

基于Isaaclab的人形机器人仿真与强化学习训练

赵梓霖 吉林大学软件学院

2025年浙江大学软件学院夏令营 | 大模型与工业智能分营 | 题目2

开源代码，实验数据，视频展示: <https://github.com/sssn-tech/Learn2Walk>

目录

Task0：Isaaclab环境配置

- 常见坑点
- 软件安装

Task1：基础环境搭建

- 要求
- 完成情况
- 关键代码（随机动作部分）

Task2：复现宇树机器人行走

- 要求
- 完成情况

Task3：跨平台步态迁移

- 要求
- 完成情况
- 更多思考

Task0: Isaaclab环境配置

常见坑点

Isaaclab的初步环境搭建就涉及到非常多的坑点，这里列举一些。

这些坑点的主要原因是Isaaclab的官方文档将Isaacsim（Isaaclab的基础环境）的安装一笔带过了，官方给出了一个Isaacsim安装命令，但是没有作出解释和明确需求。

我强烈建议所有尝试安装Isaaclab的用户对照检查自身环境情况。逐点对照参考Isaacsim的官方需求文档：

```
https://docs.isaacsim.omniverse.nvidia.com/latest/installation/requirements.html
```

- Next, install a CUDA-enabled PyTorch 2.5.1 build based on the CUDA version available on your system. This step is optional for Linux, but required for Windows to ensure a CUDA-compatible version of PyTorch is installed.

```
pip install torch==2.5.1 torchvision==0.20.1 --index-url https://download.pytorch.org
```

- Before installing Isaac Sim, ensure the latest pip version is installed. To update pip, run

```
pip install --upgrade pip
```

- Then, install the Isaac Sim packages.

```
pip install 'isaacsim[all,extscache]==4.5.0' --extra-index-url https://pypi.nvidia.com
```

Verifying the Isaac Sim installation

Isaacsim的安装有比较苛刻的条件
远没有一行命令这么简单

- Make sure that your virtual environment is activated (if applicable)
- Check that the simulator runs as expected:

↑ Back to top

尝试在AutoDL这样的容器算力平台上搭建

尽管AutoDL官方没有告诉用户（文档里也没有），AutoDL的所有环境均存在于Docker之内，而不是完整的虚拟机。

Isaac Sim作为Isaaclab的必要基础，依赖于CUDA、OptiX、RTX等组件，必须要求：

- 宿主机支持 NVIDIA GPU 驱动
- Docker 容器使用 `--gpus all` 并挂载 `/dev/nvidia*`、`nvidia-container-runtime` 等
- 安装特定版本的 NVIDIA 驱动、CUDA 工具包等

AutoDL 平台虽然支持 GPU，但不允许控制底层 NVIDIA 驱动和容器运行时配置，也不暴露相关接口。如果你尝试在AutoDL平台上安装Isaacsim，是可以成功安装的（无论是二进制，还是 `pip install`）。但如果你尝试运行，会提示缺少驱动接口，或者缺少关键驱动文件。这通常可以在宿主环境内为Docker挂载，但AutoDL显然不允许用户访问宿主机。

Note

尽管Isaaclab可以使用 `--headless` 参数取消渲染，只运行仿真功能，但这个参数没有完全取消他对于GPU接口的底层依赖。因此，这个参数无法使它可以运行的环境要求放宽。这个参数的优化空间也许很大：目前仓库里还有弱交互，非交互式脚本的仿真脚本不能通过 `--headless` 运行。

Note

Isaaclab是一个比较新的产品，社区也仍在构建中。上述问题反映了Isaaclab在描述自身安装需求时候存在部分不清晰。在后面你还会看到，我遇到了**Isaaclab官方脚本不完善，和宇树官方脚本运行错误**的问题。需要提出Issue，和开发者直接沟通。

尝试在Nvidia Tesla T4或者类似的GPU上搭建

Isaaclab必须RTX技术来支持底层功能，这在Tesla T4上行不通。同样的，`--headless` 参数不能解决问题。

尝试在本地/远程的纯净Ubuntu上搭建

准备面对在Linux上安装Nvidia驱动这个被喷了无数年的，经久不衰的话题

- 假酒驱动，`nvidia-smi` 正常，但是用不了，知乎 <https://zhuanlan.zhihu.com/p/19148652598>
- 换驱动导致暴毙，知乎 <https://zhuanlan.zhihu.com/p/6518449814>
- 安全启动毙掉驱动，Nvidia官方论坛 <https://forums.developer.nvidia.com/t/nvidia-drivers-not-working-while-secure-boot-is-enabled-after-updating-to-ubuntu-24-04/305351>
- Mock注册了，但是无效
- 装好了Nvidia驱动，重启一下发现网卡驱动挂了

你可能在想：

- 英伟达官网搜索显卡型号，下载 `run.sh` 离线安装
- 依照自动建议安装 `sudo ubuntu-drivers autoinstall`
- 在英伟达官网查表安装 `sudo apt install nvidia-driver-525`
- 更多看似规范，看似官方的安装

在经历长久的折磨后，我认为这个话题在互联网上被长久讨论是有自己的原因的，人生苦短，请找个成品驱动镜像用。



Reddit · r/linuxquestions

50+ 条评论 · 2年前

⋮

Why is it still so hard to install drivers for Nvidia?

Nvidia is much more common than AMD in laptops, yet **installing the drivers is a PITA**. Only Ubuntu makes it easy, and I want to move off of Ubuntu to Fedora.

[What is the proper way of installing drivers on ...](#) 12 个帖子 2021年10月12日

[Why are drivers so annoying on linux and is there ...](#) 14 个帖子 2021年11月19日

[www.reddit.com站内的其它相关信息](#)



Quora

3个回答 · 6年前

⋮

Why is graphics driver installation under Linux so much ...

There are multiple reasons but the main one is **lack of ABI stability compared to Windows**. Each new Linux kernel and xorg release will very likely break ...

[Why is it so hard to find drivers for Linux, even after ...](#) 3 个回答 2025年7月2日

[What makes it challenging to install a WiFi driver on ...](#) 2 个回答 2024年1月29日

[www.quora.com站内的其它相关信息](#)



NVIDIA Developer Forums

<https://forums.developer.nvidia.com> · why... · 翻译此页

⋮

Why is it so hard to install CUDA on linux

2012年5月10日 — **The kernels are changing so fast it is not easy to keep pace with it.** The safest option is not to update kernel or distribution until it is sure ...

