写在前面

PS:gitee(码云)在国内可以访问

项目介绍

基于 C++11 的可定制化线程池实现

github 地址：https://github.com/Haroyee/Threadpool

gitee 地址：https://gitee.com/poida/threadpool

平台工具：vim+vscode开发、makefile+g++编译、gdb调试分析死锁问题

项目描述:

1. 基于 C++11 可变参模板和引用折叠特性，设计了灵活的submitTask系列接口（submitHighTsk/submitNormalTsk/submitLowTsk），支持任意类型函数、任意数量参数的任务提交
2. 采用unordered_map管理线程对象，priority_queue实现带优先级的任务队列（支持高 / 中 / 低三级优先级），通过对Task类运算符重载实现任务的优先级调度
3. 基于mutex和condition_variable实现多线程同步机制，解决任务提交线程与执行线程的通信问题，包括任务提交等待、任务执行通知、线程动态增减通知等
4. 支持fixed模式和cached模式的线程池定制
5. 结合packaged_task和future实现任务返回值获取

项目问题

1. 线程池提交任务阶段直接陷入死锁，无任何报错
2. MODE_CACHED模式下，当任务数量短时间内急剧增加时，会触发大量线程创建，导致系统资源消耗激增

分析定位问题

1. 通过 gdb 调试工具定位到死锁根源：在任务入队阶段获取到锁之后调用其他成员函数后，成员函数内又上了一次锁，导致二次上锁，造成死锁。通过移除该成员函数的上锁操作得到解决
2. 针对MODE_CACHED的资源消耗问题，通过增强MODE_FIXED模式的稳定性，限制最大并发线程数，结合任务队列上限阈值（taskUpperThresh）控制任务堆积，避免系统过载

编写该项目中遇到的问题

1、如何实现线程中任务的切换?

起初由于对thread的性质不了解，考虑过在thread外部使用函数指针，直接将函数替换掉。但在实际编程中，线程在一个函数执行完成后过一段时间操作系统会自动回收线程，在线程外部无法得函数的执行情况，直接用函数指针将函数替换掉线程并不会执行替换后的函数。

解决方法：写一个线程主函数，这个主函数包装进线程里，在这个主函数里实现取任务，执行任务，既保证任务切换，保证线程保持运行不被回收。

2、该线程池如何实现任意类型函数、任意数量参数的任务提交

第一个想到的肯定是利用模版来实现，但是不知道如何实现，有点焦头烂额，因为有点无从下手，通过网上查询资料，通过可变参模版实现任意数量和类型的参数，使用decltype，declval，auto自动推导出返回值类型，declval生成一个临时的右值引用，模拟函数的执行，实际不执行，配合decltype在不执行函数的情况下推导出函数返回值。

利用bind将函数和参数绑定，包装成可调用无参函数，在包装进package_task,使用future异步获取结果，所以任务提交函数实际的返回值是一个future类型的值。

3、cache模式下如何实现线程的销毁

起初是将线程分离然后，在线程内部空闲超过一段时间后自我销毁，但是这样会使线程脱离控制。所以单独开辟一个线程用于销毁空闲线程，并用另声明一个数组用于存储当前空闲的线程id，在指定线程中统一查询无序map中空闲的线程将其销毁。

销毁过程也有一些小插曲，开始使用jionable()判断线程是否处于空闲状态，比如使用for循环迭代器进行销毁，导致迭代器失效，实际上只销毁了一个线程后续的空闲线程都没有没销毁。所以另声明了一个数组存储空闲线程id，在获取线程的状态的同时便于查询空闲线程并销毁相应线程。

4、线程池的执行流程

提交任务->唤醒增加线程的线程，若当前线程小于任务池任务数，新建线程，但要小于上限线程数->线程取任务，执行任务，取任务如此循环->线程空闲一段时间，将id放入存储空闲线程id的数组->唤醒销毁线程的线程统一销毁空闲线程。

一、线程池初步设计

1.1 设计目标

本线程池旨在提供高效的多线程任务调度能力，支持两种线程管理模式（固定数量线程、动态调整线程），并允许按优先级提交任务，同时保证线程安全和资源合理利用。具体目标包括：

- 支持固定数量线程（MODE_FIXED）和动态扩展线程（MODE_CACHED）两种模式
- 任务可设置优先级（高、中、低），优先级高的任务优先执行
- 提供任务提交接口，支持任意参数和返回值的函数
- 实现线程动态管理（创建 / 销毁），避免资源浪费
- 保证任务队列和线程状态的线程安全
- 支持线程池优雅关闭，确保资源正确释放

1.2 核心组件设计

线程池由以下核心组件构成：

- **Task 类**：封装任务的优先级和具体执行逻辑，支持优先级比较
- **Threadpool 类**：线程池核心管理类，负责线程创建 / 销毁、任务队列管理、任务调度等
- **同步机制**：使用互斥锁（mutex）和条件变量（condition_variable）保证多线程安全访问共享资源
- **线程容器**：存储当前活跃线程，支持快速查询和删除
- **任务队列**：基于优先级的队列（priority_queue），按任务优先级排序

1.3 模式与优先级设计

- **线程池模式**：
- - MODE_FIXED：线程数量固定为初始设置值，不会动态增减

- MODE_CACHED：线程数量可动态调整，根据任务量自动创建新线程（不超过上限），空闲线程超时后自动销毁（保留初始数量线程）

- **任务优先级**：
  - HIGH（高优先级）：数值 2，优先执行
  - NORMAL（普通优先级）：数值 1，默认优先级
  - LOW（低优先级）：数值 0，最后执行
  - 任务队列通过大顶堆实现，优先级高的任务排在队列前端

