

欢迎来到PA-1学习之旅! 🚀

📢 欢迎!

- 👋 欢迎来到**高级程序设计**2025春季课程的PA-1项目!
- 😊 本文档作为实验手册, 将帮助你顺利完成这个PA项目。
- 😎 通过这个PA课程, 你将学习并掌握以下内容:

- 🎯 加深对**面向对象**编程概念的理解。
- 💡 掌握C++类的设计与实现。
- 🔧 学习构建一个完整的软件系统。

⚠️ Warning

PA-1 项目严禁抄袭, 包括**直接复制、抄袭他人代码或未注明引用**。抄袭可能导致零分、学术处分或更严重后果。为避免抄袭, 请独立完成项目, 引用他人成果时注明来源, 并遵守开源许可。

✓ Let's Start Now!

🎉 让我们正式开启**虚拟文件系统**的实验项目编写之旅!

PA-1: 虚拟文件系统

📖 带命令行界面的虚拟文件系统 (VFS) 🌐

在这个项目中, 你将构建一个**虚拟文件系统 (VFS)**, 模拟现实世界中的文件管理。该系统将包括:

- **FileSystem**类: 管理文件和目录。
- **ClientInterface**类: 模拟用户命令行交互。
- **VFS**: 集成并管理整个系统。

0. 背景

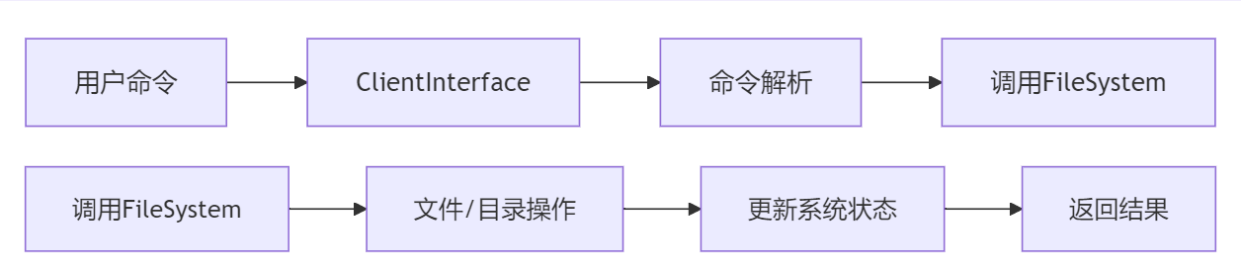
📄 什么是虚拟文件系统 (VFS)?

虚拟文件系统 (VFS) 是一个模拟真实文件系统的软件实现。通过面向对象的设计，VFS 提供了基本的文件和目录管理功能，帮助你理解文件系统的核心概念和面向对象编程原则。

🔗 VFS的工作原理

- 1. **文件抽象**：文件和目录被抽象为类（File、Directory），并使用基类 FileObj 来实现继承和多态。
- 2. **目录组织**：FileSystem 类组织文件和目录，支持灵活的路径导航。
- 3. **用户交互**：ClientInterface 提供了一个命令行界面，支持类Unix风格的文件操作。
- 4. **系统集成**：VFS 集成了 FileSystem 和 ClientInterface，进行统一管理。

📋 系统流程图



1. 核心组件

🔗 关键组件

组件	描述
ClientInterface	处理命令解析和用户交互
FileSystem	实现文件和目录管理
FileObj及其派生类	提供文件和目录的具体实现
VFS	集成所有组件进行统一管理

🔗 项目任务

使用**面向对象编程**来实现这三个核心组件，并构建一个完整的**VFS**系统。

2. 学习目标

📌 你将学习的内容

目标类型	详情
📁 类设计	掌握C++类的设计与实现
🧩 继承与多态	理解并应用继承与多态
🧠 系统构建	使用组合和嵌套类来构建复杂系统
🔒 封装	实现访问控制与封装

3. 预备知识

✍️ 你应该掌握的内容

在开始PA-1之前，请熟悉以下内容：

- **数据结构：**
 - 数组：`vector<T>` 及其方法（`push(T)`、`pop()`）。
 - 字典：`unordered_map<key, value>` 及其增删改查操作。
 - 集合：`set<T>` 及其方法（`find(T)`、`insert(T)`）。
- **数据类型：**`enum`、`size_t`。
- **字符串：**`string`。
- **输入/输出流：**`stringstream`。
- **OOP原则：**继承、多态、封装和抽象。

💡 **小贴士：**在设计时，始终提前思考一个抽象层次！

4. 项目结构

📌 FileSimulator 项目树

```
FileSimulator/
├── CMakeLists.txt
├── include/
│   ├── FileObj.h
│   ├── Directory.h
│   ├── FileSystem.h
│   ├── VFS.h
│   └── ClientInterface.h
├── src/
│   ├── main.cpp
│   ├── FileObj.cpp
│   ├── Directory.cpp
│   ├── FileSystem.cpp
│   ├── VFS.cpp
│   └── ClientInterface.cpp
└── README.md
```

