**package** signal;*//signal包，signal 是一个包（package）的名称。在Java中，包是用来组织和管理类的一种机制。它可以包含多个类文件。该包中包含用于处理和传递信号的类，比如定义信号、信号处理程序、发送信号等。*  
  
**import** com.po.dts.api.script.BaseSignalScript;*//导入BaseSignalScript类*  
**import** com.po.dts.api.model.VirtualModel;*//导入VirtualModel类*  
**import** com.po.dts.framework.feature.ModelDriverFeature;*//导入ModelDriverFeature类*  
**import** com.po.dts.api.reference.VirtualModelReference;*//导入VirtualModelReference类*  
**import** com.po.dts.api.service.ValueReferenceService;  
**import** javax.inject.Inject;  
**import** com.po.dts.framework.feature.CameraFeature;  
**import** java.util.List;  
**import** com.po.dts.api.model.ActuatorModel;  
**import** com.po.dts.api.structure.BoxVisual;  
**import** com.po.dts.api.reference.GeometryVisualReference;  
**import** com.jme3.math.Vector3f;  
**import** com.po.dts.core.manager.SignalGraphManager;  
**import** com.po.dts.core.bean.SignalConnection;  
**import** com.po.dts.core.bean.SignalGraphNodeInfo;  
**import** java.util.ArrayList;  
**import** com.po.dts.framework.service.ModelService;  
**import** com.po.dts.core.helper.RenderThreadHelper;  
**import** com.po.dts.framework.feature.RayFixtureFeature;  
  
**public** **class** SignalScript **extends** BaseSignalScript {*// 定义SignalScript类，继承自BaseSignalScript类*  
  
 *// 定义一些常量，用于表示信号名和驱动器名*  
 **public** **static** String X\_SPEED\_PROPERTY="XSpeed";  
 **public** **static** String Y\_SPEED\_PROPERTY="YSpeed";  
 **public** **static** String Z\_SPEED\_PROPERTY="ZSpeed";  
 **public** **static** String R\_SPEED\_PROPERTY="RSpeed";  
   
 **public** **static** String X\_DRIVE\_NAME="X轴驱动器";  
 **public** **static** String Y\_DRIVE\_NAME="Y轴驱动器";  
 **public** **static** String Z\_DRIVE\_NAME="Z轴驱动器";  
 **public** **static** String R\_DRIVE\_NAME="R轴驱动器";  
  
 *// 定义一些信号的名称常量*  
 **public** **static** String X\_ABSOLUTE\_SIGNAL\_NAME="X\_AbsolutePosition";  
 **public** **static** String X\_RELATIVE\_SIGNAL\_NAME="X\_RelativePosition";  
 **public** **static** String X\_ORIGIN\_SIGNAL\_NAME="X\_MoveOrigin";  
 **public** **static** String X\_STOP\_SIGNAL\_NAME="X\_StopMove";  
 **public** **static** String X\_SPEEDE="X\_MoveSpeed";  
 **public** **static** String X\_ARRIVE\_SIGNAL\_NAME="X\_ArrivePosition";  
 **public** **static** String X\_ARRIVE\_ORIGIN\_SIGNAL\_NAME="X\_ArriveOrigin";  
 **public** **static** String X\_RUNNING\_SIGNAL\_NAME="X\_Running";  
 **public** **static** String X\_CURRENT\_POSITION="X\_CurrentPosition";  
  
 **public** **static** String Y\_ABSOLUTE\_SIGNAL\_NAME="Y\_AbsolutePosition";  
 **public** **static** String Y\_RELATIVE\_SIGNAL\_NAME="Y\_RelativePosition";  
 **public** **static** String Y\_ORIGIN\_SIGNAL\_NAME="Y\_MoveOrigin";  
 **public** **static** String Y\_STOP\_SIGNAL\_NAME="Y\_StopMove";  
 **public** **static** String Y\_SPEEDE="Y\_MoveSpeed";  
 **public** **static** String Y\_ARRIVE\_SIGNAL\_NAME="Y\_ArrivePosition";  
 **public** **static** String Y\_ARRIVE\_ORIGIN\_SIGNAL\_NAME="Y\_ArriveOrigin";  
 **public** **static** String Y\_RUNNING\_SIGNAL\_NAME="Y\_Running";  
 **public** **static** String Y\_CURRENT\_POSITION="Y\_CurrentPosition";  
  
