
机器人操作系统二（ROS2）浅析

[美] Jason M. O'Kane 著

肖军浩 译

更新部分：

张瑞雷 编

刘锦涛 审

作者通讯地址：

国防科学技术大学机电工程与自动化学院

肖军浩 博士

地址：湖南省长沙市开福区砚瓦池正街 137号

邮编：410073

版权信息：

©2015，肖军浩，版权所有。

本书上传至互联网供读者免费下载，版权归肖军浩个人所有。未经许可，不得以任何方式复制或抄袭本书之部分或全部内容用于商业目的。

前 言

因为集成了全世界机器人领域顶级科研机构，包括斯坦福大学、麻省理工学院、慕尼黑工业大学、加州大学伯克利分校、佐治亚理工大学、弗莱堡大学、东京大学等多年的研究成果，开源机器人操作系统（Robot Operating System，简称 ROS）一问世便受到了科研人员的广泛关注。随后，ROS 又借助开源的魅力吸引了世界各地机器人领域的仁人志士群策群力，推动其不断进步。2013 年麻省理工学院科技评论（MIT Technology Review）指出：“从 2010 年发布 1.0 版本以来，ROS 已经成为机器人软件的事实标准“（*de facto standard*）”。

译者在德国汉堡大学攻读博士学位期间，有幸于 2010 年成为 ROS 的第一批用户，并将其介绍给身边的同事以及国防科技大学的老师与同学。2013 年，译者所在研究团队将 NuBot 中型组足球机器人的软件系统移植到了 ROS 框架下，并于 2014 年和 2015 年分别参加了在巴西若昂佩索阿和中国合肥举办的机器人足球世界杯。使用 ROS 后 NuBot 足球机器人软件系统的鲁棒性、易用性和可维护性均有大幅度提高。对我们将 ROS 用于中型组足球机器人的工作，国际同行给予了非常积极的评价。

译者连续两年将 ROS 的使用作为国防科技大学高年级本科生实践教学的重要环节，发现一个普遍问题：学生能够迅速理解 ROS 的框架结构和基本概念，但是实际使用时问题却层出不穷，而其中大部分是共性问题！O’Kane 教授这本“A Gentle Introduction to ROS”对 ROS 初学过程中的常见问题做了全面的总结。当然，这

本书不仅仅是常见问题汇编，它还对概念和工具做了比在线文档更深入的剖析！故而，译者在征求 O'Kane 教授的同意后，将该书译为中文，供中国的 ROS 初学者在实际使用时参考。

经过多年迭代，ROS 2.0版本已经正式发布，修订版针对ROS 2.0更新特性进行完善，并将原书ROS 1.0版本内容由indigo更新为melodic。

为什么选择ROS 2.0?

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。

首先,感谢 O’Kane 教授对于本书翻译工作的肯定和支持。其次,本书的翻译得到国防科学技术大学“控制科学与工程高级专题”课程师生的大力支持,其中赵云云、李峻翔、肖志鹏、贾凡、朱琪、郭昭宇、王志强、陈春玉、魏翔宇分别参与了部分章节的翻译工作。此外,王祥科博士审阅了初稿并提出了许多宝贵的意见,对此,译者表示诚挚的谢意。最后,感谢 NuBot 研究团队全体成员对于本书的支持和帮助。

限于译者水平,书中难免会有不足之处,热切地希望得到各位读者的宝贵意见。作者的 E-mail 地址是: junhao.xiao@ieee.org。

译者

2015 年 9 月于长沙

目 录

第 1 章	绪论.....	1
1.1	选择 ROS 的理由.....	1
1.2	内容概述.....	5
1.3	行文约定.....	7
1.4	更多信息.....	7
1.5	下一章简介.....	10
第 2 章	入门概述.....	11
2.1	安装 ROS.....	11
2.2	配置账户.....	14
2.3	使用 TURTLESIM 的小例子.....	16
2.4	功能包/软件包 (PACKAGES)	18
2.5	节点管理器 (THE MASTER)	22
2.6	节点 (NODES)	23
2.7	话题和消息.....	26
2.8	一个更复杂的例子.....	39
2.9	问题检查.....	43
2.10	展望.....	43
第 3 章	编写 ROS 程序.....	45
3.1	创建工作区和功能包.....	45
3.2	你好, ROS!.....	48
3.3	发布者程序.....	55
3.4	订阅者程序.....	65
3.5	展望.....	71
第 4 章	日志消息.....	73
4.1	严重级别.....	73
4.2	示例程序.....	74
4.3	生成日志消息.....	76
4.4	查看日志消息.....	81
4.5	启用和禁用日志消息.....	88
4.6	展望.....	92
第 5 章	计算图源命名.....	93

5.1 全局名称.....	93
5.2 相对名称.....	95
5.3 私有名称.....	97
5.4 匿名名称 (ANONYMOUS NAMES)	98
5.5 展望.....	100
第 6 章 启动文件.....	101
6.1 使用启动文件.....	101
6.2 创建启动文件.....	105
6.3 在命名空间内启动节点.....	110
6.4 名称重映射 (REMAPPING NAMES)	113
6.5 启动文件的其他元素.....	119
6.6 展望.....	126
第 7 章 参数.....	127
7.1 通过命令行获取参数.....	127
7.2 例: TURTLESIM 中的参数.....	131
7.3 使用 C++ 获取参数.....	134
7.4 在启动文件中设置参数.....	137
7.5 展望.....	140
第 8 章 服务.....	141
8.1 服务的专用术语.....	141
8.2 从命令行查看和调用服务.....	142
8.3 客户端程序.....	148
8.4 服务器程序.....	153
8.5 展望.....	159
第 9 章 消息录制与回放.....	161
9.1 录制与回放包文件.....	161
9.2 示例: 正方形运动轨迹的包文件.....	163
9.3 启动文件里面的包文件.....	167
9.4 展望.....	170
第 10 章 总结.....	171
10.1 下一步.....	171
10.2 展望.....	174

第1章 绪论[†]

合抱之木，生于毫末；九层之台，起于累土；千里之行，始于足下。

——老子

本章主要介绍 ROS 系统的优势和本书的框架结构。

1.1 选择 ROS 的理由

近年来，机器人领域取得了举世瞩目的进展。性价比较高的机器人平台，包括地面移动机器人、旋翼无人机和类人机器人等，得到了广泛应用。更令人感到振奋的是，越来越多的高级智能算法让机器人的自主等级逐步提高。

尽管如此，对于机器人软件开发人员来说，仍然存在着诸多挑战。本书主要介绍一个软件平台，即机器人操作系统¹（**Robot Operating System**，或简称 **ROS**），它可以帮助提高机器人软件的开发效率。ROS 系统的官方定义如下：

ROS 是面向机器人的开源的元操作系统（meta-operating system）¹。它能够提供类似传统操作系统的诸多功能，如硬件抽象、底层设备控制、常用功能实现、进程间消息传递和程序包管理等。此外，它还提供相关工具和库，用于获取、编译、编辑代码以及在多个计算机之间运行程序完成分布式计算。

虽然上述定义很准确，强调了 ROS 与传统操作系统异同，

[†]本章由肖军浩、赵云云翻译。

¹<http://wiki.ros.org/ROS/Introduction>
<https://index.ros.org/doc/ros2/>

但可能仍然无法让读者抓住 ROS 的核心要义，尤其是使用 ROS 能够给机器人软件开发带来哪些优势。一般而言，学习一个新的系统框架，特别是 ROS 这样复杂多样的框架，往往要耗费大量的时间和精力，因此我们必须确保付出这个代价是有意义的。下面简单列举几个使用 ROS 能够解决的机器人软件开发问题。

分布式计算 现代机器人系统往往需要多个计算机同时运行多个进程，例如：

- > 一些机器人搭载多台计算机，每台计算机用于控制机器人的部分驱动器或传感器；
- > 即使只有一台计算机，通常仍将程序划分为独立运行且相互协作的小模块来完成复杂的控制任务，这也是常见的做法；
- > 当多个机器人需要协同完成一个任务时，往往需要互相通信来支撑任务的完成；
- > 用户通常通过台式机、笔记本或者移动设备发送指令控制机器人，这种人机交互接口可以认为是机器人软件的一部分。

单计算机或者多计算机不同进程间的通信问题是上述例子中的主要挑战。ROS 为实现上述通信提供两种相对简单、完备的机制，我们将在第三章和第八章进行详细讨论。

软件复用 随着机器人研究的快速推进，诞生了一批应对导航、路径规划、建图等通用任务的算法。当然，任何一个算法实用的前提是其能够应用于新的领域，且不必重复实现。事实上，如何将现有算法快速移植到不同系统一直是一个挑战，ROS 通过以下两种方法解决这个问题。

- > ROS 标准包（Standard Packages）提供稳定、可调式的各类重要机器人算法实现。

> **ROS**通信接口正在成为机器人软件互操作的事实标准，也就是说绝大部分最新的硬件驱动和最前沿的算法实现都可以在**ROS**中找到。例如，在**ROS**的官方网页²上有着大量的开源软件库，这些软件使用**ROS**通用接口，从而避免为了集成它们而重新开发新的接口程序。

综上所述，开发人员如果使用 **ROS** 可以——当然，在具备 **ROS** 基础知识后——将更多的时间用于新思想和新算法的设计与实现， 尽量避免重复实现已有的研究结果。

快速测试 为机器人开发软件比其他软件开发更具挑战性，主要是因为调试准备时间长，且调试过程复杂。况且，因为硬件维修、经费有限等因素，不一定随时有机器人可供使用。**ROS** 提供两种策略来解决上述问题。

> 精心设计的 **ROS** 系统框架将底层硬件控制模块和顶层数据处理与决策模块分离，从而可以使用模拟器替代底层硬件模块，独立测试顶层部分，提高测试效率。

> **ROS** 另外提供了一种简单的方法可以在调试过程中记录传感器数据及其他类型的消息数据，并在试验后按时间戳回放。通过这种方式，每次运行机器人可以获得更多的测试机会。例如，可以记录传感器的数据，并通过多次回放测试不同的数据处理算法。在 **ROS** 术语中，这类记录的数据叫作包（**bag**），一个被称为 **rosbag** 的工具可以用于记录和回放包数据，见第9章。

² <http://wiki.ros.org/browse>
<http://design.ros2.org/>

采用上述方案的一个最大优势是实现代码的“无缝连接”，因为实体机器人、仿真器和回放的包可以提供同样（至少是非常类似）的接口，上层软件不需要修改就可以与它们进行交互，实际上甚至不需要知道操作的对象是不是实体机器人。

当然，ROS 操作系统并不是唯一具备上述能力的机器人软件平台。在笔者看来，ROS 的最大不同在于来自机器人领域诸多开发人员的认可和支持，这种支持将促使 ROS 的未来不断发展、完善、进步。

对 **ROS** 的误解... 最后，让我们用一点时间来澄清容易对 ROS 产生的误解，另一个侧面来认识 ROS。

> ROS不是一种编程语言。实际上，ROS的主要代码由C++语言³编写，本书后续章节也会介绍如何在ROS中使用C++进行编程。客户端库的编写还可以使用Python⁴、Java⁵和Lisp⁶等其他多种语言⁷编写。

> ROS 不仅是一个函数库，除包含客户端库（**Client Libraries**）外，还包含一个中心服务器（**Central Server**）、一系列命令行工具、图形化界面工具以及编译环境。

> ROS不是集成开发环境。虽然ROS没有规定软件开发环境，但几乎所有的主流IDE⁸都可用于基于ROS的软件开发。此外，我们还可以根据个人喜好，使用普通的文本编辑器和命令行来完成相应的开发，而无需任何IDE。

³ <http://wiki.ros.org/roscpp>

⁴ <http://wiki.ros.org/rospy>

⁵ <http://wiki.ros.org/rosjava>

⁶ <http://wiki.ros.org/roslisp>

<http://docs.ros2.org/dashing/>

⁷ <http://wiki.ros.org/ClientLibraries>

<https://index.ros.org/doc/ros2/Concepts/ROS-2-Client-Libraries/>

1.2 内容概述

本书的主要目的是对 ROS 的概念和技术进行整体介绍，从而使读者具备独立编写 ROS 软件的基本技能，但并不涵盖以下几点：

- > 本书不是一本介绍如何编程的书籍。我们不打算详细讨论编程的基本概念。本书假设读者已经具备阅读、编写和理解 C++ 代码的基本能力。
- > 本书不是一本参考手册。关于 ROS 已有大量的资料可以查阅，包括在线指导手册⁹和参考资料¹⁰。本书无意也不可能取代这些资源。相反，本书只是介绍 ROS 最基本的特征，为后面更好地使用 ROS 提供一个好的开端。
- > 本书不是一本关于机器人算法的书籍。机器人学的研究，尤其是自主机器人控制算法的研究，可以说是相当有趣的，且针对这些问题已经提出了令人眼花缭乱的算法。本书并不针对某个具体算法进行讲解，而是告诉读者如何使用好 ROS 这一工具，更好地实现和测试感兴趣的算法。

⁸ <http://wiki.ros.org/IDEs>

⁹ <http://wiki.ros.org/ROS/Tutorials>
<https://index.ros.org/doc/ros2/Tutorials/#tutorials>

¹⁰ <http://wiki.ros.org/APIs>

章节及其依赖关系 本书的基本构架如图 1.1 所示，其中章节用矩形框表示，箭头描述他们之间的主要依赖关系，读者可以在这些约束下按照不同顺序阅读本书。

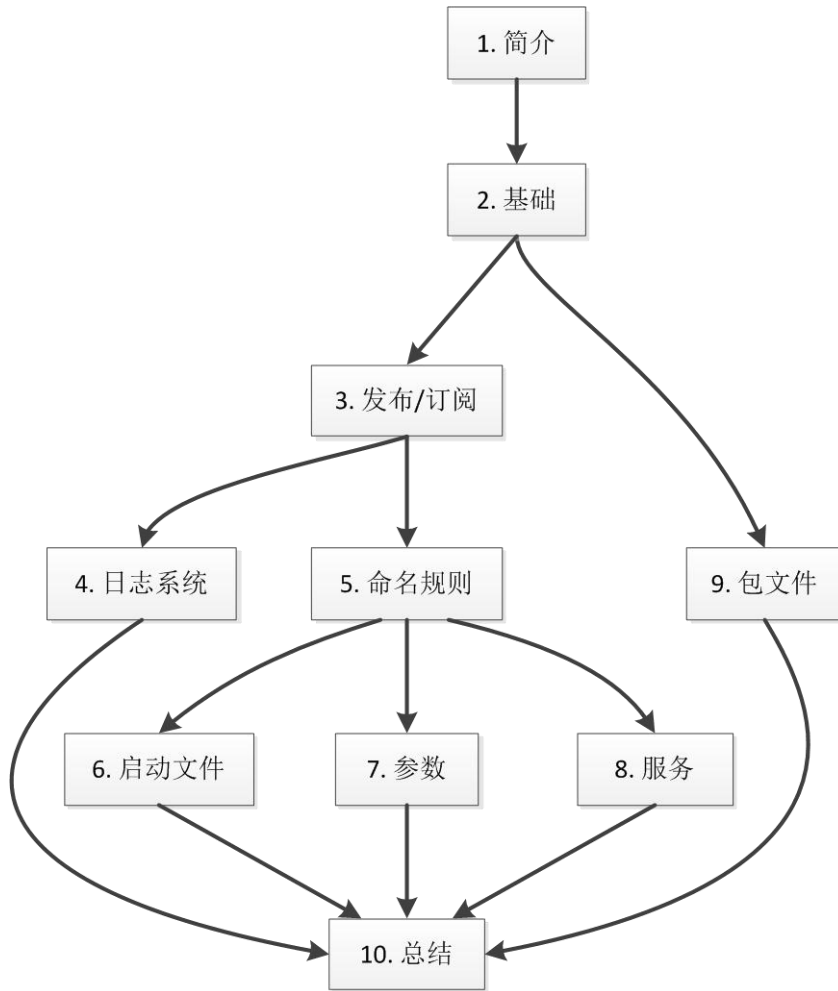


图 1.1 章节之间的相互依赖关系。

补充： action行动的相关介绍和说明。

> ROS1: <http://wiki.ros.org/actionlib>

> ROS2: <https://index.ros.org/doc/ros2/Tutorials/Actions/>

目标读者 本书能够为从事机器人技术相关领域的学生、研究人员和爱好者提供一个快速上手 ROS 的参考手册。我们希望读者对 Linux 系统有所了解（如基本的命令行操作、安装软件、编辑文件和设置环境变量等），熟悉 C++，并且对通过编程控制机器人感兴趣。我们进一步假设读者使用的 Linux 操作系统版本是 Ubuntu 14.04（在成文过程中最新的官方支持版本），并且命令行使用 bash shell。然而，这些选择对其他系统配置并不会造成太大影响，使用其他 Linux 发行版（尤其是基于 deb 包的发行版）和 shell 通常不会有问题。

更新： Ubuntu 18.04和Windows10同步介绍。

1.3 行文约定

在全书中，我们尝试预先提示一些可能出现的共性问题。这类警告是值得关注的，特别是当运行结果与期望有出入时，使用如下符号标记：



这个危险符号提示读者可能会出现的错误。

此外，某些章节中包含了只有部分读者可能会感兴趣的内容，这些内容对于概念理解没有影响，用如下符号进行标记：

该快进符号用于描述一些阅读过程中，尤其是初次接触本书，暂时可以跳过的内容。

1.4 更多信息

如前所述，本书并不是一本完备的 ROS 参考手册。几乎可以肯定，要真正动手使用 ROS 将要了解更多的细节。幸运的是，网上有关 ROS 和 ROS2 的信息十分丰富。

> 最为重要的是ROS开发联盟维护的的维基文档¹¹，其中包括大量教程。本书给出了其中一些页面的链接，在书中用脚注符号标记。如果使用主流的PDF阅读器阅读本书的电子版，则可以直接点击该符号在浏览器中打开对应网页。

> 如果在调试过程中遇到问题，可以登录类似Stack Exange（译者注：Stack Exchange是一系列的问答网站，每一个网站包含不同领域的问题）风格的Q&A网站¹²进行查询和提问。

另一个有效获取最新信息的途径就是ROS用户（ros-user）邮件列表¹³。

这里主要介绍两个细节以便读者明确相关概念。

发行版本（Distributions） ROS的主要版本称为发行版，其版本号¹⁴以顺序字母作为版本名的首字母来命名（这种方式与其他大型工程的版本命名方式类似，如Ubuntu、Android）。在书写本书的过程中，当前版本号indigo（2015年），其后续版本被命名为jade，将于2015年5月发布¹⁵（译者注：事实上，ROSjade已经与2015年5月23日按计划日期发布）。ROS的旧版本号包括hydro、groovy、feurte、electric、diamondback、C Turtle和box turtle。这些名称将会多次出现在本书中。

补充：推荐优先学习长期支持版本LTS的ROS1和ROS2，如melodic和dashing。

¹¹ <http://wiki.ros.org>

¹² <http://answers.ros.org>

¹³ <http://lists.ros.org/mailman/listinfo/ros-users>

¹⁴ <http://wiki.ros.org/Distributions>

¹⁵ <http://wiki.ros.org/indigo/Planning>



为了确保简单和时效性，本书假设读者使用 ROS indigo。

补充：ROS1和ROS2，如melodic和dashing。

如果需要在某些情况下使用 **hydro** 版本，本书中的绝大多数例子可以不经修改就可以使用。上述结论对 **groovy** 版本同样奏效，但有一点需要特别注意：在 **groovy** 之上的版本中，对于 **turtlesim** 仿真器的速度指令已经更新为用于很多实体机器人的标准信息类型和话题名称。其区别如下表所示：

版本	话题名称	消息类型
groovy	/turtle1/command_velocity	Turtlesim/ Velocity
indigo, hydro	/turtle1/cmd_vel	geometry_msgs/Twist

这个改变将引起一些实现上的变化：

当添加依赖库时，需要使用

turtlesim

来替换

geometry_msgs。

相应的头文件也要替换为

turtlesim/Velocity.h

而不再使用

geometry_msgs/Twist.h

turtlesim/Velocity 消息类型只包含两个域，即 **linear** 和 **angular**。上述两个域与 **geometry_msgs/Twist** 中的 **linear.x** 和 **angular.z** 含义相同。这个改变同样出现在命令行和 C++ 代码中。

补充：ROS2 出现较多变化，在后续介绍中详细说明。

编译系统（ Build Systems ）从groovy版本开始，ROS对软件的编译进行了重大改动。在groovy及之前的版本中，ROS采用rosbuild系统来完成软件的编译，而在新的版本中，则改用catkin编译系统。了解这一点非常重要，尤其在阅读参考教程时需要注意，编译系统的改变导致很多教程分为rosbuild和catkin两个版本。版本的选择可以通过操作教程页面顶端的按钮完成。虽然在有些情况下，rosbuild是更好的选择¹⁶，但本书主要介绍catkin系统的使用。

更新：

发行版本（ **Distributions** ）：

ROS 1.0:

目前，共提供Indigo、Kinetic和Melodic三个长期支持版本，并将于2020年5月推出Noetic。其中Melodic支持期限为2018年5月-2023年5月，也是本书更新ROS 1.0版本。

ROS 2.0:

最新版为Eloquent，长期支持版本为Dashing，并将于2020年5月推出Foxy。其中Dashing版本支持期限为2019年5月-2021年5月，也是本书更新ROS 2.0版本。

> <https://index.ros.org/doc/ros2/Releases/>

编译系统（ Build Systems ）：

ROS 1.0使用catkin:

> <https://catkin-tools.readthedocs.io/en/latest/>

ROS 2.0使用colcon:

> <https://colcon.readthedocs.io/en/released/>

1.5 下一章简介

在下一章，我们学习 ROS 的基本概念和工具，并开始使用 ROS。

预习：

安装

- [Dashing](#)
- [Eloquent](#)

普通用户

- [配置 ROS 2 环境](#)
- [介绍 turtlesim 和 rqt](#)
- [了解学习 ROS 2 节点 nodes](#)
- [了解学习 ROS 2 主题 topics](#)
- [了解学习 ROS 2 服务 services](#)
- [了解学习 ROS 2 参数 parameters](#)
- [了解学习 ROS 2 行动 actions](#)

- [使用 rqt_console](#)
- [创建启动文件 launch](#)
- [记录和播放数据 rosbag](#)

开发人员

- [创建工作区 ws](#)
- [创建一个 ROS 2 包](#)
- [编写简单的发布者和订阅者 \(C++\)](#)
- [编写简单的发布者和订阅者 \(Python\)](#)
- [编写简单的服务端和客户端 \(C++\)](#)
- [编写简单的服务端和客户端 \(Python\)](#)
- [ros2doctor 入门](#)

¹⁶ http://wiki.ros.org/catkin_or_rosbuild

第2章 入门概述[†]

工欲善其事，必先利其器。

——孔子

本章中，我们将简述 *ROS* 安装过程，介绍一些 *ROS* 的基本概念，并与运行中的 *ROS* 系统进行一些基本的交互。

在介绍用 *ROS* 写程序的细节之前，我们有必要先了解如何启动、运行 *ROS*，并理解 *ROS* 使用的一些基本概念。作为后续内容的基础，本章首先简单介绍如何安装 *ROS* 和配置用户账户，接着我们将浏览一个运行中的 *ROS* 系统（更确切地说，是一个 *turtlesim* 仿真器），最后将学习如何用命令行工具与这个系统进行交互。

2.1 安装 *ROS*

当然，使用 *ROS* 之前，必须首先确认已经在计算机上成功安装了 *ROS* 软件。如果你正在使用一台别人已经安装过 *ROS* 的计算机，包括我们后续章节将频繁使用的 *ros-indigo-turtlesim* 功能包，那么你可以直接跳到 2.2 节。事实上，安装过程有详细的帮助文档，

而且大部分都很简单^{1,2}。下面是对一些必要步骤的小结。

添加 **ROS** 软件源 使用超级用户（译者注：Ubuntu 下使用 `sudo` 命令得到超级用户权限）在根目录下，创建文件

```
/etc/apt/sources.list.d/ros-latest.list
```

[†] 本章由肖军浩、李峻翔、肖志鹏翻译。

¹ <http://wiki.ros.org/ROS/Installation>

² <http://wiki.ros.org/indigo/Installation/Ubuntu>

<https://index.ros.org/doc/ros2/Installation/#installationguide>

并将如下内容添加到此文件中：

```
deb http://packages.ros.org/ros/ubuntu trusty main
```



这一行是针对 Ubuntu 14.04 的，其代号为 **trusty**。但如果你用的是 Ubuntu 13.10，则可用 **saucy** 替 **trusty**。

其他Ubuntu版本——无论是更新的或更老的版本——都不支持预编译二进制包的安装。但是，如果是比

Ubuntu14.04 更新的版本，从源码安装ROS是一个较好的选择。

如果不确定自己的 Ubuntu 版本，可以在终端中使用这个命令：

```
lsb_release -a
```

输出将显示系统的代号和版本号。

安装软件包授权密钥 安装 ROS 软件包之前，你必须要有它的授权密钥。首先在下列网址下载：

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
```

如果下载成功，当前目录下会有一个较小的名为 **ros.key** 的二进制文件。接下来，用这个密钥配置 **apt** 软件包管理系统：

```
sudo apt-key add ros.key
```

完成该步后（**apt-key** 提示“ok”），可以放心删除 **ros.key**。

更新

Ubuntu:

Melodic:

设置软件源:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

设置密钥:

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

> <http://wiki.ros.org/melodic/Installation/Ubuntu>

Dashing:

设置区域:

```
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
```

设置软件源和密钥:

```
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 -
```

```
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'
```

> <https://index.ros.org/doc/ros2/Installation/Dashing/Linux-Install-Debian/>

Windows:

安装Visual Studio 2019;

Melodic:

设置软件源:

```
choco source add -n=ros-win -  
s="https://roswin.azurewebsites.net/api/v2" --priority=1
```

安装桌面完整版:

```
choco upgrade ros-melodic-desktop_full -y --execution-timeout=0
```

Dashing:

设置软件源:

```
choco source add -n=ros-win -  
s="https://roswin.azurewebsites.net/api/v2" --priority=1
```

安装桌面完整版:

```
choco upgrade ros-dashing-desktop -y --execution-timeout=0 --pre
```

> <http://wiki.ros.org/Installation/Windows>

³ <http://wiki.ros.org/indigo/Installation/Source>

<https://index.ros.org/doc/ros2/Installation/#installationguide>

下载软件包列表一旦配置完软件版本仓库（**repositories**）之后，可以用下列命令得到最新的可用软件包列表：

```
sudo apt-get update
```

需要注意的是，这会更新你系统中所有的软件版本仓库，而不仅仅是新添加的 **ROS** 库。

安装 **ROS** 软件包 现在可以安装 **ROS** 软件。最简单的方法是将 **ROS**

的核心系统进行完整安装：

```
sudo apt-get install ros-indigo-desktop-full
```

Melodic:

```
sudo apt-get install ros-melodic-desktop-full
```

Dashing:

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

如果磁盘空间足够——至少要几个 **GB**——上述安装方案将是最好的选择。如果需要，有些命令可以对安装进行裁剪，包括 **ros-indigo-desktop** 和 **ros-indigo-ros-base**。这些安装会少装一些软件包，从而应对硬盘空间紧张的情况。

安装 **turtlesim** 功能包 本书中将多次使用一个简单的仿真器

turtlesim，它能够帮助我们理解 **ROS** 的工作机制。如果你希望运行书中的参考范例——毫无疑问——你需要安装 **turtlesim**。安装命令如下：

```
sudo apt-get install ros-indigo-turtlesim
```

Melodic:

```
sudo apt-get install ros-melodic-turtlesim
```

Dashing:

```
sudo apt-get install ros-dashing-turtlesim
```

为系统设置 **rosdep** 安装完 ROS 包后，需要执行下面的命令：

```
sudo rosdep init
```

这个初始化步骤是一次性的，一旦 ROS 正常工作，多数用户不再需要访问 **rosdep**。

正如其名字所示一样，这个命令主要是初始化**rosdep**，它是用于检查和安装软件包的依赖的一种工具，它与操作系统无关⁴。比如在Ubuntu中，**rosdep**是**apt-get**的一个前端。我们不会直接使用**rosdep**，但我们使用的工具将会在后台使用它。如果**rosdep**没有正常安装，这些工具可能会不“开心”。



在线文档中偶尔会提到一个叫做**rosinstall**的工具，使用它可以 从源码安装ROS系统^{5,6}。本书中我们需要的软件在Ubuntu deb软件源中都包含了，所以可以不用**rosinstall**。

2.2 配置账户

不管是安装 ROS 还是使用一台预装了 ROS 的电脑，要想在你的账户中使用 ROS，就必须完成下面两步重要的配置。为用户账户中设置 **rosdep** 首先，在你的账户中初始化 **rosdep** 系统，命令如下：

```
rosdep update
```

该命令将在你的根目录下保存一些文件，文件夹名为**.ros**。因为是初始化，这条命令只需要运行一次。

⁴ <http://wiki.ros.org/rosdep>

⁵ <http://wiki.ros.org/rosinstall>

⁶ <http://www.ros.org/doc/independent/api/rosinstall/html>



这里需要注意，不像上文的 `rosdep init`，`rosdep update` 命令是在你的普通账户下运行，而不使用超级用户前缀 `sudo`。

设置环境变量 ROS 要依据一些环境变量来定位文件。设置这些环境变量，你需要使用以下命令⁷执行 ROS 提供的脚本 `setup.bash`：

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

Melodic：

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

Dashing：

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

然后，用下行命令确认环境变量已经设置正确：

```
export | grep ROS
```

Melodic：

```
declare -x ROSLISP_PACKAGE_DIRECTORIES=""
declare -x ROS_DISTRO="melodic"
declare -x ROS_ETC_DIR="/opt/ros/melodic/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/opt/ros/melodic/share"
declare -x ROS_PYTHON_VERSION="2"
declare -x ROS_ROOT="/opt/ros/melodic/share/ros"
declare -x ROS_VERSION="1"
```

Dashing：

```
declare -x ROS_DISTRO="dashing"
declare -x ROS_PYTHON_VERSION="3"
declare -x ROS_VERSION="2"
```

如果一切工作正常，你应该看到了一组值（显示 `ROS_DISTRO` 和

`ROS_PACKAGE_PATH` 等环境变量的值）。如果 `setup.bash` 尚未运行，则此命令的输出通常为空。



如果从 ROS 指令中得到“`command not found`”错误（本章稍后将介绍），最可能的原因是，`setup.bash` 尚未在当前 shell 中运行。

但是请注意，上面列出的步骤仅适用于当前的 `shell`。每次启动一个新的 `shell`，且要在这个 `shell` 中运行 ROS 时，只要运行上述 `source` 命令，ROS 就能顺利工作。但这种操作在频繁使用 ROS 时非常烦人，又容易忘记，尤其是当你考虑到 ROS 系统的模块化设计往往需要同时在不同终端中执行不同的命令时。

⁷ <http://wiki.ros.org/rosbash>

因此，你可能想配置自己的账户，使其每启动一个新的 `shell` 时，都自动运行脚本 `setup.bash`。要做到这一点，编辑账户根目录中的文件 `.bashrc`，并在最下面添加前文的 `source` 命令。

```
# ROS 1.0 melodic or ROS 2.0 Dashing
echo Hello, ROS 1.0 or ROS 2.0? 1=Melodic 2=Dashing
read ROS
if (($ROS==1));then
source /opt/ros/melodic/setup.bash
# export
ROS_PACKAGE_PATH=/home/relaybot/RobTool/ROS1/Wiki/src:$ROS_PACKAGE_PATH
# export ROS_MASTER_URI=http://192.168.1.2:11311
# export ROS_IP=192.168.1.2
echo "Melodic"
elif (($ROS==2));then
source /opt/ros/dashing/setup.bash
echo "Dashing"
else
echo "Non-ROS"
fi
```

除了设置环境变量外，`setup.bash` 命令还能定义了一些 ROS 系统的 `bash` 函数，包括 `roscd` 和 `rosls`（下面将要介绍），这些函数在 `rosbash` 软件包⁸中定义。

2.3 使用 `turtlesim` 的小例子

在开始考察 ROS 工作的细节之前，我们先讲一个例子。这个快速练习有多个目的：首先确认 ROS 已正确安装，其次介绍经常出现在许多在线文档以及贯穿本书的 `turtlesim` 仿真器^{9[9]}，最后还能提供一个工作系统（虽然很简单），且我们将在本章剩余部分多次回顾这个系统。

启动 `turtlesim` 在三个不同的终端中，分别执行以下三个指令：

```
roscore
```

```
roslaunch turtlesim turtlesim_node
```

```
roslaunch turtlesim turtlesim_key
```

Dashing:

