|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **数据结构课程设计报告** | | | | |
|  | | | | |
| 题目：迷宫问题-2 | | | | |
|  | 专业： | ： | 软件工程 |  |
| 学号： | ： | 202300300075 |
| 姓名： | ： | 林星宇 |
| 任课教师： | ： | 吴昊 |
|  | ： |  |
| 2025年2月15日 | | | | |

目录

[一、系统开发平台 1](#_Toc20155)

[1.1 架构概述 1](#_Toc3819)

[1.2 开发平台、技术栈 3](#_Toc4054)

[1.3 开发环境配置 3](#_Toc30546)

[1.4 系统特点与创新 4](#_Toc13848)

[二、 问题描述 5](#_Toc15941)

[2.1 任务陈述 5](#_Toc11186)

[2.2 任务目标 5](#_Toc5948)

[三、 系统定义 6](#_Toc10273)

[3.1 系统输入 6](#_Toc28185)

[3.2 用户视图 7](#_Toc6316)

[3.3 系统组件 8](#_Toc12337)

[四、需求分析 8](#_Toc15832)

[4.1功能需求 8](#_Toc2670)

[4.2非功能需求 9](#_Toc11642)

[五、数据结构设计 10](#_Toc11678)

[5.1 MazeData 类的设计 10](#_Toc9802)

[5.1 迷宫数据结构 13](#_Toc29233)

[5.1.1 迷宫存储方案 13](#_Toc18727)

[5.1.2 迷宫存储的状态压缩 13](#_Toc16116)

[5.1.3 迷宫边界的动态计算 14](#_Toc11631)

[5.2 路径表示与存储 15](#_Toc29319)

[5.2.1 路径的表示 15](#_Toc12563)

[5.2.2 路径的存储 16](#_Toc19873)

[5.3 优先队列（最大堆）实现 18](#_Toc10834)

[六、算法设计 21](#_Toc3170)

[6.1 最短路径算法 21](#_Toc4156)

[6.1.1 A\* 搜索算法 21](#_Toc3904)

[6.1.2 BFS 算法 26](#_Toc6257)

[6.1.3 最短路算法对比 29](#_Toc14419)

[6.2 选择与优化 33](#_Toc31675)

[6.2.1 自适应路径搜索算法 33](#_Toc2060)

[6.2.2 Brian Kernighan 算法 35](#_Toc24679)

[七、应用程序设计 37](#_Toc7327)

[7.1 迷宫生成与显示 37](#_Toc8338)

[7.1.1 迷宫初始化 37](#_Toc4507)

[7.1.2 迷宫显示 38](#_Toc5064)

[7.2 迷宫编辑功能 39](#_Toc27028)

[7.2.1  迷宫编辑功能 39](#_Toc32420)

[7.2.2  迷宫管理功能 40](#_Toc14793)

[7.3 迷宫持久化存储功能 41](#_Toc8269)

[7.3.1 序列化与反序列化的作用 41](#_Toc20509)

[7.3.2 迷宫序列化功能 41](#_Toc21765)

[7.3.3 迷宫反序列化功能 42](#_Toc3570)

[7.3.4 迷宫文件格式 43](#_Toc15921)

[7.4 游戏逻辑与交互 43](#_Toc18069)

[7.5 路径的生成与显示 44](#_Toc9076)

[八、测试和运行 44](#_Toc9828)

[8.1 测试环境与方法 44](#_Toc7642)

[8.2 运行结果 44](#_Toc4057)

[九、总结 44](#_Toc25894)

# 一、系统开发平台

## 1.1 架构概述

本系统的架构设计采用模块化结构，主要包括以下几个核心模块，每个模块都有独立的功能，协同工作来实现迷宫游戏的整体逻辑和交互。

**主要模块及其功能**

**客户端模块：**

客户端模块负责实现游戏的用户交互界面，玩家通过 **Unity 引擎** 与游戏进行互动。玩家可以通过键盘的方向键来控制老鼠的上下左右移动，实时查看迷宫的变化。在客户端中，迷宫被呈现在 3D 空间内，玩家可以通过摄像机视角调整查看角度，以便更清晰地查看迷宫的布局。同时，客户端模块还提供了编辑迷宫的功能，玩家可以改变迷宫的墙壁和道路，创建新的迷宫布局。

**迷宫模块：**

迷宫模块的主要任务是管理迷宫的生成与存储。系统使用 **长整型数组** 搭配 **状态压缩算法** 来表示迷宫中的每一行，这种方式不仅节省了存储空间，还提高了处理速度。每个迷宫单元格的状态（墙壁或道路）通过二进制位来表示，简化了数据结构的管理。该模块还支持迷宫的动态编辑功能，玩家可以通过客户端修改迷宫中的墙壁和路径，及时更新迷宫的状态。

**路径计算模块：**

路径计算模块是系统的核心部分，负责计算老鼠从起点到终点的路径。系统实现了多种路径搜索算法，包括 **A\* 算法** 和 **广度优先搜索（BFS）算法**，采用了 **剪枝策略**，该策略根据当前迷宫的墙壁比例动态调整。A\* 算法通过启发式搜索，能够高效地找到最短路径，而 BFS 算法则能够找到所有可行的路径。

此外，路径计算模块还实现了 **深度优先搜索（DFS）算法**，用于在给定的深度限制内查找所有可能的路径。DFS 算法通过递归或栈的方式，逐步探索每一个可能的路径，直到到达终点或者路径深度超过限制为止。在探索过程中，系统会标记已经访问过的节点，避免出现循环路径。DFS 的优势在于它可以探索迷宫中所有的路径，从而提供更多的选择，适合用来分析所有可能的通行路径。

为了提高 DFS 算法的效率和避免不必要的重复计算，系统采用了路径剪枝策略。在遇到已经访问过的节点时，直接跳过当前路径，从而减少了无效路径的探索。此外，DFS 还配合 **超时参数** 使用，通过 **CancellationToken** 来限制搜索时间，避免在复杂迷宫中造成性能问题。该功能使得系统能够在合理时间内找到所有可行路径，适合在路径数量较多时使用。

DFS 算法的实现通过栈（Stack）来管理当前的路径状态，递归地遍历每个可达的节点，并将路径保存下来。一旦到达终点，当前路径将被加入到结果集中。系统允许用户设定路径深度限制和最大路径数，从而控制搜索的范围和结果数量。

该模块提供了 **获取所有路径** 的功能，玩家可以查看迷宫中从起点到终点的所有可能路径，对于需要精确路径规划的用户来说，DFS 提供了多种选择。同时，该模块还支持 **最短路径** 的计算，帮助用户在复杂迷宫中迅速找到最优路线。

**存储模块：**

存储模块提供了对迷宫数据的序列化与反序列化支持，用户可以将当前的迷宫布局和路径保存到文件中，以便后续读取。系统使用了 **Newtonsoft.Json** 库进行数据的序列化和反序列化，确保迷宫数据的完整性和一致性。通过存储模块，玩家不仅可以保存自己编辑的迷宫，还可以读取预先保存的迷宫地图，加载历史的迷宫布局。

**游戏逻辑模块：**

游戏逻辑模块负责处理游戏的初始化、计时、胜利判断等核心功能。在游戏开始时，系统会根据迷宫的初始状态创建游戏场景，并为玩家设置起点和终点。计时功能会记录玩家操控老鼠的时间，并在老鼠到达粮仓时判断是否成功。在规定的时间内，如果老鼠成功到达终点，系统会提示玩家胜利，否则提示失败。此外，该模块还管理游戏的暂停和重新开始等功能。

## 1.2 开发平台、技术栈

**开发平台**

Unity 6（6000.0.33f1）：

选择 Unity 引擎作为游戏开发平台，主要是因为其强大的图形处理能力和丰富的组件库，可以帮助快速实现游戏的用户界面、动画效果以及物理引擎。Unity 提供的跨平台支持，使得游戏可以在多种设备上运行，为用户提供更好的体验。

**编程语言**

C# 2.1：

使用 C# 语言进行编程，C# 具有良好的面向对象特性，能够帮助组织代码，提高代码可维护性。

C# 与 Unity 引擎的结合使得开发者能够高效地利用 Unity 提供的组件和功能，快速实现游戏逻辑。

## 1.3 开发环境配置

**1. Unity 编辑器安装与配置**

首先，下载并安装 Unity Hub，这是管理 Unity 编辑器版本和项目的官方工具。在 Unity Hub 中，选择所需的 Unity 编辑器版本进行安装，并在安装过程中勾选所需的构建支持模块，如 WebGL 和 Android。安装完成后，使用 Unity Hub 创建新项目或打开现有项目。

**2. Visual Studio 安装与配置**

在安装 Unity 编辑器时，建议同时安装 Visual Studio 2019 或 2022，并确保勾选“使用 Unity 的游戏开发”工作负载，以便获得与 Unity 的无缝集成。安装完成后，打开 Unity 编辑器，导航至“编辑”>“首选项”>“外部工具”，在“外部脚本编辑器”下拉菜单中选择已安装的 Visual Studio 版本。这将使 Unity 与 Visual Studio 关联，方便进行代码编写和调试。

[learn.microsoft.com](https://learn.microsoft.com/zh-cn/visualstudio/gamedev/unity/get-started/getting-started-with-visual-studio-tools-for-unity?utm_source=chatgpt.com" \t "_blank)

**3. UnityUI 的使用**

UnityUI 是 Unity 提供的官方用户界面系统，用于构建游戏的用户界面。在项目中，可以通过“GameObject”>“UI”菜单创建各种 UI 元素，如按钮、文本框和面板。通过 UnityUI，可以方便地设计和实现游戏的用户交互界面。

**4. 版本控制与调试工具**

在项目开发过程中，使用 Git 作为版本控制工具，管理项目的源代码和资源。可以通过 GitHub、GitLab 或 Bitbucket 等平台托管代码库，方便团队协作和代码管理。此外，利用 Visual Studio 的调试功能，可以在 Unity 编辑器中设置断点，进行单步调试，帮助开发者快速定位和修复问题。

## 1.4 系统特点与创新

本系统具有以下创新点：

1. **迷宫存储方式**：通过 长整型数组 来表示迷宫的每一行，采用二进制位、状态压缩算法来存储墙壁和道路信息，这种存储方式不仅节省了内存，还提高了搜索和更新效率。
2. **路径搜索算法的优化**：系统实现了 A\* 算法 和 BFS 算法，采用了剪枝策略，该策略根据当前迷宫的墙壁比例、迷宫的复杂度动态选择最优路径搜索算法，能够处理不同难度的迷宫。
3. **最大堆优先队列**：为了优化 A\* 算法的性能，系统使用了 最大堆 实现优先队列，这使得在处理大规模迷宫时，A\* 算法的效率大大提高。
4. **迷宫编辑功能**：玩家可以在游戏过程中实时编辑迷宫，改变迷宫的布局。这一功能增加了游戏的互动性和可玩性。
5. **数据持久化**：系统支持迷宫地图的序列化与存储，玩家不仅可以保存和加载迷宫，还可以保存路径信息，以便后续查看和复用。
6. **开源化与社区贡献：**本项目在 GitHub 平台上开源，任何开发者都可以访问代码库、查看源码，并进行二次开发或贡献改进建议。通过开源模式，项目可以吸引更多开发者的参与，不断优化游戏性能和功能，同时促进算法与数据结构的学习和交流。

这些创新不仅提升了游戏的玩法体验，还展示了不同算法和数据结构在实际应用中的优势。

# 问题描述

## 2.1 任务陈述

本项目的目标是实现一个迷宫游戏，用户通过控制一个老鼠角色，在迷宫中寻找从起点到终点的路径。游戏的背景设定为老鼠在迷宫中寻找粮仓，玩家通过键盘上的方向键来控制老鼠的运动。迷宫中有墙壁和可通行的路径，玩家需要避开墙壁，成功到达终点才能完成任务。

此外，本系统支持迷宫编辑功能，允许玩家动态修改当前迷宫的结构，例如将墙壁变为通路，或是将通路变为墙壁。系统还实现了路径搜索功能，可以计算出老鼠从起点到终点的所有可能路径，并特别标识出最短路径。

系统需要支持迷宫地图的序列化与存储，使得用户可以将当前迷宫保存为文件，并在未来的游戏中继续加载和编辑该地图。

## 2.2 任务目标

本迷宫游戏系统的功能需求包括以下几个方面：

1. **操控老鼠的运动：** 游戏允许用户通过键盘上的方向键（上下左右）来操控老鼠在迷宫中移动。老鼠的运动是实时响应的，且只能在没有墙壁的区域内移动，不能穿越墙壁。老鼠的初始位置设定为迷宫的中心，终点为迷宫的右下角。
2. **检测成功与失败：** 游戏需要实时检测老鼠是否到达终点。如果老鼠在规定的时间内成功到达粮仓（终点），则提示“成功”，否则提示“失败”。此外，若老鼠碰到墙壁或无法继续前进，也应给出相应的提示。
3. **编辑迷宫功能：** 游戏支持迷宫的实时编辑，用户可以通过修改迷宫中的墙壁与通路，动态改变迷宫的结构。每个格子的状态可以在墙壁和通路之间切换，用户可以通过鼠标点击或其他方式来编辑迷宫地图。此功能提供更高的灵活性，使得玩家能够自定义游戏环境。
4. **计算所有路径与最短路径：** 系统需要实现路径计算功能，支持计算从起点到终点的所有可行路径，并找出其中的最短路径。路径计算采用多种搜索算法（如 A\* 算法、BFS 算法等），根据迷宫的结构自动选择最优的算法。最短路径和所有路径的显示能够帮助玩家更好地理解迷宫的布局和走向。
5. **迷宫地图序列化与存储：** 系统需要支持迷宫地图的序列化存储功能。迷宫和路径的数据将保存到文件中，以便后续读取和加载。这使得用户可以保存自己编辑的迷宫并随时继续编辑或重新开始游戏。地图的存储与加载功能将保证游戏的持久性，避免玩家丢失当前进度。

