现在已经是凌晨两点半了, 熟悉我的朋友都知道,我是一个熬不了夜的人, 2018年打robocon的时候队友在造车调车,我作为操作手在场地上睡大觉, 2022年打robomaster, 队友在调车, 我在睡觉… … 可能冥冥之中自有天意, 今天晚上我失眠了,因为有个bug一直困扰着我,就是robot\_hw和ros\_control接口不通的问题……

我从研一的时候开始学ROS,现在都博一了,还在学ROS. ros\_control那一系列的东西我从研一的时候就知道这东西在实时性要求高的控制器场合上需要用的到.但是学习的进度一直很缓慢.

前几次的教程写了urdf的创建,写了自己写一个ros\_control,写了如何将插件和gazebo通讯,但是硬件抽象层robot\_hw一直没有搞定.今天我终于跑通了.在此撒个花.

废话不多说,进入到今天的教程

robot\_hw,全称:robot\_hardware.翻译过来是机器人硬件,但是应该叫它硬件抽象层.上次我们讲解到控制器的概念,控制器对电机发出指令,但是控制器并不直接与硬件进行通讯,他只是输出相应的指令,比如力矩指令,速度指令,或者位置指令..如果你控制的是仿真机器人,那么gazebo集成了一系列硬件抽象层插件,有力矩的,有位置的,有速度的,还有轨迹的等等.只要定义好接口类型,那么就可以完成对接.

如果我们要控制自己的机器人,那么ros不知道你的电机到底支持什么格式,到底通讯协议是什么样子的,所以硬件抽象层就需要自己去编写.理论上只要接口和ros\_control的接口匹配,就可以正常工作…

今天,我带大家手把手写一个ros\_control和与之匹配的robot\_hw

以下是步骤:

新建功能包的部分我就不仔细讲解了

我的功能包名字叫robot\_hw\_test

依赖项:roscpp rospy std\_msgs(好像没用到,习惯性加上了)controller\_manager(控制器管家库) hardware\_interface(硬件抽象层库) joint\_state\_controller(关节状态接口) robot\_state\_publisher pluginlib(插件有关的)

整体文件路径

robot\_hw\_test

config

hardware.yaml

include

joint\_controller\_test.h

robot\_hw\_test.h

src

hw\_interface\_node.cpp

joint\_controller\_test.cpp

robot\_hw\_test.cpp

launch

controller\_only.launch

hw\_test.launch

CMakeLists.txt

package.xml

myrobots\_controller.xml

myrobots\_hw\_plugin.xml

因为ros\_control部分和之前的几乎一毛一样,所以我就不做过多细致讲解,直接附上代码

首先是include

joint\_controller\_test.h

#include <ros/node\_handle.h>

#include <ros/ros.h>

#include <urdf/model.h>

#include <control\_toolbox/pid.h>

#include <boost/scoped\_ptr.hpp>

#include <boost/thread/condition.hpp>

#include <realtime\_tools/realtime\_publisher.h>

#include <hardware\_interface/joint\_command\_interface.h>

#include <controller\_interface/controller.h>

#include <control\_msgs/JointControllerState.h>

#include <std\_msgs/Float64.h>

#include <control\_msgs/JointControllerState.h>

#include <realtime\_tools/realtime\_buffer.h>

#include <controller\_interface/controller.h>

#include <hardware\_interface/joint\_command\_interface.h>

#include <controller\_interface/multi\_interface\_controller.h>

#include <pluginlib/class\_list\_macros.h>

namespace robot\_hw\_test\_ns

{

class joint\_controller\_test: public controller\_interface::MultiInterfaceController

<hardware\_interface::EffortJointInterface>

{

private:

std::vector<hardware\_interface::JointHandle> joint\_handles\_;

ros::NodeHandle nh;

std::vector<double> joint\_state\_;

/\* data \*/

public:

joint\_controller\_test();

~joint\_controller\_test();

bool init(hardware\_interface::RobotHW\* robot\_hw,ros::NodeHandle& nh);

void starting(const ros::Time& time\_now);

void update(const ros::Time& time\_now, const ros::Duration& period);

