[返回主页](http://www.cnblogs.com/sjhrun2001/)

[**sjhrun2001**](http://www.cnblogs.com/sjhrun2001/)

技术之美

* [博客园](http://www.cnblogs.com/)
* [首页](http://www.cnblogs.com/sjhrun2001/)
* [新闻](http://news.cnblogs.com/)
* [新随笔](https://i.cnblogs.com/EditPosts.aspx?opt=1)
* [联系](http://msg.cnblogs.com/send/%E4%B8%80%E6%B2%99)
* [管理](https://i.cnblogs.com/)
* [订阅](http://www.cnblogs.com/sjhrun2001/rss) [订阅](http://www.cnblogs.com/sjhrun2001/rss)

随笔- 48  文章- 10  评论- 10

[**ttf文件结构解析**](http://www.cnblogs.com/sjhrun2001/archive/2010/01/19/1651274.html)

TrueType字体通常包含在单个TrueType字体文件中，其文件后缀为.TTF。OpenType字体是以类似   于TrueType字体的格式编码的POSTSCRIPT字体。OPENTYPE字体使用.OTF文件后缀。OPENTYPE还允许把多个OPENTYPE字体组合在一个文件中以利于数据共享。这些字体被称为TrueType字体集（TrueType   collection），其文件后缀为.TTC。  
      TrueType字体用machintosh的轮廓字体资源的格式编码，有一个唯一的标记名"sfnt"。windows没有macintosh的位图字体资源格式，字体目录   包含了字体格式的版本号和几个表，每个表都有一个tableentry结构项，tableentry结构包含了资源标记、校验和、偏移量和每个表的大小。下面是TrueType字体目录的c语言定义：  
typedef   sturct  
{  
char   tag[4];  
ULONG   checkSum;  
ULONG   offset;  
ULONG   length;  
}TableEntry;  
  
typedef   struct  
{  
Fixed   sfntversion;   //0x10000   for   version   1.0  
USHORT   numTables;  
USHORT   searchRange;  
USHORT   entrySelector;  
USHORT   rangeShift;  
TableEntry   entries[1];//variable   number   of   TableEntry  
}TableDirectory;  
  
        TrueType   字体中的所有数据都使用big-endian编码，最高位字节在最前面（因为TrueType字体最初是由apple公司定义的，而apple公司的os运行在motorola的cpu上）。如果一人TrueType字体以00   01   00   00   ,00   17开头，我们就可以知道它的格式是轮廓字体资源（"sfnt"）版本1.0的格式，有23个表。  
  
        TableDirectory结构的最后一个字段是可变长度的tableentry结构的数组，安体中的每个表对应其中一项。TrueType字体中的每个表都保存了不同的逻辑信息-----如图元中数据、字符到图元的映射、字距调整信息等等。有表是必须的，有些是可选的。下表列出了TrueType字体中常见的表。  
    
head           字体头                                           字体的全局信息  
cmap           字符代码到图元的映射               把字符代码映射为图元索引  
glyf           图元数据                                       图元轮廓定义以及网格调整指令  
maxp           最大需求表                                   字体中所需内存分配情况的汇总数据  
mmtx           水平规格                                       图元水平规格  
loca           位置表索引                                   把元索引转换为图元的位置  
name           命名表                                           版权说明、字体名、字体族名、风格名等等  
hmtx           水平布局                                       字体水平布局星系：上高、下高、行间距、最大前进宽度、最小左支撑、最小右支撑  
kerm           字距调整表                                   字距调整对的数组  
post           PostScript信息                           所有图元的PostScript   FontInfo目录项和PostScript名  
PCLT           PCL   5数据                                     HP   PCL   5Printer   Language   的字体信息：字体数、宽度、x高度、风格、记号集等等  
OS/2           OS/2和Windows特有的规格         TrueType字体所需的规格集  
        在TableDirectory结构中，所有的TableEntry结构都必须根据它们的标记名排序。比如，cmap必须出现在head前，而head必须在glyf前。但是实际的表可以出现在TrueType字体文件中的任意位置。  
        Win32API   提供了一个应用程序可用于查询原始TrueType字体信息的函数：  
DWORD   GetFontData(HDC   hDC,DWORD   dwTable   ,DWORD   dwOffset,   LPVOID   lpbBuffer   ,DWORD   cbData);  
        GetFontData函数可以用于查询设备上下文中当前逻辑字体所对应的TrueType字体，因此传递的不是逻辑字体句柄，而是设备上下文句柄。你可以查询整个TrueType文件基是文件中的一个表。要查询整个文件的话dwTable参数应该为0；否则，应该传递要查询的表的四字符标记的DWORD格式。参数dwOffset是要查询的表中的起始偏移，要查询整个表的话应该为0；参数;pvBuffer是缓冲区的地址，cbData是缓冲区的大小。如果最后个参数为NULL和0，GetFontData函数返回字体文件或表的大小；就会把到的数据拷贝到应用程序所提供的缓冲区中。  
下面的例和查询整个TrueType字体的原始数据：  
  TableDirctory   \*   GetTrueTypeFont   (HDC   hDC   ,DWORD   &nFontSize)  
{  
//query   font   size  
nFontSize=GetFontData(hDC,0,0,NULL,0);  
  
