相对位置

相对位置加入逻辑

一、问题背景

当前行车控制系统仅使用**绝对坐标系**。

- 每次归零后都需要重新设置限位信息；
- 用户操作时不方便进行基于相对坐标的移动。

目标：引入**相对位置信息**。以便于：

- 用户以相对零点的方式进行控制（更符合直觉）；
- 系统依旧使用绝对位置进行限位判断（安全）。

二、现有问题

编号	问题描述	影响
1	若每次移动都将绝对位置写入数据库，性能开销大	数据库频繁访问
2	若由多套外设分别记录零点，可能出现数据不一致	零点漂移、限位失效
3	点动与位置移动逻辑中，未引入零点的绝对位置	无法同时兼顾相对位置与限位判断
4	每次位置判断都从蓝牙盒子读取零点信息	性能浪费

三、解决思路

1. 统一零点管理

- 在蓝牙盒子中**集中存储每个电机的零点绝对位置**；
- 所有外设读取同一份数据，实现零点统一；
- 避免限位不一致问题。

2.引入两层位置体系

定义三个关键量：

名称	含义
`L0 = getZeroLocation()`	零点的绝对位置（来自蓝牙盒子）
`L1 = getLocation()`	当前电机相对零点的相对位置
`L_abs = L0 + L1`	电机真实绝对位置

> **限位判断** 使用 `L_abs`
> **用户操作** 使用 `L1`

四、逻辑修改点

1. 位置移动逻辑（`actionDir(int dir)`）

- 原逻辑：
  目标 = 当前绝对位置 + 移动距离
- 新逻辑：
  目标 = (零点绝对位置 + 相对零点位置) + 移动距离
- 判断限位时，使用目标的绝对位置判断。

2. 点动逻辑

- **点动前检查**：
  判断 `(L0 + L1)` 是否处于限位范围内；
- **点动过程中**（`checkJog()`）：
  实时计算 `(L0 + L1)`，若超出限位则停止。

3. 零点更新

- 初始化时：
  程序启动 → 从蓝牙盒子读取零点绝对位置 → 存入全局变量；
- 更新时（`setOrigin()`）：
  - 重新测量当前绝对位置作为新零点；
  - 同步更新蓝牙盒子；
  - 同步更新全局缓存

五、实施建议

1. **数据结构**
   - 定义一个 `MotorState` 类/结构，集中管理：

     ```
     struct MotorState {
         double zeroAbsPos;   // L0
         double relPos;       // L1
         double absPos() const { return zeroAbsPos + relPos; }
     };
     ```

   - 所有逻辑只操作 `MotorState`。
2. **性能优化**
   - 零点只在启动和更新时读取；
   - 运行中不再访问蓝牙盒子
3. **安全性**
   - 限位判断始终使用绝对位置；
   - 只允许修改零点的模块更新全局变量和蓝牙盒子。

六、存疑点

如果现在全局变量`MotorState`保存了电机的位置，并且在进程A中，限制了只有修改零点的模块（修改`zeroAbsPos`）和获取位置的模块（修改`relPos`）可以修改这个位置信息。

那么如果此时另一个进程B也在控制这个电机，那么如果此进程B设置了零点或者移动了电机，那就会导致两个进程缓存的位置信息不一致。从而导致A进程位置信息混乱。如何解决?

> 多个进程各自维护位置缓存（`MotorState`）会产生状态分裂，进而导致限位误判或运动逻辑错误。

这个问题引出来的核心是:

电机状态必须有唯一权威源（source of truth）。

无论是蓝牙盒子还是一个守护进程，都要让所有控制者“读写同一个真值”，否则局部缓存必然分裂。

状态分裂风险

“多进程控制同一物理设备的状态一致性问题”:

> 多个进程各自维护位置缓存（`MotorState`）会产生状态分裂，进而导致限位误判或运动逻辑错误。

分三步分析:
1️⃣ 明确问题本质

2️⃣ 可能的解决架构

3️⃣ 推荐实现方式（可落地）

一、问题本质：状态分裂（State Divergence）

假设:

- 进程 A：主控制界面
  - 持有自己的 `MotorState`（含 `zeroAbsPos`, `relPos`）
- 进程 B：另一控制客户端
  - 同样持有自己的 `MotorState`
  - 也能操作同一个电机（通过 Modbus / 蓝牙盒子）

风险在于:

- B 调用 `setOrigin()` 更新了零点（蓝牙盒子存储变化），但 A 的缓存仍是旧值;
- 或者 B 执行点动 / 位置移动，使电机物理位置变化，但 A 的 `relPos` 未更新;
- A 后续发出的动作基于“旧状态”计算目标位置 → 造成错误动作甚至撞限位。

这就是**分布式状态同步问题**，典型的“多控制源 + 本地缓存”导致的非一致性。

二、集中状态管理解决

让**蓝牙盒子**或**中间服务进程**充当“状态中心”，所有状态读写统一走它。

架构图：

```
+--------------------+
|   蓝牙盒子/状态服务 |
|  - 保存 zeroAbsPos |
|  - 保存 relPos 或 absPos |
|  - 提供读/写接口   |
+---------+----------+
          |
    ---------------------
    |                   |
进程A                 进程B
    |                   |
    ---> 读/写 API <----
```

