# OpenFOAM 中的对象注册机制

2020-01-30 OpenFOAM 2741 words 6 mins read 12876 times read

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。

在这篇文章中，我们将会依次回答以下几个问题：

- 什么是树状结构?
- 树状结构如何管理?
- 如何读写注册对象?

## 树状结构

树状结构是一种常见的数据结构，通常由一个根节点，若干个子节点构成。若节点无父节点，则该节点为根节点。非根节点分为两类：一类是包含子节点的，称为分支节点；另一类是不包含子节点的，称为叶节点。

对象注册中树状结构用 `objectRegistry` 和 `regIOobject` 这两个类的派生类来描述节点。其中根节点和分支节点用 `objectRegistry` 的派生类描述，叶节点用 `regIOobject` 的派生类描述。其中， `objectRegistry` 派生自 `regIOobject` 类，增加了对象管理功能。

一个典型的树状结构如下所示（以 `icoFoam` 为例）：

```
runTime                    <-- Time         (objectRegistry)
└── region0                  <-- fvMesh           (objectRegistry)
    ├── fvSchemes              <-- IOdictionary         (regIOobject)
    ├── fvSolution             <-- IOdictionary         (regIOobject)
    ├── transportProperties    <-- IOdictionary         (regIOobject)
    ├── data                   <-- data                 (regIOobject)
    ├── solutionControl        <-- pisoControl          (regIOobject)
    ├── points                 <-- pointIOField         (regIOobject)
    ├── faces                  <-- faceCompactList      (regIOobject)
    ├── owner                  <-- labelIOList          (regIOobject)
    ├── neighbour              <-- labelIOList          (regIOobject)
    ├── boundary               <-- polyBoundaryMesh     (regIOobject)
    ├── pointZones             <-- pointZoneMesh        (regIOobject)
    ├── faceZones              <-- faceZoneMesh         (regIOobject)
    ├── cellZones              <-- cellZoneMesh         (regIOobject)
    ├── U                      <-- volVectorField       (regIOobject)
    ├── phi                    <-- surfaceScalarField   (regIOobject)
    └── p                      <-- volScalarField       (regIOobject)
```

除了 `Time` 和 `fvMesh` 是派生自 `objectRegistry` 外，其他都是派生自 `regIOobject` 。

此外还有一个 `IOobject` 类，这三个类是初学者最容易混淆的。下面分别介绍这三个类。

### `IOobject` 类

`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。部分主要属性见下表：

| 属性 | 类型 | 默认值 | 说明 |
| --- | --- | --- | --- |
| `name_` | `word` | 无 | 对象的名字 |
| `instance_` | `fileName` | 无 | 对象的路径 |
| `local_` | `fileName` | 无 | 对象输出路径下的子路径 |
| `db_` | `objectRegistry` | 无 | 对象的父对象 |
| `rOpt_` | `IOobject::readOption` | `NO_READ` | 读取方式 |
| `wOpt` | `IOobject::writeOption` | `NO_WRITE` | 写入方式 |
| `registerObject_` | `bool` | `true` | 是否注册到父对象 |

`IOobject` 通常不单独使用，而是定义后立即作为参数传递给 `regIOobject` 对象的定义。例如：

```
1  IOdictionary transportProperties
2  (
3      IOobject
4      (
5          "transportProperties",
```

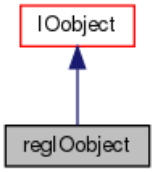
```
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);
```

以上这段代码定义了一个 `IOdictionary` （派生自 `regIOobject` ）对象：

- 对象名字为 *transportProperties*，可以用名字作为关键字查找返回该对象；
- 对象路径为 *constant* ，表示从 *constant* 目录读取或者往 *constant* 目录写入；
- 对象的父对象为 *mesh*，表示注册为 *mesh* 的一个子节点，拓扑结构可参考上面给出的例子；
- 读选项为 修改后重新读入（*MUST_READ_IF_MODIFIED*），写选项为 不写 （*NO_WRITE*）。

