信息科学系
低年级讨论班

（5）

裘宗燕

北京大学数学学院信息科学系

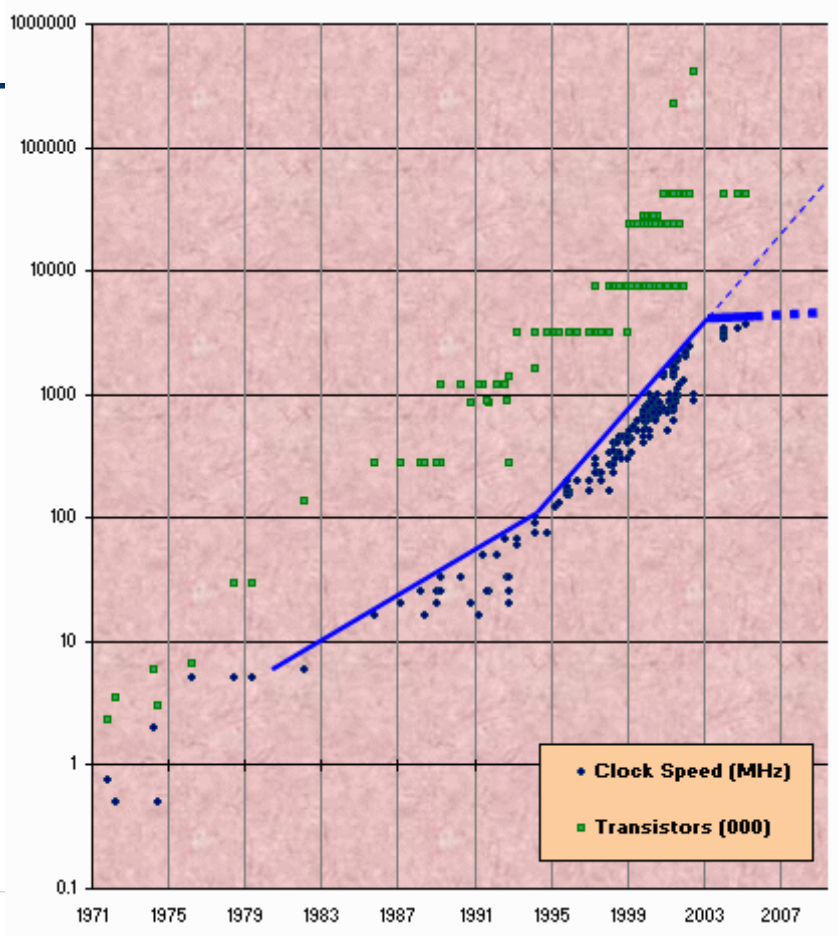
2010年3月～6月

并发之美

《代码之美》第24章

引言

- 靠提高主频提高计算能力的时代已经过去。**Herb Sutter 2005**年的文章：**“The Free Lunch Is Over”**中给出了右边的图
- 过去，我们总可以准备买下一代**CPU**，不用修改程序
- 现在就必须考虑多核和并行处理了
- 再想程序跑得快，能充分利用手头计算机的能力，就必须学会写并行程序



引言

- 并发程序中有多个计算进程同时进行，计算结果可能依赖于不同进程的相对进展速度，并发程序的执行和执行效果有本质的非确定性
- 由于存在非确定性，程序里的**bug**可能不是每次执行都出现，可能在某种执行情况下出错，大多数情况下并不出错
  - 程序里的错误可能很难重现，因此测试非常困难
  - 对并发程序，作者认为漂亮的程序应该是“如此之简单和优雅，以至于明显不可能有错”，而不是“看起来没有明显的错”
- 要想编写出在任何情况下都能可靠运行的并发程序，必须特别关注程序的美感。但是，完成同样工作的并发程序
  - 通常没有与之对应的顺序程序漂亮
  - 特别是比较缺乏模块性（**less modular**）
- 模块性：功能独立性，直接用在不同的环境里，容易与其他部分共处，容易组合、扩充，得到的组合容易修改和调整，等等

引言

- 本章介绍软件事务存储（**software transactional memory，STM**）
  - 这是一项有很好前景的技术
  - 是一种针对共享内存处理器编程的新方法（框架）
  - 共享内存处理器：多个**CPU**共享同一片内存区域。目前的多核和众核处理器都属于共享内存处理器
  - 它一种特有的方式支持模块性，而传统并发编程方式没这种功能
- 本章要通过具体的实例展示 **STM** 的价值
- 当然，任何技术都不是万能的，**STM** 也不是万能仙丹
- 但 **STM** 确实是攻克并发程序这座顽固堡垒的一种漂亮的绝技

简单实例：银行账户 (1)

- 本节用一个简单实例展示并发编程中的困难和问题
- 问题：要求编写一段程序
  - 它把一笔钱从一个银行账户转到另一个银行账户
  - 假定两个账户都在内存
  - 要求代码在并发环境中也能工作（执行环境里可能有其他同时执行的进程，它们可能访问本进程正在使用的内存位置）
  - 可能同时有多个进程来调用这一转帐函数
  - 正确执行：任何进程都不能看到系统处于不一致的状态（例如：看到某个账户已经被取出了一笔钱而另一账户还没得到这笔钱）
- 本例显然是有意编造的，但
  - 它很简单，能帮助看清最关键的问题
  - **Haskell** 与 **STM** 结合，可以给出它的一种全新解决方案

银行账户 (2)：传统方法

先看看传统解决方案

- 目前用于协调并发执行的程序的主要机制是锁和条件变量

  在支持并发的面向对象语言里（如 **Java**），每个对象隐含地带有一个锁，用 **synchronized** 方法获取锁并加锁，其原理和非面向对象语言里的传统并发模型一样

- 在这种 **OO** 语言里，账户类可以定义为：

```
class Account {
  Int balance;
  synchronized void withdraw( Int n ) {  balance = balance - n; }
  void deposit( Int n ) {   withdraw( -n ); }
}
```

  withdrew 是 **synchronized** 方法，这样，即使有两个线程同时来调用它，也只能有一个获得锁并进入对象。只有先进入的线程完成操作退出，其他进程才能进入。这样就不会出错

银行账户 (3)：传统方法

- 有了账户类，就可以定义转帐函数了：

```
void transfer( Account from, Account to, Int amount ) {
    from.withdraw( amount );
    to.deposit( amount );   }
```