小结

以上是一些Isaaclab安装环境的常见坑点，我个人认为官方可以出一个小巧的脚本，用来检查当前所处的环境是否满足安装要求。让用户不必花费大量时间下载，部署，然后在运行过程中报错崩溃。或者最少的，也应该在官方文档中，Isaacsim的安装命令下注释需求文档地址。

软件安装

示例配置

从这里开始，我开始使用阿里云ECS作为试验环境。实例规格 `ecs.gn7i-c32g1.8xlarge`：

- GPU: Nvidia A10
- CPU: 32vCPU Intel(R) Xeon(R) Platinum 8369B CPU @ 2.90GHz
- 内存: 32G, 实际上过剩了
- 镜像: 云市场镜像->搜索驱动+Ubuntu->选择v3

镜像市场[华东1 (杭州)]

X

The screenshot shows a search interface for Docker images. The search bar has '驱动' (Driver) selected. Below it, there are dropdown filters for 'Ubuntu' (set to 'Ubuntu'), 'x86-64位操作系统' (set to 'x86-64位'), and '使用人数' (set to '1'). A red box highlights the first search result, which is 'Ubuntu 22.04 64位预装NVIDIA GPU 550.90.07驱动镜像'. This entry includes details like '基础系统: linux 架构: x86-64 最新更新: 2025-07-02', version '550.90.07', 1930 users, and a rating of 0 stars. A red annotation on the right side of the page says '这个相对好用，但不是通用，Tesla T4就用不了' (This is relatively good, but not universal, Tesla T4 cannot use it). Other results listed include Python environments, a Websoft9 developer platform, and a specific Ubuntu 20.04 image.

(i) Note

阿里云ECS能跑的本质原因是他是虚拟机，而不是Docker容器。

! Caution

不能觉得阿里驱动市场的镜像一定work，开机挂掉的比比皆是，但我挑的这个驱动和容器规格肯定没问题

安装流程

| 安装Isaacsim和Issaclab

阅读到这里，你应该已经可用的运行环境，这个环境

- GPU支持RTX，CUDA版本较新
- Nvidia驱动正常，不仅可以显示版本，还能往GPU迁移数据（可以参见上面的“假驱动”帖子）
- 级别为虚拟机，或者物理机

现在你可以按照Issaclab官方文档的流程来安装软件，并尝试运行

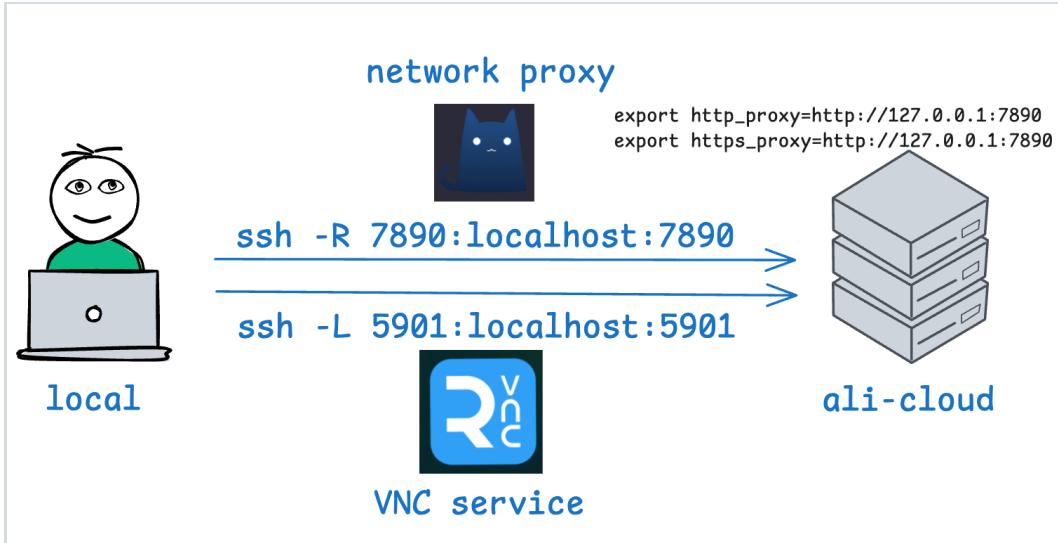
1. 安装Miniconda
2. 创建一个python 3.10环境
3. 安装Isaacsim
 - 如果你是Ubuntu 22.04，你应该用 `pip` 安装
 - 如果你是Ubuntu 20.04，你应该用二进制安装
4. 安装Issaclab
 1. `git clone` 官方仓库
 2. `conda activate [你的环境]`
 3. `chmod +x Issaclab.sh && Issaclab.sh --install`

可以参照官方文档，文档已经有的流程就不赘述了

```
https://isaac-sim.github.io/IsaacLab/main/source/setup/installation/index.html
```

| 更多配置

为了更方便的观测试验运行情况，你最好安装远程桌面软件，并配置网络代理。我在这里给出一个可行的解决方案：



① Caution

Isaaclab的很多脚本会调用网络资源，这些资源位于 <http://omniverse-content-production.s3-us-west-2.amazonaws.com/Assets/Isaac/>，网络代理是必须的

