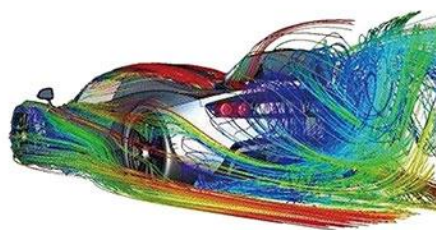
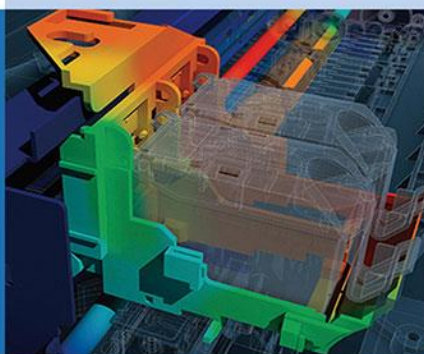
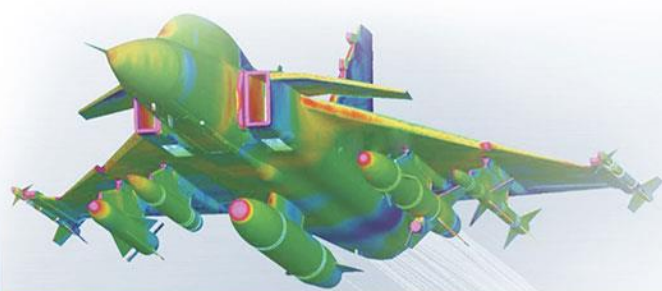




FastCAE 2.0

用户开发手册



FastCAE研发小组
2020年1月

引言

本目录下包含了与 FastCAE 使用相关的四个文档——《代码结构与 API 说明》，《Python 命令说明》，《Python 拓展》，《插件开发说明》。您可以根据需要查看这些文档，由于文档数量多内容繁杂，下面我们将为您介绍一下各文档的大概内容，以方便您查阅。

《代码结构与 API 说明》详细介绍了 FastCAE 代码结构与设计思想，将代码中重要的 API 进行了汇总说明。

《Python 命令说明》介绍了目前 FastCAE 平台封装的 Python 接口，以及各个接口的功能。

《Python 功能拓展》介绍了如何利用 Python 语言进行自定义的功能拓展，以及如何调用新拓展的功能，并通过具体示例具体说明。

《插件开发指南》介绍了 FastCAE 开发的插件接口和插件类型，并通过具体示例的形式对开发流程进行说明。

➤ 如果您觉得 FastCAE 能够满足您的一部分需要，但是需要进行一些拓展，您可以参考《插件开发指南》和《Python 功能拓展》。

➤ 如果您对 C++ 和 Python 都比较熟悉，想在 FastCAE 的基础上进行深度定制与联合开发，您可以参考《Python 命令说明》《Python 功能拓展》和《代码结构与 API 说明》

页码分配如下：

《代码结构与 API 说明》	1
《Python 命令说明》	48
《Python 功能拓展》	64
《插件开发指南》	69



代码结构与 API 说明

FastCAE 研发小组

2020 年 1 月

目录

一、代码总览.....	1
二、数据类组织关系及 API.....	1
2.1 数据基类组织关系.....	2
2.2 数据基类 API.....	4
2.3 几何数据组织关系.....	12
2.4 几何数据 API.....	14
2.5 网格数据组织关系.....	16
2.6 网格数据 API.....	18
2.7 模型数据组织关系.....	21
2.8 模型数据 API.....	22
2.9 其他数据.....	27
2.10 其他数据 API.....	27
三、视图类组织关系及 API.....	29
3.1 主界面.....	30
3.2 控制台与进度窗口.....	33
3.3 控制面板.....	33
3.4 几何网格树形菜单.....	34
3.5 前后处理窗口.....	36
3.6 算例树形菜单.....	44
四、控制类相关说明.....	46
4.1 几何建模命令.....	46
4.2 输入输出.....	47
4.3 Gmsh 集成.....	47

一、代码总览

FastCAE 采用 C++ 语言编写，基于 Qt5.4.2 搭建 GUI 图形交互界面，支持跨平台运行。现阶段全部代码超过 20 万行，全部代码编译为 1 个可执行文件和 29 个动态链接库。全部的类可以分为三大类：数据类（模型类），视图类和控制类。三者相互关联，相互作用。

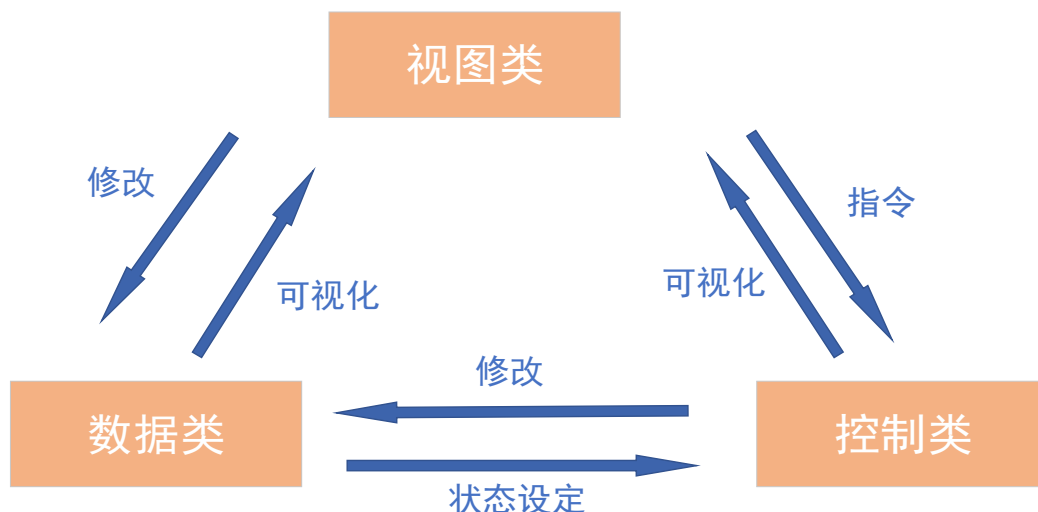


图 1 FastCAE 三大类

数据类是指在软件运行过程中的操作的全部数据源，包括网格数据，几何数据，模型数据，以及其他的设定项参数等。数据类是软件运行的最底层，也是操作和处理的最终对象和载体。

控制类是指在运行过程中对数据进行操作的一类，负责接受来自视图类的用户输入指令对数据进行创建修改和删除等操作。控制类是中间类，较为有代表性的为几何模型创建与编辑修改命令，求解控制，数据的导入导出类等。

视图类是软件运行过程中与用户直接交互的类，负责实现数据的可视化展示，以及接受用户的操作指令，并将相关的修改和命令下发给数据类和控制类。视图类是人机交互的最前端，在整个软件结构中处于高层。较为有代表性的是主窗口，各个绘图窗口和各种弹出对话框等。

二、数据类组织关系及 API

软件中的数据主要可以分为五部分，几何数据，网格数据，模型数据，配置

数据和环境设定参数。五部分数据通过五个数据单例进行管理维护。最主要的参数是前面三类。本部分将着重介绍前面三类。

2.1 数据基类组织关系

几何数据、网格数据、模型数据、配置数据这些数据类及其成员都有共同的基类：*DataProperty:: DataBase*。这个类里面包含了属性和参数两种类型的成员列表，属性是指在软件运行的过程中进行分配的，不允许随意更改的信息，例如 ID，可见性，名称等。参数是指允许用户修改的信息，例如材料属性相关的参数以及物理模型设置相关的信息。在软件运行的过程中，控制面板下方的属性窗口会有明显的可视化区分，如下图：

名称	值
基础信息	
ID	1
Type	测试材料
Name	Material_1

参数	
参数组1	
布尔参数	<input checked="" type="checkbox"/>
字符串	TestChar
枚举选择	选项2
参数组2	
表格类型	Row:2, Col:3
路径参数	
整数参数	0
浮点型参数	100.0000

图 2 属性窗口

数据基类的代码位于 *DataProperty* 文件夹，组织与继承关系如下：

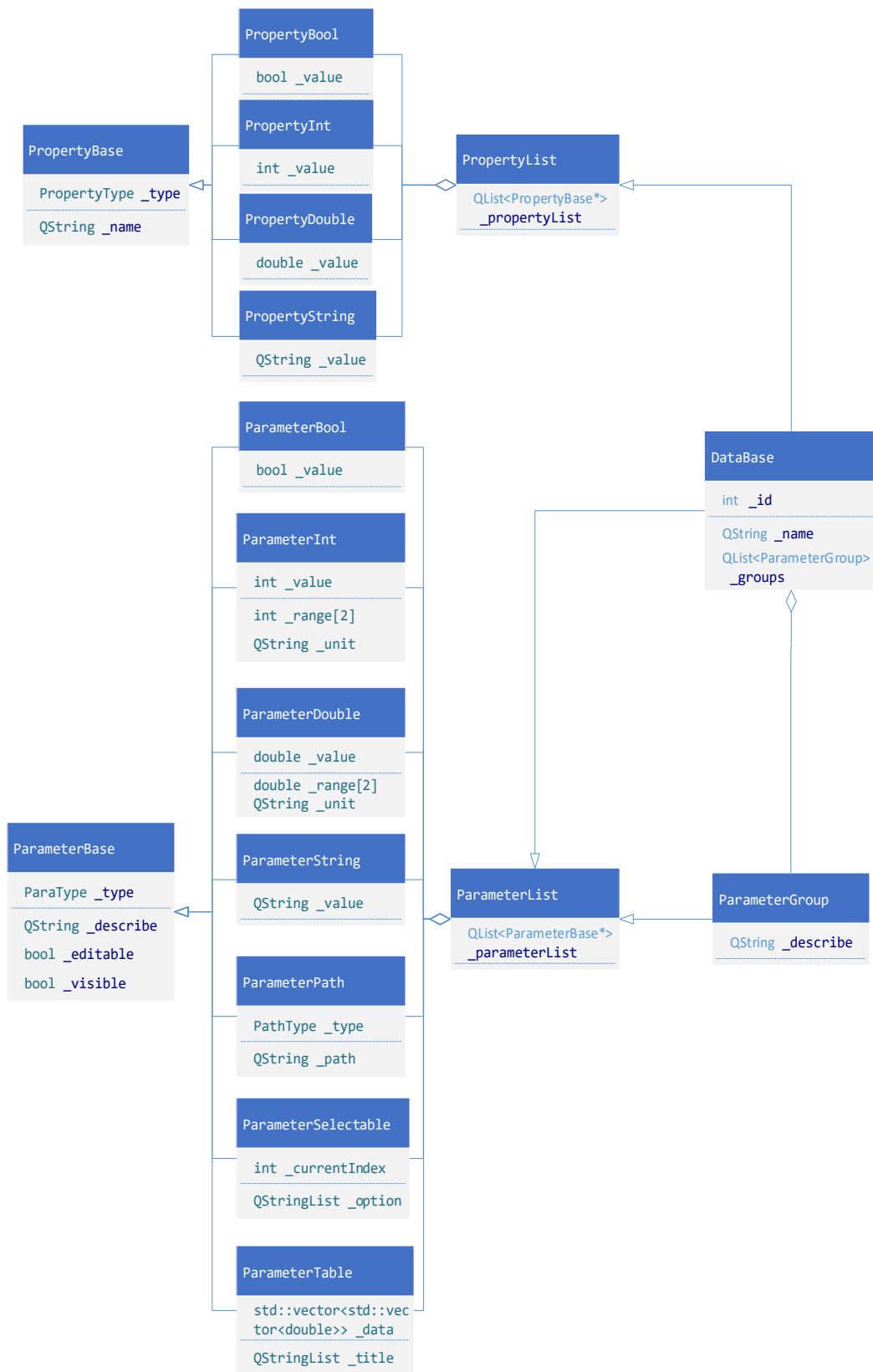


图 3 组织与继承关系

2.2 数据基类 API

各数据类的命名空间为 `DataProperty`，其中包含参数和属性两部分的类的定义，他们各自通过枚举类型进行标记，枚举类型的定义和主要 API 如下：

```
enum PropertyType
{
    Prop_Blank = 0,
    Prop_INT,
    Prop_Double,
    Prop_Color,
    Prop_String,
    Prop_Bool,
    Prop_Point,
    Prop_IDList,
};

enum ParaType
{
    Para_Blank = 0,
    Para_Int,
    Para_Double,
    Para_Color,
    Para_String,
    Para_Bool,
    Para_Selectable,
    Para_Path,
    Para_Table,
};
```

表 1 数据基类主要 API

类名	父类	主要 API
<i>PropertyBase</i> 属性基类		//构造函数 PropertyBase(PropertyType type); //设置类型 void setType(PropertyType tp); //获取类型 PropertyType getType(); //设置名称 void setName(const QString &name); //获取名称 QString getName(); //类型转换 virtual QVariant getVariant();
<i>PropertyBool</i> 布尔类型属性	PropertyBase	//构造函数 PropertyBool(); //构造函数 PropertyBool(QString name, bool val); //设置值 void setValue(bool v); //获取值 bool getValue(); //类型转换

		<code>QVariant getVariant() override;</code>
<i>PropertyDouble</i> 浮点数类型属性	<u>PropertyBase</u>	//构造函数 <code>PropertyDouble();</code> //构造函数 <code>PropertyDouble(QString name, double val);</code> //设置值 <code>void setValue(double v);</code> //获取值 <code>double getValue();</code> //类型转化 <code>QVariant getVariant() override;</code>
<i>PropertyInt</i> 整数类型属性	<u>PropertyBase</u>	//构造函数 <code>PropertyInt();</code> //构造函数 <code>PropertyInt(QString name, int val);</code> //设置值 <code>void setValue(int v);</code> //获取值 <code>int getValue();</code> //类型转换 <code>QVariant getVariant() override;</code>
<i>PropertyString</i> 字符串类型属性	<u>PropertyBase</u>	//构造函数 <code>PropertyString();</code> //构造函数 <code>PropertyString(QString name, QString val);</code> //设置值 <code>void setValue(QString v);</code> //获取值 <code>QString getValue();</code> //类型转换 <code>QVariant getVariant() override;</code>
<i>PropertyList</i> 列表类型属性		//获取属性个数 <code>int getPropertyCount();</code> //获取第 index 个属性 <code>PropertyBase* getPropertyAt(const int index);</code> //通过名字获取属性 <code>PropertyBase* getPropertyByName(const QString &name);</code> //拷贝 propList 的内容 <code>void copy(PropertyList* propList);</code> //通过不同的方法添加属性 name 为属性名称，第二个参数为属性的值

		<pre> void appendProperty(QString name, int value); void appendProperty(QString name, double va lue); void appendProperty(QString name, QString s tring); void appendProperty(QString name, bool valu e); void appendProperty(QString name, double *c); void appendProperty(QString name, double x, double y, double z); </pre>
<i>ParameterBase</i> 参数基类	QObject	<pre> ParameterBase(ParaType t); ParameterBase() = default; ~ParameterBase() = default; //从 ori 中拷贝内容，子类重写 virtual void copy(ParameterBase* ori); //获取类型 ParaType getParaType(); //设置类型 void setParaType(ParaType type); //获取名称 QString getDescribe(); //设置名称 void setDescribe(QString s); //是否可编辑修改 bool isEditable(); //设置是否可见 void setVisible(bool v); //获取可见状态 bool isVisible(); //设置中文名称，与 setDescribe()对应 void setChinese(QString chinese); //获取中文名称 QString getChinese(); //将值转化为字符串，子类重写 virtual QString valueToString(); //从字符串读取设置数据，子类重写 virtual void setValueFromString(QString v); //设置模块类型 void setModuleType(ModuleType t); //获取模块类型 ModuleType getModuleType(); //获取数据 ID(DataBase) </pre>

