书籍地址：

<https://www.bookstack.cn/read/CPP-Concurrency-In-Action-2ed-2019/content-about_this_book-chinese.md>

1. 你好，C++的并发世界

\*并发：单个系统里同时执行多个独立地任务，而非顺序的进行一些活动。

\*分离关注点（SOC） Separation of concerns

+ 一是分层，二是面向接口编程

\*native\_handle()：直接操作底层实现

1. 线程管理

\*detach过的线程为 守护线程

\*发后即忘（fire and forget）的任务使用到守护进程方式（detach函数）。

\* void process\_big\_object(std::unique\_ptr<big\_object>);

std::unique\_ptr<big\_object> p(new big\_object);

p->prepare\_data(42);

std::thread t(process\_big\_object, std::move(p));

\*scoped\_thread 用法

1. class scoped\_thread
2. {
3. std::thread t;
4. public:
5. explicit scoped\_thread(std::thread t\_): // 1
6. t(std::move(t\_))
7. {
8. if(!t.joinable()) // 2
9. throw std::logic\_error(“No thread”);
10. }
11. ~scoped\_thread()
12. {
13. t.join(); // 3
14. }
15. scoped\_thread(scoped\_thread const&)=delete;
16. scoped\_thread& operator=(scoped\_thread const&)=delete;
17. };
18. struct func; // 定义在清单2.1中
19. void f()
20. {
21. int some\_local\_state;
22. scoped\_thread t(std::thread(func(some\_local\_state))); // 4
23. do\_something\_in\_current\_thread();
24. } // 5

\*std::thread::hardware\_concurrency() 返回 cpu核心数量

\*std::this\_thread::get\_id() ; t1.get\_id(); 返回线程id

1. 线程间共享数据

\*互斥量：std::mutex lock(), unlock()操作

\*std::lock\_guard<std::mutex> guard(some\_mutex); 来保证正确解锁

+ C++17版本简化为： std::lock\_guard guard(some\_mutex);

+ 加强版数据保护机制：std::scoped\_lock guard(some\_mutex);

\*返回指向弹出值的指针时，使用 std::shared\_ptr

\*std::lock: 一次性锁住多个互斥量，避免死锁

举例：

void swap(X& lhs, X& rhs)

{

if (&lhs == &rhs)

return;

std::lock(lhs.m, rhs.m); // 1

std::lock\_guard<std::mutex> lock\_a(lhs.m, std::adopt\_lock); // 2

std::lock\_guard<std::mutex> lock\_b(rhs.m, std::adopt\_lock); // 3

swap(lhs.some\_detail, rhs.some\_detail);

}

C++17版本 std::scoped\_lock 来替换 std::lock

void swap(X& lhs, X& rhs)

{

if (&lhs == &rhs)

return;

std::scoped\_lock guard(lhs.m, rhs.m); // 1

swap(lhs.some\_detail, rhs.some\_detail);

}

\*避免死锁的进阶指导

1; 避免嵌套锁

2; 避免在持有锁时调用用户提供的代码

3; 使用固定顺序获取锁

4; 使用锁的层次结构

+hierarchical\_mutex high\_level\_mutex(10000);

+hierarchical\_mutex low\_level\_mutex(5000);

std::lock\_guard<hierarchical\_mutex> lk(high\_level\_mutex);

\*std::unique\_lock 要比 std::lock\_guard更灵活， 内部保存mutex状态

举例对比：

void swap(X& lhs, X& rhs)

{

if (&lhs == &rhs)

return;

std::unique\_lock<std::mutex> lock\_a(lhs.m, std::defer\_lock); // 1

std::unique\_lock<std::mutex> lock\_b(rhs.m, std::defer\_lock); // 1 std::defer\_lock 留下未上锁的互斥量

std::lock(lock\_a, lock\_b); // 2 互斥量在这里上锁

swap(lhs.some\_detail, rhs.some\_detail);

}

\*初始化时需要保护共享数据的方法， 要避免《双重检查》， 使用std::call\_once, std::once\_flag来做保护。

例子：

std::shared\_ptr<some\_resource> resource\_ptr;

std::once\_flag resource\_flag; // 1

void init\_resource()

{

resource\_ptr.reset(new some\_resource);

}

void foo()

{

std::call\_once(resource\_flag, init\_resource); // 可以完整的进行一次初始化

resource\_ptr->do\_something();

