

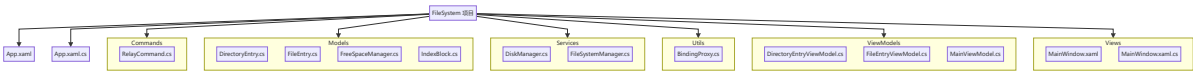
# 文件系统项目文档

## 一、项目结构简介

本项目为基于 C# 的 WPF 桌面应用，采用 MVVM 架构，实现了一个简易的文件系统模拟器。系统支持目录树操作、文件增删查改、持久化存储及空间管理，并通过 WPF 前端实现友好的可视化界面交互。

## 二、项目结构目录

项目结构如图所示：



- **Commands**：命令实现（如 `RelayCommand`），用于 MVVM 命令绑定
- **Models**：核心数据结构，包括目录、文件、空间管理等
- **Services**：业务逻辑，如磁盘和文件系统管理器
- **Utils**：辅助工具（如 `BindingProxy`）
- **ViewModels**：所有 ViewModel 层，负责数据与命令逻辑
- **Views**：界面层（如 `MainWindow.xaml` 和后台代码）

## 三、实验原理

### 概述

文件系统的核心任务是对文件在磁盘上的存储空间进行有效管理。主要包括三大技术要点：**文件分配方式**、**空闲空间管理方法**和**目录结构设计**。

- **分配方式**：指文件占用磁盘空间时，如何为其分配物理块。常见的分配方式有：
  1. **连续分配**：文件的所有数据块在磁盘上是连续的。优点是随机访问速度快，缺点是易产生外部碎片，扩展困难。
  2. **链接分配**：文件系统链接分配方式主要分为显式链接和隐式链接两类：**显式链接**指的是所有数据块之间的链接关系在磁盘的某个“专门区域”显式集中地保存。例如，FAT（File Allocation Table，文件分配表）就是显式链接的典型代表。在FAT方式中，磁盘有一张专门的分配表，表的每一项记录每个数据块的“下一个块号”。这样，系统可以直接通过查表来确定任意块的后继块，提高了随机访问的效率。优点：集中管理、随机访问更快（查表即可）。缺点：FAT表需常驻内存，表太大时不适合超大容量磁盘。**隐式链接**指的是每个数据块本身（块的尾部或头部）保存下一个数据块的指针。这样，块与块之间的关系是隐含在每个数据块自身的，类似链表。优点：结构简单，无需额外表结构。缺点：随机访问慢，需顺序遍历链表才能访问到指定块，不便于直接定位。
  3. **索引分配**：为每个文件分配专门的索引块，所有数据块的地址都记录在索引块中。优点是支持大文件、随机访问效率高，是现代文件系统常用的方法。
- **空闲空间管理**：指文件系统如何管理和分配未被占用的磁盘空间。常见方法有：
  1. **空闲块链表法**：所有空闲块通过链式指针串联，适用于分配和回收频繁的场景，但查找和维护略慢。

- 2. **位图法**：用一位表示一个磁盘块的使用状态，空间利用高，但适合磁盘块数不是特别巨大的场景。
- 3. **成组链接法**：将空闲块分组，每组维护部分块的指针，兼顾了链表和数组的优点，分配回收效率高，适合大容量磁盘。
- **目录结构**：文件系统为方便用户管理大量文件，采用了多种目录结构：
  - 1. **单级目录**：所有文件都放在同一个目录下，简单但不适合管理大量文件。
  - 2. **两级目录**：每个用户有一个独立的目录，提升了管理灵活性。
  - 3. **多级目录（树形目录）**：支持任意层级的子目录，是现代操作系统的标准做法，最大程度提高了文件管理的灵活性和可扩展性。

本项目采用**索引分配方式**，**成组链接法**，**多级目录**实现项目。

## 1. 索引分配方式

索引分配是一种高效的文件存储分配方式。其核心思想是：为每个文件分配一个专门的索引块，索引块中存储该文件所有数据块的物理地址（编号）。

- **优点**：支持大文件、快速随机访问、更便于管理碎片。
- **实现过程**：
  - 创建文件时，先分配一个索引块（如本项目中的 `IndexBlock`），所有数据块的编号通过指针保存在索引块中。
  - 写文件时，先计算所需数据块数，分配空闲块，将数据写入对应块，并把块号记录到索引块。
  - 读文件时，遍历索引块指针，顺序读取所有数据块内容。

