学号	姓名	贡献率	制品与贡献
24214558	曾祯	50%	1.需求规格/用例分析 50% 2.数独code导入实现 100% 3.探索回溯实现 100% 4.策略集成：算法策略实现+正确性测试 100%
24214528	伍庆富	50%	1.需求规格/用例分析 50% 2.下一步提示 100% 3.策略集成：算法策略调用逻辑+性能测试 100%

一、需求规格

1.1 用例分析

本项目的用例如下：

（1）用例名称：生成数独题目

- **主要参与者**：用户
- **目标**：为用户生成不同难度的数独题目
- **基本流程**：
    1. 用户选择难度（简单、中等、困难等）。
    2. 系统根据选择生成一个新的数独题目。
    3. 用户可以开始游戏。
- **扩展功能**：
    - 支持用户通过分享链接解析数独题目URL并导入。
    - 支持用户通过sudokuwiki中的题目URL导入（new）

（2）用例名称：填写数独游戏

- **主要参与者**：用户
- **目标**：在提供的数独界面上完成题目的填写
- **基本流程**：
    1. 用户选择一个单元格进行填写。
    2. 系统高亮显示相关行、列、区域的冲突单元格（如有）。
    3. 用户可以填写候选值或直接填写答案。
    4. 填写完成后，用户提交结果。
- **扩展功能**：
    1. 提供Redo/Undo功能，用户可以撤销之前填写的内容取消之前的撤销步骤（new）
    2. 提供回溯功能，当用户在分支点进行探索出现错误答案时，可以回溯到当时的分支点（new）
    3. 采用多种策略进行提示的生成，用户点击提示单元格时可以看到当前单元格提示的依据（new）

(3) 用例名称：游戏结果记录

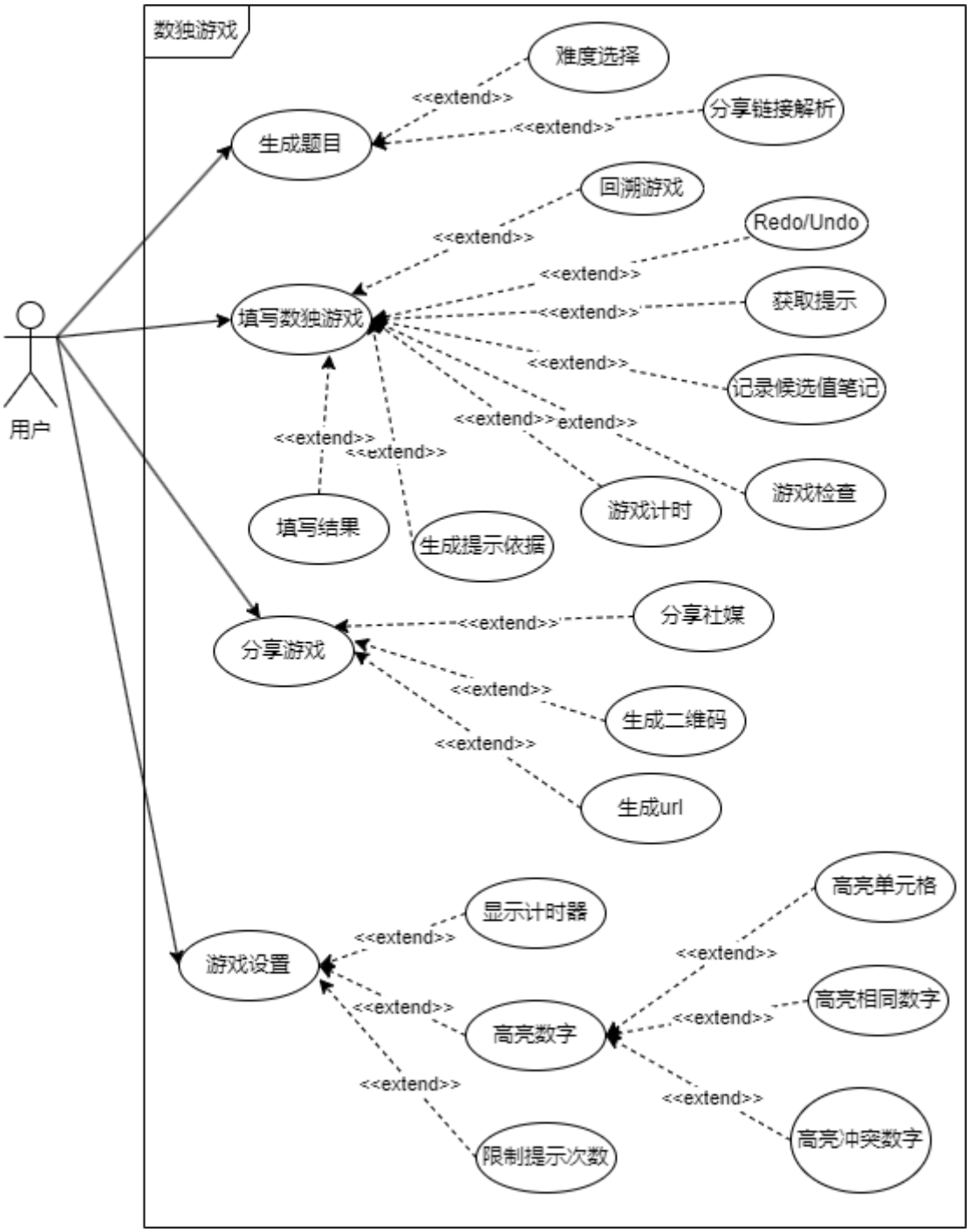
- **主要参与者**：用户
- **目标**：记录并显示游戏结果
- **基本流程**：
    1. 用户完成数独填写并提交结果。
    2. 系统检测是否满足所有规则。
    3. 如果游戏胜利，记录完成时间和难度等级。
    4. 系统提示用户可以分享结果到社交平台。
- **扩展功能**：
    - 提供实时计时器功能，记录游戏时长。

(4) 用例名称：分享游戏

- **主要参与者**：用户
- **目标**：分享游戏题目
- **基本流程**：
    1. 用户点击“分享”按钮。
    2. 系统生成一个唯一链接或二维码。
    3. 用户可以将链接/二维码分享到社交平台。
- **扩展功能**：
    - 支持生成社交媒体优化的分享内容（包含图片、文本等）。

(5) 用例名称：游戏设置

- **主要参与者**：用户
- **目标**：调整游戏相关设置
- **基本流程**：
    1. 用户进入“设置”界面。
    2. 系统提供可选项：调整高亮设置、限制提示次数、启用/禁用候选值功能等。
    3. 用户保存设置并返回游戏界面。
- **扩展功能**：
    - 提供实时预览功能（例如高亮效果的预览）。



1.2 领域模型

(1) SudokuGame（数独游戏核心类）

- **职责**:
  - 作为整个数独游戏的核心类，负责协调游戏的主要逻辑。
    - 包含以下关键属性:
      - `difficulty`: 当前数独的难度。
      - `paused`: 标志游戏是否处于暂停状态。
      - `hintsAvailable`: 剩余的提示次数。
      - `board`: 关联 `Board` 类（即棋盘对象），管理数独棋盘的状态。
      - `settings`: 关联 `Settings` 类，用于处理用户的配置。

        - 包含以下关键方法：
            - `pause() / resume()`：用于暂停或恢复游戏。
            - `getHints()`：提供提示功能。
            - `noteCandidates()`：记录候选数字。
            - `createSudoku()`：生成数独题目。
- **分析**：
    - `SudokuGame` 是领域模型的核心协调类，负责将不同的功能模块（棋盘、设置、计时器等）整合起来。
    - 其设计体现了“高内聚、低耦合”的原则，将与游戏整体相关的逻辑集中在一个类中。

(2) Grid（棋盘）

- **职责**：
    - 数独棋盘的抽象，由 81 个 `Cell` 对象组成。
    - 关键属性：
        - `cells`：包含 81 个单元格（`Cell`）对象。
    - 关键方法：
        - `set()`：用于设置或更新棋盘单元格的值。
- **分析**：
    - 棋盘是数独游戏的核心领域对象。通过将棋盘分解为 81 个 `Cell` 对象，实现了棋盘的模块化设计。

