Windows操作系统

C/C++ 程序实验

姓名：\_\_\_\_\_\_\_陈展博\_\_\_\_\_\_

学号：\_\_\_\_\_\_\_1221001003\_\_

班级：\_\_\_\_\_\_\_计科1班\_\_\_\_

院系：\_\_\_\_\_\_\_信工\_\_\_\_\_\_\_\_

\_\_\_\_\_\_2024\_\_\_\_\_\_\_\_年\_\_12\_\_\_月\_\_4\_\_\_日

## 实验九 Windows 快速文件系统

### 一、背景知识

### 二、实验目的

### 三、工具/准备工作

### 四、实验内容与步骤

#### 1. 快速文件系统

**步骤1**：登录进入Windows 。

**步骤2**：在“开始”菜单中单击“程序”-“Microsoft Visual Studio Code”。

**步骤3：**新建项目名为“9-1”，并且新建项“9-1.cpp”。

**步骤4：**将项目所需txt文件复制到项目文件夹下。

**步骤5：**按“F5”开始调试，注意路径里不要含有中文。

**步骤6：**按暂停按钮可暂停程序的执行，按终止按钮可终止程序的执行。



操作能否正常进行？如果不行，则可能的原因是什么？

操作能够正常进行，如果不能正常进行，可能的原因是因为文件路径中包含中文，或者代码中的字符集有错误。

运行结果是：

文本

描述已自动生成

图片 1 运行结果截图

结果分析：

使用文件高速缓存模式对文件进行操作的平均用时会比使用无模式高速缓存模式和异步传输模式的平均用时要快很多。这里异步传输模式和无高速缓存模式的速度差不多是因为这里主要调用文件操作的函数，若能增加更多与文件操作无关的操作，异步传输模式会更有优势。

