

C#调用 OpenVINO™ 部署 AI 模型项目开发项目，简称 OpenVinoSharp，这是一个示例项目，该项目实现在 C#编程语言下调用 Intel 推出的 OpenVINO™ 工具套件，进行深度学习等 AI 项目在 C#框架下的部署。该项目由 C++语言编写 OpenVINO™ API 接口，并在 C#语言下实现应用。

C#调用 OpenVINO™ 部署 AI 模型项目

颜国进

目录

第 1 章 C#调用 OpenVINO™ 部署 AI 模型项目开发	1
1.1 项目概述.....	1
1.1.1 项目介绍.....	1
1.1.2 OpenVINO™	1
1.1.3 项目方案.....	1
1.1.4 安装方式.....	2
1.2 软件安装.....	2
1.2.1 Microsoft Visual Studio 2022 安装	2
1.2.2 OpenVINO™ 安装	3
1.2.3 OpenCV 安装	5
1.3 OpenVINO™ 推理模型与测试数据集.....	6
1.3.1 模型种类与下载方式	6
1.3.2 PaddleDetection 模型	6
1.3.3 PaddleClas 模型	6
1.4 创建 OpenVINO™ 方法 C++动态链接库.....	6
1.4.1 新建解决方案以及项目文件	6
1.4.2 配置 C++项目属性	7
1.4.3 编写 C++代码	10
1.4.4 编写模块定义文件	14
1.4.5 生成 dll 文件.....	14
1.5 C#构建 Core 类.....	15
1.5.1 新建 C#类库	15
1.5.2 引入 dll 文件中的方法.....	15
1.5.3 创建 Core 类.....	15
1.5.4 编译 Core 类库	16
1.6 C#实现 OpenVINO™ 方法的调用.....	17
1.6.1 新建 C#项目	17
1.6.2 添加 OpenCVsharp.....	17
1.6.3 添加项目引用	17
1.6.4 编写代码测试花卉分类模型	18
1.6.5 编写代码测试车辆识别模型	20
1.7 程序时间分析	21
1.8 项目总结.....	22

第1章 C#调用 OpenVINO™ 部署 AI 模型项目开发

1.1 项目概述

1.1.1 项目介绍

C#调用 OpenVINO™ 部署 AI 模型项目开发项目，简称 OpenVinoSharp，这是一个示例项目，该项目实现在 C#编程语言下调用 Intel 推出的 OpenVINO™ 工具套件，进行深度学习等 AI 项目在 C#框架下的部署。该项目由 C++语言编写 OpenVINO™ API 接口，并在 C#语言下实现应用。

项目可以实现在 C#编程语言下调用 Intel 推出的 OpenVINO™ 工具套件，进行深度学习等 AI 项目在 C#框架下的部署，目前可以支持的 AI 模型格式：

- Paddlepaddle 飞桨模型 (.pdmodel)
- ONNX 开放式神经网络交换模型 (.onnx)
- IR 模型 (.xml, .bin)

目前该项目针对 Paddlepaddle 飞桨现有模型进行了测试，主要有：

- PaddleClas 飞桨图像识别套件
- PaddleDetection 目标检测模型套件

1.1.2 OpenVINO™

OpenVINO™ 是英特尔基于自身现有的硬件平台开发的一种可以加快高性能计算机视觉和深度学习视觉应用开发速度工具套件，支持各种英特尔平台的硬件加速器上进行深度学习，并且允许直接异构执行。支持在 Windows 与 Linux 系统，官方支持编程语言为 Python 与 C++语言。

OpenVINO™ 工具套件 2022.1 版于 2022 年 3 月 22 日正式发布，根据官宣《OpenVINO™ 迎来迄今为止最重大更新，2022.1 新特性抢先看》，OpenVINO™ 2022.1 将是迄今为止最大变化的版本。从开发者的角度来看，对于提升开发效率或运行效率有用的特性有：

- 提供预处理 API 函数
- ONNX 前端 API
- AUTO 设备插件
- 支持直接读入飞桨模型

该项目开发环境为 OpenVINO™ 2022.1 最新版本，因此使用者需在使用时将自己电脑上的 OpenVINO™ 版本升级到 2022.1 版，不然会有较多的问题。

1.1.3 项目方案

该项目主要通过调用 dll 文件方式实现。通过 C++调用 OpenVINO™，编写模型推理接口，将我们所用到的推理方法在 C++中实现，并将其生成 dll 文件，在 C#调用 dll 文件，重写 dll 文件接口，并重新组建 Core 类，用于在 C#中进行模型的推理，其方案如图 1-1 所示。

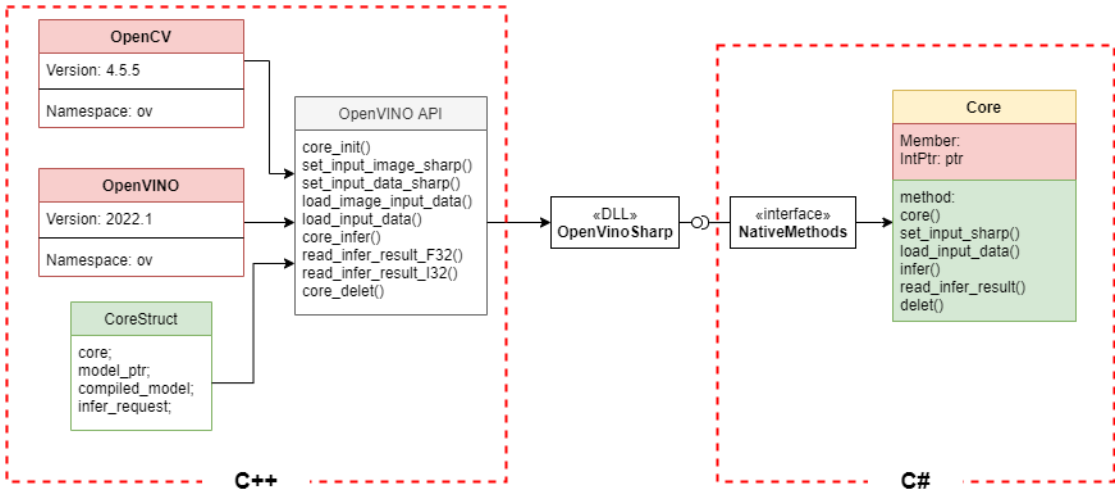


图 1-1 项目解决方案

1.1.4 安装方式

该项目所有文件已经上传到 Github 和 Gitee 远程代码仓，大家可以通过 Gi 在本地进行克隆。

- 系统平台：
Windows
- 软件要求：
Visual Studio 2022 / 2019 / 2017
OpenCV 4.5.5
OpenVINO 2022.1
- 安装方式
在 Github 上克隆下载：

```
git clone https://github.com/guojin-yan/OpenVinoSharp.git
```

在 Gitee 上克隆下载：

```
git clone https://gitee.com/guojin-yan/OpenVinoSharp.git
```

1.2 软件安装

1.2.1 Microsoft Visual Studio 2022 安装

Microsoft Visual Studio（简称 VS）是美国微软公司的开发工具包系列产品。VS 是一个基本完整的开发工具集，它包括了整个软件生命周期所需要的大部分工具，如 UML 工具、代码管控工具、集成开发环境(IDE)等等。其支持 C、C++、C#、F#、J#等多门编程语言。

本次项目所使用的编程语言为 C++与 C#两门编程语言，在 VS 中完全可以实现，可选择安装版本 VS2017、VS2019 或 VS2022 版本。对于 VS 不同版本的选择，该项目不做较多要求，就笔者使用来说，VS2017 版本推出时间较久，不建议使用，其一些编程语言规范有一些变动，对于该项目所提供的范例可能会有部分不兼容；VS2019 和 VS2022 版本相对更新，是由起来差异不大，建议选择这两个版本，并且新版 OpenVINO™ 支持 VS2022 版本

Cmake。

笔者电脑安装的为 Microsoft Visual Studio Community 2022 版本，其安装包可由 VS 官网直接下载，下载时选择社区版，按照一般安装步骤进行安装即可。在安装中，工作负荷的选择图 1-2 所示。



图 1-2 Visual Studio 2022 安装负荷

安装完成后，可以参照网上相关教程，进行学习 VS 的使用。

1.2.2 OpenVINO™ 安装

该项目所使用的 OpenVINO™ 版本为 2022.1 版本，是 Intel 公司在 2022 年第一季度发布的最新版本。该版本基于之前版本有了较大变动，不在默认包含 OpenCV 工具；其次，对代码做了更进一步的优化，使得代码在使用时更加灵活。其具体安装方式如下：

(1) OpenVINO™ 下载

访问 OpenVINO™ 官网<www.openvino.ai>，点击 Free Download->，进入到软件下载页面，按照图 1-3 所示，选择安装内容

Choose a Preferred Package

You can customize the selections to fit your needs.

