**Iterators on mesh-like containers**

注：第一部分着重讲**迭代器**iterator

Deal.II中的类Triangulation、DoFHandler、hp::DoFHandler都可以理解为mesh。它们都定义了一组迭代器，用于在所有网格（包括面、边、点等）上循环。这些迭代器基本上是继承自TriaIterator类的。TriaIterator类模板为：

TriaIterator<Accessor>

从概念上，这种类代表了某种类似指针的东西，它指向Access类的对象。一般来说，用户用到的迭代器是如：Triangulation::cell\_iterator的（是typedefs）。

我们下面先介绍iterators，再介绍accessors：

与C++中的迭代器不同，deal.II中的迭代器是双向的。

**Iterators as pointers into sets of objects**

Deal.II中组成网格的对象包括lines，quads，和hexes，它们用Accessor类来表示。Accessor类作为模板参数传给iterator。

Triangulation使用规则树的计算概念来存储数据。可以这样理解：把稀疏网格的cells当做根；然后，如果某个cell被加密，将有2dim个孩子。也就是说，稀疏网格的每个cell可以作为二叉树（1D）的根，或者四叉树（2D）的根，或者八叉树（3D）的根。从所有根开始延伸出的所有分支构成森林，就代表了一个triangulation，终点的（没有再分叉）节点就是active cells。

一个triangulation同时有lines的森林（分两个叉），有quads的森林（分四个叉），有hexes的森林（分八个叉）。根据维数的不同，这些对象又被称为cells或者faces。

Iterator则是循环访问森林中的元素，active iterator是只访问终结点元素。Deal.II的迭代器采用广度优先的搜索方式，也就是说先访问coarse mesh的全部元素（cells，faces，etc）；然后移动到近亲子元素那一层进行访问；然后移动到孙子那一层。

**Different kinds of iterators**

迭代器有两个性质：它们指向什么（即模板参数Accessor的类型），它们循环的集合的准确定义。一般来说，iterator总是这样声明：

KindIterator<Accessor>

这里，kind表示accessor的性质。比如：

Iterator<Accessor>

在构成网格的所有对象上循环，而

ActiveIterator<Accessor>

跳过了所有有孩子的对象，只在最终的active cell上循环。由accessor类对象告诉iterator它们是不是active的。

判断一个对象是不是active的可定义为“predicate”，即一种性质，表明要么是真要么是假。Filtered iterators可用于进一步限定循环的范围。比如可在active cells上作人工标记，再用过滤迭代器在这些有标记的网格上进行循环。这通常可通过FilteredIterator<BaseIterator>来实现，其中BaseIterator是上述讨论的常用迭代器之一。在命名空间IteratorFilters下可找到一些已经实现的predicate。

**Iterating over objects**

只要是同类的迭代器，作用于同样的几何对象上，其迭代的顺序是相同的。比如使用DoFHandler和Triangulation在同样的网格上迭代，或者DoFHandler使用不同的有限元在同一网格上迭代，其顺序是完全一样的。

**Accessors**

迭代器就如同指针，但他们的魔力在于它们指向的对象，即Accessor。对C++的指针而言，他们指向某种实实在在的，存储了数据的对象。但deal.II中，iterator在解引用的时候，并不是返回对某个实在对象的引用，而是返回一个特殊的叫做accessor的对象，该对象知道如何获取网格的数据（accessor的成员函数用来直接操作更底层的数据结构）。也就是说，iterator指向的对象自身并不存储顶点位置、邻居类型等数据。但是，该对象知道如何从triangulation建立的数组、表和列表中梳理出这种信息。

访问网格相关的数据总是通过Accessor来完成的，表达式：i->xxx()可获得这个Accessor的全部属性。

例如，你可以询问一个iterator得到性质：

cell->vertex(1); //等价于(\*cell).vertex(1)，即cellAccessor::vertex(1)

line->child(0);

hex->face(3);

cell->at\_boundary();

face->boundary\_id();

