

Giskard's CFD Learning Tricks

CFD and Scientific Computing

2016-03-12 • OPENFOAM

OpenFOAM 中的 Run Time Selection 机制

[source flux 博客](#) 曾经出过一个解释 Run Time Selection(RTS) 机制的系列博文，推荐想理解 RTS 的读者去仔细阅读。本篇算是我在读完以后做的一个笔记，以及一些总结，供读者参考。

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。算例具体使用的是那个模型，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。下面通过一个 [source flux 博客](#) 提供的代码，来解读 RTS 机制的实现原理。

为了方便解读，这里将代码摘录如下，代码所有权归 [source flux 博客](#) 所有：

```
1  #include "word.H"
2  #include "messageStream.H"
3  #include "argList.H"
4
5  using namespace Foam;
6
7  #include "typeInfo.H"
8  #include "runTimeSelectionTables.H"
9  #include "addToRunTimeSelectionTable.H"
```

```

9  // #include "addRunTimeSelectionTables.H"
10
11 // Main program:
12 class AlgorithmBase
13 {
14     public:
15
16     // Declare the static variable typeName of the class Algor
17     TypeName ("base");
18
19     // Empty constructor.
20     AlgorithmBase () {};
21
22     // Word constructor.
23     AlgorithmBase (const word& algorithmName) {};
24
25     // Destructor: needs to be declared virtual since
26     virtual ~AlgorithmBase() {};
27
28     // Macro for declaring stuff required for RTS
29     declareRunTimeSelectionTable
30     (
31         autoPtr,
32         AlgorithmBase,
33         Word,
34
35         (
36             const word& algorithmName
37         ),
38         (algorithmName)
39     )
40
41     // static Factory Method (selector)
42     static autoPtr<AlgorithmBase> New (const word& algorithmName)
43     {
44
45         // Find the Factory Method pointer in the RTS Table

```

```
45 // (HashTable<word, autoPtr<AlgorithmBase>*)(word))
46 WordConstructorTable::iterator cstrIter =
47     WordConstructorTablePtr_->find(algorithmName);
48
49 // If the Factory Method was not found.
50 if (cstrIter == WordConstructorTablePtr_->end())
51 {
52     FatalErrorIn
53     (
54         "AlgorithmBase::New(const word&)"
55     ) << "Unknown AlgorithmBase type "
56     << algorithmName << nl << nl
57     << "Valid AlgorithmBase types are :" << endl
58     << WordConstructorTablePtr_->sortedToc()
59     << exit(FatalError);
60 }
61
62 // Call the "constructor" and return the autoPtr<Algor
63 return cstrIter()(algorithmName);
64
65 }
66
67 // Make the class callable (function object)
68 virtual void operator>()()
69 {
70     // Overridable default implementation
71     Info << "AlgorithmBase::operator>()()" << endl;
72 }
73 };
74
75 defineTypeNameAndDebug(AlgorithmBase, 0);
76 defineRunTimeSelectionTable(AlgorithmBase, Word);
77 addToRunTimeSelectionTable(AlgorithmBase, AlgorithmBase, Word);
78
79 class AlgorithmNew
80 :
```

```
81     public AlgorithmBase
82 {
83     public:
84
85         // Declare the static variable typeName of the class Algor
86         TypeName ("new");
87
88         // Empty constructor.
89         AlgorithmNew () {};
90
91         // Word constructor.
92         AlgorithmNew (const word& algorithmName) {};
93
94         // Make the class callable (function object)
95         virtual void operator()()
96         {
97             Info << "AlgorithmNew::operator()()" << endl;
98         }
99 };
100
101 defineTypeNameAndDebug(AlgorithmNew, 0);
102 addToRunTimeSelectionTable(AlgorithmBase, AlgorithmNew , Word);
103
104 class AlgorithmAdditional
105 :
106     public AlgorithmNew
107 {
108     public:
109
110         // Declare the static variable typeName of the class Algor
111         TypeName ("additional");
112
113         // Empty constructor.
114         AlgorithmAdditional () {};
115
116         // Word constructor.
```

```
117     AlgorithmAdditional (const word& algorithmName) {};  
118  
119     // Make the class callable (function object)  
120     virtual void operator()()  
121     {  
122         // Call base operator explicitly.  
123         AlgorithmNew::operator()();  
124         // Perform additional operations.  
125         Info << "AlgorithmAdditional::operator()()" << endl;  
126     }  
127 };  
128  
129 defineTypeNameAndDebug(AlgorithmAdditional, 0);  
130 addToRunTimeSelectionTable(AlgorithmBase, AlgorithmAdditional, Wo  
131  
132 int main(int argc, char *argv[])  
133 {  
134     argList::addOption  
135     (  
136         "algorithmName",  
137         "name of the run-time selected algorithm"  
138     );  
139  
140     argList args(argc, argv);  
141  
142     if (args.optionFound("algorithmName"))  
143     {  
144         // Get the name of the algorithm from the arguments passed  
145         // application.  
146         const word algorithmName = args.option("algorithmName");  
147  
148         // RTS call.  
149         autoPtr<AlgorithmBase> algorithmPtr = AlgorithmBase::New(a  
150  
151         // Get the reference to the algorithm from the smart point  
152         AlgorithmBase& algorithm = algorithmPtr();
```

```
153
154     // Call the algorithm.
155     algorithm();
156 }
157 else
158 {
159     FatalErrorIn
160     (
161         "main()"
162     ) << "Please use with the 'algorithmName' option." << endl;
163     << exit(FatalError);
164 }
165
166 Info<< "\nEnd\n" << endl;
167
168 return 0;
169 }
```

