大作业实验报告

组长：

- 蓝俊玮 PB20111689

小组成员：

- 蓝俊玮 PB20111689　工作：完成 `sudoku.py` 和 `music.py` 的设计与实现，以及后续内容的设计，整体代码的重构，实验报告的撰写
- 吴远韬 PB20061233　　工作：共同完成 `client.py`，`coop_sudoku.py`，`protocol.py`，`server.py` 的设计与实现
- 田佳林 PB20061251　　工作：共同完成 `client.py`，`coop_sudoku.py`，`protocol.py`，`server.py` 的设计与实现

实验环境：

- Windows 10 Python 3.9.0 Pycharm Community Edition 2021（蓝俊玮）
- Ubuntu 22.04 Python 3.10 Pycharm Community Edition 2022 （吴远韬）
- Windows 11 vscode + Python 3.8.7（田佳林）

1. 项目内容

1.1 项目简介

我们的项目基于 `level4-5参考项目整理.pdf` 文件中提及的音乐播放器和游戏应用项目，最后实现了一个游戏项目：基于 `Socket` 合作以及对战的数独游戏和音乐播放器二合一应用。本项目突破了传统单机小游戏项目只能进行单人游戏，通过增加合作功能和对战功能，既可以给传统单机小游戏增加竞技趣味，又可以为传统单机小游戏增加合作趣味。同时为每个用户都增加了音乐播放的功能，能够在解题的同时享受音乐，沉浸在解数独的乐趣中。最后总共实现代码行数约有 1500 行加。

1.2 项目原理与模型

项目基本的原理就是 `Socket` 通信原理，通过 `Socket` 通信，向网络发出请求或者应答网络请求，使主机间或者一台计算机上的进程间可以通讯，并且采用 `TCP` 协议，采取基于连接的协议，只有当网络用户成功地建立起可靠的连接（即匹配），才可以正式地收发数据，从而实现可靠性保证。 同时使用了多线程技术来实现，从而能够解决 `recv()` 和 `send()` 会阻塞线程的问题。

项目的模型采用的是 `Pygame` 模块，`Pygame` 是一个开源的 `Python` 模块，可以用于 2D 游戏制作，包含对图像、声音、视频、事件、碰撞等的支持。`Pygame` 建立在 `SDL` 的基础上，`SDL` 是一套跨平台的多媒体开发库，用 `C` 语言实现，被广泛的应用于游戏、模拟器、播放器等的开发。`Pygame` 让游戏开发者不再被底层语言束缚，可以更多的关注游戏的功能和逻辑。通过 `Pygame` 实现，使我们的程序具有良好的兼容性，能够在主流的 `Windows` 系统和 `Ubuntu` 系统上运行，并且不需要安装过多的依赖，让游戏的使用变得简单。

1.3 项目功能

- 用鼠标点击选择输入框
- 用鼠标选择是否显示答案
- 用键盘 `[W, S, A, D]` 以及 `[K_UP, K_DOWN, K_LEFT, K_RIGHT]` 实现输入框的上下左右更改
- 鼠标滚轮向上或向下改变播放音乐的音量
- 用键盘 `[1, 2, 3, 4, 5, 6, 7, 8, 9]` 向输入框内输入数字，用 `[0]` 将输入框内的数字删除

- 用鼠标进行播放上一首、播放与暂停、播放下一首以及更改音乐播放模式的操作
- 用鼠标点击变化音乐的播放进度以及音乐的音量
- 选择数独对战模式，进行数独竞技比赛，且一方完成游戏后结束游戏，判定胜负
- 选择数独合作模式，进行数独合作完成，当双方共同完成时退出