TableDirectory   \*   pFont   =(TableDirectory   \*)new   BYTE(nFontSize);  
if   (pFont==NULL)  
return     NULL;  
GetFontData(hDC,0,0,pFont,nFontSize);  
  
return   pFont;  
}  
        GetFontData使得应用程序能够在自己的文档中内嵌TrueType字体，以确保这些文档能在没有相应字体的其他机器上显示。它的做法是允许应用程序查询字体数据，然后写入到文档中作为文档的一部分，在文档被打于时再安装该字体以确保文档能以创建时同样的方式显示。比如，Windows   NT/2000的假脱机程序在打印到远端服务器时会在假脱机文件中内嵌入TrueType字体以保证文档能在另一台机器上正确地打印。  
一旦接受到TrueType字体的原始数据，它的头中的TableDirectory结构很容易分析。需要检查的只有版本号和表的数目，然后就可以检查单个的表。我们来看一些重要的和有趣的表。  
1.字体头  
    字体头表（head表）中包含了TrueType字体的全局信息。下面是字体头表的结构。  
typedef   sturct  
{  
Fixed   Table;//x00010000   ro   version   1.0  
Fixed   fontRevision;//Set   by   font   manufacturer.  
ULONG   checkSumAdjustment;  
ULONG   magicNumer;   //Set   to   0x5f0f3cf5  
USHORT   flags;  
USHORT   unitsPerEm;   //Valid   range   is   from   16   to   16384  
longDT   created;   //International   date   (8-byte   field).  
longDT   modified;   //International   date   (8-byte   field).  
FWord   xMin;   //For   all   glyph   bounding   boxes.  
FWord   yMin;   //For   all   glyph   bounding   boxes.  
FWord   xMax;   //For   all   glyph   bounding   boxes.  
FWord   xMax;   //For   all   glyph   bounding   boxes.  
USHORT   macStyle;  
USHORT   lowestRecPPEM;   //Smallest   readable   size   in   pixels.  
SHORT   fontDirctionHint;  
SHORT   indexToLocFormat;   //0   for   short   offsets   ,1   for   long.  
SHORT   glyphDataFormat;     //0   for   current   format.  
}Table\_head;  
  
        字体的历史记录在三个字段中：字全版本号、字体最初创建时间和字体最后修改时间。有8   个字节用于记录时间戳，记录的是从1904年1月1日午夜12：00开始的秒数，因此我们不用担心y2k问题，或是什么y2m问题。  
        字体设计时是针对一个参考网格设计的，该网格被称为em-square，字体中的图元用网格中的坐标表示。因此em-squrare的大小决定胃该字体的图元被缩放的方式，同时也反映胃该字体的质量。字体头中保存了每个em-square的格数和能   包含所有图元的边界框。Em-square的有效值是从16到16384，常见的值是2048、4096和8192。比如，Windings字体的em-square的格数是2048，图元的边界框是[0,-432,2783,1841]。  
        字体头表中的其他信息包括最小可读像素大小、字体方向、在位置表中图元索引的格式和图元数据格式等等。  
最大需求表  
        TrueType字体是一种非常灵活的数据结构，它可以包含可变数目的图元，每个图元可以有不同数目的控制点，甚至还可以有数量可变的图元指令。最大需求表的目的是告知字体栅格器（rasterizer）对内存的需求，以便   在出来字体前分配合适大小的内存。因为性能对字体栅格器非常重要，像MFC的CAarray那样需要频繁进行数据拷贝操作的动态增长的数据结构不合要求。下面是maxp表的结构。  
typedef   struct  
{  
Fixed   Version;//0x00010000   for   version   1.0.  
USHORT   numGlypha;   //Number   of   glyphs   in   the   font   .  
USHORT   maxPoints;   //Max   points   in   noncomposite   glyph   .  
RSHORT   maxContours;   //Max   contours   in   noncomposite   glyph.  
USHORT   maxCompositePoints;//Max   points   in   a   composite   glyph.  
USHORT   maxCompositeContours;   //Max   contours   in   a   composite   glyph.  
USHORT   maxZones;//   1   if   not   use   the   twilight   zone   [Z0],  
                                //or   2   if   so   use   Z0;2   in   most   cases.  
USHORT   max   TwilightPoints   ;/   Maximum   points   used   in   Z0.  
USHORT   maxStorage;   //Number   of   storage   area   locations.  
USHORT   maxFunctionDefs;   //Number   of   FDEFs.  
USHORT   maxStackElements;   //Number   of   depth.  
USHORT   maxSizeOfInstructions;   //Max   byte   count   for   glyph   inst.  
USHORT   maxComponentElements;   //Max   number   top   components   refernced.  
USHORT   maxComponentDepth;       //Max   levels   of   recursion.  
}Table\_maxp;  
  
