GIF(Graphics Interchange Format，图形交换格式)文件是由 CompuServe公司开发的图形文件格式，版权所有，任何商业目的使用均须 CompuServe公司授权。

GIF图象是基于颜色列表的（存储的数据是该点的颜色对应于颜色列表的索引值），最多只支持8位（256色）。GIF文件内部分成许多存储块，用来存储多幅图象或者是决定图象表现行为的控制块，用以实现动画和交互式应用。GIF文件还通过LZW压缩算法压缩图象数据来减少图象尺寸（关于LZW算法和GIF数据压缩>>...）。

2.GIF文件存储结构

~——~~~

GIF文件内部是按块划分的，包括控制块（ Control Block ）和数据块（Data Sub-blocks）两种。控制块是控制数据块行为的，根据不同的控制块包含一些不同的控制参数；数据块只包含一些8-bit的字符流，由它前面的控制块来决定它的功能，每个数据块大小从0到255个字节，数据块的第一个字节指出这个数据块大小（字节数），计算数据块的大小时不包括这个字节，所以一个空的数据块有一个字节，那就是数据块的大小0x00。下表是一个数据块的结构：

BYTE	7	6	5	4	3	2	1	0	BIT
0	块大小	Block Size - 块大小，不包括这个这个字节（不计算块大小自身）							
1		Data Values - 块数据，8-bit的字符串							
2									
...									
254									
255									

一个GIF文件的结构可分为文件头(File Header)、GIF数据流(GIF Data Stream)和文件终结器(Trailer)三个部分。文件头包含GIF文件署名(Signature)和版本号(Version)；GIF数据流由控制标识符、图象块(Image Block)和其他的一些扩展块组成；文件终结器只有一个值为0x3B的字符（';'）表示文件结束。下表显示了一个GIF文件的组成结构：

	GIF署名	**文件头**	
	版本号		
	逻辑屏幕标识符	GIF数据流	
	全局颜色列表		
	…		
	图象标识符	图象块	
	图象局部颜色列表图		
	基于颜色列表的图象数据		
	…		
	GIF结尾	文件结尾	

下面就具体介绍各个部分:

文件头部分(Header)

~~~~

GIF署名(Signature)和版本号(Version)

~~~~~~

GIF署名用来确认一个文件是否是GIF格式的文件，这一部分由三个字符组成："GIF";文件版本号也是由三个字节组成,可以为"87a"或"89a".具体描述见下表:

BYTE	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**BIT**
1	'G'	GIF文件标识							
2	'I'								
3	'F'								
4	'8'	GIF文件版本号：87a - 1987年5月 89a - 1989年7月							
5	'7'或'9'								
6	'a'								

GIF数据流部分(GIF Data Stream)

~~~~~~~~
~~~~~~~~

逻辑屏幕标识符(Logical Screen Descriptor)

这一部分由7个字节组成，定义了GIF图象的大小(Logical Screen Width & Height)、颜色深度(Color Bits)、背景色(Blackground Color Index)以及有无全局颜色列表(Global Color Table)和颜色列表的索引数(Index Count)，具体描述见下表：

BYTE	7	6	5	4	3	2	1	0	BIT	
1	逻辑屏幕宽度	像素数，定义GIF图象的宽度								
2										
3	逻辑屏幕高度	像素数，定义GIF图象的高度								
4										
5	m	cr	s	pixel	具体描述见下...					
6	背景色	背景颜色(在全局颜色列表中的索引，如果没有全局颜色列表，该值没有意义)								
7	像素宽高比	像素宽高比(Pixel Aspect Radio)								

m - 全局颜色列表标志(Global Color Table Flag)，当置位时表示有全局颜色列表，pixel值有意义.
cr - 颜色深度(Color ResoluTion)，cr+1确定图象的颜色深度.
s - 分类标志(Sort Flag)，如果置位表示全局颜色列表分类排列.
pixel - 全局颜色列表大小，pixel+1确定颜色列表的索引数（2的pixel+1次方）.

