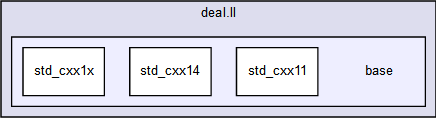
****

**Deal.II/base/subscriptor.h**

Subscribe翻译为订购、署名、预订。对象调用subscribe()函数可以认定某个订购者（使用该对象的称为subscriber，这样对象就“认识”谁在使用它了）。

当某个对象作为引用（传给构造函数）被存储后，则应满足：在依赖它的对象被删除前，不得删除它。

程序上可这样实现：使对象是由这个Subscriptor类派生出的，且让用户subscribe()到这个对象（引用了这个对象的称为subscriber），则此时存在订阅关系subscriptions。当它作为被引用对象，调用从Subscriptor继承的析构函数，由于此时对象身上还标记有subscription，则会报错。

void Subscriptor::subscribe(const char \*id) const

{

#ifdef DEBUG

if (object\_info == nullptr)

object\_info = &typeid(\*this);

++counter;

// This feature is disabled when we compile with threads: see the

// documentation of this class.

# ifndef DEAL\_II\_WITH\_THREADS

const char \*const name = (id != 0) ? id : unknown\_subscriber;

map\_iterator it = counter\_map.find(name);

if (it == counter\_map.end())

counter\_map.insert(map\_value\_type(name, 1U));

else

it->second++;

# else

(void)id;

# endif

#else

(void)id;

#endif

}

**Deal.II/base/iterator\_range.h**

这个类用来表示一个迭代器集合，目的是Triangulation和DoFHandler类通过函数Triangulation::cells()使用这个类时，能返回迭代器的范围。例如：想在每个激活网格上设user flag：

Triangulation<dim> triangulation;

...

for (auto cell : triangulation.active\_cell\_iterators())

cell->set\_user\_flag();

也就是说，函数triangulation.active\_cell\_iterators()返回的东西可认为是一个迭代器集合。

**template<typename Iterator>  
class IteratorRange< Iterator >**

**设计这个类的动机：**

C++11中新增了一个“range-based for”循环语法：

Container c；

for（auto v：c）

statement；

这里c是一个容器或集合，它等价于下述循环：

Container c；

for（auto tmp=c.begin(); tmp!=c.end()； ++tmp）

{

auto v= \*tmp；

statement；

}

也就是说在C++11的用法中，实际上v是作为某个临时迭代器的解引用，它代表了集合中的元素。

但我们在deal.II中并不想这样，我们仍想保留循环变量是个迭代器，而不是一个值。也就是说比如调用函数Triangulation::active\_cell\_iterators()时，返回的是一个迭代器集合{begin，begin+1，...，end-1}。

**Deal.II/base/tensor.h**

有任意rank的张量。提供了索引操作，但多数函数功能最终都传递到底层的rank-1操作上去了，或者放在别的模板函数中去了，比如contract。

用rank2的张量比使用matrix好，因为matrix的维数是在运行时确定的，不如tensor经济。

**template<int rank\_, int dim, typename Number>**

**class Tensor< rank\_, dim, Number >**

模板参数：

rank\_：代表张量秩的整数。rank-0就是标量，rank-1就是有dim个分量的向量，rank-2就是有dim-by-dim个分量的矩阵，等等。

dim：代表张量在dim维空间中起作用。等于确定一个点和一个向量需要的坐标数。

Number：表示张量以什么样的数据类型存储。一般是double。

定义了一些与tensor操作有关的模板函数如：

1）Tensor对象的输出函数（重载<<）

2）Tensor对象的空间操作函数（加减乘除）

3）Tensor对象的contraction运算和outer product运算（张量运算）

4）一些特殊运算（如rank2的求行列式、迹、转置、逆等）

**Deal.II/base/point.h**

由Tensor<1, dim, Number>继承而来。代表了dim维笛卡尔空间中的一点。替代了类似double f(const double x, const double y)这样的函数，可以用double f(const Point<dim> &p)把程序写成通用的形式。