(3) Cell（单元格）

- **职责**：
    - 数独棋盘上的每个单元格对象，包含单个数字或候选值。
    - 关键属性：
        - `value`：当前单元格的数字值。
        - `cellX` / `cellY`：单元格的坐标。
        - `candidates`：与 `Candidates` 对象关联，用于存储候选值。
        - `disabled`：标志单元格是否不可编辑。
        - `selected`：标志单元格是否被选中。
        - `usernumber`：用户填入的数字。
    - 关键方法：
        - `setValue(pos, num)`：设置单元格的值。
        - `clear()`：清除单元格内容。
- **分析**：
    - `Cell` 类实现了对单个单元格的精细化管理。其设计体现了单一职责原则，专注于处理单元格相关的逻辑。

(4) Candidates（候选值）

- **职责**：
  - 存储单元格的候选值列表。
  - 关键属性：
    - `candidates`：候选值数组。
- **分析**：
  - 将候选值单独建模，简化了 `Cell` 类的设计，并为候选值的算法实现提供了更大的灵活性。

(5) Timer（计时器）

- **职责**：
  - 实现游戏计时功能。
  - 关键属性：
    - `timeBegan`：计时开始时间。
    - `timeStopped`：计时停止时间。
    - `timeInterval`：当前计时间间隔。
    - `stoppedDuration`：计时暂停的时间。
    - `running`：标志计时器是否正在运行。
  - 关键方法：
    - `start()`：开始计时。
    - `stop()`：停止计时。
    - `reset()`：重置计时器。
- **分析**：
  - 计时器是游戏功能的重要辅助模块，其实现与核心逻辑分离，符合高内聚低耦合的原则。

(6) Settings（设置）

- **职责**：
  - 存储并管理游戏的用户设置。
  - 关键属性：
    - `displayTimer`：是否显示计时器。
    - `highlightCells`：是否高亮单元格。
    - `highlightSame`：是否高亮相同数字。
    - `highlightConflicting`：是否高亮冲突数字。
    - `minHintLevelEachStep`：每步最小提示等级。
- **分析**：
  - 通过单独的设置类，简化了游戏核心逻辑的复杂性，使用户设置与游戏逻辑解耦。

(7) StrategyRegister（解题策略）

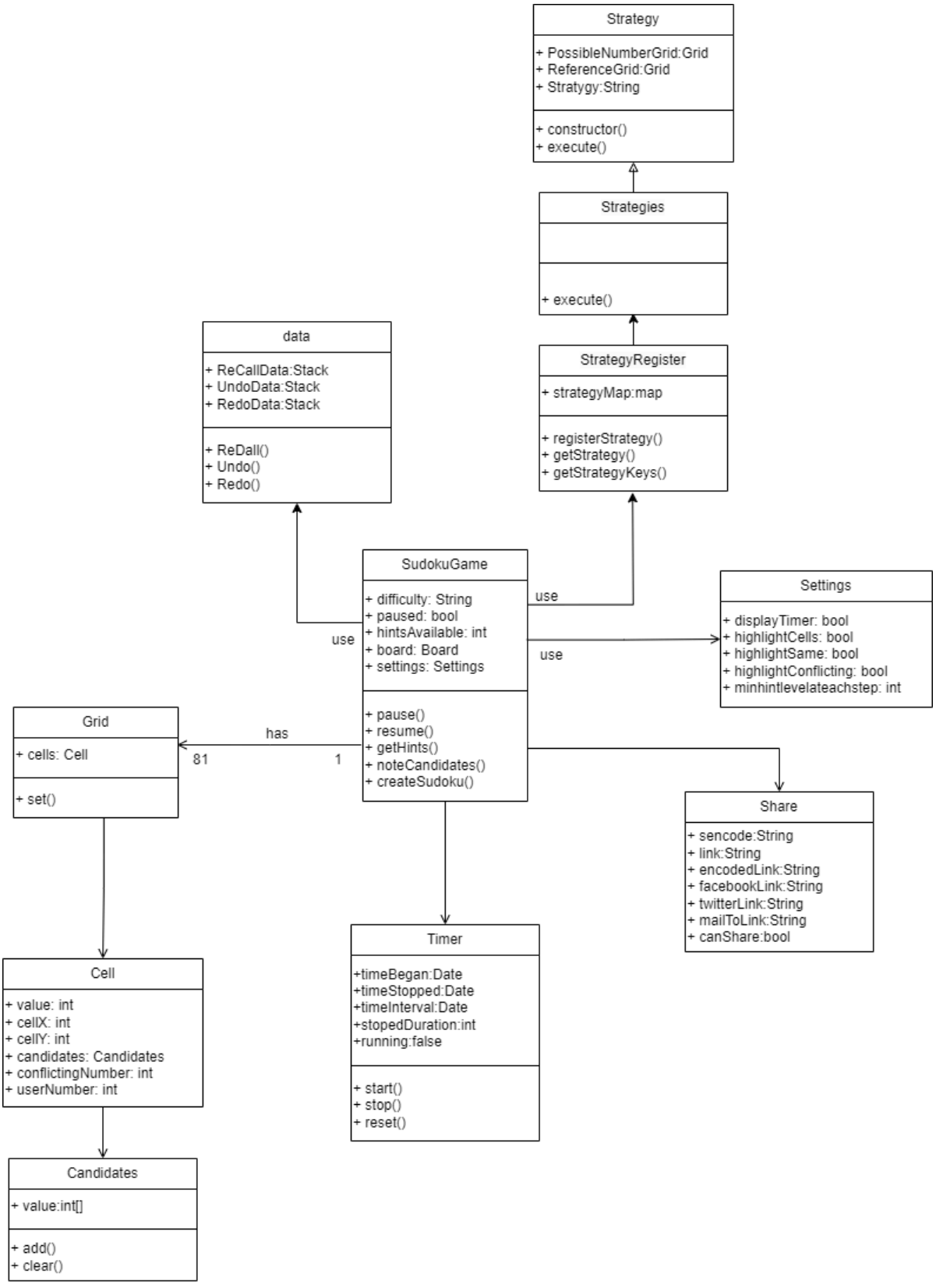
- **职责**：
  - 管理数独解题算法的实现。
  - 关键类：
    - `strategyRegister`：用于注册新策略。
  - 实现类：
    - `strategy`：策略基类。
    - `strategies`：继承策略基类后实现的具体算法类。
- **分析**：
  - 将数独解题算法单独建模，便于扩展新策略，体现了开放封闭原则。

(8) Share（分享功能）

- **职责**：
  - 实现游戏结果的分享功能。
  - 关键属性：
    - `sencode`：数独题目的编码。
    - `link`：分享链接。
    - `encodedLink`：编码后的分享链接。
    - `facebookLink` / `twitterLink` / `mailToLink`：对应不同平台的分享链接。
    - `canShare`：是否允许分享。
- **分析**：
  - 分享功能的单独建模提高了代码的模块化，便于后续扩展分享渠道。

(9) data（撤销/回溯功能）

- **职责**：
  - 存储并管理用户操作记录，用于实现撤销和回溯功能。
  - 关键属性：
    - `ReCallData`：记录当前的游戏状态。
    - `UndoData`：记录用户的撤销操作栈。
    - `RedoData`：记录用户的重做操作栈。
- 关键方法：
  - `ReCall()`：恢复到指定状态。
  - `Undo()`：撤销一步操作。
  - `Redo()`：重做一步操作。
- **分析**：
  - 通过回溯和撤销功能的建模，增强了用户体验，也为实现更复杂的用户交互提供了可能。


Strategy
+ PossibleNumberGrid:Grid
+ ReferenceGrid:Grid
+ Stratygy:String
+ constructor()
+ execute()
Strategies
+ execute()
StrategyRegister
+ strategyMap:map
+ registerStrategy()
+ getStrategy()
+ getStrategyKeys()
data
+ ReCallData:Stack
+ UndoData:Stack
+ RedoData:Stack
+ ReDall()
+ Undo()
+ Redo()
SudokuGame
+ difficulty: String
+ paused: bool
+ hintsAvailable: int
+ board: Board
+ settings: Settings
+ pause()
+ resume()
+ getHints()
+ noteCandidates()
+ createSudoku()
use
use
use
Settings
+ displayTimer: bool
+ highlightCells: bool
+ highlightSame: bool
+ highlightConflicting: bool
+ minhintlevelateachstep: int
Grid
+ cells: Cell
+ set()
has
81
1
Share
+ sencode:String
+ link:String
+ encodedLink:String
+ facebookLink:String
+ twitterLink:String
+ mailToLink:String
+ canShare:bool
Timer
+timeBegan:Date
+timeStopped:Date
+timeInterval:Date
+stopedDuration:int
+running:false
+ start()
+ stop()
+ reset()
Cell
+ value: int
+ cellX: int
+ cellY: int
+ candidates: Candidates
+ conflictingNumber: int
+ userNumber: int
Candidates
+ value:int[]
+ add()
+ clear()

