# Item 39: Consider void futures for one-shot event communication

# Item 39:对于一次性事件通信考虑使用无返回futures

有时，一个任务通知另一个异步执行的任务发生了特定的事件很有用，因为第二个任务要等到特定事件发生之后才能继续执行。事件也许是数据已经初始化，也许是计算阶段已经完成，或者检测到重要的传感器值。这种情况，什么是线程间通信的最佳方案?

一个明显的方案就是使用条件变量（condvar）。如果我们将检测条件的任务称为检测任务，对条件作出反应的任务称为反应任务，策略很简单：反应任务等待一个条件变量，检测任务在事件发生时改变条件变量。代码如下：

```
std::condition_variable cv; // condvar for event
std::mutex m; // mutex for use with cv
```

检测任务中的代码不能再简单了：

```
... // detect event
cv.notify_one(); // tell reacting task
```

如果有多个反应任务需要被通知，使用`notify_all()`代替`notify_one()`，但是这里，我们假定只有一个反应任务需要通知。

反应任务对的代码稍微复杂一点，因为在调用`wait`条件变量之前，必须通过`std::unique_lock`对象使用互斥锁`mutex`来同步（lock a mutex是等待条件变量的经典实现。`std::unique_lock`是C++11的易用API），代码如下：

```
... // propare to react
{ // open critical section
  std::unique_lock<std::mutex> lk(m); // lock mutex
  cv.wait(lk); // wati for notify; this isn't correct!
  ... // react to event(m is blocked)
} // close crit. section; unlock m via lk's dtor
... // continue reacting (m now unblocked)
```

这份代码的第一个问题就是有时被称为*code smell*：即使代码正常工作，但是有些事情也不是很正确。这种问题源自于使用互斥锁。互斥锁被用于保护共享数据的访问，但是可能检测任务和反应任务不会同时访问共享数据，比如说，检测任务会初始化一个全局数据结构，然后给反应任务用，如果检测任务在初始化之后不会再访问这个数据，而反应任务在初始化之前不会访问这个数据，就不存在数据竞争，也就没有必要使用互斥锁。但是条件变量必须使用互斥锁，这就留下了令人不适的设计。

即使你忽略了这个问题，还有两个问题需要注意：

- **如果检测任务在反应任务`wait`之前通知条件变量，反应任务会挂起**。为了能使条件变量唤醒另一个任务，任务必须等待在条件变量上。如果检测任务在反应任务`wait`之前就通知了条件变量，反应任务就会丢失这次通知，永远不被唤醒
- **`wait`语句虚假唤醒**。线程API的存在一个事实（不只是C++）即使条件变量没有被通知，也可能被虚假唤醒，这种唤醒被称为*spurious wakeups*。正确的代码通过确认条件变量进行处理，并将其作为唤醒后的第一个操作。C++条件变量的API使得这种问题很容易解决，因为允许lambda（或者其他函数对象）来测试等待条件。因此，可以将反应任务这样写：

```
cv.wait(lk,
       [] { return whether the evet has occurred; });
```

要利用这个能力需要反应任务能够确定其等待的条件为真。但是我们考虑的情况下，它正在等待的条件是检测线程负责识别的事件。反应线程可能无法确定等待的事件是否已发生。这就是为什么需要一个条件变量的原因

在很多情况下，使用条件变量进行任务通信非常合适，但是也有不那么合适的情况。

对于很多开发者来说，他们的下一个诀窍是共享的boolean标志。flag被初始化为false。当检测线程识别到发生的事件，将flag设置为true；

```
std::atomic<bool> flag(false); // shared flag; see Item 40 for std::atomic
... // detect event
flag = true; // tell reacting task
```

就其本身而言，反应线程轮询该flag。当发现flag被设置为true，它就知道等待的事件已经发生了：

```
... // prepare
while(!flag); // wait for event
... // react to event
```

这种方法不存在基于条件变量的缺点。不需要互斥锁，在反应变量设置flag为true之前轮询不会出现问题，并且不会出现虚假唤醒。好，好，好。

不好的一点是反应任务中轮询的开销。在等待flag为设置为true的时候，任务基本被锁住了，但是一直占用cpu。这样，反应线程占用了可能给另一个任务使用的硬件线程，每次启动或者完成的时间片都会产生上下文切换的开销，并且保持CPU核心运行（否则可能会停下来省电）。一个真正blocked的任务不会这样，这也是基于条件变量的优点，因为等待调用中的任务真的blocked。

