图像处理中的即时代码生成

Charles Petzold

《代码之美》第8章

张可

引言

- Bill Gosper: 数据只不过是一种笨程序
- 作者认为，代码也只不过是一种“聪明的数据”。聪明之处在于，它可以驱使处理器执行一些有用的动作。

- 本章中，作者为了提高数字图像过滤器的效率，提出了一种即时生成代码的方法
- 也就是让我们的程序自己根据数据的具体情况生成一段中间代码，并运行这些代码

- 这篇文章将会介绍：在什么情况下需要即时生成代码？并将通过一个实例演示即时生成代码的具体做法。

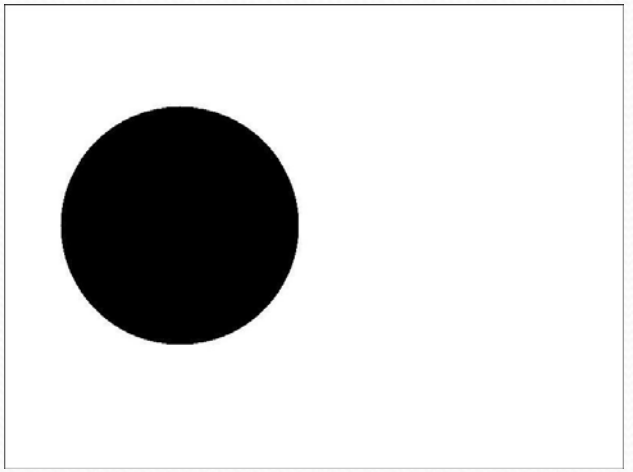
从Windows的BitBlt函数说起

- Bitblt 是 Windows 1.0 中的一个图形函数，其中就使用了即时代码生成技术。
- 这个函数的目标是这样的：
- 最基本地， Bitblt可以将一块长方形的像素块从一个地方传输到另一个地方，比如PrintScreen
- 其次，应该可以对这块长方形像素进行一些操作，比如求逆

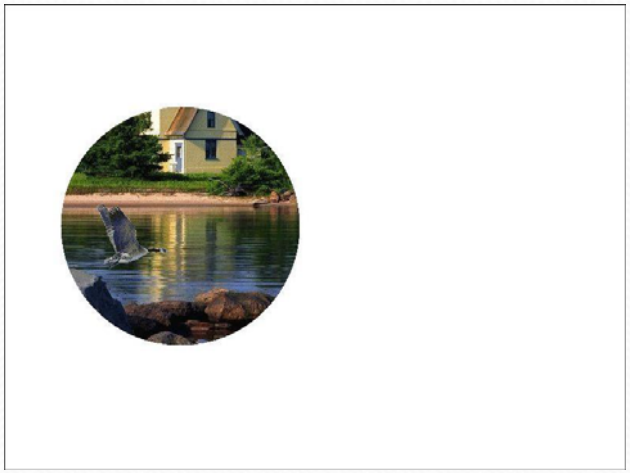
从Windows的BitBlt函数说起

- 新的功能：在传输位图时只传输对应于目的地上那些被预置为黑色的部分。

源位图



目标位图



结果

从Windows的BitBlt函数说起

- 进一步，可以考察一种更一般的情形：
- 考虑一个黑白图形系统，其中无论是源位图还是目标位图，每个像素都由1 bit表示。
- 在这个系统中，源位图和目标位图都用一个bit数组表示。
- 我们得到的结果也是一个bit数组，其中每一个元素都取决于相对应的源位图的元素和目标位图的元素。
- 这种操作被称为光栅操作，总共有2^4=16种

从Windows的BitBlt函数说起

可能的组合

参数名	参数值	
源 (S):	1 1 0 0	
目标 (D):	1 0 1 0	
操作	输出	逻辑表示
光栅操作0:	0 0 0 0	0
光栅操作1:	0 0 0 1	~(S | D)
光栅操作2:	0 0 1 0	~S & D
光栅操作3:	0 0 1 1	~S
光栅操作4:	0 1 0 0	S & ~D
光栅操作5:	0 1 0 1	~D
光栅操作6:	0 1 1 0	S ^ D
光栅操作7:	0 1 1 1	~(S & D)
光栅操作8:	1 0 0 0	S & D
光栅操作9:	1 0 0 1	~(S ^ D)
光栅操作10:	1 0 1 0	D
光栅操作11:	1 0 1 1	~S | D
光栅操作12:	1 1 0 0	S
光栅操作13:	1 1 0 1	S |~D
光栅操作14:	1 1 1 0	S | D
光栅操作15:	1 1 1 1	1

