

M I D I S

开 发 文 档

微 波 智 能 集 成 设 计

Microwave Intelligent Design Integration Studio

MIDIS 3.0



IEI 智能电子学研究所

2019-04

<https://www.iei-hust.com>

目 录

1 MIDIS 开发环境配置	1
1.1 Python 与相关库安装	1
1.2 SSH 服务配置	2
1.3 数据库配置	3
2 MIDIS 主体框架	7
2.1 wxPydocview 框架	7
2.2 服务插件的编写	10
2.3 本章小结	14
3 分布式计算的实现	15
3.1 任务计算流程	15
3.2 分布式实现	16
3.3 任务管理	22
4 数据库实现	26
4.1 数据库接口类与数据库工具使用	26
4.2 数据库与仿真数据交互	31
4.3 基于数据库的数据分析插件	34
5 MIDIS 用户操作流程	38
5.1 MIDIS 具体操作流程	38
5.2 MIDIS 操作简化流程	48
5.3 MIDIS 源代码打包	49

第 1 章 MIDIS 开发环境配置

本章主要内容

- ◊ Python 与相关库安装
- ◊ SSH 配置
- ◊ 数据库配置

1.1 小节 Python 与相关库安装

由于 Python 各版本之间的兼容性，MIDIS 中需要的第三方库比较多，所以对 Python 环境与库的版本都有一定要求。MIDIS 是使用 Python2.7.13 进行开发，如果使用官网 Python，还需要安装许多额外的第三方库。**Anconda 2 中 2.7 以上版本**集成了很多科学计算的相关库，只需要再装几个库即可运行。如图 1-1 所示是需要额外添加的库及其版本号，其中红色部分是使用 Anconda 环境后需要安装的库。

库名称	版本
wxpython	3.0
numpy	1.11.3
paramiko	1.15.2
six	1.10.0
matplotlib	2.0.0
pyparsing	2.1.4
xlrd	1.0.0
xlwt	1.2.0
xlutils	2.0.0
pyneurgen	0.3.1
neurolab	0.3.5
ecdsa	0.13
pyexcelerator	0.6.4.1
pymongo	3.4.0

图 1-1 MIDIS 中需要添加的第三方库

由于相关库版本的不断更新，有些库在网上难以找到。请开发人员务必对相关库进行保存备份。MIDIS 的编辑器推荐使用 Eclipse 或 VScode。

1.2 小节 SSH 服务配置

MIDIS 分布式计算是利用局域网内的机组互相通信来实现的，首先我们需要通过交换机组建一个局域网机组，局域网内的机器安装 freeSSH 软件，freeSSHD 是 SSH 服务器的免费实现。它通过 Internet 等提供强大的加密和身份验证。由于 built-in SFTP 服务器，用户可以打开远程控制台甚至访问他们的远程文件。针

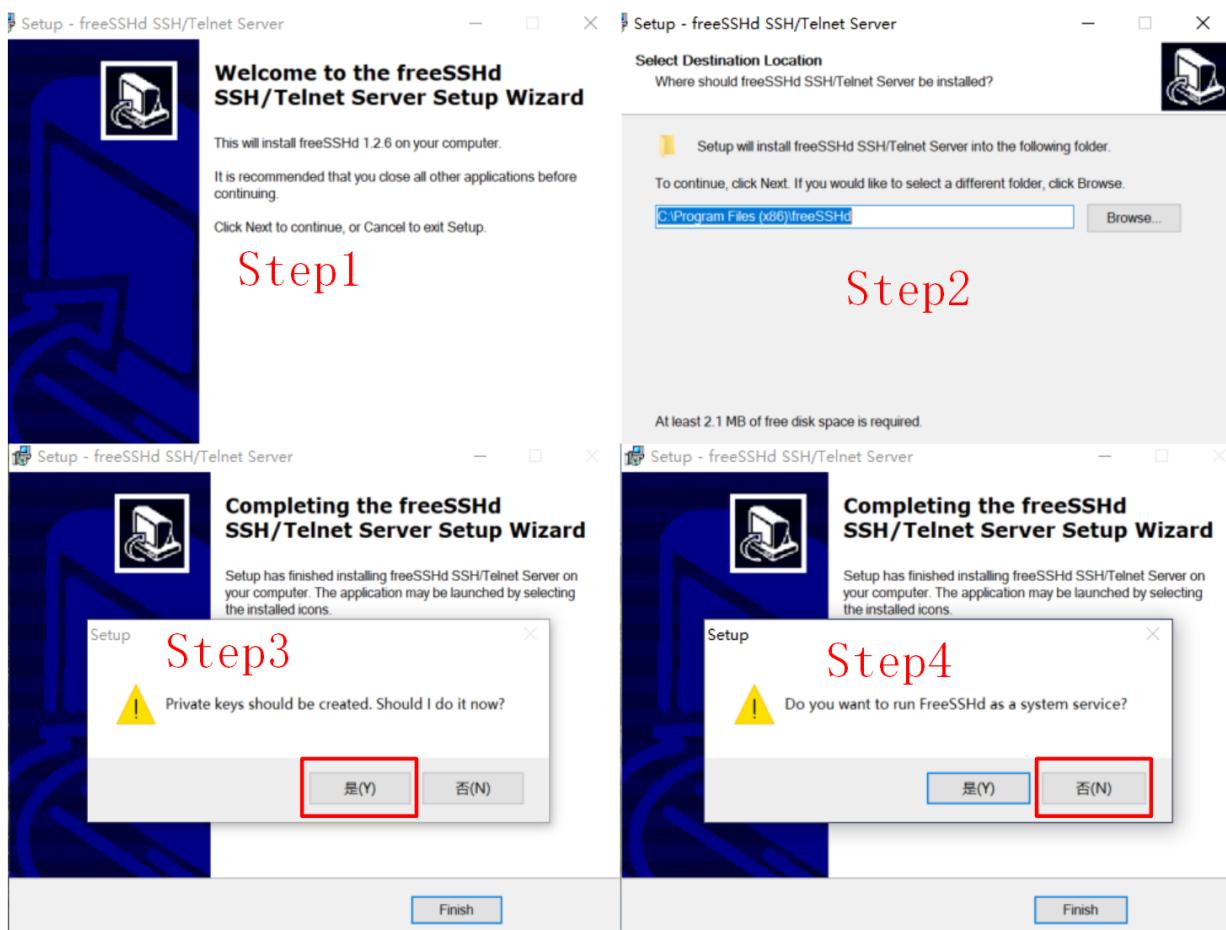


图 1-2 FreeSSHD 安装流程

对 MIDIS 中的 freeSSH，需要一些特定的设置，接下来会详细说明，首先是软件安装，我们使用的是 freeSSHD 1.2.6 版本。如图 1-2 所示，前面的操作按照默认选项即可，关键在于最后的两个步骤，需要按照图中红色方框操作。

如图 1-3 所示，软件安装成功后，需要设置用户和启动服务，在 Users 选项卡中 Add 一个用户，用户名需要和计算机的用户名一致，实验室机群电脑

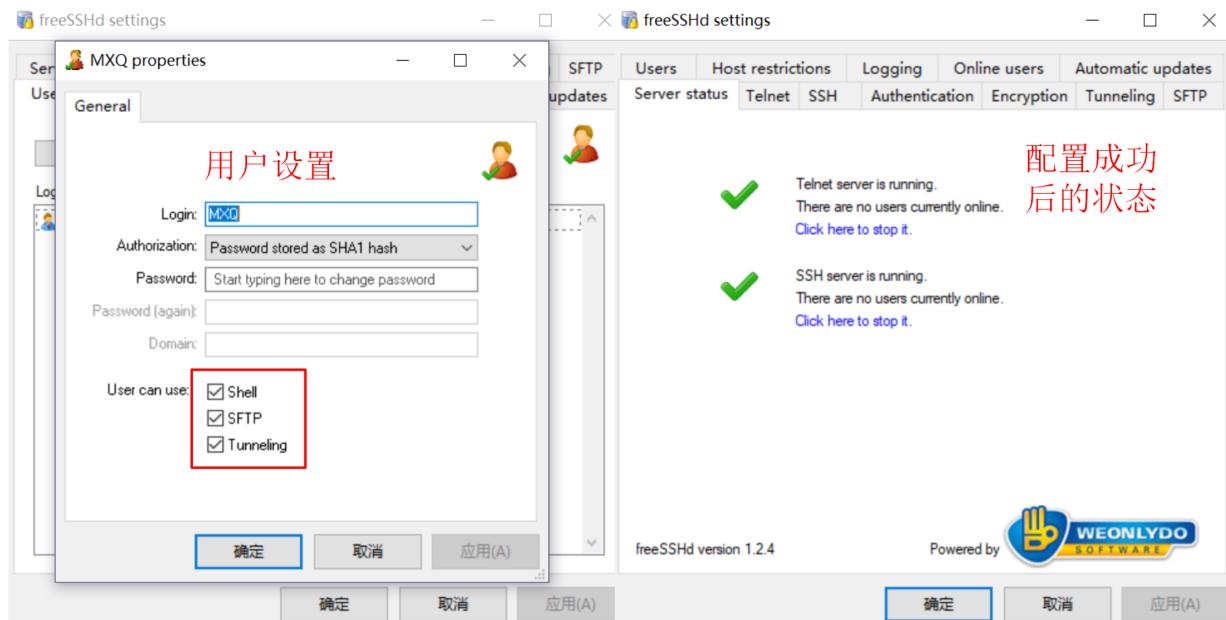


图 1-3 FreeSSHd 用户配置

的用户名都是 Administrator，认证方式选择密码认证，用户名和密码都需要和 MIDIS 中相关代码保持一致，勾选红色方框的选项用来开启相关服务，最后查看 Server Status，开启服务，出现绿色状态表示配置成功。通过 FreeSSHd 接收的文件会存储在 C:\Users\用户名\MIDIS 目录下。

1.3 小节 数据库配置

MIDIS 中使用的是非关系型数据库 MongodB，首先安装 MongodB，这里推荐使用 3.4.4 版本。安装之后，设置环境变量如图 1-4 所示。

然后通过下面脚本启动数据库服务。只有一个变量 -dbpath 用来指定数据保存路径。

1 mongod –dbpath D:\dbpath

2 pause

数据库服务启动之后，需要创建我们需要的数据库，则可以通过以下脚本创建用户名为 IEI3，密码为 IEI 的数据库 HFSS DataBase。以此类推，可以创建其他不同用户下的不同名称的数据库。

1 db.createUser(

```
2 ... {  
3 ...     user: "IEI3",  
4 ...     pwd: "IEI",  
5 ...     roles: [ { role: "readWrite", db: 'HFSSDataBase' } ]  
6 ... }  
7 ... )
```

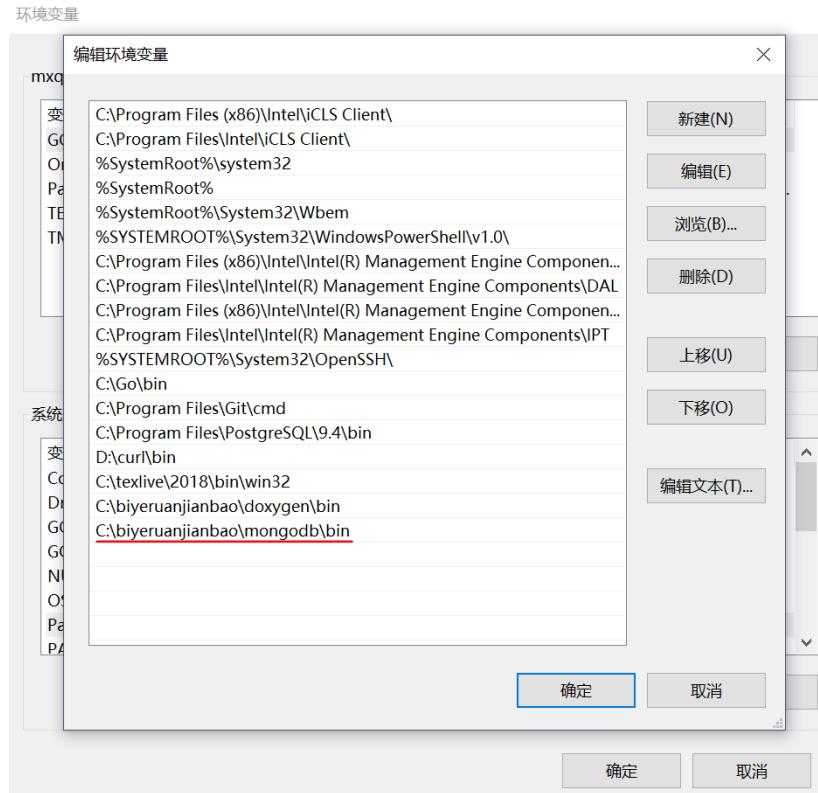


图 1-4 Mongodb 环境变量设置

为方便查看数据库，推荐安装 Mongodb 可视化工具 RoboMongo 1.0.0，效果图如图 1-5 所示。

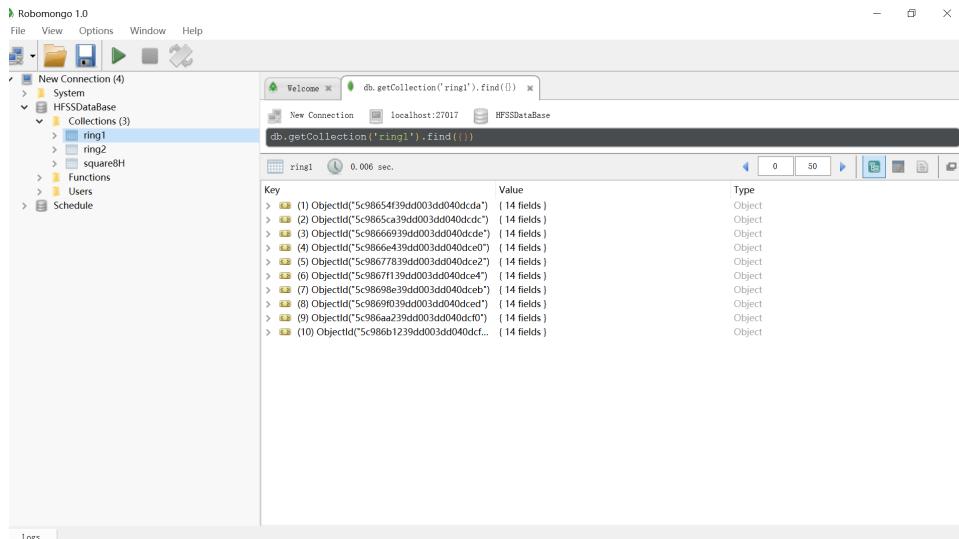


图 1-5 RoboMongo 效果图

这里要说一下，对于数据库 IP 和服务器 IP 的设置，如果是早些版本，这些 IP 可能写在了代码里面，如果使用老版本只能在代码中修改这两种 IP 地址，如果是新的版本，可以通过图 1-6 所示的操作进行配置，配置界面如图 1-7 所示。

