1 仿真应用流程概述

仿真应用流程是通过流程管理可视化编排系统进行组件编排，控制仿真软件完成和真实场景一样的流程模拟，实现模拟真实工业场景运行的一个编排流程。
下面将使用仿真分拣方块场景，带领大家体验仿真应用流程的开发过程，（该流程涉及仿真深度相机，仿真物体识别及坐标转换，仿真机械臂三个组件的开发与编排）。

2 仿真环境准备

系统地址：http://8.134.85.230/
账号：admin
密码：admin
远程地址：
ToDesk设备代码: 602 467 732
临时密码: j3scn6ci
点击链接直接进行远程控制：
https://wechat.todesk.com/invite-page?id=075MYbubndjjT47ByAPAo

3 开发步骤

下面开始开发该流程需要使用的三个组件(仿真深度相机，仿真物体识别及坐标转换，仿真机械臂)

3.1 登录

访问系统输入账号密码进入系统

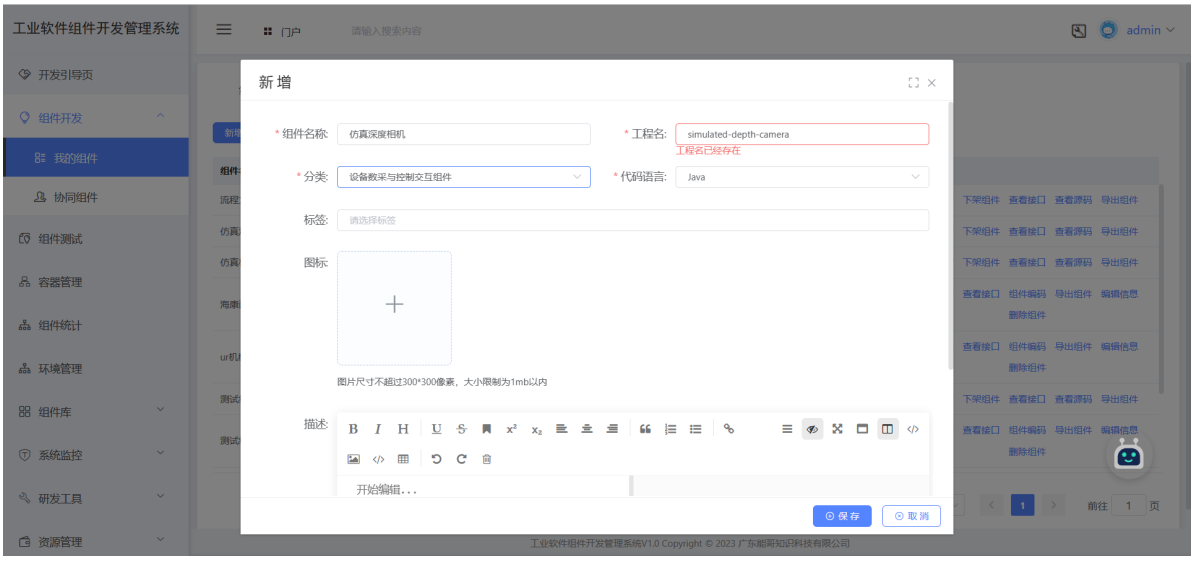
3.2 组件开发

接下来进行组件开发

3.2.1 开发仿真深度相机组件

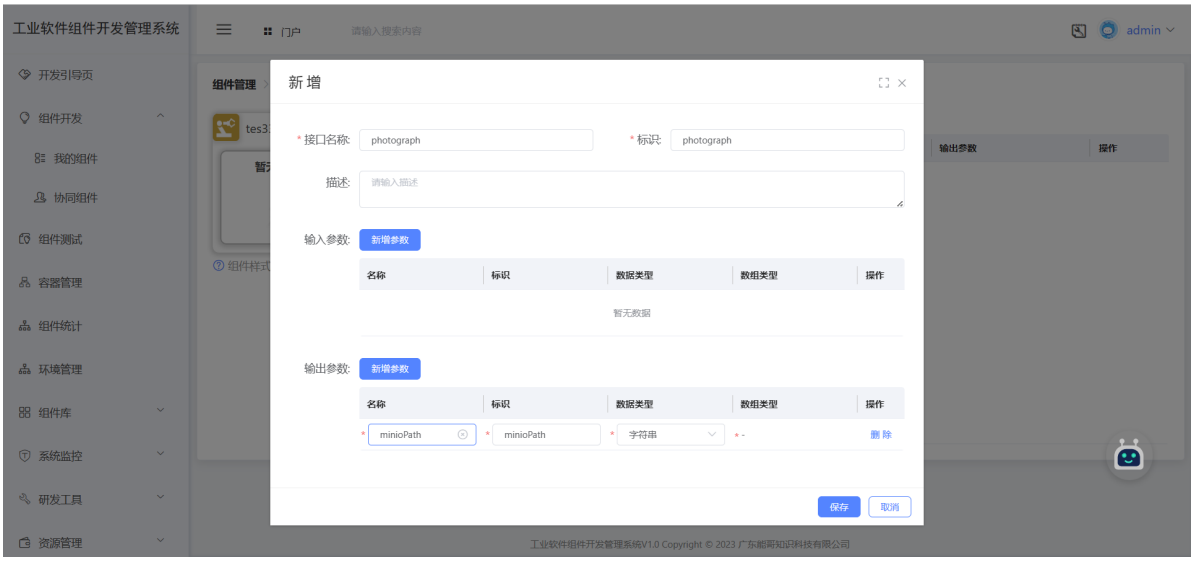
3.2.1.1 创建基本信息

名称和工程名重名时请自行1，2，3以此类推修改
组件名称：仿真深度相机
工程名：simulated-depth-camera
分类: 数据采集与控制交互组件
代码语言：Java

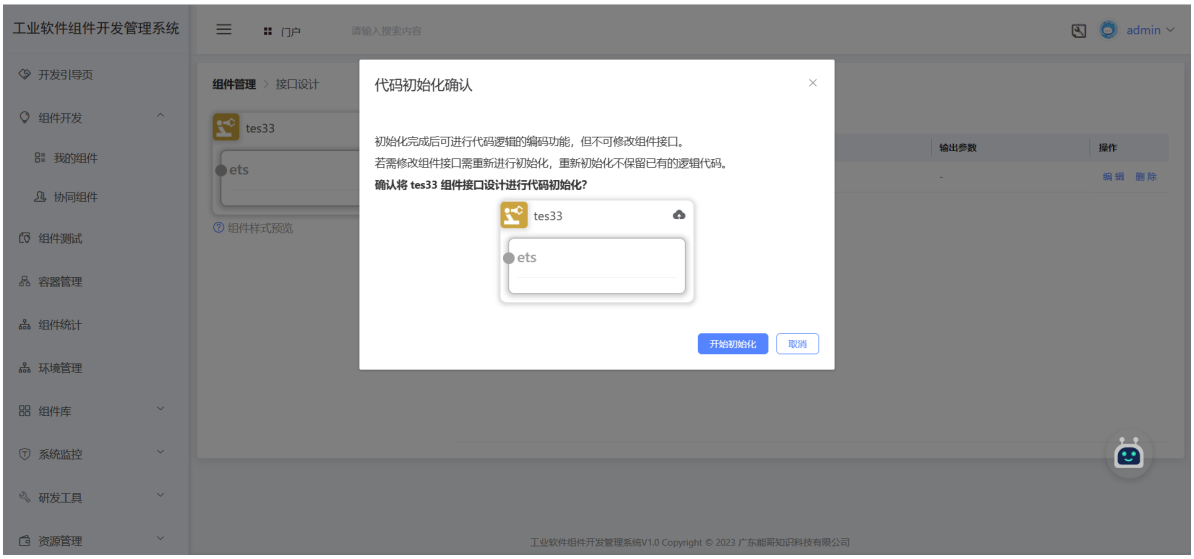


3.2.1.2 组件设计

新增一个操作：photograph
点击配置参数，新增输出参数:
名称，标识为：minioPath
参数类型为：字符串

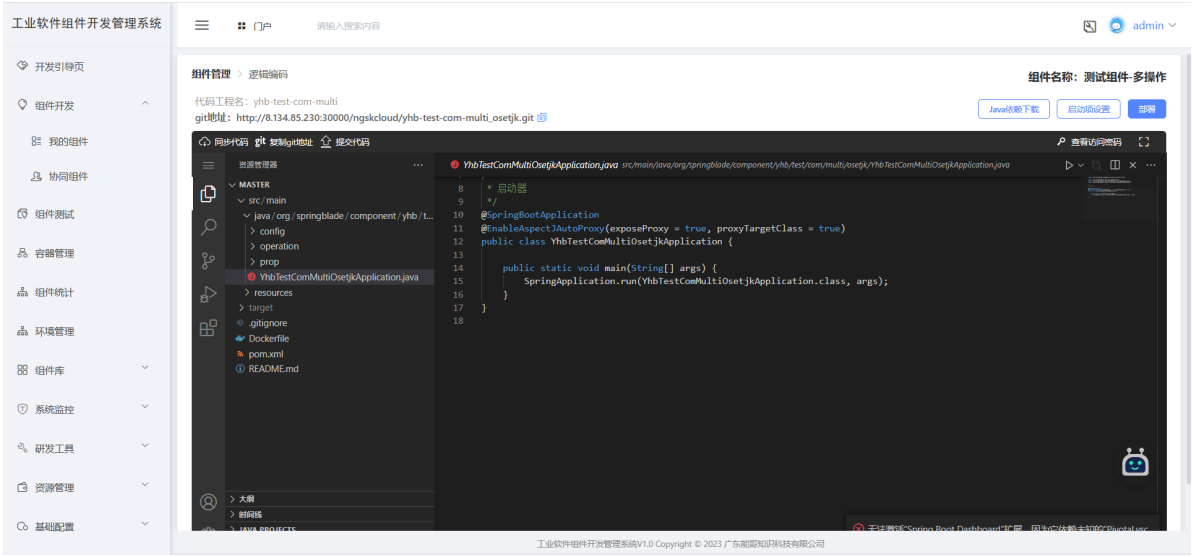


点击进行初始化



3.2.1.3 组件编码

在编码界面进行编码，首次
需要把代码中对应报名部分改为具体创建的包名



创建一个client文件夹
创建CameraResultHandler.java

```
Package 对应包名.client;
import 对应包名.domain.SimulatedResult;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.ReferenceCountUtil;
import lombok.extern.slf4j.Slf4j;
import org.springblade.isdp.starter.utils.JsonUtils;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;
import java.util.Objects;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.TimeUnit;
@Slf4j
@ChannelHandler.Sharable
@Component
public class CameraResultHandler extends ChannelInboundHandlerAdapter {
```

```
    private static final LinkedBlockingDeque<String> cameraResultQueue = new
LinkedBlockingDeque<>(10);
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object message) throws
Exception {
        final String msg = (String) message;
        try {
            log.debug("----------------------received json message --------------
-------");
            log.debug(msg);
            log.debug("----------------------------------------------------------
-------");
            final SimulatedResult simulatedResult = JsonUtils.parse(msg,
SimulatedResult.class);
            if (Objects.isNull(simulatedResult)) {
                log.error("Simulated camera response error,response json:{}",
msg);
                return;
            }
            if (simulatedResult.getType() == 2) {
                return;
            }
            final SimulatedResult.SimulatedData simulatedData =
simulatedResult.getData();
            // 相机base64图片
            final String scanResult = simulatedData.getScanResult();
            if (!StringUtils.hasText(scanResult)) {
                return;
            }
            try {
                cameraResultQueue.clear();
                cameraResultQueue.offerLast(scanResult, 2, TimeUnit.SECONDS);
            } catch (InterruptedException e) {
                log.error("the camera queue is full,please clear",
e.getCause());
            }
        } finally {
            ReferenceCountUtil.release(msg);
        }
    }
    public static String getCameraResult() throws Exception {
        final String cameraResult = cameraResultQueue.pollLast(5,
TimeUnit.SECONDS);
        if (!StringUtils.hasText(cameraResult)) {
            throw new Exception("get newest camera result failed, please
check!");
        }
        cameraResultQueue.clear();
        return cameraResult;
    }
}
```

创建：NettyClient.java文件

```
package 对应包名.client;
import 对应包名.config.CameraConfig;
import io.netty.bootstrap.Bootstrap;
```

```java
import io.netty.channel.Channel;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;
import javax.annotation.PreDestroy;
import static io.netty.channel.ChannelOption.SO_KEEPALIVE;
import static io.netty.channel.ChannelOption.TCP_NODELAY;
import static java.util.concurrent.TimeUnit.SECONDS;
@Slf4j
@Component
@RequiredArgsConstructor
public class NettyClient implements ApplicationRunner {
    private static final long RECONNECT_SECONDS = 2;
    private final NettyClientHandlerInitializer nettyClientHandlerInitializer;
    private final CameraConfig cameraConfig;
    /**
     * 线程组，用于客户端对服务端的链接、数据读写
     */
    private final EventLoopGroup eventGroup = new NioEventLoopGroup();
    /**
     * Netty Client Channel
     */
    private volatile Channel channel = null;
    @Override
    public void run(final ApplicationArguments args) {
        start();
    }
    /**
     * 启动 Netty Client
     */
    public synchronized void start() {
        // 创建 Bootstrap 对象，用于 Netty Client 启动
        if (channel != null && channel.isActive()) {
            return;
        }
        final Bootstrap bootstrap = new Bootstrap();
        final String ip = cameraConfig.getIp();
        final int port = Integer.parseInt(cameraConfig.getPort());
        // 设置 Bootstrap 的各种属性。
        bootstrap.group(eventGroup) // 设置一个 EventLoopGroup 对象
                .channel(NioSocketChannel.class)  // 指定 Channel 为客户端
NioSocketChannel
                .remoteAddress(ip, port) // 指定链接服务器的地址
                .option(SO_KEEPALIVE, true) // TCP Keepalive 机制，实现 TCP 层级的心
跳保活功能
                .option(TCP_NODELAY, true) // 允许较小的数据包的发送，降低延迟
                .handler(nettyClientHandlerInitializer);
        // 链接服务器，并异步等待成功，即启动客户端
        bootstrap.connect().addListener((ChannelFutureListener) future -> {
            // 连接失败
            if (!future.isSuccess()) {
                log.error("Netty 连接服务器 {}:{} 失败!", ip, port);
```

```
                reconnect();
                return;
            }
            // 连接成功
            channel = future.channel();
            log.info("Netty 连接服务器 {}:{} 成功!", ip, port);
        });
    }
    public void reconnect() {
        eventGroup.schedule(() -> {
            log.warn("RECONNECTING");
            start();
        }, RECONNECT_SECONDS, SECONDS);
        log.warn("Netty Client 将在 {} 秒后将发起重连!", RECONNECT_SECONDS);
    }
    public void shutdownChannel() {
        // 关闭 Netty Client
        if (channel != null) {
            channel.close();
        }
        channel = null;
    }
    /**
     * 关闭 Netty Server
     */
    @PreDestroy
    public void shutdown() {
        shutdownChannel();
        // 优雅关闭一个 EventLoopGroup 对象
        eventGroup.shutdownGracefully();
    }
    /**
     * @param message 消息
     *                <p>
     *                发送消息
     */
    public void send(String message) throws Exception {
        if (channel == null) {
            throw new Exception("连接不存在");
        }
        if (!channel.isActive()) {
            throw new Exception("连接断开");
        }
        // 发送消息
        channel.writeAndFlush(message);
        log.info("成功发送消息:{}", message);
    }
    public Channel getChannel() {
        return channel;
    }
}
```