在数独对战模式中，两个用户的棋盘和音乐都是独立的，每个用户可以根据自己的喜好添加与播放音乐。

在数独合作模式中，两个用户的棋盘是共享的，音乐是独立的，每个用户可以根据自己的喜好添加与播放音乐。

1.4 项目运行方式

首先安装所依赖的包：

```
pip install -r requirements.txt
```

- 选择单人完成数独时，运行：

  ```
  python main.py
  ```

- 选择双人合作或者对战时，先运行（退出服务器时可以输入 `quit` 或者 `q`）：

  ```
  python server.py
  ```

  然后再开两个终端分别运行：

  ```
  python client.py
  ```

  ```
  python client.py
  ```

2. 项目总体设计

2.1 音乐设计

由音乐的功能，我们设计如下变量：

```
class Music:
    def __init__(self, display):
        self.display = display
        self.font = pygame.font.SysFont("SimHei", 18)
        self.PATH_ROOT = os.path.split(sys.argv[0])[0]  # 当前的根目录
        self.PAUSE_IMG_PATH = "/".join([self.PATH_ROOT, "img/pause.png"])
        # 其他 IMG_PATH ......

        self.music_count = 0    # img 文件夹内总歌曲的数量
        self.music_list = self.get_all_music()
        self.random_list = [i for i in range(0, self.music_count)]  # 随机播放的序
列
        self.random_index = 0   # 随机播放音乐的索引，通过这个来确认当前播放音乐的索引
        self.current_index = 0  # 当前播放音乐的索引
        self.music_offset = 0   # 音乐播放的进度条位置
        self.play_music(self.current_index)
```

```
        self.volume = 0.5        # 音乐的播放声音
        pygame.mixer.music.set_volume(self.volume)  # 设置音量
        self.playing = True      # 当前是否正在播放
        self.loop_mode = 0       # 0 -> loop_all 顺序循环; 1 -> loop_off 随机循环; 2
-> loop_one 单曲循环

        self.button_size = 60    # 设置播放按键的大小
        self.pause_img = pygame.image.load(self.PAUSE_IMG_PATH).convert_alpha()
        # 其他 img 载入
```

- `music_count` 是从 `/music` 文件夹读取到 `*.mp3` 的总数量，即能播放音乐的总数量。
- `music_list` 记录着所有的音乐名称以及音乐时长。
- `random_list` 用来记录随机播放索引序列，这样可以在随机播放模式中，能够保持每首音乐上下音乐的不变。即根据随机播放的定义，整体播放顺序是随机的，但是播放顺序前后是固定的。`random_index` 保持着线性变化，用来访问该随机索引序列。
- `current_index` 用来记录当前播放音乐的索引。
- `music_offset` 用来记录音乐播放的进度条位置，即播放起始位置的初始偏移。pygame 中的音乐模块获取时间并不是直接获取当前音乐播放的时间位置，而是获取从 `pygame.mixer.music.play()` 开始到现在的时间。所以会出现进度条快结束时，而才刚刚调用音乐播放函数的情况导致时间信息不匹配的情况。同时每当使用鼠标拖动音乐播放进度条时，需要更新这个偏移量。
- `playing` 用来表示当前是在播放状态还是暂停状态。
- `loop_mode` 用来表示当前音乐播放器的播放模式。该音乐播放器共有顺序播放、随机播放以及单曲循环三种播放模式。

2.2 数独设计

由数独的功能，我们设计如下变量：

```
class Sudoku:
    def __init__(self, display):
        self.music = Music(display)
        self.flag = [False for _ in range(0, 325)]  # 记录格、行、列、宫的数字情况
        self.init_flag = [False for _ in range(0, 82)]  # 初始生成的记录
        self.sudoku = []    # 记录玩家的数独记录
        for i in range(0, 10):
            self.sudoku.append([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
        self.dfs_flag = [False for _ in range(0, 325)]
        self.dfs_sudoku = []    # dfs搜索保存的数独记录
        for i in range(0, 10):
            self.dfs_sudoku.append([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
        self.dfs_answer = []    # 数独的解记录
        for i in range(0, 10):
            self.dfs_answer.append([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
        self.show_answer = False    # 当前是否为显示解答
        self.dfs_done_flag = False  # 记录是否搜索过解答，可以减少多次搜索带来的时间损耗
        self.font = pygame.font.SysFont("SimHei", 36)
        self.choose = [1, 1]        # 用户当前选择填入数字的位置
        self.rand_sudoku()
```

- `music` 变量是记录音乐的信息。
- `flag` 变量记录用户当前所能更改的数字及位置，程序根据该标记来判断用户是否能够在某个位置输入。每当用户输入或删除一个数，都会更改 `flag` 标记。
- `init_flag` 变量记录程序初始化数独的位置，用户无法更改其相应信息。

- `sudoku` 变量记录玩家当前数独的完成记录，当 `sudoku` 填满时，则完成求解数独，游戏结束。
- `dfs_flag` 变量用来记录求解时所用的标记信息，由于在 DFS 中需要不断地更新标记信息，故将其从 `flag` 中分离出来。同理，`dfs_sudoku` 变量也是为了求解而将其从 `sudoku` 中分理出来。
- `dfs_answer` 变量用来记录求解得到的数独答案。而 `show_answer` 表示当前界面是否显示答案（还是选择显示用户的输入），`dfs_done_flag` 变量用来记录是否已经求解过答案了，如果已经求解过并得到答案后，在后面的选择显示答案时，无需再次进行 DFS 搜素，而是直接显示已经得到的答案。
- `choose` 变量用来记录用户当前选择输入数字的位置。用户每次按下数字键后，都会在该位置对应的数独内进行输入。

