轮 趣 科 技

ROS2 入门教程

推荐关注我们的公众号获取更新资料

版本说明：

版本	日期	内容说明
V1.0	2024/4/16	第一次发布

网址：www.wheeltec.net

序言

本教程旨在为广大 ROS2 初学者提供一个入门级的学习指南。通过本教程的学习，您将能够掌握 ROS2 的基本概念、核心组件以及开发流程，为后续的机器人项目开发打下坚实的基础。

需要注意的是，机器人技术的发展日新月异，ROS2 也在不断更新和完善。因此，本教程在编写过程中，尽可能涵盖了 ROS2 的最新版本和最新特性。但部分内容可能与您实际使用的 ROS2 版本存在细微差异。建议您在学习过程中，结合 ROS2 官方文档和社区资源，以获取最新的技术资讯和解决方案。

此外，我们强烈推荐您关注我们的公众号“轮趣科技 WHEELTEC”或者 B 站账号“WHEELTEC”，以获取本教程的更新资料和最新动态。我们将不定期发布 ROS2 的学习技巧、实战经验和行业资讯，帮助您更好地掌握 ROS2 技术，并在机器人研发领域取得更大的成就。

最后，感谢您选择本教程作为 ROS2 入门的学习资料。我们衷心希望本教程能够为您的机器人研发之路提供有力的支持和帮助。如果您在学习过程中遇到任何问题或建议，请随时与我们联系，我们将竭诚为您解答和服务。

祝愿您在 ROS2 的学习和实践过程中取得丰硕的成果！

目录

1. ROS2 概述与框架认识

1. 1ROS2 简介

① ROS2 概述

机器人是一种高度复杂的系统性实现，其设计涵盖了机械结构设计、硬件设计、嵌入式软件设计、上层软件设计等多个模块，它是各种硬件与软件的紧密结合，甚至可以说，机器人系统是当今工业体系的高度集成者。

随着国内机器人行业的兴起，ROS 机器人操作系统（Robot Operating System）因其作为快速搭建机器人的通用软件框架的优势，在机器人行业的开发中得到了广泛应用。ROS 遵循“不要重复发明轮子，提高机器人软件复用率”的原则，使得开发者能够复用已有功能，并方便、快捷地拓展新功能。

ROS2 作为第二代机器人操作系统，不仅继承了 ROS 强大的生态基础，更采用了全新的架构设计。这一改进使得 ROS2 能够满足现代机器人系统对实时性、安全性、标准性以及可靠性的高要求。

图 1-1-1　WHEELTEC 小车及其生态支持

学习 ROS2 有几个重要网站，大家之后在学习过程中有些常见问题可以直接在下面这些网站进行查询、搜索：

表 1-1　常用的搜索网站

https://docs.ros.org/en/humble/index.html	ROS2 官方文档
https://www.ros.org/	ROS 官网
https://answers.ros.org/questions/	ROS 问答网站
https://index.ros.org/	ROS Index 是搜索 ROS 和 ROS 2 资源的入口点
https://github.com/	Github 官网，开源社区平台

② ROS2 特性

1. 开源、免费：ROS2 是一个开源的机器人操作系统框架，旨在支持机器人软件开发的标准化和共享。

2. 多机器人系统：ROS2 为多机器人系统的应用提供了标准方法和通信机制。

3. 跨平台：机器人应用场景不同，使用的控制平台也会有很大差异，为了让所有机器人都可以运行 ROS2，ROS2 可以跨平台运行于 Linux、Windows、MacOS、RTOS，甚至是没有任何系统的微控制器（MCU）上。

4. 实时性：机器人运动控制和行为策略要求机器人具备实时性，ROS2 为这样的实时性需求提供了保障。

5. 网络连接：ROS2 在各种网络环境下都能有效保障机器人大量数据的完整性和安全性。

6. 产品化：ROS2 不仅适用于机器人研发阶段，还可直接集成于产品中，满足消费市场需求，这要求 ROS2 具备高度的稳定性和强壮性。

7. 项目管理：机器人开发涉及多个复杂的工程环节，ROS2 提供了全面的项目管理工具和机制，从设计、开发、调试、测试到部署，都为用户提供了极大的便利，使机器人开发变得更加高效和流畅。

③ ROS2 组成系统

ROS2 组成体系整个 ROS 生态由通信（Plumbing）、工具（Tools）、功能（Capabilities）与社区（Community）四大部分组成。

图 1-1-2　ROS 系统组成部分

1. 通信(Plumbing)

通信是整个 ROS 系统的核心实现，它内置了一个高效的消息传递系统，通常称之为 middleware（中间件）或 Plumbing（管道）。这个系统负责在 ROS 节点之间传递消息，实现节点间的通信与协作。

2. 工具(Tools)

ROS 内置了一系列开发工具，包括 launch、调试、可视化、绘图、录制回放等功能。这些工具不仅提高了应用程序的开发效率，还可以在发布产品时直接集成到产品之中，为开发者提供了极大的便利。

3. 功能(Capabilities)

ROS2 以其强大的功能集合，为机器人开发者提供了全方位的支持。无论你需要的是高精度的传感器驱动程序，还是复杂的机器人运动规划算法，ROS2 都能为你提供。

4. 社区(Community)

ROS 社区规模庞大、多样且全球化，从学生和业余爱好者到跨国公司和政府机构，各行各业的人和组织都在推动着 ROS 项目的发展。

④ ROS 2 与 ROS1 的区别

1. 去中心化

在 ROS1 中系统依赖于 master 节点进行管理和调度，一旦 master 节点出现异常，整个系统将面临崩溃的风险。ROS2 采用了去中心化的设计，各节点之间无需通过 master 节点进行关联，实现了节点间的对等通信和相互发现，提高了系统的稳定性。

2. 全新通信底层实现

ROS2 遵循“不重复发明轮子”的原则，不再自行实现通信底层，而是直接采用了 DDS 通信机制。这使得 ROS2 在通信的实时性、可靠性和连续性方面相较于 ROS1 有了显著的提升。

3. 大量采用新技术、新的设计理念

随着 ROS 十数年的发展，众多新技术逐渐成熟并被广泛采用。ROS2 积极引入并应用了一些新技术和设计理念，以进一步提升系统的性能和稳定性。

4. 应用场景更为广泛

ROS1 在设计之初就存在一定的局限性，例如对多机器人编队支持不佳、对小型嵌入式平台支持不够理想、实时性能较差、数据传输受网络质量影响较大以及产品化难度较大等。ROS2 的推出在很大程度上解决了这些问题，使得其应用场景更为广泛。

⑤ ROS2 的应用方向

许多 ROS 团队伴随 ROS 成长到今日，其规模已经发展到足以被认为是独立组织的程度了。在导航、机械臂、无人驾驶、无人机等诸多领域大放异彩，下面列出了其中的一些团队项目，这些项目对我们以后的进阶发展，也提供了指导。

1. NAV2 （https://navigation.ros.org/）

Nav2 项目继承自 ROS Navigation Stack。该项目旨在可以让移动机器人从 A 点安全的移动到 B 点。它也可以应用于涉及机器人导航的其他应用，例如跟随动态点。Nav2 将用于实现路径规划、运动控制、动态避障和恢复行为等一系列功能。

图 1-1-3 NAV2 导航

2. OpenCV （https://opencv.org/）

OpenCV（Open Source Computer Vision Library）是一个开源的计算机视觉和机器学习软件库。OpenCV 旨在为计算机视觉应用程序提供通用基础架构，并加速机器感知在商业产品中的使用。OpenCV 允许企业轻松地使用和修改代码。

