操作系统项目说明文档

——文件管理系统

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 王冬青

学院、 专业： 软件学院 软件工程

****

**目 录**

[1.项目分析 1](#_Toc13101)

[1.1 项目背景 1](#_Toc21362)

[1.2 项目要求 1](#_Toc21739)

[1.3 项目需求分析 1](#_Toc14167)

[2.项目设计 2](#_Toc29303)

[2.1 开发环境选择 2](#_Toc5400)

[2.2 类设计 2](#_Toc32675)

[2.2.1 物理块类（Block） 2](#_Toc28106)

[2.2.2 磁盘空间类（Disk） 3](#_Toc28106)

[2.2.3 文件类（File） 3](#_Toc28106)

[2.2.4 文件夹类（Folder） 4](#_Toc28106)

[2.2.5 文件结点类（FileNode） 5](#_Toc28106)

[2.2.6 文件工具类（FileTools） 5](#_Toc28106)

[2.2.7 名字——类型结构体（NameTypePair） 6](#_Toc28106)

[2.3 算法设计 6](#_Toc7531)

[2.3.1 新建 6](#_Toc28106)

[2.3.1.1 算法思路 6](#_Toc31155)

[2.3.1.2 算法代码 7](#_Toc31155)

[2.3.2 删除 7](#_Toc18684)

[2.3.2.1 算法思路 7](#_Toc31155)

[2.3.2.2 算法代码 8](#_Toc31155)

[2.3.3 打开 10](#_Toc6362)

[2.3.3.1 算法思路 1](#_Toc31155)0

[2.3.3.2 算法代码 1](#_Toc31155)1

[2.3.4 重命名 14](#_Toc16606)

[2.3.4.1 算法思路 1](#_Toc31155)4

[2.3.4.2 算法代码 1](#_Toc31155)4

[2.3.5 显示属性 15](#_Toc3889)

[2.3.5.1 算法思路 1](#_Toc31155)5

[2.3.5.2 算法代码 1](#_Toc31155)6

[2.3.6 文本编辑&写入数据 17](#_Toc6765)

[2.3.6.1 算法思路 1](#_Toc31155)7

[2.3.6.2 算法代码 1](#_Toc31155)7

[2.3.7 格式化磁盘 2](#_Toc6765)1

[2.3.7.1 算法思路](#_Toc31155) 21

[2.3.7.2 算法代码 2](#_Toc31155)1

[2.3.8 对实际磁盘的I/O操作 2](#_Toc6765)2

[2.3.8.1 算法思路 2](#_Toc31155)2

[2.3.8.2 算法代码 2](#_Toc31155)2

[3.项目测试 2](#_Toc29303)4

[4.项目评价 2](#_Toc29303)9

[4.1 可改进的地方 29](#_Toc5591)

[4.2 解决了的特殊问题 30](#_Toc26412)

**1.项目分析**

1.1 项目背景

磁盘空间存储着各种各样的文件，为了让用户可以对文件进行正常的读写等操作，文件管理系统需要对磁盘空间进行管理，同时也要管理所有文件形成的结构和每个文件的具体信息。

1.2 项目要求

在内存中开辟一个空间作为文件存储器，在其上实现一个简单的文件系统;退出这个文件系统时，需要该文件系统的内容保存到磁盘上，以便下次可以将其回复到内存中来。

文件存储空间管理可采取链接结构（如FAT文件系统中的显式链接等）或者其他学过的方法；空闲空间管理可采用位图或者其他方法；文件目录采用多级目录结构，目录项目中应包含：文件名、物理地址、长度等信息。

文件系统能提供的操作：格式化、创建子目录、删除子目录、显示目录、更改当前目录、创建文件、打开文件、关闭文件、写文件、读文件、删除文件......

1.3 项目需求分析

对于请求调页存储管理方式模拟系统，需考虑以下需求：

**·正确性**

程序应当能够正确提供文件管理的相关操作，同时在退出时可以将文件系统的内容保存到真正的磁盘中，尽可能正确的模拟出文件管理的情况。

**·健壮性**

程序应当对磁盘空间已满、命名中出现重名进行合适的处理。

**·交互性**

程序要以系统的形式出现，涉及到按按键等功能，需要给用户提供足够的提示与说明，使用方式要友好。同时由于有属性信息，显示时需要根据用户想知道且有权知道显示信息。

**2.项目设计**

2.1 开发环境选择

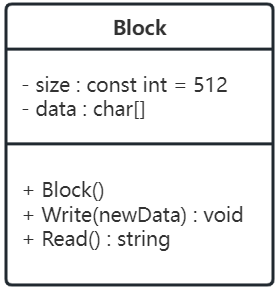
本项目的开发环境：

操作系统：Windows10 编程语言：C# 界面设计：WPF

2.2 类设计

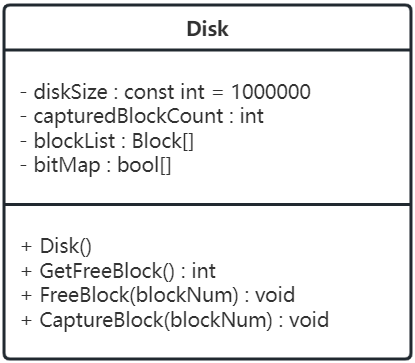
**2.2.1 物理块类（Block）**

物理块类（Block）封装了一个有512B数据的物理块，同时也提供了读和写的操作，其UML图如下：



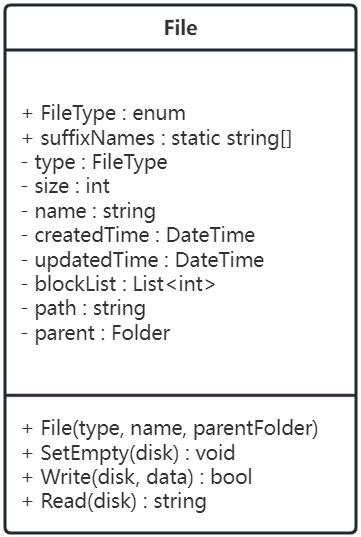
**2.2.2 磁盘空间类（Disk）**

磁盘空间类（Disk）封装了一个有106个物理块（仅该块被占用时才创建实例）及其位图的模拟磁盘空间，同时也提供了寻找空闲块位置、将块设置为占用或空闲状态的操作，其UML图如下：



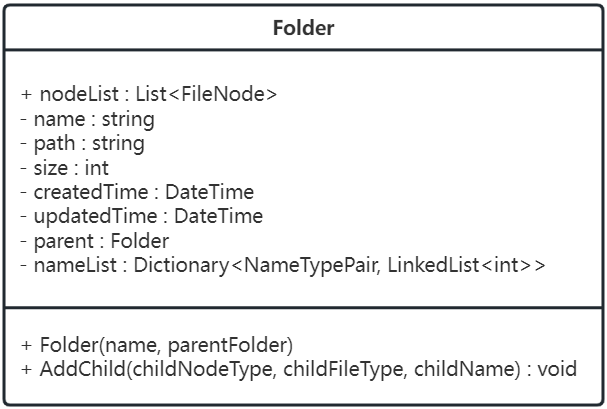
**2.2.3 文件类（File）**

文件类（File）并非文件本身，更接近于实际应用中的文件控制块。其封装了一个文件的具体信息（类型、名称、大小、创建时间、路径等）及其索引表，也封装了其双亲文件夹以便回溯。同时也提供了清空文件数据、尝试写入数据（若磁盘空间已满则写入失败）、读文件数据的操作。其UML图如下：



