

ROS Notebook

Wu Yutian

2021.11.13

前言

主要参考了胡春旭的《ROS 机器人开发实践》一书。

Wu Yutian

2021.11.13

Contents

| | | |
|----------|---|----------|
| 1 | ROS 的基本架构——ROS1 | 1 |
| 1.1 | 整体架构 | 1 |
| 1.2 | 计算图的视角 | 2 |
| 1.2.1 | 节点 Node | 2 |
| 1.2.2 | 话题 Topic | 2 |
| 1.2.3 | 服务 Service | 2 |
| 1.2.4 | 节点管理器 Master | 2 |
| 1.3 | 文件系统 | 2 |
| 1.3.1 | 功能包 | 2 |
| 1.4 | 通信机制 | 3 |
| 1.4.1 | 话题通信机制——Topic | 3 |
| 1.4.2 | 服务通信机制——Service | 4 |
| 1.4.3 | 参数管理机制——Parameter | 5 |
| 2 | ROS 基础 | 6 |
| 2.1 | turtlesim 功能包 | 6 |
| 2.2 | 创建工作空间和功能包 | 6 |
| 2.2.1 | 创建工作空间 | 6 |
| 2.2.2 | 创建功能包 | 7 |
| 2.3 | 工作空间的覆盖 | 7 |
| 2.4 | Topic 中的 Publisher 和 Subscriber | 7 |
| 2.4.1 | Publisher 的创建 | 7 |
| 2.4.2 | Subscriber 的创建 | 8 |
| 2.4.3 | 自定义话题消息 | 9 |
| 2.4.4 | CMakeLists 的编写 | 10 |
| 2.5 | Service 中的 Client 和 Server | 11 |
| 2.5.1 | 创建 Client | 11 |
| 2.5.2 | 创建 server | 12 |

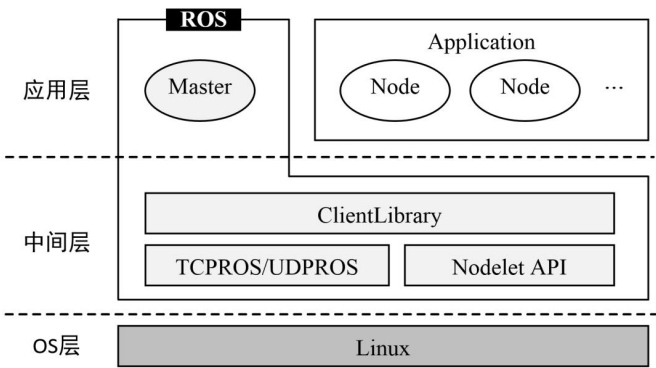
| | | |
|-------|--|----|
| 2.5.3 | 自定义服务数据 | 13 |
| 2.5.4 | CMakeLists 的编写 | 13 |
| 2.6 | ROS 中的命名空间 | 14 |
| 2.6.1 | 有效的命名 | 14 |
| 2.6.2 | 命名解析 | 14 |
| 2.6.3 | 命名重映射 | 15 |
| 2.7 | 多机通信 | 16 |
| 3 | ROS 中的常用组件 | 17 |
| 3.1 | launch 文件 | 17 |
| 3.1.1 | 启动节点 | 17 |
| 3.1.2 | 系统参数设置 | 18 |
| 3.1.3 | 设置内部变量 | 18 |
| 3.1.4 | 重映射机制 | 18 |
| 3.1.5 | 嵌套复用 | 18 |
| 3.2 | TF 坐标变换 | 19 |
| 3.2.1 | TF 辅助工具 | 19 |
| 3.2.2 | TF 中的 Boardcaster 和 Listener | 20 |
| 3.3 | Qt 工具箱 | 22 |
| 3.3.1 | 日志输出工具 rqt_consile | 22 |
| 3.3.2 | 计算图可视化工具 rqt_graph | 23 |
| 3.3.3 | 数据绘制工具 rqt_plot | 23 |
| 3.3.4 | 参数动态配置工具 rqt_reconfigure | 23 |
| 3.4 | rviz 三维可视化平台 | 23 |
| 3.5 | Gazebo 仿真环境 | 23 |
| 3.6 | rosvbag 数据记录与回放 | 23 |
| 3.6.1 | 记录数据 | 23 |
| 3.6.2 | 回放数据 | 24 |
| 4 | 机器人的建模与仿真 | 25 |
| 4.1 | URDF 文件 | 25 |
| 4.1.1 | <link> 标签 | 25 |
| 4.1.2 | <joint> 标签 | 26 |
| 4.1.3 | <robot> 标签 | 27 |
| 4.1.4 | <gazebo> 标签 | 27 |
| 4.2 | 创建 URDF 模型 | 28 |
| 4.2.1 | 创建功能包 | 28 |
| 4.2.2 | URDF 模型代码 | 28 |

- 4.3 使用 xacro 优化 URDF 模型 31
 - 4.3.1 xacro 的三个机制 31
 - 4.3.2 引用 xacro 文件 33
 - 4.3.3 显示 xacro 优化后的模型 33
- 4.4 添加传感器模型 33
- 4.5 基于 ArbotiX 和 rviz 的仿真器 34
 - 4.5.1 在 ROS-melodic 中安装 ArbotiX 34
 - 4.5.2 配置 ArbotiX 控制器 35
 - 4.5.3 运行仿真 36
- 4.6 ros_control 36
 - 4.6.1 ros_control 的框架 36
 - 4.6.2 控制器 36
 - 4.6.3 硬件接口 37

Chapter 1

ROS 的基本架构——ROS1

1.1 整体架构



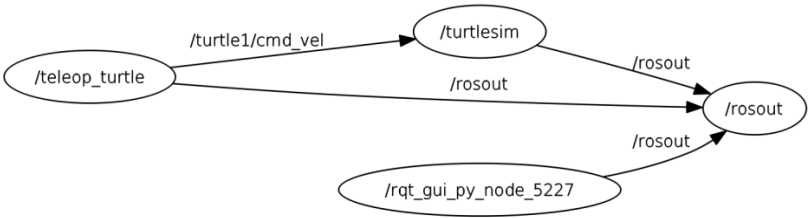
OS 层：是 ROS 依托的底层操作系统，一般是 Ubuntu。

中间层：最重要的就是基于 TCP/UDP 网络，进行封装形成的 TCPROS/UDPROS 通信系统，这其中包括了 Topic 的发布、订阅的通信方式，Service 的客户端、服务器的通信方式等。另外 ROS 还提供了一种进程内通信的方式——Nodelet，可以为多进程通信提供一种更优化的数据传输方式，适合对实时性要求较高的应用。

在通信机制的基础上，ROS 还在中间层提供了大量的机器人开发相关的实用功能，如：数据类型定义、坐标变换、运动控制等。

应用层：ROS 需要运行一个管理者——Matser，负责整个系统的正常运行。其他的一些相关的 ROS 功能包都是以节点（Node）的方式运行，一般来说，简单的开发工作只需要关注节点的标准输入输出接口，而不需要关注模块的内部实现。

1.2 计算图的视角



从计算图的视角来看 ROS 的功能模块，它们都是以节点为单位独立运行的，甚至可以分布于不同的主机中。

1.2.1 节点 Node

节点就是一些执行运算任务的进程，它们之间可以相互通信。

1.2.2 话题 Topic

消息以一种发布/订阅 (publish/subscribe) 的方式传递，发布者和订阅者并不了解彼此的存在，系统中可能有多个节点发布或者订阅同一个话题的消息。

1.2.3 服务 Service

对于双向的同步传输模式，采用基于客户端/服务器 (Client/Server) 的模型，包含请求和应答，类似于 Web 服务器，ROS 中只允许有一个节点提供指定命名的服务。