将条件变量和flag的设计组合起来很常用。一个flag表示是否发生了感兴趣的事件，但是通过互斥锁同步了对该flag的访问。因为互斥锁阻止并发该flag，所以如Item 40所述，不需要将flag设置为`std::atomic`。一个简单的bool类型就可以，检测任务代码如下：

```
std::conditon_variable cv;
std::mutex m;
bool flag(false); // not std::atomic
... // detect event
{
  std::lock_guard<std::mutex> g(m); // lock m via g's ctor
  flag = true; // tell reacting task(part 1)
} // unlock m via g's dtor
cv.notify_one(); // tell reacting task (part 2)
```

反应任务代码如下：

```
... // prepare to react
{
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk, []{ return flag; }); // use lambda to avoid spurious wakeups
  ... // react to event (m is blocked)
}
... // continue reacting (m now unblocked)
```

这份代码解决了我们一直讨论的问题。无论是否反应线程在调用`wait`之前还是之后检测线程对条件变量发出通知都可以工作，即使出现了虚假唤醒也可以工作，而且不需要轮询。但是仍然有些古怪，因为检测任务通过奇怪的方式与反应线程通信。（译者注：下面的话挺绕的，可以参考原文）检测任务通知条件变量告诉反应线程等待的事件可能发生了，反应线程必须通过检查flag来确保事件发生了。检测线程设置flag来告诉反应线程事件确实发生了，但是检测线程首先需要通知条件变量唤醒反应线程来检查flag。这种方案是可以工作的，但是不太优雅。

一个替代方案是让反应任务通过在检测任务设置的future上`wait`来避免使用条件变量，互斥锁和flag。这可能听起来也是个古怪的方案。毕竟，Item 38中说明了future代表了从被调用方（通常是异步的）到调用方的通信的接收端，这里的检测任务和反应任务没有调用-被调用的关系。然而，Item 38中也说说明了通信新到发送端是`std::promise`，接收端是`future`不只能用在调用-被调用场景。这样的通信信道可以被在任何你需要从程序一个地方传递到另一个地方的场景。这里，我们用来在检测任务和反应任务之间传递信息，传递的信息就是感兴趣的事件是否已发生。

方案很简单。检测任务有一个`std::promise`对象（通信信道的写入），反应任务有对应的`std::future`（通信信道的读取）。当反应任务看到事件已经发生，设置`std::promise`对象（写入到通信信道）。同时，反应任务在`std::future`上等待。`wait`会锁住反应任务直到`std::promise`被设置。

现在，`std::promise和futures(std::future and std::shared_future)`都是需要参数类型的模板。参数表明被传递的信息类型。在这里，没有数据被传递，只需要让反应任务知道`future`已经被设置了。我们需要的类型是表明在`std::promise`和`futures`之间没有数据被传递。所以选择`void`。检测任务使用`std::promise<void>`，反应任务使用`std::future<void> or std::shared_future<void>`。当感兴趣的事件发生时，检测任务设置`std::promise<void>`，反应任务在`futures`上等待。即使反应任务不接收任何数据，通信信道可以让反应任务知道检测任务是否设置了`void`数据（通过对`std::promise<void>`调用`set_value`）。

所以，代码如下：

```
std::promise<void> p; // promise for communications channel
```

检测任务代码如下：

```
... // detect event
p.set_value(); // tell reacting task
```

反应任务代码如下：

```
... // prepare to react
p.get_future().wait(); // wait on future corresponding to p
... //react to event
```

像使用flag的方法一样，此设计不需要互斥锁，无论检测任务是否在反应任务等待之前设置`std::promise`都可以工作，并且不受虚假唤醒的影响（只有条件变量才容易受到此影响）。与基于条件变量的方法一样，反应任务真是被blocked，不会一直占用系统资源。是不是很完美？

当然不是，基于`future`的方法没有了上述问题，但是有其他新的问题。比如，Item 38中说明，`std::promise 和 future`之间有共享状态，并且共享状态是动态分配的。因此你应该假定此设计会产生基于堆的分配和释放开销。

