在 FoxTable 中直接使用 PythonNet 调用 Python 代码确实有一些限制，因为 FoxTable 的脚本环境不完全支持 .NET 的所有特性。不过，你可以通过创建一个简单的 .NET 类库来封装 PythonNet 的调用，然后在 FoxTable 中引用这个类库。这样可以简化 FoxTable 中的代码，并且更容易维护。

**步骤 1：创建 .NET 类库项目**

1. 打开 Visual Studio 或其他 .NET 开发环境。
2. 创建一个新的类库项目（Class Library）。
3. 添加对 Python.Runtime.dll 的引用（该文件通常位于 Python 安装目录下的 Lib\site-packages\pythonnet 文件夹中）。

**步骤 2：编写 .NET 类库代码**

在类库项目中创建一个新的类文件，例如 PythonCaller.cs：

csharp

using System;

using Python.Runtime;

namespace PythonCallerLibrary

{

 public class PythonCaller

 {

 public static void ExecutePythonCode(string pythonCode)

 {

 using (Py.GIL()) *// 获取 Python 的全局解释器锁*

 {

 PythonEngine.Exec(pythonCode);

 }

 }

 public static string ExecutePythonFunction(string moduleName, string functionName, params object[] args)

 {

 using (Py.GIL()) *// 获取 Python 的全局解释器锁*

 {

 dynamic pyModule = Py.Import(moduleName);

 dynamic result = pyModule.GetType().GetMethod(functionName).Invoke(null, args);

 return result.ToString();

 }

 }

 public static string ExecutePythonScript(string scriptPath, string functionName, params object[] args)

 {

 using (Py.GIL())

 {

 dynamic pyModule = Py.Import(Path.GetFileNameWithoutExtension(scriptPath));

 dynamic result = pyModule.GetType().GetMethod(functionName).Invoke(null, args);

 return result.ToString();

 }

 }

 }

}

**步骤 3：编译并引用类库**

1. 编译类库项目，生成 PythonCallerLibrary.dll。
2. 在 FoxTable 中引用这个 DLL 文件：
	* 打开 FoxTable 项目。
	* 在菜单栏中选择 工具 -> 选项 -> 其他 -> .NET 程序集。
	* 点击 添加 按钮，选择 PythonCallerLibrary.dll 文件。

**步骤 4：在 FoxTable 中调用 .NET 类库**

在 FoxTable 的按钮点击事件或其他事件中编写以下代码：

vb

*' 调用 Python 代码*

Private Sub CallPythonCode()

 Dim pythonCode As String = "print('Hello from Python')"

 PythonCallerLibrary.PythonCaller.ExecutePythonCode(pythonCode)

End Sub

*' 调用 Python 函数并获取返回值*

Private Sub CallPythonFunction()

 Dim moduleName As String = "math"

 Dim functionName As String = "sqrt"

 Dim arg As Double = 16.0

 Dim result As String = PythonCallerLibrary.PythonCaller.ExecutePythonFunction(moduleName, functionName, arg)

 MessageBox.Show("Result from Python function: " & result)

End Sub

*' 调用 Python 脚本中的函数并获取返回值*

Private Sub CallPythonScriptFunction()

 Dim scriptPath As String = "C:\Path\To\Your\script.py"

 Dim functionName As String = "my\_function"

 Dim arg As Double = 25.0

 Dim result As String = PythonCallerLibrary.PythonCaller.ExecutePythonScript(scriptPath, functionName, arg)

 MessageBox.Show("Result from Python script function: " & result)

End Sub

**注意事项**

1. ​**路径问题**：确保 scriptPath 的路径正确，并且 FoxTable 有权限访问该路径。
2. ​**Python 环境**：确保 Python 环境和 Python.Runtime.dll 的版本兼容。
3. ​**错误处理**：在实际应用中，需要添加错误处理机制，确保 Python 脚本执行成功，并处理可能的异常情况。

