目錄

教程

引言

ROS2是新的ROS版本。相对与旧版本更加接近工业化场景，更加稳定，同时功能也更加丰富。

Linux下的安装

通过Debian包安装

在Beta2版本我们生成了ROS2的deb软件包（用于Ubuntu 16.04版本）。这些文件在一个为了测试用的临时软件源内。下面的链接和指令用于安装最新的ROS2版本-当前版本ardent。

相关资源信息

- Jenkins Instance
- Repositories
- Status Pages (amd64 arm64)

设置软件源

想要安装deb软件包，你需要现在自己的apt软件源列表内添加我们的软件源。

首先你要像下面一样添加gpg密钥到自己的电脑中

```
sudo apt update && sudo apt install curl
curl http://repo.ros2.org/repos.key | sudo apt-key add -
```

然后执行下面的指令添加软件源到自己的apt源列表中

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

安装ROS2软件包

下面的指令会安装所有的**ros-ardent-*** 软件包除了**ros-ardent-ros1-bridge** 和 **ros-ardent-turtlebot2-***。因为这两个软件包依赖于ROS 1的软件包。

按照下面的指令安装对应的软件包

```
sudo apt update
sudo apt install `apt list "ros-ardent-*" 2> /dev/null | grep "/" | awk -F/ '{print $1}' | grep -v -e ros-arden
t-ros1-bridge -e ros-ardent-turtlebot2- | tr "\n" " "`
```

环境变量设置

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

如果你安装了Python的软件包 argcomplete(0.8.5 或以上版本)你可以通过执行下面的指令来为ROS2的命令行工具添加自动补全功能。

```
source /opt/ros/ardent/share/ros2cli/environment/ros2-argcomplete.bash
```

argcomplete可以通过

```
sudo pip install argcomplete
```

来安装。

选择 RMW 实现

默认的RMW实现是 FastRTPS。

通过设置环境变量 RMW_IMPLEMENTATION=rmw_opensplice_cpp。你可以切换成OpenSplice。

额外的依赖于ROS1的软件包

ros1_bridge和TurtleBot的例子程序一样都是依赖于ROS1的软件包。

为了能够安装这些软件包，请安装这里的说明添加ROS1的软件源。

如果你在使用docker，你可以设置docker image为ros:kinetic或osrf/ros:kinetic-desktop。由于软件已经集成在镜像中这样就省去了配置安装的麻烦。

安装完成之后，现在你可以开始安装下面剩余的软件包了。

```
sudo apt update
sudo apt install ros-ardent-ros1-bridge ros-ardent-turtlebot2-*
```

当然如果你对这些软件包没有兴趣，也可以不用安装。

Windows下的安装

系统要求

在beta-1版本我们支持Windows8.1和Windows10. 在beta-2版本我们只支持Windows10。

安装依赖

安装Chocolatey

Chocolatey 是一个windows下的软件包管理程序。可以通过他们的安装说明安装。

https://chocolatey.org/

之后你会用Chocolatey来安装其他开发工具

安装Python

打开一个命令行工具。同时按下Win + R在弹出的窗口中输入cmd。在命令行工具中输入下面的指令通过Chocolatey来安装Python。

```
choco install -y python
```

注意如果安装失败，比如我就遇到了这个情况。你可以手动安装，直接下载Python官网的安装包，然后安装，不过要安装到C:\Python36这个路径下面。

安装OpenSSL

从这个页面中下载OpenSSL安装包。

按照默认设置安装这个软件。接着添加环境变量(下面的指令默认你是按照默认参数安装的)

- `setx -m OPENSSL_CONF C:\OpenSSL-Win64\bin\openssl.cfg`
- 添加 C:\OpenSSL-Win64\bin\ 到你的PATH环境变量中

