



## 2 ROS 基础实验

### 2.1 引言

早期的工业机械臂，其控制系统多采用单片机，因此并未上升到采用操作系统的地步。随着机器人的发展，机器人需要完成的任务越来越复杂，需要采用操作系统来完成机械臂的运动控制及其他工作，保证实时性，实现多任务调度。工业机器人发展到可在线示教编程阶段以后，各大厂家的工业机械臂，一般随机器人同时提供控制器与示教编程器，操作人员可以在示教编程器上在线点动示教，或在示教编程器上直接编程，然后下载到控制器去控制机器人在线示教，这个阶段的控制器与示教编程器一般采用嵌入式操作系统。（控制器上面的嵌入式操作系统，仅负责完成机械臂各驱动关节的控制以及一些 I/O 控制，机器人如果需要通过实现更上层的智能控制，需要另外由上位机（采用 PC 或工业控制电脑）来完成。随着机器人的发展进入智能机器人阶段，出现了机器人专用操作系统。机器人专用操作系统除了实现机器人的控制之外，一般还提供机器人开发环境及中间件。

目前有较多类似的机器人系统开发软件框架，例如 Player/Stage、Orocos、OpenRAVE、CARMEN、YARP、Microsoft Robotics Studio 以及 ROS（Robot Operation System）等。

- Player/Stage 最初是由美国南加州大学机器人研究实验室，于 1999 年开发的一个为多机器人系统提供内部接口和仿真环境的项目。目前，作为一个开源项目，它已经被世界上许多机器人研究者使用、修改和扩展。
- Orocos（Open Robot Control Software）是由欧盟支持开发的一个通用的、免费的用于机器人控制的模块化架构，它由实时工具集、运动学与动力学算法集、贝叶斯过滤库及 Orocos 组件库组成，能够提供机器人实时控制应用。
- OpenRAVE 是由美国卡内基梅隆大学开发的一个开源的机器人软件架构，集成了可视化仿真、规划算法、环境脚本和控制算法，它的系统架构由插件层、脚本层和机器人数据库层组成，能够实现跨平台开发
- CARMEN（Carnegie Mellon Robot Navigation Toolkit）Carmen 实现了一个模块化的架构，它提供了基本的导航原语，例如故障排除、路径规划和绘图。除了提供一个二维仿真器之外，Carmen 还可以支持几个在 Linux 上运行的物理机器人平台。
- YARP（Yet Another Robot Platform）是一个使用 C++编写的开源软件包，用于连接机器人的传感器、处理器和制动器。
- Microsoft Robotics Developer Studio（MRDS、MSRDS）是 Windows 环境下



用来撰写机器人程式或机器人情境模拟的集成开发环境。Microsoft Robotics Studio 应用开发环境包括一个运行时程序,一个仿真器,一个可视化编程语言以及一套工具。运行时环境可以工作在目前的机器人技术中使用的各种 8、16 和 32 位处理器上。该软件重点是让用户编写简单的模块化命令程序,并如同服务那样动作。这种程序一般不在目标机器人的有限处理器和存储器上运行,而是通过机器人定义的许多通信协议中的一种与机器人进行交互。比如 iRobot 公司的 Roomba 真空吸尘器,它是最受欢迎的消费类机器人之一,至今销量已经超过 100 万台,采用的就是飞思卡尔半导体公司提供的 16 位简单处理器。

- 2008 年,Stanford 大学人工智能实验室与 Willow Garage 公司合作,基于 Linux 操作系统建立了 ROS 开源共享机器人操作系统平台。ROS 基于 RPC 的通信机制,采用 Node、Messages、Topic 和 Services 等基本服务,构建了一个分布式的机器人应用软件框架。ROS 最大的特点是实现了与现有软件平台的兼容,如 Player, OpenCV, OpenRAVE 等,可以直接调用这些软件平台上的资源。ROS 还是一个可供远程合作开发的网络平台,它提供了 Python、C++、Java、Lisp、Octave 等多种语言支持,因而吸引了全世界的机器人研究人员基于 ROS 平台进行软件开发和集成。经过多年的发展,ROS 平台已经集成了数百个机器人应用软件包,包括 PCL(点云库)、OMPL(开放的运动规划库)、OpenCV 等。为了方便基于 ROS 进行机器人软件开发,ROS 还提供了三维仿真、调试、在线运行监测等开发工具支持。

近年来国际上一些研究机构和企业纷纷推出了一些有特色的研究平台,例如 PR2、Robonaut-II、Baxter、Meka 等,都采用了 ROS 操作系统。PR2 是美国 Willow Garage 公司结合 ROS 同步推出的一个家庭服务机器人研究平台,机构部分由两个 4 自由度手臂、2 个 3 自由度手腕、2 个单自由度手抓和全向移动底盘组成,集成了立体视觉、前后激光雷达、Kinect 等多种传感器系统。为了配合 ROS 的推广,2010 年,Willow Garage 公司为 11 个世界一流的机器人研究机构提供了两年的 PR2 机器人使用权,包括斯坦福大学、伯克利大学加州分校、麻省理工学院、佐治亚理工学院、南加州大学、宾夕法尼亚大学、博世公司和东京大学等,要求这些研究机构将在 PR2 上开发的软件成果免费回馈给 ROS 社区。基于 PR2 机器人实现 ROS 平台共享开发的计划是机器人领域协同开发的成功案例,参加计划的研究机构在 PR2 机器人上测试了自己在机器人领域多年累积的软件成果,并回馈到 ROS 社区与其它机器人研究机构共享。这迅速吸引了全世界的机器人研究人员加入 ROS 社区,使 ROS 在短短几年多的时间积累了数百个机器人应用软件包,并得到机器人领域众多研究机构 and 企业的广泛支持。机器人航天员 Robonaut-II



是由美国 NASA 和通用公司合作推出的双臂协作机器人，主要用于替代人类航天员完成危险的太空作业，Robonaut 支持 ROS 的目的是无需关注系统控制平台，而将研究重点放在机器人性能提升和应用功能开发方面。Baxter 则是由著名机器人公司 iRobot 创始人，美国麻省理工学院教授 Rodney Brooks 在 2012 年创立的 Rethink 公司推出的，是一款低成本的双臂协作工业机器人，全部机构采用串联弹性驱动（SEA，Series Elastic Actuators）技术，实现了与人协作环境下的安全操作，由于采用了 ROS 平台，该机器人软件开发周期缩短了 2-3 年。Meka 是一个全柔性的与人协作双臂机器人，最初也是由 Rodney Brooks 在 MIT 指导的博士生 Edsinger 等人开发的，在设计初期，就采用了 ROS 平台，2014 年初，Meka 公司被谷歌公司收购，证明了该公司的价值。

这几年，随着 ROS 的普及和推广应用，各种基于 ROS 的应用软件开发成为主流。由于使用 ROS 开发可以使机器人应用开发者无需关注机器人操作系统和底层硬件平台的具体实现细节，而把研究重点放到应用功能开发方面，从而大大加快了机器人应用软件开发的效率和水平。

## 2.2 ROS 简介

ROS（机器人操作系统，Robot Operating System），是专为机器人软件开发所设计出来的一套操作系统架构。它是一个开源的元级操作系统（后操作系统），提供类似于操作系统的服务，包括硬件抽象描述、底层驱动程序管理、共用功能的执行、程序间消息传递、程序发行包管理，它也提供一些工具和库，用于获取、建立、编写和执行多机融合的程序。

ROS 的运行架构是一种使用 ROS 通信模块实现模块间 P2P 的松耦合的网络连接的处理架构，它执行若干种类型的通讯，包括基于服务的同步 RPC（远程过程调用）通讯、基于 Topic 的异步数据流通讯，还有参数服务器上的数据存储。

ROS 的首要设计目标是在机器人研发领域提高代码复用率。ROS 是一种分布式处理框架（又名 Nodes）。这使可执行文件能被单独设计，并且在运行时松散耦合。这些过程可以封装到数据包（Packages）和堆栈（Stacks）中，以便于共享和分发。ROS 还支持代码库的联合系统。使得协作亦能被分发。这种从文件系统级别到社区一级的设计让独立地决定发展和实施工作成为可能。上述所有功能都能由 ROS 的基础工具实现。

为了实现“共享与协作”这一首要目标，人们制订了 ROS 架构中的其他支援性目标：

“轻便”：ROS 是设计得尽可能方便简易。您不必替换主框架与系统，因为 ROS 编写的代码可以用于其他机器人软件框架中。毫无疑问的，ROS 更易于集成与其他机器人软件框架。事实上 ROS 已完成与 OpenRAVE、Orocos 和 Player 的整



合。

ROS-agnostic 库:【agnostic: 不可知论】建议的开发模型是使用 clear 的函数接口书写 ROS-agnostic 库。

语言独立性: ROS 框架很容易在任何编程语言中执行。我们已经能在 Python 和 C++ 中顺利运行,同时添加有 Lisp、Octave 和 Java 语言库。

测试简单: ROS 有一个内建的单元/组合集测试框架,称为“rostopic”。这使得集成调试和分解调试很容易。

扩展性: ROS 适合于大型实时系统与大型的系统开发项目。

## 2.3 ROS 组成与架构

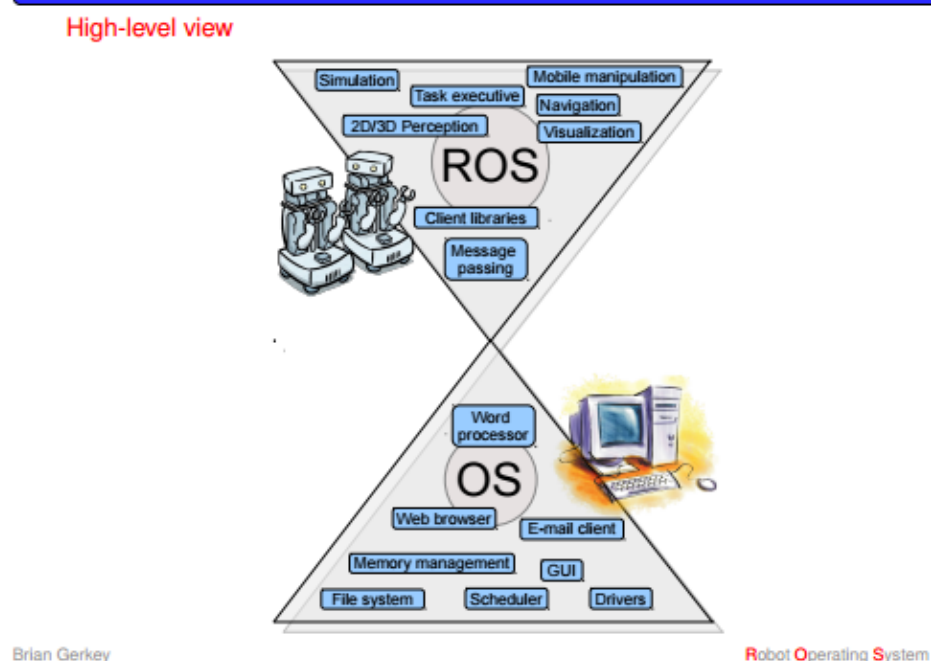


图 2.1 ROS 架构

ROS 是建立在计算机操作系统基础上的元级操作系统,如图 2.1 所示,它支持在多个计算机及多个机器人上获取、构建、编写和运行机器人相关代码。目前,ROS 的软件主要在 Ubuntu 和 Mac OS X 系统上测试,同时 ROS 社区仍持续支持 Fedora, Gentoo, Arch Linux 和其它 Linux 平台。

根据 ROS 系统代码的维护者和分布来划分,主要分成 main 和 universe 两大部分,组成 ROS 的完整生态系统(ecosystem),如图 2.2 所示。

(1) main: 核心部分,主要由 Willow Garage 公司和一些开发者设计、提供以及维护。它提供了一些分布式计算的基本工具(rxgraph, rostopic, roslaunch 等),以及整个 ROS 的核心部分中间件(middleware)的程序编写,包括 rosmaster、roscpp、rospy、roscpp、rospy、roscpp 等。

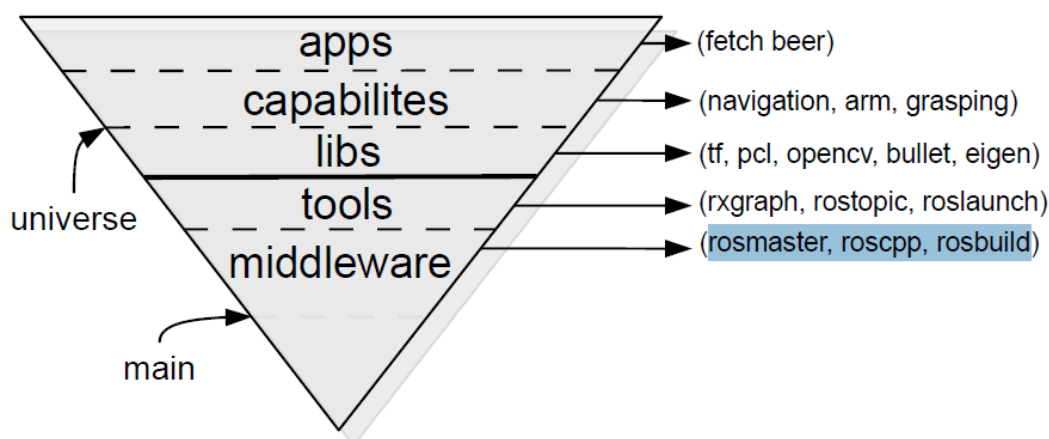


图 2.2 ROS 生态系统

(2) **universe**: 全球范围的代码，由不同国家的 ROS 社区组织开发和维护。最下层是库代码，如 OpenCV、PCL、moveit、industry、tf、bullet、eigen 等；库的上一层是从功能角度出发开发的代码，如导航、机械臂控制、抓取等，它们调用下层相关库；最上层为应用层代码，完成机器人特定具体功能。

