ROSの基本操作

Updated on: 2019-11-12

トップページ

基本的なROS上で動くプログラムの書き方とコンパイル方法を学習します。

基本的な用語

パッケージ
　ノードや設定ファイル、コンパイル方法などをまとめたもの
ノード
　ROSの枠組みを利用して動作する実行ファイル
メッセージ
　ノード間でやりとりするデータ
トピック
　ノード間でメッセージをやりとりする際にメッセージを格納する場所

ノード、メッセージ、トピックの関係は以下の図のように表せます。

基本的に、ソフトウェアとしてのROSはノード間のデータのやりとりをサポートするための枠組みです。
加えて、使い回しがきく汎用的なノードを世界中のROS利用者で共有するコミュニティも大きな意味でのROSの一部となっています。

ソースコードを格納するワークスペース

ROSでは、プログラムをビルドする際にcatkinというソフトウェアパッケージを使用しています。
また、catkin は、 cmake というソフトウェアを使っており、ROS用のプログラムのパッケージ毎にcmakeの設定ファイルを作成することで、 ビルドに必要な設定を行います。

以下の手順で本作業用の新しいワークスペースを作ります。

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
Creating symlink "/home/[ユーザ名]/catkin/src/CMakeLists.txt"
    pointing to "/opt/ros/kinetic/share/catkin/cmake/toplevel.cmake"
$ ls
CMakeLists.txt
```

/

```
$ cd ..
$ ls
src
$
```

catkin_wsディレクトリ内にある、build、develは、catkinシステムがプログラムをビルドする際に使用するものなので、 ユーザが触る必要はありません。
catkin_ws/srcディレクトリは、ROSパッケージのソースコードを置く場所で、中にあるCMakeLists.txt は、 ワークスペース全体をビルドするためのルールが書かれているファイルです。

このディレクトリに、本作業用のパッケージをダウンロードします。

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/gbiggs/rsj_tutorial_2017_ros_intro.git
$ ls
CMakeLists.txt  rsj_tutorial_2017_ros_intro
$
```

gitは、ソースコードなどの変更履歴を記録して管理する分散型バージョン管理システムと呼ばれるものです。
今回のセミナーでは詳細は触れませんが、研究開発を行う上では非常に有用なシステムですので、利用をお勧めします。
公式の解説書、Pro Gitなどを参考にして下さい。

GitHubは、ソースコードなどを格納、保管、管理するためのWebリポジトリサービスです。
オープンソースソフトウェアの開発、共同作業および配布のためによく利用されており、 ROSではソースコードの保存と配布する場所としてもっとも人気なサービスとなっています。
バイナリパッケージとして配布されているROSパッケージ以外の利用をする場合、GitHubを利用します。
URLが分かれば上の手順だけで簡単にROSのパッケージが自分のワークスペースにインポートし利用することができます。

では、次にパッケージのディレクトリ構成を確認します。
ダウンロードしているパッケージがバージョンアップされている場合などには、下記の実行例とファイル名が異なったり、ファイルが追加・削除されている場合があります。

```
$ cd ~/catkin_ws/src/rsj_tutorial_2017_ros_intro/
$ ls
CMakeLists.txt  launch  msg  package.xml  src
$ ls launch/
say_hello.launch
$ ls msg/
Greeting.msg
$ ls src/
displayer.cpp  greeter.cpp
$
```

CMakeLists.txtとpackage.xmlには、使っているライブラリの一覧や生成する実行ファイルとC++のソースコードの対応など、 このパッケージをビルドするために必要な情報が書かれています。
launchディレクトリには、複数のノードでできたシステムの定義が、msgディレクトリには、このパッケージ独自のデータ形式の定義が、srcディレクトリには、このパッケージに含まれるプログラム(ノード)のソースコードが含まれています。

catkin_makeコマンドで、ダウンとロードしたrsj_tutorial_2017_ros_introパッケージを含む、ワークスペース全体をビルドします。catkin_makeは、ワークスペースの最上位ディレクトリ(~/catkin_ws/)で行います。