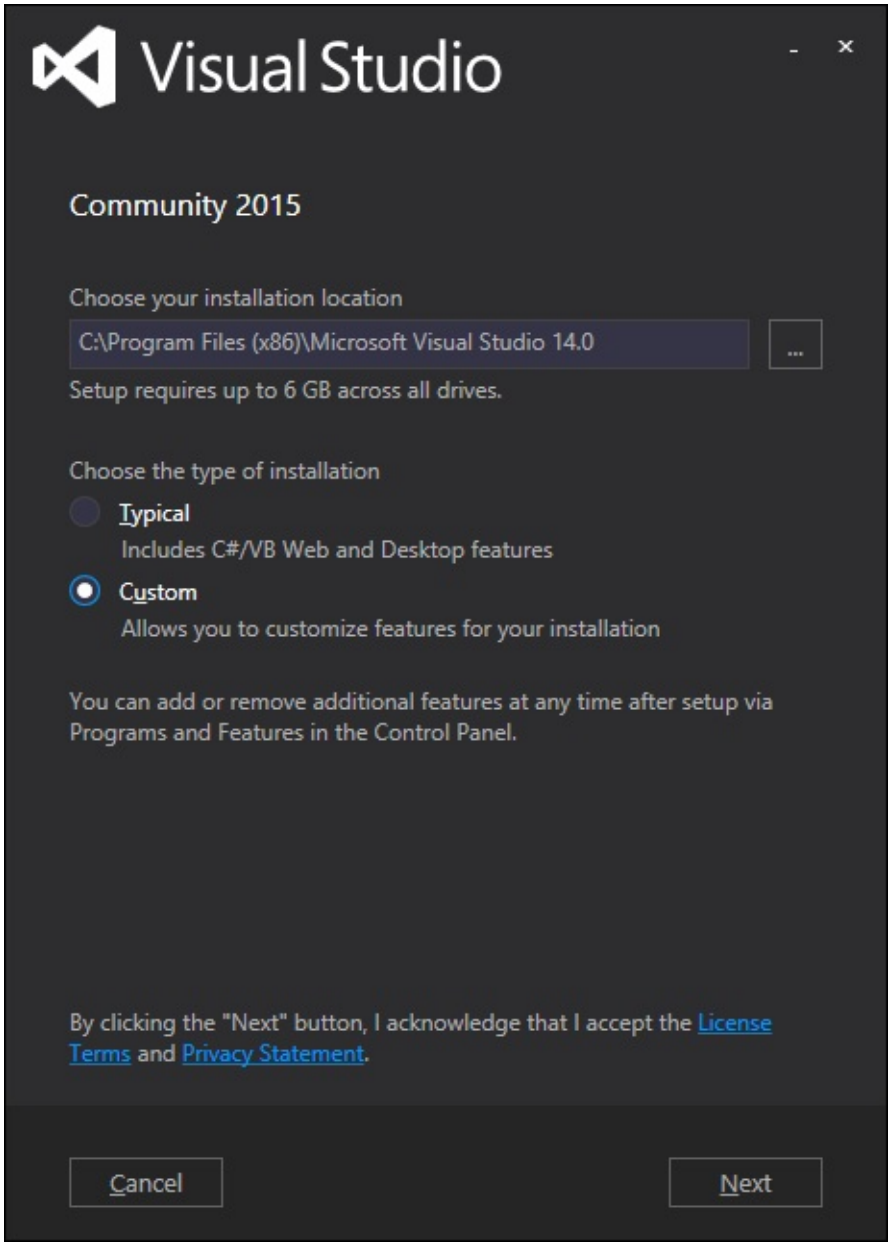
安装Visual Studio Community 2015

Microsoft 提供了一个Visual Studio的免费版本，叫做community。我们可以用它来编译ROS2的应用程序。

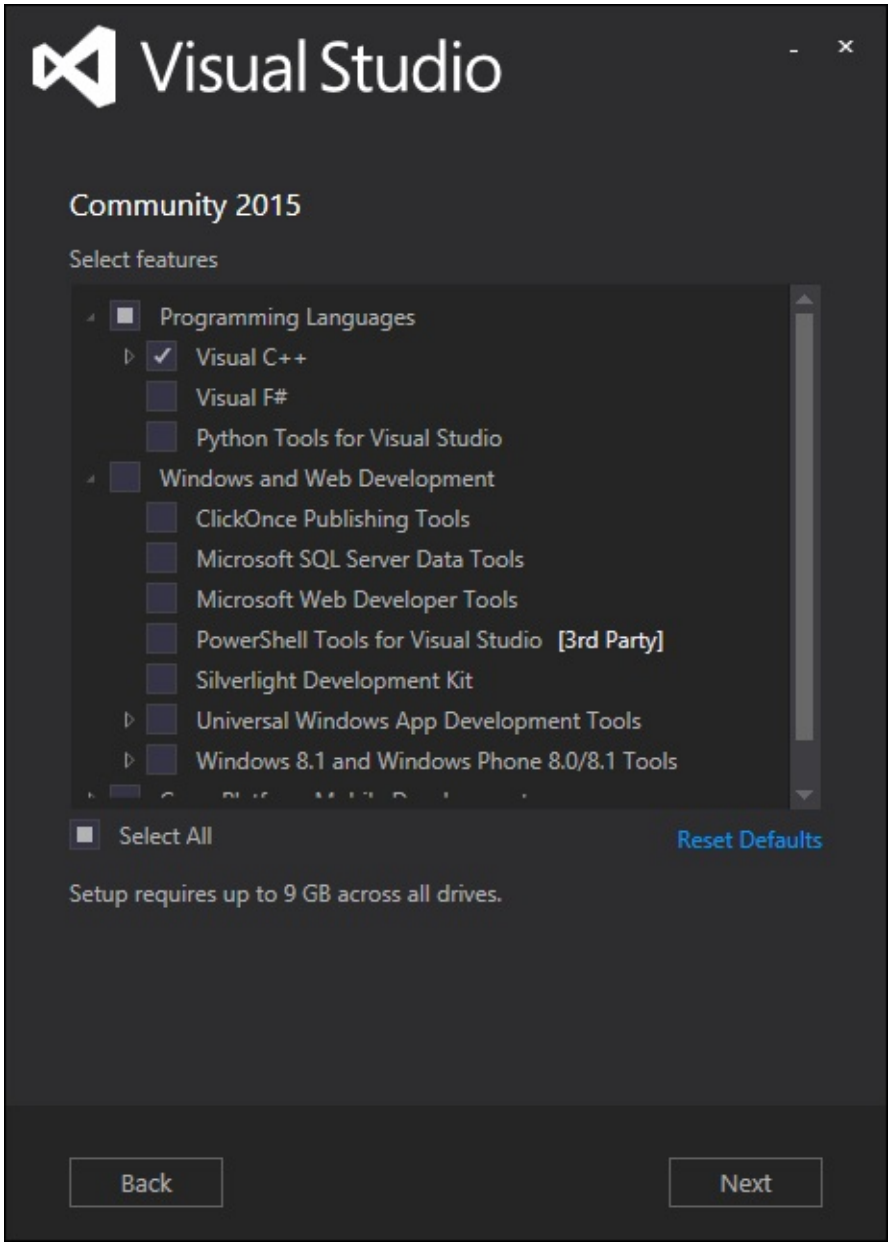
建议选择英文版安装，因为出错时的错误信息是英语，这样搜索的时候更容易搜到。

https://www.visualstudio.com/vs/older-downloads/

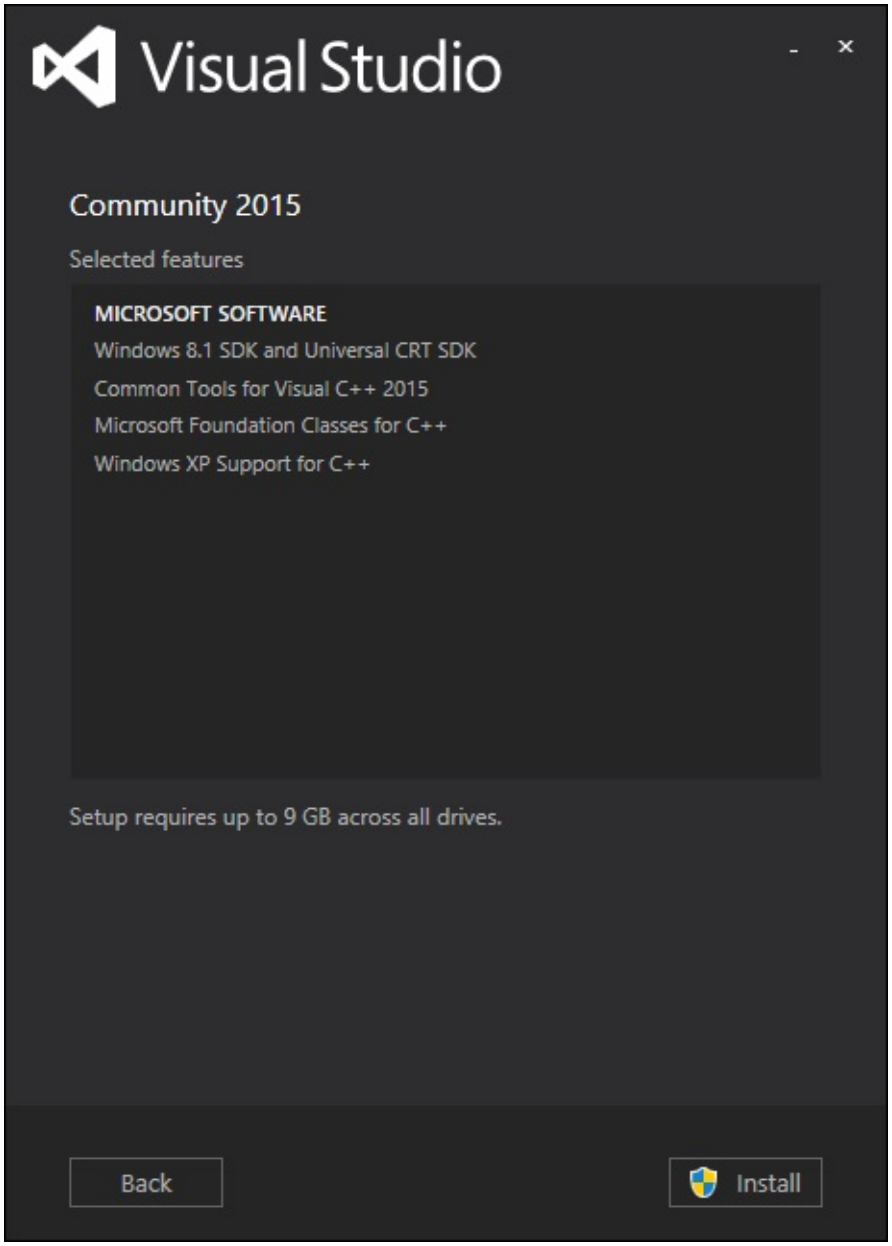
确保在安装时选择了Visual C++的功能。首先选择Custom installlation



接着选上Visual C++



再次确认已经选择上了对应的功能



安装一个DDS程序

二进制软件包中已经默认绑定了 eProsima FastRTPS 和 Adlink OpenSplice 作为中间件。如果你想用其他的DDS软件。那么你需要使用源代码安装。

eProsima FastRTPS & Boost(只能在beta-1或之后的版本中使用)

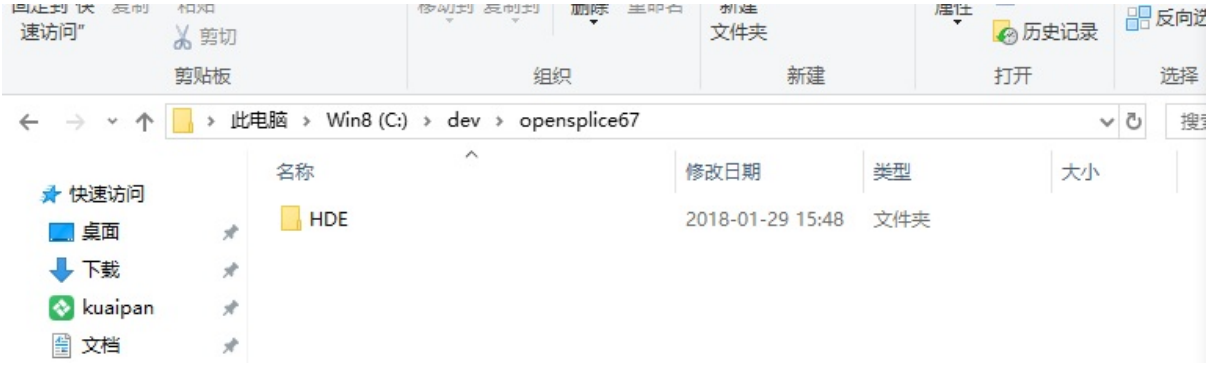
FastRTPS 依赖于boost库。在这里下载安装包后安装。

这个安装包默认安装在C:\local。安装完成后添加下面的环境变量到系统中。

```
PATH=C:\local\boost_1_61_0\lib64-msvc-14.0
```

Adlink OpenSplice

如果你想使用默认的DDS程序即上面的FastRTPS那么你不需要进行下面的操作。 如果你想使用OpenSplice作为DDS软件。你需要下载最新的版本(ROS2要求的最低版本为6.7.170912).解压到C:\dev\opensplice67，注意文件夹的层级。你的文件夹结构应该如下图所示



