DOI:10.16184/j.cnki.comprg.2011.13.001

Linux 运行前探秘之二
——内核解压缩（一）

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：deflate；lz77；HSF 编码

1 概述

Linux 内核是采用 deflate 压缩规范进行压缩的，这是一种无专利限制的非常好的通用压缩算法，它采用 lz77+Huffman 编码的组合。对 Linux 内核解压的流程和 defalte 压缩规范做了介绍，并对相应的数据结构以及实现解压的外围算法和代码作必要的分析。

2 内核解压缩流程

如图 1 所示。

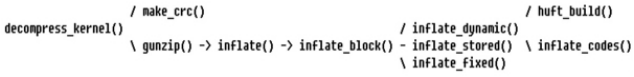

图 1 函数调用图

boot/compressed/head.S 调用 boot/compressed/misc.c 中的 decompress_kernel () 函数。

对于小内核 [zImage] ,decompress_kernel () 调用 setup_normal_output_buffer () 设置解压缩输出的目标起始位置 output_data=0x100000;对于大内核 [bzImage] ,decompressed_kernel () 解压的目标代码分两块存放，所以调用 setup_output_buffer_if_we_run_high ( ) 设置两个目标地址 low_buffer_start =0x2000 和 high_buffer_start =max (&end + 0x3000,0x100000 + low_buffer_size)。低地址部分涵盖 0x2000~0x90000，大小是 low_buffer_size；高地址是取 &end+0x3000 和 0x100000 + low_buffer_size 中的较大值，这样既能保证离自解压代码结束地址 end 后留有 0x3000 的堆空间 (HEAP_SIZE)，又保证高解压数据部分距离 0x100000 有不小于 low_buffer_size 的距离，这样就能使合并已解压的内核的时候不会被 low_buffer 的数据覆盖掉。此外，因为在现阶段内核尚未正式运行，因此解压过程中的内存分配是使用 compressed/misc.c 中的 malloc () 实现的，malloc () 维护着堆的当前指针 free_mem_ptr，free_mem_ptr 的最初位置就是 &end，结束位置 free_mem_end_ptr 是两种情况之一，小内核是 0x90000，大内核是 &end+0x3000。malloc (size) 分配 size 字节的内存，成功则 free_mem_ptr 后移 size 字节，如果发现已经到了堆末尾 (free_mem_ptr>=free_mem_end_ptr 时) 就报错。堆内存的回收函数 free () 被定义为空操作。因为每次解压完一个压缩数据块 block 后 ,inflate () 函数都会调用 gzip_release () 还原 free_mem_ptr 的值为初始值 (&end) 的，而每次解压一个 block 前会先调用 gzip_mark () 备份这个初始指针。

decompress_kernel () 调用 linux/lib/inflate.c 中定义的 gunzip ()，gunzip () 再调用 inflate () 进行解压。压缩数据 block (块) 为单位进行解码，inflate () 循环调用 inflate_block () ,每次解压一个块。inflate_block () 先读取块头的几位，第一位是结束块标志 (最后一个块)，这个值会返回给调用者 inflate () ，以决定是否结束循环;紧接着的两位是块类型，块被分为 3 种类型,dynamic、stored 和 fixed，分别表示为 10、00、01，inflate_block () 根据不同的值分别调用 inflate_dynamic ()、inflate_stored ()、inflate_fixed () 进行解码。stored 块是没经过压缩的，它的数据会原封不动地拷贝到输出端，另两种类型的块的解压过程都会经过两个阶段：调用 huft_build () 构造 huffman 解码表和调用 inflate_codes () 进行解码，这也是下面代码分析中要着重讨论的。

由于内核是压缩后与 boot/compressed/head.S 和 boot/compressed/misc.c 的目标代码连接的，因此 boot/compressed/misc.c 中的 decompress_kernl () 不可能调用到 linux/lib/inflate.c 中的 gunzip () 及其他函数，那这个函数怎么调用的呢？

在 misc.c 开始部分有 #include " ../../../../lib/inflate.c"，从而在预编译阶段,就把 linux/lib/inflate.c 中的函数代码都包含进 misc.c 中了，最后都编译在 misc.o 中，因此解压阶段调用的是自己的 gunzip ()。

