

# CodroidPython SDK 手册

版本: 2.1.2 | 包名: `codroid-robot-sdk` | Python: 3.7+

## 目录

#	章节	说明
1	<a href="#">快速上手</a>	安装、连接并运行第一个程序
2	<a href="#">核心概念</a>	生命周期、TCP 模型、单位约定、异常处理
3	<a href="#">CodroidSession / CodroidClient API</a>	主客户端完整 API 参考
4	<a href="#">运动 API</a>	JointPoint、CartesianPoint、MoveInstruction、MotionWaitOptions
5	<a href="#">数据类型与枚举</a>	CommonResponse、CriRealTimeData、RobotFrame、GlobalVariable
6	<a href="#">CRI 实时数据与控制</a>	CriRealtimeDispatcher、TrajectoryGenerator、PacketParser
7	<a href="#">IO 与寄存器</a>	DI/DO/AI/AO 操作、寄存器读写
8	<a href="#">辅助工具</a>	发布/订阅、全局变量、运动学、ConsoleUtf8

## 环境要求

项目	要求
Python	3.7+ (CPython / PyPy, 3.7 ~ 3.14)
操作系统	Linux、Windows、macOS
运行时依赖	无 (可选 <code>colorama</code> 用于彩色终端输出)

## 安装

```
pip install codroid-robot-sdk
```

可选彩色终端输出：

```
pip install "codroid-robot-sdk[color]"
```

## 验证安装

```
python3 -c "import codroid; print(codroid.__version__)"
```

## API 命名约定

所有公共方法使用 **PascalCase**（与 C# / C++ SDK 一致）。

```
# 正确
robot.ConnectRemoteAndSwitchOn()
di = robot.GetDi(0)

# 错误 - snake_case 已在 2.1.1 移除
robot.connect_remote_and_switch_on() # 不存在
```

## 单位约定

层级	线性	角度
SDK 公共 API	mm	deg（度）
TCP JSON 协议	mm	deg
CRI UDP 二进制（线路层）	m	rad（弧度）
<code>CriRealTimeData</code> （已解析）	mm	deg

`CriRealtimePacketParser.parse()` 和 `CriRealtimeDispatcher`（`convert_to_si=True`）会自动处理 m 与 mm、rad 与 deg 的换算。

# 快速上手

---

## 安装

### 通过 pip 安装

```
pip install codroid-robot-sdk
```

### 从源码安装

```
git clone https://github.com/guybod/CodroidSDK.git
cd CodroidSDK/CodroidPython
pip install -e .
```

---

## 最小示例

连接控制器，进入远程模式，上使能。

```
from codroid import CodroidControlInterface, InitConsoleUtf8

InitConsoleUtf8() # Windows cmd 下中文日志不乱码; Linux 上为 no-op

ROBOT_IP = "192.168.1.136"

with CodroidControlInterface(host=ROBOT_IP) as robot:
    robot.EnterRemoteModeViaAuto()
    robot.SwitchOn()
```

运行：

```
python3 demo.py
```

---

## 完整工作流示例

```
from codroid import (
    CodroidControlInterface,
    JointPoint,
    CartesianPoint,
    MoveInstruction,
    InitConsoleUtf8,
)

InitConsoleUtf8()

ROBOT_IP = "192.168.1.136"

with CodroidControlInterface(host=ROBOT_IP) as robot:
```

```
# 1. 连接并上电
robot.EnterRemoteModeViaAuto()
robot.SwitchOn()

# 2. IO 操作
di0 = robot.GetDi(0)
robot.SetDo(10, di0)

# 3. 寄存器
reg_val = robot.GetRegisterValue(0)
robot.SetRegisterValue(0, reg_val + 1)

# 4. 关节运动
robot.MovJ(JointPoint.Degrees([0, 0, 90, 0, 90, 0]), speed=40, acceleration=100)

# 5. 直线运动
robot.MovL(CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]),
           speed=150, acceleration=500)
```

---

## 使用 CodroidClient

`CodroidClient` 继承自 `CodroidSession`，使用后台线程接收数据，支持 publish/subscribe 事件分发。

```
from codroid import CodroidClient, InitConsoleUtf8

InitConsoleUtf8()

with CodroidClient(host="192.168.1.136") as robot:
    robot.EnterRemoteModeViaAuto()
    robot.SwitchOn()
```

---

## 运行示例项目

```
# 基本用法
PYTHONPATH=src python examples/01_basic_usage.py --robot 192.168.8.136

# 运动示例
PYTHONPATH=src python examples/08_move.py --robot 192.168.8.136

# 阻塞式运动
PYTHONPATH=src python examples/15_sync_motion.py --robot 192.168.8.136

# 机器人设置
PYTHONPATH=src python examples/14_robot_parameters.py --robot 192.168.8.136
```

---

## 错误处理

所有 TCP 指令在失败时抛出异常：

异常	条件
<code>CodroidError</code>	基础异常类
<code>CodroidCommandException</code>	控制器返回 <code>err</code> 字段
<code>CodroidNetworkError</code>	TCP 连接或通信失败
<code>CodroidTimeoutError</code>	操作超时

```
from codroid import CodroidControlInterface, CodroidError, CodroidTimeoutError

try:
    with CodroidControlInterface(host="192.168.1.136") as robot:
        robot.EnterRemoteModeViaAuto()
        robot.SwitchOn()
        robot.MovJ([0, 0, 90, 0, 90, 0], speed=40, acceleration=100)
except CodroidTimeoutError:
    print("操作超时")
except CodroidError as e:
    print(f"SDK 错误: {e}")
```

## Windows 控制台 UTF-8

在 `cmd`（非 Windows Terminal）下运行含中文的示例时，请在入口调用：

```
from codroid import InitConsoleUtf8

InitConsoleUtf8()
```

所有 `examples/*.py` 已在 `if __name__ == "__main__"` 首行调用。自建 CLI 请同样处理；`chcp 65001` 不能替代此调用。Linux / macOS 上为 no-op。

# 核心概念

## 客户端生命周期

`CodroidSession` (别名 `CodroidControlInterface`) 和 `CodroidClient` 的典型生命周期:

```
from codroid import CodroidClient

# 方式一: with 语句 (推荐)
with CodroidClient(host="192.168.1.136") as robot:
    robot.EnterRemoteModeViaAuto()
    robot.SwitchOn()
    # ... 使用 API ...
# 自动调用 Disconnect()

# 方式二: 手动管理
robot = CodroidClient(host="192.168.1.136")
robot.Connect()
try:
    robot.EnterRemoteModeViaAuto()
    robot.SwitchOn()
    # ... 使用 API ...
finally:
    robot.Disconnect()
```

## CodroidSession vs CodroidClient

特性	CodroidSession	CodroidClient
传输层	<code>JsonStreamClient</code> (同步阻塞)	<code>TransportClient</code> (后台线程)
请求/响应匹配	同步发送-接收	异步 ID 匹配
Publish/Subscribe	不支持	支持
适用场景	简单脚本、一次性操作	持续收包、事件驱动

## TCP 命令模型

SDK 通过 TCP JSON 与控制器通信。每条指令的流程:

- SDK 分配自增 `id`
- 发送请求: `{"id": N, "ty": "command/path", "db": {...}}`
- 控制器响应: `{"id": N, "ty": "...", "db": {...}, "err": ...}`
- SDK 按 `id` 匹配请求与响应

```
# SDK 内部自动处理 id 分配和匹配
response = robot._send_command("Robot/switchOn", "")
# response = CommonResponse(id=1, ty="Robot/switchOn", db=None, err=None)
```

## CommonResponse

所有 TCP 指令返回 `CommonResponse`：

```
@dataclass
class CommonResponse:
    id: Union[int, str]    # 请求 ID
    ty: str               # 响应类型
    db: Optional[Any]     # 响应数据
    err: Optional[Any]    # 错误信息 (None 表示成功)

    @property
    def is_success(self) -> bool:
        return self.err is None
```

## 异常处理

SDK 定义了四种异常类型：

异常	触发条件
<code>CodroidError</code>	基础异常类；参数校验失败、非法操作
<code>CodroidCommandException</code>	控制器返回 <code>err</code> 字段（协议层错误）
<code>CodroidNetworkError</code>	TCP 连接失败、通信中断
<code>CodroidTimeoutError</code>	操作超时（连接、CRI 等待、阻塞运动）

```
from codroid import (
    CodroidError,
    CodroidCommandException,
    CodroidNetworkError,
    CodroidTimeoutError,
)

try:
    with CodroidClient(host="192.168.1.136") as robot:
        robot.ConnectRemoteAndSwitchOn()
except CodroidNetworkError:
    print("无法连接控制器")
except CodroidTimeoutError:
    print("连接超时")
except CodroidCommandException as e:
    print(f"控制器错误: {e}")
except CodroidError as e:
```

```
print(f"SDK 错误: {e}")
```