**选作：**探究如何设计程序可以更好发挥出异步传输的性能，并尝试实现。

参考提示：设计一个函数int filter(char source，char \*sink，int f)，其中：

1、source：源文件，即从哪个文件读。

2、sink：目标文件，即写到哪个文件。

3、f：一个与文件内容无关的操作（比如空循环）。

a)仍然设计三种模式的filter函数：无缓存模式、文件高速缓存模式和异步模式。

b)给出10个大小相同内容不同的源文件，分别调用三种模式的filter函数10次，读出文件的内容，并写到10个不同的目标文件中去。

c)在调用filter函数的过程中，加入一些与文件内容无关的操作，在此情况下观察三种模式之间的用时区别（与文件无关的操作耗时不能太短，否则效果将不明显）。

应该注意的是：在调用filter函数时，要加入与文件内容无关的f操作；由于用了10个不同的源文件，对每种传输模式，可以考虑创建10个线程来并发地执行文件传输。

d)记录每次函数调用的时间，以及10次操作的总时间，分析实习结果，从中体会异步传输模式的优越性，写出心得体会。

请描述你所做的工作：

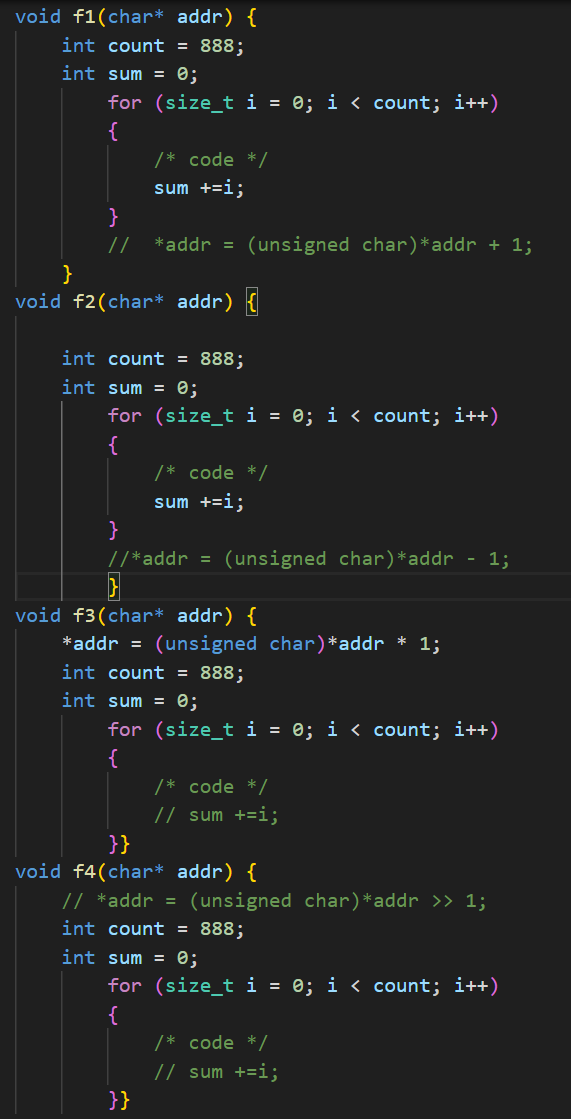


图片 2 编写nobuffer多线程处理文件函数，将每次对文件处理单独作为一个线程处理

文本

描述已自动生成

图片 3 编写sequen和overlp多线程处理文件函数，将每次对文件处理单独作为一个线程处理



图片 4 对f1234函数做修改，减少文件操作比重

文本

描述已自动生成

图片 5 对f5函数修改，减少文件操作比重

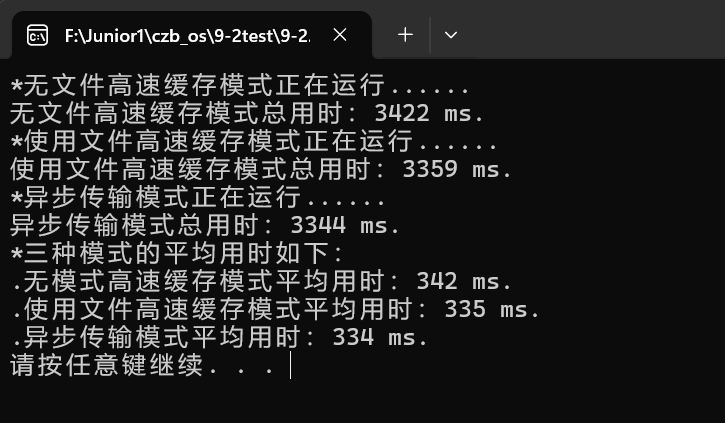
图形用户界面, 文本

描述已自动生成

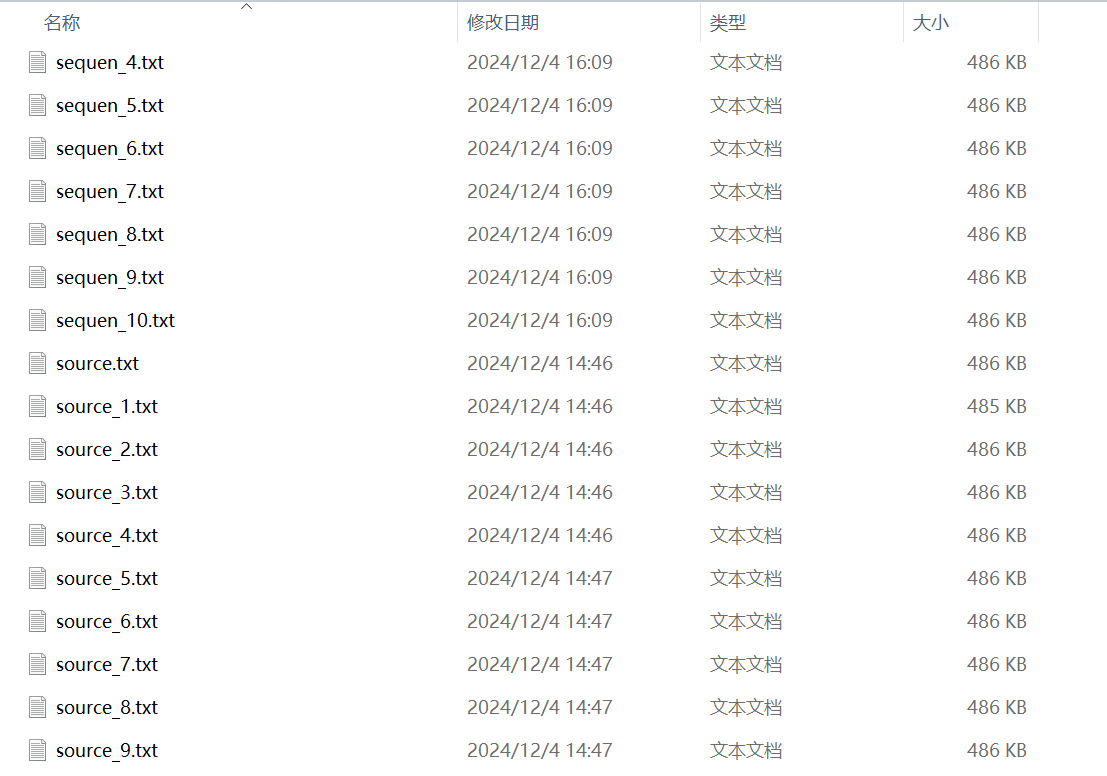
图片 6 主函数写明输入输出文件对



图片 7 主函数调用文件操作函数，并记录时间



图片 8 运行结果



图片 9 源文件一览

实验分析与心得体会

**无缓存模式**  
直接操作硬盘，文件读写速度慢。由于增加了耗时操作，这种模式的性能可能下降更多。

**顺序扫描模式**  
缓存优化后，性能有所提升。适合大文件处理，但仍然受到同步机制的限制。

**异步模式**  
异步操作减少了等待时间，与耗时操作并发运行，表现出显著优势。

附源代码:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*-------------------------------------------------------------

/\*三种模式

/\*  1.  FILE\_FLAG\_NOBUFFER

/\*  2.  FILE\_FLAG\_SEQUENTIAL\_SCAN

/\*  3.  FILE\_FLAG\_BUFFERING|FILE\_FLAG\_OVERLAPPED

/\*------------------------------------------------------------

/\*五种操作

/\*  1.  charactor   +1

/\*  2.  charactor   -1

/\*  3.  charactor   -32

/\*  4.  charactor   +32

/\*  5.  charactor   \*1

/\*---------------------------------------------------------------

\*/

#include<iostream>

#include<windows.h>

#include <thread>

#include <vector>

#include <string>

using namespace std;

//三种模式

void filter\_nobuffer(char\* source, char\* sink, void(\*func)(char\* addr));

void filter\_sequen(char\* source, char\* sink, void(\*func)(char\* addr));

//Overlap 用于异步重叠操作

void filter\_overlp(char\* source, char\* sink, void(\*func)(char\* addr));

//五种不同功能的操作