全局颜色列表(Global Color Table)

~~~~~~~~~~

全局颜色列表必须紧跟在逻辑屏幕标识符后面，每个颜色列表索引条目由三个字节组成，按R、G、B的顺序排列。

BYTE	7	6	5	4	3	2	1	0	BIT
1	索引1的红色值								
2	索引1的绿色值								
3	索引1的蓝色值								
4	索引2的红色值								
5	索引2的绿色值								
6	索引2的蓝色值								
7	...								

图象标识符(Image Descriptor)

~~~~~~~~~

一个GIF文件内可以包含多幅图象，一幅图象结束之后紧接着下是一幅图象的标识符，图象标识符以0x2C(',')字符开始，定义紧接着它的图象的性质，包括图象相对于逻辑屏幕边界的偏移量、图象大小以及有无局部颜色列表和颜色列表大小，由10个字节组成：

BYTE	7	6	5	4	3	2	1	0	BIT	
1	0	0	1	0	1	1	0	0	图象标识符开始，固定值为','	
2	X方向偏移量	必须限定在逻辑屏幕尺寸范围内								
3										
4	Y方向偏移量									
5										
6	图象宽度									
7										
8	图象高度									
9										

BYTE	7	6	5	4	3	2	1	0	BIT	
10	m	i	s	r	pixel	m - 局部颜色列表标志(Local Color Table Flag)				
							置位时标识紧接在图象标识符之后有一个局部颜色列表，供紧跟在它之后的一幅图象使用；值否时使用全局颜色列表，忽略pixel值。 i - 交织标志(Interlace Flag)，置位时图象数据使用交织方式排列（详细描述...），否则使用顺序排列。 s - 分类标志(Sort Flag)，如果置位表示紧跟着的局部颜色列表分类排列. r - 保留，必须初始化为0. pixel - 局部颜色列表大小(Size of Local Color Table)，pixel+1就为颜色列表的位数			

局部颜色列表(Local Color Table)

~———

如果上面的局部颜色列表标志置位的话，则需要在这里（紧跟在图象标识符之后）定义一个局部颜色列表以供紧接着它的图象使用，注意使用前应线保存原来的颜色列表，使用结束之后回复原来保存的全局颜色列表。如果一个GIF文件即没有提供全局颜色列表，也没有提供局部颜色列表，可以自己创建一个颜色列表，或使用系统的颜色列表。局部颜色列表的排列方式和全局颜色列表一样：RGBRGB.....

基于颜色列表的图象数据(Table-Based Image Data)

~————~~

由两部分组成：LZW编码长度(LZW Minimum Code Size)和图象数据(Image Data)。

BYTE	7	6	5	4	3	2	1	0	BIT
1	LZW编码长度	LZW编码初始码表大小的位数，详细描述见LZW编码...							
	...	图象数据，由一个或几个数据块(Data Sub-blocks)组成							
数据块									
...									

GIF图象数据使用了LZW压缩算法（详细介绍请看后面的『LZW算法和GIF数据压缩』），大大减小了图象数据的大小。图象数据在压缩前有两种排列格式：连续的和交织的(由图象标识符的交织标志控制)。连续方式按从左到右、从上到下的顺序排列图象的光栅数据；交织图象按下面的方法处理光栅数据：

创建四个通道(pass)保存数据，每个通道提取不同行的数据：
第一通道(Pass 1)提取从第0行开始每隔8行的数据；
第二通道(Pass 2)提取从第4行开始每隔8行的数据；
第三通道(Pass 3)提取从第2行开始每隔4行的数据；
第四通道(Pass 4)提取从第1行开始每隔2行的数据；

下面的例子演示了提取交织图象数据的顺序：

行	通道1	通道2	通道3	通道4	
0 ----------	1				
1 ----------				4	
2 ----------			3		
3 ----------				4	
4 ----------		2			
5 ----------				4	
6 ----------			3		
7 ----------				4	
8 ----------	1				
9 ----------				4	
10 ----------			3		
11 ----------				4	
12 ----------		2			
13 ----------				4	
14 ----------			3		
15 ----------				4	
16 ----------	1				
17 ----------				4	
18 ----------			3		
19 ----------				4	
20 ----------		2			

图形控制扩展(Graphic Control Extension)

~━━━━━~~~~

这一部分是可选的（需要89a版本），可以放在一个图象块(图象标识符)或文本扩展块的前面，用来控制跟在它后面的第一个图象（或文本）的渲染(Render)形式，组成结构如下：

