SLAM技术文档

1. 平台

为了脱离ROS且提供高性能的通信机制、轻量化、跨平台、设计简洁的平台，同时避免重复造轮子且可以集成到cobotsys中，在前期调研中参考了[MRPT](https://www.mrpt.org/)、[CARMEN](http://carmen.sourceforge.net/)、[MOOS](http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php)等开源平台、在综合考虑稳定性、易用性、扩展性等要求下选择[Player](http://playerstage.sourceforge.net/)。该平台足够轻量化，经过修改的完整核心运行代码小于10Mb；该平台为cmake工程，编译依赖仅有Eigen、jpeg、fltk、opengl库，因此甚至可以直接运行在裸ARM板上；同时，该平台足够完备和易于拓展，提供了包括位姿、图像、点云、运动规划等接口且定义新接口简单；更重要的是，player可以在主流仿真平台[stage](http://playerstage.sourceforge.net/index.php?src=stage)、[gazebo](http://gazebosim.org/)仿真。得益于以上特性，player可以说是可以替代ROS的最佳可商业版本（GNU General Public License），目前已知商用此平台的公司有小米、石头科技。

Player的设计架构为：服务器（驱动、算法）+客户端（用户）；服务器为一多线程的

接口类，提供如下四个接口，用户只需实现如下四个接口即可完成算法或硬件驱动。

* virtual MainSetup();
* virtual MainQuit();
* Main();
* ProcessMessage();

客户端如提供了操作服务的多种协议Proxy，如操作里程计的Position2dProxy、激光LaserProxy、栅格地图MapProxy等，支持C++、C、python、java等语言，上手简单。

阅读下面的材料时，读者应该了解移动机器人领域的常见算法，如MIT出版的《概率机器人》中第2章 递归状态估计、第3章 高斯滤波、第4章 非参数滤波、第5章 机器人运动、第6章 机器人感知 、第８章 移动机器人定位:栅格与蒙特卡罗、第9章 占用栅格地图构建、第11章 GraphSLAM算法

链接: https://pan.baidu.com/s/16ADf8ZL1dZBIdI-h21UtUg 提取码: x6xg

1. 安装：

sudo apt-get install libjpeg-dev libfltk1.3-dev

mkdir build && cd build

make -j12

sudo make install

cd extensions/cobotslam

mkdir build &&cd build

make -j12

后会生成libcobotslam.so文件，将此文件安装到/usr/lib目录或cfg文件所在目录即可调用CartoSLAM

1. 实例
2. 运行slam

在cobotsys/examples文件夹下包含cartoslam.cfg、libcobotslam.so、backpack\_2d.lua,他们分别是启动的player配置文件，cartographer运行库，cartographer配置文件.重点关注cartoslam.cfg：

Cfg文件内包含若干个驱动，RobotBase为小车驱动，该驱动提供了position2d接口，此接口为里程计（即位姿和速度信息），０为索引，因为存在多个position2d的接口，port 为串口名字，串口的名字需要配置，在cobotsys/scripts/create\_all\_device\_rules.py已经提供了配置小车串口和雷达串口的配置脚本，运行此脚本后即可绑定串口的名字.

Linuxjoystick为手柄驱动，与上不同的是他需要控制小车，所以requires项需要提供小车的控制接口。

下面的几个驱动也于此类似，name指定驱动的名字，provides指定提供的接口，requires指定订阅的接口，plugin指定动态链接库文件，动态链接库可放在同级目录，或系统lib目录。

driver

(

name "RobotBase"

provides ["position2d:0"]

port "/dev/robotbase"

addimu true

)

driver

(

name "linuxjoystick"

provides ["position2d:10"]

requires ["position2d:0"]

port "/dev/input/js0"

alwayson 1

)

driver(

name "Rplidar"

provides ["laser:1"]

port "/dev/rplidar"

is\_laser\_boundary false

)

driver(

name "To3d"

requires ["position2d:0"]

provides ["position3d:1" ]

)

driver(

name "Toscanangle"

requires ["laser:1"]

provides ["laser:2" ]

)

driver

(

name "Cartoslam"

plugin "libCobotslam"

requires ["position3d:1" "laser:2" ]

provides ["map:0"]

configuration\_directory "/home/cobot/Desktop"

configuration\_basename "backpack\_2d.lua"

laser\_pose [0.27 0 0 0 0 0]

imu\_pose [0 0 0]

)

