

程序设计综合课程设计 报告

完成日期: 2024 年 3 月 14 日

课程报告成绩记载单				
评分项		满分	得分	备注
报告内容 (100 分)	问题描述与分析	10		
	程序总体设计	15		
	数据结构	15		
	算法设计和理论分析	15		
	测试计划及测试分析	15		
	复杂度分析	10		
	总结（思政）	5		
	总结（思维、创新）	5		
	总结（体会、遇到问题及解决方法）	10		
报告规范 (100 分)	文档结构是否完整	40		
	文字描述是否清晰	30		
	文档格式是否规范	30		
教师签名：日期：				

综合成绩=功能实现×50%+设计特色+代码规范+报告内容×25%+报告规范×5%



目 录

一、系统需求分析	1
二、总体设计	2
三、数据结构设计	4
四、详细设计	7
五、系统实现	15
六、运行测试与复杂度分析	21
七、总结	32
附录一 参考文献	34
附录二 主要源程序片段	35
附录三 程序使用说明	45





一、系统需求分析

本次课程设计要开发一个系统，实现对磁盘中目录和文件进行扫描、统计和模拟操作，并提供相应的输出结果。具体要求如下：

1. 输入数据的描述：系统需要读取磁盘中的目录和文件，并获取它们的属性信息。需要注意的是，在获取属性信息时要排除一些无实际意义的文件和目录，并找出下一级的文件和子目录。同时，还需要能够获取文件的修改时间，并进行格式转换显示。

2. 输出结果的描述： a) 系统目录中含有子目录的数量，文件的总数量，目录的层数，最长的带全路径的文件名和长度； b) 构造的目录树的深度； c) 指定目录中的文件信息，包括最早时间的文件（含文件名、大小、时间）、最晚时间的文件、文件总数和总的文件大小； d) 模拟文件和目录操作后的文件信息变化； e) 两次统计文件信息与初始的目录文件属性统计信息的变化。

3. 系统应满足的功能要求：

(1) 磁盘文件属性获取： a) 使用 C 语言的库函数来获取目录和文件的属性； b) 去除无实际意义的文件和目录，获取下一级的文件和子目录； c) 统计子目录的数量； d) 统计文件的总数量； e) 统计目录的层数； f) 找出最长的带全路径的文件名和长度；

(2) 生成 SQL 语句文件： a) 生成插入数据库的 SQL 语句； b) 设计数据库表结构，包括目录表和文件表； c) 生成可以导入数据库的多个 SQL 文件； d) 将数据导入到数据库中；

(3) 构造目录树： a) 在内存中构造目录树，使用有效的数据结构，并计算树的深度； b) 采用非递归的层次遍历算法来访问目录树；

(4) 模拟操作与统计： a) 统计指定目录中的文件信息； b) 模拟文件和目录操作，并统计和比较文件信息的变化。

4. 性能要求： 系统需要能够在合理的时间内处理大量的目录和文件，并且能够同时处理多个操作请求。

二、总体设计

根据实验信息，我设计了一个系统来实现对磁盘中目录和文件的扫描、保存到数据库、内存中构造目录结构、模拟文件操作和统计文件属性变化的功能。系统的总体设计如下：

1. **功能模块划分：**系统由以下五个功能模块组成：a) 获取磁盘文件属性：采用非递归遍历的方式调用系统函数获取磁盘目录和文件的信息。b) 文件信息保存到数据库：手工建立数据库表，并使用SQL语句将文件信息保存到数据库中。c) 内存中重构目录结构：在内存中构建目录树结构，保存文件的相关属性信息。d) 模拟文件和目录操作：根据数据文件提供的模拟操作要求，对文件和目录进行模拟操作，修改文件的属性信息。e) 属性变化的统计和检查：统计指定目录中文件的属性信息，并与初始值进行比较，检查文件属性的变化情况。

2. **系统框架图：**在系统的总体设计中，各个功能模块之间存在紧密的关联和交互。磁盘目录文件扫描和查询系统框架图如下图所示：

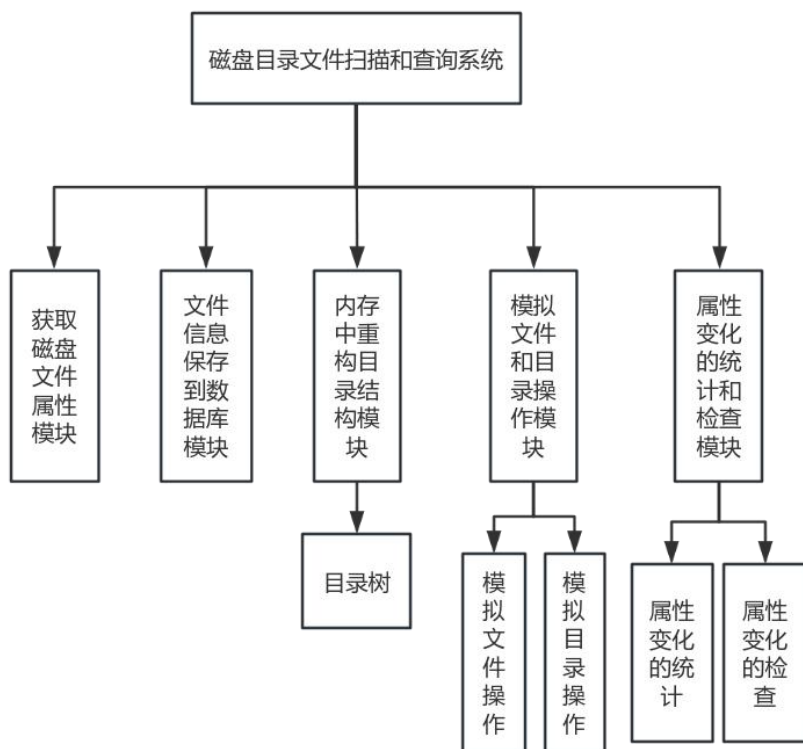


图 2-1 磁盘目录文件扫描和查询系统框架图



3. 各功能模块详细设计:

a) 获取磁盘文件属性模块: 使用非递归遍历的方式调用系统函数获取磁盘目录和文件的信息。需要考虑磁盘中子目录和文件的数量, 确保系统能够支持足够层次的遍历调用。获取文件的属性信息, 包括文件名、时间和字节数大小。

b) 文件信息保存到数据库模块: 手工建立数据库表, 包括目录表和文件表, 用于保存文件信息。将获取的文件信息转化为SQL语句, 通过数据库连接执行插入操作, 将文件信息保存到数据库中。并且将其分割成多个较小的SQL文件进行导入操作。

c) 内存中重构目录结构模块: 在内存中通过孩子兄弟树保存目录和文件的关系和属性信息。使用适当的算法构建目录树, 并计算树的深度。使用非递归的层次遍历访问目录树, 以获取文件信息。

d) 模拟文件和目录操作模块: 根据提供的数据文件中的指定操作, 对内存中的目录树或数据库中的表进行文件/目录模拟操作。如果找不到对应的操作目标, 将对应操作视为无效操作。

e) 属性变化的统计和检查模块: 根据数据文件中的指定目录, 统计目录中文件的属性信息。统计最早和最晚时间的文件, 文件总数和总文件大小。进行模拟操作后, 再次统计文件属性信息, 并与初始属性进行比较, 检查变化情况。



三、数据结构设计

本系统所涉及的数据结构有结构体 FileInfo、DirectoryStack、CFileInfo、TreeNode，哈希表 unordered_map 以及孩子兄弟二叉树等。接下来我将根据数据结构类型分别具体讲解一下我的课程设计中数据结构设计思路。

1. 线性数据结构：

(1) **数组（Array）**：固定大小，元素在内存中连续排列。

FileInfo 结构体：用于保存文件的基本信息，包括文件路径、文件大小和文件修改时间。结构体具体内容如表 3-1 所示。

使用方法：FileInfo 结构体可用于存储在文件遍历过程中收集的每个文件的信息，以方便后续处理和统计。

表 3-1 结构体 FileInfo 数据类型说明

char path[MAX_PATH]	文件路径
ULONGLONG size	文件大小，以字节为单位
FILETIME modifiedTime	文件最后修改时间

CFileInfo 结构体：用于统计文件系统中的文件和目录的数量、最大深度、最长路径等信息。结构体具体内容如表 3-2 所示。

使用方法：在文件和目录的遍历过程中，不断更新 CFileInfo 结构体中的信息，最终得到文件系统的总体统计信息。

表 3-2 结构体 CFileInfo 数据类型说明

int numFiles	文件总数
int numDirectories	目录总数
int maxDepth	目录树的最大深度
char longestPath[MAX_PATH]	最长路径
int longestPathLength	最长路径的长度

(2) **链表（Linked List）**：由节点组成，每个节点包含数据部分和指向下一个节点的指针。

在 unordered_map 存储和查询过程中会存在链表的使用，具体数据结构如图示 3-1。

(3) **栈 (Stack)**：遵循后进先出 (LIFO) 原则的集合。在程序设计中关于栈我使用了两种方式，一种是调用 C++ STL 中的封装的 `stack`，一种是定义了 `DirectoryStack` 结构体。

a) `stack<pair<TreeNode*, string>> stack;`

b) `DirectoryStack` 结构体：用于实现目录的深度优先遍历，存储当前遍历到的目录路径、深度和相关的查找句柄。结构体具体内容如表 3-3 所示。

使用方法：在遍历文件系统时，用 `DirectoryStack` 实现一个堆栈，用于记录当前遍历的目录信息，并在回溯时恢复上一层目录的遍历状态。

表 3-3 结构体 `DirectoryStack` 数据类型说明

<code>char path[MAX_PATH]</code>	当前遍历到的目录路径
<code>int depth</code>	当前目录在目录树中的深度
<code>HANDLE hFind</code>	用于查找目录中文件的句柄

(4) **队列 (Queue)**：遵循先进先出 (FIFO) 原则的集合。

`queue<TreeNode*> q;`

2. 分支数据结构：

二叉树 (Binary Tree)：每个节点最多有两个子节点的树。

`TreeNode` 结构体：用于构建目录树，每个节点代表一个文件或目录，并包含名称、类型、大小、修改时间等信息，以及指向父节点、第一个子节点和下一个兄弟节点的指针。结构体具体内容如表 3-4 所示。

使用方法：使用 `TreeNode` 结构体创建目录树，每次遍历到一个新的文件或目录时，创建一个 `TreeNode* newNode` 节点，并将其插入到合适的位置。通过遍历目录树，可以方便地访问和处理每个文件或目录的信息。

表 3-4 结构体 `TreeNode` 数据类型说明

<code>char name[MAX_PATH]</code>	目录或文件名
<code>int isDirectory</code>	标记是否为目录，1 为目录，0 为文件
<code>long long size</code>	文件大小（仅针对文件），以字节为单位
<code>FILETIME file_time, time_t time</code>	修改时间
<code>struct TreeNode* parent</code>	指向父节点的指针
<code>struct TreeNode* child</code>	指向第一个子节点的指针
<code>struct TreeNode* sibling</code>	指向下一个兄弟节点的指针

3. 散列数据结构:

散列表 (Hash Table): 通过散列函数可以快速访问的数据结构, 也称为哈希表。其数据结构如图 3-1 所示。

```
unordered_map<string, TreeNode*> nodeIndex; // 定义全局哈希表变量
```

使用方法: 在建树的过程中调用 `void addToIndex` 函数将节点添加到哈希表中和在查找的过程中调用 `TreeNode* findNodeByPath` 函数通过哈希表快速寻找路径的节点, 提高寻找节点的效率。