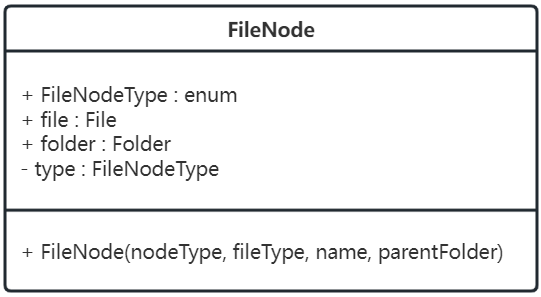
**2.2.4 文件夹类（Folder）**

文件夹类（Folder）除了封装了一些自身的信息（名称、大小等），还有一个子结点列表（表示在当前文件夹下的文件或文件夹）和一个名字字典（键是名称和类型的结构体，值是一个整型的链表，以管理重名情况）。同时也提供了向文件夹中添加子结点的操作。其UML图如下：



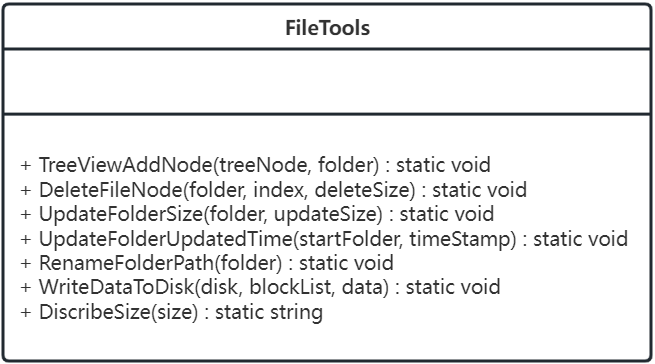
**2.2.5 文件结点类（FileNode）**

由于文件和文件夹构成了一个树形结构，而两者的类型完全不一致，为了有一个统一的文件树结点形式便于管理，FileNode类应运而生。它的构造函数指定其要么内含一个文件，要么内含一个文件夹，有一个成员会指示该FileNode实例内含什么。同时也提供了直接获取内含文件或文件夹的操作。其UML图如下：



**2.2.6 文件工具类（FileTools）**

文件工具类提供一些对特定的文件对象或窗口信息没有什么依赖、通用性很强且多处使用的方法，如递归更改文件夹大小或修改时间、给树形视图添加结点、输入字节数自动转换为以KB、MB等为结尾的大小。方法全声明为静态。其UML图如下：

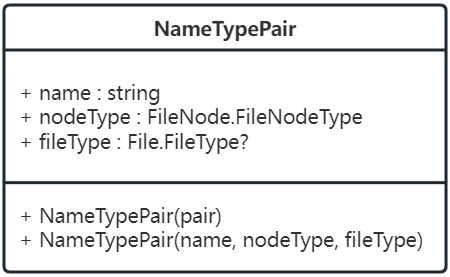


**2.2.7 名字——类型结构体（NameTypePair）**

名字——类型结构体（NameTypePair）有两类成员：名字和类型。前者描述一个文件或文件夹的名字，后者描述一个文件或文件夹的类型。

由于在同一文件夹下，同类且完全同名的两项不能出现，幸运的是在Windows中有以下形式：文件名(数字)，该方式可以管理同一文件夹下同类同名的项。而有哪些数字的项已存在，是文件新建、重命名时必须要考虑的。因此，每个文件夹下会有一个字典，键就是NameTypePair，值是存int型的链表，因为同名同类只能对应一种NameTypePair，故其所有重名情况下的有数字项全部在值的链表中。

其UML图如下：



2.3 算法设计

**2.3.1 新建**

**2.3.1.1 算法思路**

新建文件夹时，首先弹出一个窗口请求输入新建项的名称。若为空则警告，若存在重名则经过处理后返回合法的名称并创建，若名称无问题则直接创建。刚创建出的项为灰色，若文件夹有子项或文本文件有内容，则图标变为青色。

**2.3.1.2 算法代码**

/// <summary>

        /// 列表视图右键菜单：新建——文件夹

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void CreateFolderInListView(object sender, EventArgs e)

        {

            NameInputWindow window = new NameInputWindow(ref curFolder, null);  //创建名称输入窗口，传入参数为文件夹的参数

            window.callBack += RefreshAllOver;  //挂接更新所有视图的方法，在输入完成后调用

            window.ShowDialog();  //显示名称输入窗口，并完成输入及检查过程

        }

        /// <summary>

        /// 列表视图右键菜单：新建——文本文件

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void CreateFileInListView(object sender, EventArgs e)

        {

            NameInputWindow window = new NameInputWindow(ref curFolder, File.FileType.TXT);  //创建名称输入窗口，传入参数为文本文件的参数

            window.callBack += RefreshAllOver;  //挂接更新所有视图的方法，在输入完成后调用

            window.ShowDialog();  //显示名称输入窗口，并完成输入及检查过程

        }

**2.3.2 删除**

**2.3.2.1 算法思路**

只有在列表视图中的项有被选中的时候才进行删除，且只取第一个被选中的项进行删除。首先删除该结点，若该节点是文件夹则递归删除其所有子结点，若为文件则释放其所有的块，其次把结点对应的项从所在文件夹的名字字典中删除，以便其对应的名字能被重新使用。在此过程中统计删除项的大小总和，再自下而上更改含有删除项文件夹的大小和修改时间。

**2.3.2.2 算法代码**

/// <summary>

        /// 列表视图右键菜单：删除

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void DeleteFromMenuList(object sender, EventArgs e)

        {

            if (fileListView.SelectedItems.Count == 0)  //若列表视图没有项被选中，则不做任何操作

                return;

            ListViewItem item = fileListView.SelectedItems[0];  //若有选中则取第一个选中的项

            for (int i = 0; i < curFolder.nodeList.Count; i++)

            {

                if (listViewItems[i] == item)  //在当前列表视图中找到文件树中对应的结点

                {

                    DialogResult result = MessageBox.Show("确定删除" + curFolder.nodeList[i].Name +"吗？",

                        "Delete",

                        MessageBoxButtons.OKCancel,

                        MessageBoxIcon.Question,

                        MessageBoxDefaultButton.Button2);  //显示消息框进行二次确认

                    if (result == DialogResult.Cancel)  //若点击取消则不做任何操作

                        return;

                    int deleteSize = 0;  //删除的大小总和，初始为0

                    FileTools.DeleteFileNode(curFolder, i, ref deleteSize);  //删除该结点

                    FileTools.UpdateFolderSize(curFolder, deleteSize \* -1);  //更新所有有关文件夹的大小

                    FileTools.UpdateFolderUpdatedTime(curFolder, DateTime.Now);  //更新修改时间

                    RefreshAllOver();  //更新所有视图

                    break;

                }

            }

        }

/// <summary>

        /// 删除文件夹中的一个结点（若该结点为文件夹则递归删除其子结点）

        /// </summary>

        /// <param name="folder"> 待删除结点的双亲文件夹 </param>

        /// <param name="index"> 待删除结点，在双亲文件夹子结点列表中的下标 </param>

        /// <param name="deleteSize"> 本次删除文件的大小 </param>

        public static void DeleteFileNode(Folder folder, int index, ref int deleteSize)

        {

            if (folder.nodeList.Count == 0)  //若双亲文件夹无子结点，则不做任何操作

                return;

            string nodeName = folder.nodeList[index].Name;  //取待删除结点的名字

            List<NameTypePair> emptyPairs = new List<NameTypePair>();  //删除后为空的键值对

            foreach (var pair in folder.NameList)  //检查双亲文件夹的名字列表

            {

                int result = NameInputWindow.SubNameCheck(pair.Key.name, ref nodeName, false);  //将键值对的名字与待删除结点的名字比对

                if (result == -1)  //不符合则检查下一个对

                    continue;

                else  //符合则在值的链表中删除对应的下标，若删除后链表为空，则记录下该键值对

                {

                    pair.Value.Remove(result);

                    if (pair.Value.Count == 0)

                        emptyPairs.Add(new NameTypePair(pair.Key));

                }

            }

            foreach (var pair in emptyPairs)  //删除值的链表已经为空的键值对

            {

                folder.NameList.Remove(pair);