二、实现线程池

2.1 枚举类实现

2.1.1 线程池模式

- MODE_FIXED：线程数量固定为初始设置值，不会动态增减
- MODE_CACHED：线程数量可动态调整，根据任务量自动创建新线程（不超过上限），空闲线程超时后自动销毁（保留初始数量线程）

```
enum class PoolMode // 线程池模式
{
    MODE_FIXED,  // 固定数量的线程
    MODE_CACHED, // 线程数量可动态增长（cache模式）
};
```

2.1.2 任务优先级

- HIGH（高优先级）：数值 2，优先执行
- NORMAL（普通优先级）：数值 1，默认优先级
- LOW（低优先级）：数值 0，最后执行

```
enum class Priority // 任务优先级
{
    HIGH = 2,   // 高优先级
    NORMAL = 1, // 普通优先级
    LOW = 0     // 低优先级
};
```

2.2 Task 类实现

Task 类用于封装单个任务的优先级和执行逻辑，核心功能如下：

2.2.1 核心成员

- prio：任务优先级（Priority 类型）
- func：任务执行逻辑（std::function<void ()> 类型，封装具体函数）

2.2.2 关键方法

- **构造函数**：接收优先级和函数对象，初始化成员变量

```
Task::Task(Priority &p, std::function<void()> &f) : prio(p), func(f) {}
```

- **优先级比较重载**：重载<运算符，用于 priority_queue 的大顶堆排序（优先级高的任务 "小于" 优先级低的任务，确保大顶堆顶部为最高优先级任务）

```
bool Task::operator<(const Task &tsk) const {
    return static_cast<int>(this->prio) > static_cast<int>(tsk.prio);
}
```

- **访问接口**：提供getPrio()和getFunc()方法，用于获取任务优先级和执行函数

2.2.3 完整代码

threadpool.h 声明部分

```
class Task // 包装任务
{
public:
    Task(Priority &p, std::function<void()> &);
    ~Task() = default;
    /*重载运算符，用于堆的排序*/
    bool operator<(const Task &) const; // 小于返回true，优先级更高，但是大顶堆
    Priority &getPrio();
    const std::function<void()> &getFunc() const;

private:
    Priority prio;               // 任务优先级
    std::function<void()> func; // 用于接受std::package_task类型，std::package_task是一个无返回值无参对象
};
```

threadpool.cpp 实现部分

```
/*Task类*/
// 构造函数
Task::Task(Priority &p, std::function<void()> &f) : prio(p), func(f) {}
// 操作符号重构
bool Task::operator<(const Task &tsk) const // 大顶堆
{
    return static_cast<int>(this->prio) > static_cast<int>(tsk.prio);
}

Priority &Task::getPrio() // 返回优先级
{
    return prio;
};
const std::function<void()> &Task::getFunc() const
{
    return func;
}
```

2.3 Threadpool 类核心实现

2.3.1 成员变量设计

线程池核心成员变量按功能分为以下几类：

- **线程管理相关**：
  - threads：存储线程的哈希表（key 为线程 ID，value 为线程智能指针）
  - initThreadSize：初始线程数量
  - threadUpperThresh：MODE_CACHED 模式下线程数量上限
  - idleThreadSize：当前空闲线程数量（原子变量，确保线程安全）
- **任务队列相关**：
  - tasks：优先级任务队列（priority_queue）
  - taskUpperThresh：任务队列最大容量
- **同步机制**：
  - taskQueMtx：保护任务队列的互斥锁
  - taskCv：任务队列条件变量（用于通知任务提交）
  - threadCv：线程条件变量（用于通知线程执行任务）
- **控制参数**：
  - poolMode：当前线程池模式（MODE_FIXED/MODE_CACHED）
  - stopFlag：线程池停止标志（原子变量）
  - destroyWaitTime：MODE_CACHED 模式下空闲线程销毁等待时间
  - submitWaitTime：任务提交时的等待超时时间

```
/*线程相关变量*/
std::unordered_map<std::thread::id, std::unique_ptr<std::thread>> threads; // 线程哈希表
size_t initThreadSize;                                                     // 初始线程数量
size_t threadUpperThresh;                                                  // cache模式下线程上限阈值
std::atomic<size_t> idleThreadSize;                                        // 当前空闲线程数量

/*任务相关变量*/
std::priority_queue<Task> tasks; // 任务队列，默认大顶堆
size_t taskUpperThresh;          // 任务上限阈值

/*线程互斥锁与条件变量*/
mutable std::mutex taskQueMtx;    // 保证任务队列的线程安全,mutable确保可以在const内修改
std::condition_variable taskCv;   // 任务条件变量
std::condition_variable threadCv; // 线程条件变量

PoolMode poolMode;          // 当前线程池的工作模式
std::atomic<bool> stopFlag; // 线程池停止标志

/*时间变量*/
std::chrono::milliseconds destroyWaitTime; // 空闲线程销毁等待时间
std::chrono::milliseconds submitWaitTime;  // 提交等待时间
```

