ROS 2

Author: Hongwei Xue (SSRVodka)

Chapter 0. 概览

一个快速搭建机器人的软件库（核心是通信）和工具集。追求：稳定、实时、安全的通信能力。

ROS 的核心通信功能是由 DDS（数据分发服务）有线协议提供，它在 ROS 中作为中间件存在。ROS 本身支持多种遵循 DDS 协议的 vendors，详细参见 ROS DDS REPS；

分发版本：ROS2 Releases (Jazzy Doc)；

安装教程：ROS 2: Jazzy Jalizco；

具体的能力：

- 四大通信机制：
  - 话题（topic）：基于 pub/sub 机制的通信，允许节点间异步传输数据；
  - 服务（service）：同步通信方式，类似同步 gRPC；
  - 参数（parameter）：专用于机器人参数设置和读取；
  - 动作（action）：支持复杂行为的通信方式。例如支持服务端反馈（双工、复杂控制流），支持客户端取消等；
- 可视化工具和调试工具；
- 丰富数学建模工具（例如坐标变换）、运动学工具；
- 强大的开源社区和框架（例如 Gazebo 仿真环境、Navigation2 导航框架、Moveit2 机械臂运动规划框架）；

不足之处：

- ROS 是个框架，本身还需要依赖于操作系统；
- 实时性依赖于 OS 和硬件提供，通信速度受内存、网络等限制；

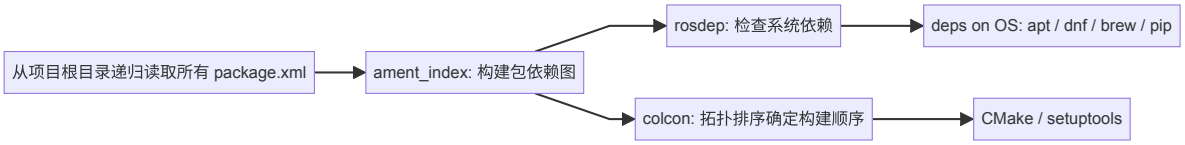
Chapter 1. 环境、功能包与构建系统

ROS 2 是相当庞大的框架，在设计构建系统时遵循几个理念：

1. **去中心化**：放弃 ROS 1 的 `catkin_make` 集中式构建，采用分布式构建（每个包独立构建）；
2. **工具链分离**：构建工具（`colcon`）与构建逻辑（`ament`）解耦；
3. **多语言支持**：ROS 客户端层原生支持 C++、Python，扩展支持 Rust、Java 等。因此构建系统会对不同语言分别处理；
4. **环境隔离**：严格隔离不同工作空间的构建/安装环境；
5. **安装空间导向**：构建目标直接生成可安装的 Artifacts；

组件	作用	替代 ROS 1 组件
`colcon`	元构建工具（核心入口）	`catkin_make`
`ament_cmake`	C++包的构建框架	catkin CMake 宏
`ament_python`	Python包的构建框架	`setup.py` + catkin

组件	作用	替代 ROS 1 组件
`CMake`	C++项目的实际构建系统	同 ROS 1
`setuptools`	Python包的安装工具	同 ROS 1



应用层客户端组件的 功能包（ROS package）一般的项目结构：

```
workspace/
├── src/                    # 源代码空间
│   └── my_pkg/             # ROS 包
│       ├── CMakeLists.txt # ament_cmake 构建规则
│       ├── package.xml    # 包元数据/依赖声明
│       ├── resource/      # 非代码资源文件
│       └── setup.py       # ament_python 构建入口
├── build/                  # 构建中间文件（每个包独立）
├── install/                # 安装空间（最终产出）
└── log/                    # 详细构建日志
```

注意 `workspace/{src,build,install,log}` 的结构是约定俗成的，是 ROS 中的包的工作空间（workspace）；

Chapter 2. 节点

ROS 中节点抽象就是完成通信功能的最小模块，目前就了解这么多即可。

2.1 `rclpy/rclcpp::Node`

看几段代码不接入机器人硬件的代码来熟悉 ROS 框架：

```python
import rclpy
from rclpy.node import Node

def main():
    rclpy.init()
    node_name = "demo_py_node"
    node = Node(node_name)
    _logger = node.get_logger()
    _logger.info(f"Hello from {node_name}")

    # 监听在 node 订阅的话题上，直至满足退出条件
    rclpy.spin(node)
    rclpy.shutdown()


if __name__ == '__main__':
```

```
    main()
```

```cpp
#include <rclcpp/rclcpp.hpp>

int main(int argc, char *argv[]) {

    rclcpp::init(argc, argv);

    std::string node_name = "demo_cpp_node";
    auto node = std::make_shared<rclcpp::Node>(node_name);
    auto logger = node->get_logger();
    RCLCPP_INFO(logger, "hello from %s", node_name.c_str());

    rclcpp::spin(node);
    rclcpp::shutdown();

    return 0;
}
```

对于 Python，由于安装 ROS 时已经将 Python 库引入系统环境的 Python 中了（如果你按教程执行了 `setup/local-setup` 的话），我们可以在系统环境 Python 中直接执行。但对于 C++ 而言，我们需要手动指定编译时需要引用的库，例如这个 CMakeLists 文件：

```cmake
cmake_minimum_required(VERSION 3.12)

project(demo_cpp_node)

add_executable(${PROJECT_NAME} node_demo.cpp)

# Manually find packages
find_package(rclcpp REQUIRED)
target_include_directories(${PROJECT_NAME} PUBLIC ${rclcpp_INCLUDE_DIRS})
target_link_libraries(${PROJECT_NAME} ${rclcpp_LIBRARIES})
```

2.2 补充：功能包的使用

目前的做法有些问题和需要改进的地方：

1. 如何将节点/项目抽象成一个功能包，方便分发、高内聚低耦合地开发？（无论对 Python 包还是 C++ 包）
2. 如何减少 C++ 的 ROS 包的配置，不需要记像 `rclcpp_INCLUDE_DIRS` 等生成变量？
3. 如何简化依赖管理的方式？
4. ……

现在我们需要回顾上一章的知识点，并了解如何创建功能包的项目框架，它能高效地解决上述问题。

我们可以通过 `ros2` 工具程序帮助我们创建一个功能包的项目骨架，方便开发：

```
ros2 pkg create <pkg_name> --build-type [ament_python|ament_cmake] --license
[Apache-2.0|GPL-3.0-only|MIT|...]
```

> **Tip**

> 在 Python 原生的 wheel 构建系统 `setup.py` 中，`setup` 函数是 `setuptools` 打包工具的核心。想要了解文件是如何配置的，建议自己写两个原生的 Python 包并打包练习一下。
>
> 这里介绍一个可能有用的前置知识点：你可以在 `setup` 的 `entry_points` 参数的 `console_scripts` 选项中添加命令行指令与模块的快捷方式。例如：
>
> ```
> setup(
>  name=package_name,
>  # ...
>  entry_points={
>      'console_scripts': [
>          'demo_pyn = demo_py_node.demo:main'
>      ],
>  },
> )
> ```
>
> 表示将此包安装后，会在 Python 环境的 bin 目录下生成一个快捷方式脚本，可以直接在命令行中使用 `demo_pyn` 即可代替 `python3 -m ...(一串包名)` 的用法。

现在如何遇到依赖，无论是 Python 还是 C++ 都可以直接在 `package.xml` 中添加：

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
    <name>demo_cpp_node</name>
    <version>0.0.0</version>
    <description>TODO: Package description</description>
    <maintainer email="root@todo.todo">root</maintainer>
    <license>Apache-2.0</license>

    <buildtool_depend>ament_cmake</buildtool_depend>

    <!-- C++ 项目如果用到 rclcpp 组件，添加这一行即可（Python 对应的是 rclpy） -->
    <depend>rclcpp</depend>

    <test_depend>ament_lint_auto</test_depend>
    <test_depend>ament_lint_common</test_depend>

    <export>
        <build_type>ament_cmake</build_type>
    </export>
</package>
```

上面的依赖还可以在命令行创建 package 时用 `--dependencies` 参数传入（空格分割）。

如果是 C++，还建议在生成的 CMakeLists 文件中加入类似内容，以符合规范：

```
# ...
find_package(ament_cmake REQUIRED)

# 在这里添加

# 其中会设置相应的 cmake 变量
```

```
find_package(rclcpp REQUIRED)
add_executable(${PROJECT_NAME} src/node_demo.cpp)
# 记得将依赖提醒 ament_cmake，可以帮我们完成繁琐的 target_link_* / target_include_*
操作
ament_target_dependencies(${PROJCT_NAME} rclcpp)
# 添加安装时的复制行为。注意 ament_cmake 的行为和 ament_python 不同，需要手动指定
install 的位置
# 这条指令会将当前项目的产物安装到 install/lib/${PROJECT_NAME} 下
install(TARGETS ${PROJECT_NAME} DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
# ...
```

这些依赖都会交给 `colcon` 并完成构建和安装（过程参见上一节）。执行构建：

