**Manifold indicator** //摘自Glossary entry on manifold indicators

组成Triangulation的每一个对象(cells,faces,edges,etc.)，都附带了一个独一的数字，用于当细化网格需要创建新的点时，识别由哪种流形对象来负责创建新点。

默认情况下，网格的全部流形指示器都被设置为numbers::invalid\_manifold\_id。典型的把流形指示器设置为其他值的代码如下，在这里把所有中心坐标的x分量小于零的网格的流形指示器设为42：

for (typename Triangulation<dim>::active\_cell\_iterator

cell = triangulation.begin\_active(); cell != triangulation.end(); ++cell)

if (cell->center()[0] < 0)

cell->set\_manifold\_id (42);

在这里我们调用了函数TriaAccessor::set\_manifold\_id()。也可以换作调用TriaAccessor::

set\_all\_manifold\_ids，循环地在每个face(和edge, 若在3d中)上设置流形id。若需要询问某个对象边的流形指示器，使用TriaAccessor::manifold\_id().

上面的代码只是在网格的特定部分设置了流形指示器，但它自身并没有改变网格在细化时的方式。为此，还需要调用函数Triangulation::set\_manifold()把某个特定的流形对象附加给某个特定的流形指示器。这样，就允许Triangulation对象使用一种不同的方式来创建新点(在细化网格时)。默认的方式是在所有faces和edges上使用FlatManifold对象。

NOTE：在细化网格时，母网格的流形指示器会继承给子网格。

**Boundary indicator**  //摘自Glossary entry on boundary indicator

Triangulation对象的边界的每个部分，都附带了一个独一的数字，用于指定在边界的某个部分应该施加什么边界条件。

默认情况下，所有边界指示器都设为0，除非你是从某个显式地设置成了别的值的文件读入网格，或者除非你使用了GridGenerator下的有着‘colorize‘选项的网格生成函数来生成。典型的把边界指示器设置为其他值的代码如下，在这里把所有位于x=-1的face的边界指示器设为42：

for (typename Triangulation<dim>::active\_cell\_iterator

cell = triangulation.begin\_active(); cell != triangulation.end(); ++cell)

for (unsigned int f=0; f<GeometryInfo<dim>::faces\_per\_cell; ++f)

if (cell->face(f)->at\_boundary())

if (cell->face(f)->center()[0] == -1)

cell->face(f)->set\_boundary\_id (42);

在这里我们调用了函数TriaAccessor::set\_boundary\_id()。3d情况中，也可以换作调用TriaAccessor::

set\_all\_boundary\_ids(在每个选定的faces上)。若需要询问某个面或边的边界指示器，使用函数TriaAccessor::boundary\_id。

在deal.II的更老的版本中(8.2之前)，如果你想要改变细化网格时triangulation对待边界的方式，可使用函数Triangulation::set\_boundary来把一个边界对象联系到指定的边界指示器上去。这个方式现在仍然支持，默认的对待边界的方式是使用StraightBoundary对象。在step-49的结果区有一个例子说明了所有的这些操作。

但从8.2版以后，我们更建议把边界的几何描述与它的物理意义分离开处理，即**分别使用manifold\_ids和boundary\_ids。前者是用于描述几何上应该怎么改变，后者是用于识别使用哪种边界条件**。

NOTE：在细化网格时，母网格faces的边界指示器会继承给子网格。

流形描述的必要性体现在两方面：

* 网格加密：每当细化一个网格时，需要引入一个新的角点。在最简单的情况下，认为triangulation是由直线段、双线性面及三线性体构成的。那么细化网格的方式就是直接在原先对象的中点创建新的点。这是triangulation类默认的操作，是由FlatManifold类来描述的。

但是，如果需要处理曲面几何，则这种方式就不合适了。我们需要Manifold的派生类来描述几何区域。可使用函数Triangulation::set\_manifold()及附带于它的一个流形id(见types::manifold\_id)，把Manifold派生类对象添加到网格对象上，并通过函数TriaAccessor::set\_manifold\_id()在应该由这个流形来描述的cells、faces或edges上来使用这个流形id，然后triangulation会询问这个流形对象如果细化网格的话，应该在哪里创建新的点。已经有了几个类可支持最常见的几何，如圆柱流形或PolarManifold，分别代表了当你使用柱坐标或极坐标来表述你的空间时获得的几何。

