

第二十八章：软件事务内存 (STM)

在并发编程的传统线程模型中，线程之间的数据共享需要通过锁来保持一致性(consistentBalance)，当数据产生变化时，还需要使用条件变量(condition variable)对各个线程进行通知。

某种程度上，Haskell 的 `MVar` 机制对上面提到的工具进行了改进，但是，它仍然带有和这些工具一样的缺陷：

- 因为忘记使用锁而导致条件竞争(race condition)
- 因为不正确的加锁顺序而导致死锁(deadblock)
- 因为未被捕捉的异常而造成程序崩溃(corruption)
- 因为错误地忽略了通知，造成线程无法正常唤醒(lost wakeup)

这些问题即使在很小的并发程序里也会经常发生，而在更加庞大的代码库或是高负载的情况下，这些问题会引发更加糟糕的难题。

比如说，对一个只有几个大范围锁的程序进行编程并不难，只是一旦这个程序在高负载的环境下运行，锁之间的相互竞争就会变得非常严重。另一方面，如果采用细粒度(fineo-grained)的锁机制，保持软件正常工作将会变得非常困难。除此之外，就算在负载不高的情况下，加锁带来的额外的簿记工作(book-keeping)也会对性能产生影响。

基础知识

软件事务内存(Software transactional memory)提供了一些简单但强大的工具。通过这些工具我们可以解决前面提到的大多数问题。通过 `atomically` 组合器(combinator)，我们可以在一个事务内执行一批操作。当这一组操作开始执行的时候，其他线程是觉察不到这些操作所产生的任何修改，直到所有操作完成。同样的，当前线程也无法察觉其他线程的所产生的修改。这些性质表明的操作的隔离性(isolated)。

当从一个事务退出的时候，只会发生以下情况中的一种：

- 如果没有其他线程修改了同样的数据，当前线程产生的修改将会对所有其他线程可见。
- 否则，当前线程的所产生的改动会被丢弃，然后这组操作会被重新执行。

`atomically` 这种全有或全无(all-or-nothing)的天性被称之为原子性(atomic)，`atomically` 也因为得名。如果你使用过支持事务的数据库，你会觉得STM使用起来非常熟悉。

一些简单的例子

在多玩家角色扮演的游戏里，一个玩家的角色会有许多属性，比如健康，财产以及金钱。让我们从基于游戏人物属性的一些简单的函数和类型开始去了解STM的精彩内容。随着学习的深入，我们也会不断地改进我们的代码。

STM的API位于 `stm` 包，模块 `Control.Concurrent.STM`。

```
-- file: ch28/GameInventory.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Control.Concurrent.STM
import Control.Monad

data Item = Scroll
  | Wand
  | Banjo
  deriving (Eq, Ord, Show)

newtype Gold = Gold Int
  deriving (Eq, Ord, Show, Num)

newtype HitPoint = HitPoint Int
  deriving (Eq, Ord, Show, Num)

type Inventory = TVar [Item]
type Health = TVar HitPoint
type Balance = TVar Gold

data Player = Player {
  balance :: Balance,
```

 v: latest ▾

```
health :: Health,
inventory :: Inventory
}
```

参数化类型 `TVar` 是一个可变量，可以在 `atomically` 块中读取或者修改。为了简单起见，我们把玩家的背包(`Inventory`)定义为物品的列表。同时注意到，我们用到了 `newtype`，这样不会混淆财富和健康属性。

当需要在两个账户(`Balance`)之间转账，我们所要做的就只是调整下各自的 `TVar`。

```
-- file: ch28/GameInventory.hs
basicTransfer qty fromBal toBal = do
  fromQty <- readTVar fromBal
  toQty    <- readTVar toBal
  writeTVar fromBal (fromQty - qty)
  writeTVar toBal   (toQty + qty)
```

让我们写个简单的测试函数

```
-- file: ch28/GameInventory.hs
transferTest = do
  alice <- newTVar (12 :: Gold)
  bob   <- newTVar 4
  basicTransfer 3 alice bob
  liftM2 (,) (readTVar alice) (readTVar bob)
```

如果我们在`ghci`里执行下这个函数，应该有如下的结果

```
ghci> :load GameInventory
[1 of 1] Compiling Main           ( GameInventory.hs, interpreted )
Ok, modules loaded: Main.
ghci> atomically transferTest
Loading package array-0.4.0.0 ... linking ... done.
Loading package stm-2.3 ... linking ... done.
(Gold 9,Gold 7)
```

