Reversi 黑白棋的前端界面与 AI 实现

0. 目录

1. 黑白棋

黑白棋, 又叫翻转棋 (Reversi), 奥赛罗棋 (Othello), 苹果棋或正反棋 (Anti reversi). 黑白棋在西方和日本很流行. 游戏通过相互翻转对方的棋子, 最后以棋盘上谁的棋子多来判断胜负. 它的游戏规则简单, 因此上手很容易, 但是它的变化又非常复杂.

在线网址: https://reversi.orangex4.cool/

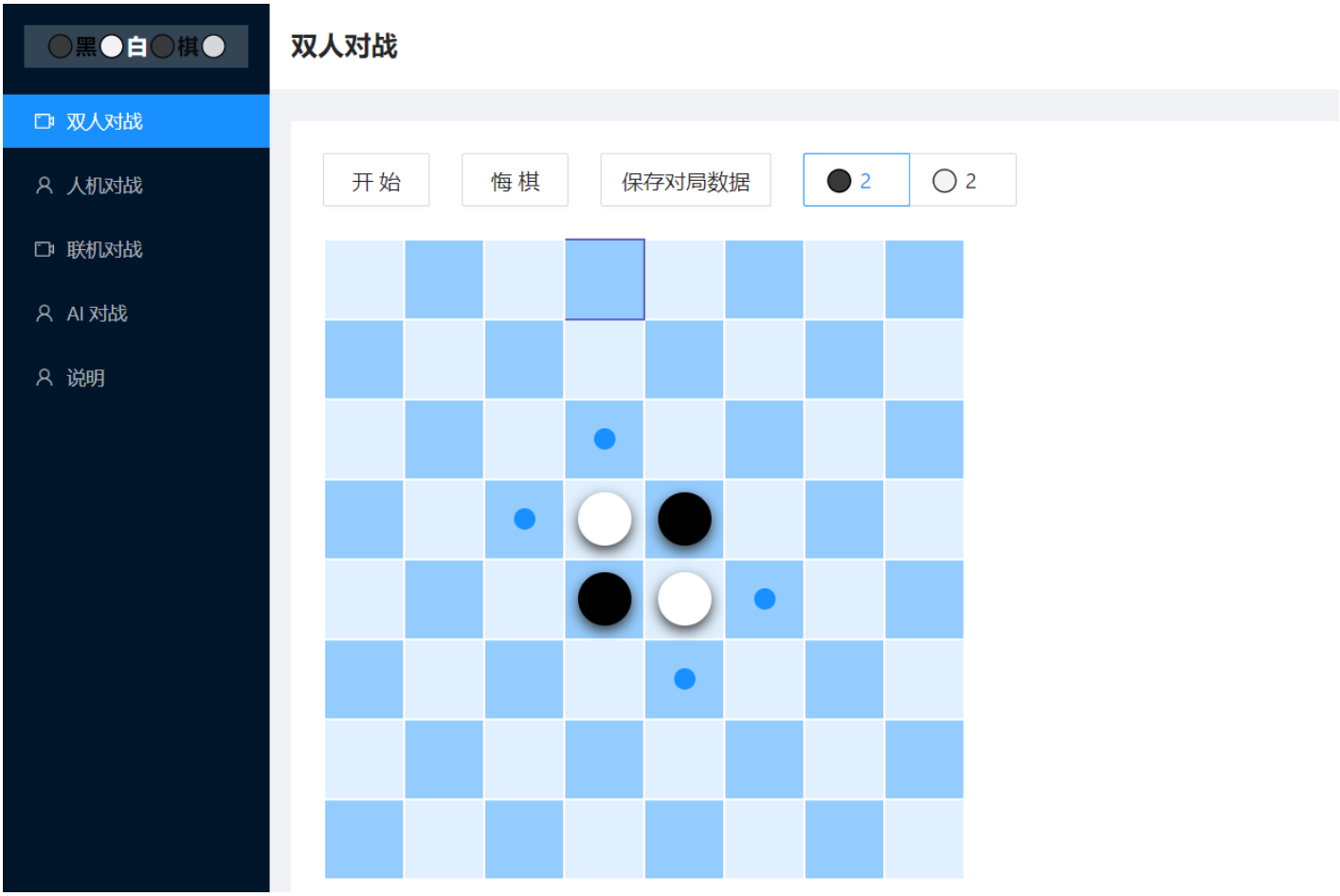
2. 游戏逻辑和前端界面

2.1 双人对战

拥有基本的黑白棋游戏判断逻辑, 和简洁的界面显示.

