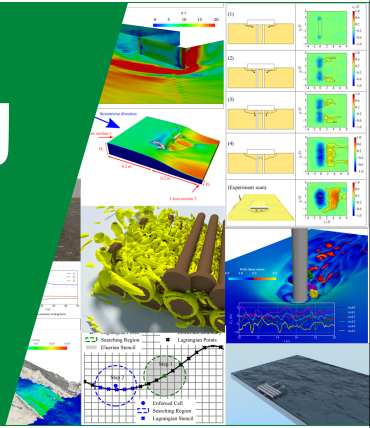
中国农业大学　流体机械与流体工程系

2023 年春季《计算流体动力学编程实践》

OpenFOAM® 代码结构

徐云成

✉ycxu@cau.edu.cn



2023 年 2 月 28 日

面向对象程序设计

OpenFOAM®uses C++ Object-Oriented programming

面向过程（Procedure Oriented 简称 PO ： 如 C 语言）

从名字可以看出它是注重过程的。当解决一个问题的时候，面向过程会把事情拆分成：一个个函数和数据（用于方法的参数）。然后按照一定的顺序，执行完这些方法（每个方法看作一个过程），等方法执行完了，事情就搞定了。

面向对象（Object Oriented 简称 OO ： 如 C++，JAVA 等语言）

看名字它是注重对象的。当解决一个问题的时候，面向对象会把事物抽象成对象的概念，就是说这个问题里面有哪些对象，然后给对象赋一些属性和方法，然后让每个对象去执行自己的方法，问题得到解决。

来源https://zhuanlan.zhihu.com/p/75265007

面向对象程序设计

OpenFOAM®uses C++ Object-Oriented programming

面向过程（Procedure Oriented 简称 PO ： 如 C 语言）

从名字可以看出它是注重过程的。当解决一个问题的时候，面向过程会把事情拆分成：一个个函数和数据（用于方法的参数）。然后按照一定的顺序，执行完这些方法（每个方法看作一个过程），等方法执行完了，事情就搞定了。

面向对象（Object Oriented 简称 OO ： 如 C++，JAVA 等语言）

看名字它是注重对象的。当解决一个问题的时候，面向对象会把事物抽象成对象的概念，就是说这个问题里面有哪些对象，然后给对象赋一些属性和方法，然后让每个对象去执行自己的方法，问题得到解决。

来源https://zhuanlan.zhihu.com/p/75265007

面向对象程序设计

OpenFOAM®uses C++ Object-Oriented programming

面向过程（Procedure Oriented 简称 PO ： 如 C 语言）

从名字可以看出它是注重过程的。当解决一个问题的时候，面向过程会把事情拆分成：一个个函数和数据（用于方法的参数）。然后按照一定的顺序，执行完这些方法（每个方法看作一个过程），等方法执行完了，事情就搞定了。

面向对象（Object Oriented 简称 OO ： 如 C++，JAVA 等语言）

看名字它是注重对象的。当解决一个问题的时候，面向对象会把事物抽象成对象的概念，就是说这个问题里面有哪些对象，然后给对象赋一些属性和方法，然后让每个对象去执行自己的方法，问题得到解决。

来源https://zhuanlan.zhihu.com/p/75265007

面向对象程序设计

OpenFOAM®uses C++ Object-Oriented programming

面向过程（Procedure Oriented 简称 PO ： 如 C 语言）

从名字可以看出它是注重过程的。当解决一个问题的时候，面向过程会把事情拆分成：一个个函数和数据（用于方法的参数）。然后按照一定的顺序，执行完这些方法（每个方法看作一个过程），等方法执行完了，事情就搞定了。

面向对象（Object Oriented 简称 OO ： 如 C++，JAVA 等语言）

看名字它是注重对象的。当解决一个问题的时候，面向对象会把事物抽象成对象的概念，就是说这个问题里面有哪些对象，然后给对象赋一些属性和方法，然后让每个对象去执行自己的方法，问题得到解决。

来源https://zhuanlan.zhihu.com/p/75265007

面向过程 vs 面向对象

问题：洗衣机里面放有脏衣服，怎么洗干净？

面向过程

1. 执行加洗衣粉方法；
2. 执行加水方法；
3. 执行洗衣服方法；
4. 执行清洗方法；
5. 执行烘干方法.

