## 文件管理模拟系统 设计方案报告

### 目录

* [开发环境](#开发环境)
* [项目结构](#项目结构)
* [界面](#界面)
* [操作说明](#操作说明)
* [实现的功能](#实现的功能)
* [系统分析](#系统分析)
  + [实现的类](#实现的类)
    - [FCB类](#fcb类)
    - [Node类](#node类)
    - [Category类](#category类)
* [系统实现](#系统实现)
  + [为文件内容分配磁盘空间](#为文件内容分配磁盘空间)
  + [获取文件内容](#获取文件内容)
  + [删除文件内容](#删除文件内容)
  + [更新文件内容](#更新文件内容)
  + [检查同目录下是否有同名文件](#检查同目录下是否有同名文件)
  + [保存文件内容](#保存文件内容)
  + [创建目录树](#创建目录树)
  + [磁盘文件的写入和读取](#磁盘文件的写入和读取)
  + [目录文件的写入和读取](#目录文件的写入和读取)

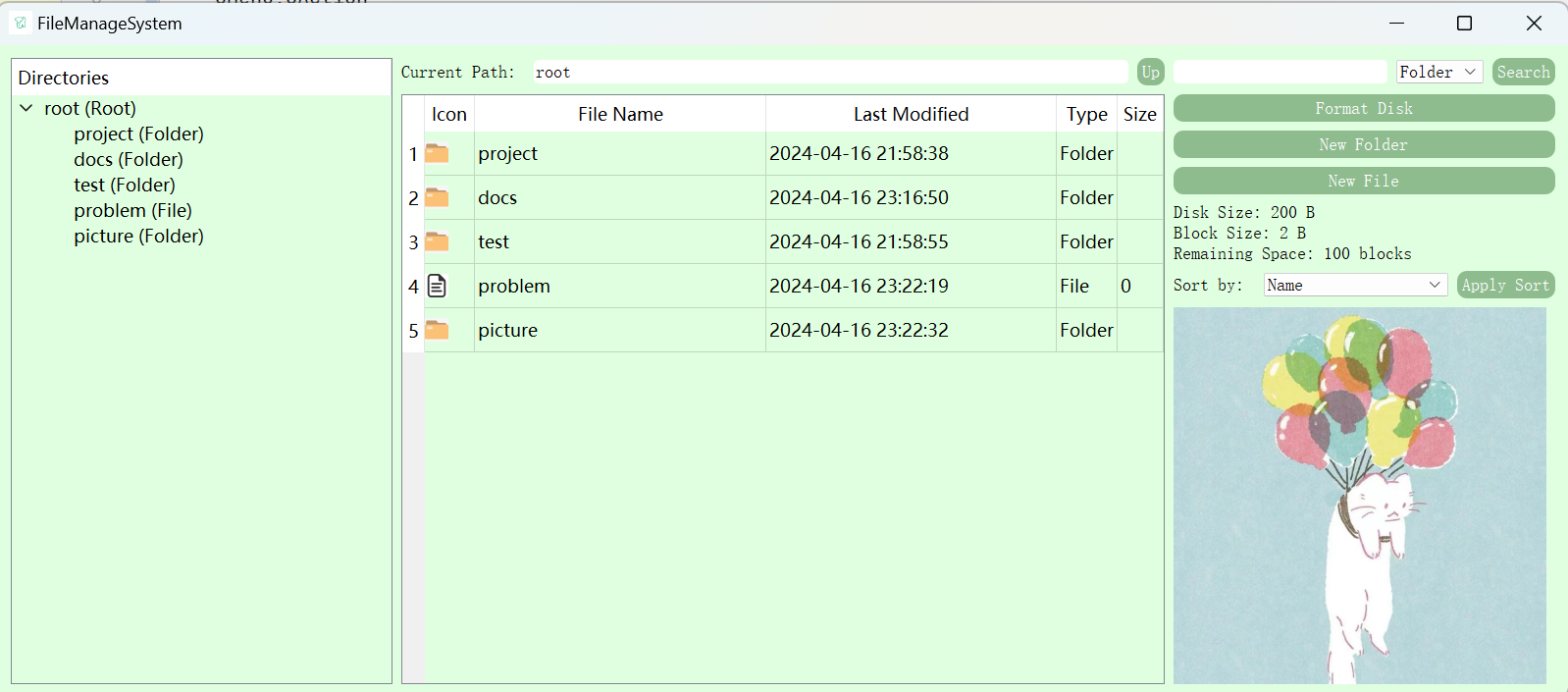
### 开发环境

* 开发环境：Windows 11
* 开发软件：Pycharm
* 开发语言：Python 3.11
* 主要引用模块：
  + PyQt5
  + sys
  + os

### 项目结构

FileManageSystem/  
 BitMapInfo.txt  
 Category.py  
 CategoryInfo.txt  
 FCB.py  
 main.py  
 MyDiskInfo.txt  
 project\_structure.txt  
 VirtualDisk.py  
 运行时截图.png  
 docs/  
 文件管理模拟系统 设计方案报告.md  
 res/  
 cat.jpg  
 file.png  
 folder.jpg  
 icon.jpg

### 界面



### 操作说明

* 点击右侧按钮可以新建文件，文件夹，格式化磁盘；
* 当用鼠标右键点击空白区域时，可以新建文件或文件夹；
* 当用鼠标右键点击文件或文件夹名称时，可以打开或者删除文件和文件夹；
* 点击文件或文件夹名可以打开文件编辑框或者打开文件夹；
* 当退出文件编辑框时可以选择是否保存修改；
* 界面右侧上方可以输入名称，选择类型，然后在当前目录下进行搜索；
* 可以按文件类型，文件名和修改日期对文件夹和文件进行排序。

### 实现的功能

* 当前目录下文件和文件夹信息的显示；