# 系统定义

## 3.1 系统输入

系统输入主要包括两大类数据：迷宫数据和用户的操作指令。迷宫数据由系统初始化时加载，或由用户通过编辑功能手动修改。这些数据包括迷宫的结构、墙壁的分布以及起点与终点的位置等信息。迷宫的生成与编辑基于二维网格表示，墙壁与通道分别用不同的数值进行标记（例如，1 代表墙壁，0 代表通道）。在编辑迷宫时，用户可以自由选择路径，墙变路，路变墙，实时更新迷宫的状态。

玩家的操作指令包括键盘上的方向键输入，通过这些输入，玩家控制老鼠在迷宫中移动。每次输入方向键时，系统会根据用户的指令更新老鼠的位置，并进行相应的碰撞检测，确保老鼠不能穿越墙壁。如果用户在规定的时间内成功到达粮仓，则系统判定为胜利，否则提示失败。

为了提高系统的可扩展性和跨平台兼容性，系统采用了 **虚拟轴和虚拟按键** 技术，为未来将游戏部署至多个平台（如 Windows、MacOS、Web、移动设备等）奠定了基础。通过这一技术，系统能够适配不同设备的输入与显示特性，提供一致的用户体验。此设计使得系统在未来支持更多的平台时，能够无缝迁移，并减少额外的开发成本。

## 3.2 用户视图

系统输出主要呈现为图形化的用户界面，其中最重要的部分是迷宫地图的实时显示、老鼠的运动轨迹、路径计算结果以及游戏成功或失败的提示。

1. **迷宫地图：**显示当前的迷宫布局，包括墙壁、路径、起点、终点等元素。用户可以在界面上查看整个迷宫结构，并在编辑模式下修改地图。
2. **老鼠的运动轨迹：**通过动画形式展示老鼠在迷宫中的移动，每当用户按下方向键时，老鼠会沿着预定路径移动，路径上的每一步都会在界面上更新。
3. **路径计算结果：**路径计算模块会实时计算并展示从起点到终点的路径，用户可以查看最短路径或所有可行路径。
4. **成功或失败提示：**在玩家控制老鼠完成迷宫任务后，系统会根据玩家是否在规定时间内成功到达粮仓提供相应的提示。如果老鼠到达终点，显示“成功”；如果时间耗尽或老鼠走错路线，显示“失败”。
5. **视角切换与聚焦功能：**为了提高用户的沉浸感和操作体验，系统支持视角切换功能，玩家可以自由切换视角，查看迷宫的不同部分。玩家能够选择局部视角，聚焦于老鼠的当前位置，以便精确控制老鼠的移动。此外，玩家还可以切换至全局视角，查看整个迷宫的布局，包括老鼠的当前位置、路径计算结果以及周围的环境。这种视角切换功能让玩家能够更好地进行策略规划和决策，尤其在迷宫较为复杂时，提供了更大的操作灵活性。

**3.3 系统组件**

系统由以下几个核心组件构成：

1. **迷宫数据存储与处理模块：**负责迷宫数据的加载、编辑与存储。通过长整型数组表示迷宫的每一行，同时提供存储和加载功能，支持用户在不同设备间分享和加载自定义迷宫。
2. **用户输入与界面交互模块：**负责接收用户的操作输入（如方向键），并将这些输入转化为老鼠的移动指令。同时，处理用户在编辑模式下对迷宫的修改，实时更新迷宫的状态。该模块提供了与用户的互动界面，确保用户体验的流畅性。
3. **路径计算模块：**实现了多种路径搜索算法（如 A\*、BFS、DFS），用于计算老鼠从起点到终点的最短路径或所有路径。该模块对路径计算进行了优化，采用动态剪枝策略，能够根据迷宫的特性选择最合适的算法，提高计算效率

# 四、需求分析

## 4.1功能需求

1. **老鼠移动的控制与检测：**  
   玩家需要通过键盘方向键控制老鼠在迷宫中的移动，支持上下左右四个方向。每次老鼠移动后，系统应实时检测老鼠是否碰到墙壁或是否到达终点。如果老鼠碰到墙壁，则阻止移动，并提示“撞墙了”；如果到达终点，显示“成功到达粮仓”。
2. **迷宫编辑功能：**  
   用户可以通过简单的界面交互来编辑迷宫，支持添加墙壁、清除墙壁、设置起点和终点等功能。编辑功能应确保操作直观且反馈及时，玩家可以随时修改迷宫布局。
3. **路径计算与搜索功能：**  
   系统提供路径计算功能，包括最短路径和所有可行路径的计算。采用 A\* 算法或 BFS 等路径搜索算法，根据迷宫的布局实时计算老鼠从起点到终点的路径。路径计算模块应支持动态更新路径结果，以便玩家查看不同路径的选择。
4. **结果的判定与提示：**  
   游戏的胜负条件包括成功到达粮仓和失败（例如超时）。系统应实时判断玩家的操作结果，在玩家完成任务时给出相应的提示：若老鼠成功到达粮仓，则显示“胜利”；如果时间用尽或老鼠走错路径，则显示“失败”。此外，路径计算模块的提示也需要反馈给用户。
5. **迷宫文件的序列化存储与读取：**  
   游戏应具备保存和读取迷宫的功能。用户可以保存当前编辑的迷宫结构，保存格式支持JSON或其他常见格式。用户在重新加载游戏时，可以读取之前保存的迷宫数据，继续进行游戏。存储功能应保证数据完整性和读取的高效性。

## 4.2非功能需求

1. **性能要求：**  
   游戏应具备较高的响应速度和流畅度，尤其是在复杂迷宫的路径计算时。路径搜索算法（如 A\*）应在一定的时间限制内完成计算，避免因算法性能问题导致游戏卡顿或响应延迟。系统应具备超时处理机制，当计算时间超过一定限制时，提示用户路径计算超时。
2. **可扩展性和可维护性：**  
   系统应具备良好的可扩展性和可维护性，以便后期添加新的功能或优化现有功能。例如，可以方便地添加新的路径搜索算法，或者支持多种不同的迷宫布局。代码结构应遵循良好的编程规范，模块化设计，减少模块之间的耦合，使得后期的修改和更新能够更加便捷。同时，应有良好的文档支持，以方便开发人员理解和维护代码。

# 五、数据结构设计

**5.1 MazeData 类的设计**

MazeData 类是迷宫的核心数据结构，其设计围绕**空间效率**和**快速访问**展开。以包含了多个关键的变量和方法。下面是对其中主要变量和方法的分类介绍：

**一、主要变量**

1. name  
   类型：string  
   用途：迷宫的名称，用于标识该迷宫。
2. size  
   类型：int  
   用途：迷宫的尺寸，表示迷宫的行列数。假设迷宫是一个方形迷宫，size 表示一个边的长度。
3. fileName  
   类型：string  
   用途：保存迷宫的文件名，用于存储或加载迷宫数据。
4. grid  
   类型：long[]  
   用途：迷宫的布局，以每行的二进制表示存储，long 数字的二进制位表示一个迷宫的每一行。
5. \_shortestPath  
   类型：byte[]  
   用途：存储最短路径的二进制数据。该数组通过 shortestPath 属性转换成路径。
6. shortestPath  
   类型：List<(int, int)> **计算属性**（Computed Property）  
   用途：最短路径的坐标列表（使用 (int, int) 元组表示）。它们并不在内存中直接存储数据，通过 getter 和 setter 将 \_shortestPath 与路径之间进行转换。
7. start  
   类型：(int, int) **计算属性**（Computed Property）  
   用途：迷宫的起点坐标，默认情况下是迷宫的中心（size / 2 - 1, size / 2），并不在内存中直接存储数据，。
8. end  
   类型：(int, int) **计算属性**（Computed Property）  
   用途：迷宫的终点坐标，默认情况下是迷宫的左上角（size - 1, 0），并不在内存中直接存储数据，。
9. directions  
   类型：(int, int)[]  
   用途：用于存储迷宫中可移动的方向，包含四个方向（上、下、左、右）。
10. **主要方法**
11. Initialize(int size)  
    用途：初始化迷宫数据，包括设置 size 和初始化 grid 数组，并计算最短路径。
12. GetCell(int x, int z)  
    用途：检查指定位置 (x, z) 是否是通路（true 表示通路，false 表示障碍物）。
13. SetCell(int x, int z, bool value)  
    用途：设置指定位置 (x, z) 的值为 true 或 false，更新迷宫的格子状态。
14. IsInBounds((int, int) pos)  
    用途：检查指定坐标 (x, z) 是否在迷宫的有效范围内。
15. IsStratOrEnd((int, int) pos)  
    用途：检查指定坐标 (x, z) 是否为迷宫的起点或终点。
16. FindShortestPath()  
    用途：计算迷宫的最短路径。如果迷宫的障碍物比例大于 50%，使用 BFS（广度优先搜索）；否则，使用 A\*（A Star）算法。
17. GetObstacleRatioMoreThan(double ratio)  
    用途：计算迷宫中障碍物的比例，判断障碍物比例是否超过指定的 ratio，用于决定使用 BFS 还是 A\*。
18. BFS()  
    用途：使用广度优先搜索算法计算从起点到终点的最短路径，并返回路径坐标列表。如果无法找到路径，则返回空列表。
19. AStar()  
    用途：使用 A\* 搜索算法计算从起点到终点的最短路径，并返回路径坐标列表。该方法使用曼哈顿距离作为启发式函数。
20. ReconstructPath(Dictionary<(int, int), (int, int)> cameFrom)  
    用途：通过回溯父节点的方式构造路径，返回从起点到终点的路径。
21. CreateVisitedArray()  
    用途：创建一个 long[] 数组，用于存储迷宫中的访问标记，每个 long 数字表示 64 个位置的访问状态。
22. IsVisited(long[] visited, int x, int z)  
    用途：检查指定坐标 (x, z) 是否已经访问过。
23. MarkVisited(long[] visited, int x, int z)  
    用途：标记指定坐标 (x, z) 为已访问。
24. GetAllPaths(int maxWidth, int maxDepth, TimeSpan timeout)  
    用途：使用 DFS 算法获取迷宫中的所有路径，并考虑最大宽度、最大深度和超时限制。
25. TupleSecondItemComparer  
    用途：A\* 算法中用来比较元组的类，按第二项（路径的代价）排序，优先队列用于排序。
26. ManhattanDistance((int, int) a, (int, int) b)  
    用途：计算两个坐标之间的曼哈顿距离，作为 A\* 算法中的启发式函数。

## 5.1 迷宫数据结构

5.1.1 迷宫存储方案

在本项目中，迷宫存储结构中每个元素代表迷宫中的一个格子。 每个格子有两种可能的状态：墙壁或通路。这种存储方式直观且易于理解，方便进行路径搜索和迷宫编辑等操作。

5.1.2 迷宫存储的状态压缩

采用数组来显示存储墙和路的排列

1. **传统方案的缺陷：**

在传统方案中，使用二维布尔数组 bool[,] 来存储迷宫状态会存在空间占用较大和内存访问效率低的缺陷。例如，在一个 100x100 的迷宫中，每个布尔值占用 1 字节，因此需要 100 \* 100 = 10,000 字节（即 10 KB）来存储迷宫的状态。当需要进行序列化时，数据量大且处理效率较低。

1. **位压缩方案：**

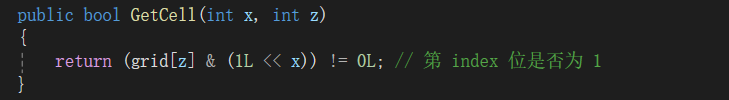
在优化方案中，采用 long[] 数组来表示每行迷宫的状态。通过位运算，long 类型的 64 位可以存储 64 个坐标点的状态。假设每个格子的状态是 true 或 false，那么每个 long 就能表示一整行的状态。

例如，当 size=64 时，使用 64 个长整型数字即可表示整个迷宫的状态。该方案在存储时显著减少了内存占用，并且提高了访问效率：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| | 方案 | | --- | | 数据结构 | 序列化数据量 | 总空间占用 |
| 传统方案 | bool[] | 每个格子 5 字节 | 20480 字节 (20 KB) |
| 压缩方案 | long[] | 每个 long 约 10 字节（十进制） | 640 字节 |

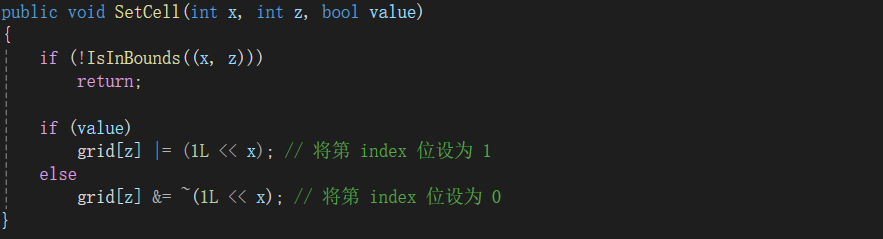
通过位压缩，压缩方案的内存占用仅为传统方案的 约 3%，且不需要额外的内存分配来存储每个布尔值。

