

関数型言語と並行処理

Haskellでのネットワークプログラミング

2013.5.24



山本和彦



自己紹介

関数プログラミング

外部言語呼び出し

直和型

ファーストクラスIO

軽量スレッド

自己紹介

西の人です

- 山口県光市出身
 - 山口市は西の京を自称しているらしい
- 1988年 九州大学工学部情報工学科に進学
- B4のとき渡米しようとした
 - Sun 本社と国際電話で面接し、入社することになっていたが、就労ビザの審査が厳しくなり渡米できなくなった
 - 注：英語は苦手です
- 1992年 九州大学大学院に進学

修士時代は記事を書いて生活

- 「ハッピー・ネットワーキング」を執筆し配布
 - 後にアスキー出版局から出版された

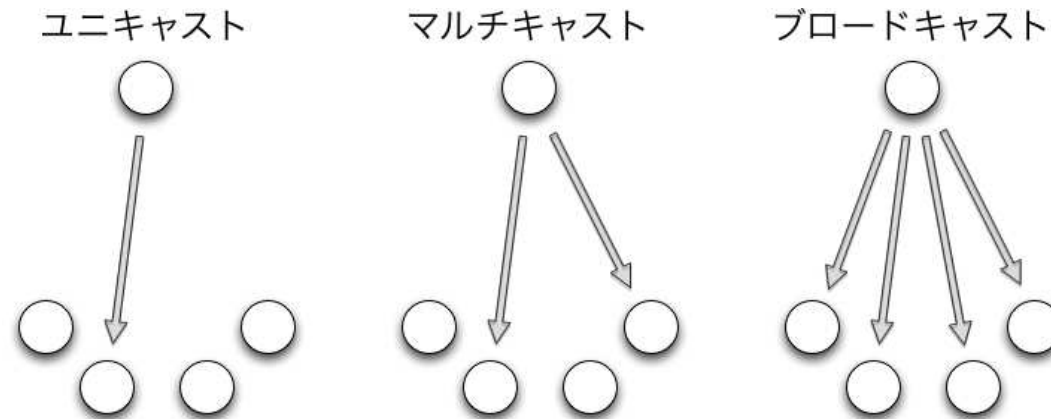


- 仕送りがなかったので記事を書いて生活
 - 「ハッピー・ネットワーキング」を Unix Magazine 編集長 見せて売り込む
- Unix Magazine
 - 「UNIXの知恵袋」を共同執筆
 - RFC の FYI (For Your Information) シリーズの訳

研究活動

■ IP マルチキャスト

- こちらは鳴かず飛ばす
- アプリがユニキャストでマルチキャストを実現する方法が主流



■ PEM (Privacy Enhanced Mail) の実験に参加

- WIDE Project
- mh-e を改造して PEM を扱えるようにした
- メールのユーザインターフェイスに興味を持つ

奈良先端大へ

- お世話になった先生が奈良先端大へ
- 修了後 奈良先端大の助手になる
 - 試験のない唯一の国家公務員？
- 京都にはときどき遊びに来てました
 - 鈴虫寺でなぞの黒い物体を食べさせられたり

物書きのはしくれ

- 「転ばぬ先のセキュリティ」
 - Unix Magazine
 - PGP も紹介
- オライリー 「PGP：暗号メールと電子署名」 監訳
 - オライリー・ジャパンの最初の本
 - 研究費として寄付して頂いたので、個人的には1円ももらってません。あんなに売れたのに。。



Mew の開発



- マルチメディアメール (MIME) の黎明期
 - 「添付」という考えさえよく分からなかった時代
 - 書式を統一して混乱に終止符を打つ
 - マルチメディア
 - 文字コード
 - 暗号、電子署名
- Mew
 - たくさんの文字コードをサポート
 - 画像さえも暗号化したり電子署名を付けたりできる
- 複数のEmacs のサポート
 - Emacs の複数のバージョン
 - XEmacs の複数のバージョン

