OpenFOAM

作者：Koishi

时间：November 26, 2024

ElegantLaTeX Program

不要以为抹消过去，重新来过，即可发生什么改变。——比企谷八幡

目录

第 1 章 网格生成

1.1 blockMesh

blockMesh 能够创建具有分级和弯曲边缘的参数网格，其运行根据位于 system 文件夹中的 blockMeshDict 字典文件。blockMeshDict 文件的包含的关键词如下所示：

关键词	含义	例子
convertToMeters	顶点坐标的放缩比例	取 0.01 时长度单位为毫米
vertices	顶点坐标集合	原点坐标为 (0 0 0)
edges	用来指定圆弧或样条曲线边缘	用两个顶点标签和中间经过点坐标指定圆弧 arc 1 4 (0.939 0.342 -0.5)
block	顶点标签集合和网格大小	hex (0 1 2 3 4 5 6 7) (10 10 1) simpleGrading (1 1 1)
patches	面的集合	symmetryPlan base ((0 1 2 3))
mergePatchPairs	合并的面的集合	

顶点的描述是按照坐标点列表的方式进行的，从上往下依次表示顶点标签的递增，例如下面是一个 block 的顶点集合表示方式：

```
vertices
(
        (0    0    0  )    // 标签0
        (1    0    0.1)    // 标签1
        (1.1  1    0.1)    // 标签2
        (0    1    0.1)    // 标签3
        (-0.1 -0.1 1  )    // 标签4
        (1.3  0    1.2)    // 标签5
        (1.4  1.1  1.3)    // 标签6
        (0    1    1.1)    // 标签7
);
```

边缘的描述需要使用两个顶点标签，不过同样要用关键词指定边缘类型，不同类型还要求更进一步的补充点坐标信息，下面是可用的关键词：

关键词	含义	所需补充信息
arc	圆弧	一个中间插值点的坐标
simpleSpline	样条曲线	一系列插值点的坐标
polyLine	一组线	一系列插值点的坐标
polySpline	一组样条曲线	一系列插值点的坐标
line	直线	

下面是通过标签 1 顶点和标签 5 顶点定义的圆弧边缘，该圆弧通过了坐标为 (1.1 0.0 0.5) 的点：

```
edges
(
        arc  1  5  (1.1 0.0  0.5)
);
```

网格块的定义涉及到点标签集、三个方向的网格数量、非均匀化比例的定义。一个 block 将会涉及到 8 个顶点标签，顶点标签的顺序决定了该网格块的坐标方向，也就会影响加密的设置顺序。顶点标签与坐标方向有下面的关系：

- 第一个顶点标签决定了坐标系的原点；
- 从第一个顶点标签位置指向第二个顶点标签位置决定了 x_1 方向；
- 从第二个顶点标签位置到第三个顶点标签位置决定了 x_2 方向；
- 前四个顶点标签位置决定了 $x_3 = 0$ 的平面位置；
- 从第一个标签位置指向第五个标签位置决定了 x_3 方向；
- 剩余三个标签类似地依次从第二、三、四个标签位置沿 x_3 方向移动得到。

1.2 snappyHexMesh

1.2.1 生成背景网格

该功能可以通过三角面片模型文件 (stl,obj,vtk,...) 来创建网格，该模型文件应当放置在 constant/triSurface 文件夹内。同时还需要背景网格来决定计算域和基本网格密度，通常通过 blockMesh 来生成，而由于一般来说都是要生成长方体规则网格，所以可以按照下面的格式来书写 blockMeshDict 文件（根据需要修改大小以及边界面类型）：

```
convertToMeters 1;

xmin    -3;
xmax    3;
ymin    -3;
ymax    3;
zmin    0;
zmax    10;

xcells  40;
ycells  40;
zcells  40;

vertices
(
    ($xmin $ymin $zmin)
    ($xmax $ymin $zmin)
    ($xmax $ymax $zmin)
    ($xmin $ymax $zmin)

    ($xmin $ymin $zmax)
    ($xmax $ymin $zmax)
    ($xmax $ymax $zmax)
    ($xmin $ymax $zmax)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) ($xcells $ycells $zcells) simpleGrading (1 1 1)
```

```
);

boundary
(
    inlet
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (1 2 6 5)
        );
    }
    ground
    {
        type patch;
        faces
        (
            (0 3 2 1)
        );
    }
    top
    {
        type patch;
        faces
        (
            (4 5 6 7)
        );
    }
    side1
    {
        type patch;
        faces
        (
            (0 1 5 4)
        );
    }
    side2
    {
        type patch;
        faces
        (
```

```
            (3 7 6 2)
        );
    }
);


mergePatchPairs
(
);
```

1.2.2 surfaceFeaturesDict

surfaceFeaturesDict 文件用来提取模型文件的表面特征，需要指定模型文件以及调用的程序，其模板为：

```
FoamFile
{
    format      ascii;
    class       dictionary;
    object      surfaceFeaturesDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

surfaces ("combined.stl");
includedAngle 150;

#includeEtc "caseDicts/surface/surfaceFeaturesDict.cfg"
```

完成后就可以在算例目录下在终端输入：

```
surfaceFeatures
```

如此就能从几何文件中提取出特征边缘，并在 triSurface 文件夹内生成 cylinder.eMesh 文件。

1.2.3 meshQualityDict

meshQualityDict 文件几乎不需要改动，其内容只有调用相应程序：

```
FoamFile
{
    format      ascii;
    class       dictionary;
    object      meshQualityDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#includeEtc "caseDicts/mesh/generation/meshQualityDict.cfg"
```

1.2.4 snappyHexMeshDict

完成上述文件后就可以开始对 snappyHexMeshDict 文件进行修改。该文件中涉及的关键词如下：

关键词	含义	常用赋值
castellatedMesh	是否创建堞状网格	true
snap	是否进行表面贴合	true
addLayers	是否添加表面边界层	false
mergeTolerance	合并公差作为初始网格边界框的分数	1e-6
debug	控制中间网格和网线打印的写入	0 只写最终网格 1 写中间网格 2 用 cellLevel 写 volScalarField 进行后期处理 4 将当前交叉点写入.obj 文件
geometry	所使用的表面图形的信息	
castellatedMeshControls	生成蜂窝网格的控制信息	
snapControls	表面贴合的控制信息	
addLayersControls	表面边界层添加的控制信息	
meshQualityControls	网格质量的控制信息	

在 geometry 关键词内设置所需的几何文件以及体加密信息。下面是一个简单的案例：

```
geometry
{
    cylinder  // 模型边界面的自定义命名
    {
        type triSurfaceMesh;
        file "cylinder.stl"; // 所用文件
    }

    refinementBox  // 体加密区域的自定义命名
    {
        type searchableBox; // 体加密类型
        min ( -5  -5  -1);
        max (15 15 5);
    }
};
```

如果在 stl 文件中已经自行定义了各个 solid 的名字，那么就可以添加 regions 关键字来提取这些面域，进而在后续的面加密中各自进行处理。例如：

```
geometry
{
    cylinder  // 模型边界面的自定义命名
    {
        type triSurfaceMesh;
        file "cylinder.stl"; // 所用文件

        regions
        {
            top { name top; }
            bottom { name bottom; }
            wall { name wall; }
        }
```

```
    }

    refinementBox  // 体加密区域的自定义命名
    {
        type searchableBox; // 体加密类型
        min (  -5   -5   -1);
        max (15 15 5);
    }
};
```

体加密除了上述方块体加密设置，还有其他可选加密方式：

加密类型	类型含义	所需参数	参数解释
searchableBox	立方体	min	最小对角点坐标
		max	最大对角点坐标
searchableCylinder	圆柱	point1	中轴线端点 1
		point2	中轴线端点 2
		radius	外圆半径
searchableSphere	球	Centre	中心点坐标
		radius	球半径
searchableCone	圆锥体 (空心)	point1	中轴线端点 1
		point2	中轴线端点 2
		radius1	断面 1 外圆半径
		radius2	断面 2 外圆半径
		innerRadius1	断面 1 内圆半径
		innerRadius2	断面 2 内圆半径
searchableRotatedBox	旋转立方体	origin	最小对角点坐标
		span	XYZ 方向尺寸长度
		e1	几何 i 方向向量
		e2	几何 j 方向向量
		e3	几何 k 方向向量
searchableExtrudedCircle	圆管	file	导入曲线文件
		radius	圆管半径

全局网格细化参数在 castellatedMeshControls 中设置，其目的为细化背景网格，通过细化背景网格以使几何特征与几何表面上拥有一定网格量，以提高几何特征捕捉的准确性。同时通过参数设置，保证网格细化时尺寸变化尽量平缓。在该关键词内需要设置的内容包括：

关键词	含义	常用赋值
maxGlobalCells	全局最大网格量	2e+06
maxLocalCells	单核最大网格量	1e+06
maxLoadUnbalance	最大负载平衡参数	0.1
minRefinementCells	最小细化单元数	0
nCellsBetweenLevels	缓冲层数	3
resolveFeatureAngle	自动检测角	30
locationInMesh	网格域控制点	(-3.25 -2.25 1.25)
allowFreeStandingZoneFaces	允许有独立面	false
features	需要细化的特性信息	
refinementRegions	用于细化的区域的信息	
refinementSurfaces	用于细化的曲面的信息	

- maxGlobalCells 功能主要目的是保证网格细化过程中，避免划分网格量太大，导致计算机内存溢出。当划分网格量超过此值时，细化过程将立即终止。此时，局部细化功能可能终止运行。
- maxLocalCells 参数主要应用于网格并行计算，其指定了细化网格过程中每个处理器处理的最大数量网格数。设置该参数时请保证一定的富余量，经常重新平衡每个处理器计算量将减慢网格生成过程。
- maxLoadUnbalance 参数主要应用于网格并行计算。当该参数值为 0 时，即强制负载平衡，即各处理器间处理的网格量严格保持单元总数/计算核数。较低的值可能会导致系统频繁的均衡网格负载量。而参数值设置为 1 时，则完全禁用网格均衡操作。
- minRefinementCells 参数指定了需细化特征的最小单元数。若特征上网格单元数量小于该参数，则停止对其细化。
- nCellsBetweenLevels 若用户设置参数值为 1，则表示不添加过渡区域。越大的值可使得网格大小过渡越平缓，但将导致网格量增加。
- resolveFeatureAngle 当曲率变化角超过该参数值时，特征区域网格使用最大面细化等级，而低于此角度的特征均采用最小面细化级别。默认参数值为 30，参数值设置为 360 时，表示关闭此功能。该参数生效的前置条件：(1) 面细化参数中最小和最大细化等级需不同。(2) 面贴合过程中特征捕捉需采用隐式方法。
- locationInMesh 参数定义是否允许几何中有独立的面存在。若设置参数值为 false，则表示在 refinementSurfaces 中指定的 faceZones 仅位于相应 cellZones 的边界上，作为不同域之间交界面。若该参数值为 true，则允许此 faceZones 作为独立面域（例如：挡板界面等）。如果用户没有指定 faceZones，则该参数不生效。

一般情况下，castellatedMeshControls 可以按照如下方式设置：

```
castellatedMeshControls
 {
    features
    (
      { file "cylinder.eMesh"; level 1; }
    );

    refinementSurfaces
    {
       cylinder  // 对应之前定义的几何表面名称
       {
          level (3 3);
          patchInfo { type wall; }
       }
```

```
    }

    refinementRegions
    {
        refinementBox  // 对应之前定义的加密区域名称
        {
            mode    inside;
            level   2;
        }
    }

    locationInMesh (-3.25 -2.25 1.25);
}
```

如果在之前 geometry 部分指明了不同的 regions，那么在 refinementSurfaces 就可以针对不同的面域进行不同程度的细化，同时赋予不同的边界类型：

```
castellatedMeshControls
 {
    features
    (
      { file "cylinder.eMesh"; level 1; }
    );

    refinementSurfaces
    {
        cylinder   // 对应之前定义的几何表面名称
        {
            level (0 0);

            regions
            {
                top
                {
                    level (1 1);
                    patchInfo { type patch; }
                }
                bottom
                {
                    level (1 1);
                    patchInfo { type patch; }
                }
                wall
                {
                    level (2 2);
                    patchInfo { type wall; }
                }
            }
        }
    }
```

```
    refinementRegions
    {
        refinementBox  // 对应之前定义的加密区域名称
        {
            mode    inside;
            level   2;
        }
    }

    locationInMesh (-3.25 -2.25 1.25);
}
```

面贴合参数在 snapControls 中设置，主要目的是将体网格节点移动到几何表面上，贴合体网格中锯齿状表面。在该关键词内需要设置的内容包括：

关键词	含义	常用赋值
tolerance	捕捉点最大相对距离	5
nSolveIter	网格贴合最大迭代次数	100
nSmoothPatch	面平滑迭代次数	0
nSmoothInternal	体网格平滑迭代次数	0
nRelaxIter	贴合松弛迭代次数	8
nFeatureSnapIter	特征边捕捉迭代次数	10
nFaceSplitInterval	面拆分迭代数	5
explicitFeatureSnap	显式特征捕捉	true
implicitFeatureSnap	隐式特征捕捉	false
multiRegionFeatureSnap	多域特征捕捉	false

- tolerance 建议 2-5 之间。该参数指定贴合算法中捕捉与特征面相关网格节点的最大相对距离，实际捕捉距离为 tolerance 参数值乘以相邻体网格尺寸。参数值必须大于或等于 1，如果值太低，则可能无法使偏差较大的网格节点移动到几何表面上。较高的值有助于增加几何的捕捉范围，但如果参数值设置过高，则有可能捕捉到与表面无关的网格节点。
- nSolveIter 该参数指定了网格贴合算法的最大迭代次数。较高的值会提高网格的质量，网格一致性更好，但网格划分时间会更长。简单模型可以将该参数值设置为 100，若贴合后网格质量不太理想，可尝试将该参数值增加到 300。
- nSmoothPatch 的 0 表示初始网格外形。该参数指定了表面上网格贴合的平滑迭代次数。增加迭代次数可以使曲面上网格平滑、贴合性更好，且能降低曲面上网格的歪斜率，但可能导致曲率突变特征（如直角等）弱化。
- nSmoothInternal 建议参数值与 nSmoothPatch 参数值一致。在执行网格平滑迭代时，边界面网格平滑迭代 nSmoothPatch 将与内部体网格平滑迭代 nSmoothInternal 联合使用。平滑迭代顺序为优先执行一次面平滑迭代（nSmoothPatch），再执行一次体网格平滑迭代（nSmoothInterna），以此循环。若用户设置 nSmoothInternal 参数值大于 nSmoothPatch 值时，平滑迭代次数统一采用 nSmoothPatch 参数值。默认值为零，表示禁用体网格平滑迭代。
- nRelaxIter 一般为 5-8 之间。该参数指定贴合过程中松弛迭代次数，用以消除质量较差的单元或网格节点。如果迭代完成后网格仍存在质量较差单元，则用户可以尝试增加此迭代次数，较高的值将确保更好的网格质量，但会花费更多计算时间。

- nFeatureSnapIter 该参数指定了特征捕捉迭代次数，以将网格点捕捉到表面边缘。如果在 nFeatureSnaplter 迭代后局部特征区域网格没有达到足够的质量标准，则取消该区域特征边捕捉并恢复到之前状态。未指定该参数，特征捕捉功能将被禁用。
- nFaceSplitInterval 当系统执行完特征边捕捉迭代步骤后，若网格边缘与特征边未完全对齐，则有可能在特征边处网格产生凹面。从而导致在添加边界层时，其投影体网格的非正交性增大。如果体网格不满足质量要求，则取消该处边界层生成。nFaceSplitInterval 参数默认值为-1（禁用），使用建议参数值设置为特征边捕捉迭代（nFeatureSnaplter）次数的一半。
- explicitFeatureSnap 显示特征捕捉方法需要用户自定义特征边文件（.eMesh），并且指定特征边的细化等级（通过 castellatedMeshControls 子字典中 features 参数指定）。
- implicitFeatureSnap 隐式方法不需要用户提取几何特征边，其特征识别自动化程度优于显示特征捕捉方法。它使用全局细化参数中 resolveFeatureAngle 参数识别曲面几何特征（例如：面的相交线、曲率变化较大的曲面特征）。但在尖角特征或者挡板界面处，显示方法捕捉特征效果优于隐式方法。
- multiRegionFeatureSnap 该参数用于捕捉多域网格间的特征面，这对于具有多个区域（例如流体区域和固体区域）的网格进行共轭传热模拟或类似操作很重要。该参数生效的前置条件为采用显示特征捕捉方法 explicitFeatureSnap。使用该参数时，它会加强特征面两边网格贴合，即内部区域和外部区域，这可能会导致特征面处网格歪斜率上升。

一般情况下，snapControls 采用下面的设置就已经能够得到相对可以接受的网格了：

```
snapControls
{
    explicitFeatureSnap true;
    implicitFeatureSnap false;
    multiRegionFeatureSnap false;

    tolerance        5;
    nSolveIter       100;
    nSmoothPatch     0;
    nSmoothInternal  0;
    nRelaxIter       8;
    nFeatureSnapIter 10;
    nFaceSplitInterval 5;
}
```

在划分边界层时，需要在全局参数设置中激活边界层划分功能，即将 addLayers 值设置为 true。边界层配置参数在 addLayersControls 子字典中设置，其参数类型可分为基本参数与高级控制参数：

关键词	含义	备注
relativeSizes	是否采用相对临近单元尺寸比值	设置为 true 时，边界层厚度参数值为相对于邻近曲面上的体网格单元大小的比值； 设置为 false 时，边界层厚度参数值直接由绝对单位的值 (单位 [m])
expansionRatio	边界层膨胀比	两个相邻层的厚度比，该值越大，各层间的高度差越大
finalLayerThickness	边界层最后一层厚度	确保边界层最后一层网格不大于该值
firstLayerThickness	边界层第一层的厚度	指定距离曲面最近的边界层的高度，确保边界层第一层网格不大于该值
thickness	边界层总厚度	所有边界层的最大厚度。设置较大的边界层总厚度值会导致体网格收缩位移相应增大，体网格变形量增加将导致网格质量降低。 当网格质量小于用户设置网格质量控制参数时，系统将取消此处边界层网格划分。
minThickness	最小总层厚度	所有边界层的总体最小厚度。若边界层挤出区域厚度小于该值，则该区域将不会生成边界层
nGrow	最大取消边界层单元数	指定未设置边界层的相邻面相交处边界层的过渡层数。 这有助于将边界层过渡到特征边附近
nSurfaceLayers	边界层层数	需在指定面设置边界层基本参数中输入，为强制性参数
maxFaceThicknessRatio	表面网格最大纵横比	当要在高度扭曲的单元上（特别是在角落）生成边界层时， 纵横比高于此值的单元上边界层停止生成，以保证边界层网格质量
featureAngle	边界层最大面夹角	当两个曲面之间的法向夹角小于参数 featureAngle 值时， 允许两个曲面的相交边处体网格向域内收缩，形成边界层划分区域
slipFeatureAngle	边界层滑移角度	使边界层面边缘处顶点延边界面滑动，以保证边界层在边缘处的划分空间。 建议使用 70-80 之间的最佳值来限制层的滑动，默认值为 featureAngle 的一半
nBufferCellsNoExtrude	边界层终止面的缓冲单元数	为边界层终止端创建缓冲收缩区，即逐渐降低边界层数。 设置值小于 0，则表示在终止端立即停止边界层
nSmoothThickness	边界层厚度平滑迭代次数	边界层网格生成前需要根据投影厚度值收缩现有体网格，可通过 nSmoothThickness 值设置投影厚度值的迭代次数
nSmoothSurfaceNormals	边界层曲面法线平滑迭代次数	边界层网格生成前，需要根据曲面法线方向收缩现有体网格，可通过 nSmoothSurfaceNormals 指定曲面法线平滑迭代次数
minMedianAxisAngle	拾取中间轴点的角度	这指定用于拾取中间轴点的角度。建议值为 90 度
maxThicknessToMedialRatio	层厚度与中间轴长度的最大比率	当比率大于指定值时，层生长减少。建议值为 0.3
nRelaxIter	体网格收缩迭代次数	单步投影厚度值的迭代中，系统需根据投影厚度值计算体网格收缩松弛系数。该值越大越有利于提高体网格网格质量。建议值为 5
nSmoothNormals	体网格平滑迭代次数	体网格收缩时，可通过 nSmoothNormals 指定体网格间移动方向的迭代次数。 该值越高，体网格间平滑性越好，但网格划分时间越长。建议值为 3
nLayerIter	边界层添加的最大总迭代次数	如果达到此迭代次数，边界层网格划分将立即停止，并保留最后一次迭代生成的边界层。建议值为 50-60
nRelaxedIter	宽松质量控制标准的起始迭代次数	划分边界层网格时会优先使用 meshQualityControls 中基础质量控制参数，检测网格是否满足要求。 若边界层添加算法迭代次数达到用户设置 nRelaxedIter 参数值后，网格依然不能达到质量控制要求， 则在此后的迭代中软件将采用用户设置的宽松质量控制标准值。参数建议值为 20

meshQualityControls 关键词中设置网格质量控制参数。在 snappyHexMesh 执行全局参数控制、面贴合、局部细化以及边界层生成时，程序都会依据网格质量控制参数不断调整网格迭代，同时，当网格位移或拓扑更改操作导致单元或面网格质量降低时，可根据控制参数撤消移动或拓扑更改操作以将网格还原为之前满足网格质量标准的状态。在该关键词内需要设定的内容如下：

关键词	含义	常见赋值
maxNonOrtho	最大非正交角	65
maxBoundarySkewness	最大边界面网格偏斜度	20
maxInternalSkewness	最大内部面网格偏斜度	4
maxConcave	最大凹度	80
minFlatness	最小平整值	0.5
minVol	最小单元体积	1e-13
minArea	最小网格面面积	-1
minTwist	最小面扭曲	0.05
minDeterminant	最小归一化单元行列式值	0.001
minFaceWeight	相邻网格间面权重最小值	0.05
minVolRatio	相邻网格间的最小体积膨胀率	0.01
minTriangleTwist	最小三角单元扭曲值	-1
minTetQuality		1e-30
nSmoothScale	每次网格缩放恢复迭代时的平滑次数	4
errorReduction	误差点处的缩放位移量	0.75
relaxed	相对宽松的上述标准列表	

- maxNonOrtho 参数指定允许的最大非正交角，其通过计算相邻两单元中心点向量与公共面法向量的夹角，此值为 0 时表示相邻两个网格完全正交。默认参考值为 65，当设置为 180 时，表示关闭此项控制。该参数是衡量网格质量的主要指标之一。

- maxBoundarySkewness 参数指定边界面网格允许的最大偏斜度。其定义一个面或体与理想几何 (即等边或等角) 的接近程度。默认参考值为 20，当设置小于 0 的值时，表示关闭此项控制。
- maxInternalSkewness 参数计算方式与最大边界面网格偏斜度一致，不过其主要测试内部网格质量。默认参考值为 4，当设置小于 0 的值时，表示关闭此项控制。该参数是衡量网格质量的主要指标之一。
- maxConcave 参数用于检查构成面的内凹角度，以允许低于该角度的凹面。0 表示直面，小于 0 表示凸面。默认参考值为 80，当设置为 180 时，表示关闭此项控制。
- minFlatness 最小投影面积和实际面积的比值，该参数值为 1 时，表示检测面为平面。默认参数为 0.5，设置为-1 时，表示禁用此项控制。
- minVol 参数为允许最小金字塔单元体积，其为网格绝对体积参数（单位 m3），默认参数为 1e-13。设置为较大的负值时（例如-1e30），表示禁用此项控制。该参数是衡量网格质量的主要指标之一。
- minArea 该参数为允许最小网格面的面积，默认为-1，该参数设置为负值时，表示禁用此项控制。
- minTwist 使用面中心将面分解为三角形单元，并通过相邻两个单元中心点向量与分解后三角形面法向量的点积计算面扭曲值。默认参考值为 0.05，当设置参数小于-1 时，表示关闭此项控制。
- minDeterminant 通过计算每一个六面体的雅可比行列式值，然后标准化行列式的矩阵来表征单元的变形。参数值取值范围为 0 到 1，参数值设置为 1 表示只允许有理想六面体网格；如果某单元行列式的值为 0，则这个立方体有一个或多个退化的边。参数值设置小于或等于 0 表示允许有负体积单元，默认参数值为 0.001。
- minFaceWeight 其定义了面相对于相邻单元间中心的相对位置（正交时为 0.5），计算方法是先计算出单元中心到公共面中心长度 L1，再计算出相邻单元中心到公共面中心长度 L2，面权重值等于 L1 与 L2 的最小值除以 L1 与 L2 之和。较小的面权重值表示相邻网格尺寸相差较大。参数值取值范围为 0 到 0.5，默认参考值为 0.05。
- minVolRatio 参数指定允许相邻网格间的最小体积膨胀率, 参数值取值范围为 0 到 1，默认参考值为 0.01。较大的比值会导致插值结果误差较大。
- minTriangleTwist 参数表示允许最小三角单元扭曲值，通过使用面中心将面分解为三角形单元，然后依据相邻的三角形单元法向量的点积计算出三角形面扭曲值。默认参数值为-1，表示禁用此功能。若参数值大于 0，则启用此功能项。其主要目的为确保生成网格与 Fluent 网格的兼容性。
- minTetQuality 通过网格单元中心和面中心将单元分解为四面体，然后根据圆周半径（Rc）和四面体体积（Vtet）计算四面体单元质量。对于一些跟踪算例时（如流线计算），该参数需要设置为一个较小正值，以确保内部单元质量检查正常运行。默认参数值为 1e-30。
- snappyHexMesh 网格划分过程中可将局部网格缩放到之前网格质量满足标准的状态。可通过 nSmoothScale 参数指定每次网格缩放恢复迭代时的平滑次数，参数默认值为 4。
- errorReduction 参数同 nSmoothScale 参数一样应用于网格缩放恢复迭代，用户可通过该参数减小误差点处的缩放位移量，参数默认值为 0.75。
- relaxed 在划分边界层网格时会优先使用 meshQualityControls 中基础质量控制参数，检测网格是否满足要求。若边界层添加算法迭代次数达到用户设置 nRelaxedIter 参数值后，网格依然不能达到质量控制要求，则在此后的迭代中软件将采用设置的宽松质量控制标准值，以提高边界层网格的覆盖率。

1.3 网格检查与索引重排

在通过 blockMesh 或 snappyHexMesh 生成网格之后，可以在算例目录下运行下面的命令来查看网格质量是否合格：

```
checkMesh
```

参考OpenFOAM 工具详解 - renumberMesh。对于一个 $N \times N$ 的对称矩阵 $\mathbf{A}$，第 i 行的 bandwith 定义为

$$\beta_i(\mathbf{A}) = \min\{j \mid a_{i,j} \neq 0\}$$

而矩阵整体的 bandwidth 定义为

$$\beta(\mathbf{A}) = \max\{\beta_i(\mathbf{A}) \mid 1 \leq i \leq N\} = \max\{|i-j| \mid a_{i,j} \neq 0\}$$

矩阵的 profile 定义为

$$\sum_i \beta_i(\mathbf{A})$$

因此，bandwidth 和 profile 反映了对称矩阵中非零元素在对角线周围的聚集程度，降低 bandwidth 和 profile 对稀疏矩阵求解的收敛性与收敛速度有很大帮助。

一般在生成完网格之后，直接如下运行 renumberMesh 命令即可：

```
renumberMesh -overwrite
```

或者通过并行的方式来运行该命令：

```
mpirun -np 4 renumberMesh -overwrite
```

这会使用默认的 CuthillMcKee 方法进行重排。为了查看重排之后的 CellID 分布情况，可以使用 foamToVTK 命令生成 vtk 文件，然后用 paraview 打开该文件即可查看 CellID 分布。

1.4 旋转轴对称算例的网格生成

首先按照一般的二维算例生成单层网格，所用的 blockMeshDict 如下：

```
backgroundMesh
{
    length 25;
    rA    0.5; // radius of inlet patch
    rB      4; // outer radius
    lengthCells 250;
    rAcells 10;
    rBcells 50;
}

convertToMeters 1;

vertices
(
    (                   0                0 -1)
    ($!backgroundMesh/length             0 -1)
    (                   0 $!backgroundMesh/rA -1)
    ($!backgroundMesh/length $!backgroundMesh/rA -1)
    (                   0 $!backgroundMesh/rB -1)
    ($!backgroundMesh/length $!backgroundMesh/rB -1)

    (                   0                0  0)
    ($!backgroundMesh/length             0  0)
    (                   0 $!backgroundMesh/rA 0)
    ($!backgroundMesh/length $!backgroundMesh/rA 0)
```

```
    (                   0 $!backgroundMesh/rB 0)
    ($!backgroundMesh/length $!backgroundMesh/rB 0)
);

blocks
(
    hex (0 1 3 2 6 7 9 8)
    ($!backgroundMesh/lengthCells $!backgroundMesh/rAcells 1)
    simpleGrading (1 1 1)

    hex (2 3 5 4 8 9 11 10)
    ($!backgroundMesh/lengthCells $!backgroundMesh/rBcells 1)
    simpleGrading (1 1 1)
);

boundary
(
    inlet
    {
        type patch;
        faces
        (
            (0 6 8 2)
        );
    }

    front
    {
        type symmetry;
        faces
        (
            (6 7 9 8)
            (8 9 11 10)
        );
    }

    back
    {
        type symmetry;
        faces
        (
            (0 1 3 2)
            (2 3 5 4)
        );
    }

    atmosphere
    {
        type patch;
```

```
        faces
        (
            (2 8 10 4)
            (4 5 11 10)
            (5 3 9 11)
            (3 1 7  9)
        );
    }
);
```

得到该基础背景网格之后，通过 extrudeMesh 功能将网格转换为用于旋转轴对称算例的楔形网格，需要使用的 extrudeMeshDict 如下：

```
FoamFile
{
    format      ascii;
    class       dictionary;
    object      extrudeProperties;
}

constructFrom patch;
sourceCase "$FOAM_CASE";

sourcePatches (front);
exposedPatchName back;

extrudeModel      wedge;

sectorCoeffs
{
    axisPt          (0 0 0);
    axis            (1 0 0);
    angle           1;
}

flipNormals false;
mergeFaces false;
```

1.5 Salome

利用开源软件 Salome 可以生成六面体网格以及混合网格，借助 salomeToOpenFOAM.py 脚本来将网格转换为 OpenFOAM 所使用的格式。下载 Salome 的压缩包并解压后，进入文件夹执行下面的命令检查所缺失的库和应用：

```
./sat/sat config SALOME-9.11.0-native --check_system
```

根据其提示信息，用 apt 安装所需的库和应用，完成后运行

```
./salome
```

即可打开 Salome 界面。

在创建主要网格时，如果要生成纯六面体网格，需要按照如下步骤进行：

1. 在 Geometry 中创建边界 Group 作为后续网格的 patch；
2. 在 3D 设置使用 Hexahedron(i,j,k) 算法；
3. 为所需边界 Group 添加边界层；
4. 在 2D 设置使用 Quadrangle:Mapping；
5. 在 1D 设置使用 Wire Discretisation 并设置 Number of Segments；
6. 运算生成网格。

如果要生成四面体网格，可以使用 NETGEN 算法实现，同样也可以添加边界层网格。

特别的，如果要生成适用于 OpenFOAM 二维算例的单层网格，可以先在面 Geometry 的时候在各个 Edge 生成相应的 Group(包括这个面本身也要添加为一个 Group)，按照面网格离散得到基础网格，然后使用 extrude 功能拉伸得到单层网格，这样可以引入三棱柱网格，同时也会自动生成对应的面 Group 来作为边界 patch。