一般来说，ROS 系统有三级概念：文件系统级 Filesystem Level、计算图级 Computation Graph Level、社区级。

**文件系统级概念**：我们在硬盘上面 ROS 源代码的组织方式。会通过功能包（package） manifest， package stack 等方式来进行组织。

**Packages**: ROS 的基本组织，可以包含任意格式文件。一个 Package 可以包含 ROS 执行时处理的文件（nodes），一个 ROS 的依赖库，一个数据集合，配置文件或一些有用的文件在一起。

**Manifests**: Manifests (manifest.xml) 提供关于 Package 元数据，包括它的许可信息和 Package 之间依赖关系，以及语言特性信息，例如编译标志（编译优化参数）。

**Stacks**: Stacks 是 Packages 的集合，它提供一个完整的功能，像 “navigation stack” Stack 与版本号关联，同时也是如何发行 ROS 软件方式的关键。

**Manifest Stack Manifests**: Stack manifests (stack.xml) 提供关于 Stack 元数据，包括它的许可信息和 Stack 之间依赖关系。

**Message(msg)types**: 信息描述位置在路径：  
my\_package/msg/MyMessageType.msg, 定义数据类型在 ROS 的 messages ROS 里面。

**Service (srv) types**: 服务描述,位置在路径: my\_package/srv/MyServiceType.srv, 定义这个请求和相应的数据结构 在 ROS services 里面。

在图 2.3 中，从该 stack 的文件视图上可以看到，该 stack 包含 bfl 与 actionlib





两个包。

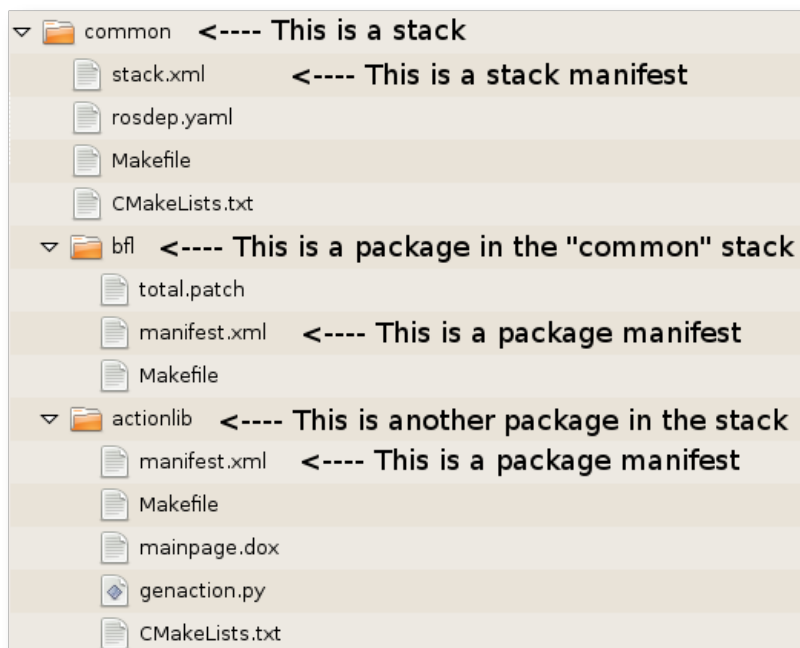


图 2.3 stack 文件视图

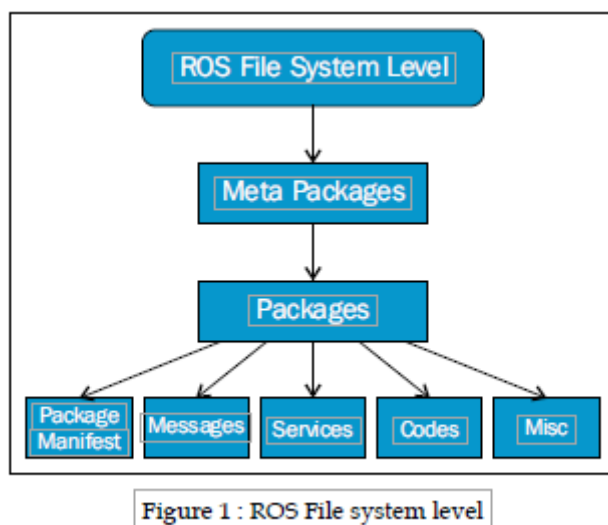


Figure 1 : ROS File system level

图 2.4 ROS 文件系统层级

一个 ROS Package 可以包含 ROS 运行时处理的多个文件（节点 nodes），ROS 系统的依赖库，数据集合，配置文件或一些有用的文件。而 Stacks 是 Packages 的集合，它提供一个完整的功能，例如“Hello stack”，Stack 与版本号关联，同时也是如何发行 ROS 软件方式的关键。

以下描述 ROS 计算图级的组成视图（Organizational Diagram）。

计算图级：程序运行时，所有进程及它们所进行的数据处理，将会通过一种



点对点的网络形式表现出来。将通过节点，节点管理器，topic， service 等进行表现。

Computation Graph Level 就是用 ROS 的 P2P (peer-to-peer 网络传输协议) 网络集中处理所有的数据。基本的 Computation Graph 的概念包括 Node， Master， Parameter Server， messages, services, topics， 和 bags， 以上所有的这些都以不同的方式给 Graph 传输数据。

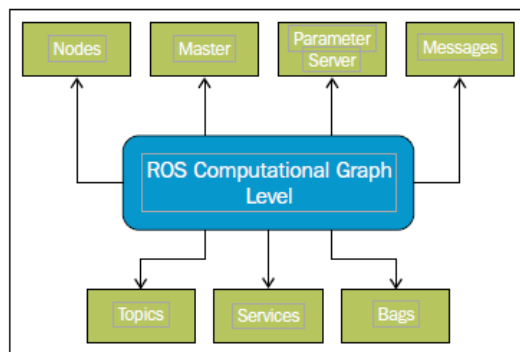


图 2.5 ROS 计算图级

**Nodes:** Nodes 是一系列运行中的程序。ROS 被设计成在一定颗粒度下的模块化系统。一个机器人控制系统通常包含许多 Nodes。比如一个 Node 控制激光雷达，一个 Node 控制车轮马达，一个 Node 处理定位，一个 Node 执行路径规划，另外一个提供图形化界面等等。一个 ROS 节点是由 Libraries ROS client library 写成的, 例如 roscpp 和 rospy.

**Master:** ROS Master 提供了登记列表和对其他计算 Graph 的查找。没有 Master，节点将无法找到其他节点，交换消息或调用服务。

**Server Parameter Server:** 参数服务器使数据按照钥匙的方式存储。目前，参数服务器是主持的组成部分。

**Messages:**节点之间通过 messages 来传递消息。一个 message 是一个简单的数据结构，包含一些归类定义的区。支持标准的原始数据类型（整数、浮点数、布尔数，等）和原始数组类型。message 可以包含任意的嵌套结构和数组（很类似于 C 语言的结构 structs）

**Topics:** Messages 以一种发布/订阅的方式传递。一个 node 可以在一个给定的 topic 中发布消息。Topic 是一个 name 被用于描述消息内容。一个 node 针对某个 topic 关注与订阅特定类型的数据。可能同时有多个 node 发布或者订阅同一个 topic 的消息；也可能有一个 topic 同时发布或订阅多个 topic。总体上，发布者和订阅者不了解彼此的存在。主要的概念在于将信息的发布者和需求者解耦、分离。逻辑上，topic 可以看作是一个严格规范化的消息 bus。每个 bus 有一个名字，每个 node 都可以连接到 bus 发送和接受符合标准类型的消息。



**Services:**发布/订阅模型是很灵活的通讯模式，但是多对多，单向传输对于分布式系统中经常需要的“请求/回应”式的交互来说并不合适。因此，“请求/回应”是通过 **services** 来实现的。这种通讯的定义是一种成对的消息：一个用于请求，一个用于回应。假设一个节点提供了一个服务提供下一个 **name** 和客户使用服务发送请求消息并等待答复。**ROS** 的客户库通常以一种远程调用的方式提供这样的交互。

**Bags:** Bags 是一种格式，用于存储和播放 **ROS** 消息。对于储存数据来说 Bags 是一种很重要的机制。例如传感器数据很难收集但却是开发与测试中必须的。

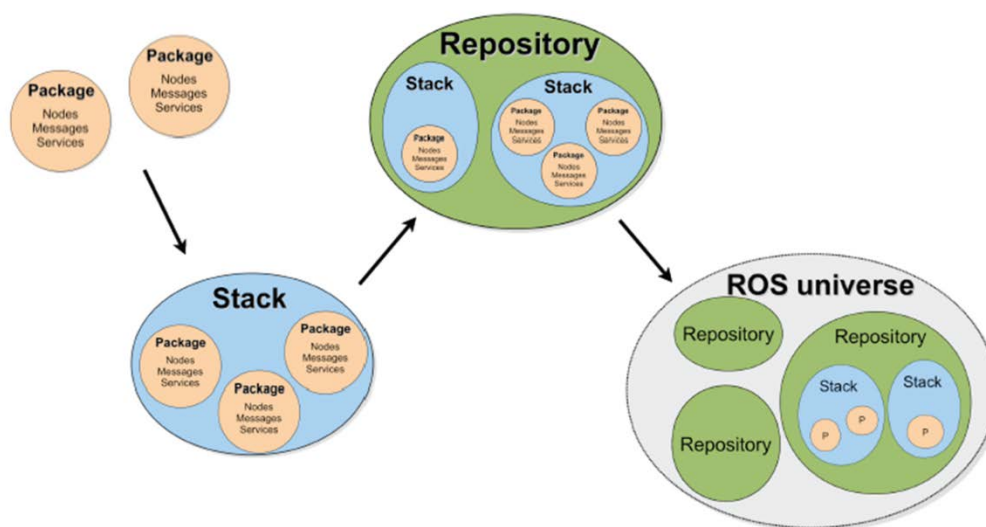


图 2.6 ROS 文件系统级别

### (1) 包(package)

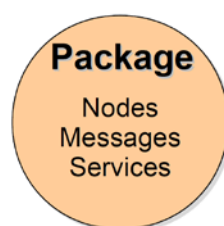


图 2.7 ROS 包

**ROS** 的软件以包的方式组织起来。包（**package**）由节点(**node**)、消息、主题、服务以及其他逻辑等构成。包的目的是提供一种易于使用的结构，以便于软件的重复使用。

节点(**node**)可以理解为执行运算任务的进程。**ROS** 采用代码模块化来实现系统的规模化增长，一个系统可以由很多节点组成的。当许多节点同时运行时，可以很方便的将端对端的通讯绘制成一个图表，在这个图表中，一个节点类似于一个进程，其中弧线连接代表端对端的连接关系。

节点之间通过传送消息进行通讯。每一个消息都是一个严格的数据结构。**ROS**





消息结构支持标准的数据类型（例如整型，浮点型，布尔型等），同时也支持原始数组类型。消息可以包含任意的嵌套结构和数组（类似于 C 语言的结构 `struct`）。

ROS 提供了主题(topic)模式和服务(service)模式两种通信方式。消息以一种发布/订阅(pub/sub)的方式进行传递，称为主题模式，它是一种异步的通信方式。节点可以在一个给定的主题中发布消息，也可以针对某个主题关注与订阅特定类型的数据。ROS 允许同时有多个节点发布或者订阅同一个主题的消息，而且发布者和订阅者并不需要知道彼此的存在。

虽然基于主题的发布/订阅模型是很灵活的通讯模式，但某些场合下，例如我们如果需要简化节点设计，采用同步传输比广播式的路径规划更为简单可靠。在这种情况下，ROS 提供了服务（service）模式进行通信。服务（service）模式用于处理 ROS 通讯中的同步通讯，采用 server/client 语义。每个 service type 拥有 请求(request) 与 应答(response)两部分,对于 service 中的服务端（server），ROS 不会检查重名（name conflict），只有最后注册的服务端（server）会生效，与客户端（client）建立连接。

ROS 中包含一个超级管理员 ROS Master, ROS Master 管理包(package)中所有节点，包括节点之间的相互通信。图 2.8 为 ROS 系统的基本结构。

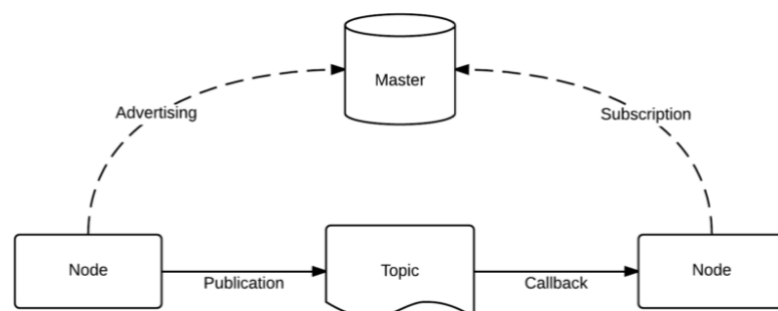


图 2.8 ROS 系统基本结构

ROS master 为 ROS 系统中的其余节点提供命名和注册服务。它跟踪发布者和订阅者的主题和服务。Master 的作用是使各个 ROS 节点能够相互定位。一旦这些节点彼此定位，就可以相互通信(peer-to-peer)。

ROS 系统各节点向 master 节点注册信息，master 发现有节点 subscribe/publish 相同的 topic 时，将 publisher 的信息通过 RPC 分发给各个 subscriber，subscriber 与 publisher 建立第一次连接，传输 topic 信息，然后再根据 publisher 返回的 topic 信息，建立第二次连接，publisher 开始传输具体的数据。

而注销（unregistration）的过程则是对称的，publisher 通过远程调用 `unregisterPublisher()`，然后 subscriber 通过 `unregisterSubscriber()` 注销。是否关闭 publisher 与 subscriber 间的数据流传输取决于节点本身。

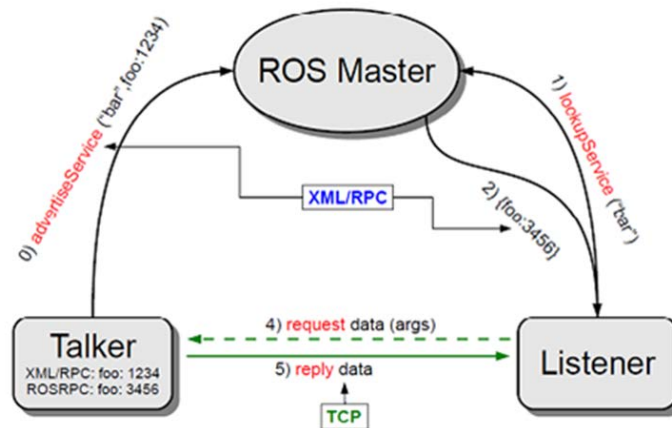


图 2.9 发布/订阅模型一

服务 **Service** 模式下的工作原理与主题 **topic** 模式有所不同，同一个 **service** 能被多个节点注册，但只有最后一个注册才能生效。一个节点需要调用某个 **service** 时，调用 **lookupService()**，通过 **master** 节点查找到相应 **service** 的 **URI**，发出一个请求 **request** 消息调用 **service** 的提供者，如果成功，服务 **service** 提供者将返回一个相应的响应 **response** 消息，失败则返回相应错误消息。

