Introduction to OpenFOAM Programming

03 - Openfoam 数据结构

王佳琪

上海交通大学

2022 年 1 月

基础类的源码组织特点

通常为一个 xxx.H 文件，外加一个 xxxI.H 文件，一个 xxx.C 文件。可能会出现的 xxxIO.C 文件，以及通过 List 类创建的 xxxList.H 文件，以及他们的 IO 处理，xxxIOList.H，以及他们的 List 创建的 xxxListIOList.H 或.C 文件（大型套娃现场）。

1. xxxI.H 文件通常为 xxx.H 文件中 inline 函数的具体实现
2. xxx.C 文件通常为 xxx.H 文件中一般函数的具体实现
3. xxxIO.C 文件是和 IO 相关的实现，例如 >> 和 << 的重定义，以及 readxxx 这种函数的实现
4. xxxList.H 其实是用 List 创建了新的类型，这里的 List 可以暂时理解成一个数组

```
1  typedef List<label> labelList;
2  typedef List<labelList> labelListList;
3  typedef List<labelListList> labelListListList;
```

xxxIOList.H 和 xxxIOList.C 中前者为新类型 typedef IOList<xxx> xxxIOList;，而后者为通过一系列的宏实现一些基础的函数功能.

```
1  // xxxListIOList.H
2  typedef IOList<xxxList> xxxListIOList;
```

03-blog

int

分 unsigned 和一般两种，前者的文件最前有一个 u 字母，源文件有:

```
int32.C  int32.H  int32IO.C
int64.C  int64.H  int64IO.C
int.H  intIO.H
label.H  label.C labelList.H  labelIOList.H  labelIOList.C labelListIOLit.H
uint32.C  uint32.H  uint32IO.C
uint64.C  uint64.H  uint64IO.C
uint.H  uintIO.H
uLabel.H  uLabel.C
```

其中继承关系为 int32 int64->int->label，其中 32 和 64 分别代表 32bit 和 64bit 的整数。他们分别使用 int32_t 和 int64_t 进行定义，并给定了和文件流相关的函数。
03-blog

Label

其实就是指 i,j,k 这类浮标使用的类型。我们通常就使用 int 就可以，但是这里也进行了封装：

```
1  typedef INT_SIZE(int, _t) label;  //label这个类型其实是int
2  //相当于int64_t，这在int64.H中有进行定义，如果机器是32位的，就是int32_t
```

继承了 Label 和 List 类

```
1  typedef List<label> labelList;
2  typedef List<labelList> labelListList;
3  typedef List<labelListList> labelListListList;
```

02-doc1 02-doc2

char

给 Foam 名字域下的 Istream 和 Ostream 提供运算符重定义和 readChar 函数

```
1 // char.H
2 char readChar(Istream&);
3 Istream& operator>>(Istream&, char&);
4 Ostream& operator<<(Ostream&, const char);
5 Ostream& operator<<(Ostream&, const char*);
6 inline bool isspace(char c)
```

第一个函数 readChar 的作用是读取文件流中的一个 char;
第二个函数代表读取文件流中的一个 char，然后将它赋值给参数;
第三第四个函数其实是同一个运算符 << 的重定义，只不过根据输入的是 char 本身还是 &char 进行了多态;

```
1 // charIO.C
2 char Foam::readChar(Istream& is) ..
3 Foam::Istream& Foam::operator>>(Istream& is, char& c) ..
4 Foam::Ostream& Foam::operator<<(Ostream& os, const char c) ..
5 Foam::Ostream& Foam::operator<<(Ostream& os, const char* s) ..
```

