**操作系统第三次大作业**

**--文件管理**

**项目需求**

**项目基本要求**

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。

本项⽬实现了⼀个基本完整的⽂件系统，实现的功能点有：

* 格式化
* 新建⼦⽬录
* 删除⼦⽬录
* 显⽰当前⽬录
* 新建⽂件
* 删除⽂件
* 更改当前⽬录
* 打开⽂件
* 关闭⽂件
* 写⽂件
* 读⽂件
* ⽂件与⽬录重命名

**文件结构**

**├──** **img**

**│** **├── delete.png**

**│ ├── file.png**

**│ ├── folder.png**

**│ ├── pc.png**

**│ └── rename.png**

**├── dialog.py**

**├── editor.py**

**├── editor.ui**

**├── editor\_ui.py（ui自动生成文件）**

**├── file\_system.py**

**├── file\_system.ui**

**├── file\_system\_ui.py（ui自动生成文件）**

**└──** **file\_system\_core.py**

**开发环境**

* 开发环境:Windows 11
* 开发软件:pycharm
* 开发语言:python 3.8
* 开发框架:pyqt5

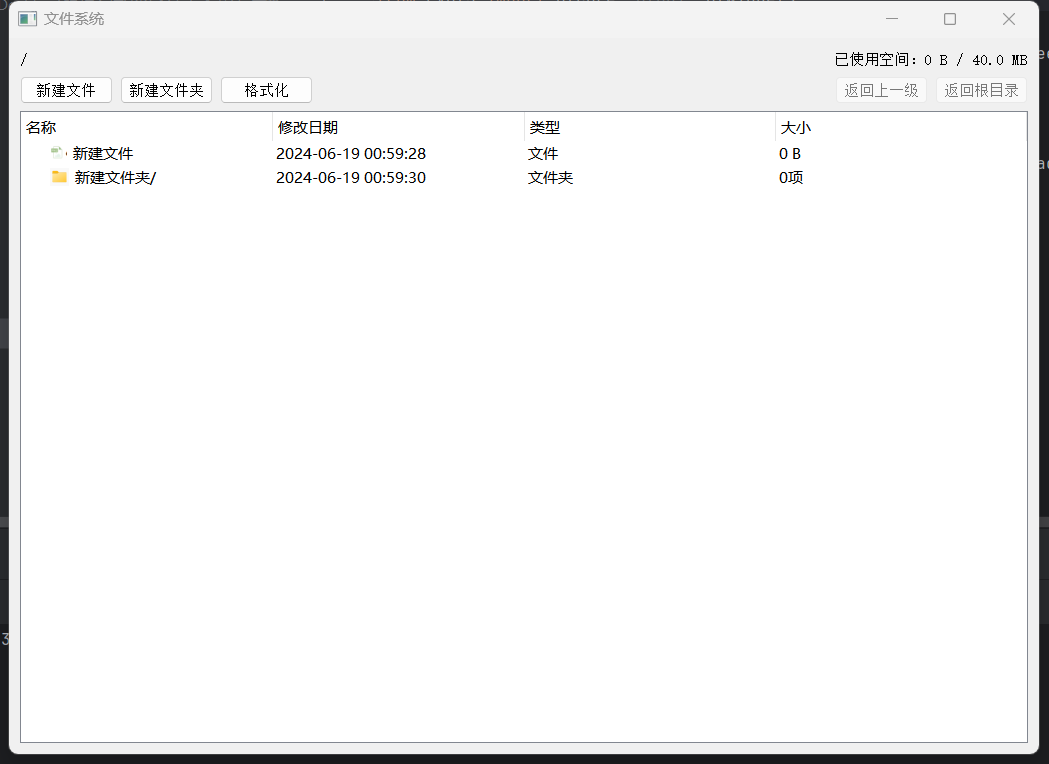
**项目展示**

**界面简介**

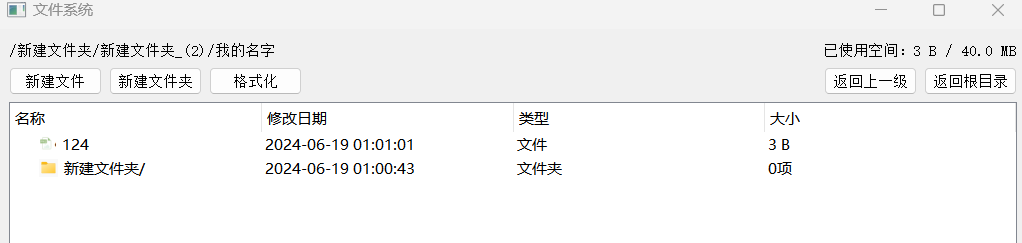
* 初始界面主要包括顶部导航界面，已使用空间占比；
* 选择新建文件，新建文件夹，格式化或者返回上一级，返回根目录的选项
* 为文件和文件夹信息预留的名称，修改日期，类型和大小导航栏

