并发编程

jask

09/10/2024

C++ 并发编程笔记 1

你好 C++ 并发世界

并发的定义

硬件并发

- 对于单核或者多核操作系统中，并发都是可以实现的，但是多核的系统更加适合并发

!(C++ 并发编程.assets/cpp1.jpeg)

- 并发具有不确定性，如果多个任务进行并发，其各个任务的进度会互不相同

!(C++ 并发编程.assets/cpp2.jpeg)

- 对于应用软件，如何充分的利用并发性能，是一个重要的点

并发的方式

多进程并发

方式：将一个应用软件拆分为多个独立的进程来运行，这些进程可以通过管道，共享内存，消息队列，信号等进行通信

优点：

- 操作系统会给并发提供高级的通信机制和额外保护，比起线程，多进程写代码不容易出错
- 通过网络连接，独立的进程能够在不同的计算机上运行，虽然增加了通信开销，但是可以有效的增强并发力度，改进性能

多线程并发

方式：每个线程都独立运行，很像轻量级进程，他们公用相同的地址空间，却可以访问大部分数据。

注意：

- 全局变量，全局可见。指向对象或数据的指针和引用能在线程间传递

缺点：

- 容易错

C++ 之所以选择多线程而不是多进程，是因为 C++ 本身尚不直接支持进程间通信

并发与并行

区别

- 并行主要考虑：它能否合理分配多个任务，让 cpu 处理的更快
- 并发主要考虑：重在解耦（分离关注点）和并发能力

什么时候使用并发

为分离关注点而并发

- 这种并发，一般来说，可以对单一的功能进行封装和测试，测试过程中不需要考虑无关代码

为性能并发

- 达成任务并行：将单一任务分解为多个部分，让其各自运作，节约时间
- 达成数据并行：将数据分为不同的部分进行操作，完成数据并行

- 利用并行资源解决规模更大的问题

注：除非潜在的性能提升或者解耦值得，否则不使用

什么时候避免并发

底层原因：收益不及代价

表层原因

- 难维护
- 复杂性增加

其他原因

- 一次性运行太多的线程，会消耗大量的系统资源
  - 例如，webserver 中不能使用一个进程连接一堆的线程，如果这么做会 j
  - 必须谨慎的使用线程池
- 线程运行的过多会让操作系统做上下文切换的时间越多，消耗更多的时间

线程的基本函数

std::thread my_thread(x) 创建线程对象，并创建对象

x.join()：发起线程，并且等到他运行结束，他的意思就是加入，汇合，说白了就是阻塞主线程，让主线程等待子线程执行完毕，然后子线程与主线程汇合

x.decatch()：发起线程，让他自己运行，然后立刻执行接下来的代码任务，他的意思就是分离，传统多线程程序需要等待子线程执行完毕，然后再退出，detach 能够实现主线程运行主线程，子线程运行子线程，互不干扰，主线程也不必等子线程运行结束。

线程管控

基本管控

- 每个 C++ 程序都至少有一个线程，就是运行 main 的线程，它由 C++ 运行时系统启动
- 在这之后程序就可以发起多个线程，他们以别的函数作为入口，这些新线程连同起始线程可以并发运行

发起线程

线程通过 std::thread 对象启动。该对象指明线程要运行的任务

- std::thread 本身能够接受各种对象
- 简单任务就是运行一个普通函数，返回空，不接受参数
- 复杂任务是一个函数对象，通过 operator () 接受参数，只有当收到某种指示信号的时候，线程才会终止

```
#include <iostream>
#include <thread>
class first
{
private:
    int i;
    int j;
public:
    first(int i,int j):i(i),j(j){};
    void operator()(){
        std::cout<<"a"<<"b";
        for(int ii=0;ii<10;i++){
            std::cout<<i<<j;
        }
    }
};
void second(){
    for(int i=0;i<10;i++){
        std::cout<<"c";
    }
}
```

```
int main(){
    first x(1,2);
    std::thread first_thread{x};//以函数对象的形式，创建线程对象
    std::thread second_thread(second);//以函数的形式，创建线程对象
    std::thread third_thread([](){//以 lambda 的形式，创建线程对象
        std::cout<<"d"<<1;
    });
    first_thread.join();
    second_thread.join();
    third_thread.join();
}
```

