# 逆向分析和数独乐乐文档

## 1 OOAD 逆向分析

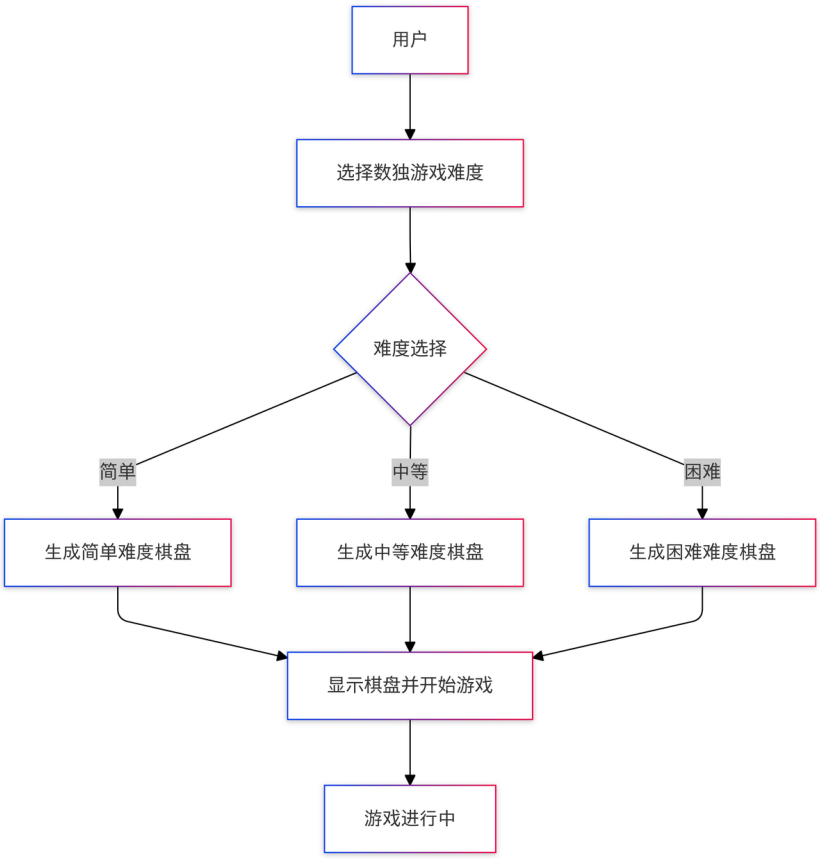
### 1.1 用户愿景

该数独项目的愿景是为用户提供一个易于操作、功能丰富的数独游戏平台。旨在满足不同水平用户（从初学者到资深玩家）对数独游戏的需求，通过直观的界面和实用的辅助功能，如提示功能，笔记功能，让用户能够轻松地进行数独游戏，享受解题的乐趣和挑战。

### 1.2 用例分析

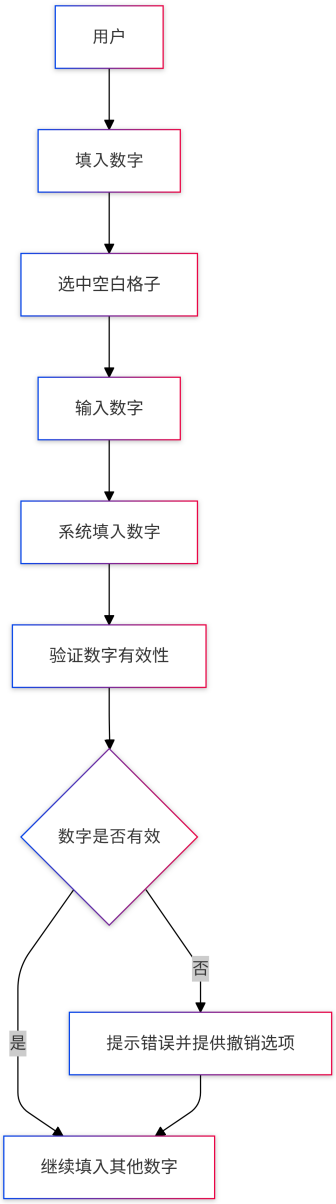
用例 1：用户开始数独游戏

用户首先选择数独游戏的难度级别，这可以是简单、中等或困难。游戏系统根据用户选择的难度级别生成相应难度的数独棋盘。生成棋盘后，系统会显示棋盘并允许用户开始游戏并进行解题。这个用例图展示了用户如何根据个人偏好选择游戏难度，并开始游戏，同时也体现了游戏系统如何响应用户的选择并提供相应的数独棋盘。



用例 2：用户填入数字

用户通过鼠标点击或键盘操作，选中数独棋盘中的一个空白格子。用户输入一个数字（1-9），系统将该数字填入选中的格子中。系统实时验证填入的数字是否符合数独规则，即检查该数字在当前行、列以及 3×3 宫格内是否唯一，若不符合则给予提示，如高亮显示错误的格子、弹出提示框等，同时可提供选项让用户撤销此次输入。

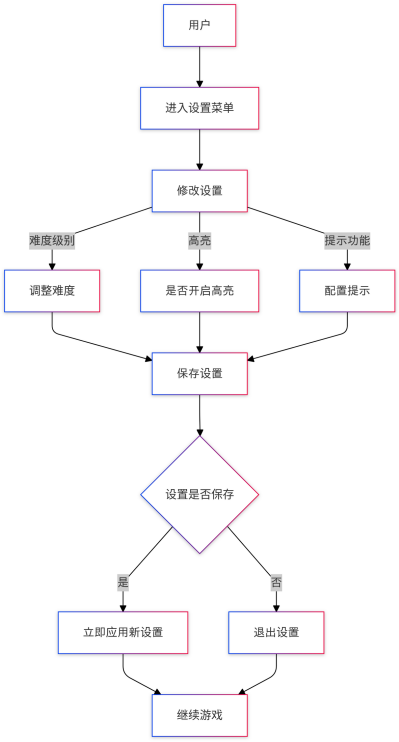



用例 3：用户开启提示功能

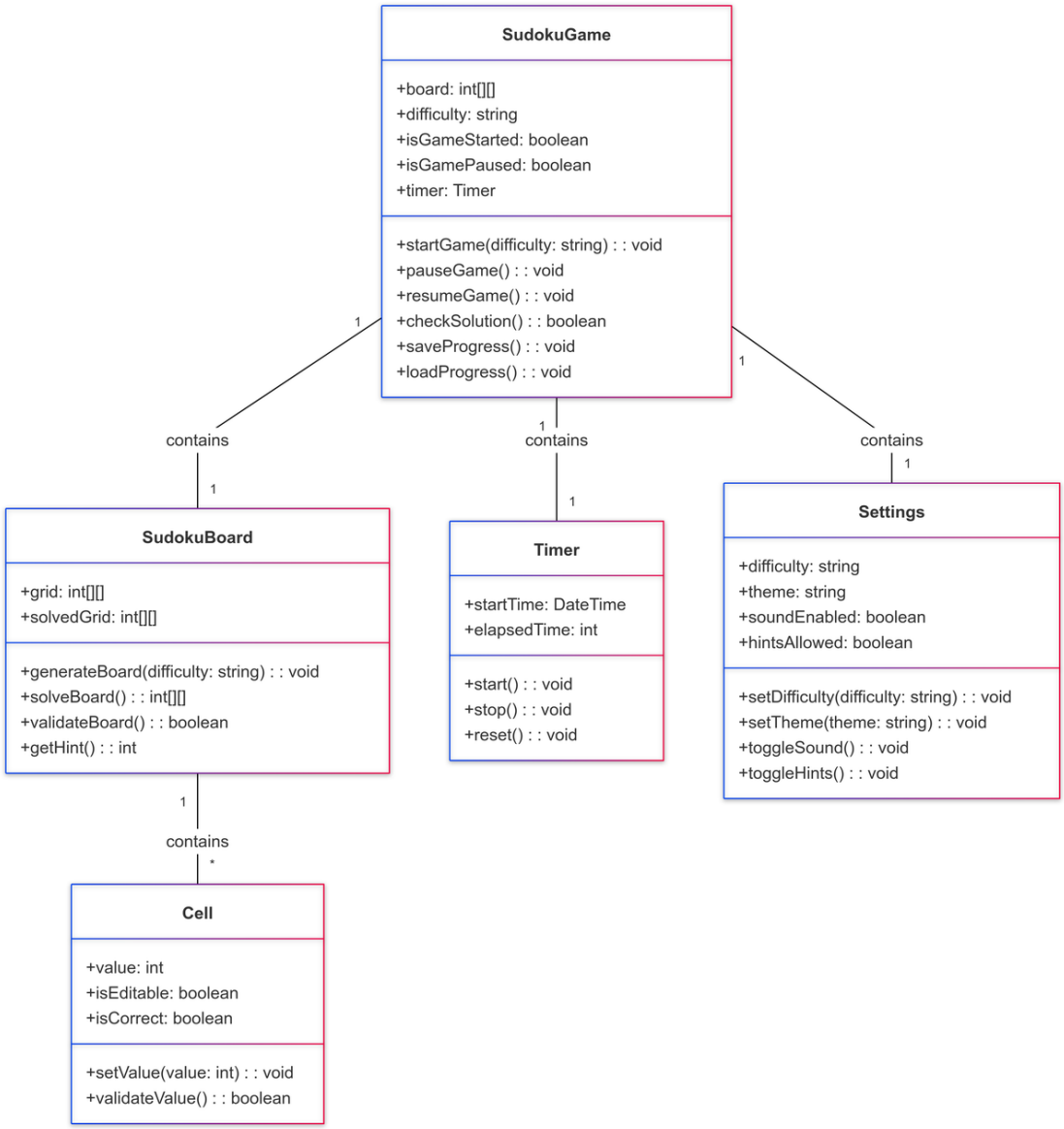
用户在解答过程中遇到困难，选择开启提示功能的操作，系统弹出提示选项，如提示一个格子的正确数字、提示错误的数字等。用户根据需要选择具体的提示类型，系统根据当前数独的状态和用户的选择，为用户提供相应的提示信息，如在某个空白格子中显示正确的数字，或高亮显示错误填写的数字及其所在行、列和宫格的冲突情况。