1.2.4 节点管理器 Master

节点管理器帮助 ROS 节点之间相互查找、建立连接，同时还为系统提供参数服务器，管理全局参数。

1.3 文件系统

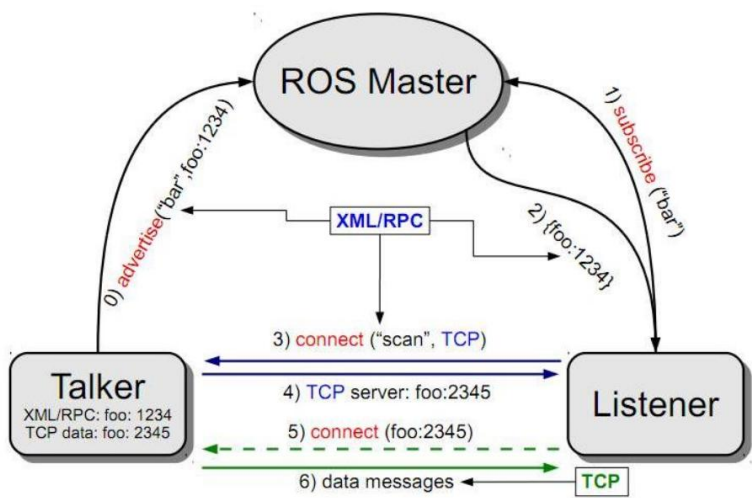
1.3.1 功能包

功能包相关的常用 ROS 命令：

| 命令 | 作用 |
|-------------------|---------------|
| catkin_create_pkg | 创建功能包 |
| rospack | 获取功能包的信息 |
| catkin_make | 编译功能包的信息 |
| roscdep | 自动安装功能包依赖的其他包 |
| roscd | 功能包目录跳转 |
| roscp | 拷贝功能包中的文件 |
| roscd | 编辑功能包中的文件 |
| roscd | 运行功能包中的可执行文件 |
| roslaunch | 运行启动文件 |

1.4 通信机制

1.4.1 话题通信机制——Topic



假设 Talker 首先启动，建立通信的详细过程：

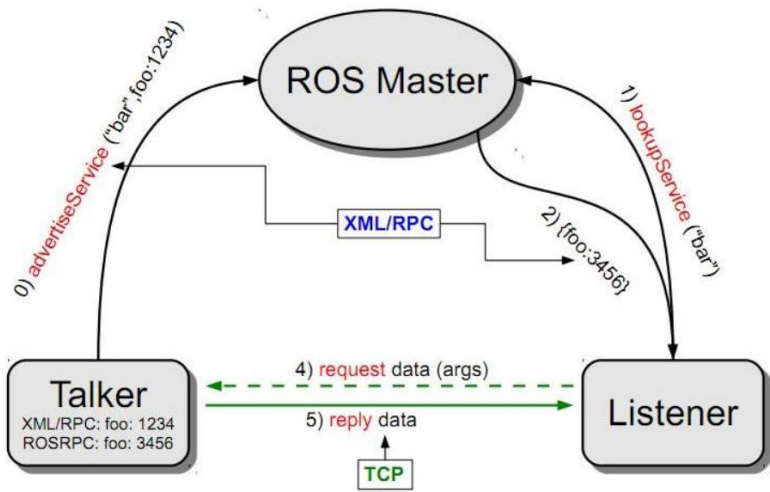
- 1、发布者（Talker）启动，通过 RPC 向 ROS Master 注册发布者的信息，包括：发布者节点信息，话题名，话题缓存大小等；Master 会将这些信息加入注册列表中；
- 2、订阅者（Listener）启动，通过 RPC 向 ROS Master 注册订阅者信息，包括：订阅者节点信息，话题名等；Master 会将这些信息加入注册列表；
- 3、Master 进行节点匹配：Master 会根据订阅者提供的信息，在注册列表中查找匹配的发布者；如果没有发布者（Talker），则等待发布者（Talker）的加入；如果找到匹配的发布者

(Talker)，则会主动把发布者 (Talker) (有可能是很多个 Talker) 的地址通过 RPC 传送给订阅者 (Listener) 节点；

- 4、Listener 接收到 Master 的发出的 Talker 的地址信息，尝试通过 RPC 向 Talker 发出连接请求 (信息包括：话题名，消息类型以及通讯协议 (TCP/UDP))；
- 5、Talker 收到 Listener 发出的连接请求后，通过 RPC 向 Listener 确认连接请求 (包含的信息为自身 TCP 地址信息)；
- 6、Listener 接收到 Talker 的确认消息后，使用 TCP 尝试与 Talker 建立网络连接；
- 7、成功连接之后，Talker 开始向 Listener 发布话题消息数据；

需要注意的是：有可能多个 Talker 连接一个 Listener，也有可能是一个 Talker 连接上多个 Listener (多对多)。

1.4.2 服务通信机制——Service



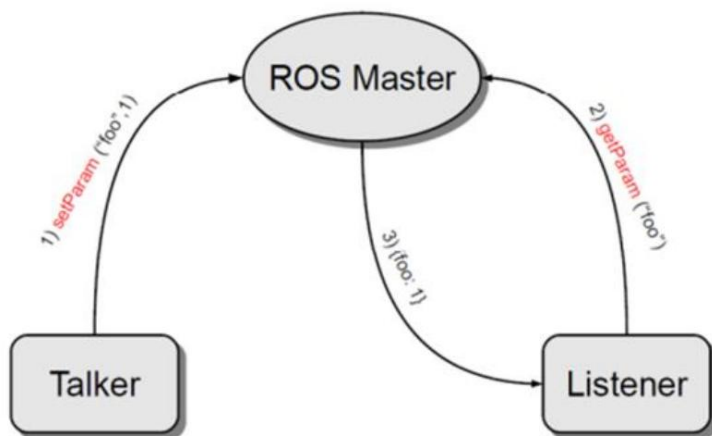
与话题的通信相比，其减少了 Listener 与 Talker 之间的 RPC 通信，建立通信的详细过程：

- 1、发布者 (Talker) 启动，通过 RPC 向 ROS Master 注册发布者的信息，包括：发布者节点信息，话题名，话题缓存大小等；Master 会将这些信息加入注册列表中；
- 2、订阅者 (Listener) 启动，通过 RPC 向 ROS Master 注册订阅者信息，包括：订阅者节点信息，话题名等；Master 会将这些信息加入注册列表；
- 3、Master 进行节点匹配：Master 会根据订阅者提供的信息，在注册列表中查找匹配的发布者；如果没有发布者 (Talker)，则等待发布者 (Talker) 的加入；如果找到匹配的发布者 (Talker)，则会主动把发布者 (Talker) (有可能是很多个 Talker) 的地址通过 RPC 传送给订阅者 (Listener) 节点；

- 4、Listener 接收到 Talker 的确认消息后，使用 TCP 尝试与 Talker 建立网络连接；
- 5、成功连接之后，Talker 开始向 Listener 发布话题消息数据；

需要注意的是：有可能是一个 Talker 连接上多个 Listener（一对多）。

1.4.3 参数管理机制——Parameter



参数共享机制类似于程序中的全局变量，Talker 去更新全局变量（共享的参数），Listener 去获取更新后的全局变量（共享的参数）；这个通信过程不涉及 TCP/UDP 的通信；

- 1、Talker 更新全局变量；Talker 通过 RPC 更新 ROS Master 中的共享参数（包含参数名和参数值）；
- 2、Listener 通过 RPC 向 ROS Master 发送参数查询请求（包含要查询的参数名）；
- 3、ROS Master 通过 RPC 回复 Listener 的请求（包括参数值）；

