SiPESC

基于 SIPESC 平台的单元插件开发指南

SiPESC Group

Dec. 2014

目录

基于 SiPESC 平台的单元插件开发指南

1. 总述。

1.1 准备。

熟悉有限单元法基本理论，熟悉 C++编程，熟悉平台插件设计的基本方法（可参见《HelloWorld 插件设计向导》）。

1.2 SiPESC 平台插件及开放式有限元系统简述。

1.2.1 SiPESC 平台插件。

SiPESC 软件平台基于“微核心（平台）+插件”的设计思想，广泛使用插件技术。插件是一种特殊的动态链接库。插件的本质在于不修改程序主体（平台）的情况下对软件功能进行扩展与加强，当插件的接口公开后，开发人员即制作插件来扩展新的功能，也就是实现真正意义上的“即插即用”软件开发。添加一个新的单元插件就是这种开放模式的体现。

为了实现平台+插件结构的软件设计需要定义两个标准接口，一个为由平台所实现的平台扩展接口，一个为插件所实现的插件接口。这里需要说明的是：平台扩展接口完全由平台实现，插件只是调用和使用，插件接口完全由插件实现，平台也只是调用和使用。平台扩展接口实现插件向平台方向的单向通信，插件通过平台扩展接口可获取主框架的各种资源和数据，可包括各种系统句柄，程序内部数据以及内存分配等。插件接口为平台向插件方向的单向通信，平台通过插件接口调用插件所实现的功能，读取插件处理数据等。

1.2.2 开放式有限元系统。

SiPESC.FEMS 开放式结构有限元系统结合有限元分析问题，设计了几十多种数据类，分别为节点数据、单元数据、材料数据、坐标转换阵、约束数据、节点控制阵、单元控制阵、载荷等。具备了结构有限元线弹性静力动力分析功能，提供节点排序、自由度排序、节点主从控制、单元主从控制、单元刚度阵管理、总刚集成与分解、位移求解等模块。提供各级计算模块的开发接口。

开放式结构有限元系统分为两个部分：结构有限元工程数据库和结构有限元分析系统。

工程数据库方面本文档只在用到时做简要说明，不再专门介绍（若想系统了解，详见陆旭泽《*SiPESC* 工程数据库从入门到精通》系列 ppt）。

结构有限元分析系统的设计实现是基于工程数据库、算法库、插件管理系统库及 Qt 库实现的。在各 C++的项目开发的 CMakeList.txt 中，需包括相关开发包，头文件和链接库的引用语句(前提是需安装 Qt 及 SiPESC 系统)。

结构有限元分析系统的所有动态扩展功能是基于插件管理系统，有两种方式调用插件中的扩展：

（1）指定插件名称，如有限元分析的总流程可能是：节点排序、自由度排序、节点控制阵计算、单元控制阵计算、单元刚度计算、总刚集成等，这样就可以按照流程依次按指定名称调用对应扩展即可；

（2）通过工厂机制，及每一个扩展都有对应的一个工厂对象，该工厂对象负责提供扩展的标识，支持级别以及创建扩展方法，外部调用模块根据支持级别决定是否调用该扩展，这种方式具备更好的可扩展能力；如在单元刚度计算中，若想动态替换已有的某类型单元，可以通过新建工厂对象，并提供对应单元类型并设置更高级别的支持类型，则外部调用模块可以动态调用新的单元计算扩展；实现具体工厂机制时，必须确定工厂对象所属的类别，才能保证正确应用。

1.3 单元插件的功能描述以及本文档主要内容。

1.3.1 单元插件的功能描述

执行有限元分析流程是在主程序运行时完成的。主程序先读入数据文件，通过各种任务类根据单元的关键字在已经加载的单元插件中搜索，找到相应的插件，而后调用之进行相关有限元的计算。

一个单元插件为该种单元提供了从单元数据的定义与读取到单刚计算再到节点应力计算的各种各样的实现类，涵盖了对该种单元进行有限元分析所需要的所有计算信息。然而在单元插件开发过程中，对于一些相对简单的有限元分析，当中的许多流程（例如求解数值积分，单刚累总刚，有限元方程的求解等）是相对固定的，因此这些共性的求解流程并不需要单元插件的开发者来编写，平台会调用其他的扩展来完成；另外对于一些问题，平台已有扩展就能解决，这时开发者只要灵活调用平台已有扩展即可（例如有的单元形成单刚的积分格式平台里是已有的，就不必重新编写）。这不仅大大简化了开发者的代码编写，也让每段代码使用率更高，扩展性更强，这就是上文提到过的 SiPESC 特有的“核心+插件”的设计思想。

单元插件包括的内容也因此比想象中少的多。对于线性静力分析，其功能具体包括单元数据定义，单元数据读取与存储，形函数及其导数计算，单元应变阵计算，单元本构阵计算，积分格式的定义，单元刚度阵计算，积分点的应变、应力计算，节点应力计算。我们可以看出，单元刚度阵计算和为计算它而进行的诸多准备是单元插件的核心内容。

1.3.2 本文档主要内容

本文档将以六面体单元（静力分析）为例，在第二章对该问题进行有限元描述，并简要介绍它的单元刚度阵形成过程；在第三章结合六面体单元插件的示例代码和第二章的相关理论，通过对示例代码中关键实现类的剖析，总结单元插件设计过程中的各个要点。

2. 空间六面体单元刚度阵形成的简要理论说明。

2.1 空间六面体单元问题描述

该单元为由 8 个节点组成的六面体单元，每个节点有 3 个位移（即 3 个自由度），图 2-1 为该单元节点和位移的示意。

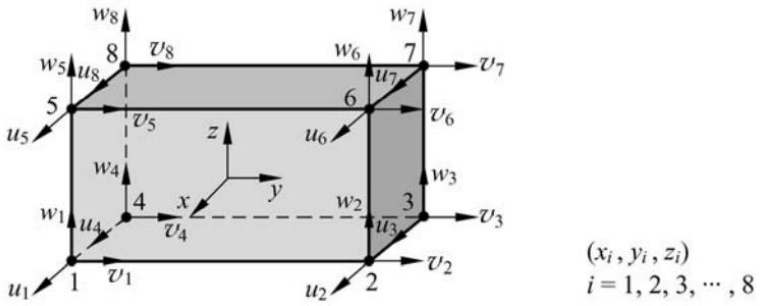


图 2-1

该单元的节点位移总共有 24 个自由度。单元的节点位移列阵 q^e 和节点力列阵 P^e 为

$$\underset{(24\times1)}{\mathbf{q}^e} = \begin{bmatrix} u_1 & v_1 & w_1 & \vdots & u_2 & v_2 & w_2 & \vdots & \cdots & \vdots & u_8 & v_8 & w_8 \end{bmatrix}^T \tag{2-1}$$

$$\underset{(24\times1)}{\mathbf{P}^e} = \begin{bmatrix} P_{x1} & P_{y1} & P_{z1} & \vdots & P_{x2} & P_{y2} & P_{z2} & \vdots & \cdots & \vdots & P_{x8} & P_{y8} & P_{z8} \end{bmatrix}^T$$

2.2 单元应变矩阵计算

2.2.1 单元插值函数

单元位移场用节点位移表示为

$$\underset{(3\times1)}{\boldsymbol{u}} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \begin{bmatrix} N_1 & 0 & 0 & \vdots & N_2 & 0 & 0 & \vdots & \cdots & \vdots & N_8 & 0 & 0 \\ 0 & N_1 & 0 & \vdots & 0 & N_2 & 0 & \vdots & \cdots & \vdots & 0 & N_8 & 0 \\ 0 & 0 & N_1 & \vdots & 0 & 0 & N_2 & \vdots & \cdots & \vdots & 0 & 0 & N_8 \end{bmatrix} \cdot \boldsymbol{q}^e = \underset{(3\times24)}{\boldsymbol{N}} \cdot \underset{(24\times1)}{\boldsymbol{q}^e} \tag{2-2}$$

引入参数坐标后形函数表示为

$$\begin{aligned} &\boldsymbol{N}_i = \frac{1}{8}(1+\xi_0)(1+\eta_0)(1+\zeta_0), \\ &\text{其中} \xi_0 = \xi_i\xi, \eta_0 = \eta_i\eta, \zeta_0 = \zeta_i\zeta. \\ &(\xi_i, \eta_i, \zeta_i \text{分别为i节点的参数坐标值}) \end{aligned} \tag{2-3}$$

2.2.2 等参坐标变换

按照偏微分规则，函数 N_i 对 ξ 的偏导数可表示为

$$\frac{\partial N_i}{\partial \xi}=\frac{\partial N_i}{\partial x}\frac{\partial x}{\partial \xi}+\frac{\partial N_i}{\partial y}\frac{\partial y}{\partial \xi}+\frac{\partial N_i}{\partial z}\frac{\partial z}{\partial \xi} \tag{2-4}$$

对于其他两个坐标 η,ζ ，可以写出类似的表达式。集合成矩阵形式，即

$$\begin{pmatrix}\frac{\partial N_i}{\partial \xi}\\ \frac{\partial N_i}{\partial \eta}\\ \frac{\partial N_i}{\partial \zeta}\end{pmatrix}=\begin{pmatrix}\frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi}\\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta}\\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta}\end{pmatrix}\begin{pmatrix}\frac{\partial N_i}{\partial x}\\ \frac{\partial N_i}{\partial y}\\ \frac{\partial N_i}{\partial z}\end{pmatrix}=\boldsymbol{J}\begin{pmatrix}\frac{\partial N_i}{\partial x}\\ \frac{\partial N_i}{\partial y}\\ \frac{\partial N_i}{\partial z}\end{pmatrix} \tag{2-5}$$

其中的雅克比矩阵可以显示的表达为

$$\boldsymbol{J}=\begin{pmatrix}\frac{\partial N_1^{'}}{\partial \xi} & \frac{\partial N_2^{'}}{\partial \xi} & \cdots & \frac{\partial N_8^{'}}{\partial \xi}\\ \frac{\partial N_1^{'}}{\partial \eta} & \frac{\partial N_2^{'}}{\partial \eta} & \cdots & \frac{\partial N_8^{'}}{\partial \eta}\\ \frac{\partial N_1^{'}}{\partial \zeta} & \frac{\partial N_2^{'}}{\partial \zeta} & \cdots & \frac{\partial N_8^{'}}{\partial \zeta}\end{pmatrix}\begin{pmatrix}x_1 & y_1 & z_1\\ x_2 & y_2 & z_3\\ \vdots & \vdots & \vdots\\ x_8 & y_8 & z_8\end{pmatrix} \tag{2-6}$$

这样 N_i 对 x,y,z 的偏导数可以用参数坐标显示的表示为

$$\begin{pmatrix}\frac{\partial N_i}{\partial x}\\ \frac{\partial N_i}{\partial y}\\ \frac{\partial N_i}{\partial \mathrm{z}}\end{pmatrix}=\boldsymbol{J}^{-1}\begin{pmatrix}\frac{\partial N_i}{\partial \xi}\\ \frac{\partial N_i}{\partial \eta}\\ \frac{\partial N_i}{\partial \zeta}\end{pmatrix} \tag{2-7}$$

除了导数之间的变换外，还有体积微元的变换，这里只给出表达式，其推导方法详见王勖成《有限单元法》第四章。

$$dV=\begin{vmatrix}\frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi}\\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta}\\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta}\end{vmatrix}d\xi d\eta d\zeta=|\boldsymbol{J}|d\xi d\eta d\zeta \tag{2-8}$$

