**ACR2025 Isaac Sim / Isaac Lab Tutorial**

|  |
| --- |
| [**视频教程**](#heading_0)  [**1. 第一部分：核心概念与机器人仿真生态**](#heading_1)  [**1.1 机器人仿真平台概览**](#heading_2)  [**1.2 Isaac 生态系统：Isaac Sim 与 Isaac Lab**](#heading_3)  [**1.3 Isaac Sim 核心架构：App， Sim， World， Stage， Scene**](#heading_4)  [**1.4 环境配置说明**](#heading_5)  [**在 GradMotion 云桌面配置环境：**](#heading_6)  [**2. 第二部分：Isaac Sim 场景创建**](#heading_7)  [**2.1 如何创建一个仿真？**](#heading_8)  [**2.2 三大核心工作流概览**](#heading_9)  [**3. 第三部分：使用 Isaac Lab 模板配置环境**](#heading_10)  [**3.1 演示1：使用配置类生成Prim**](#heading_11)  [**3.2 演示2：使用 Interactive Scene 创建多个环境**](#heading_12)  [**4. 第四部分：高级运动学：实时获取连杆位姿**](#heading_13)  [**4.1 访问机器人连杆**](#heading_14)  [**4.2 计算并打印相对变换**](#heading_15)  [**4.3 交互与观察**](#heading_16)  [**5. 作业（ddl: 10月31日 23:59）**](#heading_17)  [**截止时间**](#heading_18)  [**作业目标**](#heading_19)  [**一、准备资源与目录**](#heading_20)  [**二、安装依赖（用于键盘控制）**](#heading_21)  [**三、运行示例（手动控制 + 视角跟随）**](#heading_22)  [**五、作业要求（学生完成）**](#heading_23)  [**六、交付物（DDL：下周五晚 23:59）**](#heading_24)  [**七、常见问题**](#heading_25) |

|  |
| --- |
| **参考教程：**   * [Isaac Sim 官方文档](https://docs.isaacsim.omniverse.nvidia.com/5.0.0/index.html) * [Isaac Lab 官方文档](https://isaac-sim.github.io/IsaacLab/main/index.html) * [Lychee AI Isaac Lab Tutorial](https://www.youtube.com/watch?v=ksF8mDOqwy4) |

**视频教程**

**[1.机器人仿真和Isaac Sim Lab核心概念.mov]**

**[1.5 直观理解app,sim,world,stage,scene.mov]**

**[2.GradMotion云桌面环境配置.mov]**

**[3.Isaac Sim 操作方式--GUI.mov]**

**[4.Isaac Sim 操作方式--脚本编辑器.mov]**

**[5.使用Isaac Lab 模板创建Prim.mov]**

**[6.使用 Interactive Scene 创建多个环境.mov]**

**[7. Isaac Sim 实时获取连杆位姿并计算变换.mov]**

1. **第一部分：核心概念与机器人仿真生态**

本教程旨在提供一份精炼的 NVIDIA Isaac 生态系统指南。内容始于机器人仿真的宏观视角，随后聚焦于 Isaac Sim 的基础操作与场景构建，并最终深入探讨专为机器人学习设计的框架 Isaac Lab。

1.1 **机器人仿真平台概览**



在现代机器人学中，物理仿真平台是开发、测试和验证算法的关键工具。它们提供了一个可控的虚拟环境，能够在部署到物理硬件前，安全、高效地迭代机器人设计与控制策略。

目前主流的仿真平台各具特色。例如，**Gazebo** 因其与机器人操作系统（ROS）的深度集成而广受欢迎。**MuJoCo**（多关节动力学与接触）则以其在接触动力学方面的高精度和高效率著称，成为强化学习研究领域的常用工具。围绕 MuJoCo 也衍生出了多个学习框架，如专为机器人操作任务设计的 **robosuite** 和旨在简化 sim-to-real 流程的 **MuJoCo Playground**。

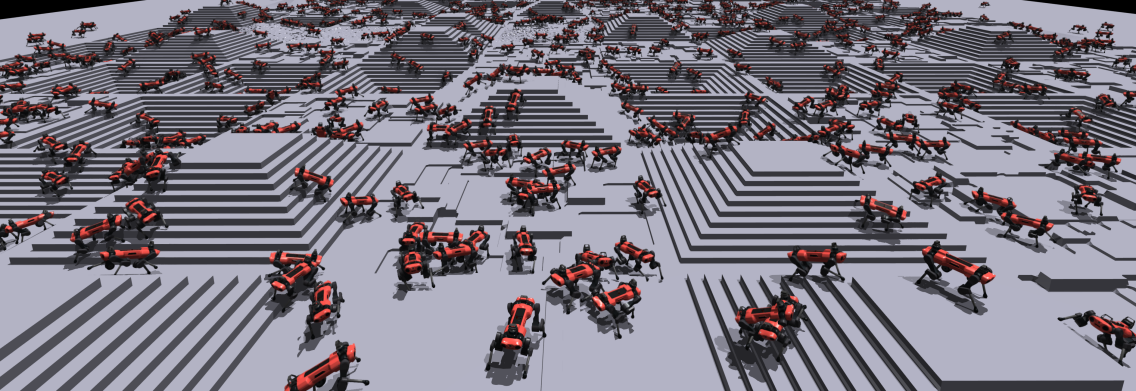
过去的普遍认识是，**MuJoCo** 因其在接触动力学方面的精确性而拥有更高精度的物理仿真，但它主要依赖 CPU 进行多线程并行化，这对于需要大规模并行训练的强化学习任务来说效率不高。因此，一个常见的工作流程是在 **Isaac Gym** 或 **Isaac Lab** 这类支持 GPU 大规模并行加速的平台上进行训练，然后将在 MuJoCo 中进行 sim-to-sim 实验以验证其物理真实性。

然而，机器人仿真领域近期发展迅速。出现了基于 Isaac Lab API 和 MuJoCo 仿真的 **MujocoLab**，它能够利用 GPU 加速在 MuJoCo 中的训练过程。同时，Isaac Sim 也推出了名为 **Newton** 的可微分仿真引擎。尽管技术在不断演进，目前我们仍先学习主流的工具——Isaac Sim 和 Isaac Lab。