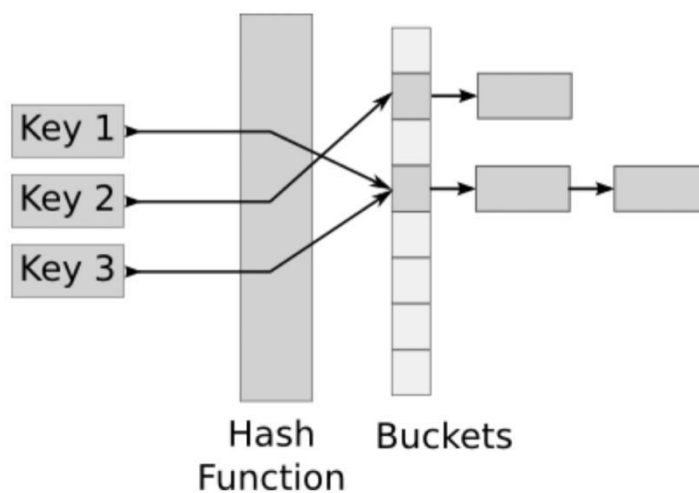


图 3-1 哈希表数据结构图

四、详细设计

1.扫描目录并生成 SQL 文件功能

初始化目录栈 DirectoryStack 和统计信息结构体 CFileInfo 的实例 Cfiles。调用 push 函数将根目录路径压入目录栈 stack，设置栈顶指针 top，并更新 Cfiles 的最大深度。利用 Windows API FindFirstFileA 开始在根目录下搜索文件，并将搜索句柄保存在栈顶元素 hFind 中。进入循环，使用 FindNextFileA 遍历当前目录下的所有文件和目录。对于每个找到的条目：如果是文件，则更新 Cfiles 中的文件数量，检查并更新最长路径，然后调用 insertIntoSQLFile 将文件信息插入相应的 SQL 文件。如果是目录，则更新 Cfiles 中的目录数量，调用 insertDirIntoSQLFile 将目录信息插入 SQL 文件，并把新目录路径压入栈顶，再次使用 FindFirstFileA 在新目录下搜索。如果遍历完当前目录下的所有条目或遇到访问权限问题，则出栈，并尝试继续遍历上一层目录。循环继续，直到栈为空，表示所有文件和目录都已经遍历完成。当达到设置的 SQL 文件最大记录数 MAX_SQL_FILE，关闭当前 SQL 文件并创建新的 SQL 文件以继续写入记录。函数结束时，所有文件和目录的信息已写入 SQL 文件，可用于后续数据库操作。扫描目录并生成 SQL 文件具体算法如图 4-1 所示：

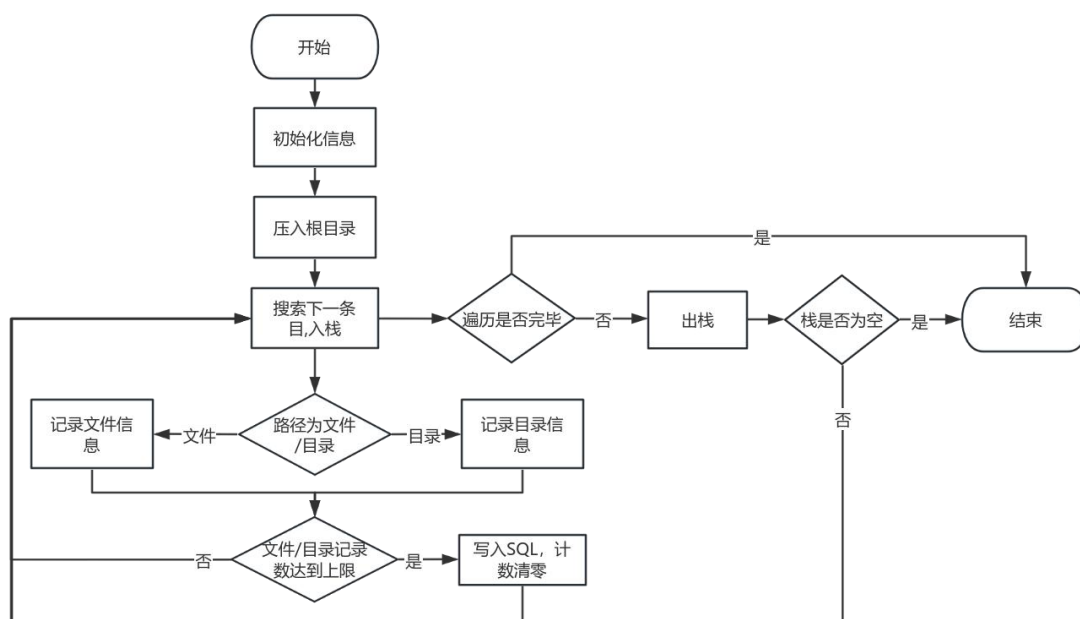


图 4-1 扫描目录并生成 SQL 文件功能流程图

(1) 在默认情况下，Visual Studio 可能会根据系统区域设置或源代码文件的编码来解释源代码中的字符串。如果源代码文件包含 UTF - 8 编码的字符串，而文件没有使用 BOM（字节顺序标记）或编译器没有被正确指示使用 UTF - 8 编码，那么在处理这些字符串时可能会出现乱码或编译错误。

如图 4-2，中文路径文件在 SQL 导入的时候会被识别为特殊字符，无法成功导入，所以需要进行编码转换。

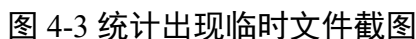


```

1. // 文件路径含有中文，字符需先从 unicode 转换为 UTF-8
2. // 转换为宽字符 (Unicode)
3. int size_needed = MultiByteToWideChar(CP_ACP, 0, newPath, -1, NULL, 0);
4. wchar_t* wpath = new wchar_t[size_needed];
5. MultiByteToWideChar(CP_ACP, 0, newPath, -1, wpath, size_needed);
6.
7. // 转换为 UTF-8
8. size_needed = WideCharToMultiByte(CP_UTF8, 0, wpath, -1, NULL, 0, NU
LL, NULL);
9. char* utf8path = new char[size_needed];
10. WideCharToMultiByte(CP_UTF8, 0, wpath, -1, utf8path, size_needed, NULL,
NULL);

```

(2) 在 Windows 系统中，文件名以"~\$"开头的文件通常是由 Microsoft Office 应用程序生成的临时文件。通过 FindFirstFileA 和 FindNextFileA 函数来扫描目录下的文件时，可能会包括以"~\$"开头的临时文件，如图 4-3 所示。



```
1. if (strcmp(findFileData.cFileName, "~$", 2) == 0) continue;
```

2.构建目录树并计算深度功能

使用 FindFirstFileA 开始在给定的路径下搜索文件和目录。如果找到的句柄无效，则输出错误信息并返回。使用一个栈来模拟递归调用过程，这个栈存储着目录树的节点和对应的路径。进入循环，处理栈中的每个元素，直到栈为空。从栈顶取出一个元素，该元素包含一个目录树节点和一个路径字符串。使用 FindFirstFileA 在该路径下搜索文件和目录，执行找到的每个条目的以下步骤：忽略代表当前目录和父目录的"."和".."; 创建新节点 TreeNode 并设置其属性，包括名称、是否为目录、文件大小、修改时间等；将新节点添加到哈希表 nodeIndex 以便快速索引；将新节点添加到父节点的子节点链表中；如果新节点是目录，则构造一个新路径代表该目录下的所有文件和目录，并将该新路径及新节点入栈。循环直到栈为空。构建目录树的算法如图 4-4 所示：

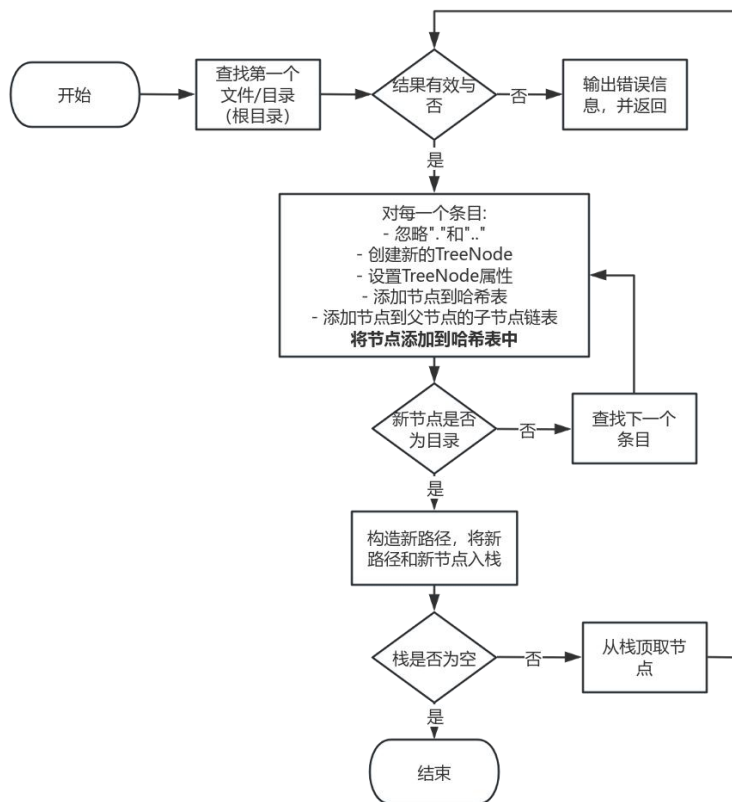


图 4-4 构建目录树流程图

使用一个队列实现 BFS 计算目录树的深度：入队根节点 root。在一个循环中处理队列中的每个节点：对于队列中的每个节点，检查其子节点，如果有子节点则入队；还需要检查每个节点的兄弟节点，并将它们也加入队列；每次内循环完成后，深度增加。当队列为空时，计算结束，返回目录树的深度。

3.统计文件信息功能

开始读取输入文件 `inputFileName`，用于检索要统计的目录路径。创建输出文件 `outputFileName`，用于写入统计结果。读取输入文件的每一行，检查是否开始或结束读取目录列表。对于遇到的每一个目录路径：使用 `findNodeByPath` 函数查找与目录路径相对应的目录树节点；如果目录节点找不到，则在输出文件中记录错误信息；如果目录节点存在，则调用 `ListFiles` 函数以统计文件信息，并将统计结果写入输出文件。整个过程持续进行，直到读取完输入文件中的所有目录路径。关闭输入和输出文件。统计文件信息功能流程如图 4-5 所示：

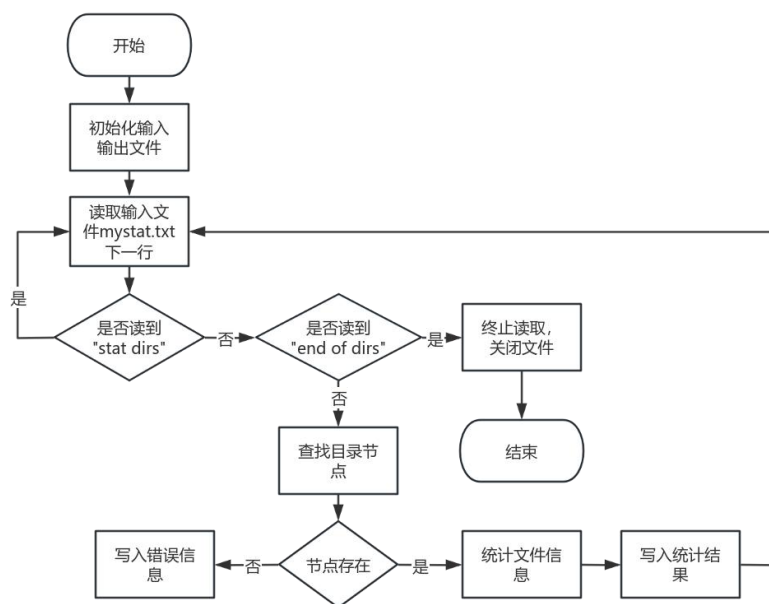


图 4-5 统计文件信息功能流程图

`ListFiles` 函数实现：遍历给定目录节点的所有子节点（文件）；对于每一个文件，更新文件计数和总文件大小，并检查是否需要更新最早或最晚文件时间；结束遍历后，汇总以上信息并写入统计结果到输出文件。

在输出文件中每一目录统计的模块如下：

```
# 1 # c:\windows\Microsoft.NET\Framework64:
总文件数: 4
总文件大小: 114688B
最早文件: c:\windows\Microsoft.NET\Framework64\sbscmp10.dll
最早文件大小: 28672B
最早时间: 2022 年 5 月 7 日 13:20:30
最晚文件: c:\windows\Microsoft.NET\Framework64\sbscmp20_perfcounter.dll
最晚文件大小: 28672B
最晚时间: 2022 年 5 月 7 日 13:20:29
```

如果目录未找到统计的模块如下：

```
# 12 # c:\windows\Microsoft.NET\assembly\GAC_MSIL:  
Error - Unable to open directory
```

4.模拟文件操作功能

打开输入文件 `inputFileName`。读取输入文件的每一行，直到文件结束。检查读取的行是否标志着模拟文件操作的开始 ("selected files") 或结束 ("end of files")。在读取到的模拟文件操作开始和结束标志之间，处理每一行文件操作信息，通过找到的特定标识符解析字符串，以区分行中的目录路径、操作模式、时间和大小。根据解析出的操作模式 `mode` 判断执行哪种操作：删除操作 D：调用 `delete_f` 函数删除对应的文件节点。修改操作 M：调用 `modify_f` 函数更新文件节点的时间和大小信息。添加操作 A：调用 `add_f` 函数在对应的目录下新增文件节点。完成所有操作后，关闭输入文件。模拟文件操作功能流程如图 4-6 所示。

