4.3 限定等待时间

之前介绍过的所有阻塞调用，将会阻塞一段不确定的时间，将线程挂起直到等待的事件发生。在很多情况下，这样的方式很不错，但是在其他一些情况下，你就需要限制一下线程等待的时间了。这允许你发送一些类似“我还存活”的信息，无论是对交互式用户，或是其他进程，亦或当用户放弃等待，你可以按下“取消”键直接终止等待。

介绍两种可能是你希望指定的超时方式：一种是“时延”的超时方式，另一种是“绝对”超时方式。第一种方式，需要指定一段时间(例如，30毫秒)；第二种方式，就是指定一个时间点(例如，协调世界时[UTC]17:30:15.045987023，2011年11月30日)。多数等待函数提供变量，对两种超时方式进行处理。处理持续时间的变量以“_for”作为后缀，处理绝对时间的变量以"_until"作为后缀。

所以，当 `std::condition_variable` 的两个成员函数wait_for()和wait_until()成员函数分别有两个负载，这两个负载都与wait()成员函数的负载相关——其中一个负载只是等待信号触发，或时间超期，亦或是一个虚假的唤醒，并且醒来时，会检查锁提供的谓词，并且只有在检查为true时才会返回(这时条件变量的条件达成)，或直接而超时。

在我们观察使用超时函数的细节前，让我们来检查一下时间在C++中指定的方式，就从时钟开始吧！

4.3.1 时钟

对于C++标准库来说，时钟就是时间信息源。特别是，时钟是一个类，提供了四种不同的信息：

- 现在时间
- 时间类型
- 时钟节拍
- 通过时钟节拍的分布，判断时钟是否稳定

时钟的当前时间可以通过调用静态成员函数now()从时钟类中获取；例如，`std::chrono::system_clock::now()` 是将返回系统时钟的当前时间。特定的时间点类型可以通过time_point的数据typedef成员来指定，所以some_clock::now()的类型就是some_clock::time_point。

时钟节拍被指定为1/x(x在不同硬件上有不同的值)秒，这是由时间周期所决定——一个时钟一秒有25个节拍，因此一个周期为 `std::ratio<1, 25>` ，当一个时钟的时钟节拍每2.5秒一次，周期就可以表示为 `std::ratio<5, 2>` 。当时钟节拍直到运行时都无法知晓，可以使用一个给定的应用程序运行多次，周期可以用执行的平均时间求出，其中最短的时间可能就是时钟节拍，或者是直接写在手册当中。这就不保证在给定应用中观察到的节拍周期与指定的时钟周期相匹配。

当时钟节拍均匀分布(无论是否与周期匹配)，并且不可调整，这种时钟就称为稳定时钟。当is_steady静态数据成员为true时，表明这个时钟就是稳定的，否则，就是不稳定的。通常情况下， `std::chrono::system_clock` 是不稳定的，因为时钟是可调的，即是这种是完全自动适应本地账户的调节。这种调节可能造成的是，首次调用now()返回的时间要早于上次调用now()所返回的时间，这就违反了节拍频率的均匀分布。稳定闹钟对于超时的计算很重要，所以C++标准库提供一个稳定时钟 `std::chrono::steady_clock` 。C++标准库提供的其他时钟可表示为 `std::chrono::system_clock` (在上面已经提到过)，它代表了系统时钟的“实际时间”，并且提供了函数可将时间点转化为time_t类型的值； `std::chrono::high_resolution_clock` 可能是标准库中提供的具有最小节拍周期(因此具有最高的精度[分辨率])的时钟。它实际上是typedef的另一种时钟，这些时钟和其他与时间相关的工具，都被定义在库头文件中。

我们马上来看一下时间点是如何表示的，但在这之前，我们先看一下持续时间是怎么表示的。

4.3.2 时延

时延是时间部分最简单的； `std::chrono::duration<>` 函数模板能够对时延进行处理(线程库使用到的所有C++时间处理工具，都在 `std::chrono` 命名空间内)。第一个模板参数是一个类型表示(比如，int，long或double)，第二个模板参数是制定部分，表示每一个单元所用秒数。例如，当几分钟的时间要存在short类型中时，可以写成 `std::chrono::duration<short, std::ratio<60, 1>>` ，因为60秒是才是1分钟，所以第二个参数写成 `std::ratio<60, 1>` 。另一方面，当需要将毫秒级计数存在double类型中时，可以写成 `std::chrono::duration<double, std::ratio<1, 1000>>` ，因为1秒等于1000毫秒。