2.2.3 计算单元应变阵

2.2.1 和 2.2.2 的内容都是为单元应变阵的计算做准备，下面来计算单元应变阵。

单元应变场用节点位移表示为

$$\underset{(6\times1)}{\boldsymbol{\varepsilon}} = \underset{(6\times3)}{[\partial]}\,\underset{(3\times1)}{\mathbf{u}} = \underset{(6\times3)}{[\partial]}\,\underset{(3\times24)}{\mathbf{N}}\,\underset{(24\times1)}{\mathbf{q}^e} = \underset{(6\times24)}{\mathbf{B}}\,\underset{(24\times1)}{\mathbf{q}^e} \tag{2-9}$$

应变矩阵表示为

$$\underset{(6\times24)}{\mathbf{B}} = \underset{(6\times3)}{[\partial]}\,\underset{(3\times24)}{\mathbf{N}} \tag{2-10}$$

其中

$$[\partial] = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \tag{2-11}$$

那么可将 **B** 阵写成分块形式

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{B}_1 & \boldsymbol{B}_2 & \cdots & \boldsymbol{B}_8 \end{bmatrix} \tag{2-12}$$

其中

$$\boldsymbol{B}_i = \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_i}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_i}{\partial z} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial z} & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} & 0 & \frac{\partial N_i}{\partial x} \end{bmatrix} \tag{2-13}$$

根据 2.2.2 中的导数变换推导，以及 2.2.1 中的插值函数，由于节点的总体坐标值是常量，我们最终把 **B** 阵表示成了参数坐标的函数组成的矩阵。（特别的，插值函数如果是一阶的（如三角型单元），应变阵将是常量矩阵）

2.3 单元本构矩阵计算

单元本构阵也叫单元弹性阵，反映的是应变和应力之间的关系。表达式为

$$\boldsymbol{\sigma} = \boldsymbol{D}\boldsymbol{\varepsilon} \tag{2-14}$$

$$D=\frac{E(1-\nu)}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ & & 1 & 0 & 0 & 0 \\ & 对 & & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ & & 称 & & \frac{1-2\nu}{2(1-\nu)} & 0 \\ & & & & & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{2-15}$$

它完全取决于弹性体材料的弹性模量 E 和泊松比ν。

2.4 数值积分与单元刚度阵计算

对于六面体单元，我们采取 3 维的 2 点高斯积分格式。积分点选作参数坐标系中的+0.577350269189626, 和-0.577350269189626，权系数都是 1.以上积分可表示为

$$\left.\begin{aligned} \xi_j=\eta_k=\zeta_w=\pm 0.577350269189626 \\ A_j=A_k=A_w=1 \end{aligned}\right\} \tag{2-16}$$

单元刚度阵为

$$\underset{(24\times24)}{K^e}=\iiint_{\Omega}\underset{(24\times6)}{B^T}\underset{(6\times6)}{D}\underset{(6\times24)}{B}\,d\Omega \tag{2-17}$$

写成参数坐标下的数值积分格式为

$$\underset{(24\times24)}{K^e}=\int_{-1}^{1}\int_{-1}^{1}\int_{-1}\boldsymbol{B}^{\boldsymbol{T}}\boldsymbol{D}\boldsymbol{B}\left|\boldsymbol{J}\right|d\xi d\eta d\zeta=\sum_{j=1}^{2}\sum_{k=1}^{2}\sum_{w=1}^{2}\left\{A_jA_kA_w\left[\underset{(24\times6)}{\boldsymbol{B}^{\boldsymbol{T}}}\underset{(6\times6)}{\boldsymbol{D}}\underset{(6\times24)}{\boldsymbol{B}}\right]\Big|_{(\xi_j,\eta_k,\zeta_w)}\cdot\left|\boldsymbol{J}(\xi_j,\eta_k,\zeta_w)\right|\right\} \tag{2-18}$$

我们可以看到，在 ***B*** 阵和雅克比矩阵 ***J*** 都表示成参数坐标的函数矩阵后，计算给定的参数坐标下高斯点上的单元刚度阵是非常简单而模式化的。

通过以上步骤，我们完成了对六面体单元单元刚度阵的计算。根据有限元方法的步骤，下面将进行总刚集成，LT 分解，有限元方程求解等步骤，由于这些一般不是单元插件实现的内容，不再论述。

3. SiPESC 平台单元插件 C++ project 的构建。

首先要使用插件设计器设计单元插件，请参照示例示例代码中的 doc 文件夹中的 org.sipesc.fems.example.sipplugin 文件。（注意：本文档仅以六面体单元静力有限元分析为例进行解释，示例代码中动力学计算相关部分不作说明）

1.使用插件设计器新建一个插件文件，填写插件名称，版本，描述等等信息，如图 3-1。

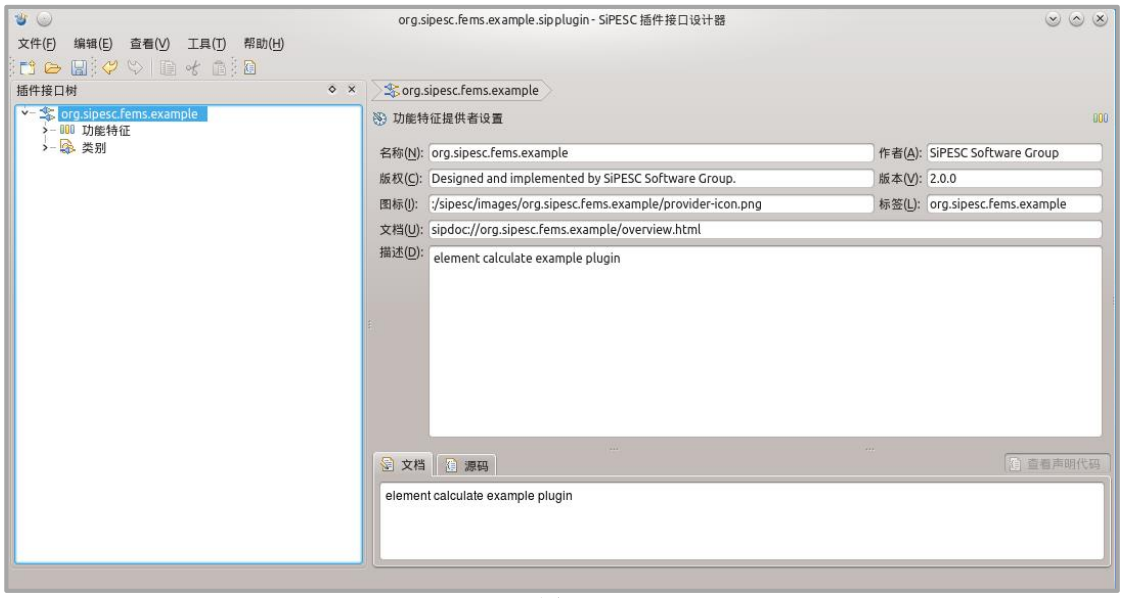


图 3-1

2.添加新的“功能特征”，其名称对应所添加的单元名称，如图 3-2.

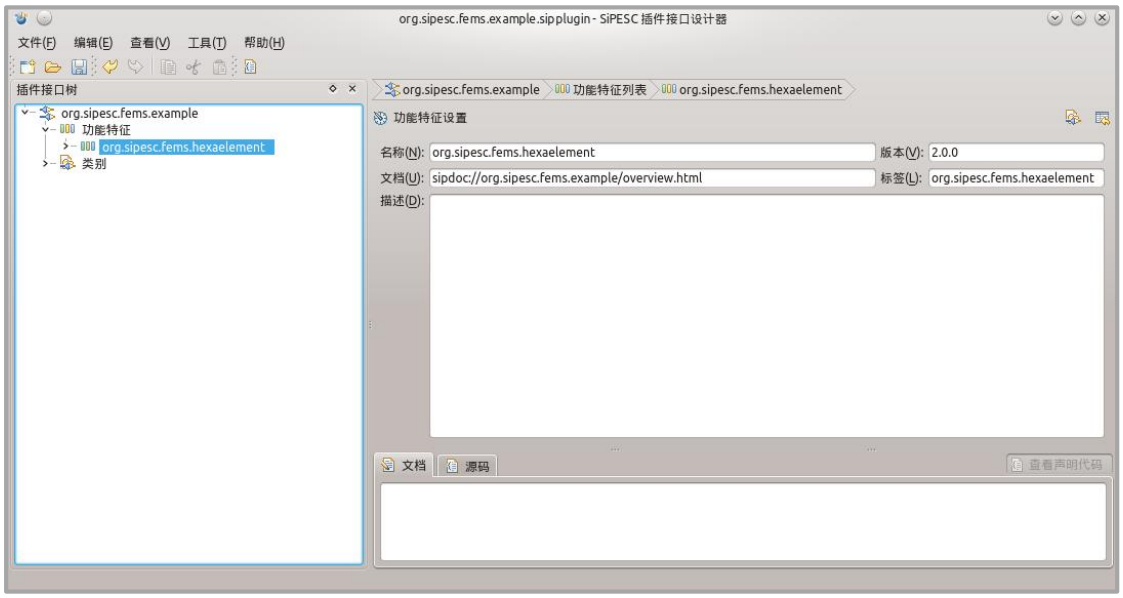


图 3-2

3.添加每一个所需扩展，并一一对基类，返回值，方法，属性等进行设置。如图 3-3 中的内容。

图 3-3

由于 SiPESC 平台有限元模块中一些基类是已有的，（如果没有特殊的开发要求）开发者的扩展只需要来继承这些基类的接口即可。例子中的六面体块单元的相关扩展就不需要添加方法，返回值等信息，只需要给定它所继承的基类。本例中的扩展和基类的继承关系如下（每个扩展的功能将在 4、5 章介绍）：

（A ⟶ B 表示 A 继承 B）

MCHEXAHandler ⟶ org.sipesc.fems.bdfimport.MBdfEntryHandler

MBrickEleConstiDParser ⟶ org.sipesc.fems.element.MEleConstiDParser

MHexaBrickStrainMatrix ⟶ org.sipesc.fems.element.MElementStrainMatrix

MHexaShpFunction ⟶ org.sipesc.fems.element.MshpFunction

MGau3D2IntegForm ⟶ org.sipesc.fems.element.MIntegForm

MHexaBrickNodeStressCalculator ⟶org.sipesc.fems.stress.MElementNodeStressCalculator

MUnvHexaBrickEleDataExportor ⟶ org.sipesc.fems.jfxexport.MdataExportor

对于本例中的 Factory，都继承相同基类，不再一一列举，格式如下。

XXXXXXXXXXXXXXXXXXFactory ⟶ org.sipesc.utilities.MExtensionFactory

对于实际用于使用而不是作为基类的扩展都要勾选“可创建”和“可脚本化”，否则可能会导致将来的编译错误。在“分类”一栏中的填写请参照示例的 sipplugin 文件中的写法，在“功能特征”同一级菜单中有“类别”一项，可从此处以“分类”一栏中所填写内容的不同为区别来查看各个扩展。 另外，在每个扩展”描述”一栏中建议写上一些扩展功能的简要说明（支持中文），以便其他用户和开发者的使用和维护。

4.生成代码。新建一个文件夹，用于存放该单元插件的 C++ project，在子目录中建立 doc 文件夹并将.sipplugin 的插件设计器工程文件保存于此。用插件设计器生成代码，该流程和 HelloWorld 文档中生成代码一致。见图 3-4.