  这段程序有问题吗？

- 如果用于顺序执行，这段代码完全没问题
- 但在并发环境里情况就不同了，注意：
  - **transfer** 在第一步锁定 **from** 对象，第二步锁定 **to** 对象
  - 但在两步之间，两个对象都没有锁定
  - 这时如有另一个线程去检查这两个账户，就会看到不一致的状态
- 虽然 **withdrew** 和 **dispose** 函数都是 **synchronized**，也无法避免上述问题，不一致情况可能被观察到

银行账户 (4)：传统方法

- 在并发环境中，如果不一致的情况有可能出现，它就可能存在任意长的时间。在金融环境中不能容许这种情况。怎么办？
- 一种“解决办法”是在操作外面再显式地加上一层锁。例如

```
void transfer( Account from, Account to, Int amount ) {
   from.lock( ); to.lock( );
    from.withdraw( amount );
    to.deposit( amount );
   from.unlock( ); to.unlock( ) }
```

- 很糟糕，作者疏忽了。这一代码是错的（无心之过）
- 改正后的代码仍然有致命错误，下面假设代码已改正
- 代码可能死锁（在某些特定环境情况下），例如：

  假定有两个线程都要执行 transfer 函数，其中一个要从账户 A 转钱给账户 B，另一个要从账户 B 转钱给账户 A。如果两线程的进度恰好是各获得一个锁，两者就都进入无穷等待了（死锁）

银行账户 (5)：传统方法

- 这个问题比较简单，容易识别（并发程序中的问题未必都这么简单）。标准解决方案是给系统里所有的锁统一排顺序编号，每当需要同时获取几个锁的时候，总按照编号递增的顺序获取
- 对上面两个锁，用下面代码：

```
if from < to
  then { from.lock(); to.lock(  ); }
  else { to.lock(); from.lock(  ); }
```

  释放的顺序没关系（想想）。修改后死锁问题就解决了

- 能用上述方案的前提是：事先知道将要获取哪些锁
- 实际情况未必能满足这一条件。设 withdrew 的设计是先考虑从 from 取钱，from 余额不足再由账户 from2 取钱。只有查看了 from 后才知道要不要获取 from2 的锁，而这时再去锁定它，已经不能保证正确顺序了。进一步说，可能 from2 需要由 from 内部获得，transfer 函数事先无法知道。即使知道，由于要处理三个锁，事情也更复杂了

银行账户 (6)：传统方法

- 如果程序运行中还可能要求阻塞，情况就更难处理了
- 例如

  假定 **from** 账户的资金不足时 **transfer** 应该阻塞在那里等待

  通常做法是让它在一个条件变量上等待，并释放当时已获取的所有的锁。如果需要做的是在两账户 **from** 和 **from2** 的余额之和不足的情况下等待，事情就更难处理了

  请自己想想这些问题如何处理

弄坏的锁

总之，目前作为并发编程领域的主导技术的锁和条件变量，有本质性的缺陷。下面列举基于锁的并发编程中的一些公认的难题（有些问题在前面的例子里有所表现）

- 很容易忘记加锁

  导致可能出现多个线程修改同一个变量的情况

- 多做了加锁动作

  损失了可能的并行执行，甚至导致死锁

- 锁加错了

  由于一个锁和被它保护的数据之间的联系可能只存在于程序员的头脑里，在程序里并不明显表示，因此很容易出错（面向对象程序将锁附着在对象上，可以缓解这一问题）

- 加锁的顺序不对

  顺序不对可能导致在某些情况下死锁，这种情况很难通过测试发现

弄坏的锁

- 错误恢复

  由于锁的存在，程序运行中出错的情况变得更难处理。例如，出现错误时是否应该释放锁？如何保证不出现可观察的不一致行为，这些都很难处理（出现错误的情况有时很难掌握）

- 忘记应该做的唤醒操作，或做了不该做的重新检查

  当一个线程释放锁或修改了条件变量的值时，有可能需要去唤醒当时正在等待这个锁或这一条件变量的真值的线程。如果忘记去唤醒，就可能导致死锁或其他错误。原本不需要检查锁或条件变量而去检查，就可能被阻塞甚至陷入死锁

- 这套机制最大的问题是不能很好支持模块化编程。模块化编程要求很容易基于小程序做出大程序，但前面例子里的情况：

  虽然 **withdrew** 和 **deposit** 都能正确工作，但不能基于它们直接去做 **transfer**，必须把锁提出来。如果还可能在账户余额不足时阻塞，更必须把加锁条件抽出来暴露给外部（不容易做好，自己想想）

软件事务内存（STM）

- 软件事务内存是应对并发性挑战的一种很有前景的技术。下面将用 **Haskell** 语言来介绍 **STM**，作者认为 **Haskell** 是最美的编程语言
- **Haskell** 里写出的 **transfer** 函数的开头是：

```
transfer :: Account -> Account -> Int -> IO ( )
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount = ...
```

  第二行是注释

  第一行声明了 **transfer** 函数的类型：有一个 **Account** 参数，另一个 **Account** 参数，一个 **Int** 参数，返回 **IO( )** 类型的值

  结果类型 **IO( )** 说明 **transfer** 将返回一个动作，动作执行时会产生副作用（**side effect**）并返回 **( )** 值。 **( )** 类似 **C** 语言的 **void**

  这样，**transfer** 的结果类型 **IO( )** 说明实现其副作用是调用这个函数的唯一目的（一般函数有返回值，可以使用其返回值）

Haskell 里的副作用

- 现在解释 **Haskell** 的副作用
- 重要概念：**mutable** 和 **immutable**（可变的和不变的）

  例如，**C** 语言里定义为 **const** 的东西是 **immutable** 的；**Java** 里 **Integer**类的对象是 **immutable** 的
- 副作用，就是对可变状态的读写。**IO** 是典型的例子
- 下面是两个有输入/输出副作用的函数的原型

  hPutStr :: Handle -> String -> IO ( )

  hGetLine :: Handle -> IO String

  类型为 **IO t** 的值都是动作（**action**）。**t** 是一个类型，例如是 **( )**
- 例子

  (hPutStr h "hello") 是动作，要求将 **hello** 输出到 **h**（如文件等）

  (hGetLine h) 是动作，要求从 **h** 读入一行作为 **String** 返回

动作组合

- **do** $\{a_1; \ldots; a_n\}$ 把几个小动作 $a_1; \ldots; a_n$ 粘成一个大动作。这里的 **do** 是 **Haskell** 关键字。下面函数读入一行返回，其中还输出一个串：

```
hEchoLine :: Handle -> IO String
hEchoLine h = do { s <- hGetLine h
                 ; hPutStr h ("I just read: " ++ s ++ "\n")
                 ; return s }
```

