9.2 中断线程

很多情况下，使用信号来终止一个长时间运行的线程是合理的。这种线程的存在，可能是因为工作线程所在的线程池被销毁，或是用户显式的取消了这个任务，亦或其他各种原因。不管是什么原因，原理都一样：需要使用信号来让未结束线程停止运行。这里需要一种合适的方式让线程主动的停下来，而非让线程戛然而止。

你可能会给每种情况制定一个独立的机制，这样做的意义不大。不仅因为用统一的机制会更容易在之后的场景中实现，而且写出来的中断代码不用担心在哪里使用。C++11标准没有提供这样的机制，不过实现这样的机制也并不困难。

在了解一下应该如何实现这种机制前，先来了解一下启动和中断线程的接口。

9.2.1 启动和中断线程

先看一下外部接口，需要从可中断线程上获取些什么？最起码需要和 `std::thread` 相同的接口，还要多加一个interrupt()函数：

```
class interruptible_thread
{
public:
  template<typename FunctionType>
  interruptible_thread(FunctionType f);
  void join();
  void detach();
  bool joinable() const;
  void interrupt();
};
```

类内部可以使用 `std::thread` 来管理线程，并且使用一些自定义数据结构来处理中断。现在，从线程的角度能看到什么呢？“能用这个类来中断线程”——需要一个断点(*interruption point*)。在不添加多余的数据的前提下，为了使断点能够正常使用，就需要使用一个没有参数的函数：interruption_point()。这意味着中断数据结构可以访问thread_local变量，并在线程运行时，对变

量进行设置，因此当线程调用interruption_point()函数时，就会去检查当前运行线程的数据结构。我们将在后面看到interruption_point()的具体实现。

thread_local标志是不能使用普通的 `std::thread` 管理线程的主要原因；需要使用一种方法分配出一个可访问的interruptible_thread实例，就像新启动一个线程一样。在使用已提供函数来做这件事情前，需要将interruptible_thread实例传递给 `std::thread` 的构造函数，创建一个能够执行的线程，就像下面的代码清单所实现。

清单9.9 interruptible_thread的基本实现

```
class interrupt_flag
{
public:
  void set();
  bool is_set() const;
};
thread_local interrupt_flag this_thread_interrupt_flag;  // 1

class interruptible_thread
{
  std::thread internal_thread;
  interrupt_flag* flag;
public:
  template<typename FunctionType>
  interruptible_thread(FunctionType f)
  {
    std::promise<interrupt_flag*> p;  // 2
    internal_thread=std::thread([f,&p]{  // 3
      p.set_value(&this_thread_interrupt_flag);
      f();  // 4
    });
    flag=p.get_future().get();  // 5
  }
  void interrupt()
  {
    if(flag)
    {
      flag->set();  // 6
    }
  }
};
```

提供函数f是包装了一个lambda函数③，线程将会持有f副本和本地promise变量(p)的引用②。在新线程中，lambda函数设置promise变量的值到this_thread_interrupt_flag(在thread_local①中声明)的地址中，为的是让线程能够调用提供函数的副本④。调用线程会等待与其future相关的promise就绪，并且将结果存入到flag成员变量中⑤。注意，即使lambda函数在新线程上执行，对本地变量p进行悬空引用，都没有问题，因为在新线程返回之前，interruptible_thread构造函数会等待变量p，直到变量p不被引用。实现没有考虑处理汇入线程，或分离线程。所以，需要flag变量在线程退出或分离前已经声明，这样就能避免悬空问题。

interrupt()函数相对简单：需要一个线程去做中断时，需要一个合法指针作为一个中断标志，所以可以仅对标志进行设置⑥。

9.2.2 检查线程是否中断

现在就可以设置中断标志了，不过不检查线程是否被中断，这样的意义就不大了。使用interruption_point()函数最简单的情况；可以在一个安全的地方调用这个函数，如果标志已经设置，就可以抛出一个thread_interrupted异常：

```
void interruption_point()
{
  if(this_thread_interrupt_flag.is_set())
  {
    throw thread_interrupted();
  }
}
```