面向对象

1. 先弄出两个对象：“洗衣机”对象和“人”对象；
2. 针对对象“洗衣机”加入一些属性和方法：“洗衣服方法”“清洗方法”、“烘干方法”；
3. 针对对象“人”加入属性和方法：“加洗衣粉方法”、“加水方法”；
4. 然后执行
   人. 加洗衣粉; 人. 加水; 洗衣机. 洗衣服; 洗衣机. 清洗; 洗衣机. 烘干

来源https://zhuanlan.zhihu.com/p/75265007

面向过程 vs 面向对象

问题：洗衣机里面放有脏衣服，怎么洗干净？

面向过程

1. 执行加洗衣粉方法；
2. 执行加水方法；
3. 执行洗衣服方法；
4. 执行清洗方法；
5. 执行烘干方法.

面向对象

1. 先弄出两个对象：“洗衣机”对象和“人”对象；
2. 针对对象“洗衣机”加入一些属性和方法：“洗衣服方法”“清洗方法”、“烘干方法”；
3. 针对对象“人”加入属性和方法：“加洗衣粉方法”、“加水方法”；
4. 然后执行
   人. 加洗衣粉; 人. 加水; 洗衣机. 洗衣服; 洗衣机. 清洗; 洗衣机. 烘干

来源https://zhuanlan.zhihu.com/p/75265007

面向过程 vs 面向对象

问题：洗衣机里面放有脏衣服，怎么洗干净？

面向过程

1. 执行加洗衣粉方法；
2. 执行加水方法；
3. 执行洗衣服方法；
4. 执行清洗方法；
5. 执行烘干方法.

面向对象

1. 先弄出两个对象：“洗衣机”对象和“人”对象；
2. 针对对象“洗衣机”加入一些属性和方法：“洗衣服方法”“清洗方法”、“烘干方法”；
3. 针对对象“人”加入属性和方法：“加洗衣粉方法”、“加水方法”；
4. 然后执行
   人. 加洗衣粉; 人. 加水; 洗衣机. 洗衣服; 洗衣机. 清洗; 洗衣机. 烘干

来源https://zhuanlan.zhihu.com/p/75265007

面向过程 vs 面向对象

面向过程

- 优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源; 比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发，性能是最重要的因素。
- 缺点：没有面向对象易维护、易复用、易扩展

面向对象

- 优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统 更加灵活、更加易于维护
- 缺点：性能比面向过程低
- 三大基本特性：封装，继承，多态

来源https://zhuanlan.zhihu.com/p/75265007

封装 Encapsulation

就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。一个类 (Class) 就是一个封装了数据以及操作这些数据的代码的逻辑实体 (instance)。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象 (object) 对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

```
Account my_account;
my_account.deposit(100.1);
my_account.withdraw(10);
```

Account 是类 (Class); my_account 是对象 (object);deposit 是函数 (function)

封装 Encapsulation

就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。一个类 (Class) 就是一个封装了数据以及操作这些数据的代码的逻辑实体 (instance)。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象 (object) 对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

```
Account my_account;
my_account.deposit(100.1);
my_account.withdraw(10);
```

`Account` 是类 (Class); `my_account` 是对象 (object);`deposit` 是函数 (function)

继承 Inheritance

继承 Inheritance

指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用父类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力。

存款账户，母类（base class）⇒ 活期账户，定期账户，子类（sub-class）

- 子类可以继承母类的数据和函数
- 子类可以有自己的数据、函数
- 子类也可以对继承的母类函数进行修改

继承 Inheritance

继承 Inheritance

指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用父类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力。

存款账户, 母类（base class）⇒ 活期账户，定期账户，子类（sub-class）

- 子类可以继承母类的数据和函数
- 子类可以有自己的数据、函数
- 子类也可以对继承的母类函数进行修改

多态 Polymorphism

多态 Polymorphism

是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

运算符号+对于不同的类scalarField 和vectorField 具有不同的定义

```
scalarField + scalarField
vectorField + vectorField
```


多态 Polymorphism

多态 Polymorphism

是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

运算符号+对于不同的类`scalarField` 和`vectorField` 具有不同的定义

```
scalarField + scalarField
vectorField + vectorField
```

