cgns2openfoam 开发详解文档

目录

1. 项目背景与目标

1.1 项目起源

在计算流体力学 (CFD) 领域，不同的软件使用不同的网格格式。CGNS (CFD General Notation System) 是一种广泛使用的标准格式，被许多商业和开源前处理器支持（如 Pointwise、ICEM CFD、Gmsh 等）。OpenFOAM 作为主流的开源 CFD 求解器，使用其特有的 polyMesh 格式。本项目旨在搭建这两种格式之间的桥梁。

1.2 核心目标

1. **完整性**：支持 CGNS 中的点、单元、边界条件的完整读取
2. **多区域支持**：处理包含多个 Zone 的复杂网格
3. **滑移网格支持**：正确处理旋转机械中的 cyclicAMI 接口
4. **健壮性**：处理各种边界情况，如名称特殊字符、未使用点等

5. **兼容性**：生成的网格能被 OpenFOAM 求解器和 ParaView 正确读取

1.3 技术栈

- **编程语言**：C++17
- **构建系统**：CMake 3.14+
- **依赖库**：
  - CGNS 库 (libcgns)：用于读取 CGNS 文件
  - HDF5 库 (libhdf5)：CGNS 的底层存储格式
- **目标平台**：Linux (已在 Ubuntu 22.04 上测试)

2. CGNS 与 OpenFOAM 格式对比

2.1 CGNS 格式特点

CGNS 是一种层次化的数据格式，其结构如下：

```
CGNS File
└── Base (可以有多个)
    ├── cellDim: 单元维度 (3D 网格为 3)
    ├── physDim: 物理空间维度 (通常为 3)
    └── Zone (可以有多个)
        ├── ZoneType: Structured 或 Unstructured
        ├── GridCoordinates
        │   ├── CoordinateX
        │   ├── CoordinateY
        │   └── CoordinateZ
        ├── Elements (多个 Section)
        │   ├── Section 1: 体单元 (TETRA_4, HEXA_8, ...)
        │   ├── Section 2: 边界面 (TRI_3, QUAD_4, ...)
        │   └── ...
        ├── ZoneBC (边界条件)
        │   ├── BC 1: name, type, point range
        │   └── ...
        └── ZoneGridConnectivity (Zone 间连接)
            ├── 1to1 连接
            └── General 连接 (用于 AMI)
```

CGNS 的关键特征：

- 节点索引从 **1** 开始 (1-based indexing)
- 单元通过节点列表定义，不直接存储面
- 边界条件通过元素范围或点集定义
- 支持结构化和非结构化网格

2.2 OpenFOAM polyMesh 格式特点

OpenFOAM 使用基于面的网格表示 (face-based representation)：

```
polyMesh/
├── points      # 所有点的坐标
├── faces       # 所有面的节点列表
├── owner       # 每个面的 owner 单元
├── neighbour   # 内部面的 neighbour 单元
├── boundary    # 边界 patch 定义
├── cellZones   # 单元区域 (可选，动网格需要)
└── faceZones   # 面区域 (可选)
```

OpenFOAM 的关键特征：

- 节点索引从 **0** 开始 (0-based indexing)
- 面是一等公民，单元通过面隐式定义
- 面列表有严格顺序：**内部面在前，边界面在后**
- 每个面有 owner，内部面还有 neighbour
- 必须满足 `owner[i] < neighbour[i]` 对于所有内部面
- 同一 patch 内的面必须按 owner 排序

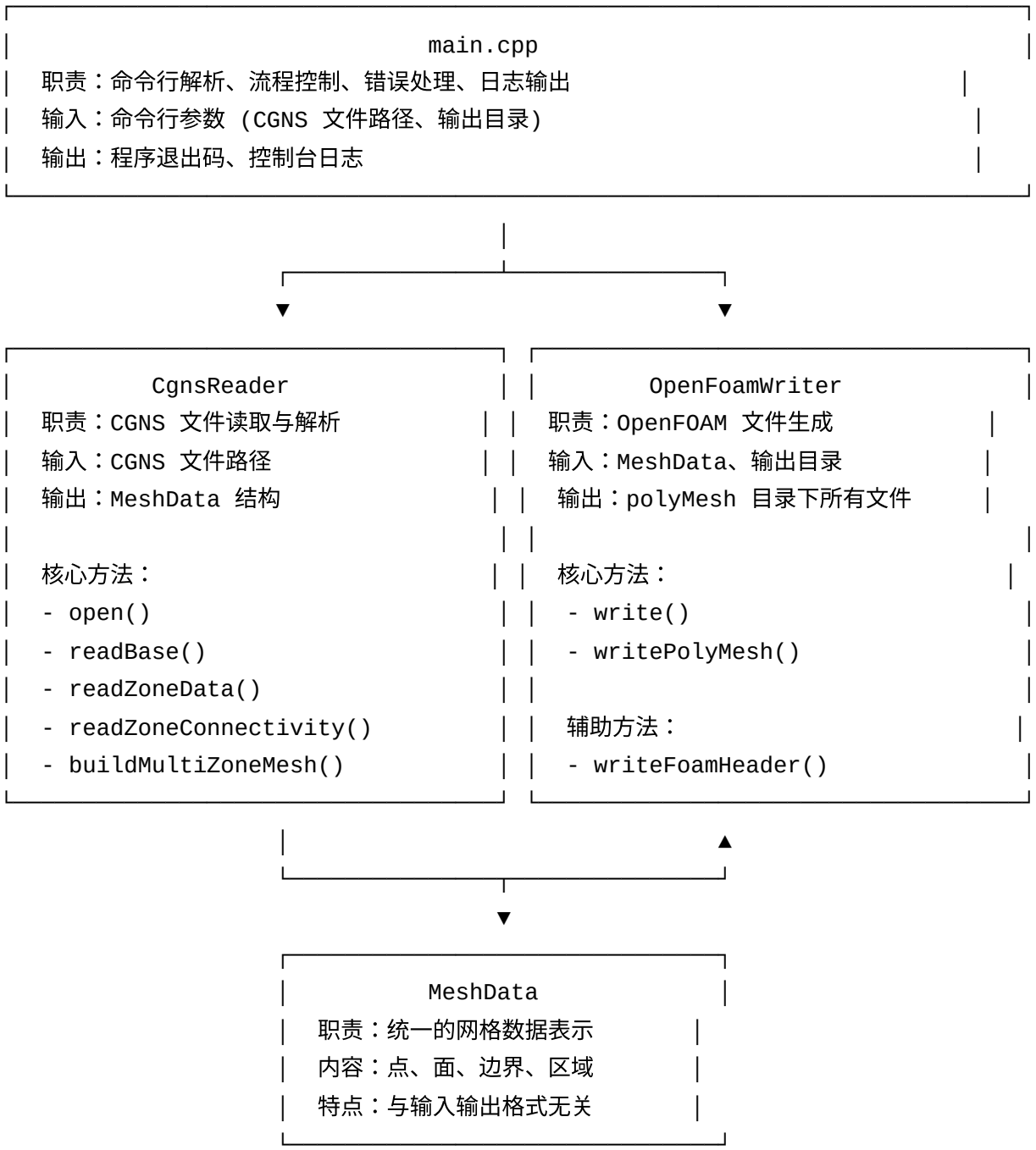
2.3 格式转换的核心挑战

挑战	描述	解决复杂度
索引转换	1-based → 0-based	简单
面提取	从单元节点推导面	中等
拓扑构建	确定 owner/neighbour	复杂
多区域合并	全局索引计算	复杂
接口处理	cyclicAMI 配对	复杂

