

# Human Resource Machine

## 分工信息

许越 2023010464 核心处理逻辑编写、项目架构与设计、GUI设计与编写

王子轩 2023011307 逻辑测试、GUI逻辑编写、动画编写、项目报告文档撰写

项目编译须知:

- qt6.7.2

可执行文件:

- bin/bin/human-resource-machine.exe

## 设计思路

游戏核心功能是模拟机器人系统执行用户编写的指令。基于 Qt6 框架并采用面向对象方法，利用信号与槽机制实现较为便捷的界面交互，将核心逻辑部分和GUI解耦。

在GUI界面的实际运行中，首先 `ROBOTSYSTEM` 类在初始化时接收一个 `LEVEL` 对象，该对象包含了当前关卡的详细信息；然后实现接收用户编写的指令的接口并实现 `execution` 函数负责执行用户编写的指令，依次调用每个指令对应的函数。`execution` 函数会返回一个整数p，表示游戏的执行结果。如果返回值为0，表示游戏成功；如果返回值为小的正数，表示程序输出不合法，或者程序发生小的错误；如果返回值为大的正数，表示程序陷入了死循环；如果返回值为负数，表示在对应的指令上发生了不可运行的错误。如果返回值不是负数，那么可以加载动画，展示机器人运行的过程；运行动画完成后，会再用 `ROBOTSYSTEM` 类跑几个样例，确保用户没有“偷鸡”。在上述结束后，`QMainGame` 类会重置游戏并重新初始化游戏界面，同时记录、保存玩家游玩到的关卡。

## 项目结构

### 1.文件目录

使用 Visual Studio 2022 进行项目构建，GUI基于Qt框架，项目结构如下：

```
human-resource-machine/
|
├── Header Files/
│   ├── humanresourcemachine.h
│   ├── input_and_output.h
│   ├── progress.h
│   └── robot_system_generation.h
|
├── Source Files/
│   ├── BasicFunctions
│   │   ├── main.cpp
│   │   ├── input_and_output.h
│   │   ├── input_and_output.cpp
│   │   ├── robot_system_generation.cpp
│   │   └── robot_system_generation.h
│   ├── background_widget.cpp
│   ├── current_line_cursor.cpp
│   ├── dialog_pause.cpp
│   ├── dlg_spare.cpp
│   └── drag_label.cpp
```

```

|   └── drag_list_events.cpp
|   └── humanresourcemachine.cpp
|   └── lt_init.cpp
|   └── main.cpp
|   └── main_game_on_pause.cpp
|   └── main_game_robot_execute.cpp
|   └── main_game_robot_move.cpp
|   └── robot_system_generation.cpp
|   └── show_level_tip.cpp
|   └── ui_level_select_init.cpp
|   └── ui_main_game_init.cpp
|   └── ui_start_menu_init.cpp
|
└── Resource Files/
    ├── md5_generator.py
    ├── img
    │   ├── game_background.png
    │   ├── level_select_background.png
    │   ├── pause.png
    │   ├── robot.png
    │   └── start_menu_background.png
    ├── levels
    │   ├── 1.dat
    │   ├── 1_2.dat
    │   ├── 1_3.dat
    │   └── ...
    ├── log
    │   └── player_progress.json
    └── test
        ├── 0.in
        ├── 0.out
        └── ...

```

oj版本main.cpp:

首先在函数 `input(string, int, int, string[])` 读取用户通过文件进行的指令输入。初始化关卡LEVEL类和机器人运行系统ROBOTSYSTEM类，通过用户输入的关卡编号，调用LEVEL类的构造函数，进入levels文件夹进行关卡读取，存储到level后再将读取到的关卡通过ROBOTSYSTEM的构造函数写进robot\_system中，接下来调用ROBOTSYSTEM::instructions\_input(int, string[])将用户的指令写入。完成了输入，调用ROBOTSYSTEM::execution()进行执行，返回0表示成功；返回-x表示在第x行执行时出错，或者用户输入指令为空；返回x表示失败，且运行时执行了x行代码，最后在主函数判断返回值并输出对应结果。

## 2. 模块详细介绍

### `robot_system_generation.cpp` (核心逻辑)

#### 常量

- `kMAX_BOX_NUMBER`：输入的箱子的最多个数。
- `kMAX_SPARE_NUMBER`：最大空地数量。
- `kMAX_NUMBER_OF_AVAILABLE_INSTRUCTIONS`：最多可用指令。
- `kMAX_NUMBER_OF_LEVEL`：关卡数量。
- `kMAX_INSTRUCTION_LENGTH`：单条指令的最大长度。

## LEVEL类

用于表示游戏的一个关卡，其中包含了积木的初始状态、目标输出、空地数量以及可用的指令集。它有两组构造函数，一个用于从给定的数组初始化关卡数据，另一个用于从文件中读取关卡数据。`LEVEL` 类提供了获取关卡信息的公有方法，如 `get_box_number()`、`get_boxes(int index)`、`get_aim_out_number()`、`get_aim_outs(int index)`、`get_spare_number()`、`get_available_instructions(int index)` 和 `get_available_instruction_number()`。这些方法让其他程序允许访问且不能修改关卡信息，并根据这些信息执行计算的操作。同时，将 `ROBOTSYS` 类设置为友元类，方便其操作。

**变量：**

- `int boxes[]`: 输入箱子的值，这里为了读取方便，存储是反过来的
- `int box_number`: 输入箱子的个数
- `int aim_outs`: 预期输出的箱子
- `int aim_out_number`: 预期输出的箱子个数
- `int spare_number`: 可用空地数量
- `string available_instructions[]`: 可用指令
- `int available_instruction_number`: 可用指令数量

**方法：**

- `LEVEL(const int boxes_input[], int box_number_input, const int aim_outs_input[], int aim_out_number_input, int spare_number_input, const string available_instructions_input[], int available_instruction_number)`: 将关卡信息初始化。
- `LEVEL(string level_path)`: 读取文件，利用关卡文件初始化关卡。
- `int get_box_number()`: 获取box\_number。
- `int get_boxes(int index)`: 获取boxes[index]。
- `int get_aim_out_number()`: 获取aim\_out\_number。
- `int get_aim_outs(int index)` 获取aim\_outs[index]。
- `int get_spare_number()`: 获取spare\_number。
- `string get_available_instructions(int index)`: 获取available\_instructions[index]。
- `int get_available_instruction_number()`: 获取available\_instruction\_number。

## ROBOTSYS类

模拟机器人执行玩家编写的指令。它的构造函数接收一个 `LEVEL` 对象，根据这个对象初始化机器人系统的状态。`instructions_input` 方法用于接收并解析玩家输入的指令，将它们转换为对应的操作函数和操作数。`execution` 方法是游戏的主执行循环，它循环执行玩家编写的指令，并根据执行结果返回相应的状态码。`ROBOTSYS` 类还定义指令方法，如 `inbox(int)`、`outbox(int)`、`add(int)`、`sub(int)`、`copyto(int)` 和 `copyfrom(int)`，这些方法对应游戏中可以执行的基本操作。

## 变量

- `LEVEL* level`: 存储关卡。
- `int spares[]`: 空地的数值。
- `bool used_spares[]`: 空地是否被占用。
- `int outs[]`: 输出的盒子的值。
- `int out_number`: 输出的盒子数量。
- `int (ROBOTSYS::*instructions[])(int)`: 输入的指令作为函数数组的存储。
- `int instruction_operators[]`: 每个指令对应一个指令数，如jump的指令数是所要跳转的行数，如果指令本身没有指令数，那就记作-1。
- `int instruction_number`: 指令数量。
- `int robot_number`: 机器人头顶存储的数字。
- `bool is_robot_has_number`: 机器人头顶是否存储数值。
- `int execute_number`: 指令执行到的位置。由于有很多跳转逻辑，因此使用一个变量专门存储指令执行到哪里。