这些语句其实是对应了成员函数Accessor::vertex()，Accessor::child()等。这些到底是怎么实现的，以及使用了什么样的数据结构，并不是使用deal.II的开发者需要关心的。通过对应用用户隐藏这些数据结构，我们能以一种更有效的方式来存储数据，而不需要以一种易懂但比较低效的方式。（译者注：这里体现了deal.II的哲学：有些底层结构确实是比较复杂难懂的，但这是为了提高效率，开发者不需要太深究这些底层的东西）

-----------------------------------------------------------------------------------------------------------------------------

注：在triangulation中cell\_iterator是这样定义的：

**typedef TriaIterator<CellAccessor<dim,spacedim>> Triangulation<dim, spacedim>::cell\_iterator**

而TriaIterator的->操作符是这样定义的：

Accessor \* TriaRawIterator<Accessor>::operator -> ()

{

return &(this->operator\* ());

}

**所以cell\_iterator->set\_flags()就等同于(\*cell).set\_flags()，即CellAccessor.set\_flags()**

-----------------------------------------------------------------------------------------------------------------------------

**Kinds of accessors**

取决于你想要访问什么样的数据，有对应的不同accessor类：

TriaAccessor：提供了识别cells，faces，lines，quads，hexes的几何性质的数据，以及母子关系。

CellAccessor：由TriaAccessor继承来

DoFAccessor：提供了对cells，faces等上的自由度的数据访问。DoFHandler就是这样实现的。是由TriaAccessor或CellAccessor继承而来，可提供比它的基类更多的信息。

通常对用户而言，不需要在程序中用到上面那些accessor的名称，一般只会用到Triangulation，DoFHandler或hp::DoFHandler中提供的typedefs的名字，以及那些产生这些对象的函数：

|  |  |  |
| --- | --- | --- |
| **Class** | **cell\_iterator type** | function call |
| **Triangulation** | Typename **Triangulation::cell\_iterator** | triangulation.begin() |
| **DoFHandler** | Typename **DoFHandler::cell\_iterator** | dof\_handler.begin() |
| **hp::DoFHandler** | Typename [**hp::DoFHandler::cell\_iterator**](https://dealii.org/developer/doxygen/deal.II/group__Iterators.html#ga60277a8a3957ba4b41c1e76a87decd30) | hp\_dof\_handler.begin() |

|  |  |  |
| --- | --- | --- |
| **Class** | **face\_iterator type** | function call |
| **Triangulation** | Typename **Triangulation::face\_iterator** | triangulation.begin\_face() |
| **DoFHandler** | Typename **DoFHandler::face\_iterator** | dof\_handler.begin\_face() |
| **hp::DoFHandler** | Typename **hp::DoFHandler::face\_iterator** | hp\_dof\_handler.begin\_face() |

类似地，active iterators有下列性质：

|  |  |  |
| --- | --- | --- |
| **Class** | **cell\_iterator type** | function call |
| **Triangulation** | Typename **Triangulation::active\_cell\_iterator** | triangulation.begin\_active() |
| **DoFHandler** | Typename **DoFHandler::active\_cell\_iterator** | dof\_handler.begin\_active() |
| **hp::DoFHandler** | Typename **hp::DoFHandler::active\_cell\_iterator** | hp\_dof\_handler.begin\_active() |

|  |  |  |
| --- | --- | --- |
| **Class** | **face\_iterator type** | function call |
| **Triangulation** | Typename **Triangulation::active\_face\_iterator** | triangulation.begin\_active\_face() |
| **DoFHandler** | Typename **DoFHandler::active\_face\_iterator** | dof\_handler.begin\_active\_face() |
| **hp::DoFHandler** | Typename **hp::DoFHandler::active\_face\_iterator** | hp\_dof\_handler.begin\_active\_face() |

除了上述这些作用于cells或faces的类（cell或face的确切含义与维数有关），还有如begin\_active\_quad()或end\_quad()这种与维数无关的，作用于line，quad或hex这种几何对象的函数。

