控制分组

一、实现目标

> 行车控制划分功能组，每组都有前后左右旋转的控制，根据电机配置打开或者关闭某一种控制。

就是最终能够实现选择组来控制：

1. 选择组1，则调用组1的整套电机
2. 选择组2，则调用组2的整套电机

每组都包含相同功能的电机（上下、前进后退、左右、旋转等）。

二、当前控制现状

就拿当前点动逻辑`actionJog()`进行分析，看下当前进程是如何控制某个特定功能电机进行移动的：

- 获取当前要操作的电机型号，调用了`getMotorID()`。
- `getMotorID()`会根据界面上选择的电机（是旋转还是左右移动的功能来区分的电机），获取到对应枚举`E_MOTOR_ID`中定义的的 ID 号

```
typedef enum {
    M_ALL=-1,
    M_X1 = 0,
    M_X2,
    M_Y1,
    M_Z1,
    M_Angle,
    M_Z2,
    M_Y2,

    M_end
}E_MOTOR_ID;
```

- 获取电机配置信息：对应当前定义的`motorID`，匹配数据库中唯一`id`对应的电机参数信息（当前就是`id = motorID` ）：

```
id, name, model, modbusID, linkStat, outsideDiameter, gearRatio,
screwPitch, minLimit, maxLimit, speed, clickDistance, currLocation,
position, dir, isEnable
```

获取信息的方法：

```
float screwPitch = DbCtrl::m_servoMotor_tb.at(motorID).screwPitch
```

- 伺服控制modubs：伺服控制器对应每一个电机都有唯一的`ModbusID`，通过`motorID`与之一一对应的关系，配置对应伺服控制器的寄存器。

这样就实现了根据界面选中的电机，最终能够控制对应的数据库配置以及伺服控制器使之移动。

```
界面选中电机 - 获取电机motorID - 预先配置好的数据库 + 预先配置好的modbusID
```

三、实现思路

其实最终就是实现，选择组0的`M_Angle`时候，就会通过查询配置项的方式，找到对应的数据库`id`以及`modbusID`。组1亦然。

笼统思路：

1. 维持`E_MOTOR_ID`的功能配置，省略其对应的ID配置。例如`M_Angle`就是单独控制旋转的电机功能
2. 分组控制需要加入一个标志位。此标志位记录了当前选择的电机组。例如
   - `groupID = 0`：控制组0
   - `groupID = 1`：控制组1
3. 获取数据库主键`id`的方法就由`E_MOTOR_ID`功能项和分组标志`groupID`共同实现。
4. 获取`modbusID`的方法，也是由`E_MOTOR_ID`功能项和分组标志`groupID`共同实现。
5. 按照`motorID`对应的`modbusID`配置即可（`PA71 - modusID`范围是1-254）

现在存在的问题是：

1. `motorID`与数据库电机信息主键`id`深度绑定

   例如获取某个电机的丝杆螺距：

   ```
   DbCtrl::m_servoMotor_tb.at(motorID).screwPitch
   ```

   问题：motorID直接作为数据库索引

2. `motorID`与电机的`modbusID`深度绑定

   例如向`modbusID`的电机写数据操作：

   ```
   MotorRegs::get_instance()->writeReg16(motorID, MotorRegs::H06_04,
   JOG_speed);
   ```

   问题：motorID直接作为Modbus地址

如何解除`motorID`与数据库主键`id`和`modbusID`的强绑定？那就不用解除啊，直接写死就完了。

控制分组2

现在需要重新审视下这个分组问题了。之前试图解除`motorID`的思路过于麻烦，现在重新梳理下我所想实现的方案。

理论方案

1. 保留`E_MOTOR_ID`作为功能枚举：
   - 保持当前枚举类的特性：作为一套电机不同功能划分
2. 划分真正的`motorID`枚举组，例如起名为`MODBUS_ID`
   - 第一套就是`A_X1 = 1`、`A_X2 =2`、...、`A_Z2`对应的就是之前`E_MOTOR_ID`的`M_X1`等
   - 第二套就是`B_X1 = 11`等，代表第二套`motorID`
3. 设置`groupID`
   - 界面创建复选框，选择对应的组，相当于设置了对应的`groupID`