二、软件设计规格

2.1 系统技术架构

1. 系统架构概述

该数独游戏是一个基于前端技术栈的客户端应用程序，使用 **Svelte** 构建用户界面，**JavaScript** 提供逻辑和状态管理，具备以下特点：

- **组件化架构**：通过 Svelte 的单文件组件开发（`.svelte`），实现页面和功能的模块化。
- **状态管理**：通过 JavaScript 编写的 `stores` 模块，管理全局和局部状态（如网格数据、计时器、候选值）。
- **响应式更新**：利用 Svelte 的响应式特性，自动处理数据与视图的双向绑定。
- **模块化代码组织**：通过合理划分文件夹（如 `components`、`stores`、`utils` 等），清晰地管理项目的功能逻辑。
- **本地存储**：在游戏保存与恢复时，可能利用浏览器的 LocalStorage 来持久化状态。

---

2. 系统架构分层

按照现代前端架构惯例，数独游戏的系统架构可以分为以下几层：

(1) 表现层（View Layer）

- **技术栈**：Svelte 组件。
- **职责**：
  - 提供用户界面（UI），如数独网格、候选值选择、计时器、提示功能等。
  - 与用户交互，通过事件监听（如点击、键盘输入）触发操作。
  - 调用 `stores`（状态管理）来读取和更新数据。
- **文件对应**：
  - **主要目录**：`/components`
  - **示例组件**：
    - `Board`（数独网格）
    - `Controls`（控制面板，包括按钮、键盘输入等）
    - `Header`（顶栏按钮，如新建游戏、难度选择）
    - `Model`（弹窗，如分享、设置、游戏结束）
  - **样式文件**：`styles/global.css`，提供全局样式支持。

(2) 状态管理层（State Management Layer）

- **技术栈**：Svelte `stores`（Svelte 原生的状态管理工具）。
- **职责**：
  - 管理全局状态（如网格数据、当前难度、计时器状态）。
  - 提供订阅和响应式数据绑定，通知界面组件实时更新。
  - 将状态持久化到本地（如 LocalStorage）。

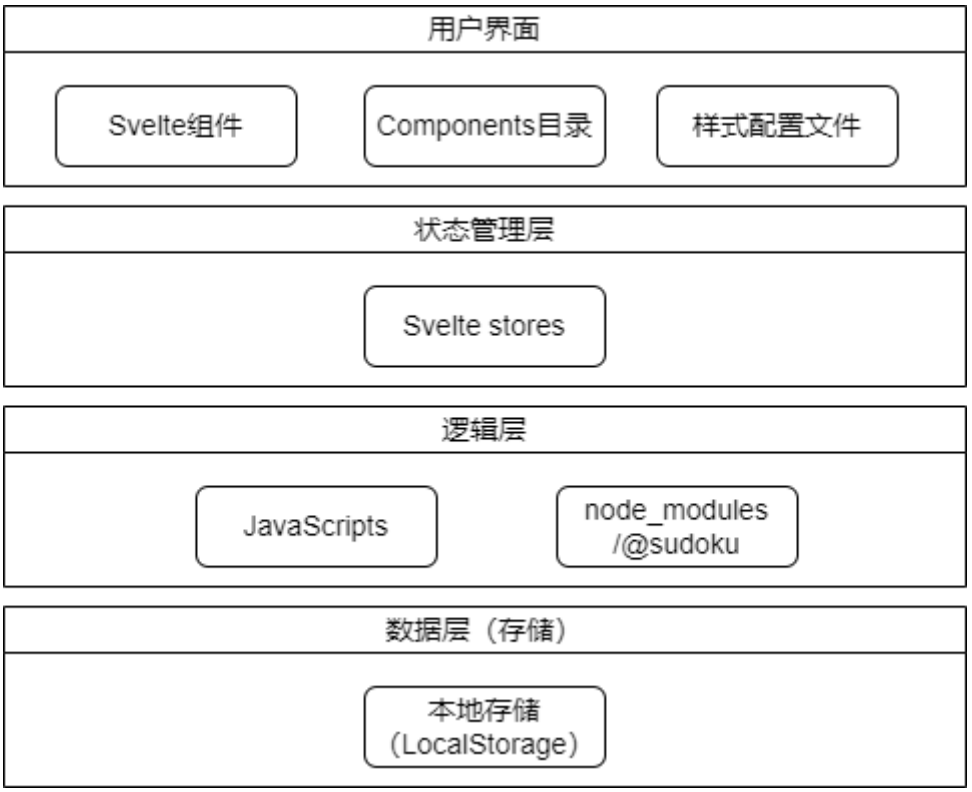
- **文件对应**：
  - **主要目录**：`/stores`
  - **示例状态管理模块**：
    - `candidates.js`：管理候选值状态。
    - `difficulty.js`：管理游戏难度。
    - `game.js`：管理游戏进度和网格状态。
    - `timer.js`：管理计时器状态。
    - `settings.js`：管理用户设置（如主题、提示开关）。

(3) 逻辑层（Logic Layer）

- **职责**：
  - 处理核心业务逻辑，如数独网格的生成、难度调整、提示逻辑。
  - 提供通用工具函数和辅助模块。
  - 与存储层和表现层交互，为表现层提供数据支持。
- **文件对应**：
  - **主要目录**：`/stores` 和 `/sencode`
  - **主要模块**：
    - `sudoku.js`：数独题目的生成逻辑。
    - `difficulty.js`：根据难度调整网格生成的模块。
    - `hints.js`：提示功能的实现。
    - `keyboard.js`：键盘输入逻辑（处理按键事件）。
    - `modal.js`：弹窗的管理逻辑。
  - **工具模块**：
    - `base62.js`：数据编码功能（如用于分享的URL生成）。
    - `clipboard.js`：实现数据复制功能。

(4) 数据层（Data Layer）

- **职责**：
  - 存储数独题目、游戏状态和用户设置。
  - 可能通过浏览器 LocalStorage 或 SessionStorage 实现持久化存储。
  - 数据可以直接通过 `stores` 模块管理，并存储在客户端。
- **文件对应**：
  - **主要目录**：`/stores` 和浏览器存储（如 LocalStorage）。
  - **实现方式**：
    - 数据的生成与存储直接在前端完成，无需后端支持。
    - 使用工具模块（如 `clipboard.js`）完成导入、导出或分享操作。



2.2 对象模型

（1）核心类分析

- **Cell (单元格类)**
  - **描述**：`Cell` 类表示数独棋盘上的一个单元格，包含数值和状态属性，支持值的设置和清空操作。
  - **属性**:
    - `value`：单元格的数值。
    - `cellX`：单元格的横坐标。
    - `cellY`：单元格的纵坐标。
    - `candidates`：候选值集合，类型为`Candidates`。
    - `disabled`：布尔值，指示单元格是否不可编辑。
    - `conflictingNumber`：当前单元格存在冲突时的数值。
    - `selected`：布尔值，是否选中。
    - `usernumber`：用户输入的数值。
  - **方法**:
    - `setValue(pos, num)`：设置单元格的值。
    - `clear()`：清空单元格内容。
- **Grid (棋盘类)**
  - **描述**：`Grid` 类由多个单元格(`Cell`)组成，表示整个数独棋盘。
  - **属性**:
    - `cells`：包含81个`Cell`对象的集合。
- **Board (棋盘视图类)**

- **描述**：`Board`类表示数独棋盘的界面显示部分，与`Grid`直接关联。
- **属性**：
  - `cells`：引用了`Grid`类中的单元格集合，用于显示棋盘。