对于每种操作：`delete_f` 函数会调用 `findNodeByPath` 查找文件节点并从目录树中删除，同时更新哈希表 `nodeIndex`。`modify_f` 函数会同样先查找文件节点，然后更新节点的时间和大小信息。`add_f` 函数会查找父目录节点，并添加一个新的文件节点，同时在哈希表 `nodeIndex` 中注册该节点。

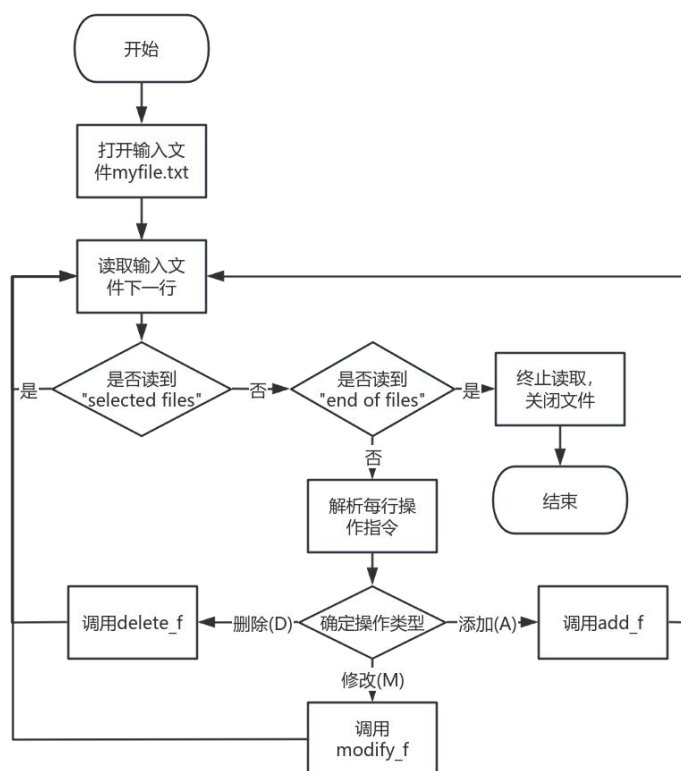


图 4-6 模拟文件操作功能流程图

5.模拟目录操作功能

打开输入文件 `inputFileName`。读取输入文件的每一行，直到文件结束。检查读取的行是否标志着模拟目录操作的开始（"selected dirs"）或结束（"end of dirs"）。在读取到的模拟目录操作开始和结束标志之间，处理每一行目录操作信息：解析字符串以获取目录路径；调用 `delete_d` 函数删除对应的目录节点及其子节点。完成所有操作后，关闭输入文件。模拟目录操作功能流程如图 4-7 所示：

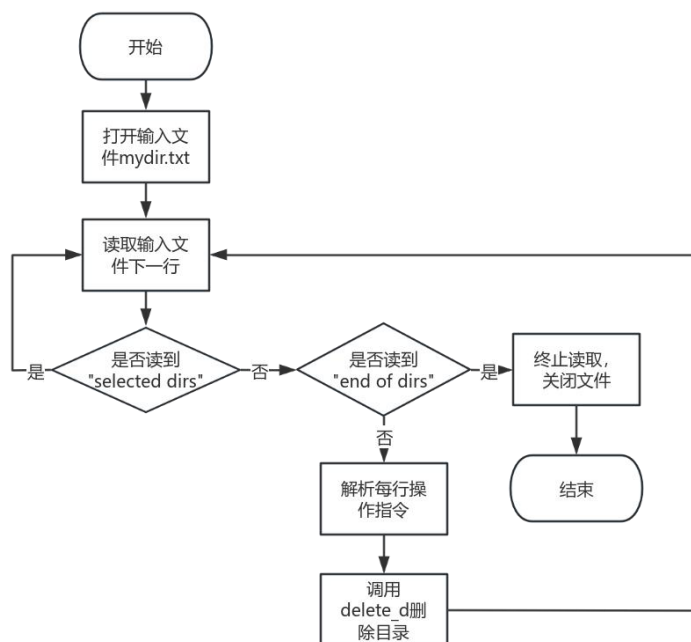


图 4-7 模拟目录操作功能流程图

`delete_d` 函数实现：调用 `findNodeByPath` 查找要删除的目录节点；如果节点存在，调用 `freeDir` 函数递归释放目录节点及其所有子节点；同时在哈希表 `nodeIndex` 中也将节点信息移除。删除节点的算法如图 4-8 所示：

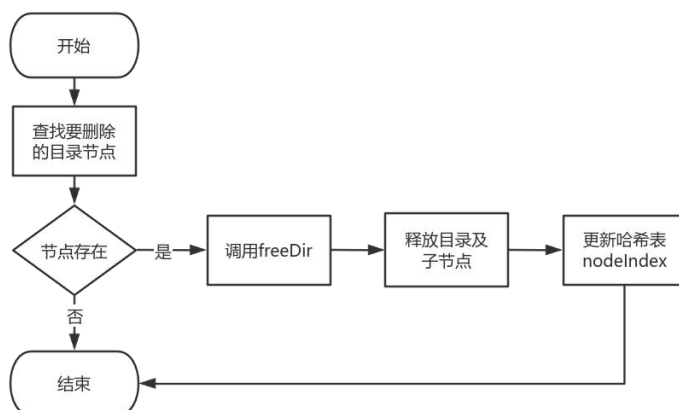


图 4-8 `delete_d` 函数流程图

6.查看统计信息变化功能

打开两个要比较的文件 file1 和 file2。同时读取两个文件的每一行内容。如果当前读取的行以“#”开始，则保存这行内容作为文件路径。如果当前读取的行不为空，则分别追加到 module1 和 module2 字符串中，这两个字符串用于保存两个文件的一个统计模块。当遇到两个文件同时出现空行时，则处理完整的一个统计信息模块：调用 processModuleInfo 函数处理 module1 和 module2；清空 module1 和 module2 以准备下一个统计信息模块的处理。重复步骤，直到读完文件。如果 module1 和 module2 非空，则处理最后一个模块信息。关闭两个文件。查看统计信息变化功能流程如图 4-9 所示：

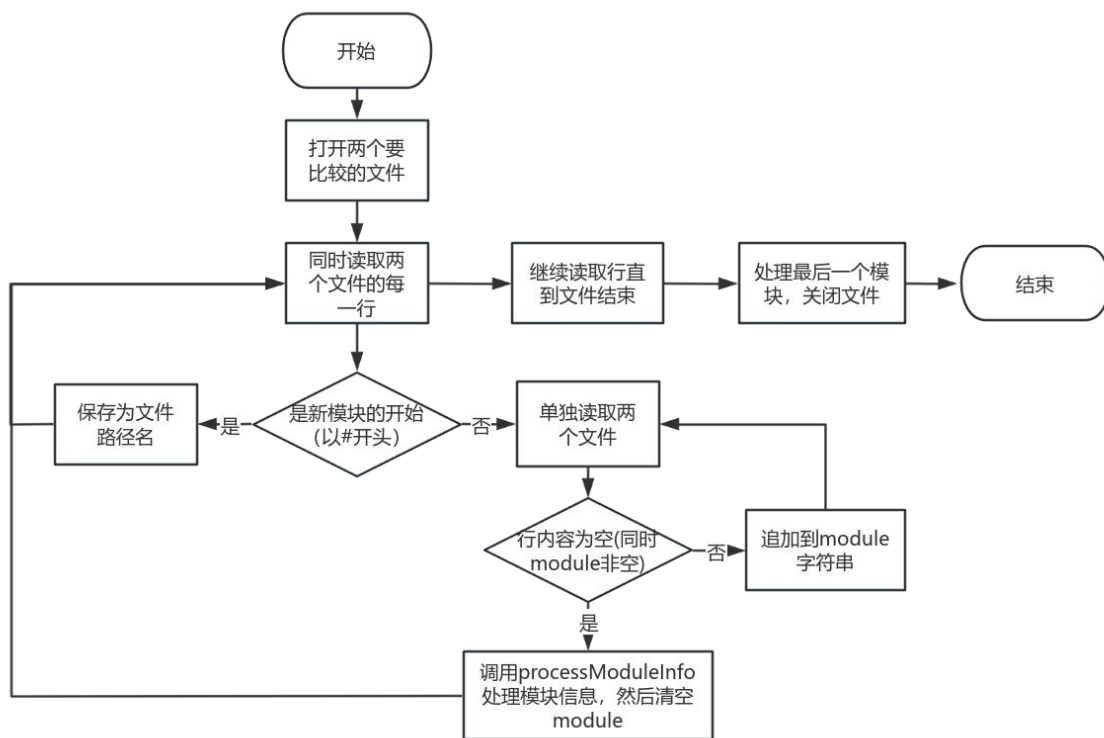


图 4-9 查看统计信息变化功能流程图

processModuleInfo 函数的流程：匹配 module2 中的特定错误信息；如果找到，则输出目标目录已删除的提示；如果未找到错误信息，则使用正则表达式在 module1 匹配所有统计字段；将各统计字段与 module2 中的对应字段进行比较；如果存在任何不匹配项，则输出该统计字段的变化。匹配模块信息的算法如图 4-10 所示：

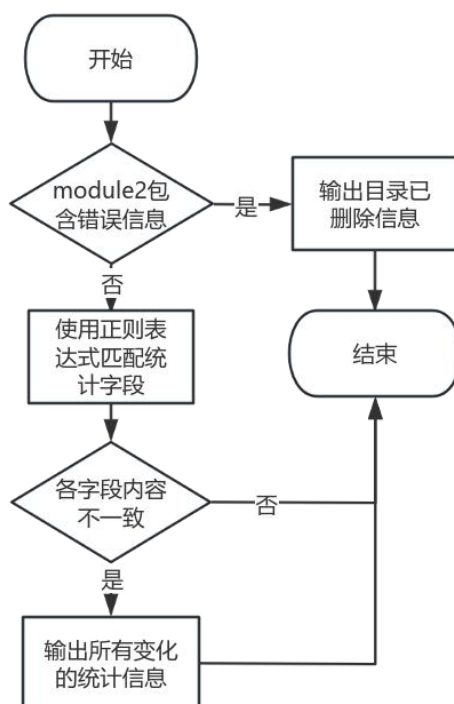


图 4-10 processModuleInfo 函数流程图

7.创新功能 Qt 交互界面——广泛统计目录文件信息

MainWindow 构造函数，创建主窗口，并设置标题和初始大小为 800x600；创建布局管理器，并添加标签、输入框、按钮和文本框；设置字体、固定宽度、拉伸因子等属性；创建菜单栏，并添加功能和关于菜单；创建工具栏并添加快捷键；创建状态栏，并显示版本号和更新时间信息。

onSelectDirectory 槽函数，弹出对话框让用户选择目录；如果用户选择了目录，则在输入框中显示该目录路径；调用 processDirectory 函数处理选定的目录。

processDirectory 函数，清空文本编辑器中的文本；使用 QDir 对象操作给定目录；初始化变量，如总字节数、文件数量、最早和最晚文件时间等；获取目录中的文件列表；遍历文件列表，计算总字节数、文件数量以及最早和最晚的文件时间和信息；构建结果文本，展示在文本编辑器中。

closeEvent 函数，显示关闭应用程序的提示框，询问用户是否关闭应用程序；根据用户选择的结果，决定是否关闭应用程序。

五、系统实现

1.开发环境

处理器：12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz

系统类型：64 位操作系统，基于 x64 的处理器

Windows 版本：Windows 11 家庭中文版

IDE：Visual Studio 2022, Qt5.12.6 , Qt Creator 4.10.2

数据库：MySQL, Navicat

2.支持包

```
1.  #define _CRT_SECURE_NO_WARNINGS
2.  #include <windows.h>
3.  #include <stdio.h>
4.  #include <stdbool.h>
5.  #include <stdlib.h>
6.  #include <string.h>
7.  #include <iostream>
8.  #include <stack>      // 添加栈所需的头文件
9.  #include <queue>      // 添加队列所需的头文件
10. #include <algorithm>
11. #include <fstream>
12. #include <string>
13. #include <regex>      // 正则表达式使用
14. #include <unordered_map> // 添加哈希表所需的头文件
15. #include <map>         // 添加 map 所需的头文件
16. #include <chrono>      // 用来测试运行时间
17.
18. #define MAX_DEPTH 100
19. #define MAX_STACK 20000
20. #define MAX_SQL_FILE 10000
21. #define LONG_TIME 2147483647
22.
23. using namespace std;
```

