

并发编程与并发设计模式 (OOP)

刘知远

`liuzy@tsinghua.edu.cn`

`http://nlp.csai.tsinghua.edu.cn/~lzy/`

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 函数对象
- 智能指针与引用计数

本讲内容提要

- 并发编程
- `thread`与主从模式
- `mutex`与互斥锁模式
- `async`、`future`、`promise`与异步

并发编程

回顾

单任务 顺序执行

■ 结构化编程

- 经典编程范式
- 顺序、选择、循环三大基本结构
- 子程序、块结构来构筑代码
- 自顶向下，逐步细化的编程思路
- 程序条理清晰、逻辑流畅
- 在《程序设计基础》课程里已经充分介绍

■ 面向对象编程

- 引入类与对象概念
- 抽象、封装、继承、多态等特性
- 程序更加便于分析、设计、理解
- 本课程前半段已经充分介绍

“并发” 与 “并行”

■ 并发 (Concurrent)

- 一个时间段内几个程序都处于启动到完成之间
- 任意时刻只有一个程序在计算单元上运行
- 宏观上是同时，微观上仍是顺序执行

■ 并行 (Parallel)

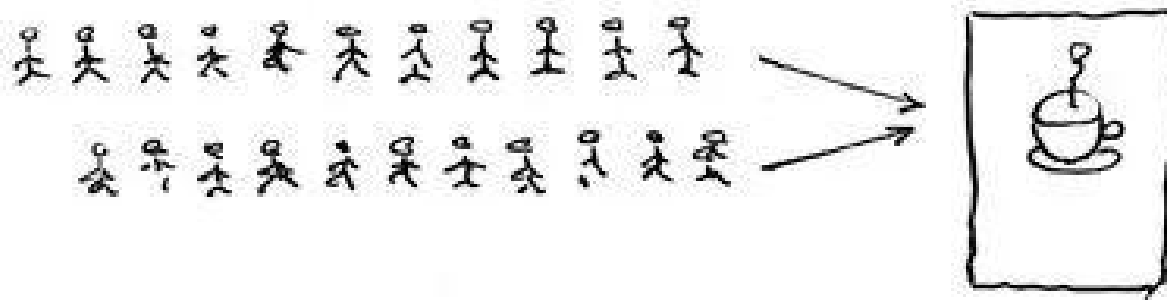
- 一个时间段内几个程序都处于启动到完成之间
- 任意时刻可以有多个程序在运行
- 宏观上是同时，微观上也可以是同时

“并发”与“并行”

■ 并发 (Concurrent)

- Joe Armstrong的例子
- 多个队列交替使用一台咖啡机就是并发行为
- 宏观上同时运行，微观上每次只有一个队伍接咖啡

Concurrent = Two Queues One Coffee Machine

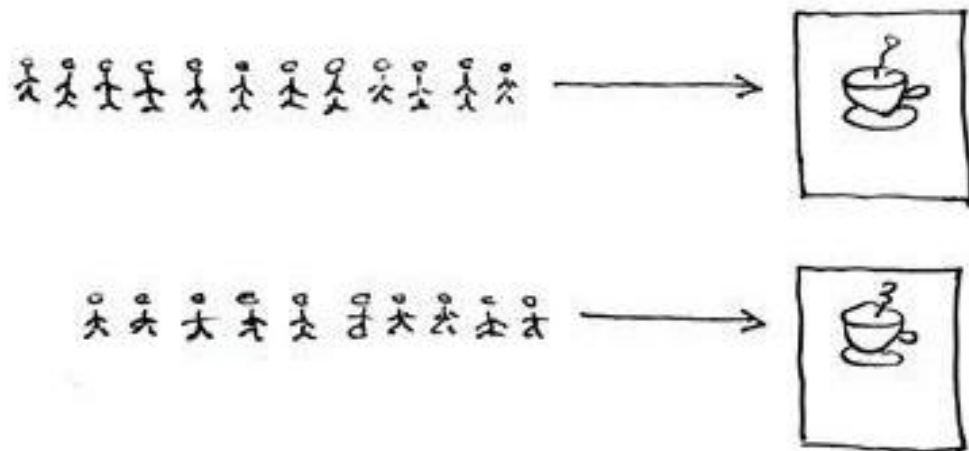


“并发”与“并行”

■ 并行 (Parallel)

- Joe Armstrong的例子
- 多个队列同时使用多台咖啡机就是**并行**行为
- 宏观上同时运行，微观上也是多个队伍同时接咖啡

Parallel = Two Queues Two Coffee Machines



“并发”与“并行”

■ 并发、并行都是多任务环境下的系统

■ 更一般的来说：

- 支持多个任务同时存在的系统，就可以认为是一个并发系统
- 如果系统还支持多个任务同时执行，就可以认为是一个并行系统
- 并发、并行的主要区别在于是否具有多个处理器同时处理多个任务

■ 出于简化，不考虑处理器数量的情况下，我们统称之为“并发”

为什么要并发编程

■ 非并发程序执行时间的影响因素

- 算法的时间复杂度
- 计算设备的性能

■ 计算设备的发展

- 经典摩尔定律不再奏效
- CPU的单核性能已经很难突破
- CPU的发展逐渐变为核的增长
- 出现GPU等高度并行的计算设备

■ 由此，掌握并发编程技术，充分发挥计算设备的运算能力，提升程序性能将是一项基本技能

“进程”与“线程”

■ 进程 (process)

- 计算机中已运行的程序

■ 线程 (thread)

- 是操作系统能够进行**运算调度**的**最小单位**
- 它被包含在进程之中，是进程中的实际运作单位。
- **一个进程**可以**并发多个线程**，设备允许的情况下，数个线程可以并行执行不同的任务
- 同一进程的多条线程**共享**该进程中的全部**系统资源**，如虚拟地址空间，文件描述符和信号处理等

“进程”与“线程”

- 程序通常会在一个进程中运行
 - 一个进程至少会包含一个线程，即“**主线程**”
- 在默认的情况下，我们的代码都是在进程的主线程中运行，除非在程序中创建了新的线程
- 系统中只有一个计算内核时
 - 所有的进程或线程会分时占用这个计算内核
- 系统中存在多个计算内核时
 - 多个进程及线程可以并行的运行在不同的计算内核上
- 我们常说的CPU核数、线程数就是指CPU的物理核心和逻辑核心数量，可以体现处理器的并行能力

“进程”与“线程”

