4.4 使用同步操作简化代码

同步工具的使用在本章称为构建块，你可以之关注那些需要同步的操作，而非具体使用的机制。当需要为程序的并发时，这是一种可以帮助你简化你的代码的方式，提供更多的函数化的方法。比起在多个线程间直接共享数据，每个任务拥有自己的数据会应该会更好，并且结果可以对其他线程进行广播，这就需要使用“期望”来完成了。

4.4.1 使用“期望”的函数化编程

术语*函数化编程*(functional programming)引用于一种编程方式，这种方式中的函数结果只依赖于传入函数的参数，并不依赖外部状态。当一个函数与数学概念相关时，当你使用相同的函数调用这个函数两次，这两次的结果会完全相同。 C++ 标准库中很多与数学相关的函数都有这个特性，例如，sin(正弦),cos(余弦)和sqrt(平方根)；当然，还有基本类型间的简单运算，例如，3+3，6*9，或1.3/4.7。一个纯粹的函数不会改变任何外部状态，并且这种特性完全限制了函数的返回值。

很容易想象这是一种什么样的情况，特别是当并行发生时，因为在第三章时我们讨论过，很多问题发生在共享数据上。当共享数据没有被修改，那么就不存在条件竞争，并且没有必要使用互斥量去保护共享数据。这可对编程进行极大的简化，例如Haskell语言[2]，在Haskell中函数默认就是这么的“纯粹”；这种纯粹对的方式，在并发编程系统中越来越受欢迎。因为大多数函数都是纯粹的，那么非纯粹的函数对共享数据的修改就显得更为突出，所以其很容易适应应用的整体结构。

函数化编程的好处，并不限于那些将“纯粹”作为默认方式(范型)的语言。 C++ 是一个多范型的语言，其也可以写出FP类型的程序。在 C++11 中这种方式要比 C++98 简单许多，因为 C++11 支持lambda表达式(详见附录A，A.6节)，还加入了Boost和TR1中的 std::bind ，以及自动可以自行推断类型的自动变量(详见附录A，A.7节)。“期望”作为拼图的最后一块，它使得*函数化编程模式并发化*(FP-style concurrency)在 C++ 中成为可能；一个“期望”对象可以在线程间互相传递，并允许其中一个计算结果依赖于另外一个的结果，而非对共享数据的显式访问。

快速排序 FP模式版

为了描述在*函数化*(PF)并发中使用“期望”，让我们来看看一个简单的实现——快速排序算法。该算法的基本思想很简单：给定一个数据列表，然后选取其中一个数为“中间”值，之后将列表中的其他

数值分成两组——一组比中间值大，另一组比中间值小。之后对小于“中间”值的组进行排序，并返回排序好的列表；再返回“中间”值；再对比“中间”值大的组进行排序，并返回排序的列表。图4.2中展示了10个整数在这种方式下进行排序的过程。