安装OpenCV

之后的教程例子有些依赖于OpenCV. 你可以下载一个预先编译好的版本 https://github.com/ros2/ros2/releases/download/release-beta2/opencv-2.4.13.2-vc14.VS2015.zip 如果你使用的是预先编译好的ROS版本。那么你需要设置下面几个环境变量来告诉ROS2在哪里找OpenCV的库。假设你把OpenCV解压到C:\dev\文件夹下。你需要在PATH环境变量里面添加下面的变量 `c:\dev\opencv-2.4.13.2-vc14.VS2015\x64\vc14\bin`

安装依赖

有些依赖文件并不在Chocolatey的软件库里面为了简化安装过程，我们提供了下面的Chocolatey软件包。

你可以在这里下载对应的软件包。

- asio.1.10.6.nupkg
- eigen-3.3.3.nupkg
- tinyxml-usestl.2.6.2.nupkg
- tinyxml2.4.1.0.nupkg 下载完成之后重新打开一个具有管理员权限的命令行程序，然后运行下面的语句

```
choco install -y -s <PATH\TO\DOWNLOADS\> asio eigen tinyxml-usestl tinyxml2
```

注意把 `<PATH\TO\DOWNLOADS>` 替换成你实际的下载位置。 你还需要安装 `pip` 和 `yaml`

```
python -m pip install -U pyyaml setuptools
```

安装时可能会出错，一般是编码的问题，你可以执行下面的语句设置命令行程序的编码

```
chcp 65001  //换成65001代码页
chcp 437    //换成美国英语
```

设置完成之后再执行上面的指令安装。

下载ROS2

- 进入ROS2的发布页面: https://github.com/ros2/ros2/releases
- 下载最新的Windows软件包
- 解压这个zip文件（推荐解压到C:\dev\ros2里面）

设置ROS2的环境

打开一个命令行程序然后source ROS2 的配置文件来自动配置好工作空间。 注意下面的指令只能通过cmd程序执行，powershell没法运行。

```
call C:\dev\ros2\local_setup.bat
```

执行过程中可能会提示找不到路径的错误。没关系可以继续进行下面的操作。 如果你下载了一个有OpenSplice支持的版本，而且你想用OpenSplice作为DDS程序，那么你可以执行下面的语句。反之则不用执行。

```
call "C:\opensplice67\HDE\x86_64.win64\release.bat"
```

这个语句要在上面一条语句执行完成之后才行执行。

尝试运行简单的例子程序

打开一个命令行程序，按照上面的说明设置好ROS2的环境。然后执行一个talker程序

```
ros2 run demo_nodes_cpp talker
```

再打开一个命令行，执行一个listener程序

```
ros2 run demo_nodes_py listener
```

```
C:\Users\newme_000>ros2 run demo_nodes_cpp talker
[INFO] [talker]: Publishing: 'Hello World: 1'
[INFO] [talker]: Publishing: 'Hello World: 2'
[INFO] [talker]: Publishing: 'Hello World: 3'
[INFO] [talker]: Publishing: 'Hello World: 4'
[INFO] [talker]: Publishing: 'Hello World: 5'
[INFO] [talker]: Publishing: 'Hello World: 6'
[INFO] [talker]: Publishing: 'Hello World: 7'
[INFO] [talker]: Publishing: 'Hello World: 8'
[INFO] [talker]: Publishing: 'Hello World: 9'
[INFO] [talker]: Publishing: 'Hello World: 10'
[INFO] [talker]: Publishing: 'Hello World: 11'
[INFO] [talker]: Publishing: 'Hello World: 12'
[INFO] [talker]: Publishing: 'Hello World: 13'
[INFO] [talker]: Publishing: 'Hello World: 14'
[INFO] [talker]: Publishing: 'Hello World: 15'
[INFO] [talker]: Publishing: 'Hello World: 16'
[INFO] [talker]: Publishing: 'Hello World: 17'
[INFO] [talker]: Publishing: 'Hello World: 18'
[INFO] [talker]: Publishing: 'Hello World: 19'
[INFO] [talker]: Publishing: 'Hello World: 20'
[INFO] [talker]: Publishing: 'Hello World: 21'
[INFO] [talker]: Publishing: 'Hello World: 22'
[INFO] [talker]: Publishing: 'Hello World: 23'
[INFO] [talker]: Publishing: 'Hello World: 24'
[INFO] [talker]: Publishing: 'Hello World: 25'
[INFO] [talker]: Publishing: 'Hello World: 26'
[INFO] [talker]: Publishing: 'Hello World: 27'
[INFO] [talker]: Publishing: 'Hello World: 28'
[INFO] [talker]: Publishing: 'Hello World: 29'
[INFO] [talker]: Publishing: 'Hello World: 30'
```

```
C:\Users\newme_000>ros2 run demo_nodes_py listener
[INFO] [listener]: I heard: [Hello World: 1]
[INFO] [listener]: I heard: [Hello World: 2]
[INFO] [listener]: I heard: [Hello World: 3]
[INFO] [listener]: I heard: [Hello World: 4]
[INFO] [listener]: I heard: [Hello World: 5]
[INFO] [listener]: I heard: [Hello World: 6]
[INFO] [listener]: I heard: [Hello World: 7]
[INFO] [listener]: I heard: [Hello World: 8]
```

你可以看到talker程序发布信息,listener程序说明自己接收到了对应的信息。 以前节点的通信通过master去实现，现在可以看到已经不需要master了。而且你可以尝试在不同的电脑上执行这个程序。不同电脑上的节点间不需要任何配置就可以通信了。

常见问题

- 如果你由于缺少dll而无法启动例子程序，那么请确认上面的软件包依赖比如OpenCV已经添加到你的PATH环境变量中。
- 如果你忘记了执行local_setup.bat，那么例子程序可能会立即崩溃停止运行。

ROS2的基本概念

ROS是一个用于在不同进程间匿名的发布、订阅、传递信息的中间件。 ROS2系统的核心部分是ROS网络(ROS Graph)。 ROS网络是指在ROS系统中不同的节点间相互通信的连接关系。 *ROS Graph*这里翻译成了*ROS*网络，因为我觉得*Graph*更加抽象，而网络的概念更容易帮助理解其内涵。

ROS网络(ROS Graph)概念简介

- 节点(Nodes)： 一个节点是一个利用ROS系统和其他节点通信的实体
- 消息(Messages)： ROS中在订阅和发布主题时所用到的数据结构
- 主题(Topics): 节点可以发布信息到一个主题，同样也可订阅主题来接收消息
- 发现(Discovery): 一个自动运行的进程，通过这个进程不同的节点相互发现，建立连接

节点(Nodes)

一个节点就是一个在ROS网络中的参与者。 ROS节点通过ROS客户端程序库(ROS client library)来和其他节点进行通信。 节点可以发布或者订阅主题 节点也可以提供ROS服务(Service)。 节点有很多可以配置的相关参数。 节点间的连接时通过一个分布式发现进程来建立的（即上面所说的发现）。 不同的节点可以在同一个进程里面，也可以在不同的进程里面，甚至可以在不同的机器上。

客户端程序库

ROS客户端程序库可以让不同的语言编写的节点进行通信。 在不同的编程语言中都有对应的ROS客户端程序库(RCL)，这个程序库实现了ROS的基本API。 这样就确保了不同的编程语言的客户端更加容易编写，也保证了其行为更加一致。

下面的客户端程序库是由ROS2团队维护的

- rclcpp = C++ 客户端程序库
- rclpy = Python 客户端程序库

另外其他客户端程序也已经有ROS社区开发出来。 可以看[[ROS 客户端程序库]]来了解详细信息

发现

节点之间的互相发现是通过ROS2底层的中间件实现的。 过程总结如下

1. 当一个节点启动后， 它会向其他拥有相同ROS域名(ROS domain， 可以通过设置ROS_DOMAIN_ID环境变量来设置)的节点进行广播，说明它已经上线。 其他节点在收到广播后返回自己的相关信息，这样节点间的连接就可以建立了，之后就可以通信了。
2. 节点会定时广播它的信息，这样即使它已经错过了最初的发现过程，它也可以和新上线的节点进行连接。
3. 节点在下线前它也会广播其他节点自己要下线了。

节点只会和具有相兼容的[服务质量]设置的节点进行通信。

例子：发布和接收

在一个终端，启动一个节点(用C++编写)，这个节点会向一个主题发布消息

```
ros2 run demo_nodes_cpp talker
```

在另一个终端，同样启动一个节点(用Python编写)，这个节点会订阅和上个节点相同的主题来接收消息。

```
ros2 run demo_nodes_py listener
```

你会看到节点自动发现了对方，然后开始互相通信。 你也可以在不同的电脑上启动节点，你也会发现，节点自动建立了它们的连接，然后开始通信。

ROS2和不同的DDS程序

ROS2是建立在DDS程序的基础上的。DDS程序被用来发现节点，序列化和传递信息。 这篇文章详细介绍了DDS程序的开发动机。 总而言之，DDS程序提供了ROS系统所需的一些功能，比如分布式发现节点(并不是像ROS1那样中心化)，控制传输中的不同的"通信质量（Quality of Service）"选项。