        numGlyphs字段保存了字体中图元的总数，这决定了到位置表的图元索引的数量，可以用于严正图元索引的有效性。TrueType字体中的每个图元都可以是合成图元或简单图元。简单图元可以有一条或多大体上轮廓中国，条用一些控制点定义。合成图元用几个其他图元的组合来定义。maxPoints\maxCountors\maxCompositePoints   maxCompositeContours这几个字段说明了图元定义的复杂度。  
        除了图元的定义，TrueType字体还使用了图元指令用于提示字体扫描器如何对控制点进行调整以得到更均衡更漂亮的光栅化后的图元。图元指令也可以出现在字体程序表（fpgm表）以及控制值程序表（“prep”）的全局字体层中。TrueType图元指令是一个伪计算机字节指令，该机类似于Java的虚拟机，这些指令可以用堆栈计算机执行。MaxStackElements   maxSizeOfInstructions两个字段同志堆栈计算机这些指令的复杂度。  
        以Windings字体为例，该字体有226个图元，图元最多有47条轮廓线，简单图元最多有268个点，合成图元最多有141个点，合成图元最多有14条轮廓线，最坏情况下需要492层堆栈，最长的指令有1119个字节。  
字符到图元索引的映射表（cmap表）定义了从不同代码页中的字符   代码到图元索引的映射关系，这是在TrueType字体中存取图元信息的关键。cmap表包含几个了表以支持不同的平台和不同的字符编码方案。  
  
下面是cmap表的结构。  
  
  typedef   struct  
{  
 USHORT   Platform;   //platform   ID  
 USHORT   EncodingID;   //encoding   ID  
 ULONG   TableOffset   ;//offset   to   encoding   table     
  
 typedef   struct   {  
 WCHAR   wcLow;  
 USHORT   cGlyphs;  
 }  
  
 typedef   struct     
 {  
 DWORD   cbThis;     //sizeof   (GLYPHSET)+sizeof(WCRANGE) \* (cRanges-1)  
 DWORD   flAccel;  
 DWORD   cGlyphsSupported;  
 DWORD   cRanges;  
 WCRANGE   ranges[1];   //ranges[cRanges]  
 }GLYPHSET;

}table\_cmap;  
  
DWORD   GetFontUnicodeRanges(HDC   hDC,LPGLYPHSET   lpgs);  
DWORD   GetGlyphIndices(HDC   hDC,LPCTSTR   lpstr,int   c   ,LPWORD   pgi,DWORD   fl);  
  
        通常一种字体只提供UNICODE字符集中的字符的一个子集。这些字符可以被分组为多个区域，cmap映射表中就是这么做的。GetFontUnicodeRanges函数在一个GLYPHSET结构中返回支持的图元的数量、支持的UNICODE区域的数量以及设备上下文中字体的这些区域的详细信息。GLYPHSET是一个可变长的结构   ，其大小取决于所支持的UNICODE区域的数量。因此，和Win32   API中支持可变长结构一样，       GetFontUnicodeRanges函数通常需要调用两   次。第一次调用时得到以NULL指针作为最后一莜参数，GDI会返回所需窨的大小。调用者然后分配所需的内存，再次调用以得到真正的数据。这两   种情况下，GetFontUnicodeRanges函数都会返回保存整个结构所需的数据大小。MSDN文档可能还是错误地描述成了如果第二个参数是NULL，GetFontUnicodeRanges函数返回指向GLYPHSET结构的指针。  
下面是用于查询上下文中当前字体GLYPHSET结构的一个简单函数。  
GLYPHSET   \*QueryUnicodeRanges(HDC   hDC)  
{  
//query   for   size  
DWORD   size=GetFontUnicodeRanges(hDC,NULL);  
  