3 deflate 压缩规范

Linux 内核解压缩的核心是 inflate ()，inflate 是针对用 deflate 压缩规范的压缩数据进行解压的算法。因此，要了解 Linux 内核的解压过程，有必要了解 deflate 规范。deflate 采用 lz77 压缩算法加上 Huffman-Shannon-Fano [HSF] 编码的组合。经典的 lz77 滑动窗口数据压缩算法使用一个 32KB 的滑动窗口 window，对于待压缩数据的下一个输入串，寻找在滑动窗口中

重复匹配的最长字串的起始位置和长度,使用长度 length 和距离 distance 表示,普通字符 literal 的值（包括不可打印字符）为 0~255，256 用来表示 block 结束标记，257~286 表示长度码(配合使用附加位)，0~30 表示距离码（配合附加位)，在此基础上对字符/长度和距离使用 HSF 编码。HSF 编码是 Huffman 编码（即最优可变长度前缀码）的变种，它除了 Huffman 编码的特点外，码字是按二进制数值递增的。

inflate 算法构建类似树的多级解码表，每个子表使用编码中的连续几位形成的二进制为索引。首先读入初始长度（比如最小码长）的位串,作为一级表的索引，根据编码搜寻到的表项或者是叶子节点（可以直接得出编码代表的值）,或者指向下一级子表，并指示接着要读入的编码位数（作为下一级表的索引)，最终达到解码的目的，使字符/长度位串（0~286）和 Huffman 码一一对应，根据 Huffman 码作为索引查询 Huffman 多级解码表最终可以确定原始值。inflate 算法就是在 lz77 编码基础上对 literal/length 和 distance 再用 Huffman 编码。因此解码的过程必须是简而言之这样的过程：读入基本长度的位串（比如 Huffman 编码的最小长度)，以其值为索引检索 literal/length 的 Huffman 表，得到 1 个 huft 结构指针 t，比较 t->e 的值，有如下几种情况：

（1） =15 表示本块 block（inflate 压缩数据由一个个数据块 block 组成）结束;

（2） =16 表示是 0~256 的字符值（包括不可打印字符)，直接输出到 Window 缓冲区;

（3） >16 表示当前表示中间表，以 t->e-16 的值为索引查询 t->v.t 指向的下一级 huffman 表;

（4）<16 表示这个编码是代表最终值（叶子节点）的，以 t->v.n 码得到长度的基本值，再读入 t->e 个附加位，两个值相加得到最终的 lz77 编码中的长度值。

distance 的得到过程与上同。两个值 length 和 distance 都解出来后，把 window [ptr-distance] 开始的 length 个字节拷贝到 window [ptr] 处，并更新 ptr 指针（后移 length 个位置)。

注：这里的 ptr 指缓冲区（或称为滑动窗口）的当前位置。每次达到最大值（Window 填满）后输出被解压的数据到目的位置 output_data，然后 Window 指针清 0 [缓冲数据仍保留直到被覆盖]。另外这里的 huffman 码使用的倒序编码,即最小值的位 LSB (least signified bit) 在最左边，最大值的位 MSB (most signified bit)，详见代码分析。

比如对于最小长度是 7 位的 9 位编码 1010110 01 来说，实际编码是这样的 10 0110101,先读入 7 位 0110101 为索引访问第一级 huffman 表，得 huft 结构指针 t，t->e 应该=16+2，表示还要读入 2 位索引，于是再读入 10，以它的值 2 为索引检索 t->v.t 指向的二级表（二级表一般有很多张，1 个一级表项就可以指向一个二级表)，得到新的 t，这时 t->e 因该是小于 16 的，指示 number of extra bits（附加位数)，t->v.n 指示基本值。

4 数据完整性校验

压缩解压过程必然伴随着数据完整性的验证，gunzip 中使用了 32 位循环冗余校验外加数据长度的检查。在压缩数据的过程中，压缩程序会不断地计算 crc 值，直到压缩过程结束，然后在最后一个压缩数据块之后写入 32 位长度的 crc 校验值和 32 位的数据原始长度值。在解压时，解压程序也会计算 crc 值，当解完最后一个数据块后 gunzup 会读入最后一个压缩数据块之后的原始 crc 值和原始长度值，把解压生成的 crc 值和解压的数据长度分别与之对比，都一致的话就表明解压成功了。以下是 32 位 CRC 校验使用的多项式：

$$2^0+2^1+2^2+2^4+2^5+2^7+2^8+2^{10}+2^{11}+2^{12}+2^{16}+2^{22}+2^{23}+2^{26}+2^{32}$$