标准库在 `std::chrono` 命名空间内，为延时变量提供一系列预定义类型：nanoseconds[纳秒] , microseconds[微秒] , milliseconds[毫秒] , seconds[秒] , minutes[分]和hours[时]。比如，你要在一个合适的单元表示一段超过500年的时延，预定义类型可充分利用了大整型，来表示所要表示的时间类型。当然，这里也定义了一些国际单位制(SI, [法]le Système international d'unités)分数，可从 `std::atto(10^(-18))` 到 `std::exa(10^(18))` (题外话：当你的平台支持128位整型);也可以指定自定义时延类型，例如， `std::duration<double, std::centi>` ，就可以使用一个double类型的变量表示1/100。

当不要求截断值的情况下(时转换成秒是没问题，但是秒转换成时就不行)时延的转换是隐式的。显示转换可以由 `std::chrono::duration_cast<>` 来完成。

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s=
       std::chrono::duration_cast<std::chrono::seconds>(ms);
```

这里的结果就是截断的，而不是进行了舍入，所以s最后的值将为54。

延迟支持计算，所以你能够对两个时延变量进行加减，或者是对一个时延变量乘除一个常数(模板的第一个参数)来获得一个新延迟变量。例如，5*seconds(1)与seconds(5)或minutes(1)-seconds(55)一样。在时延中可以通过count()成员函数获得单位时间的数量。例如，`std::chrono::milliseconds(1234).count()` 就是1234。

基于时延的等待可由 `std::chrono::duration<>` 来完成。例如，你等待一个“期望”状态变为就绪已经35毫秒：

```
std::future<int> f=std::async(some_task);
if(f.wait_for(std::chrono::milliseconds(35))==std::future_status::ready)
  do_something_with(f.get());
```

等待函数会返回一个状态值，来表示等待是超时，还是继续等待。在这种情况下，你可以等待一个“期望”，所以当函数等待超时时，会返回 `std::future_status::timeout` ； 当“期望”状态改变，函数会返回 `std::future_status::ready` ； 当“期望”的任务延迟了，函数会返回 `std::future_status::deferred` 。基于时延的等待是使用内部库提供的稳定时钟，来进行计时的；所以，即使系统时钟在等待时被调整(向前或向后)，35毫秒的时延在这里意味着，的确耗时35毫秒。当然，难以预料的系统调度和不同操作系统的时钟精度都意味着：在线程中，从调用到返回的实际时间可能要比35毫秒长。

时延中没有特别好的办法来处理以上情况，所以我们暂且停下对时延的讨论。现在，我们就要来看看“时间点”是怎么样工作的。

4.3.3 时间点

时钟的时间点可以用 `std::chrono::time_point<>` 的类型模板实例来表示，实例的第一个参数用来指定所要使用的时钟，第二个函数参数用来表示时间的计量单位(特化的 `std::chrono::duration<>`)。一个时间点的值就是时间的长度(在指定时间的倍数内)，例如，指

定“unix时间戳”(*epoch*)为一个时间点。时间戳是时钟的一个基本属性，但是不可以直接查询，或在C++标准中已经指定。通常，unix时间戳表示1970年1月1日 00:00，即计算机启动应用程序时。时钟可能共享一个时间戳，或具有独立的时间戳。当两个时钟共享一个时间戳时，其中一个time_point类型可以与另一个时钟类型中的time_point相关联。这里，虽然你无法知道unix时间戳是什么，但是你可以通过对指定time_point类型使用time_since_epoch()来获取时间戳。这个成员函数会返回一个时延值，这个时延值是指定时间点到时钟的unix时间戳锁用时。

例如，你可能指定了一个时间点 `std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>` 。这就与系统时钟有关，且实际中的一分钟与系统时钟精度应该不相同(通常差几秒)。

你可以通过 `std::chrono::time_point<>` 实例来加/减时延，来获得一个新的时间点，所以 `std::chrono::hight_resolution_clock::now() + std::chrono::nanoseconds(500)` 将得到500纳秒后的时间。当你知道一块代码的最大时延时，这对于计算绝对时间的超时是一个好消息，当等待时间内，等待函数进行多次调用；或，非等待函数且占用了等待函数时延中的时间。

你也可以减去一个时间点(二者需要共享同一个时钟)。结果是两个时间点的时间差。这对于代码块的计时是很有用的，例如：

```
auto start=std::chrono::high_resolution_clock::now();
do_something();
auto stop=std::chrono::high_resolution_clock::now();
std::cout<<”do_something() took “
  <<std::chrono::duration<double,std::chrono::seconds>(stop-start).count()
  <<” seconds”<<std::endl;
```

`std::chrono::time_point<>` 实例的时钟参数可不仅是能够指定unix时间戳的。当你想一个等待函数(绝对时间超时的方式)传递时间点时，时间点的时钟参数就被用来测量时间。当时钟变更时，会产生严重的后果，因为等待轨迹随着时钟的改变而改变，并且知道调用时钟的now()成员函数时，才能返回一个超过超时时间的值。当时钟向前调整，这就有可能减小等待时间的总长度(与稳定时钟的测量相比)；当时钟向后调整，就有可能增加等待时间的总长度。

如你期望的那样，后缀为_unitl的(等待函数的)变量会使用时间点。通常是使用某些时钟的 `::now()` (程序中一个固定的时间点)作为偏移，虽然时间点与系统时钟有关，可以使用 `std::chrono::system_clock::to_time_point()` 静态成员函数，在用户可视时间点上进行调度操作。例如，当你有一个对多等待500毫秒的，且与条件变量相关的事件，你可以参考如下代码：

清单4.11 等待一个条件变量——有超时功能