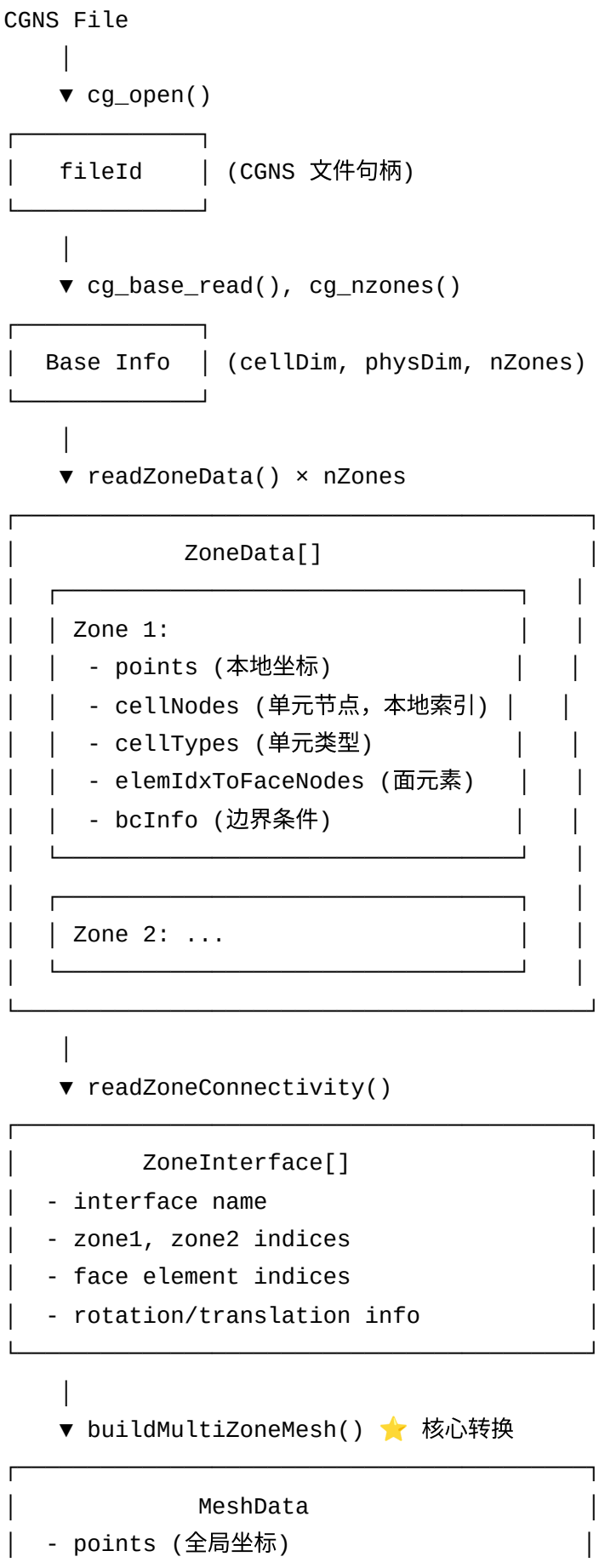
3. 程序架构设计

3.1 模块划分

程序采用三层架构设计，职责分明：



3.2 数据流图


CGNS File
cg_open()
fileId
(CGNS 文件句柄)
cg_base_read(), cg_nzones()
Base Info
(cellDim, physDim, nZones)
readZoneData() × nZones
ZoneData[]
Zone 1:
- points (本地坐标)
- cellNodes (单元节点，本地索引)
- cellTypes (单元类型)
- elemIdxToFaceNodes (面元素)
- bcInfo (边界条件)
Zone 2: ...
readZoneConnectivity()
ZoneInterface[]
- interface name
- zone1, zone2 indices
- face element indices
- rotation/translation info
buildMultiZoneMesh() ⭐ 核心转换
MeshData
- points (全局坐标)

```
  │  - faces (所有面，含 owner/neighbour)   │
  │  - nInternalFaces                       │
  │  - boundaries (边界 patch 列表)         │
  │  - cellZones (用于动网格)               │
  │  - nCells                               │
  └─────────────────────────────────────────┘

     │
     ▼ OpenFoamWriter::write()

  ┌─────────────────────────────────────────┐
  │           polyMesh/                     │
  │  - points, faces, owner, neighbour      │
  │  - boundary, cellZones, faceZones       │
  └─────────────────────────────────────────┘
```

4. 核心数据结构详解

4.1 Point3D - 三维点

```cpp
struct Point3D {
    double x, y, z;

    // 点相等比较，使用容差避免浮点误差
    bool operator==(const Point3D& other) const {
        const double tol = 1e-10;
        return std::abs(x - other.x) < tol &&
               std::abs(y - other.y) < tol &&
               std::abs(z - other.z) < tol;
    }
};
```

设计说明：使用容差比较是为了处理浮点精度问题。在点合并或去重时，两个坐标非常接近的点应被视为同一点。容差值 `1e-10` 适用于大多数工程尺度的网格。

4.2 Face - 面

```
struct Face {
    std::vector<int> nodes;  // 组成面的节点索引（有序）
    int owner = -1;          // 拥有此面的单元（索引较小者）
    int neighbour = -1;      // 共享此面的另一单元（边界面为 -1）

    // 返回排序后的节点列表，用于面的唯一标识
    std::vector<int> sortedNodes() const {
        std::vector<int> sorted = nodes;
        std::sort(sorted.begin(), sorted.end());
        return sorted;
    }
};
```

设计说明：

- `nodes` 保持原始顺序，决定面的法向量方向（按右手法则）
- `sortedNodes()` 用于生成 faceKey，识别两个单元共享的同一面
- `owner` 和 `neighbour` 的区分是 OpenFOAM 格式的核心要求

4.3 BoundaryPatch - 边界面片

```
struct BoundaryPatch {
    std::string name;       // 边界名称 (如 "inlet", "wall")
    std::string type;       // OpenFOAM 边界类型 (wall, patch, cyclicAMI, ...)
    int startFace;          // 在 faces 列表中的起始位置
    int nFaces;             // 包含的面数量

    // cyclicAMI 专用属性
    std::string neighbourPatch;             // 配对的 patch 名称
    std::array<double, 3> rotationAxis;     // 旋转轴
    std::array<double, 3> rotationCenter;   // 旋转中心
    double rotationAngle;                   // 旋转角度 (弧度)
    std::array<double, 3> separationVector; // 平移向量 (周期边界)
};
```

设计说明：

- `startFace` 和 `nFaces` 定义了该边界在全局面列表中的位置
- cyclicAMI 属性用于旋转机械中的滑移网格接口

4.4 ZoneData - CGNS Zone 数据

```
struct ZoneData {
    std::string name;           // Zone 名称
    int cgnsIndex;              // 在 CGNS 文件中的索引

    // 原始数据（本地索引）
    std::vector<Point3D> points;              // 点坐标
    std::vector<std::vector<int>> cellNodes;  // 单元的节点列表
    std::vector<int> cellTypes;               // 单元类型 (TETRA_4, HEXA_8, ...)

    // 面元素映射：elemIdx → 组成该面的节点列表
    std::map<int, std::vector<int>> elemIdxToFaceNodes;

    // 边界条件：elemIdx → (BC名称, BC类型)
    std::map<int, std::pair<std::string, std::string>> bcInfo;

    // 合并时的偏移量
    int pointOffset = 0;  // 该 Zone 的点在全局点列表中的起始位置
    int cellOffset = 0;   // 该 Zone 的单元在全局单元列表中的起始位置
};
```

