#include <deal.II/grid/tria.h>

Public types：

enum MeshSmoothing{}

cell\_iterator

active\_cell\_iterator

public function包括与下列方向有关的：

**mesh refinement**

**history of a triangulation**

**user data**

**cell iterator functions**

**cell iterator functions returning ranges of iterators**

**face iterator functions**

**vertex iterator functions**

**information about the triangulation**

**internal information about the number of objects**

static public attribute：

straight\_boundary

dimension

space\_dimension

private types：

iteratorSelector

private member functions与下面有关:

line iterator functions for internal use

quad iterator functions for internal use

cell iterator functions for internal use：

如raw\_cell\_iterator、raw\_face\_iterator、raw\_vertex\_iterator等（line/quad/hex）

Hex iterator functions for internal use：

std::vector<::internal::Triangulation::TriaLevel< dim > \* > levels

std\_cxx11::unique\_ptr<::internal::Triangulation::TriaFaces< dim > > faces

std::vector< Point< spacedim > > vertices

std::vector< bool > vertices\_used

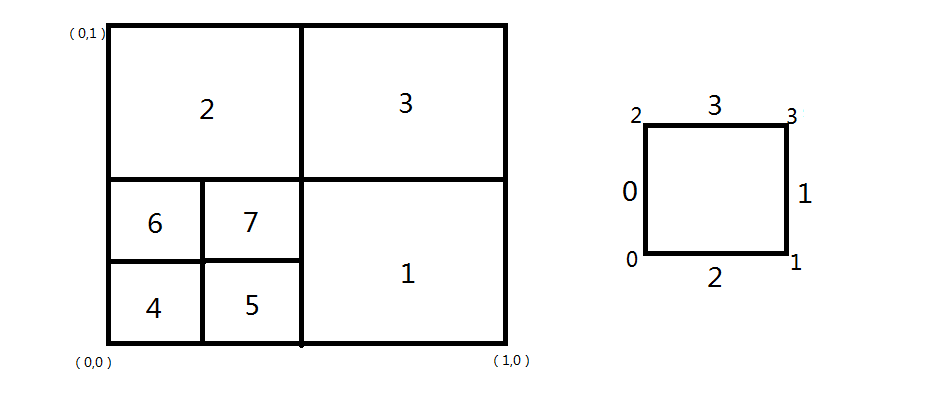
std::map< types::manifold\_id, SmartPointer< const Manifold< dim, spacedim >, Triangulation< dim, spacedim > > > manifold

**template<int dim, int spacedim = dim>**

**class Triangulation< dim, spacedim >**

triangulation是在spacedim维空间中构成一个dim维的流形（由有层级结构的元素构成）。

**先用一个例子说明triangulation上cell iterator的顺序、face iterator的顺序：**



如上图左侧的这样一个网格，是由一个(0,1)×(0,1)的单网格先全局加密一次，再调用triangulation.begin\_active()->set\_refine\_flag()标记第一个迭代器所在的网格后局部加密一次得到的。在这个triangulation上使用active\_cell\_iterator进行迭代器循环，其索引顺序标注在了图上。

若想在某个active\_cell上访问其某条边，则可使用cell->face(face\_no)返回指向该边的一个face迭代器face\_iterator。对于cell上，face\_no的索引顺序标注在上图右侧单网格上了。注意：对于没有细化过的face，从其两侧的cell上调用此函数返回的是同一个face\_iterator，例如：在上图中的网格2上调用cell->face(1)与在网格3上调用cell->face(0)返回的是同一个face迭代器。但对于细化过的face，则两侧网格各有其对应的face\_iterator。例如：在网格7上调用cell->face(1)与在网格1上调用cell->face(0)返回的是两个不同的face\_iterator。

因此，如果我们想要定义一个从face\_iterator到double的map，用于存储所有face上的积分值，如：

std::map<typename Triangulation<2>::face\_iterator, double> face\_map;

for(active\_cell\_iterator cell=triangulation.begin\_active();cell!=triangulation.end();++cell,++cell\_no){

for(unsigned int face\_no=0;face\_no<GeometryInfo<2>::faces\_per\_cell;++face\_no){

face\_map[cell->face(face\_no)] = 0;

}

}

对于上图所示的网格，这个map的大小是22。

**Structure and iterators**

一个triangulation对象的数据结构相当复杂，如果想直接在它上面进行操作是很不方便的，因为数据分布在相当广泛的数组和其他地方上。但是，我们有足够的办法在不需要知道这些数据结构间的准确关系的情况下进行操作。Deal.II使用了local typedefs类来尽可能简化操作。

Triangulation使用了iterators来进行在网格上的循环。它们的名字是由iterators类typedef而来，所以是local types to this class。包括：

1）cell\_iterator：在triangulation用到的所有cells上进行循环

2）active\_cell\_iterator：在active cells上进行循环

如果dim==1，这些iterators是这样typedef的：

typedef line\_iterator cell\_iterator;

typedef active\_line\_iterator active\_cell\_iterator;

如果dim==2，多出face iterator来是这样定义的：

typedef quad\_iterator cell\_iterator;

typedef active\_quad\_iterator active\_cell\_iterator;

typedef line\_iterator face\_iterator;

typedef active\_line\_iterator active\_face\_iterator;