原子性和隔离性保证了当其他线程同时看到 `bob` 的账户和 `alice` 的账户被修改了。

即使在并发程序里，我们也努力保持代码尽可能的纯函数化。这使得我们的代码更加容易推导和测试。由于数据并没有事务性，这也让底层的STM做更少的事。以下的纯函数实现了从我们来表示玩家背包的数列里移除一个物品。

```
-- file: ch28/GameInventory.hs
removeInv :: Eq a => a -> [a] -> Maybe [a]
removeInv x xs =
  case takeWhile (/= x) xs of
    (_,ys) -> Just ys
    []      -> Nothing
```

这里返回值用了 `Maybe` 类型，它可以用来表示物品是否在玩家的背包里。

下面这个事务性的函数实现了把一个物品给另外一个玩家。这个函数有一点点复杂因为需要判断给予者是否有这个物品。

```
-- file: ch28/GameInventory.hs
maybeGiveItem item fromInv toInv = do
  fromList <- readTVar fromInv
  case removeInv item fromList of
    Nothing    -> return False
    Just newList -> do
      writeTVar fromInv newList
      destItems <- readTVar toInv
      writeTVar toInv (item : destItems)
      return True
```

既然我们提供了有原子性和隔离型的事务，那么保证我们不能有意或是无意的从 `atomically` 执行块从脱离显得格外重要。借由 `STM monad`，Haskell 的类型系统保证了我们这种行为。

```
ghci> :type atomically
atomically :: STM a -> IO a
```

`atomically` 接受一个 `STM monad` 的动作，然后执行并让我们可以从 `IO monad` 里拿到这个结果。`STM monad` 是所有事务相关代码执行的地方。比如这些操作 `TVar` 值的函数都在 `STM monad` 里被执行。

```
ghci> :type newTVar
newTVar :: a -> STM (TVar a)
ghci> :type readTVar
readTVar :: TVar a -> STM a
ghci> :type writeTVar
writeTVar :: TVar a -> a -> STM ()
```

我们之前定义的事务性函数也有这个特性

```
-- file: ch28/GameInventory.hs
basicTransfer :: Gold -> Balance -> Balance -> STM ()
maybeGiveItem :: Item -> Inventory -> Inventory -> STM Bool
```

在 `STM monad` 里是不允许执行 I/O 操作或者是修改非事务性的可变状态，比如 `MVar` 的值。这就使得我们可以避免那些违背事务完整的操作。

重试一个事务

`maybeGiveItem` 这个函数看上去稍微有点怪异。只有当角色有这个物品时才会将它给另外一个角色，这看上去还算合理，然后返回一个 `Bool` 值使调用这个函数的代码变得复杂。下面这个函数调用了 `maybeGiveItem`，它必须根据 `maybeGiveItem` 的返回结果来决定如何继续执行。

```
maybeSellItem :: Item -> Gold -> Player -> Player -> STM Bool
maybeSellItem item price buyer seller = do
  given <- maybeGiveItem item (inventory seller) (inventory buyer)
  if given
    then do
      basicTransfer price (balance buyer) (balance seller)
      return True
    else return False
```

我们不仅要检查物品是否给到了另一个玩家，而且还得把是否成功这个信号传递给调用者。这就意味了复杂性被延续到了更外层。

下面我们来看看如何用更加优雅的方式处理事务无法成功进行的情况。`STM API` 提供了一个 `retry` 函数，它可以立即中断一个无法成功进行的 `atomically` 执行块。正如这个函数名本身所指明的意思，当它发生时，执行块会被重新执行，所有在这之前的操作都不会被记录。我们使用 `retry` 重新实现了 `maybeGiveItem`。

```
-- file: ch28/GameInventory.hs
giveItem :: Item -> Inventory -> Inventory -> STM ()

giveItem item fromInv toInv = do
  fromList <- readTVar fromInv
  case removeInv item fromList of
    Nothing -> retry
    Just newList -> do
      writeTVar fromInv newList
      readTVar toInv >>= writeTVar toInv . (item :)
```

我们之前实现的 `basicTransfer` 有一个缺陷：没有检查发送者的账户是否有足够的资金。我们可以使用 `retry` 来纠正这个问题并保持方法签名不变。

```
-- file: ch28/GameInventory.hs
transfer :: Gold -> Balance -> Balance -> STM ()
```

 v: latest ▾

```
transfer qty fromBal toBal = do
  fromQty <- readTVar fromBal
  when (qty > fromQty) $
    retry
  writeTVar fromBal (fromQty - qty)
  readTVar toBal >>= writeTVar toBal . (qty +)
```

使用 `retry` 后，销售物品的函数就显得简单很多。

```
sellItem :: Item -> Gold -> Player -> Player -> STM ()
sellItem item price buyer seller = do
  giveItem item (inventory seller) (inventory buyer)
  transfer price (balance buyer) (balance seller)
```

