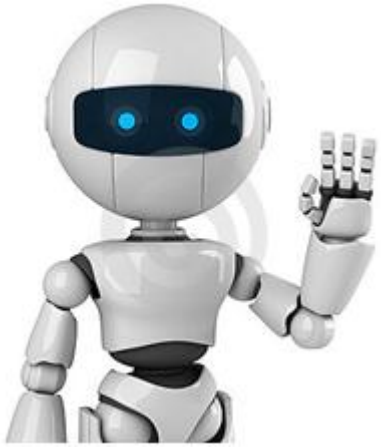
# 8. 导航功能—机器人设置



东北大学 张云洲

# 内 容 提 纲

8.1 ROS导航功能包集

8.2 创建变换

8.3 发布传感器信息

8.4 发布里程计信息

8.5 创建基础控制器

8.6 使用ROS创建地图

# 8.1 ROS导航功能包集

在使用导航功能包集之前，机器人还需要满足一些条件：

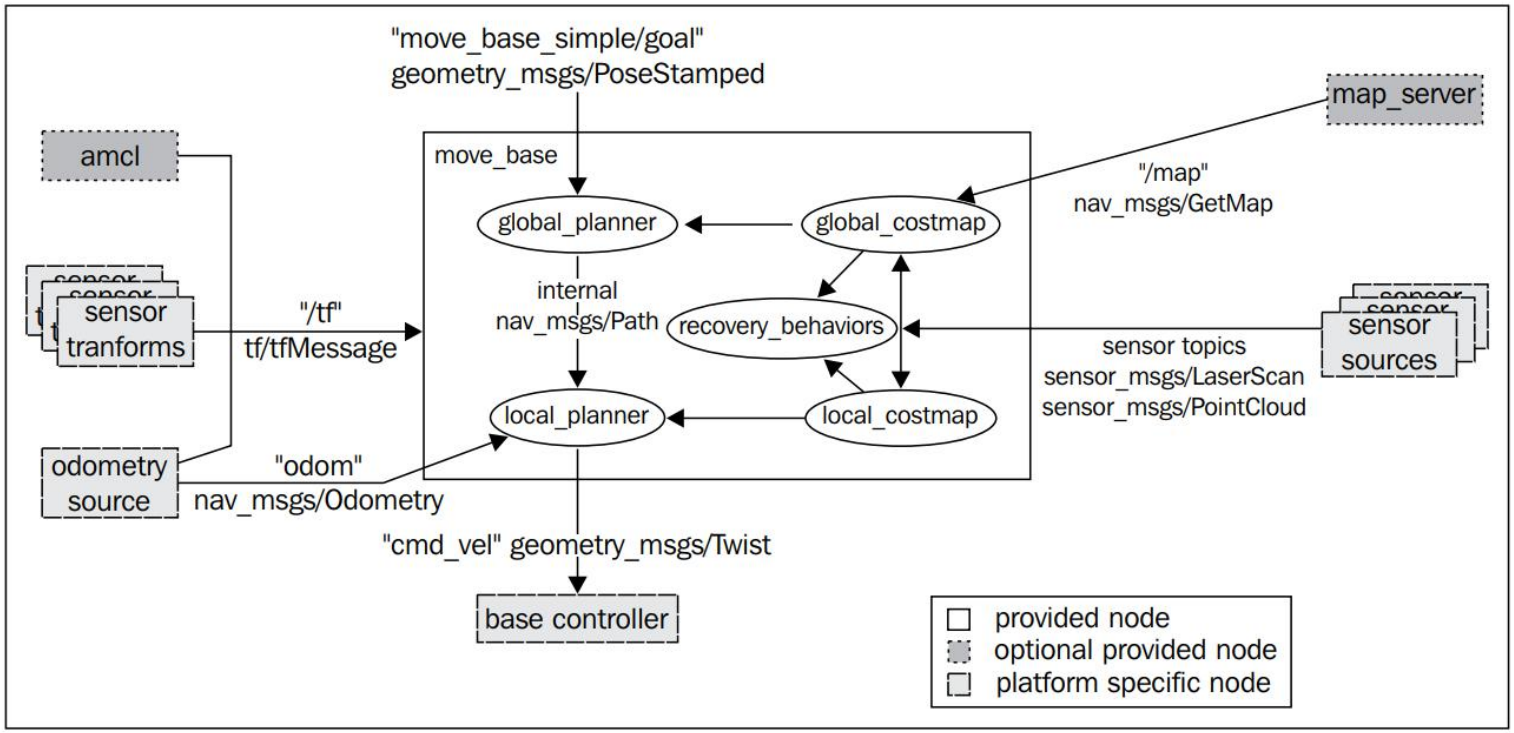
- 只能处理双轮差分驱动和完全轮驱动的机器人。双足机器人可以只用来定位，而不横向移动机器人；
- 机器人能发布关于所有关节和传感器位置关系的信息；
- 机器人能发送线速度和角速度信息；
- 机器人必须有一个平面激光雷达来完成地图构建和定位，或者能生成一些相当于激光雷达或者声呐的数据。