设计说明：

- 这是一个中间数据结构，在读取 CGNS 时填充，在构建最终网格后可以丢弃
- `elemIdxToFaceNodes` 存储 CGNS 中的面元素（用于边界面）
- `bcInfo` 将面元素与边界条件关联

4.5 MeshData - 最终网格数据

```
struct MeshData {
    std::vector<Point3D> points;  // 所有点（已去重、已重编号）

    std::vector<Face> faces;      // 所有面（内部面在前，边界面在后）
    int nInternalFaces = 0;       // 内部面数量

    std::vector<BoundaryPatch> boundaries;  // 边界列表（顺序对应面的顺序）

    std::vector<Zone> cellZones;   // 单元区域（用于动网格）
    std::vector<Zone> faceZones;   // 面区域

    int nCells = 0;  // 总单元数
};
```

设计说明：

- 这是输出数据结构，与 OpenFOAM polyMesh 格式一一对应
- `faces` 的顺序至关重要：索引 `[0, nInternalFaces)` 是内部面，之后是边界面

5. CGNS 读取算法详解

5.1 文件打开与基本信息读取

```
bool CgnsReader::open(const std::string& filepath) {
    // 使用 CGNS 库打开文件
    if (cg_open(filepath.c_str(), CG_MODE_READ, &fileId_) != CG_OK) {
        std::cerr << "Error: " << cg_get_error() << std::endl;
        return false;
    }

    // 获取 Base 数量（通常为 1）
    cg_nbases(fileId_, &nBases_);
    return true;
}
```

5.2 Zone 数据读取

`readZoneData()` 函数负责读取单个 Zone 的所有数据，这是一个复杂的过程：

5.2.1 读取坐标

```
// 获取 Zone 尺寸信息
cgsize_t sizes[3];  // [nVertex, nCell, nBoundaryVertex]
cg_zone_read(fileId_, baseIndex, zoneIndex, zoneName, sizes);

cgsize_t nVertex = sizes[0];
cgsize_t nCell = sizes[1];

// 读取 X, Y, Z 坐标
// 注意：CGNS 坐标读取需要指定范围 [rmin, rmax]
cgsize_t rmin = 1, rmax = nVertex;  // 1-based 范围
std::vector<double> x(nVertex), y(nVertex), z(nVertex);

cg_coord_read(fileId_, baseIndex, zoneIndex, "CoordinateX",
              RealDouble, &rmin, &rmax, x.data());
cg_coord_read(fileId_, baseIndex, zoneIndex, "CoordinateY",
              RealDouble, &rmin, &rmax, y.data());
cg_coord_read(fileId_, baseIndex, zoneIndex, "CoordinateZ",
              RealDouble, &rmin, &rmax, z.data());

// 存储到 ZoneData
zoneData.points.resize(nVertex);
for (cgsize_t i = 0; i < nVertex; ++i) {
    zoneData.points[i] = {x[i], y[i], z[i]};
}
```

关键点： `cg_coord_read` 需要指定读取范围，不能传 nullptr。

5.2.2 读取单元

CGNS 将单元组织成多个 Section，每个 Section 包含同一类型的元素：

```
int nSections;
cg_nsections(fileId_, baseIndex, zoneIndex, &nSections);

for (int is = 1; is <= nSections; ++is) {
    char sectName[33];
    ElementType_t elemType;
    cgsize_t start, end;
    int nBndry, parentFlag;

    // 读取 Section 信息
    cg_section_read(fileId_, baseIndex, zoneIndex, is,
                    sectName, &elemType, &start, &end, &nBndry, &parentFlag);

    // 获取元素数据大小
    cgsize_t elemDataSize;
    cg_ElementDataSize(fileId_, baseIndex, zoneIndex, is, &elemDataSize);

    // 读取元素数据（节点列表）
    std::vector<cgsize_t> elemData(elemDataSize);
    cg_elements_read(fileId_, baseIndex, zoneIndex, is, elemData.data(), nullptr);

    // 获取每个元素的节点数
    int nNodesPerElem;
    cg_npe(elemType, &nNodesPerElem);

    // 根据元素类型分类处理
    if (isCellElement(elemType)) {
        // 体单元 (TETRA_4, HEXA_8, ...)
        processCellElements(elemData, nNodesPerElem, elemType, zoneData);
    } else if (isFaceElement(elemType)) {
        // 面单元 (TRI_3, QUAD_4, ...) - 用于边界
        processFaceElements(elemData, nNodesPerElem, start, end, sectName, zoneData);
    }
}
```

单元类型判断：

```
static bool isCellElement(ElementType_t elemType) {
    return elemType == TETRA_4 || elemType == TETRA_10 ||   // 四面体
           elemType == PYRA_5 || elemType == PYRA_14 ||     // 金字塔
           elemType == PENTA_6 || elemType == PENTA_15 ||   // 三棱柱
           elemType == HEXA_8 || elemType == HEXA_20;       // 六面体
}

static bool isFaceElement(ElementType_t elemType) {
    return elemType == TRI_3 || elemType == TRI_6 ||        // 三角形
           elemType == QUAD_4 || elemType == QUAD_8;        // 四边形
}
```

5.2.3 读取边界条件

CGNS 的边界条件 (BC) 定义了哪些面属于哪个边界：

```
  int nBocos;
  cg_nbocos(fileId_, baseIndex, zoneIndex, &nBocos);

  for (int ib = 1; ib <= nBocos; ++ib) {
      char bocoName[33];
      BCType_t bocoType;
      PointSetType_t ptsetType;
      cgsize_t npnts;
      int normalIndex;
      cgsize_t normalListSize;
      DataType_t normalDataType;
      int nDataSet;

      cg_boco_info(fileId_, baseIndex, zoneIndex, ib,
                   bocoName, &bocoType, &ptsetType, &npnts,
                   nullptr, &normalListSize, &normalDataType, &nDataSet);

      // 读取点集（元素范围或点列表）
      std::vector<cgsize_t> pnts(npnts);
      cg_boco_read(fileId_, baseIndex, zoneIndex, ib, pnts.data(), nullptr);

      // 将 CGNS BC 类型转换为 OpenFOAM 类型
      std::string foamType = bcTypeToFoamType(bocoType);

      // 关联到面元素
      if (ptsetType == ElementRange) {
          // pnts[0] 和 pnts[1] 是元素索引范围
          for (cgsize_t elemIdx = pnts[0]; elemIdx <= pnts[1]; ++elemIdx) {
              zoneData.bcInfo[elemIdx] = {sanitizeName(bocoName), foamType};
          }
      }
  }
```

BC 类型映射：

```
static std::string bcTypeToFoamType(BCType_t bcType) {
    switch (bcType) {
        case BCWall:
        case BCWallViscous:
            return "wall";
        case BCInflow:
        case BCInflowSubsonic:
            return "inlet";
        case BCOutflow:
        case BCOutflowSubsonic:
            return "outlet";
        case BCSymmetryPlane:
            return "symmetryPlane";
        default:
            return "patch";
    }
}
```