02-file 03-blog

wchar

与上述 char 类似，都是给 Istream 和 Ostream 进行了运算符 >> 和 << 的重定义，这里的注释提示了是针对 output wide character (Unicode) as UTF-8，就是在输出的同时还需要转码成 UTF-8。

```
1 // wchar.H
2 //- Output wide character (Unicode) as UTF-8
3 Ostream& operator<<(Ostream&, const wchar_t);
4 Ostream& operator<<(Ostream&, const wchar_t*);
5 Ostream& operator<<(Ostream&, const std::wstring&);
```

wchar 是 char 类型定义一个扩展表达，可以表示更多的字符类型，代价是需要用更多的字节数，且在不同的库中可能会用不同的字节数，最多为 4 字节。第三个函数使用了迭代器。

```
1 // wcharIO.C
2 #include "error.H"
3 Foam::Ostream& Foam::operator<<(Ostream& os, const wchar_t wc) ..
4 Foam::Ostream& Foam::operator<<(Ostream& os, const wchar_t* wstr) ..
5 Foam::Ostream& Foam::operator<<(Ostream& os, const std::wstring& wstr) ..
```

02-file 03-blog 04-wiki

char 相关的类

继承关系为 char->string->word->wordRe fileName keyType。源文件有：

```
1  char.H charIO.C wchar.H wcharIO.c
2
3  string.H stringI.H string.C stringIO.C
4  stringIOList.H stringIOList.stringOps.H stringOps.C
5
6  word.H word.C  wordI.H wordIO.C wordIOList.H wordIOList.C
7  wordRe.H wordRe.C wordReI.H wordList.H wordReList.H wordReListMatcher.
       wordReListMatcherI.H
8
9  keyType.H keyTypeI.H keyType.C
10
11 fileName.H fileNameI.H fileName.C fileNameIO.C fileNameList.H
```

1. char 直接使用了 C 语言内置的 char，但是添加了和文件流操作相关的运算符和函数，这些话函数均可以独立调用。而 wchar 则使用了更多字节的编码形式（A wide-character and a pointer to a wide-character string）
2. string 则使用了 std:string 在 Foam 的名字空间进行封装，添加了内置的变量 typeName debug null，以及一系列的功能性函数。
3. word keyType fileName 在均为 string 的子类，他们在 string 的基础上添加了更多的功能，使得他们可以用在更加具体的情形中。

03-blog

string

```
$> ls
fileName functionName lists string stringOps variable verbatimString word wordRe
```

```
    // string.H
    #include "char.H" //
    #include "Hasher.H" //代码实现了哈希的功能
    #include <string> // 继承了std::string
    #include <cstring>
    #include <cstdlib>
```

把多态扩展到了 string。下面为类的声明，为一系列的构造函数和成员函数，以及操作符重载。

```
    // string.H
class string;
Istream& operator>>(Istream&, string&);
Ostream& operator<<(Ostream&, const string&);
Ostream& operator<<(Ostream&, const std::string&);
```

01-dox 02-file 03-blog

string

stringI.H 来自相同文件夹，其实是上述文件中所有 inline 函数的具体实现，基本上都是通过直接调用 std 名字域中的 string 变量来。而其余函数的具体实现，在 string.C 中。
除此之外，openfoam 还定义了一些自己的 string 成员函数：

```
1  // string.C
2  Foam::string::size_type Foam::string::count(const char c) const //计数
3  Foam::string& Foam::string::replace(const string& oldStr,const string&
       newStr,size_type start=0) //替换
4  ...
5  string()(const size_type i,const size_type n) //从第i个开始的n个字符
6  string()(const size_type n)  //从头开始的n个字符
7  operator=(const string&);  //赋值
8  operator=(string&&);       //赋值
```

01-dox 02-file 03-blog

stringList

套娃

```
1 #include "string.H"
2 #include "List.H"
3 namespace Foam
4 {
5     typedef UList<string> stringUList;
6     typedef List<string> stringList;
7 }
```

02-file 03-blog

UList

定义了 UList 类，类似于 array
第一个变量用来存储 UList 的尺寸，第二个变量用来存储首地址。

```
1  // UList.H
2  template<class T>
3  class UList
4  {
5      // Private Data
6          //- Number of elements in UList
7          label size_;
8          //- Vector of values of type T
9          T* __restrict__ v_;//首地址
```

我们给定一个 UList 类的对象 list 和游标 label i 输入后，i 会遍历 list 中的每一个元素。

```
1  // UList.H:426
2  // forAll宏定义
3  #define forAll(list, i) \
4      for (Foam::label i=0; i<(list).size(); i++)
```

