ROS 講習資料

文責 小林照

2021/10

1. まえがき

ロボットを動かす上で ROS の知識は欠かせないものだ．しかし，初心者にとって ROS のコマンドラインインターフェースは理解しにくい上，覚えにくい．そこで，わかりにくい ROS の概要を明文化して少しでも ROS を習得できる人が増えることを期待して，この資料を作成した．筆者は ROS および Linux への理解が深いわけではない上，急ぎ作成したものであるため，誤っている点や煩雑な点などがあるかもしれないが，ご了承願いたい．

2. ROS とは

Robot Operating System の略．ロボットを制御するためのオープンソースソフトウェアであり，本サークルではソフト班に所属する人は，このソフトウェアを使用することになる．基本的にはターミナルからコマンドを打ち込んで使用する．

3. ROS のディレクトリ構成

ROS を使用する環境であるディレクトリ構成について説明する．図 1 のディレクトリ構成図を見ながら読み進めてほしい．

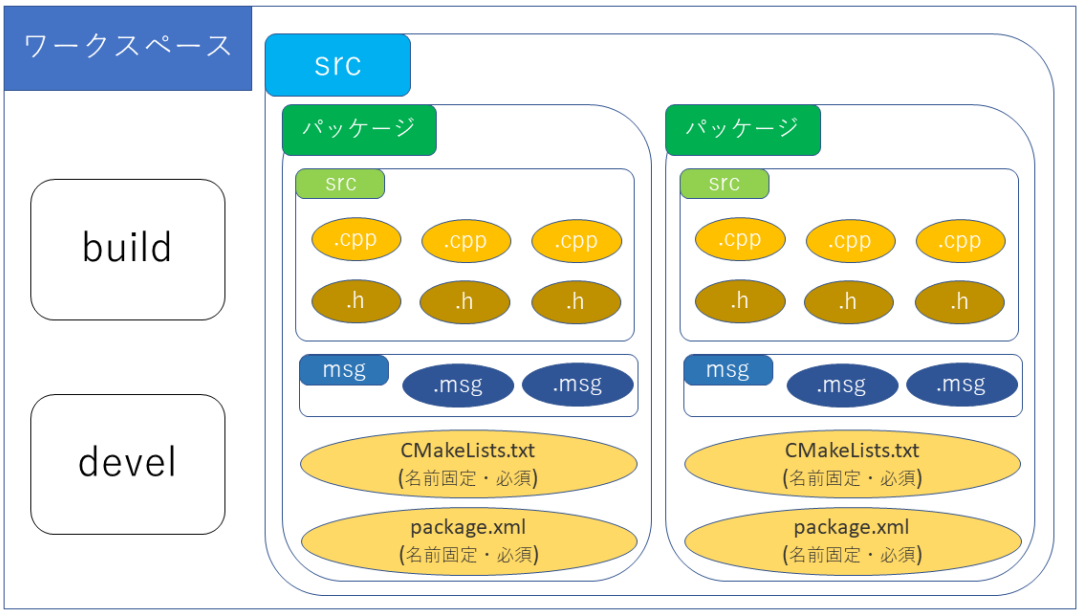

図 1　ROS のディレクトリ構成図

3.1 ワークスペース

ROS 関係のファイル全てが置かれたディレクトリ．名前は任意である．ワークスペースの中には build や devel, src などのディレクトリが存在する．src の中に各パッケージ(後述)を入れる．（この資料では build や devel にはこれ以上触れない．）

3.2 パッケージ

目的に沿ったノードやメッセージなどが入ったディレクトリ．中には src ディレクトリや CMakeLists.txt(名前固定・必須) や package.xml(名前固定・必須)などが存在する．src には.cpp や.h といったノード(後述)のソースコードが置かれる．CMakeLists.txt はビルド(コンパイル)時に読み込まれるファイルで，コンパイル条件などを記述する．package.xml にはパッケージに必須の情報を記述する．パッケージ名はディレクトリ名ではなく，package.xml の中の name タグによって決められている．一般的にはパッケージ単位で開発を行う．