创建：NettyClientHandler.java文件

```
package 对应包名.client;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
```

```
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.handler.timeout.IdleStateEvent;
import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;
import java.util.Date;
@Slf4j
@Component
@ChannelHandler.Sharable
public class NettyClientHandler extends ChannelInboundHandlerAdapter {
    // 引入客户端实现重连
    private final NettyClient nettyClient;
    public NettyClientHandler(@Lazy final NettyClient nettyClient) {
        this.nettyClient = nettyClient;
    }
    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        log.error("断开连接, 时间:" + new Date());
        // 发起重连
        nettyClient.reconnect();
        // 继续触发事件
        super.channelInactive(ctx);
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        log.error("Channel:{} exception!", ctx.channel().id(), cause);
        ctx.channel().close();
    }
    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object event)
throws Exception {
        if (event instanceof IdleStateEvent) {
            log.debug("监听到IdleStateEvent事件!成功发送心跳消息!");
            log.debug(String.valueOf(event.getClass()));
            nettyClient.shutdownChannel();
        }
        super.userEventTriggered(ctx, event);
    }
}
```

创建：NettyClientHandlerInitializer.java 文件

```
package 对应包名.client;
import io.netty.channel.Channel;
import io.netty.channel.ChannelInitializer;
import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;
import io.netty.util.CharsetUtil;
import io.netty.util.ResourceLeakDetector;
import org.springframework.stereotype.Component;
@Component
public class NettyClientHandlerInitializer extends ChannelInitializer<Channel> {
    private final NettyClientHandler nettyClientHandler;
    public NettyClientHandlerInitializer(final NettyClientHandler
nettyClientHandler) {
        this.nettyClientHandler = nettyClientHandler;
```

```
    }
    @Override
    protected void initChannel(Channel ch) {
        ch.pipeline()
                // 解码器
                .addLast(new LineBasedFrameDecoder(104857600))
                .addLast(new StringDecoder())
                .addLast(new StringEncoder(CharsetUtil.UTF_8))
                .addLast(new CameraResultHandler())
                // 客户端处理器
                .addLast(nettyClientHandler);
        ResourceLeakDetector.setLevel(ResourceLeakDetector.Level.ADVANCED);
    }
}
```

在原有的config文件下

创建CameraConfig.java文件

```
package 对应包名.config;
import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
@Data
@ConfigurationProperties("camera")
@Configuration
public class CameraConfig {
    private String ip;
    private String port;
}
```

创建MinioConfig.java文件

```
package 对应包名.config;
import io.minio.MinioClient;
import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Data
@Configuration
@ConfigurationProperties(prefix = "minio")
public class MinioConfig {
    private String endpoint;
    private String accessKey;
    private String secretKey;
    private String bucketName;
    @Bean
    public MinioClient minioClient() {
        return MinioClient.builder()
                .endpoint(endpoint)
                .credentials(accessKey, secretKey)
                .build();
    }
}
```

创建一个domain文件夹
创建SimulatedResult.java文件

```
package 对应包名.domain;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.Data;
@Data
public class SimulatedResult {
    private Integer type;
    private SimulatedData data;
    @Data
    public static class SimulatedData {
        @JsonProperty("ScanResult")
        private String scanResult;
    }
}
```

创建uitl文件夹
创建MinioUtils.java文件

```
package 对应包名.util;
import 对应包名.config.MinioConfig;
import com.alibaba.nacos.common.utils.StringUtils;
import io.minio.*;
import java.io.ByteArrayInputStream;
public class MinioUtils {
    /**
     * 文件上传 -本地文件
     *
     * @param minioClient minio 客户端
     * @param minioConfig minio配置
     * @param objectName  文件名
     * @param imageBuffer 图片缓冲区
     */
    public static String uploadFile(MinioClient minioClient, MinioConfig
minioConfig, String objectName, byte[] imageBuffer) throws Exception {
        final String bucketName = minioConfig.getBucketName();
        if (StringUtils.isEmpty(bucketName)) {
            throw new Exception("桶名不能为空");
        }
        if (!checkBucketExist(minioClient, bucketName)) {
            createBucket(minioClient, bucketName);
        }
        try (ByteArrayInputStream bis = new ByteArrayInputStream(imageBuffer)) {
            minioClient.putObject(
                    PutObjectArgs.builder()
                            .bucket(bucketName)
                            .object(objectName)//文件存储在minio中的名字
                            .stream(bis, bis.available(), -1)//上传本地文件存储的路
径
                            .build());
        }
        if (!getBucketFileExist(minioClient, objectName, bucketName)) {
            throw new Exception("minio图片保存失败");
        }
```

```
        return minioConfig.getEndpoint() + "/" + minioConfig.getBucketName() +
"/" + objectName;
    }
    /**
     * 创建桶
     *
     * @param minioClient minio 客户端
     * @param bucketName  桶名称
     */
    private static void createBucket(MinioClient minioClient, String bucketName)
throws Exception {
        try {

 minioClient.makeBucket(MakeBucketArgs.builder().bucket(bucketName).build());
        } catch (Exception e) {
            throw new Exception(e.getMessage(), e.getCause());
        }
    }
    /**
     * 检查桶是否存在
     *
     * @param bucketName 桶名称
     * @return boolean true-存在 false-不存在
     */
    private static boolean checkBucketExist(MinioClient minioClient, String
bucketName) throws Exception {
        if (!StringUtils.hasLength(bucketName)) {
            throw new Exception("检测桶的时候，桶名不能为空！");
        }
        try {
            return
minioClient.bucketExists(BucketExistsArgs.builder().bucket(bucketName).build());
        } catch (Exception e) {
            throw new Exception();
        }
    }
    /**
     * 检测某个桶内是否存在某个文件
     *
     * @param objectName 文件名称
     * @param bucketName 桶名称
     */
    private static boolean getBucketFileExist(MinioClient minioClient, String
objectName, String bucketName) throws Exception {
        if (!StringUtils.hasLength(objectName) ||
!StringUtils.hasLength(bucketName)) {
            throw new Exception("文件名和桶名不能为空！");
        }
        try {
            // 判断文件是否存在
            return
minioClient.bucketExists(BucketExistsArgs.builder().bucket(bucketName).build())
&&

 minioClient.statObject(StatObjectArgs.builder().bucket(bucketName).object(objec
tName).build()) != null;
        } catch (Exception e) {
            throw new Exception(e.getMessage(), e);
```

```
        }
    }
}
```

在operation 文件夹的PhotographOperation.java文件，编写逻辑代码

```
package 对应包名.operation;
import io.minio.MinioClient;
import lombok.RequiredArgsConstructor;
import org.bouncycastle.util.encoders.Base64;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import 对应包名.client.CameraResultHandler;
import 对应包名.client.NettyClient;
import 对应包名.config.MinioConfig;
import 对应包名.config.OperationConfiguration;
import 对应包名.prop.PhotographParam;
import 对应包名.prop.PhotographResponse;
import 对应包名.util.MinioUtils;
import org.springblade.isdp.starter.document.annotation.IsdpComponent;
import
org.springblade.isdp.starter.document.operation.OperationHandlerInterface;
import org.springblade.isdp.starter.document.operation.OperationRequest;
import org.springblade.isdp.starter.document.operation.OperationResponse;
import org.springblade.isdp.starter.utils.JsonUtils;
import org.springframework.core.env.Environment;
import java.util.UUID;
/**
 * PhotographOperation 操作处理类
 * <p>
 * 组件将通过 {@linkplain OperationConfiguration#globalOperationRoute()} 将本类注册
绑定到对应主题上；
 * <h3>使用方法</h3>
 * 默认只需要补充 {@link OperationHandlerInterface#handler(Object)} 方法即可；<br/>
 * 方法将接收String类型数据，为MQTT消息内data字段Json化数据，
 * 如果需要使用其他数据，可自定义实现 {@link
OperationHandlerInterface#convert(OperationRequest)}；<br/>
 * 方法将返回 <b>{@code OperationHandlerInterface<T,R>}</b> 中 R 的类型，组件内部将返
回数据Json序列化后发送MQTT；<br/>
 * <p>
 * 拓展点：<br>
 * 1、{@link OperationHandlerInterface#convert(OperationRequest)} convert()方法可
将MQTT数据进行直接处理
 * 2、{@link OperationHandlerInterface#listenerHandler(String, OperationRequest,
Environment)}
 * 方法可忽略组件内部处理逻辑而使用自定义逻辑，包括不限于MQTT消息返回
 */
@IsdpComponent
@RequiredArgsConstructor
public class PhotographOperation implements
OperationHandlerInterface<PhotographParam,
OperationResponse<PhotographResponse>> {
    private static final Logger log =
LoggerFactory.getLogger(PhotographOperation.class);
    private final NettyClient client;
    private final MinioClient minioClient;
    private final MinioConfig minioConfig;
```

```
    private static final String MINIO_PARENT_DIR = "componentImage";
    private static final String MINIO_SEPARATOR = "/";
    private static final String KEY_PREFIX = "ng";
    private static final String COLON = ":";
    @Override
    public PhotographParam convert(final OperationRequest<?> request) {
        return JsonUtils.parse(JsonUtils.toJson(request.getData()),
PhotographParam.class);
    }
    @Override
    public OperationResponse<PhotographResponse> handler(final PhotographParam
param) throws Exception {
        // 1. 拍照请求
        client.send("{\"type\":2,\"data\":{\"Scan\":false}}\n");
        client.send("{\"type\":2,\"data\":{\"Scan\":true}}\n");
        //2. 读取请求
        client.send("{\"type\":1,\"data\":{\"ScanResult\":\"ScanResult\"}}\n");
        client.send("{\"type\":2,\"data\":{\"Scan\":false}}\n");
        //3. 获取最新拍照结果
        final String cameraResult = CameraResultHandler.getCameraResult();
        //4. base64转byte存到redis
        final String key;
        try {
            key = MinioUtils.uploadFile(minioClient, minioConfig,
generateMinioObjectName(), Base64.decode(cameraResult));
        } catch (Exception e) {
            throw new Exception("minio upload failed,cause:", e.getCause());
        }
        final PhotographResponse response = new PhotographResponse();
        response.setKey(key);
        return OperationResponse.of(response);
    }
    private static String generateMinioObjectName() {
        return MINIO_PARENT_DIR + MINIO_SEPARATOR + KEY_PREFIX + COLON +
"simulated-depth-camera" + COLON + UUID.randomUUID() + ".bmp";
    }
}
```

在pom.xml文件添加依赖下面依赖

```
<!-- netty依赖 -->
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-all</artifactId>
  <version>4.1.91.Final</version>
</dependency>
<!-- minio driver-->
<dependency>
  <groupId>io.minio</groupId>
  <artifactId>minio</artifactId>
  <version>8.4.3</version>
</dependency>
```

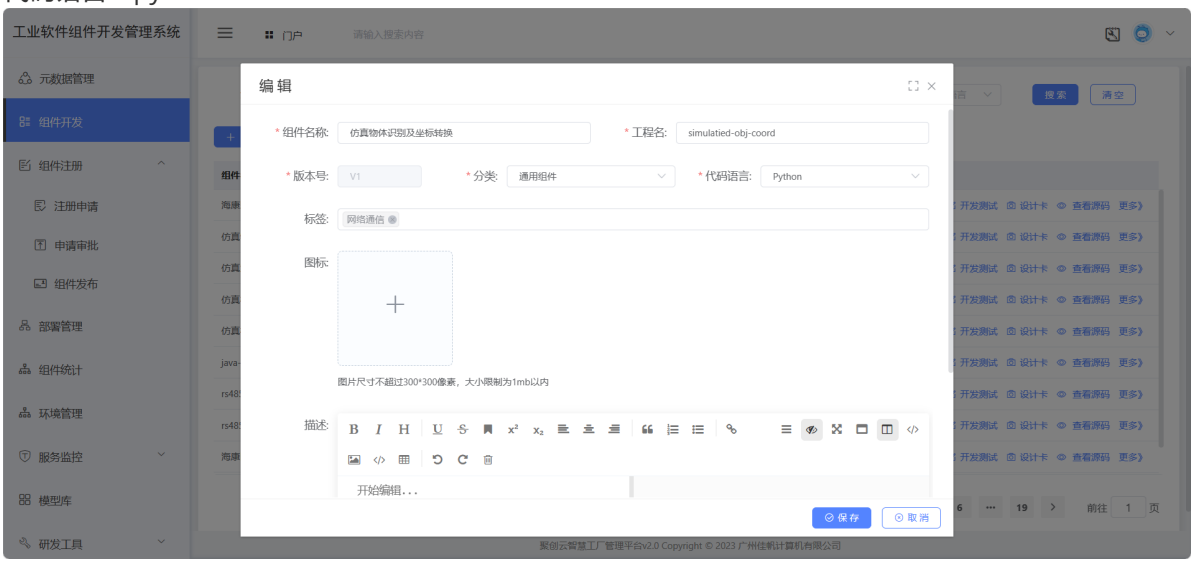
```
MinioConfig.java   pom.xml
pom.xml
        <version>3.0.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.7.9</version>
    </dependency>
    <!-- netty依赖 -->
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.91.Final</version>
    </dependency>
    <!-- minio driver-->
    <dependency>
        <groupId>io.minio</groupId>
        <artifactId>minio</artifactId>
        <version>8.4.3</version>
    </dependency>
</dependencies>

<build>
```

