__FILE__ 是C/C++和某些其他编程语言中的一个预定义宏，用于表示当前源文件的文件名（包括路径）。它在编译时被替换为一个字符串常量，表示当前源文件的名称。

主要用途
调试信息：在调试或日志记录中，__FILE__ 可以用于输出当前代码所在的文件名，帮助开发者定位问题。

错误处理：在错误处理代码中，__FILE__ 可以用于输出发生错误的文件名，便于追踪和修复问题。
相关宏
__LINE__：表示当前代码行号。

__func__ 或 __FUNCTION__：表示当前函数的名称（C99 和 C++11 标准）。

__DATE__：表示编译日期。

__TIME__：表示编译时间。
在C++中，pch.h 通常是指 预编译头文件（Precompiled Header, PCH）。预编译头文件是一种用于加快编译速度的技术，特别是在大型项目中，可以减少重复编译相同头文件的时间。

1. 预编译头文件的作用
加快编译速度：预编译头文件将常用的、不经常变化的头文件预先编译成一种中间格式，编译器在后续编译中可以直接使用这个中间格式，而不需要重新解析和编译这些头文件。

减少重复编译：在大型项目中，许多源文件可能包含相同的头文件（如标准库头文件、第三方库头文件等），使用预编译头文件可以避免重复编译这些头文件。
MYSQL_RES 是 MySQL C API 中的一个结构体类型，用于表示 查询结果集。当你执行一个查询（如 SELECT 语句）后，MySQL 会返回一个结果集，MYSQL_RES 结构体就是用来存储和管理这个结果集的。
MySQL C API 是 MySQL 提供的一组 C 语言接口，用于在 C/C++ 程序中与 MySQL 数据库进行交互。它允许开发者执行 SQL 查询、管理数据库连接、处理结果集等操作。以下是 MySQL C API 的核心功能和使用方法：

1. 核心数据结构
MySQL C API 定义了几个重要的数据结构：

MYSQL：表示一个数据库连接句柄，用于管理与 MySQL 服务器的连接。

MYSQL_RES：表示查询结果集，用于存储查询返回的数据。

MYSQL_ROW：表示结果集中的一行数据，是一个字符串数组（char **）。

MYSQL_FIELD：表示结果集中字段的元数据（如字段名、类型、长度等）。

1. MYSQL_RES 的作用
存储查询结果：MYSQL_RES 结构体包含了查询返回的所有行和列的数据。

提供结果集操作：通过 MySQL C API 提供的函数，可以遍历、读取和释放结果集。
常用函数
以下是 MySQL C API 中常用的函数：

连接和初始化
函数名 功能描述
mysql_init() 初始化一个 MYSQL 对象，用于连接数据库。
mysql_real_connect() 连接到 MySQL 服务器。
mysql_close() 关闭数据库连接并释放资源。
执行查询
函数名 功能描述
mysql_query() 执行 SQL 查询（如 SELECT、INSERT、UPDATE 等）。
mysql_real_query() 执行 SQL 查询，支持二进制数据。
处理结果集
函数名 功能描述
mysql_store_result() 获取完整的查询结果集并存储在 MYSQL_RES 中。
mysql_use_result() 初始化逐行读取结果集，适用于处理大量数据时节省内存。
mysql_fetch_row()从结果集中获取下一行数据，返回一个 MYSQL_ROW。
mysql_num_rows() 返回结果集中的行数(仅适用于 mysql_store_result())。
mysql_num_fields() 返回结果集中的列数。
mysql_fetch_lengths() 返回当前行中每个字段的长度。
mysql_free_result() 释放 MYSQL_RES 结构体占用的内存。
错误处理
函数名 功能描述
mysql_error() 返回最后一次错误的错误信息。
mysql_errno() 返回最后一次错误的错误代码。
其他功能
函数名 功能描述
mysql_affected_rows() 返回受 INSERT、UPDATE、DELETE 影响的行数。
mysql_insert_id()返回最后一次 INSERT 操作生成的自增 ID。
sprintf 函数打印到字符串中（要注意字符串的长度要足够容纳打印的内容，否则会出现内存溢出），而 printf 函数打印输出到屏幕上。sprintf 函数在我们完成其他数据类型转换成字符串类型的操作中应用广泛。
初始化连接:

使用 mysql_init 初始化 MySQL 连接对象。

使用 mysql_real_connect 连接到 MySQL 数据库。

执行查询:

使用 mysql_query 执行 SQL 查询。

获取结果集:

使用 mysql_store_result 获取查询结果集。

获取字段信息:

使用 mysql_fetch_fields 获取字段元数据。

使用 mysql_num_fields 获取字段数量。

输出字段信息:

遍历字段数组，输出字段的名称、表名、类型、长度等信息。

输出查询结果:

使用 mysql_fetch_row 逐行获取查询结果并输出。

释放资源:

使用 mysql_free_result 释放结果集。

使用 mysql_close 关闭数据库连接。
mysql_fetch_fields 返回的字段对象包含以下重要属性：

name: 字段名。

table: 字段所属的表名。

type: 字段的数据类型（如 MYSQL_TYPE_LONG 表示整数）。

length: 字段的定义长度。

max_length: 字段在结果集中的最大长度。

flags: 字段的标志（如 NOT_NULL_FLAG、PRI_KEY_FLAG 等）。

decimals: 小数位数（适用于浮点类型）。
读取自定义格式配置文件
如果配置文件是自定义格式，可以手动编写解析逻辑。

示例代码
cpp
复制
```
#include <iostream>
#include <fstream>
#include <map>
#include <string>

std::map<std::string, std::string> loadConfig(const std::string&
filename) {
    std::map<std::string, std::string> config;
    std::ifstream configFile(filename);

    std::string line;
    while (std::getline(configFile, line)) {
        size_t delimiterPos = line.find('=');
        if (delimiterPos != std::string::npos) {
            std::string key = line.substr(0, delimiterPos);
            std::string value = line.substr(delimiterPos + 1);
            config[key] = value;
        }
    }

    return config;
}

int main() {
    auto config = loadConfig("config.txt");

    // 读取配置
    std::cout << "Database Host: " << config["host"] << std::endl;
    std::cout << "Database Port: " << config["port"] << std::endl;

    return 0;
}
```
配置文件 (config.txt)
plaintext
复制
```
host=localhost
port=3306
```

username=root
password=123456
在C++中，npos 是 std::string 类中的一个静态常量，表示“未找到”或“无效位置”。它的全称是 std::string::npos，通常用于字符串查找操作中，表示查找失败或未找到目标子字符串。

1. npos 的定义
npos 是 std::string 类的一个静态成员常量，类型为 std::string::size_type（通常是无符号整数类型，如 size_t）。

它的值通常是 size_t 类型的最大值（即 size_t 的 -1，因为 size_t 是无符号类型）。

2. npos 的常见用途
npos 主要用于字符串查找函数中，表示未找到目标子字符串。以下是一些常见的用法：

1. std::string::find
find 函数用于查找子字符串或字符在字符串中的位置。

如果未找到目标，返回 npos。
在C++中，size_t 是一种无符号整数类型，通常用于表示对象的大小或索引。它是C++标准库中广泛使用的一种类型，特别是在与内存分配、数组索引和字符串操作相关的场景中。

1. size_t 的定义
size_t 是C++标准库中定义的一个类型别名，通常表示一个无符号整数类型。

它的具体定义依赖于编译器和平台，但通常是 unsigned int、unsigned long 或 unsigned long long 的别名。

在标准库中，size_t 的定义通常位于 <cstddef> 头文件中。

2. size_t 的用途
size_t 主要用于以下场景：
在 C++中，文件操作主要通过标准库中的 <fstream> 头文件实现。<fstream> 提供了 ifstream（输入文件流）、ofstream（输出文件流）和 fstream（文件流，支持输入和输出）等类，用于文件的读写操作。以下是C++文件操作的详细指南：