**导航功能包集的组织方式：**

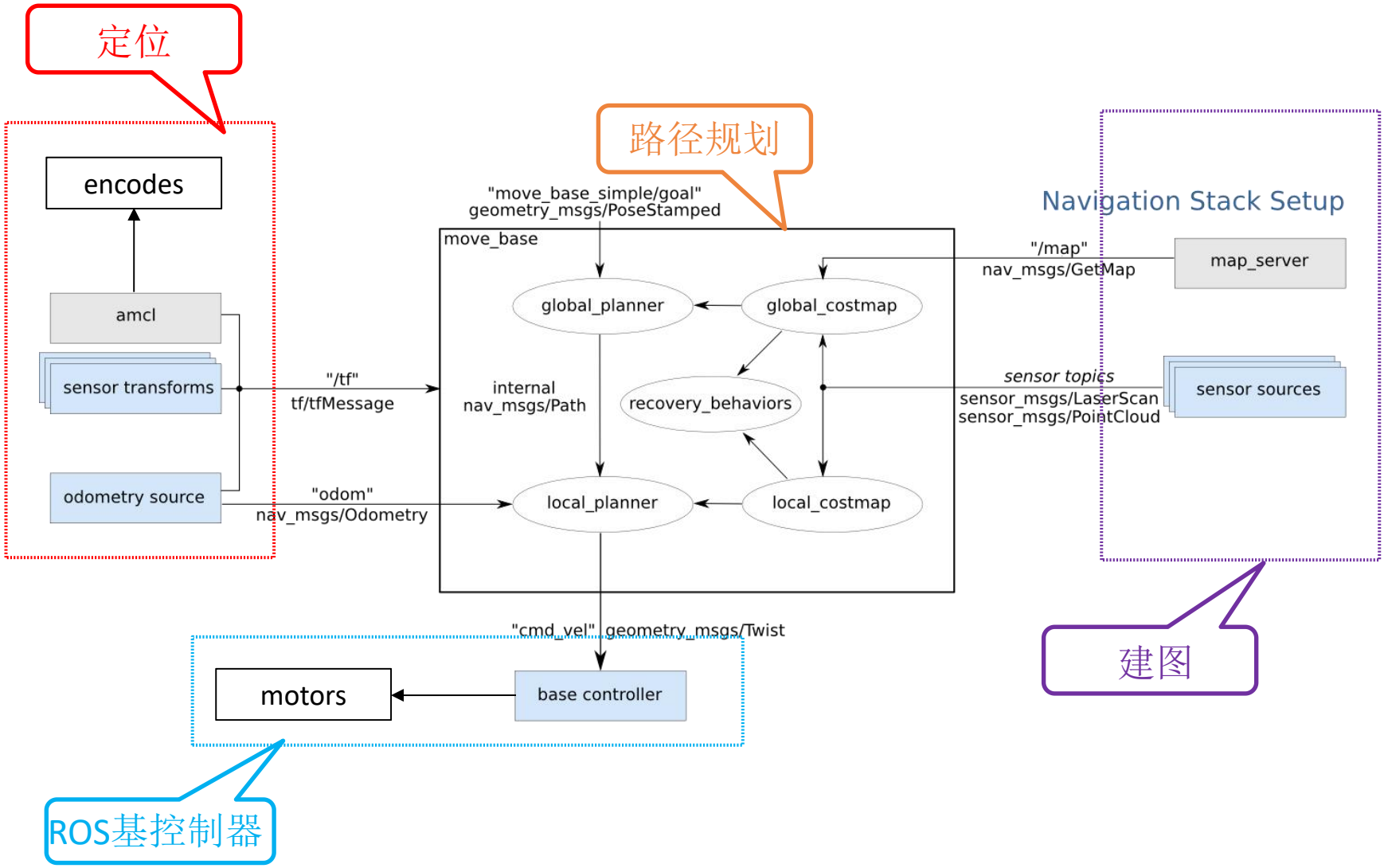
白色框：ROS的集成功能包，可以实现机器人自主导航；

灰色框：可选择的节点，不是通用的；

黑色框：平台特殊的节点。



# 运动控制的分层




运动控制分层示意图

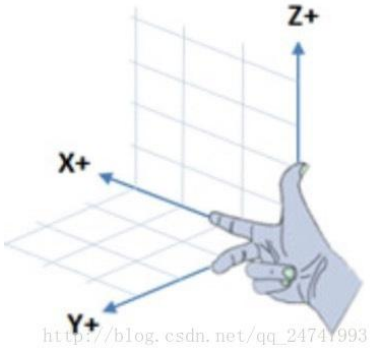
# 8.2 创建变换

•导航功能包集需要知道传感器、轮子和关节的位置。

•使用TF（Transform Frame）软件库来管理坐标转换树。

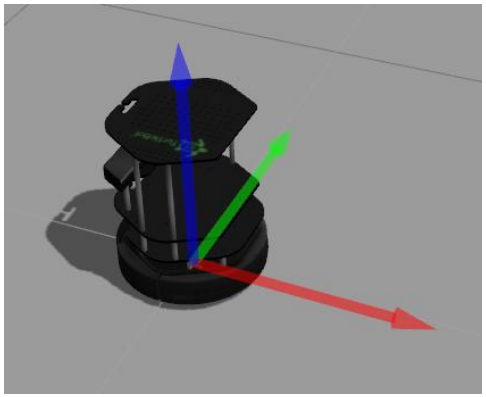
在ROS机器人控制和导航中，所有的坐标系都以**右手笛卡尔坐标系**定义。常见的坐标系有：世界坐标系、机体坐标系、里程计坐标系等。

ROS系统使用公制作为计量系统，线速度通常使用米/秒(m/s)来作为单位，角速度通常使用弧度/秒(rad/s)来作为单位。




x轴: 机器人正前方，y轴: 正左方，z轴: 正上方。

右手笛卡尔坐标系



Turtlebot机体坐标系

# TF 树的定义与构造

**TF树**[1]是一个让用户随时间跟踪多个参考系的功能包，它使用一种树型数据结构，根据时间缓冲并维护多个参考系之间的坐标变换关系。

■TF 树的作用

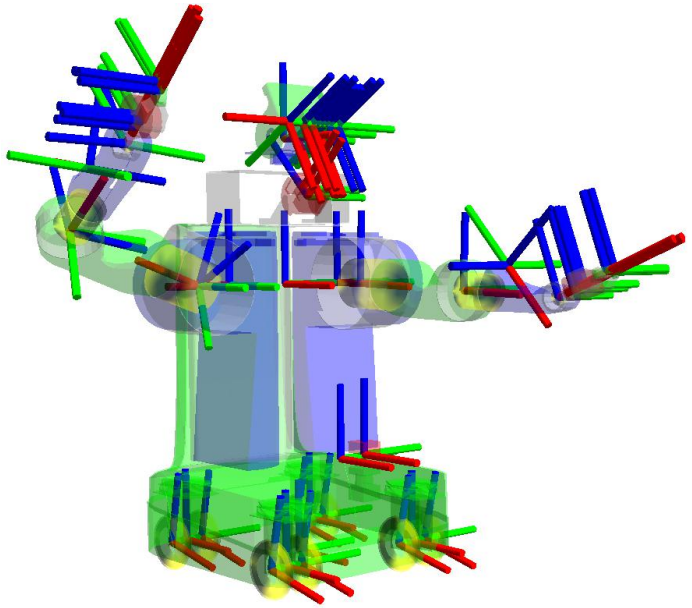
连接不同坐标系或者机器人关节

定义不同坐标系之间的转换关系

■构造

硬件：使用urdf文件

软件：使用ROS的TF广播函数



TF连接关系示意图

[1] Foote T. Tf: The transform library[C]// IEEE International Conference on Technologies for Practical Robot Applications. IEEE, 2013:1-6.

# TF 树可以做什么?

一个机器人系统通常有很多三维的参考系，而且会随着时间的推移发生变化，例如全局参考系（world frame），机器人中心参考系（base frame），机械夹参考系（gripper frame），机器人头参考系（head frame）等等。

**TF可以以时间为轴，跟踪这些参考系（默认是10秒之内的），**并且允许用户提出如下的申请：

- 五秒钟之前，机器人头参考系相对于全局参考系的关系是什么样的？
- 机器人夹取的物体相对于机器人中心参考系的位置在哪里？
- 机器人中心参考系相对于全局参考系的位置在哪里？

