1

Std::thread::id

Std::this\_thread::get\_id()

std::mutex

std::lock\_guard<std::mutex>

Std::lock() 可以锁住多个

2

1. 避免嵌套锁。获取多个锁，使用一个 std::lock 来做这件事，避免产生死锁。
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

3

条件竞争处理：

Std::once\_flag

Std::call\_once

读写锁

Std::lock\_guard<boost::shared\_mutex> //写锁

Std::unique\_lock<boost::shared\_mutex>

Boost::shared\_lock<boost::shared\_mutex> //读锁

嵌套锁(不推荐)

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

Std::unique\_lock<std::recursive\_mutex>

4

void transfer( bank\_account &from, bank\_account &to, int amount )//这里缺少一个from==to的条件判断个人觉得

{

unique\_lock<mutex> lock1( from.mMutex, defer\_lock );//defer\_lock表示延迟加锁，此处只管理mutex

unique\_lock<mutex> lock2( to.mMutex, defer\_lock );

lock( lock1, lock2 );//lock一次性锁住多个mutex防止deadlock

from.iMoney -= amount;

to.iMoney += amount;

cout << "Transfer " << amount << " from "<< from.sName << " to " << to.sName << endl;

}

5

std::unique\_lock 与std::lock\_guard都能实现自动加锁与解锁功能，但是std::unique\_lock要比std::lock\_guard更灵活，但是更灵活的代价是占用空间相对更大一点且相对更慢一点。

unique\_lock 在使用上比lock\_guard更具有弹性，和 lock\_guard 相比，unique\_lock 主要的特色在于：

       unique\_lock 不一定要拥有 mutex，所以可以透过 default constructor 建立出一个空的 unique\_lock。

    unique\_lock 虽然一样不可复制（non-copyable），但是它是可以转移的（movable）。所以，unique\_lock 不但可以被函数回传，也可以放到 STL 的 container 里。

         另外，unique\_lock 也有提供 lock()、unlock() 等函数，可以用来加锁解锁mutex，也算是功能比较完整的地方。

         unique\_lock本身还可以用于std::lock参数，因为其具备lock、unlock、try\_lock成员函数,这些函数不仅完成针对mutex的操作还要更新mutex的状态。

6

Std::condition\_variable 和 std::mutex一起工作，首选

7

Std::condition\_variable\_any

可以和任何满足最低标准的互斥量一起工作，系统资源 的使用方面产生额外的开销

8

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;

}

}

}

9

Std::future<> 关联一个事件

Std::shared\_future<> 关联多个事件

10

Std::async 异步任务， 返回一个std::future对象

11

std::future<int> the\_answer = std::async(std::ref(find\_the\_answer), 4);

Std::launch::async

Std::launch::deferred

Std::ref

Std::async

Std::promise<>

Std::packaged\_task

Std::shared\_future 多个线程需要等待相同的事件的结果, 使用多个shared\_future对象来避免数据竞争。

Std::condition\_variable wait\_for(): 持续时间 wait\_until(): 绝对事件

Std::timed\_mutex, std::recursive\_timed\_mutex, try\_lock\_for() try\_lock\_until()

future<int> myFuture=async(task,10)

12

#include <iostream> // std::cout

#include <functional> // std::ref

#include <thread> // std::thread

#include <future> // std::promise, std::future

void print\_int (std::future<int>& fut) {

int x = fut.get();//当promise::set\_value()设置了promise的共享状态值后，fut将会通过future::get()获得该共享状态值，若promise没有设置该值那么fut.get()将会阻塞线程直到共享状态值被promise设置

std::cout << "value: " << x << '\n';//输出：value: 10

}

int main ()

{

std::promise<int> prom; //创建一个promise对象

std::future<int> fut = prom.get\_future(); //获取promise内部的future，fut将和promise共享promise中的共享状态，该共享状态用于返回计算结果

std::thread th1 (print\_int, std::ref(fut)); //创建一个线程，并通过引用方式将fut传到print\_int中

prom.set\_value (10); //设置共享状态值

//

th1.join();//等待子线程

return 0;

}

atomic

13

Std::atomic\_flag f = ATOMIC\_FLAG\_INIT;

例子： f.clear(std::memory\_order\_release);

Bool x = f.test\_and\_set();

14

Std::atomic<bool> b;

Bool x = b.load(std::memory\_order\_acquire);

b.store(true);

x = b.exchange(false, std::memory\_order\_acq\_rel);

15

Compare\_exchange\_weak() compare\_exchange\_strong()

16

结论就是:

想要性能，使用compare\_exchange\_weak+循环来处理。

想要简单，使用compare\_exchange\_strong。

如果是x86平台，两者没区别

如果想在移值的时候，拿到高性能，用compare\_exchange\_weak。

17

需要注意的是，weak = true表示弱CAS，在这种情况下，就是交换成功，也有可能返回失败。

一般而言，当你拿不准，就使用strong的版本。

18

Fetch\_add(), fetch\_sub();

19

Std::atomic<T> 接口：

Load(), store(), exchange(), compare\_exchange\_weak(), compare\_exchange\_strong()

20

内存序列选项可应用于对原子类型的操作：

Memory\_order\_relaxed 自由序列

Memory\_order\_consume 获取-释放序列

Memory\_order\_acquire 获取-释放序列

Memory\_order\_release 获取-释放序列

Memory\_order\_acq\_rel 获取-释放序列

Memory\_order\_seq\_cst 排序一致序列 （默认） --最昂贵,对所有线程进行全局同步

Std::kill\_dependency() 让编译器知道这里不需要重新读取该数组的内容， 谨慎使用

21

Std::atomic\_thread\_fence(std::memory\_order\_release); //内存栅栏

Std::atomic\_thread\_fense(std::memory\_order\_acquire);

22

序列化(serialzation):线程轮流访问被保护的数据。