Environment

Dev Tools
Best option to develop and optimize deep-learning models

Runtime
You already have a model and want to run inference on it

Operating System

Windows

macOS

Linux

OpenVINO™ Version

2022.1 (recommended)
Latest standard release

2021.4.2 LTS
Latest LTS release

2020.3 LTS
Previous LTS release

Language

Python
Included by default, and cannot be unselected

C++

Distribution

Offline Installer
Recommended option

Online Installer

GitHub
Source

Gitee
Source

Docker

[Learn more about distribution options](#)
[Try OpenVINO on Intel® DevCloud](#)

Download Intel® Distribution of OpenVINO™ Toolkit

[Install instructions](#)
[Get started guide](#)
[OpenVINO Notebooks](#)

Download

图 1- 3 OpenVINO™ 安装选择

选择完成后，点击 **Download**，下载安装包。

(2) 安装软件

在安装包下载完成后，直接安装软件即可，全程默认软件安装，不需要做任何修改。

(3) 配置环境变量

在软件安装完成后，需要配置相关环境变量，防止每次使用都需要运行虚拟环境。右击我的电脑，进入属性设置，选择高级系统设置进入系统属性，点击环境变量，进入到环境变量设置，编辑系统变量下的 **Path** 变量，增加以下地址变量：

```
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\bin\intel64\Debug  
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\bin\intel64\Release
```

4 / 23

```
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\3rdparty\tbb\bin
```

该路径为默认安装路径,如果不更改安装地址直接使用上方路径即可,如果进行了修改,请将< C:\Program Files (x86)\Intel\>替换为更改的安装路径。

1.2.3 OpenCV 安装

由于最新版的 OpenVINO™ 2022.1 版本不在默认附带 OpenCV 工具,所以我们需要额外安装 OpenCV 工具。

(1) 下载并安装 OpenCV

访问 OpenCV 官网 <<https://opencv.org/>> , 选择 Library 下的 Releases, 进入到下载页面, 或直接访问<<https://opencv.org/releases/>> 进入下载页面。

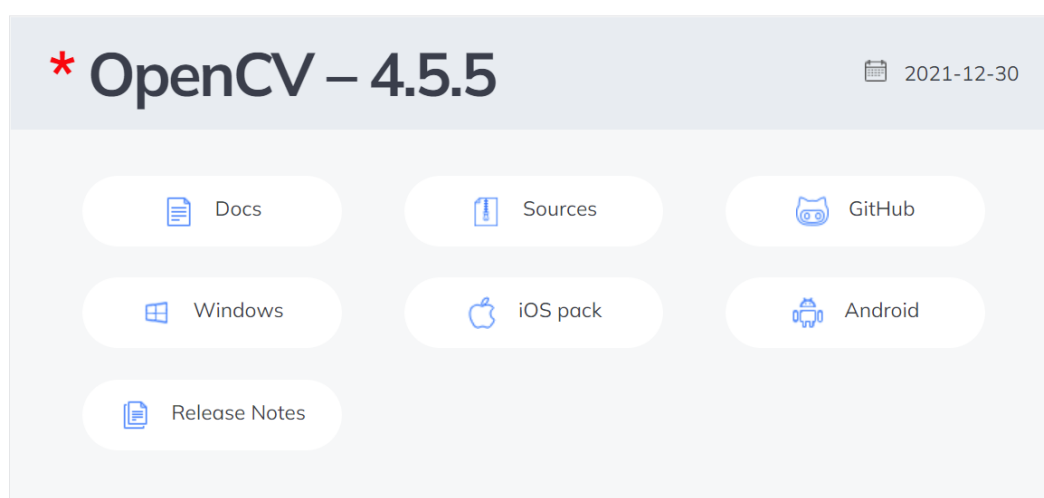


图 1- 4 OpenCV-4.5.5 版本页面

根据负载使用情况,选择 Windows 版本,如图 1- 4 所示,跳转页面后,下载文件名为: opencv-4.5.5-vc14_vc15.exe。下载完成后,直接双击打开安装文件,安装完成后,打开安装文件夹,该文件夹下 build、sources 文件夹以及 LICENSE 相关文件,我们所使用的文件在 build 文件夹中。

(2) 配置 Path 环境变量

右击我的电脑,进入属性设置,选择高级系统设置进入系统属性,点击环境变量,进入到环境变量设置,编辑系统变量下的 Path 变量,增加以下地址变量:

```
E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\bin  
E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\lib  
E:\OpenCV Source\opencv-4.5.5\build\include  
E:\OpenCV Source\opencv-4.5.5\build\include\opencv2
```

其中<E:\OpenCV Source\opencv-4.5.5>为本机安装 OpenCV 安装路径。安装完成后,如想在 VS 中使用,还需要再 VS 编辑器中设置相关配置,具体配置会在下文中说明。

1.3 OpenVINO™ 推理模型与测试数据集

1.3.1 模型种类与下载方式

为了测试该项目，我们提供并整合了训练好的 Paddlepaddle 模型，主要针对 PaddleClas 以及 PaddleDetection 现有的模型，提供了 PaddleClas 下的花卉分类模型以及 PaddleDetection 中的 Vehicle Detection 模型，并针对该模型，提供了 pdmodel、onnx 以及 IR 格式。

该项目所使用的测试模型以及数据集，均可以在本文下的 gitee 上下载，下载链接为：

1.3.2 PaddleDetection 模型

PaddleDetection 为飞桨 PaddlePaddle 的端到端目标检测套件，提供多种主流目标检测、实例分割、跟踪、关键点检测算法，配置化的网络模块组件、数据增强策略、损失函数等。该项目在此处主要使用的为 PaddleDetection 应用中的目标检测功能，使用的网络为 YOLOv3 网络，表 1-1 给出了 YOLOv3 网络输出与输入的相关信息。

表 1-1 YOLOv3 模型输入与输出节点信息

节点类型	节点名	节点形状	数据类型	备注
Input	image	[None, 3, H, W]	float32	输入网络的图像
	im_shape	[None, 2]	float32	图像经过 resize 后的大小
	scale_factor	[None, 2]	float32	输入图像比真实图像大小
Output	multiclass_nms3_0.tmp_0	[Num, 6]	float32	识别结果
	multiclass_nms3_0.tmp_2	[None]	int32	识别结果数量

注：None 表示 batch 维度，H、W 分别为图片的高和宽，Num 表示识别结果的数量。

本次测试使用的为 PaddleDetection 中 Vehicle Detection 模型，我们可以在 PaddleDetection gitee 上下载。该模型输入图片要求为 $3 \times 608 \times 608$ 大小，输出为预测框信息，其信息组成 $[class_id, score, x1, y1, x2, y2]$ ，分别代表分类编号、分类得分以及预测框对角顶点坐标。

1.3.3 PaddleClas 模型

飞桨图像识别套件 PaddleClas 是飞桨为工业界和学术界提供的的一个图像识别任务的工具集，该模型经过数据集训练，可以识别多种物品。在该项目中，我们使用 flower 数据集，使用 ResNet50 网络训练识别 102 种花卉，关于该模型的输入与输出节点信息如所示

表 1-2 ResNet50 模型输入与输出节点信息

节点类型	节点名	节点形状	数据类型	备注
Input	x	[None, 3, H, W]	float32	输入网络的图像
Output	softmax_1.tmp_0	[None, Class]	float32	各个识别结果概率