使用SSH隧道搭建两个核心服务：网络代理和VNC桌面的通道，用 curl 简要测试代理运行情况

```
(base) ecs-user@iZuf6g6vii5ci3ikxssy9lZ:~$ source ~/proxy-on.sh 7899
127.0.0.1:7899
(base) ecs-user@iZuf6g6vii5ci3ikxssy9lZ:~$ curl google.com
<HTML><HEAD><meta http-equiv="content-type" content="text/html; charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

Task1: 基础环境搭建

到这一步，你应该已经安装并测试好了IsaacLab

要求

1. 场景构建：

- 使用Isaac Sim创建 $10m \times 10m$ 的室内平面场景（可选：增加障碍物/斜坡）
- 添加1个人形机器人（例如宇树H1/Go2等）
- 部署基础光源和物理材质

2. 运动控制测试

- 通过Isaac Lab的Python API控制机器人完成基础动作
- 输出机器人完成站立动作的5秒关节角度变化曲线图

完成情况

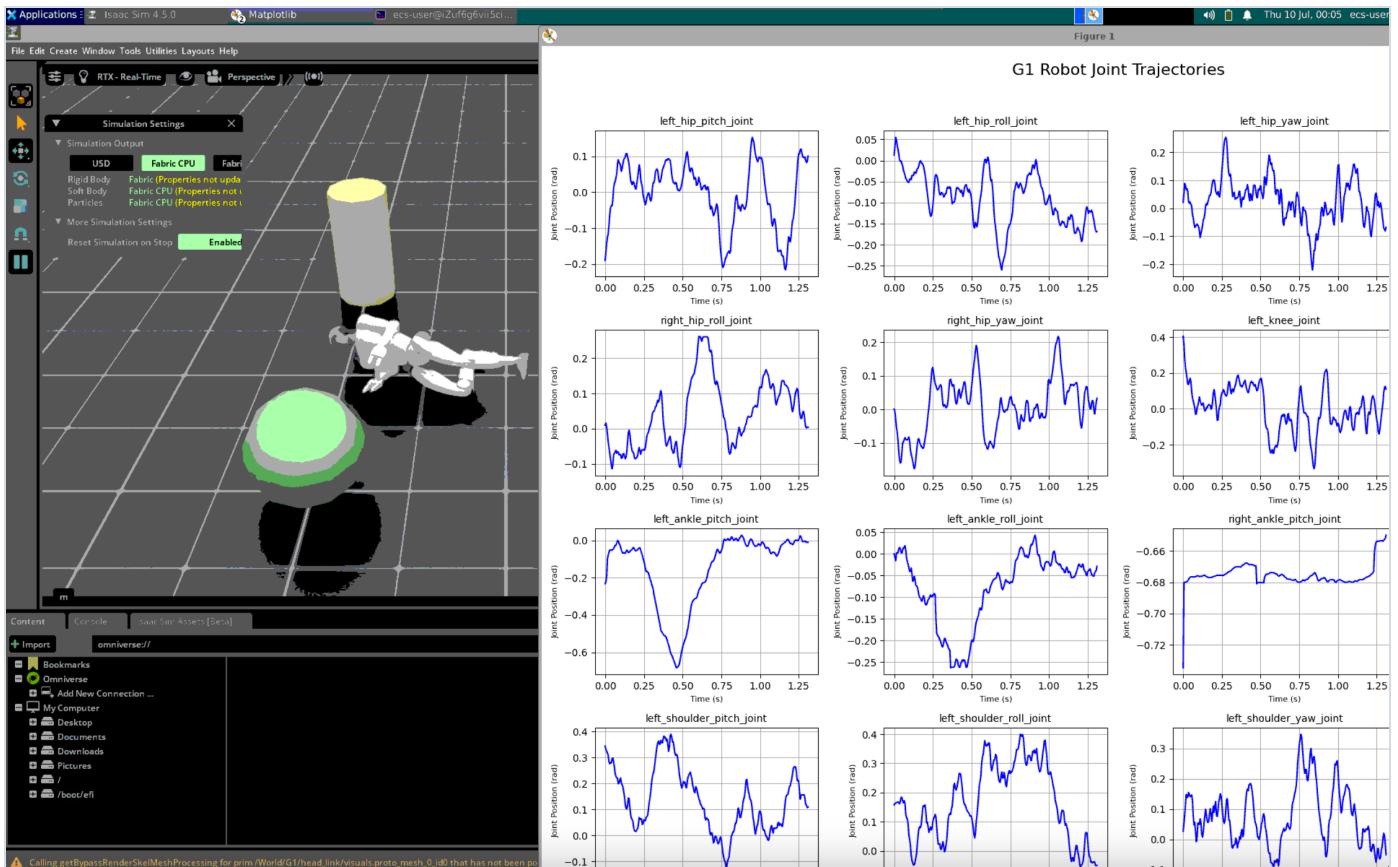
• 全部完成，源码请参考仓库

- 基础环境内含几个基本几何物体，机器人使用G1，有默认光源和材质
- 开始仿真后，机器人单脚独立，全身执行小幅度的随机动作，“抽搐”着倒下
- 记录重要关节的弧度变化并打印成折线图

• 扩展内容：演示视频的录制

- 在一个单独的 .py 脚本中录制视频竟然如此的难，调试难度可以排整个项目的前三
- 大模型的记忆都是老api，调教了一整天都无法从默认视角中抓取出照片
- 官方文档中的 `beta: StageRecorder[1]` 用不了，未知原因，录制出的文件是空的
- 官方文档中的 `ReplicatorBasicWriter[2]` 可以用，藏的有点深了，找了很久很久

```
[1] https://isaac-sim.github.io/IsaacLab/main/source/how-to/record\_animation.html
[2] https://isaac-sim.github.io/IsaacLab/main/source/how-to/save\_camera\_output.html
```



关键代码(随机动作部分)

```

for i, name in enumerate(joint_names):
    set = False
    # 跳过右腿关节 (pitch方向) 保持踢腿动作
    if any(key in name for key in ["right_hip_pitch_joint", "right_knee_joint",
"right_ankle_pitch_joint"]):
        continue # 不修改踢腿姿态
    # 匹配上肢关节
    for key, (low, high) in upper_body_joint_ranges.items():
        if key in name:
            joint_pos[:, i] = torch.rand(1, device=sim.device) * (high - low) + low
            set = True
            break
    # 匹配下肢关节
    if not set:
        for key, (low, high) in lower_body_joint_ranges.items():
            if key in name:
                joint_pos[:, i] = torch.rand(1, device=sim.device) * (high - low) +
low
                set = True
                break
    # 如果没有匹配到任何设置, 则使用默认小幅扰动
    if not set:
        joint_pos[:, i] = torch.rand(1, device=sim.device) * 0.2 - 0.1 # [-0.1, 0.1]

```

Task2: 复现宇树机器人行走

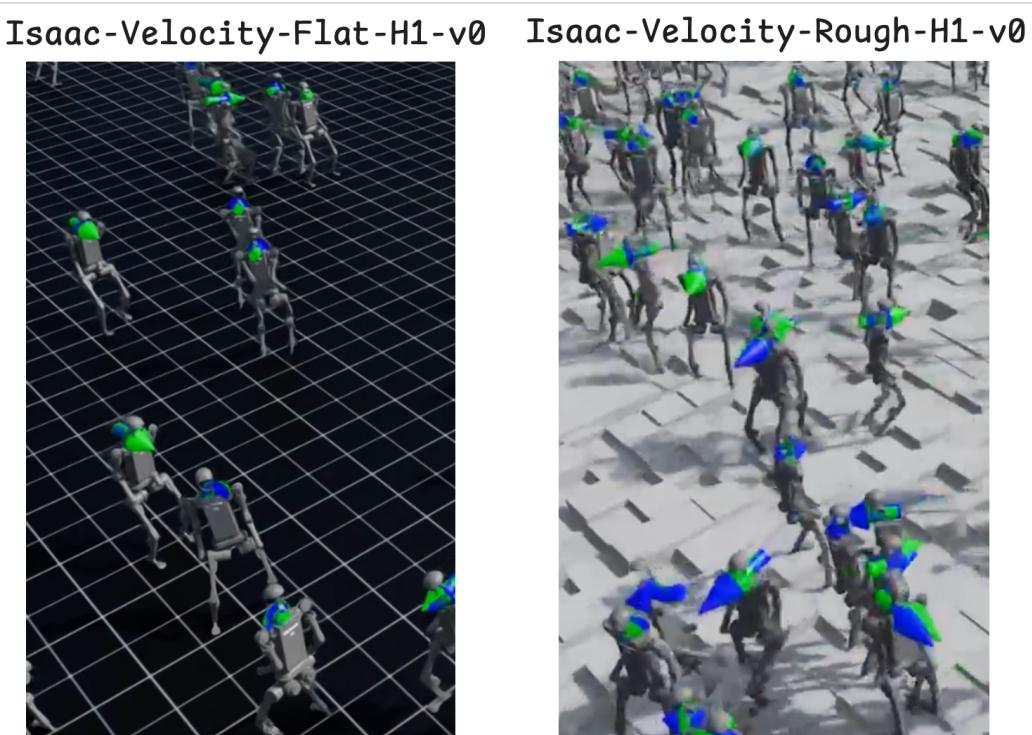
要求

目标：基于宇树开源代码实现H1机器人行走策略复现

1. 代码部署与训练
 - 使用宇树官方提供的强化学习示例
 - 解析奖励函数设计：速度跟踪奖励；步态对称性惩罚等

完成情况

- 全部完成，其中主要复现两部分实验
 - 宇树官方仓库：https://github.com/unitreerobotics/unitree_rl_lab
 - IsaacLab的新版本也包含宇树机器人行走：<https://isaac-sim.github.io/IsaacLab/main/source/overview/environments.html#locomotion>
- 扩展内容：发现了一个宇树官方代码的bug，提交issue并解决
- 扩展内容：发现了一个IsaacLab官方代码的bug，提交issue并解决



拓展内容

⚠ Caution

宇树官方仓库的代码存在的训练数值崩溃的问题