用于在一台设备上自娱自乐 (并不.

并且可以保存对局数据, 便于后续分析.



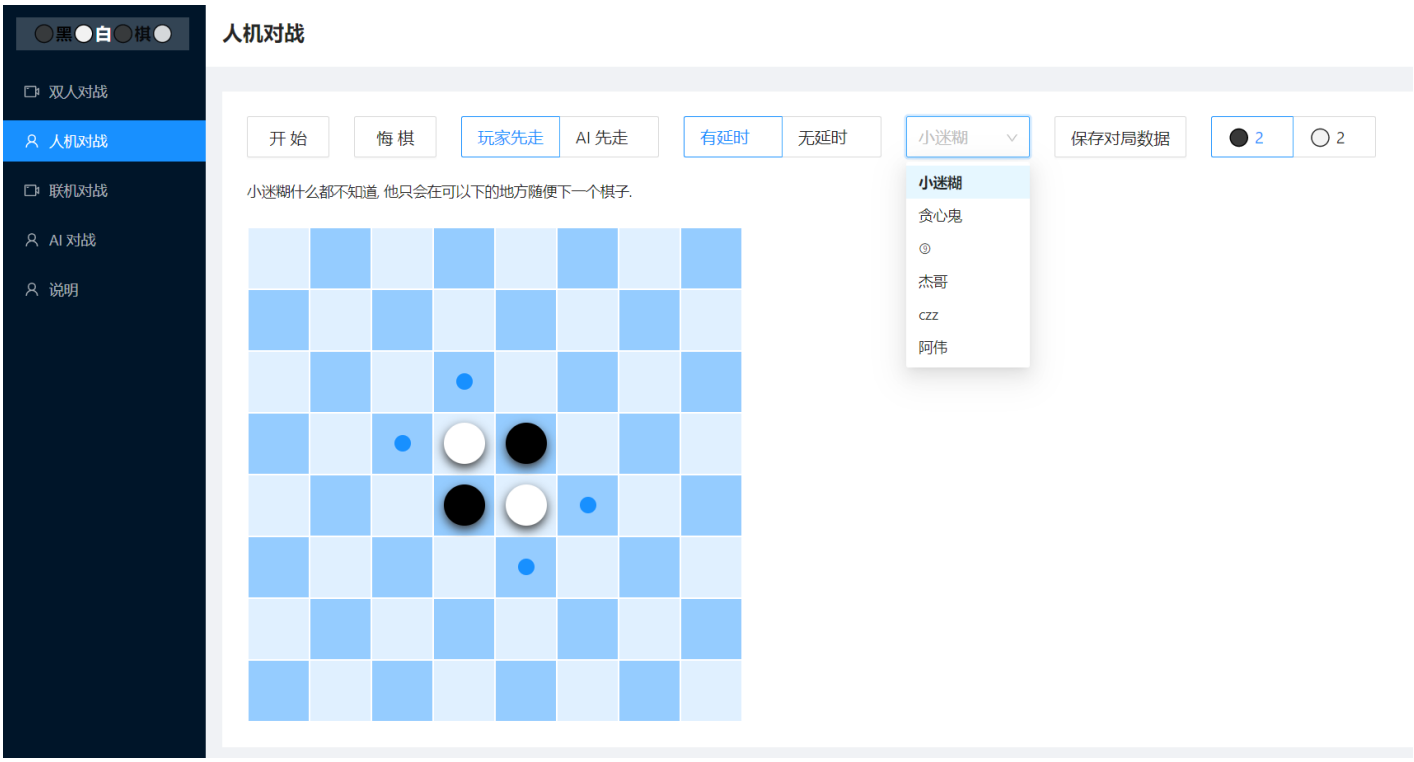
2.2 人机对战

通过前后端分离的设计, 前端用前端全家桶实现, 后端使用 Python 语言, 便于整合黑白棋 AI 算法.

目前已经有数位小伙伴把 AI 放到这里了!

比如杰哥, czz 和阿伟.

