MDCS 系统指南书

目录

一，MDCS 环境部署

一，Linux 运行环境安装

1,用环境配置脚本一键配置 mono 和.Net 环境。

将 **env_setup.tar.gz** 文件拷贝至 MDCS 文件夹相同路径，并解压 `sudo tar xzf env_setup.tar.gz` 进入 env_setup 文件夹，根据 ReadMe 的说明执行命令，运行安装脚本

2,自行配置

1. 配置 mono

根据 mono 官网要求和系统版本配置 mono 环境，要求 mono 版本>=6.12
Download - Stable | Mono (mono-project.com)

（1）联网安装

```
sudo apt install gnupg ca-certificates
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF
echo "deb https://download.mono-project.com/repo/ubuntu stable-bionic main"
| sudo tee /etc/apt/sources.list.d/mono-official-stable.list
sudo apt update
sudo apt install mono-devel
```

（2）离线安装

拷贝 6.12 的安装包至没网的电脑。然后执行以下命令。解压完后将解压目录下对应的文件夹里的内容拷贝至/usr/local 目录下对应的文件夹内。

```
tar xvf mono-6.12-linux-x64.tar.gz
```

2. 配置.Net

根据.Net 官网要求和系统版本配置 dotnet 环境，要求 dotnet 版本=5
在 Linux 上手动安装 .NET - .NET | Microsoft Docs

（1）联网

```
sudo apt-get update; \
sudo apt-get install -y apt-transport-https && \
sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-5.0
```

（2）离线

Download .NET 5.0 (Linux, macOS, and Windows) (microsoft.com) 在有网的电脑上下载对应架构的二进制文件压缩包，拷贝至没网的电脑。然后在/home/usename/MDCS/DetouLite/DetourLite 路径下执行以下命令

```
export DOTNET_ROOT=~/.dotnet
mkdir -p "$DOTNET_ROOT" && tar zxf dotnet-sdk-6.0.100-linux-x64.tar.gz -C
"$DOTNET_ROOT"
export PATH=$PATH:$DOTNET_ROOT
```

前面的 export 命令只会使 .NET CLI 命令对运行它的终端会话可用。 你可以编辑 shell 配置文件，永久地添加这些命令。 Linux 提供了许多不同的 shell，每个都有不同的配置文件。 例如：

Bash Shell：~/.bash_profile、~/.bashrc

Korn Shell：~/.kshrc 或 .profile

Z Shell：~/.zshrc 或 .zprofile

为 shell 编辑相应的源文件，并将 :$~/.dotnet 添加到现有 PATH 语句的末尾。 如果不包含 PATH 语句，则使用 export PATH=$PATH:$~/.dotnet 添加新行。

另外，将 export DOTNET_ROOT=$~/.dotnet 添加至文件的末尾。

二，Windows 运行环境安装

无需安装环境。

二，MDCS 使用方法

一，Linux 系统使用

（本内容只写了区别于 windows 部分，其他内容与 windows 一样使用）

1,路径配置

为了确保以下所有的配置和操作在任意 linux 系统中都适配，请按照以下要求构建文件夹保存 MDCS 文件:

在任意位置创建 MDCS 文件夹，文件夹下创建 Medulla、Clumsy 文件夹，每个文件夹下分别存放对应软件的相关文件。MDCS 文件夹下创建 DetouLite 文件夹，DetourLite 文件下再创建 DetourLite 文件夹用以存放相关软件，同时创建 lib 文件夹用以存放 libOpenCvSharpExtern.so。

以下操作以文件夹创建在 home/username 为例，若创建在其他路径请自行修改对应路径。

1. Medulla 配置

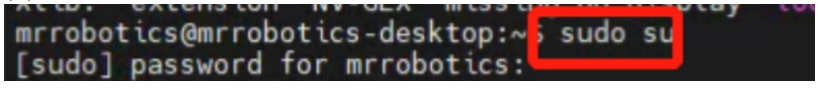
1. 首先 Medulla 文件夹里至少要有 Medulla.exe 和 plugins 文件夹，plugins 文件夹里包含 CartActivator.dll 和 LidarController.dll 等 dll 文件。
2. 将生成编译的 Medulla 适配程序（例 xxx_m.dll）放入同目录下的 plugins 文件夹，如果适配程序还引用了其它的库，也一并放入 plugins 文件夹（例如雷达、USB 摄像头等）

在 medulla 目录下新建 startup.iocmd 文件，拷入以下内容

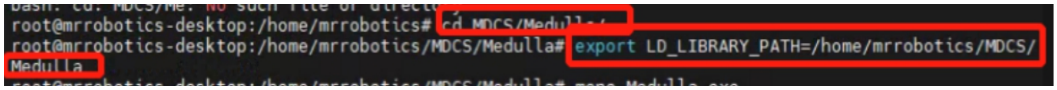
```
loader = io load plugins/CartActivator.dll
cart = loader load plugins/xxx.dll
lidar = io load plugins/LidarController.dll
frontlidar = lidar init SimulateLidar
```

4. 运行 Medulla.exe，如果打不开出现闪退，可能的原因是 MSVC 运行库未安装，官方下载地址。

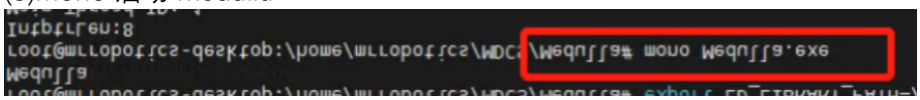
(1) 进入 ssh,获取最高权限 sudo su



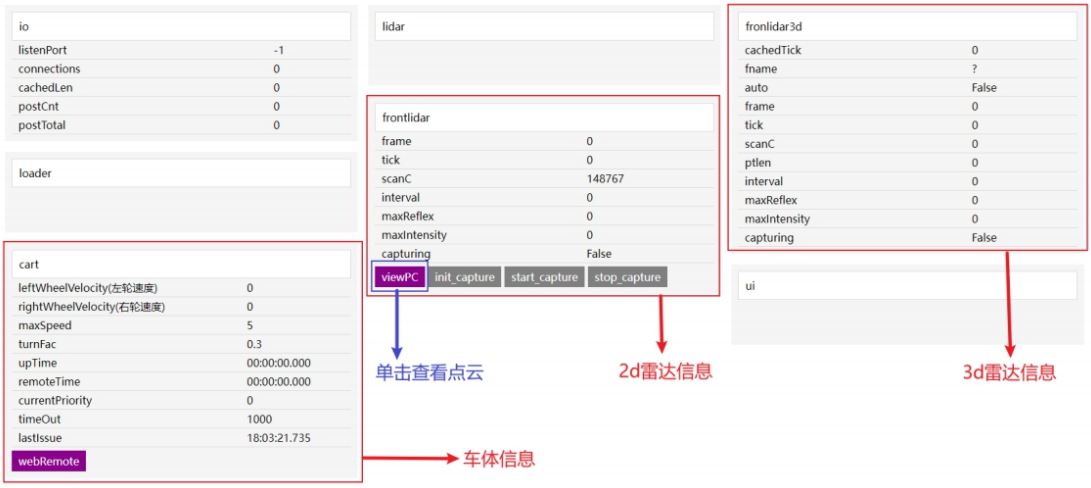
(2)进入 MDCS/Medulla 目录配置 Medulla 环境 export LD_LIBRARY_PATH=/home/username/MDCS/Medulla



(3)mono 启动 Medulla



5，打开 Edge 浏览器，输入 http://{CarIP}:8007/simple.html 进入 medulla web 界面。（请确保使用 Edge 打开页面）



2. Detour 配置

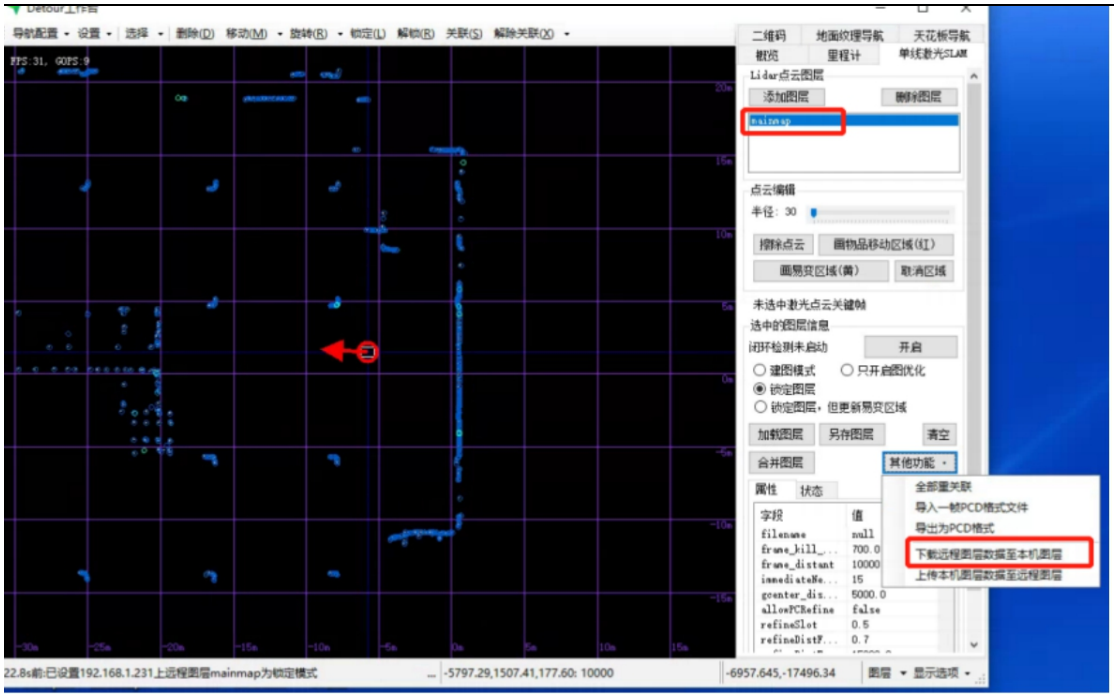
1. 打开 Detour 之前要先打开 Medulla
2. 首先 DetourLite/DetourLite 文件夹里至少要有 DetourLite.dll，DetourLite/lib 文件夹下要有 libOpenCvSharpExtern.so，没有 so 文件可能会闪退。
3，打开流程：

（1）进入 ssh，获取 root 权限，进入 cd MDCS/DetourLite 目录

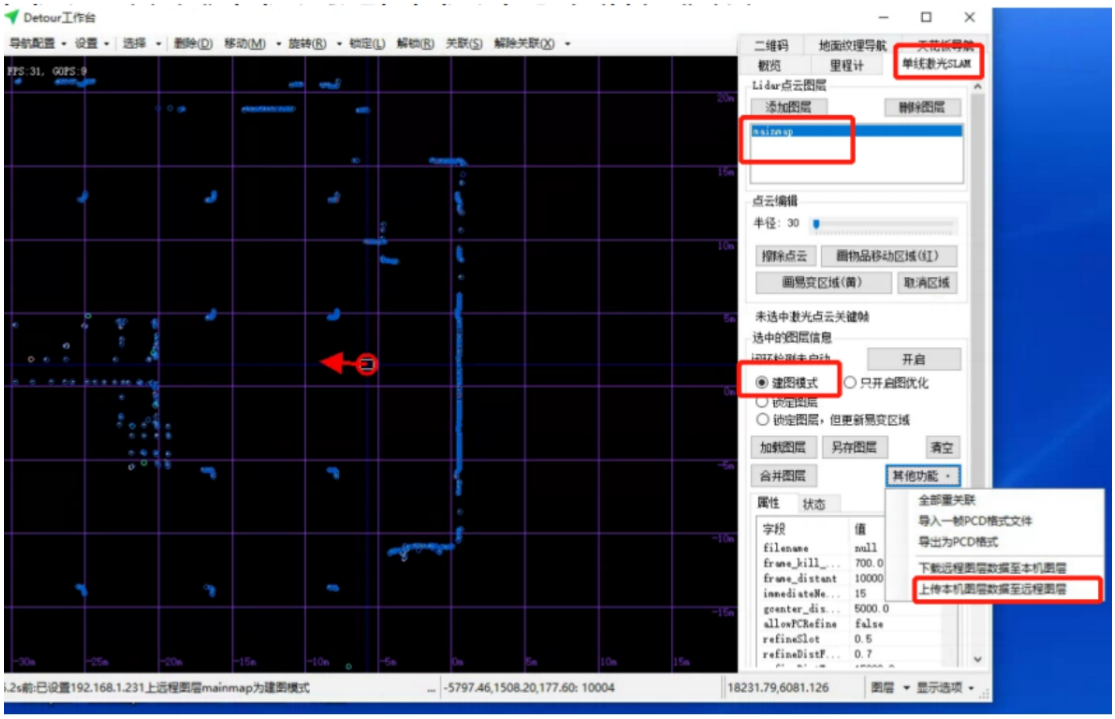
（2）打开 detour 软件，输入~/.dotnet/dotnet DetourLite.dll

（3）打开 Detour 软件（自己或服务器电脑桌面上或者 MDCS 文件夹里），选择概览-使用远程算法核-远程设备，输入 IP 点击连接

4. 如果地图已经建好，请选择 单线激光 SlAM-选择图层-下载远程地图即可



5. 如果未建图，选择建图模式-遥控小车进行运行业务路线完成后，锁定图层-上传本机图层到远程图层即可

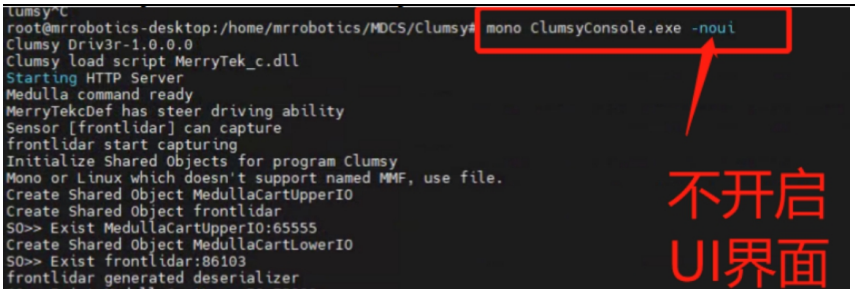


3. Clumsy 配置

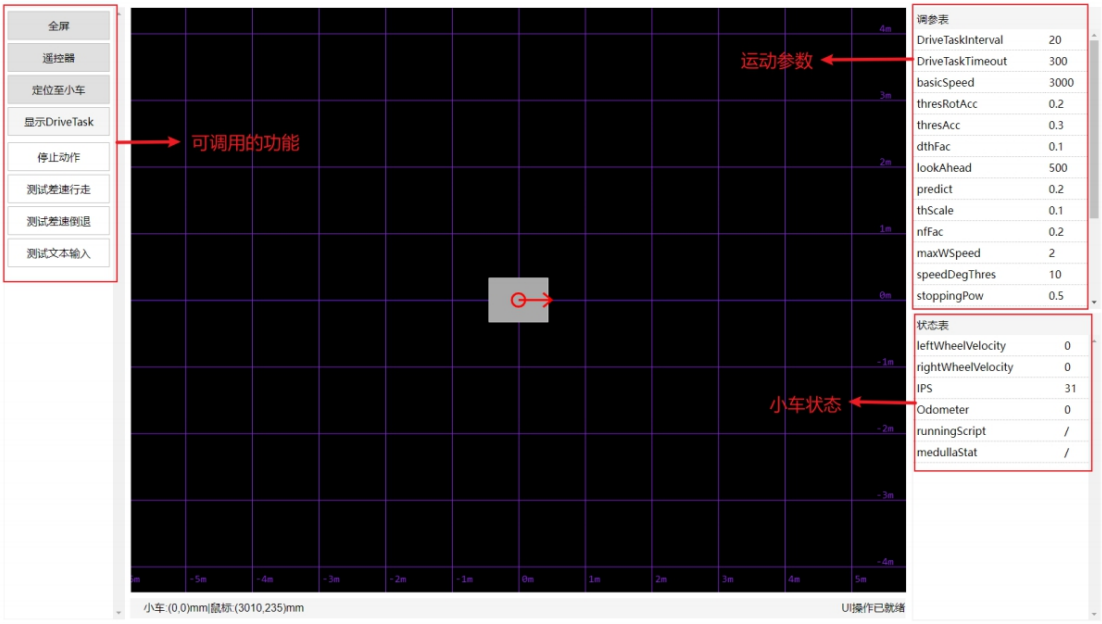
打开 Clumsy 之前要先打开 Medulla、Detour。

1. 进入 ssh,获取 root 权限，进入小车 Clumsy 目录 cd MDCS/Clumsy
2. 配置 Clums 运行环境 export LD_LIBRARY_PATH=/home/username/MDCS/Clumsy

3. 打开 Clumsy 输入 mono Clumsy.exe -noui



4. 打开浏览器，输入 http://IP:8008/simple.html 进入 clusmy web 界面



4，Simple 部分使用见 Windows 系统使用介绍

二，Windows 系统使用

一，Medulla

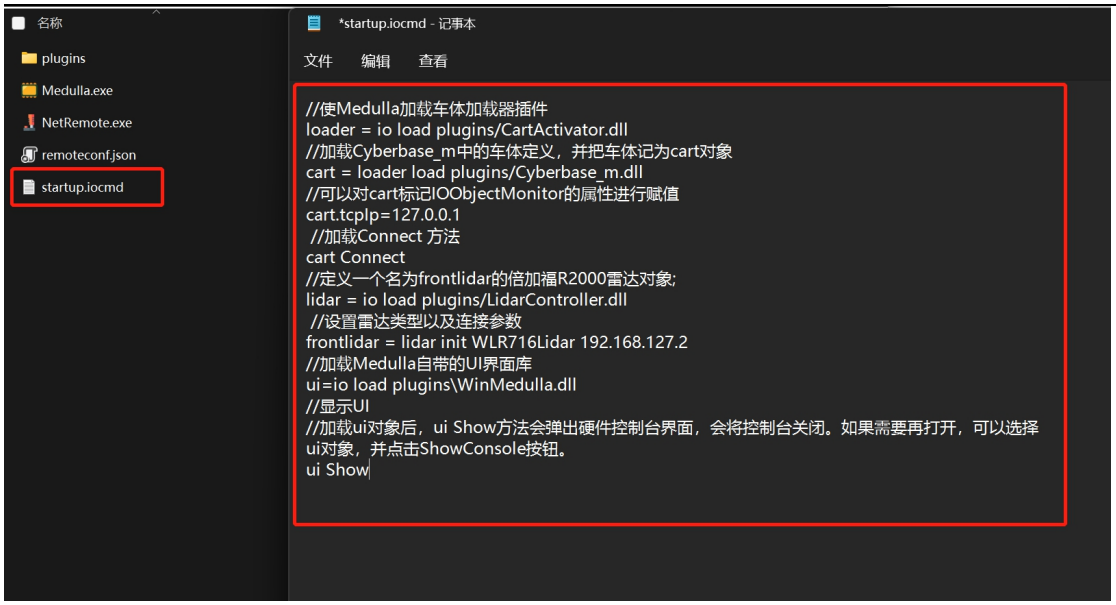
medulla 提供通信抽象，通过插件对接各类硬件。开发者适配车型后，应使用遥控器应用进行遥控车体、试验 IO 等操作，来验证适配结果，确保硬件适配层不会故障或崩溃。

1，startup.iocmd 配置

（1）首先 medulla 文件夹里至少要有 Medulla.exe 和 plugins 文件夹，文件夹里包含 CartActivator.dll 和 WinMedulla.dll

（2）将生成编译的 Medulla 适配程序（例 Cyberbase_m.dll）放入同目录下的 plugins 文件夹，如果适配程序还引用了其它的库，也一并放入 plugins 文件夹（例如雷达、USB 摄像头等）

（3）在 medulla 目录下新建 startup.iocmd 文件,具体规则如下：




```
//使 Medulla 加载车体加载器插件
loader = io load plugins/CartActivator.dll
//加载 Cyberbase_m 中的车体定义，并把车体记为 cart 对象
cart = loader load plugins/Cyberbase_m.dll
//可以对 cart 标记 IOObjectMonitor 的属性进行赋值
cart.tcpIp=127.0.0.1
//加载 Connect 方法
cart Connect
//定义一个名为 frontlidar 的倍加福 R2000 雷达对象;
lidar = io load plugins/LidarController.dll
//设置雷达类型以及连接参数
frontlidar = lidar init WLR716Lidar 192.168.127.2
如果倒装，还需要输入：frontlidar setMirror true
//加载 Medulla 自带的 UI 界面库
ui=io load plugins\WinMedulla.dll
//显示 UI 加载 ui 对象后，ui Show 方法会弹出硬件控制台界面，会将控制台关闭。如果需要再打开，可以选择 ui 对象，并点击 ShowConsole 按钮。
ui Show
```

（2）如果需要增加其他传感器，如视觉相机就按照以上格式加入进去即可

```
mv = io load plugins\MVCamera.dll
maincam=mv initMono CA013
```