ストールマンと僕と Emacs

- Mew では自家製のパスワード読み込み関数を実装
 - エコー off

Emacs にパスワードを読み込む関数を入れて下さい



プライバシーに対する脅威はあなたの身内や友達からやってきます



- ストールマンとの交渉は面倒
 - Mew を Emacs に入れてもらうことを諦めた
- その後 read-passwd が実装された
 - gnu.org がクラックされたため

IIJ へ転職

- 東京への転職先を探す
 - KAME プロジェクトを立ち上げるため

IIJ は、いつ研究所を作るんですか？



おまえが来るなら作る。

歌代
さん

- IIJ へ転職
 - 「国家公務員を辞めるなんてバカだ」と言われたが、その後大学も法人化

KAME プロジェクト

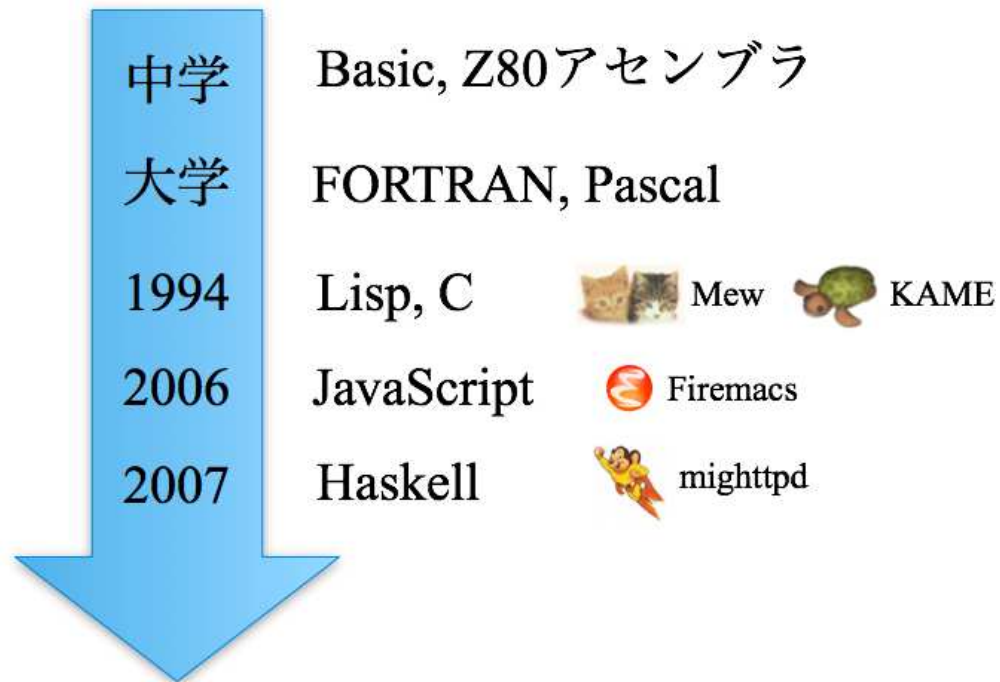


- KAME
 - WIDE プロジェクトのサブプロジェクト
 - BSD 用の IPv6、IPsec、Mobile IPv6 のコードを実装
 - 各企業のエースを集める
 - IJ、NEC、東芝、日立、富士通、横河電機
- 成果物
 - BSD に IPv6、IPsec、Mobile IPv6 のコードがマージされた
 - Mac で IPv6 を使えるのは KAME プロジェクトのおかげ

IETF と私

- Internet Engineering Task Force
 - インターネットのプロトコルを標準化する
 - 成果物は RFC (Request For Comments) として公開
 - RFC は 国際標準化機構 (ISO) に配慮した名前
 - 誰でも参加できる
 - メーリングリストと年三回の会合で議論
- ISO の OSI (Open Systems Interconnection) と戦っていたころの理念
 - We reject: kings, presidents and voting.
 - We believe in: rough consensus and running code.
- メールと IPv6 の標準化のため僕も参加
- IETF も闘争の場
 - ピュアな技術者として参加すると振り返りにある