4. ROS のグラフ概念

ROS を理解する上で重要なグラフ概念について説明する．図 2 のグラフ概念図を見ながら読み進めてほしい．

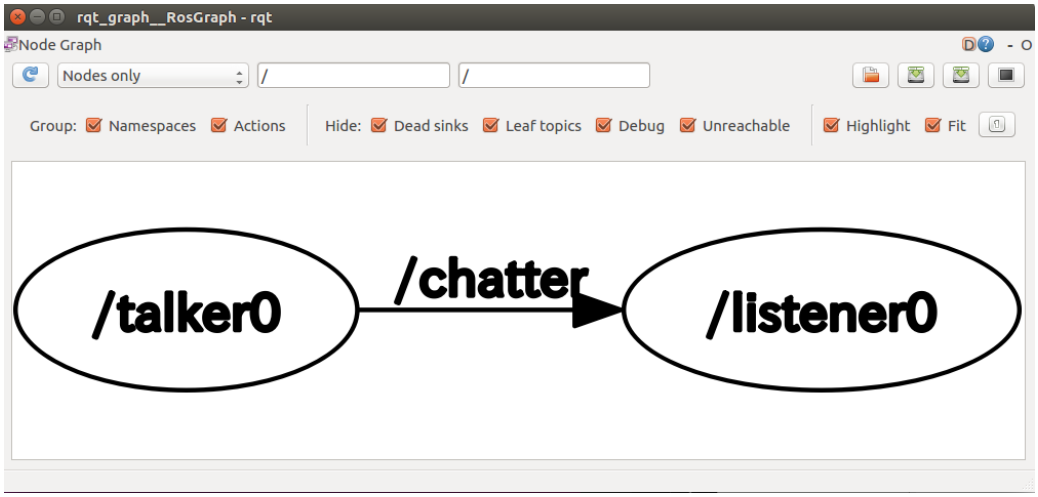

図 2　ROS のグラフ概念図

4.1 ノード

ROS パッケージ内の実行ファイルのことである．図 2 の”talker0”と”listener0”にあたる．ノードは他のノードと通信をして，情報のやりとりをする．ROS ではこのノード同士の通信によって，目的のシステムを構成していくことになる．一般的には Python や C++などのプログラミング言語を使って内容を記述する．

4.2 トピック

ノード同士がやりとりするデータのこと．図 2 の”chatter”にあたる．正確には異なるが，「ノード同士の通信の名前」と捉えるとプログラムが書きやすいかもしれない．トピックを送ることを Publish といい，送る側のノードを Publisher という．また，トピックを受け取ることを Subscribe といい，受け取る側のノードを Subscriber という．

4.3 メッセージ

トピックのデータ型のこと．Arduino や C 言語と同じように整数や浮動小数，配列などがある．カスタムメッセージをつくることで，自分で新たなメッセージ(型)を定義することもでき，整数や浮動小数などを複数もつメッセージなどを作成できる．

4.4　マスター

厳密にはグラフ概念に関する用語ではないが，後の説明を理解する上で役立つので，ここで解説しておく．マスターとは，ノードの名前の管理を行っているもののこと．ノード同士が通信をするためには，マスターが立ち上がっている必要がある．マスターを立ち上げるためのコマンドは roscore.（「5.5 ノードの実行」のところで再度でてくる．）

5. ROS の使用手順

5.1 ワークスペースの作成

ROS には，ワークスペースをつくるためのコマンドは存在しない．したがって，Linux のコマンドを使用して，ワークスペースとなるディレクトリを作成する．例えば，”catkin_ws” という名前のワークスペースを作成する場合，以下のコマンドを実行する．

```
-------------------------------------------------
mkdir -p catkin_ws/src
-------------------------------------------------
```

mkdir はディレクトリを作成するためのコマンドで，-p をつけることで複数階層のディレクトリを 1 度に作成できる．今回の場合，catkin_ws の中にさらに src を作成した．