* 积分：当使用高阶有限元方法的时候，常需要使用曲边近似来计算cell term，而非直接线性近似。这种曲边单元的实现实际上是在Mapping类中(见Mapping between reference and real cell模块)，但它需要从这里介绍的这个类中获得关于边界的信息。

在这两种情况下，我们都需要在单个网格层面上提供计算域的几何边界的信息。也就引出了所谓的manifold类。manifold类提供了供triangulation类和mapping类询问几何信息的接口。

在deal.II中，一个Manifold可以看作一些点(及点之间距离描述)的集合。新的点通常的获得方式是：

提供一个流形上的局部坐标系→在局部坐标系中识别已有的点(使用局部映射获得局部坐标值)→寻找新的点(通过已有点的加权)→转换回实际空间中(利用局部映射)。

目前在dealii中实现这种机制的类是ChartManifold类。所谓chart，是指应用的域（的一部分）是一个chart，它可以用从简单域到chart的光滑函数(push\_forward函数)及其逆(pull\_back函数)来描述。如果计算域不是一个chart(比如一个球的表面)，那么它通常可以看作是多个charts的组合(例如可看作是北半球和南半球这两个charts的组合)，此时域可看作一个atlas。如果需要用于复杂几何，用户可对这个类进行重载。

虽然这一过程在大多数应用问题中都是不平凡的，但对于多数平凡的情况如圆柱或者球等，deal.II已经提供了相对完备的实现。更复杂的应用可参考step-53和step-54中使用的技术。

一个Triangulation的边界是Manifold的一种特殊情形，在用户代码中可能需要它的一些信息，例如表面的法向量。如果你的粗网格已经足够好，你可能只对附加一个流形描述到你区域的边界部分上去感兴趣。这可以通过使用函数Triangulation::set\_boundary()来实现，它需要一个Boundary对象作参数(由Manifold派生来)。注意，Triangulation只使用Manifold接口，不使用Boundary接口。然而其他工具，可能需要在积分点上计算精确法向量，因此提供了一个询问Boundary对象的包装。

**An example**

在step-1中已经说明了为什么需要处理曲面边界，虽然在那里没有细讲。例如下面这个程序，只是细化每一个网格若干次，而根本不处理其边界：

Triangulation<2> triangulation;

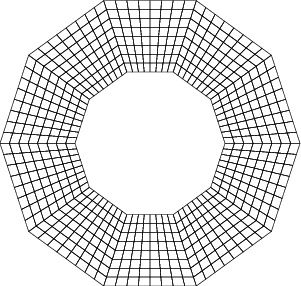
const Point<2> center(1, 0);

const double inner\_radius = 0.5, outer\_radius = 1.0;

GridGenerator::hyper\_shell (triangulation, center, inner\_radius, outer\_radius, 10);

triangulation.refine\_global(3);

这个程序生成的网格如下：



我们的目的是生成一个近似环形的网格。然后，因为我们没有把这描述给网格，所以细化时是直接在最初的粗网格的边线中点细分。

这很容易补救，在step-1中已经展示了怎么做。见下面代码：

Triangulation<2> triangulation;

const Point<2> center (1,0);

const double inner\_radius = 0.5, outer\_radius = 1.0;

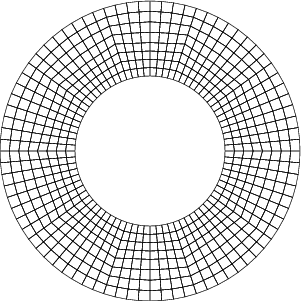
GridGenerator::hyper\_shell (triangulation, center, inner\_radius, outer\_radius, 10);

const HyperShellBoundary<2> boundary\_description(center); //添加了这个用于描述边界的对象

triangulation.set\_boundary (0, boundary\_description);

triangulation.refine\_global (3);

结果如图：



这个网格在内外边界上已经好了很多，虽然还能通过内部网格线的折角分辨出最初的10个网格。这是因为我们只在边界上附加了边界对象。

这可通过同时分配流形描述给边界及径向的线和网格来补救。如下：

Triangulation<2> triangulation;

const Point<2> center (1,0);

const double inner\_radius = 0.5, outer\_radius = 1.0;

