lagrangian関連ライブラリの調査

2014年12月13日オープンCAE勉強会@富山(富山県立大学　中川慎二)

Disclaimer

注意

本資料の内容は，OpenFOAMユーザーガイド，プログラマーズガイド，OpenFOAM Wiki，CFD Online，その他多くの情報を参考にしています。開発者，情報発信者の皆様に深い謝意を表します。

この講習内容は，講師の個人的な経験（主に，卒研生等とのコードリーディング）から得た知識を共有するものです。この内容の正確性を保証することはできません。この情報を使用したことによって問題が生じた場合，その責任は負いかねますので，予めご了承ください。

lagrangian関連ライブラリ

ソースコードの場所：/src/lagrangian

このディレクトリには多くのライブラリのソースコードが，まとめて配置されている。

階層構造を調べるために，treeコマンドを使う。第1階層の深さで表示すると次の通り。（-L オプションにて1階層を指定。）

```
tree -L 1

├── Allwmake
├── Turbulence
├── basic
├── coalCombustion
├── distributionModels
├── dsmc
├── intermediate
├── molecularDynamics
├── solidParticle
├── spray
└── turbulence
```

もう1階層分深く調査してみる。

```
tree -L 2

├── Allwmake
├── Turbulence
│   ├── Make
│   ├── lnInclude
│   ├── parcels
│   └── submodels
├── basic
│   ├── Cloud
│   ├── IOPosition
│   ├── InteractionLists
│   ├── Make
│   ├── indexedParticle
│   ├── lnInclude
│   ├── particle
│   └── passiveParticle
├── coalCombustion
│   ├── Make
```

```
│   ├── coalCloud
│   ├── coalCloudList
│   ├── coalParcel
│   ├── include
│   ├── lnInclude
│   └── submodels
├── distributionModels
│   ├── Make
│   ├── RosinRammler
│   ├── distributionModel
│   ├── exponential
│   ├── fixedValue
│   ├── general
│   ├── lnInclude
│   ├── multiNormal
│   ├── normal
│   └── uniform
├── dsmc
│   ├── Make
│   ├── clouds
│   ├── lnInclude
│   ├── parcels
│   └── submodels
├── intermediate
│   ├── IntegrationScheme
│   ├── Make
│   ├── clouds
│   ├── lnInclude
│   ├── parcels
│   ├── phaseProperties
│   └── submodels
├── molecularDynamics
│   ├── Allwmake
│   ├── molecularMeasurements
│   ├── molecule
│   └── potential
├── solidParticle
│   ├── Make
│   ├── lnInclude
│   ├── solidParticle.C
│   ├── solidParticle.H
│   ├── solidParticleCloud.C
│   ├── solidParticleCloud.H
│   ├── solidParticleCloudI.H
│   ├── solidParticleI.H
│   ├── solidParticleIO.C
├── spray
│   ├── Make
│   ├── clouds
│   ├── lnInclude
│   ├── parcels
│   └── submodels
└── turbulence
    ├── Make
    ├── lnInclude
    ├── parcels
    └── submodels
```

この結果から，各ディレクトリの中に，Makeディレクトリが存在することがわかる。これは，Makeが存在するディレクトリ毎にコンパイルされることを意味する。Make/filesの中を確認すると，最終行にLIB＝とあり，ライブラリを作成していることがわかる。

このことは，lagrangian/Allwmakeファイルの内容からも明らかである。このファイル内では，各ディレクトリに対して wmake libso を実行しており，ディレクトリ毎にライブラリが作成されている。

これらのライブラリは，完全に独立しているのではないことに注意が必要である。

例えば，lagrangian/intermediate ディレクトリから生成されるライブラリ lagrangianIntermediate を例に考える。lagrangian/intermediate /Make/options ファイルの内容を確認すると，`EXE_INC =` では，ヘッダファイルの格納されているディレクトリを指定している。この中に，次の2行がある。これは，現在のディレクトリである intermediate 以外に，/lagrangian/basic と /lagrangian/distributionModels が，lagrangianIntermediateライブラリに関与していることを表す。

```
    -I$(LIB_SRC)/lagrangian/basic/lnInclude \
    -I$(LIB_SRC)/lagrangian/distributionModels/lnInclude \
```