            }

            if (folder.nodeList[index].Type == FileNode.FileNodeType.Folder)  //若待删除结点为文件夹

            {

                for (int i = 0; i < folder.nodeList[index].folder.nodeList.Count; i++)

                {

                    DeleteFileNode(folder.nodeList[index].folder, 0, ref deleteSize);  //递归删除每一个子结点

                }

                folder.nodeList.RemoveAt(index);  //删除该结点

            }

            else  //若待删除结点为文件

            {

                deleteSize += folder.nodeList[index].file.Size;  //将该文件大小累加到deleteSize上

                folder.nodeList[index].file.SetEmpty(ref MainWindow.disk);  //SetEmpty之后就清空，故先不在此改变时间

                folder.nodeList.RemoveAt(index);  //删除该结点

            }

        }

**2.3.3 打开**

**2.3.3.1 算法思路**

分两种情况：双击列表视图的项打开或双击树形视图的项打开。

①列表视图：

只有在列表视图中的项有被选中的时候才进行打开，且只取第一个被选中的项，找到当前文件夹中对应的结点进行打开。

②树形视图

只有在列表视图中的项有被选中的时候才进行打开（树形视图有且只有一个结点被选中）。由于树形视图和文件树的构造一致，只需在树形视图中从被选中的结点出发，记录下当前结点在双亲结点子结点列表中的下标，之后再往上回溯直至回到根结点，就可以得到一个下标序列。接着从文件树出发，根据下标序列的倒序向下走，就可以找到“选中的”文件结点，将其打开即可。

③打开结点：

若该项是文件夹，则更新当前文件夹以及视图，将pathRecord当前位置之后的记录全部删除，此时也无法直接向前，前进按钮不可用，而回退按钮可用；若该项是文件，则打开文本编辑器。

**2.3.3.2 算法代码**

/// <summary>

        /// 双击列表视图时的打开操作

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void OpenFromListView(object sender, EventArgs e)

        {

            if (this.fileListView.SelectedItems.Count == 0)  //若列表视图没有项被选中，则不做任何操作

                return;

            ListViewItem item = this.fileListView.SelectedItems[0];  //若有选中则取第一个选中的项

            for (int i = 0; i < curFolder.nodeList.Count(); i++)

            {

                if (listViewItems[i] == item)  //在当前列表视图中找到文件树中对应的结点

                {

                    FileNode curNode = curFolder.nodeList[i];

                    OpenFileNode(ref curNode);  //将该文件结点打开

                    break;

                }

            }

        }

        /// <summary>

        /// 双击树形视图时的打开操作

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void OpenFromTreeView(object sender, EventArgs e)

        {

            if (this.fileTreeView.SelectedNode == null)  //若树形视图没有项被选中，则不做任何操作

                return;

            TreeNode curNode = this.fileTreeView.SelectedNode;  //当前树结点，初始为选中的结点

            TreeNode parentNode = curNode.Parent;  //双亲树结点，初始为选中结点的双亲结点

            List<int> indexes = new List<int>();   //路径记录列表

            while (parentNode != null)  //向上寻找直至根结点

            {

                for (int i = 0; i < parentNode.Nodes.Count; i++)  //将curNode在parentNode子结点列表中的下标记录在indexes中

                {

                    if (parentNode.Nodes[i] == curNode)

                    {

                        indexes.Add(i);

                        break;

                    }

                }

                //两指针向上一层

                curNode = curNode.Parent;

                parentNode = parentNode.Parent;

            }

            if (indexes.Count == 0)  //若indexes为空，则打开根文件夹

            {

                if (curFolder == rootFolder)  //若当前文件夹就是根文件夹，则不做任何操作，避免冗余操作和记录

                    return;

                OpenFileNode(ref rootFileNode);

            }

            else  //否则向下寻找打开结点

            {

                indexes.Reverse();  //记录是自下而上记录，现在自上而下寻找，反转列表的顺序

                FileNode current = rootFolder.nodeList[indexes[0]];  //初始是第一层的子项

                if (indexes.Count == 1)  //若第一层的子项就是路径终点，则直接打开

                {

                    if (current.Type == FileNode.FileNodeType.File

                        || (current.Type == FileNode.FileNodeType.Folder && curFolder != current.folder))

                        OpenFileNode(ref current);  //只有在current是文件，或与当前文件夹不同的文件夹时才打开，避免冗余操作和记录

                    return;

                }

                for (int index = 1; index < indexes.Count; index++)  //根据indexes的记录一层层向下

                {

                    current = current.folder.nodeList[indexes[index]];

                    if (index == indexes.Count - 1)  //若遇到了最后一位，则说明到达记录路径的终点，将该结点打开

                    {

                        if (current.Type == FileNode.FileNodeType.File

                        || (current.Type == FileNode.FileNodeType.Folder && curFolder != current.folder))

                            OpenFileNode(ref current);  //只有在current是文件，或与当前文件夹不同的文件夹时才打开，避免冗余操作和记录

                    }

                }

            }

        }

        /// <summary>

        /// 打开文件结点

        /// </summary>

        /// <param name="node"> 待打开的文件结点 </param>

        private void OpenFileNode(ref FileNode node)

        {

            if (node.Type == FileNode.FileNodeType.Folder)  //若代打开结点是文件夹

            {

                curFolder = node.folder;  //更新当前文件夹

                if (pathPointer < pathRecord.Count - 1)  //若pathPointer指向pathRecord的中间位置，则将这个位置之后的记录全部删除

                    pathRecord.RemoveRange(pathPointer + 1, pathRecord.Count - pathPointer - 1);

                pathRecord.Add(curFolder);  //在pathRecord中添加当前文件夹

                pathPointer++;  //pathPointer指向位置向前1位

                currentPathDataLabel.Text = curFolder.Path;  //更新当前路径标签

                buttonPathBackward.Enabled = true;  //顺序向前一步，可以回退，回退按钮可用

                buttonPathForward.Enabled = false;  //主动向前，原位置之后的记录全部清空无法前进，前进按钮不可用

                RefreshAllOver();  //更新所有视图

            }

            else

            {

                TextEditor window = new TextEditor(ref node.file);

                window.callBack += RefreshAllOver;

                window.ShowDialog();

            }

        }

**2.3.4 重命名**

**2.3.4.1 算法思路**

只有在列表视图中的项有被选中的时候才进行重命名，且只取第一个被选中的项进行重命名。根据选中项是文件或文件夹以合适的参数创建名称输入窗口并弹出，请求输入新名称。若为空则警告，若存在重名则警告名称已存在并阻止更名，若名称无问题则直接重命名。需要注意，若重命名的是文件夹，其所有子结点的路径都需要更新。

**2.3.4.2 算法代码**

/// <summary>

        /// 列表视图右键菜单：重命名

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void RenameFileNode(object sender, EventArgs e)

        {

            if (fileListView.SelectedItems.Count == 0)  //若列表视图没有项被选中，则不做任何操作

                return;

            ListViewItem item = fileListView.SelectedItems[0];  //若有选中则取第一个选中的项

            for (int i = 0; i < curFolder.nodeList.Count; i++)

            {

                if (listViewItems[i] == item)  //在当前列表视图中找到文件树中对应的结点

                {

                    FileNode node = curFolder.nodeList[i];

                    if (node.Type == FileNode.FileNodeType.Folder)

                    {

                        NameInputWindow window = new NameInputWindow(ref curFolder, null, node);  //创建名称输入窗口，传入参数为该文件夹的参数

                        window.callBack += RefreshAllOver;  //挂接更新所有视图的方法，在输入完成后调用

                        window.ShowDialog();  //显示名称输入窗口，并完成输入及检查过程

                    }

                    else

                    {

                        NameInputWindow window = new NameInputWindow(ref curFolder, node.file.Type, node);  //创建名称输入窗口，传入参数为该文件的参数