* 文件和文件夹的创建与删除；
* 文件夹的打开，文件的编辑；
* 格式化磁盘；
* 文件和文件夹的搜索；
* 树状目录结构示意图；
* 按文件名，修改实现，文件类型排列文件和文件夹；
* 返回上级目录；
* 当前路径的显示。

### 系统分析

#### 实现的类

* FCB类：文件控制块，记录文件名称，文件类型，修改日期，文件大小，在磁盘中的起始存储位置；
* Node类：存储子节点，记录父节点，映射为文件之间的关系；
* Category类：存储整个文件系统的目录信息，其中的root记录了根节点，提供了一些方法：
  + free\_category(self, p\_node)：释放指定节点的目录；
  + search(self, p\_node, file\_name, file\_type)：在指定节点下搜索文件或文件夹；
  + search\_in\_current\_directory(self, p\_node, file\_name, file\_type)：只在指定目录下搜索文件或文件夹；
  + create\_file(self, parent\_node, fcb)：创建文件或文件夹；
  + check\_same\_name(self, p\_node, name, file\_type)：判断在同一目录下是否存在同名文件或文件夹。
* VirtualDisk类：模拟对磁盘的操作，记录了磁盘大小，存储块大小，存储块数量，剩余存储块数量，内存和位图，提供了一些方法：
  + get\_block\_size(self, size)：得到指定大小所要占用的存储块数量；
  + give\_space(self, fcb, content)：为特定fcb分配存储content的空间，将内容写入磁盘中，并对位图做出修改；
  + get\_file\_content(self,fcb)：返回指定fcb在存储在磁盘中的内容；
  + delete\_file\_content(self, start, size)：删除指定起始位置和大小存储在磁盘上的内容；
  + file\_update(self,old\_start,old\_size,new\_fcb,new\_content)：更新文件内容。
* MainWindow类：主窗口，维护了主要的程序逻辑；
* HelpDialog类：操作帮助窗口，在初次打开文件管理系统时弹出，给用户提供操作帮助；
* CreateDialog类：新建文件和文件夹窗口，在新建文件和文件夹时弹出；
* NoteForm类：编辑文本文件的界面，在编辑文本文件时弹出；