01-dox 02-file 03-blog

UList

基础功能：

1. 通过内置的类 less 和 greater 规定大小的判断。
2. 通过构造函数进行赋值
3. 返回迭代器的收尾位置
4. 通过检索范围检查当前的 List 的正确性
5. 深度和前度 copy，区别在于只复制指针还是连带整个数据一起复制
6. 基本的索引 [] = 等的重定义
7. 迭代器
8. 返回当前 List 尺寸，判断为空，交换的成员函数
9. 和 IO 相关的函数
10. 类外定义的相关排序等功能。

01-dox 02-file 03-blog

List

List 继承了 UList, 并增加了一些函数:

1. 创建内存空间
2. 返回 size，以及 resize
3. clear 和 append 操作
4. = 的操作符重定义

List

用 List 和各种基础类创建的类型

```
1 typedef List<label> labelList;
2 typedef UList<scalar> scalarUList;
3 typedef List<scalar> scalarList;
```

List 中的文件读取

```
1 template<class T>
2 Foam::List<T> Foam::readList(Istream& is)
```

01-dox 02-file 03-blog

DynamicList

类似于 std::vector

DLList

类似于 std::list
DLList.H: 非侵入式双向链表
DLPtrList.H: 非侵入式双向链接指针列表
FIFOStack.H: 基于单链表的 FIFO 堆栈，先进先出
IDLList.H: 侵入式双向链表
ISLList.H: 侵入式单链表
LIFOStack.H: 基于单链表的 LIFO 堆栈，后进后出
SLList.H：非侵入式单链表

HashTable

类似于 std::map

scalar

其实就是浮点数，不过浮点数有多重类型精度，比如 float double longdouble，这里将类型统一为 scalar 这个类型使用.
这里并没有定义新的类，而是使用了 typedef，然后添加了一些新的功能:

```
1 typedef double doubleScalar;
2 typedef long double longDoubleScalar;
```

```
1 Scalar.H  Scalar.C
2 doubleFloat.H
3 doubleScalar.H  doubleScalar.C
4 floatScalar.H  floatScalar.C
5 longDoubleScalar.H  longDoubleScalar.C
6 scalar.H  scalar.C
7 scalarList.H  scalarList.C
8 scalarIOList.H  scalarIOList.C
9 scalarListIOList.H scalarListIOList.C
```

03-blog

VectorSpace

并不是 C++ 中常用的容器 vector，而是指 (x,y,z) 这样的长度为 3 的行向量，用来表示速度坐标等。这是因为 openFOAM 从 90 年代开始迭代遗留下来的习惯

```
1 //依赖关系
2 Vectorspace->Vector->vector, floatVector, labelVector, complexVector
```

向量空间使用的是类型 Cmpt 作为元素类型，Ncmpts 代表着元素的个数。数学逻辑上，这里相当于一个行向量.

```
1         // VectorSpace.H
2         template<class Form, class Cmpt, direction Ncmpts>
3         class VectorSpace
4         {
5         public;
6             Cmpt v_[Ncmpts];     //- The components of this vector space
7             typedef VectorSpace<Form, Cmpt, Ncmpts> vsType;    //- VectorSpace
                  type
8             typedef Cmpt cmptType;    //- Component type
```

02-file 03-blog

Vector

```
1  //Vector.H
2  template<class Cmpt>
3  class Vector
4  :
5      public VectorSpace<Vector<Cmpt>, Cmpt, 3>
6  {
7  public:
8      //- Equivalent type of labels used for valid component indexing
9      typedef Vector<label> labelType;
10     // Member constants
11     static const direction rank = 1;
12     //- Component labeling enumeration
13     enum components { X, Y, Z };
14
15     构造和析构函数
16
17     // Member Functions
18     用来返回x.y.z三个方向的元素，以及进行加减乘除操作
19 };
```

