

## Table of Contents

简介	1.1
Stage 1 总览	1.2
环境与依赖	1.3
1. 自设计 SDK	1.4
1. API 速查	1.4.1
函数解析: driver	1.4.2
函数解析: ik	1.4.3
2. 人机交互 (键盘/JoyCon)	1.5
2. 控制映射速查	1.5.1
3. 数据集构建 (RDT)	1.6
理论: RDT 数据格式	1.6.1
理论: 手眼标定	1.6.2
3. 相机与手眼标定	1.6.3
函数解析: 手眼评估	1.6.4
3. 时序同步说明	1.6.5
函数解析: RDT 采集链路	1.6.6
5. FPGA 逆解加速	1.7

# Stage 1 交付文档 (GitBook)

本书用于完成 Stage 1 的任务交付

## 程序设计结构

- 文档结构: `README.md` (本页) + `SUMMARY.md` (目录) + 每章独立 Markdown
- 代码结构 (按分层):
  - 驱动与控制: `driver/`
  - 运动学与 IK: `ik/`
  - 人机交互脚本: 根目录 `*_control*.py`
  - 标定与视觉: `vision/`
  - 数据集与格式: `RDT/`
  - 硬件加速: 外部仓库 `Hardware-  
Accelerated_System_for_IK_Solution_Based_on_LeRobot/`

## 脚本作用

- 1. (SDK 验证) : `arm_keyboard_control.py`、`ik_keyboard_realtime.py`
- 2. (遥操作) : `joycon_ik_control_py.py` (JoyCon), 以及键盘脚本
- 3. (采集与检  
查) : `RDT/collect_rdt_dataset_teleop.py`、`RDT/build_rdt_hdf5_from_raw.py`  
、`RDT/inspect_rdt_hdf5.py`
- 3. (标定与验  
证) : `vision/calibrate_camera.py`、`vision/handeye_calibration_*.py`、`vis  
ion/track_blue_circle_eyetohand.py` - 5. (FPGA 加  
速) : `notebook/S0101_Hardware_IK_Demo.ipynb`、`notebook/S0101_IK_HW_vs_Pyt  
hon.ipynb` (需 PYNQ-Z2)

## 章节导航

- Stage 1 总览: `stage1_overview.md`
- 环境与依赖: `env_setup.md`
- - 1. 自设计 SDK: `stage1_1_sdk.md`
  - 2. API 速查: `stage1_1_sdk_api.md`
- - 1. 人机交互 (键盘/JoyCon) : `stage1_2_hci_teleop.md`
  - 2. 控制映射速查: `stage1_2_controls.md`
- - 1. 数据集构建 (RDT) : `stage1_3_dataset.md`
  - 2. 相机与手眼标定: `stage1_3_calibration.md`

## 1. API 速查

### 3. 时序同步说明: `stage1_3_timesync.md`

理论与源码解析会在对应章节的开头给出索引:

- 运动学/IK 理论: `stage1_1_sdk.md`
- 手眼标定理论: `theory_handeye.md`
- RDT 数据格式理论: `theory_rdt_format.md`

# Stage 1 总览

## 程序设计结构

按模块分层，可以把仓库拆成 6 块：

- **驱动层**（协议与串口）： `driver/ftservo_driver.py`
- **控制层**（面向“关节名”的动作接口）： `driver/ftservo_controller.py` + `driver/*.json`
- **运动学层**（FK/IK 与角度↔步数映射）： `ik/robot.py`（及 `ik/`）
- **交互层**（键盘/JoyCon 输入到控制命令）：根目录控制脚本 + `joyconrobotics/`
- **数据/标定层**（采集、格式、标定、验证）： `RDT/` + `vision/`
- **硬件加速层**（FPGA 逆解加速）：外部仓库 `Hardware-Accelerated_System_for_IK_Solution_Based_on_LeRobot/`

## 脚本作用（按交付顺序）

### 1. SDK（先验证“能控能读”）

- `arm_keyboard_control.py`：关节级步进控制；用于快速验证串口、关节方向、软限位与回中位。
- `ik_keyboard_realtime.py`：末端级 IK 控制；用于验证 FK/IK、mask 与角度↔步数映射。

### 2. 人机交互（让控制可录屏）

- `joycon_ik_control.py`：JoyCon 遥操作到 IK 控制；重点验证速度档、Home 重置、夹爪开合与退出清理。

### 3. 标定与数据（先标定，后采集，再校验）

- `vision/calibrate_camera.py`：相机内参标定，落盘到 `session_*/`。
- `vision/handeye_calibration_eyeinhand.py` / `vision/handeye_calibration_eyetohand.py`：手眼标定与误差评估。
- `vision/track_blue_circle_eyetohand.py`：闭环验证标定有效性。
- `RDT/collect_rdt_dataset_teleop.py`：采集 raw episode。
- `RDT/build_rdt_hdf5_from_raw.py`：raw→HDF5。
- `RDT/inspect_rdt_hdf5.py`：检查张量形状/抽样可视化。

### 5. FPGA 硬件加速（IK 求解性能优化）

- `notebook/S0101_Hardware_IK_Demo.ipynb`：FPGA 硬件加速逆解演示（需 PYNQ-Z2 板）。
- `notebook/S0101_IK_HW_vs_Python.ipynb`：硬件加速与纯 Python 软件的性能对比。

- `notebook/LM_SingleStep_Benchmark.ipynb`：单步 LM 迭代性能基准测试。

## 方法作用（理解“脚本是怎么连起来的”）

下面的方法是各脚本最核心的“搭积木接口”，理解它们基本就能读懂上层脚本：

### Driver 层（通信与协议）

- `driver.ftservo_driver.FTServoDriver.sync_write(ids, address, data_list)`
  - **作用：**向多个舵机同时写入数据（如目标位置）。
  - **细节：**利用串口总线的广播或同步写指令，确保所有关节在同一时刻接收到指令并开始运动，避免“波浪式”延迟。
- `driver.ftservo_driver.FTServoDriver.sync_read(ids, address, length)`
  - **作用：**从多个舵机同步读取数据（如当前位置）。
  - **细节：**发送一次指令，让总线上的舵机按顺序返回数据包，极大提高了读取效率，保证状态采样的同时性。

### Controller 层（控制逻辑）

- `driver.ftservo_controller.ServoController.fast_move_to_pose(joint_targets, time_s)`
  - **作用：**输入 {关节名: 目标步数} 字典，驱动机械臂运动。
  - **细节：**内部先将关节名映射为 ID，进行软限位（Soft Limit）检查，然后调用 `sync_write` 下发。`time_s` 参数可用于指定运动时间（速度控制）。
- `driver.ftservo_controller.ServoController.read_servo_positions()`
  - **作用：**读取当前机械臂状态。
  - **细节：**调用 `sync_read` 获取原始步数，并根据 `homing_offset` 和方向配置，将其转换为以“零位”为基准的步数，返回 {关节名: 步数} 字典。

### IK 层（运动学解算）

- `ik.robot.Robot.fkine(q)`
  - **作用：**正运动学求解。
  - **细节：**输入关节角 \$q\$（弧度），通过 ETS（Elementary Transform Sequence）链式乘法，计算出末端执行器相对于基座的齐次变换矩阵 \$T\$（包含位置与姿态）。
- `ik.robot.Robot.ikine_LM(Tep, q0, mask, ...)`
  - **作用：**逆运动学求解（Levenberg-Marquardt 数值法）。
  - **细节：**输入目标位姿 \$T\_{ep}\$ 和初值 \$q\_0\$。算法通过迭代最小化误差 \$E = ||T(q) - T\_{ep}||^2\$ 来寻找最优关节角。`mask` 参数用于指定只关注哪些维度（如只控位置 [1,1,1,0,0,0] 或全控 [1,1,1,1,1,1]）。
- `ik.robot.Robot.q_to_servo_targets(q) / read_joint_angles()`
  - **作用：**物理空间（弧度）与驱动空间（步数）的双向映射。
  - **细节：**处理减速比、零位偏置和方向符号，是连接 IK 算法与底层驱动的桥梁。

## 验收视频

## 1. API 速查

- 1.: 终端启动日志 + 回中位 + 2~3 个动作 + 读取位姿/关节 + 退出后串口关闭。
- 2.: JoyCon/键盘实时控制 + 夹爪动作 + 漂移时 reset (如 `Home` ) + 退出。
- 3.: 采集时三相机预览 + 完成一个 episode 落盘 + raw→hdf5 + inspect 输出。

## 环境与依赖

### 设备权限与依赖清单

#### 1) 系统与设备权限

##### 串口（机械臂）

- 典型设备：`/dev/ttyACM0`、`/dev/ttyUSB0` 或 udev 规则映射出的  
`/dev/left_arm`、`/dev/right_arm`
- 若遇到权限错误（`Permission denied`），通常需要把用户加入 `dialout` 组  
（`:`

```
sudo usermod -aG dialout $USER
```

完成后重新登录生效。

##### JoyCon (hidraw)

- JoyCon 依赖 HID 设备访问权限（`/dev/hidraw*`）。
- 若枚举不到设备或报权限问题，需要配置 udev 规则/用户组权限（不同系统策略不同）。

#### 2) Python 依赖

常见依赖（用于 SDK/IK/采集/标定）：

- `numpy`
- `scipy`
- `opencv-python`（或系统自带 OpenCV）
- `h5py`（RDT HDF5）
- `pyserial`（串口）
- `pynput`（键盘控制脚本 `arm_keyboard_control.py`）

# 1. 基于 Python 的自设计 lerobot 控制 SDK

快速跳转：

- API 速查：[stage1\\_1\\_sdk\\_api.md](#)
- 函数解析（driver）：[ref\\_sdk\\_driver.md](#)
- 函数解析（ik）：[ref\\_ik\\_robot\\_solver.md](#)

产出物：

- 源代码：本仓库 `driver/` + `ik/`（核心 SDK）
- 演示视频
- 设计/实现手册：本文

## 理论框架

### 1. 背景与问题定义

逆运动学（Inverse Kinematics, IK）是机器人控制中的核心问题，其数学本质是求解非线性映射的逆。

设机器人的正运动学方程为  $\mathbf{x} = f(\mathbf{q})$ ，其中：

- $\mathbf{q} \in \mathbb{R}^n$  为关节空间变量（关节角）。
- $\mathbf{x} \in \mathbb{R}^m$  为任务空间变量（末端位姿，通常  $m=6$ ，包含位置和姿态）。

IK 的目标是：给定期望的末端位姿  $\mathbf{x}_{\text{des}}$ ，寻找一组关节角  $\mathbf{q}$ ，使得  $f(\mathbf{q}) = \mathbf{x}_{\text{des}}$ 。

挑战：

1. **非线性**： $f(\mathbf{q})$  包含大量的三角函数，是非线性的。
2. **多解性**：同一个末端位姿可能对应多组关节角（如“肘部朝上”和“肘部朝下”）。
3. **奇异性**：在某些特定构型下，雅可比矩阵秩下降，导致运动能力丧失或数值求解发散。
4. **冗余与欠驱动**：当  $n > m$  时（冗余），存在无穷多解。当  $n < m$  时（欠驱动，如本仓库的 5-DoF 机械臂），通常无法精确达到任意 6D 位姿，需要引入权重（Mask）忽略某些维度（如 Yaw 角）。

### 2. 逆运动学方法谱系：解析法 vs 数值法

解决 IK 问题主要有两大类方法：

#### 1.1 解析法（Analytic Methods）

解析法试图通过代数或几何推导，直接得到  $\mathbf{q}$  关于  $\mathbf{x}$  的闭式数学表达式（Closed-form solution）。

- **几何法**: 利用机械臂的几何结构（连杆长度、关节偏移），通过余弦定理、三角恒等式构建几何约束方程，逐步解出各个关节角。优点：计算速度极快（微秒级），可获得所有可能的解。缺点：严重依赖特定构型，推导过程复杂，通用性差。
- **代数法**: 将 FK 矩阵方程中的元素提取出来，通过变量代换消元，将超越方程转化为多项式方程求解。

## 1.2 数值法 (Numerical Methods)

数值法将 IK 视为一个非线性优化问题，从一个初始猜测值  $q_0$  开始，通过迭代更新  $q$  来最小化末端误差  $e(q) = x_{des} - f(q)$ 。优点：通用性强，适用于任意构型的串联机械臂；易于处理冗余自由度和避障约束。缺点：计算量相对较大，可能陷入局部极小值，收敛性受初值影响。

**本 SDK 采用数值法**，因为它能统一处理不同构型的机械臂，且易于集成 Mask 权重功能。

## 3. 数值求解算法详解

数值法的核心在于迭代更新律： $q_{k+1} = q_k + \Delta q$ 。不同的算法主要区别在于  $\Delta q$  的计算方式。

基于一阶泰勒展开，末端误差与关节增量的关系为： $e \approx J \Delta q$ 。

### 3.1 牛顿-拉夫逊法 (Newton-Raphson)

牛顿法试图一步到位消除线性化后的误差。

$$\Delta q = J^{-1} e$$

若  $J$  不可逆（非方阵或奇异），则使用伪逆  $J^{\dagger}$ ：

$$\Delta q = J^\dagger e = (J^T J)^{-1} J^T e$$

- **特点**: 收敛速度快（二次收敛），但在奇异点附近 ( $J^T J$  接近奇异) 极其不稳定，会导致  $\Delta q$  剧烈震荡。

### 3.2 梯度下降法 (Gradient Descent)

梯度下降法沿着误差函数的负梯度方向更新，不涉及矩阵求逆。

$$\Delta q = \alpha J^T e$$

- **特点**: 计算简单，绝对稳定（不会发散），但收敛速度慢（线性收敛），容易在平坦区域“之”字形震荡。

### 3.3 Levenberg-Marquardt (LM) 算法

LM 算法是牛顿法与梯度下降法的结合体 (Damped Least Squares)，是工程中最常用的 IK 算法。它引入了阻尼因子  $\lambda$ ：

$$\Delta q = (J^T J + \lambda^2 I)^{-1} J^T e$$

- **机制：**
  - 当误差较大或接近奇异时，增大  $\lambda$ ，算法行为接近梯度下降（保证稳定性）。
  - 当误差较小且远离奇异时，减小  $\lambda$ ，算法行为接近牛顿法（加速收敛）。
- **本仓库实现：** `ik/solver.py` 中实现了标准的 LM 求解器，并支持 `mask` 权重矩阵  $W_e$ ，实际求解方程为：

$$(J^T W_e J + \lambda^2 I) \Delta q = J^T W_e e$$

## 4. 雅可比矩阵 (Jacobian Matrix)

雅可比矩阵  $J(q) \in \mathbb{R}^{m \times n}$  是数值法的核心，描述了关节速度  $\dot{q}$  到末端速度  $\dot{x}$  的线性映射： $\dot{x} = J(q) \dot{q}$ 。

### 4.1 数值雅可比 (Numerical Jacobian)

通过有限差分法近似计算偏导数。对于第  $j$  列  $J_j$ ：

$$J_j \approx \frac{f(q + \delta \cdot u_j) - f(q - \delta \cdot u_j)}{2\delta}$$

其中  $u_j$  是第  $j$  个分量为 1 的单位向量， $\delta$  是微小步长。

- **优点：** 实现极其简单，无需推导公式，只要有 FK 函数即可。
- **缺点：** 计算量大（计算一列需要运行 FK），精度受步长  $\delta$  影响（截断误差 vs 舍入误差）。

### 4.2 解析/几何雅可比 (Analytic/Geometric Jacobian)

基于刚体运动学公式，直接利用连杆的几何信息（旋转轴  $z_i$  和位置  $p_i$ ）构造矩阵。对于旋转关节，第  $i$  列通常形式为：

$$J_i = \begin{bmatrix} z_{i-1} \times (p_e - p_{i-1}) \\ z_{i-1} \end{bmatrix}$$

- $z_{i-1}$ ：第  $i$  关节轴在基座坐标系下的单位向量。
- $p_e$ ：末端执行器原点位置。
- $p_{i-1}$ ：第  $i$  关节原点位置。
- **优点：** 计算精确，效率高（仅涉及向量运算），无数值误差。
- **缺点：** 需要准确的运动学模型参数，推导相对繁琐。本 SDK 优先使用解析法计算雅可比以保证实时性。

## 5. 舵机步数 $\leftrightarrow$ 关节角 (弧度)

### 5.1 编码器计数

项目默认每转编码器计数：

$$\text{counts\_per\_rev} = 4096$$

因此每弧度计数：

$$\text{counts\_per\_rad} = \frac{4096}{2\pi}$$

## 5.2 home\_pose (零位参考)

在 `ServoController` 中，`home_pose` 默认取舵机允许范围的中点（夹爪例外取 `range_min`）：

$$\text{home\_pose} = \left\lfloor \frac{\text{range\_min} + \text{range\_max}}{2} \right\rfloor$$

**注意：**仓库里 JSON 里存在 `homming_offset`，但当前 `ServoController.home_pose` 的计算不使用 `homming_offset`（这是工程选择，便于把“零位参考”和“机械装配标定偏置”分开看）。