1.6 网格信息获取

1.6.1 cells

要获得每个 cell 的面集合的 labelListList，可以通过 mesh.cells() 得到。其计算获取该列表数据的方法如下：

```
cellList* cfPtr_ = new cellList(mesh.nCells());
cellList& cellFaceAddr = *cfPtr_;
const labelList& own = mesh.faceOwner(); // it's different with mesh.owner()
const labelList& nei = mesh.faceNeighbour(); // it's the same as mesh.neighbour()
const label nCells = mesh.nCells();

// 1. Count number of faces per cell (start)
// Construct the list to store the number of faces that
// make the cell. Initialize the number to be 0.
labelList ncf(nCells, 0);

// Loop over faceOwner and record the existence of the face corresponding
// to own cell. Note that boundary faces are included here.
forAll(own, facei)
{
    ncf[own[facei]]++;
}

// Loop over faceNeighbour and record the existence of the face
// corresponding to nei cell.
forAll(nei, facei)
{
    if (nei[facei] >= 0)
    {
        ncf[nei[facei]]++;
    }
}
// 1. Count number of faces per cell (end)
```

```
// 2. Size and fill cellFaceAddr (start)
// Use the number of faces to set the size of lists.
forAll(cellFaceAddr, celli)
{
    cellFaceAddr[celli].setSize(ncf[celli]);
}
ncf = 0; // reset the number to be 0 for index sweep later

// Loop over faceOwner to append the face id into the
// list of corresponding own cell.
forAll(own, facei)
{
    label celli = own[facei];

    cellFaceAddr[celli][ncf[celli]++] = facei;
}

// Loop over faceNeighbour to append the face id into the
// list of corresponding nei cell.
forAll(nei, facei)
{
    label celli = nei[facei];

    if (celli >= 0)
    {
        cellFaceAddr[celli][ncf[celli]++] = facei;
    }
}
// 2. Size and fill cellFaceAddr (end)
```

1.6.2 cellCells

要获得每个 cell 的面相邻 cell 集合的 labelListList，可以通过 mesh.cellCells() 得到。其计算获取该列表数据的方法如下：

```
// 1. Count number of internal faces per cell (start)
// Construct the list to record the number of face-neighbouring
// cells for every cell. Initial the number as 0.
labelList ncc(mesh.nCells(), 0);

// Obtain the owner cells list and neighbour cells list.
const labelList& own = mesh.faceOwner();
const labelList& nei = mesh.faceNeighbour();

// Loop over the internal faces and record the existence of
// own cell and nei cell in the list.
forAll(nei, facei)
{
```

```
    ncc[own[facei]]++;
    ncc[nei[facei]]++;
}
// 1. Count number of internal faces per cell (end)

// Create the storage
labelListList* ccPtr_ = new labelListList(ncc.size());
labelListList& cellCellAddr = *ccPtr_;

// 2. Size and fill cellFaceAddr (start)
// Use the number of cells to set the size of list for every cell.
forAll(cellCellAddr, celli)
{
    cellCellAddr[celli].setSize(ncc[celli]);
}
ncc = 0; // reset the number to be 0

// Loop over internal faces and append the face-neighbouring
// cells id into the corresponding list of the cell. The ncc
// will help to advance the index for every cell.
forAll(nei, facei)
{
    label ownCelli = own[facei];
    label neiCelli = nei[facei];

    cellCellAddr[ownCelli][ncc[ownCelli]++] = neiCelli;
    cellCellAddr[neiCelli][ncc[neiCelli]++] = ownCelli;
}
// 2. Size and fill cellFaceAddr (end)
```

1.6.3 edgeCells

要获得每条边所相邻的网格集合 labelListList，可以通过 mesh.edgeCells() 得到。其计算获取该列表数据的时候用到了 invertManyToMany 函数，该方法展开后具体如下：

```
labelListList* ecPtr_ = new labelListList(mesh.nEdges());
const label nEdges = mesh.nEdges();
const labelListList& cellEdges = mesh.cellEdges(); // edges id list that construct the cell
labelListList& edges = *ecPtr_;

// Construct a list to record the number of cells per edge.
labelList nCellsPerEdge(nEdges, 0);

// Loop over the cells to record the existence of the edges
// that construct the corresponding cell.
forAll(cellEdges, celli)
{
    const labelList& pEdges = cellEdges[celli];
```

```
    forAll(pEdges, j)
    {
        nCellsPerEdge[pEdges[j]]++;
    }
}

// Use the number of cells per edge to set the size of lists.
forAll(nCellsPerEdge, edgeI)
{
    edges[edgeI].setSize(nCellsPerEdge[edgeI]);
}
nCellsPerEdge = 0; // reset the number to be 0 for index sweep later

// Loop over cells to append the id into the lists
// of edges that construct the cell.
forAll(cellEdges, celli)
{
    const labelList& pEdges = cellEdges[celli];

    forAll(pEdges, j)
    {
        label edgeI = pEdges[j];

        edges[edgeI][nCellsPerEdge[edgeI]++] = celli;
    }
}
```

1.6.4 pointCells

要获得每个点所相邻的网格集合 labelListList，可以通过 mesh.pointCells() 得到。其计算获取该列表数据的方法如下：

```
// Count number of cells per point (start)
// Obtain the faces id list for every cell
const cellList& cf = mesh.cells();

// Construct the list to record the number of neighbouring cells
// for every point. Initialize the number as 0
labelList npc(mesh.nPoints(), 0);

forAll(cf, celli)
{
    // The id list of points that construct the cell
    const labelList curPoints = cf[celli].labels(mesh.faces());

    // Loop over the id list and mark the existence of the cell
    // for every corresponding point id
    forAll(curPoints, pointi)
    {
```

```
        label ptI = curPoints[pointi];

        npc[ptI]++;
    }
}
// Count number of cells per point (end)


// Size and fill cells per point (start)
// Construct the list to record the neighbouring cell id list for every point
labelListList* pcPtr_ = new labelListList(npc.size());
labelListList& pointCellAddr = *pcPtr_;

// Use the number of cells per point to set the size
forAll(pointCellAddr, pointi)
{
    pointCellAddr[pointi].setSize(npc[pointi]);
}
npc = 0; // reset the number to be 0

forAll(cf, celli)
{
    // The id list of points that construct the cell
    const labelList curPoints = cf[celli].labels(mesh.faces());

    // Loop over the id list and append the cell id into the corresponding
    // labelListlist. npc will help to advance the index for every point
    forAll(curPoints, pointi)
    {
        label ptI = curPoints[pointi];

        pointCellAddr[ptI][npc[ptI]++] = celli;
    }
}
// Size and fill cells per point (end)
```

第 2 章 热物理模型

每个调用热物理模型库的求解器均会构建一个具体的热物理模型类。每个组分都需要输入基本的热物理参数。他们必须以组分名作为关键词，其后是子字典，包含的参数主要有 specie、thermoDynamics、transport。下面是一个经典的设置案例：

```
thermoType
{
    type          hePsiThermo;
    mixture       pureMixture;
    transport     const;
    thermo        hConst;
    equationOfState perfectGas;
    specie        specie;
    energy        sensibleInternalEnergy;
}

mixture
{
    specie
    {
        molWeight     352; // 摩尔质量g/mol
    }
    thermodynamics
    {
        Cp            385; // J/(kg.K)
        Hf            0;
    }
    transport
    {
        mu            1.83e-5; // Pa.s
        Pr            1.155;   // 普朗特数
    }
}
```

2.1 thermoType

在 physicalProperties 文件中的 thermoType 关键词下指定运行中的热物理模型。它本身包含各种基础类，决定了热物理模型的代码框架。可用的热物理模型基类有：

- psiThermo：固定组分、基于可压缩性 $\psi = (RT)^{-1}$ 的热物理模型库，R 为理想气体常数，T 为温度，调用该类的求解器主要为可压缩求解器，例如 rhoCentralFoam、uncoupledKinematicParcelFoam 以及 coldEngineFoam。
- rhoThermo：固定组分、基于密度 ρ 的热物理模型库，调用该类的求解器主要为传热类求解器，例如 buoyantSimpleFoam、buoyantPimpleFoam、rhoPorousSimpleFoam、twoPhaseEulerFoam 以及 thermoFoam。
- psiReactionThermo：基于可压缩性 ψ 的附加反应的热物理模型库，调用该类的求解器主要为燃烧求解器，例如 sprayFoam、engineFoam、fireFoam、reactingFoam，以及一些拉格朗日求解器，例如 coalChemistryFoam。

- psiuReactionThermo：基于未燃气体可压缩性 ψu 的反应混合热物理模型库，调用该类的求解器主要为燃烧求解器，这些求解器的物理模型基于层流火焰速度以及回归变量，如 XiFoam、XiEngineFoam 和 PDRFoam。
- rhoReactionThermo：基于密度 ρ 的反应混合热物理模型库，调用该类的求解器主要为 chtMultiRegionFoam 以及一些燃烧求解器，如 chemFoam、rhoReactinFoam、rhoReactingBuoyantFoam，以及一些拉格朗日求解器，如 reactingParcelFoam 和 simpleReactingParcelFoam。
- multiPhaseMixtureThermo：多相流热物理模型库，调用该类的求解器主要为可压缩多相界面捕获求解器，如 compressibleInterFoam、compressibleMultiphaseInterFoam。

2.2 type

type 关键词用来指定具体的热物理模型库：

热物理模型库	使用场景
hePsiThermo	调用 fluidThermo，fluidReactionThermo 以及 psiThermo 的求解器需指定
heRhoThermo	调用 fluidThermo，fluidReactionThermo 以及 multiphaseMixtureThermo 的求解器需指定
heheuPsiThermo	调用 psiuReactionThermo 的求解器需指定

2.2.1 hePsiThermo

这是一个基于压缩性的混合物能量类，它决定了使用 hePsiThermo 的求解器中 thermo.correct() 函数所做的内容：

```
template<class BasicPsiThermo, class MixtureType>
void Foam::hePsiThermo<BasicPsiThermo, MixtureType>::calculate()
{
    const scalarField& hCells = this->he_; // 焓或内能
    const scalarField& pCells = this->p_; // 压强

    scalarField& TCells = this->T_.primitiveFieldRef();   // 温度
    scalarField& CpCells = this->Cp_.primitiveFieldRef(); // 定压比热容
    scalarField& CvCells = this->Cv_.primitiveFieldRef(); // 定体比热容
    scalarField& psiCells = this->psi_.primitiveFieldRef(); // 可压缩性
    scalarField& muCells = this->mu_.primitiveFieldRef(); // 动力粘度
    scalarField& kappaCells = this->kappa_.primitiveFieldRef(); // 热导率

    forAll(TCells, celli) // 在温度场内的网格单元内的计算
    {
        const typename MixtureType::thermoMixtureType& thermoMixture = this->cellThermoMixture(celli
            );
        const typename MixtureType::transportMixtureType& transportMixture = this->
            cellTransportMixture(celli, thermoMixture);

        TCells[celli] = thermoMixture.THE(hCells[celli], pCells[celli], TCells[celli]); // 由焓或内
            能计算温度T，该函数定义在thermoI.H文件中
        CpCells[celli] = thermoMixture.Cp(pCells[celli], TCells[celli]);      // 根据thermo设置方式计
            算定压比热容Cp
        CvCells[celli] = thermoMixture.Cv(pCells[celli], TCells[celli]);      // 根据thermo设置方式计
            算定体比热容Cv
```

```
    psiCells[celli] = thermoMixture.psi(pCells[celli], TCells[celli]);   // 根据thermo设置方式计
        算可压缩性psi
    muCells[celli] = transportMixture.mu(pCells[celli], TCells[celli]);  // 根据transport设置方
        式计算动力粘度mu
    kappaCells[celli] = transportMixture.kappa(pCells[celli], TCells[celli]); // 根据transport设
        置方式计算热导率kappa
}

volScalarField::Boundary& pBf = this->p_.boundaryFieldRef();  // 边界处的压强
volScalarField::Boundary& TBf = this->T_.boundaryFieldRef();  // 边界处的温度
volScalarField::Boundary& CpBf = this->Cp_.boundaryFieldRef(); // 边界处的定压比热容
volScalarField::Boundary& CvBf = this->Cv_.boundaryFieldRef(); // 边界处的定体比热容
volScalarField::Boundary& psiBf = this->psi_.boundaryFieldRef(); // 边界处的可压缩性
volScalarField::Boundary& heBf = this->he().boundaryFieldRef(); // 边界处的焓或内能
volScalarField::Boundary& muBf = this->mu_.boundaryFieldRef(); // 边界处的动力粘度
volScalarField::Boundary& kappaBf = this->kappa_.boundaryFieldRef(); // 边界处的热导率

forAll(this->T_.boundaryField(), patchi) // 在温度场的边界处的计算
{
    fvPatchScalarField& pp = pBf[patchi];     // 边界处的压强
    fvPatchScalarField& pT = TBf[patchi];     // 边界处的温度
    fvPatchScalarField& pCp = CpBf[patchi];   // 边界处的定压比热容
    fvPatchScalarField& pCv = CvBf[patchi];   // 边界处的定体比热容
    fvPatchScalarField& ppsi = psiBf[patchi]; // 边界处的可压缩性
    fvPatchScalarField& phe = heBf[patchi];   // 边界处的焓或内能
    fvPatchScalarField& pmu = muBf[patchi];   // 边界处的动力粘度
    fvPatchScalarField& pkappa = kappaBf[patchi]; // 边界处的热导率

    if (pT.fixesValue()) // 如果边界处的温度设置为了固定值
    {
        forAll(pT, facei)
        {
            const typename MixtureType::thermoMixtureType& thermoMixture = this->
                patchFaceThermoMixture(patchi, facei);
            const typename MixtureType::transportMixtureType& transportMixture = this->
                patchFaceTransportMixture(patchi, facei, thermoMixture);

            phe[facei] = thermoMixture.HE(pp[facei], pT[facei]);    // 根据thermo的设置方式计算边
                界处的焓或内能
            pCp[facei] = thermoMixture.Cp(pp[facei], pT[facei]);    // 根据thermo的设置方式计算边
                界处的定压比热容
            pCv[facei] = thermoMixture.Cv(pp[facei], pT[facei]);    // 根据thermo的设置方式计算边
                界处的定体比热容
            ppsi[facei] = thermoMixture.psi(pp[facei], pT[facei]); // 根据thermo的设置方式计算边
                界处的可压缩性
            pmu[facei] = transportMixture.mu(pp[facei], pT[facei]); // 根据tansport的设置方式计算
                边界处的动力粘度
            pkappa[facei] = transportMixture.kappa(pp[facei], pT[facei]); // 根据transport的设置
                方式计算边界处的热导率
```

```
            }
        }
        else   // 如果边界处的温度没有设置为固定值
        {
            forAll(pT, facei)
            {
                const typename MixtureType::thermoMixtureType& thermoMixture = this->
                    patchFaceThermoMixture(patchi, facei);

                const typename MixtureType::transportMixtureType& transportMixture = this->
                    patchFaceTransportMixture(patchi, facei, thermoMixture);

                pT[facei] = thermoMixture.THE(phe[facei], pp[facei], pT[facei]); // 根据thermo的设置
                    方式计算边界处的由焓或内能得到的温度
                pCp[facei] = thermoMixture.Cp(pp[facei], pT[facei]);      // 根据thermo的设置方式计算
                    边界处的定压比热容
                pCv[facei] = thermoMixture.Cv(pp[facei], pT[facei]);      // 根据thermo的设置方式计算
                    边界处的定体比热容
                ppsi[facei] = thermoMixture.psi(pp[facei], pT[facei]);    // 根据thermo的设置方式计算
                    边界处的可压缩性
                pmu[facei] = transportMixture.mu(pp[facei], pT[facei]);   // 根据tansport的设置方式计
                    算边界处的动力粘度
                pkappa[facei] = transportMixture.kappa(pp[facei], pT[facei]); // 根据transport的设置
                    方式计算边界处的热导率
            }
        }
    }
}
```

2.3 mixture

mixture 关键词指定混合组分。无反应的热物理模型库通常使用 pureMixture，也即固定组分。当指定 pureMixture 的时候，相关的热物理模型系数在 mixture 子字典中指定。

2.3.1 pureMixture

```
template<class ThermoType>
Foam::pureMixture<ThermoType>::pureMixture
(
    const dictionary& thermoDict,
    const fvMesh& mesh,
    const word& phaseName
)
:
    basicMixture(thermoDict, mesh, phaseName),
    mixture_(thermoDict.subDict("mixture"))
{}
```

```
template<class ThermoType>
void Foam::pureMixture<ThermoType>::read(const dictionary& thermoDict)
{
    mixture_ = ThermoType(thermoDict.subDict("mixture")); // 读取thermoType字典中mixture关键词的设置
}
```

2.4 transport

transport 传递模型需要计算动力粘度 μ 、热导率 κ 、扩散率 (用于内能方程或焓方程中) α 。可选的 transport 模型有:

transport 模型	含义
const	粘度 μ 为常数，普朗特数由 $P_r = C_p\mu/\kappa$ 来计算，需要指定 μ 和 P_r
sutherland	通过温度 T 和两个系数 A_s, T_s 的函数来计算粘度 μ ，即 $\mu = \frac{A_s\sqrt{T}}{1+T_s/T}$ ，需要指定 A_s 和 T_s
polynomial	从一个可以指定任意阶数的函数通过温度 T 来计算粘度 μ 和热导率 ε ，即 $\mu = \sum_{i=0}^{N-1} a_i T^i$
logPolynomial	从任意阶数依据 $\ln(T)$ 来计算 $\ln(\mu)$ 和 $\ln(k)$ ，即 $\ln(\mu) = \sum_{i=0}^{N-1} a_i (\ln T)^i$
icoTabulated	使用粘度和热导率的非均匀列表数据作为温度函数
WLF	计算 $\ln(\mu)$ 和 $\ln(\kappa)$ 作为 $\ln(T)$ 的函数由任意阶 N 的多项式通过指数来计算 μ, κ ，即 $\ln(\mu) = \sum_{i=0}^{N-1} a_i [\ln(T)]^i$

2.4.1 const

```
template<class Thermo>
void Foam::constTransport<Thermo>::constTransport::write(Ostream& os) const
{
    os  << this->name() << endl;
    os  << token::BEGIN_BLOCK << incrIndent << nl;

    Thermo::write(os);

    dictionary dict("transport");
    dict.add("mu", mu_);
    if (constPr_)
    {
        dict.add("Pr", 1.0/rPr_);
    }
    else
    {
        dict.add("kappa", kappa_);
    }

    os  << indent << dict.dictName() << dict;

    os  << decrIndent << token::END_BLOCK << nl;
}
```

上面的代码表明，设置为 const 时将会查看 transport 字典中的 mu 关键词，并会检查是否存在 Pr 关键词，如果存在该关键词，则读取 Pr 关键词的值来计算普朗特数的倒数 rPr；如果不存在该关键词，则会读取 kappa 关键词。

```
template<class Thermo>
inline Foam::scalar Foam::constTransport<Thermo>::mu
(
    const scalar p,
    const scalar T
) const
{
    return mu_;  // 直接返回mu值
}


template<class Thermo>
inline Foam::scalar Foam::constTransport<Thermo>::kappa
(
    const scalar p,
    const scalar T
) const
{
    return constPr_ ? this->Cp(p, T)*mu(p, T)*rPr_ : kappa_; // 判断是否设置Pr关键词计算kappa
}


template<class Thermo>
inline void Foam::constTransport<Thermo>::operator+=
(
    const constTransport<Thermo>& st
)
{
    scalar Y1 = this->Y();

    Thermo::operator+=(st);

    if (mag(this->Y()) > small)
    {
        if ( constTransport<Thermo>::debug && (constPr_ != st.constPr_) )
        {
            FatalErrorInFunction
                //...
        }

        Y1 /= this->Y();             // 第一组分的质量分数
        scalar Y2 = st.Y()/this->Y(); // 第二组分的质量分数

        mu_ = Y1*mu_ + Y2*st.mu_;
        rPr_ = constPr_ ? 1/(Y1/rPr_ + Y2/st.rPr_) : NaN;
        kappa_ = constPr_ ? NaN : Y1*kappa_ + Y2*st.kappa_;
```

```
    }
}
```

从上面的代码可知，设置为 const 时，单组分的热传导率的计算方式为

$$\kappa = \begin{cases} \frac{c_p \mu}{\mathrm{Pr}}, & \text{if Pr exists} \\ \kappa_0, & \text{if Pr not exists} \end{cases}$$

双组分的动力粘度计算方式为

$$\mu = Y_1 \mu_1 + Y_2 \mu_2$$

如果设置了 Pr 值，那么 Pr 值的倒数的计算方式为

$$\frac{1}{\mathrm{Pr}} = \frac{1}{Y_1 \mathrm{Pr} + Y_2 \mathrm{Pr}}$$

如果没有设置 Pr 值，而是设置了 kappa 值，则双组分的热传导率的计算方式为

$$\kappa = Y_1 \kappa_1 + Y_2 \kappa_2$$

2.4.2 sutherland

使用 Sutherland's formula 来计算粘性和热传导系数：

$$\mu = \frac{A_s \sqrt{T}}{1 + T_s / T}, \quad \kappa = \mu C_v (1.32 + 1.77 R / C_v)$$

其中需要提供 Sutherland 常数 A_s ，单位为 $\mathrm{kg/(m \cdot s \cdot K^{1/2})}$ ，以及 Sutherland 温度 T_s ，单位为 K 。

一个设置为空气设置 sutherlandTransport 的示例如下：

```
transport
{
    As    1.458e-06;
    Ts    110.4;
}
```

2.4.3 polynomial

使用关于温度的多项式来计算动力粘度和热传导系数，默认是 7 阶多项式：

$$\mu = m_0 + m_1 T + m_2 T^2 + m_3 T^3 + m_4 T^4 + m_5 T^5 + m_6 T^6 + m_7 T^7$$

$$\kappa = k_0 + k_1 T + k2 T^2 + k_3 T^3 + k_4 T^4 + k_5 T^5 + k_6 T^6 + k_7 T^7$$

设置 polynomialTransport 的示例如下：

```
transport
{
    muCoeffs<8>     (1000 -0.05 0.003 0 0 0 0 0);
    kappaCoeffs<8>  (2000 -0.15 0.023 0 0 0 0 0);
}
```

2.4.4 logPolynomial

使用关于温度的自然对数的多项式来计算动力粘度和热传导系数，默认是 7 阶多项式：

$$\ln(\mu) = m_0 + m_1 \ln(T) + m_2 \ln(T)^2 + m_3 \ln(T)^3 + m_4 \ln(T)^4 + m_5 \ln(T)^5 + m_6 \ln(T)^6 + m_7 \ln(T)^7$$

$$\ln(\kappa) = k_0 + k_1 \ln(T) + k2 \ln(T)^2 + k_3 \ln(T)^3 + k_4 \ln(T)^4 + k_5 \ln(T)^5 + k_6 \ln(T)^6 + k_7 \ln(T)^7$$

设置 logPolynomialTransport 的示例如下：

```
transport
{
    muLogCoeffs<8>    (1000 -0.05 0.003 0 0 0 0 0);
    kappaLogCoeffs<8> (2000 -0.15 0.023 0 0 0 0 0);
}
```

利用 logPolynomial 可以实现粘度的逆幂律模型：

$$\mu = \mu_{\text{ref}} \left(\frac{T}{T_{\text{ref}}} \right)^{\omega} \quad \Rightarrow \quad \ln \mu = \ln \frac{\mu_{\text{ref}}}{T_{\text{ref}}^{\omega}} + \omega \ln T$$

$$\text{Pr} = \frac{C_p \mu}{\lambda} \quad \Rightarrow \quad \ln \lambda = \ln \frac{C_p}{\text{Pr}} + \ln \mu = (\ln \frac{C_p}{\text{Pr}} + \ln \frac{\mu_{\text{ref}}}{T_{\text{ref}}^{\omega}}) + \omega \ln T$$

2.4.5 WLF

使用 Williams-Landel-Ferry 模型来计算聚合物熔体的粘性：

$$\mu = \mu_0 \exp \left(- \frac{C_1 (T - T_r)}{C_2 + T - T_r} \right)$$

其中 μ_0 为参考动力粘度，T_r 为参考温度，C_1 和 C_2 为 WLF 常数。热传导系数将通过一个固定的 Prandtl 数来计算。

设置 WLFTransport 的示例如下：

```
transport
{
    mu0   50000;
    Tr    416;
    C1    20.4;
    C2    101.6;
    Pr    10000;
}
```

2.5 thermo

thermo 热力模型参数和比热容 C_p 有关，别的相关特性可以从比热容计算而来。可选的模型有：

thermo 模型	含义
hConst	指定 C_p 以及 H_f 为常量
eConst	指定 C_ν 以及 H_f 为常量
eIcoTabulated	通过插值 (T, C_p) 键值对的非均匀数据列表计算 C_ν
hIcoTabulated	通过插值 (T, C_p) 键值对的均匀数据列表计算 C_ν
ePolynomial	通过任意 N 阶多项式计算 C_ν 作为温度的函数，即 $C_\nu = \sum_{i=0}^{N-1} a_i T^i$
hPower	将 C_p 作为温度的指数来计算，即 $C_p = c_0 \left(\frac{T}{T_{\text{ref}}} \right)^{n_0}$
ePower	将 C_ν 作为温度的指数来计算，即 $C_\nu = c_0 \left(\frac{T}{T_{\text{ref}}} \right)^{n_0}$
janaf	从 JANAF 热力学表来选择参数，以温度的函数来计算

着重指出温度的计算方式如下：

```
template<class Thermo, template<class> class Type>
template<class ThermoType, class FType, class dFdTType, class LimitType>
inline Foam::scalar Foam::species::thermo<Thermo, Type>::T // 计算温度
(
    const ThermoType& thermo,
    const scalar f,  // 混合物的能量
    const scalar p,  // 混合物的压强
    const scalar T0, // 温度的初始猜测值
    FType F,         // 温度表示的能量函数
    dFdTType dFdT,   // 温度表示的能量函数对温度的导函数
    LimitType limit, // 温度范围限制
    const bool diagnostics
)
{
    if (T0 < 0)
    {
        FatalErrorInFunction
            << "Negative initial temperature T0: " << T0
            << abort(FatalError);
    }

    scalar Test = T0;
    scalar Tnew = T0;
    scalar Ttol = T0*tol_;
    int   iter = 0;

    if (diagnostics)
    {
        const unsigned int width = IOstream::defaultPrecision() + 8;

        InfoInFunction
            << "Energy -> temperature conversion failed to converge:" << endl;
        Pout<< setw(width) << "iter"
            << setw(width) << "Test"
            << setw(width) << "e/h"
            << setw(width) << "Cv/p"
            << setw(width) << "Tnew"
            << endl;
    }
    do   // 通过欧拉迭代法求解温度场
    {
        Test = Tnew;
        Tnew = (thermo.*limit)(Test - ((thermo.*F)(p, Test) - f)/(thermo.*dFdT)(p, Test));

        if (diagnostics)
        {
            const unsigned int width = IOstream::defaultPrecision() + 8;
```

```
            Pout<< setw(width) << iter
                << setw(width) << Test
                << setw(width) << ((thermo.*F)(p, Test))
                << setw(width) << ((thermo.*dFdT)(p, Test))
                << setw(width) << Tnew
                << endl;
        }

        if (iter++ > maxIter_)
        {
            if (!diagnostics)
            {
                T(thermo, f, p, T0, F, dFdT, limit, true);
            }

            FatalErrorInFunction
                << "Maximum number of iterations exceeded: " << maxIter_
                << abort(FatalError);
        }

    } while (mag(Tnew - Test) > Ttol);

    return Tnew;
}
```

2.5.1 hConst

这是一个基于焓的热力学包，它在恒压条件下使用恒定热容，满足：

$$h = c_p(T - T_{\text{ref}}) + H_{s_{\text{ref}}}$$

```
template<class EquationOfState>
Foam::hConstThermo<EquationOfState>::hConstThermo(const dictionary& dict)
:
    EquationOfState(dict),
    Cp_(dict.subDict("thermodynamics").lookup<scalar>("Cp")), // 定压比热容
    Hf_(dict.subDict("thermodynamics").lookup<scalar>("Hf")), // 生成热（生成焓）
    Tref_(dict.subDict("thermodynamics").lookupOrDefault<scalar>("Tref", Tstd)), // 线性化的参考温度
    Hsref_(dict.subDict("thermodynamics").lookupOrDefault<scalar>("Hsref", 0)) // 参考显焓，围绕它进
        行线性化
{}

template<class EquationOfState>
void Foam::hConstThermo<EquationOfState>::write(Ostream& os) const
{
    EquationOfState::write(os);

    dictionary dict("thermodynamics");
    dict.add("Cp", Cp_);
```

```
    dict.add("Hf", Hf_);
    if (Tref_ != Tstd)
    {
        dict.add("Tref", Tref_);
    }
    if (Hsref_ != 0)
    {
        dict.add("Hsref", Hsref_);
    }
    os  << indent << dict.dictName() << dict;
}
```

设置为 hConst 时将会读取 thermodynamics 字典中的 Cp 和 Hf 关键词；如果存在 Tref 关键词也会读取，否则默认值使用 Tstd；如果存在 Hsref 关键词也会读取，否则默认值使用 0。

```
template<class EquationOfState>
inline Foam::scalar Foam::hConstThermo<EquationOfState>::limit // 将温度限制在从Tlow到Thigh的范围内
(
    const scalar T
) const
{
    return T;
}

template<class EquationOfState>
inline Foam::scalar Foam::hConstThermo<EquationOfState>::Cp // 定压比热容
(
    const scalar p,
    const scalar T
) const
{
    return Cp_ + EquationOfState::Cp(p, T); // 其中EquationOfState::Cp表示由于状态方程而产生的定压比
        热容的贡献
}

template<class EquationOfState>
inline Foam::scalar Foam::hConstThermo<EquationOfState>::Hs // 显焓
(
    const scalar p,
    const scalar T
) const
{
    return Cp_*(T - Tref_) + Hsref_ + EquationOfState::H(p, T); // 其中EquationOfState::H表示由于状
        态方程而产生的焓的贡献
}

template<class EquationOfState>
inline Foam::scalar Foam::hConstThermo<EquationOfState>::Ha // 绝对焓
(
    const scalar p,
```

```
    const scalar T
) const
{
    return Hs(p, T) + Hf();
}

template<class EquationOfState>
inline Foam::scalar Foam::hConstThermo<EquationOfState>::Hf() const // 生成焓
{
    return Hf_;
}

template<class EquationOfState>
inline Foam::scalar Foam::hConstThermo<EquationOfState>::S // 熵
(
    const scalar p,
    const scalar T
) const
{
    return Cp_*log(T/Tstd) + EquationOfState::Sp(p, T); // 其中EquationOfState::Sp表示由于状态方程而
        产生的熵对Cp/T积分的贡献
}

template<class EquationOfState>
inline Foam::scalar Foam::hConstThermo<EquationOfState>::Gstd // 混合物在标准态下的吉布斯自由能
(
    const scalar T
) const
{
    return Cp_*(T - Tref_) + Hsref_ + Hf() - Cp_*T*log(T/Tstd);
}

template<class EquationOfState>
inline Foam::scalar Foam::hConstThermo<EquationOfState>::dCpdT // 定压比热容关于温度的导数
(
    const scalar p,
    const scalar T
) const
{
    return 0;
}
```