1.2 **Isaac 生态系统：Isaac Sim 与 Isaac Lab**

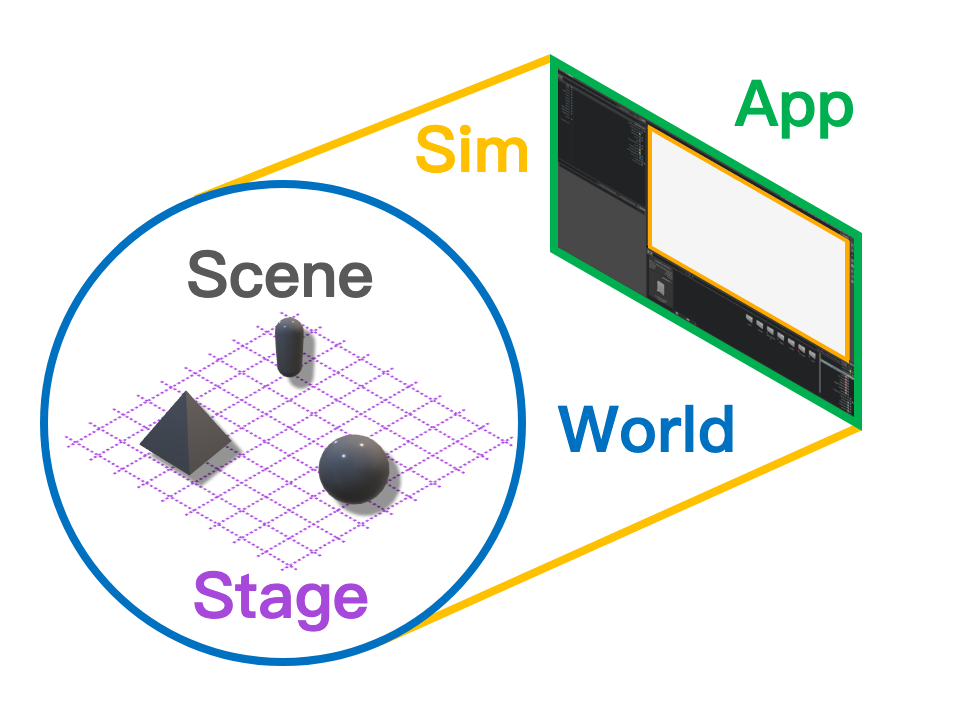
NVIDIA的机器人平台主要由两大核心组件构成，它们之间是层级关系：基础仿真平台**Isaac Sim**，以及构建于其上的机器人学习应用框架**Isaac Lab**。

* **Isaac Sim 是什么？** 它是一个通用的机器人模拟器，提供了高保真的物理引擎（PhysX）和照片级的渲染技术（RTX）。其核心任务是**构建和模拟**一个精确、逼真的虚拟环境，包括机器人模型、传感器数据和物理交互。
* **Isaac Lab 是什么？** 它是一个专为机器人学习（特别是强化学习）设计的开源框架。它本身不是一个模拟器，而是**利用**Isaac Sim提供的环境来进行大规模的AI模型训练。
* **它们的关系？**Isaac Lab完全依赖于Isaac Sim 。可以遵循一个简单的原则：“**在Isaac Sim中构建，在Isaac Lab中训练**”。Isaac Lab继承了其前身Isaac Gym在GPU并行计算方面的优势，并将其与Isaac Sim强大的渲染能力相结合，实现了在数千个并行环境中高效训练机器人策略（即“向量化”）。

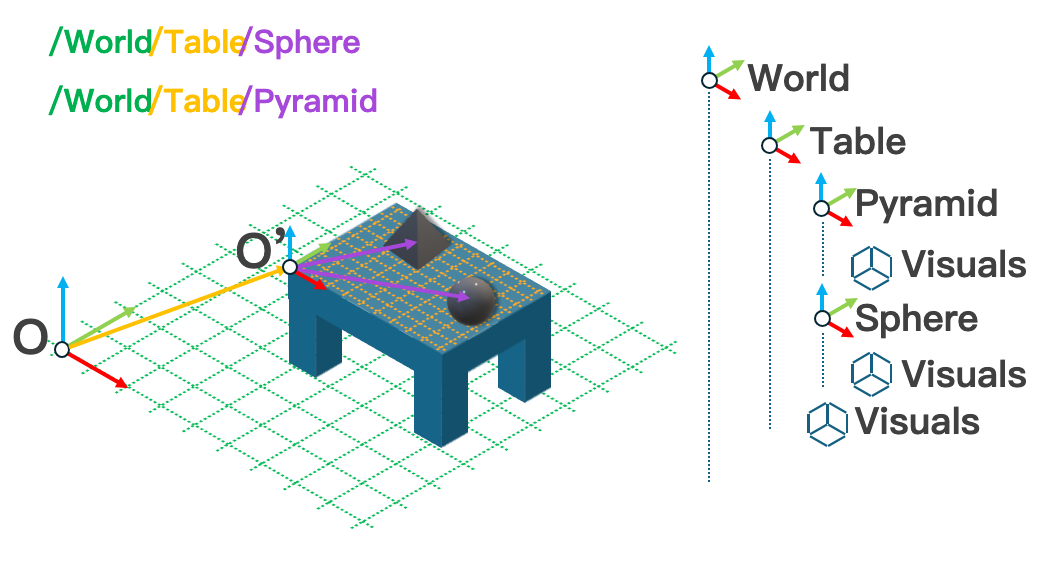


1.3 **Isaac Sim 核心架构：App， Sim， World， Stage， Scene**

要精通 Isaac Sim，必须理解其分层架构中的五个核心概念。



1. **Application （App）**：这是最高层级的管理者，负责所有资源的生命周期，包括启动和销毁仿真进程。即使用户在无头模式（headless）下运行，App 依然是整个程序的总控制器。
2. **Simulation （Sim）**：Sim 负责定义虚拟世界的“规则”，例如物理定律（如重力方向）、时间步长（dt）以及渲染频率。它将时间的每一步划分为不同的子步骤（如physics\_step和render\_step），并掌管着 World 对象。
3. **World**：World 为仿真提供了空间背景，定义了笛卡尔坐标系的原点和单位。所有关于尺寸和距离的问题都在 World 的参考系内得以解答。
4. **Stage**：Stage 是世界的“组成结构”。它以**通用场景描述（Universal Scene Description， USD）为基础，将仿真中的所有元素（如机器人、灯光、摄像机）表示为一个层级化的树状结构。这个结构中的每一个节点都是一个图元（Prim）**。



* **USD 图元（Prim）**：Prim 是 USD 场景的基本构建块，可以理解为一个容器。每个 Prim 都有一个唯一的路径（例如/World/MyRobot/Gripper），并包含定义其特性的**属性（Attributes）**（如颜色、大小）和与其他 Prim 的**关系（Relationships）**（如材质指定）。例如，一个“树”的 Prim 可以有“高度”和“颜色”等属性，同时与一个“地面”Prim 建立关系，以表明其种植位置。父级 Prim 的属性可以被其子级继承，从而实现复杂的场景组合。

1. **Scene**：Scene 是 Isaac Lab 中一个至关重要的概念，它管理着 Stage 上所有与**向量化**（vectorization）相关的图元。这些被管理的图元（如机器人、桌子、待抓取物体）被称为**仿真实体（simulation entities）**。当用户指定需要创建多个环境副本时，Scene 会自动在 Stage 上克隆这些实体，并将它们放置在不同的坐标位置，从而实现在单一世界和舞台上进行大规模并行训练。