## 方法

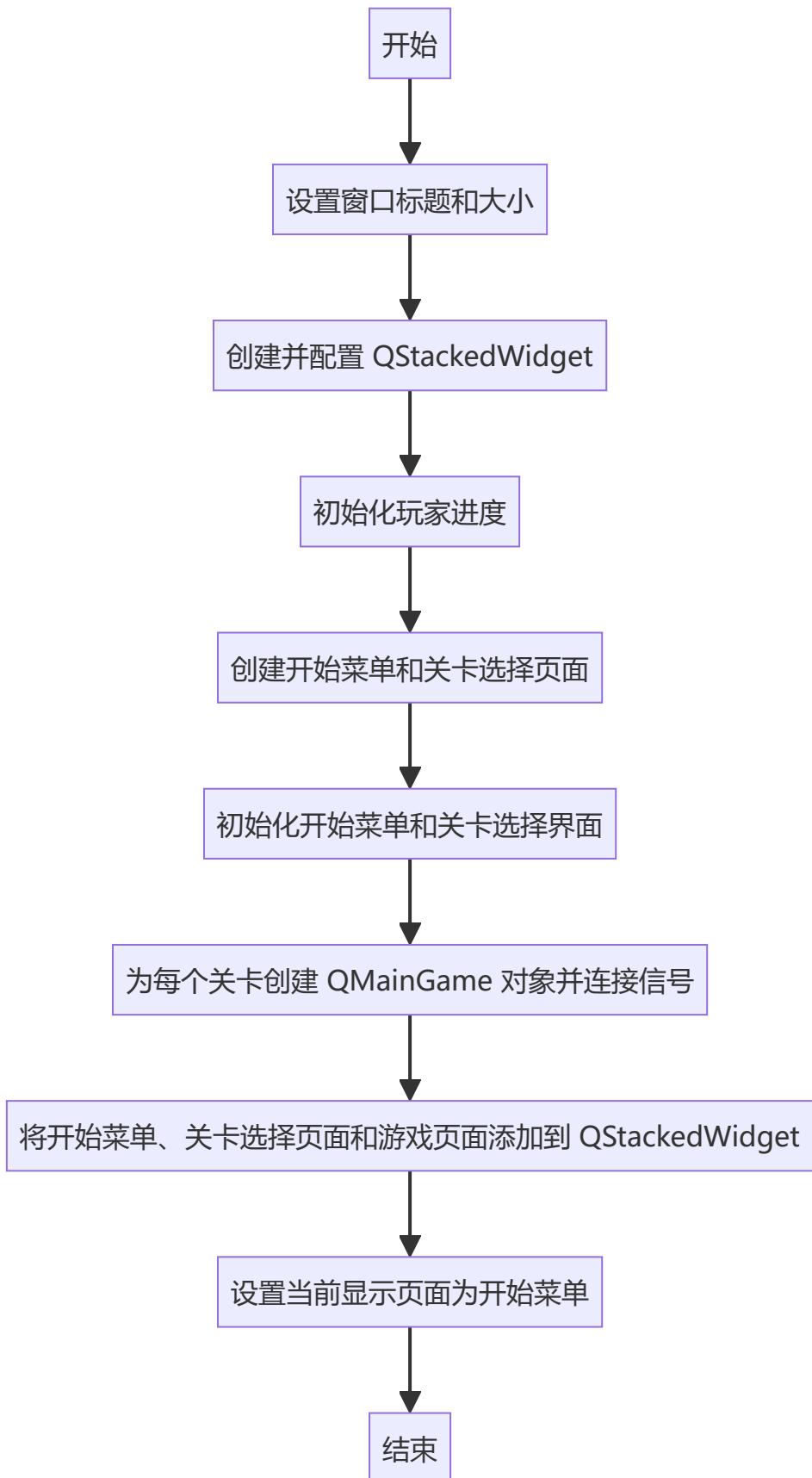
- `ROBOTSYS(LEVEL* l)`: 将关卡写入运行系统。
- `~ROBOTSYS()`: 一次性释放自己和它的关卡的空间，防止内存泄漏。
- `int inbox(int)`: 读取输入的盒子的值，默认输入inbox的参数是-1。这里规定，指令能执行且程序不中止返回1，指令能执行且程序中止（在inbox中是输入盒子已经读完）返回0，指令不能执行返回-1，下同。
- `int outbox(int)`: 将机器人头上的盒子输出到outs里。
- `int add(int)`: 将机器人头上的值修改为`robot_number+spares[op]`，其中op是指令数，下同。
- `int sub(int)`: 与add相似，相加变成相减。
- `int copyto(int)`: 将机器人头上的值赋给空地。
- `int copyfrom(int)`: 将空地的值赋给机器人。
- `int jump(int)`: 跳到第op行指令。
- `int jumpifzero(int)`: 如果机器人头上的值不是0则不跳转，否则跳转到第op行指令。这里根据题意，不管跳转是否发生，如果第op行指令不存在，直接返回-1。
- `void instructions_input(int, string[])`: 将用户输入的指令转为`int (ROBOTSYS::*)(int)`函数类型，存储到instructions函数数组里。
- `int execution()`: 运行机器人系统并返回结果。首先将所有变量全部初始化。之后开始执行运算，按指令的跳转逻辑顺序执行指令，并累加指令运行次数`execute_count`，运行时判断输出是否符合预期，若不符合则结束运行并返回+x，x是`execute_count+1`。运行结束后判断是中止、异常还是正常完成，严重异常则返回-x，x的值表示错误指令的位置（代码从1开始编号），普通异常则返回+x，x就是`execute_count+1`；中止则判断是否与`aim_outs`目标输出一致，一致则返回0，否则返回+x，含义相同。

## humanresourcemachine类

`humanresourcemachine.h` 中定义了GUI主要的类；`humanresourcemachine.cpp` 文件是 `human-resource-machine` 游戏的图形用户界面（GUI）逻辑的入口和实现部分。是游戏的主窗口类，继承自 `QMainWindow`。它包含了游戏的主界面布局、进度管理、关卡选择和游戏执行。

- 构造函数 `humanresourcemachine(QWidget* parent)`：初始化游戏主窗口，设置窗口标题和大小，创建并配置主界面布局和控件。
- 析构函数 `~humanresourcemachine()`：清理资源，如删除进度管理对象。
- `ui_start_menu_init()`：初始化开始菜单界面。
- `ui_level_select_init()`：初始化关卡选择界面。
- `start_game()`：开始新游戏。
- `exit_game()`：退出游戏。
- `start_game_from_continue()`：从上次进度继续游戏。
- `enter_level(int level_num)`：进入指定关卡。
- `exit_level()`：退出当前关卡。
- `show_level_tip(int level_num)`：显示关卡提示。
- `on_game_exit(int level_num, bool completed)`：处理游戏退出事件。
- `update_progress(int p, int level_num)`：更新玩家进度。
- `ui_level_select_re_init()`：重新初始化关卡选择界面。

## 窗口跳转流程关系



## LevelInfo 结构体

实际编程中，我们发现直接将level转成结构体存储会方便很多，所以我们又编写了这样的结构体。

- `int level_num`: 关卡号;
- `int box_number`: 积木当前数量;
- `vector<int> boxes`: 积木;
- `int aim_out_number`: 目标输出积木数;
- `vector<int> aim_outs`: 目标输出;
- `int spare_number`: 可用空地数;
- `int available_instruction_number`: 可用指令数量;
- `vector<string> available_instructions`: 可用指令。