}

\*读写锁

+ std::lock\_guard<std::shared\_mutex> lk(entry\_mutex); //写锁

+ std::shared\_lock<std::shared\_mutex> lk(entry\_mutex); //读锁

\*嵌套锁

+ std::lock\_guard<std::recursive\_mutex>

1. 同步并发操作

\*std::condition\_variable 的使用，

std::mutex mut;

std::queue<data\_chunk> data\_queue; // 1

std::condition\_variable data\_cond;

void data\_preparation\_thread()

{

while (more\_data\_to\_prepare())

{

data\_chunk const data = prepare\_data();

std::lock\_guard<std::mutex> lk(mut);

data\_queue.push(data); // 2

data\_cond.notify\_one(); // 3

}

}

void data\_processing\_thread()

{

while (true)

{

std::unique\_lock<std::mutex> lk(mut); // 4

data\_cond.wait(

lk, [] {return !data\_queue.empty(); }); // 5

data\_chunk data = data\_queue.front();

data\_queue.pop();

lk.unlock(); // 6

process(data);

if (is\_last\_chunk(data))

break;

}

}

\*std::future<>:

+ std::async 启动异步任务。

+get的时候阻塞等待。

例子：

#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(); //这里不阻塞

int result = the\_answer.get() ; //这里会被阻塞

std::cout << "The answer is " <<result << std::endl;

}

\*std::launch::async:

+std::launch::async： 立即执行，并且新创建线程执行， 是系统默认值

+std::launch::deferred：延迟调用，直到遇到get或者wait函数时才调用函数，并且在原来的线程上执行。

例子：

auto f6 = std::async(std::launch::async, Y(), 1.2); // 在新线程上执行

auto f7 = std::async(std::launch::deferred, baz, std::ref(x)); // 在wait()或get()调用时执行

auto f8 = std::async(

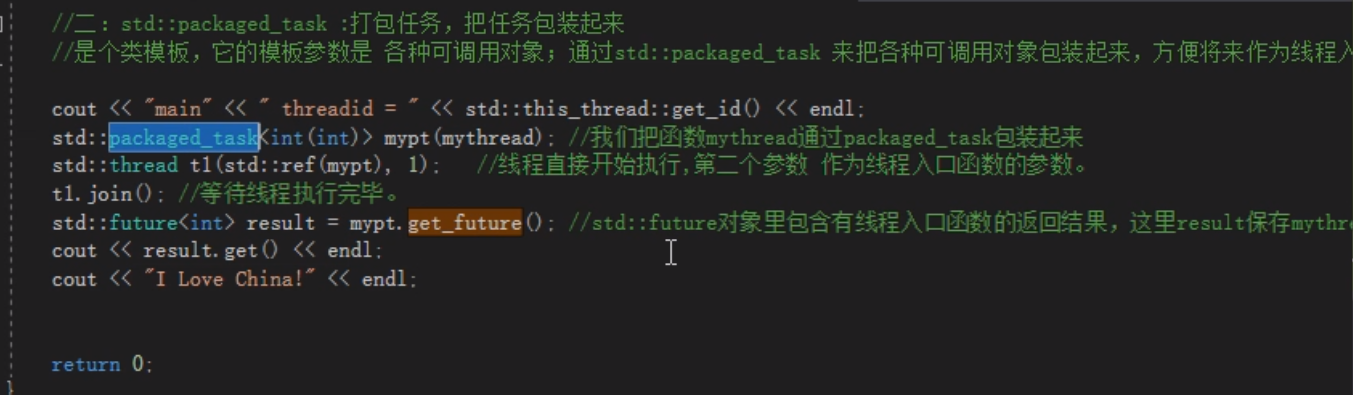
std::launch::deferred | std::launch::async,

baz, std::ref(x)); // 实现选择执行方式

auto f9 = std::async(baz, std::ref(x));

