

张永祥

C++并发编程实战

读书笔记

# 2线程管理

## 2.1 基本的线程管理

主线程main开启其他线程，并在return出结束main线程，其他线程执行完函数也将结束。

### 2.1.1 启动线程

启动线程的操作就是定义一个thread对象，这个对象的形参接收函数、函数对象与lambda表达式，线程将这些函数复制到线程空间并执行他们；例如thread t(function);

Notice:

1. 线程对象的定义有多种方式，传统的定义方式有点像定义一个函数，参数接受的是一个未命名对象，还有2种定义方式，他们通过对参数进行命名化避免定义函数的那种形式：

Thread t1((task()))与thread t2{task()}

1. 线程对象可以使用lambda表达式，因为lambda表达式是一种未命名的函数对象，所以可以作为线程处理；
2. 线程在启动后，执行结束前，要决定是join还是detach，否则将会调用terminate结束程序，及时是发生了异常；
3. 线程中有可能会访问外层函数的局部对象，当detach后，原函数执行结束，局部对象销毁，线程的访问就会出现问题，所以要避免这种情况发生；



### 2.1.2 等待线程执行结束在结束

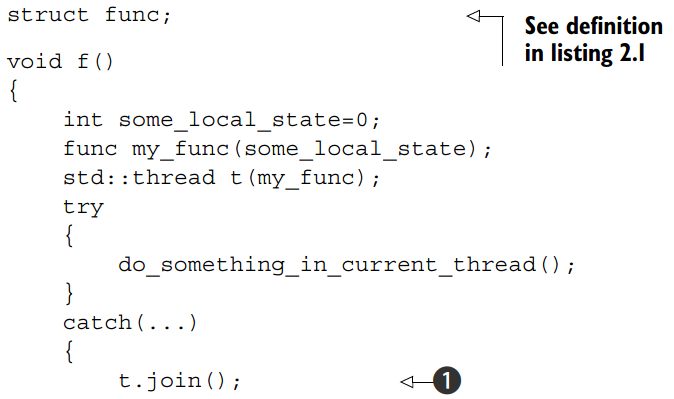
等待线程结束就是在线程对象上调用thread.join()函数。

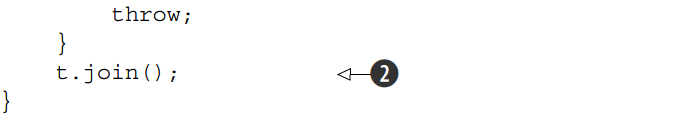
Notice：

1. 线程对象在调用join()函数之后，就是不可连接的状态。

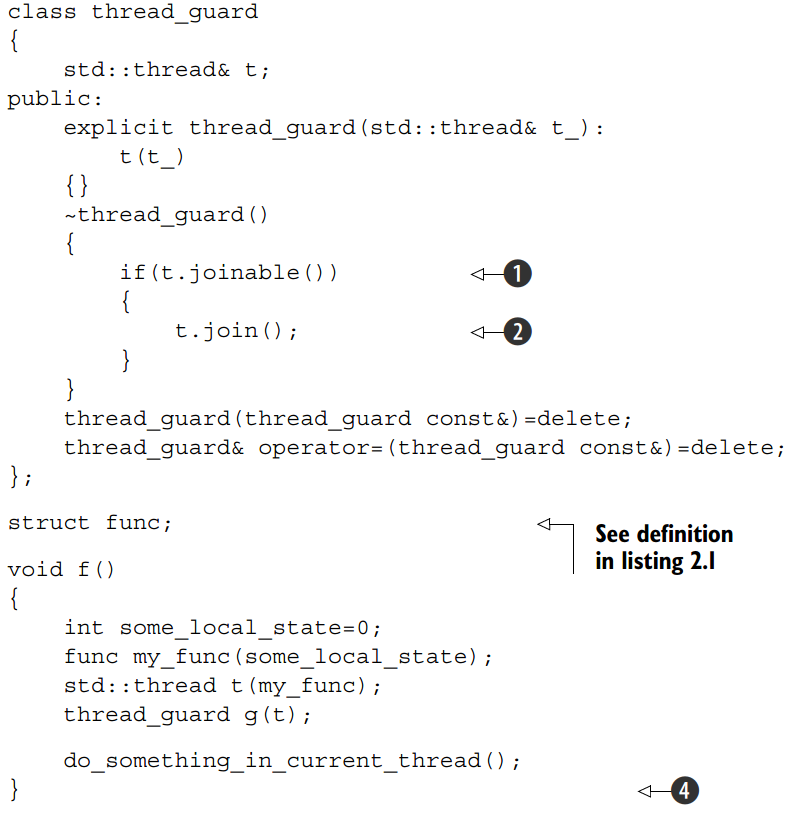
### 2.1.3 异常环境下的等待

因为线程在启动后销毁前要决定是否join或者detach，所以如果决定前发生异常，线程执行结束后就会让程序结束，所以要在异常与费异常的情况下都要调用join()函数，如：





这种方案并不理想，因为有可能在异常发生前，线程执行完了，标准的做法是提供一个线程守护类，在线程守护类的析构函数中进行join操作，当异常发生时，类的析构函数自动启动，然后判断此时的线程joinable()状态决定是否join操作。



Notice：

1. 这种在异常下的考虑是确保线程一定会执行join操作，完成线程的执行。
2. 加入是detach，线程对象与线程之间失去联系，函数结束，线程对象销毁，此时线程仍在后台执行，线程对象并不会发出terminate()操作。

### 2.1.4 后台运行线程

后台线程与Unix中的daemon process类似。就在后台中运行，不能被程序重新引用，不能通信，所有权是C++运行时库的，负责线程结束时资源回收，使用thread.detach()完成线程分离。

Notice：

1. Detach操作前需要确保thread对于当前程序来说是可引用的并且确实有一个线程运行体与线程对象向关联，这条对于join与joinable()也成立；
2. Detach的线程运行体适合数据操作并行化。

## 2.2 向线程中的函数传递参数

线程传参的形式类似于线程对象的构造函数，函数参数都包含在构造参数中，形式如下：

Void f(int I,std::string const &s);

Std::thread t(f,3,”hello”);

Notice:

1. 不论什么类型，参数都是首先按照传值的方式复制到线程的内存空间，再传入到函数里面去，所以面临传参的一些问题，这是按照实际的值类型，这里的线程对于函数的引用什么的是不敏感的；例如上例中复制了“hello”到线程内存空间中；对于指针类型，需要注意作用域的问题，可以通过指针构造新的对象完成传值而不是传指针；
2. 参数的类型如果不完全匹配，那么需要转换，但是转换过程费时，如果本地函数执行完了，本地变量就销毁了，还没复制转换完就出错了，所以在传参的时候要显示的转换或者说构造一个符合的值；
3. 传引用的情况需要特殊处理，因为线程只对实际传输类型敏感，传统的方式将会在线程空间内生成一个新的值存储，函数的引用指向了这个，而不是外层函数的，所以需要显示的指明引用；std::thread t(function\_name,arg1,std::ref(arg2));或者是提供成员函数与类对象地址的方式；std::thread t(&member\_function\_address,&class\_instance,arg1,arg2);
4. 对于传递指针的情况，可以通过std::unique\_ptr智能指针转移对象的控制权，这个指针可以做参数也可以做返回值，但是在做参数时，如果生成的是对象，直接赋值就可以，如果是2个unique\_ptr不能直接赋值，需要使用std::move()转移对象所有权；

Void function(unique\_ptr<type>);

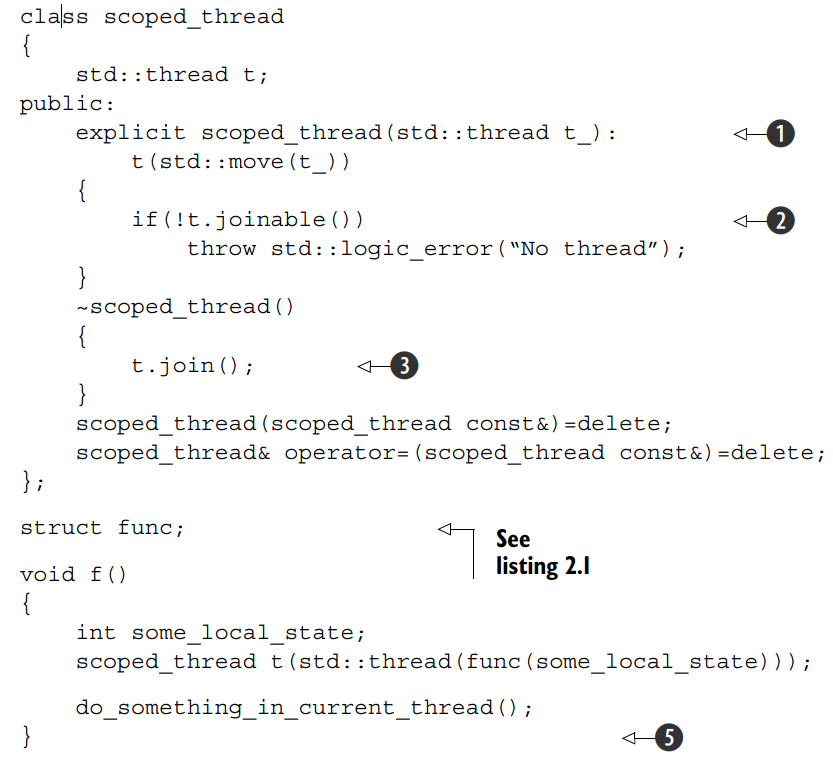
Unique\_ptr<type> p(new type);thread t(function,std::move(p));