3.主要函数原型与功能及调用关系

(1) 功能 1：扫描目录并生成 SQL 文件

a) void listCFiles(const char* path, CFileInfo* Cfiles, ofstream& fileSqlFile, ofstream& dirSqlFile);

功能：遍历指定路径下的所有文件和目录，收集文件信息并生成 SQL 文件用于数据库导入。

入口参数：要扫描的目录路径（path），用于收集统计信息的结构体（Cfiles），文件信息 SQL 脚本输出文件流（fileSqlFile），目录信息 SQL 脚本输出文件流（dirSqlFile）。出口参数：无。

调用关系：listCFiles 是功能 1 中的核心函数，一般由主函数直接调用以启动目录扫描和 SQL 文件生成的流程。

b) void insertIntoSQLFile(char* file_path, uintmax_t file_size, time_t file_time, ofstream& sqlFile);

功能：生成单个文件信息的 SQL 插入语句并写入文件。

入口参数：文件路径字符串（file_path），文件大小，以字节为单位（file_size），文件的最后修改时间，UNIX 时间戳格式（file_time），输出 SQL 脚本的文件流（sqlFile）。出口参数：无。

调用关系：在 listCFiles 函数中调用，为每个文件生成 SQL 插入语句并写入到 SQL 脚本文件中。

c) void insertDirIntoSQLFile(const string& directory, const string& subdirPath, ofstream& dirSqlFile);

功能：生成单个目录信息的 SQL 插入语句并写入文件。

入口参数：扫描的根目录路径字符串（directory），相对于根目录的子目录路径字符串（subdirPath），输出 SQL 脚本的文件流（dirSqlFile）。出口参数：无。

调用关系：在 listCFiles 函数中调用，为每个目录生成 SQL 插入语句并写入到 SQL 脚本文件中。

(2) 功能 2：构建目录树并计算深度

a) void buildTree(TreeNode* parentNode, const char* path);

功能：递归构建给定路径的目录树。

入口参数：父节点指针（parentNode），目录路径（path）。出口参数：无。

调用关系：调用此函数开始构建目录树。

b) int calculateTreeDepth(TreeNode* root);

功能：计算一棵目录树的最大深度。

入口参数：目录树的根节点（root）。出口参数：返回树的最大深度。

调用关系：目录树构建完成后，调用此函数计算树的深度。

c) `TreeNode* findNodeByPath(const string& path);`

功能：通过指定路径查找对应的目录树节点。

入口参数：待查找节点的路径字符串（path）。出口参数：返回一个指向找到的节点的指针；如果未找到，则返回 `nullptr`。

调用关系：在 `delete_d`、`modify_f`、`add_f` 等函数中调用来定位需要处理的节点。

(3) 功能 3：统计文件信息功能

a) `void ListFiles(TreeNode* p, const string& outputFileName);`

功能：列出指定目录下的所有文件，并将统计信息写入文件。

入口参数：目录树节点（p），输出文件路径（outputFileName）。出口参数：无。

调用关系：调用此函数统计特定目录下的文件信息。

b) `void ProcessDirectories(const string& inputFileName, const string& outputFileName);`

功能：处理输入文件中包含的目录，并对每个目录调用 `ListFiles` 函数进行处理。

入口参数：输入文件路径（inputFileName），输出文件路径（outputFileName）。出口参数：无。

调用关系：此函数作为统计文件信息的入口函数。

(4) 功能 4：模拟文件操作功能

a) `void modify_f(string directory, time_t time, long size);`

功能：修改指定文件的时间和大小信息。

入口参数：文件目录路径（directory），文件的新修改时间（time），文件的新大小（size）。出口参数：无。

调用关系：在处理文件修改操作时调用此函数。

b) `void delete_f(string directory);`

功能：删除指定的文件。

入口参数：文件目录路径（`directory`）。出口参数：无。

调用关系：在处理文件删除操作时调用此函数。

c) `void add_f(string directory_add, string file_name, time_t time, long size);`

功能：在指定目录下添加新文件。

入口参数：目录路径（`directory_add`），新文件名称（`file_name`），新文件的修改时间（`time`），新文件的大小（`size`）。出口参数：无。

调用关系：在处理文件添加操作时调用此函数。

d) `void ProcessFile(const string& inputFileName);`

功能：读入文件操作指令，并对每条指令调用相应的处理函数。

入口参数：包含文件操作指令的输入文件路径（`inputFileName`）。出口参数：无。

调用关系：该函数是模拟文件操作的主入口函数，独立调用。

(5) 功能 5：模拟目录操作功能

a) `void delete_d(string directory);`

功能：在目录树中删除特定目录及其所有子节点。

入口参数：包含目录路径的字符串（`directory`）。出口参数：无。

调用关系：在处理目录删除操作时被 `ProcessDir` 函数调用。

b) `void freeDir(TreeNode* root);`

功能：递归释放指定目录树节点及其所有子节点。

入口参数：待释放目录树的根节点（`root`）。出口参数：无。

调用关系：在 `delete_d` 函数内部调用来递归释放整个目录树。

c) `void ProcessDir(const string& inputFileName);`

功能：根据输入文件中的目录操作信息，执行删除目录操作。

入口参数：包含目录操作信息的输入文件路径（`inputFileName`）。出口参数：无。

调用关系：这是独立的功能函数，通常直接由主函数或功能测试函数调用。

(6) 功能 6：查看统计信息变化功能



a) `void processModuleInfo(const string& module1, const string& module2, const string& name);`

功能：比较并输出两个统计信息模块的变化。

入口参数：原始统计信息模块字符串（`module1`），目标统计信息模块字符串（`module2`），表示文件路径的字符串（`name`）。出口参数：无（函数通过控制台直接输出结果）。

调用关系：该函数被 `compareFiles` 函数调用，以处理两个文件中相同路径的统计信息变化。

b) `void compareFiles(const string& file1, const string& file2);`

功能：打开两个文件，并比较它们的统计信息变化。

入口参数：原始统计信息文件的路径（`file1`），目标统计信息文件的路径（`file2`）。出口参数：无（函数通过控制台直接输出比较结果）。

调用关系：这是独立的功能函数，通常直接由主函数或功能测试函数调用。

综上所述，在 `main` 函数中根据输入依次调用上述六个功能的实现函数，并且相关函数相互有效调用层次清晰严密，最终正确实现系统预期。

(7) 创新功能 Qt 交互界面

a) `MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent)`

功能：构造主窗口，设置窗口标题、大小，创建界面组件并添加到布局中，创建菜单栏、工具栏和状态栏，启动定时器更新状态信息。

入口参数： `QWidget` 指针 `parent`。出口参数：无。

b) `void MainWindow::onSelectDirectory()`

功能：当选择目录按钮被点击时调用，弹出文件选择对话框获取用户选择的目录，并处理选定的目录。

入口参数：无。出口参数：无。

c) `void MainWindow::processDirectory(const QString& directory)`

功能：处理给定目录的函数，计算目录中文件的数量、总字节数以及最早和最晚的文件时间和信息，并在文本编辑器中显示处理结果。

入口参数：目录路径（`directory`）。出口参数：无。

d) `void MainWindow::closeEvent(QCloseEvent* event)`



功能： 关闭窗口事件处理函数，在关闭窗口时询问用户是否确认关闭应用程序，根据用户选择来决定是否关闭应用程序。

入口参数： `QCloseEvent* event`。出口参数： 无。

这些函数之间的调用关系为：

`MainWindow::MainWindow` 中连接了选择目录按钮的点击事件，点击后调用 `onSelectDirectory` 函数。

`onSelectDirectory` 函数在用户选择了目录后会调用 `processDirectory` 函数来处理选定的目录。

`MainWindow::MainWindow` 中还连接了关于菜单中使用说明选项的触发事件，点击后会弹出一个包含使用说明的对话框。

`MainWindow::closeEvent` 函数会在关闭窗口时被调用，根据用户的选择来确定是否关闭应用程序。

综上所述，整体结构是 `MainWindow` 构造主窗口界面，处理目录选择和显示结果；同时处理菜单和工具栏的功能，以及关闭窗口事件。

六、运行测试与复杂度分析

1.主要功能运行测试

主菜单的界面如图 6-1，分为六个功能，分别为扫描目录并生成 SQL 文件、构建目录树并计算深度、统计文件信息、模拟文件操作、模拟目录操作和查看统计信息变化。可通过不同的输入选择指定功能进行操作。

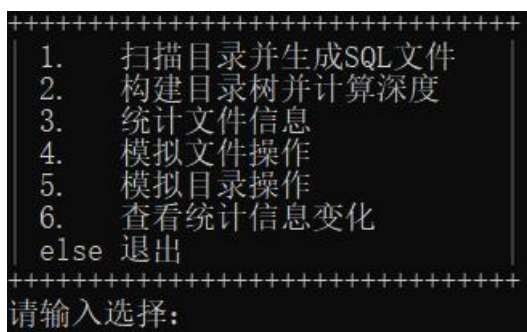


图 6-1 主菜单界面截图

(1) 主要功能 1 磁盘文件属性获取

输入 1 来进行测试，运行结果如图 6-2。



图 6-2 主要功能 1 运行结果截图

由截图可知，程序正确输出了系统目录中含有的子目录数量，含有的文件总数量，目录的层数/深度以及最长的带全路径的文件名及长度，实现了主要功能 1，并且运行过程中完成了生成 SQL 语句文件。

在 sql_dir 文件夹中生成了 6 个“dir_num”格式的 SQL 文件，而在 sql_file 文件夹中生成了 14 个“file_num”格式的 SQL 文件，并通过记事本打开 SQL 文件查看内容，随机选取其中部分截图展示如下图 6-3。

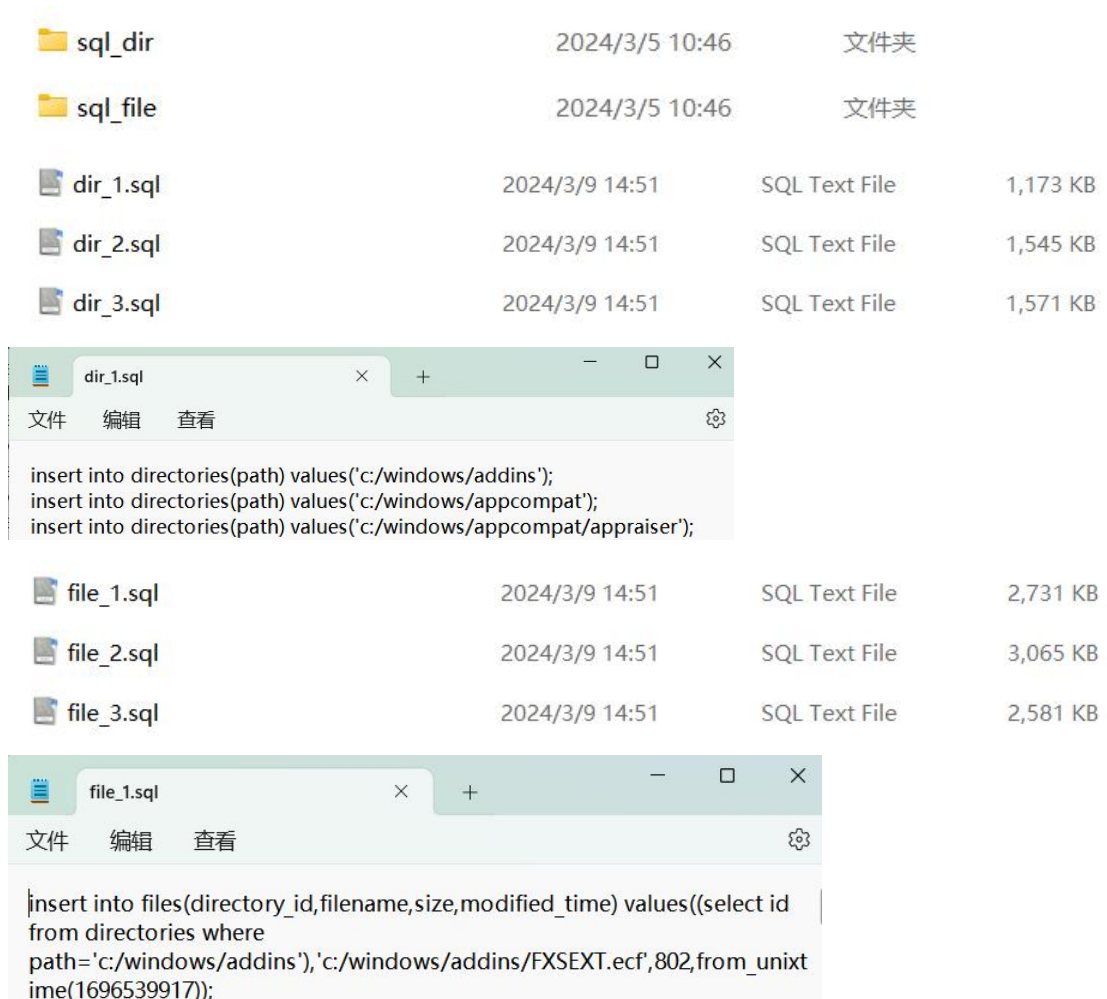


图 6-3 生成 SQL 语句文件截图(部分)

