《面向对象技术》

数独乐乐

数独乐乐-2：软件设计规格

学　　院　计算机学院

专　　业　计算机技术

年　　级　2024 级

成　　员　方正、李涛、陈骏伟

1 系统技术架构

1.1 总体框架

目前数独乐乐项目的实现可以分为 3 层：

- 表示层：负责与用户交互，展示数独游戏信息。
- 业务层：串联表示层和逻辑层，接收表示层的输入，使用逻辑层提供的数据结构进行处理，反馈操作结果。
- 逻辑层：负责支持业务层的功能实现，包括各种数据结构。

整个项目的运行是由用户在表示层的操作驱动的。用户的操作首先传递到业务层，业务层根据操作类型和逻辑，调用逻辑层的相应功能和数据结构进行处理，处理结果再通过业务层反馈到表示层，更新用户界面，形成一个完整的交互循环。这种分层结构使得各层职责明确，便于开发、维护和扩展。表示层专注于用户体验和界面展示，业务层负责协调和处理业务流程，逻辑层则专注于底层数据和逻辑的处理，各层之间通过接口进行交互，降低了层与层之间的耦合度，提高了系统的灵活性和可维护性。

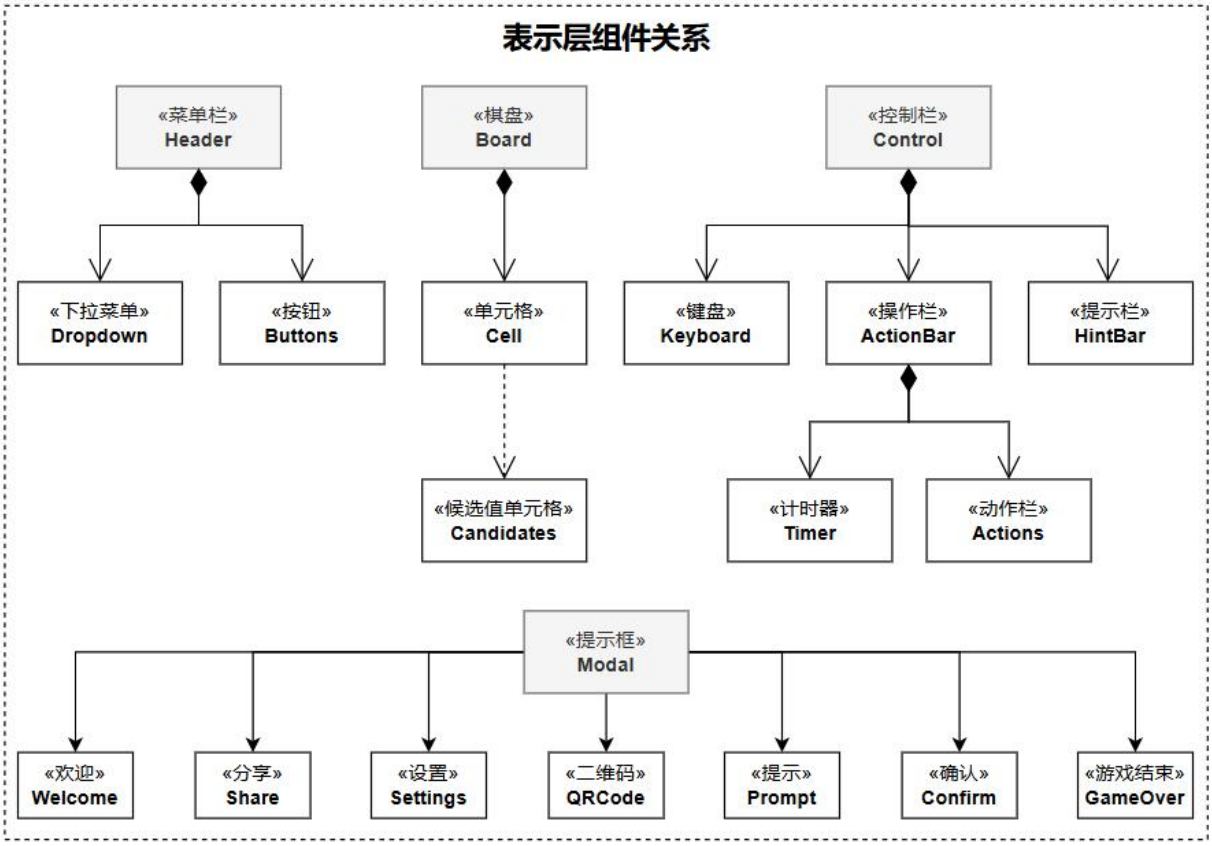
1.2 表示层架构

表示层负责处理与用户的交互，需要为用户提供界面显示和使用各个功能的组件。表示层的代码位于 src/components 目录下，该目录包含了 Header、Board、Controls 和 Modal 四个主要组件及其相关的工具类 Utils。

- Header（顶部菜单栏）：位于页面顶端的菜单栏。
  - Dropdown：下拉菜单，用于设置难度，重新开始游戏。
  - Buttons：分享按钮和设置按钮。
- Board（棋盘）：数独游戏的棋盘，包括所有单元格。
  - Cell：显示单元格，包括单一数字（包括空白）和候选值两种情况。
    - Candidates：为单元格提供显示候选值的功能。
