

ROS 2講習 第一回

4S altair

参考文献:ROS2とPythonで作って学ぶAIロボット入門

環境

- python 3.10
- ThinkPad L380 ubuntu22.04.3tls
- ROS2 humble

ROS 2の基礎知識

ROS 2は多くの実行中のプログラム間で通信してロボットを動かす。このプログラムのことを **ノード (node)** という

つまりロボットを動かすにはROS 2の通信を理解しないといけない。

通信方法

ros2の通信方法には

トピック通信 **サービス通信** **アクション通信** **パラメーター通信**

の4つがある。

1. トピック (Topic)

概要

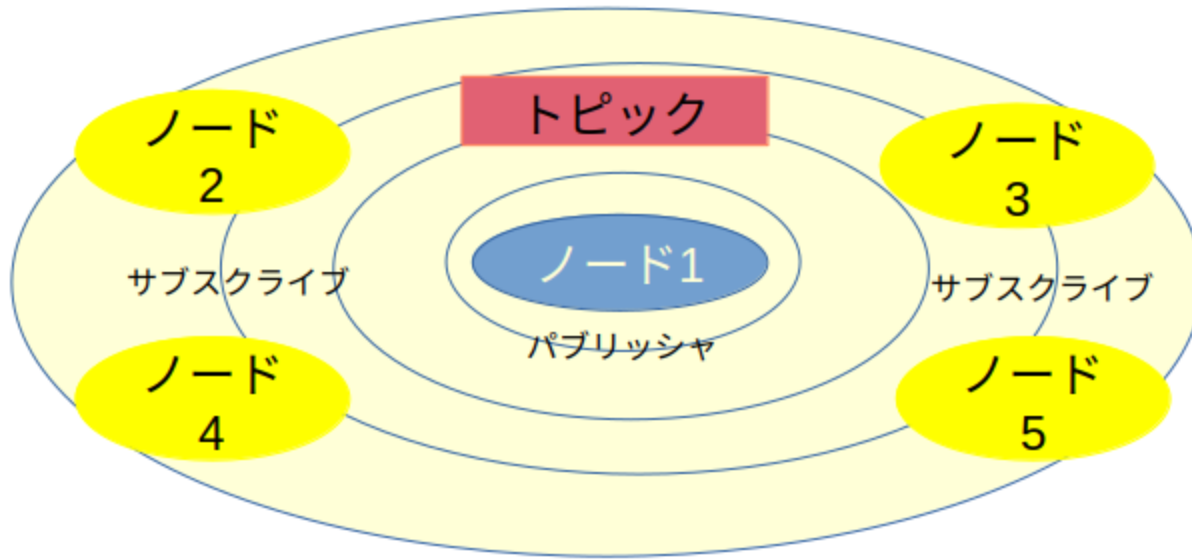
- 1方向・非同期通信

用途

多くのノードに同じデータを送る場合

センサーデータの配信・受信

カメラ, LIDARとか



イメージ：テレビ

テレビ局に該当するデータの送り手を **パブリッシャ(publisher)**
視聴者に該当する受け手を **サブスクライバ(subscriber)** という
番組のチャンネルが **トピック** といいデータを **メッセージ** と呼ぶ

2. サービス (Service)

