使ってみよう Conduit

2012.2.5

山本和彦

Conduit とは何か？

- IO にまつわるコードを「生産者」と「消費者」に分離し、独立したコードとして開発し、後から合成する仕組み
- ピュアでのアナロジー
  - 悪しき一枚岩

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n x = x : replicate (n - 1) x
```

  - 合成

```
replicate n x = take n $ repeat x
```

- ファイルのコピー

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile src dst = runResourceT $
    CB.sourceFile src $$ CB.sinkFile dst
```

登場人物

- ResourceT
  - 最後の結果。runResourceT で走らせる
  - 生産者 $$ 消費者 → ResourceT
- Source
  - 生産者
  - Monoid のインスタンス
  - 生産者 ‘mappend‘ 生産者 → 生産者
- Sink
  - 消費者
  - Monad のインスタンス
  - 消費者 >>= 消費者 → 消費者
- Conduit
  - データ変換器
  - 変換器 =$ 消費者 → 消費者
  - 生産者 $= 変換器 → 生産者
  - 変換器 =$= 変換器 → 変換器

おまじない

```
{-# LANGUAGE OverloadedStrings #-}

module Sample where

import Control.Monad.IO.Class (liftIO)
import Data.ByteString (ByteString)
import qualified Data.ByteString as BS
import Data.ByteString.Char8 ()
import Data.Conduit
import qualified Data.Conduit.Binary as CB
import qualified Data.Conduit.List as CL
import Data.Monoid
```

消費者を作る

- 基本的な消費者から消費者を作る

```
consumer :: Sink ByteString IO ()
consumer = do
  mw <- CB.head
  case mw of
    Nothing -> return ()
    Just w  -> do
      liftIO . putStr $ "XXX "
      liftIO . BS.putStrLn . BS.singleton $ w
      consumer
```

- 走らせる

```
runResourceT $ CL.sourceNull $$ consumer
```

- sourceNull は mempty のこと

生産者を作る

- 基本的な関数を使う

```
listFeeder :: Source IO ByteString
listFeeder = CL.sourceList ["12", "34"]
```

- 走らせる

```
runResourceT $ listFeeder $$ consumer
```

生産者を増やす

- "5678" と書いてる "File"を用意
- 基本的な関数を使う

```
fileFeeder :: Source IO ByteString
fileFeeder = CB.sourceFile "File"
```

- 走らせる

```
runResourceT $ (listFeeder ‘mappend‘ fileFeeder)
            $$ consumer
```

消費者を増やす

- 基本的な消費者から消費者を作る

```
consumer2 :: Sink ByteString IO ()
consumer2 = do
  mw <- CB.head
  case mw of
    Nothing -> return ()
    Just w  -> do
      liftIO . putStr $ "YYY "
      liftIO . BS.putStrLn . BS.singleton $ w
    -- 再帰してないよ
```

- 走らせる

```
runResourceT $ (listFeeder ‘mappend‘ fileFeeder)
            $$ (consumer2 >> consumer)
```

変換器を使う

- 走らせる1

```
runResourceT $ listFeeder
            $$ CB.isolate 2 =$ consumer
```

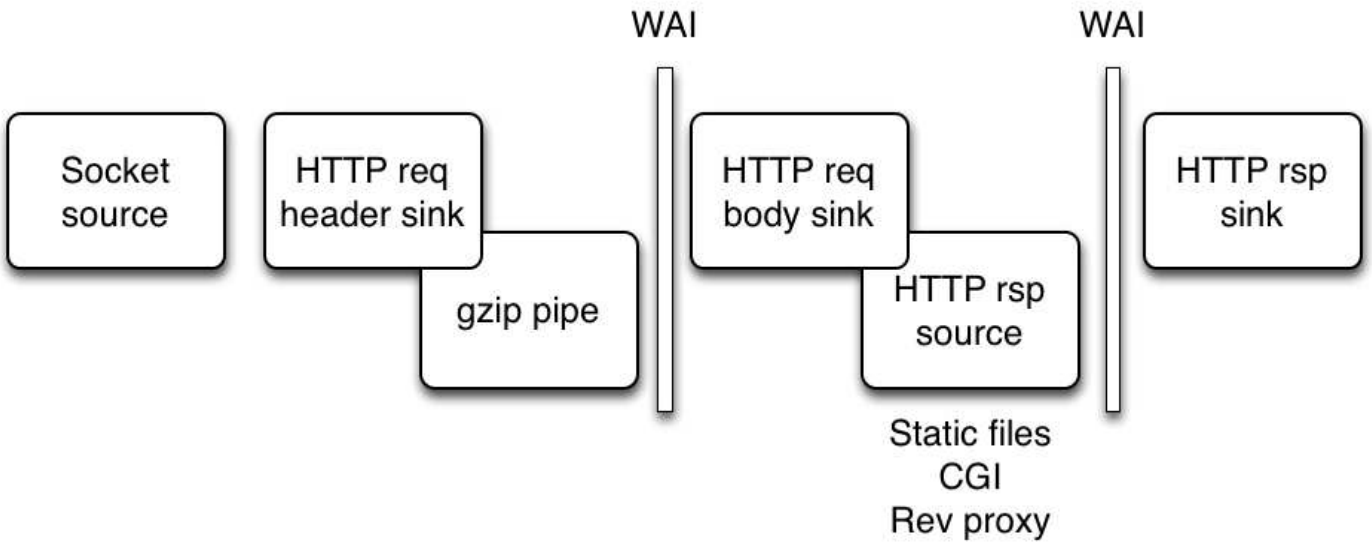
- 走らせる2

```
runResourceT $ listFeeder $= CB.isolate 2
            $$ consumer
```

歴史

- 第0世代：getContents
  - ○ 簡単
  - × 資源管理ができない
- 第1世代：Enumerator/Iteratee
  - ○ 生産者と消費者の分離
  - ○ 生産者の抽象化
  - ○ 生産者での資源管理
  - ○ 合成可能
  - × 消費者で資源管理ができない
  - × 例外処理が大変
  - × 生産者を引き回せない
  - × 分かりにくい
- 第2世代：Conduit
  - ○ 消費者の抽象化
  - ○ 消費者で資源管理ができる
  - ○ IO と同じ感覚の例外処理
  - ○ 生産者を引き回せる
  - ○ かなり簡単

Warp の例


WAI
WAI
Socket source
HTTP req header sink
gzip pipe
HTTP req body sink
HTTP rsp source
HTTP rsp sink
Static files
CGI
Rev proxy