RobotVisionSuite

二次开发教程

PERCIPIO.XYZ

RobotVisionSuite

图漾科技

2023.07

关于本手册

本手册主要介绍图漾 RobotVisionSuite 平台软件（下简称 RVS ）基于 python 与 C++ 进行二次开发。文档结构如下:

章节	标题	内容
序章	开发人员基础技术能力要求	阐述对开发人员的一些基础能力要求
第一章	C++二次开发实现	提供了RVS中常用的函数接口，介绍了如何使用RVS中的编译环境对TCP通讯类算子进行编写和编译，实现一些简单的业务功能，以及分别以排序算法和模板匹配的算子为例，介绍了如何编写功能类算子和线程类算子
第二章	Python二次开发实现	介绍了使用Python进行二次开发的流程，介绍PythonThread算子、常见的错误操作以及数据类型定义格式
附录A	支持资源	介绍软件工具和相关的技术与文档支持。

发布说明

日期	版本	发布说明
2022.06	V1.0	第一次发布
2022.10	V2.0	新增Python二次开发
2023.02	V2.1	新增数据类型定义格式
2023.03	V2.2	对python算子以及AI算子进行了调整
2023.03	V2.3	新增对windows版二次开发的引导

免责和版权声明

这个文档本身可能包含印刷错误和产品技术说明方面的错误。图漾有权在不通知用户的情况下，对产品的使用说明做更改。客户在购买产品的时候，须向当地经销商索取最新的产品使用说明。
图漾保证本产品符合注明的质量标准，并在质保期内承担产品的质量保责任。但本产品只能用作指定用途，将产品挪作它用而造成的损失，图漾不承担任何责任。

1. 第一章 开发人员基础技术能力要求

RobotVisionSuite（以下简称RVS）是图漾信息科技有限公司（以下简称图漾）自行研发的一款3D机器视觉开发平台。可以利用已有的算子快速搭建项目、验证项目可行性、开发出原型程序以供客户测试。RVS软件，提供了一种 docker 的安装版本作为完整的二次开发平台，同时也提供了两种二次开发编程语言（C++、Python），使得客户可以将已有的算法或新的算法重新在RVS软件平台中开发、编译实现自己需求算子功能，极大降低了应用工程师的开发难度、开发周期以及项目交接难度，同时 RVS 提供了加密保护的功能，使得他人无法直接查看算法的具体内容，保护客户的知识产权。

开发人员基础技术能力要求：

- 有一定的系统使用经验（如Linux）
- 熟练掌握 RVS 软件平台启动和加载运行工程文件 xml
- 自行创建 3D 机器视觉工程，并熟练运用基本 3D 视觉算子节点
- 熟练使用 RVS 软件平台提供的手眼标定算法和 AI 训练算法
- 能看懂RVS软件Log窗口的各种信息提示
- 有一定C++、Python编程经验
- 可以使用RVS软件平台提供的API接口自行编写3D视觉软件插件算子节点
- 有一定的机器人方面的知识和空间坐标转换方面的知识

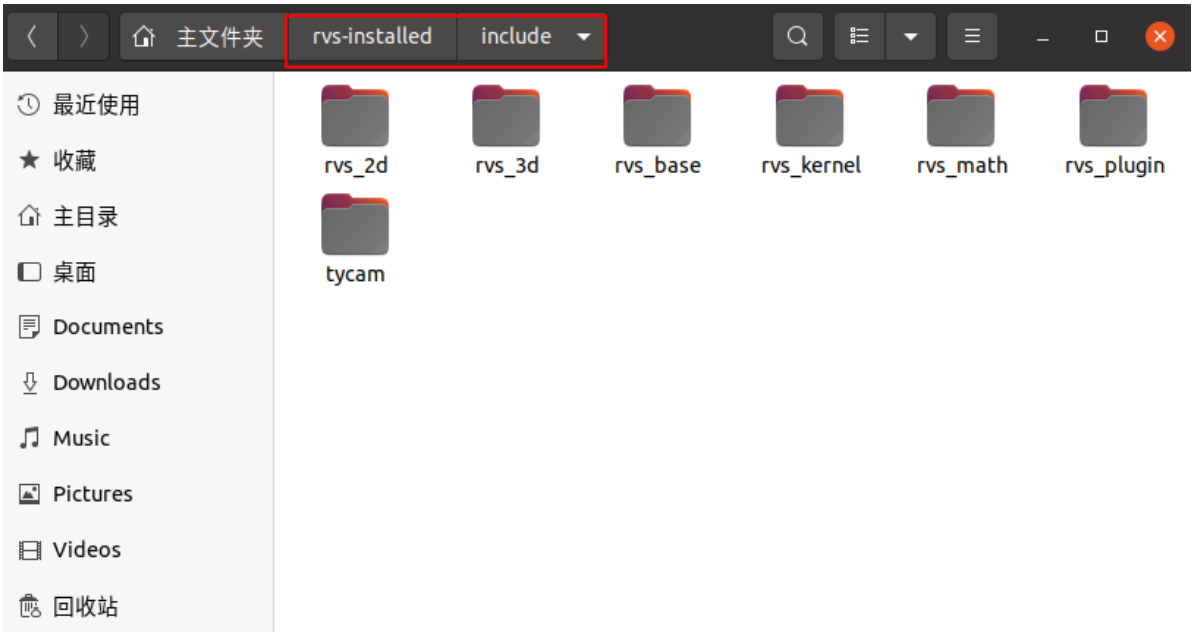
  当您开始 RVS 二次开发教程的时候，您应该已经可以熟练的使用 RVS 软件了，因此本教程不再涉及基础操作和算子介绍部分。

注：此文档中使用的所有案例代码，均提供在 Rvs_ProgRamming.zip 中，详情请下载参阅。

2. 第二章 C++二次开发实现

2.1. 2.1 RVS提供函数接口列表

RVS 软件提供了大量的函数接口，为了方便查询，将一些常用的接口统一整理在此处，除此外还有部分对于数据处理或相机参数接口，请在安装目录下的 `include\` 文件夹下自行查询。



2.1.1. 2.1.1添加参数

定义于头文件： < CommandServerNode > < Node > < TCPServerNode > < ThreadNode >	
void AppendBoolParameter(const std::string& name, bool& variable, bool default_val=false)	在属性面板上添加 Bool 类型的参数
void AppendIntParameter(const std::string& name, int& variable, int default_val=0)	添加 Int 类型的参数
void AppendFloatParameter(const std::string& name, float& variable, float default_val=0.f)	添加 Float 类型的参数
void AppendDoubleParameter(const std::string& name, double& variable, double default_val=0.0)	添加 Double 类型的参数
void AppendStringParameter(const std::string& name, std::string& variable, const std::string& default_val="")	添加 String 类型的参数
void AppendFileParameter(const std::string& name, std::string& variable,const std::string& extension="",const std::string& default_val="")	添加 File 类型的参数，可在本地路径选择文件
void AppendDirectoryParameter(const std::string& name, std::string& variable, const std::string& default_val="")	添加 Dir 类型的参数，可在本地路径选择文件夹
void AppendEnumParameter(const std::vector< std::string >& enum_names, const std::string& name, std::string& variable,const std::string&default_val="")	添加Enum类型的参数，在属性面板上呈现下拉框，下拉框中内容需提前定义
void AppendPoseParameter(const std::string& name, Pose& pose,double default_x=0.0, double default_y=0.0, double default_z=0.0,double default_roll=0.0, double default_pitch=0.0, double default_yaw=0.0)	添加Pose类型的参数

2.1.2. 2.1.2添加/删减算子端口

定义于头文件：< CommandServerNode > < Node > < TCPServerNode > < ThreadNode >	
void RegisterInput(const std::string & type,const std::string & name)	注册输入节点，Input 类型
void RegisterInputList(const std::string & type,const std::string & name)	注册输入节点，InputList 类型
void RegisterOutput(const std::string & type,const std::string & name)	注册输出节点，Output 类型
void RegisterOutputList(const std::string & type,const std::string & name)	注册输出节点，OutputList 类型
bool UnregisterInput(unsigned int index)	取消注册输入，填入 index
bool UnregisterOutput(unsigned int index)	取消注册输出，填入 index
unsigned int GetInputDimension()	获得输入的维度
unsigned int GetOutputDimension()	获得输出的维度

2.1.3. 2.1.3定义事件

定义于头文件：< CommandServerNode > < Node > < TCPServerNode > < ThreadNode >	
typedef std::function<void(void)> EventFunction	用户自定义事件函数
void RegisterInputEvent(const std::string & name, EventFunction func)	注册一个Input端口，去触发这个算子中用户自定义的事件函数，与算子中Process()不冲突
bool IsEventIdle(int index)	检测这个事件是否时空闲状态
bool IsEventTriggered(int index)	检测这个事件是否被触发
bool IsEventError(int index)	检测这个事件是否出现错误
void SetEventIdle(int index)	当这个事件空闲时，定义这个算子需要完成什么业务
void SetEventTrigger(int index)	当这个事件被成功触发时，定义这个算子需要完成什么业务
void SetEventError(int index)	当这个事件发生错误时，定义这个算子需要完成什么业务

2.1.4. 2.1.4对不同类型数据操作

定义于头文件： < CommandServerNode > < Node > < TCPServerNode > < ThreadNode >	
const String * GetStringInput(int index) const std::vector< String> * GetStringInputList(int index) String * GetStringOutput(int index) std::vector< String> * GetStringOutputList(int index)	对String类型的数据进行操作，包含获得输入/输出类型为 Input 或 OutPut 和InputList 或 Output 的 String 数据
const Pose * GetPoseInput(int index) const std::vector< Pose > * GetPoseInputList(int index) Pose * GetPoseOutput(int index) std::vector< Pose> * GetPoseOutputList(int index)	以 Pose 为操作类型数据
const Image * GetImageInput(int index) const std::vector< Image > * GetImageInputList(int index) Image * GetImageOutput(int index) std::vector< Image > * GetImageOutputList(int index)	以 Image 为操作类型数据
const ImagePoints * GetImagePointsInput(int index) const std::vector< ImagePoints> * GetImagePointsInputList(int index) ImagePoints * GetImagePointsOutput(int index) std::vector< ImagePoints > * GetImagePointsOutputList(int index)	以 ImagePoints 为操作类型数据
const PointCloud * GetPointCloudInput(int index) const std::vector< PointCloud > * GetPointCloudInputList(int index) PointCloud * GetPointCloudOutput(int index) std::vector< PointCloud > * GetPointCloudOutputList(int index)	以 PointCloud 为操作类型数据
const Cube * GetCubeInput(int index) const std::vector< Cube > * GetCubeInputList(int index) Cube * GetCubeOutput(int index) std::vector< Cube > * GetCubeOutputList(int index)	以 Cube 为操作类型数据

定义于头文件： < CommandServerNode > < Node > < TCPServerNode > < ThreadNode >	
const Cylinder * GetCylinderInput(int index) const std::vector< Cylinder > * GetCylinderInputList(int index) Cylinder * GetCylinderOutput(int index) std::vector< Cylinder > * GetCylinderOutputList(int index)	以 Cylinder 为操作类型数据
const JointArray * GetJointArrayInput(int index) const std::vector< JointArray > * Get JointArray InputList(int index) JointArray * Get JointArray Output(int index) std::vector< JointArray > * Get JointArray OutputList(int index)	以 JointArray 为操作类型数据

2.1.5. 2.1.5 CommandServerNode重写函数

定义于头文件: < CommandServerNode >	
virtual void ParameterChanged(const std::string& name) {}	用于更改参数
virtual bool ReceivedCommand(const std::string & command, const std::string & message) { return true; }	CommandServerNode中将命令的接受和返回分为了两个函数，在这里主要进行信息的接受处理
virtual std::string AcknowledgeCommand(const std::string & command) { return ""; }	在这个函数中进行命令的返回处理

2.1.6. 2.1.6 TCPServerNode重写函数

定义于头文件： < TCPServerNode >	
virtual void ParameterChanged(const std::string& name) {}	用于更改参数
virtual std::string ProcessInputMessage(const std::string& message) { return "OK"; }	TCPServerNode中对于信息Message的主要处理在这个函数中实现

2.1.7. 2.1.7 Node重写函数

定义于头文件： < TCPServerNode >	
virtual void ParameterChanged(const std::string& name) {}	用于更改参数
virtual int Process() {return 0; }	Node的主要业务处理在这个函数中实现