发起线程的时候，注意：如果在函数创建线程的时候包含指针或者引用，那么除非保证线程肯定会在该函数前结束，否则不能这么做。有两种方法可以避免

- 将数据完整的复制到线程内部，而不是共享数据
- 汇合新线程，确保主线程函数退出前，新线程执行完毕

简单的异常处理

对于已经调用 `join()` 的线程，在线程内部可能会发生异常，一旦发送异常，就会出现 `main()` 里面当前线程代码的后续代码都不会被调用

为了保证其运行，我们需要使用一个标准的 `RAII` 手法，在析构函数中接着继续完成

```
class threadx{
    std::thread &t;
public:
    explicit threadx(std::thread &t):t(t){}
    threadx(threadx&)=delete;
    threadx& operator=(threadx&)=delete;
    ~threadx{
        if(t.joinable()){
            t.join();
        }
    }
};
```

- 首先，对于线程，为了防止发生两次析构。要求 `oop=` 和拷贝构造都要为 `false`

在后台运行线程

使用成员函数 `detach()` 就能完成

`detach` 使用后会和 `main()` 分离，他的归属权和管理权都交给 `C++` 运行时库（`runtime library`），一旦线程退出，与之关联的资源都会被正确回收

调用 `detach` 的函数就会变得不可汇合

向线程传递参数

方法：直接向 `std::thread` 参数继续加入即可

```
void x(int a);
std::thread t(x,a);
```

注意

- 线程具有内部储存空间，参数会按照默认方式（比如 `std::string` 在复制的时候会被转化为 `char const*` 复制之后再转化为 `std::string`）复制到该处，然后新的线程才能访问他们
- 这些复制的参数会被当作临时变量，然后以**右值**的形式传入新线程上的函数/可调用对象，即使是引用也是如此
- 如果我们想要传递一个引用，需要在传递引用的参数前面加上 `std::ref();`

转移对象所有权

对于 `std::uniqueptr`，我们需要转交对象所有权，这个时候就需要 `std::move` 上场。

```
void x(std::unique_ptr<big_object>);
std::unique_ptr<big_object> p(new big_object);
```

```
p->xx(32);
std::thread t(x,std::move(p));
```

- 这里最后一步 `std::move` 的意思是：先进入新创建线程内部的储存空间，再转移给 `x` 函数

对于任意执行线程，线程的归属权可以在实列之间转移，准许程序员在其对象之间转移线程归属权

```
std::thread t1(x);
std::thread t2=std::move(t1);
std::thread t3;
t3=std::move(t2);//正确，可以正确的转移function
t2=std::thread(other);//转换t3所管理的线程
t2=std::move(t3);//错误，一旦原先的线程没有被detach，那么就会终止整个进程
```

`std::thread` 支持右值，就可以让他方便的通过函数进行传入和传出

识别线程

线程所属类别是 `std::thread::id`

获取 `id` 的方法

- 对于 `std::thread` 对象：调用成员函数 `get_id();`
- 对于自己当前的所在线程：调用成员函数 `std::this_thread::get_id();`

对于线程的比较，标准库提供了全套的比较函数，可以进行全套的比较，没有任何限制

在线程间共享数据

条件竞争

定义：在并发编程中，操作由两个或者多个线程负责，他们争先让线程执行各自的操作，结果取决于他们的相对次序

恶性条件竞争的条件：完成一个操作，动用了大于 `1` 份的数据。

防止恶性条件竞争

宗旨：采取保护措施去包装数据结构，确保变量不会破坏，中间的改动过程只是对线程可见

方法：

- 事务
  - 把需要执行的数据读写的操作写成一个完整的序列
  - 先用事务储存日志记录
  - 再将序列当作单一步骤提交执行
  - 如果别的线程改动了和数据，那么事务重新开始
- 互斥
  - 互斥是保护共享数据最基本的方式

在 *C++* 中使用互斥

在 *C++* 中我们可以使用 `std::mutex` 来构建互斥，然后使用 `lock()` 加锁/`unlock()` 解锁

C++ 中有一个函数 `std::lock_guard<>` 可以保证互斥能够被正确的解锁

```
#include <list>
#include <mutex>
#include <algorithm>
std::list<int> some_list;  ---
std::mutex some_mutex;  ---
void add_to_list(int new_value)
{
std::lock_guard<std::mutex> guard(some_mutex);  ---
some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{
std::lock_guard<std::mutex> guard(some_mutex);  ---
```

```
return std::find(some_list.begin(),some_list.end(),value_to_find)
!= some_list.end();
}
```