BYTE	7	6	5	4	3	2	1	0	BIT
1	扩展块标识	Extension Introducer - 标识这是一个扩展块，固定值0x21							
2	图形控制扩展标签	Graphic Control Label - 标识这是一个图形控制扩展块，固定值0xF9							
3	块大小	Block Size - 不包括块终结器，固定值4							
4	保留	处置方法	i	t	i - 用户输入标志；t - 透明色标志。详细描述见下...				
5	延迟时间	Delay Time - 单位1/100秒，如果值不为1，表示暂停规定的时间后再继续往下处理数据流							
6									
7	透明色索引	Transparent Color Index - 透明色索引值							
8	块终结器	Block Terminator - 标识块终结，固定值0							

处置方法(Disposal Method)：指出处置图形的方法，当值为：

0 - 不使用处置方法
1 - 不处置图形，把图形从当前位置移去
2 - 回复到背景色
3 - 回复到先前状态
4-7 - 自定义

用户输入标志(Use Input Flag)：指出是否期待用户有输入之后才继续进行下去，置位表示期待，值否表示不期待。用户输入可以是按回车键、鼠标点击等，可以和延迟时间一起使用，在设置的延迟时间内用

户有输入则马上继续进行，或者没有输入直到延迟时间到达而继续
透明颜色标志(Transparent Color Flag)：置位表示使用透明颜色

注释扩展(Comment Extension)
~~~~~~~~

这一部分是可选的（需要89a版本），可以用来记录图形、版权、描述等任何的非图形和控制的纯文本数据(7-bit ASCII字符)，注释扩展并不影响对图象数据流的处理，解码器完全可以忽略它。存放位置可以是数据流的任何地方，最好不要妨碍控制和数据块，推荐放在数据流的开始或结尾。具体组成：

BYTE	7	6	5	4	3	2	1	0	BIT
1	扩展块标识	Extension Introducer - 标识这是一个扩展块，固定值0x21							
2	注释块标签	Comment Label - 标识这是一个注释块，固定值0xFE							
	…	Comment Data - 一个或多个数据块(Data Sub-Blocks)组成							
注释块									
…									
	块终结器	Block Terminator - 标识注释块结束，固定值0							

图形文本扩展(Plain Text Extension)
~~~~~~~~~~~~

这一部分是可选的（需要89a版本），用来绘制一个简单的文本图象，这一部分由用来绘制的纯文本数据（7-bit ASCII字符）和控制绘制的参数等组成。绘制文本借助于一个文本框（Text Grid）来定义边界，在文本框中划分多个单元格，每个字符占用一个单元，绘制时按从左到右、从上到下的顺序依次进行，直到最后一个字符或者占满整个文本框（之后的字符将被忽略，因此定义文本框的大小时应该注意到是否可以容纳整个文本），绘制文本的颜色索引使用全局颜色列表，没有则可以使用一个已经保存的前一个颜色列表。另外，图形文本扩展块也属于图形块(Graphic Rendering Block)，可以在它前面定义图形控制扩展对它的表现形式进一步修改。图形文本扩展的组成：

BYTE	7	6	5	4	3	2	1	0	BIT
1	扩展块标识	Extension Introducer - 标识这是一个扩展块，固定值0x21							
2	图形控制扩展标签	Plain Text Label - 标识这是一个图形文本扩展块，固定值0x01							
3	块大小	Block Size - 块大小，固定值12							
4	文本框左边界位置	Text Glid Left Posotion - 像素值，文本框离逻辑屏幕的左边界距离							
5									
6	文本框上边界位置	Text Glid Top Posotion - 像素值，文本框离逻辑屏幕的上边界距离							
7									
8	文本框高度	Text Glid Width -像素值							
9									
10	文本框高度	Text Glid Height - 像素值							
11									
12	字符单元格宽度	Character Cell Width - 像素值，单个单元格宽度							
13	字符单元格高度	Character Cell Height- 像素值，单个单元格高度							
14	文本前景色索引	Text Foreground Color Index - 前景色在全局颜色列表中的索引							

BYTE	7	6	5	4	3	2	1	0	BIT
15	文本背景色索引	Text Blackground Color Index - 背景色在全局颜色列表中的索引							
N	...	Plain Text Data - 一个或多个数据块(Data Sub-Blocks)组成，保存要在显示的字符串。							
文本数据块									
...									
N+1	块终结	Block Terminator - 标识注释块结束，固定值0							

