逐层核对驱动、框架构建、容器与设备可见范围

完整操作教程 · 参考固定 · 往返恢复验收

nvidia-smi可见设备而PyTorch找不到GPU，说明驱动工具可访问并不等于当前Python环境加载了兼容CUDA构建或容器暴露了设备。按驱动、设备可见性、框架构建逐层排查。

运行边界：诊断脚本不分配GPU；需在已授权且实际分配GPU的作业中做运算验收。

nvidia-smi与PyTorch检查的是不同层

`nvidia-smi`能运行，通常说明当前环境中的NVIDIA管理工具可以与驱动通信并显示设备；它不能证明当前Python环境中的torch是CUDA构建、CUDA运行时兼容，也不能证明容器或作业被分配了GPU。PyTorch官方建议根据操作系统、包管理器和CUDA平台选择对应安装命令，并用`torch.cuda.is_available()`检查当前进程是否可使用CUDA。

因此按层次排查：宿主机是否有GPU和驱动；当前作业是否获得设备分配；容器是否透传GPU设备；`CUDA_VISIBLE_DEVICES`是否过滤设备；Python导入的是哪个torch和Python；torch是否带CUDA构建；驱动与框架运行时是否兼容。任何一层失败，后面都可能显示不可用。不要直接重装驱动或覆盖torch，先采集版本和路径证据。

先确认出错命令使用的Python解释器和虚拟环境，运行`which python`、`python -m pip show torch`、`python -c 'import torch; print(torch.__file__, torch.__version__, torch.version.cuda)'`。如果`torch.version.cuda`为None，通常表示当前包没有CUDA运行时构建；但还需结合包来源和平台核实。若`CUDA_VISIBLE_DEVICES`为空，任务可能被调度为CPU任务或容器未获设备访问。

现象	优先核对	避免做法
torch.version.cuda为None	包构建与Python环境	反复改系统驱动
CUDA_VISIBLE_DEVICES为空	作业分配和容器透传	手工伪造环境变量
设备数为0	设备节点、权限和容器配置	只看宿主nvidia-smi
驱动初始化报错	驱动、运行时和框架版本	在共享节点盲目升级驱动

检查作业和容器能看到的设备

如果在Slurm、Kubernetes或其他调度系统中运行，确认作业申请了GPU资源，并从作业内部检查设备可见性。登录节点能看到GPU不代表计算节点上的作业获得GPU。环境变量`CUDA_VISIBLE_DEVICES`可能把物理GPU重新编号或隐藏未分配设备；不要自行将其设成“0”以绕过调度规则。

容器环境还要确认NVIDIA Container Toolkit已配置，启动参数或编排清单请求了GPU。宿主机`nvidia-smi`可用但容器内没有设备节点，常见是容器运行时未启用GPU支持、设备权限不足或挂载的驱动库不匹配。先用管理员提供的最小CUDA容器验证透传，再测试应用镜像。不要在生产容器里手工挂载宿主机所有驱动目录。

确认PyTorch包和运行时来自同一环境

Python环境混乱时，shell中的pip可能把包装到另一个解释器里。统一用`python -m pip`，打印sys.executable、torch.__file__、torch.__version__、torch.version.cuda和CUDA设备数。conda、pip和系统包管理器混装可能出现重复库；先在新虚拟环境安装官方选择器给出的匹配版本，比较结果，不要立即删除旧环境。

PyTorch预编译包通常包含所需CUDA运行时组件的一部分，但仍依赖兼容的NVIDIA驱动和设备。CUDA toolkit本地版本与torch.version.cuda不一定要完全相同；判断兼容要看PyTorch安装渠道和NVIDIA驱动支持范围，而不是仅比较

`nvcc --version`。更换torch版本后重启Python进程，因为已加载的动态库不会在当前解释器中自动刷新。

从可用性检查到实际分配验收

`torch.cuda.is_available()`返回true是必要检查之一，但最小验收还应读取设备名称、建立张量并执行一次简单运算后同步设备。随后监控显存分配和释放，确认当前作业看到的设备ID符合分配结果。若CUDA初始化失败，保存完整错误、torch版本、驱动版本、设备可见变量、容器镜像digest和作业ID。