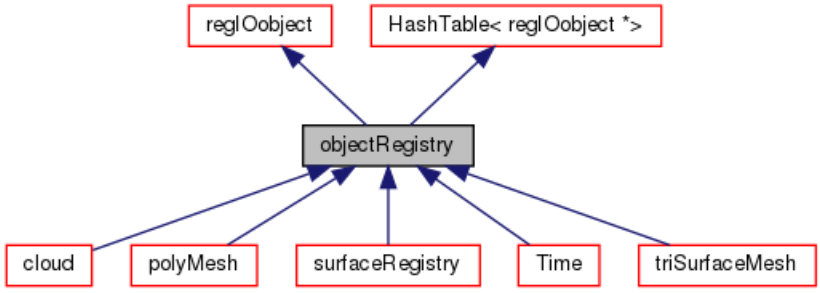
### `regIOobject` 类

`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。




### `objectRegistry` 类

`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。

## 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：

```
Foam::regIOobject::regIOobject(const IOobject& io, const bool isTime)
:
    IOobject(io),
    registered_(false),
    ownedByRegistry_(false),
    watchIndices_(),
    eventNo_                // Do not get event for top level Time database
    (
        isTime
      ? 0
      : db().getEvent()
    )
{
    // Register with objectRegistry if requested
    if (registerObject())
    {
        checkIn();
    }
}
```

若 `registerObject_` 为 `true` ，则执行 `checkIn` ，增加节点。

### 删除节点

删除节点的操作通过 `checkOut` 实现，这个操作用到的地方较少。

## 树状结构的读写

### 从磁盘读取

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。

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：

```
Foam::fvSchemes::fvSchemes(const objectRegistry& obr)
:
    IOdictionary
    (
        IOobject
        (
            "fvSchemes",
            obr.time().system(),
            obr,
            (
                obr.readOpt() == IOobject::MUST_READ
             || obr.readOpt() == IOobject::READ_IF_PRESENT
              ? IOobject::MUST_READ_IF_MODIFIED
              : obr.readOpt()
            ),
            IOobject::NO_WRITE
        )
    ),
        // ...
{
    if
    (
        readOpt() == IOobject::MUST_READ
     || readOpt() == IOobject::MUST_READ_IF_MODIFIED
     || (readOpt() == IOobject::READ_IF_PRESENT && headerOk())
    )
    {
        read(schemesDict());
    }
}
```

第二种情况较为复杂。每次执行 `Time::run` 时会调用 `Time::readModifiedObjects` 函数：

```
bool Foam::Time::run() const
{
        // ...
    if (running)
    {
        if (!subCycling_)
        {
            const_cast<Time&>(*this).readModifiedObjects();
                        // ...
        }

        // Re-evaluate if running in case a function object has changed things
        running = this->running();
    }

    return running;
}
```

`Time::readModifiedObjects` 先读取 *system/controlDict*，然后调用 `objectRegistry::readModifiedObjects` 函数：

```
void Foam::Time::readModifiedObjects()
{
     if (runTimeModifiable_)
     {
                // ...

        // Time handling is special since controlDict_ is the one dictionary
        // that is not registered to any database.
```

```
        if (controlDict_.readIfModified())
        {
            readDict();
            functionObjects_.read();

            if (runTimeModifiable_)
            {
                // For IOdictionary the call to regIOobject::read() would have
                // already updated all the watchIndices via the addWatch but
                // controlDict_ is an unwatchedIOdictionary so will only have
                // stored the dependencies as files.

                fileHandler().addWatches(controlDict_, controlDict_.files());
            }
            controlDict_.files().clear();
        }

        bool registryModified = objectRegistry::modified();

        if (registryModified)
        {
            objectRegistry::readModifiedObjects();
        }
    }
}
```

`objectRegistry::readModifiedObjects` 递归遍历树状结构，并调用每个节点的 `readIfModified` 函数：

```
void Foam::objectRegistry::readModifiedObjects()
{
    for (iterator iter = begin(); iter != end(); ++iter)
    {
        if (objectRegistry::debug)
        {
            Pout<< "objectRegistry::readModifiedObjects() : "
                << name() << " : Considering reading object "
                << iter.key() << endl;
        }

        iter()->readIfModified();
    }
}
```