对应的 *I.H 文件对其中的 inline 函数进行了具体的实现

02-file 03-blog

vector

前面的 Vector.H 中指定元素的类型模版，这里为把元素类型指定为 scalar 的特例

```
1  //vector.H
2  typedef Vector<scalar> vector;
3  //floatvector.H:typedef Vector<float> floatVector
4  template<>
5  inline bool contiguous<vector>() {return true;}
6  template<class Type>
7  class flux : public innerProduct<vector, Type> {};
8  template<> class flux<scalar> {
9      public:
10     typedef scalar type;       };
```

对应的.C 文件则是对原来的 VectorSpace 中的成员变量进行了修改:

```
1  const char* const Foam::vector::vsType::typeName = "vector";
2  const char* const Foam::vector::vsType::componentNames[] = {"x", "y", "z"};
3  const Foam::vector Foam::vector::vsType::vsType::zero(vector::uniform(0));
4  const Foam::vector Foam::vector::vsType::one(vector::uniform(1));
5  const Foam::vector Foam::vector::vsType::max(vector::uniform(vGreat));
6  const Foam::vector Foam::vector::vsType::min(vector::uniform(-vGreat));
7  const Foam::vector Foam::vector::vsType::rootMax(vector::uniform(rootVGreat));
8  const Foam::vector Foam::vector::vsType::rootMin(vector::uniform(-rootVGreat));
```

02-file 03-blog

MatrixSpace

```
1 //依赖关系
2 VectorSpace->MatrixSpace->Tensor->tensor,floatTensor,labelTensor
```

首先 VectorSpace 为一个一维数组，模板的第三个输入值即为数组的长度，在逻辑上为一个行向量。而当前用它创建矩阵，相当于在内存上，矩阵的元素为连续存储的，在逻辑上才是矩阵。在运算过程中，需要根据下标计算从首地址向后的位移。

```
1  // MatrixSpace.H
2  template<class Form, class Cmpt, direction Mrows, direction Ncols>
3  class MatrixSpace
4  : public VectorSpace<Form, Cmpt, Mrows*Ncols>
5  {
6  public:
7      //- MatrixSpace type
8      typedef MatrixSpace<Form, Cmpt, Mrows, Ncols> msType;
9      // Member constants
10     static const direction mRows = Mrows;
11     static const direction nCols = Ncols;
12     //构造函数和析构函数
13     //返回其中的某个元素，或者整体size的函数
14     //等号的操作符重定义
15 };
```

02-file 03-blog

Tensor

```
1  //Tensor.H
2  template<class Cmpt>
3  class Tensor
4  :   public MatrixSpace<Tensor<Cmpt>, Cmpt, 3, 3>
5  {
6  public:
7      //- Equivalent type of labels used for valid component indexing
8      typedef Tensor<label> labelType;
9      // Member constants
10     //- Rank of Tensor is 2
11     static const direction rank = 2;
12     // Static Data Members
13      static const Tensor I;
14     //- Component labeling enumeration
15     enum components { XX, XY, XZ, YX, YY, YZ, ZX, ZY, ZZ };
16      构造函数和析构函数
17      返回其中的某个元素，以及等于的重定义
18 };
```

相当于是 MatrixSpace 的一个 3*3 的特殊情况，不过元素类型还需要指定，此时的各个元素可以单独编号为 XX, XY, XZ, YX, YY, YZ, ZX, ZY, ZZ。因为 VectorSpace 中已经实现了大多数功能，这里需要写的东西比较少了。并且也同样只有源码 *I.H 没有.C，它实现了其中的 inline 函数.

02-file 03-blog

tensor

相当于前面 Tensor 的指定元素为 scalar 类型。因为是浮点数操作，添加了一些特征值相关的函数，头文件如下

```
namespace Foam
{
typedef Tensor<scalar> tensor;
// floatTensor.H: typedef Tensor<float> floatTensor;
vector eigenValues();
vector eigenVector();
tensor eigenVectors();
//几个函数均为多态，有多重可输入的形式

} // End namespace Foam
```

其.C 文件提供了具体的实现，因为是 3*3 的矩阵，求特征值并不会很困难。

```
template<>  const char* const Foam::tensor::vsType::typeName = "tensor";
template<>  const char* const Foam::tensor::vsType::componentNames[] =
{
    "xx", "xy", "xz",
    "yx", "yy", "yz",
    "zx", "zy", "zz"
};
Foam::vector Foam::eigenValues(const tensor& t) ...
```