2.3 竞技合作用户设计

```
class User:
    def __init__(self, id, addr, sock: socket.socket):
        self.id = id
        self.addr = addr
        self.sock = sock
        self.status = None
        self.other = None
        self.alive = True
        self.game = None
        self.u1_or_u2 = None
```

- `id` 是用户的唯一标识，在多用户对战或者协作的场景下，可以用于各个用户之间的区分。
- `sock` 用来保存该用户的 `socket` 信息。
- `game` 是一个 `Game` 类的对象，用于存储该用户正在进行的游戏的信息。
- `state` 用来保存用户的状态，用户状态有以下几种：`waiting battle`，`win`，`lose`，`waiting cooperate`，`cooperate finish`，`gaming`。在不同游戏模式和不同的状态下，可以使用 `change_state()` 方法修改用户的状态。
- `alive` 用来记录用户是否在线中。
- `u1_or_u2` 用于双人模式中，区分两位玩家，可以取的值是 1 或 2。
- `other` 是双人模式中作为另一位玩家的引用，可以利用这个属性修改对另一个玩家进行操作。

2.4 竞技合作游戏设计

```
class Game:
    def __init__(self, file_num, u1: User, u2: User):
        self.file_num = file_num
        self.sudoku = []
        for i in range(0, 10):
            self.sudoku.append([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
        self.mutex = threading.Lock()
        self.user1 = u1
        self.user2 = u2
        self.u1_choose = [1, 1]
        self.u2_choose = [1, 1]
        self.load_sudoku()
```

- `file_num` 变量用来记录程序的一次执行中使用了哪个数独文件进行初始化。
- `sudoku` 变量记录玩家当前数独的完成记录。
- `mutex` 互斥锁，用于不同用户互斥地访问相关的临界资源（即数独的填写增删）。
- `user1`，`user2` 用来记录两位玩家的信息

- `u1_choose`, `u2_choose` 用来记录两位玩家的 `socket` 是否匹配并选中，即 `user1` 和 `user2` 是否为有效的玩家信息。

2.5 服务器设计

```
class Server:
    def __init__(self):
        self.index = 0
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.bind((host, port))
        self.sock.listen(10)

    def accept_client(self):
        while True:
            conn, addr = self.sock.accept()
            user = User(self.index, addr, conn)
            self.index += 1
            g_users.append(user)
            client_th = threading.Thread(target=Server.message_handle, args=
(self, user))
            client_th.setDaemon(True)
            client_th.start()
```

- `index`：用于记录用户的索引，每当一个用户接入服务器，`index` 属性就加 1。服务器可以支持用户随时的加入与退出
- `sock`：面向连接的套接字，服务器会将两个等待匹配中的用户匹配在一起。

对于服务器 `Server` 的类方法设计如下：

- `accept_client()`：添加新用户的方法，通过 `socket` 的 `accept()` 方法，当 `accept` 到新加入的客户时，就调用该方法，添加用户的信息，增加用户的数量，将其加入到在线用户列表中 `g_users`，并且创建处理该用户相应的线程。
- `message_handle()`：用于处理来自客户端的信息的方法。信息根据其发出的事件的种类可以分为：`match battle`, `battle finish`, `match cooperate`, `choose`, `set`, `cooperate_finish` 几种，该方法可以根据消息的种类使服务器做出对应的响应。
- `match()`： 匹配完成时，修改两个用户的 `other` 属性，更新两个用户的状态，同时选择备选的四道数独题目中的一道作为两人共同的题目，随后向两名用户发送对方用户的标识符 `id` 和题目内容；
- `run()`：控制游戏运行的方法。可以根据用户输入查看用户数量或停止游戏。

2.5.1 Python 多线程使用

`Python` 多线程实现在旧版本中使用 `thread` 模块，而在 `python2.7` 和 `python3` 中则使用 `threading` 模块代替，我们的程序使用操作更加丰富的 `threading` 模块实现。

`Threading` 模块包含了关于线程的丰富操作，例如：常用线程函数，线程对象，锁对象，递归锁对象，事件对象，条件变量对象，信号量对象，定时器对象，栅栏对象。

互斥锁的使用：互斥锁需要用到 `threading` 模块中的 `lock` 类。`lock` 类是一种同步原语。可以使用 `threading.Lock()` 创建一个 `lock` 对象。`lock` 类最常使用的方法有三个：

- `acquire()`： 对互斥锁进行上锁操作，锁在被使用之前，一直保持阻塞状态。
- `release()`： 解锁。
- `locked()`： 检测此时互斥锁是否上锁。