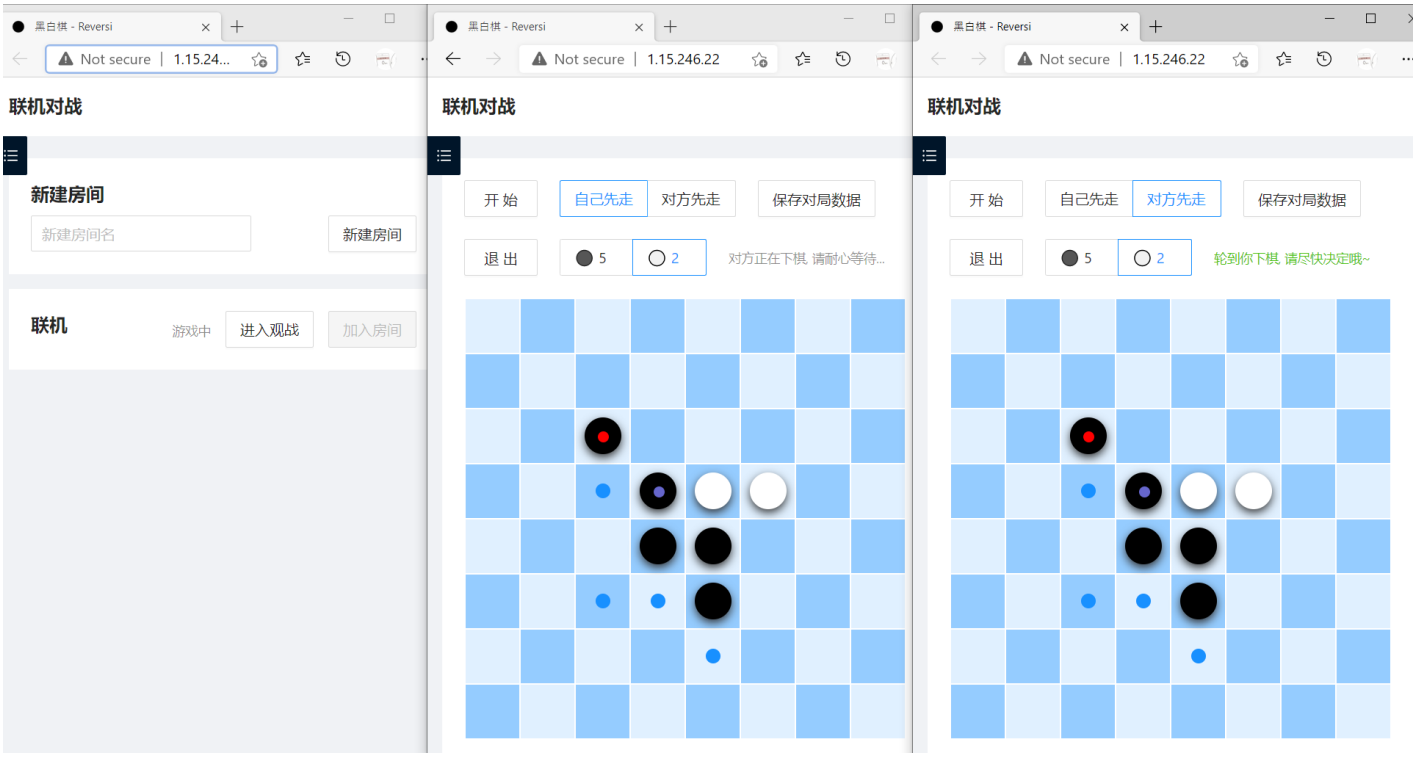
还有我自己的 AI: OrangeX5.



2.3 联机对战

经过好几天的奋斗, 终于加入了联机功能!

虽然实现的不是很优雅, 但是支持十个对局的 "高 (di) 并发" 应该没问题??