プログラミング言語歴



- 2006 年頃から新たな言語、新たな考え方を学び始めた
- オブジェクト指向もかなり勉強したが、しっくりこず、関数プログラミングへ

JavaScript

- JavaScript の勉強がてらに Firemacs を作った
- Douglas Crockford の文章をたくさん読む
 - 「JavaScript: 世界で最も誤解されているプログラミング言語」
波括弧や不格好な for 文がある
JavaScript の C ライクな文法を見ると、
通常の命令型言語のように思える。
 - 「JavaScript: The Good Parts」
ある人 「JSLint は switch のフォールスルーを警告すべき」
Crockford 「メリットの方が大きい」
ある人 「JSLint にはバグがあります」
Crockford 「フォールスルーが原因だった！」 → 悟り
- どちらも関数プログラミングでは当たり前だった
 - for 文は後述
 - 制御構造は木構造であるべき

山本和彦のジレンマ

山本和彦が Emacs Lisp は知っていたが
Haskell は知らなかったときの話：



Lisp は関数型言語だと言われている



Emacs Lisp も Lisp である



山本和彦は関数型プログラマーだと思われていた



しかし、本人は何が関数プログラミングなのか分かっていなかった

なぜ Haskell を始めたか

Q1) Haskell では破壊的代入がないのに
プログラミングできるの？

- A) 関数プログラミングのパラダイムを理解すればできます
- A) ただし破壊的代入もあります

Q2) Haskell で Web サーバなんか作れるの？

- Marlow 先生の論文
 - 「Developing a high-performance web server in Concurrent Haskell」
 - どこにコードがあるのか、どうやってインストールするのかさえ分からず

A) IO という型を使って入出力ができます

A) Haskell は「世界で最高の命令型言語」でもあります
(c) Simon Peyton Jones

	自己紹介
→	関数プログラミング
	外部言語呼び出し
	直和型
	ファーストクラスIO
	軽量スレッド

関数プログラミング

パラダイムの違い

命令プログラミング

命令を列挙する

A; B; C;

状態がある

破壊的代入を使う

関数プログラミング

関数を引数に
適用する

状態はない
(引数で渡す)

(値を破壊したくなったら)
新たな値を作る

例題

- 入力として整数のリストあるいは配列
{ 10, 20, 30, 40, 50 } がある
- 0 から数えて n 番目の要素には n を掛ける
- それらをすべて足し合わせる

- つまり、以下のような計算をする

$$10 * 0 + 20 * 1 + 30 * 2 + 40 * 3 + 50 * 4 = 400$$

$$10 * 0 + 20 * 1 + 30 * 2 + 40 * 3 + 50 * 4 = 400$$

命令プログラマなら
for 文か類似のループで
問題を解く

不格好な for 文

- Douglas Crockford のエッセイ
「JavaScript: 世界で最も誤解されているプログラミング言語」
<http://www.crockford.com/javascript/javascript.html>

波括弧や不格好な for 文がある
JavaScript の C ライクな文法を見ると、
通常の命令型言語のように思える。

これは誤解を与えやすい。
なぜなら、JavaScript は、
C や Java とよりも
関数型言語 Lisp や Scheme との方が
共通点が多いからだ。

for 文って不格好なの？



for の秘密

- 山本和彦は、for について授業で熱く語っていた
- for の意味
 - for は期間の for だ！
 - 例) for two days
- 非対称範囲を使え！
 - `for (i = 0; i < N; i++) { }`
 - 左の境界は入るが、右の境界は入らない
 - 0 から始めると配列と相性がよい
 - N をそのまま使える
 - 個数 = 最後 - 最初 + 1
 - $(N - 1) - 0 + 1 = N$
 - 不等号を使うと安全性
 - コンピュータでは、 $1/3 + 1/3 + 1/3 \neq 1$

Java で不格好な for

```
public static int func(int[] ar) {  
    int ret = 0;  
    for (int i = 0; i < ar.length; i++) {  
        ret = ret + ar[i] * i;  
    }  
    return ret;  
}
```

```
int[] inp = {10, 20, 30, 40, 50};  
func(inp);  
→ 400
```


Haskell で MapReduce

```
zip [0..] [10,20,30,40,50]  
→ [(0,10), (1,20), (2,30), (3,40), (4,50)]
```

```
map (\(i,x) -> x*i) 上の式  
→ [0,20,60,120,200]
```