比如光栅操作12，就是直接把源位图直接传输到目的地，而不管目的地原来是什么颜色。

从Windows的BitBlt函数说起

- 实际Windows的Bitblt是更为复杂的。

- 决定输出像素的不仅是源像素和目标像素，还有画刷（pattern）。画刷是一个和源一样大的bit数组，它是为了在不知道目标位图的情况下对源位图执行一些操作。

Example

可能的组合	
参数名	参数值
画刷（P）:	1 1 1 1 0 0 0 0
源（S）:	1 1 0 0 1 1 0 0
目标（D）:	1 0 1 0 1 0 1 0
操作	输出
光栅操作0x00:	0 0 0 0 0 0 0 0
光栅操作0x01:	0 0 0 0 0 0 0 1
光栅操作0x02:	0 0 0 0 0 0 1 0
...	...
光栅操作0x60:	0 1 1 0 0 0 0 0
...	...
光栅操作0xFD:	1 1 1 1 1 1 0 1
光栅操作0xFE:	1 1 1 1 1 1 1 0
光栅操作0xFF:	1 1 1 1 1 1 1 1

从Windows的BitBlt函数说起

- Bitblt支持所有这256种光栅操作。在用户使用的时候，可以通过源，目标和画刷之间的逻辑运算来调用任意一种光栅操作。
- 虽然这256种光栅操作中真正有用的并不多，但作者认为这样完备和多样选择的感觉可以让人安心。

- 另外，虽然这里举的例子是单色的图像系统，但把它扩展到彩色图像系统上也是容易的。一般来说，在彩色系统中，每个像素由24bit表示，RGB的分量各占8bit。对于一个光栅操作(比如0xCF，也即~P |S)，只要把这里的~和|当作位操作就可以了，也即把源和画刷的24bit依次执行~P |S运算，得到的就是光栅操作的结果。

BitBlt函数的实现

- 如果让我们自己来写Bitblt函数，我们应该怎么写？

- 考虑一个灰度系统，每个像素占一个字节大小，0x00代表黑色，0xFF代表白色
- 我们有S，D，P三个二维数组，分别代表源、目标和画刷。比如S[y][x]就可以访问源位图第y行x列的元素。
- 这三个二维数组是一样大的，都是cy*cx。

- 用rop表示需要执行的光栅操作的代码（从0到255编号）

BitBlt函数的实现

- 最直接的做法是用switch-case：

```
for (y = 0; y < cy; y++)
for (x = 0; x < cx; x++)
{
    switch(rop)
    {
    case 0x00:
        D[y][x] = 0x00;
        break;
    ...
    case 0x60:
        D[y][x] = (D[y][x] ^ S[y][x]) & P[y][x];
        break;
    ...
    case 0xFF:
        D[y][x] = 0xFF;
        break;
    }}
```

BitBlt函数的实现

- 上面这段代码形式上算是漂亮的代码，但是它的运行效率是一个灾难。
- 因为位图的特点是庞大，如今的数码相机动辄都能包含几百万像素。对这几百万像素分别执行一遍那个大大的switch语句，效率可想而知。
- 由此可以想到一个简单的改进：我们不需要对每个像素都判断一遍switch语句，只需要在一开始判断需要执行哪种光栅操作，然后再执行双重for循环，效率就能得到显著的提高。

BitBlt函数的实现

```
switch(rop)
{
case 0x00:
    for (y = 0; y < cy; y++)
    for (x = 0; x < cx; x++)
        D[y][x] = 0x00;
    break;
...
case 0x60 :
    for (y = 0; y < cy; y++)
    for (x = 0; x < cx; x++)
        D[y][x] = (D[y][x] ^ S[y][x]) & P[y][x];
    break;
...
case 0xFF:
    for (y = 0; y < cy; y++)
    for (x = 0; x < cx; x++)
        D[y][x] = 0xFF;
    break;
}
```