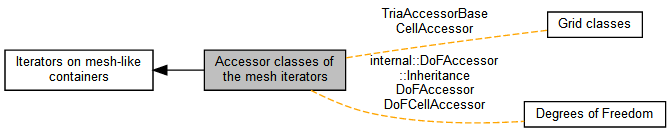
**Iterator and accessor internals**

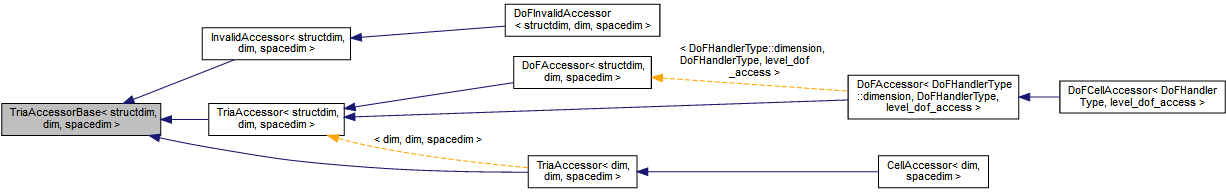
Iterators，类似pointers，好像是指向了实在的对象，但实际上在解引用的时候返回的是accessor。而accessor对象知道它代表了什么对象。Accessor保存了这样的信息：它属于哪个Triangulation，这个triangulation是什么level，这个level下的某个cell的index。

Cells是以等级结构来存放的，但faces不是。对应cells的accessors来说，有变量present\_level和present\_index，用来标记处于哪个level，是哪个index。

**Accessor classes of the mesh iterators**

注：第二部分着重讲**访问器**accessor





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

最基本的是**TriaAccessorBase**，然后是TriaAccessor，DoFAccessor等，然后是CellAccessor。

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

**class TriaAccessorBase< structdim, dim, spacedim >**

这个基本的accessor类，是与基本的TriaRawIterator配合使用的（作为参数传给Iterator类）。这个类只保存必要的数据成员，用于回应iterators的请求。注意，**只有accessor才是真正进行数据访问的类，iterator不是！**

当structdim=dim时（cells）或structdim<dim时（faces或edges），这个类的行为是不相同的。对于后者，present\_level始终是0。对于cells，则任意level都有可能。

构造函数：

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

TriaAccessorBase< structdim, dim, spacedim >::TriaAccessorBase

( const Triangulation< dim, spacedim > \* parent = nullptr,

const int level = -1,

const int index = -1,

const AccessorData \* = nullptr

)

成员函数：

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

int TriaAccessorBase< structdim, dim, spacedim >::**level** ( ) const

对于cells，这个函数返回这个cell所处的网格等级level。对其他对象，此函数返回0.

注：在triangulation对象中，cells唯一地被pair(level, index)所标记。level是cell所处的细化等级，index是cell在这个等级中的编号索引。与此相反，如果当前对象是face或edge的话，则唯一地被index标记，不含有细化等级。

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

int TriaAccessorBase< structdim, dim, spacedim >::**index** () const

返回当前指向的元素在当前level中的index。注意，同一等级中的编号可能不是连续的。

Accessor必须满足这样一些要求：

1）必须有两个成员变量present\_level和present\_index，用来存放当前accessor代表的元素的地址。

2）必须有构造函数参数是：指向triangulation的指针、两个整数。两个整数是代表初始level和index。3）对于TriaIterator和TriaActiveIterator类，必须有成员函数bool used()/bool active()

4）必须有void operators++和-

5）必须声明一个局部typedef AccessorData，用来指明（作为第四个构造函数参数）被接收的数据类型。

**Past-the-end iterators**

若present\_level==-1 且 present\_index==-1，则迭代器指向了最后一个对象的下一个（past the end），此时迭代器失效；

若指向Triangulation的指针失效或为0，则迭代器也失效