2.1.8. 2.1.8 ThreadNode重写函数

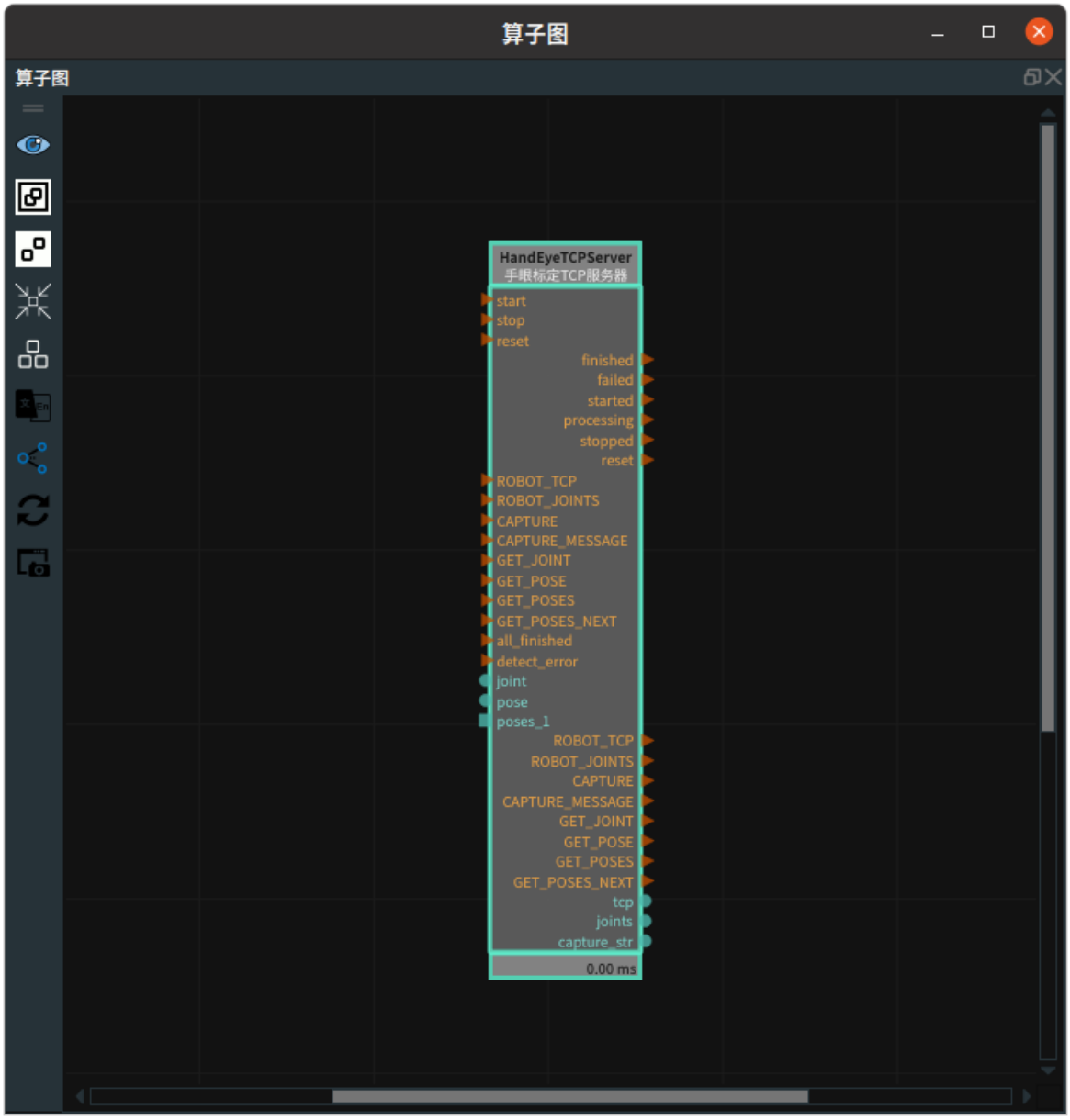
定义于头文件： < ThreadNode >	
virtual void ParameterChanged(const std::string& name) {}	用于更改参数
virtual void ThreadInit() { return; }	初始化线程，在线程开始时调用
virtual bool BeforeProcess() { return true; }	在线程处理前，应将Node中的数据拷贝到子线程中，如果返回False，则线程不会开始
virtual int ThreadProcess() { return -1; }	子线程中的主要处理函数在这里进行，这个线程不会进行sleep或wait，如果需要，应该在这里定义
virtual void ThreadFinish() { return; }	在这个函数中会结束子线程
virtual bool AfterProcess() { return true; }	在线程结束后，应将线程中的数据拷贝到Node中，如果返回False则Node会触发failed信号

2.2. 2.2 TCP通讯类算子以及编译

TCP通讯类算子的主要目的是，用于RVS软件同机器人或其他工控机程序之间的通讯。算子运行时会在RVS软件底层新建一个线程来创建并维持一个TCP Server端。

2.2.1. 2.2.1. 业务功能

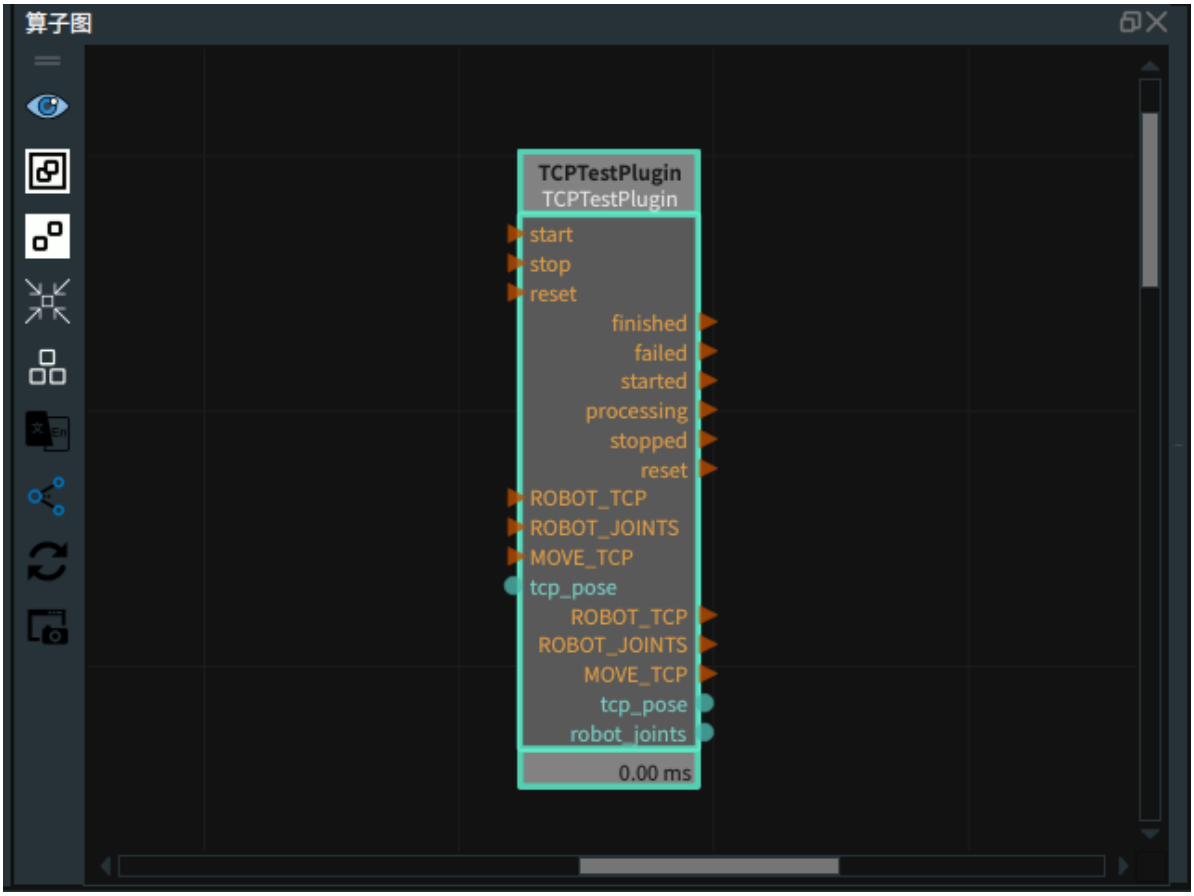
RVS软件中提供了一个标准的TCP通讯模块。如下：



类似 HandEyeTCPServer ，我们将重新编写一个简单的 TCPTestServer，要求：

- 信号流中具备传输 ROBOT_TCP、ROBOT_JOINTS、MOVE_TCP字 符串功能
- 数据流中具备接收 TCP 和 JOINTS 数据，发送 TCP 数据的功能

最后在RVS打开这个Node的效果应该如下：



2.2.2. 2.2.2. 代码解析

TCPTestPluginNode.h

需引入两个头文件，由于我们目标算子的类型是通讯类，在RVS中提供了相应的接口，此处只需引入 CommandServerNode.h 。同时我们希望算子工作时能正常记录 Log 日志，并显示在 UI 界面的 Log 显示区域中，此处也需引入 rvsLogging.h。

```
1.#include "rvs_plugin/CommandServerNode.h"
2.#include "rvs_plugin/rvsLogging.h"
```

定义派生类名称，注意要以 PluginNode 结尾，同时该类需要继承 rvs：：CommandServerNode ，继承方式为 public ，在继承后将自动包含 start、stop、finished、reset 等基本通用信号流，我们二次开发的目的只需要将特定功能集成在算子当中，基础框架是无法进行更改的。

```
3.class TCPTestPluginNode : public rvs::CommandServerNode
4.{
5.private:
6.
```

包含构造函数和析构函数各一个，分别用于创建算子时初始化，和销毁这个算子时回收资源，名称应与派生类名称一致。

```
7.public:
8.    TCPTestPluginNode();
9.    ~TCPTestPluginNode();
10.
```

在接口中我们定义了两个虚函数等待重写，作用分别是 ReceivedCommand 接受指令，并拆分为数据流和信号流、AcknowledgeCommand 确认好指令后反馈给客户端，并告知我们已经接受到了数据。

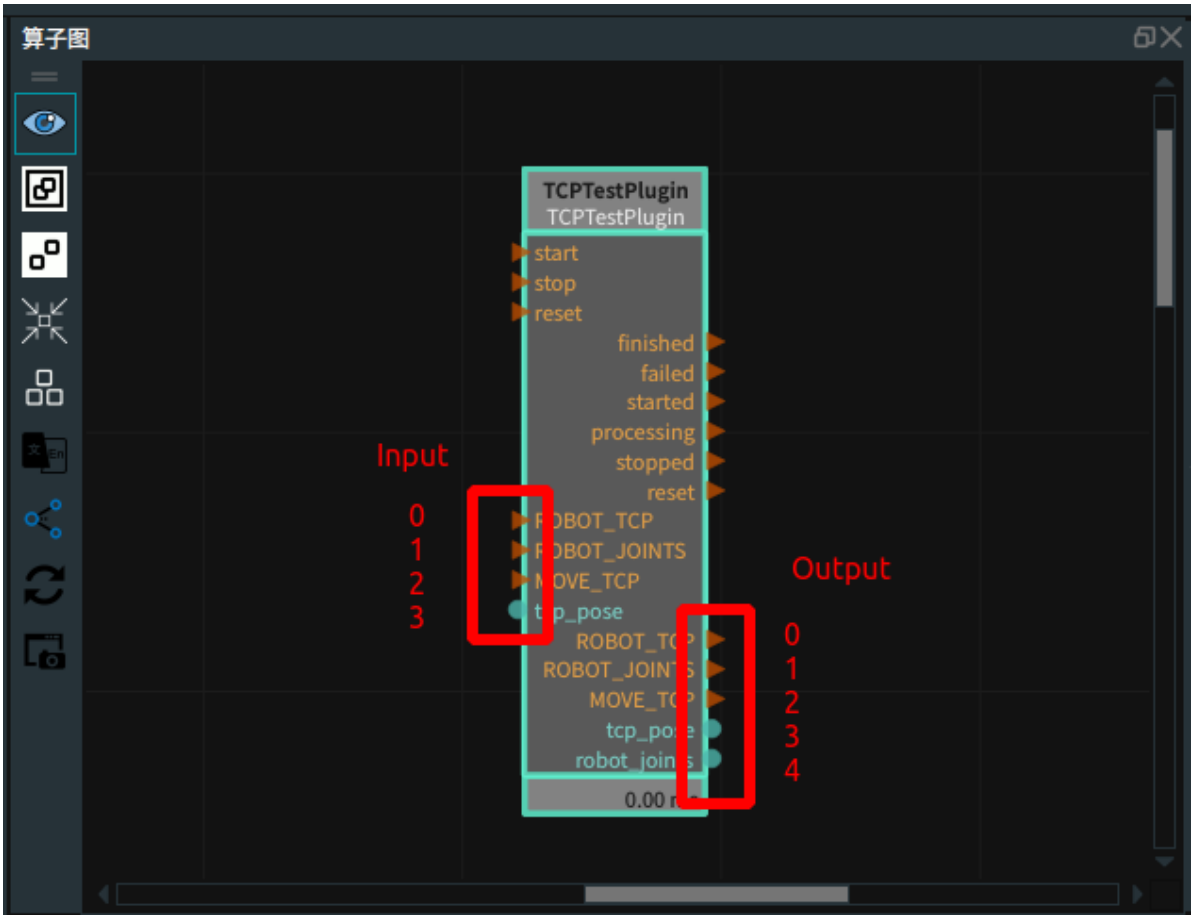
```
11.protected:
12.    bool ReceivedCommand(const std::string &command , const std::string & message)override;
13.    std::string AcknowledgeCommand(const std::string &command) override;
14.};
```

TCPTestPluginNode.cpp

引入TCPTestPluginNode.h头文件

```
1.#include "TCPTestPluginNode.h"
```

构造方法内：RegisterCommand用于给算子添加信号流，如运行图所示，此处示例添加了 ROBOT_TCP、ROBOT_JOINTS、MOVE_TCP 共 3 种信号流，同时请注意下方注释中的数字，其作为算子对应流的序号（从 0 开始），用于 GetPoseOutput()、GetJointArrayOutput() 等。GetXXXOutput()、GetXXXInput() 的参数，序号分为输入和输出流，不区分信号流和数据流，可对应运行图进行查看。



同时在这里使用了 RVS 的日志接口：RVS_PLUGIN_NODE_LOG()，括号内可以填入 fatal、error、warning、info、debug、trace状态，并在日志区域以不同的安全级别显示，此处填入warning，意味着当这个算子拖入时，在日志视图中会有一条黄色的日志抛出，并显示“tcp test server init success”提示算子生成成功。

```
2.TCPTestPluginNode::TCPTestPluginNode() : rvs::CommandServerNode("TCPTestPluginNode")
3.{
4.    RegisterCommands({"ROBOT_TCP","ROBOT_JOINTS","MOVE_TCP"});// 0 ,1 ,2
5.    RegisterOutput("Pose","tcp_pose"); // 3
6.    RegisterOutput("JointArray","robot_joint"); // 4
7.    RegisterInput("Pose","tcp_pose"); // 3
8.    RVS_PLUGIN_NODE_LOG(warning) << "tcp test server init success";
9.}
10.
11.TCPTestPluginNode::~TCPTestPluginNode()
12.{
13.}
```

ReceivedCommand 方法用于算子接收到客户端传输的数据后根据对应的命令执行操作，并将数据拆分为信号流和数据流传递给其他Node，方法参数 command为接收到的命令，message 为接收到除命令外的信息， 此处通过GetPoseOutput(3)绑定了运行途中右侧第4位输出流TCP_POSE（索引为3），调用 rvs_math 中关于pose的库（由于 CommandServerNode.h 中调用了 rvs_math/Pose.h，不需要重复调用）中的方法 FromString()，它会将所接受到的信息转换为 Pose 类型。