用例 4：用户对数独进行设置修改

用户选择进入设置菜单的操作，系统展示设置选项，如难度级别调整、是否高亮、提示功能配置等。用户根据自己的喜好和需求，对各项设置进行修改，例如选择不同的难度级别以获得更具挑战性或更轻松的谜题，更换主题颜色以改善视觉效果，开启或关闭音效以适应不同的使用环境，调整提示功能的使用限制等。用户完成设置修改后，选择保存设置的操作，系统将修改后的设置应用到当前游戏或后续游戏中，若用户在修改设置时处于游戏中，系统可询问用户是否立即应用新设置或在下一次游戏时生效。



## 1.3 领域模型




类分析

1. SudokuGame 类

- 描述：代表整个数独游戏，管理游戏的状态和行为。
- 属性：
  - board：二维数组，存储数独棋盘的数据。
  - difficulty：字符串，表示当前游戏的难度级别。
  - isGameStarted：布尔值，表示游戏是否已开始。
  - isGamePaused：布尔值，表示游戏是否已暂停。
  - timer：计时器对象，用于记录游戏时间。

- 方法：
  - startGame(difficulty)：开始新游戏，根据难度生成数独棋盘。
  - pauseGame()：暂停游戏，停止计时器。
  - resumeGame()：恢复游戏，继续计时。
  - checkSolution()：检查数独棋盘是否正确解答。
  - saveProgress()：保存游戏进度。
  - loadProgress()：加载游戏进度。

2. SudokuBoard 类

- 描述：表示数独棋盘，负责棋盘的生成和验证。
- 属性：
  - grid：二维数组，存储棋盘的初始数据。
  - solvedGrid：二维数组，存储数独的解答。
- 方法：
  - generateBoard(difficulty)：生成符合难度的数独棋盘。
  - solveBoard()：求解数独棋盘，返回解答。
  - validateBoard()：验证棋盘是否符合数独规则。
  - getHint()：提供一个格子的正确数字作为提示。

3. Cell 类

- 描述：表示数独棋盘中的一个单元格。
- 属性：
  - value：整数，单元格的值（1-9）。
  - isEditable：布尔值，表示单元格是否可编辑。
  - isCorrect：布尔值，表示单元格的值是否正确。
- 方法：
  - setValue(value)：设置单元格的值。
  - validateValue()：验证单元格的值是否符合数独规则。

4. Timer 类

- 描述：用于记录游戏时间。
- 属性：
  - startTime：日期时间，游戏开始的时间。
  - elapsedTime：整数，已过去的时间（秒）。

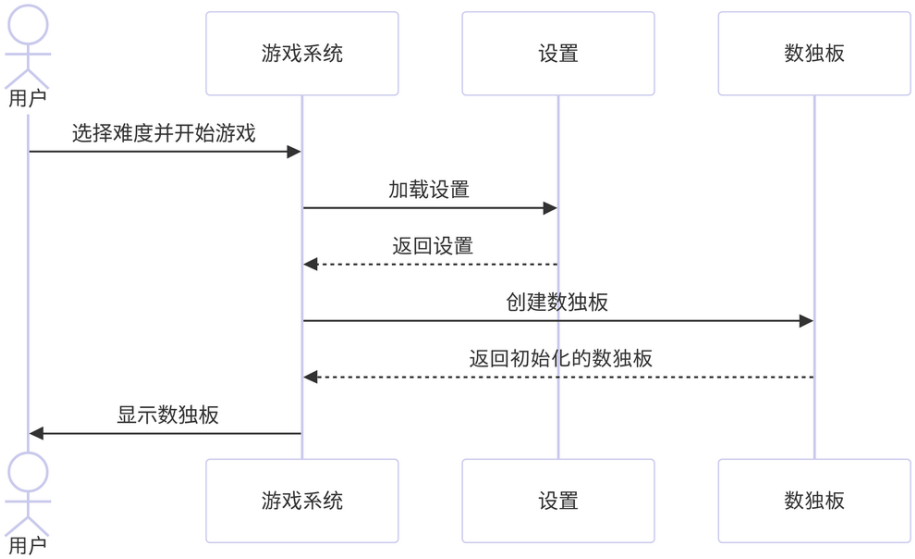
- 方法：
  - start()：开始计时。
  - stop()：停止计时。
  - reset()：重置计时器。

5. Settings 类

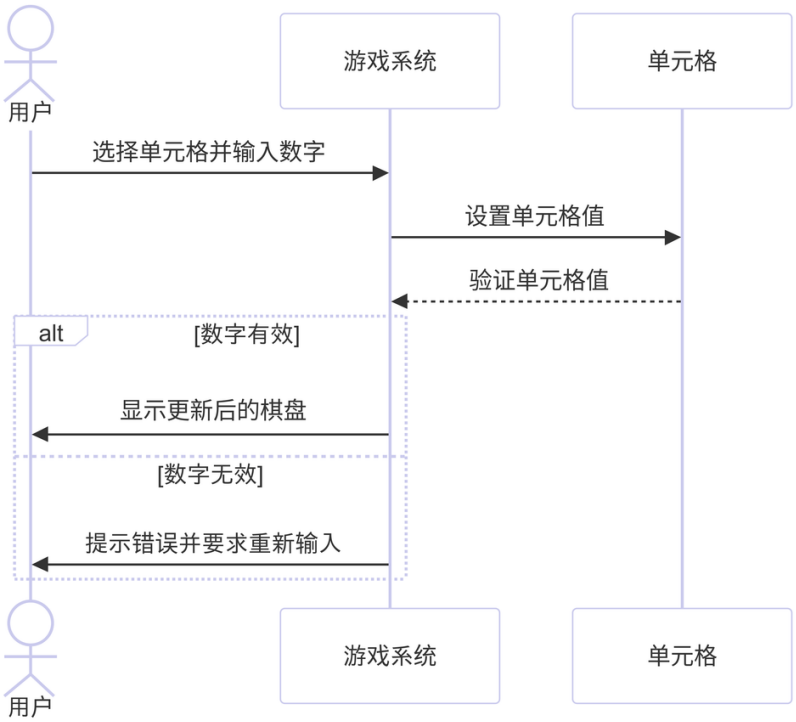
- 描述：管理游戏设置。
- 属性：
  - difficulty：字符串，当前难度级别。
  - highlightEnabled：布尔值，表示高亮是否开启。
  - hintsAllowed：布尔值，表示提示功能是否允许。
- 方法：
  - setDifficulty(difficulty)：设置难度级别。
  - toggleHighlight()：切换音效开关。
  - toggleHints()：切换提示功能开关。

## 1.4 行为模型

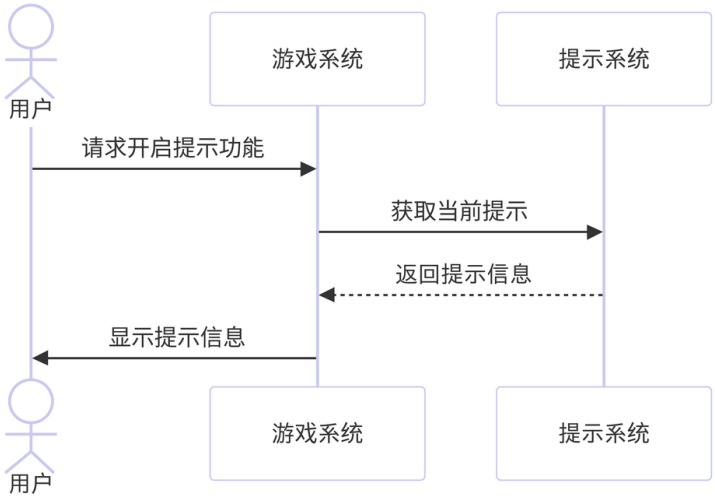
- 开始游戏：用户启动数独游戏，并选择不同难度开始游戏，系统加载默认设置并显示游戏界面。