  这里 **s <-** 表示给读入的串关联一个名字 **s**

  要特别指出，**return** 并不是语言内部的结构，就是一个普通函数

  return :: a -> IO a

  例如，**return v** 执行时返回 **v**，没有副作用

  这里的 **a** 是类型变量，代表任意的类型
- 在 **Haskell** 里写函数作用，只需把函数名和参数逐个列出，不需要括号
- **h** 称为"句柄"，就像 **C** 语言里的 **FILE***，这里也有 **stdin**，**stdout**

读写可变变量

- 输入/输出是一类重要的副作用，另一类副作用是读写可变（mutable）变量。下面函数将一个可变变量的值加一：

```
incRef :: IORef Int -> IO ( )
incRef var = do { val <- readIORef var
                ; writeIORef var (val+1) }
```

incRef var 是一个动作，它先执行 readIORef var 取得 var 的值，将这个值绑定于 val；而后执行 writeIORef 将 val+1 写入 var

- readIORef 和 WriteIORef 的类型是：

```
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ( )
```

类型为 IORef 的值可以看作一个指针或者引用，这种值引着一个类型为 t 的可变值，有点像 C 里的 *t 类型

对于 incRef，其参数的类型必须是 IORef Int，因为 incRef 只能作用于保存 Int 的可变值

顺序和并发执行

- 前面解释了如何从小的动作组合出大的动作，下面谈如何调用动作
- 在Haskell 里，一个程序就是定义了一个称为 main 的动作，运行这个程序就是执行动作 main。下面是一个完整程序：

```
main :: IO ( )
main = do { hPutStr stdout "Hello"
          ; hPutStr stdout " world\n" }
```

这是个顺序程序，里面是顺序组合的若干动作

- 要组合出并发程序，需要用另一个原语 forkIO，其类型是

```
forkIO :: IO a -> IO ThreadId
```

参数是 IO 动作， forkIO创建一个并发的 Haskell 线程去执行该动作

一旦创建了线程，Haskell 的运行系统就会让它与当时存在的其他线程并发执行

并发程序

- 假设把前面的 **main** 改为：

```
main :: IO ( )
main = do { forkIO (hPutStr stdout "Hello")
          ; hPutStr stdout " world\n" }
```

  这里的 **forkIO (hPutStr stdout “Hello”)** 创建一个线程去输出 **Hello**，但 **forkIO** 创建好线程就完成了

  随后是两个输出动作并发执行，一个由执行 **main** 的主线程执行，另一个由新创建的线程执行。究竟哪个先做输出是无法确定的

- **Haskell** 里的线程是“轻量级”线程，一个线程只占用几百字节内存。系统里完全可能同时存在成千的线程
- 到这里我们可能觉得 **Haskell** 太难用而且太麻烦，一个 **n++** 就要写很多话。但请注意，**Haskell** 是函数式语言，大多数程序是用函数式风格写的，其函数式内核功能强大，表达力强，写出的程序简洁清晰。上面写的是有副作用的程序，在 **Haskell** 描述副作用确实麻烦一点

Haskell 的副作用

- **Haskell** 编程的要义是：程序里应尽可能少用副作用
- 应该注意到：在 **Haskell** 里，有副作用的情况都要明确表述出来，这样做揭示出很多有用信息。考虑下面两个函数：

```
f :: Int -> Int

g :: Int -> IO Int
```

  情况一目了然：**f** 是纯函数，没有副作用。在任何时候用某个参数调用它，返回的结果都一样

  而 **g** 有副作用，即使给它同样的参数，两次执行也可能给出不同结果

- 下面会看到，将副作用明确表示出来，在实际中非常有用

Haskell 的值

- 还应指出：在 **Haskell** 里，“动作”（如输入输出动作）也是值

  可以作为参数传递给函数，作为值返回，约束于变量，等等

- 下面是用 **Haskell** 写出的模拟简单 **for** 循环的函数（非内建的）：

```
nTimes :: Int -> IO () -> IO ( )
nTimes 0 do_this = return ( )
nTimes n do_this = do { do_this; nTimes (n-1) do_this }
```

  后两行是函数定义（在 **Haskell** 里可以这样写）

  第一行：如果要求循环 **0** 次，就什么也不做，直接结束

  第二行：一般的做 **n** 次，就是先做一次 **do_this**，而后用 **n-1** 调用本函数。这里的 **do_this** 是一个（任意的）动作

- 重复输出 **20** 个 **Hello**（每次换行）：

```
Main = nTimes 20 (hPutStr stdout "Hello\n")
```


Haskell 里的事务

- 回到 **transfer** 函数，其代码如下：

```
transfer :: Account -> Account -> Int -> IO ( )
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount
  = atomically (do { deposit to amount
                   ; withdraw from amount })
```

  do 块里顺序调用 **dispose** 和 **withdrew**，这两个函数下面写

- **atomically** 以一个动作为参数，把它作为原子动作执行。两个性质
  - 原子性：**atomically act** 做的就是 **act** 动作，但它要求 **act** 执行中的副作用对其他线程是原子的，它们不能看到动作的中间过程，例如一个账户已取钱而另一个还没存入的状态
  - 隔离性：**atomically act** 执行期间，**act** 动作完全不受其他线程的影响，就像 **act** 给开始执行时的世界拍了快照，在这一快照中执行

原子块的实现

- 可能想到很简单的模型来实现 **atomically** 功能：用一个全局锁，每个 **atomically** 先设法获取这个锁，而后执行动作，完成后释放锁
- 这实现非常简单，但完全禁止多个 **atomically** 动作同时执行，互不相干的原子块也不能同时执行。这会大大影响系统的并行性和效率。竞争全局锁也会成为执行的瓶颈
- 上面模型还有第二个问题：不能保证隔离性（因此实际上是错的）

  假设在一个线程执行的原子块 **A**（取得全局锁后）里要访问某 **IORef** 变量，在 **A** 的执行期间其他线程完全可以访问该 **IORef** 变量（只要不是在原子块里访问）。这样就可能影响原子块 **A** 的执行
- 第一个问题在后面讨论“事务内存的实现”时再解决，现在先解决隔离性问题，保证实现的正确性