需要注意的是：如果 Listener 向实时知道共享参数的变化，需要自己不停的去询问 ROS Master；

Chapter 2

ROS 基础

2.1 turtlesim 功能包

接触的第一个 ROS 功能包：turtlesim，其核心是 turtlesim_node 节点。
其中包含的话题和服务如下：

| | 名称 | 类型 | 描述 |
|------|----------------------------|---------------------------------|-----------------------------------|
| 话题订阅 | turtleX/cmd_vel | geometry_msgs/ Twist | 控制乌龟角速度与线速度的 输入指令 |
| 话题发布 | turtleX/pose | turtlesim/Pose | 乌龟的姿态信息：包括 x 与 y 坐标、角度、线速度和角速度 |
| 服务 | clear | std_srvs/Empty | 清楚仿真器中的背景颜色 |
| | reset | std_srvs/Empty | 复位仿真器到初始状态 |
| | kill | turtlesim/Kill | 删除一只乌龟 |
| | spawn | turtlesim/Spawn | 新生一只乌龟 |
| | turtleX/set_pen | turtlesim/Setpen | 设置画笔的颜色和线宽 |
| | turtleX/teleport_absolute | turtlesim/ TeleportAbsolute | 移动乌龟到指定的姿态 |
| | turtleX/teleport_realative | turtlesim/ TeleportRealative | 移动乌龟到指定的角度和距离 |

2.2 创建工作空间和功能包

2.2.1 创建工作空间

工作空间初始化：

```
mkdir ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

初始化后，可以编译整个工作空间：

```
cd ~/catkin_ws/
catkin_make
```

编译后，在工作空间的根目录下会产生 build 和 devel 两个文件夹，在 devel 文件夹中有 setup.bash 形式的环境变量设置脚本，则可以使用 source 命令运行这些脚本配置环境变量，如：

```
source devel/setup.bash
```

但是 source 命令设置的环境变量只在当前终端中有效，所以为了方便，可以讲终端的配置文件（/.bashrc）中加入上面的环境变量的配置语句（要注意写全绝对路径）。

2.2.2 创建功能包

创建功能包的命令如下：

```
cd ~/catkin_ws/src
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

创建完成后，工作空间的 src 目录中会生成一个 <package_name> 的功能包，并且已经包含了 package.xml 和 CMakeList.txt 文件。其中 package.xml 文件提供描述功能包属性的信息，CMakeList.txt 文件记录功能包的编译规则。

进而可以回到工作空间的根目录下进行编译，并设置环境变量。

2.3 工作空间的覆盖

所有工作空间的路径会依次在 ROS_PACKAGE_PATH 环境变量中记录，当设置多个工作空间的环境变量后，新设置的路径在 ROS_PACKAGE_PATH 中会自动放在最前端。在运行时，ROS 会优先查找最前端的工作空间中是否存在指定的功能包，如果不存在，就顺序向后查找其他工作空间，知道最后一个工作空间为止。

2.4 Topic 中的 Publisher 和 Subscriber

2.4.1 Publisher 的创建

```
#include <sstream>
#include "ros/ros.h"
```

```

#include "std_msgs/String.h"
int main(int argc, char **argv){
    // ROS节点初始化
    ros::init(argc, argv, "talker");
    // 创建节点句柄
    ros::NodeHandle n;
    // 创建一个Publisher，发布名为chatter的topic，消息类型为std_msgs::String
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    // 设置循环的频率
    ros::Rate loop_rate(10);
    int count = 0;
    // 一旦发生异常，ros::ok()就会返回false，跳出循环
    while (ros::ok()){
        // 初始化std_msgs::String类型的消息
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        // 发布消息
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        // 循环等待回调函数
        // ros::spinOnce()函数用来处理节点订阅话题的所有回调函数
        // 虽然目前的发布节点并没有任何订阅信息，ros::spinOnce()不是必须的
        // 但是为了保证功能无误，建议所有节点都默认加入该函数
        ros::spinOnce();
        // 按照循环频率延时
        loop_rate.sleep();
        ++count;
    }
    return 0;
}

```

2.4.2 Subscriber 的创建

```

#include "ros/ros.h"
#include "std_msgs/String.h"
// 接收到订阅的消息后，会进入消息回调函数

```

```
// 当有消息到达时，会自动以消息指针作为参数
void chatterCallback(const std_msgs::String::ConstPtr& msg){
    // 将接收到的消息打印出来
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv){
    // 初始化ROS节点
    ros::init(argc, argv, "listener");
    // 创建节点句柄
    ros::NodeHandle n;
    // 创建一个Subscriber，订阅名为chatter的topic，注册回调函数chatterCallback
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
    // 循环等待回调函数
    ros::spin();
    return 0;
}
```

2.4.3 自定义话题消息

编写 msg 文件

使用 msg 文件定义自己的消息类型，一般放置在功能包根目录下的 msg 文件夹中。msg 文件中既可以定义消息类型的变量，也可以定义常量：

```
string name
uint8 sex
uint8 age

uint8 unknown = 0
uint8 male = 1
uint8 female = 2
```

对于稍复杂一些的 ROS 自定义消息，还会包含一个标准格式的头信息 std_msgs/Header:

```
uint32 seq
time stamp
string frame_id
```

其中：seq 是消息的顺序标识，不需要手动设置，Publisher 在发布消息时会自动累加；stamp 是消息中与数据相关联的时间戳，可以用于时间同步；frame_id 是消息中与数据相关联的参考坐标系 id。

编译 msg 文件

(1) 在 package.xml 中添加功能包依赖

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

(2) 在 CMakeLists.txt 文件中添加编译选项

在 find_package 中添加消息生成依赖的功能包 message_generation:

```
find_package(catkin REQUIRED COMPONENTS
    geometry_msgs
    roscpp
    rospy
    std_msgs
    message_generation
)
```

设置 catkin 依赖:

```
catkin_package(
    # INCLUDE_DIRS include
    # LIBRARIES learning_communication
    CATKIN_DEPENDS geometry_msgs roscpp rospy std_msgs message_runtime
    # DEPENDS system_lib
)
```

设置需要编译的 msg 文件:

```
add_message_files(FILES Person.msg)
generate_messages(DEPENDENCIES std_msgs)
```

然后对功能包进行编译, 自定义的消息类型就生效了。

2.4.4 CMakeLists 的编写

几个常用的编译选项:

(1) include_directories

用于设置头文件的相对路径。功能包的一些头文件会放在功能包根目录下的 include 文件夹中, 所以需要添加该文件夹。

(2) add_executable

用于设置需要编译的代码和生成的可执行文件。第一个参数为期望生成的可执行文件的名称, 后面的参数为参与的源码文件 (cpp), 如果需要多个代码文件, 可以在后面依次列出, 中间用空格分隔。

(3) target_link_libraries

用于设置链接库。第一个参数为期望生成的可执行文件的名称，后面依次列出需要链接的库，如果没有使用其他库，添加默认链接库（\${catkin_LIBRARIES}）即可。

(4) add_dependencies

用于设置依赖。在很多应用中，我们需要定义语言无关的消息类型，消息类型会在编译过程中产生相应语言的代码，如果编译的可执行文件依赖这些动态生成的代码，则需要使用 add_dependencies 添加 \${PROJECT_NAME}_generate_messages_cpp 配置，即该功能包动态产生的消息代码。

对于我们的这个例子，CMakeLists.txt 文件如下：

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker ${PROJECT_NAME}_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(talker ${PROJECT_NAME}_generate_messages_cpp)
```

2.5 Service 中的 Client 和 Server

2.5.1 创建 Client