5.3 Zone 连接读取

对于多区域网格，需要读取 Zone 之间的连接信息：

```
bool CgnsReader::readZoneConnectivity(int baseIndex, int nZones,
                                      std::vector<ZoneData>& zones,
                                      std::vector<ZoneInterface>& interfaces) {
    for (int iz = 1; iz <= nZones; ++iz) {
        // 读取一般连接 (用于 AMI 接口)
        int nConns;
        cg_nconns(fileId_, baseIndex, iz, &nConns);

        for (int ic = 1; ic <= nConns; ++ic) {
            char connName[33], donorName[33];
            GridLocation_t location;
            GridConnectivityType_t connType;
            PointSetType_t ptsetType, donorPtsetType;
            cgsize_t npnts, nDataDonor;
            ZoneType_t donorZoneType;
            DataType_t donorDataType;

            cg_conn_info(fileId_, baseIndex, iz, ic,
                         connName, &location, &connType, &ptsetType, &npnts,
                         donorName, &donorZoneType, &donorPtsetType, &donorDataType, &nDa

            // 读取本侧和对侧的点/元素列表
            std::vector<cgsize_t> pnts(npnts);
            std::vector<cgsize_t> donorPnts(nDataDonor);
            cg_conn_read(fileId_, baseIndex, iz, ic, pnts.data(),
                         donorDataType, donorPnts.data());

            // 创建接口数据结构
            ZoneInterface iface;
            iface.name = sanitizeName(connName);
            iface.zoneIndex1 = iz - 1;
            iface.zoneName1 = zones[iz-1].name;

            // 查找对侧 Zone
            for (int dz = 0; dz < nZones; ++dz) {
                if (zones[dz].name == sanitizeName(donorName)) {
                    iface.zoneIndex2 = dz;
                    iface.zoneName2 = zones[dz].name;
                    break;
                }
            }

            // 存储面元素索引
```

```
                for (cgsize_t p : pnts) {
                    iface.faceIndices1.push_back(static_cast<int>(p));
                }

                interfaces.push_back(iface);
            }
        }
        return true;
    }
```

6. 网格拓扑构建算法详解

这是整个程序最核心、最复杂的部分。 `buildMultiZoneMesh()` 函数将各个 Zone 的数据合并成一个统一的 OpenFOAM 格式网格。

6.1 算法概述

整个算法可以分为以下主要步骤：

1. **偏移量计算**：为每个 Zone 计算点和单元的全局偏移
2. **点合并**：将所有 Zone 的点合并到一个列表
3. **接口处理**：识别并标记 Zone 间的接口面
4. **BC 映射构建**：建立面到边界条件的映射
5. **面提取**：从每个单元提取其组成面
6. **拓扑构建**：确定每个面的 owner 和 neighbour
7. **面分类**：将面分为内部面和边界面
8. **面排序**：按 OpenFOAM 要求排序
9. **未使用点移除**：清理未被引用的点

6.2 偏移量计算与点合并

```
void CgnsReader::buildMultiZoneMesh(MeshData& mesh,
                                    std::vector<ZoneData>& zones,
                                    const std::vector<ZoneInterface>& interfaces) {
    // 步骤 1: 计算偏移量并合并点
    int pointOffset = 0;
    int cellOffset = 0;

    for (auto& zone : zones) {
        zone.pointOffset = pointOffset;  // 记录该 Zone 点的起始全局索引
        zone.cellOffset = cellOffset;    // 记录该 Zone 单元的起始全局索引

        // 将该 Zone 的点追加到全局点列表
        for (const auto& p : zone.points) {
            mesh.points.push_back(p);
        }

        pointOffset += static_cast<int>(zone.points.size());
        cellOffset += static_cast<int>(zone.cellNodes.size());
    }

    mesh.nCells = cellOffset;  // 总单元数
```

图示：

```
Zone1: 1000 points, 500 cells
Zone2: 600 points, 300 cells

合并后:
Points: [Zone1 points: 0-999] [Zone2 points: 1000-1599]
Cells:  [Zone1 cells: 0-499]  [Zone2 cells: 500-799]

Zone1.pointOffset = 0,    Zone1.cellOffset = 0
Zone2.pointOffset = 1000, Zone2.cellOffset = 500
```

6.3 面的唯一标识 (faceKey)

为了识别两个单元共享的同一个面，我们需要一种方法来唯一标识一个面。由于面可能有不同的节点顺序（取决于从哪个单元看），我们使用排序后的节点列表：

```
static std::string faceKey(const std::vector<int>& nodes) {
    std::vector<int> sorted = nodes;
    std::sort(sorted.begin(), sorted.end());

    std::string key;
    for (int n : sorted) {
        key += std::to_string(n) + "_";
    }
    return key;
}
```

示例：

- 从单元 A 看到的面：节点 [3, 1, 2]
- 从单元 B 看到的面：节点 [1, 2, 3]（同一个面，不同顺序）
- 两者的 faceKey 都是 "1_2_3_"

6.4 从单元提取面

每种单元类型都有固定数量和类型的面。 `getCellFaces()` 函数根据单元类型返回其所有面：

```
std::vector<std::vector<int>> getCellFaces(int cellType, const std::vector<int>& nodes) {
    std::vector<std::vector<int>> faces;

    switch (cellType) {
        case TETRA_4:
        case TETRA_10:
            // 四面体有 4 个三角形面
            // 节点编号遵循 CGNS 约定
            //
            //        3
            //       /|\
            //      / | \
            //     /  |  \
            //    /   |   \
            //   0----+----2
            //    \   |   /
            //     \  |  /
            //      \ | /
            //        1
            //
            faces.push_back({nodes[0], nodes[2], nodes[1]});  // 底面 (法向下)
            faces.push_back({nodes[0], nodes[1], nodes[3]});  // 前面
            faces.push_back({nodes[1], nodes[2], nodes[3]});  // 右面
            faces.push_back({nodes[0], nodes[3], nodes[2]});  // 左面
            break;

        case HEXA_8:
        case HEXA_20:
        case HEXA_27:
            // 六面体有 6 个四边形面
            //
            //       7--------6
            //      /|       /|
            //     / |      / |
            //    4--------5  |
            //    |  3-----|--2
            //    | /      | /
            //    |/       |/
            //    0--------1
            //
            faces.push_back({nodes[0], nodes[3], nodes[2], nodes[1]});  // 底面
            faces.push_back({nodes[4], nodes[5], nodes[6], nodes[7]});  // 顶面
            faces.push_back({nodes[0], nodes[1], nodes[5], nodes[4]});  // 前面
```

```
                faces.push_back({nodes[2], nodes[3], nodes[7], nodes[6]});  // 后面
                faces.push_back({nodes[1], nodes[2], nodes[6], nodes[5]});  // 右面
                faces.push_back({nodes[0], nodes[4], nodes[7], nodes[3]});  // 左面
                break;

            // 其他单元类型类似...
        }

        return faces;
    }
```

