ROS2 文档

编号：190912

为什么学习和研究 ROS 2?

参考文献：https://index.ros.org/doc/ros2/Marketing/

ROS 2（机器人操作系统 2）是用于机器人应用的开源开发套件。ROS 2 之目的是为各行各业的开发人员提供标准的软件平台，从研究和原型设计再到部署和生产。 ROS 2 建立在 ROS 1 的成功基础之上，ROS 1 目前已在世界各地的无数机器人应用中得到应用。

特色

- **缩短上市时间**

  ROS 2 提供了开发应用程序所需的机器人工具，库和功能，可以将时间花在对业务非常重要的工作上。因为它是开源的，所以可以灵活地决定在何处以及如何使用 ROS 2，以及根据实际的需求自由定制，**使用 ROS 2 可以大幅度提升产品和算法研发速度!**

- **专为生产而设计**

  凭借在建立 ROS 1 作为机器人研发的**事实上的全球标准方面**的十年经验，ROS 2 从一开始就被建立**在工业级基础上**并可用于生产，包括**高可靠性**和**安全关键**系统。 ROS 2 的设计选择、开发实践和项目管理基于行业利益相关者的要求。

- **多平台支持**

  ROS 2 **在 Linux，Windows 和 macOS 上得到支持和测试，允许无缝开发和部署机器人自动化**，后端管理和用户界面。分层支持模型允许端口到新平台，例如**实时和嵌入式操作系统**，以便在获得兴趣和投资时引入和推广。

- **丰富的应用领域**

  与之前的 ROS 1 一样，**ROS 2 可用于各种机器人应用**，从室内到室外、从家庭到汽车、水下到太空、从消费到工业。

- **没有供应商锁定**

  ROS 2 建立在一个抽象层上，使机器人库和应用程序与通信技术隔离开来。抽象底层是通信代码的多种实现，包括开源和专有解决方案。**在抽象顶层，核心库和用户应用程序是可移植的。**

- **建立在开放标准之上**

  ROS 2 中的默认通信方法使用 IDL、DDS 和 DDS-I RTPS 等行业标准，这些标准已**广泛应用于从工厂到航空航天的各种工业应用**中。

- **开源许可证**

  ROS 2 代码在 Apache 2.0 许可下获得许可，在 3 条款（或“新”）BSD 许可下使用移植的 ROS 1 代码。这两个许可证允许允许使用软件，而不会影响用户的知识产权。

- **全球社区**

  超过 10 年的 ROS 项目通过发展一个**由数十万开发人员和用户组成的全球社区**，为**机器人技术创建了一个庞大的生态系统**，他们为这些软件做出贡献并进行了改进。 ROS 2 由该社区开发并为该社区开发，他们将成为未来的管理者。

- **行业支持**

  正如 ROS 2 技术指导委员会成员所证明的那样，对 ROS 2 的行业支持很强。除了开发顶级产品外，**来自世界各地的大大小小公司都在投入资源为 ROS 2 做出开源贡献**。

- **与 ROS1 的互操作性**

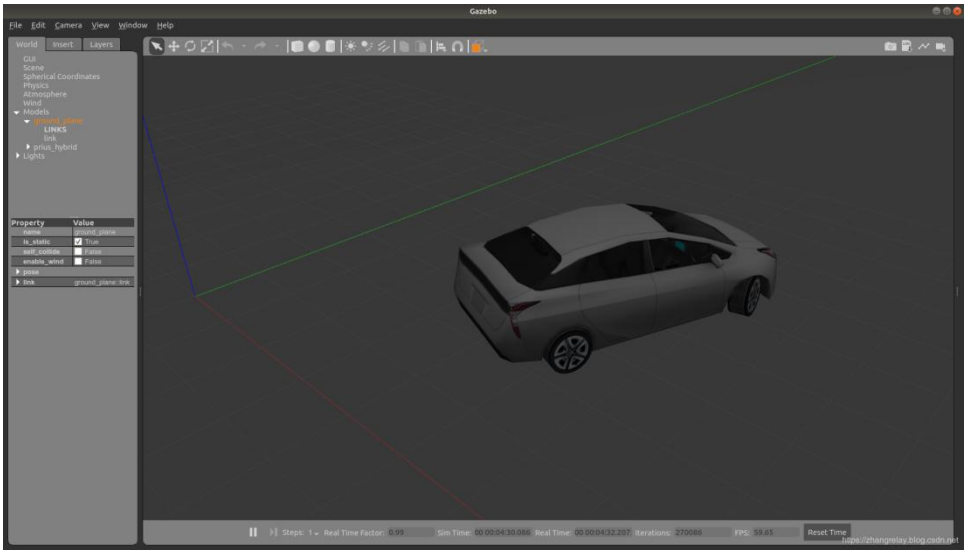
  ROS 2 包括到 ROS 1 的桥接器，**处理两个系统之间的双向通信**。如果有一个现有的 ROS 1 应用程序， 可以通过桥接器开始尝试使用 ROS 2，并根据要求和可用资源逐步移植应用程序。

ROS 2 的一些示例

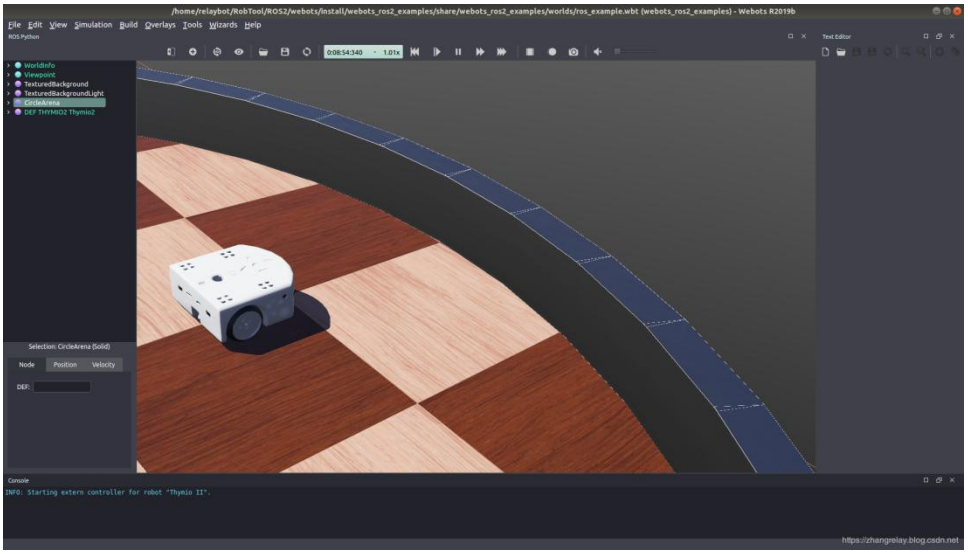
目前，ROS 2 正在快速研发中，但是已经有非常丰富的案例，并且可以成熟应用到机器人原型机和产品设计中。

➢ 仿真

仿真支持 Gazebo、Webots 和 V-Rep 等主流机器人仿真软件。



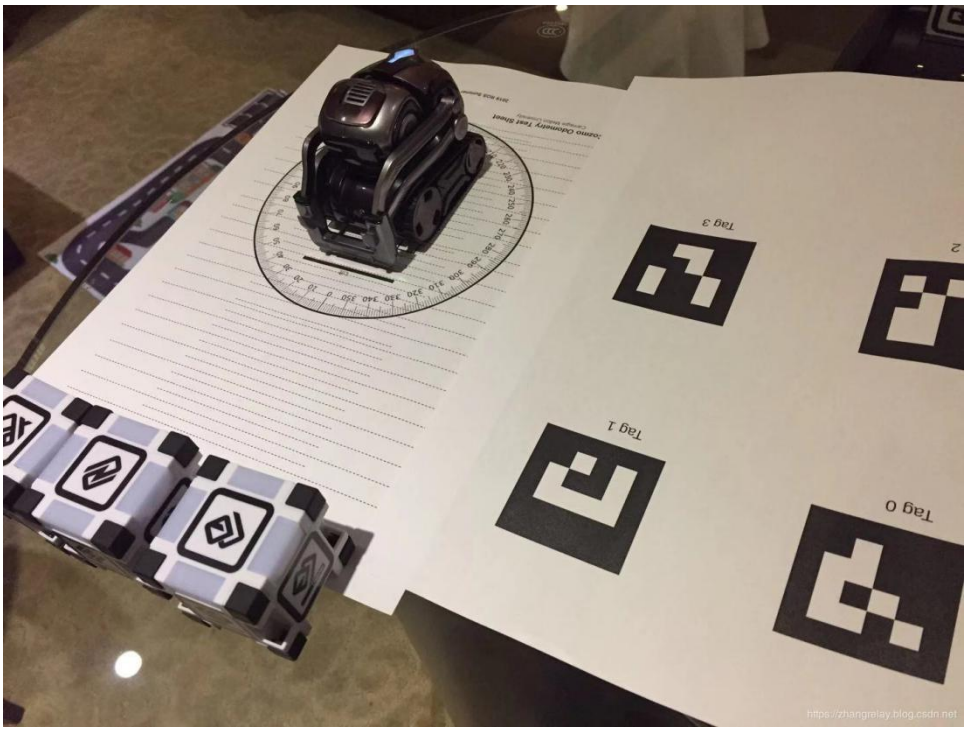
Gazebo（ https://blog.csdn.net/ZhangRelay/article/details/100547011 ）



Webots（ https://blog.csdn.net/ZhangRelay/article/details/100519183 ）

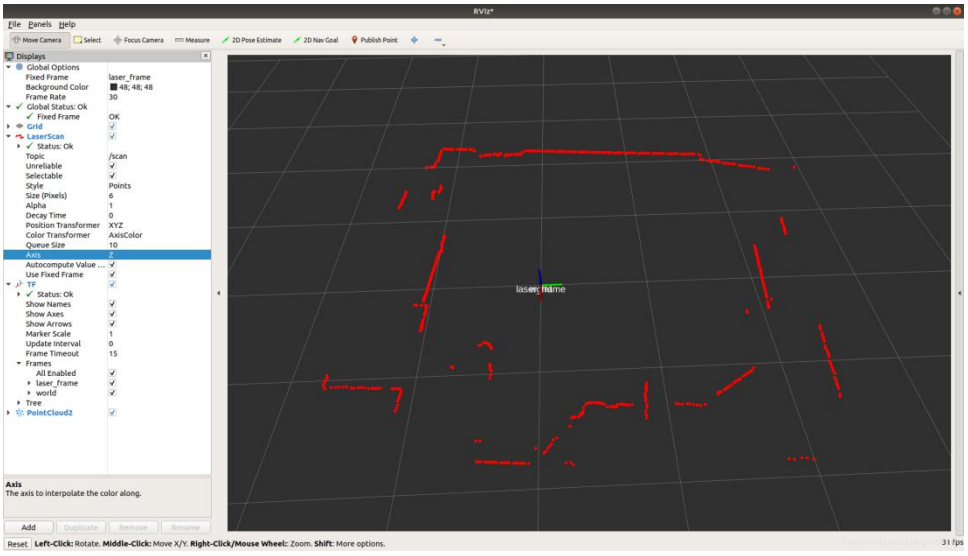
➢ 实物

除了对 Turtlebot3 的全面支持外，Cozmo 和 Tello 等也有非常好的案例。



Cozmo（https://blog.csdn.net/ZhangRelay/article/details/99937033）

➢ 其他



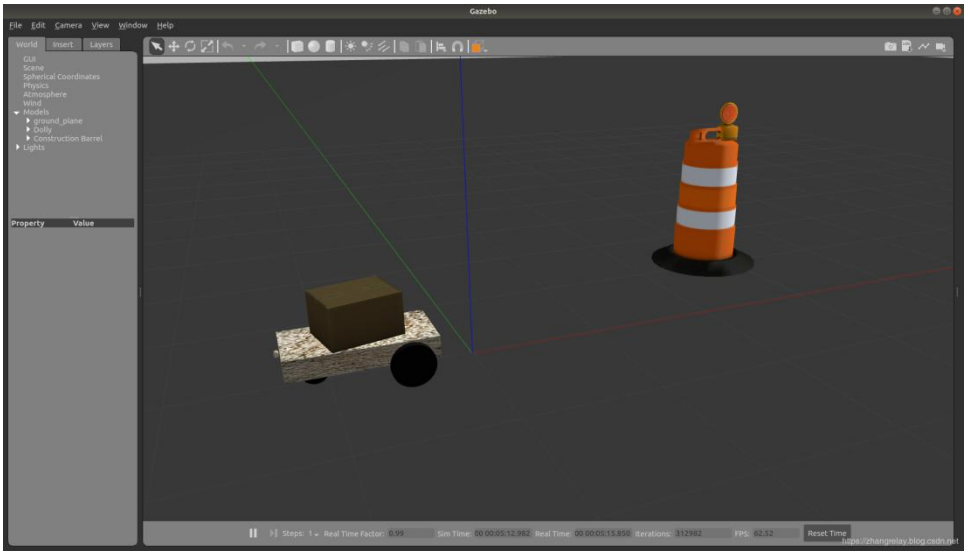
Rviz2 激光数据可视化

如果需要使用摄像头、激光雷达或者语音模块。可以参考：

1. https://blog.csdn.net/ZhangRelay/article/details/100138932
2. https://blog.csdn.net/ZhangRelay/article/details/100139316

➢ 教程

目前，ROS 1 Melodic 教程可完全适用于 ROS 2 Dashing 版本，正在测试开发，其中 ETH 的 ROS 编程基础课已经可以在 ROS 2 完全实现。https://blog.csdn.net/ZhangRelay/article/details/100086799



Gazebo 中小车通过激光传感器识别障碍物并跟随

实践案例依据本科机器人操作系统教学案例整理，ROS 1 课程开设 4 年，ROS 2 课程开设 2 年。感谢易科机器人实验室、机器人操作系统（ROS）教育基金会、常熟理工学院的支持。课程文档的整理还在完善之中，本文档主要是提供给学生进行机器人操作系统二代（ROS 2）阅读资料的整理版。

由于个人水平和时间所限，为避免不准确翻译带来的误导，文档为英中双语，并仅翻译课程中需要使用的内容。

2019-9-12

感 谢 家 人 对 我 工 作 的 理 解 和 支 持

ROS 2

Overview 概述

The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source.

机器人操作系统（ROS）是**一组软件库和工具**，用于**协助开发机器人应用程序**，从驱动程序到最先进的算法，以及强大的开发人员工具。ROS 为开发下一个机器人项目提供了所需的一切，而且它都是**开源**的。

Here you will find documentation on how to install and use software from **ROS 2**, which is a new version of ROS that is under heavy development. See below for more information on ROS 2. Also have a look at our promotional material.

这份文档中将包含有关如何**安装和使用 ROS 2（Dashing）**软件的资料，它是正在全力开发的一个新的 ROS 版本。有关 ROS 2 的更多信息，请参考下文，或者请查看宣传材料。

If you're looking for information on ROS 1 (i.e., ROS as it has existed for several years and what you might be using right now), check the ROS website or the documentation wiki.

如果需要查阅有关 **ROS 1** 的信息（即，已存在 12 年的 ROS 以及现在可能正在使用的 ROS），请查看 ROS 网站或文档维基。ROS 1 LTS 包括 indigo、kinetic 和 **Melodic**。ROS 2 LTS 为 **Dashing** 发行版。

Before you proceed here, please read the features page to get an idea of what is in the current ROS 2 release.

在此之前，请先阅读功能介绍的网页了解当前 ROS 2 版本中的内容。

Releases 发行版

See the releases page. 请参考版本发行版介绍网页。

Installation 安装

See the installation page. 请参考安装介绍网页。

Tutorials and Features 教程和功能

ROS 2 is under heavy development. ROS 2 正在快速发展。

You can check out the tutorials page to see a range of examples of what the system can do in its current state, if it fits your project today depending on the exact requirement.

查看教程内容的网页了解一些关于系统在当前状态下可以实现的功能，依据具体的要求决定它是否适合当前开发的工程案例。

For details of the current features, see features status page. 有关当前功能的详细信息，请参考功能状态网页。

If migrating code from ROS to ROS 2, check the migration guide.

如果将代码从 ROS1 迁移到 ROS 2，请查看迁移指南网页。

What's ahead 事先说明

ROS 2 is currently planning to release new versions every six month (which is twice as often as ROS 1) to give the community members an opportunity to provide early feedback on the evolving system. See the Roadmap for details on the planned upcoming features for ROS 2.

ROS 2 目前计划每六个月发布一个新版本（这是 ROS 1 的两倍），以便社区成员有机会提供有关正在开发系统的源源不断的早期反馈。有关 ROS 2 计划即将推出的功能的详细信息，请参考路线图。

Contributing 贡献

See the contributing page and the developer guide for details on how to contribute to ROS 2 developments.

有关如何为 ROS 2 开发做出贡献的详细信息，请参考参与网页和开发人员指南。

Governance 管理

See the governance page. 请参考管理网页。

Marketing 市场

See the marketing page. 查看市场网页。

Reporting problems and asking questions

报告并提出问题

See the contact page. 请参考联系网页。

About ROS 2 关于 ROS2

Since ROS was started in 2007, a lot has changed in the robotics and ROS community. The goal of the ROS 2 project is to adapt to these changes, leveraging what is great about ROS 1 and improving what isn't. There's a full article on the motivation of ROS 2 here.

自 ROS 于 2007 年启动以来，机器人学和 ROS 社区发生了很多变化。**ROS 2 项目的目标是适应这些变化，利用 ROS 1 的优点和改进不足之处。**关于 ROS 2 开发缘由的一篇完整文章在这里点击查看。

Where to find more information

哪里可以找到更多信息

There are various articles on the design of ROS 2 at design.ros2.org, such as: *Why ROS 2?*, *ROS on DDS*, and *Changes between ROS 1 and ROS 2*.

在 design.ros2.org 上有关于 ROS 2 设计的各种文章，例如：* 为什么开发 ROS 2？ *，* 基于 DDS 的 ROS *，以及* ROS 1 和 ROS 2 之间的变化 *。

The code for ROS 2 is open source and broken into various repositories. You can find the code for most of the repositories on the ros2 github organization.

ROS 2 的代码是开源的，并拆分为各种库。在 ros2 github 组织网页上找到大多数库的源代码。

docs.ros2.org contains current details about ROS 2 internal design and organization.

网页 docs.ros2.org 包含有关 ROS 2 内部设计和组织的最新详细信息。

awesome-ros2 is a "cheat sheet" style quick reference for ROS 2 packages and resources which will get (hopefully) listed on the curated list of awesome lists to help GitHub users to get to know ROS 2.

网页 awesome-ros2 是 ROS 2 包和资源的"清单"式的快速参考指南，将（前景看好的资源）列在精选列表中，以帮助 GitHub 用户了解 ROS 2。

The following ROSCon talks have been given on ROS 2 and provide information about the workings of ROS 2 and various demos:

以下 ROSCon 将与 ROS 2 相关的讨论和，介绍了 ROS 2 工作方式和非常的案例等信息：

Title 标题	**Type** 类型	**Links** 链接
Hands-on ROS 2: A Walkthrough 一步一步实践 ROS 2	ROSCon 2018	slides / video
Launch for ROS 2 ROS 2 的 Launch	ROSCon 2018	slides / video
The ROS 2 vision for advancing the future of robotics development ROS 2 愿景 - 推动未来机器人技术的发展	ROSCon 2017	slides / video
ROS 2 Update - summary of alpha releases, architectural overview ROS 2 更新 - alpha 版本摘要、架构概述	ROSCon 2016	slides / video
Evaluating the resilience of ROS2 communication layer 评估 ROS 2 通信层的弹性	ROSCon 2016	slides / video

Title 标题	**Type** 类型	**Links** 链接
State of ROS 2 - demos and the technology behind ROS 2 的状态 - 案例和背后的技术	ROSCon 2015	slides / video
ROS 2 on “small” embedded systems “小型”嵌入式系统上的 ROS 2	ROSCon 2015	slides / video
Real-time control in ROS and ROS 2 ROS 2 和 ROS 中的实时控制	ROSCon 2015	slides / video
Why you want to use ROS 2 为什么要使用 ROS 2	ROSCon 2014	slides / video
Next-generation ROS: Building on DDS 下一代 ROS：基于 DDS 编译	ROSCon 2014	slides / video

Installation 安装

ROS 2 Installation Options ROS 2 安装选项

Multiple distributions of ROS 2 are supported at a time. We recommend using the most recent release available when possible.

一次同时支持多个 **ROS 2** 发行版，但是建议尽可能使用最新版本（**Dashing**）。

Select your ROS distribution 选择 ROS 2 发行版本

ROS 2 Bouncy Bolson	ROS 2 Crystal Clemmys	**ROS 2 Dashing Diademata**
Released July 2018	Released December 2018	**Released May 2019**
Supported until July 2019	Supported until December 2019	**Supported until May 2021**

For more detailed descriptions of each release see REP-2000 有关每个版本的更详细说明，请参考 REP-2000

Why you might want an older distribution

为什么可能需要一个旧的发行版?

You may want to install an older distribution if you need ROS 2 to:

考虑到如下一些需求可能需要安装较旧的 ROS 2 发行版：

- Operate on an older platform
  在较旧的平台上运行
- Support an older package that hasn't been optimized for the latest release
  尚未支持针对最新版本进行优化的旧软件包
- Be supported for a longer period of time (the latest release isn't necessarily supported the longest)
  得到更长时间的支持（最新版本不一定支持时间最长）

General users with no special considerations should use the most recent release available.

没有特殊注意事项的一般用户应使用最新版本。

Contributing to ROS 2 core? 对 ROS 2 核心贡献一份力量?

If you plan to contribute directly to ROS 2 core packages, you can install the latest development from source.

如果计划直接为 ROS 2 核心软件包做出贡献，可以从源代码安装最新的开发版。

Installing ROS 2 Dashing Diademata

安装 ROS 2

Binary packages 二进制包

We provide ROS 2 binary packages for the following platforms:

以下平台可直接使用 ROS 2 二进制包进行安装：

- Linux (Ubuntu Bionic(18.04))
  - Debian packages
  - "fat" archive
- OS X
- Windows

Building from source 从源码编译安装

We support building ROS 2 from source on the following platforms:

支持在以下平台上从源代码编译 ROS 2：

- Linux
- OS X
- Windows

Which install should you choose?

应该选择哪种安装方式?

Installing from binary packages or from source will both result in a fully-functional and usable ROS 2 install. Differences between the options depend on what you plan to do with ROS 2. **Binary packages** are for general use and provide an already-built install of ROS 2. This is great for people who want to dive in and start using ROS 2 as-is, right away.

从二进制包或源代码安装完全功能和可用的 ROS 2 安装。选项之间的差异取决于将如何使用 ROS 2。**二进制包**是普遍用途，并提供已经编译好的 ROS 2 安装包。这对于想要进入并立即开始使用 ROS 2 的人来说非常棒的选择。

推荐使用二进制包进行安装!

Linux users have two options for installing binary packages: Linux 用户有两种安装二进制包的选项：

- Debian packages
- “fat” archive

Installing from Debian packages is the recommended method. It’s more convenient because it installs its necessary dependencies automatically. It also updates alongside regular system updates. However, you need

root access in order to install Debian packages. If you don't have root access, the "fat" archive is the next best choice.

建议使用 Debian 软件包进行安装。因为它更方便，并会自动安装必要的依赖项，还会与常规系统更新一起更新。但是，需要 root 访问权限才能安装 Debian 软件包。如果没有 root 访问权限，那么"fat"档案包是另一个最佳选择。

OS X and Windows users who choose to install from binary packages only have the "fat" archive option (Debian packages are exclusive to Ubuntu/Debian).

选择从二进制包安装的 OS X 和 Windows 用户只有"fat"档案包选项（Debian 软件包是 Ubuntu/Debian 独有的）。

Building from source is meant for developers looking to alter or explicitly omit parts of ROS 2's base. It is also recommended for platforms that don't support binaries. Building from source also gives you the option to install the absolute latest version of ROS 2.

从源代码编译适用于希望改变或明确省略部 ROS 2 基础的开发人员。对于不支持二进制文件安装的平台，也建议使用源码编译安装。从源代码编译还可以选择安装最新版本的 ROS 2。

Installing ROS2 via Debian Packages

通过 Debian 软件包安装 ROS 2

Debian packages for ROS 2 Dashing Diademata are available for Ubuntu Bionic.

用于 ROS 2 Dashing Diademata 的 Debian 软件包可用于 Ubuntu Bionic。

Setup Locale 设置区域设置

Make sure you have a locale which supports `UTF-8`. If you are in a minimal environment, such as a docker container, the locale may be something minimal like POSIX. We test with the following settings. It should be fine if you're using a different UTF-8 supported locale.

确保语言环境支持 `UTF-8`。如果处于最小的环境中，例如 docker 容器，则语言环境可能与 POSIX 一样极小。使用以下设置进行测试。如果使用不同的支持 UTF-8 的语言环境，应该没问题。

```
sudo locale-gen en_US en_US.UTF-8

sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8

export LANG=en_US.UTF-8
```

Setup Sources 设置源

You will need to add the ROS 2 apt repositories to your system. To do so, first authorize our gpg key with apt like this:

需要将 ROS 2 apt 库添加到系统中。为此，首先使用以下方式授权 gpg 密钥：

```
sudo apt update && sudo apt install curl gnupg2 lsb-release

curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
```

And then add the repository to your sources list:

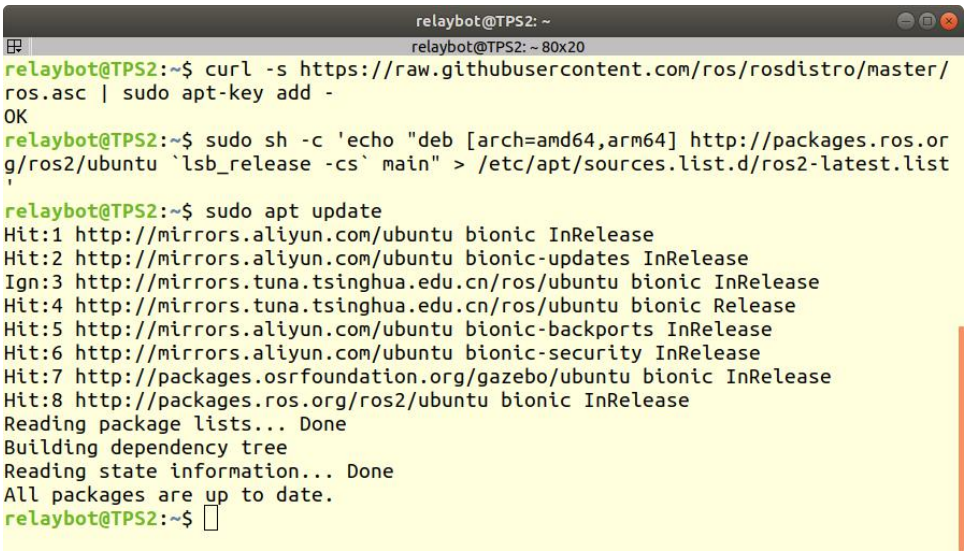
然后将库添加到源列表：

```
sudo sh -c 'echo "deb [arch=amd64,arm64] http://packages.ros.org/ros2/ubuntu `lsb_release -cs` main" > /etc/apt/sources.list.d/ros2-latest.list'
```

Install ROS 2 packages 安装 ROS 2 包

Update your apt repository caches after setting up the repositories. 设置库后更新 apt 库缓存。

```
sudo apt update
```



Desktop Install (Recommended): ROS, RViz, demos, tutorials. 桌面安装（推荐）：ROS、RViz、示例、教程。

```
sudo apt install ros-dashing-desktop
```

ROS-Base Install (Bare Bones): Communication libraries, message packages, command line tools. No GUI tools.

ROS-Base 安装（精简）：通信库、消息包、命令行工具。没有 GUI 工具。

```
sudo apt install ros-dashing-ros-base
```

See specific sections below for how to also install the ros1_bridge, TurtleBot packages, or alternative RMW packages.

有关如何安装 ros1_bridge、TurtleBot 软件包或其他 RMW 软件包的信息，请参考下面的特定部分。

Environment setup 环境设置

(optional) Install argcomplete （可选）安装 argcomplete

ROS 2 command line tools use argcomplete to autocompletion. So if you want autocompletion, installing argcomplete is necessary.

ROS 2 命令行工具使用 argcomplete 完成**自动补全功能**（Tab 补全命令的功能）。因此，如果想要自动完成，则需要安装 argcomplete。

```
sudo apt install python3-argcomplete
```

Sourcing the setup script 导入设置脚本

Set up your environment by sourcing the following file. 通过导入以下文件设置环境。

```
source /opt/ros/dashing/setup.bash
```

You may want to add this to your .bashrc. 想将此添加到.bashrc。

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

Install additional RMW implementations

安装其他 RMW 实现

By default the RMW implementation FastRTPS is used. If using Ardent OpenSplice is also installed.