02-file 03-blog

Field

```
//依赖关系
List->Field->DimensionedField->GeometricField->finiteVolume中的*Field
```

1.Field 是最基础的域，它继承了 List 的一维数组的结构用来存储域的元素
2.DimensionedField 则是继承自 Field，它在 Field 的基础上添加了和网格 Mesh 相关以及文件流 regIOobject 相关量
3.GeometricField 继承自 DimensionedField，在其基础上添加和边界相关的量以及时间戳，并建立了不同时间戳下的域之间的联系。
4. 最终通过 GeometricField 的 template 模板中的类型指定为特定类型，定义出我们平时使用的域的类型

```
        // Field.H 抽象
template<class Type>
class Field
:
    public tmp<Field<Type>>::refCount,
    public List<Type>
{

    1.构造与析构函数
    2.map函数，replace函数，拷贝函数
    3.= += *= /= << >>的操作符重定义
```

01-dox 02-file 03-blog 04-test

DimensionedField

带有单位的域，并且和 GeoMesh 类型相关，用网格信息进行域的尺寸控制

```
1  // DimensionedField.H 抽象
2  template<class Type, class GeoMesh>
3  class DimensionedField
4  :
5      public regIOobject,
6      public Field<Type>
7  {    成员变量:
8       //- Reference to mesh
9       const Mesh& mesh_;
10      //- Dimension set for this field
11      dimensionSet dimensions_;
12      成员函数:
13      构造和析构函数
14      域读取用的函数
15      网格和单位的返回
16      replace T average weightedAverage函数
17      = += -= /= */ << >>的操作符重定义
18 };
```

01-dox 02-file 03-blog

DimensionedField

构造函数:
可以看出 io 代表的是文件流的初始化，而 Mesh dimensionSet Field<Type> 也是直接通过构造函数初始化给它的成员变量。在初始化结束后，代码会检查当前的 field 的 size 是否和 mesh 的 size 一致。
就是说这里除去域本身，还添加了文件流，和网格的变量。

```
1  // DimensionedField.H
2  template<class Type, class GeoMesh>
3  DimensionedField<Type, GeoMesh>::DimensionedField
4  (
5      const IOobject& io,
6      const Mesh& mesh,
7      const dimensionSet& dims,
8      const Field<Type>& field
9  )
10 :
11     regIOobject(io),
12     Field<Type>(field),
13     mesh_(mesh),
14     dimensions_(dims)
```

01-dox 02-file 03-blog

GeometricField

在前一个父类的基础上添加了更多的成员变量. 可以看出，除去原来的和文件流以及网格相关的量之外，还添加了边界和时间戳。并且通过成员变量中的指针，建立了不同时间戳的域之间的联系。

```
1  template<class Type, template<class> class PatchField, class GeoMesh>
2  class GeometricField
3  :     public DimensionedField<Type, GeoMesh>
4  {     除去定义新的类型之外，还给定了一个新的类Boundary
5        成员变量:
6           //  Used to trigger(触发) the storing of the old-time value
7           mutable label timeIndex_;
8           //- Pointer to old time field
9           mutable GeometricField<Type, PatchField, GeoMesh>* field0Ptr_;
10          //-  Pointer to previous iteration (used for under-relaxation)
11          mutable GeometricField<Type, PatchField, GeoMesh>* fieldPrevIterPtr_;
12          //- Boundary Type field containing boundary field values
13          Boundary boundaryField_;
14 成员函数:     读取文件
15              构造函数和析构函数
16              返回内部或者边界域
17              返回时间戳
18              松弛和替换
19              最大最小值的返回
20             =    ==    +=    -=    *=    /=    <<    >> };
```