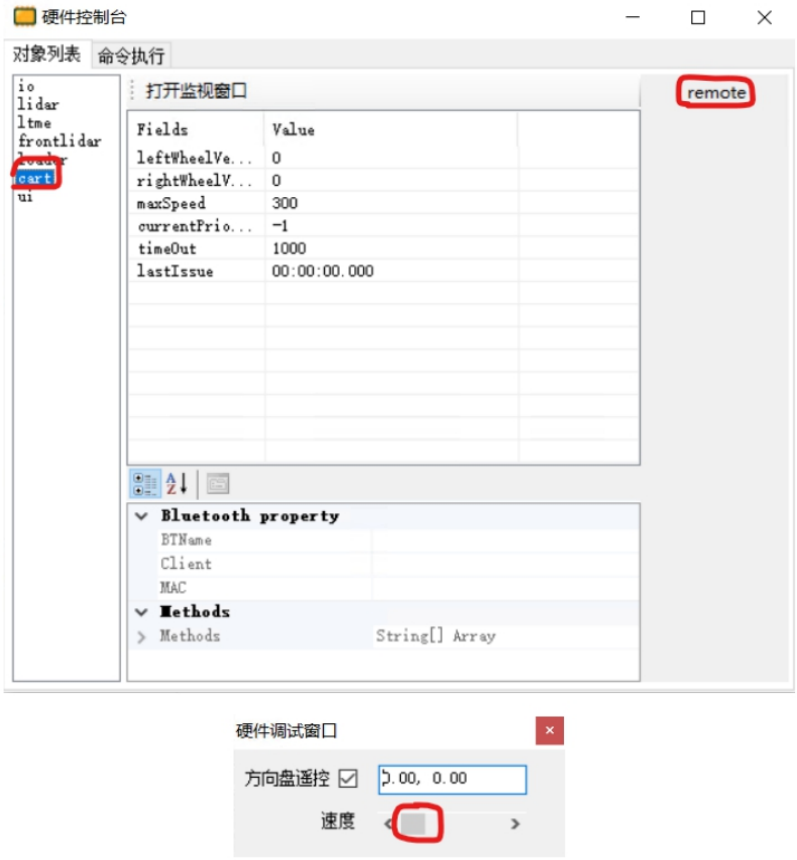
其中 CA013 是摄像头型号，initMono 为以单色方式启动。若相机是彩色相机，则使用 initColor 方法

2，遥控器的使用

1，在 windows 部署的可以使用 medulla 软件自带的 remote 遥控器，只有运控控制遥控

（1）．打开 Medulla.exe

（2）．点击左侧的 cart，然后点击右侧的 remote，出现“硬件测试窗口”，在“方向盘遥控”勾选上的情况下先把“速度”拖动条向右拖动一小部分，然后使用方向键↑↓←→即可控制小车前后左右移动

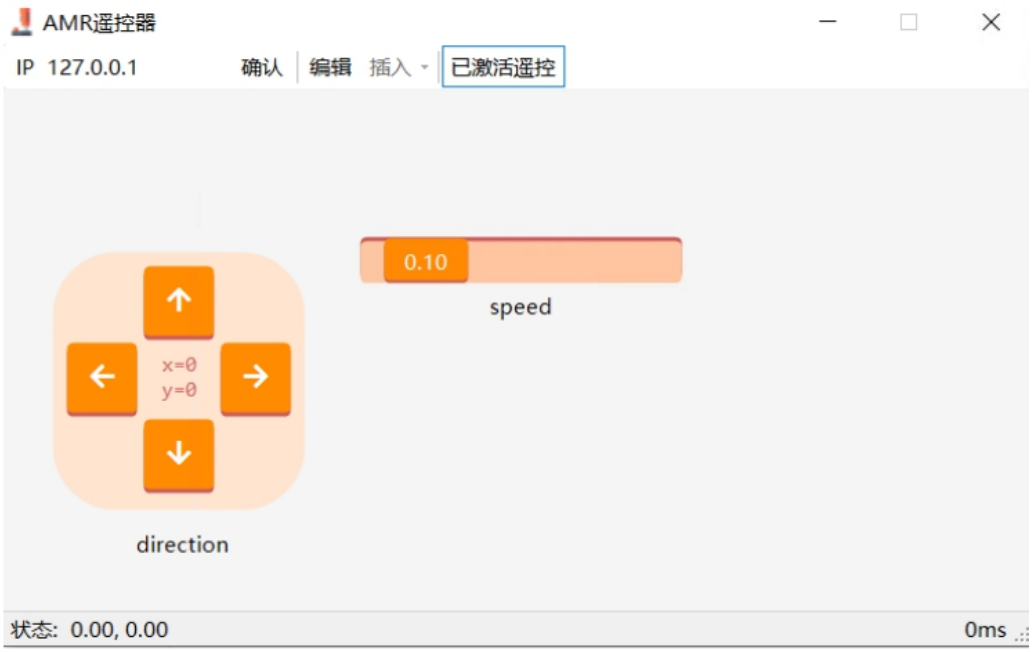


2，使用 netRemote 遥控器

1. NetRemote.exe 主程序、remoteconf.json 配置文件和 NetRemote.apk 安卓软件安装包默认存放在 medulla 目录下，也可放在其他位置

2. 当没有 json 配置文件时，打开 NetRemote.exe 会使用默认布局，左上角可更改 ip，点击“编辑”后可对小组件进行编辑，鼠标左键按住拖动位置，右键可修改名称、删除组件或设定键位，点击“保存”会覆盖掉同目录下的 remoteconf.json

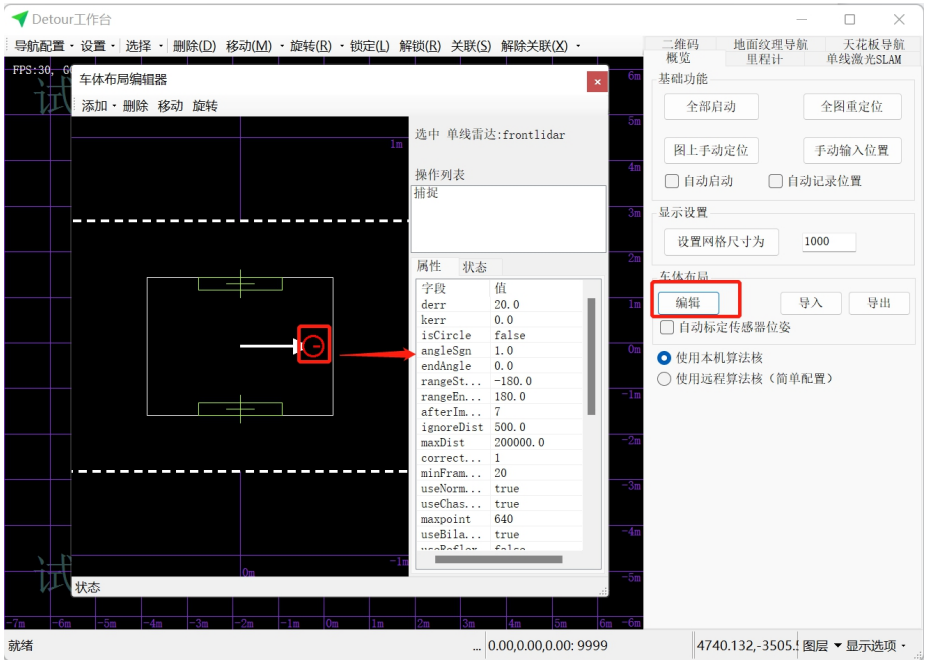
3. 若要正常使用 netremote 需要 medulla 设为管理员启动，并且小组件的名称需对应，其中 directio 需设置方向键键位，测试使用速度不能太大，在 0.10 左右



二，Detour

1,Detour 基本知识

1，Detour-车体编辑器



概念 3-1-1　车体编辑器，是一个用于编辑车体模型，激光参数的配置工具。

概念 3-1-1　车体编辑器-单线雷达位置(x,y,th)，表示雷达相对驱动中心位置，mm 单位。

概念 3-1-2　车体编辑器-单线雷达角度方向(angleSgn：±1)，表示雷达角度方向，如雷达倒装情况。

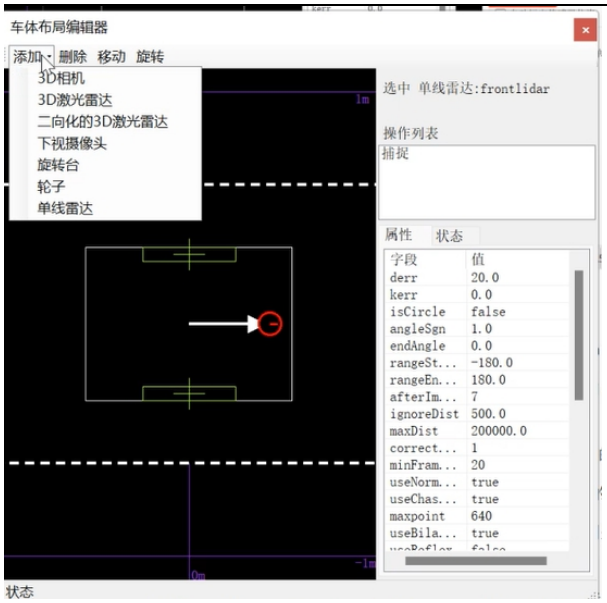
概念 3-1-1　车体编辑器-单线雷达起始角度(rangeStartAngle)，表示雷达开始扫描的角度值。

概念 3-1-2　车体编辑器-单线雷达结束角度(rangeEndAngle)，表示雷达结束扫描的角度值。

概念 3-1-3　车体编辑器-单线雷达扫描开始距离(ignoreDist)，用于屏蔽激光周围杂点或车体部分垃圾数据，mm 单位。

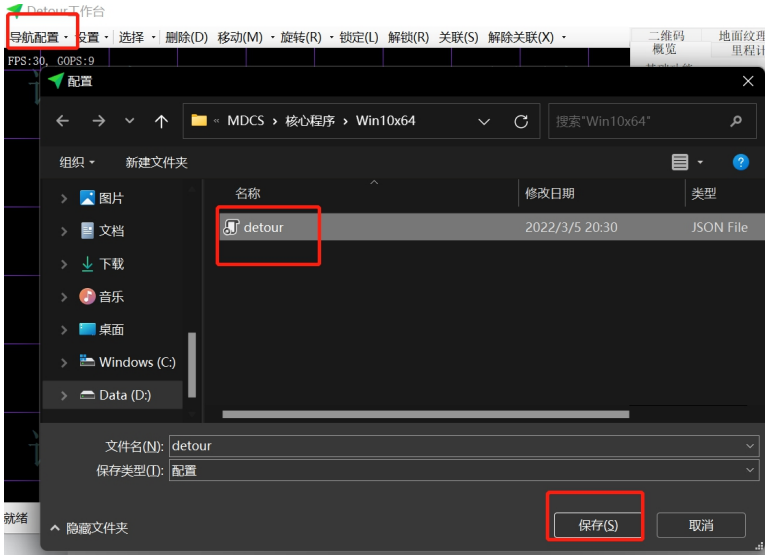
概念 3-1-4　车体编辑器-单线雷达最大扫描距离(maxDist)，表示雷达扫描最大半径，mm 单位。

1，如果我们有其他外设，可在添加中添加并且配置



2，Detour-detour.json 常用参数配置

1， 我们先打开 Detour 软件，然后界面左上角，导航配置，选择保存到 Detour 文件夹下，命名 detour 即可，会保存一个 json 配置文件，里面存储了 Detour 初始的参数，有些我们需要设置下会方便调试如下



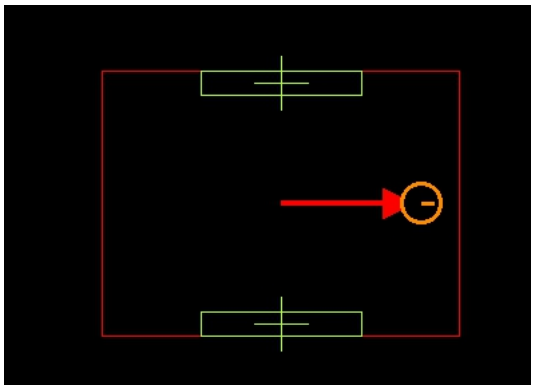
2， 具体常用的 配置参数如下，我们先打开 detour.json 文件（右击记事本打开）
"recordLastPos": true, //代表自动记录上一个位置信息，可以用于断电后或 detour 关闭后，再次打开 detour 会自动定位小车断电前的或关闭前记录的位置信息，可以省去重复的手动定位操作
"autoStart": true,//表示打开软件是否自动启动激光，代替如下图按钮

二，Detour 基础设置

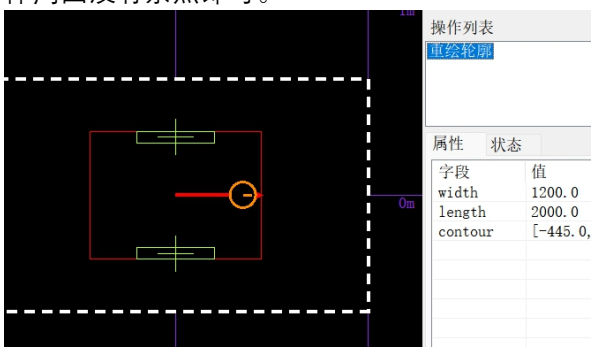
1，车体绘制

用于过滤车体部分的激光数据，以免影响后期建图

1.1 打开 Detour-概览-车体布局-选择默认车体边框，如果为红色说明选中，如下

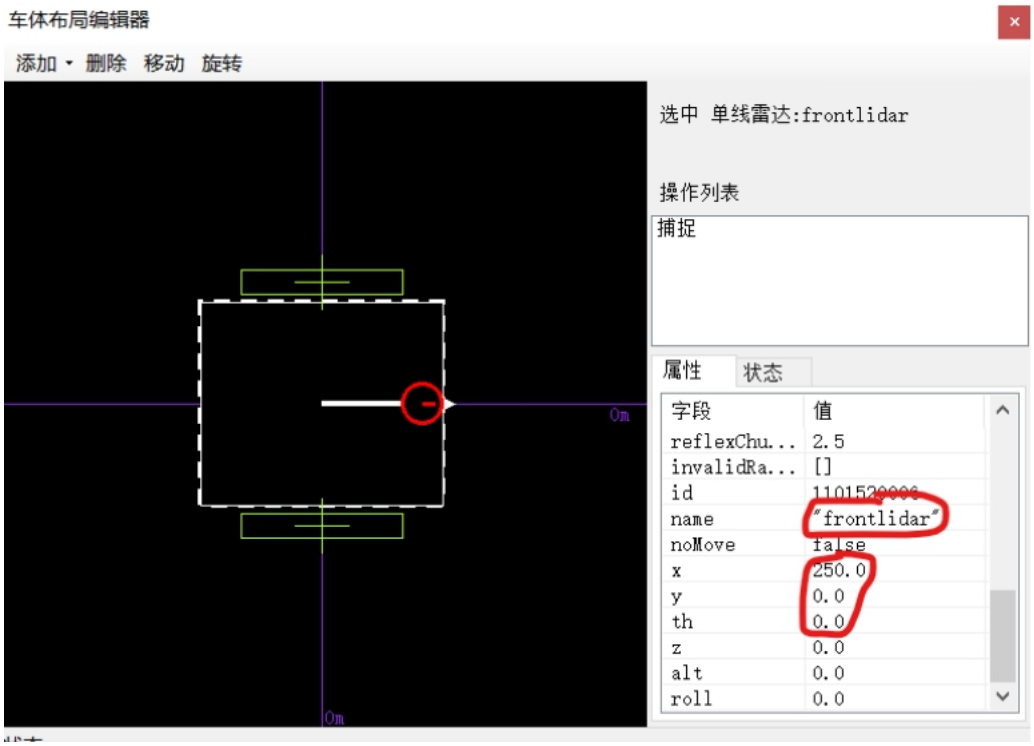


2，我们选择重绘值轮廓，在车体编辑器右上角，双击，然后我们我们根据车体大小 绘制出车体轮廓即可，下图由于离线，所以无法看到激光数据，正常是可以看到激光数据的，我们尽量让激光车体周围没有杂点即可。



3，激光参数

1，设置激光雷达参数和位置



完成车体绘制后，我们选中激光，右侧会刷新出属性，我们需要修改的主要是 xyth 参数，name 要与 medulla 里的 startup.iocmd 中的雷达对应，这里即为“frontlidar”，如果 medulla 里新添加了雷达如 backlidar，则车体布局编辑器中新添加雷达 name 也要改为“backlidar”， 其他参数 angleSgn 用于指示激光雷达扫描方向，1 为逆时针扫描，-1 为顺时针扫描。endAngle，为激光雷达扫描的最后一个点的角度。这两个参数必须配置，常见的雷达配置如下：

1. 倍加福、万集、科力、西克 Nano 系列雷达， angleSgn=-1, endAngle=180
2. 星秒雷达：angleSgn=1，endAngle=0
3. 西克 LMS 系列雷达，angleSgn=1, endAngle=270

激光物体参数填写完成后，我们需要对激光再次进行标定，来修正机械装配误差。

2，标定雷达位置/精度标定

切换到概览面板，点击全部启动，此时可在地图上看到彩色的当前的局部地图。移动物体会被标记为灰色，被遮挡的局部地图以白色显示。并引出若干根从雷达位置射向移动物体的线。用户接下来应对雷达进行位置标定，从而更方便地使用定位数据（不标定则会导致运动轨迹怪异）。以下以一些常用车型为例：

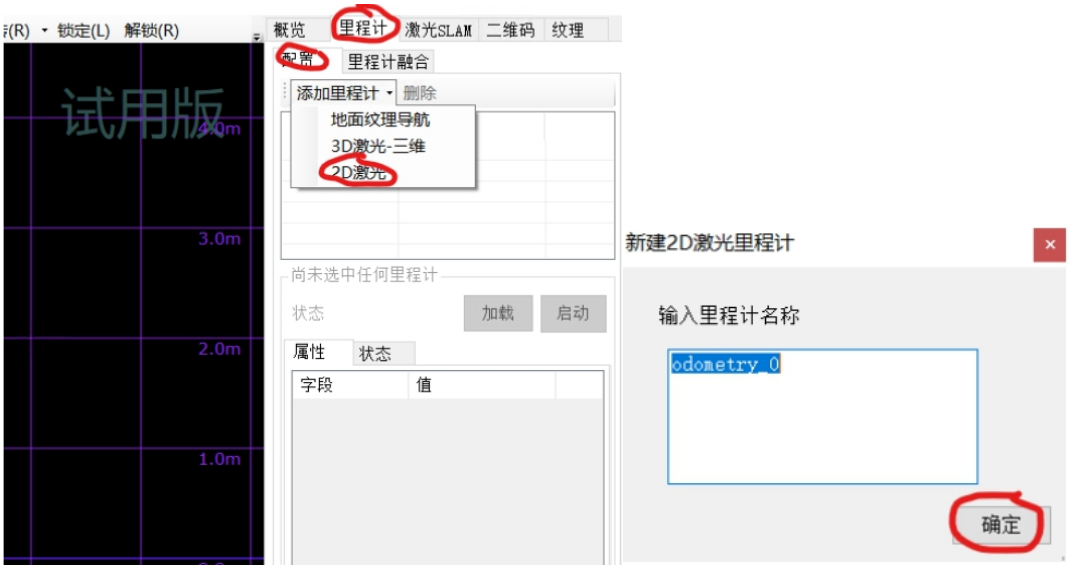
1. 对于差速运行式车型，一般输出位姿应为车体旋转中心。该车型的校准包括以下两步骤：
2. 调整角度。首先点击概览-手动输入位置，将位置重置为 0,0,0，然后手动行驶车辆准直向前行驶若干距离，查看 y 轴偏差、补偿雷达的 th（角度）并重新将位置设为 0,0,0 并重复上述调整过程，直到手动行驶车辆准直向前时，y 轴无偏差。
3. 调整 xy 坐标。首先点击概览-手动输入位置，将位置重置为 0,0,0。然后手动旋转车体 180 度，将状态栏中读到的 xy 坐标除以 2 输入到雷达的 xy 坐标中。重复该操作直到误差消除。
4. 对于单舵轮运行车型，一般输出位姿为舵轮的位姿。主要补偿其角度。首先手动输入位置为 0,0,0。然后手动行驶车辆准直向前行驶若干距离，查看 y 轴偏差并补偿雷达的 th（角度），直到手动行驶车辆准直向前时，y 轴无偏差即可。
5. 对于多舵轮运行车辆，一般输出位姿为以车体旋转中心。其校准方法类似于差速运行式车型

3，里程计配置

1. 点击里程计→配置→添加里程计→2D 激光，输入（默认）名字点击“确定”添加默认 2D 激光里程

计

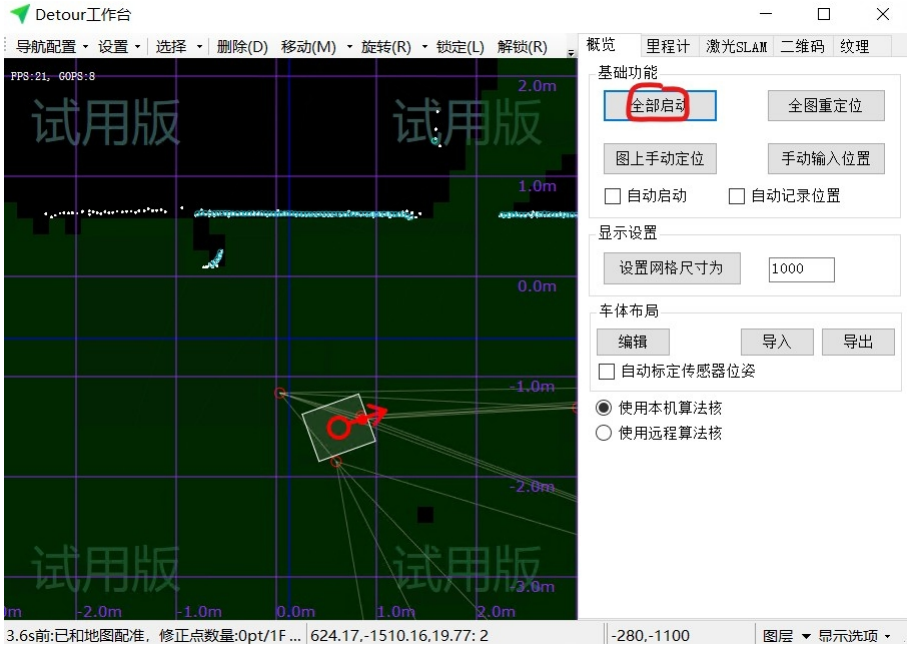
2. 2D 激光里程计属性一般保持默认即可。可以调节 switchingDist 属性，该属性表示最多靠激光里程计走多少距离必须发生一次回环检测，一般设置为 1000-7000mm。