## 2.3 移交一个线程的所有权

线程与C++中fstream等拥有资源的对象类似，是不能复制的，只能在定义的对象间移动，thread在构造好之后也只能在thread对象间移动，这可以用与权限转移，使用std::move()函数完成转移。

Notice：

1. 对于生成的未命名的临时thread对象，可以直接赋值给命名的thread对象，这是自动隐士完成的，命名thread对象间则须通过std::move()函数完成，如t1=thread(function()),t2=std::move(t1)，还需要注意的是Thread t，即t未关联任何线程，2个已经关联的命名线程不能进行转移操作，与unique\_ptr类似；
2. 移动的用处是建立守护线程的类，在类中管理线程，并且这样容易在类的析构函数中确保线程的执行处理，当然这是在异常可能发生而影响线程执行的情况下，同时权限转移到守护类中，外界无法操作线程结合还是分离，如：



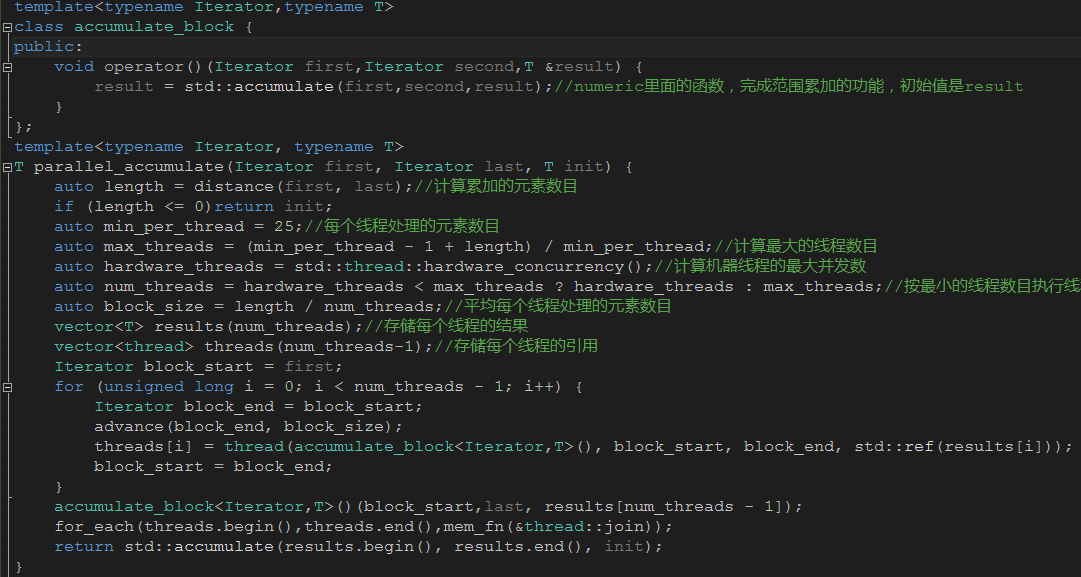
代码2-1 守护类代码

## 2.4 选择运行线程的数量

选择运行线程数量的一个原则就是按照CPU硬件的支持原则，thread::handware\_currency()函数返回CPU真实支持的并发数量，多余这个数量会造成线程切换时间，少于这个浪费时间。

Notice：

1. 清单2.8的的程序，for\_each()函数对迭代器内的数据进行逐个便利操作，并执行相应地函数，mem\_fn()为容器内的函数对象，调去S TL容器内对象的内置函数，进行操作；



代码2-2 累加函数

## 2.5 如何标识线程

线程是使用线程ID标示的。这是一个对象，可以通过thread.get\_id()获取命名线程的标识ID或者this\_thread::get\_id()获取当前执行的线程的标识ID

Notice：

1. 当线程对象与执行的线程相关联时，返回的是该线程的标识ID，当没有关联时，返回没有线程的标识；
2. 线程ID是通用的线程标识，基本能够代表线程，如果相等就代表是同一线程；
3. 线程ID这个对象定义已经定义好了各种比较运算符，可以直接进行各种烦比较操作，因而可以用在容器中，可以作为容器的索引、关键字等也可以；
4. 线程ID的重要作用就是判断是哪个线程来决定执行的操作。

# 3线程间共享数据

本章讲述共享数据，这是多线程的一大优势，因为多线程是可以直接干脆的共享程序的数据，但是这也带来一些并发的问题，需要多线程互相协调好共享数据的机制。

## 3.1 线程间共享数据的难点

共享数据协调很难是因为需要修改数据，全部读取数据不存在这样的问题，多线程修改更新共同的数据造成了一种竞争环境，对数据的破坏很严重。

### 3.1.1 竞争环境

竞争环境特指一种情况就是程序的结果取决于线程执行的相对顺序，大多数的竞争环境都是良性的，不好的竞争环境就是他们改变了共同需要的数据部分，一般的竞争环境就是指这种不好的竞争环境。

### 3.1.2 避免有问题的竞争环境发生

C++提供了几种机制避免这种情况的发生：

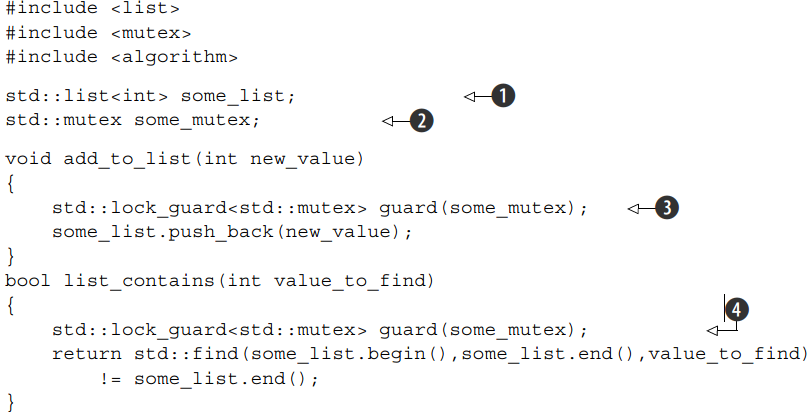
1. 将修改共同数据结构的线程操作原子化，中间状态不可访问；
2. 修改数据结构的设计
3. 事务控制，基本同1一样。

## 3.2 使用互斥量保护共享数据

C++中常用的报保护共享数据的手段就是互斥体(mutex,mutual exclusion)的使用,C++标准库保证当一个线程锁定共享数据时，其他要锁定的线程只能等待解锁后才能执行，这样做有可能会带来死锁的问题。

### 3.2.1 C++中使用互斥体

互斥体简单来说就是一个共享数据是否在使用中的标志，使用mutex定义互斥体变量，为了不在每个部分使用lock与unlock操作，使用lock\_guard对象，他在构造时锁住数据，在退出时，析构执行解锁数据，共享数据与互斥体都是共同出现的，可以封装在一个类中作为private数据，所以互斥体变量是由所有操作共享数据的函数共享的。



代码3-1

Notice：

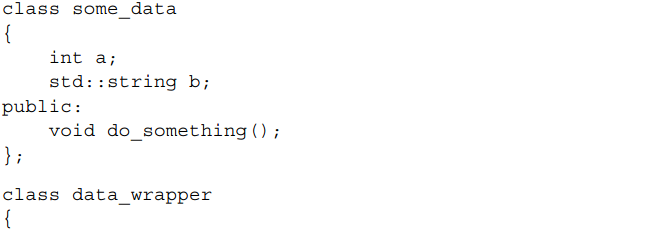
1. 函数或者成员函数不能返回共享数据的指针，因为通过指针操作数据就不需要通过锁了，所以锁只是一个共享数据是否在使用的标识；
2. 每一个共享数据都与唯一一个互斥体对象相关联。