在本项目中，`FileEntry` 对象通过 `IndexBlock` 字段与磁盘块一一对应，所有数据块的分配与释放均通过该索引块完成。

## 2. 成组链接法的空闲空间管理

成组链接是一种常见的磁盘空闲块管理方式，能高效地实现空间分配与回收。

- **主要原理**：将若干空闲块分成一组，每组通过链式指针连接，分配和回收时以组为单位进行，避免频繁扫描整个磁盘。
- **在本项目的实现**：
  - `FreeSpaceManager` 负责管理所有空闲块的分配与释放。
  - 分配新数据块或索引块时，从空闲组中取出；回收时再按组方式归还。
  - 状态持久化到 `freespace_state.json`，实现断点恢复。
  - 通过打印超级块和组表可直观查看空间管理状态。

这种方式提高了磁盘空间利用率，特别适合频繁分配与回收的文件系统场景。

## 3. 多级目录结构

多级目录是现代文件系统的基础，允许用户以树形方式组织和管理大量文件。

- **原理**：
  - 每个目录可包含若干子目录和文件，形成分层结构（树形结构）。
  - 根目录为顶层，所有文件和子目录均可以通过路径唯一定位。
- **在本项目中**：

- 用 `DirectoryEntry` 和 `FileEntry` 类分别表示目录和文件，目录对象拥有 `Children` 集合，递归描述所有子项。
- `Root` 为文件系统根目录，`CurrentDirectory` 记录当前工作目录，支持多层嵌套和递归操作。
- 支持递归新建、删除、重命名目录或文件，操作后自动持久化到 `file_state.json`。

#### 4. 三者协同实现文件系统核心

- 当用户在界面或命令行创建/删除文件或目录时，系统先更新目录树（多级目录），再通过空闲空间管理分配/回收磁盘块（成组链接），并使用索引块记录数据块位置（索引分配）。
- 所有操作最终都能映射到对磁盘块的高效操作（分配、释放、查找），保证系统一致性和高性能。
- 项目通过 MVVM 架构，实现了前后端的解耦，所有原理均可在代码和界面中直观体验。

## 四、核心数据结构

本项目核心数据结构包括：**文件/目录统一抽象（FileEntry/DirectoryEntry）**、**空闲空间成组链接管理（FreeSpaceManager）**、**索引块（IndexBlock）** 等。

### 1. 文件与目录的统一抽象与多态实现

代码片段：

```
1 public enum EntryType { File = 0, Directory = 1 }
2
3 public class FileEntry
4 {
5     public string Name { get; set; }
6     public EntryType Type { get; set; }
7     public long Size { get; set; }
8     public DateTime CreatedTime { get; set; }
9     public DateTime ModifiedTime { get; set; }
10    public IndexBlock IndexBlock { get; set; }
11    [JsonIgnore]
12    public DirectoryEntry Parent { get; set; }
13    // ... 构造函数等
14 }
15
16 public class DirectoryEntry : FileEntry
17 {
18     public List<FileEntry> Children { get; set; }
19     public DirectoryEntry() : base() { Children = new List<FileEntry>(); Type = EntryType.Directory; }
20     // ... AddChild, RemoveChild 等方法
21 }
```

实现要点：

- **继承与多态**：`DirectoryEntry` 继承自 `FileEntry`，通过 `Type` 属性区分文件和目录。这样所有文件和目录都能用统一的接口和集合（如 `List<FileEntry>`）管理，实现递归树结构。
- **多态集合**：`Children` 为 `List<FileEntry>`，可同时存放文件和子目录，实现任意层级树结构。
- **父子关系**：每个 `FileEntry` 有 `Parent` 字段，目录树可双向遍历。

