智慧港口自动化调度系统

何玥仪（baseline+优化1-2）张潇涵（优化3-6）

1. 项目概述

背景：

在一个现代化全自动港口中，你需要编写一个中央控制算法，通过调度机器人（Robot）和轮船（Ship）协同工作，将源源不断产生的货物（Goods）从仓库区搬运至泊位（Berth），并通过轮船运输变现。

目标：

在规定的 1000 帧（Frames） 时间限制内，通过合理的路径规划和任务分配，尽可能多地赚取资金。

1. 任务分配

“哪个机器人去搬哪件货”本质是一个分配问题（机器人-货物二分匹配），使用评分 + 全局贪心近似。

2. 多机器人路径规划

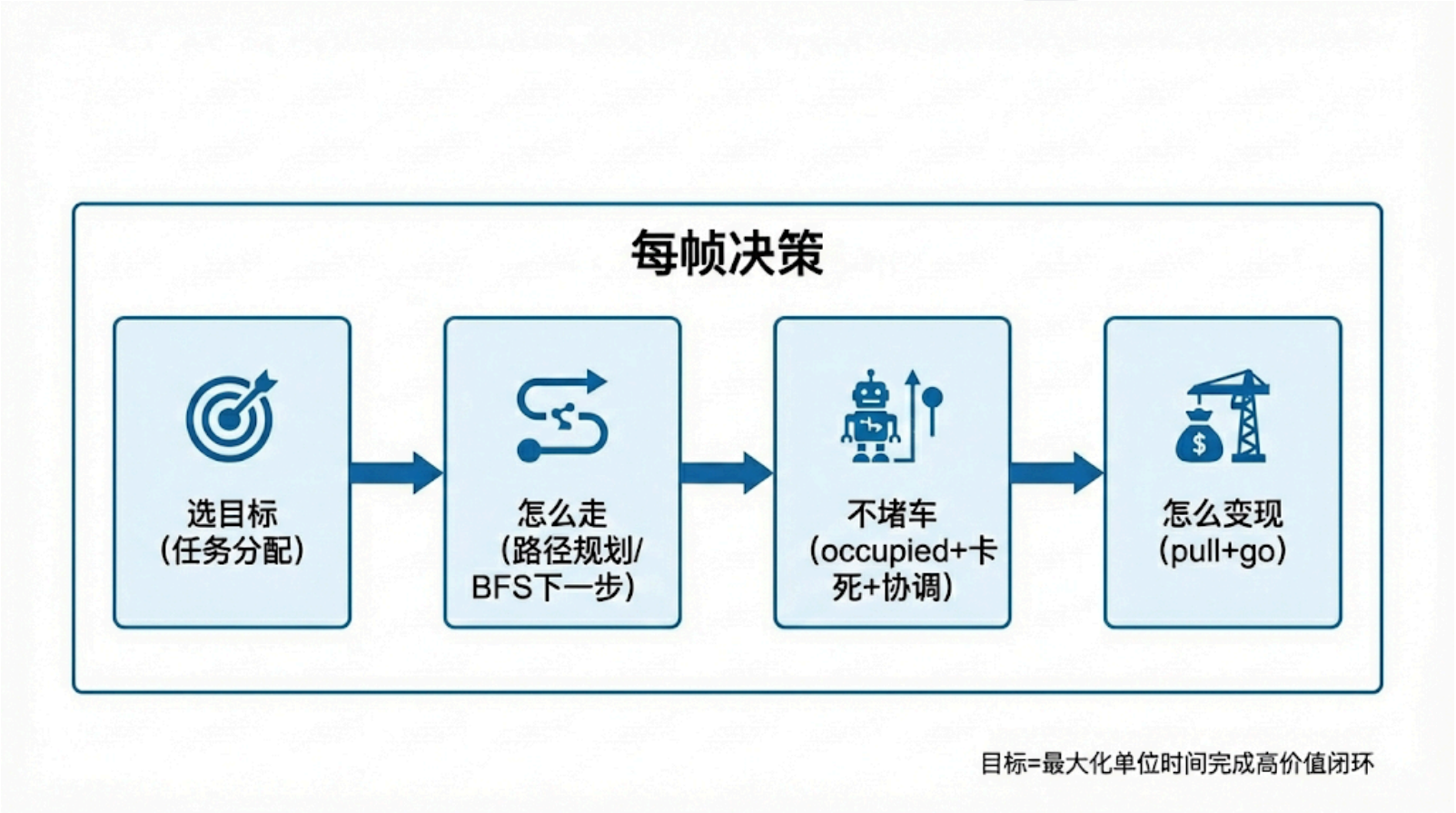
网格地图 + 障碍 + 多机器人同时移动 + 冲突约束（不能进被占格），属于多智能体路径规划问题。