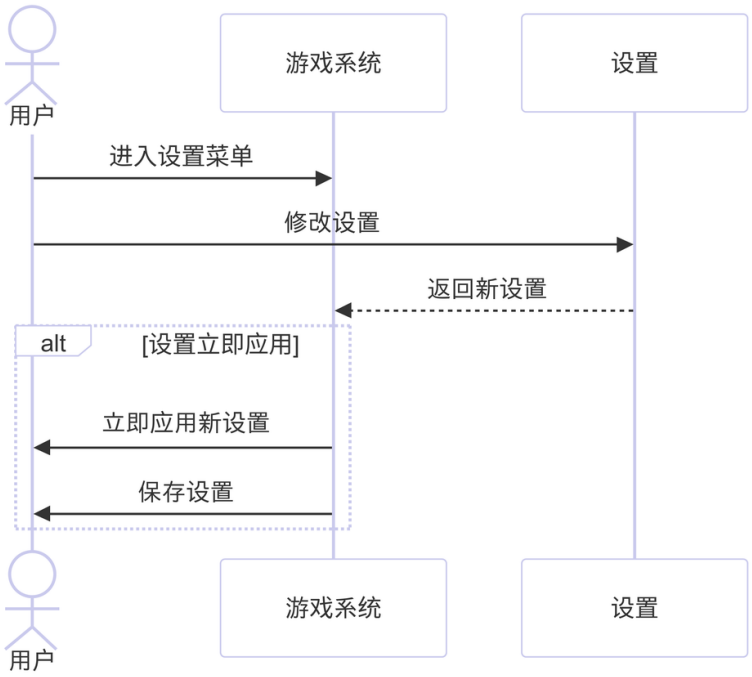

- 填入数字：用户选中空白单元格并输入数字，系统验证数字的有效性，若无效则提示错误并提供撤销选项。




- 开启提示功能：用户可以选择开启提示功能，系统根据请求提供正确数字提示或候选值提示。

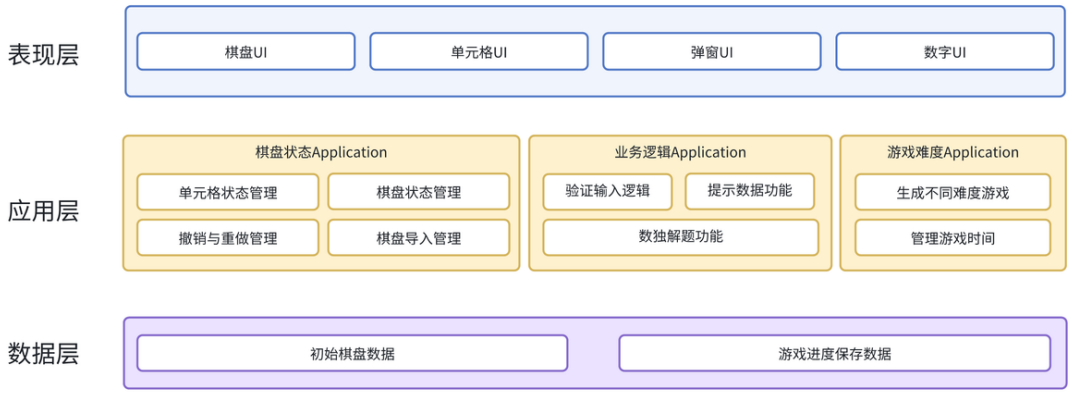



- 进行设置修改：用户进入设置菜单修改游戏设置，系统根据用户选择立即应用新设置或者退出设置。




## 1.5 系统技术架构

系统使用 Svelte 进行组件化开发，样式使用 TailwindCSS，状态管理使用 Svelte 的 store 模式。




## 1.6 设计改进建议

- 实现未完成的功能，如撤销/重做操作和创建自定义数独游戏。
- 增加更多高级功能，如提供提示、选择难度、保存游戏进度等。
- 优化现有代码逻辑，减少冗余代码，提升性能。
- 考虑引入单元测试和端到端测试，确保代码的稳定性和可靠性。

# 2 数独乐乐

## 2.1 需求分析

### 2.1.1 用户愿景

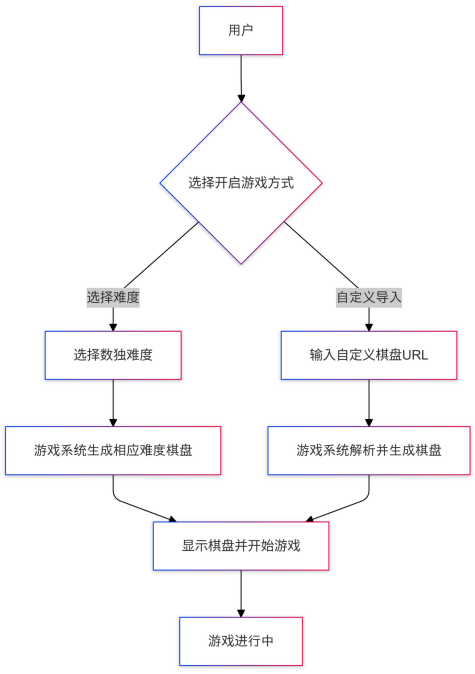
原数独项目的愿景是为用户提供一个易于操作、功能丰富的数独游戏平台。旨在满足不同水平用户（从初学者到资深玩家）对数独游戏的需求，通过直观的界面和实用的辅助功能，如提示功能，笔记功能，让用户能够轻松地进行数独游戏，享受解题的乐趣和挑战。

我们的用户需求是将其升级为数独乐乐，进一步升级游戏体验，让其更容易上手。用户可以通过下一步提示功能获取对应单元格的候选值，并可以控制提示的等级；还可以实现撤回、重做；此外在填入错误的数字导致游戏失败后，可以通过回溯一步跳转到先前的分支操作。用户还可以通过输入 url 的方式自定义棋盘，导入到数独乐乐中使用。

### 2.1.2 用例分析

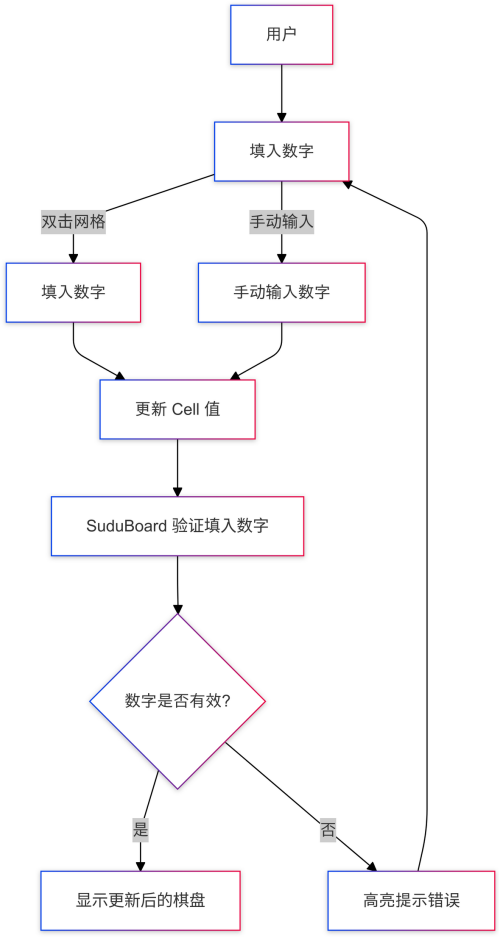
用例 1：选择不同难度棋盘或自定义导入棋盘开始游戏

在“ 数独乐乐” 游戏中，用户可以通过两种方式开启数独游戏。第一种方式是用户选择一个预设的数独难度，如简单、中等或困难，游戏系统会根据选择的难度自动生成相应难度的棋盘。第二种方式是用户通过提供一个 URL 来自定义导入棋盘 ,游戏系统会解析这个 URL 并生成对应的数独棋盘。生成棋盘后 ,系统会显示棋盘并允许用户开始游戏。



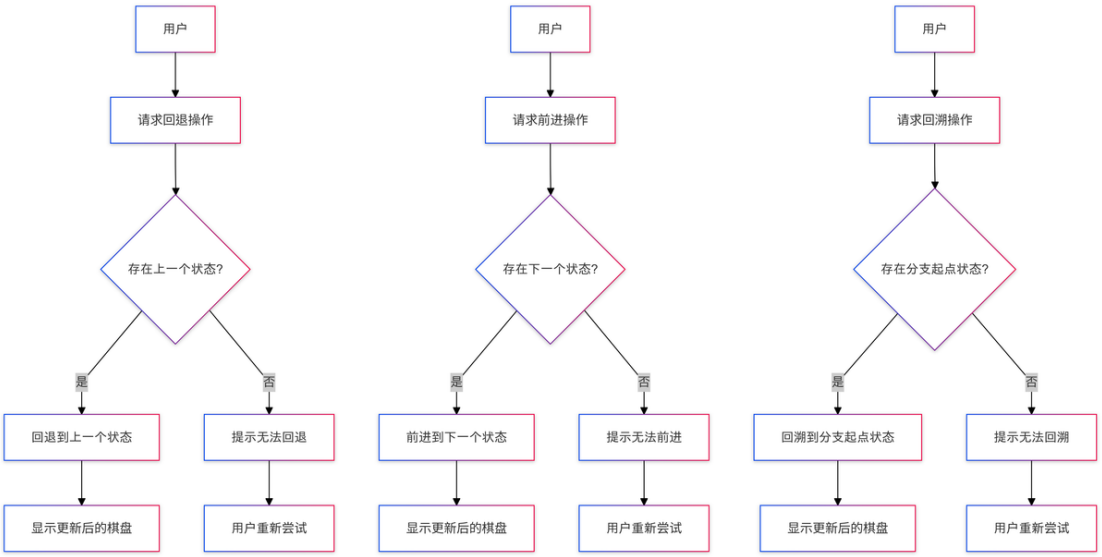
用例 2：填写数字

在“ 填入数字” 的用例中，用户首先在游戏界面选择填入数字的方式，可以是双击网格或者手动输入数字。用户通过游戏界面进行操作后，会触发填入数字的动作，这会涉及到 Cell 类，它负责接收行号、列号和数字，并更新单元格的值。接着，SuduBoard 类会验证填入的数字是否符合数独规则。如果数字有效，SuduBoard 类会更新棋盘状态，并通过游戏界面显示更新后的棋盘。如果数字无效，游戏界面会提示用户错误，并要求用户重新进行相关操作。这个用例确保了数独游戏的交互性和数独规则的正确性。