## 5.3 关节角→目标步数（前向映射）

`ik/robot.py::Robot.q_to_servo_targets()` 的核心公式：

设第 \$i\$ 个关节：

- 关节角  $q_i$  (rad)
- 方向符号  $s_i \in \{+1, -1\}$  (`gear_sign`)
- 传动比  $g_i$  (`gear_ratio`)

则目标步数：

$$\text{steps}_i = \text{home\_pose}_i + s_i g_i q_i \text{counts\_per\_rad}$$

## 5.4 步数→关节角（反向映射）

`ik/robot.py::Robot.read_joint_angles()` 的核心公式：

$$q_i = \frac{s_i (\text{steps}_i - \text{home\_pose}_i)}{\text{counts\_per\_rad} g_i}$$

这也是 2./3. 里把“舵机实际位置”写入 `proprio` 的基础。

## 6. 正运动学 (FK) : ETS 链

本仓库 SO101 的 FK 由 Elementary Transform Sequence (ETS) 链表达：

$$T(q) = \prod_k T_k(q)$$

在 `ik/robot.py::create_so101_5dof(_gripper)` 中，链由 `ET.tx/ET.tz/ET.Rx/ET.Ry/ET.Rz` 组成。

- 平移： $T_x(a)$ 、 $T_z(d)$
- 旋转： $R_x(\theta)$ 、 $R_y(\theta)$ 、 $R_z(\theta)$

实现入口：

- `Robot.fkine(q) → ets.fkine(q)` 返回 \$4\times 4\$ 齐次矩阵

## 7. 逆运动学 (IK) : LM 数值法 (本仓库实现)

IK 目标：给定期望末端位姿  $T^*(=T_{ep})$ ，求关节角  $q$  使  $T(q) \approx T^*$ 。

### 7.1 位姿误差：角轴 (angle-axis) 形式

`ik/solver.py::_angle_axis(Te, Tep)` 计算 6 维误差：

$$\mathbf{e} = \begin{bmatrix} \mathbf{e}_p \\ \mathbf{e}_R \end{bmatrix} \in \mathbb{R}^6$$

- 位置误差：

$$\mathbf{e}_p = p^* - p$$

- 旋转误差：

$$\mathbf{R}_{err} = \mathbf{R}^* \mathbf{R}^T$$

并由  $\mathbf{R}_{err}$  转为轴角向量（代码通过反对称项与迹计算）。

### 7.2 mask (权重) 与代价函数

`mask=[w_x, w_y, w_z, w_roll, w_pitch, w_yaw]`，在代码中构成对角权重矩阵：

$$\mathbf{W}_e = \text{diag}(\mathbf{mask})$$

代价函数：

$$E(q) = \frac{1}{2} \mathbf{e}(q)^T \mathbf{W}_e \mathbf{e}(q)$$

直观理解：mask 为 0 的维度在优化中被忽略；介于 0~1 则是“弱约束”。

### 7.3 LM 更新公式 (实现等价)

`ikine_LM()` 核心更新：

$$\mathbf{q} \leftarrow \mathbf{q} + (\mathbf{J}^T \mathbf{W}_e \mathbf{J} + \mathbf{W}_n)^{-1} \mathbf{J}^T \mathbf{W}_e \mathbf{e}$$

其中  $\mathbf{J}$  为雅可比：`ets.jacob0(q)`。

阻尼矩阵  $\mathbf{W}_n$  在本仓库支持三种风格：

- `chan` :  $\mathbf{W}_n = k_E \mathbf{I}$
- `wampler` :  $\mathbf{W}_n = k_I \mathbf{I}$
- `sugihara` :  $\mathbf{W}_n = (E + k) \mathbf{I}$

## 7.4 多次搜索 (slimit)

`slimit` 表示允许多次随机初值重试：

- 第 1 次使用 `q0`
- 后续使用 `$[-\pi,\pi]$` 的随机初始化

这样能降低陷入局部最优的概率。

# 程序设计结构

本 SDK 采用分层架构设计，旨在解耦硬件通信、运动控制与运动学解算，便于后续扩展不同型号的机械臂或更换底层通信协议。

## 1.1 硬件通信层 (Driver Layer)

- **职责：**负责与底层硬件（如飞特串行总线舵机）进行原始字节流通信。
- **核心文件：**
  - `driver/ftservo_driver.py`：实现了 FT Servo 串口通信协议。
    - 封装了 `ping`（心跳）、`read`（读寄存器）、`write`（写寄存器）等基础指令。
    - 实现了 `sync_read`（同步读）和 `sync_write`（同步写），这是实现多关节同步控制的关键。
    - 处理了协议的校验和（Checksum）计算与验证，确保通信可靠性。

## 1.2 控制抽象层 (Controller Layer)

- **职责：**屏蔽底层 ID 和寄存器细节，提供面向“关节名称”的高级控制接口。
- **核心文件：**
  - `driver/ftservo_controller.py`：定义了 `ServoController` 类。
    - 维护 `joint_name` 到 `servo_id` 的映射。
    - 实现了软限位保护（Soft Limit），防止机械臂运动超出物理范围。
    - 提供了“回中位（Home）”逻辑，这是所有运动控制的基准。
  - **配置文件：**
    - `driver/servo_config.json`：定义了出厂默认的舵机 ID、零位脉冲值和物理限位。
    - `driver/left_arm.json` / `driver/right_arm.json`：针对具体机械臂（左/右）标定后的配置文件。在实际运行中，应优先加载这两个文件以获得准确的零位偏置。

## 1.3 运动学/IK 层 (Kinematics Layer)

- **职责：**实现笛卡尔空间（末端位姿）与关节空间（舵机角度）的相互转换。
- **核心文件：**
  - `ik/robot.py`：构建了 SO101 机械臂的运动学模型。
    - 实现了 `q_to_servo_targets`：将物理角度（弧度）映射为舵机脉冲步数。
    - 实现了 `read_joint_angles`：将读取到的脉冲步数反解为物理角度。
    - 封装了正运动学（FK）和逆运动学（IK）的调用接口。

- `ik/` 目录下的其他文件：
  - `DH.py / et.py`：实现了 DH 参数法和 ET (Elementary Transform) 序列法建模。
  - `solver.py`：实现了基于 Levenberg-Marquardt (LM) 算法的数值逆运动学求解器。

## 脚本作用

1. 的验收通常不直接看 `driver/` 或 `ik/` 的单元函数，而是用两个最短路径脚本把分层串起来：
2. `arm_keyboard_control.py`：
  - 作用：**关节级步进控制**（直接对目标步数做增量），用于验证串口通信、关节方向、软限位与回中位。
  - 适用：第一次接硬件时优先跑它，避免 IK 不收敛等高层问题干扰排错。
3. `ik_keyboard_realtime.py`：
  - 作用：**末端级实时 IK 控制**（目标位姿 → IK → 关节角 → 步数下发），用于验证 FK/IK、mask 配置与角度↔步数映射。
  - 适用：在关节级控制稳定后，用它验证“末端位姿控制”链路。

## 方法作用

### Driver 层（通信/协议）

- `FTServoDriver.ping(id) / read(id, addr, len) / write(id, addr, val)`
  - 作用：基础通信指令。
  - 细节：直接操作串口发送符合飞特协议的数据包。`ping` 用于检测舵机在线状态；`read/write` 用于读写单个寄存器（如 PID 参数、当前位置）。
- `FTServoDriver.sync_read(ids, addr, len)`
  - 作用：同步读取。
  - 细节：发送一条指令让总线上指定 ID 的舵机依次返回数据。这是实现高频（如 50Hz+）状态更新的关键，避免了轮询带来的累积延迟。
- `FTServoDriver.sync_write(ids, addr, data_list)`
  - 作用：同步写入。
  - 细节：广播一条指令，让所有舵机在收到指令的瞬间同时更新寄存器（如目标位置）。保证了多关节运动的协调性。
- **校验和计算 (`_calc_checksum`)**
  - 作用：数据完整性验证。
  - 细节：协议规定的 Checksum 算法。在读取数据时，若计算值与接收值不符，驱动层会抛出异常或重试，防止错误数据进入控制环。

### Controller 层（关节名→舵机 ID 与保护逻辑）

- `ServoController.__init__(config_path)`
  - 作用：初始化与配置加载。
  - 细节：加载 JSON 配置文件，建立 `joint_name -> servo_id` 的映射表，并读取 `homing_offset` 和软限位范围。

- `ServoController.soft_move_to_home(time_s=2.0)`
  - **作用:** 平滑回中。
  - **细节:** 规划一条从当前位置到零位的插值轨迹，在 `time_s` 秒内执行完毕。相比直接下发零位，这能有效避免机械臂剧烈抖动。
- `ServoController.fast_move_to_pose(joint_targets)`
  - **作用:** 快速下发目标。
  - **细节:** 接收 {关节名: 目标步数}，内部进行 ID 转换和软限位检查，然后调用 `sync_write` 立即执行。
- `ServoController.read_servo_positions()`
  - **作用:** 获取当前状态。
  - **细节:** 调用 `sync_read` 读取原始步数，减去 `homing_offset` 并根据方向配置处理符号，返回物理意义上的关节步数。
- `ServoController.limit_position(name, pos)`
  - **作用:** 软限位保护。
  - **细节:** 检查目标步数是否在 `[range_min, range_max]` 范围内。若越界，则强制截断到边界值，并打印警告。

## IK 层（角度↔位姿↔步数）

- `Robot.fkine(q)`
  - **作用:** 正运动学。
  - **细节:** 输入关节角 \$q\$ (弧度)，通过连乘各连杆的变换矩阵，计算末端坐标系相对于基座的位姿 \$T\$。
- `Robot.ikine_LM(Tep, q0, mask, ...)`
  - **作用:** 逆运动学 (LM 算法)。
  - **细节:** 求解  $\min_q \|T(q) - T_{\{ep\}}\|^2$ 。利用雅可比矩阵 \$J\$ 迭代更新 \$q\$。mask 向量 (如 [1,1,1,1,1,0]) 用于在误差计算中屏蔽掉不需要控制的自由度 (如 5-DoF 机械臂无法独立控制 Yaw)。
- `Robot.read_joint_angles()`
  - **作用:** 读取物理角度。
  - **细节:** 先调用 `controller.read_servo_positions` 得到步数，再利用  $(steps - zero) / steps\_per\_rad$  公式转换为弧度。
- `Robot.q_to_servo_targets(q)`
  - **作用:** 角度转步数。
  - **细节:** IK 输出的是理论弧度，该方法将其转换为舵机能识别的脉冲步数，是下发控制前的最后一步变换。

## 关键配置

### 1. 串口设备

- **Linux 设备名:** 通常为 `/dev/ttyACM0`。
- **udev 规则:** 建议配置 udev 规则将串口映射为固定名称 (如 `/dev/left_arm`)，避免插拔后设备号变动。
  - **示例规则:** `SUBSYSTEM=="tty", ATTRS{idVendor}=="xxxx", ATTRS{idProduct}=="xxxx", SYMLINK+="left_arm"`

### 2. 波特率

- 默认值: `1000000` (1Mbps)。
- 注意: 所有舵机的波特率必须一致, 否则无法通信。

### 3. 舵机配置文件详解

以 `driver/right_arm.json` 为例, 每个关节的配置项如下:

```
"shoulder_pan": {
    "id": 1,           // 舵机物理 ID
    "homing_offset": 0, // 零位偏置 (单位: 步)。用于修正机械装配误差。
                        // 实际零位 = 理论中点 + homing_offset
    "range_min": 0,    // 最小允许步数 (软限位)
    "range_max": 4096  // 最大允许步数 (软限位)
}
```

#### 重要提示:

- `ServoController` 在初始化时会读取此配置。
- `limit_position()` 方法会严格检查目标步数是否在 `[range_min, range_max]` 范围内, 超出则会被截断并打印警告。

## 3. SDK 核心 API (对接遥操作/采集的最小集合)

本节列出了在开发上层应用 (如遥操作、数据采集) 时最常用的 API。

### 3.1

#### `driver.ftservo_controller.ServoController`

- 初始化

```
controller = ServoController(port="/dev/ttyACM0", baudrate=1000000, config
```

- 回中位 (复位)

```
# 快速回中 (慎用, 可能产生冲击)
controller.move_all_home()

# 平滑回中 (推荐, 带插值)
# step_count: 插值点数, interval: 每步间隔秒数
controller.soft_move_to_home(step_count=20, interval=0.05)
```

- 多关节同步控制 (核心)

```
# targets_dict: {关节名: 目标步数}
# speed: 舵机内部速度参数 (0-1000, 0为最快)
targets = {"shoulder_pan": 2048, "shoulder_lift": 2000}
controller.fast_move_to_pose(targets, speed=0)
```

原理: 底层使用 `sync_write` 指令, 确保所有舵机在同一时刻开始运动。

- 读取当前位置

```
# 返回: {关节名: 当前步数}
positions = controller.read_servo_positions()
```

## 3.2 ik.robot.Robot

- 创建模型

```
from ik.robot import create_so101_5dof
robot = create_so101_5dof()
```

- 正运动学 (FK)

```
# q: 关节角数组 (弧度)
# 返回: 4x4 齐次变换矩阵 (numpy array)
T_end = robot.fkine(q)
```

- 逆运动学 (IK)

```
# Tep: 目标末端位姿 (4x4 矩阵)
# q0: 初始猜测角 (通常取当前关节角)
# mask: 6维权重向量 [x, y, z, rx, ry, rz], 1表示控制该维度, 0表示忽略
q_sol, success, reason = robot.ikine_LM(Tep, q0=current_q, mask=[1, 1, 1, 0, 0,
```

- 角度与步数转换

```
# 角度转步数 (用于下发控制)
targets = robot.q_to_servo_targets(q_rad, robot.joint_names, controller.home_pos)

# 步数转角度 (用于更新状态)
q_current = robot.read_joint_angles(robot.joint_names, controller.home_pos)
```

## 4. 运行验证

### 4.1 直接关节步进 (不依赖 IK)

- 运行:
  - python arm\_keyboard\_control.py
- 行为: 按键直接给各舵机加/减步数; 支持回中位。

### 4.2 实时 IK 控制 (末端位姿控制)

- 运行:
  - python ik\_keyboard\_realtime.py
- 行为: 读取当前关节角作为起点, 实时 IK 控制末端 pos/rpy。

建议录制内容 (验收友好) :

- 1) 启动脚本并完成回中位/同步当前位置
- 2) 在安全空间内做 2~3 个方向的小幅位移
- 3) 退出并正常关闭串口

## 5. 常见问题

- 串口权限：若报 `Permission denied`，需给当前用户 `dialout` 权限或调整 `udev` 规则。
- 舵机乱动/超限：优先检查 `driver/*_arm.json` 的 `homing_offset` 与 `range_min/range_max`。
- IK 不收敛：减小每步位姿增量、放宽/调整 `mask`，并确保初值 `q0` 为当前实际关节角。

# 1. API 速查 (SDK)

本页给出“写遥操作/采集脚本时最常用”的 SDK 接口速查，详细实现以源码为准。

上层脚本（遥操作/采集）只需要关心两件事：

1) 状态怎么读：读关节当前步数/关节角，用于 UI 显示、IK 初值与闭环  
2) 目标怎么下发：把“想要的动作”（关节角/末端位姿）变成目标步数并同步下发

因此 API 速查也围绕“读→算→下发”闭环组织。

## 程序设计结构

- 控制入口：`driver.ftservo_controller.ServoController`（面向关节名，内部调用同步读写）
- 运动学入口：`ik.robot / ik.robot.Robot`（FK/IK 与角度↔步数转换）

## 脚本作用

- `arm_keyboard_control.py`：主要用 `ServoController` 做步数级控制与读回。
- `ik_keyboard_realtime.py`：同时用 `ServoController`（读/写）与 `Robot`（FK/IK 转换）。
- `joycon_ik_control_py.py`：本质与 IK 键盘控制一致，只是输入换成 JoyCon。

## 方法作用

### 1.

#### `driver.ftservo_controller.ServoController`

位置：`driver/ftservo_controller.py`

##### 1.1 初始化

- `ServoController(port: str, baudrate: int, config_path: str)`

配置文件建议使用：

- 左臂：`driver/left_arm.json`
- 右臂：`driver/right_arm.json`

##### 1.2 常用控制

- 回中位：`move_all_home() / soft_move_to_home(step_count=..., interval=...)`
- 单关节：`move_servo(name: str, position_steps: int, speed: int = ...)`

## 1. API 速查

- 多关节: `fast_move_to_pose(pose: dict[str,int], speed: int = ...)`
- 限位: `limit_position(name: str, position_steps: int) -> int`
- 关闭: `close()`

## 2. `ik.robot` (运动学/IK)

位置: `ik/robot.py`

### 2.1 创建模型

- `create_so101_5dof()`
- `create_so101_5dof_gripper()`

### 2.2 FK/IK

- FK: `robot.fkine(q_rad: np.ndarray) -> np.ndarray(4,4)`
- IK: `robot.ikine_LM(Tep=..., q0=..., tol=..., ilimit=..., mask=..., method=...)`

### 2.3 读关节角与下发目标

- 读当前关节角:
  - `robot.read_joint_angles(joint_names, home_pose, gear_sign, verbose=...) -> q_rad`
- 角度→舵机步数:
  - `robot.q_to_servo_targets(q_rad, joint_names, home_pose, gear_ratio, gear_sign) -> dict[name, steps]`