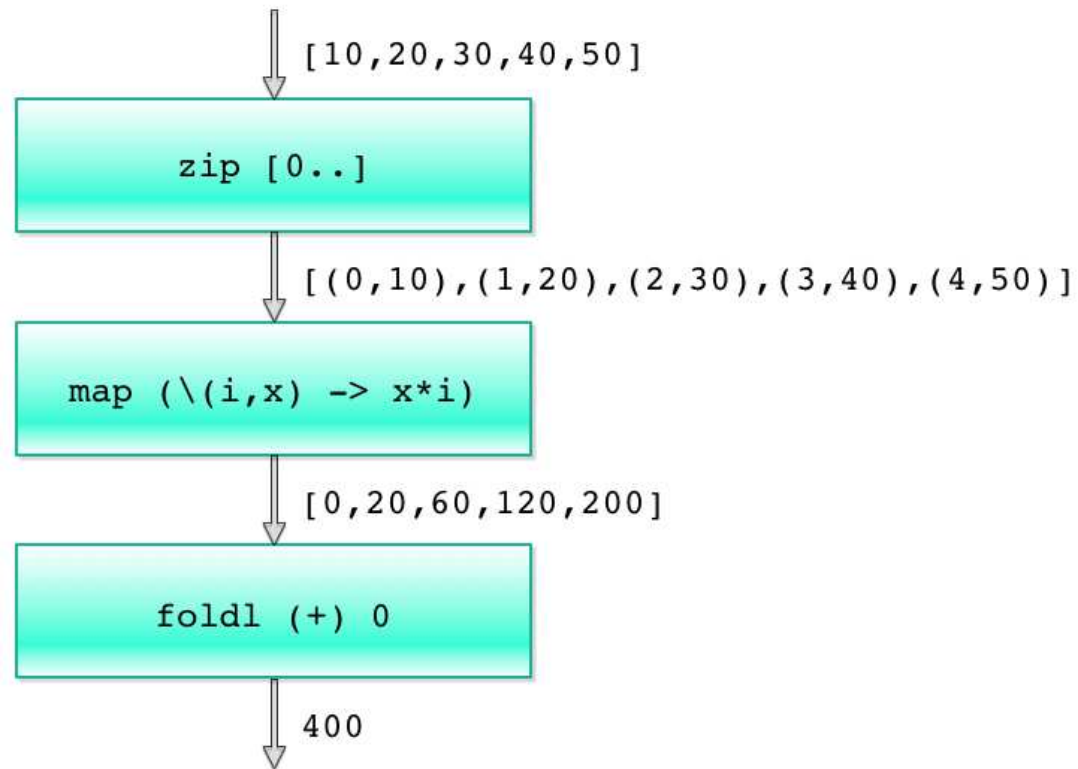
```
foldl (+) 0 上の式  
→ (((0 + 0) + 20) + 60) + 120) + 200  
→ 400
```

■ 関数を合成する

```
func = foldl (+) 0  
      . map (\(i,x) -> x*i)  
      . zip [0..]
```

```
func [10,20,30,40,50]  
→ 400
```

部品プログラミング



関数プログラミングの極意は
バグの入り込みにくい
小さな関数をつなげて
大きな関数を作ること



関数プログラミングと変数

- 再代入可能な変数は、ほとんど使わない
 - 状態は引数として渡す
 - 値を破壊したくなったら、新たに作り出す
 - これだけでも多くの問題は解ける

- 再代入可能な型も用意されている
 - 参照型
 - Haskell -- IORef
 - OCaml -- ref
 - 副作用のある関数の中で使う
 - 参照型も引数として渡す
 - グローバルな参照型は、緊急時にしか使わない