 **public** **static** String Z\_ABSOLUTE\_SIGNAL\_NAME="Z\_AbsolutePosition";  
 **public** **static** String Z\_RELATIVE\_SIGNAL\_NAME="Z\_RelativePosition";  
 **public** **static** String Z\_ORIGIN\_SIGNAL\_NAME="Z\_MoveOrigin";  
 **public** **static** String Z\_STOP\_SIGNAL\_NAME="Z\_StopMove";  
 **public** **static** String Z\_SPEEDE="Z\_MoveSpeed";  
 **public** **static** String Z\_ARRIVE\_SIGNAL\_NAME="Z\_ArrivePosition";  
 **public** **static** String Z\_ARRIVE\_ORIGIN\_SIGNAL\_NAME="Z\_ArriveOrigin";  
 **public** **static** String Z\_RUNNING\_SIGNAL\_NAME="Z\_Running";  
 **public** **static** String Z\_CURRENT\_POSITION="Z\_CurrentPosition";  
  
 **public** **static** String R\_ABSOLUTE\_SIGNAL\_NAME="R\_AbsolutePosition";  
 **public** **static** String R\_RELATIVE\_SIGNAL\_NAME="R\_RelativePosition";  
 **public** **static** String R\_ORIGIN\_SIGNAL\_NAME="R\_MoveOrigin";  
 **public** **static** String R\_STOP\_SIGNAL\_NAME="R\_StopMove";  
 **public** **static** String R\_SPEEDE="R\_MoveSpeed";  
 **public** **static** String R\_ARRIVE\_SIGNAL\_NAME="R\_ArrivePosition";  
 **public** **static** String R\_ARRIVE\_ORIGIN\_SIGNAL\_NAME="R\_ArriveOrigin";  
 **public** **static** String R\_RUNNING\_SIGNAL\_NAME="R\_Running";  
 **public** **static** String R\_CURRENT\_POSITION="R\_CurrentPosition";  
  
  
 @Inject  
 ValueReferenceService valueReferenceService;  
 @Inject  
 SignalGraphManager signalGraphManager;  
 @Inject  
 ModelService modelService;  
 @Inject  
 RenderThreadHelper renderThreadHelper;  
  
  
 @Override  
 **public** **void** onInitialize() { *// 初始化方法*  
 **super**.onInitialize();  
  
 VirtualModel model = getVirtualModel();  
*// model.setCustomerPropertyValue("InstallCamera", new VirtualModelReference(-1));*  
*// model.setCustomerPropertyValue("Suck", new VirtualModelReference(-1));*  
*// model.setCustomerPropertyValue("WorkBoard", new VirtualModelReference(-1));*  
  
 *// 设置一些信号属性为true，表示到达位置通过将这些信号属性的属性值设置为true，可能表示模型中的各个轴都已经到达了它们的目标位置或原点。*  
 model.setSignalPropertyValue(X\_ARRIVE\_SIGNAL\_NAME, true);  
 model.setSignalPropertyValue(X\_ARRIVE\_ORIGIN\_SIGNAL\_NAME, true);  
 model.setSignalPropertyValue(Y\_ARRIVE\_SIGNAL\_NAME, true);  
 model.setSignalPropertyValue(Y\_ARRIVE\_ORIGIN\_SIGNAL\_NAME, true);  
 model.setSignalPropertyValue(Z\_ARRIVE\_SIGNAL\_NAME, true);  
 model.setSignalPropertyValue(Z\_ARRIVE\_ORIGIN\_SIGNAL\_NAME, true);  
 model.setSignalPropertyValue(R\_ARRIVE\_SIGNAL\_NAME, true);  
 model.setSignalPropertyValue(R\_ARRIVE\_ORIGIN\_SIGNAL\_NAME, true);  
 model.setSignalPropertyValue("All\_ArriveOrigin", true);  
 }  
  
 @Override  
 **public** **void** onReset() {  
 **super**.onReset();*// 重置方法*  
 }  
  
 @Override  
 **public** **void** onRun() {*// 运行方法*  
 **super**.onRun();  
 }  
  
 **public** **void** onCameraPropertyUpdated(Object value){*//public void: 这是方法的访问修饰符。public表示该方法可以被其他类访问，void表示该方法不返回任何值。onCameraPropertyUpdated: 这是方法的名称，意为当相机属性更新时触发该方法。(Object value): 这是方法的参数列表。Object是Java中的基础类，表示任何类型的对象。value是参数的名称，用于接收传递给方法的值。方法体由大括号 {} 包围。*  
 **if**(value==**null**||((VirtualModelReference)value).getEntityId()==-1){*//这是一个条件语句，用于检查传递给方法的value是否为空或者其entityId是否为-1。如果是，则直接返回，不执行后续代码。*  
 **return**;  
 }  
 VirtualModelReference modelReference=(VirtualModelReference)value;*//这一行将value强制转换为VirtualModelReference类型，并赋值给变量modelReference。*  
 VirtualModel cameraModel=valueReferenceService.getReference(modelReference);*//这一行调用了valueReferenceService的getReference方法，根据modelReference获取对应的VirtualModel对象，并将其赋值给变量cameraModel。*  
  