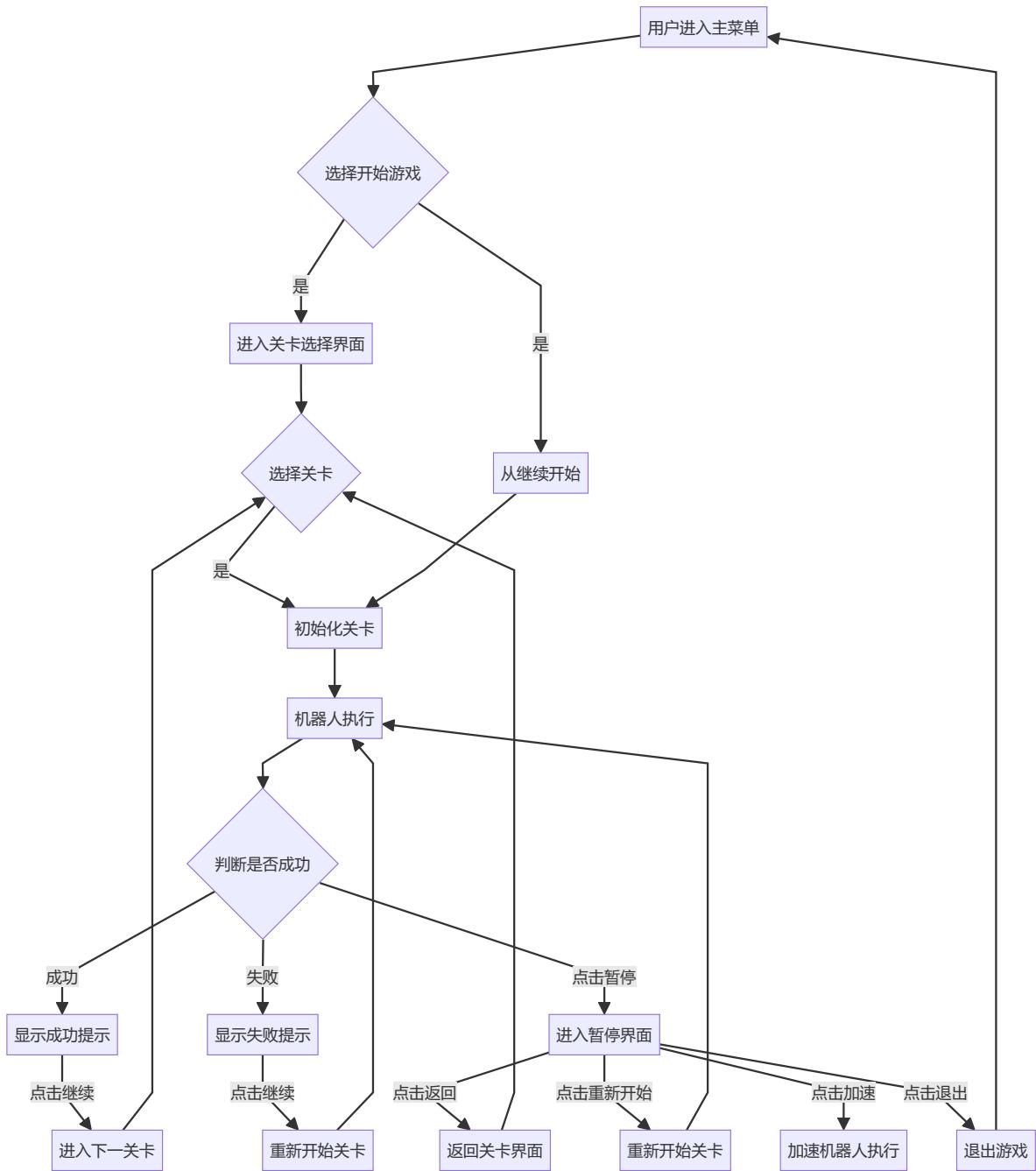
## QMainGame 类

这个类表示游戏的主要游戏界面，包含了一个关卡的所有元素，如输入输出箱、代码区域、机器人状态等。主要函数包括：

- 构造函数 `QMainGame(int level_index, QWidget* parent)`: 初始化游戏界面，根据关卡索引创建关卡和机器人系统。
- 各种布局初始化函数，如 `lt_inbox_init`、`lt_outbox_init`、`lt_robot_n_spare_init`、`lt_code_choice_init`、`lt_code_init`、`lt_robot_init`、`lt_space_init` 和 `lt_pause_init`。
- `robot_execute()`: 执行玩家编写的代码。
- 信号槽函数，如 `on_fail_exe`、`on_inbox_exe`、`on_outbox_exe`、`on_add_exe`、`on_sub_exe`、`on_copyto_exe`、`on_copyfrom_exe`、`on_btn_pause_clicked`、`on_btn_back_clicked`、`on_btn_restart_clicked`、`on_btn_accel_clicked` 和 `on_btn_exit_clicked`。

该类中的方法实现在文件

`lt_init.cpp`, `main_game_on_pause.cpp`, `main_game_robot_execute.cpp`, `main_game_robot_move.cpp` 中



## GUI模块

### QDialogSpare 类

这个类是游戏中选择空地的弹窗，继承自 `QDialog` 类。我们采用简单的方法进行空地的选择，也就是弹窗的方式，用户点击弹窗中对应的按钮就可以选中空地。它的方法实现在 `dlg_spare.cpp` 中。

- `QDialogSpare(int spares_number)`: 构造函数，根据空地数构造一个有 `spares_number` 个按钮的弹窗。
- `int get_spare_no()`: 获取用户选中的空地编号。
- `void onbuttonclicked()`: 槽函数，按钮按下时记录按下的编号，存到 `spare_no` 中。

## BackgroundWidget 类

这个类是游戏中的机器人，继承自 `QWidget` 类。就是给机器人设置背景图片。它的方法实现在 `background_widget.cpp` 中。

- `BackgroundWidget(string img_path, QWidget* parent = nullptr)`: 构造函数，根据文件路径获取图片，存到 `backgroundImage` 中。
- `void setBackgroundImage(const QString& imagePath)`: 将获取到的图片设置为机器人的背景图。
- `void paintEvent(QPaintEvent* event)`: 重载事件响应，使 UI 刷新时按照这里的指示进行绘制。这里的指示就是按照背景图绘制。

## QDragListWidget 类

这个类是让用户通过拖动输入代码的代码框，继承自 `QListWidget` 类。它的方法实现在 `drag_list_event.cpp` 中。

- `QDragListWidget(int level_spare_number);`: 构造函数，进行初始化，并记录空地数量（方便弹窗），同时将点击其中项目的事件与 `onItemClicked` 槽函数连接起来。
- `void dragEnterEvent(QDragEnterEvent* event)`: 重载拖动进入事件响应，允许拖动进入。
- `void dragMoveEvent(QDragMoveEvent* event)`: 重载拖动移动事件响应，允许拖动移动。
- `void dropEvent(QDropEvent* event)`: 重载拖动放置事件响应。这里按照拖动时的 `mimeData` 分两类，从外部（也就是提供的 `available instructions`）进入的需要先判断是否为 `add/sub/copyfrom/copyto`，若是则要弹窗让用户选择空地；同时判断是否为 `jump/jumpifzero`，若是则要添加空白条，并在 `friends` 中登记它们，从而在后续提到的 `paintEvent` 中加载箭头；从内部拖动的不需要这些判断。最后根据拖动结束时鼠标光标的位置得到这些 `item` 的放置位置，并放置。
- `void mouseMoveEvent(QMouseEvent* event)`: 重载内部的 item 移动事件响应，为它们的 `mimeData` 加上前缀 `drag_list` 从而可以识别为内部拖动；移动时鼠标光标变成移动的指令的名称。
- `void keyPressEvent(QKeyEvent* event)`: 重载按键事件响应，在按下 `delete` 键时能够删去这一项（若是 `jump/jumpifzero` 则删去两项，并在 `friends` 中移除）。
- `void dragLeaveEvent(QDragLeaveEvent* event)`: 重载拖动离开事件响应，在拖动着 item 离开时，item 会被 list 删去（同样判断 `jump/jumpifzero`）。
- `void paintEvent(QPaintEvent* event)`: 重载绘制事件，除了正常绘制外，再加上 `jump/jumpifzero` 的箭头。
- `vector<pair< QListWidgetItem*, QListWidgetItem*> > get_friends()`: 获取 `friends`。
- `void onItemClicked(QListWidgetItem* new_item)`: 槽函数，允许用户通过单击 item 来修改 `add/sub/copyfrom/copyto` 指向的空地编号。
- `void onLabelDragged()`: 槽函数，当外部元素拖动时自己的 `currentItem` 设为空，防止 `dragLeaveEvent` 响应而误删。
- `void onBtnRestart_clicked()`: 槽函数，当暂停界面的 `clear/restart` 按钮按下时清空自己的 list。

## QDragLabel 类

这个类是能拖动的标签，继承自 `QLabel`。它的方法实现在 `drag_label.cpp` 中。

- `QDragLabel(QWidget* parent = nullptr)`: 构造函数，设置它可以被拖动。

- `void mousePressEvent(QMouseEvent* event)`：重载鼠标按下事件，也就是拖动的前奏。准备好mimeData（用`outside`标记），绘制光标，并发射拖动信号`labelDragged`，提醒`QDragListWidget`调用`onLabelDragged`。
- `void mouseMoveEvent(QMouseEvent* event)`：重载鼠标移动事件，完成拖动。

## GUI窗口初始化

下列三个文件通过定义 `humanresourcemachine` 类和 `QMainGame` 类的函数，实现了游戏用户界面的初始化。它们通过创建和配置各种界面元素，如布局、按钮和标签，以及连接信号和槽，共同构建了游戏的用户界面和交互逻辑。

### `ui_level_select_init.cpp`

实现 `humanresourcemachine` 类中初始化关卡选择界面的函数 `ui_level_select_init()`。

这个函数负责创建关卡选择界面，包括背景图片、关卡按钮和返回按钮。

`i_level_select_init()` \*\*函数创建一个垂直布局 `ui_level_select`，然后加载背景图片并添加到布局中。接着，它创建一个水平布局 `buttonLayout` 来放置关卡按钮和返回按钮。对于每个关卡，它创建一个按钮，并根据玩家的进度设置按钮的样式和状态（启用或禁用）。最后，它将按钮布局添加到主布局中，并设置布局到关卡选择页面。`enter_level(int level_index)` 函数处理玩家进入关卡的事件。它首先显示关卡的提示信息，然后根据关卡索引设置 `stackedWidget` 的当前页面为对应的关卡页面。`exit_level()`：这个函数处理玩家退出关卡选择界面的事件。它将 `stackedWidget` 的当前页面设置为开始菜单页面。

### `ui_start_menu_init.cpp`

实现 `humanresourcemachine` 类中初始化开始菜单界面的函数 `ui_start_menu_init()`。这个函数负责创建开始菜单界面，包括背景图片和三个按钮：开始、继续和退出。