图 3-4

至此，单元插件 C++ project 的基本框架构建已经完成，在 4-5 章着重介绍本章设计的扩展的功能和实现方法。

4. 单元插件中单元刚度阵的形成方法。

如同 HelloWorld，插件设计器设计出的接口类内容是空的，开发者需要写每一个扩展对应的实现类。这包括每一个扩展实现类的头文件和源文件，而工厂扩展只需要编写它的头文件。

本章主要介绍六面体单元插件中涉及单刚形成的实现类。如图 4-1 所示，给出单元刚度阵计算模块各对象的关系及接口方法定义。其中蓝色部分都是需要我们写实现文件的，绿色的是平台已有的扩展。

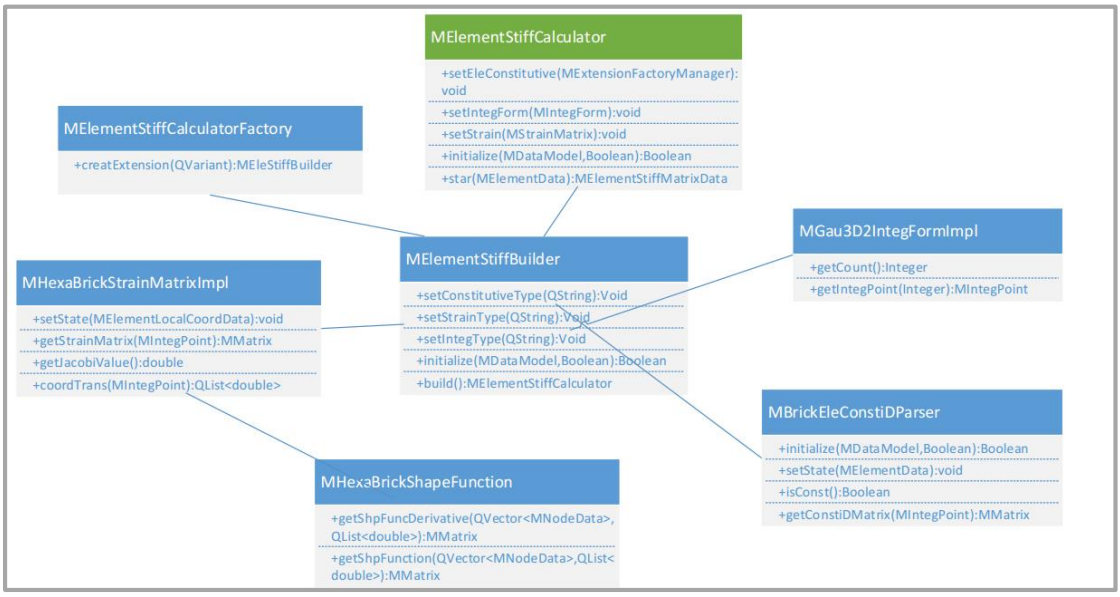


图 4-1

这里需要特别说明的是，在 SiPESC 平台单元插件的开发中，用到了工厂模式（Factory）和构造器模式（Builder）两种重要的**设计模式**。

设计模式（Design pattern）是在对软件代码复用问题设计的提取。通过获取对象之间的关系，解决方案，描述它们的参与合作，及对调用成熟解决方案的推进，提高代码可重用性，让代码更容易被他人理解、保证代码可靠性，并能提高软件的质量，缩短研发时间。

工厂模式。有限元系统开发过程中，在单元刚度计算时对于不同的单元会生成不同的计算对象，传统方法往往会用 new 来生成，因其需要代码中显示写明具体兑现，降低了代码的重用性和可扩展性。在大规模的系统开发过程中，为了避免这一问题，并考虑系统的灵活性，低耦合性，可扩展性等因素，常常不直接使用 new 来生成类的实例，而是通过一个叫做工厂 (Factory) 的类来专门生成类的实例。使用工厂方法模式，一方面，可以不用关心具体对象的实现，简化和统一调用过程；另一方面，可通过工厂提供的优先级方法，动态替换单元计算模块，使系统具有灵活的可扩展性。

构造器模式。它主要是用于创建一些复杂的对象，这些对象内部构件间的建造顺序通常是稳定的，但对象内部的构件通常面临着复杂的变化。构造器模式就是将这一复杂对象的构建与它的表示分离，使得同样的构造过程可以创建不同的表示。它的好处就是使得建造代码与标识代码分离，由于构造者隐藏了该产品是如何组装的，所以若需要改变一个产品的内部表示，只需要在定义一个具体的构造者就可以了。由于单元计算过程是由本构矩阵、应变矩阵和积分点等对象共同确定，因此单元计算前通过构造器 MElementStiffBuilder 对象把所需对象生成，传递给单元计算 MElementStiffCalculator 对象，这样单元计算对象就只负责计算任务，而不必要判读单元的相关性质。对于不同类型的单元计算只要定义相应的构造器

就可以了，体现了单元计算的通用性和可重用性。

在单元刚度阵计算过程中，首先采用工厂模式，通过单元数据获取单元类型关键词，单元刚度计算工厂 MElementStiffCalculatorFactory 生成单元刚度计算构造器 MelementStiffBuilder。由构造器 MElementStiffBuilder 类生成本构矩阵解析、应变矩阵和积分点对象，传递给单元刚度计算器 MElementStiffCalculator 类，并创建对象，由单元刚度计算器完成计算，返回单元刚度数据对象。

本章主要从代码层面详细介绍六面体单元插件中涉及单刚形成的实现类写法，帮助初学者学习 SiPESC 单元计算模块。希望读者结合示例代码，理清平台上单刚计算的流程，抓住每一个细节，加深对 SiPESC 的了解。

4.1 单元数据的定义与存储。

首先有必要简单对单元数据类以及和单元计算有关的几个数据类做一简单说明。

数据类型	功能介绍	存储位置和取出数据操作方法

单元数据类（MElementData）	目的是定义保存单元节点（在总体坐标下的节点号）、性质引用（单元属性号）和单元节点自由度贡献数据，为计算模块提供统一的接口方法，其创建方法是通过工厂。	单元信息存在数据库根目录中的 ElementPath 文件夹下，按单元类型分别储存（本例只有一种单元，为 HexaBrickElement），每个单元的信息以 MElementData 类存在 HexaBrickElement 文件中。
		取出某一个单元的单元信息：
		model //根目录
		elementPath.open(model, “ElementPath”);
		elementManager.open(elementPath, “HexaBrickElement”);
		MelementData eleData = elementManager.getData(eleId);

节点数据类（MNodeData）	主要储存每个节点的坐标值。根据节点号，储存节点在整体坐标下的坐标值。	节点信息存在数据库根目录中的 Node 文件中，以 MNodeData 类型储存。
		取出某一个节点的节点信息：
		model //根目录
		nodeManager.open(model, “Node”);
		MNodeData nodeData = nodeManager.getData(nodeId);

材料属性类（MPropertyData）	以材料号为索引储存材料的物理性质信息，本例中储存的是泊松比和弹性模量。在计算本构矩阵（式 2-15）时，将会用到它所储存的泊松比和弹性模量。	材料信息存在数据库根目录中的 Material 文件中，以 MPropertyData 类型储存，根据材料属性 Id 提取。
		取出某一种材料的信息：
		model //根目录
		materialManager.open(model, “Material”);
		MPropertyData materialData = materialManager.getData(materialId);
		materialData 中第一个数据存的是弹性模量，第二个数据是泊松比。
		弹性模量：double ex = materialData.getValue(1).toDouble();
		泊松比：double prxy = materialData.getValue(2).toDouble();

单元属性类（MPropertyRefData）	储存单元的类型和与其相关的一些信息。本例中它储存的是单元材料号信息。	单元属性信息存在数据库根目录中的 PropertyRefPath 文件夹下 General 文件中，以 MPropertyRefData 类型储存，根据单元属性 Id 获取（单元属性 Id 存于 MElementData）。

		取出某一种单元属性的信息：
		elementPath.open(model, "PropertyRefPath");
		propertyManager.open(PropertyRefPath, "General");
		MPropertyRefData propertyRefData = propertyManager.getData(propertyId);
		以提取材料属性 Id 为例，从单元属性 propertyRefData 中获取信息：
		int matId = propertyRefData.getPropertyId(
		_data->_meleGl.getValue(MElementsGlobal::MaterialId));

在本指南中的 **4.1.2 节**中的单元数据导入的实现代码中，有较为详尽的**数据库操作**介绍。更全面的使用方法介绍请参见示例代码或者卢旭泽《*SiPESC* 工程数据库从入门到精通》系列 ppt。

4.1.1 单元数据实现基类 MElementDataImpl，六面体单元数据实现类 MHexaBrickElementDataImpl，工厂实现类 MHexaBrickElementDataFctoryImpl

MElementDataImpl 是平台的单元数据基类，定义了各种单元数据的基本存取方法，代码固定，但是开发者需要把它的源文件和头文件放置在单元插件文件夹中的对应位置。

MHexaBrickElementDataImpl 继承于 **MelementDataImpl**，主要用于对六面体单元的单元数据内容作基本定义。

mhexabrickelementdataimpl.cxx（部分代码 1）	Description

MHexaBrickElementDataImpl::MHexaBrickElementDataImpl()
{
_propertyId = 0;
_number = 8;
_nodeId = _node8Id;
}
 	
MHexaBrickElementDataImpl::~*MHexaBrickElementDataImpl*()
	◆ **构造函数**
	_propertyId 是单元性质引用（单元属性号），这里设定了一个初值 0.
	_number 的赋值"8"是一个六面体单元中的节点个数。
	_nodeId 是一个大小为 8 的整型数组，用于储存一个单元内的节点号信息。在该类的头文件中有此整型数组的定义。

{
}
 	
QString MHexaBrickElementDataImpl::getType() const
{
MObjectManager manager = MObjectManager::getManager();
MElementsGlobal global = manager.getObject(
"org.sipesc.fems.global.elementsglobal");
 	
return global.getValue(MElementsGlobal::HexaBrickElement);
}
 	
QList<int> MHexaBrickElementDataImpl::getActiveDofs(int index)
{
MObjectManager manager = MObjectManager::getManager();
MElementActiveDofs dof = manager.getObject(
"org.sipesc.fems.data.elementactivedofs");
 	
return dof.getActiveDofs(MElementActiveDofs::MBrick);
}
	◆ 方法 *getType*
	
	用于获取六面体单元类型。
	
	在 MelementsGlobal 中存着平台已有的各种单元类型。**但开发者新添加一种单元的时候，新类型往往没有存进 MelementsGlobal 中，这时直接返回单元类型名的字符串即可**，例如新添加三角型 IMDKT 板单元时，**可直接写**：
	
	return "TriIMDKTPlElement";
	
	而不用再对 MelementsGlobal 进行操作，后面遇到相同情况不再解释。
	
	◆ 方法 *getActiveDofs*
	
	用于获取活动自由度（无约束下的自由度）信息。
	
	我们可以看出此处返回的是块单元的自由度，向有限元分析流程提供了六面体单元的活动自由度信息。
	
	MelementActiveDofs 中存储了不同类型单元的活动自由度信息，开发者需要查询该类的头文件，寻找和所添加单元符合的活动自由度类型。

工厂实现 **MHexaBrickElementDataFctoryImpl** 也定义在 mhexabrickelementdataimpl.cxx 文件中。

mhexabrickelementdataimpl.cxx（部分代码 2）	**Description**

MHexaBrickElementDataFactoryImpl::MHexaBrickElementDataFactoryImpl()
{
}
 	
MDataObject MHexaBrickElementDataFactoryImpl::createObject() const
{
return MDataObject(new MHexaBrickElementDataImpl);
}
 	
int MHexaBrickElementDataFactoryImpl::supportedType(
const QUuid& typeId) const
{
	◆ 方法 *createObject*
	
	用于创建数据类 MhexaBrickElementDataImpl 的对象。
	
	这也是该工厂的主要作用，后面提到的工厂大多是用于创建扩展，遇到时还会详细介绍。
	
	◆ 方法 *supportedType*
	
	用于判断该工厂的支持级别是否符合调用要求。
	
	工厂模式中，外部调用模块根据支持级别决定是否调用该工厂来创建相应对象。
	
	这个支持级别是一个标识，在这里是六

bool is = (typeId == MHexaBrickElementDataUuid);
return is ? 1 : 0;
}
	面体块单元数据类的 uuid.
	