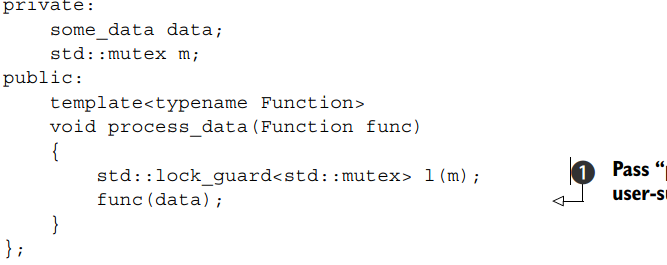
### 3.2.2 为了保护共享数据的代码组织方式

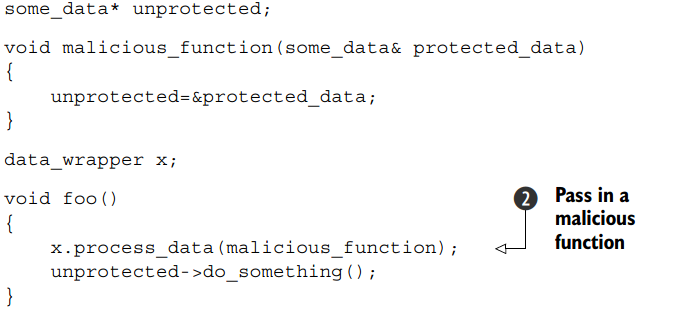
Lock\_guard保护了共享数据，但是对于共享数据的野指针或这野引用会避过互斥体，修改数据；避免这种情况就是避免函数向外传输传输指针或者引用，避免向内部函数中传输指针或者引用，指针与应用只能作用于锁的作用域内。

Notice：

1. 不能向调用的上层函数返回指针与引用；



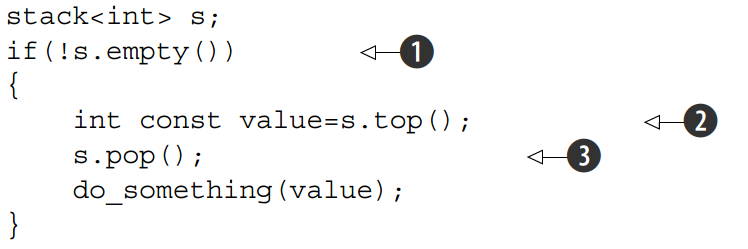


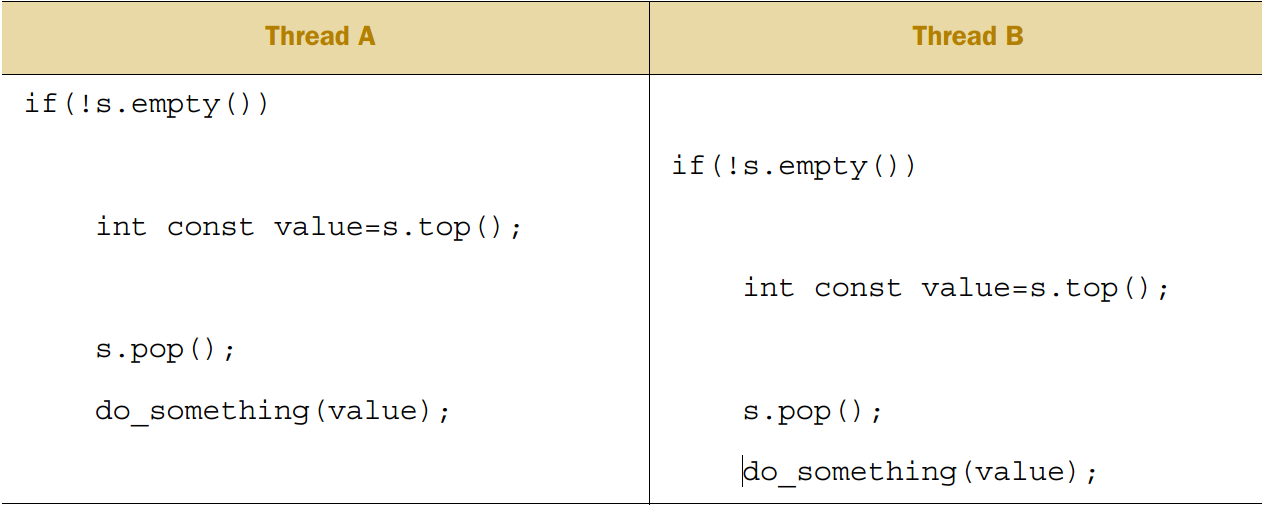


代码2-2

### 3.2.3 找到隐藏在接口中的竞争条件

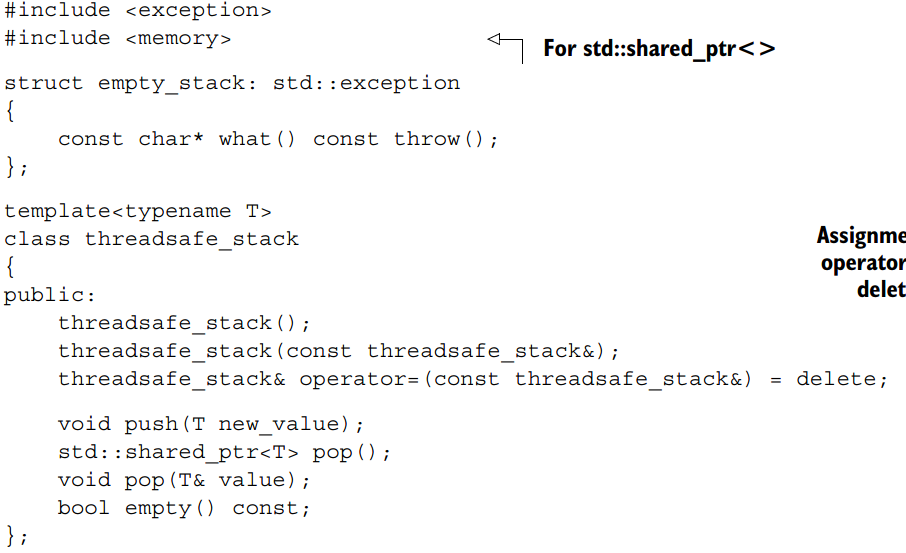
使用互斥体或者其他机制有时并不能完全的保护共享数据，特殊的例子就是接口，我们可以理解为基于一定存储的容器，比如stack。Stack内部可能是有vector或者deque等数据结构构成的，这些数据结构的内部有着防止共享数据操作的互斥体机制，但是在stack级别的并发上，共享数据并没有得到保护，其内部操作函数的并发执行可能毁坏了共享数据的保护机制；在下图中的多线程执行中，s.pop()函数可能发生在另外一个线程的s。empty()函数之后，之后，top与pop将会出现错误，因为可能最后一个数据已经被抛出了。

要从上层类的设计来解决问题，最直接的改变这种的方法就是抛出异常，但是这样影响编程的健壮性；在某些情况下，抛出异常并不能解决并发性的问题。比如如下：



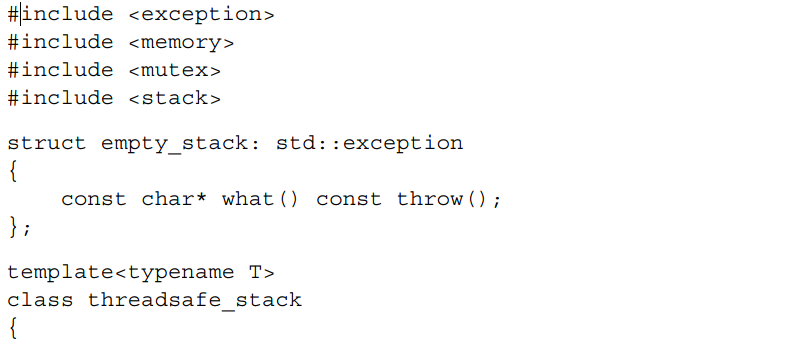
Stack中的某项数据被读取了2次，处理了2次，结果一样，而另外一个数据被直接丢弃了，在多于2个元素的stack中，这并不会抛出异常。一种有效的解决办法就是对整个操作都置于互斥体内，需要注意的一点就是这些操作可能会抛出异常；对于一个stack<vector<int>>数据结构来说，加入pop操作是先出栈，在构造1，如果没有多余的空间的话，构造函数抛出bad-alloc，但是栈中也不会存在元素，所以通常的做法是先top，然后在pop。解决这种问题有几种办法，如下：