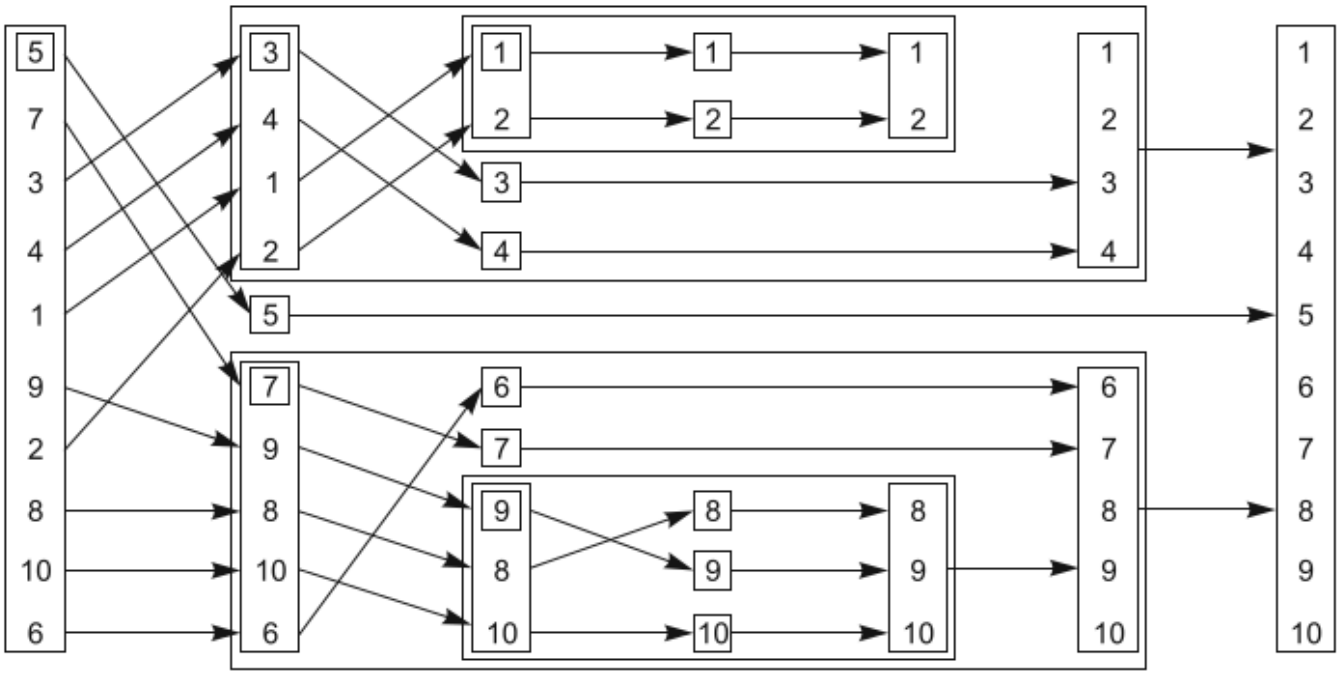

图4.2 FP-模式的递归排序

下面清单中的代码是FP-模式的顺序实现，它需要传入列表，并且返回一个列表，而非与 `std::sort()` 做同样的事情。(译者： `std::sort()` 是无返回值的，因为参数接收的是迭代器，所以其可以对原始列表直进行修改与排序。可参考sort())

清单4.12 快速排序——顺序实现版

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input)
{
  if(input.empty())
  {
    return input;
  }
  std::list<T> result;
  result.splice(result.begin(),input,input.begin());  // 1
  T const& pivot=*result.begin();  // 2

  auto divide_point=std::partition(input.begin(),input.end(),
            [&](T const& t){return t<pivot;});  // 3
```

```
14
15    std::list<T> lower_part;
16    lower_part.splice(lower_part.end(),input,input.begin(),
17                      divide_point);  // 4
18    auto new_lower(
19                      sequential_quick_sort(std::move(lower_part)));  // 5
20    auto new_higher(
21                      sequential_quick_sort(std::move(input)));  // 6
22
23    result.splice(result.end(),new_higher);  // 7
24    result.splice(result.begin(),new_lower);  // 8
25    return result;
26  }
```

虽然接口的形式是FP模式的，但当你使用FP模式时，你需要做大量的拷贝操作，所以在内部你会使用“普通”的命令模式。你选择第一个数为“中间”值，使用splice()①将输入的首个元素(中间值)放入结果列表中。虽然这种方式产生的结果可能不是最优的(会有大量的比较和交换操作)，但是对 `std::list` 做任何事都需要花费较长的时间，因为链表是遍历访问的。你知道你想要什么样的结果，所以你可以直接将要使用的“中间”值提前进行拼接。现在你还需要使用“中间”值进行比较，所以这里使用了一个引用②，为了避免过多的拷贝。之后，你可以使用 `std::partition` 将序列中的值分成小于“中间”值的组和大于“中间”值的组③。最简单的方法就是使用lambda函数指定区分的标准；使用已获取的引用避免对“中间”值的拷贝(详见附录A，A.5节，更多有关lambda函数的信息)。

`std::partition()` 对列表进行重置，并返回一个指向首元素(*不*小于“中间”值)的迭代器。迭代器的类型全称可能会很长，所以你可以使用auto类型说明符，让编译器帮助你定义迭代器类型的变量(详见附录A，A.7节)。

现在，你已经选择了FP模式的接口；所以，当你要使用递归对两部分排序是，你将需要创建两个列表。你可以用splice()函数来做这件事，将input列表小于divided_point的值移动到新列表lower_part④中。其他数继续留在input列表中。而后，你可以使用递归调用⑤⑥的方式，对两个列表进行排序。这里显式使用 `std::move()` 将列表传递到类函数中，这种方式还是为了避免大量的拷贝操作。最终，你可以再次使用splice()，将result中的结果以正确的顺序进行拼接。new_higher指向的值放在“中间”值的后面⑦，new_lower指向的值放在“中间”值的前面⑧。

快速排序 FP模式线程强化版

因为还是使用函数化模式，所以使用“期望”很容易将其转化为并行的版本，如下面的程序清单所示。其中的操作与前面相同，不同的是它们现在并行运行。

清单4.13 快速排序——“期望”并行版

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
  if(input.empty())
  {
    return input;
  }
  std::list<T> result;
  result.splice(result.begin(),input,input.begin());
  T const& pivot=*result.begin();

  auto divide_point=std::partition(input.begin(),input.end(),
                [&](T const& t){return t<pivot;});

  std::list<T> lower_part;
  lower_part.splice(lower_part.end(),input,input.begin(),
                divide_point);

  std::future<std::list<T> > new_lower(  // 1
                std::async(&parallel_quick_sort<T>,std::move(lower_part)));

  auto new_higher(
                parallel_quick_sort(std::move(input)));  // 2

  result.splice(result.end(),new_higher);  // 3
  result.splice(result.begin(),new_lower.get());  // 4
  return result;
}
```