这个实现和之前的稍微有点不同。如果有必要会阻塞以至卖家有东西可卖并且买家有足够的余额支付，而不是在发现卖家没这个物品可销售时马上返回 `False`。

retry 时到底发生了什么？

`retry` 不仅仅使得代码更加简洁：它似乎有魔力般的内部实现。当我们调用 `retry` 的时候，它并不是马上重启事务，而是会先阻塞线程，一直到那些在 `retry` 之前被访问过的变量被其他线程修改。

比如，如果我们调用 `transfer` 而发现余额不足，`retry` 会自发的等待，直到账户余额的变动，然后会重新启动事务。同样的，对于函数 `giveItem`，如果卖家没有那个物品，线程就会阻塞直到他有了那个物品。

选择替代方案

有时候我们并不总是希望重启 `atomically` 操作即使调用了 `retry` 或者由于其他线程的同步修改而导致的失败。比如函数 `sellItem` 会不断地重试，只要没有满足其条件：要有物品并且余额足够。然而我们可能更希望只重试一次。

`orElse` 组合器允许我们在主操作失败的情况下，执行一个“备用”操作。

```
ghci> :type orElse
orElse :: STM a -> STM a -> STM a
```


我们对 `sellItem` 做了一点修改：如果 `sellItem` 失败，则 `orElse` 执行 `return False` 的动作从而使这个 `sale` 函数立即返回。

```
trySellItem :: Item -> Gold -> Player -> Player -> STM Bool
trySellItem item price buyer seller =
  sellItem item price buyer seller >> return True
`orElse`
  return False
```

在事务中使用高阶代码

假设我们想做稍微有挑战的事情，从一系列的物品中，选取第一个卖家拥有的并且买家能承担费用的物品进行购买，如果没有这样的物品则什么都不做。显然我们可以很直观的给出实现。

```
-- file: ch28/GameInventory.hs
crummyList :: [(Item, Gold)] -> Player -> Player
           -> STM (Maybe (Item, Gold))
crummyList list buyer seller = go list
  where go [] = return Nothing
        go (this@(item,price) : rest) = do
          sellItem item price buyer seller
          return (Just this)
        `orElse`
          go rest
```

在这个实现里，我们有碰到了一个问题：把我们的需求和如果实现混淆在一个。再深入一点观察，则会发现两个  `v: latest` 模式。

第一个就是让事务失败而不是重试。

```
-- file: ch28/GameInventory.hs
maybeSTM :: STM a -> STM (Maybe a)
maybeSTM m = (Just `liftM` m) `orElse` return Nothing
```

第二个，我们要对一系列的对象执行否一个操作，直到有一个成功为止。如果全部都失败，则执行 `retry` 操作。由于 `STM` 是 `MonadPlus` 类型类的一个实例，所以显得很方便。

```
-- file: ch28/STMPlus.hs
instance MonadPlus STM where
    mzero = retry
    mplus = orElse
```

`Control.Monad` 模块定义了一个 `msum` 函数，而它就是我们所需要的。

```
-- file: ch28/STMPlus.hs
msum :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero
```

有了这些重要的工具，我们就可以写出更加简洁的实现了。

```
-- file: ch28/GameInventory.hs
shoppingList :: [(Item, Gold)] -> Player -> Player
            -> STM (Maybe (Item, Gold))
shoppingList list buyer seller = maybeSTM . msum $ map sellOne list
    where sellOne this@(item, price) = do
        sellItem item price buyer seller
        return this
```

既然 `STM` 是 `MonadPlus` 类型类的实例，我们可以改进 `maybeSTM`，这样就可以适用于任何 `MonadPlus` 的实例。

```
-- file: ch28/GameInventory.hs
maybeM :: MonadPlus m => m a -> m (Maybe a)
maybeM m = (Just `liftM` m) `mplus` return Nothing
```

这个函数会在很多不同情况下显得非常有用。

I/O 和 STM

`STM monad` 禁止任意的 I/O 操作，因为 I/O 操作会破坏原子性和隔离性。当然 I/O 的操作还是需要的，只是我们需要非常的谨慎。

大多数时候，我们会执行 I/O 操作是由于我们在 `atomically` 块中产生的一个结果。在这些情况下，正确的做法通常是 `atomically` 返回一些数据，在 I/O monad 里的调用者则根据这些数据知道如何继续下一步动作。我们甚至可以返回需要被操作的动作 (action)，因为他们是第一类值 (First Class values)。

```
-- file: ch28/STMIO.hs
someAction :: IO a

stmTransaction :: STM (IO a)
stmTransaction = return someAction

doSomething :: IO a
doSomething = join (atomically stmTransaction)
```

