C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B)

C++ 并发编程笔记(一)

C++ 并发编程笔记(一)

Posted by 敬方 on July 3, 2019

> *2019-7-3 20:56:48*

C++并发编程阅读笔记

参考链接:

- C++ 并发编程中文版 (https://chenxiaowei.gitbooks.io/cpp_concurrency_in_action/);
- 第二版在线地址 (https://legacy.gitbook.com/book/chenxiaowei/c-concurrency-in-action-second-edition-2019)

第1章 你好，C++的并发世界

1.1.1 何谓并发

并发：CPU交替使用时钟，模拟并发

并行：线程的真正并行执行

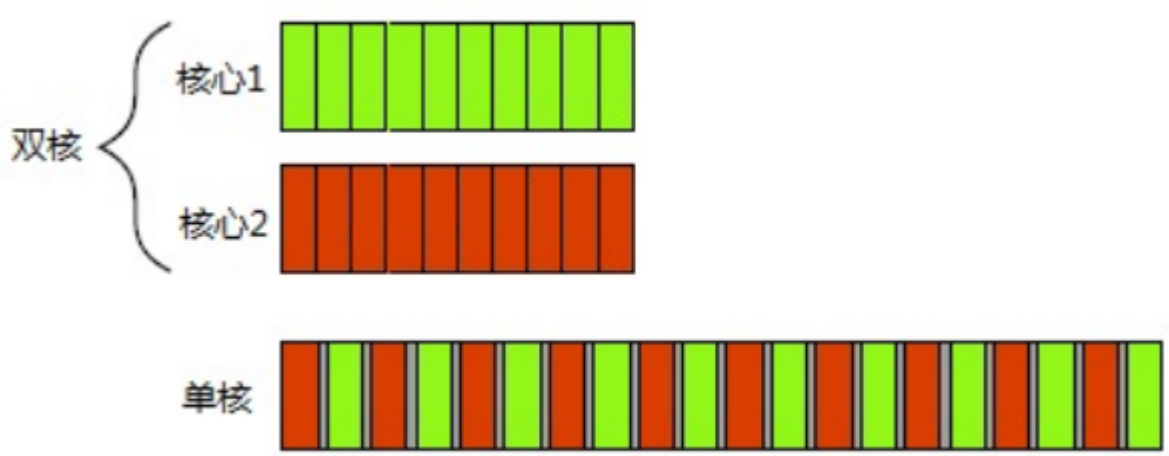

图 1.1 并发的两种方式：双核机器的真正并行 Vs. 单核机器的任务切换

1.1.2 并发的途径

- 多进程并发：将应用程序分为多个独立的进程,它们在同一时刻运行,就像同 时进行网页浏览和文字处理一样。
- 多线程并发：在单个进程中运行多个线程。线程很像轻量级的进程:每个线程相互独 立运行,且线程可以在不同的指令序列中运行。进程中的所有线程都共享地址空间, 并且所有线程访问到大部分数据———全局变量仍然是全局的,指针、对象的引用或数据可 以在线程之间传递。

注意：Linux 一般允许的最大堆栈为8-10M，需要自己设置扩容

程序并发的方式有2种：

- 程序执行并行：不同的线程执行，不同的过程
- 数据并行：不同的线程执行相同的程序，处理不同的数据。

1.4 开始入门

开始多线程编程的一个简单例子：

```
#include <iostream>
#include <thread>
void hello()
{
    std::cout<<"Hello Concurrent World\n";
}
int main()
{
    //设置线程初始函数名称

    std::thread t(hello);
    t.join();
}
```

第2章 线程管理

2.1 线程管理的基础

main()函数就是一个程序的主要线程，其它线程主要由函数的入口所决定。当线程执行完入口函数后，线程也会退出。

2.1.1 启动线程

参考链接: c++并发编程 (https://www.cnblogs.com/zhanghu52030/p/9166526.html)

线程在 `std::thread` 对象构造时就开始启动了，无返回值时，启动后自动结束，存在返回值时，参数传递完成后结束。

`std::thread` 允许使用带有函数调用符类型的实例传入 `std::thread` 类中，来替换默认的构造函数

```
class background_task
{
public:
    void operator()() const
    {
        do_something();
        do_something_else();
    }
};

background_task f;
std::thread my_thread(f);
```