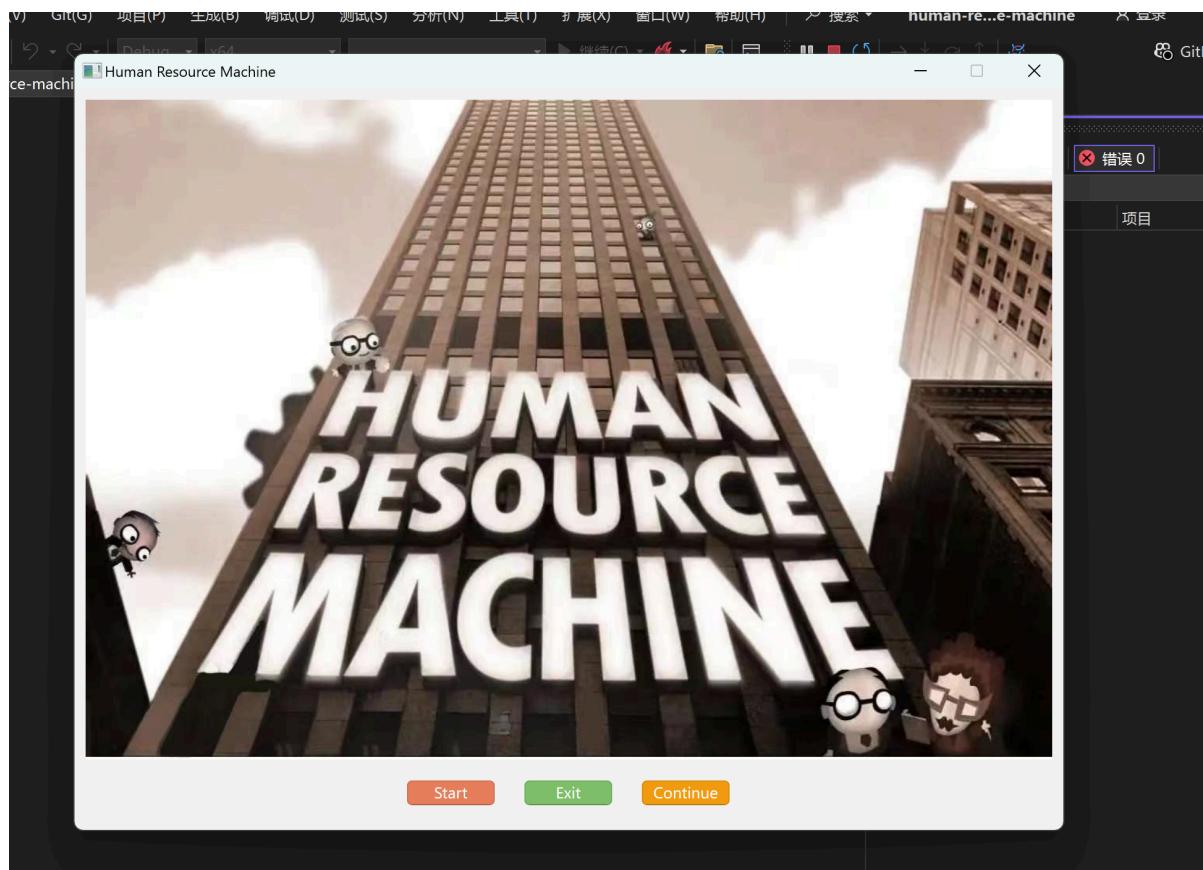
`ui_start_menu_init()` 函数创建一个垂直布局 `ui_start_menu`，然后加载背景图片并添加到布局中。接着，它创建一个水平布局 `buttonLayout` 来放置三个按钮。每个按钮都有特定的样式和点击事件处理函数。最后，它将按钮布局添加到主布局中，并设置布局到开始菜单页面。`start_game()` 函数处理玩家点击开始按钮的事件。它将 `stackedWidget` 的当前页面设置为关卡选择页面。`start_game_from_continue()` 函数处理玩家点击继续按钮的事件。如果玩家已经完成了所有关卡，它会显示一个消息框并跳转到关卡选择页面；否则，它会跳转到玩家上次未完成的关卡。`exit_game()` 这个函数处理玩家点击退出按钮的事件。它会弹出一个确认对话框，如果玩家确认退出，它会保存玩家的进度并退出应用程序。

### `ui_main_game_init.cpp`

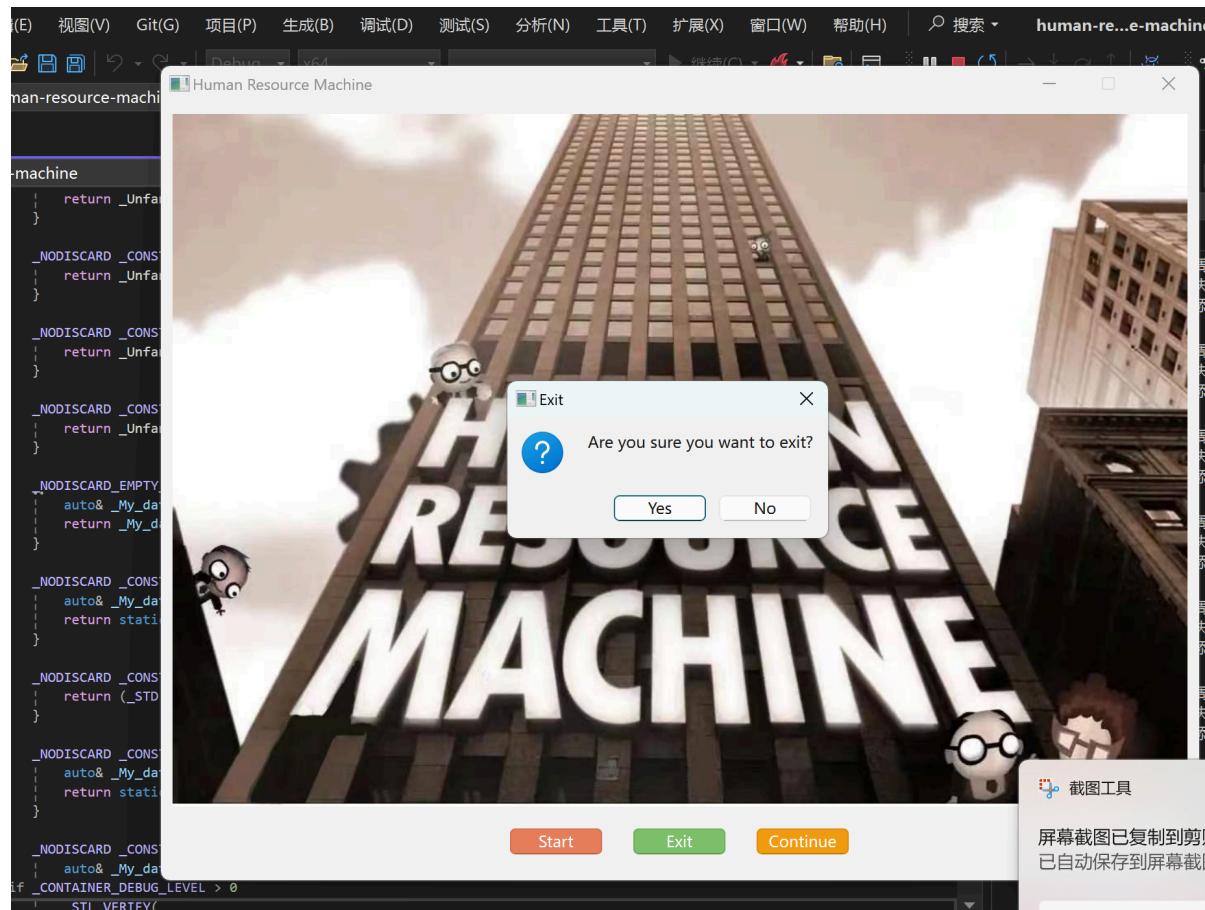
包含 `QMainGame` 类的构造函数，它负责初始化主游戏界面。`QMainGame(int level_index, QWidget* parent)` 构造函数首先根据关卡索引创建关卡和机器人系统，然后创建一个水平布局 `lt_game` 并设置为主布局。然后调用初始化函数来创建和配置游戏界面的各个部分，如输入箱、输出箱、机器人状态、代码选择区域和代码区域。最后将这些部分添加到主布局中，并设置主布局到游戏页面。

## 整体界面设计

### 菜单界面



启动游戏，进入到如图菜单界面，如图所示有三种选择：start、exit、continue，分别对应于进入关卡选择、离开游戏、继续上一轮游戏。实现了通过文件保存玩家攻关进度，再次进入游戏时候自动加载的功能。按键设计为当鼠标出于键上时候按键变色便于；玩家点击start按键即可进入选择关卡界面。若点击exit，系统会弹出确认窗口并保存用户进度。