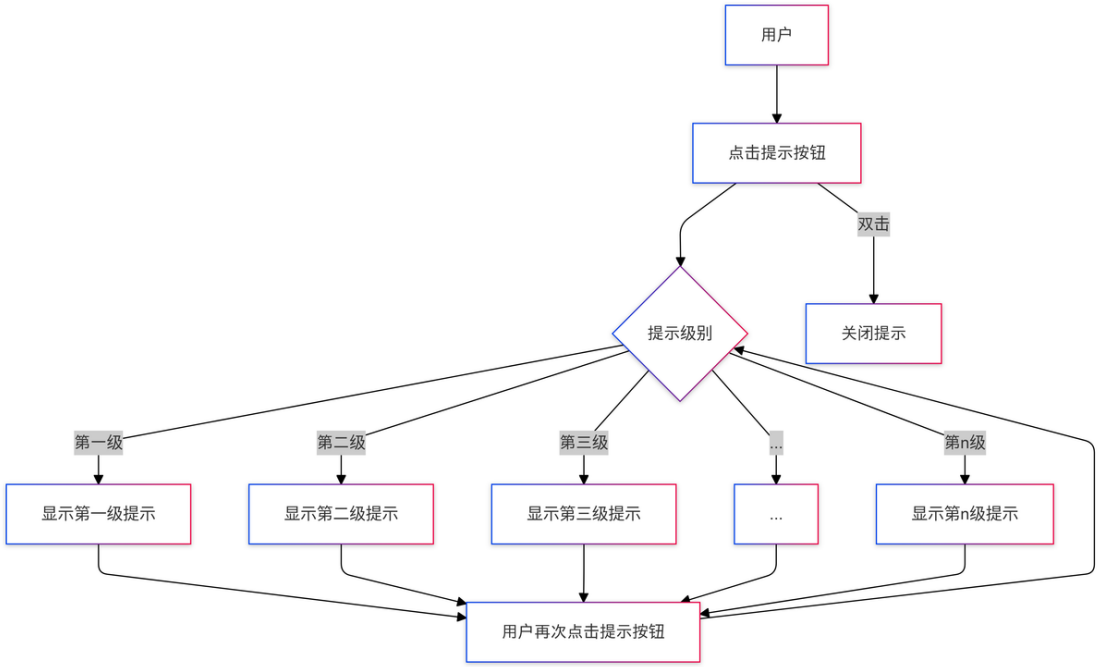

用例 3：回退、重做和回溯操作

在“ 数独乐乐”项目中，用户可以执行回退、前进和回溯操作来管理他们的游戏进度。当用户请求回退操作时，系统会检查是否存在上一个状态。如果存在，系统将回退到上一个状态并显示更新后的棋盘；如果不存在，系统会提示用户无法回退，并允许用户重新尝试。类似地，前进操作允许用户恢复之前回退的操作，系统会检查是否存在下一个状态，然后相应地前进或提示无法前进。回溯操作则是一步回到最近的分支起点，系统会检查是否存在分支起点状态，如果存在则回溯到该状态并更新棋盘，如果不存在则提示用户无法回溯，并允许用户重新尝试。这些操作为用户提供了灵活性，使他们可以在解题过程中进行自由探索尝试和更正错误，而不必担心不可逆的失误。




用例 4：开启提示

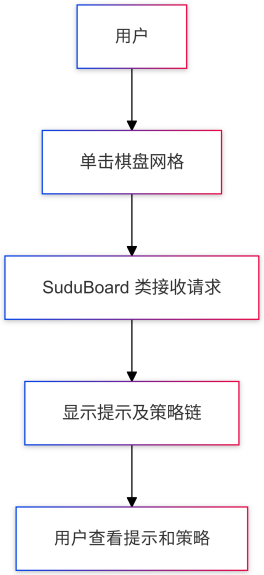
在“ 数独乐乐” 项目中，用户可以通过点击提示按钮来开启不同级别的提示。当用户首次点击提示按钮时，系统会显示第一级提示。如果用户需要更详细的提示，可以再次点击提示按钮，系统会根据用户的点击次数显示更高级别的提示，如第二级或第三级。每次点击都会递增提示级别，直到最高级别。如果用户双击提示按钮，系统会关闭提示功能。这个用例允许用户根据他们的需求和偏好，灵活地获取不同程度的帮助，同时在需要时能够快速关闭提示功能，继续独立解决数独谜题。




用例 5：查看提示的算法生成策略链

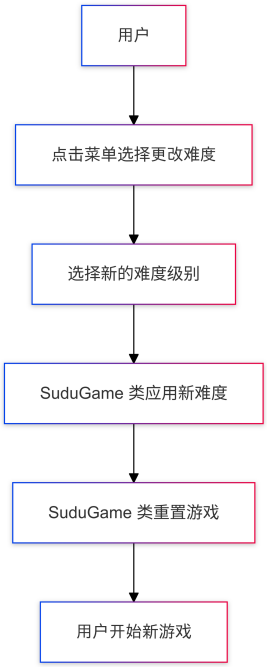
在“ 数独乐乐” 项目中，用户可以通过单击棋盘上的网格来查看与提示相关的算法策略链。当用户单击一个网格时，SuduBoard 类会接收到这个请求，然后展示了生成这个

提示所采用的算法策略链，让用户能够清楚地看到提示是如何被计算出来的。这个用例增强了游戏的互动性和教育性，帮助用户理解数独解题的逻辑和策略。




用例 6：更改数独游戏难度

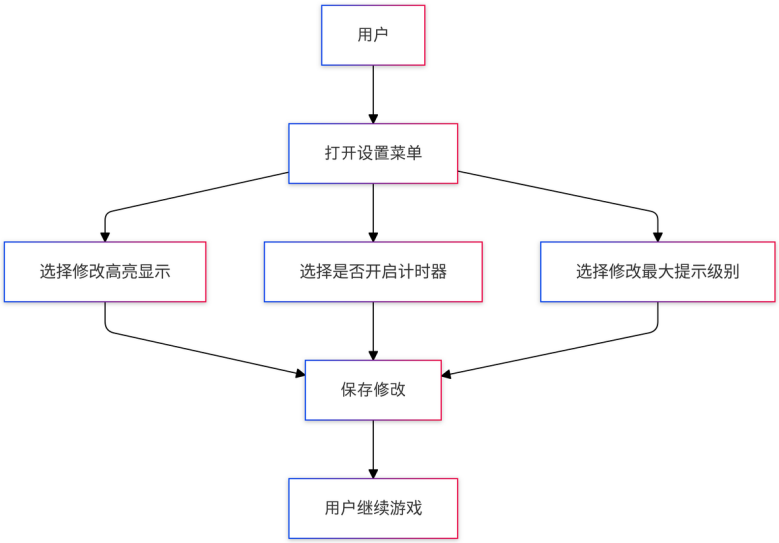
在“ 数独乐乐” 项目中，用户可以通过点击游戏菜单来更改数独游戏的难度。用户在菜单中选择一个新的难度级别，系统将重置游戏状态以匹配新难度。随后，游戏界面会显示一个与新难度级别相对应的数独棋盘，用户可以开始新游戏。这个用例允许用户根据自己的技能和偏好调整游戏挑战性，享受个性化的游戏体验。



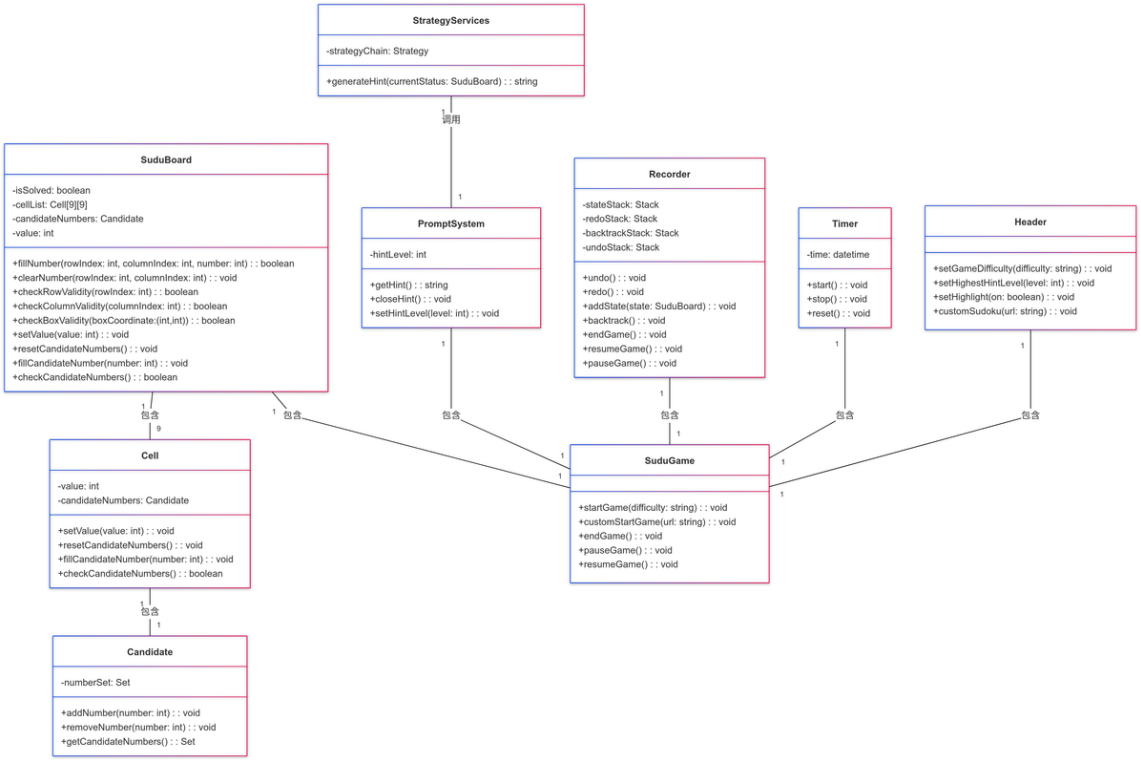

用例 7：修改数独游戏设置

在“ 数独乐乐” 项目中，用户可以通过打开设置菜单来在线修改数独游戏的设置。用

户可以选择修改是否开启高亮显示，这将由 Setting 类接收并更新设置，随后界面会应用这一新设置，改变棋盘的视觉反馈。同样，用户也可以选择修改最大提示级别，以影响游戏过程中的提示行为。此外，用户也可以选择是否开启游戏计时。这些修改可以即时生效，允许用户根据自己的偏好调整游戏的辅助功能，从而提升游戏体验。用户在完成设置修改后，可以继续他们的游戏。这个用例提供了个性化的游戏体验，使玩家能够根据自己的需求定制游戏环境。