注意： 当把函数对象传入到线程构造函数中时，如果你传递一个临时变量，而不是一个命名的变 量;C++编译器会将其解析为函数声明,而不是类型对象的定义。

```
std::thread my_thread(background_task());

// 声明了一个名为my_thread的函数，函数指针指向没有参数并返回background_task对象的函数；返回一个std::thread对象的函数，而非启动了一个线程

//使用多组括号或者新的统一初始化语法可以避免这个问题

std::thread my_thread((background_task()));
std::thread my_thread{background_task()};

//使用lambda表达式也能避免这个问题

std::thread my_thread([]{
    do_something();
    do_something_else();
});
```

如果没有指定线程的销毁 `std::thread` 对象会在析构函数中调用 `std::terminate()` 进行对象销毁和析构。

注意:

- **如果需要自己指定线程销毁，必须在 `std::thread` 对象销毁之前决定**
- **如果不主动等待线程结束，就必须保证线程结束之前，数据的有效性，避免线程还未结束，函数已经退出，变量成为销毁的局部变量**

```
struct func
{
    int& i;
    func(int& i_) :i(i_){}
    void operator() ()
    {
        for (unsigned   j=0 ;   j<1000000   ;   ++j)
        {
                do_something(i);
                // 1. 潜在访问隐患:悬空引用

        }
    }
};
void oops()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
    // 2. 不等待线程结束

}                                                                   // 3. 新线程可能还在运行
```

对于上述的处理方法是将数据复制到线程中，使得线程函数的功能齐全。而非复制到共享数据中。此外,可以通过join()函数来确保线程在函数完成前结束。但是join()只是简单粗暴的等待线程完成或者不等待。等待线程完成的例子如下：

```
struct  func;   //定义在清单2.1中

void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    //使用异常捕获，保证访问本地状态的线程退出后，函数才结束

    try
    {
        do_something_in_current_thread();
    }catch(...)
    {
        t.join();       /* 等待线程结束 */
        throw;
    }
    t.join();       /* 等待线程结束 */

}

/*使用“资源获取即初始化方式”(RAII,Resource Acquisition Is Initialization)等待线程退出*/

class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_):t(t_)
    {

    }
    ~thread_guard()
    {
        //判断线程是否已经开始加入

        if(t.joinable())
        {
            //析构函数中使得线程被加入到原始线程中
```

```
            t.join();

        }
    }
    /* 禁止使用默认的拷贝构造函数 */
    thread_guard(thread_guard&)=delete;
    /* 禁止使用赋值函数 */
    thread_guard& operator=(thread_guard const&)=delete
};

struct func;

void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_function);
    thread_guard g(t);
    thread_guard g(t);
    do_something_in_current_thread();
}
//函数结束，局部对象开始逆序销毁，
```

2.1.4 后台运行线程

使用 `detach()` 会让线程在后台运行。意味着主线程不能与该线程产生直接互交。使得线程实现分离，并且由C++编译器实现资源的回收。

通常称分离线程为守护线程(daemon threads);UNIX中守护线程是指,没有任何显式的用户接 口,并在后台运行的线程。这种线程的特点就是长时间运行;线程的生命周期可能会从某一个应用起始到结束,可能会在后台监视文件系统,还有可能对缓存进行清理,亦或对数据结构进行优化。另一方面,分离线程的另一方面只能确定线程什么时候结束,发后即忘(fire and forget)的任务就使用到线程的这种方式。

```
std::thread t(do_background_work);
t.detach();
assert(!t.joinable());

//线程分离之后与之前的主线程无关

//多线程处理文档的示例：

void edit_document(std::string   const&  filename)
{
    open_document_and_display_gui(filename);
    while(!done_editing())
    {
        user_command  cmd=get_user_input();
        if(cmd.type==open_new_document)
        {
            std::string const   new_name=get_filename_from_user();
            //启动一个新线程开始显示和处理文档

            std::thread t(edit_document,new_name);
            //分离线程

            t.detach();
        }else{
            process_user_input(cmd);
        }
    }
}
```

2.2 向线程函数传递参数

参考链接： C++11的6种内存序总结 (https://blog.csdn.net/lvdan1/article/details/54098559);

线程调用的默认参数要拷贝到线程独立内存中，几十参数是引用的形式，也可以在新线程中进行访问。

```
void f(int i,std::string const& s);
std::thread t(f,3,"hello");
```