总而言之，**App** 管理整个仿真进程，**Sim** 定义物理规则，**World** 提供空间参考，**Stage** 通过 USD 图元组织场景结构，而 **Scene** 则负责高效地复制这些结构以加速学习。

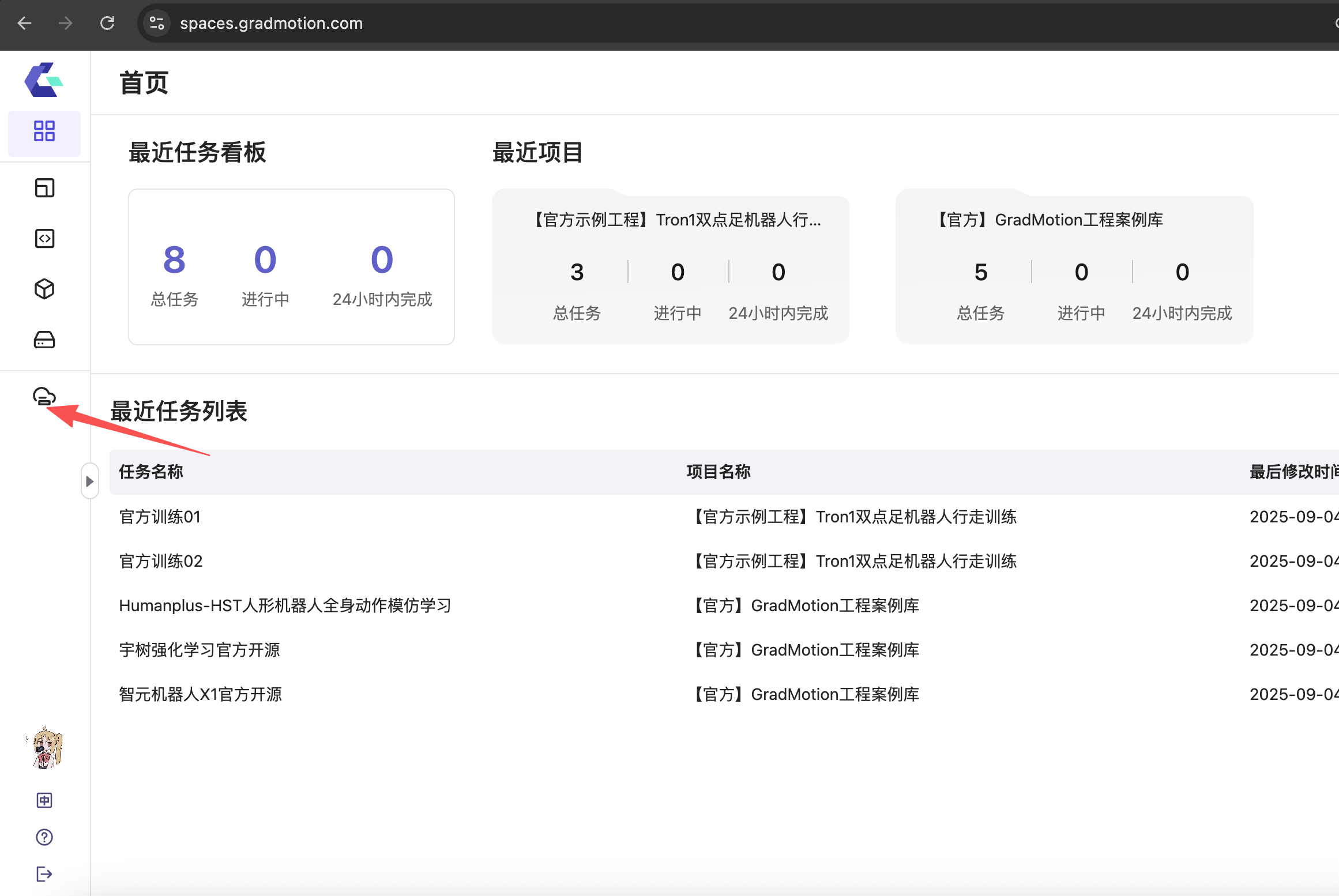
1.4 **环境配置说明**

由于本课程中学生使用构建好 Ubuntu22.04 + Isaac Sim + Isaac Lab 镜像的 GradMotion 云桌面，此部分仅作了解，希望在自己电脑上本地安装的同学可以参考该文档。

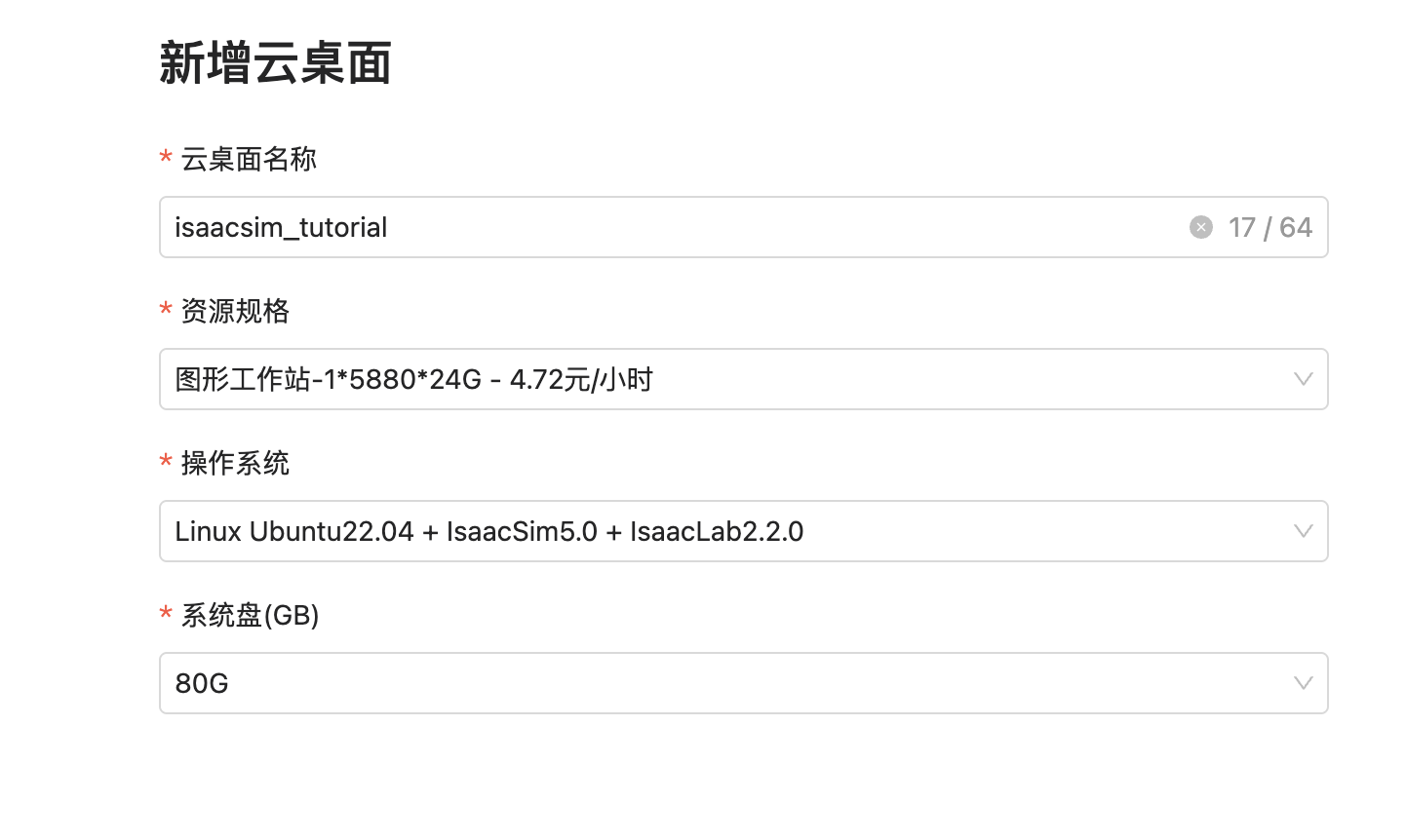
https://isaac-sim.github.io/IsaacLab/main/source/setup/installation/index.html

