3 开发者的Catalyst

李健

编写适配器代码的开发者应该了解模型代码的数据结构、VTK数据结构和Catalyst API。

High-Level 视角

连接Catalyst与模型代码需要大量的编程，对代码的影响很小。大多数情况，仅需编写3个函数，在模型代码中调用：

1. Initialize

初始化Catalyst. 依赖MPI的代码，一般在调用MPI_Init()之后初始化。在adaptor中实施初始化方法。

2 CoProcess

需要提供grid和field数据结构给adaptor，以及时间和时间步长信息。还可以提供额外的不必要的控制信息。通常在模拟代码的每个时间步更新结束后调用CoProcess，即：在已经更新fields到新时间步，或者已修改了grid。

3 Finalize

完成模拟后，代码必须调用Catalyst，结束任何state，合适地清理。依赖MPI的代码，需要再调用MPI_Finalize()之前调用。通常在adaptor中实施初始化方法。

High-Level 视角

示例代码：

```
MPI_Init(argc, argv);
#ifdef CATALYST
CatalystInit(argc, argv);
#endif
for(int timeStep=0;timeStep<numberOfTimeSteps;timeStep++)
  {
  <update grids and fields to timeStep>
#ifdef CATALYST
  CatalystCoProcess(timeStep, time, <grid info>, <field info>);
#endif
  }
#ifdef CATALYST
CatalystFinalize();
#endif
MPI_Finalize();
```

High-Level 视角

适配器代码在单独的文件中实施。适配器代码负责模型代码与Catalyst之间的接口。除了初始化和结束Catalyst外，适配器代码的另一作用是：

（1）查询Catalyst，看是否需要实施co-processing；

（2）提供调整用于co-processing的网格和场的VTK数据对象。

下面的伪代码显示了适配器的大致内容：

```
void CatalystCoProcess(int timeStep, double time, <grid info>,
                       <field_info>)
{
1. Specify current timeStep and time for Catalyst
2. Check with Catalyst if anything needs to be done this call
3. If nothing needs to be done this call, return
4. Create VTK grid
5. Create VTK fields and associate with VTK grid
6. Specify VTK grid for Catalyst
7. Call Catalyst to perform co-processing (i.e. execute VTK
   pipelines)
}
```

High-Level 视角

下面的示例代码显示了简化版的适配器。

```
// static data
vtkCPProcessor* Processor = NULL;

void CatalystInit(int numScripts, char* scripts[])
{
  if(Processor == NULL)
    {
    Processor = vtkCPProcessor::New();
    Processor->Initialize();
    }
  // scripts are passed in as command line arguments
  for(int i=0;i<numScripts;i++)
    {
    vtkCPPythonScriptPipeline* pipeline =
      vtkCPPythonScriptPipeline::New();
    pipeline->Initialize(scripts[i]);
    Processor->AddPipeline(pipeline);
    pipeline->Delete();
    }
}

void CatalystFinalize()
{
  if(Processor)
    {
    Processor->Delete();
    Processor = NULL;
    }
}

// The grid is a uniform, rectilinear grid that can be specified
// with the number of points in each direction and the uniform
// spacing between points. There is only one field called
// temperature which is specified over the points/nodes of the
// grid.
void CatalystCoProcess(
  int timeStep, double time, unsigned int numPoints[3],
  double spacing[3], double* field)
{
  vtkCPDataDescription* dataDescription =
    vtkCPDataDescription::New();
  dataDescription->AddInput("input");
  dataDescription->SetTimeData(time, timeStep);
  if(Processor->RequestDataDescription(
     dataDescription) != 0)
    {
    // Catalyst needs to output data
    // Create a uniform grid
    vtkImageData* grid = vtkImageData::New();
    grid->SetExtents(0, numPoints[0]-1, 0, numPoints[1]-1,
                     0, numPoints[2]-1);
    dataDescription->GetInputDescriptionByName("input")
      ->SetGrid(grid);
    grid->Delete();
    // Create a field associated with points
    vtkDoubleArray* array = vtkDoubleArray::New();
    array->SetName("temperature");
    array->SetArray(field, grid->GetNumberOfPoints(), 1);
    grid->GetPointData()->AddArray(array);
    array->Delete();
    Processor->CoProcess(dataDescription);
    }
  dataDescription->New();
}
```

Overview

在了解VTK和Catalyst API（在编写适配器代码时需要）的细节之前，应该了解一些细节信息：

（1）VTK编号从0开始；

（2）vtkIdType是在Catalyst配置时设置的一个整型类型；

（3）VTK手册；

（4）Paraview使用手册。

Overview

VTK是一种普适性工具，有很多种表征网格的方法。VTK需要普适性的原因就是：能够处理拓扑复杂的非结构网格，也能处理简单网格（如均匀的笛卡尔网格等）。以下是VTK支持的网格类型：