- Controls（底部控制栏）：位于棋盘下方，包含提示、计时器和数字输入等组件。
  - hintBar：显示提示推理链，说明产生提示使用的策略。
  - ActionBar：放置各个功能按钮。
    - Timer：计时器，显示游戏进行的时间，控制暂停和恢复游戏。
    - Actions：包含回溯撤销、重做、提示等操作按钮。
  - Keyboard：键盘输入，提供输入和删除数字功能的组件。
- Modal（提示框）：采用策略模式，根据具体场景选择不同的提示框弹出。
  - Welcome：游戏开始前出现的提示框，允许用户选择难度或通过输入 sencode 开始游戏。
  - Share：用于分享游戏，能够生成 sencode 并提供复制到剪贴板的按钮以及多个社交平台的分享选项。

- Settings：用于调整数独游戏的各项设置。
- QRCode：通过二维码分享数独游戏。
- Prompt：用于显示提示信息。
- GameOver：在游戏结束时显示相关信息，如用时、难度及提示次数等。
- Confirm：用于进行信息确认。

表示层的组件关系如下图所示：



为了实现新功能，主要修改了原项目中的候选值单元格 Candidates、动作栏 Actions 和提示栏 HintBar。

- 候选值单元格 Candidates 用于显示候选值。
- 动作栏 Actions 中增加了撤销、重做、回溯的功能按钮。
- 提示栏 HintBar 通过简单的文字说明提示采用的策略。

1.3 业务层架构

业务层在整个系统中处于关键位置，发挥着承上启下的重要作用。向上，它需要与表示层紧密交互，妥善处理用户的各种操作和交互需求，例如接收用户的点击、输入等指令，并及时给予正确的反馈；向下，它要对逻辑层中的数据结构进行合理管理和运用，以保证数据的正确处理。