	MhexaBrickElementDataUuid 的赋值在 mhexabrickelementdataimpl.h 中（具体参见示例代码）。这个值得获取方法是在插件设计器中随意新建一个扩展，复制其 uuid。
	
	注意：一定要在插件设计器中新建一个扩展，使用插件设计器随机生成新的 uuid，否则会发生冲突，导致程序崩溃。

4.1.2 单元数据导入实现类 MCHEXAHandlerImpl

该类主要实现读取.bdf 文件中的单元信息。

mchexhandlerimpl.cxx（部分代码）	Description

class MCHEXAHandlerImpl::Data
{
public:
Data()
{
_isInitialized = false;
}
 	
public:
MDataModel _model, _elementPath;
MDataManager _hexManager;
MObjectManager _objManager;
ProgressMonitor _monitor;
MDataFactory _eleFactory;
bool _isInitialized;
};
 	
MCHEXAHandlerImpl::MCHEXAHandlerImpl()
{
_data.reset(new MCHEXAHandlerImpl::Data);
_data->_objManager = MObjectManager::getManager();
UtilityManager util = _data->_objManager.getObject(
"org.sipesc.core.utility.utilitymanager");
	◆ 智能指针类的定义
	
	◆ 构造函数
	
	重置数据。
	
	初始化进程管理器。
	
	进程管理器主要用于监视整个进程的进行，错误信息的处理等等，在我们的代码文件中有大量使用。

```
  _data->_monitor =
  util.createProgressMonitor("MCHEXAHandler",
        ProgressMonitor());
  }

MCHEXAHandlerImpl::~MCHEXAHandlerImpl()
  {
  }

bool MCHEXAHandlerImpl::initialize(MDataModel& model, bool
isRepeated)
  {
  if (_data->_isInitialized)
    return false;
  _data->_model = model;
  MFemsGlobal global = _data->_objManager.getObject(
        "org.sipesc.fems.global.femsglobal");//①
  MElementsGlobal eleGlobal = _data->_objManager.getObject(
        "org.sipesc.fems.global.elementsglobal");//②

  MDatabaseManager baseManager =
  _data->_objManager.getObject(
        "org.sipesc.core.engdbs.mdatabasemanager");//③
  MDataFactoryManager factoryManager =
  _data->_objManager.getObject(
        "org.sipesc.core.engdbs.mdatafactorymanager");//④
  QString name = "org.sipesc.fems.data.hexabrickelementdata";
  _data->_eleFactory = factoryManager.getFactory(name); //⑤
  _data->_elementPath = baseManager.createDataModel();//⑥
  bool ok = _data->_elementPath.open(_data->_model,
        global.getValue(MFemsGlobal::ElementPath)); //⑦
  _data->_hexManager = baseManager.createDataManager();//⑧
  ok = _data->_hexManager.open(_data->_elementPath,
        eleGlobal.getValue(MElementsGlobal::HexaBrickElement));
//⑨
  _data->_isInitialized = true;
  return true;
  }
ProgressMonitor MCHEXAHandlerImpl::getProgressMonitor()
const
  {
  return _data->_monitor;
  }
```

◆ **方法 *initialize***

一般用于对数据库初始化以及对计算或者操作进行数据准备。

这里的操作涉及很多 SiPESC 平台工程数据库的使用方法，请配合卢旭泽《SiPESC 工程数据库从入门到精通》系列 ppt 理解。

创建 MFemsGlobal 的对象，MFemsGlobal 中存储了平台有限元计算中所涉及的许多数据类型，这其中包括 ElementPath，在⑦要打开以这个以 ElementPath 名字命名的 MDataModel。

②创建 MElementsGlobal 的对象，MElementsGlobal 中存储着平台已有的单元类型名，其中包括 HexaBrickElement。⑨中打开以此单元名命名的单元数据文件。

③创建 MDatabaseManager 的对象，其中的方法 *creatDataModel* 用于创建数据类 MDataModel，方法 *createDataManager* 用于创建 MdataManager。

④创建 MDataFactoryManager 的对象，在⑤中将使用它创建六面体单元数据类的工厂，该工厂用于创建六面体单元的单元数据对象。

⑤创建六面体单元数据类的工厂。在方法 *handleEntry* 中创建六面体单元数据对象。

⑥通过 MDatabaseManager 建立 MDataModel，它相当于一个子文件夹。

⑦打开这个 MDataModel（以 ElementPath 命名的文件夹）。

⑧创建 MDataManager 的对象，它相当于数据库中的具体数据文件。

⑨打开这个 MDataManager 对象，相当于打开这个名为 HexaBrickElement 的数据文件。六面体的单元数据就将存进这个对象中。

bool MCHEXAHandlerImpl::handleEntry(const MBdfEntryReader& reader)
{
MElementData eleData = _data->_eleFactory.createObject();
//单元 Id
int eleId = reader.getDataAt(0).toInt();
eleData.setId(eleId);
//单元性质引用 Id
eleData.setPropertyId(reader.getDataAt(1).toInt());
//单元节点 Id
for (int i = 0; i < 8; i++)
{
int nodeId = reader.getDataAt(2 + i).toInt();
eleData.setNodeId(i, nodeId);
}
_data->_hexManager.appendData(eleData);
 	
return true;
}
	◆ 方法 *getProgressMonitor*
	用于获取进程管理器。
	
	◆ 方法 *handleEntry*
	用于完成对单元数据的读取和储存。
	首先创建六面体单元数据类的对象 eleData。
	MbdfEntryReader 的对象 reader 中的方法 *getDataAt（i）.toInt* 作用是把在.bdf 中单元数据部分第 i 个位置的信息读入并转化成整型。
	第 1 个位置是单元编号信息，第 2 个位置是单元性质引用 Id（单元属性号），第 3 到 10 位置是单元节点编号。
	然后把这些我们读入的信息 set 到我们创建的六面体单元数据类的对象 eleData 中，最后再将存有单元信息的 eleData 添加到数据库中。

4.1.3 单元数据导入工厂实现类 MCHEXAHandlerFactoryImpl

在 1.2.2 节我们提到，平台每一个扩展都有对应的一个工厂对象，该工厂对象负责提供扩展的标识，支持级别以及创建扩展方法，外部调用模块根据自身需求和支持级别决定是否创建该扩展。下面是数据导入工厂实现类的代码，该工厂主要在有限元流程进行时创建 **MCHEXAHandler** 扩展，用于读取单元信息。

mchexhandlerfactoryimpl.h	Description

class MCHEXAHandlerFactoryImpl: public MCHEXAHandlerFactoryInterface
{
public:
virtual ~MCHEXAHandlerFactoryImpl()
{
}
QList<QVariant> getTypes() const
{
QList<QVariant> list;
list << "CHEXA";
return list;
	◆ 方法 *getTypes*
	该方法用于获取支持类型。
	工厂模式中，外部调用模块根据支持级别决定是否调用该工厂来创建相应对象。这里的支持类型的返回值是字符串“CHEXA”，它是六面体单元在 Patran 和 SiPESC 平台的关键字，是.bdf 数据文件中的单元类型名称。有限元主程序的数据导入模块读入到这样关键字的时候，调用此工厂来创建对应的扩展读取并储存单元数据。

```
    }
  QString getDescription() const
    {
    return "factory for MCHEXAHandler";
    }
  QString getIcon() const
    {
    return QString();
    }
  uint getPriority(const QVariant& key) const
    {
    return 10;
    }
  MExtension createExtension(const QVariant& key) const
{
    MCHEXAHandlerImpl *p = new MCHEXAHandlerImpl;
    return MCHEXAHandler(p);
    }
};
```

◆ **方法 *getDescription***

该方法用于获取此对工厂的描述信息。

◆ **方法 *getIcon***

该方法用于获取图标信息，这里没有图标所以返回一个空的字符串。

◆ **方法 *getPriority***

该方法用于设置该工厂的优先级是 10.

外部程序在调用工厂时，如果遇到工厂分类和扩展标识都相同的多个扩展，将会根据优先级来决定调用哪个工厂。

◆ **方法 *createExtension***

该方法实现的是创建扩展。

这也是该工厂的主要目的。

4.2 计算形函数以及形函数导数。

4.2.1 形函数计算实现类 MHexaShpFunctionImpl

该类的定义中完成了对形函数以及形函数导数的确定，这部分内容对应 2.2.1 和 2.2.2 节的内容。

注意，从本节开始，和 3.1 节中类似的初始化方法，智能指针等的用法不再重复介绍。

mhexashpfunctionimpl.cxx（部分代码）	Description

MHexaShpFunctionImpl::MHexaShpFunctionImpl()
{
MObjectManager objManager = MObjectManager::getManager();
_mFactory = objManager.getObject("org.sipesc.fems.matrix.matrixfactory");
}
MHexaShpFunctionImpl::~*MHexaShpFunctionImpl*()
{
}
	◆ **构造函数**
	初始化 MMatrixFactory 类的对象 _mfactory.
	注意现在创建 MMatrix 类需要通过工厂的方法完成。

MMatrix MHexaShpFunctionImpl::getShpFunction(
const QList<double>& paraCoord,
const QVector<MNodeData>& nodeLocalCoord) const
{
MMatrix mdn = _mFactory.createMatrix(1, 8);
double ksi, eit, cait;
ksi = paraCoord.value(0);
eit = paraCoord.value(1);
cait = paraCoord.value(2);
mdn(0, 0, (1 - ksi) * (1 - eit) * (1 - cait) / 8);
mdn(0, 1, (1 + ksi) * (1 - eit) * (1 - cait) / 8);
/ * 矩阵赋值操作，部分省略，具体了解请参见示例代码 * /
mdn(0, 6, (1 + ksi) * (1 + eit) * (1 + cait) / 8);
mdn(0, 7, (1 - ksi) * (1 + eit) * (1 + cait) / 8);
return mdn;
}
	◆ **方法 *getShpFunction***
	该方法获取形函数。在这个方法中定义中，应该包含六面体单元的形函数信息。
	关注具体代码，首先建立一个 1 行 8 列的矩阵（请读者根据示例代码总结平台矩阵类的使用方法，如创建方式，赋值方法）。再对参数坐标赋值，这个值我们将在 3.4 节讲到。最后把形函数依次（式 2-3）填入矩阵。

MMatrix MHexaShpFunctionImpl::getShpFuncDerivative(
const QList<double>& paraCoord,
const QVector<MNodeData>& nodeLocalCoord) const
{
MMatrix msfd= _mFactory.createMatrix(3, 8);
double ksi, eit, cait;
ksi = paraCoord.value(0);
eit = paraCoord.value(1);
cait = paraCoord.value(2);
msfd(0, 0, -(1 - eit) * (1 - cait) / 8);
msfd(1, 0, -(1 - ksi) * (1 - cait) / 8) ;
/ * 矩阵赋值操作，部分省略，具体了解请参见示例代码 * /
msfd(1, 7, (1 - ksi) * (1 + cait) / 8);
msfd(2, 7, (1 + eit) * (1 - ksi) / 8);
return msfd; }
	◆ **方法 *getShpFuncDerivative***
	该方法获取形函数导数.
	方法 *getShpFuncDerivative* 获取形函数导数，在第二章我们介绍过计算雅克比矩阵 J 需要用到形函数对参数坐标的导数，具体见（式 2-6）。我们需要把这些求导后的结果填入这个 3 行 8 列矩阵。