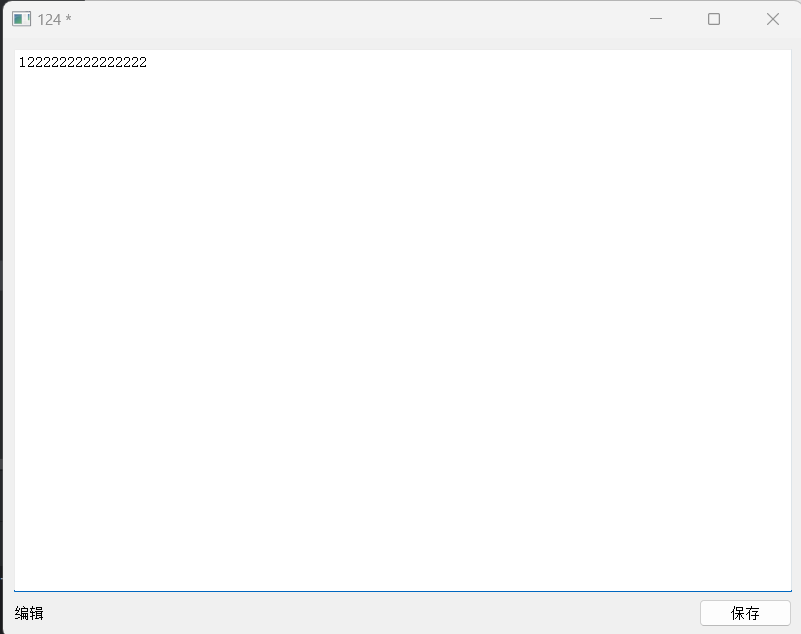


* 右键或者点击顶部新建文件/新建文件夹则会建立新的文件夹，并且存在默认命名，若命名重复，则进行后面追加“（1）”的更改默认命名。
* 右键可以删除文件，新建文件/文件夹，重命名操作



* 可以点击进入多级文件夹内部，上方导航栏会显示路径



* 双击文件可以进行文件编辑。

**设计方案**

1. **右键菜单 show\_menu 方法**

* show\_menu 方法：用于显示右键菜单，并根据用户选择执行相应的操作。
* 创建右键菜单：包括新建文件和文件夹、删除、重命名等操作。每个操作项都有对应的图标。
* 菜单项选择：根据用户选择的操作项，调用相应的槽函数执行具体的操作，如新建文件、新建文件夹、删除或重命名。

|  |
| --- |
| **def show\_menu(self, pos):**  # 创建右键菜单  menu = QMenu(self)  # 添加菜单项  createMenu = QMenu(menu)  createMenu.setTitle('新建')  new\_file\_action = createMenu.addAction("新建文件")  file\_icon\_path = r"img\file.png"  new\_file\_action.setIcon(QIcon(file\_icon\_path))  new\_directory\_action = createMenu.addAction("新建文件夹")  directory\_icon\_path = r"img\folder.png"  new\_directory\_action.setIcon(QIcon(directory\_icon\_path))  new\_action = menu.addMenu(createMenu)  delete\_action = menu.addAction("删除")  delete\_icon\_path = r"img\delete.png"  delete\_action.setIcon(QIcon(delete\_icon\_path))  rename\_action = menu.addAction("重命名")  rename\_icon\_path = r"img\rename.png"  rename\_action.setIcon(QIcon(rename\_icon\_path))  # 显示菜单，并等待用户选择  action = menu.exec\_(self.treeWidget.mapToGlobal(pos))  # 根据用户选择执行相应操作  if action == new\_file\_action:  self.new\_file\_dialog()  elif action == new\_directory\_action:  self.new\_directory\_dialog()  elif action == delete\_action:  self.delete()  elif action == rename\_action:  self.rename() |

1. **新建文件夹对话框 new\_directory\_dialog 方法**

* new\_directory\_dialog 方法：打开新建文件夹对话框。
* 自动重命名：如果文件夹名已存在，自动在名称后加上序号以区分。
* 文件设置类似

|  |
| --- |
| **def new\_directory\_dialog(self):**  original\_name = "新建文件夹"  name = original\_name  count = 1  # 检查文件夹是否存在，如果存在则自动重命名  while not self.fs.create\_directory(name):  name = f"{original\_name}\_({count})"  count += 1  self.list() |