1. **核心方法实现：**
2. **动态修改状态：**



通过位运算动态修改指定坐标的状态，实现 O(1) 时间复杂度的随机访问，这比传统数组的 O(N) 随机访问更高效。

1. **通过位掩码判断坐标状态：**



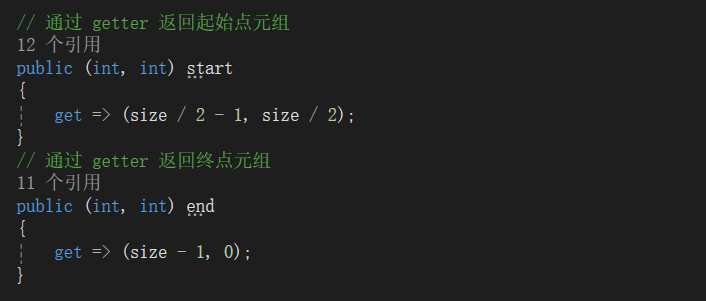
使用位掩码快速判断一个指定位置的状态，减少了传统方法中逐个元素遍历的开销。

**四、优势：**

1. **节省空间**：通过压缩方式，空间占用大大减少，特别是在大规模迷宫中表现尤为明显。
2. **访问效率**：通过位运算，获取和修改迷宫状态的效率得到了提升，能够快速定位指定坐标的状态。

5.1.3 迷宫边界的动态计算

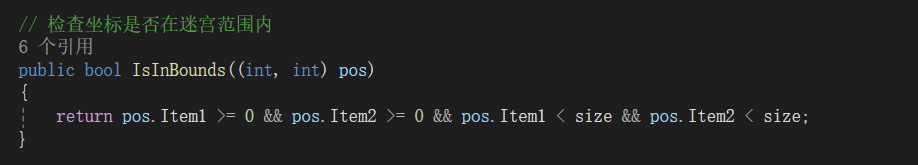
1. **起点和终点的动态生成：**



迷宫的起点和终点是通过计算属性（Computed Property）动态生成的。这确保了在迷宫尺寸发生变化时，起点和终点的位置能够自适应调整，而不需要修改硬编码的常量。

具体实现中，start 和 end 属性根据 size 动态计算，避免硬编码，保证了不同迷宫尺寸下的灵活配置。例如，size=32/64/128 时，迷宫的起点和终点位置会根据新的尺寸进行调整。

1. **动态判断迷宫边界：**



迷宫的边界检查被包装在 IsInBounds 方法中，确保在迷宫操作过程中，所有坐标都在有效的范围内，避免越界访问。

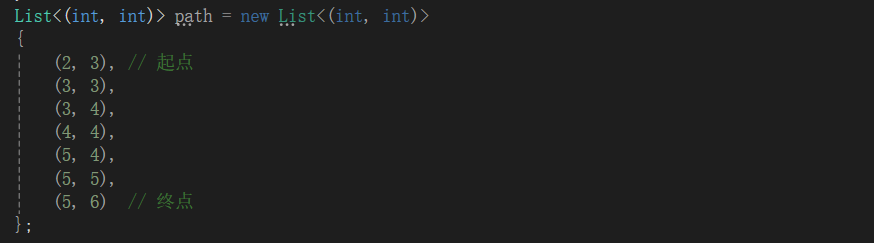
## 5.2 路径表示与存储

5.2.1 路径的表示

在本系统中，路径采用 List<(int, int)> 类型表示，其中每个元素是一个元组 (x, y)，表示路径上的一个坐标点。 路径的第一个元素为起点，最后一个元素为终点，中间的元素按顺序连接，构成从起点到终点的完整路径。

假设我们有一个迷宫，起点坐标为 (2, 3)，终点坐标为 (5, 6)，路径经过以下坐标点：

在代码中，可以使用 List<(int, int)> 来表示这条路径：

****

这种表示方式直观且易于操作，方便在路径搜索和显示过程中进行处理。 List<(int, int)> 是 C# 中的泛型集合类型，提供了动态大小和丰富的方法，如添加、删除、排序等，适用于存储和管理路径点集合。 使用元组 (int, int) 作为元素类型，使得每个路径点的坐标可以直接存储，避免了额外的数据结构，简化了代码的复杂度。 此外，List 类型的动态特性使得在路径搜索过程中，可以方便地添加或删除路径点，满足不同算法的需求。

5.2.2 路径的存储

由于最短路径存在多处调用的情况，故而对迷宫最短路进行存储。

为了提高迷宫路径存储的效率，采用二进制格式对路径数据进行紧凑且高效的序列化。

**一、序列化设计目标**

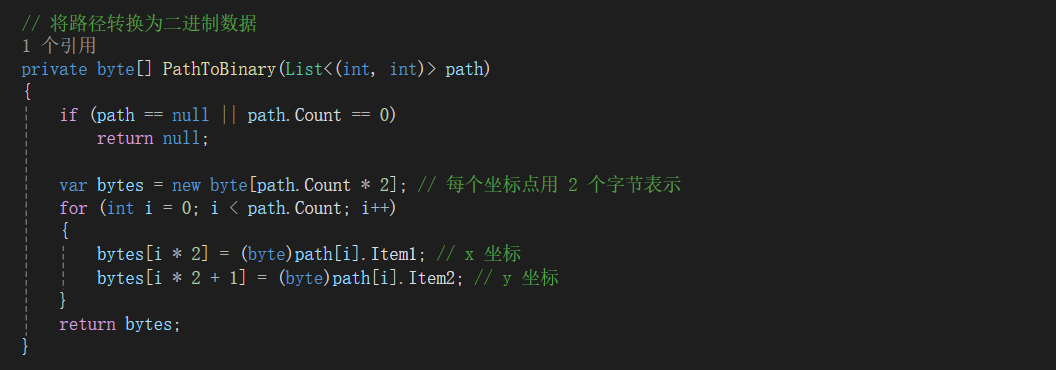
1. **紧凑性：**每个路径点的坐标 (x, y) 的值范围为 [0, size-1]，对于 size ≤ 256 的情况，可以用 byte 类型（1 字节）来存储每个坐标点的 x 和 y 坐标。因此，每个路径点占用 2 字节（1 字节存储 x 坐标，1 字节存储 y 坐标）。
2. **兼容性：**序列化后的数据以 byte[] 数组的形式保存，可以直接用于文件存储或网络传输，而无需依赖复杂的格式如 JSON 或 XML。这样不仅简化了数据处理，也提高了存储和传输的效率。

**二、序列化实现细节**

路径序列化的过程就是将每个路径点的坐标 (x, y) 转换为相应的字节流。

以下是具体步骤：

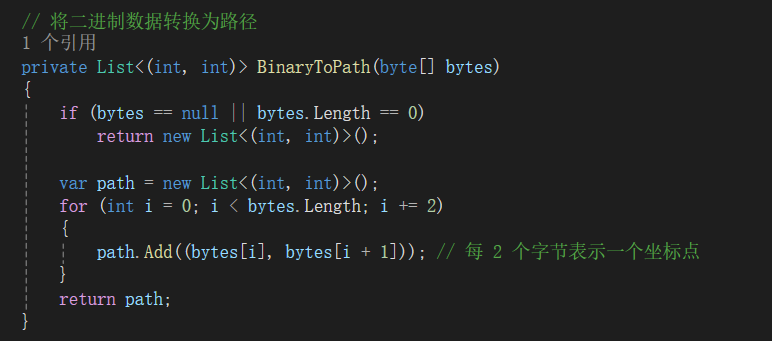
对于路径 [(2, 3), (4, 5)]，将每个坐标点拆解为两个字节：x = 2 对应 0x02，y = 3 对应 0x03，依此类推，路径将序列化为 02030405。



通过这个序列化过程，路径的存储不仅紧凑，而且可以直接保存成二进制数据，方便进一步处理。

1. **反序列化的健壮性**

反序列化时，我们将字节流按照步长 2 进行遍历，每两个字节解析为一个路径点的 x 和 y 坐标。这样确保路径点的坐标完整性。



此方法可以确保反序列化时正确读取每个路径点，并恢复出原始的路径数据。

**四、存储效率对比**

以下是不同存储格式的效率对比：

|  |  |  |
| --- | --- | --- |
| 存储格式 | 单点占用空间 | 1000长度路径的总空间 |
| JSON（文本格式） | ~20 字节 | 20KB |
| 二进制序列化 | 2 字节 | 2KB |

**压缩率**：二进制序列化格式相比 JSON 格式，节省了约 90% 的存储空间。这是因为 JSON 格式需要为每个路径点提供额外的元数据，如逗号、括号等字符，而二进制格式只存储最必要的数值数据。

## 5.3 优先队列（最大堆）实现

最大堆是一种完全二叉树结构，在每个节点上存储的值都不小于其子节点的值。也就是说，最大堆的根节点存储的是所有节点中的最大值。此性质使得最大堆特别适用于需要频繁获取最大值的场景，例如优先队列的实现。

在 A\* 搜索等路径规划算法中，优先队列通常用于动态选择当前最优节点，最大堆通过高效的插入和删除操作来维护优先级顺序。实现最大堆时，通常使用数组来存储堆元素，通过索引计算父节点和子节点的位置。

**一、最大堆的基本特性**

最大堆具有以下几个关键特性：

1. 完全二叉树：堆是一棵完全二叉树，除了最后一层外，所有的节点都有两个子节点，最后一层节点的子节点尽可能向左排列。
2. 堆性质：对于每个节点 i，其父节点 i/2（除非是根节点）存储的值总是大于等于子节点的值。根节点是最大值节点。
3. 二叉堆数组表示：堆通常通过数组表示，给定一个节点的索引 i，其子节点索引分别为 2i + 1 和 2i + 2，父节点的索引为 (i - 1) / 2。

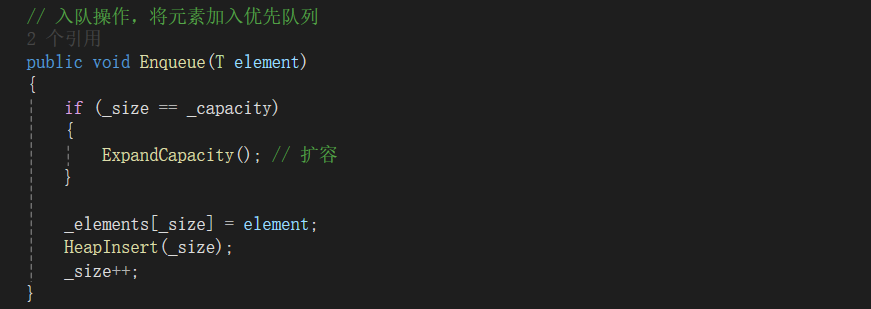
**二、最大堆的实现**

1. **定义堆类**



PriorityQueue 类定义了堆的核心属性，包括堆的大小、容量、元素数组和比较器。Top 是堆顶元素的访问器，用于获取最大值（或最小值，根据比较器的定义）

1. **入队操作 (Enqueue)**



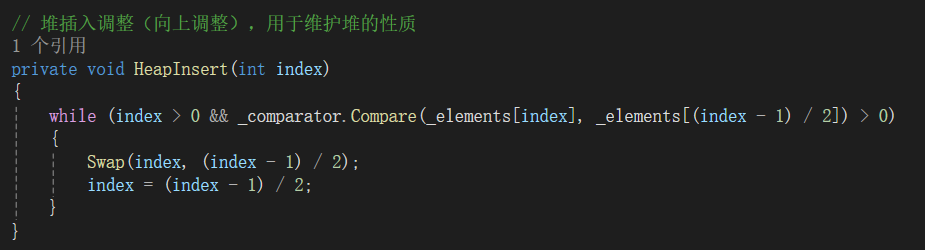
当元素入队时，首先检查堆是否已满，若已满则扩容。将新元素插入堆的末尾，并调用 HeapInsert 方法确保堆结构正确（即堆顶元素为最大值）。

1. **出队操作 (Dequeue)**

****

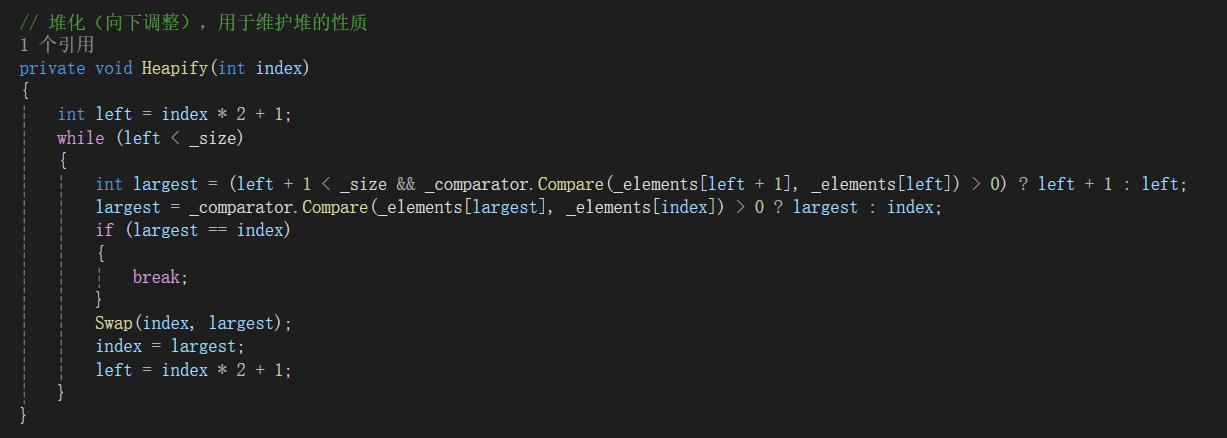
在出队操作中，首先将堆顶元素与堆的最后一个元素交换，然后减小堆的大小。接着调用 Heapify 调整堆，使得堆重新保持最大堆性质。

1. **堆插入 (HeapInsert)**

****

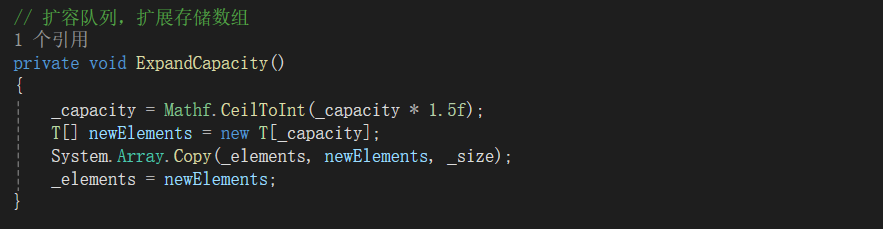
HeapInsert 用于将新元素插入堆后，并保持堆的最大堆性质。通过不断比较当前元素和父节点的值，若当前元素更大，则交换并继续向上调整。

