

リスト1 swarm_robot_utils.py の定数定義抜粋

```
###
### 各種定数（抜粋）
###

### フィールド・サイズ
FIELD_SIZE = 10.0

### 停止後、ランダム移動に移るまでの秒数（ランダム移動モード）
STOPPED_WAITING_SEC = 1.0

### ランダム移動を行う秒数（ランダム移動モード）
RANDOM_MOVE_SEC = 3.0
```

リスト2 swarm_robot_utils.pyのEnumやdataclassによる定義

```
###
### ロボット動作ステータス
###

class RobotStatus(Enum):
    """ Robot status """
    STOPPED = auto() # 停止中
    COMMAND_MOVE = auto() # 上位からのコマンドにより移動中
    RANDOM_MOVE = auto() # ランダム方向移動中
    TO_COLLECTION = auto() # 回収地点に移動中
    POS_RESET = auto() # 回収後（ランダム位置移動）

###
### ロボット（Robot）データクラス（抜粋）
###

@dataclass
class Robot:
    """ Robot dataclass """
    # ID
    id: int
    # 現在位置
    pos: List[float] = field(default_factory = lambda: [0.0, 0.0])
    # ステータス
    status: RobotStatus = RobotStatus.STOPPED
    # 移動先（指定されていれば移動）
    target: Optional[List[float]] = None
```

リスト3 swarm_robot_utils.py create_markers処理（抜粋）

```
###
### create_markers（抜粋）
###
def create_markers(
    stamp: Time, # タイムスタンプ
    robots: List[Robot], # ロボット
    garbages: List[Garbage], # ごみ
    collection_point: List[float], # 回収位置
    points: int, # 起動してからごみを捨てた回数
    collection_times: List[float], # 回収されたごみ毎の、捨てるまでにかかった時間
) -> MarkerArray:

    markers = MarkerArray()

    ### ロボットの位置をマーカーに設定して登録
    for robot in robots:
        marker = Marker()
        marker.header.frame_id = 'map'
        marker.header.stamp = stamp
        marker.ns = 'robots'
        marker.id = robot.id
        marker.type = Marker.CYLINDER
        marker.action = Marker.ADD
        marker.pose.position.x = robot.pos[0]
        marker.pose.position.y = robot.pos[1]
        marker.pose.position.z = 0.1
        marker.scale.x = 0.3
        marker.scale.y = 0.3
        marker.scale.z = 0.2

        # （中略）

        markers.markers.append(marker)

    # 以降、ごみ、回収地点も同様。
    # 得点表示もテキスト表示のマーカーを追加することで行っている。
```

```
return markers
```

リスト4 swarm_robot_random.pyの概要

```
###
### Swarm robot (random mode)
###

class SwarmRobotRandomNode(Node) :

    def __init__(self) :
        # ロボット数、ごみ数の設定
        # 各種変数の初期化

    def initialize(self) :
        # ロボットの初期化
        # ごみをランダム配置

    def update_simulation(self) :
        # ロボットを単位時間 (0.1秒) 動かす
        # ロボットに拾われたごみの処理など

    def move_robot_to_target(self, robot: Robot, speed: Optional[float] = None) :
        # ロボットを単位時間 (0.1秒) 目標に向かわせる

    def publish_markers(self) :
        # MarkerArray を publish して Rviz の表示を更新

###
### Main
###

def main() :
    rclpy.init()
    node = SwarmRobotRandomNode()
    try :
        rclpy.spin(node)
    except KeyboardInterrupt :
        pass
    finally :
        if rclpy.ok() :
            rclpy.shutdown()

if __name__ == '__main__' :
    main()
```

リスト5 move_command_callback (抜粋)

```
# ロボット移動コマンド
def move_command_callback(self, msg) :
    """ ロボット移動命令受信時、コマンドを解釈して移動命令登録 """

    # NOTE: "ID:3,X:1.2,Y:-2.5" 形式で送られてくる
    #       命令を解釈し、update_simulation時に適用

    try :
        ### 命令を分解
        parts = msg.data.split(',')
        id_part = [p for p in parts
                    if p.strip().startswith('ID:')]
        x_part = [p for p in parts
                  if p.strip().startswith('X:')]
        y_part = [p for p in parts
                  if p.strip().startswith('Y:')]
        if not id_part or not x_part or not y_part :
            return # 見つからなかったら破棄
        robot_id = int(id_part[0].split(':')[1])
        x = float(x_part[0].split(':')[1])
        y = float(y_part[0].split(':')[1])

        # 中略

        ### 登録
        robot = next((r for r in self.robots
                      if r.id == robot_id), None)
        robot.target = [x, y]
        robot.status = RobotStatus.COMMAND_MOVE
        # 移動状態にする
```