图 1-1-4 OpenCV 视觉识别

3. Moveit2 （https://moveit.ros.org/）

Moveit2 是一组 ROS2 软件包，主要包含运动规划、碰撞检测、运动学、3D

感知、操作控制等功能。它可以用于构建机械臂的高级行为。Moveit2 现在可以用于市面上的大多数机械臂，并被许多大公司使用。

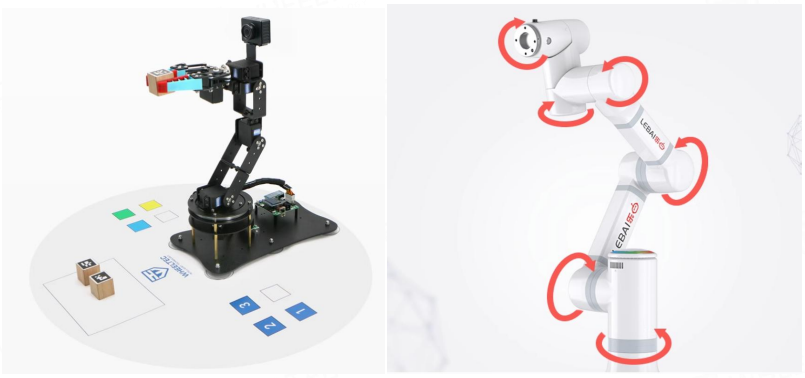


图 1-1-5 Moveit2 机械臂

4. The Autoware Foundation （https://autoware.org/）

Autoware Foundation 是 ROS 下属的非营利组织，支持实现自动驾驶的开源项目。Autoware 基金会在企业发展和学术研究之间创造协同效应，为每个人提供自动驾驶技术。

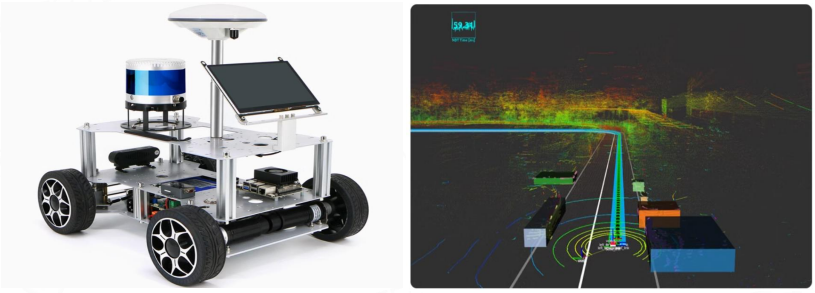


图 1-1-6 autoware 导航

5. microROS （https://micro.ros.org/）

在基于 ROS 的机器人应用中，micro-ROS 正在弥合性能有限的微控制器和一般处理器之间的差距。micro-ROS 可以在各种嵌入式硬件上运行，使 ROS 能直接应用于机器人硬件。

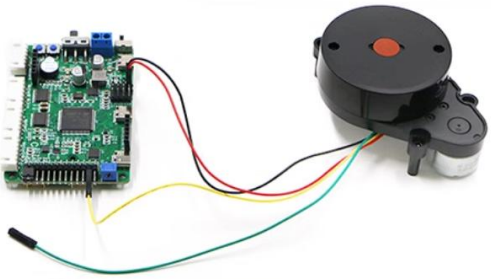


图 1-1-7 microROS 嵌入式控制

6. Open Robotics (https://www.openrobotics.org/)

Open Robotics 与全球 ROS 社区合作，为机器人创建开放的软件和硬件平台，包括 ROS1、ROS2、Gazebo 模拟器和 Ignition 模拟器。Open Robotics 使用这些平台解决一些重要问题，并通过为各种客户组织提供软件和硬件开发服务来帮助其他人做同样的事情。

7. PX4（https://px4.io/）

PX4 是一款用于无人机和其他无人驾驶车辆的开源飞行控制软件。该项目为无人机开发人员提供了一套灵活的工具，用于共享技术并为无人机应用程序创建量身定制解决方案。

图 1-1-8 无人机飞控

8. ROS-Industrial（https://rosindustrial.org/）

ROS-Industrial 是一个开源项目，将 ROS 软件的高级功能扩展到工业相关硬件和应用程序。

图 1-1-9 工业机器人

1.2 ROS2 体系框架

① ROS2 体系框架概览

功能包（SDK）是 ROS2 应用程序的核心，但是功能包不能直接构建，必须依赖于工作空间，一个 ROS2 工作空间的目录结构如下：

```
WorkSpace --- 自定义的工作空间。
    |--- build: 存储中间文件的目录，该目录下会为每一个功能包创建一个单独子目录。
    |--- install: 安装目录，该目录下会为每一个功能包创建一个单独子目录。
    |--- log: 日志目录，用于存储日志文件。
    |--- src: 用于存储功能包源码的目录。
        |-- C++功能包
            |-- package.xml: 包信息，比如:包名、版本、作者、依赖项。
            |-- CMakeLists.txt: 配置编译规则，比如源文件、依赖项、目标文件。
            |-- src: C++源文件目录。
            |-- include: 头文件目录。
            |-- msg: 消息接口文件目录。
            |-- srv: 服务接口文件目录。
            |-- action: 动作接口文件目录。
        |-- Python 功能包
            |-- package.xml: 包信息，比如:包名、版本、作者、依赖项。
            |-- setup.py: 与 C++功能包的 CMakeLists.txt 类似。
            |-- setup.cfg: 功能包基本配置文件。
            |-- resource: 资源目录。
            |-- test: 存储测试相关文件。
            |-- 功能包同名目录: Python 源文件目录。
```

② ROS2 工作空间

在 ROS 机器人开发中，我们针对机器人某些功能进行开发时，各种编写的代码、参数、脚本等文件，需要放置在某一个文件夹里进行管理，这个文件夹在 ROS 系统中就叫做工作空间。所以工作空间是一个存放项目开发相关文件的文件夹，也是开发过程中存放所有资料的大本营。

如何创建一个工作空间呢？

```
# 选择一个存储的目录，也可以放在根目录
$ mkdir -p ~/ros2_ws/src
$ cd ros2_ws
编译工作空间
$ colcon build
```

当我们在终端下执行 ROS2 程序时，需要先配置相关的工作空间环境：

```
$ source /opt/ros/humble/setup.bash
```

对于常用的工作空间，我们可以将配置环境指令写入 ”~/.bashrc“ 文件，那么每次启动终端时，不需要在手动配置环境

```
$ echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

③ ROS2 功能包

机器人具备众多功能，像移动控制、视觉感知、自主导航等等。为了降低这些功能之间的耦合度，将不同功能的代码分别组织到各自的功能包中，这样做不仅提高了代码的模块化程度，也使得功能更加清晰、易于维护。当我们需要在 ROS 社区中分享特定功能包时，只需详细说明其使用方法，其他开发者便能迅速将其集成到自己的项目中，从而大大提升了 ROS 中软件的复用效率。因此，功能包的机制无疑是提高 ROS 软件复用率的重要策略之一。

图 1-2-1　WHEELTEC 小车相关功能

④ ROS2 节点

在 ROS2 通信过程中，不论采用何种方式，通信对象的构建都依赖于节点(Node)，一般情况下每个节点都对应某一单一的功能模块(例如：雷达驱动节点可能负责发布雷达消息，摄像头驱动节点可能负责发布图像消息)。一个完整的机器人系统可能由许多协同工作的节点组成，ROS2 中的单个可执行文件可以包含一个或多个节点。

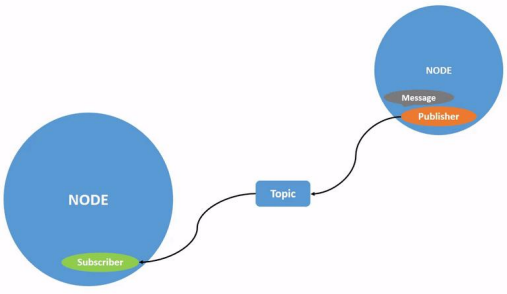


图 1-2-2　ROS2 话题通信

⑤ ROS2 话题

话题（Topic）在 ROS2 中扮演着至关重要的角色，它作为一个纽带，将相同话题的节点关联在一起，构成了通信的基础。ROS2 的跨语言特性使得不同节点可以采用不同的编程语言实现，然而这种语言差异并不会阻碍节点间的数据交互，只要它们使用了相同的话题，就能轻松实现信息的传递与共享。这种灵活性不仅增强了 ROS2 的通用性和可扩展性，也为其在复杂系统中的广泛应用提供了有力支持。

⑥ ROS2 接口

在通信过程中，需要传输数据，就必然涉及到数据载体，也即要以特定格式传输数据。在 ROS2 中，数据载体称之为接口(interfaces)。通信时使用的数据载体一般需要使用接口文件定义。常用的接口文件有三种：msg 文件、srv 文件与 action 文件。

⑦ colcon build 编译方式

colcon 是一个命令行工具，用于改进编译，测试和使用多个软件包的工作流程。它实现过程自动化，处理需求并设置环境以便于使用软件包。ROS2 中便是使用 colcon 作为包构建工具的。

可以在工作空间的路径下，执行下面的命令编译所有功能包：