我们偶尔也需要在 `STM` 里进行 I/O 操作。比如从一个肯定存在的文件里读取一些非可变数据，这样的操作并不会违背 `STM` 保证原子性和隔离性的原则。在这些情况，我们可以使用 `unsafeIOToSTM` 来执行一个 `IO` 操作。这个函数位于偏底层的一个模块 `GHC.Conc`，所以要谨慎使用。

```
ghci> :m +GHC.Conc
ghci> :type unsafeIOToSTM
unsafeIOToSTM :: IO a -> STM a
```

 v: latest ▾

我们所执行的这个 IO 动作绝对不能打开另外一个 atomically 事务。如果一个线程尝试嵌套的事务，系统就会抛出异常。

由于类型系统无法帮助我们确保 IO 代码没有执行一些敏感动作，最安全的做法就是我们尽可能的限制使用 unsafeIOToSTM。下面的例子展示了在 atomically 中执行 IO 的典型错误。

```
-- file: ch28/STMIO.hs
launchTorpedoes :: IO ()

notActuallyAtomic = do
  doStuff
  unsafeIOToSTM launchTorpedoes
  mightRetry
```

如果 mightRetry 会引发事务的重启，那么 launchTorpedoes 会被调用多次。事实上，我们无法预见它会被调用多少次，因为重试是由运行时系统所处理的。解决方案就是在事务中不要有这种类型的non-idempotent I/O操作。

线程之间的通讯

正如基础类型 TVar 那样，stm 包也提供了两个更有用的类型用于线程之间的通讯，TMVar 和 TChan。TMVar 是STM世界的 MVar，它可以保存一个 Maybe 类型的值，即 Just 值或者 Nothing。TChan 则是 STM 世界里的 Chan，它实现了一个有类型的先进先出(FIFO)通道。

[译者注：为何说 TMVar 是STM世界的 MVar 而不是 TVar？是因为从实践意义上理解的。MVar 的特性是要么有值要么为空的一个容器，所以当线程去读这个容器时，要么读到值继续执行，要么读不到值就等待。而 TVar 并没有这样的特性，所以引入了 TMVar。它的实现是这样的，newtype TMVar a = TMVar (TVar (Maybe a))，正是由于它包含了一个 Maybe 类型的值，这样就有了“要么有值要么为空”这样的特性，也就是 MVar 所拥有的特性。]

并发网络链接检查器

作为一个使用 STM 的实际例子，我们将开发一个检查HTML文件里不正确链接的程序，这里不正确的链接是指那些链接指向了一个错误的网页或是无法访问到其指向的服务器。用并发的方式解决这个问题非常得合适：如果我们尝试和已经下线的服务器(dead server)通讯，需要有两分钟的超时时间。如果使用多线程，即使有一两个线程由于和响应很慢或者下线的服务器通讯而停住(stuck)，我们还是可以继续进行一些有用的事情。

我们不能简单直观的给每一个URL新建一个线程，因为由于（也是我们预想的）大多数链接是正确的，那么这样做就会导致CPU或是网络连接超负荷。因此，我们只会创建固定数量的线程，这些线程会从一个队列里拿URL做检查。

```
-- file: ch28/Check.hs
{-# LANGUAGE FlexibleContexts, GeneralizedNewtypeDeriving,
      PatternGuards #-}

import Control.Concurrent (forkIO)
import Control.Concurrent.STM
import Control.Exception (catch, finally)
import Control.Monad.Error
import Control.Monad.State
import Data.Char (isControl)
import Data.List (nub)
import Network.URI
import Prelude hiding (catch)
import System.Console.GetOpt
import System.Environment (getArgs)
import System.Exit (ExitCode(..), exitWith)
import System.IO (hFlush, hPutStrLn, stderr, stdout)
import Text.Printf (printf)
import qualified Data.ByteString.Lazy.Char8 as B
import qualified Data.Set as S

-- 这里需要HTTP包，它并不是GHC自带的。
import Network.HTTP

type URL = B.ByteString

data Task = Check URL | Done
```

 v: latest ▾

`main` 函数显示了这个程序的主体脚手架(scaffolding)。

```
-- file: ch28/Check.hs
main :: IO ()
main = do
    (files,k) <- parseArgs
    let n = length files

    -- count of broken links
    badCount <- newTVarIO (0 :: Int)

    -- for reporting broken links
    badLinks <- newTChanIO

    -- for sending jobs to workers
    jobs <- newTChanIO

    -- the number of workers currently running
    workers <- newTVarIO k

    -- one thread reports bad links to stdout
    forkIO $ writeBadLinks badLinks

    -- start worker threads
    forkTimes k workers (worker badLinks jobs badCount)

    -- read links from files, and enqueue them as jobs
    stats <- execJob (mapM_ checkURLs files)
                (JobState S.empty 0 jobs)

    -- enqueue "please finish" messages
    atomically $ replicateM_ k (writeTChan jobs Done)

    waitFor workers

    broken <- atomically $ readTVar badCount

    printf fmt broken
            (linksFound stats)
            (S.size (linksSeen stats))
            n
    where
        fmt = "Found %d broken links. " ++
              "Checked %d links (%d unique) in %d files.\n"
```