リスト6 search_command_callback (抜粋)

```
### 状態取得コマンド
def search_command_callback(self, msg) :
    """ 状況取得命令受信時、コマンドを解釈して適切な結果をPublish """
```

全リスト.txt

```

cmd = msg.data.strip().lower()

### Robot status ... 各ロボットの位置と一番近いごみまでの位置座標を取得
if cmd == 'robot_status' :

    status_list = []
    for robot in self.robots :

        # (中略。最も近いごみを求めて min_dist, min_garbage に入れる)
        min_dist = ...
        min_garbage = ...

        ### 文字列に変換
        status_list.append(

f' ID: {robot.id}, X: {robot.pos[0]:.2f}, Y: {robot.pos[1]:.2f}, HAS_GARBAGE: {has_garbage}, MIN_DIST: {min_dist:.2f}, GARBAGE_X: {min_garbage.pos[0]:.2f}, GARBAGE_Y: {min_garbage.pos[1]:.2f}'
        )

    ### 結合してpublish
    msg_out = String()
    msg_out.data = ';'.join(status_list)
    self.robots_status_pub.publish(msg_out)

### Field status ... 環境の情報を取得
if cmd == 'field_status' :
    # 略

```

リスト7 SwarmRobotCommandMcpNode コマンド送受信機能 (抜粋)

```

class SwarmRobotCommandMcpNode(Node) :
    # 一部関数のみ抜粋。その他省略。

    def send_move_command(self, robot_id, x, y) :
        """ ロボット移動命令送信 """
        msg = String()
        msg.data = f' ID: {robot_id}, X: {x}, Y: {y}'
        self.move_pub.publish(msg)

    def wait_for_robots_status(self, timeout = 2.0) :
        """ (コマンド送信後) ロボット情報受信待ち """
        start = time.time()
        while True :
            with self.lock :
                if self.latest_robots_status is not None :
                    result = self.latest_robots_status
                    self.latest_robots_status = None
                    return result
            if time.time() - start > timeout :
                return None # タイムアウト
            rclpy.spin_once(self, timeout_sec = 0.1)
            # 一定時間 spin_once

```

リスト8 SwarmRobotCommandMcpServer MCPコマンド定義部 (抜粋)

```

class SwarmRobotCommandMcpServer :
    # 一部関数のみ抜粋。その他省略。

    def __init__(self, node, transport = 'sse',
                  port = 5000, host = '0.0.0.0') :
        """ Constructor """
        # (略)

        ### MCP Tools 登録
        self.register_tools()

    def register_tools(self) :
        """ MCP 機能登録 """

        ### move_robot ... 指定したIDのロボットを (x, y) に移動させる
        @self.mcp.tool()
        def move_robot(
            robot_id: Annotated[int, Field(description
                = 'Robot ID to move')],
            x: Annotated[float, Field(description
                = 'Target x position')],
            y: Annotated[float, Field(description
                = 'Target y position')]
        ) -> dict :
            'Move a robot to the specified position.'
            self.node.send_move_command(robot_id, x, y)
            return {'result': 'OK'} # 常に成功

        ### get_robots_status ... 各ロボットの状態取得
        @self.mcp.tool()

```

全リスト.txt

```
def get_robots_status() -> dict :
    'Get the status of all robots.'
    self.node.send_search_command(
        'robot_status')
    status_str =
        self.node.wait_for_robots_status()
    robots = parse_robots_status(status_str)
        if status_str else []
    return {'robots': robots}
```

リスト9 swarm_robot_command_auto_execute.py move_robot コマンド呼び出し部（抜粋）

```
MCP_URL = 'http://localhost:5000/sse'

### move_robot コマンドを呼び出し
async def move_robot(client, robot_id, x, y):
    payload = {'robot_id': robot_id, 'x': x, 'y': y}
    result = await client.call_tool('move_robot',
                                    payload)
    return json.loads(result.content[0].text)

### メイン処理内で robot3 を (2.2, 1.5) に移動
async def main():
    async with Client(MCP_URL) as client:
        await move_robot(client, 3, 2.2, 1.5)
```