通过 + 操作可以添加新的 Server IP，这里密码没有进行加密，主要是考虑

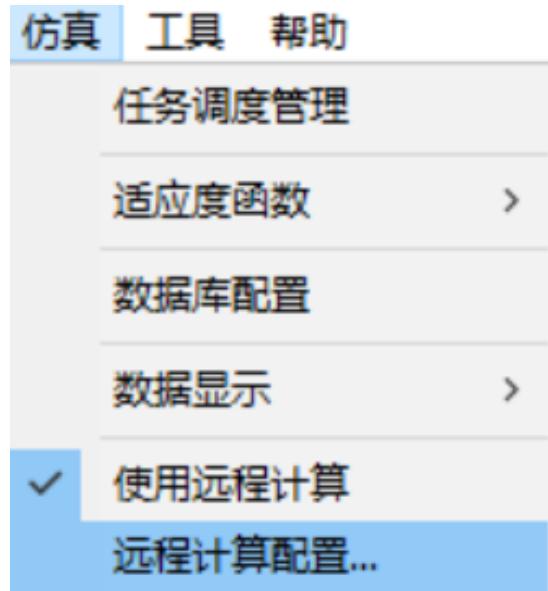


图 1-6 服务器 IP 配置操作 1

到有时需要通过直接在配置文件 ("ipconfig.ini") 中添加数据。数据库 IP 地址，对于本机而已就是 127.0.0.1。

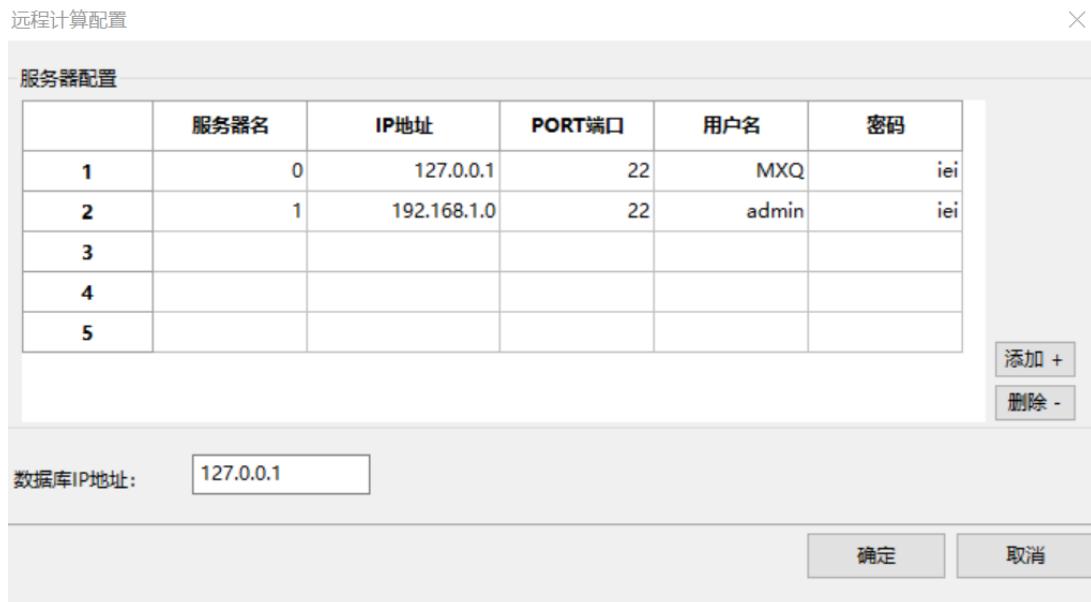


图 1-7 服务器 IP 配置操作 2

第 2 章 MIDIS 主体框架

本章主要内容

- ◊ MIDIS 的主体框架
- ◊ Pydocview 代码分析
- ◊ 服务插件的编写

2.1 小节 wxPydocview 框架

MIDIS 的 GUI 框架是在 pydocview 框架进行扩展。pydocview 是 wx 库中自带的一个 GUI 框架，通过继承的方式进行自定义。功能函数主要包括，主函数 DocApp，主界面函数 DocTabbedParentFrame，和文档管理函数 DocManager(docview.py 文件中)以及其他功能函数构成。

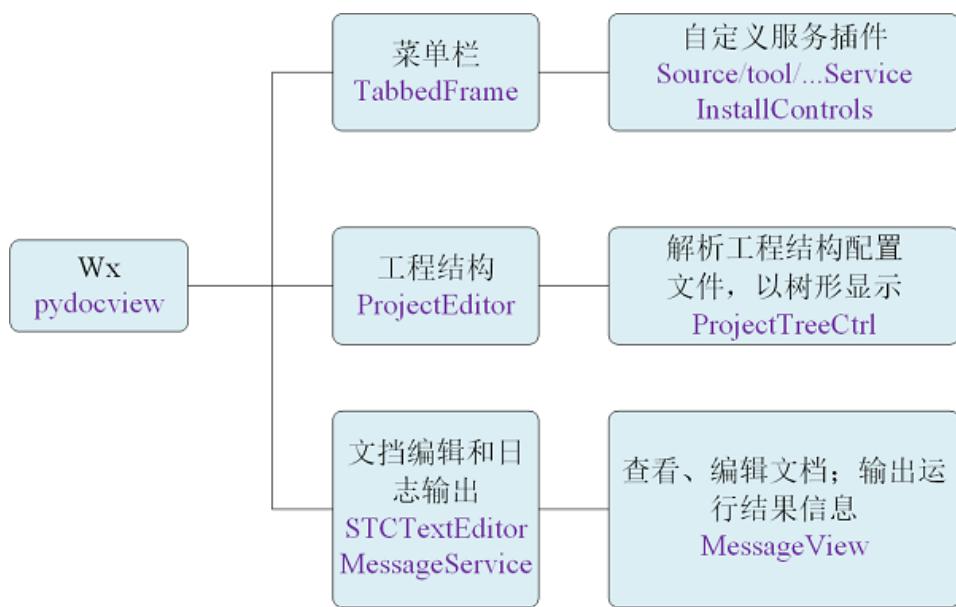


图 2-1 主体框架

如图 1-1 所示，每一个 MIDIS 中每一个区域都有相关的服务文件控制，其中菜单栏中的服务插件是我们关注的重点。其他区域的变动性不是很大，首先看一下菜单栏的相关内容。

代码清单 2-1 TabbedParentFrame.py 代码摘录

```
1 class TabbedParentFrame(pydocview.DocTabbedParentFrame):
2     def CreateDefaultMenuBar(self, sdi=False):
3         menuBar = wx.MenuBar()
4         fileMenu=wx.Menu()
5         fileMenu.Append(wx.ID_NEW,_(u"新建文件\Ctrl+N"),_(u"新建文件"))
6         fileMenu.Append(wx.ID_OPEN,_(u"打开…\Ctrl+O"),_(u"打开文档"))
7         fileMenu.Append(wx.ID_CLOSE,_(u"关闭"),_(u"关闭文件"))
8         fileMenu.Append(wx.ID_CLOSE_ALL,_(u"关闭所有"),_(u"关闭所有文件"))
9         menuBar.Append(fileMenu,(u"文件"))
10        .....
11        editMenu = wx.Menu()
12        editMenu.Append(wx.ID_UNDO, _(u"撤销\Ctrl+Z"),_("Reverses the last
13        action"))
14        wx.EVT_MENU(self, wx.ID_UNDO, self.ProcessEvent)
15        wx.EVT_UPDATE_UI(self, wx.ID_UNDO, self.ProcessUpdateUIEvent)
16        editMenu.Append(wx.ID_REDO, _(u"恢复\Ctrl+Y"),_("Reverses the last
17        undo"))
18        wx.EVT_MENU(self, wx.ID_REDO, self.ProcessEvent)
19        menuBar.Append(editMenu, _(u"编辑"))
20        menuBar.Append(viewMenu, _(u"查看"))
21        menuBar.Append(helpMenu, _(u"帮助"))
22        return menuBar
23    def CreateDefaultToolBar( self ):
24        self ._toolBar = self .CreateToolBar(wx.TB_HORIZONTAL | wx.
25        NO_BORDER | wx.TB_FLAT)
26        self ._toolBar.AddSimpleTool(wx.ID_NEW, New.GetBitmap(),_("New"),
27        _( "Creates a new document"))
28        .....
29        return self ._toolBar
30    def OnAbout(self,event):
31        dialog=AboutDialog.AboutDialog()
32        dialog.CenterOnParent()
```

```

29     if dialog .ShowModal()==wx.ID_OK:
30         dialog .Destroy()

```

首先可以看出 TabbedParentFrame 类是继承自 pydocview 中的 DocTabbedParentFrame 中的，在 pydocview 中打开这个类，可以看出类中实现了相同的函数 CreateDefaultMenuBar 和 CreateDefaultToolBar，只不过原函数中的相关字符串都是英文，这里我们将其自定义为中文。TabbedParentFrame 类在菜单栏中实现了 4 个默认的选项卡即文件，编辑，查看，帮助。在大部分软件中，这四个选项卡都会存在，而且他自定义的选项卡会通过 Service 类进行添加，后面会具体分析到。回到 Entry.py 文件，

```

1     embeddedWindows = pydocview.EMBEDDED_WINDOW_TOPLEFT pydocview.
2
3         EMBEDDED_WINDOW_BOTTOM|pydocview.
4
5             EMBEDDED_WINDOW_BOTTOMLEFT

```

设置了界面的布局分三个部分

```

1     projectservice = self . InstallService ( ProjectEditor . ProjectService (_(u"工程
2
3         ")),pydocview.EMBEDDED_WINDOW_TOPLEFT))
4
5     outlineservice = self . InstallService ( OutlineService . OutlineService ("Outline
6
7         " , pydocview.EMBEDDED_WINDOW_BOTTOMLEFT))
8
9     self . messageservice = self . InstallService (MessageService.MessageService(_(u
10
11         "信息")),pydocview.EMBEDDED_WINDOW_BOTTOM))

```

初始指定了默认选项卡的位置，工程选项卡在左上侧，Outlineservice 在左下侧（空白处，好像没有用到），日志输出在右侧。还有文档查看，这个是通过 docview 的 docManager 函数和 STCTextEditor 进行管理。在 MIDIS 中的具体效果如图 1-2 所示。对 MIDIS 普通的扩展主要是对其增加服务插件，Pydocview.py 中 class DocApp(wx.App): 是整个 GUI 的入口，在 Entry.py 函数中 class MainApp(pydocview.DocApp): 进行了重新的自定义包括设置 APP 的名称，GUI 的布局，菜单栏，状

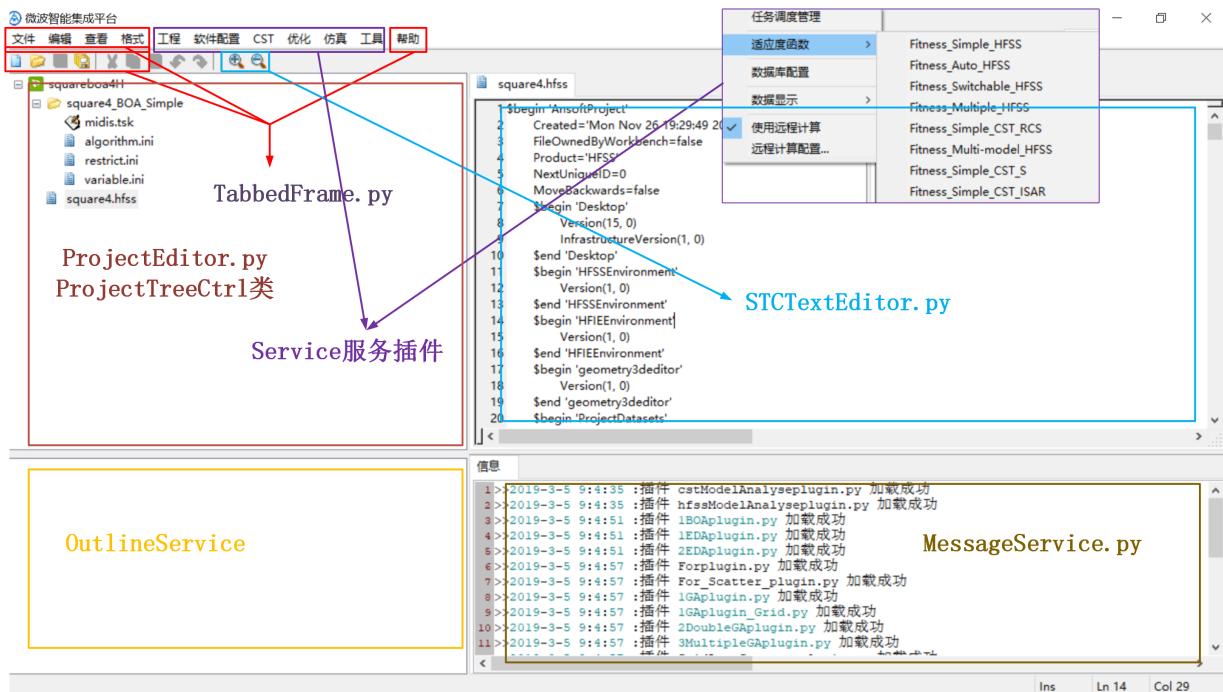


图 2-2 MIDIS 分区图

态栏，工程目录等。InstallService 函数，将各种服务添加到全局变量 _service(继承至 pydocview) 中至此各种服务插件都加载进去了，通过界面中的相关操作，调用服务插件中的各种函数即可实现相应功能。那么 MIDIS 是将我们写的选项卡添加到相应的下拉框中去的呢？我们以算法服务为例：找到：AlgorithmService.py 文件如果你需要将一个服务类添加到 GUI 中，则 InstallControls 是一个必须实现的函数，并且需要在 Entry.py 中导入该类和实例化。接下来需要弄明白两个事情，InstallControls 实现了什么功能以及如何在 Entry 将其实例化。