同时在这里也使用了RVS的日志接口：RVS_PLUGIN_NODE_LOG()，此处填入info，意味着当这个算子接收到 command 时，会在日志视图中一条普通颜色的日志抛出，并显示对应消息内容。

```
14.bool TCPTestPluginNode::ReceivedCommand(const std::string & command, const std::string & message)
15.{
16.    if(command == "ROBOT_TCP")
17.    {
18.        RVS_PLUGIN_NODE_LOG(info) <<"Rec > tcp_pose";
19.        Pose *tcp_pose = GetPoseOutput(3);
20.        if(tcp_pose->FromString(message))
21.        {
```

```
        RVS_PLUGIN_NODE_LOG(info) << "Rec > ROBOT_TCP : " << *tcp_pose;
        return true;
    }
    return false;
}
else if(command == "ROBOT_JOINTS")
{
    RVS_PLUGIN_NODE_LOG(info) <<"Rec > robot_joints";
    JointArray *tcp_joints = GetJointArrayOutput(4);
    if(tcp_joints->FromString(message))
    {
        RVS_PLUGIN_NODE_LOG(info) << "Rec > ROBOT_JOINTS : " <<*tcp_joints;
        return true;
    }
    return false;
}
else if(command == "MOVE_TCP")
{
    RVS_PLUGIN_NODE_LOG(info) << "Rec > MOVE_TCP ";
    return true;
}
else
    return false;
}
```

AcknowledgeCommand 方法用于向客户端传输数据，其返回值为传递给对方的数据。

> 注意：MOVE_TCP 命令通过 GetPoseInput(3) 绑定了左侧输入数据流 tcp_pose，可以通过前置算子传递数据至本算子，并最后发送给服务端。

> 注意：在此处为了展示多样性，将 RVS_PLUGIN_NODE_LOG() 替换成了std::cout 语句，这意味着当使用终端打开 RVS 软件时，将会有信息在终端中显示部分信息，请根据自己的需求考虑使用哪种 Log 方式。

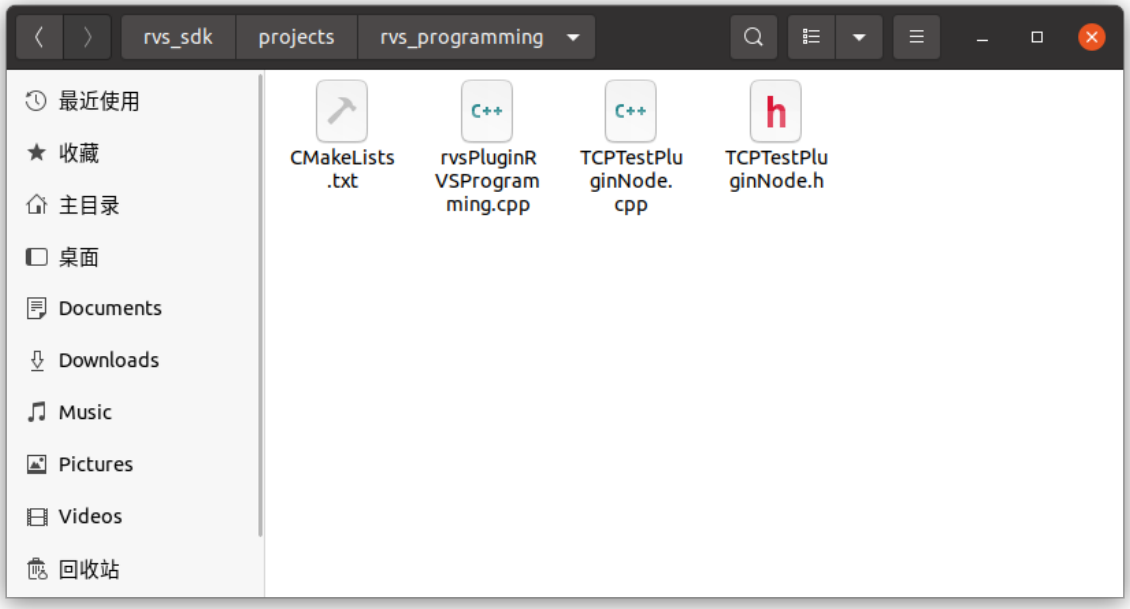
```
std::string TCPTestPluginNode::AcknowledgeCommand(const std::string &command)
{
    std::string ack_str;
    if(command == "ROBOT_TCP")
    {
        ack_str = "ROBOT_TCP";
        std::cout << "Ack > " <<ack_str <<std::endl;
        return ack_str;
    }
    else if (command == "ROBOT_JOINTS")
    {
        ack_str = "ROBOT_JOINTS";
        std::cout << "Ack > " << ack_str << std::endl;
        return ack_str;
    }
    else if(command == "MOVE_TCP")
    {
        ack_str = "MOVE_TCP";
        const Pose *to_tcp = GetPoseInput(3);
        ack_str += to_tcp->PoseToStr();
        std::cout << "Ack > " << ack_str <<std::endl;
        return ack_str;
    }
    else
        return "";
}
```

2.2.3. 2.2.3. 代码编译

当我们根据上述步骤已经完成了代码编写后，还需要通过 RVS 的 docker 环境进行编译才能将算子成功拖入到算子图当中。具体操作流程如下：

1)在安装目录的rvs_sdk/projects下,新建一个名为 rvs_programming 文件夹，并在 rvs_sdk/projects 下找到此处的 CMakeLists.txt ，填入我们要编译文件夹的名称 rvs_programming。

2)进入到 rvs_programming ，我们将写好的代码，放置到这个文件夹中，并且新创建两个文件，分别命名为 CMakeLists.txt 和rvsPluginRVSProgramming.cpp，这个文件夹的结构如下：



3)首先我们修改CMakeLists.txt中的内容，内容复制后修改即可。

> 注意：所有有关RVS_PROGRAMMING的文本内容，如果用户新建的文件夹命名不为rvs_programming请根据需要进行替换，额外注意大小写

```
1.SET(RVS_PROGRAMMING_PLUGIN_NODE_IDE_FOLDER "RVSProgrammingPluginNode")
2.SET(RVS_ PROGRAMMING _PLUGIN_NODE_HEADER_SUBDIR "RVSProgrammingPluginNode")
3.SET(RVS_ PROGRAMMING _PLUGIN_NODE_INCLUDE_DIRS
4.  ${CMAKE_CURRENT_SOURCE_DIR}
5.  ${RVS_INCLUDE_DIRS}
6.  CACHE INTERNAL "")
7.
8.# ============================================================ #
9.# Library rvs_plugin_rvs_programming
10.# ============================================================ #
```

```
11.ICMAKER_SET("rvs_plugin_rvs_programming" IDE_FOLDER ${RVS_ PROGRAMMING
_PLUGIN_NODE_IDE_FOLDER})
在此处添加Source的cpp文件
12.ICMAKER_ADD_SOURCES(
13.  rvsPluginRVSProgramming.cpp
14.  )
在此处添加Node的cpp文件
15.ICMAKER_ADD_NODES(
16.  TCPTestPluginNode.cpp
17.  )
18.
19.ICMAKER_ADD_HEADERS(
20.  )
21.
22.ICMAKER_LOCAL_CPPDEFINES(-std=c++14)
23.ICMAKER_LOCAL_CPPDEFINES(-fPIC)
24.ICMAKER_LOCAL_CPPDEFINES(-DBOOST_LOG_DYN_LINK -Wno-deprecated)
25.ICMAKER_GLOBAL_CPPDEFINES()
26.ICMAKER_INCLUDE_DIRECTORIES(${RVS_ PROGRAMMING _PLUGIN_NODE_INCLUDE_DIRS})
27.
28.ICMAKER_DEPENDENCIES(
29.  EXPORT
30.  rvs_plugin
31.  Eigen
32.  OpenCV
33.  pcl
34.  Tycam
35.  )
36.
37.ICMAKER_BUILD_LIBRARY()
38.ICMAKER_INSTALL_HEADERS(${RVS_ PROGRAMMING _PLUGIN_NODE_HEADER_SUBDIR})
```

4)然后再修改 resource 文件，打开 rvsPluginRVS PROGRAMMING.cpp

```
1.#include <iostream>
2.#include "rvs_kernel/rvsPluginClass.h"
3.#include "rvs_plugin/rvsLogging.h"
```

在此处添加Node的头文件

```
4.#include "TCPTestPluginNode.h"
5.// ===== Factory Function - Plugin EntryPoint ==== //
6.
7.RVS_PLUGIN_EXPORT_C
8.auto GetPluginFactory() -> rvsPluginBase*
9.{
10.  static rvsPluginFactory pinfo = []{
11.    auto p = rvsPluginFactory("rvsPluginRVSProgramming", "rvs-1.0.5");
```

在此处注册相应的Node

```
12.  p.RegisterNode<TCPTestPluginNode>("TCPTestPluginNode");
13.    return p;
14.  }();
```

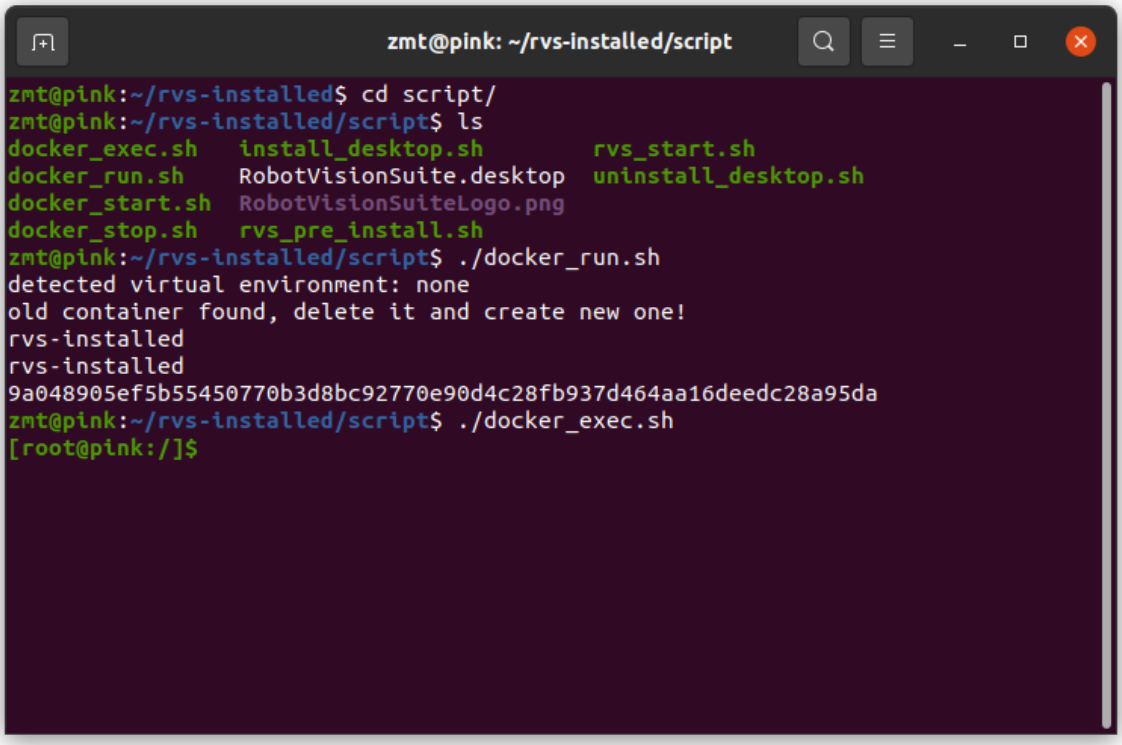
```
15.  return &pinfo;
16.}
17.
18.struct _DLLInit{
19.  _DLLInit(){
20.    RVS_PLUGIN_NODE_LOG(info) << "Shared library rvsPluginRVSProgramming loaded OK.";
21.  }
22.  ~_DLLInit(){
23.    RVS_PLUGIN_NODE_LOG(info) << "Shared library rvsPluginRVSProgramming unloaded OK.";
24.  }
25.} dll_init;
```

5)当我们已经编写好 CMakeLists.txt 和 rvsPluginRVSProgramming.cpp 后，如果我们在之前没有编译过RVS算子，则需要手动进入 rvs_sdk 目录下创建空的 build 文件夹用于编译。

6)现在让我们进入docker编译环境，如果没有安装请参考RVS安装手册。

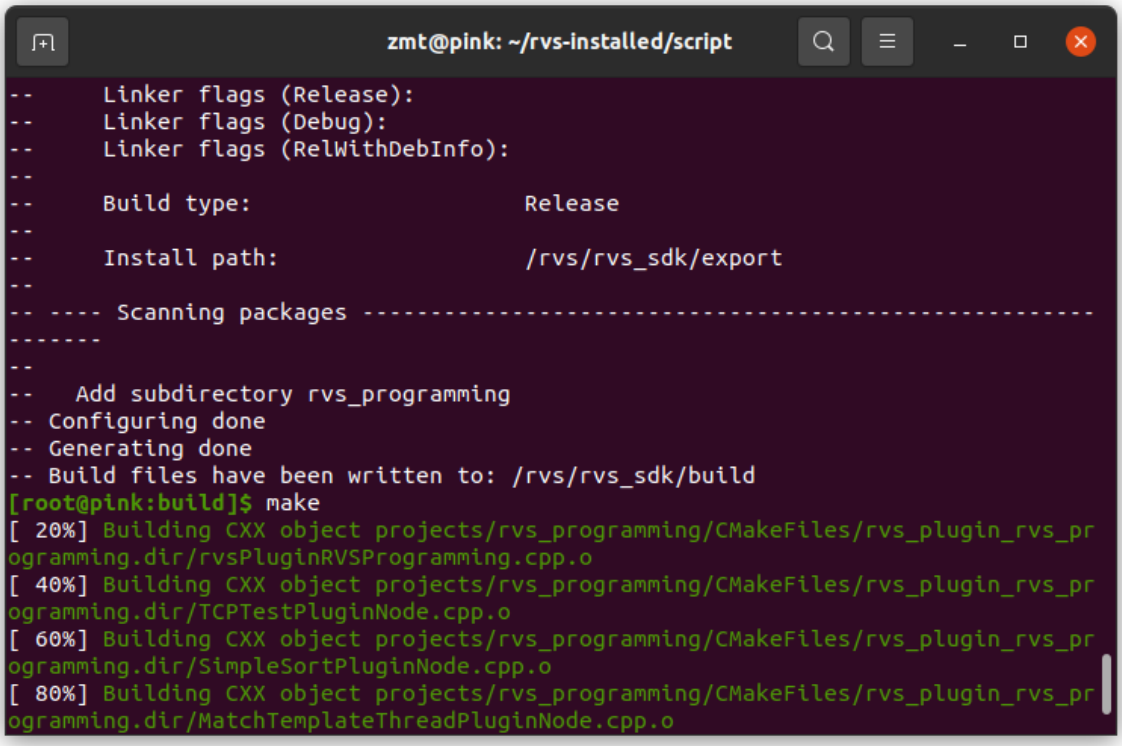
7)在终端中进入rvs安装目录下的script路径，分别输入

```
./docker_run.sh
./docker_exec.sh
```