2.6 equationOfState

状态方程 equationOfState 可选的有：

状态方程	含义
rhoConst	密度为常量
perfectGas	理想气体，即 $\rho = \frac{1}{RT}p$
incompressiblePerfectGas	不可压缩理想气体，即 $\rho = \frac{1}{RT}p_{\text{ref}}$，其中 p_{ref} 为参考压力
perfectFluid	理想液体，即 $\rho = \frac{1}{RT}p + \rho_0$，其中 p_0 为 $T = 0$ 下的密度
linear	线性状态方程，即 $\rho = \varphi p + \rho_0$，其中 φ 为可压缩性
adiabaticPerfectFluid	绝热理想气体，即 $\rho = \rho_0 \left(\frac{p+B}{p_0 B}\right)^{1/\gamma}$，其中 ρ_0, p_0 为参考密度和参考压力，B 为模型常数
Boussinesq	布辛涅司克近似，即 $\rho = \rho_0\big(1 - \beta(T - T_0)\big)$，其中 β 表示体膨胀率，ρ_0 表示参考温度 T_0 下的参考密度
PengRobinsonGas	彭-罗宾森状态方程，即 $\rho = \frac{1}{zRT}p$，其中 $z = z(p, T)$
icoPolynomial	不可压缩多项式状态方程 $\rho = \sum_{i=0}^{N-1} a_i T^i$，其中 φ 为可压缩性
icoTabulated	提供不可压缩流体 (T, ρ) 键值对的数据列表
rhoTabulated	可压缩流体的统一表格数据，计算 ρ 作为 T 的函数
rPolynomial	液体或固体的倒数多项式状态方程，即 $\frac{1}{\rho} = C_0 + C_1 T + C_2 T^2 - C_3 p - C_4 pT$，其中的 C_i 为系数

2.6.1 perfectGas

理想气体状态方程，所使用的气体常数 R 直接从 specie 或者混合物的分子量得到，所以不需要额外设定：

$$\rho = \frac{p}{RT}$$

当 equationOfState 设置为 perfectGas 之后，通过它可以调用的函数如下所示：

```
template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::rho(scalar p, scalar T) const // 密度场
{
    return p/(this->R()*T); // 通过理想气体方程求解密度
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::H(scalar p, scalar T) const // 焓的贡献
{
    return 0;
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::Cp(scalar p, scalar T) const // 定压比热容的贡献
{
    return 0;
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::E(scalar p, scalar T) const // 能量贡献
{
    return 0;
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::Cv(scalar p, scalar T) const // 定体比热容贡献
{
    return 0;
}
```

```
template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::Sp(scalar p, scalar T) const // 熵对Cp/T积分的贡献
{
    return -this->R()*log(p/Pstd);
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::Sv(scalar p, scalar T) const // 熵对Cv/T积分的贡献
{
    NotImplemented;
    return 0;
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::psi(scalar p, scalar T) const // 可压缩性场
{
    return 1.0/(this->R()*T);
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::Z(scalar p, scalar T) const // 压缩因子
{
    return 1;
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::CpMCv(scalar p, scalar T) const // Cp与Cv的差
{
    return this->R();
}

template<class Specie>
inline Foam::scalar Foam::perfectGas<Specie>::alphav(scalar p, scalar T) const // 热膨胀的体积系数
{
    return 1/T;
}
```

2.7 specie

这是热物理性质类型的基类。组分的量只有一种选择 specie，后续需要进一步输入下面的信息：

- nMoles：摩尔数。仅仅在使用反应组分均一混合回归变量燃烧模型的时候起作用，否则其为 1；
- molWeight：摩尔质量 g/mol

```
Foam::specie::specie(const dictionary& dict)
:
    name_(dict.dictName()),
    Y_(dict.subDict("specie").lookupOrDefault("massFraction", 1.0)), // 质量分数
```

```
    molWeight_(dict.subDict("specie").lookup<scalar>("molWeight")) // 摩尔质量
{}
```

```
inline const word& specie::name() const // 组分名称
{
    return name_;
}

inline scalar specie::W() const // 摩尔质量
{
    return molWeight_;
}

inline scalar specie::Y() const // 组分的质量分数
{
    return Y_;
}

inline scalar specie::R() const // 个别气体常数
{
    return RR/molWeight_; // 通过气体常数除以该组分的摩尔质量得到
}
```

2.8 energy

可以对能量方程求解的变量进行指定，其可为内能 e 也可以为焓 h ，并可以选择是否包含热源 Δh_f 。其可以通过指定 energy 关键字来实现：

- sensibleEnthalpy：利用焓值求解温度；
- sensibleInternalEnergy：利用内能求解温度；
- absoluteEnthalpy：使用绝对焓表示包含热源的情况；
- absoluteInternalEnergy：使用绝对内能求解温度。

2.8.1 sensibleEnthalpy

这是一个热力学映射类，以使用关于显式焓的函数。

```
    static bool enthalpy()
    {
        return true;  // 使用焓
    }

    static word energyName()
    {
        return "h";  // 将energyName设置为焓h
    }

    // Heat capacity at constant pressure [J/kg/K]
    scalar Cpv  // 定压比热容
```

```
(
    const Thermo& thermo,
    const scalar p,
    const scalar T
) const
{
    return thermo.Cp(p, T);
}

// Sensible enthalpy [J/kg]
scalar HE  // 显内能
(
    const Thermo& thermo,
    const scalar p,
    const scalar T
) const
{
    return thermo.Hs(p, T);
}

//- Temperature from sensible enthalpy given an initial temperature T0
scalar THE  // 在给定初始温度T0的条件下，由显内能得到的温度
(
    const Thermo& thermo,
    const scalar h,
    const scalar p,
    const scalar T0
) const
{
    return thermo.THs(h, p, T0);
}
```

2.8.2 sensibleInternalEnergy

这是一个热力学映射类，以使用关于显式内能的函数。

```
static bool enthalpy()
{
    return false;  // 不使用焓
}

static word energyName()
{
    return "e";  // 将energyName设置为内能e
}

//- Heat capacity at constant volume [J/kg/K]
scalar Cpv  // 定体比热容
(
```

```
    const Thermo& thermo,
    const scalar p,
    const scalar T
) const
{
    return thermo.Cv(p, T); // 调用thermo设置模式的函数或者使用HtoEthermo.H文件中定义的函数
}

//- Sensible internal energy [J/kg]
scalar HE  // 显内能
(
    const Thermo& thermo,
    const scalar p,
    const scalar T
) const
{
    return thermo.Es(p, T); // 调用thermo设置模式的函数或者使用HtoEthermo.H文件中定义的函数
}

//- Temperature from sensible internal energy given an initial temperature T0
scalar THE  // 在给定初始温度T0的条件下，由显内能得到的温度
(
    const Thermo& thermo,
    const scalar e,
    const scalar p,
    const scalar T0
) const
{
    return thermo.TEs(e, p, T0); // 调用thermo模块模板函数
}
```

2.8.3 absoluteEnthalpy

这是一个热力学映射类，以使用关于绝对焓的函数。

```
static bool enthalpy()
{
    return true;  // 使用焓
}

static word energyName()
{
    return "ha";  // 将energyName设置为绝对焓he
}

// Heat capacity at constant pressure [J/kg/K]
scalar Cpv  // 定压比热容
(
    const Thermo& thermo,
```

```
    const scalar p,
    const scalar T
) const
{
    return thermo.Cp(p, T);
}

// Absolute enthalpy [J/kg]
scalar HE  // 绝对焓
(
    const Thermo& thermo,
    const scalar p,
    const scalar T
) const
{
    return thermo.Ha(p, T);
}

//- Temperature from absolute enthalpy given an initial temperature T0
scalar THE  // 给定初始温度T0，由绝对焓得到的温度
(
    const Thermo& thermo,
    const scalar h,
    const scalar p,
    const scalar T0
) const
{
    return thermo.THa(h, p, T0);
}
```

2.8.4 absoluteInternalEnergy

这是一个热力学映射类，以使用关于绝对内能的函数。

```
static bool enthalpy()
{
    return false; // 不使用焓
}

static word energyName()
{
    return "ea"; // 设置energyName为绝对内能ea
}

// Heat capacity at constant volume [J/kg/K]
scalar Cpv  // 定体比热容
(
    const Thermo& thermo,
    const scalar p,
```

```
    const scalar T
) const
{
    return thermo.Cv(p, T);
}

// Absolute internal energy [J/kg]
scalar HE  // 绝对内能
(
    const Thermo& thermo,
    const scalar p,
    const scalar T
) const
{
    return thermo.Ea(p, T);
}

//- Temperature from absolute internal energy given an initial temperature T0
scalar THE  // 给定初始温度T0，由绝对内能得到的温度
(
    const Thermo& thermo,
    const scalar e,
    const scalar p,
    const scalar T0
) const
{
    return thermo.TEa(e, p, T0);
}
```

第 3 章 边界条件

边界上的某个场值，或者梯度值，其计算方法可以用如下通式表示：

$$\phi_f = A_1\phi_c + B_1$$
$$\nabla\phi_f = A_2\phi_c + B_2$$

其中 ϕ_c 表示 ϕ 在邻近边界的网格中心的值，四个系数 A_1, B_1, A_2, B_2 分别通过 valueInternalCoeffs、valueBoundaryCoeffs、gradientInternalCoeffs、gradientBoundaryCoeffs 来表示。这四个参数将通过 fvm 来对求解的矩阵系统产生影响，例如 fvm::div 将会调用函数 valueInternalCoeffs() 和 valueBoundaryCoeffs() 来构建 fvMatrix，其中 A_1 进入矩阵对角位置，B_1 进入源项；fvm::laplacian 将会调用函数 gradientInternalCoeffs() 和 gradientBoundaryCoeffs() 来构建 fvMatrix，其中 A_2 进入矩阵对角位置，B_2 进入源项。

在每个边界条件中都有两个函数 evaluate 和 updateCoeffs，两个区别在于调用的时间不同，对 updateCoeffs() 的调用作为 <matrix>.solve() 的初始步骤发生，主要用于 fvm 相关的隐式计算；evaluate() 方法内定义的等于号 operator 通过 correctboundaryconditions() 的调用来更新边界处的值，一般在矩阵求解之后调用或者独立于矩阵求解，主要用于 fvc 相关的显式计算。[1]

correctBoundaryConditions() 是在 OpenFOAM/fields/GeometricFields/GeometricField/GeometricField.C 中定义的：

```
template<class Type, template<class> class PatchField, class GeoMesh>
void Foam::GeometricField<Type, PatchField, GeoMesh>::
correctBoundaryConditions()
{
    this->setUpToDate();
    storeOldTimes();
    boundaryField_.evaluate();
}
```

其中 boundaryField_ 的定义为 Boundary boundaryField_，而 Boundary 类是 GeometricField 类的嵌套类。evaluate() 函数是在 OpenFOAM/fields/GeometricFields/GeometricField/GeometricBoundaryField.C 中定义的，该函数中会调用

```
this->operator[](patchi).initEvaluate(Pstream::defaultCommsType);
this->operator[](patchi).evaluate(Pstream::defaultCommsType);
```

或者

```
this->operator[](patchSchedule[patchEvali].patch).initEvaluate(Pstream::commsTypes::scheduled);
this->operator[](patchSchedule[patchEvali].patch).evaluate(Pstream::commsTypes::scheduled);
```

此处的 evaluate 函数会调用各个具体边界条件类型所定义的 evaluate(Pstream::commsTypes) 函数。对于某些边界条件，比如 fixedValue 边界条件没有定义自己的 evaluate 函数，那么将会调用父类 fvPatchField<Type> 的 evaluate(const Pstream::commsTypes) 函数，父类 fvPatchField<Type> 中的定义为：

```
// finiteVolume/fields/fvPatchFields/fvPatchField/fvPatchField.C
template<class Type>
void Foam::fvPatchField<Type>::evaluate(const Pstream::commsTypes)
{
    if (!updated_)
```

[1] 由于 evaluate 方法定义了 operator= 从而更新了边界处的值，所以直接影响了 fvc::div 的计算；同时也因为更新了边界处的值而影响了 fvPatchField 的成员函数 snGrad 的值，进而影响了 fvc::laplacian 的计算。

```
    {
        updateCoeffs();
    }

    updated_ = false;
    manipulatedMatrix_ = false;
}
```

而有些边界条件继承于 transform 边界，里面额外定义了通过 snGrad() 和 snGradTransformDiag() 确定四个系数的方式，查看 src/finiteVolume/fields/fvPatchFields/basic/transform 中的 transformFvPatchField.C 可以看到它们之间的关系：

transform 边界设定的系数 A_1 表达式为

$$A_1 = (1,1,1) - \text{snGradTransformDiag} = \begin{bmatrix} 1 - \text{snGradTransformDiag}_x \\ 1 - \text{snGradTransformDiag}_y \\ 1 - \text{snGradTransformDiag}_z \end{bmatrix}$$

具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::transformFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return pTraits<Type>::one - snGradTransformDiag();
}
```

transform 边界设定的系数 B_1 表达式为

$$B_1 = \vec{\phi}_p - [(1,1,1) - \text{snGradTransformDiag}] * \vec{\phi}_c = \begin{bmatrix} \phi_{px} - (1 - \text{snGradTransformDiag}_x) \cdot \phi_{cx} \\ \phi_{py} - (1 - \text{snGradTransformDiag}_y) \cdot \phi_{cy} \\ \phi_{pz} - (1 - \text{snGradTransformDiag}_z) \cdot \phi_{cz} \end{bmatrix}$$

具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::transformFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return
        *this
      - cmptMultiply
        (
            valueInternalCoeffs(this->patch().weights()),
            this->patchInternalField()
        );
}
```

transform 边界条件设定的系数 A_2 表达式为

$$A_2 = -\frac{1}{d} \times \text{snGradTransformDiag} = \begin{bmatrix} -\frac{1}{d} \times \text{snGradTransformDiag}_x \\ -\frac{1}{d} \times \text{snGradTransformDiag}_y \\ -\frac{1}{d} \times \text{snGradTransformDiag}_z \end{bmatrix}$$

具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::transformFvPatchField<Type>::gradientInternalCoeffs() const
{
    return -this->patch().deltaCoeffs()*snGradTransformDiag();
}
```

transform 边界条件设定的系数 B_2 表达式为

$$B_2 = \text{snGrad} - \big(-(\frac{1}{d}) \times \text{snGradTransformDiag} \big) \cdot \vec{\phi}_c = \begin{bmatrix} \text{snGrad}_x + \frac{1}{d} \times \text{snGradTransformDiag}_x \times \phi_{cx} \\ \text{snGrad}_y + \frac{1}{d} \times \text{snGradTransformDiag}_y \times \phi_{cy} \\ \text{snGrad}_z + \frac{1}{d} \times \text{snGradTransformDiag}_z \times \phi_{cz} \end{bmatrix}$$

具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::transformFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return
        snGrad()
      - cmptMultiply(gradientInternalCoeffs(), this->patchInternalField());
}
```

部分派生类在定义自己的 snGrad() 和 snGradTransformDiag() 会涉及到 transform 函数和 transformFieldMask。参考http://xiaopingqiu.github.io/2016/04/02/Boundary-conditions-in-OpenFOAM3/可知 transform 函数的作用为：

```
inline scalar transform(constsymmTensor&, const scalar s)
{
    return s;
}

template<class Cmpt>
inline Vector<Cmpt> transform(const symmTensor& stt, const Vector<Cmpt>& v)
{
    return stt & v;
}
```

而与 transformFieldMask 有关的下述代码经过测试返回的是 diag 本身：

```
transformFieldMask<Type>(pow<vector, pTraits<Type>::rank>(diag))
```

3.1 通用边界条件

3.1.1 fixedValue

fixedValue 边界条件对边界值和梯度值的计算方式如下：

$$x_p = a$$
$$\nabla x_p = \frac{a - x_c}{d}$$

其中 a 是自己通过 value 关键词设定的值，d 是面心与面所属网格中心的距离。于是四个参数的设置应当如下：

$$A_1 = 0, \quad B_1 = a$$
$$A_2 = -\frac{1}{d}, \quad B_2 = \frac{a}{d}$$

可以通过位于 src/finiteVolume/fields/fvPatchFields/basic/fixedValue 的 fixedValueFvPatchField.C 文件查看其内容。首先，在边界条件设置信息的读取方面，fixedValue 边界条件需要读取 value 关键词的值，具体调用的代码如下：

```
template<class Type>
Foam::fixedValueFvPatchField<Type>::fixedValueFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict,
    const bool valueRequired
)
:
    fvPatchField<Type>(p, iF, dict, valueRequired)
{}
```

因此，一般来说可以通过如下方式来设置边界条件为 fixedValue 类型：

```
inlet
{
    type          fixedValue;
    value         uniform (3 0 0);
}
```

该边界条件对各系数赋值的方式如下。对于 valueInternalCoeffs 也就是 A_1 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedValueFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type>>
    (
        new Field<Type>(this->size(), Zero)
    );
}
```

对于 valueBoundaryCoeffs 也就是 B_1，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedValueFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return *this;
}
// *this 表示类本身也即当前边界上的值
```

对于 gradientInternalCoeffs 也就是 A_2，具体调用的代码如下：[2]

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedValueFvPatchField<Type>::gradientInternalCoeffs() const
{
    return -pTraits<Type>::one*this->patch().deltaCoeffs();
}
```

对于 gradientBoundaryCoeffs 也就是 B_2，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedValueFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return this->patch().deltaCoeffs()*(*this);
}
```

3.1.2 zeroGradient

zeroGradient 对边界的值和梯度值的计算方式如下：

$$x_p = x_c$$

$$\nabla x_p = 0$$

于是可知四个参数的设置应当如下：

$$A_1 = 1, \quad B_1 = 0$$

$$A_2 = 0, \quad B_2 = 0$$

可以通过位于 src/finiteVolume/fields/fvPatchFields/basic/zeroGradient 的 zeroGradientFvPatchField.C 文件查看其内容。首先，在边界条件设置信息的读取方面，zeroGradient 边界条件不需要读取任何信息，因此一般来说可以通过如下方式来设置边界条件为 zeroGradient 类型：

```
outlet
{
    type          zeroGradient;
```

[2] this->patch().deltaCoeffs() 可以近似认为返回的是 1/d，严格来说应该是两个网格中心的距离往接触面法向的投影的倒数。该函数的定义涉及 finiteVolume/interpolation/surfaceInterpolation/surfaceInterpolation/surfaceInterpolation.C 以及 finiteVolume/fvMesh/fvPatches/fvPatch/fvPatch.C 文件。

```
}
```

该边界条件对各系数赋值的方式如下。对于 valueInternalCoeffs 也就是 A_1 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::zeroGradientFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type>>
    (
        new Field<Type>(this->size(), pTraits<Type>::one)
    );
}
```

对于 valueBoundaryCoeffs 也就是 B_1 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::zeroGradientFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type>>
    (
        new Field<Type>(this->size(), Zero)
    );
}
```

对于 gradientInternalCoeffs 也就是 A_2 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::zeroGradientFvPatchField<Type>::gradientInternalCoeffs() const
{
    return tmp<Field<Type>>
    (
        new Field<Type>(this->size(), Zero)
    );
}
```

对于 gradientBoundaryCoeffs 也就是 B_2 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::zeroGradientFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return tmp<Field<Type>>
    (
```

```
        new Field<Type>(this->size(), Zero)
    );
}
```

zeroGradient 定义的 evaluate 函数操作内容如下：

```
template<class Type>
void Foam::zeroGradientFvPatchField<Type>::evaluate(const Pstream::commsTypes)
{
    if (!this->updated())
    {
        this->updateCoeffs();
    }

    fvPatchField<Type>::operator==(this->patchInternalField());
    fvPatchField<Type>::evaluate();
}
```

3.1.3 fixedGradient

fixedGradient 对边界的值和梯度值的计算方式如下：

$$x_p = x_c + a \cdot d$$
$$\nabla x_p = a$$

其中 a 是自行通过 gradient 关键词设定的固定梯度值，d 是面心与面所属网格中心的距离。于是可知四个参数的设置应当如下：

$$A_1 = 1, \quad B_1 = a \cdot d$$
$$A_2 = 0, \quad B_2 = a$$

可以通过位于 src/finiteVolume/fields/fvPatchFields/basic/fixedGradient 的 fixedGradientFvPatchField.C 文件查看其内容。首先，在边界条件设置信息的读取方面，fixedGradient 边界条件需要读取 gradient 关键词的值，具体调用的代码以及初始化过程如下：

```
template<class Type>
Foam::fixedGradientFvPatchField<Type>::fixedGradientFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict,
    const bool gradientRequired
)
:
    fvPatchField<Type>(p, iF, dict, false),
    gradient_(p.size())
{
    if (gradientRequired)
    {
        if (dict.found("gradient"))
        {
            gradient_ = Field<Type>("gradient", dict, p.size());
```

```
            evaluate();
        }
        else
        {
            FatalIOErrorInFunction(dict)
                << "Essential entry 'gradient' missing"
                << exit(FatalIOError);
        }
    }
}
```

该边界条件对各系数赋值的方式如下。对于 valueInternalCoeffs 也就是 A_1 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedGradientFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return tmp<Field<Type>>(new Field<Type>(this->size(), pTraits<Type>::one));
}
```

对于 valueBoundaryCoeffs 也就是 B_1 ，具体调用的代码如下：[3]

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedGradientFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return gradient()/this->patch().deltaCoeffs();
}
```

对于 gradientInternalCoeffs 也就是 A_2 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedGradientFvPatchField<Type>::gradientInternalCoeffs() const
{
    return tmp<Field<Type>>
    (
        new Field<Type>(this->size(), Zero)
    );
}
```

对于 gradientBoundaryCoeffs 也就是 B_2 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
```

[3] gradient() 返回的是已经映射到场上的 gradient_ 变量，this->patch().deltaCoeffs() 返回的是 1/d。

```
Foam::fixedGradientFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return gradient(); // 设定B2的值为已经映射到场上的gradient_变量，即a
}
```

fixedGradient 边界条件定义的 evaluate 函数操作具体调用的代码如下：

```
template<class Type>
void Foam::fixedGradientFvPatchField<Type>::evaluate(const Pstream::commsTypes)
{
    if (!this->updated())
    {
        this->updateCoeffs();
    }

    Field<Type>::operator=
    (
        this->patchInternalField() + gradient_/this->patch().deltaCoeffs()
    );

    fvPatchField<Type>::evaluate();
}
```

3.1.4 mixed

mixed 边界条件对边界的值和梯度值的计算方式如下：

$$x_p = [(1 - \text{valueFraction})x_c + \text{valueFraction} \cdot \text{refValue}] + (1 - \text{valueFraction}) \cdot \text{refGradient} \cdot d$$

$$\nabla x_p = \text{valueFraction} \cdot \frac{\text{refValue} - x_c}{d} + (1 - \text{valueFraction}) \cdot \text{refGradient}$$

其中 refValue 是自行通过 refValue 关键词设定的参考值，refGradient 是自行通过 refGradient 关键词设定的参考梯度，valueFraction 是自行通过 valueFraction 关键词设定的权重值，d 是面心与面所属网格中心的距离。于是可知四个参数的设置应当如下：

$$A_1 = 1 - \text{valueFraction}, \quad B_1 = \text{valueFraction} \cdot \text{refValue} + (1 - \text{valueFraction}) \cdot \text{refGradient} \cdot d$$

$$A_2 = \frac{-\text{valueFraction}}{d}, \quad B_2 = \frac{\text{valueFraction} \cdot \text{refValue}}{d} + (1 - \text{valueFraction}) \cdot \text{refGradient}$$

可以通过位于 src/finiteVolume/fields/fvPatchFields/basic/mixed 的 mixedFvPatchField.C 文件查看其内容。首先，在边界条件设置信息的读取方面，mixed 边界条件需要分别读取字典中的 refValue、refGradient、valueFraction 关键词的值，具体调用的代码以及初始化过程如下：

```
template<class Type>
Foam::mixedFvPatchField<Type>::mixedFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict
)
:
    fvPatchField<Type>(p, iF, dict, false),
    refValue_("refValue", dict, p.size()),
```

```
    refGrad_("refGradient", dict, p.size()),
    valueFraction_("valueFraction", dict, p.size())
{
    evaluate();
}
```

该边界条件对各系数赋值的方式如下。对于 valueInternalCoeffs 也就是 A_1 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::mixedFvPatchField<Type>::valueInternalCoeffs
(
    const tmp<scalarField>&
) const
{
    return Type(pTraits<Type>::one)*(1.0 - valueFraction_);
}
// pTraits<Type>::one 的作用是调整维度
```

对于 valueBoundaryCoeffs 也就是 B_1 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::mixedFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return
        valueFraction_*refValue_
      + (1.0 - valueFraction_)*refGrad_/this->patch().deltaCoeffs();
}
// this->patch().deltaCoeffs()返回的是 1/d
```

对于 gradientInternalCoeffs 也就是 A_2 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::mixedFvPatchField<Type>::gradientInternalCoeffs() const
{
    return -Type(pTraits<Type>::one)*valueFraction_*this->patch().deltaCoeffs();
}
```

对于 gradientBoundaryCoeffs 也就是 B_2 ，具体调用的代码如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::mixedFvPatchField<Type>::gradientBoundaryCoeffs() const
{
    return
       valueFraction_*this->patch().deltaCoeffs()*refValue_
     + (1.0 - valueFraction_)*refGrad_;
```

```
}
```

mixed 边界条件定义的 evaluate 函数操作具体调用的代码如下：

```
template<class Type>
void Foam::mixedFvPatchField<Type>::evaluate(const Pstream::commsTypes)
{
    if (!this->updated())
    {
        this->updateCoeffs();
    }

    Field<Type>::operator=
    (
        valueFraction_*refValue_
      +
        (1.0 - valueFraction_)*
        (
            this->patchInternalField()
          + refGrad_/this->patch().deltaCoeffs()
        )
    );

    fvPatchField<Type>::evaluate();
}
```

3.1.5 directionMixed

该类是所有方向混合类型边界条件的基类。该类继承于 transform 边界条件，通过定义其自身的 snGrad() 和 snGradTransformDiag() 函数返回内容来实现。可以通过位于 src/finiteVolume/fields/fvPatchFields/basic/directionMixed 的 directionMixedFvPatchField.C 文件查看其内容。

首先，在边界条件设置信息的读取方面，directionMixed 边界条件需要分别读取字典中的 refValue、refGradient、valueFraction 关键词的值，具体调用的代码以及初始化过程如下：

```
template<class Type>
Foam::directionMixedFvPatchField<Type>::directionMixedFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict
)
:
    transformFvPatchField<Type>(p, iF, dict),
    refValue_("refValue", dict, p.size()),
    refGrad_("refGradient", dict, p.size()),
    valueFraction_("valueFraction", dict, p.size())
{
    evaluate();
}
```

在 directionMixed 边界条件中，snGrad 的具体定义内容为

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::directionMixedFvPatchField<Type>::snGrad() const
{
    // pif用来代表内部场x_c
    const Field<Type> pif(this->patchInternalField());
    // 定义normalValue的值，通过transform函数来计算
    tmp<Field<Type>> normalValue = transform(valueFraction_, refValue_);
    // 定义gradValue的值为 x_c + refGrad / (1/d)
    tmp<Field<Type>> gradValue = pif + refGrad_/this->patch().deltaCoeffs();
    // 定义transformGradValue的值，通过transform函数来计算。其中I是单位矩阵
    tmp<Field<Type>> transformGradValue = transform(I - valueFraction_, gradValue);

    return
        (normalValue + transformGradValue - pif)*
        this->patch().deltaCoeffs();
}
```

在 directionMixed 边界条件中，snGradTransformDiag 的具体定义内容为

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::directionMixedFvPatchField<Type>::snGradTransformDiag() const
{
    vectorField diag(valueFraction_.size()); // 定义向量场diag

    diag.replace
    (   // 将diag向量的x分量替换为 \sqrt{ |valueFraction_xx| }
        vector::X,
        sqrt(mag(valueFraction_.component(symmTensor::XX)))
    );
    diag.replace
    (   // 将diag向量的y分量替换为 \sqrt{ |valueFraction_yy| }
        vector::Y,
        sqrt(mag(valueFraction_.component(symmTensor::YY)))
    );
    diag.replace
    (   // 将diag向量的z分量替换为 \sqrt{ |valueFraction_zz| }
        vector::Z,
        sqrt(mag(valueFraction_.component(symmTensor::ZZ)))
    );

    return transformFieldMask<Type>(pow<vector, pTraits<Type>::rank>(diag));
}
```

因此在 directionMixed 边界条件中，四个系数的具体定义分别为

$$A_1 = \begin{bmatrix} 1-\sqrt{|\text{valueFraction}|} \\ 1-\sqrt{|\text{valueFraction}|} \\ 1-\sqrt{|\text{valueFraction}|} \end{bmatrix}$$

$$B_1 = \begin{bmatrix} \phi_{px} - (1-\sqrt{|\text{valueFraction}|}) \times \phi_{cx} \\ \phi_{py} - (1-\sqrt{|\text{valueFraction}|}) \times \phi_{cy} \\ \phi_{pz} - (1-\sqrt{|\text{valueFraction}|}) \times \phi_{cz} \end{bmatrix}$$

$$A_2 = -\frac{1}{d} \begin{bmatrix} \sqrt{|\text{valueFraction}|} \\ \sqrt{|\text{valueFraction}|} \\ \sqrt{|\text{valueFraction}|} \end{bmatrix}$$

$$B_2 = \begin{bmatrix} \frac{1}{d} \times \text{valueFraction} \times \text{refValue} + \frac{1}{d} \times (\sqrt{|\text{valueFraction}|} - \text{valueFraction}) \times \phi_{cx} + (1-\text{valueFraction}) \times \text{refGrad} \\ \frac{1}{d} \times \text{valueFraction} \times \text{refValue} + \frac{1}{d} \times (\sqrt{|\text{valueFraction}|} - \text{valueFraction}) \times \phi_{cy} + (1-\text{valueFraction}) \times \text{refGrad} \\ \frac{1}{d} \times \text{valueFraction} \times \text{refValue} + \frac{1}{d} \times (\sqrt{|\text{valueFraction}|} - \text{valueFraction}) \times \phi_{cz} + (1-\text{valueFraction}) \times \text{refGrad} \end{bmatrix}$$

directionMixed 边界条件定义的 evaluate 函数操作内容如下：

```
template<class Type>
void Foam::directionMixedFvPatchField<Type>::evaluate(const Pstream::commsTypes)
{
    if (!this->updated())
    {
        this->updateCoeffs();
    }

    tmp<Field<Type>> normalValue = transform(valueFraction_, refValue_);
    tmp<Field<Type>> gradValue = this->patchInternalField() + refGrad_/this->patch().deltaCoeffs();
    tmp<Field<Type>> transformGradValue = transform(I - valueFraction_, gradValue);

    Field<Type>::operator=(normalValue + transformGradValue);

    transformFvPatchField<Type>::evaluate();
}
```