2.2 小节 服务插件的编写

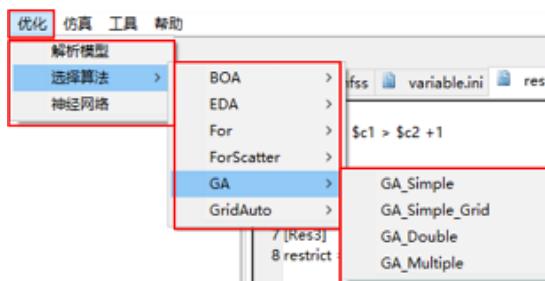


图 2-3 算法服务实例菜单

如图 1-3 所示是算法服务插件的四级菜单效果图

代码清单 2-2 算法服务中的 InstallControls 函数实现

```
1 def InstallControls ( self ,frame,menuBar = None, toolBar = None, statusBar =  
2     None, document = None):  
3     projectMenuPos = menuBar.FindMenu_(u"工程")  
4     if projectMenuPos == wx.NOT_FOUND:  
5         projectMenuPos = 0  
6     if menuBar.FindMenu_(u"优化") == wx.NOT_FOUND:  
7         optimizeMenu = wx.Menu()  
8         menuBar.Insert(projectMenuPos+1,optimizeMenu,_(u"优化"))  
9     optimizeMenu = menuBar.GetMenu(menuBar.FindMenu_(u"优化"))  
10    algSelectMenu = wx.Menu()  
11    optimizeMenu.AppendSubMenu(algSelectMenu,_(u"选择算法"))  
12    self .AlgSelectId = optimizeMenu.FindItem_(u"选择算法")  
13    wx.EVT_UPDATE_UI(frame,self.AlgSelectId,self.ProcessUpdateUIEvent)  
14    algSelectMenu = optimizeMenu.FindItemById(self. AlgSelectId) .GetSubMenu()  
15    algorithmPluginPath = os.path.join( sysutil .mainModuleDir,"Plugin",  
16                                         "algorithmplugin")  
17    i = 0  
18    for dirpath ,dirnames, filenames in os.walk(algorithmPluginPath ):  
19        i += 1  
20        if i == 1:  
21            for dirname in dirnames:  
22                if Debug:print dirname  
23                subMenu = wx.Menu()  
24                algSelectMenu.AppendSubMenu(subMenu,dirname)  
25            else :  
26                if dirpath not in sys.path:  
27                    sys.path.insert (0, dirpath )  
28                for filename in filenames:  
29                    if filename != "__init__.py" and (os.path.splitext (filename)[1]  
30                     == ".py" or  
31                         os.path.splitext (filename)  
32                         [1] == ".pyc") and "  
33                         plugin" in filename:
```

```

29         if os.path.splitext(filename)[0] not in sys.modules:
30             try:
31                 moduleName = __import__(os.path.splitext(filename)[0])
32                 module = moduleName.InstallPlugin()
33                 self._algorithmService.append(module)
34                 printMessage(u"插件%s加载成功"%filename)
35             except:
36                 printMessage(u"插件%s加载失败"%filename)
37             if dirpath in sys.path:
38                 del sys.path[0]
39         return

```

我们需要将”算法”选项插入到”优化”选项卡下的二级菜单中，首先通过 FindMenu 函数确定”优化”选项是否已经建立，如果没有建立，找到”工程”的 ID，在其后面通过 menuBar.Insert 函数创建 (3-8)。然后 AppendSubMenu 函数按服务实例化的先后顺序在优化菜单下创建”选择算法”菜单 (在 Entry.py 中模型解析服务在算法服务之前进行了实例化)，在算法插件中，我们按照不同的算法和算法实现的功能建立了多个以算法类型命名的文件夹，每一个文件夹下会有多个算法插件，文件夹名称构成了第三级菜单 (16-22)，这个菜单下的各种算法插件，插件通过”InstallPlugin”函数，其实是调用了 InstallService 函数，最终将最后的算法插件插入到最后的菜单中 (24-38)。

到底层的服务插件会需要一个响应函数，用来响应点击该菜单后的效果，因为算法插件最后的实现是在 plugin/algorithmpugin/GA/1GAPlugin.py 中，在这些文件的 InstallControl 函数下的 wx._EVTMENU (frame,self.ALGORITHM,frame.ProcessEvent) 就是用来实现响应操作，

self.ALGORITHM 表示选择的菜单选项, frame.ProcessEvent 是用来处理响应的。

在 Entry.py 服务的实例化是通过 InstallService 函数实现, 在 pydocview 中进行定义。

```
1 def InstallService ( self , service ):  
2     service . SetDocumentManager(self._docManager)  
3     self . _services . append(service )  
4     return service
```

可以看到这个函数的有一个功能就是将服务添加到全局变量 _services 中去。

```
1     self . frame=TabbedParentFrame(docManager,None,-1,self.GetName(),  
2                                     embeddedWindows=embeddedWindows,minSize=150)  
2     self . frame.Show()
```

当所有服务通过 InstallService 进行实例化之后, 将默认的菜单 embeddedWindows, 文档管理 docManager, APP 名称 self.GetName() 等组合一起显示, 通过 show() 函数就可以将所有的界面相关内容展示出来。那么我们自己自定义的服务是如何显示的呢, 这个还是得通过 embeddedWindows.py 文件, 虽然在这个文件当中, 我们只是实现了几个默认的菜单选项, 再看一下这个类的声明。

class TabbedParentFrame(pydocview.DocTabbedParentFrame): 这个类继承了 pydocview 的 DocTabbedParentFrame 类, 因为 TabbedParentFrame 没有初始化 (init) 函数, 而且在 pydocview 的 DocTabbedParentFrame 类中, 有初始化函数, 所有这个初始化函数被继承到了 TabbedParentFrame 当中, 初始化时调用了 self._InitFrame(embeddedWindows, minSize), 具体代码如下:

代码清单 2-3 DocTabbedParentFrame 类下的 _InitFrame 实现

```
1 def _InitFrame( self , embeddedWindows, minSize):  
2     for service  in wx.App().GetServices():
```

```

3     service . InstallControls ( self , menuBar = menuBar, toolBar = toolBar ,
4
5         statusBar = statusBar )
if hasattr ( service , "ShowWindow" ):
    service . ShowWindow()

```

这个函数遍历 _services 中的每一个服务，然后调用服务中的 InstallControls，从而实现了菜单栏的加载。以上了解了服务插件添加到菜单栏的流程，可以简化为图 1-4 所示流程。

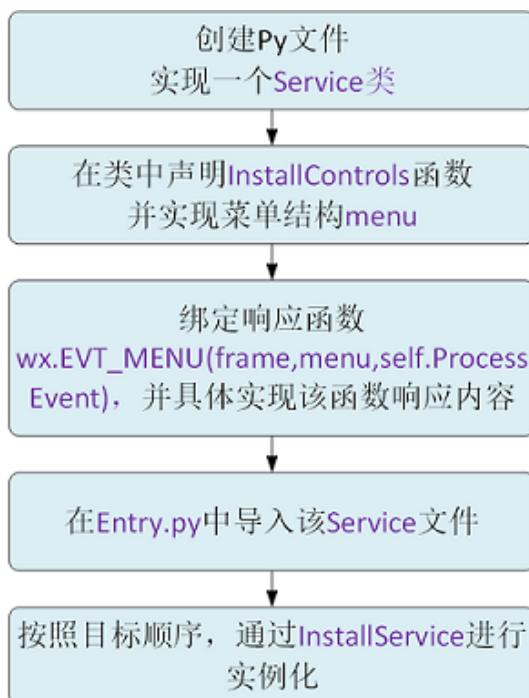


图 2-4 MIDIS 构建插件流程图

2.3 小节 本章小结

本章小结

- pydocview 是一个支持插件扩展的框架，支持文档管理，插件扩展，文档编辑等功能
- pydocview 下主要包括 pydocview.py, docview.py 等文件
- 服务插件的编写主要是通过 InstallControls 函数实现

第3章 分布式计算的实现

本章主要内容

- ◊ 计算脚本编写
- ◊ 分布式框架
- ◊ 任务调度

3.1 小节 任务计算流程

从一个初始模型到最后计算数据经历了很多过程，这其中与 MIDIS 中大部分函数都有直接或间接接触，当遇到具体的问题时都需要找到相关的函数进行细察，这里可以给出一些涉及到的文件，模型解析插件：Plugin->modelanalyseplugin、算法服务插件：Plugin->algorithmplugin、适应度插件：Plugin->fitnessplugin、任务管理：Source->tool->TaskScheduleService.py。图 3-1 展示了计算过程中主要调用函数，MIDIS 中是通过配置文件来保存一些设置数据。ParamDict.ini 保存了模型的全部变量信息、端口信息、频率信息等。algorithm.ini 中是算法信息。变量配置后产生 variable.ini 和 restrict.ini 用来保存优化的变量和变量限制条件。适应度配置文件 fitness.ini 中主要包括了需要评价的频段以及评价公式中的一些权重信息。通过 modelModify 函数产生新模型，新模型与 vbs 对应一起发送至服务器，最后服务器计算完毕返回计算结果文件至本地计算机，这个线程会一直存在一直到所有模型分发完毕。

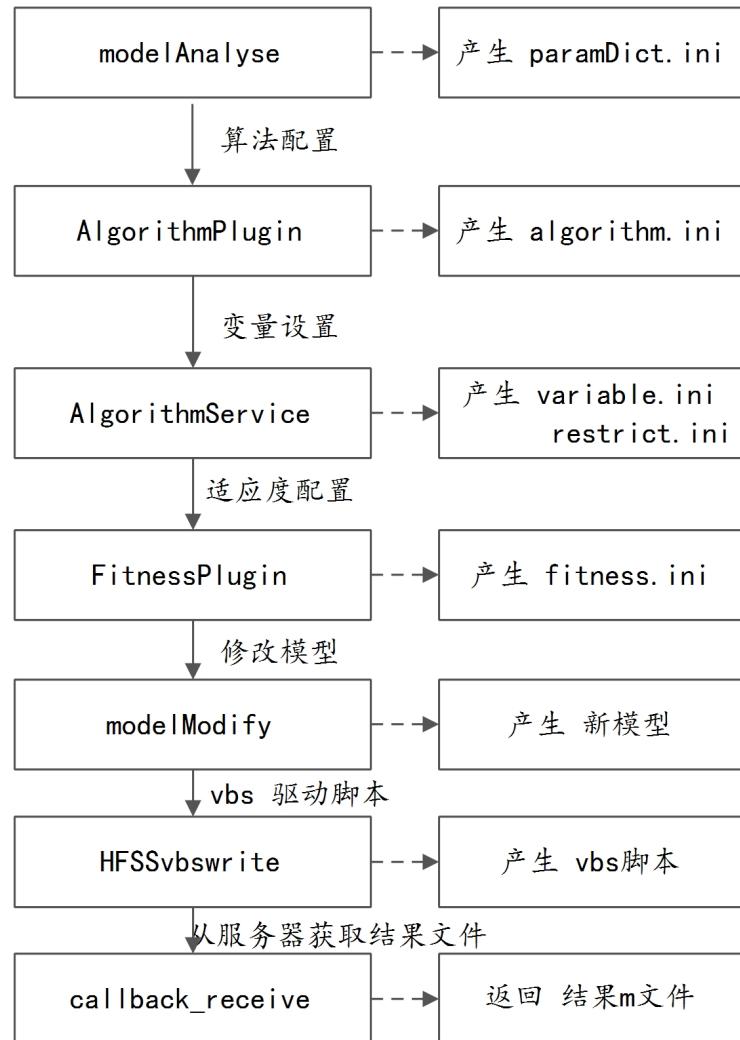


图 3-1 模型计算过程中主要流程图

3.2 小节 分布式实现

模型的具体运算是由 FitnessPlugin 和 DistributedComputeService 完成的，首先回到 FitnessPlugin 中的 `FitnessRun` 函数中去，这个函数是被算法插件直接调用，传入个体变量信息。为了符合面向对象编程的风格，我们将每一个模型的计算信息用一个计算单元类表示 (`class task_unit(object)`)，类中有很多属性。属性的设置和获取通过 `set` 和 `get` 函数实现。比如代码清单 3-1 中，类的优化变量信息 `self.varMatrix` 是通过 `set_varMatrix` 和 `get_varMatrix` 分别进行设置和获取。同理，其他属性都可以通过该方式进行处理。

代码清单 3-1 计算单元类代码片段

```
1 class task_unit( object ):
2     def __init__( self ):
3         self._vbsfilepath = """
4         self._vbsfilename = """
5         self._vbskillname = """
6         self.varMatrix=[]
7         ...
8     def set_varMatrix( self , v_matrix ):
9         self.varMatrix = v_matrix
10    def get_varMatrix( self ):
11        return self.varMatrix
12        ...
```

每一代中所有个体(对于 For 算法整个遍历的个体组成一代),通过 self._task_list.append(Task_unit) 保存到了 self._task 这个全局列表中。如何将这个变量传入到 task_classify 中进行分类,通过对个体变量信息的比较,可以知道该个体是否在数据库中存在记录、是否是同代中相同个体、是否是未计算的个体,所有形成了三种任务分类即 task_from_db、task_from_other、task_compute。前两者情况都不需要计算,从需要从数据库和其他个体那里获取数据。这里我们主要关心的是 task_compute 中的个体。

代码清单 3-2 个体任务分类实现

```
1 def task_classify( self , tasks ):
2     self.task_from_db = list()
3     self.task_from_other = list()
4     self.task_compute = list()
5     for task in tasks:
6         if task.is_finished():
7             self.task_from_db.append(task)
8             continue
9         if task.get_unit_number_point() != -1:
10            self.task_from_other.append(task)
```

```

11     continue
12     self.task_compute.append(task)