代码中可以在适当的地方使用这个函数：

```
void foo()
{
  while(!done)
  {
    interruption_point();
    process_next_item();
  }
}
```

虽然也能工作，但不理想。最好实在线程等待或阻塞的时候中断线程，因为这时的线程不能运行，也就不能调用interruption_point()函数！在线程等待的时候，什么方式才能去中断线程呢？

9.2.3 中断等待——条件变量

OK，需要仔细选择中断的位置，并通过显式调用interruption_point()进行中断，不过在线程阻塞等待的时候，这种办法就显得苍白无力了，例如：等待条件变量的通知。就需要一个新函数——interruptible_wait()——就可以运行各种需要等待的任务，并且可以知道如何中断等待。之前提到，可能会等待一个条件变量，所以就从它开始：如何做才能中断一个等待的条件变量呢？最简单的方式是，当设置中断标志时，需要提醒条件变量，并在等待后立即设置断点。为了让其工作，需要提醒所有等待对应条件变量的线程，就能确保感谢兴趣的线程能够苏醒。伪苏醒是无论如何都要处理的，所以其他线程(非感兴趣线程)将会被当作伪苏醒处理——两者之间没什么区别。interrupt_flag结构需要存储一个指针指向一个条件变量，所以用set()函数对其进行提醒。为条件变量实现的interruptible_wait()可能会看起来像下面清单中所示。

清单9.10 为 `std::condition_variable` 实现的interruptible_wait有问题版

```
void interruptible_wait(std::condition_variable& cv,
std::unique_lock<std::mutex>& lk)
{
  interruption_point();
  this_thread_interrupt_flag.set_condition_variable(cv);  // 1
  cv.wait(lk);  // 2
  this_thread_interrupt_flag.clear_condition_variable();  // 3
  interruption_point();
}
```

假设函数能够设置和清除相关条件变量上的中断标志，代码会检查中断，通过interrupt_flag为当前线程关联条件变量①，等待条件变量②，清理相关条件变量③，并且再次检查中断。如果线程在等待期间被条件变量所中断，中断线程将广播条件变量，并唤醒等待该条件变量的线程，所以这里就可以检查中断。不幸的是，代码有两个问题。第一个问题比较明显，如果想要线程安全：`std::condition_variable::wait()` 可以抛出异常，所以这里会直接退出，而没有通过条件变量删除相关的中断标志。这个问题很容易修复，就是在析构函数中添加相关删除操作即可。

第二个问题就不大明显了，这段代码存在条件竞争。虽然，线程可以通过调用interruption_point()被中断，不过在调用wait()后，条件变量和相关中断标志就没有什么系了，因为线程不是等待状态，所以不能通过条件变量的方式唤醒。就需要确保线程不会在最后一次中断检查和调用wait()间被唤醒。这里，不对 `std::condition_variable` 的内部结构进行研究；不过，可通过一种方法来解决这个问题：使用lk上的互斥量对线程进行保护，这就需要将lk传递到set_condition_variable()函数中去。不幸的是，这将产生两个新问题：需要传递一个互斥量的引用到一个不知道生命周期的线程中去(这个线程做中断操作)为该线程上锁(调用interrupt()的时候)。这里可能会死锁，并且可能访问到一个已经销毁的互斥量，所以这种方法不可取。当不能完全确定能中断条件变量等待——没有interruptible_wait()情况下也可以时(可能有些严格)，那有没有其他选择呢？一个选择就是

放置超时等待，使用wait_for()并带有一个简单的超时量(比如，1ms)。在线程被中断前，算是给了线程一个等待的上限(以时钟刻度为基准)。如果这样做了，等待线程将会看到更多因为超时而“伪”苏醒的线程，不过超时也不轻易的就帮助到我们。与interrupt_flag相关的实现的一个实现放在下面的清单中展示。

