插件系统开发与使用

说明书

青岛数智船海科技有限公司

时间	修改纲要	修改人
2024/12/18	首次编制	李宝君

一、插件系统

为满足多学科多领域的仿真分析软件的个性化功能开发需求，FastCAE 集成平台提供了插件系统。FastCAE 插件系统旨在为用户提供一个灵活且强大的工具，通过该系统，用户不仅可以访问预装的核心功能，还可以通过安装来自社区或其他来源的插件来扩展软件解决具体工程问题的能力。插件系统与组件化集成一起为软件的动态集成提供了高度的灵活性与可拓展性，插件系统作为组件化集成的补充，主要应用与个性化功能与定制化小系统的开发与集成。插件系统具有以下特点：

- 接口标准化：所有插件都必须遵守一套由 FastCAE 定义的标准 API，确保它们能与主程序无缝集成。
- 模块化设计：每个插件都是一个独立的模块，可以单独开发、测试、部署和更新，而不影响其他部分或核心应用。
- 动态加载：插件可以在不重启应用程序的情况下被加载或卸载，提高了使用的灵活性。

插件的本质是动态链接库，通过插件可以实现软件功能的重新组合与封装，其典型应用场景包含如下：

- 自定义工作流：用户可以根据项目需求创建特定的工作流程。
- 数据导入/导出：支持更多格式的数据文件，便于与其他软件交互。
- 可视化增强：添加新的图表类型或改进现有视图，提高数据分析能力。
- 算法扩展：引入新的求解器或其他算法，适应更广泛的工程问题

二、插件系统组成

插件系统是应用程序框架（FITKAppFramework）的重要组成部分。包含两部分，插件抽象类与插件管理器。

类名	所属源文件	描述
FITKAbstractPlugin	FITKAbstractPlugin	插件对象抽象类，定义抽象接口，包括加载、卸载、获取名称、执行功能等。

FITKPluginsManager	FITKPluginsManager	插件管理类，实现对插件动态加载与卸载，提供了多种对插件操作的接口。

三、插件开发示例

插件的本质是动态库，所以每一个插件都需要编译成一个动态库，在 Windows 平台下为后缀为 dll 的文件，在 Linux 平台下为结尾为 so 的文件。插件可能会调用其他的第三方库，当使用第三方的动态链接库的时候建议将第三方的动态库复制到 exe 的同级目录，否则可能会出现加载动态库失败的情况。实现一个插件的过程分为三步：

1. 插件类创建

继承插件抽象基类 AppFrame::FITKAbstractPlugin，实现 install 与 uninstall 两个纯虚函数。示例如下：

```
class FITKPluginDemo : public AppFrame::FITKAbstractPlugin
{
public:
    explicit FITKPluginDemo(QLibrary* dylibrary) :
         AppFrame::FITKAbstractPlugin(dylibrary)
    { }
    virtual ~FITKPluginDemo() = default;

    virtual QString getPluginName() override
    {
         return "PluginDemo"
    }

private:
    virtual void install() override
    {
          //do something
    }
    virtual void unInstall() override
     {
          //do something
     }
     virtual bool exec() override
```

```
24.     {
25.         //do some thing
26.     }
```

其中 getPluginName 函数返回的名称为全局唯一的插件名称，将作为插件管理器进行插件管理的重要参数，因此需要保证返回值的唯一性。install 函数是在插件加载时插件管理器主动调用的函数，在该函数中通常是实现插件的初始化操作，例如向主界面中加入按钮、向程序中注入函数指针与新的执行对象等。uninstall 函数是在插件卸载时被插件管理器主动调用的函数，实现插件的卸载，其逻辑与 install 函数相反。exec 函数则是一个抽象接口，供外部调用实现具体的功能逻辑，同时 AppFrame::FITKAbstractPlugin 抽象类是 Core::FITKVarientParams 的子类，因此可以实现抽象的参数传递。