为了实现新的功能，提供探索回溯、下一步提示等功能，我们对原有项目进行了优

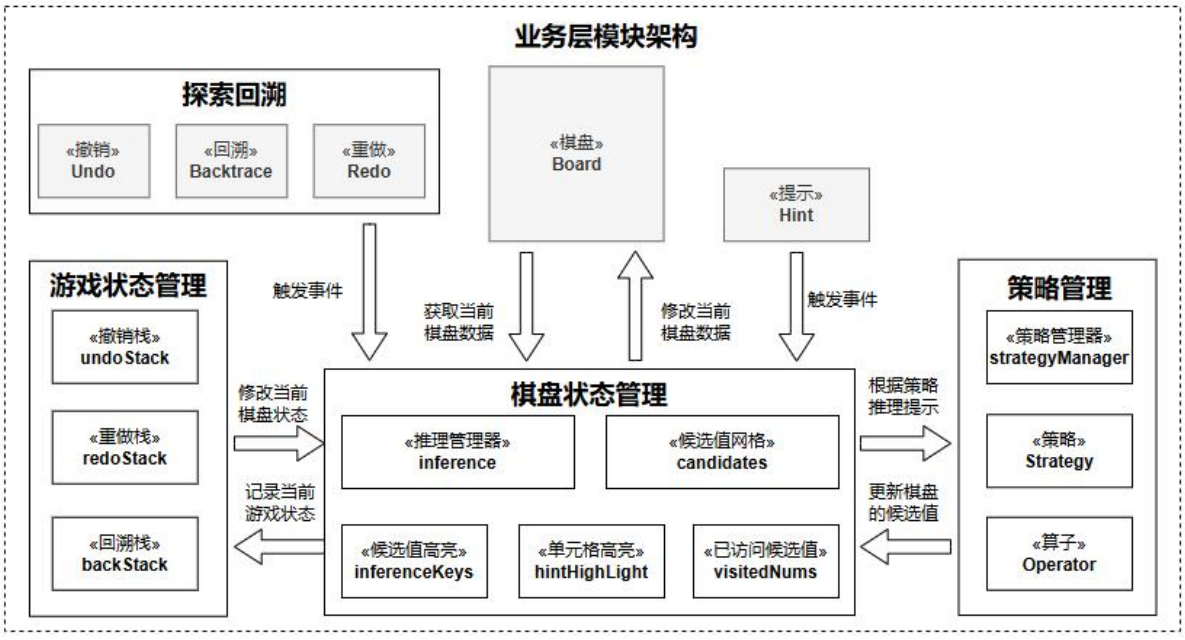
化和补充，新增了棋盘状态管理模块、游戏状态管理模块和策略管理模块。

棋盘状态管理模块主要负责对棋盘数据以及显示进行管理维护。对棋盘的操作都需要通过棋盘状态管理模块进行，当事件发生时，棋盘状态管理模块会通知其他模块进行处理，并根据处理的结果修改棋盘数据和显示方式。推理管理器 inference 对整体工作进行管理，候选值网格 candidates 记录候选值，推理依据值高亮 inferenceKeys、推理依据单元格高亮 hintHighLight 和已访问候选值 visitedNums 共同用于修改棋盘显示方式。

游戏状态管理模块的职责在于精准跟踪游戏过程中棋盘状态的每一次变更，以实现撤销、重做和回溯功能。为此，我们引入了撤销栈 undoStack、重做栈 redoStack 以及回溯栈 backStack 对游戏过程中棋盘状态的改变进行管理。这三个栈会在棋盘状态发生变更时记录信息，在收到玩家点击相关按钮时使用栈中的状态修改当前棋盘状态。

策略管理模块专注于对数独游戏的求解方式进行管理，包括了策略管理器 strategyManager、策略 Strategy 以及算子 Operator。策略管理器负责统筹求解过程中各种求解策略的运行，得到最终结果；推理网格用于将策略运行结果进行可视化；策略是求解数独的方法，策略的实现依赖于各种算子。在玩家点击提示按钮或执行其他操作导致提示需要重新获取生成时，策略管理模块会根据当前棋盘状态，生成相应的提示信息，并将这些提示信息提供给表示层，对玩家的下一步行动提供引导。