関数の分類

純粋な関数

式で構成する
計算をする
引数と同じなら同じ
結果を返す

副作用のある関数

文相当の式で構成する
副作用を起こす
引数と同じでも異なる
結果を返すこともある

一方向性

- 副作用のある関数から純粋な関数は呼び出してよい
- 純粋な関数から副作用のある関数は呼び出してはならない
 - 呼び出すと副作用のある関数になってしまうから

純粋関数型言語

言語が一方向性を要求

黒魔術としての抜け道はあり

Haskell

関数型言語

言語は一方向性を要求しない

F#, OCaml, Scala, SML

Haskell では型で純粋か分かる

■ 純粋な関数

```
(+) :: Num a => a -> a -> a  
sort :: Ord a => [a] -> [a]  
filter :: (a -> Bool) -> [a] -> [a]
```

■ 副作用がある関数

```
hGetLine :: Handle -> IO String  
writeFile :: FilePath -> String -> IO ()  
writeIORef :: IORef a -> a -> IO ()
```

- IO から純粋な関数は呼び出せる
- 純粋な関数から IO は呼び出せない

OCaml では型から純粋か分からない

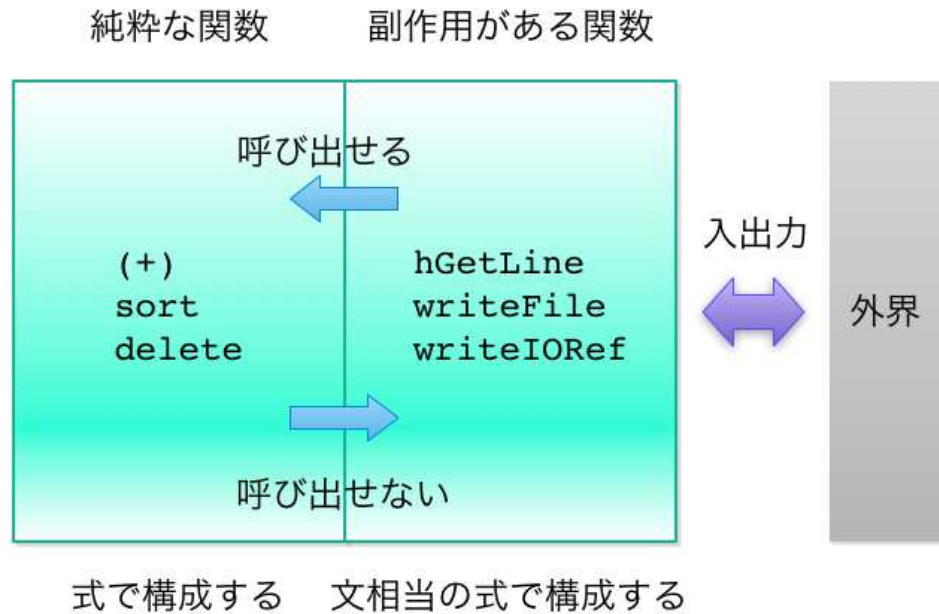
■ 純粋な関数

```
(+) : int -> int -> int  
sort : ('a -> 'a -> int) -> 'a list -> 'a list  
filter : ('a -> bool) -> 'a list -> 'a list
```

■ 副作用がある関数

```
input_line : in_channel -> string  
write : file_descr -> string -> int -> int -> int  
(:=) : 'a ref -> 'a -> unit
```


これまでのまとめ



	自己紹介
	関数プログラミング
→	外部言語呼び出し
	直和型
	ファーストクラスIO
	軽量スレッド

外部言語呼び出し

FFI(Foreign Function Interface)

- Haskell の関数から C の関数を呼び出す
 - さらに、その C の関数から Haskell の関数を呼び出し可能
 - データを変換するための機能が充実
- ブロックする C の関数も呼び出せる
 - ブロックする場合：safe 呼び出し
 - 状態を保存しネイティブスレッドを切り出す
 - ブロックしない場合：unsafe 呼び出し
 - 高速
- メモリ管理を型で区別
 - Haskell の GC が回収する：Ptr a
 - 特定のファイナライザが解放する：ForeignPtr a

純粋な関数への FFI

- すでに型が用意されているなら
foreign 宣言をするだけ

```
{-# LANGUAGE ForeignFunctionInterface #-}  
import Foreign.C.Types  
  
foreign import ccall unsafe  
    "sin" c_sin :: CDouble -> CDouble  
  
-- realToFrac は Double と CDouble を変換  
mySin :: Double -> Double  
mySin x = realToFrac (c_sin (realToFrac x))
```