```

...
File "/home/ecs-user/miniconda3/envs/is/lib/python3.10/site-
packages/torch/distributions/normal.py", line 73, in sample
    return torch.normal(self.loc.expand(shape), self.scale.expand(shape))
RuntimeError: normal expects all elements of std >= 0.0

```

这是因为在 PyTorch 中，`torch.distributions.Normal(loc, scale)` 是一个正态分布，其 `scale`（标准差）必须是非负数。

但是模型中 `scale` 出现了负值或 NaN，导致 `torch.normal(...)` 抛出异常。

这可能和梯度爆炸，数值不稳定，没有剪裁梯度有关。

我提交了 `issue[1]`，收到回复说问题在最近一次的commit修复了。

- 我在7-7克隆的代码，在7-9遇到的问题
- 修复的commit日期是7-8，可以说是非常巧了

```
[1] https://github.com/unitreerobotics/unitree_rl_lab/issues/7
```

① Caution

Isaaclab官方代码中，教程脚本有点小毛病

```

(is) ecs-user@izuf6g6vii5ci3ikxssy9lZ:~/IsaacLab$ ./isaaclab.sh -p
scripts/demos/h1_locomotion.py --headless
...
Traceback (most recent call last):
  File "/home/ecs-user/IsaacLab/scripts/demos/h1_locomotion.py", line 50, in <module>
    from omni.kit.viewport.utility import get_viewport_from_window_name
ModuleNotFoundError: No module named 'omni.kit.viewport'

```

遇到问题的时候没什么想法，提交了 `Issue[1]`

- 收到回复说这个教程脚本是交互式的，不能添加 `--headless` 参数
- 但是所有同一目录下的脚本都可以 `--headless`，开发者说会添加一个报错，提醒这个脚本不能加

```
[1] https://github.com/isaac-sim/IsaacLab/issues/2858
```

奖励函数设计

奖励函数的代码位于：

```
source/unitree_rl_lab/unitree_rl_lab/tasks/locomotion/robots/h1/velocity_env_cfg.py
238-338行
```

每一项奖励都是一个 `RewTerm` 实例，它包含：

- `func`: 奖励函数，实现具体的行为评估逻辑
- `weight`: 权重，表示该项奖励对总奖励的影响程度（正值鼓励、负值惩罚）
- `params`: 可选参数，用于配置奖励函数的行为

总共有这些实例：

| 任务相关的奖励：奖励完成任务（跟随目标速度）

track_lin_vel_xy

- 目的：鼓励机器人在XY平面上追踪目标线速度。
- 权重：1.0（较高，表明这是核心任务）

track_ang_vel_z

- 目的：鼓励机器人跟踪目标的角速度（绕Z轴旋转）。
- 权重：0.5（次重要）

alive

- 目的：鼓励机器人保持“存活”（例如不摔倒、维持平衡）。
- 权重：0.15（维持基本稳定性）

| 基础运动的奖励（运动状态相关）

base_linear_velocity

- 目的：惩罚垂直方向上的速度（Z轴），鼓励稳定。
- 权重：-2.0（惩罚力度较大）

base_angular_velocity

- 目的：惩罚绕XY轴的角速度，避免机器人剧烈晃动。
- 权重：-0.5

joint_acc

- 目的：惩罚关节加速度，鼓励平滑运动。
- 权重：-2.5e-7（非常小，次要惩罚）

action_rate

- 目的：惩罚动作变化率，鼓励连续平滑的动作。
- 权重：-0.05

dof_pos_limits

- 目的：惩罚关节位置超出限制。
- 权重：-5.0（较高惩罚）

| 关节偏差奖励（惩罚不合理的关节）

joint_deviation_arms

- 目标：肩膀、肘部等手臂关节
- 权重：-1.0
- 正则表达式：匹配肩、肘关节的名字

joint_deviation_torso

- 目标：躯干关节
- 权重：-1.0

joint_deviation_hips

- 目标：臀部连接的髋关节
- 权重：-1.0

| 机器人姿态奖励：鼓励直立，惩罚水平

flat_orientation_l2

- 目的：惩罚机器人姿态偏离水平（例如倾斜）
- 权重：-1.0

base_height

- 目的：鼓励机器人维持一个目标高度（0.9米）
- 权重：-10.0（非常重要）

| 足部奖励

gait

- 目的：鼓励机器人形成自然步态（如对称步频）
- 权重：0.5

feet_slide

- 目的：惩罚脚接触地面时滑动，鼓励稳定着地
- 权重：-0.2

feet_clearance

- 目的：鼓励抬脚时有一定高度，防止绊倒
- 权重：20.0（非常高，重要性大）

feet_contact_forces

- 目的：惩罚异常的大脚部接触力（可能表示撞击）

- 权重: -0.0002 (惩罚轻微)

| 其他

undesired_contacts

- 目的: 惩罚除脚以外的身体与地面接触 (例如手、膝盖碰地) 。
- 权重: -1.0 (防止摔倒等)

Task3：跨平台步态迁移

要求

目标：将训练策略迁移至其他人形机器人

模型适配

- 导入第三方机器人URDF文件（需对齐关节命名规范，如*_hip_pitch）
- 调整动力学参数：例如关节力矩限制（第三方机器人电机扭矩可能低于宇树H1）/腿部质量分布

完成情况

- 全部完成**
- 将上文中训练好的宇树H1行走策略迁移至宇树G1上
 - 使用代码探针分析H1和G1的关节，对照分析.usd文件
 - 对于输出（动作），建立H1到G1的关节映射表
 - 对于输入（观测），重排G1的关节顺序，对齐H1策略的输入维度和顺序
 - 修改G1的设置（初始站姿，力矩）适配H1策略
 - 流程：G1观测->转换为H1观测->输入H1策略->得到H1动作（关节动作表）->映射为G1动作

使用代码探针分析机器人建模，得到动作映射表

我们可以找到