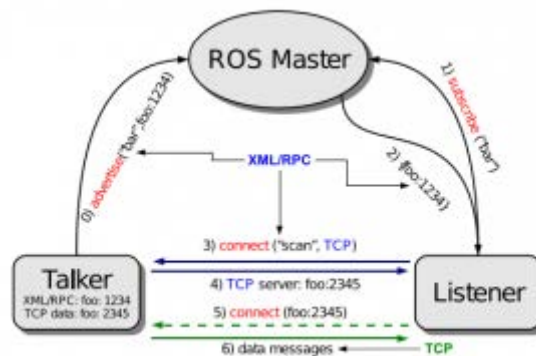


图 2.10 发布/订阅模型二

**ROS Master** 还提供了参数服务器(**Parameter Server**)。**Parameter** 可以看作为 **ROS** 系统运行时中定义的全局变量，而 **master** 节点中由参数服务器 **parameter server** 来维护这些变量。**ROS** 采用名字命名空间(namespace)，使得参数 **parameter** 拥有非常清晰的层次划分，避免重名，而且使得参数 **parameter** 访问可以单独访问也可以树状访问。

主机通常使用 **roscore** 命令运行，该命令将 **ROS** 主机与其他必要组件一起加载。

**ROS** 采用 **URI** 来定位节点 **node** 在分布式系统中的位置，格式为：**protocol://host:port**，**protocol** 一般为 **http** 或者 **rosrpc**，**host** 为主机名



(hostname) 或者 ip 地址, port 则为端口号。

ROS Master 提供了一个基于 XML/RPC (Remote Procedure Call Protocol, 远程过程调用) 的 API, 供 ROS 客户端库 (例如 roscpp 和 rospy) 调用来存储和检索信息。有关详细的 API 列表, 请参阅 Master API 页面。对于大多数 ROS 用户并不需要直接与此 API 进行交互。

RPC 不直接支持数据的流传输, ROS 通过基于 TCP 协议的 ROS 应用层数据协议 TCPROS 以及基于 UDP 协议的 ROS 应用层数据协议 UDPROS, 解析基于主题 topic 模式 或服务 service 模式下的二进制数据流。

## (2) 堆 (stack)

堆是包的集合, 它提供一个完整的功能, 比如 “navigation stack”。堆 Stack 与版本号关联, 同时也是如何发行 ROS 软件方式的关键。

ROS 采用分布式处理框架, 可执行文件能被单独设计, 并且在运行时松散耦合。这些过程可以封装到包 (Packages) 和堆 (Stacks) 中, 以便于共享和分发。

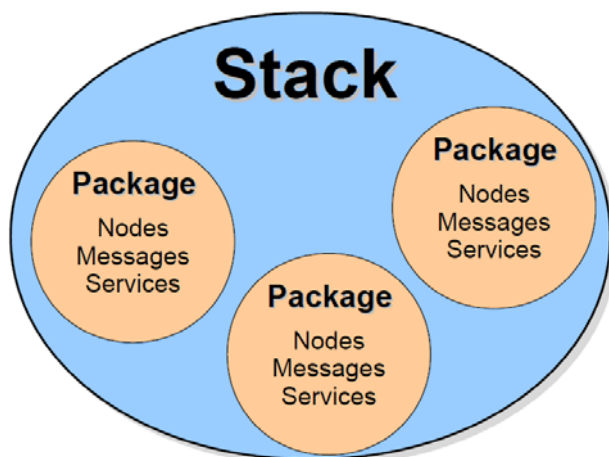


图 2.11 堆

社区级: ROS 在网络上进行代码发布的一种表现形式。因为 ROS 为了最大限度地提高社区参与, 使它能够快速的发展, 设计者们不再是由少部分人来存放, 更新和维护 ROS 代码, 而采用联合仓库的模式来处理。每个研究所和组织可以软件仓库 Repositories 为单位来发布他们的代码, 见图 2.11 所示。ROS 鼓励世界各地的开发者和用户, 提供和维护自己的 ROS 仓库代码, 而且他们对代码具有直接的所有权和控制权。

ROS 代码库的这种社区联合系统, 使得协作亦能被分发。这种从文件系统级别到社区级别的设计让独立地发展和实施工作成为可能。正是因为这种分布式的结构, 使得 ROS 迅速发展, 软件仓库中包的数量指数级增加。

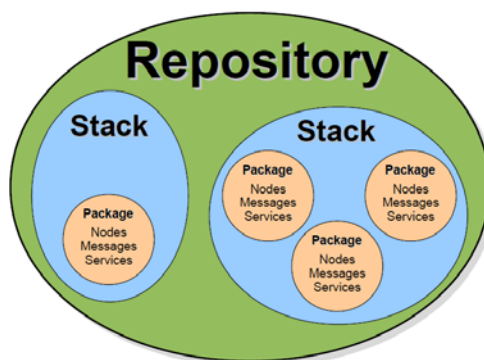


图 2.11 ROS 软件仓库

ROS 的社区级概念是 ROS 网络上进行代码发布的一种表现形式。结构如图 2.12 所示。

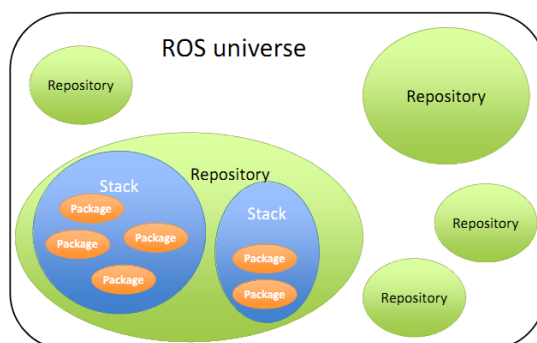


图 2.11 ROS 软件社区

在网站 <http://www.ros.org/news/repositories/> 页面包含了 ROS 最新的发布公告。软件仓库 Repositories 的数目，从 2007 年 11 月的 1 个到 2008 年 11 月的 4 个，到 2009 年 11 月的 16 个，2010 年 11 月的 52 个，到 2011 年 7 月的 79 个，到 2011 年 11 月 19 日，根据统计，institutions 的 repositories 达到 87 个，personal 的达到 14 个，single-serving 的达到 15 个。（这里的资料过时了，读者请自行查阅最新资料）

参见：<http://library.isr.ist.utl.pt/docs/roswiki/Repositories.html>

## 2.4 ROS 版本介绍与下载安装

ROS 经过多年的发展，已经支持多种操作系统，既可运行于一般 PC（AMD64、i386 等）计算机，也可运行于 ARM 等嵌入式系统。





ROS 主要支持 ubuntu 操作系统下的开发，同时也支持 Debian、Arch、Mac OS X、Fedora、Gentoo、OpenSuSe、Windows 等实验性的操作系统。

除了 ROS1.0 的各版本外，在 ROSCon 2014 上，ROS 正式发布了新一代 ROS 的设计架构（Next-generation ROS: Building on DDS），2015 年 8 月第一个 ROS2.0







的 alpha 版本落地，在经过一年多的开发，2016 年 12 月 19 日，ROS2.0 的 beta 版本正式发布。2017.12.09，机器人开源操作系统软件 ROS 2 终于推出首个正式版，新版本命名为“Ardent Apalone”，代号“ardent”（热情的美洲鳖）。ROS2.0 据说取消了 master 节点，使用了新的编译系统 Ament（ROS 为 Catkin），支持实时系统，并且支持 win10 版本了。目前为止我们还未在 ROS2.0 版本上深入研究，故本实验教程对 ROS2.0 后面不做介绍。

表 2.1 ROS 版本信息

发布版本	发布日期	发布海报	图标	维护截止日期（End Of Life）
ROS Melodic Morenia	2018.5.23			2023.5
ROS Lunar Loggerhead	2017.5.23			2019.5
ROS Kinetic Kame	2016.5.23			2021.4
Ros Jade Turtle	2015.5.23			2017.5
ROS Indigo Igloo	2014.7.22			2019.4(Trusty EOL)
ROS Hydro Medusa	2013.9.4			2015.5
ROS Groovy Galapagos	2012.12.31			2014.7
ROS Fuerte Turtle	2012.4.23			--
ROS Electric Emys	2011.8.30			--
ROS Diamondback	2011.3.2			--





ROS C Turtle	2010.8.2			--
ROS Box Turtle	2010.3.2			--

在已发布的 ROS1.0 版本中，目前为止只有 Indigo 之后的版本还在维护，各自对应的 ubuntu 版本如下：

ROS Indigo Igloo 对应 Ubuntu 14.04；

ROS Jade Turtle 对应 Ubuntu 15.04；

ROS Kinetic Kame 对应 Ubuntu 16.04；

ROS Lunar Loggerhead 对应 Ubuntu 17.04；

ROS Melodic Morenia 对应 Ubuntu 18.04。

本节以当前最稳定的版本 Kinetic 为例，介绍 ROS Kinetic 在 Ubuntu 16.04 下的安装步骤，选择已经预编译好的 ubuntu 平台 Debian 软件包，直接安装编译好的软件包。如需从源码编译安装，请另行参考 ROS WIKI 源代码下载编译方式。

**第一步：配置 Ubuntu 软件仓库。**

配置你的 Ubuntu 软件仓库(repositories) 以允许 "restricted"、"universe" 和 "multiverse"这三种安装模式。

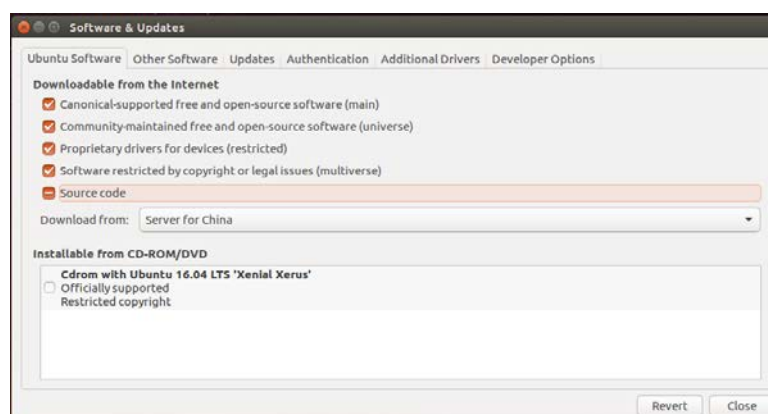


图 2.12 配置 Ubuntu 软件仓库

**第二步：添加 sources.list**

配置你的电脑使其能够安装来自 [packages.ros.org](http://packages.ros.org) 的软件。ROS Kinetic 仅支持 Vivid (15.04) 和 Xenial (16.04)版本。在终端执行命令行如下：

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

**第三步：添加密钥 keys，在终端执行命令行如下：**

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
```



0xB01FA116 或直接获取密钥安装, 在终端执行命令行如下:

```
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`。

**第四步: 开始安装。**

首先, 确保你的 `Debian` 软件包索引是最新的, 在终端执行命令行如下:

```
sudo apt-get update
```

执行如下命令完成桌面完整版 (包含 `ROS`、`rqt`、`rviz`、通用机器人函数库、2D/3D 仿真器、导航以及 2D/3D 感知功能) 安装:

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

或选择桌面版 (包含 `ROS`、`rqt`、`rviz` 以及通用机器人函数库) 安装:

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

在该步骤中, 如果有包安装失败, 需要根据提示, 重新更新并安装。

**第五步: 初始化 `rosdep`。**

`rosdep` 可以方便在需要编译某些源码的时候为其安装一些系统依赖, 同时也是某些 `ROS` 核心功能组件所必需用到的工具。在终端执行命令行如下:

```
sudo rosdep init
```

```
rosdep update
```

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`。

**第六步: 环境设置。**

添加 `ROS` 的环境变量以后, 当我们打开新的 `shell` 时, 在 `bash` 会话中会自动添加环境变量。在终端执行命令行如下:

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

```
source ~/.bashrc
```

**第七步: 安装 `roscpp`, 如不需要安装更多的 `ROS` 软件包, 该步骤可以省略。**

使用 `roscpp` 可以从源码安装 `ROS` 系统, `roscpp` 命令是一个使用非常频繁的命令, 使用这个命令可以轻松下载许多 `ROS` 软件包。在终端执行命令行如下:

```
sudo apt-get install python-roscpp
```

当以上步骤正常完成之后, `ROS` 即安装完毕。

## 2.5 ROS 命令行工具与 `Turtlesim`

`ROS` 提供三大类命令行工具 (`ROS Command-line tools`): 公共用户工具



(Common user tools)、绘图工具(Graphical tools)、非常用工具(Less-used tools)。

## 2.5.1 公共用户工具(Common user tools)

公共用户工具一般安装在\$ROS\_ROOT/bin 目录下, 在 ROS 安装过程中(上面安装过程第六步), 该目录会被添加到 ubuntu 的全局 PATH 变量中, 方便能够在用户目录下直接运行这些命令。

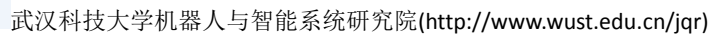
ROS 下的公共用户工具参见表 2.2。

表 2.2 公共用户工具

命令名	语法	功能描述	备注
rosvbag	rosvbag <supported commands> <topic-names>	对 ROS 包文件执行各种操作命令的工具	它可以记录一个包 bag, 从一个或多个包 bag 中重新发布消息, 总结 bag 的内容, 检查包 bag 的消息定义, 基于 Python 表达式过滤包 bag 的消息, 压缩、解压缩包 bag 以及重建包 bag 的索引。
rosvbash	source /opt/ros/<distro>/setup.bash	rosvbash 包中包含了一些有用的 bash 功能, 并增加标签完成了大量的基本 ROS 实用程序	rosvbash 不是一个命令, 而是一套命令和功能。它需要您提供 rosvbash 文件的内容
rosvcd	rosvcd locationname[/subdir]	允许通过名称直接 cd 到包, 堆栈或公共位置, 而	没有参数的 rosvcd 将会带你进入到