5. 实验指南

5-0. InodeFactory 类

 **InodeFactory 类** (`include/InodeFactory.h`)

```
class InodeFactory {
public:
    static uint64_t generateInode() {
        static uint64_t nextInode = 1;
        return nextInode++;
    }
};
```

- 在VFS中，所有系统对象（`File` 和 `Directory`）都有一个唯一的 `inode` 标识符，类型为 `uint64_t`（无符号长整型）。
- 我们提供了 `generateInode()` API供你使用。默认情况下，根目录 `/` 的 `inode` = 1。
- 任何新创建的文件或目录的 `inode` 值将满足 `inode >= 2`。（注意：VFS不会回收 `inode` 值，因此它们只会递增且不会重复。）

5-1. FileObj (基类)

FileObj 类 (include/FileObj.h)

```
class FileObj {
    string name;      // 文件/目录名称
    string path;      // 绝对路径
    string type;      // "file" 或 "directory"
    string owner;     // 所有者
    uint64_t inode;   // 唯一标识符
    FileObj* parent;  // 父目录
};
```

- 属性：
 - `inode` 由 `InodeFactory` 生成。
 - `parent` 指向父目录（对于根目录 `/`，`parent` 为 `nullptr`）。
- 我们已经实现了 `FileObj` 的构造函数。你可以将其作为其他类构造函数实现的参考。

5-2. File 类

File 类 (include/File.h)

```
class File : public FileObj {
protected:
    string content; // 文件内容
};
```

- `File` 类在 `FileObj` 的基础上增加了 `content` 属性。
- 需要实现的方法：

```
File(const string& name, const string& type, const string& owner,
     const uint64_t& inode, FileObj* parent);

virtual string read() const;           // 读取内容
virtual bool write(const string &data); // 写入内容（追加到 `content`）
virtual string getContent() const;     // 获取内容
```

- 我们已经为你实现了 `File` 类的构造函数，请参考框架代码中的详细TODO。

5-3. Directory 类

① Directory 类 (src/Directory.cpp)

```
class Directory : public FileObj {
    std::unordered_map<uint64_t, FileObj*> children; // 存储子对象
};
```

- 属性：
 - children：一个 unordered_map，用于将 inode 映射到 FileObj* 指针（存储目录中的所有子项）。
- 需要实现的方法：

```
Directory(const string& name, const string& owner,
          const uint64_t& inode, FileObj* parent);

bool add(FileObj* child);           // 添加子节点
bool remove(uint64_t inode);        // 移除文件节点
bool removeDir(uint64_t inode);     // 递归移除目录节点
FileObj* getChild(uint64_t inode);  // 获取特定子节点
std::vector<FileObj*> getAll() const; // 获取所有子节点
size_t getCount() const;           // 获取子节点数量
bool isEmpty() const;              // 检查目录是否为空
```

- 提示：使用 inode 作为移除节点的唯一标识符。请参考框架代码中的详细TODO。

5-4. FileSystem (核心功能)

① FileSystem 类 (include/FileSystem.h)

```
class FileSystem {
    Directory* root;      // 根目录
    Directory* cur;       // 当前目录
    string username;      // 当前用户
    std::set<string> users; // 所有用户
    std::unordered_map<string, uint64_t> config_table; // 路径到inode的映射
};
```

- **属性：**
 - `root`：文件系统的根目录。
 - `cur`：当前目录。
 - `username`：当前用户。
 - `users`：所有注册用户的集合。
 - `config_table`：将对象的**绝对路径 + 类型**映射到其 `inode`。
- **需要实现的方法：**

```
// 目录导航
changeDir(const uint64_t& inode);           // 切换当前目录
getCurrentPath() const;                     // 获取当前路径
resolvePath(const string& path);            // 解析路径

// 文件操作
createFile(const string& name);              // 创建文件
deleteFile(const string& name, const string& user); // 删除文件

// 目录操作
createDir(const string& name);               // 创建目录
deleteDir(const string& name, const string& user, bool recursive);
// 删除目录

// 搜索和用户管理
search(const string& name, const string& type); // 搜索文件或目录
setUser(const string& username);               // 设置当前用户
hasUser(const string& username);               // 检查用户是否存在
registerUser(const string& username);           // 注册新用户
```

- **提示：**
 1. 创建文件/目录时不需要权限控制，但删除时需要权限（只有 `owner` 或 `root` 用户可以删除）。
 2. 在 `changeDir()` 中使用 `resolvePath()`。可以使用 `strtok()` 或 `istringstream` 解析路径。
 3. 添加或删除项目时，同时维护 `children` 和 `config_table`。
 4. 不同用户共享同一个文件系统。使用 `insert()` 或 `remove()` 管理 `users` 集合。

5-5. ClientInterface (用户界面)

① ClientInterface 类 (include/ClientInterface.h)

```
class ClientInterface {
    FileSystem* filesystem; // FileSystem 实例
    string username;       // 当前用户
};
```

