LZW 数据无损压缩算法的 C++实现

王俊蛟

摘　要　本文简要介绍了一种非常巧妙且非常简洁易读的数据无损压缩算法(LZW)，并采用 C++/OOP 编程技术完整实现了该压缩算法，提供了多种函数调用接口。本文代码采用了哈希表检索、函数闭包、自释放动态数组等多种编程技术，代码执行速度很快，而且思想清晰明了，很容易读懂，不仅可以帮助读者迅速理解该压缩算法的原理，还可以作为读者在 C++编程方面的参考资料。

关键词　压缩，算法，C++，LZW

一、引言

目前，数据压缩技术已广泛应用于各种软件产品、技术规范、影像格式等，如我们每天都可能要使用 Winzip/ZipMagic/Arj 等压缩解压缩工具，浏览 gif/jpeg 压缩格式图片，或听 MP3 压缩音乐，甚至在您自己开发的产品中也包含了压缩技术，可见压缩技术应用是非常广泛的，特别是目前因特网如此拥挤的情况下，压缩技术更是必不可少；试想当我们下载未经压缩的 800 * 600 - 24b - BMP 位图影像，将不得不下载 1.37MB 数据，如果将其压缩为 jpg 格式，我们也许只需要下载 100KB 左右数据即可。看来，如果没有广泛采用压缩技术，眼前的因特网不知会阻塞到何种程度?

二、LZW 压缩算法

目前较成熟的数据压缩技术有许多种，主要分为无损压缩和有损压缩。其中较典型的无损压缩技术有 LZ77、LZ78 及其变种 LZO、LZH、LZW、ZLIB 等，有损压缩技术的代表则有 JPEG、MPEG、MP3 等。

本文将要介绍的是 LZW 数据无损压缩技术。LZW 压缩技术也是目前应用较广的一种压缩技术，著名的图像压缩格式 GIF98a 就是采用了 LZW 算法。该算法可以说是目前压缩算法中最简洁的一种，压缩算法描述仅 12 行即可，解压缩的算法描述也仅需要 15 行！这真令人吃惊。

LZW 压缩算法描述如下：

```
STRING = get input character
WHILE there are still input characters DO
    CHARACTER = get input character
    IF STRING + CHARACTER is in the string table then
        STRING = STRING + CHARACTER
    ELSE
        Output the code for STRING
        add STRING + CHARACTER to the string table
        STRING = CHARACTER
    END OF IF
END OF WHILE
output the code for STRING
```

LZW 解压缩算法描述如下：

```
read OLD_CODE
output OLD_CODE
WHILE there are still input character DO
  read NEW_CODE
  IF NEW_CODE is not in the translation table THEN
      STRING = get translation of OLD_CODE
      STRING = STRING + CHARACTER
  ELSE
      STRING = get translation of NEW_CODE
  END OF IF
  output STRING
  CHARACTER = first character in STRING
  add OLD_CODE + CHARACTER to the translation table
  OLD_CODE = NEW_CODE
END OF WHILE
```

根据以上算法，不难看出，LZW 压缩数据时，将由程序维护一个字节串表，该表用于存储各不相同的字节串，其中每个字节串都由一个代码标识，当输入流中再次出现前面已出现过的字节串时，便可用相应代码来替代这个字节串，从而实现数据压缩；当解压缩数据时，则执行相反的过程，将输入流中的代码翻译为实际字节串，从而实现解压缩。

LZW 压缩算法虽然寥寥十几行，但真要完整实现 LZW 算法，其代码量却会远远超出十几行，比想象中的要复杂一些，这种复杂性主要来自于数据流输入/输出(Stream I/O)、字节串表(String Table)的有效存储及提高字节串表检索速度方面的复杂性。本文将重点分析实现 LZW 算法中的几个难点，同时本文所附源代码涉及到 C++类(class)、类模板(Class Templates)及位元级操作(Bit Level)，因此，本文假设读者已具备了一定的 C++编程经验及初步的位元级编程能力。

三、LZW 压缩算法分析

LZW 压缩算法很简洁，但要完整实现其算法，尚需要解决以下方面的问题，才能使该算法真正有效。

1. 字节串表(String Table)的设计与实现

由 LZW 算法描述不难得知，需要一个字节串表(Table)来存储压缩/解压缩过程中的字节串，其容量为：

(1 < <lzw_bits)[其中 lzw_bits 代表压缩输出代码的位元数]