## 线程安全

- `CriData` 属性返回当前缓存的 CRI 快照，可从任意线程读取。
- TCP 方法（`GetDi`、`MovJ` 等）可从任意线程调用，但同一客户端实例不支持并发调用。
- `CriRealtimeDispatcher` 的 `SendCommand` / `SendTrajectory` 是线程安全的（UDP 无状态）。

## Publish / Subscribe

`CodroidClient` 支持订阅控制器推送的事件主题：

```
from codroid import CodroidClient, PublishTopics

def on_robot_status(notification):
    print(f"收到 {notification.ty}: {notification.db}")

with CodroidClient(host="192.168.1.136") as robot:
    sub = robot.SubscribePublishTopic(PublishTopics.ROBOT_STATUS, on_robot_status)
    # ... 运行 ...
    sub.dispose() # 取消订阅
```

可用主题见 `PublishTopics` 常量：

- `PROJECT_STATE` — 工程状态变更
- `VAR_UPDATE` — 变量更新
- `ROBOT_STATUS` — 机器人状态
- `ROBOT_POSTURE` — 机器人姿态
- `ROBOT_COORDINATE` — 机器人坐标
- `LOG` — 日志
- `ERROR` — 错误



# CodroidSession / CodroidClient API 参考

`CodroidSession`（别名 `CodroidControlInterface`）是主会话类，封装了全部 TCP 指令。`CodroidClient` 继承它并替换传输层，增加 publish/subscribe 能力。

## 构造函数

### CodroidSession

```
CodroidSession(  
    host: str = "192.168.1.136",  
    port: int = 9001,  
    local_ip: str = "192.168.1.150",  
    udp_port: int = 10086,  
)
```

参数	类型	默认值	说明
host	str	"192.168.1.136"	控制器 IP 地址
port	int	9001	TCP 端口
local_ip	str	"192.168.1.150"	本机 IP（CRI UDP 推送用）
udp_port	int	10086	本机 UDP 监听端口

### CodroidClient

```
CodroidClient(  
    host: str = "192.168.1.136",  
    port: int = 9001,  
    local_ip: str = "192.168.1.150",  
    udp_port: int = 10086,  
    timeout: float = 10.0,  
)
```

额外参数：

参数	类型	默认值	说明
timeout	float	10.0	TCP 请求超时（秒）

## 属性

### CriData

```
@property
def CriData(self) -> Optional[CriRealTimeData]
```

最新的 CRI 实时数据快照。需要先调用 `StartListenUdp()` 或 `StartCriDataPush()` 开始接收 UDP 数据。

```
robot.StartListenUdp()
robot.WaitForCriData()
data = robot.CriData
print(f"关节角: {data.joint_position}")
print(f"TCP 位姿: {data.tcp_pose}")
print(f"是否运动中: {data.status.is_moving}")
```

## 连接管理

### Connect

```
def Connect(self) -> CodroidSession
```

建立 TCP 连接。返回自身以支持链式调用。使用 `with` 语句时自动调用。

### Disconnect

```
def Disconnect(self) -> None
```

断开 TCP 连接并停止 CRI 接收。使用 `with` 语句时自动调用。

## 便捷连接

### ConnectRemoteAndSwitchOn

```
def ConnectRemoteAndSwitchOn(self) -> CommonResponse
```

组合操作: `EnterRemoteModeViaAuto()` → `SwitchOn()`。须先 `Connect()` 或在 `with` 块内。

```
with CodroidClient(host="192.168.1.136") as robot:
    robot.ConnectRemoteAndSwitchOn()
```

## EnterRemoteModeViaAuto

```
def EnterRemoteModeViaAuto(self) -> CommonResponse
```

先 ToAuto() 再 ToRemote()。

## EnterManualModeViaAuto

```
def EnterManualModeViaAuto(self) -> CommonResponse
```

先 ToAuto() 再 ToManual()。

---

## 模式切换

### SwitchOn / SwitchOff

```
def SwitchOn(self) -> CommonResponse  
def SwitchOff(self) -> CommonResponse
```

上使能 / 下使能。

### ToRemote / ToManual / ToAuto

```
def ToRemote(self) -> CommonResponse  
def ToManual(self) -> CommonResponse  
def ToAuto(self) -> CommonResponse
```

切换到远程 / 手动 / 自动模式。

### ToSimulation / ToActual

```
def ToSimulation(self) -> CommonResponse  
def ToActual(self) -> CommonResponse
```

切换到仿真 / 实机模式。

### StartDrag / StopDrag

```
def StartDrag(self) -> CommonResponse  
def StopDrag(self) -> CommonResponse
```

进入 / 退出拖拽示教模式。仅在远程模式和手动模式下可用。

# ClearSystemError

```
def ClearSystemError(self) -> CommonResponse
```

清除系统错误。

## 非阻塞运动指令

### MovJ

```
def MovJ(
    self,
    target: Union[JointPoint, CartesianPoint, MovePoint],
    speed: float,
    acceleration: float,
    blend: float = 0.0,
    coor: Optional[Sequence[float]] = None,
    tool: Optional[Sequence[float]] = None,
) -> CommonResponse
```

关节插补运动。目标可为 `JointPoint`（发 jp）或 `CartesianPoint`（发 cp+rj）。

参数	类型	说明
target	JointPoint / CartesianPoint	运动目标
speed	float	速度
acceleration	float	加速度
blend	float	平滑半径，默认 0（精确到达）
coor	Sequence[float]	用户坐标系 [x, y, z, a, b, c]
tool	Sequence[float]	工具坐标系 [x, y, z, a, b, c]

```
robot.MovJ(JointPoint.Degrees([0, 0, 90, 0, 90, 0]), speed=40, acceleration=100)
robot.MovJ(CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]), speed=40, acceleration=100)
```

## MovL

```
def MovL(  
    self,  
    target: Union[CartesianPoint, JointPoint, MovePoint],  
    speed: float,  
    acceleration: float,  
    blend: float = 0.0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

直线插补运动。目标可为 `CartesianPoint` 或 `JointPoint`。

## MovC

```
def MovC(  
    self,  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    speed: float,  
    acceleration: float,  
    blend: float = 0.0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

圆弧运动。中间点与目标均为 `CartesianPoint`。

## MovCircle

```
def MovCircle(  
    self,  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    circle_num: int,  
    speed: float,  
    acceleration: float,  
    blend: float = 0.0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

整圆运动。 `circle_num` 为圈数。

## Move

```
def Move(  
    self,  
    path: Union[MotionPath, List[MoveInstruction], List[Dict[str, Any]]],  
) -> CommonResponse
```

多段路径执行。推荐使用 `List[MoveInstruction]`。

```
path = [  
    MoveInstruction.MovJ(JointPoint.Degrees([0, 0, 90, 0, 90, 0]), speed=40, acc=100),  
    MoveInstruction.MovL(CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]), speed=150,  
acc=500),  
    MoveInstruction.MovL(JointPoint.Degrees([0, 0, 0, 0, 0, 0]), speed=150, acc=500),  
]  
robot.Move(path)
```

## 阻塞式运动指令

`*Sync` 方法发送运动指令后自动轮询 CRI 数据，直到机器人稳定到达目标。需要先启动 CRI 数据推送。

### MovJSync

```
def MovJSync(  
    self,  
    target: Union[JointPoint, CartesianPoint],  
    speed: float,  
    acceleration: float,  
    wait: Optional[MotionWaitOptions] = None,  
    blend: float = 0.0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> bool
```

阻塞式关节运动。到达目标返回 `True`。

```
robot.StartListenUdp()  
robot.WaitForCriData()  
robot.MovJSync(JointPoint.Degrees([0, 0, 90, 0, 90, 0]), speed=40, acceleration=100)
```

### MovLSync

```
def MovLSync(  
    self,  
    target: Union[CartesianPoint, JointPoint],  
    speed: float,  
    acceleration: float,  
    wait: Optional[MotionWaitOptions] = None,  
    blend: float = 0.0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> bool
```

阻塞式直线运动。

## MovCSync

```
def MovCSync(  
    self,  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    speed: float,  
    acceleration: float,  
    wait: Optional[MotionWaitOptions] = None,  
    blend: float = 0.0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> bool
```

阻塞式圆弧运动。

## MovCircleSync

```
def MovCircleSync(  
    self,  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    circle_num: int,  
    speed: float,  
    acceleration: float,  
    wait: Optional[MotionWaitOptions] = None,  
    blend: float = 0.0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> bool
```

阻塞式整圆运动。

## MoveSync

