所有的有限元程序最终都归结为解一个“稀疏”的方程。所谓稀疏，是指矩阵中每一行的非零元素个数满足：1）相对矩阵的总体尺寸是一个相对较小的量；2）即便网格进行加密，也限制在某个固定值以内，而不会随之增长。在这种情况下，存储整个矩阵的全部元素值是不经济的，为了节约内存，可以只存储非零元素的位置信息及对应的值。这就要求：对每一行，存储非零元素的列标号（我们称此为sparsity pattern），以及对应元素的值。

在deal.II中，sparsity pattern与实际sparse matrix是区分开的。因为往往几个不同的稀疏矩阵有着相同的sparsity pattern，比如用于时间推进格式的刚度矩阵与质量矩阵、或者特征值问题中的左右特征矩阵。

因而deal.II中matrix类是基于sparsity pattern类建立的。有两种主要的sparsity pattern类，如下：

**“Static” sparsity pattern**

deal.II中主要的稀疏矩阵类SparseMatrix，对每个矩阵元素只存储其值，而不存储这些元素所处的位置。为此，它会依赖于伴随它的一个sparsity pattern对象的信息（从而把元素的位置信息和值信息对应起来）。这个sparsity pattern对象必须属于SparsityPattern类。

SparsityPattern对象按两个阶段建立：首先，在一个“动态”阶段，分配非零元素的位置，预期矩阵在这些位置有非零元素；然后在“静态”阶段，这些非零的位置被“压缩”到Compressed Sparse Row(CSR)格式。在这之后，不会再加入新的非零位置。只有在压缩后sparsity pattern才能被附着到一个矩阵上去，因为后者需要前者高效的压缩数据格式。在动态阶段建立一个sparsity pattern常常利用DoFTools::make\_sparsity\_pattern()函数来实现。

这种两步建立sparsity pattern的方式的优点是，当它与一个矩阵配对被使用时，有着非常高效的数据格式。特别是，元素的位置信息以线性数组的方式进行存放，非常适合目前这种多级caches的CPU进行访问。因而，静态SparsityPattern类是唯一的deal.II的大多数SparseMatrix类能使用的类。

这种静态sparsity pattern的缺点是：其有效建立取决于对每行最多有多少元素的可靠估计。在实际构建过程中，比如在DoFTools::make\_sparsity\_pattern()函数中，最多只能分配如之前声明的那么多的元素。这就存在一个问题，因为很多时候我们无法准确预估每一行最多有几个元素。因而，一种常见的策略是先建立一个使用不那么高效的存储格式的用于过渡的sparsity pattern，之后再直接把它复制到静态的压缩格式中去。

**“Dynamic”或“compressed”sparsity pattern**

正如上面所述，往往很难对sparsity pattern中每一行元素个数的上限进行预估。因而，在糟糕的预估下试图分配一个regular的SparsityPattern需要巨大的内存，而这些存储几乎不会被用到，且在压缩后会被释放掉。

为了防止这点，deal.II含有一种“动态的”或者“压缩的”sparsity pattern叫做DynamicSparsityPattern，它会动态地根据添加当前元素需要多少内存来分配对应的内存。虽然这种方法节约了内存，但为了插入元素，需要使用不那么高效的存储格式，并且频繁的内存分配会显著增加计算时间。

这个类通常像下面这样使用：

DynamicSparsityPattern dsp (dof\_handler.n\_dofs());

//此时dsp只知道应该它应该有n\_dofs行，此外不含任何信息用以表示非零元素的位置，即此时dsp几乎为空。

DoFTools::make\_sparsity\_pattern (dof\_handler, dsp);

//计算基于dof\_handler产生的矩阵有哪些元素非零，并产生dsp对象代表了这些非零元素的位置。

constraints.condense (dsp); //必要时可使用

SparsityPattern final\_sparsity\_pattern;

final\_sparsity\_pattern.copy\_from (dsp);

//把生成的dsp拷贝给最终的sparsity\_pattern，两者应该几乎是一样的，除了在方阵情况时，会调换下元素的位置，使最终的sparsity\_pattern中每一行首元素为矩阵对角线元素所在列号。

**Dynamic block sparsity pattern**

类BlockDynamicSparsityPattern实现了用于构造block矩阵的动态sparsity pattern数组。

**SparsityPattern类**

这个类能存储矩阵中哪个元素非零（事实上也可能是零）并且为了哪个元素我们必须开辟内存以存储它的值。它使用compressed row storage(CSR)格式来存放数据，用作SparseMatrix类的基础。

**插叙：CSR格式**

CSR(compressed sparse row)或CRS(compressed row storage)格式，采用三个一维数组来代表一个矩阵。这三个数组分别存储非零元素的值、行的长度、以及列标号。这种格式允许快速的行访问，且适合于矩阵与向量相乘。

具体地讲：CSR格式利用三个(一维)数组A, IA, JA来存储一个稀疏的m×n矩阵M。用NNZ表示M中的非零元素个数。

1）数组A长度为NNZ，按照从左往右、从上往下的顺序保存了M中的所有非零元素（“row-major”的顺序）。

2）数组IA长度为m+1，它按如下递归方式定义：

IA[0]=0

IA[i]=IA[i-1]+(原始矩阵的i-1行的非零元素的个数)

因此，IA中的前m个元素存储了M中每一行的首个非零元素对应到数组A中的索引。则原始矩阵的第i行的值可从数组A的A[IA[i]]~A[IA[i+1]-1]元素获得。即，从某一行的起始元素直到下一行起始的前一个元素。

3）数组JA，保存了数组A中每个元素在矩阵M中对应的列的索引标号。因此其长度也是NNZ。

举个例子，如下矩阵：



对应：

A = [ 5 8 3 6 ]