 List<CameraFeature> list = cameraModel.getFeature().stream()*//cameraModel.getFeature()这一部分调用了cameraModel对象的getFeature()方法，返回了一个特征的列表。.stream()这一部分将特征列表转换为一个流（Stream）。*  
 .filter(feature -> feature **instanceof** CameraFeature)*//.filter(feature -> feature instanceof CameraFeature): 这一部分使用filter操作过滤流中的元素，保留那些属于CameraFeature类型的特征。feature -> feature instanceof CameraFeature是一个lambda表达式，用于判断特征是否是CameraFeature类型。*  
 .map(feature -> (CameraFeature) feature).toList();*//这行使用map操作将流中的每个特征转换为CameraFeature类型。由于之前使用了filter操作，这里不需要再次进行类型检查，因此可以直接进行强制类型转换。.toList(): 这一部分将流中的元素收集到一个列表中。//所以，整个操作的作用是从cameraModel对象的特征列表中筛选出所有的CameraFeature类型的特征，并将它们收集到一个列表中，最终赋值给List<CameraFeature> list变量。*  
 **if** (list.isEmpty()){*//这是一个条件语句，它检查list是否为空，即没有找到任何相机特征。如果list为空，则执行花括号中的代码块。*  
 warn("选中模型({})未存在相机功能，请添加相机功能！", cameraModel.getName());*//在代码块中，使用了warn()方法输出警告信息，提示用户选中的模型没有相机功能。然后使用cameraModel.getName()获取模型的名称，并将其填充到警告信息中。*  
 **return**;*//最后使用return语句结束方法的执行。*  
 }  
  
 **if**(!cameraModel.isExistSignalProperty("Scan")){*//这里接收一个展示名为”拍摄“的信号值“Scan"。这也是一个条件语句，它检查相机模型是否具有名为"Scan"的信号属性。*  
 warn("相机模型不存在Scan信号！请添加");*//如果相机模型不具有"Scan"信号属性，则输出一条警告信息，提示用户添加该信号属性。*  
 **return**;*//使用return语句结束方法的执行*  
 }  
  
 **if**(!cameraModel.isExistSignalProperty("ScanDone")){  
 warn("相机模型不存在ScanDone信号！请添加");  
 **return**;  
 }  
 VirtualModel model=getVirtualModel();*//这一行声明并初始化了一个VirtualModel类型的变量model，并调用了一个名为getVirtualModel()的方法来获取VirtualModel的实例。*  
 ActuatorModel actuatorModel=model.getRootActuator();*//这一行从model中获取根执行器（ActuatorModel）并将其赋值给变量actuatorModel。*  
 BoxVisual showNode=(BoxVisual)actuatorModel.getRootStructure().findChildByName("相机显示节点");*//这一行通过actuatorModel找到根结构（RootStructure），然后通过findChildByName("相机显示节点")方法寻找名为"相机显示节点"的子节点，并将其转换为BoxVisual类型赋值给变量showNode。*  
 **if**(showNode==**null**){*//条件语句if*  
 warn("相机展示节点不存在！");*//警告信息*  
 **return**;  
 }  
 *//接下来的代码应该是关于相机展示节点的操作，包括设置其大小等*  
 CameraFeature cameraFeature = list.get(0);*//这一行从list中获取第一个相机特征，并将其赋值给变量cameraFeature。list.get(0)表示获取列表中的第一个元素。*  
 *//适应像素*  
 **int** w=cameraFeature.getWidth();*//应该是获取相机分辨率宽度像素*  
 **int** h=cameraFeature.getHeight();*//应该是获取相机分辨率高度像素*  
 **float** f=1.0f\*w/h;*//这一行计算了宽高比（即宽度除以高度）。*  
 **float** w1;*//声明变量w1，用来存储宽度*  
 **float** h1;*//声明变量h1，用来存储高度*  
 **if**(f>1.88f){*//这是一个条件语句，根据计算得到的宽高比f的值进行判断。如果宽高比大于1.88，执行if语句块；否则执行else语句块。*  
 w1=0.22f;*//赋值操作*  
 h1=1.0f\*h/w\*w1;*//赋值操作*  
 }**else**{  
 h1=0.117f;*//赋值操作*  
 w1=1.0f\*w/h\*h1;*//赋值操作*  
 }  
 showNode.setSize(**new** Vector3f(w1,h1,0.002f));*//使用了调整后的宽度和高度值和0.002f。*  
  
 cameraFeature.setDisplayGeometry(**new** GeometryVisualReference(showNode.getEntityId()));*//这一行将相机特征的显示几何设置为相机显示节点的几何引用。通过这个设置，可以将相机的图像显示在相机显示节点上。*  
  