		<pre> int getDataID(); //设置数据 ID void setDataID(int id); //获取数据索引 int getDataIndex(); //设置数据索引 void setDataIndex(int index); //获取所在的组名称 QString getGroupName(); //设置组的名称 void setGroupName(QString group); //参数组名称/参数名称 QString genAbsoluteName(); //记录观察者 void appendObserver(ConfigOption::Parameter Observer* obs); //获取观察者列表 QList<ConfigOption::ParameterObserver*> get ObserverList(); //判断数值是否一样, 子类重写 virtual bool isSameValueWith(ParameterBase* p); //状态拷贝, 子类重写 virtual void copyStatus(ParameterBase* p); ///数据写入工程文件, 子类重写 virtual void writeParameter(QDomDocument* d oc, QDomElement* parent); ///从工程文件读入数据, 子类重写 virtual void readParameter(QDomElement* e); //类型转化为字符串, 子类重写 static QString ParaTypeToString(ParaType t) ; //字符串转化为参数类型 static ParaType StringToParaType(QString st ype); //信号, 参数值发生变化 void dataChanged(); </pre>
<i>ParameterInt</i> 整数参数	<u>ParameterBase</u>	<pre> //设置值 void setValue(int v); //获取值 int getValue(); //设置范围 void setRange(int* range); //获取范围 </pre>

		<pre> void getRange(int* range); //设置单位(量纲) void setUnit(QString unit); //获取单位(量纲) QString getUnit(); //通过字符串设置值 void setValueFromString(QString v); </pre>
<i>ParameterBoolean</i> 布尔型参数	ParameterBase <u>e</u>	<pre> //设置值 void setValue(bool ok); //获取值 bool getValue(); </pre>
<i>ParameterDouble</i> 浮点数参数	ParameterBase <u>e</u>	<pre> //设置值 void setValue(double v); //获取值 double getValue(); //设置值的有效范围 void setRange(double* range); //获取值的有效范围 void getRange(double* range); //设置精度, 小数点后位数 void setAccuracy(int a); //获取精度 int getAccuracy(); //设置单位(量纲) void setUnit(QString u); //获取单位(量纲) QString getUnit(); </pre>
<i>ParameterPath</i> 路径类型参数, 有路径、文件、文件几何三种子类型	ParameterBase <u>e</u>	<pre> //设置类型 Path, File, FileList void setType(PathType t); //获取类型 PathType getType(); //设置后缀, 类型为 File, FileList 有效 void setSuffix(QString s); //获取文件后缀 QString getSuffix(); //设置文件, 类型为 File 时有效 void setFile(QString f); //获取文件 QString getFile(); //设置文件列表, 类型为 FileList 时有效 void setFileList(QStringList sl); //获取文件列表 QStringList getFileList(); </pre>

		//设置路径, 类型为 <code>Path</code> 时有效 <code>void setPath(QString s);</code> //获取路径 <code>QString getPath();</code>
<i>ParameterSelectable</i> 固定选项参数类型	<u>ParameterBase</u> <u>e</u>	//设置可选项 <code>void setOption(QStringList s);</code> //获取可选项 <code>QStringList getOption();</code> //设置当前索引 <code>void setCurrentIndex(const int index);</code> //设置当前索引 <code>int getCurrentIndex();</code>
<i>ParameterString</i> 字符串参数类型	<u>ParameterBase</u> <u>e</u>	//获取值 <code>QString getValue();</code> //设置值 <code>void setValue(QString s);</code>
<i>ParameterTable</i> 表格参数类型	<u>ParameterBase</u> <u>e</u>	//设置行数 <code>void setRowCount(const int n);</code> //设置列数 <code>void setColumnCount(const int n);</code> //获取行数 <code>int getRowCount();</code> //获取列数 <code>int getColumnCount();</code> //获取第 <code>index</code> 行的数据 <code>QList<double> getRow(int index);</code> //获取第 <code>index</code> 列数据 <code>QList<double> getColumn(int index);</code> //获取第 <code>row</code> 行 第 <code>col</code> 列数据 <code>double getValue(int row, int col);</code> //将第 <code>row</code> 行 第 <code>col</code> 列数据设置为 <code>v</code> <code>void setValue(int row, int col, double v);</code> //设置表头 <code>void setTitle(QStringList t);</code> //获取表头 <code>QStringList getTitle();</code> //获取全部数据 <code>std::vector<std::vector<double>> getData();</code> //设置全部数据 <code>void setData(std::vector<std::vector<double>> data);</code>
<i>ParameterList</i>		//从 <code>data</code> 中拷贝参数列表

<p><i>t</i></p> <p>参数列表</p>		<pre> void copy(ParameterList* data); //添加参数 virtual void appendParameter(ParameterBase* para); //根据类型创建新的参数，并放入列表 virtual ParameterBase* appendParameter(Para Type type); //获取第 i 个参数 ParameterBase* getParameterAt(const int i); //获取参数个数 int getParameterCount(); //获取可见的参数个数 int getVisibleParaCount(); //移除参数 void removeParameter(ParameterBase* p); //移除第 i 个参数 void removeParameterAt(int i); //获取第 i 个可见的参数 ParameterBase* getVisibleParameterAt(const int i); //根据类型创建新的参数，不会放入参数列表 static ParameterBase* createParameterByType (ParaType t); //根据类型创建新的参数，不会放入参数列表 static ParameterBase* createParameterByType (QString stype); //根据创建新参数，并拷贝 p 的内容，不会放入参数 //列表 static ParameterBase* copyParameter(Paramet erBase* p); ///写出参数数据 virtual void writeParameters(QDomDocument* doc, QDomElement* parent); //读入数据 virtual void readParameters(QDomElement* el e); //根据名称获取参数 virtual ParameterBase* getParameterByName(Q String name); //获取全部参数列表 QList<ParameterBase*> getParaList(); </pre>
<p><i>ParameterGroup</i></p>	<p><u>ParameterList</u></p> <p><u>t</u></p>	<pre> //拷贝 data 中的内容 void copy(ParameterGroup* data); //设置参数组的名称 </pre>

参数组		<pre> void setDescribe(QString des); //获取参数组名称 QString getDescribe(); //设置可见性 void setVisible(bool v); //获取可见状态 bool isVisible(); //拷贝 g 的状态 void copyStates(ParameterGroup* g); </pre>
<i>DataBase</i> 所有数据类 型的基类	<u>QObject</u> <u>PropertyBase</u> <u>ParameterLis</u> <u>t</u>	<pre> //设置数据所属的模块 void setModuleType(ModuleType t); //获取数据所属的模块 ModuleType getModuleType(); //从 data 中拷贝内容 virtual void copy(DataBase* data); //设置数据 ID virtual void setID(int id); //获取数据 ID int getID(); //设置数据索引 void setIndex(int index); //获取数据索引 int getIndex(); //设置名称 virtual void setName(const QString& name); //获取名称 QString getName(); //添加参数组 void appendParameterGroup(ParameterGroup* g); //获取第 index 个参数组 ParameterGroup* getParameterGroupAt(const int index); //获取参数组个数 int getParameterGroupCount(); //移除参数组 void removeParameterGroup(ParameterGroup* g); //移除第 i 个参数组 void removeParameterGroupAt(int i); /*用于产生 MD5 */ virtual void dataToStream(QDataStream* data) override; ///数据写入工程文件 </pre>

		<pre> virtual QDomElement& writeToProjectFile(QDomDocument* doc, QDomElement* parent); ///从工程文件读入数据 virtual void readDataFromProjectFile(QDomElement* e); ///读取参数 virtual void readParameters(QDomElement* ele); ///写出参数 virtual void writeParameters(QDomDocument* doc, QDomElement* parent); ///根据名称获取参数 ParameterBase* getParameterByName(QString name) override; ///根据名称获取参数组 virtual ParameterGroup* getParameterGroupByName(QString name); ///是否需要 button bool isContainsButton(); ///获取 button 英文 QStringList getButtonText(); ///获取 button 中文 QStringList getButtonChinese(); ///获取 button 链表 QList<ButtonInfo> getButtonList(); //填充所有参数信息，包括模块，ID Index 等 virtual void generateParaInfo(); </pre>
--	--	---

2.3 几何数据组织关系

几何数据是所有几何相关操作的最终操作对象，保存了操作过程的大多数源数据，包括导入几何文件，以及几何建模过程中产生的所有数据。几何模块基于 OpenCasCade 编写，因此在数据结构中涉及到了 OCC 的数据结构。当前版本采用的 OCC 版本为 7.2.0。与几何相关的类组织结构如下图：

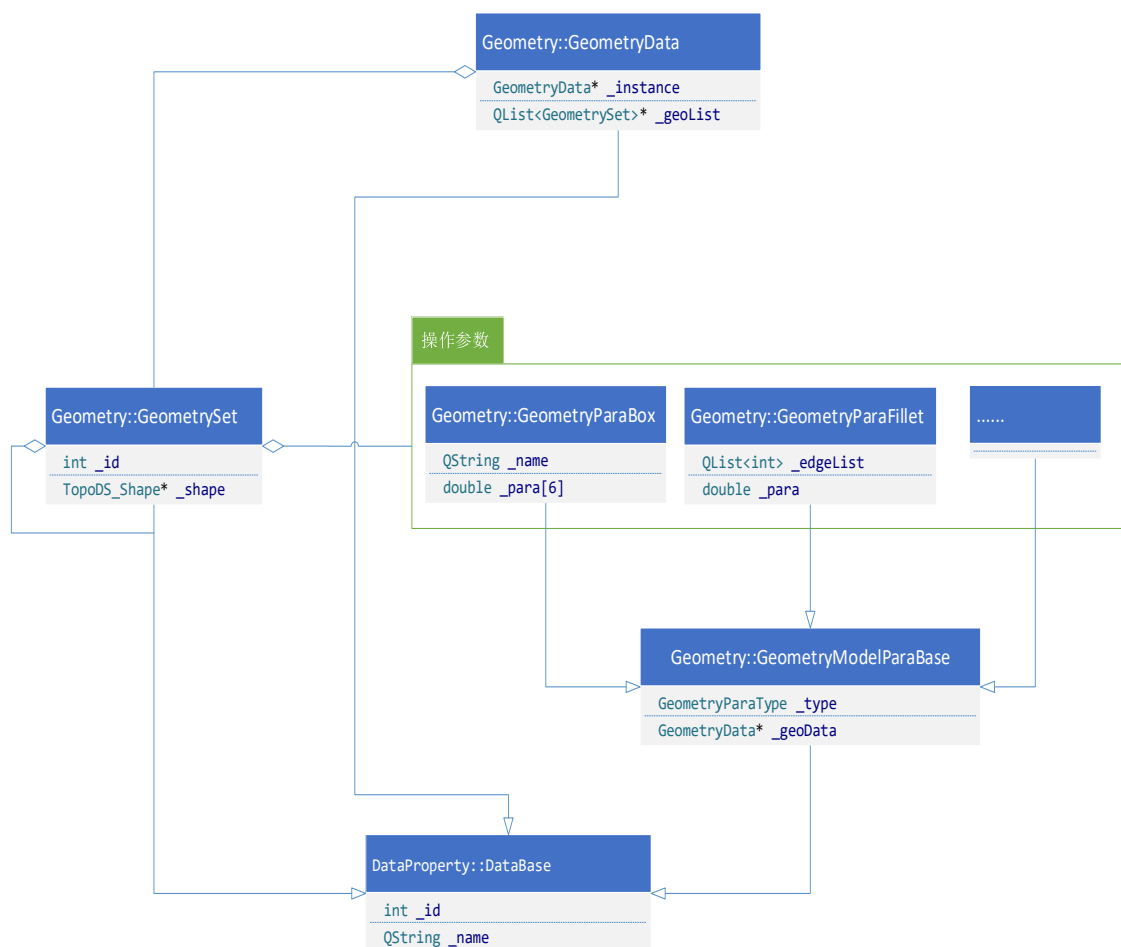


图 4 几何数据组织关系

几何数据类定义在 **Geometry** 文件夹，正如前面提到的，几何数据类的基类为 **DataBase**。几何的数据类分为三部分，首先是 **GeometrySet** 类，这个类是整个几何数据的核心，它用来表示某一个具体的形状，可以是一个点，一个曲面，一个实体，也可以表示上述元素组成的复杂几何形状。几何形状的核心表达采用 OCC 的数据类 **TopoDS_Shape** 表示，这个数据类除了可以表示具体的空间几何信息，还可以表达不同几何元素之间的拓扑关系。**GeometrySet** 类通过递归包含的方式表示数据的来源隶属关系。

另外一部分是操作参数，用于记录建模过程的参数，可以用于模型的编辑修改，他们的基类是 **GeometryModelParaBase**，每一个操作都会对应一个操作参数的类，用于记录不同操作的不同输入。每一步操作都会生成一个新的 **GeometrySet**，所以 **GeometrySet** 中包含了一个 **GeometryModelParaBase** 的子类指针来记录该步操作的参数。

每一步操作生成的 GeometrySet，都交给 GeometryData 进行管理维护，GeometryData 是一个单例类，不需要依赖于其他类对象，这样能够很好的保证数据的独立性，由于单例类方便访问的优点，该类也作为几何数据访问的总接口，其他所有几何数据都可以通过指针的形式层层访问修改。

对应到控制面板的几何标签页下，几何根节点和基准根节点下的每一个子节点都表示一个 GeometrySet。

2.4 几何数据 API

表 2 几何数据主要 API

类名	父类	主要 API
<i>GeometrySet</i> 几何形状表示	DataBase	<pre> //构造函数 GeometrySet(GeometryType type = NONE, bool needID = true); //删除全部子形状并释放子形状内存 void releaseSubSet(); //重置最大 ID，不要轻易调用 static void resetMaxID(); //根据 ID 获取子形状 GeometrySet* getSetByID(int id); //设置可见性 void setVisible(bool v); //形状是否可见 bool isVisible(); //设置类型 void setType(GeometryType type); //获取类型 GeometryType getType(); //设置形状拓扑 void setShape(TopoDS_Shape* shape); //获取形状拓扑 TopoDS_Shape* getShape(); //获取最大 ID static int getMaxID(); //设置操作参数 void setParameter(GeometryModelParaBase* p); //获取操作参数 GeometryModelParaBase* getParameter(); //移除子形状 void removeSubSet(GeometrySet* set); </pre>

		<pre> //添加子形状 void appendSubSet(GeometrySet* set); //获取子形状数目 int getSubSetCount(); //获取第 index 个子形状 GeometrySet* getSubSetAt(int index); //写出 brep 文件，路径不能出现中文 bool writeBrep(QString name); //读入 brep 文件，路径不能出现中文 bool readBrep(QString name); </pre>
<p><i>GeometryModelParaBase</i></p> <p>操作参数基类</p> <p>(这里不再介绍子类细节)</p>	DataBase	<pre> //构造函数 GeometryModelParaBase(); //获取类型 GeometryParaType getParaType(); //通过字符串创建操作参数 static GeometryModelParaBase* createParaByString(QString s); </pre>
<p><i>GeometryData</i></p> <p>几何数据管理基类，单例</p>	DataBase	<pre> //获取单例指针 static GeometryData* getInstance(); //添加形状 void appendGeometrySet(GeometrySet* set); //添加基准 void appendGeometryDatum(GeometryDatum* datum); //获取全部基准 QList<GeometryDatum*> getGeometryDatum(); //获取形状数量 int getGeometrySetCount(); //几何数据是否为空 bool isEmpty(); //获取第 index 个形状 GeometrySet* getGeometrySetAt(const int index); //移除第 index 个形状 void removeGeometrySet(const int index); //替换形状 void replaceSet(GeometrySet* newset, GeometrySet* oldset); //移除最顶层的形状，区别于递归包含的子形状 void removeTopGeometrySet(GeometrySet* set); //是否存在这个形状，递归子形状全部查找 </pre>

		<pre>bool hasGeometrySet(GeometrySet* set); //移除基准 void removeGeometryDatum(GeometryDatum* datum); //设置可见性, 不涉及子形状 void setVisible(int index, bool visible); //清空全部数据 void clear(); //根据 ID 查找形状, 递归子形状查找 GeometrySet* getGeometrySetByID(const int id); //获取第 index 个子形状 GeometryDatum* getDatumPlaneByIndex(const int index); //根据 ID 排序 void sort(); //设置草绘平面 void setSketchPlane(double* loc, double* dir); //获取草绘平面 gp_Ax3* getSketchPlane();</pre>
--	--	--

2.5 网格数据组织关系

网格数据保存了所有的网格的节点、连接关系与组件信息。网格模块对数据的检索要求很高, 需要既能够根据节点查找单元, 也能够根据单元查找节点, 为了方便我们直接采用了 VTK 的网格表示方法。支持多种网格类型, 支持低阶与高阶网格单元表示, 而且 VTK 提供了多种的网格处理算法与转换方式, 我们可以直接调用。当前采用的 VTK 版本为 7.1.1。

