地图使用文档（地图1）

1. Motphys仿真平台使用教程（https://motrixlab.readthedocs.io/zh-cn/latest/user_guide/tutorial/basic_frame.html）

基础框架

MotrixLab 是一个机器人强化学习平台，这一节我们会介绍 MotrixLab 的框架设计以及各个组成部分之间的关系。

MotrixLab 的框架设计

MotrixLab 采用分层架构设计，将训练环境与训练逻辑进行了清晰拆分：

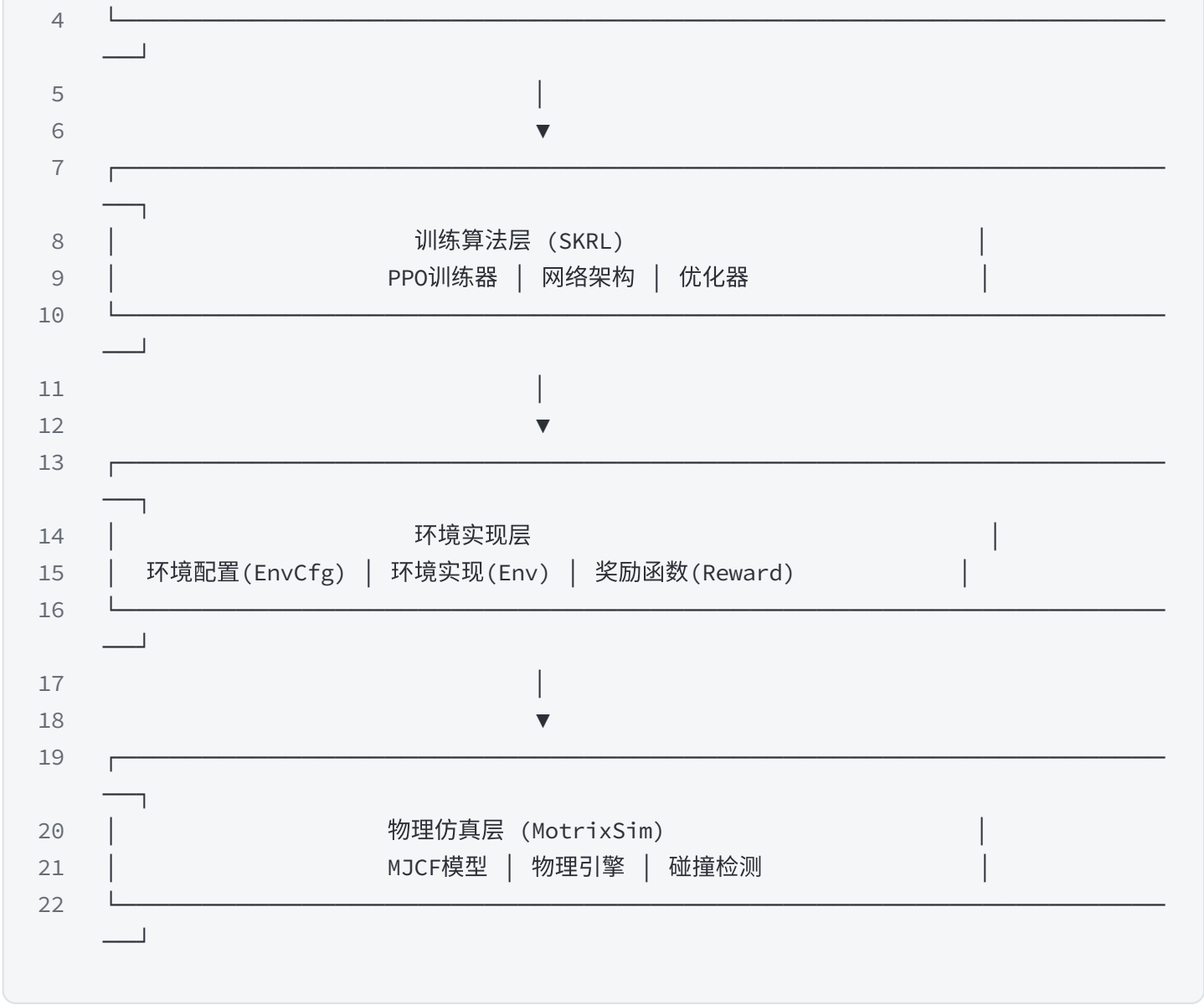
代码块

```
MotrixLab/
├── motrix_envs/                  # 环境层：物理仿真和任务定义
│   ├── basic/                     # 基础环境（cartpole、walker等）
│   ├── locomotion/                # 运动环境（GO1机器人等）
│   ├── np/                        # NumPy仿真后端框架
│   ├── base.py                    # 环境基类
│   └── registry.py                # 环境注册系统
├── motrix_rl/                   # 训练层：RL算法和配置
│   ├── skrl/                      # SKRL框架集成（JAX/PyTorch）
│   ├── base.py                    # RL配置基类
│   └── registry.py                # RL配置注册系统
└── scripts
    ├── train.py                   # 训练入口脚本
    ├── play.py                    # 测试入口脚本
    └── view.py                    # 可视化脚本
```