运行此配置文件：

在终端运行

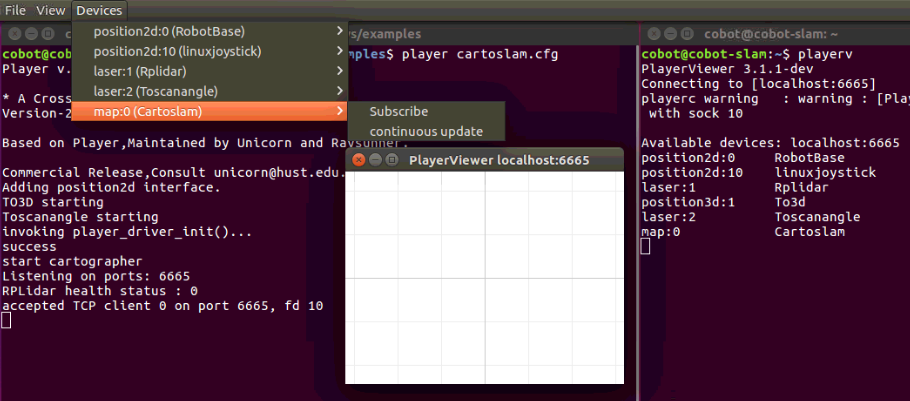
player cartoslam.cfg

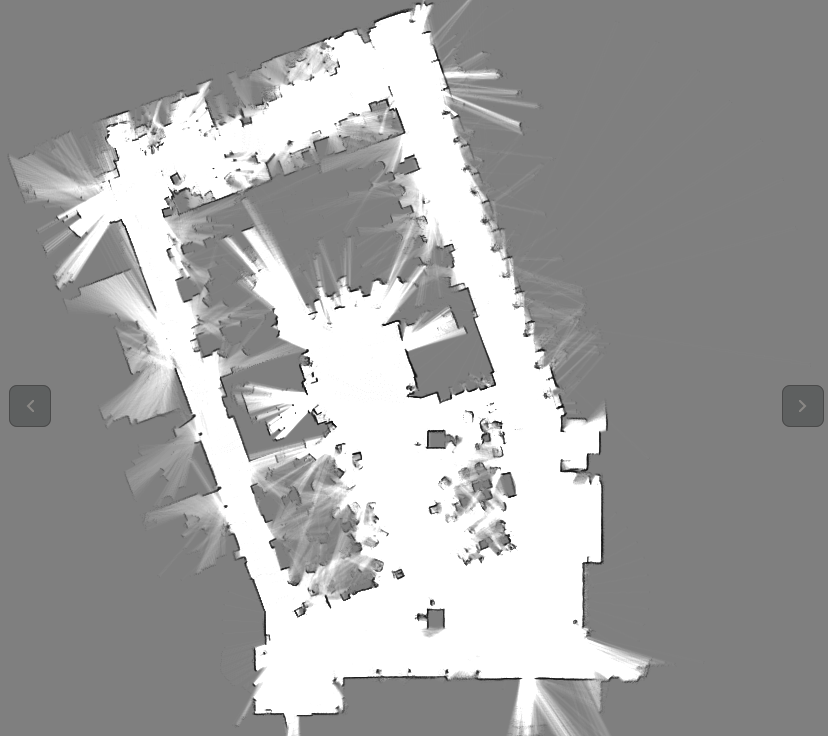
在另一个终端运行

playerv

点击devices->map:0->subscribe即可查看生成的地图，用手柄遥控小车，持续建图，当建图完成时，按ctrl-c即可关闭驱动并在驱动的MainQuit()函数中保存地图。