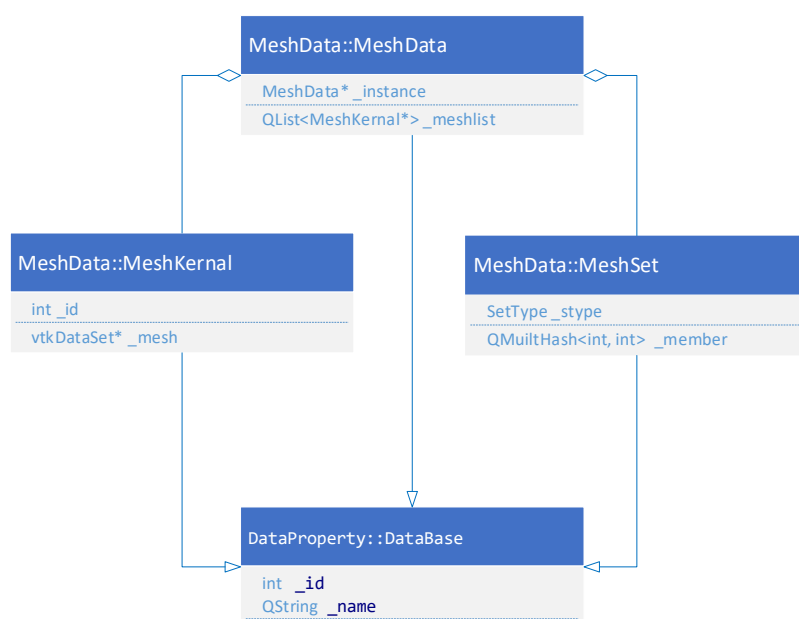


图 5 网格数据组织关系

网格部分的数据结构比较简单，和前面说的几何数据一样，他们都是 `DataBase` 的子类。网格数据部分主要包含三个类：`MeshKernal`、`MesheSet` 和 `MeshData`，代码位于 `meshData` 文件夹。`MeshKernal` 为网格数据的核心，表示了一块独立的网格，包括节点和拓扑关系，每次导入一个文件或者进行一次网格剖分会生成一个 `MeshKernal` 的实例。`MeshSet` 为网格组件，网格组件有两种类型：单元类型和节点类型，组件是网格上的一部分，通过 `Kernal` 的 ID 以及元素（单元或者节点）的索引进行标记。组件可以包含多个 `Kernal` 的元素，也可以只包含一个 `Kernal` 的元素。`MeshData` 是一个单例类，与几何数据相同，也是数据访问的接口，通过这个单例可以方便的层次化访问数据信息。

在 GUI 表示层面，在控制面板的网格标签页下，网格根节点下的每一个子节点对应一个 `MeshKernal` 实例，前面的复选框表示可见状态。下面的组件根节点下的每一个子节点对应一个 `MeshSet` 实例，根据不同的图标可以区分组件的类型。单击节点，属性框会显示对应实例的属性信息，同时在绘图区域会高亮相应的网格区域。

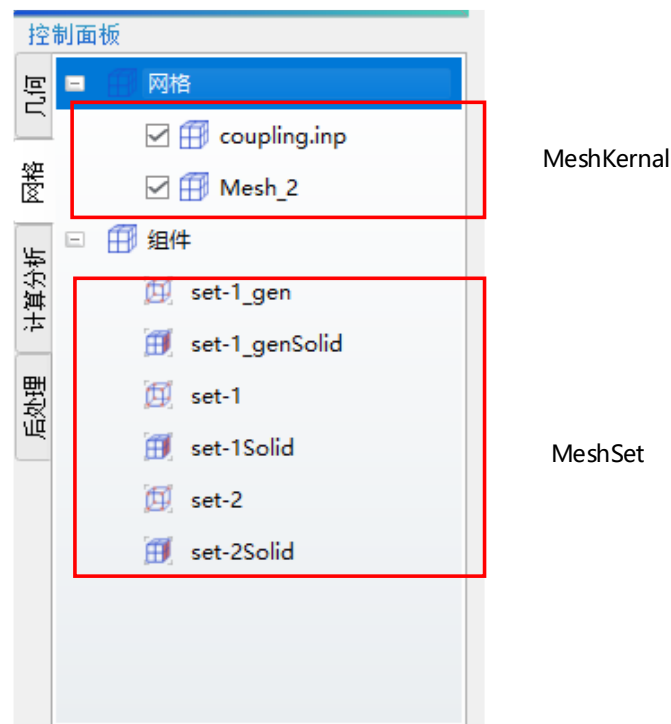


图 6 控制面板

2.6 网格数据 API

表 3 网格数据 API

类名	父类	主要 API
<i>MeshKernal</i> 网格单元和 拓扑表示	DataBase	<pre> //重置节点和单元的偏移量 static void resetOffset(); //设置网格条数据，包括节点和拓扑 void setMeshData(vtkDataSet* dataset); //获取网格数据，vtk 表示 vtkDataSet* getMeshData(); //获取节点位置，double[3],index 从 0 开始，不加偏移 double* getPointAt(const int index); //获取单元，index 从 0 开始，不加偏移 vtkCell* getCellAt(const int index); //是否可见 bool isVisible(); //设置可见状态 void setVisible(bool v); //获取节点数目 int getPointCount(); //获取单元数量 </pre>

		<pre> int getCellCount(); void setPath(const QString& path); QString getPath(); //标注网格维度 void setDimension(int d); //获取网格维度 int getDimension(); //写出二进制文件 void writeBinaryFile(QDataStream* dataStream); //读入二进制文件 void readBinaryFile(QDataStream* dataStream); </pre>
<p><i>MeshSet</i></p> <p>网格组件表示</p>	<p><u>DataBase</u></p>	<pre> //构造函数 MeshSet(QString name, SetType type); MeshSet(); //获取最大 ID int static getMaxID(); //重置最大 ID void static resetMaxID(); ///设置 ID, 谨慎调用 void setID(int id) override; ///设置类型 void setType(SetType t); ///获取类型 SetType getSetType(); //添加成员 void appendMember(int ker, int id); //获取 Kernal ID 列表 QList<int> getKernals(); //根据 kernal ID 获取成员 QList<int> getKernalMembers(int k); //获取数量 int getAllCount(); //临时保存 MemberID, 当 void setKeneralID(int id) //时清空, 指定为 Keneral 为 id 的的子集 void appendTempMem(int m); //设置 Kenenal, 与 void appendTempMem(int m)配合 //使用 void setKeneralID(int id); //是否包含 kernal bool isContainsKernal(int id); //合并组件 void merge(MeshSet* set); //减去组件 void cut(MeshSet* set); </pre>

		<pre> //写出二进制文件 virtual void writeBinaryFile(QDataStream* dataStream); //读入二进制文件 virtual void readBinaryFile(QDataStream* dataStream); //生成可以显示的模型，每个实例只能调用一次 virtual void generateDisplayDataSet(); //获取显示模型 vtkDataSet* getDisplayDataSet(); //字符串转化为枚举 static SetType stringToSettype(QString s); </pre>
<p><i>MeshData</i></p> <p>网格数据管理, 数据访问接口</p>	<p><u>DataBase</u></p>	<pre> //获取单例指针 static MeshData* getInstance(); //添加 Kernal void appendMeshKernal(MeshKernal* kernal); //获取 Kernal 数量 int getKernalCount(); //获取第 index 个 Kernal MeshKernal* getKernalAt(const int index); //通过 ID 获取 Kernal MeshKernal* getKernalByID(const int id); //通过网格的数据表示获取 Kernal ID int getIDByDataSet(vtkDataSet* dataset); //移除第 index 个 Kernal void removeKernalAt(const int index); //添加组件 void appendMeshSet(MeshSet* s); //获取组件数目 int getMeshSetCount(); //获取第 index 个组件 MeshSet* getMeshSetAt(const int index); //通过 ID 获取网格组件 MeshSet* getMeshSetByID(const int id); //通过名字获取组件，大小写敏感 MeshSet* getMeshSetByName(const QString name); //移除第 index 个组件 void removeMeshSetAt(const int index); //获取与 ID 为 kid 的 Kernal 相关的全部组件 ID QList<int> getSetIDFromKernal(int kid); //清除数据 void clear(); QString getMD5(); //写出二进制文件 </pre>

		<pre> void writeBinaryFile(QDataStream* dataStream); //读入二进制文件 void readBinaryFile(QDataStream* dataFile); ///产生全部组件的显示模型 void generateDisplayDataSet(); </pre>
--	--	---

2.7 模型数据组织关系

模型数据指的是与计算相关的物理参数和计算参数的集合，材料参数和边界条件的设置也包含在内，所以组织关系相对复杂。模型数据和控制面板的算例树相互关联，接口较多，逻辑关系复杂。

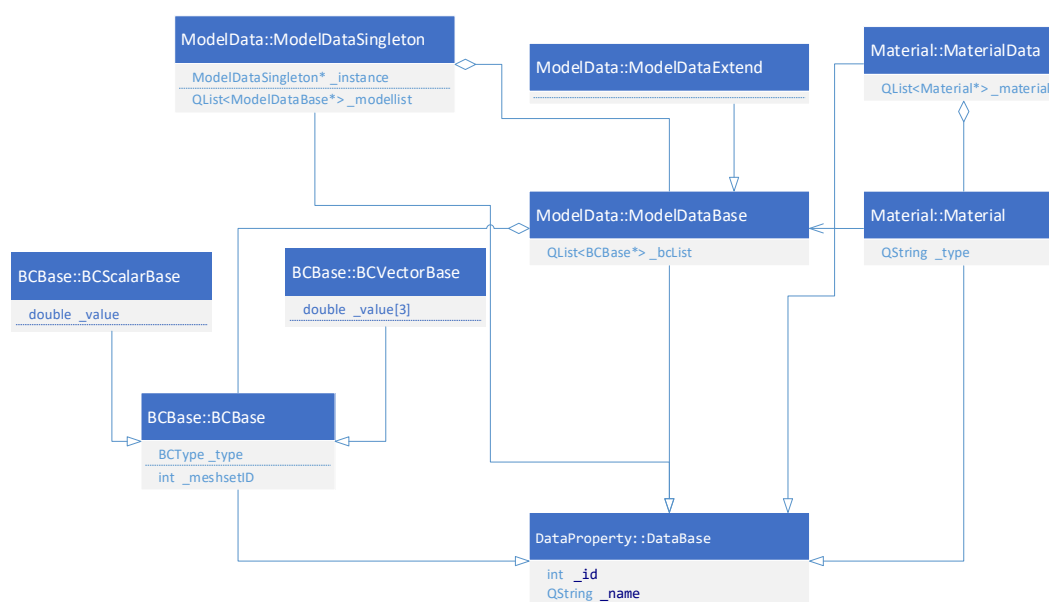


图 7 模型数据组织关系

模型数据不仅包含了设置模型仿真参数的相关信息，还包含了边界条件和材料两部分内容，这两部分内容不多，对基类的拓展也不多。模型数据与网格和几何数据之间都会形成关联与绑定的关系，而边界条件与材料与网格之间也存在关联与绑定的关系，为了简化关联绑定关系，因此将边界条件作为模型数据的集成元素。而材料在多模型的案例中可能会存在一种材料对应多个模型的多个网格的情况，为了实现复杂的关联绑定需求，所以将材料单独进行管理维护，材料的关联只需要与其唯一的标记 ID 进行标记管理即可。

所以在模型数据中存在两个数据单例的情况：模型和材料数据。两者之间通过材料的 ID 进行唯一性的标记，两个单例都作为数据访问的接口。

模型数据代码位于 `ModelData` 文件夹下，分别是基类和拓展类；边界条件相关代码位于 `BCBase` 文件夹下，材料位于 `Material` 文件夹下。

2.8 模型数据 API

表 4 模型数据 API

类名	父类	主要 API
<i>ModelData</i> <i>Base</i> 模型数据 基类	<u>DataBase</u>	<pre> //构造函数 ModelDataBase(ProjectTreeType treeType); //从配置文件中拷贝 virtual void copyFormConfig(); //将 ID 和类型信息传递给基类的各个参数 virtual void generateParaInfo() override; //获取最大的 ID static int getMaxID(); void setID(int id) override; //获取算例路径 QString getPath(); static void resetMaxID(); //设置类型 void setTreeType(ProjectTreeType type); //获取类型 ProjectTreeType getTreeType(); //求解完成时间 void setSolveTime(double t); //获取求解完成时间 double getSolveTime(); //检查数据是否可以求解 返回 True-可以求解; False- 不能求解 virtual bool checkSolveableStatus(QVector<Mod uleBase::Message> & messages) //获取 BC 数量 int getBCCount(); //添加 BC void appeendBC(BCBase::BCBase* bc); //根据类型获取 BC QList<BCBase::BCBase*> getBCByType(BCBase::BC Type type); //获取第 index 个 BC BCBase::BCBase* getBCAt(const int index); //移除边界条件 void removeBCAt(const int index); </pre>

		<pre> //获取关联的网格组件 ID QList<int> getMeshSetList(); //设置需要关联的组件 ID void setMeshSetList(QList<int> ids); //移除第 index 个组件 virtual void removeMeshSetAt(int index); //获取关联的 Kernal ID QList<int> getMeshKernalList(); //设置关联的 Kernal ID void setMeshKernalList(QList<int> k); //获取关联的几何形状 QList<int> getGeometryList(); //设置关联的几何形状 void setGeometryList(QList<int> geo); //获取仿真参数 SimlutationSettingBase* getSimlutationSetting (); //获取求解器设置 SolverSettingBase* getSolverSetting(); //组件是否使用 bool isComponentUsed(int index); </pre>
<i>BCBase</i> 边界条件 数据基类	DataBase	<pre> BCBase(); ~BCBase() = default; //拷贝数据 virtual void copy(DataBase* data) override; //设置网格组件 ID void setMeshSetID(int id); //获取网格组件 ID int getMeshSetID(); //获取网格组件名称 QString getMeshSetName(); //获取网格组件 MeshData::MeshSet* getMeshSet(); //设置边界条件类型 void setType(BCTYPE t); //获取边界条件 BCTYPE getType(); </pre>
<i>BCVectorBase</i> 向量边界 条件数据 基类	DataBase	<pre> BCVectorBase(); BCVectorBase(BCTYPE type, double* v); BCVectorBase(BCTYPE type, double x, double y, double z); //设置三个变量名称 void setVariableName(QString x, QString y, QS tring z); </pre>

		<pre> //设置值 void setValue(double* v); void setValue(double x, double y, double z); //获取值 void getValue(double* v); </pre>
<i>BCScalarBase</i> 标量边界条件数据基类	DataBase	<pre> BCScalarBase(); BCScalarBase(BCType type); ~BCScalarBase() = default; //设置值 void setValue(double v); //获取值 double getValue(); //设置变量名称 void setVariableName(QString s); </pre>
<i>Material</i> 材料数据基类	DataBase	<pre> Material(bool IDplus = true); //获取最大 ID static int getMaxID(); //获取类型 QString getType(); //设置类型 void setType(QString type); //拷贝数据 void copy(DataBase* data) override; </pre>
<i>MaterialSingleton</i> 材料单例类，数据接口	DataBase	<pre> //获取单例指针 static MaterialSingleton* getInstance(); //清空数据 void clear(); //获取材料数量 int getMaterialCount(); //获取第 i 个材料 Material* getMaterialAt(const int i); //根据 ID 获取材料 Material* getMaterialByID(const int id); //添加材料 void appendMaterial(Material* m); //根据 ID 移除材料 void removeMaterialByID(const int id); //将 ID 为 id 的材料添加至材料库 void appendToMaterialLib(const int id); //从材料库中加载 void loadFromMaterialLib(GUI::MainWindow* m); </pre>
<i>ModelDataBaseExtend</i>	ModelDataBaseExtend aBase	<pre> ModelDataBaseExtend(ProjectTreeType type); //添加报告 void appendReport(QString report); </pre>