5 HSF 编码

5.1 概述

deflate 压缩规范使用的是 Huffman 编码的一个变种，全称是 Huffman-Shannon-Fano 编码，它除了具有 Huffman 编码的优点外，还有个特点就是随着码长由小到大，编码的二进制数值也是从小到大递增的，同一码长下数值呈连续分布。这个特点使得只要确定了编码的码长，就可以算出编码。

5.2 生成 HSF 编码的例程

----------代码 5.1

```
code=0;
bl_count[0]=0;
for(bits=1;bits<=MAX BITS;bits++){
code=(code+bl_count[bits-1])<<1;
next_code[bits]=code;
}
```

----------代码 5.2

```
for(n=0;n<=max_code;n++){
len=tree[n].Len;
if(len!  =0){
tree[n].Code=next code[len];
next_code[len]++;
}
}
```

这是来自 RFC1951 文档的一段例程，演示了 deflate 中使用的 Huffman 编码的生成方法。学过数据结构的都知道,一般的 Huffman 编码是变长前缀码，是通过构造一棵 Huffman 树生成的。这里的 HSF 编码增加了两个附加规则：

所有给定长度的编码都是按字典序连续，和它们所代表的字符顺序相同。短编码在长编码前面。

HSF 的编码方法码长和码值是顺次增加的，最小最短的码排在最前，0 作为第一个码长的编码，同样长度的码数值连续递增，等该长度的编码数达到预定值后，先递增编码（加 1)，

然后移动一位（方向取决于编码顺序），作为下个编码长度的首个编码。按照这种方法就能生成变长前缀码。

再来看上述代码，代码 5.1 根据 bl_count [] 数组中预先确定的各码长编码数把各个码长的最小编码保存在 next_code [bits] 数组中（bits 代表码长）。代码 5.2 根据 next_code [] 数组生成广义字符 0~max_code 的编码。从代码可知，编码总是从 0 开始。

比如码长分别为 3、4、5，其编码数 bl_count [3] =3, bl_count [4] =2,bl_count [5] =4。那么 next_code [3] =000, next_code [4] =0110, next_code [5] =10100，顺次下来的编码清单是这样的：

```
000
001
010          <-到达 3 位码长的代码数最大值
[011] 0110     <-下一个码左移一位(这样就永远不会有重复前缀,符合前缀码的条件)
0111          <-继续递增
10000         <-再次左移为 5 位码长
10001
10010
10011
```

最后生成的编码依次是:000、001、010、0110、0111、10000、10001、10010、10011。

5.3 HSF 解码表

5.3.1 HSF 解码表结构

和传统 Huffman 编码算法使用的二叉树不同，HSF 解码使用的是多级表结构，每级表使用 HSF 编码中的连续几位作为索引，找到表项，这是一个 huft 结构，如果已经读入了编码的所有码长，那么这个表项指示的将是解码的值，否则该表项指示下一级子表以及要读入的后续码长（作为子表的索引），直至读完一个编码的所有码长。表结构如图 2 所示。

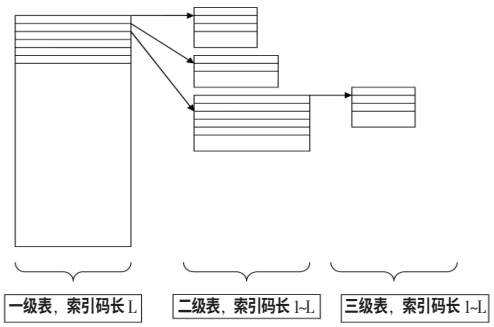

图 2 huft 多级解码表

L 是解压程序设置的解码表码长，在构建解码表时如果超出实际码长的范围会被调整为最小或最大码长，当 L=最大码长时，就只存在一级表了。

5.3.2 表项 huft 结构定义

多级解码表的每一个表项都是一个 huft 结构。

```
struct huft {
  uch e;    /* number of extra bits or operation */
  uch b;    /* number of bits in this code or subcode */
  union {
    ush n;              /* literal, length base, or distance base */
    struct huft *t;     /* pointer to next level of table */
  } v;
};
```