推荐：1.由于文本的字体(Font)和尺寸(Size)没有定义，解码器应该根据情况选择最合适的；
2.如果一个字符的值小于0x20或大于0xF7，则这个字符被推荐显示为一个空格(0x20)；
3.为了兼容性，最好定义字符单元格的大小为8x8或8x16（宽度x高度）。

应用程序扩展(Application Extension)
~~~~~~~~~~~~

这是提供给应用程序自己使用的（需要89a版本），应用程序可以在这里定义自己的标识、信息等，组成：
~~~~~~~~~~~~

BYTE	7	6	5	4	3	2	1	0	BIT
1	扩展块标识	Extension Introducer - 标识这是一个扩展块，固定值0x21							
2	图形控制扩展标签	Application Extension Label - 标识这是一个应用程序扩展块，固定值0xFF							
3	块大小	Block Size - 块大小，固定值11							
4	应用程序标识符	Application Identifier - 用来鉴别应用程序自身的标识(8个连续ASCII字符)							
5									
6									
7									
8									
9									
10									
11									
12	应用程序鉴别码	Application Authentication Code - 应用程序定义的特殊标识码(3个连续ASCII字符)							
13									
14									
N	...	应用程序自定义数据块 - 一个或多个数据块(Data Sub-Blocks)组成，保存应用程序自己定义的数据							
应用程序数据									
...									
N+1	块终结器	lock Terminator - 标识注释块结束，固定值0							

文件结尾部分
~~~

文件终结器(Trailer)
~~~~

这一部分只有一个值为0的字节，标识一个GIF文件结束.

BYTE	7	6	5	4	3	2	1	0	
1	文件终结	GIF Trailer - 标识GIF文件结束，固定值0x3B							

2.LZW算法和GIF数据压缩
~~~~~~~

GIF文件的图象数据使用了可变长度编码的LZW压缩算法(Variable-Length_Code LZW Compression)，这是从LZW(Lempel Ziv Compression)压缩算法演变过来的，通过压缩原始数据的重复部分来达到减少文件大小的目的。

标准的LZW压缩原理：
~~~~~~

先来解释一下几个基本概念：

LZW压缩有三个重要的对象：数据流(CharStream)、编码流(CodeStream)和编译表(String Table)。在编码时，数据流是输入对象（图象的光栅数据序列），编码流就是输出对象（经过压缩运算的编码数据）；在解码时，编码流则是输入对象，数据流是输出对象；而编译表是在编码和解码时都须要用借助的对象。

字符(Character)：最基础的数据元素，在文本文件中就是一个字节，在光栅数据中就是一个像素的颜色在指定的颜色列表中的索引值；
字符串(String)：由几个连续的字符组成；
前缀(Prefix)：也是一个字符串，不过通常用在另一个字符的前面，而且它的长度可以为0；
根(Root)：单个长度的字符串；
编码(Code)：一个数字，按照固定长度（编码长度）从编码流中取出，编译表的映射值；
图案：一个字符串，按不定长度从数据流中读出,映射到编译表条目.

LZW压缩的原理：提取原始图象数据中的不同图案，基于这些图案创建一个编译表，然后用编译表中的图案索引来替代原始光栅数据中的相应图案，减少原始数据大小。看起来和调色板图象的实现原理差不多，但是应该注意到的是，我们这里的编译表不是事先创建好的，而是根据原始图象数据动态创建的，解码时还要从已编码的数据中还原出原来的编译表（GIF文件中是不携带编译表信息的），为了更好理解编解码原理，我们来看看具体的处理过程：

编码器(Compressor)
~~~~~

编码数据，第一步，初始化一个编译表，假设这个编译表的大小是12位的，也就是最多有4096个单位，另外假设我们有32个不同的字符（也可以认为图象的每个像素最多有32种颜色），表示为a，b，c，d，e...，初始化编译表：第0项为a，第1项为b，第2项为c...一直到第31项，我们把这32项就称为根。

开始编译，先定义一个前缀对象Current Prefix，记为[.c.]，现在它是空的，然后定义一个当前字符串Current String，标记为[.c.]k，[.c.]就为Current Prefix，k就为当前读取字符。现在来读取数据流的第一个字符，假如为p，那么Current String就等于[.c.]p（由于[.c.]为空，实际上值就等于p），现在在编译表中查找有没有Current String的值，由于p就是一个根字符，我们已经初始了32个根索引，当然可以
~~~~~