当我们处于 `IO monad` 时，可以使用 `newTVarIO` 函数新建一个 `TVar` 值。同样的，也有类似的函数可以新建 `TMVar` 和 `TChan` 值。

在程序用了 `printf` 函数打印出最后的结果。和C语言里类似函数 `printf` 不同的是Haskell这个版本会在运行时检查参数的个数以及其类型。

```
ghci> :m +Text.Printf
ghci> printf "%d and %d\n" (3::Int)
3 and *** Exception: Printf.printf: argument list ended prematurely
ghci> printf "%s and %d\n" "foo" (3::Int)
foo and 3
```

在 `ghci` 里试试 `printf "%d" True` ,看看会得到什么结果。

支持 `main` 函数的是几个短小的函数。

```
-- file: ch28/Check.hs
modifyTVar_ :: TVar a -> (a -> a) -> STM ()
modifyTVar_ tv f = readTVar tv >>= writeTVar tv . f

forkTimes :: Int -> TVar Int -> IO () -> IO ()
forkTimes k alive act =
    replicateM_ k . forkIO $
        act
```

 v: latest ▾


```
    `finally`
    (atomically $ modifyTVar_ alive (subtract 1))
```

`forkTimes` 函数新建特定数量的相同的工作线程，每当一个线程推出时，则“活动”线程的计数器相应的减一。我们使用 `finally` 组合器确保无论线程是如何终止的，都会减少“活动”线程的数量。

下一步，`writeBadLinks` 会把每个失效或者死亡(dead)的连接打印到 `stdout`。

```
-- file: ch28/Check.hs
writeBadLinks :: TChan String -> IO ()
writeBadLinks c =
  forever $
    atomically (readTChan c) >>= putStrLn >> hFlush stdout
```

上面我们使用了 `forever` 组合器使一个操作永远的执行。

```
ghci> :m +Control.Monad
ghci> :type forever
forever :: (Monad m) => m a -> m ()
```

`waitFor` 函数使用了 `check`，当它的参数是 `False` 时会调用 `retry`。

```
-- file: ch28/Check.hs
waitFor :: TVar Int -> IO ()
waitFor alive = atomically $ do
  count <- readTVar alive
  check (count == 0)
```


检查一个链接

这个原生的函数实现了如何检查一个链接的状态。代码和 [第二十二章 Chapter 22, Extended Example: Web Client Programming] 里的 `podcatcher` 相似但有一点不同。

```
-- file: ch28/Check.hs
getStatus :: URI -> IO (Either String Int)
getStatus = chase (5 :: Int)
  where
    chase 0 _ = bail "too many redirects"
    chase n u = do
      resp <- getHead u
      case resp of
        Left err -> bail (show err)
        Right r ->
          case rspCode r of
            (3,_,_) ->
              case findHeader HdrLocation r of
                Nothing -> bail (show r)
                Just u' ->
                  case parseURI u' of
                    Nothing -> bail "bad URL"
                    Just url -> chase (n-1) url
            (a,b,c) -> return . Right $ a * 100 + b * 10 + c

    bail = return . Left

getHead :: URI -> IO (Result Response)
getHead uri = simpleHTTP Request { rqURI = uri,
                                     rqMethod = HEAD,
                                     rqHeaders = [],
                                     rqBody = "" }
```

为了避免无尽的重定向相应，我们只允许固定次数的重定向请求。我们通过查看HTTP标准HEAD信息来确认链接的有  [v: latest](#) 完整的GET请求，这样做可以减少网络流量。