01-dox 02-file 03-blog

GeometricField

构造函数:

```
1  template<class Type, template<class> class PatchField, class GeoMesh>
2  Foam::GeometricField<Type, PatchField, GeoMesh>::GeometricField
3  (
4      const IOobject& io,
5      const Mesh& mesh,
6      const dimensionSet& ds,
7      const wordList& patchFieldTypes,
8      const wordList& actualPatchTypes
9  )
10 :
11     Internal(io, mesh, ds, false),
12     timeIndex_(this->time().timeIndex()),
13     field0Ptr_(nullptr),
14     fieldPrevIterPtr_(nullptr),
15     boundaryField_(mesh.boundary(), *this, patchFieldTypes, actualPatchTypes)
```

01-dox 02-file 03-blog

finiteVolume 中的 *Field

我们常用的一些类均在 volFieldsFwd.H 中

```
1  //类的前置声明
2  class volMesh;
3  template<class Type> class fvPatchField;
4  template<class Type, template<class> class PatchField, class GeoMesh>
       class GeometricField;
5  //常用的类的类名定义
6  typedef GeometricField<scalar, fvPatchField, volMesh> volScalarField;
7  typedef GeometricField<vector, fvPatchField, volMesh> volVectorField;
8  typedef GeometricField<sphericalTensor, fvPatchField, volMesh>
       volSphericalTensorField;
9  typedef GeometricField<symmTensor, fvPatchField, volMesh>
       volSymmTensorField;
10 typedef GeometricField<tensor, fvPatchField, volMesh> volTensorField;
```

02-file 03-blog

- Space and time: polyMesh, fvMesh, Time
- Field algebra: Field, DimensionedField and GeometricField
- Boundary conditions: fvPatchField and derived classes
- Sparse matrices: lduMatrix, fvMatrix and linear solvers
- Finite Volume discretisation: fvc and fvm namespace

文件流的获取——lookup 函数

1. dictionary 一系的类，可以检索文件中的关键字，并返回一个文件流。
2. 几乎所有的基础类型都提供了文件流的导入和导出操作，所以只需要使用能够给定文件流的类，就可以读取相应的文件。

```
1  //检索文件流的过程:
2  lookup->lookupEntry->lookupEntryPtr
```

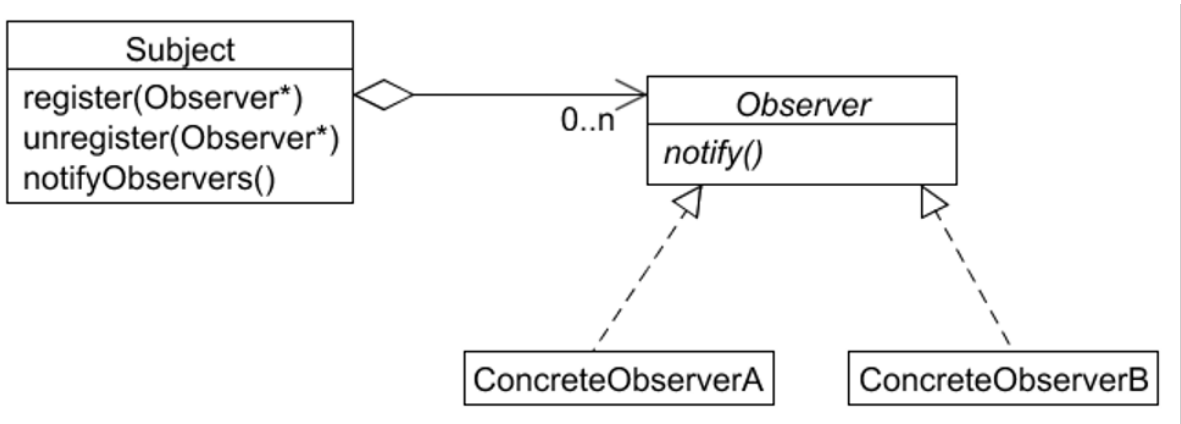
3. 函数 lookupEntry，通过调用 lookupEntryPtr 得到 entry* entryPtr。检查一下是否检索成功，并最终返回指针指向的对象。
4. 最终调用的是函数 lookupEntryPtr，他通过 dictionary 的 HashTable 中的 find，以 keyword 查找 entry*，并最终返回。中间是如果当前字典没有检索到，就扩大检索范围。
5. 函数 lookup 就只有一行 return lookupEntry(keyword, recursive, patternMatch).stream();，他通过.stream() 调用 entry 类的函数。而 entry 类中的这个函数为纯虚函数，他的实现在 primitiveEntry.C 中，内容如下:

```
1 Foam::ITstream& Foam::primitiveEntry::stream() const
2 {
3     ITstream& is = const_cast<primitiveEntry&>(*this);
4     is.rewind();
5     return is;  //即得到了一个文件流并返回。
6 }
```