如下图所示：





1. 导航

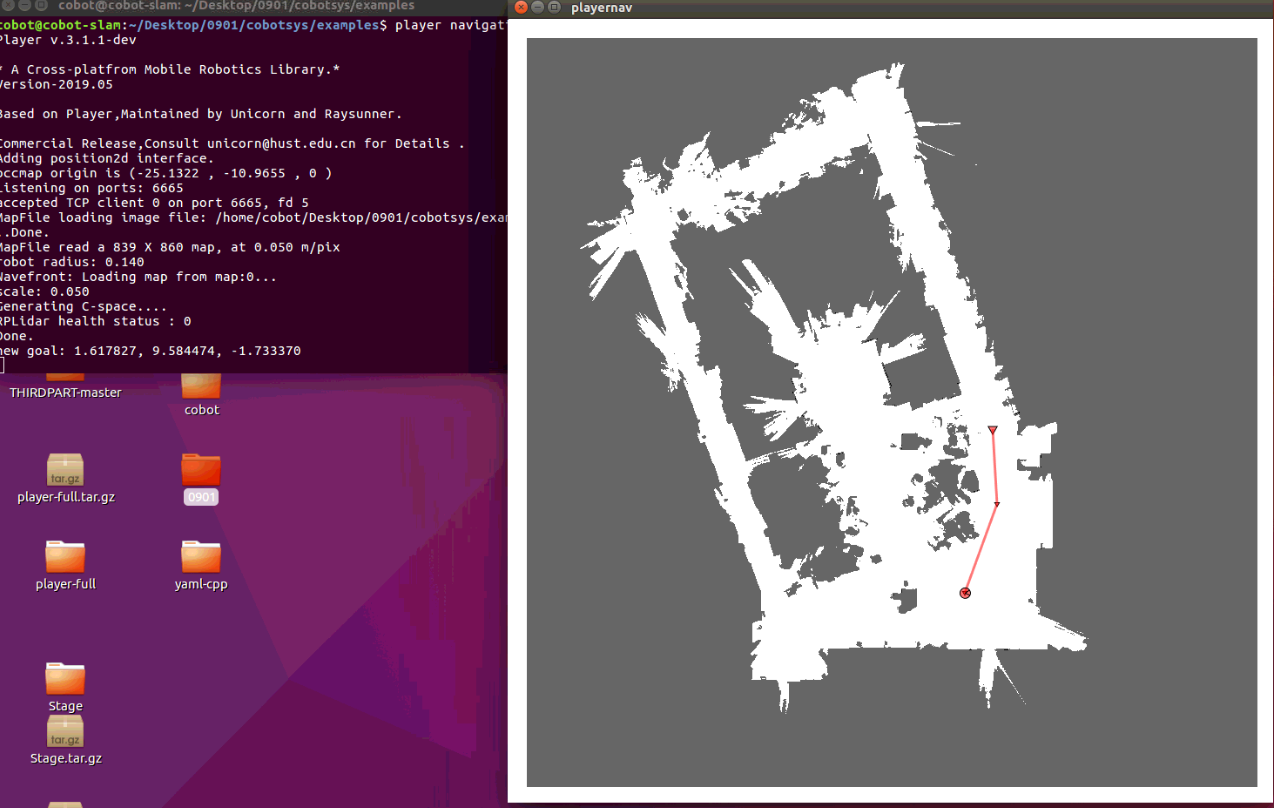
如前，在终端运行

player navigation.cfg

在另一个终端运行

playernav

鼠标给一个目标位姿或代码给出，即可导航



■教程：

官方提供了一些浅显易懂的教程，新版本的文档比较少，可以参考老版本的教程。学习的时候建议使用官方版本的[player](https://github.com/playerproject/player),本工程修改了很多驱动和算法，并且根据项目需求做了精简。

在我的仓库里有官方的stage player，前者是仿真平台

把servers/drivers下的utils拷到player工程里，再在当前的cmakelist添加一句ADD\_SUBDIRECTORY (utils)

编译player时用ccmake ..关掉kinect再make sudo make install

安装之后执行

sudo cp -r /usr/local/lib/libplayer\* /usr/lib

player就可以运行了

stage安装完之后

sudo cp -r /usr/local/lib/libstage\* /usr/lib

sudo cp -r /usr/local/lib/stage\* /usr/lib

试一下 cd cobosys-lite/sim-examples && player simle.cfg

即可在仿真环境运行cartoslam

※☆Player相比于ROS要简单很多很多，框架简单明了,没有那么多抽象的概念和接口，API少到没朋友，一天即可入门，这也是为什么选择player的原因之一！

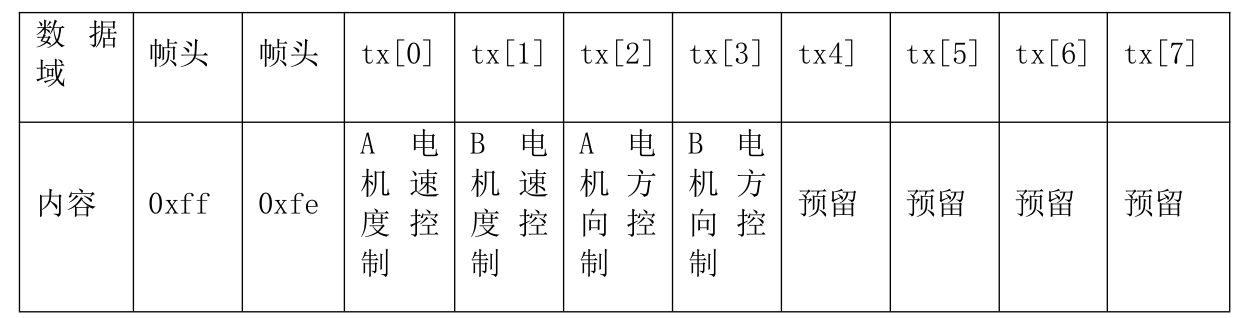
1.<http://playerstage.sourceforge.net/doc/Player-3.0.2/player/group__tutorials.html>

2.<http://playerstage.sourceforge.net/doc/Player-svn/player/group__tutorials.html>

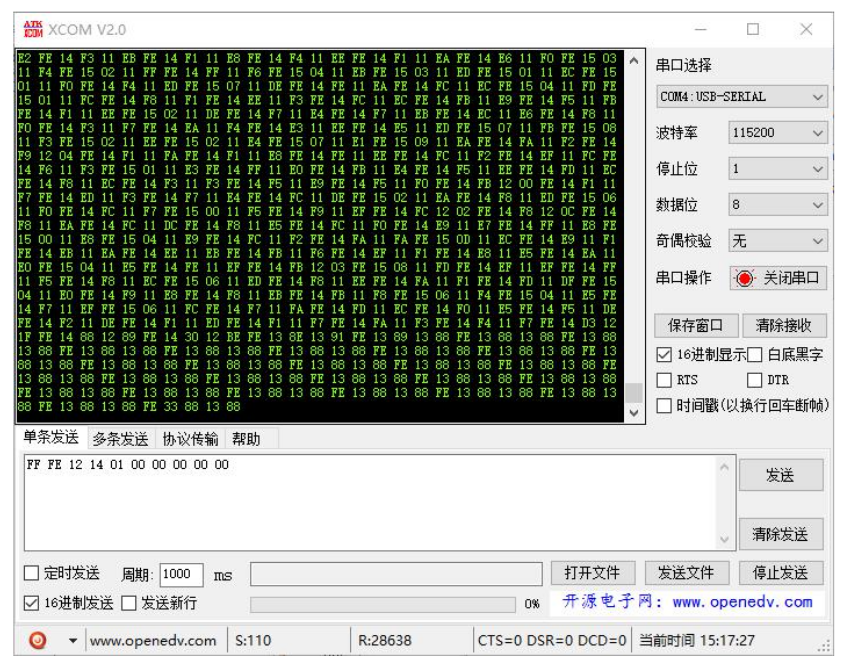
3.<https://playerproject.github.io/>

1. 算法、驱动
2. 底层驱动（STM32、Rplidar S1）
3. 拓展卡尔曼滤波融合
4. Cartographer
5. AMCL
6. 路径规划
7. 底层驱动