```
def MoveSync(  
    self,  
    path: Union[MotionPath, List[MoveInstruction], List[Dict[str, Any]]],  
    wait: Optional[MotionWaitOptions] = None,  
) -> bool
```

阻塞式路径执行。等待 CRI 确认最后一段到达目标。

## MotionWaitOptions

控制阻塞运动的等待行为：

```
@dataclass
class MotionWaitOptions:
    timeout: float = 60.0 # 整体超时 (秒)
    poll_interval: float = 0.05 # 轮询间隔 (秒)
    cri_stale_timeout: float = 0.5 # CRI 数据过期判定 (秒)
    settled_samples: int = 3 # 连续稳定采样数
    joint_tolerance_deg: float = 0.2 # 关节容差 (度)
    cartesian_position_tolerance_mm: float = 1.0 # 笛卡尔位置容差 (mm)
    cartesian_orientation_tolerance_deg: float = 1.0 # 笛卡尔姿态容差 (度)
```

```
from codroid import MotionWaitOptions

opts = MotionWaitOptions(timeout=30.0, joint_tolerance_deg=0.5)
robot.MovJSync(JointPoint.Degrees([0, 0, 90, 0, 90, 0]),
               speed=40, acceleration=100, wait=opts)
```

## 运动控制

### PauseRobotMotion / ResumeRobotMotion

```
def PauseRobotMotion(self) -> CommonResponse
def ResumeRobotMotion(self) -> CommonResponse
```

暂停 / 恢复当前运动。

### StopRobotMove

```
def StopRobotMove(self) -> CommonResponse
```

停止当前运动。

## MoveTo 指令

### MoveTo

```
def MoveTo(
    self,
    move_type: MoveToType,
    target: Optional[MoveToTarget] = None,
) -> CommonResponse
```

运动到预设位置。MoveToType 枚举值：

值	说明
STOP (-1)	停止 MoveTo



值	说明
HOME (0)	Home 点
SAFE (1)	安全位
CANDLE (2)	蜡烛位
PACK (3)	打包位
JOINT (4)	关节规划到指定目标
LINEAR (5)	直线规划到指定目标
RESUME (6)	程序恢复点

```
from codroid import MoveToType

robot.MoveTo(MoveToType.HOME) # 回 Home
robot.MoveTo(MoveToType.SAFE) # 回安全位
```

启动后须每 0.5s 调用 `MoveToHeartbeat()`。

MoveToHeartbeat

```
def MoveToHeartbeat(self) -> CommonResponse
```

MoveTo 心跳。须在 MoveTo 运动期间每隔 0.5s 调用一次。

StopMoveTo

```
def StopMoveTo(self) -> CommonResponse
```

停止当前 MoveTo 运动。

Jog 指令

StartJog

```
def StartJog(
    self,
    mode: JogMode,
    index: int,
    speed: float,
    coor_type: JogCoorType = JogCoorType.USER,
    coor_id: int = 1,
) -> CommonResponse
```

启动点动。

参数	类型	说明
mode	JogMode	JOINT (1) 关节点动 / LINEAR (2) 直线点动
index	int	关节序号(1-6) 或直线轴序号(1-6 对应 xyzabc)
speed	float	速度 (-1.0 ~ 1.0)
coord_type	JogCoordType	USER (0) / TOOL (1)
coord_id	int	用户坐标系 ID

```
from codroid import JogMode

robot.StartJog(JogMode.JOINT, index=1, speed=0.5) # 关节 1 正向点动
```

## StopJog

```
def StopJog(self) -> CommonResponse
```

停止点动。

## JogHeartbeat

```
def JogHeartbeat(self) -> CommonResponse
```

点动心跳。须在点动期间每隔 0.5s 调用一次。

## 速度倍率

### SetManualMoveRate / SetAutoMoveRate

```
def SetManualMoveRate(self, rate: int) -> CommonResponse
def SetAutoMoveRate(self, rate: int) -> CommonResponse
```

设置手动 / 自动运动倍率。rate 范围 1~100。

## CRI 数据推送

### StartListenUdp

```
def StartListenUdp(self) -> None
```

一键启动 CRI 数据接收：先停止旧推送 → 启动本地 UDP 监听 → 通知控制器开始推送。

```
robot.StartListenUdp()
robot.WaitForCriData()
data = robot.CriData
```

## WaitForCriData

```
def WaitForCriData(self, timeout: float = 5.0) -> CriRealTimeData
```

等待第一个 CRI 数据包到达。超时抛出 `CodroidTimeoutError`。

## StartCriDataPush

```
def StartCriDataPush(
    self,
    ip: str,
    port: int,
    duration: int = 100,
    high_precision: bool = True,
    mask: int = 0xFFFF,
) -> CommonResponse
```

手动开启 CRI UDP 推送。`port` 范围 10000-65534。`duration` 为推送周期（ms）。

## StopCriDataPush

```
def StopCriDataPush(self, ip: Optional[str] = None, port: Optional[int] = None) ->
CommonResponse
```

停止 CRI 推送。

## StartCriControl

```
def StartCriControl(
    self,
    filter_type: CriFilterType = CriFilterType.NONE,
    duration: int = 1,
    start_buffer: int = 3,
) -> CommonResponse
```

开启 CRI 实时控制模式。`duration` 为指令间隔（ms，1-16，且可整除 1000）。

## StopCriControl

```
def StopCriControl(self) -> CommonResponse
```

关闭 CRI 实时控制。

---

## IO 操作

### GetDi / GetDo / GetAi / GetAo

```
def GetDi(self, port: int) -> int      # 数字输入, 端口 0~15, 返回 0 或 1
def GetDo(self, port: int) -> int      # 数字输出, 端口 0~15, 返回 0 或 1
def GetAi(self, port: int) -> float    # 模拟输入, 端口 0~3
def GetAo(self, port: int) -> float    # 模拟输出, 端口 0~3
```

```
di0 = robot.GetDi(0)
do5 = robot.GetDo(5)
ai0 = robot.GetAi(0)
```

### SetDo / SetAo

```
def SetDo(self, port: int, value: int) -> CommonResponse    # value: 0 或 1
def SetAo(self, port: int, value: float) -> CommonResponse
```

```
robot.SetDo(10, 1)
robot.SetAo(0, 3.14)
```

### GetIoValues

```
def GetIoValues(self, io_requests: List[Dict[str, Any]]) -> CommonResponse
```

批量读取 IO。 `io_requests` 格式: `[{"type": "DI", "port": 0}, ...]`。

## 寄存器操作

### GetRegisterValue / GetRegisterValues

```
def GetRegisterValue(self, address: int) -> Any
def GetRegisterValues(self, addresses: List[int]) -> CommonResponse
```

```
val = robot.GetRegisterValue(0)
vals = robot.GetRegisterValues([0, 1, 2])
```

### SetRegisterValue

```
def SetRegisterValue(self, address: int, value: Any) -> CommonResponse
```

```
robot.SetRegisterValue(0, 42)
```

## SetExtendArrayType / RemoveExtendArray

```
def SetExtendArrayType(self, index: int, data_type: ExtendArrayType) -> CommonResponse
def RemoveExtendArray(self, index: int) -> CommonResponse
```

设置 / 删除扩展数组数据类型。 `index` 范围 0~999。

## 机器人设置

### GetRobotParameters

```
def GetRobotParameters(self) -> RobotParameters
```

获取设置界面参数快照。返回 `RobotParameters`，包含工具坐标系表、负载坐标系表、用户坐标系表及默认 ID。

```
params = robot.GetRobotParameters()
print(f"默认工具: {params.default_tool_id}")
print(f"默认负载: {params.default_payload_id}")
for frame in params.tool:
    print(f"  Tool[{frame.id}]: x={frame.x}, y={frame.y}, z={frame.z}")
```

### SetToolFrame / SaveToolFrames

```
def SetToolFrame(self, frame_id: int, frame) -> CommonResponse
def SaveToolFrames(self, frames) -> CommonResponse
```

`SetToolFrame` 修改单个工具坐标系（先读后改再保存， `frame_id` 仅允许 1~15）。 `SaveToolFrames` 下发完整表（16 项，id=0 须全零）。

```
from codroid import RobotFrame

robot.SetToolFrame(1, RobotFrame(id=1, x=100, y=0, z=50, a=0, b=0, c=0))
```

### SetPayloadFrame / SavePayloadFrames

```
def SetPayloadFrame(self, frame_id: int, frame) -> CommonResponse
def SavePayloadFrames(self, frames) -> CommonResponse
```

`SetPayloadFrame` 修改单个负载坐标系（ `frame_id` 仅允许 1~15）。 `SavePayloadFrames` 下发完整表。

```
from codroid import RobotPayloadFrame

robot.SetPayloadFrame(1, RobotPayloadFrame(id=1, m=2.5, mx=0, my=0, mz=50))
```

## SetUserCoordinateFrame / SaveUserCoordinateFrames