因为每输入一个字节，都需要在字节串表中查找其是否已存在；按 LZW 算法推算，也不难发现添加到表中的每个字节串长度是不固定的，至少是 2 个字节，也有可能是 3 个、4 个 ... 可见要直接存储这个变长字节串表是不易实现的，而且动态内存分配管理过程会明显降低程序执行效率，因此必须想办法进行简化。

经分析 LZW 算法，发现这些字节串有一个共性，即每个字节串都是由前一个代码所表示的字节串与刚输入的一个字节组成，为此，我们可以根据这一特点将不定长的字节串表转化为固定长度的元素表，即元素{prefix_code, character}组成的表，同时将算法描述稍作简化即可。

字节串表现已设计成功，当需要查找时，便可很简便地检索一个线性表即可。

2. 字节串/位串的输入与输出操作

当进行压缩/解压缩时，我们可以采用流式输入/输出操作，一次处理一个编码，这很符合 LZW 算法的特点。但压缩前输入的代码长度为 8 位，压缩后的数据将会有用更多位的代码(如 9 位 ... 16 位)进行输出；解压缩时的情况恰好相反。

由于输入/输出的过程不象整个字节输入/输出那样简单，必须提供位元级的操作函数才能较好分离出输入/输出模块，为了使算法本身清晰明了，本文代码中将这两个过程均封装成 C++类(TInputBuffer 和 TOutpuBuffer)，并提供了简洁的接口。这种封装在一定程度上降低了执行效率，但经实践发现，这对程序整体效率影响不算大，但在程序可读性方面得到了较大提高。

3. 如何有效提高字节串表的检索速度

当 lzw_bits 设置为 12，字节串表的长度为(1 < <12 = 4096)，当直接检索这个字节串表时，发现代码执行速度极慢，其时间复杂度为 O(n2)，简直无法忍受。为此，代码中不得不引入哈希表，经实践，证明确实能有效提高字节串表的检索效率及整体执行效率，但这需要付出(1 < <lzw_bits) * 2 字节的额外内存开销。

哈希表的容量同字节串表的容量，均为(1 < <lzw_bits)，表中每个元素是长度不固定的代码数组，这个数组用于存放哈希值相同的字节串代码，实际上对于一个较好的哈希函数产生的重复值极少。这样，检索字节串所经过的比较次数大幅减少，时间复杂度几乎接近于 O(n)，从而有效提高了代码的执行效率。

值得一提的是哈希函数的选择也极其重要，不合适的哈希函数不仅不能提高执行效率，反而会降低执行效率；本文所附源代码中写入了多种哈希函数，读者可以亲自动手试验一下，看看它如何影响程序的执行效率。

4. 如何得到字节串的首字节

由于在将变长字节串表转化为固定长度字节串表过程中，用编码替代了前面的字节串，这将引起不能直接访问整个字节串的首字节。为了解决这个问题，程序在递归输出过程中巧妙地在堆栈(stack)中记录了字节串的首字节，避免了再次重复递归搜索首字节，这也对程序执行效率作出了不小的贡献。

5. 为压缩数据流添加头部信息

由于压缩时有可能采用不同长度位元的编码，所以解压时也必须采用相同的位元长度进行流输入，否则将出错。为此，我们在压缩流的前面添加了 3 个字节的信息，以记载压缩算法类型、版本及位元数等。这样在解压缩时，便可根据输入流自动判别有关参数，避免手工去确认这些参数，从而使解压缩函数调用接口更为简单。

四、LZW 压缩算法的 C++代码

本文源代码由 Borland C++Builder5.0 进行编译调试，唯一依赖于编译器的地方就是压缩过程中的流式输入输出代码，如果您需要移植代码到其它平台或其它开发环境(如 VC++)，只需要修改流式输入输出等少量代码即可。当然如果将其编译成 DLL 动态链接库，则不必修改流式输入输出代码，只需要从 DLL 输出自己需要的接口即可。

另外，为了得到最高的执行效率，编译时应指示 C++编译器按“速度最大”方式去优化编译代码，如果没有编译器的优化帮忙，程序执行效率会打折扣的。

五、小结

LZW 压缩算法较简洁，实现起来相对较容易，较适合于图像文件等冗余较多的文件类型，但文件长度不能过长，一般 1M 内效果较佳。

LZW 压缩算法本身也具有一定的局限性，比如当输入流内容冗余较少或输入流尺寸太大，这将导致内部维护的字节串表(StringTable)很快充满，致使它很快就会丧失压缩能力，一旦字节串表被充满，继续压缩则会引起反作用，并逐渐抵消已经得到的压缩率，更严重时则会使文件尺寸反而增大；另外，由于 LZW 压缩率还依赖于位元数(lzw_bits)的设置，当输入为任意类型的数据流时，压缩率不一定随位元数增加而增加，也不一定随位元数的减少而减少，也就是说 LZW 压缩率与位元数的具体关系基本上是不确定的；当输入流内容相对确定时，为了得到较好的压缩效果，可通过设置位元数(lzw_bits)反复试验，才能得到最满意的效果。