也许更重要的是，`std::promise`只能设置一次。`std::promise 与 future`之间的通信是一次性的：不能重复使用。这是与基于条件变量或者flag的明显差异，条件变量可以被重复通知，flag也可以重复清除和设置。

一次通信可能没有你想象中那么大的限制。假定你想创建一个挂起的线程以避免想要使用一个线程执行程序的时候的线程创建的开销。或者你想在线程运行前对其进行设置，包括优先级和core affinity。C++并发API没有提供这种设置能力，但是提供了`native_handle()`获取原始线程的接口（通常获取的是POXIC或者Windows的线程），这些低层次的API使你可以对线程设置优先级和 core affinity。

假设你仅仅想要挂起一次线程（在创建后，运行前），使用`void future`就是一个方案。代码如下：

```
std::promise<void> p;
void react(); // func for reacting task
void detect() // func for detecting task
{
  st::thread t([] // create thread
               {
                 p.get_future().wait(); // suspend t until future is set
                 react();
               });
  ... // here, t is suspended prior to call to react
  p.set_value(); // unsuspend t (and thus call react)
  ... // do additional work
  t.join(); // make t unjoinable(see Item 37)
}
```

因为根据Item 37说明，对于检测任务所有路径`thread`t都要是unjoinable的，所以使用建议的`ThreadRAII`。代码如下：

```
void detect()
{
    ThreadRAII tr(
    std::thread([]
                {
                  p.get_future().wait();
                  react();
                }),
    ThreadRAII::DtorAction::join  // risky ! (see below)
  );
  ... // thread inside tr is suspended here
  p.set_value(); // unsuspend thread inside tr
  ... // do additional work
}
```

这样看起来安全多了。问题在于第一个"..."区域（注释了thread inside tr is suspended here），如果异常发生，`p.set_value()`永远不会调用，这意味着`lambda`中的`wait`永远不会返回，即lambda不会结束，问题就是，因为RAII对象tr再析构函数中join。换句话说，如果在第一个"..."中发生了异常，函数挂起，因为tr的析构不会被调用。

有很多方案解决这个问题，但是我把这个经验留给读者（译者注：[http://scottmeyers.blogspot.com/2013/12/threadraii-thread-suspension-trouble.html](http://scottmeyers.blogspot.com/2013/12/threadraii-thread-suspension-trouble.html) 中这个问题的讨论）。这里，我只想展示如何扩展原始代码（不使用RAII类）使其挂起然后取消挂起，这不仅是个例，是个通用场景。简单概括，关键就是在反应任务的代码中使用`std::shared_future`代替`std::future`。一旦你知道`std::future`的`share`成员函数将共享状态所有权转移到`std::shared_future`中，代码自然就写出来了。唯一需要注意的是，每个反应线程需要处理自己的`std::shared_future`副本，该副本引用共享状态，因此通过`share`获得的`shared_future`要被lambda按值捕获：

```
std::promise<void> p; // as before
void detect() // now for multiple reacting tasks
```

```
{
  auto sf = g.get_future().share(); // sf's type is std::shared_future<void>
  std::vector<std::thread> vt; // container for reacting threads
  for (int i = 0;  i < threadsToRun; ++i)
  {
    vt.emplace_back([sf]{
      sf.wait();
      react();
    }); // wait on local copy of sf; see Item 43 for info on emplace_back
  }
  ... // detect hangs if this "..." code throws !
  p.set_value(); // unsuspend all threads
  ...
  for (auto& t : vt) {
    t.join(); // make all threads unjoinable: see Item2 for info on "auto&"
  }
}
```

这样`future`就可以达到预期效果了，这就是你应该将其应用于一次通信的原因。

## 需要记住的事

- 对于简单的事件通信，条件变量需要一个多余的互斥锁，对检测和反应任务的相对进度有约束，并且需要反应任务来验证事件是否已发生
- 基于flag的设计避免的上一条的问题，但是不是真正的挂起反应任务
- 组合条件变量和flag使用，上面的问题都解决了，但是逻辑不让人愉快
- 使用`std::promise`和`future`的方案，要考虑堆内存的分配和销毁开销，同时有只能使用一次通信的限制