void f1(char\* addr);

void f2(char\* addr);

void f3(char\* addr);

void f4(char\* addr);

void f5(char\* addr);

#define BUFFER\_SIZE 2048    //定义缓冲区的大小，这里设为2048字节

char\*   buffer;             //这里的缓冲区被定义成char型

// 添加与文件内容无关的操作

void perform\_unrelated\_work() {

    const int count = 10e9; // 模拟耗时操作

    volatile int dummy = 0;

    for (int i = 0; i < count; ++i) {

        dummy += i % 7;

    }

}

void process\_files\_nobuffer(void (\*filter\_func)(char\*, char\*, void (\*)(char\*)),

                   const std::vector<std::pair<std::string, std::string>>& file\_pairs,

                   void (\*funcs[])(char\*)) {

    std::vector<std::thread> threads;

    std::vector<std::thread> threads\_func1;

    std::vector<std::thread> threads\_func2;

    for (size\_t i = 0; i < file\_pairs.size(); ++i) {

        threads.emplace\_back([filter\_func, &file\_pairs, funcs, i]() {

            filter\_func(const\_cast<char\*>(file\_pairs[i].first.c\_str()),

                        const\_cast<char\*>(file\_pairs[i].second.c\_str()), funcs[i]);

                        perform\_unrelated\_work();

        });

    }

    for (auto& t : threads) {

        t.join();

    }

}

void process\_files\_sequen(void (\*filter\_func)(char\*, char\*, void (\*)(char\*)),

                   const std::vector<std::pair<std::string, std::string>>& file\_pairs,

                   void (\*funcs[])(char\*)) {

    std::vector<std::thread> threads;

    std::vector<std::thread> threads\_func1;

    std::vector<std::thread> threads\_func2;

    for (size\_t i = 0; i < file\_pairs.size(); ++i) {

        threads.emplace\_back([filter\_func, &file\_pairs, funcs, i]() {

            filter\_func(const\_cast<char\*>(file\_pairs[i].first.c\_str()),

                        const\_cast<char\*>(file\_pairs[i].second.c\_str()), funcs[i]);

                        perform\_unrelated\_work();

        });

    }

    for (auto& t : threads) {

        t.join();