                        window.callBack += RefreshAllOver;  //挂接更新所有视图的方法，在输入完成后调用

                        window.ShowDialog();  //显示名称输入窗口，并完成输入及检查过程

                    }

                    break;

                }

            }

        }

/// <summary>

        /// 若重命名的是一个文件夹，则递归更改其所有子结点的路径

        /// </summary>

        public static void RenameFolderPath(Folder folder)

        {

            if (folder.nodeList.Count == 0)

                return;

            foreach (FileNode node in folder.nodeList)

            {

                node.Path = folder.Path + '\\' + node.Name;

                if (node.Type == FileNode.FileNodeType.Folder)

                    RenameFolderPath(node.folder);

            }

        }

**2.3.5 显示属性**

**2.3.5.1 算法思路**

若列表视图没有项被选中，则显示当前文件夹的属性。若列表视图有项被选中，则选择第一个项显示属性。并根据显示项的种类不同调整显示的图标。

**2.3.5.2 算法代码**

/// <summary>

        /// 列表视图右键菜单：属性

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void ShowFileNodeInfo(object sender, EventArgs e)

        {

            if (fileListView.SelectedItems.Count == 0)  //若列表视图没有项被选中，则显示当前文件夹的属性

            {

                if (curFolder.Parent != null)  //若当前文件夹有双亲文件夹，则在双亲文件夹的子结点列表中找到对应的文件结点

                {

                    foreach (FileNode node in curFolder.Parent.nodeList)

                    {

                        if (node.Name == curFolder.Name)

                        {

                            InfoWindow window = new InfoWindow(node);  //显示当前文件夹的属性

                            window.Show();

                            return;

                        }

                    }

                }

                else  //若当前文件夹无双亲文件夹，则显示根结点的属性

                {

                    InfoWindow window = new InfoWindow(rootFolder);

                    window.Show();

                    return;

                }

            }

            ListViewItem item = fileListView.SelectedItems[0];  //若列表视图有项被选中，则选择第一个项显示属性

            for (int i = 0; i < curFolder.nodeList.Count; i++)

            {

                if (listViewItems[i] == item)

                {

                    InfoWindow window = new InfoWindow(curFolder.nodeList[i]);

                    window.Show();

                    break;

                }

            }

        }

**2.3.6 文本编辑&写入数据**

**2.3.6.1 算法思路**

用户打开文本文件时，系统会根据文件的索引表逐块读出数据并显示在文本编辑窗口中。编辑过程中文本编辑窗口可以实时显示当前字数和实时设置字体大小。

当完成编辑后，系统尝试写入数据的过程如下：

①记录下原数据和原索引表

②清空文件的所有内容并释放所占的块

③计算新数据所需的块数newBlockNum（数据长度除以512向上取整）

④从磁盘中查询newBlockNum个空闲位置（每取到一个位置就占用）

⑤若在④中无法找到newBlockNum个空闲位置，则说明磁盘空间不足，释放④中已占用的空间，并根据原数据和原索引表写回原数据，返回假表示写入失败

⑥若在④中能找到newBlockNum个空闲位置，便将新数据写入并更新文件的索引表，然后从当前文件开始自下而上更新大小和修改时间，并返回真表示写入成功

**2.3.6.2 算法代码**

/// <summary>

        /// 创建一个文本编辑窗口

        /// </summary>

        /// <param name="editFile"> 待编辑的文本文件 </param>

        public TextEditor(ref File editFile)

        {

            InitializeComponent();

            this.ifEdit = false;  //未发生更改操作

            this.editFile = editFile;  //初始化正在编辑的文本文件

            this.labelFileName.Text = editFile.Name + File.suffixNames[(int)editFile.Type];  //初始化文件名标签

            this.richTextBox.LanguageOption = RichTextBoxLanguageOptions.UIFonts;  //指定新输入的字符与用户界面的字体一致

            this.richTextBoxFontSize.LanguageOption = RichTextBoxLanguageOptions.UIFonts;  //指定新输入的字符与用户界面的字体一致

            this.richTextBox.Text = editFile.Read(MainWindow.disk);  //从磁盘空间读出文件的数据，并显示在编辑窗口中

            this.labelCurrentCharInfo.Text = editFile.Size.ToString() + " 个字符";  //显示当前文件中的字符数量

        }

        /// <summary>

        /// 输入框文本发生改变时，同步显示当前文件中的字符数量

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void UpdateCurrentCharInfo(object sender, EventArgs e)

        {

            this.ifEdit = true;  //发生了更改操作，标志设为真

            this.labelCurrentCharInfo.Text = this.richTextBox.Text.Length.ToString() + " 个字符";  //同步显示字符数量

        }

        /// <summary>

        /// 按下“保存”键，完成编辑

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void SaveFile(object sender, EventArgs e)

        {

            if (this.ifEdit)  //若发生了更改，则弹出消息框进行二次确认

            {

                DialogResult result = MessageBox.Show("确定保存吗？",

                        "Save File",

                        MessageBoxButtons.OKCancel,

                        MessageBoxIcon.Question,

                        MessageBoxDefaultButton.Button2);

                if (result == DialogResult.Cancel)  //若点击取消则不做任何操作

                    return;

            }

            else  //若未发生更改则直接关闭窗口

                Close();

            bool writeResult = this.editFile.Write(ref MainWindow.disk, this.richTextBox.Text);  //尝试向文件中写入数据

            if (!writeResult)  //若写入失败，则弹出消息框告知磁盘已满

            {

                MessageBox.Show("磁盘空间已满，写入失败！",

                    "Disk Full Exception",

                    MessageBoxButtons.OK,

                    MessageBoxIcon.Error,

                    MessageBoxDefaultButton.Button1);

                return;

            }

            else if (this.callBack != null)  //若委托不为空，则调用委托挂接的方法

                this.callBack();

            Close();  //关闭窗口

        }

/// <summary>

        /// 设置文本编辑窗口的字号

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void SetEditorFontSize(object sender, EventArgs e)

        {

            int fontSize = 12;

            try

            {

                fontSize = Convert.ToInt32(this.richTextBoxFontSize.Text);

            }

            catch

            {

                this.richTextBox.Font = new Font("微软雅黑", 12);

            }

            if (fontSize > 0 && fontSize <= 72)

                this.richTextBox.Font = new Font("微软雅黑", fontSize);

            else

                this.richTextBox.Font = new Font("微软雅黑", 12);

        }

/// <summary>

        /// 向文件中写入数据

        /// </summary>

        /// <param name="disk"> 文件所在的磁盘空间 </param>

        /// <param name="data"> 待写入的数据 </param>

        /// <returns> 写入是否成功 </returns>

        public bool Write(ref Disk disk, string data)

        {

            string oldData = this.Read(disk);  //读出文件原数据

            int oldSize = this.size;           //取得文件原大小

            int totalLength = data.Length;     //写入数据的总长

            List<int> oldBlockList = this.blockList;  //取得文件原索引表

            SetEmpty(ref disk);  //清空文件原内容

            int newBlockNum = data.Length / 512 + (data.Length % 512 == 0 ? 0 : 1);  //按数据长度除以512（向上取整）取得所需块数

            List<int> newBlockList = new List<int>();  //新索引表

            for (int i = 0; i < newBlockNum; i++)

            {

                int freeBlockPos = disk.GetFreeBlock();  //取得一个空闲块位置

                if (freeBlockPos == -1)  //若取得-1，则剩余空间不够

                {

                    foreach (int pos in newBlockList)  //将检查过程中占用的块释放

                    {

                        disk.FreeBlock(pos);

