12.ROS action server

In the last lesson, we learned about the principle of action communication and how to create an action client. This lesson explains how to write an action server.

12.1. Write action client code

Under the learn_action function package, create a new file named action_server.cpp and copy the following content into it.

```
#include <ros/ros.h>
#include <actionlib/server/simple_action_server.h>
#include "learn_action/DoDishesAction.h"

typedef actionlib::SimpleActionServer<learn_action::DoDishesAction> Server;

// Callback function called after receiving the action's goal
void execute(const learn_action::DoDishesGoalConstPtr& goal, Server* as)
{
    ros::Rate r(1);
    learn_action::DoDishesFeedback feedback;

    ROS_INFO("Dishwasher %d is working.", goal->dishwasher_id);

    // Assume the progress of washing dishes, and publish progress feedback at a
frequency of 1hz
    for(int i=1; i<=10; i++)
    {
        feedback.percent_complete = i * 10;
        as->publishFeedback(feedback);
        r.sleep();
    }

    // When the action is completed, the result is returned to the client.
    ROS_INFO("Dishwasher %d finish working.", goal->dishwasher_id);
    as->setSucceeded();
}

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

    // Define a server
    Server server(n, "do_dishes", boost::bind(&execute, _1, &server), false);

    // Server started running
    server.start();

    ros::spin();

    return 0;
```

```
}
```

Modify CMakeLists.txt file and add the following content,

```
add_executable(action_server src/action_server.cpp)
target_link_libraries( action_server ${catkin_LIBRARIES})
add_dependencies(action_server ${${PROJECT_NAME}_EXPORTED_TARGETS})
```

Return to the workspace to compile,

```
cd ~/ros_ws
catkin_make
```

Let's look at the core code first,

```
// Define a server
Server server(n, "do_dishes", boost::bind(&execute, _1, &server), false);
// Server started running
server.start();
```

A server is defined and the service provided is `"do_dishes"`, and then the server starts running. The specific content of the service is the excute function.

```
// Callback function called after receiving the action's goal
void execute(const learn_action::DoDishesGoalConstPtr& goal, Server* as)
{
    ros::Rate r(1);
    learn_action::DoDishesFeedback feedback;

    ROS_INFO("Dishwasher %d is working.", goal->dishwasher_id);

    // Assume the progress of washing dishes, and publish progress feedback at a
frequency of 1hz
    for(int i=1; i<=10; i++)
    {
        feedback.percent_complete = i * 10;
        as->publishFeedback(feedback);
        r.sleep();
    }

    // When the action is completed, the result is returned to the client.
    ROS_INFO("Dishwasher %d finish working.", goal->dishwasher_id);
    as->setSucceeded();
}
```

The main service content here has two points. One is `as->publishFeedback(feedback)`, where the feedback is posted back to the client. After the client receives it, it will execute the content in the feedbackCb function;The other is `as->setSucceeded();`, which sets the status of the active target to successful. Then the client receives it and executes the content in the doneCb function.

12.2 Run DoDishes program

Combined with the client program written in the previous section, let's take a look at how the entire DoDishes runs. First, we still need to start roscore and enter in the terminal.

```
roscore
```

Run action client program,

```
rosrun learn_action action_client
```

Run action server program,

```
rosrun learn_action action_server
```

After the program is started, the client will print the feedback information sent back by the server at a frequency of one point. After 100 seconds, the entire action is completed and "Yay!The dishes are now clean", at that time, the server will also print "Dishwasher 1 finishing working.".