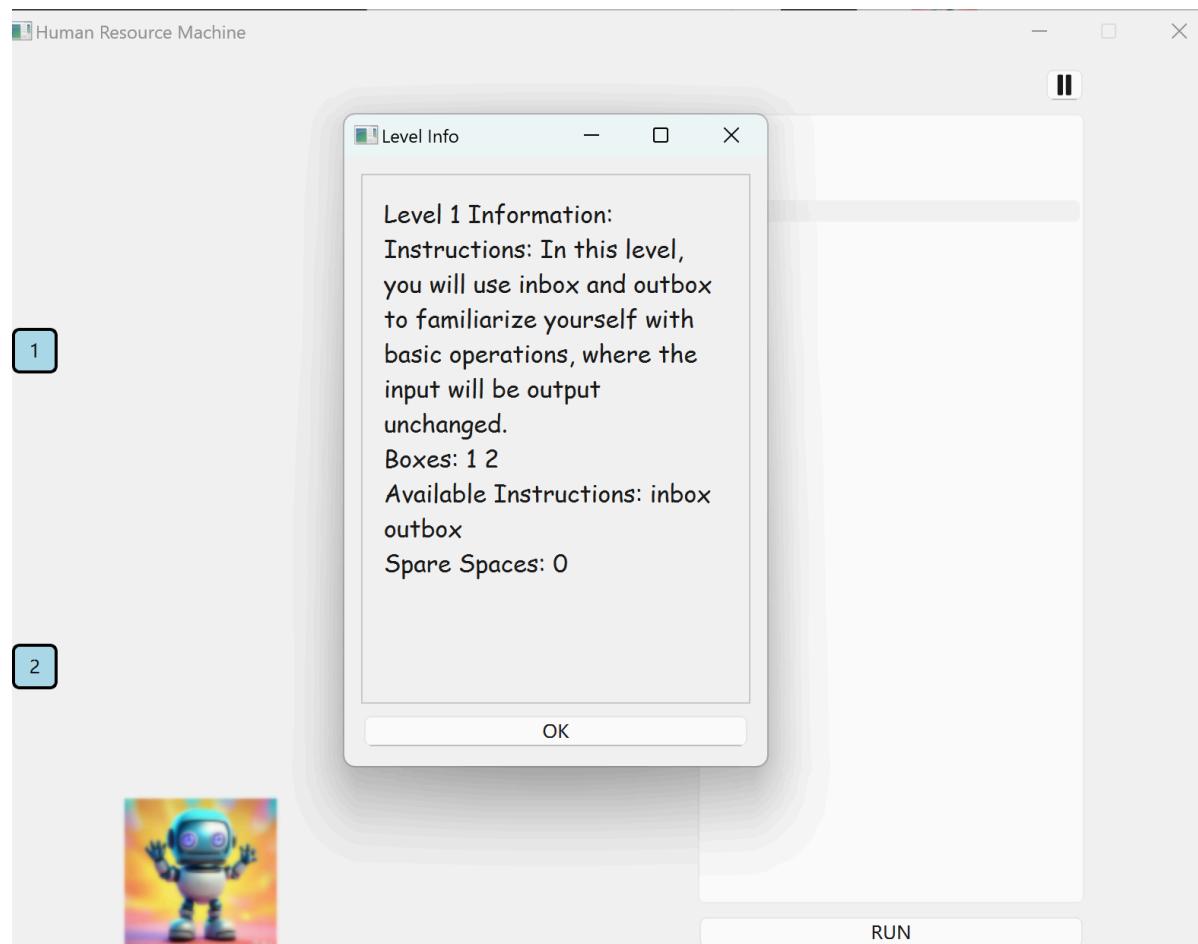


## 关卡选择

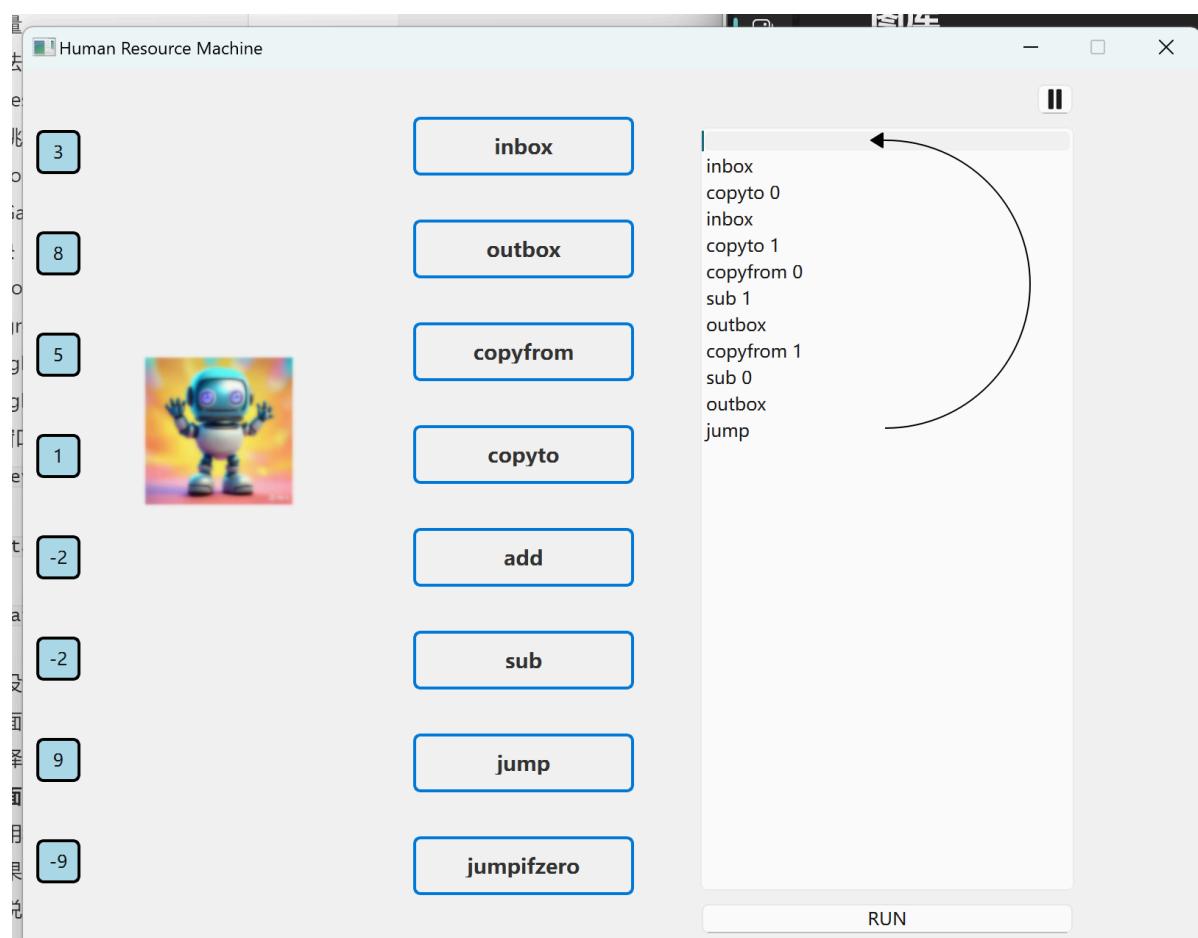


游戏共有4个关卡，如图所示，我们复原了游戏中的地图，通过PS提示玩家该关卡主题分别是“复印（即原样输出）”、“左右相减”、“相等过滤”、“简单乘法”；过关难度依次增加；关卡有先后关系的功能是通过颜色区分和选择性禁用按键实现的：蓝色表示正在攻克的关卡，绿色表示已经通过的关卡，这两种情况下鼠标放在按键上可以看到有颜色变化提示玩家可以点击进入。灰色表示未解锁的关卡，对玩家点击保持禁用状态，不可进入。