e 是一个多用途字段：

（1）>16 表示当前表项有子表，接着读入（e-16）位作为索引查询 v.t 指向的下一级表。

（2）=16 表示已达叶节点，码值就是 v.n，直接输出到 Window 缓冲区。

（3）=15 表示 block 结束。

（4）<15 表示已达叶节点，但是码值不是简单值。需要通过 v.n 得到基本值，再读入 e 个附加位，两个值相加得到最终的解码值。

b 是本表项涵盖的码长，也是解码过程中阶段性需要丢弃的位数（解完一部分就丢掉）。v 是一个 union 类型，当表项是非叶结点时，v.t 指向下一级子表；否则，v.n 指示了码值（或者基本值）。

5.3.3 解码表的内存分配与回收

解压阶段创建 huffman 解码表时使用 malloc () 分配内存，malloc () 维护一个堆指针 free_mem_ptr，指向下次分配内存开始的地址。堆位于内核上方的内存空间。

```
extern int end;
static long free_mem_ptr = (long)&end;
```

堆指针 free_mem_ptr 初始指向 end，end 是内核编译连接时连接程序生成的符号，指向自解压内核的结束地址+1。

内存回收函数 free () 是空操作，实际的内存回收通过恢复堆指针实现的，这发生于每次解码完一个块之后。

6 代码分析

（1）解压缩主函数 decompress_kernel ()

```
int de compr ess_kernel (struct moveparams *mv, void *rmode)
{
//rmode 是 boot/setup.S 传过来的参数,一般等于 0x90000,作
//为 low_buffer_end
      real_mode = rmode;
…
/* 设置输出缓冲区和堆结束地址。堆指针 free_mem_ptr 初始值是 &end,即自解压内核的结束地址,小于 0x100000,说明是小内核(被加载到内存地址 0x1000 处了),此时调用 setup_normal_output_buffer(); 否则是大内核(加载到内存地址 0x100000 处),调用 setup_output_buffer_if_we_run_high() */
      if (free_mem_ptr < 0 x100000) setup_normal_out-
```

```
put_buffer();
        else setup_output_buffer_if_we_run_high(mv);
        makecrc();//构建 crc32-tab
        puts("Uncompressing Linux... ");
        gunzip(); //调用 gunzip()解压缩
        puts("Ok, booting the kernel.\n");
/* high_loaded 是在 setup_output_buffer_if_we_run_high ()中设置为 1 的。high_loaded=1 时,表明是 big_kernel,解压缩的代码分两块 low_buffer 和 high_buffer 存放的,需要合并。这里调用 close_output_buffer_if_we_run_high () 设置 low_buffer 和 high_buffer 中的数据字节数 */
        if (high_loaded) close_output_buffer_if_we_run_high (mv);
        return high_loaded;
}
```

（2）解压缩数据输出缓冲区的设置

1）小内核的解压数据输出的目标地址指针 output_data 被设置为内存 1MB 处 (0x100000)

```
void setup_normal_output_buffer(void)
{
        output_data = (char *)0x100000; /* Points to 1M */
//小内核直接解压到 0x100000
/* 小内核的堆结束地址是 0x90000(由 boot/setup.S 传给 boot/compressed/head.S 的 esi 值决定的) */
        free_mem_end_ptr = (long)real_mode;
}
```

2）大内核的输出缓冲区

如果是大内核,解压数据有两个地方存放，低部缓冲区 low_buffer 和高部缓冲区 high_buffer。对于大内核，decompress_kernel () 调用 setup_output_buffer_if_we_run_high () 函数设置 moveparams 结构参数，其中包括了高、低两块缓冲区的起始地址和大小。

```
struct moveparams {
        uch *low_buffer_start;  //低缓冲区地址
        int lcount;             //低缓冲区中的数据字节数
        uch *high_buffer_start; //高缓冲区地址
        int hcount;             //高缓冲区中的数据字节数
};
```