在解读原理之前，先来看看这段代码。可以发现，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，经过搜索，发现都是定义在 `runTimeSelectionTables.H` 和 `addToRunTimeSelectionTable.H` 两个头文件中，而且，这些函数都是宏函数。

看来，理解 RTS 的第一步就需要仔细看看这几个宏函数。

先来看基类中的宏函数 `declareRunTimeSelectionTable` , 根据 [source flux 的博文](#) , 这个宏函数针对前面的那段代码的展开结果为:

```
1  typedef autoPtr< AlgorithmBase > (*WordConstructorPtr)( const word&
2
3  typedef HashTable< WordConstructorPtr, word, string::hash > WordCon
4  static WordConstructorTable* WordConstructorTablePtr_;
5  static void constructWordConstructorTables();
6  static void destroyWordConstructorTables();
7  template< class AlgorithmBaseType >
8  class addWordConstructorToTable
9  {
10     public:
11     static autoPtr< AlgorithmBase > New ( const word& algorithmName
12     {
13         return autoPtr< AlgorithmBase >(new AlgorithmBaseType (algo
14     }
15     addWordConstructorToTable ( const word& lookup = AlgorithmBaseT
16     {
17         constructWordConstructorTables();
18         if (!WordConstructorTablePtr_->insert(lookup, New))
19         {
20             std::cerr<< "Duplicate entry "
21                 << lookup << " in runtime selection table "
22                 << "AlgorithmBase" << std::endl;
23             error::safePrintStack(std::cerr);
24         }
25     }
26
27     ~addWordConstructorToTable()
28     {
29         destroyWordConstructorTables();
30     }
31 };
32 template< class AlgorithmBaseType >
```

```
33  class addRemovableWordConstructorToTable
34  {
35      const word& lookup_;
36
37      public:
38      static autoPtr< AlgorithmBase > New ( const word& algorithmName
39      {
40          return autoPtr< AlgorithmBase >(new AlgorithmBaseType (algo
41      }
42      addRemovableWordConstructorToTable ( const word& lookup = Algor
43      : lookup_(lookup)
44      {
45          constructWordConstructorTables();
46          WordConstructorTablePtr_->set(lookup, New);
47      }
48
49      ~addRemovableWordConstructorToTable()
50      {
51          if (WordConstructorTablePtr_)
52          {
53              WordConstructorTablePtr_->erase(lookup_);
54          }
55      }
56  };
```

注意，由于 `declareRunTimeSelectionTable` 是在基类类体里调用的，所以，以上内容都是在类体里的。这相当于在类体里定义了两个 `typedef`，一个静态数据成员，两个静态函数，还有两个类。