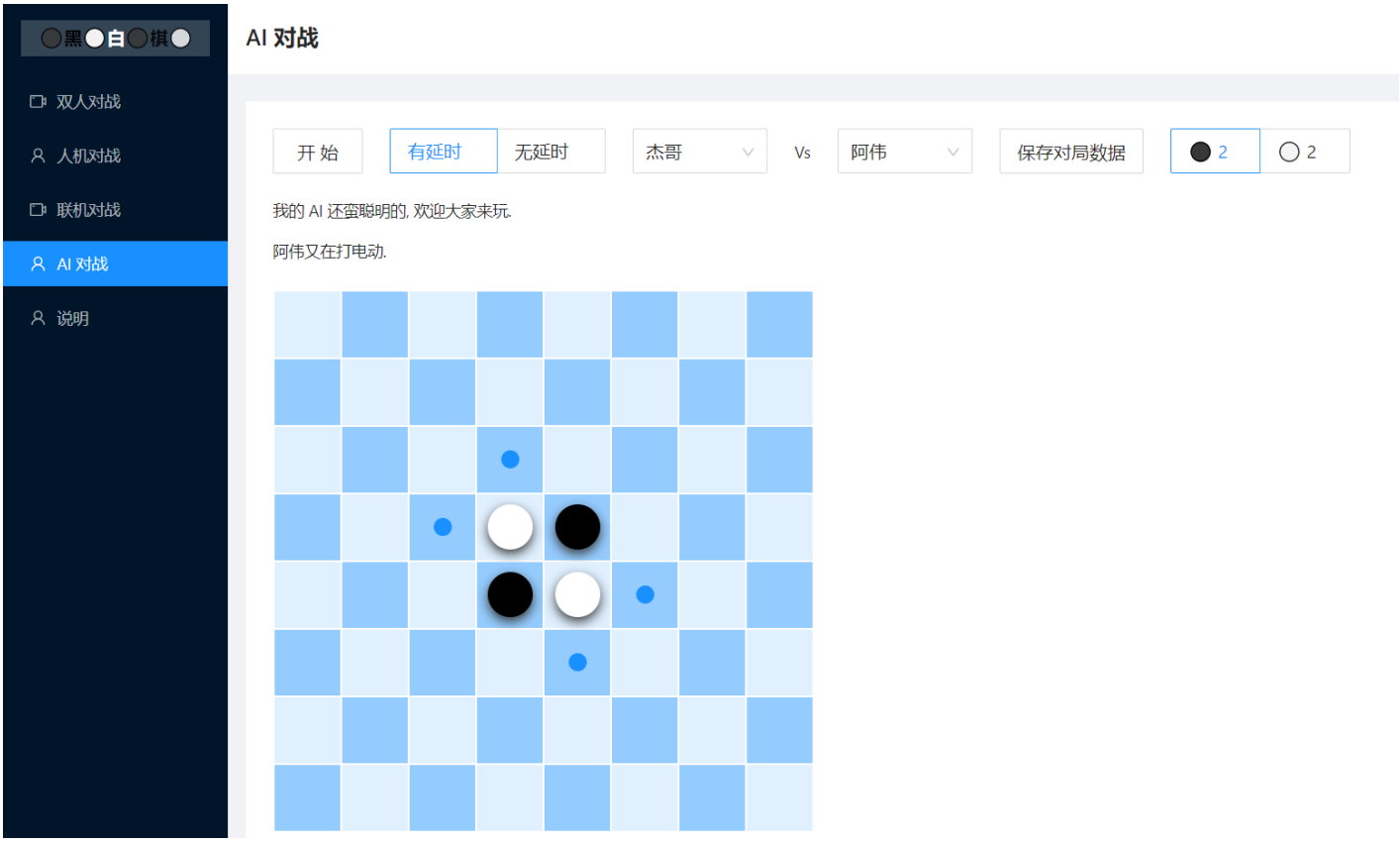


2.4 AI 对战

我们可以让 AI 打 AI!

这样就能快乐地以可视化的方式看 AI 打架了.

(虽然服务器可能撑不住...)



2.5 使用技术

- 前端必备三个基础知识: **HTML + CSS + JS**
- 前端框架: **React (React Hook + Typescript)**
- 组件库: **Ant Design**

GitHub 地址: https://github.com/OrangeX4/Reversi-Front

3. 黑白棋 AI

OrangeX5 使用了最朴实无华的算法.

基本使用了以下的方法:

1. **广度优先**, 深度设定为 4 ~ 6 层, 和蒙特卡洛的深度优先不同.
2. **Alpha-Beta 剪枝**, 在深度一定的情况对结果没有影响, 但是可以加快运算效率.
3. **缓存**, 对于一个纯函数 (pure function) 来说, 输入一定, 输出就一定, 所以可以缓存起来.
4. **评估函数** (evaluate), 用于评估当前局面:
    1. **角点策略**
    2. **稳定子策略**
    3. **前沿子策略**
    4. **行动力策略**

其实还有许多其他的策略, 因为实现起来没有上面的策略简单, 所以没有加进去.