核心组件架构

代码块

```
┌─────────────────────────────────────────────────────────────────────
──┐
│                    用户接口层                                  │
│              train.py │ play.py │ view.py                          │
```



核心组件详解

1.1 训练环境 (Training Environment)

位置：环境实现层

训练环境是 MotrixLab 的核心组件，包含三个关键部分：

- **环境配置 (EnvCfg)**：定义物理仿真参数（模型文件、时间步长、episode 长度等）和任务特定参数
- **环境实现 (Env)**：继承基础环境类，实现具体的任务逻辑、物理仿真交互和终止条件检查
- **奖励函数 (Reward)**：在环境的 step 方法中实现，根据当前状态和动作计算奖励值

环境通过装饰器注册到系统中。

1.2 奖励函数 (Reward Function)

位置：配置管理层 + 环境实现层

奖励函数在 MotrixLab 中采用双重结构设计：

- **配置层面**：在配置类中定义奖励权重、奖励组件类型和缩放参数
- **实现层面**：在环境的 `_compute_reward` 方法中根据配置参数计算具体奖励值

这种设计使得奖励函数既可以通过配置文件灵活调整，又能在代码中实现复杂的计算逻辑。

1.3 配置参数 (Configuration Parameters)

位置：配置管理层

配置参数采用分层管理结构：

- **环境配置 (EnvCfg)**：控制物理仿真和任务行为，包括仿真参数、重置噪声、时间限制等
- **训练配置 (RLCfg)**：控制强化学习算法，包括网络结构、学习率、批次大小、训练步数等

配置类支持继承、参数验证和运行时覆盖，确保参数的合理性和灵活性。

1.4 注册系统 (Registry System)

位置：连接各组件的枢纽

注册系统通过装饰器模式实现组件的自动注册：

- 环境配置类通过 `@registry.envcfg()` 注册
- 环境实现类通过 `@registry.env()` 注册，支持多后端
- RL 配置类通过 `@rlcfg()` 注册

注册系统实现了组件的解耦，使得新增环境或修改配置变得简单快捷。

数据流和工作流程

训练流程概览

```
代码块
1  用户命令 → 配置解析 → 环境创建 → 训练循环 → 模型保存
2     ↓
3  train.py --env cartpole
4     ↓
5  查找配置类 → 创建环境 → 启动PPO训练 → 保存模型
```

核心工作流程

1. **环境定义**：在 `/motrix_envs/` 中创建环境配置类和实现类
2. **自动注册**：通过装饰器将组件注册到系统中
3. **配置加载**：命令行启动时，系统自动查找并加载对应的配置
4. **环境创建**：工厂模式创建环境实例，支持参数覆盖

5. **训练执行**：PPO 算法与环境交互，收集数据并更新策略
6. **结果保存**：定期保存检查点和最终模型

配置参数的作用

配置参数在整个流程中起到关键的连接作用：

- **环境配置**决定物理仿真行为（时间步长、模型文件、噪声等）
- **奖励配置**影响学习信号（奖励权重、计算方式等）
- **训练配置**控制算法行为（网络结构、学习率、批次大小等）

多后端支持

MotrixLab 的分层设计天然支持多种后端：

- **仿真后端**：MotrixSim
- **训练后端**：JAX 和 PyTorch，支持 GPU 加速
- **算法框架**：主要集成 SKRL，易于扩展其他算法

2. 第一阶段地图使用方法

1. xml文件说明
   a. `0131_V_section00.xml` - 视觉模型（有灯光、材质、纹理）
   b. `0131_C_section01.xml` - 碰撞模型
   c. `0126_CP_section01.xml` - 检查点模型
   d. `scene_section001.xml` - vbot 超能机械狗在 section001 地形导航场景的核心配置文件
   e. `vbot.xml` - vbot 超能机械狗模型
2. 通过xml文件构建配置类和环境类

> cfg.py中的配置类（VBotSection001EnvCfg）
>
> ↓ 通过 @registry.envcfg("vbot_navigation_section001") 注册
>
> ↓
>
> 环境工厂根据配置创建实例
>
> ↓
>
> vbot_section001_np.py中的环境类（VBotSection001Env）
>
> ↓ 通过 @registry.env("vbot_navigation_section001", "np") 注册
>
> ↓
>
> 强化学习训练/测试

使用案例：

1. 将navigation文件放到和locomotion同一级目录下。
2. 在 `motrix_envs/src/motrix_envs/navigation/vbot/cfg.py` 新建一个配置类VBotSection001EnvCfg。