```
colcon build
```

如果你的项目符合上一章提到的“工作空间”的规范，那么可以指定构建一个包：

```
colcon build --packages-select <package_name>
```

一般本地构建的包默认应该位于当前执行 `colcon build` 的目录下的 `install` 子目录中（另外可看到 `build, log` 等，解释也请参见上一节）。

按工作空间的约定，我们应在 workspace 的目录下执行 `colcon build` 防止生成的目录过多弄混。

您可能需要额外执行 `install/setup.xxsh` 来将您刚构建的包加入环境变量（例如 `PYTHONPATH`、`PATH` 等），以此使用它们。加入后您应该可以执行：

```
ros2 run <package_name> <py_module_name/cpp_target_name>
```

> 注：Python 基础。在 Python 解释器执行脚本时遇到 `import` 引用的包，会到 `PYTHONPATH` 环境变量中查找存在 `__init__.py` 的同名目录，并尝试解析内容。

Chapter 3. 话题通信

3.1 基本使用

话题由几个概念组成：发布者/订阅者（pub/sub pattern 的概念）、话题名（topic name）、话题类型（topic type，即话题返回的数据结果）；

ROS 提供了常用的调试工具：

- 查看节点中的通信信息：`ros2 node info <node_name>`；
- 查看节点话题详细信息（包括话题类型等）：`ros2 topic info <topic_name> -v`；
- 查看节点话题的话题类型：`ros2 interface show <topic_type>`；
- 查看话题发布更新的频率信息：`ros2 topic hz <topic_name>`；
- 作为 subscriber 持续获取话题内容：`ros2 topic echo <topic_name>`；
- 作为 publisher 持续发布话题内容：`ros2 topic pub <topic_name> <topic_type> <data_json_str>`；

如何在代码中写话题的发布者/订阅者呢？

1. 面向对象地创建一个继承于 Node 的类，作为节点抽象；
2. 发布者：Node 内存在 `create_publisher(topic_type, topic_name, qos_profile, callback_group, event_cb)` 方法，可以创建出 publisher 对象，在需要发布时直接 `.publish`；

   > **Note**
   >
   > 注 1：C++ 中 `create_publisher` 是一个模板，模板类型参数是话题类型。
   >
   > 注 2：一般发布的方法是创建一个 timer（`create_timer`）以及一个消息队列，然后在回调函数中发布。具体情况按需求来。
   >
   > 注 3：C++ 中 timer callback 参数是普通函数，如果希望使用成员函数，建议使用 `std::bind(func_ptr, params)` 绑定为一般函数指针；

3. 订阅者：Node 内存在 `create_subscriber(topic_type, topic_name, callback, qos_profile, ...)` 方法，callback 的参数是话题类型数据结构对应的共享指针（如果是 Python 则是对象）；

3.2 自定义话题类型

如何定义 `topic_type`？举个例子，我希望定义下面的数据类型：

```
builtin_interfaces/Time stamp
string hostname
float32 cpu_percent
float32 mem_percent
float32 mem_total
float32 mem_available
float64 net_sent
float64 net_recv
```

现在我们需要引入 ROS 中的一个工具 `rosidl_default_generators`，它的作用是将 IDL（interface definition language）转换成客户端可用的代码，和 gRPC 定义文件转 stub 的道理是类似的。

然后新建一个功能包，并在包内创建新目录 `msg`（注意，这个目录名是规定好的，不能指定其他名称），并将类型定义写入 IDL 文件中，名称建议使用约定的驼峰命名法，后缀名建议 `.msg`，表示是话题类型的通信方式使用到的数据类型。

> **Tip**
>
> 除了 `builtin_interfaces/Time`，ROS 还提供了哪些内置的基本数据类型可以让我们使用呢？
>
> 我们可以使用命令 `ros2 interface list` 查看。
>
> 注：`builtin_interfaces/Time` 类型的数据如何获得？`Node` 类型有方法 `get_clock()`，可以对内置 `Clock` 类型操作即可获得。

然后！比普通代码多一步的是，我们需要在 CMakeLists 中添加一个指令，让 `rosidl_default_generators` 生成对应客户端语言的 boilerplate code，举个例子：

```
# ...
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
```

```
find_package(builtin_interfaces REQUIRED)
find_package(rosidl_default_generators REQUIRED)
# 注意，在指定构建项目代码前添加指令来生成 IDL 对应代码
rosidl_generate_interfaces(<target_name> <idl_file_list> [DEPENDENCIES
<dep_list>])

add_executable(${PROJECT_NAME} src/sys_demo.cpp)
ament_target_dependencies(${PROJCT_NAME} rclcpp builtin_interfaces
rosidl_default_generators)
install(TARGETS ${PROJECT_NAME} DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
# ...
```

再然后，还需要向外提供信息：我这个包提供了话题类型的接口，这个改 `package.xml`（给外面的使用方了解）：

```
<package format="3">
    <!-- ... -->

    <!-- 添加这一行，建议在 buildtool_depend 前添加，方便查看 -->
    <member_of_group>rosidl_interface_packages</member_of_group>

    <buildtool_depend>ament_cmake</buildtool_depend>

    <test_depend>ament_lint_auto</test_depend>
    <test_depend>ament_lint_common</test_depend>

    <!-- ... -->
</package>
```

3.3 ROS 中使用 QT

注意，这里 QT 需要的 CMakeLists 的写法和一般 QT 项目的一致，不使用 `ament_target_dependencies`，直接用 `target_link_directories` 即可，因为 QT 库并不来自于 ROS 框架，而是 ROS 顺带安装的，因此我们只需和正常情况使用 QT 一样的导入就行。

Chapter 4. 服务和参数通信

在 ROS 中，DDS 中间件其实只支持 pub/sub pattern，也就是话题类型的通信，所谓其他三个通信方式，服务通信、参数通信、动作通信方式只不过是 ROS 框架对于话题通信的封装。

例如，服务通信本质上是由两个话题通信实现的通信机制，而参数通信本质上由多个服务通信和一个话题通信实现的，动作通信本质也由服务+话题通信实现。

现在，我们只需要清楚，服务的作用其实就类似 gRPC（有参数和返回值，不需要轮询/广播），只是服务的网络信息信息全部由 ROS 帮忙管理罢了。

然后参数通信实际上是通过服务暴露了控制端点，外界可以通过这个服务来持久化（stateful）地设置系统的参数。

4.1 服务通信基本使用

和话题一样，ROS 同样提供了常用的调试工具：

（查看节点中的通信信息与上一章相同）

- 查看当前机器局域网内所有可用服务的 endpoint 和 interface：`ros2 service list -t`；

  > 这个 interface 的查看方式已经在上一节介绍了（interface show）；

- 作为 service client 调用指定服务：`ros2 service call <service_name> <interface_name> <input_data_json_str>`；

4.2 参数通信基本使用

我们已经知道参数通信就是一种特殊的服务通信，因此可以直接通过 `ros2 service list -t | grep parameter` 来获得和参数相关的服务接口（例如 set/get/describe/list 这些语义）。

- 如果想看节点究竟能配置哪些参数：`ros2 param list`；
- 查看某个参数的详细信息：`ros2 param describe <node_name> <param_name>`；
- 获取/设置指定节点上指定参数的值：`ros2 param get/set <node_name> <param_name> [param_value]`；
- 将指定节点的参数数据结构导出为 YAML 格式（默认输出到 `stdout`）：`ros2 param dump <node_name>`；

  > **Important**
  >
  > 现在我们如果想导入上述 YAML 格式的配置，就可以通过命令行完成，在启动 node 时导入配置：`ros2 run <package_name> <py_module_name/cpp_target_name> --ros-args --params-file <yaml_config_file>`；
  >
  > 如果不导入 YAML 文件，而是只规定特定的参数，那么用 `-p <param:=val>`
  >
  > 如果直接运行可执行文件，可以直接用参数 `--ros-args --params-file` 即可；
  >
  > 读者是否还记得这段代码 `rclcpp::init(argc, argv);`？它的作用之一就是解析上面的配置并初始化库中的某些参数为指定值。

除了上面的命令行调试工具，更方便的方法是使用 ROS 自带的 `rqt` 图形化界面，在命令行执行即可启动，方便我们进行上述所有调试工作（例如查看 node graph、topic 信息、service 信息、参数信息，等等，请读者自行摸索）；

4.3 人脸识别服务 Demo

现在用例子介绍服务通信的使用，顺带介绍常用的、需要注意的点。

4.3.1 自定义服务数据类型

假设我们定义服务接口的数据结构（参数和返回值类型）：