- 进程与线程是程序运行调度的基础
- 也是并发编程的操作对象

| 进程名称 | % CPU | CPU 时间 | 线程 | 闲置唤醒 | PID | 用户 |
|-------|-------|---------|----|------|-------|---------|
| 预览 | 0.0 | 3.66 | 3 | 0 | 14258 | sillyxu |
| 通知中心 | 0.0 | 10.06 | 3 | 0 | 320 | sillyxu |
| 访达 | 0.0 | 21.90 | 5 | 0 | 287 | sillyxu |
| 聚焦 | 0.0 | 6.56 | 8 | 0 | 337 | sillyxu |
| 程序坞 | 0.1 | 19.17 | 5 | 0 | 285 | sillyxu |
| 照片代理 | 0.0 | 4.56 | 6 | 0 | 523 | sillyxu |
| 活动监视器 | 3.7 | 1.90 | 11 | 4 | 23451 | sillyxu |
| 搜狗输入法 | 0.0 | 2:38.40 | 3 | 1 | 2006 | sillyxu |
| 微信 | 0.4 | 6:19.88 | 30 | 10 | 2529 | sillyxu |
| 屏幕快照 | | 0.23 | 9 | 0 | 23454 | sillyxu |
| 仪表盘 | 0.3 | 47.35 | 5 | 1 | 11405 | sillyxu |

thread与主从模式

案例一

■ 计算 1 到 5000000 之间素数个数

■ 一般写法

```
#include <iostream>
#include <cmath>
using namespace std;
//计数器
int total = 0;
//枚举是否为素数
bool check_num(int num) {
    if (num == 1)
        return false;
    for (int i = sqrt(num); i > 1; i--)
        if (num % i == 0)
            return false;
    return true;
}

int main() {
    //枚举被检测数
    for (int i = 1; i < 5000000; i++)
        if (check_num(i))
            total++;
    cout << total << endl;
    return 0;
}
```

案例一

- 枚举法耗时较大，计算完成需要12秒

```
348513
```

```
./test 12.13s user 0.01s system 99% cpu 12.140 total
```

- 能否更快得到结果?
 - 改进算法，如使用筛法（算法层面）
 - 使用并发编程进行优化（设备层面）

thread类

■ 默认构造函数

- 定义：
 - thread() noexcept;
 - 创建一个空线程对象，不代表任何可执行的线程
- 例：

//C++11之后有了标准的线程库 thread 来进行线程操作

```
#include <thread>
```

```
using namespace std;
```

```
int main() {  
    thread s1; //创建一个空线程对象 s1  
    return 0;  
}
```

thread类

■ 初始化构造函数

- 定义

- `template <class Fn, class... Args>
explicit thread (Fn&& fn, Args&& ... args);`
- 创建线程的方式就是构造一个thread对象，并指定入口函数 fn
- 入口函数 fn 函数的参数由 args 给出
- 与普通对象创建不一样的是，这里编译器会创建一个新的操作系统线程，线程启动后会执行入口函数
- **注意：线程一旦创建，线程就开始执行**

thread类

```
The num is 0
The num is 1
./test 0.00s user 0.00s system 0% cpu 3.007 total
```

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
void test(int seconds, int num) {
    //当前线程延时seconds秒
    this_thread::sleep_for(chrono::seconds(seconds));
    cout << "The num is " << num << endl;
}
```

```
int main() {
    //创建线程对象t1, 所指向线程将执行test函数, 传入seconds为2, 传入num为1
    thread t1(test, 2, 0);
    //t2的创建与t1类似, 不再赘述
    thread t2(test, 3, 1);
    t1.join(); //关于join我们会再之后详细介绍
    t2.join();
    return 0;
}
```

t1延时2秒, t2延时3秒
顺序执行需要5秒
两个线程同时运行, 只需3秒

thread类

■ 初始化构造函数

- 与可调用对象 (Callable Objects) 一起使用
- 如 Lambda表达式

```
#include <iostream>
#include <thread>

using namespace std;

int main() {
    thread t(
        [] (int a, int b) {
            cout << a + b << endl;
        },
        1, 2);
    t.join();
    return 0;
}
```

join与detach

■ 我们之前介绍过

- 实例化thread可以进行目标线程创建
- 目标线程一旦创立就开始执行

■ 那当前线程与目标线程如何进行协调呢？

■ thread提供了两种接口来处理当前线程与目标线程

- `thread::join()`
- `thread::detach()`

join与detach

■ `thread::join()`

- 调用此接口时，当前线程会一直阻塞，直到目标线程执行完成

■ `thread::detach()`

- 调用此接口时，目标线程成为守护线程（daemon threads），将完全独立执行
- 即使目标线程对应的thread对象被销毁也不影响线程的执行

join与detach

■ thread::join()

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
```

```
void test(int seconds) {
    this_thread::sleep_for(chrono::seconds(seconds));
}
```

```
int main() {
    thread t1(test, 3);
    //t1.join()将阻塞主线程，直到t1指向的线程执行完3秒的延时
    t1.join();
    //t1指向的线程执行完成后，当前线程才能进行延时6s的操作
    this_thread::sleep_for(chrono::seconds(6));
    return 0;
}
```

0.00s user 0.00s system 0% cpu 9.015 total

目标线程3s

当前线程6s

运行共计9s

join与detach

■ thread::join()

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
```

```
void test(int seconds) {
    this_thread::sleep_for(chrono::seconds(seconds));
}
```

```
int main() {
    //t1指向的线程的延时与当前线程的延时将同时开始
    thread t1(test, 3);
    this_thread::sleep_for(chrono::seconds(6));
    //由于t1指向的线程在3秒时已完成运行，此处t1.join()无法阻塞当前线程
    t1.join();
    return 0;
}
```

0.00s user 0.00s system 0% cpu 6.009 total

目标线程3s

当前线程6s



运行共计6s

join与detach

■ `thread::detach()`

0.00s user 0.00s system 0% cpu 6.009 total

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
```

```
void test(int seconds) {
    this_thread::sleep_for(chrono::seconds(seconds));
}
```

```
int main() {
    thread t1(test, 3);
    t1.detach();
    //与t1.join()不同, t1.detach()让t1指向的线程与当前线程脱钩, 两者独立
    //当前线程的延时操作不会被阻塞
    this_thread::sleep_for(chrono::seconds(6));
    return 0;
}
```