1. 文件操作的基本步骤
打开文件：

使用 open() 方法或构造函数打开文件。

需要指定文件名和打开模式（如 ios::in、ios::out 等）。

检查文件是否成功打开：

使用 is_open() 方法检查文件是否成功打开。

读写文件：

使用 >>、<<、get()、getline()、read()、write() 等方法进行读写操作。

关闭文件：

使用 close() 方法关闭文件。

2. 文件打开模式
文件打开模式是 ios 类的静态常量，可以通过位或操作符 | 组合使用。常见的模式包括：

模式　描述
ios::in　以读取方式打开文件（ifstream 默认模式）。
ios::out　以写入方式打开文件（ofstream 默认模式），如果文件存在则清空内容。
ios::app　以追加方式打开文件，写入的数据追加到文件末尾。
ios::ate　打开文件后，定位到文件末尾。
ios::trunc 如果文件存在,清空文件内容(与 ios::out 一起使用时默认启用)。
ios::binary　以二进制模式打开文件。
3. 文件读写操作
3.1 写入文件
使用 ofstream 类将数据写入文件。

示例：写入文本文件
cpp
复制
```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream outFile("example.txt", std::ios::out); // 打开文件
    if (!outFile.is_open()) {
        std::cerr << "文件打开失败！" << std::endl;
        return 1;
    }
```

```
    outFile << "Hello, World!" << std::endl; // 写入数据
    outFile << "This is a test file." << std::endl;

    outFile.close(); // 关闭文件
    std::cout << "文件写入成功！" << std::endl;
    return 0;
}
```

示例：写入二进制文件

cpp

复制

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream     outFile("example.bin",     std::ios::out     |
std::ios::binary);
    if (!outFile.is_open()) {
        std::cerr << "文件打开失败！" << std::endl;
        return 1;
    }

    int data[] = {1, 2, 3, 4, 5};
    outFile.write(reinterpret_cast<char*>(data), sizeof(data)); // 写
入二进制数据

    outFile.close();
    std::cout << "二进制文件写入成功！" << std::endl;
    return 0;
}
```

3.2 读取文件

使用 ifstream 类从文件中读取数据。

示例：读取文本文件

cpp

复制

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream inFile("example.txt", std::ios::in); // 打开文件
    if (!inFile.is_open()) {
        std::cerr << "文件打开失败！" << std::endl;
        return 1;
```

```
    }

    std::string line;
    while (std::getline(inFile, line)) { // 逐行读取
        std::cout << line << std::endl;
    }

    inFile.close(); // 关闭文件
    return 0;
}
```

示例：读取二进制文件

cpp

复制

```
#include <iostream>
#include <fstream>

int main() {
    std::ifstream      inFile("example.bin",      std::ios::in      |
std::ios::binary);
    if (!inFile.is_open()) {
        std::cerr << "文件打开失败！" << std::endl;
        return 1;
    }

    int data[5];
    inFile.read(reinterpret_cast<char*>(data), sizeof(data)); // 读取
二进制数据

    for (int i = 0; i < 5; ++i) {
        std::cout << data[i] << " ";
    }
    std::cout << std::endl;

    inFile.close();
    return 0;
}
```

3.3 文件定位

使用 seekg() 和 seekp() 方法可以移动文件指针，用于随机访问文件。

示例：文件定位

cpp

复制

```
#include <iostream>
#include <fstream>
```

```
int main() {
    std::fstream file("example.txt", std::ios::in | std::ios::out);
    if (!file.is_open()) {
        std::cerr << "文件打开失败！" << std::endl;
        return 1;
    }

    file.seekp(7, std::ios::beg); // 将写指针移动到第 7 个字节
    file << "C++"; // 写入数据

    file.seekg(0, std::ios::beg); // 将读指针移动到文件开头
    std::string line;
    std::getline(file, line); // 读取一行
    std::cout << line << std::endl;

    file.close();
    return 0;
}
```

3.4 检查文件状态

eof()：检查是否到达文件末尾。

fail()：检查文件操作是否失败。

good()：检查文件是否处于正常状态。

示例：检查文件状态

cpp

复制

```
#include <iostream>
#include <fstream>

int main() {
    std::ifstream inFile("example.txt");
    if (!inFile.is_open()) {
        std::cerr << "文件打开失败！" << std::endl;
        return 1;
    }

    char ch;
    while (inFile.get(ch)) { // 逐个字符读取
        std::cout << ch;
    }
```

```
    if (inFile.eof()) {
        std::cout << "\n 到达文件末尾！" << std::endl;
    }

    inFile.close();
    return 0;
}
```