注：None 表示 batch 维度，H、W 分别为图片的高和宽，Class 表示分类数量。

花卉训练模型要求图片输入为 $3 \times 224 \times 224$ 大小，输出结果为 102 中预测结果概率。

1.4 创建 OpenVINO™ 方法 C++动态链接库

1.4.1 新建解决方案以及项目文件

打开 vs2022，首先新建一个 C++空项目文件，并将同时新建一个解决方案命名为：OpenVinoSharp，用于存放后续其他项目文件。将 C++项目命名为：CppOpenVinoAPI。

进入项目后，右击源文件，选择添加→新建项→C++文件(cpp)，进行的文件的添加。具体操作如图 1-5 所示。

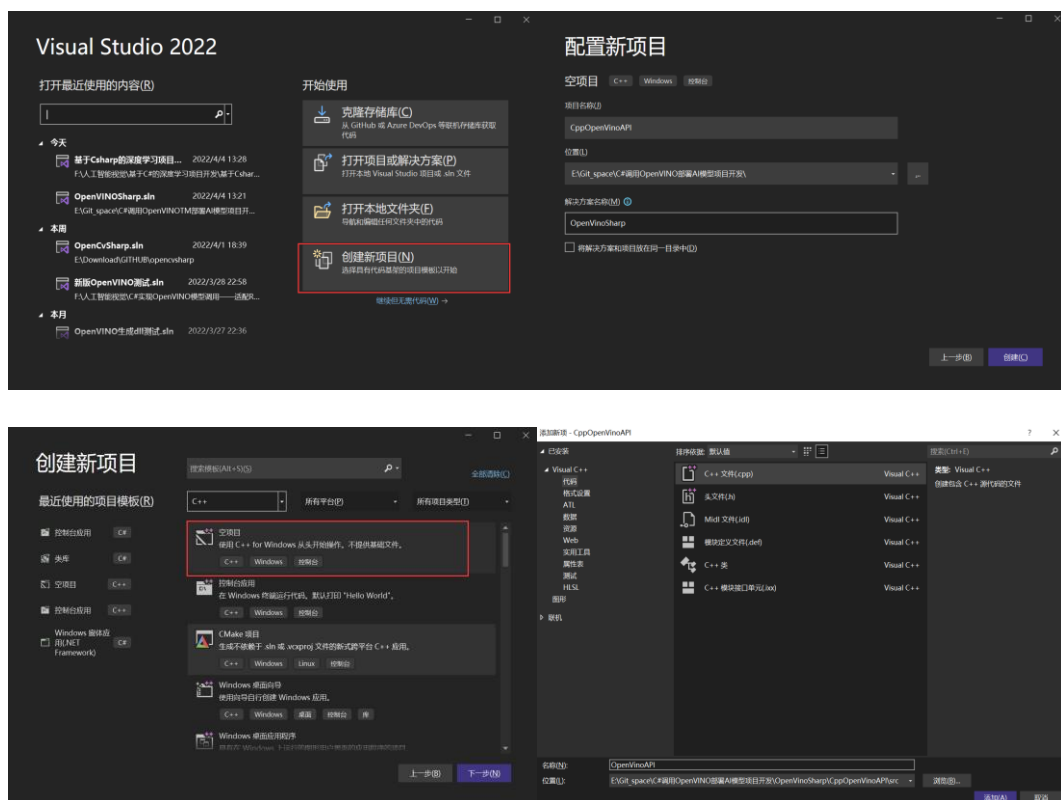


图 1-5 新建项目解决方案及 C++项目

本次我们需要添加 OpenVinoAPIcpp 以及 Source.def 两个文件，如图 1-6 所示。

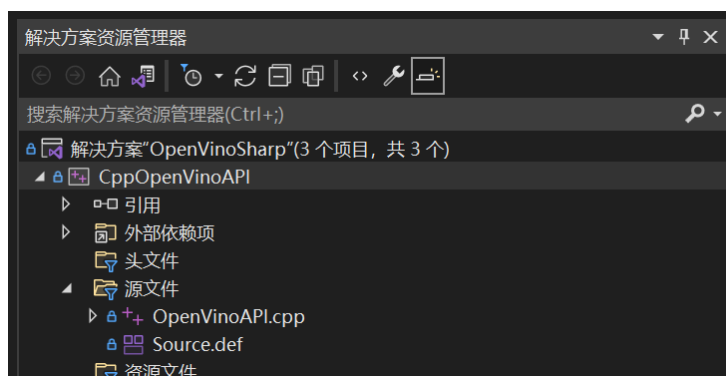


图 1-6 CppOpenVinoAPI 函数方法所需文件

1.4.2 配置 C++项目属性

右击项目，点击属性，进入到属性设置，此处需要设置项目的配置类型包含目录、库目录以及附加依赖项，本次项目选择 Release 模式下运行，因此以 Release 情况进行配置。

(1) 设置配置与平台

进入属性设置后，在最上面，将配置改为 Release，平台改为 x64。具体操作如图 1-7 所

示。

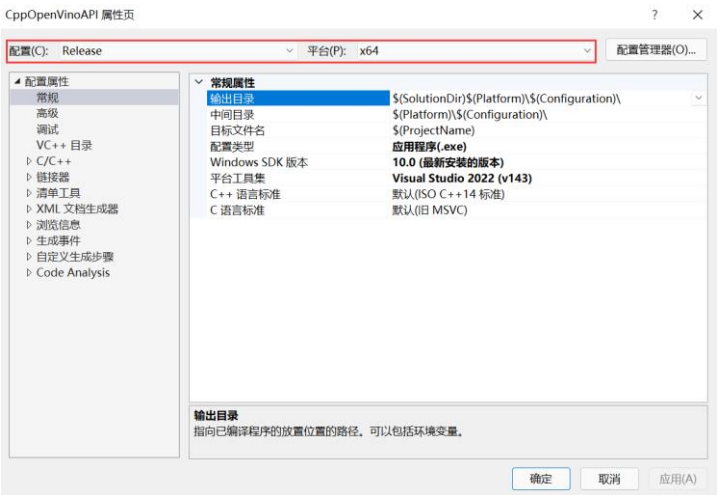


图 1-7 C++项目属性配置与平台设置

(2) 设置常规属性

常规设置下，点击输出目录，将输出位置设置为<dll>，即将生成文件放置在项目文件夹下的 dll 文件夹下；其次将目标文件名修改为：OpenVinoSharp；最后将配置类型改为：动态库(.dll)，让其生成 dll 文件。具体操作如图 1-8 所示。

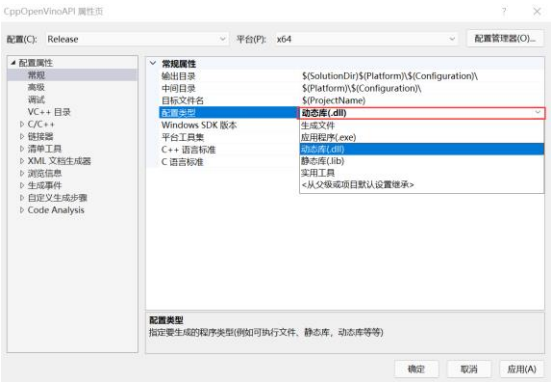
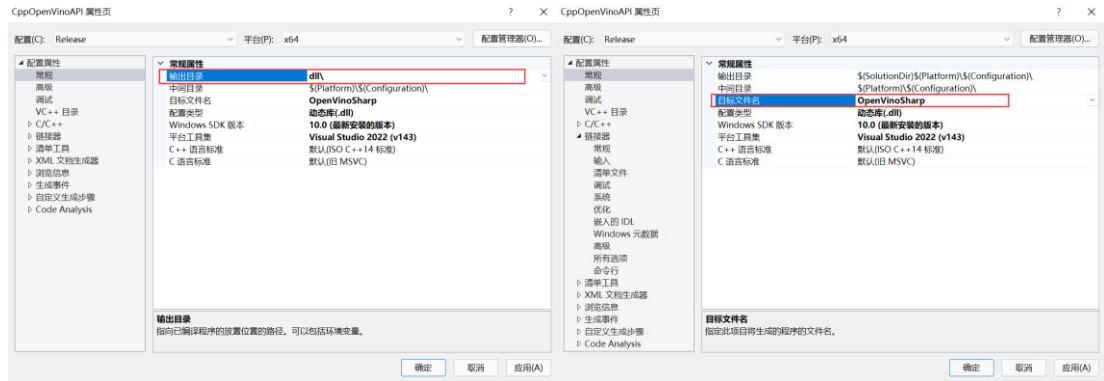