```
def SetUserCoordinateFrame(self, frame_id: int, frame) -> CommonResponse
def SaveUserCoordinateFrames(self, frames) -> CommonResponse
```

修改单个 / 下发完整用户坐标系表。

## SetDefaultToolId / SetDefaultPayloadId / SetDefaultUserCoordinateId

```
def SetDefaultToolId(self, tool_id: int) -> CommonResponse
def SetDefaultPayloadId(self, payload_id: int) -> CommonResponse
def SetDefaultUserCoordinateId(self, coordinate_id: int) -> CommonResponse
```

设置默认工具 / 负载 / 用户坐标系编号。范围 0~15。

## SetCollisionSensitivity

```
def SetCollisionSensitivity(self, sensitivity: int) -> CommonResponse
```

设置碰撞检测灵敏度。范围 0~100。仅固件 2.3.2.10+ 可用。

## SetPayload

```
def SetPayload(self, payload_id: int) -> CommonResponse
```

设置当前负载编号。仅固件 2.3.2.10+ 可用。

---

## 项目与脚本

### RunScript

```
def RunScript(
    self,
    main_script: str,
    sub_threads: Optional[Dict[str, str]] = None,
    sub_programs: Optional[Dict[str, str]] = None,
    interrupts: Optional[Dict[str, str]] = None,
    vars: Optional[Dict[str, Any]] = None,
) -> CommonResponse
```

运行 Lua 脚本。

```
robot.RunScript('print("hello")', vars={"speed": 100})
```

## Run / RunByIndex / RunStep

```
def Run(self, project_id: str) -> CommonResponse
def RunByIndex(self, index: int) -> CommonResponse
def RunStep(self, project_id: str) -> CommonResponse
```

运行 / 单步运行工程。

## PauseProject / ResumeProject / StopProject

```
def PauseProject(self) -> CommonResponse
def ResumeProject(self) -> CommonResponse
def StopProject(self) -> CommonResponse
```

暂停 / 恢复 / 停止工程。

## EnterRemoteScriptMode

```
def EnterRemoteScriptMode(self) -> CommonResponse
```

进入远程脚本模式。

## SetStartLine / ClearStartLine

```
def SetStartLine(self, line: int) -> CommonResponse
def ClearStartLine(self) -> CommonResponse
```

设置 / 清除启动行。

---

## RS485 通信

### Rs485Init

```
def Rs485Init(
    self,
    baudrate: Union[RS485BaudRate, int],
    stop_bit: RS485StopBits = RS485StopBits.ONE,
    parity: RS485Parity = RS485Parity.NONE,
) -> CommonResponse
```

初始化末端 RS485。

```
from codroid import RS485BaudRate

robot.Rs485Init(RS485BaudRate.B115200)
```

## Rs485Flush

```
def Rs485Flush(self) -> CommonResponse
```

清空 RS485 读取缓存。

## Rs485Read

```
def Rs485Read(self, length: int, timeout: int = 3000) -> CommonResponse
```

读取 RS485 数据。length 最大 128 字节，timeout 最大 3000ms。

## Rs485Write

```
def Rs485Write(self, data: Union[List[int], bytes]) -> CommonResponse
```

发送 RS485 数据。最大 127 字节。

---

## 运动学

### AposToCpos

```
def AposToCpos(  
    self,  
    jp: Sequence[float],  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
    ep: Sequence[float] = [],  
) -> CommonResponse
```

正解（关节 → 笛卡尔）。jp 为 6 个关节角（度）。

### CposToApos

```
def CposToApos(  
    self,  
    cp: Sequence[float],  
    rj: Optional[Sequence[float]] = None,  
    ep: Sequence[float] = [],  
) -> CommonResponse
```

逆解（笛卡尔 → 关节）。cp 为 [x, y, z, a, b, c]，rj 为参考关节角（默认 [20, 20, 20, 20, 20, 20]）。



## CalculateRelativePose

```
def CalculateRelativePose(  
    self,  
    pos: Sequence[float],  
    offset: Sequence[float],  
    coord_type: CoordinateType = CoordinateType.TOOL,  
    pos_coord: Optional[Sequence[float]] = None,  
    coord: Optional[Sequence[float]] = None,  
) -> CommonResponse
```

笛卡尔坐标偏移计算。

---

## Publish / Subscribe（仅 CodroidClient）

### SubscribePublishTopic

```
def SubscribePublishTopic(  
    self,  
    topic_ty: str,  
    handler: Callable[[PublishNotification], None],  
    tc_milliseconds: int = 100,  
) -> PublishTopicSubscription
```

订阅控制器推送主题。返回 `PublishTopicSubscription`，调用 `.dispose()` 取消订阅。

```
from codroid import CodroidClient, PublishTopics  
  
def on_status(notification):  
    print(f"{notification.ty}: {notification.db}")  
  
with CodroidClient(host="192.168.1.136") as robot:  
    sub = robot.SubscribePublishTopic(PublishTopics.ROBOT_STATUS, on_status)  
    # ... 运行 ...  
    sub.dispose()
```

---

## 调试

```
robot.debug = True # 打印发送/接收的原始 JSON
```

# 运动 API 参考

## 运动目标类型

### JointPoint

```
@dataclass
class JointPoint:
    jp: List[float] # 6 个关节角 (度)
```

关节目标。业务层用于声明「这是六轴关节角」，避免与 TCP 位姿混淆。

#### 工厂方法

```
JointPoint.Degrees(joints_deg: Sequence[float]) -> JointPoint
```

由六轴关节角（度）构造。

```
jp = JointPoint.Degrees([0, 0, 90, 0, 90, 0])
```

### CartesianPoint

```
@dataclass
class CartesianPoint:
    cp: List[float] # TCP 位姿 [x,y,z,rx,ry,rz] (mm + 度)
    rj: Optional[List[float]] = None # 逆解参考关节角 (度)
```

笛卡尔目标。`cp` 必填；`rj` 可选（打包时缺省为控制器默认参考关节）。

#### 工厂方法

```
CartesianPoint.MmDeg(pose_mm_deg: Sequence[float]) -> CartesianPoint
```

TCP 位姿 `[x,y,z,rx,ry,rz]` (mm + 度)，无参考关节。

```
CartesianPoint.MmDegWithRef(
    pose_mm_deg: Sequence[float],
    ref_joints_deg: Sequence[float],
) -> CartesianPoint
```

TCP 位姿 + 逆解参考关节（度）。当 `movJ/movL` 到 TCP 且在意姿态解时推荐使用。

```
cp = CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90])
cp_with_ref = CartesianPoint.MmDegWithRef(
    [400, 200, 500, 180, 0, 90],
    [0, 0, 90, 0, 90, 0],
)
```

---

## MovePoint

```
@dataclass
class MovePoint:
    jp: Optional[Sequence[float]] = None
    cp: Optional[Sequence[float]] = None
    rj: Optional[Sequence[float]] = None
    ep: Optional[Sequence[float]] = None
```

通用运动点位定义。通常不直接使用，而是通过 `JointPoint` / `CartesianPoint` 构造。

### 工厂方法

```
MovePoint.FromJoint(joint: JointPoint) -> MovePoint
MovePoint.FromCartesian(cart: CartesianPoint) -> MovePoint
```

---

## 运动指令

### MoveInstruction

```
@dataclass
class MoveInstruction:
    motion_type: MotionType
    target: MovePoint
    speed: float
    acc: float
    blend: float = 0.0
    relative_blend: int = 0
    middle: Optional[MovePoint] = None
    circle_num: Optional[int] = None
    coor: Optional[Sequence[float]] = None
    tool: Optional[Sequence[float]] = None
```

单段 `Robot/move` 指令。请用类方法构建，勿手写 `type` + 裸 `MovePoint`。

### 工厂方法

#### MoveInstruction.MovJ

```
MoveInstruction.MovJ(
    target: Union[JointPoint, CartesianPoint],
    speed: float,
    acc: float,
    blend: float = 0.0,
    relative_blend: int = 0,
    coor: Optional[Sequence[float]] = None,
    tool: Optional[Sequence[float]] = None,
) -> MoveInstruction
```

关节运动 movJ。目标可为关节或 TCP。

### MoveInstruction.MovL

```
MoveInstruction.MovL(  
    target: Union[CartesianPoint, JointPoint],  
    speed: float,  
    acc: float,  
    blend: float = 0.0,  
    relative_blend: int = 0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> MoveInstruction
```

直线运动 movL。目标可为 TCP 或关节。

### MoveInstruction.MovC

```
MoveInstruction.MovC(  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    speed: float,  
    acc: float,  
    blend: float = 0.0,  
    relative_blend: int = 0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> MoveInstruction
```

圆弧运动 movC。中间点与目标均为 TCP。

### MoveInstruction.MovCircle