隔离和类型

- 这里的方案是用类型系统，给 **atomically** 函数下面类型：

  atomically :: STM a -> IO a

  其参数是类型为 **STM a** 的动作
- **STM** 动作类似于 **IO** 动作，可能有副作用，但 **STM** 动作产生副作用的范围比较小，只能用于读写事务变量（**TVar a** 变量）

  回忆：**IO** 动作可以读写任意可变变量
- 对 **TVar** 变量的读写就像在 **IO** 动作里对 **IORef** 变量读写

  ```
  readTVar  :: TVar a -> STM a
  writeTVar :: TVar a -> a -> STM ( )
  ```
- 同样可以用 **do** 组合 **STM** 动作，**return** 也能用于 **STM** 动作

  可以认为 **do** 块和 **return** 对 **IORef** 和 **STM** 重载了（就像 **+** 可以用于整数和浮点数）。实际上 **IORef** 和 **STM** 都是某种更一般结构的特例，那个基础结构称为 **monad**（见书上的脚注）

STM 操作

- 下面是 withdrew 的代码：

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ( )
withdraw acc amount
  = do { bal <- readTVar acc
       ; writeTVar acc (bal - amount) }
```

Account 是保存 Int 的 STM 事务变量，其中的值表示账户余额

withdrew 是一个 STM 动作，它减少账户里的值

- 可以用 withdrew 定义dispose

```
deposit :: Account -> Int -> STM ( )
deposit acc amount = withdraw acc (- amount)
```

这样就有了 transfer 的完整定义

STM 操作

- 注意，transfer 最终执行了四次基本的 STM 变量读写动作：对 to 账户读一次并写一次，对 from 账户也一样

这四个操作位于一个原子块里，可以保证按原子动作的方式执行

这样就解决了开始提出的问题

- Haskell 的类型系统保证我们不能在事务之外读写 TVar。假如写：

```
bad :: Account -> IO ( )
bad acc = do { hPutStr stdout "Withdrawing..."
             ; withdraw acc 10 }
```

代码不能通过编译，因为 hPutStr 是 IO 动作，不是 STM 动作

- 可以写下面的代码（把 withdrew 放进原子块）

```
good :: Account -> IO ( )
good acc = do { hPutStr stdout "Withdrawing..."
              ; atomically (withdraw acc 10) }
```

事务内存的实现

- 人们用事务内存的根本原因就是原子性和隔离性的保证，但有一个合理的实现模型帮助建立直观理解，同样也很有价值
- 下面介绍一种实现模型，但这只是一种可能的方法。**STM** 抽象的最漂亮的地方就在于它提供了一个很小而又非常清晰的接口，允许以各种各样方式来实现，可以非常简单，也可以非常复杂
- 一种很吸引人的实现方式来自数据库领域，称为**最优执行**。梗概：
  - 执行 **atomically act** 时为线程分配一个事务日志，初始为空
  - **act** 执行中每次调用 **writeTVar** 写 **TVar** 变量时，并不真写入有关变量，而是写入日志
  - 每次调用 **readTVar** 读 **Tvar** 变量时先到日志里找（可能该 **TVar** 变量已写过），如找不到就从原 **TVar** 变量读取值，并把这个变量和它的值记入日志文件
  - 一个原子块执行其间完全可能有其他线程也在运行它们的原子块，也在不断地读写自己的 **TVar** 变量版本

事务内存的实现

- 当这种 **act** 动作执行完成时，实现首先去确认其日志。如果确认成功就提交这一日志（也就是说，按照日志修改实际变量）
- 确认的工作是检查日志里记录的每个 **readTVar**，看它们的值是不是和相应的实际 **TVar** 一样。如果都一样则确认成功

  注意：确认和提交步骤的执行不允许打断。**STM** 实现将禁止硬件中断，或用一个锁，或用 **compare-and-swap**（**CAS**，比较并交换）指令，或者别的必要机制，保证从其他线程的角度看，这个线程做确认和提交的动作就像一个原子动作（不能被打断）。这些都由实现处理，程序员不需要关心其实现细节
- 另一方面，如果确认失败，说明线程对于内存没有一个前后一致的观点（视图），这时令本事务中止（**abort**），重新初始化日志并再次执行 **act** 动作。这称为**再次执行**（**re-execution**）

  由于 **act** 执行的效果还没有提交（没有对实际变量做任何修改），可以保证再次执行是安全的

事务内存的实现

- 有一个问题必须注意：事务 **act** 执行中不能出现读写 **TVar** 变量之外的副作用，这一点至关重要。作为例子，考虑下面情况：

```
atomically (do { x <- readTVar xv
               ; y <- readTVar yv
               ; if x>y then launchMissiles
                        else return ( ) })
```

  其中 **launchMissiles::IO( )** 将产生极其严重的国际性的副作用

  由于原子块没加锁，这使执行线程可能出现对内存状态的不统一观点（如果它执行期间有其他线程并发地修改 **xv** 或 **yv**）

  如果出现了不一致的内存观点的情况，这个线程就可能误发射了一颗火箭（**launchMissiles**）。当系统在整个原子块执行完试图去确认时失败时，所犯的错误已经没办法补偿了

- 好在 **Haskell** 类型系统能检查出这类问题并拒绝上面代码：它禁止在一个 **STM** 动作里运行 **IO** 动作。这也说明确实应该区分 **IO/STM**

阻塞和选择

- 像上面讨论的这样的原子块还缺乏一种能力，它们完全不能用于协调并发程序，因为其中缺乏两种关键机制：阻塞和选择
- 下面说明怎样对 **STM** 接口做一些精细加工，使之能提供这两种能力，而且这种扩充是以完全模块化的方式进行的
- 假定有一个线程，它试图从一个账户提款，当这个账户的余额不足时，我们希望这个线程阻塞（等待到账户有了足够余额）
- 在并发程序里经常能看到这种情况。例如：如果线程需要从一个缓冲区读数据，而当时该缓冲区是空的，这一线程就会阻塞在这里等待；另一种常见情况是线程需要等待一个特定事件
- 在 **STM** 里做这件事的方法就是加一个函数 **retry**，其类型是