};

}

然后是cpp文件joint\_controller\_test.cpp

#include "joint\_controller\_test.h"

namespace robot\_hw\_test\_ns

{

joint\_controller\_test::joint\_controller\_test(/\* args \*/)

{

}

joint\_controller\_test::~joint\_controller\_test()

{

}

bool joint\_controller\_test::init(hardware\_interface::RobotHW\* robot\_hw,ros::NodeHandle& nh)

{

std::vector<std::string> joint\_names;

if (!nh.getParam("joint",joint\_names))

{

ROS\_ERROR(

"Agv\_Controller: Invalid or no joint\_names parameters provided, aborting "

"controller init!fuck!!!!!");

return false;

}

auto\* joint\_interface=robot\_hw->get<hardware\_interface::EffortJointInterface>();

for (uint8\_t i = 0; i < joint\_names.size(); i++)

{

joint\_handles\_.push\_back(joint\_interface->getHandle(joint\_names[i]));

ROS\_ERROR\_STREAM("controller info: my name is "<<joint\_handles\_[i].getName());

}

joint\_state\_.resize(joint\_names.size());

return true;

}

void joint\_controller\_test::starting(const ros::Time&)

{

ROS\_WARN("controller starting");

}

这里的update需要解释一下, 我将一个余弦信号通过接口发送出去, 对方如果能接收到这样一个变化的信号,那么就说明接口正常工作,通过打印getposition的返回值,获取robot\_hw传回来的反馈是否正常..

void joint\_controller\_test::update(const ros::Time& time\_now, const ros::Duration& period)

{

double temp = std::cos(time\_now.toSec());

for (size\_t i = 0; i < joint\_handles\_.size(); i++)

{

joint\_handles\_[i].setCommand(temp);

ROS\_ERROR("recive from robot\_hw feedback is %f",joint\_handles\_[i].getPosition());

}

}

} //namespace

PLUGINLIB\_EXPORT\_CLASS(robot\_hw\_test\_ns::joint\_controller\_test,

controller\_interface::ControllerBase)

然后是插件定义部分:myrobots\_controller.xml

<library path="lib/libjoint\_controller\_test">

<class name="robot\_hw\_test\_ns/joint\_controller\_test"

type="robot\_hw\_test\_ns::joint\_controller\_test"

base\_class\_type="controller\_interface::ControllerBase" />

<description>

fuck test??????????

</description>

</library>

编译规则中将这个cpp文件变成一个动态连接库加载进去

add\_library(joint\_controller\_test

src/joint\_controller\_test.cpp

)

target\_link\_libraries(joint\_controller\_test ${catkin\_LIBRARIES})

接下来,我们来编写今天的重头戏,robot\_hw

通过阅读网络上的教程,我发现robot\_hw需要单独开一个程序,从程序中调用controller\_manager来加载控制器.

首先编写robot\_hw的插件部分,和ros\_control的套路比较类似.注释如下

include文件:robot\_hw\_test.h

#include <iostream>

#include <hardware\_interface/joint\_state\_interface.h>

#include <hardware\_interface/joint\_command\_interface.h>

#include <hardware\_interface/robot\_hw.h>

#include <pluginlib/class\_list\_macros.hpp>

#include <ros/ros.h>

namespace robot\_hw\_test\_ns

{

//继承robothw类

class MyRobotInterface : public hardware\_interface::RobotHW

{

public:

MyRobotInterface();//构造函数

MyRobotInterface(ros::NodeHandle& nh);//构造函数,传入ros句柄,应用ros的资源

~MyRobotInterface();//析构函数

bool init(ros::NodeHandle& root\_nh, ros::NodeHandle& robot\_hw\_nh);//初始化代码

void read(const ros::Time& time, const ros::Duration& period);//将硬件资源读取到接口中,发送给控制器

void write(const ros::Time& time, const ros::Duration& period);//将控制器给出的指令发送给硬件

private:

ros::NodeHandle nh\_;//ros句柄

//interfaces

hardware\_interface::JointStateInterface joint\_state\_interface;//硬件接口,状态反馈

hardware\_interface::EffortJointInterface effort\_joint\_interface;//硬件接口,力矩接口

int num\_joints;//

std::vector<std::string> joint\_name;

std::vector<double> joint\_position\_state;//位置状态

std::vector<double> joint\_velocity\_state;//速度状态

std::vector<double> joint\_effort\_state;//力矩状态

std::vector<double> joint\_effort\_command;//力矩指令

};