```
sensor_msgs/Image image     # 原始图像数据
---
int16 number         # 人脸个数
float32 use_time     # 识别耗时
int32[] top          # 人脸在图像中的像素坐标
int32[] right
int32[] bottom
int32[] left
```

> 注：如果需要定义常量，使用 `int16 test = 1`（包含赋值语句）即可。

我们使用 ROS 内置的数据类型 `sensor_msgs/Image` 作为输入的图像数据类型。

为服务数据类型创建的功能包的完整步骤，和话题是类似的（都是使用 `rosidl_default_generators`），只是命名不同：将所有的 `msg` 换成 `srv`；

另外相较于 topic，`rosidl_default_generators` 也会多做一些事，例如生成一些 Service 需要使用的构造函数，例如对应的 Request 数据结构、Response 数据结构。我们在后文的示例代码中就能见到。

> **Tip**
>
> 有些资源需要放在 ROS 功能包中分发出去，例如程序可能使用到的固定的图像/文件资产，这个时候我们在代码中如何灵活锁定这些资源的位置呢？
>
> 以 Python 为例，我们可以使用功能包帮我们生成的 `resource` 目录。将资源放入该目录后，配置 `setup.py` 中 `setup` 函数参数 `data_files`（原生的 Python `setuptools` 框架中，它的意思是打包时注意需要打包除了代码外的其他指定数据文件），一般选择放在 `share` 目录下：
>
> ```
> setup(
>     # ...
>  data_files=[
>      ('share/ament_index/resource_index/packages',
>          ['resource/' + package_name]),
>      ('share/' + package_name, ['package.xml']),
>      # 在这里添加
>  ],
>  # ...
> )
> ```
>
> 然后在代码中导入工具函数：
>
> ```
> from ament_index_python.packages import get_package_share_directory
> ```
>
> 这个 `get_package_share_directory(<package_name>) -> str` 就可以得到功能包在对应阶段的 `share` 目录的绝对路径。

4.3.2 ROS 中的图像数据格式

前面介绍过，ROS 中内置的图像格式 `sensor_msgs/Image`，它实际与 OpenCV 中保存的图像格式不同，因此真正想从 ROS 拿到图像数据交给 OpenCV 处理前，需要进行一些转换工作。这里有一个现成的工具：**CV Bridge**，它实现了 `sensor_msgs/Image` 和 OpenCV 图像数据的相互转换。

4.3.3 ROS 服务定义

- 服务端创建：Node 类型实例方法 Python `create_service(<srv_ty>, <srv_name>, <callback>)`，C++ `create_service<srv_ty>(<srv_name>, <callnback>)`；
- 客户端操作：这里仅以 Python 为例，C++ 类似。
  - Node 类型实例方法，创建 `create_client(<srv_ty>, <srv_name>) -> Client`；
  - Client 类型实例方法，连接健康检查：`wait_for_service(timeout_sec)`、调用方法 `call_async/call`（Python）、`async_send_request`（C++）；
- 回调函数类型：Python `def on_callback(request_msg, response_msg)`，注意需要返回指定的 response msg 类型，这些信息最终会由 ROS 管理并进行转发操作；

> **Important**
>
> 服务这里有个很重要的问题需要说明。ROS Python 客户端框架中，启动一个 Node 默认就是一个线程，如果你在异步调用服务时这么写：
>
> ```
> def send_response(self):
>  while not self.client.wait_for_service(timeout_sec=1):
>      pass
>  # rosidl_default_generators 帮我们生成的对应 FaceDetector 专用的 Request 请求体数据结构
>  req = FaceDetector.Request()
>  request.image = self.bridge.cv2_to_imgmsg(self.cv_image)
>  # 异步请求服务
>  future = self.client.call_async(req)
>  while not future.done():
>      time.sleep(1.0)
>  resp = future.result()
>
>  # 下面继续处理响应数据 ...
> ```
>
> 会导致程序效率低下的问题。因为整个 ROS 节点默认共用一个线程，如果后来这个节点要做的事情多了（例如回调函数多了），`time.sleep` 会将该线程 ROS 框架休眠，降低节点处理效率。因此有几种方案：
>
> 1. 使用 Python `future` 添加 callback：`future.add_done_callback`，现在就完全不需要阻塞处理了；
> 2. 目前没有建成 pipeline，没必要异步，直接使用同步调用 `Client.call`；
> 3. 使用 `rclpy.spin_until_future_complete(<node>, <future>)` 阻塞，这个方法会允许 ROS 框架主动收回控制流，然后 spin 调度其他的回调函数，不会真正休眠当前线程。和 `rclpy.spin(<node>)` 区别是后者会完全将当前线程交给当前节点的回调函数，直至核心的 event loop 被打断（例如 signal 导致退出 loop）；

4.3.4 参数化

首先需要声明参数很简单，使用 `Node` 自带方法 `declare_parameter<ParamT>(<param_name>, [default_value])` 即可（Python 不需要有模板参数）。

代码中想要获取参数的值，只需要 `Node` 方法 `get_parameter(<param_name>, <output_ref>)`（未声明/定义可能会导致抛出异常，参加代码注释文档）即可（Python 直接 `output = get_parameter(<param_name>).value`）；

现在的问题是，我们这样都是获取当前的参数值，如何外界动态更新参数值？这就需要我们订阅参数更新事件。我们回忆一下，参数通信本质上也是由服务通信和话题通信实现，我们可以通过查看节点通信信息发现有形如 `set_parameters` 这样的自动生成的服务接口。

因此外部可以通过调用这些服务接口来提示修改参数（**通用接口 `/<node_name>/set_parameters`，参数类型固定 `rcl_interfaces/srv/SetParameters`**、返回值类型固定 **`rcl_interfaces/srv/SetParametersResult`**），而 `Node` 内部也提供了相应的回调注册函数，用来处理更新参数的事件。以 Python 为例：

```
from rcl_interfaces.srv import SetParametersResult

# 建议在 __init__ 设置回调，它会一口气更新所有参数
self.add_on_set_parameters_callback(self.update_params)

# 自己定义一个回调函数
def update_params(self, params) -> SetParametersResult:
    for param in params:
        match param.name:
            case 'paramA':
                pass
            case 'paramB':
                pass
            case _:
                pass
    return SetParametersResult(successful=True)
```

再以 C++ 为例，`add_on_set_parameters_callback` 的回调函数类型定义为：

```
using OnSetParametersCallbackType =
    std::function<
    rcl_interfaces::msg::SetParametersResult(
      const std::vector<rclcpp::Parameter> &)>;
```

客户端想要更新参数就遵循 `rcl_interfaces/srv/SetParameters` 的类型。以 Python 为例：

```
from rcl_interfaces.srv import SetParameters
from rcl_interfaces.msg import Parameter, ParameterValue, ParameterType # 前两个是接口类型，后一个是枚举

# ...
# 构造请求体
# 构造一个参数为例
param = Parameter()
param.name = "..."
param_value = ParameterValue()
param_value.type = ParameterType.PARAMETER_STRING   # 看实际情况
param_value.string_value = "..."
param.value = param_value

req = SetParameters.Request()
req.parameters = [param]    # 这里只传了一个参数，实际可以同时更新多个

# 接下来用 req 请求 SetParameters 服务吧!
```

客户端 set parameters 的代码 C++ 和 Python 使用方法几乎一样，不再赘述。

4.4 补充：动作通信（Action）

特性	Service	Action
通信模型	同步请求-响应 (RPC)	异步目标-反馈-结果
交互过程	客户端发送请求 -> 服务器处理 -> 返回响应	客户端发送目标 -> 服务器开始执行 -> 持续反馈进度 -> 最终返回结果
执行时间	设计用于短时间完成的任务	设计用于长时间运行的任务（秒、分钟、小时）
进度反馈	无	有。服务器可以定期向客户端发送进度更新 (`Feedback`)。
是否可取消	非常困难/不可靠。请求发出后基本无法取消。	支持。客户端可以发送取消请求 (`CancelGoal`)，服务器应响应并停止执行。
抢占 (Preemption)	不支持。新请求会排队或并行（取决于服务器实现）。	支持。客户端可以发送新目标请求抢占当前执行的目标。
状态跟踪	无内置状态。客户端只知道请求成功或失败。	有状态机。客户端可以查询目标状态 (`Accepted`, `Executing`, `Canceled`, `Succeeded`, `Aborted`)。
底层实现	基于一对 Topic (隐藏): `_service` (请求), `_service_reply` (响应)	基于 3 个 Service + 2 个 Topic 的复合结构（对用户透明）
典型应用场景	查询状态、开关设备、简单计算、快速数据获取	导航到点、机械臂轨迹规划与执行、大文件传输、复杂计算、需要监控进度的任务

Chapter 5. 工具

5.1 Launch 启动脚本进行多节点管理

ROS 框架提供了多个节点启动和其他生命周期管理的功能，方便用户在多节点复杂情况下方便地运行整个系统，而不需要多个终端依次手动启动和停止。这个就是 ROS 的 `launch` Python 库（注意这个库 ROS 只提供了 Python 版本）。

使用方法：

1. 无论是 Python 还是 C++ 功能包，都需要在功能包中添加一个 `launch` 目录（名称固定），并且新建管理这个节点启动的 Python 脚本，建议后缀：`.launch.py`；
2. 补全下面的脚本：