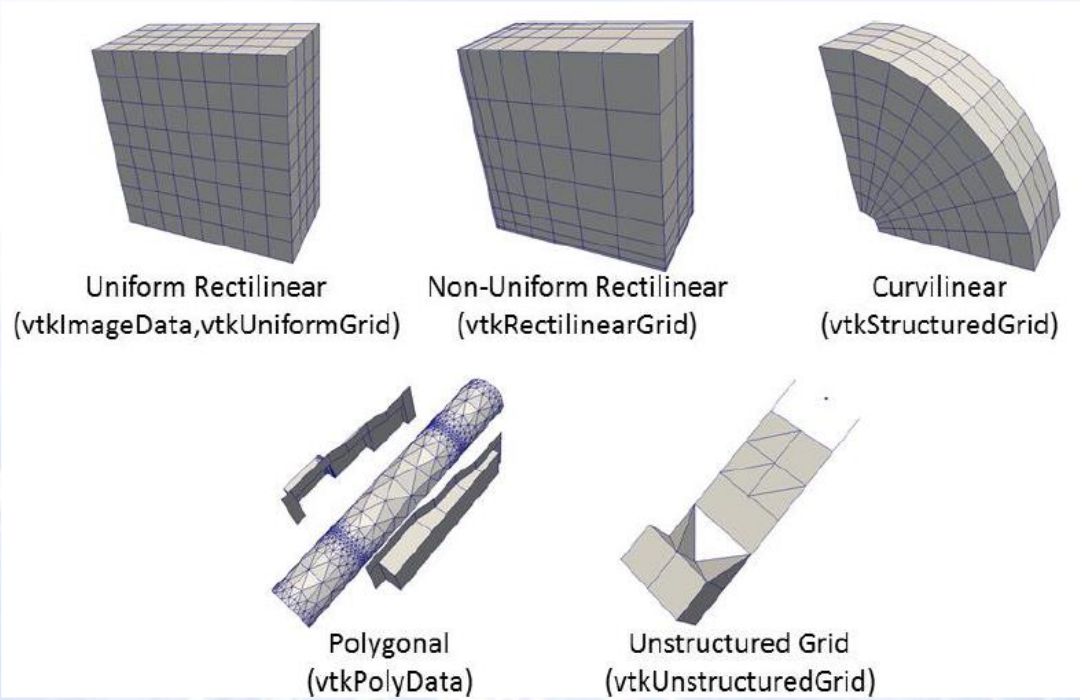

图3.1 VTK数据集类型

Overview

此外，VTK还支持很多2D/3D单元类型，如三角形、四面性、四面体、金字塔、棱柱、六面体等。VTK还支持数据集中各节点或单元上的相关场信息。VTK中，一般称为**属性数据(attribute data)**，当与数据集中的节点或单元相关时，称之为点数据(point data)或单元数据(cell data)。

