CS314. Функциональное программирование

Лекция 21. Обработка файлов и исключения. Конкурентное программирование

В. Н. Брагилевский

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

16 декабря 2016 г.

Содержание

Содержание

Постановка задачи

Дан текстовый файл, в каждой строке которого записано два натуральных числа, разделённых запятой, например:

```
2,4
5,2
1,9
5,5
```

Найти сумму чисел в каждой строке и вывести результат в новый файл следующего вида:

```
2+4=6
5+2=7
1+9=10
5+5=10
```

Идеи решения

- Чтение содержимого файла
- Построчная обработка
- Преобразование строки: `2,4` $\mapsto$ 2+4=6
- Вывод результатов

Простейшее решение

```
import System.Environment

oneLine :: String -> String
oneLine xs = concat [before, "+", after, "=", res]
  where
    (before, (_ : after)) = break (==',') xs
    n1 = read before
    n2 = read after
    res = show $ n1 + n2

allLines :: String -> String
allLines = unlines . map oneLine . lines

main = do
  [inf, outf] <- getArgs
  allLines <$> readFile inf >>= writeFile outf
```

Проблемы простейшего решения

- Слабая производительность: на обработку файла размером 8Мб требуется около 7с.
- Неустойчивость к ошибкам: отсутствие входного файла, некорректный формат файла.

Содержание

Строки String и их замена на Data.Text

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T
import Data.Text.Read

oneLine :: T.Text -> T.Text
oneLine xs = T.concat [before, "+", after, "=", res]
  where
    [before, after] = T.splitOn "," xs
    Right (n1, "") = decimal before
    Right (n2, "") = decimal after
    res = T.pack $ show $ n1 + n2

allLines :: String -> String
allLines = T.unpack
         . T.unlines . map oneLine . T.lines
         . T.pack
```

Переход к строгому IO

```haskell
import qualified Data.Text.IO as TIO

allLines :: T.Text -> T.Text
allLines = T.unlines . map oneLine . T.lines

main = do
  [inf, outf] <- getArgs
  allLines <$> TIO.readFile inf >>= TIO.writeFile outf
```

- Общий результат (по двум изменениям): время обработки 8Мб-файла сократилось с 7с до 2,4с.
- Дополнительная способность Data.Text: работа с кодировками.

Содержание

Борьба за устойчивость

Возможные проблемы

- Отсутствие входного файла
- Некорректный формат строки: `<значение>,<значение>`
- Некорректный формат числа

Стратегии обработки проблем с форматом

- Пропуск некорректных строк
- Прерывание обработки и вывод сообщения пользователю

- Основная проблема: проблемы с форматом обнаруживаются в чистых функциях, а сообщать о них надо в IO.

Решение с пропуском некорректных строк

```
oneLine :: T.Text -> Either String T.Text
oneLine xs = check $ T.splitOn "," xs
  where
    check xs@[before, after]
      = (\res -> T.concat [before, "+", after, "=", res])
        . T.pack . show . sum  <$> mapM parseNumber xs
    check _ = Left "incorrect line format"

    parseNumber s
      | Right (res, "") <- decimal s = Right res
    parseNumber _ = Left "incorrect numeric format"

allLines :: T.Text -> T.Text
allLines = T.unlines . map (\(Right s) -> s)
           . filter isRight . map oneLine . T.lines
```

Решение с прерыванием обработки (исключения)

```
data FormatException =
        IncorrectLineFormat | IncorrectNumericFormat
        deriving (Show, Typeable)
instance Exception FormatException

oneLine :: T.Text -> T.Text
oneLine xs = check $ T.splitOn "," xs
  where
    check xs@[before, after]
      = (\res -> T.concat [before, "+", after, "=", res])
        . T.pack . show . sum
        $ map parseNumber xs
    check _ = throw IncorrectLineFormat
    parseNumber s
      | Right (res, "") <- decimal s = res
    parseNumber _ = throw IncorrectNumericFormat
```

Решение с прерыванием обработки (исключения)

```
allLines :: T.Text -> T.Text
allLines = T.unlines . map oneLine . T.lines

main = do
  [inf, outf] <- getArgs
  (allLines <$> TIO.readFile inf >>= TIO.writeFile outf)
    `catch` \e -> do
      print (e :: FormatException)
```

Заключение по примеру

- Вместо String нужно всегда использовать Data.Text.
- Строгий или ленивый ввод-вывод — зависит от обстоятельств. В случае проблем с производительностью ленивого IO можно пробовать переходить на строгий IO или смотреть в сторону библиотек pipes/conduit.
- Исключения можно рассматривать как альтернативу типам (Maybe/Either).
- Вообще-то разбор содержимого файлов — это задача для парсеров (Parsec/attoparsec).

Содержание

Содержание

Запуск потока

```
forkIO :: IO () -> IO ThreadId
```

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  _ <- forkIO (replicateM_ 100000 (putChar 'A'))
  replicateM_ 100000 (putChar 'B')
```

Результат запуска

```
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
...
```

- По умолчанию программа использует одно ядро процессора.

```
$ ghc thr.hs -threaded -rtsopts
$ ./thr +RTS -N2
```

Генерация исключения в потоке

```
throwTo :: Exception e => ThreadId -> e -> IO ()
```

```
import Control.Concurrent
import Control.Exception
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  t1 <- forkIO (replicateM_ 100000 (putChar 'A'))
  t2 <- forkIO (replicateM_ 100000 (putChar 'B'))
  _ <- getLine
  throwTo t1 ThreadKilled
  throwTo t2 ThreadKilled
  putStrLn "\nBye..."
```

