C++多线程的用法

编译命令如下：

```
g++ 7.thread_pool.cpp -lpthread
```

简单用法：

```
#include<iostream>
#include<thread>
using namespace std;
#define BEGINS(x) namespace x {
#define ENDS(x) }

BEGINS(thread_usage)
void func() {
    cout << "hello world" << endl;
    return ;
}

int main() {
    thread t1(func);//t1已经开始运行了
    t1.join(); // 等待t1线程结束
    return 0;
}
ENDS(thread_usage)

int main() {
    thread_usage::main();

    return 0;
}
```

所谓的多线程只不过就是指定的某一个函数为**入口函数**，的另外一套执行流程。

那么如何给入口函数传参呢？

在C++中就很简单：

```
void print(int a, int b) {
    cout << a << " " << b << endl;
    return ;
}
int main() {
    thread t2(print, 3, 4);
    t2.join();
    return 0;
}
```

多线程封装思维：在相应的功能函数内部，不要去访问全局变量，类似于一个原子操作的功能。本身就支持多线程并行。

多线程程序设计：线程功能函数和入口函数要分开。（高内聚低耦合）

能不用锁就不用锁，因为这个锁机制会给程序运行效率带来极大的影响。

实现素数统计的功能

```
bool is_prime(int x) {
    for (int i = 2; i * i <= x; i++) {
        if (x % i == 0) return false;
    }
    return true;
}
// 多线程功能函数:
int prime_count(int l, int r) {
    // 从l到r范围内素数的数量
    int ans = 0;
    for (int i = l; i <= r; i++) {
        ans += is_prime(i);
    }
    return ans;
}
// 多线程入口函数:
void worker(int l, int r, int &ret) {
    cout << this_thread::get_id() << "begin" << endl;
    ret = prime_count(l, r);
    cout << this_thread::get_id() << "done" << endl;
    return ;
}
int main() {
    #define batch 500000
    thread *t[10];
    int ret[10];
    // 开辟十个线程:
    for (int i = 0, j = 1; i < 10; i++, j += batch) {
        t[i] = new thread(worker, j, j + batch - 1, ref(ret[i]));
    }
    for (auto x : t) x->join();
    int ans = 0;
    for (auto x : ret) ans += x;
    for (c++auto x : t) delete x;
    cout << ans << endl;
    return 0;
    #undef batch
}
```

查看运行时间：

```
time ./a.out
```

加锁版：

什么是临界资源？多线程情况下，大家都能访问到的资源。

```
int ans = 0;
std::mutex m_mutex;

bool is_prime(int x) {
    for (int i = 2; i * i <= x; i++) {
        if (x % i == 0) return false;
    }
    return true;
}
// 多线程功能函数:
void prime_count(int l, int r) {
    cout << this_thread::get_id() << "begin" << endl;
    // 从l到r范围内素数的数量
    for (int i = l; i <= r; i++) {
        unique_lock<mutex> lock(m_mutex); // 临界区
        ans += is_prime(i); // ans为临界资源
        lock.unlock();
    }
    cout << this_thread::get_id() << "done" << endl;
    return ;
}
int main() {
    #define batch 500000
    thread *t[10];
    // 开辟十个线程:
    for (int i = 0, j = 1; i < 10; i++, j += batch) {
        t[i] = new thread(prime_count, j, j + batch - 1);
    }
    for (auto x : t) x->join();
    for (auto x : t) delete x;
    cout << ans << endl;
    return 0;
    #undef batch
}
```