- **设计：** ClientInterface 类遵循**访问者模式**。用户通过此接口与文件系统交互，而无需直接嵌入文件系统。
- **需要实现的方法：**

```
// 命令处理
parseCommand(const string& cmdLine); // 解析命令行
execueCommand(const vector<string>& cmd); // 执行命令
processCommand(const string& cmdLine); // 处理命令

// 文件操作
createFile(const string& name); // 创建文件
deleteFile(const string& name); // 删除文件
readFile(const string& name); // 读取文件
writeFile(const string& name, const string& data); // 写入文件

// 目录操作
createDir(const string& name); // 创建目录
deleteDir(const string& name, bool recursive); // 删除目录
changeDir(const string& path); // 切换目录
listCurrentDir(); // 列出目录内容
getCurrentPath() const; // 获取当前路径

// 其他命令
showHelp() const; // 显示帮助信息
search(const string& name, const string& type); // 搜索文件或目录
```

- **命令处理：**
 - parseCommand：将命令行拆分为多个 token。
 - execueCommand：根据第一个 token，(cmd[0]) 执行命令。
 - processCommand：结合 parseCommand 和 execueCommand。

5-6. 需要实现的命令

① 支持的命令


```
# 文件操作
create <文件名...>      # 创建一个或多个文件
delete <文件名...>      # 删除一个或多个文件
read <文件名...>        # 读取一个或多个文件
write <文件名> <文本>    # 写入文件（支持多行文本和转义字符）

# 目录操作
mkdir <目录名>           # 创建目录
rmdir [-r] <目录名>      # 删除目录（-r 表示递归删除）
cd <路径>                # 切换目录（支持相对路径和绝对路径）
ls                       # 列出目录内容
pwd                     # 打印当前工作目录
clear                   # 清空终端

# 系统命令
help                   # 显示帮助信息
exit                  # 注销
quit                  # 退出系统
```

5-7. 注意事项

⚠ 错误处理

- 需要处理的错误包括：
 - 文件已存在
 - 路径不存在
 - 权限不足
 - 无效参数

🔑 用户管理

- 用户创建
- 权限验证
- 会话管理

🔧 文件操作

- 处理空文件

- 递归删除
- 路径解析
- 权限验证

⚠️ 内存管理

- 正确释放对象
- 避免内存泄漏
- 防止悬空指针

🚀 6. 构建与运行

6-1. Windows 环境

📘 Windows 构建与运行

1. 生成项目：

```
# 在项目根目录下运行
mkdir build
cd build
cmake ..
cmake --build .
```

2. 运行程序：

```
# 在 build/bin 或 build/bin/Release 目录下
FileSimulator.exe
```

6-2. Linux 环境

📘 Linux 构建与运行

1. 安装依赖：

```
# Ubuntu/Debian
sudo apt-get install build-essential cmake

# CentOS/RHEL
sudo yum groupinstall "Development Tools"
sudo yum install cmake
```

2. 生成并编译:

```
mkdir build
cd build
cmake ..
make
```

3. 运行程序:

```
# 在 build/bin 目录下
./FileSimulator
```

6-3. 常见问题与故障排除

⚠️ 常见问题

1. CMake 错误:

- 确保 CMake 版本 ≥ 3.10 。
- 验证编译器是否正确安装。
- 确保环境变量设置正确。

2. 编译错误:

- 检查是否缺少依赖项。
- 确保使用了正确的生成器。
- 查看详细的编译日志。

3. 运行时错误:

- 确保从正确的目录运行程序。
 - 检查是否缺少动态库。
 - 查看程序的错误输出。
-

7. 示例代码

≡ 示例用法

```
# Start the system
$ ./FileSimulator
File System Simulator Started
Please login with your username
Login: root

# Show help
root@FileSimulator:/$ help
Available commands:
  create <filename...>      - Create one or more new files
  delete <filename...>      - Delete one or more files
  read <filename...>        - Read content from one or more files
  write <filename> <text>    - Write text to file (supports '\\n' for
newline)
  mkdir <dirname>           - Create a new directory
  rmdir [-r] <dirname>      - Remove directory (-r for recursive
deletion)
  cd <path>                 - Change directory (supports
relative/absolute paths)
  ls                        - List current directory contents
  pwd                       - Show current working directory
  whoami                    - Show current user name
  clear                     - Clear current command line
  help                      - Show this help message
  exit                      - Logout current user
  quit                      - Exit program

# Some Basic operations
root@FileSimulator:/$ mkdir docs
root@FileSimulator:/$ cd docs
root@FileSimulator:/docs$ create readme1 readme2
root@FileSimulator:/docs$ write readme1 "This is a test file."
root@FileSimulator:/docs$ write readme2 "Hello PA-1"
root@FileSimulator:/docs$ read readme1 readme2
=== readm1 ===
This is a test file.
=== readm2 ===
Hello PA-1
root@FileSimulator:/docs$ pwd
/docs
root@FileSimulator:/docs$ ls
readme1
readme2
root@FileSimulator:/docs$ exit
```

```
# exit the program
User Login (Please input who you are):
exit
Bye!
```

温馨提示

- 如您在实验过程中遇到任何问题，请及时联系助教，我们非常乐意解答您的问题。
- 感谢您的阅读，祝您的 PA-1 实验顺利！