```
$ colcon build
```

--packages-select，此参数用于仅构建单个包（或指定的多个包），而不考虑其他包的依赖关系。

```
$ colcon build --packages-select  <package1>  <package2>
```

2. ROS2 通信工具的使用

2.1 ROS2 通信机制简介

在 ROS2 中通信方式有多种，但是不同通信方式的组成要素都是类似的，比如：通信是双方或多方行为、通信时都需要将不同的通信对象关联、都有各自的模型、交互数据时也必然涉及到数据载体等等。

① 通信模型简介

不同的通信对象通过话题关联到一起之后，以何种方式实现通信呢？在 ROS2 中，常用的通信模型有四种：

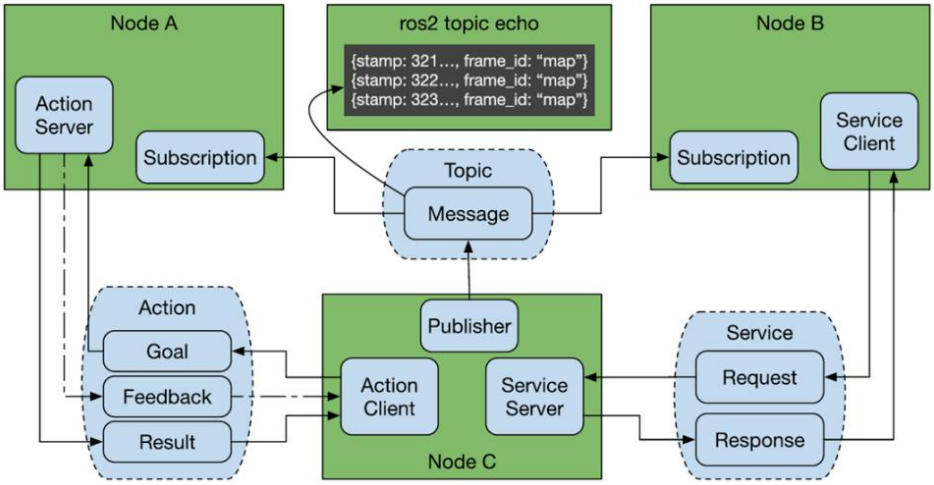


图 2-1-1　ROS2 通信模型示例

1.话题通信：是一种单向通信模型，在通信双方中，发布方发布数据，订阅方订阅数据，数据流单向的由发布方传输到订阅方。话题通信的发布方与订阅方是一种多对多的关系。

2.服务通信：是一种基于请求响应的通信模型，在通信双方中，客户端发送请求数据到服务端，服务端响应结果给客户端。服务端与客户端是一对多的关系。

3. 动作通信：是一种带有连续反馈的通信模型，在通信双方中，客户端发送请求数据到服务端，服务端响应结果给客户端，但是在服务端接收到请求到产生最终响应的过程中，会发送连续的反馈信息到客户端。

4. 参数服务：是一种基于共享的通信模型，在通信双方中，服务端可以设置数据，而客户端可以连接服务端并操作服务端数据。

② 通信接口简介

通信接口是 ROS2 在通信过程中所使用的核心载体，它承载着信息的传递与交互。在 ROS2 生态系统中，已经确立了一系列统一的接口标准，这些标准不仅降低了程序之间的依赖关系，同时也方便了之后代码的复用。当然，在自主设计软件程序时，我们也可以根据实际需求设计自定义的通信接口。

在这里我们可以用激光雷达的来举例，不同的厂家生产出不同的类型的激光雷达，每种雷达驱动方式、扫描速率等等都不相同。当机器人进行导航时，需要使用激光雷达的扫描数据来进行通信，假如没有统一的接口，每次更换一个种类的雷达，都需要重新做程序适配。在 ROS2 中定义了一个统一的接口叫做 sensor_msgs/msg/LaserScan,现在几乎每个雷达的厂家都会编写程序将自己雷达的数据变成 sensor_msgs/msg/LaserScan 格式，提供给用户使用。

在 ROS2 中，常用的接口文件主要有三种类型：msg 文件、srv 文件和 action 文件。

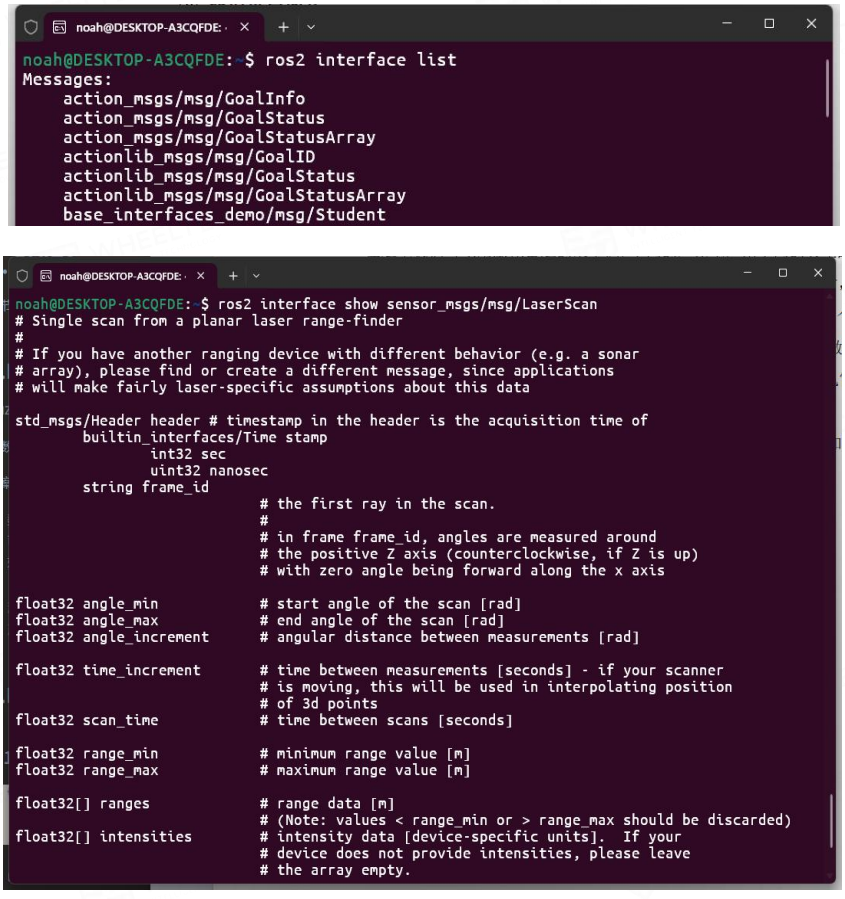


图 2-1-2 ROS2 接口的使用

2.2 ROS2 命令行工具

ROS2 中常用的命令如下：

```
ros2 pkg: 功能包管理工具
ros2 run: 运行功能包节点程序
ros2 node: 节点相关命令行工具
ros2 topic: 话题通信相关的命令行工具
ros2 interface: 接口(msg、srv、action)消息相关的命令行工具
ros2 service: 服务通信相关的命令行工具
ros2 action: 动作通信相关的命令行工具
ros2 param: 参数服务相关的命令行工具
```

利用 ROS2 的命令行工具，开发者可以高效地测试运行状态和软件功能，为机器人软件的开发与调试提供极大的便利。在使用过程中可以通过命令 -h 或命令 --help 的方式查看命令帮助文档，用法：ros2 node -h 或 ros2 node list --help。

1. ros2 pkg

```
基本用法:
create      创建一个功能包。
格式:  ros2 pkg create --build-type <编译类型> <功能包名> <依赖 1> <依赖 2>
$ ros2 pkg create  --build-type ament_python  <pkg_name>  rclpy std_msgs
sensor_msgs
executables   输出特定包的可执行文档列表
list          输出可用包列表
prefix        输出一个包的前缀路径
xml          输出软件包清单或特定标签的 XML
```

2. ros2 run