また，その後に以下のコマンドを実行して，エラーが出ずにビルドできることを確認してみよう．

```
-------------------------------------------------
cd catkin_ws
catkin build
source devel/setup.bash
```

---
cd はディレクトリを移動するためのコマンドで，上記の例では catkin_ws の中に移動している．catkin build はビルドを行うためのコマンドで，ワークスペース内のプログラムやファイルの内容を ROS に解釈させる(読み込ませる)．プログラムの構文やファイルに誤りがある場合はエラーが出る．ワークスペース内のファイルに変更を加えた場合は，プログラムを実行する前に必ず catkin build を実行し，エラーが出ないことを確認する癖をつけよう．もしエラーが出た場合はまず source devel/setup.bash のコマンドを実行してみて，改善しないときはエラーコードで検索をかけるなどして調べるか，先輩に聞いてみよう．

5.2 パッケージの作成

ROS にはパッケージを作成するためのコマンドが用意されている．例えば，”beginner_tutorials”という名前のパッケージを作成する場合，cd コマンドでワークスペース内の src の中まで移動してから，以下のコマンドを実行する．

```
---------------------------------------------------------------
catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
---------------------------------------------------------------
```

5.3 メッセージの作成

ここでは，オリジナルのメッセージファイル(カスタムメッセージ)を作成してみよう．cd コマンドで, 5.2 で作成したパッケージの中まで移動してから, 以下のコマンドを実行する．

```
-----------------------------------------------
mkdir msg
echo "int64 num" > msg/Num.msg
-----------------------------------------------
```

mkdir コマンドで msg というディレクトリを作成している．次に echo コマンドで msg の中に Num.msg というメッセージファイルを作成すると同時に，Num.msg に”int64 num”という内容を書き込んでいる．”int64 num”とは 64bit 長の整数型の num という名前の変数を表している．つまり，Num.msg は上記の変数 num だけをもつメッセージということになる．

これから package.xml や CMakeLists.txt の中身を変更したり，プログラムを書いたりしていくが，この際必ず**半角**を使うこと．特に全角のスペースなどは入れないようにする．

メッセージファイル自体は作成できたが，次は作成したメッセージファイルをパッケージに認識させる必要がある．そこでパッケージ内にある package.xml を開いて図 3 の行があることを確認し，図 4 のようにコメントを解除した後，保存する．

```
  <!-- The *depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use depend as a shortcut for packages that are both build and exec dependencies -->
  <!--   <depend>roscpp</depend> -->
  <!--   Note that this is equivalent to the following: -->
  <!--   <build_depend>roscpp</build_depend> -->
  <!--   <exec_depend>roscpp</exec_depend> -->
  <!-- Use build_depend for packages you need at compile time: -->
ココ➡<!--   <build_depend>message_generation</build_depend> -->
  <!-- Use build_export_depend for packages you need in order to build against this package:
  <!--   <build_export_depend>message_generation</build_export_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use exec_depend for packages you need at runtime: -->
ココ➡<!--   <exec_depend>message_runtime</exec_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <!-- Use doc_depend for packages you need only for building documentation: -->
  <!--   <doc_depend>doxygen</doc_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
```

図 3　メッセージ作成時の package.xml 変更前

```
  <!-- The *depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use depend as a shortcut for packages that are both build and exec dependencies -->
  <!--   <depend>roscpp</depend> -->
  <!--   Note that this is equivalent to the following: -->
  <!--   <build_depend>roscpp</build_depend> -->
  <!--   <exec_depend>roscpp</exec_depend> -->
  <!-- Use build_depend for packages you need at compile time: -->
ココ➡<build_depend>message_generation</build_depend>
  <!-- Use build_export_depend for packages you need in order to build against this package:
  <!--   <build_export_depend>message_generation</build_export_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use exec_depend for packages you need at runtime: -->
ココ➡<exec_depend>message_runtime</exec_depend>
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <!-- Use doc_depend for packages you need only for building documentation: -->
  <!--   <doc_depend>doxygen</doc_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
```