**TF可以在分布式系统中进行操作，即：一个机器人系统中所有的参考系变换关系，对于所有节点组件都是可用的。**所有订阅tf消息的节点都会缓冲一份所有参考系的变换关系数据，因此这种结构不需要中心服务器来存储任何数据。

http://www.guyuehome.com/279

# TF 树的使用流程

使用tf功能包，总体上分为以下两个步骤：

**（1） 监听tf变换**

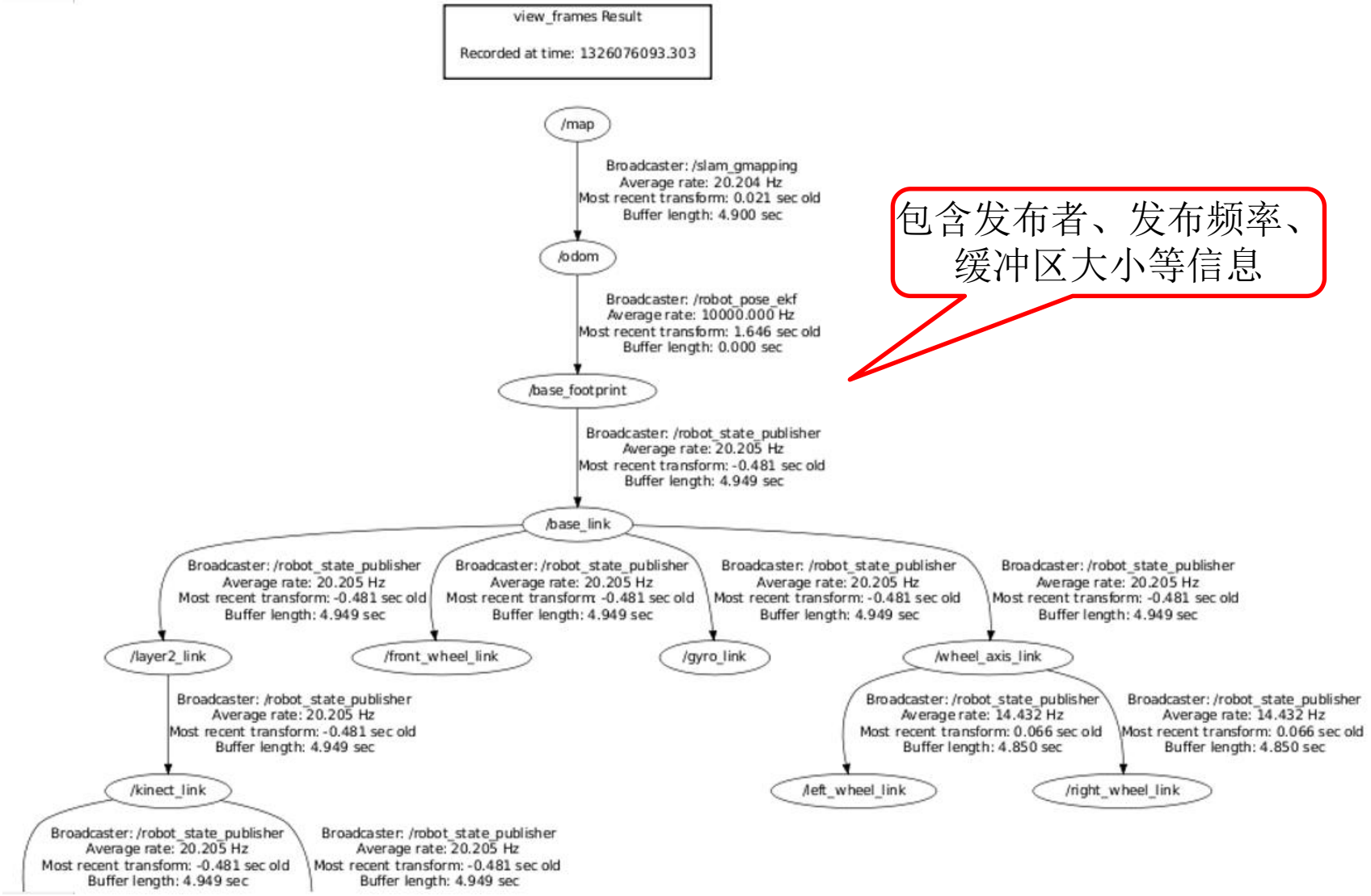
接收并缓存系统中发布的所有参考系变换，并从中查询所需要的参考系变换。

**（2） 广播tf变换**

向系统中广播参考系之间的坐标变换关系。系统中更可能会存在多个不同部分的tf变换广播，每个广播都可以直接将参考系变换关系直接插入tf树中，不需要再进行同步。

http://www.guyuehome.com/279

# TF 树示意图


view_frames Result
Recorded at time: 1326076093.303
/map
Broadcaster: /slam_gmapping
Average rate: 20.204 Hz
Most recent transform: 0.021 sec old
Buffer length: 4.900 sec
/odom
Broadcaster: /robot_pose_ekf
Average rate: 10000.000 Hz
Most recent transform: 1.646 sec old
Buffer length: 0.000 sec
/base_footprint
Broadcaster: /robot_state_publisher
Average rate: 20.205 Hz
Most recent transform: -0.481 sec old
Buffer length: 4.949 sec
/base_link
Broadcaster: /robot_state_publisher
Average rate: 20.205 Hz
Most recent transform: -0.481 sec old
Buffer length: 4.949 sec
/layer2_link
/front_wheel_link
/gyro_link
/wheel_axis_link
Broadcaster: /robot_state_publisher
Average rate: 20.205 Hz
Most recent transform: -0.481 sec old
Buffer length: 4.949 sec
/kinect_link
Broadcaster: /robot_state_publisher
Average rate: 14.432 Hz
Most recent transform: 0.066 sec old
Buffer length: 4.850 sec
/left_wheel_link
/right_wheel_link
包含发布者、发布频率、
缓冲区大小等信息

# TF 树的广播

TF变换的广播可以通过多种方式定义，在定义某个TF变换的时候要包含：**变换关系、时间戳、父坐标系、子坐标系**四个关键要素。

■C++

```
ros::Rate r(100);
tf::TransformBroadcaster broadcaster;
while(n.ok())
{
    broadcaster.sendTransform(
    tf::StampedTransform(
    tf::Transform(tf::Quaternion(0, 0, 0, 1),
    tf::Vector3(0.1, 0.0, 0.2)),
    ros::Time::now(),"base_link", "base_laser"));
    r.sleep();
}
```

■Launch文件

```
<launch>
    <node pkg="tf" type="static_transform_publisher" name="map"
          args="0 0 0.25 -1.57 0 -1.57 /base_link /base_laser 100" />
    </node>
</launch>
```

# TF 树的监听

在ROS中，TF树的监听是指实时获取指定两个Frame之间的变换关系，属于比较重要的一个环节。

C++示例代码

```
// Get the current Position and Yaw angular of the robot
Pose_Type Get_RobotPose()
{
    tf::TransformListener listener;
    tf::StampedTransform  transform;
    tf::Quaternion Orientation;
    tf::Vector3 Position;
    bool flag = true;
    double Roll, Pitch, Yaw;
    try
    {
        if(flag)
        {
            listener.waitForTransform("/odom", "/base_footprint", ros::Time(0), ros::Duration(3.0));
            flag = false;
        }
        listener.lookupTransform("/odom", "/base_footprint", ros::Time(0), transform);
    }
    catch (tf::TransformException ex)
    {
        ROS_ERROR("%s",ex.what());
        ros::Duration(1.0).sleep();
    }
    Position = transform.getOrigin();
    Orientation = transform.getRotation();
    tf::Matrix3x3(Orientation).getRPY(Roll, Pitch, Yaw);
    return std::make_pair(Position, Yaw);
}
```

# 查看TF 树

在调试TF变换的过程中，需要查看TF树的连接结构和变换关系，下面是四种常用来查看TF变换方法。

■tf_echo

$ rosrun tf tf_echo [frame1] [frame2]

在终端下直接打印出变换关系

■Rqt_tf_tree

$ rosrun rqt_tf_tree rqt_tf_tree

在ros的rqt中生成TF树

■rviz和tf

$ rosrun rviz rviz -d /
rospack find package_name

在Rviz下查看各Frame的相对关系

■View_Frames

$ rosrun tf view_frames
$ evince frame.pdf

将TF树保存为pdf文件

## 8.2 创建变换

### Step1:创建广播机构

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>

int main(int argc, char** argv)
{
    ros::init(argc, argv, "robot_tf_publisher");
    ros::NodeHandle n;
    ros::Rate r(100);

    tf::TransformBroadcaster broadcaster;
    while(n.ok())
    {
      broadcaster.sendTransform(
        tf::StampedTransform(
          tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.0, 0.0, 0.0)),
                    ros::Time::now(),"base_link", "base_laser"));
      r.sleep();
    }
}
```

四元数

父坐标系的名字

子坐标系的名字

五个参数：

**1.两个参考系之间的旋转变换**

以四元数存储旋转变换的参数，两个参考系没有发生旋转变换，因此倾斜角、滚动角、偏航角都是0。

**2. 坐标的位移变换**，用到的两个参考系在X轴和Z轴发生了位置，根据位移值填入到Vector3 向量中。

**3. 时间戳**，直接调用ROS的API。

**4. 父节点存储的参考系**，即base_link

**5. 子节点存储的参考系**，即base_laser。

## 8.2 创建变换

TF::stampedtransform

TF的一种特殊情况：它需要frame_id和stamp以及child_frame_id。

```
/** \brief The Stamped Transform datatype used by tf */
class StampedTransform : public tf::Transform
{
public:
  ros::Time stamp_; ///< The timestamp associated with this transform 时间戳
  std::string frame_id_; ///< The frame_id of the coordinate frame  in which this transform is defined  定义转换坐标框架的frameID
  std::string child_frame_id_; ///< The frame_id of the coordinate frame this transform defines 的坐标系变换定义的id
  StampedTransform(const tf::Transform& input, const ros::Time& timestamp, const std::string & frame_id, const std::string & child_frame_id):
    tf::Transform (input), stamp_ ( timestamp ), frame_id_ (frame_id), child_frame_id_(child_frame_id){ };

  /** \brief Default constructor only to be used for preallocation */
  StampedTransform() { };

  /** \brief Set the inherited Traonsform data */
  void setData(const tf::Transform& input){*static_cast<tf::Transform*>(this) = input;};

};
```

# 8.2 创建变换

- chapter8_tutorials/src文件夹：

  创建tf_broadcaster.cpp，将上述代码放入文件中。

- 修改CMakeLists.txt文件中以创建新的可执行文件。

```
add_executable(tf_broadcaster src/tf_broadcaster.cpp)
target_link_libraries(tf_broadcaster ${catkin_LIBRARIES})
```

# 8.2 创建变换

**Step2:创建侦听器**

chapter8_tutorials/src文件夹：
创建tf_listener.cpp，加入代码：

```
#include <ros/ros.h>
#include <geometry_msgs/PointStamped.h>
#include <tf/transform_listener.h>

void transformPoint(const tf::TransformListener& listener){
    //we'll create a point in the base_laser frame that we'd like to transform to the
base_link frame
    geometry_msgs::PointStamped laser_point;
    laser_point.header.frame_id = "base_laser";
    //we'll just use the most recent transform available for our simple example
    laser_point.header.stamp = ros::Time();
```

```
    //just an arbitrary point in space
    laser_point.point.x = 1.0;
    laser_point.point.y = 2.0;
    laser_point.point.z = 0.0;

    geometry_msgs::PointStamped base_point;
    listener.transformPoint("base_link", laser_point, base_point);

    ROS_INFO("base_laser: (%.2f, %.2f. %.2f) -----> base_link:
(%.2f, %.2f, %.2f) at time %.2f",
    laser_point.point.x, laser_point.point.y, laser_point.point.z,
    base_point.point.x, base_point.point.y, base_point.point.z,
    base_point.header.stamp.toSec());

    ROS_ERROR("Received an exception trying to transform a
point from \"base_laser\" to \"base_link\": %s", ex.what());
}
```

```
int main(int argc, char** argv)
{
    ros::init(argc, argv, "robot_tf_listener");
    ros::NodeHandle n;
    tf::TransformListener listener(ros::Duration(10));