02-file 03-blog 04-test

对象注册机制

对象注册（object registry）机制是 OpenFOAM 的一大特点。对象注册所做的工作可以总结为一句话：在内存中利用树状结构组织数据，并实现数据的管理及输入输出。
The object registry is an implementation of the Observer Pattern:
观察者模式 (Observer Pattern): 定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新




```
1 //继承关系:
2 IOobject->regIOobject->objectRegistry
```

05-design
06-youtube

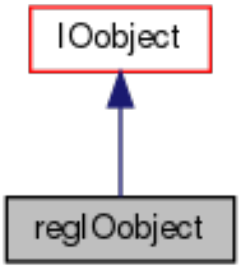
IOobject

IOobject 是树状结构中节点属性的集合。树状结构中的每一个节点都是一个 regIOobject 注册对象。将这个对象的属性提取出来，用 IOobject 类描述。而 regIOobject 继承自 IOobject，获得所有属性。

```
1  IOdictionary transportProperties
2  (
3      IOobject
4      (
5          "transportProperties",   //对象名字，作为关键字查找返回该对象;
6          runTime.constant(),      //对象路径，表示从 constant 目录读取或写入;
7          mesh,                    //对象的父对象，表示注册为 mesh 的一个子节点
8          IOobject::MUST_READ_IF_MODIFIED,//读选项为 修改后重新读入
9          IOobject::NO_WRITE       //写选项为 不写
10     )
11 );
```

regIOobject

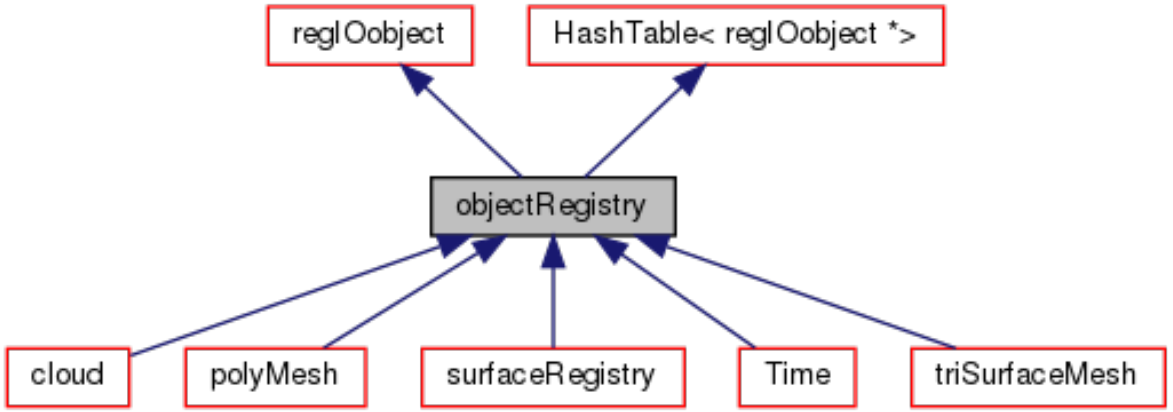
regIOobject 继承自 IOobject。根据 OpenFOAM 源码中的注释，这个类和 objectRegistry 一起实现了自动对象注册，可以理解为整个树状结构的管理及输入输出。和 IOobject 相比，regIOobject 实现了树状结构的增加和删除节点等管理操作，以及对象如何从文件读取或写入到文件等读写操作。