DDS是一个被很多公司实现的工业标准。比如RTI的实现Connext和eProsima的实现Fast RTPS。 ROS2 支持多种实现方式。因为没有必要“一个尺码的鞋给所有人穿”。用户有选择的自由。 在选择DDS实现的时候你要考虑很多方面：比如法律上你要考虑他们的协议，技术上要考虑是否支持跨平台。 不同的公司也许会为了适应不同的场景提出不止一种的DDS实现方式。 比如RTI为了不同的目标就有很多种他们的Connext的变种。从小到微处理器到需要特殊资质的应用程序（我们支持标准的桌面版）。

为了能够在ROS2中使用一个DDS实现，需要一个ROS中间件(RMW软件包), 这个包需要利用DDS程序提供的API和工具实现ROS中间件的接口。 为了在ROS2中使用一个DDS实现，有大量的工作需要做。但是为了防止ROS2的代码过于绑定某种DDS程序必须支持至少几种DDS程序。因为用户可能会根据他们的项目需求选择不同的DDS程序。

支持的RMW实现

名称	协议	RMW 实现	状态
eProsima *Fast RTPS*	Apache 2	`rmw_fastrtps_cpp`	完全支持. 默认的RMW. 已经打包在发布的文件中.
RTI *Connext*	commercial, research	`rmw_connext_cpp`	完全支持. 需要从源码编译支持.
RTI *Connext* (dynamic implementation)	commercial, research	`rmw_connext_dynamic_cpp`	停止支持. alpha 8.* 之前版本完全支持
PrismTech *Opensplice*	LGPL (only v6.4), commercial	`rmw_opensplice_cpp`	停止支持. alpha 8.* 之前版本完全支持
OSRF *FreeRTPS*	Apache 2	--	部分支持. 开发暂停.

*暂停支持意味着从 *ROS2 alpha 8* 版本以后新的添加进*ROS2*的功能还没有添加到这些中间件的实现中来。 这些中间件的实现也许以后会会有也许以后也不会有。

对于如何同时使用多个RMW实现的方法，可以看[[使用多个RMW实现]]页面

简介

客户端程序库是用户在写自己的ROS程序时使用到的ROS API程序。 他们就是用户来访问ROS的基本概念比如节点，主题，服务等等时所使用的程序。 客户端程序库有很多不同语言的实现，这样用户就可以根据他们自己的使用场景灵活的选择最合适的语言。 例如你也许想用Python来写图形程序，因为写起来很方便。但是对于对性能要求比较高的地方，这些程序最好还是用C++来写。

使用不同语言编写的程序之间可以自由的共享信息。因为所有的客户端程序库都提供了代码生成程序，这些程序为用户提供了能够处理ROS的接口文件的能力。

客户端程序除了提供不同语言特定的通信工具之外，还为用户提供了一些使得ROS更加ROS化的核心功能。 比如，这些就是可以通过客户端程序库操作的核心功能

- 名称和命名空间
- 时间（实际的或者模拟的）
- 参数
- 终端输出
- 线程模型
- 跨进程通信

支持的客户端程序库

C++ 客户端程序库(rclcpp)和Python客户端程序库(rclpy)都提供了RCL的常用功能。

C++和Python的客户端程序库由 ROS 2 团队维护，ROS2的社区成员同时也支持了以下额外的客户端程序库：

- JVM 和 Android
- Objective C 和 iOS
- C#
- Swift
- Node.js

通用的功能：RCL

大部分客户端程序库提供的功能并不只是在特定的语言里才有。比如参数的效果，命名空间的逻辑，在理想的情况下应该在所有的语言下保持一致。 正是因为这一点，与其为每种语言从零开始实现一遍，倒不如使用一个通用的核心ROS客户端程序库接口。这个接口实现了程序逻辑和ROS中语言无关的功能。这样就使得ROS客户端程序库更小也更容易开发。 由于这个原因，RCL通用功能暴露出了C的程序接口。因为C是其他程序语言最容易进行包装的语言。

使用通用的核心库的优点不止可以使得客户端程序更加小型化，同时也使得不同语言的客户端程序库能够保持高度的一致性。 如果通用核心库内的任何功能发生了变化，比如说命名空间，那么所有的客户端程序库的这个功能都会跟着变化。 再者使用通用的核心程序库也意味着当修复bug的时候维护多个客户端程序会更加方便。

RCL 的 API文档可以看这里

和语言相关的功能

对于和语言相关的功能，并没有在RCL中实现，而是在各个语言的客户端程序库中去实现。 例如，在spin中使用到的线程模型，完全由各语言客户端程序自己实现。

例子

对于一个使用rclpy的发布者和一个使用rclcpp的订阅者之间的消息传递的例子，我们建议你去看这个视频this ROSCon talk的17:25处。这有一些相关的幻灯片

与ROS1相比

在ROS1中所有的客户端程序都是从零开始编写的。 这样可以使Python的客户端程序库完全由Python编写。这样就有不用编译源代码的好处。 但是命名规则和其他一些特性并不能和其他客户端程序库保持一致。bug的修复需要在不同的地方重复很多遍。而且有很多功能只有在特定语言的客户端程序库中才实现。

总结

通过把ROS通用客户端核心程序库抽出来，不同的语言的客户端程序变得更加容易开发也更能保证功能的一致性。

ROS 接口

1. 背景

ROS程序一般通过一种或两种接口进行通信：消息和服务 ROS使用了一种简化的描述语言来描述这些接口。这种描述语言使得ROS的工具更加容易的自动生成对应语言的源代码。

在这篇文章中，我们将介绍支持的类型和如何创建你的 msg/srv文件

2. 消息描述说明

消息的描述文件是在ROS软件包msg文件夹内的.msg文件。 .msg文件宝行两个部分：变量域（Fields）和常量(constants)

这里将*Field*翻译成变量域以和*constants*做作对比区分。如果直接翻译成域，会让人不明所以。

2.1 域

每一个域包含两个部分, 类型和名称。中间用空格隔开，例如

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

又如

```
int32 my_int
string my_string
```

2.1.1 变量域类型

变量域的类型可以是一下几种

- 内部定义类型
- 有用户自己定义的类型，比如 "geometry_msgs/PoseStamped"

内部定义的类型现在支持一下几种： | 类型名称 | C++ | Python | DDS type | ---------- | ---------- | ----- | ---- | | bool | bool | builtins.bool | boolean | | byte | uint8_t | builtins.bytes | *octet* | | *char* | *char* | *builtins.str* | char | | float32 | float | builtins.float | *float* | | *float64* | *double* | *builtins.float* | double | | int8 | int8_t | builtins.int | *octet* | | *uint8* | *uint8_t* | *builtins.int* | octet | | int16 | int16_t | builtins.int | *short* | | *uint16* | *uint16_t* | *builtins.int* | unsigned short | | int32 | int32_t | builtins.int | *long* | | *uint32* | *uint32_t* | *builtins.int* | unsigned long | | int64 | int64_t | builtins.int | *long long* | | *uint64* | *uint64_t* | *builtins.int* | unsigned long long | | string | std::string | builtins.str | string |

每种内部定义类型都可以用来定义数组 | 类型名称 | C++ | Python | DDS type | ---------- | ---------- | ----- | ---- | | static array | std::array | builtins.list | *T[N]* | | *unbounded dynamic array* | *std::vector* | *builtins.list* | *sequence* | | *bounded dynamic array* | *custom_class* | *builtins.list* | sequence | | bounded string | std::string | builtins.str* | string |

所有比ROS变量定义中范围更广，更加宽松的变量，都会被软件限制在ROS所定义的范围中。

使用数组和限制类型的消息定义的例子

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

string string_of_unbounded_size
string<=10 up_to_ten_characters_string

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

2.1.2 变量域名称

变量域名称必须以小写字母开始，同时以下划线作为单词的分割符。不能以下划线结束，也不允许有两个连续的下划线。

2.1.3 变量域默认值

默认值可以设置成变量域类型所允许的任意值。 当前默认值还不能支持字符串数组和复杂类型。（也就是没有出现在内部定义类型里面的，同样也适用于所有的嵌套消息）

定义默认值可以通过在变量域定义中添加第三个元素来实现。也就是

```
变量域名称 变量域类型 变量域默认值
```

比如

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

特别说明:

- 字符串类型默认值必须用单引号或者双引号括起来
- 当前的字符串类型是没有被转义的

2.2 常量

常量的定义就好像有默认值的变量域定义。除了常量的值是永远不能由程序改变的。常量通过等号进行赋值。也就是

```
常量类型 常量名称=常量值
```

例如

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

特别说明：常量名必须是大写

3. 服务定义说明

服务描述由位于ROS包下的srv文件夹内的.srv文件定义。

一个服务描述文件包含了一个请求和一个回应的消息类型。之间用---分割。任意的两个消息类型连接起来，并在中间用---分割都是一个合法的服务描述。

下面是一个非常简单的服务的例子。这个服务接收一个字符串然后返回一个字符串：

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

当然我们也可以更加复杂一点（如果你想引用来自同一个软件包内的消息类型，那么你一定不要包含这个软件包的名字）：

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

你不能在一个服务中嵌入另外一个服务。

简介

ROS2 提供了提供了非常丰富的服务质量控制规则。利用这些规则你可以优化微调节点间的通信。 合适的服务质量设置可以使得ROS2像TCP协议一样可靠谱或者像UDP协议一样高效。亦或者在之间的无线可能的状态之中。 在ROS1中，我们只能支持TCP协议。ROS2则受益于底层的DDS传输可以灵活设置。对于在一个容易丢失数据的无线网环境下，可以使用一个高效的服务质量规则。对于在实时计算情况下可以使用高可靠性的服务质量规则。

一组服务质量规则组合在一起就构成了一个服务质量配置文件。 由于在不同的场景中设置合适的服务质量配置文件并不是一个简单的事情。ROS2预先提供了一些常用情景下的配置文件。（比如传感器数据） 同时用户也可以更改服务质量的具体配置。

服务质量文件可以专门用来配置发布者，订阅者，服务提供者和客户端。 一个服务质量文件可以独立应用于不同的上面所说的各种实体。但是如果它们之间使用了不同的服务质量文件那么有可能它们之间会无法建立连接。

服务质量规则

一个基本服务质量规则配置文件包含下面的几个规则：

- 历史 (History)
  - Keep last: 只存储最多N个样本，通过队列深度设置
  - Keep all: 存储所有样本，由底层中间件的资源大小限制
- 深度 (Depth)
  - 队列的大小：只有在和Kepp last一起时才会起作用
- 可靠性 (Reliability)
  - 最高效率： 尝试发送数据，但是在网络不好的情况下有可能丢包
  - 高可靠性: 保证数据发送成功，但是可能会重试发送多次
- 耐久力 (Durability)
  - 本地缓存(Transient local): 发送者会为还未加入的节点保存未接收的数据
  - 自动挥发(Volatile): 不会特意保存数据

对于每个规则，都还有一个系统默认选项。这些值使用了来自底层中间件的参数。默认值可以通过DDS工具进行设置(比如 xml配置文件) DDS本身有着更多的规则可以配置。 由于和ROS1中的功能很相似，所以以上的规则被暴露出来。在未来很有可能会在ROS2中暴露更多的配置规则。

和ROS1的对比

历史和深度规则组合起来和ROS1中的队列大小是一样的。

ROS2中的可靠性规则和ROS1中也有对应。ROS1的UDPROS(只在roscpp里面)对应于高效模式。TCPROS(ROS1的默认模式)对应于高可靠性模式。 特别注意即使是ROS2里面的高可靠性模式也是用UDP去实现的。在合适的场景下还可以用来进行广播。

持久性规则和depth=1的规则组合起来和ROS1中的"latching" subscribers很类似。

QoS 配置文件

配置文件使得开发者可以专注于开发自己的应用程序，不用关心每个服务质量设置。 服务质量配置文件包含了一系列的规则，保证程序能够在特定的使用场景下工作。

目前已经定义的服务质量配置有

- 默认的发布者和订阅者的服务质量配置

  为了能够从ROS1过渡到ROS2，保证网络的相似性是很必要的。 在默认条件下，发布者和订阅者在ROS2中都是配置为可靠模式，自动挥发，和存储最后。

- 服务

  和发布者和订阅者一样服务也是可靠的 对于服务来说使用挥发耐久力是必须的，因为否则当服务提供者可能会受到已经过时的请求。 尽管这样可以保证客户端不会受到多个应答。服务端却无法保证不受到已经过时的请求。

- 传感器数据

  对于传感器数据，在大多数情况下在及时的受到数据更加重要。而保证收到所有的数据并没有这么重要。 也就是开发者希望能尽快的收到最新的数据，尽管可能会有数据丢失。 由于这个原因传感器数据被设置成了高效模式和一个较低队列深度。

- 参数

  ROS2中的参数设置是基于服务的。所以他们有相似的配置。 不同的是参数会有一个更大的队列来保证请求不会丢失。比如当参数客户端无法连接到参数服务器的时候。

- 系统默认

  使用系统默认的规则

点击这里可以查看上面的配置使用到的特殊规则。 根据社区的反馈上面的一些规则还需要进一步的调整。

尽管ROS2提供了一些服务质量配置，但是直接配置DDS所提供的规则可以更大程度的发挥出DDS程序的性能。

服务质量兼容性

注意: 这一部分只说明了发布者和订阅者的信息，但是这些内容对于服务提供者和服务客户端同样也是适用的。

服务配置文件可以独立配置发布者和订阅者。 只有在发布者和订阅者的服务质量规则是相兼容的时候，它们才能相互建立连接。 服务质量配置文件的兼容性判定是基于“请求对比提供者”模型。只有在请求方，也就是订阅者的规则没有发布者更加严格的条件下才能建立连接。 两者中的更为严格的那个规则会被用于它们之间的这个连接。

在ROS2中暴露的服务质量规则能够影响到连接的是可靠性和耐久性。 下面的表格表示了不同的配置规则和结果：、

服务质量耐久度兼容性

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

服务质量可靠性兼容性

Publisher	Subscriber	Connection	Result
Best effort	Best effort	Yes	Best effort

Best effort	Reliable	No	-
Reliable	Best effort	Yes	Best effort
Reliable	Reliable	Yes	Reliable

为了保证能够建立一个连接。所有可能影响到兼容性的规则都要注意保证兼容。 也就是即使一个发布者和订阅者有相互兼容的服务质量可靠性规则，但是如果它们的服务质量耐久性规则不兼容，也是无法建立连接的。反之亦然。

简介

这篇文章会简要介绍如何快速配置和使用一个ament工作空间。 这是一个比较实际的操作教程，并不是为了替换核心文档而写。

背景

Ament是catkin编译工具的优化迭代版本。 关于Ament的设计上的信息可以看这个文档

ament的源代码可以在这里找到

开发环境

确保你已经按照之前从源代码编译文档的要求配置好你的开发环境。

基础知识

一个Ament工作空间，是一个有着特定目录结构的文件夹。 通常会有一个src子文件夹。 在这个子文件夹下是各个软件包的源代码。 通常这个文件夹初始情况下是空的。

Ament脱离源代码进行编译。 默认情况下它会在src文件夹旁创建一个build和一个install文件夹。 build文件夹会用来存放编译过程中产生的中间文件。 对于每个软件包都会创建一个对应的文件夹，然后cmake在对应的文件夹下运行。 install文件夹是软件包的安装位置

注意:和catkin相比这里没有devel文件夹

创建目录结构

下面在 ~/ros2_ws 下创建一个基本的工作空间目录结构

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws
```

下面就是你所预想的目录结构

```
.
└── src

1 目录, 0 文件
```

添加一些源代码

在开始之前我们需要先配置一些依赖

```
wget https://raw.githubusercontent.com/ros2/ros2/master/ros2.repos
vcs import ~/ros2_ws/src < ros2.repos
```

下面是在添加源代码之后~/ros2_ws的目录结构（注意具体提的目录结构，文件夹和文件的数据可能会变化）。

```
.
├── ros2.repos
└── src
    ├── ament
    │   ├── ament_cmake
    │   ├── ament_index
    │   ...
    │   ├── osrf_pycommon
    │   └── uncrustify
    ├── eProsima
    │   ├── Fast-CDR
    │   └── Fast-RTPS
    ├── ros
    │   ├── class_loader
    │   └── console_bridge
    └── ros2
        ├── ament_cmake_ros
        ├── common_interfaces
        ├── demos
        ...
        ├── urdfdom
        ├── urdfdom_headers
        └── vision_opencv

51 directories, 1 file
```

开始编译

由于我们把ament源代码放在了工作空间中，我们需要执行ament.py的全路径。 注意：在未来ament将会默认安装在系统中，或者在一个底层的工作空间中。这样就不需要再执行上面的步骤了。

在安装时每个软件包都支持 `--symlink-install` 安装选项。这样可以通过创建符号链接进行安装。当源码发生变化时，安装位置的文件也会跟着发生变化。（例如python或者其他不需要编译的文件）这样可以实现更快的软件迭代开发。