```
ros2 run turtlesim turtlesim_node
```

```
ros2 run turtlesim turtlesim_key
```

分三个终端的目的是让三个指令同时进行（ROS2为两个终端）。如果一切正常，你应该看到类似于图 2.1 的左边部分的图形窗口。这个窗口显示了一个模拟龟形机器人，它生活在方形“世界”。（每台机器海龟的外观可能会有所不同，事实上，每次运行时，仿真器从 ROS 所有发行版中一系列的“吉祥物”海龟中选择一个进行显示。），如果切换到第三个终端（执行命令为 `turtlesim_key`），然后按上、下、左、或右键，海龟就会响应你的命令开始移动了，并在屏幕上留下运动轨迹。

⁸ <http://wiki.ros.org/roslaunch>

⁹ <http://wiki.ros.org/turtlesim>

<https://index.ros.org/doc/ros2/Tutorials/Configuring-ROS2-Environment/>

<https://index.ros.org/doc/ros2/Tutorials/Turtlesim/Introducing-Turtlesim/>



如果按了键盘而海龟却没有响应，请先确定 `turtle_teleop_key` 窗口已经聚焦，比如可用鼠标点击该窗口中央。当仿真窗口可见时，你需要仔细地排序你的窗口，以保证 `turtle_teleop_key` 窗口聚焦。

让虚拟海龟画线本身或许并不能让你兴奋不已，然而，这个例子足以说明ROS系统中更有趣的主要思想*。



图 2.1 通过键盘控制海龟运动在屏幕留下的轨迹。

接下来保持这三个终端的运行状态，在以下各节的例子中将介绍其他方式来与这个系统交互。

* 例如，本书作者第一次在电脑屏幕上画龟是在 1987 年前后。

2.4 功能包/软件包（Packages）

在 ROS 中，所有软件都被组织为软件包的形式，称为 ROS 软件包或功能包，有时也简称为包。ROS 软件包是一组用于实现特定功能的相关文件的集合，包括可执行文件和其他支持文件。比如说，我们前面使用的两个可执行文件 `turtlesim_node` 和 `turtle_teleop_key` 都属于 `turtlesim` 包。



注意 ROS 软件包和操作系统软件包管理器中软件包的区别，比如 Ubuntu 系统使用的 `deb` 软件包。尽管概念相似，安装一个 `deb` 软件包可以添加一个或多个 ROS 软件包，因此两者并不等同。

毫不夸张地说，所有的 ROS 软件都是一个软件包或其他软件包的一部分。需要着重指出的是，这里包括了用户创建的新程序。我们将在 3.1 节中介绍如何创建新的软件包。在此期间，ROS 提供多个命令用于与已安装的软件包进行交互。

查看软件包列表和定位软件包 使用下行命令，可以获取所有已安装的 ROS 软件包列表清单^{10,11}：

```
rospack list
```

Dashing:

```
ros2 pkg list
```

在笔者的系统里，这条语句一共列出了 188 个软件包。

每个程序包由一个清单文件（文件名为 `package.xml`）定义。该文件定义关于包的一些细节，包括其名称、版本、维护者和依

¹⁰ <http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem>

¹¹ <http://wiki.ros.org/rospack>

赖关系。包含 `package.xml` 文件的目录被称为软件包目录（其实，这也是 ROS 软件包的定义：任何 ROS 能找到且包含 `package.xml` 文件的目录就是软件包目录。）。这个目录存储所在软件包的大部分文件。

一个非常重要的特例是，对于大部分功能包——确切地说，那些已经更新为使用新的 `catkin` 编译构建系统的功能包

——编译产生的可执行文件并未存放在功能包目录下，而是存放在一个单独的标准化目录层次结构中。对于使用 `apt-get` 安装的功能包，其所在根目录为 `/opt/ros/indigo`。可执行文件存储在这个根目录下的 `lib` 子目录里。同样，在自动生成的头文件存储在 `include` 子目录下。当需要时，ROS 通过搜索 `CMAKE_PREFIX_PATH` 环境变量列出的目录，这个环境变量由 `setup.bash` 自动设置。相比于 `fuerte` 和更老的版本，这种源码外编译是由 `catkin` 在 `groovy` 引入的主要变化之一。一般来说，这一切都在幕后进行，我们可以信任 ROS 能够找到它需要的文件。

要找到一个软件包的目录，使用 `rospack find` 命令：

```
rospack find package-name
```

Dashing:

```
ros2 pkg prefix package-name
```

当然，有时你并不知道（或不记得）感兴趣的软件包的完整名称，这时使用 `rospack` 命令非常方便，因为该命令支持 `tab` 命令补全。例如，你可以键入

```
rospack find turtle
```

Dashing:

```
ros2 pkg prefix turtle
```

并且，在按 **Enter** 键之前，按 **Tab** 键两次来查看所有已安装的 ROS 件包的列表中以 **turtle** 开头的软件包。

事实上，大多数 ROS 命令支持这种 **tab** 命令补全，不仅仅是查找软件包名，几乎在每一个需要的地方都能用得上它。在上面的命令中，你也可以使用 **tab** 来补全 **rospack** 命令以及其后的 **find** 关键字。



频繁使用 **tab** 补全功能可以大大减少需要记忆的内容，包括功能包、节点、话题、消息类型和服务的全名。计算机非常擅长存储和检索，因此我建议让你的电脑为你完成这类工作。

查看软件包：要查看软件包目录下的文件，使用如下命令：

```
rosls package-name
```

如果想“访问”某软件包目录，可以将当前目录切换至此软件包目录，使用如下命令：

```
roscd package-name
```

举一个简单的例子，假设你想查看 **turtlesim** 使用的一系列海龟图像。表 2.1 显示了如何使用 **rosls** 和 **roscd** 看这些图像的列表，并查看它们中的某一个。

读者可能会看到某些在线文档中出现功能包集（**stack**）的概念¹²。功能包集是紧密相关的功能包的集合。从 ROS 的

groovy 版本开始，功能包集的概念被逐步淘汰，取而代之的是元功能包（**metapackages**）^{13,14}。两者最大的区别是分

¹² <http://wiki.ros.org/rosbuild/Stacks>

¹³ http://wiki.ros.org/catkin/conceptual_overview

层结构“扁平化”：元功能包像其他功能包一样有功能包清单，但其目录下没有其他的功能包，而功能包集则将其包含的功能包存放在其目录下。新用户很少会直接与功能包集打交道。

```
1 $ rosls turtlesim
2 cmake
3 images
4 msg
5 package.xml
6 srv
7 $ rosls turtlesim /images
8 box-turtle . png
9 diamondback . png
10 electric . png
11 fuerte . png
12 groovy . png
13 hydro . png
14 hydro . svg
15 indigo . png
16 indigo . svg
17 palette . png
18 robot-turtle . png
19 sea-turtle . png
20 turtle . png
21 $ roscd turtlesim/images/
22 $ eog box-turtle . png
```

表 2.1 使用 `rosls` 和 `roscd` 命令查看 `turtlesim` 功能包使用的海龟图像。命令 `eog` 是“Eye of Gnome”图像查看器。

¹⁴ <http://wiki.ros.org/catkin/package.xml>

2.5 节点管理器（The Master）

补充：此部分ROS2出现较大变化。

至此我们已经介绍了文件的相关概念，以及它们是如何被组织成软件包的。接下来我们谈谈如何实际执行 ROS 软件。

ROS 的一个基本目标是使机器人专家设计的很多称为节点（**node**）的几乎相对独立的小程序能够同时运行。为此，这些节点必须能够彼此通信。ROS 中实现通信的关键部分就是 ROS 节点管理器。要启动节点管理器，使用如下命令：

```
roscore
```

在 `turtlesim` 的例子中我们已经使用过这个命令。这个命令非常简单易用：`roscore` 命令不带任何参数，也无需任何配置。

节点管理器应该在使用 ROS 的全部时间内持续运行。一个合理的工作流程是在一个终端启动 `roscore`，然后打开其他终端运行其他程序。除非你已经完成 ROS 的相关工作，否则一般没有理由终止 `roscore` 命令。当结束时，可以通过在 `roscore` 终端键入 `Ctrl-C` 停止节点管理器。

上文提到一般没有理由终止 `roscore`，但有时候重新启动 `roscore` 也许是个好主意，虽然这种情况不多。例如：切换到一组新的日志文件（见第 4 章）或清理参数服务器（见第 7 章）。



大多数 ROS 节点在启动时连接到节点管理器上，如果运行中连接中断，则不会尝试重新连接。因此，如果 `roscore` 被终止，当前运行的其他节点将无法建立新的连接，即使稍后重启 `roscore` 也无济于事。

这里的 `roscore` 命令用来显式启动 ROS 的节点管理器。在第六章中，我们将学习一个称为 `roslaunch` 的工具，其目的是一次性启动多个节点。这是一个自适应工具，如果启动多节点时没有节点管理器运行，它会自动启动节点管理器；如果已经有一个节点管理器在运行，则会使用已有的。

更新：ROS 2.0 使用 DDS 实现。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 Connex.

如果要安装 Connex DDS-Security 插件，请参考[此网页](#)。

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 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-Connex 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 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 Connex 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）。

为了使用 DDS / RTPS 实现与 ROS 2，ROS 中间接口“ROS Middleware 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：

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 Connex	commercial, research	rmw_connext_cpp	Full support. Support included in binaries, but Connex installed separately. 全力支持。支持包含在二进制文件中，但 Connex 单独安装
RTI Connex (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

2.6 节点（Nodes）

一旦启动roscore后，便可以运行ROS程序了。ROS程序的运行实例被称为节点（**node**）¹⁵。

这个定义中的“运行实例”（**running instance**）很重要。如果我们同时执行相同程序的多个副本——注意确保每个副本使用不同的节点名——则每个副本都被当做一个单独的节点。我们将在 2.8 节中看到这种差异。

在 `turtlesim` 的例子中，我们创建了两个节点。第一个节点是可执行文件 `turtlesim_node` 的实例化。这个节点负责创建 `turtlesim` 窗口和模拟海龟的运动。第二节点是可执行文件 `turtle_teleop_key` 的实例化。`teleop` 是 **teleoperation**（遥操作）的缩写，是指人通过在远程发送运动指令控制机器人。这个节点的作用是捕捉方向键被按下的事件，并将方向键的按键信息转换为运动指令，然后将命令发送到 `turtlesim_node` 节点。

启动节点 启动节点（也称运行ROS程序）的基本命令是`roslaunch`¹⁵：

```
roslaunch package-name executable-name
```

Dashing：

```
ros2 run package-name executable-name
```

可以看到，`roslaunch` 命令有两个参数，其中第一个参数是功能包的名称，这在 2.4 节中已经讨论过；第二个参数是该软件包中的可执行文件的名称。

¹⁵ <http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes>
<https://index.ros.org/doc/ros2/Tutorials/Understanding-ROS2-Nodes/>

¹⁶ <http://wiki.ros.org/rosbash#roslaunch>

`roslaunch` 没有什么“神奇”的：它只不过是一个 `shell` 脚本，能够理解 `ROS` 的文件组织结构，知道到哪里能找到与给定包名称对应的可执行文件。一旦它找到你要的程序，`roslaunch` 就会正常启动节点。例如，如果你真的想要，你可以像执行任何其他程序一样直接执行 `turtlesim_node`：

```
/opt/ros/indigo/lib/turtlesim/turtlesim_node
```

这里还要强调一点，通过节点管理器注册成为 `ROS` 节点发生在程序的内部，而不是通过 `roslaunch` 命令。

查看节点列表`ROS`提供了一些方法来获取任意时间运行节点的信息。要获得运行节点列表，使用如下命令¹⁷：

```
roslaunch list
```

Dashing:

```
ros2 node list
```

如果你在 2.3 节的命令后输入上行命令，你会看到三个节点的列表：

```
/ rosout
/ teleop_turtle
/ turtlesim
```

Dashing:

```
/teleop_turtle
/turtlesim
```

¹⁷ <http://wiki.ros.org/roslaunch>
<https://index.ros.org/doc/ros2/Tutorials/Understanding-ROS2-Nodes/>

关于这个列表以下三点需要引起注意。

> **rosout** 节点是一个特殊的节点，通过 **roscore** 自动启动。其作用有点类似于控制台程序中使用的标准输出（即 `std::cout`）。我们将在 4.4.2 节中更详细地介绍 **rosout**。

`/rosout` 前面的反斜杠“/”表明该节点名称属于全局命名空间。ROS 有一个丰富的命名节点和其他对象的体系（将在第 5 章中详细讨论），该体系使用命名空间组织各种资源¹⁸。

- > 其他两个节点应该相当明确：它们是从 2.3 节就开始介绍的海龟仿真器（**turtlesim**）程序和遥操作（**teleop_turtle**）程序。
- > 比较 **roslaunch** 的输出与 2.3 节 **roslaunch** 命令中可执行文件的名称，你会发现节点名并不一定与对应可执行文件名称相同。

事实上，可以使用 **roslaunch** 命令显式设置节点的名称，语法如下：

```
roslaunch package-name executable-name_name:=node-name
```

这种方法将使用 **node-name** 参数给出的名称覆盖节点的默认名。因为 ROS 中要求每个节点有不同的名称，因此该方法很重要（我们会在第 2.8 节用 **_name** 构建一个稍大的演示系统）。一般来讲，即使你经常用 **_name** 指定节点名称，你仍可能要使用一个启动文件（见第 6 章），而不是单独启动每个节点。

¹⁸ <http://wiki.ros.org/Names>

查看节点 要获得特定节点的信息，使用如下命令：

```
roscall node-name
```

Dashing:

```
ros2 node info node-name
```

这个命令的输出包括话题列表——节点是这些话题的发布者 (publisher) 或订阅者 (subscriber)，关于话题请参考 2.7.2 节；服务列表——这些服务是该节点提供的，关于服务请参考第 8 章；其 Linux 进程标识符 (process identifier, PID)；以及和与其他节点的所有连接。

终止节点 要终止节点，使用如下命令：

```
roscall node-name
```

不像终止和重启节点管理器那样，终止和重启节点通常不会对其他节点有较大影响；即使节点间正在相互交换消息 (message)，这些连接也会在节点终止时断开，在节点重启时重新连接。

补充：关于 ROS2 节点更多详细内容参考：

> <https://index.ros.org/doc/ros2/Tutorials/Node-arguments/>



还可以用 **Ctrl-C** 命令终止节点。但使用这种方法时可能不会在节点管理器中注销该节点，因此会导致已终止的节点仍然在 **roscall** 列表中。这虽然没有什么坏处，但可能会让用户对当前系统的行为感到困扰。此时可以使用下面的命令将节点从列表中删除：

```
roscall cleanup
```

2.7 话题和消息

显然，在我们之前建立的 **turtlesim** 例子中，遥控节点和仿真节点必须以某种方式进行对话。否则，在仿真节点中移动的海龟如何响应你在遥控节点中的按键操作呢？

ROS节点之间进行通信所利用的最重要的机制就是消息传递。在ROS中，消息有组织地存放在话题里¹⁹。消息传递的理念是：当一个节点想要分享信息时，它就会发布(**publish**)消息到对应的一个或者多个话题；当一个节点想要接收信息时，它就会订阅(**subscribe**)它所需要的一个或者多个话题。ROS节点管理器负责确保发布节点和订阅节点能找到对方；而且消息是直接地从发布节点传递到订阅节点，中间并不经过节点管理器转交。

2.7.1 查看节点构成的计算图

要查看节点之间的连接关系，恐怕将其表示为图形是最便于查看的。在 ROS 系统中查看节点之间的发布-订阅关系的最简单方式就是在终端输入如下命令：

```
rqt_graph
```

在这个命令中，r 代表 ROS，qt 指的是用来实现这个可视化程序的 Qt 图形界面（GUI）工具包。输入该命令之后，你将会看到一个图形界面，其中大部分区域用于展示当前系统中的节点。通常情况下，你将会看到如图 2.2 类似的图形界面。在该图中，椭圆形表示节点，有向边表示其两端节点间的发布-订阅关系。该计算图告诉我们，/teleop_turtle 节点向话题/turtle1/cmd_vel 发布消息，而/turtlesim 节点订阅了这些消息(“cmd_vel”是“command velocity”的缩写。)

¹⁹ [http:// wiki.ros.org/ROS/Tutorials/UnderstandingTopics](http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics)

<https://index.ros.org/doc/ros2/Tutorials/Topics/Understanding-ROS2-Topics/>

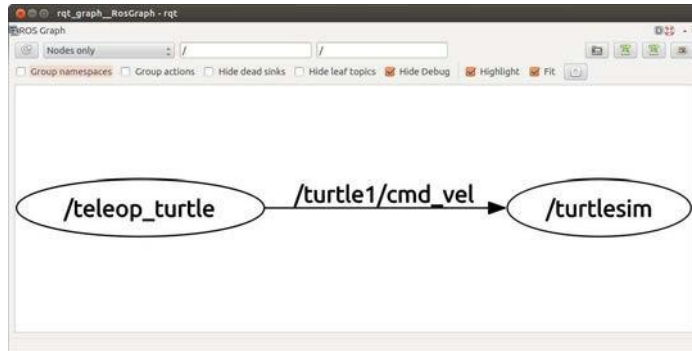


图 2.2 rqt_graph 界面，展示了 turtlesim 例子中的节点图结构。其中省略了调节点，包括 rosout 节点。

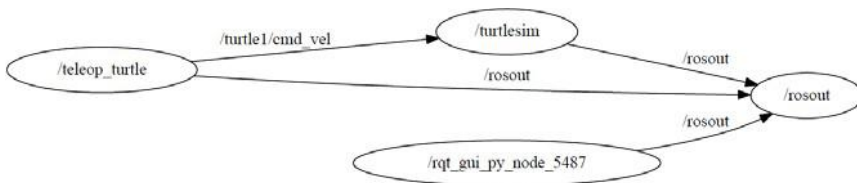
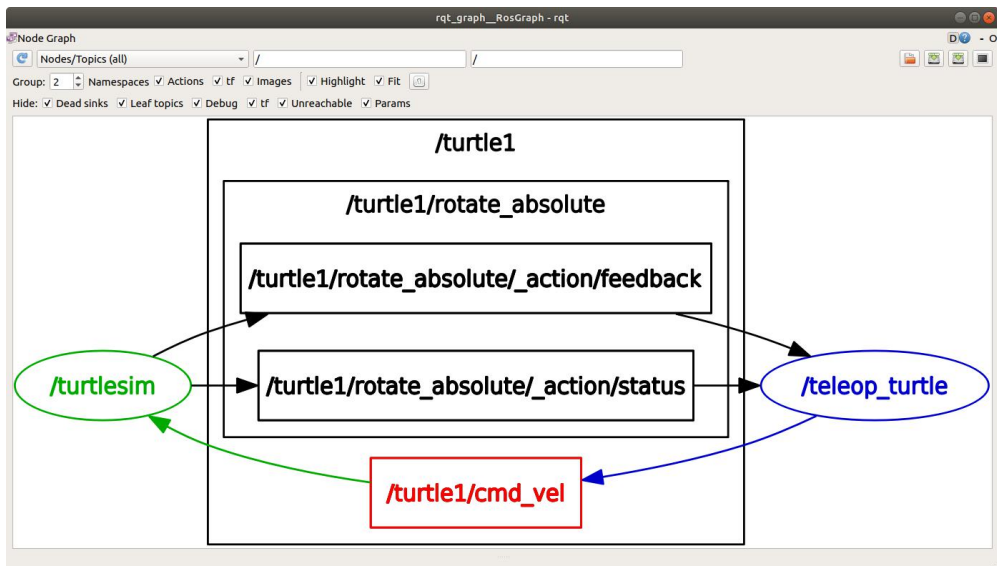


图 2.3 完整的 turtlesim 图结构，包括 rqt_graph 认为的调节点。



dashing

你可能会注意到，我们在 2.6 节看到的 rosout 节点并不在此图中。这是因为，在默认情况下，rqt_graph 隐藏了其认为只在调试过程中使用的节点。你可以通过取消“Hide debug”选项来禁止这个特性，图 2.3 展示了这个结果。

> 请注意 `rqt_graph` 本身就是一个节点。

所有的节点发布都向话题/`rosout` 发布消息，该话题由同名的/`rosout` 节点订阅。这个话题的作用是用来生成各个节点的文本日志消息。第 4 章将会给出更多关于 ROS 中日志的内容。



此处的名称/`rosout` 既指节点又指话题。但 ROS 并不会因这种重复的名字而混淆，因为 ROS 会根据上下文来推测我们讨论的是/`rosout` 节点还是/`rosout` 话题。

展示调试节点既有利又有弊：有利的是可以很直观地知道当前所有节点的状态；不利的是使计算图结构更加复杂，同时会引入一些冗余信息。

`rqt_graph` 还有其他一些选项来微调显示的计算图。作者的个人偏好是将下拉选项中的 **Nodes only** 改为 **Nodes/Topics(all)**，并取消 **Hide Debug** 以外所有的复选框。这种设置的好处在于能用矩形框显示所有的话题，以区别于节点的椭圆形表示，此时的计算图如图 2.4 中所示。从图中可见，除了订阅速度命令外，`turtlesim` 节点还发布了它的当前位姿和从仿真彩色传感器那里获得的数据。在查看一个新的 ROS 系统时，使用 `rqt_graph` 工具，尤其是按照上述进行设置，能帮助你发现自己的程序可以用哪些话题来和现有节点进行通信。



允许存在没有被订阅的话题看上去像是一个 **bug**，但是这种现象实际上很普遍，因为 ROS 节点通常设计成了只管发布它们有用的信息，而不需要担心是否有其他节点来订阅这些消息。这样有助于减少各个节点之间的耦合度。

现在，我们对 `turtlesim` 遥控系统的工作原理有了一定的理解。当你按下一个键时，`/teleop_turtle` 节点会以消息的形式将这些运动控制命令发布到话题 `/turtle1/cmd_vel`；

与此同时，因为 `turtlesim_node` 订阅了该话题，因此它会接收到这个消息，控制海龟按照该预定的速度移动。其中的关键点在于：

> 仿真海龟不关心（或者甚至不知道）哪个程序发布了这些 `cmd_vel` 消息。任何向这个话题发布了消息的程序都能控制这个海龟。

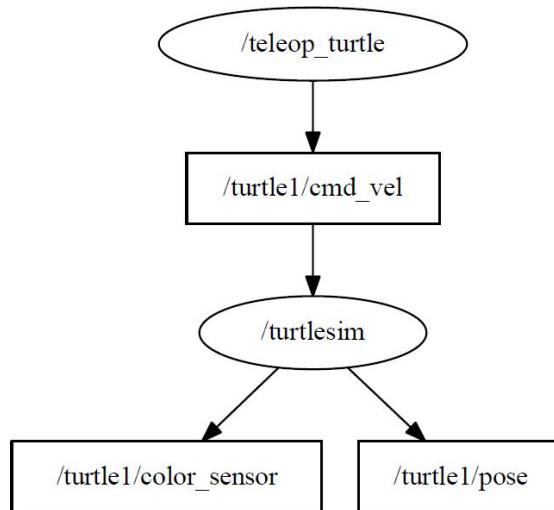


图 2.4 turtlesim 节点的计算图结构，展示了包括没有发布者或者订阅者的所有话题。

> 远程操作程序不关心（或者甚至不知道）哪个程序订阅了它发布的 `cmd_vel` 消息。任何订阅了相关话题的程序都能自主选择是否响应这些命令。

顺便指出，这些话题的名称中之所以包含 `/turtle1`，是因为创建这些话题都是为了其他节点和默认的仿真海龟进行消息传递，而这个默认的海龟正好叫做 `/turtle1`。在第 8 章，我们将学习如何在当前的仿真器中添加更多的海龟。

2.7.2 消息和消息类型

目前为止，我们已经了解了这些节点能相互传递消息，但这些消息里到底包含了什么信息，我们对此还是一无所知。下面，我们将深入探讨话题和消息。

话题列表 为了获取当前活跃的话题，使用如下命令 ²⁰:

```
rostopic list
```

Dashing:

```
ros2 topic list
```

在我们的例子中，这将列出如下五个话题：

```
/rosout  
/rosout_agg  
/turtle1/cmd_vel  
/turtle1/color_sensor  
/turtle1/pose
```

Dashing:

```
/parameter_events  
/rosout  
/turtle1/cmd_vel  
/turtle1/color_sensor  
/turtle1/pose
```

当然，这个列表列举的话题和 `rqt_graph` 中展示的话题应该是一样的，但是用文本的形式展示出来更方便。

打印消息内容 为了查看某个话题上发布的消息，可以利用

`rostopic` 命令的如下形式：

```
rostopic echo topic-name
```

Dashing:

```
ros2 topic echo topic-name
```

这条命令将会在终端里显示出指定话题里发布的任何消息。当键入下面的命令后

```
rostopic echo /turtle1/cmd_vel
```

Dashing:

```
ros2 topic echo /turtle1/cmd_vel
```

每当 `/teleop_turtle` 接收到按键信息时，将会显示类似表 2.2 的结果。输出中的每一个“---”线表示一个消息的结束以及另一个消息的起始。该例中，总共展示了三条消息。

```
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -2.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

²⁰ <http://wiki.ros.org/rostopic>
<https://index.ros.org/doc/ros2/>
<https://index.ros.org/doc/ros2/Tutorials/Topics/Understanding-ROS2-Topics/>

测量发布频率 有两个命令可以用来测量消息发布的频率以及这些消息所占用的带宽：

```
rostopic hz topic-name rostopic bw topic-name
```

Dashing:

```
ros2 topic hz topic-name
```

```
ros2 topic bw topic-name
```

这些命令订阅指定的话题，并且输出一些统计量，其中第一条命令输出每秒发布的消息数量，第二条命令输出每秒发布消息所占的字节量。



即使你一点都不关心这个特定的频率，但是这些命令对调试很有帮助，因为它们提供了一种简单的方法来验证这些消息确实有规律地在向这些特定的话题发布。

查看话题 利用 `rostopic info` 命令，你可以获取更多关于话题的信息：

```
rostopic info topic-name
```

Dashing:

```
ros2 topic info topic-name
```

例如，根据如下命令：

```
rostopic info /turtle1/color_sensor
```

你可以看到类似于这样的输出：

```
Type: turtlesim/Color Publishers:
*/turtlesim (http://donatello:46397/) Subscribers: None
Dashing:
Topic: /turtle1/color_sensor
Publisher count: 1
Subscriber count: 0
```

1	linear :
2	x:2.0
3	y:0.0
4	z:0.0
5	angular :
6	x:0.0
7	y:0.0
8	z:0.0
9	---
10	linear:
11	x:0.0
12	y:0.0
13	z:0.0
14	angular : 15
	x:0.0
16	y:0.0
17	z: -2.0
18	---
19	linear:
20	x:2.0
21	y:0.0
22	z:0.0
23	angular : 24
	x:0.0
25	y:0.0
26	z:0.0
27	---

表 2.2 rostopic echo 的典型输出。

输出中最重要的部分是第一行，给出了该话题的消息类型。因此，在 `/turtle1/color_sensor` 话题中发布订阅的消息类型是 `/turtlesim/Color`。Type 在文本输出中表示数据类型。理解消息的类型很重要，因为它决定了消息的内容。也就是说，一个话题的

消息类型能告诉你该话题中每个消息携带了哪些信息，以及这些信息是如何组织的。

查看消息类型 要想查看某种消息类型的详情，使用类似下面的命令^{21,22}：

```
rosmmsg show message-type-name
```

Dashing:

```
ros2 msg show message-type-name
```

让我们对上面用到的/turtle1/color_sensor消息类型尝试下这个命令：

```
rosmmsg show turtlesim/Color
```

Dashing:

```
ros2 msg show turtlesim/msg/Color
```

其输出为：

```
uint8 r
uint8g
uint8b
```

上述输出的格式是域（**field**）的列表，每行一个元素。每一个域由基本数据类型（例如 `int8`、`bool`、或者 `string`）以及域名称定义。上述输出告诉我们 `turtlesim/Color` 包含三个无符号 8 位整型变量 `r`，`g` 和 `b`。任何话题的消息只要是 `turtlesim/Color` 类型，都由上述三个域的值定义。（正如你猜测的那样，这些数字对应的是海龟中心下面像素的红-绿-蓝强度值。）

另一个例子是 `geometry_msgs/Twist`，书中将多次使用这种消息类型。该消息类型对应/turtle1/cmd_vel 话题，而且要稍微复杂一些：

²¹ <http://wiki.ros.org/rosmmsg>

²² <http://wiki.ros.org/msg>
[https://index.ros.org/doc/ros2/Tutorials/Defining-custom-interfaces-\(msg-srv\)/](https://index.ros.org/doc/ros2/Tutorials/Defining-custom-interfaces-(msg-srv)/)

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

Dashing:
geometry_msgs/msg/Twist
    Vector3 linear
    Vector3 angular
```

该例中，`linear` 和 `angular` 都是复合域，其数据类型是 `geometry_msgs/Vector3`。缩进格式表示命名为 `x`、`y` 和 `z` 的域是对应的上级两个域之一的成员。也就是说，`geometry_msgs/Twist` 消息包含六个成员，并且以两个向量的形式组织，分别为 `linear` 和 `angular`。其中每个数值都是基本数据类型 `float64`，即每个数值都是 64 位浮点型数据。

一般来说，一个复合域是由简单的一个或者多个子域组合而成，其中的每一个子域可能是另一个复合域或者独立的域，而且它们一般也都由基本数据类型组成。同样的思想也出现在 C++ 以及其他面向对象的编程语言中，即对象的数据成员可能是其他对象。

值得注意的是，上述复合域本身也可以作为消息类型。例如，一个具有消息类型 `geometry_msgs/Vecotr3` 的话题是完全符合语法的，该类型的消息包含三个顶层域，即 `x`、`y` 和 `z`。

这种嵌套组织方法有助于提高代码的复用率，尤其是在该系统中很多消息类型共享相同的数据类型。一个常见的例子是 `std_msgs/Header` 消息类型，其包含一些基本的序列号，时间戳以及坐标系等信息。这种类型将作为一个复合域（一般称作 `header`）出现在在上百个其他的消息类型中。

幸运的是，`rosmmsg show` 命令在显示消息类型时自动向下展开复合域直到基本类型为止，同时使用缩进的形式来展示这种嵌套结构，因此一般没有必要直接查看这些内层结构的消息类型。

消息类型同样可以包含固定或可变长度的数组（用中括号[]表示）和常量（一般用来解析其他非常量的域）。然而这些特性没有在 `turtlesim` 中使用。使用了这些特性的消息类型如 `sensor_msgs/NavSatFix`，其表示了单个 GPS 的定位数据。

用命令行发布消息 大多数时候，发布消息的工作是由特定的程序完成的*。但是，你会发现有时候手动发布消息是很实用的。要实现该功能，利用 `rostopic` 命令行工具²³：

```
rostopic pub -r rate-in-hz topic-name message-type message-content
```

Dashing:

```
ros2 topic pub -r rate-in-hz topic-name message-type message-content
```

这条命令重复地按照指定的频率给指定的话题发布指定的消息。

该命令最后的参数 `message-content` 应该按顺序提供消息类型中所有域的参数值。例如：

```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[2,0,0]' '[0,0,0]'
```

Dashing:

```
ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist '{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}'
```

这些数值按照 `rosmmsg show` 命令显示的变量顺序赋给了消息中的域变量。在此例中，前面三个数字表示期望的位移线速度，后面三个数字表示期望的角速度。用单引号（'...'）和中括号（[...]）组织这些数值赋给它们对应的两个顶层复合域变量。正如你所猜测

* 实际上，编写这些程序是本书最基本的问题！

²³ <http://wiki.ros.org/rostopic>

的，本例中生成的消息将会控制海龟沿直线前进（沿着它的 x 轴），而没有转动。

同样的，类似的命令将会控制小海龟沿着它的 z 轴（垂直于电脑屏幕）旋转。

```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[0,0,0]' '[0,0,1]'
```

Dashing:

```
ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist '{linear: {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.0}}'
```



事实上，上述两个例子中的非零域——位移 **linear.x** 以及角度 **angular.z**——是 **turtlesim** 重点关注的两个变量，而这两个变量是 **geometry_msgs/Twist** 消息中仅有的两个变量。因为其他四个变量表示的运动对于该二维仿真来说是不可能发生的，所以 **turtlesim** 程序忽略了这些变量。

上述语法有一个明显的缺点，那就是你必须记住消息类型里所有的域以及这些域出现的顺序。另一种替代方式是以YAML 字典的形式给出一个参数，该参数将所有域进行赋值（注：YAML是"YAML Ain't a Markup Language"（YAML不是一种置标语言）的递归缩写）。下面这条命令（实际上包含了换行符）和上述命令是等价的，只不过它显式地指明了结构域中变量名和对应值的映射关系：

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
  x:2.0
  y:0.0
  z:0.0
Angular:
  x:0.0
  y:0.0
  z:0.0"
```

在**bash**和**YAML**之间交互还有很多小技巧，在线文档有一整页来介绍在命令行中如何使用**YAML**²⁴²⁵。利用**Tab**键补齐输入是保证语法正确的最简单方法。在输入消息类型之后按下**Tab**键，将会插入一个完整的**YAML**字典，其中包含给定消息类型中所有的结构域变量。**Tab**键产生的消息将会使用默认值（**zero**、**false**、**empty string**等等），但是你可以根据需要修改消息内容。

下面还有一些关于 **rostopic pub** 可能会用到的附加选项。

> 此处展示的命令利用了 **-r** 来指定话题以频率模式发布消息 即以一定的时间周期发布消息。这条命令同样支持一次性发布的模式（**-l**“虚线后为数字 1”）和特别的锁存模式（**-l**“虚线后为字母 L 的小写”），锁存模式虽然也只是发布一次消息，但是会确保该话题的新订阅者也会收到消息。实际上，锁存模式是默认的模式。