清单9.11 为 `std::condition_variable` 在interruptible_wait中使用超时

```
class interrupt_flag
{
  std::atomic<bool> flag;
  std::condition_variable* thread_cond;
  std::mutex set_clear_mutex;

public:
  interrupt_flag():
    thread_cond(0)
  {}

  void set()
  {
    flag.store(true,std::memory_order_relaxed);
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    if(thread_cond)
    {
      thread_cond->notify_all();
    }
  }

  bool is_set() const
  {
    return flag.load(std::memory_order_relaxed);
  }

  void set_condition_variable(std::condition_variable& cv)
  {
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    thread_cond=&cv;
  }

  void clear_condition_variable()
  {
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    thread_cond=0;
  }
```

```
38
39    struct clear_cv_on_destruct
40    {
41      ~clear_cv_on_destruct()
42      {
43        this_thread_interrupt_flag.clear_condition_variable();
44      }
45    };
46  };
47
48  void interruptible_wait(std::condition_variable& cv,
49    std::unique_lock<std::mutex>& lk)
50  {
51    interruption_point();
52    this_thread_interrupt_flag.set_condition_variable(cv);
53    interrupt_flag::clear_cv_on_destruct guard;
54    interruption_point();
55    cv.wait_for(lk,std::chrono::milliseconds(1));
56    interruption_point();
57  }
```

如果有谓词(相关函数)进行等待，1ms的超时将会完全在谓词循环中完全隐藏：

```
1  template<typename Predicate>
2  void interruptible_wait(std::condition_variable& cv,
3                          std::unique_lock<std::mutex>& lk,
4                          Predicate pred)
5  {
6    interruption_point();
7    this_thread_interrupt_flag.set_condition_variable(cv);
8    interrupt_flag::clear_cv_on_destruct guard;
9    while(!this_thread_interrupt_flag.is_set() && !pred())
10   {
11     cv.wait_for(lk,std::chrono::milliseconds(1));
12   }
13   interruption_point();
14 }
```

这会让谓词被检查的次数增加许多，不过对于简单调用wait()这套实现还是很好用的。超时变量很容易实现：通过制定时间，比如：1ms或更短。OK，对于 `std::condition_variable` 的等待，就需要小心应对了； `std::condition_variable_any` 呢？还是能做的更好吗？

9.2.4 使用 std::condition_variable_any 中断等待

`std::condition_variable_any` 与 `std::condition_variable` 的不同在于，`std::condition_variable_any` 可以使用任意类型的锁，而不仅有 `std::unique_lock<std::mutex>` 。可以让事情做起来更加简单，并且 `std::condition_variable_any` 可以比 `std::condition_variable` 做的更好。因为能与任意类型的锁一起工作，就可以设计自己的锁，上锁/解锁interrupt_flag的内部互斥量set_clear_mutex，并且锁也支持等待调用，就像下面的代码。