```
source/isaaclab_tasks/isaaclab_tasks/manager_based/locomotion/velocity/velocity_env_c
fg.py
109行开始
...
@configclass
class ActionsCfg:
    """Action specifications for the MDP."""
    joint_pos = mdp.JointPositionActionCfg(asset_name="robot", joint_names=[".*"],
    scale=0.5, use_default_offset=True)
```

这里定义了动作输出时所有关节。为了提高扩展性，这个定义几乎没有我们可以参考的地方，我们要使用代码探针分析运行时环境的构造

① Note

选择rsl_rl的理由只是因为大部分环境都提供了它的训练入口

修改rsl库的训练代码，他位于：

```
scripts/reinforcement_learning/rsl_rl/train.py
```

加入代码探针，这是一个运行时参数 `--extract`，代表不训练，而是分析环境内容

这个参数：

- 在嵌套的环境中递归搜索名为名为 `robot` 的对象，如果找到，打印它的关节顺序，这将用来对齐动作空间（网络输出）
- 提取一次观测，即 `observation`，打印形状，这将用来对齐观测空间（网络输入）

```

...
# 新增：提取信息的操作
parser.add_argument("--extract", action="store_true", default=False, help="Extract
joint names & observations and exit.")

...
# 如果启用了 --extract 参数，则只提取关节并退出
if args_cli.extract:
    def find_robot(env): #在嵌套环境中递归的寻找名为robot的对象
        max_depth = 10
        for _ in range(max_depth):
            if hasattr(env, "scene"):
                if hasattr(env.scene, "robot"):
                    return env.scene.robot
                if hasattr(env.scene, "articulations") and "robot" in
env.scene.articulations:
                    return env.scene.articulations["robot"]
            if hasattr(env, "envs"):
                env = env.envs[0]
            elif hasattr(env, "env"):
                env = env.env
            elif hasattr(env, "unwrapped"):
                env = env.unwrapped
            else:
                break
        raise AttributeError("Could not find robot in environment.")

    robot = find_robot(env)
    joint_names = robot.joint_names

    print("Extracted joint names:")
    for name in joint_names: # 找到的robot，打印关节顺序
        print(name)

```

使用探针可以分析得到H1和G1的关节全称，其中H1有19个关节，G1有37个，分别是

```

h1_joint_names = ["left_hip_yaw", "right_hip_yaw", "torso", "left_hip_roll",
"right_hip_roll", "left_shoulder_pitch", "right_shoulder_pitch", "left_hip_pitch",
"right_hip_pitch", "left_shoulder_roll", "right_shoulder_roll", "left_knee",
"right_knee", "left_shoulder_yaw", "right_shoulder_yaw", "left_ankle", "right_ankle",
"left_elbow", "right_elbow"]

g1_joint_names = ["left_hip_pitch_joint", "right_hip_pitch_joint", "torso_joint",
"left_hip_roll_joint", "right_hip_roll_joint", "left_shoulder_pitch_joint",
"right_shoulder_pitch_joint", "left_hip_yaw_joint", "right_hip_yaw_joint",
"left_shoulder_roll_joint", "right_shoulder_roll_joint", "left_knee_joint",
"right_knee_joint", "left_shoulder_yaw_joint", "right_shoulder_yaw_joint",
"left_ankle_pitch_joint", "right_ankle_pitch_joint", "left_elbow_pitch_joint",
"right_elbow_pitch_joint", "left_ankle_roll_joint", "right_ankle_roll_joint",
"left_elbow_roll_joint", "right_elbow_roll_joint", "left_five_joint",
"left_three_joint", "left_zero_joint", "right_five_joint", "right_three_joint",
"right_zero_joint", "left_six_joint", "left_four_joint", "left_one_joint",
"right_six_joint", "right_four_joint", "right_one_joint", "left_two_joint",
"right_two_joint"]

```

Important

上面的关节顺序表是有序的！他的顺序和策略输出维度一一对应

我们再将模型的结构打印出来，验证上文得到的关节数目

H1的Actor网络

```

Actor MLP: Sequential(
    (0): Linear(in_features=69, out_features=128, bias=True)
    (1): ELU(alpha=1.0)
    (2): Linear(in_features=128, out_features=128, bias=True)
    (3): ELU(alpha=1.0)
    (4): Linear(in_features=128, out_features=128, bias=True)
    (5): ELU(alpha=1.0)
    (6): Linear(in_features=128, out_features=19, bias=True)
)

```

G1的Actor网络

```

Actor MLP: Sequential(
    (0): Linear(in_features=123, out_features=256, bias=True)
    (1): ELU(alpha=1.0)
    (2): Linear(in_features=256, out_features=128, bias=True)
    (3): ELU(alpha=1.0)
    (4): Linear(in_features=128, out_features=128, bias=True)
    (5): ELU(alpha=1.0)
    (6): Linear(in_features=128, out_features=37, bias=True)
)

```

可以看到输出维度是19和37，这与我们得到的关节数目匹配，说明我们的关节数目没问题。

ⓘ Note

打印网络不仅可以得到输出维度（关节数目），还得到了输入维度（观测维度），本节主要是分析关节的对齐，观测的对齐在下一节阐述

现在我们可以再确认一下关节结构，并顺带获取机器人的.usd建模文件。

我用到一个有趣的工程技巧：IsaacLab的官方仓库里没有机器人的.usd文件，但他能正常完成训练，这说明他大概率调用了网络上的资源。我们可以手动断网，清理缓存，然后运行训练，等待连接超时报错：

```
(is) ecs-user@iZuf6g6vii5ci3ikxssy9lZ:~/IsaacLab$ source ~/proxy-on.sh 9999 # 设置一个  
不存在的代理  
127.0.0.1:9999  
(is) ecs-user@iZuf6g6vii5ci3ikxssy9lZ:~/IsaacLab$ curl baidu.com # 确认一下网络是否真的  
失效了  
curl: (7) Failed to connect to 127.0.0.1 port 9999 after 0 ms: Connection refused  
(is) ecs-user@iZuf6g6vii5ci3ikxssy9lZ:~/IsaacLab$ python  
scripts/reinforcement_learning/rsl_rl/train.py --task=Isaac-Velocity-Flat-G1-v0 --  
headless  
...  
File "/home/ecs-  
user/IsaacLab/source/isaaclab/isaaclab/sim/spawners/from_files/from_files.py", line  
66, in spawn_from_usd  
    return _spawn_from_usd_file(prim_path, cfg.usd_path, cfg, translation,  
orientation)  
    File "/home/ecs-  
user/IsaacLab/source/isaaclab/isaaclab/sim/spawners/from_files/from_files.py", line  
237, in _spawn_from_usd_file  
    raise FileNotFoundError(f"USD file not found at path: '{usd_path}'")  
FileNotFoundError: USD file not found at path: 'http://omniverse-content-  
production.s3-us-west-  
2.amazonaws.com/Assets/Isaac/4.5/Isaac/IsaacLab/Robots/Unitree/G1/g1_minimal.usd'.
```

在等待数分钟后，python抛出了一个找不到文件的异常，我们看到这个模型文件是