図 4　メッセージ作成時の package.xml 変更後

次に CMakeLists.txt を開いて図 5 の行を探し，図 6 のように変更した後，保存する．

```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)

## System dependencies are found with CMake's conventions
# find_package(Boost REQUIRED COMPONENTS system)


## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

図 5　メッセージ作成時の CMakeLists.txt の変更前

```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)

## System dependencies are found with CMake's conventions
# find_package(Boost REQUIRED COMPONENTS system)


## Generate messages in the 'msg' folder
add_message_files(
  FILES
  Num.msg
)

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

図 6　メッセージ作成時の CMakeLists.txt 変更後

5.4 ノードの作成

今回は図 2 のように，Publisher と Subscriber を一つずつ作って通信させてみる．

まず，cd コマンドでパッケージ内の src に移動した後，以下のコマンドを実行して Publisher と Subscriber の.cpp ファイルを作成しておく．（ノードは Python で書くこともできるが，ここでは C++で書くものとして説明する．）

---

touch talker.cpp
touch listener.cpp

---

touch はファイル作成を行うコマンド．

この後，talker.cpp を Publisher，listener.cpp を Subscriber としてプログラムを書き込んでいく．

5.4.1 Publisher ノードの作成

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

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

  ros::init(argc, argv, "talker");

  ros::NodeHandle n;

  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok()) {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
```

```
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

上記のプログラムを talker.cpp に書き込み，保存する．

以下はソースコードの解説.

```
#include "ros/ros.h"
```

ros/ros.h は ROS のノードを書くときに，使用するヘッダファイル．ROS のノードを書くときはとりあえず，このヘッダファイルを include する．

```
#include "std_msgs/String.h"
```

std_msgs は，既存で用意されたメッセージファイルがあるディレクトリのこと．String.h で，文字列を通信できる既存のメッセージファイル String.msg を読み込んでいる．

```
ros::init(argc, argv, "talker");
```

ROS の初期化を行っている．また第 3 引数にノード名を渡す．マスターは，このノード名によって各ノードを区別しているため，名前が他のノードと被ってはいけない．

```
ros::NodeHandle n;
```

ノードハンドル n を宣言している．この資料で扱うコードでは，Publish，Subscribe に関する情報をマスターに伝えるコードを書くために必要．（ノードハンドルについては筆者もこれ以上のことは理解していません．ごめんなさい m(_ _)m）

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

「chatter_pub という名前の Publisher は，chatter というトピックに，メッセージファイルとして std_msgs の中の String.msg を使って，メッセージを送る」 という内容をマスターに伝えるためのコード．非常に複雑であるため，以下に構文を示す．

ros::Publisher Publisher の名前 = ノードハンドル名.advertise<メッセージファイル名>(“トピック名”, 1000);

第 2 引数の 1000 は，捨てずにとっておける情報量を表すが，基本的にはこの値は 1000 にしておけばよい．（少なくとも筆者は 1000 以外にしたことはない．）

```
ros::Rate loop_rate(10);
```

下の while 文のループ間隔が，1 秒間に 10 回になるように設定している．

```
while (ros::ok()) {
```

この while 文は ROS が起動している間，ループし続ける．

```
std_msgs::String msg;


std::stringstream ss;
```

String メッセージを扱う変数 msg と，文字列を扱う変数 ss を宣言している．(stringstream は，sstream を include した際に使える C++のクラスで，ROS のクラスではない．)

```
ss << "hello world " << count;
msg.data = ss.str();
```

”hello world ”という文字列と，count の値の文字列をつなげた文字列を ss に代入し，さらにそれを msg.data に代入している．通常，std_msgs に用意されたメッセージのデータには，.data をつけることでアクセスできる．なお，「5.3 メッセージの作成」のところで作成した Num.msg のデータにアクセスするには，.num をつける．(ss.str()は，ss に格納された文字列を取り出している．stringstream の変数は，=だけでは代入できないことに注意．)

```
ROS_INFO("%s", msg.data.c_str());
```