* 小车串口控制



可对每个电机单独进行闭环控制,tx[0]控制 A 电机的速度大小,tx[1]控制 B 电机的速度大小,tx[2]控制 A 电机的速度方向,非零是正,零是负。tx[3]控制 B 电机的速度方向,非零是正,零是负。通过电脑串口调试助手即可控制。如图。

这样相当于控制 A 电 机 以 18 的 速度运动。下面解释一下 这 个 18 的 单 位 ,10ms 转 18(0x12) 个脉 冲 , 1s 转 1800 个脉冲,车轮转一圈,输出(编码器线数 500\*减速比 27\*4 倍频)=54000 个脉冲,实际上因为光电编码器精度非常高,为了方便信号处理与传输,在代码里面已经除 以 了 20 的 , 所 以 相 当 于 车 轮 转 一 圈 输 出 2700 个 脉 冲 。 也 就 是 每 秒 转1800/2700 圈,再结合轮胎的直径信息,就可以得到小车的运行速度。注意重载型是 51 减速比。

另外也可通过 TTL 电平的串口 3 控制小车。比如我们使用 STM32 控制器,

发送 10 个字节就可以构成一个完整的控制指令:

Usart\_Send(0xff);

Usart\_Send(0xfe);

Usart\_Send(0x12);

Usart\_Send(0x14);

Usart\_Send(0x01);

Usart\_Send(0x00);

Usart\_Send(0x00);

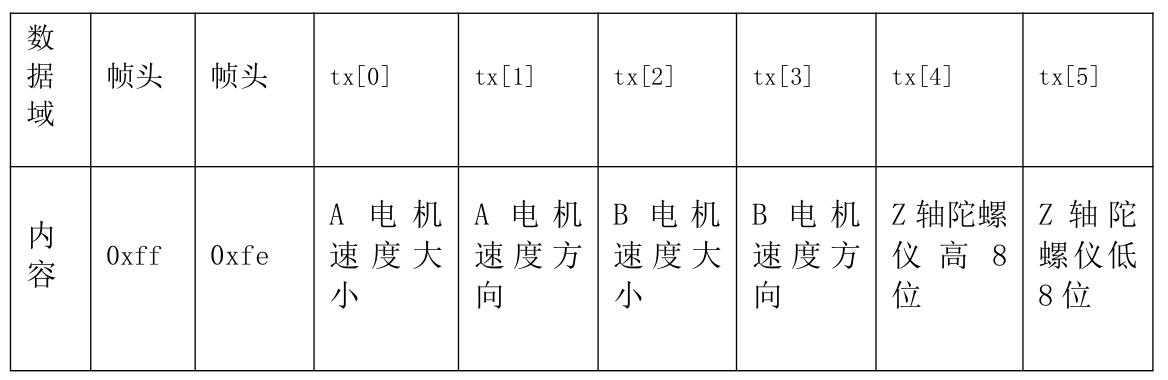
Usart\_Send(0x00);

Usart\_Send(0x00);

Usart\_Send(0x00);

这个指令实现的控制效果和刚才使用电脑发送的指令一致。需要注意的是,发送的数据结构是 10 个字节,不是字符串

另外,通过小车上面的串口 1 和串口 3 都可以采集 IMU 和里程计。可以两个串口同时接收,或者选择任意的串口接收。一帧数据有 8 个字节,其中0xff、0xfe 是帧头,详细说明如下



速度信息很容易理解,单位的解释参考控制部分。方向信息的解释:0 是正,2 是负,1 就代表数据为 0。其中 Z 轴陀螺仪的数据是+32768 再发过来的,所以,处理之后的真实 Z 轴陀螺仪数据应该是:

GyroZ=(tx[4]\*256+tx[5]-32768)/16.4 deg/s

* 获取(差分轮，不同的运动结构有不同的运动学)⾥程计:

利⽤编码器精度(即转⼀圈的脉冲数) \* 减速⽐ \* 2(A, B两个相位) / 轮⼦周⻓(单位m) \* 在1附近的⽐例尺就得到了每⽶的脉冲数,然后经过固定时常后的总脉冲数减去上⼀时刻脉冲数, 然后利⽤差值除以上⾯得到的每⽶的脉冲数,就得到每个轮⼦在固定时⻓dt⾛的距离, 这样就得到每个轮⼦在dt内的速度, 然后把当前时刻的脉冲数更新到上⼀时刻的脉冲数

利⽤运动学解析算出整个⼩⻋的运动速度, 两轮的整体线速度为两个轮胎线速度的平均值, ⾓

速度为右轮减左轮除以轮间距

vx = (vl + vr) / 2

vth = (vr - vl) / l 　　　// l 为两轮之间的间距

根据得到的速度和时间推算出⾥程计(x,y,theta)

x += vx \* cos(theta) \* dt

y += vx \* sin(theta) \* dt

theta= vth \* dt

见server/drivers/position/RobotBase/robotbase.cpp中WriteMotorVelocity（）和ProcessOdometry（）