```

通过 `set_taskunit_func` 函数来设置计算任务是使用本地计算还是远程计算。如果设置为远程计算，则 `callback_send`, 和 `callback_receive` 函数必须定义，`send` 是用来发送文件并执行命令，`receive` 是用来检查服务器端的文件是否按计划的方式生成。如果生成，则要使用 `server.set_used()`

代码清单 3-3 根据用户的设置分别进行本地和远程计算

```

1 def set_taskunit_func ( self ,func ,remote):
2     if remote:
3         self.send = self.callback_send
4         self.receive = self.callback_receive
5         self.post = self.post_process
6         self.do_task_func = func
7     else :
8         self.send = None
9         self.receive = None
10        self.post = None
11        self.do_task_func = self.local_compute
12    return True

```

最后通过 `taskunit_compute` 来启动计算, 真正执行计算的是 `do_task_func` 函数，那么这个函数是如何定义的呢？现在我们需要回到 `DistributedComputeService.py` 中去。

```

1 def taskunit_compute( self ):
2     return self.do_task_func( self.send, self.receive, self.post, self.
task_compute,schedulename)

```

在 `DistributedComputeService.py` 中下面这两行代码是用来定义 `do_task_func` 函数的。

```

1 distrComputeService = wx.GetApp().GetService(DisCompute.
distributedComputeService)
2 self.fitservice.set_taskunit_func (distrComputeService.compute_start, True)

```

下面重点是要回到在 `DistributedComputeService.py` 中去,首先了解一下 `compute_start` 的实现。通过 `init_server` 确认可以使用的 IP 节点。`loopcheck` 是一个
多线程, 这里为什么要使用多线程, 如果使用单个线程, 则我们的程序只能按
照顺序依次执行函数即, 我们的程序运行最后一步只会卡死在模型的循环计算
中, 一直持续到任务结束, 但是我们希望此时还能进行其他界面上的操作, 多
线程可以保证这些操作都相互独立。

代码清单 3-4 `compute_start` 函数实现

```
1 def compute_start( self ,callback_send , callback_receive , post_process , task_list  
 ,schedulename):  
2     self . init_server (schedulename)  
3     loopcheck = threading .Thread( target = self .loop_check, args = (  
        callback_send , callback_receive , post_process , task_list ,schedulename))  
4     loopcheck. start ()  
5     loopcheck. join ()  
6     return True
```

多线程真正执行的是目标函数 `self.loop_check`, 所以整个分布式计算的核心就是这个函数。其实这个函数还是有点复杂, 可以简化成图 3-2 所示的流程图。这里建立了一个待计算任务队列 `task_index` 和 `server` 的 `unused` 和 `used` 属性。首先对任务队列进行检查, 这是跳出整个循环的标志, 如果队列为空则表示计算完毕。当有计算任务时, 对 IP 节点进行检查, 当计算节点没有被占用时, 给该节点分配任务队列中的第一个任务, 同时任务队列中的任务被弹出(减少), 并设置该节点被战役。对于被占用的节点要检查是否有计算数据产生, 如果有则释放该节点, 如果没有表示计算出错, 重新将给计算任务加入任务对列, 以此循环。

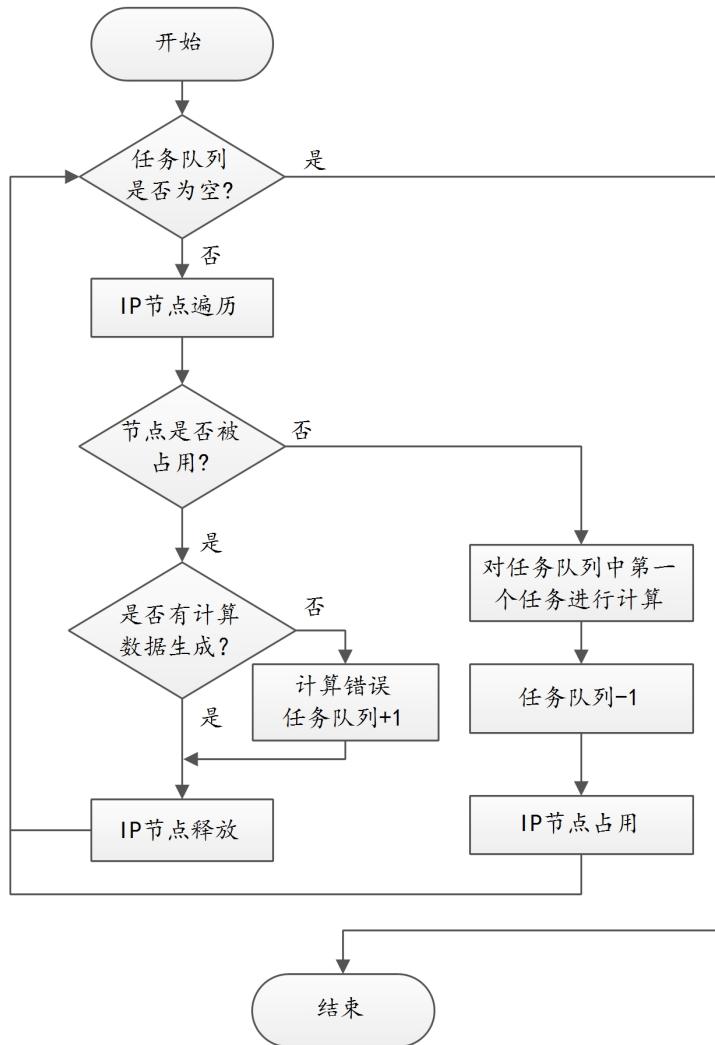


图 3-2 分布式计算简化流程

代码清单 3-5 loop_check 函数实现

```

1 def loop_check( self , callback_send ,  callback_receive ,  post_process ,  task_list ,
2 schedulename):
3     task_compute =  task_list
4     task_num = len( task_list )
5     task_index =  list ()
6     for i in range(task_num-1,-1,-1):
7         task_index.append(i)
8     global Auto_Compute
9     Auto_Compute = True
10    server_inuse = 0
11    db = DBService.DBclass()
  
```

```
11 while(Auto_Compute):
12     #time.sleep(1)
13     if not self.server_valid :
14         db.updateschedulelist(schedulename,-1,-1,-1,0,-1,-1,-1)
15         wx.CallAfter(pub.sendMessage,"update_listctrl")
16         break
17     status = db.queryspeci(schedulename)[ 'status' ]
18     if status == 0:
19         t = threading.Thread(target=self.OnTaskKill(schedulename), name
20                               ='LoopThread')
21         t.start()
22         t.join()
23         wx.CallAfter(pub.sendMessage,"update_listctrl")
24         Auto_Compute = False
25         return False
26     for server in self.server_valid :
27         time.sleep(0.05)
28         #print server.ip," this server"
29         if server.has_used():
30             if server.thread_run is not None and not server.thread_run.
31                 isAlive():
32                 if server.run_correct:
33                     server.set_predict_time()
34                     if callback_receive(server):
35                         server.set_unused()
36                         server_inuse -= 1
37                     else:
38                         server.set_unused()
39                         server_inuse -= 1
40                         task_index.append(server.task_index)
41                     if not server.has_used:
42                         server.init()
43                 else:
44                     printMessage(u"远程主机%s,%s可能出现错误"%(server.ip,server.user))
```

```

43         server.set_unused()
44         callback_receive(server)
45         server_inuse -= 1
46         task_index.append(server.task_index)
47     elif task_index:
48         index = task_index.pop()
49         task = task_compute[index]
50         server = self.select_server()
51         server.set_task(task)
52         server.task_index = index
53         if callback_send(server):
54             server.set_used()
55             server.set_start_time()
56             server_inuse += 1
57     elif not task_index and server_inuse == 0:
58         Auto_Compute = False
59         break
60     post_process()
61     for server in self.server_valid:
62         server.set_unused()
63         server.init()
64     return

```

3.3 小节 任务管理

任务管理的一个重要功能是将创建工程的过程进行持久化存储，从而达到提高任务管理效率的目的。我们知道 MIDIS 在创建工程过程中会产生很多配置文件，这些配置文件都需要我们通过点击相应的服务插件加载到内存中，所以只要我们想开启一个任务都相应这些点击操作，不然我们是无法知道这些配置文件的路径和内容。通过数据库将这些需要的配置文件路径信息和内容存储到数据库中是简化该流程的一个有效方式。

TaskScheduleService.py 中的 Addtotask 类是用来将所有的服务、配置文件的路径以及一些用户输入信息，如：任务节点个数等信息存储到数据库中，存储时调用了 writepathtodb 函数。真正计算时调用的是如下函数，这是一个多线

程启动的脚手架函数 run。

```
1 def run( self ):  
2     self . algservice . AlgorithmRun(temp_scheduledname,self.algParam, self .  
        modelParam,self. configlist , self . fitservice , ReflectService )
```

我们可以看出如果希望能够正确执行此命令，我们需要知道算法服务插件 algservice、优化参数 algParam、模型参数 modelParam、适应度服务插件 fitservice。这些信息在我们初始创建工程时都可以通过一步步的配置过程加载到内存中，即通过 wx.GetApp().GetService 和 wx.ConfigBase_Get() 找到对应的服务插件和工程配置文件路径，具体实现如下。所以我们可以将这些信息存储至数据库中，直接从数据库中读取这些信息，传入执行函数开启任务。

```
1 self . algService = wx.GetApp().GetService(AlgorithmService . algorithmService ).  
    GetSpecifiedAlgorithm ()  
2 self . fitService = wx.GetApp().GetService( FitnessService . fitnessService ).  
    GetSpecifiedFitness ()  
3 config = wx.ConfigBase_Get()  
4 projectPath = config . Read(' ProjectPath ')
```

我们知道服务插件都是具体的类，我们是无法将整个类存储在数据库中的，我们该存储那些信息并且能够从这些信息中恢复出我们需要的服务插件，我们的所有算法信息、制约条件、数据库配置、适应度都在工程文件的目录中，我们只需要存储工程目录的路径即可找到这些信息，找到这些相信之后通过 getconalgParam() 函数得到 algParam,modelParam,algName,fitName,Restrict。依次对算法插件的名称进行遍历，发现与 algName 相同的则加载该类。同理对算法插件进行遍历，找到与 fitName 相同的适应度插件。

代码清单 3-6 从数据库中解析服务插件

```
1 result = db.queryspeci(temp_schedulename)
2 self.configlist = result[ ' configlist ' ]
3 self.task_path= self.configlist [1]
4 params = getconalgParam( self.task_path )
5 self.algParam = params[0]
6 self.modelParam = params[1]
7 algName = params[2]
8 fitName = params[3]
9 Restrict = params[4]
10 algSer = wx.GetApp().GetService(AlgorithmService . algorithmService )
11 for modelService in algSer._algorithmService :
12     if modelService._algorithmName == algName:
13         self.algservice = modelService
14     if hasattr( self.algservice , ' SetRestrict '):
15         self.algservice . SetRestrict ( Restrict )
16 fitSer = wx.GetApp().GetService( FitnessService . fitnessService )
17 for modelService in fitSer._fitnessService :
18     if modelService._fitnessName == fitName: #如： variable . ini 中 algName = ,
19         self.fitservice = modelService
20 distrComputeService = wx.GetApp().GetService(DisCompute.
21                                         distributedComputeService)
21 self.fitservice . set_taskunit_func (distrComputeService . compute_start ,True)
```

任务管理的另外一个功能是对任务进行监控，即了解任务的进度、任务的运行情况、任务的 Ip 使用情况等。我们知道这些信息都是在任务进行过程中不断更新的，比如完成进度是在每一个个体计算完成后进行更新。这些信息是如果传入到任务管理的显示界面中呢，这里我们用到了一个重要的库 wx.lib.pubsub 它是专门用来解决 wx 各种窗口之间的多线程问题，也就是说使用它可以将一个窗口的数据传递到另一个窗口并且立刻刷新。首先在窗口函数中通过 pub.subscribe() 函数注册命令，第一个参数为响应函数，第二个参数为其代号，当接收到这个代号时就会调用响应函数。然后通过 wx.CallAfter() 函数进行发送命令，第一个

参数为固定值表示在 wx 内发送信息，第二个参数是已经被注册了的代号。在 FitnessPlugin 和 DistributedComputeService 中都有 wx.CallAfter() 函数进行运行状态和完成进度的更新。

代码清单 3-7 使用 pub 来注册窗口命令

```
1 pub.subscribe( self . start_task ,” starttask ”)
2 pub.subscribe( self . update_list , ” update_listctrl ”)
3
4 wx.CallAfter(pub.sendMessage, ” starttask ”)
5 wx.CallAfter(pub.sendMessage, ” starttask ”)
```

第4章 数据库实现

本章主要内容

- ◊ 数据库工具使用
- ◊ 数据库接口实现
- ◊ 数据分析插件实现

4.1 小节 数据库接口类与数据库工具使用

数据库的相关操作在 Source—>tool—>DBclassService.py 文件中。该文件的类主要实现了数据库的 C(创建)、R(查询)、U(更新)、D(删除) 等基本操作。Python 是通过 pymongo 库与 mongodb 交互，通过该库可以连接数据库和其他数据库操作。代码清单 4-1 实现了连接本地数据库 database 的操作。这里需要注意的是在本地计算机和机群中 Ip 不同，其中 Source—>tool 中 DBclassService.py、DBconfigService.py、DistributedComputerService.py、TaskScheduleService.py 几个文件中都有关于 Ip 的绑定，所以在本地和机群中使用时请分别进行设置。即在本地开发时使用本地 ip:127.0.0.1、在机群中部署时使用局域网 ip，如 84 号机的 Ip 为 192.168.1.84。

代码清单 4-1 Mongodb 数据库的连接

```
1 from pymongo import MongoClient  
2 connect=MongoClient('127.0.0.1')  
3 databse = connect['databse']
```

在实现数据存储之前，有必要了解一下需要存储哪些键值。一个模型的全部变量和优化变量都是需要的，频率、仿真数据都是列表。仿真数据可以根据不同的情况，设置每一个元素的值，比如是反射率、或幅值和相位两个值。其他信息，比如模型位置、模型存储位置、时间等可作为辅助信息。

initialparam 函数用来初始化有些固定的变量值，比如变量的名称，模型文

件的路径等信息，这些信息在创建任务的时候就已经固定下来，不会改变。

`insertfile` 是提供给外部的数据存储接口，同时它会调用 `insertData2Table` 函数。主要看一下 `insertData2Table` 函数的实现。