- **构造与保护**：构造函数初始化类型和子集合，防止空指针。
- **扩展性**：如需实现符号链接等新类型，仅需继承 `FileEntry`。

## 2. 目录树的序列化与反序列化

代码片段：

```
1 public void SaveToFile(string filePath) {
2     var settings = new JsonSerializerSettings { ... };
3     string json = JsonConvert.SerializeObject(this, settings);
4     File.WriteAllText(filePath, json);
5 }
6 public static DirectoryEntry LoadFromFile(string filePath) {
7     string json = File.ReadAllText(filePath);
8     var settings = new JsonSerializerSettings { Converters = { new
9     FileSystemConverter() } };
10    var root = JsonConvert.DeserializeObject<DirectoryEntry>(json,
11    settings);
12    RebuildParentReferences(root);
13    return root;
14 }
```

- **序列化**：采用Json.NET将目录树对象转为JSON字符串，便于持久化存储和可视化调试。
- **自定义反序列化**：利用 `FileSystemConverter`，根据 `Type` 字段动态实例化 `DirectoryEntry` 或 `FileEntry`，实现多态反序列化。
- **递归恢复父指针**：`RebuildParentReferences` 递归修复父子关系，解决JSON无法自动还原父节点的问题。

## 3. 成组链接法的空闲空间管理

代码片段：

```
1 public class FreeSpaceManager
2 {
3     public int totalBlocks, groupSize, superBlockIndex;
4     public Dictionary<int, List<int>> groupTable;
5     // 初始化
6     public void InitialFreeSpace() { ... } // 将磁盘块分组，组内用栈，组间链表
7     public int AllocateBlock() { ... } // 分配块，动态切换超级块
8     public void FreeBlock(int blockNum) { ... } // 释放块，可能更换超级块
9     public void SaveToFile(string filePath) { ... }
10    public static FreeSpaceManager LoadFromFile(string filePath) { ... }
11 }
```

- **数据结构**：用 `Dictionary<int, List<int>>` 模拟磁盘的成组链接表，将空闲块分组管理（每组为一个管理块）。
- **分配/回收算法**：分配时先用超级块，满后切换下一组；释放时判断超级块是否可容纳，不可容纳时该块变为新超级块，完全模拟成组链接机制。
- **持久化**：用 `System.Text.Json` 序列化/反序列化空闲空间状态，方便实验演示和断点恢复。

## 4. 索引块的实现

代码片段:

```
1 public class IndexBlock
2 {
3     public int BlockSize, PointerSize, MaxPointers, UsedCount;
4     private int[] DataBlockPointers;
5     public int indexBlockId { get; private set; }
6     public IndexBlock(int blockSize, int pointersSize, int id) { ... }
7     public bool AddPointer(int blockNumber) { ... }
8     public bool RemovePointer(int blockNumber) { ... }
9     public int GetPointer(int index) { ... }
10    // ...
11 }
```

- **封装**: 所有指针和内部状态均封装为私有/只读属性, 通过方法维护一致性。
- **算法**: 支持动态分配/回收数据块指针, 判断是否已满, 支持随机访问和清空指针。
- **设计亮点**: 每个文件独立拥有一个索引块, 支持高效随机存取, 结构与Unix文件系统极为相似。

## 5. 多态与自定义序列化器

代码片段:

```
1 public class FileSystemConverter : JsonConverter
2 {
3     public override bool CanConvert(Type objectType) => objectType ==
4     typeof(FileEntry);
5     public override object ReadJson(JsonReader reader, Type objectType,
6     object existingValue, JsonSerializer serializer)
7     {
8         JObject jo = JObject.Load(reader);
9         var type = (EntryType)jo["Type"].Value<int>();
10        FileEntry entry = type == EntryType.Directory ? new DirectoryEntry()
11        : new FileEntry();
12        serializer.Populate(jo.CreateReader(), entry);
13        return entry;
14    }
15    // ...
16 }
```

- **实现要点**: 根据JSON中的 `Type` 字段, 动态实例化具体类型对象, 解决多态反序列化问题。
- **优势**: 支持任意层级、任意类型组合的文件系统树结构持久化与还原。