3. 吞吐最大化

只有轮船go才得分，得分来自闭环：get → move → pull → go，目标等价于在固定帧数1000内最大化完成闭环的总价值。

2. 建模拆解：四个子问题

1. **选目标**：哪个机器人去搬哪件货?
2. **怎么走**：地图有障碍，下一步怎么走?
3. **防止碰撞**：机器人同时走同一个位置，怎么办?
4. **怎么变现**：货到了泊位还不算分，资金只在轮船执行 `go` 时增加



3. Baseline

3.1 选目标：先让机器人“有事做”

- 空载机器人：选一个货物当目标（最朴素：选“看起来最近”的货物）
- 载货机器人：选一个泊位当目标（最朴素：选“看起来最近”的泊位）

```
if (!robots[i].has_goods) {
    // 空载，target = nearest_good(...)
} else {
    // 载货，target = nearest_berth(...)
}
```

- 用曼哈顿距离估算：d = |x - tx| + |y - ty|，判断最近

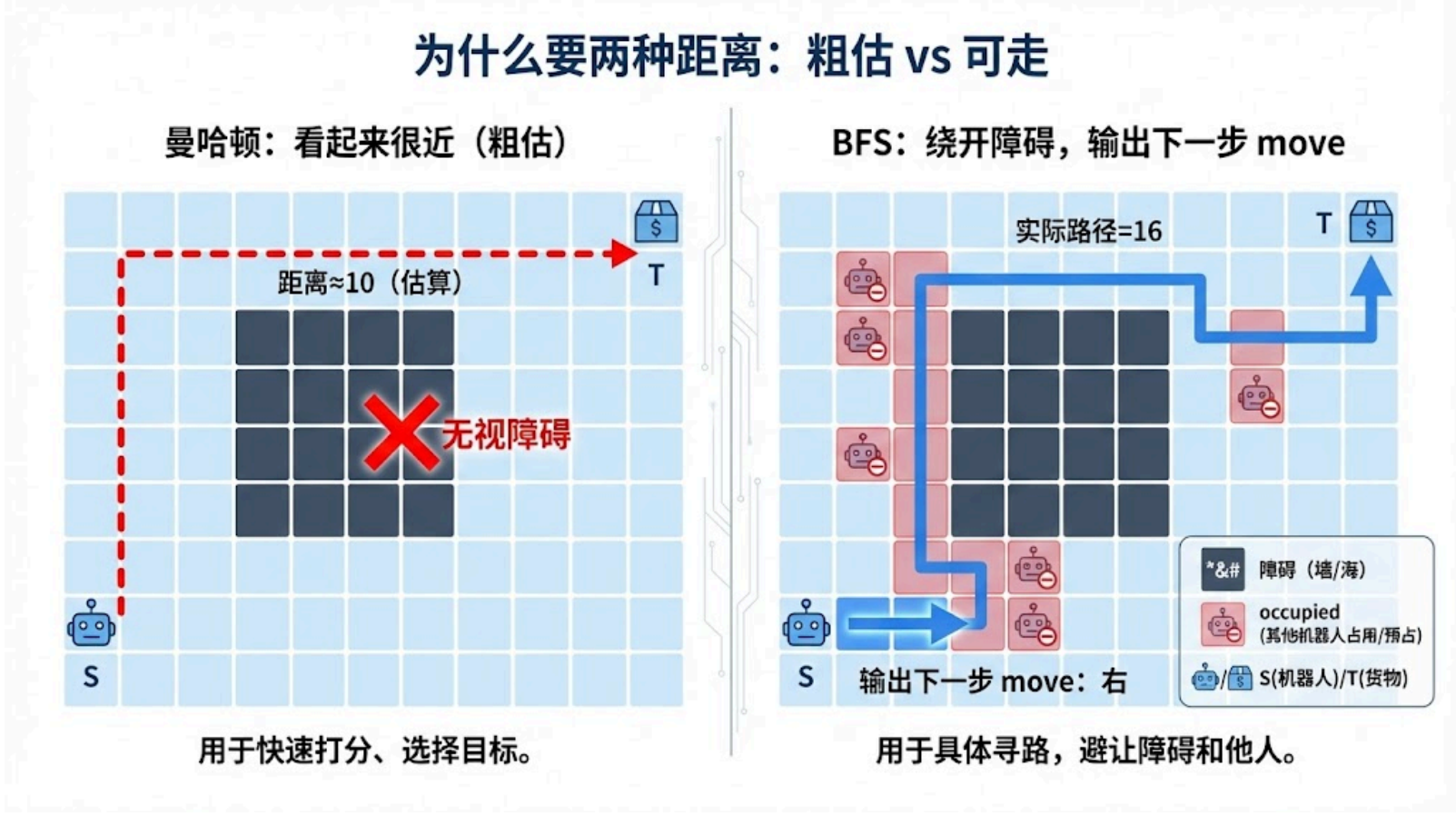
3.2 怎么变现：到点就 pull，船持续 go

```
// 载货到泊位：pull；船每帧 go；最后必须输出 OK
if (robots[i].has_goods && robots[i].x == berth_x && robots[i].y == berth_y) cout << "pull " << i << "\n";
for (int s = 0; s < 5; s++) cout << "go " << s << "\n";
cout << "OK\n";
```