这里最大的变化是，当前线程不对小于“中间”值部分的列表进行排序，使用 `std::async()` ①在另一线程对其进行排序。大于部分列表，如同之前一样，使用递归的方式进行排序②。通过递归调用parallel_quick_sort()，你就可以利用可用的硬件并发了。 `std::async()` 会启动一个新线程，这样当你递归三次时，就会有八个线程在运行了；当你递归十次(对于大约有1000个元素的列表)，如果硬件能处理这十次递归调用，你将会创建1024个执行线程。当运行库认为这样做产生了太多的任务时(也许是因为数量超过了硬件并发的最大值)，运行库可能会同步的切换新产生的任务。当任务过多时(已影响性能)，这些任务应该在使用get()函数获取的线程上运行，而不是在新线程上运行，这样就能避免任务向线程传递的开销。值的注意的是，这完全符合 `std::async` 的实现，为每一个任务启动一个线程(甚至在任务超额时；在 `std::launch::deferred` 没有明确规定的情况下)；或为了同步执行所有任务(在 `std::launch::async` 有明确规定的情况下)。当你依赖运行库的自动缩放，建议你去查看一下你的实现文档，了解一下将会有怎么样的行为表现。

比起使用 `std::async()` ，你可以写一个spawn_task()函数对 `std::packaged_task` 和 `std::thread` 做简单的包装，如清单4.14中的代码所示；你需要为函数结果创建一个 `std::packaged_task` 对象， 可以从这个对象中获取“期望”，或在线程中执行它，返回“期望”。其本身并不提供太多的好处(并且事实上会造成大规模的超额任务)，但是它会为转型成一个更复杂的实现铺平道路，将会实现向一个队列添加任务，而后使用线程池的方式来运行它们。我们将在第9章再讨论线程池。使用 `std::async` 更适合于当你知道你在干什么，并且要完全控制在线程池中构建或执行过任务的线程。

清单4.14 spawn_task的简单实现

```
template<typename F,typename A>
std::future<std::result_of<F(A&&)>::type>
   spawn_task(F&& f,A&& a)
{
  typedef std::result_of<F(A&&)>::type result_type;
  std::packaged_task<result_type(A&&)>
       task(std::move(f)));
  std::future<result_type> res(task.get_future());
  std::thread t(std::move(task),std::move(a));
  t.detach();
  return res;
}
```

其他先不管，回到parallel_quick_sort函数。因为你只是直接递归去获取new_higher列表，你可以如之前一样对new_higher进行拼接③。但是，new_lower列表是 `std::future<std::list<T>>` 的实例，而非是一个简单的列表，所以你需要调用get()成员函数在调用splice()④之前去检索数值。在这之后，等待后台任务完成，并且将结果移入splice()调用中；get()返回一个包含结果的右值引用，所以这个结果是可以移出的(详见附录A，A.1.1节，有更多有关右值引用和移动语义的信息)。

即使假设，使用 `std::async()` 是对可用硬件并发最好的选择，但是这样的并行实现对于快速排序来说，依然不是最理想的。其中， `std::partition` 做了很多工作，即使做了依旧是顺序调用，但就现在的情况来说，已经足够好了。如果你对实现最快并行的可能性感兴趣的话，你可以去查阅一些学术文献。