注意：节点顺序决定了面的法向量方向。按照约定，面的法向量应该指向单元外部（对于体单元提取的面）。

6.5 Owner/Neighbour 确定算法

这是拓扑构建的核心。算法基于一个关键观察：**每个内部面恰好被两个单元共享，每个边界面只被一个单元拥有**。

```
struct FaceInfo {
    std::vector<int> nodes;   // 面的节点
    int owner = -1;           // 第一个遇到的单元
    int neighbour = -1;       // 第二个遇到的单元（边界面为 -1）
    std::string bcName;       // 边界名称（如果是边界面）
    std::string bcType;       // 边界类型
    bool isInterface = false; // 是否是 Zone 间接口
};

std::unordered_map<std::string, FaceInfo> faceMap;

// 遍历所有 Zone 的所有单元
for (size_t zi = 0; zi < zones.size(); ++zi) {
    auto& zone = zones[zi];
    int nCells = static_cast<int>(zone.cellNodes.size());

    for (int ci = 0; ci < nCells; ++ci) {
        // 计算全局单元索引
        int globalCellIdx = ci + zone.cellOffset;

        // 将节点转换为全局索引
        std::vector<int> globalNodes;
        for (int n : zone.cellNodes[ci]) {
            globalNodes.push_back(n + zone.pointOffset);
        }

        // 从单元提取面
        auto faces = getCellFaces(zone.cellTypes[ci], globalNodes);

        // 处理每个面
        for (auto& faceNodes : faces) {
            std::string key = faceKey(faceNodes);

            auto it = faceMap.find(key);
            if (it == faceMap.end()) {
                // 第一次遇到这个面，当前单元是 owner
                FaceInfo info;
                info.nodes = faceNodes;
                info.owner = globalCellIdx;
                faceMap[key] = info;
            } else {
                // 第二次遇到这个面，这是一个内部面
                // 确保 owner < neighbour（OpenFOAM 要求）
```

```
                if (globalCellIdx < it->second.owner) {
                    it->second.neighbour = it->second.owner;
                    it->second.owner = globalCellIdx;
                    it->second.nodes = faceNodes;  // 使用新 owner 的节点顺序
                } else {
                    it->second.neighbour = globalCellIdx;
                }
            }
        }
    }
}
```

算法图示：

```
单元 A (index=5):          单元 B (index=8):

    ┌────────┐                    ┌────────┐
    │        │   共享面           │        │
    │   A    ├────────────────────┤   B    │
    │        │  faceKey="1_2_3"   │        │
    └────────┘                    └────────┘

第一次遍历到 (从单元 A):
  faceMap["1_2_3"] = {nodes=[3,1,2], owner=5, neighbour=-1}

第二次遍历到 (从单元 B):
  faceMap["1_2_3"] = {nodes=[3,1,2], owner=5, neighbour=8}

因为 5 < 8，所以 owner=5, neighbour=8 保持不变。
```

6.6 应用边界条件和接口信息

在确定了 owner/neighbour 后，需要为边界面分配边界条件：

```
for (auto& [key, info] : faceMap) {
    if (info.neighbour < 0) {  // 只有一个单元拥有的面是边界面
        // 检查是否有预定义的边界条件
        auto bcIt = faceKeyToBc.find(key);
        if (bcIt != faceKeyToBc.end()) {
            info.bcName = bcIt->second.name;
            info.bcType = bcIt->second.type;
        }

        // 检查是否是 Zone 间接口
        auto ifIt = interfaceInfo.find(key);
        if (ifIt != interfaceInfo.end()) {
            info.bcName = ifIt->second.patchName;
            info.bcType = "cyclicAMI";
            info.isInterface = true;
            info.neighbourPatch = ifIt->second.neighbourPatch;
        }
    }
}
```

6.7 面分类与排序

OpenFOAM 要求面按特定顺序排列：

```
// 分类面
std::vector<Face> internalFaces;
std::map<std::string, BoundaryEntry> boundaryFacesByName;
std::vector<Face> unnamedBoundaryFaces;

for (auto& [key, info] : faceMap) {
    Face face;
    face.nodes = info.nodes;
    face.owner = info.owner;
    face.neighbour = info.neighbour;

    if (info.neighbour >= 0) {
        // 内部面
        internalFaces.push_back(face);
    } else {
        // 边界面
        if (!info.bcName.empty()) {
            boundaryFacesByName[info.bcName].faces.push_back(face);
            boundaryFacesByName[info.bcName].bcType = info.bcType;
        } else {
            unnamedBoundaryFaces.push_back(face);
        }
    }
}

// 排序内部面：按 (owner, neighbour) 升序
std::sort(internalFaces.begin(), internalFaces.end(),
    [](const Face& a, const Face& b) {
        if (a.owner != b.owner) return a.owner < b.owner;
        return a.neighbour < b.neighbour;
    });

// 每个 patch 内的面按 owner 排序
for (auto& [name, entry] : boundaryFacesByName) {
    std::sort(entry.faces.begin(), entry.faces.end(),
        [](const Face& a, const Face& b) { return a.owner < b.owner; });
}

// 构建最终面列表
mesh.faces.clear();

// 先添加内部面
for (auto& f : internalFaces) {
```

```
        mesh.faces.push_back(f);
    }
    mesh.nInternalFaces = internalFaces.size();

    // 再添加边界面，同时构建 boundary 列表
    for (auto& [bcName, entry] : boundaryFacesByName) {
        BoundaryPatch patch;
        patch.name = bcName;
        patch.type = entry.bcType;
        patch.startFace = mesh.faces.size();
        patch.nFaces = entry.faces.size();

        for (auto& f : entry.faces) {
            mesh.faces.push_back(f);
        }

        mesh.boundaries.push_back(patch);
    }
```

面的最终顺序：

```
mesh.faces:
┌─────────────────────────────────────────────────────────┐
│  Internal faces (sorted by owner, then neighbour)       │
│  Index: 0 to nInternalFaces-1                           │
├─────────────────────────────────────────────────────────┤
│  Boundary patch 1 faces (sorted by owner)               │
│  Index: patch1.startFace to patch1.startFace+nFaces-1   │
├─────────────────────────────────────────────────────────┤
│  Boundary patch 2 faces (sorted by owner)               │
│  Index: patch2.startFace to patch2.startFace+nFaces-1   │
├─────────────────────────────────────────────────────────┤
│  ...                                                    │
└─────────────────────────────────────────────────────────┘
```

6.8 未使用点的移除

CGNS 文件中可能包含一些没有被任何单元引用的点。这些点会导致 ParaView 等软件出现问题：

```
// 收集所有被引用的点
std::set<int> usedPoints;
for (const auto& face : mesh.faces) {
    for (int n : face.nodes) {
        usedPoints.insert(n);
    }
}

// 如果有未使用的点，进行清理
if (usedPoints.size() < mesh.points.size()) {
    std::cout << "Removing " << (mesh.points.size() - usedPoints.size())
              << " unused points..." << std::endl;

    // 创建旧索引到新索引的映射
    std::vector<int> oldToNew(mesh.points.size(), -1);
    std::vector<Point3D> newPoints;
    newPoints.reserve(usedPoints.size());

    int newIdx = 0;
    for (int oldIdx : usedPoints) {  // usedPoints 是有序的
        oldToNew[oldIdx] = newIdx++;
        newPoints.push_back(mesh.points[oldIdx]);
    }

    // 更新所有面的节点索引
    for (auto& face : mesh.faces) {
        for (int& n : face.nodes) {
            n = oldToNew[n];
        }
    }

    mesh.points = std::move(newPoints);
}
```

7. 多区域网格处理

7.1 多区域网格的特点

在旋转机械（如风扇、涡轮）模拟中，网格通常包含多个区域：

- **静止区域 (Stator)**：包围旋转部件的外部区域
- **旋转区域 (Rotor)**：包含叶片的内部区域

这些区域之间通过滑移网格接口 (Sliding Mesh Interface) 连接，在 OpenFOAM 中使用 cyclicAMI (Arbitrary Mesh Interface) 实现。