}

cpp文件robot\_hw\_test.cpp

#include "robot\_hw\_test.h"//包含自定义头文件

namespace robot\_hw\_test\_ns

{

MyRobotInterface::MyRobotInterface()

{

;

}

MyRobotInterface::MyRobotInterface(ros::NodeHandle& nh)

{

;

}

MyRobotInterface::~MyRobotInterface()

{;}

//构造函数我都没管他

//init函数

bool MyRobotInterface::init(ros::NodeHandle& root\_nh, ros::NodeHandle& robot\_hw\_nh)

{

bool ret = robot\_hw\_nh.getParam("/mylink/robot\_hw\_test/joints", joint\_name);//从参数服务器中获取name, 也就是从config文件夹下的yaml文件中获取参数,这个yaml文件我在后面会给出来,大家可以按照规则自己编写里面的内容,补充一点c++编程小知识,这里的 joint\_name是一个vector类型的变量,直接写这个名字,返回的是这个变量第一个数的指针,那么这个getparam的第二个参数实际上是一个传出参数,将获取到的内容赋值给 joint\_name变量.

ROS\_ERROR("getParam ret= %d",ret);//这里纯粹是调试用的,前期调bug的时候看看是不是运行正常

num\_joints = joint\_name.size();//获取joint有几个

for (size\_t i = 0; i < num\_joints; i++)

{

ROS\_ERROR("jointname=%s",joint\_name[i].c\_str());//打印一下,当初调试用的

}

joint\_position\_state.resize(num\_joints);//将下面的这些double类型的vector 的大小重构一下

joint\_velocity\_state.resize(num\_joints);

joint\_effort\_state.resize(num\_joints);

joint\_effort\_command.resize(num\_joints);

for(int i=0; i<num\_joints; i++)

{

//State

hardware\_interface::JointStateHandle jointStateHandle(joint\_name[i].c\_str(), &joint\_position\_state[i], &joint\_velocity\_state[i], &joint\_effort\_state[i]);

//这里是重头戏,将对应的jointname和对应的位置,速度,力 绑定到 jointStateHandle中去

joint\_state\_interface.registerHandle(jointStateHandle);

我们通过 joint\_state\_interface调用注册函数registerHandle,将刚才的那一串注册到接口中去.从而,获取joint\_i的位置,速度,力矩等信息,就绑定到了固定的一块内存区域,只要有接口要调用,就会从这几个内存区域调用

ROS\_INFO("joint\_name[%d].c\_str()=%s",i,jointStateHandle.getName().c\_str());

//Effort

hardware\_interface::JointHandle jointEffortHandle(joint\_state\_interface.getHandle(joint\_name[i]), &joint\_effort\_command[i]);

effort\_joint\_interface.registerHandle(jointEffortHandle);

//同理

}

registerInterface(&joint\_state\_interface); //将类中的接口注册到ros中

registerInterface(&effort\_joint\_interface);

return true;

}

void MyRobotInterface::read(const ros::Time& time, const ros::Duration& period)

{

static int t =0 ;

if (t<3)

{

ROS\_ERROR("read is run");

}

t++;

double temp = std::sin(time.toSec());

//本篇的重点在于ros\_control和robot\_hw的通讯,硬件抽象层与硬件的交互不作为重点,这里只要ros\_control能正常读到发过去的反馈,就代表成功

for(int i=0;i < num\_joints;i++){

joint\_position\_state[i]=temp+1;

}

}

void MyRobotInterface::write(const ros::Time& time, const ros::Duration& period)

{

for(int i=0;i < num\_joints;i++)

{

ROS\_WARN("recive from controller joint %d is %f",i,joint\_effort\_command[i]);