f7.wait(); // 调用延迟函数

\*std::packaged\_task: 函数包装

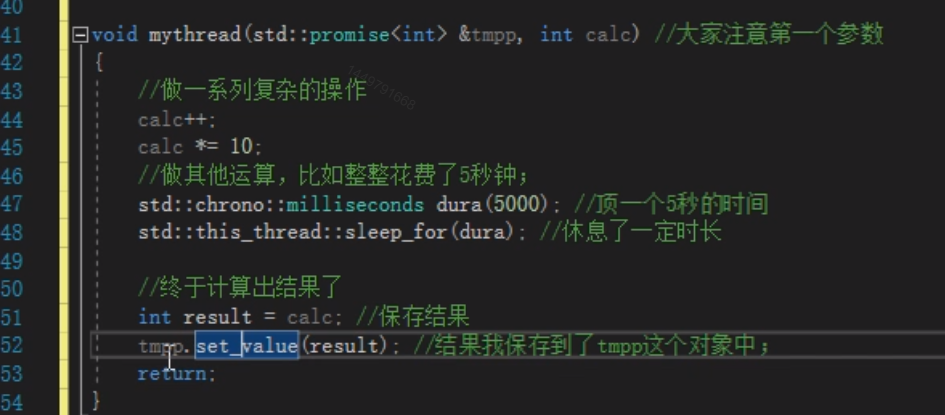


1. #include <deque>
2. #include <mutex>
3. #include <future>
4. #include <thread>
5. #include <utility>
6. std::mutex m;
7. std::deque<std::packaged\_task<void()> > tasks;
8. bool gui\_shutdown\_message\_received();
9. void get\_and\_process\_gui\_message();
10. void gui\_thread() // 1
11. {
12. while(!gui\_shutdown\_message\_received()) // 2
13. {
14. get\_and\_process\_gui\_message(); // 3
15. std::packaged\_task<void()> task;
16. {
17. std::lock\_guard<std::mutex> lk(m);
18. if(tasks.empty()) // 4
19. continue;
20. task=std::move(tasks.front()); // 5
21. tasks.pop\_front();
22. }
23. task(); // 6
24. }
25. }
26. std::thread gui\_bg\_thread(gui\_thread);
27. template<typename Func>
28. std::future<void> post\_task\_for\_gui\_thread(Func f)
29. {
30. std::packaged\_task<void()> task(f); // 7
31. std::future<void> res=task.get\_future(); // 8
32. std::lock\_guard<std::mutex> lk(m);
33. tasks.push\_back(std::move(task)); // 9
34. return res; // 10
35. }