3.2.2 开发仿真物体识别及坐标转换

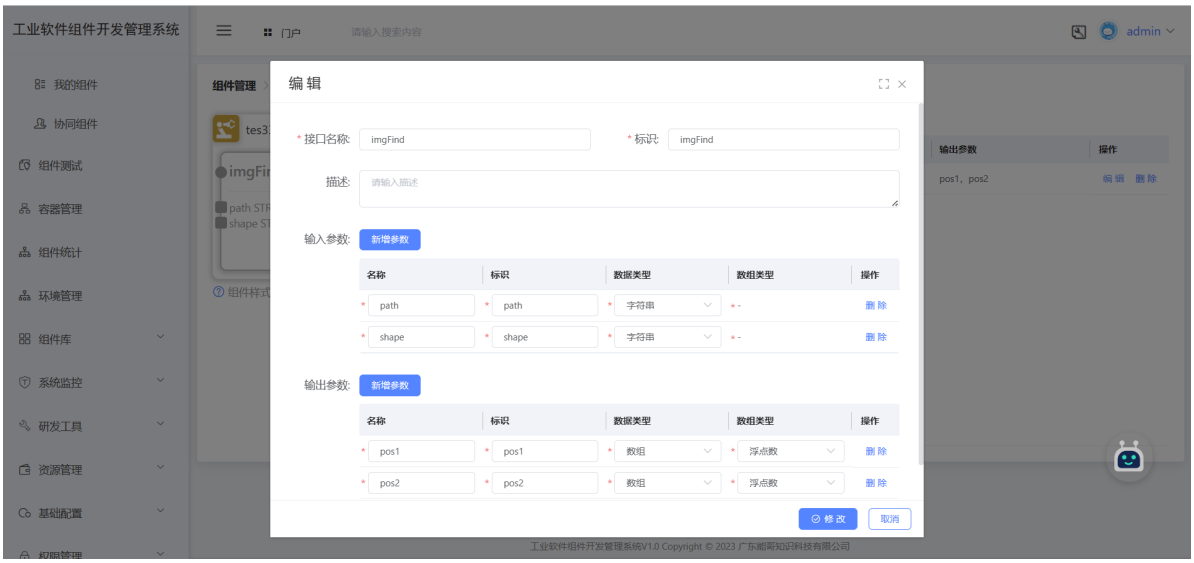
3.2.2.1 创建基本信息

组件名称：仿真物体识别及坐标转换
工程名：simulatied-obj-coord
分类：视觉AI组件
代码语言：python



3.2.2.2 组件设计

新增操作：imgFind
配置参数
输入参数：
名称，标识：path，数据类型：字符串
名称，标识：shape，数据类型：字符串
输出参数：
名称，标识：pos1，数据类型：数组，数组类型：浮点数
名称，标识：pos2，数据类型：数组，数组类型：浮点数



然后进行初始化，进入编码界面

3.2.2.3 组件编码

编写functions.py文件

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import cv2 as cv
import numpy as np
import server
def getRedisConn():
    return server.redisConn()
def releaseRedisConn(conn):
    conn.close()

def classify_Square(contour):
    # 使用轮廓逼近函数，将轮廓近似为多边形
    epsilon = 0.04 * cv.arcLength(contour, True)
    approx = cv.approxPolyDP(contour, epsilon, True)
    # 获取多边形的边数
    sides = len(approx)
    # 根据边数进行形状分类
    if sides == 4:
        return True
    return False
def classify_Hexagon(contour):
    # 使用轮廓逼近函数，将轮廓近似为多边形
    epsilon = 0.04 * cv.arcLength(contour, True)
    approx = cv.approxPolyDP(contour, epsilon, True)
    # 获取多边形的边数
```

```
    sides = len(approx)
    if sides == 6:
        return True
    return False
def classify_Circle(contour):
    epsilon = 0.04 * cv.arcLength(contour, True)
    approx = cv.approxPolyDP(contour, epsilon, True)
    circularity = 4 * np.pi * cv.contourArea(contour) / (cv.arcLength(contour,
True) ** 2)
    epsilon = 0.04 * cv.arcLength(contour, True)
    approx = cv.approxPolyDP(contour, epsilon, True)
    # 获取多边形的边数
    sides = len(approx)
    if circularity >= 0.85 and sides > 6:
        return True
    return False
def classify_Rectangle(contour):
    # 获取轮廓的近似多边形
    epsilon = 0.04 * cv.arcLength(contour, True)
    approx = cv.approxPolyDP(contour, epsilon, True)
    # 如果近似多边形有四个顶点，认为是矩形
    if len(approx) == 4:
        # 计算多边形的内角
        angles = []
        for i in range(4):
            p1 = approx[i][0]
            p2 = approx[(i + 1) % 4][0]
            p3 = approx[(i + 2) % 4][0]
            angle = np.degrees(np.arctan2(p3[1] - p2[1], p3[0] - p2[0]) -
np.arctan2(p1[1] - p2[1], p1[0] - p2[0]))
            angle = (angle + 360) % 360  # 确保角度在0到360度之间
            angles.append(angle)
        # 检查内角是否接近90度
        if all(80 <= angle <= 100 for angle in angles):
            return True
    return False
def preprocess_contour(contour_image):
    # 边缘检测
    edges = cv.Canny(contour_image, threshold1=30, threshold2=100)
    # 去噪
    blurred = cv.GaussianBlur(edges, (5, 5), 0)
    # 轮廓提取
    contours, _ = cv.findContours(blurred, cv.RETR_EXTERNAL,
cv.CHAIN_APPROX_SIMPLE)
    # 轮廓近似
    simplified_contours = []
    for contour in contours:
        epsilon = 0.02 * cv.arcLength(contour, True)
        approx = cv.approxPolyDP(contour, epsilon, True)
        simplified_contours.append(approx)
    return simplified_contours

def Blob_area_Limit(contours, area_min, area_max):
    contours_Limit = []
    area_List = []
    for cnt in contours:
        area = cv.contourArea(cnt)
        if area > area_min and area < area_max:
```

```
            contours_Limit.append(cnt)
            area_List.append(area)
    return contours_Limit, area_List
def robotcamchange(pixel_x, pixel_y, pixel_z):
    transform_matrix = np.array([
        [
            [-9.99353718e-01,  4.97718682e-03, -3.56002044e-02,
-3.37661928e+02],
            [-4.93836123e-05,  9.90176832e-01, 1.39820736e-01,  2.22597010e+02],
            [ 3.59464115e-02 , 1.39732130e-01, -9.89536653e-01,
 1.66154847e+04],
            [0, 0, 0, 1]
]
    ])
    transform_matrixT = np.linalg.inv(transform_matrix)
    camera_matrix = np.array([[2.23039874e+04, 0.00000000e+00, 2.55599469e+02],
 [0.00000000e+00, 2.23084372e+04, 2.55491691e+02],
 [0.00000000e+00, 0.00000000e+00, 1.00000000e+00]])
    camera_matrix2 = np.linalg.inv(camera_matrix)
    pixel_coordinates = np.array([[pixel_x * pixel_z], [pixel_y * pixel_z],
[pixel_z]])
    # pixel_coordinates = np.array([[pixel_x ], [pixel_y ], [pixel_z]])
    T2 = np.dot(camera_matrix2, pixel_coordinates)
    T3 = np.vstack([T2, [1]])
    robot_coordinates = np.dot(transform_matrixT, T3)
    x = float(robot_coordinates[0][0])
    y = float(robot_coordinates[0][1])
    z = float(robot_coordinates[0][2])
    data1 = [x/1000, y/1000-0.005, 0.150, 180, 0, 0]
    data2 = [x/1000, y/1000-0.005, -0.01, 180, 0, 0]
    return data1, data2
#
# if __name__ == '__main__':
#     print(robotcamchange(277,231,1.66154847e+04))
```

编写operations.py文件

```
# import urllib.request
import json
import cv2 as cv
import numpy as np
import redis
import functions
import requests
from functions import classify_Square, classify_Hexagon, classify_Circle,
classify_Rectangle, robotcamchange,preprocess_contour
min_contour_area = 1000
max_contour_area = 5000
def imgFind(data):
    shape = data["shape"]
    image_path = data["path"]
    if image_path.startswith("http"):
        # 如果图像路径是一个 URL，则下载图像并保存到本地
        response = requests.get(image_path)
        image_data = response.content
        image_np = np.frombuffer(image_data, np.uint8)
        img_target_np = cv.imdecode(image_np, cv.IMREAD_COLOR)
    else:
        # 如果图像路径是本地文件路径，则直接读取图像
        img_target_np = cv.imread(image_path)
```

```
    # conn = functions.getRedisConn()
    # data = conn.get(key)
    # functions.releaseRedisConn(conn)
    # img_target_np = cv.imdecode(np.frombuffer(data, np.uint8),
cv.IMREAD_COLOR)
    # cv.imshow("Image from Redis", img_target_np)
    # cv.waitKey(0)
    # cv.destroyAllWindows()
    # 转换为HSV颜色空间
    img_target_hsv = cv.cvtColor(img_target_np, cv.COLOR_BGR2HSV)
    lower_red = np.array([0, 0, 46])
    upper_red = np.array([180, 43, 220])
    # 创建红色掩膜
    red_mask = cv.inRange(img_target_hsv, lower_red, upper_red)
    # 寻找白色物块的轮廓
    contours = preprocess_contour(red_mask)
    for i, contour in enumerate(contours):
        if i < len(contours):  # 检查轮廓列表是否足够长
            ret = False
            if shape == "square":
                ret = classify_Square(contour)
            if shape == "circle":
                ret = classify_Circle(contour)
            if shape == "hexagon":
                ret = classify_Hexagon(contour)
            if shape == "rectangle":
                ret = classify_Rectangle(contour)
            if ret:
                area = cv.contourArea(contour)
                print(area)
                M = cv.moments(contour)
                if area > min_contour_area and area < max_contour_area:
                    if M["m00"] != 0:
                        cX = int(M["m10"] / M["m00"])
                        cY = int(M["m01"] / M["m00"])
                        print(f"Center (X, Y): ({cX}, {cY}), Shape: {shape}")
                        # cv.circle(img_target_np, (cX, cY), 5, (0, 255, 0), -1)
                        # cv.putText(img_target_np, f"Center (X, Y): ({cX},
{cY})", (cX - 50, cY - 10),
                        #            cv.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0),
2)
                        # 显示识别出的形状
                        # cv.drawContours(img_target_np, [contour], -1, (0, 255,
0), 2)
                        # cv.imshow("Detected Objects", img_target_np)
                        # cv.waitKey(0)  # 等待用户按下任意键
                        # cv.destroyAllWindows()
                        pixel_x = cX
                        pixel_y = cY
                        pixel_z = 1.66154847e+04
                        pos1, pos2 = robotcamchange(pixel_x, pixel_y, pixel_z)
                        return {"event":"analyseDone", "data": {"pos1": pos1,
"pos2": pos2}, "msg": "recognition_success", "code": 200}
    return {"event":"noneShape", "msg": "recognition_fail", "code": 200}
# if __name__ == '__main__':
#     print(imgfind({"shape": "square", "path": "./5.bmp"}))
```

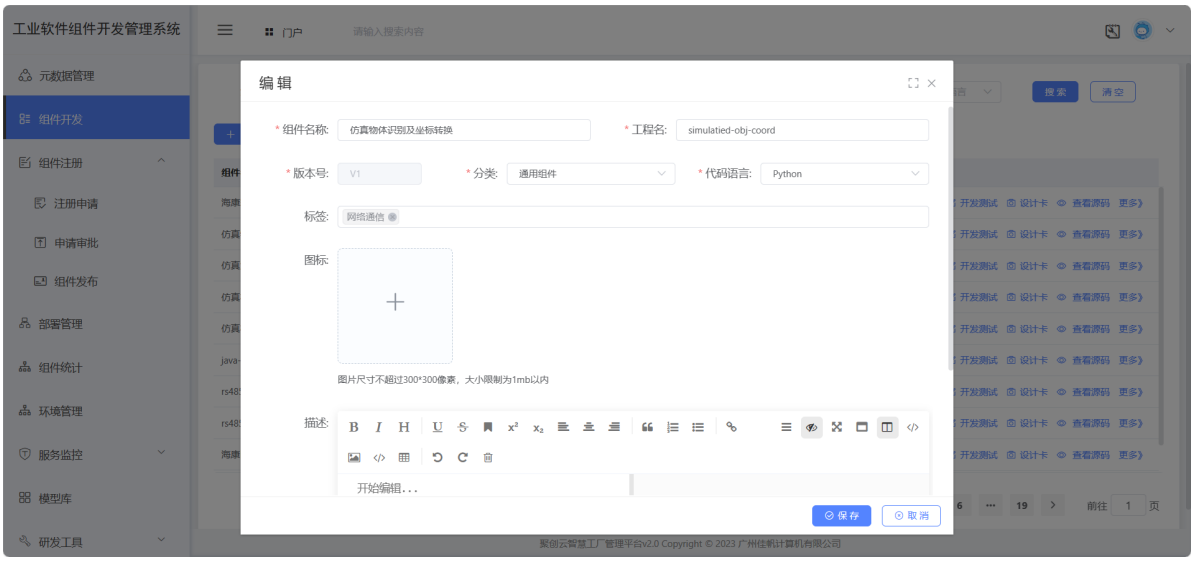
编写requirements.txt文件