GridGenerator::hyper\_shell (triangulation, center, inner\_radius, outer\_radius, 10);

const SphericalManifold<2> boundary\_description(center); //定义流形对象

triangulation.set\_manifold (0, boundary\_description); //把流形对象联系到值为0的流形id

Triangulation<2>::active\_cell\_iterator

cell = triangulation.begin\_active(),

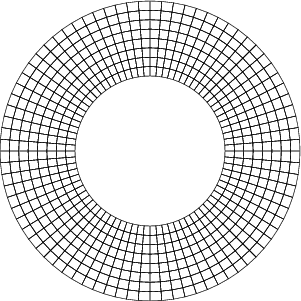
endc = triangulation.end();

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

cell->set\_all\_manifold\_ids (0); //把所有网格的流形id设置为0，从而都对应了上面的流形对象

triangulation.refine\_global (3);

这样得到的网格是：



我们为什么需要这样做呢？毕竟后面这两个网格所代表的domain是一样的，而我们知道只要网格足够细，最后收敛到的结果肯定是一样的。

对于这一问题有两个回答：首先，求解一个偏微分方程到一定的数值精度所需要的数值努力通常依赖于网格的质量，因为在误差估计形式中常数C依赖于所有网格的最小外接圆与最大内切圆的最大比值。因此，创建足够好的网格是值的的。虽然对于上面的网格，这一问题不是很明显，但考虑下面的代码和网格，你就能明白其重要性了：

Triangulation<2> triangulation;

const Point<2> center (1,0);

const double inner\_radius = 0.5, outer\_radius = 1.0;

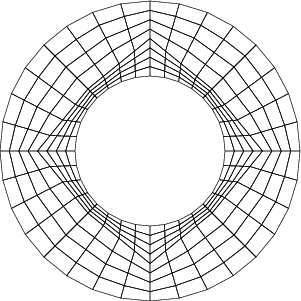
GridGenerator::hyper\_shell (triangulation, center, inner\_radius, outer\_radius, 4);

// four circumferential cells

const HyperShellBoundary<2> boundary\_description(center);

triangulation.set\_boundary (0, boundary\_description);

triangulation.refine\_global (3);



显然，这样的网格的纵横比很差。如果初始网格为3个周向网格，那得到的结果更差，甚至会得到负体积的扭曲网格。然而，如果不仅仅是在边界上附加几何描述对象，而是同时在边界和内部网格上附加，那么即便初始网格很少，得到的结果也是好的：

Triangulation<2> triangulation;

const Point<2> center (1,0);

const double inner\_radius = 0.5, outer\_radius = 1.0;

GridGenerator::hyper\_shell (triangulation, center, inner\_radius, outer\_radius, 3);

// three circumferential cells

const SphericalManifold<2> boundary\_description(center);

triangulation.set\_manifold (0, boundary\_description);

Triangulation<2>::active\_cell\_iterator

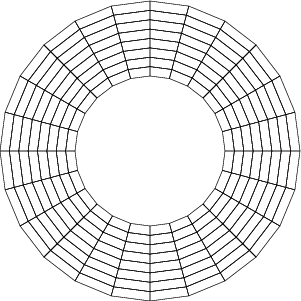
cell = triangulation.begin\_active(),

endc = triangulation.end();

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

cell->set\_all\_manifold\_ids (0);

triangulation.refine\_global (3);



**代码示例**(我自己写的一段生成环形域的代码)：

//生成一个环形域gmsh格式的网格

void generate\_annulus(){

//环向上初始网格为20个,最后那个参数true用于在不同边界上设置不同的boundary\_id

Triangulation<2> triangulation;

const Point<2> center(0,0);

const double inner\_radius = 0.5, outer\_radius = 10.0;

GridGenerator::hyper\_shell (triangulation, center, inner\_radius, outer\_radius, 20, true);

//创建一个用于描述圆流形的对象boundary\_description

const SphericalManifold<2> boundary\_description(center);

//把这个流形对象分配给所有标记了manifold\_id=0的网格

triangulation.set\_manifold(0, boundary\_description);