■rate\_是校准系数，不同的负载、路面环境需要校准，校准的原则是：若里程计大于实际距离，rate\_往＜１调整，反之往＞１调整，直至里程计近似等于实际距离。

* ＥＫＦ融合

由差分轮的运动模型

x += vx \* cos(theta) \* dt

y += vx \* sin(theta) \* dt

theta= vth \* dt

vth = (vr - vl) / l

可见，theta、vth极不准确，积分累计误差大且会影响ｘｙ的值，故通过ｉｍｕ融合修正里程计，该部分代码见server/drivers/position/RobotBase/player\_ekf\_fusion.h

读者应熟悉常见的参数及非参数滤波器如贝叶斯滤波、卡尔曼滤波、拓展卡尔曼滤波、粒子滤波、信息滤波等，这些滤波器都是后面会用到的，该部分不加推导的给出此卡尔曼滤波的具体形式：

状态向量： X\_k（x,y,yaw,vx,vyaw）vyaw为里程计yaw角速度

观测向量：Z\_k(vx,vyaw,vimu)　vimu为imu测量yaw角速度

运动方程：X\_k+1=f(X\_k)　f为运动模型，见代码

X\_k\_ << pos\_data.pos.px + pos\_data.vel.px \* cos(pos\_data.pos.pa) \* dt,

pos\_data.pos.py + pos\_data.vel.px \* sin(pos\_data.pos.pa) \* dt,

pos\_data.pos.pa + pos\_data.vel.pa \* dt,

pos\_data.vel.px,

pos\_data.vel.pa;

f的雅克比为

1, 0, -pos\_data.vel.px \* dt \* sin(pos\_data.pos.pa), dt \* cos(pos\_data.pos.pa), 0,

0, 1, -pos\_data.vel.px \* dt \* cos(pos\_data.pos.pa), dt \* sin(pos\_data.pos.pa), 0,

0, 0, 1, 0, dt,

0, 0, 0, 1, 0,

0, 0, 0, 0, 1;

观测方程：Z\_k+1=Z\_k

更新过程代码代码给出了详细的注释，这里不再给出。

* Rplidar S1

读者应该熟悉激光雷达的传感器模型，简单的说，激光雷达的数据是在极坐标系下按照角度分辨率为步长的一组角度-距离值，例如在半径为１的圆弧障碍物中心放置一激光雷达,该激光雷达分辨率为１°，最小角０°，最大角３５９．９°激光数据为[１,1.2,1.01,.....1.0]，共有３６０组距离值（包含噪声），通常激光雷达包含强度值，表示激光信号强度，强度值越大，距离值越可靠。

该部分驱动位于cobotsys/server/drivers/laser中，包含rplidar sdk和驱动文件,该驱动移植自[ros](https://github.com/slamtec/rplidar_ros)，具体的协议细节见[sdk文档](http://www.slamtec.com/cn/Support" \l "rplidar-s1)。

■该驱动需要注意一个参数is\_laser\_boundary，这个参数是指激光无反射时（超过最大距离时）如何处理距离，在slam中把他设成０以避免大范围的噪声，在导航时设置为最大距离（该雷达最大扫描距离40m）.

* Joystick

手柄的驱动位于cobotsys/server/drivers/joystick，该驱动直接调用linux下的linux/joystick.h，该驱动响应JS\_EVENT\_AXIS，并读取轴上数值，作为速度控制指令发送到小车上，具体见readJoystick()、sendVelocity()。

* 地图

地图驱动位于cobotsys/server/drivers/map，地图文件位于cobotsys/examples/maps

地图为占据栅格地图，栅格地图在形式上是个图像，灰度值代表概率值，通常有如下参数：

分辨率：每个像素的长度　CartoSLAM构建的地图分辨率为0.05m/pixels

地图原点：起始点，用于对齐地图与里程计，即地图左下角格子的坐标，相当于ROS中map->odom的坐标变换

阈值：occupied\_thresh、free\_thresh空闲和占据的阈值，目前取0.95为占据阈值 0.1为空闲阈值

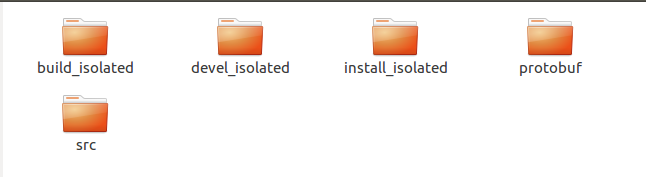
Mapfile驱动读取pgm地图文件和yaml参数文件，发布ｍａｐ作为导航的输入。此文件为CartoSLAM的建图结果，读取ｐｇｍ用到了gdk-pixbuf库，读取ｙａｍｌ文件用到了[yaml-cpp](https://github.com/jbeder/yaml-cpp)