注意： **指向动态变量的指针作为参数传递给线程的情况，可能在传递过程中产生未定义的行为。使得程序崩溃**

```
void f(int i,std::string const&  s);
void oops(int some_param)
{
    //指针变量

    char buffer[1024];
    //指针变量指向输入参数

    sprintf(buffer,"%i",some_param);
    //创建线程，输入指针函数

    std::thread t(f,3,buffer);
    /* 线程执行与主线程分离 */
    t.detach();
}

//在从char*到std::string类型转换的过程中，函数很有可能在转化成功之前崩溃；但是`std::thread`的构造函数会复制提供的变量，就只复制了没有转换成期

//只要能在构造函数之前将字面值转换为`string`对象就可以了。

void f(int i,std::string const& s);

void not_oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    //使用显示转换避免指针悬垂

    std::thread t(f,3,std::string(buffer));
    t.detach();
}
```

注意:

- 当想要在线程构造函数中传递参数为引用参数，使得变量在线程中进行更改的时候，需要使用 `std::ref` 将参数转换成为引用的形式，避免其在构造的过程中使用默认拷贝。如 `std::thread t(update_data_for_widget,w,std::ref(data));` ;这样data就在线程前后发生了改变。
- 可以使用 `std::move` 将一个参数，移动到线程中去。

```
//输如参数是一个只允许一个使用的指针

void process_big_object(std::unique_ptr<big_object>);

std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
//使用move函数，进行移动语义，右值引用；将指针的所有权，交给线程内部的函数库

std::thread t(process_big_object,std::move(p));
```

2.3 转移线程所有权

为了保证线程控制句柄的有效性， `std::thread` 支持使用 `std:move()` 函数实现函数的转移，**但是线程的拷贝构造和赋值操作符决定了，它在进行复制时会先使用 `std::terminate()` 强制结束原来已有的线程，再接受新句柄；即说明：不能通过赋一个新值给 std::thread 对象的方式来“丢弃”一个线程。**

```
void some_function();
void some_other_function();
//直接构造线程对象

std::thread t1(some_function);
//移动线程句柄到t2，之后t1无用

std::thread t2=std::move(t1);
//创建拷贝的时候，默认隐式调用std::move()函数

t1=std::thread(some_other_function);
//创建线程对象

std::thread t3;
//t2线程句柄，交给t3

t3=std::move(t2);
//t1执行std::terminate()终止程序继续运行，保证与std::thread的析构函数行为一致，需要在

t1=std::move(t3);
```

建议的线程拷贝使用如下：

```
class scoped_thread
{
    std::thread t;
public:
    //直接获取线程的句柄函数

    explicit scoped_thread(std::thread t_):
                t(std::move(t_))
    {
        //线程以及结束过一次就返回失败

        if(!t.joinable())
            throw   std::logic_error(“No thread”);
    }
    //析构函数结束线程

    ~scoped_thread()
    {
        t.join();
    }
    //拷贝构造函数

    scoped_thread(scoped_thread const&)=delete;
    //赋值操作符

    scoped_thread&  operator=(scoped_thread const&)=delete;
};
struct  func;   // 定义在清单2.1中
void f()
{
    int some_local_state;
    //条用拷贝构造函数

    scoped_thread   t(std::thread(func(some_local_state)));
    //执行程序

    do_something_in_current_thread();
}
```

```
void do_work(unsigned id);

void f1()
{
    std::vector<std::thread> threads;
    for(unsigned i=0; i<20; ++i)
        {
            //产生线程

            threads.push_back(std::thread(do_work,i));
        }
        //对每个线程调用join()；注意这里的mem_fn直接获取对象的函数指针

        std::for_each(threads.begin(),threads.end(),std::mem_fn(&std::thread::join));
}
```

