主代码

```
int main(int argc, char* argv[]) {


    const int batch_size = 1;


    std::vector<int> input_shape = { batch_size,  INPUT_H, INPUT_W, CHANNELS };
    //输入数组，batch_size * CHANNELS * INPUT_H * INPUT_W为输入数据总量，1.0为初始值
    std::vector<float> input_data(batch_size * CHANNELS * INPUT_H * INPUT_W,
1.0);



    static const int num_anchors = 3;
    int dims = num_anchors * (NUM_CLASSES + 5);
    int output_size = 1 * 7581 * dims;
    //用于保存图片的大小数值



    using namespace nvinfer1;

    IRuntime* runtime{ nullptr };
    //运行时接口，用于管理在特定硬件中执行推理过程，用于引擎的建立和销毁，设置配置。

    ICudaEngine* engine{ nullptr };
    //引擎接口，用于将深度学习模型编译为可以在硬件上执行的二进制文件，

    IExecutionContext* context{ nullptr };
    //执行上下文接口，用于执行推理，有设置输入输出的数据，执行推理和获得推理结果。
    NVLogger gLogger_test;
    //创建NVLgoger类，将日志信息输出到指定的位置，控制台



    //载入模型
    const char* engine_file_path_in = "../trt_model/jyz_jiayouji_sim.trt";

    //载入设备
    cudaSetDevice(DEVICE);
    // create a model using the API directly and serialize it to a stream


    //设置一个char*类型对象，存放载入的模型
    char* trtModelStream{ nullptr };
    size_t size{ 0 };

    //直接使用API创建模型，并将其序列化为流
    const std::string engine_file_path{ engine_file_path_in };
    std::ifstream file(engine_file_path, std::ios::binary);
    if (file.good()) {
        file.seekg(0, file.end);
        size = file.tellg();
        file.seekg(0, file.beg);
        trtModelStream = new char[size];
        assert(trtModelStream);
        file.read(trtModelStream, size);
        file.close();
    }
    else
    {
        std::cout << "yolov4_test : Bad engine file!" << std::endl;
        exit(-1);
    }

    std::cout << "yolov4_test : Success read engine file!" << engine_file_path
<< std::endl;



    runtime = createInferRuntime(gLogger_test.getTRTLogger());
    //创建一个运行时实例，接受Ilogger对象
    assert(runtime != nullptr);
    //创建引擎，载入模型
    engine = runtime->deserializeCudaEngine(trtModelStream, size);
    assert(engine != nullptr);
    //创建一个执行上下文对象
    context = engine->createExecutionContext();
    assert(context != nullptr);
    //销毁对象
    delete[] trtModelStream;





    std::vector<std::string> all_img_paths5;
    if (all_img_paths5.size() <= 0)
    {
        std::vector<cv::String> cv_all_img_paths;

        cv::glob("../testjyj_xunjian_imgs/", cv_all_img_paths);
        //获得指定文件下的所有文件路经

        //将读到的文件路径输入到all_img_path5容器中
        for (const auto& img_path : cv_all_img_paths) {
            all_img_paths5.push_back(img_path);
        }
    }

    for (int i = 0; i < all_img_paths5.size(); i++)
    {
        // std::vector<std::size_t> shape = { 1,7581,dims };

        //设置输出容器
        std::vector<float> out_data(output_size, 1.0);
        std::string input_image_path = all_img_paths5[i];
        //读入图片
        cv::Mat img = cv::imread(input_image_path);
        //读取宽高
        int img_w = img.cols;
        int img_h = img.rows;

        //重新设置大小
        cv::Mat pr_img = static_resize(img);
        std::cout << "static_resized image" << std::endl;

        //python: img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        cv::Mat rgb_img;
        //将图片的色彩转化为RGB
        cv::cvtColor(pr_img, rgb_img, cv::COLOR_BGR2RGB);


        //todo:前处理
        blobFromImageAndNormalize(rgb_img, input_data);

        //获得当前时间。
        auto start = std::chrono::system_clock::now();

        //进行处理
        doInference(*context, input_data.data(), out_data.data(), output_size,
pr_img.size());

        auto end = std::chrono::system_clock::now();

        std::cout << "infer time:" <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() <<
"ms" << std::endl;

        std::cout << "output num is " << out_data.size();



        std::vector<Object> objects;
        //对输出的结果进行处理。
        decode_outputs(out_data, /*out*/objects, img_w, img_h);
        //绘制结果
        draw_objects(img, objects, input_image_path);

    }

    context->destroy();
    engine->destroy();
    runtime->destroy();



    return 0;
}
```

```
inline cv::Mat static_resize(cv::Mat & img) {

    float scale_x = float(INPUT_W) / (img.cols * 1.0);
    float scale_y = float(INPUT_H) / (img.rows * 1.0);

    // r = std::min(r, 1.0f);
    //为了取整?
    int unpad_w = scale_x * img.cols;
    int unpad_h = scale_y * img.rows;
    //设置一个mat类
    cv::Mat re(unpad_h, unpad_w, CV_8UC3);

    //重新设置img大小
    cv::resize(img, re, re.size(), 0, 0, cv::INTER_CUBIC);

    //设置一个彩色图像out
    cv::Mat out(INPUT_H, INPUT_W, CV_8UC3, cv::Scalar(114, 114, 114));
    //把re灰度图像内容复制到彩色图像out的第一个通道中
    re.copyTo(out(cv::Rect(0, 0, re.cols, re.rows)));
    return out;
}
```