**template<int dim, typename Number = double>**

**class Point< dim, Number >**

Point<dim>和Tensor<1, dim>的区别？

前者继承了后者的特性，新增的特性并不多，故这两者常常可以混用。新增的函数如计算空间中两个点的距离的函数distance()。

但两者从语义上是有差别的。当表示两点间相差的那个向量时，使用Tensor<1, dim>，若单纯表某个点，就用Point<dim>

**Deal.II/base/geometry\_info.h**

这个类提供了参考网格的拓扑结构的全部信息，它被广泛地用在Triangulation对象的几何描述中。它是deal.II库的核心，提供了参考网格上的顶点、线、面的排列顺序数。提供了以dimension independent的方式来写代码的可能。比如在二维情况下，写一个在所有顶点上的循环，不用写成：0<=v<4；而是写成：0<=v<GeometryInfo<dim>::vertices\_per\_cell。它最常用的内容是静态数据变量如：vertices\_per\_cell, faces\_per\_cell等。但这个类也提供了一些抽象的信息如面的方向等。

**template<int dim>**

**struct GeometryInfo< dim >**

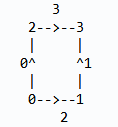
2D下的实现：

1）顶点以字典形式编号（lexicographic ordering）

2）面（2D中的边）以-x, x, -y, y的顺序编号

3）线的方向以从点0到点1的方向表示，且始终是坐标方向之一

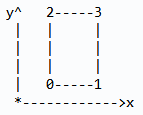
4）位于3d中的面的边是有顺序的，从而用右手定则定义出一个面的法向。面0,2的法向是指向单元内部；面1,3的法向指向单元外部

如图：

定义子线与母线的方向是一样的，第一个子线是位于vertex(0)那端的那条。即：

line->child(0)->vertex(0)==line->vertex(0) and line->child(1)->vertex(1)==line->vertex(1)

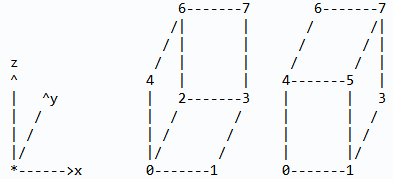
为参考网格定义如图坐标系：



这里，顶点0是原点，顶点1坐标为(1, 0)，顶点2为(0, 1)，顶点3为(1, 1)。在运行的时候可利用函数GeometryInfo<dim>::unit\_cell\_vertex()来询问这些信息。

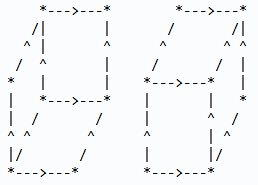
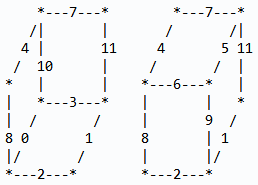
3D下的实现：

参考网格顶点及坐标系如图：



注意左图表示左、下、后面。

线如图：



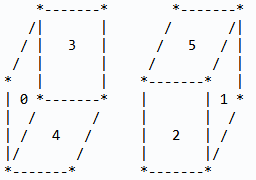
注：first the lines of face (z=0) in 2d line ordering,

then the lines of face (z=1) in 2d line ordering,

finally the lines in z direction in lexicographic ordering

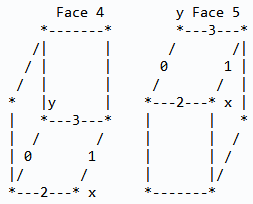
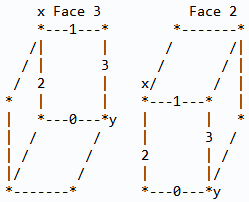
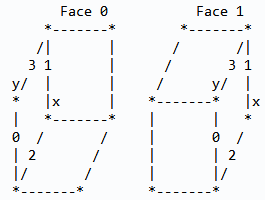
线的方向也和2D情况一致，都是坐标轴正方向。

面如图：



按-x, x, -y, y, -z, z的顺序排号。

每个面有局部坐标系，定义出法向。面0,2,4的法向指向单元内部；面1,3,5的法向指向外部。可看出对边的方向是一致的，这种方向的约定称为标准方向：



面上的线编号（0,1,2,3）对应了单元的线编号：

Face 0: lines 8, 10, 0, 4;

Face 1: lines 9, 11, 1, 5;

Face 2: lines 2, 6, 8, 9;

Face 3: lines 3, 7, 10, 11;

Face 4: lines 0, 1, 2, 3;