1. **堆化操作 (Heapify)**



Heapify 用于删除堆顶元素后的堆结构恢复。删除堆顶后，堆的最后一个元素被移到堆顶，通过比较子节点值，若其较大，则交换并继续调整堆。

1. **扩容操作 (ExpandCapacity)**



如果堆的元素数量超出当前容量，则进行扩容。扩容时容量按 1.5 倍增加，然后将原来的元素复制到新的更大数组中。

# 六、算法设计

## 6.1 最短路径算法

### 6.1.1 A\* 搜索算法

**一、A\* 算法的基本原理**

A\*（A-Star）算法是一种启发式搜索算法，结合了广度优先搜索（BFS）的完备性和贪心最佳优先搜索（Greedy Best-First Search）的高效性。其核心思想是通过动态评估每个节点的综合代价，优先扩展最有希望接近目标的节点。具体来说，A\* 通过以下两个函数指导搜索过程：

1. **实际代价函数（gScore）：**从起点到当前节点的实际路径长度。
2. **启发式函数（hScore）：**当前节点到终点的预估代价（需满足可采纳性，即不高估真实代价）。

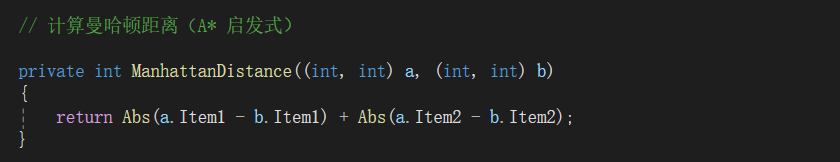
算法的总代价计算公式为：

通过优先扩展 fScore 最小的节点，A\* 在保证找到最短路径的前提下，显著减少搜索空间。

1. **启发式函数的设计**

在迷宫路径搜索中，**曼哈顿距离（Manhattan Distance）**是常用的启发式函数，其定义为两点在网格坐标系中沿轴对齐方向的水平和垂直距离之和：

**代码实现**



**选择依据**

1. **可采纳性保证：**曼哈顿距离始终不大于实际路径长度（避免错过最优解）。
2. **网格环境适配性：**迷宫为规则网格结构，仅允许四方向移动（上、下、左、右），与曼哈顿距离的计算逻辑完全匹配。
3. **计算高效性：**仅需两次绝对值和一次加法，时间复杂度为 O(1)。

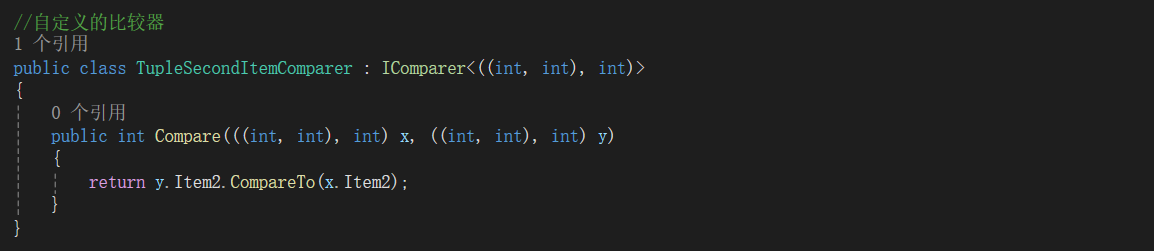
**三、算法实现细节**

1. **数据结构设计**

* **优先队列（PriorityQueue）**：  
  按 fScore 从小到大排序，确保每次扩展代价最小的节点。



* + 自定义比较器：TupleSecondItemComparer 根据 fScore 排序。



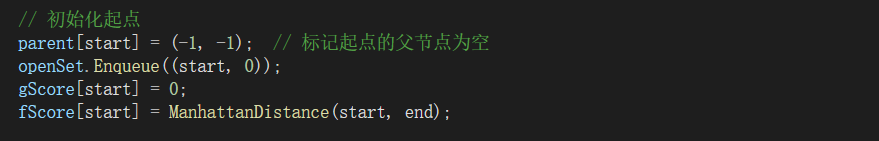
* + 容量优化：初始容量设为，避免频繁扩容。
* **字典与集合**：
  + parent：记录每个节点的父节点，用于路径回溯（键为当前节点，值为父节点）。
  + gScore 和 fScore：分别存储实际代价和综合代价。
  + done：标记已找到最短路径的节点，避免重复处理。

1. **算法流程**

* **初始化阶段：**

将起点加入 openSet，设置其 gScore=0，fScore 为起点到终点的曼哈顿距离。

标记起点的父节点为 (-1, -1)（特殊值表示无父节点）。

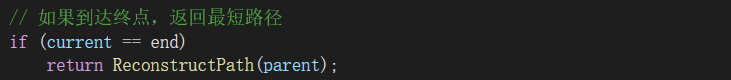


* **主循环阶段：**

**节点出队：**从优先队列中取出 fScore 最小的节点 current。



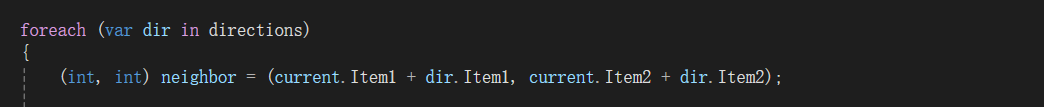
**终点检测：**若 current 为终点，调用 ReconstructPath 返回路径。



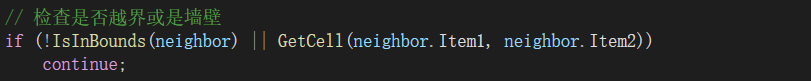
**剪枝优化：**若 current 已在 done 集合中，跳过后续处理（避免重复计算）。



**邻居扩展：**遍历四个方向生成相邻节点 neighbor。



**有效性检查：**跳过越界节点和障碍节点。



**代价计算：**计算 neighbor 的 tentativeGScore（当前路径的实际代价）。



**路径更新：**若新路径更优，更新 parent、gScore 和 fScore，并将 neighbor 加入队列。



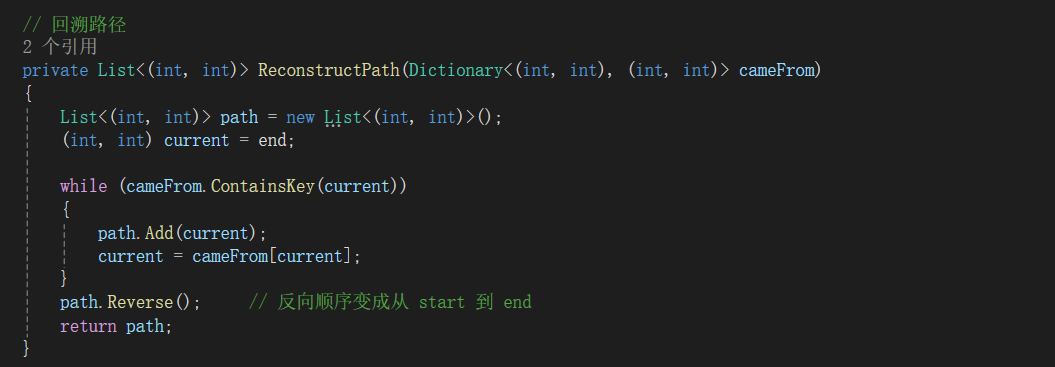
* **关键优化点**

**done 集合的剪枝作用：**  
节点一旦出队，说明已找到其最短路径，后续再次遇到该节点时直接跳过。此设计基于 A\* 的可采纳性保证，避免重复扩展。

**启发式引导的优先级：**  
通过 fScore 的排序，算法始终优先扩展靠近终点的节点，减少无效搜索。

**四、路径重建与优化**

**路径回溯实现**

****

**步骤解析：**

1. 反向回溯：从终点 end 开始，通过 parent 字典逐节点回溯至起点 start。
2. 路径反转：回溯生成的路径顺序为 end → start，需反转以得到 start → end 的正确顺序。
3. 时间复杂度：O(L)，L 为路径长度，效率远高于递归实现。

**优化意义**

1. 空间效率：仅需存储父节点关系，无需记录完整路径，内存占用从 O(N²) 降为 O(N)。
2. 健壮性保障：通过字典键存在性检查（cameFrom.ContainsKey(current)），避免无效访问。
3. **复杂度分析**

**1.时间复杂度分析**

**（1）理论最坏情况：**

A\* 算法的时间复杂度取决于启发式函数的质量和迷宫结构。

优先队列的插入/删除操作（基于最小堆实现）的时间复杂度为 O(log K)，其中 K 为队列长度。最坏情况下 K=O(N²)，因此总时间为 O(N² log N)。

在最坏情况下（如启发式函数完全无效，例如 ），A\* 退化为 Dijkstra 算法，时间复杂度为 ，其中：

N²：迷宫的总节点数（假设迷宫为 N×N 的网格）。

log N：优先队列（最小堆）的插入和删除操作的时间复杂度。

**（2）实际优化场景：**

使用曼哈顿距离等可采纳启发式函数时，A\* 的时间复杂度接近 O(bᵈ)，其中：

b：平均分支因子（四方向迷宫中 b=4）。

d：起点到终点的最短路径长度。

在低障碍密度迷宫中，A\* 的启发式引导能显著减少扩展的节点数，实际时间复杂度可降至 O(N log N)。

**（3）代码实现中的关键因素：**

**优先队列操作：**每次插入和删除节点的时间复杂度为 O(log K)，其中 K 为队列中当前节点数。

**剪枝优化：**done 集合确保每个节点最多被处理一次，避免重复计算。

**2.空间复杂度分析**

**（1）数据结构占用：**

**优先队列：**存储待扩展节点，最坏情况下需存储所有可达节点，空间复杂度为 O(N²)。

**字典与集合：**

gScore 和 fScore 字典：各需 O(N²) 空间。

parent 字典和 done 集合：各需 O(N²) 空间。

**总计：**空间复杂度为 O(N²)。

**（2）实际优化效果：**

在稀疏迷宫（低障碍密度）中，优先队列和字典的实际空间占用远小于理论最坏值。

通过 done 集合剪枝，避免无效节点的重复存储。

### 6.1.2 BFS 算法

**一、BFS 算法的原理**

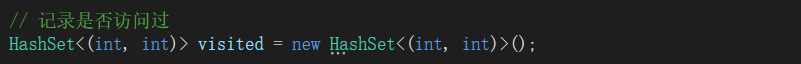
广度优先搜索（Breadth-First Search, BFS）是一种基于层序扩展的图遍历算法，其核心思想是**按距离起点由近及远的顺序逐层探索节点**，确保最先找到的路径即为最短路径。在迷宫问题中，BFS 的适用性体现在以下方面：

1. **最短路径保证：**由于逐层扩展特性，BFS 能严格保证找到的路径是迷宫中的最短路径。
2. **无启发式依赖：**不依赖任何估值函数，适用于复杂或未知环境。
3. **时间复杂度稳定：**无论迷宫结构如何，时间复杂度始终为 O(N²)（N 为迷宫尺寸）。
4. **BFS 的实现**
5. **数据结构设计**

* **队列（Queue）：**  
  用于管理待扩展的节点，遵循“先进先出”（FIFO）原则，确保层序扩展。



* **哈希表（HashSet）：**  
  记录已访问节点，避免重复处理。



* **父节点字典（Dictionary）：**存储每个节点的父节点坐标，用于路径回溯。

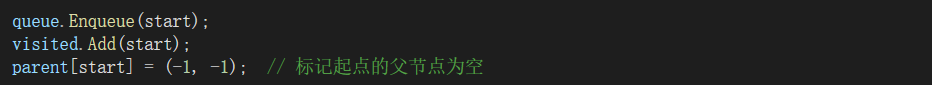


1. **算法流程**

* **初始化阶段：**

将起点加入队列，并标记为已访问。

设置起点的父节点为 (-1, -1)（表示路径终点）。



* **主循环阶段：**

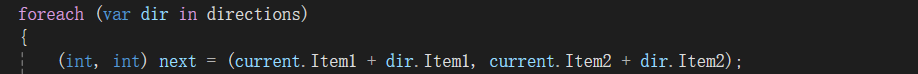
**节点出队：**从队列头部取出当前节点 current。



**终点检测：**若 current 为终点，调用 ReconstructPath 返回路径。



**邻居扩展：**遍历四个方向生成相邻节点 next。



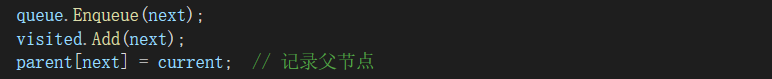
**有效性检查：**

节点是否在迷宫范围内（IsInBounds(next)）。

节点是否为通路（!GetCell(next.Item1, next.Item2)）。

节点是否未被访问（!visited.Contains(next)）。

**入队与记录：**若 next 有效，则加入队列并记录父节点。



* **终止条件：**

若队列为空且未到达终点，返回空路径（迷宫无解）。

****

1. **复杂度分析**

**1.时间复杂度分析**

**（1）理论复杂度：**

BFS 的时间复杂度为 O(N²)，其中 N 为迷宫尺寸。

**推导依据：**

每个节点最多被访问一次（通过 visited 集合保证）。

每个节点的邻居扩展次数固定（四方向迷宫中每个节点最多生成 4 个邻居）。

**（2）实际性能：**

在高障碍密度迷宫中，实际访问的节点数可能远小于 N²（例如障碍占比 70% 时，复杂度降为 O(0.3N²)）。

在完全连通的空旷迷宫中，需遍历所有节点，时间复杂度严格为 O(N²)。

**2.空间复杂度分析**

**（1）数据结构占用：**

**队列：**最坏情况下存储所有节点，空间复杂度为 O(N²)。

**哈希表与字典：**

visited 集合：存储所有已访问节点，空间复杂度 O(N²)。

parent 字典：存储每个节点的父节点，空间复杂度 O(N²)。

**总计：**空间复杂度为 O(N²)。

**（2）实际优化效果：**

在稀疏迷宫中，队列和哈希表的实际空间占用较低。

通过提前终止（到达终点时立即返回）减少队列的最大长度。

### 6.1.3 最短路算法对比

**一、A \*与 BFS 的复杂度对比**

|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | 时间复杂度（最坏） | 空间复杂度（最坏） | 关键影响因素 |
| A\* | O(N² log N) | O(N²) | 启发式函数质量、障碍密度 |
| BFS | O(N²) | O(N²) | 迷宫连通性、障碍密度 |