```
命令格式: $ ros2 run pkg_name node_name
```

3. ros2 node

```
基本用法:
info  输出节点信息
list  输出运行中的节点的列表
```

4. ros2 topic

```
基本用法:
bw        输出话题消息传输占用的带宽
delay     输出带有 header 的话题延迟
echo      输出某个话题下的消息
find      根据类型查找话题
hz        输出消息发布频率
info      输出话题相关信息
list      输出运行中的话题列表
pub       向指定话题发布消息
```

```
type      输出话题使用的接口类型 ros2 interface
```

5. ros2 interface

```
基本用法:
list      输出所有可用的接口消息
package   输出指定功能包下的
packages  输出包含接口消息的功能包
proto     输出接口消息原型
show      输出接口消息定义格式
```

6. ros2 service

```
基本用法:
call  向某个服务发送请求
find  根据类型查找服务
list  输出运行中的服务列表
type  输出服务使用的接口类型
```

7. ros2 action

```
基本用法:
info       输出指定动作的相关信息
list       输出运行中的动作的列表
send_goal  向指定动作发送请求
```

8. ros2 param

```
基本用法:
delete    删除参数
describe  输出参数的描述信息
dump      将节点参数写入到磁盘文件
get       获取某个参数
list      输出可用的参数的列表
load      从磁盘文件加载参数到节点
set       设置参数
```

2.3 RVIZ2 数据可视化工具

① RVIZ2 简介

ROS2 内置了强大的数据可视化工具 RVIZ2，它能够以图形化的方式直观地展现机器人系统中的抽象数据。借助 RVIZ2，我们能够形象、直观地观察到传感器数据的真实样态，这不仅有助于我们在调试过程中快速定位问题，还能更直观地理解和修改系统的运行状态。

② RVIZ2 的使用

可以在终端输入下面的指令启动 rviz2：

```
$ rviz2
```

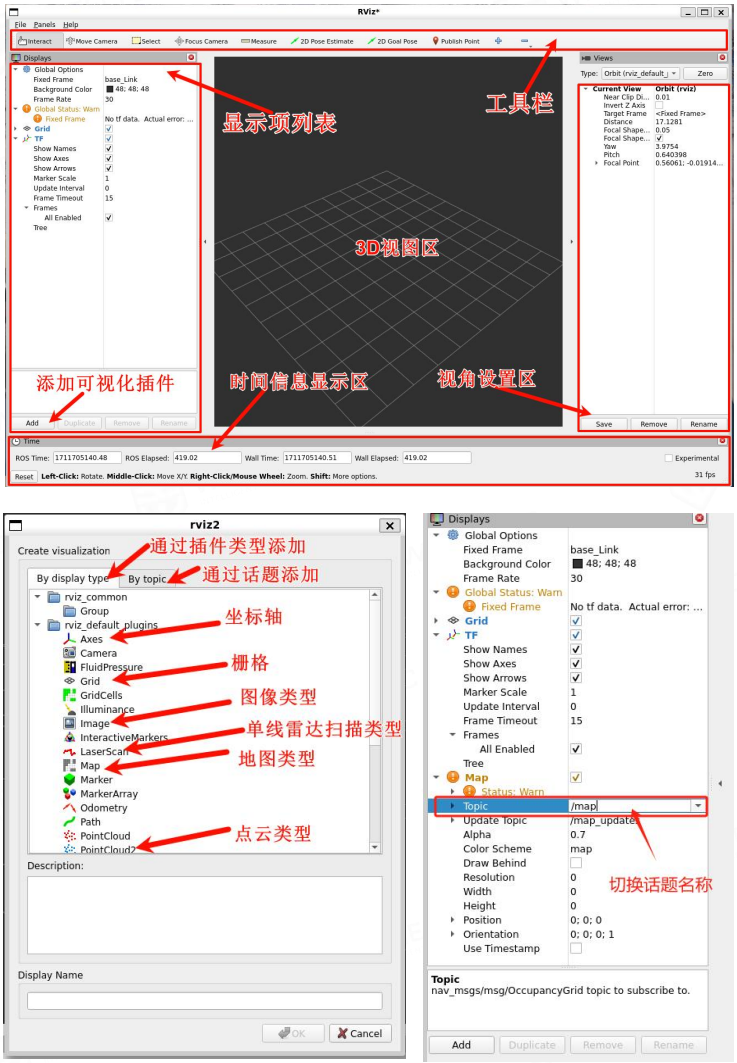


图 2-3-1　RVIZ2 界面说明

2.4 RQT 工具箱

① RQT 简介

RQT 是基于 Qt 开发的一个 GUI（用户界面）框架，以插件的形式实现了各种 GUI 工具，方便开发者进行可视化操作、调试和监控。通过 RQT 工具，开发者可以更加高效地开发和调试 ROS2 应用。

② RQT 的使用

启动 RQT 的命令方式有两种：

```
方式 1:  $ rqt
方式 2:  $ ros2 run rqt_gui rqt_gui
```

启动 rqt 之后，需要通过 plugins 添加所需的插件：

1. Node Graph（节点图）

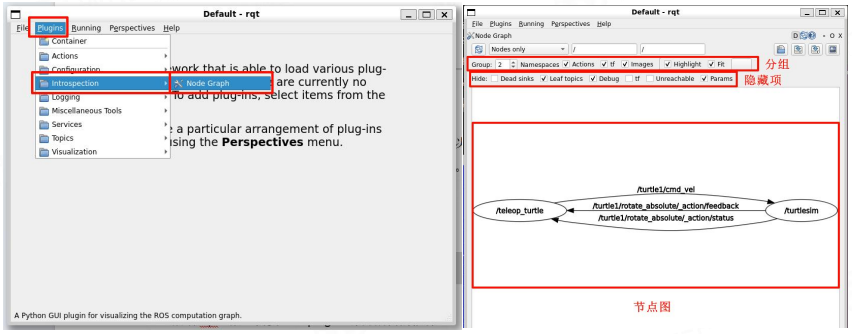


图 2-4-1　rqt 中的 Node Graph

2. Topic Monitor（话题监视器）

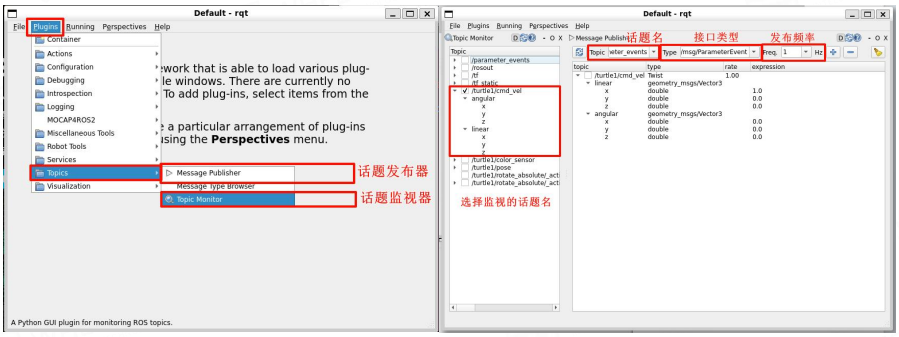


图 2-4-2　rqt 中的 Topic Monitor

3. Image View（图像查看器）

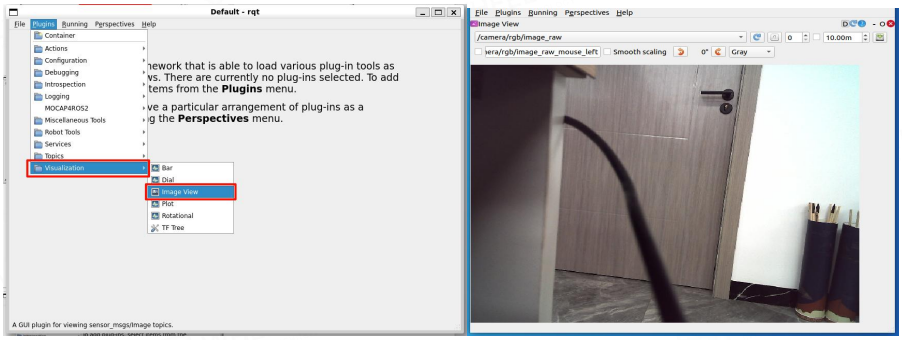


图 2-4-3　rqt 中的 Image View

4. TF Tree（TF 树）

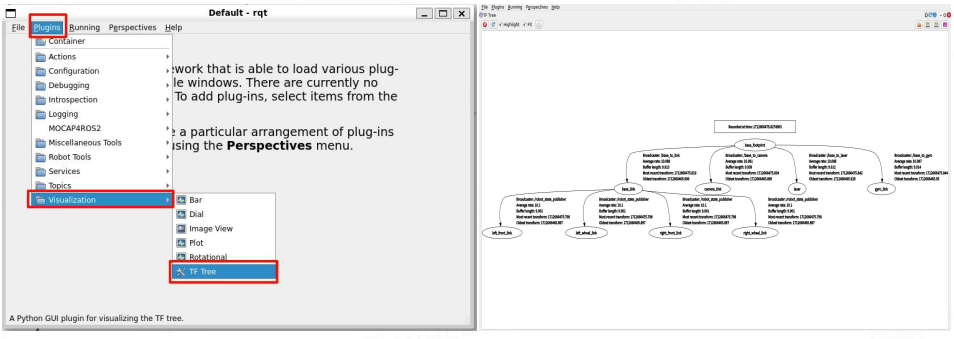


图 2-4-4　rqt 中的 TF tree

2.5 ros2 bag 包管理工具

① ros2 bag 简介

在 ROS2 中 bag 是一个用于记录和回放机器人数据的工具。它允许用户捕获传感器数据、节点之间的消息传递等，并将其保存为 bag 文件，以便后续分析和处理。bag 工具在机器人系统的开发、调试和测试过程中发挥着重要作用。

② ros2 bag 的使用