2.3.2 构造与析构

- **构造函数**：初始化线程池参数，默认初始线程数为硬件核心数，线程上限为核心数的 2 倍

```
Threadpool::Threadpool(size_t initSize = std::thread::hardware_concurrency(),
                       size_t upperSize = std::thread::hardware_concurrency() *
2)
    : initThreadSize(initSize), threadUpperThresh(upperSize), ... {}
```

- **析构函数**：设置停止标志，唤醒所有线程，等待线程结束并释放资源

```
Threadpool::~Threadpool() {
    if (!stopFlag) stopFlag = true;
    threadCv.notify_all();
    for (auto &i : threads) {
        if (i.second->joinable()) i.second->join();
    }
}
```

2.3.3 线程池启动与关闭

- **启动线程池（start ()）**：根据初始线程数量创建线程，添加到线程容器中

```
void Threadpool::start() {
    for (size_t i = 0; i < initThreadSize; ++i)
        addThread(); // 调用addThread创建线程
}
```

- **关闭线程池（shutDown ()）**：设置停止标志，清空任务队列，终止所有线程

```
void Threadpool::shutDown() {
    std::lock_guard<std::mutex> lock(taskQueMtx);
    stopFlag = true;
    while (tasks.size()) tasks.pop();
}
```

2.3.4 任务提交

线程池通过模板方法submit接收任务，支持任意参数和返回值，并提供按优先级提交的接口（submitHighTsk/submitNormalTsk/submitLowTsk）。

核心流程：

1. 用std::bind绑定函数和参数，包装为std::packaged_task
2. 获取std::future用于获取任务返回值
3. 将任务转换为std::function<void()>，封装为 Task 对象
4. 等待任务队列有空闲位置（超时则抛出异常），将任务入队
5. 若为 MODE_CACHED 模式且需要扩展线程，则创建新线程
6. 唤醒一个等待的工作线程执行任务

关键代码：

```
template <typename F, typename... Args>
```

```
auto submit(Priority prio, F &&func, Args &&...args) ->
std::future<decltype(std::declval<F>()(std::declval<Args>()...))> {
    // 绑定函数与参数，包装为packaged_task
    auto boundFunc = std::bind(std::forward<F>(func), std::forward<Args>
(args)...);
    auto taskPackage = std::make_shared<std::packaged_task<ReturnType()>>
(boundFunc);
    std::future<ReturnType> resultFuture = taskPackage->get_future();

    // 转换为function<void()>并封装为Task
    std::function<void()> taskFunc = [taskPackage]() { (*taskPackage)(); };

    // 任务入队（带超时等待）
    std::unique_lock<std::mutex> lock(taskQueMtx);
    if (!taskCv.wait_for(lock, submitWaitTime, [this]() {
        return tasks.size() < taskUpperThresh && !stopFlag;
    })) {
        throw std::runtime_error("Task submission failed");
    }
    tasks.emplace(prio, taskFunc);

    // 动态扩展线程（MODE_CACHED模式）
    if (poolMode == PoolMode::MODE_CACHED && idleThreadSize == 0 &&
        getCurTskSize() > 0 && getCurThdSize() < threadUpperThresh) {
        addThread();
    }

    lock.unlock();
    threadCv.notify_one(); // 唤醒工作线程
    return resultFuture;
}
```

2.3.5 工作线程逻辑

工作线程（workThread）是线程池的核心执行单元，负责循环获取并执行任务，逻辑如下：

1. 循环等待任务队列非空或线程池停止
2. MODE_CACHED 模式下，若任务队列为空且线程数超过初始值，超时后销毁当前线程
3. 从任务队列获取最高优先级任务，执行任务
4. 任务执行完成后，更新空闲线程数，通知任务提交者

关键代码：

```
void Threadpool::workThread() {
    while (true) {
        std::unique_lock<std::mutex> lock(taskQueMtx);
        // MODE_CACHED模式下的超时销毁逻辑
        if (poolMode == PoolMode::MODE_CACHED && tasks.empty() &&
            getCurThdSize() > initThreadSize && idleThreadSize > 0 || stopFlag)
{
            if (!threadCv.wait_for(lock, destroyWaitTime, [this]() {
                return !tasks.empty() || stopFlag;
            }) || stopFlag) {
                --idleThreadSize;
                threads.erase(std::this_thread::get_id()); // 销毁当前线程
                return;
            }
```

```
        } else {
            // 等待任务或停止信号
            threadCv.wait(lock, [this]() { return !tasks.empty() || stopFlag;
});
        }

        // 执行任务
        --idleThreadSize;
        if (tasks.empty()) {
            ++idleThreadSize;
            continue;
        }
        auto taskFunc = tasks.top().getFunc();
        tasks.pop();
        lock.unlock();
        taskFunc(); // 执行任务
        lock.lock();
        ++idleThreadSize;
        lock.unlock();
        taskCv.notify_one(); // 通知任务提交者队列有空闲
    }
}
```

2.3.6 线程动态管理（MODE_CACHED 模式）

- **线程创建（addThread ()）**：创建新线程并添加到线程容器，递增空闲线程数

```
void Threadpool::addThread() {
    ++idleThreadSize;
    auto t = std::make_unique<std::thread>(&Threadpool::workThread, this);
    threads.emplace(t->get_id(), std::move(t));
}
```

- **线程销毁**：工作线程在超时等待后自动从线程容器中移除并终止（见workThread逻辑）

2.3.7 配置参数调整

线程池提供一系列接口调整核心参数，适应不同场景需求：

- setPoolMode：切换线程池模式（MODE_FIXED/MODE_CACHED）
- setThreadUpperThresh：设置 MODE_CACHED 模式下线程上限
- setTaskUpperThresh：设置任务队列最大容量
- setDestroyWaitTime：设置空闲线程销毁等待时间
- setSubmitWaitTime：设置任务提交超时时间