<p>模型数据 基类拓展， 支撑平台 的默认功 能</p>	<pre> //获取报告数量 int getReportCount(); //获取第 index 个报告 QString getReportAt(int index); //移除第 index 个报告 void removeReportAt(int index); //设置材料，网格组件 ID 与材料 ID void setMaterial(int setID, int materialID); //获取组件的材料 ID int getMaterialID(int setid); //组件是否被设置材料 bool isMaterialSetted(int setid); //解除材料与组件的绑定 void removeMaterial(int setid); //获取节点数量 int getConfigDataCount(); //获取监控文件的名称 QStringList getMonitorFile(); //获取监控文件的绝对路径 QStringList getAbsoluteMonitorFile(); //根据文件获取监控的变量列表 QStringList getMonitorVariables(QString file) ; //获取监控曲线 QList<ConfigOption::PostCurve*> getMonitorCurves(); //获取第 index 的监控曲线 ConfigOption::PostCurve* getMonitorCurveAt(const int index); //获取后处理二维曲线文件 QStringList getPost2DFiles(); //获取后处理二维曲线文件的绝对路径 QStringList getAbsolutePost2DFiles(); //根据文件获取二维曲线文件中的变量 QStringList getPost2DVariables(QString f); //获取三维后处理文件 QString getPost3DFile(); //获取三维后处理标量 void get3DScalars(QStringList &node, QStringList &ele); //获取三维后处理向量 void get3DVector(QStringList &node, QStringList &ele); //添加二维后处理曲线 </pre>
---	--

		<pre> void appendPlotCurve(ConfigOption::PostCurve * c); //获取已绘制曲线 QList<ConfigOption::PostCurve*> getPlotCurves (); //移除二维曲线 void removePlotCurve(int index); //清除全部二维曲线 void clearPlotCurve(); //后处理曲线是否存在 bool isPostCurveExist(QString name); //添加标量变量名称 void appendScalarVariable(QString v); //获取标量变量名称 QStringList getScalarVariable(); //移除第 index 个标量 void removeScalarVariable(int index); //添加向量变量名称 void appendVectorVariable(QString v); //获取向量变量名称 QStringList getVectorVariable(); //移除第 index 个向量名称 void removeVectorVariable(int index); //清除三维变量 void clear3DVariable(); //设置二维绘图窗口 void setPost2DWindow(Post::Post2DWindowInterf ace* p2d); //获取二维窗口 Post::Post2DWindowInterface* getPost2DWindow(); //设置三维渲染窗口 void setPost3DWindow(Post::Post3DWindowInterf ace* p3d); //获取三维渲染窗口 Post::Post3DWindowInterface* getPost3DWindow(); </pre>
<i>ModelData</i> <i>Singleton</i> 模型单例 类，数据接	DataBase	<pre> //获取单例 static ModelDataSingleton* getInstance(); //添加模型 void appendModel(ModelDataBase* model); //获取第 index 个模型 ModelDataBase* getModelAt(const int index); </pre>

□		<pre> //获取模型个数 int getModelCount(); //通过 ID 获取模型 ModelDataBase* getModelByID(const int id); //通过名称获取模型 ModelDataBase* getModelByName(QString name); //清空数据 void clear(); //移除 ID 为 id 的模型 void removeModelByID(const int id); //获取第 index 个模型的 ID int getModelIDByIndex(const int index); </pre>
---	--	--

2.9 其他数据

除了上述主要的数据之外，软件运行过程还有其他的数据类示例，包括配置信息读取与保存，以及相关的设置选项信息，例如绘图选项、语言和工作目录等。配置信息的保存和读取源码位于 `configOption` 中，这些数据也是通过一个单例进行管理和维护的，访问方式与其他单例相同，具体的逻辑与数据关系这里不再做介绍。

软件相关的设置选项信息相关代码位于 `settings` 文件夹中，这部分没有复杂的逻辑关系，这部分也是通过一个单例进行管理和维护的，这些信息都将在软件正常关闭时写入 `ini` 文件，并在软件启动时读入，并加载到运行环境。下面将介绍一下这部分的 API。

2.10 其他数据 API

表 5 其他数据 API

类名	父类	主要 API
<i>BusAPI</i> 设置项单例类, 访问主接口		<pre> //获取单例指针 static BusAPI* instance(); //写出 ini void writeINI(); //设置工作路径 void setWorkingDir(); //获取工作路径 QString getWorkingDir(); //设置语言 “Chinese” “English” </pre>

		<pre> void setLanguage(const QString lang); //获取当前语言 QString getLanguage(); //获取绘图选项 GraphOption* getGraphOption(); //获取最近文件 QStringList getRecentFiles(); //添加最近文件 void appendRecentFile(QString f); //设置已经加载的插件 void setPlugins(QStringList p); //获取已经加载的插件 QStringList getPlugins(); </pre>
<p><i>GraphOption</i></p> <p>绘图选项</p>		<pre> //前处理顶部背景颜色 void setBackgroundTopColor(QColor c); QColor getBackgroundTopColor(); //前处理底部背景颜色 void setBackgroundBottomColor(QColor c); QColor getBackgroundBottomColor(); //高亮颜色 void setHighLightColor(QColor c); QColor getHighLightColor(); //预高亮颜色 void setPreHighLightColor(QColor c); QColor getPreHighLightColor(); //几何面颜色 void setGeometrySurfaceColor(QColor c); QColor getGeometrySurfaceColor(); //几何线颜色 void setGeometryCurveColor(QColor c); QColor getGeometryCurveColor(); //几何点颜色 void setGeometryPointColor(QColor c); QColor getGeometryPointColor(); //网格面颜色 void setMeshFaceColor(QColor c); QColor getMeshFaceColor(); //网格边颜色 void setMeshEdgeColor(QColor c); QColor getMeshEdgeColor(); //网格节点颜色 void setMeshNodeColor(QColor c); QColor getMeshNodeColor(); //网格节点大小 </pre>

		<pre> void setMeshNodeSize(float s); float getMeshNodeSize(); //网格边宽度 void setMeshEdgeWidth(float s); float getMeshEdgeWidth(); //几何点大小 void setGeoPointSize(float s); float getGeoPointSize(); //几何边宽度 void setGeoCurveWidth(float s); float getGeoCurveWidth(); //透明度 void setTransparency(int t); int getTransparency(); </pre>
--	--	--

三、视图类组织关系及 API

视图的功能以主界面为所有的功能与操作接口，主要接口位置位于主界面菜单栏工具栏以及控制面板的树形菜单。主界面的菜单栏和工具栏为通用功能的接口，而控制面板的树形菜单与仿真流程相关性较大，其中计算分析标签页下的树形菜单可以根据用户自己的仿真流程需要进行可视化定制，或者是基于插件接口进行定制化的开发。

软件启动之后会显示主界面，这也是用户进行人机交互的主要位置，根据功能区分，主界面可以分为五个区域，分别为：菜单栏/工具栏、控制面板、绘图区、控制台和进程窗口，它们的各自功能分别为：

菜单工具栏：软件各功能的主要入口，为用户提供快捷方便的交互操作接口。

控制面板：软件主要的功能实现，控制整个软件的运行状态，实现业务的主要流程。控制面板下半部分为属性窗口，显示属性与参数，同时用户可以直接对参数进行修改。

绘图区：绘图区是软件界面中最大的区域，负责实现图形的绘制渲染，同时支持用户的交互操作。

控制台：输出软件运行的状态信息，包括错误警告以及脚本命令，也包括求解器的输出内容。同时也可以在窗口键入脚本命令驱动程序的执行。

进程窗口：以进度条的形式显示当前程序的运行的进度信息，包括网格剖分进程，求解进程等较为耗时的操作状态。

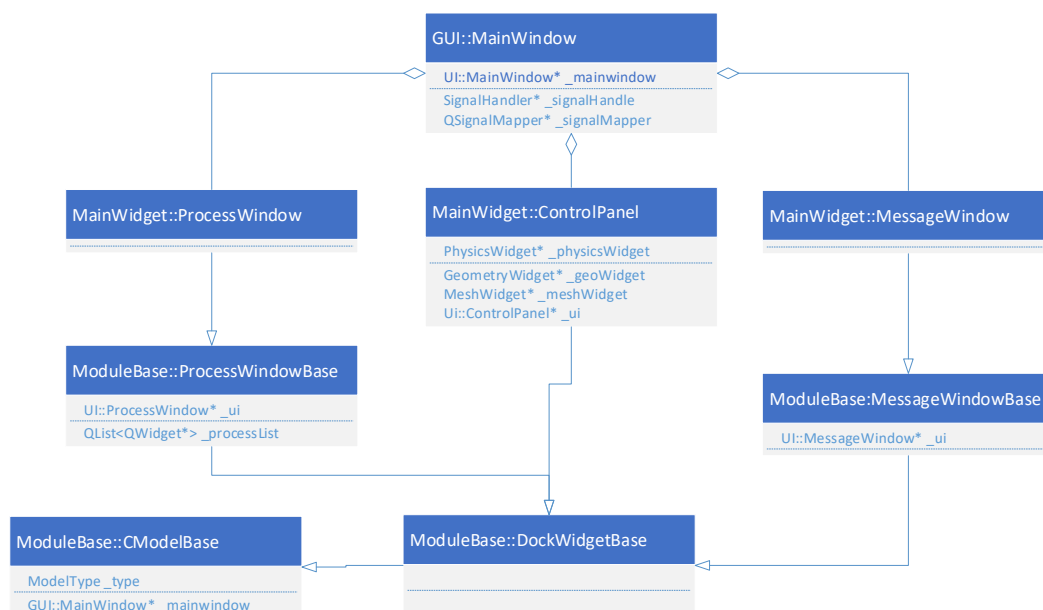


图 8 主界面

在相关的功能实现中，各个类之间大多以聚合的形式进行组织，例如，主界面 `MainWindow` 中聚合了控制面板类，控制台类和进度窗口类，而控制面板又聚合了四个标签页的窗口类。

3.1 主界面

主界面是整个软件运行时的大脑，负责信息的收集与分发，软件所有重要的交互操作都是通过信号的形式传递到主窗口，然后由主窗口负责将信号下发到相关的功能模块或者视图类进行渲染。举个例子，当点击网格组件某一个子节点时，会产生一个点击事件的信号，这个信号传递到主窗口之后会由两个窗口进行响应，一个是控制面板的属性框会显示对应节点的属性，另外绘图界面将会对该部分组件进行高亮显示，这个消息的分发就是通过主窗口实现的。

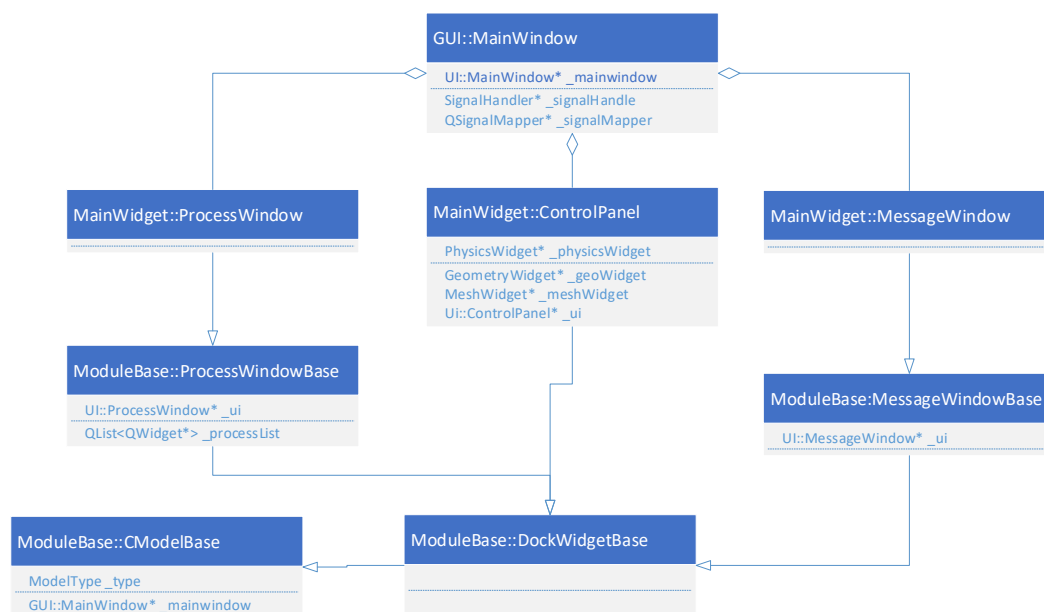


图 9 消息分发机制

主界面代码实现位于 **MainWindow** 文件夹，主要包含三个类：**MainWindow**、**SignalHandler** 以及 **SubWindowManager**。**MainWindow** 是 **QMainWindow** 的子类，实现了主界面的架构，包括菜单栏和工具栏的建立以及信号槽的连接，除此之外，上面所述的信号中转分发也是在这个类中实现的。**SignalHandler** 作为 **MainWindow** 的辅助，实现了重要的槽函数，以及一些简单的信号连接。**SubWindowManager** 是中间子窗口的管理类，实现对中间多窗口区域子窗口的统一管理，包括新建关闭以及状态维护等工作。

表 6 信号槽

类名	父类	主要 API
<i>SignalHandler</i> 信号处理		<pre> bool importMesh(const QStringList &filenames); bool importGeometry(const QStringList &filenames); bool exportGeometry(QString f); ///清除数据 void clearData(); /*求解 */ void on_actionSolve(); /*切换为英语 */ void on_actionEnglish(); /*切换为中文 */ void on_actionChinese(); /*处理模型树事件 */ </pre>

		<pre> void handleTreeMouseEvent(int eventtype, QTreeWi dgetItem*item, int proID); ///求解 void solveProjectPy(int projectIndex, int solver Index); void solveProject(int projectIndex, int solverIn dex); ///生成面网格 void generateSurfaceMesh(); ///生成体网格 void generateSolidMesh(); ///生成网格 void genMesh(); //添加求解器生成网格 void appendGeneratedMesh(QString name, vtkDataSe t* dataset); //导出网格 void exportMeshByID(QString filename, int kenerl ID = -1); void exportMeshPy(QString filename); ///刷新 Action 状态 void updateActionsStates(); //独立打开 2D 后处理窗口 void open2DPlotWindow(); void open2DPlotWindowPy();//提交 py 代码 //独立打开 3D 后处理窗口 void open3DGraphWindow(); void open3DGraphWindowPy(); //关闭后处理窗口 void closePostWindow(Post::PostWindowBase* p); </pre>
<p><i>SubWindowMa</i> <i>nager</i> 子窗口管理类</p>		<pre> //更新前处理 Actor void updatePreActors(); //更新前处理网格 Actor void updatePreMeshActor(); //更新前处理几何 Actor void updatePreGeometryActor(); //获取前处理窗口 MainWidget::PreWindow* getPreWindow(); //前处理窗口是否打开 bool isPreWindowOpened(); //后处理窗口是否打开 bool isPostWindowOpened(); //更新翻译 void reTranslate(); </pre>

		<pre> /*获取创建几何窗体*/ GenerateGeometry::GenerateGeometryWidget* getGeometryWindow(); //关闭所有窗口 void closeAllSubWindow(); ///获取当前激活的 MDI 子窗口 ModuleBase::GraphWindowBase* getCurrentWindow(); ModuleBase::GraphWindowBase* getWindowByTypeID(QString type, int id); //打开开始页，并加载网页 void openUrl(QString web); //打开起始页 void openStartPage(); /*打开前处理窗口 */ void openPreWindow(); </pre>
--	--	---

3.2 控制台与进度窗口

控制台与进度窗口是主窗口的重要组成部分，控制台显示软件运行过程中的相关状态与提示信息，同时负责实现 Python 脚本命令的交互输入。进度窗口主要实现对求解进程和网格划分进程状态与进度的实时监控。二者的代码实现位于 MainWidget 文件夹。

表 7 控制台与进度窗口

类名	父类	主要 API
<i>MessageWindow</i> 主窗口控制台类	<i>MessageWindowBase</i>	<pre> //翻译 virtual void reTranslate() override; //执行 python 命令 void executePyscript(); </pre>
<i>ProcessWindow</i> 主窗口进度类	<i>ProcessWindowBase</i>	<pre> //翻译 virtual void reTranslate() override; //添加进程 void addProcess(QWidget* w); //移除进程 void removeWidget(QWidget* w); </pre>

3.3 控制面板

控制面板是仿真流程控制的主要位置，也是整个软件的核心，控制面板包含

四个标签页，每一个标签一个对应不同的模块区分，包含几何、网格、计算分析和后处理四大部分。从代码实现上主要包含五个类，控制面板，几何标签页，网格标签页，计算分析标签页以及属性框。后四者都是以聚合的形式聚合于控制面板总类。这里只说明控制面板类和属性框，其他关键标签页将在下面单独进行更详细的说明。这部分的代码实现也是全部位于 MainWidgets 文件夹。