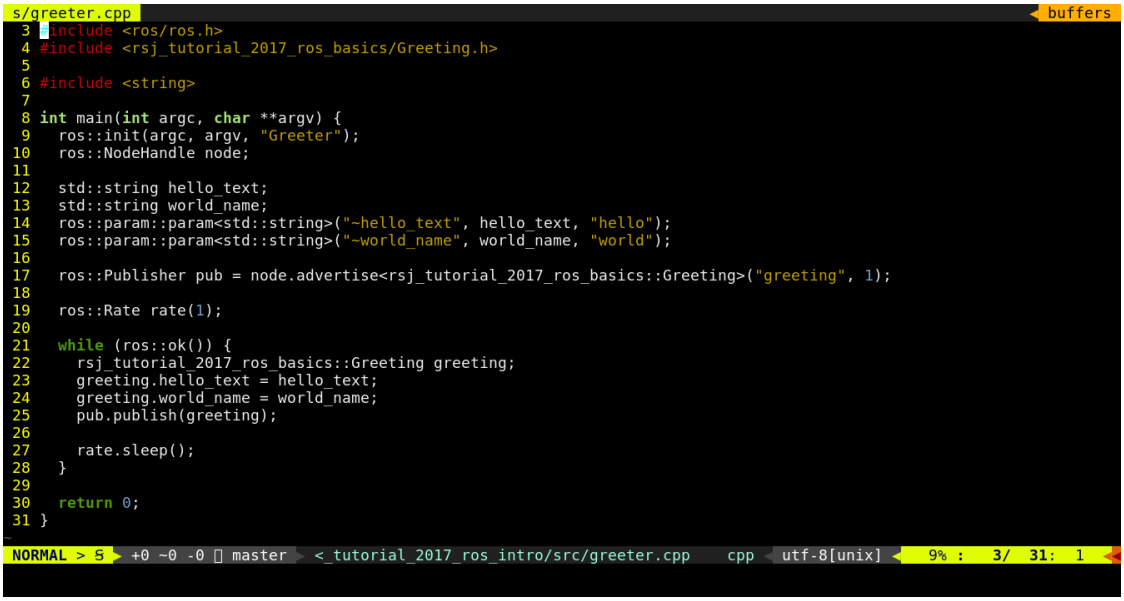
ROSノードの理解とビルド・実行

先作成したワークスペースを利用します。
ターミナルを開き、パッケージが正しく存在しているか確認します。

```
$ cd ~/catkin_ws/src/
$ ls
CMakeLists.txt  rsj_tutorial_2017_ros_intro
$ cd ..
$
```

ソースファイルの編集にはお好みのテキストエディタが利用可能です。
Linuxでのプログラム開発がはじめての方には、Ubuntuにデフォルトでインストールされているgeditがおすすめです。

お好みのテキストエディタで~/catkin_ws/src/rsj_tutorial_2017_ros_intro/src/greeter.cppを開きます。



送信ノードの作成 （基本的なコードを読み解く）

このコードが実行されたときの流れを確認しましょう。

まず、先頭部分では、必要なヘッダファイルをインクルードしています。

```
1  #include <ros/ros.h>
```

続いて、本ノードが利用するメッセージのヘッダファイルをインクルードしています。

```
1  #include <rsj_tutorial_2017_ros_basics/Greeting.h>
```

`std::string`が利用されるので、ヘッダファイルをインクルードします。

```
1  #include <string>
```

続いて、C++のmain関数が定義されています。
本ノードは非常に簡単な構成としているため、すべての機能をmain関数に入れています。
複雑な機能や色々なデータを持つノードには、クラスとしての実装することをおすすめします。

```
1  int main(int argc, char **argv) {
2    ros::init(argc, argv, "Greeter");
3    ros::NodeHandle node;
4
5    std::string hello_text;
6    std::string world_name;
7    ros::param::param<std::string>("~hello_text", hello_text, "hello");
8    ros::param::param<std::string>("~world_name", world_name, "world");
9
10   ros::Publisher pub =
11     node.advertise<rsj_tutorial_2017_ros_basics::Greeting>("greeting", 1);
12
13   ros::Rate rate(1);
14
15   while (ros::ok()) {
16     ros::spinOnce();
17
18     ROS_INFO("Publishing greeting '%s %s'", hello_text, world_name);
19     rsj_tutorial_2017_ros_basics::Greeting greeting;
20     greeting.hello_text = hello_text;
21     greeting.world_name = world_name;
22     pub.publish(greeting);
23
24     rate.sleep();
25   }
```

```
26
27      return 0;
28    }
```

main関数は、まずノードのセットアップを行います。

ros::initはROSのインフラストラクチャの初期設定を行いノードを初期化します。
1、2番目の引数には、main関数の引数をそのまま渡し、3番目の引数には、このノードの名前(この例では"Greeter")を与えています。

その次にあるros::NodeHandle nodeは、ノードを操るための変数を初期化します。

次の４行はパラメータの初期化です。ROSでは、純粋コマンドラインを利用するよりROSのパラメータ機能を利用することが標準的です。
こうすることで、コマンドラインだけではなくて、roslaunch（複数のノードを起動するためのツール）やGUIツールからもパラメータの設定が簡単にできます。