3，录制地图

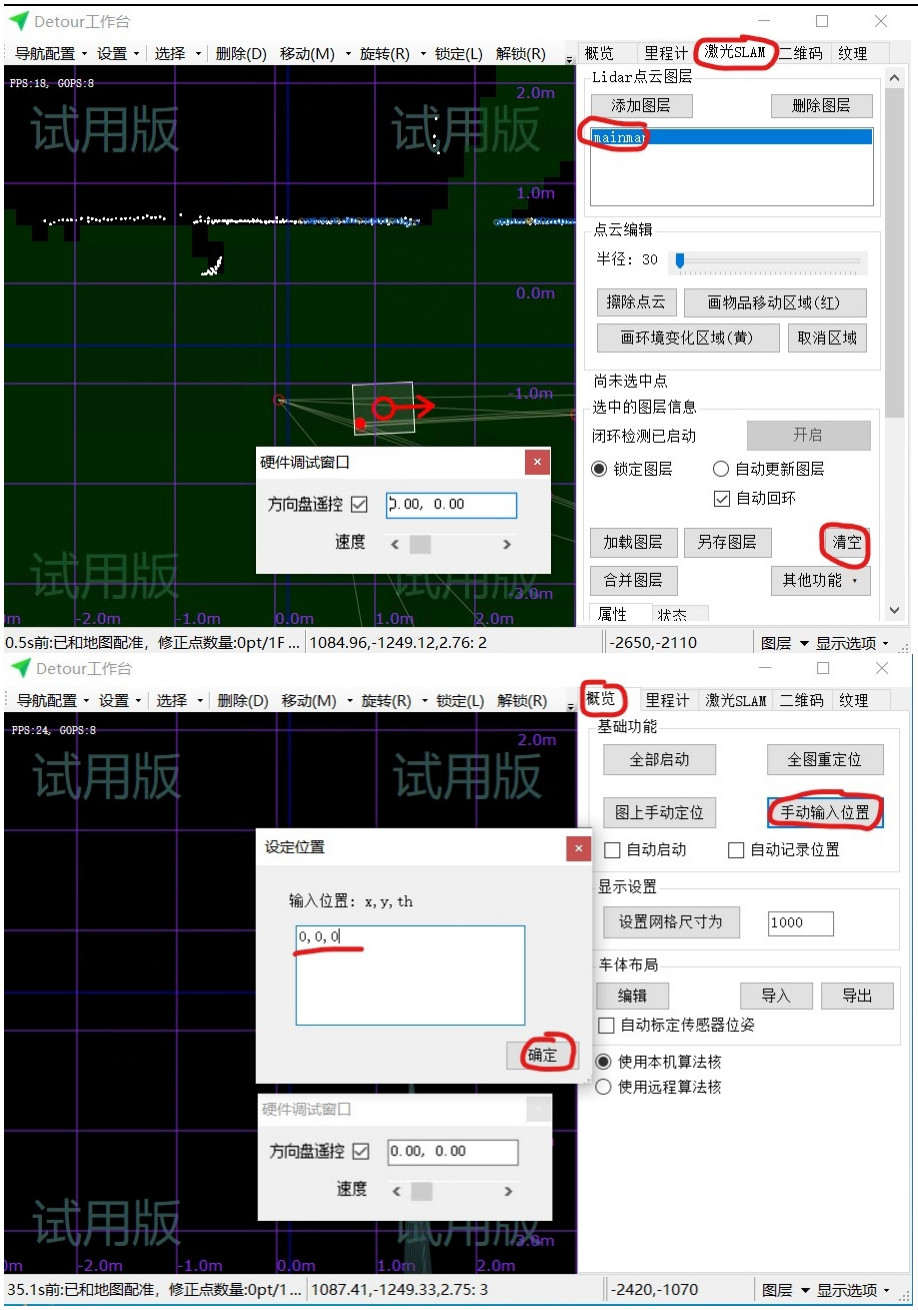
1，先打开 Medulla.exe，然后打开 Detour.exe

2，点击“概览”→“基础功能”→“全部启动”，界面会显示雷达点云



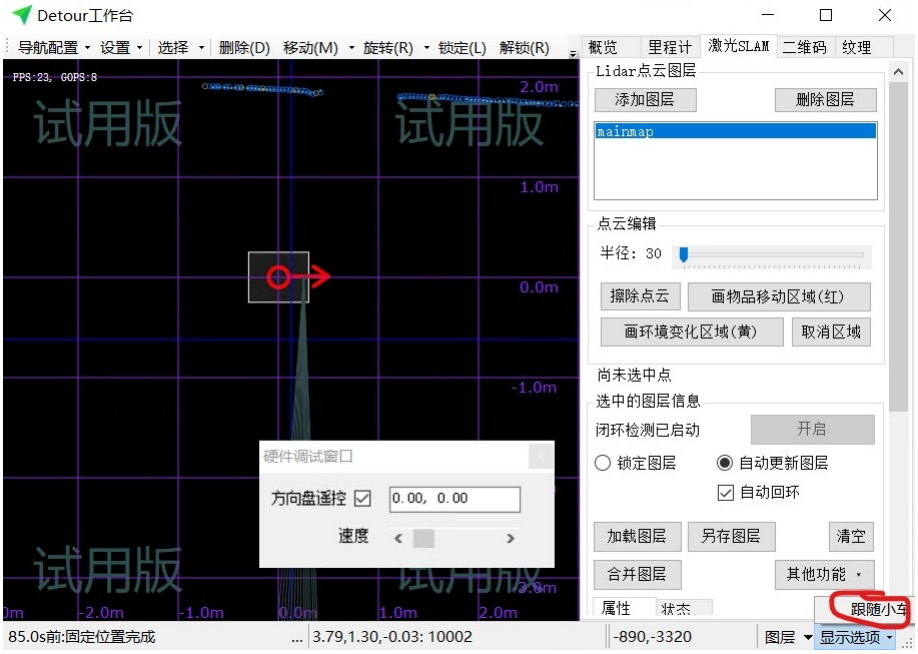
3，首先使用 Medulla.exe 中的 cart→remote 将小车开到一个与墙面平行的位置。如果原来有地图，鼠标框住所有地图元素，点击菜单栏里的“删除”或者在“激光 SLAM”中点击“mainmap”，然后点击“清空”。然后点击“概览”中的“手动输入位置”设定初始位置，输入 0,0,0 确定

MDCS 系统指南

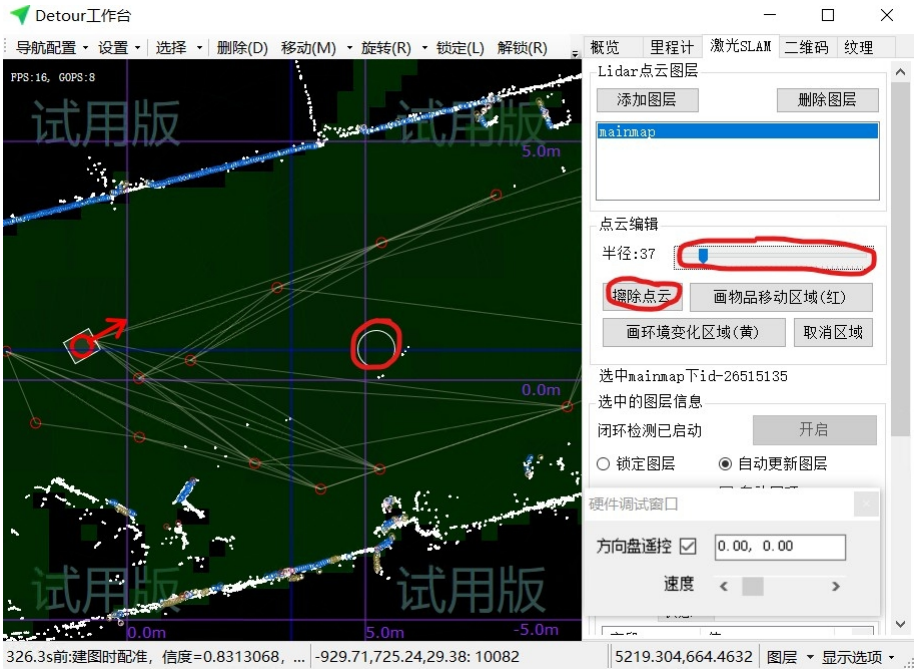

Detour工作台
导航配置 设置 选择 删除(D) 移动(M) 旋转(R) 锁定(L) 解锁(R)
概览 里程计 激光SLAM 二维码 纹理
Lidar点云图层
添加图层
删除图层
mairmar
点云编辑
半径: 30
擦除点云
画物品移动区域(红)
画环境变化区域(黄)
取消区域
尚未选中点
选中的图层信息
闭环检测已启动
开启
锁定图层
自动更新图层
自动回环
加载图层
另存图层
清空
合并图层
其他功能
属性
状态
硬件调试窗口
方向盘遥控
速度
试用版
0.5s前:已和地图配准，修正点数量:0pt/1F ...
1084.96,-1249.12,2.76: 2
-2650,-2110
图层
显示选项
Detour工作台
导航配置 设置 选择 删除(D) 移动(M) 旋转(R) 锁定(L) 解锁(R)
概览 里程计 激光SLAM 二维码 纹理
基础功能
全部启动
全图重定位
图上手动定位
手动输入位置
自动启动
自动记录位置
显示设置
设置网格尺寸为
1000
车体布局
编辑
导入
导出
自动标定传感器位姿
使用本机算法核
使用远程算法核
设定位置
输入位置: x, y, th
0, 0, 0
确定
硬件调试窗口
方向盘遥控
速度
试用版
35.1s前:已和地图配准，修正点数量:0pt/1 ...
1087.41,-1249.33,2.75: 3
-2420,-1070
图层
显示选项

4，选择“激光 SLAM”→“Lidar 点云图层”→点击“mainmap”，在“选中的图层信息”里选择“自动更新图层”，并确保“自动回环”勾选上

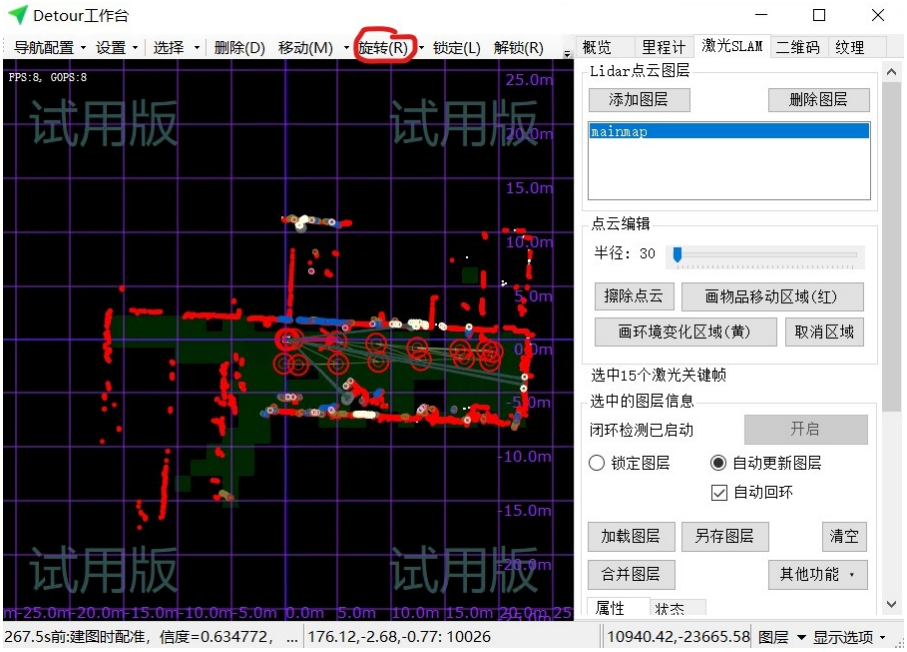
5，根据喜好选择是否点击右下角的“显示选项”→“跟随小车使小车始终在屏幕中央

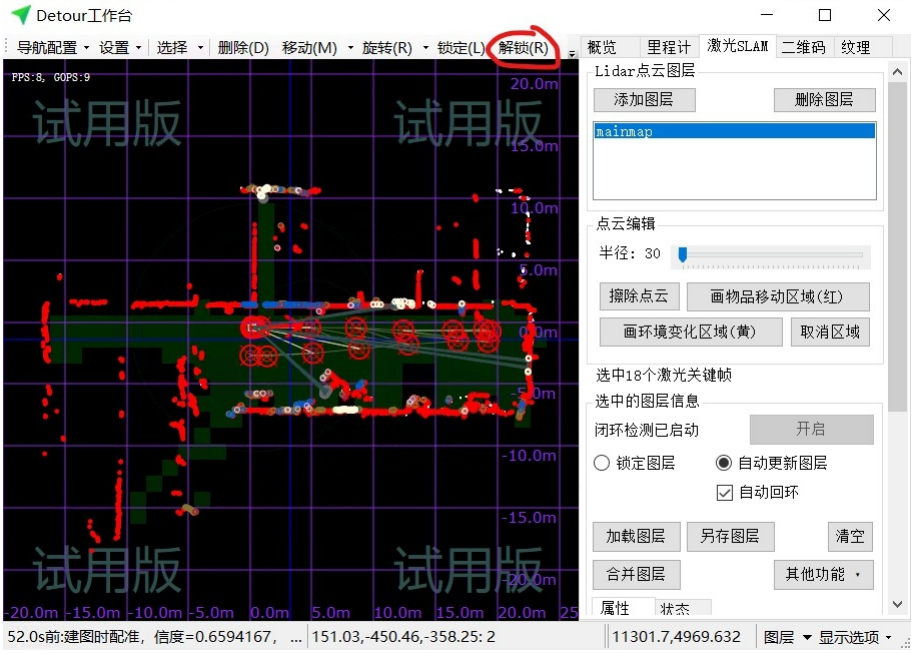


6，打开 Medulla.exe 中的 cart→remote 操控小车缓慢移动进行建图，建图完成后点击激光雷达 SLAM→点云编辑→拖动选择“半径”，点击“擦除点云”去除地图上的杂点，例如在建图 时人经过时产生的点或其他可以擦除的点，如果不小心擦多了，可以将车开过去，打开“自动更新 图层”即可

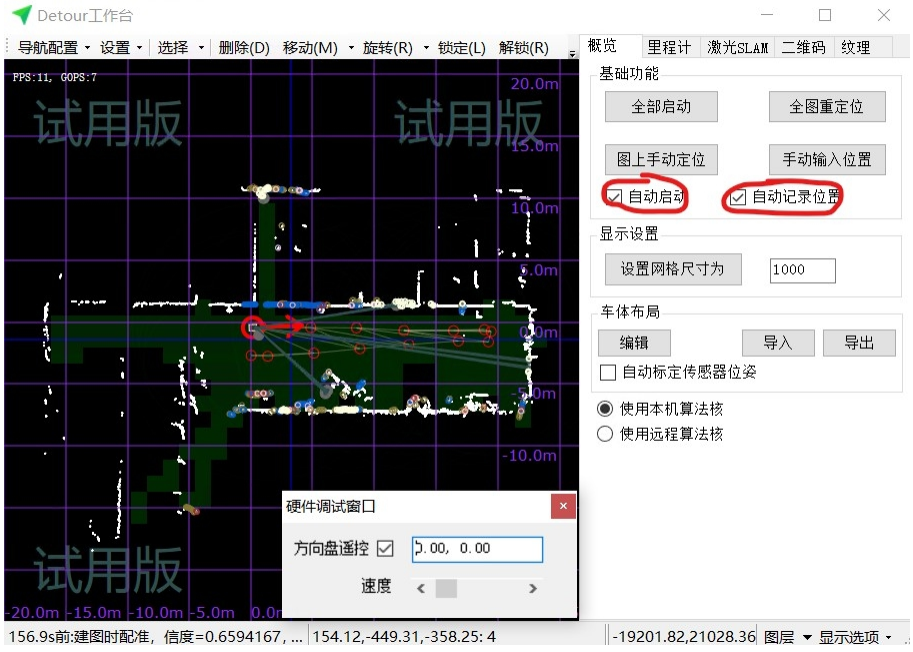


7，如果地图不正，可以鼠标拖动方框选择所有地图元素，然后点击上方的“旋转”，向左向右移动控制 旋转方向，将其摆正，然后再次全选点击“解锁”把关键帧解锁

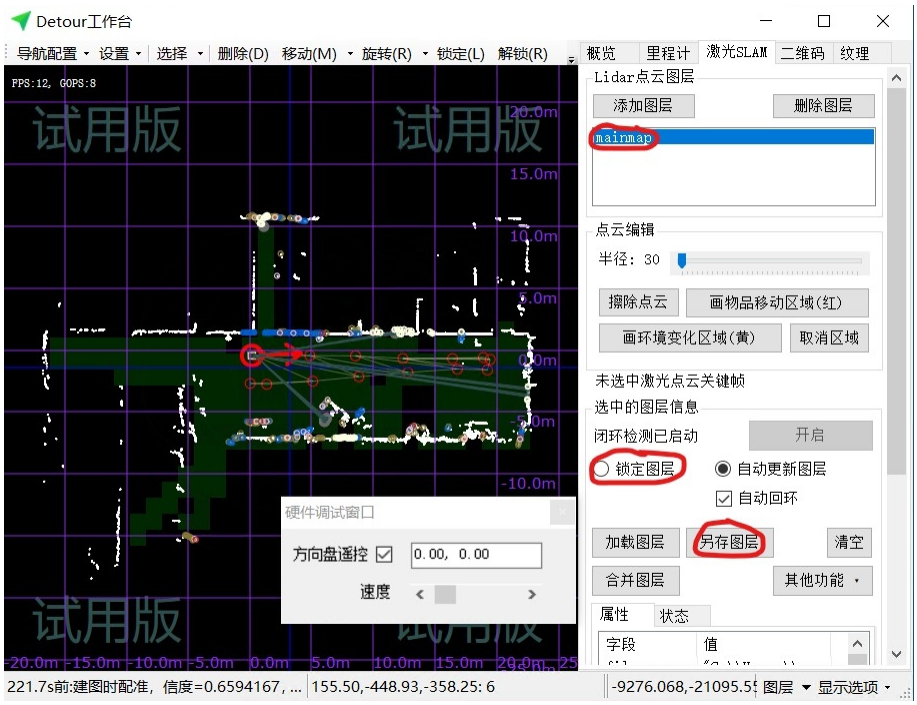


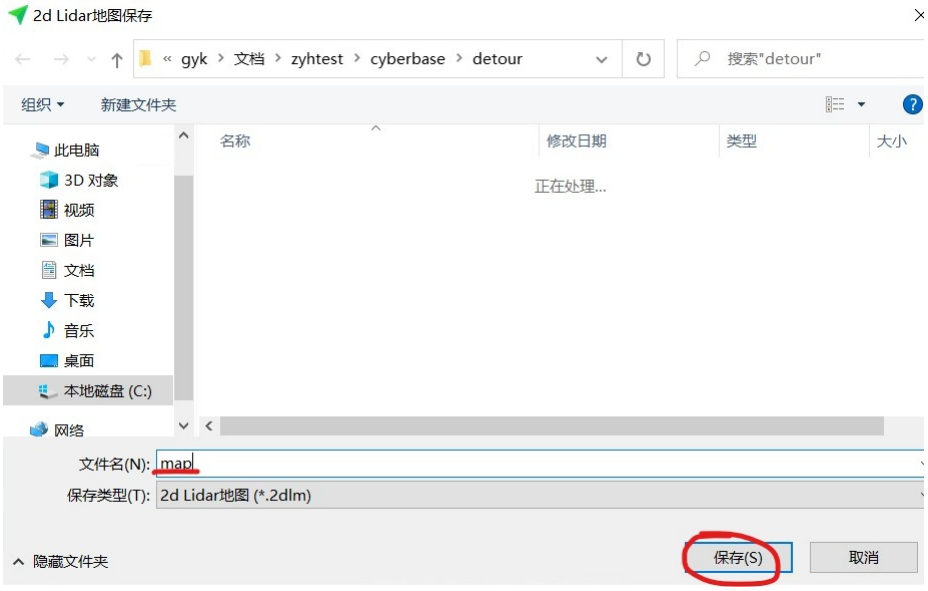


8，勾选“概览”中“自动启动”和“自动记录位置”让程序开启后自动恢复上次关闭程序时的位置并启动定位

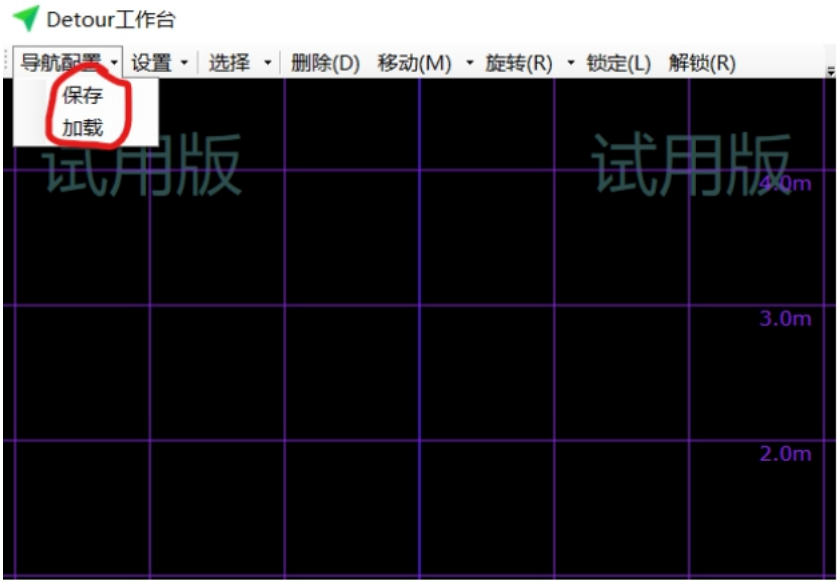


9，建图完全结束后，点击“mainmap”后点击“锁定图层”，然后点击“另存图层”在同目录下存储 2dlm 格式地图。然后点击左上角的“导航配置”→“保存”覆盖掉原来的 detour.json