概要

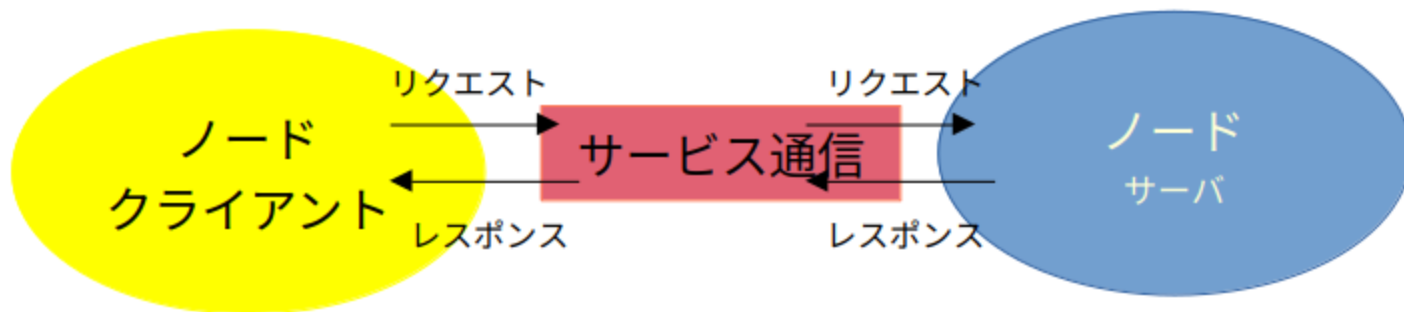
- 双方向・同期通信

用途

すぐに終了するタスク

電源の起動確認，モード切替

カメラ，LIDARなどのセンサーの動作モード切替



イメージ：Google検索

データの送り手を クライアント (client)

そのデータを処理して応答する側を サーバ(server) という。

クライアントからサーバに仕事を依頼することを リクエスト ，そのデータを リクエストメッセージ ，サーバが結果をクライアントに返すことを レスポンス ，そのデータを レスポンスメッセージ と呼ぶ

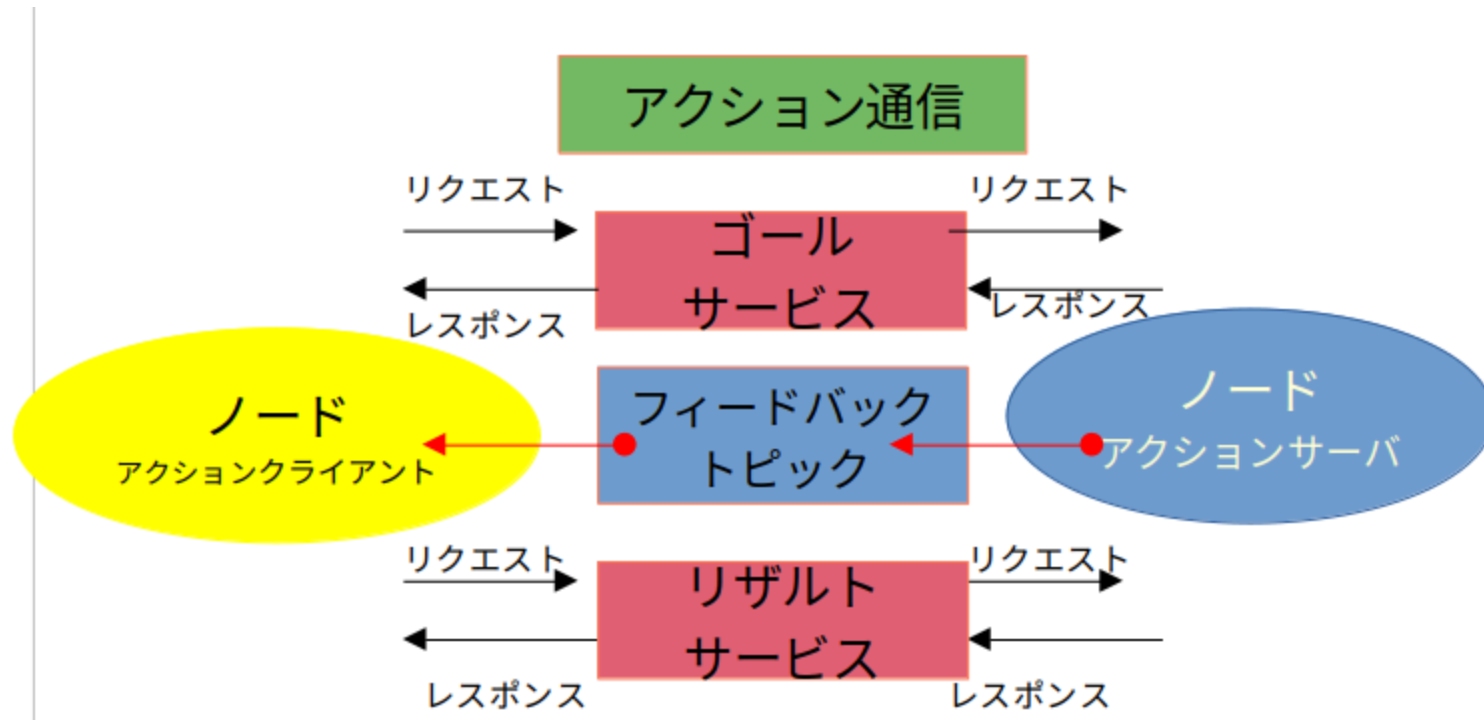
3. アクション (Action)