```
MoveInstruction.MovCircle(  
    middle: CartesianPoint,  
    target: CartesianPoint,  
    circle_num: int,  
    speed: float,  
    acc: float,  
    blend: float = 0.0,  
    relative_blend: int = 0,  
    coor: Optional[Sequence[float]] = None,  
    tool: Optional[Sequence[float]] = None,  
) -> MoveInstruction
```

整圆运动 movCircle。

---

## MotionType

```
class MotionType(str, Enum):
    MOVJ = "movJ"
    MOVL = "movL"
    MOVC = "movC"
    MOVCIRCLE = "movCircle"
```

## MoveTo 目标

### MoveToTarget

```
@dataclass
class MoveToTarget:
    cp: Optional[Sequence[float]] = None
    jp: Optional[Sequence[float]] = None
    ep: Sequence[float] = field(default_factory=list)
```

MoveTo 专用目标结构。

### 工厂方法

```
MoveToTarget.Joint(joint: JointPoint) -> MoveToTarget
MoveToTarget.Cartesian(cart: CartesianPoint) -> MoveToTarget
```

## MoveToType

```
class MoveToType(IntEnum):
    STOP = -1      # 停止 MoveTo
    HOME = 0       # Home 点
    SAFE = 1       # 安全位
    CANDLE = 2     # 蜡烛位
    PACK = 3       # 打包位
    JOINT = 4      # 关节规划
    LINEAR = 5     # 直线规划
    RESUME = 6     # 程序恢复点
```

## 点动参数

### JogMode

```
class JogMode(IntEnum):
    JOINT = 1      # 关节点动
    LINEAR = 2     # 直线点动
```

## JogCoorType

```
class JogCoorType(IntEnum):
    USER = 0      # 用户坐标系
    TOOL = 1      # 工具坐标系
```

## MotionPath (过渡 API)

```
class MotionPath:
    def add(self, instruction: MoveInstruction) -> MotionPath
    def MovJ(self, target, speed, acc, blend=0.0) -> MotionPath
    def MovL(self, target, speed, acc, blend=0.0) -> MotionPath
    def MovC(self, target, middle, speed, acc, blend=0.0) -> MotionPath
    def clear(self) -> None
    def get_commands(self) -> List[Dict[str, Any]]
```

运动路径构建器。新代码优先使用 `List[MoveInstruction]` + `Move()`。

## 完整多段路径示例

```
from codroid import (
    CodroidClient,
    JointPoint,
    CartesianPoint,
    MoveInstruction,
    MotionWaitOptions,
    InitConsoleUtf8,
)

InitConsoleUtf8()

with CodroidClient(host="192.168.8.136") as robot:
    robot.ConnectRemoteAndSwitchOn()

    # 四组合路径
    path = [
        MoveInstruction.MovJ(JointPoint.Degrees([0, 0, 90, 0, 90, 0]),
                             speed=40, acc=100),
        MoveInstruction.MovJ(CartesianPoint.MmDeg([400, 200, 500, 180, 0, 90]),
                             speed=40, acc=100),
        MoveInstruction.MovL(CartesianPoint.MmDeg([400, -200, 500, 180, 0, 90]),
                             speed=150, acc=500),
        MoveInstruction.MovL(JointPoint.Degrees([0, 0, 0, 0, 0, 0]),
                             speed=150, acc=500),
    ]
    robot.Move(path)

    # 阻塞式路径执行
    robot.StartListenUdp()
```

```
robot.WaitForCridata()  
robot.MoveSync(path, wait=MotionWaitOptions(timeout=120.0))
```

## 运动参数说明

### speed / acceleration

- `speed`：速度值，具体单位取决于运动类型和控制器配置。
- `acceleration`：加速度值。

### blend

平滑半径。默认 `0.0`（精确到达目标点）。设置大于 0 的值时，机器人在接近目标点时平滑过渡，不停留。

### coor / tool

可选的用户坐标系和工具坐标系，格式为 `[x, y, z, a, b, c]`（mm + 度）。

# 数据类型与枚举

## 通讯模型

### CommonResponse

```
@dataclass
class CommonResponse:
    id: Union[int, str]
    ty: str
    db: Optional[Any] = None
    err: Optional[Any] = None

    @property
    def is_success(self) -> bool:
        return self.err is None
```

TCP JSON 通用响应。所有 TCP 指令返回此类型。

字段	类型	说明
id	int / str	请求 ID
ty	str	响应类型
db	Any	响应数据
err	Any	错误信息（None 表示成功）

历史别名：CodroidResponse = CommonResponse

### CodroidRequest

```
@dataclass
class CodroidRequest:
    id: Union[int, str]
    ty: str
    db: Optional[Any] = None
```

SDK 通用请求结构。内部使用。



# CRI 实时数据

## CriRealTimeData

```
@dataclass
class CriRealTimeData:
    timestamp: int = 0
    status: CriStatus = field(default_factory=CriStatus)
    joint_pos: List[float] = field(default_factory=list)
    joint_vel: List[float] = field(default_factory=list)
    cartesian_pos: List[float] = field(default_factory=list)
    cartesian_vel: List[float] = field(default_factory=list)
    tcp_speed: float = 0.0
    joint_torque: List[float] = field(default_factory=list)
    external_torque: List[float] = field(default_factory=list)
    extra_axis_pos: List[float] = field(default_factory=list)
```

CRI 解析后实时数据。所有值已转换为 mm + 度。

字段	属性别名	类型	单位	说明
timestamp	timestamp_ms	int	ms	时间戳
status	—	CriStatus	—	状态位
joint_pos	joint_position	List[float]	deg	6 轴关节角
joint_vel	joint_velocity	List[float]	deg/s	6 轴关节速度
cartesian_pos	tcp_pose	List[float]	mm, deg	TCP 位姿 [x,y,z,rx,ry,rz]
cartesian_vel	tcp_velocity	List[float]	mm/s, deg/s	TCP 速度
tcp_speed	tcp_linear_velocity	float	mm/s	TCP 线速度
joint_torque	joint_output_torque	List[float]	Nm	关节输出力矩
external_torque	joint_external_force	List[float]	Nm	外部力矩
extra_axis_pos	external_axis_position	List[float]	—	外部轴位置

历史别名：CRIData = CriRealTimeData

## CriStatus

```
@dataclass
class CriStatus:
    status1_raw: int = 0
    status2_raw: int = 0
    project_running: bool = False
    project_stopped: bool = False
    project_paused: bool = False
    is_enabling: bool = False
```

```
is_disabled: bool = False
is_manual: bool = False
is_dragging: bool = False
is_moving: bool = False
collision_stop: bool = False
is_at_safe_pos: bool = False
has_alarm: bool = False
is_simulation: bool = False
is_emergency_stop: bool = False
is_rescue: bool = False
is_auto: bool = False
is_remote: bool = False
rt_control_mode: bool = False
error_code: int = 0
```

CRI 状态位解析。

字段	说明
project_running	工程运行中
project_stopped	工程已停止
project_paused	工程已暂停
is_enabling	已使能
is_disabled	未使能
is_manual	手动模式
is_dragging	拖拽模式
is_moving	运动中
collision_stop	碰撞停止
is_at_safe_pos	在安全位置
has_alarm	有报警
is_simulation	仿真模式
is_emergency_stop	急停按下
is_rescue	救援模式
is_auto	自动模式
is_remote	远程模式
rt_control_mode	实时控制模式
error_code	错误码（高 8 位）

历史别名: CRIStatus = CriStatus

## 机器人设置

### RobotFrame

```
@dataclass
class RobotFrame:
    id: int
    x: float = 0.0
    y: float = 0.0
    z: float = 0.0
    a: float = 0.0
    b: float = 0.0
    c: float = 0.0
```

工具 / 用户坐标系单帧。

字段	类型	说明
id	int	槽位 ID (0~15, 0 为保留项)
x, y, z	float	位置 (mm)
a, b, c	float	姿态 (度)

```
frame = RobotFrame(id=1, x=100, y=0, z=50, a=0, b=0, c=0)
```

### RobotPayloadFrame

```
@dataclass
class RobotPayloadFrame:
    id: int
    m: float = 0.0
    mx: float = 0.0
    my: float = 0.0
    mz: float = 0.0
```

负载坐标系单帧。

字段	类型	说明
id	int	槽位 ID (0~15, 0 为保留项)
m	float	质量 (kg)
mx, my, mz	float	质心位置 (mm)

```
payload = RobotPayloadFrame(id=1, m=2.5, mx=0, my=0, mz=50)
```

## RobotParameters