8)在docker的环境中进入rvs/rvs_sdk/build路径，分别输入：

```
cmake ..
make
```

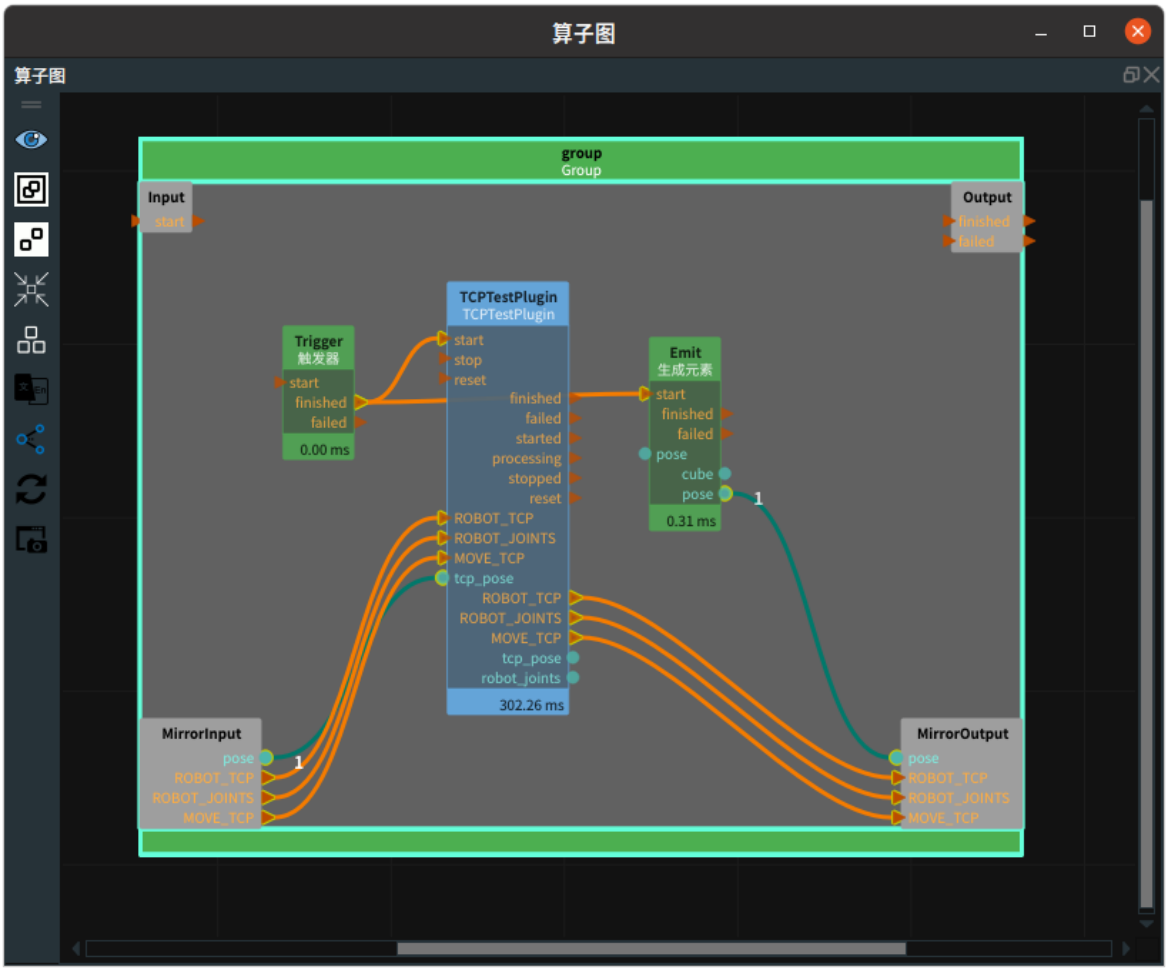


9)至此，已经完成算子的编译过程，需要用户重新打开RVS软件即可在算子列表中输入名称搜索到该算子。

2.2.4. 2.2.4. 代码测试

当我们已经完成了二次开发代码的编写及编译，我们现在可以进入到RVS软件当中测试我们新写的算子。

1)请根据下图，完整的连接出一个TCP本地通讯模组



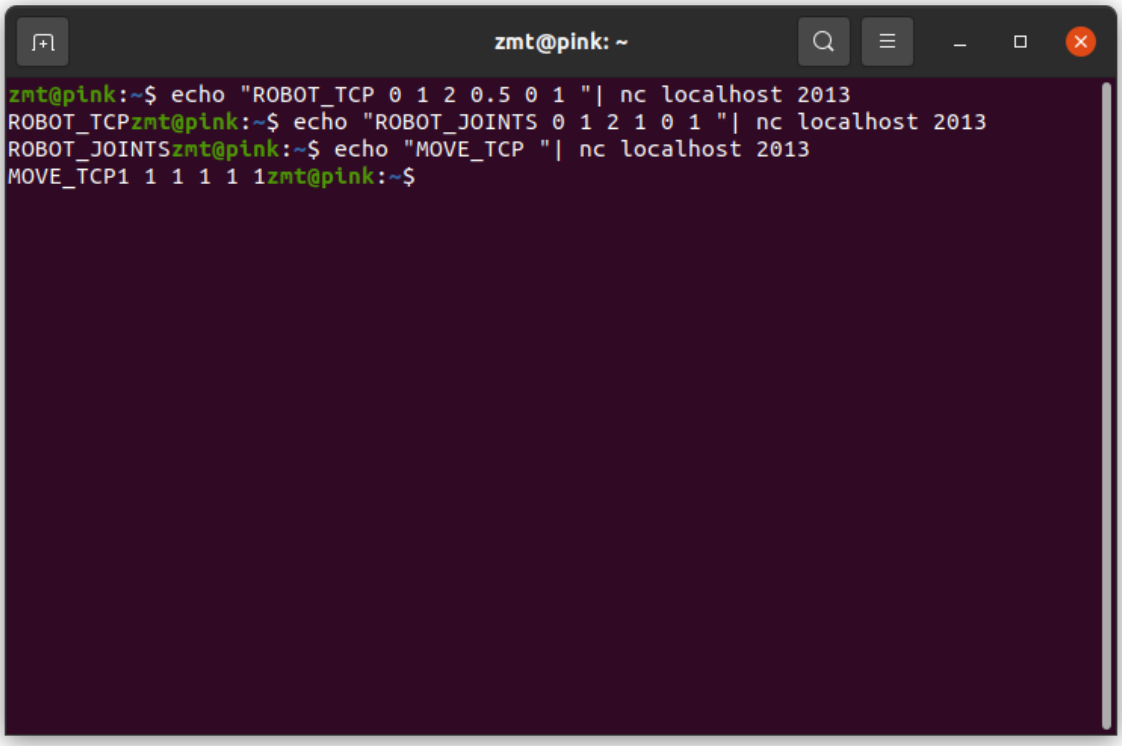
对于TCPTestPlugin算子参数无需任何改动，默认即可.Emit-Cube中在 pose 参数中，随意填写即可，本案例中填写的均为1，这意味着当我们发送MOVE_TCP将会收到 1 1 1 1 1 1的数据。

> 注意：在算子图中生成TCPTestPlugin时日志日图中是否有如下提示：

2023-04-24 16:38:21.157892	rvs_plugin_node	warning	tcp test server init success

2)当程序启动后，我们打开终端在本地进行通讯测试(x代表随意填入的数值，注意引号内结尾前的空格)：

```
echo “ROBOT_TCP x x x x x x ” | nc localhost 2013
echo “ROBOT_JOINTS x x x x x x ” | nc localhost 2013
echo “MOVE_TCP ” | nc localhost 2013
```



我们在终端中接收的信息，和我们代码中预期是一致的。

3)同时我们也可以在RVS中观察命令是否被正确拆分，先点击 TCPTestPlugin ，鼠标分别移动到数据流 tcp_pose、robot_joints、tcp_pose 上，会出现一个蓝色的提示框，观察其中的数值是否和我们终端输入的一致，也可以在日志视图中查看 info 级别的 Log 。

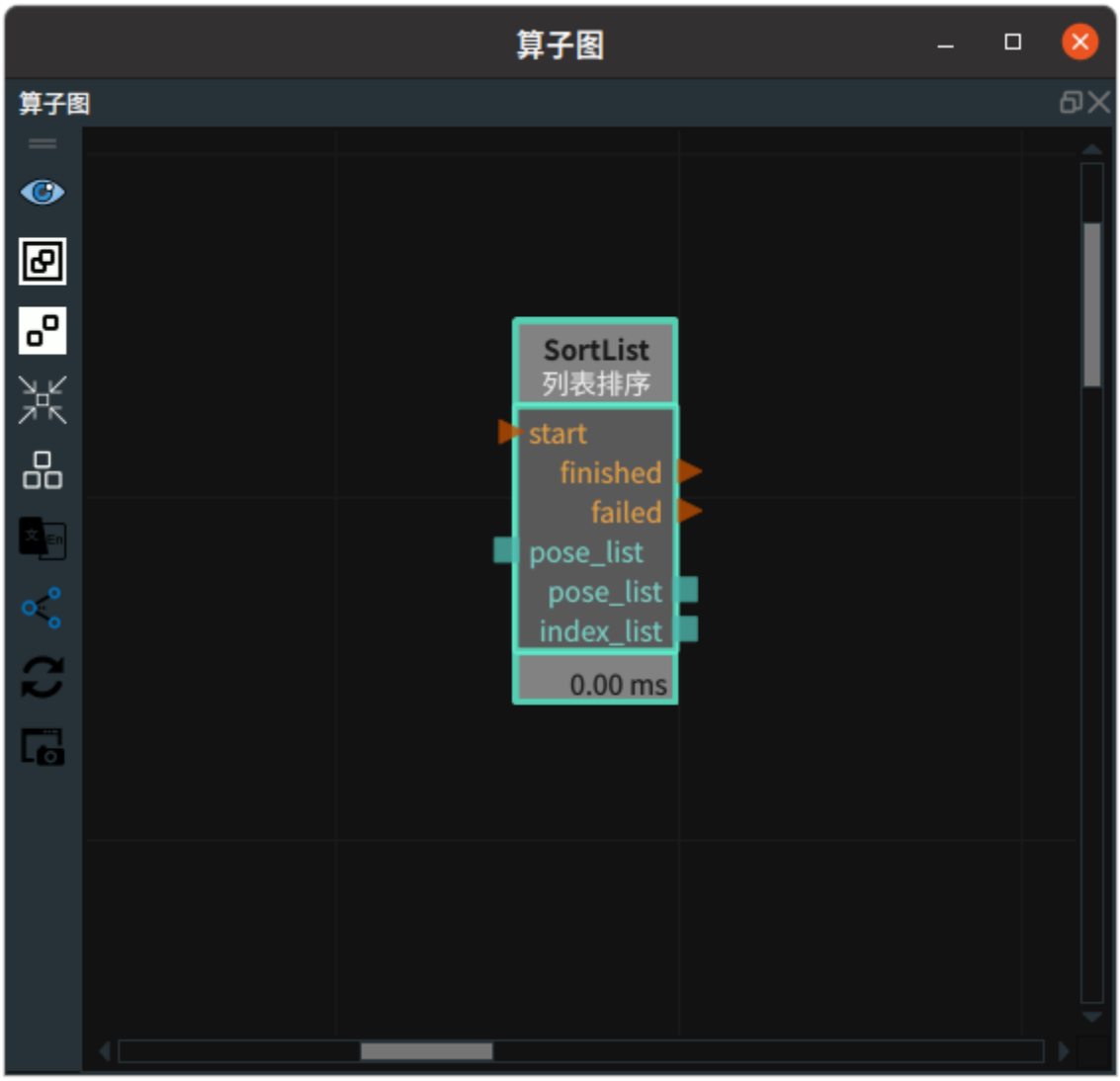
2023-04-24 16:41:09.591555	rvs_plugin_node	info	Rec > tcp_pose
2023-04-24 16:41:09.591604	rvs_plugin_node	info	Rec > ROBOT_TCP : 0 1 2 0.5 0 1
2023-04-24 16:41:09.591621	rvs_plugin	info	TCPTestPlugin: Triggered command via server: 3312: Trigger
2023-04-24 16:41:13.638264	rvs_plugin_node	info	Rec > robot_joints
2023-04-24 16:41:13.638297	rvs_plugin_node	info	Rec > ROBOT_JOINTS : 0 1 2 1 0 1
2023-04-24 16:41:13.638313	rvs_plugin	info	TCPTestPlugin: Triggered command via server: 3392: Trigger
2023-04-24 16:41:24.362825	rvs_plugin_node	info	Rec > MOVE_TCP
2023-04-24 16:41:24.362841	rvs_plugin	info	TCPTestPlugin: Triggered command via server: 3604: Trigger

