4. ros2常用工具

4.1 launch

我们已经知道了，一个机器人的各种功能需要通过多个节点来实现，但我们在启动一个机器人的所有功能时，不可能一个一个地`ros2 run ...`，再一个一个关闭。ros2为我们提供了launch文件这个工具来管理节点的启动。简单来说，我们可以通过运行一个launch文件，来启动多个节点。

ros1中launch文件只支持xml格式的.launch文件，ros2支持python、yaml、xml三种格式的launch文件。

以ros2 demo_nodes_cpp中的talker_listener.launch.py为例。

大部分使用的例程源码来自：
https://github.com/ros2/demos/tree/humble/demo_nodes_cpp

创建launch_ros.actions.Node对象时的常见参数：

```
package: 包名
name: 启动后的节点名
namespace: 命名空间，防止同名节点冲突
executable: 可执行文件名/脚本名
parameters: 参数，常用yaml文件
remappings: 重映射
```

一个launch文件还可以包含其他的launch文件：

```
   other_launch = IncludeLaunchDescription(        # 包含指定路径下的另外一个launch文件
      PythonLaunchDescriptionSource([os.path.join(
         get_package_share_directory('example_bringup'), 'launch'),
         'param_bringup.launch.py'])
      )
```

launch文件还可以从yaml文件加载参数：

```
    config = os.path.join(                # 找到参数文件的完整路径
      get_package_share_directory('example_bringup'),
      'config',
      'config.yaml'
      )
```

找到参数文件后，把需要的launch_ros.actions.Node对象的parameters参数设置为config。以listener为例：

```
    listener_node= Node(
            package='listener',
            executable='listener_node',
            name='listener_node',
            parameters=[config]
        )
```

TimerAction 可以在指定的时间后执行一个action，下面的例子就是在2秒后再启动listener

```
    delay_listener_node = TimerAction(
        period=2.0,
        actions=[listener],
    )
```

4.2 component

很多时候我们并不用main函数来启动一个节点，而是使用组件。component会被构建成一个动态库，然后在运行时由容器进程加载。

使用的例程源码来自：https://github.com/ros2/demos/tree/humble/composition

以talker_component.cpp为例：

无需再写main函数，在Node类的子类的.cpp文件最后添加下面两句,让这个类使用来自包 `rclcpp_components` 的宏注册自身，使得当前Node被进程加载时能发现这个组件

```
#include "rclcpp_components/register_node_macro.hpp"
RCLCPP_COMPONENTS_REGISTER_NODE(composition::Talker)
```

在CMakeLists.txt中添加：

```
add_library(talker_component SHARED
   src/talker_component.cpp)
rclcpp_components_register_nodes(talker_component
"composition::Talker")
```

在launch中使用ComposableNodeContainer类实现在同一进程中启动多个组件：示例如下：

```
def generate_launch_description():
    """Generate launch description with multiple components."""
    container = ComposableNodeContainer(
            name='my_container',
            namespace='',
            package='rclcpp_components',
            executable='component_container',
            composable_node_descriptions=[
                ComposableNode(
                    package='composition',
                    plugin='composition::Talker',
                    name='talker'),
                ComposableNode(
                    package='composition',
                    plugin='composition::Listener',
                    name='listener')
            ],
            output='screen',
    )

    return launch.LaunchDescription([container])
```

4.3 可视化工具

- rviz
- rqt
- foxglove

4.4 命令行工具

```
ros2 topic -h
ros2 node -h
ros2 service -h
...
```

4.5 rosbag

录制和回放工具。

```
ros2 bag record topic-name
```

录制所有话题

```
ros2 bag record -a
```

播放

```
ros2 play xxx
```

4.6 URDF

使用的例程源码来自：https://github.com/ros/urdf_tutorial/tree/ros2

URDF是一种使用xml格式的机器人描述工具。
URDF建模的关键任务就是描述清楚机器人的每一个 `<link>` 和 `<joint>`，这个描述包括外观、一些物理属性、link和joint之间的连接关系等等。

1. link

link用来描述机器人某个**刚体**部分的外观和物理属性，外观包括尺寸、颜色、形状，物理属性包括质量、惯性矩阵、碰撞参数等。

```
<?xml version="1.0"?>
<robot name="myfirst">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```

link标签中的name表示该连杆的名称，我们可以自定义，未来描述joint和link之间的连接时，会使用到这个名称。`<visual>` 部分用来描述机器人的外观，`<geometry>` 表示几何形状、 `<origin>`表示坐标系相对初始位置的偏移，分别是x、y、z方向上的平移，和roll、pitch、yaw旋转，不需要偏移的话，就全为0。

link不只有 `<visual>`这一个标签来描述连杆的性质，还有 `<collision>`、`<inertial>...`

2. joint

每一个link之间通过joint连接。