因为避开了共享易变数据，函数化编程可算作是并发编程的范型；并且也是*通讯顺序进程*(CSP,Communicating Sequential Processer[3],)的范型，这里线程理论上是完全分开的，也就是没有共享数据，但是有通讯通道允许信息在不同线程间进行传递。这种范型被Erlang语言所采纳，并且在MPI(*Message Passing Interface*，消息传递接口)上常用来做C和 `C++` 的高性能运算。现在你应该不会在对学习它们而感到惊奇了吧，因为只需遵守一些约定， `C++` 就能支持它们；在接下来的一节中，我们会讨论实现这种方式。

4.4.2 使用消息传递的同步操作

CSP的概念十分简单：当没有共享数据，每个线程就可以进行独立思考，其行为纯粹基于其所接收到的信息。每个线程就都有一个状态机：当线程收到一条信息，它将会以某种方式更新其状态，并且可能向其他线程发出一条或多条信息，对于消息的处理依赖于线程的初始化状态。这是一种正式写入这些线程的方式，并且以有限状态机的模式实现，但是这不是唯一的方案；状态机可以在应用程序中隐式实现。这种方法咋任何给定的情况下，都更加依赖于特定情形下明确的行为要求和编程团队的专业知识。无论你选择用什么方式去实现每个线程，任务都会分成独立的处理部分，这样会消除潜在的混乱(数据共享并发)，这样就让编程变的更加简单，且拥有低错误率。

真正通讯顺序处理是没有共享数据的，所有消息都是通过消息队列传递，但是因为 C++ 线程共享一块地址空间，所以达不到真正通讯顺序处理的要求。这里就需要有一些约定了：作为一款应用或者是一个库的作者，我们有责任确保在我们的实现中，线程不存在共享数据。当然，为了线程间的通信，消息队列是必须要共享的，具体的细节可以包含在库中。

试想，有一天你要为实现ATM(自动取款机)写一段代码。这段代码需要处理，人们尝试取钱时和银行之间的交互情况，以及控制物理器械接受用户的卡片，显示适当的信息，处理按钮事件，吐出现金，还有退还用户的卡。

一种处理所有事情的方法是让代码将所有事情分配到三个独立线程上去：一个线程去处理物理机械，一个去处理ATM机的逻辑，还有一个用来与银行通讯。这些线程可以通过信息进行纯粹的通讯，而非共享任何数据。比如，当有人在ATM机上插入了卡片或者按下按钮，处理物理机械的线程将会发送一条信息到逻辑线程上，并且逻辑线程将会发送一条消息到机械线程，告诉机械线程可以分配多少钱，等等。

一种为ATM机逻辑建模的方式是将其当做一个状态机。线程的每一个状态都会等待一条可接受的信息，这条信息包含需要处理的内容。这可能会让线程过渡到一个新的状态，并且循环继续。在图4.3中将展示，有状态参与的一个简单是实现。在这个简化实现中，系统在等待一张卡插入。当有卡插入时，系统将会等待用户输入它的PIN(类似身份码的东西)，每次输入一个数字。用户可以将最后输入的数字删除。当数字输入完成，PIN就需要验证。当验证有问题，你的程序就需要终止，就需要为用户退出卡，并且继续等待其他人将卡插入到机器中。当PIN验证通过，你的程序要等待用户取消交易或选择取款。当用户选择取消交易，你的程序就可以结束，并返还卡片。当用户选择取出一定量的现金，你的程序就要在吐出现金和返还卡片前等待银行方面的确认，或显示“余额不足”的信息，并返还卡片。很明显，一个真正的ATM机要考虑的东西更多、更复杂，但是我们来说这样描述已经足够了。