    //we'll transform a point once every second
    ros::Timer timer = n.createTimer(ros::Duration(1.0),
        boost::bind(&transformPoint, boost::ref(listener)));

    ros::spin();
}
```

注意：需修改CMakeList.txt文件以生成可执行文件。

编译这个功能包，并在每个终端中分别通过以下命令运行这两个节点：

```
$ catkin_make
$ rosrun chapter8_tutorials tf_broadcaster
$ rosrun chapter8_tutorials tf_listener
```

※※不要忘记每次要首先运行roscore

然后可以看到如下效果：

```
[ INFO] [1368521854.336910465]: base_laser: (1.00, 2.00. 0.00) -----> 
base_link: (1.10, 2.00, 0.20) at time 1368521854.33
[ INFO] [1368521855.336347545]: base_laser: (1.00, 2.00. 0.00) -----> 
base_link: (1.10, 2.00, 0.20) at time 1368521855.33
```

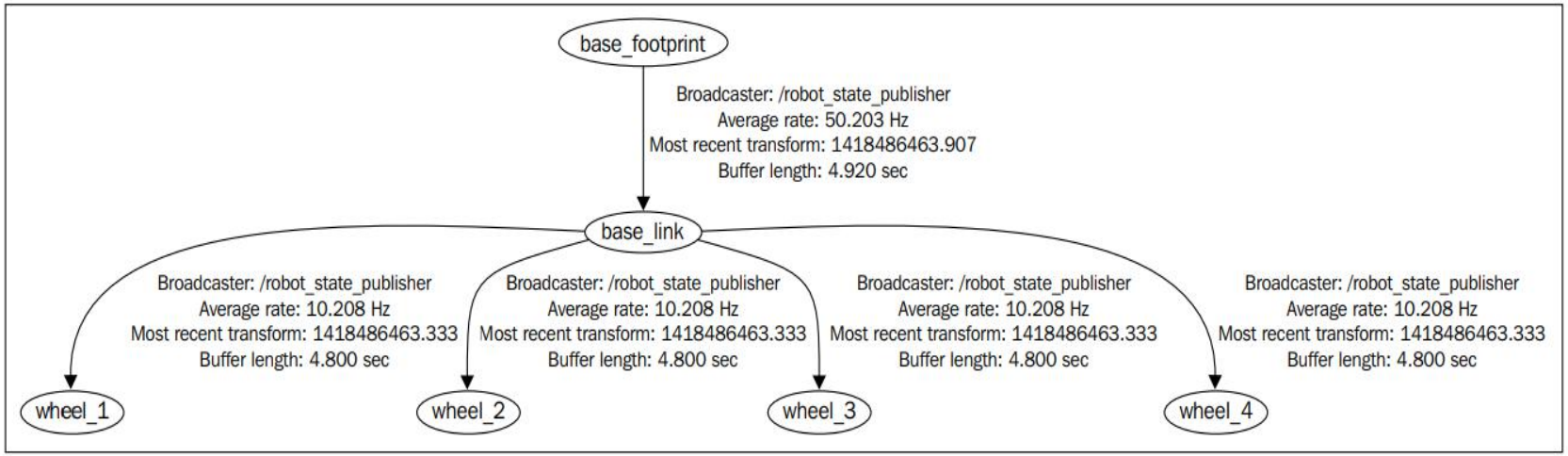
## Step3：查看坐标变换树

查看机器人的变换树，使用以下命令：

$ roslaunch chapter8_tutorials gazebo_map_robot.launch model:="`rospack find chapter8_tutorials`/urdf/robot1_base_01.xacro"

$ rosrun tf view_frames　查看：$ evince frames.pdf

坐标系结构：




$ rosrun rqt_tf_tree rqt_tf_tree

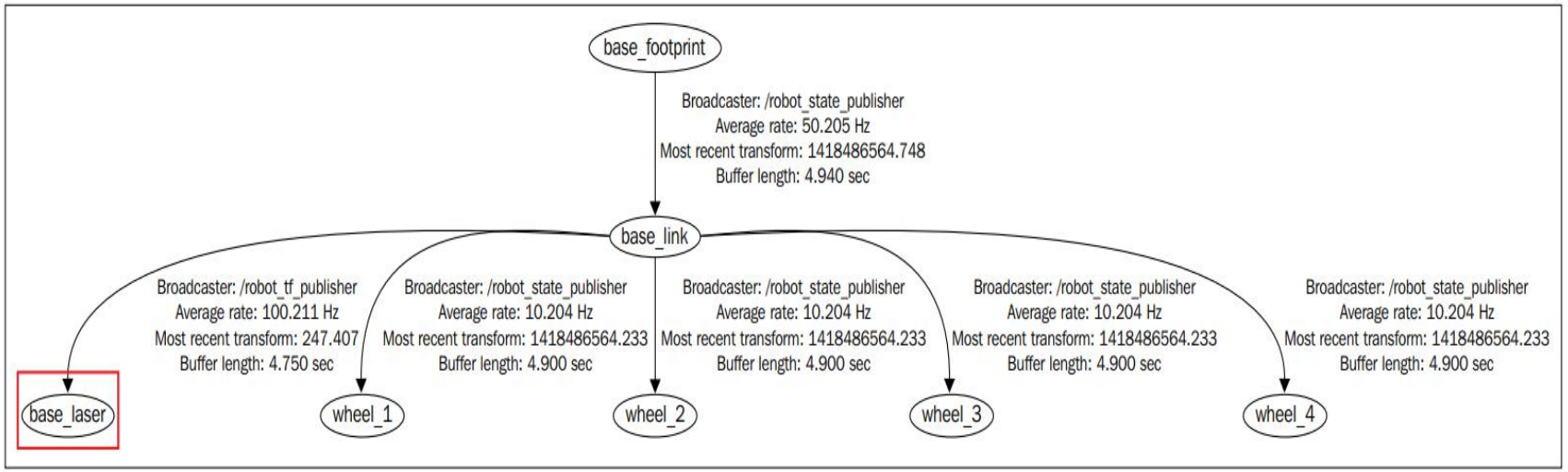
再一次运行tf_broadcaster和rosrun tf view_frames命令，

$ rosrun chapter8_tutorials tf_broadcaster

$ rosrun tf view_frames

$ evince frame.pdf

通过代码创建的坐标系：



# 8.3 发布传感器信息

导航功能包集仅支持使用平面激光雷达传感器，因此传感器必须使用以下格式发布数据：sensor_msgs/LaserScan或sensor_msgs/PointCloud2。

命令：$ rosmsg show sensor_msgs/LaserScan

输出：

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

## 创建激光雷达节点：

创建chapter8_tutorials/src/laser.cpp：

```
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>
int main(int argc, char** argv)
{
    ros::init(argc, argv, "laser_scan_publisher");
    ros::NodeHandle n;
    ros::Publisher scan_pub = n.advertise("scan", 50);

    unsigned int num_readings = 100;
    double laser_frequency = 40;
    double ranges[num_readings];
    double intensities[num_readings];
    int count = 0;
    ros::Rate r(1.0);
```

真实激光雷达？？

```
while(n.ok())
    {
        //generate some fake data for our laser scan
        for(unsigned int i = 0; i < num_readings; ++i)
        {
            ranges[i] = count;
            intensities[i] = 100 + count;
        }

        ros::Time scan_time = ros::Time::now();

        //LaserScan message
        sensor_msgs::LaserScan scan;
        scan.header.stamp = scan_time;
        scan.header.frame_id = "base_link";
        scan.angle_min = -1.57;
        scan.angle_max = 1.57
```

**frame_id的提示：**

必须是.urdf文件中已创建的坐标系之一，并且其坐标系相关数据已发布到tf坐标变换之中。