4. 实现电机功能选择 `E_MOTOR_ID` 和 `groupID` 共同决定的 `MODBUS_ID` 的函数
    - 用户选择了对应电机组以及对应的电机后，调用此函数会获取到对应 `MODBUS_ID` 下的 `motorID`

      例如对应关系为：

      ```
      MODBUS_ID = groupID * 10 + E_MOTOR_ID
      ```

5. 保持现在的数据库读取以及伺服控制函数的原型即可。
6. 数据库加入新的 `MODBUS_ID` 对应的主键以及各项配置
7. 配置第二套电机的 `modbusID` 为 `MODBUS_ID` 定义的编号

您的思路更清晰且更实用，我来帮您完善这个方案。这个方案确实更简洁，改动量也更小。

方案完善

一、理论方案完善

1. 保留现有 `E_MOTOR_ID` 功能枚举

```
typedef enum {
    M_ALL = -1,
    M_X1 = 0,       // 前后移动左
    M_X2,           // 前后移动右
    M_Y1,           // 左右移动上轴
    M_Z1,           // 垂直上下
    M_Angle,        // 旋转
    M_Z2,           // 固定位上下
    M_Y2,           // 左右移动下轴

    M_end
} E_MOTOR_ID;
```

2. 定义新的 `MODBUS_ID` 枚举组

```
// 第一套电机组 (组0)
typedef enum {
    A_X1 = 1,       // Modbus地址1
    A_X2 = 2,       // Modbus地址2
    A_Y1 = 3,       // Modbus地址3
    A_Z1 = 4,       // Modbus地址4
    A_Angle = 5,    // Modbus地址5
    A_Z2 = 6,       // Modbus地址6
    A_Y2 = 7,       // Modbus地址7

    GROUP_A_END
} MODBUS_ID_GROUP_A;

// 第二套电机组 (组1)
typedef enum {
    B_X1 = 11,      // Modbus地址11
    B_X2 = 12,      // Modbus地址12
```

```
    B_Y1 = 13,      // Modbus地址13
    B_Z1 = 14,      // Modbus地址14
    B_Angle = 15,   // Modbus地址15
    B_Z2 = 16,      // Modbus地址16
    B_Y2 = 17,      // Modbus地址17

    GROUP_B_END
} MODBUS_ID_GROUP_B;
```

3. 分组映射函数

```
class GroupManager {
private:
    static int m_current_group; // 0=组A, 1=组B

public:
    static void setCurrentGroup(int group_id) {
        m_current_group = group_id;
    }

    static int getCurrentGroup() {
        return m_current_group;
    }

    // 核心映射函数：根据功能+组号 -> MODBUS_ID
    static int getModbusID(E_MOTOR_ID motor_func) {
        if (m_current_group == 0) {
            // 组A映射
            switch(motor_func) {
                case M_X1: return A_X1;
                case M_X2: return A_X2;
                case M_Y1: return A_Y1;
                case M_Z1: return A_Z1;
                case M_Angle: return A_Angle;
                case M_Z2: return A_Z2;
                case M_Y2: return A_Y2;
                default: return -1;
            }
        } else {
            // 组B映射
            switch(motor_func) {
                case M_X1: return B_X1;
                case M_X2: return B_X2;
                case M_Y1: return B_Y1;
                case M_Z1: return B_Z1;
                case M_Angle: return B_Angle;
                case M_Z2: return B_Z2;
                case M_Y2: return B_Y2;
                default: return -1;
            }
        }
    }

    // 反向映射：MODBUS_ID -> 数据库主键id
    static int getDbId(int modbus_id) {
```

```
        // 这里需要根据您的数据库设计来映射
        // 假设数据库主键id与MODBUS_ID有对应关系
        return modbus_id; // 或者通过查找表
    }
};
```

二、数据库设计修改

1. 扩展数据库表结构

```
-- 保持现有表结构，但需要为第二组电机添加记录
-- 现有记录对应组A (modbusID 1-7)
-- 新增记录对应组B (modbusID 11-17)

-- 为组B插入电机配置记录
INSERT INTO servoMotor_tb (name, model, modbusID, linkStat, outsideDiameter,
                           gearRatio, screwPitch, minLimit, maxLimit, speed,
                           clickDistance, currLocation, position, dir,
isEnable)
SELECT
    -- 名称可以加后缀区分
    name || '_B' as name,
    model,
    -- Modbus地址+10
    modbusID + 10 as modbusID,
    linkStat,
    outsideDiameter,
    gearRatio,
    screwPitch,
    minLimit,
    maxLimit,
    speed,
    clickDistance,
    currLocation,
    position,
    dir,
    isEnable
FROM servoMotor_tb
WHERE modbusID BETWEEN 1 AND 7;
```