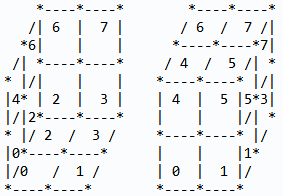
Face 5: lines 4, 5, 6, 7;

可以用函数GeometryInfo<3>::face\_to\_cell\_lines()询问这些编号。

可以用GeometryInfo<dim>::unit\_normal\_orientation询问面的法向。

子网格：

各向同性划分，8个子网格按照它们临近的顶点序号排号：



下列子网格与对应面相邻，其顺序与面的法向有关：

Face 0: children 0, 2, 4, 6;

Face 1: children 1, 3, 5, 7;

Face 2: children 0, 4, 1, 5;

Face 3: children 2, 6, 3, 7;

Face 4: children 0, 1, 2, 3;

Face 5: children 4, 5, 6, 7.

可用函数GeometryInfo<3>::child\_cell\_on\_face()来询问这些编号。

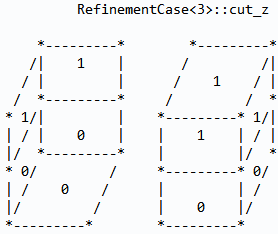
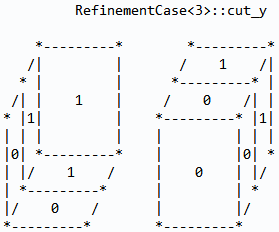
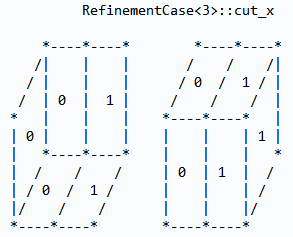
也可以用GeometryInfo<3>::face\_to\_cell\_vertices()来询问编号。

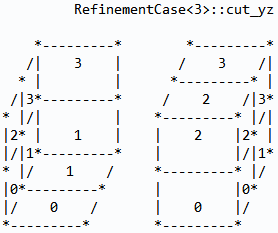
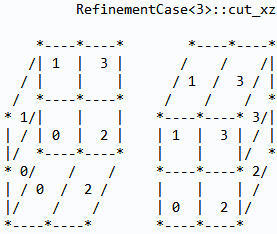
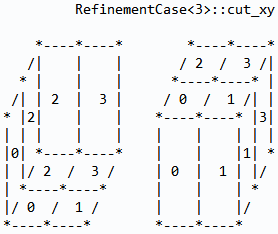
在计算有悬挂点的面上的jump terms的时候，常需要询问哪个子网格在哪个面上的哪个位置这样的信息，使用FESubfaceValues对象。处在某个网格上，你可以使用：

cell->neighbor\_child\_on\_subface(face\_no,subface\_no)

来得到正确的信息。

各向异性划分，子网格编号采用如下约定：





这些信息也可用函数GeometryInfo<3>::child\_cell\_on\_face来得到。

成员函数：

1）int GeometryInfo<dim>::n\_children(const RefinementCase<dim>& refinement\_case)

返回网格的子网格数（用ref\_case进行划分）

2）int GeometryInfo<dim>::n\_subfaces(const internal::SubfaceCase<dim>& subface\_case)

返回一个面的子面数（用internal::SubfaceCase face\_ref\_case进行划分）

3）static RefinementCase<dim-1> GeometryInfo<dim>::

face\_refinement\_case(const RefinementCase<dim>& cell\_refinement\_case,

const int face\_no,

const bool face\_orientation=true,

const bool face\_flip=false,

const bool face\_rotation=false)

给一个用RefinementCase cell\_refinement\_case划分的网格，返回第face\_no个面上的SubfaceCase。

4）static RefinementCase<1> GeometryInfo<dim>::

line\_refinement\_case(const RefinementCase<dim>& cell\_refinement\_case,

const int line\_no)

给定一个用RefinementCase cell\_refinement\_case划分的网格，返回第line\_no个面上的RefinementCase。

**template<int dim>  
class RefinementCase< dim >**

RefinementCase<dim>这个类从RefinementPossibilities<dim>继承而来，存储了某个dim维对象可能的划分方式（各向同性、异性），比如位于三维空间的二维四边形，有哪几种划分方式。