 connect(model, "InstallCameraScan", "InstallScanDone", cameraModel, "ScanDone", "Scan");*//这一行调用了一个名为connect的方法，model: 这是一个VirtualModel类型的对象，表示一个虚拟模型。InstallCameraScan": 这是一个字符串参数，表示model对象的一个信号属性名称。InstallScanDone": 这是一个字符串参数。cameraModel: 这是另一个VirtualModel类型的对象，表示另一个虚拟模型，可能是相机模型。"ScanDone": 这是一个字符串参数，表示cameraModel对象的一个信号属性名称，可能是拍摄结束信号。"Scan": 这是一个字符串参数，表示cameraModel对象的另一个信号属性名称，应该是拍摄信号。具体作用可能是将model的"InstallCameraScan"信号与cameraModel的"ScanDone"信号和"Scan"信号相连接。*  
 }  
  
 **public** **void** onSuckPropertyUpdated(Object value){  
 **if**(value==**null**||((VirtualModelReference)value).getEntityId()==-1){  
 **return**;  
 }  
 VirtualModelReference modelReference=(VirtualModelReference)value;  
 VirtualModel suckModel=valueReferenceService.getReference(modelReference);  
  
 List<RayFixtureFeature> list = suckModel.getFeature().stream()  
 .filter(feature -> feature **instanceof** RayFixtureFeature)  
 .map(feature -> (RayFixtureFeature) feature).toList();  
 **if** (list.isEmpty()){  
 warn("选中模型({})未存在相机功能，请添加相机功能！", suckModel.getName());  
 **return**;  
 }  
  
 VirtualModel model=getVirtualModel();  
 connect(model, "InstallSuck", "InstallSuckDone", suckModel, "SuckDone", "Suck");  
 connect(model, "InstallHeightLight", "", suckModel, "", "Light");  
 }  
  
 *// 连接信号的方法*  
 **private** **void** connect(VirtualModel workModel,String workOut,String workIn,VirtualModel toolModel,String toolOut,String toolIn){  
 **int** sceneId = modelService.getCurrentSceneModel().getEntityId();*// 获取当前场景的实体ID*  
 SignalGraphNodeInfo workInfo = signalGraphManager.getNodeInfoByModelEntityId(sceneId, workModel.getEntityId());*// 获取工作模型的信号节点信息*  
  
 List<SignalConnection> connections1 = signalGraphManager.getConnectionByItemName(sceneId, workInfo.getId(), workOut);*// 获取工作模型的输出连接列表*  
 *//remove*  
 *// 删除连接*  
 **for** (SignalConnection connection : **new** ArrayList<SignalConnection>(connections1)) {  
 signalGraphManager.removeSignalConnection(sceneId,connection);  
 }  
 List<SignalConnection> connections2 = signalGraphManager.getConnectionByItemName(sceneId, workInfo.getId(), workIn);*// 获取工作模型的输入连接列表*  
 *//remove*  
 *//删除*  
 **for** (SignalConnection connection : **new** ArrayList<SignalConnection>(connections2)) {  
 signalGraphManager.removeSignalConnection(sceneId,connection);  
 }  
 *//add*  
 *//添加*  
 SignalGraphNodeInfo toolInfo = signalGraphManager.getNodeInfoByModelEntityId(sceneId, toolModel.getEntityId());  
  
 SignalConnection connection = **new** SignalConnection(workInfo.getId(), workOut, toolInfo.getId(), toolIn);  
 signalGraphManager.addSignalConnection(sceneId,connection);  
 **if**("".equals(workOut)){  
 **return**;  
 }  
 SignalConnection connection2 = **new** SignalConnection(toolInfo.getId(), toolOut, workInfo.getId(), workIn);  
 signalGraphManager.addSignalConnection(sceneId,connection2);  
 }  
  
  
   
 *// 移动操作的方法*  
 **private** **void** move(String actName,String arrivePosition,String running,String speedName,String arriveOrigin,String curPosition,**float** target){  
 VirtualModel model = getVirtualModel();*// 获取虚拟模型实例*  
 **if**((**boolean**)model.getSignalPropertyValue(arrivePosition)){*// 如果到达位置信号为真*  
   
 model.setSignalPropertyValue(arrivePosition, false);*// 设置到达位置信号为假*  
 model.setSignalPropertyValue(running, true);*// 设置运行信号为真*  
 model.setSignalPropertyValue(arriveOrigin, false);*// 设置到达原点信号为假*  
 model.setSignalPropertyValue("All\_ArriveOrigin", false);*// 设置所有到达原点信号为假*  
   
