5. ROS2 Function Packages

1. Introduction to Function Packages

Each robot may have many functions, such as motion control, visual perception, and autonomous navigation. While it's possible to lump the source code for all of these functions together, when we want to share some of these functions with others, we often find the code all mixed together, making it difficult to separate them.

Function packages work in this way. We separate the code for different functions into separate packages, minimizing the coupling between them. When sharing with others in the ROS community, we only need to explain how to use the package, and others can quickly start using it.

Thus, the function package mechanism is one of the key methods for increasing software reuse in ROS.

2. Creating Function Packages

How do I create a function package in ROS2? We can use this command:

```
ros2 pkg create <package_name> --build-type <build-type> --dependencies
<dependencies> --node-name <node-name>
```

In the ros2 command:

- **pkg**: Indicates the functions associated with the package;
- **create**: Creates the package;
- **package_name**: Required: The name of the new package;
- **build-type**: Required: Indicates whether the newly created package is C++ or Python. If using C++ or C, use ament_cmake; if using Python, use ament_python;
- **dependencies**: Optional: Indicates the package's dependencies. A C++ package must include rclcpp; a Python package must include rclpy, as well as other required dependencies;
- **node-name**: Optional: The name of the executable program. The corresponding source files and configuration files will be automatically generated.

For example, to create C++ and Python versions of the package in the terminal:

- Switch to the src directory of the workspace.
- Replace "workspace" with your actual folder path.

```
cd workspace/src
```

- Create a C++ package example

```
ros2 pkg create pkg_helloworld_cpp --build-type ament_cmake --dependencies
rclcpp --node-name helloworld
```

- Create a Python package example

```
ros2 pkg create pkg_helloworld_py --build-type ament_python --dependencies rclpy
--node-name helloworld
```

3. Compile the package

In the created package, we can continue writing code. We will then need to compile and configure environment variables for proper operation:

- Switch to the workspace directory
- Compile all packages

```
colcon build
```

- Compile a specific package

```
colcon build --packages-select pkg1 pkg2
```

4. Complete Workspace Structure with Feature Packages

The directory structure of a ROS2 workspace is as follows:

```
WorkSpace --- Customized workspace.
    |--- build: Directory for storing intermediate files. A separate subdirectory is created within this directory for each feature package.
    |--- install: Installation directory. A separate subdirectory is created within this directory for each feature package.
    |--- log: Log directory for storing log files.
    |--- src: Directory for storing feature package source code.
        |-- C++ Feature Package
            |-- package.xml: Package information, such as package name, version, author, and dependencies.
            |-- CMakeLists.txt: Configuration of compilation rules, such as source files, dependencies, and target files.
            |-- src: Directory for C++ source files.
            |-- include: Directory for header files.
            |-- msg: Directory for message interface files.
            |-- srv: Directory for service interface files.
            |-- action: Action interface file directory.
        |-- Python package
            |-- package.xml: Package information, such as package name, version, author, and dependencies.
            |-- setup.py: Similar to the CMakeLists.txt file for a C++ package.
            |-- setup.cfg: Basic package configuration file.
            |-- resource: Resource directory.
            |-- test: Stores test-related files.
            |-- Package directory with the same name: Python source file directory.
```

In addition, both Python and C++ packages can customize directories related to configuration files.

```
|-- C++ or Python package
    |-- launch: Stores launch files.
    |-- rviz: Stores rviz2 configuration files.
    |-- urdf: Stores robot model files.
    |-- params: Stores parameter files.
    |-- world: Stores simulation environment-related files.
    |-- map: Stores map files required for navigation.
    |-- ......
```

These directories can also be defined with other names, or additional directories can be created as needed.