一个互斥锁并不是被它所“上锁”的线程所**拥有**的，而是共享的，另一个线程也可以对其进行解锁操作。但是在使用互斥锁的时候需要避免死锁现象。

2.5.2 Python Socket 使用

程序中两个用户之间的数据传输是通过 `socket` 实现的。

`python` 的 `socket` 模块提供了 `python` 对 `BSD` `socket` 的接口。

可以使用以下的函数创建一个 `AF_INET STREAM` 套接字：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

其中，`AF_INET` 是地址簇，表示使用的网络协议是 `IPv4`；`SOCK_STREAM` 便是使用的文件传输协议是 `TCP`。

函数 `socket.socket()` 创建了一个 `socket`，并返回了一个描述符一边后续使用和 `socket` 相关的函数。

`socket` 模块是套接字在 `python` 中的接口，对于 `socket` 收发信息的操作可以使用 `python` 的内建函数 `read()`，`write()` 等实现。

2.5.3 Socket TCP 用户层协议

`socket` 的 `TCP` 协议为流式协议，与 `UDP` 不同，若不规定用户层协议，则 `TCP` 会出现“沾包分包”问题。也就是说，`TCP` 协议的 `socket.recv()` 一次，返回数据可能是一个请求，也可能是多个请求，也可能不足一个请求。所以需要规定用户层协议，以区分一次 `recv` 中返回数据的不同请求。 定义发送数据格式：`header(4bytes) + body`

`header` 为 4 个字节，记录 `body` 的大小，`body` 为发送的主体信息。如此设计便能在接受消息时根据 `header` 的 `bodysize` 来区分 `recv` 返回的是一个请求还是多个请求或是半个请求。

```
def package(send_dict: dict) -> bytes:
    body = json.dumps(send_dict)  # body 为发送的数据，格式为 json 字符串
    header = len(body)  # header为 4 字节数据，表示该请求 body 的大小
    header_pack = struct.pack('1I', header)
    if DEBUG:
        print(header)
    return header_pack + body.encode(encoding='utf-8')
```

发送数据时规定代码如下，用一个字典存储，发送打包数据：

```
send_dict = {'type': type}
#...构建send_dict
sock.send(package(send_dict))
```

接收数据规定代码如下：

```
def recv():
    dataBuffer = bytes()    # 设置一个接受缓冲区
    while True:
        try:
            recv_bytes = sock.recv(512)
        except:     # 处理 tcp 连接断开的情况
            #connect error
            break
```

```
        if recv_bytes == b"":    # 高版本中 tcp 连接断开时，会返回一个空数据
            #connect error
            break
        if recv_bytes:       # 当接收到数据时
            dataBuffer += recv_bytes
            while True:
                if len(dataBuffer) < HEADER_SIZE:    # 当结束数据超出缓冲区时
                    break
                header = struct.unpack('1I', dataBuffer[:HEADER_SIZE])  # 将得到的
数据解包
                body_size = header[0]
                if DEBUG:
                    print(body_size)
                if len(dataBuffer) < HEADER_SIZE + body_size:
                    break
                body = dataBuffer[HEADER_SIZE:HEADER_SIZE + body_size]
                handle(body)
                dataBuffer = dataBuffer[HEADER_SIZE + body_size:]
```

即接受数据之前首先设置一个缓冲区，将读取的有效信息传入到缓冲区内，再调用 `handle()` 处理缓冲区的内容。

3. 项目模块具体实现

3.1 项目绘图

3.1.1 数独绘图

```
def draw_background(self, display):
    display.fill(Color("0xF0CCFF"))     # 0xFFB6C1
    x, y = 60, 120
    for i in range(0, 10):
        pygame.draw.line(display, Color("white"), (x, y + i * 40), (420, y + i *
40))
        pygame.draw.line(display, Color("white"), (x + i * 40, y), (x + i * 40,
480))
    for i in range(0, 4):    # 将每一个九宫格区分开，更明显
        pygame.draw.line(display, Color("0xFF5555"), (x, y + i * 120), (420, y +
i * 120))
        pygame.draw.line(display, Color("0xFF5555"), (x + i * 120, y), (x + i *
120, 480))
    self.grain(display)     # 绘制纹路
    if not self.show_answer:
        self.draw_button(display, 400, 10, 470, 45, "答案")
    else:
        self.draw_button(display, 400, 10, 470, 45, "返回")
    self.draw_button(display, 300, 10, 370, 45, "清空")
```