**二、A \*与 BFS 的特殊场景分析**

在实际应用中，A\* 和 BFS 的表现因迷宫结构和障碍分布的不同而存在显著差异。以下通过具体场景对比两者的优缺点，并结合代码实现说明其适用性。

1. **普通障碍密度迷宫**

**场景描述：**

假设迷宫障碍物的密度小于 50%，虽然迷宫中有许多障碍物，但大部分区域仍然是可通行的，起点和终点之间存在至少一条通路。在这种情况下，障碍物的分布虽然使得某些区域变得狭窄，但并不会导致大部分位置无法到达。

**示例迷宫：**



**A\* 的表现：**

* **启发式：**A\* 使用了启发式函数（通常是曼哈顿距离或欧几里得距离）来估计每个节点到终点的最短距离。这使得 A\* 能够优先扩展离终点更近的节点，而不是盲目地扩展所有可能的路径。
* **搜索策略：**由于 A\* 会优先扩展距离目标最近的节点，在障碍物密度较高的迷宫中，A\* 会显著减少无效搜索路径，避免遍历许多无关的区域。通过启发式的引导，A\* 不会像 BFS 那样在所有方向上扩展，可能会减少遍历的节点数量。
* **扩展节点：**在这种迷宫场景中，A\* 会根据启发式函数的引导较早找到通向目标的路径，从而扩展的节点数较少，通常大约在 20-30 个节点之间。这个数字可能会因为具体的障碍物布局而变化，但总体上，A\* 能够更快地找到目标，避免了无意义的路径探索。

**BFS 的表现：**

* **广度优先搜索：**BFS 在没有启发式的情况下，逐层遍历每一层的所有节点，扩展每个可行路径的所有方向。由于它没有任何启发式信息来引导搜索，因此它会盲目地探索整个迷宫。
* **搜索策略：**BFS 会从起点 (0, 0) 开始，逐层扩展所有可能的路径，直到它找到终点。尽管 BFS 保证找到最短路径，但它可能会扩展大量的节点，即使某些节点完全不相关。尤其是当迷宫中有大量障碍物时，BFS 可能会花费大量的时间来检查每个节点，甚至需要遍历一些已经被障碍物隔断的区域。
* **扩展节点：**在障碍密度较高的迷宫中，BFS 可能会扩展 60-80 个节点（甚至更多），因为它会遍历所有可能的路径。每一层都会展开所有可能的方向，直到找到目标。

**复杂度对比：**

示例迷宫中障碍占比约为 29.6%（24/81），属于低障碍密度迷宫。

* **BFS 的耗时分析**

需绕行第 5 行的连续障碍。

扩展节点数：约 40-50 个节点。

BFS 需要逐层扩展，绕过迷宫中的局部障碍（如第 5 行的连续障碍 1 1 1 1 1）。

实际耗时：约 8-12ms（假设每节点处理耗时 0.2ms）。

* **A\* 的耗时分析**

沿右侧快速逼近终点。

扩展节点数：约 20-30 个节点。

曼哈顿距离的启发式引导能有效绕过障碍，直接向终点方向搜索。

实际耗时：约 4-6ms（优先队列维护和启发式计算增加单节点耗时，但总扩展数少）。

**2. 高障碍密度迷宫（障碍占比 > 60%）**

**场景描述：**  
迷宫内存在大量随机障碍，形成多个“死胡同”或“隔离区域”，但起点和终点仍存在一条连通路径。

**示例迷宫：**



**A \*的表现：**

* 问题：启发式函数（曼哈顿距离）难以有效引导搜索方向，优先队列频繁插入低优先级节点（例如绕行死胡同的路径）。
* 退化现象：A\* 退化为类似 Dijkstra 算法，需维护大量无效节点的 fScore，时间复杂度接近 O(N² log N)。
* 代码表现：openSet 队列长度激增，done 集合无法有效剪枝，导致内存占用和计算时间显著增加。

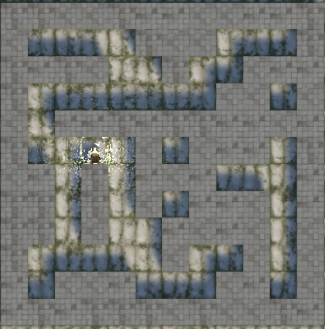
**BFS 的表现：**

* 优势：层序扩展无需启发式函数，直接覆盖所有可行区域，避免无效优先级计算。
* 效率：时间复杂度稳定为 O(N²)，实际耗时可能低于 A\*（例如障碍占比 70% 时，BFS 耗时比 A\* 减少 50%）。
* 代码表现：queue 按固定顺序扩展节点，visited 集合直接过滤无效访问。

1. **无通路迷宫**

场景描述：  
迷宫内起点和终点不存在一条连通路径。

示例迷宫：



**A \*的表现：**

启发式函数（曼哈顿距离）引导搜索方向完全无效，优先搜索最靠近目标的点，但如果所有点都无路可通，它仍然会继续取最优的点尝试，直到遍历完所有可达点，得路径不存在的结论，然而优先队列会有额外的优先级调整开销。

**BFS 的表现：**

最终遍历该连通分支内的所有路径，得路径不存在的结论

**复杂度对比：**

示例迷宫中，BFS 访问 30 个点，总时间 O(30) = 30 \* O(1) = 30。

A\* 访问 30 个点，总时间 ，，所以 A\* 大约是 30 \* 5 = 150，比 BFS 慢了 5 倍！

随着迷宫规模的增大，带来的复杂度差距只会成级别的差距

## 6.2 选择与优化

### 6.2.1 自适应路径搜索算法

在路径规划问题中，A\* 算法通常由于其启发式搜索的特性，能够在大多数情况下显著优于广度优先搜索（BFS），特别是在具有较少障碍物且路径较为清晰的环境中。A\* 的优势在于它能够结合启发式函数（如曼哈顿距离）来引导搜索过程，从而避开一些不必要的搜索区域，减少了计算开销。然而，当面对具有大量障碍物或者极为复杂的迷宫环境时，A\* 的性能可能会受到影响，导致它的效率低于 BFS，甚至在某些情况下，A\* 可能会出现比 BFS 更慢的情况。

**一、A\* 算法的潜在瓶颈**

1. **障碍物密集环境下的启发式失效：**在一个障碍物较多的环境中，启发式函数的引导可能会失效，A\* 算法无法有效缩小搜索范围，导致它与 BFS 相比的优势大打折扣。在这种情况下，A\* 可能会继续在不必要的区域进行搜索，从而增加了计算的复杂度。
2. **优先队列开销：**A\* 使用优先队列（如堆）来管理待扩展的节点。在每次扩展节点时，优先队列需要进行 push 和 pop 操作，每次操作的时间复杂度为 O(logN)，这对于大规模问题或者障碍物密集的场景来说，可能会导致额外的开销，尤其是在节点扩展数量相当大的情况下。与此相比，BFS 仅仅依赖队列的先进先出（FIFO）机制，虽然它没有启发式的引导，但是其队列操作的时间复杂度仅为 O(1)，因此在某些情形（通路不存在，必须连通分支内的所有路径）下，BFS 的效率可能高于 A\*。

**二、动态选择算法的优化策略**

为了克服上述问题，可以根据迷宫的障碍物密度动态选择合适的搜索算法，以达到更高的搜索效率。具体地，考虑到障碍物密度对算法性能的影响，可以设定一个墙壁比例阈值（ratio），根据这一比例来决定是使用 A\* 还是 BFS 算法：

1. **高障碍物密度环境（BFS优先）：**当迷宫的墙壁比例大于某一预设阈值（如 50%）时，迷宫中的可行区域较少，路径规划的复杂度增加。在这种情况下，A\* 算法的启发式函数可能无法有效引导搜索，优先队列的开销也会显著增加。此时，使用 BFS 更为合适，因为它不依赖于启发式函数，能够保证遍历所有可能的路径，直到找到目标点，且时间复杂度较低。BFS 的简单队列结构在这种密集障碍环境下比 A\* 的优先队列更为高效。
2. **低障碍物密度环境（A\*优先）：**当迷宫的墙壁比例小于该阈值时，障碍物较少，路径搜索空间较大，A\* 的启发式函数能够更好地引导搜索过程，避免无关区域的扩展。因此，A\* 算法在这种空旷的环境中能够显著减少搜索的时间和空间开销，找到最短路径的效率更高。

**三、应用场景和算法选择**

考虑到当迷宫的障碍物密度大于50%时，用户在实际应用中很难判断起点和终点之间是否连通，因此可以将这一阈值（50%）作为划分标准，动态选择合适的算法。具体而言：

* 当迷宫的障碍物密度大于50%时，采用 BFS 算法，以避免 A\* 的启发式函数失效带来的额外开销。
* 当迷宫的障碍物密度小于50%时，采用 A\* 算法，以利用其启发式引导减少搜索空间，提高路径规划效率。

通过动态调整算法选择，不仅能提高算法的适应性，还能在不同环境下实现路径搜索的优化，从而有效提升系统的性能和用户体验。

### 6.2.2 Brian Kernighan 算法

在迷宫搜索问题中，墙壁的分布密度直接影响路径搜索算法的效率。尤其是当迷宫中墙壁的比例较大时，传统的启发式搜索（如 A\*）的效率可能受到影响，反而广度优先搜索（BFS）可能会更高效。因此，在动态选择算法时，我们需要一个高效的方式来计算迷宫中墙壁的比例，以便在合适的情况下切换算法。

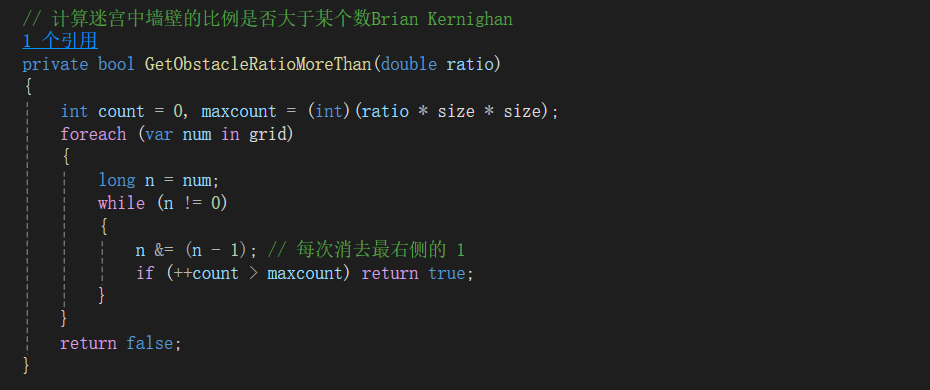
为了优化这一过程，采用了 Brian Kernighan 算法 来计算墙壁的数量比例。该算法通过对每个迷宫网格单元进行位操作，快速统计墙壁的数量。

1. **Brian Kernighan 算法的原理**

Brian Kernighan 算法是一种优化的 位计数 方法。对于每个数，它使用位操作将最右边的 1 置为 0。每进行一次操作，计数器加 1，直到数值变为 0。该算法的时间复杂度为 O(k)，其中 k 是二进制表示中 1 的数量。相较于传统的逐位检查所有比特位，Brian Kernighan 算法可以显著减少操作次数，特别是在大量 0 的情况下。

具体而言，我们在本方法中使用该算法来统计迷宫中墙壁的数量（墙壁表示为 1），并与给定的比例进行比较。如果墙壁的数量超过了预设的比例阈值（如 50%），则返回 true，表示应使用 BFS 进行路径搜索；否则，返回 false，表示可以使用 A\* 算法。

1. **代码实现**

****

1. **性能分析**

Brian Kernighan 算法通过逐步消去最右侧的 1，避免了检查每一位的操作，因此它的时间复杂度是 O(k)，其中 k 是二进制表示中 1 的个数。相比传统的逐位统计方法，该算法能大大提高处理速度，尤其是在迷宫较大、墙壁分布稀疏的情况下。

由于每次处理一个网格单元时的操作数仅与该单元的二进制表示中 1 的数量相关，因此在统计过程中，该方法比直接遍历每个比特位更加高效。

## 6.3 所有路径的求解

# 七、应用程序设计

## 7.1 迷宫生成与显示

7.1.1 迷宫初始化

迷宫的初始化主要在 MapManager 类的 Start 方法中完成，具体步骤以及关键代码如下：

1. **加载迷宫数据：**

通过 fileManager.LoadAllMazes(mazeList) 加载所有迷宫数据到到  maze- List 列表中，并设置当前迷宫索引 currentMaze 为加载的默认迷宫。