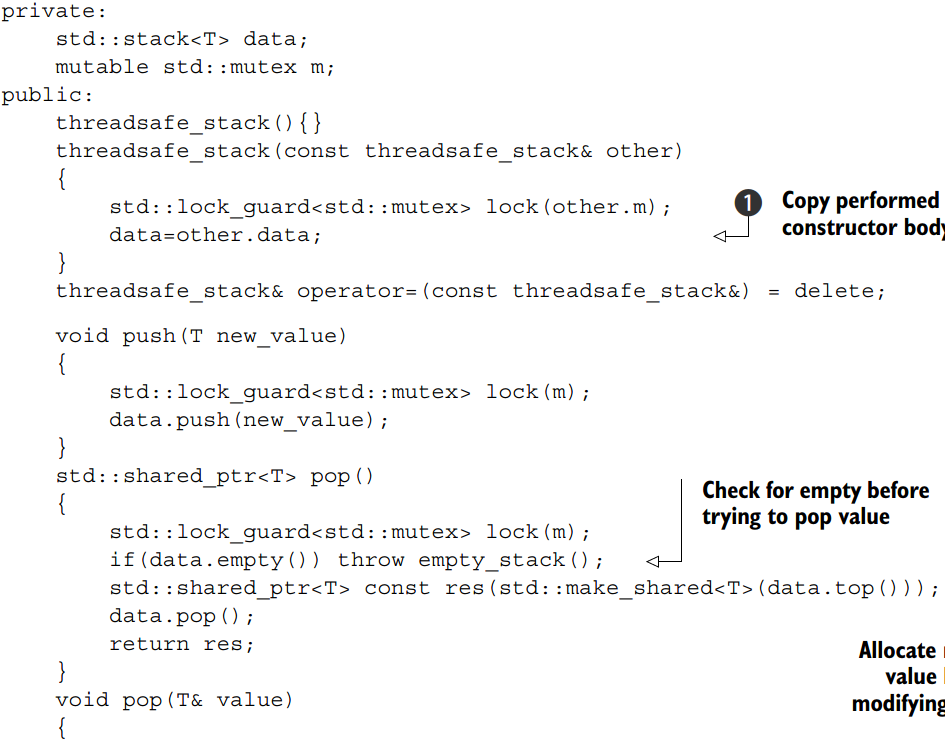
1. 向pop函数中传入引用参数，这是很常见的列子，缺点就是需要事先构造一个值对象，并且传参需要这个对象能够支持赋值操作运算符；
2. 需要一个不会抛出异常的拷贝构造函数与移动构造函数，在C++中大多数的类型的拷贝构造函数与移动构造函数都是不会抛出异常的，但是这种情况不适用于自定义类型，因为自定义类型总会抛出异常，这时需要使用std::is\_nothrow\_copy\_constructible与std::is\_nothrow\_move\_constructible这2个函数来判断；
3. 抛出一个对于出栈项的指针，出栈数据返回指针的优势就是不会抛出异常，劣势就是需要认真的进行内存管理，C++使用std::shared\_ptr()智能指针管理；

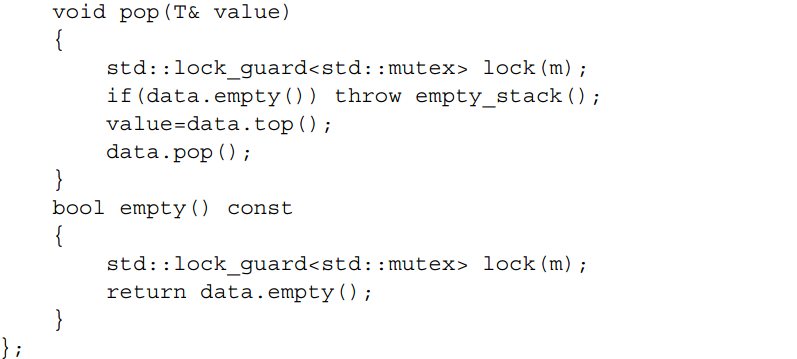


代码3-X 线程安全的类

完整的线程安全类如下代码，通过修改栈的详细操作达到不破坏底层数据结构互斥体的目的，可以确保安全的多线程操作。





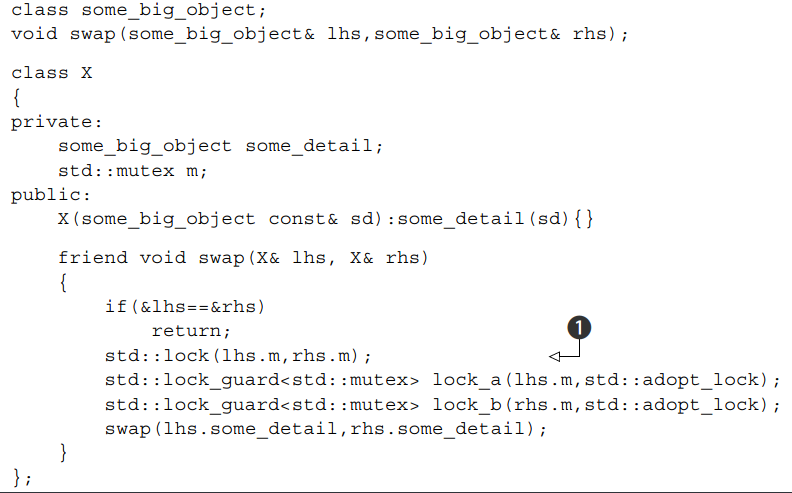


上面的代码中显示，线程安全的栈是客拷贝的，对于接口程序来说，太小的互斥体操作会引起竞争环境，造成并发的难题，但是太大的互斥体范围有影响程序的并发执行，需要保持适度的细粒度划分。

### 3.2.4 死锁：难题与解决方案

举例2个孩子敲鼓，分为鼓与鼓槌，2个人分别有一个，那么没办法敲鼓；这种现象在线程中就是死锁（线程互相请求对方锁住的资源，导致无法运行下去）；一种简单的解决办法就是规定所有并发线程按照规定的顺序访问锁，这种方法粗暴有效，但是也要注意，假如需要交换2个对象的数据，需要同时锁住这2个对象，不能顺序锁，否则会造成死锁，简而言之，死锁的每一个步骤都是原子性的；

C++11标准库的std::lock函数改进了这点，可以锁住多个锁，避免死锁；如下图代码：



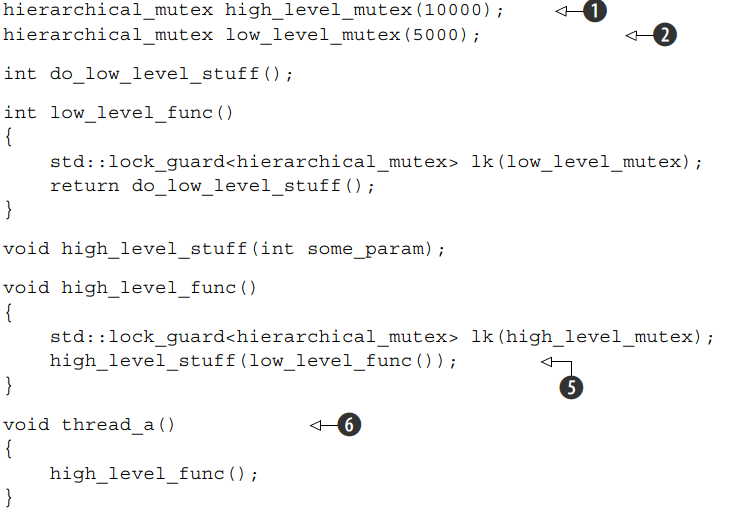
实质上上就是传入锁的参数，进行所操作；记住不能对同一个锁进行2次加锁操作，所以函数内部需要判断是否是同一个对象；参数std::adopt\_lock告诉构造函数锁已经被锁了，不需要构造函数重复在锁而造成的二次加锁。

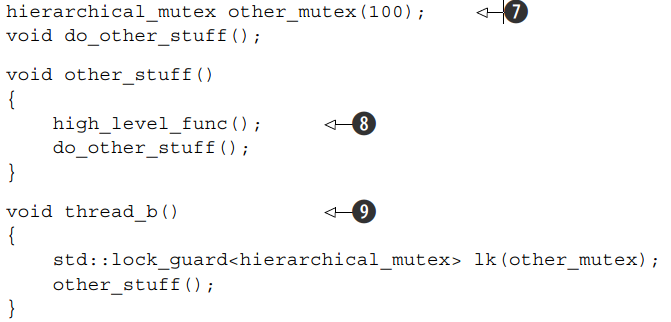
### 3.2.5 避免死锁的高级指导原则

死锁并不只是伴随lock而生的，比如2个线程中都调用了对方的join()函数，那么这2个函数就会锁住都在等待对方，这个现象可以推广到多个线程中去，一个简单的原则就是加入其中一个在等待另一个，那么另一个就不能等待另一个，这个与上述的顺序原则本质上都是一样的，就是规定锁或者等待的顺序避免死锁；

具体的指导原则如下：

1. 避免重复锁，就是已经对互斥量加锁的情况下，不应该重复进行锁的操作，实质上还是顺序原则，就是释放了再加锁，但是这样线程之间还是可能锁住，如果要加多次锁，可以在一个操作中完成，也是用lock函数完成的。
2. 持有一个锁的时候避免调用用户代码，简而言之，就是加锁后，避免调用用户函数，因为用户函数中不确定会做什么操作，这样会造成死锁；这是第一个指导原则的详细形式；
3. 按规定的顺序访问临界区，关键的问题是如何定义这个顺序，或者说以什么样的顺序加锁，有时候它很简单，有时候有很困难；比如，对链表的访问，不只是考虑当前访问节点的锁，还要考虑锁着前面节点，这个要根据数据结构来分析，假如2个线程按相反方向遍历链表，那么一定会锁住，所以要按照一定的方向遍历链表；假如要删除A与C中的B节点，获得了B的锁，但是正好遍历的线程获得了A锁，那么进入死锁状态；
4. 建立一个加锁的层次体系，就是将应用程序分层，然后记录互斥量所在的层次，加锁后记录调用线程的层次，一个线程加入已经锁了下层的一个互斥量，那么在加锁时是不允许的，代码如下图：