---

## 五、核心服务类 FileSystemManager

`FileSystemManager` 是核心服务类, 实现了文件系统的主要操作和状态维护, 关键属性和方法如下:

- **属性说明**:
  - `Root`: 根目录对象 (`DirectoryEntry` 类型)
  - `CurrentDirectory`: 当前工作目录

- `diskManager`: 虚拟磁盘块管理
- `spaceManager`: 空闲空间的分配与回收
- 配置参数如块大小、指针大小、总块数等
- **主要功能:**
  - 目录和文件的创建、重命名、删除 (支持递归删除目录)
  - 文件内容的写入和读取 (分块存储, 支持大文件写入)
  - 文件空间分配与回收, 持久化存储状态到 json 文件
  - 支持命令行模式操作及持久化
  - 提供当前目录下所有文件/文件夹的信息获取
- **持久化设计:**
  - `freespace_state.json`: 记录空间分配状态
  - `file_state.json`: 记录目录和文件结构
  - 启动时自动加载, 未找到则初始化
- **示例代码结构:**

```

1 public class FileSystemManager
2 {
3     // ... 属性定义
4     public FileSystemManager() { ... }
5     public DirectoryEntry CreateDirectory(string name) { ... }
6     public void DeleteDirectory(string name) { ... }
7     public FileEntry CreateFile(string name) { ... }
8     public void WriteFile(FileEntry fileEntry, byte[] data) { ... }
9     public void DeleteFile(string name, DirectoryEntry dir) { ... }
10    public string ReadFile(DirectoryEntry dir, string name, DiskManager
    disk) { ... }
11    public void RenameEntry(string oldName, string newName) { ... }
12    public List<EntryInfo> GetCurrentDirectoryInfos() { ... }
13    // 其他命令行与测试辅助函数
14 }

```

## 六、MVVM 架构说明

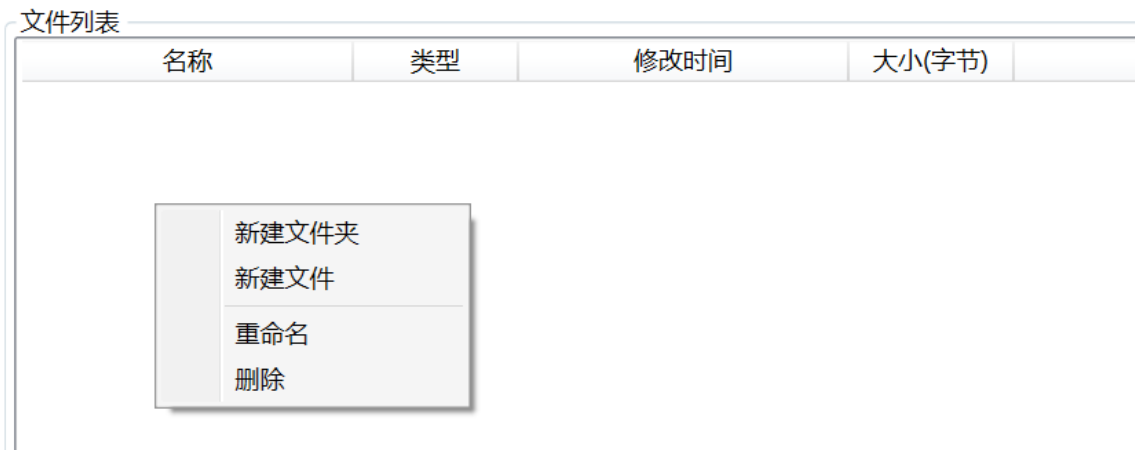
- **Model**: 负责数据结构 (如文件、目录、索引块、空闲块管理)
- **ViewModel**: 如 `MainViewModel`, 负责命令与数据绑定, 处理所有与界面相关的业务逻辑
- **View**: XAML 文件, 负责数据展示与用户交互
- **命令实现**: 如 `RelayCommand`, 配合 ViewModel, 响应界面操作
- `Models/DirectoryEntry.cs`、`FileEntry.cs`: 目录与文件结构定义
- `Models/FreeSpaceManager.cs`、`IndexBlock.cs`: 磁盘空间与索引块管理
- `Services/FileSystemManager.cs`: 文件系统核心服务
- `Services/DiskManager.cs`: 虚拟磁盘块读写
- `ViewModels/MainViewModel.cs`: 主界面数据和命令绑定