lock_guard<> 函数是 RAII 手法：构造的时候加锁，析构的时候解锁，因此他只能保护其作用域内的函数不被修改

一般我们加锁并不会像上面那 1，2 那样使用，而是把加锁的对象和锁放在同一个 class 里面，辅以函数封装加强保护

若利用互斥保护共享数据，就需要谨慎的设计函数接口，确保互斥的现行锁定，再对受保护的对象进行访问，保证不留后门

发现接口固有的条件竞争

```
//下面代码在单一线程中可以安全，但是在多线程中就不会了
stack<int> s;
if(!s.empty()){
    const int value=s.top();//在这一部之前，可能会有其他人调用pop（）导致原先的s.empty()判定就失效了
    s.pop();即使上一个没有发生，在这一部可能也会有人调用pop(),导致之前的s.top()失效了
    doxxxx(value);
}
```

想解决的方法主要是要解决接口竞争，使用粒度更小的锁，方法有以下几个

- 传入引用
  - 好处：可以将上面的两步合二为一，直接将目标送到对应的函数内
  - 坏处：其构造函数不一定带有参数，而且栈容器储存的类别还需要是可赋值的
- 提供不抛出异常的拷贝构造函数/移动构造函数
  - 好处：利用这一点，可以满足让特别的元素让其返回
  - 坏处：部分容器可能不支持
- 返回指针，指向弹出的元素
  - 好处：使用 share——ptr 可以让他自由的做复制，且不会抛出异常，内存分配策略由标准库全权管控，调用者没有这个麻烦
  - 坏处：返回指针所指向的东西千差万别，需要合理的方式管理内存

死锁：问题和解决办法

死锁的定义：有两个线程，需要同时锁住两个互斥，才能进行某项操作，但是他们都分别锁住了一个互斥，都等着给另外一个互斥加锁，且都在等待对方解锁

防范死锁：始终按照相同的顺序对两个互斥加锁

C++ 拥有 std::lock() 函数，可以同时锁死多个锁，保证没有发生死锁的任何风险

- 1.std::lock 传入多个锁，可以同时锁住
- 2.3. 利用 std::adopt_lock 告知 lock_guard 对象，将锁的权利转交给 lock_a，lock_b 对象，由他利用 RAII 进行加锁和解锁

```
class some_big_object;
void swap(some_big_object& lhs, some_big_object& rhs);
class X
{
private:
some_big_object some_detail;
std::mutex m;
public:
X(some_big_object const& sd):some_detail(sd){}
friend void swap(X& lhs, X& rhs)
{
if(&lhs==&rhs)
return;
std::lock(lhs.m,rhs.m); ---
std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock); ---
std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock); ---
swap(lhs.some_detail,rhs.some_detail);
}
};
```

C++17 还有一个特殊函数，std::scope_lock(); 他接受一个可变参数模板，能将 123 合为一步，完美上锁

```
void swap(X& lhs, X& rhs)
{
if(&lhs==&rhs)
return;
std::scoped_lock guard(lhs.m,rhs.m);  ---
swap(lhs.some_detail,rhs.some_detail);
}
```

防范死锁的几个准则

避免嵌套锁：如果已经持有锁本身，就最好不要试图再获取第二个锁。这能从根本上避免死锁

一旦持锁，就避免调用由用户提供的程序接口：主要是避免用户提供的接口会再一次上锁，造成两次上锁，出现错误

依从固定顺序获取锁：在无法使用 `std::lock()` 一次性锁住全部的情况下，那么使用 `lock` 按照次序依次加锁和解锁

按照层级加锁：我们把应用程序分层，并且明确每个互斥位于那个层级，若某线程已经对于低层级加锁，那么就不允许他再对高层级加锁

运用 `std::unique lock<>` 加锁

`std::unique lock<>` 相比于 `std::lock_guard<>` 实现多了一条，他可以做无锁或者有锁构造，`lock_guard` 只能做有锁构造

他有两种状态 `defer_lock` 和 `adopt_lock`

- `defer_lock` 构造的时候不上锁
- `adopt_lock` 构造的时候上锁

```
class some_big_object;
void swap(some_big_object& lhs, some_big_object& rhs);
class X
{
private:
some_big_object some_detail;
std::mutex m;
public:
X(some_big_object const& sd):some_detail(sd){}
friend void swap(X& lhs, X& rhs)
{
if(&lhs==&rhs)
return;
std::unique_lock<std::mutex> lock_a(lhs.m, std::defer_lock);  ---
std::unique_lock<std::mutex> lock_b(rhs.m, std::defer_lock);  ---
std::lock(lhs.m,rhs.m);  ---
swap(lhs.some_detail,rhs.some_detail);
}
};
```