```
#include <cstdlib>
#include "ros/ros.h"
#include "learning_communication/AddTwoInts.h"

int main(int argc, char **argv){
    ros::init(argc, argv, "add_two_ints_client");
    // 从终端命令行获取两个加数，argv[0]是路径，argv[1]和[2]是两个输入参数
    if (argc != 3){
        ROS_INFO("usage: add_two_ints_client X Y");
        return 1;
    }

    ros::NodeHandle n;
    // 创建一个client，请求add_two_int service
    // service消息类型是learning_communication::AddTwoInts
```



```

ros::ServiceClient client = n.serviceClient\
    <learning_communication::AddTwoInts>("add_two_ints");
// 创建learning_communication::AddTwoInts类型的service消息
// 该变量包含两个成员：request和response
learning_communication::AddTwoInts srv;
// atoll()函数将字符串转化为整数
srv.request.a = atoll(argv[1]);
srv.request.b = atoll(argv[2]);
// 发布service请求，等待加法运算的应答结果
// 调用过程会发生阻塞，调用成功后返回true
if (client.call(srv)){
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
}
else{
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
}
return 0;
}

```

2.5.2 创建 server

```

#include "ros/ros.h"
#include "learning_communication/AddTwoInts.h"
// service回调函数，输入参数req，输出参数res
bool add(learning_communication::AddTwoInts::Request &req,
        learning_communication::AddTwoInts::Response &res){
    // 将输入参数中的请求数据相加，结果放到应答变量中
    res.sum = req.a + req.b;
    ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
    ROS_INFO("sending back response: [%ld]", (long int)res.sum);
    return true;
}

int main(int argc, char **argv){
    ros::init(argc, argv, "add_two_ints_server");
    ros::NodeHandle n;
    // 创建一个名为add_two_ints的server，注册回调函数add()
    ros::ServiceServer service = n.advertiseService("add_two_ints", add);
}

```

```
// 循环等待回调函数
ROS_INFO("Ready to add two ints.");
ros::spin();
return 0;
}
```

2.5.3 自定义服务数据

编写 srv 文件

使用 srv 文件定义自己的消息类型，一般放置在功能包根目录下的 srv 文件夹中。该文件包含 request 和 response 两个数据域，两个数据域之间用“—”（三个减号）分隔，如：

```
int64 a
int64 b
---
int64 sum
```

编译 srv 文件

- (1) 在 package.xml 中添加功能包依赖（与自定义话题消息相同）

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

- (2) 在 CMakeLists.txt 文件中添加编译选项

与自定义话题消息相同也是添加 message_generation 包，

```
find_package(catkin REQUIRED COMPONENTS
    geometry_msgs
    roscpp
    rospy
    std_msgs
    message_generation
)
add_service_files(FILES AddTwoInts.srv)
```

2.5.4 CMakeLists 的编写

与 Topic 类似：

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(server src/server.cpp)
target_link_libraries(server ${catkin_LIBRARIES})
add_dependencies(server ${PROJECT_NAME}_gencpp)

add_executable(client src/client.cpp)
target_link_libraries(client ${catkin_LIBRARIES})
add_dependencies(client ${PROJECT_NAME}_gencpp)
```

2.6 ROS 中的命名空间

2.6.1 有效的命名

- 1、首字符必须是 ([a-z|A-Z])、波浪线 (~) 或者左斜杠 (/)
- 2、后续字母可以是字母或数字 ([0-9|a-z|A-Z])、下划线 (_) 或者左斜杠

2.6.2 命名解析

全局名称: /global/name

全局名称的首字符是左斜杠，它之所以称为全局，是因为它的解析度最高，可以在全局范围内直接访问。

但是在系统中，全局名称越少越好，因为过多的全局名称会影响功能包的可移植性。

相对名称: relative/name

相对名称由 ROS 提供默认的命名空间，不需要带有开头的左斜杠，ROS 会对一个相对名称进行解析，进而得到一个全局名称来使用，就类似与我们平时使用的相对路径。相对名称的使用会提高可移植性。

例如：在默认命名空间/relative 内使用相对名称 name，则系统会将其解析为全局名称：/relative/name。

ROS 提供的三种指定默认命名空间的方式：

- 1、通过命令参数设置

调用 `ros::init()` 的程序会接受一个名为 `__ns` 的命令行参数，用来设置默认命名空间：

```
__ns:=default-namespace
```

- 2、在 launch 文件中设置

在 launch 文件中可以通过参数 `ns` 来设置默认命名空间：

```
<node pkg="turtlesim" type="turtlesim_node" name="turtlesim\_node" ns="sim1"/>
```

- 3、使用环境变量设置

在执行 ROS 程序的终端中设置默认命名空间的环境变量：

```
export ROS_NAMESPACE = default-namespace
```

私有名称： private/name

私有名称是一个节点内部私有的资源名称，只会在节点内部使用。私有名称以波浪线 “~” 开始。类似相对名称，也需要 ROS 为其解析，成为一个有意义的全局名称，不同的是，私有名称并不使用当前的默认命名空间，而是使用节点的全局名称作为命名空间。

例如有一个节点的全局名称是/sim1/pubvel，其中的一个私有名称为 ~/max_vel，则其会被解析成全局名称： /sim1/pubvel/max_vel。

ROS 命名解析总结

| 节点 | 全局名称 | 相对名称 (默认) | 私有名称 |
|-----------|---------------------|----------------------|-----------------------------|
| /node1 | /bar ->/bar | Bar ->/bar | ~bar ->/node1/bar |
| /wg/node2 | /bar ->/bar | Bar ->/wg/bar | ~bar ->wg/node2/bar |
| /wg/node3 | /foo/bar ->/foo/bar | foo/bar ->wg/foo/bar | ~foo/bar ->wg/node3/foo/bar |

2.6.3 命名重映射

所有的 ROS 节点内的资源名称都可以在节点启动的时候进行重映射，这一特性支持我们同事打开多个相同的节点，而不会发生命名冲突。

命名重映射语法：

```
old\_name:=new\_name
```

例如，要将 chatter 重映射为/wg/chatter，在节点启动时候可以输入如下命令：

```
$ rosrun rospy_tutorials talker chatter:=wg/chatter
```

需要注意：ROS 的命名解析是在命名重映射之前发生的。所以当我们使用 “foo:=bar” 时，会将节点内所有 foo 命名映射为 bar，而如果我们重映射 “/foo:=bar” 时，ROS 只会讲全局解析为/foo 的名称重映射为 bar。

命名重映射和命名解析之间的关系：

| 节点命名空间 | 重映射参数 | 匹配名称 | 解析名称 |
|--------|------------------|--------------|------------|
| / | foo:=bar | foo,/foo | /bar |
| /baz | foo:=bar | foo,/baz/foo | /baz/bar |
| / | /foo:=bar | foo,/foo | /bar |
| /baz | /foo:=bar | /foo | /baz/bar |
| /baz | /foo:=/a/b/c/bar | /foo | /a/b/c/bar |

2.7 多机通信

设置 IP 地址

- 1、确保所有计算机处于同一网络中，使用 ifconfig 命令查看本机的局域网 ip 地址。
- 2、分别在每台计算机的/etc/hosts 文件中添加其他计算机的 ip 地址和对应的计算机名称。
- 3、测试是否能够 ping 通其他计算机。

设置 ROS_MASTER_URI

因为系统中只能存在一个 Master，所以从机需要知道 Master 的位置，可以在从机中使用如下命令，将 Master 的地址写入环境变量中：

```
$ echo "export ROS_MASTER_URI = http://<主机名>::11311" >> ~/.bashrc
```

Chapter 3

ROS 中的常用组件

3.1 launch 文件

launch 文件是 ROS 中同时启动多个节点的途径，它还可以自动启动 ROS Master 节点管理器，并且实现每个节点的各种配置。

launch 文件采用 XML 的形式进行描述，XML 文件必须包含一个根元素，launch 文件的根元素采用 `<launch>` 标签定义，文件中的其他内容都必须包含在这个标签中。