2.4 运行时决定线程数量

`std::accumulate` 可以用来计算容器中元素的和，总体思路是利用 `std::thread::hardware_concurrency()` 返回并发在一个程序中的线程数量，由此来决定最终的线程数目。

```
template<typename Iterator,typename T>
//定义小块计算的类

struct accumulate_block
{
    void operator()(Iterator frist,Iterator last,T& result)
    {
        //使用计算结果

        result=std::accumulate(first,last,result);
    }
};

//定义并行计算的模板函数

template<typename Iterator,typename T>

T parallel_accumulate(Iterator first,Iterator last,T init)
{
    //计算元素的总数量

    unsigned long const length=std::distance(first,last);
    //长度为0直接返回初始值

    if(!length)
    {
        return init;
    }
    //最小线程数量

    unsigned long const min_per_thread=25;
    //最大线程数量

    unsigned long const max_threads=(length+min_per_thread-1)/min_per_thread;
    //计算正在运行的线程数量

    unsigned long const hardware_threads=std::thread::hardware_concurrency();
    //计算真正的线程数量
```

```
unsigned long const num_threads=std::min(hardware_threads!=0?hardware_threads:2,max_threads);
//计算分块运行的数量

unsigned long const block_size=length/num_threads;
//准备结果数据

std::vector<T> results(num_threads);
//准备线程

std::vector<std::thread> threads(num_threads-1);
//遍历初始化数据
Iterator block_start=frist;

for(unsigned long i=0;i<(num_threads-1);++i)
{
    //临时分块的终点

    Iterator block_end=block_start;
    //移动迭代器将其指向末尾

    std::advance(block_end,block_size);
    //初始化线程

    threads[i]=std::thread(
        accumulate_block<Iterator,T>(),
        block_start,
        block_end,
        std::ref(results[i])
        );
    //更新迭代器位置

    block_start=block_end;
}
//将剩余的数全部分到最后一个线程块

accumulate_block<Iterator,T>()(
    block_start,last,results[num_threads-1]
    );
//等待创建线程的完成
```

```
    std::for_each(threads.begin(),
        threads.end(),
        std::mem_fn(&std::thread::join)
        );
    //计算求和

    return std::accumulate(results.begin(),results.end(),init);
}
//注意：这里的迭代器都必须是前向迭代器，并且必须保证T存在默认的构造函数
```

2.5 识别线程

线程标示类型是 `std::thread::id` ， 可以通过两种方式进行检索。

- `std::thread::get_id()` 函数直接来获取。当没有绑定线程时，函数返回 `std::thread::type`
- 在当前线程中调用 `std::this_thread::get_id()` 获取当前的线程标识符

第3章 线程间共享数据

参考链接： C++并发编程学习——3.在线程间共享数据 (https://blog.csdn.net/qq_37303711/article/details/78576671);线程间共享数据 (https://www.jianshu.com/p/c342d65c951c);

3.1 共享数据带来的问题

对于只读的数据是不存在操作之间的共享数据问题的，但是对于读写操作或者删除操作，需要添加而锁实现，数据的互斥和共享。注意使用条件竞争操作，避免恶性条件竞争。

避免思路：

- 对数据结构采取某种保护机制，确保只有进行修改的线程才能看到不变量被破坏时的中间状态。
- 对数据结构和不变量的设计进行修改，修改完成的结构必须能完成一系列不可分割的变化。即 **无锁编程**。
- ,使用事务的方式去处理数据结构的更新(这里的”处理”就如同对数据库进行更新一样)。所需的一些数据和读取都存储在事务日志中,然后将之前的操作合为一步,再进行提交。在被另外一个线程修改后，提交就无法进行。这成为“软件事务内存”。但是C+++中没有对 `STM` 进行支持。

C++中保护共享数据的最基本方式是使用 C++标准库提供的互斥量

3.2 使用互斥两保护共享数据

3.2.1 C++中使用互斥量

c++中通过使用 `std::mutex` 创建互斥量，通过调用成员函数 `lock()/unlock()` 进行上锁/解锁。同时可以使用 `std::lock_guard()` ，在构造的时候提供已锁的互斥，并在析构的时候进行解锁，从而保证总是会被正确的解锁。 使用示例：

```
#include <list>
#include <mutex>
#include <algorithm>
//创建全局的互斥保护变量

std::list<int> some_list;
//互斥信号变量

std::mutex some_mutex;

void add_to_list(int new_value)
{
    //使用std::lock_guard互斥访问数据

    std::lock_guard<std::mutex> guard(some_mutex);
    //添加数据

    some_param.push_back(new_value);
}

bool list_contains(int value_to_find)
{
    //互斥访问数据，为信号变量加锁

    std::lock_guard<std::mutex> guard(some_mutex);
    //查找对应的变量

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

//大多数情况下，互斥变量会直接定义在同一个类中，而不是定义成全局变量。互斥量 和要保护的数据都必须定义为priavte成员。

//注意：函数返回的是被保护数据或者成员的指针时
```