```
        scan.angle_increment = 3.14 / num_readings;
        scan.time_increment = (1 / laser_frequency) / (num_readings);
        scan.range_min = 0.0;
        scan.range_max = 100.0;
        scan.ranges.resize(num_readings);
        scan.intensities.resize(num_readings);

        for(unsigned int i = 0; i < num_readings; ++i)
        {
            scan.ranges[i] = ranges[i];
            scan.intensities[i] = intensities[i];
        }

        scan_pub.publish(scan);
        ++count;
        r.sleep();
    }
}
```

**代码解析：**

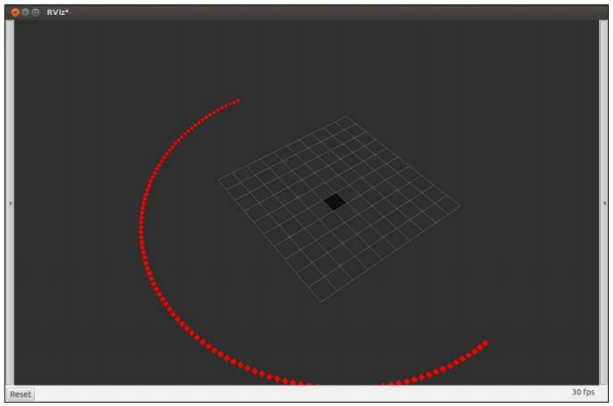
使用正确的名称创建主题：

ros::Publisher scan_pub = n.advertise<sensor_msgs::LaserScan>(“scan”,50);

需发布header、stamp和frame_id等字段的数据。为了让导航功能包集运行起 来，必须有这些数据的支持：

scan.header.stamp = scan_time;

scan.header.frame_id = "base_link";



通过该示例模板可以使用任何激光雷达，而无需考虑ROS中是否有它的驱动，甚至于将其它传感器通过数据格式转换伪装成一个激光雷达。

# 8.4 发布里程计信息

导航功能包集需要获取机器人的里程信息（机器人相对于某一点的距离）。

消息类型：nav_msgs/Odometry。

可使用指令查看消息的 数据结构：

$ rosmsg show nav_msgs/Odometry

位置

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
```

欧拉坐标系

四元数

速度

```
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

线速度

角速度

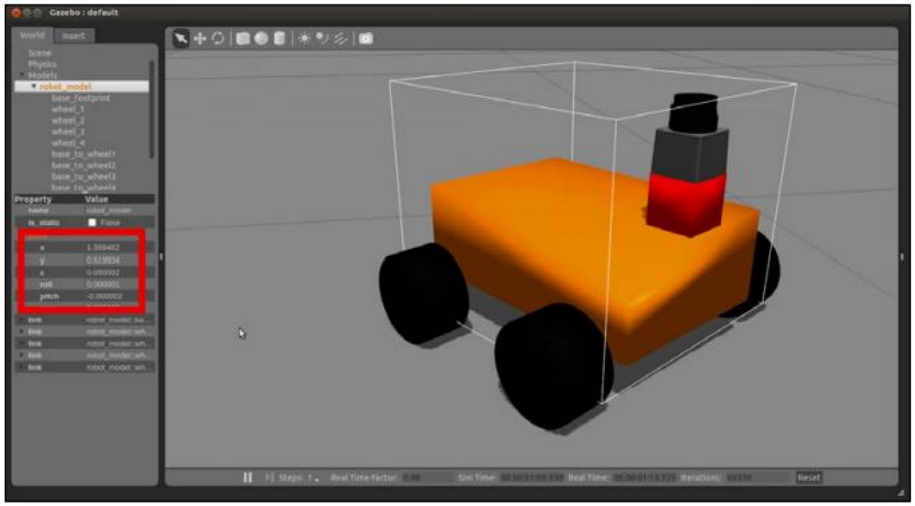
## Step1：Gazebo获取里程数据

在Gazebo中执行示例机器人并查看里程数据。输入以下命令：

$ roslaunch chapter8_tutorials gazebo_xacro.launch model:="`rospack find robot1_description`/urdf/robot1_base_04.xacro" [1]

$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py [2]

通过teleop远程操作节点，移动机器人一段时间，以便在odometry主题中生成足够多的数据。



排错：[1] https://blog.csdn.net/x_r_su/article/details/53504784

[2]https://answers.ros.org/question/199363/error-run-teleop_twist_keyboard-package/

Gazebo会不间断地发布里程数据。可以点击主题来查看具体发布的数据，也可以在命令行窗口中输入以下命令：

$ rostopic echo /odom/pose/pose

```
---
position:
  x: 1.36988769868
  y: 0.620282427846
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z:  0.28708429626
  w: 0.957905322477
---
```

**Gazebo如何获取这些数据？**

插件的源文件保存在gazebo_plugins功能包中，文件名为

gazebo_ros_skid_steer_drive.cpp

下载地址为：源码

最重要的部分是publishOdometry 函数：

```
void GazeboRosSkidSteerDrive::publishOdometry(double step_time)
{
  ros::Time current_time = ros::Time::now();
  std::string odom_frame =
  tf::resolve(tf_prefix_, odometry_frame_);
  std::string base_footprint_frame =
  tf::resolve(tf_prefix_, robot_base_frame_);
  // TODO create some non-perfect odometry!
  // getting data for base_footprint to odom transform
  math::Pose pose = this->parent->GetWorldPose();
  tf::Quaternion qt(pose.rot.x, pose.rot.y, pose.rot.z, pose.rot.w);
  tf::Vector3 vt(pose.pos.x, pose.pos.y, pose.pos.z);
  tf::Transform base_footprint_to_odom(qt, vt);
  if (this->broadcast_tf_)
  {
    transform_broadcaster_->sendTransform(
    tf::StampedTransform(base_footprint_to_odom, current_time,
    odom_frame, base_footprint_frame));
  }
  // publish odom topic
  odom_.pose.pose.position.x = pose.pos.x;
  odom_.pose.pose.position.y = pose.pos.y;
  odom_.pose.pose.orientation.x = pose.rot.x;
  odom_.pose.pose.orientation.y = pose.rot.y;
  odom_.pose.pose.orientation.z = pose.rot.z;
  odom_.pose.pose.orientation.w = pose.rot.w;
  odom_.pose.covariance[0] = 0.00001;
  odom_.pose.covariance[7] = 0.00001;
  odom_.pose.covariance[14] = 1000000000000.0;
  odom_.pose.covariance[21] = 1000000000000.0;
  odom_.pose.covariance[28] = 1000000000000.0;
  odom_.pose.covariance[35] = 0.01;
  // get velocity in /odom frame
  math::Vector3 linear;
  linear = this->parent->GetWorldLinearVel();
  odom_.twist.twist.angular.z = this->parent->GetWorldAngularVel().z;
  // convert velocity to child_frame_id (aka base_footprint)
  float yaw = pose.rot.GetYaw();
  odom_.twist.twist.linear.x = cosf(yaw) * linear.x + sinf(yaw) *
linear.y;
  odom_.twist.twist.linear.y = cosf(yaw) * linear.y - sinf(yaw) *
linear.x;
  odom_.header.stamp = current_time;
  odom_.header.frame_id = odom_frame;
  odom_.child_frame_id = base_footprint_frame;
  odometry_publisher_.publish(odom_);
}
```

可以看出：结构体的各个字段如何被赋 值以及如何设定里程主题的名称（这里是odom）。机器人位姿数据创建是在代码的其它部分完成的。

## Step2：创建自定义里程数据

chapter8_tutorials/src文件夹：

odometry.cpp

```
#include <string>
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>

int main(int argc, char** argv) {

ros::init(argc, argv, "state_publisher");
    ros::NodeHandle n;
    ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom",
10);

    // initial position
    double x = 0.0;
    double y = 0.0;
    double th = 0;

    // velocity
    double vx = 0.4;
    double vy = 0.0;
    double vth = 0.4;
    ros::Time current_time;
    ros::Time last_time;
    current_time = ros::Time::now();
    last_time = ros::Time::now();

    tf::TransformBroadcaster broadcaster;
    ros::Rate loop_rate(20);

    const double degree = M_PI/180;

    // message declarations
    geometry_msgs::TransformStamped odom_trans;
    odom_trans.header.frame_id = "odom";
    odom_trans.child_frame_id = "base_footprint";

    while (ros::ok()) {
```

创建一个坐标变换的结构体变量

```
        current_time = ros::Time::now();

        double dt = (current_time - last_time).toSec();
        double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
        double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
        double delta_th = vth * dt;

        x += delta_x;
        y += delta_y;
        th += delta_th;

        geometry_msgs::Quaternion odom_quat;
        odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);

        // update transform
        odom_trans.header.stamp = current_time;
        odom_trans.transform.translation.x = x;
        odom_trans.transform.translation.y = y;
        odom_trans.transform.translation.z = 0.0;
        odom_trans.transform.rotation = tf::createQuaternionMsgFromYa
w(th);

        //filling the odometry
        nav_msgs::Odometry odom;
        odom.header.stamp = current_time;
        odom.header.frame_id = "odom";
        odom.child_frame_id = "base_footprint";
        // position
        odom.pose.pose.position.x = x;
        odom.pose.pose.position.y = y;
        odom.pose.pose.position.z = 0.0;
        odom.pose.pose.orientation = odom_quat;

        // velocity
        odom.twist.twist.linear.x = vx;
        odom.twist.twist.linear.y = vy;
        odom.twist.twist.linear.z = 0.0;
        odom.twist.twist.angular.x = 0.0;
        odom.twist.twist.angular.y = 0.0;
        odom.twist.twist.angular.z = vth;

      last_time = current_time;