默认情况下，使用 RMW 实现 FastRTPS。如果还使用了 Ardent OpenSplice。

To install support for OpenSplice or RTI Connext on Bouncy:

在 Bouncy 上安装对 OpenSplice 或 RTI Connext 的支持：

```
sudo apt update
sudo apt install ros-dashing-rmw-opensplice-cpp # for OpenSplice
sudo apt install ros-dashing-rmw-connext-cpp # for RTI Connext (requires license agreement)
```

By setting the environment variable RMW_IMPLEMENTATION=rmw_opensplice_cpp you can switch to use OpenSplice instead. For ROS 2 releases Bouncy and newer, RMW_IMPLEMENTATION=rmw_connext_cpp can also be selected to use RTI Connext.

通过设置环境变量 RMW_IMPLEMENTATION=rmw_opensplice_cpp，可以切换为使用 OpenSplice。对于 ROS 2 Bouncy 或更新版本，如 Dashing，RMW_IMPLEMENTATION=rmw_connext_cpp 也可以选择使用 RTI Connext。

If you want to install the Connext DDS-Security plugins please refer to this page.
如果要安装 Connext DDS-Security 插件，请参考此网页。
University, purchase or evaluation options are also available for RTI Connext.
RTI Connext 也提供大学，购买或试用选项。

Install additional packages using ROS 1 packages

使用 ROS 1 包安装其他软件包

The `ros1_bridge` as well as the TurtleBot demos are using ROS 1 packages. To be able to install them please start by adding the ROS 1 sources as documented here.
在 `ros1_bridge` 还有 TurtleBot 示例等使用 ROS 1 包。为了能够安装它们，请首先添加 ROS 1 源，如此处所述。
If you're using Docker for isolation you can start with the image `ros:melodic` or `osrf/ros:melodic-desktop` (or Kinetic if using Ardent). This will also avoid the need to setup the ROS sources as they will already be integrated. 如果使用 Docker 进行隔离，则可以从镜像 `ros:melodic` 或者 `osrf/ros:melodic-desktop` 开始，(如果使用 Ardent 则使用 Kinetic)。这也将避免设置 ROS 源的需要，因为它们已经被集成。
Now you can install the remaining packages: 现在可以安装剩余的包：

```
sudo apt update
sudo apt install ros-dashing-ros1-bridge
```

The turtlebot2 packages are not currently available in Dashing. 目前，Dashing 中不提供 turtlebot2 软件包。
However, Dashing supports the turtlebot3 package. 但是，**Dashing 支持 turtlebot3 软件包。**

Build your own packages 编译自定义的包

If you would like to build your own packages, refer to the tutorial "Using Colcon to build packages".
如果您想编译自定义的包，请参考教程“使用 Colcon 编译包”。

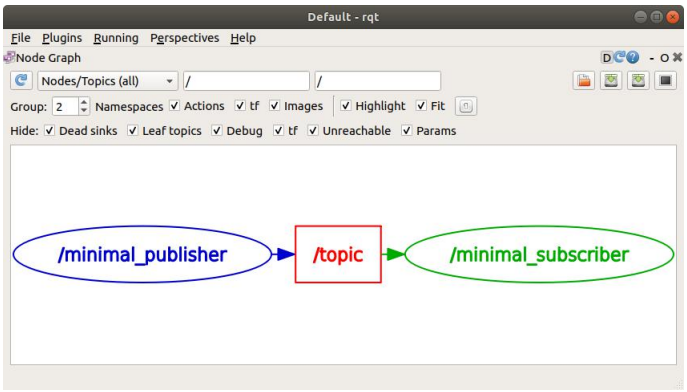
测试一下，ROS Dashing 是否安装配置正常吧，分别在不同终端输入如下命令：

```
ros2 run examples_rclcpp_minimal_subscriber subscriber_lambda
ros2 run examples_rclcpp_minimal_publisher publisher_lambda
```

```
relaybot@TPS2: ~
relaybot@TPS2: ~ 80x11
relaybot@TPS2:~$ ros2 run examples_rclcpp_minimal_subscriber subscriber_lambda
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 0'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 1'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 2'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 3'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 4'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 5'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 6'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 7'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 8'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 9'
relaybot@TPS2: ~ 80x11
relaybot@TPS2:~$ ros2 run examples_rclcpp_minimal_publisher publisher_lambda
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 0'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 1'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 2'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 3'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 4'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 5'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 6'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 7'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 8'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 9'
```

命令分别打开一个订阅器和一个发布器，用于收发 hello world 消息。

使用 rqt 图形化工具可以看到：



简要看一下如下代码，和常见 C++或 ROS 1 代码相比，有何相同和不同之处?

发布器：

```
#include <chrono>
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses a fancy C++11 lambda
 * function to shorten the callback syntax, at the expense of making the
 * code somewhat more difficult to understand at first glance. */

class MinimalPublisher : public rclcpp::Node
{
public:
```

```
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
    auto timer_callback =
      [this]() -> void {
        auto message = std_msgs::msg::String();
        message.data = "Hello, world! " + std::to_string(this->count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
        this->publisher_->publish(message);
      };
    timer_ = this->create_wall_timer(500ms, timer_callback);
  }

private:
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

接收器：

```
#include <iostream>
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class MinimalSubscriber : public rclcpp::Node
{
public:
  MinimalSubscriber()
```

```
  : Node("minimal_subscriber")
  {
    subscription_ = this->create_subscription<std_msgs::msg::String>(
      "topic",
      10,
      [this](std_msgs::msg::String::UniquePtr msg) {
        RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
      });
  }

private:
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

试一试 ROS 2 的小乌龟案例吧：

Turtlesim（https://blog.csdn.net/ZhangRelay/article/details/99932959）

Roadmap 路线图

Please see the page of the upcoming distribution Eloquent Elusor for more information what is planned to be part of that release.

有关新计划发行版的更多信息，请参考即将发布的 Eloquent Elusor 发行版网页。

Please see the Distributions page for the timeline of and information about future distributions.

有关未来发行版的时间表和信息，请参考“ 发行版”网页。

For more information on the design of ROS 2 please see design.ros2.org. The core code for ROS 2 is on the ros2 github organization. The Discourse forum/mailing list for discussing ROS 2 design is ng-ros. Questions should be asked on ROS answers, make sure to include at least the `ros2` tag and the rosdistro version you are running, e.g. `ardent`.

ROS 2 设计的更多信息，请参考 design.ros2.org 网页。ROS 2 的核心代码在 ros2 github 组织上。讨论 ROS 2 的交流论坛/邮件列表是 ng-ros。提问并获取有关 ROS 的答案，确保至少包含 `ros2` 标签以及正在运行的发行版本信息，例如 `dashing`。

Distributions 发行版本

What is a Distribution? 什么是发行版本?

See wiki.ros.org/Distributions. 请参考 wiki.ros.org/Distributions。

List of Distributions 发行版本清单

Distro 发行版本	Release date 发布日期	EOL date 停止支持日期
Dashing Diademata	**May 31st, 2019**	**May 2021**
Crystal Clemmys	December 14th, 2018	Dec 2019
Bouncy Bolson	July 2nd, 2018	Jul 2019
Ardent Apalone	December 8th, 2017	Dec 2018
beta3	September 13th, 2017	Dec 2017
beta2	July 5th, 2017	Sep 2017
beta1	December 19th, 2016	Jul 2017
alpha1 - alpha8	August 31th, 2015	Dec 2016

目前，如果学习 ROS 2，Dashing 发行版是最佳选择。

Distribution Details 发行版本细节

For details on the distributions see each releases page. For the supported platforms and versions of common dependencies and other considerations, see the official ROS 2 Target Platforms REP 2000.
有关发行版本的详细信息，请参考每个版本页面。有关发行版本支持的平台和常见依赖关系以及其他注意事项，请参考官方 ROS 2 目标平台 REP 2000。

Future Distributions 未来发行版本

For details on upcoming features see the roadmap. 有关即将推出功能的详情，请参考路线图。
Currently there is a new ROS 2 distribution roughly every 6 months. The following information are best estimates and are subject to change.
目前大约每 6 个月就会有一个新的 ROS 2 发行版。以下信息是最佳估计值，实际可能会有所变化。

Distro 发行版本	Release date 发布日期	Supported for 支持时间	Planned changes 目标平台
Eloquent Elusor	November 22nd, 2019	1 year	
<F-turtle>	**May 2020**	**3+ years**	**Target Ubuntu 20.04**

ROS 2 F 发行版，将是第二个 LTS 的版本。

Concepts 概念

- Overview of ROS 2 Concepts ROS 2 概念概述
- ROS 2 and different DDS/RTPS vendors ROS 2 和不同的 DDS / RTPS 供应商
- About Quality of Service Settings 关于服务质量设置
- About ROS 2 Interfaces 关于 ROS 2 接口
- About ROS2 client libraries 关于 ROS2 客户端库
- Logging and logger configuration 日志记录和日志记录器配置

See also http://docs.ros2.org/ for ROS 2 high level documentation.

有关 ROS 2 高级文档，另请参考 http://docs.ros2.org/。

Overview of ROS 2 Concepts

ROS 2 概念综述

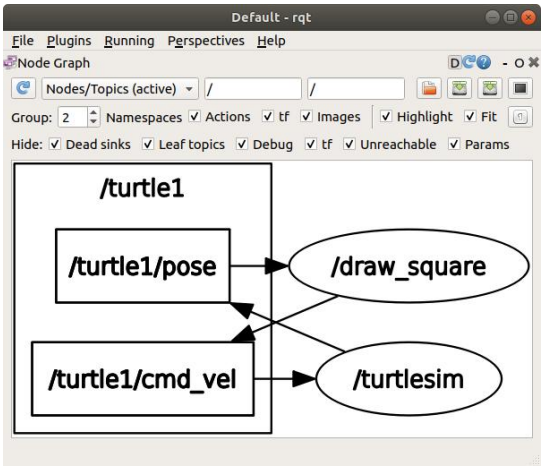
Table of Contents 目录

ROS is a middleware based on an anonymous publish/subscribe mechanism that allows for message passing between different ROS processes.

ROS 是一种基于匿名发布/订阅机制的中间件，允许在不同的 ROS 进程之间传递消息。

At the heart of any ROS 2 system is the ROS graph. The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate.

任何 ROS 2 系统的核心是 ROS 图。ROS 图指的是 ROS 系统中的节点网络以及它们之间通信的连接。



ROS2 turtlesim Graph

Quick Overview of Graph Concepts

图概念的快速概述

- Nodes: A node is an entity that uses ROS to communicate with other nodes.
  节点：节点是使用 ROS 与其他节点通信的实体（或实例）。
- Messages: ROS data type used when subscribing or publishing to a topic.
  消息：订阅或发布到主题时 ROS 使用的数据类型。
- Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
  主题：节点可以将消息发布到主题，也可以订阅主题以接收消息。
- Discovery: The automatic process through which nodes determine how to talk to each other.
  发现：节点确定如何相互通信的自动过程。

Nodes 节点

A node is a participant in the ROS graph. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service. There are configurable Parameters associated with a node. Connections between nodes are established through a distributed discovery process. Nodes may be located in the same process, in different processes, or on different machines. These concepts will be described in more detail in the sections that follow.

节点是 ROS 图中的参与者。ROS 节点使用 ROS 客户端库与其他节点通信。节点可以发布或订阅主题。节点还可以提供或使用服务。可以对节点相关联的参数进行配置。节点之间的连接是通过分布式发现过程建立的。节点可以位于相同的进程中，也可以位于不同的进程中，还可以位于不同的机器上。这些概念将在以下部分中更详细地介绍说明。

Client Libraries 客户端库

ROS client libraries allow nodes written in different programming languages to communicate. There is a core ROS client library (RCL) that implements common functionality needed for the ROS APIs of different languages. This makes it so that language-specific client libraries are easier to write and that they have more consistent behavior.

ROS 客户端库**支持不同编程语言编写的节点进行通信**。一个核心的 ROS 客户端库（ROS client library, RCL）实现了不同语言 ROS API 所需的通用功能。这使得特定语言的客户端库更易于编写，并且它们具有更一致的行为。

The following client libraries are maintained by the ROS 2 team:

以下客户端库由 ROS 2 团队维护：

- rclcpp = C++ client library
  rclcpp = C ++客户端库
- rclpy = Python client library
  rclpy = Python 客户端库

Additionally, other client libraries have been developed by the ROS community. See the ROS 2 Client Libraries article for more details.

此外，ROS 社区还开发了其他客户端库。有关这方面内容的更多详细信息，请参考 ROS 2 客户端库文章。

Discovery 消息发现机制

Discovery of nodes happens automatically through the underlying middleware of ROS 2. It can be summarized as follows: 节点的发现通过 ROS 2 的底层中间件自动实现完成。可以总结如下：

1. When a node is started, it advertises its presence to other nodes on the network with the same ROS domain (set with the ROS_DOMAIN_ID environment variable). Nodes respond to this advertisement with information about themselves so that the appropriate connections can be made and the nodes can communicate.
   启动节点时，它会将其存在通告给具有相同 ROS 域的网络上的其他节点（使用 ROS_DOMAIN_ID 环境变量设置 ROS 域）。节点使用有关自身的信息响应此通告，以便可以进行适当的连接并且节点可以进行通信。
2. Nodes periodically advertise their presence so that connections can be made with new-found entities, even after the initial discovery period.
   节点定期通告其存在，以便即使在初始发现期之后也可以与新发现的实体建立连接。
3. Nodes advertise to other nodes when they go offline.
   节点在离线时通告其他节点。

Nodes will only establish connections with other nodes if they have compatible Quality of Service settings.

只有节点具有兼容的服务质量设置，它们之间才可以建立连接。

Example: talker-listener 例如：发布器-订阅器

In one terminal, start a node (written in C++) that will publish messages on a topic.

在一个终端中，启动将在主题上发布消息的节点（用 C ++编写）。

```
ros2 run demo_nodes_cpp talker
```

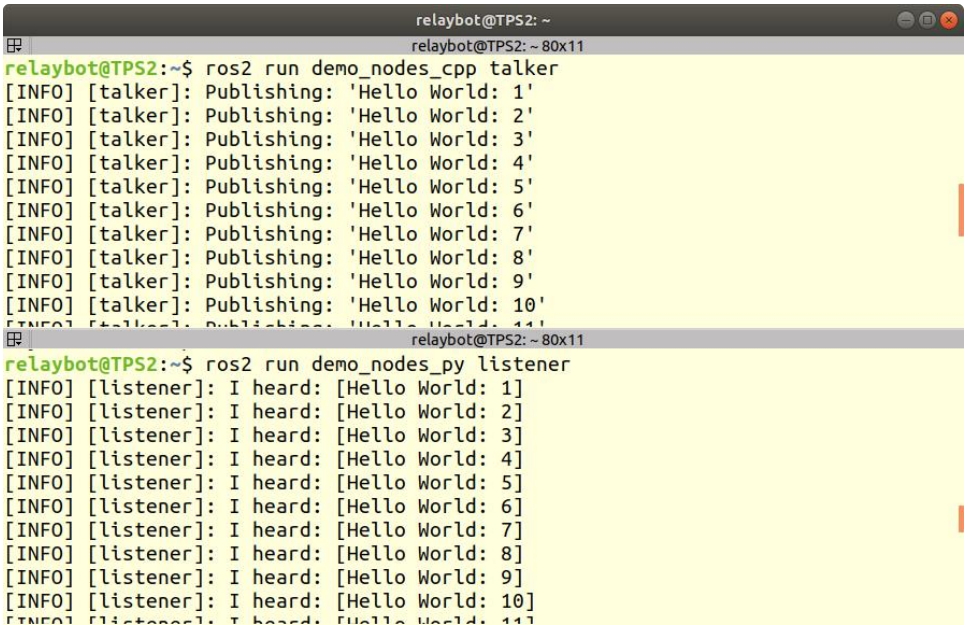
In another terminal, start a second node (written in Python) that will subscribe to messages on the same topic.

在另一个终端中，启动第二个节点（用 Python 编写），该节点将订阅同一主题的消息。

```
ros2 run demo_nodes_py listener
```

You should see that these nodes discover each other automatically, and begin to exchange messages.

如果运行正常，应该看到这些节点自动发现彼此，并开始交换消息。



多试几组案例，熟练掌握 **ros2 run** 和 **rqt** 的基本使用。

ROS 2 and different DDS/RTPS vendors

ROS 2 和不同的 DDS / RTPS 供应商

ROS 2 is built on top of DDS/RTPS as its middleware, which provides discovery, serialization and transportation. This article explains the motivation behind using DDS implementations, and/or the RTPS wire protocol of DDS, in detail, but in summary DDS is an end-to-end middleware that provides features which are relevant to ROS systems, such as distributed discovery (not centralized like in ROS 1) and control over different "Quality of Service" options for the transportation.

ROS 2 建立在 DDS / RTPS 之上，将其作为中间件，提供发现、序列化和传输等功能。本节详细介绍采用 **DDS 实现和（或）DDS 的 RTPS 有线协议**的缘由，但先总述一下，DDS 是一个端到端的中间件，它提供了 ROS 系统相关的功能，例如分布式发现（并不是 ROS 1 采用的集中式，如 roscore）和控制传输不同"服务质量"的选项。

DDS is an industry standard which is then implemented by a range of vendors, such as RTI's implementation Connext or ADLink's implementation OpenSplice RTPS (a.k.a. DDSI-RTPS) is the wire protocol used by DDS to communicate over the network, and there are implementations of that which do not fulfill the full DDS API, but provide sufficient functionality for ROS 2, such as eProsima's implementation Fast RTPS.

DDS 是一个行业标准，然后由一系列供应商实施，如：RTI 的实现版本-Connext 或 ADLink 的实现版本-OpenSplice RTPS（又名 DDSI-RTPS）是 DDS 用于通过网络进行通信的有线协议，虽然有些实现并不能满足完整的 DDS API，但可以为 ROS 2 提供足够的功能，例如 eProsima 的实现版本-快速 RTPS。

ROS 2 supports multiple DDS/RTPS implementations because it is not necessarily "one size fits all" when it comes to choosing a vendor/implementation. There are many factors you might consider while choosing a middleware implementation: logistical considerations like the license, or technical considerations like platform availability, or computation footprint. Vendors may provide more than one DDS or RTPS implementation targeted at meeting different needs. For example, RTI has a few variations of their Connext implementation that vary in purpose, like one that specifically targets microcontrollers and another which targets applications requiring special safety certifications (we only support their standard desktop version at this time).

ROS 2 支持多种 DDS / RTPS 实现，因此在选择供应商/实现时，并不是"一刀切"。在选择中间件实现时，可能会考虑许多因素：许可、技术、平台可用性或计算占用空间等因素。供应商可能会提供多个针对满足不同需求的 DDS 或 RTPS 实现版本。例如，RTI 有一些 Connext 实现的变化，其目的各不相同，例如专门针对微控制器而另一个针对需要特殊安全认证的应用（目前 ROS 2 仅支持其标准桌面版本）。

In order to use a DDS/RTPS implementation with ROS 2, a "**R**OS **M**iddle**w**are interface" (a.k.a. `rmw` interface or just `rmw`) package needs to be created that implements the abstract ROS middleware interface using the DDS or RTPS implementation's API and tools. It's a lot of work to implement and maintain RMW packages for supporting DDS implementations, but supporting at least a few implementations is important for ensuring that the ROS 2 codebase is not tied to any one particular implementation, as users may wish to switch out implementations depending on their project's needs.

为了使用 DDS / RTPS 实现与 ROS 2，ROS 中间接口“**R**OS **M**iddle**w**are interface, RMW”（又名 rmw 接口或 rmw）封装需要创建一个使用 DDS 实现或 RTPS 实现的 API 和工具抽象 ROS 中间件接口。实现和维护 RMW 包用于支持 DDS 开发需要做很多工作，但至少支持一些实现对于确保 ROS 2 代码库不依赖于任何一个特定实现非常重要，因为用户依据具体项目的需求，可能希望根据需要切换实现版本。具体如下：

Supported RMW implementations

支持的 RMW 实现版本

Product name 产品名称	**License** 许可	**RMW implementation** **RMW** 实现	**Status** 状态
eProsima *Fast RTPS*	Apache 2	rmw_fastrtps_cpp	Full support. Default RMW. Packaged with binary releases. 全力支持。默认 RMW。打包二进制版本
RTI *Connext*	commercial, research	rmw_connext_cpp	Full support. Support included in binaries, but Connext installed separately. 全力支持。支持包含在二进制文件中，但 Connext 单独安装
RTI *Connext* (dynamic implementation) 动态实现	commercial, research	rmw_connext_dynamic_cpp	Support paused. Full support until alpha 8.* 支持暂停。完全支持直到 alpha 8
ADLINK *Opensplice*	Apache 2, commercial	rmw_opensplice_cpp	Partial support. Support included in binaries, but OpenSplice installed separately. 部分支持。支持包含在二进制文件中，但 OpenSplice 单独安装
OSRF *FreeRTPS*	Apache 2	–	Partial support. Development paused. 部分支持。发展暂停

“Partial support” means that one or more of the features required by the rmw interface is not implemented.
“部分支持”意味着 rmw 接口所需的一个或多个功能并未实现。

For practical information on working with multiple RMW implementations, see the "Working with multiple RMW implementations" tutorial.
有关使用多种 RMW 实现的实用信息，请参考“使用多种 RMW 实现”教程。

About Quality of Service Settings

关于服务质量设置

Overview 概述

ROS 2 offers a rich variety of Quality of Service (QoS) policies that allow you to tune communication between nodes. With the right set of Quality of Service policies, ROS 2 can be as reliable as TCP or as best-effort as UDP, with many, many possible states in between. Unlike ROS 1, which primarily only supported TCP, ROS 2 benefits from the flexibility of the underlying DDS transport in environments with lossy wireless networks where a “best effort” policy would be more suitable, or in real-time computing systems where the right Quality of Service profile is needed to meet deadlines.

ROS 2 提供丰富的服务质量（QoS）策略，支持调整节点之间的通信。通过正确的服务质量策略集，ROS 2 可以像 TCP 一样可靠，也可以像 UDP 那样尽力而为，其间有许多可能的状态。与主要仅支持 TCP 的 ROS 1 不同，ROS 2 更好支持具有损耗的无线网络环境中基础 DDS 传输的灵活性，其中“尽力而为”策略将更合适，或者在具有准确质量的实时计算系统中需要服务配置文件来设置最后期限。

A set of QoS “policies” combine to form a QoS “profile”. Given the complexity of choosing the correct QoS policies for a given scenario, ROS 2 provides a set of predefined QoS profiles for common usecases (e.g. sensor data). At the same time, users are given the flexibility to control specific profiles of the QoS policies.

一组 QoS“策略”组合形成了 QoS“配置文件”。考虑到为给定方案选择正确的 QoS 策略的复杂性，ROS 2 为常见案例（例如传感器数据）提供了一组预定义的 QoS 配置文件。同时，用户可以灵活地控制 QoS 策略的特定配置文件。

QoS profiles can be specified for publishers, subscribers, service servers and clients. A QoS profile can be applied independently to each instance of the aforementioned entities, but if different profiles are used it is possible that they will not connect.

可以为发布器、订阅器、提供服务的服务器和客户端指定 QoS 配置文件。QoS 配置文件可以独立地应用于前述实体的每个实例，但是如果使用不同的配置文件，则它们可能不会建立连接。

QoS policies QoS 策略

The base QoS profile currently includes settings for the following policies:

基本 QoS 配置文件当前包含以下策略的设置：

- History 历史
  - Keep last: only store up to N samples, configurable via the queue depth option.
    保留最后（最新）：仅存储 N 个样本，可通过队列深度选项进行配置。
  - Keep all: store all samples, subject to the configured resource limits of the underlying middleware.
    全部保留：根据底层中间件的配置资源限制存储所有样本。
- Depth 深度
  - Size of the queue: only honored if used together with “keep last”.

队列的大小：只有与“保留最后（最新）”一起使用，才可实现。

- Reliability 可靠性
  - Best effort: attempt to deliver samples, but may lose them if the network is not robust.
    尽力而为：尝试传输样本，但如果网络不健全，可能会丢失样本。
  - Reliable: guarantee that samples are delivered, may retry multiple times.
    可靠：保证样本已交付，可多次重试。
- Durability 持续性
  - Transient local: the publisher becomes responsible for persisting samples for “late-joining” subscribers.
    瞬态本地：发布器负责为“迟到的”订阅器保留样本。
  - Volatile: no attempt is made to persist samples.
    易失性：没有试图持续的样本。

For each of the policies there is also the option of “system default”, which uses the default of the underlying middleware which may be defined via DDS vendor tools (e.g. XML configuration files). DDS itself has a wider range of policies that can be configured. These policies have been exposed because of their similarity to features in ROS 1; it is possible that in the future more policies will be exposed in ROS 2.
对于每个策略，还有“system default”选项，它使用可以通过 DDS 供应商工具（例如 XML 配置文件）定义的底层中间件的默认值。DDS 本身具有可以配置的更广泛的策略。由于它们与 ROS 1 中的特征相似，因此开放了这些配置；有可能在未来更多的配置将在 ROS 2 中开放。

Comparison to ROS 1 与 ROS 1 的比较

The history and depth policies in ROS 2 combine to provide functionality akin to the queue size in ROS 1.
ROS 2 中的历史和深度策略结合起来提供类似于 ROS 1 中的队列大小的功能。

The reliability policy in ROS 2 is akin to the use of either UDPROS (only in `roscpp`) for “best effort”, or TCPROS (ROS 1 default) for reliable. Note however that even the reliable policy in ROS 2 is implemented using UDP, which allows for multicasting if appropriate.
ROS 2 中的可靠性策略类似于使用 UDPROS（仅支持 `roscpp`）用于“尽力而为”或 TCPROS（ROS 1 默认值）用于可靠性。但请注意，即使 ROS 2 中的可靠策略使用 UDP 实现的，如果合适，它允许进行多点广播。

The durability policy combined with a depth of 1 provides functionality similar to that of “latching” subscribers.
持续性策略与深度 1 相结合提供了类似于“锁定”订阅器的功能。

QoS profiles QoS 配置文件

Profiles allow developers to focus on their applications without worrying about every QoS setting possible. A QoS profile defines a set of policies that are expected to go well together for a particular use case.
配置文件允许开发人员专注于应用程序，而无需担心每个 QoS 设置。QoS 配置文件定义了一组策略，这些策略可以在特定用例中很好地协同工作。

The currently-defined QoS profiles are: 当前定义的 QoS 配置文件是：

- Default QoS settings for publishers and subscribers 发布器和订阅器默认的 QoS 设置

In order to make the transition from ROS 1 to ROS 2, exercising a similar network behavior is desirable. By default, publishers and subscribers are reliable in ROS 2, have volatile durability, and “keep last” history.
为了实现从 ROS 1 到 ROS 2 的转换，期望执行类似的网络行为。默认情况下，发布器和订阅器在 ROS 2 中是可靠的，具有不稳定的持续性，并且“保持最后（最新）”的历史记录。

- Services 服务

  In the same vein as publishers and subscribers, services are reliable. It is especially important for services to use volatile durability, as otherwise service servers that re-start may receive outdated requests. While the client is protected from receiving multiple responses, the server is not protected from side-effects of receiving the outdated requests.
  与发布器和订阅器一样，服务也是可靠的。对于使用 volatile（易失）持续性的服务尤其重要，否则重新启动的服务器可能会收到过时的请求。虽然保护客户端不接收多个响应，但是服务器不受保护,避免受接收过期请求的副作用。

- Sensor data 传感器数据

  For sensor data, in most cases it’s more important to receive readings in a timely fashion, rather than ensuring that all of them arrive. That is, developers want the latest samples as soon as they are captured, at the expense of maybe losing some. For that reason the sensor data profile uses best effort reliability and a smaller queue depth.
  对于传感器数据，在大多数情况下，及时接收读数更重要，而不是确保所有读数都到达。也就是说，开发人员一旦捕获就会想要最新的样本，但可能会损失一些样本。因此，传感器数据配置文件使用尽力而为的可靠性和较小的队列深度。

- Parameters 参数

  Parameters in ROS 2 are based on services, and as such have a similar profile. The difference is that parameters use a much larger queue depth so that requests do not get lost when, for example, the parameter client is unable to reach the parameter service server.
  ROS 2 参数是基于服务的，因此具有类似的配置文件。不同之处在于参数使用更大的队列深度，以便在参数客户端无法访问参数服务服务器时请求不会丢失。

- System default 系统默认

  This uses the system default for all of the policies.
  这将使用所有策略的系统默认值。

Click here for the specific policies in use for the above profiles. The settings in these profiles are subject to further tweaks, based on the feedback from the community.
单击此处查看上述配置文件使用的特定策略。根据社区的反馈，这些配置文件中的设置可能会进一步调整。

While ROS 2 provides some QoS profiles for common use cases, the use of policies that are defined in DDS allows ROS users to take advantage of the vast knowledge base of existing DDS documentation for configuring QoS profiles for their specific use case.
虽然 ROS 2 为常见用例提供了一些 QoS 配置文件，但使用 DDS 中定义的策略允许 ROS 用户利用现有 DDS 文档的庞大知识库来为其特定用例配置 QoS 配置文件。（支持自定义）

QoS compatibilities QoS 兼容性

Note: This section refers to publisher and subscribers but the content applies to service servers and clients in the same manner.