#### 显式链接法

* 本文件系统中, 文件存储空间管理使用显示链接的方法，文件中的内容存放在磁盘不同的块中，每次创建文件时为文件分配数量合适的空闲块。每次写文件时按顺序将文件内容写在相应块中; 删除文件时将原先有内 容的位置置为空即可。

#### 位图、FAT表

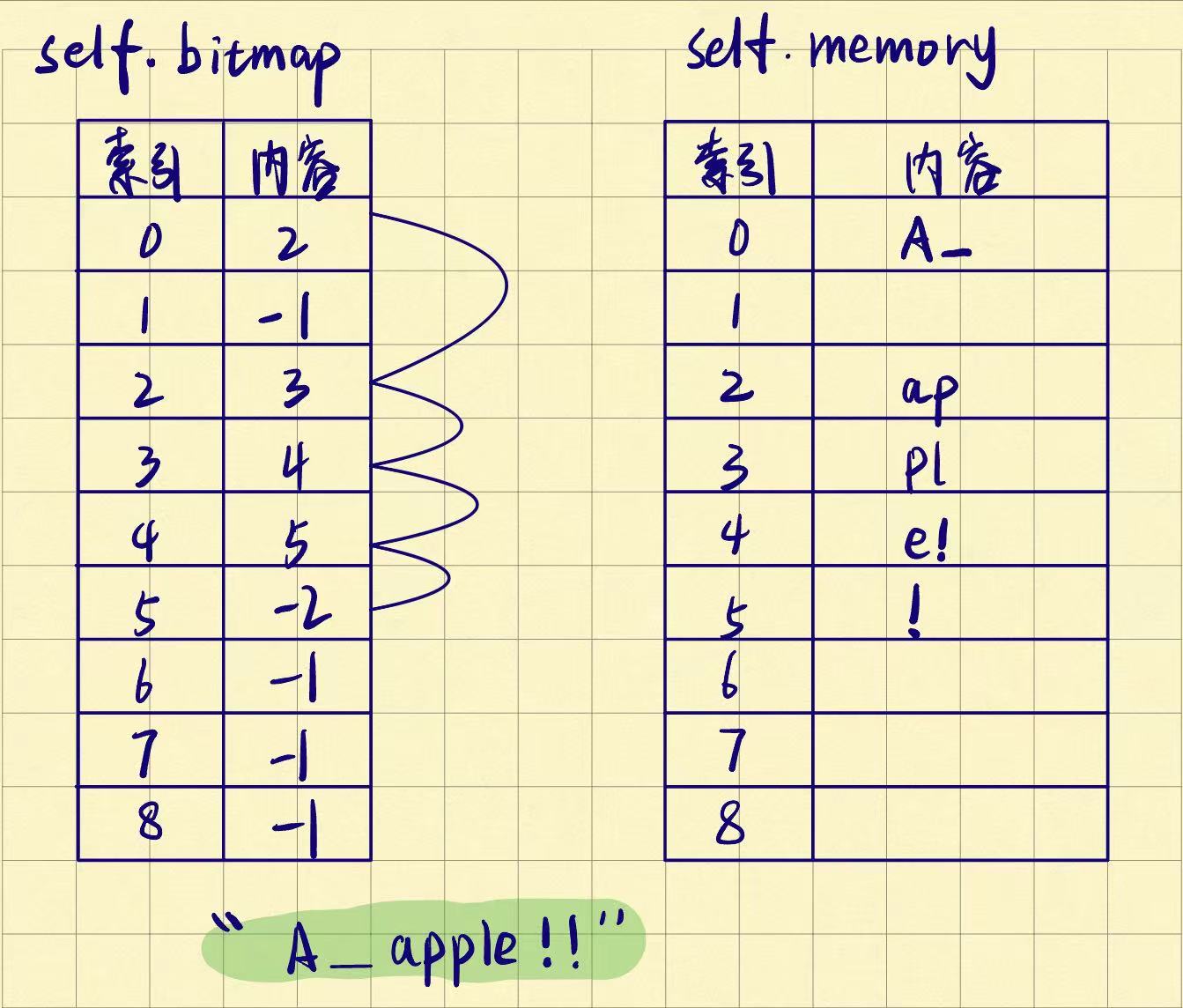
* 磁盘空闲空间管理在位图的基础上进行改造，将存放磁盘上文件位置信息的FAT表与传统的位图进行结合， 磁盘空闲的位置使用EMPTY = -1标识，放有文件的盘块存放文件所在的下一个盘块的位置，文件存放结 束的盘块位置使用END = -2标识。