(2) 主要功能 2 生成 SQL 语句文件

功能 1 成功生成了可以插入目录和文件的 SQL 文件，数据库的表结构如

图 6-4:

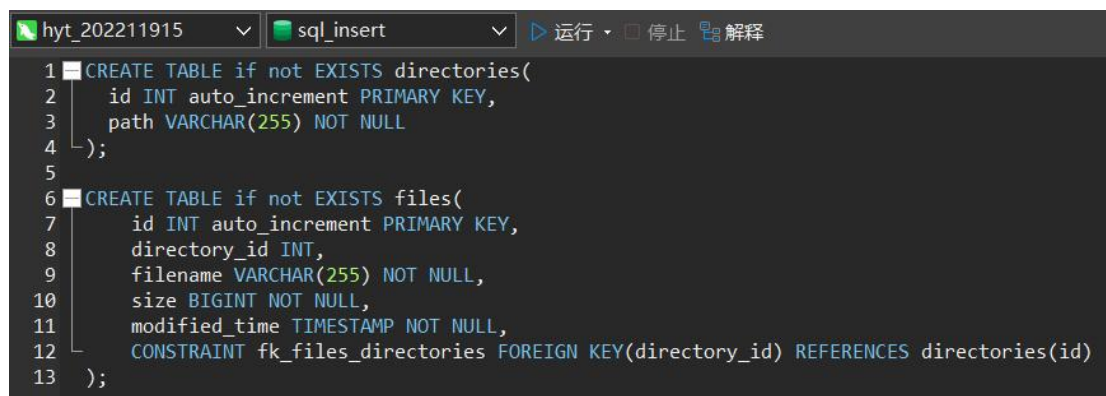
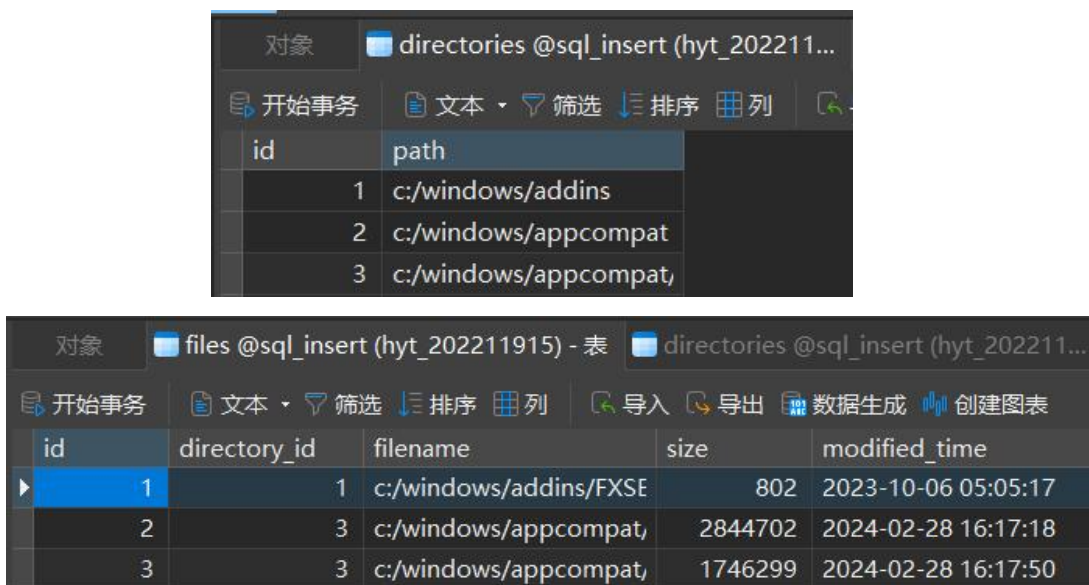


图 6-4 数据库的表结构截图

在 Navicat 中导入 SQL 文件，将目录和文件数据导入到数据库中如图 6-5:



The figure shows two screenshots of the Navicat database management tool. The top screenshot shows a table named 'directories' with two columns: 'id' and 'path'. It contains three rows of data. The bottom screenshot shows a table named 'files' with five columns: 'id', 'directory_id', 'filename', 'size', and 'modified_time'. It contains three rows of data.

id	path
1	c:/windows/addins
2	c:/windows/appcompat
3	c:/windows/appcompat

id	directory_id	filename	size	modified_time
1	1	c:/windows/addins/FXSE	802	2023-10-06 05:05:17
2	3	c:/windows/appcompat	2844702	2024-02-28 16:17:18
3	3	c:/windows/appcompat	1746299	2024-02-28 16:17:50

图 6-5 数据库数据结构截图(部分)

(3) 主要功能 3 构造目录树

接下来我们输入 2 来进行测试，正确计算出树的深度，运行结果如图 6-6。

```
2
# 构建目录树成功! #
目录树的深度: 35464
请按任意键继续. . .
```

图 6-6 主要功能 3 运行结果截图

(4) 主要功能 4 统计文件信息。

a) 输入 3 完成第一次统计文件信息，统计信息被保存在 output1 中，如图 6-7:

```
3
# 第一次统计文件信息完成 #
```

output1.txt 2024/3/9 16:33 文本文档 94 KB

图 6-7 第一次统计文件信息运行结果截图

b) 输入 4 进行模拟文件操作，并展示 3 条操作后的文件信息变化，于此同时进行第二次统计文件信息，统计信息被保存在 output2 中，如图 6-8。

```
4
查看第二次统计信息和差异（任选3条）：
操作前文件信息
c:\windows\Setup\State\State.ini信息为：
时间：2023年10月06日 05:18:30 大小：42B

c:\windows\SystemApps\MicrosoftWindows.Client.CBS_cw5nlh2txyewy\WebExperienceHost.dll信息为：
时间：2024年02月14日 04:23:44 大小：1850368B

未找到c:\windows\SystemApps\Microsoft.ECApp_8wekyb3d8bbwe\GazeInteraction.dll12401
# 模拟文件操作完成 #

操作后文件信息
c:\windows\Setup\State\State.ini信息为：
时间：2024年02月29日 16:23:39 大小：42B

未找到c:\windows\SystemApps\MicrosoftWindows.Client.CBS_cw5nlh2txyewy\WebExperienceHost.dll
c:\windows\SystemApps\Microsoft.ECApp_8wekyb3d8bbwe\GazeInteraction.dll12401信息为：
时间：2024年02月29日 16:23:39 大小：1007616B
# 第二次统计文件信息完成 #
```

output2.txt 2024/3/9 16:38 文本文档 94 KB

图 6-8 模拟文件操作运行结果截图

由截图 6-8 可知，第一个路径文件时间被修改(M)，第二个路径文件被删除(D)，第三个路径文件信息被准确添加(A)，三个操作均成功完成。

c) 输入 5 进行模拟目录操作，并展示该目录和下层子目录的文件信息变化，于此同时进行第三次统计文件信息，统计信息被保存在 output3 中，如图 6-9。

```
5
查看第三次统计信息和差异（任选1条）：
选择输出的目录为c:\windows\Microsoft.NET\assembly\GAC_MSIL\
操作前文件信息
c:\windows\Microsoft.NET\assembly\GAC_MSIL\Accessibility\v4.0_4.0.0.0__b03f5f7f11d50a3a\Accessibility.dll信息为：
时间：2022年05月07日 13:20:28 大小：31032B
# 模拟目录操作完成 #

操作后文件信息
未找到c:\windows\Microsoft.NET\assembly\GAC_MSIL\Accessibility\v4.0_4.0.0.0__b03f5f7f11d50a3a\Accessibility.dll
# 第三次统计文件信息完成 #
```

output3.txt 2024/3/9 16:48 文本文档 93 KB

图 6-9 模拟目录操作运行结果截图

由截图 6-9 可知，选择需要删除目录下的一个文件，在模拟目录操作之前先将信息进行输出，然后进行操作后，将文件信息再次输出，可见文件被删除，可知目录被正确删除，故模拟目录操作成功完成。

d) 输入 6 进行统计信息的检查，统计信息可以分为三种，统计文件、统计目录和统计文件和目录。我设计了一个统计信息菜单界面，可以分别对这三种进行选择统计，如图 6-10：

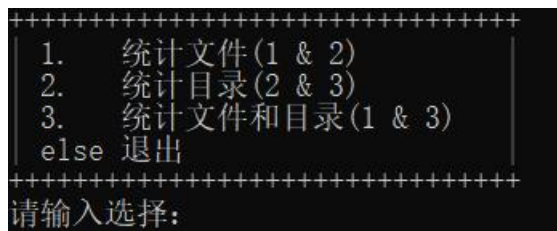


图 6-10 统计信息菜单界面截图

输入 1 检查第一次和第二次统计信息的区别,由图 6-11 呈现内容可以查看具体信息的变化,可以知晓应该是目录下的文件被删除了。

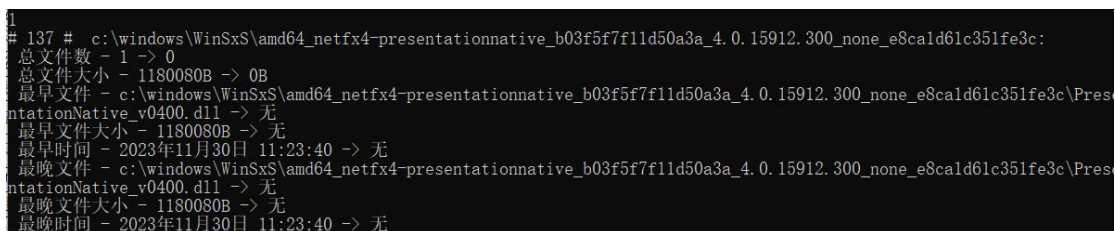


图 6-11 统计文件运行结果截图

输入 2 检查第二次和第三次统计信息的区别,可以清晰地知晓哪些目录被删除并且查看其原信息,便于工程的完备性和可检查性,如图 6-12。

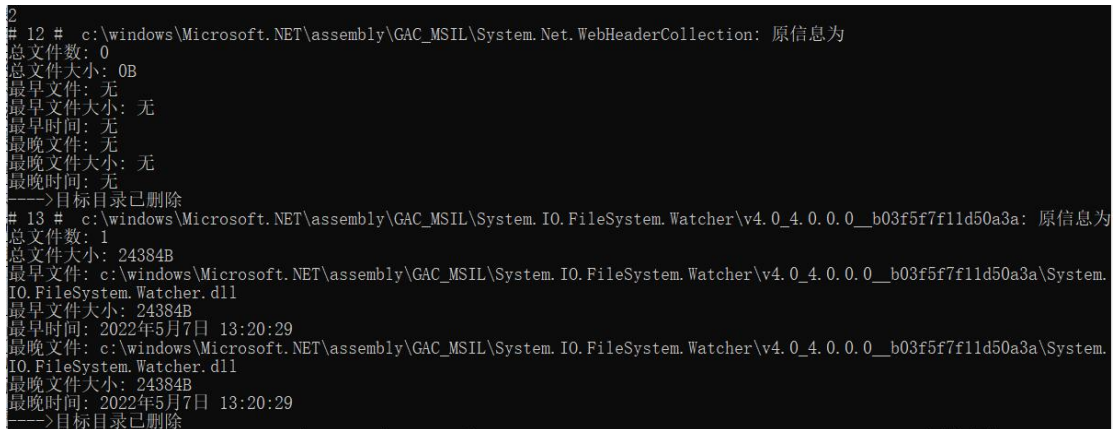


图 6-12 统计目录运行结果截图(部分)