这个结构是 head.S 中调用 decompress_kernel () 时压入堆栈中的，函数返回后，根据函数的返回值 (eax=high_loaded)，不为 0 的话转去执行合并程序。合并程序根据这个堆栈中的缓冲区结构参数将两个缓冲区中的解压数据合并成完整的内核，并放到内存地址从 0x100000 开始的地方。以下是 setup_output_buffer_if_we_run_high () 函数分析：

```
void setup_output_buffer_if_we_run_high(struct moveparams *mv)
{
/* 开始时先设置高缓冲区地址 high_buffer_start 起始于 &end+0x3000(自解压内核结束地址+0x3000),留出的 0x3000 字节是内存堆的大小 */
        high_buffer_start = (uch *)(((ulg)&end) + HEAP_SIZE);
//低缓冲区地址地址开始于 0x2000,这也是当前输出解压数据
//的地址(output_data)
        mv ->low_buffer_start = output_data = (char *)LOW_BUFFER_START;
/* 低缓冲区结束于 0x90000 和 real_mode 的较小值 (一般是 0x90000) */
                low_buffer_end = ((unsigned int)real_mode > LOW_BUFFER_MAX?
 LOW_BUFFER_MAX : (unsigned int)real_mode) & ~0xfff;
        low_buffer_size = low_buffer_end - LOW_BUFFER_START; //低缓冲区大小
/* 置 high_loaded 标志,表明解压数据是分两块存放的,解压完毕后,合并程序会读此标志以决定是否走合并流程。*/
       high_loaded = 1;
        free_mem_end_ptr = (long)high_buffer_start; //堆结束地址为 high_buffer_start
/* 当高缓冲区地址距离 0x100000 小于低缓冲区的大小，调整高缓冲区的地址使其距离 0x100000 正好为 low_buffer_size 个字节,这样合并时高缓冲区便无需移动,高缓冲区大小也无需设置,这里设 mv->hcount 为 0 */
        if ( (0x100000 + low_buffer_size) > ((ulg)high_buffer_start)) {
        high_buffer_start = (uch *)(0x100000 + low_buffer_size);
        mv ->hcount = 0; /* say: we need not to move high_buffer */
        }
/* 否则合并时需要移动高缓冲区，先把高缓冲区的字节数初始化为-1,等解压结束后,会填入正确的大小 */
        else mv->hcount = -1;
        mv->high_buffer_start = high_buffer_start;
}
```

3）解压后设置两个解压缓冲区的大小

decompress_kernel () 调用 gunzip () 解压内核成功后，如果是大内核，还要设置解压数据所在的两个缓冲区 (low_buffer 和high_buffer) 的大小，这是由下面的函数实现的：

```
void close_output_buffer_if_we_run_high(struct moveparams *mv) //参数是 moveparams 结构指针
{
/* bytes_out 是全局变量,保存了解压输出的全部数据大小。当 bytes_out >low_buffer_size 时低缓冲区大小设为 low_buffer_size*/
        if (bytes_out > low_buffer_size) {
                mv->lcount = low_buffer_size;
// mv->hcount!=0,表明合并过程中高缓冲区的数据也要移动,
//填入高缓冲区中的数据
                if (mv->hcount)
                        mv->hcount = bytes_out - low_buffer_size;
//高区数据量=总数据量-低区数据量
        } else {
//否则低缓冲区已容纳了全部解压内核数据,大小就是实际的数
```

```
//据大小,而高缓冲区中无数据
            mv->lcount = bytes_out;
            mv->hcount = 0;
        }
}
```

(3) 读入压缩内核数据

1) 获取 1 字节 get_byte ()

内核解压缩时是用 get_byte () 读入字节数据。这是一个宏定义，如下：

```
extern char input_data[];
extern int input_len;
#define get_byte()  (inptr < insize ? inbuf[inptr++] : fill_inbuf())
static int fill_inbuf(void)
{
        if (insize ! = 0) {
            error("ran out of input data\n");
        }
        inbuf = input_data;
        insize = input_len;
        inptr = 1;
        return inbuf[0];
}
```

fill_inbuf () 的功能是初始化数据指针 inbuf，指向 input_data，大小是 input_len。这里的 input_data 和 input_len 都是连接程序 ID 生成的符号，input_data 指示压缩内核所在的内存地址，input_len 是压缩内核的长度。首次运行 get_byte () 时,inptr=insize=0，所以调用 fill_inbuf () 设置inbuf 指针并返回压缩数据的第一个字节 inbuf [0] ,以后则返回后续字节 inbuf [inptr++]。