```
src/ament/ament_tools/scripts/ament.py build --build-tests --symlink-install
```

运行测试程序

为了能够运行测试程序，你需要在编译的时候增加 `--build-tests` 选项。之后运行下面的指令

```
src/ament/ament_tools/scripts/ament.py test
```

如果你在之前编译的时候没有添加 `--build-tests` 选项,在编译测试的时候，你可以直接跳过编译和安装的步骤来加速编译测试程序的过程

```
src/ament/ament_tools/scripts/ament.py test --skip-build --skip-install
```

source 你的环境

当ament编译完成之后，生成的文件会在install文件夹里面。 为了能够使用你编译的文件，你需要把 `install/bin` 添加到你的系统路径里面去。 Ament会在install文件夹中自动生成bash文件，用来帮助你配置环境变量。 这些文件会向你的系统路径和库路径添加必要的元素。

```
. install/local_setup.bash
```

注意:这里和catkin有一些不同 这里的 `local_setup.*` 文件和 `setup.*` 文件有一些不同。 `local_setup.*` 文件只会应用当前工作空间的设置。 当你在使用多个工作空间时，你仍然需要source `setup.*` 文件。这样所有父工作空间的设置也能添加进来。

运行一个例子

当你source 环境之后你可以执行下面的指令。这是由ament编译生成的。

```
ros2 run demo_nodes_cpp listener &
ros2 run demo_nodes_cpp talker
```

你会看到数字在不断地增加。

下面我们先停止这两个节点，然后来创建我们自己的工作空间

```
^-C
kill %1
```

开发你自己的软件包

Ament 使用和catkin一样的package.xml文件。（这个文件在REP 140中定义）

你可以直接在src文件夹内创建自己的软件包。但是如果你只是想写几个软件包，推荐你重新创建一个工作空间。

创建一个工作空间

首先创建一个新的文件夹 `~/ros2_overlay_ws`

```
mkdir -p ~/ros2_overlay_ws/src
cd ~/ros2_overlay_ws/src
```

然后开始之前我们要先下载ROS2例子。我们会这基础上进行修改。

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

开始编译，我们用debug模式编译，这样就可以获取到debug的相关文件。

```
cd ~/ros2_overlay_ws
ament build --cmake-args -DCMAKE_BUILD_TYPE=Debug
```

现在我们看看talker程序到底指向那个工作空间的文件。因为系统自身有一个talker程序，之前的工作空间也有一个talker程序，现在的工作空间同样也有一个talker程序。

如果你source了 `~/ros2_overlay_ws/install/local_setup.bash` ，那么talker程序会指向当前工作空间的文件。

如果你打开了一个新的终端程序，然后想要使用刚才创建的工作空间，那么你只需要 source `~/ros2_overlay_ws/setup.bash` ，这个程序会自动的设置好所有的父工作空间的环境变量。

创建你自己的软件包

你可以开始创建自己的软甲包了。 `catkin_create_package` 的等效软件会被移植到ament里面，但是目前还没有完成。

Ament支持多种编译方式。 推荐的方式是 `ament_cmake` 和 `ament_python` 。 同样还支持纯粹的cmake软件包 将来还会添加更多的编译方式。

一个像 `demo_nodes_cpp` 的软件包使用 `ament_cmake` 编译方式，同时使用cmake作为编译工具。

注意

- 如果你不想编译某个软件包，你可以在那个包的文件夹下放一个叫做 `AMENT_IGNORE` 的空文件。这样这个软件包就不回被处理。 "Catch all" 选项和其他的 --cmake-args参数一样应该被放在其他参数之后，或者使用--开头

```
ament build . --force-cmake-configure --cmake-args -DCMAKE_BUILD_TYPE=Debug -- --ament-cmake-args -DCMAKE_BUILD
_TYPE=Release
```

- 如果你只是想执行某个软件包的测试程序:

```
ament test --only-packages YOUR_PKG_NAME --ctest-args -R YOUR_TEST_IN_PKG
```

简介

ROS 2 包含了一些列的命令行工具去方便测试开发ROS 2的应用程序。

使用方法

这些工具的主要入口是 `ros2` 指令。这个指令包含了大量的用来测试开发的子命令。这些命令可以方便的操作节点，主题，服务和其他。

想要看到所有的子命令可以执行

```
ros2 --help
```

```
ros2 run demo_nodes_cpp talker -t chatter2
```

例子

为了能够使用命令行工具实现经典的发送者订阅者例子，我们要使用topic子命令。它可以用来向某个主题发送或接受信息。

在一个终端发布消息

```
$ ros2 topic pub /chatter std_msgs/String "data: Hello world"
publisher: beginning loop
publishing std_msgs.msg.String(data='Hello world')

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

在一个终端显示接收到的消息

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

data: Hello world
```

实现过程

ROS2 使用了分布式发现进程去自动的连接不同节点。 这个过程故意不适用一个中心化的发现机制（就好像ROS1里面的主节点一样）。发现ROS网络中的其他节点可能会需要一些时间。 正是因为如此，在后台有一个长期运行的程序，它保存了ROS网络中的信息。这样就可以更加快速的处理请求和提供快速响应。例如节点的名称列表。

这个后台程序在你第一次运行命令行工具的时候就会开始执行。 你可以运行 `ros2 daemon --help` 来查看更多的和后台程序交互的选项。

代码实现

ros2命令的源代码在这里ros2

ros2工具实现了一个框架，可以通过插件去扩展功能。 例如，`sros2` 软件包提供了“security” 子命da令。当这个软件包安装之后ros2指令会自动的检测到这个程序并添加对应的子命令。

使用多个ROS2中间件的实现

这篇文章详细说明了默认的ROS2中间件实现和如何选择其他的中间件。

开始之间

你需要已经读过[[DDS和ROS中间件]]这篇文章。

多个RMW实现

当前的ROS二进制包发布版本已经支持了一种RMW实现（FastRTPS）由于其他支持的RMW无法随意的发布。

尽管ROS的二进制发布版本只支持一种RMW实现。一个从源码编译的工作空间，可以用来同时编译安装多个RMW实现。 当ROS2的核心代码编译的时候，如果有任何的RMW已经在电脑上安装配置完成，那么对应的RMW也会被同时编译进去。 例如如果Fast RTPS的RMW 软件包在工作空间中，当发现 已经安装过Fast RTPS时，这个软件包也会被同时编译。

在很多情况下你会发现使用不同的RMW实现的节点是能够通信的。然而并不是在所有情况下都成立。 将来会列出能够在不同RWM实现件通信的配置列表。

默认的RMW实现

如果一个ROS2的工作空间有多个RMW实现，那么如果里面有Fast RTPS，那么它就会被作为默认实现。 如果没有Fast RTPS，那就会按照字母顺序选择默认的RMW实现。 这个字母顺序是按照提供RMW实现的ROS包的名字来的。 比如如果同时安装有 `rmw_opensplice_cpp` 和 `rmw_connext_cpp` ROS软件包。 `rmw_connext_cpp` 会被选择为默认的RMW实现。 如果同时安装有 `rmw_fastrtps_cpp` ，那么它就会被作为默认实现。 下面会介绍在运行ROS2例子的时候如何选择RMW实现。

注意：对于ROS2 alpha版本直到 alpha 8版本 在选择默认版本中，只考虑字母顺序原则。Fast RTPS 相对于其他的实现并没有什么优先级。

指定RMW实现

为了能够同时使用多种实现，你必须要从源码安装ROS

C++

ROS 2 C++ 示例程序会编译生成一组可执行文件比如'talker' 默认情况下这些程序会使用默认的RMW实现。 想要使用不同的RMW实现，可译设置环境变量 `RMW_IMPLEMENTATION` 。比如

- `RMW_IMPLEMENTATION=rmw_connext_cpp ros2 run demo_nodes_cpp talker` .这样talker会使用Connext作为默认的RMW实现。

beta 1 和之前版本

beta1和之前版本并不支持 `RMW_IMPLEMENTATION` 环境变量。 ROS 2 C++ 例子程序会针对每一个安装好的RMW实现编译一个版本的可执行文件。比如会编译生成下面的可执行文件。

- 一个叫做 `talker__rmw_fastrtps_cpp` 的可执行文件，此文件使用Fast RTPS作为RMW实现。
- 一个叫做 `talker__rmw_connext_cpp` 的可执行文件，此文件使用Connext作为RMW实现。
- 一个叫做 `talker` 的可执行文件，此文件使用默认的RMW实现。

Python

ROS2的Python例子默认情况下使用默认的RMW实现。 例如假设你已经安装配置完成你的ROS2工作空间。下面的命令会执行talker发布者，其使用默认的RMW实现。

```
ros2 run demo_nodes_py talker
```

和C++一样，你也可以通过设置 `RMW_IMPLEMENTATION` 环境变量来更改RMW实现。