if   (size==0)   return   NULL;  
GLYPHSET   \*pGlyphSet=(GLYPHSET   \*)new   BYTE(size);  
  
//get   real   data  
pGlyphSet->cbThis=size;  
size=GetFontUnicodeRanges(hDC,pGlyphSet);  
  
return   pGlyphSet;  
}  
  
  
        如果在一些Windows   TrueType字体上试着调用GetFontUnicodeRanges函数，你会发现这些字体通常支持1000个以上的图元，这些图元被分成几百个UNICODE区域。比如，“Times   New   Roman”有我143个图元，分布在145个区域中，和一个区域是0x20到0x7f，即可打印的7位ASCII代码区域。  
GetFontUnicodeRanges函数只使用了TrueType字体“cmap”表的一部分部分信息，即从UNICODE到图元索引的映射域。GetGlyphIndices函数则能真正使用这些映射关系把一个字符串转换为一个图元索引的数组。它接收一个设备上下文句柄、一个字符串指针、字符串长度、一个WORD数组的指针和一个标志。生成的图元索引将保存在WORD数组中。如果标志为GGI\_MASK\_NONEXISTING\_GLYPHS,找不到的字符的图元索引会被标注成0xFFFF。此函数得到的图元索引可以传给其他GDI函数,如ExtTextOut函数。  
  
2.位置索引  
  
        TrueType字体中最有用的信息是glyf表中的图元数据。有了图元索引，要找到相应的图元，需要表（loca表）索引以把图元索引转换为图元数据表内的偏移量。  
位置索引表中保存了n+1个图元数据表的索引，其中n是保存在最大需求表中的图元数量。最后一个额外   的偏移量并不指向一个新图元，而是指向最后一个图元的偏移量和当前图元的偏移量和当前图元的偏移量间的差值得到图元的长度。  
位置索引表中的每一个索引以无符号短整数对齐的，如果使用了短整数格式，索引表实际存储的是WORD偏移量，而不是BYTE偏移量。这合得短整数格式的位置索引表能   支持128KB大小的图元数据表。  
  
3.图元数据  
      图元数据（glyf表）是TrueType字体的核心信息，因此通常它是最大的表。因为的位置索引是一张单独的表，图元数据表就完全只是图元的序列而已，每个图元以图元头结构开始：  
typedef   struct     
{  
WORD   numberOfContours;   //contor   number,negative   if   composite  
FWord   xMin;       //Minimum   x   for   coordinate   data.  
FWord   yMin;       //Minimum   y   for   coordinate   data.  
FWord   xMax;       //Maximum   x   for   coordinate   data.  
FWord   yMax;       //Maximum   y   for   coordinate   data.  
}GlyphHeader;  
  