パラメータの初期化が終わったら、データ送信のためのパブリッシャーを初期化します。
この変数の作成によりトピックが作成され、**このノードからデータの送信が可能になります。**
以下の引数を与えています。

"greeting"
　　トピック名：データをこのトピックに送信する
1
　　メッセージのバッファリング量を指定 (大きくすると、処理が一時的に重くなったときなどに受け取り側の読み飛ばしを減らせる)

advertise関数についている<rsj_tutorial_2017_ros_basics::Greeting>の部分はメッセージの型を指定しています。
これは、幾何的・運動学的な値を扱うメッセージを定義しているrsj_tutorial_2017_ros_basicsパッケージの、並進・回転速度を表すGreeting型です。
(この指定方法は、C++のテンプレートという機能を利用していますが、ここでは「advertiseのときはメッセージの型指定を<>の中に書く」とだけ覚えておけば問題ありません)

セットアップの最後として、ros::Rate rate(1)で周期実行のためのクラスを初期化しています。
初期化時の引数で実行周波数(この例では1 Hz)を指定します。

while(ros::ok())で、メインの無限ループを回します（すなわちこのノードのメーンプロセッシングループです）。
ros::ok()をwhileの条件にすることで、ノードの終了指示が与えられたとき(Ctrl+c が押された場合も含む)には、ループを抜けて終了処理などが行えるようになっています。

ループ中では、まず、*ros::spinOnce()*を呼び出して、*ROS*のメッセージを受け取る といった処理を行います。
spinOnceは、その時点で届いているメッセージの受け取り処理を済ませた後、すぐに処理を返します。
rate.sleep()は、先ほど初期化した実行周波数を維持するようにsleepします。

ros::spinOnce()とrate.sleep()の間に本ノードの処理を入れました。

最初は、ROSでログ情報を画面などに出力する際に用いるROS_INFO()関数を呼び出してメッセージを表示しています。
他にも、ROS_DEBUG()、ROS_WARN()、ROS_ERROR()、ROS_FATAL()などのデバッグログ関数が用意されています。

その後、データを送信します。
まずは送信するデータ型（rsj_tutorial_2017_ros_basics::Greeting）を初期化し、値を設定します。
さきほどセットアップで作成したパラメータの値を利用します。
こうすることで、送信されるデータの内容は実行するときに自由に変更できます。

そして、pub.publish(greeting)によってデータを送信します。
この行でデータはバッファーに入れられ、別のスレッドが自動的にサブスクライバに送信します(ROSのデフォルトでは非同期送信となります）。

main ループが終了すると作成した変数は自動的にクリーンアップを実行し、ノードのシャットダウンを行います。

ビルド＆実行

ROS パッケージをビルドするためには、catkin_makeコマンドを用います。

下記コマンドをターミナルで実行してみましょう。

```
1
2
```
```
$ cd ~/catkin_ws/
$ catkin_make
```

ROSシステムの実行の際、ROSを通してノード同士がデータをやりとりするために用いる「roscore」を起動しておく必要があります。
2つ目のターミナルを開き、それぞれで以下を実行して下さい。

1つ目のターミナルで下記を実行します。

```
$ roscore
```

ROSでワークスペースを利用するとき、ターミナルでそのワークスペースに環境変数などを設定することが必要です。
このためにワークスペースの最上位のディレクトリで`source devel/setup.bash`を実行します。
このコマンドはワークスペースの環境編情報などを利用中のターミナルに読み込みます。
しかし、 ターミナルごとに環境変数はリセットされますので、新しいターミナルでワークスペースを利用しはじめるときには、
まず`source devel/setup.bash`を実行しなければなりません。
一つのターミナルで一回だけ実行すれば十分です。そのターミナルを閉じるまで有効となります。

2つ目のターミナルで下記を実行します。

```
$ cd ~/catkin_ws/
$ source devel/setup.bash
$ rosrun rsj_tutorial_2017_ros_basics greeter
[ INFO] [1494840089.900580884]: Publishing greeting 'hello world'
```

上記のようなログが表示されれば成功です。

ソースコードにパラメータを利用したので、コマンドラインからパラメータ設定を試してみましょう。
ノードを実行した2つ目のターミナル（**注意：roscoreのターミナルではなくて**）に `Ctrl+c` を入力してノードを終了します。
そして以下を実行してください。

```
$ rosrun rsj_tutorial_2017_ros_basics greeter \
  _hello_text:=gidday _world_name:=planet
[ INFO] [1494840247.644756809]: Publishing greeting 'gidday planet'
```