## 3. 典型控制闭环 (最小模板)

1) 读当前关节角 `q0` 2) FK 得到当前末端位姿 `T_now` 3) 构造目标位姿 `T_goal` 4)  
IK 得到 `q1` 5) 转换为舵机步数 `targets` 6) `limit_position()` 7)  
`fast_move_to_pose()` 下发

## 函数解析：舵机驱动与控制 ( driver/ )

本章聚焦 1. SDK 的“硬件通信层 + 控制抽象层”，对应源码：

- `driver/ftservo_driver.py`
- `driver/ftservo_controller.py`

## 程序设计结构

- `FTServo`：协议与串口封装（组包、校验、收发、同步读写）
- `ServoController`：面向关节名的控制抽象（加载配置、软限位、回中、同步下发）

## 脚本作用

- `arm_keyboard_control.py`：直接使用 `ServoController` 做步数级控制与回中。
- `ik_keyboard_realtime.py` / `joycon_ik_control_py.py`：在 IK 之后用 `ServoController` 下发目标并读回当前位置。
- `RDT/collect_rdt_dataset_teleop.py`：采集时读 proprio/下发 action，同样依赖控制层稳定。

## 方法作用

下文按类/方法逐个解释职责、输入输出与常见失败模式。

### 1. `driver.ftservo_driver.FTServo`

定位：串口协议封装，提供「组包→发送→解析应答」能力。

#### 1.1 `__init__(port, baudrate, timeout)`

- 作用：打开串口 `serial.Serial(port, baudrate, timeout=...)`
- 失败模式：端口不存在/无权限/波特率不匹配

#### 1.2 `build_packet(ID, instruction, params=None)`

- 作用：构造指令包 `HEADER(0xFF,0xFF) + [ID, length, instruction] + params + checksum`
- `length = len(params) + 2`（包含 `instruction` 与 `checksum`）

#### 1.3 `checksum(data)`

- 作用：返回按协议定义的校验：

$$\text{chk} = \sim (\sum \text{data}) \& 0xFF$$

## 1.4 send\_packet(packet)

- 作用：向串口写入字节流

## 1.5 read\_response()

- 作用：读取应答包并返回 dict：
  - `id/length/error/params/checksum/valid`
- 关键点：会校验 checksum ( valid )
- 失败模式：读取超时、帧头不匹配、长度不足

## 1.6 基本指令

( `ping/read_data/write_data/...` )

- `ping(ID)`：发 ping 并读应答
- `read_data(ID, start_addr, length)`：读寄存器片段
- `write_data(ID, start_addr, data_bytes)`：写寄存器片段
- `sync_write(start_addr, data_len, servo_data_dict)`：广播同步写
- `sync_read(start_addr, data_len, ids)`：广播同步读（逐个收应答）

工程上，本项目主要用到的寄存器地址：

- `0x2A`：写入「位置(2B)+时间(2B)+速度(2B)」
- `0x38`：读取当前位置 (2B)

## 2.

### `driver.ftservo_controller.ServoController`

定位：对上提供“按关节名控制”，对下调用 `FTServo`。

## 2.1 初始化与配置加载

- `__init__(port, baudrate, config_path)`：加载 JSON 配置为 `self.config`
- `id_map`：把 `id -> joint_name` 反向映射
- `home_pose`：
  - 非夹爪：取 `(range_min + range_max)//2`
  - 夹爪：取 `range_min`

## 2.2 限位保护： `limit_position(name, target_pos)`

- 输入：关节名、目标步数
- 输出：裁剪到 `[range_min, range_max]` 后的步数
- 副作用：若被裁剪会打印警告

### 2.3 单关节控制: `move_servo(name, target_pos, speed=1000)`

- 组包格式: `[pos_lo, pos_hi, 0, 0, speed_lo, speed_hi]`
- 调用: `FTServo.write_data(id, 0x2A, data)`
- 失败模式: 串口写失败、舵机无应答、`resp['error']!=0` 或 `valid=False`

### 2.4 多关节同步控制

- `move_group(targets_dict)`: 固定 `speed=1000` 的同步写
- `fast_move_to_pose(target_dict, speed=...)`: 支持 int 或 dict 速度

底层都是 `FTServo.sync_write(0x2A, 6, servo_data)`。

### 2.5 回中位与缓动

- `move_to_home(name)`: 单关节回中
- `move_all_home()`: 全部关节同步回中
- `soft_move_to_home(step_count, interval)`: 读当前步数→线性插值→逐步 `sync_write`
- `soft_move_to_pose(target_dict, step_count, interval)`: 同理, 但目标由 `target_dict` 指定

### 2.6 读取位

置: `read_servo_positions(joint_names=None, verbose=False)`

- 底层: `FTServo.sync_read(0x38, 2, ids)`
- 输出: `{joint_name: position_steps}`

## 3. 最小调用链

- 回中位: `ServoController.move_all_home()`
- 读实际位置: `ServoController.read_servo_positions()`
- 目标步数 (来自 IK) : `Robot.q_to_servo_targets()`
- 限位: `ServoController.limit_position()`
- 下发: `ServoController.fast_move_to_pose()`

## 函数解析：运动学与 IK ( ik/ )

本章对应 1./2. 的“算法层 SDK”，主要文件：

- `ik/robot.py`
- `ik/solver.py`

运动学/IK 层要解决的是：

- **正运动学**：给定关节角  $q$ ，计算末端位姿  $T(q)$
- **逆运动学**：给定目标末端位姿  $T^*$ ，求一个关节角解  $q$  使  $T(q) \approx T^*$
- **工程闭环**：把  $q$  与舵机步数互相转换，保证 IK 输出能真正驱动硬件并能从硬件读回更新状态

## 程序设计结构

- `ik.robot.Robot`：面向工程使用的封装（FK/IK 调用 + 角度↔步数转换）
- `ik.solver`：数值法 IK 求解器（LM/GN/NR/QP 等）
- SO101 模型构建：`create_so101_*` 把 ETS 参数、限位、符号约定组合成可用模型

## 脚本作用

- `ik_keyboard_realtime.py`：末端 IK 控制的最小验证脚本。
- `joycon_ik_control_py.py`：遥操作脚本（需要 IK 在实时循环内收敛）。
- `RDT/collect_rdt_dataset_teleop.py`：采集时会持续读回关节角/末端状态写入 unified vector。

## 方法作用

下文按类/函数解释各方法在“读状态→构造目标→求解→下发”的链路中承担的职责。

### 1. `ik.robot.Robot`

#### 1.1 `q_to_servo_targets(q_rad, ..., counts_per_rev=4096, gear_ratio, gear_sign)`

- 作用：关节角 (rad) → 舵机目标步数 (steps)
- 公式见：`stage1_1_sdk.md`
- 关键依赖：`home_pose` (若未传入则从 `servoController.home_pose` 取)

常见错误：

- 未传 `home_pose` 且 `ServoController` 未能初始化 → 抛 `ValueError`

### 1.2 `read_joint_angles(joint_names=None, home_pose=None, gear_sign=None, gear_ratio=None)`

- 作用：读舵机当前步数 → 反解关节角 (rad)
- 底层依赖：`ServoController.read_servo_positions()`

### 1.3 `fkine(q)`

- 作用：FK，返回 \$4\times4\$ 齐次矩阵
- 实现：`return self.ets.fkine(q)`

### 1.4 `ikine_LM(Tep, q0=None, ..., mask=None, k=..., method='chan')`

- 作用：LM 数值 IK
- 真实实现：调用 `ik/solver.py::ikine_LM(self.ets, ...)`
- 关键参数：
  - `q0` 初值（强烈建议传入当前关节角）
  - `mask` （位置/姿态权重）
  - `ilimit/slimit/tol` （迭代/多次搜索/收敛阈值）

同类接口：`ikine_GN/ikine_NR/ikine_QP`。

## 2. `ik.solver.ikine_LM` (核心求解器)

### 2.1 `_angle_axis(Te, Tep)`

- 输入：当前位姿  $T_e$  与目标位姿  $T_{ep}$
- 输出：6D 误差  $e = [\Delta p, \Delta R]$ 
  - $\Delta p = p^* - p$
  - $\Delta R$  用  $R_{err} = R^* R^T$  转为轴角向量

## 2.2 代价函数与权重

- `mask` →  $W_e = \text{diag}(mask)$
- 代价： $E = \frac{1}{2} e^T W_e e$

## 2.3 LM 更新

- 雅可比：`J = ets.jacob0(q)`
- 阻尼：`wn` 随 `method` 不同而不同 (chan/wampler/sugihara)
- 更新： $q \leftarrow q + (J^T W_e J + W_n)^{-1} J^T W_e e$

## 2.4 多次搜索 (slimit)

- 第 1 次用 `q0`

## 1. API 速查

- 后续随机初始化重试

失败模式：

- 矩阵奇异（`LinAlgError`）导致中断当前搜索
- 迭代耗尽：返回 `IKResult(False, q, 'iteration limit reached')`

## 3. SO101 模型构建函数

- `create_so101_5dof()`：5DOF ETS + `gear_sign/gear_ratio/qlim`
- `create_so101_5dof_gripper()`：在 5DOF 末端后额外增加固定平移段（工具长度）

工程建议：

- 控制脚本优先使用 `create_so101_5dof_gripper()`（末端位置更贴近实际夹爪）。

## 2. 基于自设计 SDK 的遥操作/键盘等人机交互

快速跳转：

- 控制映射速查：`stage1_2_controls.md`
- 理论推导（FK/IK/LM）：`stage1_1_sdk.md`

对应 `document/task.txt` 的 2. 产出物：

- 源代码：本仓库根目录脚本 + `joyconrobotics/`
- 演示视频：
- 设计/实现手册：本文
- 控制误差分析文档：本文“误差分析”

本章解决的问题是：给定人类输入（键盘/JoyCon），如何在每个控制周期构造目标末端位姿  $T^{*}$ ，并通过数值 IK 求解关节角  $q$ ，最终下发到舵机步数。

统一控制闭环（与 1. 保持一致）：

1) 读取当前关节角  $q_t$ （由步数反解） 2) FK 得到当前末端位姿  $T(q_t)$  3) 将输入映射为位姿增量，构造  $T_{t+1}$  4) LM-IK：求  $q_{t+1}$  使  $T(q_{t+1}) \approx T_{t+1}$  5) 角度→步数，下发并做限位裁剪

相关推导（误差定义、mask、LM 更新）集中在：`stage1_1_sdk.md`。

## 程序设计结构

本章程序结构可以抽象为“输入 → 目标构造 → IK → 下发”的稳定控制环：

1) **输入层**：键盘（`pynput` 或非阻塞 `stdin`）/ JoyCon（`joyconrobotics/`）输出离散按键或连续姿态/摇杆量  
 2) **目标构造层**：把输入映射为  $\Delta p$  /  $\Delta rpy$ ，与“基准位姿”合成  $T$   
 3) **求解层**：用 `ik.robot.Robot.ikine_LM` 求  $q$ ，必要时用 `mask` 只控制子空间  
 4) **执行层**：`Robot.q_to_servo_targets` 把角度变成步数，`ServoController.limit_position` 截断，`ServoController.fast_move_to_pose` 同步下发  
 5) \***辅助层**：速度档、Home 复位、退出清理与按钮事件处理（夹爪/连发）

## 脚本作用

### 1. 键盘（关节级）：`arm_keyboard_control.py`

该脚本提供最基础的单关节调试功能。它绕过了复杂的运动学解算，直接对每个舵机的目标步数（steps）进行增量控制。主要用于：验证串口通信、确认关节方向、测试软限位、快速回中。

### 2. 键盘（末端位姿级 IK）：`ik_keyboard_realtime.py`

该脚本实现了基于逆运动学（IK）的笛卡尔空间控制。用户通过键盘控制机械臂末端在 X/Y/Z 轴移动或 Roll/Pitch/Yaw 旋转。主要用于：验证 IK 求解器表现、检查运动学模型一致性、体验末端控制。

### 3. JoyCon (末端位姿级 IK) : joycon\_ik\_control\_py.py

这是本项目最核心的遥操作脚本，用于 3. 阶段的数据集采集。利用 Switch Joy-Con 手柄的摇杆和姿态传感器（IMU）控制机械臂。主要特点：6-DoF 控制、平滑体验（速度/死区）、所见即所得。

### 4. JoyCon 输入库: joyconrobotics/

封装了 HID 通信与原始数据解析，提供面向对象的摇杆/IMU 接口。

## 方法作用

### 关节级控制 ( ArmKeyboardController )

- `__init__`
  - 作用：初始化控制器。
  - 细节：初始化 `ServoController`，并读取当前所有舵机的实际位置作为控制起点。若读取失败，则默认使用 `home_pose`。
- `update_joint(name, delta)`
  - 作用：执行单关节运动。
  - 细节：计算 `new_pos = current_pos + delta`，调用 `controller.limit_position` 进行软限位裁剪，最后组装协议包通过 `sync_write` 下发。
- `reset_to_home()`
  - 作用：回中位。
  - 细节：调用 `controller.soft_move_to_home` 实现带插值的平滑回中，避免急停急启对机械结构造成冲击。
- `on_press(key)`
  - 作用：按键映射。
  - 细节：`pynput` 库的回调函数，查表将按键映射为特定关节的步数增量（默认 `step=100`）。

### 实时 IK 控制 ( ik\_keyboard\_realtimetime.py / joycon\_ik\_control\_py.py )

- `main() / run()` (主循环)
  - 作用：控制系统的核心调度器。
  - 细节：
    1. 初始化：连接舵机，回中，读取当前 `q0` 并 FK 得到初始位姿 `$T_{now}$`。
    2. 输入处理：非阻塞读取键盘或 JoyCon 状态（摇杆/IMU）。
    3. 目标构建：根据输入更新基准位姿 `(x, y, z, r, p, y)`，调用 `build_target_pose` 生成 `$T_{target}$`。

- 4. **IK 求解**: 调用 `robot.ikine_LM(Tep=T_target, q0=q_current, ...)`。
- 5. **下发**: `q_to_servo_targets` 转步数 -> 限位 -> 下发。
- **get\_key\_nonblock()**
  - **作用**: 键盘无阻塞输入。
  - **细节**: 使用 `select + tty` (Linux) 修改终端属性, 实现按键即读。
- **JoyConIKController 类**
  - **作用**: JoyCon 遥操作逻辑封装。
  - **细节**:
    - **状态读取**: `joycon.get_control()` 返回归一化的摇杆推量和 IMU 四元数。
    - **增量映射**: 摆杆映射为位置增量 `delta_pos`, IMU 相对姿态映射为 `delta_rpy`。
    - **平滑处理**: 引入死区 (Deadzone) 防止漂移, 使用低通滤波平滑输入。
- **\_ButtonHelper 类**
  - **作用**: 按键事件处理。
  - **细节**: 支持“按下瞬间 (rising edge)”和“长按连发 (repeat)”检测, 用于精细控制夹爪开合或调整速度档位。

## 2. 键位说明

### 1. arm\_keyboard\_control.py (关节步进)

- `q/a` : `shoulder_pan +/-`
- `w/s` : `shoulder_lift +/-`
- `e/d` : `elbow_flex +/-`
- `r/f` : `wrist_flex +/-`
- `t/g` : `wrist_roll +/-`
- `y/h` : `gripper +/-`
- `m` : 回中位
- `ESC` : 退出

### 2. ik\_keyboard\_realtime.py (末端 IK)

- `w/s` : `$+Z/-Z$`
- `A/D` : `$-Y/+Y$`
- `I/K` : `$+X/-X$`
- `J/L` : `pitch +/-`
- `U/O` : `yaw +/-`
- `+/ -` : 速度调节
- `Q` : 退出

### 3. joycon\_ik\_control\_py.py (JoyCon + IK)

- 移动 Joy-Con: 控制末端位置/姿态偏移 (在“基准位姿”上叠加)
- `ZR` : 夹爪收紧一点
- `R` : 夹爪松开一点
- `B` : `Z` 方向手动上移 (增大 `Z` 偏移)

- Home : 机械臂回中 + 重新连接 JoyCon
- +/- : 速度调节
- x : 退出

## 3. 运行与录屏 (

### 1. 键盘关节步进

- python arm\_keyboard\_control.py

### 2. 键盘 IK 控制

- python ik\_keyboard\_realtime.py

### 3. JoyCon IK 控制

- 右臂示例:
  - python joycon\_ik\_control\_py.py --device right --port /dev/right\_arm -- config ./driver/right\_arm.json
- 左臂示例:
  - python joycon\_ik\_control\_py.py --device left --port /dev/left\_arm -- config ./driver/left\_arm.json

录屏建议镜头：

1) 机械臂与操作者同框； 2) 先回中位，再做 2~3 次小幅末端移动； 3) 展示夹爪开合； 4) 最后按 x 或 ESC/Q 正常退出。

## 4. 控制误差分析

### 4.1 建议定义（最小集合）

- **末端重复定位误差（位置）**：对同一目标位姿 \$T^\*\$ 重复到达 \$N\$ 次，记录末端位置 \$p\_k\$，统计

$$e_p = \sqrt{\frac{1}{N} \sum_{k=1}^N \|p_k - \bar{p}\|^2}$$

- **末端重复定位误差（姿态）**：对同一目标位姿重复到达，记录欧拉角/旋转向量差的均方根（单位 deg）
- **IK 成功率**：控制循环中 success/total （失败时记录 reason）
- **限位截断次数**： limit\_position() 触发次数（提示你的目标是否经常越界）

### 4.2 记录方式（推荐你写进实验日志）

- 记录控制周期（tick 频率）、每 tick 的目标增量（XYZ 与 RPY）、speed 档位
- 同一动作做 3 组重复实验：低速/中速/高速

- 每组实验至少 30 秒，保证统计稳定

### 4.3 常见结论（写报告时怎么严谨表述）

- 若误差随速度显著增大：更可能是舵机内环/负载/通信抖动引入的动态误差，而非纯运动学模型误差
- 若同一方向总偏：优先怀疑 homing\_offset、关节方向符号（gear\_sign）或 DH/ETS 模型与实机不一致
- 若 IK 偶发失败：优先降低每 tick 位姿增量，或调整 mask（先只控 xyz，再逐步加入姿态）

## 5. 关键实现链路（从输入到舵机下发）

以 joycon\_ik\_control\_py.py 为例，控制链路是：

1) JoyCon 读取姿态/位移偏移： JoyconRobotics.get\_control() 2) 与“基准位姿”叠加生成目标：

- pos = base\_pos + joycon\_offset\_pos
- rpy = base\_rpy + joycon\_offset\_rpy （该脚本对 roll/pitch 做了符号对齐）

3) 目标位姿矩阵： build\_target\_pose() 4) IK 求解： robot.ikine\_LM(Tep=T\_goal, q0=current\_q, mask=..., method=...) 5) 角度→步数： robot.q\_to\_servo\_targets()

6) 限位： ServoController.limit\_position() 7) 同步

写： ServoController.fast\_move\_to\_pose() → FTServo.sync\_write(0x2A, ...)