概要

- 双方向・同期通信，サービス通信より高性能

用途

ナビゲーションなどタスク終了までに時間がかかり，途中経過も知りたい複雑なタスクのとき



サービスとトピックの両方を使用している

4. パラメータ (Parameter)

概要

- ノードの振る舞いを変えるパラメータ専用通信

用途

ロボットの動作などを実行中に変えたいとき

コマンド

ROS 1 はLinux環境でしか使用できなかったため,ROS2もLinux文化の影響が強く GUIではなく、コマンドを使用するCUIアプリを使用することが一般的

コマンドが使えないとROS 2を使うことは非っ常にキビシー

ROS2ではコマンドとしてros2コマンドを使う
どんなのがあるか調べてみよう

```
ros2 --help
```

```
ros2 サブコマンド --help
```

サブコマンド	内容
action	アクション通信関連
bag	Rocbag関連.rosbagはセンサ情報を記録，再生する強力なツール
launch	ローチンファイルを実行. ローチンファイルは複数ののードやその設定などをまとめて起動するために使うファイル
node	ノード関連
pkg	パッケージ関連
run	パッケージの実行可能ファイルを実行
service	サーブス通信関連

初めてのROS2プログラミング

プログラムの流れ

1. ワークスペースの作成：作業スペースを作る，パッケージはこの中に
2. パッケージの作成：ROS2はパッケージ単位でプログラムを作る
3. ソースコードの作成：ノードのソースコードを作る
4. ビルド：ノードの実行ファイルを作る
5. 設定ファイルの反映：ノードが実行できるように環境を整える
6. ノードの実行

1.ワークスペースの作成

ROS2ではプログラムをパッケージと呼ばれる単位でつくります．はじめにパッケージを保存するための作業用ディレクトリを作らなければなりません．それがワークスペースです．1つのワークスペースに何個でもパッケージを作れます．

また，自作パッケージを使うにはROSシステムのワークスペース環境（アンダーレイ）と自作ワークスペース環境(オーバーレイ)を整えなければならない．

- 設定ファイルの実行

`source` コマンドでアンダーレイの設定ファイルを実行

```
source /opt/ros/humble/setup.bash
```

- ワークスペース用のディレクトリ作成

`mkdir` コマンドでディレクトリを作成

```
mkdir -p ~/hazimete/src
```

2. パッケージの作成

パッケージは自作したROS2コードの入れ物．パッケージを使うことでかんたんにビルでできたり一般公開できるようになる

ファイル構成

- setup.py: パッケージのインストール法が書かれたファイル
- setup.cfg: パッケージ実行ファイルがある場合に必要なファイル (ros2 run)
- package.xml: パッケージに関する情報
- パッケージ名: パッケージと同じ名前のディレクトリ

- パッケージの作成

`cd` コマンドでディレクトリを移動する

```
cd ~/hazimete/src
```

つぎに `ros2 pkg create` コマンドでパッケージを作成

```
ros2 pkg create --build-type ament_python hazimetepkg
```

なお,

```
ros2 pkg create --build-type ament_python --node-name hazimetenode hazimetepkg
```

とするとノードも作れる.