```
export RMW_IMPLEMENTATION=rmw_connext_cpp
ros2 run demo_nodes_py talker
```

beta 1和之前版本

在beta1和之前版本，同样也不支持 `RMW_IMPLEMENTATION` 环境变量。 然而你可以通过设置 `RCLPY_IMPLEMENTATION` 环境变量来指定RMW实现。 例如想要指定 `rmw_fastrtps_cpp` 作为RMW实现，在Linux上你可以执行：

```
RCLPY_IMPLEMENTATION=rmw_fastrtps_cpp talker_py
```

在你的工作空间添加RMW实现

假设你在编译你的工作空间的时候只安装使用了Fast RTPS 作为中间件。 那么其他的RMW实现的ROS包，比如 `rmw_connext_cpp` ,就很有可能无法找到相关的DDS实现的安装。 即使你之后又安装了其他的DDS实现，比如说Connext，你必须在Connext RMW实现编译的时候重新触发再次检查Connext的安装位置。 你可以在你下次编译时设置 `--force-cmake-configure` 标记来触发。然后你就能看到新的编译结果已经能够支持新的加入的DDS实现了。

当你在使用 `--force-cmake-configure` 选项进行编译的时候，你很有可能会遇到问题说默认的RMW发生了改变。为了解决这个问题。你可以通过设置 `RMW_IMPLEMENTATION` 环境变量，把RMW设置成和原来的RMW一样。也可以直接删除报错这个错误的软件包的build文件夹，然后再添加上 `--start-with <软件包名称>` 参数后重新编译。

常见问题

确保使用特定的RMW实现

ROS2 ardent 和之后版本

如果 `RMW_IMPLEMENTATION` 环境变量设置为一个尚未支持或安装的RMW版本。你会看类似于下面的错误信息

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

如果你安装了多个RMW实现，但是你设置了一个并未安装的实现，那么你会看到类似下面的错误信息。

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

如果发生了以上的情况，请再次确认你时候已经安装了对应的RMW实现。

ROS 2 beta 2和之后版本

在ROS 2 beta 2/ beta 3, 设置一个无效的RMW实现并不会产生任何错误。 如果你想确认时候已经使用了否个版本RMW实现，你可以设置 `RCL_ASSERT_RMW_ID_MATCHES` 环境变量。这个环境变量会让节点只允许使用指定的RMW实现。

```
RCL_ASSERT_RMW_ID_MATCHES=rmw_connext_cpp RMW_IMPLEMENTATION=rmw_connext_cpp ros2 run demo_nodes_cpp talker
```

在同一个进程中使用多个节点

ROS 1 - Node 和 Nodelets

在ROS1中你可以写一个节点也可以写一个小节点(Nodelet)。 ROS 1 的节点会被编译成一个可执行文件。 ROS 1的小节点会被编译成一个动态链接库。当程序运行的时候会被动态的加载到容器进程里面。

ROS 2 - 统一API

在ROS2里面，推荐编写小节点——我们称之为组件 `Component` 。 这样我们就更容易为已经存在的代码添加一些通用的概念，比如生命周期. 使用不同的API所带来的麻烦完全被ROS2给避免了。节点进和小节点在ROS2中完全使用相同的API。

> 你也可以继续使用节点的风格的主函数，但是一般是并不推荐的

通过把进程的结构变成一个部署是的选项，用户可以自由的在下面的模式进行选择

- 在不同的进程中运行多个节点。这样可以使不同的进程独立开。一个崩溃其他可以正常运行。也更方便调试各个节点。
- 在同一个进程中运行多个节点。这样可以使得通信更加高效。

在未来的roslaunch版本中，会支持配置进程的结构。

编写一个组件

由于一个组件会被编译生成一个共享链接库。所以它并没有一个主函数入口。(see Talker source code) 组件继承自 `rclcpp::Node` 类 由于它并不由一个线程直接控制，所以不要在构造函数里面执行很长时间的任务，甚至是阻塞的任务。 你可以用定时器来实现周期性的提示。 另外它可以创建发布者，订阅者，服务提供者和服务客户端。

为了使这样一个类能够成为一个组件，很重要的一点是使用 `class_loader` 软件包注册自己（可以查看最后一行的源代码）。 这样组件就能够在库文件被加载进进程时被发现。

使用组件

composition软件包包含了几种不同的使用组件的方式。 最常见的是下面几种

1. 你启动了一个通用的容器进程(1) 然后调用由此容器提供的ROS服务load_node。 这个ROS服务接着会根据传入的软件包名称和组件名称载入对应的组件开始执行。 除了使用程序来调用ROS服务外，你还可以使用命令行工具去传入参数触发ROS服务。

2. 你可以创建一个自定义的可执行文件。这个文件中包含多个节点。 这种方法要求每个组件都有一个头文件。（第一个方法并不需要这样）

运行例子程序

来自composition软件包的程序可以通过下面的指令执行

使用ROS服务的运行时组件（第一种方式）。此组件有发布者和订阅者。

在第一终端执行：

ros2 run composition api_composition

在第二个终端执行：

ros2 run composition api_composition_cli composition composition::Talker

现在第一个终端会显示已经翟茹一个组件，同时会不停的显示已经发送了一个消息。

在刚才第二个终端中执行

ros2 run composition api_composition_cli composition composition::Listener

现在第一个终端应该会不停的显示收到的消息。

> 这个例子在代码中使用了固定的主题名称，所以你不可以运行两遍`api_composition`总的来说你也可以分别运行两个容器进程，然后分别载入发布者和订阅者程序。你会发现他们也是可以相互通信的。

使用ROS服务的运行时组件（第一种方式）。此组建有服务提供者和服务客户端

这个例子和上面的例子非常相似。

首先在第一个终端中输入

ros2 run composition api_composition

在第二个终端中输入

ros2 run composition api_composition_cli composition composition::Server ros2 run composition api_composition_cli composition composition::Client

在这个例子中客户端发送请求至服务端，服务端处理请求然后返回响应。客户端将收到的响应打印出来。

使用ROS服务的编译时组件（第二种方式）

这个例子展示了相同的共享链接库可以重用然后编译出一个使用多个组件的可执行程序。这个程序包含了以上的四个组件：发布者，订阅者，服务提供者，服务客户端

在终端执行

ros2 run composition manual_composition

终端会重复显示来自两对程序的信息。

使用dlopen的运行时组件

这个例子展示了第一种的替代方式。启动一个通用的容器进程，然后不通过ROS接口直接传递给它要载入的库。 这个进程会载入每一个库，然后创建对应的"rclcpp::Node"类示例。

Linux 在终端中执行

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

OSX 在终端中执行

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

Windows 在命令行中执行

ros2 pkg prefix composition

来获取composition软件包的安装位置，然后执行

ros2 run composition dlopen_composition \bin\talker_component.dll \bin\listener_component.dll

现在在终端会重复输出每个发送和接收的消息。

自定义接口 （消息、服务）

未完成

尽管我们鼓励重用已有的标准消息和服务定义。然而在很多情况下你还是要自己定义消息或者服务。 自定义消息和服务第一步时创建 `.msg` 或 `.srv` 文件。定义方式可以参照[[ROS接口]]

为了方便起见， `.msg` 文件放置于软件包文件夹下的msg文件夹内。 `.srv` 文件放置于srv文件夹内。

在写完你的 `.msg` 或 `.srv` 文件后，你需要在你的CmakeLists.txt文件内添加一些代码。使得代码生成程序能够处理你的定义文件。先要了解更加详细的教程可以参照pendulum_msgs package，作为一个例子。你可以在这个包的CMakeLists.txt文件中看到相关的CMake的调用。

ROS2 接口中的新功能

尚未完成

ROS2的接口定义语言和ROS1的非常接近。 基本上所有的ROS1的 `.msg` 和 `.srv` 文件都可以在ROS2里面重用。 ROS2在1的基础上有新增加了一些功能即

- **bounded arrays**： ROS1只支持没有限制的数组（比如 `int32[] foo` ）和固定大小的数据（比如 `int32[5] bar` ），ROS2 支持有限制的数据(`int32[<5] bat`) 在有些应用场景下是可以给数组的大小定一个上界的，这样可以节省大量的空间。
- **bounded strings**: 同样ROS1也只支持没有限制的字符串。ROS2可以支持有限制的字符串。（string<=5 bar）
- **default values**: ROS1中支持常量(`int32 X=123`)但没有默认值, ROS 2 支持默认值(`int32 X 123`) 当创建一个消息或服务时，如果没有额外设置，则会采用默认值。如果设值则默认值会被覆盖。

注意： 在 beta 1版本中默认值只支持数字类型，数字数组类型，和字符串类型（没有经过转义和编码）。

这篇文章会教你如何给发布者和订阅者写一个自定义的内存分配器。这样当你的ROS节点程序运行时就会通过你的内存分配器给你的程序分配内存，而不会使用默认的内存分配器。 这个例程的源代码可以在这里获取

背景

假如你想写实时运行的安全可靠的代码，那么你一定听说过使用 new 来实时分配内存时可能造成的种种危险。因为默认的在各种平台上的内存分配器都是不确定的。

默认情况下。很多标准的C++库数据结构在数据增加的时候都会自动分配内存。比如 `std::vector` .然而这些数据结构也接收一个"内存分配器"参数。如果你给其中的一种数据结构指定了内存分配器，那么它就会使用你的内存分配器而不用系统的内存分配器去分配内存。你的内存分配器可以是一个已经提前分配好的内存栈。这样就更适合于实时运行的程序。

在ROS2 C++客户端程序中(rclcpp),我们和C++标准库有着类似的原则。发布者订阅者和执行者接受一个内存分配器参数。这个分配器在运行时控制着整个程序。

编写一个内存分配器

为了写一个和ROS2接口兼容的内存分配器，你的分配器必须兼容C++标准库的内存分配器接口。

C++标准库提供了一个叫做 `allocator_traits` 的东西。C++标准库的内存分配器只需要实现很少的几个方法就可以按照标准的方式去分配和回收内存。 `allocator_traits` 是一个通用的结构，它提供了通过通用内存分配器编写一个内存分配器所需要的其他参数。

例如，下面的对于一个内存分配器的声明就满足 `allocator_traits` (当然，你还是需要实现这个声明中的各种函数)