		不必知道包路径。	\$ ROS_ROOT
rosclean	rosclean <command>	清理由 ROS 创建的文件系统资源（例如日志文件）。	
roscore	roscore 或指定端口运行 roscore -p 1234	roscore 运行 ROS 核心堆栈，是基于 ROS 的系统的先决条件的节点和程序的集合。	如果你使用 roslaunch，它会自动开始 roscore 如果它检测到尚未运行。
rosdep	rosdep install PACKAGE_NAME	rosdep 是用于安装系统依赖关系的命令行工具。	
roscd	roscd packagename filename	允许您直接按包名称编辑包中的文件，而不必知道包路径。	如果文件名不是在包中唯一定义的，菜单将提示您选择要编辑的可能文件。
roscat pkg	roscat pkg pkgname	roscat pkg 创建新的 ROS 包所需的 Manifest, CMakeLists, Doxygen 和其他文件。	
roscat stack	roscat stack	roscat stack 创建常见的 Stack Manifest, CMakeLists 和新的 ROS 堆栈所需的其他文件	

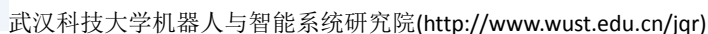


roslaunch	roslaunch package_name file.launch	roslaunch 从 XML 配置文件启动一组节点，并支持在远程计算机上启动。	
roscd	roscd [package-or-stack]	roscd 允许你运行任意的包的可执行文件，而无需首先 cd（或 roscd）。	
rosmake	rosmake [options] [PACKAGE]	rosmake 是帮助构建 ROS 包的工具。它有助于构建具有依赖关系的包。	
rosmmsg	rosmmsg show [msg name]	rosmmsg 是一个用于显示有关 ROS 消息类型信息的命令行工具，显示消息数据结构定义。	
rostopic	rostopic <supported commands> <topic name>	rostopic 是一个用于显示有关 ROS 话题的调试信息的命令行工具，包括发布，订阅和连接。	
rospack	rospack <supported commands> <package-name>	rospack 是一个	在 ROS
rosls	rosls [package-or-stack]	rosls 找到存储 ROS 包的存储库，例如，rosls svn tf。它可以轻松快速检出包的来源	在 ROS 1.4 中删除：现在是 rosinstall 工具的一部分
rosversion	rosversion [package-name]	rosversion 显示 ROS 包的版本信息	





	<code>commands&gt; [package_name]</code>	用于检索文件系统上可用的 ROS 包的信息的命令行工具。	Fuerte 之前, rospack 被包括在 ros 堆栈中。从 Fuerte 开始, 这是一个独立的工具。
rosparam	<code>rosparam &lt;supported commands&gt; [&lt;namespace&gt;/&lt;parameter-name&gt;]</code>	rosparam 可以使用 YAML 编码的文本从命令行获取和设置参数服务器值。	rosparam 使用 parameter 类型和 YAML 类型之间是 1 对 1 的关系
rossrv	<code>rossrv &lt;command&gt;</code>	rossrv 命令行工具显示有关 ROS 服务的信息。显示 Service srv 数据结构定义。	与 rosmmsg 具有完全相同的用途
rosservice	<code>rosservice &lt;supported commands&gt; &lt;service-name&gt;</code>	rosservice 显示有关服务的运行时信息, 还可以打印发送到主题的消息。	
rostack	<code>rostack &lt;supported commands&gt; [stack_name]</code>	rostack 是一个用于检索文件系统上可用的 ROS 堆栈信息的命令行工具。	
rostopic	<code>rostopic &lt;supported commands&gt; &lt;topic-name&gt;</code>	rostopic 显示有关主题的运行时信息, 还可以打印发送到主题的消息。	
rosversion	<code>rosversion</code>	显示 ROS 堆栈的版本。	



这里简单介绍 `roscore`、`roslaunch`、`roscpp`、`rostopic` 等常用命令。

\$ roscore

## --一个主控节点 ROS Master

## --一个参数服务器 ROS Parameter Server

## --一个 roscout 日志文件节点

运行后界面如图 2.13 所示。

```

roscore http://mhs:11311/
mhs@mhs:~$ roscore
... logging to /home/viki/.ros/log/c89def4a-13d1-11e7-9956-000c292fcb40/roslaunch
h-mhs-3074.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://mhs:37596/
ros_comm version 1.11.8

SUMMARY
=====

PARAMETERS
* /rostdistro: indigo
* /rosversion: 1.11.8

NODES

auto-starting new master
process[master]: started with pid [3086]
ROS_MASTER_URI=http://mhs:11311/

setting /run_id to c89def4a-13d1-11e7-9956-000c292fcb40
process[rosout-1]: started with pid [3099]
started core service [/rosout]

```

图 2.13 roscore

```
$ rosrun <package> <executable>
```

举例，如下命令将运行 turtlesim 包中的 turtlesim node 节点。

```
$ rosrun turtlesim turtlesim node
```

turtlesim node 节点运行后界面如图 2.14 所示。



图 2.14 turtlesim\_node

下面我们分别在三个终端窗口中运行如下三个命令：

`$roscore`

`$roslaunch turtlesim turtlesim_node`

`$roslaunch turtlesim turtle_teleop_key`

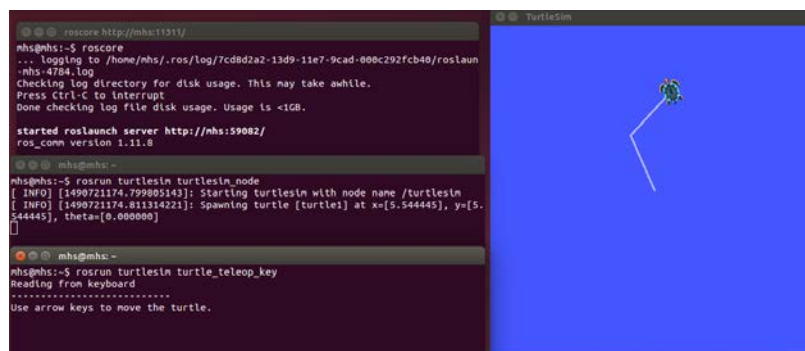


图 2.15 键盘控制

这三个命令将分别开启一个 ROS 主控节点，一个 turtlesim\_node 节点，在 TurtleSim 窗口绘制出一个乌龟，一个 turtle\_teleop\_key 节点，可以使用键盘的方向键遥控乌龟在 TurtleSim 窗口上画线。运行效果如图 2.15 所示。

使用 `rostopic` 命令可以显示关于 ROS 节点的调试信息，包括发布，订阅和连接等信息。

运行 `rostopic` 命令之前最少应该先运行 `roscore` 命令。`rostopic` 命令表见表 2.3 所示。

表 2.3 rostopic 命令表

命令	功能
<code>\$rostopic list</code>	列出活动节点
<code>\$rostopic ping [options] &lt;node&gt;</code>	测试连接节点
<code>\$rostopic info node1 [node2...]</code>	打印节点信息



\$rosclose node kill	关闭一个正在运行的节点
\$rosclose node machine machine-name	列出在特定机器上运行的节点

```
mhs@mhs: ~  
mhs@mhs:~$ rosclose node info turtlesim  
-----  
Node [/turtlesim]  
Publications:  
* /turtle1/color_sensor [turtlesim/Color]  
* /rosout [rosgraph_msgs/Log]  
* /turtle1/pose [turtlesim/Pose]  
  
Subscriptions:  
* /turtle1/cmd_vel [unknown type]  
  
Services:  
* /turtle1/teleport_absolute  
* /turtlesim/get_loggers  
* /turtlesim/set_logger_level  
* /reset  
* /spawn  
* /clear  
* /turtle1/set_pen  
* /turtle1/teleport_relative  
* /kill  
  
contacting node http://mhs:51949/ ...  
Pid: 4052  
Connections:  
* topic: /rosout  
* to: /rosout  
* direction: outbound  
* transport: TCPROS  
  
mhs@mhs:~$
```

图 2.16 节点信息

图 2.16 所示为使用\$rosclose node info turtlesim 命令显示出的 turtlesim 节点的所有信息。

rostopic 命令可以提供有关主题的信息，并允许在主题上发布消息。

表 2.4 rostopic 命令表

命令	功能
\$ rostopic list	列出活动主题
\$ rostopic echo /topic	在当前屏幕上打印主题信息
\$ rostopic info /topic	打印节点信息
\$ rostopic pub /topic type args	发布数据给一个主题

我们可以使用 rostopic pub 命令向某个主题发布数据，例如：

在运行 turtlesim 节点时，为了让乌龟以 0.2m/s 的速度向前移动，我们可以发布 geometry\_msgs/Twist 类型消息给主题/turtle1/cmd\_vel，命令格式如下：

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0},  
angular: {x: 0, y: 0, z: 0}}'
```

一般来说，ROS 中的运动控制通常采用 Twist 消息类型，执行下面的命令，可以查看 Twist 消息里有哪些组件：

```
$ rosmmsg show geometry_msgs/Twist
```



接下来我们可以看到输出的 `geometry_msgs/Twist` 消息结构如下：

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

一般来说，这里的线速度通常指定为 `m/s`；角速度指定为 `rad/s`。

本例的乌龟机器人是一个仿真的移动机器人，在上例中，我们指定 `Twist` 消息中的 `x` 轴线速度为 `0.2m/s`。乌龟机器人将沿水平线直线运动一段距离。一些诸如 `cmd_vel` 类型的消息具有预定义的超时，超过这个预定义的时间，消息将停止发布。如果我们需要持续发送消息，我们可以采用 `-r` 参数，并指定消息循环发送的频率。例如，我们可以执行如下命令，间隔 `100` 毫秒循环发送消息，让乌龟一直沿着一个圆形的轨迹运动。

```
$ rostopic pub /turtle1/cmd_vel -r 10 geometry_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0.5}}'
```

运行效果如图 2.17 所示。

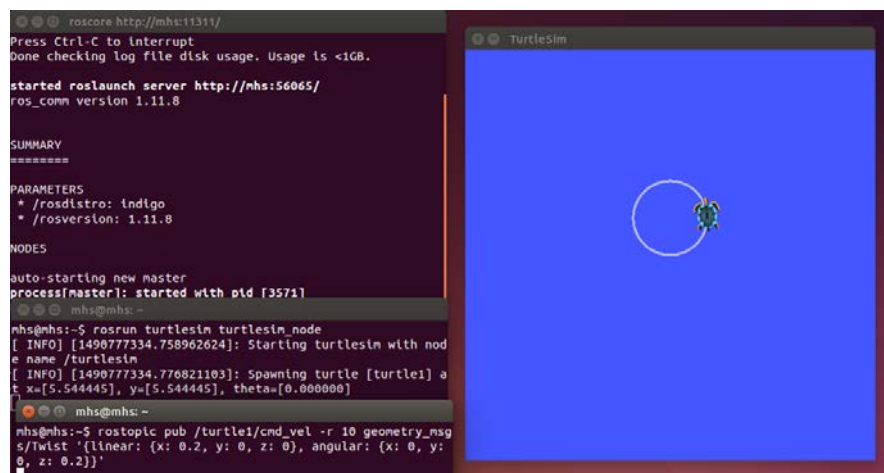


图 2.17 圆周运动

## 2.5.2 绘图工具(Graphical tools)

ROS 图形工具在使用之前通常需要额外的依赖关系，例如 `graphviz` 和 `Python GTK`。用户可以使用 `bash <(rosdep satisfy PACKAGE_NAME)` 快速安装这些工具。ROS 图形工具见表 2.5。





表 2.5 ROS 图形工具

命令名	语法	功能描述	备注
rqt_bag	rqt_bag	rqt_bag 是一个用于查看 ROS 包文件中的数据的数据的图形工具。	自 ROS groovy 以来 rqt_bag 取代了 rxbag。
rqt_deps	rqt_deps	生成一个 ROS 依赖关系的 PDF。	
rqt_graph	rosgraph	rqt_graph 显示 ROS 节点和主题的交互图。	
rqt_plot	rqt_plot	rqt_plot 绘制一段时间内 ROS 主题的数值数据。	

rqt\_graph 工具是 rqt 软件包的一部分，通过它可以创建系统中正在运行的节点、主题之间的一个动态关系图。如未安装，可通过如下命令安装。

```
$ sudo apt-get install ros-<distro>-rqt
```

```
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

注意在<distro>处用当前使用的 ROS 版本名替代。

让我们来看看，如果运行如下命令，会发生什么情况：

```
$roscore
```

```
$roslaunch turtlesim turtlesim_node
```

```
$roslaunch turtlesim turtle_teleop_key
```

当我们运行以上三个命令后，我们即可使用键盘上的方向键来控制小海龟的移动。如果我们想弄清楚 turtlesim 和 teleop\_turtle 节点之间到底是如何联系的，我们可以通过节点视图来查看。运行命令：

```
$roslaunch rqt_graph rqt_graph
```

我们即可观察到节点 teleop\_key 和节点 turtlesim 之间基于 turtle1/cmd\_vel 主题进行通信。

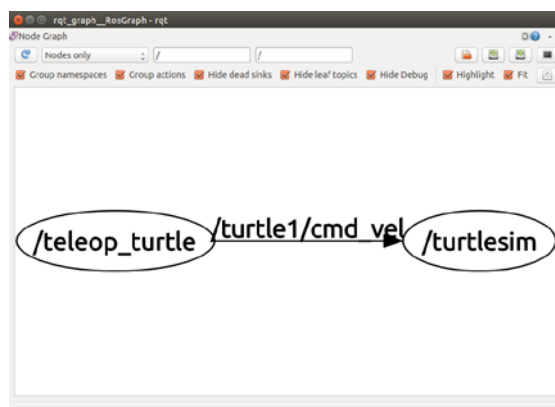


图 2.18 节点视图



接下来我们再运行一个命令：

```
$ rostopic pub /turtle1/cmd_vel -r 10 geometry_msgs/Twist '{linear: {x: 2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 1}}'
```