//这里打印一下通过接口获取到的roscontrol发来的数据,有变化的数据就代表成功

}

}

}

PLUGINLIB\_EXPORT\_CLASS(robot\_hw\_test\_ns::MyRobotInterface, hardware\_interface::RobotHW)

//注册插件

编写myrobots\_hw\_plugin.xml

<library path="lib/libMyRobotInterface">

<class name="robot\_hw\_test\_ns/MyRobotInterface"

type="robot\_hw\_test\_ns::MyRobotInterface"

base\_class\_type="hardware\_interface::RobotHW">

<description>

Interface for MyRobot1

</description>

</class>

</library>

和ros\_control差不多的格式

package.xml添加描述:

<export>

<!-- Other tools can request additional information be placed here -->

<hardware\_interface plugin="${prefix}/myrobots\_hw\_plugin.xml"/>//注意区分,不要写错了插件类型

<controller\_interface plugin="${prefix}/myrobots\_controller.xml"/>

</export>

整个文件:

<?xml version="1.0"?>

<package format="2">

<name>robot\_hw\_test</name>

<version>0.0.0</version>

<description>The robot\_hw\_test package</description>

<!-- One maintainer tag required, multiple allowed, one person per tag -->

<!-- Example: -->

<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->

<maintainer email="wangxiao@todo.todo">wangxiao</maintainer>

<!-- One license tag required, multiple allowed, one license per tag -->

<!-- Commonly used license strings: -->

<!-- BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->

<license>TODO</license>

<!-- Url tags are optional, but multiple are allowed, one per tag -->

<!-- Optional attribute type can be: website, bugtracker, or repository -->

<!-- Example: -->

<!-- <url type="website">http://wiki.ros.org/robot\_hw\_test</url> -->

<!-- Author tags are optional, multiple are allowed, one per tag -->

<!-- Authors do not have to be maintainers, but could be -->

<!-- Example: -->

<!-- <author email="jane.doe@example.com">Jane Doe</author> -->

<!-- The \*depend tags are used to specify dependencies -->

<!-- Dependencies can be catkin packages or system dependencies -->

<!-- Examples: -->

<!-- Use depend as a shortcut for packages that are both build and exec dependencies -->

<!-- <depend>roscpp</depend> -->

<!-- Note that this is equivalent to the following: -->

<!-- <build\_depend>roscpp</build\_depend> -->

<!-- <exec\_depend>roscpp</exec\_depend> -->

<!-- Use build\_depend for packages you need at compile time: -->

<!-- <build\_depend>message\_generation</build\_depend> -->

<!-- Use build\_export\_depend for packages you need in order to build against this package: -->