> 同样也可以从文件中读取消息（利用 **-f** 参数）或者从标准的输入（把 **-f** 参数和消息的内容从命令中都删掉）中读取。两种情况下，输入应该符合 **rostopic echo** 的输出格式。



你很有可能已经开始设想，为了自动测试你的程序，编写一种脚本将 **rostopic echo** 和 **rostopic pub** 结合起来作为“记录”和“回放”消息的方式。如果是这样的话，你会对 **rosbag** 工具感兴趣（第 9 章），该工具是上述想法的更加完善的实现。

²⁴ <http://wiki.ros.org/YAMLOverview>

²⁵ <http://wiki.ros.org/ROS/YAMLCommandLine>

理解消息类型的命名和 ROS 里其他的程序一样，每条消息类型都属于一个特定的包。消息类型名总会包含一个斜杠，斜杠前面的名字是包含它的包：

`package-name/type-name`

例如，`turtlesim/Color` 消息类型按如下方式分解：

<code>turtlesim</code>	+	<code>Color</code>	<code>turtlesim/Color</code>
功能包名		类型名称	消息类型

这种分解消息类型名的方法有如下几个目的：

- 最直接地，把包的名字包含在消息类型名里能避免命名冲突。例如，`geometry_msgs/Pose` 和 `turtlesim/Pose` 是有区别的消息类型，它们包含了不同的（但概念上是类似的）数据。
- 正如我们将在第 3 章看到的那样，当我们编写 ROS 程序的时候，如果也用到了其他包的消息类型，那么我们需要声明对它们的依赖关系。把功能包的名称和消息类型名一起写出来会使得这些依赖关系看上去更明朗。
- 最后一点，包名和其含有的消息类型放在一起将有助于猜测它的含义。例如，消息类型 `ModelState` 单独出现可能会让人产生迷惑，但是以 `gazebo/ModelState` 的形式出现后，就会指明这个消息类型是 Gazebo 仿真器中的一部分，而且很有可能包含了这个仿真器中某个模型的状态信息。

2.8 一个更复杂的例子

本章到目前为止，我们已经学会如何启动 ROS 主节点，如何启动 ROS 节点，以及如何查看节点之间通信的话题。本节综合我们前面介绍的知识来学习一个比之前 2.3 小节更复杂的例子，主要侧重于更详细地展示话题和消息的工作原理。

在开始之前，停止目前所有有可能在运行的节点。如果节点管理器现在不处于激活状态的话，运行 `roscore`。然后，在四个独立的终端中运行如下四条命令：

```
roslaunch turtlesim turtlesim_node_name:=A roslaunch turtlesim
turtlesim_node_name:=B roslaunch turtlesim turtle_teleop_key_
name:=C roslaunch turtlesim turtle_teleop_key_name:=D
```

Dashing:

```
ros2 run turtlesim turtlesim_node --ros-args --remap
__node:=my_turtle
```

这些命令将会运行两个 `turtlesim` 仿真器的实例——将会出现两个独立的窗口——以及两个 `turtlesim` 遥控节点的实例。

上述唯一可能不熟悉的是 `roslaunch` 命令中的 `_name` 参数。这些参数覆盖了每个节点赋予自己的默认名称。覆盖是必要的，因为 ROS 节点管理器不允许多个节点拥有相同的名称。

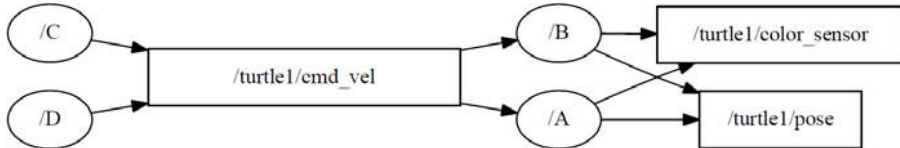


图 2.5 一个略为复杂的 ROS 图结构，其中包括两个 `turtlesim` 节点 A 和 B 以及两个遥控节点 C 和 D。

如果试图运行两个同名的节点，新节点将会正常运行，而之前的节点将会终止并显示如下信息：

```
[WARN] [1369835799.391679597]: Shutdown request received.
```

```
[WARN] [1369835799.391880002]: Reason given for shutdown:  
[new node registered with same name]
```

即使我们正在试图避免这种问题，但通常来讲，这种举措是很有用的。尤其是在你调试和修改节点的时候，因为这种机制会保证你不会错误地运行多个版本的同名节点。

注意：在ROS 2.0中，试图运行两个同名的节点，并不会报错，并且都可以正常运行。

在我们讨论上述例子前，你可能希望思考一会儿这个系统将会如何运转。用 `rqt_graph` 工具显示这个系统的图结构会是什么样子的呢？两个海龟和两个遥控节点之间的响应关系是怎样的呢？

希望你预测的图结构和图 2.5 类似，并且当遥控节点之一发送按键命令时，两个海龟都会有同样的运动来响应。让我们看看为什么。

2.8.1 话题通信的多对多机制

或许你以为每个远程操作节点分别连接了一个小海龟节点，因此生成了两个独立可控的仿真^{*}。但是，注意这两种节点分别发布和订阅了话题 `/turtle1/cmd_vel`。所以，不管哪个节点发布了该话题 `/turtle1/cmd_vel` 的消息，这些消息将会传送给每个订阅了该话题的节点。

在本例中，远程操作节点 C 发布的每条消息都会传送给 A 和 B 两个仿真节点。同样的，D 节点发布的消息也会传送给 A 和 B。当这些消息到达仿真节点时，海龟将会相应地移动，而不管这条消息是哪个节点发布的。此处要强调的是，基于话题和消息的通信机制是多对多的，即多个发布者和多个订阅者可以共享同一个话题。

^{*} 在第 6 章，我们将会学习正确地创造这些并行且独立的 `turtlesim` 仿真的方法。

2.8.2 节点之间的松耦合关系

毫无疑问，你已经注意到了我们不需要对 `turtlesim` 仿真器进行重新编程来接收多个源节点生成的运动命令；或者远程操作节点也不需要重新设计来一次性控制多个仿真实例。实际上，这个例子很容易扩展到上述两种类型之一的任意多个节点^{*}。

另一个极端的例子是，考虑只运行 `turtlesim` 仿真器，而没有其他节点。在这种情况下，仿真器将会等待 `/turtle1/cmd_vel` 话题类型的消息，显然，这个话题没有任何的发布者。

出现上述现象的根本原因在于，我们的 `turtlesim` 节点之间——更一般的，对于绝大多数设计精巧的 ROS 节点——是松耦合的。每个节点都不需要显式知道其他节点的存在与否；它们的唯一交互方式是间接地发生在基于话题和消息的通信层。这种节点之间的独立性，以及其支持的任务可分解特性（即复杂任务分解成可重用的小模块），是 ROS 最关键的设计特性之一。

> “生产”消息的程序（例如 `turtle_teleop_key`）只管发布该消息，而不用关心该消息是如何被“消费”的。

> “消费”消息的程序（例如 `turtlesim_node`）只管订阅该话题或者它所需要消息的所有话题，而不用关心这些消息数据是如何“生产”的。

此外，ROS 为更加直接的一对一通信提供了一种称为服务（`services`）的机制。第二种通信机制更为少见，但是也有其应用价值。第 8 章将会描述如何生成和使用服务。

补充，行动 `action`，主题、服务和行动。

> <https://index.ros.org/doc/ros2/Tutorials/Actions/>

> <https://index.ros.org/doc/ros2/Tutorials/Understanding-ROS2-Actions/>

^{*}当然是在合理范围内。例如，当激活了大约 100 个 `turtlesim_node` 仿真实例后，作者的电脑开始不能负担如此多的 X client 了。

2.9 问题检查

当ROS没有按你的预期运行时，一种可能有帮助的工具，也是本章要学习的最后一个命令行工具，是`roswtf`^{26,27}，该命令可以不带参数运行：

`roswtf`

这条命令会进行全面而深入的检测，包括检测你的环境变量、安装的文件以及运行的节点。例如，`roswtf`将会检测在安装过程中 `rosdep` 初始化是否完成，任何节点是否出现了意外的挂起或者终止，以及活跃的节点是否正确地和其他节点相连接等。可惜的是，由 `roswtf` 检测的完整列表只能在 Python 源码中才能找到。

Eloquent:

`ros2doctor`

> <https://index.ros.org/doc/ros2/Tutorials/Introspection-with-command-line-tools/>

> <https://index.ros.org/doc/ros2/Tutorials/Getting-Started-With-Ros2doctor/>

当ROS 2安装程序未按预期运行时，您可以使用该`ros2doctor`工具检查其设置。`ros2doctor` 检查ROS 2的所有方面，包括平台，版本，网络，环境，正在运行的系统等，并警告您可能的错误和问题原因。

2.10 展望

本章的目的是介绍一些 ROS 中基本的对象，例如节点、消息、话题以及一些用来和这些对象交互的命令行工具。在下一章中，我们将在与现有程序交互上更进一步，开始学习编写新的程序。

*此工具的名字在本文档里没有解释，但是作者相当确信这是“Why The Failure?”的缩写。

²⁶ <http://wiki.ros.org/roswtf>

²⁷ <http://wiki.ros.org/ROS/Tutorials/Gettingstartedwithroswtf>
<https://index.ros.org/doc/ros2/Tutorials/Getting-Started-With-Ros2doctor/>

第3章 编写ROS程序[†]

知之者不如好之者,好之者不如乐之者。
——孔子

本章中，学习编写能够发布和订阅消息的 ROS 程序。

到目前为止，我们已经介绍了一部分核心的 ROS 特性，包括功能包、节点、话题和消息等。我们也花了一点时间探索建立在这些特性基础上的一些现有软件。现在，终于是时候开始创建你自己的 ROS 程序了。本章将介绍如何建立一个开发工作区，并且在此工作区编写三个短程序，包括标准的”hello world”例程和另外两个展示如何发布和订阅消息的程序。

补充：学习此部分之前，最好先大致浏览一下如下链接：

开发人员（ROS 2 Dashing or Eloquent）

- 创建工作区ws
- 创建一个ROS 2包
- 编写简单的发布者和订阅者（C++）
- 编写简单的发布者和订阅者（Python）
- 编写简单的服务端和客户端（C++）
- 编写简单的服务端和客户端（Python）
- ros2doctor入门

3.1 创建工作区和功能包

我们在 2.4 节中看到，所有的 ROS 软件，包括我们创建的软件，都被组织成功能包。在我们写任何程序之前，第一步是创建一个容纳我们的功能包的工作区，然后再创建功能包本身。

创建工作区 我们所创建的包，应该全部放到一个叫做工作区的目录中¹。例如，作者的工作区是路径/home/jokane-/ros，但你可以用任何你喜欢的名字命名你的工作区，并且把目录存储在你账号中的任何位置。现在请使用标准的mkdir命令行去创建一个目录，后文中我们将把这个新的目录称作你的工作区目录。

[†] 本章由肖军浩、贾凡翻译。

¹ http://wiki.ros.org/catkin/Tutorials/create_a_workspace

<https://index.ros.org/doc/ros2/Tutorials/Workspace/Creating-A-Workspace/>

对于许多用户来说，确实没有必要使用多个 **ROS** 工作区。但是，**ROS** 的 **catkin** 编译系统（我们将在 3.2.2 节中介绍），试图一次性编译同一个工作区中的所有功能包。因此，如果你的工作涉及大量的功能包，或者涉及几个相互独立的项目，则维护数个独立的工作区可能是有帮助的。

创建工作区还需要最后一步，即在工作区目录中创建一个叫做 **src** 的子目录。你可能已经猜到了，这个子目录将用于存放功能包的源代码。

创建功能包 创建一个新**ROS**功能包的命令应该在你工作区中的

src目录下运行，如下所示²：

```
catkin_create_pkg package-name
```

Dashing:

CMake:

```
ros2 pkg create --build-type ament_cmake <package_name>
```

Python:

```
ros2 pkg create --build-type ament_python <package_name>
```

其实，这个功能包创建命令没有做太多工作，它只不过创建了一个存放这个功能包的目录，并在那个目录下生成了两个配置文件。

- 第一个配置文件，叫做 **package.xml**，是我们在 2.4 节讨论过的清单文件。
- 第二个文件，叫做 **CMakeLists.txt**，是一个 **Cmake** 的脚本文件，**Cmake** 是一个符合工业标准的跨平台编译系统。这个文件包含了一系列的编译指令，包括应该生成哪种可执行文件，需要哪些源文件，以及在哪里可以找到所需的头文件和链接库。当然，这个文件表明 **catkin** 在内部使用了 **Cmake**。

在接下来的小节中，你将会对上述两个文件作一些修改来配置你的新包。目前，只要知道 `catkin_create_pkg` 并没有做任何神奇的事情就足够了。它的工作不过是创建两个包目录和这两个配置文件的默认版本，从而使用户的操作更加方便一点。



这个三层的目录结构（一个工作区目录，包含了一个 `src` 目录，其中又包含了一个功能包目录）对于简单项目和小工作区似乎有些大材小用，但是 `catkin` 编译系统需要它。

ROS 包的命名遵循一个命名规范，只允许使用小写字母、数字和下划线，而且首字符必须是一个小写字母。一些 ROS 工具，包括 `catkin`，不支持那些不遵循此命名规范的包。

本书中所有的例程都属于 `agitr` 包，此包由书名的首字母（译者注：英文版名称为 `agenteintroductiontoROS`）组成。如果你想自己重新创建这个包，你可以在工作区 `src` 目录中运行如下命令：

```
catkin_create_pkg agitr
```

另一种方法来创建自己的 `agitr` 包是从本书的网站上下载这个包的压缩包，并且将其解压至你的工作区目录中。

编辑清单文件 创建包后，你可能希望编辑其 `package.xml` 文件，其中包含一些描述这个包的元数据。通过 `catkin_create_pkg` 创建的功能包默认含有很丰富的注释，对于读者在很大程度上是不言自明的。然而，请注意，无论是在编译时还是在运行时，其中的大部分信息 ROS 并没有使用，这些信息只有在你公开发布代码时才变得重要。本着保持文档与实际功能同步的精神，至少填写 `description` 和 `maintainer` 两部分可能是比较合理的。表 3.1 展示了我们 `agitr` 包的清单文件内容。

Dashing:

`ros2 pkg create --build-type ament_cmake agitr`

创建agitr自定义ROS2包

`ros2 pkg create --build-type ament_cmake --node-name hello agitr`

创建带有节点hello的agitr自定义ROS2包

1. going to create a new package
2. package name: agitr
3. destination directory: /home/relaybot/RobTool/ROS2/dev_ws/src
4. package format: 3
5. version: 0.0.0
6. description: TODO: Package description
7. maintainer: ['relaybot <zhangrelay@sohu.com>']
8. licenses: ['TODO: License declaration']
9. build type: ament_cmake
10. dependencies: []
11. cpp_node_name: hello
12. creating folder ./agitr
13. creating ./agitr/package.xml
14. creating source and include folder
15. creating folder ./agitr/src
16. creating folder ./agitr/include/agitr
17. creating ./agitr/CMakeLists.txt
18. creating ./agitr/src/hello.cpp

² <http://wiki.ros.org/ROS/Tutorials/CreatingPackage>

<https://index.ros.org/doc/ros2/Tutorials/Creating-A-ROS2-Package/>

3.2 你好，ROS!

现在我们已经创建了包，可以开始编写 ROS 程序了。

3.2.1 一个简单的程序

表 3.2 显示的是学习各种编程语言中经常遇到的“Hello,world!”规范程序的 ROS 版本。这个名为 `hello.cpp` 的源文件也存放在你的功能包文件夹中，挨着 `package.xml` 和 `CMakeLists.txt`。



一些在线教程建议在你的功能包目录中创建 `src` 目录来存放 C++ 源文件。这个附加的组织结构可能是有益的，特别是对于那些含有多种类型文件的较大的功能包，但它不是严格必要的。

```
1 <?xml version="1.0"?>
2 <package>
3     <name>agir </name>
4     <version>0.0.1 </version>
5     <description>
6         Examples from A Gentle Introduction to ROS
7     </description>
8     <maintainer email="jokane@cse.sc.edu">
9         Jason O' Kane
10    </maintainer>
11    <license>TODO</license>
12    <buildtool_depend>catkin </buildtool_depend>
13    <build_depend>geometry_msgs</build_depend>
14    <run_depend>geometry_msgs</run_depend>
15    <build_depend>turtlesim </build_depend>
16    <run_depend>turtlesim </run_depend>
17 </package>
```

表 3.1 本书 `agir` 包的清单文件（`package.xml`）。

Dashing (turtlesim/package.xml)

```
1  <?xml version="1.0"?>
2  <?xml-model href="http://download.ros.org/schema/package_format3.xsd"
    schematypens="http://www.w3.org/2001/XMLSchema"?>
3  <package format="3">
4  <name>turtlesim</name>
5  <version>1.0.2</version>
6  <description>
7  turtlesim is a tool made for teaching ROS and ROS packages.
8  </description>
9  <maintainer email="dthomas@osrfoundation.org">Dirk Thomas</maintainer>
10 <license>BSD</license>

11 <url type="website">http://www.ros.org/wiki/turtlesim</url>
12 <url type="bugtracker">https://github.com/ros/ros_tutorials/issues</url>
13 <url type="repository">https://github.com/ros/ros_tutorials</url>
14 <author>Josh Faust</author>

15 <build_depend>qt5-qmake</build_depend>
16 <build_depend>qtbase5-dev</build_depend>

17 <buildtool_depend>ament_cmake</buildtool_depend>
18 <buildtool_depend>roscpp_default_generators</buildtool_depend>

19 <exec_depend>libqt5-core</exec_depend>
20 <exec_depend>libqt5-gui</exec_depend>
21 <exec_depend>roscpp_default_runtime</exec_depend>

22 <depend>ament_index_cpp</depend>
23 <depend>geometry_msgs</depend>
24 <depend>roscpp</depend>
25 <depend>roscpp_action</depend>
26 <depend>std_msgs</depend>
27 <depend>std_srvs</depend>

28 <member_of_group>roscpp_interface_packages</member_of_group>

29 <export>
30 <build_type>ament_cmake</build_type>
31 </export>
32 </package>
```

注意对比，ROS1和ROS2的package.xml风格差异。

我们很快就介绍如何编译和运行这个程序，但在这之前让我们先来查看下代码。

头文件 `ros/ros.h` 包含了标准 ROS 类的声明，你将会在每一个你写的 ROS 程序中包含它。

```
1//This is a ROS version of the standard "hello, world"
2// program.
3
4// This header defines the standard ROS classes.
5#include <ros / ros.h>
6
7int main ( int argc , char ** argv ) {
8    // Initialize the ROS system .
9    ros::init( argc , argv , "hello_ros" );
10
11    // Establish this program as a ROS node .
12    ros::NodeHandle nh;
13
14    // Send some output as a log message .
15    ROS_INFO_STREAM( "Hello, ROS!" );
16 }
```

表 3.2 一个非常简单的 ROS 程序，文件名为 `hello.cpp`。

> `ros::init` 函数初始化 ROS 客户端库。请在你程序的起始处调用一次该函数³。函数最后的参数是一个包含节点默认名的字符串。

这个默认名可以通过启动文件或者通过 `roslaunch` 命令行参数覆盖。

³ <http://wiki.ros.org/roscpp/Overview/InitializationandShutdown>

> `ros::NodeHandle`（节点句柄）对象是你的程序用于和ROS系统交互的主要机制⁴。创建此对象会将你的程序注册为ROS节点管理器的节点。最简单的方法就是在整个程序中只创建一个`NodeHandle`对象。

在内部，`NodeHandle`类维护一个引用计数，仅仅在第一个`NodeHandle`对象创建时才会在节点管理器注册新的节点。同样，只有当所有的`NodeHandle`对象都销毁后，节点才会注销。这一细节有两个影响：首先，如果愿意，你可以为一个节点创建很多`NodeHandle`对象。这样做只有在一些特殊的应用中才有意义。本书就有这样的一个例子。其次，这意味着，在一个程序中使用标准的`roscpp`接口来运行多个节点是不可能的。

> `ROS_INFO_STREAM`宏将生成一条消息，且这一消息被发送到不同的位置，包括控制台窗口。在第4章我们可以看到关于此类日志消息的更多细节。

Dashing:

```
1  #include <cstdio>

2  #include <chrono>
3  #include <memory>

4  #include "rclcpp/rclcpp.hpp"

5  using namespace std::chrono_literals;

6  class MinimalInfo : public rclcpp::Node
7  {
8  public:
9      MinimalInfo()
10     : Node("minimal_info", count_(0))
11     {
12         auto timer_callback =
13             [this]() -> void {
14                 RCLCPP_INFO(this->get_logger(), "Hello ROS2, 你好 ROS2.");
15             };
16         timer_ = this->create_wall_timer(500ms, timer_callback);
17     }

18 private:
19     rclcpp::TimerBase::SharedPtr timer_;
20     size_t count_;
```

```
21  };

22  int main(int argc, char ** argv)
23  {
24      (void) argc;
25      (void) argv;
26      printf("hello world agitr package\n");
27      rclcpp::init(argc, argv);
28      rclcpp::spin(std::make_shared<MinimalInfo>());
29      rclcpp::shutdown();
30      return 0;
31  }
```

3.2.2 编译 Hello 程序

我们该如何编译和运行这个程序呢？这些交给ROS的catkin 编译系统来处理。一共有四个步骤⁵。

声明依赖库 首先，我们需要声明程序所依赖的其他功能包。对于c++程序而言，此步骤是必要的，以确保catkin能够向c++编译

⁴ <http://wiki.ros.org/roscpp/Overview/NodeHandles>

⁵ <http://wiki.ros.org/ROS/Tutorials/BuildingPackages>

器提供合适的标记来定位编译功能包所需的头文件和链接库。

为了给出依赖库，编辑包目录下的 `CMakeLists.txt` 文件。该文件的默认版本含有如下行：

```
find_package(catkin REQUIRED)
```

Dashing:

```
find_package(ament_cmake REQUIRED)
```

所依赖的其他 `catkin` 包可以添加到这一行的 `COMPONENTS` 关键字后面，如下所示：

```
find_package(catkin REQUIRED COMPONENTS package-names)
```

Dashing:

```
find_package(<dependency> REQUIRED)
```

对于 `hello` 例程，我们需要添加名为 `roscpp` 的依赖库，它提供了 ROS 的 C++ 客户端库。因此，修改后的 `find_package` 行如下所示：

```
find_package(catkin REQUIRED COMPONENTS roscpp)
```

Dashing:

```
find_package(rclcpp REQUIRED)
```

我们同样需要在包的清单文件中列出依赖库，通过使用 `build_depend`（编译依赖）和 `run_depend`（运行依赖）两个关键字实现：

```
<build_depend>package-name</build_depend>
```

```
<run_depend>package-name</run_depend>
```

Dashing:

```
<buildtool_depend>package-name</buildtool_depend>
```

在我们的例程中，`hello` 程序在编译时和运行时都需要 `roscpp` 库，因此清单文件需要包括：

```
<build_depend>roscpp</build_depend>
```

```
<run_depend>roscpp</run_depend>
```

Dashing:

```
<buildtool_depend>roscpp</buildtool_depend>
```

然而，在清单文件中声明的依赖库并没有在编译过程中用到；如果你在此处忽略它们，你可能不会看到任何错误消息，直到发布你的包给其他人，他们可能在没有安装所需包的情况下编译你发布的包而导致报错。

声明可执行文件 接下来，我们需要在 `CMakeLists.txt` 中添加两行，来声明我们需要创建的可执行文件。其一般形式是：

```
add_executable(executable-name source-files)
target_link_libraries(executable-name ${catkin_LIBRARIES})
```

第一行声明了我们想要的可执行文件的文件名，以及生成此可执行文件所需的源文件列表。如果你有多个源文件，把它们列在此处，并用空格将其区分开。第二行告诉 `Cmake` 当链接此可执行文件时需要链接哪些库（在上面的 `find_package` 中定义）。如果你的包中包括多个可执行文件，为每一个可执行文件复制和修改上述两行代码。

在我们的例程中，我们需要一个名为 `hello` 的可执行文件，它通过名为 `hello.cpp` 的源文件编译而来。所以我们需要添加如下几行代码到 `CMakeLists.txt` 中：

```
add_executable(hello hello.cpp)
target_link_libraries(hello ${catkin_LIBRARIES})
```

Dashing:

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

ament_target_dependencies(hello roscpp)

target_include_directories(hello PUBLIC

    ${CMAKE_CURRENT_SOURCE_DIR}/include

    ${CMAKE_CURRENT_BINARY_DIR}/include)

install(DIRECTORY include DESTINATION include)
```

为了便于参考，表 3.3 展示了一个足以满足我们例程需求的简短的CMakeLists.txt。

通过catkin_create_pkg创建的CMakeLists.txt的默认版本包含一些注释掉的提示，用于实现其他功能；对于大多数简单的程序，类似于此处展示的简单版本已经足够了。

编译工作区 一旦你的CMakeLists.txt文件设置好，你就可以编译你的工作区，使用如下命令来编译所有包中的所有可执行文件：

```
catkin_make
```

Dashing:

```
colcon build
```

因为被设计成编译你的工作区中的所有包，这个命令必须从你的工作区目录运行。它将会完成一些配置步骤（尤其是你第一次运行此命令时），并且在你的工作区中创建 **devel** 和 **build** 两个子目录。这两个新目录用于存放和编译相关的文件，例如自动生成的编译脚本、目标代码和可执行文件。如果你喜欢，当完成功能包的相关工作后（译者注：即完成了编写、调试、测试等一系列工作后，此时代码基本定型），可以放心地删除 **devel** 和 **build** 两个子目录。

如果有编译错误，你会在执行此步骤时看见它们。在更正它们以后，你可以重新运行 **catkin_make** 来完成编译工作。



如果你看到来自 `catkin_make` 的错误，提示无法找到 `ros/ros.h` 头文件，或者 `ros::init` 等 ROS 函数未定义的错误，最大的可能性是你的 `CMakeLists.txt` 没有正确声明对 `roscpp` 的依赖。

Sourcing `setup.bash` 最后的步骤是执行名为 `setup.bash` 的脚本文件，它是 `catkin_make` 在你工作区的 `devel` 子目录下生成的。

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

Dashing:

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

这个自动生成的脚本文件设置了若干环境变量，从而使 ROS 能够找到你创建的功能包和新生成的可执行文件。它类似于 2.2 节介绍的全局 `setup.bash`，但是是专门为你的工作区量身定做的。除非目录结构发生变化，否则你只需要在每个终端执行此命令一次，即使你修改了代码并且用 `catkin_make` 执行了重编译。

```
1 # What version of CMake is needed ?
2 cmake_minimum_required(VERSION 2.8.3)
3
4 # Name of this package .
5 project ( agitr )
6
7 # Find the catkin build system , and any other packages on
8 # which we depend .
9 find_package ( catkin REQUIRED COMPONENTS roscpp )
10
11 # Declare our catkin package .
12 catkin_package ()
13
14 # Specify locations of header files .
15 include_directories ( include ${catkin_INCLUDE_DIRS} )
16
17 # Declare the executable, along with its source files . If
18 # there are multiple executables , use multiple copies of
19 # this line .
20 add_executable( hello hello.cpp )
21
22 # Specify libraries against which to link. Again, this
23 # line should be copied for each distinct executable in
24 # the package .
25 target_link_libraries ( hello ${catkin_LIBRARIES} )
```

表 3.3 用于编译hello.cpp 的 CMakeLists.txt 文件。

Dashing:

```
1 cmake_minimum_required(VERSION 3.5)
2 project(agitr)

3 # Default to C99
4 if(NOT CMAKE_C_STANDARD)
5 set(CMAKE_C_STANDARD 99)
6 endif()
```

```
7 # Default to C++14
8 if(NOT CMAKE_CXX_STANDARD)
9 set(CMAKE_CXX_STANDARD 14)
10 endif()

11 if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES
    "Clang")
12 add_compile_options(-Wall -Wextra -Wpedantic)
13 endif()

14 # find dependencies
15 find_package(ament_cmake REQUIRED)
16 # uncomment the following section in order to fill in
17 # further dependencies manually.
18 # find_package(<dependency> REQUIRED)
19 find_package(rclcpp REQUIRED)

20 add_executable(hello src/hello.cpp)
21 ament_target_dependencies(hello rclcpp)
22 target_include_directories(hello PUBLIC
23 ${<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>}
24 ${<INSTALL_INTERFACE:include>})

25 install(TARGETS hello
26 EXPORT export_${PROJECT_NAME}
27 DESTINATION lib/${PROJECT_NAME})

28 if(BUILD_TESTING)
29 find_package(ament_lint_auto REQUIRED)
30 # the following line skips the linter which checks for copyrights
31 # uncomment the line when a copyright and license is not present in all source files
32 #set(ament_cmake_copyright_FOUND TRUE)
33 # the following line skips cpplint (only works in a git repo)
34 # uncomment the line when this package is not in a git repo
35 #set(ament_cmake_cpplint_FOUND TRUE)
36 ament_lint_auto_find_test_dependencies()
37 endif()

38 ament_package()
```

3.2.3 执行 hello 程序

当所有这些编译步骤完成后，新的 ROS 程序就可以使用 `roslaunch`（见 2.6 节）来执行，就像任何其他 ROS 程序一样。对于我们的例程，命令是：

```
roslaunch agitr hello
```

Dashing:

```
ros2 run agitr hello
```

这个程序会在终端打印类似于下面这样的输出：

```
[INFO] [1416432122.659693753]: Hello, ROS!
```

Dashing:

```
[INFO] [minimal_info]: Hello ROS2, 你好 ROS2.
```

不要忘了首先要启动 `roscore`：这个程序是一个节点，节点需要一个节点管理器才可以正常运行。顺便说一下，这行输出中的数字代表时间，从 1970 年 1 月 1 日开始以秒计数，这一时间是我们的 `ROS_INFO_STREAM` 开始执行的时间。

注意：ROS2 可以直接运行节点，无需启动 `master`，采用 DDS。



这个 `roslaunch` 命令，以及其他一些 ROS 命令，可能会产生如下所示的错误：

```
[rospack] Error: stack/package package-name not found
```

此错误的两个可能的原因是：（1）错误地拼写了功能包名；
（2）没有成功运行你的工作区中的 `setup.bash`。

3.3 发布者程序

上一节的hello程序展示了如何去编译和运行一个简单的ROS程序。那个程序对于介绍catkin命令是有帮助的，ROS2为colcon命令，但是就像所有的“Hello, World!”程序一样，它并没有做任何有用的事情。在本节中，我们会看到一个与ROS有更多交互的程序⁶。具体来说，我们将看到如何发送随机生成的速度指令到一个turtlesim海龟，使它漫无目的地巡游。这个程序的源文件称为pubvel，其代码见表3.4。这个程序展示了从代码中发布消息涉及的所有要素。

⁶ [http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber\(c++\)](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++))
<https://index.ros.org/doc/ros2/Tutorials/Writing-A-Simple-Cpp-Publisher-And-Subscriber/>
<https://index.ros.org/doc/ros2/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber/>

3.3.1 发布消息

`pubvel` 和 `hello` 程序主要的区别都是由于发布消息的需求导致的。

包含消息类型声明 你应该还能回忆起我们在 2.7.2 节中谈到过每一个 ROS 话题都与一个消息类型相关联。每一个消息类型都有一个相对应 C++ 头文件。你需要在你的程序中为每一个用到的消息类型包含这个头文件，代码如下所示：

```
#include <package_name/type_name.h>
```

这里需要注意的是，功能包名应该是定义消息类型的包的名称，而不一定是你自己的包的名称。在 `pubvel` 程序中，我们想发布一条类型为 `geometry_msgs/Twist` 的消息（名为 `geometry_msgs` 的包所拥有的类型为 `Twist` 的消息），我们应该这样：

```
#include <geometry_msgs/Twist.h>
```

Dashing:

```
#include "geometry_msgs/msg/twist.hpp"
```

这个头文件的目的是定义一个 C++ 类，此类和给定的消息类型含有相同的数据类型成员。这个类定义在以包名命名的域名空间中。这样命名的实际影响是当引用 C++ 代码中的消息类时，你将会使用双分号（`::`）来区分开包名和类型名，双分号也称为范围解析运算符。在我们的 `pubvel` 例程中，头文件定义了一个名为 `geometry_msgs::Twist` 的类。

```

1 // This program publishes randomly-generated velocity
2 // messages for turtlesim.
3 #include <ros/ros.h>
4 #include <geometry_msgs/Twist.h> // For geometry_msgs::Twist
5 #include <stdlib.h> // For rand() and RAND_MAX
6
7 int main ( int argc , char ** argv ) {
8     // Initialize the ROS system and become a node .
9     ros::init ( argc , argv , " publish_velocity " ) ;
10    ros::NodeHandle nh ;
11
12    // Create a publisher object .
13    ros::Publisher pub = nh . advertise <geometry_msgs::Twist>(
14    "turtle1 /cmd_vel" , 1000 ) ;
15
16    // Seed the random number generator .
17    srand ( time ( 0 ) ) ;
18
19    // Loop at 2Hz until the node is shut down.
20    ros::Rate rate ( 2 ) ;
21    while ( ros::ok ( ) ) {
22        // Create and fill in the message . The other four
23        // fields , which are ignored by turtlesim , default to 0.
24        geometry_msgs :: Twist msg ;
25        msg . linear . x = double ( rand ( ) ) / double ( RAND_MAX ) ;
26        msg . angular . z = 2 * double ( rand ( ) ) / double ( RAND_MAX ) - 1 ;
27
28        // Publish the message .
29        pub . publish ( msg ) ;
30
31        // Send a message to roscore with the details .
32        ROS_INFO_STREAM ( "Sending random velocity command : "
33        << " linear=" << msg . linear . x
34        << " angular=" << msg . angular . z ) ;
35
36        // Wait until it's time for another iteration .
37        rate . sleep ( ) ;
38    }
39 }

```

表 3.4 pubvel.cpp 程序的代码，用于给 turtlesim 仿真器中的海龟发布随机生成的运动速度指令消息。

Dashing:

```
1  #include <iostream>
2  #include <chrono>
3  #include "rclcpp/rclcpp.hpp"
4  #include "geometry_msgs/msg/twist.hpp"

5  using namespace std::chrono_literals;

6  int main(int argc, char * argv[])
7  {
8      rclcpp::init(argc, argv);
9      auto node = rclcpp::Node::make_shared("publish_velocity");
10     auto publisher = node->create_publisher<geometry_msgs::msg::Twist>("/turtle1/cmd_vel", 10);
11     geometry_msgs::msg::Twist message;
12     rclcpp::WallRate loop_rate(500ms);

13     while (rclcpp::ok()) {
14         message.linear.x = ((double)rand()/(RAND_MAX));
15         message.angular.z = 4.0*((double)rand()/(RAND_MAX))-2;
16         RCLCPP_INFO(node->get_logger(), "Publishing /turtle1/cmd_vel : linear=%f,angular=%f",
            message.linear.x, message.angular.z);
17         publisher->publish(message);
18         rclcpp::spin_some(node);
19         loop_rate.sleep();
20     }
21     rclcpp::shutdown();
22     return 0;
23 }
```

创建发布者对象 发布消息的实际工作是由类名为`ros::Publisher`的一个对象来完成的⁷。类似下面这行的代码创建了我们需要的对象：

```
ros::Publisher pub =
node_handle.advertise<message_type>(topic_name, queue_size);
```

让我们看下这一行代码的每一部分。

> `node_handle` 是 `ros::NodeHandle` 类的一个对象，是你在程序的开始处创建的。我们将调用这个对象的 `advertise` 方法。

> 在尖括号中的 `message_type` 部分，其正式名称为模板参数，是我们要发布的消息的数据类型。这个应该是上面讨论过的头文件中定义的类型名。在例程中，我们使用 `geometry_msgs::Twist` 类。

> **topic_name** 是一个字符串，它包含了我们想发布的话题的名称。它应该和 **rostopic list** 或者 **rqt_graph** 中展示的话题名称一致，但通常没有前斜杠 (/)。我们丢掉前斜杠使话题名为一个相对名称；第 5 章解释了相对名称的机制和目的。在此例程中，话题名为 **turtle1/cmd_vel**。



请注意话题名和消息类型的区别。如果不小心将二者混淆，将会产生很多潜在的令人困惑的编译错误。


> **advertise** 最后的参数是一个整数，表示这个发布者发布的消息序列的大小。在大多数情况下，一个相对比较大的值，比方说 **1000**，是合适的。如果你的程序迅速发布比队列可以容纳的更多的消息，最早进入队列的未发送的消息将被丢弃。

⁷ <http://wiki.ros.org/roscpp/Overview/PublishersandSubscribers>

▶▶ 需要这个参数是因为大多数情况下，消息必须发送到其他节点。这个通信过程可能是耗时的，特别是相比于创建消息所需要的时间。**ROS** 通过以下方法来减小延迟，当有新的消息产生时，**publish** 方法（见下文）只负责在“发件箱”中存储这条消息，而后台有一个单独的线程负责实际发送消息。此处的整数值是消息的数目，而并不是你可能认为的消息队列可以容纳的字节数。

有趣的是，**ROS**客户端库足够聪明，知道什么时候发布者和订阅者节点是相同进程的一部分。在这种情况下，消息会直接传送给订阅者，而不会使用任何网络传输。这一特点对于创建**nodelets**⁸尤其重要，即将多个节点动态载入到相同的进程。

如果你想从同一个节点发布关于多个话题的消息，你需要为每个话题创建一个独立的 **ros::Publisher** 对象。

 要注意你的 **ros::Publisher** 对象的生命周期。创建一个发布者是一个很耗时的操作，所以每当你想发布一个消息就去创建一个新的 **ros::Publisher** 对象是很不明智的。相反，建议为每一个话题创建一个发布者，并且在你程序执行的全过程中一直使用那个发布者。在 **pubvel** 中，我们通过在 **while** 循环外面声明发布者来达到这个目的。

⁸ <http://wiki.ros.org/nodelet>

创建并填充消息对象 下一步，我们创建消息对象本身。当我们创建 `ros::Publisher` 对象时已经引用了消息类。对于消息类型的每个域，这个类都有一个可公共访问的数据成员。

我们使用 `rosmmsg show`（见 2.7.2 节）命令看到消息类型 `geometry_msgs/Twist` 有两个顶层域（`linear` 和 `angular`），每个域都包含了三个子域（`x`、`y` 和 `z`）。每个子域是一个 64 位的浮点数，许多 C++ 编译器称为 `double` 型。表 3.4 中的代码创建了一个 `geometry_msgs::Twist` 对象，并且把伪随机数赋值给其中两个数据成员：

```
geometry_msgs::Twist msg;
msg.linear.x = double(rand())/double(RAND_MAX);
msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
```

这段代码设置线速度为 0 到 1 之间的某个值，角速度为 -1 到 1 之间的某个值。因为 `turtlesim` 忽略了其他四个域（`msg.linear.y`、`msg.linear.z`、`msg.angular.x` 和 `msg.angular.y`），我们让它们保持为默认值，即 0。

当然，大多数消息类型含有 `float64` 类型之外的域。幸运的是，从 ROS 域类型到 C++ 数据类型的映射正如你所期望的方式工作⁹。一个可能未被注意到的事实是，数组类型（在 `rosmmsg` 中用方括号表示）在 C++ 代码中是通过 STL 向量实现的。

发布消息 在所有的前期工作完成后，使用 `ros::Publisher` 对象的 `publish` 方法可以很简单地发布消息。例如下面所示：

```
pub.publish(msg);
```

⁹ <http://wiki.ros.org/msg>

这个方法将所给的消息添加到发布者的输出消息队列中，从这里，它会尽快被发送到相同话题的订阅者那里。

定义输出格式 表3.4 中的 `ROS_INFO_STREAM` 行尽管和发布速度命令不是直接相关的，但还是值得一看的。这是关于宏 `ROS_INFO_STREAM` 可以做什么的一个更加完整的例证，因为它演示了在输出中除了插入字符串还可以插入其他数据的能力。4.3 节有关于 `ROS_INFO_STREAM` 如何工作的详细信息。

3.3.2 消息发布循环

上一节中介绍了发布消息的细节。我们的 `pubvel` 例程在 `while` 循环中重复发布消息的步骤，随着时间的推移发布不同的消息。程序在这个循环中使用了两个附加的构造函数。

节点是否停止工作的检查 `pubvel` 的 `while` 循环的条件是：

```
ros::ok()
```

通俗地说，这个函数检查我们的程序作为 `ROS` 节点是否仍处于运行良好的状态。它会一直返回 `true`，除非这个节点有某种原因使其停止工作。有如下几个原因会使 `ros::ok()` 返回 `false`：

你可能对节点使用了 `rostop kill` 命令。

你可能给程序发送了一个终止信号（`Ctrl-C`）。

►► 有趣的是，`ros::init()` 给该信号放置了一个句柄，用它来发起正常的关机程序。其过程是 `Ctrl-C` 可以被用来使 `ros::ok()` 返回 `false`，但不会立刻终止程序。如果在程序退出前完成有一些必要的收尾工作（写日志文件、保存结果、说再见等）的话，这个设计是很重要的。

你可能在程序的某个位置调用了

```
ros::shutdown()
```

这个函数是在你的代码中发送节点工作已经完成信号的一个很有用的方法。

你可能以相同的名字启动了其他节点，经常是因为你启动了一个相同程序的新实例。

控制消息发布频率 pubvel的最后一个新知识点是它使用了ros::Rate对象¹⁰:

```
ros::Rate rate(2);
```

这个对象控制循环运行速度，其构造函数中的参数以赫兹（Hz）为单位，即每秒钟的循环数。这个例子创建了旨在规范每秒钟执行两个迭代循环的速率对象。邻近每次循环迭代的结尾，我们调用此对象的 sleep 方法：

```
rate.sleep();
```

每次调用此方法时就会在程序中产生延迟。延迟的持续时间被用来阻止循环的迭代速率超过指定的速率。没有这种控制，程序会以计算机允许的最快速度发布消息，这样会占满发布和订阅的序列，并且浪费计算和网络资源（在作者的电脑上，一个不加控制的程序会每秒发布大约 6300 条消息。）。

你可以使用 `rostopic hz` 命令来确认这种调控是否工作正常。对于 pubvel 程序，输出结果类似于：

```
average rate: 2.000
```

¹⁰ <http://wiki.ros.org/roscpp/Overview/Time>

min: 0.500s max: 0.500s std dev: 0.00006s window: 10

我们看到消息以每秒 2 条的速率发布，且时间上偏差非常小。



你可能会想到在每一个循环迭代中使用简单的固定延迟（例如 `sleep` 或者 `usleep` 命令）来代替 `ros::Rate`。`ros::Rate` 对象相对于这种方法的优势在于，`ros::Rate` 可以考虑循环中其他部分消耗的时间。如果每个迭代包含复杂计算（如我们所期待的真正的程序），通过该计算所消耗的时间应该从延迟中减除。在极端情况下，循环所使用的时间比预期的速率还要长，通过 `sleep()` 方法产生的延迟会减少到 0。

3.3.3 编译 pubvel

编译 `pubvel` 的过程大部分和 `hello` 例程的相同，只需要适当修改 `CMakeLists.txt` 和 `package.xml`，然后用 `catkin_make` 来编译你的工作区。然而，它和 `hello` 例程有一个很重要的区别。

声明消息类型依赖库 因为 `pubvel` 使用了来自 `geometry_msgs` 包的消息类型，我们必须声明对这个包的依赖关系，这和 3.2.2 节中讨论的 `roscpp` 依赖库的形式相同。具体而言，除了 `roscpp`，我们必须修改 `CMakeLists.txt` 文件的 `find_package` 行来声明 `geometry_msgs`：

```
find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs)
```

注意：我们是修改已有的 `find_package` 行，而不是新建一行。在 `package.xml` 文件中，我们添加新的依赖项：

```
<build_depend>geometry_msgs</build_depend>
```

```
<run_depend>geometry_msgs</run_depend>
```



如果你跳过（或者遗忘）这一步，`catkin_make` 可能无法找到 `geometry_msgs/Twist.h` 头文件。因此在编译 ROS 程序时当你看到找不到头文件的错误，最好确认一下你的功能包的依赖关系。

3.3.4 执行 pubvel

最后，我们已经准备好运行 `pubvel` 了。像往常一样，`roslaunch` 可以做这个工作。

```
roslaunch agitr pubvel
```

Dashing:

```
ros2 run agitr pubvel
```

同时，你可能想运行一个 `turtlesim` 模拟器，这样你就可以看到海龟响应 `pubvel` 发布的运动命令：

```
roslaunch turtlesim turtlesim_node
```

Dashing:

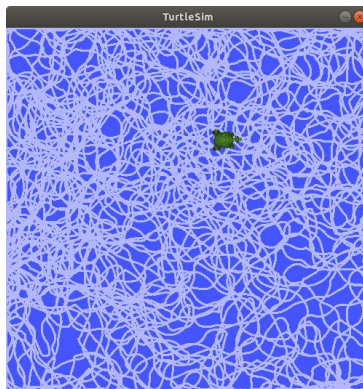
```
ros2 run turtlesim turtlesim_node
```

图 3.1 以示例形式展示了运行结果。



图 3.1 一个 `turtlesim` 海龟响应 `pubvel` 发布的随机速度命令。

Dashing:



```
relaybot@TP52: ~/RobTool/ROS2/dev_ws
relaybot@TP52: ~/RobTool/ROS2/dev_ws 79x22
lar='1.402721'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.139198',angu
lar='0.747049'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.955263',angu
lar='0.345175'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.757009',angu
lar='0.352699'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.069023',angu
lar='1.402401'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.128404',angu
lar='0.268239'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.771707',angu
lar='1.659522'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.419112',angu
lar='1.971565'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.432672',angu
lar='1.625448'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.363107',angu
lar='0.260136'
[INFO] [publish_velocity]: Publishing /turtle1/cmd_vel : linear='0.883350',angu
lar='0.634071'
```

3.4 订阅者程序

到目前为止，我们已经学习了一个发布消息的例程。当然，这仅仅完成了与其他节点通过消息进行通信的一半工作。现在，让我们看看一个节点如何订阅其他节点发布的信息¹¹。

我们继续使用 `turtlesim` 作为测试平台，订阅 `turtlesim_node` 发布的 `/turtle1/pose` 话题。这一话题的消息描述了海龟的位姿（位置和朝向）。表 3.5 展示了一个订阅这些消息的简短的程序，并且通过 `ROS_INFO_STREAM` 为我们输出到终端。尽管目前你已经对这一程序的某些部分感到熟悉了，但这里还是有三个新的知识点。

编写回调函数 发布和订阅消息的一个重要的区别是订阅者节点无法知道消息什么时候到达。为了应对这一事实，我们必须把响应收到消息事件的代码放到回调函数里，ROS 每接收到一个新的消息将调用一次这个函数。订阅者的回调函数类似于：

```
void function_name(const package_name::type_name &msg)
{
    ...
}
```

其中参数 `package_name` 和 `type_name` 和发布消息时的相同，它们指明了我们想订阅的话题的消息类。回调函数的主体有权限访问接收到消息的所有域，并以它认为合适的方式存储、使用或丢弃接收到的数据。与往常一样，我们必须包含定义该类的头文件。

本例程中，回调函数接收类型为 `turtlesim::Pose` 的消息，所

¹¹ [http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber\(c++\)](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++))

以我们需要的头文件是 `turtlesim/Pose.h`。(我们可以使用 `ros-topic info` 命令来确定这是正确的消息类型，请回忆 2.7.2 节的知识)。这个回调函数仅仅是通过 `ROS_INFO_STREAM` 在终端打印消息数据，包括 `x`、`y` 和 `theta` 数据成员。(我们可以使用 `rosmmsg` 来查看这个消息类型拥有哪些数据域，同样见 2.7.2 节)。与这里不同，一个真正的程序当然会使用消息来做一些有意义的事情。

注意订阅者的回调函数的返回值类型为 `void`。其实这样安排是合理的，因为调用此函数是 ROS 的工作，返回值也要交给 ROS，所以我们的程序无法获得返回值，当然非 `void` 的返回值类型也就没有意义了。

创建订阅者对象 为了订阅一个话题，我们需要创建一个 `ros::Subscriber` 对象¹²：

```
ros::Subscriber sub = node_handle.subscribe  
    (topic_name, queue_size, pointer_to_callback_function);
```

这个构造函数有三个形参，其中大部分与 `ros::Publisher` 声明中的类似：

`node_handle` 与我们之前多次见到的节点句柄对象是相同。

`topic_name` 是我们想要订阅的话题的名称，以字符串的形式表示。本例程中是 `"turtle1/pose"`。再次强调，我们忽略了前斜线使其成为相对名称。

`queue_size` 是本订阅者接收消息的队列大小，是一个整数。通常，你可以使用一个较大的整数，例如 1000，而不用太多关心队列处理过程。

¹² <http://wiki.ros.org/roscpp/Overview/PublishersandSubscribers>

```
1 // This program subscribes to turtle1/pose and shows its
2 // messages on the screen .
3 #include <ros/ros.h>
4 #include <turtlesim / Pose.h>
5 #include <iomanip> // for std::setprecision and std::fixed
6
7 // A callback function. Executed each time a new pose
8 // message arrives .
9 void poseMessageReceived ( const turtlesim::Pose& msg ) {
10     ROS_INFO_STREAM( std::setprecision(2) << std::fixed
11         << "position=(" << msg.x << " , " << msg.y << " )"
12         << " *direction=" << msg.theta );
13 }
14
15 int main ( int argc , char ** argv ) {
16     // Initialize the ROS system and become a node .
17     ros::init( argc , argv , "subscribe_to_pose" );
18     ros::NodeHandle nh;
19
20     // Create a subscriber object .
21     ros::Subscriber sub = nh.subscribe ( "turtle1/pose" , 1000 ,
22         &poseMessageReceived );
23
24     // Let ROS take over .
25     ros::spin();
26 }
```

表 3.5 程序subpose.cpp 订阅 turtlesim 机器人发布的位姿数据。

Dashing:

```
1  #include "rclcpp/rclcpp.hpp"
2  #include "turtlesim/msg/pose.hpp"
3  rclcpp::Node::SharedPtr g_node = nullptr;

4  void topic_callback(const turtlesim::msg::Pose::SharedPtr msg)
5  {
6      RCLCPP_INFO(g_node->get_logger(), "I heard: turtle1/pose position=%f,%f;
        direction=%f", msg->x, msg->y, msg->theta);
7  }
8  int main(int argc, char * argv[])
9  {
10     rclcpp::init(argc, argv);
11     g_node = rclcpp::Node::make_shared("subscribe_to_pose");
12     auto subscription =
13         g_node->create_subscription<turtlesim::msg::Pose>("turtle1/pose", 10, topic_callback);
14     rclcpp::spin(g_node);
15     rclcpp::shutdown();
16     // TODO(clalancette): It would be better to remove both of these nullptr
17     // assignments and let the destructors handle it, but we can't because of
18     // https://github.com/eProsima/Fast-RTPS/issues/235 . Once that is fixed
19     // we should probably look at removing these two assignments.
20     subscription = nullptr;
21     g_node = nullptr;
22     return 0;
23 }
```


当新的消息到达时，它们会被保存在一个队列中，直到 ROS 有机会去执行相应的回调函数。此参数表示 ROS 在队列中同一时刻可以存储的消息的最大值。如果新消息到达时队列已满，最早到达的还没有被处理的消息将会被丢弃以便腾出空间来。表面上看起来这可能很类似于发布消息时使用的方法（见第 58 页），但确是非常不同的：ROS 清空一个发布序列的速率取决于实际上给订阅者传输消息所占用的时间，而这个时间在很大程度上是不受控制的。相反，ROS 清空订阅序列的速度取决于我们处理回调函数有多快。因此，我们可以通过如下两个方法减少订阅者队列溢出的可能性：（1）通过调用 `ros::spin` 或者 `ros::spinOnce` 确保允许回调发生；（2）减少每个回调函数的计算时间。

最后一个参数是指向回调函数的指针，当有消息到达时要通过这个指针找到回调函数。在 C++ 中，你可以通过对函数名使用符号运算符（`&`，“取址”）来获得函数的指针。在我们的实例中，其方法如下：

`&poseMessageReceived`



不要在函数名后面添加括号 `()` 甚至 `(msg)` 这样常见的错误。这些括号和参数只有在你想调用函数的时候才需要，而不是你想获得函数指针时。ROS 在它调用你的回调函数时会提供合适的参数。

关于 C++ 语法：这个取址 (&) 运算符是可选的，许多程序会忽略它。编译器会分辨出你需要的是一个函数指针而不是函数运行的返回值，因为函数名后面没有跟着括号。作者的建议是包含它，因为这会使读者更明显地看出我们使用的是指针。

你可能注意到了，创建 `ros::Subscriber` 对象时，我们没有在任何地方显式地提到消息类型。实际上，`subscribe` 方法是模板化的（译者注：原书中这里误将 `subscribe` 写成了 `advise`），C++ 编译器会根据我们提供的函数指针中的数据类型判断出正确的消息类型。



如果使用错误的消息类型作为回调函数的参数，编译器将无法检测出错误。相反，你会看到关于类型不匹配的运行时错误消息。这些错误，可能来自发布者节点或者订阅者节点，具体取决于时间。

对于 `ros::Subscriber` 对象，一个可能违反直觉的事实是，它们的方法几乎很少被调用。相反，这些对象的生命周期才是最相关的部分：当我们构造一个 `ros::Subscriber`，我们的节点会与所有所订阅话题的发布者建立连接。当此对象被销毁时（越界或者 `delete` 了 `new` 操作符创建的对象），这些连接也随之失效。

给 ROS 控制权 最后的复杂之处在于只有当我们明确给 ROS 许可时，它才会执行我们的回调函数¹³。实际上有两个略微不同的方式来做这一点，其中一个版本如下所示：

¹³ <http://wiki.ros.org/roscpp/Overview/CallbacksandSpinning>

```
ros::spinOnce();
```

这个代码要求 ROS 去执行所有挂起的回调函数，然后将控制权限返回给我们。另一个方法如下所示：

```
ros::spin();
```

这个方法要求 ROS 等待并且执行回调函数，直到这个节点关机。换句话说，`ros::spin()`大体等于这样一个循环：

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

使用 `ros::spinOnce()` 还是使用 `ros::spin()` 的建议如下：你的程序除了响应回调函数，还有其他重复性工作要做吗？如果答案是“否”，那么使用 `ros::spin()`；否则，合理的选择是写一个循环，做其他需要做的事情，并且周期性地调用 `ros::spinOnce()` 来处理回调。表 3.5 使用 `ros::spin()`，因为程序唯一的工作就是接收和打印接收到的位姿消息。



订阅者程序中常见的一个错误是不小心忽略了调用 `ros::spinOnce` 和 `ros::spin`。在这种情况下，ROS 永远没有机会去执行你的回调函数。忽略 `ros::spin` 会导致你的程序在开始运行后不久就退出。忽略 `ros::spinOnce` 使程序表现的好像没有接收到任何消息。

1	[INFO][1370972120.089584153]: position=(2.42,2.32)direction=1.93
2	[INFO][1370972120.105376510]: position=(2.41,2.33)direction=1.95
3	[INFO][1370972120.121365352]: position=(2.41,2.34)direction=1.96
4	[INFO][1370972120.137468325]: position=(2.40,2.36)direction=1.98
5	[INFO][1370972120.153486499]: position=(2.40,2.37)direction=2.00
6	[INFO][1370972120.169468546]: position=(2.39,2.38)direction=2.01
7	[INFO][1370972120.185472204]: position=(2.39,2.39)direction=2.03


表 3.6 程序subpose.cpp 的输出样例，展示机器人位姿的逐渐变化。

Dashing:

```
relaybot@TP52: ~/RobTool/ROS2/dev_ws
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.184135','6.269610'; direction='-2.117953'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.183519','6.268693'; direction='-2.148392'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.182882','6.267791'; direction='-2.162830'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.182226','6.266903'; direction='-2.185269'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.181549','6.266030'; direction='-2.207707'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.180853','6.265173'; direction='-2.230146'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.180138','6.264331'; direction='-2.252584'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.179404','6.263506'; direction='-2.275023'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.178652','6.262697'; direction='-2.297461'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.177882','6.261906'; direction='-2.319900'
[INFO] [subscribe_to_pose]: I heard: turtle1/pose position='7.177135','6.261100'; direction='-2.342338'
```

3.4.1 编译并运行 subpose

编译和运行这个例程的步骤与我们之前看到的两个例程相同。



不要忘记确认为你的包添加了 **turtlesim** 依赖库，因为我们使用了 **turtlesim/Pose** 消息类型。参考 3.3.3 节复习如何声明依赖库。

当 `turtlesim_node` 和 `pubvel` 两个程序同时运行时，程序的输出如表 3.6 所示。

3.5 展 望

本章的主要内容是如何编写、编译和运行一些简单的程序，包括实现发布和订阅消息等 ROS 核心操作的程序。每一个程序都使用了 `ROS_INFO_STREAM` 宏来生成消息日志。在下一章中，我们会系统地学习 ROS 的日志系统，`ROS_INFO_STREAM` 仅仅是其中的一小部分。

第4章 日志消息[†]

业精于勤，荒于嬉；行成于思，毁于随。

——韩愈

本章将学习如何生成和查看日志消息。

在第 3 章 示例程序中，已经看到，一个叫做 `ROS_INFO_STREAM` 的宏给用户展示了丰富的消息。这些消息就是日志消息的例子。`ROS` 提供了一个包括 `ROS_INFO_STREAM` 在内、具有很多其他特性的日志系统。本章中，我们将学习如何使用这个日志系统。

4.1 严重级别

与大多数通用软件日志系统类似，`ROS` 日志系统的核心思想，就是使程序生成一些简短的文本字符流，这些字符流便是日志消息。在 `ROS` 中，日志消息分为五个不同的严重级别，也可简称为严重性或者级别。按照严重性程度递增，这些级别有¹：

DEBUG INFO WARN ERROR FATAL

其中，`DEBUG` 消息可能会比较频繁地出现，但是只要程序能够正常工作就不必过于在意。而反过来，`FATAL` 消息很少出现，但是却很重要，它通常表明程序中存在问题，导致已经无法继续运行。其余的三个级别：`INFO`，`WARN` 和 `ERROR` 则代表着 `DEBUG` 和 `FATAL` 这两端之间中间程度的严重性。图 4.1 以示例形式展示了一些 `ROS` 系统的不同级别日志内容。

[†] 本章由肖军浩、朱琪翻译。

¹ <http://wiki.ros.org/ROS/VerbosityLevels>

服务级别	示例消息
DEBUG	reading header from buffer
INFO	Waiting for all connections to establish
WARN	Less than 5GB of space free on disk
ERROR	Publisher header did not have required element: type
FATAL	You must call ros::init() before creating the first NodeHandle

图 4.1 各个严重级别的日志消息示例。

划分各种重要级别旨在提供一种区分和管理日志消息的全局方法。举个简单的例子，我们将学习如何通过定义允许的严重级别来过滤或者强调一些消息。然而，这些级别本身并不包含任何内在的含义：生成一个 FATAL 消息并不会终止你的程序；同样地，生成一个 DEBUG 消息并不会调试你的程序。

Dashing:

Logging macros for severity DEBUG.

```
#define RCLCPP_DEBUG(logger, ...)
```

```
#define RCLCPP_DEBUG_ONCE(logger, ...)
```

```
#define RCLCPP_DEBUG_EXPRESSION(logger, expression, ...)
```

```
#define RCLCPP_DEBUG_FUNCTION(logger, function, ...)
```

```
#define RCLCPP_DEBUG_SKIPFIRST(logger, ...)
```

Eloquent:

Logging macros for severity DEBUG.

```
#define RCLCPP_DEBUG(logger, ...)
```

#define	RCLCPP_DEBUG_ONCE (logger, ...)
#define	RCLCPP_DEBUG_EXPRESSION (logger, expression, ...)
#define	RCLCPP_DEBUG_FUNCTION (logger, function, ...)
#define	RCLCPP_DEBUG_SKIPFIRST (logger, ...)
#define	RCLCPP_DEBUG_THROTTLE (logger, clock, duration, ...)
#define	RCLCPP_DEBUG_SKIPFIRST_THROTTLE (logger, clock, duration, ...)
#define	RCLCPP_DEBUG_STREAM (logger, stream_arg)
#define	RCLCPP_DEBUG_STREAM_ONCE (logger, stream_arg)
#define	RCLCPP_DEBUG_STREAM_EXPRESSION (logger, expression, stream_arg)
#define	RCLCPP_DEBUG_STREAM_FUNCTION (logger, function, stream_arg)
#define	RCLCPP_DEBUG_STREAM_SKIPFIRST (logger, stream_arg)
#define	RCLCPP_DEBUG_STREAM_THROTTLE (logger, clock, duration, stream_arg)
#define	RCLCPP_DEBUG_STREAM_SKIPFIRST_THROTTLE (logger, clock, duration, stream_arg)

4.2 示例程序

本章剩余部分主要学习如何生成和查看日志消息。通常，通过一个具体的例子来描述将要学习的内容是很有帮助的，使用 `turtlesim` 就可以达到这个目的，因为在适当的条件下，`turtlesim_node` 可以生成除 `FATAL` 以外各个级别的日志消息。虽然不能生成所有严重级别的日志消息，但是从学习的目的出发，使用这样一个能够按要求生成可预测严重程度日志消息的程序非常方便。

表 4.1 展示了前面所述例程的程序。它能够在所有的五个严重级别上生成稳定的消息流。表 4.2 是它的一个命令行的输出示例。在本章余下的部分我们将使用这个例子。

```
1 //This program periodically generates log messages at
2 // various severity levels .
3 #include <ros/ros.h>
4
5 int main (int argc, char **argv) {
6     // Initialize the ROS system and become a node .
7     ros::init(argc, argv, "count_and_log");
8     ros::NodeHandle nh;
9
10    //Generate log messages of varying severity regularly .
11    ros::Rate rate(10);
12    for(int i=1; ros::ok(); i++){
13
14        ROS_DEBUG_STREAM("Counted to?"<<i);
15        if((i%3)==0){
16            ROS_INFO_STREAM(i<<"is?divisible?by?3.");
17        }
18        if((i%5)==0){
19            ROS_WARN_STREAM(i<<"is?divisible?by?5.");
20        }
21        if((i%10)==0){
22            ROS_ERROR_STREAM(i<<"is?divisible?by?10.");
23        }
24    }
```

```

23     if((i%20)==0){
24         ROS_FATAL_STREAM(i<<"?is?divisible?by?20.");
25     }
26     rate.sleep();
27 }
28 }

```

表 4.1 程序count.cpp 将在所有五个日志级别上产生日志消息。

```

1 [INFO] [1375889196.165921375]: 3 is divisible by 3.
2 [WARN] [1375889196.365852904]: 5 is divisible by 5.
3 [INFO][1375889196.465844839]: 6 is divisible by 3.
4 [INFO][1375889196.765849224]: 9 is divisible by 3.
5 [WARN] [1375889196.865985094]: 10 is divisible by 5.
6 [ERROR] [1375889196.866608041]: 10 is divisible by 10.
7 [INFO][1375889197.065870949]: 12 is divisible by 3.
8 [INFO][1375889197.365847834]: 15 is divisible by 3.

```

表 4.2 count 程序运行几秒后的输出样本。这个输出中不包含任何 DEBUG 级别的消息，因为默认的最小级别是 INFO。

独立写出Dashing程序，参考如下：

```

1  #include <iostream>
2  #include <chrono>
3  #include "rclcpp/rclcpp.hpp"

4  using namespace std::chrono_literals;

5  int main(int argc, char * argv[])
6  {
7      rclcpp::init(argc, argv);
8      auto node = rclcpp::Node::make_shared("count_and_log");
9      rclcpp::WallRate loop_rate(1000ms);

```

```

10 for(int i=1;rcldcpp::ok();i++)
11 {
12   RCLCPP_DEBUG(node->get_logger(),"Counted to %d",i);
13   if((i%3)==0)
14     RCLCPP_INFO(node->get_logger(),"%d is divisible by 3",i);
15   if((i%5)==0)
16     RCLCPP_WARN(node->get_logger(),"%d is divisible by 5",i);
17   if((i%10)==0)
18     RCLCPP_ERROR(node->get_logger(),"%d is divisible by 10",i);
19   if((i%20)==0)
20     RCLCPP_FATAL(node->get_logger(),"%d is divisible by 20",i);
21
22   rclcpp::spin_some(node);
23   loop_rate.sleep();
24 }
25 rclcpp::shutdown();
26 return 0;
27 }

```

```

relaybot@TPS2: ~/RobTool/ROS2/dev_ws
relaybot@TPS2: ~/RobTool/ROS2/dev_ws$ source install/setup.bash
relaybot@TPS2: ~/RobTool/ROS2/dev_ws$ ros2 run agitr count
[INFO] [count_and_log]: 1 is divisible by 3
[WARN] [count_and_log]: 5 is divisible by 5
[INFO] [count_and_log]: 6 is divisible by 3
[INFO] [count_and_log]: 9 is divisible by 3
[WARN] [count_and_log]: 10 is divisible by 5
[ERROR] [count_and_log]: 10 is divisible by 10
[INFO] [count_and_log]: 12 is divisible by 3
[INFO] [count_and_log]: 15 is divisible by 3
[WARN] [count_and_log]: 15 is divisible by 5
[INFO] [count_and_log]: 18 is divisible by 3
[WARN] [count_and_log]: 20 is divisible by 5
[ERROR] [count_and_log]: 20 is divisible by 10
[FATAL] [count_and_log]: 20 is divisible by 20
[INFO] [count_and_log]: 21 is divisible by 3
[INFO] [count_and_log]: 24 is divisible by 3
[WARN] [count_and_log]: 25 is divisible by 5
[INFO] [count_and_log]: 27 is divisible by 3
[INFO] [count_and_log]: 30 is divisible by 3
[WARN] [count_and_log]: 30 is divisible by 5
[ERROR] [count_and_log]: 30 is divisible by 10

```

4.3 生成日志消息

下面让我们更加完整地了解怎样从 C++ 代码生成日志消息。生成简单的日志消息 总共有五个基本的 C++ 宏用来产生日志消息，其中每个宏对应一个严重级别：