```
#include <condition_variable>
#include <mutex>
#include <chrono>

std::condition_variable cv;
bool done;
std::mutex m;

bool wait_loop()
{
  auto const timeout= std::chrono::steady_clock::now()+
      std::chrono::milliseconds(500);
  std::unique_lock<std::mutex> lk(m);
  while(!done)
  {
    if(cv.wait_until(lk,timeout)==std::cv_status::timeout)
      break;
  }
  return done;
}
```

这种方式是我们推荐的，当你没有什么事情可以等待时，可在一定时限中等待条件变量。在这种方式中，循环的整体长度是有限的。如你在4.1.1节中所见，当使用条件变量(且无事可待)时，你就需要使用循环，这是为了处理假唤醒。当你在循环中使用wait_for()时，你可能在等待了足够长的时间后结束等待(在假唤醒之前)，且下一次等待又开始了。这可能重复很多次，使得等待时间无边无际。

到此，有关时间点超时的基本知识你已经了解了。现在，让我们来了解一下如何在函数中使用超时。

4.3.4 具有超时功能的函数

使用超时的最简单方式就是，对一个特定线程添加一个延迟处理；当这个线程无所事事时，就不会占用可供其他线程处理的时间。你在4.1节中看过一个例子，你循环检查“done”标志。两个处理函数分别是 `std::this_thread::sleep_for()` 和 `std::this_thread::sleep_until()` 。他们的工作就像一个简单的闹钟：当线程因为指定时延而进入睡眠时，可使用sleep_for()唤醒；或因指定时间点睡眠的，可使用sleep_until唤醒。sleep_for()的使用如同在4.1节中的例子，有些事必须在指定时间范围内完成，所以耗时在这里就很重要。另一方面，sleep_until()允许在某个特定时间点

将调度线程唤醒。这有可能在晚间备份，或在早上6:00打印工资条时使用，亦或挂起线程直到下一帧刷新时进行视频播放。

当然，休眠只是超时处理的一种形式；你已经看到了，超时可以配合条件变量和“期望”一起使用。超时甚至可以在尝试获取一个互斥锁时(当互斥量支持超时时)使用。 `std::mutex` 和 `std::recursive_mutex` 都不支持超时锁，但是 `std::timed_mutex` 和 `std::recursive_timed_mutex` 支持。这两种类型也有try_lock_for()和try_lock_until()成员函数，可以在一段时期内尝试，或在指定时间点前获取互斥锁。表4.1展示了C++标准库中支持超时的函数。参数列表为“延时”(*duration*)必须是 `std::duration<>` 的实例，并且列出为*时间点*(time_point)必须是 `std::time_point<>` 的实例。

表4.1 可接受超时的函数

类型/命名空间	函数	返回值
std::this_thread[namespace]	sleep_for(duration)	N/A
sleep_until(time_point)		
std::condition_variable 或 std::condition_variable_any	wait_for(lock, duration)	std::cv_status::time_out 或 std::cv_status::no_timeout
wait_until(lock, time_point)		
	wait_for(lock, duration, predicate)	bool —— 当唤醒时，返回谓词的结果
wait_until(lock, duration, predicate)		
std::timed_mutex 或 std::recursive_timed_mutex	try_lock_for(duration)	bool —— 获取锁时返回true，否则返回fasle
try_lock_until(time_point)		
std::unique_lock<TimedLockable>	unique_lock(lockable, duration)	N/A —— 对新构建的对象调用owns_lock();
unique_lock(lockable, time_point)	当获取锁时返回true，否则返回false	
	try_lock_for(duration)	bool —— 当获取锁时返回true，否则返回false

try_lock_until(time_point)		
std::future<ValueType>或 std::shared_future<ValueType>	wait_for(duration)	当等待超时，返回 std::future_status::timeout
wait_until(time_point)	当“期望”准备就绪时，返回 std::future_status::ready	
当“期望”持有一个为启动的延迟函数，返回std::future_status::deferred		

现在，我们讨论的机制有：条件变量、“期望”、“承诺”还有打包的任务。是时候从更高的角度去看待这些机制，怎么样使用这些机制，简化线程的同步操作。