4. 文件操作的高级用法

4.1 追加写入

使用 ios::app 模式将数据追加到文件末尾。

示例：追加写入

cpp

复制

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream outFile("example.txt", std::ios::app);
    if (!outFile.is_open()) {
        std::cerr << "文件打开失败！" << std::endl;
        return 1;
    }

    outFile << "This is appended text." << std::endl;

    outFile.close();
    std::cout << "数据追加成功！" << std::endl;
    return 0;
}
```

4.2 文件重命名和删除

使用标准库函数 rename() 和 remove() 可以重命名和删除文件。

示例：文件重命名和删除

cpp

复制

```
#include <iostream>
#include <cstdio>

int main() {
    if (std::rename("old.txt", "new.txt") != 0) {
        std::cerr << "文件重命名失败！" << std::endl;
    } else {
        std::cout << "文件重命名成功！" << std::endl;
```

```
    }

    if (std::remove("new.txt") != 0) {
        std::cerr << "文件删除失败！" << std::endl;
    } else {
        std::cout << "文件删除成功！" << std::endl;
    }

    return 0;
}
```

5. 总结

C++文件操作主要通过 <fstream> 实现，包括 ifstream、ofstream 和 fstream。

文件操作的基本步骤：打开文件、检查文件状态、读写文件、关闭文件。

支持文本模式和二进制模式的文件读写。

可以使用 seekg() 和 seekp() 实现文件随机访问。

文件重命名和删除可以通过 rename() 和 remove() 实现。

掌握这些操作后，您可以轻松处理 C++中的文件读写任务！

++ 操作的非线程安全性

++ 操作通常分为以下步骤：

读取变量的当前值。

将变量的值加 1。

将新值写回变量。

在多线程环境中，如果多个线程同时对同一个变量执行 ++ 操作，可能会导致以下问题：

竞争条件：多个线程同时读取变量的旧值，然后分别加 1 并写回，导致最终结果不正确。

数据不一致：变量的最终值可能小于预期值。

互斥锁：实现简单，但性能较低，因为每次操作都需要加锁和解锁。

原子操作：性能较高，适合简单的操作（如 ++）。

线程局部存储：性能最高，但仅适用于不需要共享变量的场景。

std::atomic 的作用

提供线程安全的操作，避免竞争条件（Race Condition）。

支持对基本数据类型（如 int、bool、指针 等）的原子操作。

提供内存顺序控制（Memory Order），允许开发者优化性能。
CAS（Compare-And-Swap） 是一种原子操作，用于实现无锁编程（Lock-Free Programming ） 。 在 C++ 中 ， CAS 操 作 可 以 通 过 std::atomic 的 compare_exchange_weak 和 compare_exchange_strong 成员函数来实现。
std::unique_ptr
特点：

独占所有权，同一时间只能有一个 std::unique_ptr 指向一个对象。

不能复制，只能移动（通过 std::move）。

当 std::unique_ptr 被销毁时，它会自动释放所管理的对象。

适用场景：

适用于需要独占资源所有权的场景。
std::shared_ptr
特点：

共享所有权，多个 std::shared_ptr 可以指向同一个对象。

使用引用计数管理资源，当最后一个 std::shared_ptr 被销毁时，对象会被释放。

可以复制和移动。

适用场景：

适用于需要共享资源所有权的场景。
wait_for 函数的作用
阻塞线程：调用 wait_for 的线程会被阻塞，直到被其他线程通过 notify_one 或 notify_all 唤醒，或者超时。

释放锁：在阻塞期间，wait_for 会释放与 std::unique_lock 关联的互斥锁，允许其他线程获取锁并修改共享数据。

重新获取锁：当线程被唤醒或超时后，wait_for 会重新获取锁，然后继续执行。

无条件的 wait_for
返回值类型

std::cv_status 枚举类型，包含两个值：

std::cv_status::no_timeout：线程被唤醒（通过 notify_one 或 notify_all）。

std::cv_status::timeout：线程因超时而被唤醒。
带条件的 wait_for
bool 类型：

true：条件 pred 为 true（线程被唤醒且条件满足）。

false：超时（线程因超时而被唤醒，且条件 pred 仍为 false）。
clock_t 是用于存储时钟滴答数的类型。
CLOCKS_PER_SEC 是一个宏，表示每秒的时钟滴答数。

通常值为 1000000，表示 1 秒 = 1,000,000 个时钟滴答。
clock() 函数返回程序从启动到当前时刻的 CPU 时间。