```
http://omniverse-content-production.s3-us-west-  
2.amazonaws.com/Assets/Isaac/4.5/Isaac/IsaacLab/Robots/Unitree/G1/g1_minimal.usd
```

这正是我们要找的.usd文件！我们可以写一个脚本分析他的结构：

```
def explore_usd_file(path):  
    stage = Usd.Stage.Open(path)  
    if not stage:  
        print(f"Failed to open {path}")  
        return  
    print(f"\n--- Exploring: {path} ---")  
    # 遍历所有 Prim (物体节点)
```

```

for prim in stage.Traverse():
    print(f"Prim: {prim.GetPath()} Type: {prim.GetTypeName()}")
    # 示例: 如果是 Mesh, 打印点数
    if prim.GetTypeName() == "Mesh":
        mesh = UsdGeom.Mesh(prim)
        points_attr = mesh.GetPointsAttr()
        points = points_attr.Get()
        print(f"  Mesh with {len(points)} points")

```

然后可以得到

```

--- Exploring: /home/ecs-user/working-labs/H1transferG1/assets/g1_minimal.usd ---
Prim: /physicsScene Type: PhysicsScene
Prim: /g1 Type: Xform
Prim: /g1/pelvis Type: Xform
Prim: /g1/pelvis/visuals Type:
Prim: /g1/pelvis/left_hip_pitch_joint Type: PhysicsRevoluteJoint
Prim: /g1/pelvis/pelvis_contour_joint Type: PhysicsFixedJoint
Prim: /g1/pelvis/right_hip_pitch_joint Type: PhysicsRevoluteJoint
...

```

这就是详细的结构了，我们可以对照上面的列表分析关节，此外，我们将 `.usd` 文件下载到了本地，这在下面的工作中非常有用。

在得到关节命名后，我们可以建立H1到G1的关节映射表。因为G1的关节比H1更多，我们将冗余的自由度全部设为0。

i Note

回顾一下流程：

G1观测->转换为H1观测->输入H1策略->得到H1动作（关节动作表）->映射为G1动作

这说明我们只需要H1到G1的单向关节映射，而不需要G1到H1的

```

H1_TO_G1_JOINT_MAP = {
    # 下肢 (左腿)
    "left_hip_yaw": "left_hip_yaw_joint",
    "left_hip_roll": "left_hip_roll_joint",
    "left_hip_pitch": "left_hip_pitch_joint",
    "left_knee": "left_knee_joint",
    "left_ankle": "left_ankle_pitch_joint",
    # G1多出的踝关节roll
    "left_ankle_roll_joint": 0.0,
    # 下肢 (右腿)
    "right_hip_yaw": "right_hip_yaw_joint",
    "right_hip_roll": "right_hip_roll_joint",
    "right_hip_pitch": "right_hip_pitch_joint",
    "right_knee": "right_knee_joint",
    "right_ankle": "right_ankle_pitch_joint",
}

```

```

# G1多出的踝关节roll
"right_ankle_roll_joint": 0.0,
# 躯干
"torso": "torso_joint",
# 上肢 (左臂)
"left_shoulder_pitch": "left_shoulder_pitch_joint",
"left_shoulder_roll": "left_shoulder_roll_joint",
"left_shoulder_yaw": "left_shoulder_yaw_joint",
"left_elbow": "left_elbow_pitch_joint",
# G1多出的肘关节roll
"left_elbow_roll_joint": 0.0,
# 上肢 (右臂)
"right_shoulder_pitch": "right_shoulder_pitch_joint",
"right_shoulder_roll": "right_shoulder_roll_joint",
"right_shoulder_yaw": "right_shoulder_yaw_joint",
"right_elbow": "right_elbow_pitch_joint",
# G1多出的肘关节roll
"right_elbow_roll_joint": 0.0,
}

```

有了这张完整的关节映射表，我们可以写出H1到G1的动作映射函数。

Important

下面的动作映射函数正确前提是关节名列表有序，这需要和神经网络输出做一一对应。这也是为什么上文花了很多功夫确认关节列表顺序。

```

def map_h1_action_to_g1(h1_action, h1_joint_names=h1_joint_names,
g1_joint_names=g1_joint_names):
    # H1关节名到action值的映射
    h1_action_dict = dict(zip(h1_joint_names, h1_action))
    g1_action = []
    for g1_joint in g1_joint_names:
        # 反查映射表，找到H1关节名
        h1_key = None
        for k, v in H1_TO_G1_JOINT_MAP.items():
            if v == g1_joint:
                h1_key = k
                break
        if h1_key is not None and h1_key in h1_action_dict:
            g1_action.append(h1_action_dict[h1_key])
        elif g1_joint in H1_TO_G1_JOINT_MAP and H1_TO_G1_JOINT_MAP[g1_joint] == 0.0:
            g1_action.append(0.0)
        else:
            # 其它G1关节（如手指等），也补零
            g1_action.append(0.0)
    return g1_action

```

至此，我们得到了关键函数 `map_h1_action_to_g1`，这解决了迁移流程中重要的部分：将H1策略模型的输出应用在G1上。

使用代码探针分析观测数据构造，得到观测映射表

注意到

```
source/isaaclab_tasks/isaaclab_tasks/manager_based/locomotion/velocity/velocity_env_cfg.py
116行开始
```

```
@configclass
class ObservationsCfg:
    """Observation specifications for the MDP."""

    @configclass
    class PolicyCfg(ObsGroup):
        """Observations for policy group."""