        // publishing the odometry and the new tf
```

机器人的位姿信息

计算一段时间后机器人的理论位置

机器人只能前后运动和转向，仅给x和rotation字段赋值。

```
        broadcaster.sendTransform(odom_trans);
        odom_pub.publish(odom);

      loop_rate.sleep();
    }
    return 0;
}
```

发布数据

```
geometry_msgs::TransformStamped odom_trans;
    odom_trans.header.frame_id = "odom";
    odom_trans.child_frame_id = "base_footprint";
```

创建一个坐标变换的结构体变量

```
double dt = (current_time - last_time).toSec();
double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
double delta_th = vth * dt;

x += delta_x;
y += delta_y;
th += delta_th;

geometry_msgs::Quaternion odom_quat;
odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);
```

机器人的位姿信息

计算一段时间后机器人的理论位置

```
odom_trans.header.stamp = current_time;
odom_trans.transform.translation.x = x;
odom_trans.transform.translation.y = 0.0;
odom_trans.transform.translation.z = 0.0;
odom_trans.transform.rotation =
tf::createQuaternionMsgFromYaw(th);
```

机器人只能前后运动和转向，仅给x和rotation字段赋值。

```
// publishing the odometry and the new tf
broadcaster.sendTransform(odom_trans);
odom_pub.publish(odom);
```

发布数据

修改CMakeLists.txt文件：

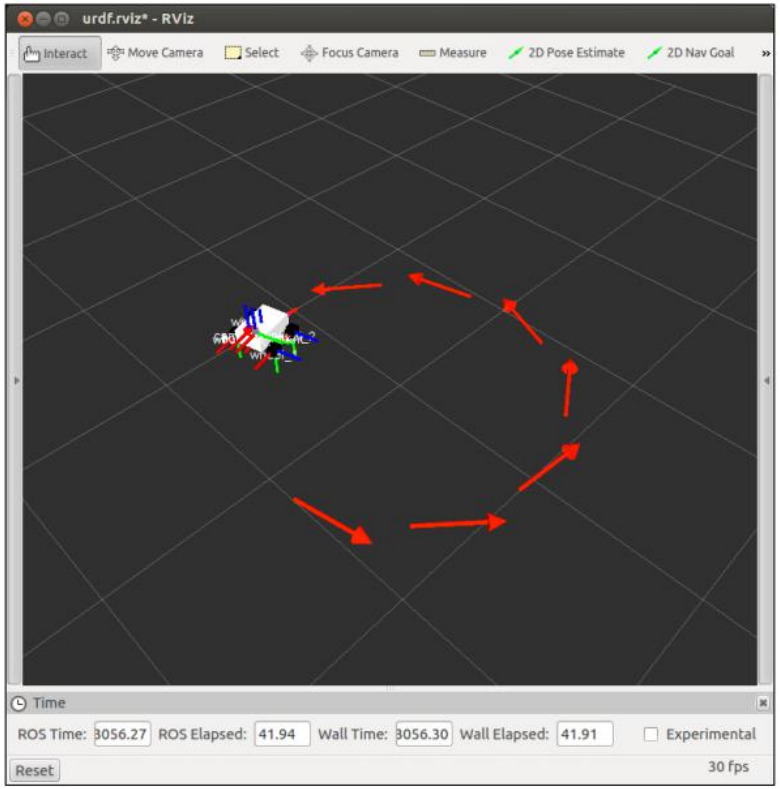
```
add_executable(odometry src/odometry.cpp)
target_link_libraries(odometry ${catkin_LIBRARIES})
```

编译完功能包之后，不使用Gazebo而只用rviz可视化机器人模型及其运动：

$ roslaunch chapter8_tutorials display_xacro.launch model:="`rospack find chapter8_tutorials`/urdf/robot1_base_04.xacro"

运行里程数据处理节点：

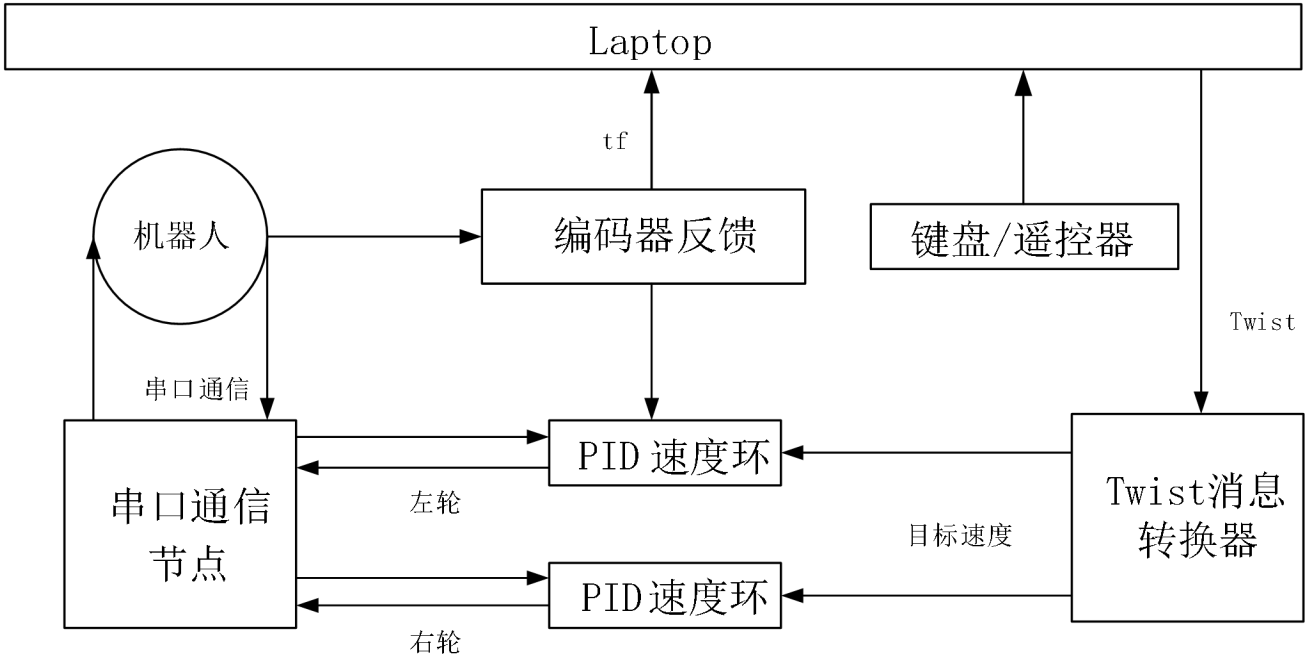
$ rosrun chapter8_tutorials odometry



# 8.5 ROS基控制器

ROS的基控制器是机器人导航中的执行器和控制器，包含了电机、驱动程序、PID控制器以及机器人和Laptop的通信。主要订阅/cmd_vel消息，然后将其转为轮子的目标转速，并提供实时的位置反馈。

ROS使用Twist消息发布运动指令给基控制器。

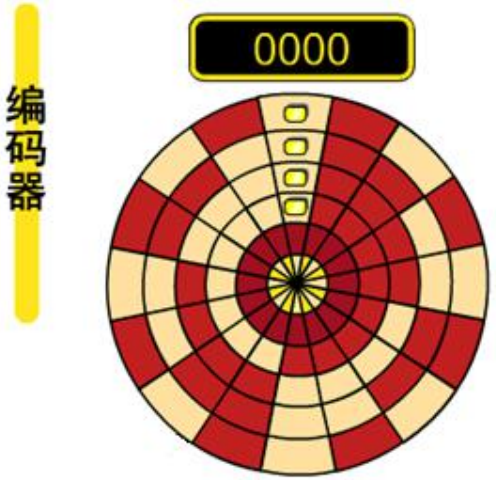



ROS基控制器结构图

# ROS反馈传感器

■编码器

Turtlebot等利用差分控制运动的机器人都安装有编码器，用来统计轮子的转动的圈数，进而计算出机器人移动的距离。




光电编码器原理图

■陀螺仪

编码器仅能够提供机器人的位置，不能提供机器人的姿态，机器人需要从陀螺仪中获取角速度，在积分之后得到实时姿态。

三轴陀螺仪

ROS并不提供任何标准的基础控制器，必须自己编写针对自己移动平台的基础控制器。

机器人通过geometry_msgs/Twist类型的消息控制的。消息的结构:

$ rosmsg show geometry_msgs/Twist

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

线速度向量linear包含x、y和z轴的线速度。角速度向量angular包含各个轴向的角速度。机器人是基于差分轮驱 动平台，仅需要线速度x和角速度z（驱动它的两个电动机只能够让机器人前进、后退或者转向）

在 Gazebo中运行机器人，理解基础控制器的作用。

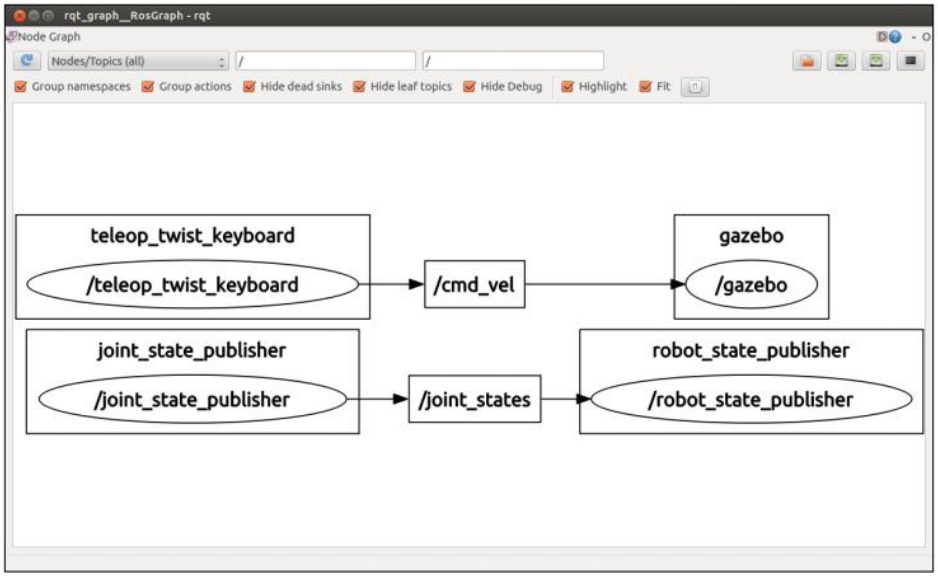
新的命令行窗口，运行以下命令：

$ roslaunch chapter8_tutorials gazebo_xacro.launch model:="`rospack find robot1_description`/urdf/robot1_base_04.xacro"

$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py

在所有节点都启动和正常运行之后，查看各个节点之 间的关系：

$ rosrun rqt_graph



## Step1：使用Gazebo创建里程数据

Gazebo内部的工作原理，仿真程序插件diffdrive_plugin.cpp 的代码，链接地址为：源码