生产GPU任务还需验证真实应用的依赖和显存需求。一个小张量可以成功，不代表生信模型、单细胞整合或大矩阵分析能在设备内存中运行。用少量授权数据做短试跑，测峰值显存、CPU内存、数据加载和运行时间，再估算全量任务。最后保存环境lockfile、安装命令、容器摘要和验收日志，避免下一次会话切换到另一个Python解释器。

什么时候需要调整环境

若驱动、容器透传和设备权限都正常，当前torch为CPU构建，优先在隔离环境按PyTorch官方选择器安装匹配的CUDA包。若框架版本较旧、CUDA运行时和驱动不兼容，由管理员评估兼容矩阵和共享节点影响；不要自行升级全机驱动。若任务没有申请GPU资源，先修改作业资源请求，而不是在Python环境里反复设置设备变量。

计算环境交付应提供可验收的GPU型号、驱动、框架版本、设备分配机制和用户权限范围。租赁环境能否运行特定软件栈，需要按任务和兼容组合逐项确认；`nvidia-smi`可见只是排查链路中的一层。

用一张兼容清单避免反复试装

将GPU型号、驱动版本、宿主CUDA能力、容器运行时、容器镜像、Python、PyTorch版本和PyTorch CUDA构建写成兼容矩阵。记录来源命令和执行位置，例如宿主机、容器内部或Slurm作业。很多排障时间浪费在比较不同机器或解释器的输出；每条结果带上主机名、容器digest、作业ID和时间，才能判断是否在同一环境。

在虚拟环境里确认`sys.executable`和`torch.__file__`。系统上可能有多个Python、多个CUDA模块或旧版torch残留。使用`python -m pip show torch`和`pip list`核查当前解释器环境；conda环境则同时记录conda list和channel。不要在没有备份的现有环境里连续安装不同CUDA wheel，避免混合依赖难以恢复。

若宿主机`nvidia-smi`正常、容器内没有设备，先由容器管理员确认NVIDIA Container Toolkit、运行时配置和设备请求。Docker单独支持GPU的参数依赖运行时；Kubernetes则通过设备插件或调度资源分配。设备被调度器分配后，`CUDA_VISIBLE_DEVICES`可能重新编号；应用中看到cuda:0不一定对应宿主物理GPU 0。应以作业分配信息和UUID映射为准。

若容器内能看到GPU但torch不可用，检查PyTorch是否为CPU包、CUDA构建是否与驱动兼容、动态库解析路径和镜像依赖。`LD_LIBRARY_PATH`乱加系统CUDA目录可能让预编译torch加载错误库。优先构建一个干净环境，按PyTorch官方安装选择器和目标平台验证，然后与原环境并行对照。只有确认版本组合后才替换生产镜像。

最小验收应分设备可见和实际计算两步

第一步让PyTorch报告CUDA是否可用、设备数量、设备名称和当前可见设备映射。第二步在一张指定设备上分配小张量，执行运算并同步，确认内核实际运行；只调用is_available不覆盖所有后续计算错误。第三步运行目标框架最小入口，检查其是否使用GPU而非静默退回CPU。最后在真实工作负载小样本上测量显存、CPU内存、运行时间和结果结构。

测试时不要申请超大显存或无界循环。记录运行前后显存、框架峰值分配、设备UUID、batch size和精度模式。PyTorch缓存分配器保留的显存可能使nvidia-smi读数与当前张量实际占用不同；报告中说明使用的是allocated、reserved还是整卡占用。完成后释放张量、退出进程，再确认设备资源回收。

如果`torch.cuda.is_available()`为false，保存完整环境信息，不要先通过修改`CUDA_VISIBLE_DEVICES`尝试绕过。若它为true但张量分配失败，可能是显存不足、设备被其他任务占用、驱动错误或限制触顶；保存具体异常类型、错误文本和设备状态。框架能看到设备但应用仍运行在CPU时，检查应用配置、模型device参数和数据张量的device。

升级驱动和CUDA前先评估共享节点

GPU驱动是宿主机层组件，升级会影响同节点多个用户、容器和正在运行的作业。普通研究者不应直接覆盖驱动或创建符号链接替代系统库。先确认当前驱动支持目标PyTorch构建，再由管理员在维护窗口安排升级。CUDA toolkit本地安装主要影响编译或特定开发场景；预编译PyTorch使用的运行时包与系统toolkit不一定同版本，不能仅因版本号不同就判断错误。