4.2.2 形函数计算工厂实现类 MHexaShpFunctionFactoryImpl

在此想特别说明的是，上文介绍过用于创建单元数据对象的工厂 **MhexaBrickElementDataFctory** 和用于创建单元数据读入扩展的工厂 **MCHEXAHandlerFactory**。该处的工厂目的也是创建扩展，但扩展标识不同于 **MCHEXAHandlerFactory** 的”CHEXA”，而是**“HexaShpFunction”**。其它部分都很类似，下文中对类似工厂不再做解释，请关注示例代码。

mhexashpfunctionfactoryimpl.h（获取类型部分）	Description

public:
QList<QVariant> getTypes() const
{
MObjectManager objManager =
MObjectManager::getManager();
MElementsGlobal global = objManager.getObject(
"org.sipesc.fems.global.elementsglobal");
QList<QVariant> list;
list << global.getValue(MElementsGlobal::HexaShpFunction);
return list;
}
	◆ 方法 *getTypes*
	获取工厂创建扩展的标识 HexaShpFunction，这个名称也存在 MElementsGlobal 中。
	这里同样要注意开发者新添加一种单元的时候，新类型往往没有存进 MElementsGlobal 中，这时直接返回形函数类型名的字符串即可，例如新添加三角型 IMDKT 板单元时，可直接写：
	return " TriIMDKTPlShpFunction";
	而不用再对 MElementsGlobal 进行操作。

4.3 计算单元应变矩阵。

4.3.1 单元应变计算实现类 MHexaBrickStrainMatrixImpl

该类用于计算单元应变阵 **B**。

mhexashpfunctionfactoryimpl.h（获取类型部分）	Description

```
public:
    bool initialize()
 {
       MObjectManager objManager =
MObjectManager::getManager();
       _mFactory = objManager.getObject(
           "org.sipesc.fems.matrix.matrixfactory");//①②③
       MExtensionManager extManager =
MExtensionManager::getManager();
       _shpFunction = extManager.createExtension(
           "org.sipesc.fems.hexaelement.MHexaShpFunction");//②
       _mTools = extManager.createExtension(
           "org.sipesc.fems.matrix.MMatrixTools");//③
       return true;
       }
  };
        /*此处省略构造函数和析构函数代码*/
void MHexaBrickStrainMatrixImpl::setState(const MDataObject&
localData)
  {
  _data->_coordData = localData;
  MNodeData nodeData;
  MMatrix node = _data->_mFactory.createMatrix(8, 3);
  for (int i = 0; i < 8; i++)
    {
    nodeData = _data->_coordData.getNodes().value(i);
    Q_ASSERT(!nodeData.isNull());
    node(i, 0, nodeData.getX());
    node(i, 1, nodeData.getY());
    node(i, 2, nodeData.getZ());
    }
  _data->_nodeCoord = node;
  }
```

◆ **initialize**

①初始化 MMatrixFactory，后面成员函数中建立矩阵需要用到。该操作在六面体单元形函数实现类中使用过。

②创建六面体单元形函数扩展，此扩展也是上一节我们写好的扩展。

注意：这里的扩展的创建方法——用 MExtensionManager 通过扩展名来创建。请回顾 1.2.2 节“两种调用平台扩展的方法”。另外一种方法是通过工厂创建扩展。

③实例化了 MMatrixTools，它是一个矩阵处理工具，我们后面要用它求矩阵的行列式和逆阵。

◆ **方法 *setState***

该方法主要对后面的应变矩阵计算提供数据准备。

在此处读取了数据库中的节点坐标数据（单元局部坐标下），并存到一个 8 行 3 列的矩阵中。我们知道，计算雅克比矩阵需要用到这些单元节点坐标数据（见式 2-6）。

```
MMatrix MHexaBrickStrainMatrixImpl::getStrainMatrix(
    const MIntegPoint& integ)
  {
  MMatrix shpFuncDerivative = _data->_mFactory.createMatrix(3,
8);
  shpFuncDerivative =
_data->_shpFunction.getShpFuncDerivative(
      integ.getIntegCoord()); //形函数导数矩阵（在积分点的值）
  MMatrix jacobiMatrix = _data->_mFactory.createMatrix(3, 3);
  jacobiMatrix = shpFuncDerivative * _data->_nodeCoord;
  _data->_jacobiValue =
_data->_mTools.determinate(jacobiMatrix); // 计算雅克比矩阵行列
式值
  MMatrix invJacobiMatrix = _data->_mFactory.createMatrix(3, 3);
  invJacobiMatrix = _data->_mTools.inverse(jacobiMatrix); //雅克
比矩阵的逆
  MMatrix B = _data->_mFactory.createMatrix(6, 24);
  MMatrix BI = _data->_mFactory.createMatrix(3, 1);
  MMatrix tmp = _data->_mFactory.createMatrix(3, 1);
  for (int i = 0; i < 8; i++)
    {
    BI(0, 0, shpFuncDerivative(0, i));
    BI(1, 0, shpFuncDerivative(1, i));
    BI(2, 0, shpFuncDerivative(2, i));
    tmp = invJacobiMatrix * BI;
    B(0, i * 3, tmp(0, 0));
    B(0, i * 3 + 1, 0.0);
    B(0, i * 3 + 2, 0.0);
    B(1, i * 3, 0.0);
    B(1, i * 3 + 1, tmp(1, 0));
```

◆ **方法 *getStrainMatrix***

此方法是该类的核心，它返回单元应变矩阵，也就是单刚计算中的 B 矩阵。

在该方法的实现中，首先得到单元在高斯点上的形函数导数矩阵 shpFuncDerivative，然后得到雅克比矩阵 jacobiMatrix（参见式 2-6）。对雅克比矩阵进行求行列式值得到雅克比值 _jacobiValue。求逆，得到雅克比矩阵的逆矩阵 invJcobiMatrix。下面的 for 循环其实是把形函数对整体坐标的导数值（矩阵 tmp 包含的内容）填入式 2-12 和式 2-13 的相应位置。

```
    B(1, i * 3 + 2, 0.0);
    B(2, i * 3, 0.0);
 /* 矩阵赋值操作，部分省略，具体了解请参见示例代码 */
    B(5, i * 3, tmp(2, 0));
    B(5, i * 3 + 1, 0.0);
    B(5, i * 3 + 2, tmp(0, 0));
    }
  return B;
  }
double MHexaBrickStrainMatrixImpl::getJacobiValue() const
  {
  return _data->_jacobiValue;
  }
QList<double> MHexaBrickStrainMatrixImpl::coordTrans(
    const MIntegPoint& integ) const
  {
  MMatrix shpFunc = _data->_mFactory.createMatrix(1, 8);
  shpFunc =
_data->_shpFunction.getShpFunction(integ.getIntegCoord()); //①
  MMatrix co = _data->_mFactory.createMatrix(1, 3);
  co = shpFunc * _data->_nodeCoord; //②
  MMatrixData mData = _data->_coordData.getTransMatrixData();
  MMatrix m = _data->_mFactory.createMatrix();
  m << mData; //③

  MMatrix coord = _data->_mFactory.createMatrix(3, 1);
  coord(0, 0, co(0, 0));
  coord(1, 0, co(0, 1));
  coord(2, 0, co(0, 2)); //④
  MMatrix tmp = _data->_mTools.transpose(m) * coord; //⑤
  QList<double> tmpCoord;
  tmpCoord.append(tmp(0, 0));
  tmpCoord.append(tmp(1, 0));
  tmpCoord.append(tmp(2, 0)); //⑥
  return tmpCoord;
  }
```

◆ **方法 *getJacobiValue***

该法用来获取雅克比行列式的值，用于单元刚度计算中高斯积分公式。

该方法直接返回在***getStrainMatrix*** 方法中计算出的_jacobiValue.

◆ **方法 *coordTrans***

该方法用于获取积分点在总体坐标系下的坐标。

值得注意的是，平台有限元计算中一共涉及 3 种坐标系。

1.总体坐标。有限元分析流程中读取的数据文件中的节点坐标就是基于总体坐标的。输出的位移、应力也都是基于总体坐标的。

2.单元局部坐标。平台有限元计算中单元刚度阵是在单元局部坐标下建立的。在方法 setstate 中的的参数就是单元局部坐标数据。有限元分析流程中的任务类 org.sipesc.fems.controlmatrix.

MelementLocalCoordTransStandardManager 用于得到总体坐标到单元局部坐标的每一个转换矩阵。

3.单元的参数坐标。等参元中其坐标转换关系和位移插值是一致的。高斯点的确定是基于此坐标系的。

该方法的流程是①取出积分点的形函数；②得到积分点在单元局部坐标系下的坐标；③从坐标数据_coordData 中得到坐标转换矩阵，注意这里取出的其实是旋转矩阵 R，它的转置是坐标转换阵 T，这个转换阵（3×3）将局部坐标下的点转换成总体坐标下；④是把刚才算出的局部坐标下的高斯点坐标转换成列向量；⑤把积分点从单元局部坐标转换到总体坐标；⑥保存数据

4.3.2 单元应变计算工厂实现类 MHexaBrickStrainMatrixFactoryImpl

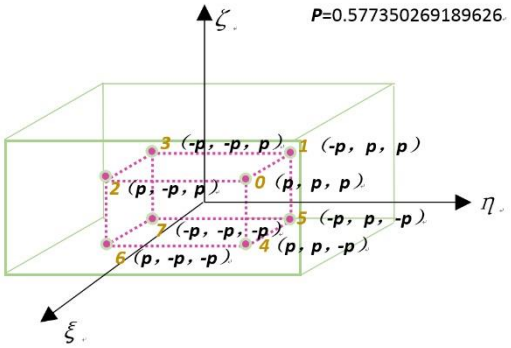
写法类似之前介绍的工厂，下文类似的工厂略去介绍内容。

4.4 积分格式的添加

在 3.2 和 3.3 的我们都直接或者间接地用到了单元的参数坐标，这些计算也都是基于高斯点的。本节将介绍 3 维的两点积分格式添加方法。相应的读者可以根据数值积分相关理论，模仿该格式添加 3 维的 3 点，4 点积分格式。

4.4.1 积分格式实现类 MGau3D2IntegFormImpl

在 2.4 节已经介绍过数值积分的方法和六面体单元的积分格式，以下内容可对照 2.4 节帮助理解。主要实现的是把积分点权系数的乘积，积分点在参数坐标下的值，以及积分点的编号 apend 到_integ 对象中。

mgau3d2integformimpl.cxx	Description
六面体在参数坐标下的积分点示意  	◆ **示意图说明** 绿色框轮廓表示的是参数坐标系下的六面体单元示意图。红色点是高斯积分的积分点位置。金色数字是对积分点的标号。

public:
bool initialize()
{
MExtensionManager extManager = MExtensionManager::getManager();
QList<double> weights, integCoord;
weights.append(1.0);
integCoord.append(0.577350269189626);
weights.append(1.0);
integCoord.append(-0.577350269189626);
	◆ **方法 *initialize***
	在 2.4 节已经介绍过数值积分的方法和六面体单元的积分格式，以下内容可对照 2.4 节帮助理解。主要实现的是把积分点权系数的乘积，积分点在参数坐标下的值，以及积分点的编号 apend 到_integ（Qlist 类型）对象中。
	参见本例中的图示可以清楚了解积分点的添加方式。

```
        for (int i = 0; i < 2; i++)
          for (int j = 0; j < 2; j++)
            for (int k = 0; k < 2; k++)
              {
              MIntegPoint integPoint = extManager.createExtension(
                  "org.sipesc.fems.femsutils.MIntegPoint");
              integPoint.setWeight(
                  weights.value(i) * weights.value(j) * weights.value(k));
              QList<double> paraCoord;
              paraCoord.append(integCoord.value(k));
              paraCoord.append(integCoord.value(j));
              paraCoord.append(integCoord.value(i));
              integPoint.setIntegCoord(paraCoord);
              int index = i * 4 + j * 2 + k;
              integPoint.setIndex(index);
              _integ.append(integPoint);
              }
        return true;
        }
    };

MGau3D2IntegFormImpl::MGau3D2IntegFormImpl()
    {
    _data.reset(new MGau3D2IntegFormImpl::Data);
    }

MGau3D2IntegFormImpl::~MGau3D2IntegFormImpl()
    {
    }

int MGau3D2IntegFormImpl::getCount() const
    {
    return 8;
    }

MIntegPoint MGau3D2IntegFormImpl::getIntegPoint(const int& index)
const
    {
    return _data->_integ.value(index);
    }
```

◆ **方法 *getCount***

返回积分点个数 8.