- **SudokuGame (数独游戏类)**
  - **描述**：`SudokuGame`类为数独游戏的整体管理类，包含棋盘、控制面板和头部信息。
  - **属性**：
    - `header`：表示游戏头部，包含按钮和下拉菜单。
    - `board`：表示数独棋盘部分。
    - `controls`：表示控制区域，包括计时器和按钮功能。
- **Controls (控制面板类)**
  - **描述**：`Controls`类管理游戏的操作部分，包括动作栏、键盘输入和计时器。
  - **属性**：
    - `actionbar`：动作栏，包含主要的功能按钮（如暂停、提示）。
    - `keyboard`：虚拟键盘，用于用户输入。
    - `Timer`：计时器类，显示游戏时间。
- **ActionBar (动作栏类)**
  - **描述**：`ActionBar`类为用户提供主要操作功能按钮的实现。
  - **属性**：
    - `actions`：动作按钮集合（由`Actions`类提供）。
    - `timer`：计时器对象。
- **Header (头部类)**
  - **描述**：`Header`类为游戏头部，包含分享按钮和设置菜单等功能。
  - **属性**：
    - `buttons`：头部的按钮，支持分享和设置。
    - `dropdown`：下拉菜单，支持难度选择、自定义题目输入等功能。
  - **方法**：
    - `handleShareButton()`：处理分享按钮的点击操作。
    - `handleSettingsButton()`：处理设置按钮的点击操作。

(2) 功能模块分析

- **候选值模块（Candidates）**
  - **描述**：该模块用于管理单元格的候选值集合，支持动态更新和显示。
  - **属性**：
    - `candidates`：存储候选值的数组。
- **键盘模块（Keyboard）**
  - **描述**：支持用户通过键盘输入数值或移动光标。
  - **方法**：
    - `handleKeyButton(num)`：处理数字键输入。

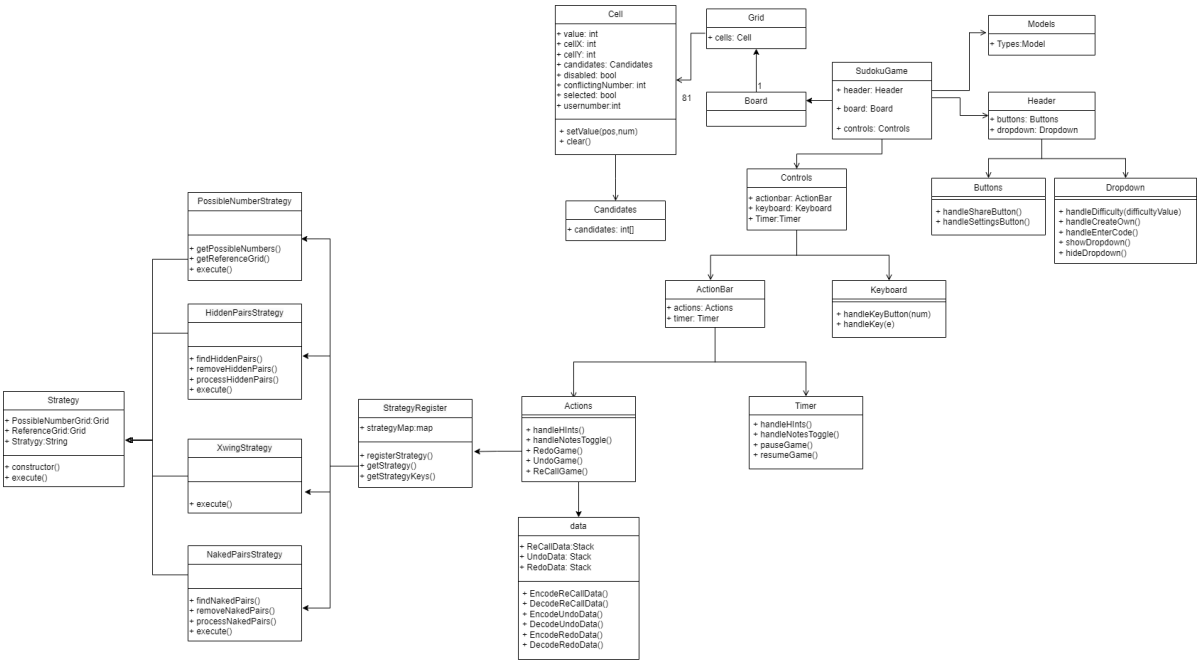
        - `handleKey(e)`：处理键盘事件。
- **计时器模块（Timer）**
    - **描述**：显示游戏时长，支持暂停与恢复操作。
    - **方法**：
        - `pauseGame()`：暂停计时器。
        - `resumeGame()`：恢复计时器。
- **Actions (动作处理类)**
    - **描述**：`Actions`类封装了用户操作的具体实现，如提示和暂停功能。
    - **方法**：
        - `handleHints()`：提供提示功能。
        - `handleNotesToggle()`：启用/关闭候选值功能。
        - `pauseGame()`：暂停游戏。
        - `resumeGame()`：恢复游戏。
- **按钮模块（Buttons）**
    - **描述**：实现分享和设置功能按钮的操作。
    - **方法**：
        - `handleShareButton()`：生成分享链接或二维码。
        - `handleSettingsButton()`：打开设置界面。
- **下拉菜单模块（Dropdown）**
    - **描述**：实现难度选择、自定义题目等功能的下拉菜单操作。
    - **方法**：
        - `handleDifficulty(difficultyValue)`：设置游戏难度。
        - `handleCreateOwn()`：创建自定义题目。
        - `handleEnterCode()`：解析用户输入的题目编码。
- **策略模块（Strategy）**
    - **描述**：策略算法基类
    - **属性**：
        - `PossibleNumberGrid`：候选值棋盘
        - `ReferenceGrid`：参考值棋盘
        - `Strategy`：策略名称
    - **方法**：
        - `constructor`：构造方法
        - `execute`：策略执行方法
- **PossibleNumber策略（PossibleNumberStrategy）**
    - **描述**：实现PossibleNumber算法，继承自策略算法基类，属性相同
    - **方法**：
        - `getPossibleNumbers`：获取候选值
        - `getReferenceGrid`：获取参考值

        - `execute`：策略执行方法
- **HiddenPairs策略（HiddenPairsStrategy）**
    - **描述**：实现HiddenPairs算法，继承自策略算法基类，属性相同
    - **方法**：
        - `findHiddenPairs`：寻找HiddenPairs
        - `removeHiddenPairs`：移除HiddenPairs
        - `processHiddenPairs`：执行HiddenPairs策略
        - `execute`：策略执行方法
- **NakedPairs策略（NakedPairsStrategy）**
    - **描述**：实现NakedPairs算法，继承自策略算法基类，属性相同
    - **方法**：
        - `findNakedPairs`：寻找NakedPairs
        - `removeNakedPairs`：移除NakedPairs
        - `processNakedPairs`：执行NakedPairs策略
        - `execute`：策略执行方法
- **Xwing策略（XwingStrategy）**
    - **描述**：实现XwingNumber算法，继承自策略算法基类，属性相同
    - **方法**：
        - `execute`：策略执行方法
- **策略注册器（StrategyRegister）**
    - **描述**：策略注册器，实现的策略需通过其注册
    - **属性**：
        - `strategyMap`：存储策略类
    - **方法**：
        - `registerStrategy`：注册策略
        - `getStrategy`：获取对应策略
        - `getStrategyKeys`：获取策略名称
- **数据存储模块（Data）**
    - **描述**：存储需要回退、回溯的数据。
    - **属性**：
        - ReCallData：存储需要回溯的数据，为栈结构。
        - UndoData：存储需要回退的数据，为栈结构。
        - RedoData：存储回退后需要前进的数据，为栈结构
    - **方法**：
        - `EncodeReCallData()`：存储回溯数据
        - `DecodeReCallData()`：读取回溯数据
        - `EncodeUndoData`：存储回退数据
        - `DecodeUndoData`：读取回溯数据

- `EncodeRedoData`：存储需要前进的数据
- `DecodeRedoData`：读取回溯后需要前进的数据

(3) 类间关系分析

- **聚合关系**：
  - `SudokuGame`类聚合了`Header`、`Board`和`Controls`，表示游戏的核心逻辑和界面部分。
  - `Controls`类聚合了`ActionBar`、`Keyboard`和`Timer`，实现控制面板功能。
- **依赖关系**：
  - `Grid`依赖于多个`Cell`对象。
  - `Header`依赖`Buttons`和`Dropdown`，处理具体的用户交互功能。
  - `Actions`类依赖`Data`和`Strategy`，实现具体的回溯和提示等功能。



2.3 设计说明（原则，模式）

(1) 单一职责原则（SRP）

- **体现**：
  - 各类模块职责更加单一。例如：
    - `data`类只负责回溯、撤销和前进方法的实现
    - `Strategy` 类独立负责数独的解题策略，解耦了算法实现和游戏逻辑。
- **好处**：
  - 代码更易于维护，每个模块的职责清晰，修改某一功能不会影响其他部分。