清单9.12 为 `std::condition_variable_any` 设计的interruptible_wait

```
class interrupt_flag
{
  std::atomic<bool> flag;
  std::condition_variable* thread_cond;
  std::condition_variable_any* thread_cond_any;
  std::mutex set_clear_mutex;

public:
  interrupt_flag():
    thread_cond(0),thread_cond_any(0)
  {}

  void set()
  {
    flag.store(true,std::memory_order_relaxed);
    std::lock_guard<std::mutex> lk(set_clear_mutex);
    if(thread_cond)
    {
      thread_cond->notify_all();
    }
    else if(thread_cond_any)
    {
      thread_cond_any->notify_all();
    }
  }

  template<typename Lockable>
  void wait(std::condition_variable_any& cv,Lockable& lk)
  {
    struct custom_lock
    {
```

```
      interrupt_flag* self;
      Lockable& lk;

      custom_lock(interrupt_flag* self_,
                  std::condition_variable_any& cond,
                  Lockable& lk_):
        self(self_),lk(lk_)
      {
        self->set_clear_mutex.lock();  // 1
        self->thread_cond_any=&cond;  // 2
      }

      void unlock()  // 3
      {
        lk.unlock();
        self->set_clear_mutex.unlock();
      }

      void lock()
      {
        std::lock(self->set_clear_mutex,lk);  // 4
      }

      ~custom_lock()
      {
        self->thread_cond_any=0;  // 5
        self->set_clear_mutex.unlock();
      }
    };
    custom_lock cl(this,cv,lk);
    interruption_point();
    cv.wait(cl);
    interruption_point();
  }
  // rest as before
};

template<typename Lockable>
void interruptible_wait(std::condition_variable_any& cv,
                        Lockable& lk)
{
  this_thread_interrupt_flag.wait(cv,lk);
}
```

自定义的锁类型在构造的时候，需要所锁住内部set_clear_mutex①，对thread_cond_any指针进行设置，并引用 `std::condition_variable_any` 传入锁的构造函数中②。Lockable引用将会在之后进行存储，其变量必须被锁住。现在可以安心的检查中断，不用担心竞争了。如果这时中断标志已经设置，那么标志一定是在锁住set_clear_mutex时设置的。当条件变量调用自定义锁的unlock()函数中的wait()时，就会对Lockable对象和set_clear_mutex进行解锁③。这就允许线程可以尝试中断其他线程获取set_clear_mutex锁；以及在内部wait()调用之后，检查thread_cond_any指针。这就是在替换 `std::condition_variable` 后，所拥有的功能(不包括管理)。当wait()结束等待(因为等待，或因为伪苏醒)，因为线程将会调用lock()函数，这里依旧要求锁住内部set_clear_mutex，并且锁住Lockable对象④。现在，在wait()调用时，custom_lock的析构函数中⑤清理thread_cond_any指针(同样会解锁set_clear_mutex)之前，可以再次对中断进行检查。

9.2.5 中断其他阻塞调用

这次轮到中断条件变量的等待了，不过其他阻塞情况，比如：互斥锁，等待future等等，该怎么办呢？通常情况下，可以使用 `std::condition_variable` 的超时选项，因为在实际运行中不可能很快的将条件变量的等待终止(不访问内部互斥量或future的话)。不过，在某些情况下，你知道知道你在等待什么，这样就可以让循环在interruptible_wait()函数中运行。作为一个例子，这里为 `std::future<>` 重载了interruptible_wait()的实现：

```
template<typename T>
void interruptible_wait(std::future<T>& uf)
{
  while(!this_thread_interrupt_flag.is_set())
  {
    if(uf.wait_for(lk,std::chrono::milliseconds(1)==
       std::future_status::ready)
      break;
  }
  interruption_point();
}
```

等待会在中断标志设置好的时候，或future准备就绪的时候停止，不过实现中每次等待future的时间只有1ms。这就意味着，中断请求被确定前，平均等待的时间为0.5ms(这里假设存在一个高精度的时钟)。通常wait_for至少会等待一个时钟周期，所以如果时钟周期为15ms，那么结束等待的时间将会是15ms，而不是1ms。接受与不接受这种情况，都得视情况而定。如果这必要，且时钟支持的话，可以持续削减超时时间。这种方式将会让线程苏醒很多次，来检查标志，并且增加线程切换的开销。

OK，我们已经了解如何使用interruption_point()和interruptible_wait()函数检查中断。

当中断被检查出来了，要如何处理它呢？

9.2.6 处理中断

从中断线程的角度看，中断就是thread_interrupted异常，因此能像处理其他异常那样进行处理。

特别是使用标准catch块对其进行捕获：

```
try
{
  do_something();
}
catch(thread_interrupted&)
{
  handle_interruption();
}
```