```
nacos_sdk_python==0.1.12
paho_mqtt==1.6.1
PyYAML==6.0.1
opencv_python_headless==4.7.0.72
numpy==1.24.4
redis==5.0.1
```

3.2.3 开发仿真机械臂

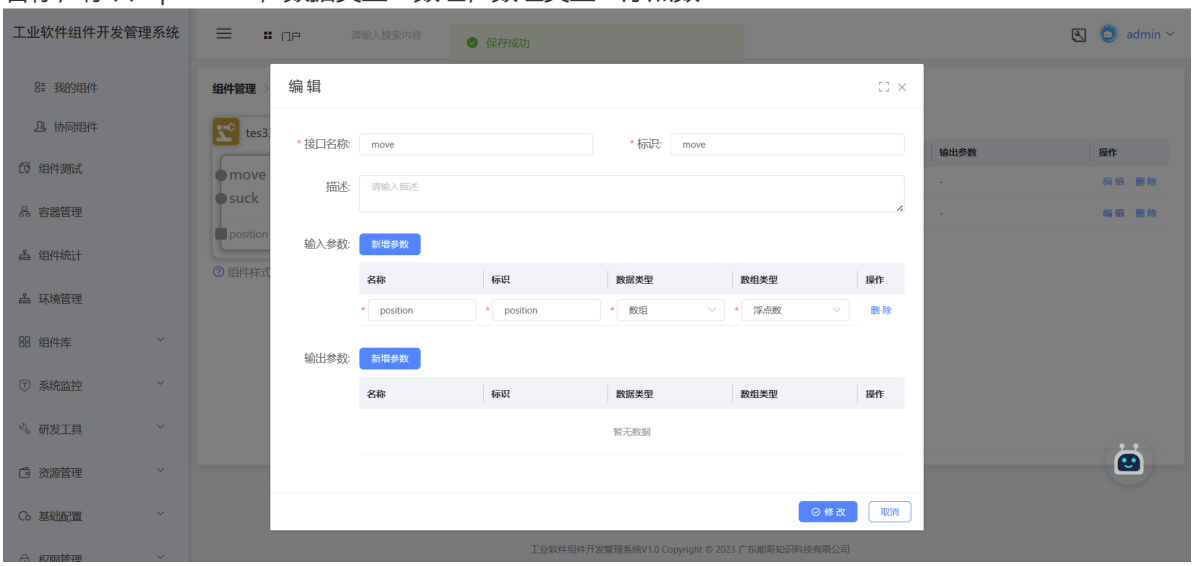
3.2.3.1 创建基本信息

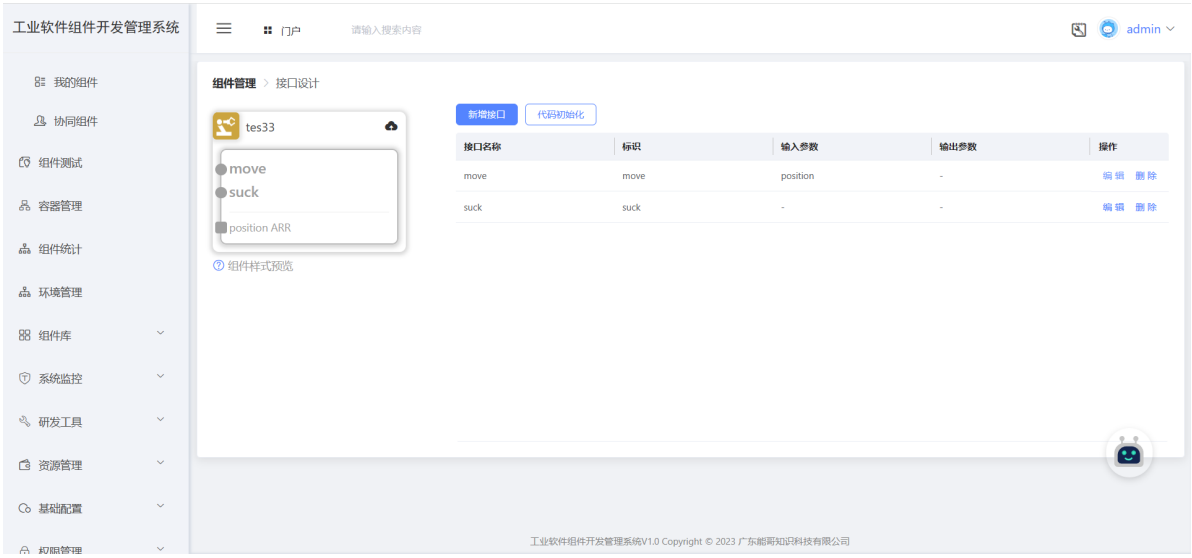
组件名称：仿真机械臂
工程名：simulated-mechanical-arm
分类：数据采集与控制交互组件
代码语言：Java



3.2.3.2 组件设计

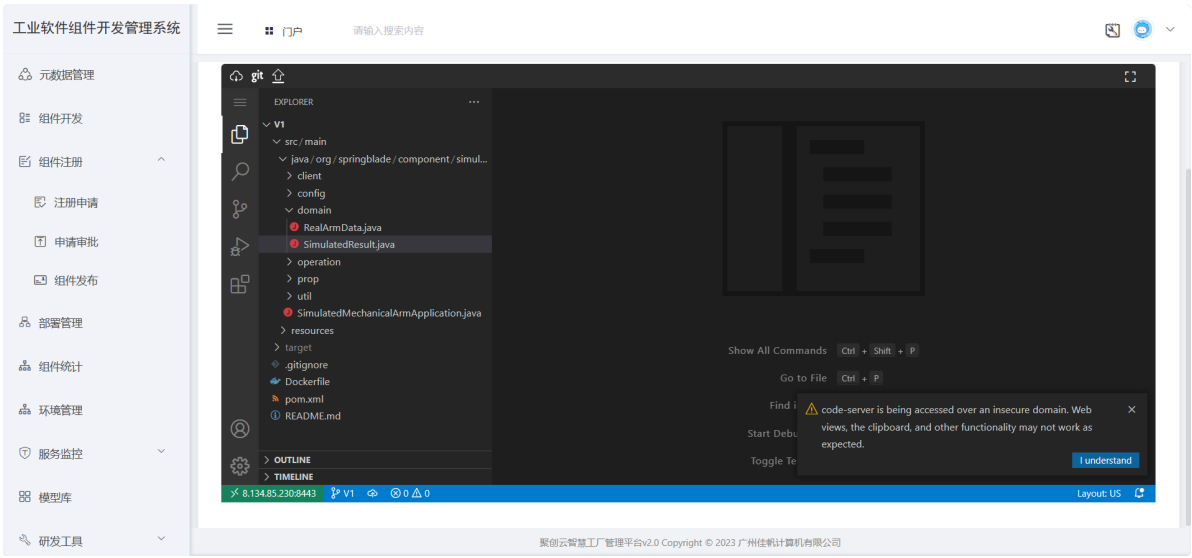
新增操作：move，suck，并给move操作配置参数
move输出参数配置：
名称，标识：position，数据类型：数组，数组类型：浮点数





3.2.3.3 组件编码

点击代码初始化进入编码
编写逻辑代码



创建client文件夹
创建ArmResultHandler.java文件：

```
package 对应包名.client;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.ReferenceCountUtil;
import lombok.extern.slf4j.Slf4j;
import 对应包名.domain.RealArmData;
import 对应包名.domain.SimulatedResult;
import org.springblade.isdp.starter.utils.JsonUtils;
import org.springframework.stereotype.Component;
import java.util.Objects;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.TimeUnit;
@Slf4j
@ChannelHandler.Sharable
@Component
public class ArmResultHandler extends ChannelInboundHandlerAdapter {
    private static final LinkedBlockingDeque<RealArmData> armDataQueue = new
LinkedBlockingDeque<>(100);
```

```
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object message) throws
Exception {
        final String msg = (String) message;
        try {
            log.debug("----------------------received json message --------------
-------");
            log.debug(msg);
            log.debug("------------------------------------------------------------
-------");
            final SimulatedResult simulatedResult = JsonUtils.parse(msg,
SimulatedResult.class);
            if (Objects.isNull(simulatedResult)) {
                log.error("Simulated camera response error,response json:{}",
msg);
                return;
            }
            if (simulatedResult.getType() == 2) {
                return;
            }
            final SimulatedResult.ArmData armData = simulatedResult.getData();
            if (Objects.isNull(armData)) {
                log.error("The arm Data is null, please check!");
                return;
            }
            final RealArmData realArmData = new RealArmData(armData);
            // 保存获取实时机械臂状态
            try {
                if (armDataQueue.size() >= 90) {
                    armDataQueue.clear();
                }
                armDataQueue.offerLast(realArmData, 2, TimeUnit.SECONDS);
            } catch (InterruptedException e) {
                log.error("the armData queue is full,please clear",
e.getCause());
            }
        } finally {
            ReferenceCountUtil.release(msg);
        }
    }
    public static RealArmData getRealArmData() throws Exception {
        try {
            final RealArmData realArmData = armDataQueue.pollLast(5,
TimeUnit.SECONDS);
            armDataQueue.clear();
            return realArmData;
        } catch (Exception e) {
            throw new Exception("Get arm Data failed,cause:", e.getCause());
        }
    }
}
```

创建NettyClient.java文件:

```
package 对应包名.client;
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
```

```java
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import 对应包名.config.ArmConfig;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;
import javax.annotation.PreDestroy;
import static io.netty.channel.ChannelOption.SO_KEEPALIVE;
import static io.netty.channel.ChannelOption.TCP_NODELAY;
import static java.util.concurrent.TimeUnit.SECONDS;
@Slf4j
@Component
@RequiredArgsConstructor
public class NettyClient implements ApplicationRunner {
    private static final long RECONNECT_SECONDS = 2;
    private final NettyClientHandlerInitializer nettyClientHandlerInitializer;
    private final ArmConfig armConfig;
    /**
     * 线程组，用于客户端对服务端的链接、数据读写
     */
    private final EventLoopGroup eventGroup = new NioEventLoopGroup();
    /**
     * Netty Client Channel
     */
    private volatile Channel channel = null;
    @Override
    public void run(final ApplicationArguments args) {
        start();
    }
    /**
     * 启动 Netty Client
     */
    public synchronized void start() {
        // 创建 Bootstrap 对象，用于 Netty Client 启动
        if (channel != null && channel.isActive()) {
            return;
        }
        final Bootstrap bootstrap = new Bootstrap();
        final String ip = armConfig.getIp();
        final int port = Integer.parseInt(armConfig.getPort());
        // 设置 Bootstrap 的各种属性。
        bootstrap.group(eventGroup) // 设置一个 EventLoopGroup 对象
                .channel(NioSocketChannel.class)  // 指定 Channel 为客户端
NioSocketChannel
                .remoteAddress(ip, port) // 指定链接服务器的地址
                .option(SO_KEEPALIVE, true) // TCP Keepalive 机制，实现 TCP 层级的心
跳保活功能
                .option(TCP_NODELAY, true) // 允许较小的数据包的发送，降低延迟
                .handler(nettyClientHandlerInitializer);
        // 链接服务器，并异步等待成功，即启动客户端
        bootstrap.connect().addListener((ChannelFutureListener) future -> {
            // 连接失败
            if (!future.isSuccess()) {
                log.error("Netty 连接服务器 {}:{} 失败!", ip, port);
```

```
                reconnect();
                return;
            }
            // 连接成功
            channel = future.channel();
            log.info("Netty 连接服务器 {}:{} 成功!", ip, port);
        });
    }
    public void reconnect() {
        eventGroup.schedule(() -> {
            log.warn("RECONNECTING");
            start();
        }, RECONNECT_SECONDS, SECONDS);
        log.warn("Netty Client 将在 {} 秒后将发起重连!", RECONNECT_SECONDS);
    }
    public void shutdownChannel() {
        // 关闭 Netty Client
        if (channel != null) {
            channel.close();
        }
        channel = null;
    }
    /**
     * 关闭 Netty Server
     */
    @PreDestroy
    public void shutdown() {
        shutdownChannel();
        // 优雅关闭一个 EventLoopGroup 对象
        eventGroup.shutdownGracefully();
    }
    /**
     * @param message 消息
     *                <p>
     *                发送消息
     */
    public void send(String message) throws Exception {
        if (channel == null) {
            throw new Exception("连接不存在");
        }
        if (!channel.isActive()) {
            throw new Exception("连接断开");
        }
        // 发送消息
        channel.writeAndFlush(message);
        log.info("成功发送消息:{}", message);
    }
    public Channel getChannel() {
        return channel;
    }
}
```

创建NettyClientHandler.java文件:

```
package 对应包名.client;
import io.netty.channel.ChannelHandler;
import io.netty.channel.ChannelHandlerContext;
```

```
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.handler.timeout.IdleStateEvent;
import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;
import java.util.Date;
@Slf4j
@Component
@ChannelHandler.Sharable
public class NettyClientHandler extends ChannelInboundHandlerAdapter {
    // 引入客户端实现重连
    private final NettyClient nettyClient;
    public NettyClientHandler(@Lazy final NettyClient nettyClient) {
        this.nettyClient = nettyClient;
    }
    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        log.error("断开连接, 时间:" + new Date());
        // 发起重连
        nettyClient.reconnect();
        // 继续触发事件
        super.channelInactive(ctx);
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        log.error("Channel:{} exception!", ctx.channel().id(), cause);
        ctx.channel().close();
    }
    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object event)
throws Exception {
        if (event instanceof IdleStateEvent) {
            log.debug("监听到IdleStateEvent事件!成功发送心跳消息!");
            log.debug(String.valueOf(event.getClass()));
            nettyClient.shutdownChannel();
        }
        super.userEventTriggered(ctx, event);
    }
}
```

创建NettyClientHandlerInitializer.java文件：

```
package 对应包名.client;
import io.netty.channel.Channel;
import io.netty.channel.ChannelInitializer;
import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;
import io.netty.util.CharsetUtil;
import io.netty.util.ResourceLeakDetector;
import org.springframework.stereotype.Component;
@Component
public class NettyClientHandlerInitializer extends ChannelInitializer<Channel> {
    private final NettyClientHandler nettyClientHandler;
    public NettyClientHandlerInitializer(final NettyClientHandler
nettyClientHandler) {
        this.nettyClientHandler = nettyClientHandler;
```

```
    }
    @Override
    protected void initChannel(Channel ch) {
        ch.pipeline()
                // 解码器
                .addLast(new LineBasedFrameDecoder(104857600))
                .addLast(new StringDecoder())
                .addLast(new StringEncoder(CharsetUtil.UTF_8))
                .addLast(new ArmResultHandler())
                // 客户端处理器
                .addLast(nettyClientHandler);
        ResourceLeakDetector.setLevel(ResourceLeakDetector.Level.ADVANCED);
    }
}
```

在config文件夹下
创建ArmConfig.java文件：

```
package 对应包名.config;
import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
@Data
@ConfigurationProperties("arm")
@Configuration
public class ArmConfig {
    private String ip;
    private String port;
    @Data
    @ConfigurationProperties("arm.deviation")
    @Configuration
    public static class Deviation {
        private Double x;
        private Double y;
        private Double z;
    }
}
```

创建ArmSchedule.java文件：

```
package 对应包名.config;
import lombok.RequiredArgsConstructor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import 对应包名.client.NettyClient;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
@RequiredArgsConstructor
@EnableScheduling
@EnableAsync
@Component
public class ArmSchedule {
    private final NettyClient client;
```

```
    private static final Logger log =
LoggerFactory.getLogger(ArmSchedule.class);
    @Scheduled(initialDelay = 2000, fixedDelay = 500)
    @Async
    public void timedSendGetTcpPosition() {
        try {
            client.send("{\"type\":1,\"data\":
{\"TcpPosition\":\"TcpPosition\",\"SuckDone\":\"SuckDone\"}}\n");
        } catch (Exception e) {
            log.error("Timed arm schedule error ,please check, caused: {}",
e.getMessage());
        }
    }
}
```

创建domain文件夹
创建RealArmData.java文件：

```
package 对应包名.domain;
import lombok.AllArgsConstructor;
import lombok.Data;
@Data
@AllArgsConstructor
public class RealArmData {
    private Double x;
    private Double y;
    private Double z;
    private Boolean isSuck;
    public RealArmData(SimulatedResult.ArmData armData) {
        final String position = armData.getTcpPosition();
        final Boolean suckDone = armData.getSuckDone();
        // 移除括号，并按逗号分割字符串
        final String[] values = position.replaceAll("[()]", "").split(",");
        // 解析并设置x、y、z的值
        this.x = Double.parseDouble(values[0].trim());
        this.y = Double.parseDouble(values[1].trim());
        this.z = Double.parseDouble(values[2].trim());
        this.isSuck = suckDone;
    }
}
```

创建SimulatedResult.java文件：

```
package 对应包名.domain;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.Data;
@Data
public class SimulatedResult {
    private Integer type;
    private ArmData data;
    @Data
    public static class ArmData {
        @JsonProperty("TcpPosition")
        private String tcpPosition;
        @JsonProperty("SuckDone")
        private Boolean suckDone;
```