找到，把p设为Current Prefix的值，不做任何事继续读取下一个字符，假设为q，Current String就等于[.c.]q（也就是pq），看看在编译表中有没有该值，当然。没有，这时我们要做下面的事情：将Current String的值（也就是pq）添加到编译表的第32项，把Current Prefix的值（也就是p）在编译表中的索引输出到编码流，修改Current Prefix为当前读取的字符（也就是q）。继续往下读，如果在编译表中可以查找到Current String的值([.c.]k)，则把Current String的值([.c.]k)赋予Current Prefix；如果查找不到，则添加Current String的值([.c.]k)到编译表，把Current Prefix的值([.c.])在编译表中所对应的索引输出到编码流，同时修改Current Prefix为k，这样一直循环下去直到数据流结束。伪代码看起来就像下面这样：

Initialize String Table; [.c.] = Empty; [.c.]k = First Character in CharStream; while ([.c.]k != EOF) { if ([.c.]k is in the StringTable) { [.c.] = [.c.]k; } else { add [.c.]k to the StringTable; Output the Index of [.c.] in the StringTable to the CodeStream; [.c.] = k; } [.c.]k = Next Character in CharStream; } Output the Index of [.c.] in the StringTable to the CodeStream;

来看一个具体的例子,我们有一个字母表a，b，c，d.有一个输入的字符流abacaba。现在来初始化编译表：#0=a,#1=b,#2=c,#3=d.现在开始读取第一个字符a，[.c.]a=a，可以在在编译表中找到，修改[.c.]=a;不做任何事继续读取第二个字符b，[.c.]b=ab，在编译表中不能找，那么添加[.c.]b到编译表：#4=ab，同时输出[.c.]（也就是a）的索引#0到编码流，修改[.c.]=b；读下一个字符a，[.c.]a=ba，在编译表中不能找到：添加编译表#5=ba，输出[.c.]的索引#1到编码流，修改[.c.]=a；读下一个字符c，[.c.]c=ac，在编译表中不能找到：添加编译表#6=ac，输出[.c.]的索引#0到编码流，修改[.c.]=c；读下一个字符a，[.c.]c=ca，在编译表中不能找到：添加编译表#7=ca，输出[.c.]的索引#2到编码流，修改[.c.]=a；读下一个字符b，[.c.]b=ab，编译表的#4=ab，修改[.c.]=ab；读取最后一个字符a，[.c.]a=aba，在编译表中不能找到：添加编译表#8=aba，输出[.c.]的索引#4到编码流，修改[.c.]=a；好了，现在没有数据了，输出[.c.]的值a的索引#0到编码流，这样最后的输出结果就是：#0#1#0#2#4#0.

解码器(Decompressor)

~~~~~

好了，现在来看看解码数据。数据的解码，其实就是数据编码的逆向过程，要从已经编译的数据（编码流）中找出编译表，然后对照编译表还原图象的光栅数据。

首先，还是要初始化编译表。GIF文件的图象数据的第一个字节存储的就是LZW编码的编码大小（一般等于图象的位数），根据编码大小，初始化编译表的根条目（从0到2的编码大小次方），然后定义一个当前编码Current Code，记作[code]，定义一个Old Code，记作[old]。读取第一个编码到[code]，这是一个根编码，在编译表中可以找到，把该编码所对应的字符输出到数据流，[old]=[code]；读取下一个编码到[code]，这就有两种情况：在编译表中有或没有该编码，我们先来看第一种情况：先输出当前编码[code]所对应的字符串到数据流，然后把[old]所对应的字符（串）当成前缀prefix [...]，当前编码[code]所对应的字符串的第一个字符当成k，组合起来当前字符串Current String就为[...]k，把[...]k添加到编译表，修改[old]=[code]，读下一个编码；我们来看看在编译表中找不到该编码的情况，回想一下编码情况：如果数据流中有一个p[...]p[...]pq这样的字符串，p[...]在编译表中而p[...]p不在，编译器将输出p[...]的索引而添加p[...]p到编译表，下一个字符串p[...]p就可以在编译表中找到了，而p[...]pq不在编译表中，同样将输出p[...]p的索引值而添加p[...]pq到编译表，这样看来，解码器总比编码器『慢一步』，当我们遇到p[...]p所对应的索引时，我们不知到该索引对应的字符串（在解码器的编译表中还没有该索引，事实上，这个索引将在下一步添加），这时需要用猜测法：现在假设上面的p[...]所对应的索引值是#58，那么上面的字符串经过编译之后是#58#59，我们在解码器中读到#59时，编译表的最大索引只有#58，#59所对应的字符串就等于#58所对应的字符串(也就是p[...])加上这个字符串的第一个字符(也就是p)，也就是p[...]p。事实上，这种猜测法是很准确（有点不好理解，仔细想一想吧）。上面的解码过程用伪代码表示就像下面这样：
~~~~~