同样以 `fvSchemes` 类为例，该类没有重写 `readIfModified` 方法，则调用的实际是 `regIOobject` 的 `readIfModified` 方法，实际调用自身的 `read` 方法：

```
bool Foam::regIOobject::readIfModified()
{
    // ...
    if (modified != -1)
    {
        // ...
        return read();
    }
    else
    {
        return false;
    }
}
```

而 `fvSchemes` 的 `read` 方法被重写，具体如下：

```
bool Foam::fvSchemes::read()
{
    if (regIOobject::read())
    {
        // Clear current settings except fluxRequired
        clear();

        read(schemesDict());

        return true;
    }
    else
    {
        return false;
    }
}
```

## 写入到磁盘

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：

```
bool Foam::Time::writeObject
(
    IOstream::streamFormat fmt,
    IOstream::versionNumber ver,
    IOstream::compressionType cmp,
    const bool write
) const
{
    if (writeTime())
    {
        bool writeOK = writeTimeDict();

        if (writeOK)
        {
            writeOK = objectRegistry::writeObject(fmt, ver, cmp, write);
        }

        if (writeOK)
        {
            // Does the writeTime trigger purging?
            if (writeTime_ && purgeWrite_)
            {
                if
                (
                    previousWriteTimes_.size() == 0
                 || previousWriteTimes_.top() != timeName()
                )
                {
                    previousWriteTimes_.push(timeName());
                }

                while (previousWriteTimes_.size() > purgeWrite_)
                {
                    fileHandler().rmDir
                    (
                        fileHandler().filePath
                        (
                            objectRegistry::path(previousWriteTimes_.pop())
                        )
                    );
                }
            }
        }

        return writeOK;
    }
    else
    {
        return false;
    }
}
```

- 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；
- 调用 `writeTimeDict` 函数，往 *[time]/uniform/time* 文件中写入和时间相关的变量；

```
bool Foam::Time::writeTimeDict() const
{
    const word tmName(timeName());

    IOdictionary timeDict
    (
        IOobject
        (
            "time",
            tmName,
            "uniform",
            *this,
            IOobject::NO_READ,
            IOobject::NO_WRITE,
            false
        )
    );

```

```
    timeDict.add("value", timeName(timeToUserTime(value()), maxPrecision_));
    timeDict.add("name", string(tmName));
    timeDict.add("index", timeIndex_);
    timeDict.add("deltaT", timeToUserTime(deltaT_));
    timeDict.add("deltaT0", timeToUserTime(deltaT0_));

    return timeDict.regIOobject::writeObject
    (
        IOstream::ASCII,
        IOstream::currentVersion,
        IOstream::UNCOMPRESSED,
        true
    );
}
```

- 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；

```
bool Foam::objectRegistry::writeObject
(
    IOstream::streamFormat fmt,
    IOstream::versionNumber ver,
    IOstream::compressionType cmp,
    const bool write
) const
{
    bool ok = true;

    forAllConstIter(HashTable<regIOobject*>, *this, iter)
    {
        if (objectRegistry::debug)
        {
            Pout<< "objectRegistry::write() : "
                << name() << " : Considering writing object "
                << iter.key()
                << " of type " << iter()->type()
                << " with writeOpt " << iter()->writeOpt()
                << " to file " << iter()->objectPath()
                << endl;
        }

        if (iter()->writeOpt() != NO_WRITE)
        {
            ok = iter()->writeObject(fmt, ver, cmp, write) && ok;
        }
    }

    return ok;
}
```

  - 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；

```
bool Foam::fvMesh::writeObject
(
    IOstream::streamFormat fmt,
    IOstream::versionNumber ver,
    IOstream::compressionType cmp,
    const bool write
) const
{
    bool ok = true;
    if (phiPtr_)
    {
        ok = phiPtr_->write(write);
    }

    // Write V0 only if V00 exists
    if (V00Ptr_)
    {
        ok = ok && V0Ptr_->write(write);
    }

    return ok && polyMesh::writeObject(fmt, ver, cmp, write);
}
```

- 最后是 purgeWrite 的相关操作。

CONTENTS