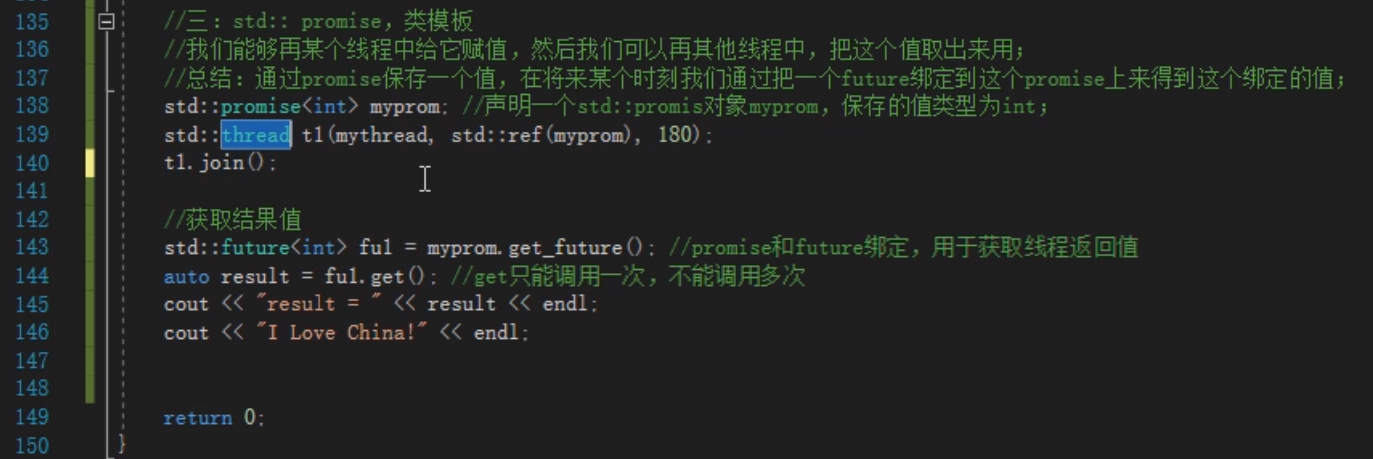
\*std::promise/std::future:期望值可以阻塞等待线程

+std::promise: 我们能够在某个线程中给它赋值，然后可以在其他线程中取出该值。

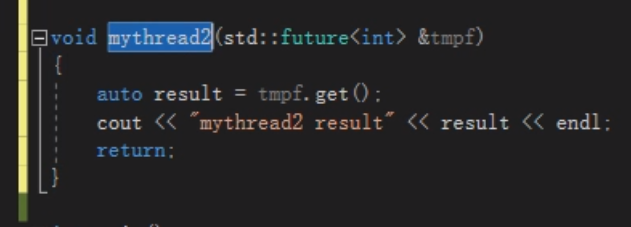
形参的形式返回线程中的值

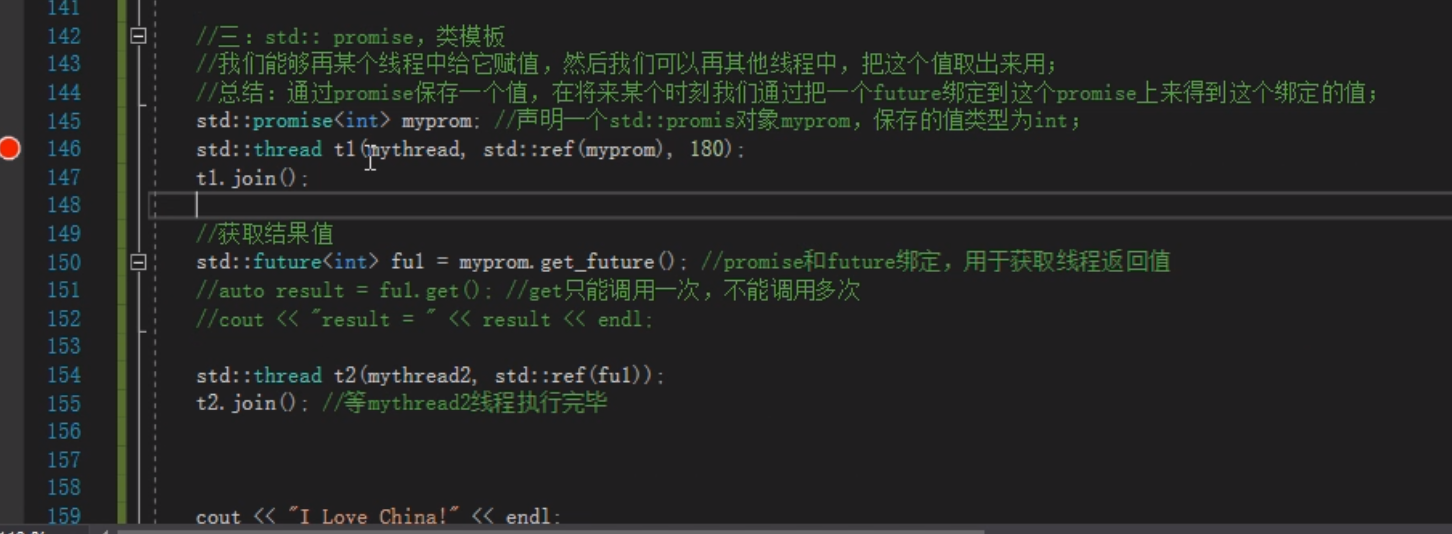


main函数调用方式



线程2中使用线程1返回的结果的例子。





1. #include <future>
2. void process\_connections(connection\_set& connections)
3. {
4. while(!done(connections)) // 1
5. {
6. for(connection\_iterator // 2
7. connection=connections.begin(),end=connections.end();
8. connection!=end;
9. ++connection)
10. {
11. if(connection->has\_incoming\_data()) // 3
12. {
13. data\_packet data=connection->incoming();
14. std::promise<payload\_type>& p=
15. connection->get\_promise(data.id); // 4
16. p.set\_value(data.payload);
17. }
18. if(connection->has\_outgoing\_data()) // 5
19. {
20. outgoing\_packet data=
21. connection->top\_of\_outgoing\_queue();
22. connection->send(data.payload);
23. data.promise.set\_value(true); // 6
24. }
25. }
26. }
27. }

\*std::shared\_future:多个线程等待事件

1. std::promise<std::string> p;
2. std::shared\_future<std::string> sf(p.get\_future()); // 1 隐式转移所有权