3.1.6 inletOutlet

inletOutlet 边界条件是一个一般性的出流条件，同时给定了特定的入流条件。对于入流，该条件相当于 fixedValue；对于出流，该条件相当于 zeroGradient。该条件继承于 mixed 边界条件，通过设置不同的 refValue、refGrad 和 valueFraction 以及等于号 operator 内容来实现。其中 refGrad 被初始化为零。

inletOutlet 边界条件需要读取 inletValue 关键词的赋值赋，也可以提供 value 关键词的赋值，具体调用的代码以及初始化过程如下：

```
template<class Type>
Foam::inletOutletFvPatchField<Type>::inletOutletFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict
```

```
)
:
    mixedFvPatchField<Type>(p, iF),
    phiName_(dict.lookupOrDefault<word>("phi", "phi"))
{
    this->refValue() = Field<Type>("inletValue", dict, p.size());

    if (dict.found("value"))
    {
        fvPatchField<Type>::operator=
        (
            Field<Type>("value", dict, p.size())
        );
    }
    else
    {
        fvPatchField<Type>::operator=(this->refValue());
    }

    this->refGrad() = Zero;
    this->valueFraction() = 0.0;
}
```

因此，inletOutlet 边界条件的一般设置方式如下所示：

```
patchName
{
    type          inletOutlet;
    phi           phi;
    inletValue    uniform 0;
    value         uniform 0;
}
```

intletOutlet 边界条件通过设置自己的 updateCoeffs 函数来修改 mixed 边界条件的相关参数，同时定义自己的等于号 operator 操作内容来实现自身效果，之后通过调用 mixed 边界条件的 evaluate 函数起作用。intletOutlet 边界条件的 updateCoeffs 函数定义如下：[4]

```
template<class Type>
void Foam::inletOutletFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const Field<scalar>& phip =
        this->patch().template lookupPatchField<surfaceScalarField, scalar>
        (
            phiName_
```

[4]neg(phip) 表示当 phip 小于零时取 1，当 phip 大于等于零时取 0，具体定义可以查看 src/OpenFOAM/lnInclude/Scalar.H 文件。

```
        );

    this->valueFraction() = neg(phip);

    mixedFvPatchField<Type>::updateCoeffs();
}
```

intletOutlet 边界条件的等于号 operator 定义如下：

```
template<class Type>
void Foam::inletOutletFvPatchField<Type>::operator=
(
    const fvPatchField<Type>& ptf
)
{
    fvPatchField<Type>::operator=
    (
        this->valueFraction()*this->refValue()
        + (1 - this->valueFraction())*ptf
    );
}
```

3.1.7 freestream

这个边界条件提供了一个自由流条件。它是由 inletOutlet 条件衍生而来的混合条件，其中操作模式根据通量的符号在固定 (自由流) 值和零梯度之间切换。该边界条件继承于 inletOutlet 边界条件。可以认为 freestreamValue 和 inletValue 的作用是等价的。

在边界条件设置的读取方面，freestream 边界条件需要读取 phi、freestreamValue 关键词的赋值，也可以提供 value 关键词的赋值，调用的代码以及初始化过程如下：

```
template<class Type>
Foam::freestreamFvPatchField<Type>::freestreamFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict
)
:
    inletOutletFvPatchField<Type>(p, iF)
{
    this->phiName_ = dict.lookupOrDefault<word>("phi","phi");

    freestreamValue() = Field<Type>("freestreamValue", dict, p.size());

    if (dict.found("value"))
    {
        fvPatchField<Type>::operator=
        (
            Field<Type>("value", dict, p.size())
        );
```

```
    }
    else
    {
        fvPatchField<Type>::operator=(freestreamValue());
    }
}
```

因此 freestream 边界条件的一般设置方式如下所示：

```
patchName
{
    type          freestream;
    freestreamValue uniform (300 0 0);
}
```

3.2 对称性边界条件

3.2.1 basicSymmetry

对于标量，该边界条件等价于 zeroGradient。对于矢量的情况，边界上的值等于其临近网格中心的值的切向分量，满足：

$$\vec{\phi}_p = \vec{\phi}_c - (\vec{\phi}_c \cdot \vec{n}) \cdot \vec{n} = [\vec{\phi}_c + (I - 2\vec{n} \otimes \vec{n}) \cdot \vec{\phi}_c] \times \frac{1}{2}$$

各系数在 OpenFOAM 中的矩阵表示如下：

$$A_1 = \begin{bmatrix} 1 - |n_x| \\ 1 - |n_y| \\ 1 - |n_z| \end{bmatrix}$$

$$B_1 = \begin{bmatrix} \phi_{px} \\ \phi_{py} \\ \phi_{pz} \end{bmatrix} - \begin{bmatrix} (1 - |n_x|)\phi_{cx} \\ (1 - |n_y|)\phi_{cy} \\ (1 - |n_z|)\phi_{cz} \end{bmatrix}$$

$$A_2 = -\frac{1}{d} \begin{bmatrix} |n_x| \\ |n_y| \\ |n_z| \end{bmatrix}$$

$$B_2 = -\frac{1}{d} \begin{bmatrix} n_x(\phi_{cx}n_x + \phi_{cy}n_y + \phi_{cz}n_z) \\ n_y(\phi_{cx}n_x + \phi_{cy}n_y + \phi_{cz}n_z) \\ n_z(\phi_{cx}n_x + \phi_{cy}n_y + \phi_{cz}n_z) \end{bmatrix} + \frac{1}{d} \begin{bmatrix} |n_x|\phi_{cx} \\ |n_y|\phi_{cy} \\ |n_z|\phi_{cz} \end{bmatrix}$$

可以通过位于 src/finiteVolume/fields/fvPatchFields/basic/basicSymmetry 的 basicSymmetryFvPatchField.C 文件查看其内容。首先，在边界条件设置信息的读取方面，basicSymmetry 边界条件不需要读取设置信息。

在 basicSymmetry 边界条件中，snGrad 的具体定义内容为

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::basicSymmetryFvPatchField<Type>::snGrad() const
{
    tmp<vectorField> nHat = this->patch().nf(); // 用nHat表示面法向量
    const Field<Type> iF(this->patchInternalField()); // 用iF表示 \vec{\phi}_{c}
```

```
    return
        (transform(I - 2.0*sqr(nHat), iF) - iF)
       *(this->patch().deltaCoeffs()/2.0);
}
```

在 basicSymmetry 边界条件中，snGradTransformDiag 的具体定义内容为

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::basicSymmetryFvPatchField<Type>::snGradTransformDiag() const
{
    const vectorField nHat(this->patch().nf()); // 用nHat表示面法向量
    vectorField diag(nHat.size());

    diag.replace(vector::X, mag(nHat.component(vector::X)));
    diag.replace(vector::Y, mag(nHat.component(vector::Y)));
    diag.replace(vector::Z, mag(nHat.component(vector::Z)));

    return transformFieldMask<Type>(pow<vector, pTraits<Type>::rank>(diag));
}
```

basicSymmetry 边界条件定义的 evaluate 函数操作内容为

```
template<class Type>
void Foam::basicSymmetryFvPatchField<Type>::evaluate(const Pstream::commsTypes)
{
    if (!this->updated())
    {
        this->updateCoeffs();
    }

    tmp<vectorField> nHat = this->patch().nf();  // 用nHat表示面法向量
    const Field<Type> iF(this->patchInternalField()); // 用iF表示 \vec{\phi}_{c}

    Field<Type>::operator=
    (
        (iF + transform(I - 2.0*sqr(nHat), iF))/2.0
    );

    transformFvPatchField<Type>::evaluate();
}
```

3.2.2 wedge

wedge 边界条件用于二维轴对称算例，它与周期性边界非常相似，只是涉及到内部变量向边界处的旋转投影。旋转投影所用到的参数在创建网格的时候计算，从 src/OpenFOAM/meshes/polyMesh/polyPatches/constraint/wedge/wedgePolyPatch.H 以及 wedgePolyPatch.C 文件中可以看到：

```
class wedgePolyPatch
:
```

```
    public polyPatch
{
    // Private Data

        //- Axis of the wedge
        vector axis_;

        //- Centre normal between the wedge boundaries
        vector centreNormal_;

        //- Normal to the patch
        vector n_;

        //- Cosine of the wedge angle
        scalar cosAngle_;

        //- Face transformation tensor
        tensor faceT_;

        //- Neighbour-cell transformation tensor
        tensor cellT_;
};
```

```
void Foam::wedgePolyPatch::calcGeometry(PstreamBuffers&)
{
    const vectorField& nf(faceNormals());
    n_ = gAverage(nf);

    centreNormal_ =
        vector
        (
            sign(n_.x())*(max(mag(n_.x()), 0.5) - 0.5),
            sign(n_.y())*(max(mag(n_.y()), 0.5) - 0.5),
            sign(n_.z())*(max(mag(n_.z()), 0.5) - 0.5)
        );
    centreNormal_ /= mag(centreNormal_);

    cosAngle_ = centreNormal_ & n_;

    const scalar cnCmptSum =
        centreNormal_.x() + centreNormal_.y() + centreNormal_.z();

    axis_ = centreNormal_ ^ n_;
    scalar magAxis = mag(axis_);

    axis_ /= magAxis;

    faceT_ = rotationTensor(centreNormal_, n_);
    cellT_ = faceT_ & faceT_;
```

```
}
```

其中所使用到的 rotationTensor 定义在 src/OpenFOAM/primitives/transform/transform.H 中，内容如下：

```
//- Rotational transformation tensor from unit vector n1 to n2
inline tensor rotationTensor
(
    const vector& n1,
    const vector& n2
)
{
    const scalar c = n1 & n2;
    const vector n3 = n1 ^ n2;
    const scalar magSqrN3 = magSqr(n3);

    // n1 and n2 define a plane n3
    if (magSqrN3 > small)
    {
        // Return rotational transformation tensor in the n3-plane using
        // Rodrigues' rotation formula
        return
            c*I
          + (n2*n1 - n1*n2)
          + (1 - c)*sqr(n3)/magSqrN3;
    }
    // n1 and n2 are contradirectional
    else if (c < 0)
    {
        // Return rotational transformation tensor in a plane with arbitrary
        // normal perpendicular to n1 and n2 using a subset of Rodrigues'
        // rotation formula
        const vector n4 = perpendicular(n1);
        const scalar magSqrN4 = magSqr(n4);
        return - I + 2*sqr(n4)/magSqrN4;
    }
    // n1 and n2 are codirectional
    else
    {
        // Return null transformation tensor
        return I;
    }
}
```

于是在 src/finiteVolume/fields/fvPatchFields/constraint/wedge/wedgeFvPatchField.C 中可以看到最终的实现方式：

```
template<class Type>
Foam::tmp<Foam::Field<Type>> Foam::wedgeFvPatchField<Type>::snGrad() const
{
    const Field<Type> pif(this->patchInternalField());

    return
```

```
    (
        transform(refCast<const wedgeFvPatch>(this->patch()).cellT(), pif) - pif
    )*(0.5*this->patch().deltaCoeffs());
}


template<class Type>
void Foam::wedgeFvPatchField<Type>::evaluate(const Pstream::commsTypes)
{
    if (!this->updated())
    {
        this->updateCoeffs();
    }

    fvPatchField<Type>::operator==
    (
        transform
        (
            refCast<const wedgeFvPatch>(this->patch()).faceT(),
            this->patchInternalField()
        )
    );
}


template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::wedgeFvPatchField<Type>::snGradTransformDiag() const
{
    const vector diagV
    (
        0.5*diag(I - refCast<const wedgeFvPatch>(this->patch()).cellT())
    );

    return tmp<Field<Type>>
    (
        new Field<Type>
        (
            this->size(),
            transformMask<Type>
            (
                pow
                (
                    diagV,
                    pTraits<typename powProduct<vector, pTraits<Type>::rank>
                    ::type>::zero
                )
            )
        )
```

```
    );
}
```

3.3 速度边界条件

3.3.1 noSlip

这个边界条件使壁面处的速度固定为零。它继承于 fixedValue 边界条件。

```
Foam::noSlipFvPatchVectorField::noSlipFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchVectorField(p, iF, dict, false)
{
    operator==(Zero); // 继承fixedValue边界条件，设定为固定值 0
}
```

笔记 noSlip 边界条件实际上将无滑移条件设置成了 Dirichlet 类边界，在继承自 fixedValue 边界条件的时候也导致相应两个系数所表示的速度梯度成了 $\nabla U = \frac{\vec{0}-\mathbf{U}_p}{d}$，fvPatchField 类本身定义的 snGrad 函数也是如此。然而根据牛顿内摩擦定律以及粘性底层的性质，边界处的速度梯度应该只与平行于边界的速度分量相关，因此在 fvm::laplacian 和 fvc::laplacian 操作中都会引入误差，不过当边界处的网格划分地足够细密时，垂直于边界的分量基本忽略不计，因此误差也相对可以忽略。

3.3.2 partialSlip

对于 partialSlip 边界，流体在边界法向上的速度为零，边界切向上的速度沿法向成梯度下降，满足

$$\vec{\phi}_p = (1-\text{valueFraction})[\vec{\phi}_c - (\vec{\phi}_c \cdot \vec{n}) \cdot \vec{n}] = (1-\text{valueFraction})(I - \vec{n} \otimes \vec{n}) \cdot \vec{\phi}_c$$

其中 valueFraction 为自行定义的衰减系数。各系数的表达式如下：

$$A_1 = \begin{bmatrix} (1-\text{valueFraction})(1-|n_x|) \\ (1-\text{valueFraction})(1-|n_y|) \\ (1-\text{valueFraction})(1-|n_z|) \end{bmatrix}$$

$$B_1 = \begin{bmatrix} \phi_{px} \\ \phi_{py} \\ \phi_{pz} \end{bmatrix} - \begin{bmatrix} (1-\text{valueFraction})(1-|n_x|)\phi_{cx} \\ (1-\text{valueFraction})(1-|n_y|)\phi_{cy} \\ (1-\text{valueFraction})(1-|n_z|)\phi_{cz} \end{bmatrix}$$

$$A_2 = -\frac{1}{d} \begin{bmatrix} \text{valueFraction} + (1-\text{valueFraction})|n_x| \\ \text{valueFraction} + (1-\text{valueFraction})|n_y| \\ \text{valueFraction} + (1-\text{valueFraction})|n_z| \end{bmatrix}$$

$$B_2 = \frac{1}{d} \begin{bmatrix} (1-\text{valueFraction})\big(n_x - n_x(\phi_{cx}n_x + \phi_{cy}n_y + \phi_{cz}n_z)\big) - \phi_{cx} \\ (1-\text{valueFraction})\big(n_y - n_y(\phi_{cx}n_x + \phi_{cy}n_y + \phi_{cz}n_z)\big) - \phi_{cy} \\ (1-\text{valueFraction})\big(n_z - n_z(\phi_{cx}n_x + \phi_{cy}n_y + \phi_{cz}n_z)\big) - \phi_{cz} \end{bmatrix} + \frac{1}{d} \begin{bmatrix} \big(\text{valueFraction} + (1-\text{valueFraction})|n_x|\big)\phi_{cx} \\ \big(\text{valueFraction} + (1-\text{valueFraction})|n_y|\big)\phi_{cy} \\ \big(\text{valueFraction} + (1-\text{valueFraction})|n_z|\big)\phi_{cz} \end{bmatrix}$$

通过查看位于 src/finiteVolume/fields/fvPatchFields/derived/partialSlip 的 partialSlipFvPatchField.C 文件可知，partialSlip 边界条件需要读取 valueFraction 关键词，调用的代码如下：

```
template<class Type>
Foam::partialSlipFvPatchField<Type>::partialSlipFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF,
    const dictionary& dict
)
:
    transformFvPatchField<Type>(p, iF),
    valueFraction_("valueFraction", dict, p.size())
{
    evaluate();
}
```

partialSlip 边界条件继承于 transform 边界，其定义 snGrad 函数的操作内容如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::partialSlipFvPatchField<Type>::snGrad() const
{
    tmp<vectorField> nHat = this->patch().nf(); // 用nHat来表示面法向量
    const Field<Type> pif(this->patchInternalField()); // 用pif来表示内部场的值 \vec{\phi}_{c}

    return
    (
        (1.0 - valueFraction_)*transform(I - sqr(nHat), pif) - pif
    )*this->patch().deltaCoeffs();
}
```

partialSlip 边界条件定义的 snGradTransformDiag 函数的操作内容如下：

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::partialSlipFvPatchField<Type>::snGradTransformDiag() const
{
    const vectorField nHat(this->patch().nf()); // 用nHat来表示面法向量
    vectorField diag(nHat.size()); // 定义一个矢量场diag

    diag.replace(vector::X, mag(nHat.component(vector::X))); // 替换对角矩阵x分量值为 |n_x|
    diag.replace(vector::Y, mag(nHat.component(vector::Y))); // 替换对角矩阵y分量值为 |n_y|
    diag.replace(vector::Z, mag(nHat.component(vector::Z))); // 替换对角矩阵z分量值为 |n_z|
    return
        valueFraction_*pTraits<Type>::one
      + (1.0 - valueFraction_)
       *transformFieldMask<Type>(pow<vector, pTraits<Type>::rank>(diag));
}
```

partialSlip 边界条件定义的 evaluate 操作内容如下：

```
template<class Type>
void Foam::partialSlipFvPatchField<Type>::evaluate
(
    const Pstream::commsTypes
)
{
    if (!this->updated())
    {
        this->updateCoeffs();
    }

    tmp<vectorField> nHat = this->patch().nf();

    Field<Type>::operator=
    (
        (1.0 - valueFraction_)
      *transform(I - sqr(nHat), this->patchInternalField())
    );

    transformFvPatchField<Type>::evaluate();
}
```

3.3.3 slip

这个边界条件提供了一个滑移约束。一般用于无粘性流体的情况。该边界条件继承于 basicSymmetry 边界条件。noSlip 边界等价于 basicSymmetry 边界的效果，即法向为零，切向保持。

```
template<class Type>
Foam::slipFvPatchField<Type>::slipFvPatchField
(
    const fvPatch& p,
    const DimensionedField<Type, volMesh>& iF
)
:
  basicSymmetryFvPatchField<Type>(p, iF)
{}
```

下面是使用该边界条件的一般设置方式：

```
patchName
{
    type          slip;
}
```

3.3.4 freestreamVelocity

该边界条件为速度提供了自由流条件。这是一种使用速度方向在正常入口的固定值和正常出口流量的零梯度之间连续混合的进出口条件。该边界条件继承于 mixed 边界条件。freestreamValue 等价与 refValue。

freestreamVelocity 边界条件将会读取设置中 freestreamValue 关键词的赋值，也可以提供 value 关键词的赋值，调用的代码以及初始化过程如下：

```
Foam::freestreamVelocityFvPatchVectorField::freestreamVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    mixedFvPatchVectorField(p, iF)
{
    freestreamValue() = vectorField("freestreamValue", dict, p.size());

    if (dict.found("value"))
    {
        fvPatchVectorField::operator=
        (
            vectorField("value", dict, p.size())
        );
    }
    else
    {
        fvPatchVectorField::operator=(freestreamValue());
    }

    refGrad() = Zero;
    valueFraction() = 1;
}
```

freestreamVelocity 边界条件通过定义自己的 updateCoeffs() 函数来更改 mixed 边界条件中 valueFraction 的值，具体操作内容如下：

```
void Foam::freestreamVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const Field<vector> Up(0.5*(patchInternalField() + *this)); // 定义Up，通过内部速度场和value的设
        置进行计算
    const Field<scalar> magUp(mag(Up)); // 定义magUp为Up的模长大小

    const Field<vector> nf(patch().nf()); // 定义nf为网格面单位法向量

    Field<scalar>& vf = valueFraction(); // 定义vf为valueFraction的值

    forAll(vf, i)
    {
```

```
        if (magUp[i] > vSmall)
        {
            vf[i] = 0.5 - 0.5*(Up[i] & nf[i])/magUp[i]; // 当Up的模长大小不是小量时valueFraction的设
                置方法
        }
        else
        {
            vf[i] = 0.5;  // 当Up的模长为小量时设置valueFraction为0.5
        }
    }

    mixedFvPatchField<vector>::updateCoeffs(); // 更新mixed边界条件的参数
}
```

3.3.5 pressureInletVelocity

这个速度入口边界条件适用于指定压力的边界面。流入速度由与边界面法线方向的通量得到。该边界条件继承于 fixedValue 边界条件，并且一般需要压力场配合使用 totalPressure 边界条件。

查看 src/finiteVolume/fields/fvPatchFields/derived/pressureInletVelocity/pressureInletVelocityFvPatchVectorField.C 文件，可以看到 pressureInletVelocity 边界条件需要读取 phi、rho、value 关键词，具体调用的代码如下：

```
Foam::pressureInletVelocityFvPatchVectorField::
pressureInletVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchVectorField(p, iF, dict),
    phiName_(dict.lookupOrDefault<word>("phi", "phi")),
    rhoName_(dict.lookupOrDefault<word>("rho", "rho"))
{}
```

因此 pressureInletVelocity 边界条件的一般设置方式为

```
patchName
{
    type          pressureInletVelocity;
    phi           phi;
    rho           rho;
    value         uniform 0;
}
```

pressureInletVelocity 通过定义自己的 updateCoeffs() 函数来设置自己的 operator= 应用于所继承的 fixedValue 边界条件当中：

```
void Foam::pressureInletVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
```

```
    {
        return;
    }

    const surfaceScalarField& phi = db().lookupObject<surfaceScalarField>(phiName_);
    const fvsPatchField<scalar>& phip = patch().patchField<surfaceScalarField, scalar>(phi);

    tmp<vectorField> n = patch().nf();
    const Field<scalar>& magS = patch().magSf();

    if (phi.dimensions() == dimFlux)
    {
        operator==(n*phip/magS);
    }
    else if (phi.dimensions() == dimMassFlux)
    {
        const fvPatchField<scalar>& rhop = patch().lookupPatchField<volScalarField, scalar>(rhoName
            _);

        operator==(n*phip/(rhop*magS));
    }
    else
    {
        FatalErrorInFunction
            << "dimensions of phi are not correct"
            << "\n    on patch " << this->patch().name()
            << " of field " << this->internalField().name()
            << " in file " << this->internalField().objectPath()
            << exit(FatalError);
    }

    fixedValueFvPatchVectorField::updateCoeffs();
}
```

3.3.6 pressureInletUniformVelocity

该速度入口边界条件适用于指定了压力的边界面。通过求边界面上通量的平均值来得到均一的入流速度，然后将其应用到边界面的法线方向。该边界条件继承于 pressureInletVelocity 边界条件。

查看 src/finiteVolume/fields/fvPatchFields/derived/pressureInletUniformVelocity/pressureInletUniformVelocityFvPatchVectorFiel 文件可以发现 pressureInletUniformVelocity 边界条件只是修改了 pressureInletVelocity 定义的 updateCoeffs() 函数中设置的 operator= 操作内容，调用的代码如下：

```
void Foam::pressureInletUniformVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }
```

```
    pressureInletVelocityFvPatchVectorField::updateCoeffs();

    operator==(patch().nf()*gSum(patch().Sf() & *this)/gSum(patch().magSf()));
}
```

3.3.7 pressureDirectedInletVelocity

这个速度入口边界条件适用于指定了压力的边界。流入速度由指定进口方向的“方向通量”得到。该边界条件继承于 fixedValue 边界条件。

查看 src/finiteVolume/fields/fvPatchFields/derived/pressureDirectedInletVelocity 中的 pressureDirectedInletVelocityFvPatchVectorField.C 文件，可以看到 pressureDirectedInletVelocity 边界条件需要读取 phi、rho、inletDirection、value 关键词的赋值，调用的代码如下：

```
Foam::pressureDirectedInletVelocityFvPatchVectorField::
pressureDirectedInletVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchVectorField(p, iF, dict),
    phiName_(dict.lookupOrDefault<word>("phi", "phi")),
    rhoName_(dict.lookupOrDefault<word>("rho", "rho")),
    inletDir_("inletDirection", dict, p.size())
{}
```

因此 pressureDirectedInletVelocity 边界条件的一般设置方式为

```
patchName
{
    type          pressureDirectedInletVelocity;
    phi           phi;
    rho           rho;
    inletDirection uniform (1 0 0);
    value         uniform 0;
}
```

pressureDirectedInletVelocity 边界条件通过定义自己的 updateCoeffs() 函数并设置自己的 operator= 来更改 fixedValue 边界条件的作用：

```
void Foam::pressureDirectedInletVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const surfaceScalarField& phi = db().lookupObject<surfaceScalarField>(phiName_);
```

```
    const fvsPatchField<scalar>& phip = patch().patchField<surfaceScalarField, scalar>(phi);

    tmp<vectorField> n = patch().nf();
    tmp<scalarField> ndmagS = (n & inletDir_)*patch().magSf();

    if (phi.dimensions() == dimFlux)
    {
        operator==(inletDir_*phip/ndmagS);
    }
    else if (phi.dimensions() == dimMassFlux)
    {
        const fvPatchField<scalar>& rhop = patch().lookupPatchField<volScalarField, scalar>(rhoName
            _);

        operator==(inletDir_*phip/(rhop*ndmagS));
    }
    else
    {
        FatalErrorInFunction
            << "dimensions of phi are not correct"
            << "\n   on patch " << this->patch().name()
            << " of field " << this->internalField().name()
            << " in file " << this->internalField().objectPath()
            << exit(FatalError);
    }

    fixedValueFvPatchVectorField::updateCoeffs();
}
```

3.3.8 pressureInletOutletVelocity

这是一个速度入口/出口边界条件，其中需要压力场配合使用 fixedValue、totalPressure、entrainmentPressure 等边界条件。该边界条件继承于 directionMixed 边界，对于出流等价于 zeroGradient 边界条件；对于入流将会为法向分量设置为 zeroGradient 而为切向分量设置为 fixedValue。外部切向速度可以被任意指定，如果没有设置则视为零。

pressureInletOutletVelocity 边界条件需要读取 phi、value 关键词的赋值，也可以提供 tangentialVelocity 关键词的赋值，调用的代码和初始化过程如下：

```
Foam::pressureInletOutletVelocityFvPatchVectorField::
pressureInletOutletVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    directionMixedFvPatchVectorField(p, iF),
    phiName_(dict.lookupOrDefault<word>("phi", "phi"))
{
```

```
    fvPatchVectorField::operator=(vectorField("value", dict, p.size()));

    if (dict.found("tangentialVelocity"))
    {
        tangentialVelocity_ = Function1<vector>::New("tangentialVelocity", dict);
    }

    refValue() = Zero;
    refGrad() = Zero;
    valueFraction() = Zero;
}
```

因此 pressureInletOutletVelocity 边界条件的一般设置方式为

```
patchName
{
    type          pressureInletOutletVelocity;
    phi           phi;
    tangentialVelocity (0 0 0);
    value         uniform 0;
}
```

pressureInletOutletVelocity 边界条件定义的 updateCoeffs() 函数如下：

```
void Foam::pressureInletOutletVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    if (tangentialVelocity_.valid())
    {
        const scalar t = this->db().time().userTimeValue();
        const vector tangentialVelocity = tangentialVelocity_->value(t);
        const vectorField n(patch().nf());
        refValue() = tangentialVelocity - n*(n & tangentialVelocity);
    }

    const fvsPatchField<scalar>& phip = patch().lookupPatchField<surfaceScalarField, scalar>(
        phiName_);

    valueFraction() = neg(phip)*(I - sqr(patch().nf()));

    directionMixedFvPatchVectorField::updateCoeffs();
    directionMixedFvPatchVectorField::evaluate();
}
```

3.3.9 pressureDirectedInletOutletVelocity

该速度入口/出口边界条件应用于指定了压力的边界。对于出流，该边界条件等价于 zeroGradient(由通量定义)；对于入流，速度由指定进口方向的通量得到。该边界条件继承于 mixed 边界条件。

```
Foam::pressureDirectedInletOutletVelocityFvPatchVectorField::
pressureDirectedInletOutletVelocityFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    mixedFvPatchVectorField(p, iF), // 继承于mixed边界条件
    phiName_(dict.lookupOrDefault<word>("phi", "phi")), // 读取通量场名称，默认为phi
    rhoName_(dict.lookupOrDefault<word>("rho", "rho")), // 读取密度场名称，默认为rho
    inletDir_("inletDirection", dict, p.size()) // 读取指定的进口速度
{
    fvPatchVectorField::operator=(vectorField("value", dict, p.size())); // 读取value关键词进行初始
        化
    refValue() = *this; // 初始化refValue为当前值
    refGrad() = Zero; // 初始化refGrad为零
    valueFraction() = 0.0; // 初始化valueFraction为零
}
```

下面是 pressureDirectedInletOutletVelocity 边界条件的一般设置方式：

```
patchName
{
    type          pressureDirectedInletOutletVelocity;
    phi           phi;
    rho           rho;   // 可压缩流
    inletDirection uniform (1 0 0); // 入流方向
    value         uniform 0;
}
```

通过如下方式计算 mixed 边界条件新的参数值并进行 updateCoeffs：

```
void Foam::pressureDirectedInletOutletVelocityFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const surfaceScalarField& phi = db().lookupObject<surfaceScalarField>(phiName_);
    const fvsPatchField<scalar>& phip = patch().patchField<surfaceScalarField, scalar>(phi); // 获
        取通量场

    tmp<vectorField> n = patch().nf(); // 获取边界面单位面法向量
    tmp<scalarField> ndmagS = (n & inletDir_)*patch().magSf();
```

```
    if (phi.dimensions() == dimFlux) // 不可压缩流体，通量为体积通量
    {
        refValue() = inletDir_*phip/ndmagS; // 计算refValue
    }
    else if (phi.dimensions() == dimMassFlux) // 可压缩流体，通量为质量通量
    {
        const fvPatchField<scalar>& rhop = patch().lookupPatchField<volScalarField, scalar>(rhoName
            _); // 获取密度场

        refValue() = inletDir_*phip/(rhop*ndmagS); // 计算refValue
    }
    else
    {
        FatalErrorInFunction
            << "dimensions of phi are not correct"
            << "\n    on patch " << this->patch().name()
            << " of field " << this->internalField().name()
            << " in file " << this->internalField().objectPath()
            << exit(FatalError);
    }

    valueFraction() = neg(phip); // 根据通量正负号设置valueFraction

    mixedFvPatchVectorField::updateCoeffs();
}
```

3.4 压强边界条件

3.4.1 freestreamPressure

该边界条件为压力提供了自由流动条件。这是一种出口-进口条件，使用速度方向在正常入口的零梯度和正常出口流量的固定值之间连续混合。该边界条件继承于 mixed 边界条件，通过利用速度设置 valueFraction 的值来实现。

```
Foam::freestreamPressureFvPatchScalarField::
freestreamPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    mixedFvPatchScalarField(p, iF), // 继承于mixed边界条件
    UName_(dict.lookupOrDefault<word>("U", "U")),
    supersonic_
    (
```

```
        dict.lookupOrDefault<Switch>("supersonic", false) // 读取字典关键词supersonic的赋值，默认是
            false
    )
{
    freestreamValue() = scalarField("freestreamValue", dict, p.size());

    if (dict.found("value"))
    {
        fvPatchScalarField::operator=
        (
            scalarField("value", dict, p.size()) // 如果字典中有value关键词就先将边界的值初始化为value
                的值
        );
    }
    else
    {
        fvPatchScalarField::operator=(freestreamValue()); // 如果没有value关键词就将边界初始化为
            freestreamValue的值
    }

    refGrad() = Zero;    // 初始化refGradient的值为0
    valueFraction() = 0; // 初始化valueFraction的值为0
}
```

freestreamPressure 将会读取速度场的名称，并且查找边界设置的 freestreamValue、supersonic 和 value 三个关键词。下面是使用 freestreamPressure 边界条件的一般设置方式：

```
patchName
{
    type            freestreamPressure;
    U               U;
    freestreamValue uniform 1e5;
    supersonic      false;
}
```