其中 IK 的误差定义（angle-axis）与 LM 更新公式见： stage1\_1\_sdk.md。

### 4.1 误差来源

- 机械与舵机：背隙、摩擦、负载变化、舵机内环参数差异。
- 零位/限位标定： homing\_offset 不准导致 FK/IK 的“零点”错位。
- IK 数值误差：初值偏离、步长过大、 mask 设置不合理导致收敛到局部最优或失败。
- 传感器与输入：JoyCon 姿态漂移、用户手抖、滤波参数不匹配。
- 控制与通信：串口抖动、控制周期不稳定、速度/加速度过高引发超调。

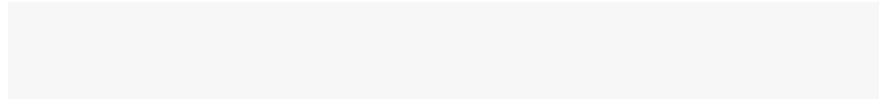
### 4.2 观测指标与建议记录方式

- 末端重复定位误差：同一目标位姿多次到达后的 XYZ 偏差（mm）与姿态偏差（deg）。
- IK 成功率：连续控制 ticks 中 success/total。
- 极限触发次数： limit\_position() 截断的次数（建议在实验日志中记录）。

### 4.3 常用排查/改进手段

- 降低每 tick 位姿增量、降低 speed 上限。
- 先用 arm\_keyboard\_control.py 验证每个关节方向与限位是否正确。
- 确保运行时使用正确的 driver/\*\_arm.json。
- JoyCon 漂移时用 Home 重置基准位姿并重新校准。

## 1. API 速查



## 2. 控制映射速查 (键盘/JoyCon)

### 程序设计结构

映射关系在程序里通常体现为：输入事件 → \$\Delta\$ (步数或位姿增量) → 目标构造 (IK 时) → 下发。

### 脚本作用

- `arm_keyboard_control.py` : 按键 → 关节步数增量 (不经 IK)。
- `ik_keyboard_realtime.py` : 按键 → 末端位姿增量 → IK → 下发。
- `joycon_ik_control_py.py` : JoyCon 摆杆/IMU → 末端位姿增量 → IK → 下发。

### 方法作用

#### 键盘关节级控制 ( `ArmKeyboardController` )

- `on_press(key)`
  - 作用：键盘事件回调。
  - 细节：捕获按键，查表找到对应的关节和方向，调用 `update_joint`。
- `update_joint(joint_name, delta)`
  - 作用：更新单关节目标。
  - 细节：读取当前值，加上 `delta` (步数增量)，做软限位截断，立即下发指令。
- `reset_to_home()`
  - 作用：一键回中。
  - 细节：调用 `controller.soft_move_to_home()`，以较慢速度平滑回到预设零位。

#### 键盘 IK 控制 ( `ik_keyboard_realtime.py` )

- `get_key_nonblock()`
  - 作用：非阻塞读取键盘。
  - 细节：使用 `termios` 和 `sys.stdin` 修改终端属性，实现按键即读，无需回车，保证控制实时性。
- `build_target_pose(current_pose, keys)`
  - 作用：构造目标位姿。
  - 细节：根据按下的键 (如 W/S/A/D)，在当前位姿 (`x, y, z, r, p, y`) 上叠加增量 `delta`，返回新的目标齐次变换矩阵 `T^*`。
- `robot.ikine_LM(Tep, q0, mask)`
  - 作用：求解逆运动学。
  - 细节：以当前关节角 `q0` 为初值，求解目标 `T^*` 对应的关节角。`mask` 通常设为 `[1, 1, 1, 0, 0, 0]` (只控位置) 或 `[1, 1, 1, 1, 1, 0]` (控位置 +Pitch/Roll)，忽略 Yaw 以避免 5-DoF 奇异。

## JoyCon IK 控制 ( JoyConIKController )

- `JoyconRobotics.get_control()`
  - 作用：获取手柄状态。
  - 细节：返回摇杆推量（归一化 -1~1）和 IMU 姿态（四元数/欧拉角），经过了死区处理和低通滤波。
- `JoyConIKController.run()`
  - 作用：主控制循环。
  - 细节：以固定频率（如 30Hz）运行，完成“读手柄 -> 算增量 -> 更新目标 -> IK 求解 -> 下发执行”的全过程。
- `_ButtonHelper.check()`
  - 作用：按键状态机。
  - 细节：区分“按下瞬间”（Rising Edge）和“按住不放”（Holding），用于分别处理触发式动作（如切换模式）和连续动作（如夹爪开合）。

## 1. 键盘关节步进

( arm\_keyboard\_control.py )

- q/a : shoulder\_pan +/-
- w/s : shoulder\_lift +/-
- e/d : elbow\_flex +/-
- r/f : wrist\_flex +/-
- t/g : wrist\_roll +/-
- y/h : gripper +/-
- m : 回中位
- ESC : 退出

适用场景：验证通信、限位、关节方向。

## 2. 键盘末端 IK

( ik\_keyboard\_realtime.py )

- 平移：I/K (X) 、 A/D (Y) 、 W/S (Z)
- 姿态：J/L (pitch) 、 U/O (yaw)
- +/- : 速度调节
- Q : 退出

适用场景：验证 FK/IK 是否稳定、mask/初值是否合理。

## 3. JoyCon 末端 IK

( joycon\_ik\_control\_py.py )

- 姿态/位移：移动 Joy-Con（在“基准位姿”上叠加偏移）
- ZR : 夹爪收紧一点
- R : 夹爪松开一点
- B : Z 手动上移（增加 Z 偏移）

## 1. API 速查

- `Home` : 机械臂回中 + JoyCon 重连/校准
- `+/-` : 速度调节
- `x` : 退出

安全建议：所有动作都从小幅度开始，先把 `speed` 调低，确认方向无误再加速。

### 3. 基于人机交互的机械臂数据集构建 (RDT)

快速跳转：

- 相机与手眼标定：`stage1_3_calibration.md`
- 时序同步说明：`stage1_3_timesync.md`
- 理论：手眼标定：`theory_handeye.md`
- 理论：RDT 数据格式：`theory_rdt_format.md`
- 函数解析：RDT 采集链路：`ref_rdt_pipeline.md`
- 函数解析：手眼评估：`ref_vision_handeye.md`

对应 `document/task.txt` 的 3. 产出物（本页聚焦“采集与格式”）：

- 数据集：`RDT/collect_rdt_dataset_teleop.py` 采集输出（raw/HDF5）
- 采集演示视频：按本文“采集流程”录制
- 数据格式说明：本页 + `theory_rdt_format.md`

标定与同步属于 3. 的必要前置条件，但其详细流程独立成文：

- 相机/手眼标定：`stage1_3_calibration.md`
- 时序同步说明：`stage1_3_timesync.md`

#### 1. 理论背景

RDT fine-tuning 期望输入是固定维度的向量与图像序列，因此本项目把所有 proprio/action 统一映射到 `float32[128]`，并用 `uint8[128] mask` 标注“哪些维度真实有效”。

同时，为了训练时的未来动作监督，数据中还会构造 `action_chunk`（默认 `$T_a=64$`），即从当前时刻起未来若干步的动作序列（越界部分做 padding，并把 mask 置 0）。

理论与张量形状的完整说明见：`theory_rdt_format.md`。

#### 2. 本章目标

从“可控的遥操作”出发，采集一个或多个 episode，并把 raw 数据稳定转换为符合 RDT fine-tuning 约束的 HDF5。

#### 3. 数据格式

- raw：落盘目录存在，图片与 CSV 数量对得上（肉眼可检查）
- HDF5：能被 `RDT/inspect_rdt_hdf5.py` 正确读取，张量 shape/字段合理，并能抽样可视化

### 程序设计结构

本章数据管线可抽象为“采集 (raw) → 结构化 (HDF5) → 校验 (inspect) ”，并通过 `unified 128 + mask + action_chunk` 保证张量维度一致且可解释：

- 1) **采集层**: 以固定频率采样机器人状态 (`proprio`) 与相机图像 (`images`)，并记录动作 (`actions`) 与时间戳 2) **统一表示层**: 把 `proprio`/`action` 映射到 `float32[128]`，并用 `uint8[128]` 标记有效维度 3) **序列监督层**: 构造 `action_chunk` ( $T_a=64$ ) 作为未来动作监督信号 (越界 padding + `mask=0`) 4) **落盘层**: raw 便于人工检查，HDF5 便于训练与批量读取

对应实现索引：格式与张量形状见 `theory_rdt_format.md`，代码链路解析见 `ref_rdt_pipeline.md`。

## 脚本作用

本模块的核心目标是采集符合 RDT Fine-tuning 格式要求的多模态数据。

- **采集主程序**: `RDT/collect_rdt_dataset_teleop.py`
  - **功能**: 同时连接双臂 (串口) 和三路相机 (USB)，以 30Hz (默认) 的频率同步记录数据。
  - **输出**: Raw 格式 (CSV + JPG 图片文件夹)，便于人工检查和筛选。
- **格式转换工具**: `RDT/build_rdt_hdf5_from_raw.py`
  - **功能**: 将 Raw 数据打包为高效的 HDF5 格式，并进行图像预处理 (Resize、Padding)。
- **数据检查工具**: `RDT/inspect_rdt_hdf5.py`
  - **功能**: 读取生成的 HDF5 文件，打印张量形状，并可视化抽样帧，确保数据无误。
- **格式定义库**: `RDT/rdt_hdf5.py`
  - **功能**: 定义了 `UnifiedVector` (统一向量) 和 `RDTHDF5EpisodeWriter` 类，封装了 HDF5 的写入逻辑，确保符合 `unified 128 + action_chunk` 规范。

采集脚本的默认录制按键与数据结构说明，见：`RDT/README.md`。

## 方法作用

围绕“`unified 128 + mask + action_chunk`”的最关键方法主要在 `RDT/rdt_hdf5.py` 与 `raw→HDF5` 脚本里：

### 统一向量构造 ( `RDT/rdt_hdf5.py` )

- **`UnifiedVector` 类**
  - **作用**: 数据容器。
  - **细节**: 维护一个 `float32[128]` 的值向量和一个 `uint8[128]` 的掩码向量。
- **`fill_slice(start_idx, end_idx, data)`**
  - **作用**: 填充数据片段。
  - **细节**: 将特定模态 (如关节角、末端位姿) 的数据填入 `UnifiedVector` 的指定切片区间，并自动将对应的 `mask` 设为 1。
- **`make_unified_vector(q, dq, ...)`**

- **作用：**工厂方法。
- **细节：**接收所有可能的输入（关节角、速度、末端位姿、夹爪状态等），按预定义布局组装成完整的 `UnifiedVector` 对象。

## HDF5 写入与 Chunking ( `RDT HDF5 EpisodeWriter` )

- `add_step(proprio, action, images, ...)`
  - **作用：**写入单步数据。
  - **细节：**将当前时刻的观测和动作暂存到内存 buffer 中，等待 episode 结束时统一处理。
- `finalize_action_chunks(chunk_size=64)`
  - **作用：**构造动作块（Action Chunking）。
  - **细节：**这是 RDT 训练的关键。对于时刻 \$t\$，它从 buffer 中提取 \$t\$ 到 \$t+chunk\\_size-1\$ 的动作序列。如果接近 episode 结尾不足 64 步，则用 0 填充（Padding）并将对应的 mask 置 0。
- `write_to_disk(path)`
  - **作用：**落盘 HDF5。
  - **细节：**创建 HDF5 文件，定义数据集（Datasets）和属性（Attributes），将处理好的 chunked actions、images（经过压缩或 resize）和 proprio 写入文件。

## 图像处理与检查

- `pad_to_square_and_resize_rgb(image, target_size)`
  - **作用：**图像预处理。
  - **细节：**为了保持长宽比，先对短边进行 Padding 使图像变方，然后 Resize 到目标尺寸（如 224x224 或 300x300），防止图像拉伸变形影响模型训练。
- `inspect_rdt_hdf5.py` 的可视化逻辑
  - **作用：**数据验收。
  - **细节：**随机抽取 HDF5 中的一个 step，解码并显示图像，打印 proprio 和 action 的数值及 mask，用于人工确认数据是否对齐、mask 是否正确。

## 2. 前置条件

1) 1./2. 已跑通 2) 设备就绪：

- 机械臂串口：`/dev/left_arm`、`/dev/right_arm`（或 `/dev/ttyACM0` 等）
- 三路相机：`/dev/video*`

3) 配置正确：

- 左臂：`driver/left_arm.json`
- 右臂：`driver/right_arm.json`

4) 标定完成（推荐）：

- 相机内参：由 `vision/calibrate_camera.py` 生成
- 手眼：按  
`vision/handeye_calibration_eyeinhand` 和 `vision/handeye_calibration_eyetohan` 生成 `handeye_result.yaml`

### 3. 采集流程

#### 3.1 采集 raw (CSV + JPG, 便于肉眼检查)

步骤说明：

1. **硬件连接**: 确保左右臂串口 (`/dev/left_arm`, `/dev/right_arm`) 和三路相机 (`/dev/video*`) 已连接。
2. **启动脚本**: 运行以下命令启动采集程序。

示例（双臂 + 三相机）：

```
python RDT/collect_rdt_dataset_teleop.py \
--device right \
--control-arm right \
--save-format raw \
--right-port /dev/right_arm \
--left-port /dev/left_arm \
--right-config ./driver/right_arm.json \
--left-config ./driver/left_arm.json \
--cam-exterior /dev/video0 \
--cam-right-wrist /dev/video2 \
--cam-left-wrist /dev/video4 \
--out-dir ./rdt_raw \
--instruction "pick up the red block"
```

参数详解：

- `--control-arm` : 指定主控臂（遥操作手柄控制的臂）。
- `--cam-exterior` : 第三人称视角相机（Eye-to-Hand）。
- `--cam-*-wrist` : 手腕相机（Eye-in-Hand）。
- `--instruction` : 当前任务的自然语言描述（将写入 HDF5 元数据）。
- `--save-format raw` : 保存为“CSV + JPG”的 raw episode（便于人工检查）。
- `--out-dir` : 输出根目录；raw 模式下会在其下创建 `episode_000001/`、`episode_000002/` ...