业务层模块架构如下图所示：



1.4 逻辑层架构

为了支持业务层的功能实现，逻辑层中需要提供相应的数据结构。

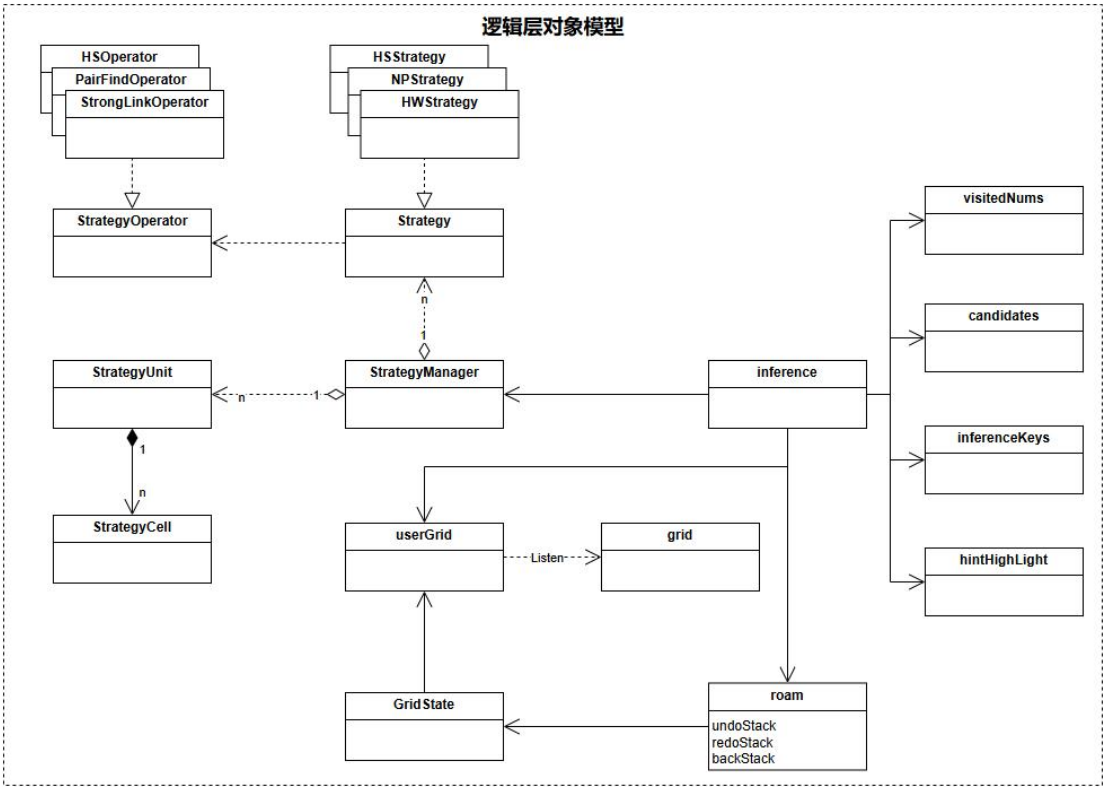
对棋盘状态管理模块，inference 类用于管理棋盘的推理数据和显示方式。在数据管

理方面，inference 会将 userGrid 和 grid 中的数据传递给 StrategyManager 和 roam 对象进行操作，得到操作结果后修改棋盘推理数据和显示方式。在棋盘显示方面，visitedNums 用于存储在游戏过程中已经访问过的数字，candidates 用于保存每个单元格的候选数字，inferenceKeys 用于对单元格中需要高亮的候选值进行标记，表示推理的数据来源，hintHighLight 用于表示需要高亮的单元格。这些对象共同为 inference 提供了丰富的数据支持，使其能够承载复杂的推理和交互功能。

对游戏状态管理模块，通过 roam 类保存 undoStack、redoStack 和 backStack，实现探索回溯操作。roam 类并不直接对 userGrid 进行操作，而是对封装的 GridState 进行操作。

对策略管理模块，提供 StrategyManager 类对策略进行管理。Strategy 类是策略类的基类，通过继承这个类（如 HSStrategy、NPStrategy 和 HWStrategy）实现不同的数独求解策略，StrategyManager 与 Strategy 存在一对多的关系，StrategyManager 可以包括多个策略，在求解数独时采取多个策略进行。StrategyOperator 是策略算子类，将数独策略的共性操作提取出来，通过继承它实现不同的算子（如 HSStrategy、NPStrategy 和 HWStrategy），Strategy 和 StrategyOperator 存在一对多的关系，一个策略可以采用多个算子完成。StrategyUnit 是策略求解结果的单元，StrategyCell 是策略求解结果单元中每个单元格的内容用于记录策略运行结果，他们之间存在一对多的关系，StrategyManager 与 StrategyUnit 存在一对多的关系。