对1D来说，一个cell的face指的是一个点；对2D来说，一个cell的face指的是一条线line。

下面举例说明了这些iterator的用法：

1）计算某个level上的cells数量：

templat<int dim, int spacedim>

int Triangulation<dim, spacedim>::n\_cells (cons tint level) const

{

cell\_iterator cell = begin (level), endc = end (level);

int n = 0;

for(; cell!=endc; ++cell)

++n;

return n;

}

2)或者使用另一种方法，使用std::distance

template <int dim>

int Triangulation<dim>::n\_cells (cons int level) const

{

int n = 0;

distance (begin(level), (level == levels.size()-1 ? cell\_iterator(end()) : begin(level+1)), n);

return n;

}

3）细化triangulation的全部cells

template<int dim>

void Triangulation<dim>::refine\_global()

{

active\_cell\_iterator cell = begin\_active(), endc = end();

for(; cell!=endc; ++cell)

cell->set\_refine\_flag();

execute\_coarsening\_and\_refinement();

}

**Usage**

triangulation的使用主要是通过iterator来使用的。下面这个例子很好地说明了怎么用：

int main()

{

Triangulation<2> tria;

//读入一个coarse grid文件

//我们想要记录细化的历史

ofstream history(“mesh.history“);

//细化第一个cell

tria.begin\_active()->set\_refine\_flag();

tria.save\_refine\_flags(history);

tria.execute\_coarsening\_and\_refinement();

//在coarsest level上细化第一个active cell

tria.begin\_active()->set\_refine\_flag();

tria.save\_refine\_flags(history);

tria.execute\_coarsening\_and\_refinement();

Triangulation<2>::active\_cell\_iterator cell;

for(int i=0; i<17; ++i)

{

//细化当前倒数第二个cell17次

cell = tria.last\_active(tria.n\_levels()-1);

--cell;

cell->set\_refine\_flag();

tria.save\_refine\_flags(history);

tria. execute\_coarsening\_and\_refinement();

};

//输出网格

ofstream out(“grid.1“);

GridOut::write\_gnuplot(tria, out);

}

**Creating a triangulation**

存在几种创建triangulation的方式：

1）用内置的GridGenerator类来生成网格并填充triangulation

2）读入triangulation：使用GridIn类

3）显式地创建triangulation：提供一个vertices的list和一个cells的list。每个这样的cell都由一个vector构成，vector中存放了indices，indices用于检索cell的顶点；用这些indices去vertex list中就能找到对应的顶点。详见GridIn<dim>::read\_\*函数。调用函数creat\_triangulation()。

想从仅仅保存了vertex信息的cells上创建出层级关系是比较复杂的任务。例如在2D中，我们必须创建出vertices之间的lines，还必须创建出邻居信息。故读入的网格信息不应该太大了，读入细化的网格就不经济。

你需要保证cells有正确的orientation。为了保证这点，在输入的存放cell list的vector中，每个cell的vertex indices必须有正确定义的顺序。详见GeometryInfo<dim>文档。在2D,3D情况下很难验证这种顺序到底对没对，如果错了，最后的矩阵项可能符号出错。

为了满足正确的vertex numbering，还有一些细节的条件需要满足。不仅读入UCD数据的时候，在传给create\_triangulation()的数据中也需要满足。详见GridIn文档，尤其是GridReordering类，解释了与这有关的很多问题和算法。

4）复制一个triangulation：当计算非定常问题或自适应问题时，常需要复制网格，可利用copy\_triangulation函数实现。

**Refinement and coarsening of a triangulation**

细化网格有几种方式。最低级的方式是利用迭代器在active\_cell上标记需要细化的网格。再执行execute\_coarsening\_and\_refinement()函数执行细化。这个函数首先调用prepare\_coarsening\_and\_refinement函数来regularize网格：因为两个cells间的face只能被细化一次（也就是说相邻两个网格的细化程度不能相差超过两级），所以除了人工标记的那些需要被细化外，还需要标记一些额外的网格来使这种细化的过渡足够平缓。这会增加最后的网格数但得到的网格是足够光顺的，当然，更重要的目的是使得数据结构和算法不要太过复杂。

粗化网格的方法也是类似的，先标记，再粗化。

**Smoothing of a triangulation**

可在triangulation的构造函数中设置参数指定是否需要在每次执行函数execute\_coarsening\_and\_refinement()的时候进行网格smooth操作。默认情况下是不做的，因为这会导致产生多余的cells。

**Material and boundary information**

每个cell，face或edge保存了对象的材料属性参数或边界条件参数。

可在创建triangulation的时候设置material\_id（通过CellData数据结构），或之后使用cell iterator进行设置。相关用法参见step-28。GridGenerator命名空间下的函数通常把material ID设为0.

**History of a triangulation**

可以从网格的细化历史中重建一个网格。通过save\_refine\_flags和load\_refine\_flags函数来实现保存历史和载入历史的操作。代码类似这样：

//打开output文件

ofstream history(“mesh.history“);

//细化10次

for(int step=0; step<10; ++step)

{

...;

//根据某些标准标记网格

...;

tria.save\_refine\_flags(history);

tria.execute\_coarsening\_and\_refinement();

}

//打开input文件

ifstream history(“mesh.history“);

//细化十次

for(int step=0; step<10; ++step)

{

tria.load\_refine\_flags(history);

tria.execute\_coarsening\_and\_refinement();

}

**User flags and data**

**Boundary approximation**

**Getting notice when a triangulation changes**

**Serializing (loading or storing) triangulations**

**Technical details:**

**Algorithms for mesh regularization and smoothing upon refinement**