将棋盘大小设置为 360×360，每一个小的棋格大小设置为 40，从 $(60, 120)$ 画到 $(420, 480)$，并且在右上角绘制清空和答案按钮。同时在棋盘下面绘制音乐播放器的按钮以及音乐播放进度条和播放音量进度条。

3.1.2 绘制音乐播放器的进度条

```
def draw_music_progress(self, val=0):
    rect_progress_all = pygame.Rect(20, 570, 200, 20)
    rect_progress_music = pygame.Rect(20, 570, int(200 * val), 20)
    pygame.draw.rect(self.display, Color("grey"), rect_progress_all)
    pygame.draw.rect(self.display, Color("pink"), rect_progress_music)

def draw_volume_progress(self, val):
    rect_progress_all = pygame.Rect(260, 570, 200, 20)
    rect_progress_volume = pygame.Rect(260, 570, int(200 * val), 20)
    pygame.draw.rect(self.display, Color("grey"), rect_progress_all)
    pygame.draw.rect(self.display, Color("pink"), rect_progress_volume)
```

使用 `Rect` 块来实现进度条的动态变化，采用一个总的进度条和一个当前播放进度条。播放进度条根据当前播放音乐的百分比进行更新，将更新的 `Rect` 块覆盖在总的进度条上。

3.1.3 绘制音乐播放器的按钮

```
def draw_buttons(self):
    self.display.blit(self.prev_img, (150, 490))  # 绘制播放上一首按钮
    if self.playing:
        self.display.blit(self.pause_img, (150 + self.button_size, 490))  # 播放模式时绘制暂停按钮
    else:
        self.display.blit(self.play_img, (150 + self.button_size, 490))  # 暂停模式时绘制播放按钮
    self.display.blit(self.next_img, (150 + self.button_size * 2, 490))  # 绘制播放下一首按钮
    if self.loop_mode == 0:
        self.display.blit(self.loop_all_img, (150 + self.button_size * 3, 490))
# 绘制顺序播放按钮
    elif self.loop_mode == 1:
        self.display.blit(self.loop_off_img, (150 + self.button_size * 3, 490))
 # 绘制随机播放按钮
    else:
        self.display.blit(self.loop_one_img, (150 + self.button_size * 3, 490))
 # 绘制单曲循环按钮
```

绘制音乐播放器按钮时，只需要根据当前音乐播放器的模式绘制相应按钮即可。当音乐正在播放时，则绘制暂停按钮；当音乐暂停时，则绘制播放按钮。而当音乐播放器模式为顺序播放模式时，则绘制当前顺序播放的按钮；当其为随机播放模式时，则绘制当前随机播放的按钮；当其为单曲循环按钮时，则绘制当前单曲循环的按钮。

3.2 更改音乐进度条以及音量进度条

```
def play(self):
    # <-- snip -->
    # 拖动播放音乐的进度条
    elif 20 <= event.pos[0] <= 220 and 570 <= event.pos[1] <= 590:
        music_pos = ((event.pos[0] - 20) / 200) *
self.music.get_music_total_time()
        pygame.mixer.music.play(0, music_pos)
        self.music.set_music_offset((event.pos[0] - 20) / 200)
    # 拖动播放声音的进度条
    elif 260 <= event.pos[0] <= 460 and 570 <= event.pos[1] <= 590:
        volume_pos = (event.pos[0] - 260) / 200
        self.music.set_volume(volume_pos)
        pygame.mixer.music.set_volume(volume_pos)
    # <-- snip -->
```

当鼠标点击到音乐进度条时，将点击位置对应转化为音乐播放的时间，并用 `pygame.mixer.music.play(0, music_pos)` 直接从偏移位置开始播放，同时以百分比形式更新音乐播放的偏移量；当鼠标点击到音量进度条时，直接将其转化为对应的音量百分比，并用 `set_volume(volume_pos)` 设置音乐的播放音量。

3.3 数独判重标记

根据数独的规则，一个数字所在的行、列和宫内不能存在相同的数字，所以我们需要使用一个列表 `flag[325]` 保存记录情况。规定 `flag[1:81]` 为数独总计 81 个格子内是否已经存在数字，`flag[82:162]` 为数独总计 9 行内每一行分别存在了哪些数字，`flag[163:243]` 为数独总计 9 列内每一行分别存在了哪些数字，`flag[244:324]` 为数独总计 9 个宫内每一个宫内分别存在了哪些数字。根据这些规定，如果有一个数在 `i` 行 `j` 列的数字 `sudoku[i][j]`，则下列标记记为 `True`：

```
current = (i - 1) * 9 + j
row = (i - 1) * 9 + sudoku[i][j] + 81
col = (j - 1) * 9 + sudoku[i][j] + 162
house = (int((i - 1) / 3) * 3 + int((j - 1) / 3)) * 9 + sudoku[i][j] + 243
flag[current] = flag[row] = flag[col] = flag[house] = True
```