## 2.1.3 领域模型




类图展示了“ 数独乐乐” 游戏的核心类及其关系和主要功能，清晰地展示了各个类之

间的关系和交互，为开发和理解项目提供了清晰的蓝图。这些类共同构成了数独游戏的核心逻辑，包括游戏的玩法、提示系统、状态记录和回溯功能等。以下是对类图中各个类及其方法的详细说明：

SuduBoard 类

- 属性：
  - `isSolved`：布尔值，表示数独板是否已解决。
  - `cellList`：二维数组，包含 9x9 的 Cell 对象。
  - `candidateNumbers`：Candidate 对象，存储每个单元格的候选数字。
  - `value`：整数，表示单元格的值。
- 方法：
  - `fillNumber(rowIndex, columnIndex, number)`：在指定位置填入数字，并返回操作是否成功。
  - `clearNumber(rowIndex, columnIndex)`：清除指定位置的数字。
  - `checkRowValidity(rowIndex)`：检查指定行是否有效。
  - `checkColumnValidity(columnIndex)`：检查指定列是否有效。
  - `checkBoxValidity(boxCoordinate)`：检查指定宫格是否有效。
  - `setValue(value)`：设置单元格的值。
  - `resetCandidateNumbers()`：重置单元格的候选数字。
  - `fillCandidateNumber(number)`：填入单元格的候选数字。
  - `checkCandidateNumbers()`：检查单元格的候选数字是否有效。

Cell 类

- 属性：
  - `value`：整数，单元格的值。
  - `candidateNumbers`，存储单元格的候选数字。
- 方法：
  - `setValue(value)`：设置单元格的值。
  - `resetCandidateNumbers()`：重置单元格的候选数字。
  - `fillCandidateNumber(number)`：填入单元格的候选数字。
  - `checkCandidateNumbers()`：检查单元格的候选数字是否有效。

Candidate 类

- 属性：

    - `numberSet`：集合，存储单元格的候选数字。
- 方法：
    - `addNumber(number)`：向候选数字集合中添加数字。
    - `removeNumber(number)`：从候选数字集合中移除数字。
    - `getCandidateNumbers()`：获取候选数字集合。

StrategyServices 类

- 属性：
    - `strategyChain`：Strategy 对象，用于生成提示。
- 方法：
    - `generateHint(currentStatus)`：根据当前数独板状态生成提示。

PromptSystem 类

- 属性：
    - `hintLevel`：整数，表示提示的等级。
- 方法：
    - `getHint()`：获取当前的提示。
    - `closeHint()`：关闭提示。
    - `setHintLevel(level)`：设置提示的等级。

Recorder 类

- 属性：
    - `stateStack`：栈，存储数独板的状态。
    - `redoStack`：栈，存储可以重做的宫格坐标和数值。
    - `backtrackStack`：栈，存储回退栈元素的长度。
    - `undoStack`：栈，存储回退的宫格坐标和数值。
- 方法：
    - `undo()`：回退到上一个状态。
    - `redo()`：重做上一次回退的操作。
    - `addState(state)`：添加当前状态到状态栈。
    - `backtrack()`：回溯到某个状态。
    - `endGame()`：结束游戏。
    - `resumeGame()`：恢复游戏。
    - `pauseGame()`：暂停游戏。

SuduGame 类

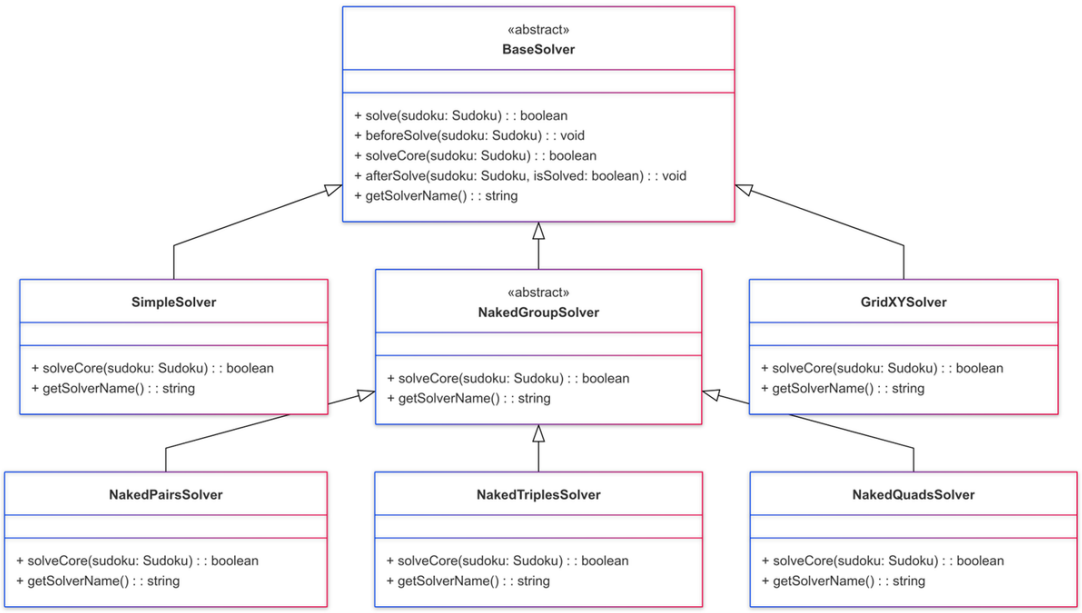
- 方法：
  - `startGame(difficulty)`：开始新游戏，根据难度设置数独板。
  - `customStartGame(url)`：通过 URL 自定义开始游戏。
  - `endGame()`：结束当前游戏。
  - `pauseGame()`：暂停当前游戏。
  - `resumeGame()`：从暂停中恢复游戏。

Timer 类

- 属性：
  - `time`：日期时间，记录游戏时间。
- 方法：
  - `start()`：开始计时。
  - `stop()`：停止计时。
  - `reset()`：重置计时器。

Header 类

- 方法：
  - `setGameDifficulty(difficulty)`：设置游戏的难度。
  - `setHighestHintLevel(level)`：设置提示的最高级别。
  - `setHighlight(on)`：设置是否高亮显示。
  - `customSudoku(url)`：通过 URL 自定义数独板。




这个类图展示了一个数独游戏求解器的策略模式实现，其中包含了多个抽象类和具体策略类，用于生成提示候选值，解决数独谜题。同时该类图也展示了数独求解器的层次结构和多态性，其中抽象类定义了通用接口，而具体类则实现了针对不同求解策略的核心算法。通过这种设计，数独游戏可以灵活地应用不同的求解策略来解决谜题。

BaseSolver 抽象类

- 描述：作为所有求解器策略的基类，定义了求解器的基本接口。
- 方法：
  - solve(sudoku: Sudoku): boolean：尝试解决数独谜题，返回是否解决成功。
  - beforeSolve(sudoku: Sudoku): void：在求解之前执行的准备操作。
  - solveCore(sudoku: Sudoku): boolean：核心求解逻辑，具体由子类实现。
  - afterSolve(sudoku: Sudoku, isSolved: boolean): void：在求解之后执行的收尾操作。
  - getSolverName(): string：返回求解器的名称。

SimpleSolver 类

- 描述：实现了基础求解策略，可能包含简单的数独解决算法。
- 继承自：BaseSolver
- 方法：
  - solveCore(sudoku: Sudoku): boolean：实现简单的核心求解逻辑。
  - getSolverName(): string：返回求解器名称。

NakedGroupSolver 抽象类

- 描述：定义了基于裸数（Naked Numbers）的求解策略，裸数是指在某一行、列或

宫格中唯一可能的数字。

- 继承自：BaseSolver
- 方法：
  - solveCore(sudoku: Sudoku): boolean：实现基于裸数的核心求解逻辑。
  - getSolverName(): string：返回求解器名称。

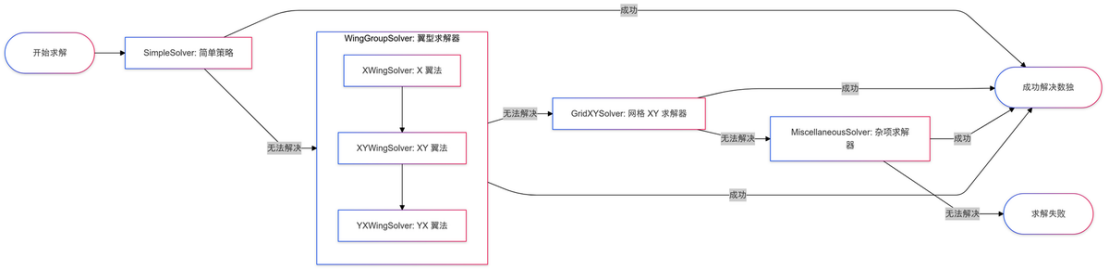
GridXYSolver 类

- 描述：可能实现了基于网格坐标的求解策略，例如对角线或特定行列的求解方法。
- 继承自：BaseSolver 或 NakedGroupSolver
- 方法：
  - solveCore(sudoku: Sudoku): boolean：实现基于网格坐标的核心求解逻辑。
  - getSolverName(): string：返回求解器名称。