目标线程3s

当前线程6s



运行共计6s

程序约几秒运行完成

A

约6秒

B

约8秒

C

约9秒

D

约12秒

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
void test(int seconds) {
    this_thread::sleep_for(chrono::seconds(seconds));
}
int main() {
    thread t1(test, 3);
    thread t2(test, 4);
    this_thread::sleep_for(chrono::seconds(2));
    t1.join();
    this_thread::sleep_for(chrono::seconds(1));
    t2.detach();
    this_thread::sleep_for(chrono::seconds(2));
    return 0;
}
```

提交

thread类

■ 拷贝构造函数

- 定义
 - `thread (const thread&) = delete;`
 - 拷贝构造函数(被禁用)
 - 因而 **thread 不可被拷贝构造**
 - 核心原因：线程涉及系统底层，无法拷贝

■ 移动构造函数

- 定义
 - `thread (thread&& x) noexcept;`
 - 有默认的移动构造函数
 - 调用成功之后 `x` 不指向任何具体执行线程

thread类

■ 移动构造函数

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
```

```
void test(int seconds) {
    this_thread::sleep_for(chrono::seconds(seconds));
}
```

```
int main() {
    thread t1;
    thread t2(test, 3);
    t1 = move(t2);
    t1.join();
    return 0;
}
```

move后，t1指向t2的对应目标线程，t2则不再指向任何具体执行线程

若将t1.join()替换为t2.join()，则会产生系统错误

joinable

■ `thread::joinable()`

- 可以判断thread实例指向的线程是否可以join或者detach
- 返回 `true` 表示是joinable的，可以调用`join()`或者`detach()`

■ 三种情况会使得thread实例不是joinable的

- 默认构造函数创建的实例
- 被移动构造函数操作过的实例
- 已经调用过`join()`或者`detach()`的实例

joinable

- 启动目标线程后，我们必须决定当前线程是要等待目标线程结束（join），还是让目标线程独立（detach），我们**必须二选一**
- 如果目标线程的thread对象销毁时依然没有调用join或者detach，thread对象在析构时将导致程序进程异常退出
- 换言之，thread**析构时，必须是非joinable**的状态

相关功能性接口

■ `this_thread::get_id`

- 返回当前线程的id，可以标识不同的线程

■ `this_thread::sleep_for`

- 使当前线程的执行停止一定时间
- 之前的内容中我们已经使用过了

■ `this_thread::sleep_until`

- 与sleep_for类似，以具体的时间点为参数进行停止

■ `this_thread::yield`

- 如果当前线程任务已经完成，将处理器让给其他任务使用

■ 这里与本次课程主线关系不大，留给大家课后自学

案例一

■ 在学习了thread之后，我们如何通过并发达到更高效的计算

- 将总任务[1, 5000000)进行划分，如：
 - 子任务0 [1, 1250001)
 - 子任务1 [1250001, 2500001)
 - 子任务2 [2500001, 3750001)
 - 子任务3 [3750001, 5000000)
- 每个子任务由一个线程单独解决
- 合并子任务的结果作为最终结果

案例一

■ 并发写法

```
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>
using namespace std;
thread* threads[4];
//每个线程的计数器
int thread_total[4];
//总计数器
int total = 0, mi, mx;
//枚举是否为素数
bool check_num(int num) {...}
//统计[1,r)之间的素数个数
//存入thread_total[num]中
void check(int l, int r, int num)
{
    thread_total[num] = 0;
    for (int i = l; i < r; i++)
        if (check_num(i))
            thread_total[num]++;
}
```

```
int main() {
    mi = 1;
    for (int i = 0; i < 4; i++) {
        mx = mi + 5000000 / 4;
        if (mx > 5000000) mx = 5000000;
        //为第i个线程分配[mi,mx)区间的任务
        threads[i] = new thread(check, mi, mx, i);
        mi = mx;
    }
    //阻塞主线程，等待所有子线程完成统计
    for (int i = 0; i < 4; i++)
        threads[i]->join();
    //汇总子线程的统计结果，释放thread实例
    for (int i = 0; i < 4; i++) {
        total += thread_total[i];
        delete threads[i];
    }
    //输出
    cout << total << endl;
    return 0;
}
```

案例一

- 在使用了4个线程之后，耗时由12秒降为4.5秒

```
348513  
./test 13.13s user 0.01s system 286% cpu 4.589 total
```

- 实际上，这就是一种典型的并发设计模式——主从模式

主从模式

■ 主从模式是最常用的并发设计模式

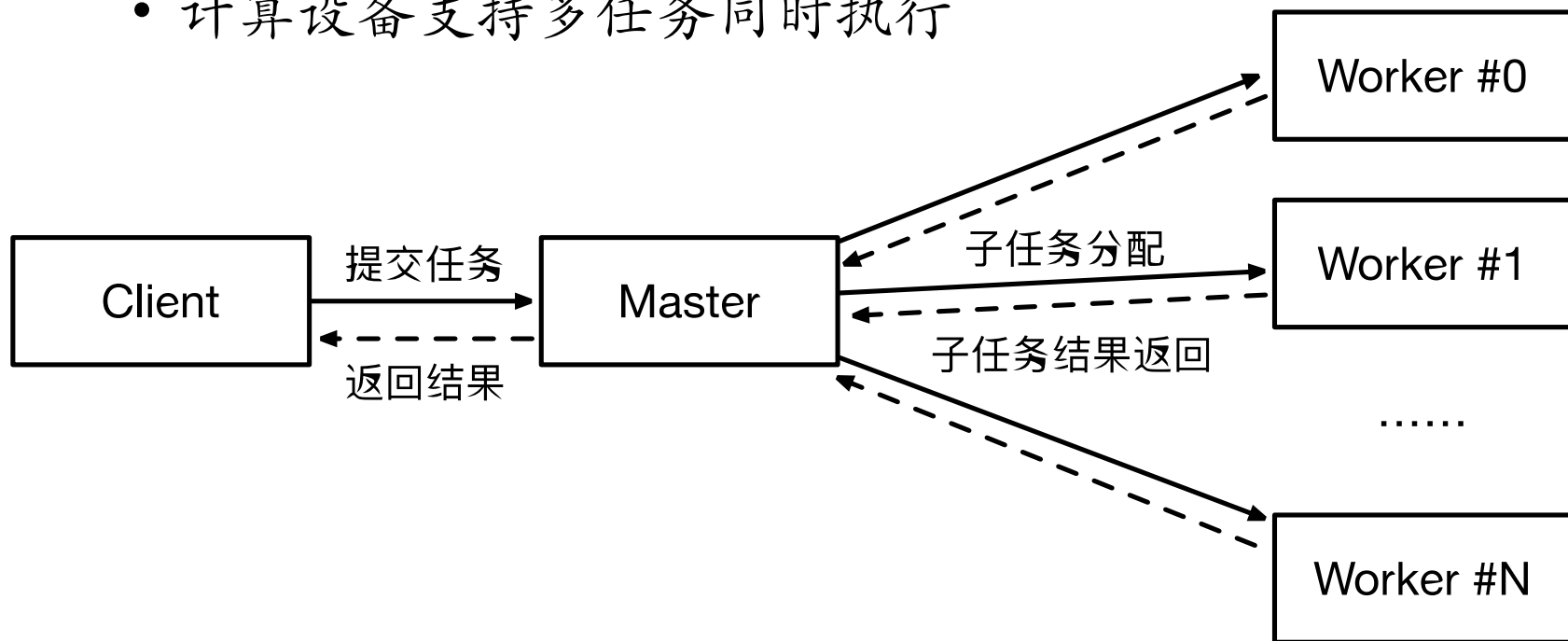
- 系统由两个角色组成，Master和Worker
- Master负责接收和分配任务
- Worker负责处理子任务
- 任务处理过程中
 - Worker负责工作
 - Master负责监督任务进展和Worker的健康状态
 - Master将接收Client提交的任务，并将任务的进展汇总反馈给Client