```
ROS_DEBUG_STREAM(message);  
ROS_INFO_STREAM(message);  
ROS_WARN_STREAM(message);  
ROS_ERROR_STREAM(message);  
ROS_FATAL_STREAM(message);
```

其中各个宏的参数 **message** 可以处理 C++ 中标准输出流（`ostream`）

中的各种表达式，比如 `std::cout`。这包括在 `int` 或者 `double` 这种基本数据类型上使用插入操作符（`<<`），以及已经重载这个操作符的复合数据类型，或者是标准流操作符，比如 `std::fixed`，

`std::setprecision` 或者 `std::boolalpha`。



需要指出的是，上述标准流操作符的作用范围只限于其出现的日志消息中。在其他日志消息中，任何想要使用的操作符必须重新插入。

这里介绍流操作符存在这种限制的原因：就像它们的名称中字母全部大写所暗示的一样，`ROS_..._STREAM` 都是宏。每一个宏都可以展开为其对应的一个短小的代码块，这些代码构造了一个标准字符串流（`std::stringstream`）对象，并将你提供的参数插入这个字符串流。然后，这些代码将字符串流中完全格式化的内容发送至一个内部的日志系统，即 `log4cxx`^[2]。由于在该过程完成时，字符串流对象将被销毁，因此，它的内部状态（包括流操作符建立的任意形式的配置）都将丢失。

如果你更喜欢用格式输出函数（`printf`）风格的接口来替换 C++ 风格的流，ROS 还提供了不带 `_STREAM` 后缀的宏。比如，宏

```
ROS_INFO(format, ...)
```

可以生成 `INFO` 级别的日志消息。只要你熟悉 `printf`，这些宏将按照你的期望工作。一个具体的例子是，表 3.4 中的代码大致等价于：

```
ROS_INFO("Position=(%0.2f, %0.2f) direction=%0.2f",  
msg.x,msg.y, msg.theta);
```

当然也有 `printf` 风格的只执行一次的（`..._ONCE`）和生成频率受控的（`..._THROTTLE`）宏，这些特性将在下面讲到，相关宏的名称也去掉了 `_STREAM` 部分。

² <http://wiki.ros.org/ROS/VerbosityLevels>

请注意，这里没有必要使用 `std::endl` 或者其他行终止符，因为日志系统本身就是面向行的。调用这些宏中任意一个将生成显示为一行的一个完整的日志消息。

生成一次性日志消息 在循环中或者在被频繁调用的函数中产生的日志消息对用户来说是非常重要的，但是有时这种重复性也让人感到烦恼。处理这种问题有一种很自然的途径，就是使用一个静态变量来确保消息只生成一次，即在第一次调用时。表 4.3 展示了一个可以实现该功能的 C++ 代码片段。为了避免重复这些冗长的代码块——将其封装为一个函数是无法实现一次性日志消息功能的，因为这种方法需要对每个函数声明设置不同的静态变量——幸运的是，ROS 提供了可以仅仅生成一次日志消息的简单的宏。

```
ROS_DEBUG_STREAM_ONCE(message);  
ROS_INFO_STREAM_ONCE (message);  
ROS_WARN_STREAM_ONCE (message);  
ROS_ERROR_STREAM_ONCE (message);  
ROS_FATAL_STREAM_ONCE (message);
```

程序运行中第一次到达这些宏时，它们将产生与相应的无 `-ONCE` 后缀版本宏一样的日志消息。第一次执行后，这些声明将不再发挥作用。表 4.4 展示了一个最小的例子，该循环的第一次迭代中，日志宏将各自产生一条消息，而在后续的迭代中这些宏将会被忽略。

```

1 // Don't do this directly. Use ROS_..._STREAM_ONCE instead.
2 {
3     static bool first_time = true ;
4     if (first_time) {
5         ROS_INFO_STREAM("Here's some important information"
6             <<" that will only appear once.");
7         first_time =false;
8     }
9 }

```

表 4.3 一个 C++代码片段，该段代码将在第一次执行后不再产生日志消息。ROS_..._STREAM_ONCE 宏可以展开与这类似的代码块。

```

1 // This program generates a single log message at each
2 //severity level .
3 #include <ros/ros.h>
4
5 int main ( int argc , char **argv ) {
6     ros::init(argc, argv,"log_once");
7     ros::NodeHandle nh;
8     while( ros::ok()){
9         ROS_DEBUG_STREAM_ONCE("This _ appears _ only _ once.");
10        ROS_INFO_STREAM_ONCE("This _ appears _ only _ once.");
11        ROS_WARN_STREAM_ONCE("This _ appears _ only _ once.");
12        ROS_ERROR_STREAM_ONCE("This _ appears _ only _ once.");
13        ROS_FATAL_STREAM_ONCE("This _ appears _ only _ once.");
14    }
15 }
16 }

```

表 4.4 只产生五条日志消息的示例程序 `once.cpp`。

生成频率受控的日志消息 还有一些宏可以用来控制日志消息出现的频率。

```
ROS_DEBUG_STREAM_THROTTLE(interval, message);  
ROS_INFO_STREAM_THROTTLE(interval, message);  
ROS_WARN_STREAM_THROTTLE(interval, message);  
ROS_ERROR_STREAM_THROTTLE(interval, message);  
ROS_FATAL_STREAM_THROTTLE(interval, message);
```

参数 **interval** 是 **double** 类型的，它表示以秒为单位的时间量，这是相邻日志消息出现的最小时间间隔。**ROS_..._STREAM_THROTTLE** 宏的每一个实例在第一次执行时都会生成日志消息（与不带 **_THROTTLE** 后缀版本宏的日志消息相同），随后的执行都会被忽略，直到经过了指定的时间间隔。每个宏的实例的时间被单独跟踪，方法是使用一个局部静态变量来存储上一次生成日志的时间。

表 4.5 展示了一个使用这些宏来获得与表 4.1 中程序非常相似行为的程序。除了消息内容，关键的不同是表 4.5 中的程序将会消耗更多的计算时间，因为它使用轮询机制而不是休眠机制，来决定什么时候应该生成新的日志消息。当然，这种轮询机制在实际程序中通常是一个糟糕的思路。

```

1 // This program generates log messages at varying severity
2 // levels, throttled to various maximum speeds.
3 #include <ros/ros.h>
4
5 int main(int argc, char *argv) {
6     ros::init (argc, argv, "log_throttled");
7     ros::NodeHandle nh;
8     while(ros::ok()){
9         ROS_DEBUG_STREAM_THROTTLE(0.1,
10             "This _ appears _ every _ 0.1 _ seconds." );
11         ROS_INFO_STREAM_THROTTLE(0.3,
12             "This _ appears _ every _ 0.3 _ seconds." );
13         ROS_WARN_STREAM_THROTTLE(0.5,
14             "This _ appears _ every _ 0.5 _ seconds." );
15         ROS_ERROR_STREAM_THROTTLE(1.0,
16             "This _ appears _ every _ 1.0 _ seconds." );
17         ROS_FATAL_STREAM_THROTTLE(2.0,
18             "This _ appears _ every _ 2.0 _ seconds.");
19     }
20 }
21 }

```

表 4.5 产生频率受控的日志消息的示例程序 `throttle.cpp`。

4.4 查看日志消息

到目前为止，我们已经花费了大量篇幅来讨论如何生成日志消息，但是很少涉及这些消息将会传递到何处。实际上，日志消息有三个不同的目的地：每一个日志消息可以在控制台输出，可以是 `rostop` 话题的消息，也可写入到日志文件中。让我们来学习如何使用这些功能。

4.4.1 控制台

首先，最明显的是，日志消息可以被送至控制台。具体而言，DEBUG和INFO消息被打印至标准输出（**standard output**），而WARN、ERROR和FATAL消息将被送至标准错误（**standard error**）³。

这里，标准输出和标准错误之间的区别是无关紧要的，除非你想要将其中一个或者这两个流输出重定向到一个文件或管道，这种情况比较复杂，通常的文件重定向技术

command > file

只重定向标准输出，不包括标准错误。为了将所有日志消息重定向到同一个文件，可以使用类似如下命令：

command &> file

然而，需要注意的是，由于这两种流的缓存方式不同，可能导致消息不按照顺序出现——在结果中 **DEBUG** 和 **INFO** 消息可能出现得比你预想的靠后。你可以使用 **stdbuf** 命令使标准输出采用行缓存方式，从而强制让消息按照正常顺序输出：

stdbuf -oL command &> file

最后，注意 **ROS** 会在输出中插入人类和软件无法理解的 **ANSI** 颜色编码，这些编码类似 **^[[0m]**，即使这些输出不是定向到终端中。查看一个包含这些编码的文件，可以使用如下命令：

less -r file

³ <http://wiki.ros.org/roscpp/Overview/Logging>

格式化控制台消息 可以通过设置 `ROSCONSOLE_FORMAT` 环境变量来调整日志消息打印到控制台的格式。该环境变量通常包含一个或多个域名，每一个域名由一个美元符号`$`和一对大括号`{}`来表示，用来指出日志消息数据应该在何处插入。默认的格式是：

```
[${severity}] [${time}]: ${message}
```

这个格式可能适合大部分的应用，但是还有一些其他的域也是很有用的⁴：

为了插入生成日志消息的源代码位置，可以使用`${file}`、`${line}` 和`${function}`域的组合形式。

为了插入生成日志消息的节点名称，可以使用`${node}`域。



roslaunch 工具（将在第 6 章介绍）默认并不会将标准输出和标准错误从其生成的节点导入至自己的输出流。为了查看使用 **roslaunch** 启动节点的输出，你必须显式地使用 `output="screen"` 属性，或者对 **roslaunch** 命令使用 `—screen` 参数来强制使所有节点应用这个属性。

⁴ <http://wiki.ros.org/rosconsole>
<https://index.ros.org/doc/ros2/Tutorials/Rqt-Console/Using-Rqt-Console/>

4.4.2 rosout 上的消息

除了在控制台上显示，每一个日志消息都被发布到话题 `/rosout` 上。该话题的消息类型是 `rosgraph_msgs/Log`。表 4.6 展示了该数据类型的各个域，其中包含了严重级别、消息本身和其他一些相关的元数据。

1	byte DEBUG=1
2	byte INFO=2
3	byte WARN=4
4	byte ERROR=8
5	byte FATAL=16
6	std_msgs/Header header
7	uint32 seq
8	time stamp
9	string frame_id
10	byte level
11	string name
12	string msg
13	string file
14	string function
15	uint32 line
16	string [] topics

表 4.6 `rosgraph_msgs/Log` 消息类型的域。

你可能已经注意到了，这里每一条消息的内容与前一小节讨论的控制台输出的消息非常相似。相比于控制台输出，`/rosout` 话题的主要作用是它在一个流中包含了系统中所有节点的日志消息。所有这些日志消息都可以通过 `/rosout` 查看，而与它们的节点在什么位置、什么时间、以何种方式启动的都无关，甚至与它们在哪台计算机上运行都是无关的。

由于/rosout 只是一个普通的话题，你当然可以通过

```
rostopic echo /rosout
```

Dashing:

```
ros2 topic echo /rosout
```

这条命令直接查看消息内容。如果你一定要坚持，甚至可以自己写一个程序来订阅/rosout话题，以你自己喜欢的方式来显示或者处理这些消息。然而，查看/rosout消息最简单的方式是使用下面这条命令^{5,6}：

```
rqt_console
```

Dashing:

```
rqt
```

图 4.2 展示了运行上述命令得到的图形界面。它显示了来自所有节点的日志消息，每一条消息独占一行，还有一些可以用来控制隐藏或高亮显示某类消息的过滤器。图形界面本身应该不需要额外的解释。

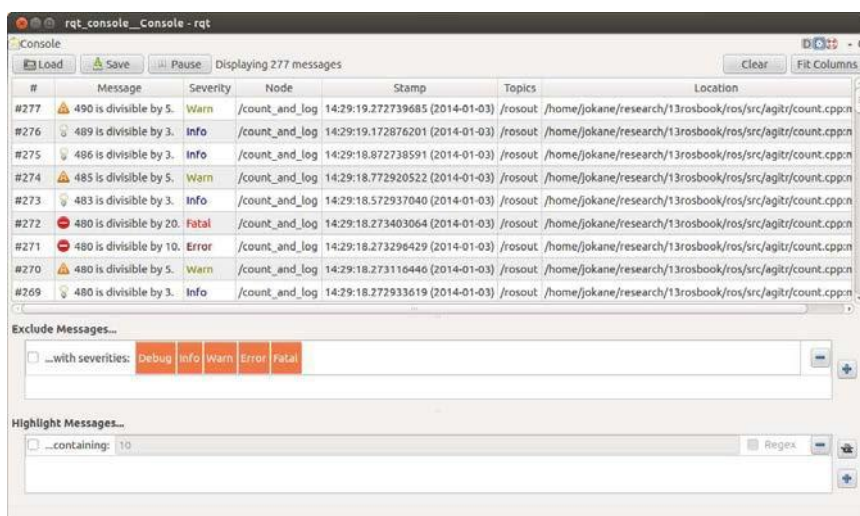
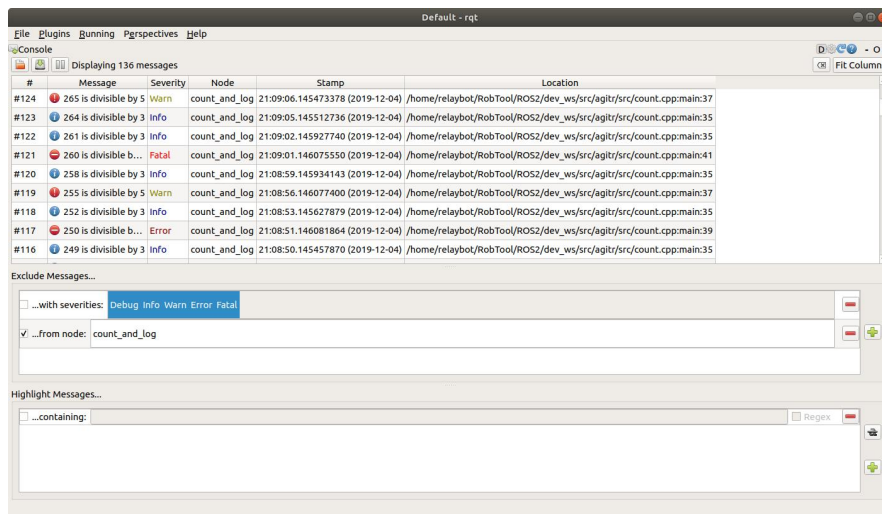


图 4.2 rqt_console 命令的图形界面。

Dashing:



⁵ <http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch>
<https://index.ros.org/doc/ros2/Tutorials/Launch-system/>

⁶ http://wiki.ros.org/rqt_console
<https://index.ros.org/doc/ros2/Tutorials/Rqt-Console/Using-Rqt-Console/>

4.4.3 日志文件

日志消息的第三个，也是最后一个目的地，是由 `rosout` 节点生成的日志文件。作为 `/rosout` 话题回调函数的一部分，该节点可以将日志消息作为一行写入到一个日志文件，文件名类似于：

```
~/ros/log/run_id/rosout.log
```

这里的 `rosout.log` 日志文件是纯文本文件。它可以利用 `less`、`head` 或者 `tail` 等命令行工具、或者是你喜欢的文本编辑器查看。运行标识码 (`run_id`) 是一个通用唯一识别码 (UUID)，它在节点管理器开始运行时基于你计算机的 MAC 地址和当前的时间生成。下面是一个 `run_id` 的例子：

```
57aa1860-d765-11e2-a830-f0def1e189cc
```

使用这类唯一标识码，可以区分来自不同 ROS 会话的日志文件。

查找运行标识码 至少有两种简单方法可以查看当前会话的 `run_id`

通过检查 `roscore` 生成的输出，在靠近输出末端的位置，可以看到与下面内容类似的一行

```
setting/run_idtorun_id
```

通过以下命令向节点管理器询问当前的

```
run_id
```

```
rosparam get/run_id
```

Dashing:

```
ros2 param
```

由于 `run_id` 存放在参数服务器上，因此该命令是有效的。关于参数的更多详情，请参考第 7 章。

检查和清除日志文件 这些日志文件将随着时间累积。如果你长期在一个对存储空间有着严格限制（由于账户配额或者是硬件限制）的系统上运行ROS，就会出现这个问题。`rosclean`和`roslaunch`运行时都会检查和监测已经存在的日志的大小，并会在日志文件大小超过 1GB时提醒你，但是并不会采取任何的措施来减小日志文件大小。也可以使用下面这条命令来查看当前账户中被ROS日志消耗的硬盘空间⁷：

`rosclean check`

如果日志正在消耗过多的硬盘空间，可以通过下面的命令删除所有已经存在的日志：

`rosclean purge`

如果你愿意，也可以手动删除这些日志文件。

4.5 启用和禁用日志消息

如果你亲自执行了表 4.1、表 4.4 和表 4.5 中的程序（或者仔细阅读了表 4.2 中的输出样本），你可能已经发现没有生成 `DEBUG` 级的消息，即使那些程序调用了 `ROS_DEBUG_STREAM` 宏。`DEBUG` 级别的消息发生了什么？答案是，ROS 的 C++ 程序默认只生成 `INFO` 或者更高级别的消息，尝试生成 `DEBUG` 级别的消息将会被忽略。

这是一个关于日志级别的具体例子，对每一个节点都指明了最低的严重级别。默认的日志级别是 `INFO`，这就解释了在我们的示例程序中为什么没有生成 `DEBUG` 级别的消息。日志级别背后通常的思路是在运行时提供调整每个节点日志细节程度的能力。

⁷ <http://wiki.ros.org/rosclean>



设置日志级别类似于 `rqt_consolt` 中的日志级别过滤选项。不同的是，改变日志级别将阻止日志消息的源头生成相应的消息，而 `rqt_console` 会接收任何输入的日志消息，其过滤选项只是选择性地显示其中的一部分。除了系统开销以外，效果是类似的。

对于那些由于日志消息级别而禁用的日志消息，消息的表达式并不会被评估。之所以存在这种可能，因为 `ROS_INFO_STREAM` 等构建方式是宏而不是函数调用。这些宏对应的展开代码会检查消息是否被启用，并且只会评估那些被启用消息的表达式。这意味着（1）你不应该依赖于构造消息字符串时可能发生的结果，（2）即使日志宏的参数需要消耗很多时间来评估，一旦这些日志消息被禁用，那么它们就不会使你的程序变慢。

设置某个节点的日志级别有几种方法。

通过命令行设置日志级别 为了通过命令行设置一个节点的日志级别，可以使用与以下类似的命令：

```
rosservice call /node-name/set_logger_level ros.package-name level
```

该命令调用 `set_logger_level` 服务，该服务由各个节点自动提供。（我们将在第 8 章详细探讨服务的相关内容。）

node-name 是你期望设置日志级别的节点名称

package-name 正如你猜测的一样，是拥有这个节点的功能包的名称

level 参数是 `DEBUG`、`INFO`、`WARN`、`ERROR`、`FATAL` 中的一个

字符串，即为节点设置的日志级别。

例如，为了在示例程序中启用 **DEBUG** 级别的消息，我们可以使用下面这条命令：

```
rosservice call /count_and_log/set_logger_level ros.agitr DEBUG
```

注意，由于这条命令直接与节点进行交互，我们不能在节点启动之前使用它。如果一切正常，这个对 **rosservice** 的调用将输出一个空行。



如果你拼错期望的日志级别，**set_logger_level** 服务将会报错，但是如果你拼错 **ros.package-name** 部分，它并不会报错。

rosvservice 的参数 **ros.package-name** 是必需的，用来指明我们期望配置的日志记录器（**logger**）的名称。从内部实现来讲，**ROS** 使用 **log4cxx** 这个库来实现其日志功能。本章中我们讨论的任何日志背后默认的日志记录器的名称都是 **ros.package-name**。

但是，**ROS C++** 客户端库内部也使用几种其他的日志记录器，来跟踪用户一般不太感兴趣的内容，如底层的读写字节、建立和关闭连接、调用回调函数。因为 **set_logger_level** 服务提供了所有这些日志记录器的配置接口，我们必须显式地指明我们想要配置的日志记录器。

这些额外的繁琐事情导致 **rosservice** 命令不会对你拼错日志记录器的名称做出反应。**log4cxx** 不会报错（当然，可以增加这个报错功能，但是意义不大），而是根据你指定的名称创建一个新的日志记录器。

通过图形界面设置日志级别 如果你更喜欢使用图形界面，而不是上述命令行接口，可以尝试以下命令：

rqt_logger_level

图 4.3 是运行该命令得到的窗口，该窗口可以让你从一个节点列表、一个日志记录器列表、以及一个日志级别列表中进行选择。使用这个工具改变日志级别的效果与前面提到的 `rosservice` 命令的效果是一致的，因为它也是对各节点使用了相同的服务调用。

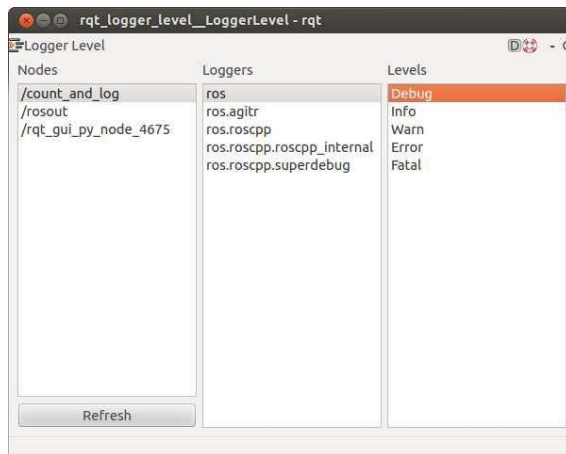


图 4.3 rqt_logger_level 命令的图形界面。

通过 C++代码设置日志级别 节点改变自身的日志级别也是可能的。最直接的方式是调用 ROS来实现日志功能的 `log4cxx` 提供的接口，代码如下：

```
#include <log4cxx/logger.h>

...

log4cxx::Logger::getLogger(ROSCONSOLE_DEFAULT_NAME)-
    >setLevel( ros::console::g_level_lookup[ros::console::levels::Debug]
);

ros::console::notifyLoggerLevelsChanged();
```

除了必要的语法理解，这些代码应该很容易识别是将日志级别设置为 **DEBUG**。Debug 这个标识当然可以替换为 **Info**、**Warn**、**Error** 或者 **Fatal**。

调用 `ros::console::notifyLoggerLevelsChanged()` 是有必要的，因为每个日志的启用或者禁用是缓存了的。如果你在生成任何日志消息之前就已经设置了日志级别，那么上述调用就可以省略。

4.6 展望

本章，我们学习了如何在 **ROS** 程序中生成日志消息以及如何通过不同的方式查看这些消息。这些消息对于跟踪和调试复杂 **ROS** 系统的行为是很有用的，特别是这些系统拥有大量不同节点时。下一章将讨论 **ROS** 的命名规则，如果使用得当，它也可以帮助我们用较小部分组成复杂的多节点系统。

补充：关于**ROS2**日志消息和配置参考如下链接：

> <https://index.ros.org/doc/ros2/Concepts/Logging/>

第5章 计算图源命名[†]

博学之，审问之，慎思之，明辨之，笃行之。

——礼记·中庸

本章我们将要学习 *ROS* 如何为其各类计算图资源（如节点、话题、参数和服务等）命名和解析。

在第三章，我们用字符串（例如“hello_ros”和“publish_velocity”）来给节点命名，用“turtle1/cmd_vel”和“turtle1/pose”等字符串作为话题的名称。这些都是计算图源名称的例子。*ROS* 具有一个能够接受几种不同类型名的灵活的命名系统（例如，上述四个字符串均是相对名称。）。本章我们将小费周章地来解释不同的计算图源名称以及 *ROS* 是如何解决命名问题的。我们之所以把这些相当简单的内容单独列为一章，是因为它们和本书后半部分中绝大部分的概念相关。

5.1 全局名称

节点、话题、服务和参数统称为计算图源，而每个计算图源由一个叫计算图源名称（**graph resource name**）的短字符串标识。事实上，计算图源名称在 *ROS* 命令行和代码中是广泛存在的，前面我们已经多次接触它们。例如，前面学习的 `rostopic info` 命令行工具和 `ros: : init` 函数都将节点名作为其参数，`rostopic echo` 命令行工具和 `ros: : Publisher` 构造函数则都要求提供话题名作为参数，这些都是计算图源名称的具体实例。

下面是一些我们已经遇到过的特定计算图源名称：

[†]本章由肖军浩、郭昭宇翻译。

```
/teleop_turtle
/turtlesim
/turtle1/cmd_vel
/turtle1/pose
/run_id
/count_and_log/set_logger_level
```

这些计算图源名称都属于全局名称，之所以叫做全局名称因为它们在任何地方（译者注：包括代码、命令行工具、图形界面工具等的任何地方）都可以使用。无论这些名称用作众多命令行工具的参数还是用在节点内部，它们都有明确的含义。这些名称从来不会产生二义性，也无需额外的上下文信息来决定名称指的哪个资源。

以下是一个全局名称的几个组成部分：

> 前斜杠“/”表明这个名称为全局名称。

> 由斜杠分开的一系列命名空间（**namespace**），每个斜杠代表一级命名空间。你可能已经猜到了，命名空间用于将相关的计算图源归类在一起。上述名称例子包含了两个显式的命名空间，分别为 **turtle1** 和 **count_and_log**。ROS 允许多层次的命名空间，所以下面这个包含了 11 个嵌套名称空间的名称也是有效的全局名称，虽然看起来不太可能有人这么用。

```
/a/b/c/d/e/f/g/h/i/j/k/l
```

如果没有显式提及所属的命名空间，包括上述三个例子在内，则对应的计算图源名称是归在全局命名空间中的。

> 描述资源本身的基本名称（**base name**）。上述例子中的基本名称分别为：**teleop_turtle**、**cmd_vel**、**pose**、**run_id** 和

`set_logger_level`。

我们必须意识到，如果在任何地方都使用全局名称，除了可能使人更容易追踪变量外，并不能从使用复杂的命名空间中受益多少。这种命名系统的真正优势来自于相对名称和私有名称。

5.2 相对名称

我们刚刚已经看到，使用全局名称时，为了指明一个计算图源，需要完整列出其所属的命名空间，尤其是有时候命名空间层次比较多，这可能会让你抓狂。这时，一个主要替代方案是让 ROS 为计算图源提供一个默认的命名空间，具有此特征的名称叫做相对计算图源名称（**relative graph resource name**），或简称为相对名称（**relative name**）。相对名称的典型特征是它缺少全局名称带有的前斜杠“/”。下面是一些相对名称的例子：

```
teleop_turtle
turtlesim
cmd_vel
turtle1/pose
run_id
count_and_log/set_logger_level
```

理解相对名称的关键是，如果不知道 ROS 解析某个计算图源时所使用的默认命名空间，相对名称并不能和特定计算图源匹配。

解析相对名称 将相对名称转化为全局名称的过程相当简单。ROS 将当前默认的命名空间的名称加在相对名称的前面，从而将相对名称解析为全局名称。比如，如果我们在默认命名空间为 `/turtle1` 的地方使用相对名称 `cmd_vel`，那么 ROS 通过下面的组合方法得

到全局名称:

$$\begin{array}{ccccc} \text{/turtle1} & + & \text{cmdvel} & = & \text{/turtle1 /cmdvel} \\ \text{默认命名空间} & & \text{相对名称} & & \text{全局名称} \end{array}$$

相对名称也可以以一系列的命名空间开始，这些命名空间被看作是默认命名空间中的嵌套空间。举个例子，如果我们在默认命名空间为/a/b/c/d/e/f的地方使用相对空间 g/h/i/j/k，ROS 将会将其进行下面的组合：

$$\begin{array}{ccccc} \text{/a/b/c/d/e/f} & + & \text{g/h/i/j/k/l} & = & \text{/a/b/c/d/e /f/g/h/i/j/k/l} \\ \text{默认命名空间} & & \text{相对名称} & & \text{全局名称} \end{array}$$

然后，得到的全局名称就可以用于确定一个特定的计算图源，就像前面介绍全局名称的使用时一样。

设置默认命名空间 默认的命名空间是单独地为每个节点设置的，而不是在系统范围进行。如果你不采取下面介绍的步骤来设置默认命名空间，那么 ROS 将会如你所期望的那样，使用全局命名空间 (/) 作为此节点的默认命名空间。为节点选择一个不同的默认命名空间的最好也是最常用的方法是在启动文件中使用命名空间 (ns) 属性（参见 6.3 节）。ROS 还提供了一些其他机制支持这种操作。

1、大部分 ROS 程序，包括调用 `ros::init` 的所有 C++ 程序，接受叫做 `_ns` 的命令行参数，此参数将为程序指定一个默认命名空间。

`__ns:=default-namespace`

2、还可以利用环境变量为在 shell 内执行的 ROS 程序设置默认命名空间。

Export ROS_NAMESPACE=default-namespace

请注意，只有当没有其他由 `__ns` 参数指定的默认命名空间时，这个环境变量才有效。

理解相对名称的目的 除了如何为相对名称设置默认命名空间这个问题外，有人可能会问另一个问题：“谁在意呢？”（译者注：我为什么要使用相对名称？）。乍一看，提出相对名称这个概念似乎只是为了避免每次都使用完整的全局名称编程。尽管相对名称提供了这种便捷方式，但是其真正意义在于它使得在小系统基础上实现复杂系统变得更加容易。

当一个节点内的计算图源全部使用相对名称时，这本质上给用户提供了一种非常简单的移植手段，即用户能方便地将此节点和话题移植到其他的（比如用户自己程序的）命名空间，而节点的原设计者并不一定参与这个过程。这种灵活性可以使得一个系统的组织结构更清晰，更重要的是能够防止在整合来自不同来源的节点发生名称冲突。作为对比，如果所有节点都使用全局名称命名自己的计算图源，就很难实现这种高效资源整合。所以，除非一些特殊情况有特殊要求，否则编写节点时并不推荐使用全局名称。

5.3 私有名称

私有名称，以一个波浪字符（~）开始，是第三类也是最后一类计算图源名称。和相对名称一样，私有名称并不能完全确定它们自身所在的命名空间，而是需要 ROS 客户端库将这个名称解析为一个全局名称。与相对名称的主要差别在于，私有名称不是用当前默认命名空间，而是用的它们节点名称作为命名空间。

例如，有一个节点，它的全局名称是 `/sim1/pubvel`，ROS 将

其私有名称`~max_vel` 转换至如下全局名称:

<code>/sim1/pubvel</code>	+	<code>~max_vel</code>	<code>/sim1/pubvel/max_vel</code>
节点名		私有名称	全局名称

这种命名方式基于如下事实，每个节点内部都有这样一些资源，这些资源只与本节点有关，而不会与其他节点打交道，这些资源就可以使用私有名称。私有名称常常用于参数——`roslaunch` 有专门的功能用于设置私有名称可以访问的参数，参见第 137 页；私有名称也用于管理一个节点运算的服务。将话题命名为私有名称是个常见错误，因为如果我们想要保持节点的松耦合性，那么没有一个话题是被任意某个特定节点所“拥有的”。



私有名称的关键字“**private**”仅仅表示其他节点不会使用它们所在的命名空间，也就是仅在命名空间层面上有意义。对于其他节点来讲，只要知道私有名称解析后的全局名称，都可以通过其全局名称访问这些计算图源。这和 **C++** 等其他类似编程语言中的关键字“**private**”是不同的，在这些编程语言中，系统中的其他部分是不能访问某个类的私有成员变量的。

5.4 匿名名称 (Anonymous names)

除了以上三种基本的命名类型，ROS 还提供了另一种被称为匿名名称的命名机制，一般用于为节点命名（译者注：这里的匿名并不是指没有名字，而是指非用户指定而又没有语义信息的名子）。匿名名称的目的是使节点的命名更容易遵守唯一性的规则。其思路是，当节点调用 `ros::init` 方法时可以请求一个自动分配的唯一名称。

为了请求一个匿名名称，节点需要将

`ros::init_options::AnonymousName`

作为第四个参数传递给

`ros::init` 方法：

`ros::init(argc, argv, base_name, ros::init_options::AnonymousName);`

这个附加选项的作用是在节点的基本名称后面追加某个额外的文本，以确保节点的名字是唯一的。

虽然 ROS 追加了什么特定额外文本对于用户来讲并不是那么重要，但作者觉得有必要指出 `ros::init` 使用处理器时间（wall clock time）生成匿名名称。

```
1 // This program starts with an anonymous name, which
2 // allows multiple copies to execute at the same time ,
3 // without needing to manually create distinct names
4 // for each of them.
5 #include <ros/ros.h>
6
7 int main ( int argc, char **argv ) {
8     ros::init ( argc, argv, "anon",
9         ros::init_options::AnonymousName );
10    ros::NodeHandle nh;
11    ros::Rate rate(1);
12    while (ros::ok()) {
13        ROS_INFO_STREAM( "This _ message _ is _ from _ "
14            << ros::this_node::getName( ));
15        rate.sleep( );
16    }
17 }
```

表 5.1 一个名为 `anon.cpp` 的程序，其中启动的节点使用匿名机制。因此，我们可以启动同一节点的大量副本，而不导致命名冲突。

表 5.1 给出了一个运用匿名机制的样本程序。这个程序启动的节点名称不仅简单包含用户命名的部分 (`anon`)，它们的名称看起来是这样的：