//把全部网格的manifold\_id都设为0，那么全部网格都与上面所定义的流形对象联系起来

Triangulation<2>::active\_cell\_iterator

cell = triangulation.begin\_active(),

endc = triangulation.end();

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

cell->set\_all\_manifold\_ids(0);

//各向异性加密，只在径向加密6次，每次都是在靠近内边界的网格上切分

for(unsigned int step=0;step<6;++step)

{

Triangulation<2>::active\_cell\_iterator

cell = triangulation.begin\_active(),endc = triangulation.end();

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

for(unsigned int v=0;v<GeometryInfo<2>::vertices\_per\_cell;++v)

{

const double dist\_from\_center = center.distance(cell->vertex(v));

if(std::fabs(dist\_from\_center-inner\_radius)<1e-10)

{

cell->set\_refine\_flag(RefinementCase<2>::cut\_y);

break;

}

}

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

}

//最后再进行全局加密2次

triangulation.refine\_global(2);

//把triangulation按gmsh格式写出

std::ofstream out("annulus.msh");

GridOut grid\_out;

grid\_out.set\_flags(GridOutFlags::Msh(true,true));

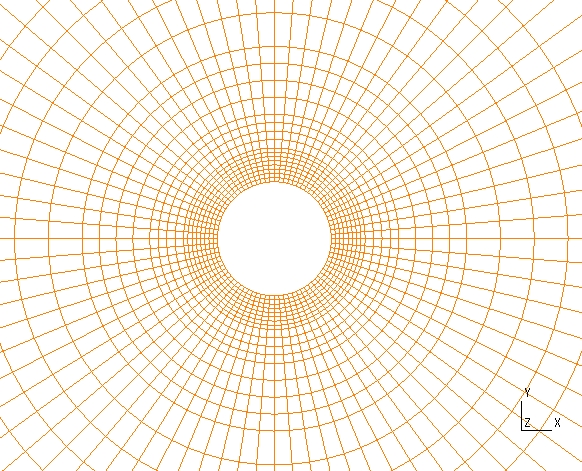
grid\_out.write\_msh(triangulation, out);

//.最后，再把流形对象移除，这样在析构它的时候它是没有处于被triangulation使用的状态的。必须加这行，不然程序执行完毕时会报错。

triangulation.set\_manifold(0);

}

得到的网格如图：



==========================================================================

**ChartManifold**

这个类描述可以用charts来表达的mapping。所谓“图”chart，是指我们考察的计算域(的一部分)，可以通过从一个简单得多的域(我们称之为参考域)依靠光滑函数(push\_forward函数)映射得到。它特殊化了一个嵌入在一个spacedim维流形中的，chartdim维的流形，你可显式地使用pull\_back()和push\_forward()变换。它的用法在step-53中有详细介绍。

当你有一个从chartdim维的欧几里得空间到spacedim维的欧几里得空间（代表了你的流形显式映射）的时候，这个类会是一个很有用的辅助类。即，当你的流形M可通过映射（即push\_forward()函数）：



来表达，且满足逆变换（即pull\_back()函数）：



ChartManifold类的get\_new\_point()函数是通过对所有surrounding\_points来调用pull\_back()函数，计算它们在chartdim欧几里得空间中的加权平均，再调用push\_forward()函数得到最终的点来实现的：



派生类需要实现push\_forward()和pull\_back()方法。则所有其他mappings要求的函数（除了push\_forward\_gradient()函数）都会由这个类来提供。

**Providing function gradients**

为了计算相切于流形的向量(例如，相切于嵌在高维空间中的表面)，还需要push\_forward函数F的梯度。这个梯度是个矩阵 ，在某个点x，是个spacedim×chartdim维的矩阵。

只有部分有限元程序可能需要用到流向的切向量，因此，派生类需要实现抽象虚函数push\_forward()和pull\_back()函数，但不一定非要实现虚函数push\_forward\_gradient()函数。因此这个虚函数有个默认的实现，但这个默认实现不会计算出任何有意义的结果而是会触发一个exception。

**A note on the template arguments**

维数参数chartdim，dim和spacedim必须满足下列关系：

dim≤spacedim； chartdim≤spacedim

然而，在dim和chartdim间并没有确定的优先关系。例如，如果你想描述嵌在3d空间中的2d网格的边（1d对象）的mapping，你可以通过以下参数化来实现：



这种情况下chartdim=1。另一方面，当然也可以把这描述成映射：



也就是线段[0,1]×{0}×{0}恰好被映射到边上。这里chartdim=3。这可能看起来有点傻，但满足...未完待续...