输入 3 检查第一次和第三次统计信息的区别,由截图 6-13 可以看出具体信息的变化,同时展示了文件和目录信息的变化。

```
# 21 # c:\windows\Microsoft.NET\assembly\GAC_MSIL\PresentationFramework-SystemXmlLinq: 原信息为
总文件数: 0
总文件大小: 0B
最早文件: 无
最早文件大小: 无
最早时间: 无
最晚文件: 无
最晚文件大小: 无
最晚时间: 无
---->目标目录已删除
# 137 # c:\windows\WinSxS\amd64_netfx4-presentationnative_b03f5f7f11d50a3a_4.0.15912.300_none_e8cald61c351fe3c:
总文件数 - 1 -> 0
总文件大小 - 1180080B -> 0B
最早文件 - c:\windows\WinSxS\amd64_netfx4-presentationnative_b03f5f7f11d50a3a_4.0.15912.300_none_e8cald61c351fe3c\Prese
ntationNative_v0400.dll -> 无
最早文件大小 - 1180080B -> 无
最早时间 - 2023年11月30日 11:23:40 -> 无
最晚文件 - c:\windows\WinSxS\amd64_netfx4-presentationnative_b03f5f7f11d50a3a_4.0.15912.300_none_e8cald61c351fe3c\Prese
ntationNative_v0400.dll -> 无
最晚文件大小 - 1180080B -> 无
最晚时间 - 2023年11月30日 11:23:40 -> 无
```

图 6-13 统计文件和目录运行结果截图(部分)

综上测试结果可知，程序编译运行正常，无报错提示，而且实现了题目要求的所有功能，运行结果准确。

2.创新功能运行测试

将功能重复再制作成 Qt 界面的功能显得意义不大，只需要将部分代码进行改写成 Qt 语法，添加控件进行分别运行操作并将结果呈现即可，这在之前的计网和数据库实验中已经完成过。而在本次课程设计功能中统计信息操作具有广泛实用性，所以我将其制作成单独的交互界面进行功能的拓展创新。



图 6-14 Qt 人机交互主窗口界面截图

由图 6-14 可知，界面设置了可操作的菜单栏 `QMenuBar`，工具栏 `QToolBar` 以及状态栏 `QStatusBar` 等，而且我对布局进行了严谨地设计，随机拖动窗口大小均可以完美布局，同时工具栏可以进行拖动和位置的修改。菜单栏可查看菜单项 `QMenu`，并有具体动作 `QAction`，为了方便阅读，我在菜单项之间添加了分割线，同时“使用说明”项可以在工具栏使用快捷键启动，状态栏可实时显示时间和版本号信息，菜单栏下菜单项界面如图 6-15 所示。



图 6-15 菜单栏下菜单项界面截图

可以先点击“使用说明”查看相关信息，会弹出一个新窗口，我使用了 `HTML` 标记来创建了丰富的文本内容，并调用了 `setFixedSize` 方法来设置消息框的大小。而且添加了图片使程序更加灵活。如图 6-16 所示。



图 6-16 使用说明界面截图

可以依次点击功能菜单项，功能 1 可以直接在主窗口进行操作，点击“选择目录”按钮控件，选择想统计的目录再点击“选择文件夹”，如图 6-17 所示。

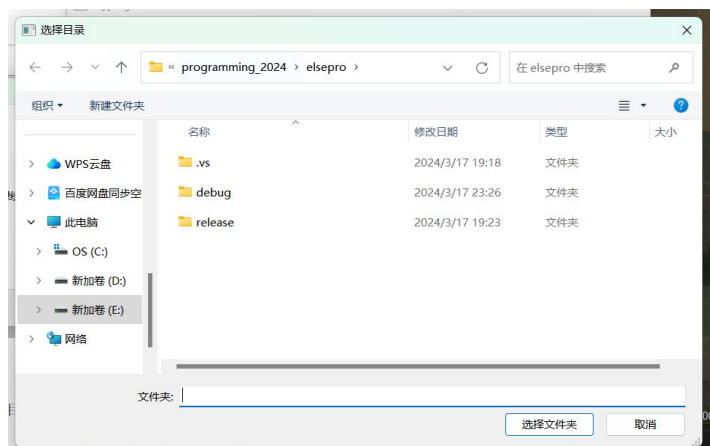


图 6-17 选择文件夹界面截图

然后统计信息会详细清晰的显示在下方的显示框中，同时对目录信息是否有文件的情况进行了区分，确保统计信息的完备性如图 6-18 所示。

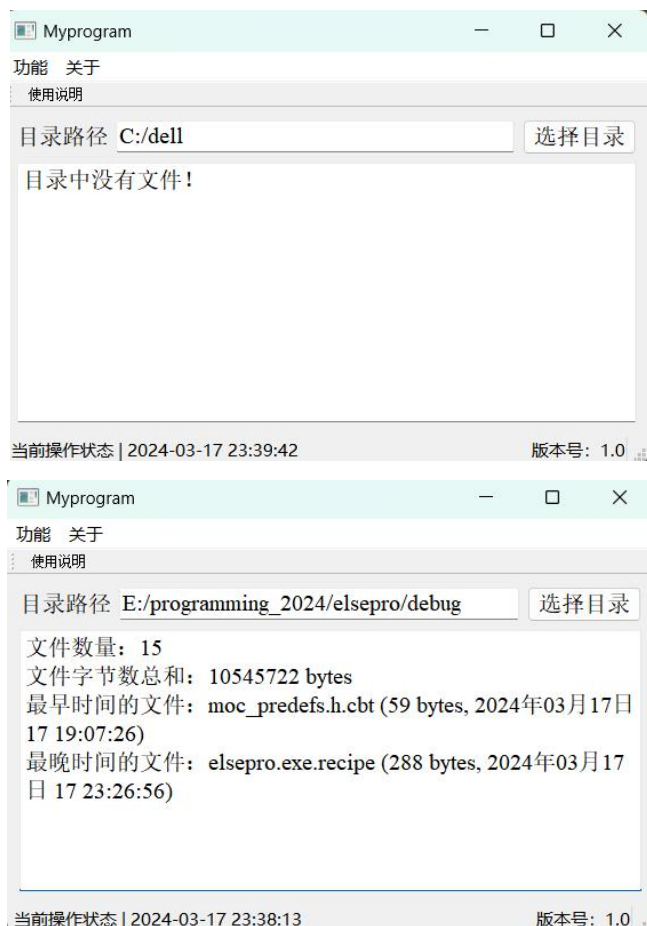


图 6-18 目录信息统计情况界面截图

其他功能项可以进行进一步地开发，我暂时设置为了非模态对话框。最后我重写了 `closeEvent` 方法，在窗口关闭事件发生时弹出一个问题对话框询问用户是

否关闭程序。根据用户的选择，我们要么忽略关闭事件（用户取消关闭操作），要么继续执行关闭操作。确保程序的仿真全面性，如图 6-19 所示。

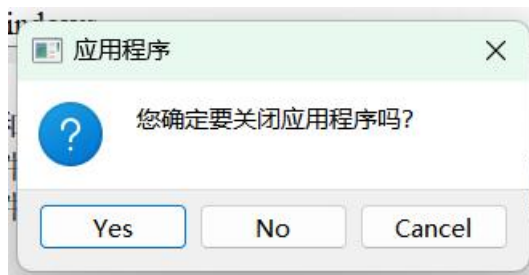


图 6-19 关闭应用程序界面截图

综上测试结果可知，程序编译运行正常，无报错提示，创新功能部分测试完毕，更多的功能可以在后续进一步完善和拓展。

3.复杂度分析与优化

(1) 计算树的深度和目录的层数借助队列实现非递归层次遍历

递归方法的空间复杂度取决于递归调用堆栈的深度 $O(h)$ ，而非递归方法的空间复杂度通常更低。特别是当树的高度很大时，递归方法可能导致堆栈溢出或占用大量内存。相比之下，非递归方法使用队列作为辅助数据结构，空间占用相对较小。

(2) SQL 文件分块和表结构外键的设置

SQL 文件分块的时间复杂度，将一个大文件分割成小文件需要额外的时间成本，包括文件分割、文件读写等操作。但是，这个时间成本相对整体导入的时间成本可能要小很多，因为小文件的导入速度可能更快。而且分块还有以下优势：

a) 避免大文件导入问题：一次性导入 14 万条数据可能会导致数据库性能下降、内存占用增加，甚至可能引发导入失败等问题。分块导入可以避免这些问题，保证数据库的稳定性和性能。

b) 提高导入效率：分块导入可以并行处理多个小文件，从而提高导入的效率。每个小文件可以独立导入，减少整体导入的时间。

c) 易于管理和维护：分块导入可以更好地管理数据的导入过程。每个小文件都是独立的单元，容易跟踪和排查问题，也便于回滚和重试。

同时数据库的表结构设计时，外键的设计作为一个创新点，使表结构更加清晰和完善，但是在导入的时候效率区别较大，如图 6-20，左为未设置外键，右为设置外键。因本程序设计并未使用数据库完成，可以选择不设置运行效率更快。

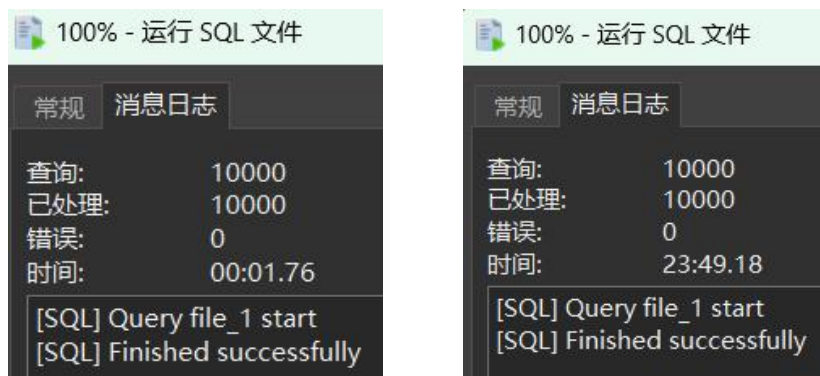


图 6-20 导入 SQL 文件运行时间对比截图

(3) Hash table / map 映射

在程序设计中经常需要寻找指定路径的节点，则需要从头节点开始寻找到指定路径名称的节点再完成操作要求，时间复杂度为 $O(n)$ ，这里可以借助哈希表将路径名称与节点指针建立联系，来实现快速匹配，时间复杂度为 $O(1)$ 。而 map 映射同理，但因为它内部会进行排序使得时间和空间复杂度相对 hashtable 提高，下面仅分析 hashtable 的优化。

如图 6-21 可知，使用哈希表后运行效率得到了极大的提升。



图 6-21 使用哈希表前(左)后(右)运行时间对比截图

(4) 目录树结构体定义 parent 节点

因为删除节点时同样需要通过头节点来寻找目标节点的父节点，时间复杂度为 $O(n)$ ，所以在建立哈希表的基础上，改变目录树的结构，添加父节点，时间复杂度为 $O(1)$ 。但是注意在删除和添加的运算中需要对父节点的指向进行判断和修改，避免父节点错误造成造作失败。

如图 6-22，只有在模拟操作时才涉及到寻找父节点，故可与 6-21 右边两张图进行比较，可知定义父节点有较明显的提速。

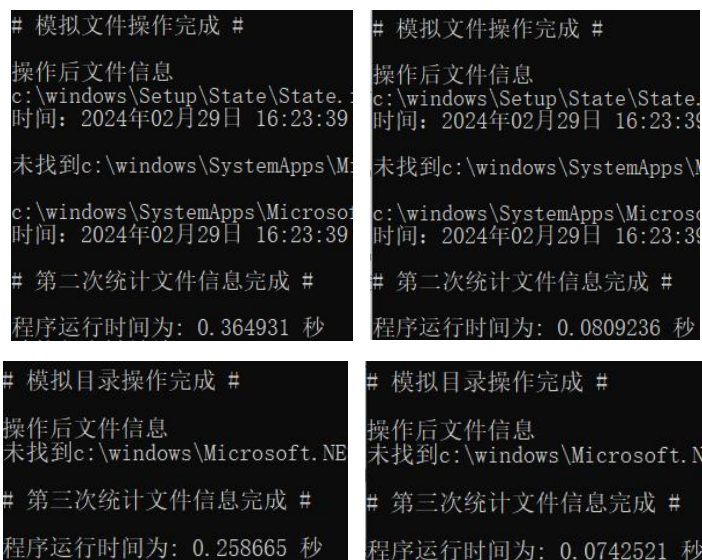


图 6-22 使用 parent 节点前(左)后(右)运行时间截图

(5) 其他一些优化习惯

- 输入输出优先使用 `scanf` 和 `printf`，除开使用 `string` 系列函数时，例如：`s.find`, `s.substr` 等。换行优先使用 `'\n'` 而非 `endl`。
- 充分使用 C++ STL 内置函数，例如：`vector` 动态数组，`set` 集合，`map` 映射，`unordered_map` 哈希表，`list` 链表等。
- 将磁盘扫描和建立 SQL 文件等操作放在一次遍历中，减少了冗余代码量同时提高了运行效率。
- 存储时使用相对地址而非绝对地址，增强代码的可移植性。
- 及时释放无需使用的 `malloc` 和 `new` 的节点的空间，避免空间的无效占用。