上記のようなログが表示されれば成功です。

実行後は両方のターミナルで `Ctrl+c` でノードと`roscore`を終了します。

受信ノードの作成

前節で作成したノードはメッセージを送信するノードでした。
次にメッセージを受信するノードを作成してみましょう。

以下のソースは`rsj_tutorial_2017_ros_basics/src/displayer.cpp`ファイルにあります。

```
#include <ros/ros.h>
#include <rsj_tutorial_2017_ros_basics/Greeting.h>

#include <iostream>

void callback(const rsj_tutorial_2017_ros_basics::Greeting::ConstPtr &msg) {
  std::cout << msg->hello_text << " " << msg->world_name << '\n';
}
int main(int argc, char **argv) {
  ros::init(argc, argv, "Displayer");
  ros::NodeHandle node;

  ros::Subscriber sub = node.subscribe("greeting", 10, callback);

  ros::spin();

  return 0;
}
```

本ノードは`greeting`というトピックから取得したデータをターミナルに表示します。
`greeter`からの差は以下のようです。

まずは`callback`関数です。
この関数はトピックのデータ型に合っているポインタを引数としてもらいます。
トピックからデータを受け取ったら、`callback`関数は呼ばれます。
そのデータを`std::cout`に出力し終了します。

`const rsj_tutorial_2017_ros_basics::Greeting::ConstPtr` は、const型(内容を書き換えられない)、`rsj_tutorial_2017_ros_basics`パッケージに含まれる、`Greeting`型のメッセージの、const型ポインタを表しています。
`&msg`の`&`は、参照型(内容を書き換えられるように変数を渡すことができる)という意味ですが、(const型なので)ここでは特に気にする必要はありません。
msgはクラスへのポインタなので「->」を用い、以降はクラスのメンバ変数へのアクセスなので「.」を用いてアクセスしています。

`main`関数内にパラメータの初期化はなくなりました。本ノードはパラメータを利用しません。

`ros::Publisher`の初期化もなくなり、代わりに`ros::Subscriber`を初期化します。
この行はトピックへのアクセスを初期化し、データが取得したらの対応を示します。
引数は以下です。

`"greeting"`
　トピック名
`10`
　バッファーサイズ
`callback`
　メッセージを受け取ったときに呼び出す関数を指定 (`callback`関数)

最後に、main ループでの処理は不要となります。
本ノードはデータが届くとき以外何もしないので、無限ループになる`ros::spin()`を呼びます。
`ros::spin()`は`greeter`の`while(...)`と`ros::spinOnce()`と類似の機能を中で持つので、ノードがシャットダウンされるまでに戻りません。

ビルド＆実行

作成したノードを実行してみましょう。
1つ目のターミナルで以下を実行します。

```
1  $ roscore
```

そして2つ目のターミナルで以下を実行します。

```
1  $ cd ~/catkin_ws/
2  $ source devel/setup.bash
3  $ rosrun rsj_tutorial_2017_ros_basics greeter
4  [ INFO] [1494840089.900580884]: Publishing greeting 'hello world'
```

最後に、3番目のターミナルを開いて、下記を実行します。

```
1  $ cd ~/catkin_ws/
2  $ source devel/setup.bash
3  $ rosrun rsj_tutorial_2017_ros_basics displayer
```

「hello world」と表示されれば成功です。

以上の手順で、ROSパッケージに含まれるノードのソースコードを編集し、ビルドして、実行できるようになりました。

システムとして実行する

`roslaunch`を利用して、複数のノードでできたシステムを開始、終了することができます。

手動でノードを１ノードずつ起動することは面倒ですし、ミスを誘発する可能性も高まります。
そのためにROSに`roslaunch`というツールがあります。
`roslaunch`を利用すると、システム全体をまとめて起動し、状況をモニターし、そしてまとめて終了することが可能となります。

launchファイルを読み解く

launchファイルは、ノードやパラメータの組み合わせを定義するためのファイルです。
フォーマットはXMLです。
システムに含まれるノードとそのノードの起動方法を一つずつ定義します。

`rsj_tutorial_2017_ros_basics`パッケージに以下のファイルが`launch/say_hello.launch`として存在します。
このファイルはさきほど手動で起動したシステムを定義します。

```
1  <launch>
```

```
  <node name="greeter" pkg="rsj_tutorial_2017_ros_basics" type="greeter">
    <param name="hello_text" value="allo"/>
    <param name="world_name" value="earth"/>
  </node>

  <node name="displayer"
        pkg="rsj_tutorial_2017_ros_basics"
        type="displayer" output="screen"/>
</launch>
```

nodeタグは２つあります。
属性は下記の通りです。

`name`
　ノードインスタンスの名
`pkg`
　ノードを定義するパッケージ名
`type`
　ノードの実行ファイル名
`output`
　stdoutの先：定義しないとstdout（ROS_INFOやstd::coutへの出力等）はターミナルで表示されず、~/.ros/log/に保存されるログファイルだけに出力される。