NakedPairsSolver、NakedTriplesSolver、NakedQuadsSolver 类

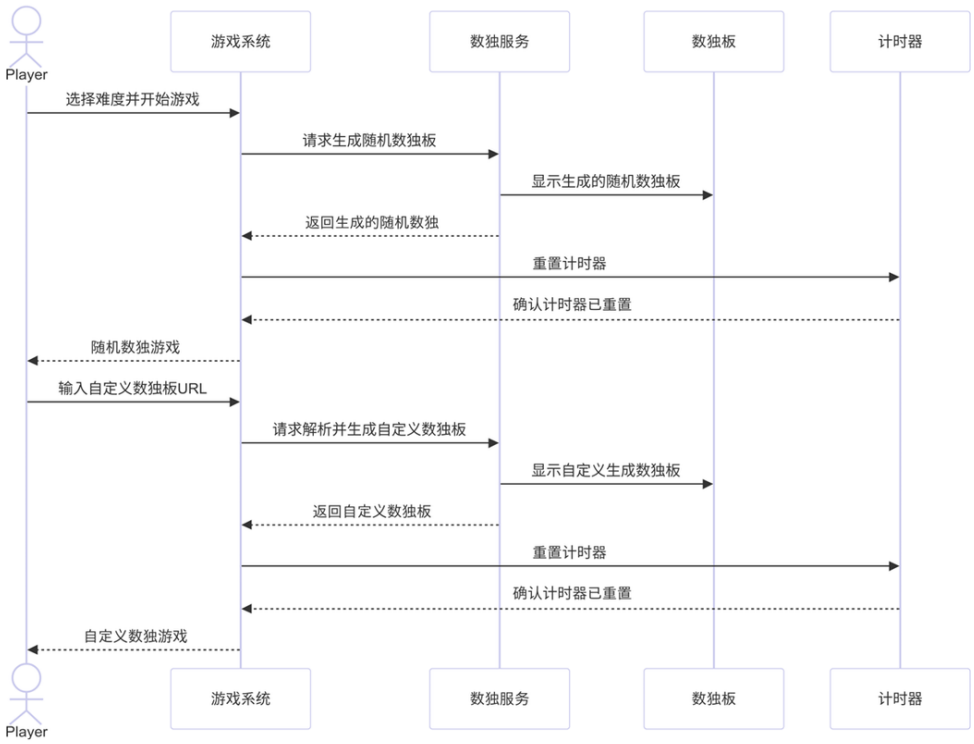
- 描述：这些类实现了更复杂的基于裸数的求解策略，分别对应于一对、三个或四个裸数的组合。
- 继承自：NakedGroupSolver
- 方法：
  - solveCore(sudoku: Sudoku): boolean :实现基于裸数对、三元组或四元组的核心求解逻辑。
  - getSolverName(): string：返回求解器名称。

## 2.1.4 行为模型

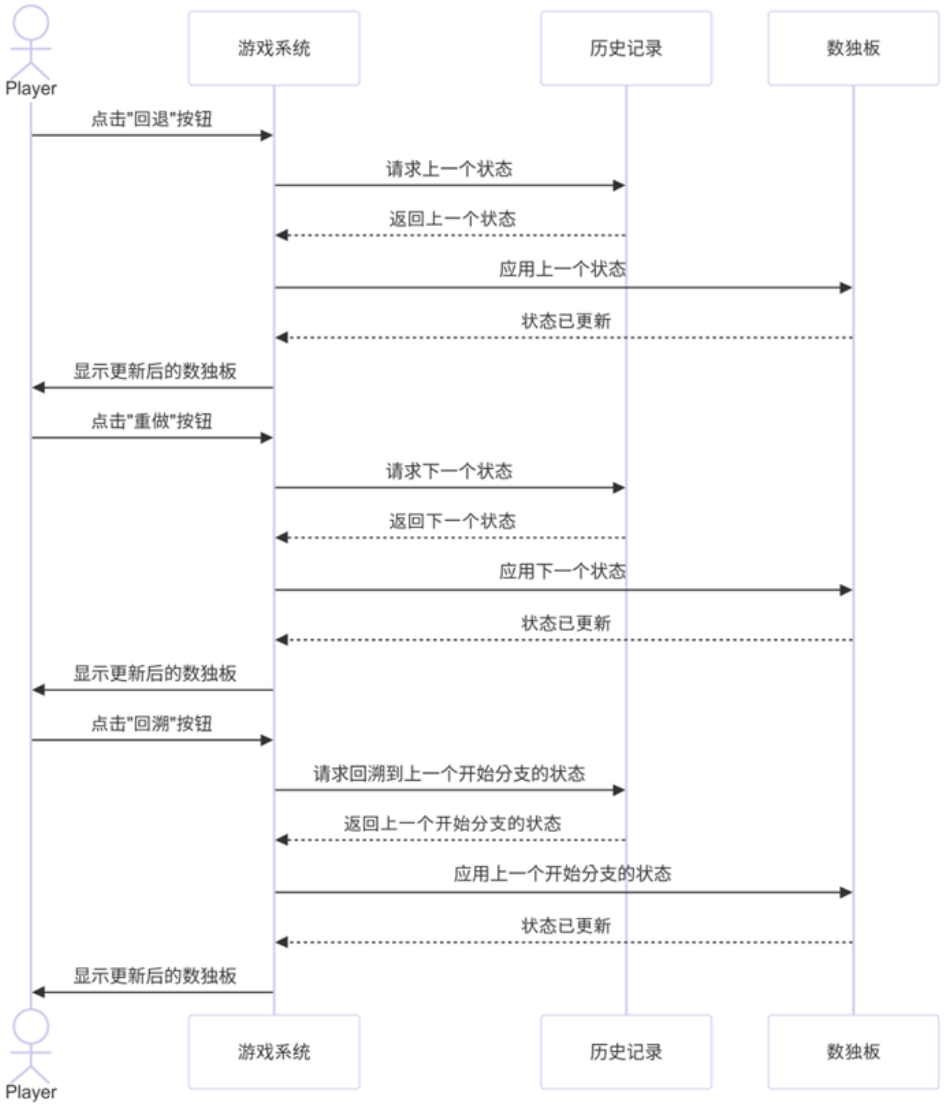



这个行为模型图展示了数独求解器在解决数独谜题时所采用的策略链。开始时，求解器首先尝试使用 SimpleSolver 应用简单策略来解决数独。如果简单策略无法解决，它将依次尝试更复杂的策略：首先是 WingGroupSolver 翼型求解器，包括 XWingSolver、XYWingSolver 和 YXWingSolver。如果这些策略仍然无法解决数独，求解器将尝试 GridXYSolver 网格 XY 求解器。如果网格 XY 求解器也失败，最后会尝试 MiscellaneousSolver 杂项求解器。在每个策略尝试后，如果数独被成功解决，则结

束求解过程；如果所有策略都无法解决数独，则求解失败。这个模型图清晰地描绘了求解器如何通过一系列策略逐步增加解决数独的复杂性，直到找到解决方案或确认无法解决。




这个用户行为时序图描述了“ 数独乐乐” 游戏中，玩家与游戏系统之间的交互流程。玩家首先选择游戏难度并开始游戏，这可以是随机数独游戏或自定义数独游戏。如果是随机数独游戏，游戏系统会向数独服务请求生成一个随机数独板，数独服务生成后返回给游戏系统，然后游戏系统显示这个随机数独板并重置计时器，以便玩家开始游戏。如果玩家选择自定义数独游戏，他们需要输入一个自定义数独板的 URL。游戏系统随后请求数独服务解析这个 URL 并生成自定义数独板。数独服务完成解析并生成数独板后，将其返回给游戏系统，游戏系统再显示这个自定义数独板并重置计时器，为玩家提供开始游戏的准备。在整个过程中，计时器组件负责跟踪玩家的游戏时间，确保在游戏开始时计时器被重置，以便准确记录游戏时长。这个时序图展示了游戏系统如何响应玩家的选择，以及数独服务如何支持游戏过程中的数独板生成和解析，同时确保计时器的正确管理。




这个时序图描述了“数独乐乐”游戏中玩家与游戏系统之间的交互流程，特别是关于数独棋盘状态的回退、重做和回溯操作。

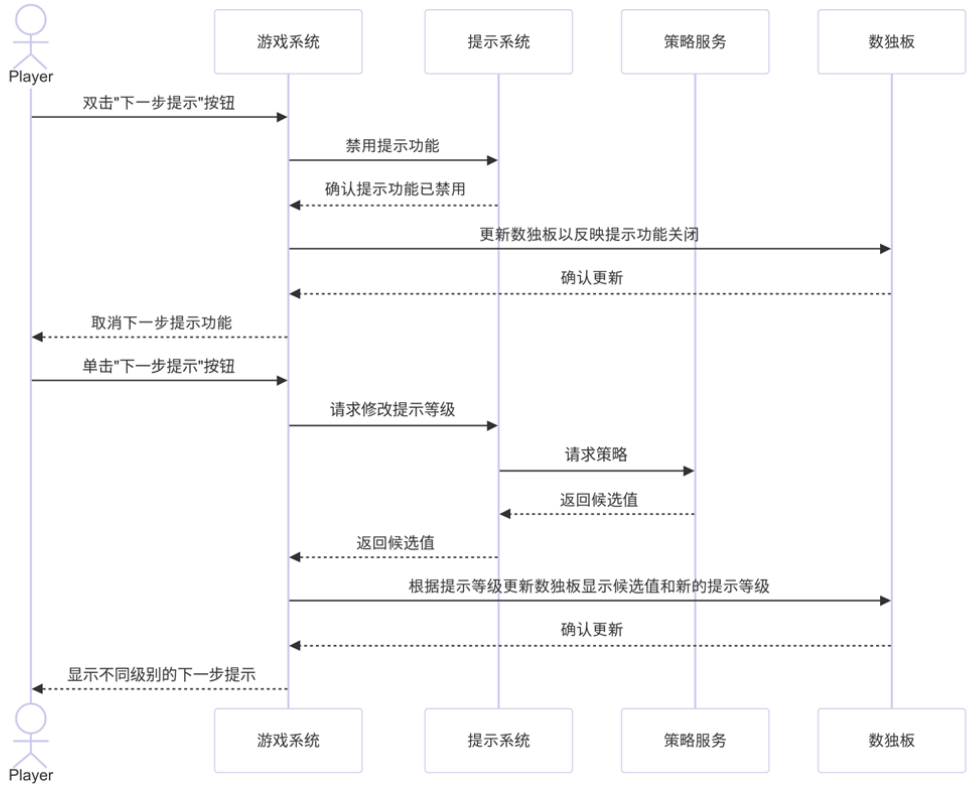
1. 回退操作：玩家点击“回退”按钮，游戏系统向历史记录请求上一个状态。历史记录返回上一个状态，游戏系统应用这个状态，数独板更新为之前的状态，然后显示更新后的数独板。