3.4 选择框的显示

根据我们的设计，我们为数独输入位置做出了显示框的设计，方便用户清楚的意识到自己当前会填入的位置。读取当前选择输入格子的位置 `choose[0]` 和 `choose[1]`，根据该位置得到该格子的四个顶角坐标，然后以它们作为线的起始点以及终点用不同的颜色进行绘图，从而达到选择框的效果：

```
def draw_choose(self, display):
    x, y = 60, 120
    # 得到四个顶角的坐标位置
    left_up = (x + (self.choose[1] - 1) * 40, y + (self.choose[0] - 1) * 40)
    right_up = (x + self.choose[1] * 40, y + (self.choose[0] - 1) * 40)
    left_down = (x + (self.choose[1] - 1) * 40, y + self.choose[0] * 40)
    right_down = (x + self.choose[1] * 40, y + self.choose[0] * 40)
    # 顺时针进行绘图
    pygame.draw.line(display, Color("black"), left_up, right_up)
    pygame.draw.line(display, Color("black"), right_up, right_down)
    pygame.draw.line(display, Color("black"), right_down, left_down)
    pygame.draw.line(display, Color("black"), left_down, left_up)
```

3.5 数独解的搜索策略

我们的实验采用的是 DFS 算法对数独解进行搜索，从坐标 `(1,1)` 开始搜索到坐标 `(9,9)`，并且将得到的数独解用 `dfs_answer` 记录保存。在每次搜索时，如果当前位置已经有数字了，则会跳过当前位置，进行下一次搜索。如果当前没有数字，则采取 DFS 进行搜索，从 1 到 9 开始尝试，当满足数独规则可以填入时，则填入并进行下一次搜索，同时保存行列宫信息；如果当 1 到 9 都不能填入时，则回溯到上一次搜索过程，并撤销该数字的行列宫信息。

```
def dfs(self, x, y):
    # 当前位置不是 0 时，则不需要搜索
    if self.dfs_sudoku[x][y] != 0:
        if x == 9 and y == 9:   # 找到数独的解
            self.show_answer = True
            self.dfs_answer = copy.deepcopy(self.dfs_sudoku)
            return
        if y == 9:  # 从下一行开始搜索
            self.dfs(x + 1, 1)
        else:       # 从下一列开始搜索
            self.dfs(x, y + 1)
    else:
        for dfs_num in range(1, 10):
            current = (x - 1) * 9 + y
            row = (x - 1) * 9 + dfs_num + 81
            col = (y - 1) * 9 + dfs_num + 162
            house = (int((x - 1) / 3) * 3 + int((y - 1) / 3)) * 9 + dfs_num +
243
            if not self.dfs_flag[row] and not self.dfs_flag[col] \
                    and not self.dfs_flag[house] and not
self.init_flag[current]:
                self.dfs_sudoku[x][y] = dfs_num
                self.dfs_flag[current] = self.dfs_flag[row] = True
                self.dfs_flag[col] = self.dfs_flag[house] = True
                if x == 9 and y == 9:   # 找到数独的解
                    self.show_answer = True
                    self.dfs_answer = copy.deepcopy(self.dfs_sudoku)
                    self.dfs_sudoku[x][y] = 0   # 这里要记得回溯，不然(9,9)会出现问题
                    self.dfs_flag[current] = self.dfs_flag[row] = False
                    self.dfs_flag[col] = self.dfs_flag[house] = False
                    return
                if y == 9:  # 从下一行开始搜索
                    self.dfs(x + 1, 1)
                else:       # 从下一列开始搜索
                    self.dfs(x, y + 1)
                # 回溯
                self.dfs_sudoku[x][y] = 0
                self.dfs_flag[current] = self.dfs_flag[row] = False
                self.dfs_flag[col] = self.dfs_flag[house] = False
```

3.6 服务器收发消息

3.6.1 客户端

```
def socket_recv(coop_mode=False):
    global find_flag
    global Opponent
    global win
    global lose
    global file_num
    data_buffer = bytes()    # 接收信息的缓冲区
    while True:
        try:
            data = sock.recv(512)
            if data:         # 当收到消息时
                data_buffer += data
                while True:
                    if HEADER_SIZE > len(data_buffer):  # 如果接受的信息有丢失
                        break
                    header = struct.unpack('1I', data_buffer[:HEADER_SIZE]) # 解
包信息
                    body_size = header[0]
                    if HEADER_SIZE + body_size > len(data_buffer):
                        break
                    body = data_buffer[HEADER_SIZE:HEADER_SIZE + body_size]
                    status = handle(body)    # 处理数据，返回程序状态
                    if status == 3:
                        return
                    if coop_mode and status & 1 == 1:
                        return
                    data_buffer = data_buffer[HEADER_SIZE + body_size:]
        except:
            break
```