                    }

                    FileTools.WriteDataToDisk(ref disk, oldBlockList, oldData);  //将原数据写回

                    this.size = oldSize;  //文件大小恢复

                    return false;  //写入失败，返回假

                }

                disk.CaptureBlock(freeBlockPos);  //若取得块号正常，则预先占用该块

                newBlockList.Add(freeBlockPos);   //将该块加入新索引表

            }

            FileTools.WriteDataToDisk(ref disk, newBlockList, data);  //将新数据写入

            this.blockList = newBlockList;  //更新索引表

            //除更新文件自身的以外，自下而上更新大小和修改时间

            DateTime timeStamp = DateTime.Now;

            this.size = totalLength;

            this.updatedTime = timeStamp;

            FileTools.UpdateFolderUpdatedTime(this.parent, timeStamp);

            FileTools.UpdateFolderSize(this.parent, totalLength - oldSize);

            return true;

        }

**2.3.7 格式化磁盘**

**2.3.7.1 算法思路**

用户可以直接从主窗口上方的菜单栏进行格式化，会有消息窗口弹出进行二次确认，之后直接将根目录下的所有子项进行删除，最后进行一些收尾工作，例如复位当前文件夹、设置前进/回退键等即可。需要注意，为了节省开销，在根目录为空时格式化磁盘功能不可用。

**2.3.7.2 算法代码**

/// <summary>

        /// 格式化磁盘

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void ResetDisk(object sender, EventArgs e)

        {

            DialogResult result = MessageBox.Show("确定格式化磁盘吗？",

                "Reset The Whole Disk",

                MessageBoxButtons.OKCancel,

                MessageBoxIcon.Question,

                MessageBoxDefaultButton.Button2);  //显示消息框进行二次确认

            if (result == DialogResult.Cancel)  //若点击取消则不做任何操作

                return;

            int deleteSize = 0;  //删除的大小总和，初始为0

            int listCount = rootFolder.nodeList.Count;  //根文件夹子节点数量

            for (int i = 0; i < listCount; i++)

            {

                FileTools.DeleteFileNode(rootFolder, 0, ref deleteSize);  //逐个删除子结点

            }

            rootFolder.Size -= deleteSize;  //更新根文件夹大小

            curFolder = rootFolder;  //当前文件夹回到根文件夹

            pathRecord.Clear();      //访问过的文件夹的列表清空，并加入根文件夹

            pathRecord.Add(rootFolder);

            pathPointer = 0;         //pathPointer指向0号位

            buttonPathForward.Enabled = false;   //根文件夹当前无子文件夹，无法前进，前进按钮不可用

            buttonPathBackward.Enabled = false;  //在根文件夹无法回退，回退按钮不可用

            RefreshAllOver();  //更新所有视图

        }

**2.3.8 对实际磁盘的I/O操作**

**2.3.8.1 算法思路**

本项目采用了对象序列化的思想，将一些会同时出现多个实例的类，如块、文件、文件夹等，和程序中模拟的磁盘空间做上标号，并通过流对象写入实际的磁盘文件。为了增强逻辑性，模拟的磁盘空间的相关信息，如位图和块数据，写在一个文件中，其余文件信息和树形结构写在另一个文件中。读取操作也是类似地通过标号逆向读取实例的所有信息。

**2.3.8.2 算法代码**

/// <summary>

        /// 将磁盘文件的信息读入

        /// </summary>

        /// <param name="diskFolderDir"> 磁盘文件所在文件夹路径 </param>

        /// <param name="diskCatalogDir"> rootCatalog.txt所在路径 </param>

        /// <param name="diskBitMapDir"> bitMap.txt所在路径 </param>

        private void IO\_ReadDiskFile()

        {

            if (!System.IO.Directory.Exists(diskFolderDir)

                || !System.IO.File.Exists(diskCatalogDir)

                || !System.IO.File.Exists(diskBitmapDir))

                return;

            FileStream ostreamTree, ostreamBitmap;

            BinaryFormatter binaryFormatter = new BinaryFormatter();

            ostreamTree = new FileStream(diskCatalogDir, FileMode.Open, FileAccess.Read, FileShare.Read);

            rootFileNode = binaryFormatter.Deserialize(ostreamTree) as FileNode;

            ostreamTree.Close();

            ostreamBitmap = new FileStream(diskBitmapDir, FileMode.Open, FileAccess.Read, FileShare.Read);

            disk = binaryFormatter.Deserialize(ostreamBitmap) as Disk;

            ostreamBitmap.Close();

        }

        /// <summary>

        /// 关闭主窗口时将所有信息保存到磁盘文件

        /// </summary>

        /// <param name="sender"></param>

        /// <param name="e"></param>

        private void IO\_SaveDiskFile(object sender, FormClosingEventArgs e)

        {

            DialogResult result = MessageBox.Show("是否保存到磁盘文件？",

                "Quit",

                MessageBoxButtons.YesNo,

                MessageBoxIcon.Question,

                MessageBoxDefaultButton.Button1);  //显示对话框，询问是否保存

            if (result == DialogResult.No)  //若按否键，则不做任何操作

                return;

            //若文件夹缺失，则创建文件夹（若文件缺失会在FileStream初始化时创建，故在此不处理）

            if (!System.IO.Directory.Exists(diskFolderDir))

                System.IO.Directory.CreateDirectory(diskFolderDir);

            FileStream ostreamTree, ostreamBitmap;

            BinaryFormatter binaryFormatter = new BinaryFormatter();

            ostreamTree = new FileStream(diskCatalogDir, FileMode.Create);

            binaryFormatter.Serialize(ostreamTree, rootFileNode);

            ostreamTree.Close();

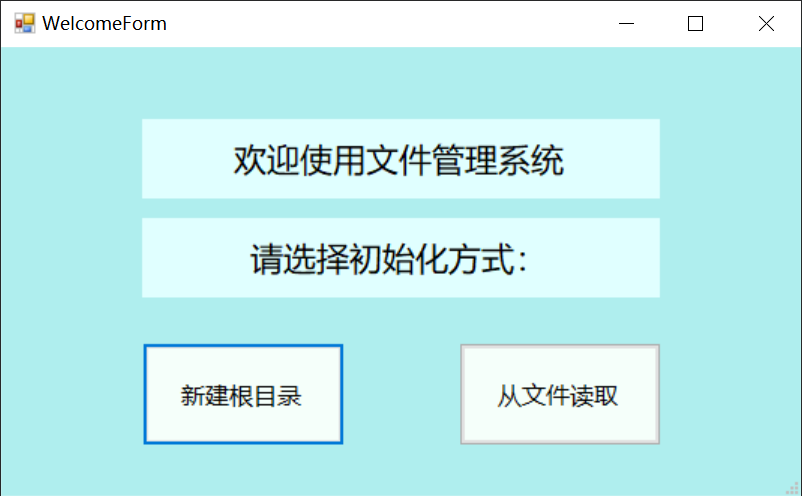
            ostreamBitmap = new FileStream(diskBitmapDir, FileMode.Create);