リスト10 LLMにごみ処理をさせるため

(a) 入力文

swarm_robot MCP は複数のロボットを動かし、ごみ処理をさせる機能を有しています。
get_robots_status は各ロボットの状態（位置、最も近いゴミの位置など）を取得できます。
move_robot は指定したIDのロボットを指定位置に移動させることができます。
ロボットはごみを回収すると、自動で回収地点まで向かい、その後ランダムな位置に戻ります。

まず get_robots_status でごみまでの距離を取得後、
move_robot を呼び出して最短距離のロボットをごみ処理に向かわせてください。

(b) 回答文

タスクは、swarm_robot MCPサーバの get_robots_status でロボットの状態（ごみまでの距離含む）を取得し、最短距離のロボットを move_robot でごみ処理に向かわせること。

まず get_robots_status を呼び、レスポンスから最もごみに近いロボットを特定し、そのロボットのIDとごみの座標を move_robot に渡す。

（中略。get_robots_status を呼び出して最短距離のロボットのIDを得る）

get_robots_statusの結果から、最短距離（min_dist）のロボットはid:9（min_dist:0.22）。
このロボットの座標は（4.76, 2.8）、ごみの座標は（4.83, 2.59）。
move_robotでid:9のロボットを（4.83, 2.59）に移動させる。

（略。move_robotを呼び出してごみを拾わせる）

リスト11 効率的に実行する方法を考えさせる

(a) 入力文

field_status ではロボット数、ごみ数が取得できます。これらの情報を用いて効率よく move_robot を出していただけますか。

(b) 回答文

ユーザから「field_status で取得できるロボット数・ごみ数を用いて効率よく move_robot を出して★欲しい」と依頼。

まず get_field_status でロボット数（robot_num）とごみ数（garbage_num）を取得し、
次に get_robots_status で各ロボットのごみまでの距離・座標を取得し、
ごみ数分だけ、最も近いロボットをそれぞれのゴミ座標へ move_robot するのが効率的。

ただし、get_robots_status の結果から、各ロボットに割り当てられている garbage_x, garbage_y が異なる場合は、
ごみ数分だけ「最も近いロボット」を選び、重複しないように move_robot を出す必要がある。

（中略。まず get_field_status を実行してロボット数、ごみ数を取得）
（中略。get_robots_status を実行してロボットごとの最短ごみへの距離を取得）

抽出されるゴミ座標（重複除去）：

- (-2.61, -2.2)
- (1.88, 3.51)
- ... (略)

各ゴミ座標に最も近いロボットを割り当てる：

- 1. (-2.61, -2.2) : id=8 (min_dist=1.6)
- 2. (1.88, 3.51) : id=4 (min_dist=2.66)
- ... (略)

それぞれ move_robot を順に実行する。まずは id=8 を (-2.61, -2.2) へ移動。

（以下略。順番に move_robot を呼び出す）

リストA Docker環境の操作

全リスト.txt

```
# 事前に docker コマンドがインストールされていること
# swarm_robot_demo フォルダ内で以降を実行
```

```
# ビルド
$ docker compose build↵
# 起動
$ docker compose up -d↵
# bash に入る
$ docker compose exec swarm_robot_demo bash↵
# 作業後, bash 環境から exit で抜ける
$ exit↵
# 終了
$ docker compose down↵
```

リストB 手動でのパッケージ配置

```
# 事前に ROS2 コマンドの環境変数を読み込む
$ source /opt/ros/jazzy/setup.bash↵
# ワークスペースを作成してパッケージを移動
$ mkdir ros2_ws/src↵
$ mv -r swarm_robot_demo ros2_ws/src↵
# 依存モジュールを導入してビルド (rosdep設定済みであることを前提としている)
$ rosdep install --from-paths src --ignore-src -r -y↵
$ colcon build --cmake-args -DCMAKE_BUILD_TYPE=Release --symlink-install ↵
# ビルド後にワークスペースの環境変数を読み込む
$ source ./install/setup.bash↵
```

リストC ROS2 launch 実行

```
# ランダム動作モード+Rviz2
$ ros2 launch swarm_robot_demo swarm_random_demo.launch.py↵
# 上位操作モード+上位操作モードMCP Server+Rviz2
$ ros2 launch swarm_robot_demo swarm_command_demo.launch.py↵
```

リストD ROS2 run 実行

```
# ランダム動作モード
$ ros2 run swarm_robot_demo swarm_robot_random↵
# 上位操作モード
$ ros2 run swarm_robot_demo swarm_robot_command↵
# 上位操作モードMCP Server
$ ros2 run swarm_robot_demo swarm_robot_command_mcp_server↵
```