3.2.2 精心组织代码来保护共享数据

针对返回指针或者引用会使得成员变量失去保护。这样互斥变量形同虚设。但是这些只要你确保没有在使用过程中显式调用就没有问题，但是更危险的是 **将一个保护数据作为运行时参数** 例如下面，无意中使用了保护数据的引用

```
class some_data
{
    int a;
    std::string b;
public:
    void do_something();
};

class data_warpper
{
private:
    some_data data;
    std::mutex m;
public:
    template<typename Function>
    void process_data(Function func)
    {
        std::lock_guard<std::mutex> 1(m);
        //在这里传递保护数据给用户函数

        func(data);
    }
}
//非保护数据

some_data* unprotected;
//恶意拷贝数据

void malicious_function(some_data& protected_data)
{
    unprotected=&protected_data;
}
//数据指针

data_wrapper x;

void foo()
{
    //传递一个恶意值，将类中的保护成员赋值给外部的非保护变量
```

```
    x.process_data(malicious_function);
    //执行操作，更改 内部的data值

    nprotected->do_something();
}
```

3.2.3 发现接口内在的条件竞争

针对上面的问题，比较好的解决方案是将方法和接口封装好，避免在外部调用使得方法操作的安全性。例如下面一个stack的实现

```
//定义模板类和方法

template<typename   T,typename  Container=std::deque<T> >
class   stack
{
public:
        explicit    stack(const Container&);
        explicit    stack(Container&&   =   Container());
        template    <class  Alloc>  explicit    stack(const Alloc&);
        template    <class  Alloc>  stack(const Container&, const   Alloc&);
        template    <class  Alloc>  stack(Container&&,  const   Alloc&);
        template    <class  Alloc>  stack(stack&&,  const   Alloc&);
        bool    empty() const;
        size_t  size()  const;
        T&  top();
        T   const&  top()   const;
        void    push(T  const&);
        void    push(T&&);
        void    pop();
        void    swap(stack&&);
};
```

为了防止多线程时的empty()和size()函数的操作，需要进行变量的互斥保护;保护的主要思想是通过线程锁，使得两个线程的存取操作具有原子事物性，操作的顺序先后固定。下面是一种可能的顺序

Thread A	**Thread B**
if (!s.empty);	
int const value=s.top();	
s.pop()	
do_something(value);	s.pop()

标准库的stack类使用将这个操作分为了两个部分： 1. 获取顶部元素(top());2.从栈中移除(pop())。这样在不安全的将元素拷贝出去的情况下，栈中的这个数据依旧存在，没有丢失。

下面介绍一些其它的方法：

1. 传入一个引用，作为想要的弹出值来进行使用。

```
std::vector<int> result;
some_stack.pop(result);
```

1. 无异常抛出的拷贝构造函数或则移动构造函数
2. 返回指向弹出值的指针

这样可以方便自由拷贝，并且不会产生异常，缺点是返回一个指针需要对内存分配进行管理，对于简单数据类型，内存管理的开销远大于直接返回值。通常使用 `std::shared_ptr` 进行操作。

3. “选项1+选项2”或“选项1+选项3”

下面是一个线程安全的堆栈的定义

```
#include <exception>
#include <memory>
#include <mutex>
#include <stack>
// 定义空栈函数

struct empty_stack: std::exception
{
    const char* what() const throw() {
        return "empty stack";
    };

};
//定义模板类

template<typename T>
class threadsafe_stack {
private:
    //保护成员变量

    std::stack<T> data;
    //互斥信号量

    mutable std::mutex m;
public:
    threadsafe_stack():data(std::satck<T>()){}

    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        //在构造体中执行拷贝

        data=other.data;
    }
    //禁用默认赋值

    threadsafe_stack& operator=(const threadsafe_stack&)=delete;
    //添加
```

```
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(new_value);
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        //在调用pop前，检查栈是否为空

        if (data.empty())
        {
            throw empty_stack();
        }
        //在修改堆栈之前，分配出返回值

        std::shared_ptr<T> const res(std::make_shared<T>(data.top()));
        data.pop();
        return res;
    }

    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();

        value=data.top();
        data.pop();
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }

};

//上面的检查拷贝操作全部都加上了锁，虽然性能有所损耗，但是实现了线程级安全。
```