BitBlt函数的实现

- 即使这样，对于Bitblt这种极其重要的函数，还是不够快。
- 用汇编语言来写能比C语言快一些，但Windows的程序员们想出了更加神奇的方法，这种方法“比汇编还快”：
- Windows的Bitblt实现包括了一个迷你编译器，它可以先读入参数rop、位图大小cx*cy等，然后根据需要执行的特定的光栅操作，Bitblt会在栈上建立一个由机器代码构成的子进程，这个子进程不需要任何判断，而是对每个像素直接执行特定的光栅操作。

即时生成代码的常见动机

- 有一个对执行效率要求极高的函数，它需要执行许多重复的操作。
- 在这些重复操作的执行过程中需要一些参数，如果知道参数的具体值，函数的执行效率就能提高很多。
- 但是编写函数时，我们并不知道参数的具体值。那么，我们就得让函数充当编译器的角色，让它在运行时查看各参数值，然后生成针对特定参数值执行的、更加高效的代码。

数字图像过滤器

- 通过上面的例子，我们也可以了解数字图像处理的一个特点：不管使用什么样的编码技术，一定要让每像素的处理尽可能的快！
- 作者在用C#写数字图像过滤器时，也想到了这种即时代码生成的方法。

- 右图是一个简单的数字图像过滤器，它可以把一张源位图转换成大小相同的目标位图。目标位图中每一个像素的值，由源位图中对应的像素以及其周围八个像素的值取平均。

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

模糊过滤器

数字图像过滤器

```
for (yDestination = 0; yDestination < cyBitmap; yDestination++)
for (xDestination = 0; xDestination < cxBitmap; xDestination++)
{
    double pixelsAccum = 0;
    double filterAccum = 0;
    for (yFilter = 0; yFilter < cyFilter; yFilter++)
    for (xFilter = 0; xFilter < cxFilter; xFilter++)
    {
        int ySource = yDestination + yFilter - cyFilter / 2;
        int xSource = xDestination + xFilter - cxFilter / 2;
        if (ySource >= 0 && ySource < cyBitmap && xSource >= 0 && xSource < cxBitmap)
        {
            pixelsAccum += F[y][x] * S[y][x];
            filterAccum += F[y][x];
        }
    }
    if (filterAccum != 0)  pixelsAccum /= filterAccum;
    if (pixelsAccum < 0)  D[y][x] = 0;
    else if (pixelsAccum > 255)  D[y][x] = 255;
    else  D[y][x] = (unsigned char) pixelsAccum;
}
```

cxBitmap和cyBitmap分别是源位图的宽和高

cxFilter和cyFilter分别是过滤器的宽和高

D[y][x]是存储目标位图的数组

数字图像过滤器

- 不难发现，为了计算目标位图的每一个像素，源位图和过滤阵列都必须被访问(cxFilter×cyFilter)遍。

- 当位图的分辨率增加的时候，通常过滤阵列的大小也要随之增加，因为只有这样才能确保对图像产生一个可观察到的影响。

- 这个工作量还是挺大的。为了测试它的效率，作者首先用C#编写了一个过滤器的函数。这个函数的基本思想和刚才的C代码是一样的，只是把存储源位图和目标位图的二维数组都改成了一维数组（为了提高效率）。

FilterMethodCS方法

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30    } }
```

stride是源位图和目标位图每行所占的字节数

FilterMethodCS方法

- 但是，上面这个FilterMethodCS的效率并不能让人满意，而且这段代码的速度看起来已经很难提高了。
- FilterMethodCS的真正问题在于它是一个泛化了的方法：它能够处理任意的位图和过滤阵列。导致FilterMethodCS中的大量代码都是在循环和索引。

- 如果只需要让FilterMethodCS处理一种特殊的情况：源位图每像素为32位，大小为CX*CY(常数)，并且总是使用一个特定的过滤阵列，其中所有的元素F11~F33的值都是固定的。

F11	F12	F13
F21	F22	F23
F31	F32	F33

解循环

```
// Filter cell F11
int iSrc = iDst - 4 * CX - 4;
if (iSrc >= 0 && iSrc < 4 * CX * CY)
{
    pixelsAccum += src[iSrc] * F11;
    filterAccum += F11;
}

// Filter cell F12
iSrc = iDst - 4 * CX;
if (iSrc >= 0 && iSrc < 4 * CX * CY)
{
    pixelsAccum += src[iSrc] * F12;
    filterAccum += F12;
}

// Filter cells F13 through F32...