```
import launch
import launch_ros

# 注意这个函数签名是固定不能改变的，ROS launch 库会找这个函数直接执行，并通过返回值配置启动
def generate_launch_description() -> launch.LaunchDescription:

    # 可选，如果希望给某个特定的 node 传递参数的话
    # 这里的作用是声明，将命令行来的参数存放到框架内保存，类似于 Python 原生的 ArgumentParser
    launch.actions.DeclareLaunchArgument('<store_name>', [default_value='xxx'])

    action1 = launch_ros.actions.Node(
        package='package name',
        executable='py_module_name or cpp_target_name',
        # 同样可选
        parameters=[
            # 这里的作用类似于原生 Python 从 ArgumentParser 中拿出参数
            {'<arg_name>': launch.substitutions.LaunchConfiguration('<store_name>', [default='xxx'])}
        ],
        output='screen',
    )
    action2 = ... # (同理)

    return launch.LaunchDescription([
        action1, action2,
    ])
```

3. 修改构建脚本，让构建后的该脚本安装到 `install/share/<package_name>/launch` 目录下，ROS 框架会取寻找的！

   以 C++ 功能包的 CMakeLists 为例：

   ```
   # 记得加上它，这会将源码目录下的 launch/ 递归地复制到 install/share/<package_name> 下
   install(DIRECTORY launch DESTINATION share/${PROJECT_NAME})
   ```

   以 Python 功能包的 `setuptools` 的配置 `setup.py` 为例，这也很好理解：

   ```
   setup(
       # ...
    data_files=[
        # 其他内容省略 ...
        ('share/' + package_name + '/launch', glob('launch/*.launch.py')),
    ],
    # ...
   )
   ```

现在就完成了，`source install/setup.xxsh` 后，我们直接通过 **`ros2 launch <package_name> <name>.launch.py [--debug]`** 即可启动多个节点！

现在我们总结一下，实际上 launch 提供了 3 大功能：

- 动作（`actions`），例如上面的参数声明。当然也可以支持调用终端、
- 替换（`substitutions`），例如上面的参数读取；
- 条件（`conditions`），利用条件决定哪些动作执行，哪些不执行（一般结合声明的命令行参数使用）；

5.2 ROS 的 TF (坐标变换) 工具库

在机器人学中，坐标变换是相当麻烦的，ROS 中将坐标转换的工具封装成了类似服务通信的工具，方便开发者使用。

本章的代码需要额外安装 ROS TF 工具库：`apt install ros-$ROS_DISTRO-tf-transformations`（如果你的 ROS 正确安装了，应该可以通过 `$ROS_DISTRO` 环境变量获得你安装 ROS 时的发行版名称）；

5.2.0 前置知识：计算机空间表示、欧拉角、四元数

Krasjet / quaternion - Github；

5.2.1 命令行工具

先介绍命令行的调用方法，方便调试。我们以 `base_link`（机器人坐标系原点在世界坐标系中的坐标）、`base_laser`（机器人雷达组件的工具坐标系原点在世界坐标系中的坐标）、`target_pos`（目标坐标在世界坐标系中的坐标）。如何获取 `target_pos` 在机器人坐标系原点的坐标？

一般情况下雷达测距后，我们只知道目标在雷达的工具坐标系下的坐标（也就是 target 在 laser 下的相对坐标），如何转换成机器人坐标系中的坐标以方便后续判断工作？我们可以利用 TF 工具库进行下面操作：

1. 发布 base-link 与 base-laser 间的坐标变换：

   ```
   ros2 run tf2_ros static_transform_publisher --x <x> --y <y> --z <z> --roll <roll> --pitch <pitch> --yaw <yaw> --frame-id base_link --child-frame-id base_laser
   ```

   注意到上面的指令是将坐标系 `frame-id` 与 `child-frame-id` 间的变换关系，`x, y, z, roll, pitch, yaw` 表示 `child-frame-id` 坐标系相对于 `frame-id` 坐标系的变换方法；

2. 发布 base-laser 与 target 间的坐标变换：

   ```
   ros2 run tf2_ros static_transform_publisher --x <x> --y <y> --z <z> --roll <roll> --pitch <pitch> --yaw <yaw> --frame-id base_laser --child-frame-id target
   ```

3. 我们现在可以通过直接查询的方法获得 base-link 与 target 间的关系：

   ```
   ros2 run tf2_ros tf2_echo base_link target
   ```

5.2.2 原理与代码使用

在我们运行 `tf2_ros` 包的 `static_transform_publisher` 模块后，ROS 框架会管理一个话题接口 `/tf_static`，话题类型 `tf2_msgs/msg/TFMessage`；

详细的定义比较简单易懂，请使用指令自行查看。

上述介绍的 `tf_static_transform` 一般用于相对静态的结构的坐标转换（例如固定在机器人上的雷达系统），ROS 中还提供了 `tf_transform`（动态坐标转换），一般是像轮子/机械臂这样实时运动的工具坐标系，ROS 允许它们动态更新并转换，话题接口 `/tf`；

本章 **C++/Python** 的使用方式相差有点大（主要是内置类型太多了，叙述起来不直观），我们用两个 **Demo** 分别演示。

5.2.3 TF 和手眼标定 Demo (Python)

我们知道在机器人学中，手眼标定有两类，眼在手上和眼在手外。我们先考虑比较简单的情况：眼在手外。

考虑上面的情况，`base_link` 是机器人基坐标系相对于世界坐标系的表示、`camera_link` 是摄像机基坐标系相对于世界坐标系的表示，`bottle_link` 是物体相对于世界坐标系的表示；

眼在手外标定的目的就是，获得转换矩阵使得我们能很容易的由“物体在摄像机基坐标系下的相对坐标”转换到“物体在机器人基坐标系下的相对坐标”。后者是很多机器人执行像抓取物体这样 pick-and-place 任务的必须数据。

注意，这里物体（`bottle_link`）可能是实时移动的，因此我们不应该用前面的 `tf_static_transform` 来发布变换。

> **Tip**
>
> 另外，读者可能想到了，眼在手上是不是在这种情况下是眼在手外的一种情况？也就是 `camera_link` 和 `bottle_link` 一样都可以实时变化。
>
> 的确可以这么做，不过我本节先假设 `camera_link` 是固定的，眼在手上以后讨论。

我们先使用 `tf_static_transform` 发布一个 `tf_static` 话题，表示 `camera_link` 对于 `base_link` 的变换：

```
from rclpy.node import Node
from tf2_ros import StaticTransformBroadcaster
from geometry_msgs.msg import TransformStamped
# 欧拉角转四元数工具
from tf_transformations import quaternion_from_euler


class DemoStaticTFNode(Node):
    def __init__(self):
        super().__init__('static_tf_node_demo')
        self.sbroadcaster = StaticTransformBroadcaster(self)
        self.publish_myself()

    def publish_myself():
        # 构建话题类型的数据结构
        transform = TransformStamped()
        transform.header.stamp = self.get_clock().now().to_msg()
        transform.header.frame_id = "base_link"
        transform.child_frame_id = "camera_link"

        transform.transform.translation.x = <x>
        transform.transform.translation.y = <y>
        transform.transform.translation.z = <z>
        # 旋转部分使用四元数传入（注意不是欧拉角 RPY）
```

```
        # 因此这里需要从欧拉角 RPY 转四元数
        quat = quaternion_from_euler(<r>, <p>, <y>) # unit: rad
        transform.transform.rotation.x = quat[0]
        transform.transform.rotation.y = quat[1]
        transform.transform.rotation.z = quat[2]
        transform.transform.rotation.w = quat[3]