3.2.4 死锁：问题描述及解决方案

问题描述在操作系统中有着较为详细的叙述，再次不做过多的解释。

解决问题的主要思路是：

- 死锁预防：
  - 破坏互斥条件：资源互斥使用，无法破坏互斥条件
  - 破坏不剥夺条件：后来者，强制抢夺资源，但会增加系统开销，降低吞吐量
  - 破坏请求和保持条件：不再一直请求和保持状态；会严重浪费资源，还可能导致饥饿现象
  - 破坏循环条件：会浪费系统资源，并造成编程不便
- 死锁避免：
  - 安全状态：找到一个分配资源的序列能让所有进程都顺利完成。
  - 银行家算法：采用预分配策略监察分配完成时系统是否处在安全状态。
- 死锁检测： 利用死锁定力简化资源分配图以检测死锁的存在。
- 死锁解除：
  - 资源剥夺法：挂起某些死锁进程并抢夺它的资源，以便让其他进程继续推进。
  - 撤销进程法：强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源。
  - 进程回退法：让进程回退到足以回避死锁的地步。

对于线程的具体实现思路是：让两个互斥量，总是以相同的顺序上锁。避免多线程对数据的修改时的死锁。c++标准库，提供了 `lock` 方法，可以一次锁住多个互斥量，并没有死锁风险。下面是一个简单的使用示例：

```
#include <mutex>
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);

class X {
private:
    //目标对象

    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;
        //锁住两个互斥变量

        std::lock(lhs.m,rhs.m);
        //表示 std::lock_guard 对象可获取锁之外,还将锁交由 std::lock_guard 对象管理,而不需要std::lock_guard 对象再去构建新的锁。保证退出时


        std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);

        std::lock_guard<std::mutex> lock_b(rhs.m,sth::adopt_lock);

        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

3.2.5 避免死锁的进阶指导

避免相关线程发生死锁的建议：

- 避免嵌套锁：当前线程已经取得锁时，在未解锁前，不获取新锁。

- 避免在持有锁的时候调用用户提供的代码，用户行为很容易产生未定义行为。
- 使用固定顺序获取锁：当线程需要多个资源配合完成，而不得不使用多个锁时，尽量将锁按照顺序执行，避免死锁。在遍历需要获取多个锁的时候，必须按照固定顺序得到多个相关锁。每个线程的遍历次序必须相同。
- 使用锁的层次结构：提供对于运行时鱼等是否被坚持的检查。

```
//使用层次锁来避免死锁

//层级锁，层级为10000

hierarchical_mutex  high_level_mutex(10000);
//层级锁，层级为5000

hierarchical_mutex  high_level_mutex(5000);

int do_low_level_stuff();

int low_level_func()
{
    //层级锁互斥

    std::lock_guard<hierarchical_mutex> lk(low_level_mutex);
    return do_low_level_stuff();
}
void high_level_stuff(int some_param);

void high_level_func()
{
    //让hight_level_mutex上锁

    std::lock_guard<hierarchical_mutex> lk(high_level_mutex);
    //调用low_level_func()会对low_level_mutex上锁

    high_level_stuff(low_level_func());
}
//线程a遵守层级规则，所以运行灭有问题

void thread_a()
{
    high_level_func();
}

hierarchical_mutex  other_mutex(100);
void do_other_stuff();
void other_stuff()
```

```
{
    high_level_func();
    do_other_stuff();
}
//线程b无视层级规则

void thread_b()
{
    //先锁住other_mutex；它的层级只有100，意味着低层级已经被保护，再使用high_level_func()，会造成此hierarchical_mutex将会产生一个错误。导致

    std::lock_guard<hierarchical_mutex> lk(other_mutex);
    other_stuff();
}
```

简单层级的互斥实现

```
class hierarchical_mutex
{
    //互斥信号量

    std::mutex internal_mutex;
    //接受的层级值

    unsigned long const hierarchical_value;
    //原来的层级值

    unsigned long previous_hierarchy_value;
    //静态指针，指向线程的层级值

    static thread_local unsigned long this_thread_hierarchy_value;
    void check_for_hierarchy_violation()
    {
        //检查层级值保持递减的序列

        if(this_thread_hierarchy_value<=hierarchical_value)
        {
            throw std::logic_error(“mutex   hierarchy   violated”);
        }
    }

    void update_hierachy_value()
    {
        //更改层级值前，先将原来的层级值保存

        previous_hierarchy_value=this_thread_hierarchy_value;
        //更改当前的层级值

        this_thread_hierarchy_value=hierarchical_value;
    }
public:
    explicit hierarchical_mutex(unsigned long value):hierarchical_value(value),previous_hierarchy_value(0)
    {}
    ~hierarchical_mutex();
    //加锁函数
```