```
/anon_1376942789079547655
```

```
/anon_1376942789079550387
```

```
/anon_1376942789080356882
```

这个程序的行为本身并没有什么特别。但是，正因为它请求了一个匿名名称，所以我们可以同时运行任意多的以上程序的副本。这很容易理解，每个程序开始运行时，它就会得到一个具备唯一性的名字。

5.5 展望

在这一章，我们了解了 ROS 如何命名和解析计算图源名称。特别值得指出的是，具有有着很多相互交互节点的复杂 ROS 系统受益于使用相对名称或私有名称机制带来的灵活性。下一章将介绍一种称为 `roslaunch` 的工具，它将简化启动和配置多节点 ROS 会话的过程。

第6章 启动文件[†]

吾尝终日而思矣，不如须臾之所学也。

——荀子

利用启动文件一次性配置和运行多个节点。

如果到目前为止你已经走通了本书中所有的例程，那么现在你也会有些沮丧，因为除了搞定 `roscore` 外，你还需要在这么多不同的终端启动这么多不同的节点。幸运的是，ROS 提供了一个同时启动节点管理器（`master`）和多个节点的途径，即使用启动文件（`launch file`）。事实上，在 ROS 功能包中，启动文件的使用是非常普遍的。任何包含两个或两个以上节点的系统都可以利用启动文件来指定和配置需要使用的节点。本章将对启动文件和运行启动文件的 `roslaunch` 工具进行介绍。

补充：Dashing 请先阅读如下链接内容

> <https://index.ros.org/doc/ros2/Tutorials/Launch-Files/Creating-Launch-Files/>

> <https://index.ros.org/doc/ros2/Tutorials/Launch-system/>

6.1 使用启动文件

首先，我们来看一下 `roslaunch` 是如何同时启动多个节点的。其基本思想是在一个 XML 格式的文件内将需要同时启动的一组节点罗列出来¹。表 6.1 为一个启动文件示例，该示例同时启动一个 `turtlesim` 仿真器节点和一个遥控节点，这两个节点是我们在第二章遇到过的，还有一个是我们在第三章写的订阅节点。这个启动文件保存在 `agtr` 包的根目录下，文件名为 `example.launch`。在详细介绍启动文件的结构之前，我们先看一下这些文件是如何使用的。

[†] 本章由肖军浩、王志强翻译。

¹ <http://wiki.ros.org/roslaunch/XML>
<https://index.ros.org/doc/ros2/Tutorials/Launch-Files/Creating-Launch-Files/>

```
1 <launch>
2 <node
3 pkg="turtlesim "
4 type="turtlesim_node"
5 name="turtlesim "
6 respawn="true "
7 />
8 <node
9 pkg="turtlesim "
10 type="turtle_teleop_key"
11 name="teleop_key"
12 required="true "
13 launch -prefix="xterm -e"
14 />
15 <node
16 pkg="agitr "
17 type="subpose"
18 name="pose_subscriber "
19 output="screen "
20 />
21 </launch>
```

表 6.1 可以同时启动三个节点的启动文件 `example.launch`。

Dashing (turtlesim_mimic_launch.py) :

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            node_namespace='turtlesim1',
            node_executable='turtlesim_node',
            node_name='sim'
        ),
        Node(
            package='turtlesim',
            node_namespace='turtlesim2',
            node_executable='turtlesim_node',
            node_name='sim'
        ),
        Node(
            package='turtlesim',
            node_executable='mimic',
            node_name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])

```

加入如下内容：

```
Node(
    package='turtlesim',
    node_namespace='teleop_key',
    node_executable='turtle_teleop_key',
    node_name='sim'
),
Node(
    package='agitr',
    node_namespace='pose_subpose',
    node_executable='subpose',
    node_name='sim'
),

```

执行启动文件 想要运行一个启动文件，可以像下面这样使用
roslaunch命令²：

```
roslaunch package-name launch-file-name
```

Dashing:

```
ros2 launch package-name launch-file-name
```

因此，可以使用如下指令来运行该示例启动文件：

```
roslaunch agitr example.launch
```

Dashing:

```
ros2 launch example.launch
```

更多内容参考：

> <https://github.com/ros2/launch/blob/master/launch/doc/source/architecture.rst>

如果一切顺利，该指令将启动三个节点，此时你应该看到
turtlesim 仿真器窗口和另外打开一个可以接收键盘方向指令来远

² <http://weili.ros.org/roslaunch/CommandlineTools>

程控制海龟运动的窗口。运行 `roslaunch` 指令的终端会显示出由 `subpose` 程序记录的姿态信息。注意，在启动任意节点之前，`roslaunch` 首先会判断 `roscore` 是否正在运行；如果没有，则自动启动 `roscore`。



不要将 `roslaunch` 和 `roslaunch` 混为一谈，`roslaunch` 一次只能启动一个节点，而 `roslaunch` 可以同时启动多个节点。

值得指出的是，ROS 也允许使用不归属于任何功能包的启动文件，此时需要向 `roslaunch` 提供启动文件的路径，而不像上面的例子需要功能包名参数。例如，如下指令即可执行我们的示例启动文件，忽略该文件属于 `agitr` 包的事实（译者注：读者可以自己将此文件拷贝到非 ROS 功能包的目录下，再使用这种方法执行该启动文件）。

```
roslaunch ~/ros/src/agitr/example.launch
```

然而，这种变通方案没有按照功能包的形式来管理文件，并不是一个好的想法，只适合简单、简短的实验测试。

关于 `roslaunch` 工具一个重要但容易忘记的事实是，文件内所有的节点几乎都在同一时刻启动。因此，事实上你无法确定各个节点的启动顺序。不过这并没有什么影响，设计良好的 ROS 节点是不关心各个节点的启动顺序的。（7.3 节会列举一个启动顺序起重要作用的示例）。

该特性也表明，在 ROS 中，每一个节点都应该和其他节点尽可能保持独立。（回忆一下在 2.8 节中讨论的节点的松耦合关系）。只有在特定启动顺序下才能运行良好的节点不符合这种模块化思想。因此，这样的节点需要进行重新设计，从而克服启动顺序的约束。

请求详细信息（**Requesting verbosity**）像很多命令行工具一样，**roslaunch** 有一个可以请求输出详细信息的选项：

```
roslaunch -v package-name launch-file-name
```

表 6.2 显示的是该选项生成的除基本状态信息之外的额外信息。该信息可以用来观察 **roslaunch** 如何解释你的启动文件，有时在调试的时候会有所帮助。

终止启动过程可以使用 **Ctrl-C** 来终止一个活跃的 **roslaunch**，该命令首先尝试正常关闭所有由这个 **roslaunch** 启动的活跃节点，并且会强力关闭那些没有在短时间内退出的节点。

```
1 ...loading XML file [/opt/ros/indigo/etc/ros/roscore.xml] 2 ...
executing command param [ rosversion roslaunch]
3 Added parameter [/rosversion]
4 ...executing command param [rosversion-d] 5
Added parameter [/rostdistro]
6 Added core node of type [rosout/ rosout] in namespace [/]
7 ... loading XML file [/home/jokane/ ros / agitr /example . launch ] 8
Added node of type [ turtlesim /turtlesim_node ] in namespace [/] 9
Added node of type [ agitr /pubvel ] in namespace [/]
10 Added node of type [ agitr /subpose ] in namespace [/]
```

表 6.2 由 **roslaunch** 更多信息模式生成的额外信息。

6.2 创建启动文件

在了解如何使用启动文件之后，我们来介绍一下如何创建自己的启动文件。

6.2.1 启动文件的存储位置

和其他 ROS 文件一样，每一个启动文件都应该和一个特定的功能包关联起来。通常的命名方案是以 `.launch` 作为启动文件的后缀。最简单的方法是把启动文件直接存储在功能包的根目录中。当查找启动文件的时候，`roslaunch` 工具会同时搜索每个功能包目录的子目录。包括 ROS 核心包在内的很多功能包都是利用这一特性，将所有启动文件统一存放在一个子目录中，该子目录通常取名为 `launch`。

6.2.2 启动文件的基本元素

最简单的启动文件由一个包含若干节点元素（`node elements`）的根元素（`root element`）组成。

插入根元素 启动文件是 XML 文件，每个 XML 文件都必须包含一个根元素。对于 ROS 启动文件，根元素由一对 `launch` 标签定义：

```
<launch>
```

```
...
```

```
</launch>
```

每个启动文件的其他元素都应该包含在这两个标签之内。

启动节点 任何启动文件的核心都是一系列的节点元素，每个节点元素指向一个需要启动的节点³。节点元素的形式为：

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



在节点标签末尾的斜杠“/”是很容易忘记的，但是它很重要。它表明不必再等待结束标签（“</node>”），并且该节点元素是完整的。XML 语法分析器对于这样的细节要求得非常严格。如果你忽略了这个斜线，将会出现这样的错误信息：

Invalid roslaunch XML syntax: mismatched tag

你也可以这样显式地给出结束标签：

```
<node pkg="..." type="..." name="..."></node>
```

事实上，如果该节点有子节点，例如 **remap** 或者 **param** 元素，那么该显式结束标签是必不可缺少的。这两个元素将在

6.4 节和 **7.4 节**分别进行介绍。

每个节点元素有如下三个必需的属性。

pkg 和 **type** 属性定义了 ROS 应该运行哪个程序来启动这个节点。这些和 **roslaunch** 的两个命令行参数的作用是一致的，即给出功能包名和可执行文件的名称。

³ <http://wiki.ros.org/roslaunch/XML/node>

`name` 属性给节点指派了名称，它将覆盖任何通过调用 `ros::int` 来赋予节点的名称。

该属性可以覆盖代码中提供给`ros::int`的命名信息，包括节点可能发出的匿名请求（参见 5.4 节）。如果要在启动文件内使用一个匿名名称，可以使用一个`anon`关键字替换名称属性⁴，例如：

```
name="$(anon base_name)"
```

请注意，对同一个基本名称的重复利用将会产生相同的匿名名称。这说明(1)我们可以在启动文件的其他位置使用该名称，(2)对于每一个想要匿名的节点，都要确保使用不同的基本名称。

查看节点日志文件使用`roslaunch`启动一组节点与使用`roslaunch`单独启动每个节点的一个重要不同点是，在默认状态下，从启动文件启动节点的标准输出被重定向到一个日志文件中，而不是在控制台显示^{*}。该日志文件的名称是：

```
~/ros/log/run_id/node_name-number-stout.log
```

其中，`run_id` 是节点管理器启动时生成的一个唯一标示符。（在第 87X 页中介绍了如何去查看当前会话的 `run_id`）。文件名中的数字是对节点的标号。例如，运行表 6.1 的启动文件会将两个节点的标准输出存储到如下名称的日志文件中：

⁴ <http://wiki.ros.org/roslaunch/XML>

^{*} 在当前版本的 `roslaunch` 中，标准错误（包括控制台中的 `ERROR` 和 `FATAL` 级别的日志消息）的输出是显示在控制台上的，而不是在记录文件中。然而，`roslaunch` 源代码的注释中标明将在今后对这一点做出改变。

```
turtlesim-1-stdout.log  
telep_key-3-stdout.log
```

这些日志文件的内容可以在你喜欢的任意文本编辑器中查看。

在控制台中输出信息 对于某个单独的节点，只需在节点元素中配置 **output** 属性就可以达到该目的：

```
output="screen"
```

配置了该属性的节点会将标准输出显示在屏幕上而不是记录到之前讨论的日志文档。示例程序对 **subpose** 节点配置了该属性，这就是为什么该节点的 **INFO** 信息会出现在控制台中，同时也说明了为什么该节点没有出现在之前提到的日志文件列表中。

除了影响单个节点输出信息的 **output** 属性之外，我们还可以使用 **--screen** 命令行选项来令 **roslaunch** 在控制台中显示所有节点的输出：

```
roslaunch --screen package-name launch-file-name
```



如果一个从 **roslaunch** 中启动的程序没有产生预期的输出，你应该检查一下是否为该节点设置了 **output** 属性

(**output="screen"**)。

请求复位 在启动了启动文件中所有的请求节点之后，**roslaunch** 会监视每一个节点，记录哪一个节点是活跃的。对于每个节点，我们可以设置 **respawn** 属性为真，这样当节点停止的时候，**roslaunch** 会重新启动该节点。

```
respawn="true"
```

这个功能是有意义的，比如在某个节点因为软件崩溃或硬件

故障以及其他原因导致过早退出系统的时候会起到作用。

在我们的示例中，`respawn` 属性实际上是不需要的，因为三个程序都很可靠。但是我们在 `turtlesim_node` 中包含了该属性，用来展示该属性如何工作。当我们关闭 `turtlesim` 窗口的时候，相应的节点也会停止，`ROS` 很快就会识别到这样的变化。由于该节点设置了复位属性，一个新的 `turtlesim` 节点（有其对应的窗口）将会运行来代替之前的对应节点。

必要节点（**requiring node**）复位之外的另一种策略是将一个节点声明为必要节点。

`required="true"`

当一个必要节点终止的时候，`roslaunch` 会终止所有其他活跃节点并退出。当出现(1)某个节点非常重要，此节点的故障会导致整个 `ROS` 会话的瘫痪；(2)节点即便设置了复位属性也不能很好地重启的时候，该需求属性的作用就会体现出来。

示例中对 `turtle_teleop_key` 节点配置了 `required` 属性。如果你关闭了该远程控制节点所在的窗口，`roslaunch` 将会终止另外两个节点，然后退出。



由于 `respawn` 和 `required` 二者的作用是相互矛盾的，因此，如果对一个节点同时配置了这两种属性，`roslaunch` 会报错。

为节点维护独立的窗口 使用 `roslaunch` 的一个缺点是所有的节点共享一个终端，而使用 `roslaunch` 的时候，每个节点都有一个独立的终端。这对于不接收控制台输入，只生成日志消息的节点是可行的（通常也是有帮助的），但是对于依赖控制台输入的节点来说，例如 `turtle_teleop_key`，更倾向于为节点保留独立的终端。

幸运的是，`roslaunch` 提供了一个实现该目的的简洁方法——对节点元素使用启动前缀（`launch-prefix`）属性：

`Launch-prefix="command-prefix"`

事实上，`roslaunch` 在启动节点时的内部工作原理是调用相应的命令行工具，即 `roslaunch`。启动前缀的主要思想是在其命令行前面添加给出的命令行前缀。在我们的例子中，遥操作节点使用了这个功能：

`launch-prefix="xterm-e"`

在这个属性的作用下，节点元素和在命令行中输入下面的命令基本上是等价的：

`xterm -e roslaunch turtlesim turtle_teleop_key`

正如你所知，`xterm` 命令将打开一个简单终端窗口。参数 `-e` 告诉 `xterm` 在新打开的终端中执行该命令行的剩余部分（这里是 `roslaunch turtle_teleop_key`），运行的结果是基于纯文本的程序 `turtle_teleop_key` 出现在一个图形窗口中。

当然，启动前缀属性并不只局限于 `xterm`。该属性也可以用 来调试（通过 `gdb` 或 `valgrind`）或降低一个进程的优先级（通过 `nice`）⁵。

6.3 在命名空间内启动节点

在 5.2 节中，我们看到 ROS 支持相对名称，相对名称用到了默认命名空间的概念。对一个节点设置默认命名空间——这个过

⁵ http://wiki.ros.org/rqt_console

程通常叫做压入（**pushing down**）命名空间——的通常方法是使用一个启动文件，并对其节点元素配置命名空间（**ns**）属性：

`ns="namespace"`

```
1 <launch>
2   <node
3     name="turtlesim_node"
4     pkg="turtlesim "
5     type="turtlesim_node"
6     ns="sim1"
7   />
8   <node
9     pkg="turtlesim "
10    type="turtle_teleop_key"
11    name="teleop_key"
12    required="true "
13    launch -prefix="xterm -e"
14    ns="sim1"
15  />
16  <node
17    name="turtlesim_node"
18    pkg="turtlesim "
19    type="turtlesim_node"
20    ns="sim2"
21  />
22  <node
23    pkg="agitr "
24    type="pubvel"
25    name="velocity_publisher "
26    ns="sim2"
27  />
28 </launch>
```

表 6.3 启动文件 `doublesim.launch` 启动两个独立的 `turtlesim` 仿真。

一个仿真中海龟由随机生成的速度指令控制，另一个仿真中海龟是远程控制的。

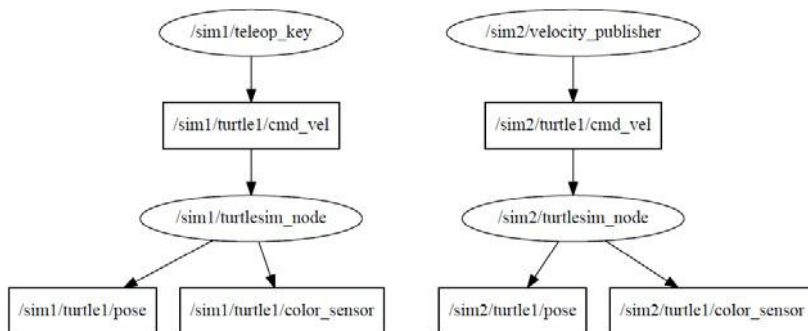


图 6.1 由 `doublesim.launch` 生成的节点（椭圆中）和话题（长方形中）。

表 6.3 是一个示例启动文件，该启动文件使用命名空间属性来创建两个独立的 `turtlesim` 模拟器。图 6.1 所示的是该启动文件生成的节点和话题。

从图中可以看到，前面我们多次见到的 `turtlesim` 节点相关的话题名称（`turtle1/cmd_vel`、`turtle1/color_sensor` 以及 `turtle1/pose`）已经从全局命名空间被移动到了两个独立的命名子空间 `/sim1` 和 `/sim2`。这种变动产生的原因是 `turtlesim_node` 节点的代码在创建 `ros::Publisher` 和 `ros::Subscriber` 对象时使用了像 `turtle1/pose` 这样的相对名称（而不是 `/turtle1/pose` 这样的全局名称）。

同样，启动文件中的节点名称是相对名称。在该例子中，两个节点有相同的相对名称——`turtlesim_node`。完全一样的相对名称并不会造成问题，因为他们对应的全局名称即 `/sim1/turtlesim_node` 和 `/sim2/turtlesim_node` 并不相同。

事实上 `roslaunch` 要求启动文件中的节点名称是基名称，即不涉及任何命名空间的相对名称。如果节点元素的名称属性中出现了全局名称，`roslaunch` 将会报错。

该例子与 2.8 节讨论的系统有一些相似性*。在两种情况中，我们都运行了多个 `turtlesim` 节点，然而它们的结果却很不相同。

在 2.8 节，我们只改变了节点名称，并把所有节点放在同一个命名空间。因此，两个 `turtlesim` 节点向同一个话题订阅和发布信息，没有办法单独同两个仿真中的一个进行交互。而在表 6.3 的例子中，我们将每一个节点都压入自己的命名空间中去，这样使两个模拟器真正地独立起来，从而可以给每个节点发送不同的速度指令。

▶▶ 在该例中，由 `ns` 属性定义的命名空间本身是相对名称。也就是说，我们在一个默认命名空间是全局命名空间“/”的启动文件内使用了 `sim1` 和 `sim2` 这两个名称。因此，两个节点的默认命名空间就解析到 `/sim1` 和 `/sim2`。

从技术层面看，完全可以做到为这个属性设置一个全局名称；然而，这是一个非常糟糕的想法。因为本质上，这和节点内使用全局名称的做法如出一辙。这样做将会阻止启动文件被压入到其他命名空间中，例如被另一个启动文件包含的时候。

6.4 名称重映射 (Remapping names)

除了相对名称和私有名称，ROS 节点还支持重映射 (`remapping`)，它可以从更精细的层面控制对所用节点名称的修

* 此外，既然我们已经看到了启动文件，你应该能够将一系列繁琐的 `roslaunch` 命令替换为简单的启动文件。

改⁶。重映射是基于替换的思想：每个重映射包含一个原始名称和一个新名称。每当节点使用重映射中的原始名称时，ROS客户端库就会将它默默地替换成其对应的新名称。

6.4.1 创建重映射

当启动一个节点的时候，有两种方法来创建重映射。

当使用命令行启动节点时，分别给出原始名称和新名称，中间由一个:=分开，如下所示：

```
original-name := new-name
```

例如，运行一个 `turtlesim` 的实例，如果想要把海龟的姿态数据发布到话题 `/tim` 而不是 `/turtle1/pose`，就可以使用如下命令：

```
roslaunch turtlesim turtlesim_node turtle1/pose:=tim
```

通过启动文件的方式，只需在启动文件内使用重映射 (`remap`) 元素即可⁷：

```
<remap from="original-name" to "new-name"/>
```

如果该属性在顶层，即作为 `launch` 元素的子元素出现，重映射将会应用到所有的后续节点。这些重映射元素也可以作为一个节点元素的子元素，如：

```
<node name="node-name" type="node-name">
```

```
  <remap from="original-name" to "new-name"/>
```

```
</node>
```

⁶ <http://wiki.ros.org/RemappingArguments>

⁷ <http://wiki.ros.org/roslaunch/XML/remap>

在这种情况下，给出的重映射只应用于其所在的节点。例如，上面的命令行本质上等价于如下结构的启动文件：

```
<node pkg="turtlesim" type="turtlesim_node"
name="turtlesim_node">
  <remap from="turtle1/pose"to "tim"/>

</node>
```

关于重映射的使用，有一点需要牢记：在 ROS 应用任何重映射之前，所有的名称都要先解析为全局名称，包括重映射中的原始名称和新名称。因此，虽然在重映射中出现的名称通常是相对名称，但当名称解析完成后，重映射就可以通过直接的字符串比较来实现，即在所有重映射中查找某个节点使用的原始名称。

6.4.2 反向海龟（Reversing a turtle）

下面我们举一个具体的例子来说明这些重映射是如何起作用的。考虑这样一个方案，我们还是使用 `turtle_teleop_key` 来控制 `turtlesim` 中海龟的运动，但是让每一个方向键的含义都翻转过来。也就是说，我们让左键控制顺时针旋转，右键控

制逆时针旋转，上键控制它倒退，下键控制它前进。这么做看起来也许有些不合情理，但是它却能够代表一类问题，即一个节点发布的消息一定要转化为另一个节点所期望的格式。

一个显而易见的方案是复制 `turtle_teleop_key` 的源代码，并对其进行修改，以达到我们想要的效果。但是这个方案令人非常不满，因为它要求用户首先理解代码，更糟糕的是，有可能造成代码重复。另一种方案是编写一个新的程序，利用它来把遥操作节点发送的速度指令进行反转。

```

1 // This program subscribes to turtle1/cmd_vel and
2 // republishes on turtle1/cmd_vel_reversed ,
3 // with the signs inverted .
4 #include <ros/ros.h>
5 #include <geometry_msgs/Twist.h>
6
7 ros::Publisher *pubPtr;
8
9 void commandVelocityReceived (
10 const geometry_msgs::Twist& msgIn
11 ) {
12     geometry_msgs::Twist msgOut;
13     msgOut.linear.x = -msgIn.linear.x ;
14     msgOut.angular.z = -msgIn.angular.z;
15     pubPtr->publish (msgOut);
16 }
17
18 int main( int argc , char * * argv ) {
19     ros::init ( argc , argv , " reverse_velocity " );
20     ros::NodeHandle nh;
21
22     pubPtr = new ros::Publisher (
23         nh.advertise <geometry_msgs::Twist>(
24             " turtle1/cmd_vel_reversed" ,
25             1000 ) );
26
27     ros::Subscriber sub = nh.subscribe (
28         " turtle1/cmd_vel" , 1000 ,
29         &commandVelocityReceived );
30
31     ros::spin();
32
33     delete pubPtr;
34 }

```

表 6.4 一个名为 reverse_cmd_vel 的能反转 turtlesim 速度指令的 C++ 程序。

表 6.4 所示的是能够完成这一功能的一个简短的程序：向 `turtle1/cmd_vel` 订阅信息，然后反转所有接收到消息中的线速度和角速度指令，并将反转后的指令通过话题 `turtle1/cmd_vel_reversed` 发布。

这里唯一复杂之处在于 `turtlesim` 仿真器事实上并没有订阅被反转的速度信息，这也是把该例子放到重映射这一节的原因。事实上，如果此时使用 `roslaunch` 指令启动上述三个节点，将得到如图 6.2 所示的计算图。从该图可以看出，系统不能得到想要的结果。因为速度指令仍然直接从 `teleop_turtle` 节点传递到 `turtlesim` 节点，海龟仍然按照正常的方式响应控制指令，而不会出现反转。

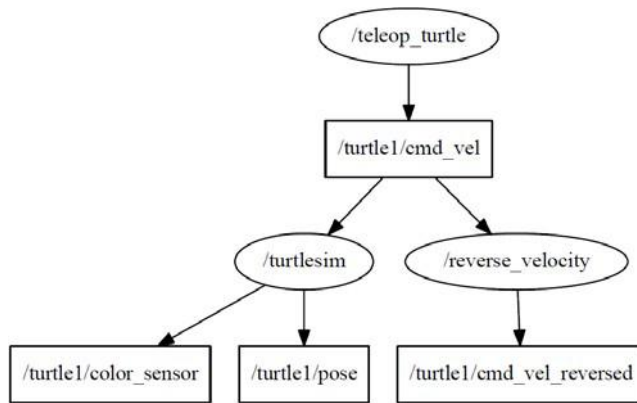


图 6.2 尝试使用 `reverse_cmd_vel` 来反转 `turtlesim` 时得到的 ROS 图。

这种情况下，一些节点向“错误”的话题订阅信息，这正是需要使用重映射的场合。在该例中，我们可以通过对 `turtlesim` 使用重映射来解决问题，用 `turtle1/cmd_vel_reversed` 来替换 `turtle1/cmd_vel`。表 6.5 所示的是启动三个节点的启动文件，其中 `turtlesim_node` 的节点元素包含了正确的重映射元素；图 6.3 是由此得到的正确的 ROS 计算图。

```
1 <launch>
2 <node
3   pkg="turtlesim "
4   type="turtlesim_node"
5   name="turtlesim" 6
6   >
7 <remap
8   from="turtle1 /cmd_vel"
9   to="turtle1 /cmd_vel_reversed" 10
10  />
11 </node>
12 <node
13   pkg="turtlesim "
14   type="turtle_teleop_key"
15   name="teleop_key"
16   launch -prefix="xterm -e"
17 />
18 <node
19   pkg="agitr "
20   type="reverse_cmd_vel"
21   name="reverse_velocity" 22
22 />
23 </launch>
```

表 6.5 启动文件 reverse.launch，启动一个被远程控制方向反转的 turtlesim。

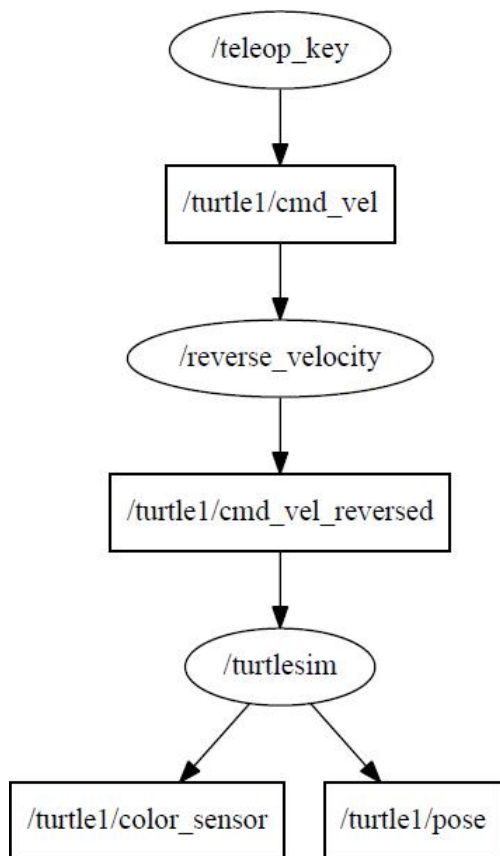


图 6.3 由 `reversed.launch` 得到的正确的 ROS 图。重映射元素使得节点之间能够恰当地连接。

6.5 启动文件的其他元素

本节介绍一些额外的 `roslaunch` 结构⁸。为了举例说明这些特征，我们使用表 6.6 中的启动文件。该启动文件根据启动方式的不同，可以启动两个或三个独立的 `turtlesim` 仿真器。

⁸ <http://wiki.ros.org/ROS/Tutorials/Roslaunchtipsforlargerprojects>

6.5.1 包含其他文件

如果想在启动文件中包含其他启动文件的内容（包括所有的节点和参数），可以使用包含（`include`）元素⁹：

```
<include file="path-to-launch-file">
```

此处 `file` 属性的值应该是我们想包含的文件的完整路径。由于直接输入路径信息很繁琐且容易出错，大多数包含元素都使用查找（`find`）命令搜索功能包的位置来替代直接输入路径：

```
<include file="$(find package-name)/launch-file-name">
```

如果给定的功能包存在，则上面小括号中的 `find` 及其参数将展开为这个功能包的路径，并在这个路径下找到想要的启动文件要容易得多。示例就是利用该技巧来包含我们之前例子的启动文件，即 `doublesim.launch`。



不要忘记当 `roslaunch` 命令行工具在搜索一个功能包中的启动文件时，会搜索功能包内的所有子目录。相反，这里讲的包含元素一定要给出确切的文件路径，而不能依赖它对子目录进行搜索。这个不同就解释了上面的包含元素有时会导致错误的原因——即使调用 `roslaunch` 命令行时用了相同的包名和启动文件名可以正常运行。

包含元素同样支持命名空间属性，可以将内容压入一个指定的命名空间中去：

```
<include file="..." ns="namespace"/>
```

应该说该设置是十分普遍的，尤其是当被包含进来的启动文件属于另一个功能包的时候，设置的操作应当独立于其他节点。

⁹ <http://wiki.ros.org/roslaunch/XML/include>

```
1 <launch>
2 <include
3   file="$(find agitr)/doublesim.launch" 4
4 />
5 <arg
6   name="use_sim3"
7   default="0"
8 />
9
10 <group ns="sim3" if="$(arg use_sim3)">
11 <node
12   name="turtlesim_node"
13   pkg="turtlesim"
14   type="turtlesim_node" 15
15 />
16 <node
17   pkg="turtlesim"
18   type="turtle_teleop_key"
19   name="teleop_key"
20   required="true"
21   launch-prefix="xterm -e"
22 />
23 </group>
24 </launch>
```

表 6.6 名为 `triplesim.launch` 的启动文件，用来说明 `group`、`include` 和 `arg` 参数。

6.5.2 启动参数（launch arguments）

为了使启动文件便于配置，`roslaunch`还支持启动参数，有时也简称为参数甚至 **args**，其功能有点像可执行程序中的局部变量¹⁰。这样做的优点是设置参数来描述节点在不同ROS会话中运行时可能需要改变的一小部分，从而避免代码重复。为了说明这一点，示例文件中使用了名为 `use_sim3` 的参数，用来确定启动三个还是两个 `turtlesim` 节点。



尽管在计算机领域术语中很多情况下 **parameter** 和 **argument** 都是可以互换的，但是在 ROS 中，二者的含义是非常不同的（译者注：在中文中一般都称为参数，更没有明显的区分，因此后面还是使用它们的英文原文说明问题）。**parameter** 是运行中的 ROS 系统使用的数值，存储在参数服务器（**parameter server**）中，每个活跃的节点都可以通过 `ros::param::get` 函数来获取 **parameter** 的值，用户也可以通过 `rosparam` 来获得 **parameter** 的值（参见第 7 章）。而 **argument** 只在启动文件内才有意义；他们的值是不能被节点直接获取的。

声明参数 为了声明一个参数，可以使用 **arg** 元素。

```
<arg name="arg-name">
```

然而这样的声明并不是必须的（除非你想要给它赋值或设置为默认值，见后续内容），但是这是一个好的做法，因为这样能使读者比较清楚启动文件需要哪些参数。

¹⁰ <http://wiki.ros.org/roslaunch/XML/arg>

参数赋值 启动文件中的每一个参数都要进行赋值。赋值有很多种方法，例如可以像下面在 `roslaunch` 命令行中提供该值：

```
roslaunch package-name launch-file-name arg-name:=arg-value
```

除此之外，你也可以使用以下两种语法，将参数值作为 `arg` 声明的一部分：

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

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

两者的唯一区别在于命令行参数可以覆盖默认值 `default`，但是不能覆盖参数值 `value`。在示例中，`use_sim3` 有默认值 0，但是可以用如下命令行对参数值进行修改：

```
roslaunch agitr riplesim.launch use_sim3:=1
```

如果我们想修改启动文件，用 `value` 替换 `default`，那么该命令会产生错误，因为由 `value` 设置的参数值是不能更改的。

获取参数值 一旦参数值被声明并且被赋值，你就可以利用下面的 `arg` 替换（`arg substitution`）语法来使用该参数值了：

```
$(arg arg-name)
```

每个该替换出现的地方，`roslaunch` 都将它替换成参数值。在示例中，我们在 `group` 元素中的 `if` 属性使用了一次 `use_sim3` 参数（稍后将会介绍 `if` 和 `group`）。