通过下面的方式修改 mixed 边界条件的相关参数并进行 updateCoeffs：

```
void Foam::freestreamPressureFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const Field<vector>& Up = // 定义Up为当前的速度场
        patch().template lookupPatchField<volVectorField, vector>
        (
            UName_
        );

    const Field<scalar> magUp(mag(Up)); // 定义magUp为当前速度场的模长大小
```

```
    const Field<vector> nf(patch().nf()); // 定义nf为网格面单位法向量
    Field<scalar>& vf = valueFraction(); // 定义vf为valueFraction的值

    if (supersonic_) // 如果设置supersonic为true
    {
        forAll(vf, i)
        {
            if (magUp[i] > vSmall) // 如果速度场大小不是一个小量
            {
                vf[i] = 0.5 - 0.5*(Up[i] & nf[i])/magUp[i]; // 设置valueFraction的值
            }
            else
            {
                vf[i] = 0.5;  // 如果速度场大小是一个小量则取valueFraction = 0.5
            }
        }
    }
    else   // 如果设置supersonic为false
    {
        forAll(vf, i)
        {
            if (magUp[i] > vSmall) // 如果速度场大小不是一个小量
            {
                vf[i] = 0.5 + 0.5*(Up[i] & nf[i])/magUp[i]; // 设置valueFraction的值
            }
            else
            {
                vf[i] = 0.5;  // 如果速度场大小是一个小量则取valueFraction = 0.5
            }
        }
    }

    mixedFvPatchField<scalar>::updateCoeffs(); // 更新mixed边界条件参数设置
}
```

3.4.2 dynamicPressure

这个边界条件提供了一个动态压力条件。它从参考压力中减去动能项得到边界上固定的值。它构成了总压力和夹带压力条件的基类。

```
Foam::dynamicPressureFvPatchScalarField::dynamicPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict, false), // 继承于fixedValue边界条件
    rhoName_(dict.lookupOrDefault<word>("rho", "rho")),
```

```
    psiName_(dict.lookupOrDefault<word>("psi", "none")),
    gamma_(dict.lookupOrDefault<scalar>("gamma", 1)),
    p0_("p0", dict, p.size())
{
    if (dict.found("value"))
    {
        fvPatchField<scalar>::operator=
        (
            scalarField("value", dict, p.size())
        );
    }
    else
    {
        fvPatchField<scalar>::operator=(p0_);
    }
}
```

```
void Foam::dynamicPressureFvPatchScalarField::updateCoeffs
(
    const scalarField& p0p, // 静压
    const scalarField& K0mKp // 单位动能
)
{
    if (updated())
    {
        return;
    }

    if (internalField().dimensions() == dimPressure) // 如果内部场单位是Pa，即可压缩流情况
    {
        if (psiName_ == "none") // 变密度和低速可压缩流的情况
        {
            const fvPatchField<scalar>& rho = patch().lookupPatchField<volScalarField, scalar>(
                rhoName_); // 定义密度场

            operator==(p0p + rho*K0mKp); // 计算值设置为fixedValue
        }
        else   // 高速可压缩流的情况
        {
            const fvPatchField<scalar>& psip = patch().lookupPatchField<volScalarField, scalar>(
                psiName_); // 定义可压缩性场

            if (gamma_ > 1) // 比热容比大于1的情况
            {
                const scalar gM1ByG = (gamma_ - 1)/gamma_;

                operator==
                (
                    p0p/pow(scalar(1) - psip*gM1ByG*K0mKp, 1/gM1ByG)
```

```
                );
            }
            else   // 比热容小于等于1的情况
            {
                operator==(p0p/(scalar(1) - psip*K0mKp));
            }
        }
    }
    else if (internalField().dimensions() == dimPressure/dimDensity) // 不可压缩流情况
    {
        operator==(p0p + K0mKp);
    }
    else
    {
        FatalErrorInFunction
            //...
    }

    fixedValueFvPatchScalarField::updateCoeffs(); // 调用fixedValue边界条件的updateCoeffs
}
```

3.4.3 totalPressure

基于恒定总压假设的流入、流出和夹带压力边界条件。对于流出，贴片压力设置为外部静压。对于流入，贴片压力是根据贴片速度和外部静压 p0 和外部速度 U0 来评估的，外部速度 U0 是根据贴片速度边界条件 (如果使用该边界条件) 中的可选 tangentialVelocity 入口查找的，否则 U0 被假设为零，外部总压力等于外部静压。

```
Foam::totalPressureFvPatchScalarField::totalPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    dynamicPressureFvPatchScalarField(p, iF, dict), // 继承于dynamicPressure边界条件
    UName_(dict.lookupOrDefault<word>("U", "U")),
    phiName_(dict.lookupOrDefault<word>("phi", "phi"))
{}
```

下面是 totalPressure 的一般设置方式：

```
patchName
{
    type          totalPressure;
    rho           rho;            // 可压缩流
    psi           none;           // 低速可压缩流
    gamma         1.4             // 可压缩流
    U             U;
    phi           phi;
    p0            uniform 1e5;
```

```
}
```

```
void Foam::totalPressureFvPatchScalarField::updateCoeffs()
{
    // 获取通量场，用来表明方向，后续根据其正负号来运算
    const fvsPatchField<scalar>& phip = patch().lookupPatchField<surfaceScalarField, scalar>(
        phiName_);

    // 获取速度场
    const fvPatchField<vector>& Up = patch().lookupPatchField<volVectorField, vector>(UName_);

    if (isA<pressureInletOutletVelocityFvPatchVectorField>(Up)) // 如果速度场的边界条件设为了
        pressureInletOutletVelocity
    {
        const pressureInletOutletVelocityFvPatchVectorField& Upiov =
            refCast<const pressureInletOutletVelocityFvPatchVectorField>(Up);

        if (Upiov.tangentialVelocity().valid())
        {
            const scalar t = this->db().time().userTimeValue();

            dynamicPressureFvPatchScalarField::updateCoeffs
            (
                p0_,
                0.5*neg(phip)*magSqr(Upiov.tangentialVelocity()->value(t))
              - 0.5*neg(phip)*magSqr(Up)
            );

            return;
        }
    }

    dynamicPressureFvPatchScalarField::updateCoeffs
    (
        p0_,
        -0.5*neg(phip)*magSqr(Up)
    );
}
```

可以看到 totalPressure 边界条件继承于 dynamicPressure 边界条件，就是通过字典关键词 p0 来设置 dynamicPressure 中 p0p 的值，并且确定了单位动能的计算方式为

$$K = \begin{cases} 0, & \phi \geq 0 \\ -\frac{1}{2}|\mathbf{U}|^2, & \phi < 0 \end{cases}$$

而当速度边界条件设置为 pressureInletOutletVelocity 时，单位动能的计算方式则有所不同：

$$K = \begin{cases} 0, & \phi \geq 0 \\ \frac{1}{2}|\mathbf{U}_{\tan}(t)|^2 - \frac{1}{2}|\mathbf{U}|^2, & \phi < 0 \end{cases}$$

3.5 温度边界条件

3.5.1 totalTemperature

这个边界条件提供了一个总温度条件。该边界条件继承于 fixedValue 边界条件。

```
Foam::totalTemperatureFvPatchScalarField::totalTemperatureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict, false),
    UName_(dict.lookupOrDefault<word>("U", "U")),         // 读取速度场名称，默认为U
    phiName_(dict.lookupOrDefault<word>("phi", "phi")),  // 读取通量场名称，默认为phi
    psiName_(dict.lookupOrDefault<word>("psi", "thermo:psi")), // 读取可压缩性场名称，默认为thermo:
        psi
    gamma_(dict.lookup<scalar>("gamma")),                 // 读取比热容比的赋值
    T0_("T0", dict, p.size())                             // 读取参考温度的赋值
{
    if (dict.found("value")) // 如果有value关键词则用value关键词的赋值进行初始化
    {
        fvPatchField<scalar>::operator=
        (
            scalarField("value", dict, p.size())
        );
    }
    else
    {
        fvPatchField<scalar>::operator=(T0_); // 否则使用T0的值进行初始化
    }
}
```

```
void Foam::totalTemperatureFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const fvPatchVectorField& Up = patch().lookupPatchField<volVectorField, vector>(UName_); // 获
        取速度场
    const fvsPatchField<scalar>& phip = patch().lookupPatchField<surfaceScalarField, scalar>(
        phiName_); // 获取通量场
    const fvPatchField<scalar>& psip = patch().lookupPatchField<volScalarField, scalar>(psiName_);
        // 获取可压缩性场
    scalar gM1ByG = (gamma_ - 1.0)/gamma_;
```

```
        operator==
        (
            T0_/(1.0 + 0.5*psip*gM1ByG*neg(phip)*magSqr(Up)) // 计算边界处的赋值
        );

        fixedValueFvPatchScalarField::updateCoeffs();
    }
```

上面的代码表明，totalTemperature 边界条件的边界温度计算方式为

$$T = \begin{cases} T_0, & \phi \geq 0 \\ \frac{T_0}{1+\frac{1}{2}\psi\frac{\gamma-1}{\gamma}|\mathbf{U}|^2}, & \phi < 0 \end{cases}$$

3.6 无反射边界条件

3.6.1 advective

参考https://blog.csdn.net/weixin_39124457/article/details/120152679。该边界条件提供了一个平流流出条件，基于求解边界处的 DDt(W, field) = 0，其中 W 为波速，field 为应用该边界条件的场。该边界条件支持标准时间格式 (Euler, backward, CrankNicolson, localEuler)。此外，还提供了一种可选机制，通过指定松弛长度尺度 lInf 和远场值 fieldInf，将边界处的值放松到指定的远场值。出口的流/波速度 (w) 由虚函数 advectionSpeed() 提供，该函数的默认实现需要通量场的名称 (phi)，如果给出的是质量通量而不是体积通量则还需要密度 (rho)。[5]

边界处无反射在物理意义上可以认为是该物理量在边界处的物质导数为零，即由当地时间变化所引起的变化率与由流体通过边界流出引起的变化率的和恒为零：

$$\frac{D\phi}{Dt} = \frac{\partial\phi}{\partial t} + \mathbf{U}\cdot\nabla\phi \approx \frac{\partial\phi}{\partial t} + U_n\cdot\frac{\partial\phi}{\partial \mathbf{n}} = 0$$

其中 ϕ 就是需要无反射处理的流场变量。在后续，$\frac{\partial\phi}{\partial t}$ 项需要根据时间离散格式进行不同的处理，U_n 则通过 advectionSpeed() 函数计算。

```
    template<class Type>
    Foam::advectiveFvPatchField<Type>::advectiveFvPatchField
    (
        const fvPatch& p,
        const DimensionedField<Type, volMesh>& iF,
        const dictionary& dict
    )
    :
        mixedFvPatchField<Type>(p, iF), // 继承于mixed边界条件
        phiName_(dict.lookupOrDefault<word>("phi", "phi")), // 获取通量场的名称
        rhoName_(dict.lookupOrDefault<word>("rho", "rho")), // 获取密度场的名称
        fieldInf_(Zero), // 远场值初始化为零
        lInf_(-great)    // 松弛长度尺度初始化为-great
    {
        if (dict.found("value")) // 如果设置了value
        {
            fvPatchField<Type>::operator=
            (
```

[5] 出口的流/波速度可以通过从这个类中派生一个专门的 BC 并覆盖 advectionSpeed() 来改变，例如在 waveTransmissiveFvPatchField 中，advectionSpeed() 计算并返回流速度加上声波速度，从而创建一个声波传输边界条件。

```
            Field<Type>("value", dict, p.size()) // 则用value初始化边界值
        );
    }
    else
    {
        fvPatchField<Type>::operator=(this->patchInternalField()); // 否则使用内部场的值初始化
    }

    this->refValue() = *this;  // refValue初始化为边界的当前值
    this->refGrad() = Zero;    // refGrad初始化为零
    this->valueFraction() = 0.0; // valueFraction初始化为零

    if (dict.readIfPresent("lInf", lInf_)) // 如果设置了松弛长度尺度则读取
    {
        dict.lookup("fieldInf") >> fieldInf_; // 并进一步读取远场值

        if (lInf_ < 0.0) // 如果松弛长度尺度为负数则报错
        {
            FatalIOErrorInFunction
                //...
        }
    }
}
```

上面的代码表明，refGrad 被初始化为零，并且后续也没有修改。如果在字典中设置了松弛长度尺度 lInf，那么它必须大于等于 0，否则将会报错；而且只有设置了 lInf 之后才会进一步读取远场值 fieldInf。

下面是 advective 边界条件一般的设置方式：

```
patchName
{
    type          advective;
    phi           phi;
    rho           rho;    // 可压缩流的情况
    lInf          1;      // 松弛长度尺度
    fieldInf      2902;   // 远场值
}
```

```
template<class Type>
Foam::tmp<Foam::scalarField>
Foam::advectiveFvPatchField<Type>::advectionSpeed() const
{
    const surfaceScalarField& phi = this->db().objectRegistry::template lookupObject<
        surfaceScalarField>(phiName_);
    // 获取通量场
    const fvsPatchField<scalar>& phip = this->patch().template lookupPatchField<surfaceScalarField,
         scalar>(phiName_);

    if (phi.dimensions() == dimMassFlux) // 可压缩流的情况，通量为质量通量
    {
        // 获取密度场
```

```
        const fvPatchScalarField& rhop = this->patch().template lookupPatchField<volScalarField,
            scalar>(rhoName_);

        return phip/(rhop*this->patch().magSf());
    }
    else
    {
        return phip/this->patch().magSf();
    }
}
```

上面的代码定义了 advectionSpeed() 函数返回的内容，返回值为

$$w = \begin{cases} \frac{\phi}{|\mathbf{S}_f|}, & \text{if } \phi \text{ is volume flux} \\ \frac{\phi}{\rho|\mathbf{S}_f|}, & \text{if } \phi \text{ is mass flux} \end{cases}$$

```
    template<class Type>
    void Foam::advectiveFvPatchField<Type>::updateCoeffs()
    {
        if (this->updated())
        {
            return;
        }

        const fvMesh& mesh = this->internalField().mesh();

        word ddtScheme
        (
            mesh.schemes().ddt(this->internalField().name())
        );
        scalar deltaT = this->db().time().deltaTValue(); // 获取时间步长

        const GeometricField<Type, fvPatchField, volMesh>& field = // 获取内部场
            this->db().objectRegistry::template
            lookupObject<GeometricField<Type, fvPatchField, volMesh>>
            (
                this->internalField().name()
            );

        // Calculate the advection speed of the field wave
        // If the wave is incoming set the speed to 0.
        const scalarField w(Foam::max(advectionSpeed(), scalar(0))); // 计算advectionSpeed，如果是入
            流则取0

        // Calculate the field wave coefficient alpha (See notes)
        const scalarField alpha(w*deltaT*this->patch().deltaCoeffs()); // 计算场波系数alpha

        label patchi = this->patch().index();

        // Non-reflecting outflow boundary
```

```
// If lInf_ defined setup relaxation to the value fieldInf_.
if (lInf_ > 0) // 如果松弛长度尺度大于0
{
    // Calculate the field relaxation coefficient k (See notes)
    const scalarField k(w*deltaT/lInf_); // 计算场松弛系数k

    if   // 如果时间离散格式设置的是Euler或CrankNicolson格式
    (
        ddtScheme == fv::EulerDdtScheme<scalar>::typeName
     || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
    )
    {
        this->refValue() = (field.oldTime().boundaryField()[patchi] + k*fieldInf_)/(1.0 + k);
             // 计算refValue
        this->valueFraction() = (1.0 + k)/(1.0 + alpha + k); // 计算valueFraction
    }
    else if (ddtScheme == fv::backwardDdtScheme<scalar>::typeName) // 如果时间离散格式设置的
         是backward格式
    {
        this->refValue() = // 计算refValue
        (
            2.0*field.oldTime().boundaryField()[patchi]
          - 0.5*field.oldTime().oldTime().boundaryField()[patchi]
          + k*fieldInf_
        )/(1.5 + k);

        this->valueFraction() = (1.5 + k)/(1.5 + alpha + k); // 计算valueFraction
    }
    else if  // 如果时间离散格式设置的是localEuler格式
    (
        ddtScheme == fv::localEulerDdtScheme<scalar>::typeName
    )
    {
        const volScalarField& rDeltaT = fv::localEulerDdt::localRDeltaT(mesh);

        // Calculate the field wave coefficient alpha (See notes)
        const scalarField alpha // 更新场波系数alpha
        (
            w*this->patch().deltaCoeffs()/rDeltaT.boundaryField()[patchi]
        );

        // Calculate the field relaxation coefficient k (See notes)
        const scalarField k(w/(rDeltaT.boundaryField()[patchi]*lInf_)); // 更新场松弛系数k

        this->refValue() = (field.oldTime().boundaryField()[patchi] + k*fieldInf_)/(1.0 + k);
             // 计算refValue
        this->valueFraction() = (1.0 + k)/(1.0 + alpha + k); // 计算valueFraction
    }
    else   // 当设置的时间离散格式不是Euler,CrankNicolson,backward,localEuler中的其中一种时将
```

```
        报错
    {
        FatalErrorInFunction
            //...
    }
}
else   // 如果松弛长度尺度等于0
{
    if   // 如果时间离散格式设置的是Euler或CrankNicolson格式
    (
        ddtScheme == fv::EulerDdtScheme<scalar>::typeName
     || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
    )
    {
        this->refValue() = field.oldTime().boundaryField()[patchi]; // 计算refValue

        this->valueFraction() = 1.0/(1.0 + alpha); // 计算valueFraction
    }
    else if (ddtScheme == fv::backwardDdtScheme<scalar>::typeName) // 如果时间离散格式设置的
        是backward格式
    {
        this->refValue() = // 计算refValue
        (
            2.0*field.oldTime().boundaryField()[patchi]
          - 0.5*field.oldTime().oldTime().boundaryField()[patchi]
        )/1.5;

        this->valueFraction() = 1.5/(1.5 + alpha); // 计算valueFraction
    }
    else if  // 如果时间离散格式设置的是localEuler格式
    (
        ddtScheme == fv::localEulerDdtScheme<scalar>::typeName
    )
    {
        const volScalarField& rDeltaT = fv::localEulerDdt::localRDeltaT(mesh);

        // Calculate the field wave coefficient alpha (See notes)
        const scalarField alpha // 更新场波系数alpha
        (
            w*this->patch().deltaCoeffs()/rDeltaT.boundaryField()[patchi]
        );

        this->refValue() = field.oldTime().boundaryField()[patchi]; // 计算refValue
        this->valueFraction() = 1.0/(1.0 + alpha); // 计算valueFraction
    }
    else   // 当设置的时间离散格式不是Euler,CrankNicolson,backward,localEuler中的其中一种时将
        报错
    {
        FatalErrorInFunction
```

```
                //...
            }
        }

        mixedFvPatchField<Type>::updateCoeffs();
    }
```

上面的代码表明，advective 边界条件设置的边界值与松弛长度尺度的设置以及时间离散格式的设置有关。当松弛长度尺度 lInf 设置为 0 时，refValue 设置方式如下：

1. 如果时间离散格式为 Euler 或 CrankNicolson 或 localEuler，则

$$\text{refValue} = \phi_{p_{\text{old}}}$$

2. 如果时间离散格式为 backward，则

$$\text{refValue} = \frac{2}{3} \times (2\phi_{p_{\text{old}}} - \frac{1}{2}\phi_{p_{\text{oldold}}})$$

当松弛长度尺度 lInf 设置为 0 时，valueFraction 的设置方式如下：

1. 如果时间离散格式为 Euler 或 CrankNicolson，则

$$\text{valueFraction} = \frac{1}{1+\alpha} = \frac{1}{1 + w\Delta t \times \frac{1}{d}}$$

2. 如果时间离散格式为 backward，则

$$\text{valueFraction} = \frac{1.5}{1.5+\alpha} = \frac{1.5}{1.5 + w\Delta t \times \frac{1}{d}}$$

3. 如果时间离散格式为 localEuler，则

$$\text{valueFraction} = \frac{1}{1+\alpha'} = \frac{1}{1 + (w \times \frac{1}{d})/\text{rDeltaT}}$$

当松弛长度尺度 lInf 设置为大于零的时候，refValue 的设置方式如下：

1. 如果时间离散格式为 Euler 或 CrankNicolson，则

$$\text{refValue} = \frac{\phi_{p_{\text{old}}} + k \times \text{fieldInf}}{1+k} = \frac{\phi_{p_{\text{old}}} + \frac{w\Delta t}{\text{lInf}} \times \text{fieldInf}}{1 + \frac{w\Delta t}{\text{lInf}}}$$

2. 如果时间离散格式为 backward，则

$$\text{refValue} = \frac{2\phi_{p_{\text{old}}} - 0.5\phi_{p_{\text{oldold}}} + k \times \text{fieldInf}}{1.5+k} = \frac{2\phi_{p_{\text{old}}} - 0.5\phi_{p_{\text{oldold}}} + \frac{w\Delta t}{\text{lInf}} \times \text{fieldInf}}{1.5 + \frac{w\Delta t}{\text{lInf}}}$$

3. 如果时间离散格式为 localEuler，则

$$\text{refValue} = \frac{\phi_{p_{\text{old}}} + k' \times \text{fieldInf}}{1+k'} = \frac{\phi_{p_{\text{old}}} + \frac{w}{\text{rDeltaT}\times\text{lInf}} \times \text{fieldInf}}{1 + \frac{w}{\text{rDeltaT}\times\text{lInf}}}$$

当松弛长度尺度 lInf 设置为大于零的时候，valueFraction 的设置方式如下：

1. 如果时间离散格式为 Euler 或 CrankNicolson，则

$$\text{valueFraction} = \frac{1+k}{1+\alpha+k} = \frac{1 + \frac{w\Delta t}{\text{lInf}}}{1 + \frac{w\Delta t}{d} + \frac{w\Delta t}{\text{lInf}}}$$

2. 如果时间离散格式为 backward，则

$$\text{valueFraction} = \frac{1.5+k}{1.5+\alpha+k} = \frac{1.5 + \frac{w\Delta t}{\text{lInf}}}{1.5 + \frac{w\Delta t}{d} + \frac{w\Delta t}{\text{lInf}}}$$

3. 如果时间离散格式为 localEuler，则

$$\text{valueFraction} = \frac{1+k'}{1+\alpha'+k'} = \frac{1 + \frac{w}{\text{rDeltaT}\times\text{lInf}}}{1 + \frac{w}{d\times\text{rDeltaT}} + \frac{w}{\text{rDeltaT}\times\text{lInf}}}$$

下面以 Euler 格式以及 lInf 设置为 0 的情况为例，说明各个系数为何设置成如上形式：

$$\frac{\partial \phi}{\partial t} + U_n \cdot \frac{\partial \phi}{\partial \mathbf{n}} = \frac{\phi_p - \phi_{\text{old}}}{\Delta t} + w \times \frac{\phi_p - \phi_c}{d}$$

于是可以得到

$$\phi_p = \frac{\frac{1}{\Delta t}\phi_{\text{old}}}{\frac{1}{\Delta t} + w \times \frac{1}{d}} + \frac{w \times \frac{1}{d}\phi_c}{\frac{1}{\Delta t} + w \times \frac{1}{d}} = \left(1 - \frac{1}{1 + w\Delta t \times \frac{1}{d}}\right)\phi_c + \frac{1}{1 + w\Delta t \times \frac{1}{d}}\phi_{\text{old}}$$

据此可以得到 refValue、refGrad 和 valueFraction 的表达式。

对于 lInf 设置大于零的情况，以 Euler 格式为例，用上述方法进行整理可以得到其求解的方程为

$$\frac{\partial \phi}{\partial t} + U_n \cdot \frac{\partial \phi}{\partial \mathbf{n}} = \frac{\phi_p - \phi_{p_{\text{old}}}}{\Delta t} + w \times \left(\frac{\phi_p - \phi_c}{d} + \frac{\phi_p - \text{fieldInf}}{\text{lInf}}\right) = 0$$

由此可知，lInf 可以理解为当前边界面与设想远场位置的距离。

3.6.2 waveTransmissive

该边界条件通过求解边界处 DDt(W, field) = 0 提供了一个波透射流出条件，W 为波速，field 为应用该边界条件的场。该边界条件继承于 advective 边界条件，不过求解的方程改为了

$$\frac{D\phi}{Dt} \approx \frac{\partial \phi}{\partial t} + (U_n + c) \cdot \frac{\partial \phi}{\partial \mathbf{n}} = 0$$

所以 waveTransmissive 边界条件就是设置了自己的 advectionSpeed() 函数，其中添加了音速。

```
template<class Type>
Foam::tmp<Foam::scalarField>
Foam::waveTransmissiveFvPatchField<Type>::advectionSpeed() const
{
    // Lookup the velocity and compressibility of the patch
    const fvPatchField<scalar>& psip = // 获取可压缩性场
        this->patch().template
            lookupPatchField<volScalarField, scalar>(psiName_);

    const surfaceScalarField& phi =
        this->db().template lookupObject<surfaceScalarField>(this->phiName_);

    scalarField phip // 获取通量场
    (
        this->patch().template
            lookupPatchField<surfaceScalarField, scalar>(this->phiName_)
    );

    if (phi.dimensions() == dimMassFlux) // 可压缩流的情况，通量为质量通量
    {
        const fvPatchScalarField& rhop = // 获取密度场
            this->patch().template
                lookupPatchField<volScalarField, scalar>(this->rhoName_);

        phip /= rhop;  // 提前将通量除以密度，统一为体积通量
    }

    // Calculate the speed of the field wave w
    // by summing the component of the velocity normal to the boundary
    // and the speed of sound (sqrt(gamma_/psi)).
    return phip/this->patch().magSf() + sqrt(gamma_/psip); // 求解波速度w
}
```

上面的代码表明，在 waveTransmissive 边界条件中，advectionSpeed() 返回的波速度被修改为了

$$w = \begin{cases} \frac{\phi}{|\mathbf{S}_f|} + \sqrt{\frac{\gamma}{\psi}}, & \text{if } \phi \text{ is volume flux} \\ \frac{\phi}{\rho|\mathbf{S}_f|} + \sqrt{\frac{\gamma}{\psi}}, & \text{if } \phi \text{ is mass flux} \end{cases}$$

下面是 waveTransmissive 边界条件的一般设置方式：

```
patchName
{
    type          waveTransmissive;
    phi           phi;
    psi           psi;
    gamma         1.4;
    lInf          1;
    fieldInf      2900;
}
```

3.7 自定义边界条件

3.7.1 codedFixedValue

下面是在射流外设置一个线性分布的速度场的例子：

```
AirInlet
{
    type       codedFixedValue;
    value      uniform (0 0 6.10);

    name   linearVelocity;
    code
    #{
        const vectorField& Cf = patch().Cf(); //center of the patch
        vectorField& field = *this;       //the target velocity field

        const scalar Umax = 6.1;
        const scalar r1 = 0.02;           //radius of the hot coflow
        const scalar r2 = 0.06;           //radius of the domain

        forAll(Cf, faceI)
        {
            const scalar x = Cf[faceI][0];
            const scalar y = Cf[faceI][1];
            const scalar R = sqrt(x*x+y*y);

            if(R>r1){
                field[faceI] = vector(0,0,Umax*(r2-R)/(r2-r1));
            }
        }
    #};
}
```

其他应用案例：

```
code
#{
    const fvPatch& boundaryPatch = patch();
    const vectorField& Cf = boundaryPatch.Cf();
    vectorField& field = *this;

    const scalar pi = constant::mathematical::pi;
    scalar x0=0.654157, z0=0.115013, d=0.004554;
    scalar umax=26.6, fjet=138.5;
    scalar theta = 18.31163/pi;

    forAll(Cf, faceI)
    {
        scalar x = Cf[faceI].x();
        scalar z = Cf[faceI].z();
        scalar t = this->db().time().value();

        scalar kesi = pow( (x-x0)*(x-x0)+(z-z0)*(z-z0),0.5);
        scalar ujmag = 6*umax*( (kesi/d)-pow(kesi/d,2) )*sin(2*pi*fjet*t);
        scalar ujet = ujmag*sin(theta);
        scalar vjet = ujmag*cos(theta);

        field[faceI] = vector(ujet,0,vjet);
    }
#};
```

```
inlet
{
    type          codedFixedValue;
    value         uniform 0;
    name          codeinlet_air;

    code
    #{
        const vectorField& Cf = patch().Cf();

        scalarField& field = *this;

        scalar D = 0.0252;
        scalar ymax = 5.82*D;

        forAll(Cf,faceI)
        {
            if ( Cf[faceI].y() > ymax )
            {
              field[faceI] = 1;
            }
        }
```

```
    #};
}
```

```
inlet
{
    type          codedFixedValue;
    value         uniform (0 0 0); //default value
    name          powvelocity;    //name of new BC type

    code
    #{
        const fvPatch &boundaryPatch = patch();
        const fvBoundaryMesh &boundaryMesh = boundaryPatch.boundaryMesh();
        const fvMesh &mesh = boundaryMesh.mesh();
        const fvPatchField<vector> &U = boundaryPatch.lookupPatchField<volVectorField, vector>("U");

        const vectorField FC = mesh.Cf();
        const scalarField y = FC&vector(0,1,0);

        vectorField U1 (U.size(), Zero);
        // const scalar pi = M_PI;
        const scalar U_0 = 0.5;
        const scalar D = 0.03;

        forAll(boundaryPatch, i)
        {
            U1[i] = vector(U_0*(pow(y[i]/D,1/7)), 0., 0.);
        }

        operator==(U1);
    #};
}
```

如需要设置随时间变化的边界条件，可采用如下方法得到 OF 运行时间：

```
scalar t = this->db().time().value();
```

添加下面的语句来解决 error: 'mesh' was not declared in this scope 的问题：

```
codeOptions
#{
-I$(LIB_SRC)/meshTools/lnInclude
#};
```

3.8 壁面函数

壁面函数作为一种边界条件继承于 FvPatchField 类，它们提供 Dirichlet 和 Neumann 边界条件，继承于 fixedValue 和 zeroGradient 边界条件。它们的实现将会涉及到 upateCoeffs() 和 evaluate() 功能。另外，OpenFOAM 还使用了 yPlusLam 这个量来表示粘性底层与对数分布区的交界位置，对于一些提供了两种计算模式的壁面函数，则

会在类的最开始计算 yPlusLam。同时，壁面函数的计算过程也可以分为两类，一类是计算第一层网格面上的值，另一类则是计算临近壁面的网格中心的值。

OpenFOAM 的壁面函数定义在 src/MomentumTransportModels/momentumTransportModels/derivedFvPatchFields/wallFunctio 当中，总共有 6 大类壁面函数，而各类细分后总共有 12 种壁面函数。在壁面函数的构建过程中，会使用到 src/MomentumTransportModels/momentumTransportModels/momentumTransportModel.H 中定义的调用函数来获取所需物理量。

3.8.1 nutWallFunction

该边界条件是一个抽象类，提供了基于湍流动能湍流运动粘度条件。不少其他壁面函数会继承这里面定义的一些调用函数。在 nutWallFunctionFvPatchScalarField.H 文件中可以看到声明的变量的含义及相关的调用函数。在直接使用该壁面函数时，将会读取设置中的 Cmu、kappa 和 E 关键词的赋值。

nutWallFunction 继承于 fixedValue 边界条件，默认情况下会初始化 $C_\mu = 0.09$ ， $\kappa = 0.41$ ， $E = 9.8$ 。

```
Foam::nutWallFunctionFvPatchScalarField::nutWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict), // 继承于fixedValue
    Cmu_(dict.lookupOrDefault<scalar>("Cmu", 0.09)),
    kappa_(dict.lookupOrDefault<scalar>("kappa", 0.41)),
    E_(dict.lookupOrDefault<scalar>("E", 9.8)),
    yPlusLam_(yPlusLam(kappa_, E_))
{
    checkType();
}
```