１番目の<node>はgreeterノードの定義です。
本要素の中でパラメータの設定も行っています。
なお、パラメータの設定を行わない場合はノードのソースに定義したディフォルト値が利用されるので、記述は必須ではありません。

２番目の<node>はdisplayerノードの定義です。
パラメータはありませんが、出力されることをターミナルで表示するようにします。

roslaunchでシステムを起動

開いているターミナルにroscoreや起動中のノードをすべて Ctrl+c で停止します。
その後、いずれか１つのターミナルで以下を実行します。

```
$ cd ~/catkin_ws
$ source devel/setup.bash
$ roslaunch rsj_tutorial_2017_ros_basics say_hello.launch
... logging to /home/geoff/.ros/log/40887b56-395c-11e7-b8
68-d8cb8ae35bff/roslaunch-alnilam-11087.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://alnilam:40672/

SUMMARY
========

PARAMETERS
 * /greeter/hello_text: allo
 * /greeter/world_name: earth
 * /rosdistro: kinetic
 * /rosversion: 1.12.7

NODES
  /
    displayer (rsj_tutorial_2017_ros_basics/displayer)
    greeter (rsj_tutorial_2017_ros_basics/greeter)

auto-starting new master
process[master]: started with pid [11098]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 40887b56-395c-11e7-b868-d8cb8ae35bff
process[rosout-1]: started with pid [11111]
started core service [/rosout]
process[greeter-2]: started with pid [11118]
process[displayer-3]: started with pid [11126]
allo earth
allo earth
```

「allo earth」が繰り返して表示されたら成功です。

Ctrl+c でシステムを停止します。

```
allo earth
allo earth
allo earth
[Ctrl+c]
^C[displayer-3] killing on exit
[greeter-2] killing on exit
[rosout-1] killing on exit
[master] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
$
```

これでシステムの起動、停止が簡単にできるようになりました。

roslaunchを利用する場合は、別のターミナルでのroscoreの実行は不要です。
roslaunchは中でroscoreを起動したり停止したりします。

ROSノードの作成

パッケージとノードを自分で作成しマニピュレータのサーボモータを操作します。

パッケージの作成

ワークスペースに新しいパッケージを作成するために、以下を実行してください。

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg rsj_2017_servo_control roscpp dynamixel_controllers \
  dynamixel_msgs
Created file rsj_2017_servo_control/CMakeLists.txt
Created file rsj_2017_servo_control/package.xml
Created folder rsj_2017_servo_control/include/rsj_2017_servo_control
Created folder rsj_2017_servo_control/src
Successfully created files in /home/geoff/catkin_ws/src/rsj_2017_servo_control.
Please adjust the values in package.xml.
```

引数の１番目はパッケージ名です。
２番目と３番目は依存パッケージの定義です。
今回はC++で記述するため、roscppに依存します。
そしてDynamixelのサーボを利用するのでハードウェアとインターフェースするdynamixel-controllersに依存します。
また、サーボコントローラにコマンドを送るためにDynamixel用のメッセージタイプの利用が必要なのでdynamixel-msgsにも依存します。

生成されたパッケージの中身を確認します。

```
$ cd rsj_2017_servo_control/
$ ls
CMakeLists.txt  include  package.xml  src
```

以前と同様に、package.xmlはパッケージの情報を定義し、CMakeLists.txtはパッケージのビルド方法を定義します。

package.xmlをエディターで開くと以下の行が含まれていると見えます。

```
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>dynamixel_controllers</build_depend>
  <build_depend>dynamixel_msgs</build_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>dynamixel_controllers</run_depend>
  <run_depend>dynamixel_msgs</run_depend>
  <run_depend>roscpp</run_depend>
```

これらはパッケージの依存を定義します。ROSとcatkinはこれらの利用によってパッケージのビルド順番等を決めます。

普段はpackage.xmlのメール等（<maintainer email>タグ等）も編集するべきですが、今回は時間のために省略します。

以上、パッケージの作成ができました。

ノードを作成

作成したパッケージにノードを作成します。

ノードを作成するために以下の手順を行います。

1. CMakeLists.txtにノードのコンパイル方法を追加する
2. ノードのソースファイルを作成する

CMakeLists.txtにノードを追加

rsj_2017_servo_controlパッケージにある
CMakeLists.txt（~/catkin_ws/src/rsj_2017_servo_control/CMakeLists.txt）をエディタで開き、以下のようにソースを編集します。
5行目、12行目、17行目および20行目を追加しました。
ファイル内の133行目ぐらいから始まります。
インストールされたcatkinのバージョンにより、編集する具体的な行番号が変化する可能性があります。