**在 GradMotion 云桌面配置环境：**

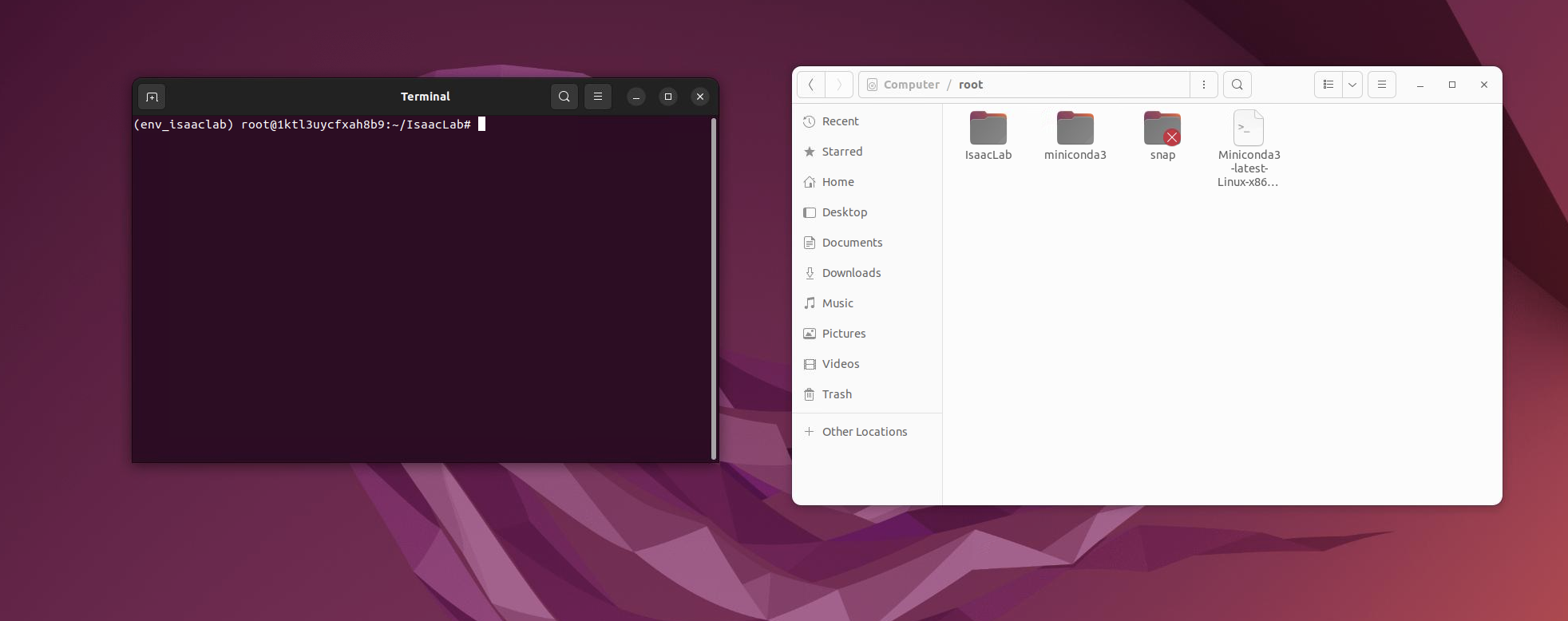
1. 首先打开 GradMotion 平台并登录：https://gradmotion.com/
2. 进入后选择左侧“云桌面”。



1. 创建一个新的云桌面并开机，配置如下：



1. 完成后连接进入云桌面，鼠标右键--在终端中打开，即可看到目前已经完成 isaac sim 5.0 以及 isaac lab 2.2.0 的安装，此时终端处于/root/IsaacLab下，并且已经打开conda环境env\_isaaclab。



1. 在终端直接输入isaacsim，即可打开仿真。

2. **第二部分：Isaac Sim 场景创建**

掌握在 Isaac Sim 中构建、操纵和模拟虚拟场景是核心技能。本节将简要介绍三种核心工作流程，并重点解析其中最强大和最适用于自动化任务的独立 Python 脚本方法。此处详细的入门操作步骤，建议学生直接跟随官方的快速入门教程进行实践：https://docs.isaacsim.omniverse.nvidia.com/5.0.0/introduction/quickstart\_isaacsim.html

2.1 **如何创建一个仿真？**

回顾[1.3节](https://iyrna6v2lz.feishu.cn/wiki/Zjhbw2ieAibcLMkZkaBcN5FxnaH#share-P2XldgOTxojofvxktxyc2oWBnUC)中中讲述的内容，创建一个仿真，本质上是定义一个**层级结构**。我们可以沿用第一部分的概念来理解需要指定的元素：

|  |  |
| --- | --- |
|  |  |

1. **App (应用程序)**：首先，你需要一个SimulationApp实例。这是整个仿真的入口和最高管理者，负责启动和关闭所有底层服务。
2. **Sim (仿真上下文)**：其次，你需要一个SimulationContext。它定义了仿真的“物理规则”，比如时间步长dt和渲染频率。
3. **World & Stage (世界与舞台)**：SimulationContext会自动创建一个World和Stage。你的任务是填充这个Stage，即向其中添加**图元 (Prims)**。
4. **Prims (图元)**：这些是构成场景的基本元素，例如：