// Filter cell F33
iSrc = iDst + 4 * CX + 4;
if (iSrc >= 0 && iSrc < 4 * CX * CY)
{
    pixelsAccum += src[iSrc] * F33;
    filterAccum += F33;
}
```

```
for (int iFilter = 0; iFilter < cFilter; iFilter++)
{
   int yFilter = iFilter / cyFilter;
   int xFilter = iFilter % cxFilter;
   int iSrc = iDst + stride * (yFilter - cyFilter/2)
                   + bytesPerPixel * (xFilter - cxFilter/2);

   if (iSrc >= 0 && iSrc < cBytes)
   {
      pixelsAccum += filter[iFilter] * src[iSrc];
      filterAccum += filter[iFilter];
   }
}
```

经过对比可以发现，这种手动解循环的方法消除了循环逻辑，简化了iSrc的计算，消除了对过滤阵列的访问。

不仅如此，由于你知道过滤阵列所有元素F11~F33的确切值，你甚至可以在代码中删去那些过滤阵列元素值为0的部分，并简化过滤阵列值为1或-1的情况。

效率测试

	FilterMethodCS	手动解循环
800*600, 3Bytes/Pixel, Filter	328ms	156ms
800*600, 3Bytes/Pixel, Filter	936ms	328ms
800*600, 3Bytes/Pixel, Filter	898ms	172ms
1920*1200, 3Bytes/Pixel,	1703ms	816ms
1920*1200, 3Bytes/Pixel,	4266ms	1391ms
1920*1200, 3Bytes/Pixel,	4172ms	750ms

1	1	1
1	1	1
1	1	1

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

0	0	-1	0	0
0	-1	-2	-1	0
-	-2	1	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

问题

- 我们已经知道：
  - 我们的程序需要快速处理各种源位图和过滤阵列
  - 对于任意一个**特例**，我们有办法提高它的效率

- 现在的问题是：
  - 怎样使得**所有**的情况下程序的效率提高，无论源位图的大小如何，无论过滤阵列是什么？

- 答案就是在程序运行的时候，**“即时地”**根据参数的具体值（位图的大小、深度，以及过滤阵列的大小和元素值）生成类似于手动解循环的代码即可。

.NET中间语言

- 接下来，我们将用C#生成中间语言代码。在C#程序里面，可以在内存里创建一个包含中间语言指令的静态方法并执行。

- 在整个过程中需要编写的全都是托管代码，而.NET的JIT会自动将它编译成目标机器代码。

- 因为最终需要生成类似于手动解循环的代码，那么需要把过滤阵列的所有信息、以及位图的大小和深度都“硬编码”在生成的静态方法里。为了生成这些代码当然需要一些额外的时间开销，但跟一副庞大的位图所需要的巨量操作比起来，这点开销是微乎其微的。

FilterMethodIL方法

```
void FilterMethodIL(byte[] src, byte[] dst, int stride, int bytesPerPixel)
{
    int cBytes = src.Length;
```

FilterMethodIL需要的参数和FilterMethodCS是一样的。

```
    DynamicMethod dynameth = new DynamicMethod("Go", typeof(void),
          new Type[] { typeof(byte[]), typeof(byte[]) }, GetType( ));
```

然后创造一个静态方法对象，它返回void，需要的参数是两个字节数组。

接下来就可以生成中间代码了，为此首先创建一个ILGenerator对象：

```
    ILGenerator generator = dynameth.GetILGenerator( );
```

后面我们主要都在用这个generator生成中间代码。

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11          for (int iFilter = 0; iFilter < cFilter; iFilter++)
12          {
13             int yFilter = iFilter / cyFilter;
14             int xFilter = iFilter % cxFilter;
16             int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19             if (iSrc >= 0 && iSrc < cBytes)
20             {
21                pixelsAccum += filter[iFilter] * src[iSrc];
22                filterAccum += filter[iFilter];
23             }
24          }
25          if (filterAccum != 0)
26             pixelsAccum /= filterAccum;
28          dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                      (byte)255 : (byte)pixelsAccum);
30    } }
```

FilterMethodIL方法

可以从定义局部变量开始：

```
generator.DeclareLocal(typeof(int));          // Index 0 = iDst
generator.DeclareLocal(typeof(double));       // Index 1 = pixelsAccum
generator.DeclareLocal(typeof(double));       // Index 2 = filterAccum
```