```
    }
}
```

创建uitl文件夹
创建UrRobotUtil.java文件：

```
package 对应包名.util;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import 对应包名.client.ArmResultHandler;
import 对应包名.config.ArmConfig;
import 对应包名.domain.RealArmData;
import java.util.List;
import java.util.Objects;
public class UrRobotUtil {
    private static final Logger log =
LoggerFactory.getLogger(UrRobotUtil.class);
    /**
     * 判断坐标是否正确
     *
     * @param position 机械臂末端坐标集合
     */
    public static void judgePosition(final List<Double> position) throws
Exception {
        if (Objects.isNull(position) || position.size() < 6) {
            throw new Exception("position坐标不正确");
        }
        if (Objects.isNull(position.get(0)) || Objects.isNull(position.get(1))
|| Objects.isNull(position.get(2)) || Objects.isNull(position.get(3)) ||
Objects.isNull(position.get(4)) || Objects.isNull(position.get(5))) {
            throw new Exception("position坐标不正确");
        }
    }
    /**
     * 根据坐标和额外参数拼接命令
     *
     * @param position 机械臂末端坐标集合
     * @return 拼接后的命令
     */
    public static String moveCommand(final List<Double> position) {
        final Double x = position.get(0);
        final Double y = position.get(1);
        final Double z = position.get(2);
        final Double rx = position.get(3);
        final Double ry = position.get(4);
        final Double rz = position.get(5);
        return String.format("{\"type\":2,\"data\":{\"TargetPosition\":\"
(%s,%s,%s, %s, %s, %s)\"}}\n", x, y, z, rx, ry, rz);
    }
    /**
     * 根据目标坐标和偏差 等待机械臂是否到达此位置
     *
     * @param targetPosition 机械臂目标末端坐标
     * @param deviation      允许坐标偏差
     */
    public static void waitToTarget(final List<Double> targetPosition, final
ArmConfig.Deviation deviation) throws Exception {
```

```
        int count = 1;
        final Double targetX = targetPosition.get(0);
        final Double targetY = targetPosition.get(1);
        final Double targetZ = targetPosition.get(2);
        while (true) {
            final RealArmData realArmData = ArmResultHandler.getRealArmData();
            final Double actualX = realArmData.getX();
            final Double actualY = realArmData.getY();
            final Double actualZ = realArmData.getZ();
            // x轴方差
            final Double deviationX = deviation.getX();
            final double xDiff = targetX - actualX;
            final boolean isXCorrect = (-1 * deviationX) < xDiff && xDiff <
deviationX;
            // y轴方差
            final Double deviationY = deviation.getY();
            final double yDiff = targetY - actualY;
            final boolean isYCorrect = (-1 * deviationY) < yDiff && yDiff <
deviationY;
            // z轴方差
            final Double deviationZ = deviation.getZ();
            final double zDiff = targetZ - actualZ;
            final boolean isZCorrect = (-1 * deviationZ) < zDiff && zDiff <
deviationZ;
            if (isXCorrect && isYCorrect && isZCorrect) {
                break;
            }
            log.debug("x偏差:" + xDiff + ", y偏差:" + yDiff + ", z偏差: " + zDiff);
            count++;
            if (count >= 40) {
                throw new Exception("机械臂运动超时");
            }
            Thread.sleep(500);
        }
    }
}
```

在operation文件夹
编写MoveOperation.java文件逻辑

```
package 对应包名.operation;
import lombok.RequiredArgsConstructor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import 对应包名.client.NettyClient;
import 对应包名.config.ArmConfig;
import 对应包名.config.OperationConfiguration;
import 对应包名.prop.*;
import 对应包名.util.UrRobotUtil;
import org.springblade.isdp.starter.document.annotation.IsdpComponent;
import
org.springblade.isdp.starter.document.operation.OperationHandlerInterface;
import org.springblade.isdp.starter.document.operation.OperationRequest;
import org.springblade.isdp.starter.document.operation.OperationResponse;
import org.springblade.isdp.starter.utils.JsonUtils;
import org.springframework.core.env.Environment;
import java.util.List;
```

```
import java.util.Objects;
/**
 * MoveOperation 操作处理类
 * <p>
 * 组件将通过 {@linkplain OperationConfiguration#globalOperationRoute()} 将本类注册绑定到对应主题上；
 * <h3>使用方法</h3>
 * 默认只需要补充 {@link OperationHandlerInterface#handler(Object)} 方法即可；<br/>
 * 方法将接收String类型数据，为MQTT消息内data字段Json化数据，
 * 如果需要使用其他数据，可自定义实现 {@link OperationHandlerInterface#convert(OperationRequest)}；<br/>
 * 方法将返回 <b>{@code OperationHandlerInterface<T,R>}</b> 中 R 的类型，组件内部将返回数据Json序列化后发送MQTT；<br/>
 * <p>
 * 拓展点：<br>
 * 1、{@link OperationHandlerInterface#convert(OperationRequest)} convert()方法可将MQTT数据进行直接处理
 * 2、{@link OperationHandlerInterface#listenerHandler(String, OperationRequest, Environment)}
 * 方法可忽略组件内部处理逻辑而使用自定义逻辑，包括不限于MQTT消息返回
 */
@IsdpComponent
@RequiredArgsConstructor
public class MoveOperation implements OperationHandlerInterface<MoveParam, OperationResponse<MoveResponse>> {
    private static final Logger log = LoggerFactory.getLogger(MoveOperation.class);
    private final NettyClient client;
    private final ArmConfig.Deviation deviation;
    @Override
    public MoveParam convert(final OperationRequest<?> request) {
        return JsonUtils.parse(JsonUtils.toJson(request.getData()), MoveParam.class);
    }
    @Override
    public OperationResponse<MoveResponse> handler(final MoveParam param) throws Exception {
        // 1. 判断坐标
        final List<Double> position = param.getPosition();
        UrRobotUtil.judgePosition(position);
        // 2. 拼接命令
        final String moveCommand = UrRobotUtil.moveCommand(position);
        client.send(moveCommand);
        log.debug("----------moveCommand----------:{}", moveCommand);
        // 3.等待执行结果
        UrRobotUtil.waitToTarget(position, deviation);
        return OperationResponse.of(new MoveResponse());
    }
}
```

编写SuckOperation.java文件逻辑

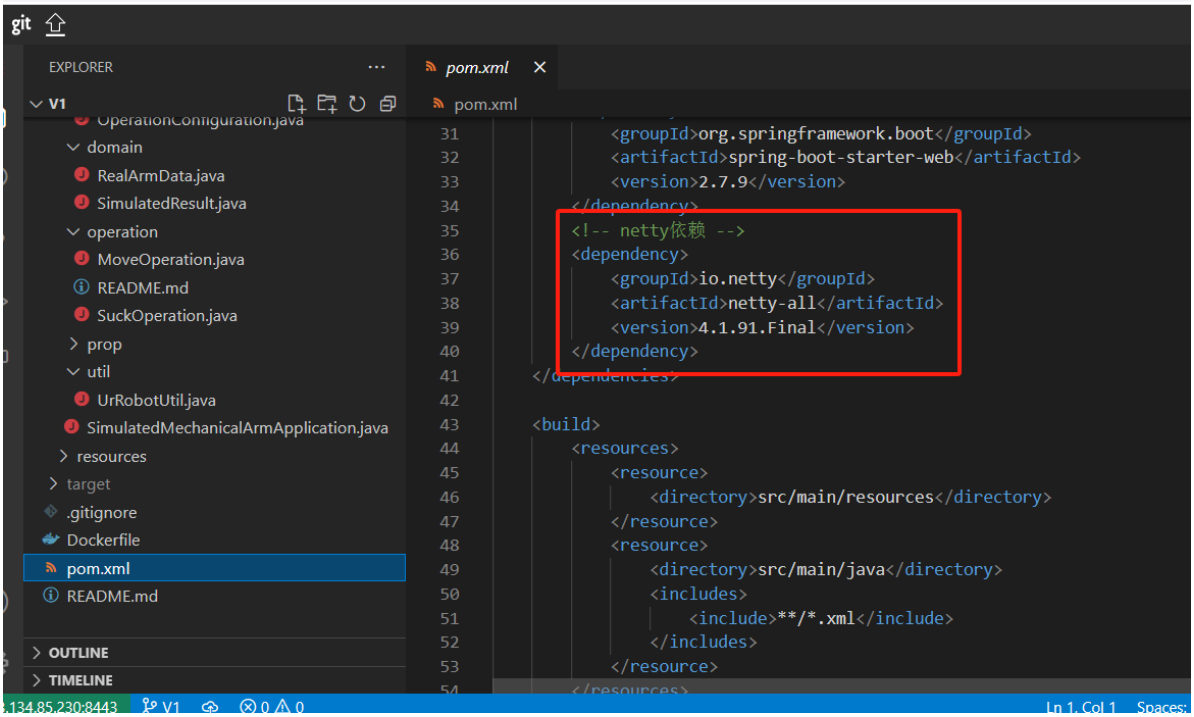
```
package 对应包名.operation;
import lombok.RequiredArgsConstructor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import 对应包名.client.ArmResultHandler;
```

```java
import 对应包名.client.NettyClient;
import 对应包名.config.OperationConfiguration;
import 对应包名.domain.RealArmData;
import 对应包名.prop.SuckParam;
import 对应包名.prop.SuckResponse;
import org.springblade.isdp.starter.document.annotation.IsdpComponent;
import
org.springblade.isdp.starter.document.operation.OperationHandlerInterface;
import org.springblade.isdp.starter.document.operation.OperationRequest;
import org.springblade.isdp.starter.document.operation.OperationResponse;
import org.springblade.isdp.starter.utils.JsonUtils;
import org.springframework.core.env.Environment;
/**
 * SuckOperation 操作处理类
 * <p>
 * 组件将通过 {@linkplain OperationConfiguration#globalOperationRoute()} 将本类注册
绑定到对应主题上；
 * <h3>使用方法</h3>
 * 默认只需要补充 {@link OperationHandlerInterface#handler(Object)} 方法即可；<br/>
 * 方法将接收String类型数据，为MQTT消息内data字段Json化数据，
 * 如果需要使用其他数据，可自定义实现 {@link
OperationHandlerInterface#convert(OperationRequest)}；<br/>
 * 方法将返回 <b>{@code OperationHandlerInterface<T,R>}</b> 中 R 的类型，组件内部将返
回数据Json序列化后发送MQTT；<br/>
 * <p>
 * 拓展点：<br>
 * 1、{@link OperationHandlerInterface#convert(OperationRequest)} convert()方法可
将MQTT数据进行直接处理
 * 2、{@link OperationHandlerInterface#listenerHandler(String, OperationRequest,
Environment)}
 * 方法可忽略组件内部处理逻辑而使用自定义逻辑，包括不限于MQTT消息返回
 */
@IsdpComponent
@RequiredArgsConstructor
public class SuckOperation implements OperationHandlerInterface<SuckParam,
OperationResponse<SuckResponse>> {
    private static final Logger log =
LoggerFactory.getLogger(SuckOperation.class);
    private final NettyClient client;
    @Override
    public SuckParam convert(final OperationRequest<?> request) {
        return JsonUtils.parse(JsonUtils.toJson(request.getData()),
SuckParam.class);
    }
    @Override
    public OperationResponse<SuckResponse> handler(final SuckParam param) throws
Exception {
        //1. 获取当前夹爪状态
        RealArmData armData = ArmResultHandler.getRealArmData();
        final Boolean suckDone = armData.getIsSuck();
        final String suckCommand = String.format("{\"type\":2,\"data\":
{\"Suck\":%b}}\n", !suckDone);
        log.debug("-------------suckCommand-------------:{}", suckCommand);
        client.send(suckCommand);
        //2. 获取状态
        int count = 0;
        while (true) {
            if (count >= 10) {
```

```
                throw new Exception("suck error,please check");
            }
            count++;
            armData = ArmResultHandler.getRealArmData();
            if (armData.getIsSuck() == !suckDone) {
                break;
            }
            Thread.sleep(500);
        }
        return OperationResponse.of(new SuckResponse());
    }
}
```

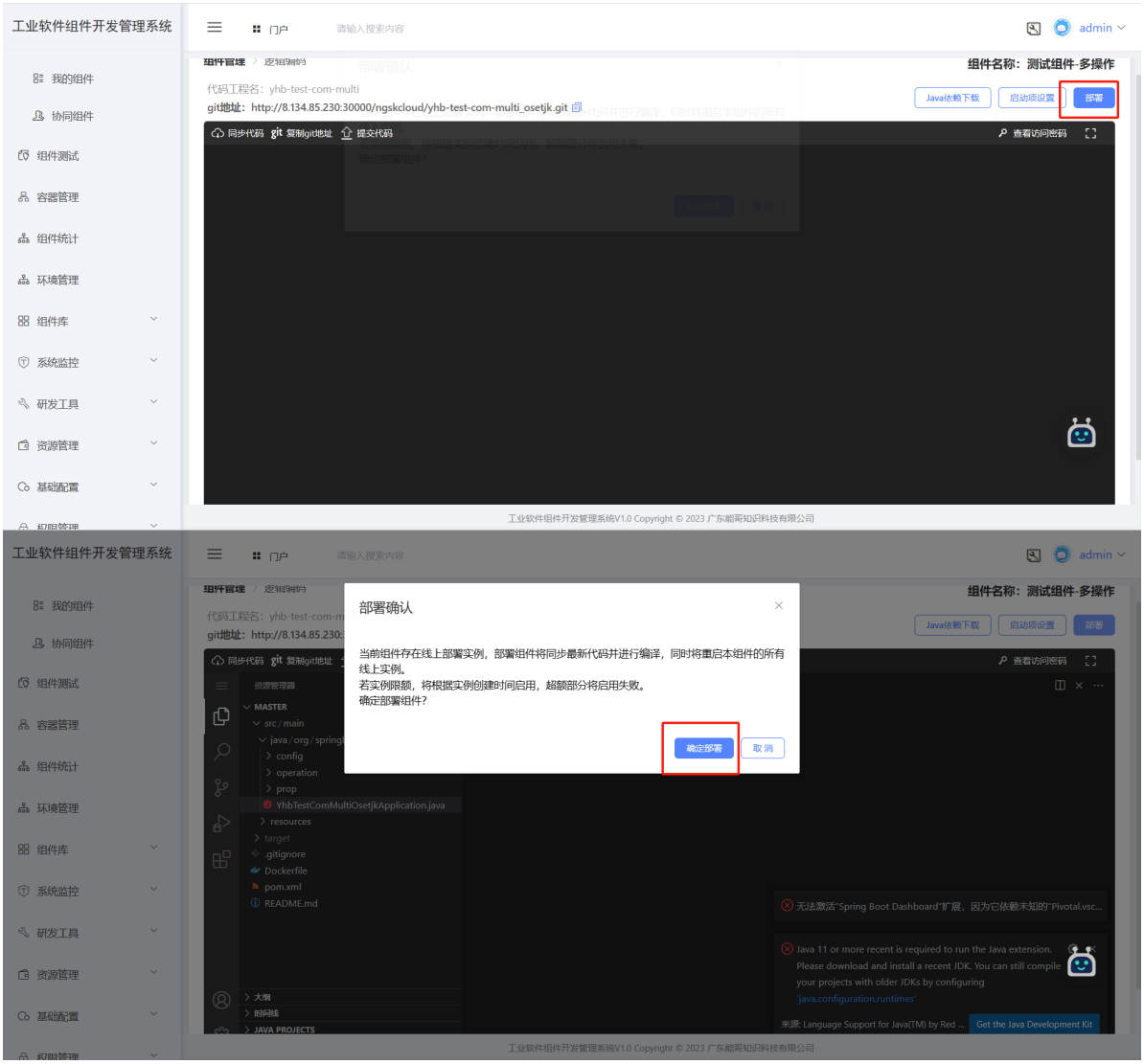
在pom.xml文件添加依赖