これら2つのディレクトリからは，lagrangianライブラリとdistributionModelsライブラリが生成されている。このことは，LIB_LIBS = 部分に，これらのライブラリが指定されていることからも確認できる。

なお，Makeディレクトリが存在するのと同じディレクトリには，lnIncludeディレクトリも存在している。これは，wmake実行時に作成されるディレクトリである。同じディレクトリ以下にある.Cおよび.Hファイルへのリンクが格納される。これによって，このlnIncludeディレクトリを指定することで，このライブラリに関するすべてのソースコードに到達することができる。

Run-Time selection への対応

lagrangian関連ライブラリでは，モデルによって，Run-Time selectionを実現するためのコードにちがいがあるようだ。基本的な仕組みは同一であるが，マクロの記載場所が異なる。

方法1（例：lagrangian/distributionModels）

通常，Run-Time selectin機能を使うためには，ベースとなるクラスを作成し，そのベースクラスを継承して個別のモデルを実装したクラスを作成する。

例えば，distributionModelであれば，lagrangian/distributionModelsディレクトリに関連するクラスが格納されている。この中でベースとなるのは，lagrangian/distributionModels/distributionModel に入っているdistributionModelクラスである。

このベースクラス（distributionModel）では，distributionModel.H，distributionModel.C，distributionModelNew.C の3つのソースコードがある。Run-Time selectionのベースクラスでは，クラス名＋New.C というファイルが存在することが特徴である。このファイルでは，distributionModel::New関数が実装されている。これはselectorと呼ばれ，継承された個別クラスを呼び出す役目を担当する。

個別のクラスでは，自身をRun-Time selectionテーブルに登録するためのマクロ（addToRunTimeSelectionTable）を使っている。例えば，lagrangian/distributionModels/exponential.C では下記となる。

```
    namespace distributionModels
    {
        defineTypeNameAndDebug(exponential, 0);
        addToRunTimeSelectionTable(distributionModel, exponential, dictionary);
    }
```

この場合，各クラスのソースコード（.C）を Make/files ファイルに記述して，コンパイルしている。

方法2（例：lagrangian/intermediate/IntegrationScheme）

lagrangian/intermediate/IntegrationScheme では，マクロ部分を lagrangian/intermediate/IntegrationScheme /makeIntegrationSchemes.C に記載している。

```
namespace Foam
{
    makeIntegrationScheme(scalar);
    makeIntegrationSchemeType(Euler, scalar);
    makeIntegrationSchemeType(Analytical, scalar);

    makeIntegrationScheme(vector);
    makeIntegrationSchemeType(Euler, vector);
    makeIntegrationSchemeType(Analytical, vector);
}
```

同じ makeIntegrationSchemes.C では，その冒頭で，ベースクラス IntegrationScheme，Run-Time に選択できる2つの方式 Euler と Analytical のヘッダを読み込んでいる。

上記の中にあるmakeIntegrationScheme は，lagrangian/intermediate/IntegrationScheme/IntegrationScheme/IntegrationScheme.Hの下部において，下記の様に定義されている。#defineはマクロを作成するための命令である。マクロは1行で書く必要があり，行末に連結記号が書かれている。

```
#define makeIntegrationScheme(Type)                                             \
                                                                                \
    defineNamedTemplateTypeNameAndDebug(IntegrationScheme<Type>, 0);            \
                                                                                \
    defineTemplateRunTimeSelectionTable                                         \
    (                                                                           \
        IntegrationScheme<Type>,                                                \
        dictionary                                                              \
    );


#define makeIntegrationSchemeType(SS, Type)                                     \
                                                                                \
    defineNamedTemplateTypeNameAndDebug(SS<Type>, 0);                           \
                                                                                \
    IntegrationScheme<Type>::adddictionaryConstructorToTable<SS<Type> >         \
        add##SS##Type##ConstructorToTable_;
```