三、代码修改方案

1. 修改点动控制函数

```
void actionJog(E_MOTOR_ID motor_func, JogDirection dir) {
    // 1. 获取当前组的MODBUS_ID
    int modbus_id = GroupManager::getModbusID(motor_func);

    // 2. 根据MODBUS_ID获取数据库主键id
    int db_id = GroupManager::getDbId(modbus_id);

    // 3. 获取电机配置（保持现有逻辑，但使用新的db_id）
    // 这里需要修改DbCtrl的接口，支持通过db_id获取配置
```

```
    T_servoMotor_tb motor_config = DbCtrl::getMotorConfigById(db_id);

    // 4. 使用配置参数
    float screwPitch = motor_config.screwPitch;
    float speed = motor_config.speed;

    // 5. Modbus通信（使用新的modbus_id）
    MotorRegs::get_instance()->writeReg16(modbus_id, MotorRegs::H06_04,
JOG_speed);

    // 6. 其他原有逻辑...
}
```

2. 修改数据库访问接口

```
// 在DbCtrl类中添加新方法
class DbCtrl {
public:
    // 新增：根据数据库主键id获取电机配置
    static T_servoMotor_tb getMotorConfigById(int db_id) {
        for (const auto& motor : m_servoMotor_tb) {
            if (motor.id == db_id) {
                return motor;
            }
        }
        return T_servoMotor_tb(); // 返回空配置
    }

    // 保持原有的通过索引访问的方法（向后兼容）
    static T_servoMotor_tb getMotorConfigByIndex(int index) {
        if (index >= 0 && index < m_servoMotor_tb.size()) {
            return m_servoMotor_tb.at(index);
        }
        return T_servoMotor_tb();
    }
};
```

3. 界面组选择实现

```
// 在主界面添加组选择控件
QComboBox* group_combo = new QComboBox();
group_combo->addItem("组A", 0);
group_combo->addItem("组B", 1);

// 组切换信号
connect(group_combo, QOverload<int>::of(&QComboBox::currentIndexChanged),
        [this](int index) {
    int group_id = group_combo->itemData(index).toInt();
    GroupManager::setCurrentGroup(group_id);
    // 可以在这里更新界面状态
    updateUIForCurrentGroup();
});
```

四、具体实施步骤

阶段1：数据库准备

1. 执行SQL脚本，为组B插入电机配置记录
2. 验证两组数据都正确导入

阶段2：核心代码修改

1. 实现`GroupManager`类
2. 在`DbCtrl`中添加`getMotorConfigById`方法
3. 修改`actionJog`等核心控制函数

阶段3：界面修改

1. 添加组选择控件
2. 连接组切换信号

阶段4：测试验证

1. 测试组A功能是否正常
2. 测试组B功能是否正常
3. 测试组切换是否流畅

五、优势分析

这个方案的优势：

1. **改动量小**：保持现有`E_MOTOR_ID`枚举，不需要大规模重构
2. **逻辑清晰**：`功能 + 组号 -> MODBUS_ID`的映射关系明确
3. **数据库兼容**：通过插入新记录支持第二组，不影响现有数据
4. **易于扩展**：如果需要第三组，只需定义`MODBUS_ID_GROUP_C`并插入数据
5. **风险可控**：可以分阶段实施，逐步验证

六、注意事项

1. **数据库主键**：确保`servoMotor_tb`的`id`字段能够容纳新的MODBUS_ID值
2. **Modbus地址冲突**：确保两组电机的Modbus地址不冲突
3. **初始化顺序**：确保`GroupManager`在控制函数调用前已初始化
4. **错误处理**：在映射函数中添加错误检查，返回无效值时给出明确提示

这个方案既满足了分组控制的需求，又最大限度地减少了代码修改量，是一个务实且可行的解决方案。