1. **生成地图基础结构：**

**地板生成：**调用 GenerateFloor 方法，在网格范围内随机放置地板预制体。

**边界生成：**调用 GenerateBorder 方法，在迷宫外围放置边界方块。

**迷宫方块生成：**调用 GenerateBlock 方法，根据迷宫数据初始化每个方块的状态（激活或非激活），并缓存入blockDictionary的字典中，降低后续调用开销，同时避免重复生成。

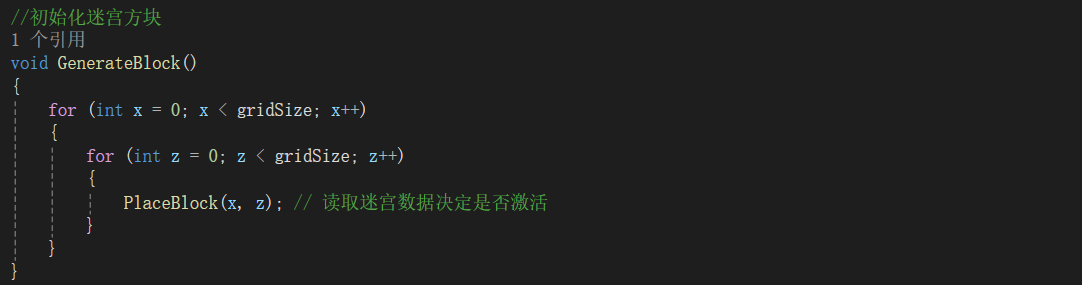
1. **加载迷宫：**

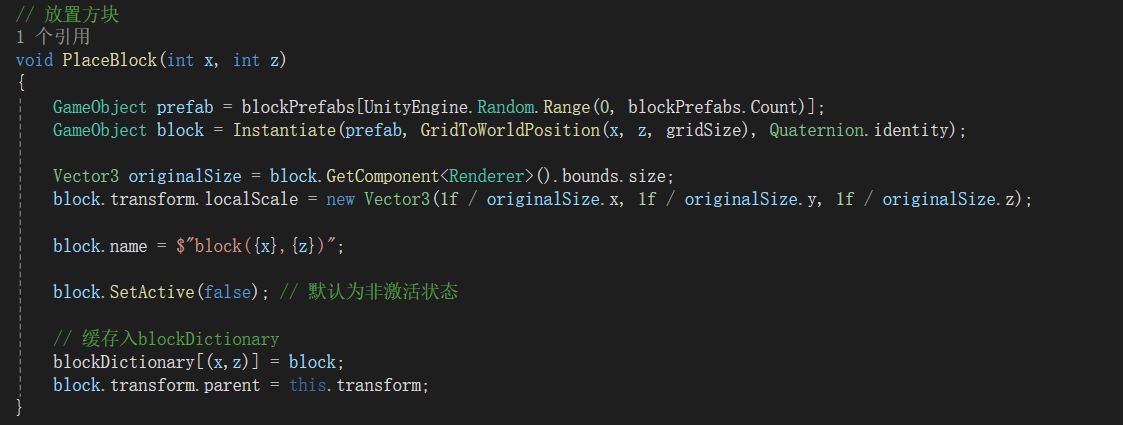
调用 LoadMaze 方法，将当前迷宫数据应用到场景中，激活或隐藏对应的方块。

1. **添加终点标志：**

调用 AddCheese 方法，在迷宫的终点位置放置一个标志物（奶酪）。

**关键代码**

****

****

7.1.2 迷宫显示

迷宫的显示通过 LoadMaze 方法实现，具体逻辑如下：

1. **读取当前迷宫数据：**

获取当前迷宫的 MazeData 对象，并复制其网格数据到临时数组 tempGrid。

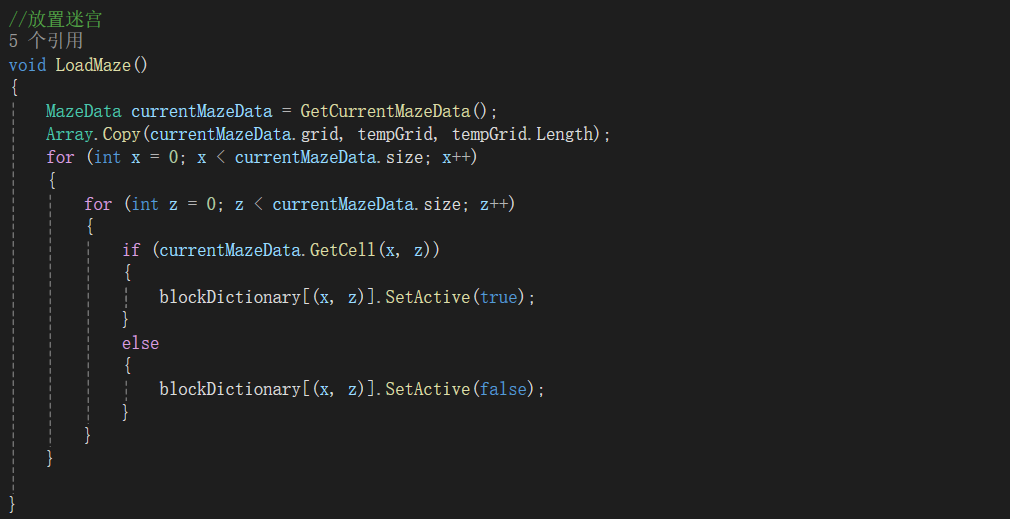
1. **更新方块状态：**

遍历迷宫网格的每个坐标 (x, z)，根据 MazeData.GetCell(x, z) 的值决定是否激活对应的方块。并通过 blockDictionary 字典快速查找并设置方块的激活状态。

1. **场景更新：**

激活的方块显示为障碍物，非激活的方块隐藏，形成迷宫的可见结构。

**关键代码**



## 7.2 迷宫编辑功能

7.2.1  迷宫编辑功能

迷宫的编辑功能通过 changeBlock 方法实现，支持用户通过鼠标点击修改迷宫结构：

1. **鼠标点击检测：**

在 Update 方法中监听鼠标左键点击事件（Input.GetMouseButtonDown(0)）。

获取鼠标点击的世界坐标，并转换为网格坐标 (x, z)。

1. **边界与特殊点检查：**

检查坐标是否在迷宫范围内（IsInBounds）。

检查坐标是否为起点或终点（IsStratOrEnd），避免修改关键点。

1. **切换方块状态：**

调用 SetTempGridCell 方法，修改临时网格 tempGrid 中对应坐标的状态。

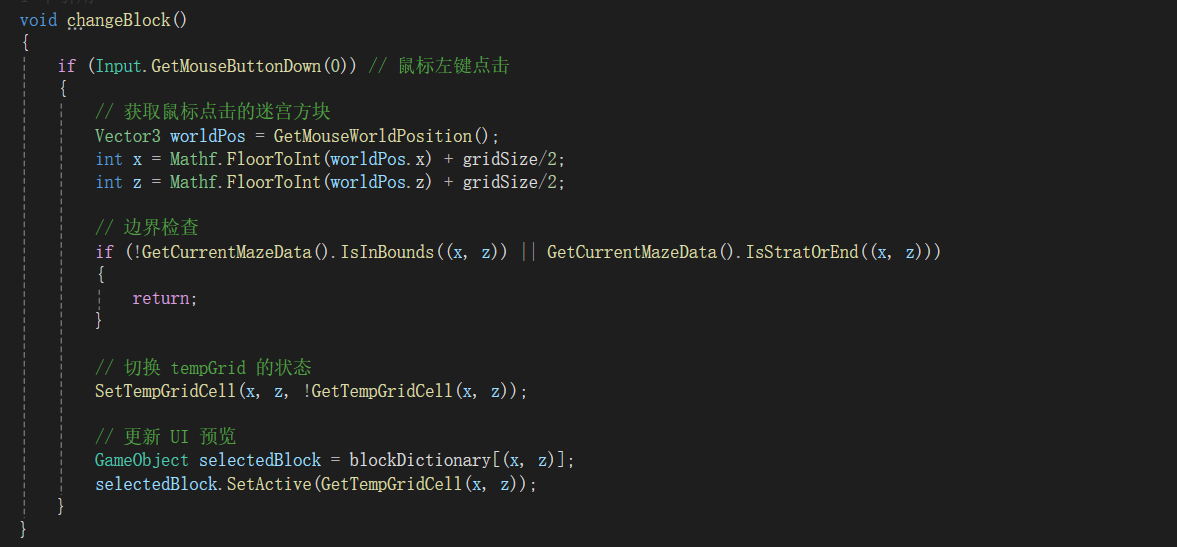
更新场景中方块的激活状态，实时反馈修改结果。

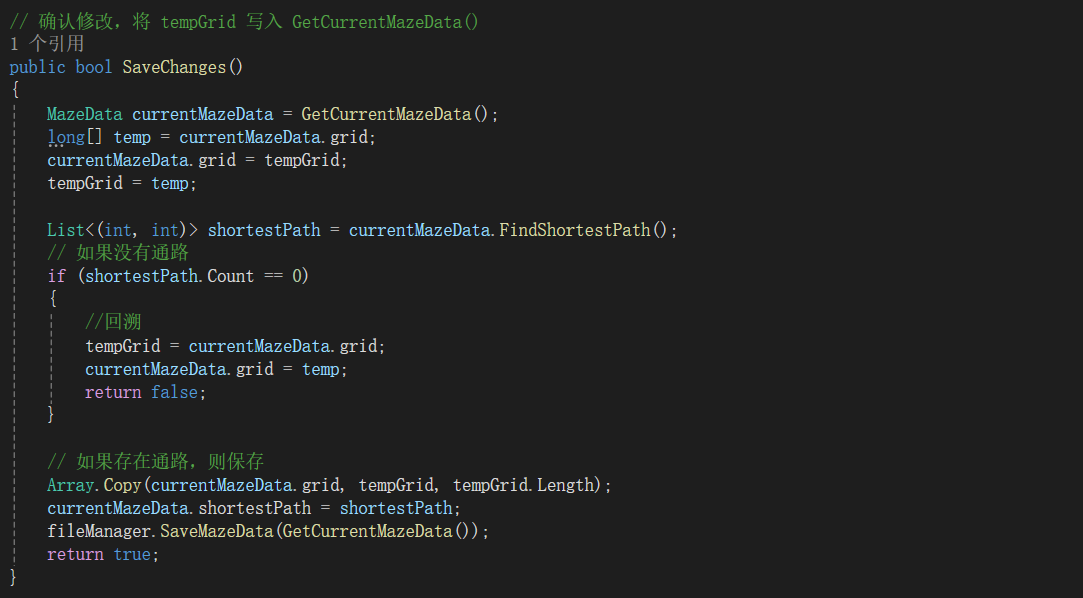
1. **保存修改：**

调用 SaveChanges 方法，将临时网格数据写回当前迷宫的 grid 属性。

检查迷宫是否连通（调用 FindShortestPath），若存在通路则保存数据，否则回滚修改。

**关键代码**

****



7.2.2  迷宫管理功能

迷宫管理功能通过 RefreshMazeManagePanel 方法实现，支持用户切换、创建和删除迷宫：

1. **刷新迷宫列表：**

清空下拉菜单选项，重新加载 mazeList 中的所有迷宫名称。

设置当前选中迷宫为 currentMaze。

1. **创建新迷宫：**

调用 fileManager.CreateNewMaze 方法，生成一个新的迷宫数据并添加到 mazeList。

刷新界面并加载新迷宫。

1. **删除当前迷宫：**

调用 fileManager.DeleteMazeData 方法，删除当前迷宫的存储文件。

重新加载迷宫列表并刷新界面。

**关键代码**



## 7.3 迷宫持久化存储功能

7.3.1 序列化与反序列化的作用

序列化是将对象转换为可存储或传输的格式（如 JSON、二进制），反序列化则是将存储的数据重新转换为对象。在迷宫管理系统中，序列化用于：

1. **持久化存储：**将迷宫数据保存到文件中，以便下次加载。
2. **数据传输：**将迷宫数据通过网络传输，支持多人协作或云端存储。
3. **版本控制：**通过文件格式的标准化，支持不同版本迷宫的兼容性。