7.2 全局索引计算

每个 Zone 在 CGNS 中使用本地索引。合并时需要转换为全局索引：

```
Zone 1 (airZone):     Zone 2 (bladeZone):
  点: 0 - 1999          点: 0 - 999
  单元: 0 - 4999        单元: 0 - 1999

合并后:
  Zone 1: pointOffset=0,    cellOffset=0
  Zone 2: pointOffset=2000, cellOffset=5000

  总点数: 3000 (2000 + 1000)
  总单元: 7000 (5000 + 2000)
```

当处理 Zone 2 的单元时：

```
for (int localNode : zone2.cellNodes[ci]) {
    int globalNode = localNode + zone2.pointOffset;  // +2000
}
int globalCellIdx = ci + zone2.cellOffset;  // +5000
```

7.3 边界名称冲突处理

不同 Zone 可能有相同名称的边界（如都有 "inlet"）。为避免冲突：

```
bool multiZone = zones.size() > 1;

for (auto& zone : zones) {
    for (auto& [elemIdx, bcData] : zone.bcInfo) {
        std::string bcName = bcData.first;

        // 多区域时添加 Zone 名称前缀
        if (multiZone) {
            bcName = zone.name + "_" + bcName;
        }

        // "inlet" -> "airZone_inlet" 或 "bladeZone_inlet"
    }
}
```

8. cyclicAMI 接口处理

8.1 cyclicAMI 简介

cyclicAMI (Arbitrary Mesh Interface) 是 OpenFOAM 中用于连接两个非共形网格区域的边界条件。它允许：

- 两侧网格可以有不同的密度
- 支持相对运动（滑移网格）
- 通过插值在接口处传递数据

8.2 接口识别与配对

从 CGNS 的 GridConnectivity 读取接口信息：

```
struct InterfaceData {
    std::string patchName;       // 本侧 patch 名称
    std::string neighbourPatch;  // 对侧 patch 名称
    bool isPeriodic;
    std::array<double, 3> rotationAxis;
    std::array<double, 3> rotationCenter;
    double rotationAngle;
};

// 处理每个接口
for (const auto& iface : interfaces) {
    auto& zone1 = zones[iface.zoneIndex1];
    auto& zone2 = zones[iface.zoneIndex2];

    // 接口两侧的 patch 名称
    std::string patchName1 = iface.name + "_" + zone1.name;  // "Interface_1_airZone"
    std::string patchName2 = iface.name + "_" + zone2.name;  // "Interface_1_bladeZone"

    // Zone 1 侧的面
    for (int elemIdx : iface.faceIndices1) {
        auto it = zone1.elemIdxToFaceNodes.find(elemIdx);
        if (it != zone1.elemIdxToFaceNodes.end()) {
            std::vector<int> globalNodes;
            for (int n : it->second) {
                globalNodes.push_back(n + zone1.pointOffset);
            }
            std::string key = faceKey(globalNodes);

            interfaceInfo[key] = {
                patchName1,           // 本侧 patch
                patchName2,           // 对侧 patch (neighbourPatch)
                iface.isPeriodic,
                iface.rotationAxis,
                iface.rotationCenter,
                iface.rotationAngle
            };
        }
    }

    // Zone 2 侧的面 (注意旋转角度取反)
    for (int elemIdx : iface.faceIndices2) {
        // 类似处理，但 rotationAngle 取反
        interfaceInfo[key] = {
```

```
            patchName2,
            patchName1,  // 配对反向
            iface.isPeriodic,
            iface.rotationAxis,
            iface.rotationCenter,
            -iface.rotationAngle  // 角度取反
        };
    }
}
```

8.3 cyclicAMI 边界输出

在 boundary 文件中，cyclicAMI 需要额外的配对信息：

```
// boundary 文件中的 cyclicAMI 条目
Interface_1_airZone
{
    type            cyclicAMI;
    nFaces          12566;
    startFace       798664;
    neighbourPatch  Interface_1_bladeZone;  // 关键：指定配对 patch
}
Interface_1_bladeZone
{
    type            cyclicAMI;
    nFaces          12566;
    startFace       811230;
    neighbourPatch  Interface_1_airZone;    // 互相指向
}
```

9. OpenFOAM 文件写入

9.1 文件格式概述

OpenFOAM 的 polyMesh 文件采用统一的格式：

```
static void writeFoamHeader(std::ostream& os, const std::string& className,
                            const std::string& object) {
    os << "/*--------------------------------*- C++ -*----------------------------------*'
       << "| =========                 |
       << "| \\\\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
       << "|  \\\\    /   O peration     | Website:  https://openfoam.org
       << "|   \\\\  /    A nd           | Version:  converted by cgns2openfoam
       << "|    \\\\/     M anipulation  |
       << "\\*---------------------------------------------------------------------------
       << "FoamFile\n"
       << "{\n"
       << "    version     2.0;\n"
       << "    format      ascii;\n"
       << "    class       " << className << ";\n"
       << "    object      " << object << ";\n"
       << "}\n"
       << "// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * /.
}
```

9.2 points 文件

```
// points 文件内容示例
161714          // 点数
(
(0.001 0.002 0.003)    // 点 0
(0.004 0.005 0.006)    // 点 1
...
)

// 写入代码
f << mesh.points.size() << "\n(\n";
f << std::fixed << std::setprecision(16);
for (const auto& p : mesh.points) {
    f << "(" << p.x << " " << p.y << " " << p.z << ")\n";
}
f << ")\n";
```

9.3 faces 文件

```
// faces 文件内容示例
864752          // 面数
(
3(1 3 2)        // 面 0: 3 个节点
4(5 6 7 8)      // 面 1: 4 个节点
...
)

// 写入代码
f << nTotalFaces << "\n(\n";
for (const auto& face : mesh.faces) {
    f << face.nodes.size() << "(";
    for (size_t i = 0; i < face.nodes.size(); ++i) {
        f << face.nodes[i];
        if (i + 1 < face.nodes.size()) f << " ";
    }
    f << ")\n";
}
f << ")\n";
```

9.4 owner 和 neighbour 文件

```
// owner 文件
864752          // 面数（所有面都有 owner）
(
0               // 面 0 的 owner
0               // 面 1 的 owner
1               // 面 2 的 owner
...
)

// neighbour 文件
798664          // 内部面数（只有内部面有 neighbour）
(
1               // 面 0 的 neighbour
2               // 面 1 的 neighbour
...
)
```

9.5 boundary 文件

```
// boundary 文件内容示例
4               // patch 数量
(
    Interface_1_airZone
    {
        type            cyclicAMI;
        nFaces          12566;
        startFace       798664;
        neighbourPatch  Interface_1_bladeZone;
    }
    Interface_1_bladeZone
    {
        type            cyclicAMI;
        nFaces          12566;
        startFace       811230;
        neighbourPatch  Interface_1_airZone;
    }
    airZone_outlet
    {
        type            patch;
        nFaces          39976;
        startFace       823796;
    }
    bladeZone_blade
    {
        type            wall;
        nFaces          980;
        startFace       863772;
    }
)
```