```
retry :: STM a
```

  retry 的语义很简单：它执行时简单地丢掉当前事务，等待将来某个时候再转过来重新执行

阻塞和选择

- 下面是基于 **retry** 修改过的 **withdrew**

```
limitedWithdraw :: Account -> Int -> STM ( )
limitedWithdraw acc amount
  = do { bal <- readTVar acc
       ; if amount > 0 && amount > bal
         then retry
         else writeTVar acc (bal - amount) }
```

  当提款额大于账户现款时执行 **retry**

- 问题是什么时候再次做?
  - 立刻重试也是正确的。但可能发现账户没变化，结果依旧。不断的反复重试会消耗大量计算资源，不是高效的实现
  - 高效的 **STM** 实现会让线程等待，直到有其他线程写了这个 **acc** 变量（修改这一账户的余额）后再让这个线程重试
  - 怎样确定该线程等待 **acc**？很简单：这一事务是在到 **retry** 的路径上读 **acc**，这件事肯定记录在日志里，查一下就可以确定了

阻塞和选择

- **limitedWithdraw** 里的条件展示了一种特别常用的模式：检查一个逻辑条件，条件成立或不成立时继续或 **retry**。可以抽象为函数

```
check :: Bool -> STM ( )
check True  = return ( )
check False = retry
```

- 利用 **check** 可以把 **limitedWithdraw** 写得更简洁一点：

```
limitedWithdraw :: Account -> Int -> STM ( )
limitedWithdraw acc amount
  = do { bal <- readTVar acc
       ; check (amount <= 0 || amount <= bal)
       ; writeTVar acc (bal - amount) }
```

- 现在研究**选择**，考虑前面提过的另一个问题：

  假设准备这样做：如果 **A** 账户余额足够就从它取，否则就从账户 **B** 取。描述这种动作，要能在第一个动作 **retry** 时转到另一可能性

阻塞和选择

- 为支持选择，**STM Haskell** 提供了另一原语 **orElse**，其类型是

  orElse :: STM a -> STM a -> STM a

  orElse 把参数动作粘合为一个大动作（这一点与 **atomically** 类似）

  (orElse a1 a2) 的语义：先执行 **a1**；如果 **a1** 重试（调用 **retry**）就转去执行 **a2**；如果 **a2** 也重试，那就是整个 **orElse** 动作重试

- 完成前面问题的函数定义：

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ( )
-- (limitedWithdraw2 acc1 acc2 amt) withdraws amt from acc1,
-- if acc1 has enough money, otherwise from acc2.
-- If neither has enough, it retries.
limitedWithdraw2 acc1 acc2 amt
  = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

- 由于 **orElse** 的结果也是 **STM** 动作，可以把它作为另一个 **orElse** 的参数，这样可以写出多分支的选择结构【**orElse** 有没有结合性？】

基本 STM 小结

- 前面介绍了 **STM Haskell** 提供的最关键的事务内存操作
- 下表是一个总结：

操作	类型
atomically	**STM a -> IO a**
retry	**STM a**
orElse	**STM a -> STM a -> STM a**
newTVar	**a -> STM (TVar a)**
readTVar	**TVar a -> STM a**
writeTVar	**TVar a -> a -> STM ( )**

■前面没介绍过的唯一操作是 **newTVar**，它创建一个新的 **TVar** 单元（变量）。下面会用到

圣诞老人问题：描述

- 下面展示一个完整的可运行的 **STM** 程序
- 圣诞老人问题的简单描述：

  圣诞老人喜欢睡觉，直到他被

  从假期归来的九头驯鹿中的某一头叫醒，或

  10 个小矮人中的任（一组）三个叫醒

  如果是被驯鹿叫醒，圣诞老人就会把这些驯鹿套上雪橇，驾着雪橇去给孩子们送礼物，最后把驯鹿放开，让它们再去休假

  如果是被一组（三个）小矮人叫醒，他就带这组小矮人进自己的研究室，指导他们做玩具研发，最后让他们一个个离开，回去工作

  如果同时有驯鹿和一组小矮人叫醒他，圣诞老人优先去办驯鹿的事
- 这是讨论并发性的有名实例，有人用信号量、**Ada**、**Polyphonic C#**（**C#** 的一个并发扩展，加入了 **Joint Calculus** 并发编程模型）。下面用 **STM** 写，可以与它们比较

圣诞老人问题：基本构想

用 **STM Haskell** 解决这一问题的基本构想：

- 为小矮人建立一个 **Group**，也为驯鹿建立一个 **Group**
- 每个小矮人和驯鹿都试图去加入自己的那个 **Group**
- 如果成功加入，返回时就得到两个 **Gate**
  - 第一个 **Gate** 使圣诞老人控制何时允许小矮人进入研究室，且能知道他们是否都进入了（或将驯鹿套上雪橇，是否都套上）
  - 第二个 **Gate** 控制他们何时离开研究室（或完成送玩具工作）
- 圣诞老人等待他的两个 **Group** 之一就绪，并用 **Group** 的两个 **Gate** 引领其帮手（小矮人或驯鹿）进行工作
- 小矮人和驯鹿都运行在自己的无穷循环里：
  - 试图加入自己的 **Group**，在圣诞老人控制下通过两个 **Gate**
  - 在等待一段随机的时间后再次试图加入 **Group**

圣诞老人问题：小矮人和驯鹿

- 下面是描述小矮人行为的代码：

```
elf1 :: Group -> Int -> IO ( )
elf1 group elf_id = do { (in_gate, out_gate) <- joinGroup group
                       ; passGate in_gate
                       ; meetInStudy elf_id
                       ; passGate out_gate }
```

这个函数做一次循环，下面将基于它定义 elf（小矮人）

elf1 的参数是一个 Group 和一个 Int（作为具体小矮人的唯一标识），返回一个动作

- 小矮人标识只在 meetInStudy 用，输出一条信息说明发生的事件：

```
meetInStudy :: Int -> IO ( )
meetInStudy id =
        putStr ("Elf " ++ show id ++ " meeting in the study\n")
```

putStr 是库函数，调用 hPutStr stdout

圣诞老人问题：小矮人和驯鹿

- 小矮人调用 joinGroup 加入其 Group，然后调用 passGate 穿过门。函数的类型是：

```
joinGroup :: Group -> IO (Gate, Gate)
passGate  :: Gate  -> IO ( )
```

- 驯鹿的代码和小矮人代码几乎一样，只是输出信息的函数不同：