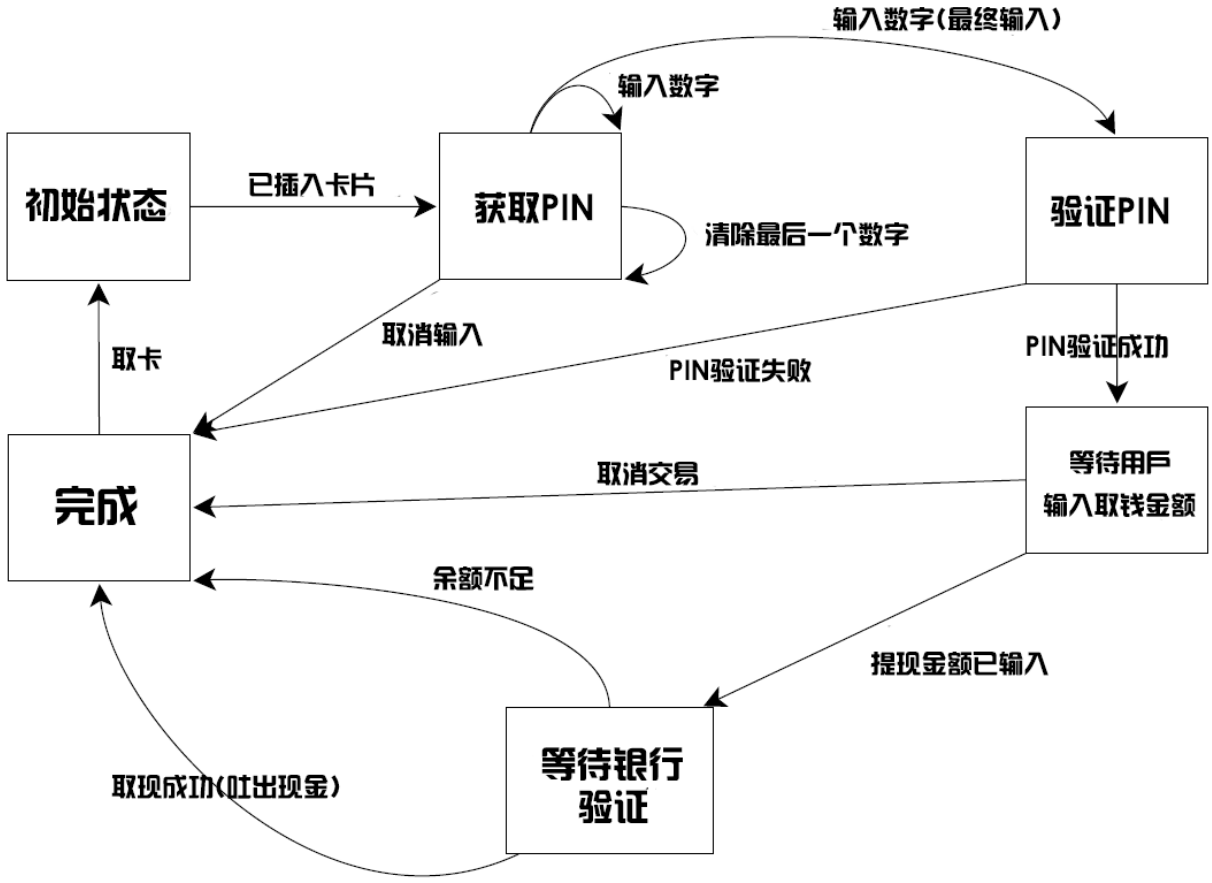

图4.3 一台ATM机的状态机模型(简化)

我们已经为你的ATM机逻辑设计了一个状态机，你可以使用一个类实现它，这个类中有一个成员函数可以代表每一个状态。每一个成员函数可以等待从指定集合中传入的信息，以及当他们到达时进行处理，这就有可能触发原始状态向另一个状态的转化。每种不同的信息类型由一个独立的struct表示。清单4.15展示了ATM逻辑部分的简单实现(在以上描述的系统中，有主循环和对第一状态的实现)，并且一直在等待卡片插入。

如你所见，所有信息传递所需的的同步，完全包含在“信息传递”库中(基本实现在附录C中，是清单4.15代码的完整版)

清单4.15 ATM逻辑类的简单实现

```
struct card_inserted
{
  std::string account;
};

class atm
{
  messaging::receiver incoming;
  messaging::sender bank;
```

```
  messaging::sender interface_hardware;
  void (atm::*state)();

  std::string account;
  std::string pin;

  void waiting_for_card()  // 1
  {
    interface_hardware.send(display_enter_card());  // 2
    incoming.wait().  // 3
      handle<card_inserted>(
      [&](card_inserted const& msg)  // 4
      {
       account=msg.account;
       pin="";
       interface_hardware.send(display_enter_pin());
       state=&atm::getting_pin;
      }
    );
  }
  void getting_pin();
public:
  void run()  // 5
  {
    state=&atm::waiting_for_card;  // 6
    try
    {
      for(;;)
      {
        (this->*state)();  // 7
      }
    }
    catch(messaging::close_queue const&)
    {
    }
  }
};
```