2.3. 2.3 功能类算子

编写功能类算子的主要目的是，创建在RVS软件中完成一个特定功能或解决一系列特殊问题的算子。这个算子是依托于RVS的主线程内进行运算的。

2.3.1. 2.3.1. 业务功能

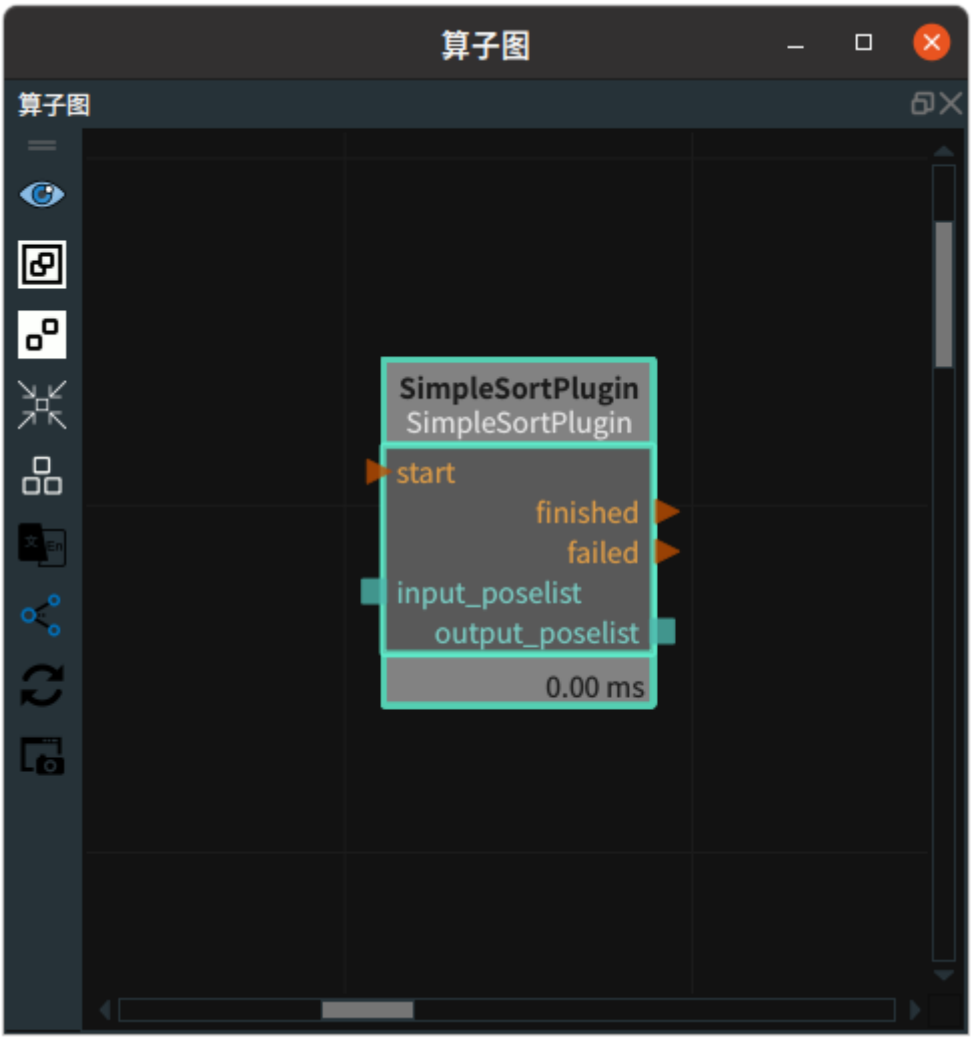
RVS软件中提供了非常多的功能类算子。如下：



类似 SortList，我们将重新编写一个简单的 SimpleSortPlugin，要求：

- 由于只是简单的排序演示，本案例设定只对 xy 方向进行排序
- 既可以设定 xy 的排序的优先顺序，又可以分别设定 xy 两个方向增序降序模式
- 设置 x 和 y 的误差精度，当 x 的差值小于设定值时，认为他们的 x 是相同的

最后在RVS打开这个Node的效果应该如下：



2.3.2. 2.3.2. 代码解析

SimpleSortPluginNode.h

需引入两个头文件，由于我们目标算子的类型是普通功能类的算子，在RVS中提供了相应的接口，此处需引入Node.h。同时我们也希望算子工作时能正常记录 Log日志，同样的此处也需引入 rvsLogging.h。对于算子的实现过程，在 Node.h 中给出了虚函数 Process() 等待重写。其余内容与 TCPTestPluginNode.h 基本一致，不再赘述。

```
1.#include "rvs_plugin/Node.h"
2.#include "rvs_plugin/rvsLogging.h"
3.class SimpleSortPluginNode : public rvs::Node
4.{
5.public:
6.  SimpleSortPluginNode();
7.  ~SimpleSortPluginNode();
8.
9.protected:
10.  virtual int Process() override;
11.
```

在这里，我们设置一些参数，有些将会被用于算子属性面板上的暴露，具体实现在cpp中，同时声明一个比较方法用于排序。

```
12.private:
13.  float x_dif_range;
14.  float y_dif_range;
15.  std::vector<std::string> m_sort_mode_name_list;
16.  std::string m_sort_mode_name;
17.  std::vector<std::string> m_sort_mode_list;
18.  std::string m_sort_mode;
19.  std::string m_sort_x_mode;
20.  std::string m_sort_y_mode;
21.  bool Compare(const Pose & pose1, const Pose & pose2);
22.};
```

SimpleSortPluginNode.cpp

首先我们还是引入 SimpleSortPluginNode.h 头文件。

```
1.#include "SimpleSortPluginNode.h"
```

构造方法内：使用 AppendEnumParameter() 方法为算子属性面板中设置一个下拉框，这个方法是 Node.h 中提供的标准接口，括号内的参数分别为下拉框内的选项、这个下拉框的名称、确认后赋值对象、初始默认值，因此我们就需要在在 AppendEnumParameter() 前定义好下拉框所有选项的名称。使用AppendFloatParameter() 方法为算子属性面板中设置一个浮点参数输入框，同样这个方法也是 Node.h 中提供的标注接口，括号内的参数分别为输入框名称、复制对象、初始默认值。其余注册算子的输入输出与上文保持一致。

其中当注册 outputlist 时，会自动继承父类的属性，即会在属性面板上出现 visibility 和 scale 的参数设置。

```
2.SimpleSortPluginNode::SimpleSortPluginNode()
3.  :rvs::Node("SimpleSortPluginNode")
```

```
4.{
5.  // append parameter
6.  m_sort_mode_name_list={"xy","yx"};
7.  m_sort_mode_list={"Descending","Ascending"};
8.  AppendEnumParameter(m_sort_mode_name_list, "sort_mode", m_sort_mode_name,
9.                  m_sort_mode_name_list[0]);
10.  AppendEnumParameter(m_sort_mode_list,"sort_x_mode",m_sort_x_mode,m_sort_mode_list[0]);

11.  AppendEnumParameter(m_sort_mode_list,"sort_y_mode",m_sort_y_mode,m_sort_mode_list[0]);

12.  AppendFloatParameter("x_dif_range", x_dif_range, 0.1f);
13.  AppendFloatParameter("y_dif_range", y_dif_range, 0.1f);
14.
15.  // register input
16.  RegisterInputList("Pose","input_poselist");   // 0
17.
18.  // register output
19.  RegisterOutputList("Pose","output_poselist"); // 0
20.}
21.
22.SimpleSortPluginNode::~SimpleSortPluginNode(){}
23.
```

方法 Process() 可以被视作算子运行的主程序，当算子开始运行时，这部分编写的内容将会承担算子功能的主体，那么在这个过程中总共分为三大步骤：

- 获取前项算子传输过来的一系列 poselist 值，判断是否为空，若不为空则进行下一步，若为空值则应该抛出一个黄色的 Log 提示用户进行操作，这个过程还是通过 RVS_PLUGIN_NODE_LOG() 实现
- 按照要求进行排序，针对于这个要求我们将在 Comapre 函数中进行实现
- 将已经排序好的 poselist ，遍历赋值给 output (需要清空，避免多次结果累计堆叠)，给下面的算子进行传输

```
24.int SimpleSortPluginNode::Process()
25.{
26.  // get input_poselist
27.  const PoseList* input_poselist = GetPoseInputList(0);
28.  if (input_poselist==NULL || input_poselist->empty())
29.  {
30.    RVS_PLUGIN_NODE_LOG(warning) << "input_poselist is null or empty";
31.    return 0;
32.  }
33.  // sort
34.  std::vector<Pose> poselist = *input_poselist;
35.  std::sort(poselist.begin(),poselist.end(),[this]
(const Pose & p1, const Pose & p2) { return Compare(p1, p2); });
36.  // get out_poselist
37.  std::vector<Pose>* output_poselist = GetPoseOutputList(0);
38.  output_poselist->clear();
39.  for (unsigned int i = 0; i < poselist.size(); i++)
40.  {
41.    output_poselist->push_back(poselist[i]);
42.  }
43.  return 1;
44.}
45.
```

下面是排序方法Compare的主要实现，先对于xy的排序顺序先进行判断，再对于误差精度进行判断，最后再根据增降序进行总结：

```
bool SimpleSortPluginNode::Compare(const Pose & pose1, const Pose & pose2)
{
  if (m_sort_mode_name=="xy")
    {
      if (abs(pose1.x-pose2.x)<=x_dif_range)
        {
          if (m_sort_y_mode=="Ascending")
            return pose1.y < pose2.y;
          else
            return pose1.y > pose2.y;
        }
      else
        {
          if (m_sort_x_mode=="Ascending")
            return pose1.x < pose2.x;
          else
            return pose1.x > pose2.x;
        }
    }
  else if (m_sort_mode_name=="yx")
    {
      if (abs(pose1.y-pose2.y)<=y_dif_range)
        {
          if (m_sort_x_mode=="Ascending")
            return pose1.x < pose2.x;
          else
            return pose1.x > pose2.x;
        }
      else
        {
          if (m_sort_y_mode=="Ascending")
            return pose1.y < pose2.y;
          else
            return pose1.y > pose2.y;
        }
    }
  else
    {
      RVS_PLUGIN_NODE_LOG(warning) << "m_sort_field_name: not supported";
      return false;
    }
  return false;
}
```

2.3.3. 2.3.3. 代码编译

当我们根据上述步骤已经完成了代码编写后，请先将两个文件放入我们编译的目录下，后与 2.2.3 中代码编译内容相似，也需要通过 RVS 的 docker 环境进行编译才能将算子成功拖入到算子图当中，这时候我们仅需要对原有文件进行简单修改即可。

1)首先我们修改rvs_sdk/projects/CMakeLists.txt中的内容

```
1.ICMAKER_ADD_NODES(
2.  TCPTestPluginNode.cpp
3.  SimpleSortPluginNode.cpp
4.  )
```

2)然后再修改相同路径下的 rvsPluginRVSProgramming.cpp 中的两处内容，第一处：

```
1.#include "rvs_plugin/rvsLogging.h"
2.#include "SimpleSortPluginNode.h"
```

第二处：

```
1. static rvsPluginFactory pinfo = []{
2.    auto p = rvsPluginFactory("rvsPluginRVSProgramming", "rvs-1.0.5");
3.    p.RegisterNode<SimpleSortPluginNode>("SimpleSortPluginNode");
4.    p.RegisterNode<TCPTestPluginNode>("TCPTestPluginNode");
5.    return p;
6.  }();
```

3)同样在 docker 的环境中进入 rvs/rvs_sdk/build 路径，分别输入：

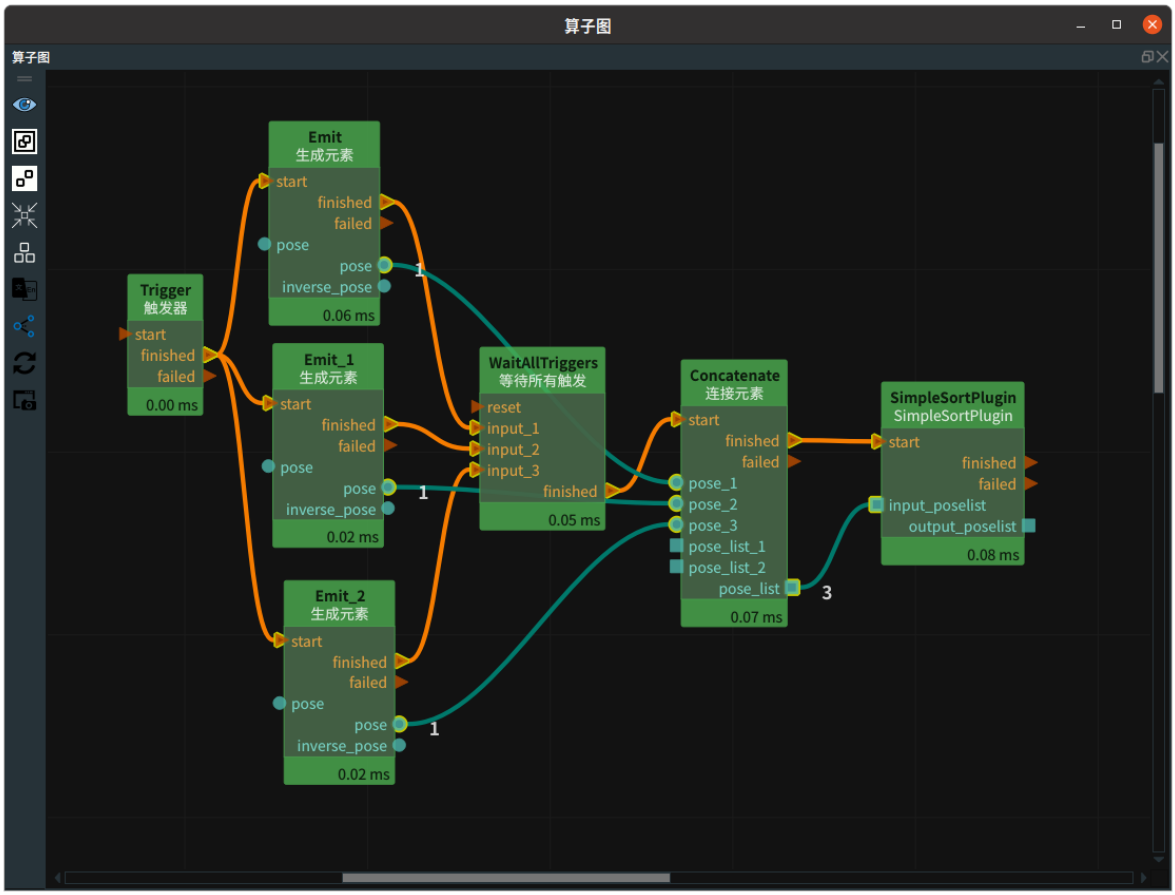
```
cmake ..
make
```

4)至此，已经完成排序算子的编译过程，需要用户重新打开 RVS 软件即可在算子列表中输入名称搜索到该算子 。

2.3.4. 2.3.4. 代码测试

当我们已经完成了排序算子二次开发代码的编写及编译，我们现在可以进入到RVS软件当中测试我们新写的算子

1)请根据下图，完整的连接出一个排序模组。



2)其中我们简单设置一下参数，对于 emitpose 我们只需设置 xy 即可，那么我们对于 emitpose、emitpose_1、emitpose_2 的 xy 分别设置(2,1)、(1,2)、(2,2)；对于SimpleSortPlugin算子，我们分别将参数设置为 xy、Descending、Descending、0.1、0.1。