2,3：构造都是构造时候不上锁，最后在 1 的时候进行上锁了

使用场景

他的使用场景一般情况下会在

- 互斥加锁的时机取决于当前程序状态
- 某函数负责执行加锁操作并返回 `std::unique_lock` 对象（也就是转移）

转移锁的好处

- `std::unique_lock` 可以做转移操作，将当前作用域的锁转移到另外一端，这样就可以更精细的管理锁，在某个锁没有必要持有的情况下，可以主动销毁锁

按照合适的粒度加锁

- 单独一个互斥全程加锁，不但很可能加剧锁的争夺，还将难以缩短锁的持续时间，这是双重损失
- 尽可能使用颗粒度细的锁
- 一般的若要执行某项操作，那么就需要在最短的时间内持锁即可

在初始化的过程中保护共享数据

有的时候我们会进行延迟初始化，利用指针，在需要的时候才进行创建。为了能正确且安全的创建唯一一份对象，保证其他线程不会在没有分配资源的情况下就去调用函数，我们使用 `std::once_flag,std::call_once()` 函数

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;  ---
void init_resource()
{
resource_ptr.reset(new some_resource);
}
void foo()
{
std::call_once(resource_flag,init_resource);  ---  初始化函数准确地
被唯一一次调用
resource_ptr->do_something();
}
```

保护甚少更新的数据结构

对于 `std::mutex`，这种锁一旦上锁，就不会允许任何的其他操作进行读和写

为了解决这种情况，我们引进了 `std::stare_mutex`（对应 `std::share_lock)`，这种锁就是共享锁，他允许多个线程并发的读取。类似于数据库

- 一般情况下，`std::mutex` 充当排他锁，`std::share_mutex` 充当共享锁
- 和数据库的一样，共享锁可以无限制的获取，但是如果有人试图获取排他锁，那么就会发送堵塞，直到所有的共享锁全部释放为之，反之如果任意线程持有排他锁，就会发生堵塞无法获取任何共享/排他锁，直到线程释放排他锁

还有一种递归锁 `std::recursive_mutex`，但是使用递归锁的情况下一般都是设计发生失误，不建议使用

等待事件/等待其他事件

假设现在有两个线程，分别为甲，乙，甲要等待乙完成之后再继续，这种情况我们可以利用条件变量等待条件变量成立。

这种情况的话一般可以用中的 `std::condition_variable` 和 `std::condition_variable_any` 做决定。其中前者只能和 `std::mutex` 一起使用，后者条件更加广阔一点，但是性能，灵活性差一些

重要函数

- `std::condition_variable` ；变量等待条件，他有两个重要的成员函数
  - `wait`：接受一个锁，和一个函数，这个函数是判断是否进行读取的关键，`wait` 先获取锁，函数返回 `true` 那么就后继续执行，函数返回 `false` 那么就解锁并挂起等待，一般情况下这个锁使用 `std::unique_lock`，而不是 `lock_guard`。因为 `lock_guard` 没有前者灵活
  - `notify_one`：一次性通知，并且挂起。告知自己已经做完了

例子

```
std::mutex mut;
std::queue<data_chunk> data_queue;  ---
std::condition_variable data_cond;
void data_preparation_thread() // 由线程乙运行
{
while(more_data_to_prepare())
{
data_chunk const data=prepare_data();
{
std::lock_guard<std::mutex> lk(mut);
data_queue.push(data);  ---
}
data_cond.notify_one();  ---
}
}
void data_processing_thread() // 由线程甲运行
{
while(true)
```

```
{
std::unique_lock<std::mutex> lk(mut);  ---
data_cond.wait(
lk,[]{return !data_queue.empty();});  ---
data_chunk data=data_queue.front();
data_queue.pop();
lk.unlock();  ---
process(data);
if(is_last_chunk(data))
break;
}
}
```

1. 创建一个线程安全的队列

2.3 完成数据处理之后做一次通知

4.5: 判断是否完成处理条件，拿到处理条件继续开始。

使用 future 等待一次性事件发生

C++ 标准库使用 future 来模拟一次性事件，他以短暂的事件反复检查目标事件是否发生

C++ 标准库有两种 future,分别由两个类模板实现,他们定义于,包含两种:std::future<>(unique future),std::shared_future<>(shared
两种 future 本身并不能提供同步访问，所以是需要上锁保护的

从后台任务返回值

如果我们要是不是急需线程运算所需要的值，我们可以使用 std::async() 按照异步的方式存储对象，他返回一个 std::future 对象，只要调用 get() 就能拿到值

