CS314. Функциональное программирование

Лекция 22. Сетевое и распределённое программирование

В. Н. Брагилевский

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

17 декабря 2016 г.

Содержание

Содержание

Пример: загрузка веб-ресурсов

```
import Control.Concurrent
import Data.ByteString as B
import GetURL
```

```
getURL :: String -> IO ByteString
```

```
main = do
  m1 <- newEmptyMVar
  m2 <- newEmptyMVar
  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Shovel"
    putMVar m1 r
  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Spade"
    putMVar m2 r
  r1 <- takeMVar m1
  r2 <- takeMVar m2
  print (B.length r1, B.length r2)
```

```haskell
module GetURL (getURL) where

import Network.HTTP
import Network.Browser
import Network.URI
import Data.ByteString (ByteString)

getURL :: String -> IO ByteString
getURL url = Network.Browser.browse $ do
  setCheckForProxy True
  setDebugLog Nothing
  setOutHandler (const (return ()))
  (_,rsp) <- request $ getRequest' $ escapeURIString isUnescapedInURI url
  return (rspBody rsp)
 where
   getRequest' :: String -> Request ByteString
   getRequest' urlString =
    case parseURI urlString of
      Nothing -> error ("getRequest: Not a valid URL - " ++ urlString)
      Just u  -> mkRequest GET u
```

Пример решения с библиотекой Async

```
import Control.Concurrent.Async

main = do
  a1 <- async (getURL "http://www.wikipedia.org/wiki/Shovel")
  a2 <- async (getURL "http://www.wikipedia.org/wiki/Spade")
  r1 <- wait a1
  r2 <- wait a2
  print (B.length r1, B.length r2)
```

```
data Async a
async :: IO a -> IO (Async a)
wait :: Async a -> IO a
waitCatch :: Async a -> IO (Either SomeException a)
```

Пример с обработкой ошибок

```
main = do
  a1 <- async (getURL "http://ww.wikipedia.org/wiki/Shovel")
  a2 <- async (getURL "http://ww.wikipedia.org/wiki/Spade")
  handle (\e  -> print (e::SomeException)) $
    do
      r1 <- wait a1
      r2 <- wait a2
      print (B.length r1, B.length r2)
```

Запуск множества асинхронных действий

```
sites = ["http://www.google.com",
         "http://www.bing.com",
         "http://www.yahoo.com",
         "http://www.wikipedia.com/wiki/Spade",
         "http://www.wikipedia.com/wiki/Shovel"]

timeDownload :: String -> IO ()
timeDownload url = do
  (page, time) <- timeit $ getURL url
  printf "downloaded: %s (%d bytes, %.2fs)\n"
           url (B.length page) time

main = do
 as <- mapM (async . timeDownload) sites
 mapM_ wait as
```

Слияние событий

```
waitBoth :: Async a -> Async b -> IO (a, b)
waitEither :: Async a -> Async b -> IO (Either a b)
waitAny :: [Async a] -> IO (Async a, a)
```

Пример: ожидание завершения первой загрузки

```
download url = do
    r <- getURL url
    return (url, r)

main :: IO ()
main = do
  as <- mapM (async . download) sites
  (url, r) <- waitAny as
  printf "%s was first (%d bytes)\n" url (B.length r)
  mapM_ wait as
```

- Результат асинхронного действия можно запрашивать вторично!

Аннулирование асинхронного действия

```
cancel :: Async a -> IO ()
cancel a = throwTo (asyncThreadId a) ThreadKilled
```

- Если действие уже завершено, то ничего не происходит

Пример: аннулирование текущих загрузок

```
main = do
  as <- mapM (async . timeDownload) sites

  forkIO $ do
     hSetBuffering stdin NoBuffering
     forever $ do
        c <- getChar
        when (c == 'q') $ mapM_ cancel as

  rs <- mapM waitCatch as
  printf "%d/%d succeeded\n" (length (rights rs)) (length rs)
```

Содержание

Распределённое программирование

- Что такое распределённое программирование и распределённая программа?
- Коммуникация: передача сообщений и альтернативные подходы.
- Проблемы именования.
- Синхронизация.
- Согласованность состояния и репликация.
- Время в распределённых системах.
- Отношение к отказам и их обработка.