下面的代码定义了 yPlusLam 是如何通过 kappa 和 E 的值进行计算的：

```
Foam::scalar Foam::nutWallFunctionFvPatchScalarField::yPlusLam
(
    const scalar kappa,
    const scalar E
)
{
    scalar ypl = 11.0;

    for (int i=0; i<10; i++)
    {
        ypl = log(max(E*ypl, 1))/kappa;
    }

    return ypl;
}
```

3.8.2 kqRWallFunction

该边界条件为高雷诺数流动情况下的湍流 k、q 和 R 场提供了合适的条件。它本质上是对 zeroGradient 条件的简单包装。

```
template<class Type>
void Foam::kqRWallFunctionFvPatchField<Type>::evaluate
(
    const Pstream::commsTypes commsType
)
{
    zeroGradientFvPatchField<Type>::evaluate(commsType);
}
```

3.8.3 kLowReWallFunction

该边界条件为低雷诺数和高雷诺数湍流流动情况提供了湍流动能壁函数条件。该模型在两种模式下运行，基于计算得到的层流-湍流转换 y+ 值，该值由对应的 nutWallFunction 中指定的 kappa 和 E 导出。

```
kLowReWallFunctionFvPatchScalarField::kLowReWallFunctionFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchField<scalar>(p, iF, dict),
    Ceps2_(dict.lookupOrDefault<scalar>("Ceps2", 1.9))
{}
```

上面的代码表明 kLowReWallFunction 边界条件其实继承于 fixedValue 边界条件，并初始化 Ceps2 的值为 1.9。

```
void kLowReWallFunctionFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const label patchi = patch().index();

    const momentumTransportModel& turbModel =
        db().lookupObject<momentumTransportModel>
        (
            IOobject::groupName
            (
                momentumTransportModel::typeName,
                internalField().group()
            )
        );
```

```
    const nutWallFunctionFvPatchScalarField& nutw =
        nutWallFunctionFvPatchScalarField::nutw(turbModel, patchi);

    const scalarField& y = turbModel.y()[patchi]; // 定义第一层网格中心到壁面的距离y

    const tmp<volScalarField> tk = turbModel.k(); // 定义第一层网格中心的湍动能k
    const volScalarField& k = tk();

    const tmp<scalarField> tnuw = turbModel.nu(patchi); // 定义层流粘性nuw
    const scalarField& nuw = tnuw();

    const scalar Cmu25 = pow025(nutw.Cmu()); // 计算了Cmu系数的0.25次幂，记作Cmu25

    scalarField& kw = *this;

    // Set k wall values
    forAll(kw, facei)
    {
        label celli = patch().faceCells()[facei];

        scalar uTau = Cmu25*sqrt(k[celli]); // 计算摩擦速度

        scalar yPlus = uTau*y[facei]/nuw[facei]; // 计算y+

        if (yPlus > nutw.yPlusLam())
        {
            scalar Ck = -0.416;
            scalar Bk = 8.366;
            kw[facei] = Ck/nutw.kappa()*log(yPlus) + Bk;
        }
        else
        {
            scalar C = 11.0;
            scalar Cf = (1.0/sqr(yPlus + C) + 2.0*yPlus/pow3(C) - 1.0/sqr(C));
            kw[facei] = 2400.0/sqr(Ceps2_)*Cf;
        }

        kw[facei] *= sqr(uTau); // 更新壁面的湍流动能
    }

    // Limit kw to avoid failure of the turbulence model due to division by kw
    kw = max(kw, small); // 避免后续除以kw的时候出现数值错误，即防止除以零

    fixedValueFvPatchField<scalar>::updateCoeffs();

    // TODO: perform averaging for cells sharing more than one boundary face
}
```

从上面的代码可以看到阻力速度 u_τ 和 y^+ 的计算方式为

$$u_\tau = C_\mu^{0.25}\sqrt{k}, \quad y^+ = \frac{u_\tau y}{\nu}$$

壁面处的 k^+ 计算方式为

$$k^+ = \begin{cases} \frac{C_k}{\kappa}\log(y^+) + B_k, & y^+ > \text{yPlusLam} \\ \frac{2400}{C_{eps}^2} \times \left(\frac{1}{(y^+ + C)^2} + \frac{2y^+}{C^3} - \frac{1}{C^2}\right), & y^+ \le \text{yPlusLam} \end{cases}$$

其中 $C_k = -0.416$ ， $B_k = 8.366$; $C = 11$ ， C_{eps2} 通过读取设置赋值得到。

最后将无量纲数 k^+ 计算为壁面边界处的 k 的方式为:

$$k = k^+ u_\tau^2$$

3.8.4 v2WallFunction

该边界条件为低雷诺数和高雷诺数湍流流动情况提供了一个垂直于流线壁函数的湍流应力条件。该模型在两种模式下运行，基于计算得到的层流-湍流转换 y+ 值，该值由对应的 nutWallFunction 中指定的 kappa 和 E 导出。

```
void v2WallFunctionFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const label patchi = patch().index();

    const momentumTransportModel& turbModel =
        db().lookupObject<momentumTransportModel>
        (
            IOobject::groupName
            (
                momentumTransportModel::typeName,
                internalField().group()
            )
        );

    const nutWallFunctionFvPatchScalarField& nutw =
        nutWallFunctionFvPatchScalarField::nutw(turbModel, patchi);

    const scalarField& y = turbModel.y()[patchi]; // 定义到壁面的距离y

    const tmp<volScalarField> tk = turbModel.k(); // 定义湍流动能k
    const volScalarField& k = tk();

    const tmp<scalarField> tnuw = turbModel.nu(patchi); // 定义层流粘性nuw
    const scalarField& nuw = tnuw();

    const scalar Cmu25 = pow025(nutw.Cmu()); // 计算了Cmu系数的0.25次幂，记作Cmu25
```

```
    scalarField& v2 = *this;

    // Set v2 wall values
    forAll(v2, facei)
    {
        label celli = patch().faceCells()[facei];

        scalar uTau = Cmu25*sqrt(k[celli]); // 计算摩擦速度

        scalar yPlus = uTau*y[facei]/nuw[facei]; // 计算y+

        if (yPlus > nutw.yPlusLam())
        {
            scalar Cv2 = 0.193;
            scalar Bv2 = -0.94;
            v2[facei] = Cv2/nutw.kappa()*log(yPlus) + Bv2;
        }
        else
        {
            scalar Cv2 = 0.193;
            v2[facei] = Cv2*pow4(yPlus);
        }

        v2[facei] *= sqr(uTau); // 更新壁面的v2
    }

    fixedValueFvPatchField<scalar>::updateCoeffs();

    // TODO: perform averaging for cells sharing more than one boundary face
}
```

从上面的代码可以看到 $(v^2)^+$ 的计算方式为

$$(v^2)^+ = \begin{cases} \frac{C_{V^2}}{\kappa}\log(y^+) + B_{V^2}, & y^+ > \text{yPlusLam} \\ C_{V^2}(y^+)^4, & y^+ \le \text{yPlusLam} \end{cases}$$

其中 $C_{V^2} = 0.193$ ， $B_{V^2} = -0.94$ 。

最后将无量纲数 $(v^2)^+$ 计算为壁面处的 v^2 的方式为

$$v^2 = (v^2)^+ u_\tau^2$$

3.8.5 fWallFunction

该边界条件提供了湍流阻尼函数 f，低雷诺数和高雷诺数的壁面函数条件，湍流流动情况下的模型在两种模式下运行，基于计算得到的层流到湍流的转换 y+ 值，由对应的 nutWallFunction 中指定的 kappa 和 E 导出。

```
void fWallFunctionFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }
```

```
const label patchi = patch().index();

const momentumTransportModel& turbModel =
    db().lookupObject<momentumTransportModel>
    (
        IOobject::groupName
        (
            momentumTransportModel::typeName,
            internalField().group()
        )
    );
const v2fBase& v2fModel = refCast<const v2fBase>(turbModel);

const nutWallFunctionFvPatchScalarField& nutw =
    nutWallFunctionFvPatchScalarField::nutw(turbModel, patchi);

const scalarField& y = turbModel.y()[patchi]; // 定义到壁面的距离y

const tmp<volScalarField> tk = turbModel.k(); // 定义湍流动能k
const volScalarField& k = tk();

const tmp<volScalarField> tepsilon = turbModel.epsilon(); // 定义湍流动能耗散率epsilon
const volScalarField& epsilon = tepsilon();

const tmp<volScalarField> tv2 = v2fModel.v2(); // 定义垂直于流线的湍流应力v2
const volScalarField& v2 = tv2();

const tmp<scalarField> tnuw = turbModel.nu(patchi); // 定义层流粘度nuw
const scalarField& nuw = tnuw();

const scalar Cmu25 = pow025(nutw.Cmu()); // 计算了Cmu系数的0.25次幂，记作Cmu25

scalarField& f = *this;

// Set f wall values
forAll(f, facei)
{
    label celli = patch().faceCells()[facei];

    scalar uTau = Cmu25*sqrt(k[celli]); // 计算摩擦速度

    scalar yPlus = uTau*y[facei]/nuw[facei]; // 计算y+

    if (yPlus > nutw.yPlusLam())
    {
        scalar N = 6.0;
        scalar v2c = v2[celli];
        scalar epsc = epsilon[celli];
```

```
            scalar kc = k[celli];

            f[facei] = N*v2c*epsc/(sqr(kc) + rootVSmall);
            f[facei] /= sqr(uTau) + rootVSmall;
        }
        else
        {
            f[facei] = 0.0;
        }
    }

    fixedValueFvPatchField<scalar>::updateCoeffs();

    // TODO: perform averaging for cells sharing more than one boundary face
}
```

f 的计算方式为

$$f = \begin{cases} \frac{Nv^2\varepsilon}{k^2u_\tau^2}, & y^+ > \text{yPlusLam} \\ 0, & y^+ \leq \text{yPlusLam} \end{cases}$$

其中系数 $N = 6$ 。

注 该壁面函数的实现方式与理论存在较大差异，在使用该壁面函数时需要注意是否满足自身需求。

3.8.6 epsilonWallFunction

该边界条件为低雷诺数和高雷诺数湍流模型提供了湍流耗散壁约束。该条件可应用于壁面边界，计算出湍流耗散场和湍流产生场，并指定近壁面 ϵ 值。

3.9 滑流边界条件

参考文献Langmuir–Maxwell and Langmuir–Smoluchowski boundary conditions for thermal gas flow simulations in hypersonic aerodynamics。在气体-表面相互作用中，气体分子可以按照相对于表面的流动方向分为入射分子和出射分子。在表面上形成的粘性曳力是由于入射流的切向动量与出射流的切向动量的差异产生的。气体分子的切向动量贡献了剪切力 $\vec{F}$ 。Maxwell 认为入射气体分子和出射气体分子对表面的总剪切应力有着相同的贡献，即

$$(\frac{1}{2}\vec{F})_{\text{approaching}} + (\frac{1}{2}\vec{F})_{\text{receding}} = (\vec{F})_{\text{total}}$$

从表面出射的分子流被假设等价于一个简单的射流。一个典型的射流速度分布将会是独立分子流通过一个平面上的小孔所形成的流动，这个小孔的直径的量级是分子平均自由程。射流的总分子数 Γ_n 可以用平均分子速度 $\bar{v}$ 和分子数密度 η 来表达：

$$\Gamma_n = \frac{1}{4}\eta\bar{v}$$

上述离开表面的分子数量将与表面的滑动速度 $\vec{U}$ 相乘来得到出射流对表面总剪切力的贡献。滑流速度本身意味着在表面处的气体的切向动量被部分保留。为了确定滑流速度同时保证动量守恒，Maxwell 引入了切向动量适应系数 σ_u 。考虑到 $\rho = \eta m$ ，就可以得到

$$\sigma_u\left(\frac{1}{2}\vec{F} + \frac{1}{4}\rho\bar{v}\vec{U}\right) = \vec{F}$$

上式可以改写为

$$\vec{U} = 2\left(\frac{2-\sigma_u}{\sigma_u}\right)\frac{\vec{F}}{\rho\bar{v}}$$

平均分子速度 $\bar{v}$ 可以由 Maxwell 平衡态分布得到

$$\bar{v} = 2\sqrt{\frac{2RT}{\pi}}$$

而 Maxwell 分子模型定义的平均自由程的表达为

$$\lambda = \frac{\mu}{\rho}\sqrt{\frac{\pi}{2RT}}$$

那么就能得到 Maxwell 滑移速度边界更加一般的表达：

$$\vec{U} = \left(\frac{2-\sigma_u}{\sigma_u}\right)\frac{\lambda}{\mu}\vec{F}$$

上式忽略了热蠕变的影响，实际上表面切向的温度梯度会朝着温度增加的方向产生额外的滑移速度。在三维情况下、考虑了热蠕动的一般 Maxwell 滑流边界可以写成如下的形式：

$$\vec{U} = \left(\frac{2-\sigma_u}{\sigma_u}\right)\frac{\lambda}{\mu}\mathbf{S}\cdot(\vec{n}\cdot\boldsymbol{\tau}) - \frac{3}{4}\frac{\mathrm{Pr}(\gamma-1)}{\gamma p}\vec{q} + \vec{U}_w$$

其中 $\vec{n}$ 是表面的单位法向量 (指向流域)[6]，$\mathbf{S} = \mathbf{I} - \vec{n}\otimes\vec{n}$ 是切向转换张量，$\vec{q} = \mathbf{S}\cdot\vec{Q}$ 是表面切向的热通量，$\boldsymbol{\tau} = \mu\Big(\nabla\vec{U} + (\nabla\vec{U})^T - \frac{2}{3}\mathbf{I}\,\mathrm{tr}(\nabla\vec{U})\Big)$ 是偏应力张量。将热通量 $\vec{Q} = -\kappa\nabla T$ 、理想气体状态方程 $p = \rho RT$ 以及 $\mathrm{Pr} = C_p\mu/\kappa$ 代入上式可以得到

$$\vec{U} - \vec{U}_w = \left(\frac{2-\sigma_u}{\sigma_u}\right)\frac{\lambda}{\mu}\mathbf{S}\cdot(\vec{n}\cdot\boldsymbol{\tau}) + \frac{3}{4}\frac{\mu}{\rho T}\mathbf{S}\cdot\nabla T$$

然而，直接按照上式进行数值计算会产生非常严重的稳定性问题。因此，OpenFOAM 采用了一种半隐式处理，其核心在于将应力分解为速度梯度部分以及偏斜部分 $\boldsymbol{\tau}_{\mathrm{MC}}$ ，即

$$\boldsymbol{\tau} = \mu\Big(\nabla\vec{U}\Big) + \mu\Big((\nabla\vec{U})^T - \frac{2}{3}\mathbf{I}\,\mathrm{tr}(\nabla\vec{U})\Big)$$

此时，$\mathbf{S}\cdot(\vec{n}\cdot\boldsymbol{\tau})$ 的速度梯度部分所贡献的项就可以近似表示为[7]

$$\mathbf{S}\cdot(\vec{n}\cdot\nabla\vec{U}) \approx \frac{\mathbf{S}\cdot\vec{U}_c - \vec{U}}{|\Delta\vec{x}|}$$

其中 $\vec{U}_c$ 表示边界相邻内网格的体心值，$|\Delta\vec{x}|$ 为相邻内网格体心到边界的距离。偏斜部分 $\boldsymbol{\tau}_{\mathrm{MC}}$ 的贡献则正常按照公式显式计算，经过整理即可得到

$$\left(1 + \frac{2-\sigma_u}{\sigma_u}\frac{\lambda}{|\Delta\vec{x}|}\right)\vec{U} = \vec{U}_w + \frac{2-\sigma_u}{\sigma_u}\frac{\lambda}{|\Delta\vec{x}|}\mathbf{S}\cdot\vec{U}_c + \frac{2-\sigma_u}{\sigma_u}\frac{\lambda}{\mu}\mathbf{S}\cdot(\vec{n}\cdot\boldsymbol{\tau}_{\mathrm{MC}}) + \frac{3}{4}\frac{\mu}{\rho T}\mathbf{S}\cdot\nabla T$$

在 rhoCentralFoam 配套的边界条件中 maxwellSlipU 边界的实现如下：

```
class maxwellSlipUFvPatchVectorField
:
    public mixedFixedValueSlipFvPatchVectorField
{
    //- Temperature field name, default = "T"
    word TName_;

    //- Density field name, default = "rho"
    word rhoName_;

    //- Compressibility field name, default = "thermo:psi"
    word psiName_;

    //- Dynamic viscosity field name, default = "thermo:mu"
```

[6]需要注意的是，在 OpenFOAM 中边界网格面的法向量总是指向计算域之外 (从域内有限体积元的角度来说是合理的)，因此在这里是方向相反的。

[7]这里实际上忽略了边界的取切向的操作，理论上可以使用矩阵求逆的方式严格地进行计算，但是考虑到 Maxwell 滑流边界本身建模的误差，这里进行的简化几乎不会造成结果的显著偏差。而且当能够保证最终的边界速度是切向的时候，这个简化是自洽的。

```
    word muName_;

    //- tauMC field name, default = "tauMC"
    word tauMCName_;

    // Accommodation coefficient
    scalar accommodationCoeff_;

    // Wall velocity
    vectorField Uwall_;

    // Include thermal creep term (default on)
    Switch thermalCreep_;

    // Include boundary curvature term (default on)
    Switch curvature_;
};
```

```
void Foam::maxwellSlipUFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const fvPatchScalarField& pmu = patch().lookupPatchField<volScalarField, scalar>(muName_);
    const fvPatchScalarField& prho = patch().lookupPatchField<volScalarField, scalar>(rhoName_);
    const fvPatchField<scalar>& ppsi = patch().lookupPatchField<volScalarField, scalar>(psiName_);

    Field<scalar> C1
    (
        sqrt(ppsi*constant::mathematical::piByTwo)
      * (2.0 - accommodationCoeff_)/accommodationCoeff_
    );

    Field<scalar> pnu(pmu/prho);
    valueFraction() = (1.0/(1.0 + patch().deltaCoeffs()*C1*pnu));

    refValue() = Uwall_;

    if (thermalCreep_)
    {
        const volScalarField& vsfT =
            this->db().objectRegistry::lookupObject<volScalarField>(TName_);
        label patchi = this->patch().index();
        const fvPatchScalarField& pT = vsfT.boundaryField()[patchi];
        Field<vector> gradpT(fvc::grad(vsfT)().boundaryField()[patchi]);
        vectorField n(patch().nf());
```

```
        refValue() -= 3.0*pnu/(4.0*pT)*transform(I - n*n, gradpT);
    }

    if (curvature_)
    {
        const fvPatchTensorField& ptauMC =
            patch().lookupPatchField<volTensorField, tensor>(tauMCName_);
        vectorField n(patch().nf());

        refValue() -= C1/prho*transform(I - n*n, (n & ptauMC));
    }

    mixedFixedValueSlipFvPatchVectorField::updateCoeffs();
}
```

3.10 温度跳跃边界

参考文献Langmuir–Maxwell and Langmuir–Smoluchowski boundary conditions for thermal gas flow simulations in hypersonic aerodynamics。实验观察发现表面处的稀薄气体的温度并不等于表面的温度 T_w，其差异称为温度跳跃，并且是由于边界法向方向的热通量导致的。与 Maxwell 处理滑流速度的方式类似，入射分子撞击单位面积表面的能量贡献了通过表面的热。Smoluchowski 假设入射流的分子的热传导与到边界有一定距离的区域的气体的热传导具有相同的量级，入射流和出射流分别贡献热传导的一半。为了确定温度跳跃同时保证能量守恒，引入了一个热适应系数 σ_T $(0 \leq \sigma_T \leq 1)$。气体与固体表面的理想能量交换对应 $\sigma_T = 1$，而无能量交换对应 $\sigma_T = 0$。因此我们得到

$$-(\kappa \nabla T \cdot \vec{n}) = \sigma_T \Big(-(\frac{1}{2}\kappa \nabla T \cdot \vec{n})_{\text{approaching}} - (\frac{1}{2}\kappa \nabla T \cdot \vec{n})_{\text{receding}} \Big)$$

其中热传导系数 κ 通过 Prandtl 数得到为 $\kappa = \frac{C_p \mu}{\text{Pr}}$，其中 $C_p = \gamma C_v$。单位质量的分子的内能就是 $e = C_p T$。与单位面积的边界撞击的分子的平动能量要比相同温度下的平衡态的分子平均平动动能大 $4/3$，于是碰撞分子的能量需要乘以一个系数 $(\gamma + 1)/2$，对应于能量 $(C_p + R/2)T$。与单位面积的边界碰撞的分子质量为 $\rho\bar{v}/4$，所以单位面积的内能为 $\rho\bar{v}(\gamma + 1)e/8$。通过出射流产生的热传导是气体内能 e 和表面内能 e_w，因此

$$\Big(\frac{1}{2}\kappa \nabla T \cdot \vec{n}\Big) = \frac{1}{8}\rho\bar{v}(\gamma + 1)(e - e_w)$$

考虑到 $\rho\bar{v} = 2\mu/\lambda$，$e = C_p T$ 以及 $e_w = C_p T_w$ 可以将上式改写为

$$\Big(\frac{1}{2}\kappa \nabla T \cdot \vec{n}\Big) = \frac{1}{4}\frac{\mu}{\lambda}(\gamma + 1)C_p(T - T_w)$$

最后就能够得到 Smoluchowski 温度跳跃边界条件的表达：

$$T - T_w = \frac{2 - \sigma_T}{\sigma_T}\frac{2\gamma}{(\gamma + 1)\text{Pr}}\lambda \nabla T \cdot \vec{n}$$

直接按照上式进行数值计算同样也有一定的稳定性问题，因此可以对 $\nabla T \cdot \vec{n}$ 采取隐式操作，此时

$$\nabla T \cdot \vec{n} = \frac{T_c - T}{|\Delta \vec{x}|}$$

其中 T_c 为边界相邻内网格体心值，$|\Delta \vec{x}|$ 为相邻内网格体心到边界的距离。经过整理即可得到

$$\Big(1 + \frac{2 - \sigma_T}{\sigma_T}\frac{2\gamma}{(\gamma + 1)\text{Pr}}\frac{\lambda}{|\Delta \vec{x}|}\Big)T = T_w + \frac{2 - \sigma_T}{\sigma_T}\frac{2\gamma}{(\gamma + 1)\text{Pr}}\frac{\lambda}{|\Delta \vec{x}|}T_c$$

在 rhoCentralFoam 配套的边界条件中 smoluchowskiJumpT 边界的实现如下：

```
class smoluchowskiJumpTFvPatchScalarField
:
```

```
    public mixedFvPatchScalarField
{
    //- Velocity field name, default = "U"
    word UName_;

    //- Density field name, default = "rho"
    word rhoName_;

    //- Compressibility field name, default = "thermo:psi"
    word psiName_;

    //- Dynamic viscosity field name, default = "thermo:mu"
    word muName_;

    //- Accommodation coefficient
    scalar accommodationCoeff_;

    //- Wall surface temperature
    scalarField Twall_;

    //- Heat capacity ratio (default 1.4)
    scalar gamma_;
};
```

```
void Foam::smoluchowskiJumpTFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const fvPatchScalarField& pmu =
        patch().lookupPatchField<volScalarField, scalar>(muName_);
    const fvPatchScalarField& prho =
        patch().lookupPatchField<volScalarField, scalar>(rhoName_);
    const fvPatchField<scalar>& ppsi =
        patch().lookupPatchField<volScalarField, scalar>(psiName_);
    const fvPatchVectorField& pU =
        patch().lookupPatchField<volVectorField, vector>(UName_);

    // Prandtl number reading consistent with rhoCentralFoam
    const dictionary& thermophysicalProperties =
        db().lookupObject<IOdictionary>(physicalProperties::typeName);

    dimensionedScalar Pr
    (
        "Pr",
        dimless,
        thermophysicalProperties.subDict("mixture").subDict("transport")
```

```
        .lookup("Pr")
    );

    Field<scalar> C2
    (
        pmu/prho
        *sqrt(ppsi*constant::mathematical::piByTwo)
        *2.0*gamma_/Pr.value()/(gamma_ + 1.0)
        *(2.0 - accommodationCoeff_)/accommodationCoeff_
    );

    Field<scalar> aCoeff(prho.snGrad() - prho/C2);
    Field<scalar> KEbyRho(0.5*magSqr(pU));

    valueFraction() = (1.0/(1.0 + patch().deltaCoeffs()*C2));
    refValue() = Twall_;
    refGrad() = 0.0;

    mixedFvPatchScalarField::updateCoeffs();
}
```

第 4 章 controlDict 设置

4.1 常规设置

位于 system 文件夹内的 controlDict 文件用于设置时间和输入输出控制，包括时间控制、计算结果输出控制、时间步长自适应控制、字典读取控制和运行时加载库或者函数五个部分。

时间控制部分 startFrom 关键词决定了开始计算的时间，可选的取值包括：

关键词	可用赋值	含义
startFrom	firstTime	所有时间目录中最早的时间步
	latestTime	所有时间目录中最近的时间步
	startTime	从指定开始的时间步开始计算，需要在下面指定 startTime 关键词的值
stopAt	endTime	在指定的结束时间停止计算，需要在下面指定 endTime 关键词的值
	writeNow	当前时间步完成时结束计算并且输出计算结果
	noWriteNow	当前时间步完成时结束计算但是不输出计算结果
	nextWrite	下一个时间步完成时结束计算并且输出计算结果
writeControl	timeStep	按照时间步输出，间隔由关键字 writeInterval 指定，单位为 [步]
	runTime	按照计算时间输出，间隔由关键字 writeInterval 指定，单位为 [s]
	adjustableRunTime	时间自适应下按照计算时间输出，间隔由关键字 writeInterval 指定，单位为 [s]
	cpuTime	按照 CPU 时间输出，间隔由关键字 writeInterval 指定，单位为 [s]
	clockTime	按照真实时间输出，间隔由关键字 writeInterval 指定，单位为 [s]
writeFormat	ascii	文本格式，由 writePrecision 关键词控制有效数字位数
	binary	二进制格式，直接将内存内容写入文件，能够减少文件存储空间，提高读写效率
writeCompression	uncompressed	不压缩
	compressed	gzip 压缩格式
timeFormat	fixed	±m.dddddd，其中 d 由 timePrecision 来控制
	scientific	±m.dddddd ± xx，其中 d 由 timePrecision 来控制
	general	使用 scientific 格式，默认小数点后有 4 位有效位数，也可通过 timePrecision 来调节

自适应时间步长需要指定自适应开关 adjustTimeStep，最大库朗数 maxCo 时间步长最大值 maxDeltaT 的数值，例如：

```
adjustTimeStep yes; //是否开启时间步长自适应
maxCo          0.5; //最大库郎数
maxDeltaT      1.0; //最大时间步长
```

在计算过程当中，一些字典文件可能会被修改，这时需要打开读取开关 runTimeModifiable，以让求解器更新一些参数。

4.2 etc/controlDict

位于 etc 文件夹内的 controlDict 文件配置了 OpenFOAM 的基本控制状态以及一些基本物理参数，例如将 DebugSwitches 字典内的 SolverPerformance 赋值从默认的 1 改为 0，则可以取消 solver 的输出信息。

```
DebugSwitches
{
    dimensionSet      1;
    fileName          2;
```

```
    GAMGAgglomeration 1;
    level            2;
    lduMatrix        1;
    SolverPerformance 0; // set to 1 for standard solver output
    vtkUnstructuredReader 1;
}
```

4.3 运行时加载库或者函数控制

设置运行时需要加载的库或函数如下所示：

```
libs      libUser1.so;  //库列表，库文件位于$LD_LIBRARY_PATH
functions probes;       //函数列表
```

在 controlDict 文件内的 functions 字典内添加相应的子字典设置来使用 postProcess 功能。通过在终端输入下面的命令进行后处理：

```
<solver> -postProcess
```

4.3.1 CourantNo

Calculates and outputs the Courant number as a volScalarField. The field is stored on the mesh database so that it can be retrieved and used for other applications.

```
 Foam::tmp<Foam::volScalarField::Internal>
 Foam::functionObjects::CourantNo::byRho
 (
     const tmp<volScalarField::Internal>& Co
 ) const
 {
     if (Co().dimensions() == dimDensity)
     {
         return Co/obr_.lookupObject<volScalarField>(rhoName_);
     }
     else
     {
         return Co;
     }
 }


 bool Foam::functionObjects::CourantNo::calc()
 {
     if (foundObject<surfaceScalarField>(fieldName_))
     {
         const surfaceScalarField& phi = lookupObject<surfaceScalarField>(fieldName_);

         tmp<volScalarField> tCo
         (
```

```
            volScalarField::New
            (
                resultName_,
                mesh_,
                dimensionedScalar(dimless, 0),
                zeroGradientFvPatchScalarField::typeName
            )
        );

        tCo->ref() = byRho( (0.5*mesh_.time().deltaT()) * fvc::surfaceSum(mag(phi))()()/mesh_.V() );

        tCo->correctBoundaryConditions();

        return store(resultName_, tCo);
    }
    else
    {
        cannotFindObject<surfaceScalarField>(fieldName_);

        return false;
    }
}
```

4.3.2 MachNo

Calculates and writes the Mach number as a volScalarField.

```
bool Foam::functionObjects::MachNo::calc()
{
    if
    (
        foundObject<volVectorField>(fieldName_)
     && foundObject<fluidThermo>(physicalProperties::typeName)
    )
    {
        const fluidThermo& thermo = lookupObject<fluidThermo>(physicalProperties::typeName);

        const volVectorField& U = lookupObject<volVectorField>(fieldName_);

        return store
        (
            resultName_,
            mag(U)/sqrt(thermo.gamma()*thermo.p()/thermo.rho())
        );
    }
    else
    {
        return false;
    }
```

```
}
```