实现逻辑：当线程池处于启动状态就不可修改

```
void Threadpool::setPoolMode(PoolMode mode) // 设置线程模式
{
    if (stopFlag)
        return;
    poolMode = mode;
}

void Threadpool::setInitThreadSize(const size_t &size) // 设置初始线程数量
{
```

```
    if (stopFlag)
        return;
    initThreadSize = size;
}
void Threadpool::setThreadUpperThresh(const size_t &size) // 设置cache模式下线程上限
阈值
{
    if (stopFlag)
        return;
    threadUpperThresh = size;
}
void Threadpool::setTaskUpperThresh(const size_t &size) // 设置任务上限阈值
{
    if (stopFlag)
        return;
    taskUpperThresh = size;
}
void Threadpool::setDestroyWaitTime(const size_t &time) // 空闲线程销毁等待时间
{
    if (stopFlag)
        return;
    destroyWaitTime = std::chrono::milliseconds(time);
}
void Threadpool::setSubmitWaitTime(const size_t &time) // 设置提交等待时间
{
    if (stopFlag)
        return;
    submitWaitTime = std::chrono::milliseconds(time);
}
```

2.3.8 部分成员变量状态获取

- getCurThdSize: 获取当前线程数量
- getIdleThdSize: 获取当前空闲线程数量
- getCurTskSize: 获取当前任务队列的任务数量

```
const size_t Threadpool::getCurThdSize() const // 获取当前线程数量
{
    return threads.size();
}

const size_t Threadpool::getCurTskSize() const // 获取当前任务数量
{

    return tasks.size();
}

const size_t Threadpool::getIdleThdSize() const // 获取空闲线程数量
{

    return this->idleThreadSize;
}
```

2.3.9 完整代码

threadpool .h声明部分

```
class Threadpool
{
public:
    Threadpool(size_t, size_t);
    ~Threadpool();
    // 拷贝函数与拷贝赋值函数删除
    Threadpool(const Threadpool &) = delete;
    Threadpool &operator=(const Threadpool &) = delete;

    void start(); // 启动线程

    const size_t getCurThdSize() const;               // 获取当前线程数量
    const size_t getIdleThdSize() const;              // 获取当前空闲线程数量
    const size_t getCurTskSize() const;               // 获取当前任务数量
    void setPoolMode(const PoolMode);           // 设置线程模式
    void shutDown();                            // 关闭线程池
    void setInitThreadSize(const size_t &);     // 设置初始线程数量
    void setThreadUpperThresh(const size_t &); // 设置cache模式下线程上限阈值
    void setTaskUpperThresh(const size_t &);    // 设置任务上限阀值
    void setDestroyWaitTime(const size_t &);    // 空闲线程销毁等待时间
    void setSubmitWaitTime(const size_t &);     // 设置提交等待时间

    template <typename F, typename... Args>
    auto submit(Priority prio, F &&func, Args &&...args)
        -> std::future<decltype(std::declval<F>()(std::declval<Args>()...))>
    {
        // 定义返回类型别名
        using ReturnType = decltype(std::declval<F>()(std::declval<Args>()...));

        // 包装任务
        // 将函数与参数用bind绑定
        auto boundFunc = std::bind(std::forward<F>(func), std::forward<Args>
(args)...);
        // 将package_task包装进shared_ptr,便于转换为function<void()>进行捕获,
function类型需要函数可拷贝
        std::shared_ptr<std::packaged_task<ReturnType()>> taskPackage =
std::make_shared<std::packaged_task<ReturnType()>>(boundFunc);
        // 获取future
        std::future<ReturnType> resultFuture = taskPackage->get_future();
        // 转换为function<void()>
        std::function<void()> taskFunc = [taskPackage]()
        { (*taskPackage)(); };

        // 任务入队
        std::unique_lock<std::mutex>
            lock(taskQueMtx);
        // 等待超过1s视为提交失败
        if (!this->taskCv.wait_for(lock, submitWaitTime, [this]() -> bool
                                   { return this->tasks.size() < this-
>taskUpperThresh && !this->stopFlag; }))
            throw std::runtime_error("Task submission failed, Please try again
later.");
```

```
        tasks.emplace(prio, taskFunc); // 入队
        // 若线程池处于cache模式,且没有空闲线程，任务队列有任务，当前线程量小于最大线程数时，
增加新线程
        if (poolMode == PoolMode::MODE_CACHED && idleThreadSize == 0 && 0 <
getCurTskSize() && getCurThdSize() < threadUpperThresh)
            addThread();

        lock.unlock();
        threadCv.notify_one();

        return resultFuture;
    }

    template <typename F, typename... Args>
    auto submitLowTsk(F &&func, Args &&...args) // 低优先级提交任务
        -> std::future<decltype(std::declval<F>()(std::declval<Args>()...))>
    {
        return submit(Priority::LOW, func, args...);
    }

    template <typename F, typename... Args>
    auto submitNormalTsk(F &&func, Args &&...args) // 普通优先级提交任务
        -> std::future<decltype(std::declval<F>()(std::declval<Args>()...))>
    {
        return submit(Priority::NORMAL, func, args...);
    }

    template <typename F, typename... Args>
    auto submitHighTsk(F &&func, Args &&...args) // 高优先级提交任务
        -> std::future<decltype(std::declval<F>()(std::declval<Args>()...))>
    {
        return submit(Priority::HIGH, func, args...);
    }

private:
    void workThread(); // 工作线程函数
    void addThread();  // 添加线程

private:
    /*线程相关变量*/
    std::unordered_map<std::thread::id, std::unique_ptr<std::thread>> threads;
// 线程哈希表
    size_t initThreadSize;
// 初始线程数量
    size_t threadUpperThresh;
 // cache模式下线程上限阈值
    std::atomic<size_t> idleThreadSize;
 // 当前空闲线程数量

    /*任务相关变量*/
    std::priority_queue<Task> tasks; // 任务队列，默认大顶堆
    size_t taskUpperThresh;          // 任务上限阈值

    /*线程互斥锁与条件变量*/
    mutable std::mutex taskQueMtx;    // 保证任务队列的线程安全,mutable确保可以在const
内修改
    std::condition_variable taskCv;   // 任务条件变量
    std::condition_variable threadCv; // 线程条件变量
```

```
    PoolMode poolMode;           // 当前线程池的工作模式
    std::atomic<bool> stopFlag; // 线程池停止标志

    /*时间变量*/
    std::chrono::milliseconds destroyWaitTime; // 空闲线程销毁等待时间
    std::chrono::milliseconds submitWaitTime;  // 提交等待时间

    /*线程管理器*/
    std::unique_ptr<std::thread> threadManager;
};
```