多GPU、MIG、MPS和容器设备隔离会影响设备可见数和显存限制。MIG实例与整卡不同，设备名称和显存容量可能变化；MPS客户端可能受服务端内存限制。测试环境必须反映生产分配方式，不能在管理员独占GPU的shell里通过后就假定普通用户容器可用。

把验收过程纳入软件环境交付

交付清单保存GPU型号与UUID、驱动、容器工具包、框架版本、CUDA构建、Python路径、设备可见变量、最小计算输出、任务限制和已知边界。更新镜像或驱动后重跑最小验收并记录新digest。若工作负载依赖特定CUDA扩展、编译的算子或第三方库，还要额外验证这些组件，不能用PyTorch基础张量通过替代整个软件栈通过。

GPU环境出现故障时，提供版本矩阵、作业ID和脱敏日志，比只说“nvidia-smi正常”更有助于定位。租赁或部署环境是否满足GPU任务要求，要以实际任务小样本验收和明确交付范围为准，不应仅凭显卡型号或设备列表作保证。

示例：输出当前Python进程的GPU诊断信息

在发生问题的同一个作业、容器和Python解释器里运行。该脚本只输出设备可见性和版本信息，不分配大块显存，也不修改驱动或环境变量。

```
import os, sys, torch
print("python:", sys.executable)
print("torch:", torch.__file__, torch.__version__)
print("torch CUDA build:", torch.version.cuda)
print("CUDA_VISIBLE_DEVICES:", os.getenv("CUDA_VISIBLE_DEVICES"))
print("available:", torch.cuda.is_available())
if torch.cuda.is_available():
    print("device_count:", torch.cuda.device_count())
    for i in range(torch.cuda.device_count()):
        print(i, torch.cuda.get_device_name(i))
```

如果设备可用，再在获批测试作业中创建一个小张量并执行简单运算，随后同步并释放。这里不尝试cuda:0超出范围的编号，也不将设备可见列表手工改成整卡。诊断输出附上主机名、作业ID、容器摘要和时间，发给管理员时遮盖内部主机信息。

修复后将诊断输出纳入部署验收清单，并在新作业和新容器中复跑。CPU-only torch、错误Python解释器、未分配GPU和不兼容驱动需要不同处理路径，不能使用同一个安装命令覆盖所有情况。只有在实际申请资源的环境里完成最小计算和目标软件试跑，才算验证到对应层级。

常见误判与升级边界

若PyTorch显示CUDA可用但训练异常慢，可能是程序实际把输入留在CPU、设备间拷贝频繁、batch过小、数据加载或存储成为瓶颈。检查模型参数和输入tensor的device，监控GPU利用率、显存、CPU和IO，再做一个短的代表性任务。不要以单次nvidia-smi利用率为依据断言GPU没有被使用；采样时间间隔可能错过短kernel。

如果宿主机没有nvidia-smi但任务声称要GPU，先确认节点是否分配了GPU以及设备管理工具是否安装；普通用户容器里可能无法看到宿主管理工具，但CUDA设备仍可能透传。反过来，宿主机能看到GPU不代表用户权限、容器、torch构建都正确。通过管理员提供的只读节点信息和最小容器测试逐层确认，不要自行安装驱动或修改共享系统目录。

版本兼容升级需由节点管理员审核并安排维护窗口。记录所有依赖用户、正在运行任务和回滚方案，先在测试节点验证目标镜像、PyTorch CUDA build、设备分配和代表性生信程序，再切换生产环境。若只需一个项目使用新框架，可先在隔离容器中验证，避免升级全机驱动影响其他课题组。

配套练习包

本PDF对应的ZIP包含本文代码、主题模板和SHA256校验文件。运行前核对包内清单，并按文章说明在独立目录验证。

参考资料

PyTorch本地安装选择器 · 来源资料

torch.cuda.is_available文档 · 来源资料

NVIDIA Container Toolkit安装说明 · 来源资料

操作建议：先在合成小样本上验证流程，再依据项目参考、标签合同、存储策略与获配资源验收真实数据。