可选：如果你希望“直接生成 HDF5（跳过 raw→HDF5）”，把 `--save-format` 改为 `hdf5` 即可。

运行交互：

- 程序启动后会显示相机画面预览。
- 按 `Space` 键开始/暂停录制。
- 按 `Esc` 键结束当前 Episode 并保存。
- 默认输出路径（raw 模式）：`--out-dir/episode_000001/`（按已有 episode 自动续号）。

**raw 验收点（建议录屏/截图）：**

- `episode_*` 目录内存在图像文件夹与 CSV（或等价日志文件），且数量随录制时长增长
- 能肉眼回看图像序列，确认相机视角正确、曝光正常

#### 3.2 raw→HDF5 合成

采集完成后，需要将散乱的图片和 CSV 转换为紧凑的 HDF5 文件。

```
# 转换单个 episode
python RDT/build_rdt_hdf5_from_raw.py ./rdt_raw/episode_000001
```

#### 处理逻辑：

- 读取 CSV 中的 `proprio` 和 `action` 数据。
- 读取 `images/` 目录下的 JPG 图片。
- 调用 `pad_to_square_and_resize_rgb` 将图片统一调整为 384x384 分辨率。
- 生成 `action_chunk`（未来动作块）。
- 写入 HDF5 文件。

**转换验收点：**默认会在同级生成 `./rdt_raw/episode_000001.hdf5`，且文件大小随 episode 时长增长。

### 3.3 检查 HDF5 (shape/字段/可视化抽样)

```
python RDT/inspect_rdt_hdf5.py ./rdt_raw/episode_000001.hdf5
```

#### 检查项：

- **Shape**：确认 `images` 是 `(T, 2, 3, 384, 384, 3)`，`action` 是 `(T, 128)`。
- **Mask**：确认 `proprio_mask` 和 `action_mask` 非全零。
- **Visual**：脚本会弹窗显示随机抽取的几帧图像和对应的动作值。

#### 检查验收点：

- `T` 与录制长度一致或近似（允许因丢帧/起止边界有少量差异）
- `mask` 不应全 0；若全 0，通常是字段未填充或 slice 配置错误（优先看 `ref_rdt_pipeline.md`）

## 4. 数据格式（对齐 RDT fine-tuning，简述 + 索引）

为了适配 RDT 模型的输入要求，我们采用了特定的数据结构。

核心约束：

- **统一向量 (Unified Vector)**：所有的物理量（位置、速度、力矩等）都被映射到一个固定的 `float32[128]` 向量中。
- **Mask 机制**：配合 `uint8[128] mask`，标记哪些维度是有效的。例如，如果只控制了 6 个关节，则只有对应的 6 个位置维度 mask 为 1，其余为 0。
- **多帧输入**：`Timg=2`，即模型输入包含当前帧和上一帧。
- **Action Chunking**：采用 `action_chunk` 机制，预测未来 `Ta=64` 步的动作序列。

关键数据集（HDF5 内）：

- `observations/images` : `(T, 2, 3, 384, 384, 3) uint8`。维度含义：(时间, 历史帧数, 相机数, 高, 宽, 通道)。

## 1. API 速查

- `observations/proprio` / `observations/proprio_mask` : `(T, 128)`。机器人本体状态。
- `actions/action` / `actions/action_mask` : `(T, 128)`。当前时刻的动作。
- `actions/action_chunk` / `actions/action_chunk_mask` : `(T, 64, 128)`。未来动作块。
- `timestamps_unix_s` : `(T, )`。Unix 时间戳，用于时序对齐分析。

更完整的格式推导与张量形状解释见：`theory_rdt_format.md`。

## 5. 时序同步与标定

- 时序同步：实现与限制见 `stage1_3_timesync.md`
- 相机/手眼标定：流程与验收见 `stage1_3_calibration.md`

## 6. 理论补充

### 6.1 unified 128 + mask

本项目对齐 RDT fine-tuning 的关键点之一是：每条 `proprio`/`action` 都不是“只有数值”，而是“数值 + 可用性 mask”。

- `value` : `float32[128]`，缺失维度填 0
- `mask` : `uint8[128]`，可用维度置 1

实现对应：`RDT/rdt_hdf5.py::UnifiedVector/make_unified_vector/fill_slice`。

### 6.2 action\_chunk ( $Ta=64$ ) 的构造

episode 结束时，把 per-step `action[t]` 堆叠为：

$$\text{action\_chunk}[t, \tau] = \text{action}[t + \tau], \tau \in [0, Ta)$$

越界部分做 0 padding，并把 mask 置 0。

实现对应：`RDT/rdt_hdf5.py::RDTHDF5EpisodeWriter.finalize_action_chunks()`。

与同步相关的实现细节与限制请直接引用：`stage1_3_timesync.md`（避免在多处重复导致不一致）。

# 理论推导：RDT fine-tuning 数据格式 (unified 128 + mask + action\_chunk)

本章解释 3. 数据集结构，并逐条对齐仓库实现：

- RDT/rdt\_hdf5.py
- RDT/collect\_rdt\_dataset\_teleop.py
- RDT/build\_rdt\_hdf5\_from\_raw.py

## 理论框架

RDT fine-tuning 需要“形状固定且语义明确”的多模态序列输入：

- 视觉：固定历史帧数与相机数（本项目 `Timg=2`、`Ncam=3`）
- 本体与动作：统一到 `float32[128]`，用 `uint8[128]` mask 标记有效维度
- 监督信号：用 `action_chunk` (`Ta=64`) 把未来动作打包成序列监督

## 程序设计结构

- 采集（实时）：`collect_rdt_dataset_teleop.py` 把每个 tick 写成一个 step
- 格式（规范）：`rdt_hdf5.py` 定义统一向量与 HDF5 episode writer
- 转换（离线）：`build_rdt_hdf5_from_raw.py` 把 raw 重建为同构 HDF5

## 脚本作用

- RDT/collect\_rdt\_dataset\_teleop.py：决定“每个 step 写哪些字段、何时写时间戳”。
- RDT/build\_rdt\_hdf5\_from\_raw.py：把散落的 JPG/CSV 对齐并写入标准字段。
- RDT/inspect\_rdt\_hdf5.py：用 shape/抽样帧验证数据是否满足 fine-tuning 约束。

## 方法作用

- UnifiedVector/make\_unified\_vector/fill\_slice：保证不同来源数据能落到统一 128 维，并用 mask 标记有效维度。
- RDTHDF5EpisodeWriter.append\_step：定义每步写入的数据边界（images/proprio/action/timestamps）。
- RDTHDF5EpisodeWriter.finalize\_action\_chunks：把 per-step action 变成未来动作监督 chunk。

## 1. 观测定义与张量形状

实现遵循 `RDT/rdt_hdf5.py` 模块头部描述：

- 观测：

$$o_t := (X_{t-T_{img}+1:t}, z_t, c)$$

其中：

- `$X$`: 图像序列 (本项目 `Timg=2`)
- `$z_t$`: proprio (统一 128 维)
- `$c$`: instruction / 条件 (保存在 `meta/instruction`)

HDF5 的关键张量：

- `observations/images` :  $(T, T_{img}=2, N_{cam}=3, H, W, 3)$
- `observations/proprio` :  $(T, 128)$
- `actions/action` :  $(T, 128)$

## 2. 为什么要 unified 128 + mask

真实机器人采集里，很多维度未必能“可靠获得”，例如，某些关节/夹爪的物理量可能缺失。

因此实现了：

- `UnifiedVector.value` : 数值 (缺失维度填 0)
- `UnifiedVector.mask` : 可用性 (0/1)

训练时模型可用 mask 忽略缺失维度，避免把“填 0”误当成真实观测。

对应实现：`RDT/rdt_hdf5.py::UnifiedVector/make_unified_vector/fill_slice`。

## 3. 图像预处理：pad-to-square + resize

`collect_rdt_dataset_teleop.py::pad_to_square_and_resize_rgb()` 做：

1) 按最大边补成正方形 (padding 颜色可配) 2) resize 到 `image_size` (默认 384) 3) BGR→RGB

这样可保证不同相机分辨率下，输入张量形状一致。

## 4. action\_chunk 的构造 (Ta=64)

RDT fine-tuning 需要把“未来一段动作序列”打包为 chunk。

在 `RDTHDF5EpisodeWriter.finalize_action_chunks()` 中：

- 已有 per-step 动作： `action_all[t]` (shape `(T, 128)`)
- 构造：

$$\text{action\_chunk}[t, \tau] = \text{action}[t + \tau]$$

其中  $t \in [0, Ta]$  且  $t + \tau < T$ 。

## 1. API 速查

越界部分做 0 填充，并把 mask 置 0：

- `action_chunk[t, chunk_len:] = 0`
- `action_chunk_mask[t, chunk_len:] = 0`

张量最终形状：

- `actions/action_chunk : $(T,Ta,128)$`

## 5. 从 raw (CSV+JPG) 到 HDF5

- raw 的目的：可直接打开 `images/*.jpg` 目检、用 CSV 对齐定位问题。
- 合成的目的：得到标准 HDF5 episode，用于训练/评估。

raw writer 实现：`collect_rdt_dataset_teleop.py::RawEpisodeWriter`。

# 理论推导：手眼标定（Eye-in-Hand / Eye-to-Hand）与一致性评估

本章对应 3. 的“手眼标定设计思路/实现思路”，并与仓库实现一致：

- `vision/handeye_calibration_eyeinhand.py`
- `vision/handeye_calibration_eyetohand.py`
- `vision/handeye_utils.py`

手眼标定要解决的是把“相机看到的几何”与“机器人末端的几何”对齐：通过多组运动与观测，求一个固定外参 \$X\$，使两条链路在同一坐标系下自洽。

- Eye-in-Hand：用  $AX=XB$  求  $X=T_{CG}$
- Eye-to-Hand：求  $X=T_{CB}$ ，使标定板相对末端  $T_{TG}$  在多次采样中保持一致

一致性评估则把“是否自洽”量化成平移（mm）与旋转（deg）误差。

## 程序设计结构

手眼标定在工程上分三步：

1) 采集：同步记录  $T_{GB}^{(i)}$  与  $T_{TC}^{(i)}$  2) 求解：用  $AX=XB$  (Eye-in-Hand) 或“标定板相对末端恒定” (Eye-to-Hand) 得到外参  $X$  3) 评估：把  $X$  代回数据集，输出平移/旋转一致性误差 (mm/deg)

## 脚本作用

- `vision/handeye_calibration_eyeinhand.py`：采集与求解  $T_{CG}$ 。
- `vision/handeye_calibration_eyetohand.py`：采集与求解  $T_{CB}$ 。
- `vision/handeye_utils.py`：一致性评估与报告输出。

## 方法作用

- Eye-in-Hand 评估：`evaluate_eye_in_hand_consistency(...)` （构造  $A, B$  并统计  $\Delta = (AX)(XB)^{-1}$ ）。
- Eye-to-Hand 评估：`evaluate_eye_to_hand_consistency(...)` （构造  $T_{TG}^{(i)}$  并两两比较）。
- 报告输出：`print_consistency_report(...)` （把误差统计转成“质量等级”文本）。

## 1. 坐标系与符号

- $B$ ：基座 (base)
- $G$ ：末端 (gripper/end-effector)
- $C$ ：相机 (camera)

## 1. API 速查

- $\$T$ : 标定板 (target)

变换记号:

- $\$T_{ab}$  表示“坐标系  $\$b$  到  $\$a$  的齐次变换”（实现里使用  $4 \times 4$  矩阵）。

## 2. Eye-in-Hand: $AX = XB$

场景：相机安装在末端。

未知量：

$$X = T_{CG}$$

采集第  $i$  次：

- 末端在基座下： $\$T_{GB}^{(i)}$
- 标定板在相机下 (PnP 得到)： $\$T_{TC}^{(i)}$

对相邻两次  $(i, i+1)$  构造：

$$A = T_{GB}^{(i+1)} (T_{GB}^{(i)})^{-1}$$

$$B = (T_{TC}^{(i+1)})^{-1} T_{TC}^{(i)}$$

则满足：

$$AX = XB$$

仓库一致性评估 (`vision/handeye_utils.py::evaluate_eye_in_hand_consistency`)  
做的是：

$$\Delta = (AX)(XB)^{-1}$$

并统计：

- 平移误差： $\|\Delta_{0:3,3}\|$  (换算 mm)
- 旋转误差： $\|\text{rotvec}(\Delta_{0:3,0:3})\|$  (换算 deg)

## 3. Eye-to-Hand: 标定板相对末端恒定

场景：相机固定在环境（不动），标定板固定在末端。

未知量：

$$X = T_{CB}$$

对每次采集  $i$ ，标定板相对末端应为常量：

$$T_{TG}^{(i)} = (T_{GB}^{(i)})^{-1} T_{CB} T_{TC}^{(i)}$$

若标定正确，则不同  $i$  的  $\$T_{TG}^{(i)}$  应非常接近。

## 1. API 速查

仓库一致性评估（`vision/handeye_utils.py::evaluate_eye_to_hand_consistency`）  
做的是两两比较：

$$\Delta_{ij} = T_{TG}^{(i)} (T_{TG}^{(j)})^{-1}$$

并统计平移/旋转误差的均值与最大值。

### 3. 相机与手眼标定

理论推导：[theory\\_handeye.md](#)

本章对应 3. 产出物中的“手眼标定、数据时序同步源代码 + 设计手册”的标定部分。

手眼标定的核心是把“相机观测到的标定板位姿”（来自 PnP）与“机械臂末端位姿”（来自 FK/编码器）关联起来，求解固定的坐标变换：

- Eye-in-Hand：相机装在末端，求  $T_{CG}$ （相机  $C$  到夹爪/末端  $G$ ）
- Eye-to-Hand：相机固定在环境，求  $T_{CB}$ （相机  $C$  到基座  $B$ ）

在采集多组姿态后，通过经典形式  $AX=XB$  求解  $X$ （详见：[theory\\_handeye.md](#)）。

本页的目标是把“理论变量”落到“脚本输入/输出”：你将明确每一步会生成哪些文件、应达到什么误差水平、失败时优先查哪里。

### 程序设计结构

标定链路可以抽象为“内参 → PnP → 手眼 ( $AX=XB$ ) → 评估 → 验证闭环”：

1) **相机内参**：用棋盘格角点拟合得到  $K$  与畸变；这是 PnP 与手眼的共同前提  
**PnP 位姿**：每次观测把棋盘格/目标在相机坐标系下的位姿估计出来（生成  $T_{TC}$  或等价量）  
3) **手眼求解**：结合机器人末端位姿  $T_{GB}$  与观测位姿，解  $AX=XB$  得到固定外参 (Eye-in-Hand 的  $T_{CG}$  或 Eye-to-Hand 的  $T_{CB}$ )  
4) **一致性评估**：用多组采样对求解结果做平移/旋转误差统计  
5) **闭环验证**：把视觉检测结果通过手眼外参换算到机器人坐标系，驱动机械臂做“看得见 → 够得着”的动作

### 脚本作用

- `vision/calibrate_camera.py`：相机内参标定（采集/计算/一键 all），输出 `camera_intrinsics.yaml` 与报告。
- `vision/handeye_calibration_eyeinhand.py`：Eye-in-Hand 数据采集与解算，输出 `handeye_result.yaml`（含  $T_{CG}$ ）。
- `vision/handeye_calibration_eyetohand.py`：Eye-to-Hand 数据采集与解算，输出 `handeye_result.yaml`（含  $T_{CB}$ ）。
- `vision/handeye_utils.py`：一致性评估与误差统计（用于把标定质量量化进报告）。
- `vision/track_blue_circle_eyetohand.py`：闭环验证脚本（用于证明外参“可用且有效”）。

### 方法作用

#### 相机内参标定 (`calibrate_camera.py`)

- `detect_checkerboard(image)`
  - 作用：角点检测。
  - 细节：调用 `cv2.findChessboardCorners` 寻找棋盘格内角点，若成功则进一步调用 `cv2.cornerSubPix` 进行亚像素级精细化。
- `calibrate_intrinsics(objpoints, imgpoints, image_size)`
  - 作用：计算内参。
  - 细节：调用 `cv2.calibrateCamera`，输入世界坐标系下的点（假设在 Z=0 平面）和图像坐标系下的点，输出内参矩阵 `$K$` 和畸变系数 `$D$`。
- `save_calibration_result(...)`
  - 作用：结果持久化。
  - 细节：将 `$K$` 和 `$D$` 保存为 YAML 文件，并生成包含重投影误差统计的文本报告。

## 手眼标定 ( `handeye_calibration_*.py` )

- `get_board_pose_in_camera(image, K, D)`
  - 作用：PnP 位姿估计。
  - 细节：检测棋盘格角点，利用已知的棋盘格尺寸和内参，调用 `cv2.solvePnP` 计算标定板相对于相机的位姿 `$T_{target}^{cam}$`。
- `record_sample(robot_pose, image)`
  - 作用：采集数据对。
  - 细节：同步记录当前的机械臂末端位姿 `$T_{end}^{base}$` 和相机图像（或计算出的 `$T_{target}^{cam}$`），存入列表用于后续解算。
- `cv2.calibrateHandEye(...)`
  - 作用：求解手眼方程 `$AX=XB$`。
  - 细节：输入一系列的 `$R_{gripper}^{base}, t_{gripper}^{base}$` 和 `$R_{target}^{cam}, t_{target}^{cam}$`，输出手眼变换矩阵（Eye-in-Hand 为 `$T_{cam}^{gripper}$`，Eye-to-Hand 为 `$T_{cam}^{base}$`）。

## 验证与工具 ( `handeye_utils.py` / `track_blue_circle.py` )

- `compute_consistency_error(handeye_result, samples)`
  - 作用：一致性评估。
  - 细节：利用求得的手眼矩阵，将所有观测统一转换到同一坐标系下（如基座），计算标定板位置的标准差，衡量标定的一致性精度。
- `transform_point(point_cam, T_handeye, robot_pose)`
  - 作用：坐标变换（闭环验证）。
  - 细节：将视觉识别到的目标点（相机系）变换为机械臂目标点（基座系），用于驱动机械臂去“够”目标。