```
@dataclass
class RobotParameters:
    default_tool_id: int = 0
    default_payload_id: int = 0
    default_coordinate_id: int = 0
    max_payload: float = 0.0
    tool: List[RobotFrame] = field(default_factory=list)
    payload: List[RobotPayloadFrame] = field(default_factory=list)
    coordinate: List[RobotFrame] = field(default_factory=list)
```

GetRobotParameters() 返回的设置界面参数快照。

字段	类型	说明
default_tool_id	int	默认工具坐标系编号
default_payload_id	int	默认负载编号
default_coordinate_id	int	默认用户坐标系编号
max_payload	float	最大负载（kg）
tool	List[RobotFrame]	工具坐标系表（16 项）
payload	List[RobotPayloadFrame]	负载坐标系表（16 项）
coordinate	List[RobotFrame]	用户坐标系表（16 项）

## 全局变量

### GlobalVariable

```
@dataclass
class GlobalVariable:
    value: Any
    note: Optional[str] = None
```

全局变量数据模型。

字段	类型	说明
value	Any	变量值（支持 int, float, str, list, dict）
note	str	变量备注

```
var = GlobalVariable(value=42, note="计数器")
robot.SaveGlobalVar("counter", var)
```

## 枚举类型

### CoordinateType

```
class CoordinateType(str, Enum):
    USER = "user"
    TOOL = "tool"
```

坐标系类型。

### IOType

```
class IOType(str, Enum):
    DI = "DI" # 数字输入
    DO = "DO" # 数字输出
    AI = "AI" # 模拟输入
    AO = "AO" # 模拟输出
```

### ExtendArrayType

```
class ExtendArrayType(str, Enum):
    BOOL = "Bool"
    UINT8 = "UInt8"
    INT8 = "Int8"
    UINT16 = "UInt16"
    INT16 = "Int16"
    UINT32 = "UInt32"
    INT32 = "Int32"
    FLOAT32 = "Float32"
```

扩展数组数据类型。

### RS485BaudRate

```
class RS485BaudRate(IntEnum):
    B110 = 110
    B300 = 300
    B600 = 600
    B1200 = 1200
    B2400 = 2400
    B4800 = 4800
    B9600 = 9600
    B14400 = 14400
    B19200 = 19200
    B38400 = 38400
    B56000 = 56000
```

```
B57600 = 57600
B115200 = 115200
B128000 = 128000
B230400 = 230400
```

## RS485StopBits

```
class RS485StopBits(IntEnum):
    ONE = 1
    TWO = 2
```

## RS485Parity

```
class RS485Parity(IntEnum):
    NONE = 0
    ODD = 1
    EVEN = 2
```

## CriMask

```
class CriMask(IntFlag):
    TIMESTAMP = 1 << 0
    STATUS_1 = 1 << 1
    STATUS_2 = 1 << 2
    JOINT_POS = 1 << 8
    JOINT_VEL = 1 << 9
    CARTESIAN_POS = 1 << 10
    CARTESIAN_VEL = 1 << 11
    TCP_SPEED = 1 << 12
    JOINT_TORQUE = 1 << 13
    EXTERNAL_TORQUE = 1 << 14
    EXTRA_AXIS_POS = 1 << 15
```

CRI 推送位掩码。

历史别名: `CRIMask = CriMask`

## CriFilterType

```
class CriFilterType(IntEnum):
    NONE = 0
    AVERAGE = 1
    LOW_PASS = 2
    ELLIPTIC = 3
```

CRI 实时控制滤波类型。

历史别名: `CRIFilterType = CriFilterType`

---

# 异常

## CodroidError

```
class CodroidError(Exception):  
    pass
```

基础异常类。参数校验失败、非法操作等。

## CodroidCommandException

```
class CodroidCommandException(CodroidError):  
    pass
```

控制器返回 `err` 字段。

## CodroidNetworkError

```
class CodroidNetworkError(CodroidError):  
    pass
```

TCP 连接或通信失败。

## CodroidTimeoutError

```
class CodroidTimeoutError(CodroidError):  
    pass
```

操作超时。

# CRI 实时数据与控制 API 参考

## CriRealtimePacketParser

```
class CriRealtimePacketParser:
    @staticmethod
    def parse(data: bytes) -> Optional[CriRealTimeData]
```

解析 308 字节 CRI UDP 包为 `CriRealTimeData` (mm + 度)。非 308 字节的包返回 `None`。

自动完成线路层单位转换：

- 关节角：rad → deg
- TCP 位置：m → mm
- TCP 姿态：rad → deg
- 速度：m/s → mm/s, rad/s → deg/s

```
from codroid import CriRealtimePacketParser

data = CriRealtimePacketParser.parse(raw_bytes)
if data is not None:
    print(f"关节角: {data.joint_position}")
    print(f"TCP 位姿: {data.tcp_pose}")
    print(f"运动中: {data.status.is_moving}")
```

## CriStreamHandler

```
class CriStreamHandler:
    def __init__(
        self,
        high_precision: bool = True,
        mask: int = 0xFFFF,
        joint_count: int = 6,
        extra_axis_count: int = 0,
    )
```

可变 mask / 精度的 CRI UDP 包解析。支持灵活的位掩码和精度配置。

参数	类型	默认值	说明
high_precision	bool	True	双精度 (Float64) / 单精度 (Float32)
mask	int	0xFFFF	位掩码，控制解析哪些字段
joint_count	int	6	关节数
extra_axis_count	int	0	外部轴数



## bind

```
def bind(self, port: int) -> None
```

绑定 UDP 端口。

## parse\_packet

```
def parse_packet(self, data: bytes) -> CriRealTimeData
```

解析单个 CRI 数据包。

```
from codroid import CriStreamHandler

handler = CriStreamHandler(high_precision=True, mask=0xFFFF, joint_count=6)
handler.bind(10086)

try:
    while True:
        data, addr = handler._sock.recvfrom(2048)
        parsed = handler.parse_packet(data)
        print(f"关节角: {parsed.joint_position}")
finally:
    handler._sock.close()
```

## CriRealtimeDispatcher

```
class CriRealtimeDispatcher:
    def __init__(
        self,
        controller_ip: str,
        controller_udp_port: int = 9030,
        convert_to_si: bool = True,
    )
```

UDP 实时命令下发器。发送 64 字节控制指令到控制器。

参数	类型	默认值	说明
controller_ip	str	—	控制器 IP
controller_udp_port	int	9030	控制器 UDP 端口
convert_to_si	bool	True	自动将 mm/deg 转换为 m/rad

支持 with 语句。

## SendCommand

```
def SendCommand(self, position6: Sequence[float], space: TrajectorySpace) -> None
```

发送单帧实时控制指令。

参数	类型	说明
position6	Sequence[float]	6 个位置值（mm+deg 或 deg）
space	TrajectorySpace	JOINT 或 CARTESIAN

## SendTrajectory

```
def SendTrajectory(  
    self,  
    trajectory: Iterable[TrajectoryPoint],  
    space: TrajectorySpace,  
    period_ms: int,  
) -> None
```

发送完整轨迹。按 period\_ms 间隔逐帧发送。

```
from codroid import CriRealtimeDispatcher, TrajectorySpace  
  
with CriRealtimeDispatcher("192.168.1.136") as dispatcher:  
    dispatcher.SendCommand([0, 0, 90, 0, 90, 0], TrajectorySpace.JOINT)
```

## TrajectoryGenerator

```
class TrajectoryGenerator:  
    @staticmethod  
    def generate(  
        start: Sequence[float],  
        target: Sequence[float],  
        request: TrajectoryRequest,  
    ) -> List[TrajectoryPoint]  
  
    @staticmethod  
    def generate_multi_segment(  
        waypoints: Sequence[Sequence[float]],  
        request: TrajectoryRequest,  
    ) -> List[TrajectoryPoint]
```

离线轨迹生成。

## generate

单段轨迹生成。`start` 和 `target` 均为 6 元素数组。

## generate\_multi\_segment

多段轨迹生成。`waypoints` 至少包含 2 个路径点。

```
from codroid import (
    TrajectoryGenerator,
    TrajectoryRequest,
    TrajectorySpace,
    TrajectoryProfile,
)

request = TrajectoryRequest(
    space=TrajectorySpace.JOINT,
    frequency_hz=100,
    profile=TrajectoryProfile.CUBIC,
    acceleration=100,
    speed=40,
)

trajectory = TrajectoryGenerator.generate(
    [0, 0, 0, 0, 0, 0],
    [0, 0, 90, 0, 90, 0],
    request,
)

for point in trajectory:
    print(f"t={point.t:.3f}s, pos={point.position}")
```

## TrajectoryRequest