10，地图图层保存好后，我们想实现自动加载地图，我们可以先加载使用的地图，然后点击左上角导航配置，点击保存，替换 detour.json 即可

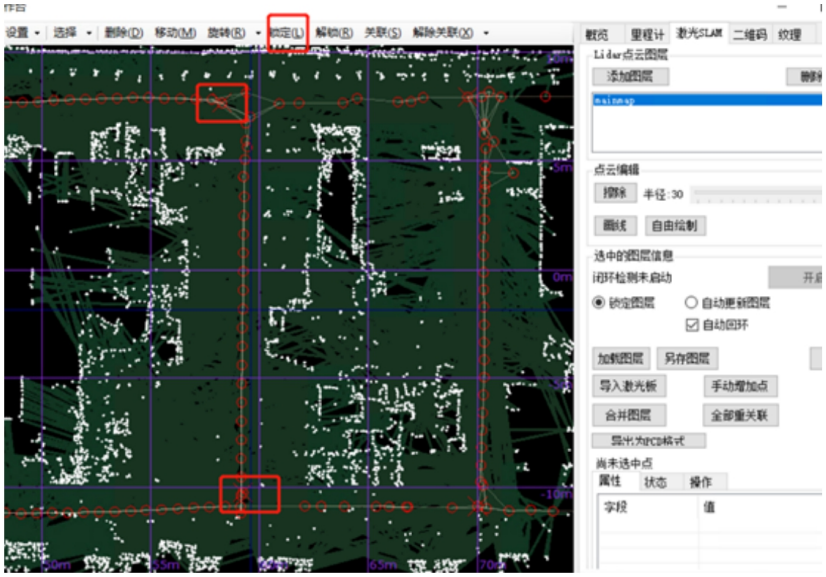


11，如果定位错误，可以“概览”中的“全图重定位”，小车会自动在地图上自定位，在参照物较多且重复少的地方容易自定位，长走廊等位置容易定位出错。如果重定位后还有偏差可以点击“图上手动定位”然后鼠标拖动方向进行手动定位，直到蓝色的线和白色的线重合

5，建图技巧

1. 建图应当监控着 Detour 工作台，若建图有歪斜应及时修正，防止影响到后续的回环匹配等。

2. 尽量减少大回路，应做小回路建图。若大回路不可避免，则最好暂时关闭自动回环功能，防止出现回环错误。走完大回路后，选择回路最后一个关键帧，然后点击移动按钮，将关键帧手动移动到大致正确的位置。然后选择两个可以匹配的关键帧，点击“关联”即可。若发生关联后很快连接断开，则关联后快速点击“解锁”即可。



3. 地图不要出现过多的“上锁点”。旋转、移动后，若不是要锁定位置，请点击“解锁”，使得 Detour 可以自行优化地图。

4. 地图关键帧的锁定，在需要追加地图的时候如果不想追加的部分影响到其他地图，一定要将路口关键帧锁住，如下图，锁住的关键帧会有个 X 号，如果要更新某个区域的地图，建议一定要锁住某些关键的路口，否则整个地图会有一定的修正，导致 simple 路线发生偏移。

6，接口

1，暂停 Detour 地图匹配功能

此功能主要用于，特殊位置由于变化比较大，防止带飞小车定位，我们在小车不运动的时候关闭实时匹配功能。

http://{ip}:4321/switchPosMatch?disabled=true //true 是关闭匹配，false 是打开

2，重定位

此功能主要用于现场加入丢失定位，现场人员不会操作 AGV，可以把小车开到复位点，通过按钮或者触摸屏进行该点重定位。

http://{ip}:4321/setLocation?x=11&y=22&th=33

3，获取定位结果

此功能主要用于上传小车当前位姿，供第三方调度界面显示以及使用

http://{ip}:4321/getPos

常用接口如上，全部接口的如下表 Detour 接口.html

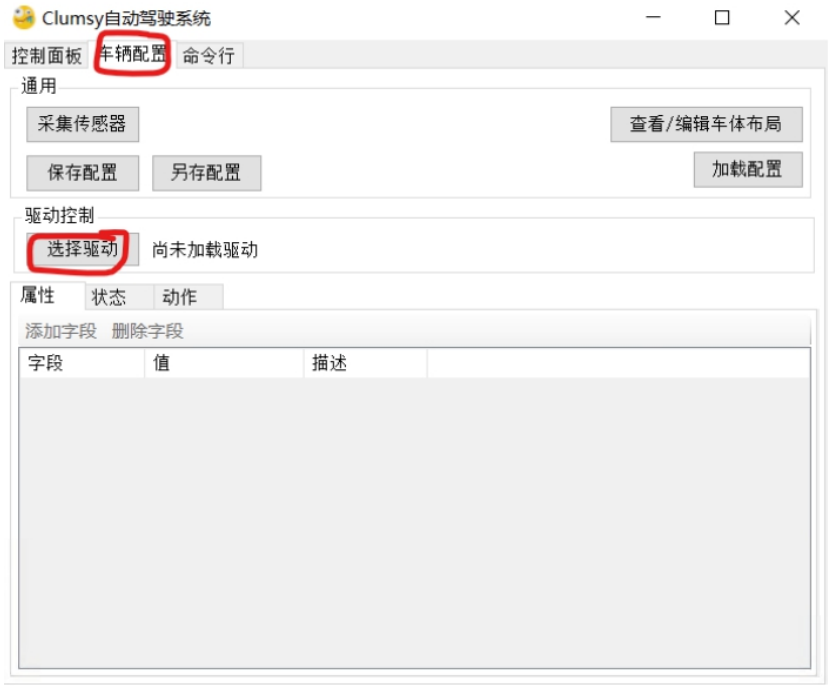
Detour接口 http://{ip}:4321/

接口	参数	说明	返回
/pause		里程计暂停运行	json，格式为{success: true}
/resume		里程计恢复运行	json，格式为{success: true}
/setLidar2DOdometryMask	odometry：string，所设置的里程计名称 字符串：一串 XY 坐标点构成的 json 字符串	设置激光雷达的无效范围，在一系列点构成的闭包范围内的激光点将被忽略，这些点的坐标为车体坐标系下的值	json，格式为 设置成功：{success: true} 设置失败：{error='No odometry named {指定里程计名称}'} 示例：http://127.0.0.1:4321/setLidar2DOdometryMask?odometry=odometry_1&[{"X":123,"Y":456},{"X":123,"Y":456},{"X":123,"Y":456}]
/debug		将程序设置为 debug 模式运行，将输出 debug 日志文件	json，格式为{success: true}
/getSensors		获取激光雷达当前的扫描数据，包括对应设备在世界坐标系下的位姿	byte[]，格式参见示例程序
/getPos		获取当前车体在世界坐标系下的位姿	json，格式为{x: float,y: float,th: float,l_step: int,tick: long}
/getConf		获取程序的配置参数	json，格式参见示例程序
/switchSLAMMode	update：bool，是否更新地图（即建图） name：string，指定地图名称，不指定该参数则对所有图层生效 dynamic：bool，是否图优化，仅在 update 为 false 时生效，地面纹理导航暂不支持该项	切换 SLAM 的运行模式	json，格式为{performed: true}
/switchPosMatch	disabled：bool，是否关闭地图匹配 name：string，指定地图名称，不指定该参数则对所有图层生效	切换是否进行地图匹配	json，格式为{performed: true}
/switchPosMatch	disabled：bool，是否关闭地图匹配 name：string，指定地图名称，不指定该参数则对所有图层生效	切换是否进行地图匹配	json，格式为{performed: true}
/relocalize		进行一次全局匹配	json，格式为{performed: true}
/getStat		获取程序运行状态，包括里程计状态、地图状态等	json，格式参见示例程序
/loadMap	name：string，指定的图层名称，默认为 mainmapfn：string，指定加载的地图文件名称，默认为 mainmap.2dlm	使远端算法核加载远端已保存的某个地图	json，格式为{performed: true}
/saveMap	name：string，指定的图层名称，默认为 mainmap fn：string，指定保存的地图文件名称，默认为 mainmap.2dlm	使远端算法核将某指定图层当前的建图结果保存至文件	json，格式为{performed: true}
/uploadRes	fn：string，上传到远端后文件的命名 bytes：byte[]，待上传数据	以字节数组的方式，向远端算法核上传文件	json，格式为{performed: true}
/downloadRes	fn：string，需下载文件的文件名	以字节数组的方式，将远端算法核的文件下载至本地	byte[]
/setLocation	x：float，x 坐标值 y：float，y 坐标值 th：float，角度值	手动设定车体当前在世界坐标系下的位姿	json，内容为重新定位后的车体位姿，格式与/getPos 相同

三，Clumsy

1，Clumsy 配置

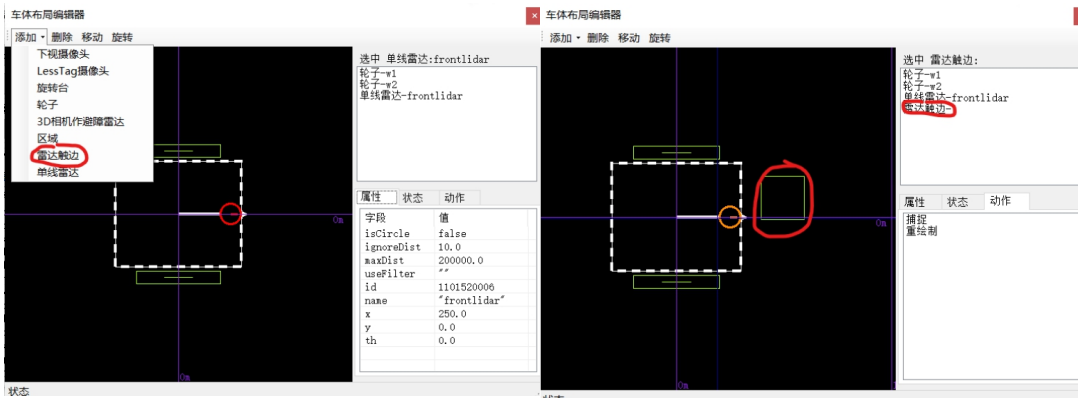
1. 打开 Clumsy.exe 之前要先打开 Medulla.exe、Detour.exe
2. clumsy 程序在开启时会查找当前目录的 clumsy.json 来加载配置，如果没有，则会使用空配置，需要用户进行初始配置
3. 首先 clumsy 文件夹里至少要有 ClumsyConsole.exe，同时复制 Cyberbase_c.dll 到同目录，打开 ClumsyConsole.exe，选择车辆配置→驱动控制→选择驱动，选择 Cyberbase_c.dll，此时驱动加载成功，下方表格显示默认参数，此时点击“保存配置”会生成/覆盖掉 Clumsy.exe 同目录下的 clumsy.json，同时“另存配置”和“加载配置”可用，同时空白的控制面板在重启 Clumsy.exe 后正常显示



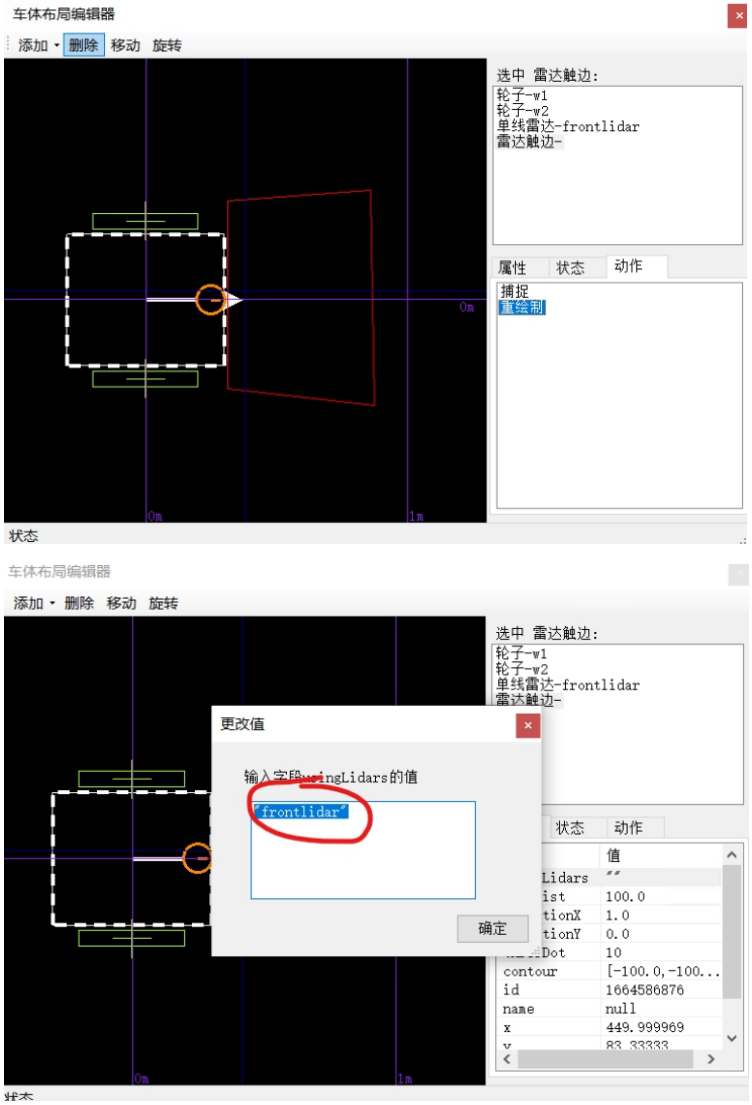
4. 点击“查看/加载车体布局”，使得底盘和雷达的布局和 detour 的车体布局一样，轮子同样参数无效，不修改

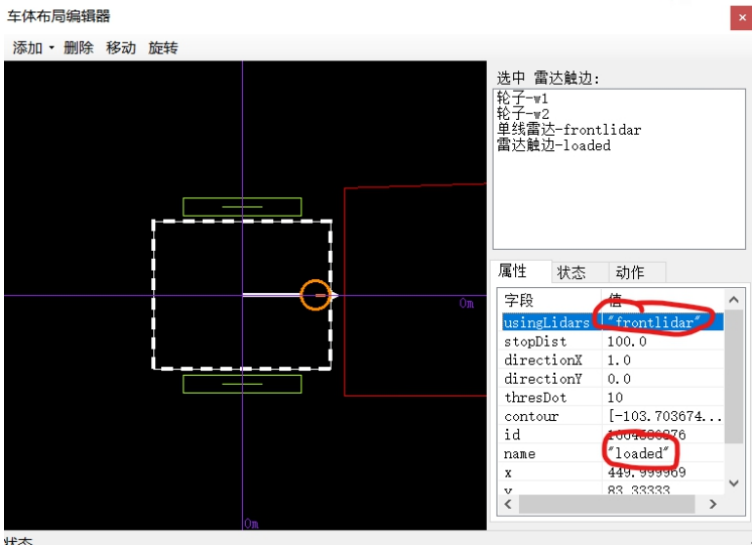
2，雷达触边绘制

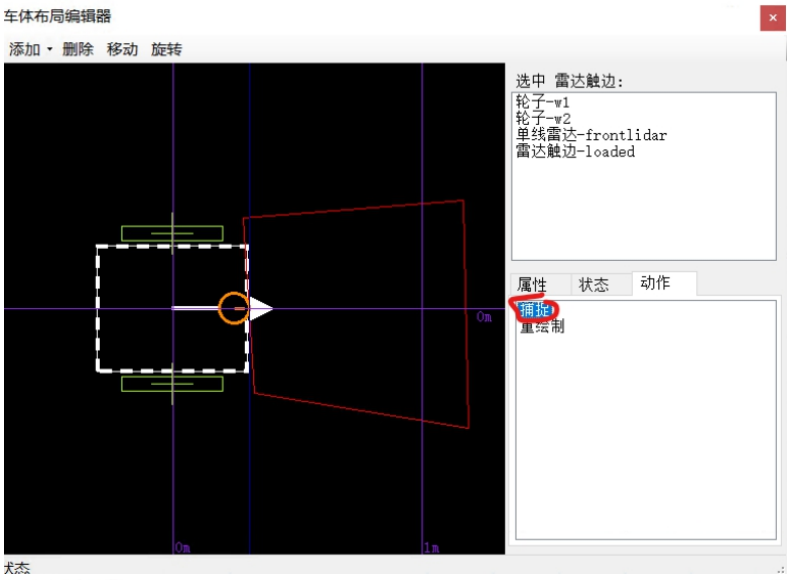
1. 点击“车辆配置中”的“查看/加载车体布局”，点击添加“雷达触边”，然后在雷达前方点击一下出现“雷达触边

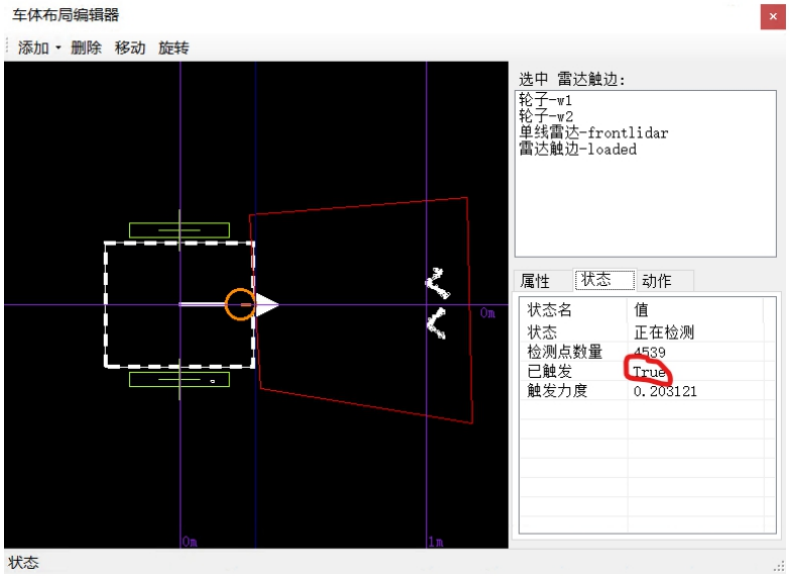


3. 点击“动作”中的“重绘制”，然后依次点击雷达前方想识别区域的顶点即可添加完成，“属性”里的 usingLidars 一定要修改为"frontlidar"，否则会出错，然后 name 里编辑雷达区域名称"loaded"，双击“动作”中的“捕捉”，然后看“状态”中的“已触发”，在框内有障碍物时会显示“True”，完成后“保存配置”