处理前的预处理

```
inline void blobFromImageAndNormalize(cv::Mat& img,std::vector<float>&
arrvectorOut){

    int img_h = img.rows;
    int img_w = img.cols;

    for (size_t  h = 0; h < img_h; h++)
    {
        for (size_t w = 0; w < img_w; w++)
        {
                for (size_t c = 0; c < CHANNELS; c++)
                {
                   //通过算法把图片转化为一位的浮点数数组
                    arrvectorOut[h*img_w*CHANNELS  + w*CHANNELS + c] =
                        ((float)img.at<cv::Vec3b>(h, w)[c]) / 255.0;
                }


        }
    }



    return ;
}
```

处理图形

```
void doInference(IExecutionContext& context, float* input, float* output, const
int output_size, cv::Size input_shape) {
    const ICudaEngine& engine = context.getEngine();

    // Pointers to input and output device buffers to pass to engine.
    // Engine requires exactly IEngine::getNbBindings() number of buffers.
    //获得引擎捆绑的数量
    assert(engine.getNbBindings() == 2);
    void* buffers[2];

    // In order to bind the buffers, we need to know the names of the input and
output tensors.
    // Note that indices are guaranteed to be less than IEngine::getNbBindings()

    //在创建 TensorRT 引擎时，需要指定输入和输出张量的名称、数据类型、形状等信息，以便在后续的
推理过程中进行输入和输出的对接。
    //获得输入张量在引擎上的索引
    const int inputIndex = engine.getBindingIndex(INPUT_BLOB_NAME);
    //检查是否正确
    assert(engine.getBindingDataType(inputIndex) ==nvinfer1::DataType::kFLOAT);
    //获得输出张量在引擎上的索引
    const int outputIndex = engine.getBindingIndex(OUTPUT_BLOB_NAME);
    assert(engine.getBindingDataType(outputIndex)==nvinfer1::DataType::kFLOAT);
    //获得引擎支持的最大批处理大小
    int mBatchSize = engine.getMaxBatchSize();

    // Create GPU buffers on device
    //创建GPU上分配一段内存空间
    CHECK(cudaMalloc(&buffers[inputIndex], 3 * input_shape.height *
input_shape.width * sizeof(float)));
    CHECK(cudaMalloc(&buffers[outputIndex], output_size*sizeof(float)));

    // Create stream
    //创建流
    cudaStream_t stream;
    CHECK(cudaStreamCreate(&stream));

    // DMA input batch data to device, infer on the batch asynchronously, and DMA
output back to host
// DMA向设备输入批处理数据，对批处理进行异步推断，然后DMA输出回主机
    //将input中的数据复制到GPU内存中。
    CHECK(cudaMemcpyAsync(buffers[inputIndex], input, 3 * input_shape.height *
input_shape.width * sizeof(float), cudaMemcpyHostToDevice, stream));
    //表示进行推理操作。
    //1表示要执行的批处理大小
    context.enqueue(1, buffers, stream, nullptr);
    //将GPU内存中的outputIndex数据复制到主机内存中。
    CHECK(cudaMemcpyAsync(output, buffers[outputIndex], output_size *
sizeof(float), cudaMemcpyDeviceToHost, stream));
    //等待流上的所有操作进行完后返回。
    cudaStreamSynchronize(stream);

    // Release stream and buffers
    //释放 流和缓冲区
    cudaStreamDestroy(stream);
    CHECK(cudaFree(buffers[inputIndex]));
    CHECK(cudaFree(buffers[outputIndex]));
}
```