向包括的启动文件中发送参数值 目前已介绍的参数设定技巧的局限在于，它并未提供任何方法将参数传递到通过包含元素导入的次级启动文件中。这一点很重要，因为像局部变量一样，参数仅定义在对其进行声明的启动文件中，而不能被包含的启动文件“继承”。

该问题的解决方案就是将 **arg** 元素作为一个包含元素的子元素，如下所示：

```
<include file="path-to-launch-file">
  <arg name="arg-name" value="arg-value"/>
  ...
</include>
```

请注意，该 **arg** 元素的用法和之前我们看到的 **arg** 声明是不同的。在 **include** 标签之间的参数是属于被包含文件的，而不是它们出现的启动文件。因为目的是为被包括的启动文件设置响应的参数值，所以该处需要使用 **value** 属性。

一种常见的情况是两个启动文件（包含文件和被包含文件）有一些共同的参数，在这种情况下，我们希望参数在传递的时候不会改变。这样的元素在两个地方使用相同的参数名：

```
<arg name="arg-name" value="$(arg arg-name)">
```

在该例子中，第一个参数名字通常是指被包含文件中的参数，第二个参数名是指当前文件中的参数。这样做之后，给定参数在两个启动文件中有相同的值。

6.5.3 创建组（Creating groups）

这里介绍启动文件的最后一个特征，即组(**group**)元素，它提供了一种在大型启动文件内管理节点的便捷方式¹¹。具体来讲，组元素可以完成如下两种任务：

¹¹ <http://wiki.ros.org/roslaunch/XML/group>

组可以把若干个节点放入同一个命名空间内。

```
<group ns="namespace"/>
```

...

```
</group>
```

组内的每个节点都从给定的默认命名空间启动。

如果一个组元素内的某个节点有它自己的命名空间属性，并且其名称是（原则上也应该是）相对名称，那么该节点将会在一个默认命名空间内启动，这个默认的命名空间是将此节点命名空间嵌入到组元素命名空间的结果。这个规则和前面讲到的名称解析是相符的，并且这个规则也适用于组元素的嵌套。

组可以有条件地使能或禁用一个节点。

```
<group if="0 or 1"/>
```

```
</group>
```

如果 `if` 属性的值为 1，则组元素标签内的元素被正常包含。反之，如果 `if` 属性的值为 0，则标签内元素被忽略。还有一个 `unless` 属性的用法是相似的，只是意思相反。

```
<group unless="1 or 0"/>
```

...

```
</group>
```

当然，通常不会直接赋值 0 或 1。结合前面讲的参数替换技巧，可以使启动文件的可配置性变得更好。



请注意，只有 0 和 1 才是 **if** 和 **unless** 属性的合法取值。特别需要提醒的是，读者熟悉的布尔型运算符 **AND** 和 **OR** 在这里不能直接使用。

我们的示例中使用一个单独的组演示了这两种使用目的。该组既使用了命名空间属性将组内的两个节点放入命名空间 **sim3**，又使用了 **if** 属性实现了基于 **use_sim3** 参数值使能或禁用第三个仿真器。



注意，组元素并不是严格必须的。如果不使用组，我们也可以对可能包括到组内的每个元素单独设置 **ns**、**if** 和 **unless** 等属性。然而，使用组可以减少代码重复——命名空间和条件设置仅出现一次——并且使启动文件的结构更加清晰。

不幸的是，只有前面提到的三个属性可以通过组进行传递。例如，尽管我们很想，但不能为组元素设置 **output="screen"**，必须直接为每一个我们想要应用它的节点设置该属性。

6.6 展望

经过本章的介绍，我们已经了解了如何创建通过消息传递来进行通信的节点，如何在（可能有的）复杂配置下一一次性启动多个节点。下一章我们将介绍参数服务器，向大家展现提供节点部分配置信息的集中式方法。

第7章 参数[†]

学而不思则罔，思而不学则殆。

——孔子

本章中，我们通过调整参数配置节点。

除了前面介绍过的消息传递，ROS 还提供另一种参数

(**parameters**) 机制用于获取节点的信息。其主要思想是使用集中参数服务器 (**parameter server**) 维护一个变量集的值，包括整数、浮点数、字符串以及其他数据类型，每一个变量用一个较短的字符串标识^{1,2}。由于允许节点主动查询其感兴趣的参数的值，它们适用于配置那些不会随时间频繁变更的信息。

本章将介绍参数，并且演示如何通过命令行、节点内部以及启动文件访问它们。

7.1 通过命令行获取参数

让我们通过一些命令行工具来查看参数如何工作。

查看参数列表使用下面的命令可以查看所有参数的列表³：

```
rosparam list
```

Dashing:

```
ros2 param list
```

在作者的系统中，没有节点运行时（译者注：事实上有一个

[†] 本章由肖军浩、陈春玉翻译。

¹ <http://wiki.ros.org/roscpp/Overview/ParameterServer>

² <http://wiki.ros.org/ParameterServer>

³ <http://wiki.ros.org/rosparam>

<https://index.ros.org/doc/ros2/Tutorials/Parameters/Understanding-ROS2-Parameters/>

特殊节点，即节点管理器在运行)，输出结果如下：

```
/roscdistro
/roslaunch/uris/host_donatello_38217
/rosversion
/run_id
```

这里每一个字符串都是一个名称，具体来说是全球计算图源名称（详见第 5 章），且在参数服务器中与某些值相关联。

在当前版本的 ROS 中，参数服务器是节点管理器的一部分，因此，它总是通过 **roscore** 或者 **roslaunch** 自动启动。在所有情况下，参数服务器都能在后台正常工作，因此没有必要去花心思考虑它。然而，需要铭记的是，所有的参数都属于参数服务器而不是任何特定的节点。这意味着参数——即使是由节点创建的——在节点终止时仍将继续存在。

查询参数 向参数服务器查询某个参数的值，可以使用 **roscparam get** 命令：

```
roscparam get parameter_name
```

Dashing:

```
ros2 param get <node_name> <parameter_name>
```

例如，如下命令将读取参数/roscdistro 的值：

```
roscparam get /roscdistro
```

Dashing:

```
ros2 param get /turtlesim background_g
```

其输出为字符串 **indigo**，这并不奇怪（译者注：**roscdistro** 指 ROS 的版本，如果读者使用其他的发行版，将产生不同的输出）。除此以外，还可以检索给定命名空间中的每一个参数的值，其命令为：

```
rosparam get namespace
```

例如，通过查询全局命名空间，我们可以一次性看到所有参数的值：

```
rosparam get /
```

在作者的计算机上输出为：

```
rostdistro: indigo
roslaunch:
uris: host_donatello_38217:'http://donatello:38217/'
rosversion: 1.11.9
run_id: e574a908-70c5-11e4-899e-60d819d10251
```

设置参数 如下命令用于设置参数的值：

```
rosparam set parameter_name parameter_value
```

Dashing:

```
ros2 param set <node_name> <parameter_name> <value>
```

该命令可以修改已有参数的值或者创建一个新的参数。例如，以下命令可以创建一系列字符串参数，用以存储一组卡通鸭子的颜色偏好：

```
rosparamset/duck_colors/hueyred
rosparamset/duck_colors/dewey blue
rosparamset/duck_colors/louie green
rosparamset/duck_colors/webby pink
```

Dashing:

```
ros2 param set /turtlesim background_r 150
```

另外，我们可以同时设置同一命名空间中的几个参数：

```
rosparam set namespace values
```

这里要以 **YAML** 字典的形式表示参数和对应值的映射关系。下面的例子和前面四行命令具有同样的作用：

```
rosparam set /duck_colors "huey: red
```

```
dewey: blue
```

```
louie: green
```

```
webby: pink"
```

需要注意的是，这种语法需要在命令中使用换行符。当然，这并不会造成什么问题，因为第一个引号告诉 **bash** 命令尚未完成。当在引号内按下回车时，终端将插入一个换行符而不是执行命令。



冒号后的空格是非常重要的，以确保 **rosparam** 将其作为一个 **/duck_colors** 命名空间内的参数集，而不是全局命名空间中的单个字符串参数 **duck_colors**。

创建和加载参数文件 为了以 **YAML** 文件的形式存储命名空间中的所有参数，可以使用 **rosparam dump** 命令：

```
rosparam dump filename namespace
```

Dashing:

```
ros2 param dump <node_name>
```

```
ros2 param dump /turtlesim
```

与 **dump** 相反的命令是 **load**，它从一个文件中读取参数，并

将它们添加到参数服务器：

```
rosparam load filename namespace
```

Dashing:

```
ros2 run <package_name> <executable_name> --ros-args --  
params-file <file_name>
```

```
ros2 run turtlesim turtlesim_node --ros-args --params-  
file ./turtlesim.yaml
```

对于这些命令，命名空间参数是可选的，默认值为全局命名空间 (/)。存储和加载的组合可以用来测试，因为它提供了一种快捷方式获取一定时间内的参数“快照”，并且可以进行场景复现。

7.2 例：turtlesim 中的参数

作为一个参数的具体例子，让我们来看看在 turtlesim 中如何使用它们的。如果你启动 roscore 和 turtlesim_node，然后查询 rosparam 列表，会看到像下面这样的输出：

```
/background_b  
  
/background_g  
  
/background_r  
  
/rostdistro  
  
/roslaunch/uris/host_donatello_59636  
  
/rosversion  
  
/run_id
```

Dashing:

> <https://index.ros.org/doc/ros2/Tutorials/Parameters/Understanding-ROS2-Parameters/>

我们已经看到过后面四个由节点管理器创建的参数。此外，这里还看到 turtlesim_node 节点创造了三个参数。它们的名称表

明它们指定在 `turtlesim` 窗口中使用背景色，分别为红色、绿色和蓝色通道。

这说明了节点有时可以创建和修改参数值。在这种情况下，`turtlesim_node` 将设置这三个参数作为其初始化的一部分。从这个角度讲，`turtlesim_node` 的行为是非典型的，因为它将改变那些已经设置好的相同参数的值。其结果是，不管我们用什么方法指定一个不同的初始背景颜色，每一个 `turtlesim_node` 启动后都采用相同的蓝色背景，至少在短时间内。



一个好的策略，也是演示真正的 ROS 节点如何工作的更好案例，是 `turtlesim` 首先测试这些参数是否存在，当且仅当这些参数不存在时，才指定默认的蓝色。

获取背景颜色 可以使用 `rosparam get` 命令获取背景参数的值：

```
rosparam get/background_r
```

```
rosparam get/background_g
```

```
rosparam get/background_b
```

这些命令的返回值分别是 69、86 和 255。由于这些值是比较小的整数，让人容易联想到，而且确实是，每个通道是一个 8 位整数，范围从 0 到 255。因此，`turtlesim` 默认其背景颜色为 (69,86,255)，这正是我们平时所称的深蓝色。

设置背景颜色 如果想把背景颜色由蓝色改变成明亮的黄色，在 `turtlesim` 节点启动后，可以尝试通过改变参数的数值来实现：

```
rosparam set/background_r 255
```

```
rosparam set/background_g 255
```

```
rosparam set/background_b 0
```

然而，即使我们设置完参数后，背景颜色仍然是原来的颜色。这是为什么呢？这是因为只有当 `turtlesim_node` 的 `/clear` 服务被调用时，它才会从参数服务器读取这些参数的值。调用这个服务可以使用以下命令：

```
rosservice call /clear
```

在这个服务调用完成后，背景颜色将会发生改变，变化效果如图 7.1 所示。

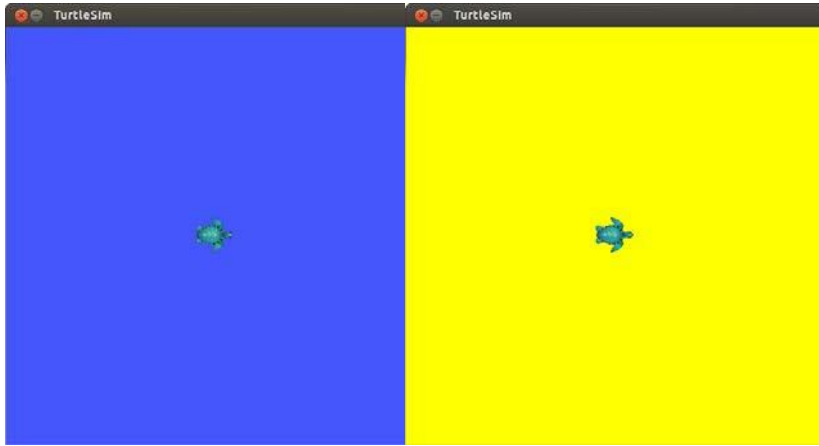


图 7.1 改变turtlesim 节点的背景颜色，左图为改变前，右图为改变后。

需要引起高度注意的是，更新的参数值不会自动“推送”到节点。恰恰相反，如果节点关心它的一些或者所有参数是否改变，必须明确向参数服务器请求这些参数的值。同样地，如果希望改变某个活跃节点的值，我们必须知道该节点如何或是否重新查询参数。（通常来讲，turtlesim中没有使用，解决方案是基于一个叫做dynamic_reconfigure的子系统，但该命令不在讨论范围内⁴。）

对一个节点来说，可以请求参数服务器在参数更新后将新参数值发给它，这个功能可以通过 `ros::param::getCached` 命令替换 `ros::param::get` 来实现。然而，这种方法只是为了提高效率，并没有消除节点检查参数值的需要。

⁴ http://wiki.ros.org/dynamic_reconfigure

7.3 使用 C++ 获取参数

使用 ROS 参数的 C++ 接口是相当简单的⁵:

```
void ros::param::set(parameter_name, input_value); bool  
ros::param::get(parameter_name, output_value);
```

在这两种情况下，参数名是一个字符串，它可以是全局的、相对的或者是私有的。set 函数中的输入值 input_value 可以是 std::string、bool、int 或 double 类型；get 函数的输出值应该是上述某个类型的变量（通过引用传递）。如果参数值读取成功，则 get 函数返回 true；如果出现了问题，通常表示所请求的参数还没有被指定一个值，则该函数返回 false。为了查看函数的行为，让我们来看看下面两个实例。

> 表 7.1 说明了 ros::param::set 的功能，它为 turtlesim 背景颜色三个参数赋值。本程序的代码在启动 turtlesim 节点前等待 /clear 服务调用结束，从而确保 turtlesim 不会覆盖这里设置的值。并且通过调用 /clear 服务强制 turtlesim 读取我们设置的参数值。（本章关注的重点是参数本身；关于服务的详细介绍见第 8 章。）

> 表 7.2 是命令 ros::param::get 的例程。作为原 pubvel 例程（表 3.4）的拓展，这里读取私有浮点参数 max_vel 的值，并使用这个参数值来调整随机生成线速度的大小。

该程序需要其私有命名空间中 max_vel 的参数值，且这个参数值必须在程序启动之前设置：

⁵ http://wiki.ros.org/roscpp_tutorials/Tutorials/Parameters

```
1 // This program waits for a turtlesim to start up , and
2 // changes its background color .
3 #include <ros/ros .h>
4 #include<std_srvs/Empty.h>
5
6 int main( int argc , char ** argv ) {
7   ros::init(argc,argv,"set_bg_color");
8   ros::NodeHandle nh ;
9
10  // Wait until the clear service is available , which
11  // indicates that turtlesim has started up , and has
12  // set the background color parameters .
13  ros::service::waitForService("clear");
14
15  // Set the background color for turtlesim ,
16  // overriding the default blue color .
17  ros::param::set("background_r",255);
18  ros::param::set("background_g",255);
19  ros::param::set ("background_b" , 0);
20
21  // Get turtlesim to pick up the new parameter values .
22  ros::ServiceClient clearClient
23  =nh.serviceClient<std_srvs::Empty>("/clear");
24  std_srvs::Empty srv;
25  clearClient.call(srv);
26
27 }
```

表 7.1 用 C++编写的 set_bg_color.cpp 例程，设置 turtlesim 窗口背景颜色。

```

1 // This program publishes random velocity commands, using
2 // a maximum linear velocity read from a parameter.
3 #include <ros/ros.h>
4 #include <geometry_msgs/Twist.h>
5 #include <stdlib.h>
6
7 int main( int argc , char ** argv ) {
8   ros::init(argc,argv,"publish_velocity");
9   ros::NodeHandle nh ;
10  ros::Publisher pub = nh.advertise <geometry_msgs::Twist>(
11    "turtle1/cmd_vel" , 1000) ;
12  srand( time(0) );
13
14  // Get the maximum velocity parameter .
15  const std::string PARAM_NAME = "~max_vel";
16  double maxVel;
17  bool ok = ros::param::get(PARAM_NAME, maxVel);
18  if ( !ok ) {
19    ROS_FATAL_STREAM("Could _ not _ get _ parameter _ "
20      << PARAM_NAME) ;
21    exit (1) ;
22  }
23
24  ros::Rate rate(2);
25  while ( ros::ok() ) {
26    // Create and send a random velocity command.
27    geometry_msgs::Twist msg ;
28    msg.linear.x = maxVel * double ( rand () ) / double ( RAND_MAX );
29    msg.angular.z = 2 * double ( rand () ) / double ( RAND_MAX ) - 1;
30    pub.publish(msg);
31
32    // Wait until it's time for another iteration .
33    rate.sleep ();
34  }
35 }

```

表 7.2 用 C++ 编写的 `pubvel_with_max.cpp` 例程，通过从参数中读取最大线性速度来扩展原来的 `pubvel.cpp` 例程。

```
rosparam set /publish_velocity/max_vel 0.1
```

如果这个参数不存在，那么，该程序将会报错并且终止。

通过在命令行中使用类似重映射的语法为节点的私有参数赋值在技术上是可行的（但是有些混乱），可以通过给参数名添加下划线前缀实现，如下所示：

```
_param-name:=param-value
```

这种命令行参数将被转换为 `ros::param::set` 的调用，通过

`ros::init` 将“_”替换为“~”构成私有参数的形式。例如，我们可以使用下面的命令成功启动 `pubvel_with_max` 节点：

```
roslaunch agitr pubvel_with_max _max_vel:=1
```

7.4 在启动文件中设置参数

另外一种常用的参数设置方法是在启动文件中设置。设置参数可以使用 `param` 元素请求 `roslaunch` 设置参数值⁶：

```
<param name="param-name" value="param-value" />
```

这个元素，正如你所猜测的，将给定的值赋给给定名称的参数。通常，这个参数名是相对的。例如，下面的启动文件片段相当于第 129X 页的 `rosparam` 命令集：

```
<group ns="duck_colors">
```

```
<param name="huey" value="red" />
```

```
<param name="dewey" value="blue" />
```

⁶<http://wiki.ros.org/roslaunch/XML/param>

```
<param name="louie" value="green" />
```

```
<param name="webby" value="pink" />
```

```
</group>
```

设置私有参数 另一个可选方法是在节点元素中包含`param` 元素。

```
<node ...>
```

```
<param name="param-name" value="param-value" />
```

```
...
```

```
</node>
```

在该结构下，参数名将被当做该节点的私有名称。

这是 ROS 命名规则的一个例外。作为节点元素的子集时，`param` 元素中给出的参数名总是被当做私有名称解析，无论它们是否以`~`或者`/`开始。

例如，我们可以使用下面的代码启动 `pubvel_with_max` 节点，并且正确设置其私有参数 `max_vel`：

```
<node
```

```
  pkg="agitr" type="pubvel_with_max"
  name="publish_velocity"
```

```
/>
```

```
<param name="max_vel" value="3" />
```

```
</node>
```

表 7.3 中展示了一个完整的启动文件，该文件启动了 1 个 `turtlesim` 和 2 个案例。其结果应显示一个 `turtlesim` 龟迅速穿过黄色的背景。

```
1 <launch>
2 <node
3 pkg="turtlesim "
4 type="turtlesim_node"
5 name="turtlesim "
6 />
7 <node
8 pkg="agitr "
9 type="pubvel_with_max"
10 name="publish_velocity "
11 >
12 <param name="max_vel" value="3"/> 13
</node>
14 <node
15 pkg="agitr "
16 type="set_bg_color"
17 name="set_bg_color"
18 />
19 </launch>
```

表 7.3 启动文件 `fast_yellow.launch`，启动 7.1 和 7.2 中的例子并且设置 `max_vel` 参数。

在文件中读取参数 最后，启动文件也支持与 `rosparam load` 等价的命令，可以一次性从文件中加载多个参数⁷：

```
<rosparam command="load" file="path-to-param-file" />
```

这里列出的参数文件通常是通过 `rosparamdump` 命令创建的。与其他特定文件的引用相同（例如 6.5.1 节的 `include` 元素），使用查找替换来指定功能包的相对路径是常见做法：

```
<rosparam
```

⁷ <http://wiki.ros.org/roslaunch/XML/rosparam>

```
command="load"
```

```
file="$(find package-name)/param-file"
```

```
/>
```

与 `rosparam load` 一样，这个功能有助于测试，因为它允许用户重现在过去的某个时间有效的参数。

7.5 展望

参数的思想虽然简单，但大大提高了 ROS 节点的灵活性和可配置性。下一章将讨论 ROS 的最后一种通信机制——即服务，它实现了一对一的双向信息流。

补充：关于 ROS2 参数的更多内容参考：

> <https://index.ros.org/doc/ros2/Tutorials/Parameters/Understanding-ROS2-Parameters/>

第8章 服务[†]

不积跬步，无以至千里；不积小流，无以成江海。

——荀子

本章，我们讨论如何调用服务并对服务请求进行响应。

在第2章和第3章中，我们讨论了消息如何在节点之间传递。消息传递机制尽管是ROS系统中节点通信的主要方法，但确实受到了一定的限制。本章引入另一种用于通信的方法，称之为服务调用（**service calls**）。服务调用与消息的区别主要体现在两个方面。

- > 服务调用是双向的，一个节点给另一个节点发送信息并等待响应，因此信息流是双向的。作为对比，当消息发布后，并没有响应的概念，甚至不能保证系统内有节点订阅了这些消息。

- > 服务调用实现的是一对一通信。每一个服务由一个节点发起 对这个服务的响应返回同一个节点。另一方面，每一个消息都和一个话题相关，这个话题可能有很多的发布者和订阅者。

除上述（非常重要的！）区别外，服务与消息十分相似。在本章中，我们将弄清如何从命令行查看和调用服务，还有如何按照服务的客户或者服务器来写节点。

除了服务之外还需掌握行动（**action**），每次行动包含多次信息交互。ROS2服务和行动教程链接如下：

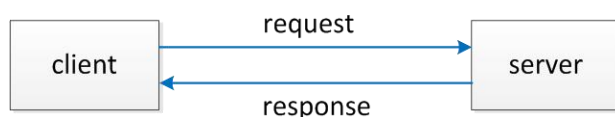
- > <https://index.ros.org/doc/ros2/Tutorials/Services/Understanding-ROS2-Services/>

- > <https://index.ros.org/doc/ros2/Tutorials/Understanding-ROS2-Actions/>

8.1 服务的专用术语

下面是服务调用的基本信息流：

[†]本章由肖军浩、魏翔宇翻译。



其过程是一个客户端（**client**）节点发送一些称为请求（**request**）的数据到一个服务器（**server**）节点，并且等待回应。服务器节点接收到请求后，采取一些行动（计算、配置软件或硬件、改变自身行为等），然后发送一些称为响应（**response**）的数据给客户端节点。

请求和响应数据携带的特定内容由服务数据类型（**service data type**）来决定，它与决定消息内容的消息类型是类似的（回忆 2.7.2 节）。同消息类型一样，服务数据类型也是由一系列域构成的。唯一的区别就在于服务数据类型分为两部分，分别表示请求（客户端节点提供给服务器节点）和响应（服务其节点反馈给客户端节点）。

8.2 从命令行查看和调用服务

尽管服务通常由节点内部的代码调用，但是也确实存在一些命令行工具来与之交互。利用这些工具开展实验，能够帮助我们更容易地理解服务调用的工作原理。

列出所有服务 通过下面这条指令，可以获取目前活跃的所有服务¹：

```
rosservice list
```

Dashing:

```
ros2 service list
```

在作者的电脑上，只有一个 **turtlesim** 节点运行，服务列表如下所示：

¹<http://wiki.ros.org/ROS/Installation>

```
/clear  
/kill  
/reset  
/rosout/get_loggers  
/rosout/set_logger_level  
/spawn  
/turtle1/set_pen  
/turtle1/teleport_absolute  
/turtle1/teleport_relative  
/turtlesim/get_loggers  
/turtlesim/set_loggers_level
```

Dashing:

```
/clear  
/kill  
/reset  
/spawn  
/teleop_turtle/describe_parameters  
/teleop_turtle/get_parameter_types  
/teleop_turtle/get_parameters  
/teleop_turtle/list_parameters  
/teleop_turtle/set_parameters  
/teleop_turtle/set_parameters_atomically  
/turtle1/set_pen  
/turtle1/teleport_absolute  
/turtle1/teleport_relative  
/turtlesim/describe_parameters  
/turtlesim/get_parameter_types  
/turtlesim/get_parameters  
/turtlesim/list_parameters  
/turtlesim/set_parameters  
/turtlesim/set_parameters_atomically
```

每一行都表示一个当前可以调用的服务名。服务名是计算图源名称，同其他资源名称一样，可以划分为全局的、相对的或者私有的名称。`rosservice list` 命令的输出是所有服务的全局名称。

本例中的服务以及很多 ROS 服务总的来讲可以划分为两个基本类型。

一些服务，例如上表中 `get_loggers` 和 `set_logger_level` 服务，是用来从特定的节点获取或者向其传递信息的。这类服务通常将节点名用作命名空间来防止命名冲突，并且允许节点通

过私有名称来提供服务，例如 `~get_loggers` 或者 `~set_logger_level`（参照4.5了解loggers和logger level的详情）。

其他服务表示更一般的不针对某些特定节点的服务。例如，名为 `/spawn` 的服务用于生成一个新的仿真海龟，是由 `turtlesim` 节点提供的。但是在不同的系统中，这个服务完全可能由其他节点提供；当我们调用 `/spawn` 时，我们只关心是否有一个新的海龟出现，而不关心具体哪个节点起作用这种细节。上表列出的所有服务，除了 `get_loggers` 和 `set_logger_level`，都可以归入此类。这类服务都有特定的名称来描述它们的功能，却不会涉及任何特定节点。

查看某个节点的服务类型 要查看一个特定节点提供的服务，使用 `rostopic info` 命令：

```
rostopic info node-name
```

下面的例子给出了这条命令在 `turtlesim` 节点上输出结果的相应部分：

Services:

```
*/turtle1/teleport_absolute
*/turtlesim/get_loggers
*/turtlesim/set_logger_level
*/reset
*/spawn
*/clear
*/turtle1/set_pen
*/turtle1/teleport_relative
*/kill
```

可见，大多数目前可以调用的服务都是 `turtlesim` 节点提供的，这当然也不意外（仅有的两个例外是由 `rostopic` 提供的两个日志服务）。

查找提供服务的节点 要完成这个反向查询，即查找提供给定服务的节点，可以使用这条指令：

```
rostopic node service-name
```

正如期望的，当查询关于任意由 `roscallinfo /turtlesim` 指令列出的服务时，这条指令将给出 `/turtlesim`；而查询其他两条服务时，返回的节点是 `/rosout`。

查找服务的数据类型 可以利用下面的指令来确定一个服务的数据类型：

```
rosservice info service-name
```

例如，在终端中输入下面这条命令

```
rosservice info /spawn
```

得到的结果是：

```
Node: /turtlesim
```

```
URI: rosrpc://Donatello:47441 Type: turtlesim/Spawn
```

```
Args: x y theta name
```

可以看到，`/spawn` 服务的数据类型是 `turtlesim/Spawn`。同消息类型类似，服务数据类型有两个部分，其一声明拥有此数据类型的包，其二声明类型本身：

<code>turtlesim</code>	+	<code>Spawn</code>	<code>turtlesim/Spawn</code>
功能包名		类型名	服务数据类型

服务数据类型总是按照这些完整命名来引用。

查看服务数据类型 当服务的数据类型已知时，我们可以使用 `rossrv` 指令来获得此服务数据类型的详情：

```
rossrv show service-data-type-name
```

Dashing:

```
ros2 service type <service_name>
```

```
ros2 service list -t
```

```
ros2 interface show <type_name>.srv
```

```
ros2 interface show turtlesim/srv/Spawn.srv
```

例如，

```
rossrv show turtlesim/Spawn
```

Dashing:

```
ros2 interface show turtlesim/srv/Spawn.srv
```

输出如下结果：

```
float32 x float32 y float32 theta string name
---
string name
```

在这里，短横线（---）之前的数据是请求项，这是客户节点发送到服务节点的信息。短横线之后的所有字段是响应项，或者说是服务节点完成请求后发送回请求节点的信息。



一定要谨慎地区别 **rosservice** 和 **rossrv**。前者是用来与目前由某些节点提供的服务进行交互的。后者名称来源于用来声明服务数据类型文件的扩展名 **.srv**，是用来查看服务数据类型，无论现有可获得的服务有无此种类型。其区别与 **rostopic** 和 **rosmmsg** 指令的区别类似：

有一点要引起注意，服务数据类型中的请求或响应字段可以为空，甚至两个字段可以同时为空。例如，在由 **turtlesim_node** 提供的 **/reset** 服务中，由数据类型 **std_srvs/Empty** 定义，即请求和响应字段均为空。这一点大致和 C++ 中函数可以接受空的参数并返回 **void** 相类似。虽然没有信息进出，但仍然可能有用（这里指的是副作用）。

	<i>Topics</i>	<i>Services</i>
<i>active things</i>	rostopic	rosservice
<i>data types</i>	rosmmsg	rossrv

从命令行调用服务 为了直观地观察服务是如何工作的，你可以使用下面这条指令，从命令行中调用它们：

```
rosservice call service-name request-content
```

Dashing:

```
ros2 service call <service_name> <service_type> <arguments>
```

```
ros2 service call /clear std_srvs/srv/Empty
```

请求内容部分应该列出请求每一个域的值，正如 `rossrv show`

指令所展示的。下面是一个例子：

```
rosservice call /spawn 3 3 0 Mikey
```

Dashing:

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"
```

这条服务调用的效果是在现有仿真器中，位置 (x,y) (3,3)处创建一个名为“Mikey”的新海龟，其朝向角度 0。



这只新的海龟有它自己的资源集，包括 `cmd_vel`、`pose`、`color_sensor` 话题、`set_pen` 服务、`teleport_absolute` 服务、`teleport_relative` 服务。在本例中，这些新的资源在一个名为 **Mikey** 的命名空间中。这些资源在常用的 **turtle1** 命名空间之外，也符合其他节点想要独立控制这些海龟的需求。这一点很好地阐明了命名空间可以有效地阻止命名冲突。

`rosservice call` 指令的输出展示了服务器节点的响应数据。例如在上例中，响应应该是：

```
name: Mikey
```

在这个实例中，服务器节点将新海龟的名字作为响应的一部分。

除了发送响应数据，服务器节点同时也告诉客户服务调用成

功与否。例如，在 `turtlesim` 中，每一只海龟必须拥有一个唯一的名字。如果我们运行上面的 `rosservice call` 例子两次，第一次调用是成功的，但是第二次会产生如下所示的错误：

ERROR: service [/spawn] responded with an error:

错误产生的原因是我们试图创建两只同名的海龟。

补充：ROS2可以重名.....

读者可能有些纳闷，为什么这个错误是以冒号结尾的？这是由于 `turtlesim` 以一个空的错误信息应答。事实上，当服务调用错误发生时，ROS 的内在机制能够返回错误信息的消息字符串。但是，`turtlesim` 正在使用的 C++ 的客户库难以提供非空的错误信息。

8.3 客户端程序

从命令行调用服务对于系统检查和其他只要偶尔完成的事件是非常方便的，但是显然能够从自己代码调用服务会更有帮助²。表 8.1 展示了一个如何完成上述调用的简短例子。该例说明了一个服务的客户端程序所需的基本要素。

声明请求和响应的类型 就像消息类型一样（参照 3.3.1 节），每一种服务数据类型都对应一个我们必须包含的相关 C++ 头文件：

```
#include <package_name/type_name.h>
```

在例子中，我们声明

```
#include <turtlesim/Spawn.h>
```

来包含名为 `turtlesim::Spawn` 类的定义，这个类定义了我们想要调用的服务的数据类型，包括全部请求和响应字段。

创建客户端对象 在按照节点要求初始化后（通过调用 `ros::init` 和创建一个 `NodeHandle` 对象），我们的程序必须创建一个类型为

² [http://wiki.ros.org/ROS/Tutorials/WritingServiceClient\(c++\)](http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(c++))

`ros::ServiceClient` 的对象，实际上是这个对象完成服务调用。按照如下声明一个 `ros::ServiceClient` 对象：