```
基本用法:
  convert  给定一个 bag 文件，写出一个新的具有不同配置的 bag 文件;
  info     输出 bag 文件的相关信息;
  list     输出可用的插件信息;
  play     回放 bag 文件数据;
  record   录制 bag 文件数据;
  reindex  重建 bag 的元数据文件。
```

1. 录制 bag 文件：

```
# 记录单个话题
$ ros2 bag record /topic-name
# 记录多个话题
$ ros2 bag record topic-name1 topic-name2
# 记录所有话题
$ ros2 bag record -a
```

2. 查看 bag 文件相关信息

```
# 假设录制的 file 为 rosbag2_lidar
$ ros2 bag info rosbag2_lidar
```

3. 播放并查看相关数据

```
# 播放录制的数据包
$ ros2 bag play rosbag2_lidar
# 查看录制的话题数据
$ ros2 topic echo /topic-name
# 倍速播放   -r 10   就是 10 倍速播放
$ ros2 bag play rosbag2_lidar -r 10
# 循环播放 -l
$ ros2 bag play rosbag2_lidar -l
# 播放单个话题
$ ros2 bag play rosbag2_lidar --topics /topic-name
```

3. ROS2 通信机制的实现

3.1 ROS2 话题通信

① 话题通信的使用场景

话题通信是 ROS 中使用频率最高的一种通信模式，话题通信是基于发布订阅模式的，也即：一个节点发布消息，另一个节点订阅该消息。话题通信一般应用于不断更新的、少逻辑处理的数据传输场景。

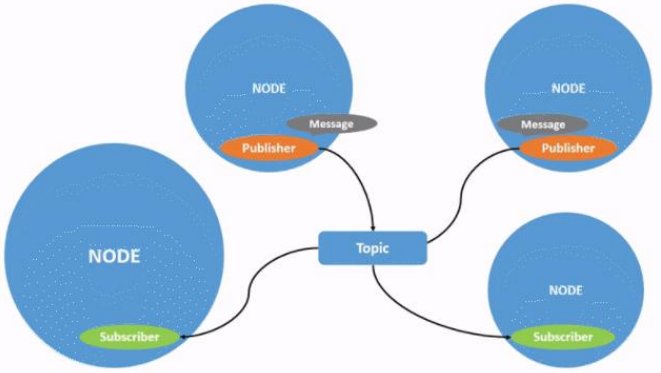


图 3-1-1　多节点话题通信

② 话题通信的实现（c++）

1. 创建功能包

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_cmake cpp_pubsub
//也可以直接在创建功能包时添加依赖关系
$ ros2 pkg create --build-type ament_cmake cpp_pubsub --dependencies rclcpp
std_msgs
```

2. 话题通信的发布方实现（c++）

进入 cpp_pubsub 功能包的 src 文件夹，新建 publisher_member_function.cpp

```
/* 实现：以某个固定频率发送文本“hello world!”，文本后缀编号，每发送一条消息，编号递增 1。 */
// 1.包含头文件;
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
// 方便时间的表示
using namespace std::chrono_literals;
// 3.定义节点类;  继承 rclcpp::node 创建节点类
class MinimalPublisher : public rclcpp::Node
{
```

```
  public:
    //公共构造函数将节点命名，并初始化参数
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
      // 3-1.创建发布方;
      publisher_ =
            this->create_publisher<std_msgs::msg::String>("topic", 10);
      // 3-2.创建定时器;
      timer_ = this->create_wall_timer(500ms,
                    std::bind(&MinimalPublisher::timer_callback, this));
    }
  private:
    // 定义定时器回调函数
    void timer_callback()
    {
      // 3-3.组织消息并发布。
      auto message = std_msgs::msg::String();
      message.data = "Hello, world! " + std::to_string(count_++);
      RCLCPP_INFO(this->get_logger(), "发布的消息: '%s'",
                  message.data.c_str());
      publisher_->publish(message);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    size_t count_;
};
int main(int argc, char * argv[]){
  // 2.初始化 ROS2 客户端;
  rclcpp::init(argc, argv);
  // 4.调用 spin 函数，并传入节点对象指针。
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  // 5.释放资源;
  rclcpp::shutdown();
  return 0;
}
```

3. 新建 subscriber_member_function.cpp

```
/* 实现：订阅发布方发布的消息，并输出到终端。*/
// 1.包含头文件;
#include "rclcpp/rclcpp.hpp"
// 占位符，代替回调函数中的第一个参数
#include "std_msgs/msg/string.hpp"using std::placeholders::_1;
// 3.定义节点类；继承 rclcpp:: node 创建节点类
class MinimalSubscriber : public rclcpp::Node
```

```
{
  public:
    // 公共构造函数将节点命名
    MinimalSubscriber()
    : Node("minimal_subscriber")
    {
      // 3-1.创建订阅方;
      subscription_ =
        this->create_subscription<std_msgs::msg::String>("topic", 10,
        std::bind(&MinimalSubscriber::topic_callback, this, _1));
    }
  private:
    // 3-2.处理订阅到的消息;
    void topic_callback(const std_msgs::msg::String & msg) const
    {
      // 打印话题消息的字符串信息
      RCLCPP_INFO(this->get_logger(), "订阅的消息: '%s'", msg.data.c_str());
    }
    // 订阅者字段的声明
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};
int main(int argc, char * argv[]){
  // 2.初始化 ROS2 客户端;
  rclcpp::init(argc, argv);
  // 4.调用 spin 函数, 并传入节点对象指针。
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  // 5.释放资源;
  rclcpp::shutdown();
  return 0;
}
```

4. 编辑配置文件

在 C++功能包中，配置文件主要是 package.xml 与 CMakeLists.txt。

<package.xml>

添加依赖项，在<depend>标签下面添加源文件所需的依赖，配置内容如下：

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

<CMakeLists.txt>

首先添加搜索库，在 find_package 下面添加搜索源文件所需的库：

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

然后，还需要我们添加生成可执行文件的源文件链接以及依赖关系：

```
# 将源文件命名为 talker 可执行文件
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)
# 将源文件命名为 listener 可执行文件
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)
```

最后，增加可执行文件生成的路径，这样方便我们找到可执行文件：

```
install(TARGETS
  talker
  listener
  DESTINATION lib/${PROJECT_NAME})
```