图 1-8 C++项目常规属性设置

(3) 设置包含目录

点击 VC++ 目录，然后点击包含目录，进行编辑，在弹出的新页面中，添加以下路径：

```
E:\OpenCV Source\opencv-4.5.5\build\include
E:\OpenCV Source\opencv-4.5.5\build\include\opencv2
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\include
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\include\ie
```

其中路径<E:\OpenCV Source\opencv-4.5.5>为 OpenCV 的安装路径，<C:\Program Files (x86)\Intel\openvino_2022.1.0.643>为 OpenVINO 安装路径，个人根据自己安装的位置及版本确定。操作过程如图 1-9 错误!未找到引用源。所示。

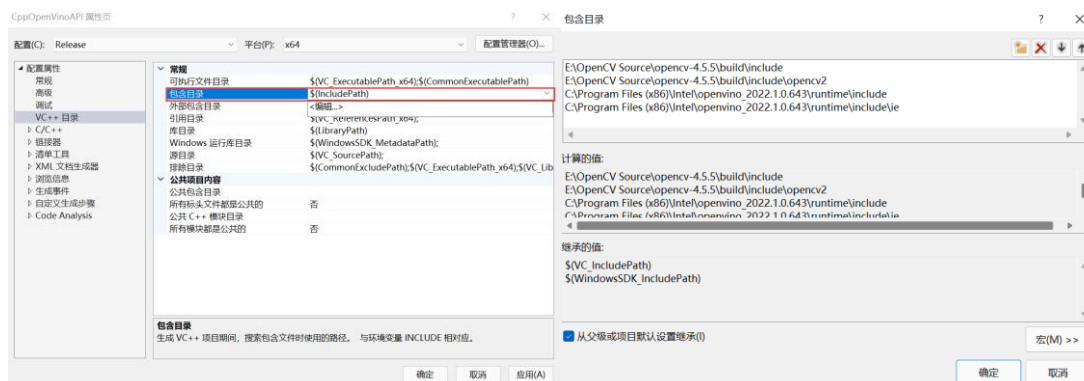


图 1-9 C++项目属性库目录设置

(4) 设置库目录

同样的方式，在 VC++ 目录下，点击下方的库目录，点击编辑，在弹出来的页面中增加以下路径：

```
E:\OpenCV Source\opencv-4.5.5\build\x64\vc15\lib
C:\Program Files (x86)\Intel\openvino_2022.1.0.643\runtime\lib\intel64\Release
```

如果时配置 Debug 模式，则需要将 Release 文件路径改为 Debug 即可。

(5) 设置附加依赖项

点击展开链接器，点击输入，在附加依赖项中点击编辑，在弹出来的新的页面，添加以下文件名：

```
opencv_world455.lib
openvino.lib
```

具体操作步骤参考如图 1-10 所示。新版 OpenCV 与 OpenVINO™ 都将依赖库文件合成了一个文件中，这极大地简化了使用，如果使用老版本的，需要把所有的.lib 文件放置在此处即可。

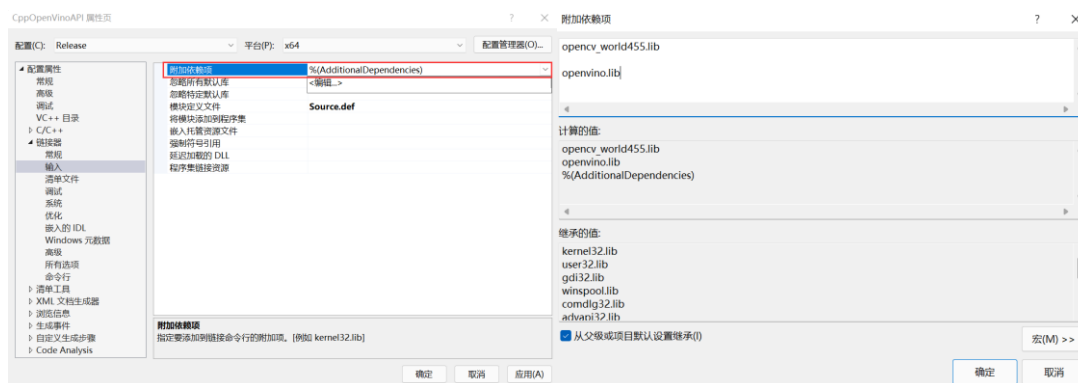


图 1- 10 C++项目属性附加依赖项设置

1.4.3 编写 C++代码

(1) 推理引擎结构体

Core 是 OpenVINO™ 工具套件里的推理核心类，该类下包含多个方法，可用于创建推理中所使用的其他类。在此处，需要在各个方法中传递的仅仅是所使用的几个变量，因此选择构建一个推理引擎结构体，用于存放各个变量。

```
// @brief 推理核心结构体
typedef struct openvino_core {
    ov::Core core; // core 对象

    std::shared_ptr<ov::Model> model_ptr; // 读取模型指针

    ov::CompiledModel compiled_model; // 模型加载到设备对象

    ov::InferRequest infer_request; // 推理请求对象
} CoreStruct;
```

其中 Core 是 OpenVINOTM 工具套件里的推理机核心，该模块只需要初始化；shared_ptr<ov::Model>是读取本地模型的方法，新版更新后，该方法发生了较大改动，可支持读取 Paddlepaddle 飞桨模型、onnx 模型以及 IR 模型；CompiledModel 指的是一个已编译的模型类，其主要是将读取的本地模型应用多个优化转换，然后映射到计算内核，由所指定的设备编译模型；InferRequest 是一个推理请求类，在推理中主要用于对推理过程的操作。

(2) 接口方法规划

经典的 OpenVINO™ 进行模型推理，一般需要八个步骤，主要是：初始化 Core 对象、读取本地推理模型、配置模型输入&输出、载入模型到执行硬件、创建推理请求、准备输入数据、执行推理计算以及处理推理计算结果。我们根据原有的八个步骤，对步骤进行重新整合，并根据推理步骤，调整方法接口。

对于方法接口，主要设置为：推理初始化、配置输入数据形状、配置输入数据、模型推理、读取推理结果数据以及删除内存地址六个大类，其中配置输入数据形状要细分为配置图片数据形状以及普通数据形状，配置输入数据要细分为配置图片输入数据与配置普通数据输入，读取推理结果数据细分为读取 float 数据和 int 数据，因此，总共有 6 类方法接口，9 个方法接口。

(3) 初始化推理模型

OpenVINO™ 推理引擎结构体是联系各个方法的桥梁，后续所有操作都是在推理引擎结构体中的变量上操作的，为了实现数据在各个方法之间的传输，因此在创建推理引擎结构体

时，采用的是创建结构体指针，并将创建的结构体地址作为函数返回值返回。推理初始化接口主要整合了原有推理的初始化 Core 对象、读取本地推理模型、载入模型到执行硬件和创建推理请求步骤，并将这些步骤所创建的变量放在推理引擎结构体中。

初始化推理模型接口方法为：

```
extern "C" __declspec(dllexport) void* __stdcall core_init(const wchar_t* model_file_wchar, const wchar_t* device_name_wchar);
```

该方法返回值为 CoreStruct 结构体指针，其中 model_file_wchar 为推理模型本地地址字符串指针，device_name_wchar 为模型运行设备名指针，在后面使用上述变量时，需要将其转换为 string 字符串，利用 wchar_to_string() 方法可以实现将其转换为字符串格式：

```
std::string model_file_path = wchar_to_string(model_file_wchar);
std::string device_name = wchar_to_string(device_name_wchar);
```

模型初始化功能主要包括：初始化推理引擎结构体和对结构体里面定义的其他变量进行赋值操作，其主要是利用 InferEngineStruct 中创建的 Core 类中的方法，对各个变量进行初始化操作：

```
CoreStruct* p = new CoreStruct(); // 创建推理引擎指针
p->model_ptr = p->core.read_model(model_file_path); // 读取推理模型
p->compiled_model = p->core.compile_model(p->model_ptr, "CPU"); // 将模型加载到设备
p->infer_request = p->compiled_model.create_infer_request(); // 创建推理请求
```