* **环境元素**：如地面（GroundPlane）、灯光（DistantLight）。
* **物体**：如立方体（DynamicCuboid）、机器人（Articulation）。每个图元都需要在Stage上有一个唯一的路径（prim\_path），并可以设置其位置、颜色等属性。

2.2 **三大核心工作流概览**

按照官方文档的步骤进行练习：

https://docs.isaacsim.omniverse.nvidia.com/5.0.0/introduction/quickstart\_isaacsim.html

Isaac Sim 提供了三种主要的工作方式，以适应从快速原型设计到大规模自动化训练的不同需求 ：

1. **图形用户界面 （GUI）**：这是最直观的入门方式，允许用户通过点击和拖拽来构建场景、添加物体并赋予物理属性。它非常适合视觉探索和手动设计复杂的静态环境 。
2. **交互式 Python 脚本 （脚本编辑器）**：Isaac Sim 内置了一个脚本编辑器，允许用户在 GUI 运行时执行 Python 代码片段，实时操纵场景中的对象。这是学习和测试 API 命令的绝佳工具 。
3. **独立 Python 应用程序 （Standalone Python）**：这是最强大和可扩展的工作方式。用户可以编写一个完整的 Python 脚本，从命令行启动、控制和关闭整个仿真过程，无需 GUI 界面（即无头模式）。这是所有自动化任务，特别是强化学习训练的基石 。

3. **第三部分：使用 Isaac Lab 模板配置环境**

现在我们进入Isaac Lab的世界，首先学习如何使用其配置驱动的API来高效构建仿真场景，然后将这些知识应用到构建一个完整的大规模并行强化学习环境中。

3.1 **演示1：使用配置类生成Prim**

虽然直接使用USD API（如DynamicCuboid）可以创建场景，但Isaac Lab提供了一个更简洁、配置驱动的接口，即sim.spawners模块。这种方法将物体的属性定义（配置）与其实例化（生成）分离开来，使代码更具可读性和模块性。

下面的脚本（spawn\_prims.py）就是一个很好的例子，它展示了如何使用不同的配置类（如GroundPlaneCfg、DistantLightCfg、ConeCfg等）来构建一个包含多种物体的复杂场景。



|  |
| --- |
| Python # Copyright (c) 2022-2025, The Isaac Lab Project Developers (https://github.com/isaac-sim/IsaacLab/blob/main/CONTRIBUTORS.md). # All rights reserved. # # SPDX-License-Identifier: BSD-3-Clause  """This script demonstrates how to spawn prims into the scene.  .. code-block:: bash   # Usage  ./isaaclab.sh -p scripts/tutorials/00\_sim/spawn\_prims.py  """  """Launch Isaac Sim Simulator first."""   import argparse  from isaaclab.app import AppLauncher  # 创建参数解析器 parser = argparse.ArgumentParser(description="Tutorial on spawning prims into the scene.") # 添加AppLauncher的命令行参数 AppLauncher.add\_app\_launcher\_args(parser) # 解析参数 args\_cli = parser.parse\_args() # 启动Omniverse应用 app\_launcher = AppLauncher(args\_cli) simulation\_app = app\_launcher.app  """Rest everything follows."""  import isaacsim.core.utils.prims as prim\_utils  import isaaclab.sim as sim\_utils from isaaclab.utils.assets import ISAAC\_NUCLEUS\_DIR   def design\_scene():  """通过生成地面、灯光、物体和从USD文件加载的网格来设计场景。"""  # 地面  cfg\_ground = sim\_utils.GroundPlaneCfg()  cfg\_ground.func("/World/defaultGroundPlane", cfg\_ground)   # 生成平行光  cfg\_light\_distant = sim\_utils.DistantLightCfg(  intensity=3000.0,  color=(0.75, 0.75, 0.75),  )  cfg\_light\_distant.func("/World/lightDistant", cfg\_light\_distant, translation=(1, 0, 10))   # 为所有要生成的对象创建一个新的xform图元作为父级  prim\_utils.create\_prim("/World/Objects", "Xform")  # 生成一个红色的圆锥体（仅视觉，无物理属性）  cfg\_cone = sim\_utils.ConeCfg(  radius=0.15,  height=0.5,  visual\_material=sim\_utils.PreviewSurfaceCfg(diffuse\_color=(1.0, 0.0, 0.0)),  )  cfg\_cone.func("/World/Objects/Cone1", cfg\_cone, translation=(-1.0, 1.0, 1.0))  cfg\_cone.func("/World/Objects/Cone2", cfg\_cone, translation=(-1.0, -1.0, 1.0))   # 生成一个带碰撞体和刚体属性的绿色圆锥体  cfg\_cone\_rigid = sim\_utils.ConeCfg(  radius=0.15,  height=0.5,  rigid\_props=sim\_utils.RigidBodyPropertiesCfg(),  mass\_props=sim\_utils.MassPropertiesCfg(mass=1.0),  collision\_props=sim\_utils.CollisionPropertiesCfg(),  visual\_material=sim\_utils.PreviewSurfaceCfg(diffuse\_color=(0.0, 1.0, 0.0)),  )  cfg\_cone\_rigid.func(  "/World/Objects/ConeRigid", cfg\_cone\_rigid, translation=(-0.2, 0.0, 2.0), orientation=(0.5, 0.0, 0.5, 0.0)  )   # 生成一个带可变形体属性的蓝色长方体  cfg\_cuboid\_deformable = sim\_utils.MeshCuboidCfg(  size=(0.2, 0.5, 0.2),  deformable\_props=sim\_utils.DeformableBodyPropertiesCfg(),  visual\_material=sim\_utils.PreviewSurfaceCfg(diffuse\_color=(0.0, 0.0, 1.0)),  physics\_material=sim\_utils.DeformableBodyMaterialCfg(),  )  cfg\_cuboid\_deformable.func("/World/Objects/CuboidDeformable", cfg\_cuboid\_deformable, translation=(0.15, 0.0, 2.0))   # 从USD文件向场景中生成一个桌子  cfg = sim\_utils.UsdFileCfg(usd\_path=f"{ISAAC\_NUCLEUS\_DIR}/Props/Mounts/SeattleLabTable/table\_instanceable.usd")  cfg.func("/World/Objects/Table", cfg, translation=(0.0, 0.0, 1.05))   def main():  """主函数"""   # 初始化仿真上下文  sim\_cfg = sim\_utils.SimulationCfg(dt=0.01, device=args\_cli.device)  sim = sim\_utils.SimulationContext(sim\_cfg)  # 设置主相机视角  sim.set\_camera\_view([2.0, 0.0, 2.5], [-0.5, 0.0, 0.5])  sim.reset()  # 设计场景  design\_scene()  # 播放模拟器  sim.reset()  # 准备就绪！  print("[INFO]: Setup complete...")   # 模拟物理过程  while simulation\_app.is\_running():  # 执行一步  sim.step()   if \_\_name\_\_ == "\_\_main\_\_":  # 运行主函数  main()  # 关闭仿真应用  simulation\_app.close() |