<!-- <build\_export\_depend>message\_generation</build\_export\_depend> -->

<!-- Use buildtool\_depend for build tool packages: -->

<!-- <buildtool\_depend>catkin</buildtool\_depend> -->

<!-- Use exec\_depend for packages you need at runtime: -->

<!-- <exec\_depend>message\_runtime</exec\_depend> -->

<!-- Use test\_depend for packages you need only for testing: -->

<!-- <test\_depend>gtest</test\_depend> -->

<!-- Use doc\_depend for packages you need only for building documentation: -->

<!-- <doc\_depend>doxygen</doc\_depend> -->

<buildtool\_depend>catkin</buildtool\_depend>

<build\_depend>roscpp</build\_depend>

<build\_depend>rospy</build\_depend>

<build\_depend>std\_msgs</build\_depend>

<build\_export\_depend>roscpp</build\_export\_depend>

<build\_export\_depend>rospy</build\_export\_depend>

<build\_export\_depend>std\_msgs</build\_export\_depend>

<exec\_depend>roscpp</exec\_depend>

<exec\_depend>rospy</exec\_depend>

<exec\_depend>std\_msgs</exec\_depend>

<build\_depend>controller\_manager</build\_depend>

<build\_export\_depend>controller\_manager</build\_export\_depend>

<exec\_depend>controller\_manager</exec\_depend>

<build\_depend>hardware\_interface</build\_depend>

<build\_export\_depend>hardware\_interface</build\_export\_depend>

<exec\_depend>hardware\_interface</exec\_depend>

<build\_depend>hardware\_interface</build\_depend>

<build\_export\_depend>hardware\_interface</build\_export\_depend>

<exec\_depend>hardware\_interface</exec\_depend>

<build\_depend>controller\_manager</build\_depend>

<build\_depend>joint\_state\_controller</build\_depend>

<build\_depend>robot\_state\_publisher</build\_depend>

<build\_depend>roscpp</build\_depend>

<build\_depend>rospy</build\_depend>

<build\_export\_depend>controller\_manager</build\_export\_depend>

<build\_export\_depend>joint\_state\_controller</build\_export\_depend>

<build\_export\_depend>robot\_state\_publisher</build\_export\_depend>

<build\_export\_depend>roscpp</build\_export\_depend>

<build\_export\_depend>rospy</build\_export\_depend>

<exec\_depend>controller\_manager</exec\_depend>

<exec\_depend>joint\_state\_controller</exec\_depend>

<exec\_depend>robot\_state\_publisher</exec\_depend>

<exec\_depend>roscpp</exec\_depend>

<exec\_depend>rospy</exec\_depend>

<build\_depend>pluginlib</build\_depend>

<build\_export\_depend>pluginlib</build\_export\_depend>

<exec\_depend>pluginlib</exec\_depend>

<build\_depend>controller\_interface</build\_depend>

<build\_export\_depend>controller\_interface</build\_export\_depend>

<exec\_depend>controller\_interface</exec\_depend>

<build\_depend>hardware\_interface</build\_depend>

<build\_export\_depend>hardware\_interface</build\_export\_depend>

<exec\_depend>hardware\_interface</exec\_depend>

<!-- The export tag contains other, unspecified, tags -->

<export>

<!-- Other tools can request additional information be placed here -->

<hardware\_interface plugin="${prefix}/myrobots\_hw\_plugin.xml"/>

<controller\_interface plugin="${prefix}/myrobots\_controller.xml"/>

</export>

</package>

编译规则:CMakeLists.txt

将robothw添加到动态连接库中去,后面写总文件

add\_library(MyRobotInterface

src/robot\_hw\_test.cpp

)

target\_link\_libraries(MyRobotInterface ${catkin\_LIBRARIES})

上述两个插件都写好了,控制器可以用controller\_manager去启动,直接可以在launch文件中启动.

但是robot\_hw不能这样子启动.所以我们需要写一个节点专门启动robot\_hw

hw\_interface\_node.cpp

内容如下:

#include <ros/ros.h>

#include <ros/callback\_queue.h>

#include <iostream>

#include <combined\_robot\_hw/combined\_robot\_hw.h>

#include <controller\_manager/controller\_manager.h>

#include "robot\_hw\_test.h"

//添加的头文件们我就不解释了

int main(int argc, char \* argv[])

{

ros::init(argc, argv, "hw\_interface\_node");

//节点名称...

ros::AsyncSpinner spinner(1);//这个其实我一直不是很懂,有懂的小伙伴可以私心我.,如果是2的话如何开启第二个线程呢?我其实很好奇

spinner.start();//好象是线程开启

ros::NodeHandle nh;//ros句柄

robot\_hw\_test\_ns::MyRobotInterface hw;//生成我们自定义的robothw类

int ret = hw.init(nh,nh);//将我们的句柄传进去,教程就是这样写的,我就直接抄进来了..好像有一个句柄没有用到,不过也没关系,反正直接抄就行了

ROS\_INFO("hw init ret=%d",ret);//调试巴拉巴拉巴拉

controller\_manager::ControllerManager cm(&hw,nh);//生成控制器管理器,这里我是直接抄的例程,其实并不是很懂

sleep(1);

ros::Duration period(1.0/125);//设定循环周期,注意这里周期是多少,控制器的周期就会跟着变

ROS\_INFO("2joint hw run");

while (ros::ok())

{

hw.read(ros::Time::now(),period);//读取数据

cm.update(ros::Time::now(),period);//将控制器管理者管理的控制器进入update函数

hw.write(ros::Time::now(),period);//将ros\_control给的指令转发给硬件

period.sleep();