逻辑层的对象模型如下图所示：



2 资源集成设计与实现

（1）题目导入

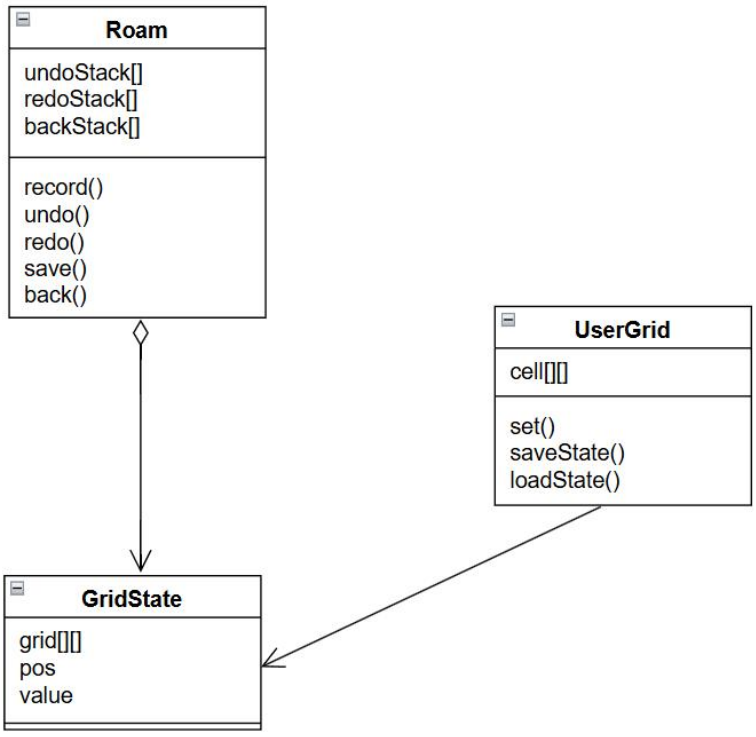
题目导入部分相对较为简单，实现时并未使用特殊的设计或模式。主要涉及 URL 的解析和验证，项目中直接使用 URL API 工具将 url 字符串转换为 url 对象。如果能成功创建 url 对象，并能验证对应部分的字段值能有效匹配符合要求，则为有效的 url 输入。随后只需将 bd 参数获取，将其按顺序输入到 grid 数组中即可。

（2）算法策略集成

通过模板方法模式，定义策略基类，随后根据不同的算法策略实现基类以及对应算子，再将具体算法策略注册到策略管理器中即可。这部分将在第 4 节中详细叙述。

3 探索回溯设计与实现

探索回溯的设计实现采用**备忘录模式 + 三栈结构（undoStack、redoStack 和 backStack）**实现。其设计模式图如下：



（1）设计思想

使用备忘录模式，将 userGrid 修改后的二维棋盘和修改坐标 pos 以及修改值 value 封装成备忘录类 GridState。并将其存储到 Roam 中进行管理。当需要撤回或重做时，Roam 将对应的备忘录对象返回，应用到 userGrid 上即可。