◆ **方法 *getIntegPoint***

根据积分点的编号返回积分点的信息。

4.4.2 积分格式工厂实现类 MGau3D2IntegFormFactoryImpl

略。

4.5 单元本构矩阵的计算

本节理论和公式部分在 2.4 节。非线性问题的单元本构阵是不断变化的，涉及平台更多操作，然而连续体线性问题的本构矩阵比较容易得到，这里只针对本例中的问题做简单介绍。

4.5.1 单元本构解析实现类 MbrickEleConstiDParserImpl

mbrickeleconstidparserimpl.cxx	Description

public:
 	
MMatrix findConstiDMatrix(const int& index)
{
//首先在 Map 中查找
if (_matMatrixMap.contains(index))
return _matMatrixMap[index];
 	
MPropertyData matData;
matData = _materials.getData(index);
_matParser.setState(matData);
 	
MMatrix constMatrix;
constMatrix = _matParser.getElasticD();
_matMatrixMap.insert(index, constMatrix);
 	
return constMatrix;
}
};
	◆ 方法 ***findConstiDMatrix***
	
	用于获取单元本构阵 D。
	
	它的形参是材料属性号，根据材料属性号，我们可以在 MPropertyData 中取出材料的泊松比 υ 和弹性模量 E，从而轻易得到单元本构阵 D。
	
	由于在一次有限元计算中有许多重复单元类型，重复的还有材料的属性，那么意味着同维问题的 D 阵也是相同的。这里巧妙的引入了 QMap 类，避免了**重复计算**。这个方法的开始就先进行一个判断：_matMatrixMap 对象中是否已包含此次计算的材料号对应的 D 阵，如果有，直接返回这个 D 阵，如果不包含，才进行下面计算。
	
	MsolidMaterialParser 的对象_matParser，在赋给它材料信息后，可以通过 ***getElasticD*** 方法获取自动生成的 D 阵。在返回数据前，把这次算出 D 阵插入到 _matMatrixMap 中以便下次的简化计算。
	
	◆ 关于后面的代码
	
	材料号储存在 MpropertyRefData 中，取出材料号需要对应的单元属性号，单元属性号储存于 MelementData 中（详见 3.1 节内容），可见得到一个单元对应的材料属性要依次对数个数据库进行操作。该源文件的后面部分主要就是进行这些操作获取对应材料号。
	
	相信了解了前文内容，参照代码注释会很容易理解后面的每个方法进行的操作，这里也略去此部分内容的具体介绍。

4.5.2 单元本构解析工厂实现类 MBrickEleConstiDParserFactoryImpl

略。

4.6 组装单元刚度矩阵

有了以上 3.1-3.5 的准备工作，单元刚度阵的计算变的已经非常简单。在平台上，单元刚度阵的计算器会自动进行数值积分。它需要三个信息：**B** 矩阵，**D** 矩阵，积分格式。

4.5.1 单元刚度计算工厂实现类 MHexaBrickEleStiffCalculatorFactoryImpl

mhexabrickelestiffcalculatorfactoryimpl.h	Description

public:
QList<QVariant> getTypes() const
{
QList<QVariant> list;
list <<
_global.getValue(MElementsGlobal::HexaBrickElement);
return list;
}
QString getDescription() const
{
return "factory for MHexaBrickEleStiffCalculator";
}
QString getIcon() const
{
return QString();
}
uint getPriority(const QVariant& key) const
{
return 10;
}
MExtension createExtension(const QVariant& key) const
{
MExtensionManager extManager =
MExtensionManager::getManager();
MElementStiffBuilder eleStiffBuilder =
extManager.createExtension(
"org.sipesc.fems.element.MElementStiffBuilder");
eleStiffBuilder.setStrainType(
	◆ 方法 *getTypes*
	支持级别返回单元名。
	
	◆ 方法 *createExtension*
	创建的扩展是构造器 MElementStiffBuilder 的对象 eleStiffBuilder。
	把积分格式 Gau3D2IntegForm，单元应变阵 HexaBrickStrainMatrix 和本构阵求解分析器”brickeleconstidparser”三部分内容 set 到对象 eleStiffBuilder 中。
	eleStiffBuilder 把本构矩阵解析、应变矩阵和积分点对象，传递给单元刚度计算器 MelementStiffCalculator.从而完成单刚计算。
	由于采用由于采用**构造器设计模式**，我们在此可以灵活的**替换**积分格式，应变阵，本构阵，从而完成更具**扩展性**和**灵活性**的单元刚度计算。
	对于一些**组合单元**，如三角形 DKT 壳单元的单元刚度阵，可采用三角形 DKT 板和三角形膜单元的刚度阵的组合拼装而成。如果平台已有三角形 DKT 板和三角形膜单元的单刚计算模块。在求解三角形 DKT 壳单元时，开发者不需要像写普通单元一样把积分格式，应变阵和本构阵重新写一遍，而只需要在 MTriDKTShEleStiffCalculatorFactoryImpl 中创

```
_global.getValue(MElementsGlobal::HexaBrickStrainMatrix));
        eleStiffBuilder.setConstType("brickeleconstidparser");
        eleStiffBuilder.setIntegType(

_global.getValue(MElementsGlobal::Gau3D2IntegForm));
        return eleStiffBuilder;
        }
```

建扩展 MAssEleStiffBuilder 来完成单元的组装。这大大提高了代码的**复用率**，增加了**开发效率**。

接口 createExtension 的实现如下：

```
MExtension createExtension(const
QVariant& key) const
        {
        MExtensionManager extManager
= MExtensionManager::getManager();
        MAssEleStiffBuilder
eleAssStiffBuilder =
extManager.createExtension(
"org.sipesc.fems.element.MAssEleStiffBui
lder");
eleAssStiffBuilder.setAssembType("org.si
pesc.fems.data.tridrillmembelementdata"
,
"org.sipesc.fems.data.tridktplelementdat
a");

        return eleAssStiffBuilder;
        }
```

5. 有限元后处理计算和数据导出。

第 2 和第 3 章分别讲了形成单元刚度阵的理论和平台插件的具体代码。由于节点应力的计算方式相对直接，理论和代码都在本章进行介绍。

5.1 节点应力计算（外推计算）的理论基础

在形成单刚后，有限元的流程会继续往下进行，完成对位移的求解，根据我们最初位移法有限元的假设，此时求出的是节点上的位移。如果想求节点上的应力，需要将局部坐标下的节点应力乘上 **B** 阵计算出高斯点上的应变，再根据 **D** 阵得到高斯点上的应力值。这时需要把高斯点上的应力值外推到节点上，方法是通过插值函数，具体如下：

假设我们已算出一个单元里 8 个高斯点上的应力值，每个点有 6 个应力分量，所有高斯点的第 n 个应力分量列阵可表示为

$$\underset{(8\times1)}{\boldsymbol{\sigma}_{gn}}=\begin{bmatrix}\sigma_{g(1)n}\\ \sigma_{g(2)n}\\ \vdots\\ \sigma_{g(8)n}\end{bmatrix}(n=1,2,3,4,5,6) \tag{5-1}$$

应力的插值函数我们也取作

$$\begin{aligned}&N_i^{'}=\frac{1}{8}(1+\xi_0)(1+\eta_0)(1+\zeta_0),\\ &\text{其中}\xi_0=\xi_i\xi,\eta_0=\eta_i\eta,\zeta_0=\zeta_i\zeta.\\ &(\xi_i,\eta_i,\zeta_i\text{分别为i节点的参数坐标值})\end{aligned} \tag{5-2}$$

设该单元所有节点的第 n 个应力分量列阵为

$$\underset{(8\times1)}{\boldsymbol{\sigma}_{n}}=\begin{bmatrix}\sigma_{(1)n}\\ \sigma_{(2)n}\\ \vdots\\ \sigma_{(8)n}\end{bmatrix}(n=1,2,3,4,5,6) \tag{5-3}$$

根据应力插值函数的原理，第 k 个高斯点上的第 n 个应力分量为

$$\sigma_{gn}=\sum_{i=1}^{8}N_{i(k)}\sigma_{(i)n} \tag{5-4}$$

其中，$N_{i(k)}$ 为把第 k 个高斯点坐标带入到式 4-2 中得出的结果。

由此可以建立一个单元内所有节点的第 n 个应力分量和高斯点第 n 个应力分量的对应关系：

$$\underset{(8\times1)}{\boldsymbol{\sigma}_{gn}}=\underset{(8\times8)}{\boldsymbol{N}}\underset{(8\times1)}{\boldsymbol{\sigma}_{n}} \tag{5-5}$$

其中，

$$\underset{(8\times8)}{\boldsymbol{N}}=\begin{bmatrix} N_{1(1)} & N_{2(1)} & \cdots & N_{8(1)} \\ N_{1(2)} & N_{2(2)} & \cdots & N_{8(2)} \\ \vdots & \vdots & \ddots & \vdots \\ N_{1(8)} & N_{2(8)} & \cdots & N_{8(8)} \end{bmatrix} \tag{5-6}$$

$N_{i\,(k)}$ 代表形函数 N_i 在第 k 个高斯点的值。

换言之，节点应力可表示为

$$\underset{(8\times1)}{\boldsymbol{\sigma}_n}=\underset{(8\times8)}{\boldsymbol{N}^{-1}}\underset{(8\times1)}{\boldsymbol{\sigma}_{gn}} \tag{5-7}$$

至此，我们已可以根据高斯点上的应力计算出节点上的应力。当然，对于一个六面体上的节点，其实属于 8 个单元共有的，从这 8 个单元高斯点外推出的节点应力很可能都不相同，平台会采取一些措施进行平滑处理。然而这并不是普通单元插件的一部分，所以不再深入。

5.2 节点应力计算的平台实现。

节点应力计算部分也是有限元平台单元插件的一部分重要内容。4.1 节已经介绍了思路和算法，下面我们来关注代码。特别说明，本节有很多和第三章单元刚度阵计算相同的内容，例如有些主要进行初始化，数据库提取数据等操作的方法，这些都不再作重复介绍，请读者根据代码注释和前文所介绍的内容进行理解。

5.2.1 单元应变计算工厂实现类 MHexaBrickEleStrainCalculatorFactoryImpl

该工厂完成了对高斯点应变的计算。它的写法类似于单元刚度阵计算工厂，下面也做简要介绍。

mhexabrickelestraincalculatorfactoryimpl.h（部分代码）	Description

MExtension createExtension(const QVariant& type) const
{
MExtensionManager extManager = MExtensionManager::getManager();
MElementStrainBuilder builder = extManager.createExtension("org.sipesc.fems.strain.MElementStrainBuilder");
builder.setStrainType(_global.getValue(MElementsGlobal::HexaBrickStrainMatrix));
builder.setIntegType(_global.getValue(MElementsGlobal::Gau3D2IntegForm));
return builder;
}
	◆ 方法 *createExtension*
	类似于单元刚度阵的计算，我们只需要把应变阵和积分格式两个信息 set 给 MelementStrainBuilder 的对象。有限元的流程会调用单元应变计算模块计算出高斯点的应变。