（4）配置输入数据形状

在新版 OpenVINO™ 2022.1 中，新增了对 Paddlepaddle 模型以及 onnx 模型的支持，Paddlepaddle 模型不支持指定默认 batch 通道数量，因此需要在模型使用时指定其输入；其次，对于 onnx 模型，也可以在转化时不指定固定形状，因此在配置输入数据前，需要配置输入节点数据形状。其方法接口为：

```
extern "C" __declspec(dllexport) void* __stdcall set_input_image_shape(void* core_ptr, const wchar_t* input_node_name_wchar, size_t * input_size);
extern "C" __declspec(dllexport) void* __stdcall set_input_data_shape(void* core_ptr, const wchar_t* input_node_name_wchar, size_t * input_size);
```

由于需要配置图片数据输入形状与普通数据的输入形状，在此处设置了两个接口，分别设置两种不同输入的形状。该方法返回值是 CoreStruct 结构体指针，但该指针所对应的数据中已经包含了对输入形状的设置。第一个输入参数 core_ptr 是 CoreStruct 指针，在当前方法中，我们要读取该指针，并将其转换为 CoreStruct 类型：

```
CoreStruct* p = (CoreStruct*)core_ptr;
```

input_node_name_wchar 为待设置网络节点名，input_size 为形状数据数组，对图片数据，需要设置 [batch, dim, height, width] 四个维度大小，所以 input_size 数组传入 4 个数据，其设置在形状主要使用 Tensor 类下的 set_shape() 方法：

```
std::string input_node_name = wchar_to_string(input_node_name_wchar); // 将节点名转为 string 类型
ov::Tensor input_image_tensor = p->infer_request.get_tensor(input_node_name); // 读取指定节点 Tensor
input_image_tensor.set_shape({ input_size[0],input_size[1],input_size[2],input_size[3] }); // 设置节点数据形状
```

（5）配置输入数据

在新版 OpenVINO™ 中，Tensor 类的 T* data() 方法，其返回值为当前节点 Tensor 的数据内存地址，通过填充 Tensor 的数据内存，实现推理数据的输入。对于图片数据，其最终也

是将其转为一维数据进行输入，不过为方便使用，此处提供了配置图片数据和普通数据的接口，对于输入为图片的方法接口：

```
extern "C" __declspec(dllexport) void* __stdcall load_image_input_data(void* core_ptr, const wchar_t*
input_node_name_wchar, uchar * image_data, size_t image_size);
```

该方法返回值是 CoreStruct 结构体指针，但该指针所对应的数据中已经包含了加载的图片数据。第一个输入参数 core_ptr 是 CoreStruct 指针，在当前方法中，我们要读取该指针，并将其转换为 CoreStruct 类型；第二个输入参数 input_node_name_wchar 为待填充节点名，先将其转为 string 字符串：

```
std::string input_node_name = wchar_to_string(input_node_name_wchar);
```

在该项目中，我们主要使用的是以图片作为模型输入的推理网络，模型主要的输入为图片的输入。其图片数据主要存储在矩阵 image_data 和矩阵长度 image_size 两个变量中。需要对图片数据进行整合处理，利用创建的 data_to_mat () 方法，将图片数据读取到 OpenCV 中：

```
cv::Mat input_image = data_to_mat(image_data, image_size);
```

接下来就是配置网络图片数据输入，对于节点输入是图片数据的网络节点，其配置网络输入主要分为以下几步：

首先，获取网络输入图片大小。

使用 InferRequest 类中的 get_tensor()方法，获取指定网络节点的 Tensor，其节点要求输入大小在 Shape 容器中，通过获取该容器，得到图片的长宽信息：

```
ov::Tensor input_image_tensor = p->infer_request.get_tensor(input_node_name);
int input_H = input_image_tensor.get_shape()[2]; //获得"image"节点的 Height
int input_W = input_image_tensor.get_shape()[3]; //获得"image"节点的 Width
```

其次，按照输入要求，处理输入图片。

在这一步，我们除了要按照输入大小对图片进行放缩之外，还要根据 PaddlePaddle 对模型输入的要求进行处理。因此处理图片其主要分为交换 RGB 通道、放缩图片以及对图片进行归一化处理。在此处我们借助 OpenCV 来实现。

OpenCV 读取图片数据并将其放在 Mat 类中，其读取的图片数据是 BGR 通道格式，PaddlePaddle 要求输入格式为 RGB 通道格式，其通道转换主要靠一下方式实现：

```
cv::cvtColor(input_image, blob_image, cv::COLOR_BGR2RGB);
```

接下来就是根据网络输入要求，对图片进行压缩处理：

```
cv::resize(blob_image, blob_image, cv::Size(input_H, input_W), 0, 0, cv::INTER_LINEAR);
```

最后就是对图片进行归一化处理，其主要处理步骤就是减去图像数值均值，并除以方差。查询 PaddlePaddle 模型对图片的处理，其均值 mean=[0.485, 0.456, 0.406]，方差 std=[0.229, 0.224, 0.225]，利用 OpenCV 中现有函数，对数据进行归一化处理：

```
std::vector<float> mean_values{ 0.485 * 255, 0.456 * 255, 0.406 * 255 };
std::vector<float> std_values{ 0.229 * 255, 0.224 * 255, 0.225 * 255 };
std::vector<cv::Mat> rgb_channels(3);
cv::split(blob_image, rgb_channels); // 分离图片数据通道
for (auto i = 0; i < rgb_channels.size(); i++){
//分通道依此对每一个通道数据进行归一化处理
```



```

    rgb_channels[i].convertTo(rgb_channels[i], CV_32FC1, 1.0 / std_values[i], (0.0 - mean_values[i]) /
std_values[i]);
}
cv::merge(rgb_channels, blob_image); // 合并图片数据通道

```

最后，将图片数据输入到模型中。

在此处，我们重写了网络赋值方法，并将其封装到 `fill_tensor_data_image(ov::Tensor& input_tensor, const cv::Mat& input_image)` 方法中，`input_tensor` 为模型输入节点 `Tensor` 类，`input_image` 为处理过的图片 `Mat` 数据。因此节点赋值只需要调用该方法即可：

```
fill_tensor_data_image(input_image_tensor, blob_image);
```

对于普通数据的输入，其方法接口如下：

```

extern "C" __declspec(dllexport) void* __stdcall load_input_data(void* core_ptr, const wchar_t*
input_node_name_wchar, float* input_data);

```

与配置图片数据不同点，在于输入数据只需要输入 `input_data` 数组即可。其数据处理哦在外部实现，只需要将处理后的数据填充到输入节点的数据内存中即可，通过调用自定义的 `fill_tensor_data_float(ov::Tensor& input_tensor, float* input_data, int data_size)` 方法即可实现：

```

std::string input_node_name = wchar_to_string(input_node_name_wchar);
ov::Tensor input_image_tensor = p->infer_request.get_tensor(input_node_name); // 读取指定节点 tensor
int input_size = input_image_tensor.get_shape()[1]; // 获得输入节点的长度
fill_tensor_data_float(input_image_tensor, input_data, input_size); // 将数据填充到 tensor 数据内存上

```

（6）模型推理

上一步中我们将推理内容的数据输入到了网络中，在这一步中，我们需要进行数据推理，这一步中我们留有一个推理接口：

```
extern "C" __declspec(dllexport) void* __stdcall core_infer(void* core_ptr)
```

进行模型推理，只需要调用 `CoreStruct` 结构体中的 `infer_request` 对象中的 `infer()` 方法即可：

```

CoreStruct* p = (CoreStruct*)core_ptr;
p->infer_request.infer();

```

（7）读取推理数据

上一步我们对数据进行了推理，这一步就需要查询上一步推理的结果。对于我们所使用的模型输出，主要有 `float` 数据和 `int` 数据，对此，留有了两种数据的查询接口，其方法为：

```

extern "C" __declspec(dllexport) void __stdcall read_infer_result_F32(void* core_ptr, const wchar_t*
output_node_name_wchar, int data_size, float* infer_result);
extern "C" __declspec(dllexport) void __stdcall read_infer_result_I32(void* core_ptr, const wchar_t*
output_node_name_wchar, int data_size, int* infer_result);