```
<!-- netty依赖 -->
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-all</artifactId>
  <version>4.1.91.Final</version>
</dependency>
```



3.3 组件部署

每个组件在编码结束那里进行部署



点击部署管理，搜索创建的组件名，发布组件，并点击新增实例

选择部署环境：生产

主机：8.134.85.230

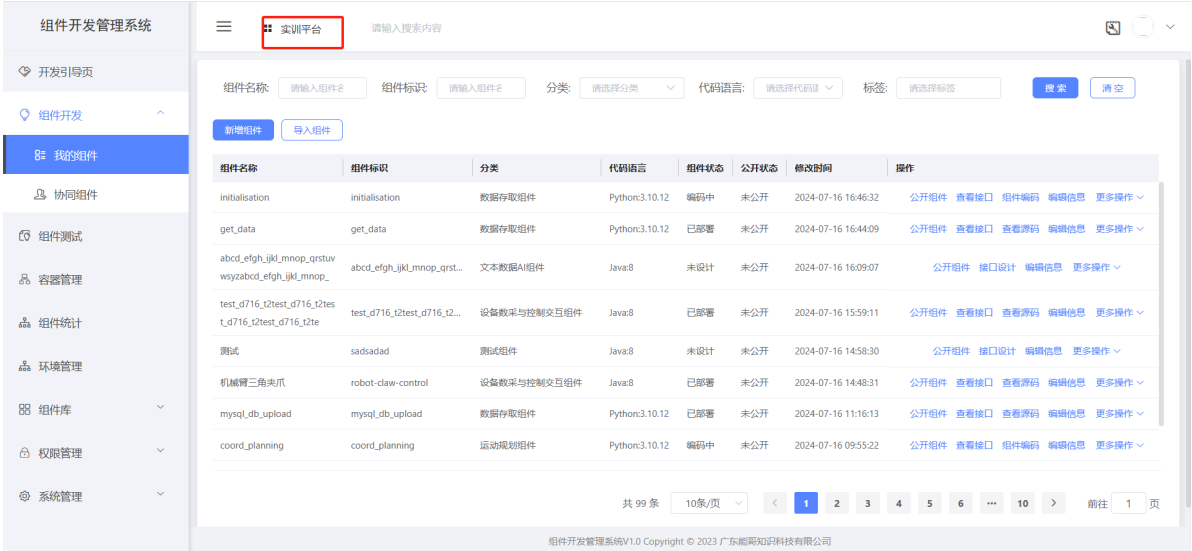
最大内存：2GB

网卡：bridge

新增完毕后展开实例列表点击启用

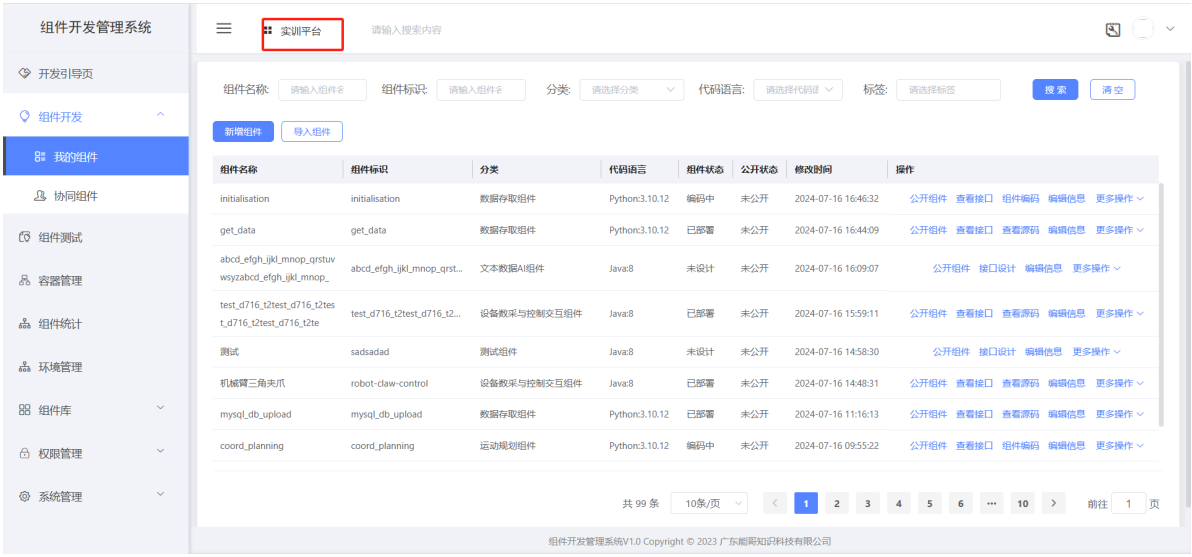
3.4 编排流程

返回门户进入流程管理与可视化编排系统

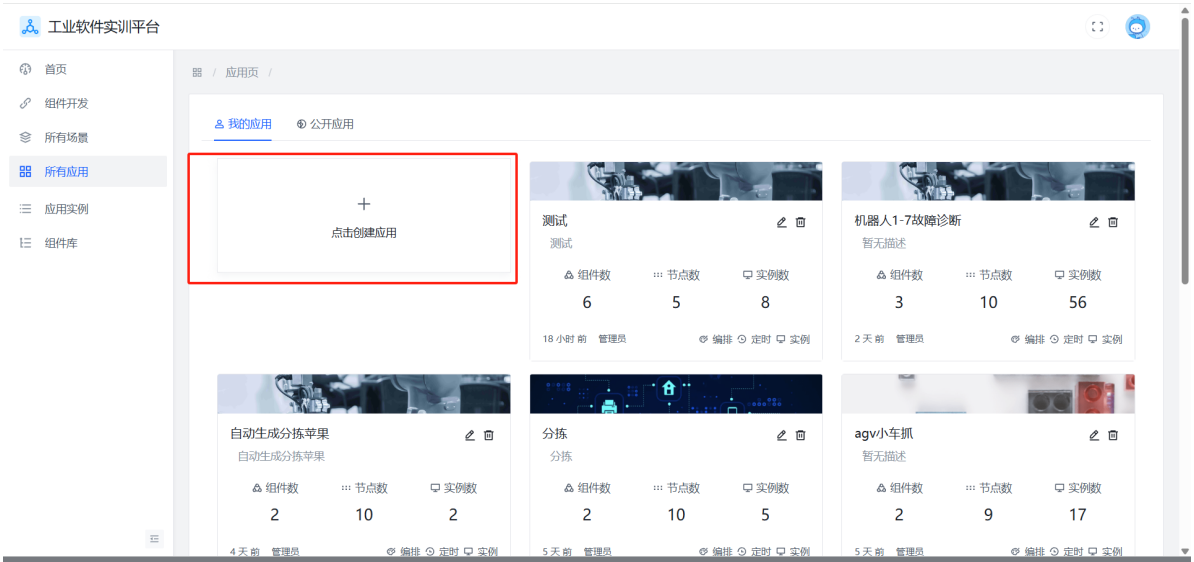


点击进入仿真应用，流程设计

切换到实训平台

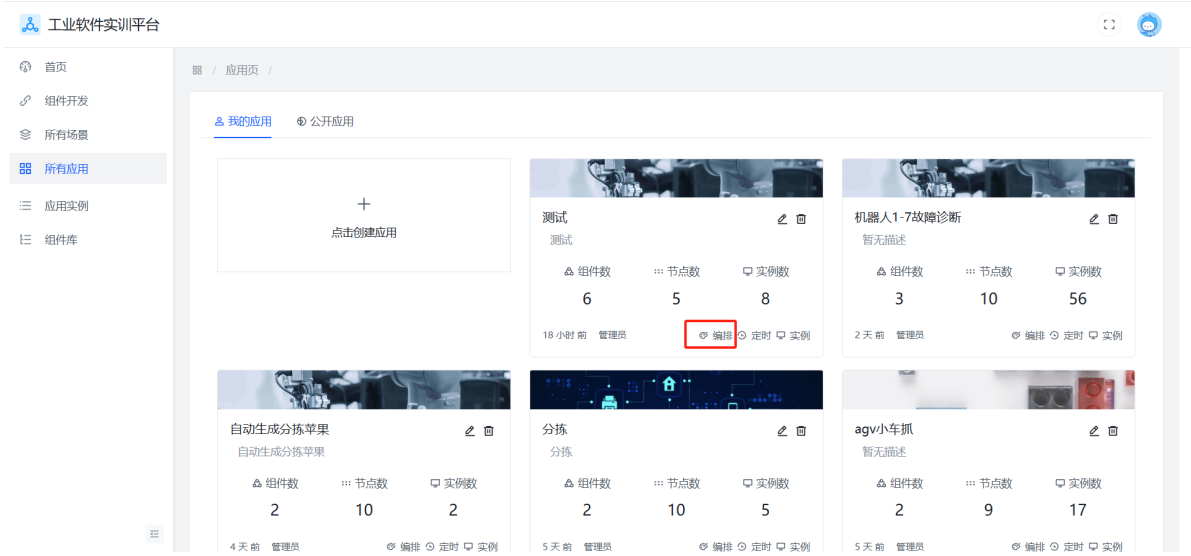

组件开发管理系统
实训平台
请输入搜索内容
开发引导页
组件开发
我的组件
协同组件
组件测试
容器管理
组件统计
环境管理
组件库
权限管理
系统管理
组件名称:
组件标识:
分类:
代码语言:
标签:
搜索
清空
新增组件
导入组件
组件名称
组件标识
分类
代码语言
组件状态
公开状态
修改时间
操作
initialisation
get_data
测试
sadsadad
机械臂三角夹爪
robot-claw-control
mysql_db_upload
coord_planning
共 99 条
10条/页
前往
页
