BotAi

La classe `BotAi` fournit une interface haut niveau pour interagir avec le serveur de combat de bots. Il permet à un client de s'inscrire avec un bot, de lire les messages entrants du serveur et de réaliser des actions telles que se déplacer, tourner et tirer.

Propriétés de la classe

- `bot_id` : une propriété en lecture seule qui retourne l'identificateur du bot attribué par le serveur.

Méthodes de la classe

- `__init__(self, bot_name: str, team_id: str)` : Initialise l'instance de `BotAi` avec le `bot_name` et le `team_id` donnés.
- `__enter__(self)` : Permet l'utilisation de la déclaration `with` pour s'assurer que la méthode `close` est appelée.
- `__exit__(self, exc_type, exc_val, exc_tb)` : Appelé lorsqu'une déclaration `with` est quittée.
- `close(self)` : Ferme tous les threads ouverts.
- `enroll(self, bot_id: str = str()) -> str` : Inscrit ou réinscrit un bot sur le serveur. Donner un identificateur de bot existant vous permet de connecter ce bot sans en ajouter un nouveau au jeu. Retourne l'identificateur du bot attribué par le serveur.
- `read_scanner(self) -> dict` : Lit et retire un élément de la file d'attente du scanner. Retourne un dictionnaire contenant les données. **Exemples**
    - Un arbre et un bot sont détectés:


```
{
    "msg_type": "object_detection",
    "source": "scanner",
    "data": [
        {
            "from": 26.5,
            "to": 31,
            "object_type": "tree",
            "name": "Tree",
            "distance": 6.844473640068633
        },
        {
            "from": 34,
            "to": 39.5,
            "object_type": "bot",
            "name": "MyBot01",
            "distance": 5.777348380771669
        }
```

```
    ]
}
```

- `read_game_message(self) -> dict` : Lit et retire un élément de la file d'attente de messages de jeu. Retourne un dictionnaire contenant les données. **Exemples**
    - État de la partie:

```
{
    "msg_type": "game_status",
    "source": "server",
    "data": {
        "value": false
    }
}
```

    - Points de vies restants au bot:

```
{
    "msg_type": "health_status",
    "source": "bot",
    "data": {
        "value": 95
    }
}
```

    - Si le bot a été assomé (déplacements impossibles):

```
{
    "msg_type": "stunning_status",
    "source": "bot",
    "data": {
        "value": true
    }
}
```

    - Si le bot a commencé ou arrêté de se déplacer:

```
{
    "msg_type": "moving_status",
    "source": "bot",
    "data": {
        "value": false
    }
}
```

- Si le bot a commencé ou arrêté de tourner:

```
{
    "msg_type": "turning_status",
    "source": "bot",
    "data": {
        "value": false
    }
}
```

- `move(self, state: str)` : Commence ou arrête de faire avancer le bot. `state` peut être soit `"start"` ou `"stop"`.
- `turn(self, direction: str)` : Commence ou arrête de tourner le bot dans une direction. `direction` peut être `"left"`, `"right"` ou `"stop"`.
- `shoot(self, angle: float)` : Tire à l'angle souhaité, spécifié en degrés.

Exceptions

- `RestException` : Levée en cas d'erreur lors d'un appel d'API REST.

Exemple d'utilisation

```
import logging
from time import sleep
from threading import Event, Thread
from battlebotslib.BotAi import BotAi


G_GAME_IS_STARTED = False

G_BOT_HEALTH = 100
G_BOT_IS_MOVING = False
G_BOT_IS_TURNING = False


def thread_read_scanner_queue(e: Event, bot_ai: BotAi):
    """
    Thread continuously reading messages from the bot scanner queue.
    """
    while not e.is_set():
        scanner_message = bot_ai.read_scanner()
        logging.debug(f"[SCANNER] {scanner_message}")
        handle_scanner_message(scanner_message)


def thread_read_game_queue(e: Event, bot_ai: BotAi):
```

```
    """
    Thread continuously reading messages from the game queue.
    """
    while not e.is_set():
        game_message = bot_ai.read_game_message()
        logging.debug(f"[GAME] {game_message}")
        handle_game_message(game_message)


def handle_scanner_message(message: dict):
    """
    Handle a new scanner message.
    """
    try:
        if 'msg_type' in message and message['msg_type'] == 'object_detection':
            for detected_object in message['data']:
                # Checking if an object is detected
                if detected_object['name'] is not None:
                    angle = (detected_object['from'] + detected_object['to']) / 2
                    logging.info(
                        f"[SCANNER] {detected_object['name']} detected at a
distance of "
                        f"{detected_object['distance']} ({angle}°)"
                    )
        else:
            logging.error("Not an object detection scanner message")
    except:
        logging.error("Bad scanner message format")


def handle_game_message(message: dict):
    """
    Handle a new game message.
    """
    try:
        if 'msg_type' in message:
            # Health update message
            if message['msg_type'] == 'health_status':
                global G_BOT_HEALTH
                current_health = message['data']['value']
                G_BOT_HEALTH = current_health
                logging.info(f"[BOT] Health: {current_health}")
            # Game update message
            if message['msg_type'] == 'game_status':
                global G_GAME_IS_STARTED
                G_GAME_IS_STARTED = message['data']
            # Bot moving update message
            elif message['msg_type'] == 'moving_status':
                global G_BOT_IS_MOVING
                # The message tells us the bot has stopped moving
                if not message['data']['value']:
                    # Bot has been stopped
                    G_BOT_IS_MOVING = False
            # Bot turning update message
```

```
            elif message['msg_type'] == 'turning_status':
                global G_BOT_IS_TURNING
                # The message tells us the bot has stopped turning
                if not message['data']['value']:
                    # Bot has been stopped
                    G_BOT_IS_TURNING = False
        else:
            logging.error("Not a game message")
    except:
        logging.error("Bad game message format")


if __name__ == "__main__":
    # Logging
    logging.basicConfig(
        level=logging.DEBUG, datefmt='%d/%m/%Y %I:%M:%S', format='[%(levelname)s]
%(asctime)s - %(message)s'
    )

    # Creating a new Bot
    with BotAi(bot_name="MyBot", team_id="given-team-id") as bot:
        def stop():
            # Closing messages reading threads
            scanner_message_thread_event.set()
            game_message_thread_event.set()

        # Enrolling the new bot on the server
        bot_id = bot.enroll()

        # Bot scanner messages handler thread
        scanner_message_thread_event = Event()
        Thread(target=thread_read_scanner_queue, args=
(scanner_message_thread_event, bot)).start()

        # Game messages handler thread
        game_message_thread_event = Event()
        Thread(target=thread_read_game_queue, args=(game_message_thread_event,
bot)).start()

        try:
            # Waiting for the game to start
            while not G_GAME_IS_STARTED:
                sleep(0.1)

            # While the bot is alive and the game is running
            while G_BOT_HEALTH > 0 and G_GAME_IS_STARTED:
                # ################################################################
                #
                # AI logic goes here
                #
                # ################################################################
                pass

            # Game has stopped or the bot is dead
```

```
            stop()

        except KeyboardInterrupt:
            logging.info("Bot has been aborted")
            stop()

        except BotAi.RestException as ex:
            if ex.name == 'GAME_NOT_STARTED':
                logging.info("Game has been stopped")
                stop()
            else:
                raise
```