（这里没看懂）

1. 在锁之外应用这些指导原则，死锁野火出现在锁之外，只要是需要线程等待的情况都要执行上述的指导原则，除了上述的lock与lock\_guard可以完成简单的避免死锁操作，为了增加弹性，可以加入std::unique\_lock()函数。

### 3.2.6 使用std::unique\_lock()弹性加锁

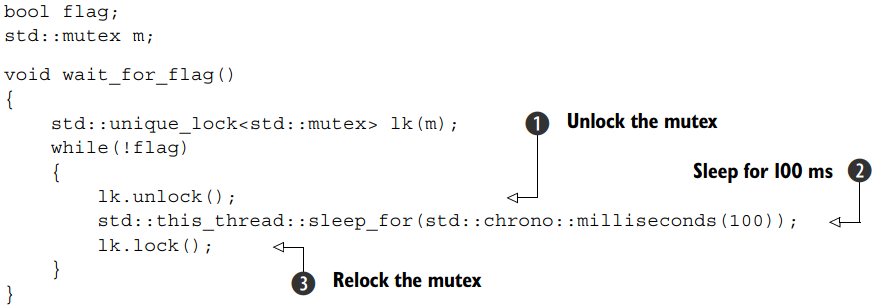
# 同步并发操作

在多线程中保护共享数据具有多种方式，有时还需要等待线程之间操作事件，需要根据C++标准库的future或条件变量同步多线程的操作，同步就是等待以便发生的条件得到满足。

## 4.1 等待一个事件完成或者条件为真

假设你在通宵列车上，到指定地点下车，如何完美的准确在正确地点下车是一个难题，列车晚点以及闹钟失效会早点醒来或者错过站点；假设线程A需要等待线程B完成，设置一个flag，被B锁住，A循环访问，造成了A的等待时间变长，flag此时只能被B使用等；因为等待会占用资源，而占用资源不会释放反而造成了线程更长时间的等待；（mutex是C++11中的互斥量，这实际上就是一个对临界区访问的标志，构造函数，std::mutex不允许拷贝构造，也不允许 move 拷贝，最初产生的 mutex 对象是处于 unlocked 状态的。lock()，调用线程将锁住该互斥量。线程调用该函数会发生下面 3 种情况：(1). 如果该互斥量当前没有被锁住，则调用线程将该互斥量锁住，直到调用 unlock之前，该线程一直拥有该锁。(2). 如果当前互斥量被其他线程锁住，则当前的调用线程被阻塞住。(3). 如果当前互斥量被当前调用线程锁住，则会产生死锁(deadlock)。unlock()， 解锁，释放对互斥量的所有权。try\_lock()，尝试锁住互斥量，如果互斥量被其他线程占有，则当前线程也不会被阻塞。线程调用该函数也会出现下面 3 种情况，(1). 如果当前互斥量没有被其他线程占有，则该线程锁住互斥量，直到该线程调用 unlock 释放互斥量。(2). 如果当前互斥量被其他线程锁住，则当前调用线程返回 false，而并不会被阻塞掉。(3). 如果当前互斥量被当前调用线程锁住，则会产生死锁(deadlock)。）

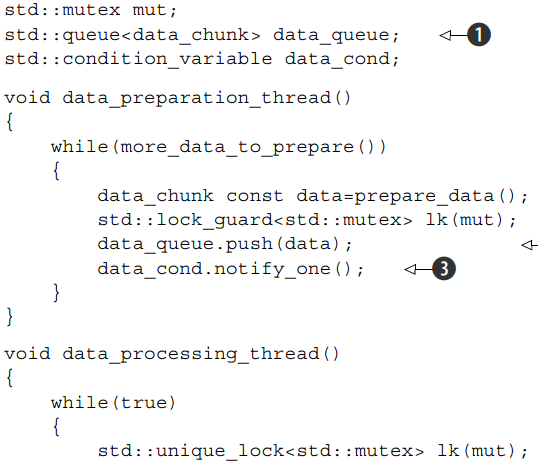
还有一种选择的方案是先睡眠一段时间在来检测标志事件状态，可以通过std::this\_thread::sleep\_for()函数让等待的线程睡眠；

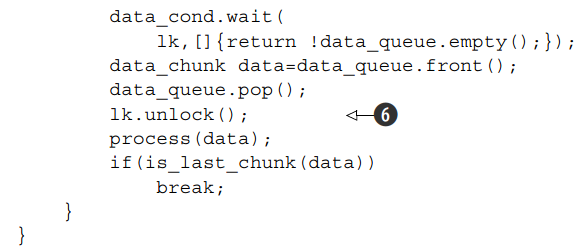


循环中的解锁确保另一个线程可以改变标志量的状态。但是睡眠时间的指定是苦困难的，太短几乎还会浪费资源，太长有可能错过状态改变，虽然这几乎对程序不会有太大影响；最推荐的C++11标准推荐的等待方案，最基本机制是，当一些线程等待某个事件时，操作这个事件的线程一旦确定这个事件或者条件变量已经满足等待线程的条件，就会通知或者触发这些等待线程执行，达成了一种异步。

### 4.1.1 等待条件变量（condition variable）值的真假

C++提供了std::condition\_variable与std::condition\_variable\_any 2个条件变量，在condition\_variable头文件中；他们都需要跟mutex合作提供线程之间的同步，\_any更普遍，与任何类似于mutex的对象都可以合作，所以会有性能资源的额外开销，假如不需要额外的灵活性，前面的就已经最好了；

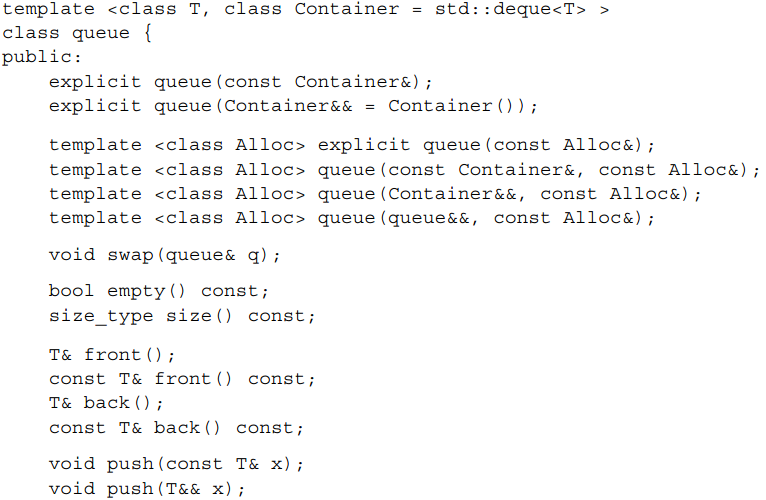


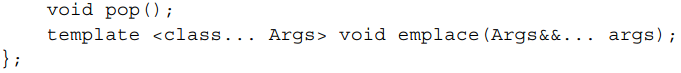


建立一个队列，第一个函数负责向队列中写入数据，操作队列需要加锁，使用lock\_guard()函数，操作完了调用条件变量的通知函数，通知等待的线程已经加入数据了（通知-等待表示具体的事件以及完成了，不能从锁这里来理解条件变量，因为它跟队列与锁没有关系，2个是分离的东西），处理线程首先锁队列，但是用的是unique\_lock函数，这是因为这个对象有解锁函数，在wait()函数中，加入lambda表达式返回的是false，那么就会解锁（lock\_guard没有解锁函数），并阻塞，加入数据函数可以获取锁加入数据，然后再通知，此时，处理函数再次加锁并计算lambda表达式值；wait()函数中的判断值可是是任何返回bool值得表达式；此时wait()函数的阻塞状态还是获取锁的阻塞状态，加入在此时又获取了锁而不是别的线程的通知把它从睡眠中唤醒，这种庄涛叫做假死。