 **float** speed=(**float**) model.getCustomerPropertyValue(speedName);*// 获取速度*  
  
 ModelDriverFeature feature=(ModelDriverFeature)model.getFeature(actName);*// 获取模型驱动器功能*  
 waitAsyncTask(feature.driveInSpeed(target, speed));*// 异步等待移动到目标位置*  
   
 **float** value=feature.getActuatorReference().getReference().getMotionDriver().getValue();*// 获取移动驱动器的当前值*  
 **boolean** isArriveOrgin=value==0.0f; *// 检查是否到达原点*  
 info("{} value:{},speed:{}",actName,value,speed);*// 输出信息*  
  
 *// 设置到达原点和所有到达原点信号*  
 model.setSignalPropertyValue(arriveOrigin, isArriveOrgin);  
 model.setSignalPropertyValue("All\_ArriveOrigin", checkAllArriveOrigin(model));  
 model.setSignalPropertyValue(arrivePosition, true);*// 设置到达位置信号为真*  
 model.setSignalPropertyValue(running, false);*// 设置运行信号为假*  
 model.setSignalPropertyValue(curPosition, value); *// 设置当前位置信号*  
 }  
 }  
  
 **private** **boolean** checkAllArriveOrigin(VirtualModel model){*//private boolean: 这是方法的访问修饰符。private表示该方法只能在同一类中访问，boolean表示该方法返回一个布尔值。(VirtualModel model): 这是方法的参数列表。VirtualModel是一个类类型，model是参数的名称，用于接收传递给方法的VirtualModel对象。*  
 **boolean** b=true;*//这一行声明并初始化了一个布尔变量b，初始值为true。*  
 String[] acts={"X","Y","Z","R"};*//这一行声明并初始化了一个字符串数组acts，包含了四个元素，分别是"X"、"Y"、"Z"和"R"。*  
 **for** (**int** i=0; i<acts.length; i++) {*//这是一个for循环，用于遍历acts数组中的元素。循环变量i从0开始，逐步增加，直到达到acts数组的长度。*  
 ModelDriverFeature feature=(ModelDriverFeature)model.getFeature(acts[i]+"轴驱动器");*//这一行调用了model对象的getFeature方法，根据字符串数组acts中的元素动态获取对应名称的特征，并将其强制转换为ModelDriverFeature类型。（acts[i]+"轴驱动器"）这里使用了字符串拼接来构造特征名称。这里的“+”是拼接的意思不是加号，比如（“1”+“1”）是11而不是相加等于2。*  
 **float** value=feature.getActuatorReference().getReference().getMotionDriver().getValue();*//这一行调用了feature对象的一系列方法来获取特征对应的值，并将其赋给value变量。*  
 **boolean** isArriveOrgin=value==0.0f;*//这一行将获取的值与0.0f进行比较，得到一个布尔值，表示该特征是否到达原点。*  
 **if**(!isArriveOrgin)  
 **return** false;{*//这是一个条件语句，如果特征未到达原点，则直接返回false*  
 }  
 }  
 **return** true; *//反之返回true*  
 }  
  
 **private** **void** stop(String actName){  
 VirtualModel model = getVirtualModel();  
 ModelDriverFeature feature=(ModelDriverFeature)model.getFeature(actName);  
 **if**(feature.isMotionRunning()){  
 feature.finishMotion();  
 }  
 }  
  
  
 */\* --X--\*\*/*  
 *// 当 X 轴绝对位置更新时触发的方法*  
 **public** **void** onX\_AbsolutePositionUpdated(Object value){  
 **float** target=(**float**)value;*// 获取目标位置*  
 move(X\_DRIVE\_NAME, X\_ARRIVE\_SIGNAL\_NAME, X\_RUNNING\_SIGNAL\_NAME, X\_SPEED\_PROPERTY,X\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,X\_CURRENT\_POSITION, target);*// 调用移动方法，将 X 轴移动到目标位置*  
 }  
 *// 当 X 轴相对位置更新时触发的方法*  
 **public** **void** onX\_RelativePositionUpdated(Object value){  
 VirtualModel model = getVirtualModel();*// 获取虚拟模型实例*  
 ModelDriverFeature feature=(ModelDriverFeature)model.getFeature(X\_DRIVE\_NAME);*// 获取 X 轴驱动器功能*  
 **float** curPosition=feature.getActuatorReference().getReference().getMotionDriver().getValue();*// 获取当前位置*  
  
 **float** target=(**float**)value+curPosition;*// 计算目标位置*  
 move(X\_DRIVE\_NAME, X\_ARRIVE\_SIGNAL\_NAME, X\_RUNNING\_SIGNAL\_NAME, X\_SPEED\_PROPERTY,X\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,X\_CURRENT\_POSITION, target);*// 调用移动方法，将 X 轴移动到目标位置*  
 }  
   