表 8 控制面板主要 API

类名	父类	主要 API
<i>ControlPanel</i> 控制面板总类	<i>DockWidget</i> <i>Base</i>	<pre> //鼠标点击事件，包括左键单击双击及右键菜单 virtual void on_TreeMouseEvent(int eventType , QTreeWidgetItem* item, int proID); //更新属性框 void updatePropertyTab(DataProperty::DataBase * popList); //设置属性窗当前页 void changePropTabByProjectPage(int index); //更新属性窗体 void updateParaWidget(QWidget* w); ///更新两个 post widget void updatePostWidget(QWidget* tree, QWidget* prop); //清空属性窗 widget void clearWidget(); //根据配置信息更新开放接口 void registerEnabledModule(); </pre>
<i>PropertyTable</i> 属性框窗体	<i>QWidget</i>	<pre> //重置宽度 void resize(int w = -1); //更新属性窗 void updateTable(DataProperty::DataBase* data); //翻译 void retranslate(); </pre>

3.4 几何网格树形菜单

几何和网格的树形菜单与内存中的几何与网格数据密切相关，这两个树形菜单是对内存重要信息的可视化展示。这两个树形菜单的代码实现也是位于 MainWidgets 文件夹中。

表 9 几何网格树形菜单主要 API

类名	父类	主要 API
<i>GeometryTreeWidget</i> 几何标签页	<i>QTreeWidget</i>	<pre> //注册开放可用接口 void registerEnabledModule(); //更新树 void updateTree(); //鼠标单击事件 void singleClicked(QTreeWidgetItem*, int); //清除当前数据 void removeData(); //编辑当前形状 void editGeometry(); //隐藏全部 void hideAll(); //显示全部 void showAll(); 信号: //更新显示状态 void updateDisplay(int index, bool visible); //移除第 index 个数据 void removeGeoData(int index); //显示属性 void displayProp(DataProperty::DataBase* pops); //更新 action 状态 void updateActionStates(); //通过主窗口统一接口显示非模态对话框 void showGeoDialog(QDialog*); //高亮几何 void highlightGeometrySet(Geometry::GeometrySet*, bool); //清除高亮 void clearHighlight(); </pre>
<i>MeshWidget</i> 网格标签页	<i>QTreeWidget</i>	<pre> //注册开放可用接口 void registerEnabledModule(); //更新树 void updateTree(); //更新网格 kernal 子树 void updateMeshTree(); //更新组件子树 </pre>

		<pre> void updateMeshSetTree(); //鼠标单击事件 void singleClicked(QTreeWidgetItem*, int); //移除当前 Kernal void removeMeshData(); ///移除当前组件 void removeSetData(); 信号: //更新显示状态 void updateDisplay(int index,bool visable); //移除第 index 个 Kernal void removeMeshData(int index); //高亮组件 void higtLightSet(MeshData::MeshSet* set); //高亮 Kernal void higtLightKernal(MeshData::MeshKernal* k) ; //更新属性 void displayProp(DataProperty::DataBase* pops); //更新状态 void updateActionStates(); //显示参数窗体 void dispalyParaWidget(QWidget*); //更新前处理窗口 void updatePreMeshActor(); //清除高亮 void clearHighLight(); </pre>
--	--	--

3.5 前后处理窗口

前后处理窗口指的是数据的可视化窗口，前处理为一个单独的窗口，实现几何与网格的可视化以及选取高亮等交互操作，后处理分为两部分，一部分是二维部分，可视化对象通常为曲线等二维数据信息，这部分也包含可以实时可视化数据变化的动态曲线的功能。另外一部分为三维可视化渲染部分，主要包含云图矢量图，切线和切面等功能，也包含动画的功能。前处理部分代码实现位于 MainWidgets 文件夹，后处理部分代码实现则位于 postWidgets 文件夹。为了方便窗体的统一管理该部分的组织关系相对复杂：

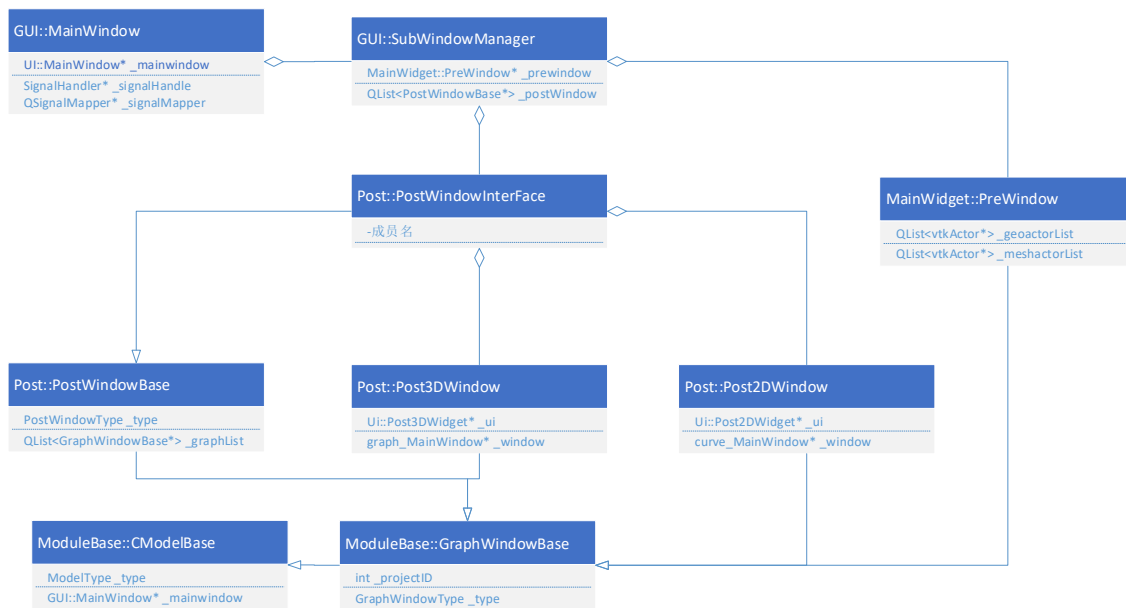


图 10 前后处理流程

本部分的主要 API 如下：

表 10 前后处理主要 API

类名	父类	主要 API
<i>GraphWindowBase</i> 绘图类基类	<i>CModuleBase</i>	<pre> GraphWindowBase(GUI::MainWindow* mainwindow ,int proID,GraphWindowType type = GraphWindowType::U nDefined, bool connectToMainwindow = false); //获取类型 GraphWindowType getGraphWindowType(); //保存图片 virtual void saveImage(QString filename, int wid th, int heigh, bool showDlg); //获取 ID int getID(); //设置视角 virtual void setView(QString view); virtual void setViewValue(int x1, int x2, int x3 , int y1, int y2, int y3, int z1, int z2, int z3); //设置背景颜色 virtual void setBackground(QColor color1, QColor color2); //字符串形式获取窗口类型 QString getStringGraphWindowType(); </pre>

<p><i>Graph3Dwindow</i> 三维渲染窗口</p>	<p><i>GraphWindowBase</i></p>	<pre> Graph3DWindow(GUI::MainWindow* mainwindow, int id, GraphWindowType type, bool connectToMainwindow = false); //添加渲染对象 void AppendActor(vtkProp* actor, ActorType type = ActorType::D3, bool reRender = true, bool reset = true); //移除渲染对象 void RemoveActor(vtkProp* actor); //启用/禁用渲染处理 void enableActor(vtkActor* actor, bool show = true); void saveImage(QString filename, int w, int h, bool showdlg) override; void setViewValue(int x1, int x2, int x3, int y1, int y2, int y3, int z1, int z2, int z3) override; //重置视角 void resetCamera(); //根据设置重绘 void updateGraphOption() override; void reTranslate() override; //获取交互器 PropPickerInteractionStyle* getInteractionStyle(); //获取渲染器 vtkRenderer* getRenderer(); //获取选择模式 SelectModel getSelectModel(); //获取窗体世界坐标系下高度 double getWorldHeight(); //获取窗体世界坐标系下宽度 double getWorldWidth(); //重绘 void reRender(); </pre>
<p><i>PreWindow</i> 前处理渲染窗口</p>	<p><i>Graph3DWindow</i></p>	<pre> //获取选择的几何 QMultiHash<Geometry::GeometrySet*, int> getGeoSelectItems(); //设置选择模式 void setSelectModel(int mode) override; //设置草图类型 void setSketchType(ModuleBase::SketchType t); //更新网格渲染 </pre>

		<pre> void updateMeshActor(); //更新几何渲染 void updateGeometryActor(); //清除所有高亮对象 void clearAllHighLight(); 信号: //关闭 void closed(); //显示几何形状 void showGeoSet(Geometry::GeometrySet* set); //移除几何形状的显示 void removeGemoActors(Geometry::GeometrySet* set); //显示基准 void showDatum(Geometry::GeometryDatum*); //移除几何基准显示 void removeGeoDatumActors(Geometry::GeometryDatum*); //设置选择模式 void setGeoSelectMode(int index); //选择的几何元素 void selectGeoActorShape(vtkActor* ac, int shape , Geometry::GeometrySet* set); //更新网格渲染元素 void updateMeshActorSig(); </pre>
<i>Post2Dwindow</i> 后处理二维 可视化窗口	<i>GraphWindowBase</i>	<pre> Post2DWindow(GUI::MainWindow* mw, int proID); ~Post2DWindow(); //窗口绘制 void replot(); //打开文件 bool openFile(QString file); //从文件添加曲线 void addCurve(QString tep_filename, int tep_column_index); //不调用文件, 直接用 2 组浮点数画曲线 void addCurve(QVector<double> data_x, QVector<double> data_y, QString tep_filename); //添加曲线 x 默认为索引值 (0, 1, 2) void addCurve(QVector<double> data_y, QString filename); //更新曲线 void updateCurve(QVector<double> data_x, QVector<double> data_y, QString tep_filename); </pre>

	<pre> //更新曲线 void updateCurve(QVector<double> data_y, QString tep_filename); //删除曲线 void delCurve(QString filename, int colum_index) ; //清除全部曲线 void delAllCurves(); //设置标题名称 void setTitle(QString title); //设置标题字体颜色 0-宋体 1-黑体 2-Arial void setTitleFont(int font); //设置标题字体大小 void setTitleFontSize(int size); //设置标题字体颜色 void setTitleFontColor(QColor color); //设置绘图区的背景颜色 void setBackground(QColor color); //设置绘图区域网格化 void setPlotGrid(bool isgrid); //设置 Legend 位置 0-none 1-右上 2-右下 3-左上 4-左 下 void setLegendPosition(int p); //设置坐标轴个数 void setAxisNum(int num); //设置坐标轴名称 void setAxisName(QString axis, QString name); //设置坐标轴字体颜色 void setAxisFontColor(QString axis, QColor color); //设置坐标轴字体大小 void setAxisFontSize(QString axis, int size); //设置坐标轴字体 0-宋体 1-黑体 2-Arial void setAxisFont(QString axis, int font); //设置曲线颜色 void setCurveColor(QString filename, int colInde x, QColor color); //设置线型 void setCurveStyle(QString filename, int colInde x, Qt::PenStyle style); //设置线宽 void setCurveWidth(QString filename, int colinde x, int width); //是否显示数据点 </pre>
--	--

		<pre> void setCurveShowPoint(QString filename, int col index, bool show); //设置曲线名称 void setCurveName(QString filename, int colindex , QString name); //设置曲线坐标轴索引 void setCurveAxisIndex(QString filename, int col index, int axisIndex); //获取模型树窗口 QWidget* getTreeWidget(); //获取属性窗口 QWidget* getPropWidget(); ///设置背景颜色 void setBackground(QColor color1, QColor color2) override; ///保存图片 void saveImage(QString filename, int width, int heigh, bool showDlg) override; ///设置坐标轴范围 axis-x y x2 y2 reange[0]- min range[1]-max void setAxisRange(QString axis, double range[2]) ; ///获取属性列名称 QStringList getColumnNameList(QString filename); //语言切换 void reTranslate() override; //获取曲线属性接口 curve_line_data getCurveProp(QString tepFileName , int colIndex); //开始动画 bool startAnimate(); //结束动画 bool stopAnimate(); //设置动点颜色 void setAniPointColor(QString filename, int colu m, QColor color); //设置动点形状 void setAniPointType(QString filename, int colum , aniPointType type); </pre>
<i>Post3Dwindo</i> w 后处理三维	<i>GraphWind</i> owBase	<pre> Post3DWindow(GUI::MainWindow* mainwindow, int pr oID); ///打开文件 void openFile(QString files ,bool apply = true); //应用 </pre>

可视化窗口	<pre> void applyClick(); //开始动画 bool startAnimate(); //停止动画 bool stopAnimate(); //获取树的 widget QWidget* getTreeWidget(); //更新 pipeline Object void updatePipelineObjDataSet(PipelineObject* obj, QString filename); //获取当前 Pipeline object PipelineObject* getCurrentPipelineObj(); ///获取树 Prop widget QWidget* getPropWidget(); ///获取菜单栏 QList<QToolBar*> getToolBars(); ///清空所有绘图 void clearAll(); ///设置背景色 void setBackground(QColor color1, QColor color2) override; ///保存图片 void saveImage(QString filename, int width, int heigh, bool showDlg) override; //获取 ScalarBar 范围 void getScalarRange(PipelineObject* obj, double value[2]); //设置 ScalarBar 范围 void reScaleScalarRange(PipelineObject* obj, dou ble min, double max); //显示颜色条 void displayLegendBar(bool on); //展示云图 void viewCounter(PipelineObject* obj, QString va r); //展示矢量图 void viewVector(PipelineObject* obj, QString var); //语言切换 void reTranslate() override; //获取节点数据 vtkDataSet* getPipelineObjDataSet(PipelineObject * obj); </pre>
-------	---

<p><i>PostWindowBase</i> 后处理封装基类</p>	<p><i>GraphWindowBase</i></p>	<pre> PostWindowBase(GUI::MainWindow* mainwindow, int proID, ModuleBase::GraphWindowType gt, PostWindowType type); //设置类型 void setWindowType(PostWindowType type); //获取类型 PostWindowType getPostWindowType(); //添加绘图窗口 void appendGraphWindow(ModuleBase::GraphWindowBase* g); //移除窗口 void removeGraphWindow(ModuleBase::GraphWindowBase* g); //移除第 index 个绘图窗口 void removeGraphWindow(int index); //清空全部绘图窗口 void removeAllGraphWindow(); //设置树形窗口 void setTreeWidget(QWidget* w); //获取树形窗口 QWidget* getTreeWidget(); //设置属性窗口 void setPropWidget(QWidget* w); //获取属性窗口 QWidget* getPropWidget(); //添加工具条 void appendToolBar(QToolBar* toolbar); //移除工具条 void removeToolBar(QToolBar* toolbar); //获取全部工具条 QList<QToolBar*> getToolBarList(); //关闭窗口信号 void closeWindowSignal(Post::PostWindowBase* w); </pre>
<p><i>Post2DWindowInterface</i> 后处理二维封装</p>	<p><i>PostWindowBase</i></p>	<pre> Post2DWindowInterface(GUI::MainWindow* m, int proid); void reTranslate() override; //获取二维绘图窗口 Post2DWindow* getWindow(); //打开文件 void openfile(QString filename); //添加曲线 void addCurve(QString name, QString f, QString x, QString y); //移除曲线 </pre>

		<code>void removeCurve(QString name);</code>
<i>Post3DWindowInterface</i> 后处理三维封装	<i>PostWindowBase</i>	<code>Post3DWindowInterface(GUI::MainWindow* m, int pr oid);</code> //获取三维绘图窗口 <code>Post3DWindow* getWindow();</code> <code>virtual void reTranslate() override;</code>

3.6 算例树形菜单

算例树形菜单是仿真流程最重要的体现，在 FastCAE 中这部分也是灵活度最高的部分，这部分既可以通过定制插件进行专业定制也可以利用插件模式自己完成插件来进行功能拓展。平台内置了树形菜单的一些功能，这些是定制插件的基础，所以下面将列出主要 API。这部分代码实现位于 ProjectTree 文件夹和 ProjecttreeExtend 文件夹下，后者原本是作为二次开发的接口，但是随着插件接口的开发这部分功能已经被取代，所以可以不用考虑，后面的版本将对这个类进行删除。