### 系统实现

#### 为文件内容分配磁盘空间

def give\_space(self, fcb, content):  
 blocks = []  
 index = 0  
 while index < len(content):  
 # 特别处理换行符，保证'\r\n'不被拆分  
 if content[index:index + 2] == '\r\n' and self.block\_size == 2 and index + 2 <= len(content):  
 blocks.append(content[index:index + 2])  
 index += 2  
 elif index + self.block\_size <= len(content):  
 blocks.append(content[index:index + self.block\_size])  
 index += self.block\_size  
 else:  
 # 添加最后一个可能的小于block\_size的块  
 blocks.append(content[index:])  
 break  
  
 if not blocks: # 如果blocks为空（content为空或其他情况）  
 return True   
  
 if len(blocks) <= self.remain:  
 # 找到文件开始存放的位置  
 start = -1  
 for i in range(self.block\_num):  
 if self.bit\_map[i] == self.EMPTY:  
 self.remain -= 1  
 start = i  
 fcb.start = i  
 self.memory[i] = blocks[0]  
 break  
  
 if start == -1: # 如果没有找到空间  
 return False  
  
 # 从该位置往后开始存放内容  
 j = 1  
 i = start + 1  
 while j < len(blocks) and i < self.block\_num:  
 if self.bit\_map[i] == self.EMPTY:  
 self.remain -= 1  
 self.bit\_map[start] = i # 以链接的方式存储每位数据  
 start = i  
 self.memory[i] = blocks[j]  
 j += 1 # 处理下一个块  
 i += 1  
  
 if j == len(blocks):  
 self.bit\_map[start] = self.END # 标记文件尾  
  
 return True  
 else:  
 return False

首先将content进行分割，确保文本按照给定的块大小被分割，同时不会将 \r\n 换行符拆分到不同的块中。然后遍历磁盘，找到第一个空闲的存储块，将它的位置记录到fcb.start中，并往里面写入内容，同时将剩余的存储块数量减一。接着从该位置往后开始存放内容，在位图中以链接的方式存储块之间的联系，位图中的每一位存储下一个块的位置，文件尾用self.END表示。



#### 获取文件内容

def get\_file\_content(self,fcb):  
 if fcb.start == self.EMPTY:  
 return ""  
 else:  
 content = ""  
 start = fcb.start  
 blocks = self.get\_block\_size(fcb.size)  
  
 count = 0  
 i=start  
 while i<self.block\_num and count < blocks:  
 content += self.memory[i]  
 i=self.bit\_map[i]  
 count+=1  
  
 return content

根据位图中存储的信息将不同存储块里的内容拼接在一起。

#### 删除文件内容

def delete\_file\_content(self, start, size):  
 if start == self.EMPTY or start >= self.block\_num:  
 return # If start position is invalid or file is empty, return immediately  
  
 blocks = self.get\_block\_size(size)  
  
 count = 0  
 i = start  
 while i < self.block\_num and count < blocks:  
 next\_index = self.bit\_map[i] # Get next index before clearing  
 self.memory[i] = ""  
 self.bit\_map[i] = self.EMPTY  
 self.remain += 1  
  
 if next\_index == self.END:  
 break # If this was the last block, exit the loop  
  
 i = next\_index  
 count += 1

#### 更新文件内容

def file\_update(self,old\_start,old\_size,new\_fcb,new\_content):  
 self.delete\_file\_content(old\_start,old\_size)  
 return self.give\_space(new\_fcb,new\_content)