```
void GazeboRosSkidSteerDrive::Load(physics::ModelPtr _parent,
sdf::ElementPtr _sdf)
{
    ……
    // ROS: Subscribe to the velocity command topic (usually "cmd_vel")
    ros::SubscribeOptions so =
        ros::SubscribeOptions::create<geometry_msgs::Twist>
            (command_topic_, 1,boost::bind
                (&GazeboRosSkidSteerDrive::cmdVelCallback, this, _1),
                    ros::VoidPtr(), &queue_);
    ……
}
```

完成对主题的订阅

bind参数用法:

```
//g是以个有2个参数的可调用对象
auto g = bind(func, a, b, _2, c, _1);//func是有5个参数的函数
调用g(X, Y), 等于 func(a, b, Y, c, X)
```

消息到达时，线速度和角速度被存储为内部变量。

```
void GazeboRosSkidSteerDrive::cmdVelCallback(
    const geometry_msgs::Twist::ConstPtr& cmd_msg)
{                                                    执行对消息的处理
    boost::mutex::scoped_lock scoped_lock(lock);
    x_ = cmd_msg->linear.x;
    rot_ = cmd_msg->angular.z;
 }

void GazeboRosSkidSteerDrive::getWheelVelocities()
{                                          对每个电动机的速度进行估计
    ……
}

void GazeboRosSkidSteerDrive::UpdateChild()
{                                            对机器人走过的路径进行估计
    ……
}
```

## Step2：创建自己的基础控制器

参考实例（通过串口实现小车控制）：
https://bbs.21ic.com/icview-2832654-1-3.html

chapter8_tutorials/src/base_controller.cpp：

```
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
```

```
#include <iostream>

using namespace std;

double width_robot = 0.1;
double vl = 0.0;
double vr = 0.0;
ros::Time last_time;
double right_enc = 0.0;
double left_enc = 0.0;
double right_enc_old = 0.0;
double left_enc_old = 0.0;
double distance_left = 0.0;
double distance_right = 0.0;
double ticks_per_meter = 100;
double x = 0.0;
double y = 0.0;
double th = 0.0;
geometry_msgs::Quaternion odom_quat;

void cmd_velCallback(const geometry_msgs::Twist &twist_aux)
{
    geometry_msgs::Twist twist = twist_aux;
    double vel_x = twist_aux.linear.x;
    double vel_th = twist_aux.angular.z;
    double right_vel = 0.0;
    double left_vel = 0.0;

    if(vel_x == 0){
// turning
        right_vel = vel_th * width_robot / 2.0;
        left_vel = (-1) * right_vel;
    }else if(vel_th == 0){
// forward / backward
        left_vel = right_vel = vel_x;
    }else{
// moving doing arcs
        left_vel = vel_x - vel_th * width_robot / 2.0;
        right_vel = vel_x + vel_th * width_robot / 2.0;
    }
    vl = left_vel;
    vr = right_vel;
}
int main(int argc, char** argv){
```

```
    ros::init(argc, argv, "base_controller");
    ros::NodeHandle n;
    ros::Subscriber cmd_vel_sub = n.subscribe("cmd_vel", 10, cmd_
velCallback);
    ros::Rate loop_rate(10);

    while(ros::ok())
    {

        double dxy = 0.0;
        double dth = 0.0;
        ros::Time current_time = ros::Time::now();
        double dt;
        double velxy = dxy / dt;
        double velth = dth / dt;

      ros::spinOnce();
        dt =  (current_time - last_time).toSec();;
        last_time = current_time;

        // calculate odomety
        if(right_enc == 0.0){
          distance_left = 0.0;
          distance_right = 0.0;
        }else{
          distance_left = (left_enc - left_enc_old) / ticks_per_meter;
          distance_right = (right_enc - right_enc_old) / ticks_per_
meter;
        }

        left_enc_old = left_enc;
        right_enc_old = right_enc;

         dxy = (distance_left + distance_right) / 2.0;
         dth = (distance_right - distance_left) / width_robot;

        if(dxy != 0){
          x += dxy * cosf(dth);
          y += dxy * sinf(dth);
        }

        if(dth != 0){
          th += dth;
        }
```

```
        odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);
        loop_rate.sleep();
    }
}
```

# 8.6 使用ROS创建地图

学习使用在Gazebo中创建的机器人来创建地图、保存地图和加载地图。

chapter8_tutorials/launch/ gazebo_mapping_robot.launch：

```
<?xml version="1.0"?>
<launch>
    <param name="/use_sim_time" value="true" />  <!-- start up wg world -->
    <include file="$(find gazebo_worlds)/launch/wg_collada_world.launch"/>
    <arg name="model" />
    <param name="robot_description" command="$(find xacro)/xacro.py
        $(arg model)" />
    <node name="joint_state_publisher" pkg="joint_state_publisher“
        type="joint_state_publisher" ></node>
    <node pkg="robot_state_publisher" type="state_publisher“
        name="robot_state_publisher" output="screen" >
```

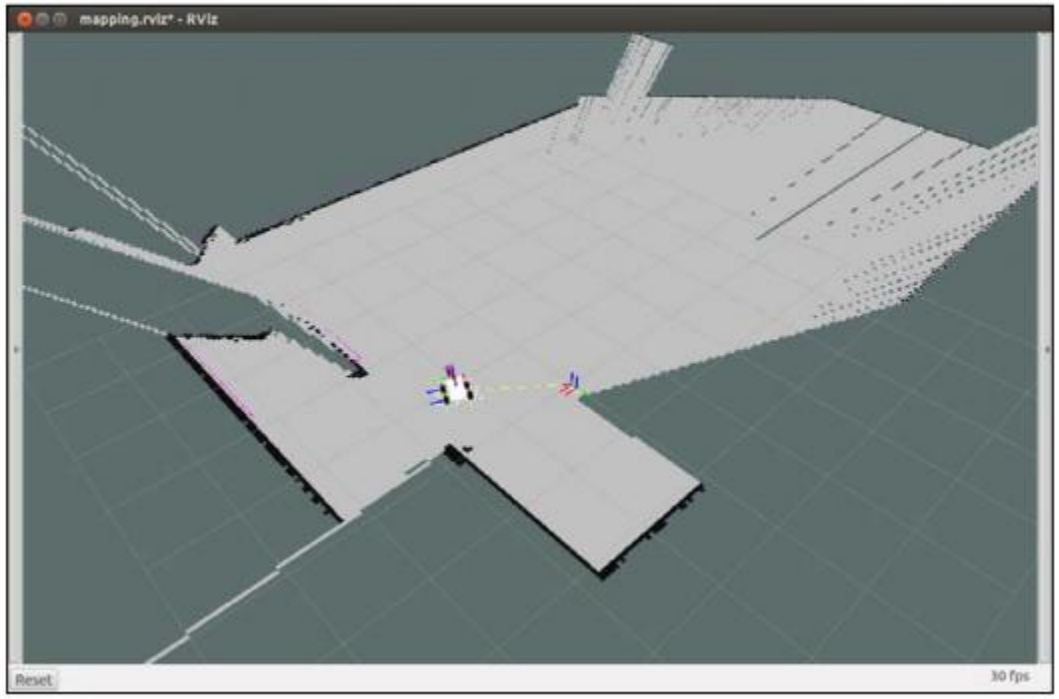
```
    <param name="publish_frequency" type="double" value="50.0" />
    </node>
    <node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
        args="-urdf -param robot_description -z 0.1 -model robot_model"
        respawn="false" output="screen" />
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
        chapter8_tutorials)/launch/mapping. rviz "/>
    <node name="slam_gmapping" pkg="gmapping" type="slam_gmapping">
        <remap from="scan" to=" /robot/laser/scan "/>
        <param name="base_link" value="base_footprint"/>
    </node>