5.2.2 单元应力计算工厂实现类 MHexaBrickEleStressCalculatorFactoryImpl

该工厂完成了对高斯点应力的计算。

类似单刚和高斯点应变计算，set 到 **MelementStressBuilder** 的信息是单元本构阵和积分格式，代码略。

5.2.3 节点应力计算实现类 MHexaBrickNodeStressCalculatorImpl

mhexabricknodestresscalculatorimpl.cxx（部分代码）	Description

public:
bool initialize()
{
MObjectManager objManager = MObjectManager::getManager();
_mFactory = objManager.getObject("org.sipesc.fems.matrix.matrixfactory");
_vFactory = objManager.getObject("org.sipesc.fems.matrix.vectorfactory");
MDataFactoryManager factoryManager = objManager.getObject("org.sipesc.core.engdbs.mdatafactorymanager");
QString name = "org.sipesc.core.engdbs.MDataObjectList";
_listFactory = factoryManager.getFactory(name);
MExtensionManager extManager = MExtensionManager::getManager();
_mTools = extManager.createExtension("org.sipesc.fems.matrix.MMatrixTools");
 	
MMatrix interpolation = _mFactory.createMatrix(8, 8); //应力插值矩阵与应力外推时的转换阵
double ksi, eit, cait;
int m = 0;
for (int k = 0; k < 2; k++)
for (int j = 0; j < 2; j++)
for (int i = 0; i < 2; i++)
{
ksi = pow(-1, i) / sqrt(3);
eit = pow(-1, j) / sqrt(3);
cait = pow(-1, k) / sqrt(3);
	◆ 方法 *initialize*
	在初始化中，先计算应力插值矩阵（式 5-6 中的 N），方法参见 5.1 节。对其求逆可以得到应力外推的转换阵 _extrapolationMatrix。
	其他初始化内容类似前文代码，不再重复介绍。

```
            interpolation(m, 0, (1 - ksi) * (1 - eit) * (1 - cait) / 8);
            interpolation(m, 1, (1 + ksi) * (1 - eit) * (1 - cait) / 8);
            interpolation(m, 2, (1 + ksi) * (1 + eit) * (1 - cait) / 8);
            interpolation(m, 3, (1 - ksi) * (1 + eit) * (1 - cait) / 8);
            interpolation(m, 4, (1 - ksi) * (1 - eit) * (1 + cait) / 8);
            interpolation(m, 5, (1 + ksi) * (1 - eit) * (1 + cait) / 8);
            interpolation(m, 6, (1 + ksi) * (1 + eit) * (1 + cait) / 8);
            interpolation(m, 7, (1 - ksi) * (1 + eit) * (1 + cait) / 8);
            m++;
            }
        _extrapolationMatrix = _mTools.inverse(interpolation);
        return true;
        }
  };
```

◆ 方法 *setState*

获取单元局部坐标系下节点的数据。

```
void MHexaBrickNodeStressCalculatorImpl::setState(
     const MDataObject& localData)
  {
  _data->_coordData = localData;
  }
```

◆ 方法 *getNodeStress*

获取单元节点应力。

主要有两部分内容：

1.计算单元局部坐标下的节点应力。

2.通过坐标转换，求出总体坐标下的节点应力。

```
MDataObject MHexaBrickNodeStressCalculatorImpl::getNodeStress(
     const MDataObject& integStressData)
  {
  MDataObjectList integralPoints = integStressData;
//integralPoints ,nodePoints 分别存放传进来的高斯点应力和坐标转换前的外推节点应力
  int eleId = integStressData.getId();

  MVectorData sigmaData1 = integralPoints.getDataAt(0);
  MVector sigma1 = _data->_vFactory.createVector(6);
  sigma1 << sigmaData1;
  MVectorData sigmaData2 = integralPoints.getDataAt(1);
  MVector sigma2 = _data->_vFactory.createVector(6);
  sigma2 << sigmaData2;
  MVectorData sigmaData3 = integralPoints.getDataAt(2);
  MVector sigma3 = _data->_vFactory.createVector(6);
  sigma3 << sigmaData3;
  MVectorData sigmaData4 = integralPoints.getDataAt(3);
  MVector sigma4 = _data->_vFactory.createVector(6);
  sigma4 << sigmaData4;
  MVectorData sigmaData5 = integralPoints.getDataAt(4);
  MVector sigma5 = _data->_vFactory.createVector(6);
```

1.计算单元局部坐标下的节点应力。

首先把高斯点应力值读入并存入不同的 MVector 对象中，每个 MVector 对象的大小是 6，存的是 6 个应力分量。sigma1 到 sigma8 代表第 1 到第 8 节点的应力分量。

```
    sigma5 << sigmaData5;
    MVectorData sigmaData6 = integralPoints.getDataAt(5);
    MVector sigma6 = _data->_vFactory.createVector(6);
    sigma6 << sigmaData6;
    MVectorData sigmaData7 = integralPoints.getDataAt(6);
    MVector sigma7 = _data->_vFactory.createVector(6);
    sigma7 << sigmaData7;
    MVectorData sigmaData8 = integralPoints.getDataAt(7);
    MVector sigma8 = _data->_vFactory.createVector(6);
    sigma8 << sigmaData8;

    MVector integralVector = _data->_vFactory.createVector(8);
    MVector tmp1 = _data->_vFactory.createVector(8);
    for (int i = 0; i < 6; i++) //i 为应力分量个数
      {
      integralVector(0, sigma1(i)); //integralVector 为八个高斯点应力值的
第 i 个分量
      integralVector(1, sigma2(i));
      integralVector(2, sigma3(i));
      integralVector(3, sigma4(i));
      integralVector(4, sigma5(i));
      integralVector(5, sigma6(i));
      integralVector(6, sigma7(i));
      integralVector(7, sigma8(i));
      tmp1 = _data->_extrapolationMatrix * integralVector; //tmp1 为转
化 8 个节点应力值的第 i 个分量
      sigma1(i, tmp1(0)); //把 tmp1 中的 8 个节点应力值的第 i 个分量分别
分给八个节点应力向量
      sigma2(i, tmp1(1));
      sigma3(i, tmp1(2));
      sigma4(i, tmp1(3));
      sigma5(i, tmp1(4));
      sigma6(i, tmp1(5));
      sigma7(i, tmp1(6));
      sigma8(i, tmp1(7));
      }

    QList<MVector> nodePoints;
    nodePoints << sigma1 << sigma2 << sigma3 << sigma4 <<
sigma5 << sigma6
          << sigma6 << sigma8;
    MMatrixData mData = _data->_coordData.getTransMatrixData();
    MMatrix m = _data->_mFactory.createMatrix();
```

然后通过循环，依次把高斯点上的应力分量通过前面算出的转换矩阵转换成节点应力分量（参照式 5-5、5-6、5-7 的理论），最后储存在 nodepoints 里。注意，这里高斯点的标号要和添加的积分格式中的标号对应。

```
m << mData;

MMatrix trans = _data->_mFactory.createMatrix(6, 6);
trans(0, 0, m(0, 0) * m(0, 0));
trans(0, 1, m(0, 1) * m(0, 1));
trans(0, 2, m(0, 2) * m(0, 2));
trans(1, 0, m(1, 0) * m(1, 0));
trans(1, 1, m(1, 1) * m(1, 1));
trans(1, 2, m(1, 2) * m(1, 2));
trans(2, 0, m(2, 0) * m(2, 0));
trans(2, 1, m(2, 1) * m(2, 1));
trans(2, 2, m(2, 2) * m(2, 2));

trans(0, 3, m(0, 0) * m(0, 1));
trans(0, 4, m(0, 1) * m(0, 2));
trans(0, 5, m(0, 2) * m(0, 0));
trans(1, 3, m(1, 0) * m(1, 1));
trans(1, 4, m(1, 1) * m(1, 2));
trans(1, 5, m(1, 2) * m(1, 0));
trans(2, 3, m(2, 0) * m(2, 1));
trans(2, 4, m(2, 1) * m(2, 2));
trans(2, 5, m(2, 2) * m(2, 0));

trans(3, 0, 2.0 * m(0, 0) * m(1, 0));
trans(3, 1, 2.0 * m(0, 1) * m(1, 1));
trans(3, 2, 2.0 * m(0, 2) * m(1, 2));
trans(4, 0, 2.0 * m(1, 0) * m(2, 0));
trans(4, 1, 2.0 * m(1, 1) * m(2, 1));
trans(4, 2, 2.0 * m(1, 2) * m(2, 2));
trans(5, 0, 2.0 * m(2, 0) * m(0, 0));
trans(5, 1, 2.0 * m(2, 1) * m(0, 1));
trans(5, 2, 2.0 * m(2, 2) * m(0, 2));

trans(3, 3, m(0, 0) * m(1, 1) + m(1, 0) * m(0, 1));
trans(3, 4, m(0, 1) * m(1, 2) + m(1, 1) * m(0, 2));
trans(3, 5, m(0, 2) * m(1, 0) + m(1, 2) * m(0, 0));
trans(4, 3, m(1, 0) * m(2, 1) + m(2, 0) * m(1, 1));
trans(4, 4, m(1, 1) * m(2, 2) + m(2, 1) * m(1, 2));
trans(4, 5, m(1, 2) * m(2, 0) + m(2, 2) * m(1, 0));
trans(5, 3, m(2, 0) * m(0, 1) + m(0, 0) * m(2, 1));
trans(5, 4, m(2, 1) * m(0, 2) + m(0, 1) * m(2, 2));
trans(5, 5, m(2, 2) * m(0, 0) + m(0, 2) * m(2, 0));
```

2.通过坐标转换，求出总体坐标下的节点应力。

上面求出的节点应力是局部坐标下的，我们需要通过坐标转换阵，转换到整体坐标下。矩阵 trans 实现的就是这个功能。局部坐标是在有限元流程中建立的，坐标轴间的方向余弦矩阵通过局部坐标数据的 ***getTransMatrixData*** 方法获得，存在 m 中。这个矩阵具体如下

$$\begin{bmatrix} l_1 & m_1 & n_1 \\ l_2 & m_2 & n_2 \\ l_3 & m_3 & n_3 \end{bmatrix} \quad (5\text{-}8)$$

坐标转换阵是一个 6 乘 6 的矩阵，具体如下

$$\begin{bmatrix} l_1^2 & m_1^2 & n_1^2 & l_1m_1 & n_1m_1 & l_1n_1 \\ l_2^2 & m_2^2 & n_2^2 & l_2m_2 & n_2m_2 & l_2n_2 \\ l_3^2 & m_3^2 & n_3^2 & l_3m_3 & n_3m_3 & l_3n_3 \\ 2l_1l_2 & 2m_1m_2 & 2n_1n_2 & l_1m_2+l_2m_1 & m_1n_2+m_2n_1 & n_1l_2+n_2l_1 \\ 2l_3l_2 & 2m_3m_2 & 2n_3n_2 & l_2m_3+l_3m_2 & m_2n_3+m_3n_2 & n_2l_3+n_3l_2 \\ 2l_1l_3 & 2m_1m_3 & 2n_3n_1 & l_3m_1+l_1m_3 & m_3n_1+m_1n_3 & n_1l_3+n_3l_1 \end{bmatrix}$$