ここに現れるdefineNamedTemplateTypeNameAndDebugなどは，src/OpenFOAM/db/typeInfo/className.H で定義されるマクロである。下記に該当する部分の一部を示す。このclassName.H は，IntegragtionScheme.Hの冒頭 `#include "dictionary.H"` を通じてインクルードされている。

```
  101 //- Define the typeName, with alternative lookup as \a Name
  102 #define defineTypeNameWithName(Type, Name)                                  \
  103     const ::Foam::word Type::typeName(Name)

  117 //- Define the typeName as \a Name for template classes
  118 # define defineTemplateTypeNameWithName(Type, Name)                         \
  119     template<>                                                              \
  120     defineTypeNameWithName(Type, Name)

  131 //- Define the typeName directly for template classes
  132 #define defineNamedTemplateTypeName(Type)                                   \
  133     defineTemplateTypeNameWithName(Type, Type::typeName_())

  157 //- Define the typeName and debug information for templates
  158 #define defineNamedTemplateTypeNameAndDebug(Type, DebugSwitch)              \
  159     defineNamedTemplateTypeName(Type);                                      \
  160     defineNamedTemplateDebugSwitch(Type, DebugSwitch)
```

integrationScheme.H では，declareRunTimeSelectionTable 関数を宣言している。さらに，末尾に下記が存在する。

```
#ifdef NoRepository
#   include "IntegrationScheme.C"
#endif
```

ifdef は，それに続くマクロが定義されている場合にのみ実行したいpreprocessor処理を記述するために使う。
https://gcc.gnu.org/onlinedocs/cpp/Ifdef.html#Ifdef

IntegrationSchemeNew.Cは，InegrationScheme.Cの末尾でインクルードされる。

makeIntegrationSchemes.C のコンパイル → IntegrationScheme.Hのインクルード部分に IntegrationScheme.C と IntegrationSchemeNew.C が展開される→ 各実装クラスのヘッダとソースが展開される となり，コンパイルされるようだ。

方法3

lagrangian/intermediate/submodelsでは，Run-Time selection関連のマクロが lagrangian/intermediate/parcels/include にまとめられている。新たなモデルを作成したときには，こちらに登録することが必要である。

参考

https://openfoamwiki.net/index.php/OpenFOAM_guide/runTimeSelection_mechanism

http://www.sourceflux.de/blog/runtime-type-selection-openfoam/

http://www.sourceflux.de/blog/run-time-type-selection-openfoam-selecting-types-based-type-name/

http://www.sourceflux.de/blog/run-time-type-selection-openfoam-implementation/

マクロ内のオペレータ

オペレータ `#`

変数名を文字列として取得するためのオペレータ。

`#variableName` とすると，変数内部の文字列として置換される。

通常はマクロの文字列置換が真っ先に行われるのに対して，このオペレータを使うとそうはならない。

もし，

```
#define str(s) #s
#define foo 4
```

というマクロを定義したとき，コード中に `str (foo)` と記載すれば，そこには `"foo"` という文字列が置換される。ダブルクォーテーション付きで。

もし，fooで置換したい数字を文字列としたいときには，下記の様に2段階のマクロを書く必要がある。

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
```

こうした時には，コード中に `xstr (foo)` と記載すると，まずはじめに置換が行われて `xstr (4)` となり，さらに `str (4)` となり，最終的に `"4"` という文字列に置き換わることとなる。

https://gcc.gnu.org/onlinedocs/cpp/Stringification.html#Stringification

オペレータ `##`

2つの文字列を連結するためのオペレータ。

例として，下記のマクロを定義したとする。

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }
```

このとき，`COMMAND(quit)` と書くと，`{ "quit", quit_command }` と置換される。

https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html#Concatenation