这个代码是典型的“marching off the left of the screen”风格。正如之前我们提到的，需要谨慎使用这样的风格。下面我们用 `ErrorT monad transformer` 和几个通用一点的方法进行了重新实现，它看上去简洁了很多。

```
-- file: ch28/Check.hs
getStatusE = runErrorT . chase (5 :: Int)
where
  chase :: Int -> URI -> ErrorT String IO Int
  chase 0 _ = throwError "too many redirects"
  chase n u = do
    r <- embedEither show =<< liftIO (getHead u)
    case rspCode r of
      (3,_,_) -> do
        u' <- embedMaybe (show r) $ findHeader HdrLocation r
        url <- embedMaybe "bad URL" $ parseURI u'
        chase (n-1) url
      (a,b,c) -> return $ a*100 + b*10 + c

-- Some handy embedding functions.
embedEither :: (MonadError e m) => (s -> e) -> Either s a -> m a
embedEither f = either (throwError . f) return

embedMaybe :: (MonadError e m) => e -> Maybe a -> m a
embedMaybe err = maybe (throwError err) return
```

工作者线程

每个工作者线程(Worker Thread)从一个共享队列里拿一个任务，这个任务要么检查链接有效性，要么让线程推出。

```
-- file: ch28/Check.hs
worker :: TChan String -> TChan Task -> TVar Int -> IO ()
worker badLinks jobQueue badCount = loop
where
  -- Consume jobs until we are told to exit.
  loop = do
    job <- atomically $ readTChan jobQueue
    case job of
      Done -> return ()
      Check x -> checkOne (B.unpack x) >> loop

  -- Check a single link.
  checkOne url = case parseURI url of
    Just uri -> do
      code <- getStatus uri `catch` (return . Left . show)
      case code of
        Right 200 -> return ()
        Right n -> report (show n)
        Left err -> report err
    _ -> report "invalid URL"

  where report s = atomically $ do
    modifyTVar_ badCount (+1)
    writeTChan badLinks (url ++ " " ++ s)
```

查找链接

我们构造了基于 `IO monad` 的状态 `monad transformer` 栈用于查找链接。这个状态会记录我们已经找到过的链接(避免重复)、链接的数量以及一个队列，我们会把需要做检查的链接放到这个队列里。

```
-- file: ch28/Check.hs
data JobState = JobState { linksSeen :: S.Set URL,
                           linksFound :: Int,
                           linkQueue :: TChan Task }

newtype Job a = Job { runJob :: StateT JobState IO a }
deriving (Monad, MonadState JobState, MonadIO)
```

 v: latest ▾

```
execJob :: Job a -> JobState -> IO JobState
execJob = execStateT . runJob
```

严格来说，对于对立运行的小型程序，我们并不需要用到 `newtype`，然后我们还是将它作为一个好的编码实践的例子放在这里。(毕竟也只多了几行代码)

`main` 函数实现了对每个输入文件调用一次 `checkURLs` 方法，所以 `checkURLs` 的参数就是单个文件。

```
-- file: ch28/Check.hs
checkURLs :: FilePath -> Job ()
checkURLs f = do
    src <- liftIO $ B.readFile f
    let urls = extractLinks src
    filterM seenURI urls >>= sendJobs
    updateStats (length urls)

updateStats :: Int -> Job ()
updateStats a = modify $ \s ->
    s { linksFound = linksFound s + a }

-- / Add a link to the set we have seen.
insertURI :: URL -> Job ()
insertURI c = modify $ \s ->
    s { linksSeen = S.insert c (linksSeen s) }

-- / If we have seen a link, return False. Otherwise, record that we
-- have seen it, and return True.
seenURI :: URL -> Job Bool
seenURI url = do
    seen <- (not . S.member url) `liftM` gets linksSeen
    insertURI url
    return seen

sendJobs :: [URL] -> Job ()
sendJobs js = do
    c <- gets linkQueue
    liftIO . atomically $ mapM_ (writeTChan c . Check) js
```

`extractLinks` 函数并没有尝试去准确的去解析一个HTMP或是文本文件，而只是匹配那些看上去像URL的字符串。我们认为这样做就够了。

```
-- file: ch28/Check.hs
extractLinks :: B.ByteString -> [URL]
extractLinks = concatMap uris . B.lines
    where uris s      = filter looksOkay (B.splitWith isDelim s)
          isDelim c    = isControl c || c `elem` " <>\"{|\`[]`"
          looksOkay s = http `B.isPrefixOf` s
          http         = B.pack "http:"
```

命令行的实现

我们使用了 `System.Console.GetOpt` 模块来解析命令行参数。这个模块提供了很多解析命令行参数的很有用的方法，不过使用起来稍微有点繁琐。

```
-- file: ch28/Check.hs
data Flag = Help | N Int
    deriving Eq

parseArgs :: IO ([String], Int)
parseArgs = do
    argv <- getArgs
    case parse argv of
        ([], files, [])      -> return (nub files, 16)
        (opts, files, [])
            | Help `elem` opts -> help
            | [N n] <- filter (/=Help) opts -> return (nub files, n)
        (_, _, errs)         -> die errs
```

 v: latest ▾

```

where
  parse argv = getOpt Permute options argv
  header    = "Usage: urlcheck [-h] [-n n] [file ...]"
  info      = usageInfo header options
  dump      = hPutStrLn stderr
  die errs  = dump (concat errs ++ info) >> exitWith (ExitFailure 1)
  help      = dump info                >> exitWith ExitSuccess