当然，LZW 压缩算法的缺陷并不是完全无法避免的，有些 LZW 变种压缩算法(如 ARC)通过在适当时候清空字节串表，来避免字节串被充满，从而能够有效提高压缩率。

本文仅起抛砖引玉作用，由于本人水平所限，文中可能有考虑不周，甚至包含错误；另外，本文源代码并未提供错误校验方面的代码，如果读者感兴趣，可以自行提供校验代码和添加输入流中原始数据的 CRC32 或 ADLER32 检验值，同时，本人也非常希望能和大家能一起探讨数据压缩技术(王俊蛟 junjiao@126.com http://junjiao.yeah.net)。

VC++中状态栏的动态编程

兰 帆

摘 要 本文介绍了一种在 Visual C++ 6.0 中动态改变状态栏内的字符串并在状态栏中动态显示图标的方法。

关键词 Visual C++,状态栏,CStatusBar

一、引言

在应用程序的状态栏上显示一个不断变化的字符串，例如，当程序运行时在状态栏显示当前的系统时间，如再在状态栏上加上动画式图标将为您的程序增色不少。有一些文章也介绍过 VC 中状态栏的动态编程，可在具体实现的时候并不能让人满意。经过一定的研究，我这里提供一种动态改变状态栏的方法，可以很好地实现状态栏的动态改变，并在状态栏中加入动画图标。

二、实现原理

首先，在 CMainFrame 类中添加 InitStatusBar 函数用来对我们的状态栏进行初始化，利用 CMainFrame 类中的 CStatusBar 类成员变量 m_wndStatusBar 来实现对状态栏的控制。在 InitStatusBar 函数中还将初始化一个 CImageList 类变量 m_BarImage，用它在状态栏中显示图标。还要在 CMainFrame 类中添加 OnUpdateMyStatus 函数用以响应状态栏更新消息，还可在 CMainFrame 类中添加 OnTimer 函数响应 WM_TIMER 消息，并在 OnTimer 函数体内改变将要显示的字符串和图标。

三、具体实现

1. 用 Visual C++ 6.0 创建一个单文档工程 Test，确认在创建过程中选中了 Initial status bar 选项。在资源中插入一个新位图资源 IDB_BITMAP1，用十种不同的颜色在位图中画十个大小为 16×16 的实心圆。

2. 打开 MainFrm.h，添加如下变量定义：

```
// MainFrm.h : interface of the CMainFrame class
/////////////////////////////////////////////////////////////////////////////
class CMainFrame : public CFrameWnd
{
  . . . . . .
// Implementation
public:
  CString TimeTextOld; //存储当前状态栏显示时间的变量
  CString TimeText; // 存储当前时间的变量
  int m_ImageNo; //存储状态栏显示的图标序号的变量
  BOOL InitStatusBar(UINT *pIndicators, int nSize);
//初始化状态栏函数
  int m_MyPane; //状态栏中我们自己添加的窗格号
  virtual ~CMainFrame();
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif
protected:  // control bar embedded members
  CStatusBar  m_wndStatusBar;
  CToolBar    m_wndToolBar;
// Generated message map functions
protected:
  CImageList m_BarImage;
  //{{AFX_MSG(CMainFrame)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  afx_msg void OnUpdateMyStatus(CCmdUI * pCmdUI);
//更新 MyPane 窗格的函数
  afx_msg void OnTimer(UINT nIDEvent); //WM_TIMER 响
```


参考文献

1.《数据结构--C++ 语言描述》西安交通大学出版社，1999 年，赵文静 著

2.《Data Structures and Algorithm Analysis》电子工业出版社，1998 年，Clifford A. Shaffer 编著《数据结构与算法分析》张铭、刘晓丹译

3.《资料压缩——原理与实务》台湾松岗电脑图书资料股份有限公司，1994 年，张真诚、蔡文辉编著

4.《Success With C++》电子工业出版社，1996 年，[美]Kris Jamsa, PH. D 编著《C++成功使用秘诀》张嵩、郝成江、张海、张虹翼等译

（收稿日期：2000 年 10 月 10 日）