Cloud Haskell

- Инструмент конкурентного и распределённого программирования в стиле языка Erlang.
- Основные компоненты:
  - интерфейс транспортного уровня и его реализации (TCP, локальная память, Microsoft Azure и др.);
  - библиотеки для передачи статических замыканий (код и данные) на удалённые узлы;
  - библиотеки, реализующие возможности, аналогичные Erlang OTP (Open Telecom Platform).
- Важные библиотеки:
  - distributed-process
  - network-transport-tcp (distributed-process-simplelocalnet)
  - distributed-process-platform

Возможности библиотеки distributed-process

- удалённый запуск процессов;
- сериализация данных Haskell для передачи в сообщениях;
- связи между процессами (получение уведомлений при остановке другого процесса);
- получение сообщений по множественным каналам;
- выделенный канал (один на процесс) для получения динамически типизированных сообщений;
- автоматическое обнаружение узлов.

Основной пример: пинг-понг

- Один ведущий процесс и один дочерний процесс.
- Ведущий процесс создаёт дочерний.
- Процессы обмениваются сообщениями ping и pong (один раунд).
- Две версии:
  - на одной машине;
  - на отдельных узлах.

Модуль Control.Distributed.Process

```
data Process   -- экземпляр Monad, MonadIO

data NodeId    -- экземпляр Eq, Ord, Show, Typeable, Binary
data ProcessId -- экземпляр Eq, Ord, Show, Typeable, Binary

getSelfPid  :: Process ProcessId
getSelfNode :: Process NodeId

spawn  :: NodeId -> Closure (Process ()) -> Process ProcessId

send   :: Serializable a => ProcessId -> a -> Process ()
expect :: Serializable a => Process a
```

Сообщения и их сериализация

```haskell
data Message = Ping ProcessId | Pong ProcessId
  deriving (Typeable, Generic)

instance Binary Message
```

```haskell
class (Binary a, Typeable a) => Serializable a
instance (Binary a, Typeable a) => Serializable a
```

Серверный процесс

```
pingServer :: Process ()
pingServer = do
  Ping from <- expect
  say $ printf "ping received from %s" (show from)
  mypid <- getSelfPid
  send from (Pong mypid)
```

```
expect :: Serializable a => Process a
say :: String -> Process ()
getSelfPid  :: Process ProcessId
send :: (Serializable a) => ProcessId -> a -> Process ()
```

Создание таблицы удалённых вызовов

«Магия» Template Haskell

```
remotable ['pingServer]
```

Ведущий процесс

```
master :: Process ()
master = do
  node <- getSelfNode
  say $ printf "spawning on %s" (show node)
  pid <- spawn node $(mkStaticClosure 'pingServer)

  mypid <- getSelfPid
  say $ printf "sending ping to %s" (show pid)
  send pid (Ping mypid)

  Pong _ <- expect
  say "pong."

  terminate
```

Запуск узла

```
{-# LANGUAGE TemplateHaskell, DeriveDataTypeable,
             DeriveGeneric #-}
import Control.Distributed.Process
import Control.Distributed.Process.Closure

import Text.Printf
import Data.Binary
import Data.Typeable
import GHC.Generics (Generic)

main :: IO ()
main = do
      backend <- initializeBackend "localhost" "44444"
                                   (Main.__remoteTable initRemoteTable)
      startMaster backend (\_ -> master)
```

Результат запуска

```
$ ./ping
pid://localhost:44444:0:3: spawning on nid://localhost:44444:0
pid://localhost:44444:0:3: sending ping to pid://localhost:44444:0:4
pid://localhost:44444:0:4: ping received from pid://localhost:44444:0:3
pid://localhost:44444:0:3: pong.
```

Итоги примера

- Монада Process
- Вызовы spawn, expect, send
- Много технического кода (remotable, mkStaticClosure, initializeBackend)

Пинг-понг на нескольких узлах

Новый ведущий процесс

```
master :: [NodeId] -> Process ()
master peers = do

  ps <- forM peers $ \nid -> do
          say $ printf "spawning on %s" (show nid)
          spawn nid $(mkStaticClosure 'pingServer)

  mypid <- getSelfPid

  forM_ ps $ \pid -> do
    say $ printf "pinging %s" (show pid)
    send pid (Ping mypid)

  waitForPongs ps

  say "All pongs successfully received"
  terminate
```