- `Views/Mainwindow.xaml`：主界面布局
- `Commands/RelayCommand.cs`：命令封装类
- `Utils/BindingProxy.cs`：辅助数据绑定

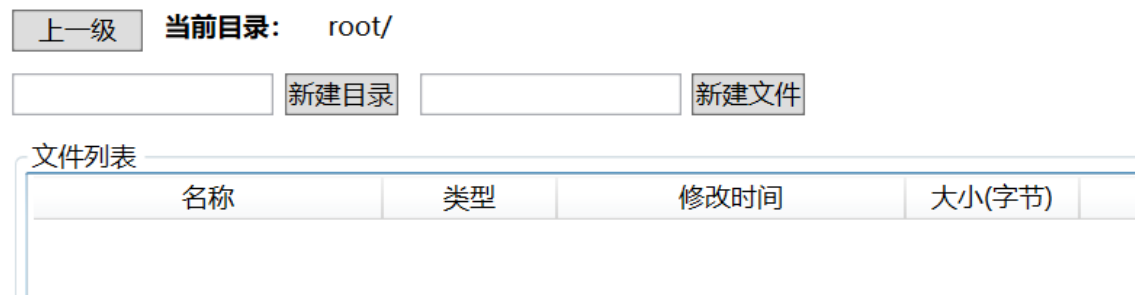
## 七、测试与展示

### 1. 新建目录/文件

- 右键点击



- 或者通过按钮创建



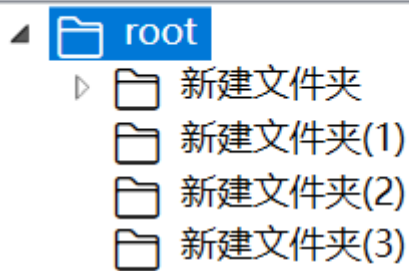
- 创建结果展示



### 2. 目录层级关系

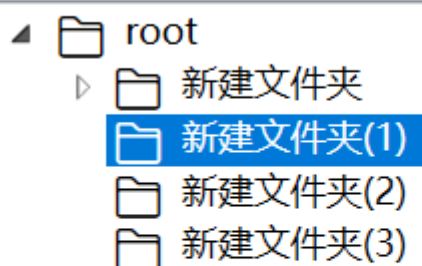
- 左侧目录树

## 目录结构




- 点击左侧栏目录进行切换

## 目录结构



- 双击右侧文件夹也可进行切换

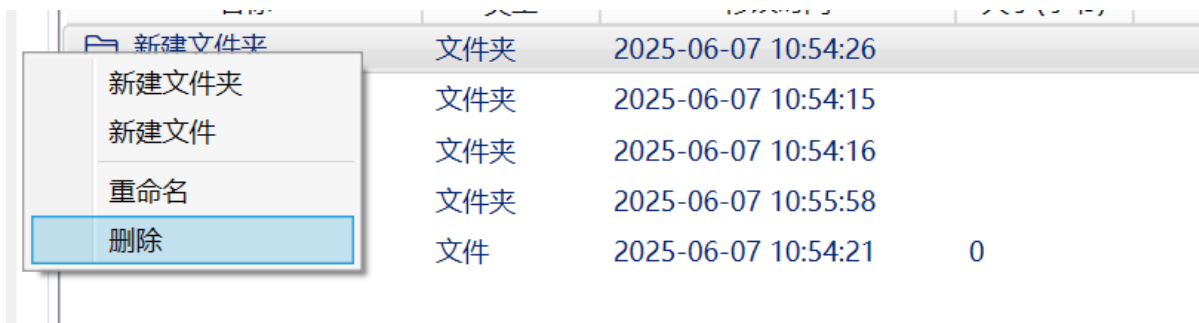
名称	类型	修改时间	大小(字
 新建文件夹	文件夹	2025-06-07 10:54:26	
 新建文件夹(1)	文件夹	2025-06-07 10:54:15	
 新建文件夹(2)	文件夹	2025-06-07 10:54:16	
 新建文件夹(3)	文件夹	2025-06-07 10:55:58	
 新建文件.txt	文件	2025-06-07 10:54:21	0