4.3.3 forces

使用 forces 功能需要在 controlDict 的 functions 字典中添加如下内容：

```
forces1
{
    // Mandatory entries
    type          forces;
    libs          ("libforces.so");
    patches       (<list of patch names>);

    // Field names
    p             p;
    U             U;
    rho           rho;
    // 如果是不可压缩流则将rho修改为如下设置
    // rho       rhoInf;
    // rhoInf    100000;

    pRef          0;        // Reference pressure [Pa]
    porosity      no;       // Include porosity effects?
    writeFields   yes;      // Store and write volume field representations of forces and moments
    CofR          (0 0 0);  // Centre of rotation for moment calculations

    // Spatial data binning
    // - extents given by the bounds of the input geometry
    /* binData
    {
        nBin      20;
        direction (1 0 0);
        cumulative yes;
    } */
}
```

forces 功能读取 controlDict 中 functions 字典的逻辑如下：

```
bool Foam::functionObjects::forces::read(const dictionary& dict)
{
    fvMeshFunctionObject::read(dict);

    initialised_ = false;

    Log << type() << " " << name() << ":" << nl;

    directForceDensity_ = dict.lookupOrDefault("directForceDensity", false); // 是否直接提供force
        density, 默认为false

    const polyBoundaryMesh& pbm = mesh_.boundaryMesh();
```

```
patchSet_ = pbm.patchSet(wordReList(dict.lookup("patches"))); // 读取边界面名称

if (directForceDensity_) // 如果直接提供force density
{
    // Optional entry for fDName
    fDName_ = dict.lookupOrDefault<word>("fD", "fD"); // 读取fD
}
else   // 如果没有直接提供force density(默认情况)
{
    // Optional phase entry
    phaseName_ = dict.lookupOrDefault<word>("phase", word::null); // 读取相名称, 默认为空

    // Optional U, p and rho entries
    pName_ = dict.lookupOrDefault<word> ( "p", IOobject::groupName("p", phaseName_) ); // 读取压强
        场名称
    UName_ = dict.lookupOrDefault<word> ( "U", IOobject::groupName("U", phaseName_) ); // 读取速度
        场名称
    rhoName_ = dict.lookupOrDefault<word> ( "rho", IOobject::groupName("rho", phaseName_) ); //
        读取密度场名称

    // Reference density needed for incompressible calculations
    if (rhoName_ == "rhoInf") // 如果密度场名称设置为了rhoInf
    {
        dict.lookup("rhoInf") >> rhoRef_; // 那么rhoRef的值即为rhoInf的值
    }

    // Reference pressure, 0 by default
    pRef_ = dict.lookupOrDefault<scalar>("pRef", 0.0); // 读取参考压强, 默认为零
}

// Centre of rotation for moment calculations specified directly, from coordinate system, or
    implicitly (0 0 0)
if (dict.found("CofR"))
{
    coordSys_ = coordinateSystem("coordSys", vector(dict.lookup("CofR")));
    localSystem_ = false;
}
else
{
    coordSys_ = coordinateSystem("coordSys", dict);
    localSystem_ = true;
}

dict.readIfPresent("porosity", porosity_); // 孔隙度
if (porosity_)
{
    Log << "  Including porosity effects" << endl;
}
else
```

```
{
    Log << "  Not including porosity effects" << endl;
}

if (dict.found("binData"))
{
    const dictionary& binDict(dict.subDict("binData"));
    binDict.lookup("nBin") >> nBin_;

    if (nBin_ < 0)
    {
        FatalIOErrorInFunction(dict) << "Number of bins (nBin) must be zero or greater" << exit(
            FatalIOError);
    }
    else if ((nBin_ == 0) || (nBin_ == 1))
    {
        nBin_ = 1;
        forAll(force_, i)
        {
            force_[i].setSize(1);
            moment_[i].setSize(1);
        }
    }

    if (nBin_ > 1)
    {
        binDict.lookup("direction") >> binDir_;
        binDir_ /= mag(binDir_);

        binMin_ = great;
        scalar binMax = -great;
        forAllConstIter(labelHashSet, patchSet_, iter)
        {
            const label patchi = iter.key();
            const polyPatch& pp = pbm[patchi];
            const scalarField d(pp.faceCentres() & binDir_);
            binMin_ = min(min(d), binMin_);
            binMax = max(max(d), binMax);
        }
        reduce(binMin_, minOp<scalar>());
        reduce(binMax, maxOp<scalar>());

        // slightly boost binMax so that region of interest is fully within bounds
        binMax = 1.0001*(binMax - binMin_) + binMin_;

        binDx_ = (binMax - binMin_)/scalar(nBin_);

        // create the bin points used for writing
        binPoints_.setSize(nBin_);
```

```
            forAll(binPoints_, i)
            {
                binPoints_[i] = (i + 0.5)*binDir_*binDx_;
            }

            binDict.lookup("cumulative") >> binCumulative_;

            // allocate storage for forces and moments
            forAll(force_, i)
            {
                force_[i].setSize(nBin_);
                moment_[i].setSize(nBin_);
            }
        }
    }

    if (nBin_ == 1)
    {
        // allocate storage for forces and moments
        force_[0].setSize(1);
        force_[1].setSize(1);
        force_[2].setSize(1);
        moment_[0].setSize(1);
        moment_[1].setSize(1);
        moment_[2].setSize(1);
    }

    resetNames(createFileNames(dict));

    return true;
}
```

在 forces 功能中计算 forces 和 moment 的方式如下：

```
void Foam::functionObjects::forces::calcForcesMoment()
{
    initialise();

    force_[0] = Zero; // pressure force per bin
    force_[1] = Zero; // viscous force per bin
    force_[2] = Zero; // porous force per bin

    moment_[0] = Zero; // pressure moment per bin
    moment_[1] = Zero; // viscous moment per bin
    moment_[2] = Zero; // porous moment per bin

    if (directForceDensity_) // 如果直接提供了force density的数据
    {
        const volVectorField& fD = obr_.lookupObject<volVectorField>(fDName_); // 读取force density
```

```
    const surfaceVectorField::Boundary& Sfb = mesh_.Sf().boundaryField(); // 边界面的面法向量

    forAllConstIter(labelHashSet, patchSet_, iter)
    {
        const label patchi = iter.key();

        const vectorField Md ( mesh_.C().boundaryField()[patchi] - coordSys_.origin() );

        const scalarField sA(mag(Sfb[patchi])); // 边界面的网格面积

        // Normal force = surfaceUnitNormal*(surfaceNormal & forceDensity)
        const vectorField fN // 法向力
        (
            Sfb[patchi]/sA * ( Sfb[patchi] & fD.boundaryField()[patchi] )
        );

        // Tangential force (total force minus normal fN)
        const vectorField fT(sA*fD.boundaryField()[patchi] - fN); // 切向力

        //- Porous force
        const vectorField fP(Md.size(), Zero); // 孔隙力

        applyBins(Md, fN, fT, fP, mesh_.C().boundaryField()[patchi]);
    }
}
else   // 如果没有直接提供force density的数据(默认情况)
{
    const volScalarField& p = obr_.lookupObject<volScalarField>(pName_); // 读取压强

    const surfaceVectorField::Boundary& Sfb = mesh_.Sf().boundaryField(); // 边界面的面法向量

    tmp<volSymmTensorField> tdevTau = devTau();
    const volSymmTensorField::Boundary& devTaub = tdevTau().boundaryField(); // 在边界面处的切应力

    // Scale pRef by density for incompressible simulations
    const scalar pRef = pRef_/rho(p); // 如果是可压缩流则不会做任何事; 如果是不可压缩流则将pRef除以
        rhoRef(rhoInf)

    forAllConstIter(labelHashSet, patchSet_, iter)
    {
        const label patchi = iter.key();

        const vectorField Md
        (
            mesh_.C().boundaryField()[patchi] - coordSys_.origin()
        );

        const vectorField fN // 法向力
        (
```

```
                alpha(patchi) * rho(p) * Sfb[patchi] * (p.boundaryField()[patchi] - pRef)
            );

            const vectorField fT(Sfb[patchi] & devTaub[patchi]); // 切向力

            const vectorField fP(Md.size(), Zero); // 孔隙力

            applyBins(Md, fN, fT, fP, mesh_.C().boundaryField()[patchi]);
        }
    }

    if (porosity_) // 如果考虑孔隙作用
    {
        // ...
    }

    Pstream::listCombineGather(force_, plusEqOp<vectorField>());
    Pstream::listCombineGather(moment_, plusEqOp<vectorField>());
    Pstream::listCombineScatter(force_);
    Pstream::listCombineScatter(moment_);
}
```

其中 applyBins() 函数的具体执行内容如下，实际上由该函数计算出最终的 forces 和 moment：

```
void Foam::functionObjects::forces::applyBins
(
    const vectorField& Md,
    const vectorField& fN,
    const vectorField& fT,
    const vectorField& fP,
    const vectorField& d
)
{
    if (nBin_ == 1)
    {
        force_[0][0] += sum(fN);
        force_[1][0] += sum(fT);
        force_[2][0] += sum(fP);
        moment_[0][0] += sum(Md^fN);
        moment_[1][0] += sum(Md^fT);
        moment_[2][0] += sum(Md^fP);
    }
    else
    {
        scalarField dd((d & binDir_) - binMin_);

        forAll(dd, i)
        {
            label bini = min(max(floor(dd[i]/binDx_), 0), force_[0].size() - 1);
```

```
            force_[0][bini] += fN[i];
            force_[1][bini] += fT[i];
            force_[2][bini] += fP[i];
            moment_[0][bini] += Md[i]^fN[i];
            moment_[1][bini] += Md[i]^fT[i];
            moment_[2][bini] += Md[i]^fP[i];
        }
    }
}
```

4.3.4 forceCoeffs

使用 forceCoeffs 功能需要在 controlDict 的 functions 字典中添加如下内容:

```
forceCoeffs1
{
    // Mandatory entries
    type          forceCoeffs;
    libs          ("libforces.so");
    patches       (<list of patch names>);

    // Field names
    p             p;
    U             U;
    rho           rho;   // 不可压缩流则设置为rhoInf

    rhoInf     100000;       // Freestream density
    magUInf       30;        // Freestream velocity magnitude [m/s]
    pRef          0;         // Reference pressure [Pa]

    porosity      no;        // Include porosity effects?
    writeFields   yes;       // Store and write volume field representations of forces and moments
    CofR          (0 0 0);   // Centre of rotation for moment calculations
    liftDir       (0 0 1);   // Lift direction
    dragDir       (1 0 0);   // Drag direction
    pitchAxis     (0 1 0);   // Pitch axis

    lRef          1;         // Reference length [m]
    Aref          1.75;      // Reference area [m2]

    // Spatial data binning
    // - extents given by the bounds of the input geometry
    /* binData
    {
        nBin      20;
        direction (1 0 0);
        cumulative yes;
    } */
}
```

$$C_l = \frac{\mathbf{F}_{total} \cdot \mathbf{n}_{lift}}{A_{ref} \times (\frac{1}{2}\rho_{Inf}|\mathbf{U}_{Inf}|^2)}$$
$$C_d = \frac{\mathbf{F}_{total} \cdot \mathbf{n}_{drag}}{A_{ref} \times (\frac{1}{2}\rho_{Inf}|\mathbf{U}_{Inf}|^2)}$$
$$C_m = \frac{totMoment \cdot pitchAxis}{A_{ref} \times l_{ref} \times (\frac{1}{2}\rho_{Inf}|\mathbf{U}_{Inf}|^2)}$$

forceCoeffs 功能读取 controlDict 中 functions 字典的逻辑如下：

```
bool Foam::functionObjects::forceCoeffs::read(const dictionary& dict)
{
    forces::read(dict); // 按照forces功能的方式先读取一遍字典

    // Directions for lift and drag forces, and pitch moment
    // Normalise to ensure that the directions are unit vectors

    dict.lookup("liftDir") >> liftDir_; // 读取lift force方向
    liftDir_ /= mag(liftDir_); // 修正为单位向量

    dict.lookup("dragDir") >> dragDir_; // 读取drag force方向
    dragDir_ /= mag(dragDir_); // 修正为单位向量

    dict.lookup("pitchAxis") >> pitchAxis_; // 读取pitch坐标系方向
    pitchAxis_ /= mag(pitchAxis_); // 修正为单位向量

    // Free stream velocity magnitude
    dict.lookup("magUInf") >> magUInf_; // 读取自由流的速度大小

    // Reference (free stream) density
    dict.lookup("rhoInf") >> rhoRef_; // 读取自由流的密度

    // Reference length and area scales
    dict.lookup("lRef") >> lRef_; // 读取参考长度
    dict.lookup("Aref") >> Aref_; // 读取参考面积

    return true;
}
```

forceCoeffs 功能计算各力系数的方式如下，它首先通过 forces 功能计算总力，再点乘对应方向的单位向量得到升力、曳力等：

```
bool Foam::functionObjects::forceCoeffs::write()
{
    forces::calcForcesMoment(); // 计算 forces 和 moment

    if (Pstream::master())
    {
        logFiles::write();

        scalar pDyn = 0.5*rhoRef_*magUInf_*magUInf_; // 动压
```

```
Field<vector> totForce(force_[0] + force_[1] + force_[2]);
Field<vector> totMoment(moment_[0] + moment_[1] + moment_[2]);

List<Field<scalar>> coeffs(3);
coeffs[0].setSize(nBin_);
coeffs[1].setSize(nBin_);
coeffs[2].setSize(nBin_);

// lift, drag and moment
coeffs[0] = (totForce & liftDir_)/(Aref_*pDyn);
coeffs[1] = (totForce & dragDir_)/(Aref_*pDyn);
coeffs[2] = (totMoment & pitchAxis_)/(Aref_*lRef_*pDyn);

scalar Cl = sum(coeffs[0]); // total lift coefficient
scalar Cd = sum(coeffs[1]); // total drag coefficient
scalar Cm = sum(coeffs[2]); // total moment coefficient

scalar Clf = Cl/2.0 + Cm; // total front lift coefficient
scalar Clr = Cl/2.0 - Cm; // total rear lift coefficient

writeTime(file(fileID::mainFile));
file(fileID::mainFile) << tab << Cm << tab << Cd << tab << Cl << tab << Clf << tab << Clr <<
    endl;

Log << type() << " " << name() << " write:" << nl
   << "   Cm    = " << Cm << nl
   << "   Cd    = " << Cd << nl
   << "   Cl    = " << Cl << nl
   << "   Cl(f) = " << Clf << nl
   << "   Cl(r) = " << Clr << endl;

if (nBin_ > 1)
{
   if (binCumulative_)
   {
      for (label i = 1; i < coeffs[0].size(); i++)
      {
         coeffs[0][i] += coeffs[0][i-1];
         coeffs[1][i] += coeffs[1][i-1];
         coeffs[2][i] += coeffs[2][i-1];
      }
   }

   writeTime(file(fileID::binsFile));

   forAll(coeffs[0], i)
   {
      file(fileID::binsFile) << tab << coeffs[2][i] << tab << coeffs[1][i] << tab << coeffs
```

```
                    [0][i];
            }

            file(fileID::binsFile) << endl;
        }

        Log << endl;
    }

    return true;
}
```

4.3.5 Lambda2

Calculates and outputs the second largest eigenvalue of the sum of the square of the symmetrical and anti-symmetrical parts of the velocity gradient tensor.

```
bool Foam::functionObjects::Lambda2::calc()
{
    if (foundObject<volVectorField>(fieldName_))
    {
        const volVectorField& U = lookupObject<volVectorField>(fieldName_);
        const tmp<volTensorField> tgradU(fvc::grad(U));
        const volTensorField& gradU = tgradU();

        const volTensorField SSplusWW
        (
            (symm(gradU) & symm(gradU))
          + (skew(gradU) & skew(gradU))
        );

        return store
        (
            resultName_,
           -eigenValues(SSplusWW)().component(vector::Y)
        );
    }
    else
    {
        cannotFindObject<volVectorField>(fieldName_);

        return false;
    }
}
```

4.3.6 Q

Calculates and outputs the second invariant of the velocity gradient tensor.

```
bool Foam::functionObjects::Q::calc()
{
    if (foundObject<volVectorField>(fieldName_))
    {
        const volVectorField& U = lookupObject<volVectorField>(fieldName_);
        const tmp<volTensorField> tgradU(fvc::grad(U));
        const volTensorField& gradU = tgradU();

        return store
        (
            resultName_,
            0.5*(sqr(tr(gradU)) - tr(((gradU) & (gradU))))
        );
    }
    else
    {
        cannotFindObject<volVectorField>(fieldName_);

        return false;
    }
}
```

第 5 章 fvSchemes 设置

5.1 对流项离散设置

5.1.1 对流项离散的实现

Gauss 型对流项离散的实现见 src/finiteVolume/finiteVolume/convectionSchemes/gaussConvectionScheme/gaussConvectionSche
文件。对于隐式离散，调用的 fvmDiv 函数如下所示：

```
template<class Type>
tmp<fvMatrix<Type>>
gaussConvectionScheme<Type>::fvmDiv
(
    const surfaceScalarField& faceFlux,
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    tmp<surfaceScalarField> tweights = tinterpScheme_().weights(vf);
    const surfaceScalarField& weights = tweights();

    tmp<fvMatrix<Type>> tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            faceFlux.dimensions()*vf.dimensions()
        )
    );
    fvMatrix<Type>& fvm = tfvm.ref();

    fvm.lower() = -weights.primitiveField()*faceFlux.primitiveField();
    fvm.upper() = fvm.lower() + faceFlux.primitiveField();
    fvm.negSumDiag();

    forAll(vf.boundaryField(), patchi)
    {
        const fvPatchField<Type>& psf = vf.boundaryField()[patchi];
        const fvsPatchScalarField& patchFlux = faceFlux.boundaryField()[patchi];
        const fvsPatchScalarField& pw = weights.boundaryField()[patchi];

        fvm.internalCoeffs()[patchi] = patchFlux*psf.valueInternalCoeffs(pw);
        fvm.boundaryCoeffs()[patchi] = -patchFlux*psf.valueBoundaryCoeffs(pw);
    }

    if (tinterpScheme_().corrected())
    {
        fvm += fvc::surfaceIntegrate(faceFlux*tinterpScheme_().correction(vf));
    }
```

```
    return tfvm;
}
```

在 OpenFOAM 中，网格面的法向定义为从小编号网格 (owner) 指向大编号网格 (neighbor)，也就是说，如果记编号大于 P 的单元为 U，则此时 P 为 owner，U 为 neighbor，P 和 U 之间的网格面上的量在 OpenFOAM 中计算为

$$\phi_f = \varpi\phi_P + (1-\varpi)\phi_U, \quad \dot{m}_f = (\rho\vec{u})_f \cdot \vec{S}_f$$

记编号小于 P 的单元为 L，则此时 L 为 owner，P 为 neighbor，P 和 L 之间的网格面上的量在 OpenFOAM 中计算为

$$\phi_f = (1-\varpi)\phi_P + \varpi\phi_L, \quad \dot{m}_f = (\rho\vec{u})_f \cdot (-\vec{S}_f)$$

从而对流项的隐式离散为

$$\begin{aligned}
\sum_{f\sim nb(P)} (\rho\vec{u}\phi)_f \cdot \vec{S}_f &= \sum_{N\in L(P)} (-\dot{m}_f\phi_f) + \sum_{N\in U(P)} \dot{m}_f\phi_f \\
&= \sum_{N\in L(P)} -\dot{m}_f\big[(1-\varpi)\phi_P + \varpi\phi_N\big] + \sum_{N\in U(P)} \dot{m}_f\big[\varpi\phi_P + (1-\varpi)\phi_N\big] \\
&= \Big(\sum_{N\in L(P)} -\dot{m}_f(1-\varpi) + \sum_{N\in U(P)} \dot{m}_f\varpi\Big)\phi_P + \sum_{N\in L(P)} -\dot{m}_f\varpi\phi_N + \sum_{N\in U(P)} \dot{m}_f(1-\varpi)\phi_N
\end{aligned}$$

到这里就可以看出，fvm.lower() 存储的是 $-\varpi\dot{m}_f$，fvm.upper() 存储的是 $\dot{m}_f(1-\varpi)$，并且 diag 存储的是

$$a_P = -\Big(\sum_{N\in L(P)} \dot{m}_f(1-\varpi) + \sum_{N\in U(P)} (-\dot{m}_f\varpi)\Big)$$

而对流项的显式离散则调用了 fvcDiv 函数，其内容如下：

```
template<class Type>
tmp<GeometricField<Type, fvPatchField, volMesh>>
gaussConvectionScheme<Type>::fvcDiv
(
    const surfaceScalarField& faceFlux,
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    tmp<GeometricField<Type, fvPatchField, volMesh>> tConvection
    (
        fvc::surfaceIntegrate(flux(faceFlux, vf))
    );

    tConvection.ref().rename
    (
        "convection(" + faceFlux.name() + ',' + vf.name() + ')'
    );

    return tConvection;
}
```

5.2 梯度项离散

5.2.1 最小二乘法

对于一个网格及其相邻网格，有

$$T_N = T_P + (\nabla T)_P \cdot \vec{d}_{PN}$$

其中已知的是体心网格的量 T_P, T_N 以及两个网格之间的距离向量 $\vec{d}_{PN}$ ，而 $(\nabla T)_P$ 则是我们要求的量。对于一个多面体，其每一个相邻网格都能得到一条上述方程，这样将这些方程组合在一起，可以写成一个矩阵形式：

$$\mathbf{d}_{PN}(\nabla T)_P = \mathbf{T}_N - \mathbf{T}_P$$

在三维的情况下，假设网格一共有 m 个面，那么 $\mathbf{d}_{PN}$ 是一个 $m \times 3$ 的矩阵，$(\nabla T)_P$ 是一个 3×1 的向量，右侧的 $\mathbf{T}_N - \mathbf{T}_P$ 是一个 $m \times 1$ 的向量。于是采用最小二乘法可以求出

$$(\nabla T)_P = (\mathbf{d}_{PN}^T \mathbf{d}_{PN})^{-1} \mathbf{d}_{PN}^T (\mathbf{T}_N - \mathbf{T}_P)$$

记其中的 $3 \times m$ 矩阵 $\mathbf{G} = (\mathbf{d}_{PN}^T \mathbf{d}_{PN})^{-1} \mathbf{d}_{PN}^T$ ，如果在计算中网格没有发生变化，那么这个完全基于网格距离计算得到的矩阵 $\mathbf{G}$ 只需要计算一次，每次只需要更新 $\mathbf{T}_N - \mathbf{T}_P$ 即可，因此这种方法计算效率尚可。

上述最小二乘法称为 unweightedLeastSquares，从 unweightedLeastSquaresVectors.C 文件中可以看到计算矩阵 $\mathbf{G} = (\mathbf{d}_{PN}^T \mathbf{d}_{PN})^{-1} \mathbf{d}_{PN}^T$ 的方法：

```
//- src/finiteVolume/finiteVolume/gradSchemes/leastSquaresGrad/unweightedLeastSquaresVectors.C
const fvMesh& mesh = mesh_;

// Set local references to mesh data
const labelUList& owner = mesh_.owner();
const labelUList& neighbour = mesh_.neighbour();

const volVectorField& C = mesh.C();

// Set up temporary storage for the dd tensor (before inversion)
symmTensorField dd(mesh_.nCells(), Zero);

forAll(owner, facei)
{
    label own = owner[facei];
    label nei = neighbour[facei];

    symmTensor wdd = sqr(C[nei] - C[own]);
    dd[own] += wdd;
    dd[nei] += wdd;
}

surfaceVectorField::Boundary& blsP = pVectors_.boundaryField();

forAll(blsP, patchi)
{
    const fvsPatchVectorField& patchLsP = blsP[patchi];

    const fvPatch& p = patchLsP.patch();
    const labelUList& faceCells = p.patch().faceCells();
```

```
    // Build the d-vectors
    vectorField pd(p.delta());

    forAll(pd, patchFacei)
    {
        dd[faceCells[patchFacei]] += sqr(pd[patchFacei]);
    }
}

// Invert the dd tensor
const symmTensorField invDd(inv(dd));

// Revisit all faces and calculate the pVectors_ and nVectors_ vectors
forAll(owner, facei)
{
    label own = owner[facei];
    label nei = neighbour[facei];

    vector d = C[nei] - C[own];

    pVectors_[facei] = (invDd[own] & d);
    nVectors_[facei] = -(invDd[nei] & d);
}

forAll(blsP, patchi)
{
    fvsPatchVectorField& patchLsP = blsP[patchi];

    const fvPatch& p = patchLsP.patch();
    const labelUList& faceCells = p.faceCells();

    // Build the d-vectors
    vectorField pd(p.delta());

    forAll(pd, patchFacei)
    {
        patchLsP[patchFacei] = (invDd[faceCells[patchFacei]] & pd[patchFacei]);
    }
}
```

但是上述最小二乘法对于具有比较大长宽比的网格会存在问题，因为它会因为长网格距离向量对应的网格贡献更大。为了解决这个问题，一般使用权重最小二乘法，即对于每一个距离向量都让其乘以一个权重，该权重为其绝对值的倒数，即 $\omega = 1/|\vec{d}|$ ，于是写成矩阵形式就有

$$\boldsymbol{\omega}\mathbf{d}_{PN}(\nabla T)_P = \boldsymbol{\omega}(\mathbf{T}_N - \mathbf{T}_P)$$

这里的 $\boldsymbol{\omega}$ 是一个 $m \times m$ 的对角矩阵，对角元上的值就是各自对应的 $1/|\vec{d_i}|$ 。因此相应的最小二乘解变成了

$$(\nabla T)_P = (\mathbf{d}_{PN}^T\boldsymbol{\omega}^T\boldsymbol{\omega}\mathbf{d}_{PN})^{-1}\mathbf{d}_{PN}^T\boldsymbol{\omega}^T\boldsymbol{\omega}(\mathbf{T}_N - \mathbf{T}_P)$$

上述加权最小二乘法称为 invDistLeastSquares，从 invDistLeastSquaresVectors.C 文件中可以看到计算矩阵 $\mathbf{G} = (\mathbf{d}_{PN}^T \boldsymbol{\omega}^T \boldsymbol{\omega} \mathbf{d}_{PN})^{-1} \mathbf{d}_{PN}^T \boldsymbol{\omega}^T \boldsymbol{\omega}$ 的方法如下：

```
//- src/finiteVolume/finiteVolume/gradSchemes/leastSquaresGrad/invDistLeastSquaresVectors.C
const fvMesh& mesh = mesh_;

// Set local references to mesh data
const labelUList& owner = mesh_.owner();
const labelUList& neighbour = mesh_.neighbour();

const volVectorField& C = mesh.C();

// Set up temporary storage for the dd tensor (before inversion)
symmTensorField dd(mesh_.nCells(), Zero);

forAll(owner, facei)
{
    label own = owner[facei];
    label nei = neighbour[facei];

    vector d = C[nei] - C[own];
    symmTensor wdd = sqr(d)/magSqr(d);
    dd[own] += wdd;
    dd[nei] += wdd;
}

surfaceVectorField::Boundary& blsP = pVectors_.boundaryField();

forAll(blsP, patchi)
{
    const fvsPatchVectorField& patchLsP = blsP[patchi];

    const fvPatch& p = patchLsP.patch();
    const labelUList& faceCells = p.patch().faceCells();

    // Build the d-vectors
    vectorField pd(p.delta());

    forAll(pd, patchFacei)
    {
        const vector& d = pd[patchFacei];

        dd[faceCells[patchFacei]] += sqr(d)/magSqr(d);
    }
}

// Invert the dd tensor
const symmTensorField invDd(inv(dd));
```

```
// Revisit all faces and calculate the pVectors_ and nVectors_ vectors
forAll(owner, facei)
{
    label own = owner[facei];
    label nei = neighbour[facei];

    vector d = C[nei] - C[own];

    pVectors_[facei] = (invDd[own] & d)/magSqr(d);
    nVectors_[facei] = -(invDd[nei] & d)/magSqr(d);
}

forAll(blsP, patchi)
{
    fvsPatchVectorField& patchLsP = blsP[patchi];

    const fvPatch& p = patchLsP.patch();
    const labelUList& faceCells = p.faceCells();

    // Build the d-vectors
    vectorField pd(p.delta());

    forAll(pd, patchFacei)
    {
        const vector& d = pd[patchFacei];

        patchLsP[patchFacei] = (invDd[faceCells[patchFacei]] & d)/magSqr(d);
    }
}
```

不过，OpenFOAM 默认采用的最小二乘法在权重上不仅考虑了距离因素，还考虑了网格面积的因素，从 leastSquaresVectors.C 文件中可以看到其计算矩阵 $\mathbf{G}$ 的方法：

```
//- src/finiteVolume/finiteVolume/gradSchemes/leastSquaresGrad/leastSquaresVectors.C
const fvMesh& mesh = mesh_;

// Set local references to mesh data
const labelUList& owner = mesh_.owner();
const labelUList& neighbour = mesh_.neighbour();

const volVectorField& C = mesh.C();
const surfaceScalarField& w = mesh.weights();
const surfaceScalarField& magSf = mesh.magSf();

// Set up temporary storage for the dd tensor (before inversion)
symmTensorField dd(mesh_.nCells(), Zero);

forAll(owner, facei)
{
    label own = owner[facei];
```

```
    label nei = neighbour[facei];

    vector d = C[nei] - C[own];
    symmTensor wdd = (magSf[facei]/magSqr(d))*sqr(d);

    dd[own] += (1 - w[facei])*wdd;
    dd[nei] += w[facei]*wdd;
}

surfaceVectorField::Boundary& pVectorsBf = pVectors_.boundaryFieldRef();

forAll(pVectorsBf, patchi)
{
    const fvsPatchScalarField& pw = w.boundaryField()[patchi];
    const fvsPatchScalarField& pMagSf = magSf.boundaryField()[patchi];

    const fvPatch& p = pw.patch();
    const labelUList& faceCells = p.patch().faceCells();

    // Build the d-vectors
    vectorField pd(p.delta());

    if (pw.coupled())
    {
        forAll(pd, patchFacei)
        {
            const vector& d = pd[patchFacei];

            dd[faceCells[patchFacei]] += ((1 - pw[patchFacei])*pMagSf[patchFacei]/magSqr(d))*sqr(d);
        }
    }
    else
    {
        forAll(pd, patchFacei)
        {
            const vector& d = pd[patchFacei];

            dd[faceCells[patchFacei]] += (pMagSf[patchFacei]/magSqr(d))*sqr(d);
        }
    }
}

// Invert the dd tensor
const symmTensorField invDd(inv(dd));

// Revisit all faces and calculate the pVectors_ and nVectors_ vectors
forAll(owner, facei)
{
    label own = owner[facei];
```

```
    label nei = neighbour[facei];

    vector d = C[nei] - C[own];
    scalar magSfByMagSqrd = magSf[facei]/magSqr(d);

    pVectors_[facei] = (1 - w[facei])*magSfByMagSqrd*(invDd[own] & d);
    nVectors_[facei] = -w[facei]*magSfByMagSqrd*(invDd[nei] & d);
}

forAll(pVectorsBf, patchi)
{
    fvsPatchVectorField& patchLsP = pVectorsBf[patchi];

    const fvsPatchScalarField& pw = w.boundaryField()[patchi];
    const fvsPatchScalarField& pMagSf = magSf.boundaryField()[patchi];

    const fvPatch& p = pw.patch();
    const labelUList& faceCells = p.faceCells();

    // Build the d-vectors
    vectorField pd(p.delta());

    if (pw.coupled())
    {
        forAll(pd, patchFacei)
        {
            const vector& d = pd[patchFacei];

            patchLsP[patchFacei] =
                ((1 - pw[patchFacei])*pMagSf[patchFacei]/magSqr(d))
               *(invDd[faceCells[patchFacei]] & d);
        }
    }
    else
    {
        forAll(pd, patchFacei)
        {
            const vector& d = pd[patchFacei];

            patchLsP[patchFacei] =
                pMagSf[patchFacei]*(1.0/magSqr(d))
               *(invDd[faceCells[patchFacei]] & d);
        }
    }
}
```

在得到了上述系数矩阵 **G** 之后，通过遍历累加的方式就可以计算出相应的梯度，从 leastSquaresGrad.C 文件中可以看到实现方式如下：

```
//- src/finiteVolume/finiteVolume/gradSchemes/leastSquaresGrad/leastSquaresGrad.C
```

```
template<class Type>
Foam::tmp<Foam::GeometricField<typename Foam::outerProduct<Foam::vector, Type>::type,Foam::
    fvPatchField,Foam::volMesh>>
Foam::fv::leastSquaresGrad<Type>::calcGrad
(
    const GeometricField<Type, fvPatchField, volMesh>& vsf,
    const word& name
) const
{
    typedef typename outerProduct<vector, Type>::type GradType;

    const fvMesh& mesh = vsf.mesh();

    tmp<GeometricField<GradType, fvPatchField, volMesh>> tlsGrad
    (
        GeometricField<GradType, fvPatchField, volMesh>::New
        (
            name,
            mesh,
            dimensioned<GradType>
            (
                "zero",
                vsf.dimensions()/dimLength,
                Zero
            ),
            extrapolatedCalculatedFvPatchField<GradType>::typeName
        )
    );
    GeometricField<GradType, fvPatchField, volMesh>& lsGrad = tlsGrad.ref();

    // Get reference to least square vectors
    const leastSquaresVectors& lsv = leastSquaresVectors::New(mesh);

    const surfaceVectorField& ownLs = lsv.pVectors();
    const surfaceVectorField& neiLs = lsv.nVectors();

    const labelUList& own = mesh.owner();
    const labelUList& nei = mesh.neighbour();

    forAll(own, facei)
    {
        label ownFacei = own[facei];
        label neiFacei = nei[facei];

        Type deltaVsf = vsf[neiFacei] - vsf[ownFacei];

        lsGrad[ownFacei] += ownLs[facei]*deltaVsf;
        lsGrad[neiFacei] -= neiLs[facei]*deltaVsf;
    }
```

```
    // Boundary faces
    forAll(vsf.boundaryField(), patchi)
    {
        const fvsPatchVectorField& patchOwnLs = ownLs.boundaryField()[patchi];

        const labelUList& faceCells = vsf.boundaryField()[patchi].patch().faceCells();

        if (vsf.boundaryField()[patchi].coupled())
        {
            const Field<Type> neiVsf(vsf.boundaryField()[patchi].patchNeighbourField());

            forAll(neiVsf, patchFacei)
            {
                lsGrad[faceCells[patchFacei]] +=
                    patchOwnLs[patchFacei]
                   *(neiVsf[patchFacei] - vsf[faceCells[patchFacei]]);
            }
        }
        else
        {
            const fvPatchField<Type>& patchVsf = vsf.boundaryField()[patchi];

            forAll(patchVsf, patchFacei)
            {
                lsGrad[faceCells[patchFacei]] +=
                    patchOwnLs[patchFacei]
                   *(patchVsf[patchFacei] - vsf[faceCells[patchFacei]]);
            }
        }
    }

    lsGrad.correctBoundaryConditions();
    gaussGrad<Type>::correctBoundaryConditions(vsf, lsGrad);

    return tlsGrad;
}
```

第 6 章 fvSolution 设置

该文件由 solvers、所采用耦合算法 (PISO、SIMPLE、PIMPLE) 以及 relaxationFactors 等子字典组成。

6.1 solvers 设置

对于所选的不同类型的 solver，都需要设置如下关键词：

- **tolerance**：绝对残差 (前后两次迭代结果的差值)。决定了求解器退出的标准。对于稳态问题应该取得很小，而对于瞬态问题则不能取得太小。
- **relTol**：相对残差 (前后两次迭代残差的比值)。当设置为非零数时将忽略 tolerance 的设置。
- **maxIter**：最大迭代次数。

6.1.1 GAMG

几何-代数多重网格求解器 (Geometric agglomerated algebraic multigrid solver)，它的特点如下：

- Requires positive definite, diagonally dominant matrix.
- Agglomeration algorithm: selectable and optionally cached.
- Restriction operator: summation.
- Prolongation operator: injection.
- Smoother: Gauss-Seidel.
- Coarse matrix creation: central coefficient: summation of fine grid central coefficients with the removal of intra-cluster face; off-diagonal coefficient: summation of off-diagonal faces.
- Coarse matrix scaling: performed by correction scaling, using steepest descent optimisation.
- Type of cycle: V-cycle with optional pre-smoothing.
- Coarsest-level matrix solved using PCG or PBiCGStab.