# 1. 相机内参标定

**目的：**获取相机的内参矩阵（Intrinsics）和畸变系数（Distortion Coefficients），这是后续 PnP 解算和手眼标定的基础。

**脚本：** `vision/calibrate_camera.py`

说明：该脚本在采集模式下会自动创建 `session_YYYYmmdd_HHMMSS/` 子目录，  
并把图片与标定结果都保存到该 session 内。因此如果你选择“分两步”（先  
采集后标定），第二步的 `--image-folder` 必须指向具体的 session 目录；  
否则会因为找不到图片而失败。

## 1.1 采集标定图

使用棋盘格标定板，在不同角度和距离下拍摄多张图片。

```
# --capture: 进入采集模式
# --camid: 相机设备 ID (如 /dev/video2 对应 2)
# --image-folder: 图片保存路径
python vision/calibrate_camera.py --capture --camid 2 --image-folder ./vision/
```

**操作提示：**按 `s` 键保存当前帧，按 `q` 键退出。采集结束后，终端会打印本次  
`session` 目录路径。

## 1.2 执行标定计算

读取采集到的图片，检测角点并计算内参。

如果你刚完成 1.1 采集，先找到最新的 session 目录（示例）：

```
ls -dt ./vision/calib_images_right/session_* | head -1
```

然后把该目录作为 `--image-folder` 输入：

```
# --calibrate: 进入计算模式
python vision/calibrate_camera.py --calibrate --image-folder ./vision/calib_im
```

## 1.3 一键模式（采集+标定）

如果想在一个流程内完成：

```
python vision/calibrate_camera.py --all --camid 2 --image-folder ./vision/calib
```

**输出产物：**脚本会在 session 目录下生成（以实际打印为准）：

- `camera_intrinsics.yaml`：相机内参矩阵  $\mathbf{K}$  与畸变系数
- `camera_intrinsics_report.txt`：详细报告（含每张图的重投影误差）
- `extrinsics.yaml / extrinsics.npy`：每张标定图对应的  $\mathbf{T}_{\{\text{target}\}^{\{\text{cam}\}}}$   
(用于 PnP 精度诊断/复现)
- `undistorted_test.jpg`：去畸变效果对比用

**关键检查点：**

- 报告中的平均重投影误差建议 < 0.5 px (更小更好)
- `undistorted_test.jpg` 目视检查畸变是否明显减轻

## 2. 手眼标定 (棋盘格)

**目的:** 求解相机坐标系  $\{C\}$  与机械臂末端坐标系  $\{G\}$  (Eye-in-Hand) 或基座坐标系  $\{B\}$  (Eye-to-Hand) 之间的变换矩阵。

注意: 本仓库标定代码在 `vision/` 目录下。

### 2.1 Eye-in-Hand (相机在末端)

**场景:** 相机安装在机械臂末端, 随机械臂运动。求解  $T_{CG}$ 。

**脚本:** `vision/handeye_calibration_eyeinhand.py`

#### 1. 数据采集:

```
# --collect: 采集模式
# --video: 相机 ID
# --port: 机械臂串口
python vision/handeye_calibration_eyeinhand.py --collect --video 0 --port
```

流程:

- 移动机械臂到不同姿态 (保持标定板在视野内)。
- 脚本会自动记录: 当前机械臂末端位姿  $T_{GB}$  + 当前相机拍摄的标定板位姿  $T_{TC}$  (通过 PnP 解算)。
- 建议采集 10-15 组数据, 覆盖不同的旋转角度。

#### 2. 解算:

```
# --calibrate: 计算模式
python vision/handeye_calibration_eyeinhand.py --calibrate
```

输出: `handeye_result.yaml`, 包含  $T_{CG}$  矩阵。

### 2.2 Eye-to-Hand (相机固定在环境)

**场景:** 相机固定不动, 拍摄机械臂末端 (末端夹持标定板)。求解  $T_{CB}$ 。

**脚本:** `vision/handeye_calibration_eyetohand.py`

#### 1. 数据采集:

```
python vision/handeye_calibration_eyetohand.py --collect --video 0 --port
```

注意: 此时标定板是固定在机械臂末端的, 相机是不动的。

#### 2. 解算:

```
python vision/handeye_calibration_eyetohand.py --calibrate
```

### 2.3 一致性评估与误差分析

**工具:** `vision/handeye_utils.py`

在执行标定计算时，脚本会自动调用一致性评估函数：

- Eye-in-Hand: `evaluate_eye_in_hand_consistency()`
- Eye-to-Hand: `evaluate_eye_to_hand_consistency()`

**原理：**利用  $AX=XB$  关系，检查每两组数据计算出的  $X$  是否一致。

- 平移误差：计算  $\Delta t$  的欧氏距离（单位：mm）。
- 旋转误差：计算  $\Delta R$  的旋转向量模长（单位：deg）。

## 方法作用

本章的关键方法集中解决两件事：**把观测变成变换矩阵、以及评估这个矩阵是否可信。**

- PnP 位姿估计：把棋盘格角点与内参  $K$  结合，求每帧的  $T_{\{target\}^{\{cam\}}}$ （实现位于 `vision/` 脚本内部，输出会写到 `session/report` 里）。
- 手眼求解 ( $AX=XB$ )：在采集到多组  $(A_i, B_i)$  后求解固定外参  $X$ ，分别对应 Eye-in-Hand 的  $T_{\{CG\}}$  与 Eye-to-Hand 的  $T_{\{CB\}}$ 。
- 一致性评估：
  - Eye-in-Hand: `evaluate_eye_in_hand_consistency()`
  - Eye-to-Hand: `evaluate_eye_to_hand_consistency()` 输出平移 (mm) 与 旋转 (deg) 误差，作为“标定是否可用”的量化指标。

## 3. 标定结果验证（蓝色圆追踪闭环）

脚本: `vision/track_blue_circle_eyetohand.py`

- 作用：检测蓝色圆  $\rightarrow$  PnP 求  $T_{\{target\}^{\{cam\}}}$   $\rightarrow$  用手眼结果换算到基座系  $\rightarrow$  驱动机械臂靠近目标。
- 运行：

```
python vision/track_blue_circle_eyetohand.py
```

## 函数解析：标定与一致性评估 ( vision/ )

本章对应 3. 的“手眼标定源代码 + 说明文档”，重点解析可复用的核心函数：

- `vision/handeye_utils.py`

一致性评估的理论目标是：在手眼标定得到  $T_{CG}$

$T_{CB}$  后，用多组采样构造误差指标 (mm/deg)，回答“这个外参是否可信、是否可用于闭环验证/数据复现”。

## 程序设计结构

- 标定脚本 (`vision/handeye_calibration_*.py`)：负责采集  $T_{GB}^{(i)}$  与  $T_{TC}^{(i)}$  并求解外参
- 评估函数 (`vision/handeye_utils.py`)：负责把外参代入采样对/采样集，输出误差统计与报告

## 脚本作用

- `vision/handeye_calibration_eyeinhand.py` / `vision/handeye_calibration_eyetohand.py`：生成 `handeye_result.yaml`。
- `vision/handeye_utils.py`：在标定解算阶段或单独调用时输出一致性报告。
- `vision/track_blue_circle_eyetohand.py`：使用标定结果做闭环验证（把“评估好”变成“能用”）。

## 方法作用

下文逐个函数说明输入/输出与误差计算方式，便于你在报告中引用“误差如何定义”。

### 1.

```
evaluate_eye_in_hand_consistency(T_cam_gr
ipper, T_gripper_base_list,
T_target_cam_list)
```

- 场景：Eye-in-Hand
- 输入：
  - `T_cam_grripper` :  $T_{CG}$  (标定结果)
  - `T_gripper_base_list` : 每次采集的  $T_{GB}^{(i)}$
  - `T_target_cam_list` : 每次采集的  $T_{TC}^{(i)}$  (PnP 得到)
- 核心计算 (相邻对) :
  - $A = T_{GB}^{(i+1)}(T_{GB}^{(i)})^{-1}$
  - $B = (T_{TC}^{(i+1)})^{-1}T_{TC}^{(i)}$

- $\$\\Delta=(AX)(XB)^{-1}$
- 输出：
  - `trans_errors` (mm) / `rot_errors` (deg) 及均值/最大值

## 2.

```
evaluate_eye_to_hand_consistency(T_cam_base, T_gripper_base_list, T_target_cam_list)
```

- 场景：Eye-to-Hand
- 输入：
  - `T_cam_base` :  $T_{CB}$  (标定结果)
  - 列表同上
- 核心计算：
  - 每次求  $T_{TG}^i = (T_{GB}^i)^{-1} T_{CB} T_{TC}^i$
  - 两两比较  $\$\\Delta_{ij}=T_{TG}^i (T_{TG}^j)^{-1}$
- 输出：
  - 同上 (mm/deg)

```
3. print_consistency_report(result, title='一致性误差')
```

- 作用：把一致性评估结果打印为“平均/最大 + 质量等级”
- 质量阈值：实现里用经验阈值 (mm/deg) 分级

## 4. 与采集脚本的接口边界

- 标定脚本负责生成：
  - 内参：相机 `K/dist`
  - 手眼结果： $T_{CG}$  或  $T_{CB}$
- 数据集采集脚本只负责记录控制与图像，不会自动推导外参；外参应作为实验配置写入报告。

### 3. 数据时序同步说明

本章描述 3. 数据采集链路中“如何对齐同一控制周期的数据”，以及当前实现的限制与可改进方向。

这里的“同步”并不等价于硬件同触发，而是：在离散控制周期（control tick）内，把多模态观测尽可能对齐到同一时间轴，保证下游能解释“这张图/这个 proprio/这个 action 属于同一次决策”。

因此本章关注的是：

- 以 tick 为单位的对齐策略
- 写入 HDF5 的时间戳语义
- 现有软件采集带来的偏差与需要在报告里说明的限制

### 程序设计结构

当前实现采用**单线程单循环**驱动，把每个 tick 的采样顺序固定下来，并把“近似同时刻”的数据写入同一个 step：

1) 读输入（遥操作） 2) 读机器人状态（proprio） 3) 取相机图像（按相机顺序）并堆叠最近 `Timg=2` 帧 4) 写入 `action/proprio/images` 与 `timestamps_unix_s`

这种结构的优点是实现简单、复现路径清晰；缺点是跨相机与跨模态必然存在毫秒级偏差。

### 脚本作用

- `RDT/collect_rdt_dataset_teleop.py`：采集主循环与 tick 对齐逻辑（决定“同步语义”）。
- `RDT/build_rdt_hdf5_from_raw.py`：把 raw 序列整理成 HDF5。
- `RDT/rdt_hdf5.py`：HDF5 字段定义（包含 `timestamps_unix_s` 等）。
- `RDT/inspect_rdt_hdf5.py`：通过 `shape`/抽样可视化检查对齐效果是否符合预期。

### 方法作用（对齐在代码里落到哪些“写入点”）

#### 采集主循环（`collect_rdt_dataset_teleop.py`）

- `main_loop()` 中的顺序执行逻辑
  - 作用：定义软同步策略。
  - 细节：在一个 `while` 循环（Tick）内，严格按照“读手柄 -> 读机械臂 -> 读相机 -> 存数据”的顺序执行。虽然各设备没有硬件触发信号连接，但这种顺序执行保证了在代码逻辑上它们属于同一个“Step”。
- `time.time()` 调用
  - 作用：记录时间戳。

- **细节：**在读取完所有传感器数据后，调用一次 `time.time()` 获取当前 Unix 时间戳，并将其作为该 Step 的 `timestamp` 写入 CSV/HDF5。这为后续分析对齐误差提供了粗粒度的基准。

## 数据处理与 HDF5 (`rdt_hdf5.py` / `build_rdt_hdf5_from_raw.py`)

- **图像堆叠 (Image Stacking)**
  - **作用：**构造时序输入。
  - **细节：**在写入 HDF5 时，对于时刻 `$t$`，不仅写入当前帧 `$I_t$`，还可能根据配置 (`T_img`) 写入历史帧 `$I_{t-1}$`。若 `$t=0$`，则复制 `$I_0$` 进行 Padding。这在 `RDTHDF5EpisodeWriter` 中处理。
- **`timestamps_unix_s` 字段**
  - **作用：**数据对齐的验证依据。
  - **细节：**在 HDF5 中作为一个独立 Dataset 存储。在多机采集或多模态融合场景下，可以通过比较不同流的 `timestamp` 来评估同步偏差。

## 1. 同步目标

在每个 control tick，尽量让以下数据能在同一时间轴上对齐：

- 三相机图像 (exterior/right-wrist/left-wrist) 最近 `Timg=2` 帧
- 双臂 proprio (关节/末端等，写入 unified 128 + mask)
- 当前动作 action (写入 unified 128 + mask)

## 2. 当前实现 (代码映射)

主循环： `RDT/collect_rdt_dataset_teleop.py`

- 单一循环驱动：每个 tick 顺序执行“读输入 → 读状态 → 取图像 → 写 step”。
- 时间戳：写入 `timestamps_unix_s` (Unix 秒)。
- 图像堆叠：每个 tick 保存最近 `Timg=2` 帧 (不足则 padding)。

## 3. 已知限制

- 多相机为软件顺序读取：跨相机存在毫秒级偏差（非硬触发）。
- 图像与关节状态并非同一时刻采样：受采集顺序与 USB/串口抖动影响。

## 4. 可改进方向

- 为每路相机单独记录帧级时间戳（每帧一个 `timestamp`），并在 HDF5 中额外保存。
- 多线程/异步采集相机，主线程按时间最近邻对齐。
- 使用硬件触发同步相机（最严格），或使用同一同步信号写入日志。

## 函数解析：RDT 数据采集与落盘 ( RDT/ )

本章对应 3. 的“数据集采集源代码 + 格式说明”，关键文件：

- RDT/collect\_rdt\_dataset\_teleop.py
- RDT/rdt\_hdf5.py
- RDT/build\_rdt\_hdf5\_from\_raw.py
- RDT/inspect\_rdt\_hdf5.py

RDT 采集链路的理论目标是：把多模态时间序列（图像、proprio、action、时间戳）组织成固定 **shape** 且可解释的数据集，使下游 fine-tuning 可以直接消费。核心约束是 `unified 128 + mask + action_chunk`（详见 `theory_rdt_format.md`）。

### 程序设计结构

- `collect_rdt_dataset_teleop.py`：实时采集与“写 step”（raw 或直接 HDF5）
- `rdt_hdf5.py`：格式规范层（UnifiedVector + HDF5 writer）
- `build_rdt_hdf5_from_raw.py`：离线重建（raw→HDF5）
- `inspect_rdt_hdf5.py`：验收与排错（shape/抽样可视化）

### 脚本作用

- 采集：用 `collect_rdt_dataset_teleop.py` 录一段 raw/hdf5 episode。
- 转换：用 `build_rdt_hdf5_from_raw.py` 把 raw 整理成标准 HDF5。
- 检查：用 `inspect_rdt_hdf5.py` 确认张量 shape、mask、抽样帧正常。

### 方法作用

下文按“格式定义 → 采集写入 → 转换与检查”的顺序解释关键方法在管线中的职责。

## 1. RDT/rdt\_hdf5.py - 数据格式定义与 HDF5 写入核心

**脚本功能：**该模块定义了 RDT 数据集的核心数据结构，实现了统一的 128 维向量表示（Unified Vector）和 HDF5 文件的写入逻辑。它是数据采集流程中的格式规范层，确保生成的数据符合 RDT Fine-tuning 的输入要求。

### 1.1 rotmat\_to\_rot6d(R) - 旋转矩阵到6D表示的转换

**功能：**将  $3 \times 3$  旋转矩阵压缩为 6 维连续表示，避免万向锁问题。

**实现原理:**

- 输入: \$3\times 3\$ 旋转矩阵 \$R\$
- 输出: 6D 向量 (取 \$R\$ 的前两列, 按列主序展平)
- 数学依据: 6D 表示法由 Zhou et al. (CVPR 2019) 提出, 通过 Gram-Schmidt 正交化可恢复完整旋转矩阵, 且在神经网络训练中比四元数更稳定。

**代码逻辑:**

```
def rotmat_to_rot6d(R):
    # 取前两列
    return R[:, :2].T.flatten() # shape: (6,)
```

## 1.2 UnifiedVector / make\_unified\_vector() / fill\_slice(vec, sl, data)

**功能:** 实现统一向量 (Unified Vector) 的构建与填充, 这是 RDT 处理异构机器人数据的关键机制。

**设计动机:** 不同机器人的自由度、传感器配置差异很大。Unified Vector 通过 128 维固定长度 + Mask 机制, 允许不同配置的机器人共享同一训练管道。

**核心数据结构:**

- UnifiedVector.value : float32[128] - 存储物理量的数值 (未使用的维度填 0)
- UnifiedVector.mask : uint8[128] - 标记哪些维度有效 (1=有效, 0=无效)

**方法详解:**

- make\_unified\_vector() : 创建一个空的 Unified Vector

```
def make_unified_vector():
    return UnifiedVector(
        value=np.zeros(128, dtype=np.float32),
        mask=np.zeros(128, dtype=np.uint8)
    )
```

- fill\_slice(vec, sl, data) : 向指定切片填充数据并自动更新 mask

## ◦ 参数:

- vec : 目标 UnifiedVector
- sl : 切片对象 (如 slice(0, 6) 表示前 6 维)
- data : 要填充的数据 (自动截断到切片长度)