- 这保证只要有货被送到泊位并装上船，就能尽快通过 go 把钱落袋。

3.3 怎么走：每帧只走一步（输出 move）

```
// 如果不在目标格，就规划“下一步方向 dir”，然后输出 move
int dir = bfs(robots[i].x, robots[i].y, tx, ty); // baseline：只避障，不考虑别人占位
if (dir != -1) cout << "move " << i << " " << dir << "\n";
```



4. Baseline 的问题

```
Starting Process: main.exe
--- Simulation Start (Total Frames: 1000) ---
Frame 100: Money = 35
Frame 200: Money = 538
Frame 300: Money = 931
Frame 400: Money = 1292
Frame 500: Money = 1536
Frame 600: Money = 1798
Frame 700: Money = 2078
Frame 800: Money = 2315
Frame 900: Money = 2527
Frame 1000: Money = 2678
--- Game Over ---
Final Score: 2678
```

cerr调试节选

```
[v0] frame=500 money=1501 goods=44
[v0] frame=600 money=1669 goods=49
[v0] frame=700 money=1721 goods=50
[v0] frame=800 money=1736 goods=50
[v0] frame=900 money=1818 goods=50
[v0] frame=1000 money=1818 goods=50
```

- `goods` 最终到达 50（上限）：说明系统吞吐量不足以消化新增货物，单位时间内完成运货到轮船go的数量太少。

5. 优化一：occupied（冲突规避）+ stuck_count（卡死检测）

- baseline没有考虑：**机器人碰撞**和 **卡住**。
- 这一版先做最基础的改造：让机器人**看见别人**（occupied），并能识别自己是否**真的卡住了**（stuck_count）。

(1) 遇到occupied就绕过

```
// BFS 扩展时：把 occupied 当作不可走（避开别人/已预占位置）
if (grid[nx][ny] == '*' || grid[nx][ny] == '#') continue;
if (occupied[nx][ny]) continue;
```

(2) 预占下一格：把“我准备走到哪”告诉后面的机器人

```
// move 成功后：预占下一格（让后处理的机器人“自动绕开”）
int nx = r.x + dx[dir], ny = r.y + dy[dir];
occupied[nx][ny] = true;
```

- 直观理解：同一帧里，我们按顺序处理机器人；先处理的机器人“先占位”，后处理的机器人在 BFS 时会避开它。

(3) stuck_count：把“原地踏步”变成可触发的信号

```
// 卡死检测：连续多帧坐标不变 -> stuck_count++
if (r.x == r.last_x && r.y == r.last_y) r.stuck_count++;
else r.stuck_count = 0;
r.last_x = r.x; r.last_y = r.y;
```

```
// 规划前临时释放自己：否则 BFS 会把“自己站的格子”也当成 occupied 障碍
occupied[r.x][r.y] = false;
```