这里的实现大概在 2.5.3 Socket TCP 用户层协议 中有介绍过。就是首先设置一个缓冲区，当接收到新消息时，将其 `recv()` 到缓冲区内，然后判断数据是否丢失或者接收信息 `recv()` 是否未完成，其中 `HEADER_SIZE` 是我们在 `config` 文件中定义的数据包大小。当其成功接受消息的时候，将其解包并处理，然后使用句柄函数 `handle()` 处理这个数据，完成操作。

3.6.2 服务端

```
def message_handle(self,u:User,body:bytes):
    msg = json.loads(body.decode(encoding='utf-8'))
    if DEBUG:
        print(msg)
    if msg['type'] == 'match battle':
        oppo = None
        mutex.acquire()  # 这里要加锁
        if battle_waiting_queue.qsize() > 0:
            oppo = battle_waiting_queue.get()    # 这里实际上只对消息队列进行了上锁与解
锁
            mutex.release()      # 解锁
            flag, _ = self.match(u, oppo)
            if flag == 1:
                u.change_status('waiting battle')
                battle_waiting_queue.put(u)
            if flag == 2:
                oppo.change_status('waiting battle')
```

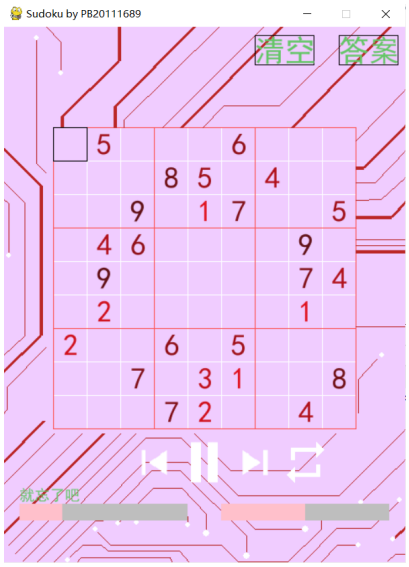
```
                battle_waiting_queue.put(u)
        else:
            mutex.release()      # 解锁
            u.change_status('waiting battle')
            battle_waiting_queue.put(u)
        print('user ' + str(u.id) + ' join matching')
    # 这里只介绍其中一部分的消息操作，其他的消息操作都是类似的
```

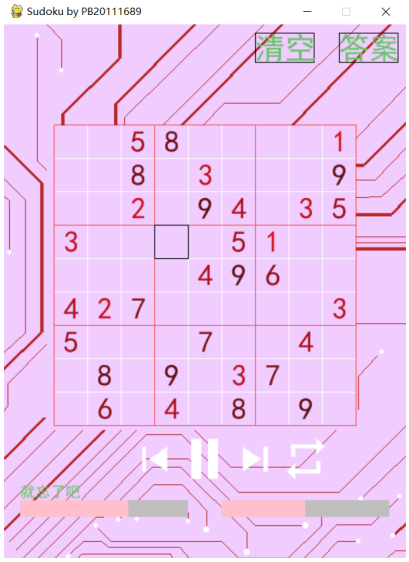
由我们刚学的操作系统知识可知，这里的合作问题是一个多生产者问题，多个生产者对临界资源的访问是有限制的，临界资源在同一时刻只能由一个生产者独立地访问，我们需要避免多个生产者同时进入临界区进行操作。同时 `Socket` 是全双工的，即可以由两个线程同时分别读取与写入。但并发地读取或并发地写入可能会造成意想不到的后果。所以我们在这里采用互斥锁的方式，保证两个用户在合作时不会进行同步写的操作，同一时刻只能由一个人访问临界资源，这样就可以解决互斥问题。那这样就要求我们在 `Socket` 中的 `recv()` 和 `send()` 操作前后上锁。

由于 `recv()` 和 `send()` 操作这两个方法是阻塞线程的，阻塞操作进行加锁容易引发问题（如因为一个阻塞操作不成功而导致锁无法释放），同时互斥锁的引入本身会对性能造成影响。为了尽可能减小锁对系统性能带来的影响，这里我们采取**细粒度锁**的方式实现：即设置一个信息队列，每次上锁与解锁的操作只针对信息队列的入队操作与出队操作（获取操作），而不再对 `recv()` 和 `send()` 函数加锁，将 `Socket` 的处理采用信息队列处理，这样可以大幅减少互斥锁的占用时长，既可以提高我们程序的完全性能，还能提高处理性能。