Initialize String Table; [code] = First Code in the CodeStream; Output the String for [code] to the CharStream; [old] = [code]; [code] = Next Code in the CodeStream; while ([code] != EOF) { if ([code] is in the StringTable) { Output the String for [code] to the CharStream; // 输出[code]所对应的字符串 [...] = translation for [old]; // [old]所对应的字符串 k = first character of translation for [code]; // [code]所对应的字符串的第一个字符 add [...]k to the StringTable; [old] = [code]; } else { [...] = translation for [old]; k = first character of [...]; Output [...]k to CharStream; add [...]k to the StringTable; [old] = [code]; } [code] = Next Code in the CodeStream; }

GIF数据压缩

~—

下面是GIF文件的图象数据结构：

BYTE	7	6	5	4	3	2	1	0	BIT
1	编码长度	LZW Code Size - LZW压缩的编码长度，也就是要压缩的数据的位数							
	...	数据块							
	块大小	数据块，如果需要可重复多次							
	编码数据								
	...	数据块							
	块终结器	一个图象的数据编码结束，固定值0							

把光栅数据序列（数据流）压缩成GIF文件的图象数据（字符流）可以按下面的步骤进行：

1.定义编码长度

GIF图象数据的第一个字节就是编码长度(Code Size)，这个值是指要表现一个像素所需要的最小位数，通常就等于图象的色深；

2.压缩数据

通过LZW压缩算法将图象的光栅数据流压缩成GIF的编码数据流。这里使用的LZW压缩算法是从标准的LZW压缩算法演变过来的，它们之间有如下的差别：

[1]GIF文件定义了一个编码大小(Clear Code)，这个值等于2的『编码长度』次方，在从新开始一个编译表（编译表溢出）时均须输出该值，解码器遇到该值时意味着要从新初始化一个编译表；

[2]在一个图象的编码数据结束之前（也就是在块终结器的前面），需要输出一个Clear Code+1的值，解码器在遇到该值时就意味着GIF文件的一个图象数据流的结束；

[3]第一个可用到的编译表索引值是Clear Code+2（从0到Clear Code-1是根索引，再上去两个不可使用，新的索引从Clare Code+2开始添加）；

[4]GIF输出的编码流是不定长的，每个编码的大小从Code Size + 1位到12位，编码的最大值就是4095（编译表需要定义的索引数就是4096），当编码所须的位数超过当前的位数时就把当前位数加1，这就需要在编码或解码时注意到编码长度的改变。

3.编译成字节序列

因为GIF输出的编码流是不定长的，这就需要把它们编译成固定的8-bit长度的字符流，编译顺序是从右往左。下面是一个具体例子：编译5位长度编码到8位字符

0	**b**	**b**	**b**	**a**	**a**	**a**	**a**	**a**
1	d	c	c	c	c	c	b	b
2	e	e	e	e	d	d	d	d
3	g	g	f	f	f	f	f	e
4	h	h	h	h	h	g	g	g
	…							
N								

4.打包

前面讲过，一个GIF的数据块的大小从0到255个字节，第一个字节是这个数据块的大小（字节数），这就需要将编译编后的码数据打包成一个或几个大小不大于255个字节的数据包。然后写入图象数据块中。