为什么不用++ 而是用+=

因为后者是原子操作而前者不是，在多线程情况下存在覆盖写的情况。

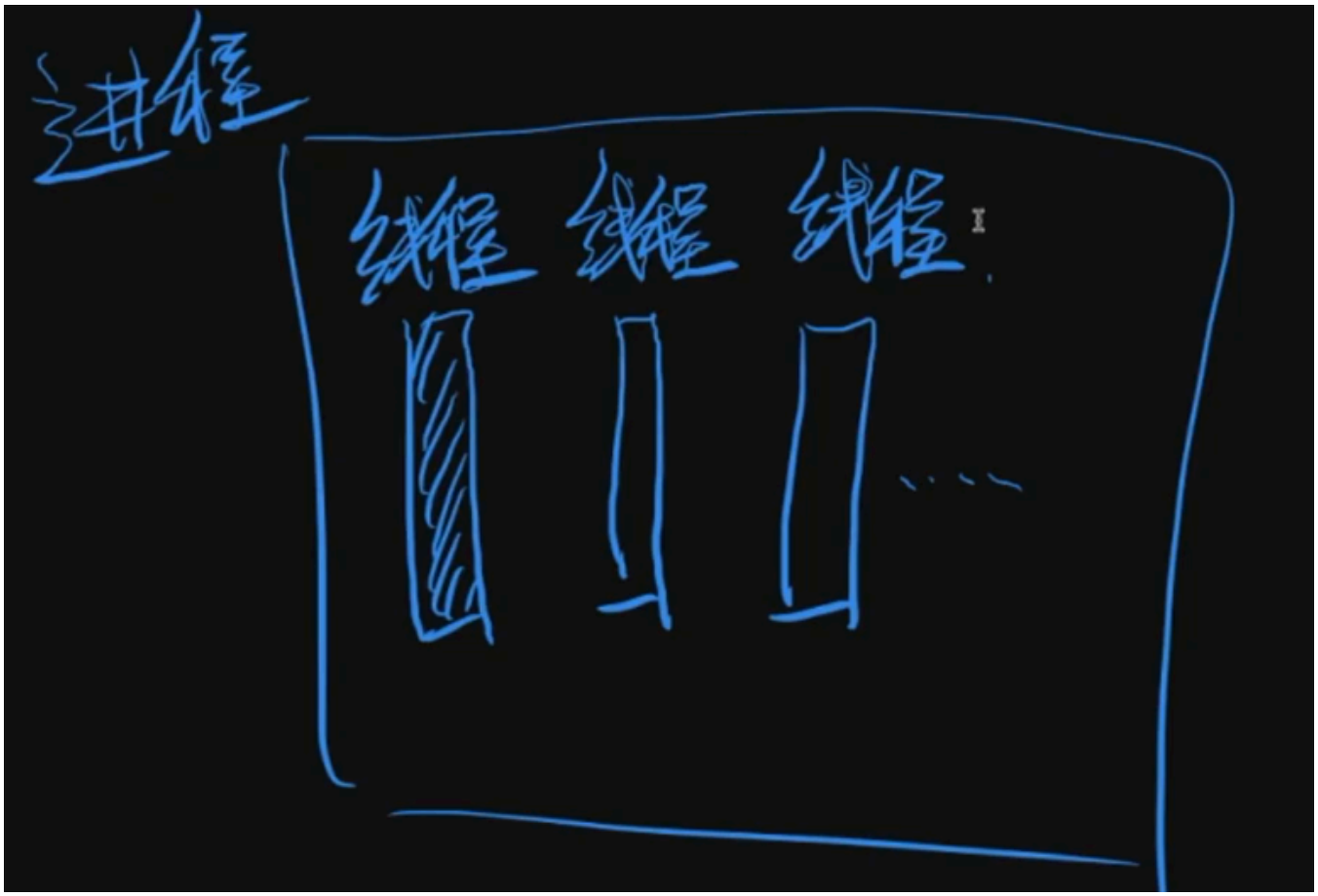
__sync_fetch_and_add

```
__sync_fetch_and_add();
// 这个函数也是原子操作
```

```
int ans = 0;
std::mutex m_mutex;

bool is_prime(int x) {
    for (int i = 2; i * i <= x; i++) {
        if (x % i == 0) return false;
    }
    return true;
}
// 多线程功能函数:
void prime_count(int l, int r) {
    cout << this_thread::get_id() << "begin" << endl;
    // 从l到r范围内素数的数量
    for (int i = l; i <= r; i++) {
        int ret = is_prime(i);
        __sync_fetch_and_add(&ans, ret);
    }
    cout << this_thread::get_id() << "done" << endl;
    return ;
}
int main() {
    #define batch 500000
    thread *t[10];
    // 开辟十个线程:
    for (int i = 0, j = 1; i < 10; i++, j += batch) {
        t[i] = new thread(prime_count, j, j + batch - 1);
    }
    for (auto x : t) x->join();
    for (auto x : t) delete x;
    cout << ans << endl;
    return 0;
    #undef batch
}
```