```
1  ## Declare a C++ executable
2  ## With catkin_make all packages are built within a single CMake context
3  ## The recommended prefix ensures that target names across packages don't collide
4  # add_executable(${PROJECT_NAME}_node src/rsj_2017_servo_control_node.cpp)
5  add_executable(${PROJECT_NAME}_set_servo_pos src/set_servo_pos.cpp)
6
7  ## Rename C++ executable without prefix
8  ## The above recommended prefix causes long target names, the following renames the
9  ## target back to the shorter version for ease of user use
10 ## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
11 # set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")
12 set_target_properties(${PROJECT_NAME}_set_servo_pos
13                       PROPERTIES OUTPUT_NAME set_servo_pos PREFIX "")
14
15 ## Add cmake target dependencies of the executable
16 ## same as for the library above
17 # add_dependencies(${PROJECT_NAME}_node
18 #                  ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
19 add_dependencies(${PROJECT_NAME}_set_servo_pos
20                  ${${PROJECT_NAME}_EXPORTED_TARGETS}
21                  ${catkin_EXPORTED_TARGETS})
22
23 ## Specify libraries to link a library or executable target against
24 target_link_libraries(${PROJECT_NAME}_set_servo_pos ${catkin_LIBRARIES})
```

なお、**必ず** ファイルトップにあるadd_compile_options(-std=c++11)の行をコメントアウトしてください。

これでcatkinにset_servo_posというノードのコンパイルを指定できました。

ノードのソースの作成

rsj_2017_servo_controlパッケージ内のsrc/ディレクトリにset_servo_pos.cppというファイル
（~/catkin_ws/src/rsj_2017_servo_control/src/set_servo_pos.cpp）を作成します。
そしてエディタでset_servo_pos.cppを開き、以下のソースを入力します。

```
1  #include <ros/ros.h>
2  #include <std_msgs/Float64.h>
3
4  #include <string>
5  #include <vector>
6
7  int main(int argc, char **argv) {
8    ros::init(argc, argv, "set_servo_pos");
9    ros::NodeHandle node;
10
11   std::string servo_command_topic;
12   ros::param::param<std::string>(
```

```
    "~servo_command_topic",
    servo_command_topic,
    "/finger_servo_controller/command");

  ros::Publisher pub = node.advertise<std_msgs::Float64>(servo_command_topic, 1);

  std::vector<std::string> my_args;
  ros::removeROSArgs(argc, argv, my_args);
  if (my_args.size() < 2) {
    ROS_FATAL("Usage: rosrun set_servo_pos [position]");
    ros::shutdown();
    return 1;
  }
  std_msgs::Float64 servo_pos;
  servo_pos.data = std::stof(my_args[1]);

  ROS_INFO("Setting servo position to %f", servo_pos.data);
  ros::Rate r(1);
  while (ros::ok()) {
    pub.publish(servo_pos);
    ros::spinOnce();
    r.sleep();
  }

  return 0;
}
```

このソースはコマンドラインから指定のサーボモータの位置を読み、パラメータで指定されたトピック（デフォルトでは/finger_servo_controller/command）にパブリッシュします。
このトピックはサーボコントローラがサブスクライブしています。

なぜサーボの位置はパラメータとしてノードに渡さなかったでしょうか。
パラメータはいわゆる「configuration」です。
ノードの振る舞いを制御するためのことです。
サーボのトピックはノードの振る舞いですが、サーボの位置はコマンドです。
このノードはコマンドラインで利用するべきなツールなので普通のコマンドラインパラメータを利用しました。

ビルド＆実行

パッケージ内のソースを変更・追加した後、必ず再ビルドが必要です。
ターミナルでcatkin_makeコマンドを実行して再ビルドします。

1つ目のターミナル：

```
$ cd ~/catkin_ws/
$ catkin_make
```

この際、ビルドエラーが出力されていないかよく確認して下さい。
エラーが出ている場合は、ソースコードの該当箇所を確認・修正して下さい。

実行する前にまずはマニピュレータが壊れないようにします。
マニピュレータのグリッパーが動くので、**電源を入れる前にマニピュレータのグリッパーは何にもぶつからないような姿勢にしましょう** 。

マニピュレータのサーボコントローラを起動することが必要です。

rsj_2017_servo_controlパッケージの中に以下のファイルを作成します。
~/catkin_ws/src/rsj_2017_servo_control/configと~/catkin_ws/src/rsj_2017_servo_control/launchディレクトリは作成した上でファイルを編集します。