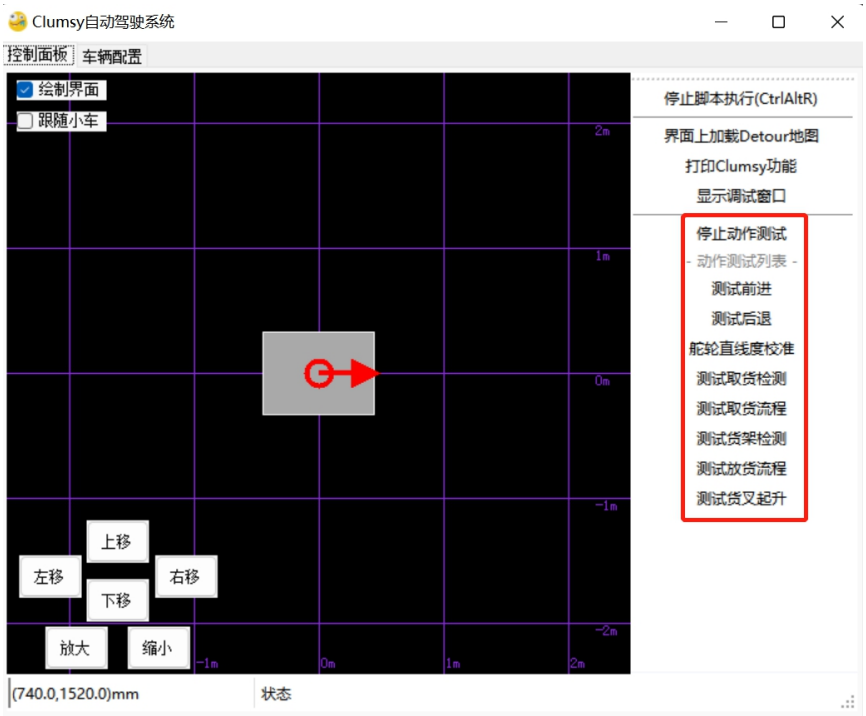




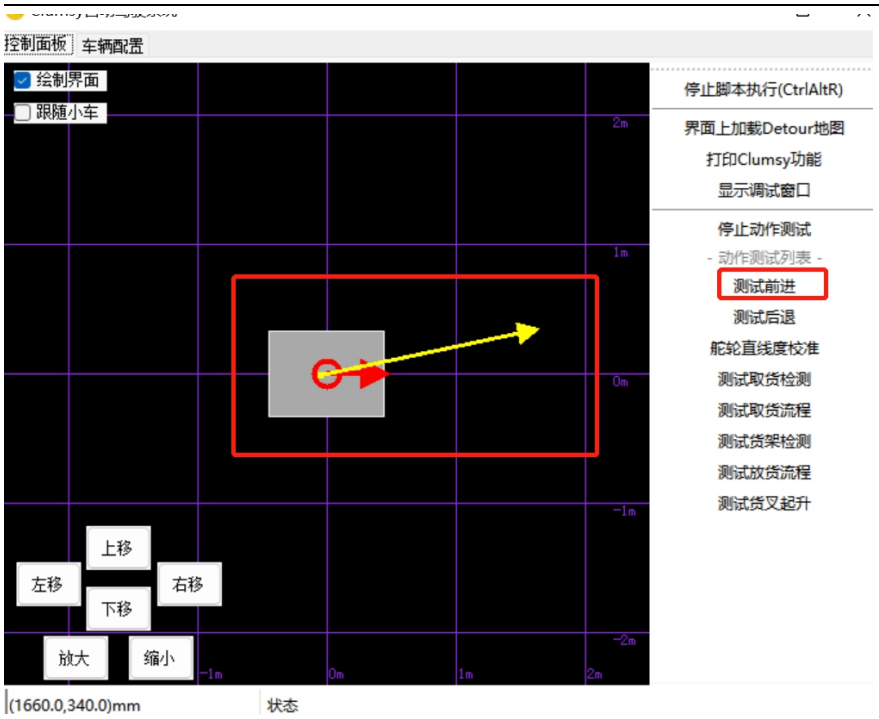
3. 点击 clumsy 的保存配置即可保存成功，clumsy 中替换带有避障检测的 Cyberbase_c.dll 即可使用避障功能，如果避障时车头左右晃可能是 clumsy 参数没调好

3，动作测试

1，我们打开 ClumsyConsole 主界面，可以看到右侧面板上有一系列的动作按钮，这里包括了小车所有执行动作的测试按钮，调试人员前期会通过这个来优化车体运动参数，以及测试动作逻辑



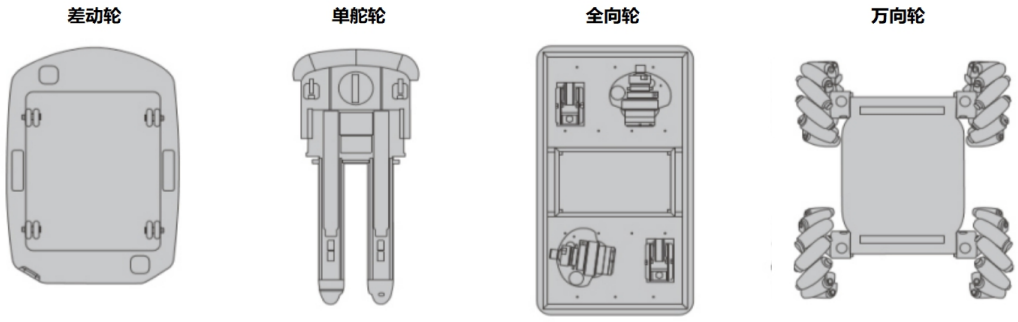
2，比如我们需要测试前进，我们点点击测试前进，然后鼠标从车体中心拖一个矢量方向，小车即会按照我们矢量的方向以及距离进行行驶



4，运动模型

1，运动模型介绍

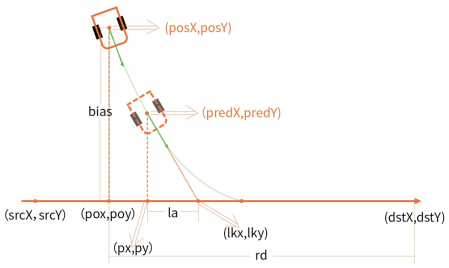
小车运动模型分为：差速轮，单舵轮，全向轮，万向轮 4 个模型



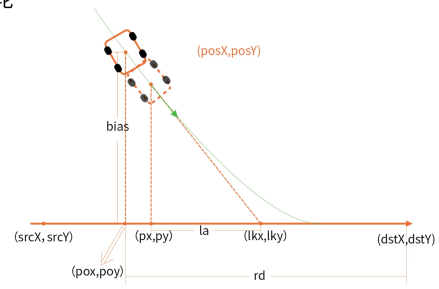
3，运动参数

1，差速车运动参数

差速轮

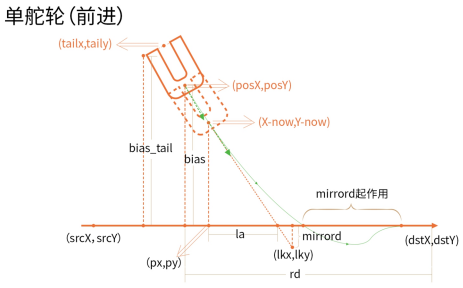


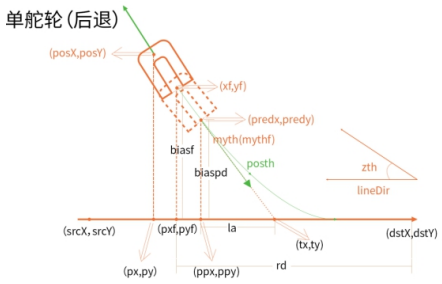
全向轮



轮系	参数	解释	说明
差速轮	thresAcc	加速度限制	调节小车加减速率
	thresWAcc	角加速度限制	
	nffac	角度负反馈修正系数	用于调节小车转向时出现的超调，如果原地转向出现超调，请适当减小 maxWspeed 以及 nffac
	maxWspeed	转弯最大速度	
	lookAhead	车到路径垂足点向前的追踪量	可调节小车贴近路径的快慢，过小容易超调，过大则调整过慢，建议高速长直线适当增大，低速高精度适当调小
	trackFinDist	判断“已到达终点”的阈值	
	wayPointFinDist	连续行走转弯半径	小车停过头，增加停车距离
	wayPointFinSpeedFactor	该值控制“当速度越大时，wayPointFinDist 增大的幅度	
	lineSnappingDist	车体到路径距离，控制 lookAhead 衰减	
	firstThAccuracy	初始旋转的对齐精度	
	stoppingPow	接近终点时速度的衰减幅度	如果小车停不下来，超过目标点，可以适当调大
	stoppingLAFac	接近终点时 lookAhead 的衰减系数	

2，单舵轮车





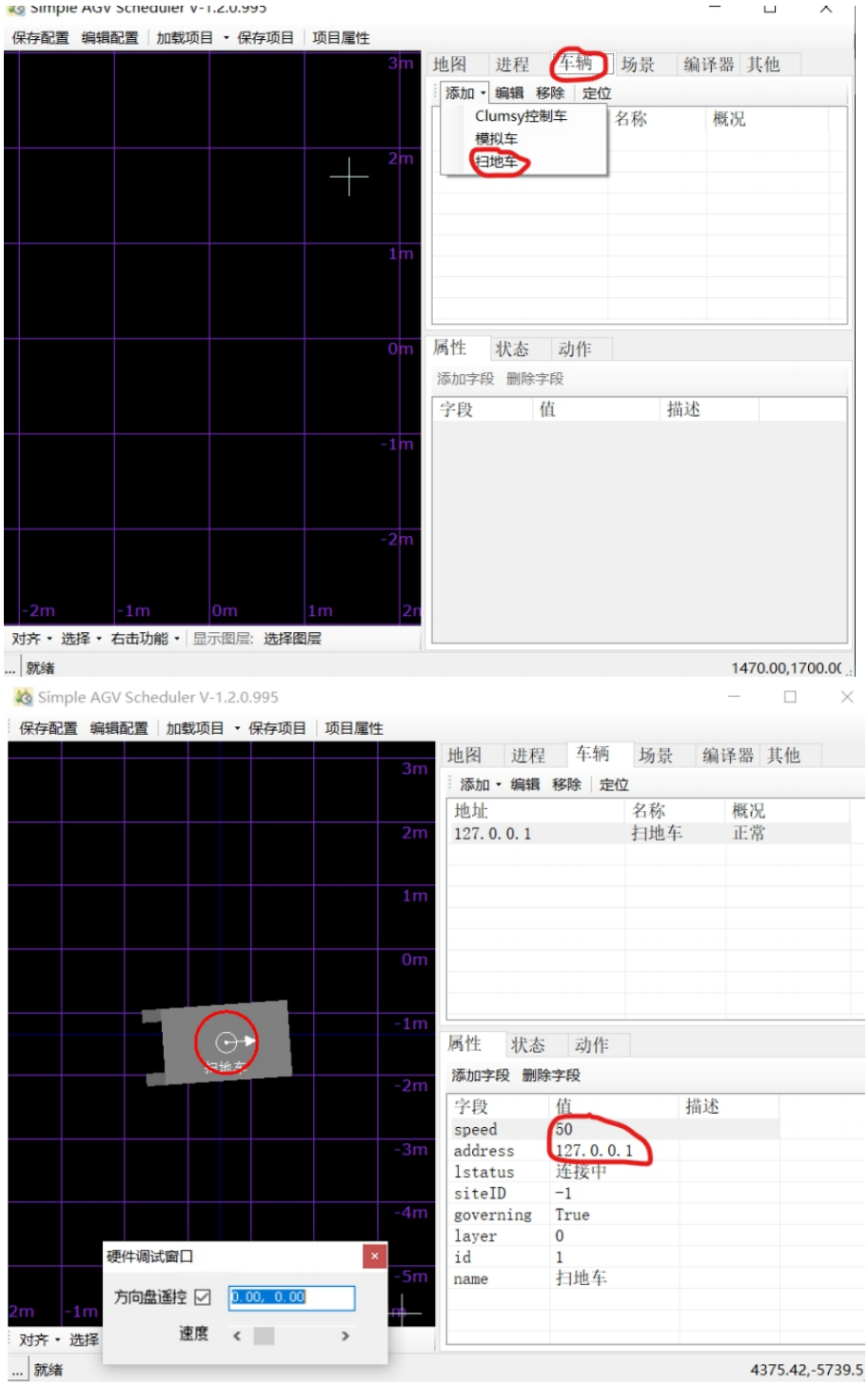
轮系	参数	解释	说明
单舵轮	thresAcc	加速度限制	调节小车加减速率
	maxAngle	最大角度下发	限制舵轮最大转向角度
	lookAhead	车到路径垂足点向前的追踪量	可调节小车贴近路径的快慢
	lookAheadFactor	追踪点前向偏移量与速度大小的线性系数，速度越大，lookAhead 越大	
	lookAheadPow	衰减速度对前向偏移量 la 的影响大小	
	mirrorThres	正向点跟踪镜像距离阈值(快速贴到直线上)	可调节小车贴近直线的快慢
	mirrorScale	正向点跟踪镜像系数(快速贴到直线上)	可调节小车贴近直线的快慢
	mirrorPow	正向点跟踪镜像幂数	可调节小车贴近直线的快慢
	anSigma	正向角度误差速度系数	控制同时转向和速度的，越大，则还没调好转向时就允许运动越小则表示必须完全调好转向再运动
	finDistFactor	正向切换线段距离速度系数	
	refAheadF	预测车体位置的偏移量大小与速度的系数	
	refAheadBaseF	预测车体位置的偏移量基础值	
	dFactorF	跟踪角预测量(PID 中 D)	
	reverseSpeedFac	速度减速比	
	lookAheadR	反向跟踪点跟踪距离	可调节小车贴近路径的快慢

	lookAheadDamperDist	反向跟踪停车前若干距离，用于提高跟踪点距离，从而使得小车不转向	
	forkLength	反向跟踪叉长，要略远于定轮位置	用于调节后退运动姿态
	refAhead	反向跟踪预测点速度系数	
	refAheadBase	反向跟踪预测点距离	
	dFactor	反向跟踪角预测量(PID 中 D)	
	angSigmaR	反向角度误差速度系数	控制同时转向和速度的，越大，则还没调好转向时就允许运动越小则表示必须完全调好转向再运动
	finDistR	反向停止距离	用于调节后退运动姿态
	stopWait	反向停车等待时间	
	tail_l	舵轮中心到定轮距离	用于调节后退运动姿态
	slowdownFac	停车减速幂数	
	dthBias	角度偏差	调节舵轮直线度
	dthScale	角度倍数	
	turnSlowSigma	转弯减速角度范围	
	turnSlowWeight	转弯减速比例	
	dthPow	角度幂数	
	thDiffPowthDiffScale	角度误差下发幂数	
	cAngle	角度误差下发乘数角度	

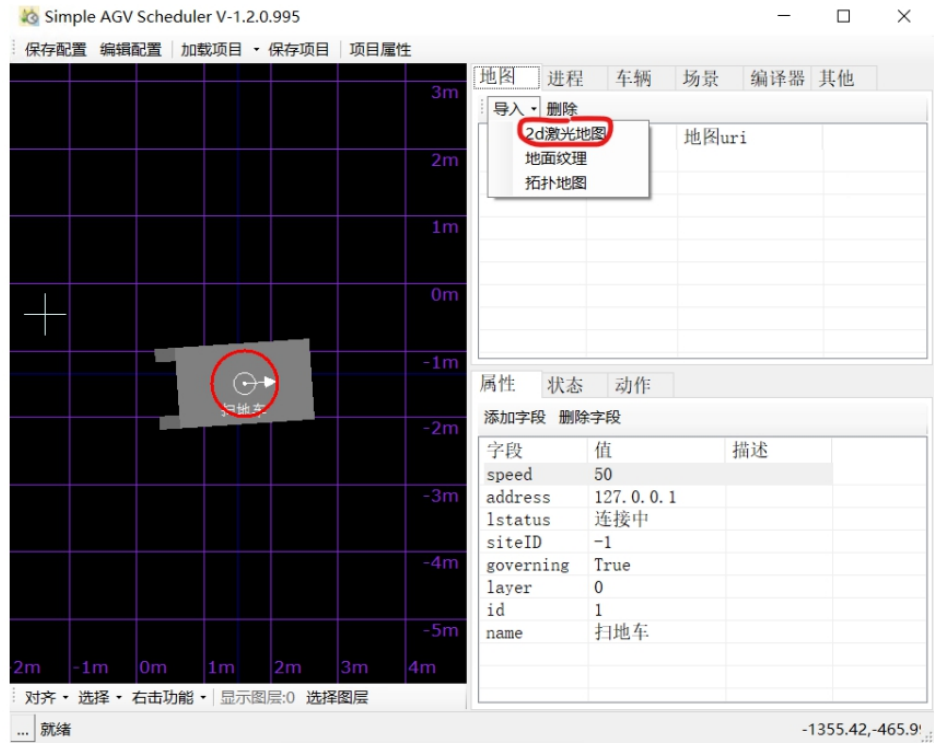
四，Simple

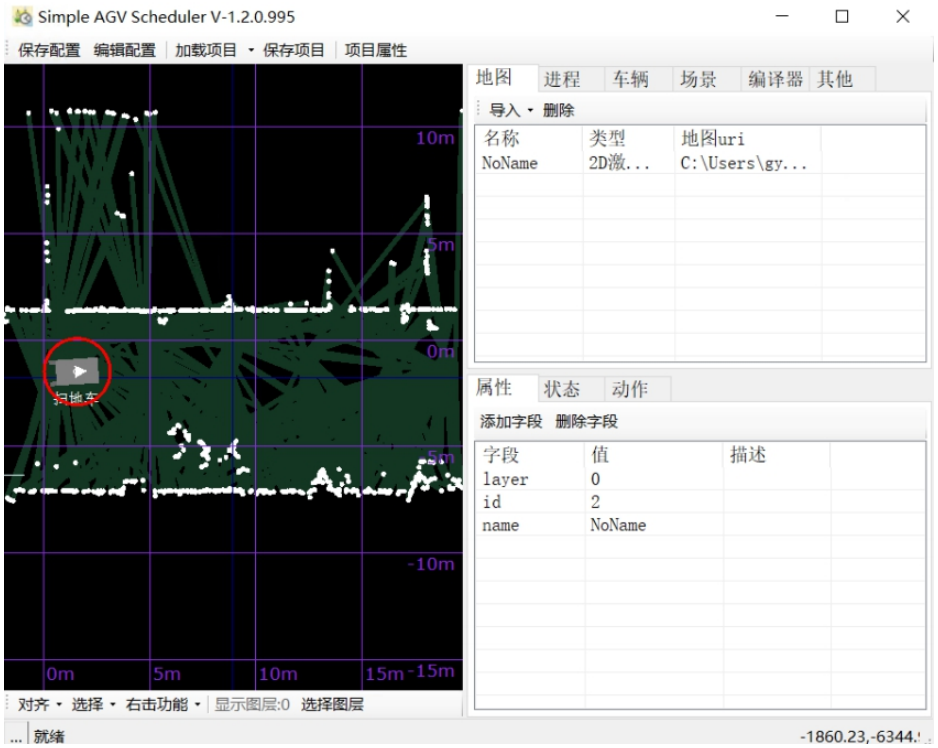
1，Simple 基础操作

1. 依次打开 Medulla.exe、Detour.exe、Clumsy.exe、Simple.exe
2. 点击“车辆”，添加“扫地车”，修改基础速度“speed”的值为 50 左右，必须要有值，否则小车走不了，修改“address”的值为 127.0.0.1，点击确定，小车应该会显示此时的定位，使用 remote 控制小车，在 detour 和 simple 上小车应该会同时移动.