如之前提到的，这个实现对于实际ATM机的逻辑来说是非常简单的，但是他能让你感受到信息传递编程的方式。这里无需考虑同步和并发问题，只需要考虑什么时候接收信息和发送信息即可。为ATM逻辑所设的状态机运行在独立的线程上，与系统的其他部分一起，比如与银行通讯的接口，以及运行在独立线程上的终端接口。这种程序设计的方式被称为*参与者模式*(Actor model)——在系统中有很多独立的(运行在一个独立的线程上)参与者，这些参与者会互相发送信息，去执行手头上的任务，并且它们不会共享状态，除非是通过信息直接传入的。

运行从run()成员函数开始⑤，其将会初始化waiting_for_card⑥的状态，然后反复执行当前状态的成员函数(无论这个状态时怎么样的)⑦。状态函数是简易atm类的成员函数。wait_for_card函数①依旧很简单：它发送一条信息到接口，让终端显示“等待卡片”的信息②，之后就等待传入一条消息进行处理③。这里处理的消息类型只能是card_inserted类的，这里使用一个lambda函数④对其进行处理。当然，你可以传递任何函数或函数对象，去处理函数，但对于一个简单的例子来说，使用lambda表达式是最简单的方式。注意handle()函数调用是连接到wait()函数上的；当收到的信息类型与处理类型不匹配，收到的信息会被丢弃，并且线程继续等待，直到接收到一条类型匹配的消息。

lambda函数自身，只是将用户的账号信息缓存到一个成员变量中去，并且清除PIN信息，再发送一条消息到硬件接口，让显示界面提示用户输入PIN，然后将线程状态改为“获取PIN”。当消息处理程序结束，状态函数就会返回，然后主循环会调用新的状态函数⑦。

如图4.3，getting_pin状态函数会负载一些，因为其要处理三个不同的信息类型。具体代码展示如下：

清单4.16 简单ATM实现中的getting_pin状态函数

```
void atm::getting_pin()
{
  incoming.wait()
    .handle<digit_pressed>(  // 1
      [&](digit_pressed const& msg)
      {
        unsigned const pin_length=4;
        pin+=msg.digit;
        if(pin.length()==pin_length)
        {
          bank.send(verify_pin(account,pin,incoming));
          state=&atm::verifying_pin;
        }
      }
      )
    .handle<clear_last_pressed>(  // 2
      [&](clear_last_pressed const& msg)
      {
        if(!pin.empty())
        {
          pin.resize(pin.length()-1);
        }
      }
      )
    .handle<cancel_pressed>(  // 3
```

```
26        [&](cancel_pressed const& msg)
27        {
28          state=&atm::done_processing;
29        }
30        );
31  }
```

这次需要处理三种消息类型，所以wait()函数后面接了三个handle()函数调用①②③。每个handle()都有对应的消息类型作为模板参数，并且将消息传入一个lambda函数中(其获取消息类型作为一个参数)。因为这里的调用都被连接在了一起，wait()的实现知道它是等待一条digit_pressed消息，或是一条clear_last_pressed肖息，亦或是一条cancel_pressed消息，其他的消息类型将会被丢弃。

这次当你获取一条消息时，无需再去改变状态。比如，当你获取一条digit_pressed消息时，你仅需要将其添加到pin中，除非那些数字是最终的输入。(清单4.15中)主循环⑦将会再次调用getting_pin()去等待下一个数字(或清除数字，或取消交易)。

这里对应的动作如图4.3所示。每个状态盒的实现都由一个不同的成员函数构成，等待相关信息并适当的更新状态。

如你所见，在一个并发系统中这种编程方式可以极大的简化任务的设计，因为每一个线程都完全被独立对待。因此，在使用多线程去分离关注点时，需要你明确如何分配线程之间的任务。

[2] 详见 http://www.haskell.org/.

[3] 《通信顺序进程》(*Communicating Sequential Processes*), C.A.R. Hoare, Prentice Hall, 1985. 免费在线阅读地址 http://www.usingcsp.com/cspbook.pdf.