==========================================================================

**FunctionManifold**

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

class FunctionManifold<dim, spacedim, chartdim>

这个流形类是由ChartManifold派生而来，基于显式的Function<spacedim>和Function<chartdim>对象（描述push\_forward()和pull\_back()函数）

你可以使用这个流形对象来描述任意外形，只要你可以用一个可逆映射来表达它，为此你需要提供forward expression和inverse expression。

构造函数

FunctionManifold<dim, spacedim, chartdim>::

FunctionManifold(const Function<chartdim>& push\_forward\_function,

const Function<spacedim>& pull\_back\_function,

const Tensor<1, chartdim>& periodicity=Tensor<1, chartdim>(),

const double tolerance = 1e-10)

FunctionManifold<dim, spacedim, chartdim>::

FunctionManifold(const std::string push\_forward\_expression,

const std::string pull\_back\_expression,

const Tensor<1,chartdim>& periodicity= Tensor<1, chartdim>(),

const typename FunctionParser<spacedim>::ConstMap

const\_map=typename FunctionParser<spacedim>::ConstMap(),

const std::string chart\_vars=

FunctionParser<chartdim>::default\_variable\_names(),

const std::string chart\_vars=

FunctionParser<chartdim>::default\_variable\_names(),

const double tolerance = 1e-10,

const double h=1e-8)

==========================================================================

==========================================================================

**Manifold**

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

**class Manifold<dim, spacedim>**

流形是用于描述域的边界几何及内部几何的。理解流形概念，最好是使用微分几何的语言，但它们的使用可以通过例子来简单地描述。

**Common use case：Creating a new vertex**

流形对象最基本的应用是用于“基于已有的点创建新的点”。例如，当需要在网格的某个网格或面或边上创建新的点时，它通过以下函数来决定新的点的坐标：

...

Point<spacedim> new\_vertex = manifold.get\_new\_point(points, weights);

...

这里，points是在spacedim维空间中的一组点，且有对应的一组权重weights。这里的points就是指网格单元或面或边的角点，而weights就是1除以点的数量。它的派生类需要实现这个Manifold::get\_new\_point()函数，从而计算出新点的坐标。最简单的情形就是FlatManifold中的实现，它里面的这个函数仅仅是计算给定点的算数平均位置(借助给定的权重)。但其他类可能不一样，比如SphericalManifold类，它是用于描述球型域的，给定边界上两个点，新建的点会位于这两点所在的某个圆上，而非位于这两点连线的中间某个位置。

Manifold::get\_new\_point()有一个默认实现：在内部，此函数调用Manifold::get\_intermediate\_point()函数去计算出成对的中间点。函数Manifold::get\_intermediate\_point()计算出给定点的凸组合(即candidate点)，再调用Manifold::project\_to\_manifold()。从而派生类可通过重载Manifold::project\_to\_manifold()来实现简单情况的应用。这在描述嵌入在高维空间中的低维流形时常常很有用，例如，球体的表面。在这些情况中，新的点可简单地通过计算给定点的(加权)平均值，再投影回球面来得到。

**Common use case：Computing tangent vectors**

这个类的第二种用途是用于计算域和边界的方向。例如，我们为了施加滑移边界，可能需要知道网格面上的法向量。或者在计算解的梯度的法向值时，也需要知道法向量。

为此，Manifold类提供了一个成员函数(在派生类中实现)Manifold::get\_tangent\_vector()，用于计算“在给定点处相切于流形的向量”。例如，在2d中，给定某个边的两个顶点，需要利用这个函数来计算沿着该边的某个切向量，再通过旋转90°来得到法向量。在3d中，则需要先计算出某个边界面的两条边的切向量，在通过求叉乘来得到其法向量。

由于一些更复杂的原因，这些方向向量是按特定方式来模化的，而非有单位模值。参加Manifold::get\_tangent\_vector()函数的文档来获取更多细节。