3. ソースコードの作成

現在の中身

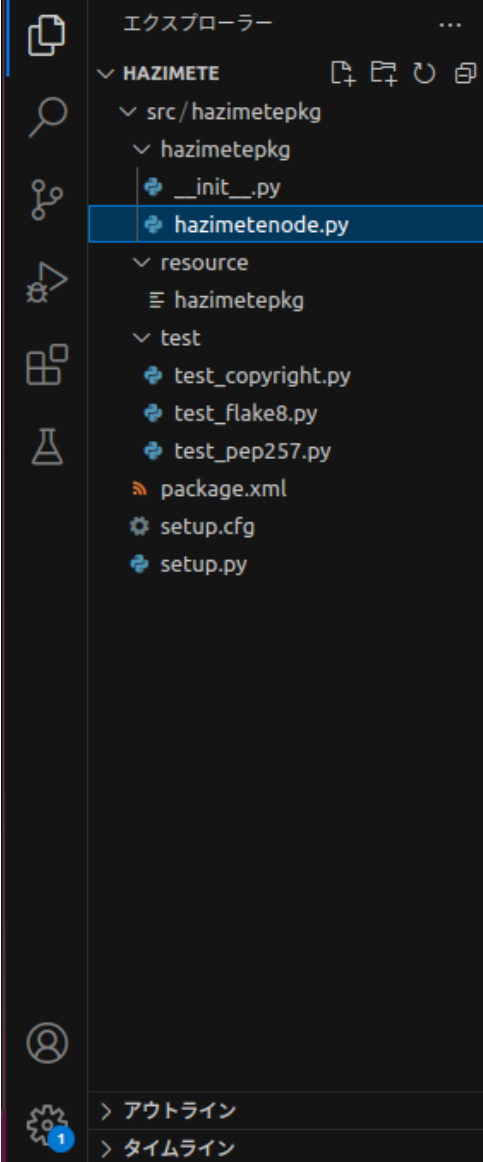
```
altair@altair-ThinkPad-L380:~/hazimete/src$ exa -T
.
├── hazimetepkg
│   ├── hazimetepkg
│   │   ├── __init__.py
│   │   └── hazimetenode.py
│   ├── package.xml
│   ├── resource
│   │   └── hazimetepkg
│   ├── setup.cfg
│   ├── setup.py
│   └── test
│       ├── test_copyright.py
│       ├── test_flake8.py
│       └── test_pep257.py
```

ここからは、使いやすいvscodeを使います

```
cd ~/hazimete
code .
```

とするとvscodeが立ち上がります。 `.` はカントディレクトリ

ファイル 編集 選択 表示 移動 実行 ターミナル ヘルプ



```
src > hazimetepkg > hazimetepkg > hazimetenode.py
1  def main():
2      print('Hi from hazimetepkg.')
3
4
5  if __name__ == '__main__':
6      main()
7
```

- package.xmlの編集

コメントアウトが変更する場所

```
<name>hazimetepkg</name> # 1.パッケージ名  
<version>0.0.0</version> # 2.パッケージのバージョン  
<description>TODO: Package description</description> # 3.パッケージの説明  
<maintainer email="Altairu@github.com">altair</maintainer> #4.保守者のメールアドレス  
<license>TODO: License declaration</license> # 5. パッケージのライセンス
```

- setup.pyの編集

setup.pyを変更しないとノードを実行できません

コメントアウトしている7つを変更しなければならない。

一般公開しない場合は7つ目のみ要変更

```

from setuptools import find_packages, setup

package_name = 'hazimetepkg' # 1.パッケージ名

setup(
    name=package_name,
    version='0.0.0', # 2.パッケージのバージョン
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
         ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='altair', #3.保守者
    maintainer_email='Altairu@github.com', #4.保守者のメールアドレス
    description='TODO: Package description', #5.パッケージの説明
    license='TODO: License declaration', #6. パッケージのライセンス
    tests_require=['pytest'],
    entry_points={ #7.エントリポイント
        'console_scripts': [
            #ノード名=パッケージ名.ノード名:main
            'hazimetenode = hazimetepkg.hazimetenode:main'#pythonファイルごとに必要
        ],
    },
)