点击节点视图的刷新按钮，我们可以发现节点视图里面多了一个椭圆图形，一直在向 turtlesim 节点发布 /turtle1/cmd\_vel 类型信息，发送频率为 10HZ，此时的节点视图如图 2.19 所示。

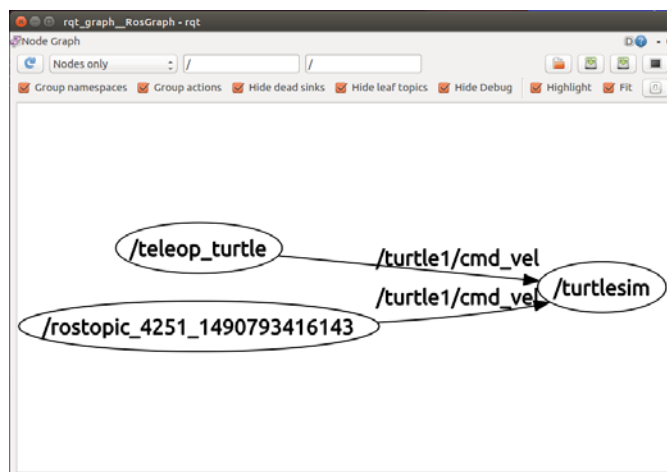


图 2.19 节点视图

使用 rqt\_plot 工具可以动态绘制发布到某一个主题上的数据图形，运行命令：

```
$ roslaunch rqt_plot rqt_plot
```

在出现的绘图窗口 Topic 输入框中输入想要绘制的主题名称，比如 /turtle1/pose 主题，点击加入按钮，我们即可观察到小海龟实时运动的角速度、线速度、坐标、方向的动态波形，如图 2.20 所示，并可暂停将数据图形存盘成图形文件以便分析。

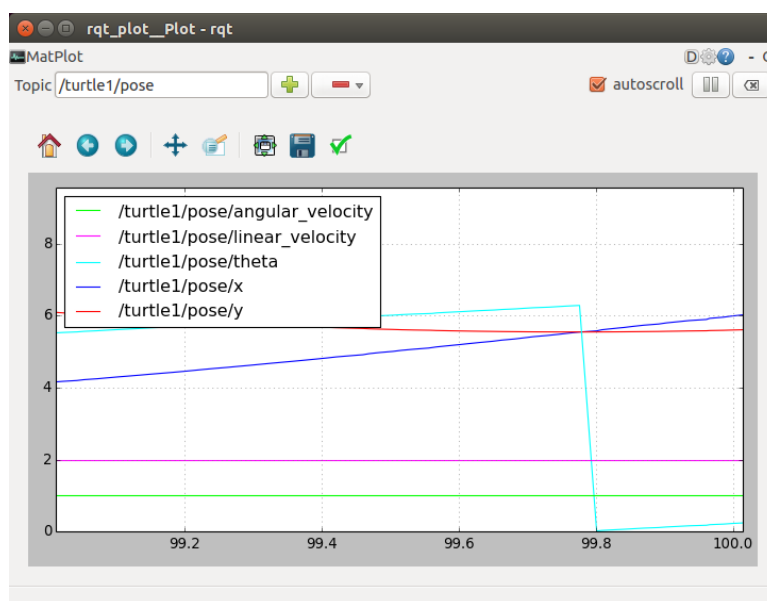


图 2.20 动态波形



## 2.5.3 非常用工具(Less-used tools)

ROS 非常用工具通常被用作内部工具,但最终用户较少需要使用这些工具。

## 2.6 如何编写 ROS 节点

创建 ROS 工作空间 Workspace 有两种方法,一种是 catkin、另一种是 rosbuilt。Catkin 被实现为自定义 CMake 宏以及一些 Python 代码,以一致和传统的方式支持大型相关软件包的开发。从 ROS 的 Groovy 以及以后的版本构建工程采用 catkin 工具,而对于旧版本的 ROS,只能使用 rosbuilt。本教材所有实验均基于 ROS Indigo 版本,采用 catkin 来创建工作空间 (Workspace)。在构建工作空间之前,确保已按前面章节的说明配置运行 ROS 命令所需要的环境设置。

### 2.6.1 开始 ROS “hello world!” 节点编写

首先创建一个工作空间—文件夹:

```
$mkdir -p ~/mhs/helloworld/src
```

然后进入到 src 路径下:

```
$cd ~/mhs/helloworld/src
```

初始化工作空间:

```
$catkin_init_workspace
```

最初,工作空间仅生成了顶级的 CMakeLists.txt 文件。虽然如此,但我们还是可以尝试去编译它:

```
$cd ~/mhs/helloworld
```

```
$catkin_make
```

一个 ROS 包 package 就是一个 catkin 工作空间内的一个目录,里面有一个 package.xml 文件,包是编译和发行的最原始单元,一个包可以包含一个或多个节点的源文件以及配置文件。

工作空间目录中的 src 子目录将用于存放功能包的源代码。在该目录下,运行如下格式命令创建包:

```
$catkin_create_pkg package-name
```

其中 package-name 为我们需要建立的包的名字,比如我们给自己建立的包取名 myPkg:

```
$catkin_create_pkg myPkg
```

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

□ 第一个配置文件是 package.xml,称为包清单文件。



□ 第二个文件是 **CMakeLists.txt**，它是一个 **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**。

接下来我们需要在 **myPkg** 目录下开始编写我们的第一个 ROS 的 C++ 程序，把它保存为 **hello.cpp**，内容如下：

```
/*This is a ROS version of the standard "hello, world"
program.*/
```

```
// This header defines the standard ROS classes.
```

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

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

```
    //Initialize the ROS system .
```

```
    ros::init (argc, argv, "hello_ros");
```

```
    //Establish this program as a ROS node.
```

```
    ros::NodeHandle nh;
```

```
    //Send some output as a log message.
```

```
    ROS_INFO_STREAM("Hello,ROS World!");
```

```
}
```

这个程序 **main** 函数只包含三行代码，**ros::init** 函数初始化 ROS 客户端库，需要在程序的起始处调用一次该函数，该函数最后的一个字符串参数代表所包含节点的默认名。**ros::NodeHandle**（节点句柄）对象是程序用于和 ROS 系统交互的主要机制，创建此对象会将程序注册为 ROS 节点。最简单的方法就是在整个程序中只创建一个 **NodeHandle** 对象。**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。

编写好 C++ 源代码后，我们需要设置 **catkin** 编译规则。ROS 的 **catkin** 编译设置一共有四个步骤：

1) 首先声明依赖库，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 **myPkg** 下的 **CMakeLists.txt** 文件。该文件的默认版本含有如下行：



```
find_package(catkin 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)
```

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)
```

我们同样需要在 myPkg 包的清单文件 package.xml 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>
```

在我们的例程中，hello 程序在编译和运行时都需要 roscpp 库，因此 package.xml 清单文件需要包括：

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

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

需要注意的是，如果在清单文件中声明的依赖库并没有在编译过程中用到，在编译时可能不会看到任何错误消息，如果发布这个包给其他人，他们可能会在没有安装所需依赖库的情况下，编译发布的包时出现错误。

2) 接下来声明可执行文件，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})
```

3) 编译工作空间。一旦 CMakeLists.txt 文件设置好，就可以编译创建的工作空间了，使用如下命令来编译所有包中的所有可执行文件：

```
catkin_make
```





因为项目被设计成编译工作空间中所有包，所以这个命令必须从创建的工作空间目录下运行。它将会完成一些配置步骤（尤其是第一次运行此命令时），并且在工作空间中创建 **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。

如果有编译错误，将会在执行此步骤时给出错误提示，更正错误之后，重新运行 **catkin\_make** 来完成编译工作。

4) 最后步骤是设置环境变量。执行名为 **setup.bash**，它是 **catkin\_make** 在工作空间的 **devel** 子目录下生成的脚本文件。命令如下：

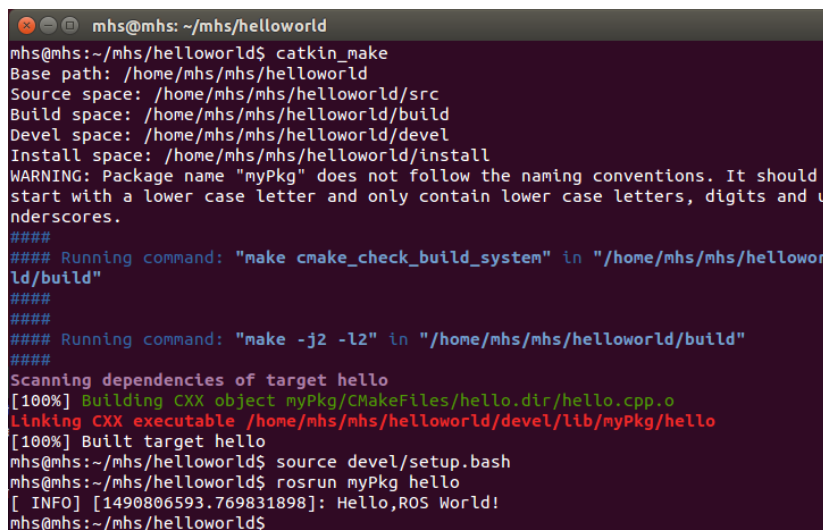
```
source devel/setup.bash
```

这个自动生成的脚本文件设置了若干环境变量，从而使 ROS 能够找到所创建的功能包和新生成的可执行文件。它类似于前面介绍 ROS 安装步骤里的全局 **setup.bash**，但这个 **setup.bash** 是专门为当前工作空间量身定做的。除非目录结构发生变化，即使修改了代码并且用 **catkin\_make** 执行了重编译，也只需要在每个终端执行此命令一次即可完成环境变量设置。

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

```
$roslaunch myPkg hello
```

运行效果如图 2.21 所示。



```
mhs@mhs: ~/mhs/helloworld
mhs@mhs:~/mhs/helloworld$ catkin_make
Base path: /home/mhs/mhs/helloworld
Source space: /home/mhs/mhs/helloworld/src
Build space: /home/mhs/mhs/helloworld/build
Devel space: /home/mhs/mhs/helloworld/devel
Install space: /home/mhs/mhs/helloworld/install
WARNING: Package name "myPkg" does not follow the naming conventions. It should
start with a lower case letter and only contain lower case letters, digits and u
nderscores.
####
#### Running command: "make cmake_check_build_system" in "/home/mhs/mhs/hellowor
ld/build"
####
####
#### Running command: "make -j2 -l2" in "/home/mhs/mhs/helloworld/build"
####
Scanning dependencies of target hello
[100%] Building CXX object myPkg/CMakeFiles/hello.dir/hello.cpp.o
Linking CXX executable /home/mhs/mhs/helloworld/devel/lib/myPkg/hello
[100%] Built target hello
mhs@mhs:~/mhs/helloworld$ source devel/setup.bash
mhs@mhs:~/mhs/helloworld$ roslaunch myPkg hello
[ INFO] [1490806593.769831898]: Hello,ROS World!
mhs@mhs:~/mhs/helloworld$
```

图 2.21 hello world 运行效果

## 2.6.2 ROS 消息和 ROS 服务

消息(msg): msg 文件就是一个描述 ROS 中所使用消息类型的简单文本。它们会被用来生成不同语言的源代码。

msg 文件存放在包工作空间的 **msg** 目录下。

消息只是一个数据结构，包括类型字段，支持标准原始数据类型（包括嵌套



数组), 可以使用的数据类型如下:

int8, 16, 32, 64

float32, 64

string

time

duration

array[]

更多信息请参见: <http://wiki.ros.org/msg>。

节点之间可以通过基于主题的模式, 采用发布/订阅机制来传递消息, 相互通信, 这时候可以使用消息文件。

在 ROS 中有一个特殊的数据类型: Header, 它含有时间戳和坐标系信息。在 msg 文件的第一行经常可以看到 Header header 的声明。

下面是一个 msg 文件的样例, 它使用了 Header, string, 和其他另外两个消息类型。

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

服务(srv): 一个 srv 文件描述一项 ROS 服务。

srv 文件存放在包工作空间的 srv 目录下。节点之间可以通过基于服务的模式, 采用同步的机制来传递消息, 相互通信, 这时候可以使用 srv 息文件。

srv 文件分为请求和响应两部分, 由'---'分隔。下面是 srv 的一个样例:

```
int64 A
int64 B
---
int64 Sum
```

其中 A 和 B 是请求, 而 Sum 是响应。

## 2.6.3 编写主题 topic 节点

通过 “hello world”例程我们学习了 ROS 节点的编写方法, 接下来, 我们来尝试编写一个新的包, 其中一个节点发布 “talk”主题消息, 另一个节点订阅 “talk”主题消息, 并将其打印在出来。

首先使用如下命令创建新的工作空间:

```
$cd ~/mhs/helloworld/src
```



\$catkin\_create\_pkg talk\_pkg std\_msgs rospy roscpp

```
mhs@mhs:~/mhs/helloworld/src$ cd ~/mhs/helloworld/src
mhs@mhs:~/mhs/helloworld/src$ catkin_create_pkg talk_pkg std_msgs rospy roscpp
Created file talk_pkg/package.xml
Created file talk_pkg/CMakeLists.txt
Created folder talk_pkg/include/talk_pkg
Created folder talk_pkg/src
Successfully created files in /home/mhs/mhs/helloworld/src/talk_pkg. Please adjust the values in package.xml.
mhs@mhs:~/mhs/helloworld/src$ ls
CMakeLists.txt  myPkg  talk_pkg
mhs@mhs:~/mhs/helloworld/src$
```

图 2.22 创建 talk\_pkg 包

从图 2.22 中，我们可以看到，catkin\_create\_pkg 命令生成了 talk\_pkg 包的框架结构，指明了包的编译依赖库有 std\_msgs 以及 python 和 C++支持。

接下来我们在 talk\_pkg 包里 src 目录下建立两个文件：publisher.cpp 和 subscriber.cpp。

publisher.cpp 文件内容如下：

```
#include "ros/ros.h"
```

```
#include "std_msgs/String.h"
```

```
#include <sstream>
```

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