threadpool.cpp实现部分

```
/*threadpool类*/
// 构造函数
Threadpool::Threadpool(size_t initSize = std::thread::hardware_concurrency(),
size_t upperSize = std::thread::hardware_concurrency() * 2)
    : initThreadSize(initSize),
      threadUpperThresh(upperSize),
      idleThreadSize(0),
      taskUpperThresh(maxTaskSize),
      poolMode(PoolMode::MODE_FIXED),
      stopFlag(false),
      destroyWaitTime(destroyTime),
      submitWaitTime(subTime)

{
}
Threadpool::~Threadpool()
{
    if (!stopFlag)
        stopFlag = true;
    this->threadCv.notify_all();

    for (auto i = this->threads.begin(); i != this->threads.end(); ++i)
    {
        if (i->second->joinable())
            i->second->join();
    }
}

void Threadpool::start() // 启动线程池
{
    for (size_t i = 0; i < initThreadSize; ++i)
        addThread();
}

const size_t Threadpool::getCurThdSize() const // 获取当前线程数量
{
    return threads.size();
}

const size_t Threadpool::getCurTskSize() const // 获取当前任务数量
{

    return tasks.size();
```

```
}

const size_t Threadpool::getIdleThdSize() const // 获取空闲线程数量
{

    return this->idleThreadSize;
}
void Threadpool::shutDown() // 关闭线程
{
    std::lock_guard<std::mutex> lock(taskQueMtx);
    this->stopFlag = true;
    while (tasks.size())
        tasks.pop();
}
void Threadpool::setPoolMode(PoolMode mode) // 设置线程模式
{
    if (stopFlag)
        return;
    poolMode = mode;
}

void Threadpool::setInitThreadSize(const size_t &size) // 设置初始线程数量
{
    if (stopFlag)
        return;
    initThreadSize = size;
}
void Threadpool::setThreadUpperThresh(const size_t &size) // 设置cache模式下线程上限
阈值
{
    if (stopFlag)
        return;
    threadUpperThresh = size;
}
void Threadpool::setTaskUpperThresh(const size_t &size) // 设置任务上限阈值
{
    if (stopFlag)
        return;
    taskUpperThresh = size;
}
void Threadpool::setDestroyWaitTime(const size_t &time) // 空闲线程销毁等待时间
{
    if (stopFlag)
        return;
    destroyWaitTime = std::chrono::milliseconds(time);
}
void Threadpool::setSubmitWaitTime(const size_t &time) // 设置提交等待时间
{
    if (stopFlag)
        return;
    submitWaitTime = std::chrono::milliseconds(time);
}

void Threadpool::addThread() // 增加线程函数，在非start()函数中需要在锁中
{
    ++idleThreadSize;
    std::unique_ptr<std::thread> t = std::make_unique<std::thread>
(&Threadpool::workThread, this);
```

```
    threads.emplace(t->get_id(), std::move(t));
}
void Threadpool::workThread() // 线程函数
{

    while (true)
    {
        std::unique_lock<std::mutex> lock(taskQueMtx);
        if (poolMode == PoolMode::MODE_CACHED && tasks.empty() &&
getCurThdSize() > initThreadSize && idleThreadSize > 0 || stopFlag)
        {
            if (!threadCv.wait_for(lock, destroyWaitTime, [this]() -> bool
                                   { return !this->tasks.empty() || this-
>stopFlag; }) ||
                stopFlag)
            {
                --idleThreadSize;
                threads.erase(std::this_thread::get_id());
                return;
            }
        }
        else
        {
            threadCv.wait(lock, [this]() -> bool
                          { return !this->tasks.empty() || this->stopFlag; });
        }
        --idleThreadSize;
        if (tasks.empty())
        {
            ++idleThreadSize;
            continue;
        }

        std::function<void()> taskFunc = tasks.top().getFunc();
        tasks.pop();
        lock.unlock();
        taskFunc();
        lock.lock();
        ++idleThreadSize;
        lock.unlock();
        taskCv.notify_one();
    }
}
```