```

getOpt 函数接受三个参数

参数顺序的定义。它定义了选项(Option)是否可以和其他参数混淆使用(就是我们上面用到的 `Permute`)或者是选项必须出现在参数之前。

选项的定义。每个选项有这四个部分组成：简称，全称，选项的描述(比如是否接受参数)以及用户说明。

参数和选项数组，类似于 `getArgs` 的返回值。

这个函数返回一个三元组，包括用户输入的选项，参数以及错误信息(如果有的话)。

我们使用 `Flag` 代数类型(Algebraic Data Type)表示程序所能接收的选项。

```

-- file: ch28/Check.hs
options :: [OptDescr Flag]
options = [ Option ['h'] ["help"] (NoArg Help)
           "Show this help message",
           Option ['n'] [] (ReqArg (\s -> N (read s)) "N")
           "Number of concurrent connections (default 16)" ]

```

`options` 列表保存了每个程序能接收选项的描述。每个描述必须要生成一个 `Flag` 值。参考上面例子中是如何使用 `NoArg` 和 `ReqArg`。 `GetOpt` 模块的 `ArgDescr` 类型有很多构造函数(Constructors)。

```

-- file: ch28/GetOpt.hs
data ArgDescr a = NoArg a
                | ReqArg (String -> a) String
                | OptArg (Maybe String -> a) String

```

`NoArg` 接受一个参数用来表示这个选项。在我们这个例子中，如果用户在调用程序时输入 `-h` 或者 `--help`，我们就用 `Help` 值表示。

`ReqArg` 的第一个函数作为参数，这个函数把用户输入的参数转化成相应的值；第二个参数是用来说明的。这里我们是将字符串转换为数值(integer)，然后再给类型 `Flag` 的构造函数 `N`。

`OptArg` 和 `ReqArg` 很相似，但它允许选项没有对应的参数。

模式守卫 (Pattern guards)

函数 `parseArgs` 的定义里其实潜在了一个语言扩展(Language Extension), Pattern guards。用它可以写出更加简要的 guard expressions。它通过语言扩展 `PatternGuards` 来使用。

一个Pattern Guard有三个组成部分：一个模式(Pattern)，一个 `<-` 符号以及一个表达式。表达式会被解释然后和模式相匹配。如果成功，在模式中定义的变量会被赋值。我们可以在一个guard里同时使用pattern guards和普通的 `Bool` guard expressions。

```

-- file: ch28/PatternGuard.hs
{-# LANGUAGE PatternGuards #-}

testme x xs | Just y <- lookup x xs, y > 3 = y
            | otherwise                    = 0

```

在上面的例子中，当关键字 `x` 存在于alist `xs` 并且大于等于3，则返回它所对应的值。下面的定义实现了同样的功能。

```

-- file: ch28/PatternGuard.hs
testme_noguards x xs = case lookup x xs of
    Just y | y > 3 -> y
    _              -> 0

```

Pattern guards 使得我们可以把一系列的guards和 `case` 表达式组合到单个guard，从而写出更加简洁并容易理解的guard~

 v: latest ▾

STM的实践意义

至此我们还并未提及STM所提供的特别优越的地方。比如它在做组合(*composes*)方面就表现的很好：当需要向一个事务中增加逻辑时，只需要用到常见的函数 (`>>=`) 和 (`>>`)。

组合的概念在构建模块化软件是显得格外重要。如果我们把两段都没有问题的代码组合在一起，也应该是能很好工作的。常规的线程编程技术无法实现组合，然而由于STM提供了一些很关键的前提，从而使在线程编程时使用组合变得可能。

STM monad防止了我们意外的非事务性的I/O。我们不再需要关心锁的顺序，因为代码里根本没有锁机制。我们可以忘记丢失唤醒，因为不再有条件变量了。如果有异常发生，我们则可以用函数 `catchSTM` 捕捉到，或者是往上级传递。最后，我们可以用 `retry` 和 `orElse` 以更加漂亮的方式组织代码。

