复次数=3+(后面的 3 位附加位值)*/
    {
      NEEDBITS(3)  //读入 3 位附加位
      j = 3 + ((unsigned)b & 7);  //j=重复次数 3~10
      DUMPBITS(3)
      if ((unsigned)i + j > n) //i+j 不能大于 n
        return 1;
      while (j--)           //i 开始的 j 个编码的长度都是 0
        ll[i++] = 0;
      l = 0;
    }
    else  /* j == 18: 0 长度重复 11~138 次,需读入 7 位附加位 */
    {
      NEEDBITS(7)
      j = 11 + ((unsigned)b & 0x7f);  /* 重复次数=11+7 位附加
位的值 */
      DUMPBITS(7)
      if ((unsigned)i + j > n)
        return 1;
      while (j--)
        ll[i++] = 0;          //i 开始的 j 个编码的长度都是 0
      l = 0;
    }
  }
DEBG("dyn4 ");
  /* free decoding table for trees */
  huft_free(tl);
DEBG("dyn5 ");
  /* restore the global bit buffer */
  bb = b;
  bk = k;
DEBG("dyn5a ");
  /* build the decoding tables for literal/length and distance
codes */
  bl = lbits;
//ll[]中现已保存有 0~286 literal/length 码和 0~30 distance 码
//的长度,可以开始构建两个 huft 解码表了
/* 首先构建的是 literal/length 码的解码表,长度数组从 ll 开始,
共 nl 个编码长度,带附加位的编码是从 257 开始;tl 是生成的解
码表指针,bl 是索引位长 */
  if ((i = huft_build(ll, nl, 257, cplens, cplext, &tl, &bl)) ! = 0)
  {
DEBG("dyn5b ");
```

```
    if (i == 1) {
      error(" incomplete literal tree\n");
      huft_free(tl);
    }
    return i;                   /* incomplete code set */
  }
DEBG("dyn5c ");
  bd = dbits;
//接着构建 distance 码的解码表,长度数组从 ll+nl 开始,共 nd
//个编码长度,带附加位的编码从 0 开始;返回 td 是解码表指针,
//bd 为索引位长
  if ((i = huft_build(ll + nl, nd, 0, cpdist, cpdext, &td, &bd)) !=
0)
  {
DEBG("dyn5d ");
    if (i == 1) {
      error(" incomplete distance tree\n");
#ifdef PKZIP_BUG_WORKAROUND
      i = 0;
    }
#else
      huft_free(td);
    }
    huft_free(tl);
    return i;                   /* incomplete code set */
#endif
  }
DEBG("dyn6 ");
  /* 使用解码表 tl、td 和索引位长 bl、bd 进行解压缩 */
  if (inflate_codes(tl, td, bl, bd))
    return 1;
DEBG("dyn7 ");
  /* free the decoding tables, return */
  huft_free(tl);
  huft_free(td);
  DEBG(">");
  return 0;
}
```

2.7 解码函数 inflate_codes ()

参数说明：tl 指向字符/长度码的 huffman 表，td 指向距离码的 huffman 表；bl 是字符/长度码的初始读入位长（也就是 huffman 表的一级表的位数），bd 是距离码的初始读入位长。

```
STATIC int inflate_codes(tl, td, bl, bd)
struct huft *tl, *td;   /* tl[]:字符/长度码 huffman 解码表 td[]:
距离码 huffman 解码表 */
int bl, bd;
/* bl:tl[]表解码的码长;bd:td[]表解码码长 */
/* inflate (decompress) the codes in a deflated (compressed)
block.Return an error code or zero if it all goes ok. */
{
  register unsigned e;  /* e 是表入口标志或者附加位位数（取
决于它的值)*/
  unsigned n, d;      /* length and index for copy */
  unsigned w;         /* current window position */
  struct huft *t;       /* 指向 huffmanbian 表项(huft 结构) */
  unsigned ml, md;   /* ml、md 是分别是码长 bl、bd 的位掩码 */
  register ulg b;       /* b 是位缓存(为提高存取速度,使用了寄
存器变量) */
  register unsigned k;  /* k 是位缓存中的有效数据位数 */
  /* 解码前把位缓存和滑动窗口当前指针的全局量保存到局部
变量中 */
  b = bb;           /* 初始化位缓存 b。bb 中存有解码上个块后遗
留的位,初始为 0 */
  k = bk;       /*k 初始化为 bb 中的有效数据位数,初始为 0*/
  w = wp;                       /* 滑动窗口当前位置,初始为 0 */
  /* inflate the coded data */
  ml = mask_bits[bl];           /* 码长 bl 的位掩码 */
  md = mask_bits[bd];           /* 码长 bd 的位掩码   */
  for (;;)          /* 主循环,知道块结束 */
  {
    NEEDBITS ((unsigned)bl)       /* 开始时读取 bl 位,b&ml 就
是 bl 位的二进制码 */
    if ((e = (t = tl + ((unsigned)b & ml))->e) > 16)
//e>16,表明还有子表
//以下循环读取子表项(如果子表项还指向子表的话,继续循
//环)
      do {
        if (e == 99)  //e==99,报错
          return 1;
        DUMPBITS(t->b)      //先丢弃先前读入的 t->b 位
        e -= 16;          //e=e-16 指示接着要读入的位数(作为子
//表索引)
        NEEDBITS(e)       //读入 e 位
      } while ((e = (t = t->v.t + ((unsigned)b & mask_bits[e]))
->e) > 16); //e=t=子表项,t->e>16,表明还有子表,继续 do..
//while{}循环
    DUMPBITS(t->b)
    //执行到此,t 所指向的一定是叶节点(t->e<=16 时,表明不
//存在子表)
    if (e == 16)                /* e==16,这是个字符,直接输出字
符值 t->v.n*/
    {
      slide[w++] = (uch)t->v.n; /* slide 指向滑动窗口 window
[],w 是其当前位置,这里把字符值输出到滑动窗口,并递增 w */
      Tracevv((stderr, "%c", slide[w-1]));
      if (w == WSIZE) //滑动窗口已满,窗口中的数据全部输出
//到目标缓冲区,并修改窗口位置 w=0[注意:原先的数据还存在
//于窗口中]
      {
        flush_output(w);
        w = 0;
      }
    }
    else                        /* 否则(e<=15),是一个 EOB(块结束)
标志或者长度码 */
```