(2) 开放封闭原则（OCP）

- **体现**：
  - 新增解题策略可以通过扩展 `Strategy` 类的子类实现，而无需修改现有代码（如`PossibleNumber`、`NakedPairs`、`HiddenPairs` 等策略作为扩展）。
  - `SudokuGame` 类的功能可通过增加方法扩展而无需改动核心结构。

- **好处**：
  - 降低了代码修改风险，方便功能扩展。

(3) 依赖倒置原则（DIP）

- **体现**：
  - 高层模块（如 `SudokuGame`）依赖于抽象（如 `Strategy` 的接口 `strategyRegistry`），而非具体的实现（如 `PossibleNumber`）。
- **好处**：
  - 减少模块间的耦合性，提高代码的灵活性。

(4) 策略模式

- **体现**：
  - `Strategy` 类通过 `strategyRegistry` 维护不同的解题策略（如 `PossibleNumber`、`NakedPairs` 等）。
  - 游戏逻辑根据需要动态选择不同的解题策略。
- **作用**：
  - 提供了解题策略的灵活扩展方式，方便新增或替换不同的解题算法。

(5) 观察者模式

- **体现**：
  - 计时器模块（`Timer`）可能使用观察者模式通知其他模块时间变化，例如提示剩余时间。
  - 棋盘更新时可能通知界面重新渲染。
- **作用**：
  - 实现模块之间的解耦，使得状态变化能够及时反映到相关模块。

(6) 命令模式

- **体现**：
  - `Undo` 和 `Redo` 功能通过 `Command` 模式实现：
    - 用户的每一步操作（如输入数字、标记候选值）都可以封装成命令对象存储在操作栈中。
    - 撤销和重做操作通过执行或回退命令实现。
- **作用**：
  - 简化了复杂的操作回滚逻辑。

(7) 状态模式

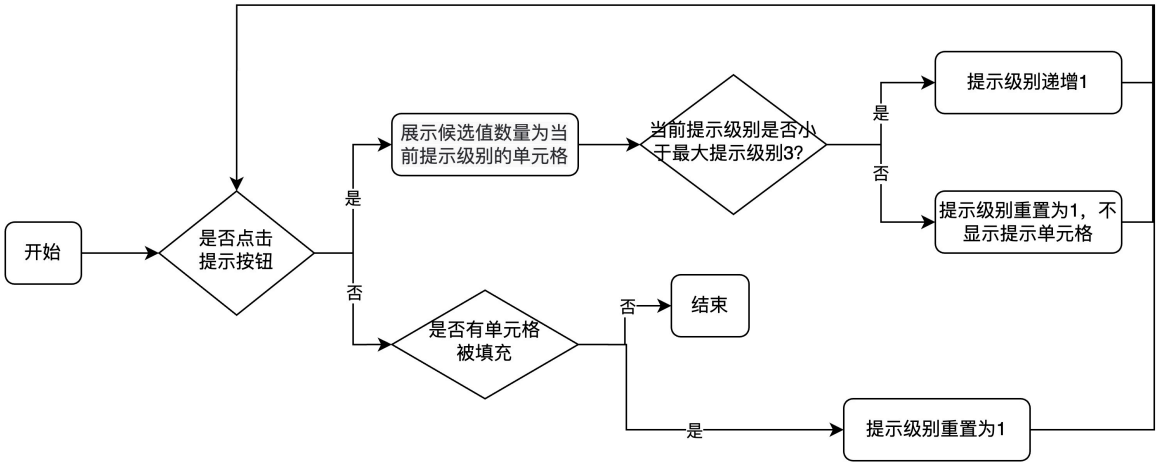
- **体现**：
  - `SudokuGame` 中使用状态模式管理不同游戏状态（如开始、暂停、结束）。
- **作用**：
  - 使得游戏状态的切换更加清晰，易于扩展新状态。

三、代码实现

3.1 下一步提示

在下一步提示功能中，设计的最大的提示级别为3，初始提示级别是1，提示级别表示**提示的单元格候选值数量。**

点击一次提示按钮会展示候选值数量为1的单元格，再次点击提示按钮会递增提示级别，然后补充提示候选值数量为2的单元格，直到最大单元格候选值数量3。此外，只要用户填充了单元格，提示级别会重置为1。




```
function reset(same) {
  clickNum.update($clickNum =%3E $clickNum = 0);
  $userGrid.forEach((row, rowIndex) => {
    row.forEach((cell, colIndex) => {
      candidates.clear({x: colIndex, y: rowIndex});
    });
  });
  if (!same) {
    localStorage.setItem('userGrid', JSON.stringify($userGrid));
    $userGrid.forEach((row, rowIndex) => {
      row.forEach((cell, colIndex) => {
        // hintGrid.clear({x: rowIndex, y: colIndex});
        strategyGrid.clear({x: rowIndex, y: colIndex});
        referenceGrid.clear({x: rowIndex, y: colIndex});
      });
      strategyContent.clear();
    });
  }
}

function solve() {
  const startTime = performance.now();
  let [possibleNumbers, referenceNumbers, strategy] = solveSudokuTest($userGrid);
// 调用策略获取提示
  $userGrid.forEach((row, rowIndex) => {
    row.forEach((cell, colIndex) => {
      candidates.clear({x: colIndex, y: rowIndex});
    });
  });
```

```
  let res = JSON.parse(JSON.stringify(possibleNumbers));

  let hasHint = false;
  res.forEach((row, rowIndex) => {
    row.forEach((element, colIndex) => {
      if (element.length > 0) {
        hasHint = true;
      }
    });
  });
  if (!hasHint) {
    modal.show('nosolution');
  }
  res.forEach((row, rowIndex) => {
    row.forEach((element, colIndex) => {
      if (element.length > level + $clickNum) {
        res[rowIndex][colIndex] = [];
        referenceNumbers[rowIndex][colIndex] = [];
        strategy[rowIndex][colIndex] = "";
      } else {
        element.forEach(value => {
          candidates.add({ x: colIndex, y: rowIndex }, value);
        });
      }
    });
  });
  strategy.forEach((row, rowIndex) => {
    row.forEach((element, colIndex) => {
      strategyGrid.set({x: rowIndex, y: colIndex}, element);
    });
  });
  referenceNumbers.forEach((row, rowIndex) => {
    row.forEach((element, colIndex) => {
      referenceGrid.set({x: rowIndex, y: colIndex}, element);
    });
  });
  clickNum.update($clickNum => $clickNum + 1);
}

// 下一步提示的主函数
function handleHint() {
  const storedUserGrid = localStorage.getItem('userGrid');
  let userGrid2 = JSON.parse(storedUserGrid);
  let same = JSON.stringify($userGrid) === JSON.stringify(userGrid2)
  if (level + $clickNum > maxLevel || !same) {
    reset(same);
  } else {
    solve();
  }
}
```

3.2 探索回溯

项目采用堆栈数据结构实现探索回溯功能，回溯栈存储的是用户当前局面，即usergrid，而前进后退栈存储的都是用户的单步操作。为防止回溯后会出现前进后退功能混乱的问题，执行回溯功能后将后退前进堆栈清空。

```
export const ReCallData = writable([])
export const BackupStack = writable([])
export const ForwardStack = writable([])

//生成可回溯序列
export function createReCalldata(userGrid){
    let newGrid = [];

    for (let y = 0; y < SUDOKU_SIZE; y++) {
        newGrid[y] = [];
        for (let x = 0; x < SUDOKU_SIZE; x++) {
            newGrid[y][x] = userGrid[y][x];
        }
    }

    ReCallData.update(stack => {
        stack.push(newGrid);
        return stack;
    })

    gameRecall.set(true);
}

export function decodeReCallData(){
    ReCallData.update(stack => {
        if (stack.length > 0) {
            const newGrid = stack.pop();
            for (let y = 0; y < SUDOKU_SIZE; y++) {
                for (let x = 0; x < SUDOKU_SIZE; x++) {
                    userGrid.set({ y, x }, newGrid[y][x]);
                    BackupData.clear();
                    ForwardData.clear();
                }
            }

            if (stack.length === 0) {
                gameRecall.set(false);
            }
        }

        return stack;
    });
}

function createBackupData(){

    return {
        subscribe: BackupStack.subscribe,
        add: (pos, value) =>{
```

```
            BackupStack.update($BackupStack => {
                $BackupStack.push({pos, value});
                return $BackupStack;
            });
            gameBackWard.set(true);
        },

        clear: () => {
            BackupStack.set([]);
            gameBackWard.set(false);
        }
    }
}

export const BackupData = createBackupData();

export function decodeBackupData(){

    BackupStack.update(stack => {
        if (stack.length > 0) {
            const data = stack.pop();
            ForwardData.add(data.pos,data.value);
            for (let y = 0; y < SUDOKU_SIZE; y++) {
                for (let x = 0; x < SUDOKU_SIZE; x++) {
                    userGrid.set(data.pos, 0);
                }
            }

            if (stack.length === 0) {
                gameBackWard.set(false);
            }
        }

        return stack;
    });
}

export function createForwardData(){

    return {
        subscribe: BackupStack.subscribe,
        add: (pos, value) =>{
            ForwardStack.update($ForwardStack => {
                $ForwardStack.push({pos, value});
                return $ForwardStack;
            });

            gameForward.set(true);
        },

        clear: () => {
            ForwardStack.set([]);
            gameForward.set(false);
        }
    }
}
```

```
export const ForwardData = createForwardData();

export function decodeForwardData(){

    ForwardStack.update(stack => {
        if (stack.length > 0) {
            const data = stack.pop();
            for (let y = 0; y < SUDOKU_SIZE; y++) {
                for (let x = 0; x < SUDOKU_SIZE; x++) {
                    userGrid.set(data.pos, data.value);
                }
            }

            if (stack.length === 0) {
                gameForward.set(false);
            }
        }
        return stack;
    });
}
```