```
template <class T>
struct custom_allocator {
  using value_type = T;
  custom_allocator() noexcept;
  template <class U> custom_allocator (const custom_allocator<U>&) noexcept;
  T* allocate (std::size_t n);
  void deallocate (T* p, std::size_t n);
};

template <class T, class U>
constexpr bool operator== (const custom_allocator<T>&, const custom_allocator<U>&) noexcept;

template <class T, class U>
constexpr bool operator!= (const custom_allocator<T>&, const custom_allocator<U>&) noexcept;
```

然后你可以通过下面的方式来访问由 `allocator_traits` 配置的内存分配器内部的各种函数和成员变量:
`std::allocator_traits<custom_allocator<T>>::construct(...)`

想要了解 `allocator_traits` 的全部功能可以看: http://en.cppreference.com/w/cpp/memory/allocator_traits

然而有些编译器只有部分的C++ 11支持。比如GCC 4.8, 这样就还需要写大量的样板代码来在标准数据结构（比如数据和字符串）中使用. 因为结构并没有在内部使用 `allocator_traits` 。因此如果你在使用一个只有部分C++11支持的编译器。你的内存分配器就会需要写成这样:

```
template<typename T>
struct pointer_traits {
  using reference = T &;
  using const_reference = const T &;
};

// Avoid declaring a reference to void with an empty specialization
template<>
struct pointer_traits<void> {
};

template<typename T = void>
struct MyAllocator : public pointer_traits<T> {
public:
  using value_type = T;
  using size_type = std::size_t;
  using pointer = T *;
  using const_pointer = const T *;
  using difference_type = typename std::pointer_traits<pointer>::difference_type;

  MyAllocator() noexcept;

  ~MyAllocator() noexcept;

  template<typename U>
  MyAllocator(const MyAllocator<U> &) noexcept;

  T * allocate(size_t size, const void * = 0);

  void deallocate(T * ptr, size_t size);

  template<typename U>
  struct rebind {
    typedef MyAllocator<U> other;
  };
};

template<typename T, typename U>
constexpr bool operator==(const MyAllocator<T> &,
  const MyAllocator<U> &) noexcept;

template<typename T, typename U>
constexpr bool operator!=(const MyAllocator<T> &,
  const MyAllocator<U> &) noexcept;
```

写一个主函数例子

当你写了一个内存分配器之后，你必须把它通过一个共享指针传递给你的发布者，订阅者和执行者。

```
  auto alloc = std::make_shared<MyAllocator<void>>();
  auto publisher = node->create_publisher<         :msg::UInt32>("allocator_example", 10, alloc);
  auto msg_mem_strat =
    std::make_shared<rclcpp::message_memory_strategy::MessageMemoryStrategy<         :msg::UInt32,
    MyAllocator<>>>(alloc);
  auto subscriber = node->create_subscription<         :msg::UInt32>(
    "allocator_example", 10, callback, nullptr, false, msg_mem_strat, alloc);

  std::shared_ptr<       :memory_strategy::MemoryStrategy> memory_strategy =
    std::make_shared<AllocatorMemoryStrategy<MyAllocator<>>>(alloc);
  rclcpp::executors::SingleThreadedExecutor executor(memory_strategy);
```

在你的代码执行的过程中所传递的消息也需要通过你的内存分配器来分配。

```
  auto alloc = std::make_shared<MyAllocator<void>>();
```

当你已经实例化一个节点，给这个节点添加一个执行者之后，就是spin的时候了

```
  uint32_t i = 0;
  while (rclcpp::ok()) {
    msg->data = i;
    i++;
    publisher->publish(msg);
    rclcpp::utilities::sleep_for(std::chrono::milliseconds(1));
    executor.spin_some();
  }
```

在进程内传递内存分配器

尽管我们已经在同一个进程中创建了一个发布者和订阅者，我们还是不能在进程内的发布者和订阅者间传递内存分配器。

进程内管理器是一个通常对用户隐藏的类。但是为了能够传递自定义的内存分配器，我们需要通过rcl context把它暴露出来。进程内管理器使用了几种标准的数据结构，所以如果没有自定义的内存管理器，它就会使用默认的new。

```
  auto context = rclcpp::contexts::default_context::get_global_default_context();
  auto ipm_state =
    std::make_shared<rclcpp::intra_process_manager::IntraProcessManagerState<MyAllocator<>>>();
  // Constructs the intra-process manager with a custom allocator.
  context->get_sub_context<       :intra_process_manager::IntraProcessManager>(ipm_state);
  auto node = rclcpp::Node::make_shared("allocator_example", true);
```

一定要确保在这样构造节点之后实例化发布者和订阅者。

测试和验证代码

如何确认你的自定义内存分配器正在被使用

最简单的方法就是数自定义内存分配器的 `allocate` 和 `deallocate` 函数调用的次数，然后把这个次数和 `new` 和 `delete` 进行对比。

在自定义内存分配器中添加计数是非常容易的

```
  T * allocate(size_t size, const void * = 0) {
    // ...
    num_allocs++;
    // ...
  }

  void deallocate(T * ptr, size_t size) {
    // ...
    num_deallocs++;
    // ...
  }
```

你可以覆盖全局的new和delete操作

```
void operator delete(void * ptr) noexcept {
  if (ptr != nullptr) {
    if (is_running) {
      global_runtime_deallocs++;
    }
    std::free(ptr);
    ptr = nullptr;
  }
}

void operator delete(void * ptr, size_t) noexcept {
  if (ptr != nullptr) {
```

```
    if (is_running) {
      global_runtime_deallocs++;
    }
    std::free(ptr);
    ptr = nullptr;
  }
}
```

这里我们增加的是一个全局的静态变量， `is_running` 是一个全局的静态布尔变量，当spin函数执行前会被设值。

例子程序 会输出各变量的值。通过输入下面的指令来执行例子程序

```
allocator_example
```

或者同时启用进程内工作流

```
allocator_example intra-process
```

你会在输出中看到下面的结果

```
Global new was called 15590 times during spin
Global delete was called 15590 times during spin
Allocator new was called 27284 times during spin
Allocator delete was called 27281 times during spin
```

我们获取到了大概2/3的在程序中allocations/deallocation 的调用。但是这剩下的1/3是从哪里来的?

实际上这来自在这个例子中使用的底层的DDS实现。

下面内容超出这篇文章的范围。但是你可以通过查看在ROS2连续集成测试中的代码。里面有代码可以追踪到某个函数调用到底来自于rmw实现还是DDS实现。

https://github.com/ros2/realtime_support/blob/master/tlsf_cpp/test/test_tlsf.cpp#L41

注意上面的测试程序并没有使用我们刚才创建的自定义内存分配器，它使用的是TLSF内存分配器。

TLSF内存分配器

ROS2 提供了TLSF（两层分离适应）内存分配器支持。它为了满足实时性要求而被设计出来:

https://github.com/ros2/realtime_support/tree/master/tlsf_cpp

注意TLSF内存分配器是dual-GPL/LGPL协议的。

下面是一个使用TLSF内存分配器的例子

https://github.com/ros2/realtime_support/blob/master/tlsf_cpp/example/allocator_example.cpp