3. 点击“地图”，导入“2d 激光地图”，选择 detour 目录下保存好的 2dlm 格式地图，此时地图应该会在 simple 上显示

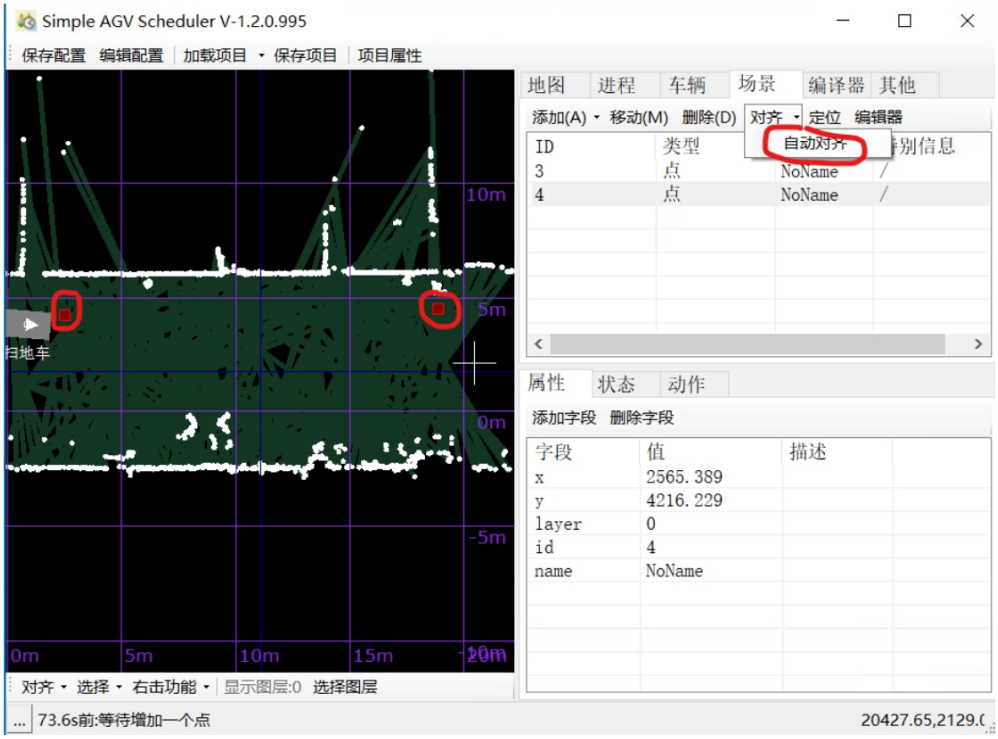


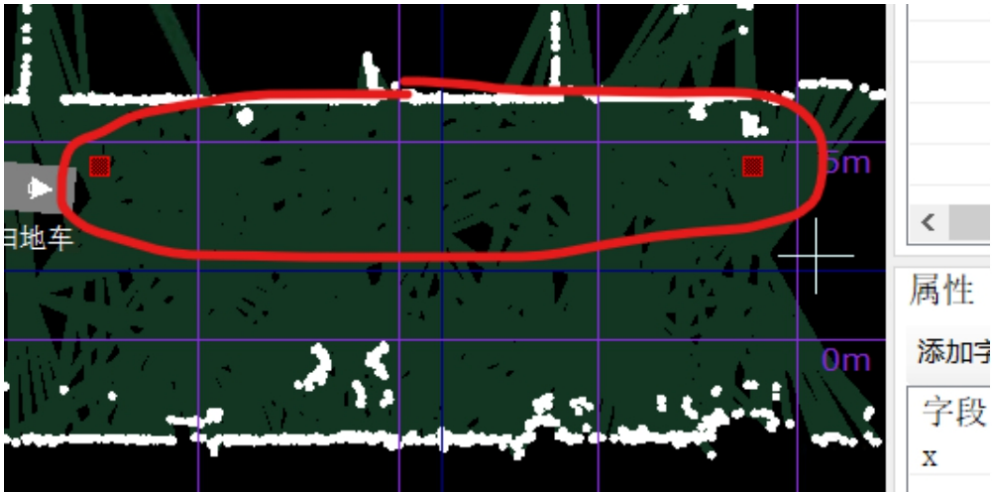


4. 点击“场景”，添加“站点”创建工位，添加“路径”连接工位，也可使用快捷键 ALT+A 添加，然后按下 S 添加站点或 T 添加路径。

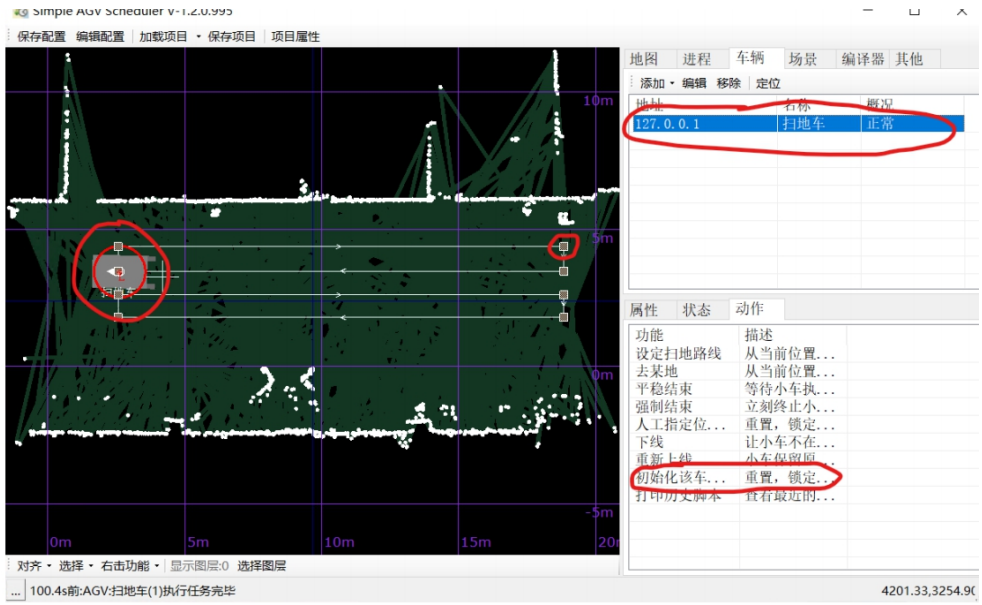
5. 点选择可以看到点的属性，可以根据现场需求增加点的功能，比如牵引上升下降，充电，堆垛等等功能。选择线段可以看到线段属性，可以修改方向或者增加线段速度，点击添加字段，direction:1 表示正走，2 表示倒走，0 表示双向；speed:2000 即可给线段附加 2000r 的速度

6. 若要使两点或多点平行/对齐，可以使用 CTRL 键或鼠标拖选两个或多个点，点击对齐→自动对齐

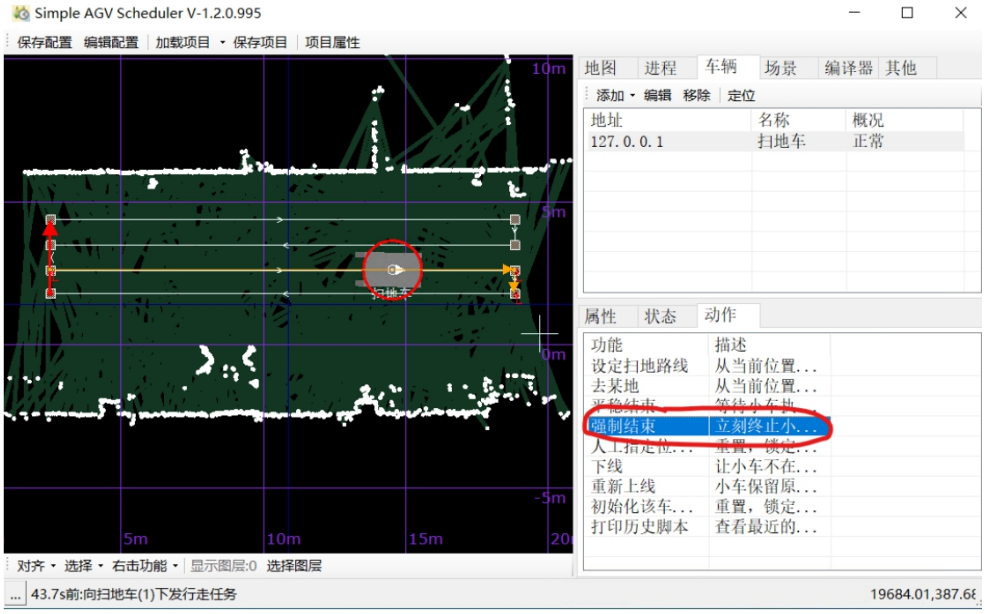




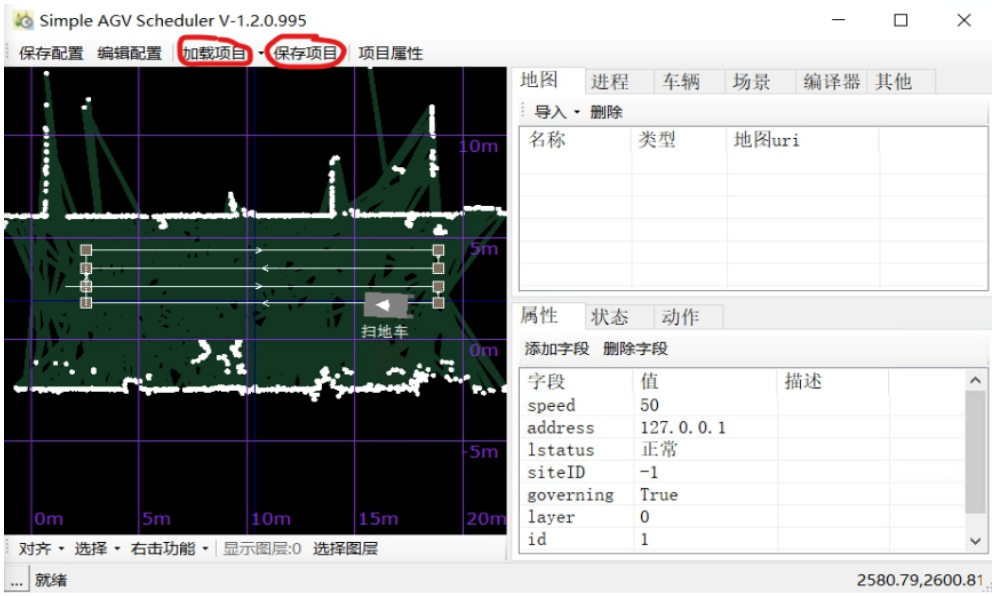
7，路线画好后，在图上选中小车或者在“车辆”中点击小车，然后右击目标站点或者双击“动作”中的“初始化该车调度数据”后右击目标站点，小车即可规划路径行走。

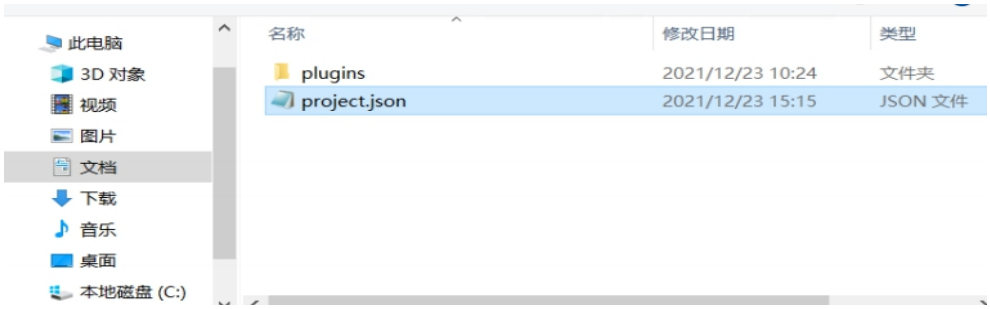


8，若要取消当前任务可以双击“动作”中的“强制结束”



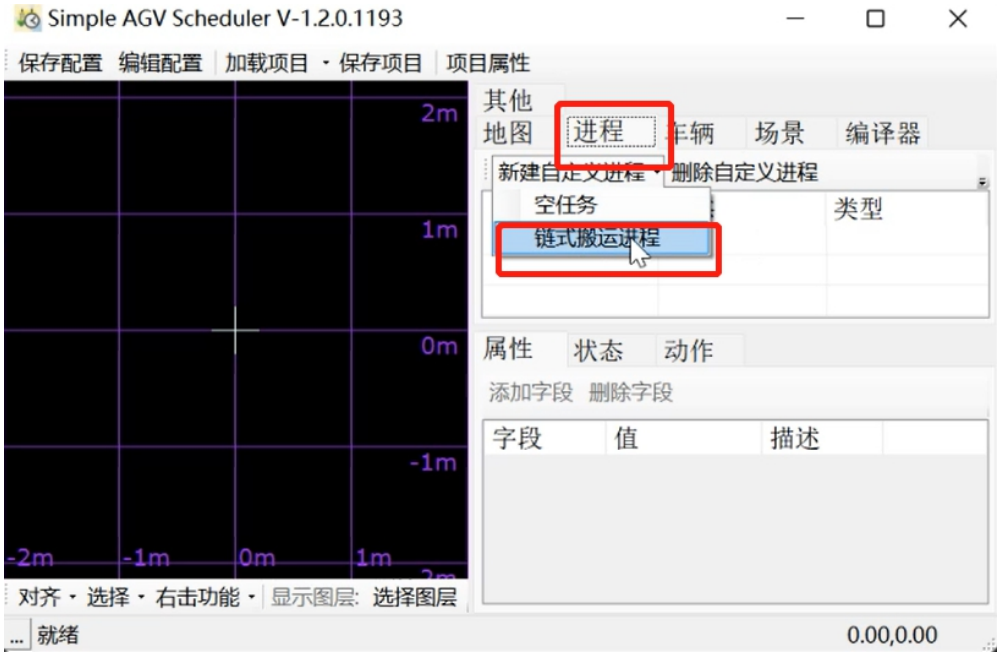
9，路线设置好了之后点击左上角的“保存项目”保存，同样可以“加载项目”，名称可以设置为 project.json



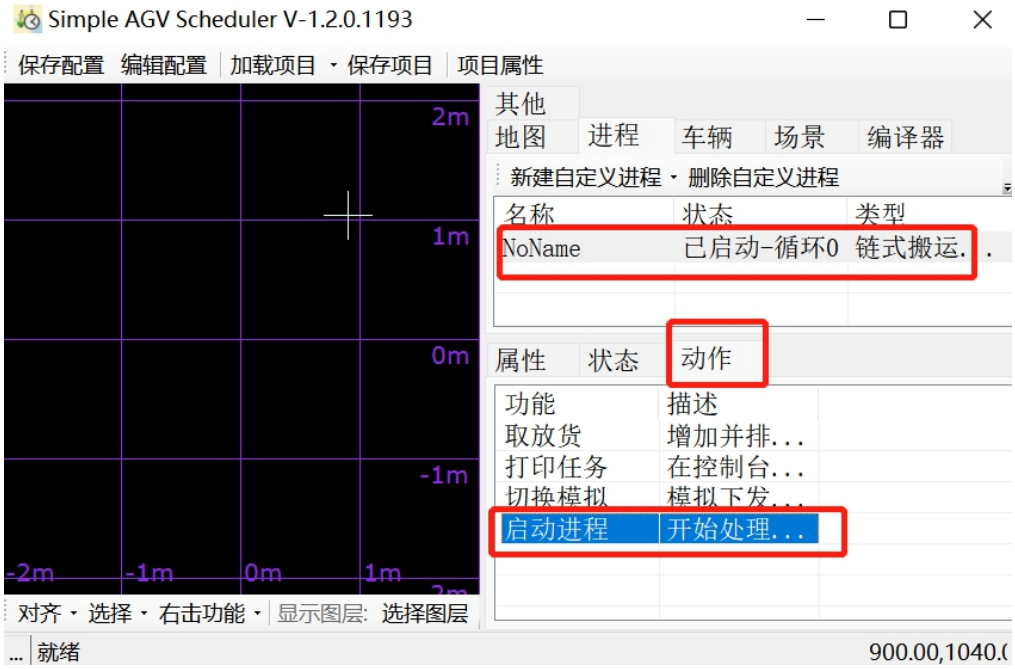


3，开启进程任务

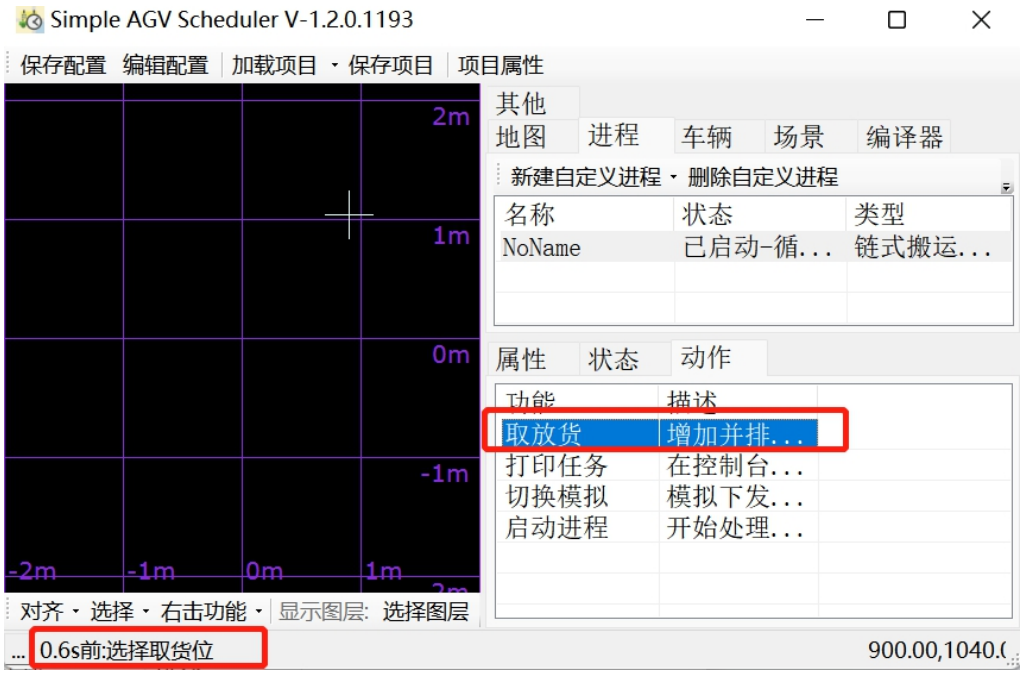
1，进入 simple 界面，在进程一栏中，选择新建自定义进程，选择要执行的进程



2，启动进程，我们选择进程，点击动作，双击启动进程按钮即可开启进程，可在状态中看到已启动即可

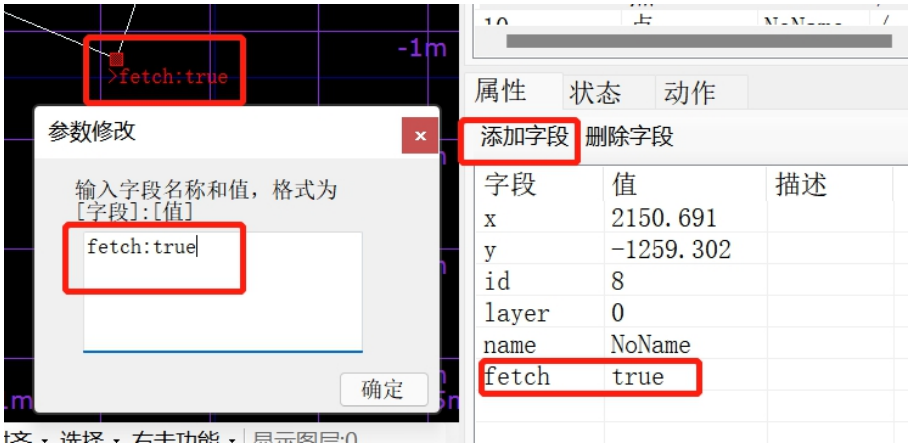


3，进程任务根据现场业务进行编写，链式任务是指调度小车自动进行取料以及放料的任务，我们也可以在 UI 界面上手动调度，选择要取货的点以及放货的点，就会自动分配小车进行任务。



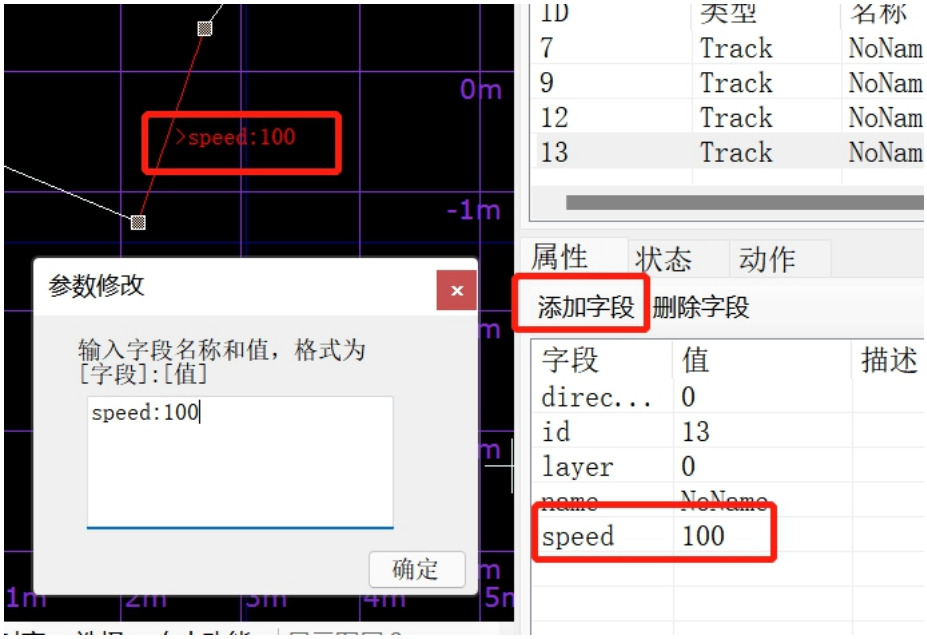
4，设置字段

1，在工位的站点我们需要执行某个功能，比如举升，那么我们可以在工位点上标记 fecth:true，这样下发路线后小车到达这个点会识别到 fetch 标记并且执行举升动作



2，字段是在程序里预设的，根据小车类型以及业务逻辑进行编写，具体可在开发文档中了解。

3，路径字段也类似站点，可以在路径上标记一些功能，比如切换速度，切换避障区域等



二，开发环境

一，IDE 和插件

（vs2019+resharper）

二，项目目录结构

请务必按照以下约定的目录结构进行项目开发！

约定 0：所有车型适配开发都应该基于提供的解决方案工程进行开发。

约定 1：首先应命名车型名称。 解决方案中存在一个称为 CartAdapters 的文件夹，应创建一个以该车型命名的文件夹，并把 ClumsyPilot（自动驾驶工程）和 MedullaAdapter（硬件适配工程）放入文件夹中。

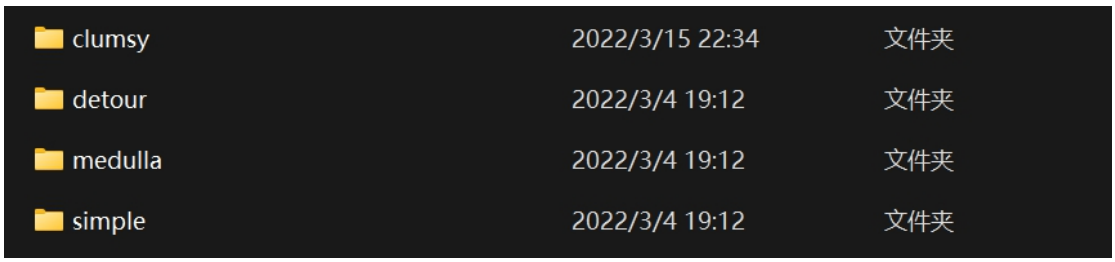
约定 2：源代码目录结构必须符合规范，所有的车型需要放到 CartAdapters 中，每个车型一个目录，并且把所有车型资料放到对应的文件夹中。（如车型图片、对接协议、文档等）。