以下是 compressed/Makefile 中连接脚本，可以看一下 input_data 和 input_len 是如何产生的：

```
-----boot/compressed/Makefile-----
piggy.o:    $(SYSTEM)
        tmppiggy=_tmp_$$$$piggy; \
        rm -f $$tmppiggy $$tmppiggy.gz $$tmppiggy.lnk; \
        $(OBJCOPY) $(SYSTEM) $$tmppiggy; \
        gzip -f -9 < $$tmppiggy > $$tmppiggy.gz; \
        echo "SECTIONS  { .data :  { input_len = .; LONG(input_data_end - input_data) input_data = .; *(.data) input_data_end = .; }}" > $$tmppiggy.lnk; \
        $(LD) -r -o piggy.o -b binary $$tmppiggy.gz -b elf32-i386 -T $$tmppiggy.lnk; \
        rm -f $$tmppiggy $$tmppiggy.gz $$tmppiggy.lnk
```

内核目标代码 system 首先被去掉 elf 文件头存为$$tmppiggy，然后被压缩为$$tmppiggy.gz，使用连接脚本$$tmppiggy.lnk 连接为 elf 格式文件 piggy.o，该文件只包含一个数据段，就是上述被压缩的内核$$tmppiggy.gz,并引出符号 input_data 和 input_len 分别表示数据部分的地址和大小，ID 程序把 boot/compressed/head.o、boot/compressed/和自解压内核连接完成后。

注：inbuf [0] 是压缩内核的第一个字节,inbuf [inptr++] 指向后续字节 (inptr = 1 起)。

2) 获取任意 n 位二进制 (n 范围是 1~32)

仅有读入 1 字节的宏 get_byte () 还是不够的，因为 HSF 编码是变长的，有的码长不足 1 字节，有的码长超过 1 字节，因此编码是跨字节边界的，需要有获取指定位长二进制的手段。

如何获取并操作跨字节边界的位串?通过定义 NEEDBITS 和 DUMPBITS 两个宏获取和丢弃指定数量的位，实现对位操作的扩展。

```
#define NEXTBYTE()  (uch)get_byte() //获取一字节
#define NEEDBITS(n) {while(k<(n)){b|=((ulg)NEXTBYTE())<<k;k+=8;}}
#define DUMPBITS(n) {b>>=(n);k-=(n);}
```

b 是解压程序中定义的 32 位变量，作为读入位串的缓存，k 指示其中的有效数据位数。NEEDBITS (n) 获取 b 中的 n 位，它首先判断 b 中的数据位数 k 是否不足 n，当不足 n 时，循环调用 get_byte () 读取字节，并调整 k=k+8，直至有效位数 k>=n 为止。此时 b 中的低 n 位 (0~n-1 位) 就是解压程序所需的 n 位数据。DUMPBITS (n) 宏则通过右移位丢弃 b 中的低 n 位，并调整 k=k-n (表示有效位数减少 n 位)。

通过以上的宏，虽然每次还是读取 8B，但是一次可以对跨字节边界的 n 位进行操作了 (n=1~32)。

(4) 输出解码数据

解压过程中，使用 32KB 的滑动窗口作为输出的缓存。这是在 boot/compressed/misc.c 中定义的：

```
#define WSIZE 0x8000            /* Window size must be at least 32k, */
static uch window[WSIZE];   /* Sliding window buffer */
```

inflate.c 中的解压函数在解码出一个字节数据后，就拷贝到滑动窗口中 (slide [w++] =?)，局部变量 w 保存窗口中的当前写入位置；而 slide 是定义为 Window 的宏。

```
#define slide window
```

当滑动窗口已满就要把其中的解压数据拷贝到输出缓冲区去，这是通过如下代码实现的：

```
//当滑动窗口数据已满,调用 flush_output()把数据拷贝到输出
//缓冲区,然后 w 调整为 0
if (w == WSIZE)
{
        flush_output(w);
        w = 0;
}
```

flush_window () 判断 high_loaded 的值，如果等于 1 则调用 flush_window_high ()，否则调用 flush_window_low ()。

```
static void flush_window(void)
{
        if (high_loaded) flush_window_high();
```

```
//编译为 bzImage 大内核时调用
      else flush_window_low(); //zImage 小内核时调用
}
```