#### 检查同目录下是否有同名文件

def check\_same\_name(self, p\_node, name, file\_type):  
 if p\_node is None:  
 return True  
 # 只检查给定节点（即父节点）的直接子节点  
 for child in p\_node.children:  
 if child.fcb.file\_name == name and child.fcb.file\_type == file\_type:  
 return False # 找到一个同名同类型的直接子节点，返回 False  
 return True # 在同级目录中没有找到同名同类型的文件，返回 True

#### 保存文件内容

def save\_content(self):  
 content = self.textEdit.toPlainText()  
 fcb = self.main\_form.category.search(self.main\_form.current\_node, self.filename, FCB.TXTFILE).fcb  
 old\_size = fcb.size  
 new\_size = len(content)  
 current\_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")  
  
 # Update file size and modification time  
 fcb.size = new\_size  
 fcb.last\_modify = current\_time  
  
 # Attempt to update the file content on the disk  
 if not self.main\_form.disk.file\_update(fcb.start, old\_size, fcb, content):  
 QMessageBox.critical(self, 'Error', 'Failed to save file on disk.')  
 else:  
 QMessageBox.information(self, 'Success', 'File saved successfully.')  
  
 # Update the modification time for all parent nodes  
 node = self.main\_form.category.search(self.main\_form.current\_node, self.filename, FCB.TXTFILE)  
 while node.parent:  
 node.parent.fcb.last\_modify = current\_time  
 node = node.parent  
  
 self.main\_form.update\_disk\_info()  
 # Assume a method to update the UI to reflect changes  
 self.main\_form.display\_file\_folder\_info(fcb.file\_name, fcb.last\_modify, fcb.file\_type, fcb.size)  
 #self.main\_form.file\_form\_init(self.main\_form.current\_node) # Refresh the view to show updated times

#### 创建目录树

def setup\_tree(self):  
 # 清除现有的树结构  
 self.tree.clear()  
 # 创建一个递归函数来填充树视图  
 def add\_items(parent\_item, node):  
 # 根据当前节点的信息创建一个新的树项目  
 if node.fcb.file\_type == FCB.FOLDER:  
 item = QTreeWidgetItem(parent\_item, [node.fcb.file\_name + " (Folder)"])  
 else:  
 item = QTreeWidgetItem(parent\_item, [node.fcb.file\_name + " (File)"])  
  
 # 递归地为每个子节点添加树项目  
 for child in node.children:  
 add\_items(item, child)  
  
 # 检查根节点是否存在  
 if self.category.root is not None:  
 # 创建根节点对应的树项目  
 root\_item = QTreeWidgetItem(self.tree, [self.category.root.fcb.file\_name + " (Root)"])  
 self.tree.addTopLevelItem(root\_item)  
  
 # 为根节点的每个子节点添加树项目  
 for child in self.category.root.children:  
 add\_items(root\_item, child)  
  
 # 展开根节点，以便默认显示所有子节点  
 root\_item.setExpanded(True)  
 else:  
 print("No root node is defined in the category.")

#### 磁盘文件的写入和读取

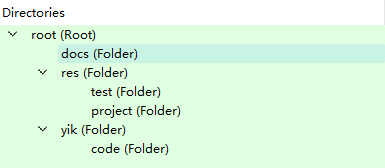
def read\_my\_disk(self):  
 path = os.path.join(os.getcwd(), "MyDiskInfo.txt")  
 if os.path.exists(path):  
 with open(path, 'r', encoding='utf-8') as reader:  
 # 首先读取磁盘的剩余容量信息  
 remain\_line = reader.readline().strip()  
 if remain\_line.startswith("Remaining Blocks:"):  
 self.disk.remain = int(remain\_line.split(":")[1].strip())  
  
 for i in range(self.disk.block\_num):  
 line = reader.readline()  
 #if line == '\n': # 检查是否是空行，只有换行符的行  
 #continue  
  