3)我们点击ConcatenatePose，鼠标移动到pose_list上，在没有排序前顺序分别为:

[0] 2 1 0 0 0 0
[1] 1 2 0 0 0 0
[2] 2 2 0 0 0 0

4)我们点击SimpleSortPlugin，鼠标移动到output_poselist上，在排序后顺序分别为:

[0] 2 2 0 0 0 0
[1] 2 1 0 0 0 0
[2] 1 2 0 0 0 0

达到了我们预期的业务功能。

2.4. 2.4 线程类算子

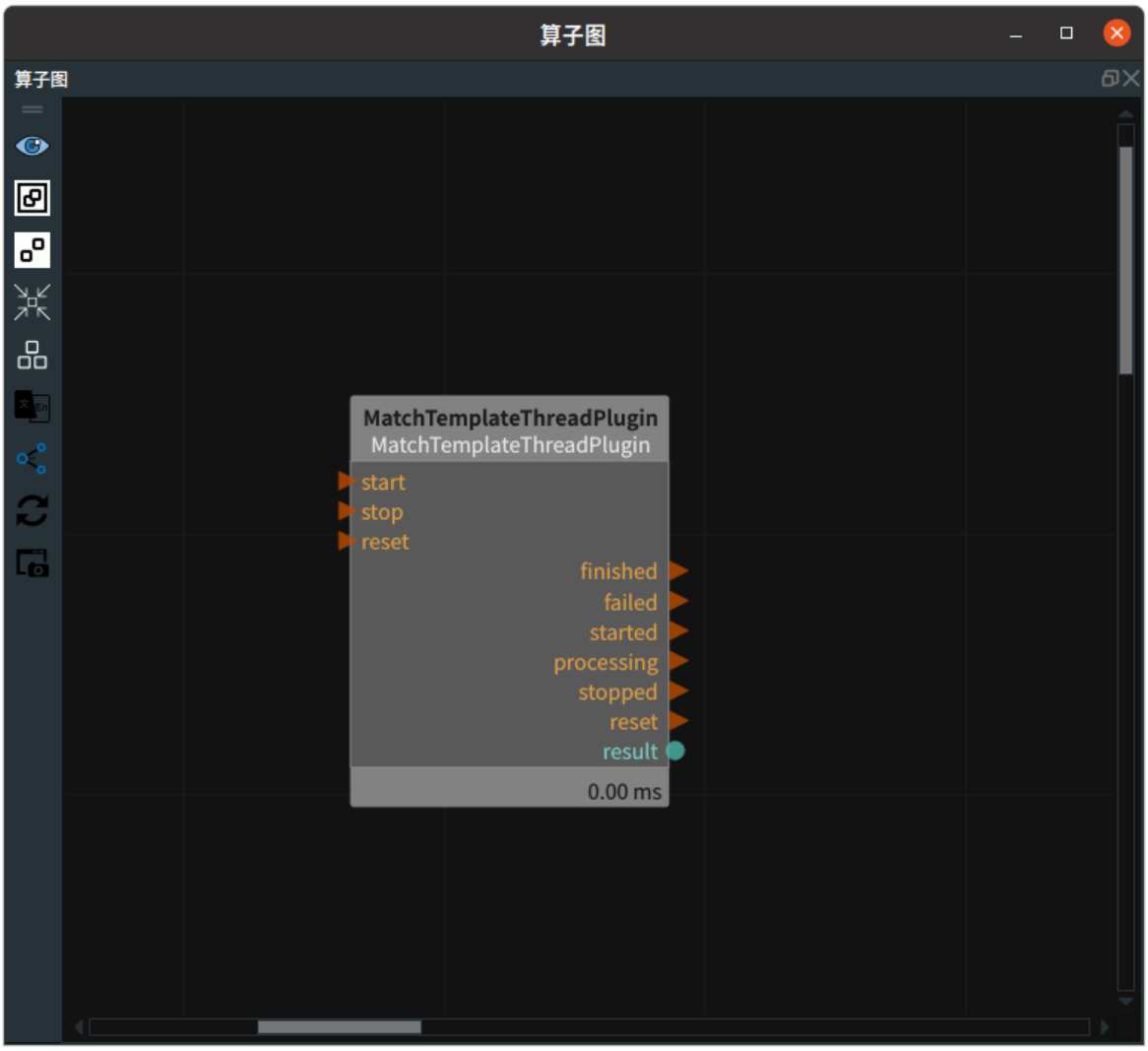
线程类算子的主要目的是，当一个普通算子要承担非常重要的计算任务时，它会花费大量的资源和时间，这样在它后面的算子就需要等到非常久才能正常运行，这时候我们就可以开辟一个新的子线程来使它在后台进行运算，不干扰主线程算子的计算来实现并行运算，加快工作节拍。与 TCP 通讯类算字不同的是，我们需要管理整个线程的初始化、加载、运算、以及回收工作。

2.4.1. 2.4.1. 业务功能

我们将编写一个简单的MatchTemplateThread，要求：

- 能实现基本的 2D 图像模板匹配
- 能通过算子读取本地文件

最后在RVS打开这个Node的效果应该如下：



2.4.2. 2.4.2. 代码解析

MatchTemplateThreadPluginNode.h

需引入三个头文件，由于我们目标算子的类型是线程类的算子，在RVS中提供了相应的接口，此处需引入ThreadNode.h；同时我们也希望算子工作时能正常记录Log日志，同样的此处也需引入rvsLogging.h；此外，由于我们需要对2D图像进行操作，则需要引入opencv.hpp文件；余内容与上述内容 基本一致，不再赘述。

```
1.#include "rvs_plugin/ThreadNode.h"
2.#include "rvs_plugin/rvsLogging.h"
3.#include "opencv2/opencv.hpp"
4.class MatchTemplateThreadPluginNode : public rvs::ThreadNode
5.{
6.private:
7.
8.public:
9.    MatchTemplateThreadPluginNode();
10.    ~MatchTemplateThreadPluginNode();
11.
```

对于算子的线程管理过程，在ThreadNode.h中给出了一系列虚函数 ThreadInit()、ThreadFinish()、ThreadProcess()、BeforeThread()、AfterThread() 等待重写。

```
12.protected:
```

```
    void ThreadInit() override;

    void ThreadFinish() override;

    bool BeforeProcess() override;

    int ThreadProcess() override;

    bool AfterProcess() override;
private:
    std::string m_img;
    std::string m_template;
    cv::Mat m_input_image;
    cv::Mat m_input_template;
};
```

MatchTemplateThreadPluginNode.cpp

引入 MatchTemplateThreadPluginNode.cpp 头文件。在这里添加FileParameter，为选择本地文件添加参数，这里注意括号内最后一个参数给的是空，默认不选择任何文件，需要手动选择。由于我们是图像的模板匹配，就要在输出时给出一个 Image 的输出。

```
#include "MatchTemplateThreadPluginNode.h"

MatchTemplateThreadPluginNode::MatchTemplateThreadPluginNode() : rvs::ThreadNode("MatchTemplateThreadPluginNode")
{
    AppendFileParameter("Img",m_img,"Load Image (*.png *.bmp *.jpg *.tiff)","");
    AppendFileParameter("Template",m_template,"Load Image (*.png *.bmp *.jpg *.tiff)","");
    RegisterOutput("Image","result"); //0
    RVS_PLUGIN_NODE_LOG(warning) << "MatchTemplateThreadPluginNode generate success";
}

MatchTemplateThreadPluginNode::~MatchTemplateThreadPluginNode()
{
}

```

ThreadInit() 的实现方法，在这里只是简单的提示线程在初始化，往往我们在做类似与AI训练和推理的时候，就需要在这个部分加载一些大文件进行缓存。

```
void MatchTemplateThreadPluginNode::ThreadInit()
{
    RVS_PLUGIN_NODE_LOG(debug)<<"MatchTemplateThreadPluginNode::ThreadInit()";
}

```

BeforeThread() 的实现方法，在这里主要完成对主线程传入的数据在子线程进行读取和加载，如果这里返回 false 那么子线程的 process 过程就不会进行。

```
bool MatchTemplateThreadPluginNode::BeforeProcess()
{
    RVS_PLUGIN_NODE_LOG(debug)<<"MatchTemplateThreadPluginNode::BeforeProcess()";
    m_input_image = cv::imread(m_img);
    m_input_template = cv::imread(m_template);
    return true;
}

```

ThreadProcess() 的实现方法，在这里主要进行的就是我们算法的主要处理部分。

```
int MatchTemplateThreadPluginNode::ThreadProcess()
{
    RVS_PLUGIN_NODE_LOG(debug)<<"MatchTemplateThreadPluginNode::ThreadProcess()";
    int result_cols = m_input_image.cols - m_input_template.cols + 1;
    int result_rows = m_input_image.rows - m_input_template.rows + 1;
    cv::Mat img_result;
    img_result.create(result_cols,result_rows,CV_32FC1);
    cv::matchTemplate(m_input_image, m_input_template, img_result, cv::TM_SQDIFF_NORMED);
    cv::normalize(img_result, img_result, 0, 1, cv::NORM_MINMAX, -1, cv::Mat());
    double minVal = -1;
    double maxVal;
    cv::Point minLoc;
    cv::Point maxLoc;
    cv::Point matchLoc;
    cv::minMaxLoc(img_result, &minVal, &maxVal, &minLoc, &maxLoc, cv::Mat());
    matchLoc = minLoc;
    cv::Point center = cv::Point(minLoc.x + m_input_template.cols / 2, minLoc.y + m_input_template.
rows / 2);
    cv::rectangle(m_input_image, matchLoc, cv::Point(matchLoc.x + m_input_template.cols, matchL
oc.y + m_input_template.rows), cv::Scalar(0, 255, 0), 2, 8, 0);
    cv::circle(m_input_image, center, 2, cv::Scalar(255, 0, 0), 2);
    return 1;
}

```

ThreadFinish()的方法实现，这里为空，不需要额外处理。

```
void MatchTemplateThreadPluginNode::ThreadFinish()
{
}

```

AfterThread() 的方法实现，在这里我们将计算好的数据传给主线程中的参数，以 Image 的形式给到下一个算子，如果这里返回 false，那么整个算子回触发 failed信号流。

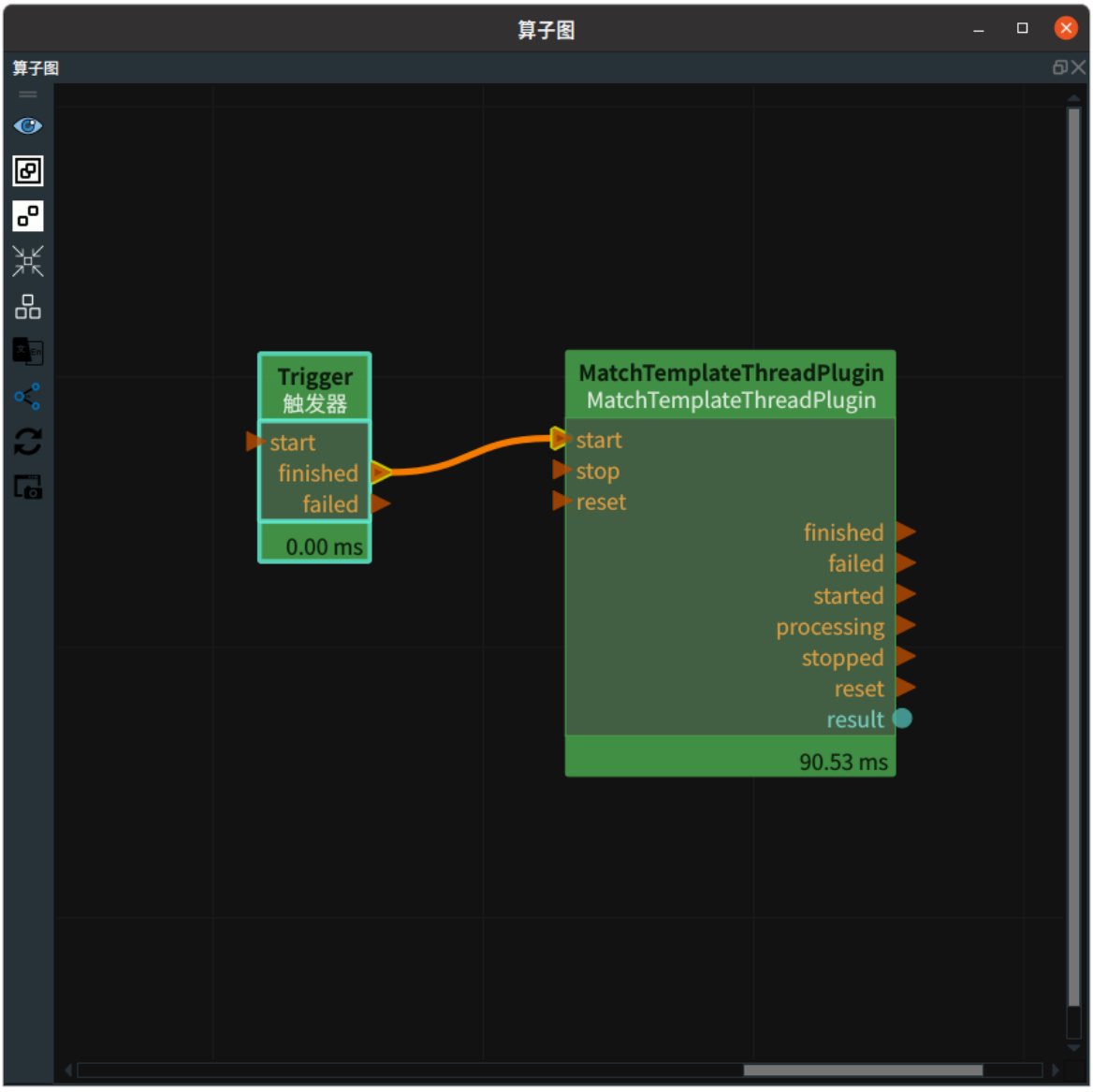
```
bool MatchTemplateThreadPluginNode::AfterProcess()
{
    RVS_PLUGIN_NODE_LOG(debug)<<"MatchTemplateThreadPluginNode::AfterProcess()";
    Image* m_result = GetImageOutput(0);
    *m_result=m_input_image;
    return true;
}
```