内核在输出解压数据的时候对滑动窗口中的解码数据进行 crc 校验值的计算。以下是输出代码清单：

```
/* 小内核 zImage 解压数据的拷贝,out_data 初始指向 0x100000 */
static void flush_window_low(void)
{
    ulg c = crc;         /* temporary variable */
//上次计算的校验值(初始位 0)存在 c 中
    unsigned n;
     uch *in, *out, ch;
/* window 是滑动窗口,也就是解压数据临时存放的缓冲区,大小为 W_SIZE 字节,每当解压输出的数据达到 W_SIZE 字节,充满了整个缓冲区后就要传送到 &output_data[output_ptr]开始的区域。 */
    in = window;
    out = &output_data[output_ptr];
//output_ptr 指向目标内存的指针
    for (n = 0; n < outcnt; n++) {  //结合上次的校验值(相当于
//模 2 除法的部分余数)继续计算本次的 crc 校验值
        ch = *out++ = *in++;
        c = crc_32_tab[((int)c ^ ch) & 0xff] ^ (c >> 8);
    }
    crc = c;                    //校验值保存在 crc 中
    bytes_out += (ulg)outcnt;   //调整输出字节数
    output_ptr += (ulg)outcnt;  //output_ptr 增加
    outcnt = 0;
}
/* bzImage 大内核解压数据的拷贝。和小内核解压基本相同,就是拷贝输出的目标地址分为两个缓冲区，并且不使用 output_ptr 指针。output_data 初始指向 low_buffer_start(0x2000),当达到上界 low_buffer_end (0x90000) 时，调整指向 high_buffer_start */
static void flush_window_high(void)
{
    ulg c = crc;         /* temporary variable */
    unsigned n;
    uch *in,  ch;
    in = window;
    for (n = 0; n < outcnt; n++) {
      ch = *output_data++ = *in++;
//output_data 到达 low_buffer 尾部时,调整指向 high_buffer
      if   ((ulg)output_data == low_buffer_end) output_data = high_buffer_start;     c = crc_32_tab [((int)c ^ ch) & 0xff] ^ (c >> 8);
    }
    crc = c;
    bytes_out += (ulg)outcnt;
    outcnt = 0;
}
```

在解压阶段，程序维持着几个指针。首先是输入数据的指针 inptr，解码前通过 inbuf [inptr++] 读取数据并后移指针；其次是滑动窗口指针 w，解码后 slide [w++] 保存解码数据并后移指针 w；最后是输出缓冲区指针，小内核使用 output_ptr，通过 out = &output_data [output_ptr] ;得到当前输出的起始地址，输出后后移 output_ptr 指针，大内核不使用 output_ptr 而直接使用并调整 output_data 指针。

(5) makecrc ()

该函数生成 8 位二进制 1~256 的 32 位 crc 码。crc_32_tab [i] 中保存各字符的 32 位 crc 值。

```
static void
makecrc(void)
{
/* Not copyrighted 1990 Mark Adler  */
  unsigned long c;      /* crc shift register */
  unsigned long e;      /* polynomial exclusive-or pattern */
  int i;                /* counter for all possible eight bit values */
  int k;                /* byte being shifted into crc apparatus */
  /* terms of polynomial defining this crc (except x^32): */
  static const int p[] = {0,1,2,4,5,7,8,10,11,12,16,22,23,26};//p 是 crc 多项式去掉最高位后剩余非 0 项的指数数组
  /* Make exclusive-or pattern from polynomial */
/*e 是 crc32 多项式生成的异或模式,用于模 2 减(crc 多项式是除数)*/
  e = 0;
  for (i = 0; i < sizeof(p)/sizeof(int); i++)
//按照 p[]数组的指数序列转化成反序的二进制 e
    e |= 1L << (31 - p[i]);
  crc_32_tab[0] = 0; //0 的 crc32 校验还是 0
// 外循环生成 1~255 的 32 位 crc 校验值
  for (i = 1; i < 256; i++)
  {
    c = 0;  //c 是部分余数,开始是 0
/* 内循环计算 i 的 crc32 值(除数是 e 的模 2 除法的余数)。i|256 是增加的标志哨，当 i 的所有位都右移掉之后,256 来到第 0 位,k==1,循环结束 */
    for (k = i | 256; k != 1; k >>= 1)    {
/* 注意:这里的模 2 除法是反序的,即第 0 位是最高位 MSB,所以是右移而不是左移位。c&1 判断余数最高位是否 1,是则右移 1 位(丢弃的那位已经和除数最高位减掉了)再和 e(去掉最高位的除数,也是反序的)异或(模 2 减),这是除法商 1 的情况;否则只是右移(除法商 0 时),
得到的 c 是新的余数 */
      c = c & 1 ? (c >> 1) ^ e : c >> 1;
/* 上面是余数和除数的运算,被除数没参与。这里判断一下当前被除数 k 的最右边的位 k&1,如果是 1,那么有两种情况:如果上面已经减了,这里要重新加上(考虑 c&1==1,那么(k^c)&1 会是 0,不该减);如果上面没减(c&1==0),那么算上被除数 k 的话应该要商 1 的,这里要减一下。不管是模 2 加还是模 2 减,都是异或操作,所以 k&1==1 时,补一个异或操作 c^=e,这时 c 才是本
```