9.6 cellZones 文件

cellZones 对于动网格至关重要，它定义了哪些单元属于旋转区域：

```
// cellZones 文件内容示例
2               // Zone 数量
(
airZone
{
    type cellZone;
    cellLabels List<label> 342533(
        0 1 2 3 4 5 6 7 8 9
        10 11 12 13 14 ...
    );
}

bladeZone
{
    type cellZone;
    cellLabels List<label> 73321(
        342533 342534 342535 ...
    );
}
)
```

10. 易错点深度分析

10.1 CGNS 索引是 1-based

问题描述：
CGNS 中所有索引（节点、元素）从 1 开始，而 C++ 数组和 OpenFOAM 使用 0-based 索引。

错误表现：

- 网格在 ParaView 中显示错位
- 面的形状完全错误
- OpenFOAM 报告索引越界

解决方案：

```
// 读取 CGNS 元素时减 1
for (int in = 0; in < nNodesPerElem; ++in) {
    int cgnsNode = static_cast<int>(elemData[idx++]);
    int cppNode = cgnsNode - 1;  // ✓ 转换为 0-based
    nodes.push_back(cppNode);
}
```

测试方法：

```
// 验证所有节点索引在有效范围内
for (const auto& face : mesh.faces) {
    for (int n : face.nodes) {
        assert(n >= 0 && n < mesh.points.size());
    }
}
```

10.2 名称中的特殊字符

问题描述：
CGNS 允许名称包含空格和特殊字符（如 "Interface 1"），但 OpenFOAM 不支持。

错误表现：

- ParaView 无法打开文件
- OpenFOAM 解析错误
- 边界条件无法正确应用

解决方案：

```
static std::string sanitizeName(const std::string& name) {
    std::string result = name;

    // 替换非法字符
    for (char& c : result) {
        if (c == ' ' || c == '/' || c == '\\' || c == ':' || c == ';' ||
            c == '(' || c == ')' || c == '[' || c == ']') {
            c = '_';
        }
    }

    // 移除首尾下划线
    while (!result.empty() && result.front() == '_') result.erase(0, 1);
    while (!result.empty() && result.back() == '_') result.pop_back();

    // 合并连续下划线
    std::regex multi_underscore("_+");
    result = std::regex_replace(result, multi_underscore, "_");

    return result;
}
```

10.3 未使用的点导致内存错误

问题描述：
CGNS 文件可能包含没有被任何单元引用的孤立点。这些点虽然不影响 OpenFOAM 求解，但会导致 ParaView 出现内存错误。

根本原因：
ParaView 的 OpenFOAM 读取器会为每个点分配内存，但当尝试可视化时，未使用的点没有关联的单元信息，导致访问异常。

解决方案：
参见 6.8 未使用点的移除

验证方法：

```
# Python 验证脚本
used_points = set()
for face in faces:
    for node in face.nodes:
        used_points.add(node)

print(f"Total points: {len(points)}")
print(f"Used points: {len(used_points)}")
print(f"Unused points: {len(points) - len(used_points)}")

# 检查索引连续性
assert max(used_points) == len(used_points) - 1, "Points not properly renumbered"
```

10.4 面的节点顺序与法向量

问题描述：
面的节点顺序决定了法向量方向。OpenFOAM 要求：

- 内部面：法向量从 owner 指向 neighbour
- 边界面：法向量指向外部

潜在问题：
如果 `getCellFaces()` 中的节点顺序定义错误，可能导致：

- 流场计算错误
- 边界条件方向反转
- 网格质量检查失败

当前处理：
节点顺序遵循 CGNS 的标准约定，并假设它们正确定义了外法向。

验证方法：

```
# 使用 OpenFOAM 的 checkMesh
checkMesh -case testCase | grep -i "face"
# 检查是否有 "faces with incorrect orientation" 警告
```

10.5 边界面必须在内部面之后

问题描述：
OpenFOAM 的面列表有严格顺序要求：`[内部面] [边界面]`

错误表现：

- `checkMesh` 报告 "face ordering error"
- 边界条件无法正确应用
- 求解器崩溃

解决方案：

```
// 确保先添加内部面
for (auto& f : internalFaces) {
    mesh.faces.push_back(f);
}
mesh.nInternalFaces = internalFaces.size();

// 然后添加边界面
for (auto& patch : boundaries) {
    patch.startFace = mesh.faces.size();  // 记录起始位置
    for (auto& f : patch.faces) {
        mesh.faces.push_back(f);
    }
}
```

10.6 cyclicAMI 必须成对出现

问题描述：
cyclicAMI 边界必须成对定义，且 `neighbourPatch` 必须正确指向。

错误表现：

- "cyclicAMI patch has no neighbour" 错误
- 接口处数据不连续
- 旋转模拟失败

解决方案：

```
// Zone1 侧
interfaceInfo[key1] = {patchName1, patchName2, ...};

// Zone2 侧 - 必须互相指向
interfaceInfo[key2] = {patchName2, patchName1, ...};
```

10.7 CGNS API 参数类型

问题描述：
某些 CGNS API 对参数类型有严格要求，使用错误类型会导致未定义行为。

常见错误：

```
// 错误：cg_base_read 需要 int*
cgsize_t cellDim, physDim;  // ✗
cg_base_read(fileId, baseIndex, name, &cellDim, &physDim);

// 正确
int cellDim, physDim;  // ✓
cg_base_read(fileId, baseIndex, name, &cellDim, &physDim);

// 错误：cg_coord_read 需要范围参数
cg_coord_read(..., nullptr, nullptr, data);  // ✗ 某些版本不支持

// 正确
cgsize_t rmin = 1, rmax = nVertex;  // ✓
cg_coord_read(..., &rmin, &rmax, data);
```

11. 调试与验证方法

11.1 网格质量检查

```
# 基本检查
checkMesh -case testCase

# 详细检查
checkMesh -case testCase -allGeometry -allTopology

# 关注以下输出
# - Mesh OK: 网格通过所有检查
# - ***Error: 存在严重问题
# - ***Warning: 存在潜在问题
```

11.2 ParaView 可视化验证

```
# 打开网格
paraview testCase/test.foam

# 检查项目：
# 1. 网格是否完整显示
# 2. 边界是否正确着色
# 3. 单元/面/点数量是否正确
# 4. 没有孤立点或异常面
```

11.3 数据一致性脚本

```python
#!/usr/bin/env python3
"""验证 polyMesh 数据一致性"""

import re
from pathlib import Path

def read_foam_list(filepath):
    """读取 OpenFOAM 列表文件"""
    with open(filepath) as f:
        content = f.read()

    # 提取列表内容
    match = re.search(r'(\d+)\s*\n\s*\(([^)]*)\)', content, re.DOTALL)
    if not match:
        return None, []

    count = int(match.group(1))
    items = match.group(2).strip().split('\n')
    return count, items

def main():
    polyMesh = Path("testCase/constant/polyMesh")

    # 读取基本信息
    nPoints, points = read_foam_list(polyMesh / "points")
    nFaces, faces = read_foam_list(polyMesh / "faces")
    nOwner, owners = read_foam_list(polyMesh / "owner")
    nNeighbour, neighbours = read_foam_list(polyMesh / "neighbour")

    print(f"Points: {nPoints}")
    print(f"Faces: {nFaces}")
    print(f"Internal faces: {nNeighbour}")
    print(f"Boundary faces: {nFaces - nNeighbour}")

    # 验证 1: owner 数量 = 面数量
    assert nOwner == nFaces, f"Owner count mismatch: {nOwner} vs {nFaces}"

    # 验证 2: neighbour 数量 < 面数量
    assert nNeighbour < nFaces, "Neighbour count should be less than face count"

    # 验证 3: 检查节点索引范围
```

```
        max_node = -1
        for face in faces:
            # 解析面定义 "3(1 2 3)"
            match = re.match(r'\d+\(([^)]+)\)', face.strip())
            if match:
                nodes = [int(n) for n in match.group(1).split()]
                max_node = max(max_node, max(nodes))

        assert max_node < nPoints, f"Node index out of range: {max_node} >= {nPoints}"

        print("\n✓ All validation passed!")

    if __name__ == "__main__":
        main()
```