先来看这两个 `typedef`。第一个，定义的是一个函数指针，这样定义的结果是，

`WordConstructorPtr` 代表一个指向参数为 `const word&`，返回类型为 `autoPtr< AlgorithmBase >` 的函数指针。第二个好理解，将一个 `key` 和 `value` 分别为 `word` 和 `WordConstructorPtr` 的 `hashTable` 定义了一个别名 `WordConstructorTable`。

静态数据成员 `WordConstructorTablePtr_` 是一个 `WordConstructorTable` 类型的指针。

两个静态成员函数，这里只是声明了，并且注意到在下面定义的两个类中用到了这两个函数。

继续看 `defineRunTimeSelectionTable(AlgorithmBase, Word)`。这个宏展开的结果为：

```

1  AlgorithmBase::WordConstructorTable* AlgorithmBase::WordConstructor
2      void AlgorithmBase::constructWordConstructorTables() {
3      static bool constructed = false;
4      if (!constructed) {
5          constructed = true;
6          AlgorithmBase::WordConstructorTablePtr_ = new Algorithm
7      }
8  };
9
10     void AlgorithmBase::destroyWordConstructorTables() {
11         if (AlgorithmBase::WordConstructorTablePtr_) {
12             delete AlgorithmBase::WordConstructorTablePtr_;
13             AlgorithmBase::WordConstructorTablePtr_ = __null;
14         }
15     };

```

这个宏函数的主要功能，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 `WordConstructorTablePtr_` 初始化为 `__null`，然后 `constructWordConstructorTables` 函数将 `WordConstructorTablePtr_` 指向一个动态分配的 `WordConstructorTable`。
`destroyWordConstructorTables` 则是对指针 `WordConstructorTablePtr_` 进行销毁。

接着，`addToRunTimeSelectionTable(AlgorithmBase, AlgorithmBase, Word)`，这宏函数展开以后其实就一句话：

```

1  AlgorithmBase::addWordConstructorToTable< AlgorithmBase > addAlgorit

```

这个语句，定义了一个 `addWordConstructorToTable` 的对象，仅此而已。但是，注意在创建一个类的对象的时候，是要调用该类的构造函数的。回头看 `addWordConstructorToTable` 类

的构造函数，有意思的地方出现了。这个类的构造函数中，首先调用了

`constructWordConstructorTables` 函数，即对指针 `WordConstructorTablePtr_` 进行了初始化。然后，对 `WordConstructorTablePtr_` 进行 `insert` 操作，即，往其指向的 `hashTable` 插入 `key-value` 对。这里的 `key` 是创建对象 `addAlgorithmBaseWordConstructorToAlgorithmBaseTable_` 时代入的模板参数对应的类的 `typeName`（这一句很长很绕，需要好好理解，因为很重要！），`value` 则是 `New` 函数。这个 `New` 函数，指的是定义在 `addWordConstructorToTable` 中的 `New` 函数。这个 `New` 函数非常重要，再写一遍：

```

1  static autoPtr< AlgorithmBase > New ( const word& algorithmName )
2  {
3      return autoPtr< AlgorithmBase >(new AlgorithmBaseType (algorithmName))
4  }
```

这个 `New` 函数，返回的是一个 `AlgorithmBaseType`（这里是 `AlgorithmBase`）类型的临时对象的指针！对应这里的情形，现在可以知道这个 `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 `WordConstructorTablePtr_` 中（看来 `addToRunTimeSelectionTable` 中创建一个 `addWordConstructorToTable` 类的对象，居然目的是为了调用其构造函数。）。如果 `insert` 操作失败（原因是想要插入的 `key` 与 `hashTable` 已有的重复了，所以每一个类都需要不同的 `typeName`！），就会报条目重复的错。

好了，看完了基类相关的，在往下看派生类。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 `AlgorithmNew`，我们来看其具体的调用语句是

```

1  addToRunTimeSelectionTable<AlgorithmBase>(AlgorithmNew, Word);
```

展开的结果应该是

```

1  AlgorithmBase::addWordConstructorToTable<AlgorithmNew> addAlgorithmNew
```

注意，这里又创建了一个 `addWordConstructorToTable` 类的对象，只是这里代入的模板参数是 `AlgorithmNew`。于是，调用类的构造函数时代入的模板参数也就变了，所以这时 `New` 函数返回的将是 `AlgorithmNew` 类的临时对象的指针。并且，`AlgorithmNew` 这个名字与其对应的 `New` 函数组成的映射对，也被 `insert` 到 `WordConstructorTablePtr_` 里面。