        # 最后发布
        self.sbroadcaster.sendTransform(transform)

# 节点启动代码请自行书写，下略
```

> **Important**
>
> 读者可能会疑惑，我们发现 `tf_static_transform` 只会在节点启动（构造）时发布一次，并不会持续发布（反正是静态的！），那么后来加入进来的订阅者为什么能收到发布的话题数据呢？
>
> 答案是 ROS 中的服务质量管理机制（QoS）。我们查看 `/tf_static` 消息接口的详细信息（`ros2 topic info /tf_static -v`）：
>
> ```
> ...
> QoS Profile:
>     ...
>     Durability: TRANSIENT_LOCAL
>     Lifespan: Infinite
>     ...
>     Liveness lease duration: Infinite
> ```
>
> 这个 `TRANSIENT_LOCAL` 持久化策略以及全局生命周期的配置，会让 ROS 库在发布这个接口时持久化最近上一次的发布结果，每当有订阅者加入时，定义接口就发送持久化的那个结果给订阅者。

然后我们使用动态坐标转换发布 `bottle_link` 关于 `camera_link` 的关系，和静态发布很类似，这里不再赘述，只是提示读者有哪些不同，只需要将上面的代码略加修改即可：

- `from tf2_ros import StaticTransformBroadcaster` 换成 `from tf2_ros import TransformBroadcaster`；
- 使用定时器实时发布（你可以根据需求调节发布频率），而不只是在构造时发布一次；

如果客户端需要订阅查询这样的关系，并且参与后续计算，则可以这么写：

```
from rclpy.node import Node
from tf2_ros import TransformListener, Buffer
from geometry_msgs.msg import TransformStamped
from tf_transformations import euler_from_quaternion


class DemoTFClient(Node):
    def __init__(self):
        super().__init__('tf_node_client_demo')
        self.buf = Buffer()
        self.sbroadcaster = TransformListener(self.buf, self)
        self.timer = self.create_timer(0.01, self.get_tranform)

    def get_tranform():
        try:
```

```
            result = self.buf.lookup_tranform(
                'base_link', 'bottle_link',
                rclpy.time.Time(seconds=0.0),       # 获取关系的时间（seconds=0
表示最新）
                rclpy.time.Duration(1.0)            # 设置超时时间，会抛出异常
            )
            trans = result.transform
            # 现在可以使用 trans.translation, trans.rotation
            # 如果需要的话，可以使用导入的 euler_from_quaternion 将四元数转回可读的欧拉
角
            rpy = euler_from_quaternion((
                trans.rotation.x,
                trans.rotation.y,
                trans.rotation.z,
                trans.rotation.w,
            ))
        except Exception as ex:
            pass
```

5.2.4 TF 与地图坐标转换 Demo (C++)

本节从另一个示例展示在 C++ 中如何使用 TF 来进行静态/动态坐标转换的话题发布/订阅。假设下面的场景：

假设机器人建好图后，知道图上的两点 `target_point`（假设根据需求是静态的）和机器人当前所在位置 `base_link`（实时移动），如何使用 TF 框架来得知 `target_point` 相对于机器人坐标系的坐标，以方便其他操作呢？

> 依赖：`rclcpp`，`tf2_ros`，`geometry_msgs`，`tf2_geometry_msgs`；

首先建立 `target_point` 相对于 map 的静态坐标转换：

```
#include <rclcpp/rclcpp.hpp>
// 提供 /tf 和 /tf_static 的话题类型定义
#include <geometry_msgs/msg/transform_stamped.hpp>
// 提供四元数工具及类型 tf2::Quaternion
#include <tf2/LinearMath/Quaternion.h>
// 提供上面四元数类型转为话题类型中的字段类型的方法
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
// 上面都是数据类型相关的必须引入
// --------------------------
// 下面是静态坐标变换发布接口
#include <tf2_ros/static_transform_broadcaster.h>


class DemoStaticTFNode: public rclcpp::Node {
public:
    DemoStaticTFNode(std::string node_name): rclcpp::Node(node_name) {
        // 注意 publisher 的类型是 tf2_ros::StaticTransformBroadcaster
        this->sbc = std::make_shared<tf2_ros::StaticTransformBroadcaster>
(this);
        this->publish_myself();
    }
private:
    void publish_myself() {
        // 准备要发送的话题数据结构
```

```
        geometry_msgs::msg::TransformStamped transT;
        // 这里和 Python 是相似的
        transT.header.stamp = this->get_clock()->now();
        transT.header.frame_id = "map";
        transT.child_frame_id = "target_point";
        transT.transform.translation.x = <x>;
        transT.transform.translation.y = <y>;
        transT.transform.translation.z = <z>;
        // 这里四元数转换稍有不同
        tf2::Quaternion quat;
        quat.setRPY(<r>, <p>, <y>); // unit: rad
        transT.transform.rotation = tf2::toMsg(quat);

        this->sbc->sendTransform(transT);
    }
};
```

然后发布 `base_link` 相对于 map 的动态坐标转换，我们这里也不需要再赘述了，同样注意动态坐标转换和静态坐标转换的不同：

- `#include <tf2_ros/static_transform_broadcaster.h>` 换成 `#include <tf2_ros/transform_broadcaster.h>`；
- 使用定时器实时发布，而不只是在构造时发布一次；

最后客户端想要查询这样的关系，可以这么写（注意头文件引用和注释）：

```
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/transform_stamped.hpp>
#include <tf2/LinearMath/Quaternion.h>

// 四元数转欧拉角需要引入新的头文件
#include <tf2/utils.h>

// 提供 listener buffer
#include <tf2_ros/buffer.h>
// 坐标转换关系订阅接口
#include <tf2_ros/transform_listener.h>

using namespace std::chrono_literals;


class DemoTFClient: public rclcpp::Node {
public:
    DemoStaticTFNode(std::string node_name): rclcpp::Node(node_name) {
        this->buf_ = std::make_shared<tf2_ros::Buffer>(this->get_clock());
        // 注意 publisher 的类型是 tf2_ros::StaticTransformBroadcaster
        // 最后一个参数表示不额外使用 executor 进行 spin。关于 ROS2 的线程/事件机制，请
参见附加章节
        this->listener_ = std::make_shared<tf2_ros::TransformListener>(*
(this->buf_), this, false);
        this->timer_ = this->create_generic_timer(10ms,
std::bind(&DemoTFClient::get_transform, this));
    }
    void get_transform() {
        try {
```

```
            const auto trans = this->buf_->lookupTransform(
                "base_link", "target_point",
                this->get_clock()->now(),               // 指定时间下的坐标转换信
息
                rclcpp::Duration::from_seconds(1.0f)    // 超时时间
            );
            // 注意一下从四元数获取欧拉角有些不同，且参数顺序也不同
            double roll, pitch, yaw;
            // 传入引用
            tf2::getEulerYPR(trans.rotation, yaw, pitch, roll);
        } catch (const std::exception &ex) {
            // TODO
        }
    }
private:
    std::shared_ptr<tf2_ros::Buffer> buf_;
    std::shared_ptr<tf2_ros::TransformerListener> listener_;
    rclcpp::TimerBase::SharedPtr timer_;
};
```

5.2.5 可视化

如果您希望可视化旋转姿态，可以安装 `apt install ros-$ROS_DISTRO-mrpt2`，然后执行 `3d-rotation-converter` 即可；

如果您希望可视化多个坐标转换服务间的关系，可以通过命令行查看 TF 树：

```
ros2 run tf2_tools view_frames
```

或者使用 RQT 插件 `rqt-tf-tree`，需要安装 `apt install ros-$ROS_DISTRO-rqt-tf-tree`，并且更新配置文件：`rm -rf ~/.config/ros.org/rqt_gui.ini`（通过删除旧文件来更新）；

5.3 RViz 可视化工具

5.2.5 中我们知道如何使用 `tf2_tools` / `rqt-tf-tree` 来观察坐标转换的情况，但是不能很清楚地渲染出形象的图像来告诉我们具体情况。

RViz 是一款在机器人领域应用广泛的数据可视化工具，不仅可以将上面的坐标转换情况更清楚、易懂地渲染成图像，而且可以实现各种机器人数据（如地图建图情况、雷达/点云数据、其他传感器数据等等）的可视化。

只需要你的 ROS 正确安装了，就能在终端执行 `rviz2` 来启动。

- `File -> Save Config As` 保存当前界面；
- 启动时使用 `-d <rviz_file>` 直接打开指定文件；

5.4 BAG 数据记录工具

机器人的数据记录和重放工具。使用：

```
ros2 bag record <topic_name>
```

来记录从现在开始的所有该话题发布的内容（保留时序），中途允许使用空格键暂停记录、恢复记录。最后写入 DB3 数据文件（一般会新建一个含有 `metadata.yaml` 的目录）。

使用下面的指令来重放发布数据：