代码编写的过程已经结束，编译的过程与本章节中 2.2.3 相同，详情请参考。

2.4.3. 2.4.3. 代码测试

当我们已经完成了二次开发代码的编写及编译，我们现在可以进入到RVS软件当中测试我们新写的算子。

1)请根据下图，完整的连接出一个 MatchTemplateThreadPluginNode 本地测试模组。



2)请点击我们编写的算子MatchTemplateThreadPlugin，在属性面板中在Img中选择一张本地的原图像，在Template 中选择一张待测图像，并将result_visibility选为true，如下图：

Img 原图

Template待测图

3)让整个程序开始运行，触发 Trigger ，在这个过程中与 TCP 通讯类和功能类算子不同，在计算时这个算子会以蓝色提示子线程正在运行，当计算完成后才会恢复绿色，代表子线程结束。在最后我们可以在 2d 显示区域可以看到算法的结果，代表算子运行成功。

2.5. 2.5 Windows系统下的二次开发引导

在上述文档的基础上，Windows 版本 RVS 允许通过 Visual Studio 工程进行二次开发，其中 Visual Studio 版本不低于 VS2019，这也是我们推荐的 VS 版本（vc142），安装 RVS 后即可在安装路径下可以找到该工程，参见：

{RVS安装目录}\ rvs_sdk\RvsPluginNodeExample\RvsPluginNodeExample.sln

> 注意：该工程需在 Release x64 下编译
> 有关在 windows 下使用 Visual Studio 进行 RVS 二次开发算子的具体操作，参见该工程文件的 ReadMe.md 文档

3. 第三章 python二次开发实现

RVS支持用户使用 python语言编写算子文件（即.py文件），但是为了实现 python 算子文件同 RVS 软件的数据交互，需要用到统一的函数接口以及数据类型，下面是操作步骤。

3.1. 3.1.Python二次开发流程

3.1.1. 3.1.1 定义数据交互端口

需要先编写一个 xml 文档来确定 python 算子文件同 RVS 软件之间的数据交互端口，可以将 xml 文档存放在 runtime 目录下。xml 文档的示例如下所示：

```
<config>
<input type="Image"> im0 </input>
<output type="Image"> im1 </output>
</config>
```

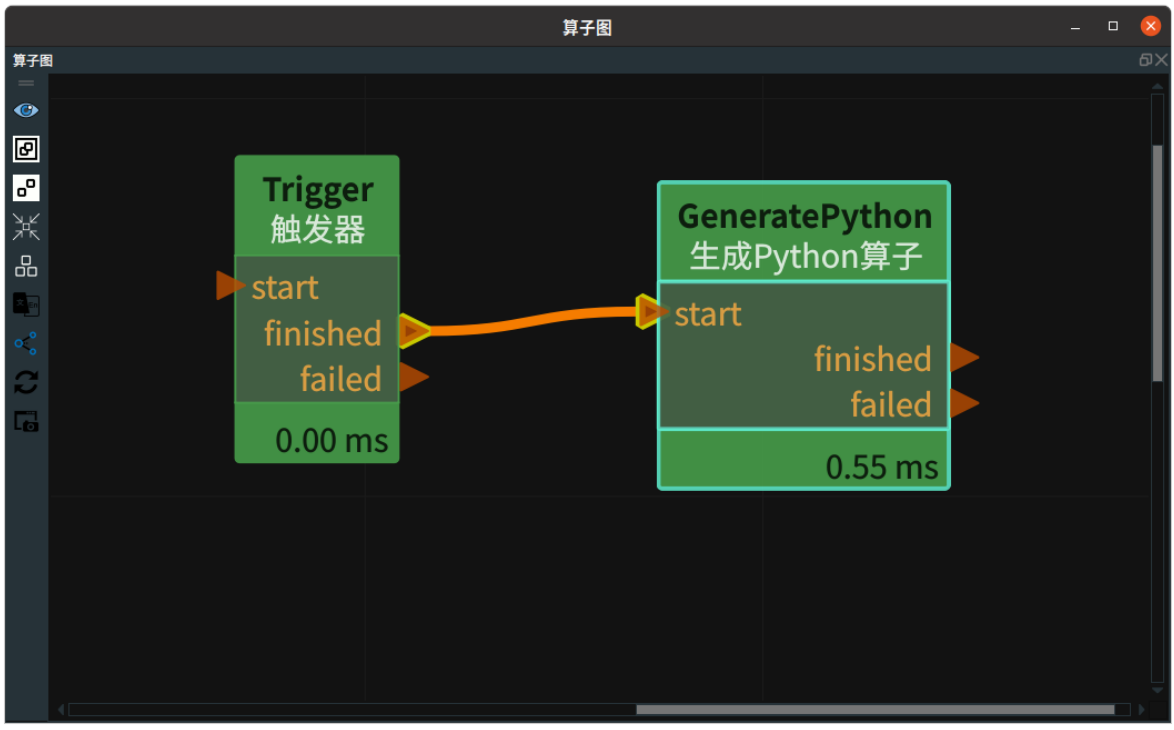
编写的 xml 文档输入内容详解如下：

- input 表示 python 脚本从 RVS 软件中接收到的数据输入
- output 表示 python 脚本返还给 RVS 软件的数据输出
- type=“ ” 表示数据类型（支持的数据类型包括 14 种，详见 3.2.4 部分 ）
- xml文档中每行内容中间的文本值，例如 Image ，属于数据变量名（注：变量名必须符合python以及C++语言关于变量名的定义规范，如不能包含空格以及特殊字符）

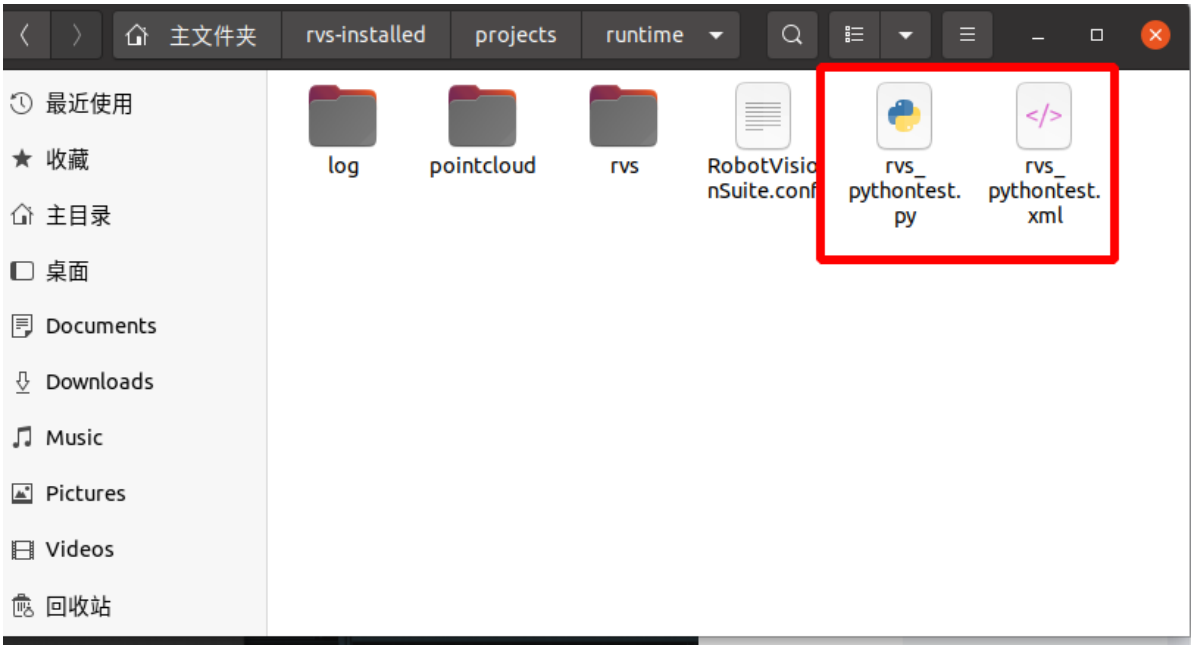
编写并保存该 xml 文档，该案例中将文档保存为“rvs_pythontest.xml”

3.1.2. 3.1.2 生成python脚本文件

打开 RVS，从算子库中选择算子 GeneratePython ，并更改算子属性 “python_config_file” 的数值为上述 xml 文档的文件路径，在该案例中，路径名为 “./rvs_pythontest.xml” 。添加 trigger 算子出发运行 GeneratePython 算子，在 RVS 中示例如下：



运行成功后，会自动在 xml 文档所在路径生成同名的 python 脚本文件，这里为 rvs_pythontest.py。



3.1.3. 3.1.3 代码解析

使用 python 脚本编辑器（比如 VSCode、PyCharm 等）打开 rvs_pyhontest.py，其内容如下所述：

```
# Note: this file is a template python file and it was generated from config_file
:example_data/python_text.xml.
###########################################
import RVS_LOG
# write log to RVS_log_file, for example:
# RVS_LOG.write("your own log string")
import numpy
# ToDo: import the moudle you need
import cv2

# The following class 'RVSPyClass' is the interface-class between your python_moudle and
RVS_python_node.
```

上述导入了 RVS_LOG 模块，这是一个内置的日志回传函数，通过 RVS_LOG 的 write 方法，可以将 string 信息回传给 RVS 程序，之后 RVS 程序会将该 string 高亮显示在 RVS 的日志栏，同时写入到本地日志文件。
同时在这个 python 模板文件中，还自动创建了接口类 RVSClass，在这个类中包含了 _ init _ 函数、inputData 函数、outputData 函数、process 函数和 re_ini 函数。

> 注意：接口类名和所有的函数类名都不可以进行修改。

```
class RVSPyClass:
```

所有的数据端口都在该类的 _ init _ 函数中进行了初始定义，Python文件中的_ init _ 函数中的初始内容仅仅作为一个显示效果，作用是告诉使用者对应的输入输出数值的格式，这些初始内容都是可以删除的。

```
  def __init__(self):
    # all io_para_name and it's type/size have been list as below.
    self.im0 = numpy.ndarray(shape=(0,0,0)) # dtype=numpy.uint8 or numpy.uint16
    self.im1 = numpy.ndarray(shape=(0,0,0)) # dtype=numpy.uint8
    ##########################################
    # ToDo: remove 'pass' and add your code for initial
    pass


  # Must not do any change of the func inputData as below.
```

在实际运行时，通过 inputData 函数，im0 会去接收 RVS 中传输的数据流。

```
  def inputData(self,input):
    self.im0 = input[0]
    pass

  # Must not do any change of the func outputData as below.
```

outputData 函数将计算结果返回到 im1端口，并通过这两个数据端口传输给下个算子。

```
  def outputData(self):
    return [ self.im1 ]
```

用户可以修改 process 函数和 re_ini 函数的主体内容，同时也可以在 Class 类中自定义用户 Function 函数被 process 和 re_ini 函数调用。

```
  def re_ini(self):
    ##########################################
    # ToDo: remove 'pass' and add your code
    pass
  def process(self):
    ##########################################
    # ToDo: remove 'pass' and add your code
    pass
  ##########################################
  # ToDo: add your own function
##############################################
# ToDo: add your own code
```

比如在这个案例中，我们可以将 process 函数的 pass 语句删除，修改为：

```
    RVS_LOG.write("start resize ...")
    self.im1 = cv2.resize(self.im0,(400,200),interpolation = cv2.INTER_AREA)
    RVS_LOG.write("finished!")
```