## 游戏界面

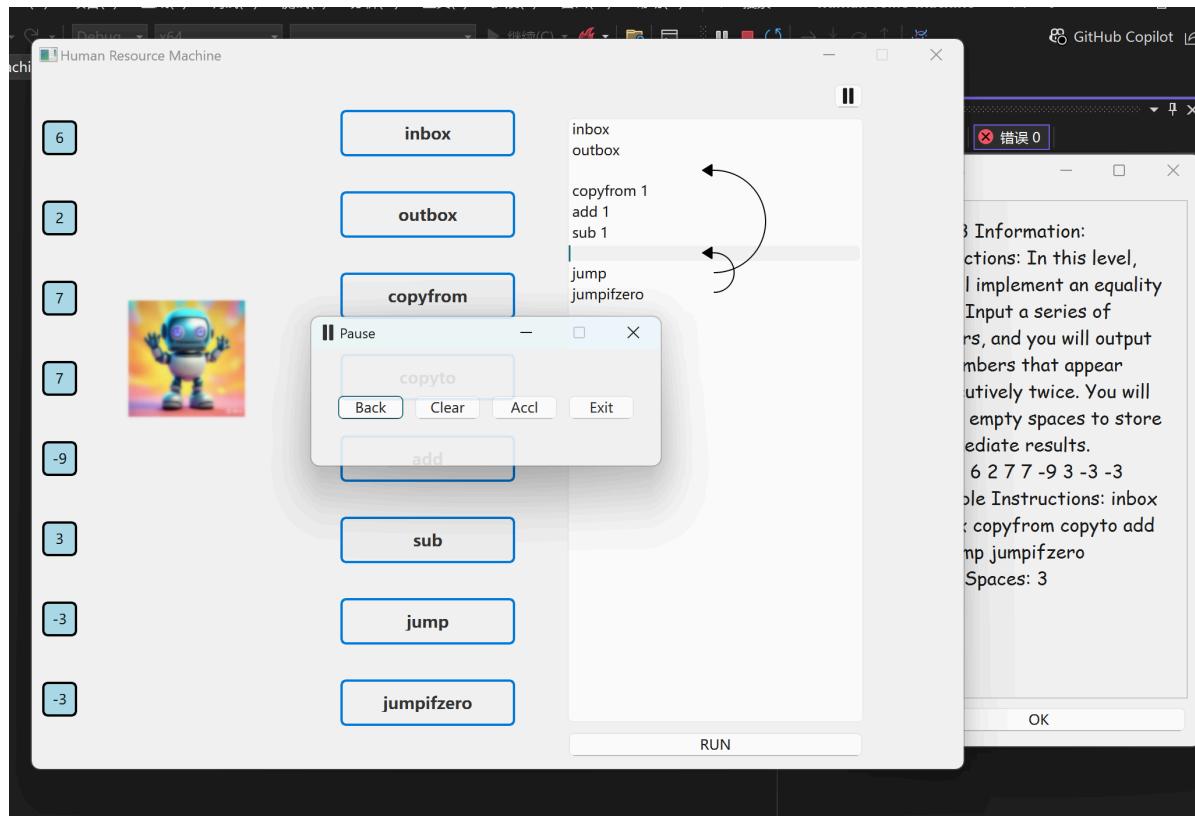


如图可以看到，点击进入某一个关卡时候，系统弹出该关卡的要求信息，包括关卡要求、输入、可用指令、空地等信息。用户可点击 ok 关闭该提示栏。

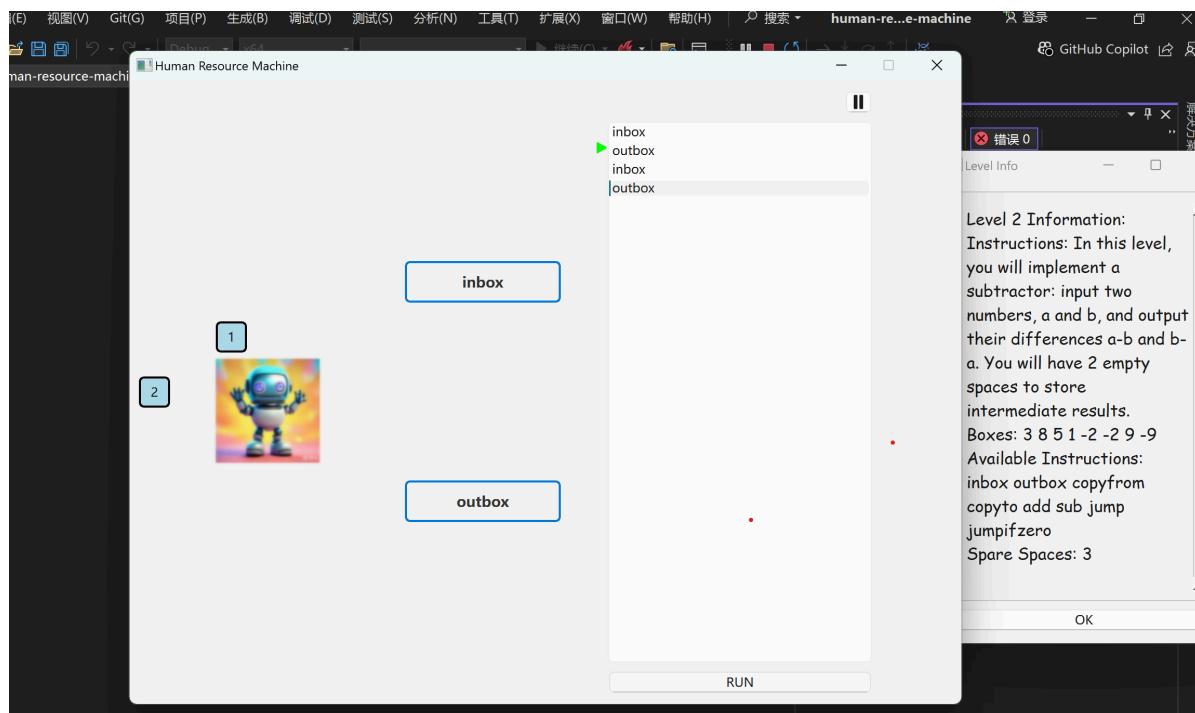


然后玩家通过拖动代码块到 codeblocks 中，交互方式完全还原了原游戏中的鼠标左键按下->拖拽->释放功能，同时可以拖动代码块在 codeblocks 中滑动、互换位置；同时特别值得关注的一点是为了确保 jump 和 jumpzero 指令清晰可见，系统可生成到跳转点的箭头，帮助玩家进行代码撰写。

为了方便玩家进行代码清空、推出等交互操作，我们开发了暂停模块，如图右上角的暂停按键按下后，系统弹出提示消息：如图

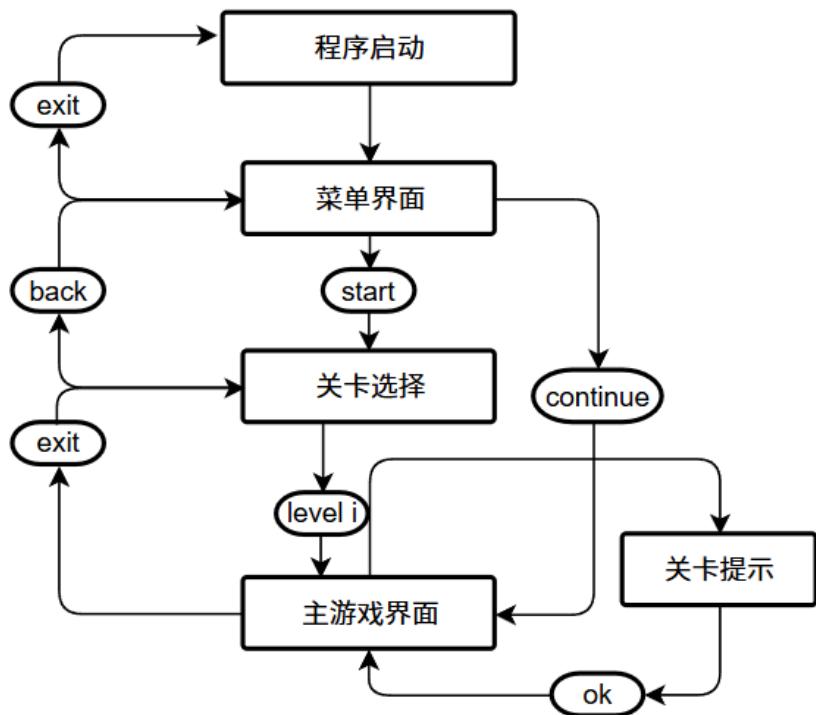


分别可以进行回退、清空代码区、加速执行、暂停的功能，非常方便玩家。



这是游戏主界面第一关运行的截图。当玩家选定代码后，点击 run 按钮可以执行程序，可以看到绿色的光标指示了代码运行位置。程序开始逐行运行后，动画会逐帧输出当前的状态，机器人会从其左手侧的 input 栏拿走一个数字块，按照指令进行操作后输出到机器人右手侧的 output 栏中。

## 用户使用程序的流程图



## 游戏测试结果

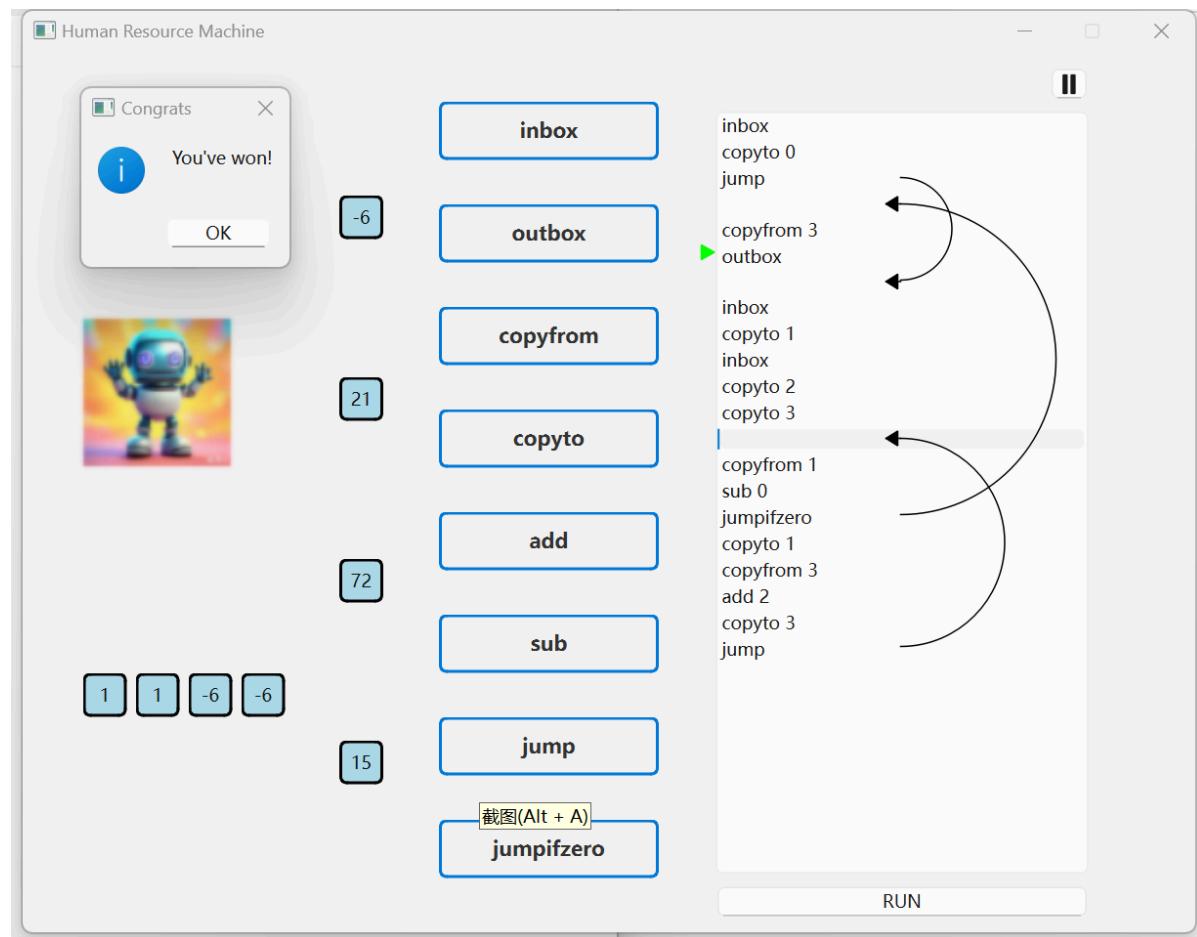
首先说明，由于GUI动画设计原则为，若发生语句错误，动画只加载到错误语句的前件，错误语句的位置由逻辑部分的 `robot_system` 给出，可保证不发生任何动画错误。