优点：

- 状态唯一；
- 任意进程操作都会被同步感知；
- 可天然扩展到网络多节点。

缺点：

- 增加了 IPC 或网络访问的延迟；
- 实现稍复杂（需要共享访问机制）。

实现方式：

- 若蓝牙盒子具备写入接口 → 直接让它成为唯一存储源；
- 若蓝牙盒子仅提供只读接口 → 可以在系统中增加一个 **守护进程** `MotorStateService`：
  - 负责与硬件通信；
  - 暴露本地 IPC 接口（如 UNIX domain socket / DBus / gRPC）；
  - 进程 A、B 均通过它进行读写。

三、实践建议

定义一个状态服务 `MotorStateService`：

```cpp
// MotorStateService.h
struct MotorState {
    double zeroAbsPos;
    double relPos;
    double absPos() const { return zeroAbsPos + relPos; }
};

class MotorStateService {
public:
    static MotorStateService& instance();
    MotorState getState();
```

```
    void setZero(double zero);
    void updateRelative(double rel);
private:
    MotorState m_state;
    std::mutex mtx;
};
```

在系统中：

- 进程 A、B 都通过 `MotorStateService` 获取或修改状态；
- 内部由服务统一与蓝牙盒子通信；
- 可扩展为 IPC 服务形式。

> **电机状态必须有唯一权威源（source of truth）。**
> 无论是蓝牙盒子还是一个守护进程，都要让所有控制者“读写同一个真值”，否则局部缓存必然分裂。

四、问题延申

如果采用统一的位置信息状态由蓝牙盒子维护

```
struct MotorState {
    double zeroAbsPos;
    double relPos;
    double absPos() const { return zeroAbsPos + relPos; }
};
```

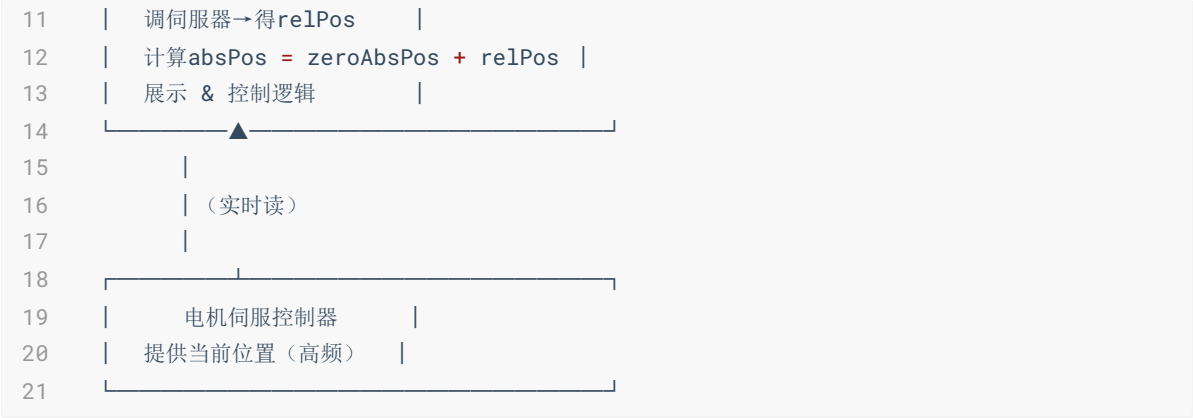
A、B进程在位置移动过程中，需要实时获取当前电机的相对位置。此时进程会调用 `getLocation()` 来获取相对位置信息。此函数是直接通过进程与对应电机的伺服控制器进行通信的，不走蓝牙盒子相关的通讯。

那么当进程实时获取位置信息后，如何修改蓝牙盒子维护的位置信息状态呢？可能的场景：

1. 进程从伺服控制器中获取当前位置，然后将更新的数据写入到蓝牙盒子（进程一收一写，性能开销较大，导致下一次实时展示的位置信息延迟而不够准确 ）
2. 进程不通过伺服控制器直接获取当前位置，直接通过访问蓝牙盒子保存统一的位置信息来获取。蓝牙盒子收到获取当前我位置请求后，需要再向伺服控制器发送获取位置信息的命令。（访问位置信息，需要等待蓝牙盒子和伺服控制器的两次读反应时间，也是不够准确）
3. 延迟写入蓝牙盒子：进程依旧在伺服控制器中获取电机的当前位置，但是需要在移动结束后，将最新的移动数据写入到蓝牙盒子的位置信息中。（感觉不错）

但是其实，有没有一种可能，蓝牙盒子不需要维护当前位置信息？实时位置是高频变化、毫秒级更新的内容，由伺服驱动器负责最合适。 蓝牙盒子其实不适合作为高频同步点。

```
┌──────────────────────────────────┐
│        蓝牙盒子                  │
│   存：zeroAbsPos                 │
│   限位范围、配置等                │
└──────────▲───────────────────────┘
           │
           │（只在设原点时同步一次）
           │
┌──────────┴───────────────────────┐
│       控制进程A/B                 │
```



> **蓝牙盒子不需要维护实时位置信息，只需维护“零点绝对位置”这一静态基准。**
> 所有实时位置信息都以伺服反馈为准，并通过 `absPos = zeroAbsPos + relPos` 统一坐标体系。