■ 直观来说，Client是甲方，Master是乙方老板，Worker是乙方苦工

主从模式

■ 适用场景

- 整体任务可以被划分为诸多子任务
- 子任务间关联较弱，可以并发执行
- 计算设备支持多任务同时执行



mutex与互斥锁模式

案例二

■ 回顾案例一

- 我们能否避免使用 `thread_total` 这样的复杂结构来存储子任务结果
- 能否让子线程直接将结果存入 `total`

```
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>

using namespace std;

thread* threads[4];
//每个线程的计数器
int thread_total[4];
//总计数器
int total = 0, mi, mx;
//枚举是否为素数
bool check_num(int num) {...}
//统计[1,r)之间的素数个数
//存入thread_total[num]中
void check(int l, int r, int num) {
    thread_total[num] = 0;
    for (int i = l; i < r; i++)
        if (check_num(i))
            thread_total[num]++;
}
```

案例二

■ 我们直接对程序进行修改

```
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>

using namespace std;

thread* threads[4];
//总计数器
int total = 0, mi, mx;
//枚举是否为素数
bool check_num(int num) {...}
//统计[1,r)之间的素数个数
void check(int l, int r) {
    for (int i = l; i < r; i++)
        if (check_num(i))
            total++;
}
```

```
int main() {
    mi = 1;
    for (int i = 0; i < 4; i++) {
        mx = mi + 5000000 / 4;
        if (mx > 5000000) mx = 5000000;
        //为第i个线程分配[mi,mx)区间的任务
        threads[i] = new thread(check, mi, mx);
        mi = mx;
    }
    for (int i = 0; i < 4; i++)
        threads[i]->join();
    for (int i = 0; i < 4; i++)
        delete threads[i];
    cout << total << endl;
    return 0;
}
```

案例二

■ 修改程序后

- 耗时上与之前一致
- 结果上却出现了偏差
- 多次运行的结果也不同

```
sillyxu@SHERMANHAN-MB0 > ~/Desktop > time ./test
347917
./test 12.97s user 0.01s system 287% cpu 4.514 total
sillyxu@SHERMANHAN-MB0 > ~/Desktop > time ./test
347947
./test 12.92s user 0.01s system 287% cpu 4.490 total
```

■ 原因是什么？

竞争条件与临界区

■ 竞争条件

- 多个线程同时访问共享数据时，只要有一个任务修改数据，那么就可能会发生问题——多个线程同时争相修改数据，导致部分修改没有成功。这种场景称为竞争条件 (race condition)

■ 临界区

- 访问共享数据的代码片段称为临界区 (critical section)，如之前代码中的total++;

■ 避免竞争条件需要对临界区进行数据保护

- 一次只让一个线程访问共享数据
- 访问完了再让其他线程接着访问

互斥量与锁

■ C++11 提供了互斥量(mutex)来进行数据保护

- 互斥量本身是一个类对象，一般也称为“锁”

■ 各个线程可以尝试用 mutex 的 lock() 接口来对临界区数据进行加锁

- 每次只有一个线程能够锁定成功，成功的标志是 lock() 成功返回
- 如果没锁成功，那么线程就会阻塞在那里
- 临界区就像一间只能反锁的房间，一个线程进入后反锁房间，其他线程只能等待它出来才能进入其中，并再次反锁房间

■ mutex 的 unlock() 接口可以解锁互斥量

案例二 (实现1)

■ 采用mutex

```
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>

using namespace std;

thread* threads[4];
int total = 0, mi, mx;
//互斥量
static mutex exclusive;
bool check_num(int num) {...}

void check_range(int l, int r) {
    for (int i = l; i < r; i++)
        if (check_num(i)) {
            exclusive.lock(); //加锁
            total++;
            exclusive.unlock(); //解锁
        }
}

int main() {
    mi = 1;
    for (int i = 0; i < 4; i++) {
        mx = mi + 5000000 / 4;
        if (mx > 5000000) mx = 5000000;
        //为第i个线程分配[mi,mx)区间的任务
        threads[i] = new thread(check, mi, mx);
        mi = mx;
    }
    for (int i = 0; i < 4; i++)
        threads[i]->join();
    for (int i = 0; i < 4; i++)
        delete threads[i];
    cout << total << endl;
    return 0;
}
```

案例二 (实现2)

■ 采用mutex

```
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>

using namespace std;

thread* threads[4];
int total = 0, mi, mx;
//互斥量
static mutex exclusive;
bool check_num(int num) {...}

void check_range(int l, int r) {
    int tmp_total = 0;
    for (int i = l; i < r; i++)
        if (check_num(i))
            tmp_total++;
    exclusive.lock(); //加锁
    total+=tmp_total;
    exclusive.unlock(); //解锁
}

int main() {
    mi = 1;
    for (int i = 0; i < 4; i++) {
        mx = mi + 5000000 / 4;
        if (mx > 5000000) mx = 5000000;
        //为第i个线程分配[mi,mx)区间的任务
        threads[i] = new thread(check, mi, mx);
        mi = mx;
    }
    for (int i = 0; i < 4; i++)
        threads[i]->join();
    for (int i = 0; i < 4; i++)
        delete threads[i];
    cout << total << endl;
    return 0;
}
```