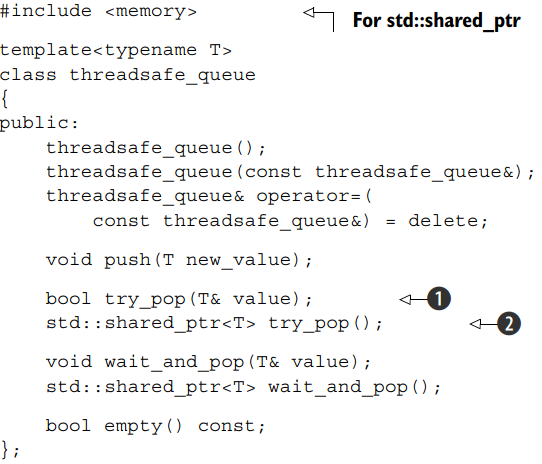
### 4.1.2 使用条件变量建立线程安全的队列

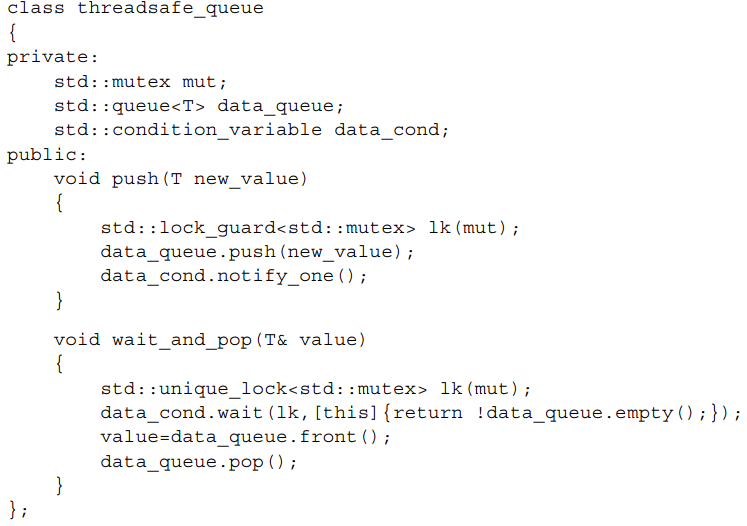
线程安全的队列，接口如下：

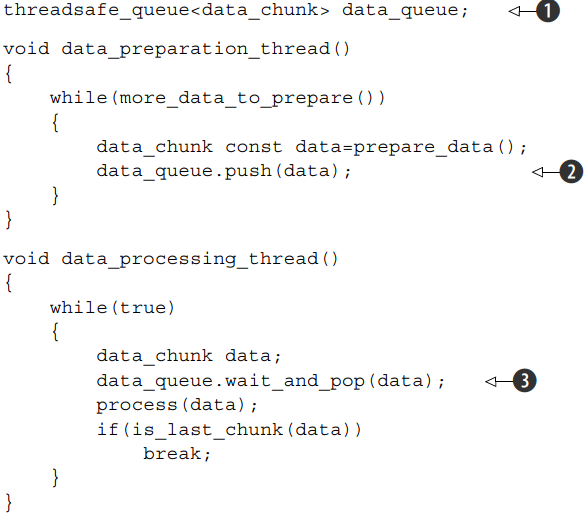




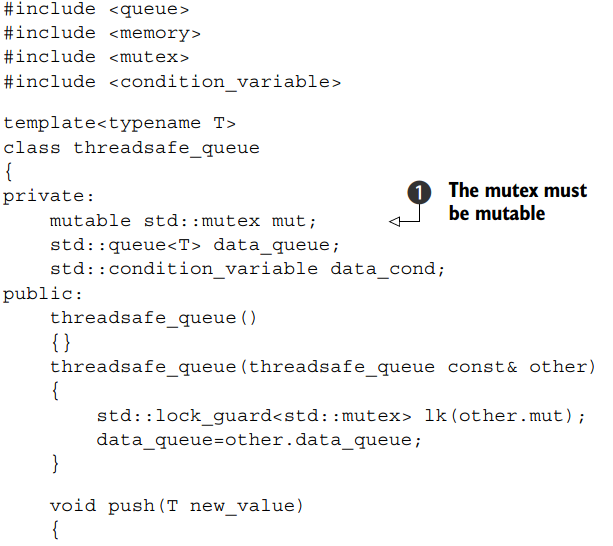
忽略拷贝、赋值函数等，分为3类，查询队列状态的empty，size等；查询元素状态的，back，front；改变元素的；根据上面的清单建立下面的代买：

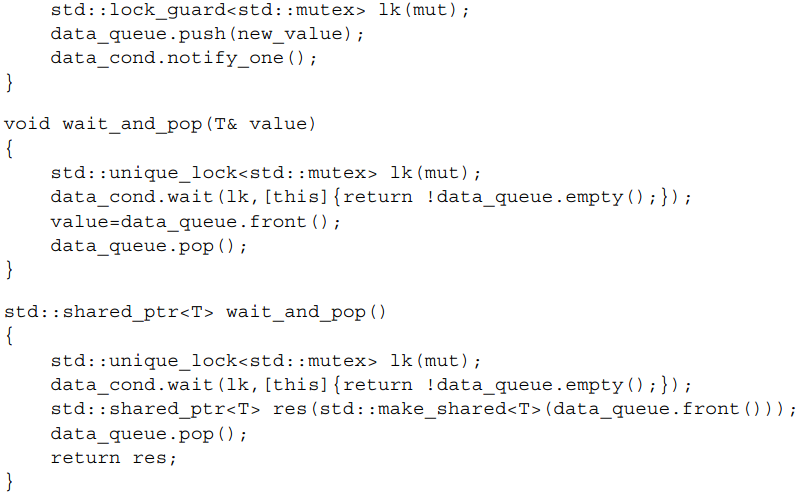


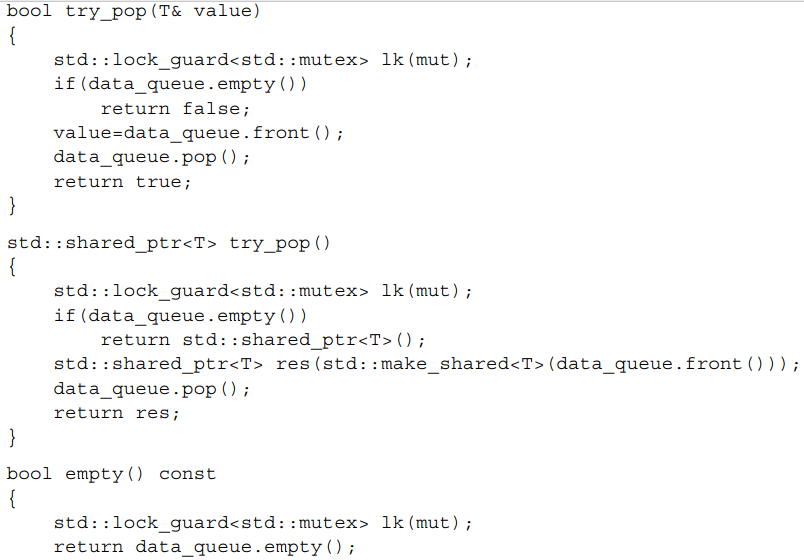




将mutex与条件变量加入到队列里，简化了外部操作；完整的函数定义如下：









虽然empty已经声明了const，但是其他函数可能会持有底层队列，进一步改变队列的状态，所以empty也要加锁，但是要声明const，mutex要加入mutable声明可改变。假如有多个线程等待条件变量的通知，通知是随机的，不知道哪个等待的线程得到数据进行处理；如果是共享数据的形式，可以调用条件变量上的notice\_all()函数，那么所有的等待线程都会检查条件；在只等待一次的条件变量的线程中，条件变量不合适，future更合适。

## 4.2 使用futrue处理等待一次性事件

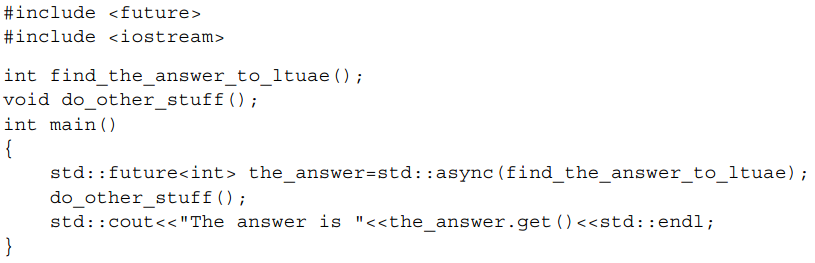
例子就是候车或者候机，告示牌就是future，加入future后，等待事件的线程周期性的检查future的状态，没检查的时候做别的事，事件完成后，线程得到执行，future状态变为ready，并且不能复位，future也是可以附带数据的，就好像告示牌上会有航班、时间、登机口等神马的信息。

C++标准库规定了2种类型的future，unique\_future与shared\_future，与unique\_ptr以及shared\_ptr类似，unique的表明实例只能管理当前这一个对象，不能随便赋值给别的，shared\_future表明实例可以被多个变量关联；future是模板的，所以可以关联多种数据，但是future不提供同步机制，需要自己处理，通常的做法是用shared\_future，每个线程管理自己的，即使关联的是同一个实例。