3.3 资源集成——题目导入

sudokuwiki题目导入实现的核心在于判断是否为有效的Url，同时我们也保留了原先项目的解析功能，只需要判断是否符合sencode的格式或url的格式即可。

```
function isValidUrl(string) {
    const urlPattern = new RegExp('^(https?:\\/\\/)?' + // protocol
        '((([a-zA-Z\\d]([a-zA-Z\\d-]*[a-zA-Z\\d])*)\\.)+[a-zA-Z]{2,}|' + //
domain name
        '((\\d{1,3}\\.){3}\\d{1,3}))' + // OR ip (v4) address
        '(\\:\\d+)?(\\/[-a-zA-Z\\d%_.~+]*)*' + // port and path
        '(\\?[;&a-zA-Z\\d%_.~+=-]*)?' + // query string
        '(\\#[-a-zA-Z\\d_]*)?$', 'i'); // fragment locator

    const parsedUrl = new URL(string);
    return urlPattern.test(string) && parsedUrl.hostname === 'www.sudokuwiki.org'
&&                                 parsedUrl.searchParams.get('bd').length ===
GRID_LENGTH;
}

function extractDomainAndBd(url) {
    // 解析 URL
    const parsedUrl = new URL(url);

    // 提取域名
    const domain = parsedUrl.hostname;

    // 提取 bd 参数
    const bdParam = parsedUrl.searchParams.get('bd');

    return bdParam;
}
```

```
export function validateSencode(sencode) {  //验证sencode是否合法

    //给定一个url，去除前面的域名，只保留后面的参数，然后判断是否符合正则表达式
    return sencode && sencode.trim().length !== 0 && (SENCODE_REGEX.test(sencode)
|| isValidUrl(sencode));
}
```

3.4 资源集成——算法策略

3.4.1 策略实现

本项目实现了四个策略：分别是`PossibleNumber`，`NakedPairs`,`HiddenPairs`以及`Xwing`。我们定义了一个策略基类，其中定义了策略成员以及执行方法`execute`，策略子类只需要继承该基类实现其方法即可。其中仅对于`PossibleNumber`方法在执行时是不需要传参的，所以在后续执行时需要另外判定。

基类的实现方法如下：

```
class Strategy{ //基类

    constructor(){
        this.newPossibleNumberGrid = [];
        this.newReferenceGrid = Array.from({ length: SUDOKU_SIZE }, () =>
Array.from({ length: SUDOKU_SIZE }, () => []));
        this.strategy = "";
    }

    execute(possibleNumberGrid){
        if(possibleNumberGrid){
            this.newPossibleNumberGrid =
JSON.parse(JSON.stringify(possibleNumberGrid));
        }

        return [this.newPossibleNumberGrid,this.newReferenceGrid,this.strategy];
    }
}
```

策略的实现方法此处不一一给出，仅给出PossibleNumber和NakedPairs策略的实现为例：

```
class PossibleNumberStrategy extends Strategy{

    constructor() {
        super();
        this.strategy = "PossibleNumber";
    }

    getRow(sudoku, row) {
        return sudoku[row];
    }

    getCol(sudoku, col) {
        return sudoku.map(row => row[col]);
    }
```

```
    getBox(sudoku, row, col) {
        const box = [];
        const startRow = row - row % BOX_SIZE;
        const startCol = col - col % BOX_SIZE;
        for (let r = 0; r < BOX_SIZE; r++) {
            for (let c = 0; c < BOX_SIZE; c++) {
                box.push(sudoku[startRow + r][startCol + c]);
            }
        }
        return box;
    }

    getPossibleNumbers(sudoku, row, col) {
        let usedNumbers = new Set([
            ...this.getRow(sudoku, row),
            ...this.getCol(sudoku, col),
            ...this.getBox(sudoku, row, col)
        ]);

        let possibleNumbers = [];

        for (let num = 1; num <= 9; num++) {
            if (!usedNumbers.has(num)) {
                possibleNumbers.push(num);
            }
        }

        return possibleNumbers;
    }

    getReferenceGrid(sudoku, rowIndex, colIndex) {
        for (let num = 0; num < SUDOKU_SIZE; num++) {
            if (sudoku[rowIndex][num] != 0) {
                this.newReferenceGrid[rowIndex][colIndex].push([rowIndex, num]);
            }
        }

        for (let num = 0; num < SUDOKU_SIZE; num++) {
            if (sudoku[num][colIndex] != 0) {
                this.newReferenceGrid[rowIndex][colIndex].push([num, colIndex]);
            }
        }

        const startRow = rowIndex - rowIndex % BOX_SIZE;
        const startCol = colIndex - colIndex % BOX_SIZE;

        for (let r = 0; r < BOX_SIZE; r++) {
            for (let c = 0; c < BOX_SIZE; c++) {
                const value = sudoku[startRow + r][startCol + c];
                if (value !== 0) {
                    this.newReferenceGrid[rowIndex][colIndex].push([startRow + r,
startCol + c]);
                }
            }
        }
```

```
        return this.newReferenceGrid;
    }

    execute() {
        userGrid.subscribe($userGrid => {
            this.newPossibleNumberGrid = $userGrid.map((row, rowIndex) =>
                row.map((cell, colIndex) => {
                    if (cell === 0) {
                        return this.getPossibleNumbers($userGrid, rowIndex,
colIndex);
                    } else {
                        return [];
                    }
                })
            );

            this.newPossibleNumberGrid.forEach((row, rowIndex) => {
                row.forEach((cell, colIndex) => {
                    if (cell.length > 0) {
                        this.getReferenceGrid($userGrid, rowIndex, colIndex);
                    }
                });
            });
        });

        return [this.newPossibleNumberGrid, this.newReferenceGrid,
this.strategy];
    }
}

class NakedPairsStrategy extends Strategy{
    constructor(possibleNumberGrid = []) {
        super(possibleNumberGrid);
        this.strategy = "NakedPairs";
    }

    findNakedPairs(cells) {
        const pairs = [];
        const pairMap = new Map();

        cells.forEach((cell, index) => {
            if (cell.length === 2) {
                const key = cell.join(',');
                if (pairMap.has(key)) {
                    pairMap.get(key).push(index);
                } else {
                    pairMap.set(key, [index]);
                }
            }
        });

        pairMap.forEach((value, key) => {
            if (value.length === 2) {
                pairs.push(value);
            }
        });
```

```
        return pairs;
    }

    removeNakedPairs(cells, pairs) {
        const executed = [];

        pairs.forEach(([index1, index2]) => {
            const [num1, num2] = cells[index1];
            cells.forEach((cell, index) => {
                if (index !== index1 && index !== index2) {
                    const newCell = cell.filter(num => num !== num1 && num !==
num2);
                    cells[index] = newCell;
                    executed.push([index, index1, index2]);
                }
            });
        });

        return executed;
    }

    processNakedPairs(grid, reference) {

        for (let i = 0; i < SUDOKU_SIZE; i++) {
            // 行
            const row = grid[i];
            const rowPairs = this.findNakedPairs(row);
            const rowexecuted = this.removeNakedPairs(row, rowPairs);

            rowexecuted.forEach(([executedIndex, index1, index2]) => {
                reference[i][executedIndex].push([i, index1], [i, index2]);
            });

            row.forEach((cell, index) => {
                grid[i][index] = cell; // 更新原始 grid 的行
            });

            // 列
            const col = grid.map(row => row[i]);
            const colPairs = this.findNakedPairs(col);
            const colexecuted = this.removeNakedPairs(col, colPairs);

            colexecuted.forEach(([executedIndex, index1, index2]) => {
                reference[executedIndex][i].push([index1, i], [index2, i]);
            });

            col.forEach((cell, index) => {
                grid[index][i] = cell; // 更新原始 grid 的列
            });

            // 宫格
            const box = [];
            const startRow = Math.floor(i / BOX_SIZE) * BOX_SIZE;
            const startCol = (i % BOX_SIZE) * BOX_SIZE;
```

```
            for (let r = 0; r < BOX_SIZE; r++) {
                for (let c = 0; c < BOX_SIZE; c++) {
                    box.push(grid[startRow + r][startCol + c]);
                }
            }

            const boxPairs = this.findNakedPairs(box);
            const boxexecuted = this.removeNakedPairs(box, boxPairs);

            boxexecuted.forEach(([executedIndex, index1, index2]) => {
                const r = Math.floor(executedIndex / BOX_SIZE);
                const c = executedIndex % BOX_SIZE;
                const r1 = Math.floor(index1 / BOX_SIZE);
                const c1 = index1 % BOX_SIZE;
                const r2 = Math.floor(index2 / BOX_SIZE);
                const c2 = index2 % BOX_SIZE;

                reference[startRow + r][startCol + c].push([startRow + r1,
startCol + c1], [startRow + r2, startCol + c2]);
            });

            box.forEach((cell, index) => {
                const r = Math.floor(index / BOX_SIZE);
                const c = index % BOX_SIZE;
                grid[startRow + r][startCol + c] = cell; // 更新原始 grid 的宫格
            });
        }

        return [grid, reference];
    }

    execute(possibleNumberGrid) {

        if(possibleNumberGrid){
            this.newPossibleNumberGrid =
JSON.parse(JSON.stringify(possibleNumberGrid));
        }

        [this.newPossibleNumberGrid, this.newReferenceGrid] =
this.processNakedPairs(this.newPossibleNumberGrid, this.newReferenceGrid);

        return [this.newPossibleNumberGrid, this.newReferenceGrid,
this.strategy];
    }
}
```