两种实现谁效率更高

A

实现1

B

实现2

实现1

```
void check_range(int l, int r) {
    for (int i = l; i < r; i++)
        if (check_num(i)) {
            exclusive.lock(); //加锁
            total++;
            exclusive.unlock(); //解锁
        }
}
```

实现2

```
void check_range(int l, int r) {
    int tmp_total = 0;
    for (int i = l; i < r; i++)
        if (check_num(i))
            tmp_total++;
    exclusive.lock(); //加锁
    total+=tmp_total;
    exclusive.unlock(); //解锁
}
```

提交

案例二

- 使用互斥量mutex之后，获得了正确结果，且耗时也没有变化，成功的简化了代码

```
sillyxu@SHERMANHAN-MB0 ~/Desktop time ./test
348513
./test 13.22s user 0.02s system 286% cpu 4.624 total
```

- 这是典型的互斥锁设计模式

互斥锁模式

■ 互斥锁模式是最基本的并行数据处理模式

- 访问共享资源之前进行加锁操作
- 访问完成之后进行解锁操作
- 加锁后，任何其他试图再次加锁的线程会被阻塞，直到当前线程解锁
- 如果解锁时有一个以上的线程阻塞，那么所有阻塞的线程都会尝试变成就绪状态，第一个变为就绪状态的线程继续执行加锁操作，其他线程继续进入阻塞状态等待下次加锁

■ 互斥锁模式的弊端

- 低效——共享资源的读操作往往并不需要互斥
- 解决方案——采用**读写锁模式**，读是共享，写是互斥

其他互斥量

■ C++11之后提供了丰富的各类互斥量

- `timed_mutex`
 - 带有超时功能，在时间范围内没能获取到锁，则直接返回，不再继续等待
- `recursive_mutex`
 - 能被同一线程递归锁定的互斥量，即在同一个线程中，同一把锁可以锁定多次
- `recursive_timed_mutex`
 - 能被同一线程递归锁定的互斥量，且带有超时功能
- `shared_mutex`
 - 共享互斥量，实际上是提供了两把锁：一把是共享锁，一把是互斥锁；常用于读写锁模式

■ 后续并发程序设计的相关课程会介绍，大家感兴趣可以自行查阅

**async、future、promise
与异步**

案例三

■ 我们将案例一的需求改为

- 不断输入整数 n
- 判断 n 是否为素数

■ 使用thread可以提高素数判断的速度，但第 $n+1$ 次输入依然需要等待第 n 次判断结束方可执行

■ 能否让输入不受判断方法的阻塞？

“同步”与“异步”

■ 同步 (Synchronous)

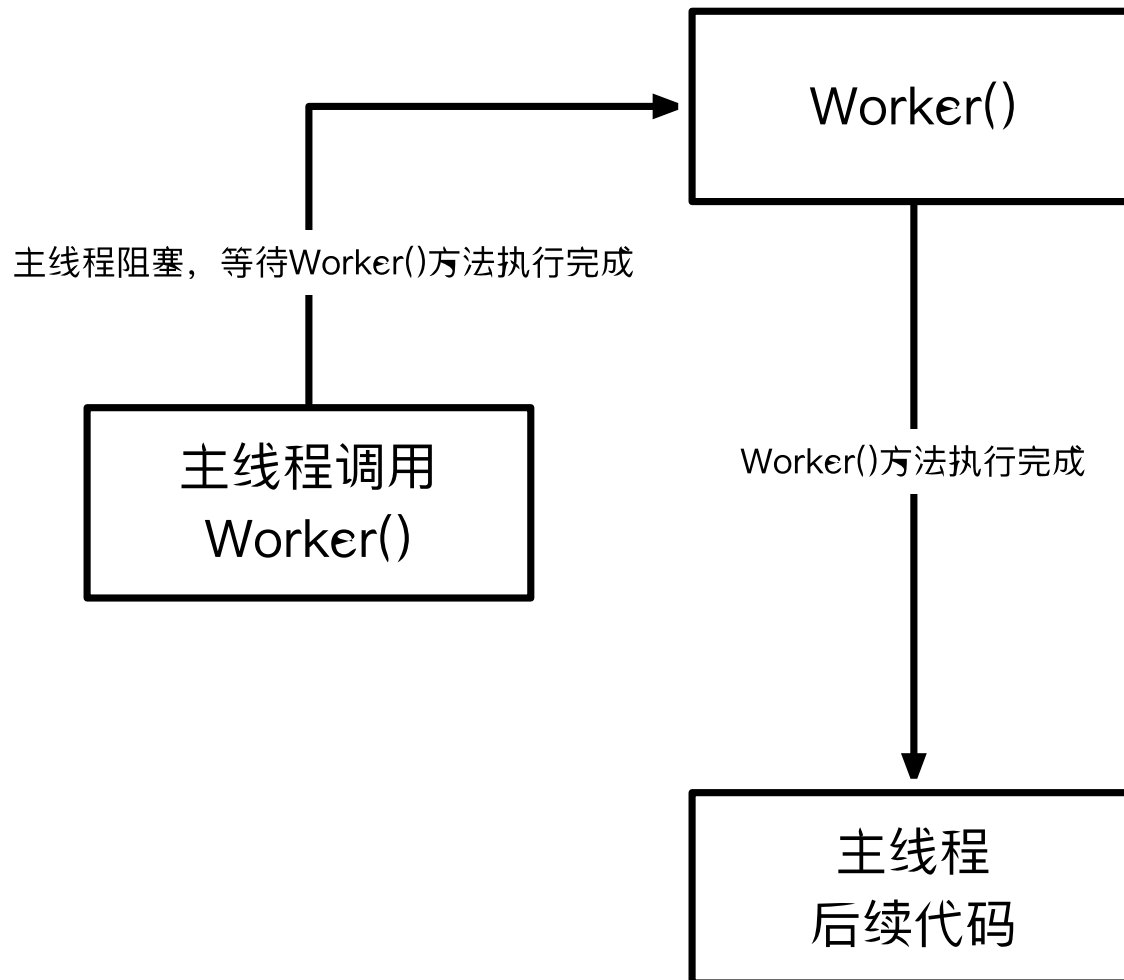
- 同步调用一旦开始，调用者必须等到调用返回结果后才能继续后续的行为

■ 异步 (Asynchronous)

- 异步调用一旦开始，被调用方法就会立即返回，调用者可以无阻塞继续后续的操作
- 被调用方法通常会在另外一个线程中默默运行，整个过程，不会阻碍调用者的工作
- 被调用方法完成后可以通过特殊机制传递信息给调用者