约定 3：编译结果目录结构必须符合规范 ：编译结果一律放到 build 文件夹中，每个车型一个文件夹。场景均放到 AGVScene 目录中，每个场景（项目）设立一个文件夹。

一，目录结构

1，bin:核心引用文件目录

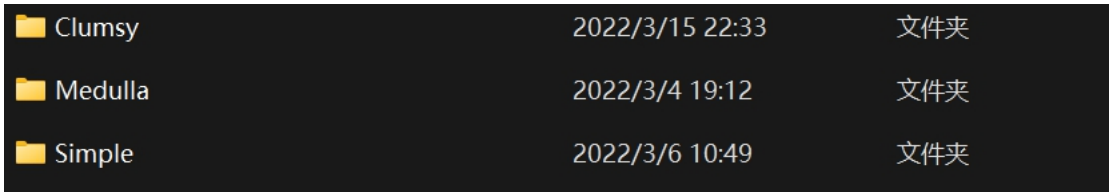
1，该目录包括了 Medulla/Clumsy/Simple 工程引用的核心文件目录



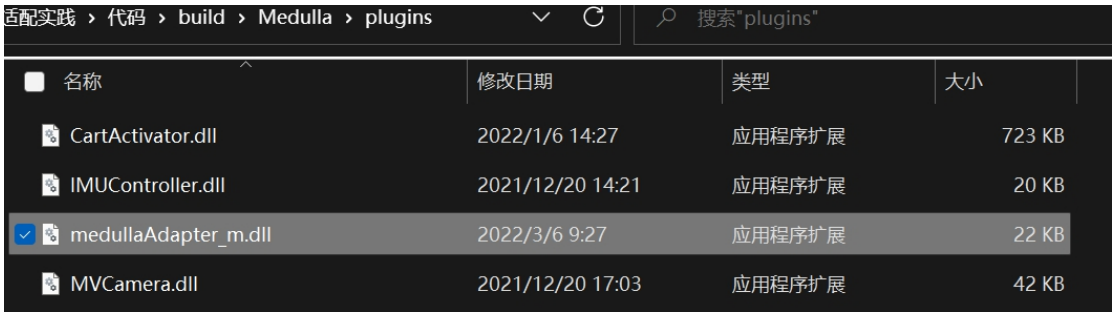
2，clumsy 目录内包含了 ClumsyPilot 工程所需引用的 2 个核心文件

2，build:插件生成目录

1，该目录包括了 Medulla/Clumsy/Simple 工程生成的插件

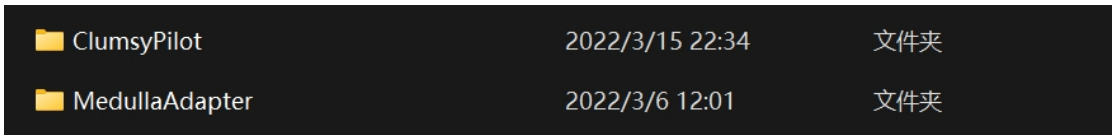


2，如下 medullaAdapter_m.dll 为 MedullaAdapter 工程生成的车型适配插件

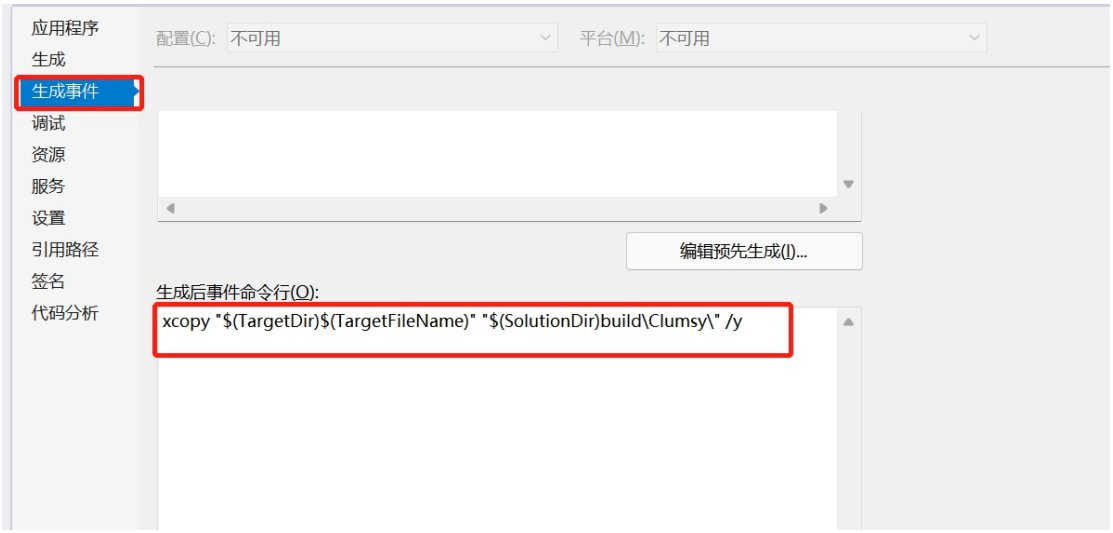


3，CartAdapters:工程适配目录

1，该目录包含 ClumsyPilot 以及 MedullaAdapter 工程文件

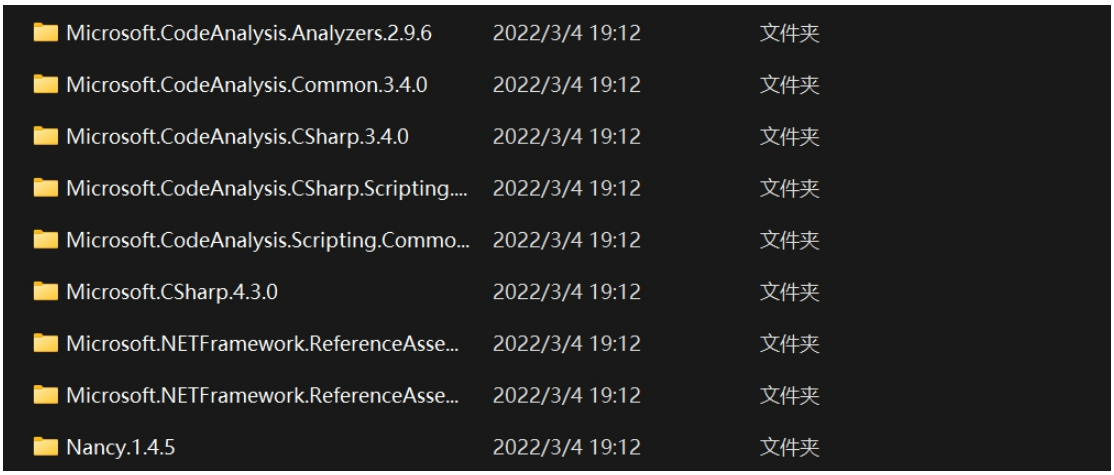


2，在工程内我们需要配置 build event，来将生成的 dll 复制到 build 目录下
Medulla:xcopy "$(TargetDir)$(TargetFileName)" "$(SolutionDir)build\Medulla\plugins" /y
Clumsy:xcopy "$(TargetDir)$(TargetFileName)" "$(SolutionDir)build\Clumsy\" /y
Simpe:xcopy "$(TargetDir)$(TargetFileName)" "$(SolutionDir)build\Simple\plugins\" /y



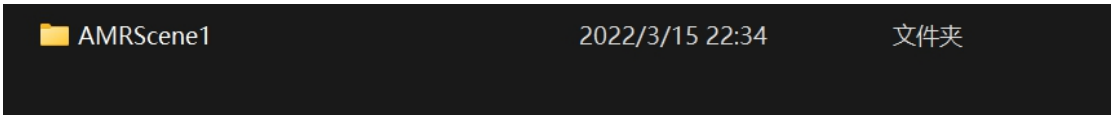
4，packages:包目录

1，该目录包含整个项目所引用的包



5，Scenes:业务场景目录

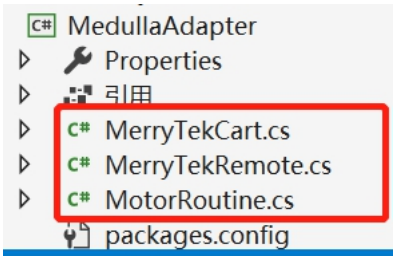
1，该目录包含业务场景项目



三，开发规范

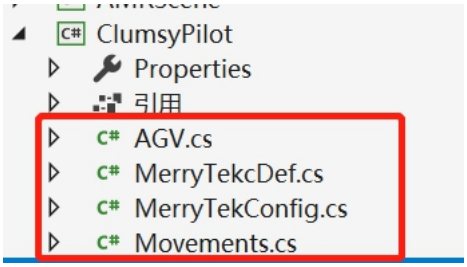
一，MedullaAdapter 规范

1，该工程分为设备定义类为{proName}Cart.cs，手动遥控{proName}Remote.cs 以及还有主要的外设类{DeviceName}Routine.cs，请严格按照规范进行项目命名,例如：



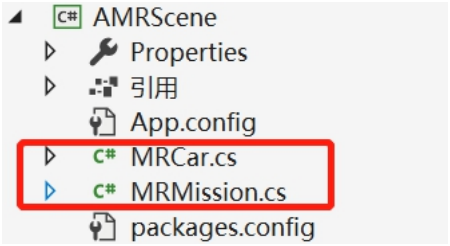
二，ClumsyPilot 规范

1，该工程分为车体模型类{proName}Def.cs，车体配置类{proName}Config.cs,动作类 Movements.cs 以及接类 AGV.cs，例如：



三，Scene 规范

1，该工程主要包含车体定义类{proName}Car.cs,进程任务类{processName}Mission.cs,例如



三，插件程序开发指南

一， MedullaOfficialPlugins

一，雷达适配

1，编码要求

1，雷达驱动派生于 Lidar2DIOObject
2，将点云数据整理为 LidarPoint2D 格式
th：-180 度~180 度，逆时针为角度正方向，x 轴正方向为 0 度
d：距离，单位 mm
intensity：强度，0 到 1
3，提交每帧数据时： 将数据存至 LidarPoint2D[] cachedLidar 更新 scanC（帧号），同时 frame += 1 output()将数据同步至共享内存。

二， CartAdapterProj

一，MedullaAdapter

1，硬件

本适配工程硬件如下：IO 模块(modbus Tcp),电机驱动(can 总线),电池（串口 485）

2，工程的组成

工程的组成主要有：设备定义类为{proName}Cart.cs 还有其他 3 种主要的外设类，采用轮询方式，分别控制 IO、电机、音乐和电池。命名为 IORoutine,MotorRoutine 和 MusicRoutine 和 BatteryRoutine，周期可以根据设备要求进行配置。

```
//梯形图逻辑，scanInterval 为扫描周期
[UseLadderLogic(logic = typeof(MotorRoutine), scanInterval = 20)]
[UseLadderLogic(logic = typeof(IORoutine), scanInterval = 100)]
[UseLadderLogic(logic = typeof(BatteryRoutine), scanInterval = 1000)]
[UseLadderLogic(logic = typeof(MusicLightRoutine), scanInterval = 100)]
```

3，基本概念

AsUpperIO:指 Clumsy 或遥控器等指定的属性，如下发速度，角度，避障区域等
AsLowerIO:指从硬件读取的信号属性，如按钮，灯光，音乐，编码器值等
IOObjectMonitor:用于界面显示供监控的变量

4，进入适配

了解以上概念，开始进入适配：

1,定义基础变量

分为与 Clumsy 交互的变量，IO 硬件控制的输入输出变量，编码器变量，通讯硬件变量，报警变量，需要显示或中转使用的变量，具体可以根据项目协议表进行编写，如下：

```
[AsUpperIO(desc = "下发速度(0-1000)", timeOutReset= true)] public float velocity;
[AsUpperIO(desc = "下发方向盘角度", timeOutReset = true)] public float theta;
[AsLowerIO(desc = "实际方向盘角度")] public float actualTh;
[AsLowerIO(desc = "实际速度")] public float actualV;
[AsLowerIO(desc = "举升控制")] public int lift=0;
[AsLowerIO(desc = "货叉高度控制")] public int liftHeight;
[AsLowerIO(desc = "启动")] public int start;
[AsLowerIO(desc = "停止")] public int stop;
[AsLowerIO(desc = "复位")] public int reset;
[AsLowerIO(desc = "急停")] public int emergencyStop;
        .........
```

2，适配 IO 类

2.1 根据协议，此处使用 Modbus TCP 协议，驱动已经写好ModbusTCP.cs，直接使用即可，首先初始化 ModbusTCP 服务，在 IORoutine 中进行编写,如下图

```
ModbusTCP mtcp = new ModbusTCP();
mtcp.start("192.168.127.1",502);//初始化 ModbusTCP 服务
```

2.2 初始化服务后，开始具体 IO 变量的读取和赋值。

```
var read = mtcp.registerBuffer(1, 0x0000, 1);//站号 起始地址 读取长度
var readData = read[0];
cart.start= readData >> 0 == 1 ? 1 : 0;
cart.stop= readData >> 1 == 1 ? 1 : 0;
........
```

2.3 将所有 IO 适配后进行测试看读取和写入是否正确，然后进入下一步

3，适配电机驱动--重要

3.1 根据 CAN 协议，确认电机控制方法，此项目协议如下，在 MotorRoutine 中进行编写。

A	B	C	D	E	F	G	H	I	J	K	L
接口	设备	ID	BYTE0	BYTE1	BYTE2	BYTE3	BYTE4	BYTE5	BYTE6	BYTE7	方向
CAN1	原车控制器	0x1A6	0x3:前进，0x5后退	0x00	速度，可输入范围0~167，表示速度0~3352rpm	角度，可输入范围-900~900，对应-90°~90°，900左转（车头方向左边），-900右转（车头方向右边）		加速时间，可输入范围1~250，对应	减速时间，可输入范围1~250，对应0.1~25s	升降速度，可输入0~255，表示0%~100%	发送报文
		0x226	电量，0-100，对应0%~100%	错误代码	实际转速，单位rpm		角度，范围-900~900，对应-90°~90°，900左转（车头方向左边），-900右转（车头方向右边）		空	空	接收报文
CAN2	语音播放器	0x200	0x01	0x51	音乐文件夹编号，可输入范围0x00~0xFF，对应AW001~AW225	音量，可输入范围0x00~0x1c，对应0~28级	0x00	0x00	前6字节异或值	0x02	发送报文
	电池	0x02F4	总电压，16进制，单位V		总电流，16进制，单位A		剩余容量，单位%	空	放电计时，单位h		接收报文
	拉绳编码器/车头转向编码器	0x00	0x01	0x00	空	空	空	空	空	空	发送报文（启动编码器，发一次即可同时启动两个编码器，启动后即可接收位置报文）
	拉绳编码器	0x182	DATA（接收值转为10进制，再乘0.1，即为实际行程量，单位mm）				空	空	空	空	接收报文
	车头转向编码器	0x183	DATA（角度=（MOD DATA/单圈分辨率）*360°），单圈分辨率14bit				空	空	空	空	接收报文
	充电桩	0x110	0x01	0x01	0x01	0x00	0x00	0x00	0x00	0x00	发送报文（启动充电0.5s周期发送，停止充电暂停发送）
		0x272	0x01	0x24	0x03	0XE8	0x00	0x00	0x00	0x00	接收报文（启动）
		0x272	0x01	0x24	0x03	0XE8	0x00	0x01	0x00	0x00	接收报文（停止）

3.2 根据协议，将速度以及角度通过报文下发给驱动器

```
public ZLGCAN can = new ZLGCAN();
public override void Operation()
 {
     var sendV = cart.velocity;
     var sendTh = cart.theta;

     can.sendMessage1(0x1a6, new byte[8]
             {
                 (byte) (sendV > 0 ? 0x3 : 0x5), 0,
                 (byte) sendV,
                 (byte) (((int) sendTh * 10) & 0xff), (byte) (((int) -sendTh * 10) >> 8),
                 10, 10, 255
             });

             if (can.canRecv.TryGetValue(0x226, out var tup1))
             {
                 cart.actualV = BitConverter.ToInt16(tup1.Item1, 2);
                 cart.actualTh = -BitConverter.ToInt16(tup1.Item1, 4) * 0.01f;
             }
   }
```

3.3 这一步完成了，我们就可以初步进行小车行走测试，通过 medulla 遥控器，在 SteerWheelCart 里面配置

```
//设置 MedullaAPI，使得 clumsy 等系统可以直接设置 IO 变量
[UseMedullaAPI(name = "move", UpperIOs = new[] {nameof(velocity), nameof(theta) })]
```

这时候我们要进行关键的一步，通过遥控器，我们需要保证，下发-90° 驱动器实际反馈也是-90,并且实际舵轮转向也是左转 90°,下发线速度，反馈的速度也应该和下发一致。

到这里 Medulla 的主要工作已经完成了，我们继续下面的其他硬件适配!

4，适配电池

4.1 此处使用了 485 通讯，因此我们要实例化 SerialPort，并且调用写的初始化串口方法，传入需要开启的端口以及端口波特率

```
public static SerialPort serialPort;
initSerialPort("COM1", 9600);//初始化 COM1 端口，波特率 9600
```

4.2，端口打开后，就根据具体的电池协议进行报文的发送和读取了-协议见附件[示范协议\BMS 通信协议.pdf]

3 读取指令

使用下面指令便可读取上述数据

Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Byte8
0x01	0x03	0x00	0x00	0x00	0x23	0x04	0x13

指令数据说明：
Byte1：设备地址，默认 0x01
Byte2：功能码，0x03 读取保持寄存器内容
Byte3-Byte4：数据起始寄存器地址，Byte3 起始寄存器高位，Byte4 起始寄存器低位
Byte5-Byte6：数据长度，Byte5 数据长度高位，Byte6 数据长度低位
Byte7-Byte8：CRC16 校验，Byte7 CRC16 校验高位，Byte8 CRC16 校验低位

例如：
发送
01 03 00 00 00 23 04 13
接收
01 03 46 11 A1 F5 F8 00 07 00 00 0C F9 0A 49 00 01 00 00 CA D0 CC B6 24 13 00 00 00 03 00 00 80 00 80 00 10 20 2C 08 20 48 0C E6 0C F9 0C DE 0C AF 0C 77 0C 3F 0C 00 0B C8 0B 8E 0B 53 0B 16 0A D8 0A A8 0A 79 0A 49 00 00 DB B0
0x01 设备地址
0x03 功能码
0x46 数据长度
……
DB B0 校验
0x11A1 对应电压 4513*10mV=45.13V

4.3 如下图 根据协议，获取电池电压和电量

```
byte[] ToReadBytes = new byte[8] { 0x01, 0x03, 0x00, 0x00, 0x00, 0x23, 0x04, 0x13 };
if (serialPort.IsOpen)
{
  serialPort.BaseStream.Write(ToReadBytes, 0, ToReadBytes.Length);//写数据
  byte[] respLine = stringToHex(serialPort.ReadLine());
  if (respLine.Length < 20)
     throw new Exception($"wrong ER response: {respLine}");
  cart.voltage = (respLine[1] & 0xFF) << 8 | respLine[0];//电压
  cart.soc = respLine[13];//电量
}
```

至此，电池就适配好了，继续下一步，声光以及按钮的逻辑编写

5，声光按钮

5.1 将声光报警分成这几个，然后针对这些模式，赋予不同的灯光以及音乐即可

```
var mode = 0;
if (cart.velocity != 0)
```

```
    mode = 1;//运动状态模式 1
if (cart.soc < 25)
    mode = 2;//低电压模式 2
if (cart.obstacleStop1==1 )
    mode = 3;//近避障模式 3
if (cart.IsFault() || cart.com_can1 != 0 || cart.com_can2 != 0)
    mode = 4;//报警模式 4
if (cart.lightFlash)
    mode = 5;//声光报警模式 5
```

5.2 根据小车运动状态，将按钮逻辑写进程序里，保证按下启动按钮后，小车可以进行移动，复位按钮可以复位报警或状态，停止或急停按钮可以停止小车，并且将报警状态显示在监控画面，具体可参考代码。

二，ClumsyPilot

1，车型定义

1，定义车型

1.1 首先在{proName}Def.cs 文件中，因为底盘是单舵轮，所以定义 SwSteering 类，并且继承 SteeringWriterClass，然后重写 WriteSteering 方法获取算法速度

```
public override void WriteSteering(float angle, float velocity)//重写 WriteSteering 获取算法速度
{
    SWDef.self.velocity = velocity;
    SWDef.self.th = angle;
}
```

1.2 车型定义文件还需要重写 Init 方法，这个方法会在 Clumsy 启动时调用。

```
public override void Init()
{
    self = this;设置 self=this，//使得其它类可以访问 SWDef 的实例
    ClumsyLib.Capture();//调用 ClumsyLib.Capture();从而启动 Clumsy 的传感器采集功能

}
```

1.3 此外还需要重写 DriveStop 方法。

```
 public override void DriveStop()
{
    velocity = 0;
    th = 0;
}
```