```
    void lock()
    {
        //确认当前值小于新值

        check_for_hierarchy_violation();
        //当前线程加锁

        internal_mutex.lock();

        update_hierachy_value();

        update_hierachy_value();
    }
    void unlock()
    {
        //还原原来的层级值

        this_thread_hierarchy_value=previous_hierarchy_value;
        internal_mutex.unlock();
    }
    bool try_lock()
    {
        check_for_hierarchy_violation();
        if(!internal_mutex.try_lock())
        {
            return false;
        }

        update_hierachy_value();
        return true;
    }
};
//直接将信号变量，初始化为最大值

thread_local unsigned long  hierarchical_mutex::this_thread_hierarchy_value(ULONG_MAX);
```

3.2.6 std::unique_lock– 灵活的锁

`std::unique_lock` 可以根据传入的第二个参数的不同，对析构函数进行管理，使用 `std::defer_lock` 作为参数的时候，表明互斥量应该保持解锁状态，方便被其它线程加锁。但是 `std::unique_lock` 会占用比较多的空间,并且比 `std::lock_guard` 稍慢一些。并且 `std::unique_lock` 实例不带互斥量：信息一倍存储，且被更新。

`std::lock()` 和 `std::unique_lock` 的使用

```
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::def_lock 留下未上锁的互斥量

        std::unique_lock<std::mutex>    lock_a(lhs.m,std::defer_lock);
        std::unique_lock<std::mutex>    lock_b(rhs.m,std::defer_lock);
        // 互斥量在这里上锁

        std::lock(lock_a,lock_b);
        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

3.2.7 不同域中互斥量所有权的传递

允许一个函数去锁住一个互斥量,并且将所有权移到调用者上,所以调用者可以在这个锁保护的范围内执行额外的动作。

下面的程序片段展示了:函数get_lock()锁住了互斥量,然后准备数据,返回锁的调用函数:

```
std::unique_lock<std::mutex> get_lock()
{
    extern  std::mutex  some_mutex;
    std::unique_lock<std::mutex>    lk(some_mutex);
    prepare_data();
    //声明自动变量，不需要调用`std::move()`直接返回信号量的锁

    return  lk;
}

void process_data()
{
    //获取线程的信号量锁转移std::unique_lock实例的所有权

    std::unique_lock<std::mutex> lk(get_lock());
    do_something();
}
```

3.2.8 锁的粒度

锁的粒度通常用来描述通过一个锁保护着的数据量的大小。细粒度保护较小的数据，粗粒度保护较多的数据。在操作过程中，自己要注意锁的粒度问题。

3.3 保护共享数据的替代设施

除了互斥量之外，还有其它的保护共享数据的方式。C++提供了一种纯粹保护共享数据初始化过程的机制。

使用一个互斥量的延迟初始化(线程安全)过程

```
std::shared_ptr<some_resource>  resource_ptr;

std::mutex  resource_mutex;

void foo()
{
    //所有线程在此序列化

    std::unique_lock<std::mutex> lk(resource_mutex);
    if(!resource_ptr)
    {
        //初始化资源

        resource_ptr.reset(new some_resource);
    }
    lk.unlock()
    //执行操作

    resource_ptr->do_something();
}
```

C++ 提供了 `std::once_flag` 和 `std::call_once` 来处理多线程的读写同步问题。比起锁住互斥量,并显式的检查指针,每个线程只需要使用 `std::call_once` ,在 `std::call_once` 的结束时,就能安全的知道指针已经被其他的线程初始化了。使用 `std::call_once` 比显式使用互斥量消耗的资源更少,特别是当初始化完成后。下面是一个使用示例:

```
std::shared_ptr<some_resource>  resource_ptr;

std::once_flag  resource_flag;
void init_resource()
{
    resource_ptr.reset(new  some_resource);
}
void foo()
{
    //可以完整的进行一次初始化

    std::call_once(resource_flag,init_resource);
    resource_ptr->do_something();
}
```