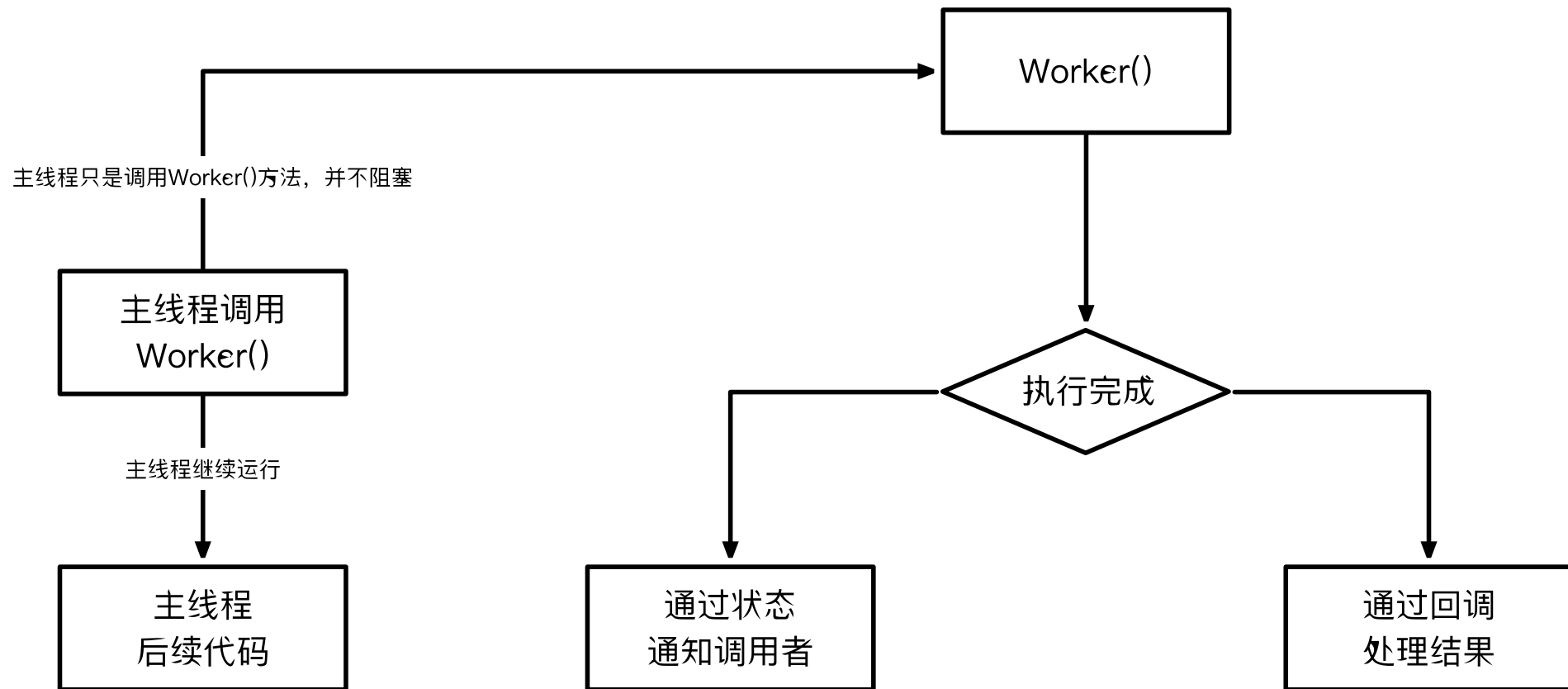
“同步”与“异步”

■ 同步 (Synchronous)



“同步” 与 “异步”

■ 异步 (Asynchronous)



async

- 很多编程语言（如JavaScript）都提供了异步的机制，使耗时的操作不影响当前线程的整体执行，C++11中**async**便有这样的功能

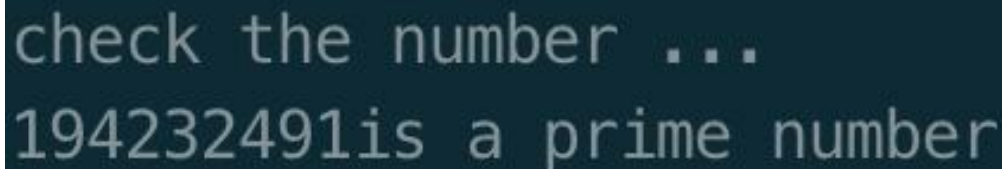
- 定义：

- `future async (Fn&& fn, Args&&... args);`
- `future async (launch policy, Fn&& fn, Args&&... args);`
- 和`thread`类似，入口函数 `fn` 函数的参数由 `args` 给出
- `async`会返回一个`future`对象，用来存储异步任务的执行状态和结果
- `policy`有三种选择
 - `launch::async` 保证异步行为，执行后，系统创建一个线程执行对应的函数
 - `launch::deferred` 表示延迟调用，在调用`future`中的`wait()`或者`get()`函数时，才执行入口函数
 - `launch::async || launch::deferred` 默认策略，由系统自己决定怎么调用

async

```
#include <iostream>
#include <future>
#include <cmath>
using namespace std;
bool check_num(int num) {
    bool flag = true;
    if (num == 1)
        flag = false;
    else
        for (int i = sqrt(num); i > 1; i--)
            if (num % i == 0) {
                flag = false;
                break;
            }
    if (flag)
        cout << num << "is a prime number" << endl;
    else
        cout << num << "is not a prime number" << endl;
    return flag;
}
```

```
int main () {
    std::future<bool> fut =
    async(check_num, 194232491);
    // 也可以写成 auto fut =
    async(check_num, 194232491);
    cout << "check the number ..." << endl;
    return 0;
}
```



```
check the number ...
194232491is a prime number
```