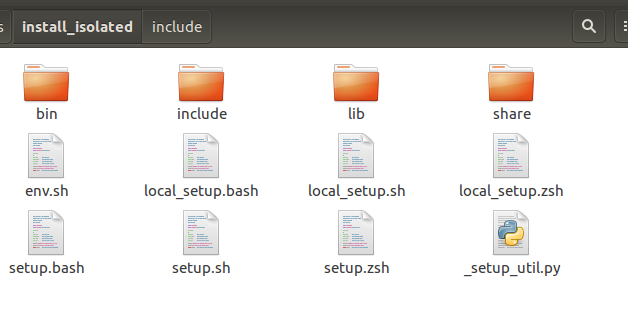
■编译yaml-cpp时注意不要采用默认的编译选项，否则无法链接，ｍａｋｅ时加参数-DBUILD\_SHARED\_LIBS=ON生成动态链接库。

* Cartoslam位于cobotsys/extensions/Cartoslam,该部分是最为复杂、庞大的一部分，首先从工程的角度上阐述如何配置、调用，然后从理论上介绍一下原理。
* Cartoslam不同与上面的驱动，它是一个动态链接库形式的驱动，至于为什么采用动态链接库，主要考虑了一下几点：

1. 直接集成进player的话，工程的makefile会变得混乱不堪，参考ros部分的makefile就可知一二。
2. 不方便开发，作为动态链接库的形式可以单独开一个工程，方便在IDE中开发。
3. 为了方便调试，集成到系统内，每更改一次代码就需要重新编译安装整个cobotsys工程，采用动态链接库的形式只需编译其本身即可。

Cartoslam的主要依赖为[cartographer](https://github.com/googlecartographer/cartographer)，ceres,protobuf，安装见此[文档](https://google-cartographer.readthedocs.io/en/latest/)，但是仅安装cartographer无法提供某些proto生成的头文件，因此便捷的方法是安装ros版本





此时工程所需的所有中间文件均可在中间目录build\devel\install中找到。

安装后建立一个CMAKE工程，cmakelists文件位于工程顶层。

已经移除了ROS下面的不是很重要，但是链接的东西还是这样！

~~下面的部分引入cartographer，否则无法找到cartographer。~~

~~set(CMAKE\_MODULE\_PATH ~/catkin\_ws/src/cartographer/cmake/modules)~~

~~set(cartographer\_DIR ~/catkin\_ws/install\_isolated/share/cartographer)~~

~~find\_package(cartographer REQUIRED)~~

~~include("${CARTOGRAPHER\_CMAKE\_DIR}/functions.cmake")~~

~~google\_initialize\_cartographer\_project()~~

~~google\_enable\_testing()~~

~~下面的部分引入player、cartogarpher等头文件~~

~~INCLUDE\_DIRECTORIES(/usr/include/eigen3)~~

~~INCLUDE\_DIRECTORIES(~/catkin\_ws/src/cartographer)~~

~~INCLUDE\_DIRECTORIES(~/catkin\_ws/build\_isolated/cartographer/install)~~

~~INCLUDE\_DIRECTORIES(/usr/local/include/player-3.1/)~~

~~include\_directories(/usr/include/lua5.2)~~

~~INCLUDE\_DIRECTORIES(~/catkin\_ws/install\_isolated/include)~~

~~下面的部分生成动态链接库并链接player、cartogarpher~~

~~add\_library(${PROJECT\_NAME} SHARED ${ALL\_SRCS})~~

~~target\_link\_libraries(${PROJECT\_NAME} PUBLIC cartographer)~~

~~target\_link\_libraries(${PROJECT\_NAME} PUBLIC litedriverdef ${LIB\_CONFIG})~~

工程的源文件为

　Cartoslam/src/Cartoslam.cpp

　Cartoslam/src/Cartoslam.h

　Cartoslam/src/map\_builder\_bridge.cpp

　Cartoslam/src/map\_builder\_bridge.h

　Cartoslam/src/sensor\_bridge.cpp

　Cartoslam/src/sensor\_bridge.h

　Cartoslam/src/start.cpp

　Cartoslam/src/start.h

　Cartoslam/src/start\_constants.h

　Cartoslam/src/start\_options.cpp

　Cartoslam/src/start\_options.h

　Cartoslam/src/time\_conversion.cpp

　Cartoslam/src/time\_conversion.h

　Cartoslam/src/trajectory\_option.h

　Cartoslam/src/trajectory\_options.cpp

　Cartoslam/src/utility.h

该部分近似对应cartographer\_ros的文件结构

　cartographer\_ros/map\_builder\_bridge.cc

　cartographer\_ros/map\_builder\_bridge.h

　cartographer\_ros/node.cc

　cartographer\_ros/node.h

　cartographer\_ros/node\_constants.cc

　cartographer\_ros/node\_constants.h

　cartographer\_ros/node\_main.cc

　cartographer\_ros/node\_options.cc

　cartographer\_ros/node\_options.h

　cartographer\_ros/sensor\_bridge.cc

　cartographer\_ros/sensor\_bridge.h

　cartographer\_ros/time\_conversion.cc

　cartographer\_ros/time\_conversion.h

　cartographer\_ros/trajectory\_options.cc

　cartographer\_ros/trajectory\_options.h

由于平台与ros中的差异，仅移植了最关键的部分，剔除了ros中的特有部分。下面分析一下各个源文件。

Cartoslam中是与player相关的驱动，如前所诉，为一多线程的驱动类，构造函数里输入传感器数据来源，包括里程计、激光雷达、（imu）。由于平台将imu和里程计融合了，再将其传入已意义不大，但为了后续平台的开发任然保留，在配置文件backpack\_2d.lua中配置TRAJECTORY\_BUILDER\_2D.use\_imu\_data = false，在（３Ｄ情况除外，因为需要ｚ轴加速度ｇ），输入的传感器数据的格式为：