Templates

- C++的特点之一，允许函数 (function) 和类 (class) 使用不同的数据类型 (data type)
- 定义 template 时，只需要通用数据类型 (generic data type)

```
template <class a_type, class b_type, ...>
class a_class
{
...
a_type a_var;
b_type b_var;
...
};
```


Templates

- C++的特点之一，允许函数 (function) 和类 (class) 使用不同的数据类型 (data type)
- 定义 template 时，只需要通用数据类型 (generic data type)

```
template <class a_type, class b_type, ...>
class a_class
{
...
a_type a_var;
b_type b_var;
...
};
```


Class templating

Examples of class templating in OpenFOAM®

- UList: A 1D array class `$FOAM_SRC/OpenFOAM/containers/Lists/UList`

```
template<class T>
class UList
{
    // Private data
        //- Number of elements in UList.
        label size_;
        //- Vector of values of type T.
        T* __restrict__ v_;
...
}
```


Class templating

Examples of class templating in OpenFOAM®

- UList: A 1D array class `$FOAM_SRC/OpenFOAM/containers/Lists/UList`

```
template<class T>
class UList
{
    // Private data
        //- Number of elements in UList.
        label size_;
        //- Vector of values of type T.
        T* __restrict__ v_;
...
}
```


模板 Function templating

Examples of function templating in OpenFOAM®

- 1D interpolation `$FOAM_SRC/OpenFOAM/interpolations/interpolateXY`

```
template<class Type>
Field<Type> interpolateXY
(
    const scalarField& xNew,
    const scalarField& xOld,
    const Field<Type>& yOld
)
{
    Field<Type> yNew(xNew.size());
    forAll(xNew, i)
    {
        yNew[i] = interpolateXY(xNew[i], xOld, yOld);
    }
    return yNew;
}
```


模板 Function templating

Examples of function templating in OpenFOAM®

- 1D interpolation $FOAM_SRC/OpenFOAM/interpolations/interpolateXY

```
template<class Type>
Field<Type> interpolateXY
(
    const scalarField& xNew,
    const scalarField& xOld,
    const Field<Type>& yOld
)
{
    Field<Type> yNew(xNew.size());
    forAll(xNew, i)
    {
        yNew[i] = interpolateXY(xNew[i], xOld, yOld);
    }
    return yNew;
}
```


复杂模板 templates

例如：OpenFOAM® 中用来定义变量场的一个非常重要的类 (class)：
GeometricField class

```
$FOAM_SRC/OpenFOAM/fields/GeometricFields/GeometricField
```

```
template<class Type, template<class> class PatchField, class GeoMesh>
class GeometricField
:
    public DimensionedField<Type, GeoMesh>
```

$FOAM_SRC/finiteVolume/fields/volFields/volFieldsFwd.H 中
volScalarField 和volVectorField 分别定义为

```
typedef GeometricField<scalar, fvPatchField, volMesh> volScalarField
typedef GeometricField<vector, fvPatchField, volMesh> volVectorField
```


复杂模板 templates

例如：OpenFOAM® 中用来定义变量场的一个非常重要的类 (class)：
GeometricField class

```
$FOAM_SRC/OpenFOAM/fields/GeometricFields/GeometricField

template<class Type, template<class> class PatchField, class GeoMesh>
class GeometricField
:
    public DimensionedField<Type, GeoMesh>
```

`$FOAM_SRC/finiteVolume/fields/volFields/volFieldsFwd.H` 中
`volScalarField` 和`volVectorField` 分别定义为

```
typedef GeometricField<scalar, fvPatchField, volMesh> volScalarField
typedef GeometricField<vector, fvPatchField, volMesh> volVectorField
```


命名空间 namespace

namespace 是用来解决全局名字命名问题的

```
// some_lib.H
namespace lib_one {
    int func(int);
    class point { ... };
}

// another_lib.H
namespace lib_two {
    int func(int);
    class point { ... };
}
```

我们可以区分为:

lib_one::point 和lib_two::point

也可以

```
using namespace lib_one {
    point a_point;
}
```


命名空间 namespace

namespace 是用来解决全局名字命名问题的

```
// some_lib.H
namespace lib_one {
    int func(int);
    class point { ... };
}

// another_lib.H
namespace lib_two {
    int func(int);
    class point { ... };
}
```

我们可以区分为:
lib_one::point 和lib_two::point
也可以

```
using namespace lib_one {
    point a_point;
}
```