副作用のある関数への FFI

- 副作用がある場合は型に IO を付ける

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign
import Foreign.C.String
import Foreign.C.Types

-- time は呼び出すごとに変わる
foreign import ccall unsafe
    "time" c_time :: Ptr CTime -> IO CTime

-- 静的な文字列を使うので副作用がある
foreign import ccall unsafe
    "ctime" c_ctime :: Ptr CTime -> IO CString

timeString :: IO String
timeString = alloca $ \ctime_ptr -> do -- 一時割り当て
    _ <- c_time ctime_ptr
    str <- c_ctime ctime_ptr -- str は静的な文字列
    peekCString str -- CString を String へ
```

環境の違いを埋める .hsc

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Data.UnixTime.Types
import Foreign.C.Error
import Foreign.C.Types
import Foreign.Marshal.Alloc
import Foreign.Ptr
import Foreign.Storable

#include <time.h>
#include <sys/time.h>

type CTimeVal = ()
type CTimeZone = ()

foreign import ccall unsafe "gettimeofday"
    gettimeofday :: Ptr CTimeVal -> Ptr CTimeZone -> IO CInt

getUnixTime :: IO UnixTime
getUnixTime =
    allocaBytes (#const sizeof(struct timeval)) $ \ p_timeval -> do
        gettimeofday p_timeval nullPtr
        sec <- (#peek struct timeval,tv_sec) p_timeval
        usec <- (#peek struct timeval,tv_usec) p_timeval
        return $ UnixTime sec usec
```

hsc2hs が作成した .hs

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Data.UnixTime.Types
import Foreign.C.Error
import Foreign.C.Types
import Foreign.Marshal.Alloc
import Foreign.Ptr
import Foreign.Storable

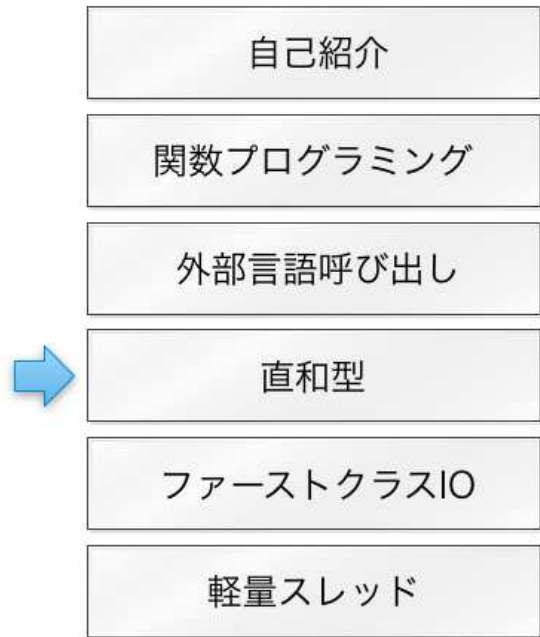
type CTimeVal = ()
type CTimeZone = ()

foreign import ccall unsafe "gettimeofday"
    c_gettimeofday :: Ptr CTimeVal -> Ptr CTimeZone -> IO CInt

getUnixTime :: IO UnixTime
getUnixTime =
    allocaBytes (8) $ \ p_timeval -> do
        c_gettimeofday p_timeval nullPtr
        sec <- ((\hsc_ptr -> peekByteOff hsc_ptr 0)) p_timeval
        usec <- ((\hsc_ptr -> peekByteOff hsc_ptr 4)) p_timeval
        return $ UnixTime sec usec
```

入出力が扱えて
C の関数が呼び出せるから
なんでも実装できる





直和型

Java でお母さんDB

```
public class Person {
    String name;
    int age;
}

HashMap<String, Person> db
    = new HashMap<String, Person>();

Person sazae = new Person("サザエ", 24);
Person fune  = new Person("フネ", 52);

map.put("タラオ", sazae);
map.put("サザエ", fune);
// "フネ" のお母さんは亡くなっている
```

おばあちゃんを探せ