* **启动与仿真上下文**：脚本的开头部分与之前的独立应用类似，使用AppLauncher启动仿真，并创建SimulationContext来管理仿真循环。
* **design\_scene()函数**：这是场景构建的核心。所有对象都在这个函数中定义和生成，并且这个函数在主仿真循环开始前被调用。
* **配置驱动的生成**：
* **无物理的视觉物体**：红色的圆锥体（Cone1, Cone2）是通过ConeCfg定义的。我们只指定了它们的几何形状（半径、高度）和视觉材质（颜色），因此它们在仿真中没有物理行为。
* **带物理的刚体**：绿色的圆锥体（ConeRigid）在配置中额外添加了rigid\_props（刚体属性）、mass\_props（质量属性）和collision\_props（碰撞属性）。这使得它成为一个可以与物理世界（如下落、碰撞）交互的动态物体。
* **其他类型**：脚本还展示了如何生成可变形体（MeshCuboidCfg）和从现有的USD文件加载模型（UsdFileCfg），展示了这套接口的灵活性。
* **cfg.func(...)**：每个配置类实例都有一个.func方法，它就是实际的生成函数。第一个参数是图元在舞台上的路径，第二个是配置对象本身，后面可以跟上位置和姿态等变换信息。

3.2 **演示2：使用 Interactive Scene 创建多个环境**

在掌握了如何使用配置类生成单个物体后，我们可以将这一思想扩展到构建一个完整的、用于强化学习的并行环境。我们将再次以倒立摆为例，但这次的重点是展示使用Interactive Scene 创建多个环境。

在本教程中，我们采用来自 “Interacting with an articulation” 教程的 cartpole 示例，并将 design\_scene 函数替换为一个 **scene.InteractiveScene** 对象。虽然在这个简单示例中使用 interactive scene 看起来有些大材小用，但随着场景中加入更多资产和传感器，它将变得更加有用。

|  |
| --- |
| Python # Copyright (c) 2022-2025, The Isaac Lab Project Developers (https://github.com/isaac-sim/IsaacLab/blob/main/CONTRIBUTORS.md). # All rights reserved. # # SPDX-License-Identifier: BSD-3-Clause  """ 本脚本演示了如何使用 InteractiveScene (交互式场景) 接口来搭建一个包含多个物体的场景。  .. code-block:: bash   # 使用方法示例，--num\_envs 用于指定并行环境的数量  ./isaaclab.sh -p scripts/tutorials/02\_scene/create\_scene.py --num\_envs 32  """  """首先，启动 Isaac Sim 仿真器。"""   import argparse  from isaaclab.app import AppLauncher  # -- 标准的 AppLauncher 启动流程 -- # 添加命令行参数解析器 parser = argparse.ArgumentParser(description="关于使用交互式场景接口的教程。") # 添加一个自定义参数 --num\_envs，用于从命令行控制环境数量 parser.add\_argument("--num\_envs", type=int, default=2, help="要生成的环境数量。") # 将 AppLauncher 的标准命令行参数 (如 --headless) 添加进来 AppLauncher.add\_app\_launcher\_args(parser) # 解析所有命令行参数 args\_cli = parser.parse\_args()  # 启动 Omniverse 应用程序 app\_launcher = AppLauncher(args\_cli) simulation\_app = app\_launcher.app  """接下来的代码是场景定义和仿真循环的核心部分。"""  import torch  import isaaclab.sim as sim\_utils from isaaclab.assets import ArticulationCfg, AssetBaseCfg from isaaclab.scene import InteractiveScene, InteractiveSceneCfg from isaaclab.sim import SimulationContext from isaaclab.utils import configclass  ## # 导入预定义的资产配置 ## from isaaclab\_assets import CARTPOLE\_CFG # isort:skip   @configclass class CartpoleSceneCfg(InteractiveSceneCfg):  """一个倒立摆场景的配置类。    InteractiveSceneCfg 允许我们以一种声明式的方式定义场景中包含的所有资产。  """   # 定义地面资产  # AssetBaseCfg 是一个通用的资产配置包装器  ground = AssetBaseCfg(  prim\_path="/World/defaultGroundPlane", # 指定该资产在 USD Stage 中的路径  spawn=sim\_utils.GroundPlaneCfg(), # 指定用于生成该资产的 Spawner 配置  )   # 定义灯光资产  dome\_light = AssetBaseCfg(  prim\_path="/World/Light", spawn=sim\_utils.DomeLightCfg(intensity=3000.0, color=(0.75, 0.75, 0.75))  )   # 定义关节机器人 (Articulation) 资产  # ArticulationCfg 是专门用于关节机器人的配置类  # {ENV\_REGEX\_NS} 是一个特殊的占位符，在场景创建时，它会被自动替换为每个环境的命名空间  # 例如：/World/envs/env\_0, /World/envs/env\_1, 等等。这实现了资产的自动克隆。  cartpole: ArticulationCfg = CARTPOLE\_CFG.replace(prim\_path="{ENV\_REGEX\_NS}/Robot")   def run\_simulator(sim: sim\_utils.SimulationContext, scene: InteractiveScene):  """运行仿真主循环。"""  # 从场景中提取我们需要的实体 (asset)。  # InteractiveScene 支持像字典一样的访问方式，键名对应于配置类中的变量名。  robot = scene["cartpole"]    # 获取物理仿真的时间步长  sim\_dt = sim.get\_physics\_dt()  count = 0  # 仿真主循环  while simulation\_app.is\_running():  # -- 重置逻辑：每 500 步重置一次环境 --  if count % 500 == 0:  # 重置计数器  count = 0  # -- 重置场景中所有实体的状态 --  # 1. 重置根物体 (base) 的状态  # 获取机器人默认的根状态 (位置、姿态、线速度、角速度)  root\_state = robot.data.default\_root\_state.clone()  # 关键步骤：将根状态的位置加上每个环境的偏移原点。  # 因为 default\_root\_state 是相对于环境自身原点(0,0,0)的，  # 而 write\_root\_pose\_to\_sim 需要的是世界坐标系中的绝对位置。  # scene.env\_origins 存储了每个克隆环境在世界坐标系中的位置。  root\_state[:, :3] += scene.env\_origins  # 将计算好的根物体位姿 (位置+姿态) 写回到仿真器中  robot.write\_root\_pose\_to\_sim(root\_state[:, :7])  # 将根物体的速度写回到仿真器中  robot.write\_root\_velocity\_to\_sim(root\_state[:, 7:])    # 2. 重置关节的状态  # 获取默认的关节位置和速度  joint\_pos, joint\_vel = robot.data.default\_joint\_pos.clone(), robot.data.default\_joint\_vel.clone()  # 为关节位置添加一些小的随机噪声，以增加训练的多样性  joint\_pos += torch.rand\_like(joint\_pos) \* 0.1  # 将计算好的关节状态写回到仿真器中  robot.write\_joint\_state\_to\_sim(joint\_pos, joint\_vel)    # 3. 重置场景管理器内部的缓冲区  scene.reset()  print("[INFO]: 正在重置机器人状态...")    # -- 应用随机动作 --  # 生成与关节数量匹配的随机力矩/力  efforts = torch.randn\_like(robot.data.joint\_pos) \* 5.0  # 设置关节力矩目标。注意：这一步只是更新了内存中的数据缓冲区，并未立即应用到仿真中。  robot.set\_joint\_effort\_target(efforts)  # 关键步骤：将所有在内存中缓冲的指令 (如此处的关节力矩) 一次性写入到仿真器中。  scene.write\_data\_to\_sim()    # -- 执行仿真 --  # 让物理引擎向前推进一步  sim.step()    # 增加计数器  count += 1    # -- 更新缓冲区 --  # 关键步骤：在物理步进之后，从仿真器中读取最新的状态数据 (如新的关节位置、速度)  # 并更新到 `robot.data` 等内存缓冲区中，以供下一个循环周期使用。  scene.update(sim\_dt)   def main():  """主函数。"""  # 创建仿真配置实例  sim\_cfg = sim\_utils.SimulationCfg(device=args\_cli.device)  # 创建仿真上下文实例  sim = SimulationContext(sim\_cfg)  # 设置一个合适的相机视角  sim.set\_camera\_view([2.5, 0.0, 4.0], [0.0, 0.0, 2.0])  sim.reset()  # -- 设计场景 --  # 创建场景配置实例，并从命令行参数传入并行环境的数量  scene\_cfg = CartpoleSceneCfg(num\_envs=args\_cli.num\_envs, env\_spacing=2.0)  # 关键步骤：实例化 InteractiveScene。  # 在这一步，它会解析 scene\_cfg，并自动完成所有资产的加载、克隆和放置。  scene = InteractiveScene(scene\_cfg)    # 第一次重置，完成物理场景的初始化  sim.reset()  # 现在，所有准备工作都已完成！  print("[INFO]: 场景搭建完成...")  # 调用并运行仿真主循环  run\_simulator(sim, scene)   if \_\_name\_\_ == "\_\_main\_\_":  # 运行主函数  main()  # 在结束后，安全地关闭仿真应用  simulation\_app.close() |