```
ros2 bag play <data_file_dir>   # DB3 数据文件所在的目录
```

还有更多可配置的内容（如变速、循环重放，等等），请使用 `ros2 bag -h` 自行查看。

Chapter 6. 建模与仿真

机器人系统抽象包括“控制系统”、“传感器”、“执行器”、“环境+机器人物理本体性质”这 4 个组成部分。仿真只需要抓住这 4 个部分即可。

常见支持 ROS 的仿真平台：Gazebo（Classic/Harmonic）、Webots、Matlab Simulink、Unity3D 等。本章将以 Gazebo 为例介绍仿真。

6.1 机器人建模

机器人建模的较为通用的语言之一是 URDF，这是一种 XML 格式，用来描述机器人的物理参数（几何性质、传感器和执行器信息），举例：

```
<?xml version="1.0"?>
<robot name="robot_voxelsky">
    <!-- robot body -->
    <link name="link_base">
        <!-- Appearance -->
        <visual>
            <!-- offsets to its own geometry center -->
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
            <!-- geometry definition -->
            <geometry>
                <cylinder radius="0.10" length="0.12"/>
            </geometry>
            <!-- material, texture & color -->
            <material name="ap_white">
                <color rgba="1.0 1.0 1.0 0.5"/>
            </material>
        </visual>
    </link>

    <!-- Robot IMU (惯性测量传感器) -->
    <link name="link_imu">
        <visual>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
            <geometry>
                <box size="0.02 0.02 0.02"/>
            </geometry>
            <material name="ap_black">
                <color rgba="0.0 0.0 0.0 0.5"/>
            </material>
        </visual>
    </link>
```

```
    <!-- Robot Joint: Combine the different links -->
    <!-- use fixed instead of revolute to represent fixed object -->
    <joint name="imu_joint" type="revolute">
        <!-- joint position -->
        <origin xyz="0.0 0.0 0.03" rpy="0.0 0.0 0.0"/>
        <parent link="link_base"/>
        <child link="link_imu"/>
        <!-- rotation axis -->
        <!-- <axis xyz="0.0 0.0 0.0"/> -->
        <!-- other limits -->
        <!-- <limit lower="0.0" upper="0.0" effort="0.0" velocity="0.0"/> --
>
    </joint>
</robot>
```

> **Tip**
>
> 注 1：可以安装 VSCode 的 URDF 插件，来提供方便的补全提示；
>
> 注 2：`urdf_to_graphviz <urdf_file>` 工具可将 URDF 格式文件转为 Graphviz 格式的图表示语言。Graphviz is an open-source suite from AT&T Labs Research for graph drawing using the DOT language.

规范来说，`*.urdf` 文件建议放在功能包的 `urdf/` 目录下。

另外，如果希望 `urdf` 的内容容易维护、提升代码复用性，可以结合 Xarco 语法传递参数（需要安装 `apt install ros-$ROS_DISTRO-xacro`），然后写在 `*.xacro` 文件中，例如：

```
<?xml version="1.0"?>
<!-- 这里添加 xarco 使用声明 -->
<robot xmlns:xacro="http://www.ros.org/wiki/xarco" name="robot_voxelsky">
    <!-- 这里改成宏函数定义，函数名为 link_base（和机器人 body 组件同名） -->
    <xarco:macro name="link_base" params="length radius">
        <!-- robot body -->
        <link name="link_base">
            <!-- Appearance -->
            <visual>
                <!-- offsets to its own geometry center -->
                <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
                <!-- geometry definition -->
                <geometry>
                    <!-- 这里就能使用变量 -->
                    <cylinder radius="${radius}" length="${length}"/>
                </geometry>
                <!-- material, texture & color -->
                <material name="ap_white">
                    <color rgba="1.0 1.0 1.0 0.5"/>
                </material>
            </visual>
        </link>
    </xarco:macro>
    <!-- 这里传递参数调用宏，来生成 <link> 块 -->
    <xarco:link_base length="0.12" radius="0.1" />
</robot>
```

最后还需要用 `xacro` 指令生成一个 URDF 文件才能使用。这个也可以写到 launch 脚本中，读者可以练习一下。

> **Note**
>
> 其他可能需要的知识点，建议结合现有的样例代码，带着问题自行学习：
>
> - 如何用 `xacro:include` 导入其他 xacro 文件？
> - 如何添加可以旋转的轮胎驱动器组件？
> - 如何让机器人模拟时贴合地面？
> - 如何添加碰撞属性（碰撞时的外观）？
> - 如何添加组件的模拟质量和转动惯量矩阵？

6.2 建模可视化

另外，上述建模可以用 RViz 可视化观察，在 RViz 中添加视图 `RobotModel` 即可。

问题是 RViz 不会主动读取 Joint 关节信息，意味着无法正确处理多个 link 之间的空间几何关系。因此还需要另外的工具 `robot_state_publisher`、`joint_state_publisher`：

- `robot_state_publisher` 直接通过 `/tf_static` 发布静态关节（`fixed`）间的坐标转换信息；
- `robot_state_publisher` 会将动态关节（如 `revolute`）的定义信息通过 `/robot_description` 发布给 `joint_state_publisher`，`joint_state_publisher` 实时订阅并计算当前状态后，再反向发布 `/joint_states` 给 `robot_state_publisher`。最终由 `robot_state_publisher` 通过 `/tf` 动态发布给需要的客户端（如 RViz）；

  > 如果需要使用这个 `joint_state_publisher` 提供动态关节计算能力，还需要安装 `apt install ros-$ROS_DISTRO-joint-state-publisher`；

这样我们就可以通过 launch 脚本来组织启动过程（启动 RViz 前启动 `robot_state_publisher` 和 `joint_state_publisher`，前者需要 URDF 文件的内容字符串作为 `robot_description` 参数才能启动）。

6.3 Gazebo 仿真

安装 gazebo sim：`apt install ros-$ROS_DISTRO-ros-gz`；

> **Note**
>
> 其他可能需要的知识点，建议结合现有的样例代码，带着问题自行学习：
>
> - 如何使用 launch 脚本启动 Gazebo，并将场景和机器人模型导入？
> - 如何配置，并使用插件（例如两轮差速驱动插件），来控制模拟器中的机器人？
> - 在新版 Gazebo Harmonic 中，为什么需要、怎么显式指定 `gz_ros_bridge`？
> - 如何在配置时，让深度相机插件对应的 link 组件的 z 轴指向机器人前进方向（正前方）？
> - 如何使用 RQT 查看模拟相机的深度/彩色图像？如何使用 RViz2 查看激光雷达、IMU 的数据结果？

6.4 `ros2_control` 驱动

上面我们在学习 Gazebo 仿真插件时知道，Gazebo 插件的架构如下图：

这样的设计只能让上层的代码设计适配 Gazebo。如果后续想要迁移到真机上还需要更改上层代码，显然不是个优雅的设计。

于是很自然地，类似于 Linux 操作系统中 VFS 的设计，我们想到可以将中间一个个包含控制器和数据接口的插件用一个统一的框架管理起来，这样对下可以无缝兼容模拟环境（Gazebo）和真实物理机器，对上也零代码适配上层设施。于是 ROS2 的 `ros2_control` 框架应运而生：

> **Important**
>
> 就像 Linux VFS 一样，这样的抽象也会带来一些问题，比如：
>
> - 忽视底层（这里是真实的硬件 / Gazebo 仿真 / 其他模拟器）的特征；
> - 实时性和性能问题：由于 ROS 2 节点和 Gazebo 进程分离，命令和状态需要通过 Transport 和 ROS 2 传输，可能引入微小延迟。对于非常高频率或硬实时仿真，这可能是个考虑点（但在大多数机器人应用场景中影响很小）；

我们可以用这条指令安装：`apt install ros-$ROS_DISTRO-ros2-control`。

另外，`ros2-control` 支持的控制器模块可以通过另外的包来安装：`apt install ros-$ROS_DISTRO-ros2-controllers`，我们可以通过 `apt info` 的依赖包信息来查询 `ros2-control` 支持哪些控制器：

例如我们用到过的两轮差速、IMU 传感器，还有像 Joint State Broadcaster（之前介绍的是发布机器人 `/tf` 信息的时候提到 ROS 原本就有的 `joint_state_publisher`）、PID 控制器、夹爪控制器等等。

现在，同时适配模拟环境（Gazebo 等）和真实物理环境就变得简单了！现在只需要上层的业务逻辑按照 `ros2-control` 的控制器接口写，下层的模拟器 / 硬件逻辑提供 `ros2-control` 约定的接口，即可优雅的进行适配！

以 Gazebo 模拟环境为例，我们需要安装 `apt install ros-$DISTRO-gazebo-ros2-control`（注意这和前面两个包各不相同），让 Gazebo 向 `ros2-control` 的接口管理器提供可用的接口。

> **Note**
>
> 其他可能需要的知识点，建议结合现有的样例代码，带着问题自行学习：
>
> - 如何配置 `ros2-control`，在不影响之前的深度相机/雷达等传感器的基础上，来替代之前的两轮差速、惯性传感器 的 Gazebo 插件？
> - 如何配置 `ros2-control` 来启用机器人力控（关节的力度）控制器？

Chapter 7. 导航与寻路

7.1 概述

注意，在一般场景下（除去自动驾驶等复杂场景），我们不使用卫星来给予机器人定位，一般是用事先建立地图（简称“建图”）的方法，再结合雷达/其他传感器进行环境探测，最终来给予机器人定位能力。

这里的问题是，如果只是使用一般的雷达传感器，会因为高斯噪声导致对环境的不准确的构建。如下图所示，我们将 decay time 设置大一点然后将机器人转身，则会出现下面边界偏移的问题：

为了解决这个问题，我们需要引入 SLAM 来对环境地图的精确构建。

完成建图后，我们需要根据 SLAM 给出的信息（机器人当前位置、环境障碍物信息等）完成代价标记，然后根据全局路径规划算法确认导航路线并且行进。

在行进过程中可能遇到变化的障碍物（例如移动的行人），这个时候就需要将新的障碍物实时添加到地图中，然后重新进行路径规划。由于这是小范围的、动态的更新，因此我们需要在这个小范围区域内重新构建一个局部的代价地图，然后再次进行路径规划。其中“确认局部路径”这一步就称为“局部路径规划”。

在考虑局部路径规划和全局路径规划后，还需要考虑一些特殊情况：例如行走过程中卡住，或者被行人遮挡，又或者找不到局部路径绕过去的时候，就需要一些相应的行为帮助机器人脱离困境。这个行为可以是预设的算法（例如前进卡住时尝试后退、被行人遮挡时播放“请让路”的音频等等），或者结合 AI Agent 考虑决策。我们将这个遇到故障时的脱困行为称为“恢复行为”。

目前业界的机器人导航系统也就围绕着这 3 个部分进行设计的。

7.2 SLAM Toolbox in ROS

本节将以二维情况为例，使用 ROS2 中的 SLAM Toolbox 进行建图。安装：`apt install ros-$ROS_DISTRO-slam-toolbox`；

启动模拟器/真机后，另起终端执行：