代码清单 4-2 数据存储函数实现

```
1
2 def insertData2Table ( self ,dbname,user, pwd, targetfile ,Collectname, Fre , Tags,
3                         Topos,Thickness ,\n
4                                 var_opti_name,temVarM,VarMatrix,Reflect,category):
5
6     """ insert one data to collection """
7
8
9     self . collectname = Collectname
10    db = self . connect[dbname]
11    #db. authenticate ( user ,pwd)
12    self . data = db[ self . collectname]
13    timeinfo = time. strftime ('%Y-%m-%d,%H:%M:%S',time.localtime())
14    try :
15        datum = {
16            'name' : Collectname,
17            'topology' : Topos,
18            'tags' : Tags,
19            'var_opti_name':var_opti_name,
20            'var_opti' : temVarM,
21            'vars' : VarMatrix ,
22            'thickness' : Thickness,
23            'performance': '' ,
24            "f":Fre,
25            "value": Reflect ,
26            "category" : category ,
27            'location' : targetfile ,
28            'time' : timeinfo
29        }
30        self . data . insert_one (datum)
31    except :
```

29

```
print u'数据存储失败, 请检查数据格式'
```

这里主要是将需要存储的数据以字典的形式保存在变量 `datum` 中, 然后通过 `pymongodb` 的 `insert_one` 函数将数据存储在数据库中。

数据库的查询是根据优化变量, 优化变量值是区分不同仿真数据的唯一标准, 因为优化变量值是以列表的形式存储, 导致优化变量在存储时顺序不同会产生不同的查询结果, 这里为了统一标准, 在选择算法 —> 设置变量范围时需要按照 HFSS 模型中的变量顺序依次添加, 不可错乱顺序。数据查询是通过 `find_one` 函数实现。返回值为整个数据的键值信息。

代码清单 4-3 数据库查询函数实现

```
1 def queryMatrix( self ,dbname,user, pwd,Collectname,value):
2     try :
3         self .collectname = Collectname
4         db = self .connect[dbname]
5         self .data = db[ self .collectname]
6         self .query_result = self .data .find_one({ 'var_opti ':value})
7     except :
8         print u'查询变量失败'
9     return
10    return self .query_result
```

`mongodb` 自带的工具可以很方便地将各种格式的数据文件进行导入、以及导出各种数据形式的文件和备份操作。使用这些工具可以编写一些脚本命令, 进行数据处理。

首先是将数据库中的数据导出为数据文件, 当我们将 `Mongodb` 添加到环境变量之后, 里面的工具就可以直接使用。

代码清单 4-4 数据导出

```
1 mongoexport -d dbname -c collectionname -o file --type json/csv -f field
```

参数说明：

-d：数据库名

-c：collection 名

-o：输出的文件名

-type：输出的格式，默认为 json

-f：输出的字段，如果-type 为 csv，则需要加上-f”字段名”

代码清单 4-5 数据导入

```
1 mongoimport -d dbname -c collectionname --file filename --headerline --type  
json/csv -f field
```

参数说明：

-d：数据库名

-c：collection 名

-type：导入的格式默认 json

-f：导入的字段名

-headerline：如果导入的格式是 csv，则可以使用第一行的标题作为导入的
字段

-file：要导入的文件

代码清单 4-6 数据备份

```
1 mongodump -h dbhost -d dbname -o dbdirectory
```

参数说明：

-h：MongoDB 所在服务器地址，例如：127.0.0.1，当然也可以指定端口号：
127.0.0.1:27017

-d：需要备份的数据库实例，例如：test

-o: 备份的数据存放位置, 例如: /home/mongodump/, 当然该目录需要提前建立, 这个目录里面存放该数据库实例的备份数据。

代码清单 4-7 数据恢复

```
1 mongorestore -h dbhost -d dbname --dir dbdirectory
```

参数说明:

-h: MongoDB 所在服务器地址

-d: 需要恢复的数据库实例, 例如: test, 当然这个名称也可以和备份时候的不一样, 比如 test2

--dir: 备份数据所在位置, 例如: /home/mongodump/itcast/

--drop: 恢复的时候, 先删除当前数据, 然后恢复备份的数据。就是说, 恢复后, 备份后添加修改的数据都会被删除, 慎用!

4.2 小节 数据库与仿真数据交互

使用数据库的目的是对仿真数据进行查询和存储,两者的交互操作在 Plugin—>fitnessplugin—>适应度插件中。图 4-1 展示了两者交互关系。当算法传入变量参数之后,首先对该变量在数据库中进行记录查询,查询时间一般可以忽略不计,在使用 For 算法进行变量空间遍历时,若计算过程出现中断,则可以通过数据库查询快速恢复到之前计算状态即实现断点续算功能。同时在使用遗传算法时,不同代之间若存在相同的个体也可以直接查询返回,大大提高了计算效率。接下来具体看一下代码实现。

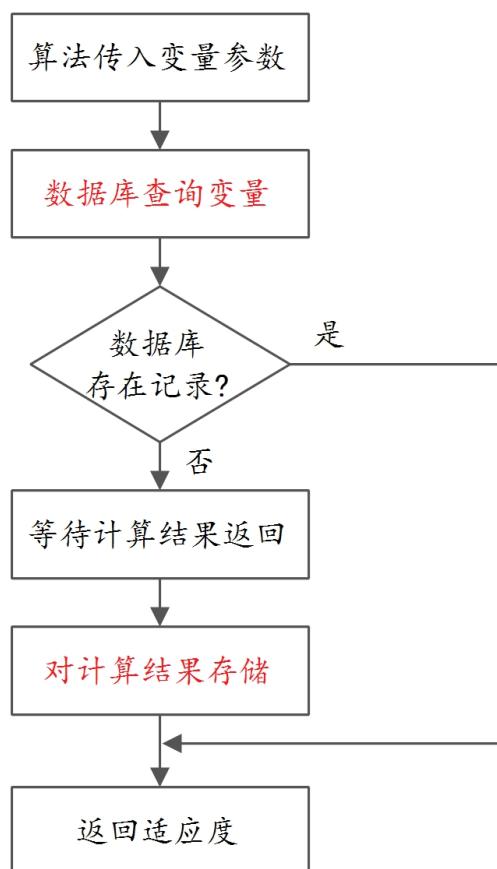


图 4-1 仿真数据与数据库交互

代码清单 4-8 FitnessRun 函数数据库查询部分代码

```
1 def FitnessRun( self ,temp_schedulename,objectnumber,varxs,algParam,modeleparam,  
    param_configlist , ReflectService ):  
2     for i in range(objectnumber[1]):
```

```
3         first_occur = varxs.index(varxs[i])
4         Task_unit = task_unit()
5         taskinfo_objectnumber=[objectnumber[0],(i+1)]
6         countinfo=1
7         Task_unit.set_varMatrix(varxs[i])
8         if first_occur == i:
9             Task_unit.set_unit_number(i)
10            query_result = self.db.queryMatrix(dbname,user, pwd, dbHFSSfilename,
11                                         varxs[i])
12            Task_unit.set_hfssnamedb(dbHFSSfilename)
13            Task_unit.set_dbname(dbname)
14            Task_unit.set_user(user)
15            Task_unit.set_pwd(pwd)
16            found = False
17            if not query_result:
18                found = False
19            else:
20                found = True
21            if found:
22                t2=query_result['f']
23                self.fre=[float(x) for x in t2]
24                countinfo=0
25                task_database_info =[self.databasefile, self.tablename, self.
26                                     projectname, self.taskname, taskinfo_objectnumber, countinfo]
27                Task_unit.set_database_info(task_database_info)
28                R = query_result['value']
29                Fitness = self.countfitness(self.fre, R)
30                Task_unit.set_finished()
```

可以看出当在数据库中查询到记录后，主要提取了模型的频率和 S 参数值，然后调用计算适应度函数。数据的存储是在 `callback_receive` 函数中，当从远程服务器获取到计算结果文件后，对结果文件进行解析存储。

代码清单 4-9 callback_receive 函数从服务器获取结果文件、进行解析存储

```
1 def callback_receive ( self , server ):  
2     if not server .connect ():  
3         return False  
4     task = server .get_task ()  
5     task_database=task .get_database_info ()  
6     server_homepath = server .get_homepath()  
7     server_file_path = os.path .join (".\\" , self .SERVER_DIR, self.projectname,  
8         self .taskname, u' generation '+u'%s'%task_database[4][0]+u'n%s'%  
9         task_database[4][1])  
10    remote_file = os.path .join ( server_file_path ,os.path.basename(task.  
11        get_output_m ()))  
12    print remote_file , ' remote_file , receive '  
13    local_file = task .get_output_m ()  
14    count = 0  
15    if server .get( remote_file , local_file ):  
16        filepath = os.path .join (server_homepath, self .SERVER_DIR, self.  
17            projectname, self .taskname, u' generation '+u'%s'%task_database[4][0]+  
18            u'n%s'%task_database[4][1])  
19        cmd = "powershell .cd .~;Remove-Item %s -recurse;quit "%filepath  
20        server .exec_command(cmd)  
21        server .set_unused ()  
22        fre ,R=get_data_R( local_file )  
23        if Debug:print fre ,R  
24        fitness_value = self .countfitness ( fre , R)  
25        self .fre=fre  
26        self .R=R  
27        task .set_private_data ( fitness_value ,R)  
28        task_database=task .get_database_info ()  
29        task_varmatrix=task .get_varMatrix ()  
30        objectnumber=u'第%s代'%task_database[4][0]+u'第%s个个体'%  
31            task_database[4][1]  
32        hfsspaths =task .get_newfileabs ()  
33        dbname = task .get_dbname()
```

```

28     user = task.get_user()
29     pwd = task.get_pwd()
30     task.get_newfileabs()
31     dbhfss = task.get_hfssnamedb()
32     var_opti_name = task.get_var_opti_name()
33     self.DATABase.insertfile(dbname, self.category, user, pwd, hfsspaths,
34                               var_opti_name, dbhfss, self.fre, task.varmatrix, R)
34     printMessage(u'%s' % objectnumber + u'计算完成')

```

4.3 小节 基于数据库的数据分析插件

在数据库的基础上，可以很方便地设计数据分析插件，对数据进行展示，提取。在设计数据分析插件的界面时，有几个功能要考虑到，数据库/集合的选择、变量解析与选择、变量之间的关系操作、适应度配置、数据导出、数据展示。如图 4-2 所示是 MIDIS 中解空间分析的界面。



图 4-2 解空间分析配置图

如图 4-3 所示，是数据分析插件的开发流程图，首先是要设计一个操作界面。通过数据库的操作获取到需要的数据，然后设计数据处理算法对数据进行处理，最后对处理的数据进行绘图展示和数据文件保存。

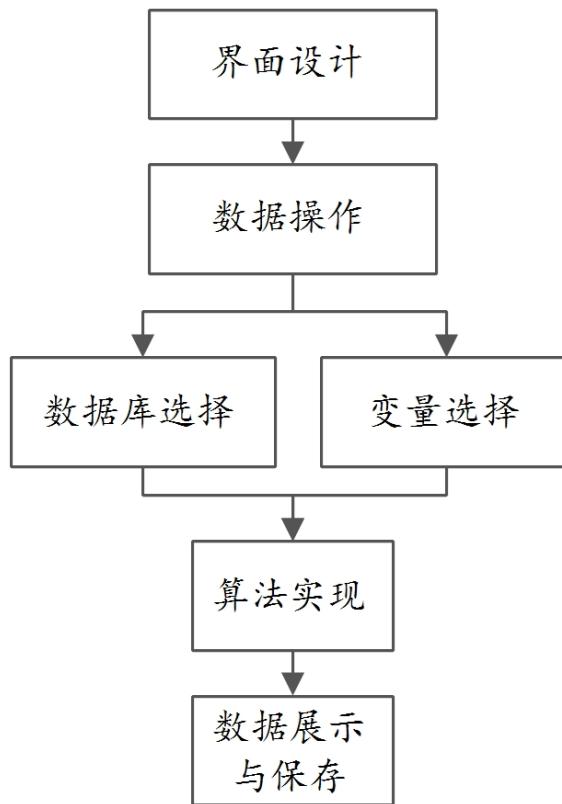


图 4-3 数据分析插件开发流程

MIDIS 中的数据分析插件在 `Plugin->dataprocess` 中，这里我们以 `DataProcessplugin.py` 解空间分析插件为例看一下其中降维算法的实现。首先 `templist` 列表存储了选择的集合的所有数据 (2),所有数据的频率都相同,可以用 `self.Frequency` 表示 (2)，变量名称也是相同的用 `varnamelist` 表示 (5)。关键点在于如何将多维的变量的组合与适应度的关系降维至二维，其实这个很简单，我们选择我们需要的两个变量其他变量的排列组合对应多个适应度 (6-12)，我们只选取其中适应度最大的一个作为这两个变量的值,这里我们使用一个字典进行存储,当遇到相同的键值时通过比较适应度大小来更新 (14-16)。

代码清单 4-10 解空间降维算法实现

```

1 def DataFromDatabase(self):
2     templist = list ( self . get_data [ self . gridview . GetCellValue (0,0) ]. find () )
3     self . Frequency = templist [0][ 'f' ]
4     DataDict = dict ()
5     varnamelist = templist [0][ 'var_opti_name' ]
6     for RefInf in templist :
7         x_index = varnamelist . index( self . gridx . GetCellValue (0,0) )
8         x_d = str (( RefInf[ 'var_opti' ]) [x_index])
9         y_index = varnamelist . index( self . gridy . GetCellValue (0,0) )
10        y_d = str (( RefInf[ 'var_opti' ]) [y_index])
11        KeyStr = x_d + ',' +y_d
12        Reflect = RefInf[ 'value' ]
13        Fit = self . countfitness ( self . Frequency, Reflect )
14        if KeyStr in DataDict.keys():
15            if Fit > DataDict[KeyStr]:
16                DataDict[KeyStr] = Fit
17            else :
18                DataDict[KeyStr] = Fit
19        for xy in DataDict.keys():
20            try :
21                self . Xdata.append( float (xy. split (',') [0]) )
22                self . Ydata.append( float (xy. split (',') [1]) )
23                self . Fitness . append(DataDict[xy])
24            except:
25                pass