```
deliverToys :: Int -> IO ( )
deliverToys id =
        putStr ("Reindeer " ++ show id ++ " delivering toys\n")
```

圣诞老人问题：小矮人和驯鹿

- 由于 IO 动作也是值，可以从上面两个函数抽出一个公共模式：

```
helper1 :: Group -> IO () -> IO ( )
helper1 group do_task =
            do { (in_gate, out_gate) <- joinGroup group
               ; passGate in_gate
               ; do_task
               ; passGate out_gate }
```

- 这样，小矮人和驯鹿的基本行为都可以基于上面函数定义：

```
elf1, reindeer1 :: Group -> Int -> IO ( )
elf1          gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```


圣诞老人问题：Gate 和 Group

- 下面考虑 Gate，它支持如下接口操作：

```
newGate     :: Int -> STM Gate
passGate    :: Gate -> IO ( )
operateGate :: Gate -> IO ( )
```

每个 Gate 有确定的容量，在创建时给定；还有一个内部的可变变量称为“剩余容量”

调用 passGate 时将剩余容量减一，一旦减到0，再调用就阻塞

创建新 Gate 时剩余容量被设置为0，所以谁也不能通过

圣诞老人用 operateGate 打开 Gate，将其剩余容量设置为给定容量

- 下面是 Gate 的一种可能实现

圣诞老人问题：Gate

```
data Gate  = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do { tv <- newTVar 0; return (MkGate n tv) }

passGate :: Gate -> IO ( )
passGate (MkGate n tv)
  = atomically (do { n_left <- readTVar tv
                   ; check (n_left > 0)
                   ; writeTVar tv (n_left-1) })

operateGate :: Gate -> IO ( )
operateGate (MkGate n tv)
  = do { atomically (writeTVar tv n)
       ; atomically (do { n_left <- readTVar tv
                        ; check (n_left == 0) }) }
```

创建新的 Gate。其中用一个新创建的 TVar变量，初值 0

用 MkGate 解析参数，用 check 等待至 TVar 变量大于 0 时将它减一

将 TVar 的值设置为 n，然后等待直至该变量的值变为0

第一行将 Gate 声明唯一个新数据类型，其构造子是 MkGate，有一个 Int 类型的容量成员和一个 TVar 成员（表示还允许多少矮人/驯鹿通过）

圣诞老人问题：Group

- Group 的接口操作如下：

```
newGroup   :: Int -> IO Group
joinGroup  :: Group -> IO (Gate,Gate)
awaitGroup :: Group -> STM (Gate,Gate)
```

新创建的 Group 为空且有一指定容量（与 Gate 类似）

小矮人调用 joinGroup 企图加入 Group，如满则阻塞

圣诞老人调用 awaitGroup 等待 Group 满，满时会得到该 Group 对应的两个 Gate，并立即用两个新 Gate 初始化该 Group，使另一组小矮人可以在该 Group 里集结

- 下面是 Group 的一个可能实现

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))

newGroup n = atomically (do{ g1 <- newGate n; g2 <- newGate n
                           ; tv <- newTVar (n, g1, g2)
                           ; return (MkGroup n tv) })
```

Group 有两个成员
第二个成员是包含三个成分的 TVar 变量

在一个原子块里完成创建先创建两个 Gate，而后用它们创建一个 TVar 变量

圣诞老人问题：Group 操作

读出 Tvar 的成分，
等待剩余容量大于0，
将剩余容量减一写回，
返回两个 Gate

- joinGroup 和 awaitGroup 根据上面数据结构定义

```
joinGroup (MkGroup n tv)
  = atomically (do { (n_left, g1, g2) <- readTVar tv
                   ; check (n_left > 0)
                   ; writeTVar tv (n_left-1, g1, g2)
                   ; return (g1,g2) })

awaitGroup (MkGroup n tv)
  = do { (n_left, g1, g2) <- readTVar tv
       ; check (n_left == 0)
       ; new_g1 <- newGate n; new_g2 <- newGate n
       ; writeTVar tv (n,new_g1,new_g2)
       ; return (g1,g2) }
```

读出 Tvar 的成分，
等待剩余容量等于0，
创建两个新 Gate 并包装
返回原来的两个 Gate

- awaitGroup 每次创建两个新 Gate，保证圣诞老人和一组小矮人在研究室工作时，另一组小矮人可以集结。保证先后两组之间的清晰隔离

圣诞老人问题：有关实现的讨论

- 回顾一下，可看到 Gate 和 Group 接口中的一些操作是 IO 操作（如 newGroup和 joinGroup），另一些是 STM 操作（如 newGate 和 awaitGroup）。为什么这样设计？下面讨论这个问题
- 例如，newGroup 具有 IO 类型，这说明绝不能在 STM 操作里调用它。实际上这样做只是为了方便：
  - 可以在其定义里去掉 atomically，并将它定义为 STM 类型操作
  - 但是一旦这样修改后，每次调用 newGroup 时都必须将它包装在 atomically 里，不能直接写 newGroup n
  - 将 newGroup 定义为 STM 类型的优点是得到的操作更容易组合，而在这个程序里，newGroup 根本不需要组合
- 另一方面，由于程序需要在 newGroup 里调用 newGate，所以上面把 newGate 定义为 STM 类型

圣诞老人问题：有关实现的讨论

- 一般而言，设计程序库时应尽可能把函数定义为 **STM** 类型，这样
  - 定义的函数像可以任意组装的积木，可以随意用 **do** 块、**retry** 或 **orElse** 把组装它们，做出更大的 **STM** 操作
  - 这样做出的组合操作仍是性质不变的积木，可以进一步组装
- 一旦用 **atomically** 把一个块包装起来，就得到了一个 **IO** 操作，它：
  - 再也不能和其他操作包装为 **atomically** 块了
  - 但 **IO** 操作可以执行任意的不可撤销的输入输出动作（例如调用 **launchMissiles** 发射导弹）
- 总言之：在良好的程序库设计里，应尽可能引出 **STM** 操作（而不是 **IO** 操作），前者组合性更强，其类型说明它们不会产生不可撤销的副作用
  - 这样做，也使库的用户在需要时很容易用 **atomically** 把 **STM** 操作包装成 **IO** 操作（反过来是不可能的）

圣诞老人问题：有关实现的讨论

- 也有些情况里必须使用 **IO** 操作。看前面的 **operateGate**，其中两次使用 **atomically** 原语，这两个调用不能合并为一个
  - 第一个原子块要产生一个外部可以观察到的副作用（打开 **Gate**）
  - 第二个原子块阻塞到所有小矮人都被唤醒并通过 **Gate**

  只有第一个操作得到确认后才能开始第二个操作
  - 因此不能把上述操作包装到一个 **atomically** 里
  - 因此，**operateGate** 必须定义为 **IO** 操作

圣诞老人问题：主程序

- 现在考虑这个程序的框架，圣诞老人的实现等会考虑
- 主程序定义如下：