**改进方法：使用文件传递输出**

如果需要获取 Python 脚本的输出，可以将结果写入文件，然后在 FoxTable 中读取文件内容。

**修改 Python 脚本**

python

import sys

def my\_function(x):

 return x ​\*\* 2

if \_\_name\_\_ == "\_\_main\_\_":

 if len(sys.argv) > 1:

 result = my\_function(float(sys.argv[1]))

 with open("output.txt", "w") as f:

 f.write(str(result))

 else:

 with open("output.txt", "w") as f:

 f.write("No argument provided")

**修改 FoxTable 代码**

vb

Private Sub CallPythonScriptFunction()

 Dim scriptPath As String = "C:\Path\To\Your\script.py"

 Dim arg As String = "25"

 Dim command As String = $"python {scriptPath} {arg}"

 Shell(command, AppWinStyle.Hide, True)

 *' 等待一段时间确保脚本执行完成*

 System.Threading.Thread.Sleep(1000)

 *' 读取输出文件*

 If File.Exists("output.txt") Then

 Dim result As String = File.ReadAllText("output.txt")

 MessageBox.Show("Result from Python script: " & result)

 Else

 MessageBox.Show("Failed to execute Python script or read output")

 End If

End Sub

**总结**

通过创建一个 .NET 类库来封装 PythonNet 的调用，可以简化 FoxTable 中的代码，并且更容易维护。虽然这需要一些额外的步骤，但它提供了一个更结构化和可维护的解决方案。如果需要更复杂的交互，建议还是考虑通过 .NET 类库或其他更高级的集成方式。

你可能还想问

这样我就可以在 FoxTable 中通过PythonCaller来调用Python了。

理想情况下，设计良好的 .NET 类库应该尽量减少因 Python 代码变化而需要修改的次数。为了实现这一点，可以采用一些设计模式和最佳实践，使得 .NET 类库更加灵活和可扩展。以下是一些改进建议：

**1. 使用配置文件管理 Python 脚本路径和函数名**

将 Python 脚本路径和函数名等配置信息放在配置文件中，而不是硬编码在 .NET 类库中。这样，当 Python 代码发生变化时，只需要修改配置文件，而不需要修改 .NET 类库代码。

**配置文件示例（appsettings.json）：**

json

{

 "Python": {

 "ScriptPath": "C:\\Path\\To\\Your\\script.py",

 "FunctionName": "my\_function"

 }

}

**修改 .NET 类库代码以读取配置文件：**

csharp

using System;

using System.IO;

using System.Text.Json;

using Python.Runtime;

namespace PythonCallerLibrary

{

 public class PythonCaller

 {

 private static string GetPythonScriptPath()

 {

 var configPath = Path.Combine(AppContext.BaseDirectory, "appsettings.json");

 var config = JsonSerializer.Deserialize<AppSettings>(File.ReadAllText(configPath));

 return config?.Python?.ScriptPath;

 }

 private static string GetPythonFunctionName()

 {

 var configPath = Path.Combine(AppContext.BaseDirectory, "appsettings.json");

 var config = JsonSerializer.Deserialize<AppSettings>(File.ReadAllText(configPath));

 return config?.Python?.FunctionName;

 }

 public static void ExecutePythonCode(string pythonCode)

 {

 using (Py.GIL()) *// 获取 Python 的全局解释器锁*

 {

 PythonEngine.Exec(pythonCode);

 }

 }

 public static string ExecutePythonFunction(string moduleName, string functionName, params object[] args)

 {

 using (Py.GIL()) *// 获取 Python 的全局解释器锁*

 {

 dynamic pyModule = Py.Import(moduleName);

 dynamic result = pyModule.GetType().GetMethod(functionName).Invoke(null, args);

 return result.ToString();

 }

 }

 public static string ExecutePythonScript(string arg = null)

 {

 string scriptPath = GetPythonScriptPath();

 string functionName = GetPythonFunctionName();

 if (string.IsNullOrEmpty(scriptPath) || string.IsNullOrEmpty(functionName))

 {

 throw new InvalidOperationException("Python script path or function name is not configured.");

 }

 using (Py.GIL())