```

最后将得到的数据进行绘图展示，这里因为我们得到的数据是离散的，在绘制二维解空间图时使用插值算法(12)，最后使用了 Matplotlib 中 contourf 绘制等高线图(15)。

代码清单 4-11 二维解空间绘图

```

1 def DrawData(self):
2     if self . VarCheck():

```

```
3     self.Xdata = list()
4     self.Ydata = list()
5     self.Fitness = list()
6     self.DataFromDatabase()
7     x = np.array(self.Xdata)
8     y = np.array(self.Ydata)
9     z = np.array(self.Fitness)
10    xi = np.linspace(min(x), max(x), 1000)
11    yi = np.linspace(min(y), max(y), 1000)
12    zi = griddata(x,y,z,xi,yi,interp=u'linear')
13    extent = (min(x),max(x),min(y),max(y))
14    plt.figure(figsize=(10,8))
15    plt.contourf(xi,yi,zi,100,linewidth=10.0,extent=extent,cmap='jet')
16    plt.colorbar().ax.tick_params(labelsize=24)
17    plt.tick_params(labelsize=30)
18    plt.xlabel(self.gridx.GetCellValue(0,0), fontsize=30)
19    plt.ylabel(self.gridy.GetCellValue(0,0), fontsize=30)
20    plt.tight_layout()
21    plt.show()
22 else:
23     print '画图失败'
```

第 5 章 MIDIS 用户操作流程

本章主要内容

- ◊ MIDIS 操作流程
- ◊ MIDIS 在使用过程中常见问题
- ◊ MIDIS 打包操作

5.1 小节 MIDIS 具体操作流程

首先启动 MIDIS，用户登录。用户名分别为：cst,hfss,root，密码为 iei813。



图 5-1 MIDIS 用户登录界面

MIDIS 的相关操作都需要依赖数据库，所以这里需要启动数据库范围，如图 5-2 所示工具 – 启动数据库。

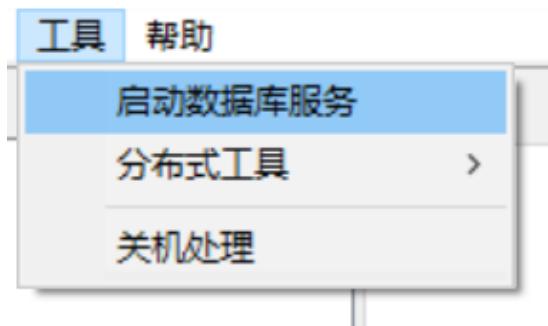


图 5-2 启动数据库

然后进入主界面，新建工程，这里需要选择工程的位置和设置工程的名字。如果需要打开已经创建的工程，则选择菜单栏：工程 → 打开工程。然后选择.project 文件。具体操作如图 5-3 所示。

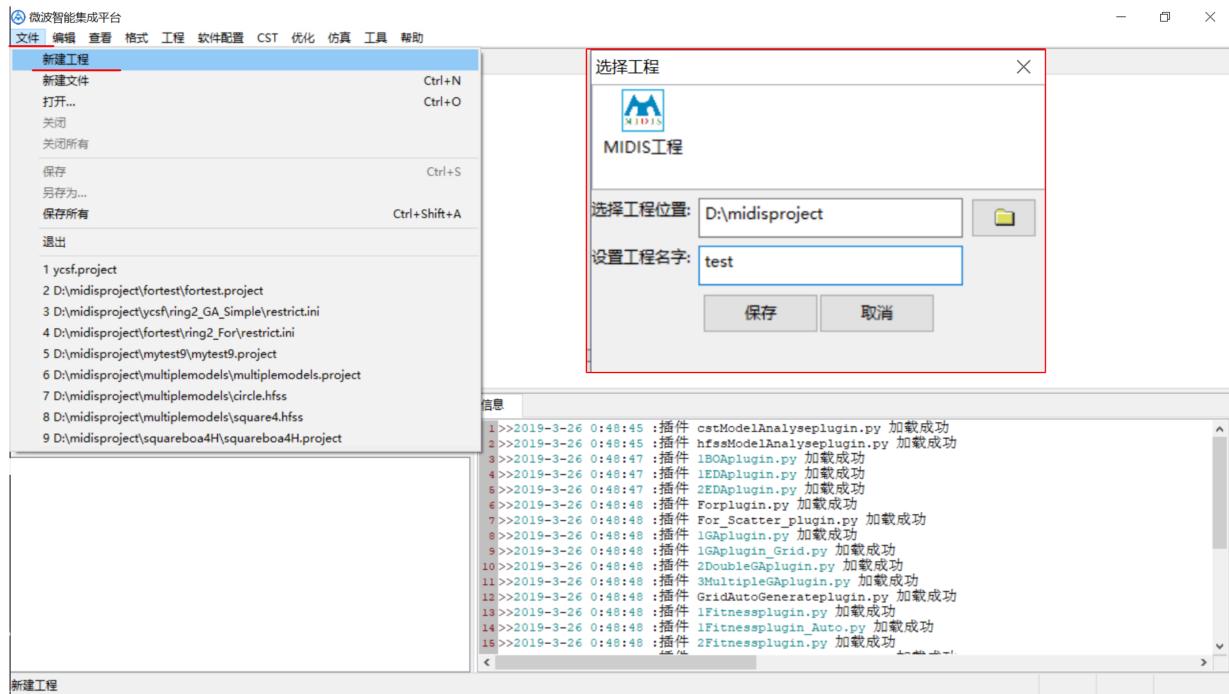


图 5-3 创建工程和打开工程

在新建的工程下添加模型文件到工程中，具体操作如图 5-4 所示。这里可以分别选择 HFSS 文件和 CST 文件。这里需要注意的是如果是添加 CST 文件，则需要将该 CST 文件通过 CST 软件打开，这时会产生一个 CST 工程目录文件，工程目录中的文件才是我们需要解析的。

添加的模型需要进行解析操作，最新的版本在导入模型后可以自动解析，如果不能自动解析，可以通过图 5-5 的右键操作进行手动解析。解析成功后，信息窗口会有解析成功提示。

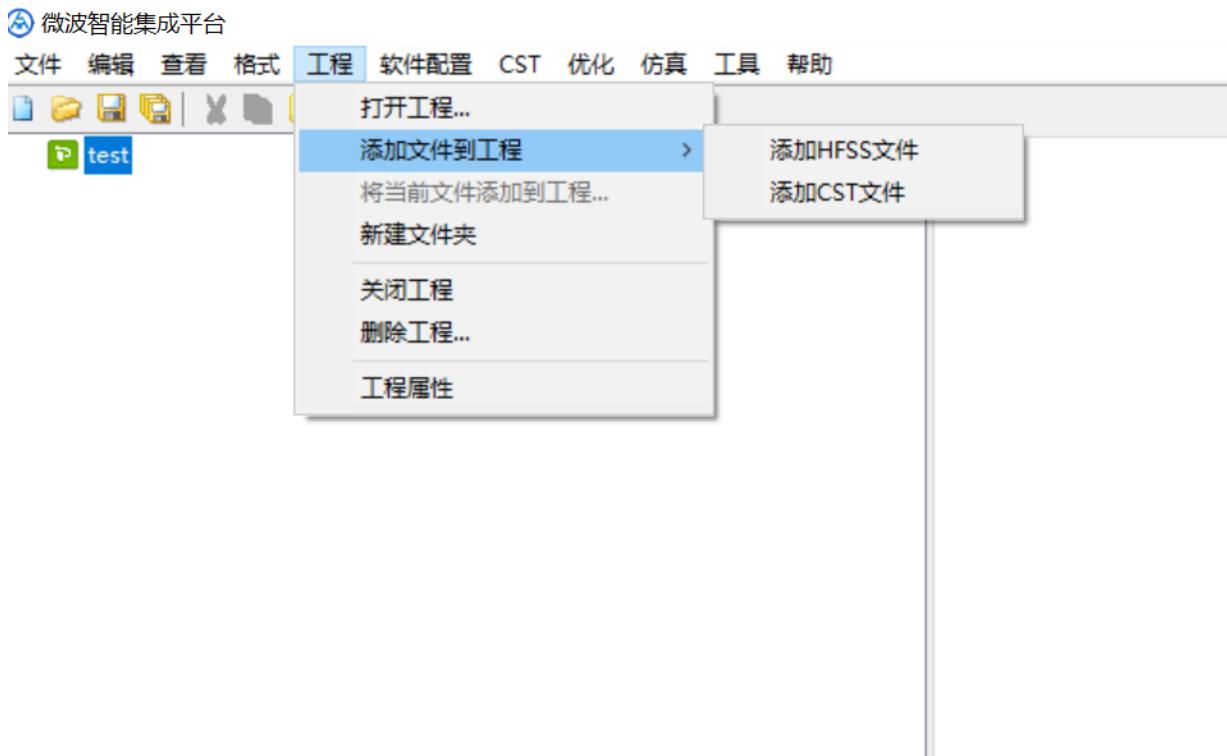


图 5-4 添加模型文件

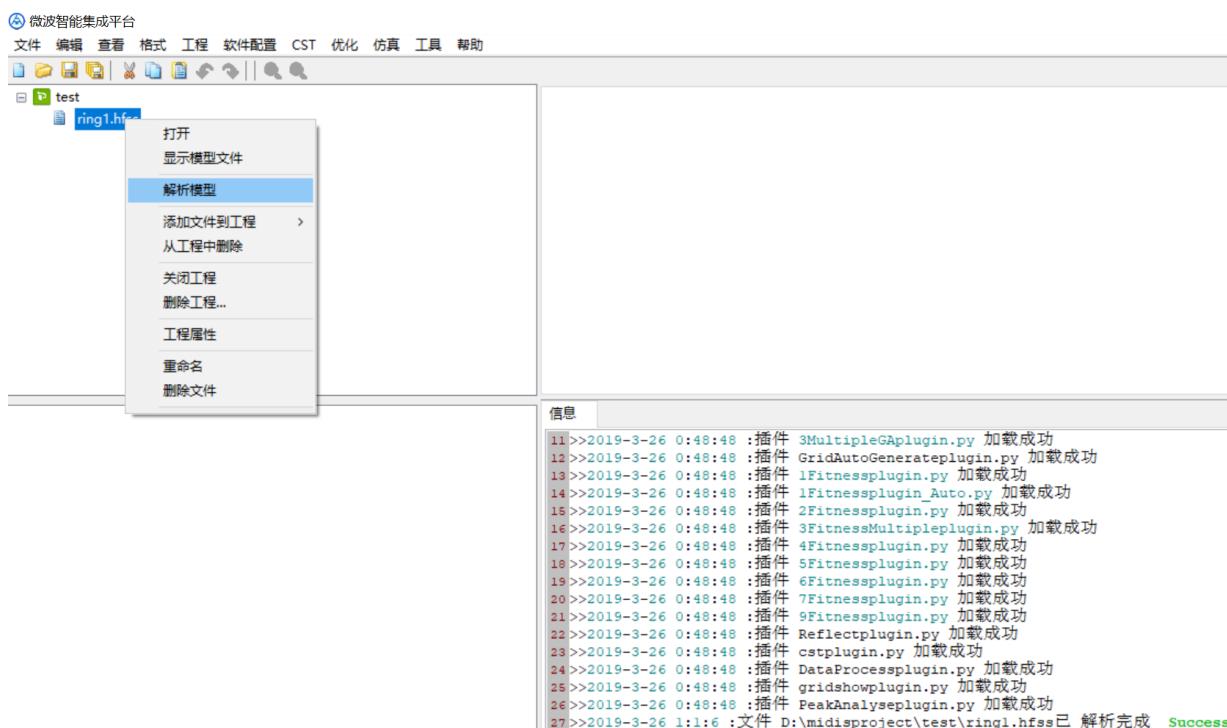


图 5-5 解析模型

接下来需要对解析后的模型进行算法配置，这里需要注意的是只要解析后的模型才能进行算法配置，此时解析后的模型需要被选中，如果发现无法选择算法，需要检查模型是否解析，模型是否选中。然后进行图 5-6 所示的操作，这里常用的两个算法有 For 算法，可以对变量空间进行遍历。遗传算法用来进行全局优化。

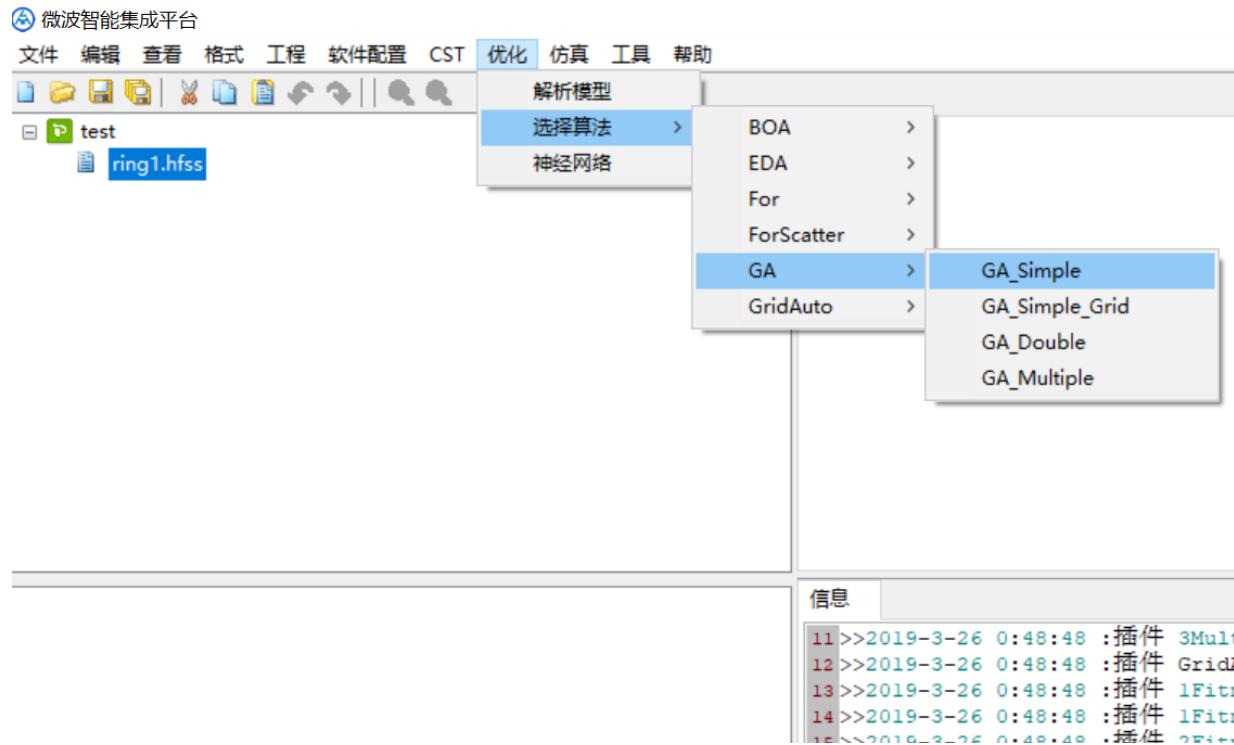


图 5-6 选择算法

以遗传算法为例，算法配置面板如图 5-7 所示，这里可以按照需求修改每代种群的个数和遗传代数，其他参数按照默认值即可。



图 5-7 遗传算法配置

然后进行变量范围设置，点击 + 添加模型中的变量，编码长度控制了变量优化时的精度，具体的公式为： $\frac{max-min}{2^n-1}$ ，其中 n 为编码长度，max, min 分别为变量最大值和最小值。一般情况下，需要勾选参数制约来设置参数之间的大小关系。当添加多个变量时，需要按照 HFSS 中的模型变量顺序依次进行添加，切不可错乱顺序。