七、总结

在完成课程设计的同时，我也了解一些相关技术的发展状况。随着信息技术的不断发展，网络安全问题也日益成为全球关注的焦点。我国作为网络空间安全领域的重要参与者和推动者，需要加强对各类网络攻击和威胁的防范能力。在磁盘目录文件扫描和查询过程中，信息安全问题尤为重要。恶意攻击者可能利用各种手段对磁盘目录文件进行非法访问、篡改或删除，导致数据泄露、系统瘫痪等严重后果。因此，需要了解和熟悉对磁盘目录文件进行扫描和查询的方法和效率，为今后提高文件系统的安全性和可靠性打下基础。

在本次课程设计中，我深入接触并实践了针对磁盘文件系统的扫描、查询以及信息统计等技术。通过实际操作，我更加理解了文件系统结构以及如何使用编程技术来处理和分析这些结构。

在过程中因为急于开始并未清晰地理解任务书的要求，对整个课程设计缺乏整体的构思布局，例如在第四步的时候并未在目录树上操作而是在磁盘上再次扫描，程序设计到扩展功能的时候，察觉到直接对磁盘操作显然是不成立的，就需要推翻前面所有的代码重新开始设计。因为假期忙于科研，课设开始稍晚，当时因为进度相对比较滞后，就有点泄气的情绪，幸好有同学的帮助和鼓励，我及时调整心态，在遇到问题的时候，我与他们进行思路的探讨，然后不断进行调试和修改，一个个战胜它们，并最终顺利完成，极大锻炼了我们的耐心和能力。老师也很耐心负责地回答群内一些普遍性的问题，并且一次次修改程序让我们得以验证我们的运行结果。非常感谢老师的辛勤付出和同学们的友善帮助！

然后我的程序因为权限开的比较多，经常会出现一些特殊的问题，比如在前文中提到的中文编码和临时文件等问题，在过程中我学会一步步调试，并将问题记录下来一点点解决，所以也逐步完善了检验的代码段，不管是树结构的输出还是其他调试的输出，我也均保留了下来，并详细进行了注释。同时为了代码的健壮性，需要多样化考虑情况并进行规范化的预见性处理，也是一步步试错不断修改得以正确指示。科学家精神让我能够理性思考问题，不断进行实验和验证，确保系统的正确性和可靠性，并且勇于尝试新的方法，解决实际遇到的问题。



同时过程中，复习了我所学的编程知识，例如 C++ STL，数据结构和算法，数据库等，先前有相关的竞赛经验对这些有一定的基础，处理起来相对得心应手，但是长期的空窗未使用造成了大量知识的遗忘。程序设计是一个很好的课程方式，有利于进行相对全面地复习和工程代码规范化学习，同时将理论与实践结合，我了解到相关技术背后的原理，并通过编写代码来实际操作这些技术。优化这个环节是我花时间较多的部分，在电脑上都有多个运行程序，不断地进行修改得到不同的版本便于进行比较，这是一种很直观的方式利于我们了解到程序的运行过程和效率。工匠精神则使我注重细节和质量，追求完美的实现方式，并在遇到困难时持续努力，寻找解决问题的方法。

最后，在思维模式上，我也得到了很大的提升：我从理论出发，通过实际编码实践将理论知识转化为能够解决具体问题的工程产品。我不断地思考如何改进现有的解决方案，增强了在面对复杂系统时的设计思维能力。面对软件设计的多方面挑战，我学会了将编程、算法、性能优化等多方面的知识综合起来考虑。我明白了学习不仅是为了掌握现有知识，更是为了创造新的东西。

故在设计方面我也在尝试创新。首先我设计了较友好的人机交互界面，例如，可选择模式，清屏和暂停，清晰的指引语等；其次我用文档将内容保存并进行检查，而不是将内容只保存在内存中，提高工程的健壮性和可检查性；同时我也更改了输出形式，可以便捷且清晰查看多种统计信息的差异变化，而非只是差异数，便于检查和统计；而且我对代码进行了规范化组织模块和注释，提高了其可阅读性，并且做了多种特殊情况的处理以及检查代码段的补充，确保工程的广泛性和可预见性；最后我进行了创新功能的设计，使用 Qt 设计了一个简单的图形人机交互界面完成程序的操作，并在细节方面尽可能做到仿真和具体。

总而言之，这次课程设计不仅提升了我的技术能力，也锻炼了我的问题解决能力和创新能力。网络安全对于磁盘目录文件扫描和查询过程至关重要，因此，我们需要不断关注和提升网络安全技术，加强磁盘目录文件的安全保护，为构建更安全、可靠的文件系统做出贡献。作为网络空间安全专业的学生，我将不断学习和成长，在保护网络安全方面发挥积极的作用。本次课设也暴露了我在代码实践方面的问题，希望自己以后能够多写一些这样的大型程序，提高编程能力，早日成为一名合格的网络工程师。



附录一 参考文献

- [1] 曹计昌, 卢萍, 李开. C 语言与程序设计. 电子工业出版社, 2013
- [2] 严蔚敏等. 数据结构 (C 语言版). 清华大学出版社,
- [3] Larry Nyhoff. ADTs, Data Structures, and Problem Solving with C++. Second Edition, Calvin College, 2005
- [4] 殷立峰. Qt C++ 跨平台图形界面程序设计基础. 清华大学出版社, 2014:192~197
- [5] 严蔚敏等. 数据结构题集 (C 语言版). 清华大学出版社
- [6] 王珊, 萨师煊. 数据库系统概论, 2014.9

附录二 主要源程序片段

1. 数据结构定义部分

```
1. // 定义堆栈结构
2. typedef struct {
3.     char path[MAX_PATH]; // 当前遍历到的目录路径
4.     int depth;           // 当前目录在目录树中的深度
5.     HANDLE hFind;        // 用于查找目录中文件的句柄
6. } DirectoryStack;
7.
8. // 定义磁盘文件的统计信息
9. typedef struct CFileInfo {
10.     int numFiles;        // 文件总数
11.     int numDirectories; // 目录总数
12.     int maxDepth;        // 目录树的最大深度
13.     char longestPath[MAX_PATH]; // 最长路径
14.     int longestPathLength; // 最长路径的长度
15. } CFileInfo;
16.
17. // 文件信息的数据结构定义
18. typedef struct {
19.     char path[MAX_PATH]; // 文件路径
20.     ULONGLONG size;       // 文件大小，以字节为单位
21.     FILETIME modifiedTime; // 文件最后修改时间
22. } FileInfo;
23.
24. // 目录树节点的数据结构定义
25. typedef struct TreeNode {
26.     char name[MAX_PATH]; // 目录或文件名
27.     int isDirectory;      // 标记是否为目录，1 为目录，0 为文件
28.     long long size;       // 文件大小（仅针对文件），以字节为单位
29.     FILETIME file_time;   // 文件最后修改时间
30.     time_t time;          // 时间戳（转换为可读时间格式）
31.     struct TreeNode* parent; // 指向父节点的指针
32.     struct TreeNode* child;  // 指向第一个子节点的指针
33.     struct TreeNode* sibling; // 指向下一个兄弟节点的指针
34. } TreeNode;
35.
36. unordered_map<string, TreeNode*> nodeIndex; // 哈希表定义
37. stack<pair<TreeNode*, string>> stack;      // 堆栈定义
38. queue<TreeNode*> q;                       // 队列定义
```

2. 关键算法

Hash table

```
1.  /**
2.   * @brief 将目录树节点添加到哈希表中
3.   * @param node 要添加到哈希表的目录树节点
4.   * @note 使用节点的名称的哈希值作为键，节点指针作为值，将其添加到
      哈希表中
5.   */
6.   void addToIndex(TreeNode* node) {
7.       // 对节点的名称进行哈希，得到一个唯一的键值
8.       string indexKey = to_string(hash<string>{}(node->name));
9.
10.      // 在哈希表中添加键值对，以便快速查找
11.      nodeIndex[indexKey] = node;
12.
13.  }
14.
15.  /**
16.   * @brief 通过路径在哈希表中查找对应的目录树节点
17.   * @param path 要查找的目录或文件的路径
18.   * @return 查找到的目录树节点指针；如果未找到，返回 nullptr
19.   * @note 此函数用于快速定位指定路径的节点
20.   */
21.  TreeNode* findNodeByPath(const string& path) {
22.      // 对查找路径进行哈希，得到索引键值
23.      string indexKey = to_string(hash<string>{}(path));
24.
25.      // 在哈希表中根据键值查找节点
26.      auto it = nodeIndex.find(indexKey);
27.      if (it != nodeIndex.end()) {
28.          // 如果找到了对应的节点，返回节点指针
29.          return it->second;
30.      }
31.      // 如果没有找到，返回空指针
32.      return nullptr;
33.  }
```

通过 parent 节点模拟文件操作

```
1.  /**
2.   * @brief 修改指定文件的时间和大小属性
3.   * @param directory 文件的目录路径
4.   * @param time 新的时间属性
5.   * @param size 新的文件大小
6.   */
7.   void modify_f(string directory, time_t time, long size) {
```



```
8.     TreeNode* node = findNodeByPath(directory);
9.     if (!node) {
10.         //cerr << "文件不存在，无法修改" << endl;
11.         return ;
12.     }
13.     // 更新节点时间和大小属性
14.     node->time = time;
15.     node->size = size;
16. }
17.
18. /**
19.  * @brief 删除指定的文件节点
20.  * @param directory 文件的目录路径
21.  */
22. void delete_f(string directory) {
23.     TreeNode* node = findNodeByPath(directory);
24.     if (!node) {
25.         //cerr << "文件不存在，无法删除" << endl;
26.         return;
27.     }
28.     TreeNode* parent_node = node->parent;
29.     //TreeNode* parent_node = find_parent(rootNode,node);
30.     // 更新父节点子节点链表，移除当前节点
31.     if (parent_node->child == node) {
32.         parent_node->child = node->sibling;
33.     }
34.     else {
35.         parent_node->sibling = node->sibling;
36.     }
37.     if (node->sibling) node->sibling->parent = parent_node;
38.
39.     //将节点从哈希表中移除
40.     std::string indexKey = std::to_string(std::hash<std::string>{}(node->name))
41.     ;
42.     nodeIndex.erase(indexKey);
43.     // 释放节点内存
44.     delete node;
45. }
46. /**
47.  * @brief 在指定目录下添加新文件节点
48.  * @param directory_add 目录的路径
49.  * @param file_name 新文件的名称
50.  * @param time 文件的时间属性
51.  * @param size 文件的大小
52.  */
53. void add_f(string directory_add, string file_name, time_t time, long size) {
54.     TreeNode* node = findNodeByPath(directory_add);
```

```
55.     if(!node) {
56.         //cerr << "文件目录不存在，无法添加" << endl;
57.         return;
58.     }
59.     // 创建新文件节点并设置属性
60.     TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
61.     newNode->time = time;
62.     strcpy(newNode->name, file_name.c_str());
63.     newNode->isDirectory = 0;
64.     newNode->size = size;
65.     newNode->child = NULL;
66.     newNode->sibling = NULL;
67.
68.     // 将新节点添加到目录下
69.     if (node->child) {
70.         node = node->child;
71.         newNode->sibling = node->sibling;
72.         if (node->sibling) node->sibling->parent = newNode;
73.         node->sibling = newNode;
74.         newNode->parent = node;
75.     }
76.     else {
77.         node->child = newNode;
78.         newNode->parent = node;
79.     }
80.
81.     addToIndex(newNode); // 将新节点添加到索引中
82.
83. }
```