```
{
```

```
    ros::init(argc, argv, "publisher"); //初始化 ROS 节点" publisher "
```

```
    ros::NodeHandle node;
```

```
    ros::Publisher talk_pub = node.advertise<std_msgs::String>("talk", 1000);
```

```
    ros::Rate loop_rate(10);
```

```
    int count = 0;
```

```
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
```

```
    {
```

```
        std_msgs::String msg;
```

```
        std::stringstream ss;
```

```
        ss << "hello world " << count;
```

```
        msg.data=ss.str();
```

```
        ROS_INFO("%s", msg.data.c_str());
```

```
        talk_pub.publish(msg);
```



```
        ros::spinOnce(); // Need to call this function often to allow ROS to
process incoming messages
```

```
        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop
rate
```

```
        count++;
    }
    return 0;
}
```

在 publisher.cpp 文件里，我们申明了一个“talk”主题消息发布者：

```
ros::Publisher talk_pub = node.advertise<std_msgs::String>("talk", 1000);
```

使用 `NodeHandle::advertise()` 函数向 ROS 主控节点注册了“talk”主题，主题消息类型为字符串，队列大小为 1000，一旦所有发布该主题的队列超出这个范围，该主题将不会被公告。

主题消息通过 `publish()` 函数发布。消息的类型必须符合申明时所调用的模板函数 `advertise <>()` 相同的参数类型。本程序示例使用 `while` 循环不停发布“talk”主题消息字符串“hello world”，发布频率设定为 10HZ。

subscriber.cpp 文件内容如下：

```
#include "ros/ros.h"
```

```
#include "std_msgs/String.h"
```

```
// Topic messages callback
```

```
void talkCallback(const std_msgs::String::ConstPtr& msg)
```

```
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

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

```
{
    // Initiate a new ROS node named "subscriber"
    ros::init(argc, argv, "subscriber");
    ros::NodeHandle node;

    // Subscribe to a given topic
```



```
ros::Subscriber sub = node.subscribe("talk", 1000, talkCallback);

// Enter a loop, pumping callbacks
ros::spin();

return 0;
}
```

在 subscriber.cpp 文件里，我们申明了一个“talk”主题消息订阅者：

```
ros::Subscriber sub = node.subscribe("talk", 1000, talkCallback);
```

subscribe()函数第一个参数是订阅主题名，第二个参数是最大队列数，第三个参数是处理消息的回调函数，我们可以在回调函数里对主题订阅消息进行处理。

ros::spin()创建一个循环，其中节点开始读取订阅的主题，并且当消息到达时，消息的回调函数将被调用。

如果 ros::ok()返回 false，那么 ros::spin()将退出循环。例如，当用户按 Ctrl + C 或当调用了 ros::shutdown()时，ros::spin()循环就会停止。

编写完主题发布者和主题订阅者示例程序后，我们需要修改 talk\_pkg 目录下的 CMakeLists.txt 文件，添加如下行文字：

```
# add_executable(talk_pkg_node src/talk_pkg_node.cpp)
add_executable(publisher src/publisher.cpp)
add_executable(subscriber src/subscriber.cpp)

## Add cmake target dependencies of the executable/library
## as an example, message headers may need to be generated before nodes
# add_dependencies(talk_pkg_node talk_pkg_generate_messages_cpp)

## Specify libraries to link a library or executable target against
# target_link_libraries(talk_pkg_node
#   ${catkin_LIBRARIES}
# )
target_link_libraries(publisher ${catkin_LIBRARIES})
target_link_libraries(subscriber ${catkin_LIBRARIES})
```

告诉 CMake 编译可执行文件 publisher 及 subscriber 所依赖的原文件及链接库文件。

做好这些工作后，使用 catkin\_make 工具来构建软件包并编译所有的节点：

```
$cd ~/mhs/helloworld
```



## \$catkin\_make

编译完成后，系统将构建完成 talk\_pkg 包，生成在两个可执行节点文件位于 /devel/lib/talk\_pkg 目录下，如图 2.23 所示。

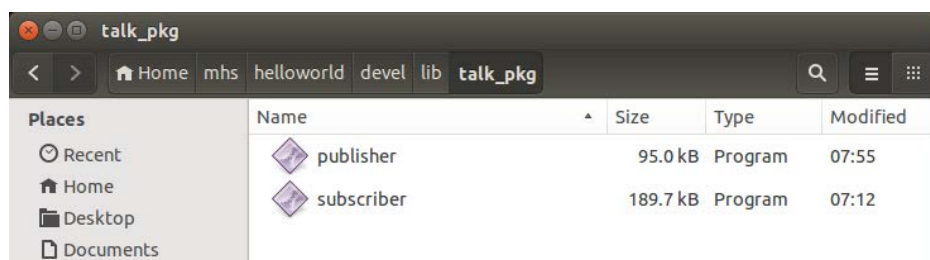


图 2.23 生成节点可执行节点程序

分别运行如下三个命令：

```
$roscore
```

```
$roslaunch talk_pkg publisher
```

```
$roslaunch talk_pkg subscriber
```

运行结果如图 2.24 所示。

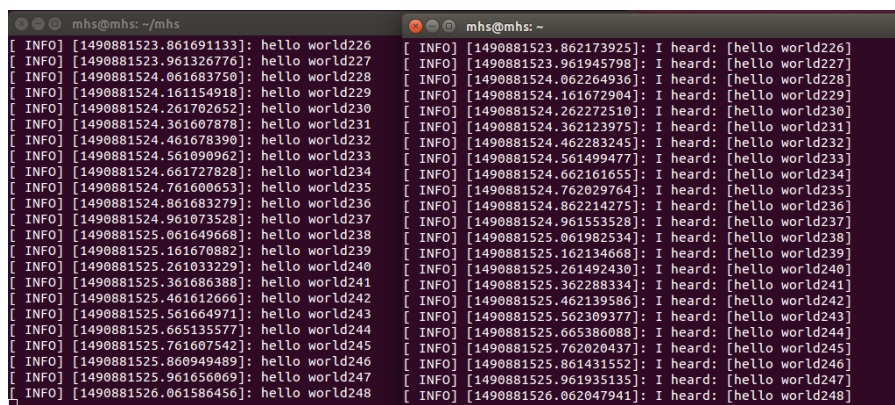


图 2.24 运行结果

我们可以使用 roscore 和 rostopic 来调试，并查看节点正在做什么，命令如下：

```
$roscore
```

```
$roslaunch talk_pkg publisher
```

```
$roslaunch talk_pkg subscriber
```

```
$rostopic list
```

```
$rostopic info /talk
```

```
$rostopic echo /talk
```

当然我们也可使用 rqt\_graph 查看动态的计算视图：

```
$roslaunch rqt_graph rqt_graph
```

运行之后，把鼠标移到上面，我们可以看到发布者节点显示为红色，订阅者显示为绿色。如图 2.25 所示。



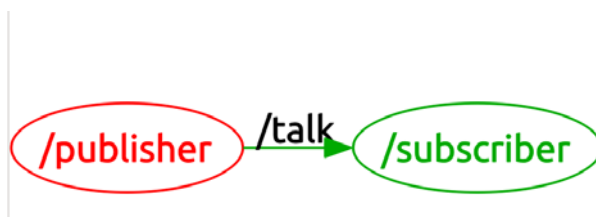


图 2.25 节点视图

接下来我们来看看如何在基于主题的通信模式中，如何使用自定义的消息类型。

首先我们在刚才的 `talk_pkg` 包下面建立一个 `msg` 目录，创建一个 `AandB.msg` 文本文件，内容如下：

```
float32 a
```

```
float32 b
```

接下来，我们修改前面的 `publisher.cpp` 文件：

```
#include "ros/ros.h"
```

```
#include "std_msgs/String.h"
```

```
#include <sstream>
```

```
#include "talk_pkg/AandB.h"
```

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

```
{
```

```
    ros::init(argc, argv, "publisher"); // Initiate new ROS node named  
    "talking"
```

```
    ros::NodeHandle node;
```

```
    //ros::Publisher talk_pub = node.advertise<std_msgs::String>("talk",  
    10);
```

```
    ros::Publisher talk_pub = node.advertise<talk_pkg::AandB>("talk", 10);
```

```
    ros::Rate loop_rate(10);
```

```
    int count = 0;
```

```
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
```

```
    {
```

```
        //std_msgs::String msg;
```

```
        talk_pkg::AandB msg;
```

```
        //std::stringstream ss;
```



```
//ss<<"hello world"<<count;

//msg.data=ss.str();
msg.a=1.0;
msg.b=2.0;

//ROS_INFO("%s", msg.data.c_str());
ROS_INFO("msg a: %.6f, msg b: %.6f", msg.a,msg.b);

talk_pub.publish(msg);

ros::spinOnce(); // Need to call this function often to allow ROS to
process incoming messages

loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the
loop rate
count++;
}
return 0;
}
```

在上面的例子中，我们使用 `talk_pkg::AandB` 消息类型替代了原先的 `std_msgs::String` 类型。

同样，在订阅主题消息的源文件 `subscriber.cpp` 文件，使用 `talk_pkg::AandB` 消息类型替代了原先的 `std_msgs::String` 类型，源代码改动如下：

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "talk_pkg/AandB.h"

// Topic messages callback
//void talkCallback(const std_msgs::String::ConstPtr& msg)
void talkCallback(const talk_pkg::AandB::ConstPtr& msg)
{
    //ROS_INFO("I heard: [%s]", msg->data.c_str());
    ROS_INFO("I heard: msg:a %f, msg:b %f,sum=%f", msg->a,
msg->b,msg->a+msg->b);
}
```



```
int main(int argc, char **argv)
{
    // Initiate a new ROS node named "subscriber"
    ros::init(argc, argv, "subscriber");
    ros::NodeHandle node;

    // Subscribe to a given topic
    ros::Subscriber sub = node.subscribe("talk", 1000, talkCallback);

    // Enter a loop, pumping callbacks
    ros::spin();

    return 0;
}
```

前面我们创建了 msg 文件，在 cpp 代码里包含了头文件：

```
#include "talk_pkg/AandB.h"
```

需要说明的是，我们需要在 package.xml 文件里添加编译支持以及运行支持：

```
<build_depend>message_generation</build_depend>
```

```
<run_depend>message_runtime</run_depend>
```

在 CMakeLists.txt 文件里添加关于 AandB..msg 文件的规则：

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

```
add_message_files(
    FILES
    AandB.msg
)
```

```
.....
```

```
generate_messages(
    DEPENDENCIES
```



```
    std_msgs
  )

  catkin_package(
    CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
  )

  include_directories(
    ${catkin_INCLUDE_DIRS}
  )

add_executable(publisher src/publisher.cpp)
add_executable(subscriber src/subscriber.cpp)

add_dependencies(publisher talk_pkg_generate_messages_cpp)
add_dependencies(subscriber talk_pkg_generate_messages_cpp)

target_link_libraries(publisher ${catkin_LIBRARIES})
target_link_libraries(subscriber ${catkin_LIBRARIES})
```

当完成这些工作后，使用 `catkin_make` 工具编译，系统会自动在 `/devel/include/talk_pkg` 目录下生成头文件 `AandB.h`，否则，系统编译将无法进行，无法找到包含的 `AandB.h` 头文件。

编译完成之后，运行如下命令测试：

```
$roscore
```

```
$roslaunch talk_pkg publisher
```

```
$roslaunch talk_pkg subscriber
```

需要注意在运行后两个命令的终端窗口里需要初始化本项目包的工作空间环境：

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

运行结果如图 2.26 所示。



```
mhs@mhs: ~
0000
[ INFO] [1490938641.738178149]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938641.838364962]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938641.938251795]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938642.038291756]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938642.138202688]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938642.238191093]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938642.338275770]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938642.438713269]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938642.538138653]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938642.637233862]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938642.740576281]: I heard: msg:a 1.000000, msg:b 2.000000,sum=3.00
0000
[ INFO] [1490938641.337773990]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938641.437825202]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938641.537763737]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938641.637747839]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938641.737767470]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938641.837860822]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938641.937775352]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938642.037828653]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938642.137779844]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938642.237803134]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938642.337796440]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938642.437947495]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938642.537793857]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938642.636846229]: msg a: 1.000000, msg b:2.000000
[ INFO] [1490938642.740096339]: msg a: 1.000000, msg b:2.000000
```

图 2.26 运行结果

## 2.6.4 编写 launch 文件

实际上, 在运行和调试过程中, 我们还可以使用 `roslaunch` 工具, 一次性启动我们所要运行的所有节点。`roslaunch` 是一个用于启动多个 ROS 节点以及在参数服务器上设置参数的工具, 它可以在一个或多个 XML 配置文件 (具有 `.launch` 扩展名) 中设置指定参数和要启动的节点。如果使用 `roslaunch`, 我们可以不必像前面范例那样手动运行 `roscore`。

以下是我们为运行调试 `talk_pkg` 包所写的示例 `talk.launch` 文件:

```
<launch>
  <node      name="publisher"      pkg="talk_pkg"      type="publisher"
output="screen"/>
  <node      name="subscriber"      pkg="talk_pkg"      type="subscriber"
output="screen"/>
</launch>
```

记住在运行 `talk.launch` 文件之前, 配置好环境路径:

```
$source ~/mhs/helloworld/devel/setup.bash
```



如果没有配置好环境,使用 `roslaunch` 命令将会出现无法找到 `publisher` 和 `subscriber` 节点的错误提示。配置好环境后,我们可以执行如下命令:

```
$roslaunch talk.launch
```

一次性运行 `roscore` 以及 `publisher`、`subscriber` 三个节点。`output="screen"` 表示 ROS 的信息输出在运行 `roslaunch` 的终端窗口。