```
  <joint name="left_gripper_joint" type="revolute">
    <axis xyz="0 0 1"/>
    <limit lower="0.0" upper="0.548"/>
    <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
    <parent link="gripper_pole"/>
    <child link="left_gripper"/>
  </joint>
```

joint用以下标签来描述：

- parent标签：描述父连杆；
- child标签：描述子连杆，子连杆会相对父连杆发生运动；
- origin表示两个连杆坐标系之间的关系，可以理解为这两个连杆该如何安装到一起；

- axis表示关节运动轴的单位向量，比如z等于1，就表示这个旋转运动是围绕z轴的正方向进行的；
- limit就表示运动的一些限制，比如最小位置，最大位置，和最大速度等。

urdf单位：m rad m/s rad/s

robot_state_publisher 将URDF描述的机器人信息发布到TF上。

4.7 xacro

urdf基础上的简化版。

教程 https://github.com/ros/xacro/wiki

- 常量与参数

  常量声明

  ```
  <xacro:property name="width" value="0.2" />
  ```

  常量使用

  ```
  <cylinder radius="${width}"length="${bodylen}"/>
  ```

  参数声明

  ```
  <xacro:arg name="xyz" default="0.10 0 0.05" />
  ```

  参数使用

  ```
  <origin xyz="$(arg xyz)" rpy="$(arg rpy)" />
  ```

- 宏定义

  一个平衡小车有2个轮子，2个轮子都一样，我们就没必要创建2个一样的link，像函数定义一样，做一个可重复使用的模块就可以了。定义方式是通过xacro:macro标签描述的。

  ```
  <xacro:macro name="leg" params="prefix reflect">
      <link name="${prefix}_leg">
          <visual>
              <geometry>
  ```

```
                <box size="${leglen} 0.1 0.2"/>
            </geometry>
            <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
            <material name="white"/>
        </visual>
        <collision>
            <geometry>
                <box size="${leglen} 0.1 0.2"/>
            </geometry>
            <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
        </collision>
        <xacro:default_inertial mass="10"/>
    </link>

    <joint name="base_to_${prefix}_leg" type="fixed">
        <parent link="base_link"/>
        <child link="${prefix}_leg"/>
        <origin xyz="0 ${reflect*(width+.02)} 0.25" />
    </joint>
    <!-- A bunch of stuff cut -->
</xacro:macro>
```

```
<xacro:leg prefix="right" reflect="1" />
<xacro:leg prefix="left" reflect="-1" />
```

- 文件包含

  复杂机器人的模型文件可能会很长，为了切分不同的模块，比如底盘、传感器，我们还可以把不同模块的模型放置在不同的文件中，然后再用一个总体文件做包含调用。

```
<xacro:include filename="$(find package)/other.xacro" />
```

- 数学计算
  在${}中实现。

```
<origin xyz="0 0 0" rpy="${-width/2} 0 ${-pi/2}" />
```

4.8 tf2

tf2是ros的坐标系管理工具。tf2会随着时间的推移跟踪所有坐标系，使用者可以获取两个坐标系之间的转换关系等。

使用的例程源码来自：

https://github.com/ros2/geometry2/tree/humble

https://github.com/ros/geometry_tutorials/tree/ros2/turtle_tf2_cpp

参考教程：

https://wiki.ros.org/tf/Tutorials

- 发布静态TF广播：两坐标系之间的相对位置不改变

```
auto tf_static_broadcaster_ =
std::make_shared<tf2_ros::StaticTransformBroadcaster>(this);
geometry_msgs::msg::TransformStamped t;

    t.header.stamp = this->get_clock()->now();
    t.header.frame_id = "world";
    t.child_frame_id = transformation[1];

    t.transform.translation.x = atof(transformation[2]);
    t.transform.translation.y = atof(transformation[3]);
    t.transform.translation.z = atof(transformation[4]);
    tf2::Quaternion q;
    q.setRPY(
      atof(transformation[5]),
      atof(transformation[6]),
      atof(transformation[7]));
    t.transform.rotation.x = q.x();
    t.transform.rotation.y = q.y();
    t.transform.rotation.z = q.z();
    t.transform.rotation.w = q.w();
tf_static_broadcaster_->sendTransform(t);
```

- 发布动态TF广播

```
// Initialize the transform broadcaster
   auto tf_broadcaster_ =
      std::make_unique<tf2_ros::TransformBroadcaster>(*this);
```

发布内容的消息类型依然是geometry_msgs::msg::TransformStamped，但是值会随机器人的状态变化。

- 监听TF广播：

```
    // 创建保存坐标变换信息的缓冲区
    auto tf_buffer_ =
      std::make_unique<tf2_ros::Buffer>(this->get_clock());
    // 创建坐标变换的监听器
    auto tf_listener_ =
      std::make_shared<tf2_ros::TransformListener>
(*tf_buffer_);
    geometry_msgs::msg::TransformStamped t;

      // Look up for the transformation between target_frame
and turtle2 frames
        try {
          // 监听当前时刻源坐标系到目标坐标系的坐标变换
          t = tf_buffer_->lookupTransform(
            toFrameRel, fromFrameRel,
            tf2::TimePointZero);
        } catch (const tf2::TransformException & ex) {
          RCLCPP_INFO(
            this->get_logger(), "Could not transform %s to %s:
%s",
            toFrameRel.c_str(), fromFrameRel.c_str(),
ex.what());
          return;
        }
```