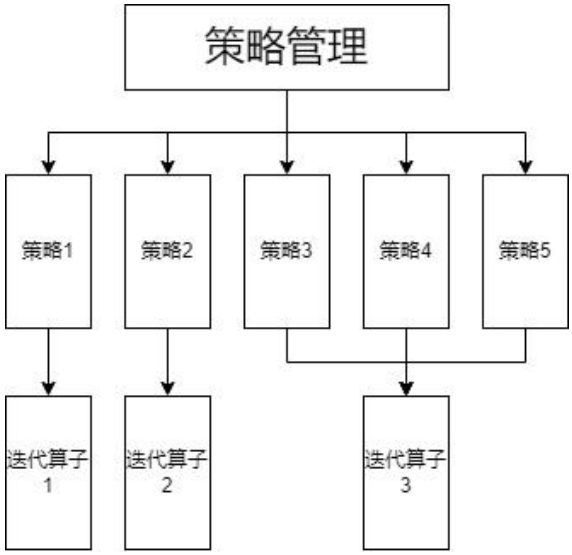
（2）实现流程

- userGrid 每次操作修改后，调用 userGrid 的 saveState 函数，将状态结果与修改信息（pos，value）封装到 GridState 中。再通过 Roam 的 record 函数将 GridState 对象添加到 undoStack 中，并清空 redoStack。若输入单元格的推理候选值不唯一，还需要调用 Roam 的 save 函数，此时需要进一步判断，如果当前推理候选值只剩用户输入值未被访问探索，则清空该单元格的访问记录，该单元格将不再是分支点。否则将当前 undoStack 的长度压入到 backStack 中。以便后续回溯到该单元格。
- **Undo 撤销**：调用 Roam 的 undo 函数：将 undoStack 的栈顶元素弹出，并压入到 redoStack 中。此时需要判断 undoStack 的长度是否小于 backStack 栈顶元素值，如果小于，则需要将 backStack 的栈顶元素弹出舍弃。最后返回 undoStack 的新栈顶元素。将其中的 grid 数组取出应用到 userGrid 上即可。
- **Redo 重做**：调用 Roam 的 redo 函数：将 redoStack 的栈顶元素弹出，并压入到 undoStack 中，最后返回 undoStack 的新栈顶元素。将其中的 grid 数组取出应用到 userGrid 上即可。
- **back 回溯**：调用 Roam 的 back 函数：取出 backStack 的栈顶元素，其值减一即为上个分支选择后的状态在 undoStack 中索引 index，从 undoStack 中获取该状态，取出修改信息 pos 和 value，将 value 添加到 pos 单元格的访问记录中。然后将 undoStack 中 index 及其之后的状态全都弹出舍弃，redoStack 清空，最后返回 undoStack 的新栈顶元素，将其中的 grid 数组取出应用到 userGrid 上即可。

4 策略模块设计与实现

该模块中集成各种数独解题策略，根据输入的棋盘，按指定顺序和策略计算得到下一步的提示结果。

4.1 模块框架



4.2 策略管理类的设计与实现

StrategyManager：策略管理器的核心类，负责管理数独解题策略的执行和候选值的更新。

（1）属性：

- candidates：存储所有单元格的候选值（9x9 的二维数组，每个元素是一个候选值数组）。
- cellTrack：存储每个单元格的候选值消除跟踪信息（9x9 的二维数组，每个元素是 StrategyUnit 的 ID 数组）。
- strategyUnits：存储所有策略找到的匹配单元（StrategyUnit 数组）。
- strategyRank：定义策略的执行顺序（如 ['HS', 'NP', 'XW']）。
- strategy：策略注册表，包含所有已注册的策略实例（如 HSStrategy、NPStrategy、XWStrategy）。

（2）方法：

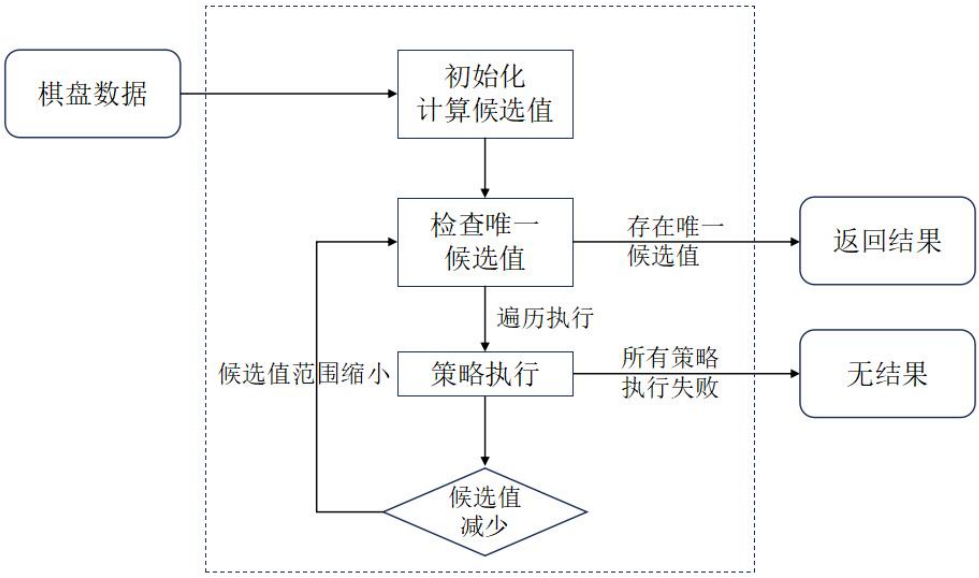
- run (grid)：运行策略管理器，迭代执行策略。
- checkOnlyOne ()：检查是否有唯一候选值的单元格。
- init (grid)：初始化数据（重置 hint、candidates、cellTrack 等）。
- getCandates (grid)：获取所有单元格的候选值。
- getPosCandidates (y, x, grid)：获取指定单元格的候选值。
- getInferenceGrid ()：返回推理候选值网格。
- getCellTrack ()：返回单元格跟踪数据。
- getStrategyUnits ()：返回策略单元列表。