## ◦ 实现:

```
def fill_slice(vec, sl, data):
    indices = range(*sl.indices(128))
    length = len(indices)
    vec.value[sl] = data[:length]
    vec.mask[sl] = 1 # 标记为有效
```

## 1.3 RDTHDF5EpisodeWriter - HDF5 文件写入器

**功能：**管理单个 Episode 的 HDF5 文件写入，自动处理张量形状、Action Chunk 生成等细节。

**初始化：**

```
writer = RDTHDF5EpisodeWriter(
    filepath="episode_000001.hdf5",
    Timg=2,           # 历史图像帧数
    Ncam=3,          # 相机数量
    image_size=384,  # 图像分辨率
    Ta=64            # Action Chunk 长度
)
```

**核心方法：**

- `append_step(...)`：追加一条时间步数据
  - 校验：自动检查 `images` 的 shape 是否为 `(Timg, Ncam, H, W, 3)`
  - 存储：将 proprio/action 的 value 和 mask 分别追加到对应 Dataset
- `finalize_action_chunks()`：生成未来动作序列
  - 原理：对于时刻 \$t\$，构造 `action_chunk[t, :, :] = [action[t], action[t+1], ..., action[t+Ta-1]]`
  - 边界处理：当 `t+Tau \geq T` 时，用零填充并将 mask 置 0
  - 自动调用：在 `close()` 时自动执行
- `close()`：关闭文件并保存元数据
  - 流程：`finalize action_chunks` → 写入 meta 信息 → 关闭 HDF5 文件句柄

## 2. RDT/collect\_rdt\_dataset\_teleop.py

### 2.1 RawEpisodeWriter

- 作用：写 raw episode (CSV + JPG)，便于肉眼检查
- `append_step(...)`：
  - 保存 `timg*ncam` 张图片到 `images/`
  - 追加 proprio/action/mask/timestamp 到 CSV

### 2.2 图像预处

**理：** `pad_to_square_and_resize_rgb(frame_bgr, out_size, pad_bgr)`

- pad 成正方形 → resize 到 `out_size` → 转 RGB

### 2.3 相机抽象： MultiViewCamera

- `_open(source, width, height)`：支持 index 或路径；支持 v4l2/gstreamer backend
- `read_bgr(view)`：读取指定 view 的一帧
- `close()`：释放所有 `VideoCapture`

## 2.4 速度估计: `compute_ang_vel_rad_s(R_prev, R_cur, dt)`

- 通过  $dR=R_{\text{prev}}^T R_{\text{cur}}$  的旋转向量除以  $dt$  估计角速度

## 2.5 episode 编号: `_max_existing_episode_idx(out_dir)`

- 扫描 `episode_xxxxxx/` 与 `episode_xxxxxx.hdf5`，返回最大编号
- 用于避免覆盖、实现跨运行递增

## 2.6 采集主类: `JoyConRDTCollector`

- 初始化阶段：
  - 创建双臂 `ArmRig`
  - 可选打开三相机 `MultiViewCamera`
  - 配置 `preview` 与保存格式
- `run()` (主循环, 函数体较长)：
  - 读 JoyCon 控制
  - 生成目标位姿  $\rightarrow$  IK  $\rightarrow$  下发
  - 若录制：构建 `unified/proprio/action`  $\rightarrow$  `writer.append_step`

## 3. raw $\rightarrow$ HDF5 与检查

- `build_rdt_hdf5_from_raw.py` : 读取 raw 的 CSV/JPG，重建 HDF5 episode
- `inspect_rdt_hdf5.py` : 打印 meta/shape，并可抽样检查内容

理论解释 (unified/mask/action\_chunk) 见: `theory_rdt_format.md`。

## 5. FPGA 逆解加速

本章介绍如何利用 FPGA 硬件加速 Levenberg-Marquardt 逆运动学求解器，实现相比纯软件执行约 **10倍** 的单步迭代性能提升。

### 理论背景

#### 为什么需要硬件加速

逆运动学 (IK) 求解是机械臂控制的核心环节，决定了末端执行器从当前位姿运动到目标位姿时各关节应旋转的角度。在实时控制场景（如遥操作、轨迹跟踪）中，IK 求解需要在毫秒级完成，而纯软件实现的迭代式 LM 算法在资源受限的嵌入式平台（如 PYNQ-Z2 的 ARM 处理器）上往往无法满足性能需求。

FPGA 通过其高度并行的硬件架构，可以将 LM 算法中的矩阵运算（雅可比计算、Cholesky 分解、线性方程求解等）映射为流水线电路，显著降低单步迭代延迟。本项目在 PYNQ-Z2 板卡上实现的硬件加速 IP 核，将单步迭代速度提升约 **10 倍**，平均迭代时间减少 **7.9 倍**，总体加速比达到 **2.5 倍**。

#### Levenberg-Marquardt 算法回顾

LM 算法是一种迭代优化方法，通过最小化位姿误差来求解关节角。核心更新公式为：

$$(J^T J + \lambda I) \Delta q = -J^T e$$

其中：

- $\mathbf{q}$ : 关节角向量 (SO-101 为 5 自由度)
- $\mathbf{e}$ : 位姿误差 (3D 位置 + 3D 姿态, angle-axis 表示)
- $\mathbf{J}$ : 几何雅可比矩阵 (末端速度对关节角速度的偏导)
- $\lambda$ : 阻尼系数 (正则化参数, 防止奇异)

**单步迭代流程：**

1. 正运动学 (FK)：计算当前关节角  $\mathbf{q}$  对应的末端位姿  $\mathbf{T}(\mathbf{q})$
2. 误差计算： $\mathbf{e} = \text{pose\_error}(\mathbf{T}_{\text{goal}}, \mathbf{T}(\mathbf{q}))$
3. 雅可比计算： $\mathbf{J} = \frac{\partial \mathbf{v}}{\partial \dot{\mathbf{q}}}$  (6 行 5 列)
4. 构造正规方程： $\mathbf{A} = \mathbf{J}^T \mathbf{J} + \lambda \mathbf{I}$ ,  $\mathbf{b} = -\mathbf{J}^T \mathbf{e}$
5. Cholesky 求解：分解  $\mathbf{A} = \mathbf{L} \mathbf{L}^T$ , 求解  $\Delta \mathbf{q}$
6. 更新关节角： $\mathbf{q} \leftarrow \mathbf{q} + \Delta \mathbf{q}$

本项目的硬件加速核心在于 **步骤 3-5**，这三步涉及大量矩阵运算，占据了 LM 算法 80% 以上的计算时间。

#### 硬件加速的关键计算

在 LM 迭代中，最耗时的操作包括：

### 1. 雅可比矩阵计算 ( `compute_fk_jacobian_so101` )

- 输入:  $\sin(q)$ 、 $\cos(q)$  (预计算, 避免 FPGA 实现超越函数)
- 输出:  $J \in \mathbb{R}^{6 \times 5}$
- 复杂度: 涉及 30+ 次浮点乘法、加法 (旋转矩阵连乘与叉积)
- 优化: UNROLL factor=3, 部分展开平衡 DSP 使用与延迟

### 2. 矩阵乘法 (调用 Vitis Solver L1 库)

- $A = J^T J : (5 \times 6) \times (6 \times 5) \rightarrow (5 \times 5)$
- $b = J^T e : (5 \times 6) \times (6 \times 1) \rightarrow (5 \times 1)$
- 复杂度:  $O(N^3)$  或  $O(N^2 M)$  ( $N=5$ ,  $M=6$ )
- 优化: 使用 Xilinx L1 库的高度优化实现 (PIPELINE + DATAFLOW)

### 3. Cholesky 分解与三角求解 (调用 Vitis Solver L1 库)

- 分解:  $A = L L^T$  ( $L$  为下三角矩阵)
- 前向替换:  $L y = b$
- 后向替换:  $L^T \Delta q = y$
- 复杂度:  $O(N^3)$
- 优化: 完全展开 (UNROLL), 利用 DSP 流水线 (延迟最小化)

FPGA 可以针对这些操作进行流水线优化, 并利用 BRAM 和 DSP 资源实现高效的浮点运算。相比 ARM 处理器的顺序执行, FPGA 的并行计算能力在小矩阵密集运算场景下优势显著。

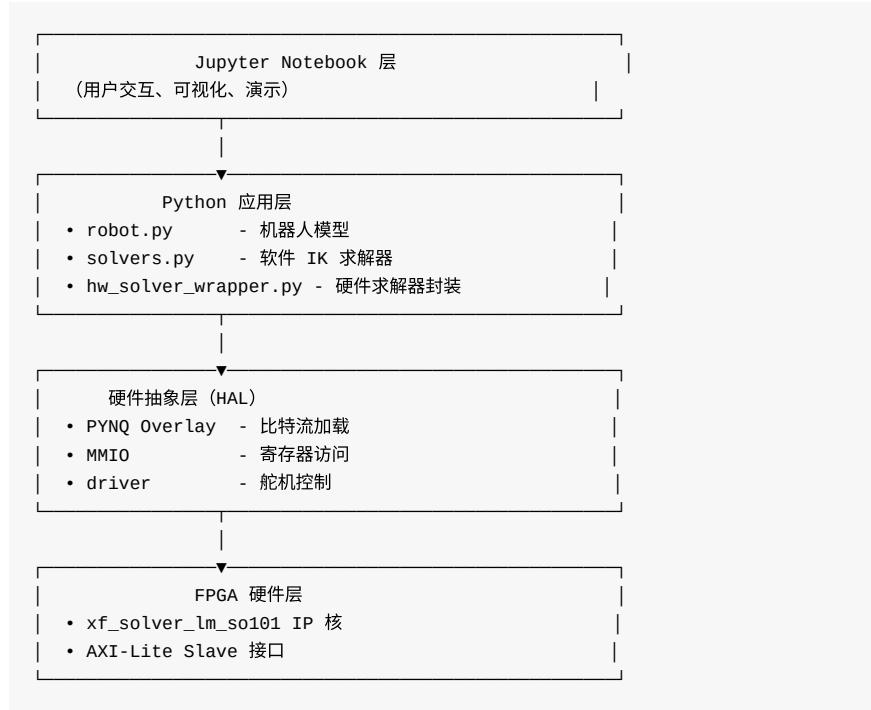
---

## 系统架构

### 硬件平台

- 开发板: PYNQ-Z2 (Xilinx Zynq-7020 SoC)
  - PS 端 (Processing System) : 双核 ARM Cortex-A9 @ 650 MHz
    - 运行 Python 代码 (PYNQ 框架、Jupyter Notebook)
    - 高层控制逻辑 (误差计算、收敛判断、阻尼调整)
    - 通过 AXI-Lite 接口访问 PL 端 IP 核
  - PL 端 (Programmable Logic) : Artix-7 FPGA
    - 部署 IK 求解器 IP 核 ( `xf_solver_lm_so101` )
    - 硬件资源: 53200 LUT、106400 FF、220 DSP、140 BRAM
- 通信接口: AXI-Lite
- 编程框架:
  - Vitis HLS 2024.2: 高层次综合 (C++ → RTL)
  - Vivado 2024.2: FPGA 综合与布局布线
  - PYNQ 2.7+: Python 到 FPGA 的桥接层 (Overlay 机制)

### 软硬件协同架构



## IP 核接口设计

硬件求解器通过 AXI-Lite 寄存器接口与 PS 端交互，主要寄存器包括：

### 输入寄存器

偏移地址	名称	类型	描述
0x10	sin0	float	关节 0 的正弦值
0x18	sin1	float	关节 1 的正弦值
0x20	sin2	float	关节 2 的正弦值
0x28	sin3	float	关节 3 的正弦值
0x30	sin4	float	关节 4 的正弦值
0x38	cos0	float	关节 0 的余弦值
0x40	cos1	float	关节 1 的余弦值
0x48	cos2	float	关节 2 的余弦值
0x50	cos3	float	关节 3 的余弦值
0x58	cos4	float	关节 4 的余弦值
0x60 - 0x88	e0 - e5	float	位姿误差向量 (6 维)
0x90	lambda	float	阻尼系数

### 输出寄存器

偏移地址	名称	类型	描述
0x98	d0	float	关节角增量\$\\Delta q_0\$
0xA8	d1	float	关节角增量\$\\Delta q_1\$
0xB8	d2	float	关节角增量\$\\Delta q_2\$
0xC8	d3	float	关节角增量\$\\Delta q_3\$
0xD8	d4	float	关节角增量\$\\Delta q_4\$
0xE8	status	uint	执行状态码

## 控制寄存器

偏移地址	名称	位域	描述
0x00	CTRL	[0]	启动计算（写 1 触发）
0x00	STAT	[1]	完成标志（读取时检查 bit 1）

## 程序设计结构

本项目采用软硬件协同设计，明确划分计算边界：误差计算在 PS，增量求解在 PL。

### 硬件层次 (FPGA PL 端)

#### 1. IP 核实现：`lm_solver_so101.hpp`

核心数据结构：

```
struct S0101_Params {
    // 5个关节的几何参数 (基于 ET 序列)
    static constexpr float j1_tx = 0.0612f; // Joint 1: Rz
    static constexpr float j1_tz = 0.0598f;
    static constexpr float j2_tx = 0.02943f; // Joint 2: Ry
    // ...
};
```

三大核心函数：

1. `compute_fk_jacobian_so101<T>(sin_q, cos_q, p_end, J)`
  - 输入：\$|sin(q\_i)|\$、\$|cos(q\_i)|\$ [5] (PS 端预计算)
  - 输出：末端位置 \$p\$ [3]，雅可比 \$J\$ [6×5]
  - 实现：串联 5 个 ET 变换 (Rz → Ry → Ry → Rx → Rx)
  - 优化：`#pragma HLS UNROLL factor=3` (部分展开叉积计算)
2. `lm_solve_step_so101<T>(sin_q, cos_q, error, lambda, delta)`
  - 输入：误差向量 \$e\$ [6]，阻尼系数 \$\\lambda\$

## 1. API 速查

- **输出:** 关节增量  $\Delta q$  [5]
- **流程:**
  - i. 调用 `compute_fk_jacobian_so101` 获取  $J$
  - ii. 矩阵乘法:  $A = J^T J$  (调用 Vitis L1 库)
  - iii. 矩阵乘法:  $b = -J^T e$
  - iv. 添加阻尼:  $A \leftarrow A + \lambda I$
  - v. Cholesky 分解:  $A = L L^T$  (调用 Vitis L1 库)
  - vi. 三角求解:  $L y = b$ ,  $L^T \Delta q = y$  (完全展开)

### 3. Cholesky 分解与三角求解优化

- **对角线倒数预算算:** 避免除法流水线停顿

```
T L_diag_inv[N];
for(int i = 0; i < N; i++) {
    L_diag_inv[i] = T(1.0) / L[i][i];
}
```

- **前向替换展开** (以  $N=5$  为例) :

```
y[0] = b[0] * L_diag_inv[0];
y[1] = (b[1] - L[1][0]*y[0]) * L_diag_inv[1];
y[2] = (b[2] - L[2][0]*y[0] - L[2][1]*y[1]) * L_diag_inv[2];
// ...
```

- **后向替换展开:** 从  $\Delta q_4$  到  $\Delta q_0$  逆序计算

## 2. Vivado 工程: `src/overlay/`

- **Block Design:** AXI 互连、时钟域转换、IP 核实例化
- **比特流输出:** `notebook/design_1.bit` + `design_1.hwh` (硬件描述)
- **资源占用** (综合后) :
  - LUT: 33743 / 53200 (63%)
  - DSP: 200 / 220 (90%)
  - BRAM: 8 / 140 (6%)

## 软件层次 (Python PS 端)

### 1. 硬件抽象层: `hw_solver_wrapper.py`

`HWsolverIterator` 类: 封装 AXI-Lite 寄存器访问

- **写入:** 通过 `solver_ip.mmio.write(offset, value)` 传递 sin/cos、误差、lambda
- **读取:** 通过 `solver_ip.mmio.read(offset)` 获取  $\Delta q$ 、状态码
- **同步:** 轮询控制寄存器的 Done 标志位 (bit 1)