```
#include <future>
#include <iostream>
int find_the_answer_to_ltuae();
void do_other_stuff();
int main()
{
std::future<int> the_answer=std::async(find_the_answer_to_ltuae);
do_other_stuff();
std::cout<<"The answer is "<<the_answer.get()<<std::endl;
}
```

和 std::thread 类似，他的后面也可以传递参数作为参数

```
#include <string>
#include <future>
struct X
{
void foo(int,std::string const&);
std::string bar(std::string const&);
};
X x;
auto f1=std::async(&X::foo,&x,42,"hello");  ---  调用p->foo(42,"hello")，其中p的值
```

除此以外，我们可以选择何时调用，利用 std::launch::deferred 或者 std::launch::async 决定是利用新的线程去调用，还是等待任何函数之后在调用

```
auto f6=std::async(std::launch::asyna,y(),1.2);//交给线程库去调用
auto f7=std::async(std::launch::deferren,baz(),1);//等到wait/get出现再调用
f7.wait()/f7.get();
```

关联 future 实例以及任务

std::packaged_task<> 连接了 future 对象和函数,std::packaged_task<> 对象在执行任务的时候,会调用关联的函数(get_future),把返回值保存在 future 的内部数据，并且令 future 准备就绪。他可作为线程池的构建单元

它属于类模板，其模板参数是函数签名，比如:std::package_task<std::string(std::vector<char>*,int)>

```
std::future<void> post_task_for_gui_thread(Func f)
{
std::packaged_task<void()> task(f);  ---
std::future<void> res=task.get_future();  ---
std::lock_guard<std::mutex> lk(m);
tasks.push_back(std::move(task));  ---
return res;  ---
}
```

7：接受一个函数，作为打包

8. 获取 future 对象，然后准备运行

9,10：上锁之后做运行的操作

创建 std::promise

与 package 不太一样的地方是，promise 是一种异步求值的方法，能延后需要读出的值

promise 可以传入对应的函数对象，一旦传入对象之后不会立刻运行。后面还需要通过 set_value() 函数传入值，一旦传入完成，future 就设置好了，就可以通过 get_future 获取，然后继续操作

```
data_packet data=connection->incoming();
std::promise<payload_type>& p=
connection->get_promise(data.id);  ---
p.set_value(data.payload);
```

将异常保存到 future 中

对于抛出异常，如果 std::async 的函数发出异常，那么他会保存在 std::future 中，并且通过 get 调出

```
std::future<double> f=std::async(square_root,-1);
double y=f.get();
```

对于 std::promise，如果需要传入一个异常，那么就要 set_exception() 而不是 set_value()

```
extern std::promise<double> some_promise;
try
{
    some_promise.set_value(calculate_value());
}
catch(...)
{
    some_promise.set_exception(std::current_exception());
}
```

如果我们能预知异常的类型的话，那么可以直接使用 std::make_exception_ptr() 直接保存新的异常，相比于 try_catch 块，这种方法简化了代码，而且利于编译器优化代码

```
some_promise.set_exception(std::make_exception_ptr(std::logic_error("foo")));
```

如果关联的 future 对应的事件未能准备就绪(抛出异常但是直接析构了)那么 future 函数将会储存 std::future_errc::broke_promise 这个异常。

多个线程一起等待

- 相比于 std::future，std::share_future 实例是能够复制出副本的，他可以持有该类多个对象，全部指向同一异步任务的状态数据
- 就算是使用 std::shared_future 我们也需要在对于访问同一个对象的时候，也必须采用锁去保证避免数据竞争
- std::share_future 的实例是通过 std::future 对象构造获得，所以如果需要转移权利，就需要 std::move

```
//注future和promise都具有成员函数vaild()判别异步状态是否有效

std::promise<int> p1;
std::promise<int> p2;
std::future<int> f(p1.get_future());
std::share_future<int> f2(p2.get_future());//隐式类型转换，将future做了对象转移
assert(f.valid());     ---    future对象f有效
std::shared_future<int> sf(std::move(f));
```