捕获中断，进行处理。其他线程再次调用interrupt()时，线程将会再次被中断，这就被称为*断点*(interruption point)。如果线程执行的是一系列独立的任务，就会需要断点；中断一个任务，就意味着这个任务被丢弃，并且该线程就会执行任务列表中的其他任务。

因为thread_interrupted是一个异常，在能够被中断的代码中，之前线程安全的注意事项都是适用的，就是为了确保资源不会泄露，并在数据结构中留下对应的退出状态。通常，让线程中断是可行的，所以只需要让异常传播即可。不过，当异常传入 `std::thread` 的析构函数时，`std::terminate()` 将会调用，并且整个程序将会终止。为了避免这种情况，需要在每个将interruptible_thread变量作为参数传入的函数中放置catch(thread_interrupted)处理块，可以将catch块包装进interrupt_flag的初始化过程中。因为异常将会终止独立进程，就能保证未处理的中断是异常安全的。interruptible_thread构造函数中对线程的初始化，实现如下：

```
internal_thread=std::thread([f,&p]{
        p.set_value(&this_thread_interrupt_flag);

        try
        {
          f();
        }
        catch(thread_interrupted const&)
```

```
        {}
      });
```

下面，我们来看个更加复杂的例子。

9.2.7 应用退出时中断后台任务

试想，在桌面上查找一个应用。这就需要与用户互动，应用的状态需要能在显示器上显示，就能看出应用有什么改变。为了避免影响GUI的响应时间，通常会将处理线程放在后台运行。后台进程需要一直执行，直到应用退出；后台线程会作为应用启动的一部分被启动，并且在应用终止的时候停止运行。通常这样的应用只有在机器关闭时，才会退出，因为应用需要更新应用最新的状态，就需要全时间运行。在某些情况下，当应用被关闭，需要使用有序的方式将后台线程关闭，其中一种方式就是中断。

下面清单中为一个系统实现了简单的线程管理部分。

清单9.13 在后台监视文件系统

```
std::mutex config_mutex;
std::vector<interruptible_thread> background_threads;

void background_thread(int disk_id)
{
  while(true)
  {
    interruption_point();  // 1
    fs_change fsc=get_fs_changes(disk_id);  // 2
    if(fsc.has_changes())
    {
      update_index(fsc);  // 3
    }
  }
}

void start_background_processing()
{
  background_threads.push_back(
    interruptible_thread(background_thread,disk_1));
  background_threads.push_back(
    interruptible_thread(background_thread,disk_2));
```

```
}

int main()
{
  start_background_processing();  // 4
  process_gui_until_exit();  // 5
  std::unique_lock<std::mutex> lk(config_mutex);
  for(unsigned i=0;i<background_threads.size();++i)
  {
    background_threads[i].interrupt();  // 6
  }
  for(unsigned i=0;i<background_threads.size();++i)
  {
    background_threads[i].join(); // 7
  }
}
```

启动时，后台线程就已经启动④。之后，对应线程将会处理GUI⑤。当用户要求进程退出时，后台进程将会被中断⑥，并且主线程会等待每一个后台线程结束后才退出⑦。后台线程运行在一个循环中，并时刻检查磁盘的变化②，对其序号进行更新③。调用interruption_point()函数，可以在循环中对中断进行检查。

为什么中断线程前，对线程进行等待？为什么不中断每个线程，让它们执行下一个任务？答案就是“并发”。线程被中断后，不会马上结束，因为需要对下一个断点进行处理，并且在退出前执行析构函数和代码异常处理部分。因为需要汇聚每个线程，所以就会让中断线程等待，即使线程还在做着有用的工作——中断其他线程。只有当没有工作时(所有线程都被中断)，不需要等待。这就允许中断线程并行的处理自己的中断，并更快的完成中断。

中断机制很容易扩展到更深层次的中断调用，或在特定的代码块中禁用中断，这就当做留给读者的作业吧。