 *// 当 X 轴移动原点更新时触发的方法*  
 **public** **void** onX\_MoveOriginUpdated(Object value){  
 **boolean** b=(**boolean**)value;*// 获取移动原点信号值*  
 **if**(b){  
 **float** target=0;*// 设置目标位置为原点*  
 move(X\_DRIVE\_NAME, X\_ARRIVE\_SIGNAL\_NAME, X\_RUNNING\_SIGNAL\_NAME, X\_SPEED\_PROPERTY,X\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,X\_CURRENT\_POSITION, target);*// 调用移动方法，将 X 轴移动到原点*  
 VirtualModel model = getVirtualModel();*// 获取虚拟模型实例*  
 model.setSignalPropertyValue(X\_ABSOLUTE\_SIGNAL\_NAME, 0);*// 将 X 轴绝对位置信号设置为 0*  
 }   
 }  
  
 *// 当 X 轴停止移动更新时触发的方法*  
 **public** **void** onX\_StopMoveUpdated(Object value){  
 **if**((**boolean**)value){*// 如果为真*  
 stop(X\_DRIVE\_NAME);*// 调用停止移动方法，停止 X 轴移动*  
 }  
 }  
  
 *// 当 X 轴移动速度更新时触发的方法*  
 **public** **void** onX\_MoveSpeedUpdated(Object value){  
 **float** v=(**float**)value;*// 获取移动速度*  
 VirtualModel model=getVirtualModel();*// 获取虚拟模型实例*  
 model.setCustomerPropertyValue(X\_SPEED\_PROPERTY, v);*// 设置 X 轴移动速度属性值*  
 }  
  
 */\*Y\*\*/*  
 *// 当 Y 轴绝对位置更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onY\_AbsolutePositionUpdated(Object value){  
 **float** target=(**float**)value;  
 move(Y\_DRIVE\_NAME, Y\_ARRIVE\_SIGNAL\_NAME, Y\_RUNNING\_SIGNAL\_NAME, Y\_SPEED\_PROPERTY,Y\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,Y\_CURRENT\_POSITION, target);  
 }  
   
 *// 当 Y 轴相对位置更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onY\_RelativePositionUpdated(Object value){  
 VirtualModel model = getVirtualModel();  
 ModelDriverFeature feature=(ModelDriverFeature)model.getFeature(Y\_DRIVE\_NAME);  
 **float** curPosition=feature.getActuatorReference().getReference().getMotionDriver().getValue();  
  
 **float** target=(**float**)value+curPosition;  
 move(Y\_DRIVE\_NAME, Y\_ARRIVE\_SIGNAL\_NAME, Y\_RUNNING\_SIGNAL\_NAME, Y\_SPEED\_PROPERTY,Y\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,Y\_CURRENT\_POSITION, target);  
 }  
  
 *// 当 Y 轴移动原点更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onY\_MoveOriginUpdated(Object value){  
 **boolean** b=(**boolean**)value;  
 **if**(b){  
 **float** target=0;  
 move(Y\_DRIVE\_NAME, Y\_ARRIVE\_SIGNAL\_NAME, Y\_RUNNING\_SIGNAL\_NAME, Y\_SPEED\_PROPERTY,Y\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,Y\_CURRENT\_POSITION, target);  
 VirtualModel model = getVirtualModel();  
 model.setSignalPropertyValue(Y\_ABSOLUTE\_SIGNAL\_NAME, 0);  
 }   
 }  
  
 *// 当 Y 轴停止移动更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onY\_StopMoveUpdated(Object value){  
 **if**((**boolean**)value){  
 stop(Y\_DRIVE\_NAME);  
 }  
 }  
  
 *// 当 Y 轴移动速度更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onY\_MoveSpeedUpdated(Object value){  
 **float** v=(**float**)value;  
 VirtualModel model=getVirtualModel();  
 model.setCustomerPropertyValue(Y\_SPEED\_PROPERTY, v);  
 }  
  
 */\*Z\*\*/*  
 *// 当 Z 轴绝对位置更新时触发的方法*  
 **public** **void** onZ\_AbsolutePositionUpdated(Object value){  
 **float** target=(**float**)value;*// 获取目标位置*  
 move(Z\_DRIVE\_NAME, Z\_ARRIVE\_SIGNAL\_NAME, Z\_RUNNING\_SIGNAL\_NAME, Z\_SPEED\_PROPERTY,Z\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,Z\_CURRENT\_POSITION, target); *// 调用移动方法，将 Z 轴移动到目标位置*  
 }  
  