```
  {
    /* e==15,块结束标志,则退出 for 循环 */
    if (e == 15)
      break;
    /* e<15 时,t->v.n 指示长度范围的基本值,e 指示获取附加值要读取的位数,读取 e 位 */
    NEEDBITS(e)
    n = t->v.n + ((unsigned)b & mask_bits[e]); //计算得到长
//度 n=基本值+附加值
    DUMPBITS(e);  //丢弃 b 中的 e 位
    /* 上面读入的是长度码,所以接下来要解的是距离码(两者成对出现),先读取 bd 位,解码步骤和长度码类似 */
    NEEDBITS((unsigned)bd)
    if ((e = (t = td + ((unsigned)b & md))->e) > 16)
      do {
        if (e == 99)
          return 1;
        DUMPBITS(t->b)
        e -= 16;
        NEEDBITS(e)
      } while ((e = (t = t->v.t + ((unsigned)b & mask_bits[e]))
->e) > 16);
    DUMPBITS(t->b)
    NEEDBITS(e)
    //根据叶节点计算距离,t->v.n 是基本值,b&mask_bits[e]是
//附加值,两者相加是相对当前位置的距离,因此 d 是指示字串在
//滑动窗口中的起始位置
    d = w - t->v.n - ((unsigned)b & mask_bits[e]);
    DUMPBITS(e)
    Tracevv((stderr,"\\[%d,%d]", w-d, n));
    /* 以下循环根据将窗口中位置 d 处的长度 n 的字串拷贝到当前位置 w,并移动 w 到下个位置 */
    do {
/* 每次循环拷贝 e 字节。d &= WSIZE-1 是环回一下,另外如果经过窗口边界的话,要分两次 copy(这也是设置循环的目的) */
      n -= (e = (e = WSIZE - ((d &= WSIZE-1) > w ? d :
w)) > n ? n : e);
#if ! defined(NOMEMCPY) && ! defined(DEBUG)
      if (w - d >= e)         /* (this test assumes unsigned
comparison) */
      {
        memcpy(slide + w, slide + d, e);
        w += e;
        d += e;
      }
      else    /* do it slow to avoid memcpy() overlap */
#endif /* ! NOMEMCPY */
        do {
          slide[w++] = slide[d++];
          Tracevv((stderr, "%c", slide[w-1]));
        } while (--e);
      if (w == WSIZE)  //窗口满则输出解压数据
      {
        flush_output(w);
        w = 0;
      }
    } while (n);  //还有待拷贝的数据,则继续循环
  }
 }
 /* 恢复窗口指针和位缓存的全局量 */
 wp = w;                        /* wp 中保存当前窗口位置 */
 bb = b;                       /* bb 保存当前位缓存 */
 bk = k;     /* bk 保存当前位缓存中的有效数据位数 */
 /* done */
 return 0;
}
```

以下举例说明单个编码的解码流程：

比如下一个待读入的反序编码 0010 1000，假设 bl=4，那么先读取 4 位码长 1000 (值为 8) ,作为索引获取 huft 结构 t=tl[8]，这时因为有子表，所以联合类型 t->v.t 表示的是子表地址，e=t->e-16=4，表明要接着读取 4 位，于是读入 0010 (值为 2)，获取子表项 t=t->v.t+2，此时因为一个编码已读完，所以当前子表项已是叶节点，不会再有子表，此时有 t->e<=16，根据 t->e 的值有 3 种情况：

a. =16 时指示 t->v.n 为字符值 (literal)；

b. =15 时表示块结束标志；

c. <15 时表示是一个长度或者距离 (length or distance) 编码，它由基本值 (base bits) 和附加值 (extra bits) 构成 [参考 RFC1951]，t->v.n 为基本值，t->e 指示要读入的附加位 (extra bits) 长度，接着读入 e 位附加值，通过基本值+附加值得到 Huffman 编码 1000 0010 代表的值。

上述编码长度是访问两级 huft 表，如果继续编码更长的话还要访问更多的子表。

3 结语

Linux 内核的解压算法和操作系统运行机制不直接相关，因此往往为人所忽视。市面上的书籍一般对内核某些模块的运行机制描述详细，但是对系统的引导特别是解压却是一笔带过，没有深入到代码层进行分析。但是要全面地理解内核，引导和解压过程的解读显然是无法回避的。

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）