 # Decode the line, handling all types of newlines  
 line = line.rstrip("\n") # Remove only the newline at the end  
 line = line.replace("|||", "\r\n").replace("|r|", "\r").replace("|n|", "\n")  
  
 self.disk.memory[i] = line  
  
 def write\_my\_disk(self):  
 path = os.path.join(os.getcwd(), "MyDiskInfo.txt")  
 if os.path.exists(path):  
 os.remove(path)  
  
 with open(path, 'w', encoding='utf-8') as writer:  
 # 写入磁盘的剩余容量  
 writer.write(f"Remaining Blocks: {self.disk.remain}\n")  
  
 for data in self.disk.memory:  
 # 输出即将被编码的原始数据  
 print("Original data:", repr(data))  
  
 # Encode the line, handling all types of newlines  
 encoded\_data = data.replace("\r\n", "|||").replace("\r", "|r|").replace("\n", "|n|")  
  
 # 打印编码后的数据以确认转换正确  
 print("Encoded data:", repr(encoded\_data))  
  
 writer.write(encoded\_data + '\n') # 写入转换后的数据加上行分隔符

对特殊字符,"\r\n"，"\r"，"\n"进行特殊编码处理，在读入时使用line = reader.readline()保证读取整行数据，包括换行符，再用line = line.rstrip("\n") 移除行末换行符。

#### 目录文件的写入和读取

def read\_category(self):  
 with open("CategoryInfo.txt", 'r') as file:  
 lines = file.readlines()  
 root\_node = None  
 parent\_stack = []  
 current\_node\_info = {}  
  
 for line in lines:  
 line = line.strip()  
 if "Node Start" in line:  
 current\_node\_info = {}  
 elif "Node End" in line:  
 fcb = FCB(current\_node\_info['File Name'],  
 int(current\_node\_info['File Type']),  
 current\_node\_info['Last Modified'],  
 int(current\_node\_info['File Size']),  
 int(current\_node\_info['Start Position']))  
 new\_node = Category.Node(fcb)  
 if parent\_stack:  
 parent\_stack[-1].add\_child(new\_node)  
 else:  
 root\_node = new\_node # 标记根节点  
 parent\_stack.append(new\_node) # 添加当前节点到栈，用作后续子节点的父节点  
 elif "Parent End" in line and parent\_stack:  
 parent\_stack.pop() # 当一个节点的所有子节点都被处理完毕，从栈中移除该节点  
 else:  
 if line:  
 parts = line.split(": ", 1)  
 if len(parts) == 2:  
 key, value = parts  
 current\_node\_info[key.strip()] = value.strip()  
  
 if not root\_node:  
 default\_fcb = FCB("root", FCB.FOLDER, "", 0)  
 root\_node = Category.Node(default\_fcb)  
  
 self.category.root = root\_node  
 self.root\_node = root\_node  
 self.current\_node = root\_node  
 self.file\_form\_init(self.category.root)  
  
 def write\_category(self):  
 with open("CategoryInfo.txt", 'w') as file:  
 def write\_node(node, parent\_name=""):  
 file.write("Node Start\n")  
 file.write(f"Parent Name: {parent\_name}\n")  
 file.write(f"File Name: {node.fcb.file\_name}\n")  
 file.write(f"File Type: {node.fcb.file\_type}\n")  
 file.write(f"Last Modified: {node.fcb.last\_modify}\n")  
 file.write(f"File Size: {node.fcb.size}\n")  
 file.write(f"Start Position: {node.fcb.start}\n")  
 file.write("Node End\n")  
 for child in node.children:  
 write\_node(child, node.fcb.file\_name)  
 file.write("Parent End\n") # 标记父节点的结束  
  
 if self.category.root:  
 write\_node(self.category.root)