注意：此部分涉及发布器和订阅器（主题），但内容同样适用于提供服务的服务器和客户端。

QoS profiles may be configured for publishers and subscribers independently. A connection between a publisher and a subscriber is only made if the pair has compatible QoS profiles. QoS profile compatibility is determined based on a “Request vs Offerer” model, wherein connections are only made if the requested policy of the subscriber is not more stringent than the that of the publisher. The less strict of the two policies will be the one used for the connection.

QoS 配置文档可以独立地为发布器和订阅器进行配置。仅当对具有兼容的 QoS 配置文件时，才会建立发布器与订阅器之间的连接。QoS 配置文件兼容性是基于“请求与提供者”模型确定的，其中仅在所请求的订阅器策略不比发布器的策略更严格时才进行连接。两种策略中较不严格的方式是用于连接的策略。

The QoS policies exposed in ROS 2 that affect compatibility are the durability and reliability policies. The following tables show the compatibility of the different policy settings and the result:

ROS 2 中开放的影响兼容性的 QoS 策略是持续性和可靠性策略。下表显示了不同策略设置和结果的兼容性：

Compatibility of QoS durability profiles: *QoS 持续性配置文件的兼容性*

Publisher	Subscriber	Connection	Result
Volatile	Volatile	Yes	Volatile
Volatile	Transient local	No	•
Transient local	Volatile	Yes	Volatile
Transient local	Transient local	Yes	Transient local

Compatibility of QoS reliability profiles: *QoS 可靠性配置文件的兼容性*

Publisher	Subscriber	Connection	Result
Best effort	Best effort	Yes	Best effort
Best effort	Reliable	No	•
Reliable	Best effort	Yes	Best effort
Reliable	Reliable	Yes	Reliable

In order for a connection to be made, all of the policies that affect compatibility must be compatible. That is, even if a publisher-subscriber pair has compatible reliability QoS profiles, if they have incompatible durability QoS profiles a connection will not be made, and vice-versa.

为了建立连接，所有影响兼容性的策略必须兼容。也就是说，发布器-订阅器需要具有兼容可靠的 QoS 配置文件，如果它们具有不兼容持续性 QoS 配置文件，则不会建立连接，反之亦然。

About ROS 2 Interfaces 关于 ROS 2 接口

Table of Contents 目录

1. Background 背景

ROS applications typically communicate through interfaces of one of two types: messages and services. ROS uses a simplified description language to describe these interfaces. This description makes it easy for ROS tools to automatically generate source code for the interface type in several target languages.
ROS 应用程序通常通过以下两种类型之一的接口进行通信：消息和服务。ROS 使用简化的描述语言来描述这些接口。此描述使 ROS 工具可以轻松地为多种目标语言中的接口类型自动生成源代码。

In this document we will describe the supported types and how to create your own msg/srv files.
在本节将介绍支持的类型以及如何创建自定义的 msg / srv 文件。

2. Message Description Specification 消息说明规格

Messages description are defined in `.msg` files in the `msg/` directory of a ROS package. `.msg` files are composed of two parts: fields and constants.
消息描述定义在 ROS 包 `msg/`目录的`.msg` 文件中。 `.msg` 文件由两部分组成：字段和常量。

2.1 Fields 字段

Each field consists of a type and a name, separated by a space, i.e:
每个字段由一个类型和一个名称组成，用空格分隔，即：

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

For example: 例如：

```
int32 my_int
string my_string
```

2.1.1 Field Types 字段类型

Field types can be: 字段类型可以是：

- a built-in-type
  内置式
- names of Message descriptions defined on their own, such as “geometry_msgs/PoseStamped”
  自定义的消息描述的名称，例如“geometry_msgs/PoseStamped”

Built-in-types currently supported: *目前支持的内置类型：*

Type name	C++	Python	DDS type
bool	bool	builtins.bool	boolean
byte	uint8_t	builtins.bytes*	octet
char	char	builtins.str*	char
float32	float	builtins.float*	float
float64	double	builtins.float*	double
int8	int8_t	builtins.int*	octet
uint8	uint8_t	builtins.int*	octet
int16	int16_t	builtins.int*	short
uint16	uint16_t	builtins.int*	unsigned short
int32	int32_t	builtins.int*	long
uint32	uint32_t	builtins.int*	unsigned long
int64	int64_t	builtins.int*	long long
uint64	uint64_t	builtins.int*	unsigned long long
string	std::string	builtins.str	string

Every built-in-type can be used to define arrays: *每个内置类型都可用于定义数组：*

Type name	C++	Python	DDS type
static array	std::array<T, N>	builtins.list*	T[N]
unbounded dynamic array	std::vector	builtins.list	sequence
bounded dynamic array	custom_class<T, N>	builtins.list*	sequence<T, N>
bounded string	std::string	builtins.str*	string

All types that are more permissive than their ROS definition enforce the ROS constraints in range and length by software
所有比 ROS 定义更宽松的类型都通过软件强制执行 ROS 范围和长度约束！

Example of message definition using arrays and bounded types: *使用数组和有界类型的消息定义示例：*

```
int32[] unbounded_integer_array
int32[5] five_integers_array
int32[<=5] up_to_five_integers_array

string string_of_unbounded_size
string<=10 up_to_ten_characters_string

string[<=5] up_to_five_unbounded_strings
string<=10[] unbounded_array_of_string_up_to_ten_characters each
string<=10[<=5] up_to_five_strings_up_to_ten_characters_each
```

2.1.2 Field Names 字段名称

Field names must be lowercase alphanumeric characters with underscores for separating words. They must start with an alphabetic character, they must not end with an underscore and never have two consecutive underscores.
字段名称必须是带有下划线的小写字母数字字符，用于分隔单词。它们必须以字母字符开头，它们不能以下划线结尾，也不能有两个连续的下划线。

2.1.3 Field Default Value 字段默认值

Default values can be set to any field in the message type. Currently default values are not supported for string arrays and complex types (i.e. types not present in the built-in-types table above, that applies to all nested messages)
可以将默认值设置为消息类型中的任何字段。当前字符串数组和复杂类型（即上面内置类型表中不存在的类型，不适用于所有嵌套消息）不支持默认值
Defining a default value is done by adding a third element to the field definition line, i.e:
通过向字段定义行添加第三个元素来定义默认值，即：

```
fieldtype fieldname fielddefaultvalue
```

For example: 例如：

```
uint8 x 42
int16 y -2000
string full_name "John Doe"
int32[] samples [-200, -100, 0, 100, 200]
```

Note: 注意：

- string values must be defined in single ' or double quotes " 字符串值必须用单引号'或双引号"定义
- currently string values are not escaped 当前字符串值不会被转义

2.2 Constants 常数

Each constant definition is like a field description with a default value, except that this value can never be changed programatically. This value assignment is indicated by use of an equal '=' sign, e.g.
每个常量定义类似于具有默认值的字段描述，但该值永远不能以编程方式更改。通过使用等于'='的符号来完成该值赋值，例如

```
constanttype CONSTANTNAME=constantvalue
```

For example: 例如：

```
int32 X=123
int32 Y=-123
string FOO="foo"
string EXAMPLE='bar'
```

Note 注意

Constants names have to be UPPERCASE 常量名称必须是大写的

3. Service Description Specification

服务描述规范

Services description are defined in .srv files in the srv/ directory of a ROS package.
服务描述在 ROS 包 srv/目录中的.srv 文件中定义。
A service description file consists of a request and a response msg type, separated by '—'. Any two .msg files concatenated together with a '—' are a legal service description.
服务描述文件由请求和响应消息类型组成，以" - "分隔。任何两个.msg 文件与' --- '连接在一起是合法的服务描述。
Here is a very simple example of a service that takes in a string and returns a string:
下面是一个非常简单的服务示例，它接受一个字符串并返回一个字符串：

```
string str
---
string str
```

We can of course get much more complicated (if you want to refer to a message from the same package you must not mention the package name):

当然可以实现更复杂的（如果想引用来自同一个包的消息，不能提到包名）：

```
#request constants
int8 FOO=1
int8 BAR=2
#request fields
int8 foobar
another_pkg/AnotherMessage msg
---
#response constants
uint32 SECRET=123456
#response fields
another_pkg/YetAnotherMessage val
CustomMessageDefinedInThisPackage value
uint32 an_integer
```

You cannot embed another service inside of a service.

无法在服务中嵌入其他服务。

About ROS2 client libraries

关于 ROS2 客户端库

Table of Contents 目录

Overview 概述

Client libraries are the APIs that allow users to implement their ROS code. They are what users use to get access to ROS concepts such as nodes, topics, services, etc. Client libraries come in a variety of programming languages so that users may write ROS code in the language that is best-suited for their application. For example, you might prefer to write visualization tools in Python because it makes prototyping iterations faster, while for parts of your system that are concerned with efficiency, the nodes might be better implemented in C++.

客户端库是允许用户实现其 ROS 代码的 API。它们是用户用来访问 ROS 概念（如节点、主题、服务等）的内容。客户端库有各种编程语言，因此用户可以使用最适合其应用程序的语言编写 ROS 代码。例如，可能更喜欢在 Python 中编写可视化工具，因为它可以更快地进行原型设计迭代，而对于系统中与效率相关的部分，可以在 C ++中更好地实现节点。

Nodes written using different client libraries are able to share messages with each other because all client libraries implement code generators that provide users with the capability to interact with ROS interface files in the respective language.

使用不同客户端库编写的节点能够彼此共享消息，因为所有客户端库都实现了代码生成器，这些代码生成器为用户提供了与相应语言的 ROS 接口文件交互的能力。

In addition to the language-specific communication tools, client libraries expose to users the core functionality that makes ROS “ROS”. For example, here is a list of functionality that can typically be accessed through a client library:

除了特定于语言的通信工具之外，客户端库还向用户展示了使 ROS 成为“ROS”的核心功能。例如，以下是通常可以通过客户端库访问的功能列表：

- Names and namespaces 命名和命名空间
- Time (real or simulated) 时间（真实或模拟）
- Parameters 参数
- Console logging 控制台记录

- Threading model 线程模型
- Intra-process communication 进程内通信

Supported client libraries 支持的客户端库

The C++ client library (`rclcpp`) and the Python client library (`rclpy`) are both client libraries which utilize common functionality in the RCL.

C ++客户端库（`rclcpp`）和 Python 客户端库（`rclpy`）都是 RCL 中常见功能的客户端库。

While the C++ and Python client libraries are maintained by the core ROS 2 team, members of the ROS 2 community have created additional client libraries:

虽然 C ++和 Python 客户端库由核心 ROS 2 团队维护，但 ROS 2 社区的成员已创建了其他客户端库：

- JVM and Android
- Objective C and iOS
- C#
- Swift
- Node.js
- Ada
- _.NET Core, UWP and C#

Common functionality: the RCL 常用功能：RCL

Most of the functionality found in a client library is not specific to the programming language of the client library. For example, the behavior of parameters and the logic of namespaces should ideally be the same across all programming languages. Because of this, rather than implementing the common functionality from scratch, client libraries make use of a common core ROS Client Library (RCL) interface that implements logic and behavior of ROS concepts that is not language-specific. As a result, client libraries only need to wrap the common functionality in the RCL with foreign function interfaces. This keeps client libraries thinner and easier to develop. For this reason the common RCL functionality is exposed with C interfaces as the C language is typically the easiest language for client libraries to wrap.

客户端库中的大多数功能并非特定于客户端库的编程语言。例如，参数的行为和命名空间的逻辑在理想情况下应该在所有编程语言中都是相同的。因此，客户端库不是从头开始实现通用功能，而是使用通用核心 ROS 客户端库（RCL）接口，该接口实现非特定语言的 ROS 概念的逻辑和行为。因此，客户端库只需要使用外部函数接口封装 RCL 中的公共功能。这使客户端库更透明（薄），更容易开发。因此，C 语言通常是用于客户端库封装的最简单的语言，因此公共 RCL 功能通过 C 接口公开。

In addition to making the client libraries light-weight, an advantage of having the common core is that the behavior between languages is more consistent. If any changes are made to the logic/behavior of the functionality in the core RCL – namespaces, for example – all client libraries that use the RCL will have these changes reflected. Furthermore, having the common core means that maintaining multiple client libraries becomes less work when it comes to bug fixes.

除了使客户端库轻量化之外，拥有共同核心的一个优点是语言之间的行为更加一致。如果对核心 RCL 中的功能的逻辑/行为进行任何更改-例如，命名空间-所有使用 RCL 的客户端库都会反映这些更改。此外，拥有通用核心意味着在修复错误时，维护多个客户端库的工作量会减少。

The API documentation for the RCL can be found here. 可以在此处找到 RCL 的 API 文档。

Language-specific functionality 语言特定功能

Client library concepts that require language-specific features/properties are not implemented in the RCL but instead are implemented in each client library. For example, threading models used by “spin” functions will have implementations that are specific to the language of the client library.

需要特定于语言的功能/属性的客户端库概念未在 RCL 中实现，而是在每个客户端库中实现。例如，“spin”函数使用的线程模型将具有特定于客户端库的语言的实现。

Demo 演示实例

For a walkthrough of the message exchange between a publisher using `rclpy` and a subscriber using `rclcpp`, we encourage you to watch this ROSCon talk starting at 17:25 (here are the slides).

有关发布器使用 `rclpy` 和订阅器使用之间的消息交换的演练 `rclcpp`，我们建议您从 17:25 开始观看此 ROSCon 演讲（这是幻灯片讲稿）。

Comparison to ROS 1 与 ROS 1 的比较

In ROS 1, all client libraries are developed “from the ground up”. This allows for the ROS 1 Python client library to be implemented purely in Python, for example, which brings benefits of such as not needing to compile code. However, naming conventions and behaviors are not always consistent between client libraries, bug fixes have to be done in multiple places, and there is a lot of functionality that has only ever been implemented in one client library (e.g. UDPROS).

在 ROS 1 中，所有客户端库都是“从头开始”开发的。例如，这允许 ROS 1 Python 客户端库纯粹用 Python 实现，这带来了诸如不需要编译代码等好处。但是，命名约定和行为在客户端库之间并不总是一致的，错误修复必须在多个位置完成，并且有许多功能只在一个客户端库（例如 UDPROS）中实现。

Summary 摘要

By utilizing the common core ROS client library, client libraries written in a variety of programming languages are easier to write and have more consistent behavior.

通过使用通用核心 ROS 客户端库，使各种编程语言编写的客户端库更易于编写并具有更一致的行为。

Logging and logger configuration

日志记录和日志记录器配置

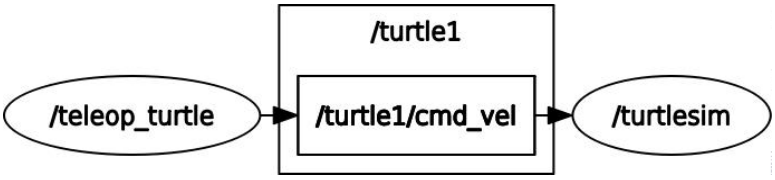
Table of Contents 目录

Overview 概述

The logging functionality currently supported is: 目前支持的日志记录功能是：

- Client libraries (`rclcpp` and `rclpy`) using a common logging library to provide:
  客户端库（`rclcpp`和`rclpy`）使用通用日志记录库来提供：
  - Log calls with a variety of filters.
    使用各种过滤器记录调用。
  - Hierarchy of loggers.
    记录器的层次结构。
  - Loggers associated with nodes that automatically use the node's name and namespace.
    与自动使用节点名称和命名空间的节点关联的记录器。
- Console output.
  控制台输出。
  - File output and functionality akin to rosout for remote consumption of messages is forthcoming.
    文件输出和功能类似于用于远程消费消息的 rosout 即将发布。

- Programmatic configuration of logger levels.
  记录器级别的编程配置。
  - Launch-time configuration of the default logger level is supported; config files and external configuration at run-time is forthcoming.
    支持默认记录器级别的启动时配置; 即将在运行时配置文件和外部配置。




```
relaybot@TPS2: ~/RobTool/ROS2/ros2_tutorials
relaybot@TPS2: ~/RobTool/ROS2/ros2_tutorials 80x11
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$ ros2 run turtlesim turtlesim_node
[INFO] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], the
ta=[0.000000]
[WARN] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.112445, y=5.5444
45])
[WARN] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.5444
45])
[WARN] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.5444
45])
[WARN] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.5444
relaybot@TPS2: ~/RobTool/ROS2/ros2_tutorials 80x11
Dashing
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$ source install/setup.bash
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$ ros2 run turtlesim
draw_square        --prefix           turtle_teleop_key
mimic              turtlesim_node
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$ ros2 run turtlesim  turtle_teleop_k
ey
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

[INFO] [turtlesim]: Starting turtlesim with node name /turtlesim

[INFO] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]

[WARN] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.112445, y=5.544445])

[WARN] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.544445])

Logger concepts 日志记录器概念

Log messages have a severity level associated with them: `DEBUG`, `INFO`, `WARN`, `ERROR` or `FATAL`, in ascending order.

日志消息与其相关联的严重性级别：`DEBUG`，`INFO`，`WARN`，`ERROR` 或者 `FATAL`，按升序排列。

A logger will only process log messages with severity at or higher than a specified level chosen for the logger.

记录器仅处理严重性等于或高于为日志记录器选择的指定级别的日志消息。

Each node (in `rclcpp` and `rclpy`) has a logger associated with it that automatically includes the node's name and namespace. If the node's name is externally remapped to something other than what is defined in the source code, it will be reflected in the logger name. Non-node loggers can also be created that use a specific name.

每个节点（如 `rclcpp` 和 `rclpy`）都有一个与之关联的记录器，它自动包含节点的名称和名称空间。如果节点的名称从外部重新映射到源代码中定义的名称以外的其他名称，则它将反映在记录器名称中。还可以创建使用特定名称的非节点记录器。

Logger names represent a hierarchy. If the level of a logger named "abc.def" is unset, it will defer to the level of its parent named "abc", and if that level is also unset, the default logger level will be used. When the level of logger "abc" is changed, all of its descendants (e.g. "abc.def", "abc.ghi.jkl") will have their level impacted unless their level has been explicitly set.

日志记录器名称表示层次结构。如果未设置名为“abc.def”的记录器的级别，它将推迟到其名为“abc”的父级别，如果该级别也未设置，则将使用默认记录器级别。当记录器“abc”的级别改变时，其所有后代（例如“abc.def”，“abc.ghi.jkl”）将对其级别产生影响，除非已明确设置其级别。

Logging usage 日志记录用法

In C++: 在 C ++中：

- See the logging demo for example usage.
  有关示例用法，请参考日志记录演示
- See the rclcpp documentation for an extensive list of functionality.
  有关功能的详尽列表，请参考 rclcpp 文档。

In Python: 在 Python 中：

- See the rclpy examples for example usage of a node's logger.
  有关节点记录器的示例用法，请参考 rclpy 示例。
- See the rclpy tests for example usage of keyword arguments (e.g. `skip_first`, `once`).
  有关关键字参数的使用示例，请参考 rclpy 测试（例如 `skip_first`，`once`）。

Logger configuration 日志记录器配置

Command line configuration of the default severity level

默认严重性级别的命令行配置

As of the Bouncy ROS 2 release, the default severity level for loggers can be configured from the command line with the following, for example (the level string is not case sensitive):
从 Bouncy ROS 2 及以后的版本开始，可以从命令行配置记录器的默认严重性级别，例如：
（级别字符串不区分大小写）

```
ros2 run demo_nodes_cpp listener _log_level:=debug
```

This will affect all loggers that have not explicitly been configured to use a particular severity level. Configuration of specific loggers from the command line is forthcoming.
这将影响未明确配置为使用特定严重性级别的所有记录器。即将从命令行配置特定记录器。

Programmatic configuration of individual loggers

各个记录器的编程配置

Logger configuration is still under development. For now, the severity level of individual loggers can be configured programmatically with, e.g.:
记录器配置仍在开发中。目前，可以通过编程方式配置各个记录器的严重性级别，例如：

In C++: 在 C ++中：

```
rcutils_logging_set_logger_level("logger_name", RCUTILS_LOG_SEVERITY_DEBUG);
```

In Python: 在 Python 中：

```
logger.set_level(rclpy.logging.LoggingSeverity.DEBUG)
rclpy.logging.set_logger_level('logger_name', rclpy.logging.LoggingSeverity.DEBUG)
```

The logging demo provides an example of manually exposing a service so that loggers can be configured externally; in the future we expect runtime configuration capabilities of loggers to be exposed automatically.
所述日志记录演示提供的手动开发服务，使得记录器可从外部配置; 在未来，希望记录器的运行时配置功能能够自动公开。

Console output configuration

控制台输出配置

By default, console output will be formatted to include the message severity, logger name, and the message. Information such as the file name, function name and line number of the log call are also available. Custom console output format can be configured with the `RCUTILS_CONSOLE_OUTPUT_FORMAT` environment variable: see the rcutils documentation for details. As `rclpy` and `rclcpp` both use `rcutils` for logging, this will effect all Python and C++ nodes.
默认情况下，控制台输出将被格式化为包括消息严重性，记录器名称和消息。还可以使用日志调用的文件名，函数名和行号等信息。可以使用 `RCUTILS_CONSOLE_OUTPUT_FORMAT` 环境变量配置自定义控制台输出格式：有关详细信息，请参考 rcutils 文档。由于 `rclpy` 和 `rclcpp` 都使用 `rcutils` 了日志记录，这将影响所有的 Python 和 C ++节点。

Features Status 功能状态

The features listed below are available in the current ROS 2 release. Unless otherwise specified, the features are available for all supported platforms (Ubuntu 18.04, OS X 10.12.x, Windows 10), DDS implementations (eProsima Fast RTPS, RTI Connext and ADLINK Opensplice) and programming language client libraries (C++ and Python). For planned future development, see the Roadmap.
下面列出的功能可在当前的 ROS 2 版本中找到。除非另有说明，否则这些功能适用于所有支持的平台（Ubuntu 18.04，OS X 10.12.x，Windows 10），DDS 实现版本（eProsima Fast RTPS、RTI Connext 和 ADLINK Opensplice）和编程语言客户端库（C ++和 Python）。有关未来发展的计划，请参考路线图。

Functionality 功能	Link 链接	Fine print
Discovery, transport and serialization over DDS 通过 DDS 进行发现，传输和序列化	Article	
Support for multiple DDS implementations,	Tutorials	Currently eProsima Fast RTPS, RTI Connext and

Functionality 功能	Link 链接	Fine print
chosen at runtime 支持在运行时选择的多个 DDS 实现		ADLINK OpenSplice are fully supported.
Common core client library that is wrapped by language-specific libraries 由特定语言的库封装的公共核心客户端库	Details	
Publish/subscribe over topics 发布/订阅主题	Sample code, Article	
Clients and services 客户端和服务端	Sample code	
Set/retrieve parameters 设置/检索参数	Sample code	
ROS 1 - ROS 2 communication bridge ROS 1 - ROS 2 通信桥接	Tutorial	Available for topics and services, not yet available for actions.
Quality of service settings for handling non-ideal networks 处理非理想网络的服务质量设置	Demo	
Inter- and intra-process communication using the same API 使用相同 API 的进程间和进程内通信	Demo	Currently only in C++.
Composition of node components at compile-, link- or `dlopen`-time 编译、链接或 `dlopen` 时间节点组件的组成	Demo	Currently only in C++.
Support for nodes with managed lifecycles 支持具有托管生命周期的节点	Demo	Currently only in C++.
DDS-Security support DDS-Security 支持	Demo	
Command-line introspection tools using an extensible framework 使用可扩展框架的命令行自检工具	Tutorial	
Launch system for coordinating multiple nodes 启动系统以协调多个节点	Tutorial	
Namespace support for nodes and topics 命名空间支持节点和主题	Article	
Static remapping of ROS names 静态重新映射 ROS 名称	Tutorial	
Demos of an all-ROS 2 mobile robot 全 ROS 2 移动机器人的演示	Demo	

Functionality 功能	Link 链接	Fine print
Preliminary support for real-time code 初步支持实时代码	Demo, demo	Linux only. Not available for Fast RTPS.
Preliminary support for “bare-metal” microcontrollers 对“bare-meta”微控制器的初步支持	Wiki	

Beside features of the platform the most impact of ROS comes from its available packages. The following are a few high profile packages which are available in the latest release:

除了平台的功能外，ROS 的最大影响来自其可用的包。以下是最新版本中提供的一些高级别功能的软件包：

- gazebo_ros_pkgs
- image_transport
- navigation2
- rosbag2
- RQt
- RViz2

Tutorials 教程

Basic 基础

- Using colcon to build packages 使用 colcon 编译包
- Using Ament 使用 Ament
- ament_cmake User Documentation ament_cmake 用户文档
- Cross-Compilation 交叉编译
- On the mixing of ament and catkin (catment) 关于 ament 和 catkin（catment）的混合
- Introspection with command line tools 使用命令行工具进行自检
- Overview and Usage of RQt RQt 的概述和使用
- Porting RQt plugins to Windows 将 RQt 插件移植到 Windows
- Passing ROS arguments to nodes via the command-line 通过命令行将 ROS 参数传递给节点
- Launching/monitoring multiple nodes with Launch 使用 Launch 启动/监控多个节点
- Migrating launch files from ROS 1 to ROS 2 将启动文件从 ROS 1 迁移到 ROS 2
- Working with multiple ROS 2 middleware implementations 使用多个 ROS 2 中间件实现
- Composing multiple nodes in a single process 在单个进程中组合多个节点
- Introduction to msg and srv interfaces msg 和 srv 接口简介
- New features in ROS 2 interfaces ROS 2 接口的新功能
- Defining custom interfaces (msg/srv) 定义自定义接口（msg / srv）
- Actions 操作
- Eclipse Oxygen with ROS 2 and rviz2 [community-contributed]
  Eclipse Oxygen 与 ROS 2 和 rviz2 [社区贡献]
- Building ROS2 on Linux with Eclipse Oxygen [community-contributed]
  使用 Eclipse Oxygen 在 Linux 上编译 ROS2 [社区贡献]
- Building Realtime Linux for ROS 2 [community-contributed]
  为 ROS 2 编译实时 Linux [社区贡献]
- Releasing a ROS 2 package with bloom
  发布 bloom 的 ROS 2 包

Advanced 高级

- Implement a custom memory allocator
  实现自定义内存分配器

Using Docker 使用

- Running 2 nodes in a single docker container [community-contributed]

在单个 docker 容器中运行 2 个节点[community-contributions]

- Running 2 nodes in 2 separate docker containers [community-contributed]
  在 2 个独立的 docker 容器中运行 2 个节点[community-contributions]

Demos 演示

- Use quality-of-service settings to handle lossy networks.
  使用服务质量设置来处理有损网络。
- Management of nodes with managed lifecycles.
  管理具有托管生命周期的节点。
- Efficient intra-process communication.
  高效的进程内通信。
- Bridge communication between ROS 1 and ROS 2.
  ROS 1 和 ROS 2 之间的桥接通信。
- Recording and playback of topic data with rosbag using the ROS 1 bridge.
  使用 ROS 1 桥接器运行 rosbag 记录和回放主题数据。
- Turtlebot 2 demo using ROS 2.
  使用 ROS 2 的 Turtlebot 2 演示。
- TurtleBot 3 demo using ROS 2. [community-contributed]
  使用 ROS 2 的 TurtleBot 3 演示。[社区贡献]
- Using tf2 with ROS 2.
  使用 tf2 和 ROS 2。
- Write real-time safe code that uses the ROS 2 APIs.
  编写使用 ROS 2 API 的实时安全代码。
- Use the rclpy API to write ROS 2 programs in Python.
  使用 rclpy API 在 Python 中编写 ROS 2 程序。
- Use the robot state publisher to publish joint states and TF.
  使用机器人状态发布器发布联合状态和 TF。
- Use DDS-Security.
  使用 DDS-Security。
- Logging and logger configuration.
  日志记录和日志记录器配置。

Examples 示例

- Python and C++ minimal examples.
  Python 和 C ++最小的例子。

Using colcon to build packages

使用 colcon 编译包

This is a brief tutorial of how to create and build a ROS 2 workspace with `colcon`. It is a practical tutorial and not designed to replace the core documentation.

这是一个使用 `colcon` 如何创建和编译 ROS 2 工作区的简要教程。这是一个实用的教程，并非旨在取代核心文档。

ROS 2 releases before Bouncy used `ament_tools` described in the ament tutorial.

如果使用 Bouncy 或之前发行版本的 ROS 2 参考 ament 教程中 `ament_tools` 使用说明。

在 Crystal 和 Dashing 中均使用 colcon 进行编译。

Background 背景

`colcon` is an iteration on the ROS build tools `catkin_make`, `catkin_make_isolated`, `catkin_tools` and `ament_tools`. For more information on the design of colcon see this document.

`colcon` 是对 ROS 编译（编译）工具 `catkin_make`，`catkin_make_isolated`，`catkin_tools` 和 `ament_tools` 迭代。有关 **colcon** 设计的更多信息，请参考此文档。

The source code can be found in the colcon GitHub organization.

源代码可以在 colcon GitHub 组织中找到。

Prerequisites 预备基础

Install colcon 安装 colcon

Linux