 *// 当 Z 轴相对位置更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onZ\_RelativePositionUpdated(Object value){  
 VirtualModel model = getVirtualModel();*// 获取虚拟模型实例*  
 ModelDriverFeature feature=(ModelDriverFeature)model.getFeature(Z\_DRIVE\_NAME);*// 获取 Z 轴驱动器功能*  
 **float** curPosition=feature.getActuatorReference().getReference().getMotionDriver().getValue();  
  
 **float** target=(**float**)value+curPosition;  
 move(Z\_DRIVE\_NAME, Z\_ARRIVE\_SIGNAL\_NAME, Z\_RUNNING\_SIGNAL\_NAME, Z\_SPEED\_PROPERTY,Z\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,Z\_CURRENT\_POSITION, target);*// 调用移动方法，将 Z 轴移动到目标位置*  
 }  
  
 *// 当 Z 轴移动原点更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onZ\_MoveOriginUpdated(Object value){  
 **boolean** b=(**boolean**)value; *// 获取移动原点信号值*  
 **if**(b){  
 **float** target=0;*// 设置目标位置为原点*  
 move(Z\_DRIVE\_NAME, Z\_ARRIVE\_SIGNAL\_NAME, Z\_RUNNING\_SIGNAL\_NAME, Z\_SPEED\_PROPERTY,Z\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,Z\_CURRENT\_POSITION, target);*// 调用移动方法，将 Z 轴移动到原点*  
 VirtualModel model = getVirtualModel();*// 获取虚拟模型实例*  
 model.setSignalPropertyValue(Z\_ABSOLUTE\_SIGNAL\_NAME, 0); *// 将 Z 轴绝对位置信号设置为 0*  
 }   
 }  
  
 *// 当 Z 轴停止移动更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onZ\_StopMoveUpdated(Object value){  
 **if**((**boolean**)value){  
 stop(Z\_DRIVE\_NAME); *// 调用停止移动方法，停止 Z 轴移动*  
 }  
 }  
  
 *// 当 Z 轴移动速度更新时触发的方法，逻辑与 X 轴类似，只是操作对象不同*  
 **public** **void** onZ\_MoveSpeedUpdated(Object value){  
 **float** v=(**float**)value;*// 获取移动速度*  
 VirtualModel model=getVirtualModel();*// 获取虚拟模型实例*  
 model.setCustomerPropertyValue(Z\_SPEED\_PROPERTY, v); *// 设置 Z 轴移动速度属性值*  
 }  
  
 */\*R\*\*\*/*  
 *// 当 R 轴绝对位置更新时触发的方法，逻辑与 Z 轴类似，只是操作对象不同*  
 **public** **void** onR\_AbsolutePositionUpdated(Object value){  
 **float** target=(**float**)value;  
 move(R\_DRIVE\_NAME, R\_ARRIVE\_SIGNAL\_NAME, R\_RUNNING\_SIGNAL\_NAME, R\_SPEED\_PROPERTY,R\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,R\_CURRENT\_POSITION, target);  
 }  
  
 *// 当 R 轴相对位置更新时触发的方法，逻辑与 Z 轴类似，只是操作对象不同*  
 **public** **void** onR\_RelativePositionUpdated(Object value){  
 VirtualModel model = getVirtualModel();  
 ModelDriverFeature feature=(ModelDriverFeature)model.getFeature(R\_DRIVE\_NAME);  
 **float** curPosition=feature.getActuatorReference().getReference().getMotionDriver().getValue();  
  
 **float** target=(**float**)value+curPosition;  
 move(R\_DRIVE\_NAME, R\_ARRIVE\_SIGNAL\_NAME, R\_RUNNING\_SIGNAL\_NAME, R\_SPEED\_PROPERTY,R\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,R\_CURRENT\_POSITION, target);  
 }  
  
 *// 当 R 轴移动原点更新时触发的方法，逻辑与 Z 轴类似，只是操作对象不同*  
 **public** **void** onR\_MoveOriginUpdated(Object value){  
 **boolean** b=(**boolean**)value;  
 **if**(b){  
 **float** target=0;  
 move(R\_DRIVE\_NAME, R\_ARRIVE\_SIGNAL\_NAME, R\_RUNNING\_SIGNAL\_NAME, R\_SPEED\_PROPERTY,R\_ARRIVE\_ORIGIN\_SIGNAL\_NAME,R\_CURRENT\_POSITION, target);  
 VirtualModel model = getVirtualModel();  
 model.setSignalPropertyValue(R\_ABSOLUTE\_SIGNAL\_NAME, 0);  
 }   
 }  
  
 *// 当 R 轴停止移动更新时触发的方法，逻辑与 Z 轴类似，只是操作对象不同*  
 **public** **void** onR\_StopMoveUpdated(Object value){  
 **if**((**boolean**)value){  
 stop(R\_DRIVE\_NAME);  
 }  
 }  
  