接下来对这些变量执行的指令是在一个虚拟的求值栈内进行的。所有可以执行的操作（目前总共有226种）都已经被定义成了OpCodes类上的静态只读字段了，这样我们就可以通过每种操作的“名字”来方便地调用它们了。

我们需要用到的所有指令见下图：

FilterMethodIL方法

名称	说明
Add	将两个值相加并将结果推送到计算堆栈上。
Mul	将两个值相乘并将结果推送到计算堆栈上。
Neg	对一个值执行求反并将结果推送到计算堆栈上。
Beq_S	如果两个值相等，则将控制转移到目标指令（短格式）。
Bge_S	如果第一个值大于或等于第二个值，则将控制转移到目标指令（短格式）。
Blt_S	如果第一个值小于第二个值，则将控制转移到目标指令（短格式）。
Br	无条件地将控制转移到目标指令。
Conv_R8	将位于计算堆栈顶部的值转换为 float64。
Conv_U1	将位于计算堆栈顶部的值转换为 unsigned int8，然后将其扩展为 int32。
Dup	复制计算堆栈上当前最顶端的值，然后将副本推送到计算堆栈上。
Ldarg_0	将索引为 0 的参数加载到计算堆栈上。
Ldc_I4	将所提供的 int32 类型的值作为 int32 推送到计算堆栈上。
Ldc_I4_0	将整数值 0 作为 int32 推送到计算堆栈上。
Ldc_R8	将所提供的 float64 类型的值作为 F (float) 类型推送到计算堆栈上。
Ldelem_U1	将位于指定数组索引处的 unsigned int8 类型的元素作为 int32 加载到计算堆栈的顶部。
Ldloc_0	将索引 0 处的局部变量加载到计算堆栈上。
Pop	移除当前位于计算堆栈顶部的值。
Stelem_I1	用计算堆栈上的 int8 值替换给定索引处的数组元素。
Stloc_0	从计算堆栈的顶部弹出当前值并将其存储到索引 0 处的局部变量列表中。

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

为了模拟**for** (**int** iDst = 0; iDst < cBytes; iDst++) {...}，首先需要：

```
generator.Emit(OpCodes.Ldc_I4_0);
generator.Emit(OpCodes.Stloc_0);
```

第一句的意思是生成一个四字节整数0并将它入栈；
第二句则是把栈顶的元素并赋给索引为0的局部变量(iDst)。

然后还需要判断循环终止条件，这里需要考虑分支和跳转问题。
中间语言只能使用类似goto的指令来模拟for和if。它为我们提供了一个方便的标签系统，只要在指令流中适当的地方插入标签，以后就可以跳转到这些标签处。

```
Label labelTop = generator.DefineLabel( );
generator.MarkLabel(labelTop);
```

这样我们就定义了一个标签，并把它插在了for循环的开头。

至于(iDst < cBytes)的判断和iDst++指令，把它放到循环的末尾执行。
接下来可以生成循环体了。

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

```
generator.Emit(OpCodes.Ldc_R8, 0.0); // 把实数0.0入栈
generator.Emit(OpCodes.Dup); // 把栈顶元素复制一个并入栈
generator.Emit(OpCodes.Stloc_1); // 把栈顶元素赋给变量1(pixelsAccum)
generator.Emit(OpCodes.Stloc_2); //把栈顶元素赋给变量2(filterAccum)
```

这几句就是把pixerlsAccum和filterAccum的值都设为0.0。

```
for (int iFilter = 0; iFilter < filter.Length; iFilter++)
{
   if (filter[iFilter] == 0)
      continue;
```

虽然这里直接用了C#的循环，但是在生成的中间代码里，会手动解循环，把过滤阵列的每个元素都硬编码在中间语言代码里。所以这里直接使用循环并不会影响最终代码的效率。

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

```
int xFilter = iFilter % cxFilter;
int yFilter = iFilter / cxFilter;
int offset = stride * (yFilter - cyFilter / 2)
                    + bytesPerPixel * (xFilter - cxFilter / 2);
```