```
sudo apt install python3-colcon-common-extensions
```

python3-colcon-common-extensions is already the newest version (0.2.0-2).

OS X

```
python3 -m pip install colcon-common-extensions
```

Windows

```
pip install -U colcon-common-extensions
```

Install ROS 2 安装 ROS2

To build the samples, you will need to install ROS 2.

编译示例前，需要先安装 ROS 2。

Follow the installation instructions.

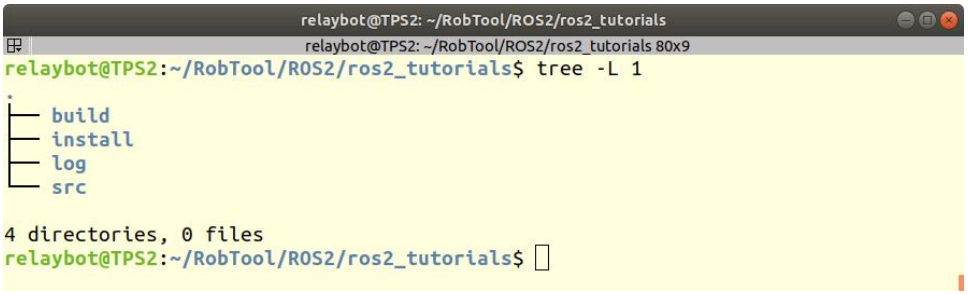
请参考安装说明。

Attention 注意

If installing from Debian packages, this tutorial requires the desktop installation.

如果从 Debian 软件包安装，本教程需要桌面安装。

Basics 基础知识



A ROS workspace is a directory with a particular structure. Commonly there is a `src` subdirectory. Inside that subdirectory is where the source code of ROS packages will be located. Typically the directory starts otherwise empty.

ROS 工作空间是具有特定结构的目录。通常有一个 `src` 子目录。这个 src 子目录中是 ROS 包的源代码所在的位置。通常，通常都从这个目录开始，否则为空。

colcon does out of source builds. By default it will create the following directories as peers of the `src` directory:

colcon 完成源代码编译。默认情况下，它将创建以下目录作为 `src` 目录的对等项：

The `build` directory will be where intermediate files are stored. For each package a subfolder will be created in which e.g. CMake is being invoked.

其中 `build` 目录将是存储中间文件的位置。对于每个包，将创建一个子文件夹，例如调用 CMake。

The `install` directory is where each package will be installed to. By default each package will be installed into a separate subdirectory.

其中 `install` 目录是每个软件包将安装到的目录。默认情况下，每个包都将安装到单独的子目录中。

The `log` directory contains various logging information about each colcon invocation.

其中 `log` 目录包含有关每个 colcon 调用的各种日志记录信息。

Note 注意

Compared to catkin there is no `devel` directory. 与 catkin 相比，没有 `devel` 目录。

Create a workspace 创建工作区

First, create a directory (`ros2_example_ws`) to contain our workspace:

首先，创建一个目录（ros2_example_ws）来包含自定义的工作区：

Linux/OS X

```
mkdir -p ~/ros2_example_ws/src
cd ~/ros2_example_ws
```

Windows

```
md \dev\ros2_example_ws\src
cd \dev\ros2_example_ws
```

At this point the workspace contains a single empty directory src:
此时工作空间包含一个空目录文件夹 src：

```
.
└── src
1 directory, 0 files
```

Add some sources 添加一些来源

Let’s clone the examples repository into the src directory of the workspace:
将示例库克隆到工作区的 src 目录中：

```
git clone https://github.com/ros2/examples src/examples
```

Attention 注意

It is recommended to checkout a branch that is compatible with the version of ROS that was installed (e.g. crystal、dashing).
建议检查与已安装的 ROS 版本兼容的分支（例如 crystal、dashing）。

```
cd ~/ros2_example_ws/src/examples/
git checkout $ROS_DISTRO
cd ~/ros2_example_ws
```

Now the workspace should have the source code to the ROS 2 examples:
现在工作区应该有 ROS 2 示例的源代码：

```
.
└── src
    └── examples
        ├── CONTRIBUTING.md
```

```
        ├── LICENSE
        ├── rclcpp
        ├── rclpy
        └── README.md
4 directories, 3 files
```

Source an underlay 导入底层

It is important that we have sourced the environment for an existing ROS 2 installation that will provide our workspace with the necessary build dependencies for the example packages. This is achieved by sourcing the setup script provided by a binary installation or a source installation, ie. another colcon workspace (see Installation). We call this environment an **underlay**.

重要的是，为现有的 ROS 2 安装提供环境，这将为编译工作区示例包提供必要的依赖性。通过获取二进制安装或源安装提供的安装脚本来实现的，即：另一个 colcon 工作区（请参考安装）。通常将此环境称为**底层**。

Our workspace, `ros2_examples_ws`, will be an **overlay** on top of the existing ROS 2 installation. In general, it is recommended to use an overlay when you plan to iterate on a small number of packages, rather than putting all of your packages into the same workspace.

工作空间 `ros2_examples_ws` 将**叠加**在现有的 ROS 2 安装之上。通常，建议在计划迭代少量软件包时使用覆盖，而不是将所有软件包放在同一个工作区中。

Build the workspace 编译工作区

Attention 注意

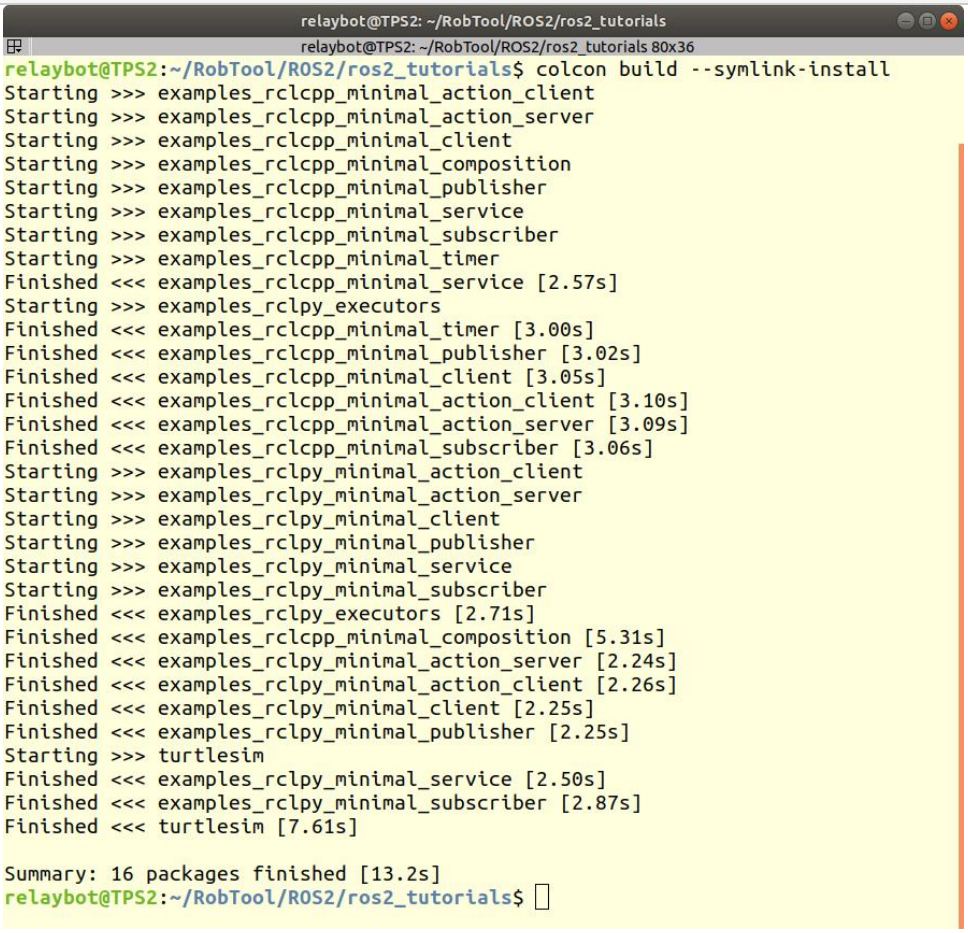
To build packages on Windows you need to be in a Visual Studio environment, see Building the ROS 2 Code for more details.

要在 Windows 上编译软件包，需要在 Visual Studio 环境中，请参考编译 ROS 2 代码以获取更多详细信息。

In the root of the workspace, run `colcon build`. Since build types such as `ament_cmake` do not support the concept of the `devel` space and require the package to be installed, colcon supports the option `--symlink-install`. This allows the installed files to be changed by changing the files in the `source` space (e.g. Python files or other not compiled resourced) for faster iteration.

在工作区的根目录中，运行 `colcon build`。由于编译类型 `ament_cmake`（例如不支持 `devel` 的概念并且需要安装包），因此 colcon 支持选项`--symlink-install`。这允许通过更改 `source` 空间中的文件（例如 Python 文件或其他未编译的资源）来更改已安装的文件，以便更快地进行迭代。

```
colcon build --symlink-install
```



After the build is finished, we should see the `build`, `install`, and `log` directories:

编译完成后，可以看到 `build`，`install` 和 `log` 文件夹目录：

```
.
├── build
├── install
├── log
└── src
4 directories, 0 files
```

Run tests 运行测试

To run tests for the packages we just built, run the following:

为刚刚编译的包运行测试，请运行以下命令：

```
colcon test
```

```
relaybot@TPS2: ~/RobTool/ROS2/ros2_tutorials
relaybot@TPS2: ~/RobTool/ROS2/ros2_tutorials 80x39
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$ colcon test
Starting >>> examples_rclcpp_minimal_action_client
Starting >>> examples_rclcpp_minimal_action_server
Starting >>> examples_rclcpp_minimal_client
Starting >>> examples_rclcpp_minimal_composition
Starting >>> examples_rclcpp_minimal_publisher
Starting >>> examples_rclcpp_minimal_service
Starting >>> examples_rclcpp_minimal_subscriber
Starting >>> examples_rclcpp_minimal_timer
Finished <<< examples_rclcpp_minimal_action_client [0.09s]
Finished <<< examples_rclcpp_minimal_action_server [0.09s]
Finished <<< examples_rclcpp_minimal_client [0.08s]
Finished <<< examples_rclcpp_minimal_composition [0.07s]
Finished <<< examples_rclcpp_minimal_service [0.06s]
Finished <<< examples_rclcpp_minimal_publisher [0.07s]
Starting >>> examples_rclpy_executors
Starting >>> examples_rclpy_minimal_action_client
Starting >>> examples_rclpy_minimal_action_server
Starting >>> examples_rclpy_minimal_client
Starting >>> examples_rclpy_minimal_publisher
Starting >>> examples_rclpy_minimal_service
Finished <<< examples_rclcpp_minimal_subscriber [0.11s]
Finished <<< examples_rclcpp_minimal_timer [0.10s]
Starting >>> examples_rclpy_minimal_subscriber
Starting >>> turtlesim
Finished <<< examples_rclpy_minimal_action_client [2.32s]
Finished <<< examples_rclpy_executors [2.32s]   [ with test failures ]
Finished <<< examples_rclpy_minimal_action_server [2.33s]
Finished <<< examples_rclpy_minimal_client [2.74s]      [ with test failures ]
Finished <<< turtlesim [0.90s]
Finished <<< examples_rclpy_minimal_publisher [3.13s]   [ with test failures ]
Finished <<< examples_rclpy_minimal_service [3.49s]     [ with test failures ]
Finished <<< examples_rclpy_minimal_subscriber [1.98s]  [ with test failures ]

Summary: 16 packages finished [4.13s]
  5 packages had test failures: examples_rclpy_executors examples_rclpy_minimal_
client examples_rclpy_minimal_publisher examples_rclpy_minimal_service examples_
rclpy_minimal_subscriber
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$
```

Source the environment 导入环境

When colcon has completed building successfully, the output will be in the `install` directory. Before you can use any of the installed executables or libraries, you will need to add them to your path and library paths. colcon will have generated bash/bat files in the `install` directory to help setup the environment. These files will add all of the required elements to your path and library paths as well as provide any bash or shell commands exported by packages.

当 colcon 成功完成编译后，输出将在 `install` 目录中。在使用任何已安装的可执行文件或库之前，需要将它们添加到路径和库路径中。colcon 将在 `install` 目录中生成 bash / bat 文件以帮助设置环境。这些文件将向路径和库路径添加所有必需元素，并提供由包导出的任何 bash 或 shell 命令。

Linux/OS X

```
. install/setup.bash
```

Or 或者

```
source install/setup.bash
```

Windows

```
call install\setup.bat
```

Try a demo 试试示例

With the environment sourced we can run executables built by colcon. Let's run a subscriber node from the examples:

在环境导入后，可以运行 colcon 编译的可执行文件。从示例中运行订阅器节点如下：

```
ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function
```

In another terminal, let's run a publisher node (don't forget to source the setup script):

在另一个终端中，运行一个发布器节点（不要忘记导入安装脚本）：

```
ros2 run examples_rclcpp_minimal_publisher publisher_member_function
```

You should see messages from the publisher and subscriber with numbers incrementing.

应该看到来自发布器和订阅器的消息，并且数字会递增。


```
relaybot@TPS2: ~
relaybot@TPS2: ~ 80x11
relaybot@TPS2:~$ ros2 run examples_rclcpp_minimal_subscriber subscriber_member_f
unction
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 0'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 1'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 2'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 3'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 4'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 5'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 6'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 7'
[INFO] [minimal_subscriber]: I heard: 'Hello, world! 8'
relaybot@TPS2: ~ 80x11
relaybot@TPS2:~$ ros2 run examples_rclcpp_minimal_publisher publisher_member_fun
ction
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 0'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 1'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 2'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 3'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 4'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 5'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 6'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 7'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 8'
```

Create your own package 创建自定义的包

colcon uses the `package.xml` specification defined in REP 149 (format 2 is also supported).

colcon 使用 REP 149 中的规范定义 `package.xml`（也支持格式 2）。

colcon supports multiple build types. The recommended build types are `ament_cmake` and `ament_python`. Also supported are pure `cmake` packages.

colcon 支持多种编译类型。推荐的编译类型是 `ament_cmake` 和 `ament_python`。也支持纯 `cmake` 包。

An example of an `ament_python` build is the ament_index_python package , where the setup.py is the primary entry point for building.

`ament_python` 编译的一个示例是 ament_index_python 包，其中 setup.py 是编译的主要入口点。

A package such as demo_nodes_cpp uses the ament_cmake build type, and uses CMake as the build tool.

诸如 demo_nodes_cpp 包使用 ament_cmake 编译类型，并使用 CMake 作为编译工具。

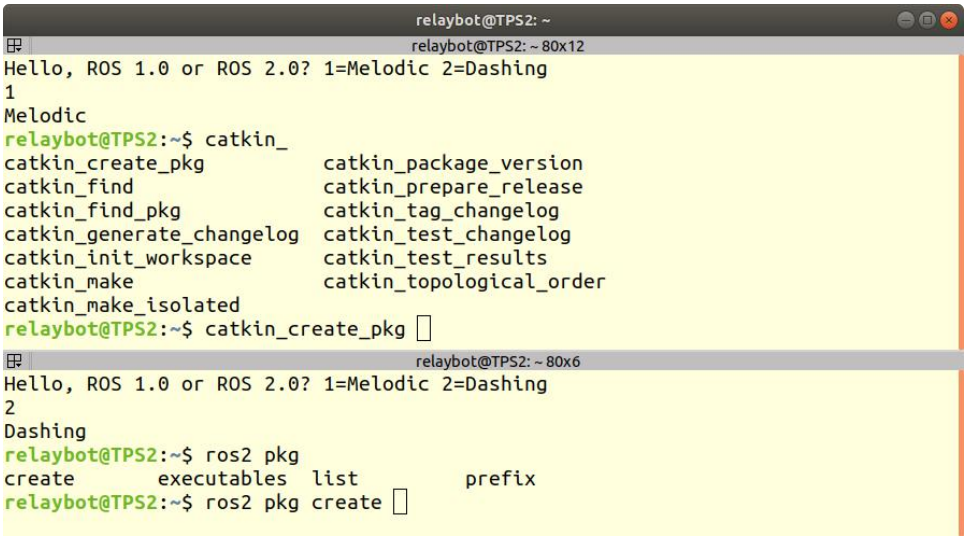
For convenience, you can use the tool ros2 pkg create to create a new package based on a template.

为方便起见，可以使用该工具 ros2 pkg create 基于模板创建新包。

Note 注意

For catkin users, this is the equivalent of catkin_create_package.

对于 catkin 用户来说，这相当于 catkin_create_package。



注意区别，ROS1 和 ROS2，创建自定义包的区别。

Tips 注意事项

If you do not want to build a specific package place an empty file named COLCON_IGNORE in the directory and it will not be indexed.

如果不想构建特定的包，请在目录中指定一个 COLCON_IGNORE 空文件，将不会编入索引。

If you want to avoid configuring and building tests in CMake packages you can pass: --cmake-args -DBUILD_TESTING=0.

如果想避免在 CMake 软件包中配置和构建测试，可以通过：--cmake-args -DBUILD_TESTING=0。

If you want to run a single particular test from a package:

如果要从包中运行单个特定测试：

```
colcon test --packages-select YOUR_PKG_NAME --ctest-args -R YOUR_TEST_IN_PKG
```

Using Ament 使用 Ament

Warning 警告

As of ROS 2 Bouncy the recommended build tool is ``colcon`` described in the colcon tutorial. The current default branch as well as releases after Bouncy do not include `ament_tools` anymore.

从 ROS 2 Bouncy 开始，推荐的编译工具是 colcon 教程中**提及的``colcon``**。当前的默认发行版以及 Bouncy 之后的版本不再将 `ament_tools` 包括在内。

由于介绍的教程针对 Crystal 和 Dashing 发行版本的 ROS 2，所以设计 ament 的章节就省略了，如果需要了解更多细节，请查阅官方文档。

Introspection with command line tools

使用命令行工具进行自检

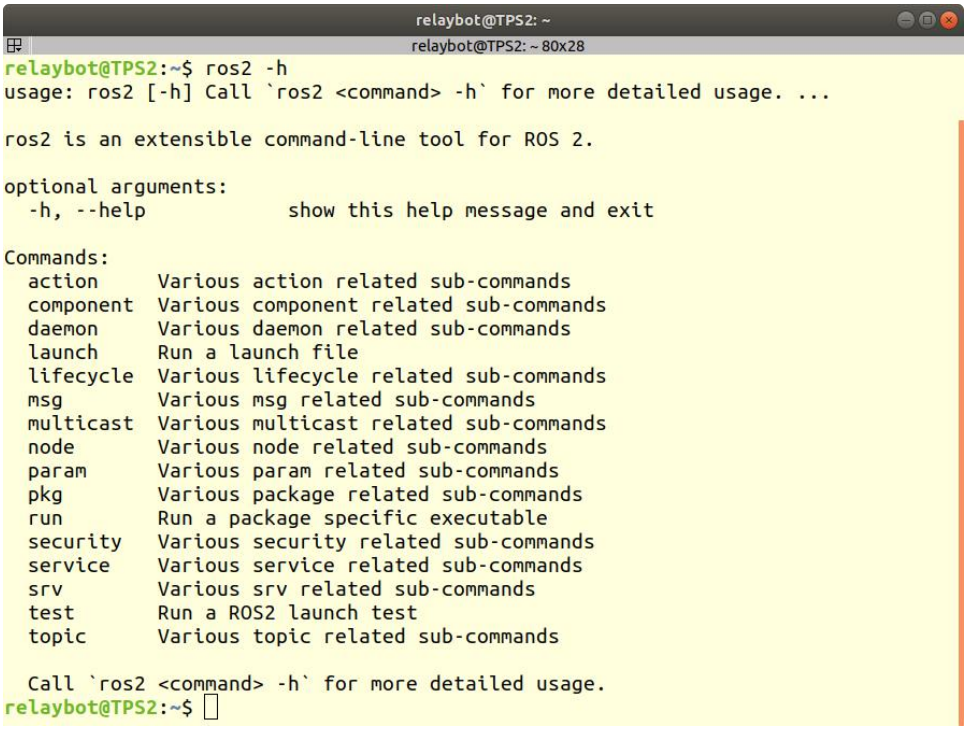
ROS 2 includes a suite of command-line tools for introspecting a ROS 2 system.
ROS 2 包括一套用于 ROS 2 系统自检的命令行工具。

Usage 用法

The main entry point for the tools is the command `ros2`, which itself has various sub-commands for introspecting and working with nodes, topics, services, and more.
这些工具的主要入口点是命令 `ros2`，它本身具有各类子命令，用于自检和处理节点、主题、服务等。

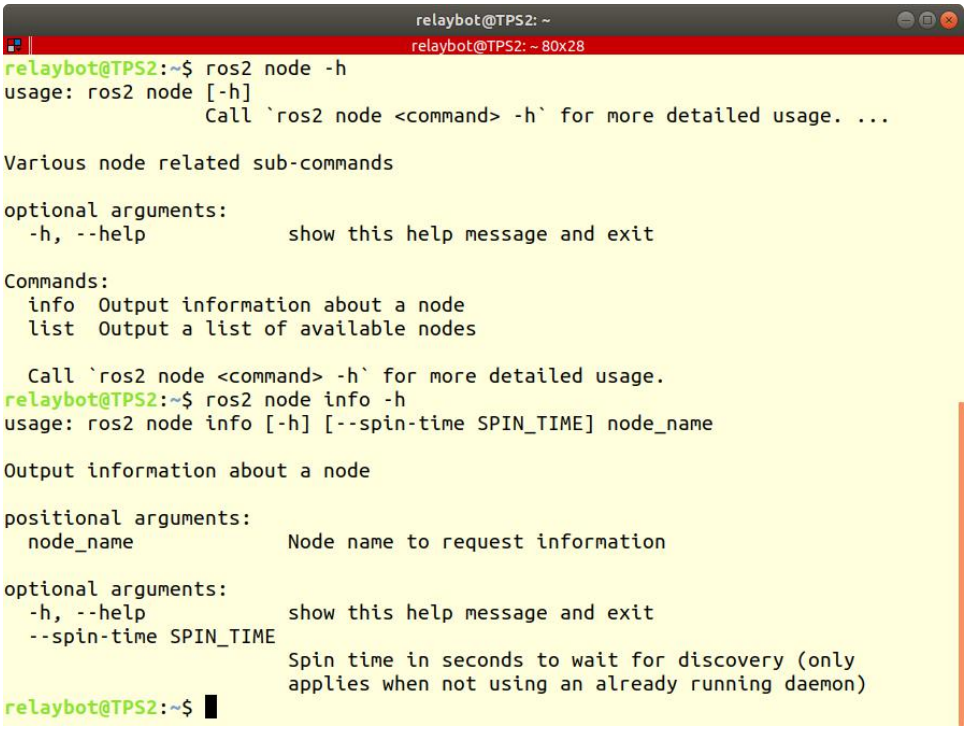
To see all available sub-commands run: 要查看运行的所有可用子命令：

```
ros2 --help
```



每个子命令的细节，如节点:

```
ros2 node -h
```



Examples of sub-commands that are available include:

可用的子命令示例包括：

- daemon: Introspect/configure the ROS 2 daemon 自检/配置 ROS 2 守护程序
- launch: Run a launch file 运行启动文件
- lifecycle: Introspect/manage nodes with managed lifecycles 使用托管生命周期自检/管理节点
- msg: Introspect `msg` types 自检 `msg` 类型
- node: Introspect ROS nodes 自检 ROS 节点
- param: Introspect/configure parameters on a node 在节点上自检/配置参数
- pkg: Introspect ROS packages 自检 ROS 包
- run: Run ROS nodes 运行 ROS 节点
- security: Configure security settings 配置安全性设置
- service: Introspect/call ROS services 自检/调用 ROS 服务
- srv: Introspect `srv` types 自检 `srv` 类型
- topic: Introspect/publish ROS topics 自检/发布 ROS 主题

Example 示例

To produce the typical talker-listener example using command-line tools, the `topic` sub-command can be used to publish and echo messages on a topic.

要使用命令行工具生成典型的 talker-listener 示例，`topic` 子命令可用于发布和回显主题上的消息。

Publish messages in one terminal with:

在一个终端中发布消息：

```
$ ros2 topic pub /chatter std_msgs/String "data: Hello world"
publisher: beginning loop
```

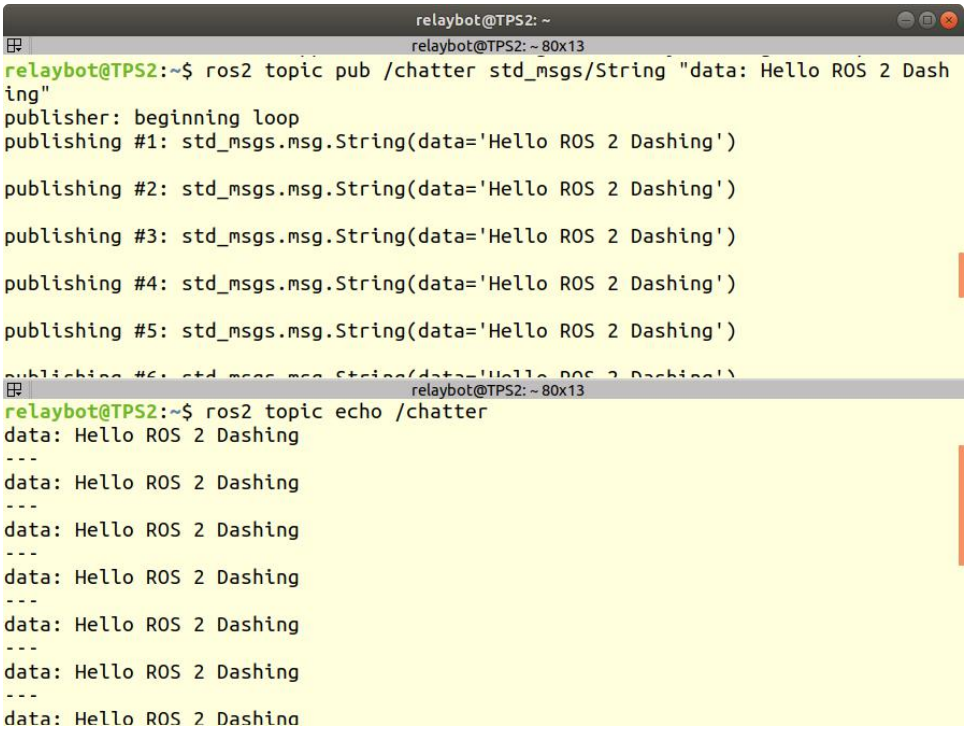
```
publishing std_msgs.msg.String(data='Hello world')

publishing std_msgs.msg.String(data='Hello world')
```

Echo messages received in another terminal with:

在另一个终端收到的回应消息：

```
$ ros2 topic echo /chatter
data: Hello world

data: Hello world
```

如果 ok，可以看到如下：



Behind the scenes 背后机制

ROS 2 uses a distributed discovery process for nodes to connect to each other. As this process purposefully does not use a centralized discovery mechanism (like the ROS Master in ROS 1), it can take time for ROS nodes to discover all other participants in the ROS graph. Because of this, there is a long-running daemon in the background that stores information about the ROS graph to provide faster responses to queries, e.g. the list of node names.

ROS 2 使用分布式发现过程来连接节点。由于此过程刻意不使用集中式发现机制（如 ROS 1 中的 ROS Master），因此 ROS 节点可能需要一段时间才能发现 ROS 图中的所有其他参与者。因此，后台存在一个长时间运行的守护程序，它存储有关 ROS 图的信息，以便更快地响应查询，例如节点名称列表。

The daemon is automatically started when the relevant command-line tools are used for the first time. You can run `ros2 daemon --help` for more options for interacting with the daemon.

首次使用相关命令行工具时，守护程序将自动启动。可以运行 ros2 daemon --help 更多选项以与守护程序进行交互。

```
relaybot@TPS2: ~
relaybot@TPS2: ~ 80x25
relaybot@TPS2:~$ ros2 daemon status
The daemon is not running
relaybot@TPS2:~$ ros2 daemon
usage: ros2 daemon [-h]
                   Call `ros2 daemon <command> -h` for more detailed usage.
                   ...