- 文件夹路径显示

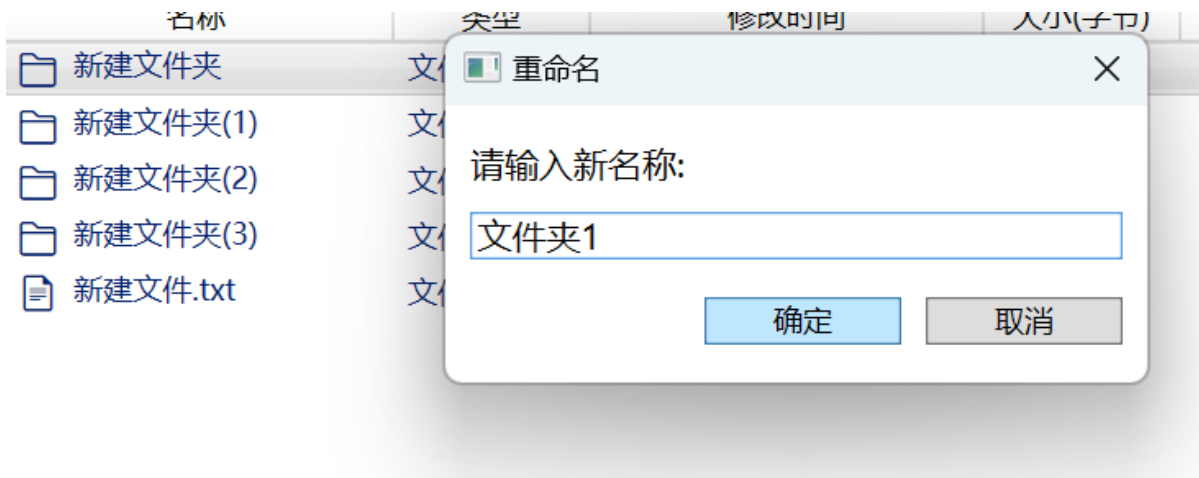
**当前目录:** root/新建文件夹(2)