4. **第四部分：高级运动学：实时获取连杆位姿**

虽然Isaac Lab为强化学习提供了高级抽象，但有时我们需要直接与仿真中的机器人进行交互，以进行调试或运动学分析。本节将展示一个最直接的方法，用于加载一个Franka机械臂，并实时打印其末端执行器相对于基座的变换矩阵。

4.1 **访问机器人连杆**

要分析机器人的运动学，首先需要获取其在仿真中特定物理连杆的句柄。在Isaac Sim中，关节连接体（Articulation）的每个连杆本身都可以被视为一个**刚体图元（RigidPrim）**。通过获取这些图元的句柄，我们就可以查询它们在世界坐标系下的位姿。

4.2 **计算并打印相对变换**

以下是一个简单、独立的Python脚本，它加载一个Franka机械臂，并在仿真循环中持续计算并打印其末端执行器相对于基座的4x4齐次变换矩阵。这个脚本不依赖于复杂的EnvCfg，让您可以直接观察和交互。

|  |
| --- |
| Python """ 实时打印Franka机械臂基座到末端执行器的齐次变换矩阵 使用方法: ./isaaclab.sh -p /path/to/this/script.py """  import argparse import numpy as np from isaaclab.app import AppLauncher  parser = argparse.ArgumentParser(description="实时打印Franka末端执行器变换矩阵") AppLauncher.add\_app\_launcher\_args(parser) args\_cli = parser.parse\_args()  app\_launcher = AppLauncher(args\_cli) simulation\_app = app\_launcher.app  import omni.usd import isaaclab.sim as sim\_utils from pxr import UsdGeom  def create\_homogeneous\_matrix(translation, rotation\_matrix):  """构建4x4齐次变换矩阵"""  transform = np.eye(4)  transform[:3, :3] = rotation\_matrix *# 旋转部分*  transform[:3, 3] = translation *# 平移部分*  return transform  def get\_world\_transform(stage, prim\_path):  """获取USD prim的世界坐标系变换"""  prim = stage.GetPrimAtPath(prim\_path)  if not prim.IsValid():  return None, None    xform = UsdGeom.Xformable(prim)  world\_transform = xform.ComputeLocalToWorldTransform(0.0)    *# 提取位置*  translation = world\_transform.ExtractTranslation()  translation = np.array([translation[0], translation[1], translation[2]])    *# 提取旋转矩阵*  rotation = world\_transform.ExtractRotationMatrix()  rotation\_matrix = np.array([  [rotation[0][0], rotation[0][1], rotation[0][2]],  [rotation[1][0], rotation[1][1], rotation[1][2]],  [rotation[2][0], rotation[2][1], rotation[2][2]]  ])    return translation, rotation\_matrix  def main():  try:  *# 获取USD Stage*  context = omni.usd.get\_context()  stage = context.get\_stage()    *# 创建地面*  ground\_cfg = sim\_utils.GroundPlaneCfg()  ground\_cfg.func("/World/Ground", ground\_cfg)    *# 创建灯光*  light\_cfg = sim\_utils.DomeLightCfg(intensity=3000.0, color=(0.75, 0.75, 0.75))  light\_cfg.func("/World/Light", light\_cfg)    *# 导入Franka机器人*  robot\_cfg = sim\_utils.UsdFileCfg(  usd\_path="http://omniverse-content-production.s3-us-west-2.amazonaws.com/Assets/Isaac/4.0/Isaac/Robots/Franka/franka\_alt\_fingers.usd"  )  robot\_prim\_path = "/World/Franka"  robot\_cfg.func(robot\_prim\_path, robot\_cfg)    *# 定义关键路径*  base\_link\_path = f"{robot\_prim\_path}/panda\_link0" *# 基座链接*  ee\_link\_path = f"{robot\_prim\_path}/panda\_hand" *# 末端执行器*    count = 0  print\_interval = 30 *# 每30帧打印一次*    *# 主循环*  while simulation\_app.is\_running():  simulation\_app.update()  count += 1    *# 定期打印变换矩阵*  if count % print\_interval == 0:  *# 获取基座和末端的世界坐标系变换*  base\_pos, base\_rot = get\_world\_transform(stage, base\_link\_path)  ee\_pos, ee\_rot = get\_world\_transform(stage, ee\_link\_path)    if base\_pos is not None and ee\_pos is not None:  *# 构建世界坐标系变换矩阵*  T\_world\_base = create\_homogeneous\_matrix(base\_pos, base\_rot)  T\_world\_ee = create\_homogeneous\_matrix(ee\_pos, ee\_rot)    *# 计算相对变换: T\_base\_ee = inv(T\_world\_base) \* T\_world\_ee*  T\_base\_ee = np.linalg.inv(T\_world\_base) @ T\_world\_ee    *# 打印结果*  print("\n" + "-"\*80)  print(f"帧数: {count}")  print("-"\*80)  print("基座到末端执行器的齐次变换矩阵 (Base -> End-Effector):\n")  print("T\_base\_to\_ee = ")  for i in range(4):  row = " [" + ", ".join([f"{T\_base\_ee[i, j]:9.6f}" for j in range(4)]) + "]"  print(row)    *# 提取并打印位置*  position = T\_base\_ee[:3, 3]  print(f"\n位置 (x, y, z): [{position[0]:.6f}, {position[1]:.6f}, {position[2]:.6f}] m")  print("-"\*80)    except KeyboardInterrupt:  print("\n程序已停止")  except Exception as e:  print(f"\nERROR: {e}")  import traceback  traceback.print\_exc()  finally:  simulation\_app.close()  if \_\_name\_\_ == "\_\_main\_\_":  main() |