3.1 广度优先

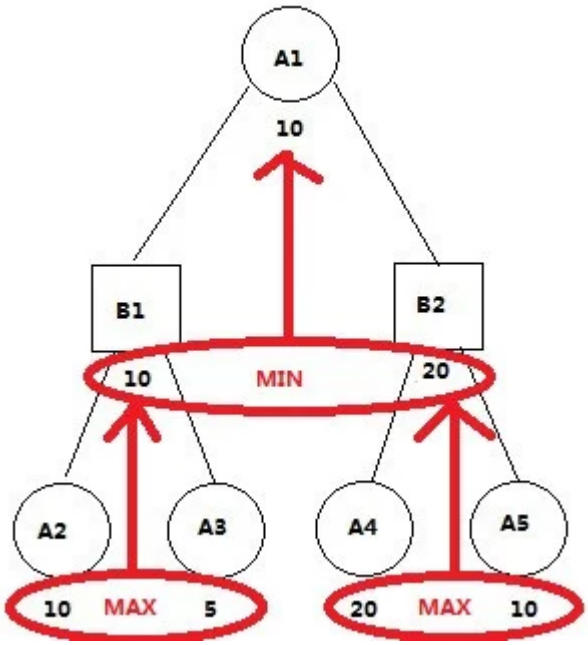
广度优先搜索算法 (英语: Breadth-First Search, 缩写为 BFS), 又译作宽度优先搜索, 或横向优先搜索, 是一种图形搜索算法. 简单的说, BFS 是从根节点开始, 沿着树的宽度遍历树的节点. 如果所有节点均被访问, 则算法中止.

对于棋类游戏来说, 我们自然不可能从一开始就把所有的局面都访问一遍, 因为可能性太多了, 对于黑白棋来说, 可达 $2^{60} = 1152921504606846976$ 种, 即 10^{10} 亿种可能性, 假设计算机每秒能够处理 100 万 个局面, 也要 10^{12} 秒, 即 10000 **年**才能算出一步.

所以我们不太可能直接运算到结尾, 那么我们就需要限定搜索的深度, 达到一定深度后, 用一种方式评估当前局面的好坏 (即评估函数 `evaluate`), 来确定要不要下这步子.

假设我们已经能够评估一个局面的好坏, 那么我们怎么根据评估的结果, 来判断我们要下的地方呢?

我们拿一个简单的例子来举例:



图片来源: www.dreamwings.cn

在这张图中, **圆圈代表自己, 方块代表对手**, 其中 A_1 是我们的一个可以下的棋, 我们可以下的位置是 $A_1, A_1', A_1'' \cdots$. 在这里我们考虑 A_1 点, 我们要做的是, 得到 A_1 这一步的分数.

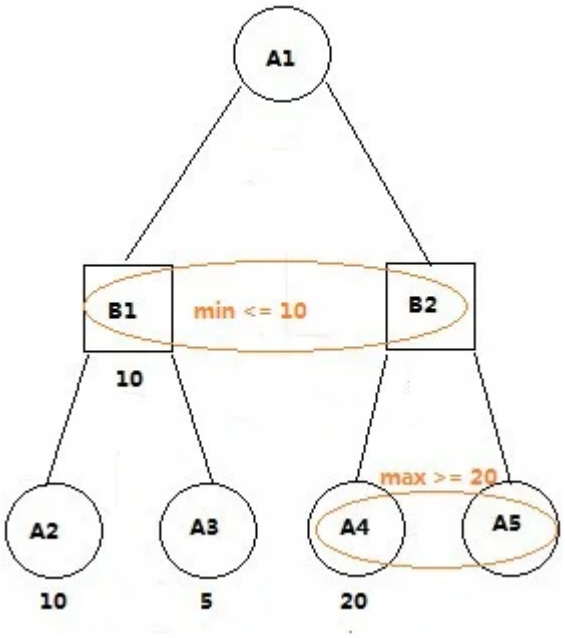
假设我们设定深度为 3 层, 在第 3 层的时候, 我们对当前局面进行评估, 即调用 `evaluate` 函数, 得到当前局面的得分, 在图中分别是 A_2, A_3, A_4, A_5.

我们可以看出, 我们虽然想要选 B_2 下的 A_4, 它的分数最高, 有 20 分, 但是对手肯定不会让我们如愿以偿, 所以对手肯定是会选 B_1, 那么我们只能选最小值 B_1 对应的最大值 A_2.

或者说我们从对手的角度来理解, 我们取得高分, 即对手取得低分, 所以只要**在返回上一层**的时候将分数取负, 那就变成了对手的分数, 这时候我们可以**统一为取最大值**.

3.2 Alpha-Beta 剪枝

这种方法遍历层数过深的话, 还是会很慢. 所以我们需要想办法将其优化.



在我们已经知道了 B_1 点的分值是 10 的时候, 如果我们求出 A_4 点的分值是 20, 可以看出 $B_1 = 10 < A_4 = 20$, 这时候知道, B_2 已经有一个值大于 B_1 了, 那么 $B_2 > B_1$ 是必定成立的事, 由对手要选最小值我们可以知道, 一定不会选 B_2, 那么 A_5 的局面自然也就不必评估了.