```
@dataclass
class TrajectoryRequest:
    space: TrajectorySpace
    frequency_hz: float
    profile: TrajectoryProfile
    acceleration: float
    speed: Optional[float] = None
    duration_seconds: Optional[float] = None
```

轨迹生成请求参数。

参数	类型	说明
<code>space</code>	<code>TrajectorySpace</code>	<code>JOINT</code> （关节空间）或 <code>CARTESIAN</code> （笛卡尔空间）
<code>frequency_hz</code>	<code>float</code>	采样频率（Hz）

参数	类型	说明
<code>profile</code>	<code>TrajectoryProfile</code>	<code>CUBIC</code> （三次多项式）或 <code>TRAPEZOIDAL</code> （梯形）
<code>acceleration</code>	<code>float</code>	加速度
<code>speed</code>	<code>float</code>	速度（与 <code>duration_seconds</code> 二选一）
<code>duration_seconds</code>	<code>float</code>	总时长（与 <code>speed</code> 二选一）

**注意：** `speed` 和 `duration_seconds` 必须且只能设置其中一个。

## TrajectorySpace

```
class TrajectorySpace(Enum):
    JOINT = "Joint"
    CARTESIAN = "Cartesian"
```

## TrajectoryProfile

```
class TrajectoryProfile(Enum):
    CUBIC = "Cubic"
    TRAPEZOIDAL = "Trapezoidal"
```

## TrajectoryPoint

```
@dataclass
class TrajectoryPoint:
    t: float          # 时间（秒）
    position: List[float] # 6 个位置值
```

## 完整 CRI 控制流程示例

```
from codroid import (
    CodroidClient,
    CriRealtimeDispatcher,
    TrajectoryGenerator,
    TrajectoryRequest,
    TrajectorySpace,
    TrajectoryProfile,
    InitConsoleUtf8,
)

InitConsoleUtf8()
```

```

ROBOT_IP = "192.168.8.136"

with CodroidClient(host=ROBOT_IP) as robot:
    robot.ConnectRemoteAndSwitchOn()

    # 1. 开启 CRI 实时控制
    robot.StartCriControl()

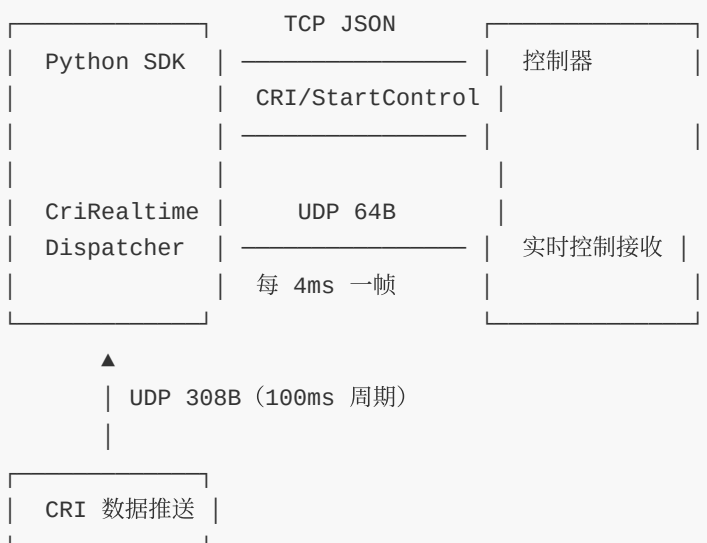
    # 2. 生成轨迹
    request = TrajectoryRequest(
        space=TrajectorySpace.JOINT,
        frequency_hz=250,
        profile=TrajectoryProfile.TRAPEZOIDAL,
        acceleration=100,
        speed=40,
    )
    trajectory = TrajectoryGenerator.generate(
        [0, 0, 0, 0, 0, 0],
        [0, 0, 90, 0, 90, 0],
        request,
    )

    # 3. 下发轨迹
    with CriRealtimeDispatcher(ROBOT_IP) as dispatcher:
        dispatcher.SendTrajectory(trajectory, TrajectorySpace.JOINT, period_ms=4)

    # 4. 关闭实时控制
    robot.StopCriControl()

```

## 工作流程图



# IO 与寄存器 API 参考

---

## IO 操作

### 数字 IO

#### GetDi

```
def GetDi(self, port: int) -> int
```

获取数字输入（DI）值。端口 0~15，返回 0 或 1。

#### GetDo

```
def GetDo(self, port: int) -> int
```

获取数字输出（DO）值。端口 0~15，返回 0 或 1。

#### SetDo

```
def SetDo(self, port: int, value: int) -> CommonResponse
```

设置数字输出（DO）值。端口 0~15，值为 0 或 1。

```
di0 = robot.GetDi(0)           # 读取 DI 端口 0
robot.SetDo(10, 1)             # 设置 DO 端口 10 为 1
robot.SetDo(10, di0)           # 将 DI 值写入 DO
```

---

## 模拟 IO

#### GetAi

```
def GetAi(self, port: int) -> float
```

获取模拟输入（AI）值。端口 0~3。

#### GetAo

```
def GetAo(self, port: int) -> float
```

获取模拟输出（AO）值。端口 0~3。

#### SetAo

```
def SetAo(self, port: int, value: float) -> CommonResponse
```

设置模拟输出（AO）值。端口 0~3。

```
ai0 = robot.GetAi(0)          # 读取 AI 端口 0
robot.SetAo(0, 3.14)         # 设置 AO 端口 0
```

## 批量 IO

### GetIoValues

```
def GetIoValues(self, io_requests: List[Dict[str, Any]]) -> CommonResponse
```

批量读取多个 IO 值。

参数	类型	说明
<code>io_requests</code>	<code>List[Dict]</code>	包含 <code>type</code> 和 <code>port</code> 的列表

```
response = robot.GetIoValues([
    {"type": "DI", "port": 0},
    {"type": "DI", "port": 1},
    {"type": "AI", "port": 0},
])
for item in response.db:
    print(f"{item['type']}{item['port']} = {item['value']}")
```

### SetIoValues

```
def SetIoValues(self, io_list: List[Dict[str, Any]]) -> List[CommonResponse]
```

批量设置 IO 值。内部通过循环调用 `SetDo` / `SetAo` 实现。

```
robot.SetIoValues([
    {"type": "DO", "port": 0, "value": 1},
    {"type": "DO", "port": 1, "value": 0},
    {"type": "AO", "port": 0, "value": 3.14},
])
```

## IO 端口范围

类型	端口范围	值类型
DI（数字输入）	0~15	0 或 1
DO（数字输出）	0~15	0 或 1
AI（模拟输入）	0~3	float
AO（模拟输出）	0~3	float

超出范围抛出 `CodroidError`。

# 寄存器操作

## GetRegisterValue

```
def GetRegisterValue(self, address: int) -> Any
```

获取单个寄存器值。

## GetRegisterValues

```
def GetRegisterValues(self, addresses: List[int]) -> CommonResponse
```

批量获取多个寄存器值。

```
val = robot.GetRegisterValue(0)
vals = robot.GetRegisterValues([0, 1, 2])
for item in vals.db:
    print(f'R{item['address']} = {item['value']}")
```

## SetRegisterValue

```
def SetRegisterValue(self, address: int, value: Any) -> CommonResponse
```

写入寄存器值。

```
robot.SetRegisterValue(0, 42)
robot.SetRegisterValue(1, 3.14)
```

# 扩展数组

## SetExtendArrayType

```
def SetExtendArrayType(self, index: int, data_type: ExtendArrayType) -> CommonResponse
```

设置扩展数组数据类型。index 范围 0~999。

数据类型	说明
ExtendArrayType.BOOL	布尔
ExtendArrayType.UINT8	无符号 8 位整数
ExtendArrayType.INT8	有符号 8 位整数
ExtendArrayType.UINT16	无符号 16 位整数
ExtendArrayType.INT16	有符号 16 位整数



数据类型	说明
<code>ExtendArrayType.UINT32</code>	无符号 32 位整数
<code>ExtendArrayType.INT32</code>	有符号 32 位整数
<code>ExtendArrayType.FLOAT32</code>	32 位浮点数

```
from codroid import ExtendArrayType

robot.SetExtendArrayType(0, ExtendArrayType.FLOAT32)
```

## RemoveExtendArray

```
def RemoveExtendArray(self, index: int) -> CommonResponse
```

删除扩展数组索引（重置数据）。`index` 范围 0~999。

## 完整 IO + 寄存器示例

```
from codroid import CodroidClient, ExtendArrayType, InitConsoleUtf8

InitConsoleUtf8()

with CodroidClient(host="192.168.8.136") as robot:
    robot.ConnectRemoteAndSwitchOn()