表 11 算例树形菜单 API

类名	父类	主要 API
<i>ProjectTreeBase</i> 算例树基类， 无基本节点	<i>QObject</i>	<code>ProjectTreeBase(GUI::MainWindow *mainwindow);</code> //从配置文件中拷贝信息 <code>void copy(ConfigOption::ProjectTreeInfo* in fo);</code> //设置名称 <code>void setName(const QString &name);</code> //获取名称 <code>QString getName();</code> //设置树的类型 <code>void setType(ProjectTreeType type);</code> //获取树的类型 <code>ProjectTreeType type() const{ return _treeT ype; }</code> //根据配置项追加节点 <code>void appendItem(ConfigOption::TreeItem* ite m);</code> //创建树 <code>virtual void createTree(QTreeWidgetItem* ro ot, GUI::MainWindow* mainwindow);</code> //设置模型数据

		<pre> virtual void setData(ModelData::ModelDataBase* data); //获取模型数据 ModelData::ModelDataBase* getData(); ///清空 QList<TreeWidgetItem*> _itemList{} 会产生严重后果，慎重调用 void clearTreeItems(); //鼠标事件 void on_MouseEvent(int eventType, QTreeWidgetItem* item); //更新树 virtual void updateTree(); //根据类型更新子树 virtual void updateTreeByType(const TreeItem type); //获取求解状态 SolveStatus getSolveStatus(); //翻译 virtual void reTranslate(); //设置中文名称 void setChinese(QString c); //获取中文名称 QString getChinese(); //设置不可见节点 void setDisableItems(QStringList s); //获取不可见节点 QStringList getDisableItems(); //文件是否存在 static bool isFileExist(QString filename); </pre>
<i>ProjectTreeWithBasicNode</i> 带有基本节点的树形菜单	<i>ProjectTreeBase</i>	<pre> ProjectTreeWithBasicNode(GUI::MainWindow* mainwindow); ~ProjectTreeWithBasicNode(); //查看云图 void viewCounterPost(QString variable); //查看向量 void viewVectorPost(QString variable); virtual void reTranslate() override; virtual void updateTree() override; //更新子树节点 virtual void updateGeometrySubTree(); virtual void updateMeshSubTree(); virtual void updateBCSubTree(); virtual void updateReportTree(); virtual void updateMonitorTree(); virtual void updatePostTree(); </pre>

四、控制类相关说明

在 FastCAE 平台中，控制类搭建起了数据类与视图类的桥梁，是软件不可或缺的重要一环，控住类的特殊性在于其实例大部分为局部变量，生命周期较短，而且控制类只能用于特定的情景之下，不具备良好的通用性，因此本部分将不会详细介绍各个部分的功能或者 API，读者只需要有大概的认识即可。

4.1 几何建模命令

在几何建模模块中，我们采用了命令模式实现了建模过程的无限步撤销重做。命令模式的大致结构如下图所示：

在命令模式中，每一个命令对应一个具体的功能实现，每一个功能都对应一个命令类，但是他们都有一个共同的基类，每一个命令根据各自的业务需要定义出 **undo** 和 **redo** 的操作。当用户操作界面对话框完成相应的输入之后，对话框会根据类型创建出一个命令子类，并将对话框设置的相关参数一并设置到命令中，并交给命令执行器执行，这里的命令执行器为 **GeoCommandList**，执行器调用命令子类的 **execuate** 函数完成功能，并将命令保存进命令列表。执行器中包含两个命令列表来实现 **Undo** 和 **redo** 功能，当主界面点击 **undo** 或者 **redo** 按钮时则执行器从相应的命令列表中拿出相应的最后一个命令执行 **undo** 和 **redo** 操作，执行完成之后需要对这条命令重新设定列表，以保证命令可以被重复操作。

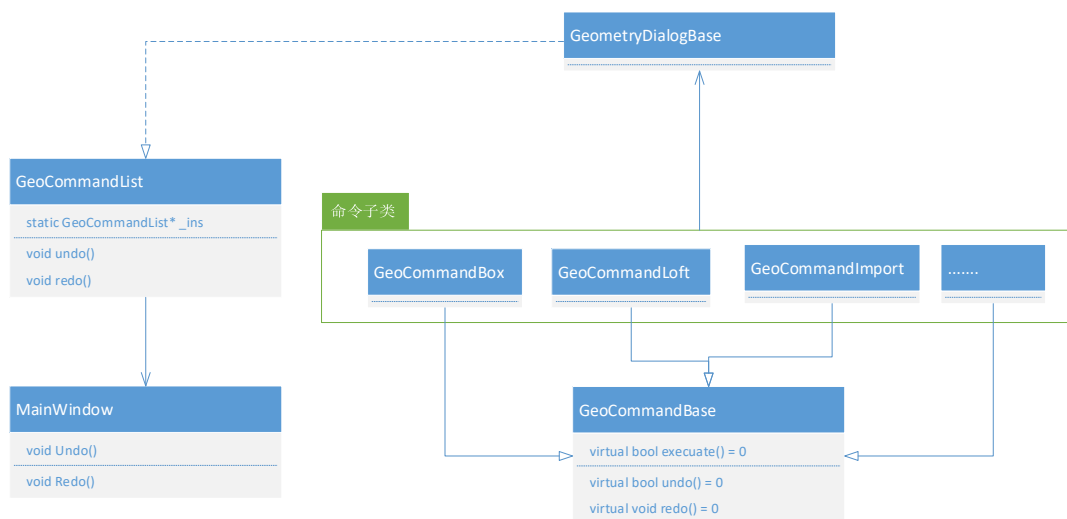


图 11 几何建模命令流程

几何建模的命令基类和子类以及执行器的代码实现位于 **GeometryCommand**

文件夹下，相关的几何建模对话框代码实现位于 `GeometryWidgets` 文件夹下。

4.2 输入输出

输入输出是指程序运行时的内存与外存文件的数据传递。包括工程文件的读入写出，求解器文件的数据交互，以及网格几何文件的导入导出。其中网格和几何文件的导入导出位于各自的数据实现文件夹下。其他部分数据读写均位于 `IO` 文件夹下。需要特别说明的是这里有一个模板替换的功能，由 `TemplateReplacer` 实现，模板要求写成下列的格式：

XXXXXXXXXXXXXXXXX%ParaName%XXXXXXXXXXXXXXXXX

其中“%ParaName%”为要被替换的内容，会被替换为 `DataBase` 中，名称为“`ParaName`”的参数的值，若参数隶属于参数组，则相应格式变成为“`GroupName/ParaName`”。需要注意的是替换以行为单位，两个百分号之间不能换行，否则不能替换。

4.3 Gmsh 集成

平台集成了 `Gmsh` 作为网格剖分器，平台通过可执行程序多进程形式调用。过程为：首先将选择的几何写出为“`geometry.brep`”文件，路径为 `FastCAE` 可执行文件的上一级的 `temp` 文件夹；然后根据设定的参数在相同的路径下生成 `gmsh.geo` 文件，这是 `Gmsh` 的脚本文件，生成的方式就是上面说的模板的形式；第三步是在该目录下用 `gmsh.geo` 驱动 `Gmsh` 运行，生成 `mesh.vtk` 文件，并监控程序的运行状态；最后当程序运行结束之后自动读取 `mesh.vtk` 文件完成网格剖分。



Python 命令说明

FastCAE 研发小组

2020 年 1 月

目录

一、概述.....	50
二、MainWindow.py.....	50
三、ControlPanel.py	51
四、Case.py.....	52
五、Material.py.....	53
六、CAD.py	53
七、Post.py.....	61
八、Mesher.py.....	62

一、概述

FastCAE 以 Python 作为脚本语言，重要的功能全部封装为 Python 接口。这些接口定义在 FastCAE 可执行程序的同级目录下后缀为 py 的文件，目前包含下面的八个文件，这些接口既可以独立调用，也可以由用户重新组合开发完成特定的功能，下面将对这八个文件中的关键 python 接口进行介绍：









 PostProcess.py	2019/11/2 12:04	JetBrains PyChar...	14 KB
 Post.py	2019/11/2 12:04	JetBrains PyChar...	1 KB
 Mesher.py	2019/12/12 18:45	JetBrains PyChar...	3 KB
 Material.py	2019/11/2 12:04	JetBrains PyChar...	1 KB
 MainWindow.py	2019/12/18 11:59	JetBrains PyChar...	20 KB
 ControlPanel.py	2019/11/2 12:04	JetBrains PyChar...	2 KB
 Case.py	2019/11/2 12:04	JetBrains PyChar...	2 KB
 CAD.py	2019/12/4 10:02	JetBrains PyChar...	28 KB

图 12 接口文件列表

二、MainWindow.py

该文件中定义的方法大部分是在由主窗口触发的动作，主要 C++ 功能实现也是位于 MainWindow 文件夹下，主要方法如下：

表 12 MainWindow.py 接口方法

方法	参数	功能
undo()		撤销
redo()		重做
importMesh(filename)	filename 文件绝对路径	导入网格
exportMesh(filename)	filename 文件绝对路径	导出网格
importGeometry(filename)	filename 文件绝对路径	导入几何
exportGeometry(filename)	filename 文件绝对路径	导出几何
openProjectFile(filename)	filename 文件绝对路径	打开工程文件
saveProjectFile(filename)	filename 文件绝对路径	保存工程文件
saveImage(w,h,id,win,file)	w 图片宽度 h 图片高度 id 窗口 id win 窗口类型	保存图片

	file 图片绝对路径	
setView(id,win,view)	Id 窗口 id Win 窗口类型 View 视角	设置固定视角
setViewRandValue(id,win,x1,x2,x3,y1,y2,y3,z1,z2,z3)	Id 窗口 id Win 窗口类型 其他: 视角参数	设置随机视角
openPost3D()		打开单独三维后处理窗口
openPost2D()		打开单独二维后处理窗口
openPreWindow()		打开前处理窗口
solveProject(projectIndex,solverIndex)	projectIndex 算例索引 solverIndex 求解器索引	求解算例

三、ControlPanel.py

该文件中定义的方法主要是由控制面板触发的动作，主要是封装界面的相关的操作，C++实现位于 MainWidgets 文件夹下，主要方法如下：

表 13 ControlPanel.py 接口方法

方法	参数	功能
createCase(name, typ)	name 算例名称 typ 算例类型	创建算例
deleteCase(caseid)	caseid 算例 id	删除算例
caseRename(pid,newname)	Pid 算例 id Newname 新名称	算例重命名
updateMeshSubTree(caseid)	caseid 算例 id	更新算例网格子树
updateBCSubTree(caseid)	caseid 算例 id	更新算例边界条件子树
updatePostTree(caseid)	caseid 算例 id	更新算例后处理子树

四、Case.py

该文件中的方法主要是和算例相关的方法，与上面的 `ControlPanel` 有一些重合的部分，不同的是该文件中的方法更加偏向底层数据，在这些方法中有对 `ControlPanel` 中定义方法的调用。相关 C++ 代码实现位于 `modelData` 文件夹下，主要方法如下：

表 14 Case.py 接口方法

方法	参数	功能
<code>importMeshComponents(caseId, addcomponentsId)</code>	<code>caseID</code> 算例 ID <code>addcommponentsID</code> 添加的网格组件 id	算例导入网格组件
<code>importGeometry(caseId, addcomponentsId)</code>	<code>caseId</code> 算例 ID <code>addcommponentsID</code> 添加的形状组件 id	算例导入几何
<code>addBC(caseId, id, bctypetostring)</code>	<code>caseID</code> 算例 ID <code>id</code> 边界条件 id <code>bctypetostring</code> 边界条件类型	添加边界条件
<code>objValChanged(value, describe, stype)</code>	<code>Value</code> 变化后的值 <code>Describe</code> 变量名称 <code>Stype</code> 变量类型	参数值发生变化
<code>setValue(caseId, svariable, stype, sValue)</code>	<code>caseID</code> 算例 ID <code>svariable</code> 变量名称 <code>stype</code> 变量类型 <code>svalue</code> 变量值	设置变量参数值
<code>setBCValue(caseId, index, svariable, stype, sValue)</code>	<code>caseID</code> 算例 ID <code>index</code> 边界条件索引 <code>svariable</code> 变量名称 <code>stype</code> 变量类型 <code>svalue</code> 变量值	设置边界条件的变量参数值

五、Material.py

该文件中的方法主要是对材料相关的方法进行的封装，目前处于技术路线的验证阶段，封装的方法有限。

表 15 Material.py 接口方法

方法	参数	功能
setValue(Id,svariable,stype,sValue):	ID 材料 ID svariable 变量名称 stype 变量类型 svalue 变量值	设置材料的变量参数值

六、CAD.py

该文件中封装了与三维几何建模相关的方法，与前面的接口封装方式不同，这部分的接口采取了面向对象的方法。相关的 C++ 代码实现位于 GeometryCommand 文件夹下，主要接口类如下：

表 16 CAD.py 接口方法

类	方法	参数	功能
Box 长方体	setName(self, name)	name 名称	设置名称
	setLocation(self, x,y,z)	x y z 坐标位置	设置位置
	setPara(self, l,w,h):	l w h 长宽高	设置形状参数
	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	create(self)		完成创建
	edit(self)		完成编辑
Cylinder 圆柱	setName(self, name)	name 名称	设置名称
	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setLocation(self, x,y,z)	x y z 坐标位置	设置位置
	setAxis(self, x,y,z)	xyz 轴线方向	设置轴线方向

	setRadius(self,radius)	radius 半径	设置半径
	setLength(self,length)	length 长度	设置长度
	create(self)		完成创建
	edit(self)		完成编辑
Cone 圆台	setName(self, name)	name 名称	设置名称
	setEditID(self, id)	Id 形状 ID	设置要编辑形状 ID
	setLocation(self, x,y,z)	x y z 坐标位置	设置位置
	setAxis(self, x,y,z)	xyz 轴线方向	设置轴线方向
	setRadius(self,radius, radius2)	radius 上端半径 radius2 下端半径	设置半径
	setLength(self,length)	length 长度	设置长度
	create(self)		完成创建
Sphere 球体	edit(self)		完成编辑
	setName(self, name)	name 名称	设置名称
	setEditID(self, id)	Id 形状 ID	设置要编辑形状 ID
	setLocation(self, x,y,z)	x y z 坐标位置	设置位置
	setRadius(self,radius)	radius 半径	设置半径
	create(self)		完成创建
Point 点	edit(self)		完成编辑
	setName(self, name)	name 名称	设置名称
	setEditID(self, id)	Id 形状 ID	设置要编辑形状 ID
	setLocation(self, x,y,z)	x y z 坐标位置	设置位置
	setOffset(self, x,y,z)	x y z 偏移量	设置偏移量
	create(self)		完成创建

	edit(self)		完成编辑
Line 直线	setName(self, name)	name 名称	设置名称
	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setStartPoint(self, x,y,z)	x y z 坐标位置	设置起始点位置
	setEndPoint(self, x,y,z)	x y z 坐标位置	设置终止点位置
	setDirection(self,x,y,z)	xyz 直线方向	设置直线方向
	setLength(self,len)	len 长度	设置长度
	setDirectionReverse(self, reverse)	reverse 是否反向	设置是否反向
	create(self)		完成创建
	edit(self)		完成编辑
Face 平面	setName(self, name)	name 名称	设置名称
	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	appendEdge(self, geoset, index)	geoset 边所在形状的 ID index 边的索引	添加围成面的边
	create(self)		完成创建
	edit(self)		完成编辑
Chamfer 倒直角	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setSectonType(self,typestr)	typestr 倒角截面类型 : ”Symmetrical” ”Asymmetrical”	设置截面形状, 对称和非对称
	setSymmetricalDistance(self,d1)	d1 倒角长度	对称截面的倒角长度
	setAsymmetricalDistances(self,d1,d2)	d1 d2 非对称截面的倒角长度	非对称截面的倒角长度
	appendEdge(self, geoset,	geoset 边所在	添加倒角的边

	index)	形状的 ID index 边的索引	
	create(self)		完成创建
	edit(self)		完成编辑
Fillet 倒圆角	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	appendEdge(self, geoset, index)	geoset 边所在 形状的 ID index 边的索引	添加倒角的边
	setRadius(self, radius)	radius 半径	设置半径
	create(self)		完成创建
	edit(self)		完成编辑
VariableFillet 可变圆角	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	VariableFilletOnEdge(self, setid, edgeindex)	setid 几何形状 ID edgeIndex 边索引	设置边
	AppendVariablePoint(self, location, radius)	Location 参数 化位置[0,1] radius 半径	添加可变点
	setBasicRad(self, basicRad)	basicRad 基本 半径	设置基本半径
	create(self)		完成创建
	edit(self)		完成编辑
BooLOperation 布尔操作	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setBoolType(self, booltype)	booltype 布尔 运算类 型 : "Cut" "Fause" "Common"	设置运算类型

	setBodysId(self,body1,body2)	body1 body2 几何形状 id	添加几何体
	create(self)		完成创建
	edit(self)		完成编辑
MirrorFeature 镜像特征	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setBodys(self,bodys)	bodys 几何形状 id, 字符串形式	添加几何体
	SaveOrigin(self, save)	save 是否保存	设置是否保存源对象
	setSymmetricPlaneMethod(self, method)	method 方法	设置镜像面指定方法
	setFace(self,faceindex,facebody):	faceindex 面索引 facebody 面所在形状 id	选择已经存在的面作为对称面
	setPlaneMethod(self,planemethod)	planemethod 坐标平面 “XOY” 等	选择坐标平面作为对称面
	setDir(self, x, y, z)	xyz 平面方向	设置任意对称平面方向
	setBasePt(self,x,y,z)	xyz 平面位置	设置任意对称面位置
	create(self)		完成创建
	edit(self)		完成编辑
MoveFeature 移动对象	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setBodys(self,bodys)	bodys 几何形状 id, 字符串形式	添加几何体
	SaveOrigin(self, save)	save 是否保存	设置是否保存源对象
	TransformMethod(self,m)	method 移动方	设置移动的方法

	ethod)	法	
	setStartPoint(self,x,y,z)	xyz 起始点坐标	设置移动参考起始点
	setEndPoint(self,x,y,z)	xyz 终止点坐标	设置移动参考终止点
	setDirection(self,x,y,z)	xyz 移动方向	设置移动方向
	setLength(self,len)	len 移动距离	设置移动距离
	setReverse(self,reverse)	reverse 是否反向	设置是否反向
	create(self)		完成创建
	edit(self)		完成编辑
RotateFeature 转动特征	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	saveOrigin(self, save)	save 是否保存	设置是否保存源对象
	reverse(self)		设置反向
	appendObject(self, id)	id 旋转形状 id	添加旋转对象
	setAxisFromBody(self, body, edge)	body 形状 id edge 边索引	从几何形状上选择边作为旋转轴
	setBasicPoint(self, x, y, z):	xyz 选择轴基准点	设置旋转轴基准点
	setAxis(self, x, y, z)	xyz 旋转轴方向	设置旋转轴方向
	setAngle(self,ang)	ang 角度, 角度制	设置旋转角度
	create(self)		完成创建
	edit(self)		完成编辑
MakeMatrix 阵列	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setBodyys(self,bodyys)	bodyys 几何形状 id, 字符串形	添加几何体