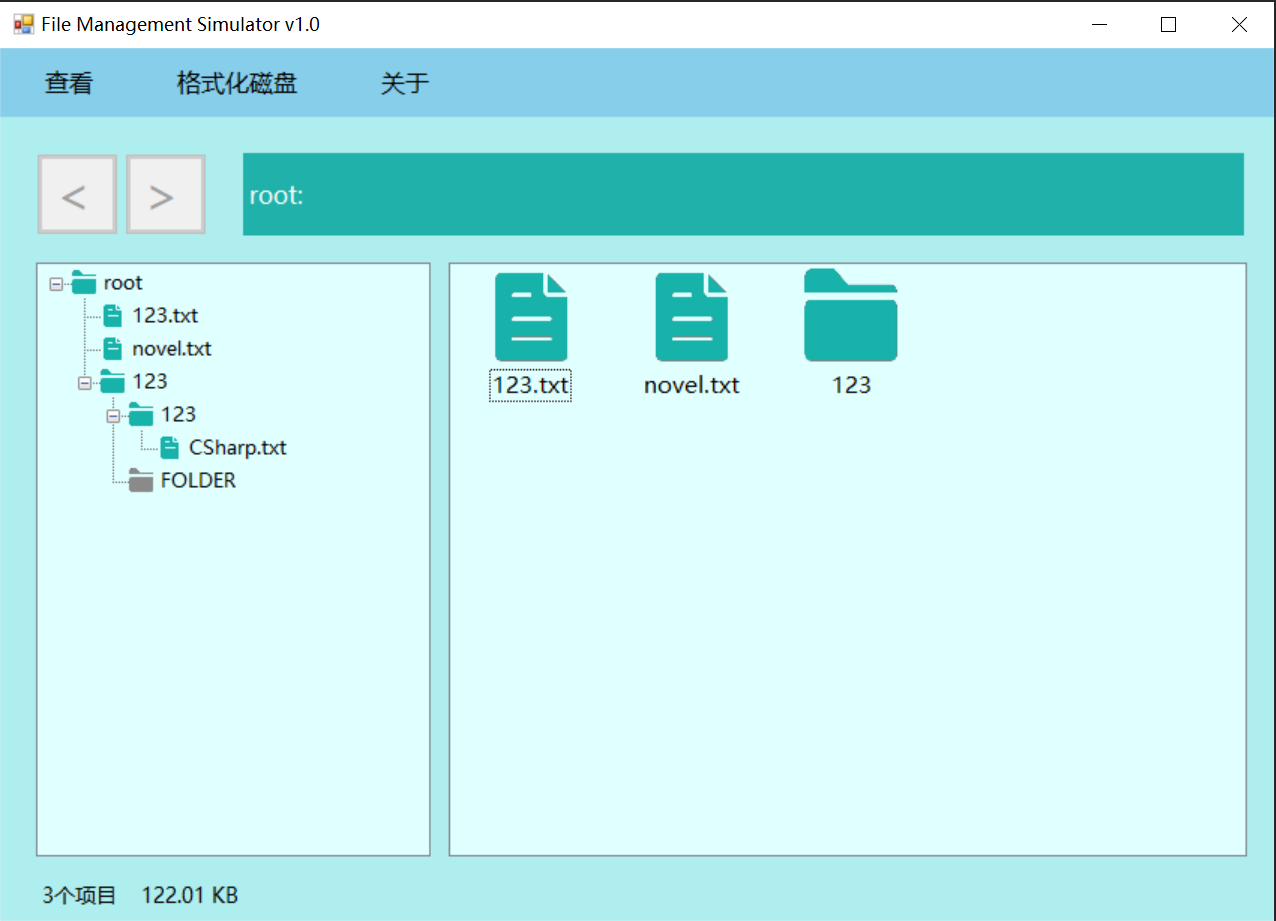
            binaryFormatter.Serialize(ostreamBitmap, disk);

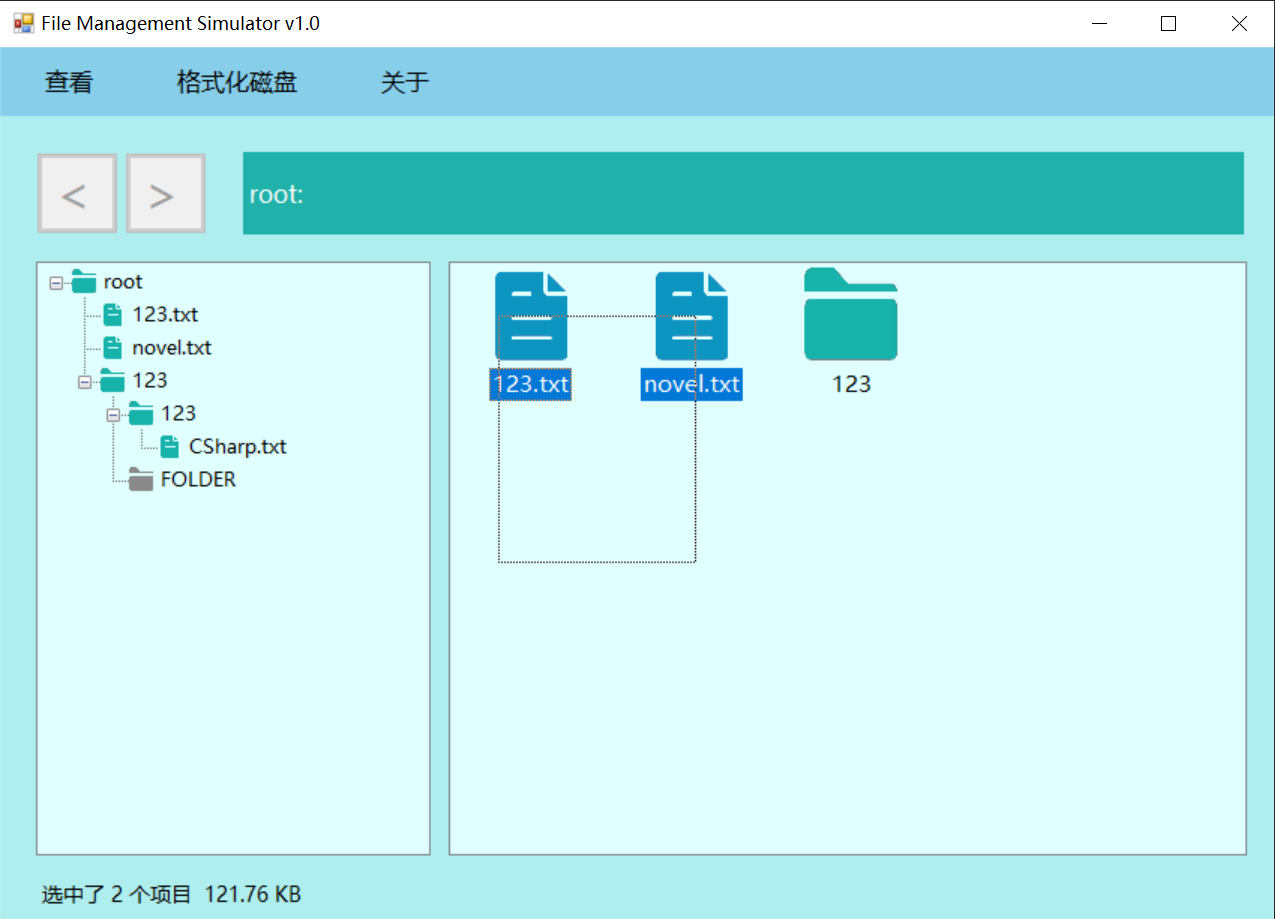
            ostreamBitmap.Close();