1. **删除文件或文件夹 delete 方法**

* delete 方法：删除文件或文件夹。
* 检查当前目录是否有可删除项：如果没有可删除的文件或文件夹，弹出警告提示。
* 确定要删除的项：根据当前选中的文件树项确定要删除的文件或文件夹，并调用相应的删除方法。

|  |
| --- |
| **def delete(self):**  current\_items = [item.text(0) for item in  self.treeWidget.findItems("", QtCore.Qt.MatchContains | QtCore.Qt.MatchRecursive)]  if not current\_items:  QMessageBox.warning(self, "错误", "当前目录下没有可以删除的文件或文件夹")  return  item = self.treeWidget.currentItem()  if item is None:  QMessageBox.warning(self, "错误", "请选择要删除的文件或文件夹")  return  name = item.text(0)  if name.endswith("/"):  self.delete\_dir(name.rstrip("/"))  else:  self.delete\_file(name) |

**4.FCB（文件控制块）**

通常包括文件的基本信息（如文件名、文件大小、创建时间、修改时间等），在这里的实现中，Inode 类就是承担了类似于 FCB 的角色，记录了文件的基本信息和文件数据块的索引。

* Inode 类表示文件的索引节点，记录文件的大小、分配的数据块索引、创建时间（ctime）、修改时间（mtime）和访问时间（atime）。
* add\_block(block\_index: int) 方法用于添加数据块的索引。
* remove\_block(block\_index: int) 方法用于移除数据块的索引。

|  |
| --- |
| **class Inode:**  def \_\_init\_\_(self):  self.file\_size = 0  self.file\_blocks\_index = []  self.ctime = datetime.now()  self.mtime = datetime.now()  self.atime = datetime.now()  def add\_block(self, block\_index: int):  self.file\_blocks\_index.append(block\_index)  def remove\_block(self, block\_index: int):  self.file\_blocks\_index.remove(block\_index) |

**5.FAT（文件分配表）**

用于记录磁盘上哪些数据块已被使用，哪些是空闲的。在这里，valid\_blocks 位图实现了类似的功能，用 0 和 1 表示空闲和已使用状态。

* FileSystem 类用于管理整个文件系统，包括根目录、当前目录、文件数据块管理等。
* file\_block\_nums 表示文件系统的块数量。
* valid\_blocks 是一个位图，用于管理块的空闲和使用状态。
* space 是文件系统的实际数据块列表。
* used\_size 记录文件系统已使用的空间大小。

|  |
| --- |
| **class File:**  def \_\_init\_\_(self, name):  self.name = name  self.inode = Inode()  self.type = "file"  def read(self, fs: FileSystem) -> bytearray:  data = bytearray()  self.inode.atime = datetime.now()  for block\_index in self.inode.file\_blocks\_index:  data += fs.space[block\_index].read()  return data  def write(self, data: bytearray, fs: FileSystem) -> bool:  valid\_block\_nums = fs.get\_valid\_block\_nums()  block\_count = len(data) // (1024\*4) + 1  if block\_count > valid\_block\_nums:  print("No more space available")  return False  self.clear(fs)  self.inode.file\_size = len(data)  fs.used\_size += self.inode.file\_size  for i in range(block\_count):  j = 0  for j in range(fs.file\_block\_nums):  if fs.valid\_blocks[j] == 0:  fs.valid\_blocks[j] = 1  block = fs.space[j]  if i == 0:  self.inode.mtime = datetime.now()  self.inode.atime = datetime.now()  block.write(  data[i \* 1024\*4: min((i + 1) \* 1024\*4, len(data))])  self.inode.add\_block(j)  break  if j == fs.file\_block\_nums-1:  print("No more space available")  return False  return True  def clear(self, fs: FileSystem):  fs.used\_size -= self.inode.file\_size  self.inode.ctime = datetime.now()  self.inode.mtime = datetime.now()  self.inode.atime = datetime.now()  for i in self.inode.file\_blocks\_index:  fs.valid\_blocks[i] = 0  self.inode.file\_blocks\_index = [] |

**文件架构以及运行**

|  |
| --- |
| 源代码：  文件管理源代码  打包exe可执行文件：  dist文件内部，一个exe可执行文件，一个压缩zip版防止不可用  说明文档：  设计文档.pdf  运行：  若想本地运行，则需配置qt虚拟环境，运用python3.8进行本地运行  若想看项目展示，则直接进行exe即可。 |