```

其中 `data_size` 为读取数据长度，`infer_result` 为输出数组指针。读取推理结果数据与加载推理数据方式相似，依旧是读取输出节点处数据内存的地址：

```

const ov::Tensor& output_tensor = p->infer_request.get_tensor(output_node_name);
const float* results = output_tensor.data<const float>();

```

针对读取整形数据，其方法一样，只是在转换类型时，需要将其转换为整形数据即可。我们读取的初始数据为二进制数据，因此要根据指定类型转换，否则数据会出现错误。将数据读取出来后，将其放在数据结果指针中，并将所有结果赋值到输出数组中：

```
for (int i = 0; i < data_size; i++) {
    *inference_result = results[i];
    inference_result++;
}
```

（8）删除推理核心结构体指针

推理完成后，我们需要将在内存中创建的推理核心结构地址删除，防止造成内存泄露，影响电脑性能，其接口该方法为：

```
extern "C" __declspec(dllexport) void __stdcall core_delet(void* core_ptr);
```

在该方法中，我们只需要调用 `delete` 命令，将结构体指针删除即可。

1.4.4 编写模块定义文件

我们在定义接口方法时，在原有方法的基础上，增加了 `extern "C"`、`__declspec(dllexport)` 以及 `__stdcall` 三个标识，其主要原因是为了让编译器识别我们的输出方法。其中，`extern "C"` 是指示编译器这部分代码按 C 语言（而不是 C++）的方式进行编译；`__declspec(dllexport)` 用于声明导出函数、类、对象等供外面调用；`__stdcall` 是一种函数调用约定。通过上面三个标识，我们在 C++ 中所写的接口方法，会在 `dll` 文件中暴露出来，并且可以实现在 C# 中的调用。

不过上面所说内容，我们在编辑器中可以通过模块定义文件（`.def`）所实现，在模块定义文件中，添加以下代码：

```
LIBRARY
    "OpenVinoSharp"
EXPORTS
    core_init
    set_input_image_sharp
    set_input_data_sharp
    load_image_input_data
    load_input_data
    core_infer
    read_infer_result_F32
    read_infer_result_I32
    core_delet
```

`LIBRARY` 后所跟的为输出文件名，`EXPORTS` 后所跟的为输出方法名。仅需要以上语句便可以替代 `extern "C"`、`__declspec(dllexport)` 以及 `__stdcall` 的使用。

1.4.5 生成 dll 文件

前面我们将项目配置输出设置为了生成 `dll` 文件，因此该项目不是可以执行的 `exe` 文件，只能生成不能运行。右键项目，选择重新生成/生成。在没有错误的情况下，会看到项目成功的提示。可以看到 `dll` 文件在解决方案同级目录下 `\x64\Release\` 文件夹下。

使用 `dll` 文件查看器打开 `dll` 文件，如图 1-11 所示；可以看到，我们创建的四方法接口已经暴露在 `dll` 文件中。

Function Name	Address	Relative Address	Ordinal
core_init	0x0000000180001470	0x00001470	3 (0x3)
set_input_image_sharp	0x0000000180001880	0x00001880	9 (0x9)
set_input_data_sharp	0x00000001800019a0	0x000019a0	8 (0x8)
load_image_input_data	0x0000000180001ab0	0x00001ab0	4 (0x4)
load_input_data	0x0000000180002040	0x00002040	5 (0x5)
core_infer	0x00000001800021b0	0x000021b0	2 (0x2)
read_infer_result_F32	0x00000001800021d0	0x000021d0	6 (0x6)
read_infer_result_I32	0x00000001800022f0	0x000022f0	7 (0x7)
core_delet	0x00000001800023d0	0x000023d0	1 (0x1)

图 1- 11 dll 文件方法输出目录

1.5 C#构建 Core 类

1.5.1 新建 C#类库

右击解决方案，添加->新建项目，选择添加 C#类库，项目名命名为 OpenVinoSharp，项目框架根据电脑中的框架选择，此处使用的是 .NET 5.0。新建完成后，然后右击项目，选择添加->新建项，选择类文件，添加 Core.cs 和 NativeMethods.cs 两个类文件。

1.5.2 引入 dll 文件中的方法

在 NativeMethods.cs 文件下，我们通过 [DllImport()] 方法，将 dll 文件中所有的方法读取到 C# 中。读取方式如下：

```
[DllImport(openvino_dll_path, CharSet = CharSet.Unicode, CallingConvention = CallingConvention.Cdecl)]
public extern static IntPtr core_init(string model_file, string device_name);
```

其中 openvino_dll_path 为 dll 文件路径，CharSet = CharSet.Unicode 代表支持中文编码格式字符串，CallingConvention = CallingConvention.Cdecl 指示入口点的调用约定为调用方清理堆栈。

上述所列出的为初始化推理模型，dll 文件接口在匹配时，是通过方法名字匹配的，因此，方法名要保证与 dll 文件中一致。其次就是方法的参数类型要进行对应，在上述方法中，函数的返回值在 C++ 中为 void*，在 C# 中对应的为 IntPtr 类型，输入参数中，在 C++ 中为 wchar_t* 字符指针，在 C# 中对应的为 string 字符串。通过方法名与参数类型一一对应，在 C# 可以实现对方法的调用。其他方法的引用类似，在此处不在一一赘述，具体可以参照项目提供的源代码。

1.5.3 创建 Core 类

为了方便地调用我们通过 dll 引入的 OpenVINO™ 方法，减少使用时的函数方法接口，我们在 C# 中重新组建我们自己的推理类，命名为 Class Core，其主要成员变量和方法如图 1- 12 所示。

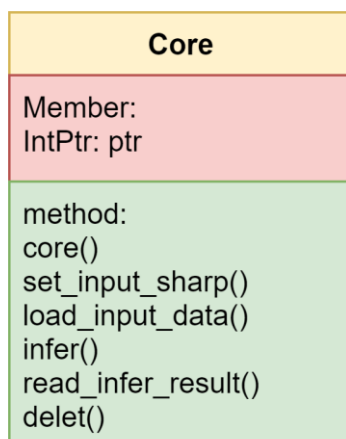


图 1- 12 Core 类图

在 Core 类中，我们只需要创建一个地址变量，作为 Core 类的成员变量，用于接收接口函数返回的推理核心指针，该成员变量我们只需要在当前类下访问，因此将其设置为私有变量：

```
private IntPtr ptr = new IntPtr();
```

接下来，构建类的构造函数，在类的初始化时，我们需要输入模型地址以及设备类型，通过调用 dll 文件中引入的方法，获取初始化指针，对成员变量进行赋值，实现类的初始化：

```
public Core(string model_file, string device_name) {
    ptr = NativeMethods.core_init(model_file, device_name);
}
```

然后构造其中的方法，在构建设置数据输入形状时，我们需要提供的为节点名以及形状数据，为了简化该方法，我们合并了图片形状设置与普通数据形状设置接口，通过判断输入数组的长度，来确定是对那一个形状的设置：

```
public void set_input_sharp(string input_node_name, ulong[] input_size);
```

对于该类中方法的构建，可以参考源码文件，在此处不做详述。并且其他方法构建方式基本相似，此处不在一一赘述，具体可以参考源码文档。

1.5.4 编译 Core 类库

右击项目，点击生成/重新生成，出现如下图 1- 13 所示，表示编译成功。

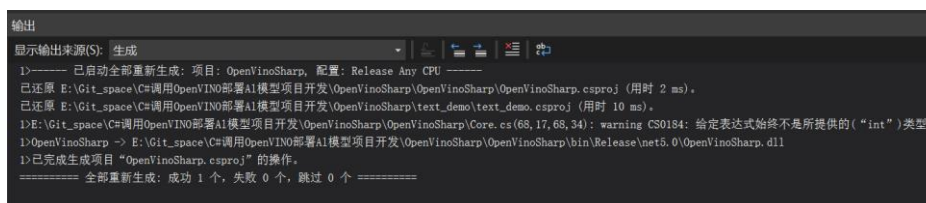


图 1- 13 Core 类编译输出

1.6 C#实现 OpenVINO™ 方法的调用

1.6.1 新建 C#项目

右击解决方案，添加->新建项目，选择添加 C#控制台项目，项目框架根据电脑中的框架选择，此处使用的是.NET 5.0。

新建完成后，右击项目，选择属性，点击新页面中的生成，在常规下，将目标平台改为 X64。具体操作如图 1- 14 所示。



图 1- 14 C#项目设置

1.6.2 添加 OpenCVsharp

右击项目，选择管理 NuGet 程序包，在新页面中选择浏览，在搜索框中输入 opencvsharp3，在搜索结果中，找到 OpenCvSharp3-AnyCPU，然后右侧点击安装，具体操作步骤如图 1- 15 所示。