Various daemon related sub-commands

optional arguments:
  -h, --help            show this help message and exit

Commands:
  start   Start the daemon if it isn't running
  status  Output the status of the daemon
  stop    Stop the daemon if it is running

  Call `ros2 daemon <command> -h` for more detailed usage.
relaybot@TPS2:~$ ros2 daemon start
The daemon has been started
relaybot@TPS2:~$ ros2 daemon status
The daemon is running
relaybot@TPS2:~$ ros2 daemon stop
The daemon has been stopped
relaybot@TPS2:~$
```

Implementation 实现

The source code for the ros2 command is available at https://github.com/ros2/ros2cli

关于 ros2 命令的源代码可在网页 https://github.com/ros2/ros2cli 上找到。

The ros2 tool has been implemented as a framework that can be extended via plugins. For example, the sros2 package provides a security sub-command that is automatically detected by the ros2 tool if the sros2 package is installed.

ros2 工具实现了可通过插件扩展的框架。例如，sros2 包提供了一个 security 子命令，ros2 如果安装了 sros2 包，工具会自动检测该子命令。

```
relaybot@TPS2: ~
relaybot@TPS2: ~ 80x22
relaybot@TPS2:~$ ros2 security -h
usage: ros2 security [-h]
                     Call `ros2 security <command> -h` for more detailed
                     usage. ...

Various security related sub-commands

optional arguments:
  -h, --help            show this help message and exit

Commands:
  create_key          Create key
  create_keystore     Create keystore
  create_permission   Create permission
  distribute_key      Distribute key
  generate_artifacts  Generate keys and permission files from a list of identiti
es and policy files
  generate_policy     Generate XML policy file from ROS graph data
  list_keys           List keys

  Call `ros2 security <command> -h` for more detailed usage.
relaybot@TPS2:~$
```

Overview and Usage of RQt

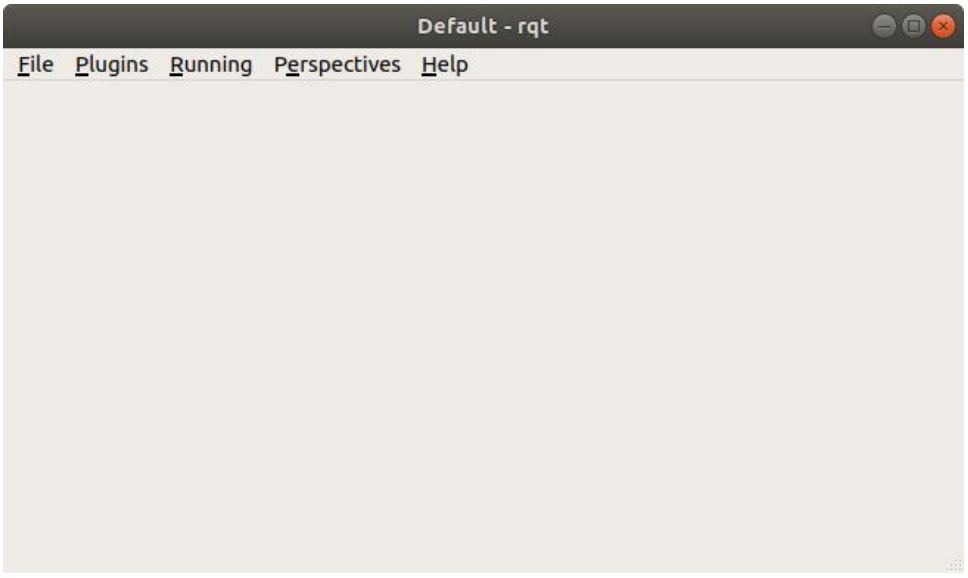
RQt 的概述和用法

Overview 概述

RQt is a graphical user interface framework that implements various tools and interfaces in the form of plugins. One can run all the existing GUI tools as dockable windows within RQt! The tools can still run in a traditional standalone method, but RQt makes it easier to manage all the various windows in a single screen layout.
RQt 是一个图形用户界面框架，以插件的形式实现各种工具和接口。可以将所有现有的 GUI 工具作为 RQt 中的可停靠窗口运行！这些工具仍然可以在传统的独立方法中运行，但 RQt 使得在单个屏幕布局中管理所有各种窗口变得更加容易。

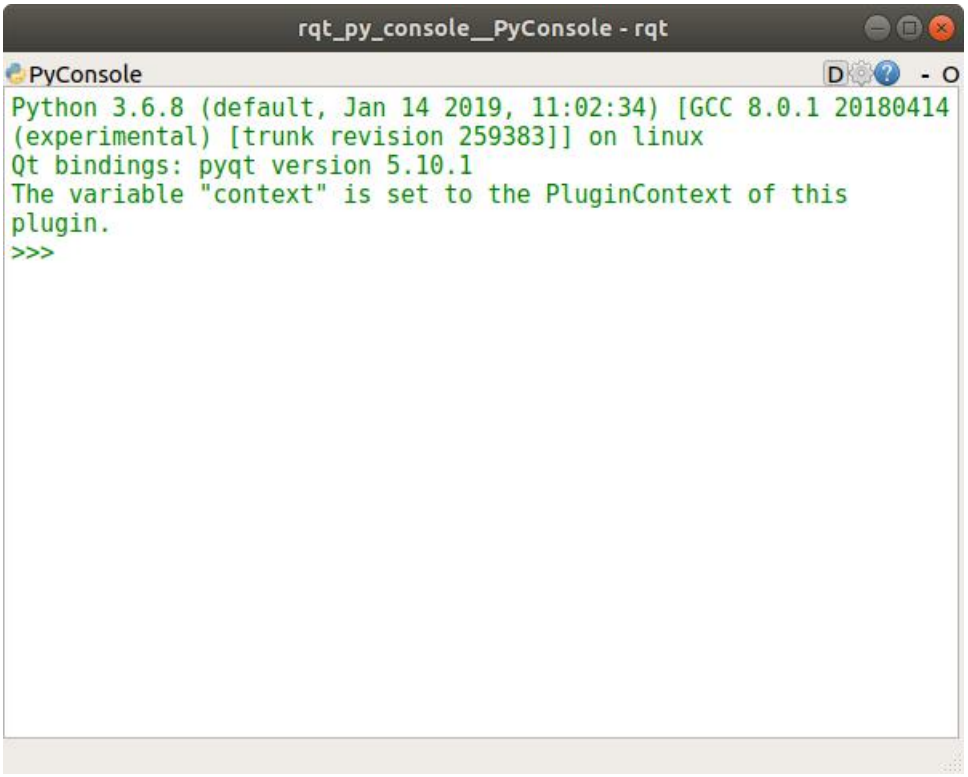
You can run any RQt tools/plugins easily by:
可以通过以下方式轻松运行任何 RQt 工具/插件（ros2 run rqt_gui rqt_gui ）：

```
rqt
```



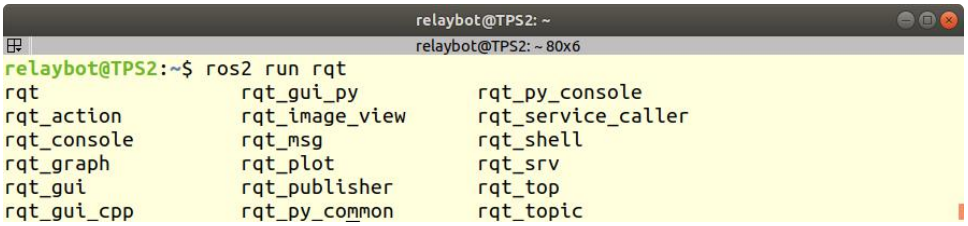
This GUI allows you to choose any available plugins on your system. You can also run plugins in standalone windows. For example, RQt Python Console:
此 GUI 允许选择系统上的任何可用插件。还可以在独立窗口中运行插件。例如，RQt Python 控制台：

```
ros2 run rqt_py_console rqt_py_console
```



Users can create their own plugins for RQt with either `Python` or `C++`. Over 20 plugins were created in ROS 1 and these plugins are currently being ported to ROS 2 (as of Dec 2018, more info).

用户创建自定义 RQT 插件的方法有两种 `Python` 或 `C++`。在 ROS 1 中创建了 20 多个插件，这些插件目前正被移植到 ROS 2（截至 2019 年 09 月，更多信息）。



System setup 系统设置

Installing From Debian 从 Debian 安装

```
sudo apt install ros-$ROS_DISTRO-rqt*
```

Building From Source 从源代码编译

See Building RQt from Source. 请参考从源编译 RQt

RQt Components Structure RQt 组件结构

RQt consists of three metapackages: RQt 由三个元数据包组成：

- rqt - core infrastucture modules.
  rqt - 核心基础设施模块。
- **rqt_common_plugins - Backend tools for building tools.**
  rqt_common_plugins **-** 用于编译工具的后端工具。

  TODO: as of Dec 2018 this metapackage isn't available in ROS 2 since not all plugins it contains have been ported yet.

  TODO：截至 2018 年 12 月，这个元数据包在 ROS 2 中不可用，因为它并不包含所有插件。
- **rqt_robot_plugins - Tools for interacting with robots during runtime.**
  rqt_robot_plugins **-** 用于在运行时与机器人交互的工具。

  TODO: as of Dec 2018 this metapackage isn't available in ROS 2 since not all plugins it contains have been ported yet.

  TODO：截至 2018 年 12 月，这个元数据包在 ROS 2 中不可用，因为它并不包含所有插件。

Advantage of RQt framework RQt 框架的优点

Compared to building your own GUIs from scratch:

与从头开始编译自己的 GUI 相比：

- Standardized common procedures for GUI (start-shutdown hook, restore previous states).
  GUI 的标准化通用过程（启动 - 关闭挂钩，恢复以前的状态）。
- Multiple widgets can be docked in a single window.
  多个小部件可以停靠在一个窗口中。
- Easily turn your existing Qt widgets into RQt plugins.
  轻松将现有的 Qt 小部件转换为 RQt 插件。
- Expect support at ROS Answers (ROS community website for the questions).
  可获得 ROS Answers（ROS 社区网站上的问题）的支持。

From system architecture's perspective:

从系统架构的角度来看：

- Support multi-platform (basically wherever QT and ROS run) and multi-language (Python, C++).
  支持多平台（基本上是 QT 和 ROS 运行的地方）和多语言（Python，C++）。
- Manageable lifecycle: RQt plugins using common API makes maintainance & reuse easier.
  可管理的生命周期：使用通用 API 的 RQt 插件使维护和重用更容易。

Further Reading 扩展阅读

ROS 2 Discourse announcment of porting to ROS2. ROS 2 Discourse 宣布移植到 ROS2。

RQt for ROS 1 documentation. RQt for ROS 1 文档。

Brief overview of RQt (from a Willow Garage intern blog post). RQt 概述（来自 Willow Garage 实习生博客文章）。

Passing ROS arguments to nodes via the command-line

通过命令行将 ROS 参数传递给节点

All ROS nodes take a set of arguments that allow various properties to be reconfigured. Examples include configuring the name/namespace of the node, topic/service names used, and parameters on the node.
所有 ROS 节点都采用一组参数，可以重新配置各种属性。示例包括配置节点的名称/命名空间，使用的主题/服务名称以及节点上的参数。

Note: all features on this page are only available as of the ROS 2 Bouncy release.
注意：此网页上的所有功能仅在 ROS 2 Bouncy 及之后发行版本中提供，适用于 Bouncy、Crystal 和 Dashing。

Name remapping 名称重新映射

Names within a node (e.g. topics/services) can be remapped using the syntax `<old name>:=<new name>`. The name/namespace of the node itself can be remapped using `__node:=<new node name>` and `__ns:=<new node namespace>`.
可以使用`<old name>:=<new name>`语法重新映射节点（例如主题/服务）的名称。可以使用 `__node:=<new node name>` 和 `__ns:=<new node namespace>`重新映射节点本身的名称/命名空间。

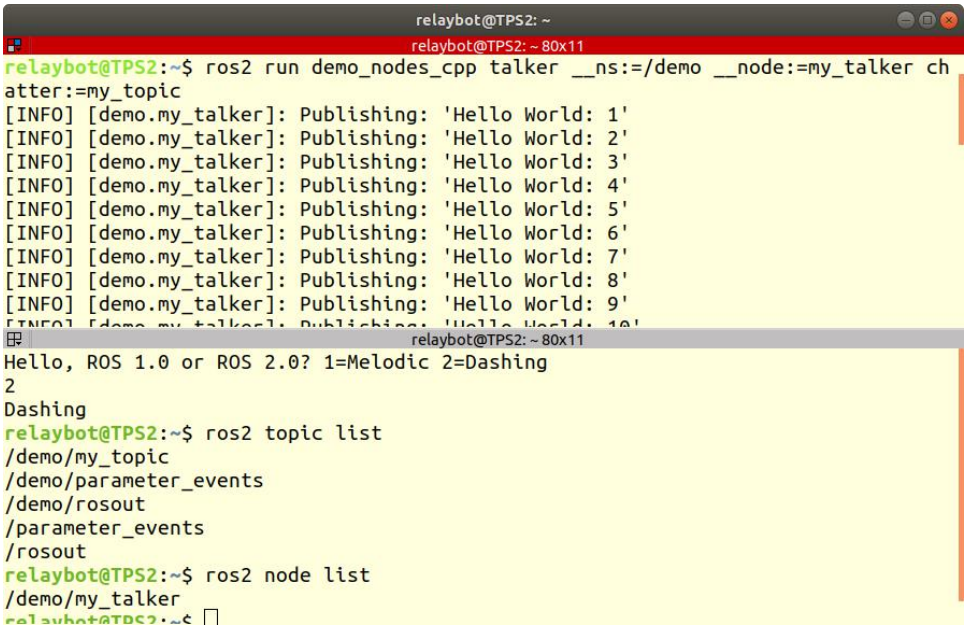
Note that these remappings are “static” remappings, in that they apply for the lifetime of the node. “Dynamic” remapping of names after nodes have been started is not yet supported.
请注意，这些重映射是“静态”重映射，因为它们适用于节点的生命周期。尚未支持节点启动后“动态”重新映射名称。

See this design doc for more details on remapping arguments (not all functionality is available yet).
有关重新映射参数的更多详细信息，请参考此设计文档（并非所有功能都可用）。

Example 示例

The following invocation will cause the `talker` node to be started under the node name `my_talker`, publishing on the topic named `my_topic` instead of the default of `chatter`. The namespace, which must start with a forward slash, is set to `/demo`, which means that topics are created in that namespace (`/demo/my_topic`), as opposed to globally (`/my_topic`).
以下调用将导致节点 `talker` 以节点名称 `my_talker` 下启动，发布消息到主题 `my_topic` 上而不是默认值 `chatter`。必须以正斜杠开头的命名空间设置为`/demo`，这意味着在该命名空间（`/demo/my_topic`）中创建主题，而不是全局（`/my_topic`）。

```
ros2 run demo_nodes_cpp talker __ns:=/demo __node:=my_talker chatter:=my_topic
```



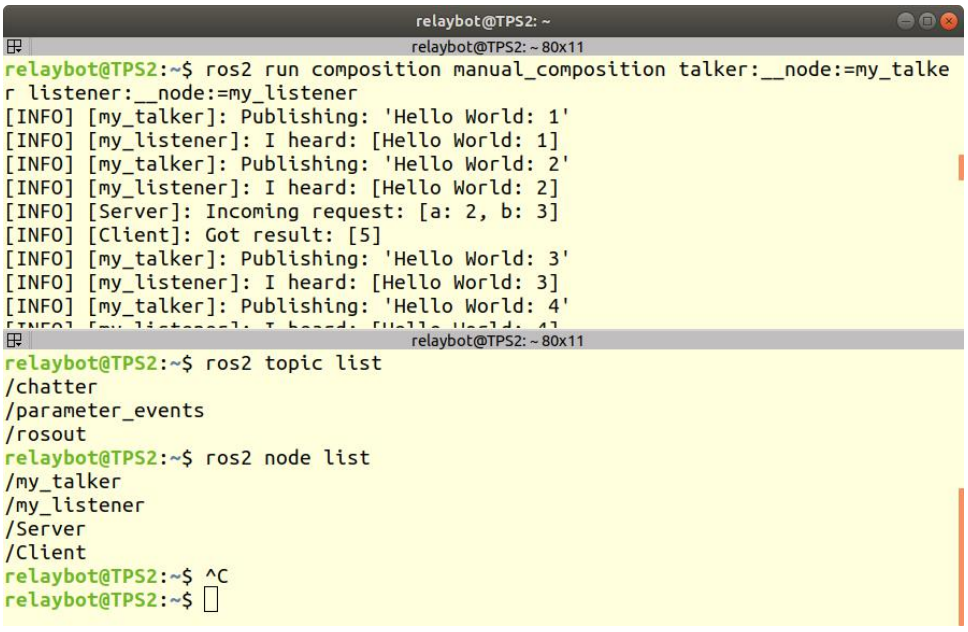
Passing remapping arguments to specific nodes

将重映射参数传递给特定节点

If multiple nodes are being run within a single process (e.g. using Composition), remapping arguments can be passed to a specific node using its name as a prefix. For example, the following will pass the remapping arguments to the specified nodes:

如果在单个进程内运行多个节点（例如，使用 Composition ），则可以使用其名称作为前缀重新映射参数传递给特定节点。例如，以下内容将重映射参数传递给指定的节点：

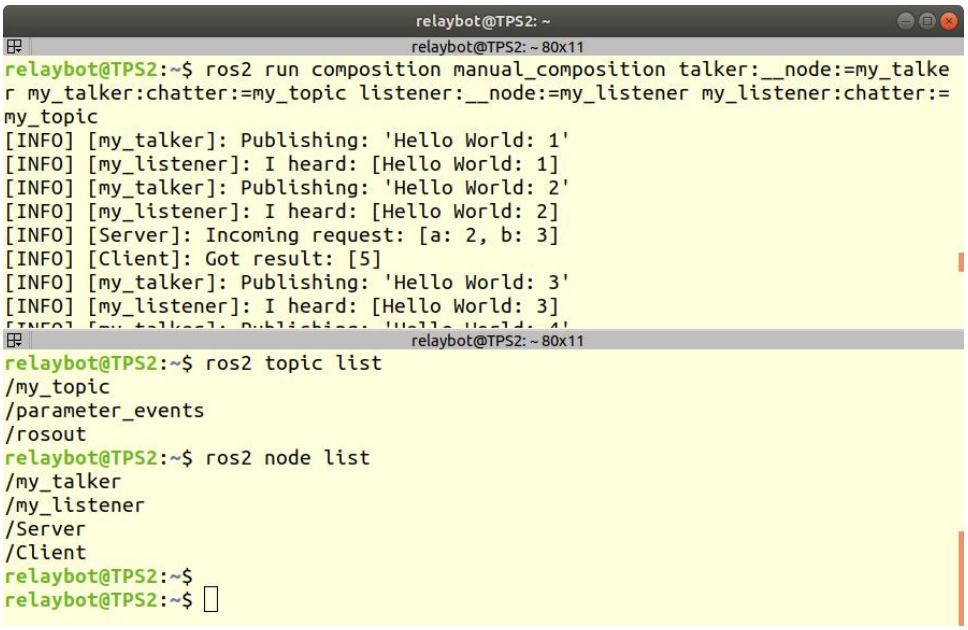
```
ros2 run composition manual_composition talker:__node:=my_talker listener:__node:=my_listener
```



The following example will both change the node name and remap a topic (node and namespace changes are always applied before topic remapping):

以下示例更改节点名称并重新映射主题（在主题重新映射之前始终应用节点和命名空间更改）：

```
ros2 run composition manual_composition talker:__node:=my_talker my_talker:chatter:=my_topic
listener:__node:=my_listener my_listener:chatter:=my_topic
```



主题名称为 my_topic 而不是默认值 chatter。

Logger configuration 日志记录器配置

See __log_level argument usage in the logging page.

请参考日志记录网页__log_level 中的参数用法。

Parameters 参数

Note: The behavior of parameters changed for Dashing and newer, so if you're using Crystal or older, see the section below for the old tutorial content.

注意：参数的行为已更改为 Dashing 和更新版本，因此如果使用的是 Crystal 或更旧版本，请参考下面的部分以获取旧教程内容。

Setting parameters from the command-line is currently only supported in the form of yaml files.

目前仅以 yaml 文件的形式支持从命令行设置参数。

See here for examples of the yaml file syntax. 有关 yaml 文件语法的示例，请参考此处。

As an example, save the following as demo_params.yaml:

例如，将以下内容另存为 demo_params.yaml：

```
parameter_blackboard:
    ros__parameters:
        some_int: 42
```

```
        a_string: "Hello world"
        some_lists:
            some_integers: [1, 2, 3, 4]
            some_doubles : [3.14, 2.718]
```

Then run the following:

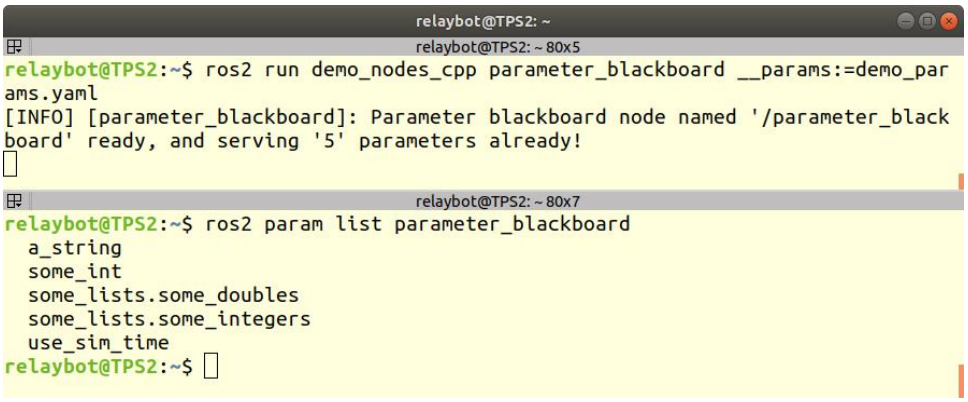
然后运行以下命令：

```
ros2 run demo_nodes_cpp parameter_blackboard __params:=demo_params.yaml
```

Other nodes will be able to retrieve the parameter values, e.g.:

其他节点将能够检索参数值，例如：

```
$ ros2 param list parameter_blackboard
a_string
some_int
some_lists.some_doubles
some_lists.some_integers
```



Crystal and Older Crystal 和之前发行版（Bouncy 等）

Parameters support for Python nodes was added in Crystal. In Bouncy only C++ nodes are supported.

Crystal 中添加了对 Python 节点的参数支持。在 Bouncy 中，仅支持 C ++ 节点。

Setting parameters from the command-line is currently supported in the form of yaml files.

目前，yaml 文件支持从命令行设置参数。

See here for examples of the yaml file syntax. 有关 yaml 文件语法的示例，请参考此处。

As an example, save the following as `demo_params.yaml`:

例如，将以下内容另存为 `demo_params.yaml`：

```
talker:
    ros__parameters:
```

```
        some_int: 42
        a_string: "Hello world"
        some_lists:
            some_integers: [1, 2, 3, 4]
            some_doubles : [3.14, 2.718]
```

Then run the following: 然后运行以下命令：

```
ros2 run demo_nodes_cpp talker __params:=demo_params.yaml
```

Other nodes will be able to retrieve the parameter values, e.g.: 其他节点将能够检索参数值，例如：

```
$ ros2 param list talker
a_string
some_int
some_lists.some_doubles
some_lists.some_integers
```

Launching/monitoring multiple nodes with Launch

使用 Launch 启动/监听多个节点

ROS 2 launch system ROS 2 launch 启动系统

The launch system in ROS 2 is responsible for helping the user describe the configuration of their system and then execute it as described. The configuration of the system includes what programs to run, where to run them, what arguments to pass them, and ROS specific conventions which make it easy to reuse components throughout the system by giving them each different configurations. It is also responsible for monitoring the state of the processes launched, and reporting and/or reacting to changes in the state of those processes.
ROS 2 中的启动系统负责协助用户描述其系统的配置，然后按照描述执行。系统的配置包括运行哪些程序、运行它们的位置、传递它们的参数以及 ROS 特定约定，这些约定使得通过为每个不同的配置提供组件，可以轻松地在整个系统中重用组件。它还负责监控已启动的流程的状态，并报告和/或响应这些流程的状态变化。
The ROS 2 Bouncy release includes a framework in which launch files, written in Python, can start and stop different nodes as well as trigger and act on various events. The package providing this framework is `launch_ros`, which uses the non-ROS-specific `launch` framework underneath.
ROS 2 Bouncy 及之后版本包含一个框架，其中用 Python 编写的 launch 文件可以启动和停止不同的节点，以及触发和处理各种事件。提供此框架的 `launch_ros` 包使用下面的非 ROS 特定 `launch` 框架。

The design document (in review) details the goal of the design of ROS 2's launch system (not all functionality is currently available).
这个设计文件（综述）详细描述 ROS 2 的 launch 启动系统的设计目标（不是所有的功能是目前可用）。

Example of ROS 2 launch concepts

ROS 2 launch 启动概念的例子

The launch file in this example launches two nodes, one of which is a node with a managed lifecycle (a "lifecycle node"). Lifecycle nodes launched through `launch_ros` automatically emit events when they transition between states. The events can then be acted on through the launch framework, e.g. by emitting other events (such as requesting another state transition, which lifecycle nodes launched through `launch_ros` automatically have event handlers for) or triggering other actions (e.g. starting another node).
此示例中的启动文件启动两个节点，其中一个节点是具有托管生命周期的节点（"生命周期节点"）。生命周期节点通过 `launch_ros` 在状态之间转换时自动发出*事件*来启动。然后可以通过启动框架对事件进行操作，例如通过发出其

他事件（例如请求另一个状态转换，生命周期节点通过 launch_ros 自动启动事件处理程序）或触发其他操作（例如，启动另一个节点）。

In the aforementioned example, various transition requests are requested of the talker lifecycle node, and its transition events are reacted to by, for example, launching a listener node when the lifecycle talker reaches the appropriate state.

在前述示例中，请求 talker 生命周期节点的各种转换请求，并且通过例如 listener 在生命周期发布器达到适当状态时启动节点来响应其转换事件。

Usage 用法

While launch files can be written as standalone scripts, the typical usage in ROS is to have launch files invoked by ROS 2 tools.

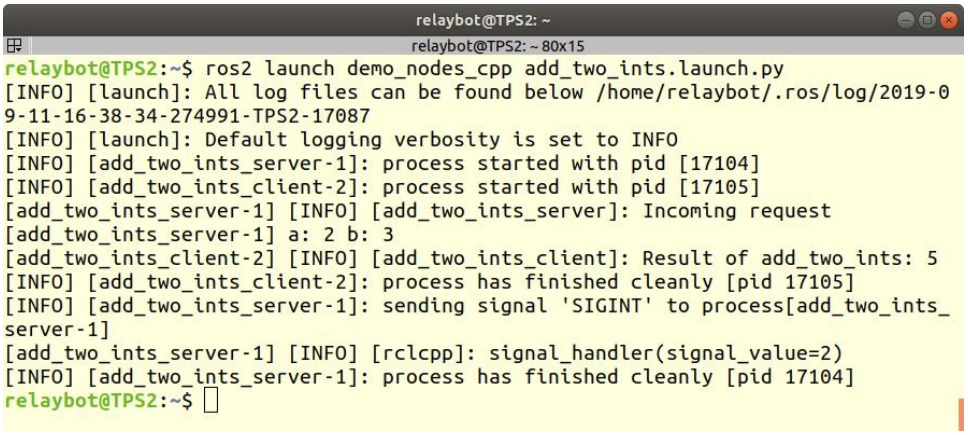
虽然启动文件可以作为独立脚本编写，但 ROS 中的典型用法是使用 ROS 2 工具调用启动文件。

For example, this launch file has been designed such that it can be invoked by ros2 launch:

例如，此启动文件的设计使其可以通过以下方式调用：ros2 launch

```
ros2 launch demo_nodes_cpp add_two_ints.launch.py
```

```
"""Launch a add_two_ints_server and a (synchronous) add_two_ints_client."""