\*std::chrono::system\_clock::now(); 返回系统时钟的当前事件

\*std::chrono::steady\_clock; 稳定时钟

\*parallel\_quick\_sort 函数实现

\*atm 实现 ？？ then ; when\_all ;

\*锁存器:是一种同步对象，当它的计数器减为0时，它就准备就绪了。同一个线程可以对计数器递减多次，或多个线程对计数器递减一次，再或是其中有些线程对计数器有两次的递减。

+std::experimental::latch

+ latch done(thread\_count);

+ done.count\_down();

+ done.wait();

+ process\_data(...);

\*栅栏机制：可复用的同步装置，用于一组线程间的内部同步。每一个线程只能在每个周期内到达栅栏一次。当线程都到达栅栏时，会对线程进行阻塞，直到所有线程都到达栅栏处，这时阻塞将会被解除。栅栏可以复用----线程可以再次到达栅栏处，等待下一个周期的所有线程。

+每个任务线程完成任务后到达栅栏， 通过arrive\_and\_wait() 等待小组的其他成员线程。 当最后一个线程到达时，所有线程将被释放，并且栅栏会被重置。组中的线程继续接下来任务。

+ std::experimental::barrier sync(num\_threads);

+ sync.arrive\_and\_wait();

1. C++内存模型和原子类型操作

\*<atomic>:

\*std::atomic\_flag 可以实现 自旋互斥锁

class spinlock\_mutex

{

std::atomic\_flag flag;

public:

spinlock\_mutex() :

flag(ATOMIC\_FLAG\_INIT)

{}

void lock()

{

while (flag.test\_and\_set(std::memory\_order\_acquire));

}

void unlock()

{

flag.clear(std::memory\_order\_release);

}

};

\*先行发生(happens-before): ???

\*同步发生(synchronizes-with): ???

\*排序一致队列：std::memory\_order\_seq\_cst;

\*松散序列：std::memory\_order\_relaxed;

\*获取--释放序列：std::memory\_order\_consume, ...

+ 松散序列加强版，引入了同步。

\*栅栏：栅栏属于全局操作，执行栅栏操作可以影响到在线程中的其他原子操作。

+std::atomic\_thread\_fence(std::memory\_order\_acquire);

总结： 本章是底层实现， 学的模糊， 需要复习和加深学习。

1. 基于锁的并发数据结构设计

\*序列化(serialization): 线程轮流访问被保护的数据。这是对数据进行串行的访问，而非并发。

\*三个代码示例； 1.lock\_guard 2. lock\_guard + condition\_variable 3. 细粒度锁

\*对hash表做细粒度锁的操作(例子代码)

1. 无锁并发数据结构设计

\*无阻碍：如果所有其他线程都暂停了，任何给定的线程都将在一定时间内完成其操作。

\*无锁：如果多个线程对一个数据结构进行操作，经过一定时间后，其中一个线程将完成其操作。

\*无等待：即时有其他线程也在对该数据结构进行操作，每个线程都将在一定的时间内完成其操作。

\*无阻塞--有锁 ：？

\*对于设计无锁数据结构的直到建议：

+使用 std::memory\_order\_seq\_cst

+对无锁内存的回收策略

+小心ABA问题 （CAS？？？）一起深入了解

+识别忙等待循环和帮助其他线程

1. 并发代码设计

\*线程处理前对数据进行划分

+递归划分

+按职责来划分

\*多线程下有两个危险需要分离关注；

+第一个是 线程间共享着很多的数据，

+或者不同的线程要相互等待。

\*两个线程需要频繁的交流，在没有其他线程时，就可以将这两个线程合为一个线程。

\*当任务会应用到相同操作序列，去处理独立数据项时，可以使用 流水线系统进行并发。

\*影响并发代码性能的因素：

\*高竞争(high contention), 低竞争(low contention) ???

\*乒乓缓存(cache ping-pong)

\*伪共享发生的原因：

+某个线程所要访问的数据过于接近另一线程的数据

+数据布局相关的陷阱会直接影响单线程的性能。

\*实现并行版本的 for\_each

1. 高级线程管理

\*

1. 并行算法
2. 测试和调试多线程应用

\*并发相关的错误有两大类： 不必要的阻塞， 条件竞争