2.4 线程安全保障

- 任务队列操作（入队 / 出队）通过taskQueMtx互斥锁保护
- 线程数量、空闲线程数等共享变量使用std::atomic确保原子操作
- 条件变量taskCv和threadCv用于线程间同步，避免忙等
- 所有修改共享状态的操作（如线程创建 / 销毁、任务数量变化）均在锁保护下进行

三、线程池改进

3.1 线程管理函数

线程管理函数的任务：

- 在任务队列不为空时开辟新线程
- 在任务队列为空时销毁空闲线程

改进方向：

- 新开辟俩个线程用于管理线程，一个线程用于增加线程，一个用于销毁线程
- 当提交任务时唤醒增加线程的线程，销毁线程的线程定期检查线程队列，销毁空闲线程
- 修改新增线程和销毁线程的条件，使其更合理

具体流程：

新增线程：

1. 获取锁
2. 检查是否满足“任务队列不为空，且当前线程数量小于线程数量上限”的条件，是则增加线程，否则阻塞线程
3. 通知一个执行任务的线程
4. 若停止标准不为true返回1循环

销毁线程：

1. 获取锁
2. 销毁线程
3. 休眠线程指定时间
4. 若停止标准不为true返回1循环

代码实现：

```
void Threadpool::addManagerThread()// 新增线程的线程管理函数
{
    while (!stopFlag)
    {
        {
            std::unique_lock<std::mutex> lock(mtx);
            addThreadCv.wait(lock, [this]()
                             { return this->getCurThdSize() < this-
>threadUpperThresh || stopFlag; });
            if (stopFlag)
                return;
            addThread();
        }
        executeTaskCv.notify_one();
    }
}
void Threadpool::subManagerThread()// 销毁线程的线程管理函数
{
    while (!stopFlag)
    {
        {
            std::unique_lock<std::mutex> lock(mtx);
            if (stopFlag)
                return;
            subThread();
        }
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }
}
```

addThread()

```
void Threadpool::addThread() // 增加线程函数，需在锁中运行
{
    if (getCurThdSize() < threadUpperThresh)
    {
        std::thread t(&Threadpool::workThread, this);
        std::thread::id thread_id = t.get_id();
        threads.emplace(thread_id, std::move(t));
        std::cout << "新增线程 :" << std::setw(2) << thread_id << " 当前线程数 :" <<
std::setw(2) << getCurThdSize() << std::endl;
    }
}
```

3.2 其他细节处理

3.2.1 线程管理器

threadpool类增加私有成员addThreadManager、subThreadManger，类型为智能指针，用于指向执行线程管理函数的线程

```
/*线程管理器*/
std::unique_ptr<std::thread> addThreadManager;
std::unique_ptr<std::thread> subThreadManager;
```

3.2.2 start函数

在初始化初始线程的同时，若线程池处于cache模式就开辟新线程执行线程管理函数

```
void Threadpool::start() // 启动线程池
{
    for (size_t i = 0; i < initThreadSize; ++i)
        addThread();
    if (poolMode == PoolMode::MODE_CACHED)
    {
        addThreadManager = std::unique_ptr<std::thread>(new
std::thread(&Threadpool::addManagerThread, this));
        subThreadManager = std::unique_ptr<std::thread>(new
std::thread(&Threadpool::subManagerThread, this));
    }
}
```

3.2.3 submit函数

去除提交任务新增线程的代码

```
auto submit(Priority prio, F &&func, Args &&...args)
        -> std::future<decltype(std::declval<F>()(std::declval<Args>()...))>
```

```
    {
        // 定义返回类型别名
        using ReturnType = decltype(std::declval<F>()(std::declval<Args>()...));

        // 包装任务
        // 将函数与参数用bind绑定
        auto boundFunc = std::bind(std::forward<F>(func), std::forward<Args>
(args)...);
        // 将package_task包装进shared_ptr,便于转换为function<void()>进行捕获,
function类型需要函数可拷贝
        std::shared_ptr<std::packaged_task<ReturnType()>> taskPackage =
std::make_shared<std::packaged_task<ReturnType()>>(boundFunc);
        // 获取future
        std::future<ReturnType> resultFuture = taskPackage->get_future();
        // 转换为function<void()>
        std::function<void()> taskFunc = [taskPackage]()
        { (*taskPackage)(); };

        {
            // 任务入队
            std::unique_lock<std::mutex> lock(mtx);
            // 等待超过1s视为提交失败
            if (!this->submitTaskCv.wait_for(lock, submitWaitTime, [this]() ->
bool
                                             { return this->tasks.size() < this-
>taskUpperThresh && !this->stopFlag; }))
                throw std::runtime_error("Task submission failed, Please try
again later.");

            tasks.emplace(prio, taskFunc); // 入队
        }

        executeTaskCv.notify_one();
        addThreadCv.notify_one();

        return resultFuture;
    }
```