 {

 dynamic pyModule = Py.Import(Path.GetFileNameWithoutExtension(scriptPath));

 dynamic result = pyModule.GetType().GetMethod(functionName).Invoke(null, new object[] { arg });

 return result.ToString();

 }

 }

 }

 public class AppSettings

 {

 public PythonConfig Python { get; set; }

 }

 public class PythonConfig

 {

 public string ScriptPath { get; set; }

 public string FunctionName { get; set; }

 }

}

**2. 使用参数化调用**

将 Python 函数的参数作为方法的参数传递，而不是硬编码在 .NET 类库中。这样可以提高代码的灵活性和可重用性。

**修改 ExecutePythonScript 方法以接受参数：**

csharp

public static string ExecutePythonScript(string scriptPath, string functionName, params object[] args)

{

 if (string.IsNullOrEmpty(scriptPath) || string.IsNullOrEmpty(functionName))

 {

 throw new InvalidOperationException("Python script path or function name is not configured.");

 }

 using (Py.GIL())

 {

 dynamic pyModule = Py.Import(Path.GetFileNameWithoutExtension(scriptPath));

 dynamic result = pyModule.GetType().GetMethod(functionName).Invoke(null, args);

 return result.ToString();

 }

}

**3. 使用反射动态调用函数**

如果 Python 函数的名称和参数可能会频繁变化，可以考虑使用反射动态调用函数。这种方法可以进一步提高代码的灵活性，但也会增加复杂性。

**动态调用函数的示例：**

csharp

public static string ExecutePythonFunctionDynamic(string moduleName, string functionName, params object[] args)

{

 using (Py.GIL())

 {

 dynamic pyModule = Py.Import(moduleName);

 var method = pyModule.GetType().GetMethod(functionName);

 if (method == null)

 {

 throw new InvalidOperationException($"Function {functionName} not found in module {moduleName}.");

 }

 dynamic result = method.Invoke(null, args);

 return result.ToString();

 }

}

**4. 使用接口和依赖注入**

如果 .NET 类库需要支持多种脚本语言或不同的调用方式，可以考虑使用接口和依赖注入（DI）来设计类库。这种方法可以提高代码的可扩展性和可测试性。

**定义接口：**

csharp

public interface IPythonCaller

{

 void ExecutePythonCode(string pythonCode);

 string ExecutePythonFunction(string moduleName, string functionName, params object[] args);

 string ExecutePythonScript(string scriptPath, string functionName, params object[] args);

}

**实现接口：**

csharp

public class PythonCaller : IPythonCaller

{

 public void ExecutePythonCode(string pythonCode)

 {

 using (Py.GIL())

 {

 PythonEngine.Exec(pythonCode);

 }

 }

 public string ExecutePythonFunction(string moduleName, string functionName, params object[] args)

 {

 using (Py.GIL())

 {

 dynamic pyModule = Py.Import(moduleName);

 dynamic result = pyModule.GetType().GetMethod(functionName).Invoke(null, args);

 return result.ToString();

 }

 }

 public string ExecutePythonScript(string scriptPath, string functionName, params object[] args)

 {

 using (Py.GIL())

 {

 dynamic pyModule = Py.Import(Path.GetFileNameWithoutExtension(scriptPath));

 dynamic result = pyModule.GetType().GetMethod(functionName).Invoke(null, args);

 return result.ToString();

 }

 }

}

**在 FoxTable 中使用依赖注入：**

vb

Private Sub CallPythonScriptFunction()

 Dim pythonCaller As IPythonCaller = New PythonCaller()

 Dim scriptPath As String = "C:\Path\To\Your\script.py"

 Dim functionName As String = "my\_function"

 Dim arg As Double = 25.0

 Dim result As String = pythonCaller.ExecutePythonScript(scriptPath, functionName, arg)

 MessageBox.Show("Result from Python script function: " & result)

End Sub

**总结**

通过以上方法，可以设计一个更加灵活和可维护的 .NET 类库，减少因 Python 代码变化而需要修改 .NET 类库的次数。具体选择哪种方法取决于你的具体需求和项目的复杂性。