2. 重做操作：玩家点击“重做”按钮，游戏系统向历史记录请求下一个状态。历史记录返回下一个状态，游戏系统应用这个状态，数独板更新为之后的状态，然后显示更新后的数独板。

3. 回溯操作：玩家点击“回溯”按钮，游戏系统请求历史记录回到上一个开始分支的状态。历史记录返回上一个分支的状态，游戏系统应用这个状态，数独板更新为分支起点的状态，然后显示更新后的数独板。

这些操作允许玩家在游戏过程中灵活地撤销或重做他们的步骤，甚至回到之前的决策点，从而提供了一种动态的方式来探索不同的解题路径。通过这种方式，玩家可以更

自由地尝试不同的解决方案，而不必担心错误会导致游戏无法继续。




这个时序图描述了“ 数独乐乐” 游戏中玩家如何通过游戏系统与提示系统交互，以管理数独游戏的提示功能。

1. 禁用提示功能：玩家双击“ 下一步提示” 按钮，游戏系统接收到这个动作后，会向提示系统发送禁用提示功能的请求。提示系统确认提示功能已被禁用，并更新数独板以反映提示功能的关闭状态，然后游戏系统确认更新。

2. 修改提示等级：玩家单击“ 下一步提示” 按钮，游戏系统向提示系统发送请求以修改提示等级，提示系统根据这些值更新数独板显示的候选数字和新的提示等级，然后游戏系统确认更新。

3. 显示不同级别的下一步提示：游戏系统根据提示等级更新数独板，并显示不同级别的下一步提示，这可能包括显示候选数字、高亮显示可能的数字位置等。

整个流程允许玩家控制数独游戏中的提示功能，使他们能够根据自己的需要调整游戏的难度和提示的详细程度。通过这种方式，玩家可以更独立地解决数独谜题，或者在需要时获得额外的帮助。

## 2.2 项目关键问题

### 2.2.1 算法部分

如何动态切换算法策略？

如何在不同算法实现中保证逻辑一致性？

如何优雅地管理复杂的对象创建？

如何高效传递请求并避免循环调用？

### 2.2.2 探索回溯

如何保证撤回、重做与回溯不存在冲突？

使用什么数据结构来保证高性能？

如何同步撤回、重做的可用性？

## 2.3 项目核心思想

1. 面向对象设计（OOD）：项目中明确将不同的功能划分为不同的类或组件，模块化满足高内聚低耦合
2. 响应式编程与状态管理：store 来管理共享的状态的机制是 Svelte 中的响应式编程模式，自动响应并更新，结合了面向对象设计中的“ 观察者模式”
3. 事件驱动架构：在状态发生变化时，通过发布-订阅机制通知其他组件。这种方式解耦了组件之间的依赖。

## 2.4 设计原则和设计模式

1. 策略模式：定义一系列算法，将它们封装起来，并使它们可以相互替换，从而让算法独立于使用它的客户端变化。
2. 模板方法模式：在一个方法中定义算法的骨架，将某些步骤延迟到子类中实现，从而使得子类可以重新定义算法的某些步骤。
3. 工厂模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类，从而将对象的创建与使用分离。
4. 组合模式：将对象组合成树形结构以表示“ 整体-部分” 的层次结构，使客户端可以以一致的方式处理单个对象和对象组合。
5. 责任链模式：将多个处理对象串成一条链，当有请求时，沿着链传递请求，直到某个对象处理它为止。

## 2.5 设计详细说明

### 2.5.1 难度选择和自定义棋盘

不同难度的棋盘初始状态是通过调用外部库生成对应难度的棋盘实现的，自定义棋盘通过导入 URL，实则是对字符串做匹配，将满足要求的一部分数字作棋盘的初始化状态。将 URL 转化为 9*9 二维数组的函数如下所示：

```
export function decodeSencodeDirect(sencode) {
    // 提取 URL 中最后的数独字符串
    const sudokuString = sencode.split('=')[1];
    // 初始化一个 9x9 的二维数组
    const grid = Array.from({ length: 9 }, () => Array(9).fill(0));
    // 将数独字符串转换为 9x9 的二维数组
    for (let i = 0; i < sudokuString.length; i++) {
        const row = Math.floor(i / 9);
        const col = i % 9;
        grid[row][col] = parseInt(sudokuString[i], 10);
    }
    return grid;
}
```

### 2.5.2 策略集

在数独求解器的开发中，策略是核心驱动力。无论是基础的简单排除法，还是复杂的 XY-Wing 或 X-Wing 策略，每种策略都对应着特定的数独求解逻辑。然而，策略的实现和调用并非孤立存在，而是依赖于设计模式的支持。通过合理运用策略模式、工厂模式、模板方法、组合模式和责任链模式，数独求解器不仅能够高效地实现多种策略的动态调用，还能在代码结构上保持高度的灵活性和可扩展性。这些设计模式的结合，使得数独求解器能够优雅地应对复杂的数独问题，同时为未来的策略扩展提供了坚实的基础。

（1）策略模式

策略模式在数独求解器中用于支持多种求解策略的动态选择和切换。每个求解策略（如简单排除法、裸对法、XY-Wing 法等）都被封装在一个独立的类中，这些类继承自基类 BaseSolver，并实现了 solver 方法。通过策略模式，数独求解器可以在运行时动态选择和执行不同的求解策略，而无需修改客户端代码。

在代码中，BaseSolver 是一个抽象基类，定义了 solver 方法和 getSolverName 方法。具体的求解策略类（如 SimpleSolver、NakedPairsSolver 等）继承 BaseSolver 并实现自己的求解逻辑。这种设计使得新增求解策略时只需添加新的子类，而无需修改现有代码，符合开闭原则。

策略模式的核心思想是将算法的选择与使用分离，使得算法的变化独立于客户端代码。在数独求解器中，策略模式的应用使得求解器能够灵活应对不同的数独问题，同时保持了代码的可扩展性和可维护性。

```
class SudokuSolver {
    constructor(strategy) {
        this.strategy = strategy; // 当前策略
    }
    setStrategy(strategy) {
        this.strategy = strategy; // 动态切换策略
    }
    solve(sudokuGrid) {
        return this.strategy.solve(sudokuGrid);
    }
}
```

（2）工厂模式

工厂模式在数独求解器中用于集中管理求解策略的实例化过程。工厂模式的核心思想是将对象的创建与使用分离，使得客户端代码无需关心具体的实例化细节。

此外，工厂模式还支持动态扩展。如果需要新增求解策略，只需在 createSolver 函数中添加新的策略类实例，而无需修改其他代码。这种设计符合单一职责原则和依赖倒置原则，使得代码更加灵活和可维护。

```
function createSolver(type) {
    switch (type) {
        case 'SimpleSolver':
            return new SimpleSolver();
        case 'NakedPairsSolver':
            return new NakedPairsSolver();
        // 其他求解器类型...
        default:
            throw new Error('Unknown solver type');
    }
}
```

（3）模板方法模式

模板方法模式在数独求解器中用于定义求解策略的执行流程。基类 BaseSolver 定义了 solver 方法的模板，子类必须实现该方法以完成具体的求解逻辑。通过模板方法模式，数独求解器确保了所有求解策略的执行流程一致，同时允许子类灵活实现具体的求解逻辑。

在代码中，BaseSolver 的 solver 方法是一个抽象方法，子类（如 SimpleSolver、NakedPairsSolver 等）必须实现该方法。这种设计使得算法的结构固定，但具体的步骤可以灵活变化。模板方法模式的核心思想是将通用的流程封装在基类中，子类只需关注具体实现。

模板方法模式的应用使得数独求解器的代码更加结构化和可复用。通过将通用的执行流程抽象到基类中，子类可以专注于实现具体的求解逻辑，避免了代码重复，符合 DRY（Don't Repeat Yourself）原则。

```
class BaseSolver {
    solve(sudokuGrid) {
        this.preprocess(sudokuGrid); // 通用的前置处理
        if (this.coreSolve(sudokuGrid)) { // 核心求解逻辑（由子类实现）
            this.postprocess(sudokuGrid); // 后置处理
            return true;
        }
        return false;
    }
    preprocess(sudokuGrid) {
        console.log("Preprocessing the Sudoku grid...");
    }
    coreSolve(sudokuGrid) {
        throw new Error("coreSolve() must be implemented by subclasses");
    }
    postprocess(sudokuGrid) {
        console.log("Postprocessing the Sudoku grid...");
    }
}
```

（4）组合模式

组合模式在数独求解器中用于将多个求解策略组合在一起，统一执行。通过组合模式，数独求解器可以将多个策略类视为一个整体，简化了客户端代码。

组合模式的核心思想是将对象组合成树形结构，以表示“ 部分-整体” 的层次关系。在数独求解器中，组合模式的应用使得多个求解策略可以被统一管理和执行，客户端代码只需调用 runAllSolversOnce 函数即可执行所有策略，而无需关心具体的策略组合。