7.3.2 迷宫序列化功能

在 FileManager 类中，迷宫的序列化通过 JsonUtility 实现，具体步骤如下：

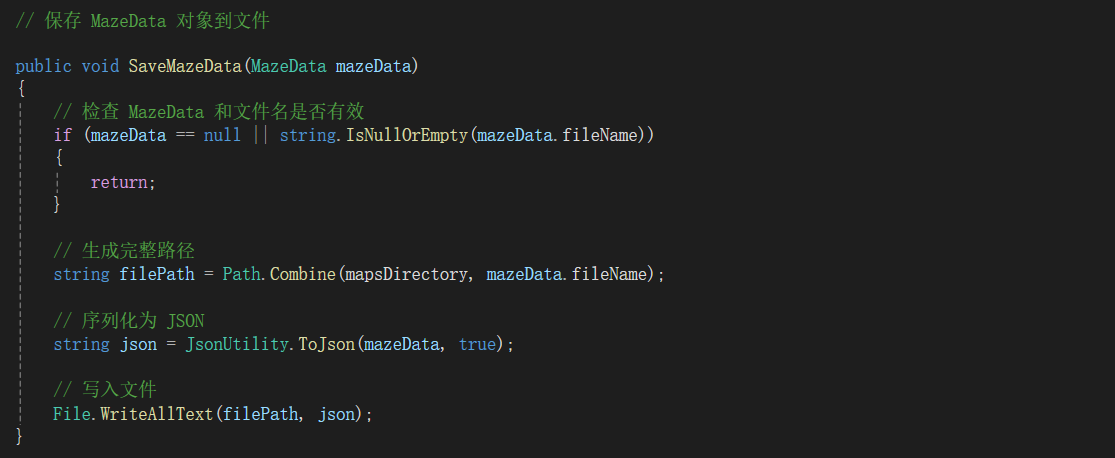
1. **将 MazeData 对象转换为 JSON 字符串：**

使用 JsonUtility.ToJson 方法将 MazeData 对象序列化为 JSON 格式的字符串。

1. **将 JSON 字符串写入文件：**

使用 File.WriteAllText 方法将 JSON 字符串保存到指定路径的文件中。

**关键代码**



7.3.3 迷宫反序列化功能

在 FileManager 类中，迷宫的反序列化通过 JsonUtility 实现，具体步骤如下：

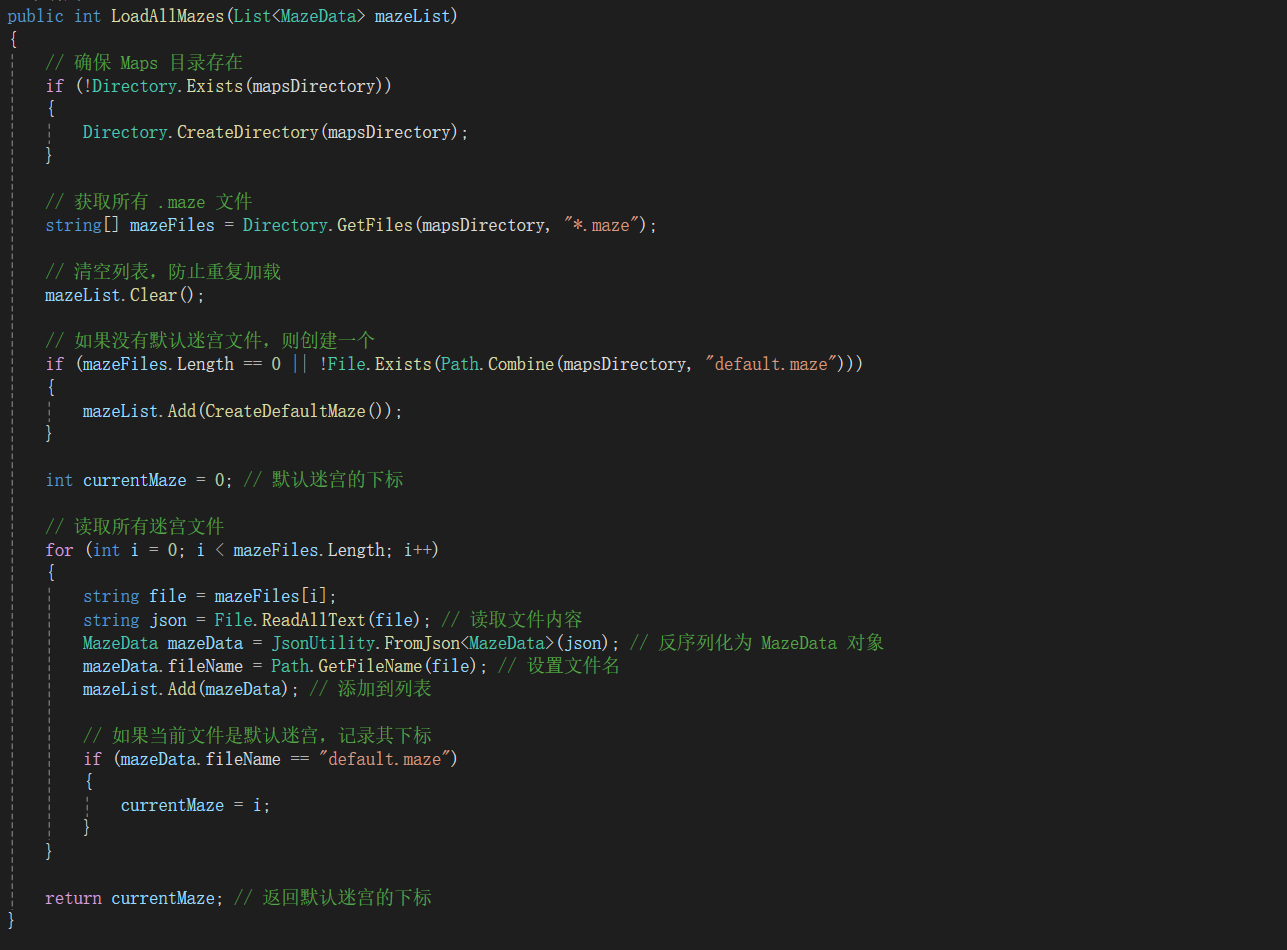
1. **从文件中读取 JSON 字符串：**

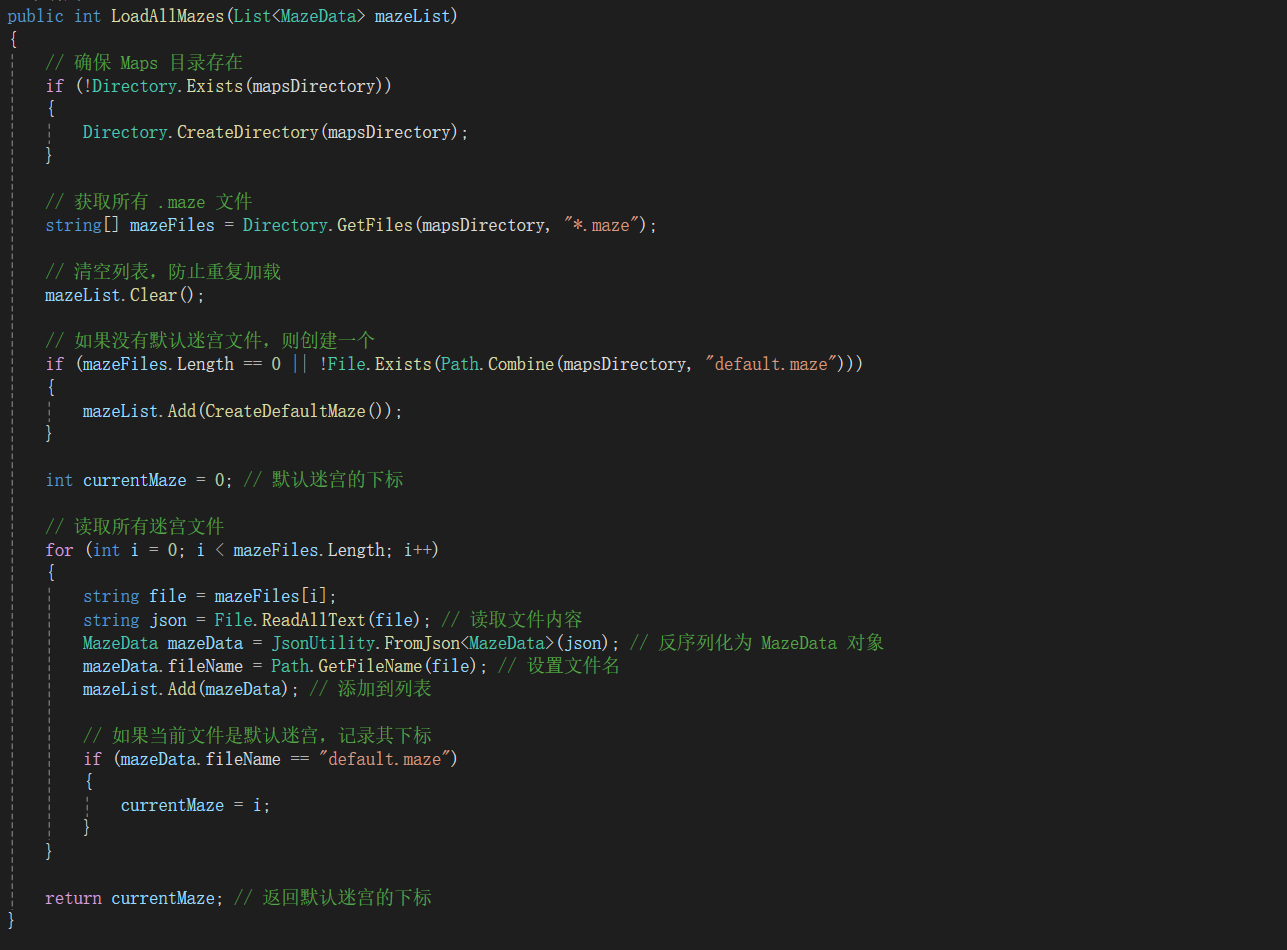
使用 File.ReadAllText 方法读取迷宫文件的内容。

1. **将 JSON 字符串转换为 MazeData 对象：**

使用 JsonUtility.FromJson 方法将 JSON 字符串反序列化为 MazeData 对象。

**关键代码**



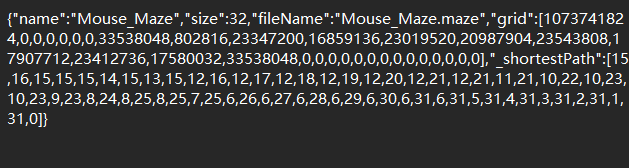


7.3.4 迷宫文件格式

迷宫文件以 .maze 为扩展名，内容为 JSON 格式，包含以下字段：

1. **name：**迷宫的显示名称。
2. **size：**迷宫的尺寸（如 16 表示 16x16 的网格）。
3. **grid：**迷宫的网格数据，使用 long[] 数组存储，每个 long 表示一行网格的二进制状态。
4. **fileName：**迷宫文件的名称（如 default.maze）。
5. **\_shortestPath：**迷宫的最短路径，使用 byte[] 存储路径点的坐标。

**示例文件内容**



## 7.4 游戏逻辑与交互

7.4.1不同功能的状态的判断和选择：通过枚举类进行控制

游戏的状态管理通过 GameState 枚举类和 GameManager 类实现，支持多种游戏模式的切换与交互。以下是各状态的功能描述与实现细节：

**1. 游戏状态枚举（GameState）**



Default：游戏初始状态，显示主界面。

Play：游戏进行中状态，玩家可以控制角色移动。

Pause：游戏暂停状态，暂停所有游戏逻辑。

Win：游戏胜利状态，显示胜利界面。

Manage：管理迷宫状态，支持迷宫的选择、创建和删除。

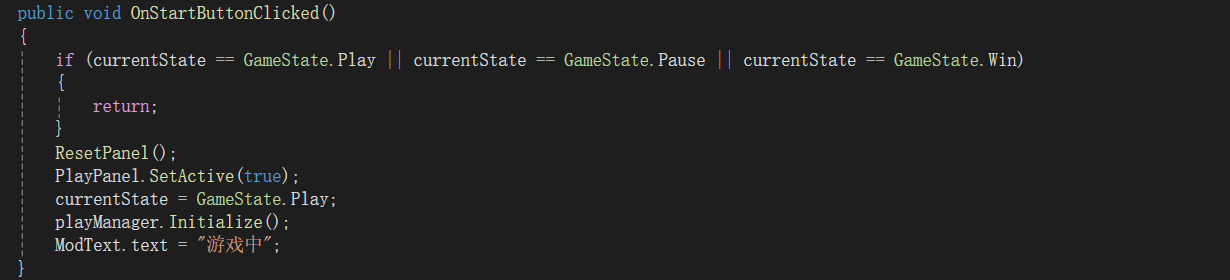
Edit：编辑迷宫状态，支持修改迷宫结构。

Path：查看路径状态，显示迷宫的最短路径或所有路径。

**2. 游戏状态切换**

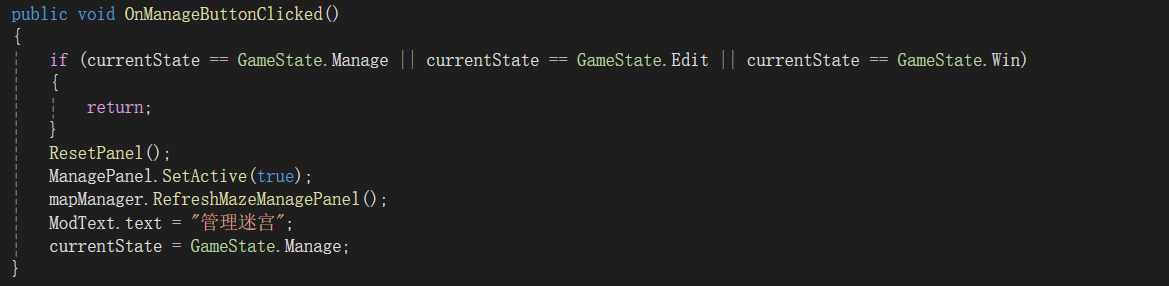
在 GameManager 类中，通过 currentState 属性记录当前状态，并根据用户输入或事件切换状态。以下是关键状态切换逻辑：

**2.1 开始游戏**



功能：切换到游戏进行中状态，初始化游戏逻辑并显示游戏界面。

**2.2 管理迷宫**



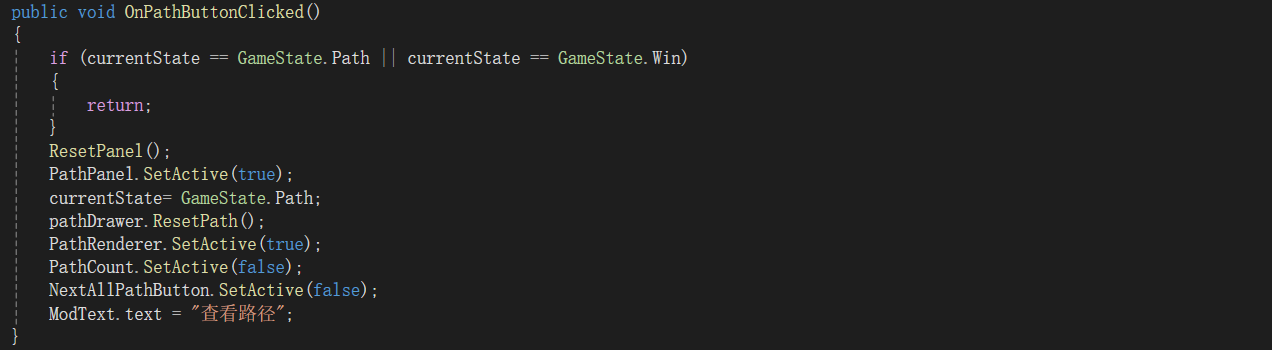
功能：切换到管理迷宫状态，显示迷宫管理界面并刷新迷宫列表。

**2.3 编辑迷宫**



功能：切换到编辑迷宫状态，显示编辑界面并允许用户修改迷宫结构。

**2.4 查看路径**



**3. 界面重置与退出**

**5.1 重置界面**



功能：隐藏所有界面并重置迷宫显示。

**5.2 退出游戏**



功能：退出游戏应用程序。

7.4.2玩家操作与老鼠的移动逻辑

在该代码中，玩家可以通过按键（WASD）来控制老鼠的移动方向。当玩家按下某个方向键时，系统会计算老鼠的新位置，确保该位置在迷宫内且可通行，然后使用协程来平滑移动老鼠。