VTK数据集的整体结构包含：网格信息，与网格上各节点和单元相关信息的数组。如下图：

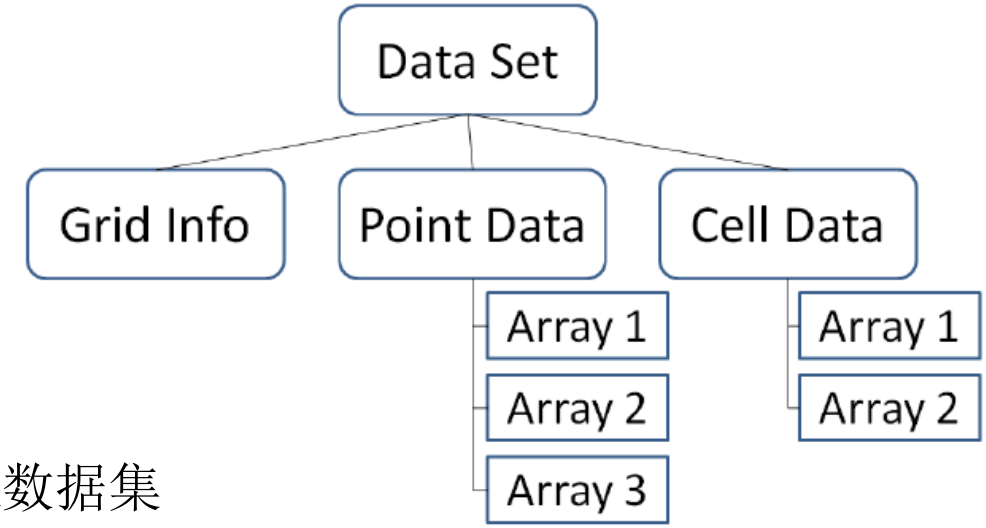

图3.2 VTK数据集

VTK Data Object API

另一个单独的PPT课件！

VTK使用管线架构来处理数据。

编写适配器代码需要了解管线架构。

Grid Partitioning

目的：适应已有的模拟网格区域分解。

假设：大多数过滤，对模拟提供的已有网格分区有很好的尺度化(scale well)。

VTK数据集和复合数据集要移动网格数据，为了合适地创建分区的VTK网格数据。

VTK默认使用cell-based的网格分区，一个单元唯一地在一个进程上表征。

结构网格分区通过范围(extent)完成，有2类：

vtkDataSet和vtkMultiBlockDataSet

Grid Partitioning

多块(Multi-block)数据集，每个进程一个块的例子如下，其中numberOfProcesses是MPI进程数, rank是一个进程的MPI进程编号：

```
vtkMultiBlockDataSet* multiBlock = vtkMultiBlockDataSet::New();
multiBlock->SetNumberOfBlocks(numberOfProcesses);
multiBlock->SetBlock(rank, dataSet);
```

Grid Partitioning

非结构网格，vtkPolyData 和 vtkUnstructuredGrid,

- 当使用cell-based分区时，直接实施区域分解。不添加Ghost cells到VTK网格。
- 对point-based的网格分区，通常在多个进程上存在的单元层有重叠(overlap)。这些单元需要唯一地分配到VTK网格的单个进程和分区。
- 与规则网格类似，vtkPolyData 和 vtkUnstructuredGrid数据集也可以插入到一个vtkMultiBlockDataSet，允许各进程上有多个块。

VTK管线

关于VTK管线架构的完整介绍参考VTK的用户手册。

本质上，创建从Catalyst的输出就是创建VTK管线，在模型运行期间，在某点上执行VTK管线。

VTK使用数据流的方法将信息转换为需要的形式。

需要的形式可能是：衍生计算量、提取变量或者图形信息。

VTK中的过滤器实施转换。这些过滤器(filter)读取数据，基于输入过滤器的一套参数执行操作。大多数过滤器执行具体的操作，但将多个过滤器串接起来，可完成多种数据转换操作。

VTK管线

只有来自其他过滤器的输入的过滤器，称为源；

不发送任何输出给其他过滤器的过滤器，称为汇。

源过滤器可能是一个文件读取程序，汇过滤器可能是一个文件写出程序。称这样的一套连接的过滤器为管线。对于Catalyst，适配器起到所有管线的源过滤器的作用。如下图所示：