    }

}

void process\_files\_overlp(void (\*filter\_func)(char\*, char\*, void (\*)(char\*)),

                   const std::vector<std::pair<std::string, std::string>>& file\_pairs,

                   void (\*funcs[])(char\*)) {

    std::vector<std::thread> threads;

    std::vector<std::thread> threads\_func1;

    std::vector<std::thread> threads\_func2;

    for (size\_t i = 0; i < file\_pairs.size(); ++i) {

        threads.emplace\_back([filter\_func, &file\_pairs, funcs, i]() {

            filter\_func(const\_cast<char\*>(file\_pairs[i].first.c\_str()),

                        const\_cast<char\*>(file\_pairs[i].second.c\_str()), funcs[i]);

                        perform\_unrelated\_work();

        });

    }

    for (auto& t : threads) {

        t.join();

    }

}

/\*------------------------------------------------------------------\*/

//对文件内容进行的5种操作

//f1  +1

//f2  -1

//f3  \*1

//f4  >>

//f5  <<

void f1(char\* addr) {

    int count = 888;

    int sum = 0;

        for (size\_t i = 0; i < count; i++)

        {

            /\* code \*/

            sum +=i;

        }

        //  \*addr = (unsigned char)\*addr + 1;

    }

void f2(char\* addr) {

    int count = 888;

    int sum = 0;

        for (size\_t i = 0; i < count; i++)

        {

            /\* code \*/

            sum +=i;

        }

        //\*addr = (unsigned char)\*addr - 1;

        }

void f3(char\* addr) {

    \*addr = (unsigned char)\*addr \* 1;

    int count = 888;

    int sum = 0;

        for (size\_t i = 0; i < count; i++)

        {

            /\* code \*/

            // sum +=i;

        }}

void f4(char\* addr) {

    // \*addr = (unsigned char)\*addr >> 1;

    int count = 888;

    int sum = 0;

        for (size\_t i = 0; i < count; i++)

        {

            /\* code \*/

            // sum +=i;

        }}

void f5(char\* addr) {

    // \*addr = (unsigned char)\*addr << 1;

    int count = 888;

    int sum = 0;

        for (size\_t i = 0; i < count; i++)

        {

            /\* code \*/

            // sum +=i;

        }}

/\*-------------------------------------------------------------\*/

//没有文件高速缓存的filter函数

void filter\_nobuffer(char\* source, char\* sink, void(\*func)(char\* addr))

{

    HANDLE handle\_src, handle\_dst;  //定义原文件与目标文件的句柄

    BOOL cycle;                     //用来判断一个缓冲区是否被写满

    DWORD NumberOfBytesRead, NumberOfBytesWrite, index; //读的字节数、写的字节数

                                                        //打开原文件

                                                        //因为是OPEN\_EXISTING所以开始时得新建source.txt

handle\_src = CreateFile(source, GENERIC\_READ, NULL, NULL, OPEN\_EXISTING, FILE\_FLAG\_NO\_BUFFERING, NULL);

    //创建目标文件

handle\_dst = CreateFile(sink, GENERIC\_WRITE, NULL, NULL, CREATE\_ALWAYS, NULL, NULL);

    if(handle\_src == INVALID\_HANDLE\_VALUE || handle\_dst == INVALID\_HANDLE\_VALUE)

    {

        cout << "CreatFile Invocation Error!" << endl;

        exit(1);

    }

    cycle = TRUE;

    //用来cycle判断文件什么时候读完

    while (cycle)

    {

        //从原文件读数据送入缓冲区

if (ReadFile(handle\_src, buffer, BUFFER\_SIZE, &NumberOfBytesRead, NULL) == FALSE)

        {

            cout << "ReadFile Error!" << endl;

            exit(1);

        }

        //当读不满一个缓冲区时，说明达到文件末尾，结束循环

        if (NumberOfBytesRead < BUFFER\_SIZE)

            cycle = FALSE;

        //对文件内容进行的操作

        for (index = 0; index < NumberOfBytesRead; index++)

            func(&buffer[index]);

        //将缓冲区中的数据写入目标文件

if (WriteFile(handle\_dst, buffer, NumberOfBytesRead, &NumberOfBytesWrite, NULL) == FALSE)

        {

            cout << "WriteFile Error!" << endl;

            exit(1);

        }

    }

    //关闭文件句柄

    CloseHandle(handle\_src);

    CloseHandle(handle\_dst);

}

/\*-------------------------------------------------------------\*/

void filter\_sequen(char\* source, char\* sink, void(\*func)(char\* addr))