先输出“check the number”说明了
async启动check_num()
后立刻执行了main函数中的cout，验证了async异步调用不会阻塞主线程

async

与可调用对象 (Callable Objects)
一起使用 如 **Lambda表达式**

```
...
int main () {
    std::future<bool> fut = async([](int num) {
        bool flag = true;
        if (num == 1)
            flag = false;
        else
            for (int i = sqrt(num); i > 1; i--)
                if (num % i == 0) {
                    flag = false;
                    break;
                }
        if (flag)
            cout << num << "is a prime number" << endl;
        else
            cout << num << "is not a prime number" << endl;
        return flag;
    }, 194232491);
    cout << "check the number ..." << endl;
    return 0;
}
```


future

■ future 类提供访问异步操作结果的接口

- `wait()` 接口，阻塞当前线程，等待异步线程结束
- `get()` 接口，获取异步线程执行结果，
 - 需要注意的是，**future 对象只能被一个线程获取值**并且在调用 `get()` 之后，就没有可以获取的值了
 - 如果多个线程调用同一个 `future` 实例的 `get()` 会出现数据竞争，其结果是未定义的
 - 调用 `get()` 如果异步线程没有结束会一直等待，类似 `wait()`
- `wait_for(timeout_duration)` 如果在指定超时间隔后仍然没有结束异步线程，则返回目标线程当前状态，并**取消 `wait_for` 的阻塞**
 - `future_status::deferred` 仍未启动
 - `future_status::ready` 结果就绪
 - `future_status::timeout` 已超过时限，异步线程仍在执行

案例三

```
#include <future>
#include <chrono>
#include <vector>
#include <thread>
#include <random>
#include <iostream>
```

```
int total = 0;
//枚举num是否为素数
bool check_num(int num) {}
//设置一个随机来代替输入
default_random_engine e;
int input() {
    return e();
}
//存放异步调用的future和输入数值
vector<future<bool>> future_lists;
vector<int> num_lists;
```

■使用async和future完成案例三

■构建运算不阻塞输入的程序

- 使用异步线程进行运算
- 使用主线程进行状态管理
- 主线程不断检查输入状态和异步线程的执行状态：有输入立刻创建新的异步线程进行处理；某一个异步线程完成后，立刻输出处理结果

案例三

■ 这种不断消耗极短时间进行检测的方式就是**轮询**

```
int main() {
    while (true) {
        int num = input(); //输入数num
        //创建异步线程来检测num是否为素数
        future_lists.push_back(async(check_num, num));
        num_lists.push_back(num);
        //通过future检测每一个异步线程是否完成
        for (int i = future_lists.size() - 1; i >= 0; i--) {
            //每个future等待0.1秒来检测状态
            future_status status = future_lists[i].wait_for(
                chrono::milliseconds(100));
            if (status == future_status::ready) {
                if (future_lists[i].get())
                    cout << num_lists[i] << " is a prime number" << endl;
                else
                    cout << num_lists[i] << " is not a prime number" << endl;
            }
            //删除已经完成任务的future
            future_lists.erase(future_lists.begin() + i);
            num_lists.erase(num_lists.begin() + i);
        }
    }
    return 0; }
```

轮询

■ 轮询是服务器和客户端开发的常用范式

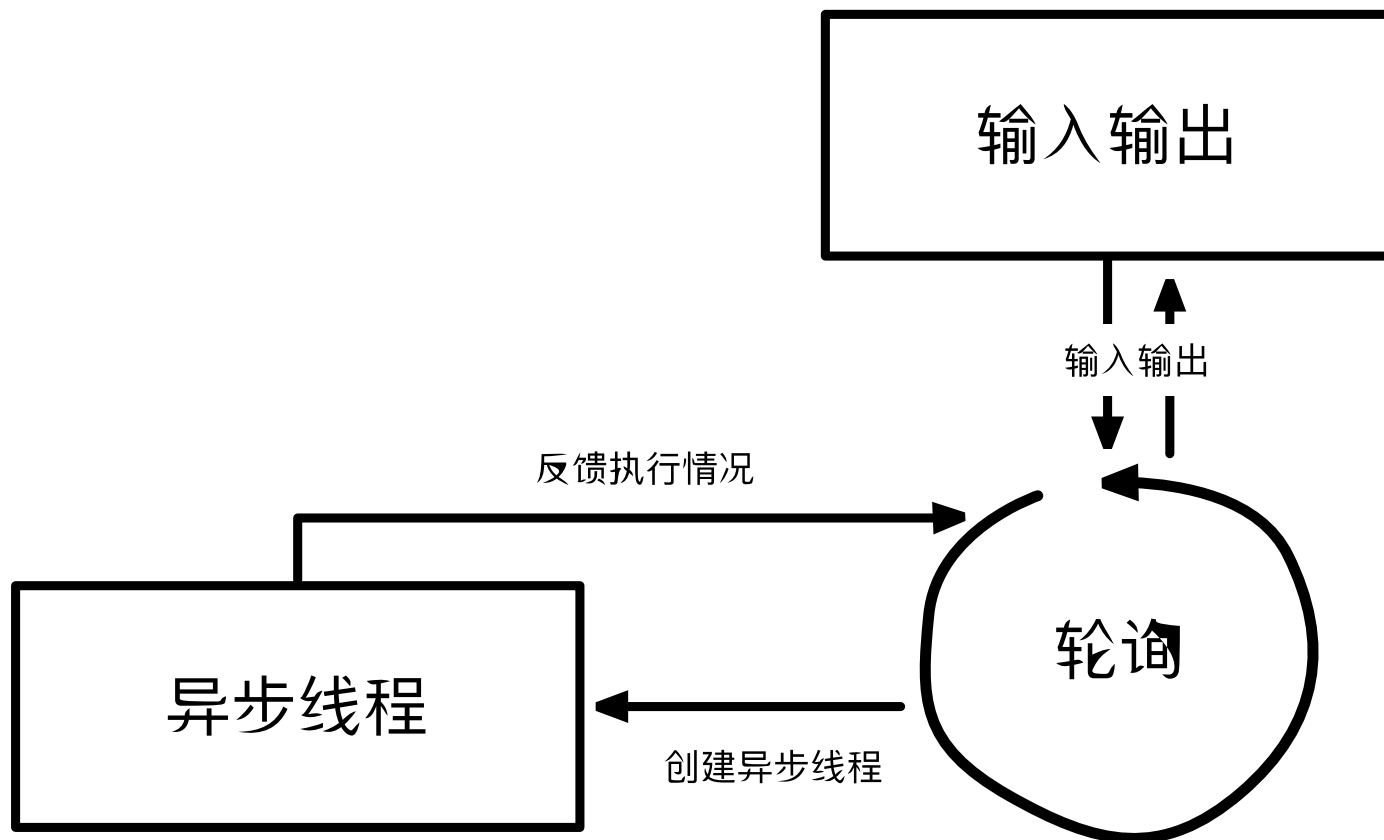
- 客户端

- 如客户端向服务器上传，客户端会每隔一段时间检测是否上传完成
- 在没有完成上传时，上传操作并不阻塞客户端其他操作，只有检测的微小瞬间产生阻塞

- 服务器端

- 服务器为客户端处理耗时较大的任务时，也会开启一个异步线程在后台处理任务
 - 间隔一段时间，服务器就会确认下任务是否完成
 - 整个过程也不会影响服务器的其他操作，如接收客户端的新请求
- 事实上，案例三的实现还可以开启单独的线程来进行轮询，这个留给大家课后操作

