1.3 ROS 编程基础

一、 回顾入门编程，编写一个订阅器一个发布器

```
//发布器的主函数:
int main(int argc, char **argv) {
	ros::init(argc, argv, "minimal_publisher");
	//此节点的名称将为“minimal_publisher”

	ros::NodeHandle n;
	//创建一个 ros NodeHandle 对象，可以这个对象可以建立节点间的网络通信

	ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1", 1);
	//实例化一个 ros：：Publisher 的对象
	//"topic1" 话题名
	//参数“1”表示，缓存队列大小为 1，这个值可以更大。比如：1000，在这种情况下，
如果我们发布得太快，在开始丢弃旧消息之前，它最多会缓冲 1000 条消息

	std_msgs::Float64 input_float; //创建一个类型为"Float64"的变量
	// 消息类型定义在: /opt/ros/melodic/share/std_msgs
	// 任何一个在 ROS 上发布的话题数据，都需要预定一个消息类型
	// 根据这个消息类型，订阅器可以知道如何去解码传输过来的数据

	input_float.data = 0.0;

	ros::Rate naptime(1.0);
	//创建一个 ros：：Rate 对象，并设置频率值
	//将 sleep timer 设置为 1Hz 的频率（arg 以 Hz 为单位）

	//主循环
	while (ros::ok())
	{
		input_float.data = input_float.data + 0.001; //每个循环增加 0.001
		my_publisher_object.publish(input_float); // 发布数据到话题 topic1 上
		naptime.sleep();//使循环达到指定的频率
	}
}
```

```
//订阅器节点的主要函数代码:
void myCallback(const std_msgs::Float64& message_holder)
{
  //真正的工作是在这个回调函数中完成的
  //每次在“topic1”上发布新消息时，它都会被唤醒
  //此函数不消耗轮询新数据的 CPU 时间
  //函数 ROS_INFO()类似于 printf（）函数
  //它将输出发布到默认的 rosout 主题，该主题防止为 display 调用减慢此函数的速度
```

```
    //可用于查看和记录的数据
    ROS_INFO("received value is: %f",message_holder.data);

}

int main(int argc, char **argv)
{
    ros::init(argc,argv,"minimal_subscriber"); //此节点的名称将为"minimal_subscriber"
    ros::NodeHandle n;
    //创建一个 ros NodeHandle 对象, 可以这个对象可以建立节点间的网络通信

    ros::Subscriber my_subscriber_object= n.subscribe("topic1",1,myCallback);
    //实例化一个 ros::Subscriber 对象
    // "topic1"为所订阅的消息话题名
    // “1”为缓存的队列大小, 如果回调函数无法跟上发布数据的频率, 数据就需要排队,
如果回调函数不能与发布保持一致, 消息就会被新消息覆盖而丢失。
    // 每当有新消息发布到 topic1 时, “myCallback”函数就会被唤醒
    // 在回调函数中写实际的功能性代码

    ros::spin(); //这本质上是一个“while (1) ”语句, 它在新数据到达时强制刷新唤醒回调函
数, 更新话题中的消息数据, 主程序本质上在这里被挂起。

    return 0; //不应该到达这一行, 除非 roscore 崩溃
}
```

二、最小仿真器和最小控制器

```
//最小仿真器
#include<ros/ros.h>
#include<std_msgs/Float64.h>
std_msgs::Float64 g_velocity; //声明全局变量
std_msgs::Float64 g_force;//声明全局变量

void myCallback(const std_msgs::Float64& message_holder) {
        ROS_INFO("received force value is: %f", message_holder.data);//打印 force value 的值
        g_force.data = message_holder.data; // 通过全局变量接收消息数据, 供主函数使用
}

int main(int argc, char **argv) {
        ros::init(argc, argv, "minimal_simulator");
        ros::NodeHandle nh;

        ros::Subscriber my_subscriber_object = nh.subscribe("force_cmd", 1, myCallback);
        ros::Publisher my_publisher_object = nh.advertise<std_msgs::Float64>("velocity", 1);

```

```
	double mass = 1.0;//定义质量为 1kg
	double dt = 0.01; //定义积分的时间间隔为 0.01s
	double sample_rate = 1.0 / dt; //计算更新频率
	ros::Rate naptime(sample_rate);//实例化更新频率对象
	g_velocity.data = 0.0; //初始化速度为 0m/s
	g_force.data = 0.0; // 初始化力的值为 0.0N，是一个全局变量，会随着回调函数而更
新
	while (ros::ok()) {