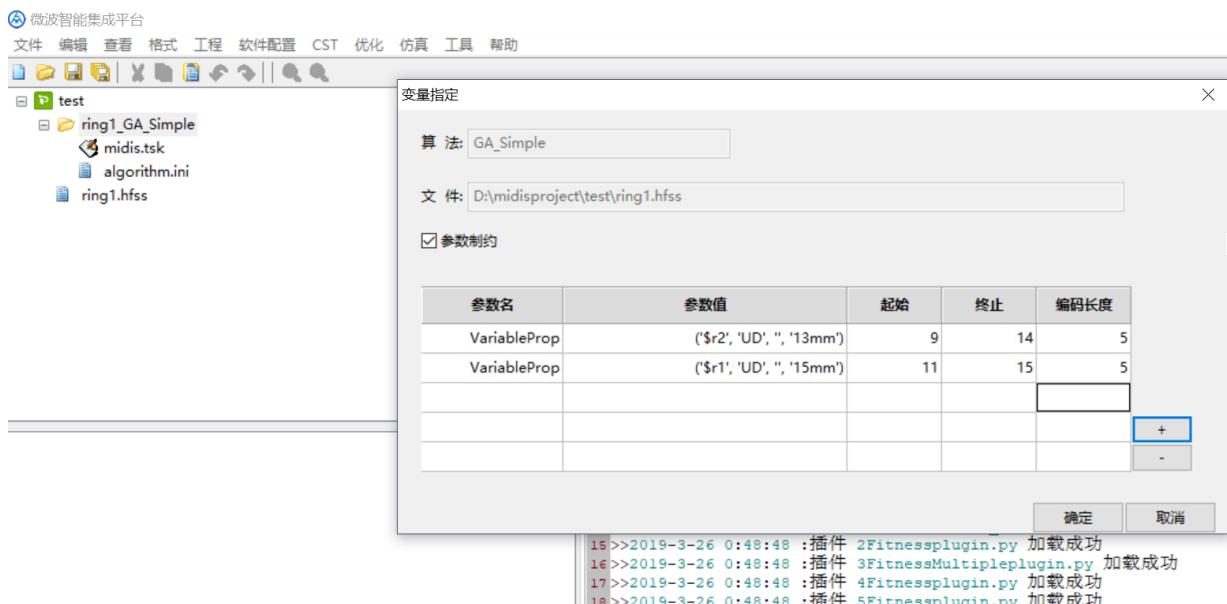


图 5-8 变量设置

参数制约需要在 restrict.ini 中进行编写。第一个制约条件为 [Res1] restrict = ...。第二个为 [Res2] restrict = ...。变量用美元符 \$ 表示，不能使用各种括号。示意图如图 5-9 所示。

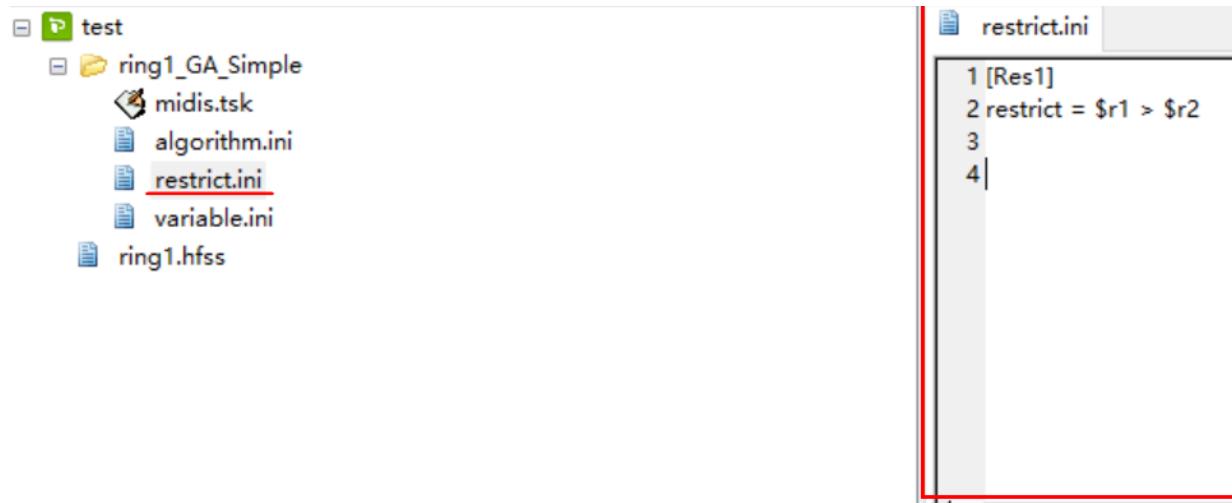


图 5-9 参数制约

数据库的配置如图 5-10 和图 5-11 所示。这里面有五个默认的数据库分别为: HFSS DataBase, Project, CST, Research, Graduate 根据需求选择数据库。用户名和密码, 这里已经有了默认值, 不需要改写。显示集合中, 会使用模型的名称作为数据库中集合的名称, 点击确认集合按钮后, 如果数据库中存在该集合, 则会提示, 你可以修改集合名称创建一个新集合, 也可以继续使用这个集合名称。相关变量配置可以设置模型的一些标签和厚度等信息。

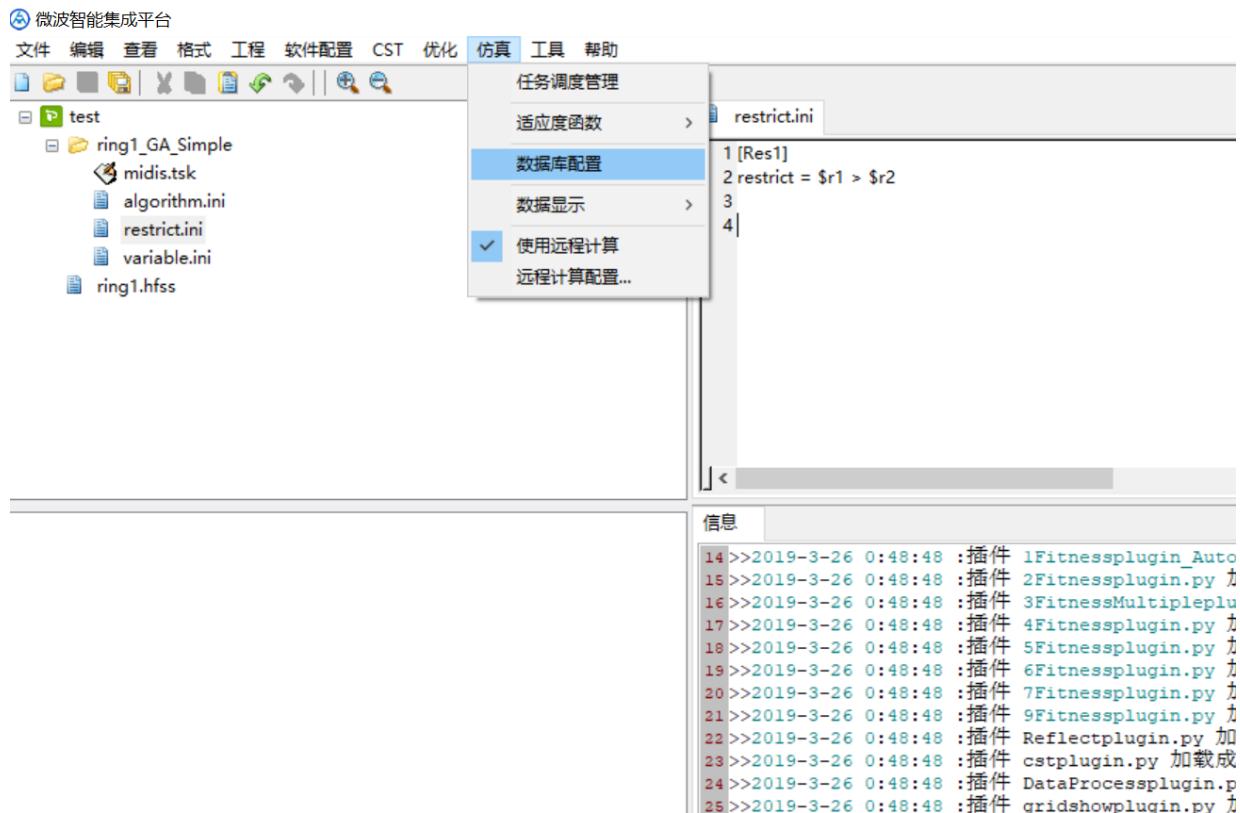


图 5-10 选择数据库配置



图 5-11 数据库配置

适应度的配置过程如图 5-12、5-13、5-14 所示。这里需要根据实际的设计需求选择不同的适应度插件，对于单个 HFSS 模型，可以选择第一项 Fitness_Simple_HFSS 插件，如图 5-13 所示。这里需要注意，起始频率和终止频率必须在 HFSS 模型中设置的扫频范围内。。如果需要对多个模型分别设置其适应度，可以选择最后一个插件 Fitness_Multiple_Phase 这里可以选择需要计算的 S 参数类型，阈值比较，相位范围等。