而 `AlgorithmAdditional` 这个类，虽然是继承自 `AlgorithmNew`，但是也是间接继承 `AlgorithmBase`。并且，在 `AlgorithmAdditional` 类的类体之后调用的宏函数 `addToRunTimeSelectionTable(AlgorithmBase, AlgorithmAdditional, Word)`，依然是将构建的映射对添加到了同一个 `hashTable` 里。

最后，再来看一下 `selector`，即基类中定义的 `New` 函数。这个函数的返回值类型为 `autoPtr<AlgorithmBase>`，参数为跟类的 `typeName` 一样，都是 `word&`。这个函数里面，首先定义了一个 `hashTable` 的迭代器 `cstrIter`，利用迭代器来遍历搜索，看 `WordConstructorTable` 里面是否能找到参数 `algorithmName` 相符的 `key` 值，如果找不到，那就报错退出，并输出当前的 `WordConstructorTable` 中可选的项的名称（即 `WordConstructorTablePtr_->sortedToc()`）；如果找到了，那就返回这个 `key` 对应的 `value`。而 `WordConstructorTable` 的 `value` 是一个函数指针，所以 `cstrIter()` 返回的是 `algorithmName` 对应的那个 `New` 函数（不要跟基类 `AlgorithmBase` 中作为 `selector` 的 `New` 函数搞混了！）。进一步看，`cstrIter()(algorithmName)` 则表示的是函数调用了，传给函数的参数正是 `algorithmName`！

所以，`cstrIter()(algorithmName)` 返回的是 `autoPtr<AlgorithmBase>`，其指向的是 `typeName = algorithmName` 的类的对象！

这样就实现了 `New` 函数作为 `selector` 的功能！

所以，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`）。

经过以上四步，就实现了 RTS 机制。

参考：[source flux 的系列博文](#)

#Code Explained #RTS

 Share

NEWER

湍流模型中的 RTS 机制分析

OLDER

为什么要将声明和定义分离

CATEGORIES

[C++](#) (2)
[DEM](#) (1)
[OpenFOAM](#) (44)
[Paraview](#) (5)
[swak4Foam](#) (1)
[test](#) (2)
[vim](#) (1)

TAGS

[Boundary conditions](#) (6)
[C++](#) (2)
[CentOS](#) (1)
[Code Explained](#) (29)
[LES](#) (1)
[LIGGGHTS](#) (1)
[ODE](#) (1)
[OpenFOAM](#) (20)
[Postprocessing](#) (9)
[Preprocessing](#) (2)
[RTS](#) (3)
[TIL](#) (1)
[Windows](#) (1)
[fvOptions](#) (2)
[groovyBC](#) (1)
[paraview](#) (1)
[test](#) (2)
[thermophysicalModels](#) (5)
[turbulence model](#) (7)
[vim](#) (1)
[wall functions](#) (4)

TAG CLOUD

[Boundary conditions](#) [C++](#) [CentOS](#) [Code Explained](#) [LES](#) [LIGGGHTS](#) [ODE](#) [OpenFOAM](#) [Postprocessing](#) [Preprocessing](#) [RTS](#) [TIL](#) [Windows](#) [fvOptions](#) [groovyBC](#) [paraview](#) [test](#) [thermophysicalModels](#) [turbulence model](#) [vim](#) [wall functions](#)

ARCHIVES

[五月 2017](#) (1)
[八月 2016](#) (8)
[六月 2016](#) (5)

[五月 2016](#) (3)
[四月 2016](#) (12)
[三月 2016](#) (6)
[十二月 2015](#) (1)
[十一月 2015](#) (2)
[十月 2015](#) (1)
[九月 2015](#) (6)
[八月 2015](#) (1)
[六月 2015](#) (2)
[五月 2015](#) (6)
[四月 2015](#) (2)

RECENTS

[多说评论系统将停止提供服务](#)
[Paraview 脚本一例](#)
[Paraview 中有关 Camera 的操作两例](#)
[Paraview 中创建 Custom Filter](#)
[在 Paraview 中画截面上的流线](#)

© 2017 Giskard Q.

Powered by [Hexo](#)