		g_velocity.data = g_velocity.data + (g_force.data / mass) * dt;
		//加速度的欧拉积分

		my_publisher_object.publish(g_velocity); // 发布系统状态，速度值
		ROS_INFO("velocity = %f", g_velocity.data);
		ros::spinOnce(); //允许从回调更新数据
		naptime.sleep();
		//等待指定时间的剩余时间；这个循环速度比
		//指定控制器的更新率为 10hz
		//然而，不管怎样，模拟器必须每 10 毫秒前进一次

	}
	return 0; // should never get here, unless roscore dies
}
```

```
// 最小控制器节点:
#include<ros/ros.h>
#include<std_msgs/Float64.h>
//全局变量
std_msgs::Float64 g_velocity;
std_msgs::Float64 g_vel_cmd;
std_msgs::Float64 g_force;

void myCallbackVelocity(const std_msgs::Float64& message_holder) {
	ROS_INFO("received velocity value is: %f", message_holder.data);
	g_velocity.data = message_holder.data; // 通过全局变量接收消息数据，供主函数使
用
}

void myCallbackVelCmd(const std_msgs::Float64& message_holder) {
	ROS_INFO("received velocity command value is: %f", message_holder.data);
	g_vel_cmd.data = message_holder.data; // 通过全局变量接收消息数据，供主函数使
用
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "minimal_controller");
    ros::NodeHandle nh;

    ros::Subscriber my_subscriber_object1 = nh.subscribe("velocity", 1,
myCallbackVelocity);
    ros::Subscriber my_subscriber_object2 = nh.subscribe("vel_cmd", 1,
myCallbackVelCmd);

    ros::Publisher my_publisher_object = nh.advertise<std_msgs::Float64>("force_cmd",
1);
    double Kv = 1.0; // 速度控制的比例值
    double dt_controller = 0.1;
    double sample_rate = 1.0 / dt_controller; // 计算更新频率为 10hz
    ros::Rate naptime(sample_rate); //

    g_velocity.data = 0.0; //初始化速度值为 0m/s
    g_force.data = 0.0; //初始化的力为 0N，会随着回调函数取回的消息值而修改
    g_vel_cmd.data = 0.0; // 初始化速度命令为 0m/s
    double vel_err = 0.0; // 速度误差值
    /
    while (ros::ok()) {
        vel_err = g_vel_cmd.data - g_velocity.data; // 计算速度误差值
        g_force.data = Kv*vel_err; //只用速度误差的比例控制得到给控制器的力的大小
        my_publisher_object.publish(g_force); //发布力的反馈值
        ROS_INFO("force command = %f", g_force.data);
        ros::spinOnce(); //从回调函数中取值
        naptime.sleep(); // 等待按照频率设置的时间周期
    }
    return 0; // 不会运行到这一行，除非中心节点崩溃
}
```

三、ros 中使用类编程，重写之前的控制器

```
// minimal_controller_class.h 的内容

#ifndef MINIMAL_CONTROLLER_CLASS_H_
#define MINIMAL_CONTROLLER_CLASS_H_

//some generically useful stuff to include...

#include <ros/ros.h> //ALWAYS need to include this
//message types used in this example code;  include more message types, as needed
#include<std_msgs/Float64.h>
```

```
// define a class, including a constructor, member variables and member functions
class ExampleRosClass
{
public:
    ExampleRosClass(ros::NodeHandle* nodehandle); //"main" will need to instantiate a
ROS nodehandle, then pass it to the constructor
    // may choose to define public methods or public variables, if desired
    std_msgs::Float64 g_velocity;
    std_msgs::Float64 g_vel_cmd;
    std_msgs::Float64 g_force; // this one does not need to be global...