4.3 **交互与观察**

运行此脚本后，一个包含Franka机械臂的Isaac Sim窗口将会打开。在终端中，您将看到实时打印出的4x4变换矩阵。

最重要的是，您现在可以在GUI中与机器人进行交互：

1. 在视口中选择机械臂的任何一个连杆。
2. 使用移动（W）或旋转（E）工具来拖动关节。
3. 观察终端中打印的矩阵，它会随着您的操作实时更新。

这种直接的反馈对于理解机器人的运动学、调试控制算法以及验证坐标系都非常有价值。

5. **作业（ddl: 10月31日 23:59）**

使用已训练四足机器人策略进行手动控制与视角跟随，并在自定义环境中成功运行

**截止时间**

* 截止（DDL）：**10月31日** 晚 **23:59** 前

**作业目标**

* 在 Isaac Lab 中加载已训练的 RL 策略进行推理与演示
* 通过键盘（WASD + QE）控制机器人
* 设置摄像机视角跟随机器人
* 在官方 Isaac Lab 中添加与注册自定义任务
* 修改环境配置，让机器人在不同场景稳定运动

**一、准备资源与目录**

将以下文件放入官方 Isaac Lab 仓库对应路径（/root/IsaacLab）：

**[play\_manual.py]**

* **放入**scripts/reinforcement\_learning/rsl\_rl/play\_manual.py

**[manual\_env\_cfg.py]**

* **放入**source/isaaclab\_tasks/isaaclab\_tasks/manager\_based/locomotion/velocity/config/anymal\_c/manual\_env\_cfg.py

**[\_\_init\_\_.py]**

* **覆盖文件**source/isaaclab\_tasks/isaaclab\_tasks/manager\_based/locomotion/velocity/config/anymal\_c/\_\_init\_\_.py

**[model\_14999.pt]**

* 将提供的模型（model\_14999.pt）放在仓库根目录或任意可访问路径。

**二、安装依赖（用于键盘控制）**

使用 Isaac Lab 自带环境安装 keyboard：

|  |
| --- |
| Bash  pip install keyboard |

强制程序使用系统的 libstdc++：

|  |
| --- |
| Bash echo 'export LD\_PRELOAD=/usr/lib/x86\_64-linux-gnu/libstdc++.so.6' >> ~/.bashrc source ~/.bashrc |

**三、运行示例（手动控制 + 视角跟随）**

|  |
| --- |
| Bash  ./isaaclab.sh -p scripts/reinforcement\_learning/rsl\_rl/play\_manual.py \  --task=Isaac-Velocity-Rough-Anymal-C-Manual-v0 \  --num\_envs 1 \  --checkpoint /绝对路径/到/你的/model\_14999.pt \  --real-time |

控制键位：

* W/S：前进/后退
* A/D：横移左/右
* Q/E：左转/右转
* ESC：退出

现在你可以用键盘控制Anymal-C机器狗在默认的环境中行走了！可以尝试让机器狗跨越障碍，爬升台阶等等

**[10月20日.mp4]**

**五、作业要求（学生完成）**

目标：注释掉默认地形生成器，完全由你“**自行搭建几何场景**”，让机器人稳定行走并跨越障碍（手动控制即可）。

1. 注释默认地形

* 打开文件：source/isaaclab\_tasks/isaaclab\_tasks/manager\_based/locomotion/velocity/config/anymal\_c/manual\_env\_cfg.py
* 在 ManualSceneCfg 中，注释整段：
* terrain = TerrainImporterCfg(...)

1. 自建几何场景（若需可加 USD 资源）

* 在 STUDENT TODO 区，用如下写法添加几何体（示例为立方体，你可以自由选择场景）：
* AssetBaseCfg(..., spawn=sim\_utils.CuboidCfg(...), init\_state=AssetBaseCfg.InitialStateCfg(pos=[...]))
* 要求最小场景：
* 一块平地（建议 ≥ 10×10 m，位置接近原点，高度略高于 0，如 z=0.01）
* 至少2级台阶或等效障碍（尺寸、宽度、层数自定，建议设置为 kinematic 固定），或者长斜坡
* 机器人需能在你搭建的场景中稳定行走、转向，并尝试越障/上台阶（允许多次尝试）。

1. 运行与演示

* 命令（参考）：

|  |
| --- |
| Bash  ./isaaclab.sh -p scripts/reinforcement\_learning/rsl\_rl/play\_manual.py \  --task=Isaac-Velocity-Rough-Anymal-C-Manual-v0 \  --num\_envs 1 \  --checkpoint /绝对路径/到/你的/model\_14999.pt \  --real-time |

* 确认：
* WASD/QE 手动控制有效
* 相机跟随正常
* 在你自建场景中能持续运动，并完成一次基本越障

**六、交付物（DDL：下周五晚 23:59）**

将以下内容打包成压缩包提交到**bb平台**：

* 代码：
* 你修改过的 manual\_env\_cfg.py（
* 视频：
* 10-20 秒，展示你自建场景中的行走/越障过程
* 截图：
* 至少 2 张：场景布局、运行中效果
* 说明：
* PDF/Markdown（≤1页）：描述你的场景设计（平地/台阶/斜坡.....大小与位置）

**七、常见问题**

* 键盘无效：先 pip install keyboard，确保 GUI 焦点在仿真窗口
* 模型加载失败：检查 --checkpoint 路径