3.2.4 workthread函数

去除自我销毁函数，并修改函数逻辑，使其更简洁

```
void Threadpool::workThread() // 线程函数
{

    while (true)
    {
        std::unique_lock<std::mutex> lock(taskQueMtx);
        if ((poolMode == PoolMode::MODE_CACHED && tasks.empty() &&
getCurThdSize() > initThreadSize && idleThreadSize > 0) || stopFlag)
        {
            if (!threadCv.wait_for(lock, destroyWaitTime, [this]() -> bool
                                   { return !this->tasks.empty() || this-
>stopFlag; }) ||
                stopFlag)
            {
```

```
                lock.unlock(); // 空闲前释放锁
                threads.at(std::this_thread::get_id())->detach();
                return; // 线程内函数停止运行交由线程管理器管理
            }
        }
        else
        {
            threadCv.wait(lock, [this]() -> bool
                           { return !this->tasks.empty() || this->stopFlag; });
        }
        --idleThreadSize;
        if (tasks.empty())
        {
            ++idleThreadSize;
            continue;
        }

        std::function<void()> taskFunc = tasks.top().getFunc();
        tasks.pop();
        lock.unlock();
        taskFunc();
        lock.lock();
        ++idleThreadSize;
        lock.unlock();
        taskCv.notify_one();
    }
}
```

3.2.5 Threadpool析构函数

增加线程管理器的join()操作

```
Threadpool::~Threadpool()
{
    if (!stopFlag)
        stopFlag = true;
    addThreadCv.notify_all();
    subThreadCv.notify_all();
    executeTaskCv.notify_all();

    if (addThreadManager->joinable())
        addThreadManager->join();
    if (subThreadManager->joinable())
        subThreadManager->join();

    for (auto i = this->threads.begin(); i != this->threads.end(); ++i)
    {
        if (i->second.second.joinable())
            i->second.second.join();
    }
}
```

3.2.6修改并增加互斥锁、条件变量

修改互斥锁名称为mtx，删除原有条件变量，新增四个条件变量，防止线程之间的通信混乱

```
std::condition_variable submitTaskCv;  // 提交任务条件变量
std::condition_variable executeTaskCv; // 执行任务条件变量
std::condition_variable addThreadCv;   // 增加线程条件变量
std::condition_variable subThreadCv;   // 减少线程条件变量
```

3.2.7删除getIdleThreadSize()函数

私有函数已有idleThreadSize成员变量，无需令设计成员函数获取

3.2.8 增加新容器

增加用于存储空闲线程id的vector

```
std::vector<std::thread::id> idleThreadId;                  // 空闲线程id队列
```

四、测试线程池

4.1 测试目的

本测试旨在验证自定义线程池（`Threadpool`）的并发处理能力、缓存模式（`MODE_CACHED`）下的任务调度效率，以及与单线程处理相同任务的性能差异，从而评估线程池在多任务场景下的实际效果。

测试代码：

```
#include "../include/threadpool.h"
#include <iostream>

// 一个耗时更长的任务
long long heavy_task(int n)
{
    long long sum = 0;
    // 关键：用固定次数的循环放大任务耗时，避免溢出/编译器优化导致的“耗时不准”
    for (int i = 0; i < n; ++i)
    {
        sum *= i; // 简单计算，避免编译器优化掉循环
    }
    return sum;
}

int main()
{
    std::vector<std::future<long long>> vr;
    Threadpool threadpool(2, 8);
    threadpool.setPoolMode(PoolMode::MODE_CACHED);
    threadpool.start();

    auto start = std::chrono::system_clock::now();
    /*第一次任务提交*/
    std::cout << "第一次任务提交 : " << std::endl;
```

```
    // 提交1000个“大”任务
    for (int i = 0; i < 10000; ++i)
    {
        vr.push_back(threadpool.submitNormalTsk(heavy_task, 1000000)); // 计算一个
大数的阶乘
    }

    // 等待所有任务完成
    for (auto &fut : vr)
    {
        fut.get();
    }
    vr.clear();
    auto end = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_seconds = end - start;
    std::cout << "线程池总耗时: " << elapsed_seconds.count() << "s\n";

    std::this_thread::sleep_for(std::chrono::milliseconds(45 * 1000));

    /*第二次任务提交*/
    start = std::chrono::system_clock::now();
    std::cout << "第二次任务提交 : " << std::endl;
    for (int i = 0; i < 10000; ++i)
    {
        vr.push_back(threadpool.submitNormalTsk(heavy_task, 1000000)); // 计算一个
大数的阶乘
    }

    // 等待所有任务完成
    for (auto &fut : vr)
    {
        fut.get();
    }
    end = std::chrono::system_clock::now();
    elapsed_seconds = end - start;

    std::cout << "线程池总耗时: " << elapsed_seconds.count() << "s\n";

    threadpool.shutDown();

    /*--- 对比单线程 ---*/
    start = std::chrono::system_clock::now();
    for (int i = 0; i < 10000; ++i)
    {
        heavy_task(1000000);
    }
    end = std::chrono::system_clock::now();
    elapsed_seconds = end - start;
    std::cout << "单线程总耗时: " << elapsed_seconds.count() << "s\n";

    std::this_thread::sleep_for(std::chrono::milliseconds(20 * 1000));

    return 0;
}
```

4.2 测试环境

- 编程语言：C++（基于 C++11标准，依赖线程、future 等并发库）
- 线程池核心参数：核心线程数 2，最大线程数 16，工作模式为缓存模式（`MODE_CACHED`）
- 测试任务：`heavy_task`函数（通过固定次数的循环计算模拟耗时操作，避免编译器优化导致的耗时失真）

4.3 测试步骤

4.3.1 线程池初始化与配置

1. 创建线程池实例，初始化核心线程数为 2、最大线程数为 16；
2. 设置线程池工作模式为`MODE_CACHED`（缓存模式，可根据任务量动态调整线程数量，空闲线程会超时回收）；
3. 启动线程池，完成内部线程的创建与初始化。