**按键控制：**玩家通过按下 W, A, S, D 键来决定老鼠的移动方向：

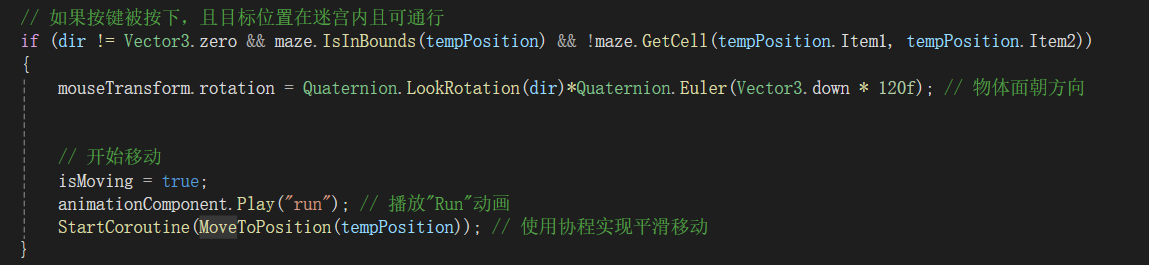
W：上 S：下 A：左 D：右

**移动逻辑：**

1. 通过输入的方向控制老鼠的移动方向。
2. 使用 Quaternion.LookRotation() 使老鼠面朝前方，确保移动的动画正确。
3. 使用协程 MoveToPosition() 来平滑地移动老鼠到目标位置。
4. 每次移动结束后，停止“run”动画并播放“wait”动画。

**防止重复移动：**通过 isMoving 变量来确保每次只允许一次移动。

**关键代码：**

****

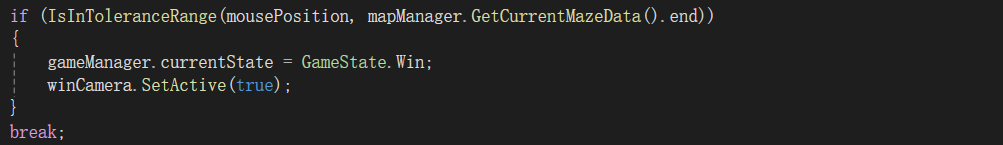


7.4.3成功与失败判定

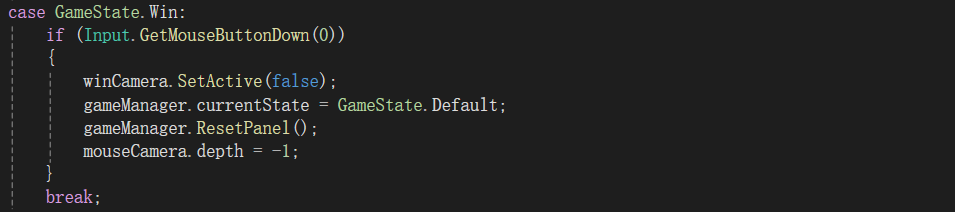
在该代码中，成功与失败的判定通过 IsInToleranceRange() 方法和 timer 来实现。

1. **成功判定：**

游戏判断是否达到了终点：当老鼠的当前位置接近终点时，游戏状态会更改为 GameState.Win，并展示胜利画面。通过 IsInToleranceRange() 判断当前位置是否与终点在容忍范围内。

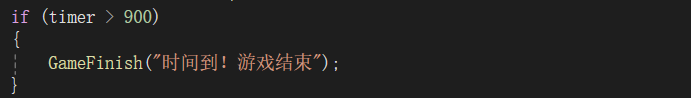


游戏状态被设为 Win 时，玩家点击鼠标左键（Input.GetMouseButtonDown(0)）后，切换到默认状态并隐藏胜利摄像头。



1. **失败判定：**

游戏会根据计时器判断是否超时。如果超过 900 秒（15 分钟），则判定游戏结束，并显示“时间到，游戏结束”的消息。



1. **结束游戏：**

在成功或失败后，点击相应按钮时会调用 GameFinish() 来结束游戏并显示提示消息。

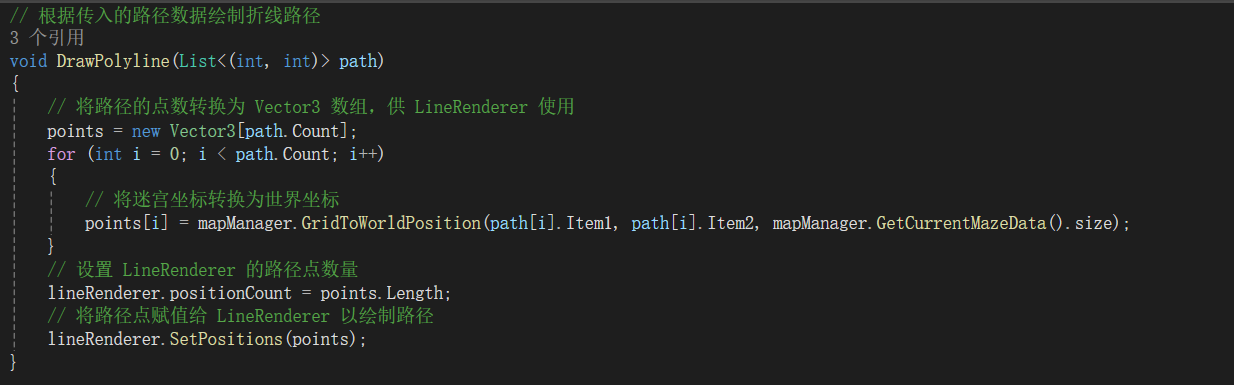


## 7.5 路径的生成与展示

路径的生成与展示通过 PathDrawer 类来实现。这个类利用 LineRenderer 来绘制迷宫中的路径，支持显示最短路径和所有路径。

7.5.1 路径的绘制

DrawPolyline() 方法接受一个路径（由一系列坐标点组成）并将这些点转换为 Vector3 数组，然后通过 LineRenderer 绘制路径。

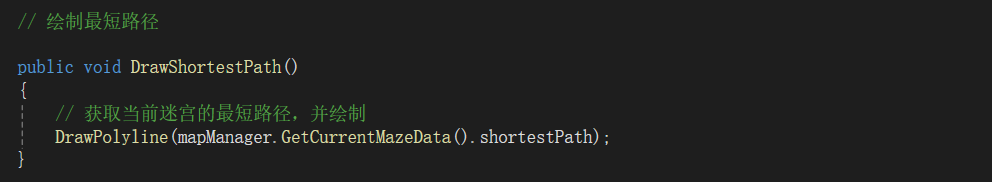


7.5.2 最短路径的展示

1. **控制路径展示：**通过 OnShortestPathButtonClicked() 方法触发，调用 pathDrawer.DrawShortestPath() 来绘制最短路径。并且隐藏了路径计数和下一路径按钮，显示“最短路径”的文本。



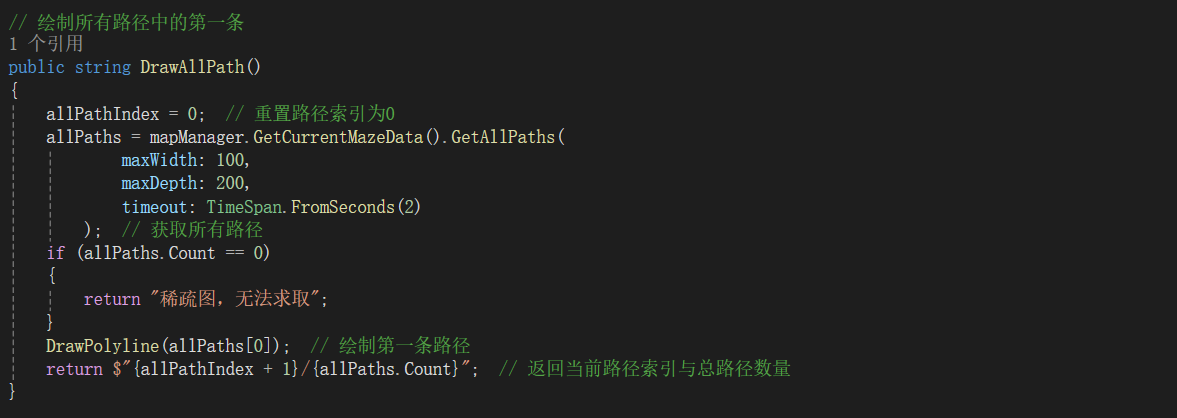
1. **获取并绘制最短路径：**通过 DrawShortestPath() 方法，获取当前迷宫数据中的最短路径，然后使用 DrawPolyline() 方法来将路径显示出来。



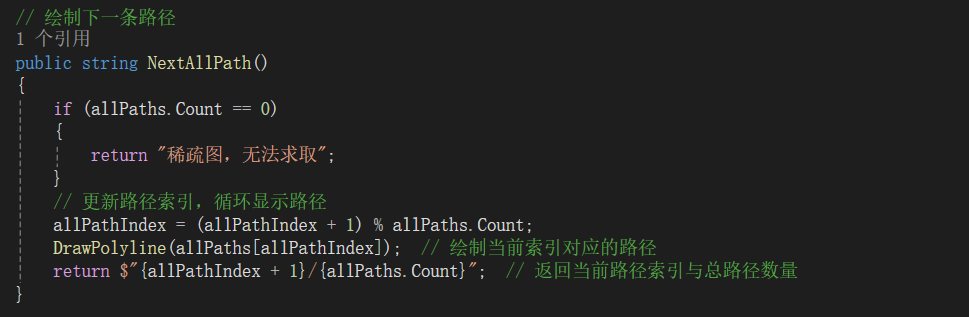
7.5.3 所有路径的展示和切换

所有路径的显示是通过 DrawAllPath() 和 NextAllPath() 方法实现的，支持按顺序显示所有路径。

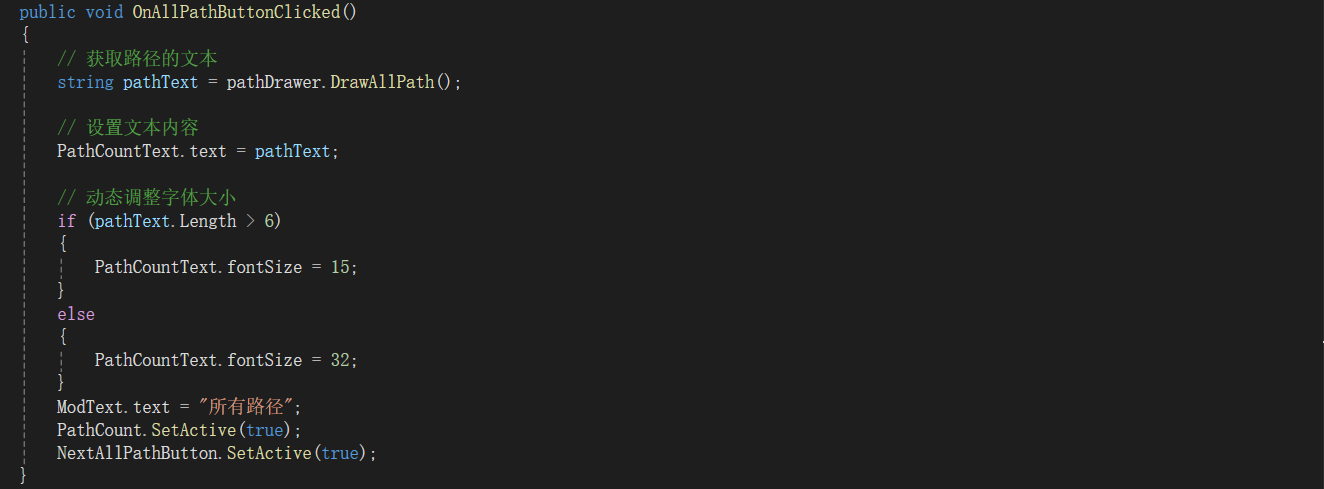
1. **绘制所有路径中的第一条**：DrawAllPath() 获取所有路径并绘制第一条路径，返回当前路径的索引和总路径数量。



1. **显示下一条路径**：NextAllPath() 方法更新路径索引，并绘制下一个路径。路径显示会循环显示。



1. **控制路径展示**：OnAllPathButtonClicked() 方法显示所有路径，更新路径数量的文本并动态调整字体大小。通过 OnNextAllPathButtonClicked() 切换到下一个路径。



# 八、测试和运行

## 8.1 测试环境与方法

单元测试与集成测试

性能测试：路径搜索效率、超时限制测试

## 8.2 运行结果

程序运行截图与功能验证

测试结果与性能评估

# 九、总结

系统的总结与评估

面临的挑战与解决方案

后续优化与扩展的方向