为使执行函数能够调用这些策略，这里定义了一个策略注册器，实现的策略只需要注册到该策略注册器即可使用。

```
class StrategyRegistry {    //策略注册器

    constructor() {
        this.strategyMap = new Map();
    }
```

```
    registerStrategy(strategyName, strategyClass) {
        if (typeof strategyClass === 'function') {
            this.strategyMap.set(strategyName, strategyClass);
        } else {
            throw new Error('Strategy class must be a function');
        }
    }

    getStrategy(strategyName) {
        return this.strategyMap.get(strategyName);
    }

    getStrategyKeys() {
        return Array.from(this.strategyMap.keys());
    }
}

// Example usage:
export const strategyRegistry = new StrategyRegistry();

// Registering strategies
strategyRegistry.registerStrategy('PossibleNumber', PossibleNumberStrategy);
strategyRegistry.registerStrategy('NakedPairs', NakedPairsStrategy);
strategyRegistry.registerStrategy('HiddenPairs', HiddenPairsStrategy);
strategyRegistry.registerStrategy('XWing', XWingStrategy);
```

3.4.2 策略集成调用

策略集成调用主要分为两部分：

- 首先获取全局的最优策略组合，以最基础的PossibleNumber策略为初始，使用深度优先搜索算法搜索能达到全局最优的策略组合，全局最优定义为所有单元格中，候选值小于等于最大提示级别的最大单元格数量。
- 考虑到全局最优策略组合对每个单元格来说可能有冗余的策略，因此再针对每个单元格选择策略组合。首先按顺序执行全局最优策略组合，并记录每个策略输出的所有单元格的候选值。然后针对每个单元格，以倒序的方式遍历全局最优策略输出的单元格候选值，找到能够达到最少候选值的策略，记录为当前单元格的推理策略。

```
export function solveSudokuTest(sudoku) {
    let level = 0;
    settings.subscribe($settings =%3E {
        level = $settings.minhintlevelateachstep;
    });

    const strategyKeys = strategyRegistry.getStrategyKeys();  // 从
strategyRegistry中获取所有策略

    // PossibleNumber策略作为初始调用策略
    let initialStrategyClass = strategyRegistry.getStrategy(strategyKeys[0]);
    let initialStrategy = new initialStrategyClass();
    let [initialPossibleGrid, initialReferenceGrid, initialStrategyName] =
initialStrategy.execute();
```

```
    let maxCellsBelowLevel = initialPossibleGrid.flat().filter(cell =>
cell.length %3C= MAXLEVEL && cell.length > 0).length;
    let possibleNumbersGrid = initialPossibleGrid;
    let bestStrategyCombination = maxCellsBelowLevel > 0 ? [initialStrategyName]
: [];

    function dfs(currentGrid, currentStrategyIndex, currentCombination) {
        if (currentStrategyIndex >= strategyKeys.length) {
            return;
        }

        for (let i = currentStrategyIndex; i < strategyKeys.length; i++) {
            let strategyKey = strategyKeys[i];
            let strategyClass = strategyRegistry.getStrategy(strategyKey);
            let strategy = new strategyClass();
            let [newPossibleGrid, newReferenceGrid, strategyName] =
strategy.execute(currentGrid);

            let cellsBelowLevel = newPossibleGrid.flat().filter(cell =>
cell.length <= MAXLEVEL && cell.length > 0).length;

            if (cellsBelowLevel > maxCellsBelowLevel) {
                maxCellsBelowLevel = cellsBelowLevel;
                bestStrategyCombination = [...currentCombination, strategyName];
                possibleNumbersGrid = newPossibleGrid;
            }

            dfs(newPossibleGrid, i + 1, [...currentCombination, strategyName]);
        }
    }

    // 获取全局的最优策略组合
    dfs(initialPossibleGrid, 1, bestStrategyCombination);

    // 依次记录最优策略组合里每个策略输出的所有单元格的候选值
    let prevPossibleGrid;
    let referenceGridMap = new Map();
    let possibleGridMap = new Map();
    for (let i = 0; i < bestStrategyCombination.length; i++) {
        let key = bestStrategyCombination[i];
        let strategyClass = strategyRegistry.getStrategy(key);
        let strategy = new strategyClass();

        let newPossibleGrid, newReferenceGrid, strategyName;
        if (key == "PossibleNumber") {

            [newPossibleGrid, newReferenceGrid, strategyName] =
strategy.execute();
        } else {
            [newPossibleGrid, newReferenceGrid, strategyName] =
strategy.execute(prevPossibleGrid);

        prevPossibleGrid = newPossibleGrid;
        possibleGridMap.set(key, newPossibleGrid);
        referenceGridMap.set(key, newReferenceGrid);
    }
```

```
    // 针对每个单元格选择策略组合
    const referenceNumbersGrid = sudoku.map((row, rowIndex) =>
        row.map((cell, colIndex) => {
            if (cell === 0 && possibleNumbersGrid[rowIndex][colIndex].length > 0)
{
                let minKey = null;
                let minLength = Infinity;
                for (let [key, grid] of possibleGridMap) {
                    let length = grid[rowIndex][colIndex].length;
                    if (length < minLength) {
                        minLength = length;
                        minKey = key;
                    }
                }
                let collectedStrategies = [];
                for (let key of bestStrategyCombination) {
                    collectedStrategies.push(key);
                    if (key === minKey) break;
                }

                for (let i = collectedStrategies.length - 1; i >= 0; i--) {
                    let key = collectedStrategies[i];
                    let value = referenceGridMap.get(key)[rowIndex][colIndex];
                    if (value.length > 0) {
                        return value;
                    }
                }
                return [];
            } else {
                return [];
            }
        })
    );

        // 针对每个单元格选择策略组合
    const strategyGrid = sudoku.map((row, rowIndex) =>
        row.map((cell, colIndex) => {
            if (cell === 0 && possibleNumbersGrid[rowIndex][colIndex].length > 0)
{
                let minKey = null;
                let minLength = Infinity;
                for (let [key, grid] of possibleGridMap) {
                    let length = grid[rowIndex][colIndex].length;
                    if (length < minLength) {
                        minLength = length;
                        minKey = key;
                    }
                }
                let strategySequence = [];
                for (let key of bestStrategyCombination) {
                    strategySequence.push(key);
                    if (key === minKey) break;
                }
                return strategySequence;
            } else {
```

```
                return [];
            }
        })
    );

    return [possibleNumbersGrid, referenceNumbersGrid, strategyGrid];
}>)
```