import launch
import launch_ros.actions

def generate_launch_description():
    server = launch_ros.actions.Node(
        package='demo_nodes_cpp', node_executable='add_two_ints_server', output='screen')
    client = launch_ros.actions.Node(
        package='demo_nodes_cpp', node_executable='add_two_ints_client', output='screen')
    return launch.LaunchDescription([
        server,
        client,
        # TODO(wjwwood): replace this with a `required=True|False` option on ExecuteProcess().
        # Shutdown launch when client exits.
        launch.actions.RegisterEventHandler(
            event_handler=launch.event_handlers.OnProcessExit(
                target_action=client,
                on_exit=[launch.actions.EmitEvent(event=launch.events.Shutdown())],
            )),
])
```



Documentation 文档

The launch documentation provides more details on concepts that are also used in `launch_ros`.

启动文档提供了有关其中使用的概念的更多详细信息 `launch_ros`。

Additional documentation/examples of capabilities are forthcoming. See the source code in the meantime.

其他文档/功能示例即将发布。在此期间查看源代码。

Migrating launch files from ROS 1 to ROS 2

将启动文件从 ROS 1 迁移到 ROS2

This tutorial describes how to write XML launch files for an easy migration from ROS 1.
本教程描述了如何编写 XML 启动文件以便从 ROS 1 轻松迁移到 ROS 2。
（仅用于 ROS 1 复习）

Background 背景

A description of the ROS 2 launch system and its Python API can be found in Launch System tutorial.
可以在 Launch System 教程中找到有关 ROS 2 启动系统及其 Python API 的说明。

Migrating tags from ROS1 to ROS2

从 ROS1 迁移标签 ROS2

launch 启动

- Available in ROS 1.
  可用于 ROS 1。
- `launch` is the root element of any ROS 2 launch XML file.
  `launch` 是任何 ROS 2 启动 XML 文件的根元素。

node 节点

- Available in ROS1.
  ROS1 中提供。
- Launches a new node.
  启动一个新节点。

Differences from ROS 1: 与 ROS 1 的区别：

- `type` attribute is now `executable`.
  `type` 属性现在 `executable`。
- The following attributes aren't available: `machine`, `respawn`, `respawn_delay`, `clear_params`.
  下面的属性是不可用：`machine`，`respawn`，`respawn_delay`，`clear_params`。

Example 示例

```
<launch>
    <node pkg="demo_nodes_cpp" exec="talker"/>
    <node pkg="demo_nodes_cpp" exec="listener"/>
</launch>
```

```
from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            node_namespace= "ros2", package='demo_nodes_cpp', node_executable='talker', output='screen'),
        launch_ros.actions.Node(
            node_namespace= "ros2", package='demo_nodes_cpp', node_executable='listener', output='screen'),
    ])
```

param 参数

- Available in ROS1.
  ROS1 中提供。
- Used for passing a parameter to a node.
  用于将参数传递给节点。
  There's no global parameter concept in ROS 2. For that reason, it can only be used nested in a `node` tag.
  Some attributes aren't supported in ROS 2: `type`, `textfile`, `binfile`, `executable`, `command`.
  ROS 2 中没有全局参数概念。因此，它只能嵌套在 `node` 标签中。某些属性不支持 ROS 2：`type`、`textfile`、`binfile`、`executablecommand`。

Example 示例

```
<launch>
    <node pkg="demo_nodes_cpp" exec="parameter_event">
        <param name="foo" value="5"/>
    </node>
</launch>
```

Type inference rules 类型推断规则

Here are some examples of how to write parameters:
以下是如何编写参数的一些示例：

```
<node pkg="my_package" exec="my_executable" name="my_node">
    <!--A string parameter with value "1"-->
    <param name="a_string" value="'1'"/>
```

```
    <!--A integer parameter with value 1-->
    <param name="an_int" value="1"/>
    <!--A float parameter with value 1.0-->
    <param name="a_float" value="1.0"/>
    <!--A string parameter with value "asd"-->
    <param name="another_string" value="asd"/>
    <!--Another string parameter, with value "asd"-->
    <param name="string_with_same_value_as_above" value="'asd'"/>
    <!--Another string parameter, with value "'asd'"-->
    <param name="quoted_string" value="\'asd\'"/>
    <!--A list of strings, with value ["asd", "bsd", "csd"]-->
    <param name="list_of_strings" value="asd, bsd, csd" value-sep=", "/>
    <!--A list of ints, with value [1, 2, 3]-->
    <param name="list_of_ints" value="1,2,3" value-sep=","/>
    <!--Another list of strings, with value ["1", "2", "3"]-->
    <param name="another_list_of_strings" value="'1';'2';'3'" value-sep=";"/>
    <!--A list of strings using an strange separator, with value ["1", "2", "3"]-->
    <param name="strange_separator" value="'1'//'2'//'3'" value-sep="//"/>
</node>
```

Parameter grouping 参数分组

In ROS 2, `param` tags are allowed to be nested. For example:

在 ROS 2 中，`param` 标签允许嵌套。例如：

```
<node pkg="my_package" exec="my_executable" name="my_node" ns="/an_absoulute_ns">
    <param name="group1">
        <param name="group2">
            <param name="my_param" value="1"/>
        </param>
        <param name="another_param" value="2"/>
    </param>
</node>
```

That will create two parameters:

这将创建两个参数：

- A `group1.group2.my_param` of value `1`, hosted by node `/an_absolute_ns/my_node`.
  一个 `group1.group2.my_param` 的值 `1`，由节点`/an_absolute_ns/my_node` 托管。
- A `group1.another_param` of value `2` hosted by node `/an_absolute_ns/my_node`.
  一个 `group1.another_param` 的值 `2`，由节点`/an_absolute_ns/my_node` 托管。

It's also possible to use full parameter names:

也可以使用完整的参数名称：

```
<node pkg="my_package" exec="my_executable" name="my_node" ns="/an_absoulute_ns">
    <param name="group1.group2.my_param" value="1"/>
    <param name="group1.another_param" value="2"/>
</node>
```

rosparam

- Available in ROS1.
  ROS1 中提供。
- Loads parameters from a yaml file.
  从 yaml 文件加载参数。
- It has been replaced with a `from` atribute in `param` tags.
  它已从 `from` 属性替换 `param` 标签。

Example 示例

```
<node pkg="my_package" exec="my_executable" name="my_node" ns="/an_absoulute_ns">
    <param from="/path/to/file"/>
</node>
```

remap 重映射

- Available in ROS 1.
  可用于 ROS 1。
- Used to pass remapping rules to a node.
  用于将重映射规则传递给节点。
- It can only be used within `node` tags.
  它只能在 `node` 标签中使用。

Example 示例

```
<launch>
    <node pkg="demo_nodes_cpp" exec="talker">
        <remap from="chatter" to="my_topic"/>
    </node>
    <node pkg="demo_nodes_cpp" exec="listener">
        <remap from="chatter" to="my_topic"/>
    </node>
</launch>
```

include 包括

- Available in ROS 1.

可用于 ROS 1。

- Allows including another launch file.
  允许包含另一个启动文件。

Differences from ROS 1: 与 ROS 1 的区别：

- Available in ROS 1, included content was scoped. In ROS 2, it's not. Nest includes in `group` tags to scope them.
  在 ROS 1 中可用，包含的内容是作用域。但在 ROS 2 中，它不是。Nest 包含 `group` 标记以限定它们。
- `ns` attribute is not supported. See example of `push_ros_namespace` tag for a workaround.
  `ns` 属性不受支持。有关 `push_ros_namespace` 标记替换方法，请参考示例。
- `arg` tags nested in an `include` tag don't support conditionals (`if` or `unless`).
  `arg` 标记嵌套在 `include` 标记中，不支持条件（`if` 或 `unless`）。
- There is no support for nested `env` tags. `set_env` and `unset_env` can be used instead.
  不支持嵌套 `env` 标记。 `set_env` 并且 `unset_env` 可以替代使用。
- Both `clear_params` and `pass_all_args` attributes aren't supported.
  这两个 `clear_params` 和 `pass_all_args` 属性不被支持。

Examples 示例

See Replacing an include tag. 请参考替换包含标记。

arg

- Available in ROS 1.
  可用于 ROS 1。
- `arg` is used for declaring a launch argument, or to pass an argument when using `include` tags.
  `arg` 用于声明启动参数，或在使用 `include` 标记时传递参数。

Differences from ROS 1: 与 ROS 1 的区别：

- `value` attribute is not allowed. Use `let` tag for this.
  `value` 属性是不允许的。使用 `let` 标签。
- `doc` is now `description`.
  `doc` 现在是 `description`。
- When nested within an `include` tag, `if` and `unless` attributes aren't allowed.
  当嵌套在一个 `include` 标签的范围内，`if` 和 `unless` 属性是不允许的。

Example 示例

```
<launch>
  <arg name="topic_name" default="chatter"/>
  <node pkg="demo_nodes_cpp" exec="talker">
    <remap from="chatter" to="$(var topic_name)"/>
  </node>
  <node pkg="demo_nodes_cpp" exec="listener">
```

```
    <remap from="chatter" to="$(var topic_name)"/>
  </node>
</launch>
```

Passing an argument via the command line 通过命令行传递参数

See ROS 2 launch tutorial. 请参考 ROS 2 launch 启动教程。

env

- Available in ROS 1.
  可用于 ROS 1。
- Sets an environment variable.
  设置环境变量。
- **It has been replaced with `env`, `set_env` and `unset_env`:**
  它已被替换为 `env`，`set_env` 和 `unset_env`：
  - `env` can only be used nested in a `node` or `executable` tag. `if` and `unless` tags aren't supported.
    `env` 只能嵌套在一个 `node` 或 `executable` 标签中使用。 不支持 `if` 和 `unless` 标签。
  - `set_env` can be nested within the root tag `launch` or in `group` tags. It accepts the same attributes as `env`, and also `if` and `unless` tags.
    `set_env` 可以嵌套在根标签 `launch` 或 `group` 标签中。它接受与 `env`，`if` 以及 `unless` 标签相同的属性。
  - `unset_env` unsets an environment variable. It accepts a `name` attribute and conditionals.
    `unset_env` 取消设置环境变量。它接受 `name` 属性和条件。

Example 示例

```
<launch>
  <set_env name="MY_ENV_VAR" value="MY_VALUE" if="CONDITION_A"/>
  <set_env name="ANOTHER_ENV_VAR" value="ANOTHER_VALUE" unless="CONDITION_B"/>
  <set_env name="SOME_ENV_VAR" value="SOME_VALUE"/>
  <node pkg="MY_PACKAGE" exec="MY_EXECUTABLE" name="MY_NODE">
    <env name="NODE_ENV_VAR" value="SOME_VALUE"/>
  </node>
  <unset_env name="MY_ENV_VAR" if="CONDITION_A"/>
  <node pkg="ANOTHER_PACKAGE" exec="ANOTHER_EXECUTABLE" name="ANOTHER_NODE"/>
  <unset_env name="ANOTHER_ENV_VAR" unless="CONDITION_B"/>
  <unset_env name="SOME_ENV_VAR"/>
</launch>
```

group

- Available in ROS 1.
  可用于 ROS 1。
- Allows limiting the scope of launch configurations. Usually used together with `let`, `include` and `push_ros_namespace` tags.

允许限制启动配置的范围。通常与 let，include 和 push_ros_namespace 标签一起使用。

Differences from ROS 1: 与 ROS 1 的区别：

- There is no ns attribute. See the new push_ros_namespace tag as a workaround.
  没有 ns 属性。将新 push_ros_namespace 标记视为替代方法。
- clear_params attribute isn't available.
  clear_params 属性不可用。
- It doesn't accept remap nor param tags as children.
  它不接受 remap 和 param 标签作子标签。

Example 示例

launch-prefix configuration affects both executable and node tags' actions. This example will use time as a prefix if use_time_prefix_in_talker argument is 1, only for the talker.
launch-prefix 配置会影响标记 executable 和 node 标记的操作。此示例将 time 用作 use_time_prefix_in_talker 参数的前缀 1，仅用于发布器。

```
<launch>
  <arg name="use_time_prefix_in_talker" default="0"/>
  <group>
    <let name="launch-prefix" value="time" if="$(var use_time_prefix_in_talker)"/>
    <node pkg="demo_nodes_cpp" exec="talker"/>
  </group>
  <node pkg="demo_nodes_cpp" exec="listener"/>
</launch>
```

machine

It is not supported at the moment. 暂不支持。

test

It is not supported at the moment. 暂不支持。

New tags in ROS 2 ROS 2 中的新标签

set_env and unset_env set_env 和 unset_env

See env tag decription. 请参考 env 标签说明。

push_ros_namespace

include and group tags don't accept an ns attribute. This action can be used as a workaround:
include 和 group 标签不接受 ns 属性。此操作可用作解决方法：

```
<!-Other tags-->
<group>
  <push_ros_namespace namespace="my_ns"/>
```

```
    <!--Nodes here are namespaced with "my_ns".-->
    <!--If there is an include action here, its nodes will also be namespaced.-->
    <push_ros_namespace namespace="another_ns"/>
    <!--Nodes here are namespaced with "another_ns/my_ns".-->
    <push_ros_namespace namespace="/absolute_ns"/>
    <!--Nodes here are namespaced with "/absolute_ns".-->
    <!--The following node receives an absolute namespace, so it will ignore the others previously pushed.-->
    <!--The full path of the node will be /asd/my_node.-->
    <node pkg="my_pkg" exec="my_executable" name="my_node" ns="/asd"/>
</group><!--Nodes outside the group action won't be namespaced.--><!-Other tags-->
```

let

It's a replacement of `arg` tag with a value attribute. 它是使用 value 属性 `arg` 标记的替换。

```
<let var="foo" value="asd"/>
```

executable

It allows running any executable. 它允许运行任何可执行文件

Example 示例

```
<executable cmd="ls -las" cwd="/var/log" name="my_exec" launch-prefix="something" output="screen" shell="true">
    <env name="LD_LIBRARY" value="/lib/some.so"/>
</executable>
```

Replacing an include tag 替换包含标记

To have exactly the same behavior as Available in ROS 1, `include` tags must be nested in a `group` tag.
要与 ROS 1 中的“可用”具有完全相同的行为，`include` 标记必须嵌套在 `group` 标记中。

```
<group>
    <include file="another_launch_file"/>
</group>
```

To replace the `ns` attribute, `push_ros_namespace` action must be used:
要替换 `ns` 属性，必须使用 `push_ros_namespace` 操作：

```
<group>
    <push_ros_namespace namespace="my_ns"/>
    <include file="another_launch_file"/>
</group>
```

Substitutions 替换

Documentation about ROS 1's substitutions can be found in roslaunch XML wiki. Substitutions syntax hasn't changed, i.e. it still follows the `$(substitution-name arg1 arg2 ...)` pattern. There are, however, some changes w.r.t. ROS 1:

有关 ROS 1 替换的文档可以在 roslaunch XML wiki 中找到。替换语法没有改变，即它仍然遵循 `$(substitution-name arg1 arg2 ...)`模式。但是，ROS 1 有一些变化：

- `env` and `optenv` tags have been replaced by the `env` tag. `$(env <NAME>)` will fail if the environment variable doesn't exist. `$(env <NAME> '')` does the same as ROS 1's `$(optenv <NAME>)`. `$(env <NAME> <DEFAULT>)` does the same as ROS 1's `$(env <NAME> <DEFAULT>)` or `$(optenv <NAME> <DEFAULT>)`.
  `env` 和 `optenv` 标签已被标签 `env` 取代。 如果环境变量`$(env <NAME>)`不存在，则会失败。 `$(env <NAME> '')` 与 ROS 1`$(optenv <NAME>)`相同。 `$(env <NAME> <DEFAULT>)`与 ROS 1`$(env <NAME> <DEFAULT>)`或 `$(optenv <NAME> <DEFAULT>)`相同。
- `find` has been replaced with `find-pkg`.
  `find` 已被 `find-pkg` 取代。
- There is a new `exec-in-pkg` substitution. e.g.: `$(exec-in-pkg <package_name> <exec_name>)`.
  有一个新的 `exec-in-pkg` 替代品。例如：`$(exec-in-pkg <package_name> <exec_name>)`。
- There is a new `find-exec` substitution.
  有一个新的 `find-exec` 替代品。
- `arg` has been replaced with `var`. It looks at configurations defined either with `arg` or `let` tag.
  `arg` 已被 `var` 取代。它查看之前使用 `arg` 或 `let` 标记定义的配置。
- `eval` and `dirname` substitutions haven't changed.
  `eval` 和 `dirname` 替换没有改变。
- `anon` substitution is not supported.
  `anon` 不支持替换。

Type inference rules 类型推断规则

The rules that were shown in `Type inference rules` subsection of `param` tag applies to any attribute. For example:

标签 `param` 子部分 `Type inference rules` 中显示的规则适用于任何属性。例如：

```
<!--Setting a string value to an attribute expecting an int will raise an error.-->
<tag1 attr-expecting-an-int="'1'"/>
<!--Correct version.-->
<tag1 attr-expecting-an-int="1"/>
<!--Setting an integer in an attribute expecting a string will raise an error.-->
<tag2 attr-expecting-a-str="1"/>
<!--Correct version.-->
<tag2 attr-expecting-a-str="'1'"/>
```

```
<!--Setting a list of strings in an attribute expecting a string will raise an error.-->
<tag3 attr-expecting-a-str="asd, bsd" str-attr-sep=", "/>
<!--Correct version.-->
<tag3 attr-expecting-a-str="don't use a separator"/>
```

Some attributes accept more than a single type, for example `value` attribute of `param` tag. It's usual that parameters that are of type `int` (or `float`) also accept an `str`, that will be later substituted and tried to convert to an `int` (or `float`) by the action.

某些属性接受多个类型，例如 `param` 标记的 `value` 属性。通常，`int` 类型（或 `float` 类型）的参数也接受 `str`，稍后将替换并尝试通过操作转换为 `int`（或 `float`）。

Working with multiple ROS 2 middleware implementations

使用多个 ROS 2 中间件实现

This page explains the default RMW implementation and how to specify an alternative.
此网页解释了默认的 RMW 实现以及如何指定备选方案。

Pre-requisites 预备基础

You should have already read the DDS and ROS middleware implementations page.
请先阅读 DDS 和 ROS 中间件实现网页。

Multiple RMW implementations 多个 RMW 实现

The current ROS 2 binary releases have built-in support for several RMW implementations out of the box (Fast RTPS, RTI Connext Pro, and ADLink OpenSplice at the time of writing), but only Fast RTPS (the default) works without any additional installation steps, because it is the only one we distribute with our binary packages.
当前的 ROS 2 二进制版本内置的几个 RMW 实现版本支持开箱即用（在撰写本文时，如，快速 RTPS、RTI Connext Pro 和 ADLink OpenSplice），但只有 Fast RTPS（默认）无需额外安装即可工作步骤，因为它是使用二进制包分发的唯一一个。
Others like OpenSplice or Connext can be enabled by installing additional packages, but without having to rebuild anything or replace any existing packages.
其他像 OpenSplice 或 Connext 可以通过安装其他软件包来启用，但无需重建任何内容或替换任何现有软件包。
Also, a ROS 2 workspace that has been built from source may build and install multiple RMW implementations simultaneously. While the core ROS 2 code is being compiled, any RMW implementation that is found will be built if the relevant DDS/RTPS implementation has been installed properly and the relevant environment variables have been configured. For example, if the code for the RMW package for RTI Connext is in the workspace, it will be built if an installation of RTI's Connext Pro can also be found. For many cases you will find that nodes using different RMW implementations are able to communicate, however this is not true under all circumstances. A list of supported inter-vendor communication configurations is forthcoming.
此外，从源编译的 ROS 2 工作空间可以同时编译和安装多个 RMW 实现。在编译核心 ROS 2 代码时，如果已正确安装相关的 DDS / RTPS 实现并且已配置相关的环境变量，则将编译找到的任何 RMW 实现。例如，如果 RTI Connext 的 RMW 包的代码在工作区中，那么如果也可以找到 RTI 的 Connext Pro 安装，它将被编译。对于许多情况，会发现使用不同 RMW 实现的节点能够进行通信，但并非所有情况下都是这样。支持的厂商间通信配置列表即将发布。

Default RMW implementation 默认 RMW 实现

If a ROS 2 workspace has multiple RMW implementations, the default RMW implementation is currently selected as Fast RTPS if it's available. If the Fast RTPS RMW implementation is not installed, the RMW implementation with the first RMW implementation identifier in alphabetical order will be used. The implementation identifier is the name of the ROS package that provides the RMW implementation, e.g. `rmw_fastrtps_cpp`. For example, if both `rmw_opensplice_cpp` and `rmw_connext_cpp` ROS packages are installed, `rmw_connext_cpp` would be the default. If `rmw_fastrtps_cpp` is ever installed, it would be the default. See below for how to specify which RMW implementation is to be used when running the ROS 2 examples.

如果 ROS 2 工作空间具有多个 RMW 实现，则默认 RMW 实现当前被选为快速 RTPS（如果可用）。如果未安装快速 RTPS RMW 实现，将使用具有按字母顺序排列的第一个 RMW 实现标识符的 RMW 实现。实现标识符是提供 RMW 实现的 ROS 包的名称，例如 `rmw_fastrtps_cpp`。例如，如果同时安装了两个 `rmw_opensplice_cpp` 和 `rmw_connext_cpp`ROS 包，则默认为 `rmw_connext_cpp`。如果 `rmw_fastrtps_cpp` 曾经安装过，那将是默认设置。请参考下文，了解如何在运行 ROS 2 示例时指定要使用的 RMW 实现。

Specifying RMW implementations 指定RMW实现

To have multiple RMW implementations available for use you must have installed our binaries and any additional dependencies for specific RMW implementations, or built ROS 2 from source with multiple RMW implementations in the workspace (they are included by default) and their dependencies are met (for example see the Linux install instructions).

要使多个 RMW 实现可供使用，必须已安装二进制文件以及特定 RMW 实现的任何其他依赖项，或者在工作区中使用多个 RMW 实现从源编译 ROS 2（默认包含它们）并且满足它们的依赖性（对于示例请参考 Linux 安装说明）。

Starting in Beta 2 and above both C++ and Python nodes support an environment variable `RMW_IMPLEMENTATION`. To choose a different RMW implemenation you can set the environment variable `RMW_IMPLEMENTATION` to a specific implementation identifier.

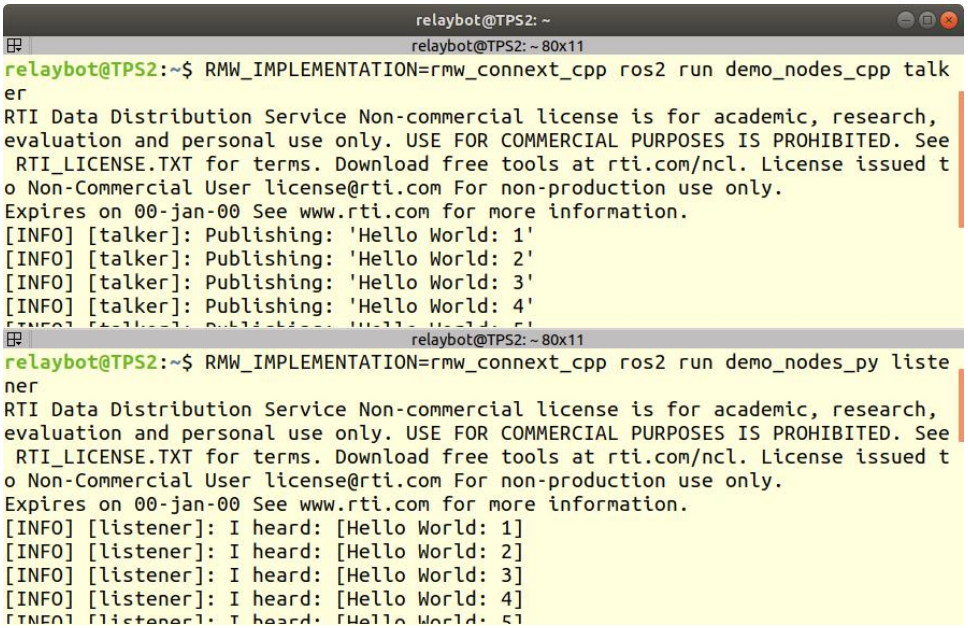
从 Beta 2 及更高版本开始，C ++和 Python 节点都支持环境变量 `RMW_IMPLEMENTATION`。要选择不同的 RMW 实现，可以将环境变量 `RMW_IMPLEMENTATION` 设置为特定的实现标识符。

To run the talker demo using the C++ and listener using python with the RMW implementation for connext:

使用 C ++和接收器使用 python 和连接器的 RMW 实现来运行发布器演示：

Bash

```
RMW_IMPLEMENTATION=rmw_connext_cpp ros2 run demo_nodes_cpp talker

# Run in another terminal
RMW_IMPLEMENTATION=rmw_connext_cpp ros2 run demo_nodes_py listener
```



Windows cmd.exe

```
set RMW_IMPLEMENTATION=rmw_connext_cpp
ros2 run demo_nodes_cpp talker

REM run in another terminal
set RMW_IMPLEMENTATION=rmw_connext_cpp
ros2 run demo_nodes_py listener
```

Adding RMW implementations to your workspace

将 RMW 实现添加到工作区

Suppose that you have built your ROS 2 workspace with only Fast RTPS installed and therefore only the Fast RTPS RMW implementation built. The last time your workspace was built, any other RMW implementation packages, `rmw_connext_cpp` for example, were probably unable to find installations of the relevant DDS implementations. If you then install an additional DDS implementation, Connext for example, you will need to re-trigger the check for a Connext installation that occurs when the Connext RMW implementation is being built. You can do this by specifying the `--cmake-force-configure` flag on your next workspace build, and you should see that the RMW implementation package then gets built for the newly installed DDS implementation.
假设已经编译了仅安装了快速 RTPS 的 ROS 2 工作区，因此仅编译了快速 RTPS RMW 实现。上次编译工作区时，任何其他 RMW 实现包，如 `rmw_connext_cpp` 都可能无法找到相关 DDS 实现的安装。例如，如果再安装其他 DDS 实现版本（例如 Connext），则需要重新触发对编译 Connext RMW 实现时发生的 Connext 安装的检查。可以通过在下一个工作区编译中指定`--cmake-force-configure` 标志来执行此操作，应该看到 RMW 实现包然后为新安装的 DDS 实现编译。

It is possible to run into a problem when “rebuilding” the workspace with an additional RMW implementation using the `--cmake-force-configure` option where the build complains about the default RMW implementation changing. To resolve this, you can either set the default implementation to what is was before with the `RMW_IMPLEMENTATION` CMake argument or you can delete the build folder for packages that complain and continue the build with `--start-with <package name>`.

当使用`--cmake-force-configure`编译提示默认 RMW 实现更改的选项使用其他 RMW 实现“重建”工作空间时，可能会遇到问题。要解决此问题，可以将默认实现设置为之前的`RMW_IMPLEMENTATION`的 CMake 参数，也可以删除提示并使用`--start-with <package name>`继续编译。

Troubleshooting 故障排除

Ensuring use of a particular RMW implementation

确保使用特定的 RMW 实现

ROS 2 Ardent and later ROS 2 Ardent 及以后

If the `RMW_IMPLEMENTATION` environment variable is set to an RMW implementation for which support is not installed, you will see an error message similar to the following if you have only one implementation installed:

如果`RMW_IMPLEMENTATION`环境变量设置为未安装支持的 RMW 实现，如果只安装了一个实现，则会看到类似于以下内容的错误消息：

```
Expected RMW implementation identifier of 'rmw_connext_cpp' but instead found 'rmw_fastrtps_cpp', exiting with 102.
```

If you have support for multiple RMW implementations installed and you request use of one that is not installed, you will see something similar to:

如果支持安装多个 RMW 实现，并且请求使用未安装的实现，则将看到类似于以下内容的内容：