```
$ cd ~/catkin_ws/src/rsj_2017_servo_control
$ mkdir config
$ mkdir launch
```

~/catkin_ws/src/rsj_2017_servo_control/config/dynamixel_test.yaml:

```
finger_servo_controller:
    controller:
        package: dynamixel_controllers
```

```
            module: joint_position_controller
            type: JointPositionController
        joint_name: finger_joint
        joint_speed: 1.17
        motor:
            id: 5
            init: 512
            min: 0
            max: 1023
```

~/catkin_ws/src/rsj_2017_servo_control/launch/dynamixel_test.launch:

```
<launch>
    <node name="dynamixel_manager" pkg="dynamixel_controllers"
      type="controller_manager.py" required="true" output="screen">
        <rosparam>
            namespace: dynamixel_controller_manager
            serial_ports:
                dxl_tty1:
                    port_name: "/dev/ttyUSB0"
                    baud_rate: 1000000
                    min_motor_id: 1
                    max_motor_id: 5
                    update_rate: 10
        </rosparam>
    </node>
    <rosparam file="$(find rsj_2017_servo_control)/config/dynamixel_test.yaml"
          command="load"/>
    <node name="finger_servo_spawner"
          pkg="dynamixel_controllers"
          type="controller_spawner.py"
          args="--manager=dynamixel_controller_manager
                --port dxl_tty1
                finger_servo_controller"
          output="screen"/>
</launch>
```

1つ目のターミナルで以下を実行し、マニピュレータのグリッパーサーボコントローラを起動します。

```
$ roslaunch rsj_2017_servo_control dynamixel_test.launch
... logging to /home/geoff/.ros/log/619c447c-396a-11e7-b868-d8cb8ae35bff/
roslaunch-alnilam-1790.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://alnilam:44912/

SUMMARY
========
（省略）
```

2つ目のターミナルで以下のコマンドを実行します。

```
$ rosrun rsj_2017_servo_control set_servo_pos 0
[ INFO] [1494851539.189274395]: Setting servo position to 0.000000
[Ctrl+cで止める]
$ rosrun rsj_2017_servo_control set_servo_pos -0.5
[ INFO] [1494851548.085785357]: Setting servo position to -0.500000
[Ctrl+cで止める]
```

サーボモータが動けば、サーボモータ制御は成功しています。

このソースは以下のURLでダウンロード可能です。

https://github.com/gbiggs/rsj_2017_servo_control

サーボの状態を確認

もう一つのノードを作成し、サーボモータの現在状況をターミナルで表示します。
上記と同じ手順でservo_statusというノードをrsj_2017_servo_controlパッケージに追加します。

ノードを作成

CMakeLists.txtにノードを追加

~/catkin_ws/src/rsj_2017_servo_control/CMakeLists.txtを編集して、新しいノードのコンパイル方法を指定します。
ファイルをエディターで開き以下の通りになるように編集します。
（6行目、14行目、20行目および24行目を追加しました。）

```
## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
# add_executable(${PROJECT_NAME}_node src/rsj_2017_servo_control_node.cpp)
add_executable(${PROJECT_NAME}_set_servo_pos src/set_servo_pos.cpp)
add_executable(${PROJECT_NAME}_servo_status src/servo_status.cpp)

## Rename C++ executable without prefix
## The above recommended prefix causes long target names, the following renames the
## target back to the shorter version for ease of user use
## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
# set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")
set_target_properties(${PROJECT_NAME}_set_servo_pos
                      PROPERTIES OUTPUT_NAME set_servo_pos PREFIX "")
set_target_properties(${PROJECT_NAME}_servo_status
                      PROPERTIES OUTPUT_NAME servo_status PREFIX "")

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(${PROJECT_NAME}_node
                   ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
add_dependencies(${PROJECT_NAME}_set_servo_pos
                 ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
add_dependencies(${PROJECT_NAME}_servo_status
                 ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

## Specify libraries to link a library or executable target against
target_link_libraries(${PROJECT_NAME}_set_servo_pos ${catkin_LIBRARIES})
target_link_libraries(${PROJECT_NAME}_servo_status ${catkin_LIBRARIES})
```

ノードのソースの作成

rsj_2017_servo_controlパッケージ内のsrc/ディレクトリにservo_status.cppというファイルを作成します。
エディター~/catkin_ws/src/rsj_2017_servo_control/src/servo_status.cppを新規作成し、以下のソースを入力します。