    ros::Subscriber my_subscriber_object1; //these will be set up within the class
constructor, hiding these ugly details
    ros::Subscriber my_subscriber_object2;
    ros::Publisher   my_publisher_object;
private:
    // put private member data here;   "private" data will only be available to member
functions of this class;
    ros::NodeHandle nh_; // we will need this, to pass between "main" and constructor
    // some objects to support subscriber, service, and publisher



    // member methods as well:
    void initializeSubscribers(); // we will define some helper methods to encapsulate the
gory details of initializing subscribers, publishers and services
    void initializePublishers();

    void velocity_Callback(const std_msgs::Float64& message_holder);
    void vel_cmd_Callback(const std_msgs::Float64& message_holder); //prototype for
callback of example subscriber
}; // note: a class definition requires a semicolon at the end of the definition

#endif   // this closes the header-include trick...ALWAYS need one of these to match
#ifndef
```

```
// minimal_controller_class.cpp 的内容

//#include<ros/ros.h>
//#include<std_msgs/Float64.h>
#include<minimal_controller_class/minimal_controller_class.h>

void ExampleRosClass:: velocity_Callback(const std_msgs::Float64& message_holder) {
    // check for data on topic "velocity"
    ROS_INFO("received velocity value is: %f", message_holder.data);
```

```
        g_velocity.data = message_holder.data; // post the received data in a global var for
access by
        //main prog.
}

void ExampleRosClass:: vel_cmd_Callback(const std_msgs::Float64& message_holder) {
        // check for data on topic "vel_cmd"
        ROS_INFO("received velocity command value is: %f", message_holder.data);
        g_vel_cmd.data = message_holder.data; // post the received data in a global var for
access by
        //main prog.
}

//
ExampleRosClass::ExampleRosClass(ros::NodeHandle* nodehandle):nh_(*nodehandle)
{
        ROS_INFO("in class constructor of ExampleRosClass");
        initializeSubscribers(); // package up the messy work of creating subscribers; do this
overhead in constructor
        initializePublishers();

        //initialize variables here, as needed
        g_velocity.data = 0.0; //initialize velocity to zero
        g_force.data = 0.0; // initialize force to 0; will get updated by callback
        g_vel_cmd.data = 0.0; // init velocity command to zero
}

void ExampleRosClass::initializeSubscribers()
{
        ROS_INFO("Initializing Subscribers");
;
        // add more subscribers here, as needed
        my_subscriber_object1 = nh_.subscribe("velocity",
1,&ExampleRosClass::velocity_Callback,this);
        //关键字 this 告诉编译器正在引用此类的当前实例
        my_subscriber_object2 = nh_.subscribe("vel_cmd", 1,
&ExampleRosClass::vel_cmd_Callback,this);

}

//member helper function to set up publishers;
void ExampleRosClass::initializePublishers()
{
        ROS_INFO("Initializing Publishers");
```

```
	my_publisher_object = nh_.advertise<std_msgs::Float64>("force_cmd", 1, true);
	//add more publishers, as needed
}


int main(int argc, char **argv)
{
	ros::init(argc, argv, "minimal_controller_class"); //name this node
	// when this compiled code is run, ROS will recognize it as a node called
"minimal_controller"
	ros::NodeHandle nh; // node handle

	ExampleRosClass exampleRosClass(&nh);
	//实例化 ExampleRosClass 对象并传入指向 nodehandle 的指针，以便构造函数使用
	//main()必须实例化构造函数使用的节点句柄，并且需要将该值传递给构造函数
	//但是构造函数本身就是负责初始化变量的，因此这就会产生一个鸡蛋相生的问题。
	//通过：ExampleRosClass::ExampleRosClass(ros::NodeHandle*
nodehandle):nh_(*nodehandle)来解决

	double Kv = 1.0; // velocity feedback gain
	double dt_controller = 0.1; //specify 10Hz controller sample rate (pretty slow, but
	//illustrative)
	double sample_rate = 1.0 / dt_controller; // compute the corresponding update
frequency
	ros::Rate naptime(sample_rate); // use to regulate loop rate
	double vel_err = 0.0; // velocity error

	while (ros::ok()) {
		vel_err = exampleRosClass.g_vel_cmd.data - exampleRosClass.g_velocity.data; //
		exampleRosClass.g_force.data = Kv*vel_err;
		exampleRosClass.my_publisher_object.publish(exampleRosClass.g_force);
		ROS_INFO("force command = %f", exampleRosClass.g_force.data);
		ros::spinOnce(); //allow data update from callback;
		naptime.sleep(); // wait for remainder of specified period;
	}


	return 0; // should never get here, unless roscore dies
}
```