### 3.文件或文件夹删除



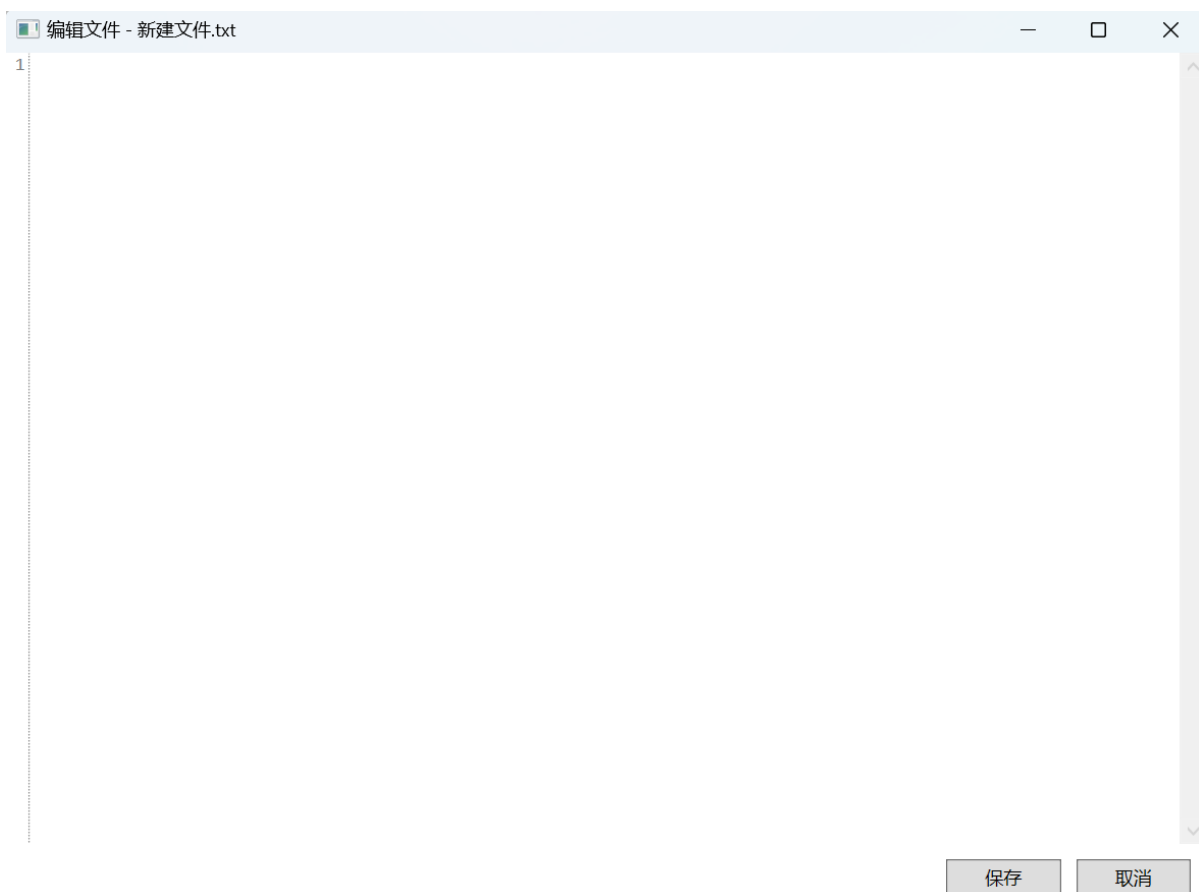


#### 4. 文件或文件夹重命名



#### 5. 文件读取和保存

双击文件可以打开读取并编辑文件



编辑结束可以实现保存



## 八、调试和模拟

本项目在 Service 层的 `FileSystemManager` 类中，专门实现了控制台交互版本的命令行虚拟文件系统。

用户只需实例化 `FileSystemManager` 并调用其 `RunCommandLine()` 方法，即可进入命令行交互界面。该模式下实现的文件系统操作与图形界面版功能完全一致。

这种设计既方便了前期的算法调试、数据结构验证，也便于后期的自动化测试和批量操作。通过控制台命令，开发者能够模拟文件的创建、删除、目录切换、内容读写、空间分配等一系列核心功能，从而高效地验证系统的正确性和健壮性。

使用方法如下：

```
1 FileSystemManager fileSystemManager = new FileSystemManager();  
2 fileSystemManager.RunCommandLine();
```

```

状态已从 freespace_state.json 加载
空闲空间状态已从 freespace_state.json 加载。
目录状态已从 file_state.json 加载。
欢迎使用简易文件系统命令行！输入 exit 退出。
===== 虚拟文件系统命令帮助 =====
mkdir 目录名          —— 创建目录
touch 文件名          —— 创建文件
write 文件名 内容     —— 写文件
readfile 文件名       —— 读文件
rm 文件名             —— 删除文件
listinfo 名称         —— 文件信息
rename 原名 新名      —— 重命名
rmdir 目录名          —— 删除目录及其内容
ls                   —— 列出当前目录内容
cd 目录名            —— 进入目录（支持..返回上级）
pwd                  —— 显示当前路径
testwrite            —— 长文本写入测试
exit                 —— 退出命令行
=====

root/>