```

エントリポイントを自動生成したければ `--node-name` のオプションをつければよし

- ソースコード作成

本来ならここでソースコードを作るが自動生成されたものを使ってみましょう
hazimetenode.pyは自動生成されたプログラムです.

```
def main():  
    print('Hi from hazimetepkg.')
```

if __name__ == '__main__':
 main()

4.ビルド

パッケージを作るためにビルドしないといけない

ROS 2 ではビルドシステムとして `ament` を使い，ビルドコマンドとして `colcon` を使う

ちなみにワークスペース名直下のディレクトリしかビルドできない

```
cd ~/hazimete  
colcon build
```

5. 設定ファイルの反映

パッケージののーどを実行する前に，ROS2の設定ファイルをsourceコマンドで反映させる

- アンダーレイ設定ファイルの反映

```
source /opt/ros/humble/setup.bash
```

- オーバーレイ設定ファイルの反映

自作ワークスペースの設定を反映させる．これで作成したワークスペースの場所がわかるようになりノードを実行できる

```
source ~/hazimete/install/setup.bash
```


6. ノードの実行

```
ros2 run hazimetestpkg hazimetenode
```

```
altair@altair-ThinkPad-L380:~/hazimete$ ros2 run hazimetestpkg hazimetenode  
Hi from hazimetestpkg.  
altair@altair-ThinkPad-L380:~/hazimete$
```