(4) 解锁手段：随机逃逸

```
// 卡住 + 没路：随机试几次，找一个“可走且不占用”的方向先动起来
if (r.stuck_count > 5 && dir == -1) {
  for (int t = 0; t < 4; t++) {
    int d = rand() % 4;
    int nx = r.x + dx[d], ny = r.y + dy[d];
    if (nx < 0 || nx >= N || ny < 0 || ny >= N) continue;
    if (grid[nx][ny] == '*' || grid[nx][ny] == '#') continue;
    if (!occupied[nx][ny]) { dir = d; break; }
  }
}
```

- 这一步的目的不是“走得最优”，而是**先解锁拥堵**，把系统从僵持态拉回可流动状态；后续会用“启发式避让”替代随机逃逸。

优化1️⃣运行结果

```
Starting Process: main.exe
--- Simulation Start (Total Frames: 1000) ---
Frame 100: Money = 44
Frame 200: Money = 532
Frame 300: Money = 1398
Frame 400: Money = 1974
Frame 500: Money = 2413
Frame 600: Money = 2990
Frame 700: Money = 3342
Frame 800: Money = 3661
Frame 900: Money = 3914
Frame 1000: Money = 4219
--- Game Over ---
Final Score: 4219
```

6. 优化二：性价比选货（从“最近”到“单位时间收益”）

- baseline 的“最近优先”只回答：**我离它近不近**，但不回答：**搬它值不值**。
- 改进：把“值钱”与“路程”合成一个分数，近似成“每走一步能赚多少钱”。

评分公式

$$\text{score} = \frac{\text{val}}{(|x_r - x_g| + |y_r - y_g|) + 1}$$

```
// “看起来最近”的距离：曼哈顿距离 d = |dx| + |dy|
inline int manhattan(int x1, int y1, int x2, int y2) {
  return abs(x1 - x2) + abs(y1 - y2);
}

double score_v2(const Robot& r, const Goods& g) {
  int d = manhattan(r.x, r.y, g.x, g.y);
  return 1.0 * g.val / (d + 1); // +1：避免除0，也避免 d=0 得分“过爆”
}
```

- 解释：`val` 越大越好、`d` 越小越好；分数高 ≈ “单位路程收益高”。

```
// 对机器人 i：遍历所有货物，挑 score 最大的那个
int best_gid = -1;
double best = -1;
for (int gid = 0; gid < (int)goods_list.size(); gid++) {
  double s = score_v2(robots[i], goods_list[gid]);
  if (s > best) { best = s; best_gid = gid; }
}
robot_target_good[i] = best_gid;
```

- 这一步解决"挑货只看距离"的问题，但仍有新问题：

多个机器人可能挑到同一件高分货，这个时候怎么分配?

7. 优化三：全局贪心分配

- 做法：把所有 (机器人, 货物) 组合都算一遍分数，按分数从高到低依次分配
- 约束：每个机器人最多一个目标、每个货物最多分配一次（避免抢同货）

```
// 候选对：把“机器人 i 搬货 j”的收益用 score 表示，score 越大越优先
struct Candidate {
    int rid, gid;
    double score;
    bool operator<(const Candidate& o) const { return score > o.score; } // 降序
};
```

- 这一步把“分配问题”转成一个可排序的列表：谁搬哪件货最划算，就排到最前面优先分配。

```
vector<Candidate> cands;
for (int i = 0; i < ROBOT_NUM; i++) {
    if (robots[i].status == 0 || robots[i].has_goods) continue;
    for (int j = 0; j < (int)goods_list.size(); j++) {
        double s = score_final(i, j); // 包含 berth_dist 的闭环评分（见 §9）
        if (s < 0) continue;          // 不可达泊位/无效货物
        cands.push_back({i, j, s});
    }
}
sort(cands.begin(), cands.end());
```