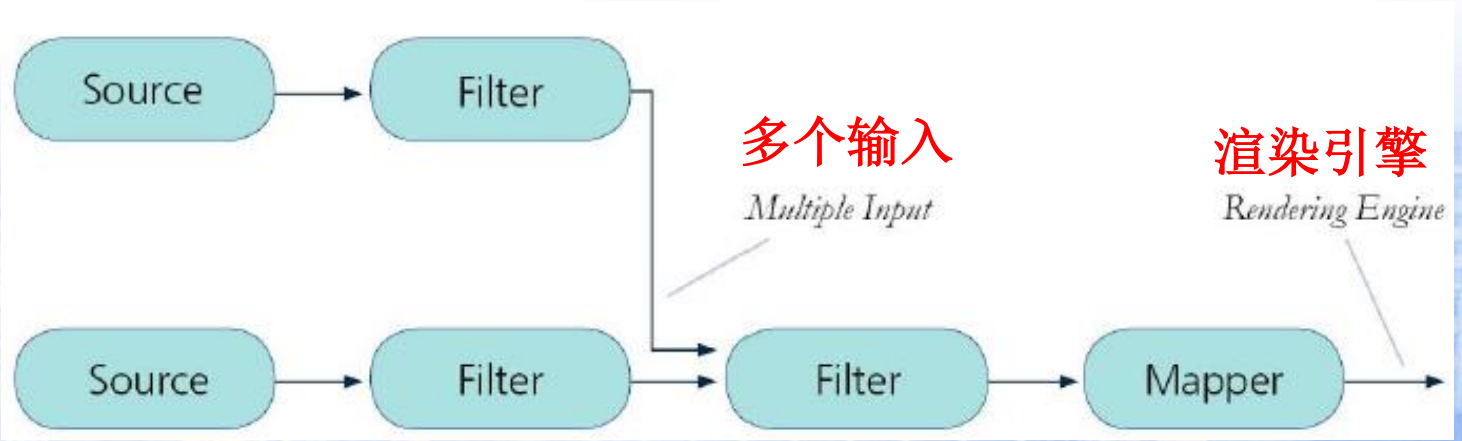

图3.5 过滤器链条

VTK管线

- 管线作用：在过滤器之间配置、执行和传递vtkDataObject。
- 管线可视为有方向的、不循环的图(DAG)。
- VTK的管线的一些重要特性：

（1）不允许过滤器修改他们的输入数据对象；

（2）当下游过滤器要求他们执行的时候，该过滤器才执行；

（3）如果请求改变或上游过滤器变化，该过滤器才重新执行；

（4）过滤器可以有多个输入和输出；

（5）过滤器可以向多个单独的过滤器发送他们的输出。

VTK管线

这将从几个方面影响Catalyst：

一、当构建VTK数据结构时，适配器可使用当前内存。原因是：任何操作数据的VTK过滤，或者创建新的复制（如果需要修改数据），或者如果无需修改数据，则通过reference counting，再利用已有数据。

二、当具体要求管线执行时，该管线才重新执行。

Catalyst API

另一个单独的PPT课件！

介绍适配器如何将信息在模型代码与Catalyst之间传递信息。

与C或FORTRAN的模型代码连接

Catalyst主要是包含很多类的C++库，用Python做了封装。这使得Catalyst连接C++和Python的模拟代码就很容易。很多模型代码是C或FORTRAN开发的，需要增加一些C++代码，来创建VTK数据对象。

在C++函数声明前，增加extern “C”，被FORTRAN或C代码调用。对于头文件，按照以下编写：

```
#ifdef __cplusplus
extern "C"
{
#endif
  void CatalystInitialize(int numScripts, char* scripts[]);
  void CatalystFinalize();
#ifdef __cplusplus
}
#endif
```

与C或FORTRAN的模型代码连接

对于FORTRAN代码，所有的数据对象都以指针传递。对于C代码，如果不用Python，添加合适的头文件CAdaptorAPI.h，主要定义的函数有：

- void coprocessorinitialize() -- initialize Catalyst.
- void coprocessorfinalize() -- finalize Catalyst.
- void requestdatadescription(int* timeStep, double* time, int* coprocessThisTimeStep) -- check the current pipelines to see if any of them need to execute for the given time and time step. The return value is in coprocessThisTimeStep and is 1 if co-processing needs to be performed and 0 otherwise.
- void coprocess() -- execute the Catalyst pipelines for the timeStep and time specified in requestdatadescription(). Note that the adaptor must update the grid and attribute information and set them in the proper vtkCPInputDataDescription object obtained through vtkCPAdaptorAPI::GetCoProcessorData() method.

编译和链接源代码

最后一步就是：编译所有的部件，并将他们链接起来。

最简单的编译方法就是使用CMAKE，编写CMakeLists.txt，示例如下：

```
cmake_minimum_required(VERSION 2.8.8)
project(CatalystCxxFullExample)

set(USE_CATALYST ON CACHE BOOL
    "Link the simulator with Catalyst")
if(USE_CATALYST)
  find_package(ParaView 3.98 REQUIRED COMPONENTS
               vtkPVPythonCatalyst)
  include("${PARAVIEW_USE_FILE}")
  set(Adaptor_SRCS FEAdaptor.cxx)
  add_library(Adaptor ${Adaptor_SRCS})
  target_link_libraries(Adaptor vtkPVPythonCatalyst)
  add_definitions("-DUSE_CATALYST")
else()
  find_package(MPI REQUIRED)
  include_directories(${MPI_CXX_INCLUDE_PATH})
endif()

add_executable(FEDriver FEDriver.cxx FEDataStructures.cxx)
if(USE_CATALYST)
  target_link_libraries(FEDriver Adaptor)
else()
  target_link_libraries(FEDriver ${MPI_LIBRARIES})
endif()
```

编译和链接源代码

使用Catalyst的参数：USE_CATALYST，编译Paraview。此时，将在FEDriver模拟代码示例，安置对Adaptor的依赖库。

如果模拟代码不需要Python接口到Catalyst，用户可避免Python依赖，做法是：

改变Paraview组件，从vtkPVPythonCatalyst变为vtkPVCatalyst。

这些组件也会引入其他的Catalyst组件和头文件，用于编译和链接。

如果不用CMAKE编译，那就要找一个示例，确定需要的用于编译的头文件路径和需要链接的库。

参考文献

Data Co-Processing for Extreme Scale Analysis Level II ASC Milestone. David Rogers, Kenneth Moreland, Ron Oldfield, and Nathan Fabian. Tech Report SAND 2013-1122, Sandia National Laboratories, March 2013.

The ParaView Coprocessing Library: A Scalable, General Purpose *In Situ* Visualization Library Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C. Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E. Jansen. In *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, October 2011, pp. 89-96.