5. 编译和运行

首先重新打开一个终端进入工作空间目录

```
$ cd ~/ros2_ws
```

编译指定的功能包：

```
$ colcon build --packages-select cpp_pubsub
```

打开新终端，分别运行话题订阅和发布节点

```
$ ros2 run cpp_pubsub listener
$ ros2 run cpp_pubsub talker
```

③ 话题通信的实现（python）

1. 创建功能包

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_python py_pubsub
```

2. 话题通信的发布方实现（python）

进入 py_pubsub 功能包的 py_pubsub，新建 publisher_member_function.py

```
    """
    需求：以某个固定频率发送文本“hello world!”，文本后缀编号，每发送一条消息，编号递增 1。
    步骤：
        1.导包；
        2.初始化 ROS2 客户端；
        3.定义节点类；
            3-1.创建发布方；
            3-2.创建定时器；
            3-3.组织消息并发布。
        4.调用 spin 函数，并传入节点对象；
        5.释放资源。
"""
# 1.导包；
```

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
# 3.定义节点类;
class MinimalPublisher(Node):
    #订阅器节点类
    def __init__(self):
        # 初始化节点
        super().__init__('minimal_publisher_py')
        # 3-1.创建发布方;
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        # 3-2.创建定时器;
        timer_period = 0.5
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0
    # 3-3.组织消息并发布。
    def timer_callback(self):
        # 定时器回调函数
        msg = String()
        # 打印并发布字符串附加计数器值的信息
        msg.data = 'Hello World(py): %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('发布的消息: "%s"' % msg.data)
        self.i += 1
def main(args=None):
    # 2.初始化 ROS2 客户端;
    rclpy.init(args=args)
    # 4.调用 spin 函数, 并传入节点对象;
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    # 5.释放资源。
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

3. 话题通信的订阅方实现（python）

新建 subscriber_member_function.py 话题订阅源文件

```
"""
    需求: 订阅发布方发布的消息, 并输出到终端。
    步骤:
        1.导包;
        2.初始化 ROS2 客户端;
        3.定义节点类;
```

```
            3-1.创建订阅方;
            3-2.处理订阅到的消息。
        4.调用 spin 函数，并传入节点对象;
        5.释放资源。
"""
# 1.导包;
import rclpyfrom rclpy.node import Nodefrom std_msgs.msg import String
# 3.定义节点类;
class MinimalSubscriber(Node):
    # 订阅器节点类
    def __init__(self):
        # 初始化节点
        super().__init__('minimal_subscriber_py')
        # 3-1.创建订阅方;
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        # 防止未使用变量警告
        self.subscription
    # 3-2.处理订阅到的消息。
    def listener_callback(self, msg):
        # 打印订阅话题的消息数据
        self.get_logger().info('订阅的消息: "%s"' % msg.data)
def main(args=None):
    # 2.初始化 ROS2 客户端;
    rclpy.init(args=args)
    # 4.调用 spin 函数，并传入节点对象;
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    # 5.释放资源。
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

4. 编辑配置文件

在 Python 功能包中，配置文件主要关注 package.xml 与 setup.py。

<package.xml>添加依赖关系，在<depend>标签下面添加源文件所需的依赖，配置内容如下：

```
<depend>rclpy</depend>
<depend>std_msgs</depend>
```

<setup.py>添加入口点。entry_points 字段的 console_scripts 中添加如下内容：

```
    entry_points={
        'console_scripts': [
            # 将源文件命名为 talker 可执行文件
            'talker = py_pubsub.publisher_member_function:main',
            # 将源文件命名为 listener 可执行文件
            'listener = py_pubsub.subscriber_member_function:main',
        ],
    },
```

5. 编译和运行

进入工作空间目录

```
$ cd ~/ros2_ws
```

编译：

```
$ colcon build --packages-select py_pubsub
```

打开新的终端，分别运行订阅和发布节点：

```
$ ros2 run py_pubsub listener
$ ros2 run py_pubsub talker
```

3.2 ROS2 服务通信

① 服务通信的使用场景

服务通信也是 ROS 中一种极其常用的通信模式，服务通信是基于请求响应模式的，是一种应答机制。服务通信用于偶然的、对实时性有要求、有一定逻辑处理需求的数据传输场景。

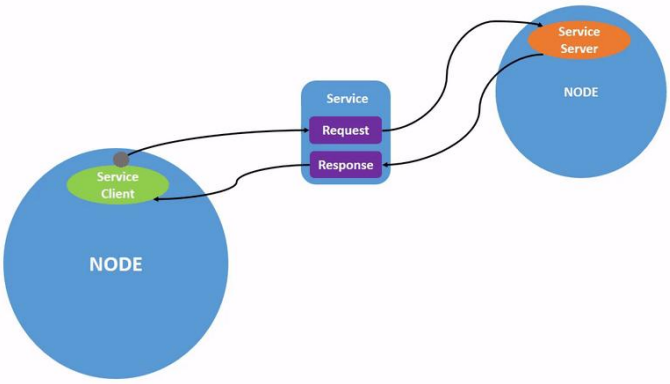


图 3-2-1　服务通信示意图

② 服务通信的使用

我们可以通过命令行的方式调用各种服务，首先查询当前有哪些服务，打开终端输入：

```
#启动小海龟仿真器
$ ros2 run turtlesim turtlesim_node
#查询服务列表
$ ros2 service list
```

然后通过命令进行调用：

```
#查询服务列表
$ ros2 service call /spawn  turtlesim/srv/Spawn  "{x: 1.0,y: 2.0,theta: 0.3,name: turtle2}"
```

3.3 ROS2 动作通信

① 动作通信的使用场景

动作通信的使用场景更适用于耗时的请求响应场景，用以获取连续的状态反馈。就结构而言动作通信由目标、反馈和结果三部分组成；就功能而言动作通信类似于服务通信，动作客户端可以发送请求到动作服务端，并接收动作服务端响应的最终结果，不过动作通信可以在请求响应过程中获取连续反馈，并且也可以向动作服务端发送任务取消请求。

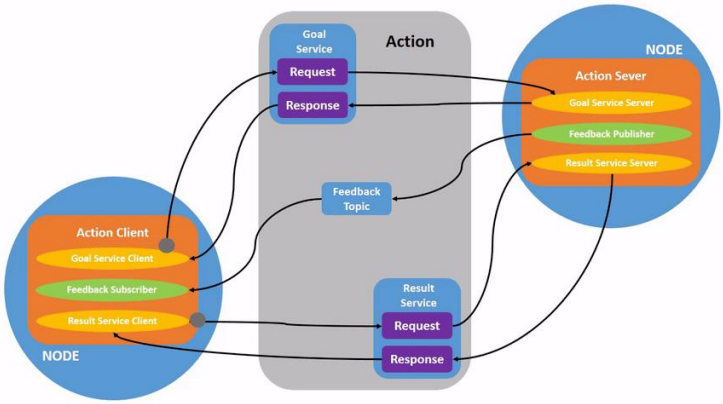


图 3-3-1　动作通信示意图

② 动作通信的使用

我们可以通过命令行的方式调用各种动作，首先查询当前有哪些动作，打开终端输入：

```
#启动小海龟仿真器
$ ros2 run turtlesim turtlesim_node
#查询服务列表
$ ros2 action list
```

然后通过命令进行调用：

```
    #查询服务列表
    $ ros2 action send_goal /turtle1/rotate_absolute
turtlesim/action/RotateAbsolute  theta:\ 0.0
```

3.4 ROS2 参数服务

① 参数服务的使用场景

参数服务是以共享的方式实现不同节点之间数据交互的一种通信模式。保存参数的节点称之为参数服务端，调用参数的节点称之为参数客户端。参数服务保存的数据类似于编程中“全局变量”的概念，可以在不同的节点之间共享数据。

② 参数服务的使用

在小海龟的例程中，仿真器也提供了不少参数，我们一起来通过这个例程，熟悉下参数的含义和命令行的使用方法。首先查询当前参数列表，打开终端输入：

```
    #启动小海龟仿真器
    $ ros2 run turtlesim turtlesim_node
    #查看参数列表
    $ ros2 param list
    # 查看某个参数的描述信息
    $ ros2 param describe turtlesim background_b
    # 查询某个参数的值
    $ ros2 param get turtlesim background_b
    # 修改某个参数的值
    $ ros2 param set turtlesim background_b 10
```