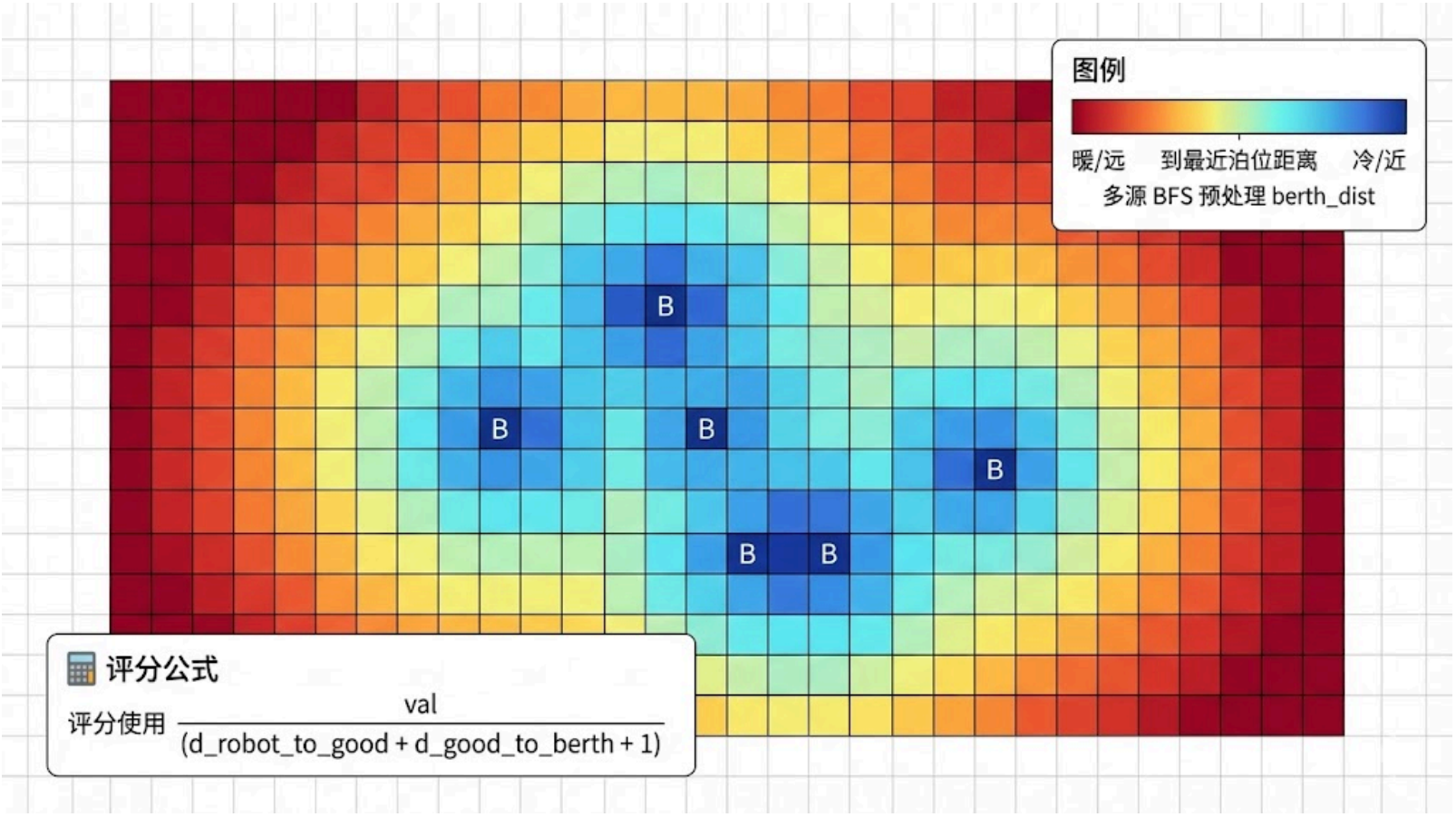
- 过滤 `s < 0`：直接把“无法送到泊位/无意义”的货物排除，避免浪费机器人。

```
vector<int> robot_target_good(ROBOT_NUM, -1);
vector<char> good_used(goods_list.size(), 0);
for (const auto& c : cands) {
    if (robot_target_good[c.rid] != -1) continue;
    if (good_used[c.gid]) continue;
    robot_target_good[c.rid] = c.gid;
    good_used[c.gid] = 1;
}
```