使用 std::call_once 作为类成员的延迟初始化(线程安全)

```
class   X
{
private:
    connection_info connection_details;
    connection_handle   connection;
    std::once_flag  connection_init_flag;
    //打开数据库

    void open_connection()
    {
        connection=connection_manager.open(connection_details);
    }
public:
    X(connection_info const& connection_details_):connection_details(connection_details_)
        {}
    //完成线程的初始化

    void send_data(data_packet const&  data)
    {
        std::call_once(connection_init_flag,&X::open_connection,this);
        connection.send_data(data);
    }
    //接受数据，返回数据包

    data_packet receive_data()
    {
        //传递一个额外的参数，完成这个操作

        std::call_once(connection_init_flag,&X::open_connection,this);
        return  connection.receive_data();
    }
};
```

3.3.2 保护很少更新的数据结构

类似于DNS域名缓存表，在boost库中也存在缓存可以被多个线程访问，并且每次更新的都是少数数据。可以使用 `boost::share_mutex` 来进行同步。

使用 boost::shared_mutex 对数据结构进行保护的简单DNS缓存

```
#include <map>
#include <string>
#include <mutex>
#include <boost/thread/shared_mutex.hpp>
class dns_entry;

class dns_cache
{
    //DNS缓冲表

    std::map<std::string,dns_entry> entries;
    mutable boost::shared_mutex entry_mutex;
public:
    dns_entry find_entry(std::string  const&  domain) const
    {
        //创建共享锁

        boost::shared_lock<boost::shared_mutex> lk(entry_mutex);
        //获取查找结果迭代器

        std::map<std::string,dns_entry>::const_iterator const  it=entries.find(domain);
        return  (it==entries.end())?dns_entry():it->second;
    }
    void update_or_add_entry(std::string const& domain,
                             dns_entry const& dns_details)
    {
        //线程独占锁，会阻止其它线程对数据结构进行修改，并阻止线程调用find_entry()

        std::lock_guard<boost::shared_mutex> lk(entry_mutex);
        entries[domain]=dns_details;
    }

}
```

3.3.3 嵌套锁

C++ 标准库提供了 `std::recursive_mutex` 类。其功能与 `std::mutex` 类似,除了你可以从同一线程的单个实例上获取多个锁。互斥量锁住其他线程前,你必须释放你拥有的所有锁,所以当你调用 `lock()` 三次时,你也必须调用 `unlock()` 三次。正确使用 `std::lock_guard<std::recursive_mutex>` 和 `std::unique_lock<std::recursive_mutex>` 可以帮你处理这些问题。主要用于成员函数嵌套调用的时候的互斥。

推荐的使用方式是：提取处一个函数作为类的私有成员，并且让其它成员函数都进行调用，这个私有成员函数不会对互斥量进行上锁。

PREVIOUS
C++ PRIMER 学习笔记(十)
(/2019/06/15/CPLUSPLUS_PRIMER_LEARN_NOTE_10/)

NEXT
C++ 并发编程笔记(二)
(/2019/07/06/CPLUSPLUS_CONCURRENCY_IN_ACTION_02/)

FEATURED TAGS (/tags/)

C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B) C/C++ (/tags/#C/C++)
后台开发 (/tags/#%E5%90%8E%E5%8F%B0%E5%BC%80%E5%8F%91) C (/tags/#C) 网络编程 (/tags/#%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B)

STL源码解析 (/tags/#STL%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90) Linux (/tags/#Linux)
操作系统 (/tags/#%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 程序设计 (/tags/#%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1)
优化 (/tags/#%E4%BC%98%E5%8C%96) UML (/tags/#UML) UNIX (/tags/#UNIX) 学习笔记 (/tags/#%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0)
面试 (/tags/#%E9%9D%A2%E8%AF%95) Java (/tags/#Java) 读书笔记 (/tags/#%E8%AF%BB%E4%B9%A6%E7%AC%94%E8%AE%B0) go (/tags/#go)
阅读笔记 (/tags/#%E9%98%85%E8%AF%BB%E7%AC%94%E8%AE%B0)

FRIENDS

WY (http://zhengwuyang.com) 简书·JF (http://www.jianshu.com/u/e71990ada2fd) Apple (https://apple.com)
Apple Developer (https://developer.apple.com/)

● (https://www.facebook.com/wangpengcheng) ● (https://github.com/wangpengcheng)

Theme on GitHub (https://github.com/wangpengcheng/wangpengcheng.github.io.git) | Star 12