}

spinner.stop();

return 0;

}

添加编译规则:CMakeLists.txt

add\_executable(hw\_interface\_node

src/hw\_interface\_node.cpp

)

target\_link\_libraries(hw\_interface\_node MyRobotInterface joint\_controller\_test ${catkin\_LIBRARIES})

//将前面那俩库全都链接过来

下面我把整个CMakeLists.txt粘贴过来

cmake\_minimum\_required(VERSION 3.0.2)

project(robot\_hw\_test)

find\_package(catkin REQUIRED COMPONENTS

controller\_manager

joint\_state\_controller

robot\_state\_publisher

roscpp

rospy

pluginlib

controller\_interface

hardware\_interface

)

add\_compile\_options(-std=c++17)

catkin\_package(

INCLUDE\_DIRS include

# LIBRARIES robot\_hw\_test

CATKIN\_DEPENDS controller\_manager joint\_state\_controller robot\_state\_publisher roscpp rospy pluginlib controller\_interface hardware\_interface

# DEPENDS system\_lib

)

include\_directories(

include

${catkin\_INCLUDE\_DIRS}

)

add\_library(MyRobotInterface

src/robot\_hw\_test.cpp

)

target\_link\_libraries(MyRobotInterface ${catkin\_LIBRARIES})

add\_library(joint\_controller\_test

src/joint\_controller\_test.cpp

)

target\_link\_libraries(joint\_controller\_test ${catkin\_LIBRARIES})

add\_executable(hw\_interface\_node

src/hw\_interface\_node.cpp

)

target\_link\_libraries(hw\_interface\_node MyRobotInterface joint\_controller\_test ${catkin\_LIBRARIES})

还有一个参数服务器,yaml文件:

hardware.yaml

mylink:

robot\_hw\_test:

type: robot\_hw\_test\_ns/MyRobotInterface

joints:

- joint\_1

- joint\_2

# robothw的参数

joint\_controller\_test:

type: robot\_hw\_test\_ns/joint\_controller\_test

joint:

- joint\_1

- joint\_2

# roscontrol的参数

下面编写launch文件

其中:controller\_only.launch 是在测试过程中想看看这个roscontrol写的是不是有问题,和上一节写的一毛一样,因为需要用到上一节的功能包,所以,我就不写出来了,达到的效果就是跟gazebo仿真环境交互,能读取到gazebo发挥来的信息,仅此而已

hw\_test.launch是我们这一节需要编写的文件

<launch>

<rosparam file="$(find robot\_hw\_test)/config/hardware.yaml" command="load"/>

<!-- 加载参数服务器 -->

<node name="hw\_interface\_node" pkg="robot\_hw\_test" type="hw\_interface\_node" output="screen"/>