(下转第 17 页)

```
'组 w 中
    For i = 0 To n – 1 '顶点与自己的距离为 0
        w(i, i) = 0
    Next
    For i = 0 To n – 1  '初始化 d 数组
        d(i) = –1
    Next
    For i = 0 To n – 1  '根据 W 数组初始化 d 数组
        d(i) = w(begin_p, i)
        pre(i) = begin_p
    Next
     '算法主体
     Dim dis As Integer   '存储当前最短距离
     Dim p_num As Integer '存储当前最短距离的点
     bj(begin_p) = True

     '对所有的顶点都进行一次对其他顶点的关于源点的松弛
'操作,就可以创造出起始点到其他各个点最短边,并把源点到
'点 i 最短距离存储在 D 数组中

    For i = 0 To n – 1
        dis = –1
        p_num = –1
        For j = 0 To n – 1
'搜索得到距离 begin_p 点最短的路径长 dis 和点 p_num
            If bj(j) = False Then
                If (dis = –1) Or ((d(j) > 0) And (dis > d(j))) Then
                    dis = d(j)
                    p_num = j
                End If
            End If
        Next
        If (p_num > 0) Then
            bj(p_num) = True  '标记点
            For j = 0 To n – 1  '修改点到起始点的距离
                If (w(p_num, j) > 0) And (bj(j) = False) Then
                    If (d(j) = –1) Or ((d(j) > –1) And (d(p_num) + w
(p_num, j) < d(j))) Then
                        d(j) = d(p_num) + w(p_num, j)
                        pre(j) = p_num
                    End If
                End If
            Next
        Else
            Exit For
        End If
    Next
     '根据 Pre 数组输出指定起始点到终点的路径
     Dim str As String
     i = end_p
     str = CStr(i)
     While (i <> begin_p)
          i = pre(i)
         str = CStr(i) + "to" + str
     End While
     MsgBox(str)
End Sub
```

4 实验数据

以一张城镇地图为例，图中的节点为城镇，无向边代表两个城镇之间的连通关系，现有一张县城的城镇地图，图中的顶点为城镇，无向边代表两个城镇间的连通关系，边上的权为两城镇之间的行车时间（单位为小时），现在要求得到指定两城镇之间行车的最短时间，以及行车路线。该问题模型为一个由10个顶点组成的带权无向图G，如图1所示。

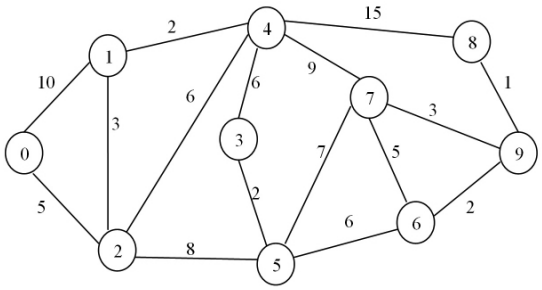

图1 带权无向图

设起始点为3，终点为9，最后输出路径为：3 to 5 to 6 to 9，路径长为：10。

（收稿日期：2011-03-21）

（上接第15页）

```
次操作的部分余数。 */
    if (k & 1)
      c ^= e;
   }
/* 除法操作完毕,c 是最后的余数,也是 i 的 crc32 校验值 */
   crc_32_tab[i] = c;
 }
 /* this is initialized here so this code could reside in ROM
*/
 crc = (ulg)0xffffffffL; /* shift register contents */
}
```

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）