ROS2プログラムの処理の流れ

1. モジュールのインポート

- ROS2でpythonプログラムを作るには `rclpy` をインポートする必要がある。

2. 初期化

- `rclpy.init()` :ROS2通信のための初期化

3. ノードの作成

- nodeクラスのインスタンス化：Nodeクラスを継承してクラスを作り,そのインスタンスをさくせいすることでノードを作成する。

4. ノード処理

- `rclpy.spin()` :繰り返し処理可能 `rclpy.spin_once()` :処理を 1 回実行

5. 終了処理

- `rclpy.shutdown()` :終了処理します。

ROS2プログラミング

ROS2では基本的にクラスを使ってプログラミングします。

test1_node.py

```
import rclpy # 1. ROS2 Python モジュールのインポート
from rclpy.node import Node # rclpy.node モジュールから Node クラスをインポート

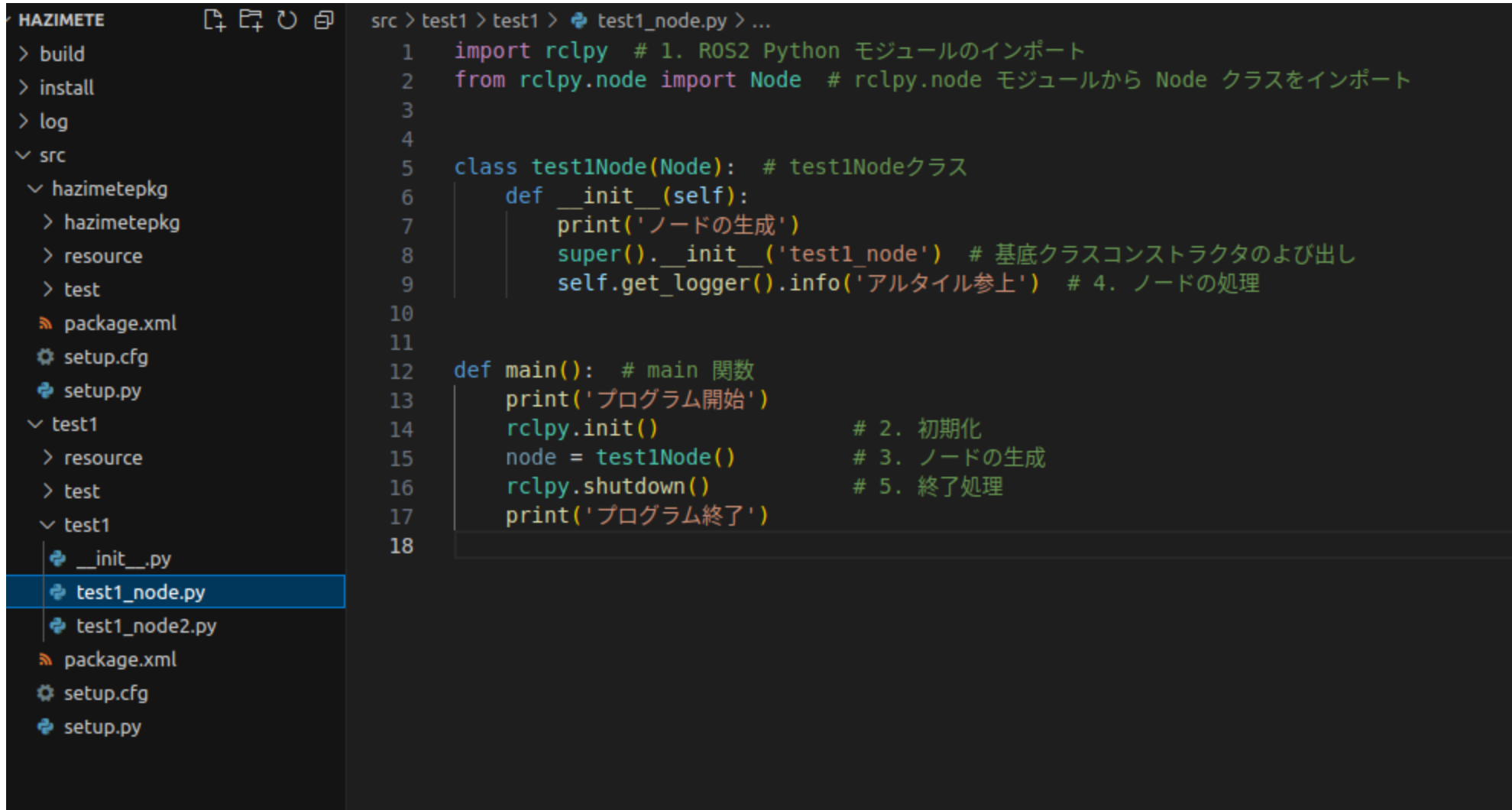
class test1Node(Node): # test1Nodeクラス
    def __init__(self):
        print('ノードの生成')
        super().__init__('test1_node') # 基底クラスコンストラクタのよび出し
        self.get_logger().info('アルタイル参上') # 4. ノードの処理

def main(): # main 関数
    print('プログラム開始')
    rclpy.init() # 2. 初期化
    node = test1Node() # 3. ノードの生成
    rclpy.shutdown() # 5. 終了処理
    print('プログラム終了')
```

- インポート (1-2行目): 1行目のrcipy は ROS2のPython モジュールなので必ずインポートしなければなりません、2行目はノードをつくるために必要でrclpy node モジュールから Node クラスをインポートします。この2行は常に必要です
- クラスの定義 (5~9行目): test1Nodeクラスを定義しています.コンストラクタの8行目で基底クラスNodeのコンストラクタを呼び出すことでノードを生成しています. 引数test1_Nodeはノード名です。9行目のget_logger().info()はノードのメソッドでログ(log)情報(この例ではアルタイル参上)を端末に表示します. print文とは違い、端末だけでなくROS2のアプリrqt_console でも読むことができます
- main()関数(12~17行目): このプログラムは main()関数から実行されるので、main()関数が前節で説明したsetup.pyのエントリポイント(開始点)です
- relpy.init()(14行目):rclpy.init()でROS2通信を初期化します.ノードをつくる前に呼び出さなければいけません
- クラスのインスタンス化 (15行目): test1Nodeクラスのインスタンスnodeを生成しています.
- rclpy.shutdown() (16行目): relpy.shutdown() で終了処理をしています.