11.4 调试输出

程序在关键步骤输出调试信息：

```
========================================
CGNS to OpenFOAM Converter
========================================

Reading base: Base (cellDim=3, physDim=3)
  Found 2 zones
  Zone 1: airZone (vertices=107370, cells=342533)
    Section 1: airZone (type=10, range=1-342533)     <- 体单元
    Section 2: outlet (type=5, range=342534-382509)  <- 边界面
    Found 2 boundary conditions
      BC 1: outlet (type=19, ptset=4, npnts=2)
  Zone 2: bladeZone (vertices=54344, cells=73321)
    ...
    Zone 1 has 1 general connections
      Conn 1: Interface 1 -> bladeZone (type=3)      <- 接口连接

  Removing 78264 unused points...                    <- 清理未使用点

Multi-zone mesh built:
  Total points: 83450      <- 清理后的点数
  Total cells: 415854
  Internal faces: 798664
  Boundary patches: 4
    - Interface_1_airZone (cyclicAMI): 12566 faces   <- 接口配对
    - Interface_1_bladeZone (cyclicAMI): 12566 faces
    - airZone_outlet (patch): 39976 faces
    - bladeZone_blade (wall): 980 faces
```

12. 性能优化建议

12.1 当前性能特点

操作	复杂度	瓶颈分析
读取 CGNS	O(n)	I/O 操作
面提取	O(cells × faces_per_cell)	CPU
faceKey 生成	O(faces × nodes_per_face × log(nodes))	string 操作

操作	复杂度	瓶颈分析
faceMap 查找	O(faces) 平均	hash 冲突
未使用点检测	O(faces × nodes)	内存访问
写入 OpenFOAM	O(n)	I/O 操作

12.2 优化建议

12.2.1 faceKey 优化

当前使用 string 作为 key，可以改用数值哈希：

```
// 当前实现 (string-based)
static std::string faceKey(const std::vector<int>& nodes) {
    std::vector<int> sorted = nodes;
    std::sort(sorted.begin(), sorted.end());
    std::string key;
    for (int n : sorted) {
        key += std::to_string(n) + "_";
    }
    return key;
}

// 优化实现 (hash-based)
static size_t faceKeyHash(const std::vector<int>& nodes) {
    std::vector<int> sorted = nodes;
    std::sort(sorted.begin(), sorted.end());

    size_t hash = 0;
    for (int n : sorted) {
        hash ^= std::hash<int>{}(n) + 0x9e3779b9 + (hash << 6) + (hash >> 2);
    }
    return hash;
}
```

12.2.2 并行处理

面提取可以并行化：

```
#pragma omp parallel for
for (int ci = 0; ci < nCells; ++ci) {
    auto faces = getCellFaces(cellTypes[ci], cellNodes[ci]);

    #pragma omp critical
    {
        for (auto& face : faces) {
            // 更新 faceMap
        }
    }
}
```

12.2.3 内存预分配

```
// 预估面数量并预分配
// 对于四面体网格：nFaces ≈ 2 × nCells
// 对于六面体网格：nFaces ≈ 3 × nCells
size_t estimatedFaces = 2.5 * mesh.nCells;
faceMap.reserve(estimatedFaces);
mesh.faces.reserve(estimatedFaces);
```

13. 扩展开发指南

13.1 支持新的单元类型

要添加新的单元类型（如 MIXED），需要：

1. 在 `isCellElement()` 中添加类型判断
2. 在 `getCellFaces()` 中添加面提取逻辑

```
case MIXED:
    // MIXED 类型需要逐个元素解析类型标记
    // elemData 格式：[type1, n1, n2, ..., type2, n1, n2, ...]
    break;
```

13.2 支持高阶单元

当前只使用线性节点。要支持高阶单元：

```
case TETRA_10:
    // TETRA_10 有 10 个节点：4 个顶点 + 6 个边中点
    // 可以选择：
    // 1. 只使用前 4 个节点（当前做法）
    // 2. 保留所有节点用于高阶求解器
    // 3. 细分为多个线性四面体
    break;
```

13.3 添加点合并功能

不同 Zone 的共享点可以合并以减少点数：

```
// 使用空间哈希表查找相近点
struct SpatialKey {
    int ix, iy, iz;  // 网格索引

    bool operator==(const SpatialKey& other) const {
        return ix == other.ix && iy == other.iy && iz == other.iz;
    }
};

struct SpatialHash {
    size_t operator()(const SpatialKey& k) const {
        return std::hash<int>{}(k.ix) ^
               (std::hash<int>{}(k.iy) << 1) ^
               (std::hash<int>{}(k.iz) << 2);
    }
};

std::unordered_map<SpatialKey, int, SpatialHash> pointHash;
double cellSize = tolerance * 10;  // 网格大小

for (const auto& p : points) {
    SpatialKey key = {
        static_cast<int>(p.x / cellSize),
        static_cast<int>(p.y / cellSize),
        static_cast<int>(p.z / cellSize)
    };

    // 检查该网格及相邻网格中是否有相近点
    // ...
}
```

13.4 添加二进制输出

OpenFOAM 支持二进制格式，可以显著减小文件大小：

```
// 在 FoamFile 头中指定
os << "    format      binary;\n";

// 二进制写入
void writeBinary(std::ostream& os, const std::vector<double>& data) {
    os.write(reinterpret_cast<const char*>(data.data()),
             data.size() * sizeof(double));
}
```

附录 A: CGNS 元素类型参考

类型	名称	节点数	描述
5	TRI_3	3	线性三角形
6	TRI_6	6	二次三角形
7	QUAD_4	4	线性四边形
8	QUAD_8	8	二次四边形（无中心）
9	QUAD_9	9	二次四边形（有中心）
10	TETRA_4	4	线性四面体
11	TETRA_10	10	二次四面体
12	PYRA_5	5	线性金字塔
13	PYRA_14	14	二次金字塔
14	PENTA_6	6	线性三棱柱
15	PENTA_15	15	二次三棱柱
16	PENTA_18	18	二次三棱柱（有面中心）
17	HEXA_8	8	线性六面体
18	HEXA_20	20	二次六面体
19	HEXA_27	27	二次六面体（有中心）

类型	名称	节点数	描述
20	MIXED	-	混合元素

附录 B: OpenFOAM 边界类型参考

类型	描述	用途
patch	通用边界	用户定义边界条件
wall	壁面	固壁边界
inlet	入口	速度/压力入口
outlet	出口	压力出口
symmetryPlane	对称面	对称边界
cyclicAMI	任意网格接口	滑移网格、非共形接口
cyclic	周期边界	周期重复结构
empty	空边界	2D 模拟的前后面

文档版本: 2.0
最后更新: 2026-02-20
作者: cgns2openfoam 开发团队