**继承关系是：**

DataOutInterface🡨DataOut\_DoFData🡨DataOut

**DataOutBase命名空间**

这是一个命名空间，其下有各种数据结构和模板函数。它是所有类的基础。其下的模板类和模板函数被DataOutInterface所用到。

输出数据被视为一组patches，并以期望的格式被写入输出流。对于每种格式，这个类都有一个函数write\_format来写该输出。

**输出数据的结构**

数据不是按照deal.II的网格结构来写的。而是依赖于派生类(如DataOut、DataOutFaces等)创建的一组patches。每个Patch代表了网格的一个逻辑单元，可能为了代表高阶多项式会被细分多次。为此，一个patch含有一个dim维的规则网格，其在每个方向上有相同数量的网格点。最简单的情况是，它含有一个单一网格元的角点。对这个局部网格的每个点，Patch含有任意数量的数据值。

通过提供这样一个接口给不同的输出格式，可以简便地把这个类推广到其他新的格式，而不依赖于如具体的网格或操作数据向量这些东西。这些东西留给派生类来提供。

在每个patch中，按通常的词典顺序组织数据，x优先，然后是y和z。格点和网格都按这个顺序来存储。3D中的每个网格按前面为xz面来存储。

**1）Patches**

Grids可以看作cells的一个集合。如果你想在这样一个grid上写出数据，你可以通过一次写一个cell的方法来实现。这个类的函数因此每次都要在一个cell上提取数据。每个cell的数据通常含有这个cell的一系列vertices以及对应的一系列数据值(如这些点上的解、误差值等)

但有些时候，这个对应了cell的接口限制性太强了。例如，你可能使用了更高阶有限元，所以仅仅输出这些vertices处的值是不够的。为此，我们不仅仅提供输出vertices处的值的功能，而且还按每个cell上是一个tensor product grid的方式来组织数据。给每个patch分配一个参数n\_subdivisions，表示该cell被划分几次用于输出；例如，n\_subdivisions==1表示cell没有被划分，n\_subdivisions==2会产生有3x3个点的网格(2维)或有3x3x3个点的网格(3维)。这些点的实际位置会依据这个patch给定的vertices的位置通过多线性变换来计算得到。对于边界上的cells，可能会用到一个映射来计算内部点的位置。在那种情况下，坐标值被存放在Patch内部。

**2）generalized patches**

一般来说，上面的那种patches可能限制性太强。例如，有时候你可能只想画出一个三维域的外部界面。那么需要被画出的对象是三维空间中的一个二维物体。对于这种情况，Patch类需要两个模板参数，第一个称为dim的代表了该对象的维数(在上述例子中就是2)，而第二个称为spacedim的参数表示了嵌入空间的维数(即3)。

**DataOutBaseInterface**

这个命名空间下的成员通常不会被用户直接调用。使用到它声明的函数的类通常是由DataOutInterface派生而来。

这个类的接口基本上包含了一些数据类型的声明：用以描述一个patch，以及一族函数：接收一系列patches作参数并按某种格式把它们输出到流上。由派生类负责提供这一系列patches。

**Querying interface询问接口**

这个类也提供用于询问这个类支持哪些输出格式的一些函数(parse\_output\_format(), get\_output\_format\_names(), default\_suffix())。使用这些函数，可知道当前库允许使用哪些格式。

**Output parameter输出参数**

所有函数接收一个类型为XFlags的参数，其中x是输出格式的名称。

注意，通常用于科学可视化的输出格式没有或仅有很少的几个参数，因为实际的可视化由使用的可视化程序来决定，而这个类产生的文件基本上只存储原始数据。

**Writing backends**

略

**DataOutBase::Patch**

template<int dim, int spacedim = dim>

struct DataOutBase::Patch< dim, spacedim >

这是描述一个在dim维空间中的数据patch的数据结构。

一个patch包含了下述数据：

* 角点the corner vertices
* 数字n\_subdivision，表示Patch在每个空间方向上有几个cells
* 附着在每个vertex上的data，(按照字典顺序排列)
* neighbors的信息。

**DataOutInterface**

这个类是对应DataOutBase命名空间下的函数的接口。所谓接口其实说这个类提供了函数的声明，在类中并没有实现的代码，其具体实现是在DataOutBase空间下就已经实现了的。

这个类被看做实际生成输出数据的类的基类。它有两个纯虚函数：get\_patches()和get\_dataset\_names()用于产生实际需要的数据。这些是唯一需要被派生类重载的函数。