3.1.1 启动节点

采用 `<node>` 标签启动 ROS 节点，语法如下：

```
<node pkg = "package-name" type = "executable-name" name = "node-name"/>
```

- pkg 定义节点所在的功能包名称
- type 定义节点的可执行文件名称
- name 定义节点运行时的名称，讲覆盖节点中 `init()` 赋予节点的名称

另外还有如下可选的属性参数：

- output = "screen": 讲节点的标准输出打印到终端 (默认输出为日志文档)
- respawn = "true": 复位属性，该节点停止时，会自动重启，默认为 false
- required = "true": 必要节点，当该节点终止时，launch 文件中的其他节点也被终止
- ns = "namespace": 命名空间，为节点内的相对名称添加命名空间前缀
- args = "arguments": 节点需要输入的参数

3.1.2 系统参数设置

使用 `<param>` 标签来设置 ROS 系统运行中的参数（即 parameter），存储在参数服务器中。launch 文件执行后，parameter 就加载到 ROS 的参数服务器上。

每个活跃的点都可以通过 `ros::param::get()` 接口来获取 parameter 的值，用户也可以在终端中通过 `rosparam` 命令获得 parameter 的值。

`<param>` 标签的语法如下：

```
<param name = "output_frame" value = "odom"/>
```

另外，ROS 也提供了一种从文件中批量加载参数的方法，使用标签 `<rosparam>`，其语法如下：

```
<rosparam file = "${find 2dnav_pr2)/config/costmap_common_params.yaml" command  
= "load" ns = "local_costmap"/>
```

`<rosparam>` 标签可以帮我们将一个 YAML 格式的文件中的全部参数加载到 ROS 中，需要将 `command` 属性设置为“load”。

3.1.3 设置内部变量

使用 `<arg>` 标签可以设置 launch 文件内部的局部变量（argument），仅限于 launch 文件内部使用，语法如下：

```
<arg name = "arg-name" default = "arg-value"/>
```

在 launch 文件中使用 argument 时，可以使用如下语法进行调用：

```
<node pkg = "package" type = "type" name = "name" args = "${arg arg-name}"/>
```

3.1.4 重映射机制

使用 `<remap>` 标签可以实现重映射的功能，可以给功能包的接口名称重映射一下，取一个别名，可以用来实现不同功能包之间的接口匹配，语法如下：

```
remap from = "turtlebot/cmd_vel" to = "/cmd_vel"/>
```

3.1.5 嵌套复用

使用 `<include>` 标签可以实现在一个 launch 文件中包含其他的 launch 文件。即可直接复用其他已有的 launch 文件中的内容，语法如下：

```
<include file = "${dirname)/other.launch"/>
```

3.2 TF 坐标变换

TF 是一个让用户随时间跟踪多个坐标系的功能包，它使用树形数据结构，根据时间缓冲并维护多个坐标系之间的坐标变换关系。

3.2.1 TF 辅助工具

1.tf_monitor

功能是打印 TF 树中所有坐标系的发布状态，使用方法如下：

```
$ tf_monitor
$ tf_monitor <source_frame> <target_frame>
```

2.tf_echo

功能是查看指定坐标系之间的变换关系，使用方法如下：

```
$ tf_echo <source_frame> <target_frame>
```

3.static_transform_publisher?

功能是发布两个坐标系之间的静态坐标变换，这两个坐标系不发生相对的位置变化，使用方法如下：

```
$ static_transform_publisher x y z yaw pitch roll frame_id child_frame_id
period_in_ms
$ static_transform_publisher x y z qx qy qz qw frame_id child_frame_id
period_in_ms
```

以上两种命令格式，需要设置坐标的偏移参数和旋转参数：偏移参数使用相对于 xyz 轴的坐标位移；旋转参数分别采用了欧拉角和四元数的表达方式，并设置发送频率以 ms 为单位。

另外，该命令还可以在 launch 文件中使用，语法如下：

```
<launch>
<node pkg = "tf" type = "static_transform_publisher" name = "link1_broadcaster"
args = "1 0 0 0 0 0 1 link1_parent link1 100"/>
<\launch>
```

4.view_frame

这是一个可视化的调试工具，可以生成 PDF 文件，显示整棵 TF 树的信息，使用方法如下：

```
$ rosrun tf view_frames
```


3.2.2 TF 中的 Broadcaster 和 Listener

以基于 TF 的乌龟自动跟踪例程为例。

创建 Broadcaster

创建一个发布乌龟坐标系与世界坐标系之间的 TF 变换的节点。

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <turtlesim/Pose.h>
std::string turtle_name;
//回调函数
void poseCallback(const turtlesim::PoseConstPtr& msg){
    // tf广播器
    static tf::TransformBroadcaster br;
    // 根据乌龟当前的位姿，设置相对于世界坐标系的坐标变换
    // setOrigin设置平移变换 setRotation设置旋转变换
    tf::Transform transform;
    transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
    tf::Quaternion q;
    q.setRPY(0, 0, msg->theta);
    transform.setRotation(q);
    // 发布坐标变换 TF消息的数据类型为tf::StampedTransform
    // 包含坐标变换、时间戳，并指定坐标变换的源坐标系(parent)和目标坐标系(child)
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world",
turtle_name));
}
int main(int argc, char** argv){
    // 初始化节点
    ros::init(argc, argv, "my_tf_broadcaster");
    if (argc != 2){
        ROS_ERROR("need turtle name as argument");
        return -1;
    };
    turtle_name = argv[1];
    ros::NodeHandle node;
    // 订阅乌龟的pose信息 订阅到之后，就会进入回调函数进行TF广播
    ros::Subscriber sub = node.subscribe(turtle_name+"/pose", 10, &poseCallback);
```

```

    ros::spin();
    return 0;
};

```

创建 Listener

监听 TF 消息，并且从中获取 turtle2 相对于 turtle1 坐标系的变换，从而控制 turtle2 移动。

```

#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <geometry_msgs/Twist.h>
#include <turtlesim/Spawn.h>
int main(int argc, char** argv){
    ros::init(argc, argv, "my_tf_listener");
    ros::NodeHandle node;
    // 通过Service, 产生第二只乌龟turtle2
    ros::service::waitForService("spawn");
    ros::ServiceClient add_turtle =
    node.serviceClient<turtlesim::Spawn>("spawn");
    turtlesim::Spawn srv;
    add_turtle.call(srv);
    // 定义turtle2的速度控制发布者
    ros::Publisher turtle_vel =
    node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);
    // tf监听器
    tf::TransformListener listener;
    ros::Rate rate(10.0);
    while (node.ok()){
        // Broadcaster发布的就是这种类型的消息
        tf::StampedTransform transform;
        try{// 查找turtle2与turtle1的坐标变换
            // 其中/turtle2为当前坐标系, turtle1为目标坐标系
            listener.waitForTransform("/turtle2", "/turtle1", ros::Time(0),
            ros::Duration(3.0));
            listener.lookupTransform("/turtle2", "/turtle1", ros::Time(0),
            transform);
        }
        catch (tf::TransformException &ex) {
            ROS_ERROR("%s",ex.what());

```