在最简单的情况下(即FlatManifold类)，这些切向量仅仅是给定两个点的向量差。但对于更复杂的情况，如SphericalManifold类中，如果给定的两点位于某个圆上，则该切向量会相切于该圆，而非直接从一个点指向另一个点。

**A unified description**

为了理解这个类，应该把它放到微分几何的框架下。具体地讲，微分几何是基于这样的假设：两个足够相近的点是通过一条有“最短距离”的线来联系的。这条线被称为“测地线”，它是从所有连接这两点的连线中选取的“距离最短”的那条，而这里的“距离”概念是通过某种描述流形的“度量metric”来测量的。例如，在地球表面上寻找一条从北京到上海最近的线，这条线应该是条曲线，因为我们在球面上的距离概念与在平面上的距离概念是有区别的。

在下面的讨论中，“度量”的概念不再那么重要，我们仅仅假设某个流形上连接两点的测地线是存在的。

有了测地线，现在我们可以更加正式地来描述前面两节讨论过的那些操作了。实际上，我们可以假设测地线是某个参数t的函数，**s**(t)描述了这条线，**s**(0)是第一个点的位置而**s**(1)是第二个点的位置。而且，**s**(t)随着t是均匀变化的(在特定度量下，在相同时间内走过相同距离)。

那么，计算两个点**x**1和**x**2的带权重w1和1-w1的中点，也就等价于计算点**s**(w1)。而计算超过两个点的加权平均点，则等价于考虑成对的测地线，先找出前两个给定点的测地线上的相应点，再找这个新点与第三个给定点间测地线上的点，依次类推。

类似地，切向量其实就变成了计算**s**(t)的速度**s**‘(t)在某个端点处的值。

**成员函数：**

Point<spacedim> Manifold<dim, spacedim>::

**get\_new\_point**(const ArrayView<const Point<spacedim>>& surrounding\_points,

const ArrayView<const double>& weights) const

给定surrounding\_points，返回其在当前流形描述下的加权平均点。

在它的默认实现中，它通过调用函数get\_intermediate\_point()来成对地依次减少点的数目。而函数get\_intermediate\_point()又默认调用project\_to\_manifold()函数。对简单的用户类而言，可只实现自定义的project\_to\_manifold()函数即可。

Point<spacedim> Manifold<dim, spacedim>::**get\_intermediate\_point**(const Point<spacedim>& p1,

const Point<spacedim>& p2,

const double w) const

给定两个点，返回中间点。

这个函数的一个实现应该返回流形上的一条参数曲线，它包含了点p1和p2以及位于区间[0，1]上的参数w。在它的默认实现中，此函数先求出p1和p2点的某种凸组合，再以之为输入参数调用了project\_to\_manifold()函数，所以用户代码实现project\_to\_manifold()即可。

Point<spacedim> Manifold<dim, spacedim>::

**project\_to\_manifold**(const ArrayView<const Point<spacedim>>& surrounding\_points,

const Point<spacedim>& candidate) const

给定一个很贴近给定流形的点candidate(应该是p1和p2点的凸组合)，这个函数修改该点并把它投影到流形上。

==========================================================================

在利用Manifold对象定义高阶边界并求高阶边界的法向量时的具体函数调用过程：

(可利用我自己写的test\_visualize\_shape\_function这个函数进行调试测试查看调用过程)

1. fe.normal\_vector(i)

fe内部有个mapping\_output对象保存了normal\_vectors[i]

这个mapping\_output对象其实是::internal::FEValuesImplementation::MappingRelatedData<dim>类，它是在

2. FEValues::reinit()🡪(call)Mapping::fill\_fe\_values()时返回的对象

3. virtual CellSimilarity::Similarity Mapping<dim>::fill\_fe\_values(cell\_iterator,

cell\_similarity,

quadrature,

internal\_data,

output\_data)

此函数在MappingManifold下的具体实现为：

virtual CellSimilarity::Similarity MappingManifold<dim>::fill\_fe\_values(cell\_iterator,

cell\_similarity,

quadrature,

internal\_data,

output\_data)

它在内部计算output\_data.normal\_vectors[point]时调用的函数为：

4. cross\_product\_2d(-DX\_t[0])，其参数是个Tensor，这个函数本身也是Tensor类的成员函数，它的作用是把切向量旋转90°得到法向量。