```
String me = "タラオ";
Person mother = db.get(me);           // "サザエ"
Person gramma = db.get(mother.name); // "フネ"

String me = "サザエ";
Person mother = db.get(me);           // "フネ"
Person gramma = db.get(mother.name); // null

String me = "フネ";
Person mother = db.get(me);           // null
Person gramma = db.get(mother.name); // 例外
```

安全におばあちゃんを探せ

```
String me = "フネ";  
Person mother = db.get(me);  
Person gramma = null;  
if (mother != null) {  
    gramma = db.get(mother.name);  
}
```

null の問題点

<Person> get (<String>)

- null は、いろんな型になれる
- 型から null を返すか分からない
 - 返すかもしれないし
 - 返さないかもしれない
- Java プログラマは必ず null を踏む
 - コンパイラの支援がないので null の処理を忘れる



Tony Hoare

nullは
10億ドルの失敗

Haskell でお母さんDB

```
data Person = Person String Int

name :: Person -> String
name (Person n _) = n

age :: Person -> Int
age (Person _ a) = a

db :: [(String, Person)]
db = [("タラオ", Person "サザエ" 24)
      , ("サザエ", Person "フネ" 52)]
```

lookup と Maybe

- DB の検索関数

```
lookup :: String  
      -> [(String, Person)]  
      -> Maybe Person
```

- Person は、失敗しない型

- Maybe Person は、失敗するかもしれない型

- 答えなし: Nothing
- 答えあり: Just (Person "サザエ" 24)

安全におばあちゃんを探せ

- Maybe Person はパターンマッチで処理
 - case ... of 式
 - すべての場合を網羅しないとコンパイラが警告を出す

```
findGramma :: String -> Maybe Person
findGramma me = case lookup me db of
  Nothing      -> Nothing
  Just mother  -> lookup (name mother) db
```

関数の振る舞いは
適切な型で表現しよう



安全にひいおばあちゃんを探せ

■ おばあちゃんを探せ (Java 版再掲)

```
String me = "フネ";
Person mother = db.get(me);
Person gramma = null;
if (mother != null) {
    gramma = db.get(mother.name);
}
```

■ ひいおばあちゃんを探せ

```
String me = "フネ";
Person mother = db.get(me);
Person gramma = null;
Person ggramma = null;
if (mother != null) {
    gramma = db.get(mother.name);
    if (gramma != null) {
        ggramma = db.get(gramma.name);
    }
}
```

安全にひいおばあちゃんを探せ

■ おばあちゃんを探せ(Haskell 版再掲)

```
findGramma :: String -> Maybe Person
findGramma me = case lookup me db of
  Nothing      -> Nothing
  Just mother  -> lookup (name mother) db
```

■ ひいおばあちゃんを探せ

```
findGGramma :: String -> Maybe Person
findGGramma me = case lookup me db of
  Nothing      -> Nothing
  Just mother  -> case lookup (name mother) db of
    Nothing      -> Nothing
    Just gramma  -> lookup (name gramma) db
```

入れ子はイヤ！
エラー処理にロジックが
埋もれるのもイヤ！



Maybe とは何なのか？

- Maybe は「直和型」の一種
 - Java の列挙型は直和型の一種
- 直和型
 - 「または」を表す型

- 真理値

```
data Bool = False | True
```

- 成功に値がある

```
data Maybe a = Nothing | Just a
```

- 成功にも失敗にも値がある

```
data Either l r = Left l | Right r
```

演算の直列化

- 真理値は直列にできる

```
isAlive me && isAlive mother
```

- Maybe a も直列にできないか？

```
findMother me &&> findMother mother
```

地下配線

■ 地下配線 &&>

```
mx &&> f = case mx of
  Nothing -> Nothing
  Just x   -> f x
```

■ 直列版

```
findGGamma :: String -> Maybe Person
findGGamma me =
  lookup me db
  &&> \mother -> lookup (name mother) db
  &&> \gramma -> lookup (name gramma) db
```

■ 入れ子版(再掲)