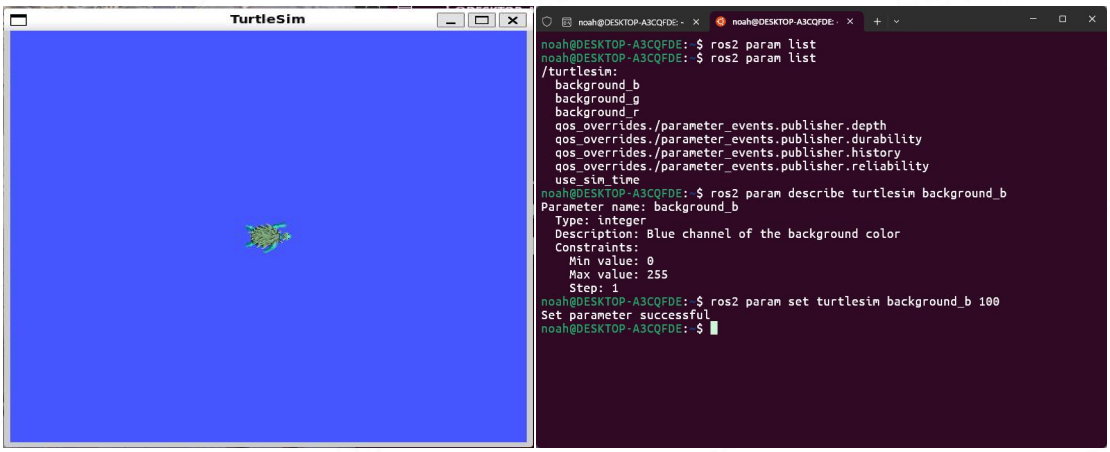


图 3-4-1　参数服务示意图

4. ROS2 工具的使用

4. 1 Launch 启动管理工具

① Launch 简介

对于一个机器人系统而言，往往需要同时启动多个节点，并且根据应用场景和机器人特性的不同，每个节点都需要特定的配置。

ROS2 设计了一套完整的语法和规则的文件来帮助我们组织节点的启动，launch 文件是多节点启动与配置的一种脚本，它可以让我们同时启动和配置多个包含 ROS 2 节点的可执行文件。

ROS2 的 launch 文件格式有 xml、yaml 和 python 格式，相较于 xml 和 yaml，Python 是一种脚本编程语言，更加的灵活，我们可以借助 Python 的特性在启动节点时实现更复杂的逻辑控制、条件判断以及参数化配置。Python 的丰富库和强大的语法特性也让我们能够更轻松地处理各种启动场景，提升机器人系统的可维护性和可扩展性。

② Launch 的使用

一般来说，我们在使用机器人的各种功能或者启动各种硬件传感器时，其相应的启动文件已经编写好了，我们只需要按照功能说明直接运行就可以了。

示例：

```
# 以启动小车底层节点为例，   ros2 launch <功能包名> <launch 文件名>
$ ros2 launch turn_on_wheeltec_robot turn_on_wheeltec_robot.launch.py
```

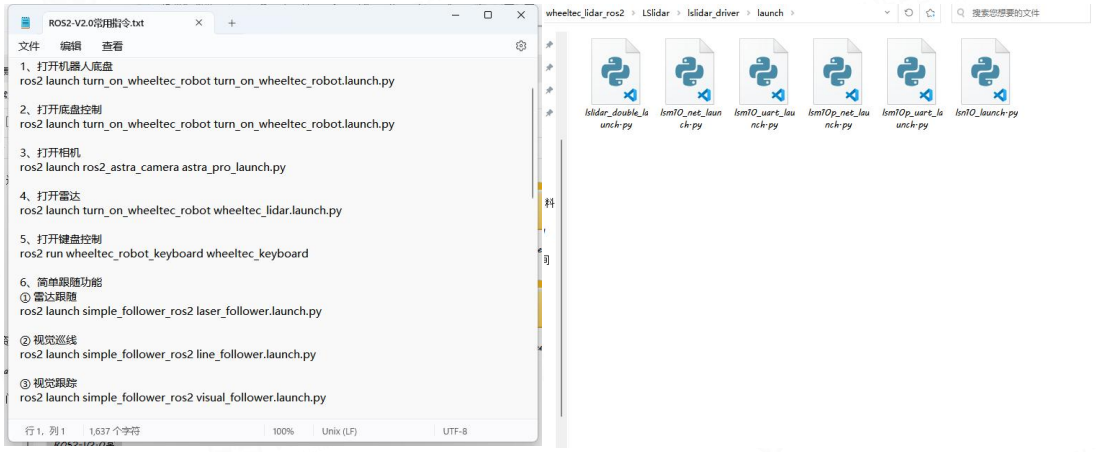


图 4-1-1 启动相关功能的 launch 文件

③ Launch 文件的 Python 实现

我们在使用 python 语言编写 ROS2 launch 文件的过程中，是把每个节点、文件、脚本等抽象成一个 action，用统一的接口来启动，主要结构是：

```
def generate_launch_description():
   return LaunchDescription([
      action_1,
      action_2,
      ...,
      action_n
   ])
```

下面我们来具体看一个 Launch 文件的实现案例：

```
# 导入相关功能包
import os
from launch import LaunchDescription
from launch_ros.actions import Node
#文件包含相关功能
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from pathlib import Path
# 获取功能包下 share 目录路径
from ament_index_python.packages import get_package_share_directory
#生成 launch 文件描述
def generate_launch_description():
    # 获取 launch 文件所在目录
    launch_dir = os.path.join(
        get_package_share_directory('turn_on_wheeltec_robot'),
        'launch'
        )
    imu_config = Path(get_package_share_directory('turn_on_wheeltec_robot'),
'config', 'imu.yaml')
    # 包含相关的 launch 文件
    wheeltec_robot = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                os.path.join(launch_dir,
                'base_serial.launch.py')),
            launch_arguments={'akmcar': 'false'}.items(),
            )
    choose_car = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                os.path.join(launch_dir,
                'robot_mode_description.launch.py')),
                )
```

```
        # 启动功能包中的可执行文件 node
        imu_filter_node =  launch_ros.actions.Node(
            #功能包名称
            package='imu_filter_madgwick',
            #可执行文件 node 名
            executable='imu_filter_madgwick_node',
            #相关参数
            parameters=[imu_config]
        )

        # 返回需要执行的 action 功能
        return LaunchDescription([
            wheeltec_robot,
            choose_car,
            imu_filter_node
            ])
```

4.2 TF2 坐标变换工具

① TF2 简介

tf2 (TransForm Frame 2)是指坐标变换，在移动机器人系统中，不同传感器和执行器可能位于不同的坐标系中，因此需要进行坐标变换以统一数据表示。TF2 提供了一套灵活且强大的工具，用于管理和查询这些坐标系之间的变换关系。

完整的坐标变换实现由坐标变换广播方和坐标变换监听方两部分组成。每个坐标变换广播方一般会发布一组坐标系相对关系，而坐标变换监听方则会将多组坐标系相对关系融合为一棵坐标树。