2. 创建动态库识别接口

由于每一个 App 的功能与接口均有所差别，因此在插件接入之前应对动态库进行识别，判断该动态库是否与当前的 App 具有对应关系。当前设计的识别模式为采用密钥的方式进行识别。首先需要在应用程序框架中个性化的指定一个字符串（密钥），示例如下：

```
1.  int main(int argc, char *argv[])
2.  {
3.      // 初始化应用框架
4.      AppFrame::FITKApplication app(argc, argv);
5.      /*注册程序的主要组件和设置
6.       ......
7.      */
8.      //设置插件密钥
9.      app.setPluginKey("MyAppPlugin");
10.     // 运行应用程序的消息循环
11.     return app.exec();
12. }
```

上述代码的第 9 行即为设置插件密钥，这里的密钥可以是任意不为空的字符串，若密钥不设置或设为空，则不能加载任何插件。在插件的动态库中，需要创建一个动态库识别的接口来实现与应用程序框架中的密钥进行识别与匹配，该函数的名称与返回值如下，不能有任何更改，否则将导致插件无法识别。

```
1.  QString FITKLibraryRecognizeFun()
```

而在实际操作中通常将这个函数声明为 C 类型的函数，以保证编译器不会将函数名进行修改。而且会加入声明为接口函数的宏对象（在 Windows 平台下用于代替__declspec(dllexport)和__declspec(dllimport)），下面的示例代码中的 FITKPluginDemoAPI 就是一个声明为接口的宏。在实际操作中的示例代码如下：

```
1.  //函数声明在XXX.h
2.  extern "C"
3.  {
4.      FITKPluginDemoAPI QString FITKLibraryRecognizeFun();
5.  }
6.  //函数实现在XXX.cpp
7.  QString FITKLibraryRecognizeFun()
8.  {
9.      return QString("MyAppPlugin");
10. }
```

3. 创建插件创建接口

插件创建接口目的是在动态库被识别之后创建并返回插件对象。简单来说就是每一个插件的动态库中都必须包含一个如下的函数，函数名、返回值、函数参数必须一致，否则将会导致插件加载失败。

```
1.  AppFrame::FITKAbstractPlugin* FITKLibraryRecogFun(QLibrary*)
```

而在实际操作中通常将这个函数声明为 C 类型的函数，以保证编译器不会将函数名进行修改。而且会加入声明为接口函数的宏对象（在 Windows 平台下用于代替__declspec(dllexport)和__declspec(dllimport)），下面的示例代码中的 FITKPluginDemoAPI 就是一个声明为接口的宏。在实际操作中的示例代码如下：

```
1.  //函数声明在XXX.h
2.  extern "C"
3.  {
4.      FITKPluginDemoAPI AppFrame::FITKAbstractPlugin* FITKLibraryLo
    adFun(QLibrary*);
5.  }
6.
7.  //函数实现在XXX.cpp
8.  #include “XXX.h”
9.  AppFrame::FITKAbstractPlugin*  FITKLibraryLoadFun(QLibrary* lib)
```

```
{
    return new FITKPluginDemo(lib);
}
```

由于第 2 步与第 3 步中的接口均为 C 语言风格的接口，因此在函数声明时，可以将二者合并，如下：

```
extern "C"
{
    FITKPluginDemoAPI QString FITKLibraryRecognizeFun();
    FITKPluginDemoAPI AppFrame::FITKAbstractPlugin* FITKLibraryLoadFun(QLibrary*);
}
```

四、插件使用

插件使用通过插件管理器(AppFrame::FITKPluginsManager)实现。插件管理器通过应用程序框架获取，代码如下：

```
AppFrame::FITKPluginsManager* pMgr = FITKAPP->getPluginsManager();
```

插件管理器(AppFrame::FITKPluginsManager)的接口主要分为三类，加载插件、卸载插件、查询插件。加载插件具有唯一的接口如下。卸载插件与查询插件的接口比较多，可根据路径、名称、索引等查询，详细接口可查阅接口文档。

```
/**
* @brief 加载动态库
* @param[i]  libPath        动态库路径，完整全路径
* @return true   加载成功
* @return false  加载失败
*/
bool installLibrary(const QString& libPath);
```

除此之外在使用插件时一定要注意在主程序中设置插件密钥，并且该密钥能够与插件匹配，否则会出现加载失败。详细说明请参考第三章第 2 部分。

在程序关闭的时候，如果插件未卸载，将会将插件加载的路径写入到配置文件（ini 文

件），等程序再次启动时，将会自动加载配置文件中的插件。