这样, 我们就达到了减去无用枝叶的效果.

更多内容可以前去 Alpha-Beta 剪枝算法 了解.

这部分我的实现为:

```python
def alpha_beta(board, current, alpha, beta, depth, is_pass = False):
    '''
    alpha-beta 剪枝算法
    参数:
    board: 棋盘数据，0 代表空格，1 代表黑棋，-1 代表白棋;
    current: 当前是谁下，传入 1 或 -1;
    alpha: 区间 [alpha, beta] 中的下限 alpha，通过 -alpha 变成下层的 beta;
    beta: 区间 [alpha, beta] 中的上限 beta，用于判断是否剪枝;
    depth: 深度，一般设为 4 ~ 6;
    is_pass: 用于判断上次是否也无子可下，上次和这次均无子可下的话，游戏结束.
    返回值:
    best_value: 评分;
    best_position: 对应的位置，如 (0, 2).
    '''
    # 开始递归判断
    best_value = -inf
    best_position = (-1, -1)
    movables = get_movables(board, current)
    for movable, reversals in movables.items():
        # 初始化
        value = 0
        new_board = get_new_board(board, current, movable, reversals)
        if depth == 0:
            # 深度为 0 时，评估当前局面，并且取反以表示当前分数
            value = -evaluate(new_board, -current)
        else:
            # 深度不为 0 时，递归调用
            value = -alpha_beta(new_board, -current, -beta, -alpha, depth - 1)[0]
        # 如果触发剪枝
        if value >= beta:
            return value, movable
        # 如果优于前面的下法
        if value > best_value:
            best_position = movable
            best_value = value
            # 更新下限
            if value > alpha:
                alpha = value
    # 无子可下的情况
    if len(movables) == 0:
        if is_pass:
            # 游戏结束，直接获取最后结果，并且得分是像 10000 这种极端分数
            best_value = get_score(board, current)
        else:
            # 没结束的话，就继续呗
            best_value = -alpha_beta(board, -current, -beta, -alpha, depth, True)[0]

    # 处理意外情况，随便选一个
    if best_position == (-1, -1) and len(movables.keys()) > 0:
        movables.keys()[randint(0, len(movables) - 1)]
    return best_value, best_position
```

3.3 缓存

函数式编程有一个很重要的原则: 尽量使用**纯函数** (pure function).

什么是**纯函数**? 纯函数就是, **给定相同的输入**, 一定**有相同的输出**, 且**没有任何副作用**的函数. 数学函数就是真正的纯函数, 如三角函数 `sin`, 给定相同的输入 `x`, 一定会有相同的输出 `y`.

纯函数有什么好处呢? **可控, 没有副作用, 支持高并发, 可以很方便地缓存**.

支持高并发是因为, 每个用户均有自己的数据, 他们不需要争夺对同一个数据的使用和修改权, 所以可以很方便地将纯函数部署到不同的服务器上, 而不用担心资源共享, 服务器通信和资源锁之类的麻烦问题.

而**缓存**就更为简单了, 这是一种用空间换时间的策略. 对于运算复杂耗时长的纯函数来说, 我们只需要, **在每一次运行这个函数时**, 把 **输入到输出的映射** 记录在一个哈希表中, 下一次运行这个函数的时候, 可以先看看这个输入在不在哈希表中, 在的话, 直接使用表中的数据, 这就是缓存.

要实现这个缓存策略非常简单, 对于这个 Alpha-Beta 算法, 我们可以实现下面的方法:

```
# 进行 Hash 缓存
hash_board_map = {}

def set_hash_board(board, current, depth, alpha, beta, value, pos):
    hash_board_map[tuple([tuple([0 if piece == 0 else (1 if piece == current else -1) for pie

def get_hash_board(board, current, depth, alpha, beta):
    key = tuple([tuple([0 if piece == 0 else (1 if piece == current else -1) for piece in lin
    return hash_board_map.get(key)
```

然后只需要修改一下原来的 Alpha-Beta 剪枝函数, 就可以加入缓存的功能:

```
def alpha_beta(board, current, alpha, beta, depth, is_pass = False):
    # 查看缓存中有没有, 有就直接返回
    cache = get_hash_board(board, current, depth, alpha, beta)
    if cache:
        return cache
    # 保存好 alpha, beta, 便于后续缓存
    saved_alpha = alpha
    saved_beta = beta

    # 开始递归判断
    # ...
    # 此处省略中间的代码
    # ...

    # 在返回之前保存数据到缓存中
    set_hash_board(board, current, depth, saved_alpha, saved_beta, best_value, best_position)
    return best_value, best_position
```

3.4 评估函数

我们需要在达到预定的深度之后, 计算当前局面的分数, 所以我们需要一个**评估函数** (evaluate).