这几个必要的值也可以直接在C#代码中计算，因为中间语言代码只需要这几个值，而不需要计算的过程。

下面需要访问src数组的元素。在中间语言里，为了访问一个数组元素，需要先把数组的引用入栈，然后把需要访问的元素下标入栈，然后执行Ldelem指令即可。

```
generator.Emit(OpCodes.Ldarg_0); // 把src数组的引用入栈
generator.Emit(OpCodes.Ldloc_0); // 把iDst入栈
generator.Emit(OpCodes.Ldc_I4, offset); // 把offset入栈
generator.Emit(OpCodes.Add); // iDst加offset(得到的是iSrc)
generator.Emit(OpCodes.Dup); // 把iSrc复制两遍
generator.Emit(OpCodes.Dup);
```

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

(iDst+offset)得到的iSrc可能会越界，需要进行判断。

```
Label labelLessThanZero = generator.DefineLabel( ); // (iSrc < 0)
Label labelGreaterThan = generator.DefineLabel( ); // (iSrc >= cBytes)
Label labelLoopBottom = generator.DefineLabel( ); // 暂时用不到

generator.Emit(OpCodes.Ldc_I4_0); // 将整数0入栈
generator.Emit(OpCodes.Blt_S, labelLessThanZero); // if(iSrc<0)则跳转

generator.Emit(OpCodes.Ldc_I4, cBytes); // 将cBytes入栈
generator.Emit(OpCodes.Bge_S, labelGreaterThan); // if(iSrc>=cBytes)跳转

generator.Emit(OpCodes.Ldelem_U1); // 读取元素src[iSrc]
generator.Emit(OpCodes.Conv_R8); // 把src[iSrc]转换成一个8字节浮点数
```

这样元素src[iSrc]的读取就完成了。

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

下面要把src[iSrc]和filter[iFilter]相乘。由于filter[iFilter]对我们来说是已知的，所以可以简化一些特殊的情况：

```
if (filter[iFilter] == 1) ; // 乘以1还得原来的数，因此不用做任何操作
else if (filter[iFilter] == -1)
    generator.Emit(OpCodes.Neg); // src[iSrc]直接取负，比乘以-1快一点
```

其他情况下，将src[iSrc]和filter[iFilter]相乘：

```
else
{
    generator.Emit(OpCodes.Ldc_R8, filter[iFilter]); // 把filter[iFilter]入栈
    generator.Emit(OpCodes.Mul); // 并把它乘以src[iSrc]
}
```

把相乘的结果加到pixelsAccum上：

```
generator.Emit(OpCodes.Ldloc_1); // pixelsAccum入栈
generator.Emit(OpCodes.Add); // 与上面的乘积相加
generator.Emit(OpCodes.Stloc_1); // 把结果存入pixelsAccum
```

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

```
generator.Emit(OpCodes.Ldc_R8, filter[iFilter]); // filter[iFilter]入栈
generator.Emit(OpCodes.Ldloc_2); // filterAccum入栈
generator.Emit(OpCodes.Add); // 把filter[iFilter]加到filterAccum上
generator.Emit(OpCodes.Stloc_2); // 把结果存入filterAccum
generator.Emit(OpCodes.Br, labelLoopBottom); // 跳转到循环结尾处
```

然后就该标记前面定义的几个标签labelLessThanZero, labelGreaterThan, labelLoopBottom了。

```
generator.MarkLabel(labelLessThanZero); // iSrc<0则栈内有三个多余元素
generator.Emit(OpCodes.Pop); // 释放栈顶元素
generator.MarkLabel(labelGreaterThan); // iSrc>=cBytes则有两个多余元素
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Pop);
generator.MarkLabel(labelLoopBottom); // iSrc未越界则没有多余元素
}
```

这样，对栈的清理工作就完成了。至此，我们已经对一个特定的目标像素计算出了pixelsAccum和filterAccum，离目标像素值已经很接近了。

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

```
generator.Emit(OpCodes.Ldarg_1); // dst数组
generator.Emit(OpCodes.Ldloc_0); // iDst下标
```

为了处理接下来的分支跳转，有必要定义一些标签：

```
Label labelSkipDivide = generator.DefineLabel( );
Label labelCopyQuotient = generator.DefineLabel( );
Label labelBlack = generator.DefineLabel( );
Label labelWhite = generator.DefineLabel( );
Label labelDone = generator.DefineLabel( );
```