（3）示例数据流

```
StrategyManager
  |
  |-- candidates: [
            [ [1,2], [3], [], ... ],   // 9x9  的候选值数组
            ...
      ]
  |-- cellTrack: [
            [ [0], [], [], ... ],   // 9x9  的  StrategyUnit ID  数组
            ...
      ]
  |-- strategyUnits: [
            StrategyUnit ("Hidden Single")
                  |-- StrategyCell (y=0, x=2, values=[5])
                  |-- StrategyCell (y=3, x=4, values=[7, 9])
            ...
      ]
  |-- strategyRank: ['HS', 'NP', 'XW']
  |-- strategy: {
            'HS': HSStrategy,
            'NP': NPStrategy,
            'XW': XWStrategy
      }
```

4.3 策略执行逻辑

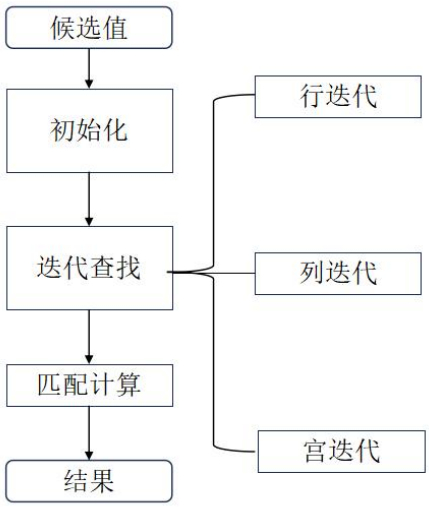
在 strategyRank 中指定策略执行的顺序以及需要执行哪些策略。循环遍历各个策略，若某一个策略能够减少棋盘上的候选值，就会跳转到开始重新遍历执行，直到找到一个答案，或所有策略都无影响为止。如此设计可以保证，各个策略之间是独立的，某个策略的执行不会影响到其他策略，只关注其输入输出。



4.4 策略的设计与实现

在具体策略上使用**模板方法模式**，将策略执行抽象成两个阶段，如下图所示，迭代查找和匹配计算。其中迭代查找在一些策略中有大量重复的操作，例如涉及到强连接的策略，都需要使用迭代来寻找行，列和宫的强连接。

所以，**设计迭代算子**可以**复用**这部分代码，将其封装进算子，其他使用到这种迭代的策略可以复用。



4.4.1 策略基类设计：Strategy 基类

所有具体策略的基类，定义了策略的**模板行为**和结构。

（1）属性：

- strategyName：策略的名称。
- operator：操作器，用于执行具体的策略逻辑（如遍历、匹配等）。
- config：迭代配置，是一个布尔数组，表示是否启用行、列、宫的迭代（如 [true, true, false] 表示启用行、列迭代，禁用宫迭代）。

（2）方法：

- run (candidates, cellTrack, strategyUnits)：运行策略，包括初始化、迭代查找和处理结果。
- init ()：初始化策略（可由子类重写）。
- iterativeSearch (candidates)：根据 config 配置迭代查找候选值（行、列、宫）。
- load (midData)：存储迭代查找的中间结果（可由子类重写）。
- handle (candidates, cellTrack, strategyUnits)：根据迭代结果处理候选值，并返回是否成功更