1回目で作ったワークスペースに違う名前で作ってパッケージを作ってみましょう

同じパッケージ名があるとビルドできないので気をつけましょう。



The screenshot shows a ROS2 development environment. On the left is a file explorer with a tree view. The root is 'HAZIMETE'. Under 'src', there is a package named 'hazimetepkg'. Inside 'hazimetepkg', there is a subdirectory 'test1'. Within 'test1', there is a file named 'test1_node.py', which is currently selected and highlighted in blue. Other files visible in the tree include 'package.xml', 'setup.cfg', and 'setup.py' at the package level, and 'resource' and 'test' subdirectories. On the right is a code editor showing the contents of 'test1_node.py'. The code is written in Python and includes comments in Japanese. It imports 'rclpy' and 'Node' from 'rclpy.node', defines a 'test1Node' class that inherits from 'Node', and a 'main' function. The 'main' function calls 'rclpy.init()', creates an instance of 'test1Node', calls 'rclpy.shutdown()', and prints status messages. Line numbers 1 through 18 are visible on the left side of the code editor.

```
src > test1 > test1 > test1_node.py > ...
1  import rclpy # 1. ROS2 Python モジュールのインポート
2  from rclpy.node import Node # rclpy.node モジュールから Node クラスをインポート
3
4
5  class test1Node(Node): # test1Nodeクラス
6      def __init__(self):
7          print('ノードの生成')
8          super().__init__('test1_node') # 基底クラスコンストラクタのよび出し
9          self.get_logger().info('アルタイル参上') # 4. ノードの処理
10
11
12 def main(): # main 関数
13     print('プログラム開始')
14     rclpy.init() # 2. 初期化
15     node = test1Node() # 3. ノードの生成
16     rclpy.shutdown() # 5. 終了処理
17     print('プログラム終了')
18
```

実行してみましょう

```
~/hazimete$ source ~/hazimete/install/setup.bash
~/hazimete$ source /opt/ros/humble/setup.bash
~/hazimete$ colcon build
~/hazimete$ ros2 run test1 test1_node
```

```
altair@altair-ThinkPad-L380:~/hazimete$ source ~/hazimete/install/setup.bash
altair@altair-ThinkPad-L380:~/hazimete$ source /opt/ros/humble/setup.bash
altair@altair-ThinkPad-L380:~/hazimete$ ros2 run test1 test1_node
プログラム開始
ノードの生成
[INFO] [1701175083.866729470] [test1_node]: アルタイル参上
プログラム終了
```

コールバックを使ったプログラム

ROS2では コールバック関数 や コールバックメソッド を多用してプログラムを作る。

- コールバック関数：プログラム中で、呼び出し先の関数の実行中に実行されるように、あらかじめ指定しておく関数

→マウスで線を書くなどの処理を実装するとき

あるイベントが起きたときになにか処理させたい場合に使われる

ROS2では `spinOnce()` や `spin()` でコールバックを明示的に呼び出す必要がある。

タイマを使ったコールバック

test1_node2.py

```
import rclpy # 1. ROS2 Python モジュールのインポート
from rclpy.node import Node # rclpy.node モジュールから Node クラスをインポート

class test1Node2(Node): # test1Node2クラス
    def __init__(self): # コンストラクタ
        print('ノードの生成')
        super().__init__('test1_node2') # 基底クラスコンストラクタのよび出し
        self.timer = self.create_timer(1.0, self.timer_callback) # タイマーの生成

    def timer_callback(self): # タイマーのコールバック関数
        self.get_logger().info('アルタイル参上！？')

def main(): # main 関数
    print('プログラム開始')
    rclpy.init() # 2. 初期化
    node = test1Node2() # 3. ノードの生成
    rclpy.spin(node) # 4. ノードの処理. コールバック関数を繰り返しよび出す.
    rclpy.shutdown() # 5. 終了処理
    print('プログラム終了')
```


- タイマの生成(8行目):タイマを使うためには,
`create_timer(timer_period,callback)` を使う. 1 番目の引数は繰り返し間隔 [s], 2 番目はコールバック
- コールバック(10~11行目):`timer_callback`はタイマによって周期的に呼び出されるコールバック. ここでは「アルタイル参上!？」って表示させてるだけ
- `rcipy.spin` (17行目):`spin`で何度もコールバックを呼び出します. プログラムはここでブロックされます.