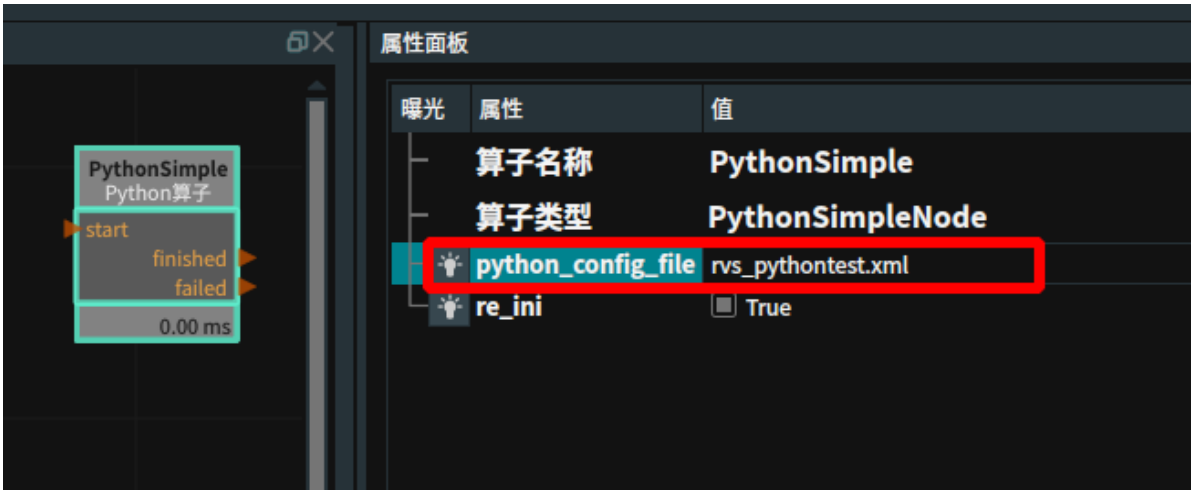
这样就实现了将一张测试图进行缩放处理的效果。

3.1.4. 3.1.4. 代码测试

我们已经完成了二次开发代码的编写，我们现在可以进入到 RVS 软件当中测试我们写的代码。

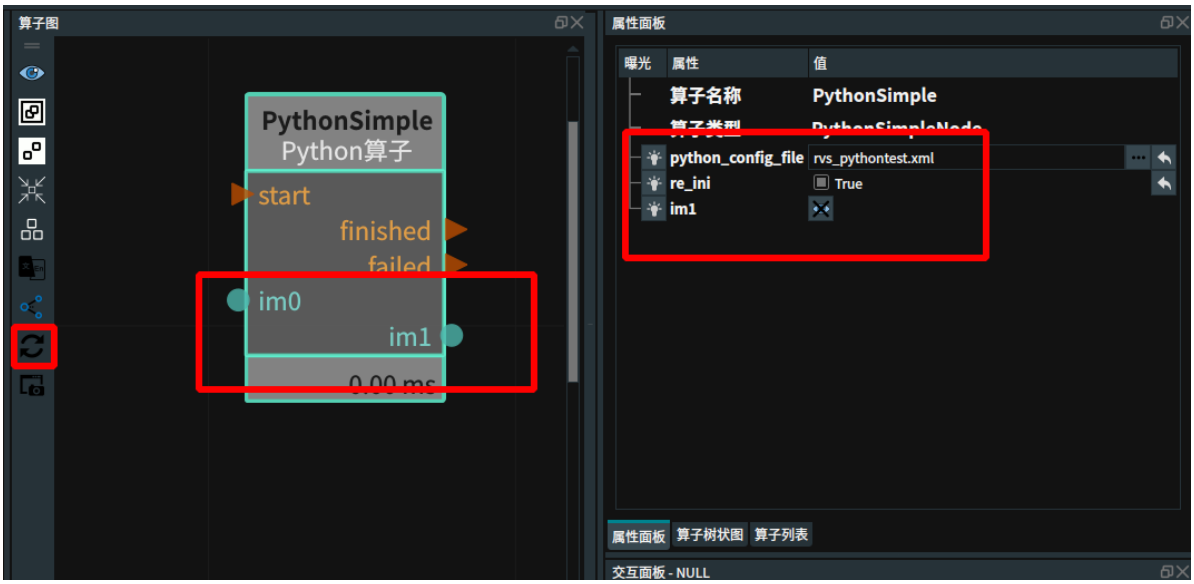
打开RVS，从算子库中选择 PythonSimple 算子，将属性 python_config_file 对应的数值改为 3.1.1 中编写的 xml 文件路径。

> 注意：只有将PythonSimple算子第一次从算子库中拖动到算子图时，才具有“更新属性 python_config_file 时自动更新其输入输出端口”的能力，如果后续想将该PythonSimple 对应的 xml 更新，需要将该算子触发执行一次以后才会更新其数据端口。

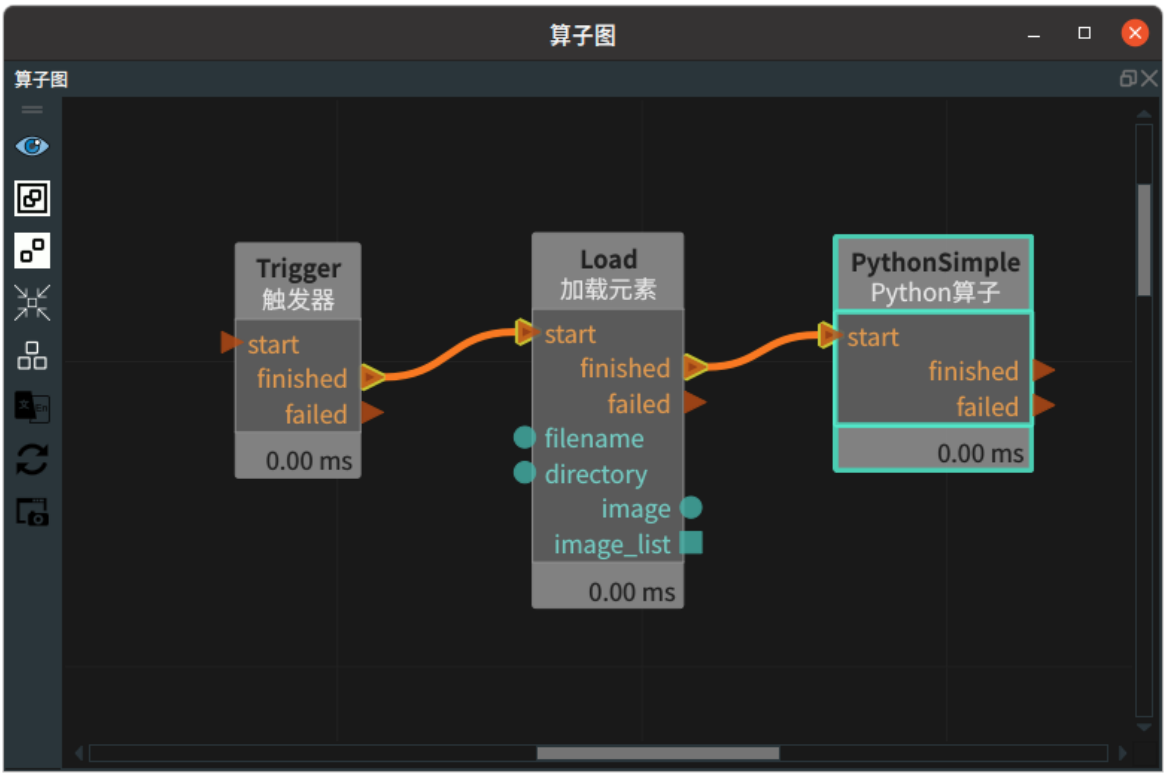


更改完成后，该算子会自动读取 xml 文件内容，同时自动更新该算子的属性栏，效果如下所示：（由于该案例中，我们总共添加了两个数据输出端口，所以下属属性栏中会自动出现这两个数据端口的可视化属性）

另外，我们需要手动点击算子图区域左侧的刷新按钮，进行算子的端口刷新，刷新后的效果如下：



后面的使用就同其他 RVS 算子一样，给该算子输入数据，并给定触发信号，算子内部会自动调用一次“rvs_pythontest.xml” 对应的同名名目录下的同名 python文件（即 rvs_pythontest.py）。



每一次RVS软件中的PythonSimple算子被触发后，都会在python文件内部按照下述顺序执行一　次：

a)RVSPyClass 的初始化函数 _ init _

b)RVSPyClass 的数据输入函数 inputData（传入的函数内参即 RVS 软件中输入给 PythonSimple 算子的实际数据）

c)RVSPyClass 的可选执行函数 re_ini（当 RVS 软件中 PythonSimple 算子的 re_ini 属性勾选以后会执行该函数，执行后该 re_ini 属性会自动恢复非勾选状态），如下图所示：

d)RVSPyClass的数据处理函数process

e)RVSPyCLass的数据输出outputData

3.2. 3.2. 补充说明

本章内容为补充部分，主要讲述 PythonThread 算子的使用、常见的错误操作，以及3.1.1中提到的全部的数据类型定义格式，由于 PythonThread 与PythonSimple 使用的流程一致，这里只阐述一下 PythonThread 算子的意义和使用过程中的注意事项。

3.2.1. 3.2.1 PythonThread算子

1. 除了PythonSimple算子，RVS中还提供了PythonThread算子。其使用方式同PythonSimple算子相同，不过PythonThread算子是额外开辟个新线程来运行python算子。由于PythonThread算子从属于Thread类算子，所以多个并列的PythonThread算子可以同时运行；而PythonSimple算子从属于普通算子，运行会占用主线程，所以多个PythonSimple算子即便并行连接，实际执行时还是先后依次运行。
2. 无论是PythonSimple算子还是PythonThread算子，都无法实现python文件的在线更新；如果您在运行了RVS以后，重新更改了自己的python算子文件，对应的PythonSimple/PythonThread均无法重新捕获该更改，必须重启RVS软件或者删掉该算子并重新加载新的Python算子，才能实现python算子文件的重新加载。另外，为了解决这个“需要临时更新python文件中某些参数的问题”，建议您将这些需要更新的参数作为String类型的数据输入，通过合理使用re_ini函数来达到这种更新效果。

3.2.2. 3.2.2 PythonThread算子

1. RVS中以ThreadNode结尾的算子都是线程类算子，它们的运行时间较长。（下文描述，基于普通算子的运行时间较短的情况）
2. thread算子在被触发之后，会极短暂的占用主线程，然后会进入到分线程执行功能函数，同时释放主线程，所以此时RVS可以继续触发执行下一个算子，这也是多个并列的thread算子可以近似同时运行的原因。
3. 所有的算子被拖动到RVS的算子图区域时会被自动赋予一个算子名，RVS底层会记录这些算子名的创建顺序作为算子顺序。
4. 在某一个并行连接结构中，比如算子A触发算子B，算子B触发算子C；且算子A触发算子D，算子D触发算子E。无论B或者D两者中是否含有thread算子，RVS的执行都是等两者全部运行完成，才会去考虑执行C或者E（此时C/E虽然不是源自同一个起点，但是执行的先后顺序也是按照并行连接来考虑）。
5. 当B以及D全部都是thread算子时，因为两个并行的thread可以同时运行，所以此时能有效节省运行时间。比如B运行3秒，D运行5秒，则两者的并行连接只需要运行5s就可以执行到C/E算子，而串形连接需要运行8s才能执行到C/E。
6. 如果将一个thread算子同一个普通算子并行连接，则理论上只能节省该普通算子的运行时间，意义不大。RVS不建议进行这种连接方式，而且底层做了处理，即便这样连接了，也是先触发普通算子，执行完成后再执行thread算子，之后再考虑执行C/E。
7. 如果多个普通算子同一个thread算子并行连接，第一个运行的一定是普通算子，执行完成后执行的第二个算子是否是thread算子则需要根据算子顺序排序。如果第二个算子是thread算子，则由于thread算子的特性，在该算子被触发之后，不用等到该thread算子执行完成，就会立刻触发执行第三个算子；如果第二个算子是普通算子，则必须等该算子执行完成后才可以执行第三个算子，依此类推。
8. 多个普通算子并行连接，按照算子顺序，依次执行。

3.2.3. 3.2.3 常见的错误操作

- 更改了python模板文件中接口类的类名 RVSPyclass、函数名 inputData、outputData、re_ini、process。或者不恰当的修改了 inputData、outputData 的函数体。
- 在 process 函数中写了大量代码，但是最后忘记将需要返回的实际变量赋值给 outputData 函数中的返回变量
- 修改了 python 模板文件的文件名，或者将该模板文件移动到其他位置。

> 注意：python 模板文件可以移动，但是一定要保证同 python_xml 文件一起移动，保证这个两个文件在同一个目录即可。

- 在任何需要输入文件路径的地方，如果需要使用/或者\，必须只能使用/（该说明仅限定 linux 版本，在 windows 版本待定）
- outputData 函数中返回的数据格式不合法

3.2.4. 3.2.4 数据类型格式定义

数据类型	格式定义
String	“或””
StringList	String构成的list
Pose或JointArray	6个float数据构成的list
PoseList或者JointArrayList	Pose或者JointArray构成的list
Image	Numpy的Ndarray对象，输入数据的数据类型可以是numpy.uint8或者numpy.uint16（对应图漾相机的深度图），uint8格式下支持灰度图和3通道彩色图,numpy.uint16格式下仅支持灰度图。输出数据只能是numpy.uint8，可以是单通道灰度图或3通道彩色图。
ImageList	由Image构成的list
Cube	9个float数据构成的list
CubeList	由Cube构成的list
CubeList	由Cube构成的list
ImagePoints	形如[x1 y1 x2 y2...]的2d像素点列表list
ImagePointsList	由ImagePoints构成的list
RotatedRect	5个float数据构成的list，即[center_x,center_y,width,height,angle]
RotatedRectList	由RotatedRect构成的list

4. 附录A 支持资源

本章描述图漾为用户提供的支持资源，包括相关的技术与文档支持。

4.1. A.1. 技术支持

我们对所有购买产品的用户提供以下两种官方技术支持方式：

1. 公众号支持：请关注官方微信公众号-图漾科技，并发表您的看法和意见
2. 邮件支持：如需其他帮助, 请发邮件至 rvs-support@percipio.xyz。

4.2. A.2. 文档支持

您可能需要了解以下文档内容：

• 图漾产品选型指南

本文档介绍图漾所有产品型号及其技术指标。

• Percipio SDK 入门指南

本文档介绍 SDK 的安装、编译和使用。

4.3. A.3. 各版本下载链接

我们对不同版本提供了以下下载链接：

Ubuntu 20.04:

• Full版本下载链接（包含全部功能模块）

• CPU版本下载链接（不带GPU相关功能）

windows版本:

• Full版本下载链接（包含全部功能模块）

• CPU版本下载链接（不带GPU相关功能）

> 说明:如需下载其他文档，请访问如下链接：downloadcenter – 图漾科技丨3D相机 (percipio.xyz)。