```

命令行模式支持所有常用文件和目录操作，并通过丰富的日志输出，帮助开发者高效观察和分析系统内部状态与行为，提升测试与维护效率。

在下面演示中设置磁盘**20块**，每块**4KB**。

- 基本功能演示

```

root/> ls
root/> touch a
分配索引块10
超级块 0: 9 1 2 3 4 5 6 7 8 9
成组链接表内容:
管理块 0: 9 1 2 3 4 5 6 7 8 9
管理块 1: 9 -1 11 12 13 14 15 16 17 18 19

状态已保存到 freespace_state.json
空闲空间状态已保存到 freespace_state.json。
目录状态已保存到 file_state.json。
文件 a 创建成功。
root/> mkdir b
目录状态已保存到 file_state.json。
目录 b 创建成功。
root/> ls
<FILE>  a
<DIR>   b
root/> write a hahaha
分配数据块9
超级块 0: 8 1 2 3 4 5 6 7 8
成组链接表内容:
管理块 0: 8 1 2 3 4 5 6 7 8
管理块 1: 9 -1 11 12 13 14 15 16 17 18 19

写入磁盘9
状态已保存到 freespace_state.json
空闲空间状态已保存到 freespace_state.json。
目录状态已保存到 file_state.json。
写入文件 a 成功, 大小 6 字节。
root/> readfile a
hahaha
root/> ls
<FILE>  a
<DIR>   b
root/>

```

- 分配回收测试

①此时磁盘的20个块均未分配

```

超级块 16: 10 1 15 17 13 12 14 11 9 8 10
成组链接表内容:
管理块 16: 10 1 15 17 13 12 14 11 9 8 10
管理块 1: 9 -1 6 5 7 3 2 4 19 18 0

```

②不断新建文件夹和文件, 进行写入

```
写入磁盘9
写入磁盘11
状态已保存到 freespace_state.json
空闲空间状态已保存到 freespace_state.json。
目录状态已保存到 file_state.json。
写入文件 testfile.txt 成功，大小 5000 字节。
root/aa> cd ..
root/> mkdir 11
目录状态已保存到 file_state.json。
目录 11 创建成功。
root/> cd 11
root/11> testwrite
分配索引块14
超级块 16: 5 1 15 17 13 12
成组链接表内容:
管理块 16: 5 1 15 17 13 12
管理块 1: 9 -1 6 5 7 3 2 4 19 18 0
```

直到磁盘空间不足

```
状态已保存到 freespace_state.json
空闲空间状态已保存到 freespace_state.json。
目录状态已保存到 file_state.json。
分配数据块5
超级块 1: 1 -1 6
成组链接表内容:
管理块 1: 1 -1 6

分配数据块6
超级块 1: 0 -1
成组链接表内容:
管理块 1: 0 -1

写入磁盘5
写入磁盘6
状态已保存到 freespace_state.json
空闲空间状态已保存到 freespace_state.json。
目录状态已保存到 file_state.json。
写入文件 testfile.txt 成功，大小 5000 字节。
root/11/mm/yy> touch p
创建失败：空间不足
```

进行文件删除

```
root/11/mm/yy> touch p
创建失败：空间不足
root/11/mm/yy> cd ..
root/11/mm> cd ..
root/11> cd ..
root/> ls
<FILE>  a
<DIR>   aa
<DIR>   11
root/>
```

删除所有文件和文件夹后，发现磁盘块全部被回收成功

```
释放数据块6
超级块 17: 9 1 18 19 0 2 3 4 5 6
成组链接表内容:
管理块 17: 9 1 18 19 0 2 3 4 5 6
管理块 1: 9 -1 10 9 11 8 12 13 14 15 16
```

```
释放索引块7
超级块 17: 10 1 18 19 0 2 3 4 5 6 7
成组链接表内容:
管理块 17: 10 1 18 19 0 2 3 4 5 6 7
管理块 1: 9 -1 10 9 11 8 12 13 14 15 16
```

```
文件 testfile.txt 删除成功。
状态已保存到 freespace_state.json
空闲空间状态已保存到 freespace_state.json。
目录状态已保存到 file_state.json。
目录 11 删除成功。
目录状态已保存到 file_state.json。
```

---

## 九、如何运行与测试

1. 用 Visual Studio 打开项目，恢复依赖并编译
  2. 直接运行Release即可，支持图形界面操作
  3. 可在Visual Studio调用命令行函数测试大文件写入、递归删除等功能
-

## 十、联系方式

如有问题或建议，请联系作者：[3056399771@qq.com](mailto:3056399771@qq.com)