```
ros2 launch slam_toolbox online_async_launch.py use_sim_time:=True
```

默认情况该工具会自动监听 TF Tree 的 Root，并创建 `map` frame 和 root frame（例如 odom）的 TF 关系。此时你打开 RViz2，查看 `Map` Display 即可看到 SLAM 对地图的探测情况了。现在你操纵机器人以一个较慢的速度遍历房间后，应该能得到类似这样的效果图：

此时需要保存地图，安装：`apt install ros-$ROS_DISTRO-nav2-map-server`；

然后在新终端中选择一个合适的目录执行：

```
ros2 run nav2_map_server map_saver_cli -f <map_name>
```

这个 server 会订阅 `/map` topic，然后读取数据，最终地图应该正确保存在你指定的 `<map_name>.pgm` 和 `<map_name>.yaml` 文件中。

前者是图像像素编码的地图文件，后者是声明伸缩比例的配置文件。下面是配置文件的含义：

```
image: room.pgm
mode: trinary   # 三色图：黑色表示障碍物（占位区域）、白色表示自由区域，灰色表示未知区域
resolution: 0.05    # 每个像素对应真实物理尺寸（单位：米）
origin: [-10.4, -6.47, 0]   # 地图原点真实物理坐标（单位：米）
negate: 0
occupied_thresh: 0.65   # 归类为占位区域的参考阈值
free_thresh: 0.25       # 归类为自由区域的参考阈值
```

这个地图中每个像素对应一个值就是它被占据的概率，因此这个地图又称为“占据栅格地图”。

7.3 ROS 导航框架：Navigation 2

7.3.1 启动

首先认识一个概念：行为树（**Behavior Tree**）。这个概念起源于游戏设计，用于描述游戏角色的行为（例如 PVZ 中植物只有当僵尸出现时才开始射击）。

在 ROS 的 Navigation 导航框架中，就使用行为树来判定机器人何时应该处于“全局路径规划”/“局部路径规划”/“恢复行为”的哪一种模式。

整个过程遵循：规划器服务器根据目标信息规划全局路径，并交给控制器服务器；控制器服务器则进行机器人状态跟随实时更新；如遇故障则尝试调用恢复器服务器相关服务插件。

安装：`apt install ros-$ROS_DISTRO-navigation2`。另外 Navigation 2 框架提供了一些启动示例以供学习，如果需要可安装 `apt install ros-$ROS_DISTRO-nav2-bringup`；

我们以 `nav2-bringup` 功能包提供的启动配置文件为模板：`/opt/ros/$ROS_DISTRO/share/nav2_bringup/params/nav2_params.yaml`；注意修改其中几个配置：

- `global_costmap` 和 `local_costmap` 中，注意适当设置 `robot_radius`，过大会导致路径不是最优解（甚至无解），过小会导致被卡住，需要按仿真器/物理硬件的实际情况设置；
- `*-frame-*` 相关字眼的配置，与 TF 系的名称有关，注意检查与模拟器设置的名称保持一致；
- `*-topic-*` 相关字眼的配置，与发布话题名称有关，同样注意检查；

> **Note**
>
> 其他可能需要的知识点，建议结合现有的样例代码，带着问题自行学习：
>
> - 如何编写 launch 脚本启动导航系统？

我们需要先启动仿真环境/物理机器，等待里程机 frame 出现后再启动导航系统。启动后一开始系统是不知道机器人在地图的具体位置的，需要我们手动告诉程序机器人的大致位置和朝向（参见 RViz2 的 2D Pose Estimate，按住然后拖动朝向），然后会立即生成一个全局和局部的代价地图。不过不需要太精确，因为机器人移动时会逐步自动校准。这个给定大致初始位置、按照雷达等信息校准的过程默认由 AMCL 组件完成（参见配置文件）。

两种代价地图如下图所示：

7.3.2 单点与路点导航

先以 RViz2 为例，在图形界面完成单点导航，只需要启动后完成 2D Pose Estimate，然后单击 Nav2 Goal，选中目标点即可进行单点导航。

另外单击 `Waypoint / Nav Through Poses Mode` 即可收集路点。

7.3.3 导航速度和膨胀半径优化

现在考虑如何优化导航时的导航速度和膨胀半径。我们知道，在 Nav 2 架构中真正实时控制导航速度（`/cmd_vel`）的机构是控制器服务器（`controller_server`），因此考虑在配置文件中找到 `controller_server -> ros__parameters -> FollowPath` 即可修改行进过程中的所有速度限制配置。

除了设置机器人的半径参考以外，`global_costmap` 和 `local_costmap` 的配置中还有一个“膨胀层”（`inflation_layer`），用来添加额外的代价，防止机器人过于接近障碍物（参考上图中品红色的区域实际上比机器人半径大），我们可以通过微调 `inflation_radius` 来得到较为理想的效果。

7.3.4 目标容差优化

可以修改配置文件中 `controller_server -> ros__parameters -> general_goal_checker`，即可发现 `*_tolerance` 的配置。

7.3.5 使用话题导航

现在考虑使用接口导航而不是 RViz GUI。注意几点：

A. 初始化位姿

AMCL 会订阅 `/initialpose [geometry_msgs/msg/PoseWithCovarianceStamped]` 话题，向该话题接口发布数据即可完成初始化位姿的操作（即上面提到的 2D Pose Estimate）；

> 退出时可以保存上一次的位姿，下次导航系统重启后，考虑直接传入原先位置数据（如果机器人没有被移动过的话）。

在 Python / C++ 中有一个包可以帮我们完成这项工作：**`nav2_simple_commander`**。我们只需要给出一个 `geometry_msgs/msg/PoseStamped` 即可由该包的接口转换为 `/initialpose` 需要的数据类型并发送，实现初始化位姿的效果。示例代码：

```python
# init_robot_pose.py
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator
import rclpy

def main():
    rclpy.init()
    navigator = BasicNavigator()
    initial_pose = PoseStamped()
    initial_pose.header.frame_id = 'map'
    initial_pose.header.stamp = navigator.get_clock().now().to_msg()
    initial_pose.pose.position.x = 0.0
    initial_pose.pose.position.y = 0.0
    initial_pose.pose.orientation.w = 1.0
    navigator.setInitialPose(initial_pose)
    navigator.waitUntilNav2Active()
    rclpy.spin(navigator)
    rclpy.shutdown()