ではノードを追加してみましょう

```
src > test1 > test1 > test1_node2.py > ...
1  import rclpy # 1. ROS2 Python モジュールのインポート
2  from rclpy.node import Node # rclpy.node モジュールから Node クラスをインポート
3
4  class test1Node2(Node): # test1Node2クラス
5      def __init__(self): # コンストラクタ
6          print('ノードの生成')
7          super().__init__('test1_node2') # 基底クラスコンストラクタのよび出し
8          self.timer = self.create_timer(1.0, self.timer_callback) # タイマーの生成
9
10     def timer_callback(self): # タイマーのコールバック関数
11         self.get_logger().info('アルタイル参上! ?')
12
13     def main(): # main 関数
14         print('プログラム開始')
15         rclpy.init() # 2. 初期化
16         node = test1Node2() # 3. ノードの生成
17         rclpy.spin(node) # 4. ノードの処理. コールバック関数を繰り返しよび出す.
18         rclpy.shutdown() # 5. 終了処理
19         print('プログラム終了')
20
```

'test1_node = test1.test1_node:main', 'test1_node2 = test1.test1_node2:main'
steup.pyに加えるの忘れずに

```
ros2 run test1 test1_node2
```

```
altair@altair-ThinkPad-L380:~/hazimete$ ros2 run test1 test1_node2
```

プログラム開始

ノードの生成

```
[INFO] [1701171151.246723732] [test1_node2]: アルタイル参上! ?  
[INFO] [1701171152.237369868] [test1_node2]: アルタイル参上! ?  
[INFO] [1701171153.237354129] [test1_node2]: アルタイル参上! ?  
[INFO] [1701171154.237482266] [test1_node2]: アルタイル参上! ?  
[INFO] [1701171155.237719240] [test1_node2]: アルタイル参上! ?  
[INFO] [1701171156.237405367] [test1_node2]: アルタイル参上! ?  
[INFO] [1701171157.237395603] [test1_node2]: アルタイル参上! ?
```

^CTraceback (most recent call last):

File "/home/altair/hazimete/install/test1/lib/test1/test1_node2", line 33, in <module>

sys.exit(load_entry_point('test1==1.0.0', 'console_scripts', 'test1_node2')())

File "/home/altair/hazimete/install/test1/lib/python3.10/site-packages/test1/test1_node2.py", line 17, in main

rclpy.spin(node) # 4. ノードの処理. コールバック関数を繰り返しよび出す.

File "/opt/ros/humble/local/lib/python3.10/dist-packages/rclpy/__init__.py", line 222, in spin

executor.spin_once()

File "/opt/ros/humble/local/lib/python3.10/dist-packages/rclpy/executors.py", line 705, in spin_once

handler, entity, node = self.wait_for_ready_callbacks(timeout_sec=timeout_sec)

File "/opt/ros/humble/local/lib/python3.10/dist-packages/rclpy/executors.py", line 691, in wait_for_ready_callbacks

return next(self._cb_iter)

File "/opt/ros/humble/local/lib/python3.10/dist-packages/rclpy/executors.py", line 588, in _wait_for_ready_callbacks

wait_set.wait(timeout_nsec)

KeyboardInterrupt

[ros2run]: Interrupt

Ctrl+Cで強制的にプログラムを終了させてください。

例外処理

```
def main(): # main 関数
    print('プログラム開始')
    rclpy.init() # 2. 初期化
    node = test1Node3() # 3. ノードの生成
    try: # 例外処理. 美しく終わるため.
        rclpy.spin(node) # 4. ノードの処理. コールバック関数を繰り返しよび出す.
    except KeyboardInterrupt:
        print('Ctrl+Cが押されました')
    rclpy.shutdown() # 5. 終了処理
    print('プログラム終了')
```

次回はトピック通信のプログラムの作り方