采用STM机制的代码不会死锁，但是导致饥饿还是有可能的。一个长事务导致另外一个事务不停的 `retry`。为了解决这样的问题，需要尽量的短事务并保持数据一致性。

合理的放弃控制权

无论是同步管理还是内存管理，经常会遇到保留控制权的情况：一些软件需要对延时或是内存使用记录有很强的保证，因此就必须花很多时间和精力去管理和调试显式的代码。然后对于软件的大多数实际情况，垃圾回收(Garbage Collection)和STM已经做的足够好了。

STM并不是一颗完美的灵丹妙药。当我们选择垃圾回收而不是显式的内存管理，我们是放弃了控制权从而获得更加安全的代码。同样的，当使用STM时，我们放弃了底层的细节，从而希望代码可读性更好，更加容易理解。

使用不变量

STM并不能消除某些类型的bug。比如，我们在一个 `atomically` 事务中从某个账号中取钱，然后返回到 `IO monad`，然后在另一个 `atomically` 事务中把钱存到另一个账号，那么代码就会产生不一致性，因为会在某个特定时刻，这部分钱不会出现的任意一个账号里。

```
-- file: ch28/GameInventory.hs
bogusTransfer qty fromBal toBal = do
  fromQty <- atomically $ readTVar fromBal
  -- window of inconsistency
  toQty   <- atomically $ readTVar toBal
  atomically $ writeTVar fromBal (fromQty - qty)
  -- window of inconsistency
  atomically $ writeTVar toBal   (toQty + qty)

bogusSale :: Item -> Gold -> Player -> Player -> IO ()
bogusSale item price buyer seller = do
  atomically $ giveItem item (inventory seller) (inventory buyer)
  bogusTransfer price (balance buyer) (balance seller)
```

在同步程序中，这类问题显然很难而且不容易重现。比如上述例子中的不一致性问题通常只存在一段很短的时间内。在开发阶段通常不会出现这类问题，而往往只有在负载很高的产品环境才有可能发生。

我们可以用函数 `alwaysSucceeds` 定义一个不变量，它是永远为真的一个数据属性。

```
ghci> :type alwaysSucceeds
alwaysSucceeds :: STM a -> STM ()
```

当创建一个不变量时，它马上会被检查。如果要失败，那么这个不变量会抛出异常。更有意思的是，不变量会在经后每个事务完成时自动被检查。如果在任何一个点上失败，事务就会推出，不变量抛出的异常也会被传递下去。这就意味着当不变量的条件被违反时，我们就可以马上得到反馈。

比如，下面两个函数给本章开始时定义的游戏世界增加玩家

```
-- file: ch28/GameInventory.hs
newPlayer :: Gold -> HitPoint -> [Item] -> STM Player
newPlayer balance health inventory =
  Player `liftM` newTVar balance
    `ap` newTVar health
    `ap` newTVar inventory

populateWorld :: STM [Player]
```

 v: latest ▾

```
populateWorld = sequence [ newPlayer 20 20 [Wand, Banjo],
                           newPlayer 10 12 [Scroll] ]
```

下面的函数则返回了一个不变量，通过它我们可以保证整个游戏世界资金总是平衡的：即任何时候的资金总量和游戏建立时的总量是一样的。

```
-- file: ch28/GameInventory.hs
consistentBalance :: [Player] -> STM (STM ())
consistentBalance players = do
    initialTotal <- totalBalance
    return $ do
        curTotal <- totalBalance
        when (curTotal /= initialTotal) $
            error "inconsistent global balance"
    where totalBalance = foldM addBalance 0 players
          addBalance a b = (a+) `liftM` readTVar (balance b)
```

下面我们写个函数来试验下。

```
-- file: ch28/GameInventory.hs
tryBogusSale = do
    players@(alice:bob:_) <- atomically populateWorld
    atomically $ alwaysSucceeds =<< consistentBalance players
    bogusSale Wand 5 alice bob
```

由于在函数 `bogusTransfer` 中不正确地使用了 `atomically` 而会导致不一致性，当我们在 `ghci` 里运行这个方法时则会检测到这个不一致性。

```
ghci> tryBogusSale
*** Exception: inconsistent global balance
```

讨论



开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号


姓名

来做第一个留言的人吧！

在 REAL WORLD HASKELL 中文版 上还有

第九章：I/O学习 —— 构建一个用于搜索文件系统的库 — Real World Haskell 中文版

3条评论 • 4年前

 在原佐为 — “继续从左往右看” 这里英文是“from right to left”，是从右到左吧。

Real World Haskell 中文版 — Real World Haskell 中文版

4条评论 • 6年前

 Jojo — 更新好慢呀


Pearls of Functional Algorithm Design — Pearls of Functional Algorithm Design

1条评论 • 6年前

 forlice — ...

第三章：Defining Types, Streamlining Functions¶

4条评论 • 4年前

 Wengel An —