Ожидание ответов

```
waitForPongs :: [ProcessId] -> Process ()
waitForPongs [ ] = return ()
waitForPongs ps = do
  m <- expect
  case m of
    Pong p -> waitForPongs (filter (/= p) ps)
    _  -> say "MASTER received ping" >> terminate
```

Узлы и новая функция main

- Несколько узлов = несколько запущенных программ (неважно, на одной машине или на нескольких).
- Программа должна знать, какой узел создаётся — ведущий или серверный (отвечающий).

```
main = do
  args <- getArgs
  let rtable = Main.__remoteTable initRemoteTable
  case args of
    [ "master" ] -> do
      backend <- initializeBackend defaultHost defaultPort rtable
      startMaster backend master
    [ "master", port ] -> do
      backend <- initializeBackend defaultHost port rtable
      startMaster backend master
    [ "slave" ] -> do
      backend <- initializeBackend defaultHost defaultPort rtable
      startSlave backend
    [ "slave", port ] -> do
      backend <- initializeBackend defaultHost port rtable
      startSlave backend
    [ "slave", host, port ] -> do
      backend <- initializeBackend host port rtable
      startSlave backend
```

Запуск

```
$ ./ping-multi slave 44445 &
[3] 58837
$ ./ping-multi slave 44446 &
[4] 58847
```

```
$ ./ping-multi master
pid://localhost:44444:0:3: spawning on nid://localhost:44445:0
pid://localhost:44444:0:3: spawning on nid://localhost:44446:0
pid://localhost:44444:0:3: pinging pid://localhost:44445:0:4
pid://localhost:44444:0:3: pinging pid://localhost:44446:0:4
pid://localhost:44446:0:4: ping received from pid://localhost:44444:0:3
pid://localhost:44445:0:4: ping received from pid://localhost:44444:0:3
pid://localhost:44444:0:3: All pongs successfully received
```

- Автоматическое обнаружение узлов.
- Ведущий процесс можно перезапустить — всё повторится.

Запуск на нескольких машинах

Эти команды выполняются на соответствующих машинах:

```
$ ./ping-multi slave 192.168.1.100 44444
$ ./ping-multi slave 192.168.1.101 44444
```

```
$ ./ping-multi master 44444
pid://localhost:44444:0:3: spawning on nid://192.168.1.100:44444:0
pid://localhost:44444:0:3: spawning on nid://192.168.1.101:44444:0
pid://localhost:44444:0:3: pinging pid://192.168.1.100:44444:0:5
pid://localhost:44444:0:3: pinging pid://192.168.1.101:44444:0:5
pid://192.168.1.100:44444:0:5: ping received from
                                    pid://localhost:44444:0:3
pid://192.168.1.101:44444:0:5: ping received from
                                    pid://localhost:44444:0:3
pid://localhost:44444:0:3: All pongs successfully received
```

Недостатки expect/send и типизированные каналы

- expect: поиск сообщения требуемого типа в очереди сообщений
- как отличать сообщения одного типа от разных отправителей?

```
data SendPort a      -- экземпляр Typeable, Binary
data ReceivePort a

newChan :: Serializable a => Process (SendPort a,
                                      ReceivePort a)

sendChan :: Serializable a => SendPort a -> a -> Process ()

receiveChan :: Serializable a => ReceivePort a -> Process a
```

Типизированные каналы: стиль взаимодействия

- клиент создаёт новый канал для взаимодействия;
- клиент посылает запрос вместе с SendPort созданного канала;
- сервер отвечает по полученному ранее SendPort;
- сервер может создать новый канал для взаимодействия с клиентом.

Пинг-понг на типизированных каналах

```haskell
data Message = Ping (SendPort ProcessId)
  deriving (Typeable, Generic)

instance Binary Message

pingServer :: Process ()
pingServer = do
  Ping chan <- expect
  say $ printf "ping received from %s" (show chan)
  mypid <- getSelfPid
  sendChan chan mypid
```

```
master :: [NodeId] -> Process ()
master peers = do
  ps <- forM peers $ \nid -> do
          say $ printf "spawning on %s" (show nid)
          spawn nid $(mkStaticClosure 'pingServer)
  ports <- forM ps $ \pid -> do
    say $ printf "pinging %s" (show pid)
    (sendport,recvport) <- newChan
    send pid (Ping sendport)
    return recvport
  forM_ ports $ \port -> do
     _ <- receiveChan port
     return ()
  say "All pongs successfully received"
  terminate
```

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`