```
<launch>
  <node      name="publisher"      pkg="talk_pkg"      type="publisher"
output="screen"/>
  <node      name="subscriber"      pkg="talk_pkg"      type="subscriber"
output="screen"/>
  <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node"/>
  <!--node pkg="turtlesim" type="turtle_teleop_key" name="teleop_key" /-->
  <node pkg="rostopic" type="rostopic" name="rostopic" args="pub
/turtle1/cmd_vel -r 10 geometry_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0}, angular:
{x: 0, y: 0, z: 0.5}}'"/>
</launch>
```

甚至我们也可以在 `launch` 文件发布主题消息给节点,在上面的 `launch` 文件中,我们还一次性加载了 `rostopic` 节点,每隔 10HZ 发布一次 `/turtle1/cmd_vel` 类型消息,使得我们在测试主题/订阅示例 `publisher`、`subscriber` 节点的同时,还使得小海龟在屏幕上沿着一个圆在运动。如果取消第 5 行的注释,我们同样也可以同时用方向键来控制小海龟运动。

## 2.6.5 编写主题 service 节点

多对多、单向传输对于分布式系统中经常需要的同步式交互来说并不合适,需要采取“请求/回应”模式来实现,这样的一种通信在 ROS 中是通过 `services` 模式来实现的。`service` 通信定义一种成对的消息:一个用于请求,一个用于回应。

我们还是在上面的 `talk_pkg` 例程中实现基于 `service` 通信。

首先我们在刚才的 `talk_pkg` 包下面建立一个 `srv` 目录,创建一个 `AddTwoInts.drv` 文本文件,内容如下:

```
int64 A
int64 B
---
int64 Sum
```

其中 `A` 和 `B` 是请求,而 `Sum` 是响应。

编写服务端程序 `add_two_ints_server.cpp` 程序代码如下:





```
#include "ros/ros.h"
#include "talk_pkg/AddTwoInts.h"

bool add(talk_pkg::AddTwoInts::Request &req,
         talk_pkg::AddTwoInts::Response &res)
{
    res.Sum = req.A + req.B;
    ROS_INFO("request: x=%ld, y=%ld", (long int)req.A, (long int)req.B);
    ROS_INFO("sending back response: [%ld]", (long int)res.Sum);
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "add_two_ints_server");
    ros::NodeHandle n;

    ros::ServiceServer service = n.advertiseService("add_two_ints", add);
    ROS_INFO("Ready to add two ints.");
    ros::spin();

    return 0;
}
```

编写客户端程序 add\_two\_ints\_client.cpp 程序代码如下:

```
#include "ros/ros.h"
#include "talk_pkg/AddTwoInts.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "add_two_ints_client");
    if (argc != 3)
    {
        ROS_INFO("usage: add_two_ints_client X Y");
        return 1;
    }
}
```



```
ros::NodeHandle n;
ros::ServiceClient client = n.serviceClient<talk_pkg::AddTwoInts>("add_two_ints");
talk_pkg::AddTwoInts srv;
srv.request.A = atoll(argv[1]);
srv.request.B = atoll(argv[2]);
if (client.call(srv))
{
    ROS_INFO("Sum: %ld", (long int)srv.response.Sum);
}
else
{
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
}
return 0;
}
```

在编译之前，我们需要在原来的 CMakeLists.txt 文件里添加如下内容：

```
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(add_two_ints_server talk_pkg_gencpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(add_two_ints_client talk_pkg_gencpp)
```

以上脚本命令的功能是告诉 catkin\_make 工具如何编译 add\_two\_ints\_server 和 add\_two\_ints\_client 两个节点的源文件、链接库包含以及生成 AddTwoInts.h 的依赖规则。

编译完成后，测试运行命令如下：（注意在运行 add\_two\_ints\_server 和 add\_two\_ints\_client 的终端窗口，使用 source 命令设置 talk\_pkg 包的环境，否则新开的终端窗口将无法自动定位节点可执行文件的存放路径）

```
$roscore
$roslaunch talk_pkg add_two_ints_server
$ roslaunch talk_pkg add_two_ints_client 1 2
运行效果如图 2.27、2.28 所示。
```



```
mhs@mhs: ~/mhs/helloworld
mhs@mhs:~/mhs/helloworld$ rosrn talk_pkg add_two_ints_server
[ INFO] [1490945961.885316253]: Ready to add two ints.
[ INFO] [1490945966.570615300]: request: x=1, y=2
[ INFO] [1490945966.570695113]: sending back response: [3]
```

图 2.27 运行结果一

```
mhs@mhs: ~/mhs
mhs@mhs:~/mhs$ rosrn talk_pkg add_two_ints_client 1 2
[ INFO] [1490945966.571328578]: Sum: 3
mhs@mhs:~/mhs$
```

图 2.28 运行结果二

## 2.6.5 编写 Action 节点

上面我们讲到了 ROS 中常用的 topic 和 service 通讯机制，一般来说如果用 topic 发布，由于 topic 没有反馈，我们无法获得控制对象的运动状态。如果使用 service 通讯机制，虽然可以获得反馈，但如果服务端迟迟没有响应，那么客户端程序将处于阻塞状态，直到超时。

为了解决这个问题，ROS 还提供了一个 actionlib 的功能包集，实现基于 action 通信，action 机制也是一种类似于 service 的问答通讯机制，不一样的是 action 还带有一个反馈机制，可以不断反馈任务的实施进度，而且可以在任务实施过程中，中止运行。其工作模式见图 2.29 所示。通俗的理解就是，采用 action 机制在调用服务器上某服务后，可以转而去执行其他的程序，周期性的获得服务器上的信息，或者取消发送到服务器上的远程调用，机制更加灵活一些。ROS action 接口见图 2.30 所示。

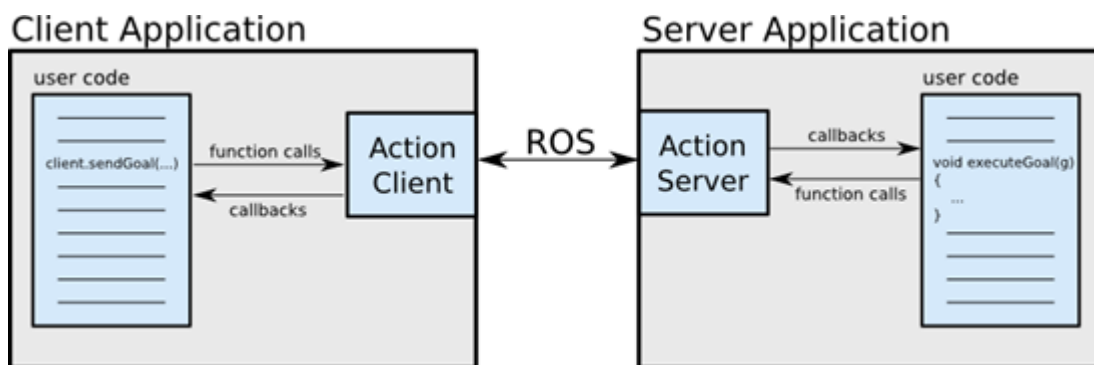


图 2.29 基于 action 的 client/server 工作模式



## Action Interface

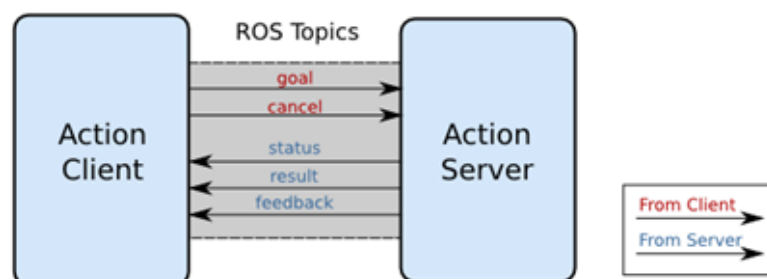


图 2.30 ROS action 接口

从图 2.30 中我们可以看出，Action client 向 Action server 端发布任务目标以及在必要的时候可以取消任务，Action server 向 Action client 发布当前的状态(status)、任务结果(result)和实时反馈(feedback)。

- ★**goal**: 用来发送一个新的任务目标给 server
- ★**cancel**: 用来发送请求取消任务
- ★**status**: 通知 client 端当前的每个目标(个 goal)的状态
- ★**feedback**: 反馈，用于向客户端发送定期辅助信息以实现目标
- ★**result**: 向 client 端发送任务的执行结果，这个 topic 只会发布一次。

现在我们开始编写一个最简单的 action 例子，完成服务器端和客户端程序的编写。

### 1) 第 1 步：创建包

```
cd ~/catkin_ws/src
catkin_create_pkg myAction roscpp actionlib message_generation std_msgs
actionlib_msgs
```

### 2) 第 2 步，创建 Action 消息

注意：

- action 有三种消息：goal, result, 以及 feedback
- 它们是通过.action 文件自动生成的。

下面我们开始创建 Fibonacci.action

```
cd ~/catkin_ws/src/myAction/
mkdir action
cd ~/catkin_ws/src/myAction/action
touch Fibonacci.action
vim Fibonacci.action
```

代码如下：

```
#goal definition
int32 order
---
```



```
#result definition
```

```
int32[] sequence
```

```
---
```

```
#feedback
```

```
int32[] sequence
```

3) 第 3 步, 接下来, 我们需要更改 CMakeLists.txt 文件, 增加 actionlib\_msgs 支持 (一般来说刚才我们使用 catkin\_create\_pkg 命令的时候, 后面已经加上了 (roscpp actionlib message\_generation std\_msgs actionlib\_msgs) 参数后缀, 所以这些支持估计已经都有了, 不需再添加), 需要有如下行, 没有则增加:

```
find_package(catkin REQUIRED COMPONENTS actionlib_msgs)
```

但需要增加 add\_action\_files:

```
add_action_files(
  DIRECTORY action
  FILES Fibonacci.action
)
```

继续添加 generate\_messages:

```
generate_messages(
  DEPENDENCIES actionlib_msgs std_msgs # Or other packages containing
msgs
)
```

继续增加 actionlib\_msgs 到 catkin\_package:

```
catkin_package(
  CATKIN_DEPENDS actionlib_msgs
)
```

4) 第 4 步, 编译并查看结果:

```
$ cd ~/catkin_ws
$ catkin_make
$ ls devel/share/myAction/msg/
```

我们可以看到系统生成了如下文件, 见图 2.31 所示。

FibonacciActionFeedback.msg	FibonacciAction.msg	FibonacciFeedback.msg
FibonacciResult.msg	FibonacciActionGoal.msg	FibonacciActionResult.msg
		FibonacciGoal.msg

图 2.31 生成的 msg 文件

并且在 devel/include/myAction/目录下有相应的头文件, 见图 2.32。

FibonacciActionFeedback.h	FibonacciAction.h	FibonacciFeedback.h	FibonacciResult.h
FibonacciActionGoal.h	FibonacciActionResult.h	FibonacciGoal.h	

图 2.31 生成对应的头文件



5) 第 5 步, 我们在 myAction/src/ 目录下开始编写这个 action 的服务器程序。

文件命名为 fibonacci\_server.cpp, 代码如下:

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

```
#include <actionlib/server/simple_action_server.h>
```

//actionlib/server/simple\_action\_server.h 是 action 库, 用来执行简单的 actions

```
#include <myAction/FibonacciAction.h>
```

//这个头文件是由 Fibonacci.action 自动生成的

```
class FibonacciAction
```

```
{
```

```
protected:
```

```
    ros::NodeHandle nh_;
```

```
    actionlib::SimpleActionServer<myAction::FibonacciAction>    as_;    //
```

NodeHandle instance must be created before this line. Otherwise strange error occurs.

```
    std::string action_name_;
```

```
    // create messages that are used to published feedback/result
```

```
    myAction::FibonacciFeedback feedback_;
```

```
    myAction::FibonacciResult result_;
```

```
public:
```

```
    FibonacciAction(std::string name) :
```

```
        as_(nh_, name, boost::bind(&FibonacciAction::executeCB, this, _1), false),
```

//executeCB 函数引用在构造函数中创建, 回调函数传递 boost 的共享指针类型 ConstPtr 的 goal 作为参数

```
        action_name_(name)
```

```
{
```

```
    as_.start();
```

```
}
```

```
    ~FibonacciAction(void)
```

```
{
```





```
}
```

// executeCB 函数引用在构造函数中创建, 回调函数传递 boost 的共享指针类型 ConstPtr 的 goal 作为参数

```
void executeCB(const myAction::FibonacciGoalConstPtr &goal)
```

```
{
```

```
    // helper variables
```

```
    ros::Rate r(1);
```

```
    bool success = true;
```

```
    // push_back the seeds for the fibonacci sequence
```

```
    feedback_.sequence.clear();
```

```
    feedback_.sequence.push_back(0);
```

```
    feedback_.sequence.push_back(1);
```

```
    // publish info to the console for the user
```

```
    ROS_INFO("%s: Executing, creating fibonacci sequence of order %i with  
seeds %i, %i", action_name_.c_str(), goal->order, feedback_.sequence[0],  
feedback_.sequence[1]);
```

```
    //内部执行开始, 显示接收的值
```

```
    //action 服务器重要功能是可以接受客户端取消目前目标的执行。
```

```
    //当客户端请求当前取消的目标优先处理, action 服务器就会清除相关内容, 并调用 setPreempted()函数。
```

```
    //也可以设置优先处理检查的频率, action 服务器就自动根据这个频率实现检查并做对应处理。
```

```
    // start executing the action
```

```
    for(int i=1; i<=goal->order; i++)
```

```
    {
```

```
        // check that preempt has not been requested by the client
```

```
        if (as_.isPreemptRequested() || !ros::ok())
```

```
        {
```

```
            ROS_INFO("%s: Preempted", action_name_.c_str());
```

```
            // set the action state to preempted
```

```
            as_.setPreempted();
```

```
            success = false;
```

```
            break;
```



```
    }
    feedback_.sequence.push_back(feedback_.sequence[i] +
feedback_.sequence[i-1]);
    // publish the feedback, Fibonacci 数列会存放在变量 feedback_里, 并
    发布这个 feedback_, 然后进入新一轮的循环
    as_.publishFeedback(feedback_);
    // this sleep is not necessary, the sequence is computed at 1 Hz for
    demonstration purposes
    r.sleep();
}
//当计算完成, action 服务器就会通知客户端已经完成, 并发送最后的结
果
if(success)
{
    result_.sequence = feedback_.sequence;
    ROS_INFO("%s: Succeeded", action_name_.c_str());
    // set the action state to succeeded
    as_.setSucceeded(result_);
}
}

};

//main 函数实现开始 action, 并启动线程处理, 运行和等待去接受 goal 值
传入
int main(int argc, char** argv)
{
    ros::init(argc, argv, "fibonacci");

    FibonacciAction fibonacci("fibonacci");
    ros::spin();

    return 0;
}
```



当 C++ 代码写完后，在编译之前我们还需要更改 CMakeLists.txt，增加：

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