配置类信息（仅供参考）：

代码块

```python
@registry.envcfg("vbot_navigation_section001")
#通过 @registry.envcfg("vbot_navigation_section001") 注册
@dataclass
class VBotSection001EnvCfg(VBotStairsEnvCfg):
    """VBot Section01单独训练配置 - 高台楼梯地形"""
    model_file: str = os.path.dirname(__file__) + "/xmls/scene_section001.xml"
    max_episode_seconds: float = 40.0  # 拉长一倍：从20秒增加到40秒
    max_episode_steps: int = 4000  # 拉长一倍：从2000步增加到4000步

    @dataclass
    class InitState:
        # 起始位置：随机化范围内生成
        pos = [0.0, -2.4, 0.5]  # 中心位置

        pos_randomization_range = [-0.5, -0.5, 0.5, 0.5]  # X±0.5m, Y±0.5m随机

        default_joint_angles = {
            "FR_hip_joint": -0.0,
            "FR_thigh_joint": 0.9,
            "FR_calf_joint": -1.8,
            "FL_hip_joint": 0.0,
            "FL_thigh_joint": 0.9,
            "FL_calf_joint": -1.8,
            "RR_hip_joint": -0.0,
            "RR_thigh_joint": 0.9,
            "RR_calf_joint": -1.8,
            "RL_hip_joint": 0.0,
            "RL_thigh_joint": 0.9,
            "RL_calf_joint": -1.8,
        }

    @dataclass
    class Commands:
        # 目标位置：缩短距离，固定目标点
        # 起始位置Y=-2.4, 目标Y=3.6, 距离=6米（与vbot_np相近）
        # pose_command_range = [0.0, 3.6, 0.0, 0.0, 3.6, 0.0]

```

```
        # 原始配置（已注释）：
        # 目标位置：固定在终止角范围远端（完全无随机化）
        # 固定目标点：X=0, Y=10.2, Z=2 (Z通过XML控制)
        # 起始位置Y=-2.4, 目标Y=10.2, 距离=12.6米
        pose_command_range = [0.0, 10.2, 0.0, 0.0, 10.2, 0.0]

    @dataclass
    class ControlConfig:
        action_scale = 0.25

    init_state: InitState = field(default_factory=InitState)
    commands: Commands = field(default_factory=Commands)
    control_config: ControlConfig = field(default_factory=ControlConfig)
```

3. 在 `motrix_rl/src/motrix_rl/cfgs.py` 新建一个配置类VBotSection001EnvCfg

参考：

代码块

```
@registry.envcfg("vbot_navigation_stairs")
@dataclass
class VBotStairsEnvCfg(VBotEnvCfg):
    """VBot在楼梯地形上的导航配置，继承flat配置"""
    model_file: str = os.path.dirname(__file__) + "/xmls/scene_stairs.xml"
    max_episode_seconds: float = 20.0  # 增加到20秒，给更多时间学习转向
    max_episode_steps: int = 2000

    @dataclass
    class ControlConfig:
        action_scale = 0.25  # 楼梯navigation使用0.2，足够转向但比平地更谨慎

    control_config: ControlConfig = field(default_factory=ControlConfig)

```

4. 新建一个vbot_section001_np.py文件夹，此文件夹和 `motrix_envs/src/motrix_envs/navigation/vbot/cfg.py` 在同级目录下。构建方式参考anymal_c文件中的构造方式中：

形如：

代码块

```
@registry.env("vbot_navigation_section001", "np")
class VBotSection001Env(NpEnv):
    """
```

```
    VBot在楼梯地形上的导航任务
    独立实现，不继承VBotEnv
    """
    _cfg: VBotSection01EnvCfg

    def __init__(self, cfg: VBotSection01EnvCfg, num_envs: int = 1):
        # 调用父类NpEnv初始化
        super().__init__(cfg, num_envs=num_envs)

        # 初始化机器人body和接触
        self._body = self._model.get_body(cfg.asset.body_name)
        self._init_contact_geometry()

        # 获取目标标记的body
        self._target_marker_body = self._model.get_body("target_marker")

        # 获取箭头body（用于可视化，不影响物理）
        try:
            self._robot_arrow_body =
self._model.get_body("robot_heading_arrow")
            self._desired_arrow_body =
self._model.get_body("desired_heading_arrow")
        except Exception:
            self._robot_arrow_body = None
            self._desired_arrow_body = None

           <省略以下代码>
         ......................
         ......................
```

5. 在 `motrix_envs\src\motrix_envs\navigation\vbot\__init__.py` 中导入 `vbot_section001_np` 和 `VBotSection001Env`，`VBotSection001EnvCfg`。
6. 运行 `uv run scripts/train.py --env vbot_navigation_section001`

出现下述图像即为**成功!**

目标点信息获取

1. 在`0131_V_section00.xml`中给了`<body name="B_body" pos="0.0 0.0 0.0"`，目标pose即为[0,0,0]。