```
main = do { elf_group <- newGroup 3
          ; sequence_ [ elf elf_group n | n <- [1..10] ]

          ; rein_group <- newGroup 9
          ; sequence_ [ reindeer rein_group n | n <- [1..9] ]

          ; forever (santa elf_group rein_group) }
```

第一行创建一个大小为 3 的 Group

第二行创建一个序列（用内涵描述方式），序列元素来自对每个 n 属于 [1..10] 调用 elf elf_group n 。也就是说，这里创建的序列等于 [elf elf_group 1, elf elf_group 2, ..., elf elf_group 10] 创建的序列

每个调用返回一个 IO 动作，该动作生成一个小矮人线程

圣诞老人问题：主程序

- 函数 sequence_ 以一个动作序列为参数，返回一个动作，该动作执行时将顺序执行作为参数的动作序列里的一个个动作

```
sequence_ :: [IO a] -> IO ( )
```

- elf 是从 elf1 构造出来的，有两点不同：

  1，希望 elf 无穷循环

  2，希望每个 elf 在独立的线程里执行

- elf 的定义

```
elf :: Group -> Int -> IO ThreadId
elf gp id = forkIO (forever (do { elf1 gp id; randomDelay }))
```

forkIO 为其参数生成一个独立的 Haskell 线程

forever 反复执行其参数动作，这里是一个 do 块

randomDelay 延迟一段随机时间（后两个函数的定义见下页）

圣诞老人问题：主程序

- forever 函数的定义

```
forever :: IO () -> IO ( )
-- Repeatedly perform the action
forever act = do { act; forever act }
```

- 主程序里 forever 执行的动作是 elf1 gp id ， forever 导致它反复执行，希望一次执行完成后随机等待一段时间，用 randomDelay 描述

```
randomDelay :: IO ( )
-- Delay for a random time between 1 and 1,000,000 μs
randomDelay =
        do { waitTime <- getStdRandom (randomR (1, 1000000))
           ; threadDelay waitTime }
```

- 创建驯鹿和创建小矮人的方式一样：

```
reindeer :: Group -> Int -> IO ThreadId
reindeer gp id =
        forkIO (forever (do { reindeer1 gp id; randomDelay }))
```


圣诞老人问题：圣诞老人的实现

- 圣诞老人是本问题里最有趣的部分。他等待，直到一组小矮人或一组驯鹿来到。这时他可能需要选择做那个 Group 的事，而后带领小矮人或驯鹿去完成工作。下面是圣诞老人的代码

```
santa :: Group -> Group -> IO ( )
santa elf_gp rein_gp
= do { putStr "----------\n"
     ; (task, (in_gate, out_gate))
          <- atomically (orElse
               (chooseGroup rein_gp "deliver toys")
               (chooseGroup elf_gp  "meet in my study"))

     ; putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
     ; operateGate in_gate
      -- Now the helpers do their task
     ; operateGate out_gate }
  where
    chooseGroup :: Group -> String -> STM (String, (Gate,Gate))
    chooseGroup gp task = do { gates <- awaitGroup gp
                             ; return (task, gates) }
```

优先选择驯鹿，得到的task是一个字符串，用于显式，还有两个 Gate

打开第一个 Gate 等全组进入后操作第二个Gate

chooseGroup 是局部定义

圣诞老人问题：改进的实现

- 上述实现完全可以工作，但下面想看另一个实现，是更一般的版本
- 圣诞老人问题揭示了一种很常见的编程模式：
  - 一个线程（这里的圣诞老人线程）在一个原子事务里选择
  - 随后是一个或几个接续而来的事务，而后重复这个过程
- 这种模式的另一个例子：
  - 从一些消息队列里收到一个消息
  - 处理得到的消息，而后重复这个过程
- 在圣诞老人实例中，对于小矮人和驯鹿的后续操作差不多，圣诞老人都是去输出一段信息后顺序打开两个 **Gate**

  如果对小矮人和驯鹿，要做的事情很不一样，上面实现就不行了

  一种方法是返回一个布尔值指明所做选择，选择后基于布尔结果指派相应动作。但如果选择的可能性更多，这种设想就不好处理

圣诞老人问题：

- 下面的方法采用的技术类似于 **C** 语言里数据驱动的程序设计技术或者面向对象里的指派技术。定义：

```
santa :: Group -> Group -> IO ( )
santa elf_gp rein_gp
  = do { putStr "----------\n"
       ; choose [(awaitGroup rein_gp, run "deliver toys"),
                 (awaitGroup elf_gp,  run "meet in my study")] }
  where
    run :: String -> (Gate,Gate) -> IO ( )
    run task (in_gate,out_gate)
      = do { putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
           ; operateGate in_gate
           ; operateGate out_gate }
```

choose 是一种“卫式命令”，其参数是一组对偶，当某对偶的第一个元素可击发时（条件满足，完成），就运行相应的第二个参数

圣诞老人问题：

注意，run String 是一个 (Gate, Gate) -> IO ( ) 函数
圣诞老人用choose等待两组之一就绪，而后执行相应的 run "..." 函数

- choice 的类型是

  choose :: [(STM a, a -> IO ())] -> IO ( )

  即从对偶 (STM a, a -> IO ()) 的序列得到一个动作

  对偶中第一个元素必须是一个 STM 动作，返回类型为 a

  第二个元素是一个函数，从 a 得到一个动作

- 在看关键代码：

```
; choose [(awaitGroup rein_gp, run "deliver toys"),
          (awaitGroup elf_gp,  run "meet in my study")] }
 where
  run :: String -> (Gate,Gate) -> IO ( )
  run task (in_gate,out_gate)
   = do { putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
        ; operateGate in_gate
        ; operateGate out_gate }
```


圣诞老人问题：choose 函数

- choose 的实现：