- 贪心规则非常直观：按 score 从高到低扫一遍，遇到“机器人没被分配且货没被抢走”就确定匹配。
- 虽然不保证全局最优，但实现简单、效果稳定。

8. 优化四：泊位距离场 berth_dist（把“送货成本”也算进来）

- 我们之前只考虑了**选货**的成本，但是没考虑**送货到泊位**的成本
- 先预处理一个距离场 `berth_dist[x][y]`：从任意格到最近泊位要走多少步（多源 BFS 一次算完），这样选货时就能把“后半程送货成本”一起考虑



```
// 多源 BFS：所有泊位同时入队，得到每个格子到最近泊位的最短距离 berth_dist
void init_berth_dist() {
    memset(berth_dist, -1, sizeof(berth_dist));
    queue<pair<int, int>> q;
    for (auto b : berths) { berth_dist[b.first][b.second] = 0; q.push(b); }
```

- 把所有泊位当作“同时出发的起点”，一次 BFS 就能得到“任意格到最近泊位”的最短距离。

```
    while (!q.empty()) {
        auto cur = q.front(); q.pop();
        int x = cur.first, y = cur.second;
        for (int dir = 0; dir < 4; dir++) {
            int nx = x + dx[dir], ny = y + dy[dir];
            if (nx < 0 || nx >= N || ny < 0 || ny >= N) continue;
            if (grid[nx][ny] == '*' || grid[nx][ny] == '#') continue;
            if (berth_dist[nx][ny] != -1) continue;
            berth_dist[nx][ny] = berth_dist[x][y] + 1;
            q.push({nx, ny});
        }
    }
}
```

9. 评分升级：把"人→货 + 货→泊位"合并成闭环成本

$$\text{score}(r,g) = \begin{cases} \dfrac{val(g)}{d_{rg} + d_{gb} + 1} \end{cases}$$

其中$d_{rg} = |x_r - x_g| + |y_r - y_g|$是曼哈顿距离；$d_{gb}$也就是刚刚多源$BFS$得到的货 → 泊位距离$berth_dist[nx][ny]$。

```
// main6 的闭环评分：人到货（快估算） + 货到泊位（距离场）
double score_final(int rid, int gid) {
    const Robot& r = robots[rid];
    const Goods& g = goods_list[gid];
    int d_rg = abs(r.x - g.x) + abs(r.y - g.y);
    int d_gb = berth_dist[g.x][g.y];
    if (d_gb == -1) return -1; // 无法送到任何泊位：忽略
    return 1.0 * g.val / (d_rg + d_gb + 1);
}
```

- 如果某个货物根本送不到任何泊位（`berth_dist == -1`），直接忽略，不浪费机器人

10. 优化五：智能避让

- 之前的随机逃逸：有时能解锁，但方向随机，可能越走越偏、浪费帧
- 启发式避让：卡住时不乱走，而是选择一个“可走且更接近目标”的方向

```
int heuristic_detour(int x, int y, int tx, int ty) {
    int best_dir = -1;        // 默认找不到可走方向
    int best_d = INT_MAX;     // 当前最优(最小)距离，先设成极大

    // 依次尝试四个方向（上/下/左/右，具体取决于 dx/dy 定义）
    for (int dir = 0; dir < 4; dir++) {
        int nx = x + dx[dir], ny = y + dy[dir];   // 假设往 dir 走一步后的新坐标

        // 1) 越界：直接排除
        if (nx < 0 || nx >= N || ny < 0 || ny >= N) continue;

        // 2) 障碍：'*' '#'(不可通行) 直接排除
        if (grid[nx][ny] == '*' || grid[nx][ny] == '#') continue;

        // 3) 冲突：该格已被其他机器人占/预占，直接排除（避免撞格）
        if (occupied[nx][ny]) continue;

        // 4) 评价“走这一步好不好”：走完后离目标还有多远（曼哈顿距离）
        int d = abs(nx - tx) + abs(ny - ty);

        // 5) 取 d 最小的方向：让下一步尽量靠近目标
        if (d < best_d) { best_d = d; best_dir = dir; }
    }

    return best_dir;  // -1 表示无可行绕行；否则返回最优方向
}
```