### 逻辑部分测试结果

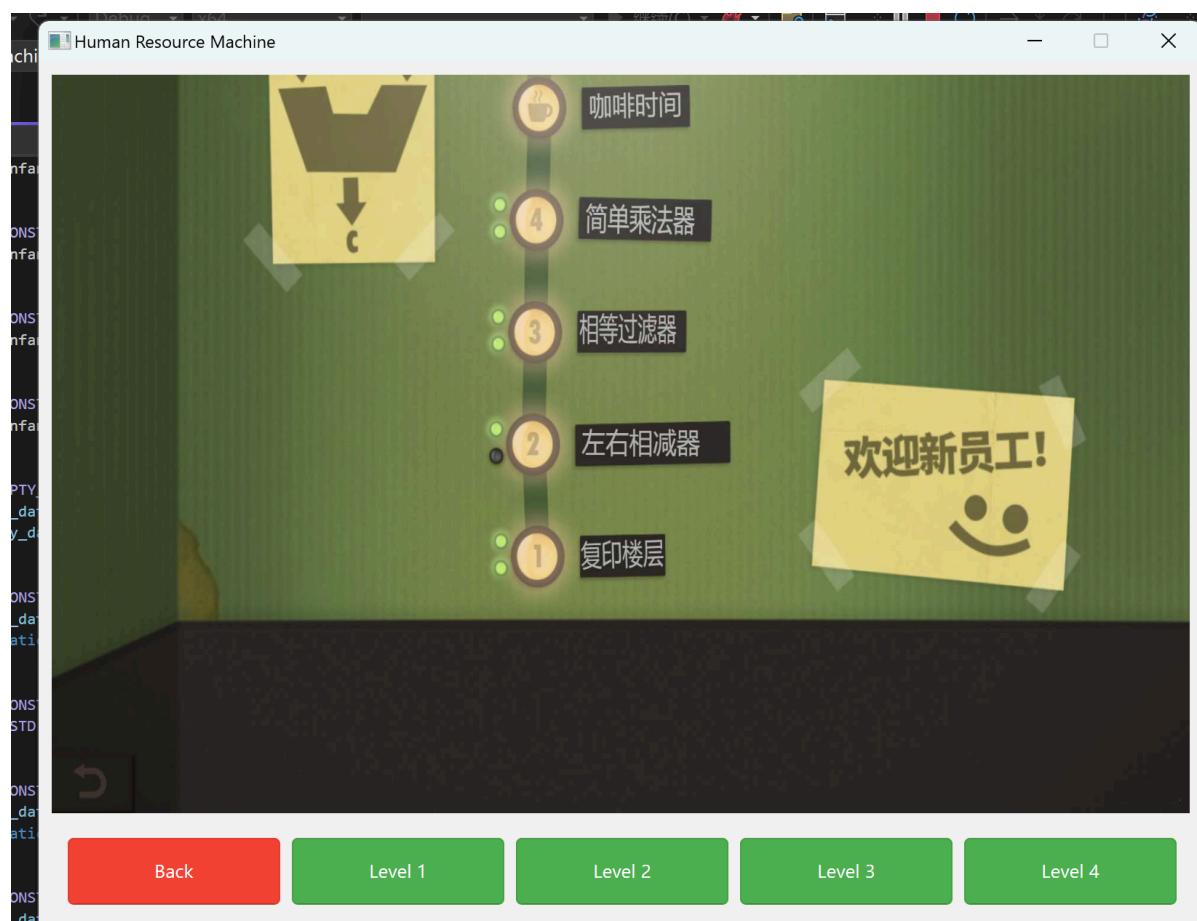
为了检验程序是否正确，我们制作了30个样例，结果均在 `test - i.in && i.out`  $i=1\sim30$ 。文件中，并针对所有可能的异常情况进行检查，全部通过。（命令行版本程序正确性检查）。

### GUI程序正确性测试

由于前一部分整体页面设计实际上给出了完整的用户使用流程，这里不妨给正确运行到的自由创新关卡（第4关）的运行成功截图：



可以看到，结果成功输出，系统提示栏弹出。退出游戏回到关卡选择窗口，可以看到所有的关卡显示通关（绿色）



## GUI程序鲁棒性检查

下面给出出现的异常交互情形以及解决的办法

1. 发现 level\_tip 弹窗出现后，此时返回关卡（虽然这么做不符合正常使用），会再生成一个弹窗提示。

修改办法：

```
void closeAllDialogs() {
    // 获取所有顶级窗口
    QList<QWidget*> windows = QApplication::topLevelWidgets();

    // 遍历并关闭所有QDialog和QMessageBox
    foreach (QWidget* widget, windows) {
        if (widget->isModal()) {
            widget->close();
        }
    }
}

class NoCloseDialog : public QDialog {
public:
    NoCloseDialog(QWidget *parent = nullptr) : QDialog(parent) {
        setWindowTitle("No Close Dialog");
    }

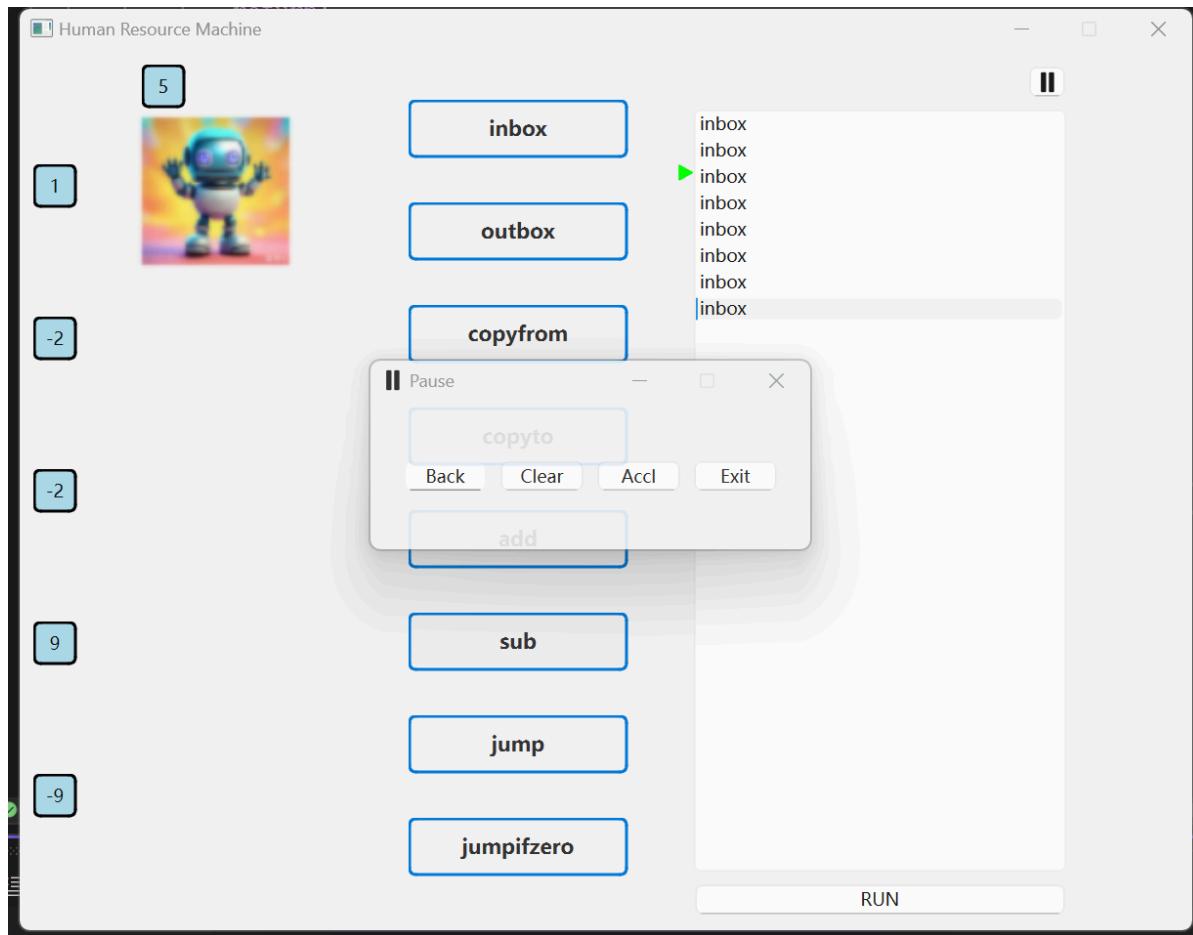
protected:
    void closeEvent(QCloseEvent *event) override {
        // 忽略关闭事件，防止通过 Alt + F4 关闭
        event->ignore();
    }

    void keyPressEvent(QKeyEvent *event) override {
        // 如果按下 Alt + F4，不做任何处理
        if (event->key() == Qt::Key_F4 && event->modifiers() == Qt::AltModifier) {
            event->ignore();
        } else {
            QDialog::keyPressEvent(event);
        }
    }
};
```