        对于简单图元，numberOfContours字段中保存的是当前图元的轮廓线的数量；对于合成图元，numberOfContours字段是一个负值。后者的轮廓线的总数必须基于组成该合成图元的所有图元的数据计算得到。GlyphHeader结构中后四个字段记录了图元的边界框。  
      对于简单图元，图元的描述紧跟在GlyphHeader结构之后。图元的描述由几部分信息组成：所有轮廓线结束点的索引、图元指令和一系列的控制点。每个控制点包括一个标志以x和y坐标。概念上而言，控制所需的信息和GDI函数PolyDraw函数所需的信息相同：一组标志和一组点的坐标。但TrueType字体中的控制点的编码要复杂得多。下面是图元描述信息的概述：  
USHORT   endPtsOfContours[n];   //n=number   of   contours  
USHORT   instructionlength;  
BYTE   instruction[i];       //i   =   instruction   length  
BYTE   flags[];                   //variable   size  
BYTE   xCoordinates[];     //variable   size  
BYTE   yCoordinates[];     //variable   size  
        图元可以包含一条或多条轮廓线。比如，字母"O"有两   条轮廓线，一条是内部的轮廓，另一条是外部的轮廓。对于每一条轮廓线，endPtsOfContours数组保存了其终点的索引，从该索引中可以计算出轮廓线中点的数量。比如，endPtsOfContours[0]是第一休轮廓线上点的数量，endPtsOfContours[1]-endPtsOfContours[0]是第二条轮廓线上点的数量。  
        终点数组后是图元指令通知度和图元指令数组。我们先跳过它们，先来讨论冬至点。图元的控制点保存在三个数组中：标志获得组、x坐标数组和y坐标数组。找到标志数组的起始点很简单，但是标志数组没有相应的长度字，也没有直接其他两个数组的方法，你必须先解码标志数组才能解释x和y坐标数组。  
        我们提到棕em-square被限制为最大为16384个网格，因此通常情况下需要各两个字节来表示x坐标和y坐标。为了节省空间，图元中保存的是相对坐标。第一个点的坐标是相对（0，0）记录的，所有随后的点记录者是和上一个点的坐标差值。有些差值可以用一个字节表示，有些差值为0，另外一些差值则无法用耽搁字节表示。标志数组保存了每个坐标的编码信息以及其他一些信息。下面是标志中各个位的含义的总结：  
  
typedef   enum  
{  
G\_ONCURVE       =   0x01,   //   on   curve   ,off   curve  
G\_REPEAT   =0x08,       //next   byte   is   flag   repeat   count     
G\_XMASK       =0x12,     
G\_XADDBYTE   =0x12,   //X   is   positive   byte  
G\_XSUBBYTE   =0x12,   //X   is   negative   byte     
G\_XSAME     =0x10,   //X   is   same  
G\_XADDINT   =0x00,   //X   is   signed   word     
  
G\_YMASK     =0x24,  
G\_YADDBYTE   =0x24,   //Y   is   positive   byte     
G\_YSUBBYTE   =0x04,   //Y   is   negative   byte  
G\_YSAME   =0x20   ,       //Y   is   same  
G\_YADDINT   =0x00,   //Y   is   signed   word  
};  
  
        在第8章中我们讨论了直线和曲线，我们提到了一段三阶Bezier曲线有四个控制点定义：位于曲线上（on-curve)的起始点、两个不在曲线上（off-curve）的控制点和一个曲线上的结束点。TureType字体中的图元轮廓是用二阶Bezier曲线定义的，有三个点：一个曲线上的点，一个曲线外的点和另一个曲线上的点。多个连续的不在曲线上的点是允许的，但不是用来定义三阶或更高阶的Bezier曲线，而是为了减少控制点的数目。比如，对于on-off-off-on模式的四个点，会加入一个隐含的点使之成为on-off-on-off-on,因此定义的是两段二阶Bezier曲线。  
        如果设置了G\_ONCURVE位，那么控制点在曲线上，否则不在曲线上。如果设置了G\_REPEAT,标志数组中的下一字节表示重复次数，当前标志应该重复指定的次数。因此，标志数组中实际使用了某种类型的行程编码。标志中的其他位用于描述相应   的x坐标和y坐标的编码方式，它们可以表示当前相寻坐标是否和上一个相同、正的单字节值、负的单字节值或有符号两字节值。  
        解码图元的描述是一个两次扫描的起始点。然后再遍历图元定义中的每一个点把它转换为更容易管理的格式。程序清单14-2列出了解码TrueType图元的函数，它是KTrueType类的一个方法。  
  
int   KTrueType::DecodeGlyph(int   index,   KCurve   &   curve,   XFORM   \*   xm)   const  
{  
const   GlyphHeader   \*   pHeader   =   GetGlyph(index);  
  
if   (   pHeader==NULL   )  
{  
// assert(false);  
return   0;  
}  
  
int nContour   =   (short)   reverse(pHeader->numberOfContours);  
  
if   (   nContour<0   )  
{  
return   DecodeCompositeGlyph(pHeader+1,   curve);   //   after   the   header  
}  
  
if   (   nContour==0   )  
return   0;  
  
curve.SetBound(reverse((WORD)pHeader->xMin),   reverse((WORD)pHeader->yMin),     
                      reverse((WORD)pHeader->xMax),   reverse((WORD)pHeader->yMax));  
  
const   USHORT   \*   pEndPoint   =   (const   USHORT   \*)   (pHeader+1);  
  
int   nPoints   =   reverse(pEndPoint[nContour-1])   +   1;     //   endpoint   of   last   contour   +   1  
int   nInst       =   reverse(pEndPoint[nContour]);       //   instructon   length   
  
curve.m\_glyphindex   =   index;  
curve.m\_glyphsize     =   (int)   GetGlyph(index+1)   -   (int)   GetGlyph(index);  
curve.m\_Ascender       =   m\_Ascender;  
curve.m\_Descender     =   m\_Descender;  
curve.m\_LineGap         =   m\_LineGap;  
  