ROS_INFO は，printf とほぼ同様のはたらきをする関数．(ただし，msg.data の中のデータ型は stringstream なので，.c_str()をつける必要がある．)

```
chatter_pub.publish(msg);
```

ここで，実際に Publish を行っている．

```
ros::spinOnce();
```

コールバック関数を呼び出しているが，このコードにおいては，あってもなくても影響はない．詳しくは「5.4.2 Subscriber の作成」にて．

```
loop_rate.sleep();
```

ros::Rate loop_rate で指定した時間でループできるように，残った時間待機する．

5.4.2 Subscriber の作成

```
#include "ros/ros.h"
#include "std_msgs/String.h"


void chatterCallback(const std_msgs::String::ConstPtr& msg) {
   ROS_INFO("I heard: [%s]", msg->data.c_str());
}


int main(int argc, char **argv) {
```

```
  ros::init(argc, argv, "listener");

  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  ros::spin();

  return 0;
}
```

上記のプログラムを listener.cpp に書き込み，保存する．

以下，ソースコードの解説．なお，「5.4.1 Publisher ノードの作成」と重複する箇所は省略する．

```
void chatterCallback(const std_msgs::String::ConstPtr& msg) {
```

これは，トピックに新しいメッセージが届くと呼び出される関数で，このような関数を一般にコールバック関数という．引数の与え方が複雑であるため，以下に構文を示す．

void コールバック関数名(const メッセージファイル名& メッセージ変数名) {

なお，メッセージ変数名はコールバック関数内でのみ使用される変数で，自由に定義できる．(今回 Subscribe するデータ型が stringstream であるため，メッセージファイル名の後ろに ConstPtr をつけることでデータ型の調整を行っている．)

```
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

「sub という名前の Subscriber は，chatter というトピックからメッセージを受け取り，その後に chatterCallback という関数を実行する．」 という内容をマスターに伝えるためのコード．非常に複雑であるため，以下に構文を示す．

ros::Subscriber Subscriber の名前 = ノードハンドル名.subscribe("トピック名", 1000, コールバック関数名);

第 2 引数の 1000 は，捨てずにとっておける情報量を表すが，基本的にはこの値は 1000 にしておけばよい．(少なくとも筆者は 1000 以外にしたことはない．)

```
ros::spin();
```

ここで，トピックにメッセージが届いているかを確認して，もし届いていたら適切なコールバック関数を呼び出す，ということをする．ros::spinOnce()との違いは，ros::spin()は ROS が起動している間，上記の処理をし続けるが，ros::spinOnce()は，一度だけ上記の処理をすることである．

次に CMakeLists.txt を開き，末尾に以下を追加して，保存する．

```
-----------------------------------------------------------------
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
-----------------------------------------------------------------
```

最後にワークスペース内で，catkin build と source devel/setup.bash を実行する．

```
-------------------------------------------------
catkin build
source devel/setup.bash
-------------------------------------------------
```

5.5 ノードの実行

まず，ターミナルを 4 つ立ち上げる．
1 つ目のターミナルに以下のコマンドを実行する．

```
-------------------------------------------------
roscore
-------------------------------------------------
```

roscore は，マスターを立ち上げるコマンドで，rosrun（後述）を実行するために必要．

2 つ目のターミナルに以下のコマンドを実行する．

```
-------------------------------------------------
rosrun beginner_tutorials talker
-------------------------------------------------
```

rosrun はノードを立ち上げるためのコマンドで，Publisher である talker を立ち上げた．構文は，“rosrun パッケージ名 ノード名”

3 つ目のターミナルに以下のコマンドを実行する.

```
rosrun beginner_tutorials listener
```

Subscriber である listener を立ち上げた.

4 つ目に以下のコマンドを実行する.

```
rosrun rqt_graph rqt_graph
```

以上 4 つのコマンドを実行した結果, 図 2 で示したのと同様のウィンドウが立ち上がれば, 成功.