```
target_link_libraries(
    fibonacci_server
    ${catkin_LIBRARIES}
)
```

```
add_dependencies(
    fibonacci_server
    ${myAction_EXPORTED_TARGETS}
)
```

6) 可以开始编译服务器代码了：

```
cd ~/catkin_ws
catkin_make
```

7) 运行测试一下

开一个终端运行：

```
$ roscore
```

再开一个新终端，执行 action 服务器：

```
$ rosrun myAction fibonacci_server
```

至此，我们完成了一个简单的 action 服务器程序的编写，下面我们来看看如何编写一个 client 程序，如何发送 goal，并获取服务器的状态。

8) 编写 action client 程序

fibonacci\_client.cpp 文件代码如下：

```
#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <myAction/FibonacciAction.h>
```

```
int main (int argc, char **argv)
{
    ros::init(argc, argv, "test_fibonacci");

    // create the action client
    // true causes the client to spin its own thread
```



```
actionlib::SimpleActionClient<myAction::FibonacciAction> ac("fibonacci",
true);
```

```
ROS_INFO("Waiting for action server to start.");
// wait for the action server to start
ac.waitForServer(); //will wait for infinite time
```

```
ROS_INFO("Action server started, sending goal.");
// send a goal to the action
myAction::FibonacciGoal goal;
goal.order = 20;
ac.sendGoal(goal); //向服务器端发送 goal
```

```
//wait for the action to return
bool finished_before_timeout = ac.waitForResult(ros::Duration(30.0));
```

```
if (finished_before_timeout)
{
    actionlib::SimpleClientGoalState state = ac.getState();
    ROS_INFO("Action finished: %s",state.toString().c_str());
}
else
    ROS_INFO("Action did not finish before the time out.");
```

```
//exit
return 0;
```

```
}
```

同样我们需要在 CMakeLists.txt 文件里添加几行：

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

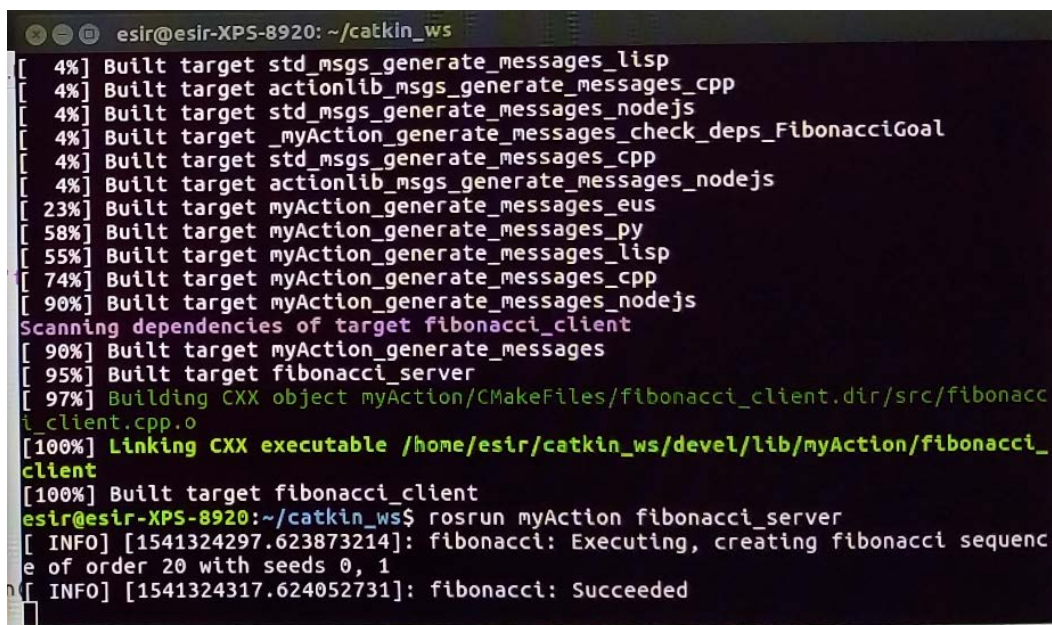
```
target_link_libraries(
    fibonacci_client
    ${catkin_LIBRARIES}
)
```



```
add_dependencies(  
  fibonacci_client  
  ${myAction_EXPORTED_TARGETS}  
)
```

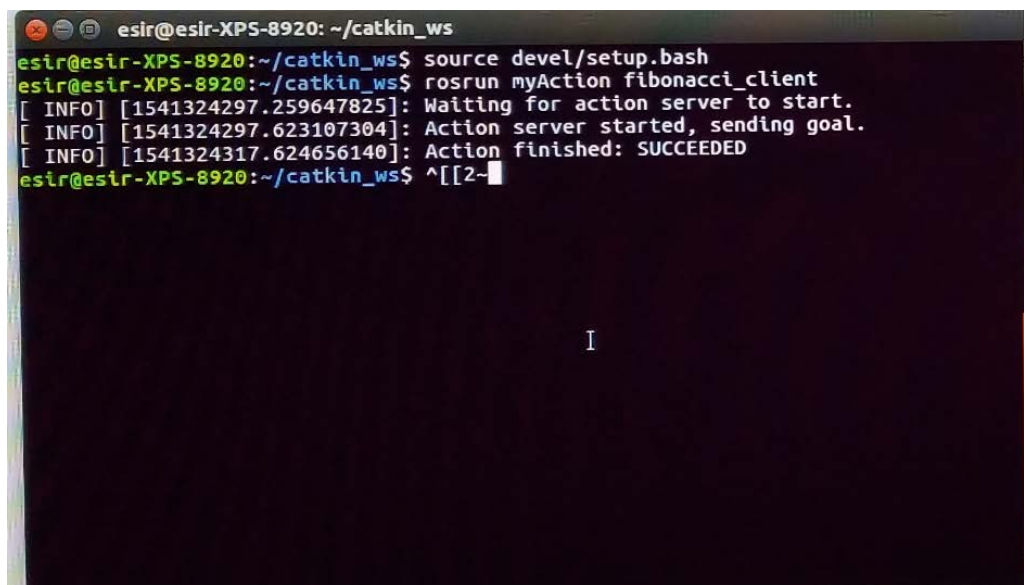
然后回到 catkin\_ws 目录，执行 catkin\_make 编译。

9) 如果客户端程序编译成功，那么我们需要另开一个终端来测试 action 的客户端程序，执行结果如图 2.31 所示。



```
esir@esir-XPS-8920: ~/catkin_ws  
[ 4%] Built target std_msgs_generate_messages_lisp  
[ 4%] Built target actionlib_msgs_generate_messages_cpp  
[ 4%] Built target std_msgs_generate_messages_nodejs  
[ 4%] Built target _myAction_generate_messages_check_deps_FibonacciGoal  
[ 4%] Built target std_msgs_generate_messages_cpp  
[ 4%] Built target actionlib_msgs_generate_messages_nodejs  
[ 23%] Built target myAction_generate_messages_eus  
[ 58%] Built target myAction_generate_messages_py  
[ 55%] Built target myAction_generate_messages_lisp  
[ 74%] Built target myAction_generate_messages_cpp  
[ 90%] Built target myAction_generate_messages_nodejs  
Scanning dependencies of target fibonacci_client  
[ 90%] Built target myAction_generate_messages  
[ 95%] Built target fibonacci_server  
[ 97%] Building CXX object myAction/CMakeFiles/fibonacci_client.dir/src/fibonacci_client.cpp.o  
[100%] Linking CXX executable /home/esir/catkin_ws/devel/lib/myAction/fibonacci_client  
[100%] Built target fibonacci_client  
esir@esir-XPS-8920:~/catkin_ws$ rosrn myAction fibonacci_server  
[ INFO] [1541324297.623873214]: fibonacci: Executing, creating fibonacci sequence of order 20 with seeds 0, 1  
[ INFO] [1541324317.624052731]: fibonacci: Succeeded
```

图 2.31 服务器端运行界面



```
esir@esir-XPS-8920: ~/catkin_ws  
esir@esir-XPS-8920:~/catkin_ws$ source devel/setup.bash  
esir@esir-XPS-8920:~/catkin_ws$ rosrn myAction fibonacci_client  
[ INFO] [1541324297.259647825]: Waiting for action server to start.  
[ INFO] [1541324297.623107304]: Action server started, sending goal.  
[ INFO] [1541324317.624656140]: Action finished: SUCCEEDED  
esir@esir-XPS-8920:~/catkin_ws$ ^[[2-
```

图 2.32 客户端运行界面

在上述的例程中，在客户端程序我们设置客户端等待服务器完成，时间间隔为 30 秒，超时则返回 false。

使用 action 的好处在于，客户端可以获取服务器端对象的状态，并且可以使



用它的 cancel 消息取消某个任务。例如：

```
ac.cancelGoal(); //取消 goal
```

这样的机制在我们后面的机器人任务规划与导航的应用中将使用到，例如我们发送某种任务(goal 数据)给机器人，机器人根据运行状态可以判断是否继续执行任务，还是取消任务。

## 2.6.6 ROS Parameter Server

ROS Master 还提供了参数服务器(Parameter Server)。**Parameter** 可以看作为 ROS 系统运行时中定义的全局变量，而 master 节点中由参数服务器 parameter server 来维护这些变量。ROS 采用名字命名空间(namespace)，使得参数 parameter 拥有非常清晰的层次划分，避免重名，而且使得参数 parameter 访问可以单独访问也可以树状访问。

有关 **Parameter** 前面已经涉及到简单的使用，本节不单独去讲参数服务器的使用，在后续实验中如何用到将再细讲。

## 2.7 ROS 与工业机器人

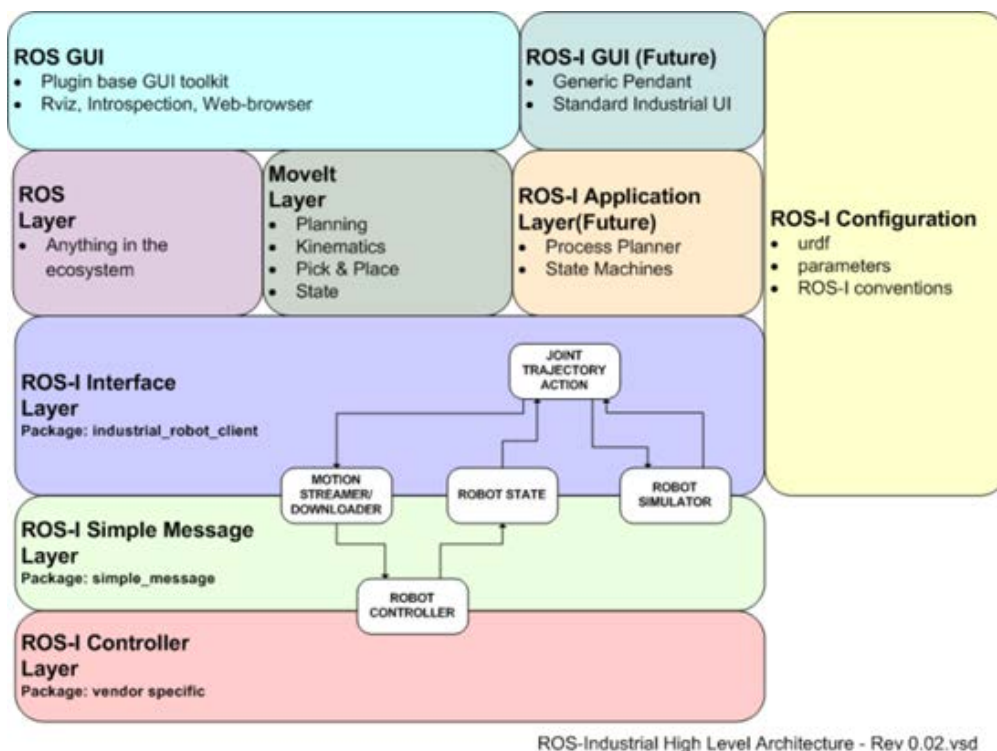


图 2.33 ROS 与工业机器人高级架构

机器人的仿真环境非常之多，本教程旨在进行机械臂、移动机器人、复合机器人的仿真与控制，涉及工业机器人的部分较多，在此简单介绍 ROS-Industrial，其架构见图 2.33 所示。



- GUI: 上层 UI 分为两个部分: 一个部分是 ROS 中现在已有的 UI 工具; 另外一个部分是专门针对工业机器人通用的 UI 工具, 不过是将来才会实现;
- ROS Layer: ROS 基础框架, 提供核心通讯机制;
- MoveIt! Layer: 为工业机器人提供规划、运动学等核心功能的解决方案;
- ROS-I Application Layer: 处理工业生产的具体应用, 也是针对将来的规划;
- ROS-I Interface Layer: 接口层, 包括工业机器人的客户端, 可以通过 simple message 协议与机器人的控制器通信;
- ROS-I Simple Message Layer: 通信层, 定义了通信的协议, 打包和解析通信数据;
- ROS-I Controller Layer: 机器人厂商开发的工业机器人控制器。

从上边的架构我们可以看到, ROS 与工业机器人复用了已有的 ROS 框架、功能基础上, 针对工业领域进行了针对性的拓展, 而且可以通用于不同厂家的机器人控制器。

这种架构的上层控制本身就是复用已有的软件包, 本教程旨在介绍如何使用这些软件包来控制工业机械臂与移动底盘, 这些软件包包括 Rviz (主要可以进行运动学仿真与控制)、Gazebo (除了运动学外, 还可以进行动力学仿真与控制) 和 Moveit (包含很多开源的运动学与轨迹规划库, 由于它比较复杂, 所以本教程刚开始将不使用 Moveit 框架进行运动学讲解) 等, 这些内容随着教程的深入将逐步涉及, 在此不做详细介绍, 在后续的实验教程中再另行解释。

## 2.8 本章版权及声明

本教程为武汉科技大学机器人与智能系统研究院闵华松教授实验室内部教学内容, 未经授权, 任何商业行为个人或组织不得抄袭、转载、摘编、修改本章内容; 任何非盈利性个人或者组织可以自由传播 (禁止修改、断章取义等) 本网站内容, 但是必须注明来源。

本章内容由闵华松 (mhuasong@wust.edu.cn)、王玉卓 (2543268429@qq.com) 编写。