01-dox

objectRegistry

objectRegistry 用来作为树状结构的根节点以及分支节点。它继承自 regIOobject，本身也是一个 regIOobject 对象，包含子节点的所有信息，这些信息存在它的另一个父类——HashTable<regIOobject> 中。继承关系如下



哈希表保存了对象名字（word 类型）-对象指针（regIOobject * 类型）的键值对，通过哈希表实现注册对象的查找和返回，具体可参考 objectRegistry 的 lookupObject 方法。

01-dox

树状结构的管理-增加节点

增加节点的操作通过 checkIn 实现，通常在定义对象时自动完成。
参考 regIOobject 的构造函数：

```
1  Foam::regIOobject::regIOobject(const IOobject& io, const bool isTime)
2  :
3      IOobject(io),
4      registered_(false),
5      ownedByRegistry_(false),
6      watchIndices_(),
7      eventNo_                    // Do not get event for top level Time
           database
8      (
9          isTime
10       ? 0
11       : db().getEvent()
12     )
13 {
14     // Register with objectRegistry if requested
15     if (registerObject())
16     {
17         checkIn(); //
18     }
19 }
```

树状结构的管理-树状结构的读取

树状结构的读取相关的操作通过 read 、readData 和 readStream 实现。在两种情况下会出发读取操作：一种是定义对象时，另一种是将对象的 rOpt_ 定义为 READ_IF_MODIFIED 并且修改磁盘文件时。
1. 从磁盘读取:
对于第一种情况，在派生类的构造函数中调用读取文件的操作。以 fvSchemes 类为例，该类在构造函数中调用了 read 函数从 system/fvSchemes 文件中读取内容:

```
1  Foam::fvSchemes::fvSchemes(const objectRegistry& obr) :
2      IOdictionary
3      (
4          IOobject
5          (
6              "fvSchemes",
7              obr.time().system(),
8              obr,
9              (//..),
10             IOobject::NO_WRITE
11         )
12     ),
13 //  ...
14         read(schemesDict());
```

树状结构的管理-树状结构的读取

树状结构的读取相关的操作通过 read 、readData 和 readStream 实现。在两种情况下会出发读取操作：一种是定义对象时，另一种是将对象的 rOpt_ 定义为 READ_IF_MODIFIED 并且修改磁盘文件时。
1. 第二种情况较为复杂。每次执行 Time::run 时会调用 Time::readModifiedObjects 函数：

```
1  bool Foam::Time::run() const
2  {
3      // ...
4      if (running)
5      {
6          if (!subCycling_)
7          {
8              const_cast<Time&>(*this).readModifiedObjects();
9              // ...
10         }
11         // Re-evaluate if running in case a function object has changed
               things
12         running = this->running();
13     }
14
15     return running;
16 }
```

- Time::readModifiedObjects 先读取 system/controlDict，然后调用 objectRegistry::readModifiedObjects 函数.
- objectRegistry::readModifiedObjects 递归遍历树状结构，并调用每个节点的 readIfModified 函数, 实际调用自身的 read 方法.

树状结构的管理-树状结构的写入

根节点为 Time 类型，通常只有为 fvMesh （及其派生）类型的子节点，其他节点都放在 mesh 底下。求解器中的 runTime.write() 触发了树状结构的写入操作。该函数实际调用的是 regIOobject::write()，而这个函数又将调用 Time::writeObject 函数：

1. 先通过 controlDict 中设置的参数判断当前时刻是否为需要写入，若需要则继续；
2. 调用 writeTimeDict 函数，往 [time]/uniform/time 文件中写入和时间相关的变量；
3. 调用 objectRegistry::writeObject 函数，这个函数将判断子节点的 wOpt_ 属性，若不为 NO_WRITE 则调用子节点的 writeObject 函数；
4. 对于 runTime 对象，调用 mesh 子节点的 writeObject 函数，该函数先写入网格相关数据（动网格相关），再调用 polyMesh::writeObject 函数，该函数没有重写，实际调用的是父类中的函数 objectRegistry::writeObject ，递归遍历 mesh 节点下子节点的 writeObject 函数；

5. 最后是 purgeWrite 的相关操作。