```

        ros::Duration(1.0).sleep();
        continue;
    }

    // 根据turtle1和turtle2之间的坐标变换, 计算turtle2需要的线速度和角速度
    // 并发布速度控制指令, 使turtle2向turtle1移动
    geometry_msgs::Twist vel_msg;
    vel_msg.angular.z = 4.0 * atan2(transform.getOrigin().y(),
                                    transform.getOrigin().x());
    vel_msg.linear.x = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                   pow(transform.getOrigin().y(), 2));

    turtle_vel.publish(vel_msg);
    rate.sleep();
}

return 0;
};

```

其中两个重要函数:

- `waitForTransform()`

给定源坐标系和目标坐标系, 等待两个坐标系之间指定时间的变换关系, 该函数会阻塞程序运行, 所以要设置超时时间 (timeout)

- `lookupTransform()`

给定源坐标系和目标坐标系, 得到两个坐标系之间指定时间的坐标变换, `ros::Time(0)` 表示获取最新一次的坐标变换。

3.3 Qt 工具箱

这是一个基于 Qt 架构的后台图形工具套件——`rqt_common_plugins`。

安装命令:

```

$ sudo apt-get install ros-kinetic-rqt
$ sudo apt-get install ros-kinetic-rqt-common-plugins

```

3.3.1 日志输出工具 `rqt_console`

`rqt_console` 用来图像化显示和过滤 ROS 系统运行状态中的所有日志消息, 包括 `info`、`warn`、`error` 等, 使用如下命令启动:

```

$ rqt_console

```

3.3.2 计算图可视化工具 rqt_graph

rqt_graph 可以图形化显示当前 ROS 系统中的计算图，使用如下命令启动：

```
$ rqt_graph
```

3.3.3 数据绘制工具 rqt_plot

rqt_plot 是一个二位数值曲线绘制工具，可以将需要显示的数据在 xy 坐标系中使用曲线绘制出来，使用如下命令启动：

```
$ rqt_plot
```

3.3.4 参数动态配置工具 rqt_reconfigure

rqt_reconfigure 可以在不重启系统的情况下，动态配置 ROS 系统中的参数，但是该功能需要在代码中设置参数的相关属性。从而支持动态配置，使用如下命令启动：

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

3.4 rviz 三维可视化平台

在 rviz 中，可以使用 XML 对机器人、周围物体等任何实物进行尺寸、质量、位置、材质、关节等属性的描述，并且在界面中呈现出来。

3.5 Gazebo 仿真环境

虽然 Gazebo 中的机器人模型与 rviz 使用的模型相同，但是需要在模型中加入机器人和周围环境的物理属性，例如质量、摩擦系数、弹性系数等。机器人的传感器信息也可以通过插件的形式加入仿真环境，以可视化的方式进行显示。

3.6 rosbag 数据记录与回放

rosbag 功能包提供了数据记录与回放的功能。

3.6.1 记录数据

开始数据记录的命令：

```
rosbag record -a
```

其中-a(all) 参数表示记录所有发布的消息。数据文件会以.bag 格式保存在当前目录下。

3.6.2 回放数据

查看数据记录文件的命令：

```
$ rosbag info <your bagfile>
```

从该命令的输出信息可以看到数据记录包中包含的所有话题、消息类型、消息数量等信息。

回放所记录的话题数据的命令：

```
$ rosbag play <your bagfile>
```

Chapter 4

机器人的建模与仿真

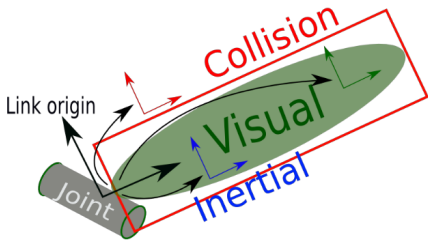
4.1 URDF 文件

URDF (Unified Robot Description Format, 统一机器人描述格式) 是 ROS 中一个非常重要的机器人模型描述格式, ROS 同时也提供了 URDF 文件的 C++ 解析器, 可以解析 URDF 文件中使用 XML 格式描述的机器人模型。

下面说明一下 URDF 文件中常用的几个 XML 标签:

4.1.1 <link> 标签

<link> 标签用于描述机器人某个刚体部分的外形和物理属性, 包括尺寸 (size)、颜色 (color)、形状 (shape)、惯性矩阵 (inertial matrix)、碰撞参数 (collision properties) 等。



从图中可以看出, 检测碰撞的 link 区域大于外观可视的区域, 这就意味着只要有其他物体与 collision 区域相交, 就认为 link 发生碰撞。

<link> 标签的一般结构如下:

```
<link name = "<name of the link>">
  <inertial> ..... </inertial>
  <visual> ..... </visual>
```

```
<collision> ..... </collision>
</link>
```

其中:

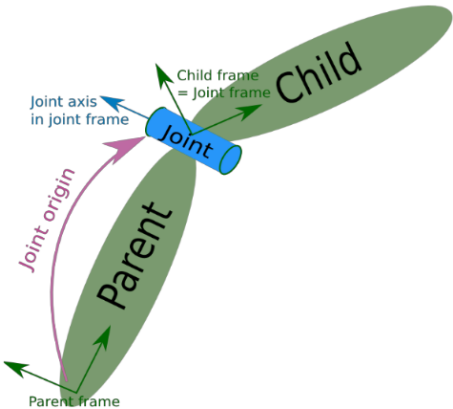
- <visual>: 用于描述机器人 link 部分的外观参数
- <inertial>: 用于描述 link 的惯性参数
- <collision>: 用于描述 link 的碰撞部分

4.1.2 <joint> 标签

<joint> 标签用于描述机器人关节的运动学和动力学属性，包括关节运动的位置和速度限制。根据机器人的关节运动形式，可以将其分为六种类型：

| 关节类型 | 描述 |
|------------|------------------------|
| continuous | 旋转关节，可以围绕单轴无限旋转 |
| revolute | 旋转关节，有旋转的角度限制 |
| prismatic | 滑动关节，沿某一轴线移动的关节，带有位置极限 |
| planar | 平面关节，允许在平面正交方向上平移或者旋转 |
| floating | 浮动关节，允许进行平移、旋转运动 |
| fixed | 固定关节，不允许运动的特殊关节 |

机器人关节的主要作用是连接两个刚体 link，这两个 link 分别称为 parent link 和 child link，如下图所示：



<link> 标签的一般结构如下：

```
<joint name = "<name of the joint>">
```

```

    <parent link = "parent_link"/>
    <child link = "child_link"/>
    <calibration .... />
    <dynamics damping .... />
    <limit effort .... />
    ....
</joint>

```

其中必须指定 joint 的 parent link 和 child link，还可以设置关节的其他属性：

- <calibration>：关节的参考位置，用来校准关节的绝对位置。
- <dynamics>：用于描述关节的物理属性，例如阻尼、静摩擦力，经常在动力学仿真中出现。
- <limit>：用于描述运动的一些极限值，包括关节运动的上下限位置、速度限制、力矩限制等。
- <mimic>：用于描述该关节与已有关节的关系。
- <safety_controller>：用于描述安全控制器参数。

4.1.3 <robot> 标签

<robot> 是完整机器人模型的最顶层标签，<link> 和 <joint> 标签都必须包含在 <robot> 标签内。robot 标签内可以设置机器人的名称，其基本语法如下：

```

<robot name = "name of the robot">
    <link> ..... </link>
    <link> ..... </link>
    <joint> ..... </joint>
    <joint> ..... </joint>
</robot>

```

4.1.4 <gazebo> 标签

<gazebo> 标签用于描述机器人模型在 Gazebo 中仿真所需要的参数，包括机器人材料的属性、Gazebo 插件。该标签不是机器人模型的必需部分，只有在 Gazebo 中仿真时才需要加入，其基本语法如下：

```

<gazebo reference = "link_1">
    <material>Gazebo/Black</material>
</gazebo>

```


4.2 创建 URDF 模型

以 MRobot 机器人为例。

4.2.1 创建功能包

使用如下命令创建一个 urdf 模型的功能包：