Пример программы: напоминалки

```
import Control.Concurrent
import Text.Printf
import Control.Monad

main = forever $ do
    s <- getLine
    forkIO $ setReminder (read s)

setReminder :: Int -> IO ()
setReminder t  = do
  printf "Ok, I'll remind you in %d seconds\n" t
  threadDelay (10^6 * t)
  printf "%d seconds is up! BING!\BEL\n" t
```

Использование программы

```
$ ./reminders
2
Ok, I'll remind you in 2 seconds
3
Ok, I'll remind you in 3 seconds
4
Ok, I'll remind you in 4 seconds
2 seconds is up! BING!
3 seconds is up! BING!
4 seconds is up! BING!
```

Завершение программы по запросу пользователя

```
main = do
    s <- getLine
    when (s /= "exit") $ do
      forkIO $ setReminder (read s)
      main
```

```
$ ./reminders2
2
Ok, I'll remind you in 2 seconds
3
Ok, I'll remind you in 3 seconds
2 seconds is up! BING!
exit
$
```

Содержание

Переменные MVar

```
data MVar a

newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

- Переменная может быть пустой.
- Операции takeMVar и putMVar могут заблокироваться.

Простейшие способы использования переменных MVar

```
main = do
  m <- newEmptyMVar
  forkIO $ putMVar m 'x'
  r <- takeMVar m
  print r
```

```
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r <- takeMVar m
  print r
  r <- takeMVar m
  print r
```

Ошибки при работе с MVar

```
main = do
  m <- newEmptyMVar
  takeMVar m
```

```
$ ./mvar
mvar: thread blocked indefinitely in an MVar operation
```

Основные варианты использования переменных MVar

- Одноместный канал для передачи сообщений между потоками.
- Контейнер для разделяемого изменяемого состояния (shared mutable state).
- Строительный блок для формирования конкурентных структур данных.

Содержание

Программная транзакционная память (STM)

- Software transactional memory
- Механизм организации конкурентного доступа к разделяемым данным (в противовес синхронизации на блокировках).
- Последовательность операций чтения и записи в виде транзакции: её результаты либо фиксируются, либо отменяются полностью.
- Упрощение программирования
- Предотвращение взаимоблокировок
- Возможность композиции транзакций

Интерфейс STM в Haskell

```
data STM a
instance Monad STM

atomically :: STM a -> IO a

data TVar a
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()

retry      :: STM a
orElse     :: STM a -> STM a -> STM a
throwSTM   :: Exception e => e -> STM a
catchSTM   :: Exception e => STM a -> (e -> STM a) -> STM a
```

Где тут композиция транзакций?

```haskell
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

main = do
  shared <- atomically $ newTVar 0
  before <- atomically $ readTVar shared
  putStrLn $ "Before: " ++ show before
  forkIO $ replicateM_ 25 (dispVar shared >> milliSleep 20)
  forkIO $ replicateM_ 10 (appV (+2) shared >> milliSleep 50)
  forkIO $ replicateM_ 20 (appV (subtract 1) shared
                           >> milliSleep 25)
  milliSleep 800
  after <- atomically $ readTVar shared
  putStrLn $ "After: " ++ show after
 where
   milliSleep = threadDelay . (*) 1000

dispVar x = (atomically $ readTVar x) >>= print
appV fn x = atomically $ readTVar x >>= writeTVar x . fn
```

```
$ ./stm
Before: 0
0
1
0
1
0
...
0
1
1
0
1
0
After: 0
```

Содержание

Протокол службы удвоения

- Сервер принимает соединения от клиентов на порту 44444.
- Если клиент присылает целое число n, то сервер отвечает удвоенным значением $2n$.
- Если клиент присылает строку “`end`”, то сервер закрывает соединение.
- Обслуживание клиентов должно проходить параллельно.

Обработка одного соединения

```
talk :: Handle -> IO ()
talk h = do
  hSetBuffering h LineBuffering
  loop
 where
  loop = do
    line <- hGetLine h
    if line == "end"
       then hPutStrLn h ("Thank you for using the " ++
                         "Haskell doubling service.")
       else do
          hPutStrLn h (show (2 * (read line :: Integer)))
          loop
```

Основная программа

```
main = withSocketsDo $ do
  sock <- listenOn (PortNumber (fromIntegral port))
  printf "Listening on port %d\n" port
  forever $ do
    (handle, host, port) <- accept sock
    printf "Accepted connection from %s: %s\n" host
                                          (show port)
    forkFinally (talk handle) (\_ -> hClose handle)

port :: Int
port = 44444
```

- withSocketsDo
- listenOn
- accept
- forkFinally

Компиляция и запуск сервера

```
1 2 3 4 [5] 6 : Tall : bravit@desktop:~
[bravit@desktop server]$ ghc server.hs -threaded -rtsopts
[1 of 1] Compiling Main             ( server.hs, server.o )
Linking server ...
[bravit@desktop server]$ ./server +RTS  -N4
Listening on port 44444
Accepted connection from localhost.localdomain: 59526
Accepted connection from localhost.localdomain: 59532
```

Примеры клиентов

```
$ nc localhost 44444
22
44
33
66
end
Thank you for using the Haskell doubling service.
```

```
$ ghc -e 'mapM_ print [1..]' | nc localhost 44444
2
4
6
...
```

Литература

1. С. Марлоу. Параллельное и конкурентное программирование на языке Haskell. Текст на английском доступен онлайн: `http://chimera.labs.oreilly.com/books/1230000000929/index.html`