		式	
	setOptionMethod(self,optionstr)	optstr 阵列方式 线性阵列 环形阵列	设置阵列方式
	setDirection1(self,x,y,z)	xyz 线性阵列方向 1	设置线性阵列方向 1
	setReverseOfDirecrtion1(self,revstr)	revstr 是否反向	设置方向 1 反向
	setDistance1(self,dis1)	dis1 间隔距离	设置方向 1 阵列间隔
	setCount1(self,count1)	count1 阵列数量	设置方向 1 阵列个数
	showDirection2(self,showdir2str)	showdir2str 是否启用方向 2	设置方向 2 启用状态
	setDirection2(self,x,y,z)	xyz 线性阵列方向 2	设置线性阵列方向 2
	setReverseOfDirecrtion2(self,revstr)	revstr 是否反向	设置方向 2 反向
	setDistance2(self,dis2)	dis2 间隔距离	设置方向 2 阵列间隔
	setCount2(self,count2)	count2 阵列数量	设置方向 2 阵列个数
	setBasicPoint(self,x,y,z)	xyz 环形阵列轴线基准	设置环形阵列轴线基准
	setAxis(self,x,y,z)	xyz 环形阵列轴线方向	设置环形阵列轴线方向
	setWireReverse(self,wirerestr)	wirerestr 是否反向	设置环形阵列是否反向
	setWireCount(self,wirecount)	wireCount 环形阵列个数	设置环形阵列数量
	setDegree(self,degree)	degree 间隔角度, 角度制	设置环形阵列角度

	create(self)		完成创建
	edit(self)		完成编辑
Extrusion 拉伸	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setName(self, name)	name 名称	设置名称
	setDistance(self,distance) :	distance 拉伸 距离	设置拉伸距离
	setDirection(self,x,y,z)	xyz 拉伸方向	设置拉伸方向
	appendEdge(self, geoset, index)	geoset 形状 id index 边的索引	添加边
	Reverse(self,reverse)	reverse 反向	设置反向
	GenerateSolid(self,solid)	solid 是否生成 实体	设置是否生成实体
	create(self)		完成创建
	edit(self)		完成编辑
Revol 旋转	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setName(self, name)	name 名称	设置名称
	setDegree(self,degree)	degree 旋转角 度, 角度制	设置旋转角度
	setAxisMethod(self,option index)	optionIndex 方 法索引	设置旋转轴的指定方 法
	appendEdge(self, geoset, index)	geoset 形状 id index 边的索引	添加边
	selectAxisOnGeo(self,axi sset,edgeindex)	axisset 边所在 形状的 ID edgeIndex 边索 引	从已经存在的形状上 选择边作为旋转轴
	setBasicPoint(self,x,y,z)	xyz 旋转轴基 准点	设置旋转轴基准点

	setCoordinate(self,x,y,z)	xyz 旋转轴方向	设置旋转轴方向
	Reverse(self,reverse)	reverse 反向	设置反向
	GenerateSolid(self,solid)	solid 是否生成实体	设置是否生成实体
	create(self)		完成创建
	edit(self)		完成编辑
Loft 放样	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setName(self, name)	name 名称	设置名称
	isSolid(self,solid)	solid 是否生成实体	设置是否生成实体
	appendSection(self,sec)	sec 截面信息, 字符串	添加截面
	create(self)		完成创建
	edit(self)		完成编辑
Sweep 扫略	setEditID(self, id)	Id 形状 ID	设置要编辑形状的 ID
	setName(self, name)	name 名称	设置名称
	isSolid(self,solid)	solid 是否生成实体	设置是否生成实体
	setPath(self,pathset,pathedge)	pathset 路径所在形状 id pathedge 边索引	设置扫略路径
	create(self)		完成创建
	edit(self)		完成编辑

七、Post.py

该文件中的封装的方法为后处理相关的方法，其 C++代码实现位于

MainWidgets 文件夹，这些方法会调用 ControlPanel.py 封装的方法，主要的接口如下：

表 17 Post.py 接口方法

方法	参数	功能
viewCounter(caseId,variable)	caseid 算例 ID variable 变量名称	查看云图
viewVector(caseId,variable)	caseid 算例 ID variable 变量名称	查看矢量图

八、Mesher.py

该文件中封装了与网格划分相关的操作，目前平台集成了 Gmsh 网格划分器，所以目前该文件中封装的是与 Gmsh 相关的操作，与几何建模部分相同，这部分也是采用了类的封装方法。主要接口如下：

表 18 Mesher.py 接口方法

类	方法	参数	功能
Gmsher Gmsh 网格 划分器封装	appendSurface(self, geoset, index)	geoset 面所在形状 id index 面索引	添加要进行网格划分的面
	appendSolid(self, id)	id 几何形状 id	添加实体
	setElementType(self, etype)	etype 网格单元类型	设置网格单元类型
	setElementOrder(self, order)	order 阶次	设置网格单元阶次
	setMethod(self, m)	m 网格划分方法	设置网格划分方法
	setSizeFactor(self, f)	f 尺寸因子	设置网格尺寸因子
	setMinSize(self, min)	min 最小单元尺寸	设置最小单元尺寸
	setMaxSize(self, max)	max 最大单元尺寸	设置最大单元尺寸
	cleanGeo(self)		启用简单几何清理
	setSmoothIteration(self, it)	it 迭代次数	设置网格光滑次数

	it)		
	startGenerationThread(se lf)		生成网格

Python 脚本封装是一个较为复杂的工作，涉及到多线程等调试难点，因此有一些功能尚未封装脚本，开发团队将会持续跟进更新。



Python 功能拓展

FastCAE 研发小组

2020 年 1 月

目录

一、总览.....	66
二、Python 代码加载	66
三、FastCAE 调用 Python.....	66
四、Python 调用 C++.....	67

一、总览

基于 Python 语言的拓展性优势,以及原生的 C 语言接口, FastCAE 以 Python 语言作为脚本,对主要的功能进行脚本封装,形成了六个脚本模块。各个模块之间通过 Python 命令相互调用,降低了模块之间的耦合度,同时也能够达到自动化流程化仿真分析的效果。这也为 FastCAE 的功能拓展提供了另外一种可能——通过 Python 程序进行拓展。

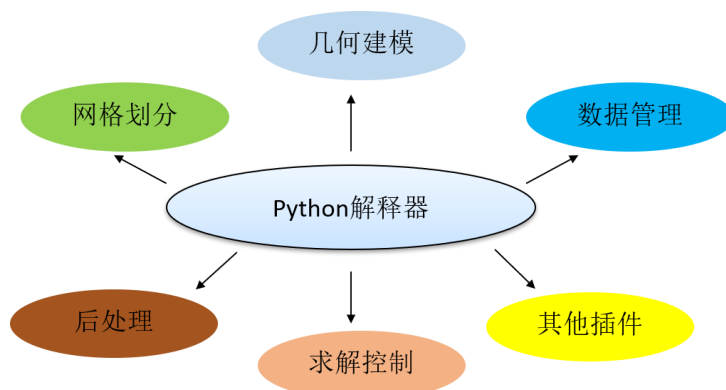


图 13 python 拓展功能

FastCAE 的 PythonModule 封装了 Python 解释器,可直接执行 Python 代码和文件,需要说明的是,这里封装的**解释器为 Python3.7 版本**,也就是说所有的 Python 代码需要支持 3.7 版本,否则可能会有错误风险。

二、Python 代码加载

在执行 Python 代码之前,需要将我们自己写的 Python 代码 import 进来, Python 源码的后缀为*.py,将这些 py 文件拷贝到 FastCAE 的可执行程序的根目录下(也就是 exe 的同级目录下),FastCAE 程序启动时 PythonModule 模块会自动加载根目录下的所有 py 文件。**需要说明的是,py 文件加载只发生在程序启动时,如果有新的 py 文件需要加载或者修改,需要重新启动程序。**

三、FastCAE 调用 Python

PythonModule 中已经定义了执行 Python 代码的方法,将 Python 语句转换成字符串的形式即可执行,执行方法位于源码 python/PyAgent.h。

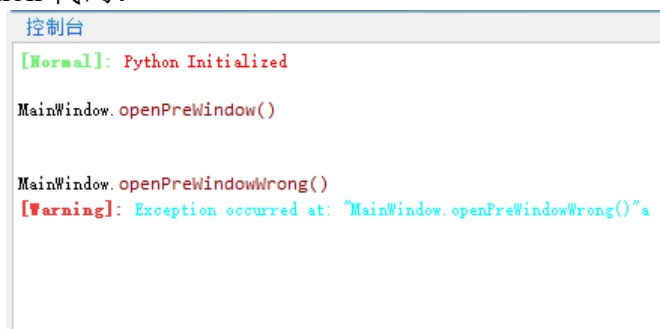
```
void submit(QString code, bool save = true);
```

参数 `code` 为字符串形式的 python 语句，`save` 为是否将该执行命令保存到命令列表，如果保存到命令列表，保存脚本时会将该命令以脚本的形式写出。

`Py::PythonAgent` 为单例类，在使用的过程中可以通过以下的 C++ 代码进行 Python 命令的提交执行：

```
QString codes = QString("**python code**");
Py::PythonAgent::getInstance()->submit(codes);
```

还有一种方法就是在主界面的控制台窗口内直接通过键盘输入 python 命令的形式执行 python 代码。



```
控制台
[Normal]: Python Initialized
MainWindow.openPreWindow()

MainWindow.openPreWindowWrong()
[Warning]: Exception occurred at: "MainWindow.openPreWindowWrong()"a
```

如果代码执行错误，将会抛出相应的异常警告。

四、Python 调用 C++

Python 的优点是开发效率高，使用方便，C++ 则是运行效率高，这两者可以相辅相成，在 Python 项目中嵌入 C++ 代码，能够提高开发效率。Python 调用 C++ 的方法有很多，各方面的资料也有很多，下面将介绍 FastCAE 源码封装的过程中用到的一种方法，这种方法用的是 python 的原生 C 语言调用接口

- 第一步需要将 C++ 功能封装出 C 语言的接口，例如：

```
extern "C"
{
    //声明 C 语言接口
    void MAINWINDOWAPI importMesh(char* f);
}
void importMesh(char* f)
{
    //调用 C++ 的函数实现具体的功能
    GUI::MainWindowPy::importMesh(f);
}
```

- 第二步，创建 py 文件，通过 python 的方法，打开动态链接库，这里动态库的名称为 `MainWindow.dll`，这里创建的 py 文件也是 `MainWindow.py`。

```
1  #-----关联动态库-----
2  import ctypes
3  import platform
4  from ctypes import *
5  system = platform.system()
6  if system == "Windows":
7      pre = "./"
8      suff = ".dll"
9  else:
10     pre = "./lib"
11     suff = ".so"
12
13     libfile = ctypes.cdll.LoadLibrary
14     filename = pre+"MainWindow"+suff
15
16     mw = libfile(filename)
17     #-----
```

- 第三步，定义 Python 函数，调用动态库中的方法。

```
32
33  def importMesh(filename):
34      str = bytes(filename,encoding='utf-8')
35      mw.importMesh(str)
36      pass
37
```

- 第四步，将这个写好的 py 文件拷贝到 exe 的同级目录下，就可以在程序中调用了。



插件开发说明

FastCAE 研发小组

2020 年 1 月

目录

一、插件总览.....	71
二、插件基类.....	71
三、插件装载.....	72
四、插件卸载.....	72
五、输入输出拓展插件.....	73
5.1 接口.....	73
5.2 示例.....	73
六、自定义工具插件.....	75
6.1 接口.....	75
6.2 示例.....	75
七、模型拓展插件.....	77
7.1 接口.....	77
7.2 示例.....	77
八、材料拓展插件.....	80
8.1 接口.....	80
8.2 示例.....	80

一、插件总览

FastCAE 是一款国产 CAE 软件研发支撑平台，旨在辅助求解程序开发者能够快速完成一款完整的 CAE 软件的开发，平台提供可较为完善的通用前后处理功能，用户可以根据需要进行选择使用。面对各种各样的求解计算程序，单一的仿真流程和固定的功能难以满足用户的个性化的定制需求，因此，平台提供了一些插件的接口用于实现用户自己的功能需求和数据处理需要。

插件从软件的结构层级来说，高于所有的软件功能模块，因此插件可以调用任意的软件接口。目前，FastCAE 的插件可以分为以下四类：输入输出拓展，用户自定义工具，模型拓展，材料拓展。

输入输出拓展为与平台进行数据交换的接口拓展，目的是增加数据适配器，增加支持的文件格式；用户自定义工具为增加交互的接口，为用户提供个性化的数据处理方法；模型拓展旨在让用户能够按照自己的需要定制仿真流程的树形菜单，以及拓展原有物理模型的数据表示。

二、插件基类

FastCAE 采用 C++ 语言开发，所有的底层 API 均为 C++ 语言，所以插件的开发也要求使用 C++ 语言，平台会对插件进行统一的维护与管理，因此要求所有的插件必须继承自基类 `Plugins::PluginBase`，其中 `Plugins` 为命名空间。该类的头文件位于 `PluginManager/pluginBase.h`。在插件开发的过程中，基类中的几个虚函数要求重写：

```
virtual bool install() = 0;
```

插件加载，插件加载时，需要向主程序注入一些接口，这些注册的接口需要在该纯虚函数中进行重写定义，返回值为是否注入成功，该函数由平台主程序调用。

```
virtual bool uninstall() = 0;
```

插件卸载，该函数要求实现插件卸载时，将向主程序注入的接口擦除。返回值为是否卸载成功，该函数由平台的主程序调用。

```
virtual void reTranslate(QString lang);
```

双语翻译，该函数要求实现插件的双语翻译，翻译的对象为自定义的 GUI 界面。参数 `lang` 的值有两种可能，分别为“English”和“Chinese”。该函数由平台调用。函数的重写实现需要遵循 Qt5 的双语处理方法。

```
virtual void readINI(QSettings* settings);
virtual void writeINI(QSettings* settings);
```

上面两个方法为从 INI 中读取配置信息的接口，readINI 由基类的 install() 函数调用，writeINI 由基类的 uninstall() 函数调用。