```
$ catkin_create_pkg mrobot_description urdf xacro
```

创建好的功能包中包含如下四个文件夹：

- urdf: 用于存放机器人模型的 URDF 文件或 xacro 文件
- meshes: 用于放置 URDF 中引用的模型渲染文件
- launch: 用于保存相关启动文件
- config: 用于保存 rviz 的配置文件

4.2.2 URDF 模型代码

part 1

```
<?xml version="1.0" ?>  
<robot name="mrobot_chassis">
```

首先在文件开头，需要生命该文件使用 XML 描述，然后使用 <robot> 根标签定义一个机器人模型，并定义机器人的名称。

part 2

```
<link name="base_link">  
  <visual>  
    <origin xyz=" 0 0 0" rpy="0 0 0" />  
    <geometry>  
      <cylinder length="0.005" radius="0.13"/>  
    </geometry>  
    <material name="yellow">  
      <color rgba="1 0.4 0 1"/>  
    </material>  
  </visual>  
</link>
```

这一段代码描述机器人的底盘 link, <visual> 标签定义底盘的外观属性;

在 <geometry> 标签下定义几何外观, 我们将底盘抽象成一个圆柱, 使用 <cylinder> 标签定义这个圆柱的半径和高;

然后声明这个底盘圆柱在三维坐标位置和旋转姿态, 使用 <origin> 标签设置底盘中心位置, 底盘中心位于界面的中心点, 所以将坐标设置为 “0 0 0”, 旋转设置也设置为 “0 0 0” 即可 (圆柱体默认是垂直地面放置的);

另外, 使用 <material> 标签设置底盘的颜色——”黄色“, 其中 <color> 标签定义颜色的 RGBA 值 (这里采用百分数描述, A 为透明度参数)。

part 3

```
<joint name="base_left_motor_joint" type="fixed">
  <origin xyz="-0.055 0.075 0" rpy="0 0 0" />
  <parent link="base_link"/>
  <child link="left_motor" />
</joint>
```

这一段代码定义一个关节 joint, 用来连接机器人底盘和左边驱动电机, joint 类型为 fixed 类型, 这种类型的 joint 是固定的 (见 subsection 4.1.2)。

<origin> 标签设置了 joint 的起点, 将起点设置在需要安装电机的底盘位置。

part 4

```
<link name="left_motor">
  <visual>
    <origin xyz="0 0 0" rpy="1.5707 0 0" />
    <geometry>
      <cylinder radius="0.02" length = "0.08"/>
    </geometry>
    <material name="gray">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
  </visual>
</link>
```

这一段代码描述了左侧电机的模型, 外形也是圆柱体, 采用 <cylinder> 标签。

关于 <origin> 标签的设置: 由于我们上面定义了一个 joint 用来将电机连接到底盘上, 电机的位置是相对于 joint 计算的。在 joint 的位置设置中, 已经将其放置到了安装电机的位置, 所以电机模型的位置设置到 “0 0 0” 坐标就可以了。

另外由于圆柱体默认垂直地面, 因此我们需要将电机模型绕 x 轴旋转 90° 放置。

part 5

```

<joint name="left_wheel_joint" type="continuous">
  <origin xyz="0 0.0485 0" rpy="0 0 0"/>
  <parent link="left_motor"/>
  <child link="left_wheel_link"/>
  <axis xyz="0 1 0"/>
</joint>

```

这一段代码定义一个关节 joint，用来连接电机和轮子，joint 类型为 continuous 类型，这种类型的 joint 可以绕一个轴旋转（见 subsection 4.1.2）。

<origin> 标签将轮子的起点设置到电机的一端，<axis> 标签定义该 joint 的旋转轴是 y 轴。

添加物理和碰撞属性

前面的代码仅创建了模型的可视化属性，还需要添加物理属性和碰撞属性，这里以机器人底盘 base_link 为例：

```

<link name="base_link">
  <inertial>
    <mass value="2"/>
    <origin xyz="0 0 0.0"/>
    <inertia ixx="0.01" ixy="0.0" ixz="0.0"
      iyy="0.01" iyz="0.0"
      izz="0.5"/>
  </inertial>

  <visual>
    <origin xyz=" 0 0 0" rpy="0 0 0" />
    <geometry>
      <cylinder length="${base_link_length}" radius="${base_link_radius}"/>
    </geometry>
    <material name="yellow">
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder length="${base_link_length}" radius="${base_link_radius}"/>
    </geometry>
  </collision>
</link>

```

```

        </geometry>
    </collision>
</link>

```

其中 `<inertial>` 标签设置惯性参数，主要包括质量和惯性矩阵，如果是规则物体，可以通过尺寸、质量等公式计算得到惯性矩阵（这里有待学习补充）。

4.3 使用 xacro 优化 URDF 模型

URDF 文件不支持代码复用的特性，因此针对 URDF 模型提出了一种精简化、可复用、模块化的描述形式——xacro。

xacro 有两点优点：精简的模型代码、提供可编程接口。模型的后缀名由 `.urdf` 变为 `.xacro`，并且需要在模型的 `<robot>` 标签中加入 xacro 的声明，代码如下：

```

<?xml version="1.0"?>
<robot name="mrobot" xmlns:xacro="http://www.ros.org/wiki/xacro">

```

4.3.1 xacro 的三个机制

使用常量定义

定义常量的语法如下：

```
<xacro:property name="M_PI" value="3.14159"/>
```

使用常量的语法如下：

```
<origin xyz="0 0 0" rpy="{M_PI} 0 0"/>
```

调用数学公式

在“\$”语句中，不仅可以调用常量，还可以使用一些常用的数学运算，包括加减乘除、负号、括号等（所有运算都会被转换成浮点数进行），语法如下：

```
<origin xyz="0 ${motor_length+wheel_length}/2 0" rpy="0 0 0"/>
```

使用宏定义

xacro 文件可以使用宏定义来声明重复使用的代码模块，而且可以包含输入参数，以 MRobot 机器人的八根支撑柱为例，宏定义的语法示例如下：

```

<xacro:macro name="mrobot_standoff_2in" params="parent number x_loc y_loc z_loc">
    <joint name="standoff_2in_${number}_joint" type="fixed">

```

```

    <origin xyz="${x_loc} ${y_loc} ${z_loc}" rpy="0 0 0" />
    <parent link="${parent}"/>
    <child link="standoff_2in_${number}_link" />
</joint>

<link name="standoff_2in_${number}_link">
  <inertial>
    <mass value="0.001" />
    <origin xyz="0 0 0" />
    <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
      iyy="0.0001" iyz="0.0"
      izz="0.0001" />
  </inertial>

  <visual>
    <origin xyz=" 0 0 0 " rpy="0 0 0" />
    <geometry>
      <box size="0.01 0.01 0.07" />
    </geometry>
    <material name="black">
      <color rgba="0.16 0.17 0.15 0.9"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
    <geometry>
      <box size="0.01 0.01 0.07" />
    </geometry>
  </collision>
</link>
</xacro:macro>

```

以上的宏定义包含五个输入参数：joint 的 parent link，支撑住的序号，支撑柱在 xyz 三个方向上的偏移。这个宏定义在定义一个支撑柱的时候，分别对其 joint 和 link 两个标签进行了定义。

当需要使用该宏模块的时候，按照如下语法进行调用：

```

<mrobot_standoff_2in parent="base_link" number="1" x_loc="-${standoff_x/2 + 0.03}"
y_loc="-${standoff_y - 0.03}" z_loc="${plate_height/2}"/>

```

4.3.2 引用 xacro 文件

引用示例如下：