```
Error getting RMW implementation identifier / RMW implementation not installed (expected identifier of 'rmw_connext_cpp'), exiting with 1.
```

If this occurs, double check that your ROS 2 installation includes support for the RMW implementation that you have specified in the `RMW_IMPLEMENTATION` environment variable.

如果发生这种情况，请仔细检查 ROS 2 安装是否包含对在`RMW_IMPLEMENTATION`环境变量中指定的 RMW 实现的支持。

Composing multiple nodes in a single process

在单个进程中组合多个节点

ROS 1 - Nodes vs. Nodelets ROS 1 -节点与 Nodelets

In ROS 1 you can write your code either as a ROS node or as a ROS nodelet. ROS 1 nodes are compiled into executables. ROS 1 nodelets on the other hand are compiled into a shared library which is then loaded at runtime by a container process.

在 ROS 1，无论是作为一个 ROS 节点或作为 ROS nodelet 都可以通过代码实现。ROS 1 节点被编译成可执行文件。另一方面，ROS 1 nodelets 被编译成共享库，然后由容器进程在运行时加载。

ROS 2 - Unified API ROS 2 - 统一 API

In ROS 2 the recommended way of writing your code is similar to a nodelet - we call it a `Component`. This makes is easy to add common concepts to existing code, like a life cycle. The biggest drawback of different APIs is avoided in ROS 2 since both approaches use the same API in ROS 2.

在 ROS 2 中，编写代码的推荐方法类似于 nodelet - 我们称之为 `Component`。这样可以很容易地将常见概念添加到现有代码中，例如生命周期。ROS 2 中避免了不同 API 的最大缺点，因为这两种方法在 ROS 2 中使用相同的 API。

Note 注意

It is still possible to use the node-like style of “writing your own main” but for the common case it is not recommended.

仍然可以使用类似节点的“编写自定义 main”样式，但对于常见情况，不推荐使用。

By making the process layout a deploy-time decision the user can choose between:

通过使流程布局成为部署时决策，用户可以选择：

- running multiple nodes in separate processes with the benefits of process/fault isolation as well as easier debugging of individual nodes and
  在单独的进程中运行多个节点，具有进程/故障隔离的好处，以及更容易调试单个节点和
- running multiple nodes in a single process with the lower overhead and optionally more efficient communication (see Intra Process Communication).
  在单个进程中运行多个节点，具有较低的开销和可选的更高效的通信（请参考进程内通信）。

Additionally `ros2 launch` can be used to automate these actions through specialized launch actions.

此外，`ros2 launch` 还可用于通过专门的启动操作自动执行这些操作。

Writing a Component 编写组件

Since a component is only built into a shared library it doesn't have a `main` function (see Talker source code). A component is commonly a subclass of `rclcpp::Node`. Since it is not in control of the thread it shouldn't perform any long running or blocking tasks in its constructor. Instead it can use timers to get periodic notification. Additionally it can create publishers, subscribers, servers, and clients.

由于组件仅编译在共享库中，因此它没有 `main` 函数（请参考 Talker 源代码）。组件通常是子类 `rclcpp::Node`。由于它不受线程控制，因此不应在其构造函数中执行任何长时间运行或阻塞任务。相反，它可以使用计时器来获得定期通知。此外，它还可以创建发布器，订阅器，服务器和客户端。

An important aspect of making such a class a component is that the class registers itself using macros from the package `rclcpp_components` (see the last line in the source code). This makes the component discoverable when its library is being loaded into a running process - it acts as kind of an entry point.

将这样的类作为组件的一个重要方面是类使用 `rclcpp_components` 包中的宏来注册自己（参考源代码中的最后一行）。这使得组件在将其库加载到正在运行的进程时可被发现 - 它充当入口点。

Additionally, once a component is created, it must be registered with the index to be discoverable by the tooling.

此外，一旦创建了组件，就必须将其注册到工具可以发现的索引。

```
add_library(talker_component SHARED
  src/talker_component.cpp)
rclcpp_components_register_nodes(talker_component "composition::Talker")
# To register multiple components in the same shared library, use multiple calls#
rclcpp_components_register_nodes(talker_component "composition::Talker2")
```

Note 注意

In order for the component_container to be able to find desired components, it must be executed or launched from a shell that has sourced the corresponding workspace.

为了使 component_container 能够找到所需的组件，必须从已获取相应工作空间的 shell 执行或启动它。

Using Components 使用组件

The composition package contains a couple of different approaches on how to use components. The three most common ones are:

该组合包中包含了一些关于如何使用的组件不同的方法。最常见的三种是：

- Start a (generic container process) and call the ROS service load_node offered by the container. The ROS service will then load the component specified by the passed package name and library name and start executing it within the running process. Instead of calling the ROS service programmatically you can also use a command line tool to invoke the ROS service with the passed command line arguments

启动（通用容器进程）并调用容器提供的 ROS 服务 load_node。然后，ROS 服务将加载由传递的包名和库名指定的组件，并在运行的进程中开始执行它。您不必以编程方式调用 ROS 服务，也可以使用命令行工具使用传递的命令行参数调用 ROS 服务

- Create a custom executable containing multiple nodes which are known at compile time. This approach requires that each component has a header file (which is not strictly needed for the first case).
  创建包含在编译时已知的多个节点的自定义可执行文件。这种方法要求每个组件都有一个头文件（第一种情况并不严格需要）。
- Create a launch file and use `ros2 launch` to create a container process with multiple components loaded.
  创建启动文件并用 `ros2 launch` 创建加载了多个组件的容器进程。

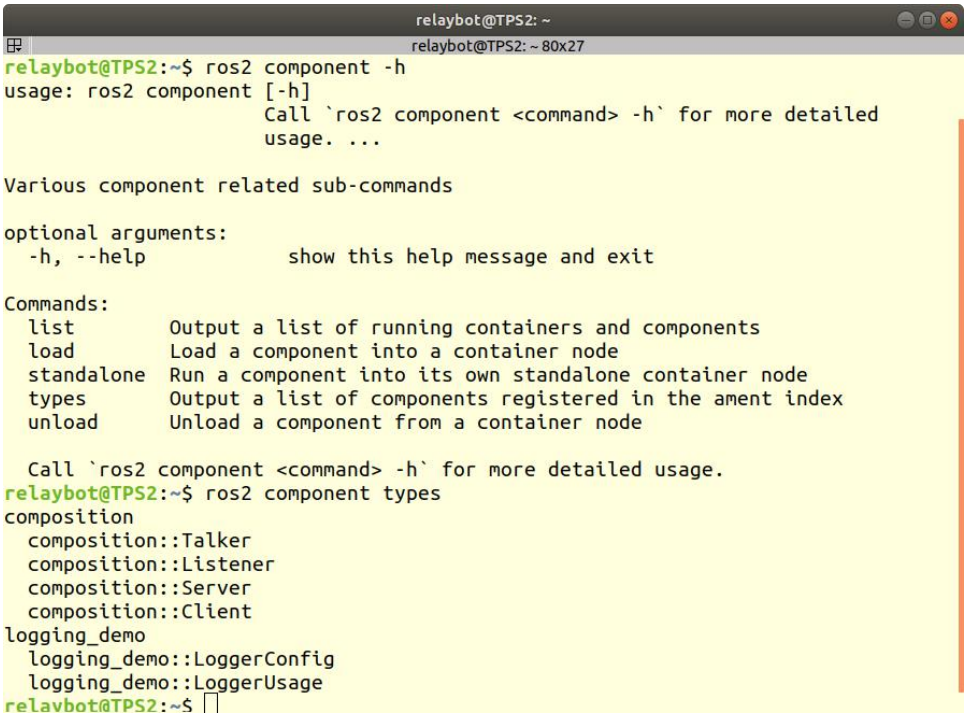
Run the demos 运行演示

The demos use executables from rclcpp_components, ros2component, and composition packages, and can be run with the following commands.
演示使用来自 rclcpp_components，ros2component 和 composition 包的可执行文件，并且可以使用以下命令运行。

Discover available components 发现可用的组件

To see what components are registered and available in the workspace, execute the following in a shell:
要查看工作区中注册和可用的组件，请在 shell 中执行以下操作：

```
$ ros2 component types
composition
  composition::Talker
  composition::Listener
  composition::Server
  composition::Client
```



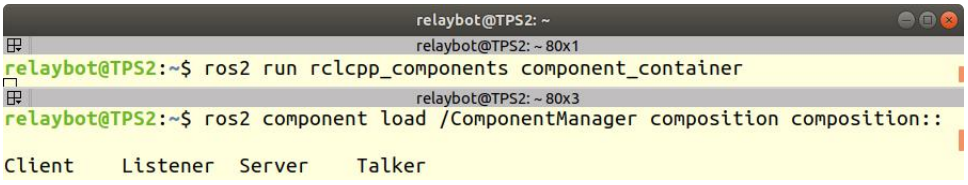
Run-time composition using ROS services (1.) with a publisher and subscriber

使用 ROS 服务（1.）与发布器和订阅器的运行时组合

In the first shell, start the component container:

在第一个 shell 中，启动组件容器：

```
ros2 run rclcpp_components component_container
```



Verify that the container is running via `ros2` command line tools:

通过 `ros2` 命令行工具验证容器是否正在运行：

```
$ ros2 component list
/ComponentManager
```

In the second shell (see talker source code). The command will return the unique ID of the loaded component as well as the node name.

在第二个 shell 中（参考 talker 源代码）。该命令将返回已加载组件的唯一 ID 以及节点名称。

```
$ ros2 component load /ComponentManager composition composition::Talker
```

```
Loaded component 1 into '/ComponentManager' container node as '/talker'
```

Now the first shell should show a message that the component was loaded as well as repeated message for publishing a message.

现在，第一个 shell 应显示已加载组件的消息以及用于发布消息的重复消息。

Another command in the second shell (see listener source code):

第二个 shell 中的另一个命令（参考监听器源代码）：

```
$ ros2 component load /ComponentManager composition composition::Listener
Loaded component 2 into '/ComponentManager' container node as '/listener'
```

The `ros2` command line utility can now be used to inspect the state of the container:

该 `ros2` 命令行实用程序现在可以用于检查容器的状态：

```
$ ros2 component list
/ComponentManager
  1  /talker
  2  /listener
```

Now the first shell should show repeated output for each received message.

现在，第一个 shell 应该显示每个收到的消息的重复输出。

Run-time composition using ROS services (1.) with a server and client

使用 ROS 服务（1.）与服务器和客户端的运行时组合

The example with a server and a client is very similar. 服务器和客户端的示例非常相似。

In the first shell: 在第一个 shell 中：

```
ros2 run rclcpp_components component_container
```

In the second shell (see server and client source code): 在第二个 shell 中（请参考服务器和客户端源代码）：

```
ros2 component load /ComponentManager composition composition::Server
ros2 component load /ComponentManager composition composition::Client
```

In this case the client sends a request to the server, the server processes the request and replies with a response, and the client prints the received response.

在这种情况下，客户端向服务器发送请求，服务器处理请求并回复响应，客户端打印接收到的响应。

Compile-time composition using ROS services (2.)

使用 ROS 服务编译时组成（2.）

This demos shows that the same shared libraries can be reused to compile a single executable running multiple components. The executable contains all four components from above: talker and listener as well as server and client.

此演示显示可以重用相同的共享库来编译运行多个组件的单个可执行文件。可执行文件包含上面的所有四个组件：talker 和 listener 以及服务器和客户端。

In the shell call (see source code):

在 shell 调用中（参考源代码）：

```
ros2 run composition manual_composition
```

This should show repeated messages from both pairs, the talker and the listener as well as the server and the client.

这应该显示来自两个对，发布器和听众以及服务器和客户端的重复消息。

Note 注意

Manually-composed components will not be reflected in the `ros2 component list` command line tool output. 手动编写的组件不会反映在 `ros2 component list` 命令行工具输出中。

Run-time composition using dlopen 使用 dlopen 运行时组合

This demo presents an alternative to 1. by creating a generic container process and explicitly passing the libraries to load without using ROS interfaces. The process will open each library and create one instance of each “rclcpp::Node” class in the library source code).

此演示提供了 1 的替代方法，通过创建通用容器进程并显式传递库而不使用 ROS 接口。该过程将打开每个库并在库源代码中创建每个“rclcpp :: Node”类的一个实例。

Linux In the shell call: **Linux** 在 shell 中调用：

```
ros2 run composition dlopen_composition `ros2 pkg prefix composition`/lib/libtalker_component.so `ros2 pkg prefix composition`/lib/liblistener_component.so
```

OSX In the shell call:

OSX 在 shell 中调用：

```
ros2 run composition dlopen_composition `ros2 pkg prefix composition`/lib/libtalker_component.dylib `ros2 pkg prefix composition`/lib/liblistener_component.dylib
```

Windows In cmd.exe call

Windows 在 cmd.exe 中调用

```
ros2 pkg prefix composition
```

to get the path to where composition is installed. Then call

获取安装组合的路径。然后调用：

```
ros2 run composition dlopen_composition <path_to_composition_install>\bin\talker_component.dll <path_to_composition_install>\bin\listener_component.dll
```

Now the shell should show repeated output for each sent and received message.

现在 shell 应该为每个发送和接收的消息显示重复输出。

Note 注意

dlopen-composed components will not be reflected in the `ros2 component list` command line tool output.

dlopen 组成的组件不会反映在命令行 `ros2 component list` 工具输出中。

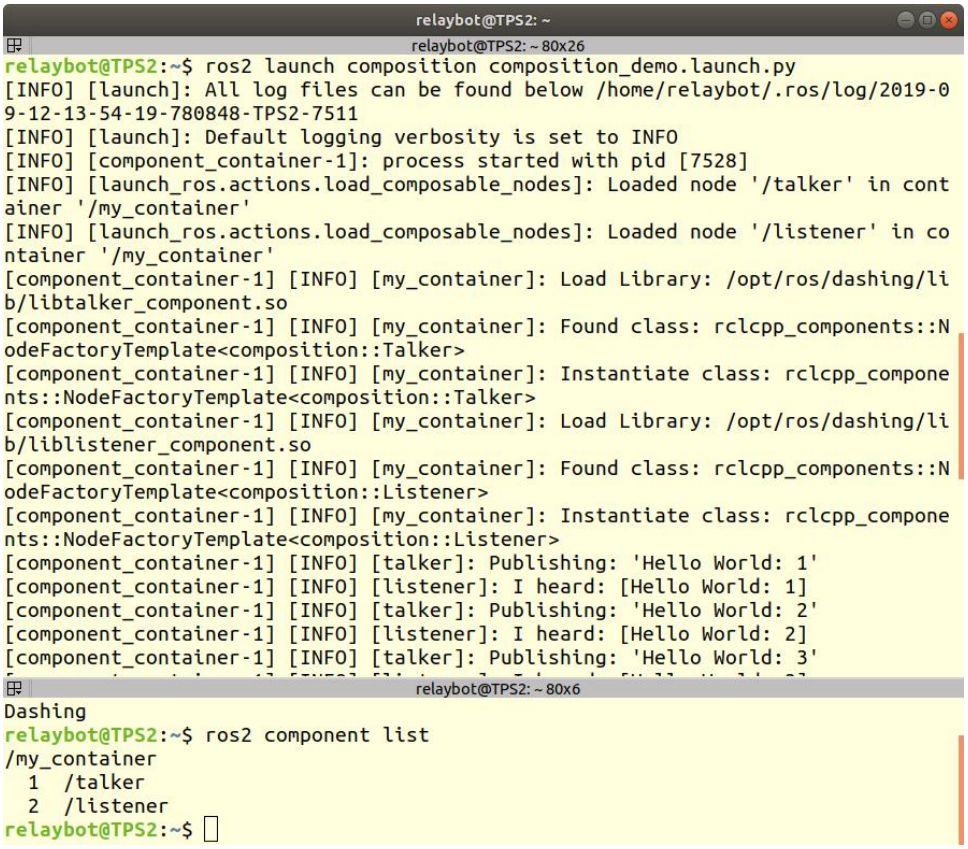
Composition using launch actions

使用 launch 启动行动的组合

While the command line tools are useful for debugging and diagnosing component configurations, it is frequently more convenient to start a set of components at the same time. To automate this action, we can use the functionality in `ros2 launch`.

虽然命令行工具对于调试和诊断组件配置很有用，但同时启动一组组件通常更方便。要自动执行此操作，可以使用 `ros2 launch` 功能。

```
ros2 launch composition composition_demo.launch.py
```



Advanced Topics 高级主题

Now that we have seen the basic operation of components, we can discuss a few more advanced topics.
现在已经看到了组件的基本操作，我们可以讨论一些更高级的主题。

Unloading components 卸载组件

In the first shell, start the component container: 在第一个 shell 中，启动组件容器：

```
ros2 run rclcpp_components component_container
```

Verify that the container is running via `ros2` command line tools: 通过 `ros2` 命令行工具验证容器是否正在运行：

```
$ ros2 component list
/ComponentManager
```

In the second shell (see talker source code). The command will return the unique ID of the loaded component as well as the node name.
在第二个 shell 中（参考 talker 源代码）。该命令将返回已加载组件的唯一 ID 以及节点名称。

```
$ ros2 component load /ComponentManager composition composition::Talker
Loaded component 1 into '/ComponentManager' container node as '/talker'
$ ros2 component load /ComponentManager composition composition::Listener
```

```
Loaded component 2 into '/ComponentManager' container node as '/listener'
```

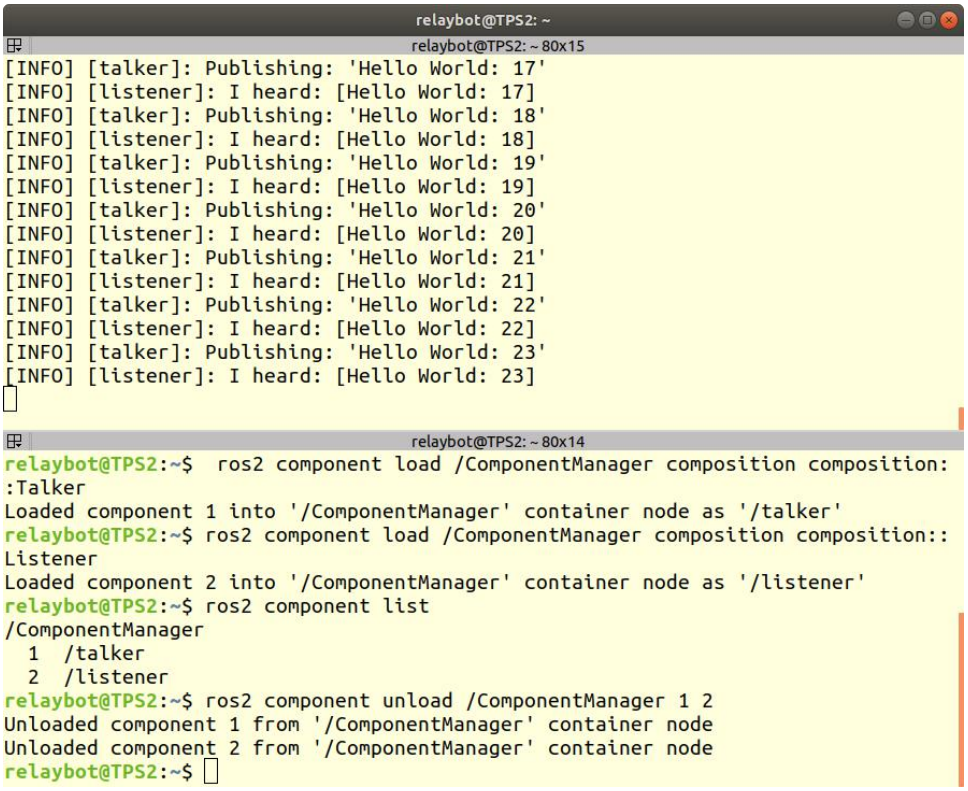
Use the unique ID to unload the node from the component container.

使用唯一 ID 从组件容器中卸载节点。

```
$ ros2 component unload /ComponentManager 1 2
Unloaded component 1 from '/ComponentManager' container
Unloaded component 2 from '/ComponentManager' container
```

In the first shell, verify that the repeated messages from talker and listener have stopped.

在第一个 shell 中，验证来自 talker 和 listener 的重复消息是否已停止。



Remapping container name and namespace

重新映射容器名称和命名空间

The component manager name and namespace can be remapped via standard command line arguments:

可以通过标准命令行参数重新映射组件管理器名称和命名空间：

```
ros2 run rclcpp_components component_container __node:=MyContainer __ns:=/ns
```

In a second shell, components can be loaded by using the updated container name:

在第二个 shell 中，可以使用更新的容器名称加载组件：

```
ros2 component load /ns/MyContainer composition composition::Listener
```

Note 注意

Namespace remappings of the container do not affect loaded components.

容器的命名空间重映射不会影响已加载的组件。

Remap component names and namespaces

重新映射组件名称和命名空间

Component names and namespaces may be adjusted via arguments to the load command.

组件名称和名称空间可以通过 load 命令的参数进行调整。

In the first shell, start the component container:

在第一个 shell 中，启动组件容器：

```
ros2 run rclcpp_components component_container
```

Some examples of how to remap names and namespaces:

有关如何重新映射名称和名称空间的一些示例：

```
# Remap node name
ros2 component load /ComponentManager composition composition::Talker --node-name talker2
# Remap namespace
ros2 component load /ComponentManager composition composition::Talker --node-namespace /ns
# Remap both
ros2 component load /ComponentManager composition composition::Talker --node-name talker3 --node-namespace /ns2
```

The corresponding entries appear in `ros2 component list`:

`ros2 component list` 相应的条目显示在：

```
$ ros2 component list
/ComponentManager
  1   /talker2
  2   /ns/talker
  3   /ns2/talker3
```

Note 注意

Namespace remappings of the container do not affect loaded components.

容器的命名空间重映射不会影响已加载的组件。

Introduction to msg and srv interfaces

msg 和 srv 接口简介

INCOMPLETE: this is a draft of an upcoming tutorial for creating and using custom ROS interfaces.

未完成：这是即将发布的用于创建和使用自定义 ROS 接口的教程的草稿。

Disclaimer: The code provided is to support the explanation, it is likely outdated and should not be expected to compile as is

免责声明：提供的代码是为了支持解释，它可能已经过时，不应该按原样编译

- msg: msg files are simple text files that describe the fields of a ROS message. They are used to generate source code for messages in different languages.
  msg：msg 文件是描述 ROS 消息字段的简单文本文件。它们用于生成不同语言的消息的源代码。
- srv: an srv file describes a service. It is composed of two parts: a request and a response. The request and response are message declarations.
  srv：srv 文件描述服务。它由两部分组成：请求和响应。请求和响应是消息声明。

msgs are just simple text files with a field type and field name per line. The field types you can use are:

msgs 只是简单的文本文件，每行有一个字段类型和字段名称。可以使用的字段类型是：

- int8, int16, int32, int64 (plus uint*)
- float32, float64
- string
- other msg files
- variable-length array[], fixed-length array[C], bounded-length array[<=C]

Here is an example of a msg that uses a string primitive, and two other msgs:

这是一个使用字符串和另外两个消息的 msg 示例：

```
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

srv files are just like msg files, except they contain two parts: a request and a response. The two parts are separated by a '—' line. Here is an example of a srv file:

srv 文件类似 msg 文件，但是它们包含两部分：请求和响应。这两部分用' - '线分开。

以下是 srv 文件的示例：

```
float64 A
float64 B
---
float64 Sum
```

In the above example, A and B are the request, and Sum is the response.

在上面的例子中，A 和 B 是请求，Sum 是响应。

msg files are stored in the `msg` directory of a package, and srv files are stored in the `srv` directory.

msg 文件存储在包的 `msg` 目录中，srv 文件存储在 `srv` 目录中。

These are just simple examples. For more information about how to create msg and srv files please refer to About ROS Interfaces.

这些只是简单的例子。有关如何创建 msg 和 srv 文件的更多信息，请参考关于 ROS 接口。

Creating a msg package 创建一个 msg 包

NOTE: only ament_cmake packages can generate messages currently (not ament_python packages).

注意：只有 ament_cmake 包可以生成当前的消息（不是 ament_python 包）。

For this tutorial we will use the packages stored in the rosidl_tutorials repository.

在本教程中，将使用存储在 rosidl_tutorials 库中的包。

```
cd ~/ros2_overlway_ws/src
git clone -b rosidl_tutorials https://github.com/ros2/tutorials.git
cd rosidl_tutorials/rosidl_tutorials_msgs
```

Creating a msg file 创建一个 msg 文件

Here we will create a message meant to carry information about an individual.

在这里，将创建一条消息，用于传递有关个人的信息。

Open `msg/Contact.msg` and you will see:

打开 `msg/Contact.msg`，会看到：

```
bool FEMALE=true
bool MALE=false

string first_name
string last_name
bool gender
uint8 age
string address
```

This message is composed of 5 fields: 此消息由 5 个字段组成：

- first_name: of type string
  first_name：类型为字符串
- last_name: of type string
  last_name：类型为字符串
- gender: of type bool, that can be either MALE or FEMALE
  性别：bool 类型，可以是男性或女性

- age: of type uint8
  年龄：uint8 型
- address: of type string
  地址：字符串类型

There's one more step, though. We need to make sure that the msg files are turned into source code for C++, Python, and other languages.
不过还有一个步骤。需要确保将 msg 文件转换为 C ++、Python 和其他语言的源代码。

Building msg files 编译 msg 文件

Open the `package.xml`, and uncomment these two lines:
打开 `package.xml`，并取消注释这两行：

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>

<exec_depend>rosidl_default_runtime</exec_depend>
```

Note that at build time, we need "rosidl_default_generators", while at runtime, we only need "rosidl_default_runtime".
请注意，在编译时，我们需要"rosidl_default_generators"，而在运行时，我们只需要"rosidl_default_runtime"。
Open the `CMakeLists.txt` and make sure that the following lines are uncommented.
打开 `CMakeLists.txt` 并确保取消以下行的注释。
Find the package that generates message code from msg/srv files:
找到从 msg / srv 文件生成消息代码的包：

```
find_package(rosidl_default_generators REQUIRED)
```

Declare the list of messages you want to generate: 声明要生成的消息列表：

```
set(msg_files
  "msg/Contact.msg"
)
```

By adding the .msg files manually, we make sure that CMake knows when it has to reconfigure the project after you add other .msg files.
通过手动添加.msg 文件，我们确保 CMake 知道在添加其他.msg 文件后何时必须重新配置项目。
Generate the messages: 生成消息：

```
rosidl_generate_interfaces(${PROJECT_NAME}
  ${msg_files}
)
```

Also make sure you export the message runtime dependency:
还要确保导出消息运行时依赖项：

```
ament_export_dependencies(rosidl_default_runtime)
```

Now you're ready to generate source files from your msg definition.
现在已准备好从 msg 定义生成源文件。

Creating an srv file 创建 srv 文件

We will now add a srv declaration to our package. 现在将向包中添加 srv 声明。
Open the srv/AddTwoFloats.srv file and paste this srv declaration:
打开 srv / AddTwoFloats.srv 文件并粘贴此 srv 声明：

```
float64 a
float64 b
---
float64 sum
```

Building srv files 编译 srv 文件

Declare the service in the `CMakeLists.txt`:
在 `CMakeLists.txt` 以下内容中声明服务：

```
set(srv_files
  "srv/AddTwoFloats.srv")
```

Modify the existing call to rosidl_generate_interfaces to generate the service in addition to the messages:
修改现有的 rosidl_generate_interfaces 调用以生成除消息之外的服务：

```
rosidl_generate_interfaces(${PROJECT_NAME}
  ${msg_files}
  ${srv_files}
)
```

Using custom messages 使用自定义消息

Using msg/srv from other packages 从其他包使用 msg/srv

Let’s write a C++ node using the Contact.msg we created in the previous section.
使用在上一节中创建的 Contact.msg 编写一个 C ++节点。

Go to the rosidl_tutorials package and open the src/publish_contact.cpp file.
转到 rosidl_tutorials 包并打开 src / publish_contact.cpp 文件。

```
#include <iostream>
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "rosidl_tutorials_msgs/msg/contact.hpp"

using namespace std::chrono_literals;
class ContactPublisher : public rclcpp::Node
{
public:
  ContactPublisher()
  : Node("address_book_publisher")
  {
    contact_publisher_ = this->create_publisher<rosidl_tutorials_msgs::msg::Contact>("contact");

    auto publish_msg = [this]() -> void {
      auto msg = std::make_shared<rosidl_tutorials_msgs::msg::Contact>();

      msg->first_name = "John";
      msg->last_name = "Doe";
      msg->age = 30;
      msg->gender = msg->MALE;
      msg->address = "unknown";

      std::cout << "Publishing Contact\nFirst:" << msg->first_name <<
        "  Last:" << msg->last_name << std::endl;

      contact_publisher_->publish(msg);
    };
    timer_ = this->create_wall_timer(1s, publish_msg);
  }
private:
  rclcpp::Publisher<rosidl_tutorials_msgs::msg::Contact>::SharedPtr contact_publisher_;
  rclcpp::timer::TimerBase::SharedPtr timer_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
```

```
  auto publisher_node = std::make_shared<ContactPublisher>();
  rclcpp::spin(publisher_node);
  return 0;
}
```

The code explained 代码解释

```
#include "rosidl_tutorials_msgs/msg/contact.hpp"
```

Here we include the header of the message that we want to use. 这里包含想要使用的消息的头文件。

```
ContactPublisher(): Node("address_book_publisher")
{
```

Here we define a node 这里定义一个节点

```
auto publish_msg = [this]() -> void {
```

A publish_msg function to send our message periodically 一个 publish_msg 函数，用于定期发送消息