这个类的目的是双重的：支持保存flags用以控制输出格式，以及一些方法，用以在运行时决定输出数据的输出格式、flags及其他事。

例如：

DataOutBase，命名空间下有模板函数为：

template<int dim, int spacedim>

void DataOutBase::write\_dx ( const std::vector< Patch< dim, spacedim > > & patches,

const std::vector< std::string > & data\_names,

const std::vector< std::tuple< unsigned int, unsigned int, std::string > > & vector\_data\_ranges,

const DXFlags & flags,

std::ostream & out

) {...}

对应在DataOutInterface类下有成员函数：

template<int dim, int spacedim>

void DataOutInterface< dim, spacedim >::write\_dx ( std::ostream & out ) const

{

DataOutBase::write\_dx (get\_patches(),

get\_dataset\_names(),

get\_vector\_data\_ranges(),

dx\_flags,

out);

}

其中get\_patches()、get\_dataset\_names()、get\_vector\_data\_ranges()是该类的另外几个成员函数，全是纯虚函数，dx\_flags是DataOutBase下定义的类DataOutBase::DXFlags的对象。

**Output flags**

在这个类中对待flags的方式与在GridOut类中的很类似。关于why’s和how’s的具体信息可参见那里的文档。

基本上，这个类对于每个DataOutBase支持的输出格式保存了一组flags。这些会在某个write\_\*函数被调用时被使用。

...

**DataOut\_DoFData**

template<typename DoFHandlerType, int patch\_dim, int patch\_space\_dim = patch\_dim>

class DataOut\_DoFData< DoFHandlerType, patch\_dim, patch\_space\_dim >

这是个抽象类，利用基类通过网格上的数据向量，提供了用于生成patches的功能。它允许把一个或多个指针附着到一个DoFHandler及其联系的node和cell数据上。

**用户可见的接口**

这个类的用户接口允许用户以两种方式指定数据。一种是使得一个DoFHandler对象被这个类访问，并添加对应于这个DoFHandler或grid cells的数据向量，这些数据向量在之后会以某种格式被写到文件中。第二种方式是传输一个DoFHandler对象连同数据向量。这使得我们可以以一种整洁的方式从不同的DoFHandlers设置数据。举例加以说明：

...

... // compute solution, which contains nodal values

...

... // compute error\_estimator, which contains one value per cell

std::vector<std::string> solution\_names;

solution\_names.emplace\_back ("x-displacement");

solution\_names.emplace\_back ("y-displacement");

DataOut<dim> data\_out;

data\_out.attach\_dof\_handler (dof\_handler);

data\_out.add\_data\_vector (solution, solution\_names);

data\_out.add\_data\_vector (error\_estimator, "estimated\_error");

data\_out.build\_patches ();

ofstream output\_file ("output");

data\_out.write\_xxx (output\_file);

data\_out.clear ();

分析：

attach\_dof\_handler()告诉这个类之后的所有操作都是基于这个DoFHandler对象及其对应的triangulation的，再添加solution vector和error estimator。注意它们有不同的维度，因为解向量是节点向量，因此包含两个分量(x-位移和y-位移)，而error estimator可能是含cell data的向量。

这个类并不会复制通过函数add\_data\_vector()传给它的数据向量，而是只保存引用。

在添加了所有数据向量后，你需要调用一个生成patches(即一些中间数据表示)的函数，它从存储的数据生成输出数据。

最后，把数据以某种格式write()到一个文件。

在当前的类中没有真正实现生成patches的函数，任意真正的函数实现应该在派生类，如DataOut中实现。

注意，在这个类的基类DataOutInterface中提供了一些函数，可帮助编写run-time决定输出格式的代码。

**派生类信息**

这个类缺乏通过存储的数据和自由度信息生成patches(用于数据输出)的方法。因为这个任务通常是依赖于具体问题的，故留给派生类来完成（填充这个类中的patches数组）。

**DataOut**

template<int dim, typename DoFHandlerType = DoFHandler<dim>>

class DataOut< dim, DoFHandlerType >

这个类是DataOut\_DoFData类中提出的函数功能的具体实现。它提供了build\_patches()函数用于生成被以图形化格式输出的数据。它的大多数功能及使用例子都在其基类中有说明。

这个类唯一提供的东西是函数build\_patches()，它在由attach\_dof\_handler()函数保存的triangulation上循环全部cells，把这些cells上的数据转换为实际的patches对象，这些patches在之后会被输出。

主要函数分析：

template<int dim, typename DoFHandlerType >

void DataOut< dim, DoFHandlerType >::build\_patches ( const unsigned int n\_subdivisions = 0 )

这是这个类的核心函数，它建立一系列patches，用以被基类的函数写出。一个patch，实质上是一种代表了每个cell上的数据的中间物。