命名空间 namespace

namespace 是用来解决全局名字命名问题的

```
// some_lib.H
namespace lib_one {
    int func(int);
    class point { ... };
}

// another_lib.H
namespace lib_two {
    int func(int);
    class point { ... };
}
```

我们可以区分为:
`lib_one::point` 和`lib_two::point`
也可以

```
using namespace lib_one {
    point a_point;
}
```


namespace in OpenFOAM®

- https://cpp.openfoam.org/v8/namespaces.html
- Examples: `Foam`, `fvc`, `fvm`, `incompressible`, `compressible`
- `Foam` 是 OpenFOAM® 特有的, 区别于其他 C++代码
- `fvc` 和`fvm`: 微分运算中的显性 (explicit) 和隐性 (implicit) 离散，例如: `fvc::div(...)` 和`fvm::div(...)`
- `incompressible` 和`compressible`:: 两种流体，例如
  `Foam::compressible::RASModels::kkLOmega`
  `Foam::incompressible::RASModels::PDRkEpsilon`
- namespace nesting 命名空间嵌套

namespace in OpenFOAM®

- https://cpp.openfoam.org/v8/namespaces.html
- Examples: `Foam`, `fvc`, `fvm`, `incompressible`, `compressible`
- `Foam` 是 OpenFOAM® 特有的, 区别于其他 C++代码
- `fvc` 和`fvm`: 微分运算中的显性 (explicit) 和隐性 (implicit) 离散，例如: `fvc::div(...)` 和`fvm::div(...)`
- `incompressible` 和`compressible::` 两种流体，例如
  `Foam::compressible::RASModels::kkLOmega`
  `Foam::incompressible::RASModels::PDRkEpsilon`
- namespace nesting 命名空间嵌套

namespace in OpenFOAM®

- https://cpp.openfoam.org/v8/namespaces.html
- Examples: `Foam`, `fvc`, `fvm`, `incompressible`, `compressible`
- `Foam` 是 OpenFOAM® 特有的, 区别于其他 C++代码
- `fvc` 和`fvm`: 微分运算中的显性 (explicit) 和隐性 (implicit) 离散，例如: `fvc::div(...)` 和`fvm::div(...)`
- `incompressible` 和`compressible::` 两种流体，例如

  `Foam::compressible::RASModels::kkLOmega`

  `Foam::incompressible::RASModels::PDRkEpsilon`

  namespace nesting 命名空间嵌套

namespace in OpenFOAM®

- https://cpp.openfoam.org/v8/namespaces.html
- Examples: `Foam`, `fvc`, `fvm`, `incompressible`, `compressible`
- `Foam` 是 OpenFOAM® 特有的, 区别于其他 C++代码
- `fvc` 和`fvm`: 微分运算中的显性 (explicit) 和隐性 (implicit) 离散，例如: `fvc::div(...)` 和`fvm::div(...)`
- `incompressible` 和`compressible::` 两种流体，例如

  `Foam::compressible::RASModels::kkLOmega`
  `Foam::incompressible::RASModels::PDRkEpsilon`

  namespace nesting 命名空间嵌套

namespace in OpenFOAM®

- https://cpp.openfoam.org/v8/namespaces.html
- Examples: `Foam`, `fvc`, `fvm`, `incompressible`, `compressible`
- `Foam` 是 OpenFOAM® 特有的, 区别于其他 C++代码
- `fvc` 和`fvm`: 微分运算中的显性 (explicit) 和隐性 (implicit) 离散，例如: `fvc::div(...)` 和`fvm::div(...)`
- `incompressible` 和`compressible::` 两种流体，例如

  `Foam::compressible::RASModels::kkLOmega`
  `Foam::incompressible::RASModels::PDRkEpsilon`

  namespace nesting 命名空间嵌套

Important classes in OpenFOAM®

Five basic classes:

- Time and space (mesh): `Time`, `polyMesh`, `fvMesh`
- Field (data): `Field`, `DimensionedField`, and `GeometricField`
- Boundary conditions: `fvPatchField` and derived classes
- Finite volume discretization: classes in `fvc` and `fvm` namespaces
- Sparse matrices: `lduMatrix`, `fvMatrix` and linear solvers

Thank you.

欢迎私下交流，请勿上传网络，谢谢！