优化5执行结果

```
Starting Process: main.exe
--- Simulation Start (Total Frames: 1000) ---
Frame 100: Money = 79
Frame 200: Money = 470
Frame 300: Money = 1027
Frame 400: Money = 1561
Frame 500: Money = 2461
Frame 600: Money = 3009
Frame 700: Money = 3343
Frame 800: Money = 3996
Frame 900: Money = 4428
Frame 1000: Money = 4943
--- Game Over ---
Final Score: 4943
```

cerr 节选（来回震荡堵塞）

```
[main5] frame=1 money=0 goods=1
[main5] robot=0 stuck=3 at (22,1) start_detour
[main5] robot=1 stuck=3 at (35,31) start_detour
[main5] robot=6 stuck=3 at (67,57) start_detour
[main5] robot=7 stuck=3 at (84,39) start_detour
[main5] robot=9 stuck=3 at (98,31) start_detour

[main5] robot=0 stuck=3 at (22,0) start_detour
[main5] robot=1 stuck=3 at (35,30) start_detour
[main5] robot=7 stuck=3 at (84,38) start_detour
[main5] robot=9 stuck=3 at (98,32) start_detour

[main5] robot=0 stuck=3 at (22,1) start_detour
[main5] robot=1 stuck=3 at (35,31) start_detour
[main5] robot=7 stuck=3 at (84,39) start_detour
[main5] robot=9 stuck=3 at (98,33) start_detour
```

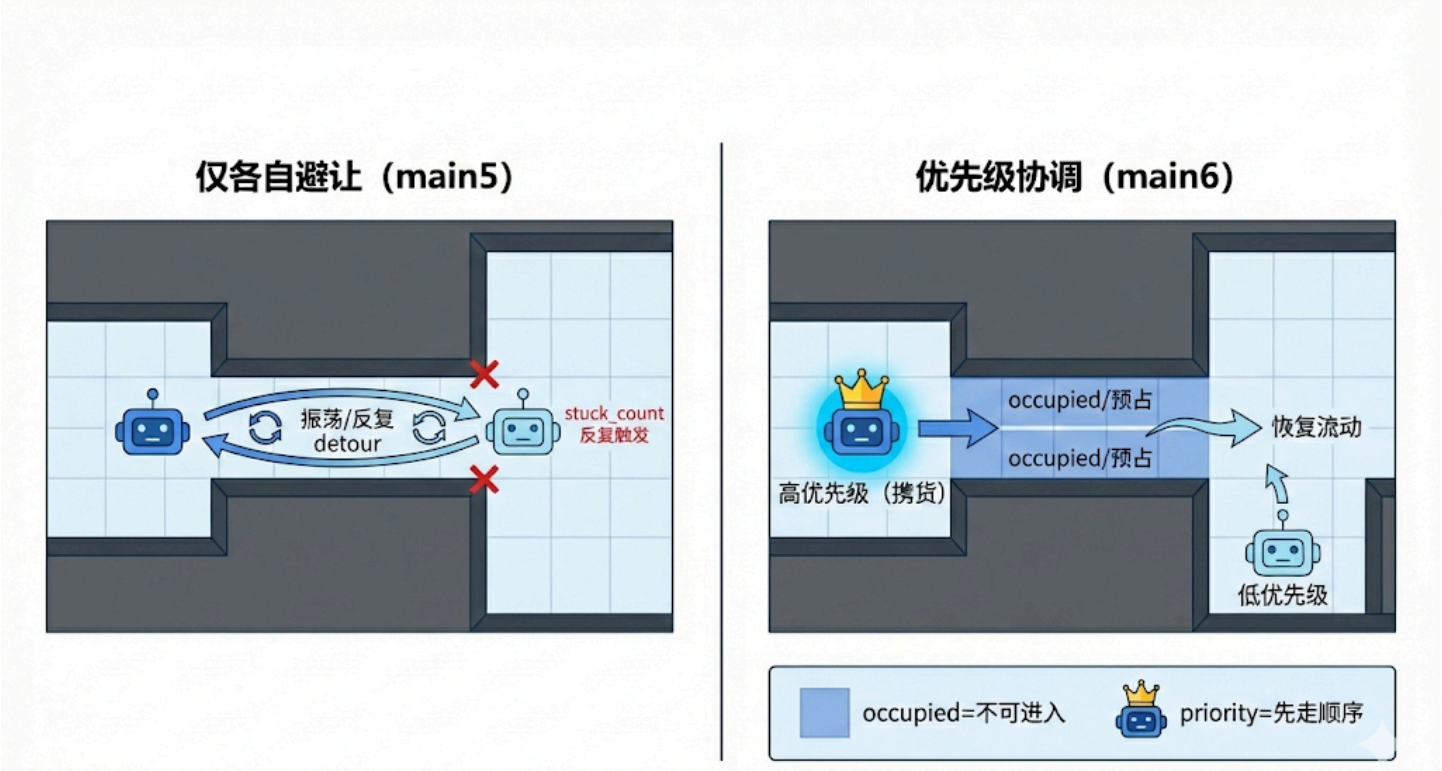
- stuck=3 的含义：机器人连续 3 帧坐标没变（或几乎没推进），系统判定“卡住”，开始 start_detour（启发式绕行）。
- 关键现象：同一批机器人在相邻坐标之间反复出现（例如 robot=0 在 (22,1) 和 (22,0) 来回，robot=1 在 (35,31) 和 (35,30) 来回）。
  这说明它们不是“找到了绕路并继续前进”，而是在狭窄区域互相抢位/互相堵住，导致来回试探（振荡）。