■里程计

位姿Pose(x,y,theta)　imu速度gyro value(roll,pitch,yaw)　imu加速度Acc value(x,y,z)

在cartographer,roll pitch,yaw的旋转是x,y,z，把这些数据放在position3d中，但是和position3d规定的意义不同了，为此在cobotsys/server/drivers/utils下有转换工具，To3d提供了上述的转换

//Robot Pose(x,y,theta)

　　p3d.pos.px=p2->pos.px;

p3d.pos.py=p2->pos.py;

p3d.pos.pz=p2->pos.pa;

//gyro value velocity (roll,pitch,yaw)

p3d.pos.proll=p2->vel.pa;

p3d.pos.ppitch=0;

p3d.pos.pyaw=0;

//Acc value(x,y,z)

p3d.vel.px=(p2->vel.px-g\_p2d.vel.px)/dt;

p3d.vel.py=(p2->vel.py-g\_p2d.vel.py)/dt;

p3d.vel.pz=9.7;

具体的对应关系见To3d.cc

■laser和我们提到的传感器模型相似，不同的是传入的类型scanangle,这种类型没有固定的角度分辨率(可能是为了用超声波来代替laser，每个超声波安装的位置有角度约束),所以还需要提供距离对应的角度值，这里角度数组就是angle=0:1:360;具体来说就是

laserDataScanangle.angles[i]=scan->min\_angle+i\*scan->resolution;

该算法位于cobotsys/server/drivers/utils/toscanangle

回到构造函数内，传入传感器数据后，还提供了供用户获取地图的接口，此接口可参考如前的mapfile，除此之外，还传入了配置参数，如雷达安装位置，ｉｍｕ安装位置，这些代码中注释很清晰，不再赘述。

MainSetup函数主要是载入配置cartographer参数，该参数文件包括

　cartographer/configuration\_files/map\_builder.lua

　cartographer/configuration\_files/map\_builder\_server.lua

　cartographer/configuration\_files/pose\_graph.lua

　cartographer/configuration\_files/trajectory\_builder.lua

　cartographer/configuration\_files/trajectory\_builder\_2d.lua

以及backpack\_2d.lua，我们重点需要关注的是backpack\_2d.lua。一般情况下调整此文件的配置即可，关于此文件，[官网](https://google-cartographer-ros.readthedocs.io/en/latest/configuration.html)已经给出了详细的介绍了，在此仅说明一下ros中关联的几个ｆｒａｍｅ的概念，frame既是坐标系，ros中每个传感器拥有一个坐标系，其相对位姿是通过ros\_tf转换的，在此我们简化了此功能，因为都是固定坐标系。

**tracking\_frame**

The ROS frame ID of the frame that is tracked by the SLAM algorithm. If an IMU is used, it should be at its position, although it might be rotated. A common choice is “imu\_link”.

因为没有ｉｍｕ坐标系，即设置为base\_link,意为小车的刚体坐标系。

**published\_frame**

The ROS frame ID to use as the child frame for publishing poses. For example “odom” if an “odom” frame is supplied by a different part of the system. In this case the pose of “odom” in the map\_frame will be published. Otherwise, setting it to “base\_link” is likely appropriate.

由于使用了ｏｄｏｍ，仍设置为base\_link。

**num\_multi\_echo\_laser\_scans**

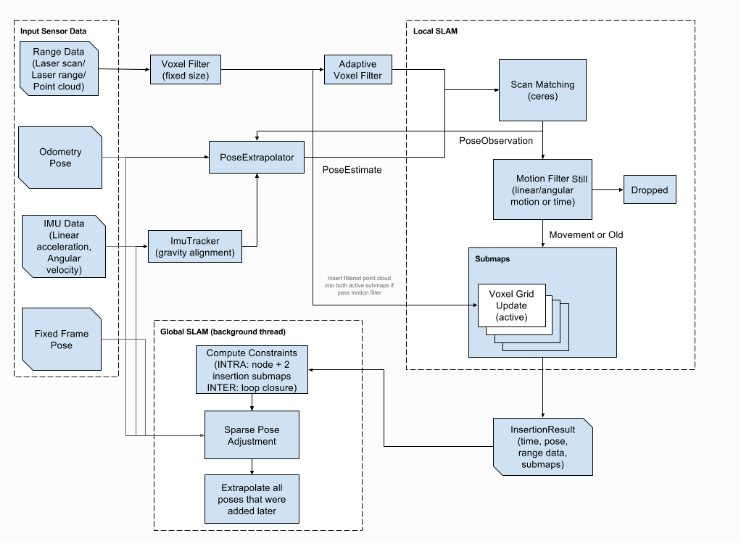
Number of multi-echo laser scan topics to subscribe to. Subscribes to[sensor\_msgs/MultiEchoLaserScan](http://docs.ros.org/api/sensor_msgs/html/msg/MultiEchoLaserScan.html) on the “echoes” topic for one laser scanner, or topics “echoes\_1”, “echoes\_2”, etc. for multiple laser scanners.