```
choose :: [(STM a, a -> IO ())] -> IO ( )
choose choices = do { act <- atomically (foldr1 orElse actions)
                    ; act }
 where
  actions :: [STM (IO ( ))]
  actions = [ do { val <- guard; return (rhs val) }
            | (guard, rhs) <- choices ]
```

这里的 actions 是一个动作表，其中每个元素是一个动作

```
do { val <- guard; return (rhs val) }
```

其中的 guard 和 rhs 取自 choose 的参数 choices

(foldr1 orElse [a_1, a_2, …, a_n]) 将表用 orElse 展开，生成

(orElse a_1 (orElse a_2 … (orElse a_{n-1} a_n)...)) 作为 aotomically 块内容

choose 在分支中选择，得到一个动作，而后执行该动作

圣诞老人问题：编译和运行

- 有了上面代码，要运行它只需加几个 **import** 语句

```
import Control.Concurrent.STM
import Control.Concurrent
import System.Random
```

说明要使用一些库提供的功能

- 而后用 **GHC（Glasgow Haskell Compiler）**编译：

```
$ ghc Santa.hs -package stm -o santa
```

- 下面是一段运行输出

圣诞老人问题：

```
$ ./santa
----------
Ho! Ho! Ho! let's deliver toys
Reindeer 8 delivering toys
Reindeer 7 delivering toys
Reindeer 6 delivering toys
Reindeer 5 delivering toys
Reindeer 4 delivering toys
Reindeer 3 delivering toys
Reindeer 2 delivering toys
Reindeer 1 delivering toys
Reindeer 9 delivering toys
----------
Ho! Ho! Ho! let's meet in my study
Elf 3 meeting in the study
Elf 2 meeting in the study
Elf 1 meeting in the study
...and so on...
```

有关 Haskell

- 前面看了一些用 **Haskell** 写程序的例子，现在回来看这个语言本身
- **Haskell** 是一种函数式编程语言，作者认为它也是世界上最漂亮的命令式语言（因为可以写动作，修改状态）。作为命令式语言，可以看到它的一些独特之处：
  - 其类型系统严格区分动作（**action**）和纯粹的值
  - 动作是第一类对象，可以送给函数，作为结果返回，放入表中等，做所有这些事情时都没有副作用（只有执行动作才有副作用）
- 由于动作是第一类值，因此程序员可以自己针对具体应用定义特殊的控制结构，而不是只能用语言设计者提供的控制结构。例如：
  - 前面定义的 **nTimes** 就是一种简单的 **for** 循环
  - 前面定义的 **choose** 就实现了一种卫式命令
  - 前面的主程序里用 **Haskell** 强大的表达能力（表内涵表示）生成了一个动作表，然后用 **sequence_** 执行
  - 更早定义 **help1** 时，由一块代码抽象出一个动作，提高了模块性

有关 Haskell

- 当然，由于圣诞老人等例子很小，为展示 **Haskell** 的功能，作者可能过分使用了 **Haskell** 的抽象功能。但对更大的程序，将动作作为值带来的益处，怎么说都不过分
- **Haskell** 还有许多文中没用到的功能（因为本文主要关注并发），如高阶函数、懒求值、数据类型、多态性、类型类（**type class**）。实际上人们写出的大多数 **Haskell** 都不像这里的例子这样“命令式”
- **Haskell** 有一个专门的网址 **http://haskell.org**。其中有许多材料

总结

- 本文主要想让人理解一个事实：用 **STM Haskell** 写出的并发程序比传统的基于锁和条件变量的并发程序更具模块性
- 首先，事务内存使人能完全避免基于锁的并发程序中种种令人头痛的常见问题（“弄坏的锁”一节），这些绝不会出现在 **STM Haskell** 程序里
- 类型系统禁止在原子块外读写 **TVar** 变量。由于这里没有程序员可见的锁，所以有关应该获取哪个锁、按什么顺序等问题，根本不会出现
- 另一些情况也应专门提出，那就是在这里不必关心何时怎样去唤醒线程的问题，处理异常的问题，从错误中恢复的问题等
- 最值得提出的还是可组合性问题，在“弄坏的锁”一节里已经说明，这是基于锁的并行程序最致命的弱点
- 在 **STM Haskell** 里，任何 **STM** 类型的函数都可以用顺序或选择的方式与任何 **STM** 类型的函数组合，得到一个新的 **STM** 类型的函数
- 进另外：单独定义的 **STM** 函数能满足的原子性方面的性质，组合得到的 **STM** 函数也都能满足

总结

- 特别的：阻塞（**retry**）和选择（**orElse**）行为在基于锁的描述方式中是特别非模块化的，而在 **STM** 里完全模块化
- 考虑下面事务（其中使用了在“阻塞和选择”一节里定义的函数）

```
atomically (do { limitedWithdraw a1 10
               ; limitedWithdraw2 a2 a3 20 })
```

  事务阻塞到 **a1** 里有 **10** 单位的钱，以及 **a2** 或 **a3** 有了 **20** 单位的钱

  程序员没明确写出复杂的阻塞条件，如果**limitedWithdraw** 由库提供，程序员可能完全不知道实际阻塞条件。由此可以看到 **STM** 的模块化，小程序可以粘接成更大程序，这样做时无须暴露小程序的实现细节
- 这一简单介绍没提到事务内存的许多方面和重要问题，如嵌套的事务、异常、进展性、饿死、不变式等等。见有关 **Haskell** 的文章

总结

- 事务内存与 **Haskell** 配合得特别好
  - 理论上说实现 **STM** 需要跟踪每次内存读写，而在 **STM Haskell** 实现里只需跟踪 **TVar** 操作，而这些是所有操作中很小的一部分
  - 动作处理为第一类值，丰富的类型系统，使我们不必对语言做任何扩充，就能得到很强的静态保证
- 人们正努力将 **STM** 概念纳入主流命令式语言，但可能无法做的如此优雅，且需要语言的特殊支持。如何做是当前非常重要的一个研究课题
- **STM** 优于传统并发编程技术，就像高级语言优于汇编语言
  - 人们还是能写出错误百出的程序
  - 但许多很微秒的程序已经不会再出现了
  - 使人可以把更多精力放在程序更高层的方面和要解决的问题上
  - 当然，正如人们所公认的，没有一种银弹能使并发编程变得很容易
  - **STM** 像是一种前景绝好的技术，它能帮助人写出美丽的并发代码

讨论时间