        # observation terms (order preserved)
        base_lin_vel = ObsTerm(func=mdp.base_lin_vel, noise=Unoise(n_min=-0.1,
n_max=0.1))
        base_ang_vel = ObsTerm(func=mdp.base_ang_vel, noise=Unoise(n_min=-0.2,
n_max=0.2))
        projected_gravity = ObsTerm(
            func=mdp.projected_gravity,
            noise=Unoise(n_min=-0.05, n_max=0.05),
        )
        velocity_commands = ObsTerm(func=mdp.generated_commands, params={
            "command_name": "base_velocity"})
        joint_pos = ObsTerm(func=mdp.joint_pos_rel, noise=Unoise(n_min=-0.01,
n_max=0.01))
        joint_vel = ObsTerm(func=mdp.joint_vel_rel, noise=Unoise(n_min=-1.5,
n_max=1.5))
        actions = ObsTerm(func=mdp.last_action)
        height_scan = ObsTerm(
            func=mdp.height_scan,
            params={"sensor_cfg": SceneEntityCfg("height_scanner")},
            noise=Unoise(n_min=-0.1, n_max=0.1),
            clip=(-1.0, 1.0),
        )

    def __post_init__(self):
        self.enable_corruption = True
        self.concatenate_terms = True

    # observation groups
    policy: PolicyCfg = PolicyCfg()
```

这个配置文件定义了观测的一般形式：

观测项	说明
base_lin_vel, base_ang_vel	当前的运动状态
projected_gravity	姿态估计
velocity_commands	想要达到的速度（任务目标）
joint_pos, joint_vel	运动系统的状态
actions	历史动作（用于构建记忆）
height_scan	地形信息

然而，仅仅知道这些是不足以构建观测映射表的。我们仍需要在训练代码中加入探针，打印出来看一下

```
# 沿用上一节提到的--extrat参数
# ===== 新增：提取 observation 示例和 shape =====
obs = env.reset()
import numpy as np
print("\nExtracted observation structure:")
if isinstance(obs, (tuple, list)) and len(obs) > 0 and isinstance(obs[0], dict) and
'policy' in obs[0]:
    policy_obs = obs[0]['policy']
    print(f"policy: shape={policy_obs.shape} dtype={getattr(policy_obs, 'dtype',
type(policy_obs))}")
# 打印第一个环境的 observation 向量
arr = policy_obs[0].detach().cpu().numpy() if hasattr(policy_obs[0], 'detach')
else np.array(policy_obs[0])
print(f" First row (len={len(arr)}): {arr}")
print(f" First 20 values: {arr[:20]}")
if hasattr(policy_obs, 'detach'):
    policy_obs = policy_obs.detach().cpu().numpy()

print("Index\t[9]\t[10]\t[11]\t[12]")
for i in range(10):
    print(f"{i:>5}\t{policy_obs[i][9]:.6f}\t{policy_obs[i]
[10]:.6f}\t{policy_obs[i][11]:.6f}\t{policy_obs[i][12]:.6f}")
else:
    print("Observation type/structure not recognized. Raw output:", obs)
# ===== 新增结束 =====
```

现在我们打印一下H1的观测空间

```
Extracted observation structure:
policy: shape=torch.Size([4096, 69]) dtype=torch.float32
First row (len=69): [-5.6066342e-02 -8.7517925e-02  4.9868785e-02 -9.6398674e-02
```

```

-1.9630268e-01 -1.1851498e-01 2.4278294e-02 3.7880894e-02
-1.0474691e+00 3.5836050e-01 0.0000000e+00 -4.6731085e-01
8.4095336e-03 -8.9571215e-03 -4.2297798e-03 4.5316387e-03
-8.4742559e-03 3.4465501e-03 8.2933642e-03 3.7253592e-03
7.2544329e-03 -9.4385361e-03 4.5920312e-03 -6.5772794e-05
-2.6932871e-03 -5.2364785e-03 8.4291194e-03 9.6111558e-03
8.7782890e-03 7.7467002e-03 -6.2260972e-03 -1.2100315e+00
5.8564234e-01 -4.7681594e-01 -7.0546228e-01 1.3577251e+00
3.5666263e-01 2.7153444e-01 -1.3317013e-01 -3.0642664e-01
-1.2162994e+00 -3.2588577e-01 -4.4944549e-01 -6.0138190e-01
-5.8937967e-01 -2.3791862e-01 9.6854997e-01 1.4231379e+00
5.8026981e-01 7.4255371e-01 0.0000000e+00 0.0000000e+00
0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
0.0000000e+00]

First 20 values: [-0.05606634 -0.08751792 0.04986878 -0.09639867 -0.19630268
-0.11851498
0.02427829 0.03788089 -1.0474691 0.3583605 0. -0.46731085
0.00840953 -0.00895712 -0.00422978 0.00453164 -0.00847426 0.00344655
0.00829336 0.00372536]

Index [9] [10] [11] [12]
0 0.358360 0.000000 -0.467311 0.008410
1 0.298718 0.000000 -0.848940 -0.000898
2 0.329300 0.000000 -0.413140 -0.008372
3 0.396957 0.000000 0.005005 -0.004949
4 0.635064 0.000000 0.069569 -0.008066
5 0.414920 0.000000 0.000215 0.006416
6 0.253440 0.000000 -0.763752 -0.002671
7 0.801332 0.000000 -0.484505 0.003594
8 0.278513 0.000000 0.841860 0.001726
9 0.966565 0.000000 -0.863862 0.000334

```

`shape=torch.Size([4096, 69])` 代表了H1的观测空间是69维的。还记得上节中提到的Actor网络吗？H1的Actor网络是

```

Actor MLP: Sequential(
  (0): Linear(in_features=69, out_features=128, bias=True)
  (1): ELU(alpha=1.0)
  (2): Linear(in_features=128, out_features=128, bias=True)
  (3): ELU(alpha=1.0)
  (4): Linear(in_features=128, out_features=128, bias=True)
  (5): ELU(alpha=1.0)
  (6): Linear(in_features=128, out_features=19, bias=True)
)

```

太好了，这证明我们的结论没有问题。

我们需要具体的观测构造方法，但是代码高度抽象，打印出来也无济于事。我们可以总结一下上文，汇总一下已知信息

- H1有19个关节，观测空间69维
- G1有37个关节，观测空间123维
- 观测空间按照顺序，分别是base_lin_vel, base_ang_vel, projected_gravity, velocity_commands, joint_pos, joint_vel, actions

当关节数目已知时，joint_pos, joint_vel, actions的维度就已知了，加之常见的速度一般是2维或者3维，我们可以解方程+合理猜测

H1的观测空间

```
[0:3]    base_lin_vel      (3)
[3:6]    base_ang_vel      (3)
[6:9]    projected_gravity (3)
[9:12]   velocity_commands (3)
[12:31]  joint_pos        (19)
[31:50]  joint_vel        (19)
[50:69]  actions          (19)
```

G1的观测空间

```
[0:3]    base_lin_vel      (3)
[3:6]    base_ang_vel      (3)
[6:9]    projected_gravity (3)
[9:12]   velocity_commands (3)
[12:49]  joint_pos        (37)
[49:86]  joint_vel        (37)
[86:123] actions          (37)
```

这是一个非常合理的观测结构！现在我们可以开始写观测映射函数：

```
def map_g1_obs_to_h1_obs(policy_obs_row, g1_joint_names=g1_joint_names,
h1_joint_names=h1_joint_names):
    """
    将G1环境的policy observation向量，重排为H1策略可用的输入格式。
    """

    Args:
        policy_obs_row: G1环境输出的单个环境的policy observation
        g1_joint_names: G1机器人的关节名称列表
        h1_joint_names: H1机器人的关节名称列表

    Returns:
        h1_obs: 重排后符合H1策略输入格式的observation向量
    """
    # 创建输出向量，初始为空
    h1_obs = []
```