（5-9）

之后代码的内容就是将式 5-8 中的 9 个方向余弦，按照 5-9 中的公式形成坐标转换矩阵。

MDataObjectList nodeStressVectors = _data->_listFactory.createObject(); //nodeStressVectors 存放坐标转换后的节点应力
nodeStressVectors.setId(eleId);
for (int i = 0; i < 8; i++)
{
MVector tmp = nodePoints[i];
MVector nodeVector = _data->_mTools.transpose(trans) * tmp;
MVectorData nodeVectorData;
nodeVector >> nodeVectorData;
nodeStressVectors.appendData(nodeVectorData);
}
 	
return nodeStressVectors;
	最后返回坐标转换后的单元节点应力。

5.2.4 节点应力计算工厂实现类 MhexaBrickNodeStressCalculatorFactoryImpl

略。

5.3 数据导出

5.3.1 结果数据导出实现类 MUnvHexaBrickEleDataExportorImpl

该扩展主要完成对计算结果的导出，主要接口为

初始化

bool initialize(const MTask& parentTask, MDataModel& model);

根据文件名导出数据

bool dataExport(QTextStream& stream, const QString& name);

其中由于节点数据部分，对于各个单元均相同，故单元设计者只需按照 Jifex 的数据文件格式完成导出单元节点编号部分。即

```
int elementCount = eleManager.getDataCount();
for (int i = 0; i < elementCount; i++)
{
    MElementData elementData;
    elementData = eleManager.getDataAt(i);
    stream << "(" << elementData.getId() << "," << "80600, 0, 0, 0, ";
```

```
stream << elementData.getNodeId(0) << ", ";
stream << elementData.getNodeId(1) << ", ";
stream << elementData.getNodeId(2) << ", ";
stream << elementData.getNodeId(3) << ", ";
stream << elementData.getNodeId(4) << ", ";
stream << elementData.getNodeId(5) << ", ";
stream << elementData.getNodeId(6) << ", ";
stream << elementData.getNodeId(7) << ";)" << endl;
```

其中的 eleManager 为打开的单元数据，80600 为 Jifex 后处理软件识别不同单元的标识号，80600 为八节点六面体单元，其他单元的标识号参见《结构有限元分析与优化设计软件系统 Jifex 有限元分析子系统用户手册》第 5.2 节。

5.3.2 结果数据导出工厂实现类 MUnvHexaBrickEleDataExportorFactoryImpl

略。

至此，单元插件所有扩展的实现代码已经全部介绍完毕。

6. 单元插件的编译，注册方式。

6.1 修改单元插件源文件 org.sipesc.fems.example.cxx 和 cmakelist.txt 文件。

这两个文件的修改类似于《HelloWorld 插件设计向导》中的操作，具体请参照 example 示例代码中这两个文件的写法，比较简单和固定，不作过多说明。

6.2 插件的编译和注册。

在终端目录 example/tmp 下，键入 **cmake -DCMAKE_BUILD_TYPE=debug .. -DCMAKE_INSTALL_PREFIX=~/.sipesc**，如没有报错，键入 **make install** ，若无报错，则已完成插件的编译和注册，图 6-1。

```
[ 56%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/src/mbrickeleconstidparserimpl.o
[ 60%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/src/mhexabrickstrainmatriximpl.o
[ 65%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/src/mhexashpfunctionimpl.o
[ 69%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/src/mgau3d2integformimpl.o
[ 73%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/src/mhexabricknodestresscalculatorimpl.o
[ 78%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/src/mhexabricklumpedmassmatriximpl.o
[ 82%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/src/mhexabrickelevolumeimpl.o
[ 86%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/src/munvhexabrickeledataexportorimpl.o
[ 91%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/org.sipesc.fems.example.xml.o
[ 95%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/qrc_org.sipesc.fems.example.o
[100%] Building CXX object CMakeFiles/org.sipesc.fems.example_2.0.0.dir/org.sipesc.fems.example_2.0.0_automoc.o
Linking CXX shared library ../femsdev/lib/fems/org.sipesc.fems.example_2.0.0.sep
[100%] Built target org.sipesc.fems.example_2.0.0
Install the project...
-- Install configuration: "debug"
-- Installing: /home/jhello/.sipesc/lib/plugins/debug/org.sipesc.fems.example_2.0.0.sep
-- Installing: /home/jhello/.sipesc/share/translations/zh_CN
-- Installing: /home/jhello/.sipesc/share/translations/zh_CN/org.sipesc.fems.example_zh_CN.qm
-- Installing: /home/jhello/.sipesc/share/features/org.sipesc.fems.example.features
-- Installing: /home/jhello/sipescworkspace/example/femsdev/features/fems/org.sipesc.fems.example.features
jhello@jhello-VirtualBox:~/sipescworkspace/example/tmp$
```

图 6-1

7. 单元计算模块插件开发步骤总结

有限元系统在添加单元类型时，主要步骤：

1. 通过插件设计器定义单元计算模块各计算对象及接口。主要包括数据文件导入类、形函数类、应变矩阵类、本构解析类、积分点类、单元刚度计算工厂类、单元应变计算工厂类、单元应力计算工厂类、节点应力计算类以及各对象相应的工厂类等。
2. 通过插件设计器导出生成插件头文件、源文件、资源文件、接口文件和 cmakelist.txt 文件。
3. 在编译器中建立相应单元插件具体实现的工程，并编写实现代码，完成对每个接口的实现，并定义单元数据类。
4. 通过编译和注册，生成单元计算插件

8. 附录

8.1 平台有限元文件配置方法。

1.可以直接拷取新版本的虚拟机，里面有最新版的平台及有限元模块。

2.如果已有虚拟机，先要在新立得或者其他软件包管理器中安装平台，再将共享中的新版平台中的有限元中文件的子文件文件夹中的文件拷贝到 home 下的隐藏文件夹.sipesc 中的对应位置。

8.2 平台有限元静力分析流程。

1） 节点单元映射

任务类名称：org.sipesc.fems.femstask.MNodeMapManager
任务描述：得到每个节点对应的单元。
存储位置：根目录 model——NodeEleMap 文件
存储类型：MNodeMapData

2） 节点排序

任务类名称：org.sipesc.fems.femstask.MNodeSortBySpectrum
任务描述：通过对节点进行排序控制带宽。在有限元计算过程中，减少带宽不但可以减少总刚的存储量，还可以大幅度减少求解运算次数。
存储位置：根目录 model——NodeSort 文件
存储类型：MByteArray——QList<int>

3） 自由度解析

任务类名称：org.sipesc.fems.controlmatrix.MDofStandardParserManager
任务描述：对节点约束信息进行解析，判断每个节点的自由度状态，如自由，固定约束，指定位移等。
存储位置：根目录 model——NodeDofDataPath 文件夹——JIFEX 文件
存储类型：

4） 自由度排序

任务类名称：org.sipesc.fems.controlmatrix.MDofNumberingStandardManager
任务描述：通过每个节点的自由度状态以及节点排序，对号形成所有节点自由度的整体排序，分为两类：自由状态的自由度排序，以及指定位移的自由度排序（固定约束的自由度不进入总刚）。即为总刚度阵的自由度排序，为集成总刚做准备。
存储位置：根目录 model——DofNumber 文件
存储类型：

5） 节点控制阵

任务类名称：org.sipesc.fems.controlmatrix.MNodeControlMatrixStandardManager
任务描述：为了处理各种各样的自由度约束关系（如位移约束，多点约束等），需要使用一种节点控制阵的概念来描述节点每个自由度对总刚度阵的贡献，同时节点的局部坐标系也可通过节点控制阵来实现。
存储位置：根目录 model——NodeControlMatrix 文件

存储类型：MNodeControlMatrixData

6） 单元局部坐标

任务类名称：org.sipesc.fems.controlmatrix.MElementLocalCoordTransStandardManager

任务描述： 得到每一个单元的局部坐标转换矩阵

存储位置：根目录 model——ElementLocalCoordPath 文件夹——QuadMembElement 文件（根据单元类型分别储存）

存储类型：MElementLocalCoordData

7） 单元控制阵

任务类名称：org.sipesc.fems.controlmatrix.MElementControlMatrixStandardManager

任务描述：是由节点控制阵和单元局部坐标阵组合而成，为了将单元局部坐标系下的单元刚度阵变换到全局坐标系下的总刚度阵做准备，单刚可以通过左乘右乘单元控制阵累加到总刚上。

存储位置：根目录 model——ElementControlMatrix 文件

存储类型：MElementControlMatrixData

8） 单元应变矩阵

任务类名称：org.sipesc.fems.femstask.MElementGradMatrixManager

任务描述： 计算得到单元的高斯点应变矩阵以及高斯点雅克比矩阵行列式值。

8.1） 高斯点应变矩阵

存储位置：根目录 model——ElementGradMatrix 文件

存储类型：MDataObjectList（单元）——MMatrixData（高斯点）

8.2） 高斯点雅可比矩阵行列式值

存储位置：根目录 model——ElementJacobiValue 文件

存储类型：MVectorData

9） 单元刚度矩阵

任务类名称：org.sipesc.fems.femstask.MSolidEleStiffManager

任务描述： 通过数值积分计算得到单元刚度矩阵（由材料弹性矩阵、高斯点应变矩阵、高斯点雅克比矩阵行列式值以及权系数加权得到）。

存储位置：根目录 model——ElementStiffMatrix 文件

存储类型：MElementStiffMatrixData（继承于 MMatrixData）

10） 单元质量矩阵

任务类名称：org.sipesc.fems.femstask.MElementMassManager

任务描述：计算得到单元质量矩阵。

存储位置：根目录 model——ElementMassMatrix 文件

存储类型：MElementStiffMatrixData

11） 指定位移载荷向量

任务类名称：org.sipesc.fems.femstask.MGivenValueLoadVectorManager

任务描述：由于指定位移自由度并不对总刚引入贡献，但却会产生相应的载荷，此任务即为指定位移载荷向量的形成。

存储位置：根目录 model——GivenValueLoadVector 文件

存储类型：MVectorData

12） 载荷分量

任务类名称：org.sipesc.fems.femstask.MLoadComponentManager

任务描述：通过各个工况的载荷调用情况，形成载荷作用点的载荷分量数据。

存储位置：根目录 model——LoadComponent 文件
存储类型：MVectorData

13） 整体载荷向量

任务类名称：org.sipesc.fems.femstask.MLoadManager
任务描述：对载荷分量数据以及指定位移形成的载荷进行累加，形成整体载荷向量。
存储位置：根目录 model——LoadVector 文件
存储类型：MVectorData

14） 求解

任务类名称：org.sipesc.fems.femstask.MResultsManager
任务描述： 此任务包含多个子任务，包括累加形成总体刚度阵、总体刚度阵的主元地址形成、总体刚度阵的 LDLT 分解、以及回代求解得到各个工况下的位移结果（此位移结果是在节点局部坐标系下的，并非最终结果，即不包含约束的位移结果）。
存储位置：根目录 model——ResultVector 文件
存储类型：MVectorData

15） 位移转换

任务类名称：org.sipesc.fems.femstask.MDisplacementsManager
任务描述：形成最终后处理显示的位移结果，即通过节点局部坐标系下的位移结果左乘节点控制阵。
存储位置：根目录 model——
存储类型：

16） 节点应力

任务类名称：org.sipesc.fems.stress.MSolidNodeStressManager
任务描述：计算得到节点应力
存储位置：根目录 model——
存储类型：

17） Jifex 输出

任务类名称：org.sipesc.fems.jfxexport.MJifexUnvExportor
任务描述：将结果输出到 Jifex 后处理文件.unv 中
存储位置：有限元文件所在目录——.unv 文件
存储类型：.unv 文件格式