</launch>
```

在仿真环境中，可以使用3D模型，通过rviz配置文件和slam_mapping实时构建地图。

*在命令行中运行这个启动文件，在另一个命令行窗口中运行远程操作节点来移动机器人。

```
$ roslaunch chapter8_tutorials gazebo_mapping_robot.launch
model:="`rospack find robot1_description`/urdf/robot1_base_04.xacro"
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```



## Step1：使用map_server保存地图

可以使用以下命令来保存地图：

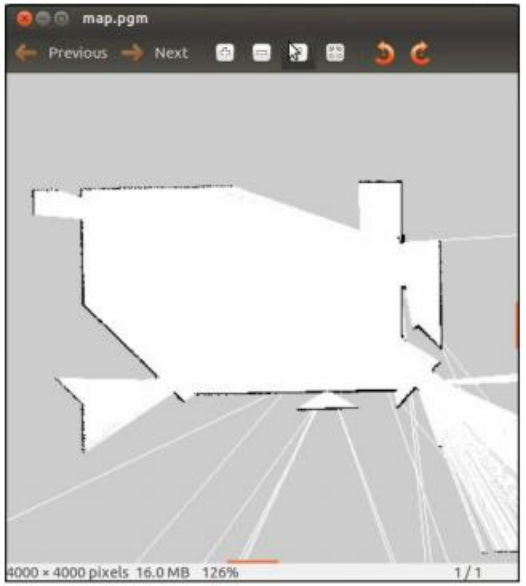
$ rosrun map_server map_saver -f map

该命令会创建两个文件，map.pgm和map.yaml。

map.pgm：以.pgm为格式的地图（可输出灰色地图格式）

Map.yaml：该地图的配置文件。

```
image: map.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```



## Step2：使用map_server加载地图

使用机器人所创建的地图时，必须将其加载到map_server功能包中：

$ rosrun map_server map_server map.yaml

为了简化工作，创建启动文件：

chapter8_tutorials/launch/gazebo_map_robot.launch：

```
<?xml version="1.0"?>
<launch>
   <param name="/use_sim_time" value="true" />
    <!-- start up wg world -->
    <include file="$(find gazebo_ros)/launch/willowgarage_world.
launch"/>
    <arg name="model" />
    <param name="robot_description" command="$(find xacro)/xacro.py
$(arg model)" />
    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
    <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
```

```
name="robot_state_publisher" output="screen" >
  <param name="publish_frequency" type="double" value="50.0" />
  </node>
    <node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
args="-urdf -param robot_description -z 0.1 -model robot_model"
respawn="false" output="screen" />
    <node name="map_server" pkg="map_server" type="map_server" args="
$(find chapter8_tutorials)/maps/map.yaml" />
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find chapter8_
tutorials)/launch/mapping.rviz" />
</launch>
```

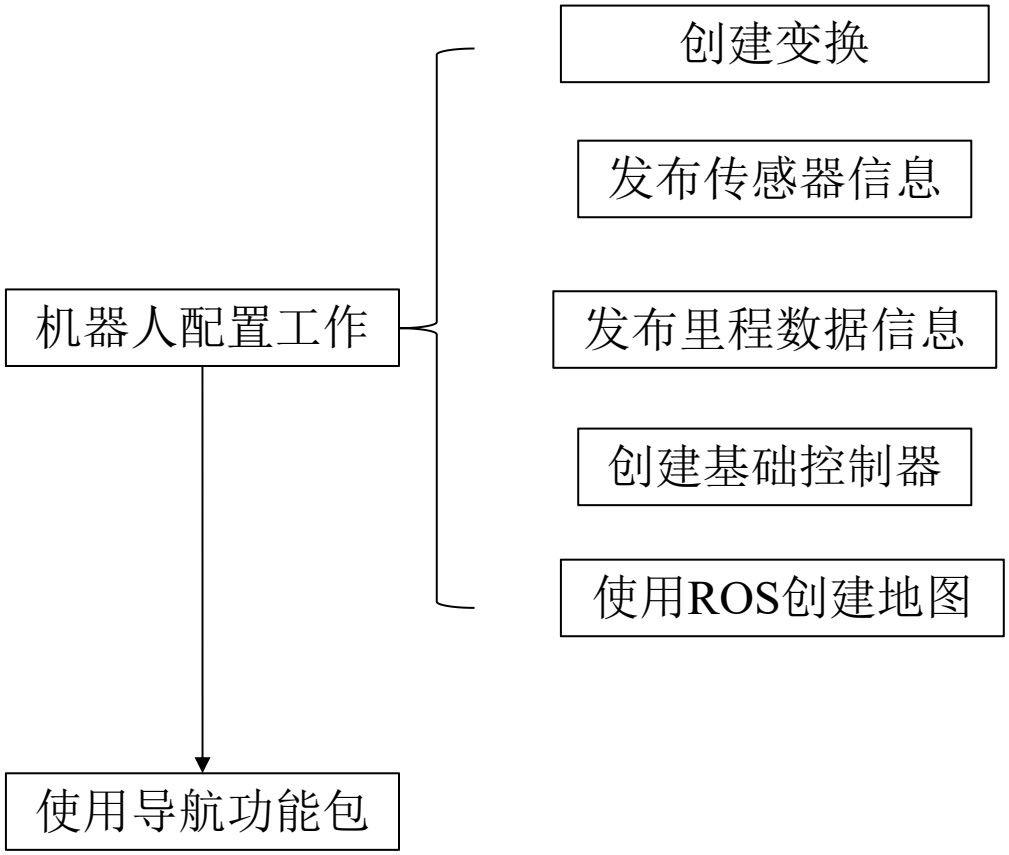
启动文件（需要指定将要使用的机器人模型）

$ roslaunch chapter8_tutorials gazebo_map_robot.launch

model:="`rospack find chapter8_tutorials`/urdf/robot1_base_04.xacro"

在rviz中，可以看到机器人和地图。

# 8.7 小结



# 补充资料：

ros::spin()

循环且监听反馈函数（callback）。

循环就是指程序运行到这里，就会一直在这里循环了。监听反馈函数的意思是，若该节点有callback函数，写ros::spin()在这里，就可以在有对应消息到来时，运行callback函数里面的内容。

注：适用于写在程序的末尾（其后的代码不会被执行），适用于订阅节点，且订阅速度没有限制的情况。

ros::spinOnce()

监听反馈函数（callback）。只能监听反馈，不能循环。需要监听时，可调用此函数。

此函数比较灵活，尤其是控制接收速度时，可配合ros::ok()使用。

# 补充资料：

```
ros::Rate loop_rate(10);
while(ros::ok())
{
    ros::spinOnce();
    loop_rate.sleep();
}
```

可以控制以10Hz速度运行callback函数。

```
while(ros::ok())
{
    ros::spinOnce();
}
```

相当于ros::spin()

# 参考资料

ROS外设：

http://wiki.ros.org/Sensors

参考书籍

《ROS机器人开发实践》胡春旭 著