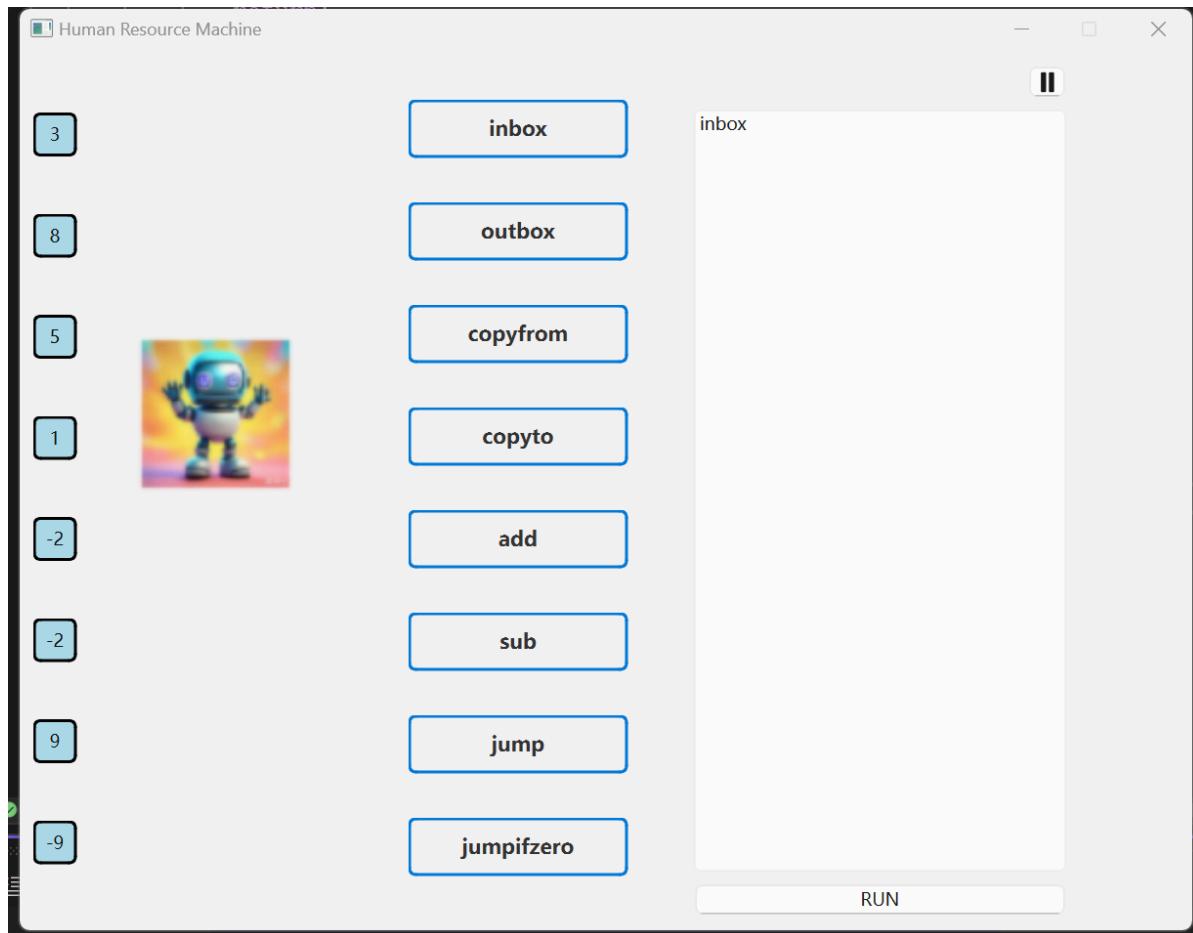
复制

2. 代码块拖入 code\_block 后难以拖出来的情况：系统响应不及时，未找到很好的解决方法。

3. 暂停呼出前，使得当前动画播放完成，阻塞下一帧动画



4. outbox刚从左边那列取出，并在list附近徘徊，但inbox不是currentitem，不会被删掉



## 创新关卡说明

首先我们来看第四关的题面：

Level 4 Information:

Instructions: In this level, you will implement a multiplier. You will output the product of two numbers, and you will have 4 empty spaces to store intermediate results.

NOTICE: You will be given a "1" first, and we will restrict the first element of every pair is positive.

Boxes: 1 3 5 9 8 7 3 1 -6

Available Instructions: inbox outbox copyfrom copyto add sub jump jumpifzero

Spare Spaces: 4

现的是一个乘法器。乘法的实现思路是通过加法来实现的。我们可以采用一种基于“加法乘法”的方法，即通过反复加法来实现两个数的乘法：假设我们需要计算  $a * b$ ，可以通过以下步骤实现：

- 初始化一个累加器，将其设为0。
- 将  $b$  减1，每次循环将  $a$  加到累加器中，重复  $b$  次，直到  $b$  变为0。
- 最终的累加器值即为  $a * b$ 。

### 解题步骤：

1. 输入第一个数（1）并将其存储到存储空间 0 中。
2. 输入第二个数并将其存储到存储空间 1 中。
3. 初始化一个存储空间 2 为0，作为结果的累加器。
4. 在一个循环中，判断第二个数（b）是否为0。如果不为0，将第一个数（a）加到结果累加器中，并将第二个数减去1。
5. 重复步骤4，直到第二个数为0，此时结果即为乘积。
6. 输出结果。

给出伪代码实现：(非常类似于汇编语言)

```
inbox ; 输入第一个数
copyto 0 ; 将第一个数存储到0 (a)
inbox ; 输入第二个数
copyto 1 ; 将第二个数存储到1 (b)
copyto 2 ; 初始化累加器为0

loop:
copyfrom 1 ; 从1读取b
sub 0 ; 减去0 (判断b是否为0)
jumpifzero end ; 如果b为0，跳转到end
copyfrom 2 ; 读取累加器
add 0 ; 将a加到累加器
copyto 2 ; 更新累加器
sub 1 ; 将b减1
jump loop ; 跳回循环

end:
```

```
copyfrom 2      ; 结果存储在2, 输出
```

```
outbox
```

```
给出最后的题解
```

```
1 inbox
2 copyto 0
3 jump 6
4 copyfrom 3
5 outbox
6
7 inbox
8 copyto 1
9 inbox
10 copyto 2
11 copyto 3
12
13 copyfrom 1
14 sub 0
15 jumpifzero 4
16 copyto 1
17 copyfrom 3
18 add 2
19 copyto 3
20 jump 11
```

## 设计灵感

灵感来源于计算机底层实现乘法：计算机最低层其实是二进制八位加法器，乘法来自于对加法循环过程的抽象。给定的存储空间和指令集来实现乘法的核心思想是通过反复加法累积计算结果，直到完成所有的加法操作。该方法在计算机硬件中其实是通过加法和移位操作实现的，而这个题目利用给出的指令集模拟这一过程。

## 代码优缺点

优点：采用了面向对象程序设计方法，使得类与类之间解耦合，增强了类代码的内聚性；使用Qt6信号和槽机制，实现类和类之间的通信。

缺点：部分代码复用性不强、GUI动画设计粗糙简略、机器人页面位置有待优化等

## 讲解视频

```
https://cloud.tsinghua.edu.cn/d/4edaa85061404f1fb41e/
```