将目录结构和文件信息写入目录文件中，如果目录结构如下：



那么目录文件的内容为：

Node Start  
Parent Name:   
File Name: root  
File Type: 1  
Last Modified: 2024-04-17 23:25:17  
File Size: 0  
Start Position: -1  
Node End  
Node Start  
Parent Name: root  
File Name: docs  
File Type: 1  
Last Modified: 2024-04-17 23:24:37  
File Size: 0  
Start Position: -1  
Node End  
Parent End  
Node Start  
Parent Name: root  
File Name: res  
File Type: 1  
Last Modified: 2024-04-17 23:25:04  
File Size: 0  
Start Position: -1  
Node End  
Node Start  
Parent Name: res  
File Name: test  
File Type: 1  
Last Modified: 2024-04-17 23:24:58  
File Size: 0  
Start Position: -1  
Node End  
Parent End  
Node Start  
Parent Name: res  
File Name: project  
File Type: 1  
Last Modified: 2024-04-17 23:25:04  
File Size: 0  
Start Position: -1  
Node End  
Parent End  
Parent End  
Node Start  
Parent Name: root  
File Name: yik  
File Type: 1  
Last Modified: 2024-04-17 23:25:17  
File Size: 0  
Start Position: -1  
Node End  
Node Start  
Parent Name: yik  
File Name: code  
File Type: 1  
Last Modified: 2024-04-17 23:25:17  
File Size: 0  
Start Position: -1  
Node End  
Parent End  
Parent End  
Parent End

为了方便说明原理，对上面文件内容添加缩进：

Node Start  
Parent Name:   
File Name: root  
File Type: 1  
Last Modified: 2024-04-17 23:25:17  
File Size: 0  
Start Position: -1  
Node End  
  
 Node Start  
 Parent Name: root  
 File Name: docs  
 File Type: 1  
 Last Modified: 2024-04-17 23:24:37  
 File Size: 0  
 Start Position: -1  
 Node End  
 Parent End  
  
 Node Start  
 Parent Name: root  
 File Name: res  
 File Type: 1  
 Last Modified: 2024-04-17 23:25:04  
 File Size: 0  
 Start Position: -1  
 Node End  
  
 Node Start  
 Parent Name: res  
 File Name: test  
 File Type: 1  
 Last Modified: 2024-04-17 23:24:58  
 File Size: 0  
 Start Position: -1  
 Node End  
 Parent End  
  
 Node Start  
 Parent Name: res  
 File Name: project  
 File Type: 1  
 Last Modified: 2024-04-17 23:25:04  
 File Size: 0  
 Start Position: -1  
 Node End  
 Parent End  
 Parent End  
  
 Node Start  
 Parent Name: root  
 File Name: yik  
 File Type: 1  
 Last Modified: 2024-04-17 23:25:17  
 File Size: 0  
 Start Position: -1  
 Node End  
  
 Node Start  
 Parent Name: yik  
 File Name: code  
 File Type: 1  
 Last Modified: 2024-04-17 23:25:17  
 File Size: 0  
 Start Position: -1  
 Node End  
 Parent End  
 Parent End  
Parent End

可以看出，该文件具有以下特点：

* 每个节点以Node Start标记开始，以Node End标记结束；
* 当一个节点的子节点全部被列出时，输出Parent End；

这些特点也可以通过write\_category(self)函数获得；

接下来说明函数read\_category(self)的原理：

* 首先读入目录文件中的所有数据；
* 逐行处理读入的数据：
  + 如果该行包括"Node Start"，将current\_node\_info清空，准备读入新节点的数据；
  + 如果是节点的其它信息，则对数据进行分割，获得key和value；
  + 如果该行包括"Node End"，说明一个节点的信息已经全部读入，创建该节点，并把它作为子节点添加到当前的父节点（栈顶元素）下，如果此时栈为空，说明该节点就是根节点。最后，把该节点入栈，作为后续节点的根节点；
  + 如果该行包括"Parent End"，说明当前父节点的所有子节点都已经处理完，弹出栈顶元素；
* 如果没有根节点信息，创建一个默认的根节点；
* 设置目录的根节点。