轮询



promise

■为了更好的满足跨线程取值的需求，C++还提供了promise类来配合future

- future对象只能被一个线程获取值并且在调用get()之后，就没有可以获取的值了，这带来了风险
- 之前future对象需在异步线程完成后返回值，这也很不方便

■一般流程

- 在当前线程中创建promise对象，并从该promise对象中获得对应的future对象
- 将promise对象传递给目标线程，目标线程通过promise的接口设置特定值，**然后可以继续执行目标线程自身的工作**
- 在特定时间，当前线程按需求通过promise对应的future取值

promise

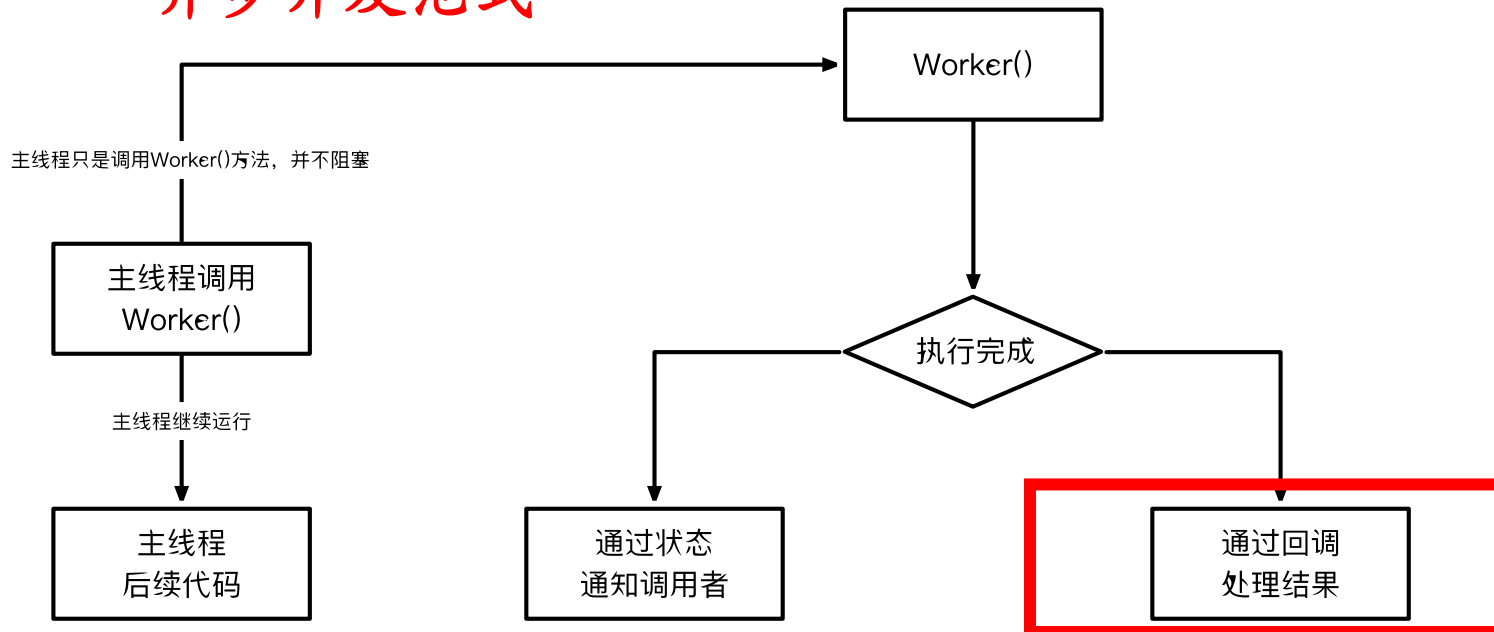
```
//为目标线程传入promise的指针来设置返回结果
void check_num(int num, promise<bool>* res_promise) {
    if (num == 1) {
        res_promise->set_value(false);
        return; }
    for (int i = sqrt(num); i > 1; i--)
        if (num % i == 0) {
            res_promise->set_value(false);
            return; }
    res_promise->set_value(true);
}

int main() {
    int num = 194232491;
    //设置promise, 通过promise的get_future()接口获取配对的future
    //bool是目标线程返回内容的类型
    promise<bool> res_promise;
    future<bool> res_future = res_promise.get_future();
    //向目标线程中传入promise的指针, promise和future保证了线程间结果传递,
    //因此可以detach目标线程, 让目标线程完全独立于当前线程
    thread worker(check_num, num, &res_promise);
    worker.detach();
    //通过future的get()获取结果, 之前已经介绍过了
    if (res_future.get())
        cout << num << " is a prime number" << endl;
    else
        cout << num << " is not a prime number" << endl;
    return 0;
}
```

promise

■ promise

- promise的功能和其自身的翻译“承诺”很类似
- 工作线程将结果存入promise
- 只有promise应允的future才能获取到值
- 到此我们介绍了通过状态通知调用者的异步方法
- 回调方法我们留给大家课后了解，这也是十分重要的异步开发范式



总结

■ C++的并发编程技术

- thread
- Mutex
- async、future、promise

■ 并发设计模式

- 主从模式
- 互斥锁模式
- 异步
- 这些都是并发的经典编程范式
- 在后面的几节课里，我们会结合更多案例讲解更多的设计模式

补充阅读

并发与并行

■ 参考书

- 《深入理解并行编程》
 - 有汉化的较好的在线版
- 《C++ Concurrency in Action》
 - 汉化版本非常差，有兴趣可以翻翻原版

设计模式

- 在日常的开发任务中，采用精心设计的程序架构可以极大方便日常的变动与修改，从而降低维护的代价
- 设计模式（Design Pattern）则是在长时间的实践之中，开发人员总结出的优秀架构与解决方案。经典的设计模式，都是经过相当长的一段时间的试验和错误总结而成的
- 学习设计模式将有助于经验不足的开发人员在实际开发中，灵活地运用面向对象特性，并能够快速构建不同场景下的程序框架，写出优质代码

设计模式

■ 在线资源

- <https://www.runoob.com/design-pattern/design-pattern-intro.html>

■ 参考书

- GoF 的《设计模式》，1994年英文版出版
- 《大话设计模式》
- 《设计模式之禅》
- 《研磨设计模式》

结束