检查filterAccum是否为0，并作相应的处理：

```
generator.Emit(OpCodes.Ldloc_1); // pixelsAccum
generator.Emit(OpCodes.Ldloc_2); // filterAccum
generator.Emit(OpCodes.Dup); // 复制filterAccum
generator.Emit(OpCodes.Ldc_R8, 0.0); // 将0.0入栈
generator.Emit(OpCodes.Beq_S, labelSkipDivide); // if(filterAccum==0)跳转
generator.Emit(OpCodes.Div); // 计算pixelsAccum/filterAccum
generator.Emit(OpCodes.Br_S, labelCopyQuotient); // 忽略下面两句
generator.MarkLabel(labelSkipDivide); // 如果没做除法：
generator.Emit(OpCodes.Pop);          // 则弹出多余的filterAccum
```

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

```
generator.MarkLabel(labelCopyQuotient);
generator.Emit(OpCodes.Dup); // 把新的pixelsAccum复制两遍
generator.Emit(OpCodes.Dup);

generator.Emit(OpCodes.Ldc_R8, 0.0); // 和0作比较
generator.Emit(OpCodes.Blt_S, labelBlack); // <0则跳转至labelBlack

generator.Emit(OpCodes.Ldc_R8, 255.0); // 和255作比较
generator.Emit(OpCodes.Bgt_S, labelWhite); // >255则跳转至labelWhite

其他情况下：（也即0 <= pixelsAccum <= 255）
generator.Emit(OpCodes.Conv_U1); // 把pixelsAccum转化为1字节整数
generator.Emit(OpCodes.Br_S, labelDone); // 并无条件跳转至labelDone
```

FilterMethodIL方法

对labelBlack的处理：

```
generator.MarkLabel(labelBlack);
generator.Emit(OpCodes.Pop); // 抛弃两个剩余元素
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Ldc_I4_S, 0); // 把pixelsAccum的值0入栈
generator.Emit(OpCodes.Br_S, labelDone); // 无条件跳转至labelDone
```

对labelWhite的处理与此相似：

```
generator.MarkLabel(labelWhite);
generator.Emit(OpCodes.Pop); // 抛弃一个剩余元素
generator.Emit(OpCodes.Ldc_I4_S, 255); // 把pixelsAccum的值255入栈
```

由于接下来就要标记labelDone标签了，所以在labelWhite最后就不用再跳转到labelDone了。

```
1  void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2  {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8        double pixelsAccum = 0;
9        double filterAccum = 0;
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13           int yFilter = iFilter / cyFilter;
14           int xFilter = iFilter % cxFilter;
16           int iSrc = iDst + stride * (yFilter - cyFilter/2) + bytesPerPixel * (xFilter - cxFilter/2);
18
19           if (iSrc >= 0 && iSrc < cBytes)
20           {
21              pixelsAccum += filter[iFilter] * src[iSrc];
22              filterAccum += filter[iFilter];
23           }
24        }
25        if (filterAccum != 0)
26           pixelsAccum /= filterAccum;
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                    (byte)255 : (byte)pixelsAccum);
30     } }
```

FilterMethodIL方法

```
generator.MarkLabel(labelDone);
generator.Emit(OpCodes.Stelem_I1); // 把指定像素值存到dst[iDst]内

最后要写的一点代码是循环终止时的iDst++和判断循环终止条件：
generator.Emit(OpCodes.Ldloc_0);    // iDst入栈
generator.Emit(OpCodes.Ldc_I4_1);   // 整数1入栈
generator.Emit(OpCodes.Add);        // iDst + 1
generator.Emit(OpCodes.Dup);        // 复制
generator.Emit(OpCodes.Stloc_0);    // 把(iDst + 1)存入iDst
generator.Emit(OpCodes.Ldc_I4, cBytes);  // cBytes入栈
generator.Emit(OpCodes.Blt, labelTop);   // 如果iDst < cBytes，继续循环
generator.Emit(OpCodes.Ret); // 返回指令，意味着这个函数的终结

所有的中间代码生成到此结束。最后，只需要调用这个静态方法即可：
dynameth.Invoke(this, new object[] { src, dst });
}
```

总结

- 特化的算法几乎总是比泛化的算法更快。本文展示的方法就是，在运行期基于参数值生成了一个特化算法。这样，既保证了算法能处理泛化的问题，又保证了它在真正执行时是特化的、高效的。

- 作者测试了FilterMethodIL的运行时间，发现它只需要FilterMethodCS的1/4的时间，有时甚至更快。

- 作者认为，虽然FilterMethodIL的代码从外表上看起来不太漂亮，但是它能把耗时降为原来的1/4，这只能用美丽来形容了。