```
auto msg = std::make_shared<rosidl_tutorials_msgs::msg::Contact>();

msg->first_name = "John";
msg->last_name = "Doe";
msg->age = 30;
msg->gender = msg->MALE;
msg->address = "unknown";
```

We create a Contact message and populate its fields. 创建一个 Contact 消息并填充其字段。

```
std::cout << "Publishing Contact\nFirst:" << msg->first_name <<
  "   Last:" << msg->last_name << std::endl;
contact_publisher_->publish(msg);
```

Finally we publish it 最后发布它

```
timer_ = this->create_wall_timer(1s, publish_msg);
```

Create a 1second timer to call our `publish_msg` function every second
创建一个 1 秒的计时器，每秒调用 `publish_msg` 函数

Now let's build it! 现在来编译吧！

To use this message we need to declare a dependency on rosidl_tutorials_msgs in the `package.xml`:

要使用此消息，需要在以下内容中声明对 rosidl_tutorials_msgs 的依赖 package.xml：

```
<build_depend>rosidl_tutorials_msgs</build_depend>
<exec_depend>rosidl_tutorials_msgs</exec_depend>
```

And also in the CMakeLists.txt: 而且在 CMakeLists.txt：

```
find_package(rosidl_tutorials_msgs REQUIRED)
```

And finally we must declare the message package as a target dependency for the executable.
最后，必须将消息包声明为可执行文件的目标依赖项。

```
ament_target_dependencies(publish_contact
  "rclcpp"
  "rosidl_tutorials_msgs"
)
```

Using msg/srv from the same package 从使用同一个包 msg/srv

While most of the time messages are declared in interface packages, it can be convenient to declare, create and use messages all in the one package.
虽然大多数时候消息是在接口包中声明的，但是在一个包中声明，创建和使用消息都很方便。

We will create a message in our rosidl_tutorials package. Create a msg directory in the rosidl_tutorials package and AddressBook.msg inside that directory. In that msg paste:
在 rosidl_tutorials 包中创建一条消息。在 rosidl_tutorials 包中创建一个 msg 目录，在该目录中创建 AddressBook.msg。在那个 msg 粘贴中：

```
rosidl_tutorials_msgs/Contact[] address_book
```

As you can see we define a message based on the Contact message we created earlier.
根据之前创建的 Contact 消息定义消息。
To generate this message we need to declare a dependency on this package in the package.xml:
要生成此消息，需要在以下内容中声明对此包的依赖 package.xml：

```
<build_depend>rosidl_tutorials_msgs</build_depend>
<exec_depend>rosidl_tutorials_msgs</exec_depend>
```

And in the CMakeLists.txt: 并在 CMakeLists.txt：

```
find_package(rosidl_tutorials_msgs REQUIRED)
set(msg_files
  "msg/AddressBook.msg"
)
rosidl_generate_interfaces(${PROJECT_NAME}
  ${msg_files}
  DEPENDENCIES rosidl_tutorials_msgs
)
```

Now we can start writing code that uses this message. 现在可以开始编写使用此消息的代码。

Open src/publish_address_book.cpp: 打开 src / publish_address_book.cpp：

```
#include <iostream>
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "rosidl_tutorials/msg/address_book.hpp"
#include "rosidl_tutorials_msgs/msg/contact.hpp"

using namespace std::chrono_literals;

class AddressBookPublisher : public rclcpp::Node{public:
  AddressBookPublisher()
  : Node("address_book_publisher")
  {
    address_book_publisher_ =
      this->create_publisher<rosidl_tutorials::msg::AddressBook>("address_book");

    auto publish_msg = [this]() -> void {
      auto msg = std::make_shared<rosidl_tutorials::msg::AddressBook>();
      {
        rosidl_tutorials_msgs::msg::Contact contact;
        contact.first_name = "John";
        contact.last_name = "Doe";
        contact.age = 30;
        contact.gender = contact.MALE;
        contact.address = "unknown";
        msg->address_book.push_back(contact);
      }
      {
        rosidl_tutorials_msgs::msg::Contact contact;
        contact.first_name = "Jane";
        contact.last_name = "Doe";
        contact.age = 20;
```

```
            contact.gender = contact.FEMALE;
            contact.address = "unknown";
            msg->address_book.push_back(contact);
        }

        std::cout << "Publishing address book:" << std::endl;
        for (auto contact : msg->address_book) {
            std::cout << "First:" << contact.first_name << "    Last:" << contact.last_name <<
                std::endl;
        }

        address_book_publisher_->publish(msg);
    };
    timer_ = this->create_wall_timer(1s, publish_msg);
  }
private:
  rclcpp::Publisher<rosidl_tutorials::msg::AddressBook>::SharedPtr address_book_publisher_;
  rclcpp::timer::TimerBase::SharedPtr timer_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);

  auto publisher_node = std::make_shared<AddressBookPublisher>();

  rclcpp::spin(publisher_node);

  return 0;
}
```

The code explained 代码解释

```
#include "rosidl_tutorials/msg/address_book.hpp"
```

We include the header of our newly created AddressBook msg.
包括新创建的 AddressBook 消息的头文件。

```
#include "rosidl_tutorials_msgs/msg/contact.hpp"
```

Here we include the header of the Contact msg in order to be able to add contacts to our address_book.
这里包含 Contact msg 的标题，以便能够将联系人添加到 address_book。

```
using namespace std::chrono_literals;
```

```
class AddressBookPublisher : public rclcpp::Node
{
public:
  AddressBookPublisher()
  : Node("address_book_publisher")
  {
    address_book_publisher_ =
      this->create_publisher<rosidl_tutorials::msg::AddressBook>("address_book");
```

We create a node and an AddressBook publisher. 创建一个节点和一个 AddressBook 发布器。

```
auto publish_msg = [this]() -> void {
```

We create a callback to publish the messages periodically
创建一个回调来定期发布消息

```
auto msg = std::make_shared<rosidl_tutorials::msg::AddressBook>();
```

We create an AddressBook message instance that we will later publish.
创建一个稍后将发布的 AddressBook 消息实例。

```
{
rosidl_tutorials_msgs::msg::Contact contact;
contact.first_name = "John";
contact.last_name = "Doe";
contact.age = 30;
contact.gender = contact.MALE;
contact.address = "unknown";
msg->address_book.push_back(person);
}
{
rosidl_tutorials_msgs::msg::Contact person;
contact.first_name = "Jane";
contact.last_name = "Doe";
contact.age = 20;
contact.gender = contact.FEMALE;
contact.address = "unknown";
msg->address_book.push_back(contact);
}
```

We create and populate Contact messages and add them to our address_book message.
创建并填充联系人消息并将其添加到 address_book 消息中。

```
std::cout << "Publishing address book:" << std::endl;
for (auto contact : msg->address_book)
{
  std::cout << "First:" << contact.first_name << "    Last:" << contact.last_name <<
      std::endl;
}
address_book_publisher_->publish(msg);
```

Finally send the message periodically. 最后定期发送消息。

```
timer_ = this->create_wall_timer(1s, publish_msg);
```

Create a 1second timer to call our `publish_msg` function every second
创建一个 1 秒的计时器，每秒调用 `publish_msg` 函数

Now let's build it! We need to create a new target for this node in the `CMakeLists.txt`:
现在来编译吧！需要在 `CMakeLists.txt` 以下位置为此节点创建新目标：

```
add_executable(publish_address_book
  src/publish_address_book.cpp)

ament_target_dependencies(publish_address_book
  "rclcpp")
```

In order to use the messages generated in the same package we need to use the following cmake code:
为了使用在同一个包中生成的消息，需要使用以下 cmake 代码：

```
get_default_rmw_implementation(rmw_implementation)
find_package("${rmw_implementation}" REQUIRED)
get_rmw_typesupport(typesupport_impls "${rmw_implementation}" LANGUAGE "cpp")

foreach(typesupport_impl ${typesupport_impls})
  rosidl_target_interfaces(publish_address_book
    ${PROJECT_NAME} ${typesupport_impl}
  )
endforeach()
```

This finds the relevant generated C++ code from msg/srv and allows your target to link against them.
这将从 msg / srv 中找到相关的生成的 C ++代码，并允许目标链接它们。
You may have noticed that this step was not necessary when the interfaces being used were from a package that was built beforehand. This CMake code is only required when you are trying to use interfaces in the same package as that in which they are built.

可能已经注意到，当使用的接口来自事先编译的包时，此步骤不是必需的。仅当尝试在与编译它们的包相同的包中使用接口时，才需要此 CMake 代码。

查看一下 turtlesim 的 msg 和 srv。

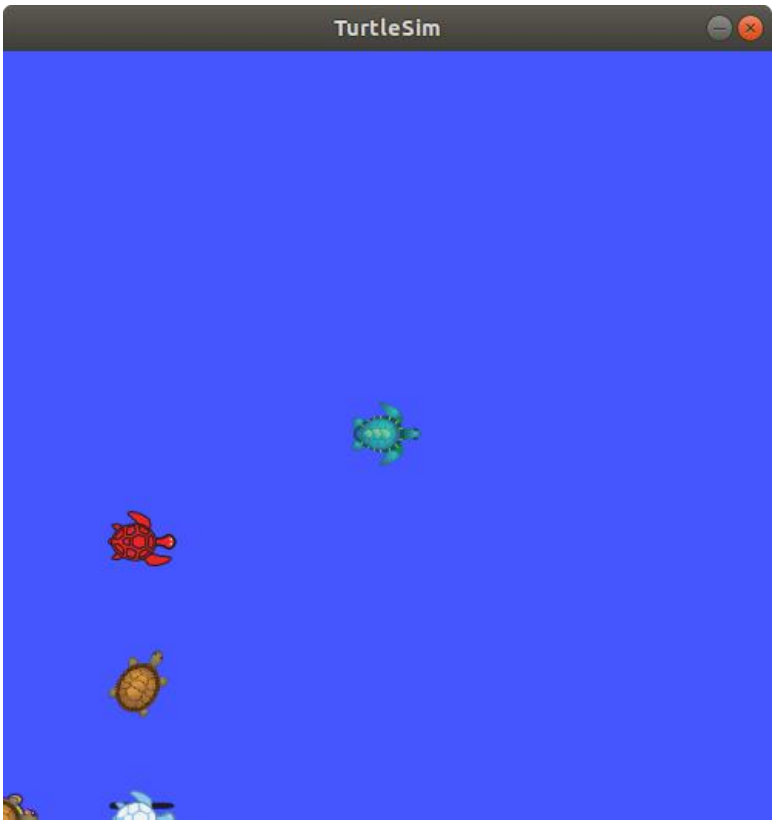
Pose.msg

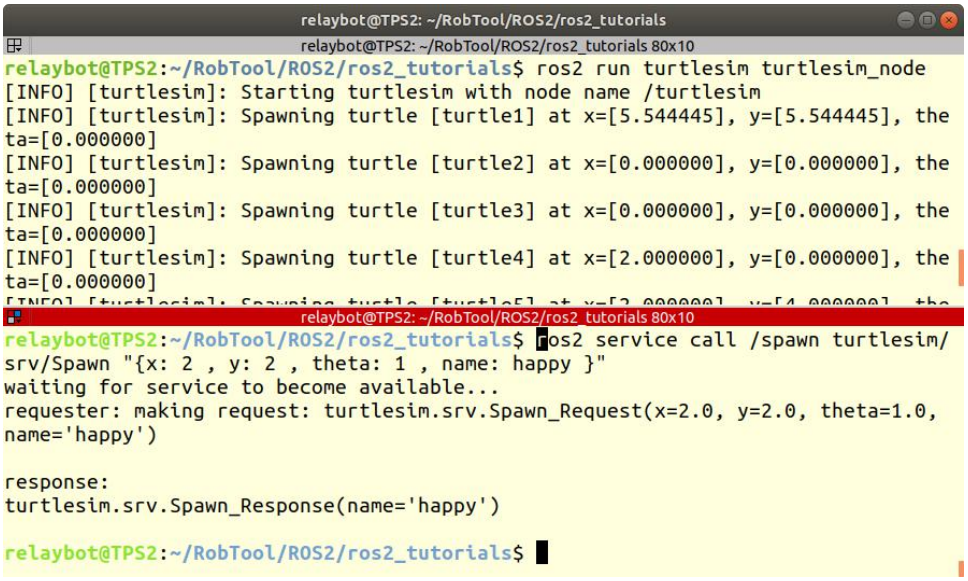
```
float32 x
float32 y
float32 theta

float32 linear_velocity
float32 angular_velocity
```

Spawn.srv

```
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be created and returned if this is empty
---
string name
```

熟练掌握 msg、service、srv 等命令的使用。




relaybot@TPS2: ~/RobTool/ROS2/ros2_tutorials
relaybot@TPS2: ~/RobTool/ROS2/ros2_tutorials 80x10
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$ ros2 run turtlesim turtlesim_node
[INFO] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], the
ta=[0.000000]
[INFO] [turtlesim]: Spawning turtle [turtle2] at x=[0.000000], y=[0.000000], the
ta=[0.000000]
[INFO] [turtlesim]: Spawning turtle [turtle3] at x=[0.000000], y=[0.000000], the
ta=[0.000000]
[INFO] [turtlesim]: Spawning turtle [turtle4] at x=[2.000000], y=[0.000000], the
ta=[0.000000]
relaybot@TPS2: ~/RobTool/ROS2/ros2_tutorials 80x10
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$ ros2 service call /spawn turtlesim/
srv/Spawn "{x: 2 , y: 2 , theta: 1 , name: happy }"
waiting for service to become available...
requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=2.0, theta=1.0,
name='happy')
response:
turtlesim.srv.Spawn_Response(name='happy')
relaybot@TPS2:~/RobTool/ROS2/ros2_tutorials$

Actions 行动

About 关于

Actions are a form of asynchronous communication in ROS. Action clients send goal requests to action servers. Action servers send goal feedback and results to action clients. For more detailed information about ROS actions, please refer to the design article.

行动是 ROS 中异步通信的一种形式。*Action 客户端*将目标请求发送到*操作服务器*。*操作服务器*将目标反馈和结果发送给*操作客户端*。有关 ROS 操作的更多详细信息，请参考设计文章。

This document contains a list of tutorials related to actions. For reference, after completing all of the tutorials you should expect to have a ROS package that looks like the package action_tutorials.

本文档包含与操作相关的教程列表。作为参考，在完成所有教程之后，需要有一个类似于 action_tutorials 的 ROS 包。

Prequisites 预备基础

Install ROS (Dashing or later) 安装 ROS（Dashing 或更高版本）

Install colcon 安装 colcon

Setup a workspace and create a package named `action_tutorials`:

设置工作区并创建名为 `action_tutorials` 的包：

Remember to source your ROS 2 installation. 记得要安装 ROS 2。

Linux / OSX:

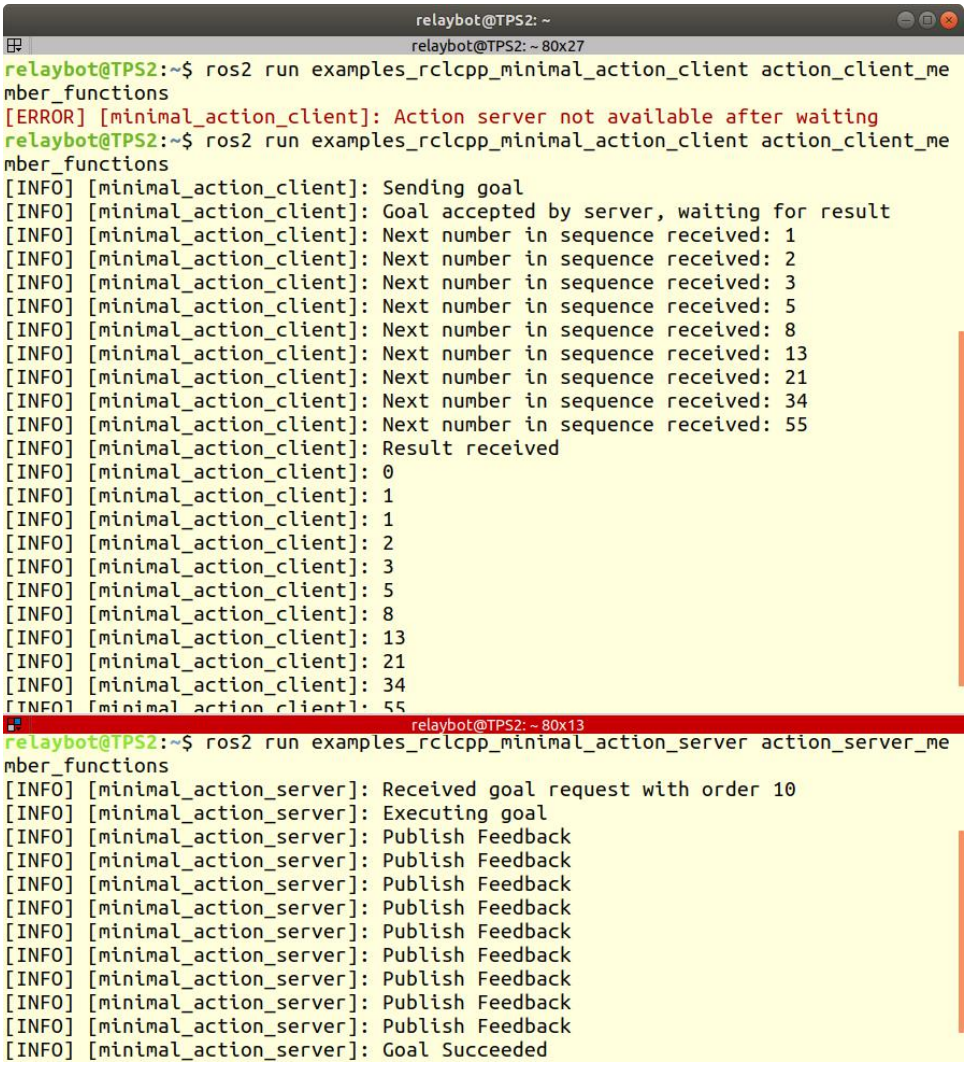
```
mkdir -p action_ws/src
cd action_ws/src
ros2 pkg create action_tutorials
```

Windows:

```
mkdir -p action_ws\srccd action_ws\src
ros2 pkg create action_tutorials
```

Tutorials

- Creating an Action
- Writing an Action Server (C++)
- Writing an Action Client (C++)
- Writing an Action Server (Python)
- Writing an Action Client (Python)



在完成文档阅读之后，学习源码：

Fibonacci.action

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

Server/member_functions.cpp

```
#include <inttypes.h>
#include <memory>
#include "example_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
// TODO(jacobperron): Remove this once it is included as part of 'rclcpp.hpp'
#include "rclcpp_action/rclcpp_action.hpp"

class MinimalActionServer : public rclcpp::Node
{
public:
  using Fibonacci = example_interfaces::action::Fibonacci;
```

```
  using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

  explicit MinimalActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
  : Node("minimal_action_server", options)
  {
    using namespace std::placeholders;

    this->action_server_ = rclcpp_action::create_server<Fibonacci>(
      this->get_node_base_interface(),
      this->get_node_clock_interface(),
      this->get_node_logging_interface(),
      this->get_node_waitables_interface(),
      "fibonacci",
      std::bind(&MinimalActionServer::handle_goal, this, _1, _2),
      std::bind(&MinimalActionServer::handle_cancel, this, _1),
      std::bind(&MinimalActionServer::handle_accepted, this, _1));
  }

private:
  rclcpp_action::Server<Fibonacci>::SharedPtr action_server_;

  rclcpp_action::GoalResponse handle_goal(
    const rclcpp_action::GoalUUID & uuid,
    std::shared_ptr<const Fibonacci::Goal> goal)
  {
    RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
    (void)uuid;
    // Let's reject sequences that are over 9000
    if (goal->order > 9000) {
      return rclcpp_action::GoalResponse::REJECT;
    }
    return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
  }

  rclcpp_action::CancelResponse handle_cancel(
    const std::shared_ptr<GoalHandleFibonacci> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
    (void)goal_handle;
    return rclcpp_action::CancelResponse::ACCEPT;
  }

  void execute(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Executing goal");
```

```
    rclcpp::Rate loop_rate(1);
    const auto goal = goal_handle->get_goal();
    auto feedback = std::make_shared<Fibonacci::Feedback>();
    auto & sequence = feedback->sequence;
    sequence.push_back(0);
    sequence.push_back(1);
    auto result = std::make_shared<Fibonacci::Result>();

    for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
      // Check if there is a cancel request
      if (goal_handle->is_canceling()) {
        result->sequence = sequence;
        goal_handle->canceled(result);
        RCLCPP_INFO(this->get_logger(), "Goal Canceled");
        return;
      }
      // Update sequence
      sequence.push_back(sequence[i] + sequence[i - 1]);
      // Publish feedback
      goal_handle->publish_feedback(feedback);
      RCLCPP_INFO(this->get_logger(), "Publish Feedback");

      loop_rate.sleep();
    }

    // Check if goal is done
    if (rclcpp::ok()) {
      result->sequence = sequence;
      goal_handle->succeed(result);
      RCLCPP_INFO(this->get_logger(), "Goal Suceeded");
    }
  }

  void handle_accepted(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
  {
    using namespace std::placeholders;
    // this needs to return quickly to avoid blocking the executor, so spin up a new thread
    std::thread{std::bind(&MinimalActionServer::execute, this, _1), goal_handle}.detach();
  }
};  // class MinimalActionServer

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
```

```
  auto action_server = std::make_shared<MinimalActionServer>();

  rclcpp::spin(action_server);

  rclcpp::shutdown();
  return 0;
}
```

Client/member_functions.cpp

```
#include <inttypes.h>
#include <memory>
#include <string>
#include <iostream>
#include "example_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
// TODO(jacobperron): Remove this once it is included as part of 'rclcpp.hpp'
#include "rclcpp_action/rclcpp_action.hpp"

class MinimalActionClient : public rclcpp::Node
{
public:
  using Fibonacci = example_interfaces::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;

  explicit MinimalActionClient(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions())
  : Node("minimal_action_client", node_options), goal_done_(false)
  {
    this->client_ptr_ = rclcpp_action::create_client<Fibonacci>(
      this->get_node_base_interface(),
      this->get_node_graph_interface(),
      this->get_node_logging_interface(),
      this->get_node_waitables_interface(),
      "fibonacci");

    this->timer_ = this->create_wall_timer(
      std::chrono::milliseconds(500),
      std::bind(&MinimalActionClient::send_goal, this));
  }

  bool is_goal_done() const
  {
    return this->goal_done_;
  }
```

```
  void send_goal()
  {
    using namespace std::placeholders;

    this->timer_->cancel();

    this->goal_done_ = false;

    if (!this->client_ptr_) {
      RCLCPP_ERROR(this->get_logger(), "Action client not initialized");
    }

    if (!this->client_ptr_->wait_for_action_server(std::chrono::seconds(10))) {
      RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
      this->goal_done_ = true;
      return;
    }

    auto goal_msg = Fibonacci::Goal();
    goal_msg.order = 10;

    RCLCPP_INFO(this->get_logger(), "Sending goal");

    auto send_goal_options = rclcpp_action::Client<Fibonacci>::SendGoalOptions();
    send_goal_options.goal_response_callback =
      std::bind(&MinimalActionClient::goal_response_callback, this, _1);
    send_goal_options.feedback_callback =
      std::bind(&MinimalActionClient::feedback_callback, this, _1, _2);
    send_goal_options.result_callback =
      std::bind(&MinimalActionClient::result_callback, this, _1);
    auto goal_handle_future = this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
  }

private:
  rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;
  rclcpp::TimerBase::SharedPtr timer_;
  bool goal_done_;

  void goal_response_callback(std::shared_future<GoalHandleFibonacci::SharedPtr> future)
  {
    auto goal_handle = future.get();
    if (!goal_handle) {
      RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
    } else {
      RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
```

```
    }
  }

  void feedback_callback(
    GoalHandleFibonacci::SharedPtr,
    const std::shared_ptr<const Fibonacci::Feedback> feedback)
  {
    RCLCPP_INFO(
      this->get_logger(),
      "Next number in sequence received: %" PRId64,
      feedback->sequence.back());
  }

  void result_callback(const GoalHandleFibonacci::WrappedResult & result)
  {
    this->goal_done_ = true;
    switch (result.code) {
      case rclcpp_action::ResultCode::SUCCEEDED:
        break;
      case rclcpp_action::ResultCode::ABORTED:
        RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
        return;
      case rclcpp_action::ResultCode::CANCELED:
        RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
        return;
      default:
        RCLCPP_ERROR(this->get_logger(), "Unknown result code");
        return;
    }

    RCLCPP_INFO(this->get_logger(), "Result received");
    for (auto number : result.result->sequence) {
      RCLCPP_INFO(this->get_logger(), "%" PRId64, number);
    }
  }
};  // class MinimalActionClient

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  auto action_client = std::make_shared<MinimalActionClient>();

  while (!action_client->is_goal_done()) {
    rclcpp::spin_some(action_client);
  }
```

```
    rclcpp::shutdown();
    return 0;
}
```

Intel ROS2 Projects

英特尔机器人操作系统二项目

Intel® Robotics Open Source Project (Intel® ROS Project) to enable object detection/location/tracking, people detection, vehicle detection, industry robot arm grasp point analysis with kinds of Intel technologies and platforms, including CPU, GPU, Intel® Movidius™ NCS optimized deep learning backend, FPGA, Intel® RealSense™ camera, etc.

英特尔®机器人开源项目（英特尔®ROS 项目），支持目标检测/定位/跟踪、人员检测、车辆检测、工业机器人手臂抓取点分析，采用各类英特尔技术和平台，包括 CPU、GPU、英特尔®Movidius™ NCS 优化了深度学习后端、FPGA、英特尔®实感™摄像头等。

Key Projects 重点项目

We are working on below ROS2 projects and publish source code through https://github.com/intel/ or ROS2 github repo gradually.

英特尔正在开发 ROS2 项目，并逐步通过 https://github.com/intel/或 ROS2 github repo 发布源代码。

- ROS2 OpenVINO: ROS2 package for Intel® Visual Inference and Neural Network Optimization Toolkit to develop multiplatform computer vision solutions.
  英特尔®视觉推理和神经网络优化工具包的 ROS2 包，用于开发多平台计算机视觉解决方案。
- ROS2 RealSense Camera: ROS2 package for Intel® RealSense™ D400 serial cameras
  英特尔®实感™D400 系列相机的 ROS2 功能包
- ROS2 Movidius NCS: ROS2 package for object detection with Intel® Movidius™ Neural Computing Stick (NCS).
  ROS2 软件包，使用英特尔®Movidius™神经计算棒（NCS）进行物体检测。
- ROS2 Object Messages: ROS2 messages for object.
  ROS2 的目标消息。
- ROS2 Object Analytics: ROS2 package for object detection, tracking and 2D/3D localization.
  用于目标检测、跟踪和 2D / 3D 定位的 ROS2 包。
- ROS2 Message Filters: ROS2 package for message synchronization with time stamp.
  ROS2 包，用于与时间戳进行消息同步。
- ROS2 CV Bridge: ROS2 package to bridge with openCV.
  与 openCV 桥接的 ROS2 包。
- ROS2 Object Map: ROS2 package to mark tag of objects on map when SLAM based on information provided by ROS2 object analytics.
  当 SLAM 基于 ROS2 目标分析提供的信息时，此 ROS2 包用于标记地图上对象。
- ROS2 Moving Object: ROS2 package to provide object motion information (like object velocity on x, y, z axis) based on information provided by ROS2 object analytics.
  ROS2 包根据 ROS2 目标分析提供的信息提供目标运动信息（如 x，y，z 轴上的目标速度）。

- ROS2 Grasp Library: ROS2 package for grasp position analysis, and compatible with MoveIt grasp interfaces.
  ROS2 包用于抓取位置分析，并兼容 MoveIt!抓取接口。
- ROS2 Navigation: ROS2 package for robot navigation, it's already integrated to ROS2 Crystal release.
  用于机器人导航的 ROS2 包，它已经集成到 ROS2 Crystal 发行版中。
- Robot SDK: An open source project which enables developers to easily and efficiently create, customize, optimize, and deploy a robot software stack to an Autonomous Mobile Robot (AMR) platform based on the Robot Operating System 2 (ROS2) framework.
  一个开源项目，使开发人员能够轻松高效地创建、定制、优化和部署基于机器人操作系统二（Robot Operating System 2，ROS2）框架的自主移动机器人（ Autonomous Mobile Robot，AMR）平台的机器人软件集。

Reference 参考

ROS components at: http://wiki.ros.org/IntelROSProject shows the relationship among those packages, which also applies to ROS2.
ROS 组件位于：http://wiki.ros.org/IntelROSProject。显示了这些包之间的关系，同时也适用于 ROS2。

2019-09-12