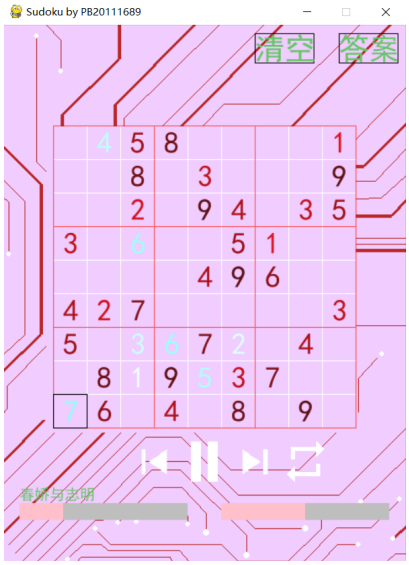
4. 项目展示

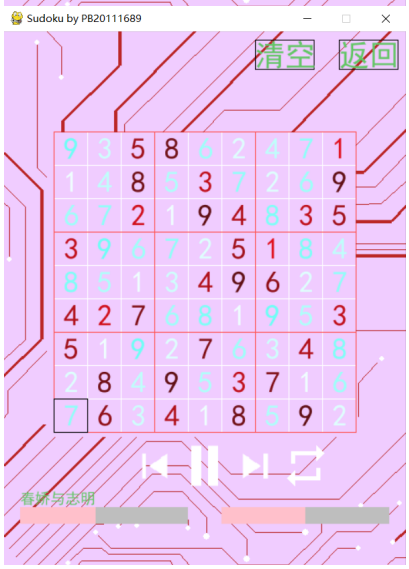
- 用鼠标点击选择输入框





- 用鼠标选择是否显示答案





- 用键盘 `[W, S, A, D]` 以及 `[K_UP, K_DOWN, K_LEFT, K_RIGHT]` 实现输入框的上下左右更改（在附件视频中可看）
- 鼠标滚轮向上或向下改变播放音乐的音量（在附件视频中可看）
- 用键盘 `[1, 2, 3, 4, 5, 6, 7, 8, 9]` 向输入框内输入数字，用 `[0]` 将输入框内的数字删除（在附件视频中可看）
- 用鼠标进行播放上一首、播放与暂停、播放下一首以及更改音乐播放模式的操作（在附件视频中可看）

- 用鼠标点击变化音乐的播放进度以及音乐的音量（在附件视频中可看）
- 选择数独对战模式，进行数独竞技比赛，且一方完成游戏后结束游戏，判定胜负（在附件视频中可看）
- 选择数独合作模式，进行数独合作完成，当双方共同完成时退出（在附件视频中可看）

百度云盘链接为：https://pan.baidu.com/s/1_q6xu_V0MfZJZ10WB8rffw 提取码为 USTC

在数独对战模式中，两个用户的棋盘和音乐都是独立的，每个用户可以根据自己的喜好添加与播放音乐。

在数独合作模式中，两个用户的棋盘是共享的，音乐是独立的，每个用户可以根据自己的喜好添加与播放音乐。

5. 总结与展望

通过本次代码设计与实现，我们运用了这学期从老师操作系统 H 课中所学的进程线程知识，实现了一个类多生产者的应用程序，在代码实现中解决了不少互斥问题和死锁问题。同时将所学的理论知识成功运用到了应用场景，结合具体的游戏项目完成了 `Socket` 通信，并为传统单机游戏定义了一种新的娱乐方式。最后我们顺利地完成了我们的既定项目目标："基于 `Socket` 合作以及对战的数独游戏和音乐播放器二合一应用"。

由于我们的竞技模式相当于是一个竞速模式，所以我们组的展望是将我们的 `Socket` 通信运用到一种竞技博弈游戏，例如象棋、围棋等游戏，即保持整体设计不变，而是实现另一种竞技博弈游戏，从而实现两个人真正意义上的竞技与博弈。同时可以参考 `level4-5参考项目整理.pdf` 中 "深度学习与游戏竞技开源平台" 中的代码，将单人模式改成实现人机对战游戏，不断优化与测试其博弈算法。所以等未来我们学习了与深度学习相关的知识方面时，可以继续发展更新我们的项目，使其的功能更加全面完善。

最后感谢老师一学期网课的辛勤付出，同时感谢助教们在线下课程的付出以及在课程当中对我们问题的解答与帮助。