进程是资源分配的最基本单位，线程是进程中的概念。



线程也是操作系统分配的一批资源。一个线程的栈所占用的空间有多大？——**8M**。查看命令：

```
ulimit -a
```

我们希望申请内存的动作是可控的。线程作为一种内存资源，在通常的设计模式下，申请线程资源并不可控。

解决办法就是**线程池**。在线程池内部，线程数量是可控的。把计算任务打包，扔到一个任务队列中。

什么是计算任务？——分为过程（函数方法）和数据（函数参数），如何进行打包——bind()方法。

线程池解决了，传统的多线程程序设计中面对不同任务我们需要实现不同程序的麻烦。直接往里塞任务就行了。

资源的有效利用和管控。

结课小项目：线程池的实现

```
BEGINS(thread_pool_test)
class Task {
public:
    template<typename FUNC_T, typename ...ARGS>
    Task(FUNC_T func, ARGS ...args) {
        // 一定要用参数的原有类型进行向下传递。
        this->func = bind(func, forward<ARGS>(args)...);
    }
    void run() {
        func();
        return ;
    }
private:
    function<void()> func;
};
class ThreadPool {
public:
    ThreadPool(int n = 1) : thread_size(n), threads(n), starting(false) {
        this->start();
        return ;
    }
    void worker() {
        auto id = this_thread::get_id();
        running[id] = true;
        while (running[id]) {
            // 取任务
            Task *t = get_task(); //只是内部的实现方法
            // 执行任务
            t->run();
            delete t;
        }
        return ;
    }
    template<typename FUNC_T, typename ...ARGS>
    void add_task(FUNC_T func, ARGS ...args) {
        unique_lock<mutex> lock(m_mutex);// 访问临界资源
        tasks.push(new Task(func, forward<ARGS>(args)...));
        m_cond.notify_one();// 通知所有等待这个条件信号的线程
        return ;
    }
    void start() {
        if (starting == true) return ;
        for (int i = 0; i < thread_size; i++) {
            threads[i] = new thread(&ThreadPool::worker, this);
        }
        starting = true;
```

```
        return ;
    }
    void stop() {
        if (starting == false) return ;
        // 毒药任务，添加到任务队列末尾。
        for (int i = 0; i < threads.size(); i++) {
            this->add_task(&ThreadPool::stop_running, this);
        }
        for (int i = 0; i < threads.size(); i++) {
            threads[i]->join();
        }
        for (int i = 0; i < threads.size(); i++) {
            delete threads[i];
            threads[i] = nullptr;
        }
        starting = false;
        return ;
    }
    ~ThreadPool() {
        this->stop();
        while (!tasks.empty()) {
            delete tasks.front();
            tasks.pop();
        }
        return ;
    }
private:
    bool starting;
    int thread_size;
    Task *get_task() {
        unique_lock<mutex> lock(m_mutex);
        while (tasks.empty()) m_cond.wait(lock);
        Task *t = tasks.front();
        tasks.pop();
        return t;
    }
    std::mutex m_mutex;
    condition_variable m_cond;
    void stop_running() {
        auto id = this_thread::get_id();
        running[id] = false;
        return ;
    }
    vector<thread *> threads;
    unordered_map<decltype(this_thread::get_id()), bool> running;
    queue<Task *> tasks;//存放每一个任务对象的地址
```

```
};
bool is_prime(int x) {
    for (int i = 2; i * i <= x; i++) {
        if (x % i == 0) return false;
    }
    return true;
}
// 多线程功能函数:
int prime_count(int l, int r) {
    // 从l到r范围内素数的数量
    int ans = 0;
    for (int i = l; i <= r; i++) {
        ans += is_prime(i);
    }
    return ans;
}
// 多线程入口函数:
void worker(int l, int r, int &ret) {
    cout << this_thread::get_id() << "begin" << endl;
    ret = prime_count(l, r);
    cout << this_thread::get_id() << "done" << endl;
    return ;
}
int main() {
    #define batch 500000
    ThreadPool tp(5);
    int ret[10];
    for (int i = 0, j = 1; i < 10; i++, j += batch) {
        tp.add_task(worker, j, j + batch - 1, ref(ret[i]));
    }
    tp.stop(); //等待所有任务的结束。
    int ans = 0;
    for (auto x : ret) ans += x;
    cout << ans << endl;
    return 0;
}
ENDS(thread_pool_test)

int main() {
    // thread_usage::main();
    // prime_count_test::main();
    // prime_count_test1::main();
    // prime_count_test2::main();
    // task_test::main();
    thread_pool_test::main();
```

```
    return 0;
}
```