 *// 当 R 轴移动速度更新时触发的方法，逻辑与 Z 轴类似，只是操作对象不同*  
 **public** **void** onR\_MoveSpeedUpdated(Object value){  
 **float** v=(**float**)value;  
 VirtualModel model=getVirtualModel();  
 model.setCustomerPropertyValue(R\_SPEED\_PROPERTY, v);  
 }  
  
 **public** **void** onRedEnabledUpdated(Object value){  
 **boolean** b=(**boolean**)value;  
 VirtualModel model =getVirtualModel();  
 renderThreadHelper.runInsideRenderAsync(()->{  
 model.setSignalPropertyValue("RedStatus", b);   
 });  
 }  
  
 **public** **void** onGreenEnabledUpdated(Object value){*//这段代码是一个Java方法的定义，名为onGreenEnabledUpdated，接受一个Object类型的参数value。*  
 **boolean** b=(**boolean**)value;*//这一行将传递给方法的value转换为boolean类型并赋值给变量b。在这里，(boolean)是一个类型转换操作符，将value强制转换为boolean类型。*  
 VirtualModel model =getVirtualModel();*//这一行声明并初始化了一个VirtualModel类型的变量model，并调用了一个方法getVirtualModel()来获取VirtualModel的实例。*  
 renderThreadHelper.runInsideRenderAsync(()->{*//这一行调用了renderThreadHelper的runInsideRenderAsync方法，并传入一个Lambda表达式作为参数。Lambda表达式的作用是在渲染线程中异步执行其中的代码块。*  
 model.setSignalPropertyValue("GreenStatus", b);*//在Lambda表达式中，调用了model对象的setSignalPropertyValue方法，将"GreenStatus"作为属性名，b作为属性值，来设置VirtualModel的信号属性值。*  
 });  
 }  
  
 *// 当光源状态更新时触发的方法*  
 **public** **void** onLightEnabledUpdated(Object value){  
 **boolean** b=(**boolean**)value;  
 VirtualModel model =getVirtualModel();  
 model.setSignalPropertyValue("LightState", b);  
 info("光源：{}", b?"开":"关");*//输出信息，里面用了一个三元表达式，当变量b为true时输出“光源：开”，反之输出“光源：关”*  
 }  
  
 *// 当拍摄信号更新时触发的方法*  
 **public** **void** onScanUpdated(Object value){  
 **boolean** b=(**boolean**)value;  
 VirtualModel model =getVirtualModel();  
 model.setSignalPropertyValue("InstallCameraScan", b);*// 设置属性值*  
 }  
  
 *// 当吸取信号更新时触发的方法*  
 **public** **void** onSuckUpdated(Object value){  
 **boolean** b=(**boolean**)value;  
 VirtualModel model =getVirtualModel();  
 model.setSignalPropertyValue("InstallSuck", b); *// 设置吸取信号属性值*  
 }  
  
 *// 当高亮信号更新时触发的方法*  
 **public** **void** onHeightLightUpdated(Object value){  
 **boolean** b=(**boolean**)value;  
 VirtualModel model =getVirtualModel();  
 model.setSignalPropertyValue("InstallHeightLight", b);*// 设置高亮信号属性值*  
 }  
  
 *// 当安装吸取完成信号更新时触发的方法*  
 **public** **void** onInstallSuckDoneUpdated(Object value){  
 **boolean** b=(**boolean**)value;  
 VirtualModel model =getVirtualModel();  
 model.setSignalPropertyValue("SuckDone", b);*// 设置安装吸取完成信号属性值。*  
 }  
  
 **public** **void** onInstallScanDoneDoneUpdated(Object value){*//这是一个Java方法的定义，方法名为onInstallScanDoneDoneUpdated，接受一个Object类型的参数value。public void: 这是方法的访问修饰符。public表示该方法可以被任何类访问，void表示该方法不返回任何值。onInstallScanDoneDoneUpdated: 这是方法的名称，采用驼峰命名法。(Object value): 这是方法的参数列表。Object是Java中的基础类，表示所有其他类的父类，因此它可以接受任何类型的参数。value是参数的名称，该参数用于接收传递给方法的值。*  
 **boolean** b=(**boolean**)value;*//这一行将传递给方法的value转换为boolean类型并赋值给变量b。在这里，(boolean)是一个类型转换操作符，将value强制转换为boolean类型。*  
 VirtualModel model =getVirtualModel();*//这一行声明并初始化了一个VirtualModel类型的变量model，并调用了一个方法getVirtualModel()来获取VirtualModel的实例。*  
 model.setSignalPropertyValue("SuckDone", b);*//这一行调用了model对象的setSignalPropertyValue方法，将"SuckDone"作为属性名，b作为属性值，来设置VirtualModel的信号属性值。*  
 }  
  
   
}