处理出来的数组

```
void decode_outputs(std::vector<float>& prob, /*out*/std::vector<Object>&
objects, const int img_w, const int img_h)
{

    static int num_anchors=3;
    int dims = num_anchors* (NUM_CLASSES + 5);

    std::vector<std::size_t> shape = { 1,7581,dims };
    //将一维数组转化为指定形状的多维数组
    auto a1 = xt::adapt(prob, shape);

    //返回a1[0]的值
    std::cout << "a1[0] :" <<xt::view(a1,0,0)<<std::endl;

    //获得a1指定位置的切片
    xt::xarray<float> output_l =  xt::view(a1,xt::all(), xt::range(0, 19 * 19),
xt::all());
    xt::xarray<float> output_m = xt::view(a1,xt::all(), xt::range( 19 * 19,19 *
19 + 38 * 38), xt::all());
    xt::xarray<float> output_s =xt::view(a1,xt::all(), xt::range(19 * 19 + 38 *
38, 38*38+19*19+76*76), xt::all());


    //将切片的转化为指定形状的多维数组
    output_l.reshape({1, input_image_shape[0]/32, input_image_shape[1]/32, 3,
5+NUM_CLASSES});
    output_m.reshape({1, input_image_shape[0]/16, input_image_shape[1]/16, 3,
5+NUM_CLASSES});
    output_s.reshape({1, input_image_shape[0]/8, input_image_shape[1]/8, 3,
5+NUM_CLASSES});

    //将output_x的类型转化成xt::dynamic_shape类型的对象
    std::cout << "output_l shape:" <<xt::adapt(output_l.shape()) <<std::endl;
    std::cout << "output_m shape:" <<xt::adapt(output_m.shape()) <<std::endl;
    std::cout << "output_s shape:" <<xt::adapt(output_s.shape()) <<std::endl;


        std::cout << "-----------------------handle output_l--------------------
-------"<<std::endl;
        xt::xarray<int> anchors_l = {{142, 110}, {192, 243}, {459, 401}};
        xt::xtensor<float, 2> boxes_result_l;
        xt::xtensor<int,1> classes_l;
        xt::xtensor<float,1> scores_l;


        decodeOne(output_l,
                  anchors_l,
                  /*out*/ boxes_result_l,
                  /*out*/classes_l,
                    /*out*/ scores_l
                    );

        std::cout << "-----------------------handle output_m--------------------
-------"<<std::endl;

        xt::xarray<int> anchors_m = {{36, 75}, {76, 55},{72, 146}};
    //2表示二维数组
        xt::xtensor<float, 2> boxes_result_m;
        xt::xtensor<int,1> classes_m;
        xt::xtensor<float,1> scores_m;


        decodeOne(output_m,
                    anchors_m,
                    /*out*/ boxes_result_m,
                    /*out*/classes_m,
                    /*out*/ scores_m
                    );

        std::cout << "-----------------------handle output_s--------------------
-------"<<std::endl;
        xt::xarray<int> anchors_s = {{12, 16}, {19, 36}, {40, 28}};
    //存储目标的输出的锚框坐标信息
        xt::xtensor<float, 2> boxes_result_s;
    //存储类型信息
        xt::xtensor<int,1> classes_s;
    //得分信息
        xt::xtensor<float,1> scores_s;

        //内部处理
        decodeOne(output_s,
                    anchors_s,
                    /*out*/ boxes_result_s,
                    /*out*/classes_s,
                    /*out*/ scores_s
                    );

    // std::cout << "-----------------------------------------------------------
---------"<<std::endl;
    //
    std::cout << "boxes_result_l shape:" <<xt::adapt(boxes_result_l.shape())