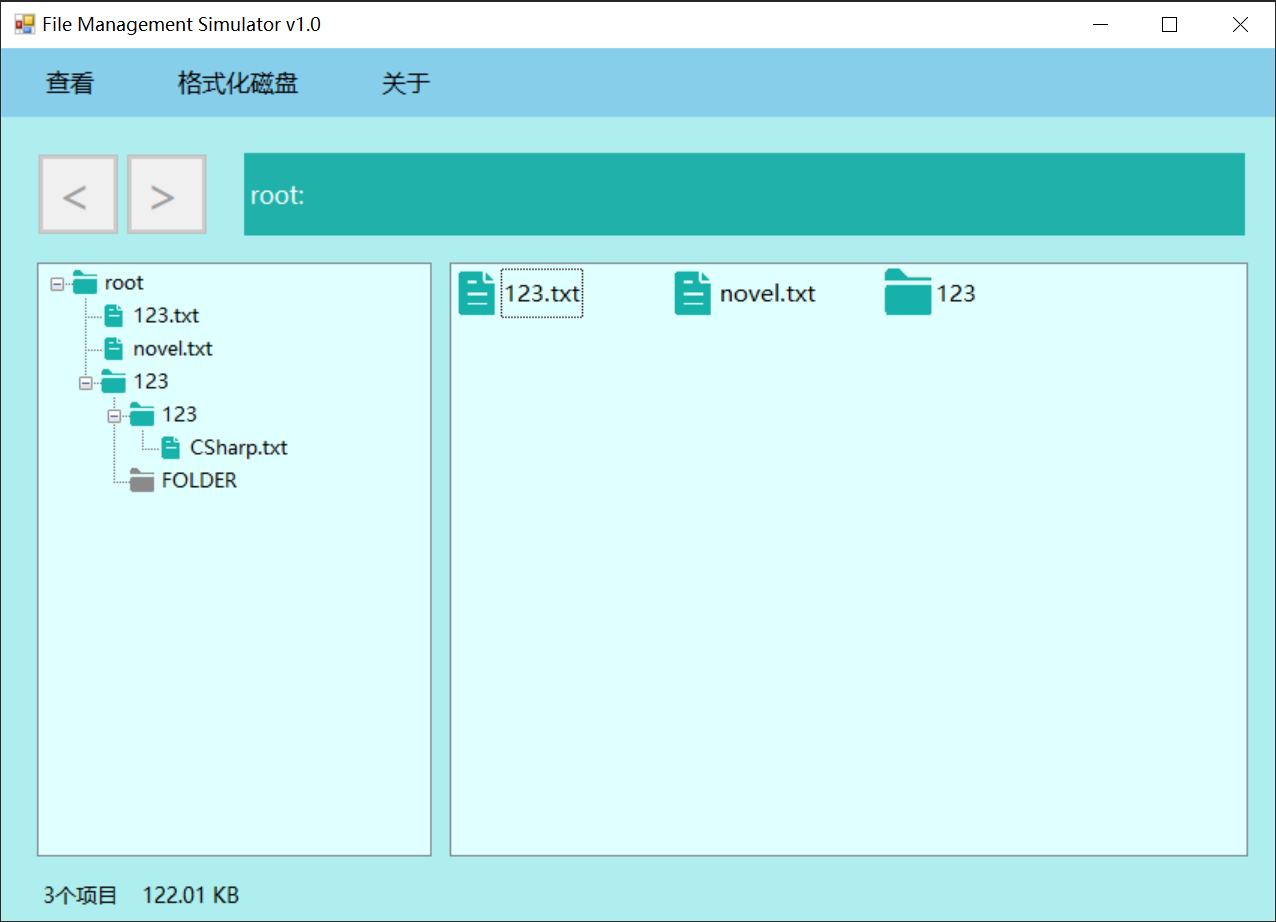
        }

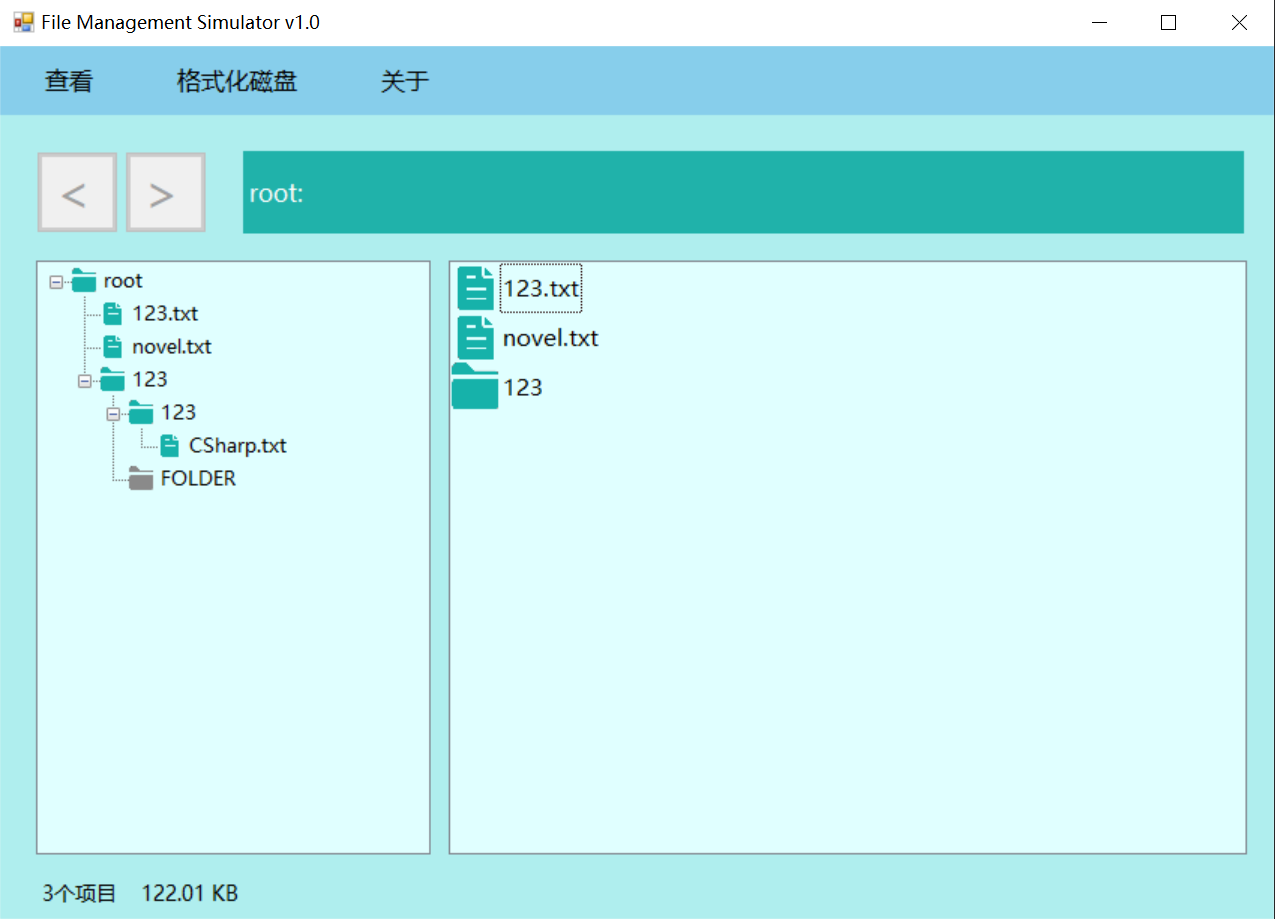
**3.项目测试**

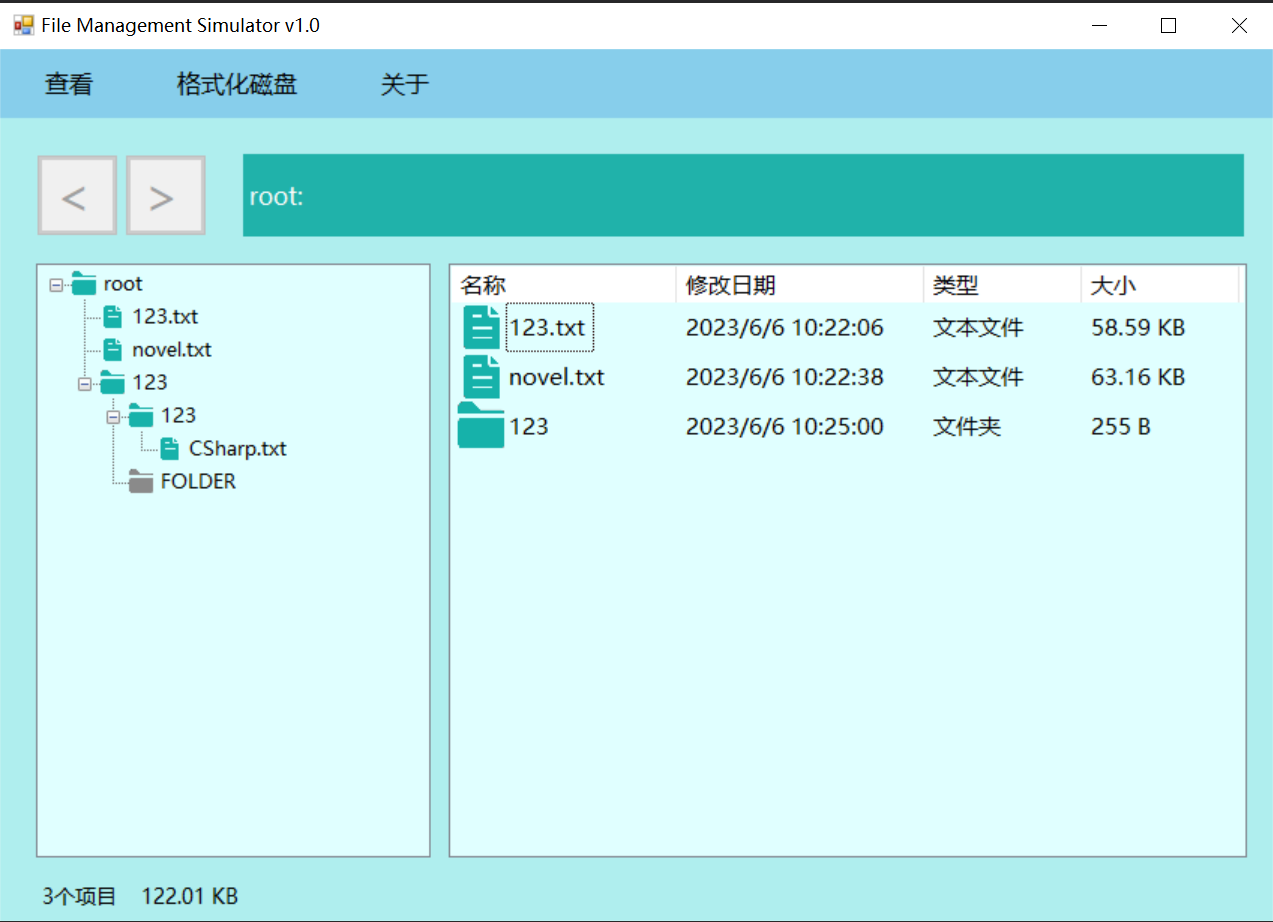
****

****

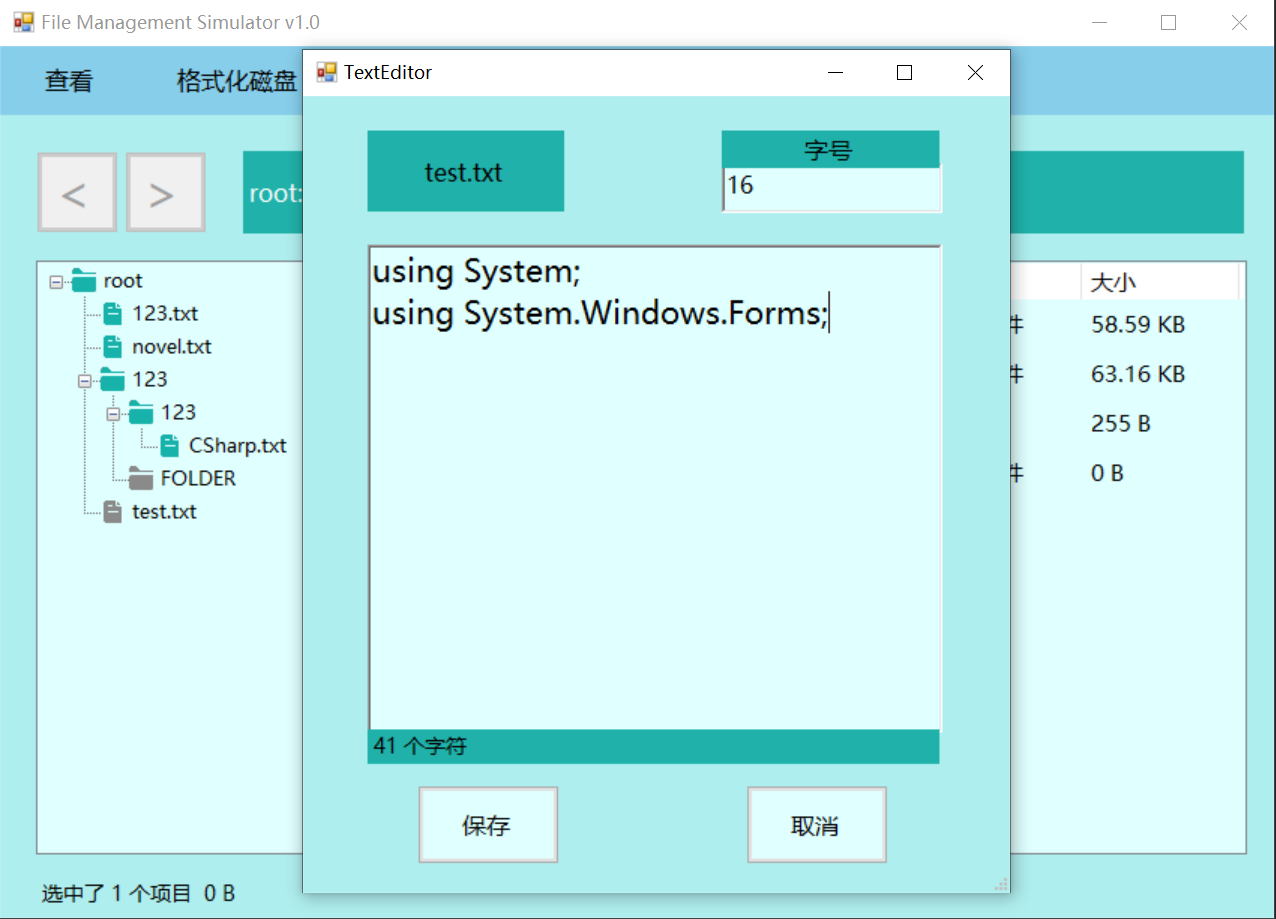
****

****

****

****

****

****

****

****

****

**4.项目评价**

4.1 可改进的地方

①未实现批量删除操作

②未实现复制、粘贴等较为高级的操作

③未实现树形视图的右键菜单栏

④未实现从子项数量上对于新建操作进行限制

⑤未实现根据文件类型和文件名对同一目录下的项进行排序，仅按照创建先后排序，观感上会造成一定的压力

⑤程序规模较大，设计的功能较多，对于测试这一方面可能会有欠缺导致存在意想不到的错误

4.2 解决了的特殊问题

①实现了多个视图展示当前文件夹中的项

②实现了当前路径的显示

③实现了双击树形视图的项进行打开，并解决了多次打开同一个文件夹出现冗余操作和记录的问题

④实现了输入名称为空时的警告和阻止

⑤实现了删除、格式化等不可逆操作前的询问

⑥实现了当前目录的前进/回退机制

⑦用字典方式管理名字，并尽可能完善了新建名字判断、重命名判断机制，避免了重名情况出现，同时实现了重命名功能

⑧解决了重命名文件夹时，其所有子结点路径更改的问题

⑨实现了在显示属性时，根据显示项种类的不同调整显示的图标

⑩实现了在文本编辑时的实时显示字数和实施设置字体大小

⑪实现了在空间上对于数据写入的限制

⑫实现了选中列表视图中的项时，实时显示选中项的个数和大小总和

⑬实现了在显示大小时根据字节数自动转换为KB、MB等单位

⑭实现了打开和关闭程序时，让用户决定是否读取/写入实际磁盘文件

⑮实现了文件缺失时，程序欢迎页面的“从文件读取”键不可用，以及写入磁盘文件时创建所缺的文件夹和文件

⑯将所有图片素材加入了资源文件，不需要依赖本地文件即可正确展示所有的图片