```
<?xml version="1.0"?>
<robot name="mrobot" xmlns:xacro="http://www.ros.org/wiki/xacro">
  <xacro:include filename="$(find mrobot_description)/urdf/mrobot_body.urdf
.xacro" />
  <!-- MRobot 机器人平台 -->
  <mrobot_body/>
</robot>
```

可以看到，在 robot 标签之间，首先使用了 xacro:include 标签，包含了另一个 xacro 模型文件，然后我们就可以在下面使用被包含文件中的模块了。接下来调用被包含文件中的机器人模型宏定义（机器人模型文件全部是在被包含文件中用一个宏来描述的）。

这样将整个机器人模型作为一个宏有什么好处呢？把机器人整体看做一个模块，方便与其他模型进行集成，比如在后续安装传感器等其他模块时。

4.3.3 显示 xacro 优化后的模型

将 xacro 文件转化成 URDF 文件

使用如下命令可以将 xacro 文件转换成 URDF 文件：

```
$ rosrunc xacro xacro.py mrobot.urdf.xacro > mrobot.urdf
```

直接调用 xacro 文件解析器

也可以省略手动转换的过程，直接在启动文件中调用 xacro 解析器，自动将 xacro 转换成 URDF 文件，在 launch 文件中使用如下语句进行配置：

```
<arg name="model" default="$(find xacro)/xacro --inorder '$(find mrobot_description)/
<arg name="gui" default="true" />
```

进而可以直接使用这个修改后的启动文件，看到 xacro 格式的机器人模型。

4.4 添加传感器模型

首先我们需要自己创建一个传感器模型（xacro 文件），或者去网上下载一个传感器的模型，这里以一个摄像头为例，其模型文件为 camera.xacro。

进而我们可以创建一个顶层 xacro 文件，将机器人主体与摄像头连接起来：

```

<?xml version="1.0"?>
<robot name="mrobot" xmlns:xacro="http://www.ros.org/wiki/xacro">

    <xacro:include filename="$(find mrobot_description)/urdf/mrobot_body.urdf
.xacro" />
    <xacro:include filename="$(find mrobot_description)/urdf/camera.xacro" />

    <xacro:property name="camera_offset_x" value="0.1" />
    <xacro:property name="camera_offset_y" value="0" />
    <xacro:property name="camera_offset_z" value="0.02" />

    <!-- MRobot机器人平台-->
    <mrobot_body/>

    <!-- Camera -->
    <joint name="camera_joint" type="fixed">
        <origin xyz="$(camera_offset_x) $(camera_offset_y) $(camera_offset_z)" rpy="0
        <parent link="plate_2_link"/>
        <child link="camera_link"/>
    </joint>

    <xacro:usb_camera prefix="camera"/>

</robot>

```

在这个顶层文件中，包含了描述摄像头的模型文件以及描述机器人的模型文件，然后使用了一个 fixed 类型的 joint 把摄像头固定到机器人的指定位置。

4.5 基于 ArbotiX 和 rviz 的仿真器

ArbotiX 提供一个差速控制器，通过接收速度控制指令更新机器人的 joint 状态，从而实现机器人在 rviz 中的运动。

4.5.1 在 ROS-melodic 中安装 ArbotiX

Arbotix 本质上就是一个功能包，我们需要像其他我们自己的功能包一样，将其放置在工作空间下的 src 目录下，直接从 git 上下载其源码：

```
$ git clone -b indigo-devel https://github.com/vanadiumlabs/arbotix_ros.git
```

然后重新编译工程即可（注意如果没有将设置环境变量的指令放到.bashrc 中，在这里要记得使用 source 命令设置环境变量）。

4.5.2 配置 ArbotiX 控制器

我们只需要适当修改原本的 launch 文件，然后再创建一个控制器相关的配置文件就可以了。

修改 launch 文件

只是在显示机器人模型的 launch 文件的基础上加上如下内容：

```
<node name="arbotix" pkg="arbotix_python" type="arbotix_driver" output="screen">
  <rosparam file="$(find mrobot_description)/config/fake_mrobot_arbotix.yaml"
  command="load" />
  <param name="sim" value="true"/>
</node>
```

从以上代码可以看出，实际上就是添加了一个控制器节点，这里在仿真环境中使用，需要配置“sim”参数为 true。另外，从这里可以看到，启动时还需要加载一个叫“fake_mrobot_arbotix.yaml”的配置文件。

创建配置文件

配置文件的目录为：功能包目录/config/下，文件内容如下：

```
controllers: {
  base_controller: {
    type: diff_controller,
    base_frame_id: base_footprint,
    base_width: 0.26,
    ticks_meter: 4100,
    Kp: 12,
    Kd: 12,
    Ki: 0,
    Ko: 50,
    accel_limit: 1.0
  }
}
```

控制器的名称为“base_controller”，类型为“diff_controller”（差速控制器），另外还给出了参考坐标系、底盘尺寸、PID 参数等。

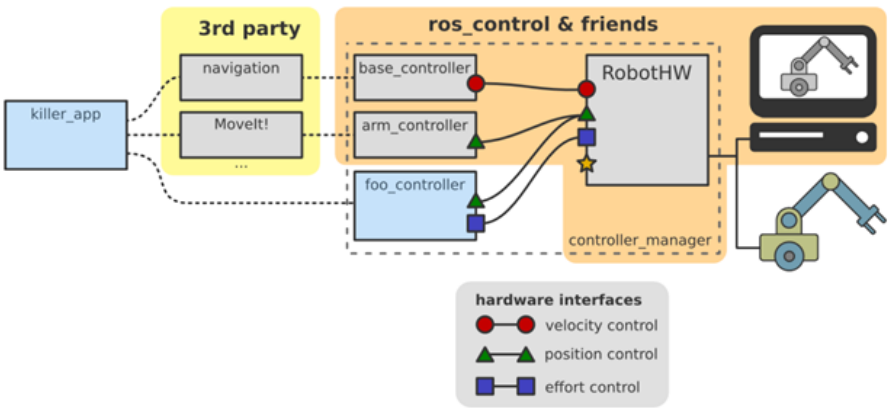
4.5.3 运行仿真

需要注意的是，我们要设置参考坐标系（fixed frame）为“odom”，才可以看到小车的移动。

4.6 ros_control

ros_control 是一套机器人控制中间件，包含一系列控制器接口、传动装置接口、硬件接口、控制器工具箱等。

4.6.1 ros_control 的框架



上图是 ros_control 的总体框架，可以看到正对不同类型的控制器（底盘、机械臂等），ros_control 可以提供多种类型的控制器，但是这些控制器的接口各不相同，为了提高代码的复用率，ros_control 还提供一个硬件的抽象层。硬件抽象层负责机器人硬件资源的管理，而 controller 从抽象层请求资源即可，并不直接接触硬件。

4.6.2 控制器

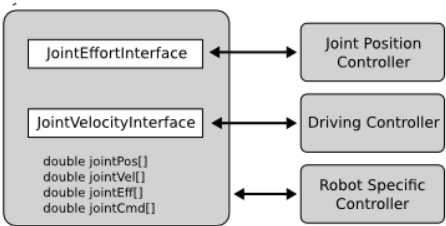
ROS 的 ros_controllers 功能包提供了一些常用的控制器：

- effort_controllers
 - joint_effort_controller
 - joint_position_controller
 - joint_velocity_controller
- joint_state_controller
 - joint_state_controller

- position_controllers
 - joint_position_controller
- velocity_controllers
 - joint_velocity_controller

另外，也可以根据自己的需求创建需要的控制器，并通过控制器管理器进行管理（具体方法有需要再补充）。

4.6.3 硬件接口



硬件接口是控制器与 RobotHW（硬件抽象层）沟通的接口，基本与控制器种类相对应。另外也可以根据自己的需求创建需要的接口（具体方法有需要再补充）。

4.6.4 传动系统

4.6.5 关约束

4.6.6 控制器管理器

4.7 Gazebo 仿真