{

    HANDLE handle\_src, handle\_dst;  //定义原文件与目标文件的句柄

    BOOL cycle;                     //用来判断一个缓冲区是否被写满

    DWORD NumberOfBytesRead, NumberOfBytesWrite, index; //读的字节数、写的字节数

                                                        //打开原文件

handle\_src = CreateFile(source, GENERIC\_READ, NULL, NULL, OPEN\_EXISTING, FILE\_FLAG\_SEQUENTIAL\_SCAN, NULL);

    //创建目标文件

handle\_dst = CreateFile(sink, GENERIC\_WRITE, NULL, NULL, CREATE\_ALWAYS, FILE\_FLAG\_SEQUENTIAL\_SCAN, NULL);

if(handle\_src == INVALID\_HANDLE\_VALUE || handle\_dst == INVALID\_HANDLE\_VALUE)

    {

        cout << "CreatFile Invocation Error!" << endl;

        exit(1);

    }

    cycle = TRUE;

    //用来cycle判断文件什么时候读完

    while (cycle)

    {

        //从原文件读数据送入缓冲区

if (ReadFile(handle\_src, buffer, BUFFER\_SIZE, &NumberOfBytesRead, NULL) == FALSE)

        {

            cout << "ReadFile Error!" << endl;

            exit(1);

        }

        //当读不满一个缓冲区时，说明达到文件末尾，结束循环

        if (NumberOfBytesRead < BUFFER\_SIZE)

            cycle = FALSE;

        //对文件内容进行的操作

        for (index = 0; index < NumberOfBytesRead; index++)

            func(&buffer[index]);

        //将缓冲区中的数据写入目标文件

if (WriteFile(handle\_dst, buffer, NumberOfBytesRead, &NumberOfBytesWrite, NULL) == FALSE)

        {

            cout << "WriteFile Error!" << endl;

            exit(1);

        }

    }

    //关闭文件句柄

    CloseHandle(handle\_src);

    CloseHandle(handle\_dst);

}

/\*-------------------------------------------------------------\*/

void filter\_overlp(char\* source, char\* sink, void(\*func)(char\* addr))

{

    HANDLE handle\_src, handle\_dst;  //定义原文件与目标文件的句柄

    BOOL cycle;                     //用来判断一个缓冲区是否被写满

    DWORD NumberOfBytesRead, NumberOfBytesWrite, index, dwError;    //读的字节数、写的字节数

    OVERLAPPED overlapped;          //overlapped 结构

                                    //打开原文件

handle\_src = CreateFile(source, GENERIC\_READ, NULL, NULL, OPEN\_EXISTING,     FILE\_FLAG\_NO\_BUFFERING | FILE\_FLAG\_OVERLAPPED, NULL);

    //创建目标文件

handle\_dst = CreateFile(sink, GENERIC\_WRITE, NULL, NULL, CREATE\_ALWAYS, NULL, NULL);

    if(handle\_src == INVALID\_HANDLE\_VALUE || handle\_dst == INVALID\_HANDLE\_VALUE)

    {

        cout << "CreatFile Invocation Error!" << endl;

        exit(1);

    }

    cycle = TRUE;

    //对overlapped结构初始化

    overlapped.hEvent = NULL;

    overlapped.Offset = -BUFFER\_SIZE;

    overlapped.OffsetHigh = 0;

    //用来cycle判断文件什么时候读完

    while (cycle)

    {

        //计算文件的偏移量

        overlapped.Offset = overlapped.Offset + BUFFER\_SIZE;

        //从原文件读数据送入缓冲区

if (ReadFile(handle\_src, buffer, BUFFER\_SIZE, &NumberOfBytesRead, &overlapped) == FALSE)

        {

            switch (dwError = GetLastError())

            {

                //读文件结尾

            case ERROR\_HANDLE\_EOF:

                cycle = FALSE;

                break;

            case ERROR\_IO\_PENDING:

if (GetOverlappedResult(handle\_src, &overlapped, &NumberOfBytesRead, TRUE) == FALSE)

                {

                    cout << "GetOverlappedResult Error!" << endl;

                    exit(1);

                }

                break;

            default:

                break;

            }

        }

        //当不满一个个缓存区时，说明达到文件末尾，结束循环

        if (NumberOfBytesRead < BUFFER\_SIZE)

            cycle = FALSE;

        //对文件内容进行的操作

        for (index = 0; index < NumberOfBytesRead; index++)

            func(&buffer[index]);

        //将缓冲区中的数据写入目标文件

if (WriteFile(handle\_dst, buffer, NumberOfBytesRead, &NumberOfBytesWrite, NULL) == FALSE)