<<std::endl;
    std::cout << "boxes_result_m shape:" <<xt::adapt(boxes_result_m.shape())
<<std::endl;
    std::cout << "boxes_result_s shape:" <<xt::adapt(boxes_result_s.shape())
<<std::endl;

    //把三个二维浮点型数组沿着一个维度拼接，结果存到新的二维数组boxes中
    xt::xtensor<float, 2> boxes = xt::concatenate(xtuple(boxes_result_l,
boxes_result_m,boxes_result_s), 0);

    xt::xtensor<int,1> classes = xt::concatenate(xtuple(classes_l, classes_m,
classes_s), 0);

    xt::xtensor<float,1> scores = xt::concatenate(xtuple(scores_l, scores_m,
scores_s), 0);
```

```
    auto w = img_w; //原图尺寸
    auto h = img_h;

    xt::xtensor<float, 1> image_dims({w, h, w, h});

    boxes = boxes * image_dims;//获得原始图像的坐标值





////////////////////////////////////////////////////////////////////////////////
////
    if(classes.size() == 0)
    {
        return;
    }

    std::vector<Object> proposals;
    std::unordered_set<int> setclsIDs ;
    for(size_t i=0;i<classes.size();i++)
    {
        setclsIDs.insert(classes(i));
    }

    for (auto& clsID : setclsIDs)
    {
        proposals.clear();

        for(size_t i=0;i<classes.size();i++)
        {
            if( classes(i)  == clsID)//类型对上
            {
                //输出到obj结构体中
                Object obj;
                obj.rect.x = boxes(i, 0);//x轴
                obj.rect.y = boxes(i, 1);//y轴
                obj.rect.width = boxes(i, 2);//框宽
                obj.rect.height = boxes(i, 3);//框高
                obj.label = classes(i);//类型
                obj.prob = scores(i);//相似度


                proposals.push_back(obj);
            }

        }


        std::cout << "clsID:"<<clsID<<",num of boxes before nms: " <<
proposals.size() << std::endl;
        //进行快排，找到最高相似度
        qsort_descent_inplace(proposals);

        std::vector<int> picked;
        //执行NMS算法,降低输出框的重叠度
        nms_sorted_bboxes(proposals, picked, NMS_THRESH);

        int count = picked.size();


        std::cout << "num of boxes after nms: " << count << std::endl;

        for (int i = 0; i < count; i++)
        {//重新放obj
            objects.push_back(proposals[picked[i]]);


            // adjust offset to original unpadded
            //将偏移量调整为原始无衬垫
            //上坐标
            float x0 = (objects.back().rect.x) ;
            float y0 = (objects.back().rect.y) ;
            //下坐标
            float x1 = (objects.back().rect.x + objects.back().rect.width) ;
            float y1 = (objects.back().rect.y + objects.back().rect.height) ;




            // clip
            //进行边框裁剪
            //
            x0 = std::max(std::min(x0, (float)(img_w - 1)), 0.f);
            //把x0限制在0~imgw-1中
            y0 = std::max(std::min(y0, (float)(img_h - 1)), 0.f);
            x1 = std::max(std::min(x1, (float)(img_w - 1)), 0.f);
            y1 = std::max(std::min(y1, (float)(img_h - 1)), 0.f);

            objects.back().rect.x = x0;
            objects.back().rect.y = y0;
            objects.back().rect.width = x1 - x0;
            objects.back().rect.height = y1 - y0;

        }

    }
}
```