想要正确评估出当前局面的好坏, 我们需要考虑各种因素, 使用多种策略.

3.4.1 角点策略

这是一张黑白棋下棋优先级的图:

1	5	3	3	3	3	5	1
5	5	4	4	4	4	5	5
3	4	2	2	2	2	4	3
3	4	2			2	4	3
3	4	2			2	4	3
3	4	2	2	2	2	4	3
5	5	4	4	4	4	5	5
1	5	3	3	3	3	5	1

我们可以看出, 四个角的点的优先级是最高的, 即可以占就一定要占, 且尽量不让对手占领.

为什么呢? 因为角点是最经典的稳定子, 这跟我们下面要说的稳定子策略也有关系. 并且只要占领了角点, 就很容易将别人下的棋子翻转过来.

并且根据角点, 我们还可以了解到, 与角点相邻的点最好别下, 否则对手很容易就能占领角点, 进而将你的棋子翻转. 与角点相隔一个格的点也是比较好的点, 因为这样可以诱惑对方下临角点, 进而你可以占领角点.

3.4.2 稳定子策略

稳定子是什么?

稳定子就是不管怎么样下都不会被对手翻转的棋子, 称为稳定子. 角点是最经典的稳定子, 当你下了角点之后, 角点旁边的边点也是稳定子. 所以, 如果要简化计算, 我们可以只关注角和边的稳定子.

3.4.3 前沿子策略

前沿子, 又称为**边缘子**, 和**内部子**概念相反.

内部子是八面均不为空的棋子, 即被棋子包裹着, 不会被对手翻转 (但是仍可能被自己翻转), 我们需要多下这种棋, 可以给一定的正面评分.

前沿子是八面至少有一面为空的棋子, 这种棋可以给一定的负面评分.

3.4.4 行动力策略

在前期, 很少能走到边和角, 这时候的策略应该以什么为标准呢?

我们可以以**行动力**为标准.

什么是**行动力**? **行动力**即当前可以放置的位置的个数. 自己的行动力越高, 就越可能找到一个好的落子让自己有更好的局面; 对手的行动力越低, 就越容易被迫走出一个 "坏子".

在前期如果以行动力为标准的话, 就很容易导致自己在前期的子**比较少**, 因为前期自己的子越少, 行动力一般越高, 四面八方都可以放. 这就隐含了另一个策略, **消失策略**: 在棋盘比试的前期,己方的子越少往往意味着局势更优. 因此在前期可采用使己方的子更少的走子.

3.4.5 具体实现

```python
def evaluate(board, current):
    '''
    评估函数，用于评估当前局面.
    常用策略有角点，稳定子，前沿子，行动力.
    '''
    # 行动力，即可以走的步数
    mobility = len(get_movables(board, current).keys())

    # 前沿子，即周围至少有一个空格的棋子，这种棋子容易被吃掉
    frontier = 0
    def is_frontier(i, j):
        for dy, dx in [(dy, dx) for dy in range(-1, 2) for dx in range(-1, 2) if dy != 0 or d
            if board[i + dy][j + dx] != 0:
                return True
        return False
    # 遍历时可以去掉边缘
    for i in range(1, 7):
        for j in range(1, 7):
            # 棋盘上的棋子不为空时，判断前沿子
            if not board[i][j] == 0 and is_frontier(i, j):
                # 容易被吃掉，所以应该要取反
                frontier -= board[i][j] * current

    # 角点和稳定子
    corner = 0
    steady = 0
    corner_map = [
        # 角点 i, j, 偏移方向 dy, dx
        [0, 0, 1, 1],
        [0, 7, 1, -1],
        [7, 0, -1, 1],
        [7, 7, -1, -1]
    ]
    for corner_i, corner_j, dy, dx in corner_map:
        if board[corner_i][corner_j] == 0:
            # 角点为空时，如果下了临角点或对角点，这些点很危险
            corner += board[corner_i][corner_j + dx] * current * -3
            corner += board[corner_i + dy][corner_j] * current * -3
            corner += board[corner_i + dy][corner_j + dx] * current * -6
            # 角点为空时，如果下了隔角点，这些点很好
            corner += board[corner_i][corner_j + 2 * dx] * current * 4
            corner += board[corner_i + 2 * dy][corner_j] * current * 4
            corner += board[corner_i + dy][corner_j + 2 * dx] * current * 2
            corner += board[corner_i + 2 * dy][corner_j + dx] * current * 2
        else:
            i, j = corner_i, corner_j
            # 角点的权值
            corner += board[corner_i][corner_j] * current * 15
            # 角点不为空时，处理稳定子，为了简化运算，仅仅考虑边稳定子
            current_color = board[corner_i][corner_j]
            while 0 <= i <= 7 and board[i][corner_j] == current_color:
                steady += current * current_color
                i += dy
            while 0 <= j <= 7 and board[corner_i][j] == current_color:
```

```
                steady += current * current_color
                j += dx


    # 以一定的权重相乘之后返回
    return 8 * corner + 12 * steady + 8 * mobility + 4 * frontier
```