        {

            cout << "WriteFile Error!" << endl;

            exit(1);

        }

    }

    //关闭文件句柄

    CloseHandle(handle\_src);

    CloseHandle(handle\_dst);

}

void main()

{

    //分配缓冲区

    buffer = new char[BUFFER\_SIZE];

    //记录执行filter函数的开始时间

    DWORD tick;

    //用于三种模式各自的平均时间

    DWORD nobuffer\_average\_time = 0;

    DWORD sequen\_average\_time = 0;

    DWORD overlp\_average\_time = 0;

    // 定义文件对和函数指针

    vector<pair<string, string>> file\_pairs\_nobuffer = {

        {"source.txt", "nobuffer\_1.txt"}, {"source\_1.txt", "nobuffer\_2.txt"},

        {"source\_2.txt", "nobuffer\_3.txt"}, {"source\_3.txt", "nobuffer\_4.txt"},

        {"source\_4.txt", "nobuffer\_5.txt"}, {"source\_5.txt", "nobuffer\_6.txt"},

        {"source\_6.txt", "nobuffer\_7.txt"}, {"source\_7.txt", "nobuffer\_8.txt"},

        {"source\_8.txt", "nobuffer\_9.txt"}, {"source\_9.txt", "nobuffer\_10.txt"}};

    vector<pair<string, string>> file\_pairs\_sequen = {

        {"source.txt", "sequen\_1.txt"}, {"source\_1.txt", "sequen\_2.txt"},

        {"source\_2.txt", "sequen\_3.txt"}, {"source\_3.txt", "sequen\_4.txt"},

        {"source\_4.txt", "sequen\_5.txt"}, {"source\_5.txt", "sequen\_6.txt"},

        {"source\_6.txt", "sequen\_7.txt"}, {"source\_7.txt", "sequen\_8.txt"},

        {"source\_8.txt", "sequen\_9.txt"}, {"source\_9.txt", "sequen\_10.txt"}};

    vector<pair<string, string>> file\_pairs\_overlp = {

        {"source.txt", "overlp\_1.txt"}, {"source\_1.txt", "overlp\_2.txt"},

        {"source\_2.txt", "overlp\_3.txt"}, {"source\_3.txt", "overlp\_4.txt"},

        {"source\_4.txt", "overlp\_5.txt"}, {"source\_5.txt", "overlp\_6.txt"},

        {"source\_6.txt", "overlp\_7.txt"}, {"source\_7.txt", "overlp\_8.txt"},

        {"source\_8.txt", "overlp\_9.txt"}, {"source\_9.txt", "overlp\_10.txt"}};

    void (\*funcs[])(char\*) = {f1, f2, f3, f4, f5, f1, f2, f3, f4, f5};

    // 记录时间并运行无缓存模式

    cout << "\*无文件高速缓存模式正在运行......" << endl;

    DWORD start\_time = GetTickCount();

    process\_files\_nobuffer(filter\_nobuffer, file\_pairs\_nobuffer, funcs);

    DWORD nobuffer\_time = GetTickCount() - start\_time;

    cout << "无文件高速缓存模式总用时：" << nobuffer\_time << " ms." << endl;

    // 记录时间并运行文件缓存模式

    cout << "\*使用文件高速缓存模式正在运行......" << endl;

    start\_time = GetTickCount();

    process\_files\_sequen(filter\_sequen, file\_pairs\_sequen, funcs);

    DWORD sequen\_time = GetTickCount() - start\_time;

    cout << "使用文件高速缓存模式总用时：" << sequen\_time << " ms." << endl;

        // 记录时间并运行异步模式

    cout << "\*异步传输模式正在运行......" << endl;

     start\_time = GetTickCount();

    process\_files\_overlp(filter\_overlp, file\_pairs\_overlp, funcs);

    DWORD overlp\_time = GetTickCount() - start\_time;

    cout << "异步传输模式总用时：" << overlp\_time << " ms." << endl;

    // 输出平均时间对比

    cout << "\*三种模式的平均用时如下：" << endl;

    cout << ".无模式高速缓存模式平均用时：" << nobuffer\_time / 10 << " ms." << endl;

    cout << ".使用文件高速缓存模式平均用时：" << sequen\_time / 10 << " ms." << endl;

    cout << ".异步传输模式平均用时：" << overlp\_time / 10 << " ms." << endl;

    system("pause");

    return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*THE END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/