```
assert(!f.valid());     ---   对象f不再有效
assert(sf.valid());     ---   对象sf开始生效
```

使用 future 里面的 share() 函数，可以直接做 share_future

```
std::promise< std::map<SomeIndexType, SomeDataType, SomeComparator,
    SomeAllocator>::iterator> p;
auto sf=p.get_future().share();
```

限时等待

时钟类

在 std::chrono 提供了各种时钟在 std::radio 里面提供了对应的计数

```
//时钟每秒计数25次
std::ratio<1,25>
//每2.5秒计数一次
std::radio<5,2>
```

时长类

std::chrono::duration<> 是一个简单的时间部件使用 shrot 计时,一分钟包含 60s:std::chrono::duration<short,std::radio<60,1

时间点类

我们一般可以使用 std::chrono::time_point 指定一个时间点，或者做加减运算

接受超时时限的函数

经典的有两个函数: std::this_thread::sleep_for() 和 std::this_thread::sleep_until() 第一个是按照指定的时长休眠，第二个是休眠到某个时刻为止

其他经典函数（图片）

利用 future 进行函数式编程

定义函数式编程是一种编程风格，他的调用结果完全取决于参数，不依赖任何外部状态。纯函数产生的作用被完全限制在返回值上，不会改动任何外部状态

future 对象可在线程之间传递，所以一个计算任务可以依赖另外一个任务的结果，却不必显式的访问共享数据

使用消息传递进行同步

定义 *CSP*（通信式串行进程范式），我们可以利用 *C++* 模拟 *CSP*，假设不存在共享数据，线程只是接受消息，那么可以单纯的依据其反应行为，独立的对线程进行完整的逻辑推断 ***C++ 无法实现 CSP 的原因*** *C++* 线程是共享地址空间的

符合并发技术规约的后续风格并发

在 std::experimental 空间内部，给了两个新的 future，这两个新的 future 支持一个新的关键特性：后续。他的功能是**一旦结果数据就绪，就接着进行某项处理**

```
std::experimental::future<int> find_the_answer;
auto fut=find_the_answer();
auto fut2=fut.then(find_the_question);
```

这种 future 支持并发处理，并且能形成一个调用链，在调用链中间弱抛出异常，不会立刻中断整个线程，而是会继续流转到最后一个链条脱离，从最后一个链条脱出，对应的 catch 块能处理全部异常

```
std::experimental::future<void> process_login(
    std::string const& username,std::string const& password)
{
    return backend.async_authenticate_user(username,password).then(
        [](std::experimental::future<user_id> id){
            return backend.async_request_current_info(id.get());
        }).then([](std::experimental::future<user_data> info_to_display){
            try{
                update_display(info_to_display.get());
```

```
            } catch(std::exception& e){
                display_error(e);
            }
        });
}
```

等待多个 future

使用 std::experimental::when_all 可以等待到所有的 future 就绪。他的作用是把所有的小的 future 合并为一个大的总领 future，当传入的 future 全部就绪，那么总领 future 也会就绪，接下来就可以运用总领 future 安排其他的工作

```
std::vector<std::experimental::future<ChunkResult>> results;
    for(auto begin=vec.begin(),end=vec.end();beg!=end;){
        size_t const remaining_size=end-begin;
        size_t const this_chunk_size=std::min(remaining_size,chunk_size);
        results.push_back(
            spawn_async(
            process_chunk,begin,begin+this_chunk_size));
        begin+=this_chunk_size;
    }
    return std::experimental::when_all(
        results.begin(),results.end()).then(     ---
        [](std::future<std::vector<
             std::experimental::future<ChunkResult>>> ready_results)
        {
            std::vector<std::experimental::future<ChunkResult>>
                all_results=ready_results .get();
            std::vector<ChunkResult> v;
            v.reserve(all_results.size());
            for(auto& f: all_results)
            {
                v.push_back(f.get());     ---
            }
            return gather_results(v);
        });
}
```

运用 std::experimental::when_any() 函数等待多个 future，直到其中之一准备就绪

std::experimental::when_any() 与 when_all() 类似，当所有的 future 其中之一就绪，那么 when_any () 就会运行，返回一个 future。

线程门和线程卡

线程门

- 线程门是一个特定对象，内涵一个计数器，一旦减到 0，就会进入就绪状态。
- 线程门是一个轻量级工具，用于等待一系列目标事件的发生

定义：头文件，接受唯一一个参数。每当等待的目标事件发生时候，我们都在线程门上调用 count_down()，一旦到 0 就进入就绪状态。就可以调用 wait()，如果检查它是否已经就绪，那么就调用 count_down_and_wait()