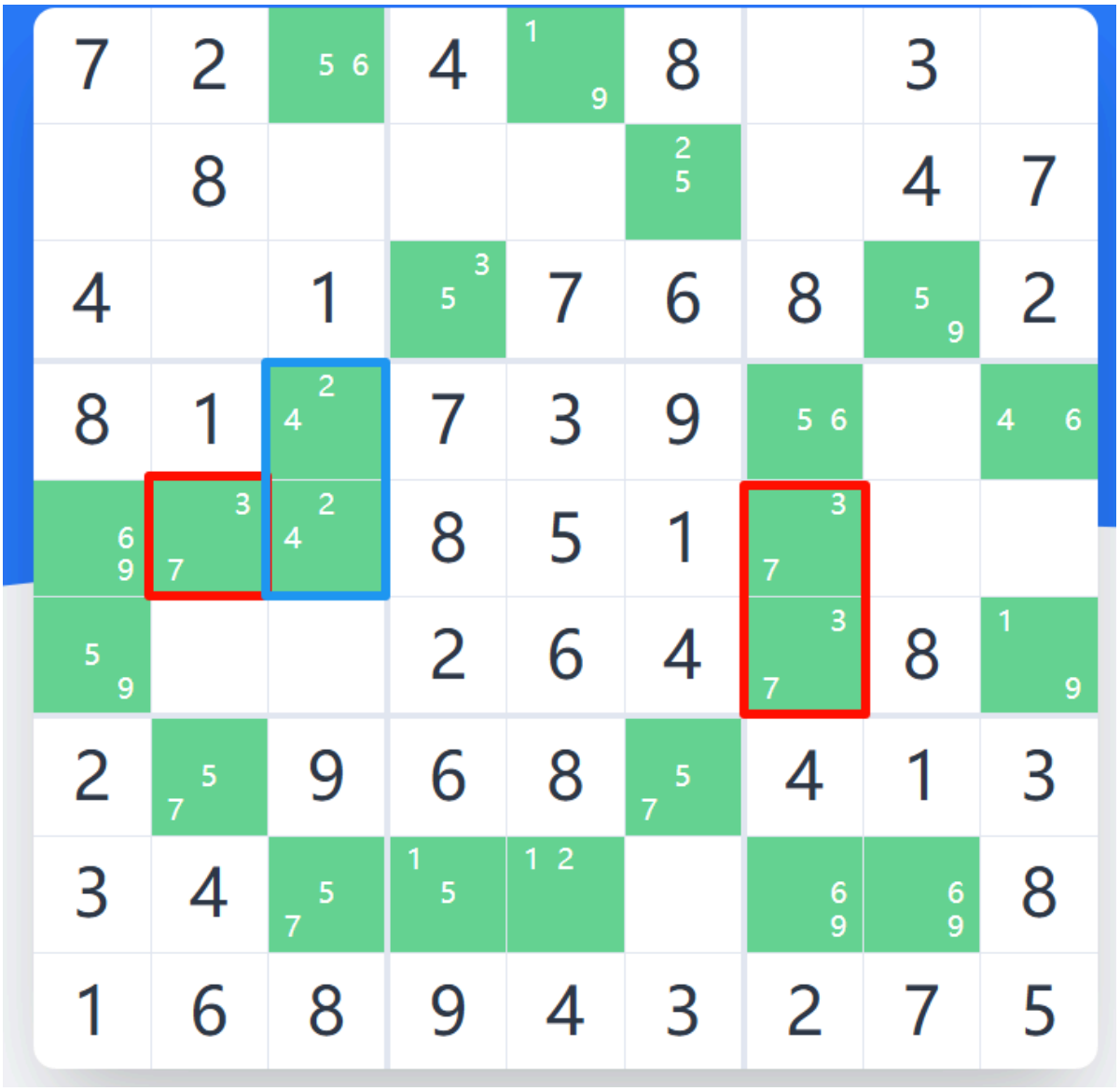
3.5 技术验证

3.5.1 策略集成正确性验证

由于各策略都单独执行互不影响，因此采用单元测试的方式对策略的正确性进行验证，可从sudokuwiki官网上找到对应策略的示例题目链接进行测试，且由于实现了参考值的查看功能，更有助于开发者确定策略是否正确实现。以下给出`HiddenPairs`策略的测试例子：

首先找到HiddenPairs的测试案例如图：

绿色单元格代表其应当保留得到的候选值，红框和蓝框内单元格代表其参考单元格，将题目导入到游戏中进行测试得到如图结果：



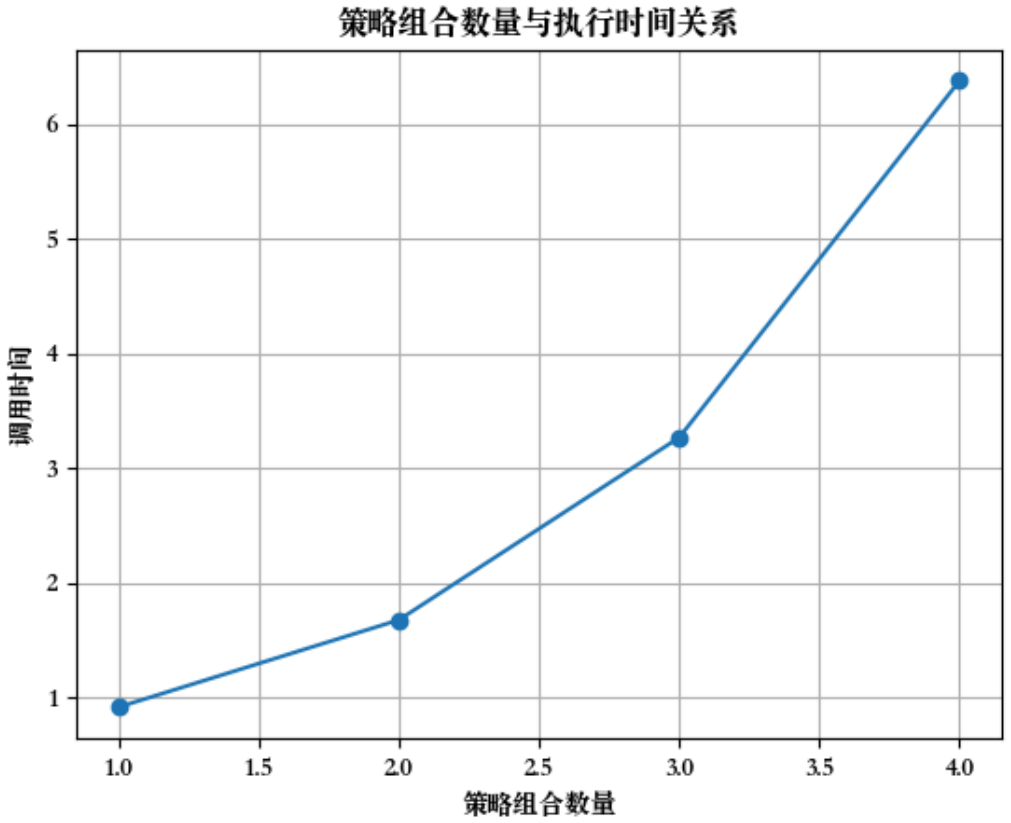
可以看到运行结果与测试案例一致，通过单元测试。

3.5.2 策略集成调用时间复杂度

使用的测试数独游戏：https://www.sudokuwiki.org/sudoku.htm?bd=030000000700000001009650800091207030040090020020406910005021400300000002000000050

下面的数值由每项都测试10次取平均值得到：

- PossibleNumber策略：0.92 ms
- PossibleNumber策略+NakedPairs策略：1.67 ms
- PossibleNumber策略+NakedPairs策略+HiddenPairs策略：3.26 ms
- PossibleNumber策略+NakedPairs策略+HiddenPairs策略+XWing策略：6.38 ms



折线图接近直线，说明策略集成调用的执行时间随着策略数量的增加而线性增加，复杂度接近线性。