    # --- IO ---
    # 读取单个 DI
    di0 = robot.GetDi(0)
    print(f"DI 0 = {di0}")

    # 写入单个 DO
    robot.SetDo(10, 1)

    # 批量读取
    io_data = robot.GetIoValues([
        {"type": "DI", "port": 0},
        {"type": "DI", "port": 1},
        {"type": "AI", "port": 0},
    ])
    for item in io_data.db:
        print(f" {item['type']}{item['port']} = {item['value']}")

    # --- 寄存器 ---
    # 读取单个
    val = robot.GetRegisterValue(0)
    print(f"R0 = {val}")

    # 写入
```

```
robot.SetRegisterValue(0, val + 1)

# 批量读取
regs = robot.GetRegisterValues([0, 1, 2])
for item in regs.db:
    print(f" R{item['address']} = {item['value']}")

# 扩展数组
robot.SetExtendArrayType(0, ExtendArrayType.FLOAT32)
robot.RemoveExtendArray(0)
```

## 辅助工具 API 参考

---

### 发布 / 订阅

#### PublishTopics

```
class PublishTopics:
    PROJECT_STATE = "publish/ProjectState"
    VAR_UPDATE = "publish/VarUpdate"
    ROBOT_STATUS = "publish/RobotStatus"
    ROBOT_POSTURE = "publish/RobotPosture"
    ROBOT_COORDINATE = "publish/RobotCoordinate"
    LOG = "publish/Log"
    ERROR = "publish/Error"
```

控制器推送主题常量。

---

#### PublishNotification

```
@dataclass
class PublishNotification:
    ty: str # 主题类型
    db: Any # 推送数据
    raw_json: str # 原始 JSON 字符串
```

---

#### PublishTopicSubscription

```
class PublishTopicSubscription:
    def dispose(self) -> None
    def Dispose(self) -> None # C# 别名
```

订阅句柄。调用 `dispose()` 取消本地回调（不会向控制器发送取消订阅）。

---

#### SubscribePublishTopic（仅 CodroidClient）

```
def SubscribePublishTopic(
    self,
    topic_ty: str,
    handler: Callable[[PublishNotification], None],
    tc_milliseconds: int = 100,
) -> PublishTopicSubscription
```

订阅控制器推送主题。首次本地订阅时自动向控制器发送订阅请求。

```
from codroid import CodroidClient, PublishTopics, InitConsoleUtf8
```

```
InitConsoleUtf8()

def on_status(notification):
    print(f"收到 {notification.ty}: {notification.db}")

def on_error(notification):
    print(f"错误: {notification.db}")

with CodroidClient(host="192.168.8.136") as robot:
    robot.ConnectRemoteAndSwitchOn()

    sub1 = robot.SubscribePublishTopic(PublishTopics.ROBOT_STATUS, on_status)
    sub2 = robot.SubscribePublishTopic(PublishTopics.ERROR, on_error)

    # ... 运行 ...

    sub1.dispose()
    sub2.dispose()
```

---

## 全局变量

### GetGlobalVars

```
def GetGlobalVars(self) -> CommonResponse
```

获取所有全局变量。

### GetGlobalVarsCatalog

```
def GetGlobalVarsCatalog(self) -> CommonResponse
```

获取全局变量目录（与 `GetGlobalVars` 相同 TCP 请求）。

### SaveGlobalVar

```
def SaveGlobalVar(self, name: str, variable: GlobalVariable) -> CommonResponse
```

保存单个全局变量。

### SaveGlobalVars

```
def SaveGlobalVars(self, variables: Dict[str, GlobalVariable]) -> CommonResponse
```

批量保存全局变量。变量名会经过 Lua 保留字校验。

## RemoveGlobalVars

```
def RemoveGlobalVars(self, names: List[str]) -> CommonResponse
```

删除全局变量。

## GetProjectVar

```
def GetProjectVar(self) -> CommonResponse
```

获取当前所有工程变量值（仅在工程运行中有效）。

```
from codroid import GlobalVariable

# 保存
robot.SaveGlobalVar("counter", GlobalVariable(value=0, note="计数器"))
robot.SaveGlobalVar("name", GlobalVariable(value="test"))

# 批量保存
robot.SaveGlobalVars({
    "x": GlobalVariable(value=100.0),
    "y": GlobalVariable(value=200.0),
})

# 读取
response = robot.GetGlobalVars()
print(response.db)

# 删除
robot.RemoveGlobalVars(["counter", "name"])

# 获取工程变量（工程运行中）
proj_vars = robot.GetProjectVar()
```

---

## 运动学

### AposToCpos

```
def AposToCpos(
    self,
    jp: Sequence[float],
    coor: Optional[Sequence[float]] = None,
    tool: Optional[Sequence[float]] = None,
    ep: Sequence[float] = [],
) -> CommonResponse
```

正解（关节 → 笛卡尔）。jp 为 6 个关节角（度）。

参数	类型	说明
jp	Sequence[float]	6 个关节角（度）
coord	Sequence[float]	用户坐标系 [x,y,z,a,b,c]
tool	Sequence[float]	工具坐标系 [x,y,z,a,b,c]
ep	Sequence[float]	外部轴位置

```
response = robot.AposToCpos([0, 0, 90, 0, 90, 0])
print(f"TCP 位姿: {response.db}")
```

CposToApos

```
def CposToApos(
    self,
    cp: Sequence[float],
    rj: Optional[Sequence[float]] = None,
    ep: Sequence[float] = [],
) -> CommonResponse
```

逆解（笛卡尔 → 关节）。cp 为 [x,y,z,a,b,c]（mm + 度）。rj 为参考关节角（默认 [20,20,20,20,20,20]）。

```
response = robot.CposToApos([400, 200, 500, 180, 0, 90])
print(f"关节角: {response.db}")
```

CalculateRelativePose

```
def CalculateRelativePose(
    self,
    pos: Sequence[float],
    offset: Sequence[float],
    coord_type: CoordinateType = CoordinateType.TOOL,
    pos_coord: Optional[Sequence[float]] = None,
    coord: Optional[Sequence[float]] = None,
) -> CommonResponse
```

笛卡尔坐标偏移计算。

参数	类型	说明
pos	Sequence[float]	当前 TCP 位姿
offset	Sequence[float]	偏移量
coord_type	CoordinateType	USER 或 TOOL
pos_coord	Sequence[float]	当前 TCP 坐标系

参数	类型	说明
<code>coord</code>	<code>Sequence[float]</code>	偏移坐标系

```
from codroid import CoordinateType

response = robot.CalculateRelativePose(
    pos=[400, 200, 500, 180, 0, 90],
    offset=[0, 0, -100, 0, 0, 0],
    coord_type=CoordinateType.TOOL,
)
print(f"偏移后位姿: {response.db}")
```

## ConsoleUtf8

### InitConsoleUtf8

```
def InitConsoleUtf8() -> None
```

Windows 控制台 UTF-8 初始化。在 `cmd`（非 Windows Terminal）下运行含中文的示例时，须在入口调用。Linux / macOS 上为 no-op。

```
from codroid import InitConsoleUtf8

InitConsoleUtf8()
```

历史别名: `init_console_utf8 = InitConsoleUtf8`

## PrintBanner

```
from codroid import PrintBanner

PrintBanner("标题", "副标题")
```

在终端打印彩色横幅标题。需要安装 `colorama`: `pip install "codroid-robot-sdk[color]"`。

## 完整辅助工具示例

```
from codroid import (
    CodroidClient,
    PublishTopics,
    GlobalVariable,
    CoordinateType,
    InitConsoleUtf8,
    PrintBanner,
)
```

```

InitConsoleUtf8()
PrintBanner("辅助工具示例", "Publish / 全局变量 / 运动学")

ROBOT_IP = "192.168.8.136"

# --- Publish / Subscribe ---
def on_status(notification):
    print(f"[Publish] {notification.ty}: {notification.db}")

with CodroidClient(host=ROBOT_IP) as robot:
    robot.ConnectRemoteAndSwitchOn()

    sub = robot.SubscribePublishTopic(PublishTopics.ROBOT_STATUS, on_status)

    # --- 全局变量 ---
    robot.SaveGlobalVar("test_var", GlobalVariable(value=42, note="测试"))
    response = robot.GetGlobalVars()
    print(f"全局变量: {response.db}")
    robot.RemoveGlobalVars(["test_var"])

    # --- 运动学 ---
    fk = robot.AposToCpos([0, 0, 90, 0, 90, 0])
    print(f"正解结果: {fk.db}")

    ik = robot.CposToApos([400, 200, 500, 180, 0, 90])
    print(f"逆解结果: {ik.db}")

    rel = robot.CalculateRelativePose(
        pos=[400, 200, 500, 180, 0, 90],
        offset=[0, 0, -100, 0, 0, 0],
        coor_type=CoordinateType.TOOL,
    )
    print(f"偏移结果: {rel.db}")

    sub.dispose()

```