GetMetrics(index,   curve.m\_advancewidth,   curve.m\_lsb);  
  
if   (   curve.m\_glyphsize==0   )  
return   0;  
  
curve.m\_instrsize     =   nInst;  
  
const   BYTE   \*   pFlag   =   (const   BYTE   \*)   &   pEndPoint[nContour]   +   2   +   nInst; //   first   byte   in   flag  
const   BYTE   \*   pX         =   pFlag;  
  
int   xlen   =   0;  
  
for   (int   i=0;   i<nPoints;   i++,   pX++)  
{  
int   unit   =   0;  
  
switch   (   pX[0]   &   G\_XMASK   )  
{  
case   G\_XADDBYTE:  
case   G\_XSUBBYTE:  
unit   =   1;  
break;  
  
case   G\_XADDINT:  
unit   =   2;  
}  
  
if   (   pX[0]   &   G\_REPEAT   )  
{  
xlen   +=   unit   \*   (pX[1]+1);     
  
i   +=   pX[1];  
pX   ++;  
}  
else  
xlen   +=   unit;  
}  
  
const   BYTE   \*   pY   =   pX   +   xlen;  
  
int   x   =   0;  
  
  
  
  
KTrueType类处理TrueType字体的装入和解码，随书光盘中有它的完整源代码。DecodeGlyph给出图元索引和可选的变换矩阵，处理的是单个图元的解码。参数curve是KCurve类，用于把TrueType图元定义保存为32位的点的赎罪以及一个标志数组，以梗用GDI进行显示。这些代码可以作为简单TrueType字体编辑器的基础。  
        代码中调用了GetGlyph方法，该方法用位置表索引找到该图元的GlyphHeader结构。从中得到图元的轮廓线数目。注意必须反转该值的字节序，因为TrueType字体用的是Big-Endian字节序。如果该值为负值，说明这是一个合成图元，应该转而调用DecodeCompositeGlyph方法。接下支的代码定位了endPtsOfContours数组，找出点的总数，然后跳过指令找到标志数组的起始位置。

       接下去需要长到的是x坐标数组的始位置和长度，这需要遍历标志数组一次。对于每一个控制点，它在x坐标数组中所占空间可能为0到2个字节，这取决于它的相对坐标是0、单个字节还是两个字节。  
        根据x坐标数组的地址和长度可以得到y坐标的地址。接下去的代码遍历所有的轮廓线，解码其中的控制点，把相对坐标转换为绝对坐标，然后把它加入到曲线对象中。如果需要的话，会对每个控制点做变换。  
回想一下，TrueType使用的是二阶Bezier曲线，允许在两个曲线上的点之间有多个不在曲线上的点。为了简化曲线绘制算法，KCurve::Add方法在每两个不在曲线上的点之间加入一个额外的在曲线上的点。  
  
        处理了简单图元之后，我们来看看合成图元。合成图元用一个经变换的图元序列定义。每个经变换的图元的定义包括三个部分：一个标志、一个图元索引和一个变换矩阵。标志字段决定了变换矩阵的编码方式。编码的目的也是为了节省一些空间，加外还说明了是否已到达序列的终点。一个完整的2D   affine变换需要6个值。但如果只是平移的话，只需要两个值（dx,dy），这两个值可以保存为两个字节或两个字。如果x和y以相同的值缩放，加外还需要一个缩放值。取一般的情况下仍然需要6个值，但是很多时候可以节省几个字节。用于变换的值以2.14的有符号定点格式保存，dx和dy值除外，这两个值以整数形式保存。得到合成图元的过程实际上是变换和组合几个图元的过程。比如，如果字体中的一个图元是另一个图元的精确镜像，它只需定义为一个合成图元，可以通过对另一个图像做镜像变换即可。程序清单14-3列出了解码合成图元的代码。  
  
  
int   KTrueType::DecodeCompositeGlyph(const   void   \*   pGlyph,   KCurve   &   curve)   const  
{  
KDataStream   str(pGlyph);  
  
unsigned   flags;  
  
int   len   =   0;  
  
do     
{  
flags             =   str.GetWord();  
  
unsigned   glyphIndex   =   str.GetWord();  
  