借助队列使用非递归实现层次遍历计算树的深度

```
1.  /**
2.   * @brief 计算目录树的深度。
3.   * @param root 目录树的根节点指针。
4.   * @return 目录树的最大深度。
5.   */
6.   int calculateTreeDepth(TreeNode* root) {
7.       if (root == nullptr) {
8.           return 0;
9.       }
10.
11.       std::queue<TreeNode*> q;
12.       q.push(root);
13.       int depth = 0;
14.
15.       while (!q.empty()) {
16.           int size = q.size(); // 当前层的节点数
17.
```

```
18.     for (int i = 0; i < size; ++i) {
19.         TreeNode* node = q.front();
20.         q.pop();
21.
22.         if (node->child != nullptr) {
23.             q.push(node->child);
24.
25.         }
26.
27.         TreeNode* sibling = node->sibling;
28.         if (sibling != nullptr) {
29.             q.push(sibling);
30.         }
31.
32.     }
33.     ++depth; // 每遍历完一层，深度加一
34. }
35. return depth;
36. }
```

3. 特色功能

out_cparg.cpp (模块化比较文本文件信息)

```
1.  /**
2.   * @brief 比较两个模块的统计信息并打印变化情况
3.   * @param module1 原始模块的统计信息字符串
4.   * @param module2 比较模块的统计信息字符串
5.   * @param name 被比较的文件目录名称
6.   */
7.  void processModuleInfo(const string& module1, const string& module2, const
    string& name) {
8.
9.      if (module2.find("Error - Unable to open directory") != string::npos) {
10.         cout << name << "原信息为\n" << module1 << "---->目标目录已删除
            " << endl;
11.         return;
12.     }
13.
14.     // 使用正则表达式匹配特定格式的内容
15.     smatch match;
16.     regex reg(R"(总文件数: (\d+)\n.*总文件大小: (\d+)B\n.*最早文
        件: (.*?)\n.*最早文件大小: (.*?)\n.*最早时间: (.*?)\n.*最晚文件: (.*?)\n.*最晚文
        件大小: (.*?)\n.*最晚时间: (.*?)\n)");
17.
18.     if (regex_search(module1, match, reg)) {
19.         // 提取各字段信息
20.         string totalFiles1 = match.str(1);
```

```
21.     string totalSize1 = match.str(2);
22.     string earliestFile1 = match.str(3);
23.     string earliestSize1 = match.str(4);
24.     string earliestTime1 = match.str(5);
25.     string latestFile1 = match.str(6);
26.     string latestSize1 = match.str(7);
27.     string latestTime1 = match.str(8);
28.
29.     regex_search(module2, match, reg); // 正则匹配第二个模块
30.     string totalFiles2 = match.str(1);
31.     string totalSize2 = match.str(2);
32.     string earliestFile2 = match.str(3);
33.     string earliestSize2 = match.str(4);
34.     string earliestTime2 = match.str(5);
35.     string latestFile2 = match.str(6);
36.     string latestSize2 = match.str(7);
37.     string latestTime2 = match.str(8);
38.
39.     // 比较各字段，并且输出不同的内容
40.     if (totalFiles1 != totalFiles2 || totalSize1 != totalSize2 || earliestFile1 != ea
rliestFile2 || earliestSize1 != earliestSize2 ||
41.         earliestTime1 != earliestTime2 || latestFile1 != latestFile2 || latestSize1 !
= latestSize2 || latestTime1 != latestTime2) {
42.         cout << name << endl;
43.
44.         // 比较字段内容并输出差异
45.     }
46. }
47. }
48.
49. /**
50.  * @brief 比较两个文件中的统计信息变化
51.  * @param file1 第一个文件的路径
52.  * @param file2 第二个文件的路径
53.  */
54. void compareFiles(const string& file1, const string& file2) {
55.     // 打开两个文件进行比较
56.     ifstream input1(file1);
57.     ifstream input2(file2);
58.
59.     // 检查文件是否成功打开
60.     if (!input1.is_open() || !input2.is_open()) {
61.         //cerr << "Error opening files!" << endl;
62.         return;
63.     }
64.
65.     string line1, line2; // 用于存储每行内容
66.     string name; // 用来保留文件路径
67.     string module1;
```

```
68.     string module2;
69.
70.     while (getline(input1, line1) && getline(input2, line2)) {
71.         if (line1.find("# ") == 0) {
72.             name = line1;
73.             continue;
74.         }
75.         // 将完整的模块信息读取到 module1 和 module2 中
76.         while (!line1.empty()) {
77.             module1 += line1 + "\n";
78.             getline(input1, line1);
79.         }
80.         while (!line2.empty()) {
81.             module2 += line2 + "\n";
82.             getline(input2, line2);
83.         }
84.         // 如果两个文件中同时出现空行，意味着一个模块的结束
85.         if (line1.empty() && line2.empty()) {
86.             // 处理完整的一个模块信息
87.             if (!module1.empty()) {
88.                 processModuleInfo(module1, module2, name);
89.                 module1.clear(); // 清空模块信息，准备下一次比较
90.                 module2.clear();
91.             }
92.         }
93.     }
94.     // 关闭文件流
95.     input1.close();
96.     input2.close();
97. }
```

mainwindow.cpp(Qt 人机交互界面)

```
1.  MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent)
2.  {
3.      // 设置主界面标题
4.      setWindowTitle("Myprogram");
5.
6.      // 设置主界面初始大小为 800x600
7.      resize(800, 600);
8.
9.      QWidget* centralWidget = new QWidget(this);
10.     setCentralWidget(centralWidget);
11.
12.     QGridLayout* layout = new QGridLayout();
13.
14.     QLabel* label = new QLabel("目录路径", this);
15.     QLineEdit* lineEdit = new QLineEdit(this);
16.     QPushButton* button = new QPushButton("选择目录", this);
```

```
17.    textEdit = new QTextEdit(this);
18.    textEdit->setReadOnly(true);
19.
20.    QFont font("Times New Roman", 12); // 设置字体和字号
21.    label->setFont(font);
22.    lineEdit->setFont(font);
23.    button->setFont(font);
24.    textEdit->setFont(font);
25.
26.    label->setFixedWidth(80); // 设置固定宽度
27.    button->setFixedWidth(100); // 设置固定宽度
28.
29.    lineEdit->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed); //
    输入框水平拉伸
30.    textEdit->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    // 文本框水平垂直拉伸
31.
32.
33.    connect(button, &QPushButton::clicked, this, &MainWindow::onSelectDir
    ectory);
34.
35.    layout->addWidget(label, 0, 0);
36.    layout->addWidget(lineEdit, 0, 1);
37.    layout->addWidget(button, 0, 2);
38.    layout->addWidget(textEdit, 1, 0, 1, 3);
39.
40.    centralWidget->setLayout(layout);
41.
42.    // 创建菜单
43.    QMenuBar* menuBar = new QMenuBar(this);
44.    QMenu* functionMenu = menuBar->addMenu("功能");
45.    QAction* statisticAction = functionMenu->addAction("功能 1 统计目录下
    信息");
46.    functionMenu->addSeparator(); // 添加分割线
47.    QAction* otherAction = functionMenu->addAction("其他功能待开发
    ing");
48.
49.    connect(otherAction, &QAction::triggered,
50.        [=]()
51.        {
52.            QDialog dlg;
53.            dlg.setWindowTitle("其他功能"); // 设置窗口标题为“其他功能”
54.            dlg.exec();
55.            //qDebug() << "IIII";
56.        }
57.    );
58.
59.
```

```
60.    QMenu* aboutMenu = menuBar->addMenu("关于");
61.    QAction* infoAction = aboutMenu->addAction("使用说明");
62.    connect(infoAction, &QAction::triggered, this, [=]() {
63.        QMessageBox messageBox;
64.        messageBox.setWindowTitle("使用说明");
65.        messageBox.setIconPixmap(QPixmap("./icon.jpg")); // 设置对话框图标
66.        QString message = "<h3>程序使用说明: </h3>"
67.            "<p>欢迎使用本程序。</p>"
68.            "<p>本程序实现了以下功能: </p>"
69.            "<ul>"
70.            "<li>功能 1: 点击“选择目录”选择想统计的目录, 统计信息会显示"
            "在下方显示框中。</li>"
71.            "<li>功能 2: .....</li>"
72.            "</ul>"
73.            "<p>祝您使用愉快! </p>"
74.            "<p>如有问题, 请联系我们。</p>";
75.        messageBox.setTextFormat(Qt::RichText);
76.        messageBox.setFixedSize(400, 300); // 设置对话框大小
77.        messageBox.setText(message);
78.        messageBox.exec();
79.    });
80.
81.    // 工具栏, 菜单栏的快捷方式
82.    QToolBar* toolBar = addToolBar("使用说明");
83.    // 工具栏添加快捷键
84.    toolBar->addAction(infoAction);
85.
86.    setMenuBar(menuBar);
87.
88.    // 创建状态栏
89.    QStatusBar* statusBar = new QStatusBar(this);
90.    setStatusBar(statusBar);
91.
92.    // 显示应用程序版本号在状态栏右侧
93.    QLabel* versionLabel = new QLabel("版本号: 1.0", this);
94.    statusBar->addPermanentWidget(versionLabel);
95.    statusBar->showMessage("当前操作状态");
96.
97.    // 更新时间和状态信息
98.    QTimer* timer = new QTimer(this);
99.    connect(timer, &QTimer::timeout, [this, statusBar]() {
100.        QString statusMessage = "当前操作状态";
101.        statusBar->showMessage(statusMessage + " | " + QDateTime::currentDate().toString("yyyy-MM-dd hh:mm:ss"));
102.    });
103.    timer->start(1000); // 每秒更新时间和状态信息
104.
```




```
105. // 设置布局管理器的列和行拉伸因子
106. layout->setColumnStretch(0, 1); // 第一列拉伸因子设置为1，让标签和输入框随窗口水平拉伸
107. layout->setColumnStretch(1, 2); // 第二列拉伸因子设置为2，让输入框占据更多空间
108. layout->setColumnStretch(2, 1); // 第三列拉伸因子设置为1，让按钮随窗口水平拉伸
109.
110. layout->setRowStretch(0, 1); // 第一行拉伸因子设置为1，让第一行高度随窗口垂直拉伸
111. layout->setRowStretch(1, 2); // 第二行拉伸因子设置为2，让文本框占据更多空间
112. }
113.
114. // 关闭窗口事件处理函数
115. void MainWindow::closeEvent(QCloseEvent* event)
116. {
117.     // 显示关闭应用程序的提示框
118.     QMessageBox::StandardButton resBtn = QMessageBox::question(this, "应用程序",
119.         tr("您确定要关闭应用程序吗? \n"),
120.         QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
121.         QMessageBox::Yes);
122.     if (resBtn != QMessageBox::Yes) {
123.         // 如果用户不选择关闭，则忽略关闭事件
124.         event->ignore();
125.     }
126.     else {
127.         // 如果用户选择关闭，则继续处理关闭事件
128.         QMainWindow::closeEvent(event);
129.     }
130. }
```



附录三 程序使用说明

1. 安装指南

- (1) 确保您的计算机操作系统为 Windows，因为本程序使用 Windows API。
- (2) 下载程序压缩包到您的计算机。
- (3) 解压缩下载的程序压缩包到指定目录。
- (4) 在解压目录中找到可执行文件（如程序设计 hash.exe），双击运行。

2. 使用前准备

- (1) 确保您有足够的权限访问要扫描的目录和文件。
- (2) 准备一个文本文件（如 mystat.txt），按行列出您要扫描的目录路径。
- (3) 为储存扫描结果准备一个或多个空的文本文件。

3. 程序界面和操作

- (1) 程序启动后，您会看到一个命令行界面。
- (2) 根据界面上的提示输入相关的选项。
- (3) 使用示例：输入 '1' 选择扫描目录并生成 SQL 文件，程序则会输出相关信息。
- (4) 按照提示操作，程序会处理相关指令。

4. 查看和导出结果

- (1) 扫描结束后，结果会直接展示在命令行界面上，并且存储在您预先设定的输出文件中。
- (2) 您可以打开输出文件查看详细的记录，或者将输出文件导入到数据处理软件（如数据库系统）中进一步分析。

5. 软件维护和升级

- (1) 定期检查软件更新，下载新版本以获取更好的性能和功能。
- (2) 遇到任何问题，请查阅随软件提供的帮助文档或联系技术支持。

6. 注意事项

- (1) 在扫描敏感目录或大量文件时，请注意运行权限和电脑性能。
- (2) 使用本软件处理数据时，请备份重要信息，避免数据丢失。
- (3) 具体执行时可查阅“可执行文件”下的“Readme.md”。