此外，组合模式还支持动态调整策略组合。如需调整策略的执行顺序或增减策略，只需修改所对应策略的数组，无需修改其他代码，使数独求解器更加灵活和可扩展。

```
class WingGroupSolver extends SolverComponent {
    constructor() {
        super("Wing Group Solver");
        this.children = [];
    }
    add(solver) {
        this.children.push(solver);
    }
    remove(solver) {
        this.children = this.children.filter(child => child !== solver);
    }
    solve(sudoku) {
        for (const solver of this.children) {
            const solved = solver.solve(sudoku);
            if (solved) return true;
        }
        return false;
    }
    display(indent = 0) {
        console.log(`${" ".repeat(indent)}+ ${this.name}`);
        this.children.forEach(child => child.display(indent + 2));
    }
}
```

（5）责任链模式

责任链模式在数独求解器中用于将多个求解策略串联起来，依次执行。在runAllSolversOnce 函数中，每个求解策略类都是一个处理者，依次对数独进行处理。如果某个策略对数独进行了修改，修改后的数独会传递给下一个策略继续处理。

责任链模式的核心思想是将请求的发送者和接收者解耦，允许多个对象有机会处理请求。在数独求解器中，责任链模式的应用使得每个求解策略类只需关注自己的处理逻辑，而无需知道其他策略的存在。这种设计使得代码更加模块化和可维护。

此外，责任链模式还支持动态调整处理链的顺序或增减处理者。如果需要调整策略的执行顺序或增减策略，只需修改 runAllSolversOnce 函数中的数组即可，而无需修改其他代码。这种设计使得数独求解器更加灵活和可扩展。

```
function runAllSolversOnce(sudoku) {
    let funcs = [
        new SimpleSolver(),
        new NakedPairsSolver(),
        new NakedTriplesSolver(),
        new NakedQuadsSolver()
        // 更多策略...
    ];
    for (let func of funcs) {
        let clone = cloneSudoku(sudoku);
        sudoku = func.solver(sudoku);
        if (!checkSudokuIsLegal(sudoku)) {
            console.log(`使用 ${func.getSolverName()} 策略出现错误`);
            sudoku = clone;
            continue;
        }
    }
    return sudoku;
}
```

### 2.5.3 下一步提示

单击提示按钮，会出现一级提示，此后每点一次会增加提示等级（提示等级就是单元格能显示的最大候选值数量），当超过设置的等级提示后，会回到一级提示。此外双击提示按钮，会关闭提示，即单元格里的候选值会消失。

提示功能是通过调用策略集生成候选值实现的，候选值显示的具体实现是设计一个 candidates 数组，用于存放每一次调用策略集生成的结果，根据 candidates 的每个位置

上的候选值数量决定前端显示的结果。此外，对于探索回溯失败的候选值禁用情况，具体实现是在有多个候选值情况下，维护一个禁用集合记录了该单元格上候选值的情况，用来控制前端显示。

### 2.5.4 用户填入数独

在进行提示操作后，单元格会出现候选数字，如果是单个候选值，可以双击单元格将数字填入棋盘，当有多个候选值时，会在单元格中显示多个候选值，用户可以直接单击某个数字填入棋盘，也可以通过输入某个数字作为该单元格填入棋盘的值。

具体实现是通过控制前端组件的响应逻辑来实现不同情况下单元格数字的填入。

### 2.5.5 撤回和重做

在进行游戏操作（如填入或删除数字）后，可执行撤回（Undo）操作。当对某单元格写入新值时（删除即将其写为 0 ）时，当前位置和原来的值会被记录到 undoStack 中。undoStack 是一个使用数组实现的栈。

```
set: (pos, value, branch = false) => {
                userGrid.update($userGrid => {
                    // 记录当前值以支持撤回
                    const previousValue = $userGrid[pos.y][pos.x];
                    undoStack.push({ pos, value: previousValue });
                    console.log({ pos, value: previousValue });
                    if (branch) {
                        branchStack.push(undoStack.length - 1);
                        branchwrongStack.push({ pos, value: value });
                    }
                    // 清空重做栈
                    redoStack.length = 0;
                    // 设置新值
                    $userGrid[pos.y][pos.x] = value;
                    updateIsUndoable();
                    updateisRedoable();
                    updateisBackToBranchable();
                    return $userGrid;
                });
            },
```

当需要撤回时，可以从 undoStack 中取出上一个状态并恢复。

```
undo: () => {
                if (undoStack.length > 0) {
                    const { pos, value } = undoStack.pop();
                    userGrid.update($userGrid => {
                        // 记录当前值以支持重做
                        const currentValue = $userGrid[pos.y][pos.x];
                        redoStack.push({ pos, value: currentValue });
                        // 恢复之前的值
                        $userGrid[pos.y][pos.x] = value;
                        updateIsUndoable();
                        updateisRedoable();
                        updateisBackToBranchable();
                        return $userGrid;
                    });
                    // 检查是否需要弹出 branchStack 中的标记
                    if (branchStack.length > 0 && undoStack.length <=
branchStack[branchStack.length - 1]) {
                        branchStack.pop();
                    }
                }
            },
```

重做功能（Redo）需要在撤回操作后实现，在执行撤回操作后，系统将当前位置和撤回前的值存储在 redoStack 中，在执行重做操作时，将弹出 redoStack 的最新值，重新写入 undoStack ,并对应修改棋盘。此外 ,在每次及逆行新的操作后， redoStack 会被清空。

```
redo: () => {
                if (redoStack.length > 0) {
                    const { pos, value } = redoStack.pop();
                    userGrid.update($userGrid => {
                        // 记录当前值以支持撤回
                        const currentValue = $userGrid[pos.y][pos.x];
                        undoStack.push({ pos, value: currentValue });
                        // 恢复之前的值
                        $userGrid[pos.y][pos.x] = value;

                        updateIsUndoable();
                        updateisRedoable();
                        updateisBackToBranchable();
                        return $userGrid;
                    });
                }
            },
```

### 2.5.6 探索和回溯

若根据现有策略已经无法得到任何单元格的唯一候选值，用户可以再次点击提示按钮开启二级提示，此时系统将在单元格内显示多个候选值，用户可以通过单击某个候选值或单击下方数字进行探索，我们将其定义为分支操作( 即写入有多个候选值的单元格 )。

在写入新值时，系统会判断该次写入是否为分支操作，如果是则将 branch 参数设置为 true，在 branchStack 栈中记录当前 undoStack 的位置。在回溯操作中，将会调用多次撤回，直到 undoStack 回退到分支前的位置。调用回溯操作时，系统将先前错误的位置和值存储到 branchwrongStack 中，以实现回溯后将错误数字标灰。

```
const branchIndex = branchStack.pop();
                    let { pos, value } = { pos: 0, value: 0 };
                    while (undoStack.length > branchIndex) {
                        ({ pos, value } = undoStack.pop());
                        userGrid.update($userGrid => {
                            const currentValue = $userGrid[pos.y][pos.x];
                            redoStack.push({ pos, value: currentValue });
                            $userGrid[pos.y][pos.x] = value;
                            return $userGrid;
                        });
                    }
```

此外，若用户发起多次撤回操作，导致 undoStack 的值小于 branchStack 记录的数值，也就是用户已经将 branchStack 记录的分支操作撤回了，此时应删除 branchStack 中对应的位置。

### 2.5.7 胜利/失败条件

检查棋盘每个单元格值是否填入且是否已经符合数独基本规则，如果当前棋盘的状态符合这两个条件，则表明数独棋盘已经找到了正确答案；如果在某一步棋盘状态不符合数独基本规则，则游戏会显示失败，对于冲突的部分会变红作为提醒，特别是但利探索分支的时候，如果提示功能无法在继续找到合适候选值也会显示 No Solution 提示，表示当前分支探索失败，游戏失败。

## 2.6 未来改进方向

1. 增强策略可视化与追踪性：目前策略实施中，未能清晰展现单元格候选值的减少由哪些单元格所致。后续计划为策略增加精细追踪体系，构建专门数据结构，记录如单元格坐标、操作时间戳等关键信息，精准回溯源头。
2. 深化策略模式多态应用：项目虽部分运用策略模式，但多态性挖掘不足，常借条件语句管控操作类型。未来将重构策略类层级，依操作共性抽象基类或接口，像不同运算策略以“运算策略”接口统领，各具体策略类按需实现。

3. 构建组件继承复用体系：当下项目继承、接口运用稀缺，组件独立性强，复用、拓展受限。下一步要系统梳理功能组件，依共性提取抽象基类，如各类图形绘制组件以“ 图形” 抽象基类整合通用要素，供子类继承拓展。

# 3 成员贡献表

| 学号 | 姓名 | 贡献率 | 制品与贡献 |
| --- | --- | --- | --- |
| 24214557 | 曾欣祺 | 25% | 1. 撤回、重做和回溯 100%<br>2. 功能完善和 debug、前后端联调 25%<br>3. 演示视频录制 100%<br>4. 中期汇报 25%<br>5. 项目文档 25% |
| 24214481 | 欧宏骏 | 25% | 1. 下一步提示 50%<br>2. 候选值显示和提示等级的设置 100%<br>3. 功能完善和 debug、前后端联调 25%<br>4. 中期汇报 25%<br>5. 项目文档 25% |
| 24214435 | 李程俊 | 25% | 1. 下一步提示 50%<br>2. 候选值禁用和无解的提示 100%<br>3. 功能完善和 debug、前后端联调 25%<br>4. 中期汇报 25%<br>5. 项目文档 25% |
| 24214341 | 王超 | 25% | 1. 数独策略类的设计与实现 100%<br>2. 数独策略的调用逻辑实现 100%<br>3. 功能完善和 debug，前后端联调 25%<br>4. 中期汇报 25%<br>5. 项目文档 25% |