```
void foo(){
    unsigned const thread_count=...;
    latch done(thread_count);     ---
    my_data data[thread_count];
    std::vector<std::future<void>> threads;
    for(unsigned i=0;i<thread_count;++i)
        threads.push_back(std::async(std::launch::async,[&,i]{     ---
            data[i]=make_data(i);
            done.count_down();     ---
            do_more_stuff();     ---
        }));
    done.wait();     ---
```

```
process_data(data,thread_count);     ---
```

线程卡

- 线程卡是可以重复使用的部件
- 线程卡的每个同步周期内，只准许每个线程唯一一次运行到其所在之处，线程卡处就会被阻塞，一直等到同组的线程全部抵达，在那一个瞬间他们会全部释放

C++ 内存模型和原子操作

原子操作概论和改动序列

原子操作内存模型精确定义了基础构建单元应当如何运转。内存模型牵涉两个方面：基础结构和并发。所有与多线程相关事项都会牵涉内存区域。

假设两个线程访问同一内存区域，却没有强制它们服从一定的访问次序，如果其中至少有一个是非原子化访问，并且至少有一个是写操作，就会出现数据竞争，导致未定义定位。要么就是在目标区域采用院子操作，强制两个线程遵从一定的访问次序

改动序列在 C++ 程序中，每个对象都具有一个改动序列，它由对象上全部写操作完成。在不同的线程上观察属于同一个变量的序列，如果所见各异，就说明出现了数据竞争和未定义行为

标准原子类型

- 标准原子类型定义于 atomic 内，这些类型的操作都是原子化的。
- 原子操作的关键用途是取代需要互斥的同步方式，比如无锁数据结构。
- 所有的原子类都含有一个静态常量表达式：is_always_lock_free。
- 只有一个原子类型不提供 is_lock_free 成员函数：std:: aromic_flag
- 由于不具备拷贝构造函数、复制操作符，所以原子类型对象无法复制，也无法赋值，但是他们可以转化为内建类型赋值，也支持成员函数处理

操作 std::atomic_flag

- 这个操作符并不常用，准确来说不会直接用到
- std::atomic_flag 类型对象必须由宏 ATOMIC——FLAG——INIT 把标志初始化为置零状态
- 构造完成后，我们执行三种操作：销毁，置零，读取原有值并设置标志成立。分别对应析构函数，成员函数 clear，成员函数 test and set
- 涉及两个对象的操作是无法原子化的

操作 std::atomic

std::atomic，是一个功能更加齐全的布尔标志，我们能根据原子布尔量创建其对象。该类型的实例还能接受非原子布尔量的赋值原子操作所支持的赋值操作符不返回引用，而是按照值返回 atomic 也可以提供更加通用的函数 exchange 来代替 test_and_set，他获取原有的值，然后还让我们自行选定新值进行替换

对于 store：他是储存操作，load 是载入操作，exchange 是读改写的操作

```
std::atomic<bool> b;
bool x=b.load(std::memory_order_acquire);
b.store(true);
x=b.exchange(false,std::memory_order_acq_rel);
```

根据原子对象当前的值决定是否保存新值

这一操作被称作：比较-交换。实现形式是 compare-exchange-weak () 和 compare-exchange-strong ()。它们都返回 bool，如果完成了就是 true，否则是 false

对于 compare-exchange-weak() 因为对于比较/交换操作，如果在 cpu 无法使用一条指令执行的情况下，就会可能导致佯败。这种操作往往是函数的执行时机不对。所以可以配合循环

```
bool expected=false;
extern atomic<bool> b; //由其他源文件的代码设定变量的值
while(!b.compare_exchange_weak(expected,true) && !expected);
```

对于 compare-exchange-strong () 来说，只有当原子变量的值不符合预期的时候，compare-exchange-strong () 才返回 false。

一般来讲，对于 weak，如果存入的值不需要耗时的计算，简单。那么可以使用 weak+ 循环，速度更快。反之使用 strong ()，可以避免重复计算

操作 std::atomic<T*>

他是原子化类型的指针，带有函数