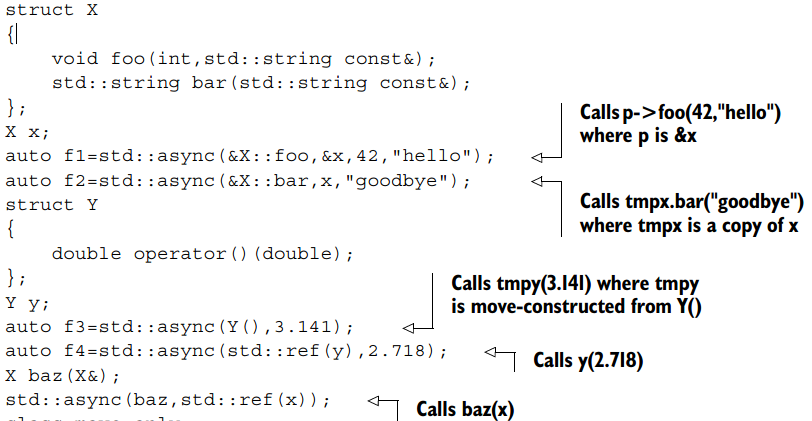
最简单的一次性事件是等待后台计算的结果，但是第二章的说后台计算无法返回结果，可以通过future来解决这一个问题。

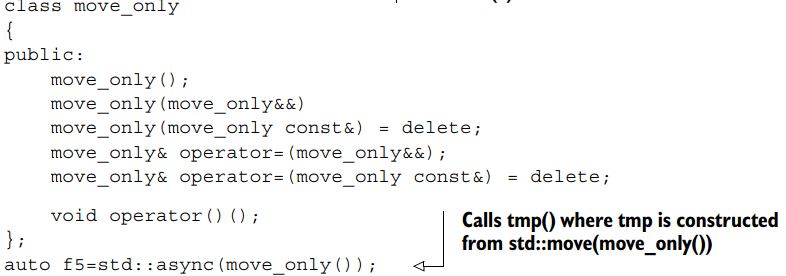
### 4.2.1 从后台任务重返回结果

加入后台计算会得到一个有用的结果，thread并没有机制得到这个数据，所以需要std::async函数声明在头文件future内；使用std::async开启一个异步的后台计算任务，创建后得到的不是一个thread对象，而是最终返回一个含有最后结果的future对象，调用get()函数时，当前线程阻塞，指导future为ready状态，就会得到结果；

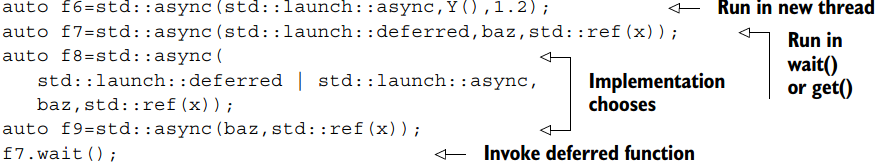


Std::async()函数开启异步线程，这个函数完成了线程的创建与执行操作，只需要等待返回数据即可；std::future:具有3个状态：deferred：异步操作还没开始，ready，异步操作完成，timeout：异步操作超时；跟线程的创建一样，async也可以传参数；第一个是参数是函数，如果是成员函数，第二个参数是成员函数所属的对象，使用地址与std::ref可以保证操作的是源对象；剩下的基本是函数参数，对于右值，他们向线程复制由移动完成。





可以使用额外的参数指定std::async()的运行方式，std::launch与std::launch::deferred指定线程推迟到future调用了get()或者wait后执行；std::launch::async指定函数必须在自己的线程上运行，如果函数执行被推迟，有可能永远也不会得到执行；



当让async并不是唯一的与future关联的方式，还有很多其他的好的方式。

### 4.2.2 将一个任务与一个future关联起来

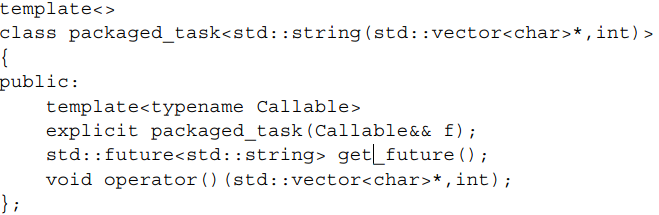
Std::packaged\_task<>可以将future与一个函数或者函数对象绑定，Std::packaged\_task<>调用时首先调用函数执行，获得结果后就会对future的ready置位，所以她可以作为线程池的基本组件，因为线程池只需要管理简单的Std::packaged\_task<>就可以了，不需要管理函数神马的。

Std::packaged\_task<>模板的参数为函数签名，比如std::packaged\_task<int(int)>函数是返回值int，具有一个int形参的函数签名，因为类型可以隐世转换，所以可以转换的签名都可以。

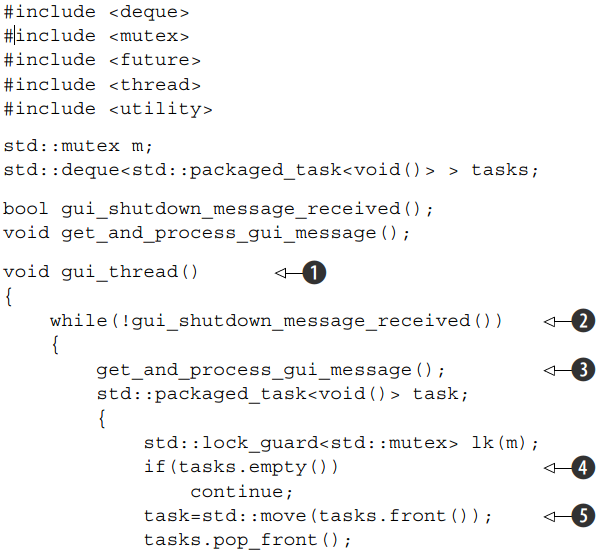
函数签名的返回类型定义了future<>的参数类型，get\_future()返回future，前面的参数定义了函数重载运算符()函数里面的形参，一个

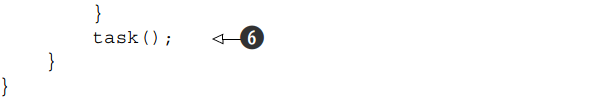
std::packaged\_task<std::string(std::vector<char>\*,int)> 的定义源代码可能是下面这样的：

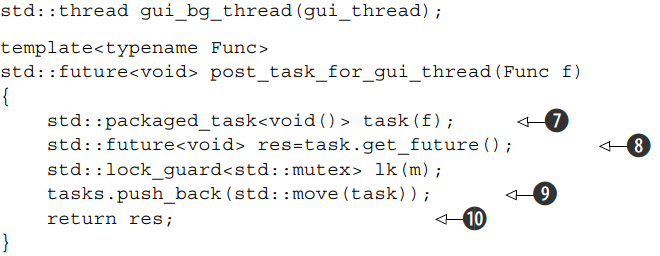
从下图的定义看，Std::packaged\_task<>可以是一个函数对象，里面重载了函数调用运算符，可以作为函数传递给thread执行，或者传递给函数，或者直接像函数那样调用，这时提供给函数对象的参数直接提供给重载函数，并且可以通过get\_future()返回异步操作的结果，可以在Std::packaged\_task<>传递到别的地方之前就获取future对象，等待结果，执行完了出发阻塞状态，获得结果



下面的代码展示了这种情况，是在线程之间传递任务对象；许多更新GUI的活动都要通过特定的线程，需要向这些线程发送消息，每个活动都会发送自己的消息，Std::packaged\_task<>可以提供了统一的消息机制发送消息，如下：





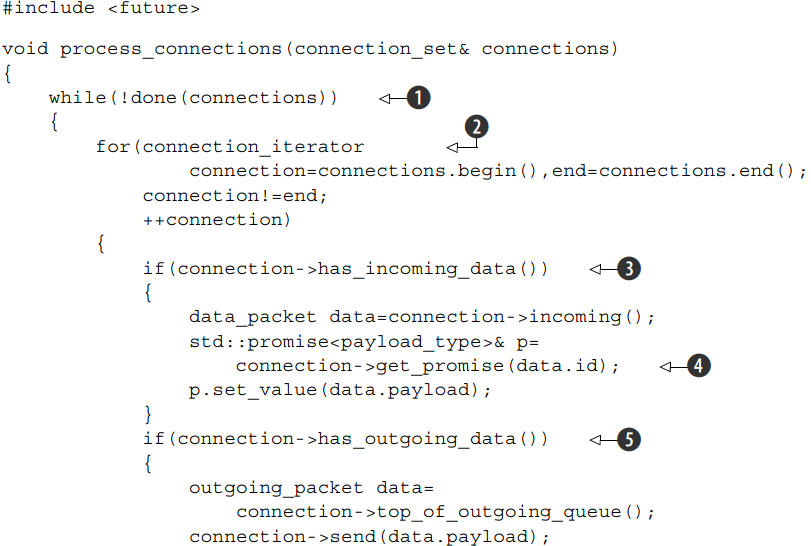


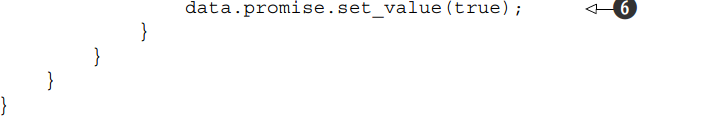
1. 定义的GUI更新线程；
2. 循环知道收到结束信号；
3. 轮训系统的消息或者队列里的消息；
4. 空，在此循环；
5. 取出队列中的任务；
6. 执行；
7. 根据传递的函数创建一个任务；
8. 提前操作获得future；
9. 加入队列；
10. 返回future对象，供上面的调用代码处理future；