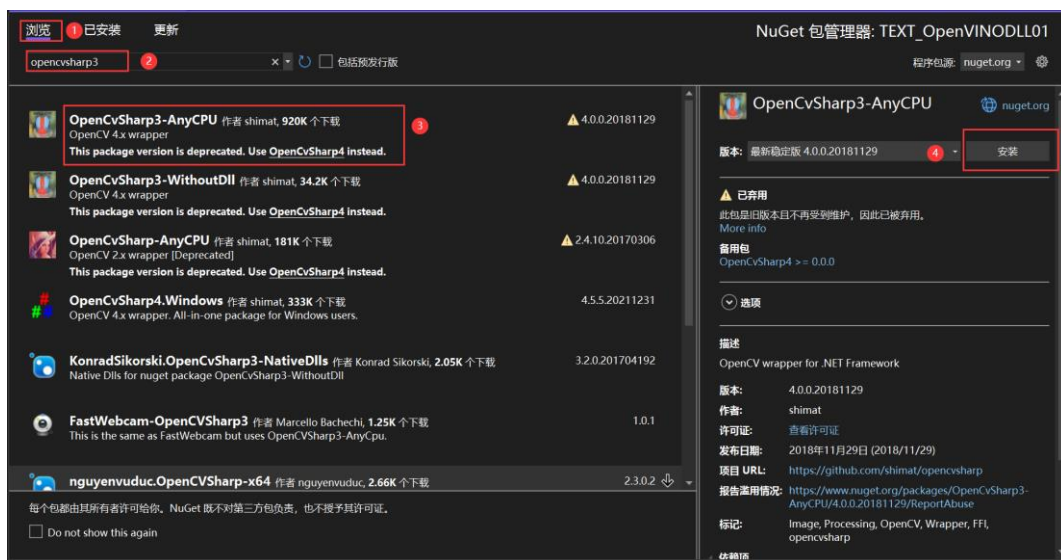


图 1- 15 NuGet 程序包安装

1.6.3 添加项目引用

上一步中我们将 dll 文件中的方法引入到 C#中，并组建了 Core 类，在这一步中，我们主要通过调用 Core 类，进行 AI 模型的部署，所以需要引入上一步的项目。

右击当前项目，选择添加，选择项目引用，在出现的窗体中，选择上一步中创建的项目

OpenVinoSharp，点击确定；然后在当前项目下，添加 using OpenVinoSharp 命名空间。具体操作如图 1- 16 所示。

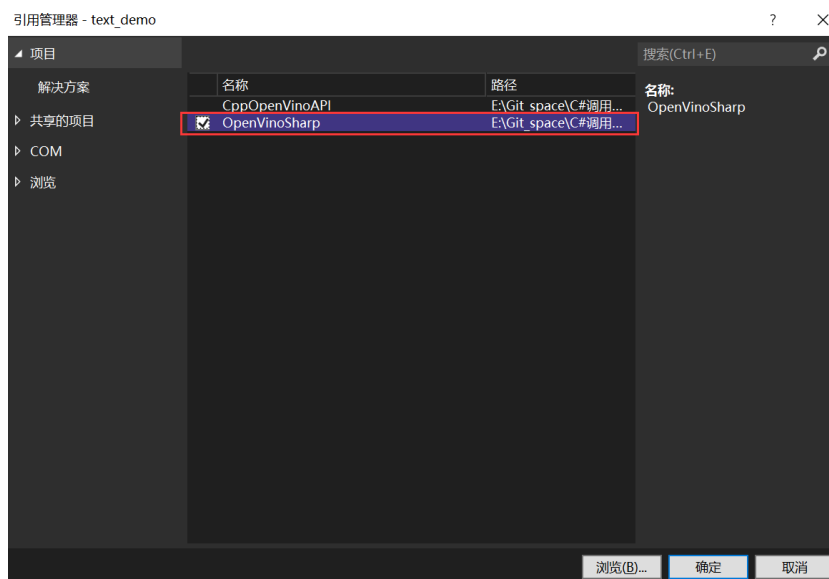


图 1- 16 添加项目引用

1.6.4 编写代码测试花卉分类模型

在该项目中，我们提供了两种推理模型，此处我们以花卉分类模型为例，简介如何通过 C#调用 OpenVINO™ 进行 AI 模型的部署。

(1) 引入相关变量

```
string device_name = "CPU";
string model_file = "E:/Text_Model/flowerclas/flower_rec.onnx";
string image_file = "E:/Text_dataset/flowers102/jpg/image_00001.jpg";
string input_node_name = "x";
string output_node_name = "softmax_1.tmp_0";
```

为了让大家更加清晰的看懂后续代码，在此处对引入的相关变量进行解释：

device_name：设备类型名称，可为 CPU、GPU 以及 AUTO（均可）；

model_file：模型地址，可以为 onnx、pdmodel 或者 xml 格式；

image_file：测试图片地址；

input_node_name：输入模型节点名，当多输入时，可以为数组；

output_node_name：输出模型节点名，当多输出时，可以为数组。

(2) 初始化 Core 类

在此处我们直接调用 Core 类的构造函数，进行初始化：

```
Core ie = new Core(model_file_paddle, device_name);
```

(3) 配置模型输入

花卉分类模型输入只有一个，即待分类花卉图片。如果我们调用的模型未指定输入大小，需要在输入数据前，调用模型输入数据形状设置方法，设置节点输入数据形状。图片数据为三维数组，再加一个 batchsize，最终为四维数据，将形状数据放在数组中，调用 set_input_shape()方法：

```
ulong[] image_sharp = new ulong[] { 1, 3, 224, 224 };
ic.set_input_sharp(input_node_name, image_sharp);
```

对于图片数据，需要将其转为矩阵数据，在此处，我们可以直接使用 `opencvsharp` 中的编解码方法，将图片数据放置在 `byte` 数组中：

```
Mat image = new Mat(image_file);
byte[] image_data = new byte[2048 * 2048 * 3];
ulong image_size = new ulong();
image_data = image.ImEncode(".bmp");
image_size = Convert.ToUInt64(image_data.Length);
```

就最后调用 `Core` 类中的 `load_input_data()` 方法，将数据加载到推理网络中：

```
ic.load_input_data(input_node_name, image_data, image_size);
```

在配置完输入数据后，调用模型推理方法，对输入数据进行推理：

```
ic.infer();
```

接下来就是读取推理结果，对于模型的推理结果输出一般为数组数据，可以通过调用 `Core` 类中读取推理数据结果的方法，对与花卉分类模型的输出，其结果为长度为 102 的浮点型数据，所以直接调用 `read_inference_result<T>()` 方法读取即可：

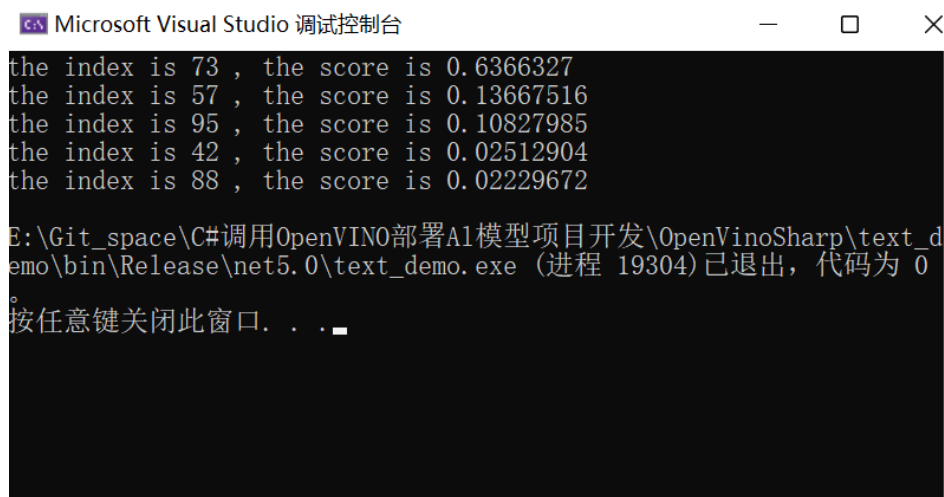
```
float[] result = new float[102];
result = ic.read_inference_result<float>(output_node_name, 102);
```