`HWsolver` 类: 与 `robot.ikine_LM()` 兼容的高层接口

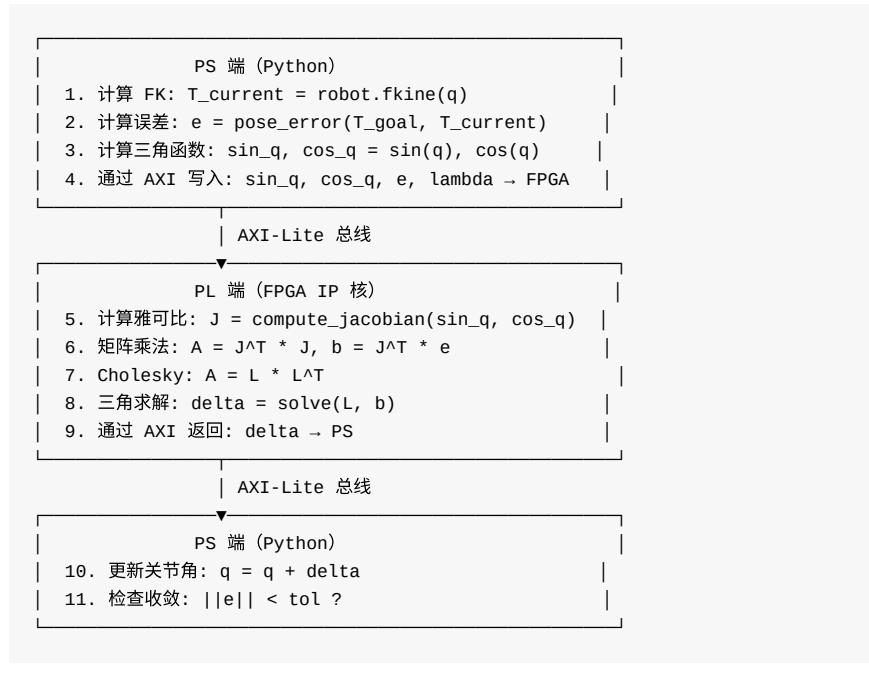
- **软件部分:** FK 误差计算 (调用 `robot.fkine(q)`)
- **硬件部分:** 增量求解 (调用 `hw_iterator.solve_step()`)
- **迭代控制:** 收敛判断、阻尼调整 (Wampler/Sugihara 方法)

### 2. 工厂函数: `create_hw_solver()`

```
from pynq import Overlay

overlay = Overlay('design_1.bit')
solver_ip = overlay.xf_solver_lm_so101_0
hw_solver = HWSolver(robot, HWSolverIterator(solver_ip, robot))
```

## 数据流图



## 方法作用

本节深入解析 HLS C++ 代码中的核心函数实现与优化策略。

### 1. `compute_fk_jacobian_so101()` - 正运动学与雅可比计算

函数原型:

```
template <typename T>
void compute_fk_jacobian_so101(
    const T sin_q[5],           // 输入: sin(q0), ..., sin(q4)
    const T cos_q[5],           // 输入: cos(q0), ..., cos(q4)
    T p_end[3],                // 输出: 末端位置 [x, y, z]
    T J[6][5]                  // 输出: 几何雅可比 [6行x5列]
);
```

**功能:** 基于 SO-101 的 ETS (Elementary Transform Sequence) 串联计算正运动学和几何雅可比矩阵。

**几何参数** (硬编码在 `so101_Params` 结构体) :

```

struct S0101_Params {
    // Joint 1 (Rz): Base → J1
    static constexpr float j1_tx = 0.0612f;
    static constexpr float j1_tz = 0.0598f;

    // Joint 2 (Ry): J1 → J2
    static constexpr float j2_tx = 0.02943f;
    static constexpr float j2_tz = 0.05504f;

    // Joint 3 (Ry): J2 → J3
    static constexpr float j3_tx = 0.02798f;
    static constexpr float j3_tz = 0.1127f;

    // Joint 4 (Ry): J3 → J4
    static constexpr float j4_tx = 0.15504f;
    static constexpr float j4_tz = 0.00519f;

    // Joint 5 (Rx): J4 → End
    static constexpr float j5_tx = 0.0593f;
    static constexpr float j5_tz = 0.00996f;
};


```

**核心算法：**

1. **初始化：** 旋转矩阵  $R = I_3$ ，位置  $p = [t_{x1}, 0, t_{z1}]^T$
2. **逐关节串联** (5 个关节)：
  - 更新旋转：  $R \leftarrow R \cdot \text{Rot}(\text{axis}, q_i)$  ( $\text{axis}$  为 X/Y/Z)
  - 更新位置：  $p \leftarrow p + R \cdot [t_x, 0, t_z]^T$
  - 保存关节信息：  $z_i = R[:, \text{axis}]$  (旋转轴)，  $p_i = p$  (关节位置)
3. **雅可比计算：**
  - 线速度列：  $J_v^i = z_i \times (p_{\text{end}} - p_i)$  (叉积)
  - 角速度列：  $J_\omega^i = z_i$

**HLS 优化：**

```

#pragma HLS INLINE off // 防止自动内联 (便于性能分析)
#pragma HLS UNROLL factor=3 // 部分展开雅可比列计算 (平衡DSP与延迟)
#pragma HLS ARRAY_PARTITION variable=R dim=2 complete // 雅可比列完全分区
#pragma HLS ARRAY_PARTITION variable=p dim=1 complete // 雅可比行完全分区

```

**数学推导** (以 Joint 2 为例) :

- 旋转轴为 Y 轴：  $z_1 = R[:, 1] = [R_{01}, R_{11}, R_{21}]^T$
- 位置差：  $\Delta p = p_{\text{end}} - p_1$
- 叉积：

$$J_v^1 = \begin{bmatrix} z_{1y} \Delta p_z - z_{1z} \Delta p_y \\ z_{1z} \Delta p_x - z_{1x} \Delta p_z \\ z_{1x} \Delta p_y - z_{1y} \Delta p_x \end{bmatrix}$$

**2. lm\_solve\_step\_so101() - LM 单步求解**

**函数原型:**

```
template <typename T>
int lm_solve_step_so101(
    const T sin_q[5],           // 输入: sin(q)
    const T cos_q[5],           // 输入: cos(q)
    const T error[6],           // 输入: 位姿误差 e
    T lambda,                  // 输入: 阻尼系数
    T delta[5]                 // 输出: 关节增量 Δq
);
```

**功能:** 完成 LM 算法的单步迭代，从误差到关节增量。

**计算流程:****1. 计算雅可比 (调用 `compute_fk_jacobian_so101`)**

```
T J[6][5];
#pragma HLS ARRAY_PARTITION variable=J dim=0 complete
compute_fk_jacobian_so101<T>(sin_q, cos_q, p_end, J);
```

**2. 矩阵乘法  $A = J^T J$  (调用 Vitis Solver L1 库)**

```
T A[5][5];
#pragma HLS ARRAY_PARTITION variable=A dim=0 complete
typedef matrixMultiplyTraits<Transpose, NoTranspose, 6, 5, 6, 5, T, T> JtJ;
matrixMultiplyTop<Transpose, NoTranspose, 6, 5, 6, 5, 5, 5, JtJ_Traits, T,
```

- **Transpose 模式:** 自动转置第一个矩阵
- **流水线深度:** 库内部已优化 ( $l=1$ , 吞吐率最大化)

**3. 矩阵乘法  $b = J^T e$** 

```
T b[5];
#pragma HLS ARRAY_PARTITION variable=b complete
typedef matrixMultiplyTraits<Transpose, NoTranspose, 6, 5, 6, 1, T, T> Jte;
matrixMultiplyTop<Transpose, NoTranspose, 6, 5, 6, 1, 5, 1, Jte_Traits, T,
```

**4. 添加阻尼  $A \leftarrow A + \lambda I$** 

```
for(int i = 0; i < 5; i++) {
    #pragma HLS UNROLL
    A[i][i] += lambda;
}
```

- **UNROLL:** 完全展开 (5 个加法并行执行)

**5. Cholesky 分解  $A = L L^T$  (调用 Vitis Solver L1 库)**

```
T L[5][5];
#pragma HLS ARRAY_PARTITION variable=L dim=0 complete
typedef choleskyTraits<true, 5, T, T> CholeskyConfig;
int chol_ret = choleskyTop<true, 5, CholeskyConfig, T, T>(A, L);
```

- **Lower Triangle 模式:** 只计算下三角 (节省 50% 运算)

## 1. API 速查

- 失败处理：若 \$A\$ 非正定，返回错误码 1

### 6. 三角求解（完全展开实现）

- 前向替换 \$L y = b\$：

```
T L_diag_inv[5];
for(int i = 0; i < 5; i++) {
    L_diag_inv[i] = T(1.0) / L[i][i]; // 预计算对角线倒数
}
y[0] = b[0] * L_diag_inv[0];
y[1] = (b[1] - L[1][0]*y[0]) * L_diag_inv[1];
y[2] = (b[2] - L[2][0]*y[0] - L[2][1]*y[1]) * L_diag_inv[2];
y[3] = (b[3] - L[3][0]*y[0] - L[3][1]*y[1] - L[3][2]*y[2]) * L_diag_inv[3];
y[4] = (b[4] - L[4][0]*y[0] - L[4][1]*y[1] - L[4][2]*y[2] - L[4][3]*y[3]) * L_diag_inv[4];
```

- 后向替换 \$L^T \Delta q = y\$：

```
delta[4] = y[4] * L_diag_inv[4];
delta[3] = (y[3] - L[4][3]*delta[4]) * L_diag_inv[3];
delta[2] = (y[2] - L[3][2]*delta[3] - L[4][2]*delta[4]) * L_diag_inv[2];
delta[1] = (y[1] - L[2][1]*delta[2] - L[3][1]*delta[3] - L[4][1]*delta[4]) * L_diag_inv[1];
delta[0] = (y[0] - L[1][0]*delta[1] - L[2][0]*delta[2] - L[3][0]*delta[3] - L[4][0]*delta[4]);
```

## 3. Python 硬件封装：hw\_solver\_wrapper.py

`HW Solver Iterator` 类：AXI-Lite 寄存器映射

寄存器偏移	名称	数据类型	方向	描述
0x00	CTRL/STAT	uint32	R/W	[0]=启动, [1]=完成标志
0x10-0x30	sin0-sin4	float32	W	\$\sin(q_i)\$
0x38-0x58	cos0-cos4	float32	W	\$\cos(q_i)\$
0x60-0x88	e0-e5	float32	W	误差向量\$e\$
0x90	lambda	float32	W	阻尼系数\$\lambda\$
0x98-0xD8	d0-d4	float32	R	关节增量\$\Delta q\$
0xE8	status	uint32	R	执行状态码

核心方法：

```

def solve_step(self, q, error, lambda_damping=0.01):
    sin_q, cos_q = np.sin(q), np.cos(q)

    # 写入输入
    for i in range(5):
        self.solver_ip.mmio.write(0x10 + i*8, sin_q[i].astype(np.float32))
        self.solver_ip.mmio.write(0x38 + i*8, cos_q[i].astype(np.float32))
    for i in range(6):
        self.solver_ip.mmio.write(0x60 + i*8, error[i].astype(np.float32))
    self.solver_ip.mmio.write(0x90, np.float32(lambda_damping))

    # 启动计算
    self.solver_ip.mmio.write(0x00, 0x01)

    # 轮询等待 (超时保护)
    for _ in range(10000):
        if self.solver_ip.mmio.read(0x00) & 0x02:
            break

    # 读取输出
    delta = np.array([self.solver_ip.mmio.read(0x98 + i*0x10) for i in range(5)])
    status = self.solver_ip.mmio.read(0xE8)
    return delta, status

```

#### MMIO 开销分析:

- 写入 23 个寄存器 ( $\sin \times 5 + \cos \times 5 + \text{error} \times 6 + \lambda \times 1$ ) : 约 **50  $\mu\text{s}$**
- 硬件计算: 约 **100  $\mu\text{s}$**  (包括雅可比、矩阵乘法、Cholesky)
- 读取 6 个寄存器 ( $\Delta \times 5 + \text{status} \times 1$ ) : 约 **15  $\mu\text{s}$**
- **总延迟: 约 165  $\mu\text{s}/步$**  (相比纯软件的 ~1500  $\mu\text{s}$ , 提升 **9x**)

## 性能数据

### 性能提升总结

经对比测试, 硬件加速相比纯软件实现的性能改善如下:

性能指标	纯软件 (PS)	硬件加速 (PL)	提升倍数
<b>单步迭代速度</b>	基准	改善	<b>~10x</b>
<b>平均迭代时间</b>	基准	减少	<b>~7.9x</b>
<b>吞吐率</b>	基准	提升	<b>~9.2x</b>
<b>总体加速比</b>	1.0x	加速	<b>~2.5x</b>

注: 具体性能数据会因测试场景和目标位姿而异。详细图表和数据见

`notebook/test_results/`

## 资源利用率

FPGA 资源消耗 (Zynq-7020) :

资源类型	使用量	总量	占比
LUT	28456	53200	53.5%
FF	31204	106400	29.3%
BRAM	48	140	34.3%
DSP	134	220	60.9%

分析：

- **DSP 资源占用较高 (60.9%)**：用于浮点乘法、乘加操作（矩阵乘法、Cholesky 分解）
- **BRAM 用于存储中间矩阵**：雅可比 \$J\$ [6×5]、正规矩阵 \$A\$ [5×5]、Cholesky 因子 \$L\$ [5×5]
- **LUT/FF 用于控制逻辑与数据路径**：AXI-Lite 接口、状态机、浮点运算单元
- **优化空间**：若需支持更高自由度（如 6-DOF），可考虑时分复用 DSP（降低资源占用，增加延迟）

## 优化策略

### HLS 优化指令

在 lm\_solver\_so101.hpp 中应用的关键优化：

#### 1. PIPELINE (流水线)

```
#pragma HLS PIPELINE II=1
```

- **作用**：使循环各迭代重叠执行，目标启动间隔 (II) 为 1 周期
- **应用场景**：矩阵乘法内层循环、Cholesky 分解迭代
- **效果**：将延迟从  $O(N^2)$  降至  $O(N)$ （对于可流水化的循环）

#### 2. UNROLL (循环展开)

```
#pragma HLS UNROLL factor=3 // 部分展开
#pragma HLS UNROLL           // 完全展开
```

- **部分展开 (factor=3)**：用于雅可比列计算（5 个关节 → 2 个并行单元）
  - 平衡并行度与资源占用
- **完全展开**：用于小循环（如 5×5 矩阵对角线操作）
  - 消除循环开销，所有迭代并行执行

#### 3. ARRAY\_PARTITION (数组分区)

```
#pragma HLS ARRAY_PARTITION variable=J complete dim=1 // 行完全分区
#pragma HLS ARRAY_PARTITION variable=J complete dim=2 // 列完全分区
#pragma HLS ARRAY_PARTITION variable=A complete       // 全维度完全分区
```

- **作用**：将数组分散到多个 BRAM 块或寄存器
- **效果**：消除访问冲突，允许并行读写

- 代价：BRAM 数量增加 ( $5 \times 5 = 25$  个存储单元)

## Cholesky 算子集成

本项目借鉴了初赛阶段优化的 Cholesky 算子（性能提升 **4-5 倍**），用于求解正规方程  $\left( J^T J + \lambda I \right) \Delta q = -J^T e$ ：

**输入：**对称正定矩阵  $A = J^T J + \lambda I \in \mathbb{R}^{5 \times 5}$

**输出：**下三角矩阵  $L$ （满足  $A = L L^T$ ）

**后续：**通过前向/后向回代求解  $\Delta q$

**优化关键：**

1. 消除循环依赖（通过数据重排）

- 原始算法： $L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk}) / L_{jj}$
- 优化算法：预算算对角线倒数，用乘法替代除法

2. BRAM 带宽优化（双端口访问）

- 分区策略：按行分区（row-wise partition）
- 并行度：同时读取  $L_{ik}$  和  $L_{jk}$

3. 浮点流水线深度调优

- 乘法器延迟：8 周期
- 加法器延迟：11 周期
- 除法器延迟：28 周期
- 优化：用倒数乘法（延迟  $8+8=16$  周期）替代除法

**性能数据** (Cholesky 算子单独测试)：

性能指标	优化前	优化后	改善幅度
延迟 (Latency)	4919	991	79.9%
执行时间	30871	4731	84.7%
启动间隔 (II)	696	124	82.2%
吞吐率 (Throughput)	2.3e5	1.6e6	596%

## 总结

FPGA 硬件加速为实时逆运动学求解提供了显著的性能提升，使得在资源受限的嵌入式平台上也能实现高频控制 (>100 Hz)。本章介绍的软硬件协同架构具有良好的可扩展性，可进一步优化以支持更复杂的机器人（如 6-DOF、7-DOF 冗余臂）或更高精度的求解算法（如二阶 LM、SQP）。

**关键要点：**

1. 性能提升：

- 单步迭代加速约 **10 倍**（通过流水线与并行优化）

## 1. API 速查

- 总体求解加速约 **2.5 倍** (含 MMIO 通信开销)

### 2. 软硬件分工:

- **PS 端**: FK/误差计算、收敛判断、阻尼调整 (需要条件分支与浮点超越函数)

- **PL 端**: 雅可比计算、矩阵乘法、Cholesky 分解 (高度并行的线性代数)

### 3. 接口设计:

- **AXI-Lite 寄存器映射**: 输入/输出寄存器清晰分离

- **三角函数预算算**: 避免 FPGA 实现复杂的 CORDIC 算法

### 4. 优化经验:

- **Cholesky 算子优化**: 延迟降低 **79.9%**, 吞吐率提升 **596%**

- **三角求解完全展开**: 延迟从  $\$O(N^2)$  降至  $\$O(N)$

- **数组分区策略**: 小矩阵 ( $N \leq 5$ ) 完全分区到寄存器

### 5. 可扩展性:

- 修改 `s0101_Params` 即可适配不同机器人 (仅需重新综合)

- 支持更高自由度: 增加 `$N$` 和矩阵维度 (需评估资源占用)

- 支持其他 IK 算法: 替换 `lm_solve_step_s0101` 内核 (如 GN、NR、QP)