这个例子比较简单，因为任务的函数基本没有什么参数，复杂一点的需要加入参数；加入任务不能用简单的函数表达或者future的数据来源于不只一个地方，这是就要用到std::promise。

### 4.2.3 使用std::promise

网络应用程序中，一个连接建立一个线程管理，这适合低连接数量的情况，随着数量增多，管理这些线程就会成为问题，还有上下文切换也要耗时，所以需要线程要保障在一定的数量内，这就需要线程依次可以处理多个连接。

一个线程来说，许多连接发送或者接收的数据包都是随机的，程序的其他部分则等待这些随机的数据被成功接收或者发送；std::promise<T> 可以设置类型为T的值，后面可以根据相关联的future读取，future这里阻塞的时候，线程可以使用配对的promise填充数据并设置ready状态；promise设置了数值，future就会变味ready状态，没有设置数值的时候promise销毁的话，future会存储一个异常；下面的代码展示了管理连接情况，使用了std::promise<bool>/std::future<bool> 代表数据传输的状态，与future相关的bool值代表了成功与失败的标志；

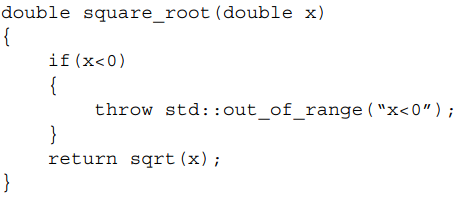


1. 判断是否处理完所有连接；
2. 遍历连接；
3. 5.判断有没有输入与输出；
4. 根据ID取promise；设置负载；
5. 6设置发送成功；

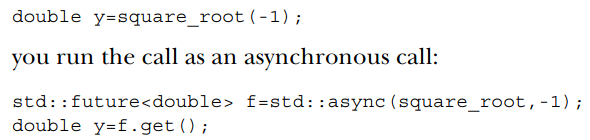
下面介绍异常在线程中存储情况；

### 4.2.4 存储future发生的异常

会抛出异常的函数

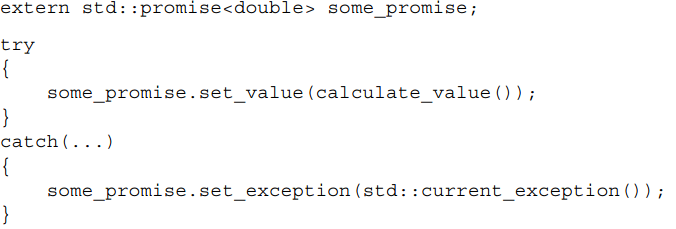


直接调用与异步调用：



不论那种方式，上层的调用函数都能获取函数内部出现了异常，异步的方式的异常存储在future的结果，然后设置future的ready状态，调用get()会重复抛出这个存储的异常，不知道抛出的是本对象还是副本（这个未定义），使用std::packaged\_task()的时候也是这样处理的；

Std::promise也具有这样的功能：



这里使用了std::current\_exception获取异常，还可以使用std::copy\_exception()创建一个新的异常：



还有一种方式就是直接销毁与future相关辆的std::promise或者std::packaged\_task，这样不需要调用promise的设置函数或者加载std::packaged\_task，析构函数就会给future存储一个代号为std::future\_errc::broken\_promise 异常结果，因为future创建的时候就预期会获取一个计算结果或者一个计算异常，直接销毁就未被承诺，所以会返回一个future异常，如果什么都不做，则会永远等待下去。

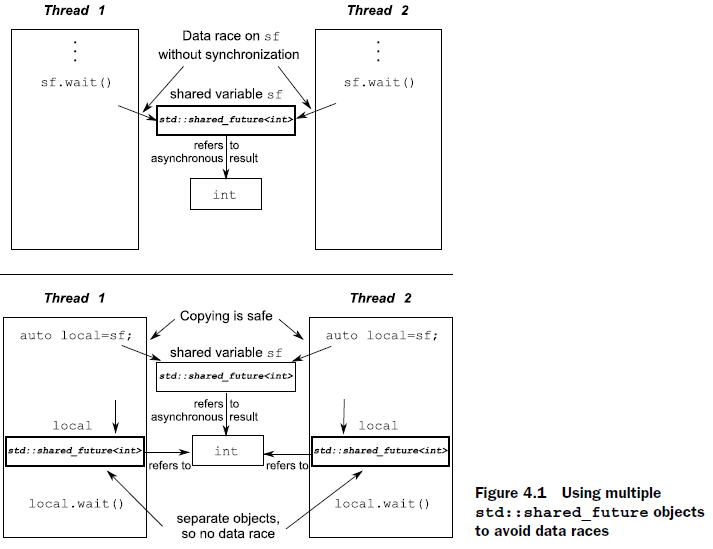
Future会有限制，因为unique\_future只能让一个线程等待结果，所以如果让多个线程都可以得到结果就要使用shared\_future.

### 4.2.5 多个线程等待结果

Std::future处理了所有的线程之间传输数据的操作，但是成员函数的访问不是同步的，因为加入多个线程都要访问future中的结果，那么只会有一个线程获得，其他等待的线程没有结果可以获取，因为他就是这么设计的，future只能有一个所有者，并且get()后，只提供一次值；

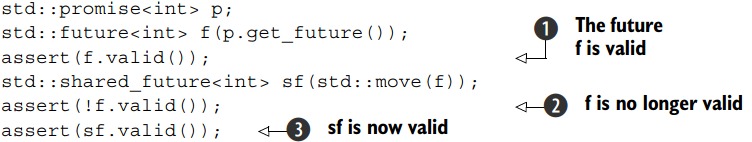
假如多线程都要等待同一个事件，那么可以使用shared\_future，因为future是只能移动的，对象只能被一个实例关联，shared\_future是可复制的，可以多个实例关联对象；但是多线程访问一个共享数据还要涉及到锁来避免数据竞争，所以直接每个线程都有自己的一个副本，那么就实现了真正的并发了。

如下图：



Std::shared\_future一个好的用处就是解决多线程的依赖关系，实现真正的并发执行，比如x,y=x\*2,z=y\*2,那么每一个公式得到值后反给更上层的公式，同时自己还在计算，因为对象是拷贝过去的；

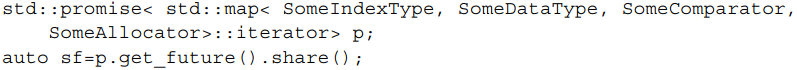
Std::shared\_future可以通过std::unique\_future对象转移所有权构造比如以下：



实际上，与移动对象一样，上面的右值的所有权的转移过程可以用一句代码来完成，中间已经完成了所有转移；



Std::future具有一个好的特性就是可以根据初始化的数值推断数据的类型，他有一个share成员函数，可以返回std::shared\_future对象，并把所有权交给他；



可以指定线程等待的时间限制，时间过了就可以去处理其他事情了。

## 4.3有时间限制的等待

前面说的线程阻塞都是不限定时间的，但是有时需要给等待时间设置一个限制，因为比如上层用户已经放弃等待，网络连接中的另一端持久未连接等，都需要告诉等待线程，不需要等待。

2类可供使用的超时：基于时间段，基于绝对时间，所有的等待函数都提供了处理时间段的变量，处理时间段的变量以\_for结尾，处理绝对时间的变量以\_until结尾。

### 4.3.1 时钟

时钟是信息的来源，是一个类，提供4个信息：现在时间、时间值类型、时间的节拍周期、是否是匀速时钟；当前时间通过now()函数获得，例如：std::chrono::system\_clock::now()返回系统时间，每一个时钟对象是都typedef了一个time\_point成员用来代表返回的时间类型。

节拍使用秒数表示，节拍类型typedef成为一个period，具体来说节拍对象是使用std::ratio<1,25>表示的，就是1s走25个节拍，节拍有时直到运行时才知道、或者运行期件会变化，节拍就不一定是period的规定值，而period值可能是平均节拍周期、最小节拍周期或者是库编写者认为的一个合适的值。

如果一个时钟均匀计时且且不能调整，该时钟为匀速时钟，