IA= [ 0 0 2 3 4 ]

JA= [ 0 1 2 1]

因此，相对于原矩阵的16个元素，CSR格式只包含了13个元素。

IA把数组A中的元素分割为行：(5, 8) (3) (6)；

而JA则把这些元素按列排好：(5, 8, 0, 0) (0, 0, 3, 0) (0, 6, 0, 0)

注：只有在NNZ<(m(n-1)-1)/2的时候CSR格式才能起到节约内存的效果。

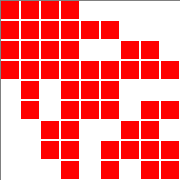
在SparsityPattern中的某个元素，对应于SparseMatrix对象中能存放非零元素的地址，是按row-by-row的方式存储的。在每一行中，按从左往右的顺序存放元素；这种规则的例外是如果存储的矩阵是方阵的话，则对角元素被作为每一行的第一个元素被存储，以便加快求Jacobi或SSOR预处理等操作。

例如：使用四个方形网格构成的mesh求解拉普拉斯方程，共有9个未知数（这9个自由度在网格上的标号顺序往往是随机的），形成的矩阵为9×9的方阵，若使用双线性有限元，则可得到其对应的sparsity\_pattern为：

|  |
| --- |
| [0,0,1,2,3] |
| [1,1,0,2,3,4,5] |
| [2,2,0,1,3,6,7] |
| [3,3,0,1,2,4,5,6,7,8] |
| [4,4,1,3,5] |
| [5,5,1,3,4,7,8] |
| [6,6,2,3,7] |
| [7,7,2,3,5,6,8] |
| [8,8,3,5,7] |

其实这就是类似上面CSR格式中的IA和JA数组给出的信息。

其中每行的首个数字表示行号，其后的数字表示非零元素所在的列号。形成的方阵应该形如（红色表示矩阵中的该元素非零）：



**SparseMatrix类**

这个类按照SparsityPattern表示的位置来存储矩阵元素的值。SparseMatrix中的元素的存储顺序与SparsityPattern类中元素的顺序保持一致。

**函数：**

**void DoFTools::make\_sparsity\_pattern**(const DoFHandlerType& dof\_handler,

SparsityPatternType& sparsity\_pattern,

const ConstraintMatrix& constraints=ConstraintMatrix(),

const bool keep\_constrained\_dofs=true,

const types::subdomain\_id subdomain\_id=numbers::invalid\_subdomain\_id

)

计算基于dof\_handler产生的矩阵有哪些元素非零，并产生sparsity pattern对象代表了这些非零元素的位置。

这个函数假定只有当一个有限元基函数的自由度是附着在网格的内部、面、边或格点上时，该基函数才是非零的。从而，在计算矩阵元素Aij时，(例如，i/j为全局索引)，只有当这些形函数对应的自由度所在的网格至少有一个是公共的，Aij才非零。因此，这个函数就是在所有网格上遍历，分辨出每个自由度的全局索引，并且假定只要某个矩阵元素是耦合了这些索引的(entries that couple any of these indices)，则该元素为非零。因为这个过程没有考虑之后将会求解的具体的方程，所以产生的sparsity pattern是对称的。

这个算法对每个网格上的全部形函数一视同仁，即，它简单地把一个网格上的某个自由度与该网格上的其他自由度耦合起来。这种做法与实际是一致的，且总是一种周全的假设。但是，如果你了解你的操作运算的结构，且它在某些形函数与某些试探函数间不产生耦合，则你可以使用下面将讲到的当前函数的某些变种。

上述方法依赖于假设：自由度间的耦合只有当形函数们至少在一个网格上有重合时才发生。对于大多数常见的使用协调有限元的公式而言，这种假设是对的。然而，对于间断有限元方法的公式，双线性泛函(bilinear form)包含了内边界上的项，这会在某个网格的形函数与其相邻网格的形函数间产生耦合。当前这个函数无法识别这种耦合，因而在sparsity pattern中也不会开辟相应的元素。这种问题可利用函数DoFTools::make\_flux\_sparsity\_pattern()来解决，它将考虑到相邻网格上自由度间的耦合问题。

**void DoFTools::make\_sparsity\_pattern**(const DoFHandlerType& dof\_handler,

const Table<2,Coupling>& coupling,

SparsityPatternType& sparsity\_pattern,

const ConstraintMatrix& constraints=ConstraintMatrix(),

const bool keep\_constrained\_dofs=true,

const types::subdomain\_id subdomain\_id=numbers::invalid\_subdomain\_id

)

这个函数是上面那个函数的一种变种，它提供了向量有限元会用到的功能，即更具体地指明在哪个方程中哪些变量有耦合。

例如，如果你想求解Stokes方程：



在二维情况下，使用稳定Q2/Q1混合元(使用FESystem类)，则你不希望在每个方程中所有自由度都耦合。具体地讲，在第一个方程中，只出现了ux和p；在第二个方程中，只出现uy和p；在第三个方程中，只出现ux和uy（注意，这里只是谈到解向量的分量及不同的方程，不涉及自由度或任何离散格式的事）我们以下面的“耦合”pattern来表示：



这些0表示，在使用表顺有限元方程进行离散之前，我们不会在矩阵中写入某个元素，使得，比如，这个元素会把压力试探函数与压力形函数耦合起来。

（未完待续。。。）

**void** **DoFTools::make\_flux\_sparsity\_pattern**(const DoFHandlerType& dof\_handler,

SparsityPatternType& sparsity\_pattern,

）

这个函数假设你使用的用来产生矩阵的bilinear form包含了网格间界面上的积分项（即包含“通量”）。

这个函数适用于间断有限元。在间断有限元里，某个网格上的全部或部分自由度会与别的网格上的自由度通过共同的界面产生耦合。