3.5 其他

一些辅助用的函数, 获取可以走的位置和创建新棋盘, 外加 AI 的使用:

```python
from math import inf
from random import randint
from copy import deepcopy
from typing import List

def fsj_ai(board: List[List[int]], current: int, newest: List[int], reversal: List[List[int]]
    '''
    输入参数:
    board: 二维数组，8 x 8 的棋盘数据，0 代表空，1 代表黑棋，2 代表白棋.
    current: 当前你的棋子颜色，1 代表黑棋，2 代表白棋.
    最重要的就是上面两个，其他输入无关紧要.
    newest: 对方下的最后一个棋子位置.
    reversal: 对方上一次翻转的棋子.
    prompt: 当前你可以下的位置，即提示. 一般来说你并不需要它.

    返回:
    返回你要下的位置，例如 [2, 1] 或 (2, 1)，要注意是从 0 开始的.
    '''

    # 获取可放置位置的数据
    board = [[-1 if piece == 2 else piece for piece in line] for line in board]
    current = -1 if current == 2 else current

    # 如果可以放角点，直接放角点
    movables = get_movables(board, current)
    # for corner in [(0, 0), (0, 7), (7, 0), (7, 7)]:
    #     if corner in movables.keys():
    #         return corner

    # alpha-beta 剪枝
    # 可行点的个数到迭代深度的映射
    movables_number = len(movables.keys())
    if movables_number < 5:
        return alpha_beta(board, current, -10000, 10000, 6)[1]
    elif movables_number < 10:
        return alpha_beta(board, current, -10000, 10000, 5)[1]
    else:
        return alpha_beta(board, current, -10000, 10000, 4)[1]

def get_movables(board, current):
    # 对于一个方向是否可以放置，比如向右边是 dy = 0, dx = 1 的情况
    def get_movable_by_step(i, j, dy, dx):
        result = []
        isEnd = False
        while True:
            i += dy
            j += dx
            if 0 > i or i > 7 or 0 > j or j > 7 or board[i][j] == 0:
                break
            elif board[i][j] == -current:
                result.append((i, j))
                isEnd = True
            elif board[i][j] == current and isEnd == False:
                break
```

```
            elif board[i][j] == current and isEnd == True:
                return result
        return []

    # 八个不同的方向是否可行
    def get_movable_for_all_direction(i, j):
        result = []
        for dy, dx in [(dy, dx) for dy in range(-1, 2) for dx in range(-1, 2) if dy != 0 or d
            result += get_movable_by_step(i, j, dy, dx)
        return result

    # 对棋盘的每一个空格进行判断
    result = {}
    for i in range(8):
        for j in range(8):
            lst = get_movable_for_all_direction(i, j)
            if board[i][j] == 0 and len(lst) > 0:
                result[(i, j)] = lst
    return result


def get_new_board(board, current, newest, reversals):
    board = deepcopy(board)
    board[newest[0]][newest[1]] = current
    for reversal in reversals:
        board[reversal[0]][reversal[1]] = current
    return board

def get_score(board, current):
    black_count = 0
    white_count = 0
    for i in range(8):
        for j in range(8):
            if board[i][j] == 1:
                black_count += 1
            elif board[i][j] == -1:
                white_count += 1
    # 反正赢了就不用考虑比分来估值了
    return (black_count - white_count) * current * 10000
```

4. 参考

1. 黑白棋 - 维基百科
2. miniAlphaGo-for-Reversi - GitHub
3. 黑白棋中的 AI - 千千
4. 黑白棋天地