优化5的局限

- 启发式避让本质是“每个机器人各自选一步最合理的方向”(局部最优)。
- 在狭窄通道/瓶颈处，多个机器人容易形成对称局面：
  大家都想往前挤、都觉得自己该走，结果互相抢同一格/互相挡路，出现**来回振荡**

怎么解决？加一个“全局让路规则”

1. 给机器人定优先级，决定谁先走
2. 按照优先级降序执行



11. 优化六：走路优先级

先决定“谁更重要”，重要的先走、先占位，其他机器人看到占位就绕开

1. **优先级定义**：
   - 携带货物的机器人优先级最高（10000）：它们只差一步 `pull/go` 就能变现。
   - 未携带货物的机器人：优先级 = 目标货物价值（价值高的先拿）。
   - 无任务：优先级最低（0）。
2. **执行顺序调整**：
   - 不再按 ID 顺序（0-9）处理机器人，而是按优先级降序处理。
   - **高优先级先规划并移动**，先写入 `occupied[next]`；低优先级在 BFS/避让时看到 occupied 后自动绕开。

```
// 1) 计算优先级
vector<int> robot_priority(ROBOT_NUM, 0);
for (int i = 0; i < ROBOT_NUM; i++) {
    if (robots[i].has_goods) robot_priority[i] = 10000;
    else if (robot_target_good[i] != -1) robot_priority[i] = goods_list[robot_target_good[i]].val;
    else robot_priority[i] = 0;
}
```

```
// 2) 得到处理顺序（降序）
vector<int> p_order(ROBOT_NUM);
iota(p_order.begin(), p_order.end(), 0);
sort(p_order.begin(), p_order.end(), [&](int a, int b) {
    return robot_priority[a] > robot_priority[b];
});
```

- 这一步把“谁先走”固定下来：高优先级机器人先规划先占位，低优先级自然避让。

```
// 3) 按顺序规划：高优先级先“预占”下一格，低优先级自动避让
for (int k = 0; k < ROBOT_NUM; k++) {
    int i = p_order[k];
    occupied[robots[i].x][robots[i].y] = false;
    // move_dir = bfs_next_step(...) 或 heuristic_detour(...)
    // 若 move：occupied[nx][ny] = true;
}
```

- “优先级协调”真正生效的地方就在这里：规划顺序 + occupied 预占一起作用，减少狭窄通道里的振荡与死锁。

最终运行结果

```
Starting Process: main.exe
--- Simulation Start (Total Frames: 1000) ---
Frame 100: Money = 318
Frame 200: Money = 753
Frame 300: Money = 1335
Frame 400: Money = 1870
Frame 500: Money = 2367
Frame 600: Money = 3084
Frame 700: Money = 3704
Frame 800: Money = 4320
Frame 900: Money = 4982
Frame 1000: Money = 5651
--- Game Over ---
Final Score: 5651
```

- 重复执行 `python judge.py` 存在一点分数波动，但大幅高于优化5

心得体会

- 纯靠局部最优（例如每个机器人各自选最近/各自避让）在多机器人场景里很容易失效：拥堵、对撞、抢同货会把大量帧消耗在无效移动上。
- 优化不应该凭感觉，而要“可观测、可解释、可验证”。
  通过 cerr 输出关键事件（无路、卡死、绕行触发、货物数量与资金变化），我们能明确定位瓶颈，再针对性优化。
- 项目协作时用git版本控制是很有必要的。