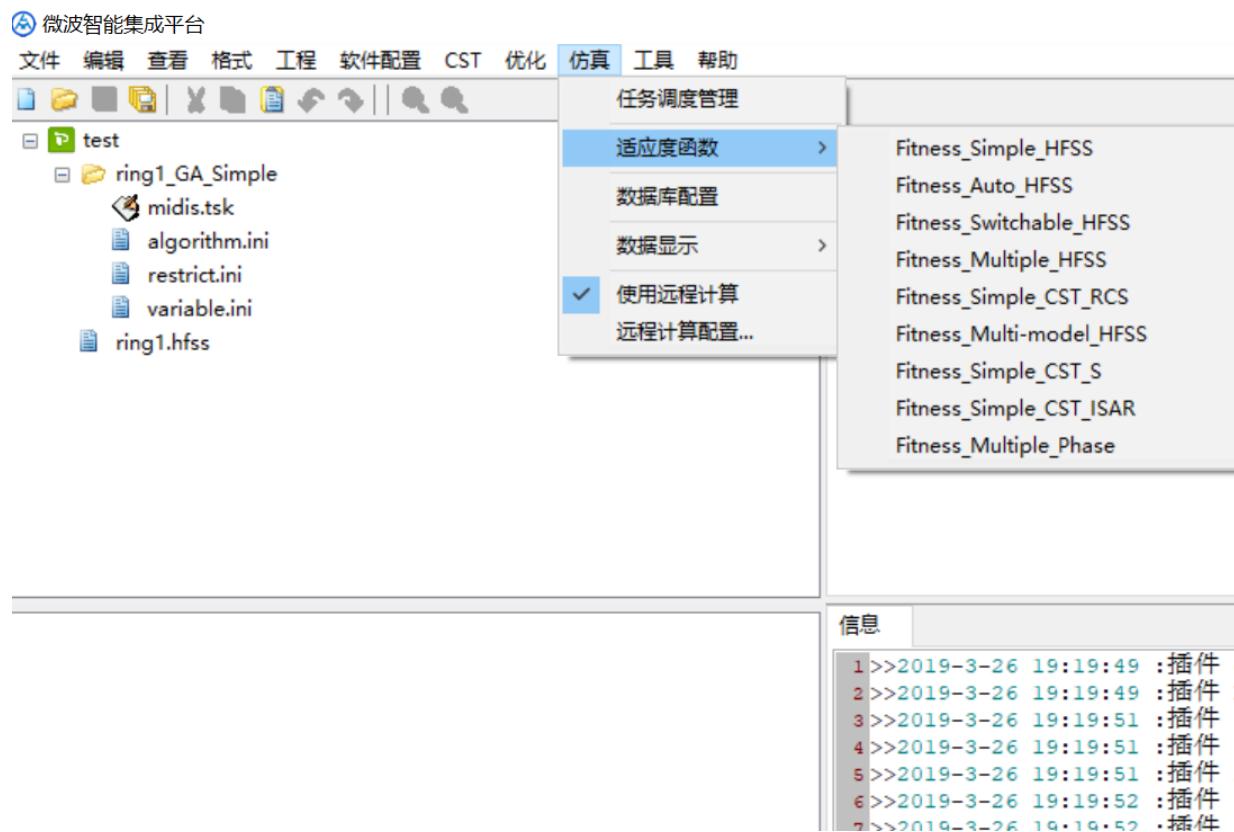


图 5-12 适应度配置 1



图 5-13 适应度配置 2

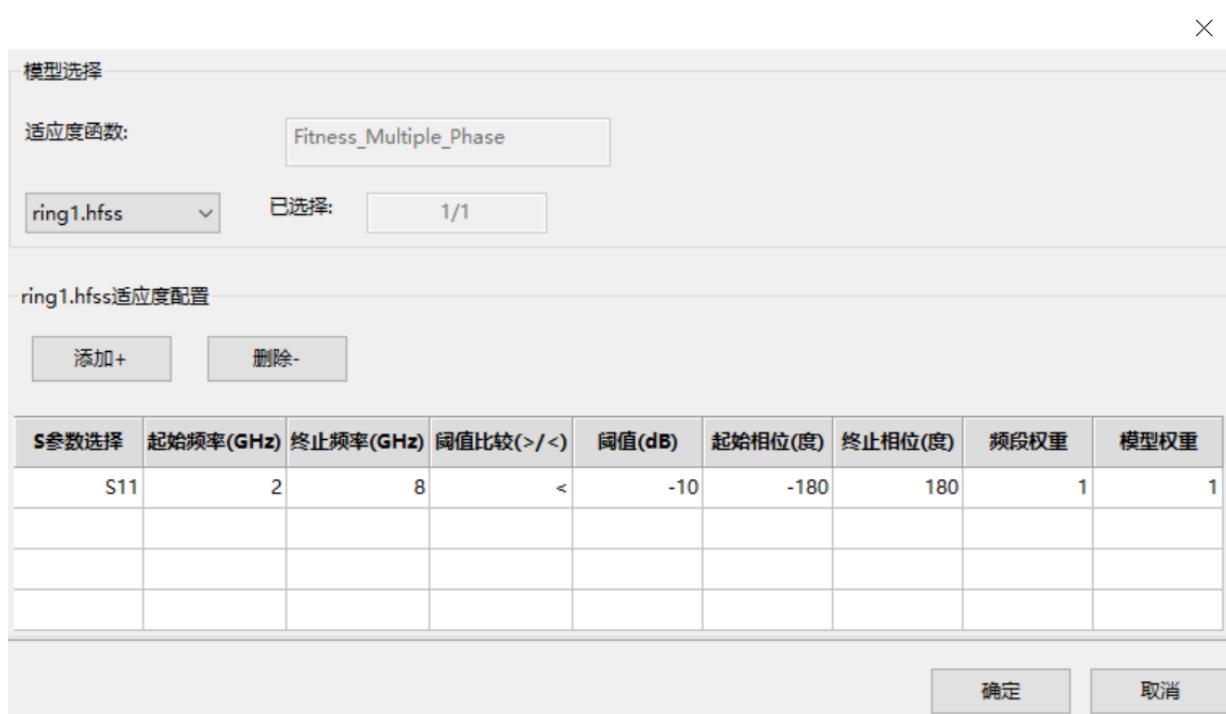


图 5-14 适应度配置 3

所有设置完成后，需要将工程进行保存，具体操作如图 5-15 所示。

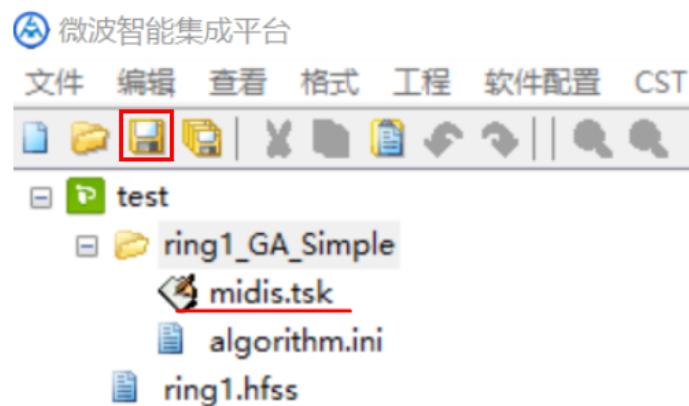


图 5-15 保存工程

打开任务管理面板，添加任务，输入需要的计算节点数目。然后点击开始按钮即可启动任务。

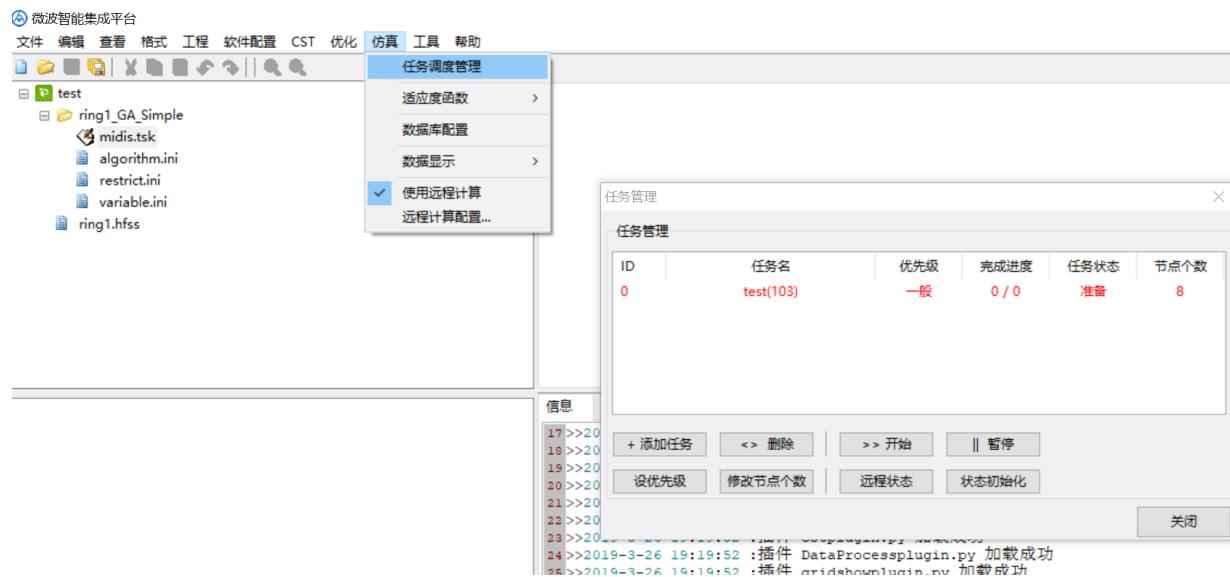


图 5-16 在任务管理面板中启动任务

5.2 小节 MIDIS 操作简化流程

第一小节具体介绍了操作的流程和需要注意的事项(红色加粗部分),可以将以上内容简化为图 5-17 所示的流程图。

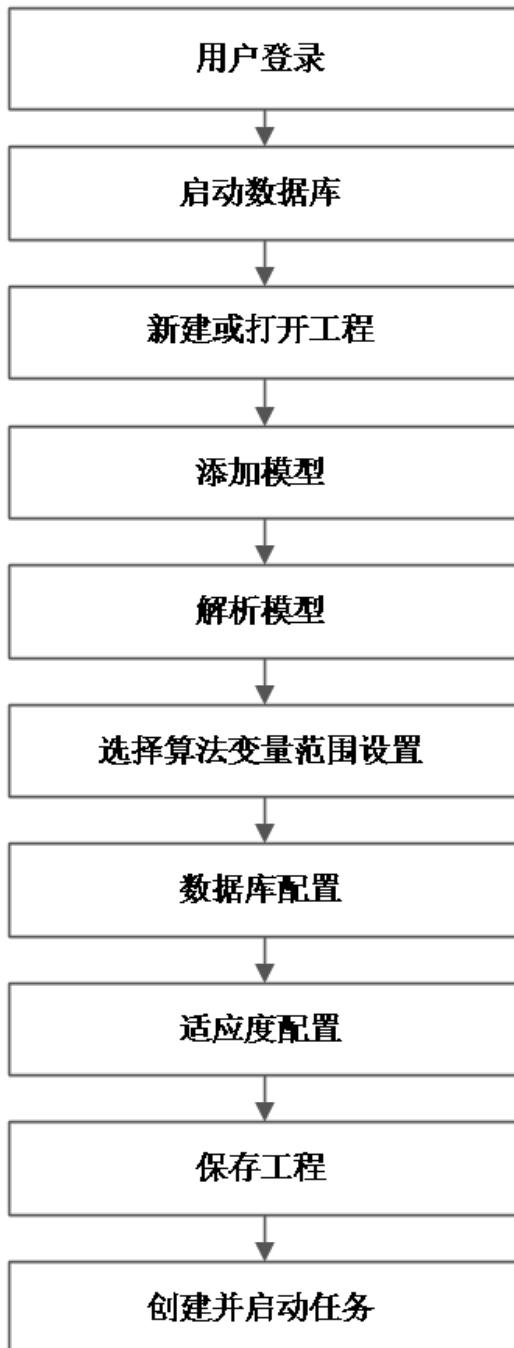


图 5-17 MIDIS 操作流程图

5.3 小节 MIDIS 源代码打包

针对 Python 的打包工具主要有三种，具体区别如图 5-18 所示。

工具	Windows	Linux	License
PyInstaller	支持	支持	GPL
Py2exe	支持	不支持	MIT
cx_Freeze	支持	支持	PSF

图 5-18 Python 打包工具对比

这里我们使用 PyInstaller 进行 MIDIS 的打包，首先按照 python2.7 版本的 PyInstaller 工具包，然后在命令行窗口定位到 MIDIS 的主 py 文件目录，这里需要注意，整个路径不能有中文字符，然后执行下面代码即可。

```
1 pyinstaller -F --icon=.\MIDIS.ico -c MIDIS.py
```

其中第一项 F 指定打包的 exe 文件的图标，-c MIDIS.py 需要打包的主 py 文件。成功打包后，会在 MIDIS 目录下产生两个文件夹，分别为 build 和 dist 其中打包的 exe 文件位于 dist 文件夹中。最后需要将 exe 文件和插件代码以及一些图片，外部文本文件拷贝在一起形成完整的打包软件。最终效果图如图 5-19 所示。

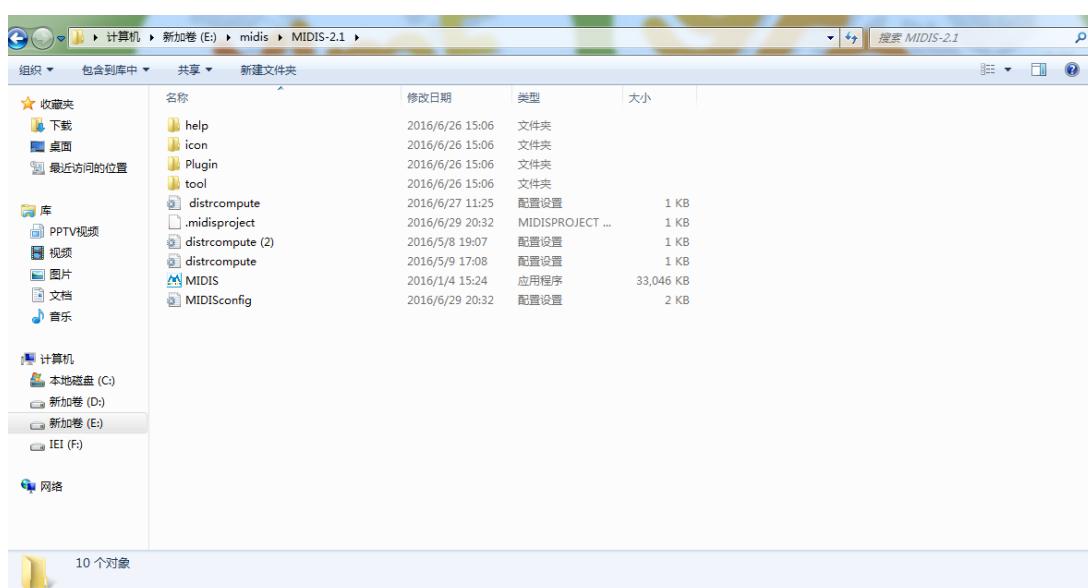


图 5-19 MIDIS 打包效果图

使用 PyInstaller 在 MIDIS 打包过程中会遇到一些问题，就本人在打包过程中遇到的一些问题与解决方案给出以下说明。

1. ascii / utf8 codec can't encode character ... in position ...

这种情况一般是因为编码问题，首先检查源码文件中是否有中文路径。然后对应着错误提示找到出错的文件，这里我遇到的是 ntpath.py 的编码问题，第一步在 python 下的 lib 下的 ntpath.py 中添加

```
1 #coding=utf-8  
2 import sys  
3 reload (sys)  
4 sys. setdefaultencoding ('utf-8')
```

第二步：将约 84 行的 result_path = result_path+ p_path 替换为 result_path = str(result_path) + str(p_path)。

2. no module named ...

很明显，在编译过程中找不到需要的库，但是我们在 Python 环境中是可以运行的，这些库也已经安装过。这里可能有一个原因，在安装 Python 库的过程中，有两种安装方式，一种是将库的源码安装到 site-packages 中，第二种方式是将库的源文件进行编译打包后安装到目录中 (egg 文件)。第二种方式 PyInstaller 是无法解析打包文件中的库，会报错。

首先我们在 Python 目录中找到报错的库，确认库存在，如果是只有 egg 文件，且没有其他同名目录文件，则需要下载源代码重新安装到目录中，或者直接将同名目录代码拷贝到 site-packages 中去。

3. No module named ... 找不到库的依赖文件

和上一个问题不同的是，它并不缺少库，而是找不到库的一个依赖文件，但是经过检查库的源文件和依赖文件也在。这时可以采用的解决方案是将该包拷贝到程序源代码目录下，然后在代码主 py 文件中强制将该包导入。在安装过程中我遇到的问题是 No module named FixTk，并且在编译日志中看到 WARNING : Removing import "FixTk" FixTk.py 是 Tkinter 库的依赖文

件。将该文件所在的目录 (lib-tk) 拷贝到代码目录中，然后在程序主 Py 文件 (MIDIS.py) 添加一行代码: `sys.path.append('lib-tk')`

4. 运行 exe 文件卡死

请检查中文路径，将文件放在全英文环境中，然后重新运行。

5. 缺少 DLL 或其他与 Windows 环境相关的错误

这可能和 Windows 环境有关，检查 microsoft visual c++ 的版本，如果版本过高，请降至 microsoft visual c++ 2010。

以上就是我遇到的一些常见问题，当然在具体操作过程中也许还会遇到一些其他问题，因为这个我没有遇到过，所有无法给出解决方案。当遇到问题时，希望能够冷静等待，不要轻易放弃，追根溯源，一定会找到一个解决办法。