//   Argument1   and   argument2   can   be   either   x   and   y   offsets   to   be   added   to   the   glyph   or   two   point   numbers.     
//   In   the   latter   case,   the   first   point   number   indicates   the   point   that   is   to   be   matched   to   the   new   glyph.     
//   The   second   number   indicates   the   new   glyph's   "matched"   point.   Once   a   glyph   is   added,   its   point   numbers     
//   begin   directly   after   the   last   glyphs   (endpoint   of   first   glyph   +   1).     
  
//   When   arguments   1   and   2   are   an   x   and   a   y   offset   instead   of   points   and   the   bit   ROUND\_XY\_TO\_GRID   is   set   to   1,     
//   the   values   are   rounded   to   those   of   the   closest   grid   lines   before   they   are   added   to   the   glyph.     
//   X   and   Y   offsets   are   described   in   FUnits.     
  
signed   short   argument1;  
signed   short   argument2;  
  
if   (   flags   &   ARG\_1\_AND\_2\_ARE\_WORDS   )  
{  
argument1   =   str.GetWord();   //   (SHORT   or   FWord)   argument1;  
argument2   =   str.GetWord();   //   (SHORT   or   FWord)   argument2;  
}     
else     
{  
argument1   =   (signed   char)   str.GetByte();  
argument2   =   (signed   char)   str.GetByte();  
}  
  
signed   short   xscale,   yscale,   scale01,   scale10;  
  
xscale     =   1;  
yscale     =   1;  
scale01   =   0;  
scale10   =   0;  
  
if   (   flags   &   WE\_HAVE\_A\_SCALE   )  
{  
xscale   =   str.GetWord();  
yscale   =   xscale; //   Format   2.14     
}     
else   if   (   flags   &   WE\_HAVE\_AN\_X\_AND\_Y\_SCALE   )     
{  
xscale   =   str.GetWord();  
yscale   =   str.GetWord();  
}     
else   if   (   flags   &   WE\_HAVE\_A\_TWO\_BY\_TWO   )     
{  
xscale     =   str.GetWord();  
scale01   =   str.GetWord();  
scale10   =   str.GetWord();  
yscale     =   str.GetWord();  
}  
  
if   (   flags   &   ARGS\_ARE\_XY\_VALUES   )  
{  
XFORM   xm;  
  
xm.eDx     =   (float)   argument1;  
xm.eDy     =   (float)   argument2;  
xm.eM11   =   xscale     /   (float)   16384.0;  
xm.eM12   =   scale01   /   (float)   16384.0;  
xm.eM21   =   scale10   /   (float)   16384.0;  
xm.eM22   =   yscale     /   (float)   16384.0;  
  
len   +=   DecodeGlyph(glyphIndex,   curve,   &   xm);  
}  
else  
assert(false);  
}     
while   (   flags   &   MORE\_COMPONENTS   );  
  
if   (   flags   &   WE\_HAVE\_INSTRUCTIONS   )  
{  
unsigned   numInstr   =   str.GetWord();  
  
for   (unsigned   i=0;   i<numInstr;   i++)  
str.GetByte();  
}  
  
//   The   purpose   of   USE\_MY\_METRICS   is   to   force   the   lsb   and   rsb   to   take   on   a   desired   value.     
//   For   example,   an   i-circumflex   (Unicode   00ef)   is   often   composed   of   the   circumflex   and   a   dotless-i.     
//   In   order   to   force   the   composite   to   have   the   same   metrics   as   the   dotless-i,     
//   set   USE\_MY\_METRICS   for   the   dotless-i   component   of   the   composite.   Without   this   bit,     
//   the   rsb   and   lsb   would   be   calculated   from   the   HMTX   entry   for   the   composite   (or   would   need   to   be     
//   explicitly   set   with   TrueType   instructions).     
  