设置方式如下所示：

```
// Mandatory entries
solver               GAMG;
smoother             <smoother>;
relTol               <relative tolerance>;
tolerance            <absolute tolerance>;


// Optional entries and associated default values

// Agglomeration
cacheAgglomeration   yes;
nCellsInCoarsestLevel 10;
processorAgglomerator <processor agglomeration method>;

// Solver
nPreSweeps           0;
preSweepsLevelMultiplier 1;
maxPreSweeps         4;
nPostSweeps          2;
postSweepsLevelMultiplier 1;
```

```
maxPostSweeps        4;
nFinestSweeps        2;
interpolateCorrection no;
scaleCorrection      yes; // Yes: symmetric No: Asymmetric
directSolveCoarsest  no;
```

6.1.2 PCG

Preconditioned conjugate gradient solver for symmetric lduMatrices using a run-time selectable preconditioner.

```
solver        PCG;
preconditioner <conditioner>;
relTol        <relative tolerance>;
tolerance     <tolerance>;
```

6.2 耦合算法设置

```
//PISO算法设置
PISO
{
    nCorrectors   2;                //PISO修正次数
    nNonOrthogonalCorrectors 0;     //非正交修正次数
    pRefCell      0;                //压力参考单元
    pRefValue     0;                //压力参考值
}
```

6.3 欠松弛处理设置

欠松弛处理主要有两种不同的使用方式：（1）在迭代之前提高系数矩阵主对角元素数值，并且降低源项来保证系数矩阵对角占优，比如速度场 U；（2）限制变量在迭代后改变的大小，比如压力场 p。欠松弛因子 α 介于 0 和 1 之间，越接近 0 求解越稳定，但同时也会降低求解效率；提高 α 的数值会提高求解效率，但同时也会降低稳定性。

欠松弛因子在子字典 relaxationFactors 下指定，通过子字典名 equations 和 fields 来设置不同的欠松弛方式，例如：

```
relaxationFactors
{
    fields
    {
        p 0.3;
    }
    equations
    {
        U 0.7;
    }
}
```

第 7 章 运算符与运算函数

在 OpenFOAM/primitives/Tensor/TensorI.H 定义了张量运算符与一些常用的张量运算函数。

```
//- Return the trace of a tensor
template<class Cmpt>
inline Cmpt tr(const Tensor<Cmpt>& t)
{
    return t.xx() + t.yy() + t.zz();
}


//- Return the spherical part of a tensor
template<class Cmpt>
inline SphericalTensor<Cmpt> sph(const Tensor<Cmpt>& t)
{
    return (1.0/3.0)*tr(t);
}


//- Return the symmetric part of a tensor
template<class Cmpt>
inline SymmTensor<Cmpt> symm(const Tensor<Cmpt>& t)
{
    return SymmTensor<Cmpt>
    (
        t.xx(), 0.5*(t.xy() + t.yx()), 0.5*(t.xz() + t.zx()),
                t.yy(),                0.5*(t.yz() + t.zy()),
                                       t.zz()
    );
}


//- Return twice the symmetric part of a tensor
template<class Cmpt>
inline SymmTensor<Cmpt> twoSymm(const Tensor<Cmpt>& t)
{
    return SymmTensor<Cmpt>
    (
        2*t.xx(), (t.xy() + t.yx()), (t.xz() + t.zx()),
                  2*t.yy(),          (t.yz() + t.zy()),
                                     2*t.zz()
    );
}


//- Return the skew-symmetric part of a tensor
template<class Cmpt>
```

```
inline Tensor<Cmpt> skew(const Tensor<Cmpt>& t)
{
    return Tensor<Cmpt>
    (
        0.0, 0.5*(t.xy() - t.yx()), 0.5*(t.xz() - t.zx()),
        0.5*(t.yx() - t.xy()), 0.0, 0.5*(t.yz() - t.zy()),
        0.5*(t.zx() - t.xz()), 0.5*(t.zy() - t.yz()), 0.0
    );
}


//- Return the skew-symmetric part of a symmetric tensor
template<class Cmpt>
inline const Tensor<Cmpt>& skew(const SymmTensor<Cmpt>& st)
{
    return Tensor<Cmpt>::zero;
}


//- Return the deviatoric part of a tensor
template<class Cmpt>
inline Tensor<Cmpt> dev(const Tensor<Cmpt>& t)
{
    return t - SphericalTensor<Cmpt>::oneThirdI*tr(t);
}


//- Return the deviatoric part of a tensor
template<class Cmpt>
inline Tensor<Cmpt> dev2(const Tensor<Cmpt>& t)
{
    return t - SphericalTensor<Cmpt>::twoThirdsI*tr(t);
}


//- Return the determinant of a tensor
template<class Cmpt>
inline Cmpt det(const Tensor<Cmpt>& t)
{
    return
    (
        t.xx()*t.yy()*t.zz() + t.xy()*t.yz()*t.zx()
      + t.xz()*t.yx()*t.zy() - t.xx()*t.yz()*t.zy()
      - t.xy()*t.yx()*t.zz() - t.xz()*t.yy()*t.zx()
    );
}


//- Return the cofactor tensor of a tensor
```

```
template<class Cmpt>
inline Tensor<Cmpt> cof(const Tensor<Cmpt>& t)
{
    return Tensor<Cmpt>
    (
        t.yy()*t.zz() - t.zy()*t.yz(),
        t.zx()*t.yz() - t.yx()*t.zz(),
        t.yx()*t.zy() - t.yy()*t.zx(),

        t.xz()*t.zy() - t.xy()*t.zz(),
        t.xx()*t.zz() - t.xz()*t.zx(),
        t.xy()*t.zx() - t.xx()*t.zy(),

        t.xy()*t.yz() - t.xz()*t.yy(),
        t.yx()*t.xz() - t.xx()*t.yz(),
        t.xx()*t.yy() - t.yx()*t.xy()
    );
}


//- Return the inverse of a tensor given the determinant
template<class Cmpt>
inline Tensor<Cmpt> inv(const Tensor<Cmpt>& t, const Cmpt dett)
{
    return Tensor<Cmpt>
    (
        t.yy()*t.zz() - t.zy()*t.yz(),
        t.xz()*t.zy() - t.xy()*t.zz(),
        t.xy()*t.yz() - t.xz()*t.yy(),

        t.zx()*t.yz() - t.yx()*t.zz(),
        t.xx()*t.zz() - t.xz()*t.zx(),
        t.yx()*t.xz() - t.xx()*t.yz(),

        t.yx()*t.zy() - t.yy()*t.zx(),
        t.xy()*t.zx() - t.xx()*t.zy(),
        t.xx()*t.yy() - t.yx()*t.xy()
    )/dett;
}


//- Return the inverse of a tensor
template<class Cmpt>
inline Tensor<Cmpt> inv(const Tensor<Cmpt>& t)
{
    return inv(t, det(t));
}
```

```
template<class Cmpt>
inline Tensor<Cmpt> Tensor<Cmpt>::inv() const
{
    return Foam::inv(*this);
}


//- Return the 1st invariant of a tensor
template<class Cmpt>
inline Cmpt invariantI(const Tensor<Cmpt>& t)
{
    return tr(t);
}


//- Return the 2nd invariant of a tensor
template<class Cmpt>
inline Cmpt invariantII(const Tensor<Cmpt>& t)
{
    return
    (
        t.xx()*t.yy() + t.yy()*t.zz() + t.xx()*t.zz()
      - t.xy()*t.yx() - t.yz()*t.zy() - t.xz()*t.zx()
    );
}


//- Return the 3rd invariant of a tensor
template<class Cmpt>
inline Cmpt invariantIII(const Tensor<Cmpt>& t)
{
    return det(t);
}
```

第 8 章 并行计算

参考https://doc.cfd.direct/openfoam/user-guide-v10/running-applications-parallel。OpenFOAM 采取的并行计算策略是算域分解，也就是将计算域和相关的场分解为多块，分别提交给不同的处理器进行计算，整个并行计算的流程包括：网格和场的分解、并行运行程序、后处理分块算例。默认使用 openMPI 的 MPI 模块。

8.1 分块与运行方法

网格和场通过 decomposePar 程序进行分块，需要通过 system/decomposeParDict 文件来设置相关参数。该文件主要需要设置的关键词如下：

关键词	含义
numberOfSubdomains	划分出分块子域的总数，一般不超过设备的物理核数
method	分块的方法，可选的有 simple、hierarchical、scotch、manual
n	使用 simple 和 hierarchical 方法时需要指定的参数，表明 x,y,z 方向上的子域数量 (n_x, n_y, n_z)
order	使用 hierarchical 方法时需要指定的参数，表明划分的顺序，如 xyz/xzy/yxz 等
processorWeights	使用 scotch 方法时需要指定的参数，表明处理器被分配的网格量的权重参数列表，各权重最终会被归一化
dataFile	使用 manual 方法时需要指定的参数，指定设置了各处理器被分配网格信息的文件
distributed	可设 yes/no，表明计算数据结果是否需要跨盘分布
roots	表明各算例文件夹的路径列表，每个路径依次对应一个节点

四种划分方法的含义如下：

- simple：将算域按照 xyz 的顺序依次进行划分，例如 x 方向划分 2 块，y 方向划分 2 块，z 方向划分 1 块，则设置 n(2 2 1)；
- hierarchical：该方法实际上与 simple 相同，只是可以通过 order 参数指定划分的顺序；
- scotch：该方法将尝试最小化各处理器之间的交互界面网格数量，可以通过 processorWeights 指定不同的权重来适应设备不同处理器性能不同的情况。还有一个额外的关键词 strategy 来控制更复杂的参数。
- manual：手动给各个处理器划分网格。

不同方法通过子字典 <method>Coeffs 来指定其所需的参数。下面是一个经典的 decomposeParDict 文件内容：

```
numberOfSubdomains 4;

method  simple
simpleCoeffs
{
  n   (2 2 1);
}

distributed  no;
roots   ();
```

在设置好 decomposeParDict 文件后，直接在算例文件夹处执行下面的命令即可进行划分：

```
decomposePar
```

划分完毕后，通过如下方式进行并行计算：[1]

[1]其中的-parallel 设置非常重要，否则会让不同处理器按次序单核计算同样的东西。

```
mpirun -np <numberOfSubdomains> <application> <otherArgs> -parallel
```

如果是在集群中并行，则可以新建一个 machines 文件，里面指明各设备名称与处理器数量，例如

```
aaa
bbb cpu=2
ccc
```

然后通过如下方式在集群中进行并行计算：

```
mpirun --hostfile <machines> -np <numberOfSubdomains> <application> <otherArgs> -parallel
```

8.2 并行编程

8.2.1 不同处理器输出信息

如果要让不同的处理器输出各自的信息，可以如下书写：

```
Pout << "Hello␣from␣process␣" << Pstream::myProcNo() << endl;
```

其中 Pout 是不同处理器均输出信息的意思。而如果使用 Info，则只有 processor0 即主处理器会输出。

第 9 章 RunTimeSelection 机制

参考http://xiaopingqiu.github.io/2016/03/12/RTS1/。OpenFOAM 中包含各个 CFD 相关的模块，每个模块，从 C++ 的角度来看，其实都是一个类的框架。基类用作接口，一个派生类则是一个具体的模型。OpenFOAM 中的模块广泛使用 RTS 机制，因此 OpenFOAM 的求解器中，只需要设定模型的调用接口。算例具体使用的是哪个模型，则是在运行时才确定的，而且可以在算例运行过程中修改选中的模型。

RTS 机制的实现跟几个函数的调用有关：declareRunTimeSelectionTable、defineRunTimeSelectionTable、defineTypeNameAndDebug、addToRunTimeSelectionTable。规律可以总结如下：

1. 基类类体里调用 TypeName 和 declareRunTimeSelectionTable 两个函数，类体外面调用 defineTypeNameAndDebug、defineRunTimeSelectionTable 和 addToRunTimeSelectionTable 三个函数；
2. 基类中需要一个静态 New 函数作为 selector；
3. 派生类类体中需要调用 TypeName 函数，类体外调用 defineRunTimeSelectionTable 和 addToRunTimeSelectionTable 两个宏函数。

以上函数都是定义在 runTimeSelectionTables.H 和 addToRunTimeSelectionTable.H 两个头文件中，而且这些函数都是宏函数。

RTS 机制的本质可以总结如下：

1. 基类里定义一个 hashTable，其 key 为类的 typeName，value 为一个函数指针，这个函数指针指向的函数的返回值是基类类型的 autoPtr，并且这个 autoPtr 指向类的一个临时对象（用 C++ 的 new 关键字创建）。这些在宏函数 declareRunTimeSelectionTable 中完成。
2. 每创建一个派生类，都会调用一次 addToRunTimeSelectionTable 宏函数。这个宏函数会触发一次 hashTable 的更新操作。具体地说，宏函数的调用，会往基类里定义的 hashTable 插入一组值，这组值的 key 是该派生类的 typeName，value 是一个函数，该函数返回的是指向派生类临时对象的指针。
3. 类及其派生类编译成库，在编译过程中，会逐步往 hashTable 增加新元素，直到可选的模型全部添加到其中。
4. 在需要调用这些类的地方，只需要定义基类的 autoPtr，并用基类中定义的 New 函数来初始化，即 autoPtr<AlgorithmBase> algorithmPtr = AlgorithmBase::New(algorithmName);。这样，New 函数就能根据调用的时候所提供的参数（即 hashTable 的 key），来从 hashTable 中选择对应的派生类（即 hashTable 的 value）。

9.1 nccr 库的 RTS 实现

在 src/nccr/basicNCCR/basicNCCR.H 文件中有如下的代码声明 RTS 功能 (无关代码被省略)：

```
#include "runTimeSelectionTables.H"

class basicNCCR
{
public:
    // 作为declareRunTimeSelectionTable宏函数的baseType参数
    TypeName("basicNCCR");

    // 声明用于RTS的基类hashTable
    declareRunTimeSelectionTable
    (
        autoPtr,
        basicNCCR,
        state,
```

```
        (
            const volScalarField& rho,
            const volScalarField& p,
            const volVectorField& U,
            const volScalarField& T,
            const volScalarField& nuEff,
            const volScalarField& kappaEff,
            fluidThermo& thermo
        ),
        (rho, p, U, T, nuEff, kappaEff, thermo)
    );

    // 充当selector的基类New函数，也就是工厂函数
    static autoPtr<basicNCCR> New
    (
        const volScalarField& rho,
        const volScalarField& p,
        const volVectorField& U,
        const volScalarField& T,
        const volScalarField& nuEff,
        const volScalarField& kappaEff,
        fluidThermo& thermo
    );
};
```

在 src/nccr/basicNCCR/basicNCCR.C 文件中调用调用如下代码来创建基类的 hashTable：

```
namespace Foam
{
    defineTypeNameAndDebug(basicNCCR, 0);
    defineRunTimeSelectionTable(basicNCCR, state);
}
```

基类的工厂函数在 src/nccr/basicNCCR/newBasicNCCR.C 中的实现如下所示：

```
Foam::autoPtr<Foam::basicNCCR> Foam::basicNCCR::New(
    const volScalarField& rho,
    const volScalarField& p,
    const volVectorField& U,
    const volScalarField& T,
    const volScalarField& nuEff,
    const volScalarField& kappaEff,
    fluidThermo& thermo
)
{
    const dictionary& subDict = U.mesh().schemes().subDict("nccr");
    word name = word(subDict.lookup("method")) + "Method";

    Info << "Selecting␣NCCR␣algorithm␣" << name << endl;

    stateConstructorTable::iterator cstrIter = stateConstructorTablePtr_->find(name);
```

```
    if (cstrIter == stateConstructorTablePtr_->end())
    {
        FatalErrorIn("Foam::basicNCCR::New(const␣fvMesh&)")
            << "Unkown␣basicNCCR␣type" << name << nl << nl
            << "Valid␣basicNCCR␣types␣are:" << nl
            << stateConstructorTablePtr_->sortedToc() << nl
            << exit(FatalError);
    }

    return autoPtr<basicNCCR>(cstrIter()(rho, p, U, T, nuEff, kappaEff, thermo));
}
```

对于派生类，需要定义 TypeName 并处理基类的虚函数，同时为了后续添加 hashTable 元素方便，可以将主文件放入 H 文件中：

```
namespace Foam
{

template<class Method>
class nccrUpdate
:
    public basicNCCR
{
public:
    TypeName("nccrUpdate");

    nccrUpdate(
        const volScalarField& rho,
        const volScalarField& p,
        const volVectorField& U,
        const volScalarField& T,
        const volScalarField& nuEff,
        const volScalarField& kappaEff,
        fluidThermo& thermo
    );
};

} // end of namespace

#ifdef NoRepository
#include "nccrUpdate.C"
#endif
```

对于派生类是通过 template 实现的，后续在 src/nccr/nccrUpdate/nccrUpdates.C 中添加 hashTable 元素的时候要借助 typedef 并显式指定名称：

```
#include "nccrUpdate.H"
#include "addToRunTimeSelectionTable.H"

#include "dafpMethod.H"
```

```
namespace Foam
{
    typedef nccrUpdate<dafpMethod> DAFP;

    defineTemplateTypeNameAndDebugWithName(
        DAFP,
        "dafpMethod",
        0
    );

    addToRunTimeSelectionTable(
        basicNCCR,
        DAFP,
        state
    );
}
```

9.2 dbns 库的 RTS 实现

在 src/dbns/basicNumericFlux/basicNumericFlux.H 文件中基类有如下代码声明 RTS 功能 (无关代码被省略):

```
#include "runTimeSelectionTables.H"

class basicNumericFlux
{
public:
    // 作为declareRunTimeSelectionTable宏函数的baseType参数
    TypeName("basicNumericFlux");

    // 声明用于RTS的基类hashTable
  declareRunTimeSelectionTable
  (
      autoPtr,
      basicNumericFlux,
      state,
      (
          const volScalarField& p,
          const volVectorField& U,
          const volScalarField& T,
          basicThermo& thermo
      ),
      (p, U, T, thermo)
  );

    // 充当selector的基类New函数, 也就是工厂函数
    static autoPtr<basicNumericFlux> New
    (
```

```
        const volScalarField& p,
        const volVectorField& U,
        const volScalarField& T,
        basicThermo& thermo
    );
};
```

其中宏函数 declareRunTimeSelectionTable 展开的结果为

```
//- 第一个typedef定义的是一个函数指针，这样定义的结果是，
// stateConstructorPtr代表一个指向参数为(p,U,T,thermo)
// 返回类型为autoPtr<basicNumericFlux>的函数指针。
typedef autoPtr<basicNumericFlux> (*stateConstructorPtr)
(
    const volScalarField& p,
    const volVectorField& U,
    const volScalarField& T,
    basicThermo& thermo
);

//- 第二个typedef将一个key和value分别为word和stateConstructorPtr函数指针
// 的hashTable定义了一个别名叫stateConstructorTable。
typedef HashTable<stateConstructorPtr, word, string::hash>
    stateConstructorTable;

//- 静态数据成员stateConstructorTablePtr_是一个stateConstructorTable类型的指针，
// 也就是指向存储了word和stateConstructorPtr函数指针的hashTable的指针
static stateConstructorTable* stateConstructorTablePtr_;

//- 两个静态成员函数，这里只是声明了，后面将通过defineRunTimeSelectionTable宏函数进行定义
// 并且注意到在下面定义的两个类中用到了这两个函数。
static void constructstateConstructorTables();
static void destroystateConstructorTables();

template<class basicNumericFluxType>
class addstateConstructorToTable
{
public:
    //- 这个New函数，返回的是一个basicNumericFlux类型的临时对象的指针，
    // 后续将在派生类中通过模板传入具体定义的basicNumericFluxType。
    // 之后在addToRunTimeSelectionTable中创建一个addstateConstructorToTable类的对象，
    // 目的是为了调用派生类的构造函数。
    static autoPtr<basicNumericFlux> New
    (
        const volScalarField& p,
        const volVectorField& U,
        const volScalarField& T,
        basicThermo& thermo
    )
    {
```

```
        return autoPtr<basicNumericFlux>(new basicNumericFluxType (p, U, T, thermo));
    }

    addstateConstructorToTable
    (
        const word& lookup = basicNumericFluxType::typeName
    )
    {
        //- 这个类的构造函数中，首先调用了constructstateConstructorTables函数，
        //  该函数通过defineRunTimeSelectionTable宏函数进行定义，
        //  实际上它是对指针stateConstructorTablePtr_进行了初始化，
        //  令其指向一个动态分配的stateConstructorTable
        constructstateConstructorTables();

        //- 然后，对stateConstructorTablePtr_进行insert操作，即往其指向的hashTable插入key-value对。
        //  这里的New函数指的就是上面的New函数。对于这里的情形，可以知道这个insert操作将创建一个
        //  “类的typeName—返回类的临时对象的引用的函数”映射对，并增加到stateConstructorTablePtr_中
        if (!stateConstructorTablePtr_->insert(lookup, New))
        {
            std::cerr<< "Duplicate␣entry␣" << lookup
                << "␣in␣runtime␣selection␣table␣" << basicNumericFlux
                << std::endl;
            error::safePrintStack(std::cerr);
        }
    }

    ~addstateConstructorToTable()
    {
        destroystateConstructorTables();
    }
};

template<class basicNumericFluxType>
class addRemovablestateConstructorToTable
{
    //...
};
```

在 src/dbns/basicNumericFlux/basicNumericFlux.C 中调用了

```
namespace Foam
{
    defineTypeNameAndDebug(basicNumericFlux, 0);
    defineRunTimeSelectionTable(basicNumericFlux, state);
}
```

其中 defineRunTimeSelectionTable 宏函数展开的结果如下，它的主要功能是对 declareRunTimeSelectionTable 中定义的静态数据成员和两个静态函数进行了定义：

```
//- 首先对静态数据成员stateConstructorTablePtr_初始化为nullptr
basicNumericFlux::stateConstructorTable*
```

```
    basicNumericFlux::stateConstructorTablePtr_ = nullptr

//- 然后constructstateConstructorTables函数将stateConstructorTablePtr_
//  指向一个动态分配的stateConstructorTable
void basicNumericFlux::constructstateConstructorTables()
{
    static bool constructed = false;
    if (!constructed)
    {
        constructed = true;
        basicNumericFlux::stateConstructorTablePtr_
            = new basicNumericFlux::stateConstructorTable;
    }
}

//- destroyWordConstructorTables则是对指针stateConstructorTablePtr_进行销毁
void basicNumericFlux::destroystateConstructorTables()
{
    if (basicNumericFlux::stateConstructorTablePtr_)
    {
        delete basicNumericFlux::stateConstructorTablePtr_;
        basicNumericFlux::stateConstructorTablePtr_ = nullptr;
    }
}
```

以上是基类相关的，现在再往下看派生类。前文已讲，派生类只需要在类体里调用 TypeName，然后在类体外调用 addToRunTimeSelectionTable。对于派生类，在 src/dbns/basicNumericFlux/makeBasicNumericFlux.H 中调用了

```
#define makeBasicNumericFlux(Flux,Limiter) \
                                           \
typedef numericFlux<Flux,Limiter>          \
    Flux##Limiter;                         \
                                           \
defineTemplateTypeNameAndDebugWithName     \
(                                          \
    Flux##Limiter,                         \
    #Flux#Limiter,                         \
    0                                      \
);                                         \
                                           \
addToRunTimeSelectionTable                 \
(                                          \
    basicNumericFlux,                      \
    Flux##Limiter,                         \
    state                                  \
)
```

而这里新定义的宏函数 makeBasicNumericFlux 则在 src/dbns/numericFlux/numericFluxes.C 中使用，目的是避免多个 Flux 与 Limiter 之间组合导致的代码冗杂：

```
namespace Foam
{

#define makeBasicNumericFluxForAllLimiters(Flux) \
makeBasicNumericFlux(Flux, firstOrderLimiter); \
makeBasicNumericFlux(Flux, BarthJespersenLimiter); \
makeBasicNumericFlux(Flux, VenkatakrishnanLimiter); \
makeBasicNumericFlux(Flux, WangLimiter);      \
makeBasicNumericFlux(Flux, MichalakGoochLimiter);

makeBasicNumericFluxForAllLimiters(rusanovFlux);
makeBasicNumericFluxForAllLimiters(betaFlux);
makeBasicNumericFluxForAllLimiters(roeFlux);
makeBasicNumericFluxForAllLimiters(hllcFlux);
makeBasicNumericFluxForAllLimiters(AUSMplusFlux);
makeBasicNumericFluxForAllLimiters(AUSMplusUpFlux);

}
```

下面就按照 Flux 取 hllcFlux、Limiter 取 firstOrderLimiter 的情况来说明派生类的 addToRunTimeSelectionTable 宏函数展开的结果：

```
typedef numericFlux<hllcFlux,firstOrderLimiter>
    hllcFluxfirstOrderLimiter;

//- 这里创建了一个addstateConstructorToTable类的对象，
//  代入的模板参数是hllcFluxfirstOrderLimiter，
//  于是决定了调用类的构造函数的时候代入的模板参数basicNumericFluxType，
//  所以这时New函数返回的将是hllcFluxfirstOrderLimiter类的临时对象的指针。
//  并且，hllcFluxfirstOrderLimiter这个名字与其对应的New函数组成的映射对
//  也被insert到stateConstructorTablePtr_里面。
basicNumericFlux::addstateConstructorToTable<hllcFluxfirstOrderLimiter>
    addhllcFluxfirstOrderLimiterstateConstructorTobasicNumericFluxTable_
```

最后，再来看一下 selector，即基类中定义的 New 函数。它定义在 src/dbns/basicNumericFlux/newBasicNumericFlux.C 中：

```
//- 这个函数的返回值类型为autoPtr<basicNumericFlux>，参数跟类的typeName一样，都是(p, U, T, thermo)
Foam::autoPtr<Foam::basicNumericFlux> Foam::basicNumericFlux::New
(
    const volScalarField& p,
    const volVectorField& U,
    const volScalarField& T,
    basicThermo& thermo
)
{
    const dictionary& subDict = p.mesh().schemes().subDict("divSchemes").subDict("dbns");
    word name = word(subDict.lookup("flux")) + "Flux" + word(subDict.lookup("limiter")) + "Limiter";
    Info<< "Selecting␣numericFlux␣" << name << endl;
```

```
    //- 首先定义了一个hashTable的迭代器cstrIter，利用迭代器来遍历搜索
    //  查看stateConstructorTable里面是否能找到与指定参数name相符的key值，
    //  比如取name为hllcFluxfirstOrderLimiter，如果找不到，那就报错退出并输出
    //  当前的stateConstructorTable中可选的项的名称，即stateConstructorTablePtr_->sortedToc()
    //  如果找到了，那就返回这个key对应的value，也即cstrIter返回该value指代的函数指针。
    stateConstructorTable::iterator cstrIter = stateConstructorTablePtr_->find(name);

    if (cstrIter == stateConstructorTablePtr_->end())
    {
        FatalErrorIn("basicNumericFlux::New(const␣fvMesh&)")
            << "Unknown␣basicNumericFlux␣type␣" << name << nl << nl
            << "Valid␣basicNumericFlux␣types␣are:" << nl
            << stateConstructorTablePtr_->sortedToc() << nl
            << exit(FatalError);
    }

    //- 而由于cstrIter是一个函数指针，所以cstrIter()返回的就是指定参数name所对应的那个New函数
    //  进一步看， cstrIter()(p, U, T, thermo)则表示的是函数调用了，传给函数的参数正是(p, U, T, thermo),
    //  所以cstrIter()(p, U, T, thermo)返回的是autoPtr<basicNumericFlux>,
    //  其指向的是typeName为hllcFluxfirstOrderLimiter的类的对象！这样就实现了New函数作为selector的功能！
    return autoPtr<basicNumericFlux>(cstrIter()(p, U, T, thermo));
}
```