2，定义与 Medulla 交互的变量

2.1 MedullaAdapter 工程中设备定义类 SWDefCart.cs 中摘抄标记了 AsUpperIO 和 AsLowerIO 的字段。这些字段中，AsLowerIO 字段会自动更新，AsUpperIO 字段会自动下发,因此需要交互的变量复制过来即可。

```
public static SWDef self;
public static MedullaInterface mInterface = new MedullaInterface();
[AsUpperIO(desc = "下发速度(0-1000)", timeOutReset = true)] public float velocity;
[AsUpperIO(desc = "下发方向盘角度", timeOutReset = true)] public float th;
[AsLowerIO(desc = "实际方向盘角度")] public float actualTh;
[AsLowerIO(desc = "实际速度")] public float actualV;
[AsUpperIO(desc = "货叉移动方向")] public int lift;

[AsLowerIO(desc = "电池充电接触器")] public int chargeContactor;
[AsLowerIO(desc = "货叉位置")] public int liftPos;
[AsLowerIO(desc = "货叉位置")] public bool getItem;
[AsLowerIO(desc = "举升目标位置")] public int liftTarget;
[AsLowerIO(desc = "障碍物减速区，为 0 时减速")] public int obstacleRetard;
[AsLowerIO(desc = "障碍物停止区 1，为 0 时停止")] public int obstacleStop1;
[AsLowerIO(desc = "障碍物停止区 2，为 0 时停止")] public int obstacleStop2;
```

这部分完成后，就开始进行自动功能的测试

2，定义动作

1，定义一个货叉举升下降的动作

动作定义在 Movement.cs 文件里面，里面已有一些常用的动作，先从简单的开始：

1.1，货叉举升的动作使用 MovementDefinition 来定义。该类的 Get 方法应是一个 Generator 函数，通过 yield return true 来表示动 作计算完毕待下发，释放 CPU 资源并等到下一个周期再继续计算动作。这种方法可以使得动作编排足够紧凑，不需要复杂的状态机设计即可完成复杂的动作序列。通过 Get()方法返回一个 IEnumerable <bool> 对象后，可产生一个 DriveTask 类进行动作执行。例子如下：

```
class LiftFork : MovementDefinition
 {
     public int liftTarget = 0;
     public int liftStatu = 0;

     public override IEnumerable<bool> Get()
       {
           Console.WriteLine($"Start lifting to {liftTarget}");
           SWDef.self.liftTarget = liftTarget;
           //根据货叉目标位置判断提升还是下降，并且记忆其状态
```

```
            if (SWDef.self.liftPos < SWDef.self.liftTarget) { SWDef.self.lift = 1;
                    liftStatu = 1; }
            else if (SWDef.self.liftPos >= SWDef.self.liftTarget) { SWDef.self.lift = -1;
                        liftStatu = -1; }
            //如果目标位置和当前位置在 10mm 以内，循环跳出
            while (Math.Abs(SWDef.self.liftPos - liftTarget) > 10)
                yield return true;
            SWDef.self.lift = 0;//货叉动作使能关闭
            SWDef.self.liftPos = liftStatu;//给定到位状态
            Thread.Sleep(500);
            Console.WriteLine($"Done lifting to {liftTarget}");
        }
}
```

1.2 以上一个货叉的动作就写好了，下面进行这个动作的测试

3，定义动作测试例程

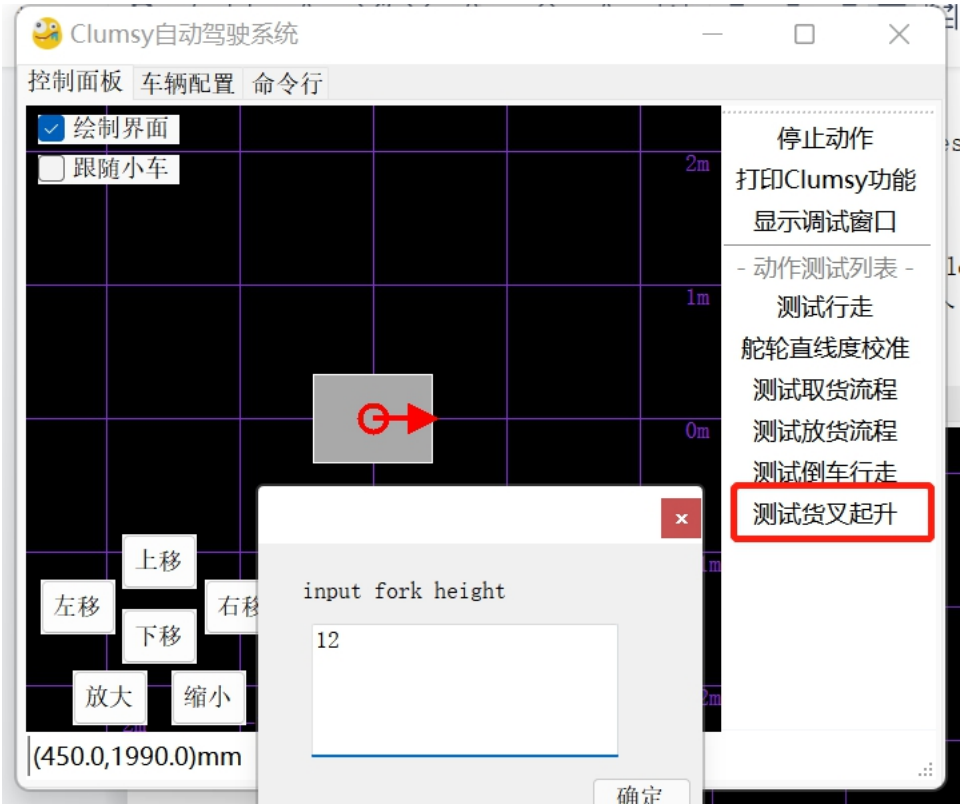
1，定义一个动作测试的流程

1.1 定义一个 ForkTest 类，继承 MovementTest，动作分为开始测试和停止，重写 TestStop 以及 Test 方法，Test 中先调出 UI 供输入货叉提升高度值，然后新建一个 DriveTask 任务，排列到任务中，进行执行，等到 LiftFork 中的 Get 方法完成返回 true ，任务结束。

```
class ForkTest : MovementTest
{
   private DriveTask dt;
   public override void TestStop()
   {
     dt?.Stop();
   }

  public override void Test()
   {
     //调出 UI 输入货叉提升高度值
     if (InputBox.ShowDialog("input fork height") != DialogResult.OK) return;
     dt = new DriveTask(new LiftFork()
      { liftTarget=Convert.ToInt32(InputBox.ResultValue) }.Get());
   }
}
```

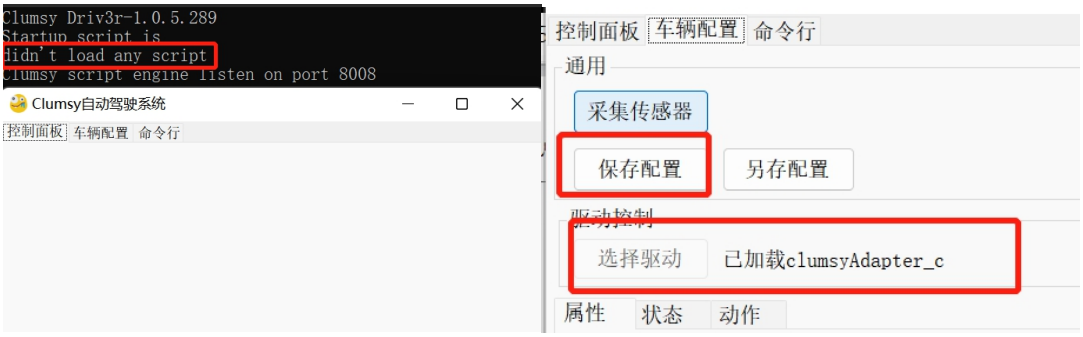
1.2,进行测试,我们重新编译 clumsy,在目录 build\clumsy 找到其生成的 dll,放到 clumsy.exe 目录下，我们打开 clsumy.exe，可以看到界面上有个“测试货叉起升”的按钮



点击即可进行动作测试，如果需要停止点击 停止动作 即可。

1.3 一个简单的动作测试就做好了，如果小车有其他功能也可以按照这个方法进行编写。

注意：如果 clumsy 打开出现界面空白，后台没有加载插件或者加载其他插件失败请，点击车辆配置，选择驱动，保存配置，再重启 clumsy 即可。



三，AMRScene

1，小车的动作编写

1，实现小车类

1.1 我们在 DemoCar.cs 中创建一个 DemoCar 并且继承 ClumsyCar，并且在 DemoCar() 方法中放

置我们需要执行的方法，用 Create()生成一个小车，给小车初始属性

```
public class DemoCar : ClumsyCar
{

  static  DemoCar()
  {
      //初始化小车需要实现的方法
  }

  public static async Task<DemoCar> Create() // boilerplate
  {
     var fl = new DemoCar()
     {
        lstatus = "连接中",
        address = "127.0.0.1",
        name = $"叉车底盘",
        speed = 50,
        haveCoordination = true
      };
     return fl;
  }
}
```

2，编写小车动作

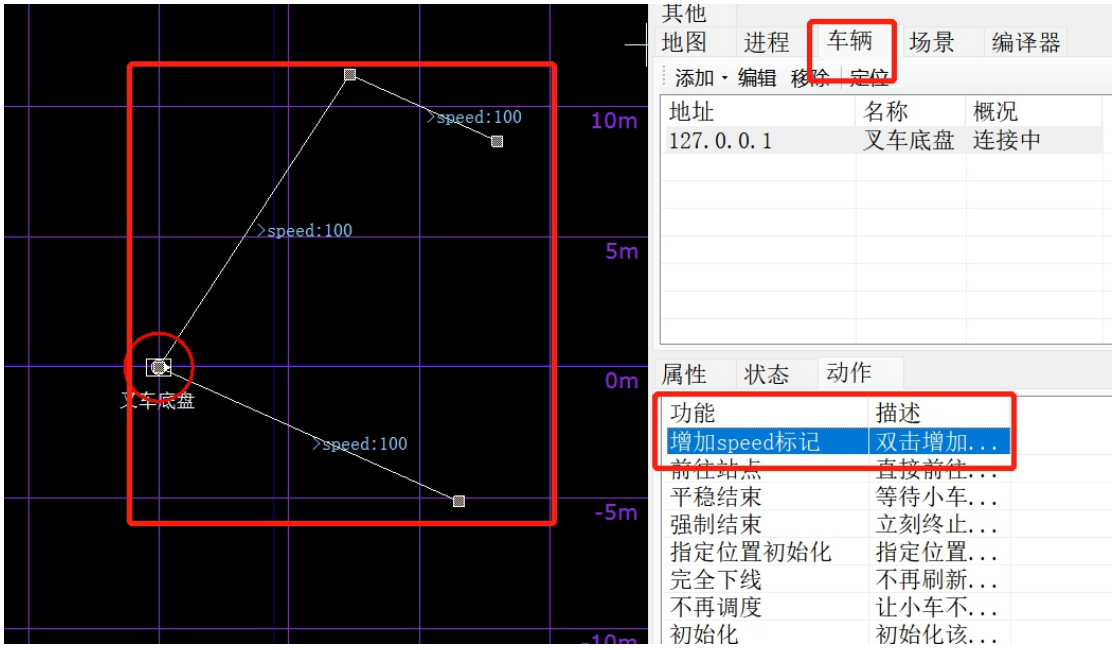
2.1 如果我想把所有线段都增加一个默认的 speed 标记，对于小场景可以手动添加，对于大场景，我们可以动态的进行添加。

2.2 对于小车动作，我们带上 MethodMember 的标记，会显示在界面-车辆-选择小车-动作里面

```
[MethodMember(name = "增加 speed 标记", desc = "双击增加 speed 标记")]
public void AddSpeed()
{
   //获取场景中所有的路线进行遍历
   foreach (var tracks in SimpleLib.GetAllTracks())
    {
      //判断路线是否有 speed 字段，如果有不添加
      if (!tracks.fields.ContainsKey("speed"))
      {
         //给所有的线段加上 100 的速度，当前也可以根据业务逻辑等灵活使用
        tracks.fields.Add("speed","100");
      }
   }
}
```

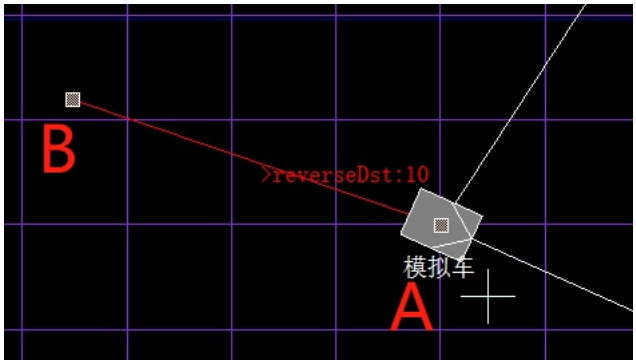
2.3 如下图，我们写好后生成 dll，到\build\Simple 目录下，打开 simpeComposer，添加托盘车，选中小车就可以看到增加 speed 标记的方法，双击就可以把所有线段增加速度

拓展 1: 这个方法的好处是我们可以根据现场业务逻辑灵活使用，如果现场都是重复性工位，每个工位都需要增加一个 shelf 字段，那么我们就可以动态的给全场景增加和删除字段了。
拓展 2：我们也可以给小车增加其他功能比如关闭障碍物，远程关机，屏蔽报警等功能



3，编写 Coder-解释性编程

3.1 编写 coder 可以根据业务场景灵活的调用 AGV.cs 方法，我们示范一个可以让叉车 A-B 倒走，B-A 正走的案例
3.2 首先我们线段标记 reverseDst:10 代表需要倒走的目标点



3.3 然后程序中，我们需要在 TrackFields 类里面增加 reverseDst 属性

```
class TrackFields
{
    public int speed = -1;
    public int reverseDst = -1;
}
```

3.4 我们在 useVerb 里面判断"track.reverseDst == dst.id" 如果条件满足，就执行 templateString 内容，调用 AGV.cs 方法里面的 ReverseGo 方法。

```
[TemplateTrackCoderSettings(
        priority = 5,
        useVerb = "track.reverseDst == dst.id",
        blockVerb = "true",
        templateString =
"agv.ReverseGo(${src.x},${src.y},${src.id},${dst.x},${dst.y},${dst.id},${track.id}," +
                        "${track.speed});",
        siteFields = typeof(SiteFields),
        trackFields = typeof(TrackFields))]
```

这样当我们需要小车从 A-B 走的时候就会自动将 ReverseGo 方法打包到 code 里面了

2，生成一个路径任务

1，创建一个移动任务

1.1 我们首先要了解 Simple 的机制，Simple 中的路线编译器实现小车移动以及功能的脚本下发，先创建一个计划 SegmentPlan，然后对这个计划进行寻路 FindRoute,会返回一个路线脚本 code（包含 A-B 路线所有站点以及路线功能）,然后 SendScript(code)即可下发给 Clumsy，Clumsy 解析然后进行移动或功能的实现。

1.2 了解以上我们开始编写一个简单的 move 动作，这一块我们在 Common.cs 里面编写，这个方法和选中小车右击去某个点类似，具体流程如下方法：

```
public static Task<bool> move(AbstractCar car ,int dstId)
{
    Log.Debug($"execute move job[src={car.siteID}, dest={dstId}]");
    //使用目标小车创建一个计划任务
    var mplan = new SegmentPlan() { usingCar = car ,};
    //对该计划进行寻路
    var weight = mplan.FindRoute(
        SimpleLib.GetSite(car.siteID),
        SimpleLib.GetSite(dstId));
    //给小车增加已占用的标记，防止重复发任务
    car.tags.Add("occupied", $"go{dstId}");
    return Task.Run(() =>
    {
        try
        {
            var code = "";
            code += $"{mplan.Code()};" +
                    $"agv.Wait();"
                    ;
            Log.Debug($"sending {code}");
            //将脚本下发给Clumsy
            car.SendScript(code);
            //更新小车当前ID
            car.siteID = dstId;
            //移除已占用标记
            car.tags.Remove("occupied");
            return true;
        }
        catch (Exception ex)
        {
            Log.Debug($"send script error {ex.Message}");
            car.tags.Remove("occupied");
            return false;
        }
    });
}
```

1.3 我们可以写一个动作或者通过其他接口来调用这个 move 方法 ，这边我们就先用选中右击站点来看下，可以看到，右击站点的时候，小车 Clumsy 命令窗口收到了这一串脚本，这个脚本就是 SendScript(code) 里面 code 的内容，调用了 AGV.cs 的 Go()方法，Go 里面的参数见绿色注释。

```
var host="127.0.0.1";
var agv=new AGV(){id=12, baseSpeed=1};
//srcx:起始点 X 坐标(mm)，srcy:起始点 Y 坐标(mm)，srcid:起始站点 ID，
 //dstx:终点 X 坐标，dsty:终点 Y 坐标，dstid：终点 ID，trackid:路径 ID,speed：路线速度
agv.Go(0,0,5,11507.2998046875,-5134.02587890625,8,9,100);
; agv.Wait();
;agv.Wait();
```

1.4 Clumsy 收到后解析器会进行脚本的解析执行

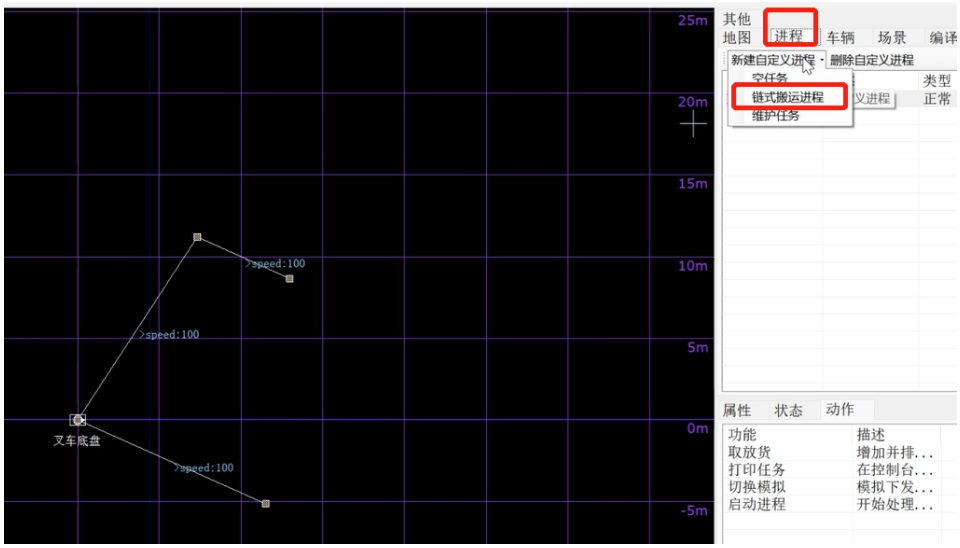
3，创建链式进程任务

1，创建链式进程任务

1.1 一般链式任务用于取放料的业务场景中，小车会先去取料再去放料，因此，我们这边写

了一个通用的链式任务方法。

1.2 进程任务继承 Mission 类，我们可以在进程，新建自定义进程，选择链式进程即可，然后我们双击下面动作的启动进程，就可以通过界面点击进行取放料连贯动作，具体可以参考 ChainedDeliveryMission.cs 文件




```
[MethodMember(name = "取放货", desc = "增加开排队搬运任务链")]
0 个引用
public async void ManualEnqueue()
{
    try
    {
        Delivery former = null;
        while (true)
        {
            var d = new Delivery();
            if (former != null)
                d.former = former;
            Site srcSite, dstSite;

                G.pushStatus("选择取货位");
                var pt1 = await Program.UI.getPoint(new UIOps.getPointOptions() { site = true });
                srcSite = SimpleLib.GetSite(pt1.site);//Scene.currentScene.Sites[pt1.site]);
                d.src = pt1.site;

                if (former == null)
                {
                    G.pushStatus("选择放货位");
                    var pt2 = await Program.UI.getPoint(new UIOps.getPointOptions() { site = true });
                    dstSite = SimpleLib.GetSite(pt2.site);//Scene.currentScene.Sites[pt2.site]);
                    d.dst = pt2.site;
                }
                else
                {
                    d.dst = d.former.src;
                }
                G.pushStatus($"排序了一个叉车搬运任务{d.id}: {srcSite.id} -> {d.dst}, 上序任务:{former?.id}");

                lock (sync)
                    queue.Enqueue(d);
                former = d;
```

1.2 这个是通过界面去调小车，我们也可以通过接口去调用，因此我们需要再写一个可以供调用的接口出来 AutoEnqueue()，也很简单只用把 ManualEnqueue()复制，把起点和终点的赋值方法改变即可，如下

```
public  void AutoEnqueue(int getId,int putId)
{
   try
    {
```

```
        Delivery former = null;
        var missionField = StringDictConvert<ChainedDeliveryParams>.Convert(fields);
        Delivery[] ls;
        while (true)
        {
            var d = new Delivery();
            if (former != null)
            d.former = former;
            Site srcSite, dstSite;
            //获取起点 ID
            srcSite =SimpleLib.GetSite(getId);
            d.src = DemoCar.getId;
            DemoCar.getId = 0;
            if (former == null)
            {
                //获取终点 ID
               dstSite = SimpleLib.GetSite(putId);
               d.dst = DemoCar.putId;
               DemoCar.putId = 0;
             }
             else
             {
                d.dst = d.former.src;
             }
             G.pushStatus($"排序了一个叉车搬运任务{d.id}: {srcSite.id} -> {d.dst}, 上序任
务:{former?.id}");
             lock (sync)
             {
                queue.Enqueue(d);
             }
                    former = d;
          }
        }
        catch (Exception ex)
        {
           G.pushStatus($"结束任务链");
           Console.WriteLine("ex", ex.Message);
        }
}
```

四，核心程序开发指南

暂无。