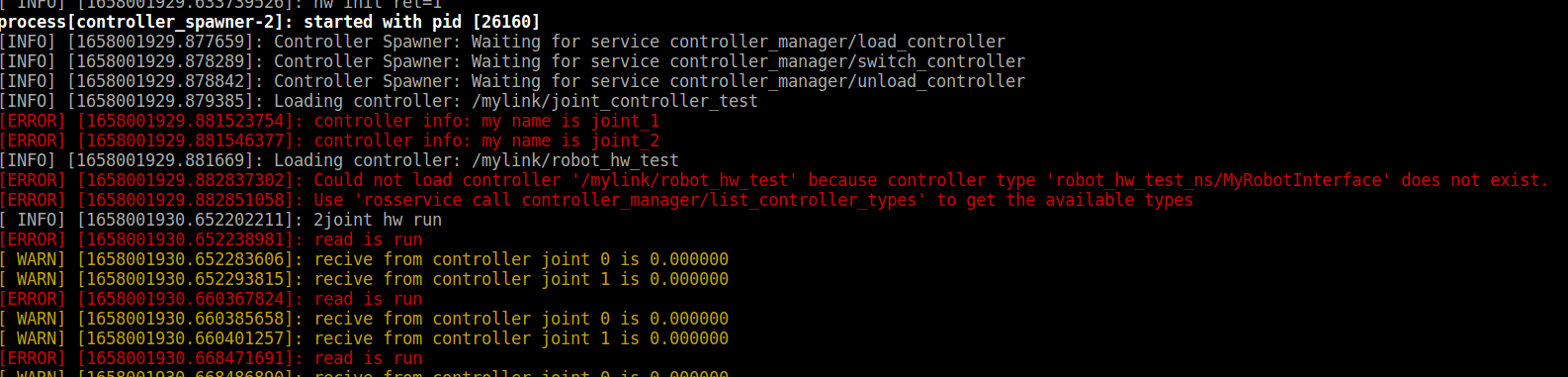
<!-- 启动robothw节点 -->

<node name="controller\_spawner" pkg="controller\_manager" type="spawner" respawn="false" output="screen" ns="/"

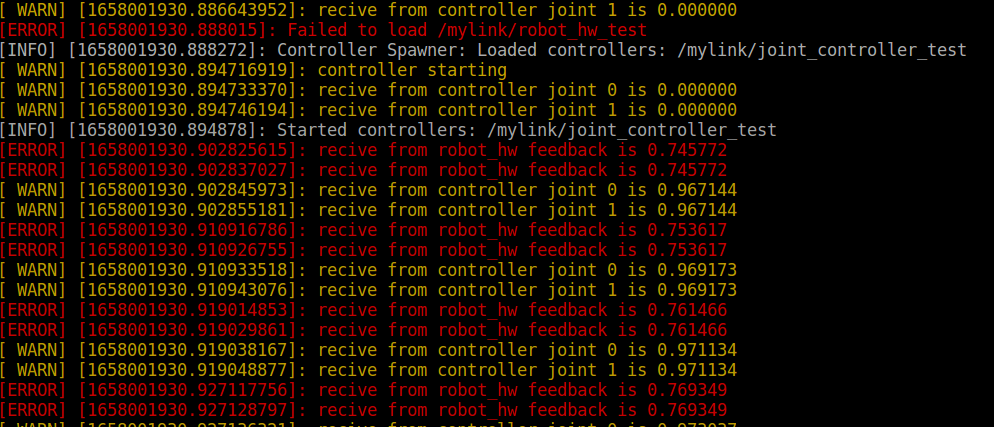
args="/mylink/joint\_controller\_test /mylink/robot\_hw\_test"/>

<!--启动控制器,这里有个很奇怪的事情,robothw理论上不能用controller\_manager启动,事实也证明会报错,说robot\_hw\_test启动失败,但是如果不加这一行,接口就连接不上,我也不知道为什么,robothw是在节点中启动的,所以在这里启动失败没关系 -->

</launch>

这里写的肯定有不规范的地方

着样子启动会报错,但是能正常运行



其中很多roserror是我自己写的,因红色显眼,所以把它当输出永乐,但是这里说

Could not load controller '/mylink/robot\_hw\_test' because controller type 'robot\_hw\_test\_ns/MyRobotInterface' does not exist.

[ERROR] [1658001929.882851058]: Use 'rosservice call controller\_manager/list\_controller\_types' to get the available types

也是意料之中,因为robot\_hw\_test是一个硬件抽象层接口,并不是一个控制器接口,所以并不能使用控制器加载..但是有一个很有趣的事情就是,如果你把robot\_hw\_test从roslaunch文件中去掉,那么^^^^^^^^^好吧我试了一下,不会有任何bug,,,好吧更新一下

<launch>

<rosparam file="$(find robot\_hw\_test)/config/hardware.yaml" command="load"/>

<!-- 加载参数服务器 -->

<node name="hw\_interface\_node" pkg="robot\_hw\_test" type="hw\_interface\_node" output="screen"/>

<!-- 启动robothw节点 -->

<node name="controller\_spawner" pkg="controller\_manager" type="spawner" respawn="false" output="screen" ns="/"

args="/mylink/joint\_controller\_test"/>

</launch>

所以,,,好吧其实不是人家的东西奇怪,我可能之前没有掌握方法…

至此,整个robot\_hw的编写就结束了,上述代码可以运行,可以直接用.累死我了,回去睡觉...