//   Note   that   the   behavior   of   the   USE\_MY\_METRICS   operation   is   undefined   for   rotated   composite   components.  
  
return   len;  
}  
  
  
  
        DecodeCompositeGlyph方法解码每个图元的标志、图元索引和变换矩阵，然后调用DecodeGlypgh方法进行解码。注意，对DecodeGlyph方法的调用包含一个有效的变换矩阵参数。当MORE\_COMPONENTS标志结束时，该方法随之结束。随书光盘中有该方法完整的源代码。  
        解码后的TrueType字体的图元要用GDI绘制还有一个小问题需要处理。GDI只绘制三阶Bezier曲线，因此从图元表解码所得的二阶Bezier曲线的控制点需要转换为三阶Bezier曲线的控制点。通过对Bezier曲线原始数学定义的研究，可以得到如下用GDI绘制二阶Bezier曲线的简单例程。  
  
//draw   a   2nd-degree   Bezier   curve   segment  
BOOL   Bezier2(HDC   hDC,int   &   x0,int   &   y0,   int   x1,   int   y1,   int   x2   ,int   y2)  
{  
//   p0   p1   p2   -   >   p0   (p0   +   2p1)/3   (2p1+p2)/3,   p2  
  
POINT   P[3]   =   {   {   (x0+2\*x1)/3,(y0+2\*y1)/3},  
                              {(2\*x1+x2)/3,(2\*y1+y2)/3},  
                              {x2,y2}   }；  
x0=x2;y0=y2;  
return   PolyBezierTo(hDC,P,3);  
}  
  
        对于用三个控制点（p0,p1,p2)定义的二阶Bezier曲线，相应的三阶Bezier曲线的控制点为（p0,(p0+2\*p1)/3,(2\*p1+p2)/3,p2)。  
  
  
4.图元指令  
        程序清单14-2和14-3给人的印象是TrueType字体的栅格器可以通过扫描和转换图元的轮廓来轻松地实现，比如，用GDI和StrokeAndFillPath函数来填充图元轮廓绘制出来的路径。这种简单的字体栅格器的实现并不是很有用，除非它只用于高分辨诣的设备如打印机等。  
        简单栅格器得到的图像笔画粗细不一，有信息的遗漏，有字符特征的损失以及不对称等缺陷。当点阵变小是，情况不会更糟。总之，简单字体栅格器在小尺寸时会产生字迹模糊的结果。在大尺寸时会产生不好看的结果，只有在点阵增大时结果才会改善。  
        当在大的em-square（典型的是2048）中定义的图元轮廓缩小为小得多的网格时（如32＊32），不可避免会损失精度并引入误差。  
  
        TrueType解决这个问题的方法是控制图元轮廓从em-square到栅格网格的缩放过程，使得到的结果看起来效果更好，和原始图元的设计尽量接近。这种技术被称为网格调整（grid   fitting），它想达到的目标有：  
        消除网格位置的可能影响，保证笔画的粗细和网格的相对位置无关。  
        控制图元中关键位置的尺寸  
        保持对称性和衬线等   重要的图元设计细节。  
      
          TrueType字体中网格调整的需求在两个地方中编码：控制值表（control   value   table)和每个图元的网格调整指令。  
        控制值表（"cvt"表)用于保存一个数组，这些值被称为网格调整指令。比如，对于有衬线的字体，基线、衬线高度、大写字母笔划的宽度等值都或以是被控制的值。它们可以以字体设计者已知的次序保存在控制值表中，然后用它们的索引来引用。在字体光栅化过程中，控制值表中的值根据点阵的大小缩放。在网络调整指令中引用这些值   可以保证使用的值与网枸的位置无关。比如，如果水平线[14，0，25，200]可以用CVT表中的两个值定义为[14，0，14+CVT[stem\_width],0+CVT[cap\_height]],那     么该线的宽度和高度会和所在网格的相对位置无关，保持不变。  
        每一个图元的定义中附加有一个指令序列，该指令序列被称为图元指令，该背景令序列用于控制此图元的网格高速。图元指令线用控制值表中的值，以保证在索引图元中这些值相同。  
        图元指令是一种基于堆栈的伪计算机的指令。堆栈计算机常用于计算机语言的解释性实现。比如，Forth(用于嵌入式系统的一种强大而简洁的语言）、RPL（HP计算器使用的语言）和Java虚拟机都是堆栈计算机。  
        堆栈计算机通常没有寄存器，所有的计算都在堆栈上进行（有些堆栈计算机使用分开的控制堆栈和数据堆栈）。比如，压入指令把一个值压入堆栈，弹出指令从堆栈中弹出上面的值，二元加法指令弹出上面的两   个值   ，然后把它们的和压入堆栈。