新候选值（可由子类重写）。

- name ()：返回策略名称。

（3）示例数据流

```
Strategy ("X-Wing")
  |
  |-- strategyName: "X-Wing"
  |-- operator: HSOperator
  |-- config: [true, true, false]   // 启用行、列迭代，禁用宫迭代
  |-- run(candidates, cellTrack, strategyUnits)
        |
        |-- init()   // 初始化
        |-- iterativeSearch(candidates)
              |
              |--  行迭代
                    |-- operator.begin()
                    |-- operator.execute(y, x, candidates)   // 遍历每一行
                    |-- operator.end()   // 返回中间结果
              |--  列迭代
                    |-- operator.begin()
                    |-- operator.execute(y, x, candidates)   // 遍历每一列
                    |-- operator.end()   // 返回中间结果
        |-- load(midData)   // 存储中间结果
        |-- handle(candidates, cellTrack, strategyUnits)   // 处理结果并更新候选值
  |-- name()   // 返回 "Hidden Single"
```

设计中 run(candidates, cellTrack, strategyUnits)和 iterativeSearch(candidates)为模板方法，各个策略执行以此为模板，只需重写实现一些算法逻辑。

4.4.2 迭代算子基类设计：StrategyOperator 类

StrategyOperator 是策略算子的基类，定义了算子的通用行为和结构。

（1）属性：

- enabled：操作器是否启用的标志。true 表示操作器已启用，可以执行逻辑；false 表示操作器未启用。
- opName：操作器的名称。

（2）方法：

- begin ()：启用操作器，将 enabled 设置为 true。此方法需要子类重写，并先执行父类方法。
- execute (y, x, candidates)：执行操作逻辑。此方法需要子类重写，并先执行父类方法，否则会输出警告信息。

- end ()：禁用操作器，并返回中间数据，将 enabled 设置为 false。此方法需要子类重写，并先执行父类方法。
- name ()：返回操作器的名称。

（3）示例数据流

```
StrategyOperator ("Hidden Single Operator")
  |
  |-- enabled: false   // 初始状态为未启用
  |-- opName: "Hidden Single Operator"
  |-- begin()   // 启用操作器
        |
        |-- enabled: true
  |-- execute(y, x, candidates)   // 执行操作逻辑
        |
        |-- 检查 enabled 是否为 true
        |-- 执行具体的策略逻辑（如查找唯一候选值）
  |-- end()   // 禁用操作器
        |
        |-- enabled: false
  |-- name()   // 返回 "Hidden Single Operator"
```

4.5 数据结构定义

（1）策略单元：StrategyUnit 类

策略单元的抽象，用于存储与某个策略相关的单元格信息。

属性：

- strategyName：策略单元的名称。
- cells：存储多个 StrategyCell 的数组，每个 StrategyCell 表示一个单元格及其候选值。

（2）StrategyCell 类

单元格的抽象，包含单元格的坐标和候选值列表。

属性：

- y：单元格的行坐标。
- x：单元格的列坐标。
- values：单元格的候选值列表。

（3）单元格策略追踪

cellTrack 存储每个单元格的候选值消除跟踪信息（9x9 的二维数组，每个元素是 StrategyUnit 的 ID 数组）。

4.6 模块的扩展性与维护

（1）扩展性

当需要添加一个新策略时，需要继承 Strategy 基类，并实现所需方法。根据具体算法设计，若已有相应功能的迭代算子可以进行复用，若没有则需要进行设计实现满足新功能的算子。

（2）维护性

Strategy 基类和 StrategyOperator 基类保证了策略执行的基本结构，其中的方法规定了各个部分的实现步骤，不同功能区分明显，提升代码可读性，对于后期代码维护友好。

4.7 测试与验证

由于设计时各个策略相对独立开，并且输入输出规定明确。所以，引入新测试时，可以禁用其他所有策略，单独测试出错的用例，并将测试用例记录下来。

当整体运行出现错误时，由于策略是按顺序循环执行，可以使用二分法快速定位出错的策略。

4.8 未来改进方向

策略优化：对策略算法进行优化，减少计算开销，优化系统响应时间。
策略阶段拆分：将策略的第二阶段进行进一步拆分，将一部分共性代码再次复用。