```
#include <ros/ros.h>
#include <dynamixel_msgs/JointState.h>

#include <string>

void callback(const dynamixel_msgs::JointState::ConstPtr &msg) {
  ROS_INFO("--- Servo status ---");
  ROS_INFO("Name: %s", msg->name.c_str());
  ROS_INFO("ID: %d", msg->motor_ids[0]);
  ROS_INFO("Temperature: %d", msg->motor_temps[0]);
  ROS_INFO("Goal position: %f", msg->goal_pos);
  ROS_INFO("Current position: %f", msg->current_pos);
  ROS_INFO("Position error: %f", msg->error);
  ROS_INFO("Velocity: %f", msg->velocity);
  ROS_INFO("Load: %f", msg->load);
  ROS_INFO("Moving: %s", msg->is_moving ? "yes" : "no");
  ROS_INFO(" ");
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "servo_status");
```

```
  ros::NodeHandle node;

  std::string servo_status_topic;
  ros::param::param<std::string>(
    "~servo_status_topic",
    servo_status_topic,
    "/finger_servo_controller/state");

  ros::Subscriber pub = node.subscribe<dynamixel_msgs::JointState>(
    servo_status_topic,
    10,
    callback);

  ros::spin();

  return 0;
}
```

コールバックの中にdynamixel_msgs::JointStateというメッセージタイプを利用します。
このメッセージタイプはdynamixel_msgsパッケージに定義され、内容は以下のようになっています。

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string name
int32[] motor_ids
int32[] motor_temps
float64 goal_pos
float64 current_pos
float64 error
float64 velocity
float64 load
bool is_moving
```

ビルド＆実行

コンパイルして実行します。
1つ目のターミナルでcatkin_makeを実行します。

```
$ cd ~/catkin_ws/
$ catkin_make
```

実行する前にマニピュレータのサーボコントローラを起動することが必要です。
1つ目のターミナルで下記を実行してマニピュレータのグリッパーサーボコントローラを起動します。

```
$ source devel/setup.bash
$ roslaunch rsj_2017_servo_control dynamixel_test.launch
... logging to /home/geoff/.ros/log/619c447c-396a-11e7-b868-d8cb8ae35bff/
roslaunch-alnilam-1790.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://alnilam:44912/

SUMMARY
========
（省略）
```

2つ目のターミナルで下記を実行します。

```
$ cd ~/catkin_ws/
$ source devel/setup.bash
$ rosrun rsj_2017_servo_control servo_status
```

```
[ INFO] [1494855697.336794278]: --- Servo status ---
[ INFO] [1494855697.336922059]: Name: finger_joint
[ INFO] [1494855697.336968787]: ID: 5
[ INFO] [1494855697.337016662]: Temperature: 37
[ INFO] [1494855697.337069086]: Goal position: 0.000000
[ INFO] [1494855697.337120585]: Current position: 0.000000
[ INFO] [1494855697.337170811]: Position error: 0.000000
[ INFO] [1494855697.337226948]: Velocity: 0.000000
[ INFO] [1494855697.337278499]: Load: 0.000000
[ INFO] [1494855697.337329531]: Moving: no
[ INFO] [1494855697.337372008]:
[ INFO] [1494855697.432684658]: --- Servo status ---
（省略）
```

（サーボモータの動作状況により上記数値が変化することがあります。）

3つ目のターミナルで下記を実行すると、servo_statusのターミナルで数値の変更が確認できます。

```
$ cd ~/catkin_ws/
$ source devel/setup.bash
$ rosrun rsj_2017_servo_control set_servo_pos 0
[ INFO] [1494851539.189274395]: Setting servo position to 0.000000
[Ctrl+cで止める]
$ rosrun rsj_2017_servo_control set_servo_pos -0.5
[ INFO] [1494851548.085785357]: Setting servo position to -0.500000
[Ctrl+cで止める]
```

このソースは以下の*URL*でダウンロード可能です。

https://github.com/gbiggs/rsj_2017_servo_control

課題

1. サーボステータスにerrorとloadとis_movingという値があります。
   これらの利用により、サーボモータがストール(外力が加わり期待通りの回転ができない状態など)したかどうか判断できます。
   サーボモータがストールした場合に警告をターミナルで表示するノードを作成してみましょう。

問い合わせ先：

高橋 三郎（パナソニック アドバンストテクノロジー） (takahashi dot saburo at jp dot panasonic dot com)

長谷川 孔明（豊橋技術科学大学）

This project is maintained by takahasi

Generated on 2019-11-13

Hosted on GitHub Pages — Theme by orderedlist