组件开发管理系统V1.0 Copyright © 2023 广东能哥知识科技有限公司

新建应用


工业软件实训平台
首页
组件开发
所有场景
所有应用
应用实例
组件库
应用页
我的应用
公开应用
点击创建应用
测试
测试
组件数
节点数
实例数
6
5
8
机器人1-7故障诊断
暂无描述
3
10
56
自动生成分拣苹果
自动生成分拣苹果
2
10
2
分拣
分拣
2
10
5
agv小车抓
暂无描述
2
9
17


工业软件实训平台
创建应用
场景
测试场景
封面图
IOT
名称
请输入应用名称
描述
请输入应用描述
0/500
是否公开应用
取消
确定

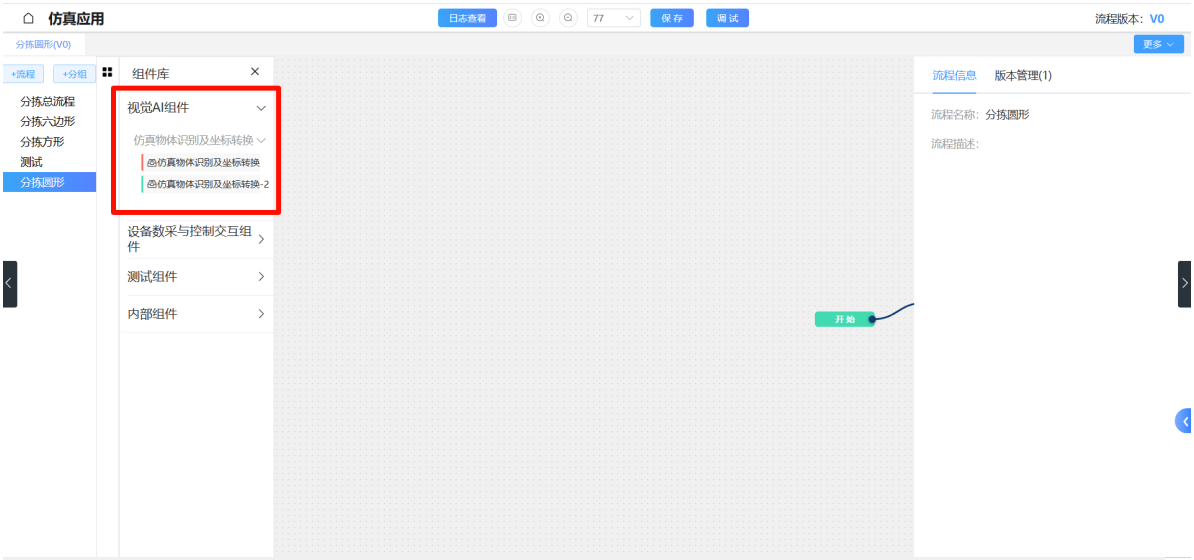
进入流程编排



该流程的主要思路为：相机进行拍照=>进行物体识别及坐标转换=>机械臂移动到物块上方=>机械臂吸取物块=>机械臂移动到物块上方=>机械臂移动至放置位置=>机械臂释放物块

1. 相机进行拍照

找到创建的仿真深度相机拖拽出来，连接photograph

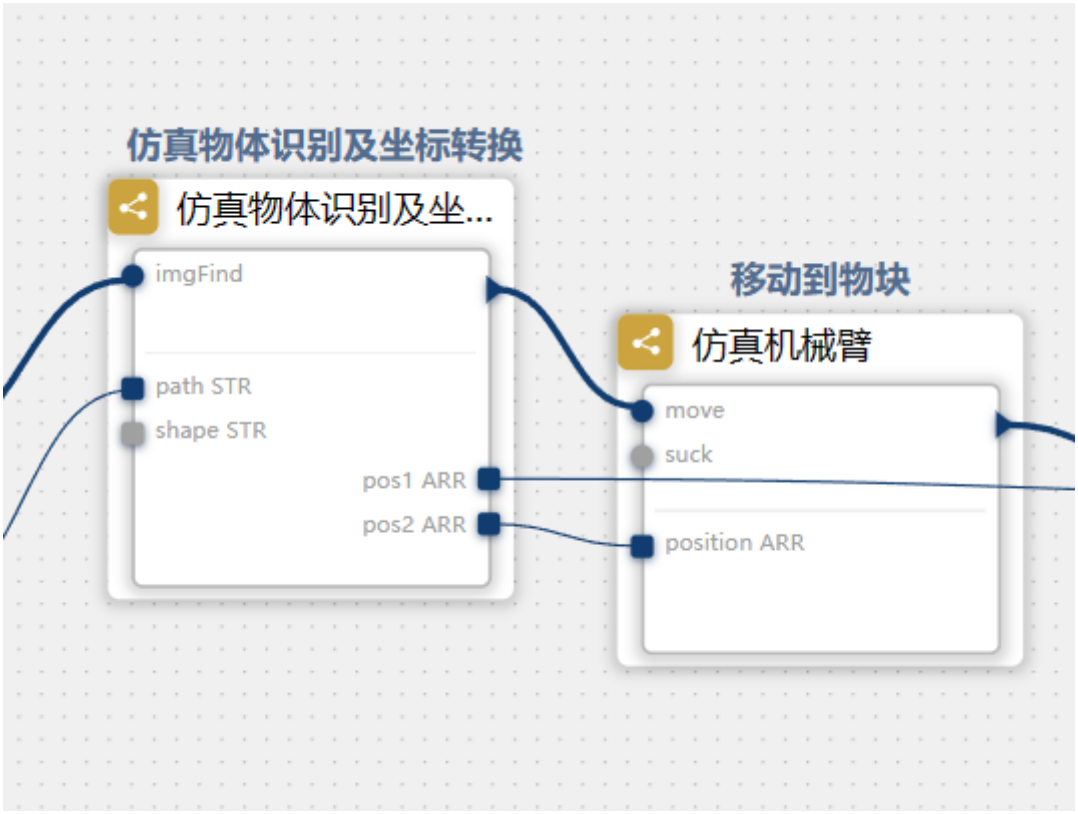


1. 进行物体识别及坐标转换

在视觉AI组件最下面找到仿真物体识别及坐标转换
接口：连接imgFind
参数：minioPath连线path

1. 机械臂移动到物块上方

在设备数采与控制交互组件创建的仿真机械臂
接口：move
参数：pos2连线position

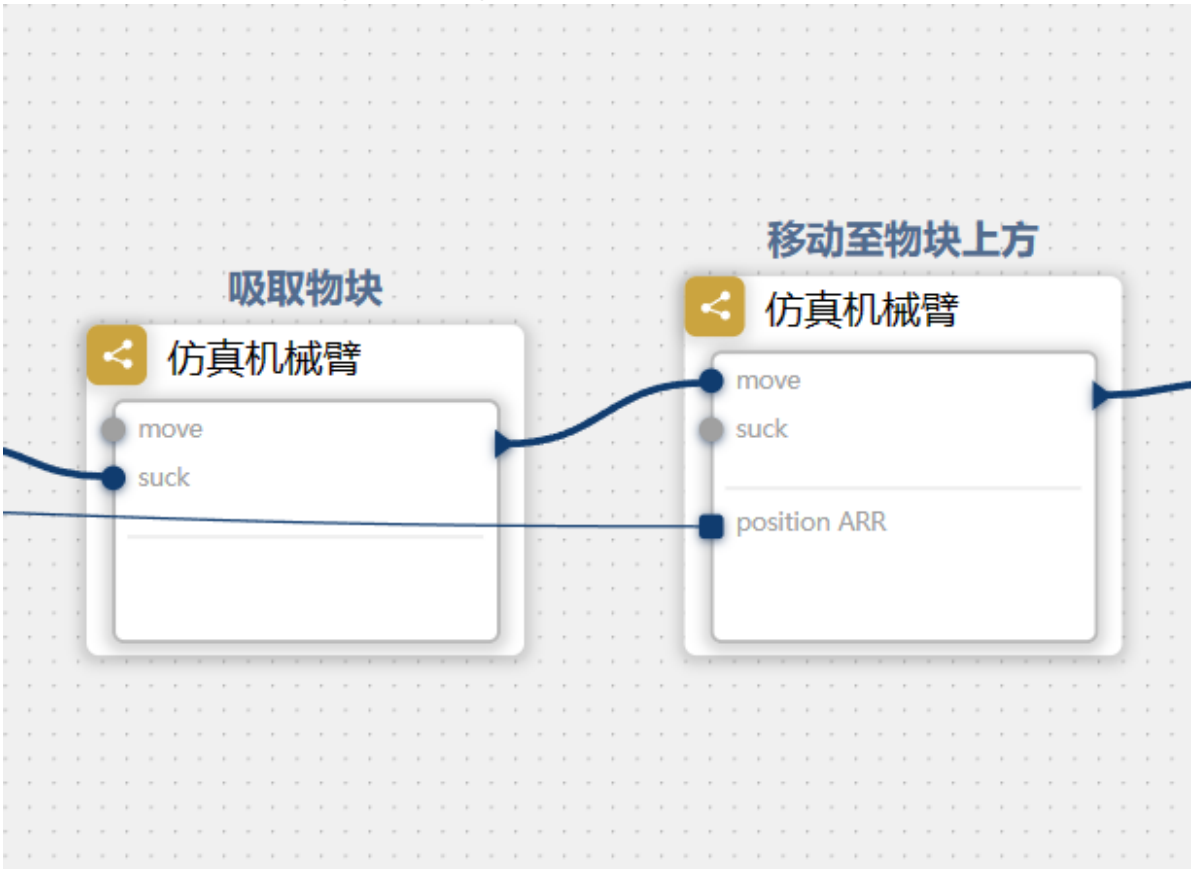


1. 机械臂吸取物块

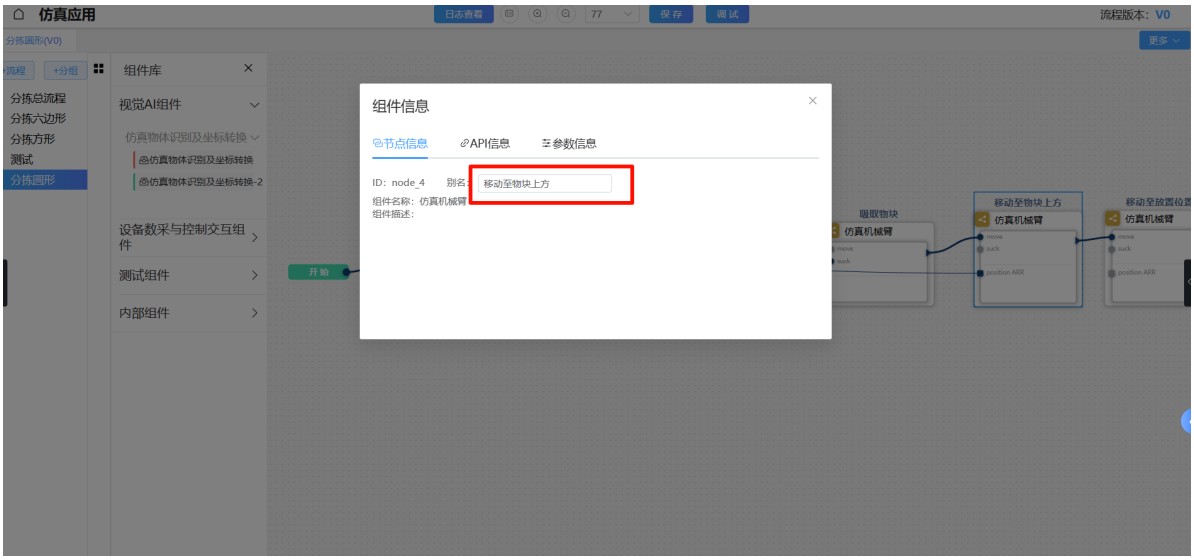
再拉出一个仿真机械臂组件
接口：suck