```
is_lock_free()
load()
store()
exchange()
compare_exchange_weak()
compare_exchange_strong()
```

它同时包装了重载运算符 +=，-=，++，--它也提供了 fetch_sub () ,fetch_add () 与上面的并不相通，比如说使用 fetchadd 会更新对象，但是返回的指针指向更新之前的数组

操作标准整数原子类型

对于 std::atomic…他包括常见的整数原子类型和操作

- 既包括常用的原子操作（load()、store()、exchange()、compare_exchange_weak() 和 compare_exchange_strong()），
- 也包括原子运算（fetch_add()、fetch_sub()、fetch_and()、fetch_or()、fetch_xor()），
- 以及这些运算的复合赋值形式（+=、-=、&=、|= 和 ^=），
- 还有前后缀形式的自增和自减（++x、x++、--x 和 x--）。

同步关系与先行关系

同步关系

同步关系只存在于原子类型的操作之间。他的基本思想是：对变量 X 执行原子写操作和原子读操作 R，且两者都有适当的标记，那么

- R 读取了 W 直接存入的值
- W 所属线程随后还执行了另一原子的写操作，R 读取了后面存入的值
- 任意线程执行一连串《读-改-写》的操作，而其中第一个操作读取的值由 W 写出

以上这些只要发生任意一个，那么就存在同步关系

先行关系

先行关系和严格先行关系是在程序中确立操作次序的基本执行要素。

我们知道先行关系和严格先行关系是在程序中确立操作次序的基本要素，他们的用途是界定那些操作能看见其他操作产生的结果，对于单一线程，如果出现多次操作，那么会随机执行，因为 C++ 没有界定执行次序

```
foo(get(),get());//不清楚是第一个get还是第二个get先执行
```

对于线程间执行相对简单

- 甲乙两个操作分别由不同线程执行，且他们同步，那么甲操作就会先与乙操作发生。这是可传递关系
- 线程间的先行关系还可与控制流程的先后关系结合：若甲操作按控制流程在乙操作之前发生，且乙操作跨线程地先于丙，那么甲就先于丙
- 假设我们要在某个线程上改动多个数据，那么要令这些改动为另一线程的后续操作可见，两个线程就需要确立一次同步关系

原子操作的内存次序

原子操作上有六种内存次序，但是也只有三种模式

- 先后一致次序（memory-order-seq-cst）
- 获取释放次序（memory-order-consume，memory-order-acquire，memory-order-release 和 memory-order-acq-rel）
- 宽松次序（memory-order-relaxed）在不同的 cpu 架构上，这几种内存模型会有不同的运行开销。

先后一致性次序

- 先后一致性次序是最直观，最符合直觉的内存次序
- 但是由于它要求在所有线程之间进行全局同步，因此也是代价最高的内存次序

非先后一致次序

- 与先后一致性次序不同，不同线程所看到的同一组操作次序和效果出现差异了
- 最主要的区别是：线程之间不必就事件发生次序达成一致

宽松次序

- 在一个线程内，对相同变量的访问次序不得不重新编排
- 宽松次序不保证其变化可以为对方可见

获取-释放次序

- 获取-释放次序比宽松次序更加严格一点，他会产生一定程度的同步效果，而不会形成服从先后一致次序的全局总操作序列
- 若多个线程从获取-释放次序，则其所见的操作序列可能各异，但是其差异的程度和方式都是受到一定条件的制约

栅栏

栅栏作用是强制增加内存次序，却无需改动任何数据栅栏的整体运作思路是：若储存操作处于释放栅栏后面，则储存结果为获取操作所见，则该释放栅栏与获取操作同步；若载入操作处于获取栅栏前面，而载入操作见到了释放操作的结果，则该获取栅栏与释放操作同步

设计基于锁并发数据结构

并发设计的内涵

- 多线程执行的操作无论异同，每个线程所见的数据结构都是自洽的，数据不会丢失或破坏，所有不变量终将成立，恶性条件竞争也不会出现
- 互斥保护数据结构是明令静止真正的并发访问，叫做事务的串行化。如果要实现更好的并发，那么保护范围越小，需要的串行化操作越少，并发的程度就有可能越高

设计并发结构需要注意的地方

- 第一：在构造函数完成以前和析构函数开始之后，访问不会发生。如果数据结构支持赋值，内部互换，拷贝构造等，那么即使其中绝大部分函数可以通过多线程安全的并发调用，但是也要看这些数据并发调用是否安全，是否需要排他访问
- 第二：如何才能只保留最必要的串行操作，将串行操作减少到最低程度。并且最大限度的实现数据的并发访问。