4.3.2 第一次任务提交测试

1. 记录任务提交开始时间；
2. 向线程池提交 10000 个`heavy_task`任务，每个任务执行 1000000 次循环计算（`heavy_task(1000000)`）；
3. 等待所有任务执行完成（通过`std::future::get()`阻塞等待）；
4. 记录任务全部完成的时间，计算总耗时并输出。

4.3.3 线程池空闲状态测试

在第一次任务完成后，通过`std::this_thread::sleep_for`让主线程休眠 60 秒，模拟线程池的空闲状态，用于验证缓存模式下线程的超时回收机制对后续任务的影响。

4.3.4 第二次任务提交测试

1. 休眠结束后，记录第二次任务提交开始时间；
2. 再次向线程池提交 10000 个相同的`heavy_task`任务（`heavy_task(1000000)`）；
3. 等待所有任务执行完成，记录总耗时并输出，对比第一次任务耗时，分析线程池在经历空闲后的处理效率。

4.3.5 单线程对比测试

1. 关闭线程池后，记录单线程任务开始时间；
2. 在主线程中串行执行 10000 个`heavy_task(1000000)`任务；
3. 记录总耗时并输出，与线程池的两次任务耗时对比，验证多线程并发的优势。

4.4 测试结果

终端输出：

```
第一次任务提交 ：
线程池总耗时: 1.52017s
第二次任务提交 ：
线程池总耗时: 1.6035s
单线程总耗时: 18.1997s
```

使用该线程池后执行速度显著提升，同样的工作量时间由18.1997s->1.52017s

调试工作台输出：

```
[New Thread 17392.0x18a0]
[New Thread 17392.0x6a8c]
[New Thread 17392.0x1060]
[New Thread 17392.0x2490]
[New Thread 17392.0x5eb0]
[New Thread 17392.0xd30]
[New Thread 17392.0x6700]
[New Thread 17392.0x5660]
[New Thread 17392.0x4544]
[New Thread 17392.0x4ee8]
[New Thread 17392.0x5f30]
[New Thread 17392.0x69ec]
[New Thread 17392.0x5b58]
[New Thread 17392.0x35c]
[New Thread 17392.0x700]
[New Thread 17392.0x1e24]
[New Thread 17392.0x52a4]
[New Thread 17392.0x2b84]
[Thread 17392.0x5eb0 exited with code 0]
[Thread 17392.0x52a4 exited with code 0]
[Thread 17392.0x6a8c exited with code 0]
[Thread 17392.0x5f30 exited with code 0]
[Thread 17392.0x5b58 exited with code 0]
[Thread 17392.0xd30 exited with code 0]
[Thread 17392.0x4ee8 exited with code 0]
[Thread 17392.0x18a0 exited with code 0]
[Thread 17392.0x5660 exited with code 0]
[Thread 17392.0x69ec exited with code 0]
[Thread 17392.0x6700 exited with code 0]
[Thread 17392.0x1e24 exited with code 0]
[Thread 17392.0x700 exited with code 0]
[Thread 17392.0x2b84 exited with code 0]
[New Thread 17392.0x18c4]
[New Thread 17392.0x5f48]
[New Thread 17392.0x3a9c]
[New Thread 17392.0x66ac]
[New Thread 17392.0x6ba8]
[New Thread 17392.0x6204]
[New Thread 17392.0x6f0c]
[New Thread 17392.0x47b0]
[New Thread 17392.0x5e54]
[New Thread 17392.0x41cc]
[New Thread 17392.0x635c]
[New Thread 17392.0x436c]
[New Thread 17392.0x529c]
[New Thread 17392.0x14a0]
[Thread 17392.0x2490 exited with code 0]
[Thread 17392.0x635c exited with code 0]
[Thread 17392.0x66ac exited with code 0]
[Thread 17392.0x5f48 exited with code 0]
[Thread 17392.0x14a0 exited with code 0]
[Thread 17392.0x6ba8 exited with code 0]
[Thread 17392.0x6204 exited with code 0]
[Thread 17392.0x35c exited with code 0]
[Thread 17392.0x47b0 exited with code 0]
```

```
[Thread 17392.0x6f0c exited with code 0]
[Thread 17392.0x529c exited with code 0]
[Thread 17392.0x41cc exited with code 0]
[Thread 17392.0x436c exited with code 0]
[Thread 17392.0x18c4 exited with code 0]
[Thread 17392.0x4544 exited with code 0]
[Thread 17392.0x3a9c exited with code 0]
[Thread 17392.0x5e54 exited with code 0]
[Thread 17392.0x1060 exited with code 0]
[Thread 17392.0x50d8 exited with code 0]
[Thread 17392.0x4c78 exited with code 0]
[Thread 17392.0x62a0 exited with code 0]
[Inferior 1 (process 17392) exited normally]
```

在提交第一次任务后线程池在cache模式下除了初始的2个线程和2个管理线程，另开辟了14个新线程；

第一次任务执行完毕后销毁了14个空闲线程,执行第二次任务时又开辟了14个新线程；

最终销毁全部线程。

4.5 测试结论

通过上述测试，可验证自定义线程池在缓存模式下能够有效提升多任务处理效率，相比单线程具有明显的性能优势，且能根据任务量动态调整资源，适合处理突发的大量耗时任务场景。