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

具体 `setInitialPose` 和 `waitUntilNav2Active` 在做什么，读者可以自行探索。

> **Important**
>
> 其实代码初始化位姿的好处不仅仅在于上次的自动位姿记录和自动初始化，更在于我们可以定制更准确的初始化位姿的流程，例如结合其他传感器数据减小位姿误差（如在原点做一个类似二维码标记，当相机看到则相当于告诉机器人位置在原点附近，然后直接调用这个方法告诉机器人当前在原点附近）；

B. 实时获取机器人位置

AMCL 节点启动后会结合初始位置向外发布 `map -> odom` 的 TF 信息（参见配置文件 `amcl -> ros__parameters`）。我们只需要订阅 `/tf`（或者你设置的 topic 名称）即可，获取代码请参见 5.2.3/5.2.4；

C. 单点导航

这里我们终于需要用到 ROS 的动作通信（action）了。动作通信特点之一是反馈机制，在客户端向服务端调用后，可以实时获取服务端的状态反馈（动作进度），我们可以类似 topic/service 地查看当前动作列表：`ros2 action list`。

我们通过 `ros2 interface show` 可以观察到，动作通信的接口类型有 3 个部分：`action goal, result, feedback`（回想 Service 有两个部分、Topic 只有一个部分）。

> **Tip**
>
> 查看 Action 的源码可知，动作通信就是由 3 个 service 和两个 topic 实现的，存在 `goal_service, result_service, cancel_service` 以及组织接口的 `feedback_topic, status_topic`；

Navigation 2 的单点导航暴露 action 接口：`/navigate_to_pose [nav2_msgs/action/NavigateToPose]`，请求这个接口即可完成任务！

我们可以先用命令行试一下：`ros2 action send_goal <action_name> <action_type> <json_data_str> [--feedback]`；

另外对于代码，`nav2_simple_commander` 也帮我们封装好了，直接使用即可，以 Python 为例：

```
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult
import rclpy
from rclpy.duration import Duration


def main():
    rclpy.init()
    navigator = BasicNavigator()
    # 等待导航启动完成
    navigator.waitUntilNav2Active()
    # 设置目标点坐标
    goal_pose = PoseStamped()
    goal_pose.header.frame_id = 'map'
    goal_pose.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose.pose.position.x = 1.0
    goal_pose.pose.position.y = 1.0
    goal_pose.pose.orientation.w = 1.0
    # 发送目标接收反馈结果
    navigator.goToPose(goal_pose)
    while not navigator.isTaskComplete():
        feedback = navigator.getFeedback()
        navigator.get_logger().info(
            f'预计:
{Duration.from_msg(feedback.estimated_time_remaining).nanoseconds / 1e9} s 后到达')
        # 超时自动取消
```

```
        if Duration.from_msg(feedback.navigation_time) >
Duration(seconds=600.0):
            navigator.cancelTask()
    # 最终结果判断
    result = navigator.getResult()
    if result == TaskResult.SUCCEEDED:
        navigator.get_logger().info('导航结果：成功')
    elif result == TaskResult.CANCELED:
        navigator.get_logger().warn('导航结果：被取消')
    elif result == TaskResult.FAILED:
        navigator.get_logger().error('导航结果：失败')
    else:
        navigator.get_logger().error('导航结果：返回状态无效')

    # 这里导航结束后，导航客户端所在节点会直接退出

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

> ⚠ **Warning**
>
> 注意到 `waitUntilNav2Active` 的默认行为是先查看是否设置了 initial pose，如果没有会自动传入 `(0, 0, 0)` 的全 0 位姿。如果机器人起始位置并不是这个位置，则很有可能导致误差过大。因此建议先主动调用初始化位姿后，再让代码运行到 `waitUntilNav2Active`；

C++ 的版本：

```
#include <memory>

#include "nav2_msgs/action/navigate_to_pose.hpp"  // 导入导航动作消息的头文件
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"  // 需要额外导入 ROS 2 的 C++ Action
客户端库!

using NavigationAction = nav2_msgs::action::NavigateToPose;  // 定义导航动作类型
为 NavigateToPose

class NavToPoseClient : public rclcpp::Node {
 public:
  using NavigationActionClient = rclcpp_action::Client<NavigationAction>;
  using NavigationActionGoalHandle =
      rclcpp_action::ClientGoalHandle<NavigationAction>;

  NavToPoseClient() : Node("nav_to_pose_client") {
    action_client_ = rclcpp_action::create_client<NavigationAction>(
        this, "navigate_to_pose");
  }

  void sendGoal() {
    while (!action_client_->wait_for_action_server(std::chrono::seconds(5)))
{
      RCLCPP_INFO(get_logger(), "等待Action服务上线。");
    }
    // 设置导航目标点
    auto goal_msg = NavigationAction::Goal();
```

```
    goal_msg.pose.header.frame_id = "map";
    goal_msg.pose.pose.position.x = 2.0f;
    goal_msg.pose.pose.position.y = 2.0f;

    auto send_goal_options =
        rclcpp_action::Client<NavigationAction>::SendGoalOptions();
    // 设置请求目标结果回调函数
    send_goal_options.goal_response_callback =
        [this](NavigationActionGoalHandle::SharedPtr goal_handle) {
          if (goal_handle) {
            RCLCPP_INFO(get_logger(), "目标点已被服务器接收");
          }
        };
    // 设置移动过程反馈回调函数
    send_goal_options.feedback_callback =
        [this](
            NavigationActionGoalHandle::SharedPtr goal_handle,
            const std::shared_ptr<const NavigationAction::Feedback>
feedback) {
          (void)goal_handle;  // 假装调用，避免 warning: unused
          RCLCPP_INFO(this->get_logger(), "反馈剩余距离:%f",
                      feedback->distance_remaining);
        };
    // 设置执行结果回调函数
    send_goal_options.result_callback =
        [this](const NavigationActionGoalHandle::WrappedResult& result) {
          if (result.code == rclcpp_action::ResultCode::SUCCEEDED) {
            RCLCPP_INFO(this->get_logger(), "处理成功！");
          }
        };
    // 发送导航目标点
    action_client_->async_send_goal(goal_msg, send_goal_options);
  }

  NavigationActionClient::SharedPtr action_client_;
};

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<NavToPoseClient>();
  node->sendGoal();
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

D. 路点导航

类似地，Navigation 2 的路点导航暴露 action 接口：`/follow_waypoints [nav2_msgs/action/FollowWaypoints]`，请求这个接口即可完成任务！

以 Python 为例：

```
from geometry_msgs.msg import PoseStamped
```

```python
from nav2_simple_commander.robot_navigator import BasicNavigator, TaskResult
import rclpy
from rclpy.duration import Duration

def main():
    rclpy.init()
    navigator = BasicNavigator()
    navigator.waitUntilNav2Active()
    # 创建点集
    goal_poses = []
    goal_pose1 = PoseStamped()
    goal_pose1.header.frame_id = 'map'
    goal_pose1.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose1.pose.position.x = 0.0
    goal_pose1.pose.position.y = 0.0
    goal_pose1.pose.orientation.w = 1.0
    goal_poses.append(goal_pose1)
    goal_pose2 = PoseStamped()
    goal_pose2.header.frame_id = 'map'
    goal_pose2.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose2.pose.position.x = 2.0
    goal_pose2.pose.position.y = 0.0
    goal_pose2.pose.orientation.w = 1.0
    goal_poses.append(goal_pose2)
    goal_pose3 = PoseStamped()
    goal_pose3.header.frame_id = 'map'
    goal_pose3.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose3.pose.position.x = 2.0
    goal_pose3.pose.position.y = 2.0
    goal_pose3.pose.orientation.w = 1.0
    goal_poses.append(goal_pose3)
    # 调用路点导航服务
    navigator.followWaypoints(goal_poses)
    # 判断结束及获取反馈
    while not navigator.isTaskComplete():
        feedback = navigator.getFeedback()
        navigator.get_logger().info(
            f'当前目标编号: {feedback.current_waypoint}')
    # 最终结果判断
    result = navigator.getResult()
    if result == TaskResult.SUCCEEDED:
        navigator.get_logger().info('导航结果: 成功')
    elif result == TaskResult.CANCELED:
        navigator.get_logger().warn('导航结果: 被取消')
    elif result == TaskResult.FAILED:
        navigator.get_logger().error('导航结果: 失败')
    else:
        navigator.get_logger().error('导航结果: 返回状态无效')

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

7.4 ROS 导航自定义算法

我们需要为 ROS2 Navigation 2 框架编写插件来完成导航算法的自定义。ROS2 中的所谓插件就是动态链接库，通过运行时的 `ClassLoader` 加载符号并执行。

我们需要安装 `apt install ros-$ROS_DISTRO-pluginlib` 来进行自定义插件的编写和使用工作。