ROS 中的坐标变换是基于右手坐标系的。

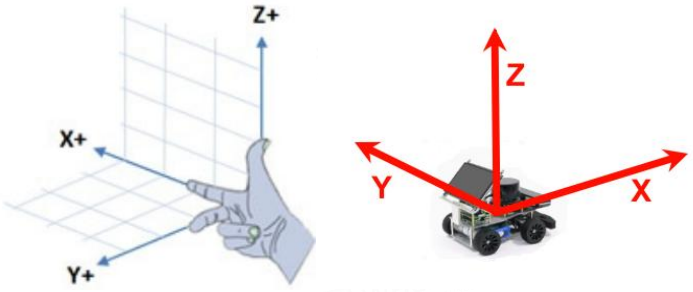


图 4-2-1 ROS2 基础坐标系

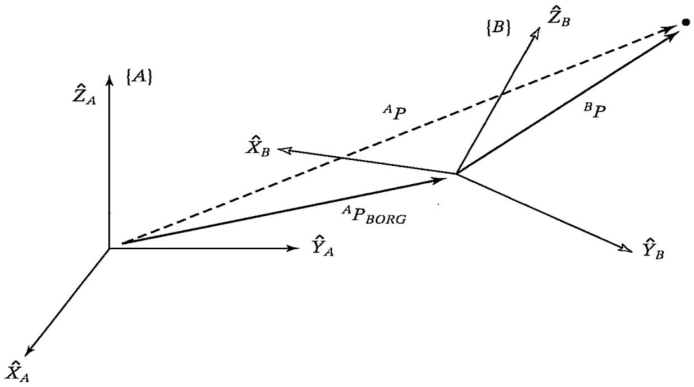


图 4-2-2　TF 坐标变换示意图

关于坐标系变换关系，可以分解为平移和旋转两个部分，两者之间的变换关系，其实是向量的数学描述在空间中画出的坐标系。

② TF2 静态坐标变换的发布

所谓静态坐标变换，是指两个坐标系之间的相对位置是固定的。如雷达和 base_link 之间的位置是固定的。

TF2 静态坐标变换命令行发布示例：

```
    # 发布 A 到 B 的位姿
    # 格式: ros2 run tf2_ros static_transform_publisher <x 轴偏移量> <y 轴偏移量> <z 轴偏移量> <偏航角> <俯仰角> <横滚角> <父级坐标系> <子级坐标系>
    $ ros2 run tf2_ros static_transform_publisher 0 0 3 0 0 3.14 A B
    # 监听/获取 TF 关系
    $ ros2 run tf2_ros tf2_echo A B
    # rviz2 可视化
    $ rviz2
```

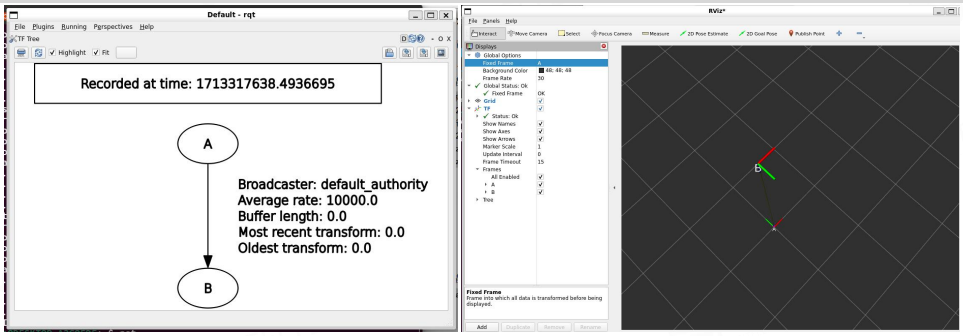


图 4-2-3　TF 坐标变换示意图

TF2 静态坐标变换 launch 文件发布示例：

```
# 在 launch 文件中添加如下 action 可以发布 base_footprint 到 laser 的 TF 坐标变换
launch_ros.actions.Node(
        package='tf2_ros',
        executable='static_transform_publisher',
        name='base_to_laser',
        arguments=['0.048 ', '0', '0.18','0', '0', '0',
                   'base_footprint','laser'],)
```

4.3 URDF 模型工具

① URDF 简介

在 ROS 2 中，URDF 文件扮演着至关重要的角色，URDF 可以清晰描述机器人自身的模型，还可以描述机器人的外部环境。通过 URDF 文件，ROS 2 能够准确地理解机器人的结构和行为，从而实现各种复杂的机器人任务。

首先观察一下我们的 wheeltec 小车结构，它可以简化成下面五个部件组成：小车车体（硬件结构）、左右轮子（驱动系统）、激光雷达和相机（传感系统）、嵌入式主板或者计算平台（控制系统）。机器人建模的过程，其实就是按照类似的思路，通过建模语言，把机器人每一个部分都描述清楚，再组合起来的过程。

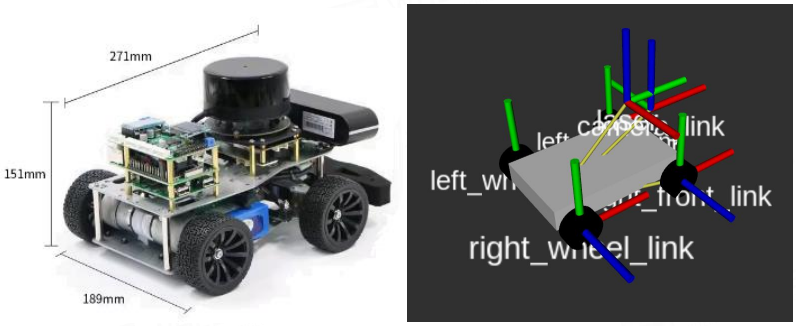


图 4-3-1　WHEELTEC 小车模型

② URDF 文件的 xml 实现

```
# 连杆 Link 的描述,标签用来描述机器人某个刚体部分的外观和物理属性,
# 外观包括尺寸、颜色、形状，物理属性包括质量、惯性矩阵、碰撞参数等。
# name 标签：描述连杆名字，名称需要唯一
  <link name="left_wheel_link">
      <visual>
          # origin 标签：设置 link 的相对便宜量以及旋转角度
          <origin xyz="0 0 0" rpy="0 0 0" />
          # geometry 标签：  用于设置 link 的形状，box（方体）、
          # cylinder（圆柱）、sphere（球体）、mesh（链接其他文件）等
```

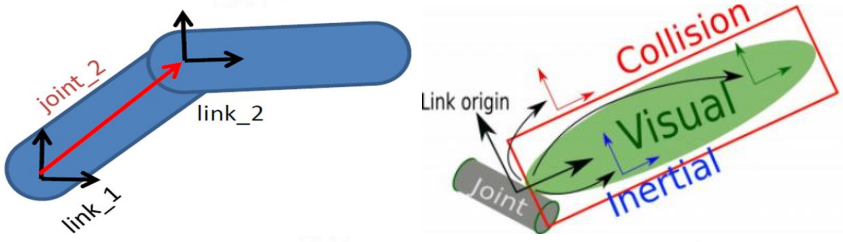
```
            <geometry>
                <cylinder radius="0.0325" length = "0.025"/>
            </geometry>
            # material 标签:视觉元素的材料
            <material name="black">
                <color rgba="0 0 0 1"/>
            </material>
        </visual>
    </link>
# 关节 Joint 描述,机器人模型中的刚体最终要通过关节 joint 连接之后,
# 才能产生相对运动。
# name 标签: 描述关节名字, 名称需要唯一
# type 标签: 设置关节类型
 <joint name="left_wheel_joint" type="continuous">
      # origin 标签: 描述子连杆相对父连杆的偏移量、偏移弧度
      <origin xyz="0 0.08 0.0325" rpy="1.57 0 0"/>
      # parent 标签: 描述父连杆;
      <parent link="base_link"/>
      # child 标签: 描述子连杆, 子连杆会相对父连杆发生运动;
      <child link="left_wheel_link"/>
      # axis 表示关节运动轴的单位向量
      <axis xyz="0 1 0"/>
    </joint>
```

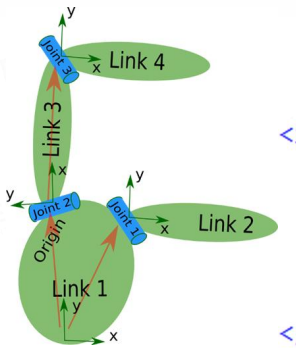
URDF 中的关节 joint 有六种运动类型：

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，类似于continuous，但是有旋转的角度极限
prismatic	滑动关节，沿某一轴线移动的关节，带有位置极限
fixed	固定关节，不允许运动的特殊关节
floating	浮动关节，允许进行平移、旋转运动
planar	平面关节，允许在平面正交方向上平移或者旋转

图 4-3-2　joint 的六种类型

连杆与节点的示意图：






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

    <joint> ..... </joint>
    <joint> ..... </joint>
</robot>
```

图 4-3-3　link 与 joint 的示意图

最终所有的 link 和 joint 标签完成了对机器人每个部分的描述和组合，全都放在一个 robot 标签中，就形成了完整的机器人模型。