1. 机械臂移动回物块上方

再拉出一个仿真机械臂组件
接口：move
参数：物体识别及坐标转换的pos1连接到position



为了方便区分相同组件不同操作我们可以给，双击组件给组件命名



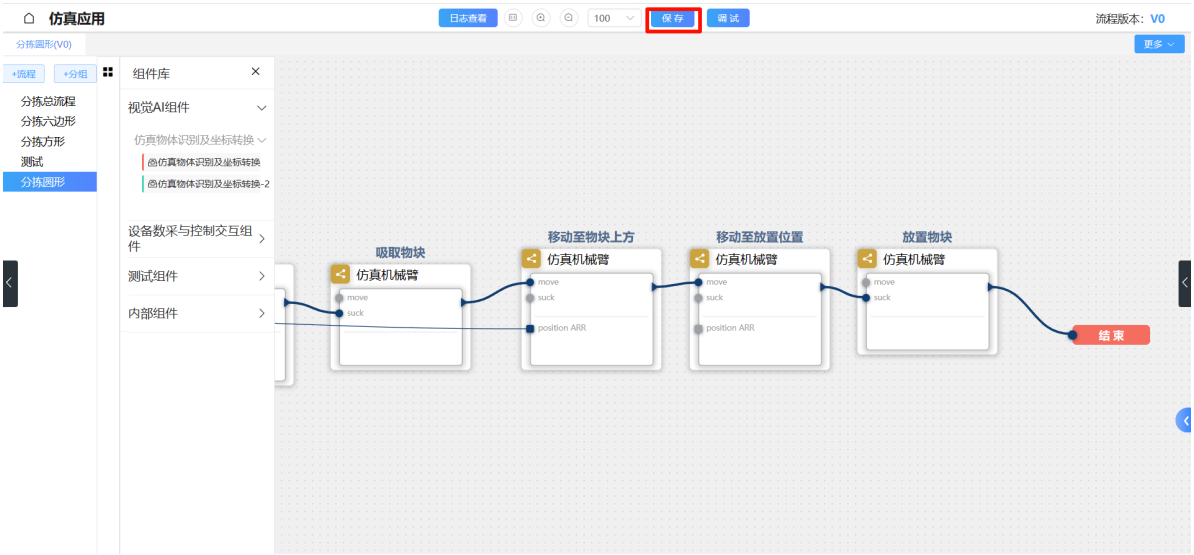
1. 机械臂移动至放置位置

再拉出一个仿真机械臂组件
接口：move

1. 机械臂释放物块

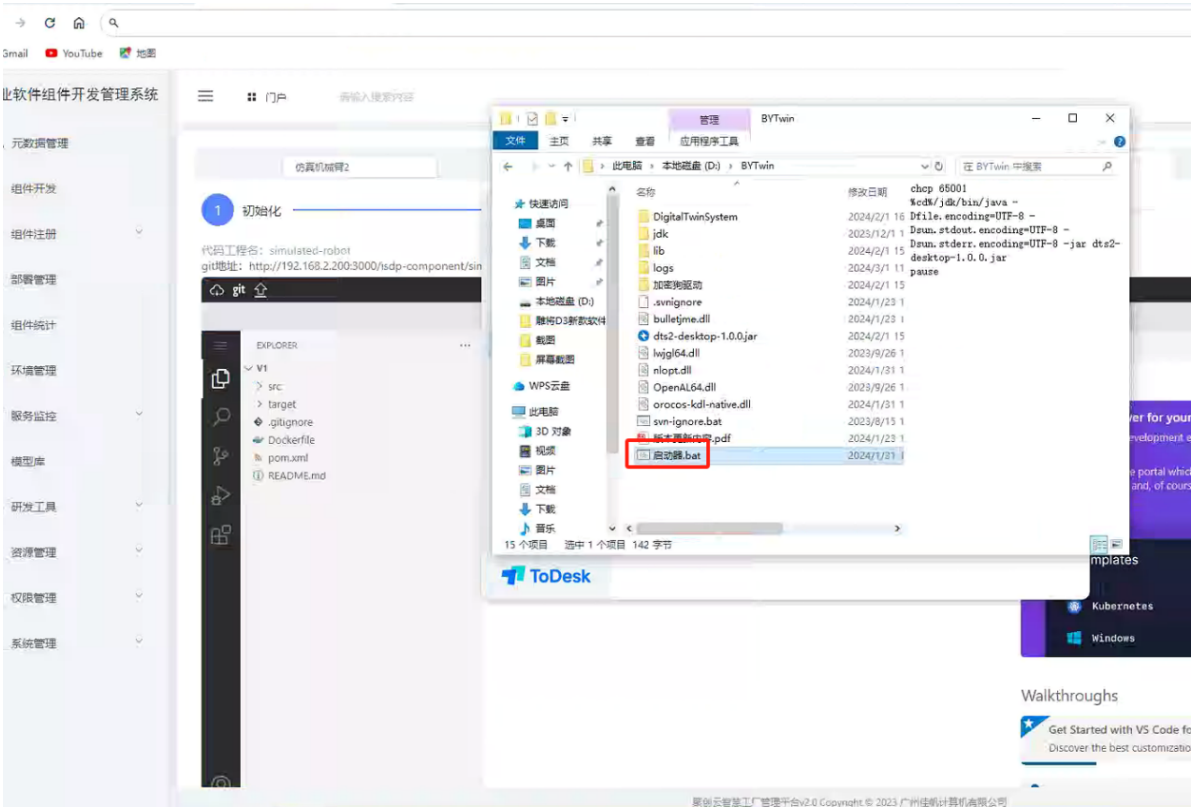
再拉出一个仿真机械臂组件
接口：suck，连接结束方块

点击编辑，点击保存，流程就创建完成了

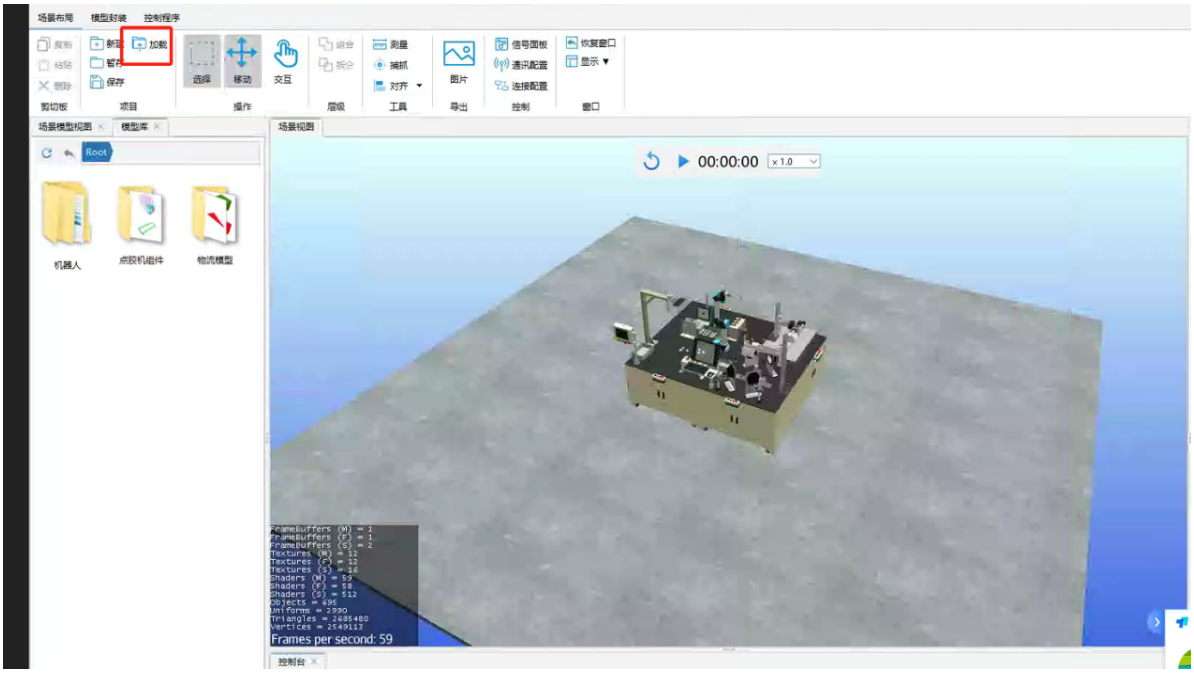


3.5 仿真加载

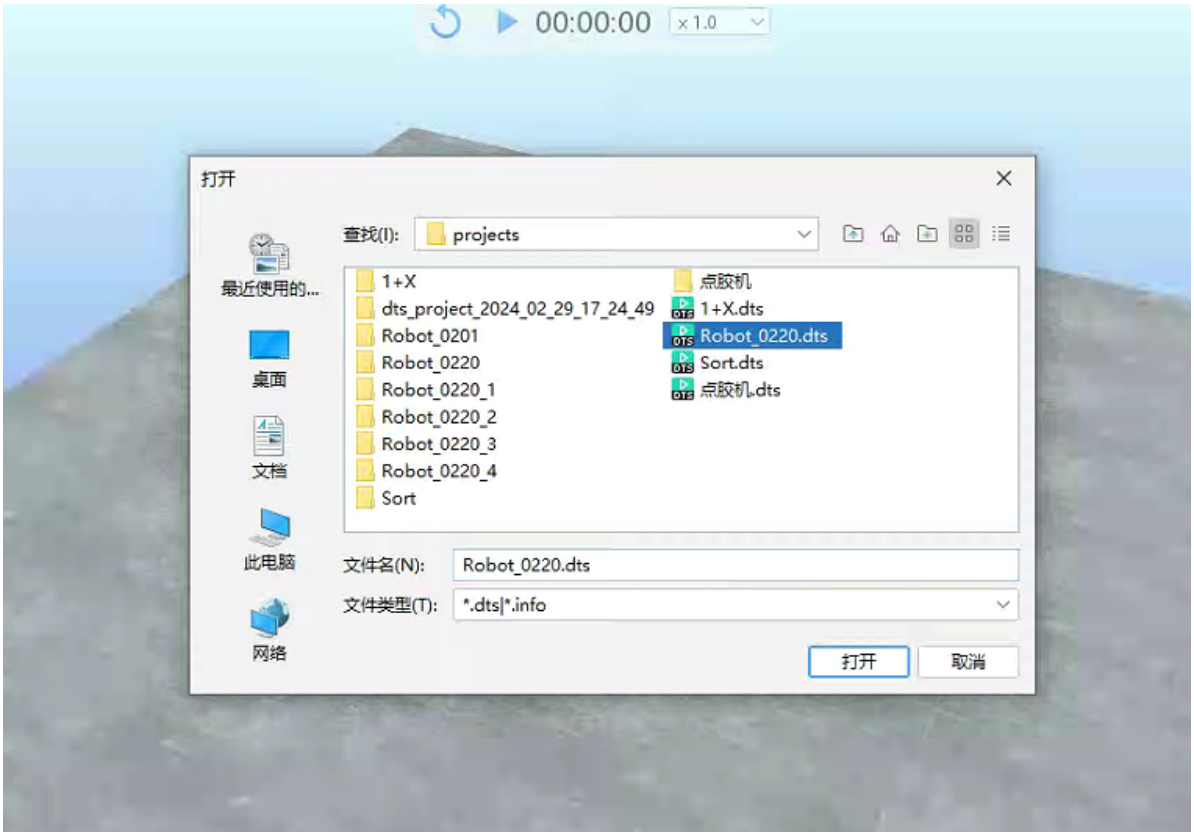
使用前面的远程桌面进入仿真软件系统
如果软件未运行双击D:\BYTwin文件夹下的



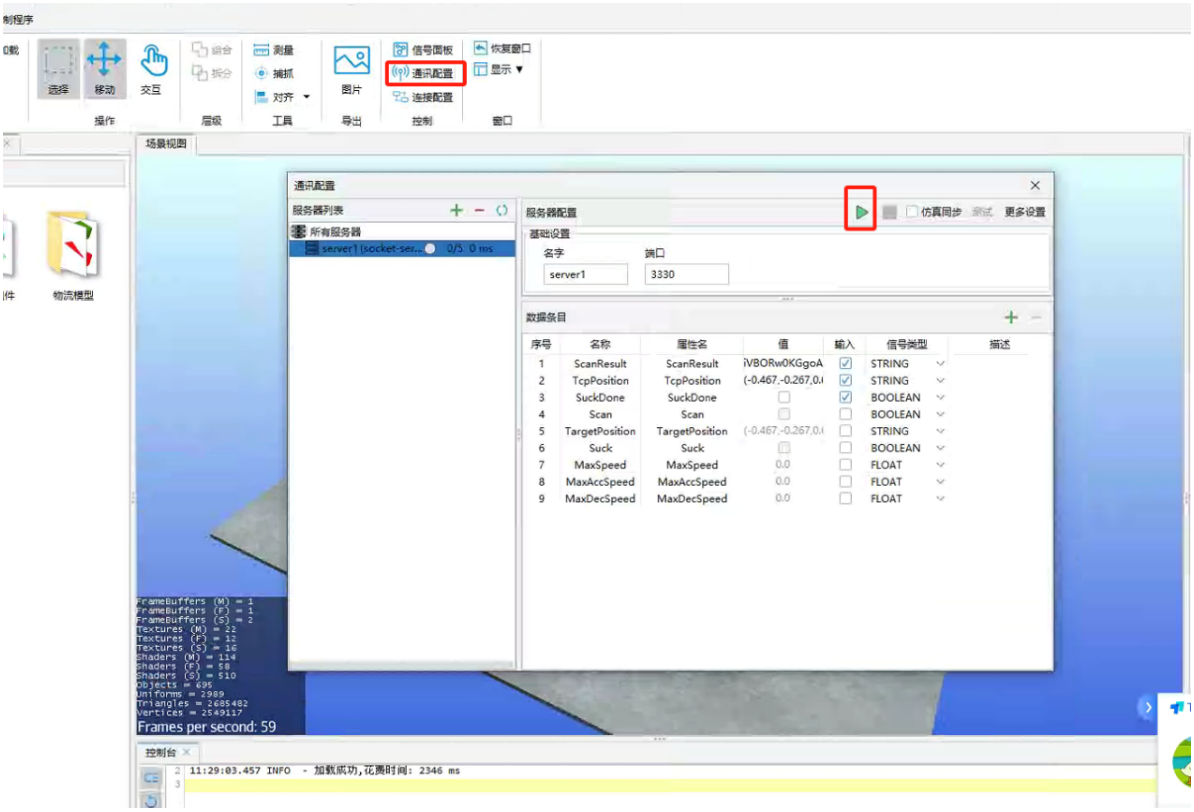
打开软件，点击左上角加载



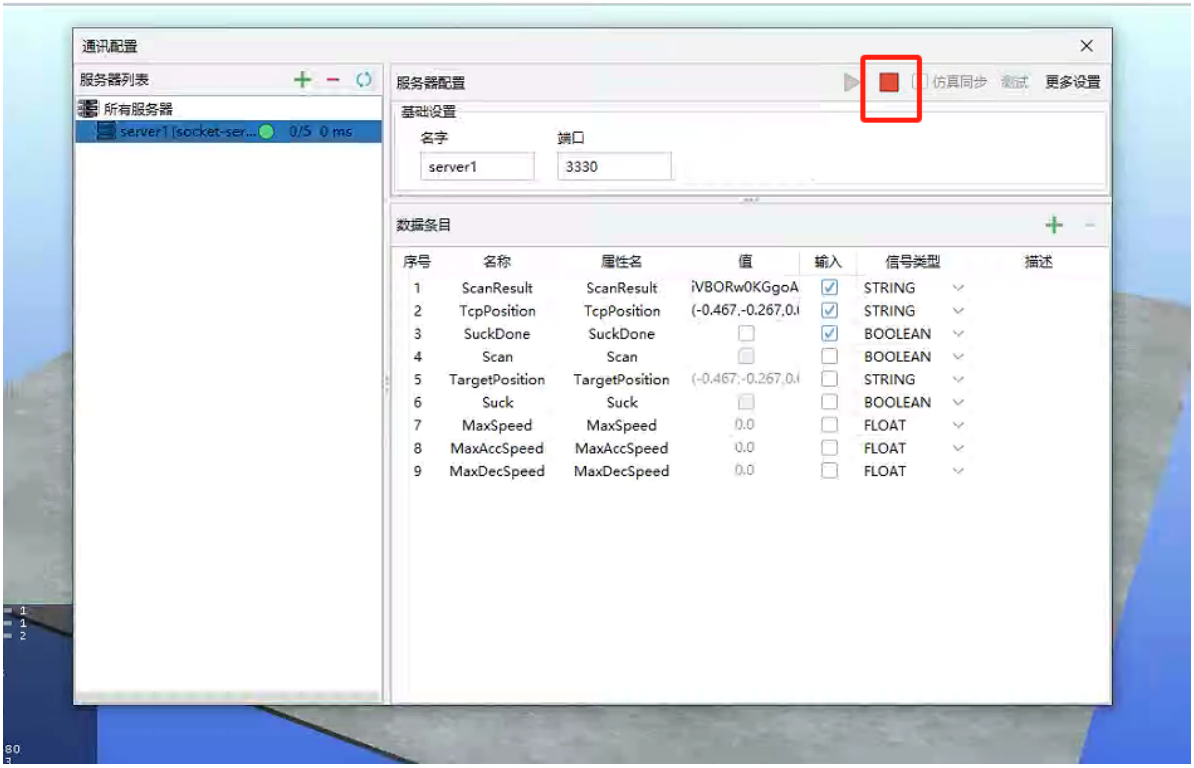
现在目标模型文件并打开



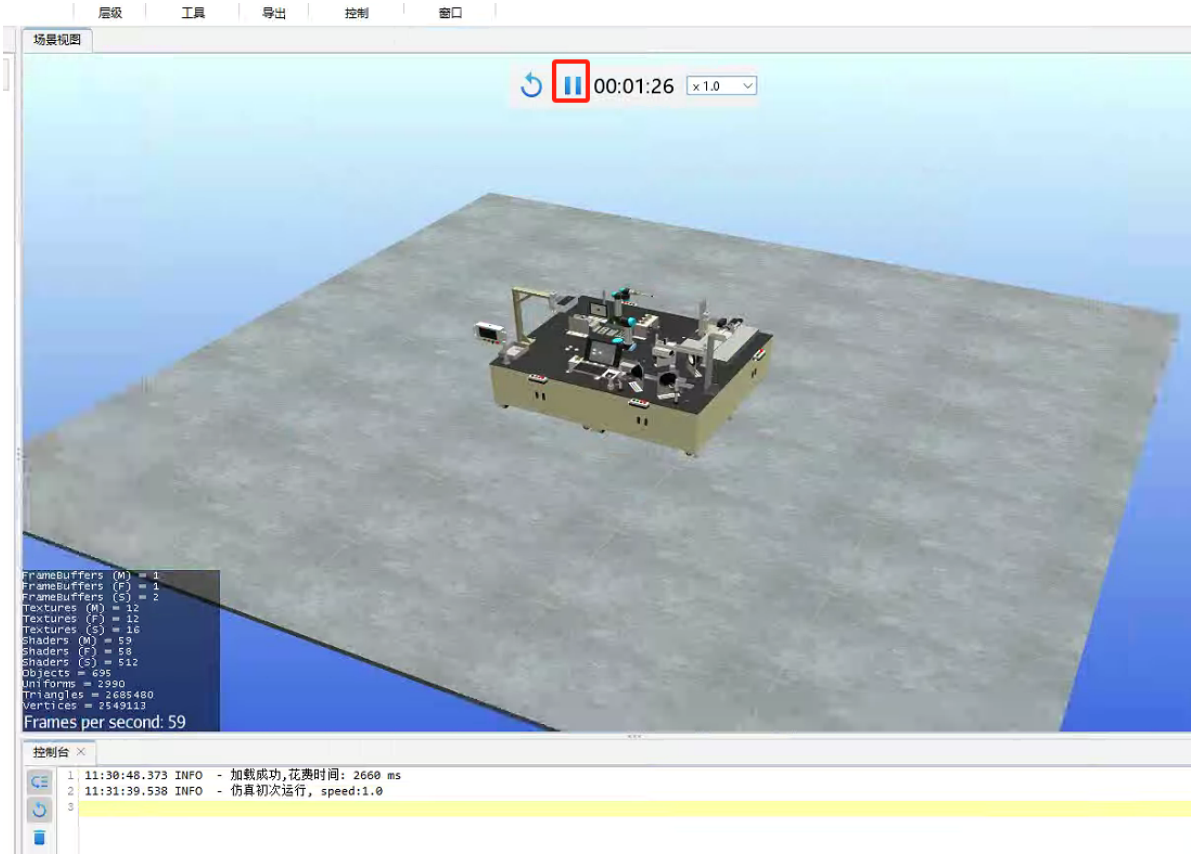
点击通信配置并启动



运行成功后返回



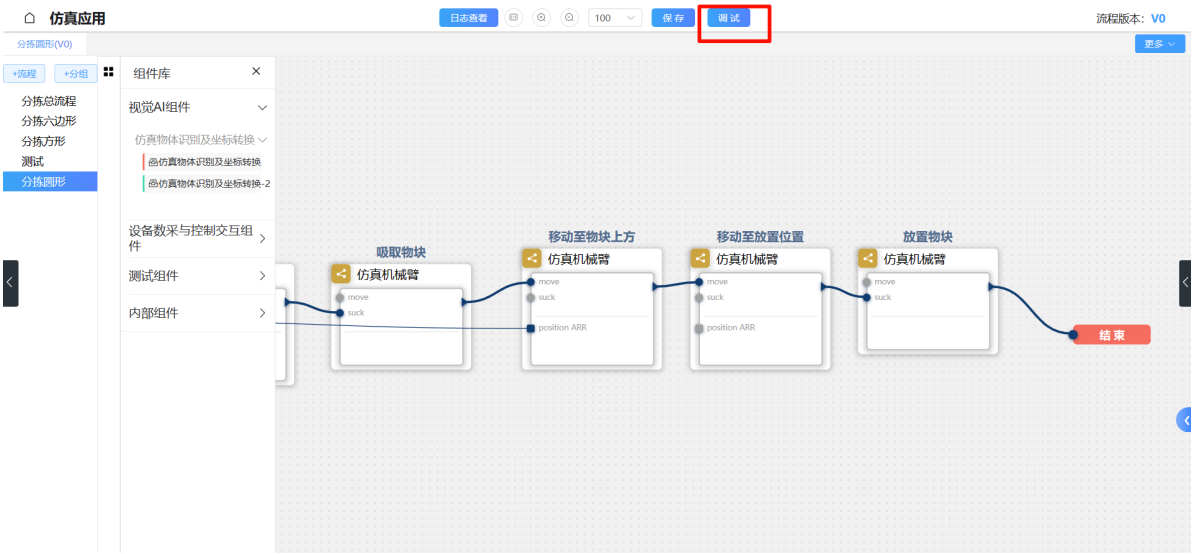
点击运行，可以看到下面运行的信息



如果组件正常，在编排运行后即可看到仿真软件里的运行效果。
保持软件运行，我们回到流程调试

3.6 流程调试

点击流程调试，进入调试页面，点击运行即可运行



点击运行，回到远程桌面既可看到机械臂的运行。