绘制图像

```
static void draw_objects(const cv::Mat& bgr, const std::vector<Object>& objects,
std::string f)
{
    //检测到的类型名
    static const char* class_names[] = {
        "oil_person",
        "oiling_machine_open",
        "open_shell",
        "open_shell2",
        "open_shell3",
        "tank_truck",
        "fire_extinguisher_35",
        "fire_extinguisher_5",
        "person",
        "work_clothe_blue",
        "work_clothe_yellow",
        "Oil_start",
        "reflective_vest",
        "rider",
        "cement_truck",
        "person_indistinct",
        "helmet_head_indistinct"
        "person",
        "person_indistinct",
        "helmet_head_indistinct",
        "traffic light",
        "fire hydrant", "stop sign", "parking meter", "bench", "bird", "cat",
"dog", "horse", "sheep", "cow",
        "elephant", "bear", "zebra", "giraffe", "backpack", "umbrella",
"handbag", "tie", "suitcase", "frisbee",
        "skis", "snowboard", "sports ball", "kite", "baseball bat", "baseball
glove", "skateboard", "surfboard",
        "tennis racket", "bottle", "wine glass", "cup", "fork", "knife",
"spoon", "bowl", "banana", "apple",
        "sandwich", "orange", "broccoli", "carrot", "hot dog", "pizza", "donut",
"cake", "chair", "couch",
        "potted plant", "bed", "dining table", "toilet", "tv", "laptop",
"mouse", "remote", "keyboard", "cell phone",
        "microwave", "oven", "toaster", "sink", "refrigerator", "book", "clock",
"vase", "scissors", "teddy bear",
        "hair drier", "toothbrush"
    };
    //复制一份Mat 类对象
    cv::Mat image = bgr.clone();


    for (size_t i = 0; i < objects.size(); i++)
    {
        const Object& obj = objects[i];
         //输出标记出的信息
        fprintf(stderr, "%d = %.5f at %.2f %.2f %.2f x %.2f\n", obj.label,
obj.prob,
            obj.rect.x, obj.rect.y, obj.rect.width, obj.rect.height);
        //获得要表示的颜色值
        cv::Scalar color = cv::Scalar(color_list[obj.label][0],
color_list[obj.label][1], color_list[obj.label][2]);
        //计算颜色对象的均值，取向量的第一个元素B通道
        float c_mean = cv::mean(color)[0];
        cv::Scalar txt_color;
        //根据均值获得对应文字颜色
        if (c_mean > 0.5) {
            txt_color = cv::Scalar(0, 0, 0);
        }
        else {
            txt_color = cv::Scalar(255, 255, 255);
        }
        //在图像指定的位置绘制一个rect对象中指定位置和大小的框
        cv::rectangle(image, obj.rect, color * 255, 2);
        char text[256];
        sprintf(text, "%s %.1f%%", class_names[obj.label], obj.prob * 100);

        int baseLine = 0;
        //设置图像中要显示的文字大小
        cv::Size label_size = cv::getTextSize(text, cv::FONT_HERSHEY_SIMPLEX,
0.4, 1, &baseLine);
        //设置文字框颜色
        cv::Scalar txt_bk_color = color * 0.7 * 255;
        //设置文字放置位置
        int x = obj.rect.x;
        int y = obj.rect.y + 1;
        //int y = obj.rect.y - label_size.height - baseLine;
        if (y > image.rows)
            y = image.rows;
        //if (x + label_size.width > image.cols)
            //x = image.cols - label_size.width;
         //在图像中绘制一个特定位置，大小为文字大小 特定颜色 -1宽的框
        cv::rectangle(image, cv::Rect(cv::Point(x, y),
cv::Size(label_size.width, label_size.height + baseLine)),
            txt_bk_color, -1);
        //在框中写上位置
        cv::putText(image, text, cv::Point(x, y + label_size.height),
            cv::FONT_HERSHEY_SIMPLEX, 0.4, txt_color, 1);
    }
    //输出图像
    std::stringstream buffer;
    buffer << "det_res_" << saveCnt++ << ".png";
    cv::imwrite(buffer.str().c_str(), image);
    fprintf(stderr, "save vis file\n");
    /* cv::imshow("image", image); */
    /* cv::waitKey(0); */
}
```

大概流程

开始->初始化输入列表和输出数值->使用流载入模型->初始化引擎->导入文件中图片->对导入的图片进行推理前预处理成模型所需的样子->进行推理->处理通过模型推理出来的模型，转化成对应的识别框（关键）->在原来的图片中绘制框出来。

总结

阅读完这篇代码，

1. 我大致了解了图像识别的流程和实现。
2. 了解了个个模块划分和模块之间的关系，大致是了解了函数之间的调用关系和变量的作用。
3. 大概了解了程序中所使用的算法，理解了nmb算法。