```

# 1. 前12维直接复制 [0:12] (base_lin_vel, base_ang_vel, projected_gravity,
velocity_commands)
h1_obs.extend(policy_obs_row[0:12])

# 2. 重排joint_pos部分 [12:49] -> [12:31]
g1_joint_pos = policy_obs_row[12:49] # G1的37个关节位置
g1_joint_pos_dict = dict(zip(g1_joint_names, g1_joint_pos)) # 构建G1关节名到位置的
映射

# 为H1创建关节位置部分
h1_joint_pos = []
for h1_joint in h1_joint_names:
    # 找到对应的G1关节名
    g1_joint = H1_TO_G1_JOINT_MAP.get(h1_joint)
    if g1_joint in g1_joint_pos_dict:
        h1_joint_pos.append(g1_joint_pos_dict[g1_joint])
    else:
        # 如果没找到对应的G1关节（理论上不应该发生），使用0作为默认值
        h1_joint_pos.append(0.0)

h1_obs.extend(h1_joint_pos)

# 3. 重排joint_vel部分 [49:86] -> [31:50]
g1_joint_vel = policy_obs_row[49:86] # G1的37个关节速度
g1_joint_vel_dict = dict(zip(g1_joint_names, g1_joint_vel)) # 构建G1关节名到速度的
映射

# 为H1创建关节速度部分
h1_joint_vel = []
for h1_joint in h1_joint_names:
    # 找到对应的G1关节名
    g1_joint = H1_TO_G1_JOINT_MAP.get(h1_joint)
    if g1_joint in g1_joint_vel_dict:
        h1_joint_vel.append(g1_joint_vel_dict[g1_joint])
    else:
        # 如果没找到对应的G1关节，使用0作为默认值
        h1_joint_vel.append(0.0)

h1_obs.extend(h1_joint_vel)

# 4. 重排actions部分 [86:123] -> [50:69]
g1_actions = policy_obs_row[86:123] # G1的37个关节动作
g1_actions_dict = dict(zip(g1_joint_names, g1_actions)) # 构建G1关节名到动作的映射

# 为H1创建动作部分
h1_actions = []
for h1_joint in h1_joint_names:
    # 找到对应的G1关节名

```

```

g1_joint = H1_TO_G1_JOINT_MAP.get(h1_joint)
if g1_joint in g1_actions_dict:
    h1_actions.append(g1_actions_dict[g1_joint])
else:
    # 如果没找到对应的G1关节，使用0作为默认值
    h1_actions.append(0.0)

h1_obs.extend(h1_actions)

# 确保输出向量维度为69
assert len(h1_obs) == 69, f"输出向量维度应为69, 当前为{len(h1_obs)}"

return h1_obs

```

至此，我们完成了G1到H1的观测映射！我们的流程：

G1观测->转换为H1观测->输入H1策略->得到H1动作（关节动作表）->映射为G1动作

已经构建完成！

修改杂项参数，对齐G1到H1

在我们跑通这个迁移之前，我们还应调整G1的初始站姿，这非常重要，因为H1的策略都是从固定的站姿开始的，如果模型收到了未知的初始值，很可能返回随机的结果，这会让机器人在一开始就无法站住

```

G1_CFG = ArticulationCfg(
...
# 修改初始状态以匹配H1的配置
init_state=ArticulationCfg.InitialStateCfg(
    pos=(0.0, 0.0, 0, 74), # 高度不用改
    joint_pos={
        # 下肢关节 - 映射自H1
        ".*_hip_yaw_joint": 0.0, # 对应H1的 *_hip_yaw
        ".*_hip_roll_joint": 0.0, # 对应H1的 *_hip_roll
        ".*_hip_pitch_joint": -0.28, # 对应H1的 *_hip_pitch (-16度)
        ".*_knee_joint": 0.79, # 对应H1的 *_knee (45度)
        ".*_ankle_pitch_joint": -0.52, # 对应H1的 *_ankle (-30度)
        ".*_ankle_roll_joint": 0.0, # G1特有, H1没有, 设为0

        # 躯干关节
        "torso_joint": 0.0, # 对应H1的 torso

        # 上肢关节 - 映射自H1
        ".*_shoulder_pitch_joint": 0.28, # 对应H1的 *_shoulder_pitch
        ".*_shoulder_roll_joint": 0.0, # 对应H1的 *_shoulder_roll
        ".*_shoulder_yaw_joint": 0.0, # 对应H1的 *_shoulder_yaw
        ".*_elbow_pitch_joint": 0.52, # 对应H1的 *_elbow
        ".*_elbow_roll_joint": 0.0, # G1特有, H1没有, 设为0
    }
)

```

```
# 手部关节 - G1特有, 保持原始值或设为中立位置
".*_five_joint": 0.0, # G1特有, 设为中立位置
".*_three_joint": 0.0, # G1特有, 设为中立位置
".*_six_joint": 0.0, # G1特有, 设为中立位置
".*_four_joint": 0.0, # G1特有, 设为中立位置
".*_zero_joint": 0.0, # G1特有, 设为中立位置
"left_one_joint": 1.0, # 保持原配置中的值
"right_one_joint": -1.0, # 保持原配置中的值
"left_two_joint": 0.52, # 保持原配置中的值
"right_two_joint": -0.52, # 保持原配置中的值
},
...
}
```

构建测试环境，跑通实验

在有了完整pipeline之后，我们可以在Task1实验的基础上搭建测试环境

1. 初始化环境为平地
2. G1机器人观测
3. 调用函数，将G1观测转换为H1观测
4. 调用模型，将H1观测推理为H1动作
5. 调用函数，将H1动作转换为G1动作
6. 迭代仿真



实验的结果是，搭载H1策略的G1可以作出行走的动作：但他会在尝试走4-6步内因为身体前倾跌倒。

更多思考

搭载H1策略的G1机器人会在

- 开始时尝试走路
- 然后原地踏步
- 接着跌倒

说明策略在一开始还“有点用”，但接着无法适应新机器人的机械结构。

主要失败原因

I 关节补零的方式导致策略失效

我们把 H1 上不存在的关节在动作空间里置 0，这会让 G1 的这些关节“僵硬”或“悬空”，从而：

- 导致 G1 的质量分布和动态行为与 H1 差异较大

I 动力学差异

H1 和 G1 的质量分布、腿长、惯性张量、摩擦系数等都不同，即使观测空间和动作空间对齐，也不能保证策略能稳定迁移。

RL Fine-Tuning

- 保留原始策略网络结构和参数
- 把策略部署到 G1 的仿真环境中
- 使用与原来相同的强化学习算法继续训练策略，优化在 G1 上的表现
- 学习率要低一点，防止“忘记”原来的技能。
- 环境奖励函数要保持一致或稍作调整以适配 G1。
- 可以冻结部分网络层（如前几层），只训练输出层。