```
ros::ServiceClient client =  
    node_handle.serviceClient<service_type>( service_name);
```

这条指令有三个重要部分。

`node_handle` 是常用的 `ros::NodeHandle` 对象，这里我们将调用它的 `serviceClient` 方法。

`service_type` 是我们在上面头文件中定义的服务对象的数据类型，在本例中，类型为 `turtlesim::Spawn`。

`service_name` 是一个字符串，说明了我们想要调用的服务名称。再次强调，这应当是一个相对名称，虽然也可以声明为全局名称。例子中使用的是相对名称“`spawn`”。

一般来讲，创建这个对象的过程并不占用多少计算量，因为除了存放之后想调用的服务详情外，并不需要做太多工作。



值得注意的是，与创建类似的 `ros::Publisher` 对象相比，创建 `ros::ServiceClient` 对象不需要队列大小。之所以会造成这种差异，是因为服务调用直到响应抵达后才会返回。由于客户端要等待服务调用完成，因此没有必要维持一个发送服务调用的队列。

创建请求和响应对象 一旦有了 `ros::ServiceClient` 实例，下一步就是创建一个请求对象来容纳送往服务器的数据。我们上面包含的头文件中分别定义了请求和响应的类，命名为 `Request` 和 `Response`。这些类必须通过功能包名称和服务类型来引用，如下所示：

```

1//This program spawns a new turtlesim turtle by calling
2 // the appropriate service .
3 #include <ros/ros.h>
4
5//The srv class for the service.
6 #include <turtlesim/Spawn.h>
7
8 int main( int argc , char ** argv ) {
9   ros::init(argc,argv,"spawn_turtle");
10   ros::NodeHandle nh ;
11
12// Create a client object for the spawn service . This
13// needs to know the data type of the service and its
14 // name.
15   ros::ServiceClient spawnClient
16   =nh.serviceClient<turtlesim::Spawn>("spawn");
17
18// Create the request and response objects.
19   turtlesim::Spawn::Request req ;
20   turtlesim::Spawn::Response resp ;
21
22// Fill in the request data members.
23   req.x =2;
24   req.y =3;
25   req.theta = M_PI/2;
26   req.name="Leo";
27
28// Actually call the service. This won't return until
29 // the service is complete .
30   bool success=spawnClient.call(req,resp);
31
32// Check for success and use the response .
33   if ( success ) {
34     ROS_INFO_STREAM("Spawned a turtle named "
35 < << resp.name) ;
36   } else {
37     ROS_ERROR_STREAM("Failed to spawn.");
38   }
39
40}

```

表 8.1 spawn_turtle.cpp 程序调用服务。

```
package_name::service_type::Request  
package_name::service_type::Response
```

每一个类都有数据成员匹配服务类型类型中的域（回想下 `rossrv show` 能够为我们列出这些域和它们的数据类型。）。这些域按照与消息域同样的方式来映射到 C++ 数据类型。`Request` 类的构造函数给这些域提供的是无意义的缺省值，因而我们要给每个域重新赋值。在例子中，我们创建了一个 `turtlesim::Spawn::Request` 对象，并且给它的 `x`、`y`、`theta` 和 `name` 域赋值。

同时我们还需要一个 `Response` 对象，在例子中是一个 `turtlesim::Spawn::Response` 对象，但是由于该信息应该来自于服务器节点，因此不应该试图给它的数据成员赋值。



在服务类型的头文件中也可以定义一个单独类（实际上是一个结构体）叫做

```
package_name::service_type
```

它同时拥有 `Request` 和 `Response` 作为数据成员。来源于这个类的对象通常称为 `srv`。如果你喜欢，如同很多在线教程偏爱的做法一样，可以传递这个类的对象给下面要介绍的调用方法，而不用使用独立的 `Request` 和 `Response` 对象。

调用服务一旦拥有了一个 `ServiceClient`、一个完整的 `Request` 以及 `Response`，我们就可以调用服务了：

```
bool success = service_client.call(request,reponse);
```

这个方法实际上完成了定位服务器节点、传输请求数据、等待响应和存储响应数据等一系列工作。

这个调用方法返回一个布尔值来表明服务调用是否成功完成。**ROS** 的机制可能导致调用失败，例如试图调用一个没有任何节点提供的服务，或者由于某个特定服务自身的原因。在例子中，调用失败通常意味着另一只有着被请求名称的海龟已经存在。



危险符：一个常见的错误是忘记核对服务调用的返回值。此时如果服务调用失败的话，可能导致难以预料的错误。只需要一两分钟就可以添加核对返回值的代码，当服务调用失败的时候，可以调用 **ROS_ERROR_STREAM** 输出错误信息。这点时间投资很有可能会带来可观的收益，即将来的调试变得更为容易。

快进符：在默认情况下，查找和连接服务器节点的过程是在调用方法内部发生的。这个连接仅在此次服务调用期间有效，在调用返回后将被关闭。**ROS** 也支持持续服务客户端的概念，**ros::ServiceClient** 构造函数建立与服务器节点的连接，然后可以被这个客户端对象的所有后续调用重用。通过传递 **true** 值给构造函数的第二个参数，可以创建一个持续的服务客户端（在前面的例子中使用了缺省值 **false**）：

```
Ros::ServiceClient client =  
node_handle.advertise<service_type>(service_name, true);
```

在线文档中对持续客户端的应用是轻微抵制的³，因为这么做并不能提升多少系统性能，作者的非正式实验表明，性能只提升了约 10%，然而其代价是导致系统在重启和修改服务器节点时的鲁棒性变差。

³ http://www.ros.org/doc/api/roscpp/html/classros_1_1NodeHandle.html

在服务调用顺利完成后，可以从传递给调用方法的 `Request` 对象中获得返回的响应数据。在例子中，响应仅仅包含来自于请求的名称域。

声明依赖 前面已经介绍完所有客户端代码相关的知识点。但是要使 `catkin_make` 正确编译一个客户端程序，必须保证程序的功能包声明了对定义服务类型的功能包的依赖。这种依赖，同消息类型的使用一样（参照 3.3.3 节），需要编辑 `CMakeLists.txt` 和清单文件 `package.xml`。为了编译例子中的程序，我们必须保证 `CMakeLists.txt` 中的 `find_package` 行涉及了 `turtlesim` 功能包，如下所示：

```
find_package(catkin REQUIRED COMPONENTS roscpp turtlesim)
```

在 `package.xml` 中，我们应当确保 `build_depend` 和 `run_depend` 元素中存在相同名称的包，即：

```
<build_depend>turtlesim</build_depend>
```

```
<run_depend>turtlesim</run_depend>
```

完成这些修改后，多次使用的 `catkin_make` 命令应该就能完成程序的编译了。

8.4 服务器程序

下面通过编写服务器端的程序，来了解一下服务调用的其他方面。表 8.2 展示的例子提供名为 `toggle_forward` 服务，同时具备控制 `turtlesim` 机器人运动的功能，每次服务调用时将在前进和转向之间转换。

可以看到，服务器端的代码与订阅话题的代码非常相似。除了名称上的区别，我们必须创建一个 `ros::ServiceServer` 来代替 `ros::Subscriber`，唯一的区别在于服务器端可以通过一个响应对象

和一个表明成功与否的布尔变量给客户端回传数据。

编写服务的回调函数 如同订阅一样，节点提供的每一个服务必须关联一个回调函数，服务的回调函数原型如下：

```
bool function_name( package_name::service_type::Request &req),
package_name::service_type::Response &resp)
){
    ...
}
```

节点每次接收到一个服务请求，ROS 就执行一次回调函数。参数 **Request** 中包含了来自于客户端的数据。回调函数的工作是给 **Response** 对象的数据成员赋值。**Request** 和 **Response** 数据类型与上面在客户端使用的一致，因此需要相同的头文件和相同的包依赖关系来编译。回调函数返回 **true** 表明成功，返回 **false** 则表示失败。

在该例中，**Request** 和 **Response** 都使用了 **std_srvs/Empty**（空字符串）作为其数据类型，因此无需针对这两个对象做任何数据处理。回调函数的唯一工作是切换一个称为 **forward** 的全局布尔变量，用于管理在主函数中发布的速度消息。

```

1//This program toggles between rotation and translation
2// commands, based on calls to a service .
3#include <ros/ros.h>
4#include <std_srvs/Empty.h>
5#include <geometry_msgs/Twist.h>
6
7bool forward = true ;
8bool toggleForward (
9 std_srvs::Empty::Request &req ,
10 std_srvs::Empty::Response &resp
11 ) {
12 forward = !forward;
13 ROS_INFO_STREAM("Now _ sending _ " << (forward ?
14 "forward" : "rotate ") << " _ commands.");
15 return true ;
16 }
17
18 int main( int argc , char ** argv ) {
19 ros::init( argc , argv , "pubvel_toggle");
20 ros::NodeHandle nh ;
21
22 //Register our service with the master .
23 ros::ServiceServer server = nh.advertiseService (
24 "toggle_forward" , &toggleForward );
25
26 //Publish commands, using the latest value for forward,
27 // until the node shuts down.
28 ros::Publisher pub = nh.advertise <geometry_msgs::Twist>(
29 "turtle1/cmd_vel" , 1000 ) ;
30 ros::Rate rate(2);
31 while( ros::ok() ) {
32 geometry_msgs::Twist msg;
33 msg.linear.x = forward ? 1.0 : 0.0 ;
34 msg.angular.z = forward ? 0.0 : 1.0 ;
35 pub.publish (msg);
36 ros::spinOnce();
37 rate.sleep();
38 }
39 }

```

表 8.2 基于提供的服务，pubvel_toggle.cpp 程序修改它发布的速度指令。

创建服务器对象 为了将回调函数与服务相关联，从而给其他节点提供服务，我们必须使用如下方法发布这个服务：

```
ros::ServiceServer server =  
node_handle.advertiseService( service_name,pointer_to_callback_func  
tion);
```

代码中所有的语法都已经出现过，这里无需赘述。

`node_handle` 是前面我们多次使用的，也乐于使用的节点句柄。

`service_name` 是我们想要提供服务的字符串名称，建议为它使用局部名称，但是并不严格限制使用全局名称。

由于在私有名称的解析上存在一些理解上的歧义，`ros::NodeHandle::advertiseService` 拒绝接受私有名称（即那些以~开始的名称）。这种约束的解决方案是基于以下事实——一种目前我们还没用使用过的节点创建方法，即可以利用节点自身的缺省命名空间来创建 `ros::NodeHandle` 对象。举个例子，我们可以按照以下方式来创建 `ros::NodeHandle`：

```
ros::NodeHandle nhPrivate("~");
```

这种情况下，发送给这个节点句柄的任意局部名称的缺省命名空间同节点名称一致。特别指出的是，这意味着如果我们用这个句柄和相对名称来发布一个服务，效果将与使用私有名称相同。例如在一个名为 `/foo/bar` 的节点，像如下广播一个名为 `/foo/bar/baz` 的服务：

```
ros::ServiceServer server = nhPrivate.advertiseService("baz",  
    Callback);
```

如果句柄愿意接受私有名称的话，这段代码的效果与使用通常的 `NodeHandle` 广播名为 `~baz` 的服务效果相同。

最后的参数是指向回调函数的指针。在第 68 页有关于函数指针的简要介绍，包括针对一些常见语法错误的提示，对这里同样有效。

与 `ros::Subscriber` 对象一样，很少需要调用 `ros::ServiceServer` 对象的任意函数。反而我们要小心地跟踪这个对象的生命期，因为只有 `ros::ServiceServer` 被销毁前，其他节点才能使用服务。

提供 ROS 控制权 千万别忘了，只有在我们通过 `ros::spin()` 或者 `ros::spinOnce()` 请求时，ROS 才会执行回调函数。（这两种函数区别的详细介绍参照 3.4 末尾部分介绍订阅者程序的内容）

在上面的例子中，我们使用 `ros::spinOnce()` 而不是 `ros::spin()`，因为在没有服务调用时还有其他工作要做，即发布速度指令。

8.4.1 运行和改进服务器程序

为了测试 `pubvel_toggle` 样例程序，编译它并且启动 `turtlesim_node` 和 `pubvel_toggle` 两个节点。两者运行的同时，你可以通过从命令行调用 `toggle_forward` 服务来在平移和转动中反复切换运动控制指令。

```
rosservice call /toggle_forward
```

图 8.1 展示了例子运行一段时间后的结果。

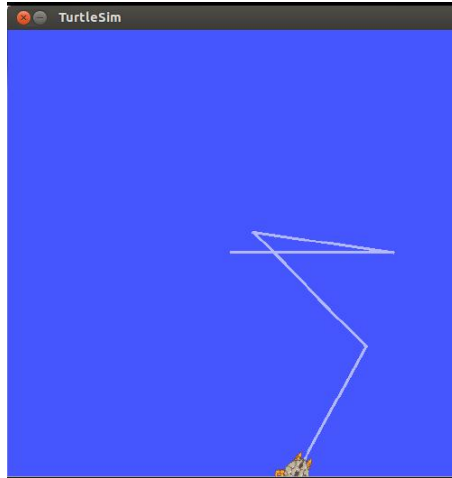


图 8.1 运行 `pubvel_toggle` 并间歇地人工调用 `/toggle_forward` 的结果。

如果你运行了这个例程，则可能已经发现，这个程序存在一个潜在的缺陷，在开始调用 `rosservice call` 和观察到海龟行动的实际变化之间有显著的延滞。这个时延中只有很小一部分来源于 `rosservice call`、`pubvel_toggle` 和 `turtlesim_node` 之间的通信时间。其余大部分的延迟来自与 `pubvel_toggle` 的工作方式。你能猜出来在哪里么？

答案是，我们通过调用 `ros::Rate` 对象的 `sleep` 方法使系统以相对较低的频率（只有 `2Hz`）循环，因此这个程序的大部分时间是用于休眠的。大部分的服务调用在 `sleep` 执行时到达，然后开始等待，直到调用 `ros::spinOnce()` 时才被执行，而这每 `0.5` 秒才发生一次。因此在每个服务调用被处理前会有最多大约 `0.5` 秒的延迟。

这个问题至少有两种解决方案：

可以使用两个分开的线程：一个发布消息，一个处理服务调用。尽管 `ROS` 没有明确指出程序要使用多个线程，但如果可以的话，这是非常便于合作的。

我们可以用`ros::spin`来代替`sleep/ros::spinOnce`循环，并且利用计数器回调函数（`timer callback`）⁴来发布消息。

诸如此类的问题看来并不重要，因为小的延迟对改变海龟的运动模式也许不会造成什么影响，但是对于实时性要求非常高的程序而言，这点区别是十分重要的。

关于此部分的ROS2实现可以参考教程：

> <https://index.ros.org/doc/ros2/Tutorials/Writing-A-Simple-Cpp-Service-And-Client/>

8.5 展 望

本章讨论的服务与消息有很强的相似性，但也有很明显的区别。下一章我们将了解一个名为 `rosbag` 的工具，它能够录制和回放消息来帮助用户快速准备和实施可重复的实验。

⁴ <http://wiki.ros.org/roscpp/Overview/Timers>

第9章 消息录制与回放[†]

三人行，必有我师焉，择其善者而从之，其不善者而改之。

——孔子

本章使用包（*bag*）文件录制和回放信息。

设计精良的 ROS 系统的一个重要特征便是系统中信息的消费者不应该关心信息的生产者。这种体系架构最明显的体现是 ROS 主要使用的消息发布 - 订阅通信模型。在这个模型下，不论什么时刻，只要有消息被发布，其订阅节点就应该正常工作，而不管是哪个或是哪些节点正在发布这些消息。

本章将介绍一个体现这种灵活性的具体例子，即一种被称为 *rosvbag* 的工具。通过 *rosvbag*，我们能够将发布在一个或者多个话题上的消息录制到一个包文件中，然后可以回放这些消息，重现相似的运行过程。将这两种能力结合，便形成了测试某些机器人软件的有效方式：我们可以偶尔运行机器人，运行过程中录制关注的话题，然后多次回放与这些话题相关的消息，同时使用处理这些数据的软件进行实验。

9.1 录制与回放包文件

术语包文件（*bag files*）是指用于存储带时间戳的 ROS 消息的特殊格式文件。*rosvbag* 命令行工具可以用来录制和回放包文件。

录制包文件 要创建一个包文件，使用 *rosvbag* 指令的如下形式：

```
rosvbag record -O filename.bag topic-names
```

Dashing:

```
ros2 bag record <topic_name>
```

多主题：

```
ros2 bag record -o subset /turtle1/cmd_vel /turtle1
```

[†]本章由肖军浩、郭昭宇翻译。

如果你不指定文件名，**rosvbag** 将基于当前的日期和时间自动生成一个。此外，还有一些命令行选项或许对使用 **rosvbag record** 进行消息录制有用。

除了像上面一样指定具体的话题外，你还可以使用 **rosvbag record -a** 记录当前发布的所有话题的消息。



对于本书中的种种小规模系统，完全可以其录制所有话题。然而，在许多真实的机器人系统中，这是一个令人沮丧的坏主意。例如，大部分搭载摄像头的机器人系统中存在多个节点发布与图像相关的话题，其中的图像经历了不同阶段的处理和不同级别的压缩。此时如果记录所有的话题，将迅速创建惊人的巨大包文件。因此，使用 **-a** 选项前要三思，或至少应该在录制过程中注意包文件的大小。

你可以使用 **rosvbag record -j** 启用包文件的压缩。与其他文件压缩一样，这里也需要折衷考虑：通常较小文件的代价是更长的读写时间。作者认为，对于包文件来说，压缩通常是个好主意。

当完成包文件录制时，使用 **Ctrl-C** 停止 **rosvbag**。回放包文件要回放一个包文件，可以使用如下命令：

```
rosvbag play filename.bag
```

Dashing:

```
ros2 bag play subset
```

存储在包文件中的消息将被回放，而且回放时会保持与其原始发布时同样的顺序和时间间隔。

检查文件包 **rosvbag info** 指令可以提供某个包文件中的丰富信息：

```
rosvbag info filename.bag
```

Dashing:

```
ros2 bag info <bag_file_name>
```

作为具体的例子，下面是将此命令用于作者在撰写下一节时所录制的包文件得到的输出：

```
path: square.bag
version: 2.0
duration: 1:08s (68s)
start: Jan 06 2014 00:05:34.66 (1388984734.66)
end: Jan 06 2014 00:06:42.99 (1388984802.99)
size: 770.8 KB
messages: 8518
compression: none [1/1 chunks]
types: eometry_msgs/Twist [9f195f881246dfa2798d1d3eebca84a]
       sim/Pose [863b248d5016ca62ea2e895ae5265cf9]
topics: /turtle1/cmd_vel 4249 msgs : geometry_msgs/Twist
       e1/pose 4269 msgs : turtlesim/Pose
```

值得一提的是，持续时间、消息计数以及话题列表三个字段似乎很有趣。

9.2 示例：正方形运动轨迹的包文件

让我们通过一个例子来感受下包文件是如何工作的。
绘制正方形轨迹（**Drawing squares**）首先，启动 `roscore` 和 `turtlesim_node` 两个节点。然后从 `turtlesim` 功能包中启动一个 `draw_square` 节点：

```
roslaunch turtlesim draw_square
```

该节点将重置仿真器（通过调用其重置服务），并发布速度指令，控制海龟的运动轨迹，使其不断重复一个近似的正方形形状。（你也可以使用我们写的任何节点来发布速度指令，不过作者准备的 `draw_square` 程序是一个很好的选择，因为不像 `pubvel` 节点，

这里海龟的运动足迹是结构化的。)

录制正方形轨迹的包文件 当海龟正在绘制正方形轨迹时，执行以下命令来记录速度指令和海龟的位置姿态信息：

```
roslaunch turtlebot3_teleop turtlebot3_teleop.launch
rosbag record -O square.bag /turtle1/cmd_vel /turtle1/pose
```

最开始的输出会告诉你 `roslaunch` 正在订阅 `/turtle1/cmd_vel` 和 `/turtle1/pose` 话题，发布在这两个话题上的消息正在被记录到文件名为 `square.bag` 的包文件中。在这一时刻，系统的计算图（使用 `rqt_graph` 得到）看起来如图 9.1 所示。新鲜和有趣的部分在于，`roslaunch` 创建了新的节点，名称为 `/record_...`，这个节点订阅了 `/turtle1/cmd_vel` 和 `/turtle1/pose` 话题。该图表明，`roslaunch` 通过订阅你需要的话题来记录消息，与其他节点一样，使用的是我们在第 3 章所学的发布订阅机制。

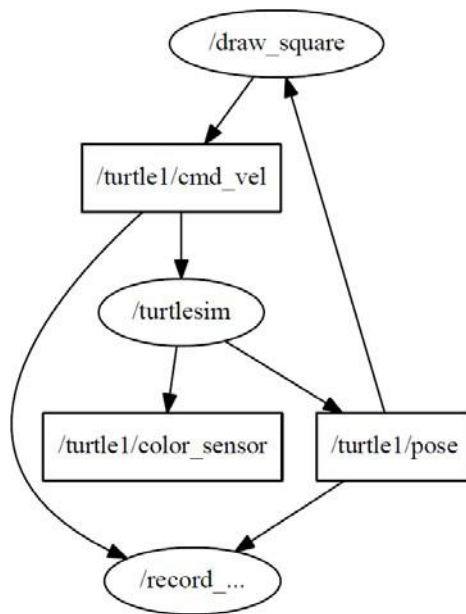


图 9.1 rosbag 记录运行时的节点和话题的计算图。



通过**rosvag**创建的节点通常使用之前在5.4节中讨论过的匿名名称。在本章中，为简单起见，我们用省略号（...）代替了数字后缀。需要注意的是，使用匿名意味着如果需要的话，我们可以同时运行多个**rosvag record**实例。

回放正方形轨迹的包文件(**Replaying the bag of squares**) 该系统运行了一段时间后——事实上一两分钟就足够，终止**rosvag**节点来停止录制，同时终止**draw_square**节点停止海龟的运动。接下来，让我们回放录制的包文件。在确认**rosvcore**和**turtlesim**节点仍运行后，使用下面的命令：

```
rosvag play square.bag
```

请注意，海龟将恢复运动。这是因为**rosvag**创建了名为**play...**的节点，且这个节点正在发布话题/**turtle1/cmd_vel**的消息，如图9.2所示。正如我们所期望的，它发布的信息和**draw_square**节点最初发布的一样。

图 9.3 所示的是这一操作序列在作者电脑上生成的图形。这个图可能有点令人惊讶，具体取决于你思考 **rosvag** 做了什么的细致程度。

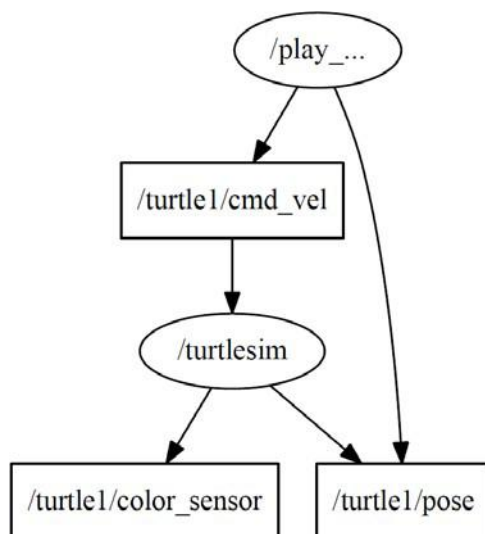


图 9.2 rosbag 运行时的节点和话题图。

在 rosbag play 运行期间，海龟画的正方形可能和 rosbag record 运行期间的正方形不在同一个地方。这是为什么呢？因为 rosbag 中录制的只是消息序列的副本，它没有复制初始条件。在 rosbag play 运行期间，海龟画的第二批正方形的起点正好是我们执行命令时海龟所在的位置。



图 9.3 [左]一个 turtlesim 龟对 draw_square 运动命令作出的反应。那些运动命令也是通过 rosbag 记录的。[右]通过重新画正方形(Replaying the bag

of squares)，我们可以将相同的信息序列发送给海龟。

原始的 `draw_square` 和 `rosvag play` 把海龟发送到了不同的地方，即使这个包文件里面有来自 `/turtle1/pose` 话题的数据。这又是为什么呢？很简单，因为在这个例子中，除 `rosvag record` 外没有其他节点订阅 `/turtle1/pose` 话题。系统中某个或某些节点（这里是 `rosvag play`）发布海龟所在的位置，和实际上海龟所在的位置是可以不同的。本例中，包文件里面的位姿数据被忽略了。

事实上，当 `turtlesim_node` 和 `rosvag play` 同时运行时，话题 `/turtle1/pose` 的消息是完全冲突的。表 9.1 是不到一秒钟的时间内连续快速发布到这个话题上的四条消息。请注意在 Y 坐标上的突变。幸运的是，没有节点订阅了这个话题，因为任何这样的节点都很难理解这些消息。

图 9.3 还说明服务调用（见第 8 章）并没有被录制到包文件



这里的建议是尽量避免（或者，至少谨慎对待）系统中 `rosvag` 和“真实”节点向同一个话题发布消息的情况。

里面。否则，包文件里面应该记录了某时刻 `draw_square` 节点调用了 `/reset` 服务，结果是海龟回到它的出发点，然后 `draw_square` 才开始发送控制命令。

9.3 启动文件里面的包文件

除了我们刚才看到的 `rosvag` 命令外，ROS 的 `rosvag` 功能包里面也提供了名为 `record` 和 `play` 的可执行文件。和 `rosvag record` 及 `rosvag play` 相比，这些程序有相同的功能，并且接受相同的命令行参数。

1	x:	5.93630695343
2	y:	4.66894054413
3	theta:	5.85922956467
4	linear_velocity:	0.0
5	angular_velocity:	0.40000000596
6	---	
7	x:	5.56227588654
8	y:	7.4833817482
9	theta:	4.17920017242
10	linear_velocity:	0.0
11	angular_velocity:	0.40000000596
12	---	
13	x:	5.93630695343
14	y:	4.66894054413
15	theta:	5.865629673
16	linear_velocity:	0.0
17	angular_velocity:	0.40000000596
18	---	
19	x:	5.56227588654
20	y:	7.4833817482
21	theta:	4.18560028076
22	linear_velocity:	0.0
23	angular_velocity:	0.40000000596
24	---	

表9.1 短时间内在/turtle1/pose 上发出的有关海龟位置的冲突的四个连续信息。注意在 y 坐标的巨大差异。这个冲突发生是因为两者turtlesimand 和 rosbag 都发布了这一话题。

这意味着，一方面，有可能使用`roslaunch`录制和回放包文件，虽然命令有些冗长，就像这样：

```
roslaunch rosbagrecord -O filename.bag topic-names
roslaunch rosbagplay filename.bag
```

更重要的是，通过这两个可执行文件可以很容易地将包文件作为启动文件的一部分，方法是包含适当的节点元素。例如，一个录制节点可能是这样的：

```
<node
  pkg="roslaunch" name="record" type="record"
  args="-O filename.bag topic-names"
/>
```

与此类似，一个回放节点可能如下所示：

```
<node
  pkg="roslaunch" name="play" type="play" args="filename.bag"
/>
```

除了需要给命令行传递必要的参数，这些节点不需要`roslaunch`的特殊处理。



危险符：在这个时刻，你可能会惊讶地看到这章就要结束了，但对如何在 **C++** 程序使用包文件没有做任何讨论。事实上，确实存在一个用于读取和写入包文件的 **API**。然而，只有一些非常特殊的应用程序才需要这个 **API**。对于简单的录制和回放操作，`roslaunch` 命令行接口已经足够了。

9.4 展望

我们就此结束了本书对 ROS 基本要素的介绍。下一章将对全书进行简短的总结，并且简单地提及一些在实际 ROS 系统中频繁出现的其他几个议题。

更多关于ROS2的bag教程参考：

> <https://index.ros.org/doc/ros2/Tutorials/Ros2bag/Recording-And-Playing-Back-Data/>

第10章 总结[†]

道生一，一生二，二生三，三生万物。

——老子

在这一章，我们将补充介绍一些内容。

在前面几章中，我们详细了解了 ROS 的基本工作机制，其中很多概念都是借助 `turtlesim` 仿真器的例子讲解的。显而易见，大部分读者最初决定学习 ROS 的动机应该不仅仅是期望控制一个仿真海龟的运动。既然如此，如果这本书对你有用，那么你应该已经做好准备开始使用 ROS 去解决机器人开发中的实际问题了。当然，这个过程中还很可能要参考在线文档。除此之外，作者建议你尽量借助现有的功能包，避免代码的重复实现，提高机器人软件开发效率。

10.1 下一步

这一节扼要介绍前面章节没有覆盖但在实际 ROS 系统中又经常碰到的几个问题，并给出它们在 ROS 官网的链接地址。

在网络环境中运行 ROS 你可能还记得，在第一章中，我们提到 ROS 的一大优势就是支持分布式机器人控制模式，即诸多程序运行在不同的计算机上，通过互相交互来完成指定任务。遗憾的是，在本书中，全部的程序均运行在同一个电脑中。

为了在某网络中的多台计算机上运行 ROS，需要在网络层和 ROS 层分别进行配置。其中，网络层的配置是确保计算机之间能

[†]本章由肖军浩、赵云云翻译。

够互相通信，而ROS层的配置是确保所有节点都能与节点管理器通信^{1 2 3}。值得庆幸的是，只要完成正确的配置，网络通信的管理就可以移交给ROS了。换句话说，从用户的角度看，在不同计算机上运行的节点之间的通信和同一台计算机上运行的节点间通信没有任何区别。

编写更规范的程序 为了让读者更容易理解，本书例程的程序在编写过程中力图做到清晰简洁，因而忽视了可扩展性和可维护性。事实上，有很多教程针对如何编写更规范的代码给出了详细的阐述。例如，一些开发人员建议使用`ros::Timer`的回调函数来替代`ros::Rate`对象⁴；一些开发人员则倾向于将节点的所有或者大部分数据封装到一个类中，然后使用回调机制调用类的方法⁵，以此减少全局变量和全局函数的数量。但上述方法的代价是增加了程序的存储空间和计算复杂度，这也是本书没有遵循这些建议的原因。

使用**rviz**使数据可视化 运行**turtlesim**的例程时，几乎所有的消息数据都很简单，包括二维空间中的位置、朝向角度和速度等。与之相反，真实的机器人系统往往十分复杂，机器人传感器获得的数据不仅复杂而且通常带有噪声，本书中提到的方法不能用于查看这些数据。为了填补这个空白，ROS提供了一个叫做**rviz**的非常强大的图形界面工具，它可以通过订阅用户选择的话题来显示机

¹ <http://wiki.ros.org/ROS/NetworkSetup>

² <http://wiki.ros.org/ROS/Tutorials/MultipleMachines>

³ <http://wiki.ros.org/ROS/EnvironmentVariables>

⁴ <http://wiki.ros.org/roscpp/Overview/Timers>

⁵ http://wiki.ros.org/roscpp_tutorials/Tutorials/UsingClassMethodsAsCallbacks

器人内部的各种信息，便于机器人的开发和调试⁶。

创建消息和服务类型 本书中的所有例程涉及到的消息和服务均采用ROS系统已有的基本数据类型。当然，我么也可以根据自身需要创建新的消息和服务类型，这并不复杂^{7 8}。

使用 **tf** 工具来管理多个坐标系 由于机器人运行在真实的物理环境中，通常的做法是用不同的坐标描述机器人不同部件的位置，包括机器人想要避开或者交互的目标。因此，如何正确描述这些坐标所在的坐标系成为了关键问题。在 ROS 中，很多消息类型均包含一个 **frame_id** 域来标识消息中数据所在的坐标系。

为了更好的地用这些不同的坐标系，我们需要知道它们的相对位置关系。具体来说，我们需要知道将某个坐标从一个坐标系到另一个坐标系的变换矩阵（**Transformation**）。ROS提供了一个标准功能包**tftf**（**transformation**的缩写），其功能就是帮助节点来完成坐标转换^{9 10 11 12}。**tf**具有很强的鲁棒性，既能够处理来自不同节点的数据，也能应对坐标系的实时变换。

使用**Gazebo**仿真ROS系统最大的优势之一就是能够实现软件的模块化设计，基于这个框架可以轻易地替换系统中的各种软件模块，从而节约系统的开发时间，也使得测试变得简单方便。第9章以

⁶ <http://wiki.ros.org/rviz>

⁷ <http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv>

⁸ <http://wiki.ros.org/ROS/Tutorials/DefiningCustomMessages>

⁹ <http://wiki.ros.org/tf/Tutorials/Introductiontotf>

¹⁰ [http://wiki.ros.org/tf/Tutorials/Writingatflistener\(C++\)](http://wiki.ros.org/tf/Tutorials/Writingatflistener(C++))

¹¹ <http://wiki.ros.org/tf/Overview/DataTypes>

¹² <http://ros.org/doc/hydro/api/tf/html/c++/>

示例的形式介绍了ROS的这种能力，基于roslaunch工具可以实现各个节点消息的录制与回放，且可以用消息的回放节点替换原系统中发布这些消息的节点，以便开发人员完成实验数据的分析。另外一个强大的工具是Gazebo，它是一个高保真的机器人仿真器¹³。在Gazebo中，我们可以建立机器人和相应场景的仿真模型，然后为仿真机器人定义与实体机器人相同的通信接口。因此，在Gazebo中用仿真机器人测试后的代码可以无缝移植到实体机器人，从而大大提高了机器人调试和开发效率。

10.2 展望

ROS 浅析到此就告一段落了，作者真心希望这本书只是你ROS 旅程的起点，并预祝你利用 ROS 打造更智能、更强大的下一代机器人。

ROS1: <http://wiki.ros.org/>

ROS2: <https://index.ros.org/doc/ros2/>
<http://docs.ros2.org/dashing/>

¹³ http://gazebosim.org/wiki/Tutorials/1.9/Overview_of_new_ROS_integration