在使用3D laser时，出于时延的考虑，把laser的数据分开发上去而不是等laser转一圈再发上去（10Hz 2D雷达也就0.1s 几乎没啥影响，因此与之关联的方法没有实现），将此设置为０，而num\_laser\_scans = 1，即一次性发上去。

再回到MainSetup，该函数中构造了start对象(对应ros中的node.cc)，start类是整个算法的入口，包括处理传感器数据，构造位姿图，优化等，理解该部分需要将注意力转移到cartographer的源码中，后续的原理部分将涉及此处。

对代码理解有问题的，可以参考cartographer\_ros的代码，相对于移植后的代码，源代码更加完整。

●原理部分



Carographer从本质上来说还是Graph SLAM，Graph SLAM主要的工作量还是在于构建位姿图，并且优化位姿图，Graph slam算法在《概率机器人》第十一章节有详细的介绍。Carographer优化的代价函数[官网](https://google-cartographer.readthedocs.io/en/latest/cost_functions.html)和论文中均给出了详细介绍，在此不在赘述。

原理这一部分不大好写，放在PPT内介绍。

■AMCL

自适应蒙特卡洛定位(AMCL)是依靠先验地图和传感器数据确定机器人在地图中的位置的一种定位算法，在ROS、MRPT等平台中都是默认的定位算法，player也提供了该算法，位于cobotsys/server/drivers/localization/amcl,它足够完备，支持多种传感器模型，包括gps、超声波、里程计、imu、激光雷达、wifi等。是一个非常具有参考意义的算法实现，它的原理是粒子滤波器，粒子数代表P(X)，即位姿的概率。通过传感器模型和里程计模型，改变粒子的分布情况。比如在机器人定位中，某个粒子A，我觉得这个粒子在这个坐标的可能性很高，那么我给他打高分。下次重新安排所有的粒子的位置的时候，就在这个位置附近多安排一些。这样多来几轮，粒子就都集中到可能性高的位置去了。

自适应体现在：1解决了机器人绑架问题，它会在发现粒子们的平均分数突然降低了（意味着正确的粒子在某次迭代中被抛弃了）的时候，在全局再重新的撒一些粒子。 2解决了粒子数固定的问题，因为有时候当机器人定位差不多得到了的时候，比如这些粒子都集中在一块了，还要维持这么多的粒子没必要，这个时候粒子数可以少一点了。

●路径规划

路径规划算法为wavefront算法，一种朴素的，反向的Dijistra路径规划算法。他的原理是：从地图中的目标点到起始点，像波浪一样不断扩张，遇到障碍物则在障碍物区域不扩张，扩张的表现为为每一个地图点增加其到目标点的距离(代价)值，然后从起始点找代价最小的点，直到到达目标点。举一个例子

设起始点为S，目标点为G，地图为cartoslam生成的栅格地图，0表示空闲区域，1表示占据区域，为G赋值2，G的Moore孩子赋值+1，为3:

S 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 1 1 1 1 1 0 0 0

0 0 0 1 1 1 1 1 1 0 0 0

0 0 0 0 0 0 0 0 0 0 3 3

0 0 0 0 0 0 0 0 0 0 3 2

进行不断扩张，为G的孩子的孩子赋值+1，为4：

S 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 1 1 1 1 1 0 0 0

0 0 0 1 1 1 1 1 1 4 4 4

0 0 0 0 0 0 0 0 0 4 3 3

0 0 0 0 0 0 0 0 0 4 3 2

只为空闲区域赋值，占据区域保持代价值1，最终生成的代价地图为：

14 13 12 11 10 9 8 7 7 7 7 7

13 13 12 11 10 9 8 7 6 6 6 6

13 12 12 1 1 1 1 1 1 5 5 5

13 12 11 1 1 1 1 1 1 4 4 4

13 12 11 10 9 8 7 6 5 4 3 3

13 12 11 10 9 8 7 6 5 4 3 2

S的值为14，G的值为2，从S的邻边不断扩充寻找代价最小且大于G的点，最终生成一条至G的路径（笛卡尔坐标系）： (0,5)->(1,5)->(2,5)->(3,5)->(4,5)->(5,5)->(6,5)->(7,4)->(8,4)->(9,3)->(9,2)->(10,1)->(11,0)

把机器人仅看成平面内的点是不现实的，为此需要考虑机器人的形状，在调用此算法时需要配置的有：

Safety\_dist distance\_epsilon angle\_epsilon分别表示与障碍物应保持的安全距离，即在障碍物的safety\_dist邻域不更新代价值；距离偏差和角度偏差，在偏差内，认为到达了目的地；

* 已知存在的问题：

1. 定位问题：机器人长时间运动会积累，加之环境动态变化，还可能会受到人为干扰（如搬到另一个位置），机器人可能会无法确定自己的确切位置，这是移动机器人领域面临的一大难题之一，可行的解决办法有：在工作区域内贴标签、二维码修正，如在天花板安装二维码，地面安装RFID；采用硬件全局定位，如UWB室内定位；定位失效即回到某已知点，如充电桩等。
2. 路径规划问题：这里的路径规划是朴素的路径规划；不同的场景，不同的任务，采用的路径规划应考虑到该场景的约束，如仓库内抓取多个位置下的多个目标的任务不应独立分解完成否则会严重影响效率。