三、插件装载

插件的本质是一个动态链接库，平台要求插件动态链接库命名以“Plugin”开头，在 linux 下以“libPlugin”开头，插件的存放目录为可执行程序的同级“plugins”文件夹。平台在加载动态链接库的时候采用根据函数名字的方法去寻找入口，因此是所有的插件必须有一个名为“Register”的函数作为入口，声明方式如下：

```
extern "C"
{
    void MODELPLUGINAPI Register(GUI::MainWindow* m, QList<Plugins::PluginBase*>* plugs);
}
```

函数实现如下：

```
void Register(GUI::MainWindow* m, QList<Plugins::PluginBase*>* ps)
{
    Plugins::PluginBase* p = new Plugins::ModelPlugin(m);
    ps->append(p);
}
```

该函数的作用是创建插件类，并将控制权交给主程序，之后主程序才能通过多态的方式调用 PluginBase 子类的重写的函数。其中 GUI::MainWindow* 为程序从主窗口指针，QList<Plugins::PluginBase*>* 为插件管理列表的指针。该函数在加载时由主程序调用。综上所述，插件的加载顺序为：

- 打开动态链接库；
- 寻找函数名为“**Register**”的函数，并执行，创建插件类，交给主程序维护；
- 调用 PluginBase 子类的 install() 函数，向主程序注入接口。

四、插件卸载

插件卸载比装载简单很多，由于插件列表在主程序维护，所以主程序可以轻易获取插件指针，执行对应的 uninstall() 函数即可完成对注入接口的擦除，完成

软件卸载。插件卸载的操作全部由主程序控制。

五、输入输出拓展插件

5.1 接口

输入输出拓展插件需要做的是将文件写入写出的方法按照后缀注册到主程序中，目前开放的注册的接口有四个：导入网格，导出网格，求解输入文件写出，求解输出文件转化。

对应的注册方法位于 IO/IOConfig.h，分别为：

```
//注册写出的后缀与方法
static void RegisterInputFile(QString suffix, WRITEINPFILE fun);
//注册结果文件转换方法
static void RegisterOutputTransfer(QString name, TRANSFEROUTFILE fun);
//注册网格文件导入的方法
static void RegisterMeshImporter(QString suffix, IMPORTMESHFUN fun);
//注册网格导出的方法
static void RegisterMeshExporter(QString suffix, EXPORTMESHFUN fun);
```

这些函数的第二个参数都是函数指针，其定义分别为：

```
//求解输入文件  写出的路径，不包含文件名称  模型指针
typedef bool(*WRITEINPFILE)(QString, ModelData::ModelDataBase*);
//结果文件转换  算例路径
typedef bool(*TRANSFEROUTFILE)(QString);
//导入网格  文件全名称，包含路径
typedef bool(*IMPORTMESHFUN)(QString);
//导出网格  文件全名称  网格 id
typedef bool(*EXPORTMESHFUN)(QString, int);
```

接口移除的方法同样在 IO/IOConfig.h，分别为：

```
//移除写出文件的后缀注册
static void RemoveInputFile(QString s);
//移除结果文件转换
static void RemoveOutputTransfer(QString name);
//移除导入网格的方法
static void RemoveMeshImporter(QString suffix);
//移除导出网格的方法
static void RemoveMeshExporter(QString suffix);
```

5.2 示例

这四个接口的使用方法基本一致，下面将以求解器输入文件的写出注册插件

为例，演示插件的使用方法与效果。

- 定义插件类与注册接口如下，其中 `WriteOut` 函数为需要注册的文件写出方法。

```
namespace Plugins
{
    class MODELPLUGINAPI ModelPlugin : public PluginBase
    {
        Q_OBJECT
    public:
        ModelPlugin(GUI::MainWindow* m);
        ~ModelPlugin();

        //加载插件
        bool install() override;
        //卸载插件
        bool uninstall() override;

    private:
        GUI::MainWindow* _mainwindow{};
    };
}
extern "C"
{
    void MODELPLUGINAPI Register(GUI::MainWindow* m, QList<Plugins::P
uginBase*>* plugs);
    //求解器文件写出
    bool MODELPLUGINAPI WriteOut(QString path, ModelData::ModelDataBa
se* d);
}
```

- 具体实现 `WriteOut` 函数，这里涉及到具体的业务逻辑，需要根据需要具体的要求完成，这里不做详细说明。
- 完成插件加载与卸载的操作如下：

```
bool ModelPlugin::install()
{
    //选择 out 文件时，执行 writeout
    IO::IOConfigure::RegisterInputFile("out", WriteOut);
    return true;
}

bool ModelPlugin::uninstall()
{
    IO::IOConfigure::RemoveInputFile("out");
    return true;
}
```

在主窗口菜单栏【插件】-【插件管理】加载插件之后，在求解器的管理对话框中出现注册的 out 选项。

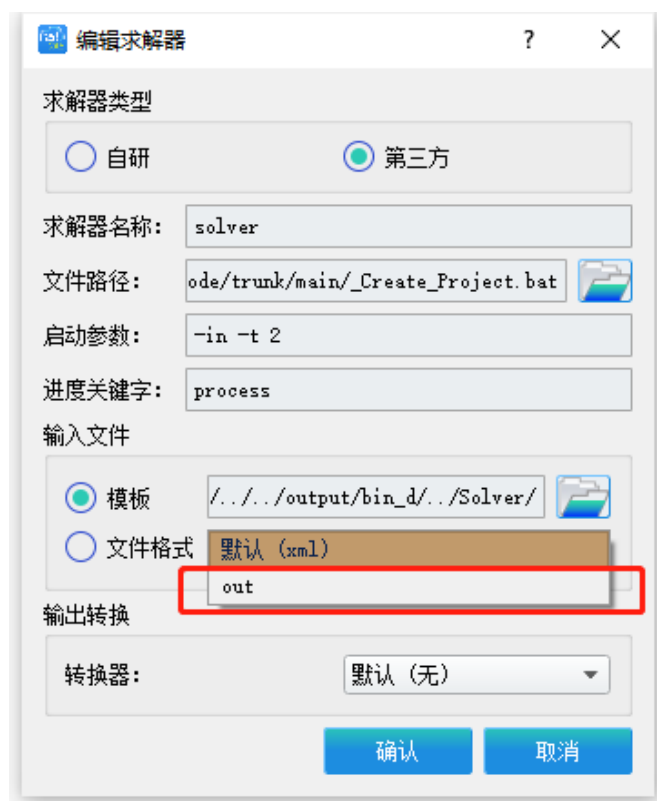


图 14 编辑求解器窗口

其他的注册也是一样的使用方法，网格导入导出注册完成之后，在网格导入导出的对话框中会出现相应的后缀供选择。求解器的输出转化注册后，会在转换器下拉菜单下有相应的选项。

六、自定义工具插件

6.1 接口

自定义工具插件的开发方式与输入输出插件开发方式基本一致，只是在装载和卸载时候执行的操作不太一样，自定义工具往往需要单独的按钮或者菜单栏，加载插件的过程往往就是在主窗口添加菜单和按钮的过程，插件的卸载过程就是在移除这些菜单和按钮的过程。添加菜单和按钮直接调用 QMainWindow 的 API 即可实现。具体的功能实现则依赖于槽函数和调用平台本身的 API 接口实现。

6.2 示例

下面是一个简单的例子：

```
bool ComplexPlugin::install()
{
    QString des = this->getDescribe();
    //创建菜单和工具栏
    _menu = _mainwindow->menuBar()->addMenu(des);
    _toolBar = _mainwindow->addToolBar(des);
    //创建动作
    QAction* cyac = new QAction(tr("Connecting Rod"), _mainwindow);
    cyac->setIcon(QIcon(":/icons/liangan1.png"));
    this->addAction(cyac);
    _actionList.append(cyac);

    QAction* boxac = new QAction(tr("Floating Body"), _mainwindow);
    boxac->setIcon(QIcon(":/icons/futi1.png"));
    this->addAction(boxac);
    _actionList.append(boxac);
    //关联信号
    connect(cyac, SIGNAL(triggered()), this, SLOT(CreateCylindricalComplex()));
    connect(boxac, SIGNAL(triggered()), this, SLOT(CreateBoxComplex()));
;

    PluginBase::install();
    return true;
}

bool ComplexPlugin::uninstall()
{
    //删除菜单栏和工具栏
    _menu->clear();
    delete _menu;
    _menu = nullptr;
    _mainwindow->removeToolBar(_toolBar);
    PluginBase::uninstall();
    return true;
}
```

插件加载之后在菜单栏和工具栏出现对应的按钮

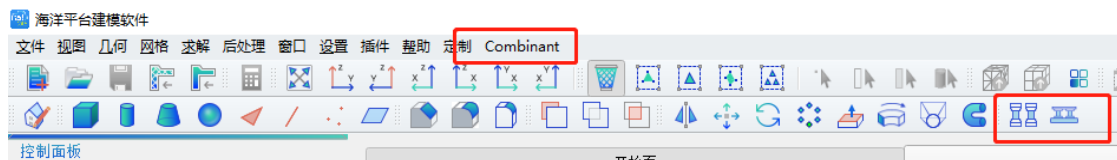


图 15 插件加载后按钮

七、模型拓展插件

模型拓展插件是指对仿真过程中的树形菜单和模型数据进行拓展，接口的功能在于注册并关联类型和创建的方法。在 FastCAE 中，每一个算例树都对应一个模型数据，二者通过一个枚举值标记类型，并进行关联。

7.1 接口

对于模型树的注册接口位于 MainWidget/ProjectTreeFactory.h，注册接口和移除接口分别为：

```
//注册树的类型和创建方法
static void registerType(int type, CREATETREE f);
//移除注册的树的类型
static void removeType(int type);
```

模型树创建方法定义如下：

```
typedef bool(*CREATETREE)(int, GUI::MainWindow*, QPair<int,ProjectTree:
:ProjectTreeBase*>* );
```

对于数据模型的注册接口位于 ModelData/ModelDataFactory.h，注册接口和移除接口分别为：

```
//注册模型的类型，名称和创建方法
static void registerType(int type, QString name, CREATEMODEL);
//移除注册的数据模型的类型
static void removeType(int type);
```

数据模型创建方法定义如下：

```
typedef bool(*CREATEMODEL)(int,QPair<int,ModelData::ModelDataBase*>*);
```

7.2 示例

基本流程这里不在赘述，这里只列举插件装载和卸载的方法。下面的 PluginDefType 为一个枚举类型，用户可以自由选择偏移，只要不重复即可。文件位置为 DataProperty/ modelTreeItemType.h。

```

bool ModelPlugin::install()
{
    ModelData::ModelDataFactory::registerType(PluginDefType+1, "PluginType0001", CreateModel);
    MainWindow::ProjectTreeFactory::registerType(PluginDefType + 1, CreateTree);
    return true;
}
bool ModelPlugin::uninstall()
{
    ModelData::ModelDataFactory::removeType(PluginDefType + 1);
    MainWindow::ProjectTreeFactory::removeType(PluginDefType + 1);

    return true;
}

```

其中注册的两个方法 CreateModel 和 CreateTree 也需要声明为：

```

extern "C"
{
    bool MODELPLUGINAPI CreateModel(int t, QPair<int, ModelData::ModelDataBase*>*>);
    bool MODELPLUGINAPI CreateTree(int, GUI::MainWindow* m, QPair<int, ProjectTree::ProjectTreeBase*>*>);
}

```

具体的实现为如下，其中 ModelData::ModelDataPlugin 为 ModelData::ModelDataBase 的子类；ProjectTree::PluginTree 为 ProjectTree::ProjectTreeBase 的子类。

```

bool CreateModel(int t, QPair<int, ModelData::ModelDataBase*>*> p)
{
    if (t == ProjectTreeType::PluginDefType + 1)
    {
        auto m = new ModelData::ModelDataPlugin((ProjectTreeType)t);
        p->first = t;
        p->second = m;
        return true;
    }
    return false;
}

bool CreateTree(int tp, GUI::MainWindow* m, QPair<int, ProjectTree::ProjectTreeBase*>*> t)
{

```

```

    if (tp == PluginDefType + 1)
    {
        t->first = tp;
        t->second = new ProjectTree::PluginTree(m);
        return true;
    }
    return false;
}

```

需要特别说明的是，在 ModelData::ModelDataBase 的子类中有几个虚函数是必须需要重写的：

```

/*获取 MD5 的 stream*/
virtual void dataToStream(QDataStream* datas) override;
//数据写出到工程文件 子类必须重写该函数
virtual QDomElement& writeToProjectFile(QDomDocument* doc, QDomElement*
ele) override;
///从工程文件块读入数据，子类必须重写此函数
virtual void readDataFromProjectFile(QDomElement* e) override;
///将数据写出文本给求解器
virtual void writeToSolverText(QTextStream* stream);
///将数据写出 XML 给求解器
virtual void writeToSolverXML(QDomDocument* doc, QDomElement* e);

```

同样的，在 ProjectTree::ProjectTreeBase 的子类中有几个虚函数是必须需要重写的：

```

///创建树
virtual void createTree(QTreeWidgetItem* root, GUI::MainWindow* mainwin
dow);
///更新树
virtual void updateTree();
///双语切换
virtual void reTranslate();
///树节点单击事件
virtual void singleClicked();
///树节点双击事件
virtual void doubleClicked();
///树节点右键菜单
virtual void contextMenu(QMenu* menu);

```

插件加载之后在创建算例时，类型下拉列表将会显示新注册的类型（如下图）：

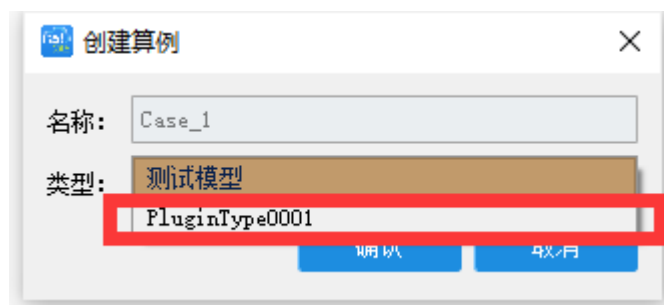


图 16 创建算例

八、材料拓展插件

8.1 接口

材料拓展插件与模型拓展插件类似，都是通过不同的类型标记来实现创建具有不同属性的材料，实现方法与上面所说的相同，都是通过函数注册的方法实现。

注册和移除接口位于 Material/MaterialFactory.h，分别为：

```
//注册材料创建方法
static void registerType(QString type, CREATMATERIAL);
//移除材料创建方法
static void removeType(QString type);
```

创建材料的方法定义如下：

```
typedef Material*(*CREATMATERIAL)(QString);
```

8.2 示例

与模型拓展插件一样，首先需要完成对插件子类的定义，这个不再做过多说明。然后需要对材料的创建方法进行声明与定义：

声明如下：

```
extern "C"
{
    //创建材料
    Material::Material* createMaterial(QString);
}
```

定义如下：

```
Material::Material* createMaterial(QString t)
{
```

```

if (t != "ma1") return nullptr;

auto m = new Material::MaterialExt;
return m;
}

```

需要说明的是，这里创建的必须是 `Material::Material` 的子类，必须要重写下面几个函数：

```

//从工程文件读入
void readDataFromProjectFile(QDomElement* e) override;
//写出到工程文件
QDomElement& writeToProjectFile(QDomDocument* doc, QDomElement* parent)
override;

```

最后需要在插件的装载与卸载函数中完成对材料创建方法的注册与移除：

```

bool ModelPlugin::install()
{
    Material::MaterialFactory::registerType("ma1", createMaterial);
    return true;
}

bool ModelPlugin::uninstall()
{
    Material::MaterialFactory::removeType("ma1");
    return true;
}

```

插件加载之后，将会在创建材料对话框中出现对应的类型：

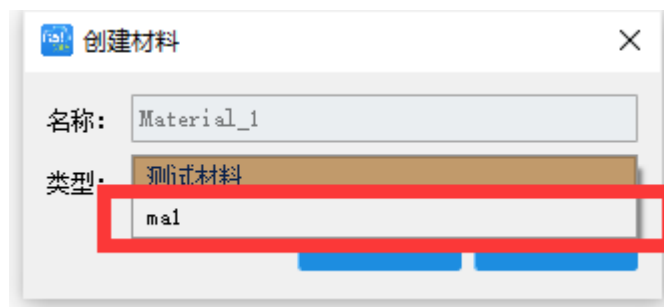


图 17 创建材料