在读取推理数据时，我们一定要根据模型的书名读取正确的结果数据，因为如果超出实际输出长度，其结果数据会掺杂其他干扰数据。

最后一步就是处理输出数据。对于不同的推理模型，其结果处理方式是不同的，对于花卉分类模型，其输出为 102 种分类情况打分，因此，在处理数据时，需要找出得分最高的哪一类即可。在此处，我们提供了一个方法，该方法可以实现提取数组中前 N 个 `max` 数据的位置，通过调用该方法，我们可以获取分类结果中分数最高的几个结果，并将结果打印输出：

```
int[] index = find_array_max(result, 5);
for (int i = 0; i < 5; i++){
    Console.WriteLine("the index is {0} , the score is {1} ", index[i], result[index[i]]);
}
```

最终输出结果如图 1-17 所示，该页面打印出来了推理结果预测分数最大的前五个分数和其对应的索引值，最后可以通过索引值查询 `flowers102_label_list.txt` 文件中对应的花卉名称。



```

Microsoft Visual Studio 调试控制台
the index is 73 , the score is 0.6366327
the index is 57 , the score is 0.13667516
the index is 95 , the score is 0.10827985
the index is 42 , the score is 0.02512904
the index is 88 , the score is 0.02229672

E:\Git_space\C#调用OpenVINO部署AI模型项目开发\OpenVinoSharp\text_demo\bin\Release\net5.0\text_demo.exe (进程 19304) 已退出, 代码为 0
按任意键关闭此窗口. . .

```

图 1- 17 花卉分类结果

在程序最后，我们该需要将前面在内存上创建推理引擎结构体进行删除，只需要调用 Core 类下的 `delete()` 即可。

1.6.5 编写代码测试车辆识别模型

对于车辆识别模型此处不再进行详细讲解，具体实现可以参考源码文件，此处只对一些不同点进行分析。

在配置输入时，除了需要配置图片数据输入，还需要配置图片长宽数据以及长宽缩放比例数据，在配置时，只需要将数据放置在数组中，通过调用 `load_input_data()` 方法实现，对于设置缩放比例数据输入，如下所示：

```

float scale_h = 608.00f / image.Height;
float scale_w = 608.00f / image.Width;
float[] scale_factor = new float[] { scale_h, scale_w };
ie.load_input_data(input_node_name[1], scale_factor);

```

对于该模型推理结果数据，总共有两个节点输出，一个是识别结果数量，一个为识别结果信息。对于识别结果数量，其数据类型为整形数据，对于单图片输入，只需要读取一位即可，利用该数据，确定识别结果信息长度。识别结果信息为 6 列 N 行数据，在数据读取时，我们将其转化为一维数据，所以在处理数据时，以 6 位数据为一组，进行处理。

识别结果信息数据中，第 1 位为识别标签，第 2 位为识别得分，第 3 位到第 6 位四个数据为位置矩形框对角点坐标，通过每 6 位读取一次数据，获取识别结果。在此处，我们提供了专门的结果处理方法，通过该方法们可以实现直接将结果绘制在原图片上：

```

image = draw_image_resule(image, resule_num[0], result, lable, 0.2f);

```

其中 `image` 为原图片，`resule_num[0]` 为识别结果数量，`result` 为识别结果数组，`lable` 为结果标签，`0.2f` 为评价得分下限。通过结果处理，将识别结果标注在图片中，并把识别结果以及得分情况打印在图片中，最终识别结果如图 1- 18 所示。

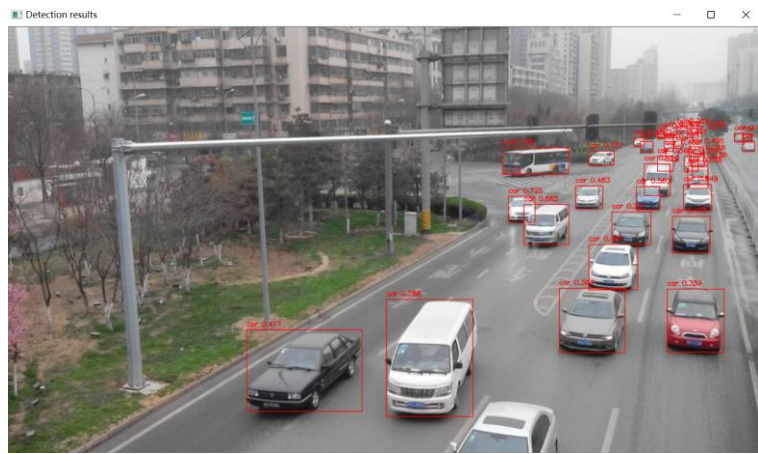


图 1- 18 车辆类型识别结果输出

1.7 程序时间分析

为了对比 C++、C#以及 Python 这三个平台下调用 OpenVINO™ 所使用的时间，我们通过测试 flower_clas 以及 vehicle_yolov3_darknet 模型运行时间进行对比，在同一台电脑相同运行环境之下，以及对模型的处理方式在不同编程语言下尽量做到相同，在程序测试 100 次之后，得到结果表 1- 3 如所示。

表 1- 3 程序运行时间

项目名称	编程语言	程序各过程运行时间/(ms)				测试环境
		总时间	前处理 (pre-process)	推理运算 (inference)	后处理 (NMS)	
flower_clas	C++	302.41	255.19	47.22	0	Shape: [1, 3, 224 ,224]
	C#	306.75	255.15	51.53	0.06	Net: ResNet50
	Python	319.31	267.16	52.16	0	CPU: AMD R7 5800H
flower_clas	C++	273.06	215.93	57.12	0.01	Shape: [1, 3, 224 ,224]
	C#	272.44	208.76	63.63	0.04	Net: ResNet50
	Python	294.49	223.65	70.81	0.	CPU: Intel (R) i9-12900K
vehicle_yolov3_darknet	C++	1184.99	611.86	572.59	0.54	Shape: [1, 3, 608 ,608]
	C#	1199.11	620.58	577.51	1.01	Net: YOLOv3
	Python	1247.55	683.55	563.54	0.47	CPU: AMD R7 5800H
vehicle_yolov3_darknet	C++	970.43	512.26	457.70	0.46	Shape: [1, 3, 608 ,608]
	C#	962.45	525.50	439.64	0.26	Net: YOLOv3
	Python	1154.43	572.84	581.16	0	CPU: Intel (R) i9-12900K

在本次检测中，我们通过 C++、C#以及 Python 分别调用 OpenVINO™ 进行模型的部署与推理，通过上述表格，一方面可以看出，C#通过调用 C++的 dll，实现模型的部署与推理，并没有太大的影响程序的运行速度；另一方面，C++与 C#部署模型推理，在总时间上来看，运行速度是优于 Python 的。

测试模型运行时间所使用的测试代码，已同步到远程代码托管仓库 gitee 与 github 中，具体在 opencvino_run_time 文件夹下，使用人员可以根据自己的设备对 C++、C#和 Python 三个平台进行测试。

1.8 项目总结

该项目通过 C++调用 OpenVINO™，创建推理方法接口，并通过调用 dll 文件的方式，在 C#中进行重新构建 Core 模型推理类，并测试了花卉分类模型以及车辆识别模型，在预测结果精度以及预测时间上，和 C++相比，并没有较大的差异。

该项目所提供的方法，证实了 C#平台调用 OpenVINO™ 的可行性，为后续在 C#部署 AI 模型项目提供了有效途径。