```
findGGamma :: String -> Maybe Person
findGGamma me = case lookup me db of
  Nothing      -> Nothing
  Just mother  -> case lookup (name mother) db of
    Nothing      -> Nothing
    Just gramma  -> lookup (name gramma) db
```


エラー処理は
地下配線に押し込めよう



自己紹介

関数プログラミング

外部言語呼び出し

直和型



ファーストクラスIO

軽量スレッド

ファーストクラスIO

Web サーバでの静的ファイルの処理

- ファイル処理
 - パス末尾が "/" なら
 - "パス/index.html.ja", "パス/index.html.en", "パス/index.html" の順に調べて存在したファイルを返す (200 OK)
 - そうでないなら
 - パスのファイルが存在すれば返す (200 OK)
- 向け直し処理
 - パスの末尾が "/" でないなら
 - "パス/index.html.ja", "パス/index.html.en", "パス/index.html" の順に調べてファイルが存在すれば "パス/" へ向け直し (MovedPermanently 301)
- デフォルト処理
 - NotFound (404) を返す

お題：これを部品プログラミングで
実装するにはどうする？

IO は値である

コード

```
ios :: [IO ()]
ios = [ putStrLn "Hello!", putStrLn "Hi!" ]

main :: IO ()
main = do
    ios !! 1
    ios !! 0
```

出力

```
Hi!
Hello!
```

注意： Hi! の方が先に出力されている

IO とは何か？

- 近似の理解
 - 命令プログラマ or Haskell初心者向け
 - IO とは副作用を表す型
 - `getChar :: IO Char` は、一文字読み込む命令
 - 呼び出すごとに違う文字を返す
- 本当の理解
 - IO とは命令書を表す型
 - `getChar :: IO Char` は、一文字読み込めという命令書
 - 評価されると常に同じ命令書が返る
 - だから「純粋」と呼ばれる
 - `main` からつらなる IO の文脈に取り出されると命令書が破られて命令が実行される
 - 命令を実行するのはランタイム

OCaml で Haskell の IO を模倣する

- io は unit を取って 'a を返す関数
 - 関数はファーストクラスだから io もファーストクラス

```
type world = unit
type 'a io = world -> 'a

(* write : string -> unit io *)
let write str = fun () -> print_string str

(* read : string io *)
let read = read_line

(* run : a' io -> 'a *)
let run prog = prog ()

(* (>>=) : 'a io -> ('a -> 'b io) -> 'b io *)
let (>>=) io f = fun () -> let res = run io
                             in run (f res);;

run (read >>= write);;
```

命令書を作れ

- ファイルのリスト
 - `files :: [FilePath]`
 - 例 ["パス/index.html.ja", "パス/index.html.en", "パス/index.html"]
 - HTTP 要求から作る
- 引数のファイルを HTTP 応答で返す関数
 - `sendFile :: FilePath -> IO ()`
- 命令書のリストを作る
 - `ios :: [IO ()]`
 - `ios = map sendFile files`
- このうち有効な最初の命令書が処理されればよい

例外の活用

- ファイルがなければ例外が起きる
- ある処理で例外が起きたら
次の処理をすれば部品化できる？

例外のジレンマ

- Java で実現すると入れ子になる
 - try ~ catch は構文

```
try {  
    処理1;  
} catch (...) {  
    try {  
        処理2;  
    } catch (...) {  
        処理3;  
    }  
}
```

Haskell での例外処理

- Haskell の catch は単なる関数

- `catch :: IO a -> (IOError -> IO a) -> IO a`

- 無理矢理 Java 風版

```
catch (  
  処理1  
) (\_ ->  
  catch (  
    処理2  
  ) (\_ ->  
    処理3  
  )  
)
```

- 注：説明を簡単にするため古い catch を使っています

IO の直列化

■ 地下配線 `||>`

```
(||>) :: IO a -> IO a -> IO a
x ||> y = catch x (\_ -> y)
```

■ 直列版

```
処理1 ||> 処理2 ||> 処理3
```

■ 無理矢理 Java 風版(再掲)

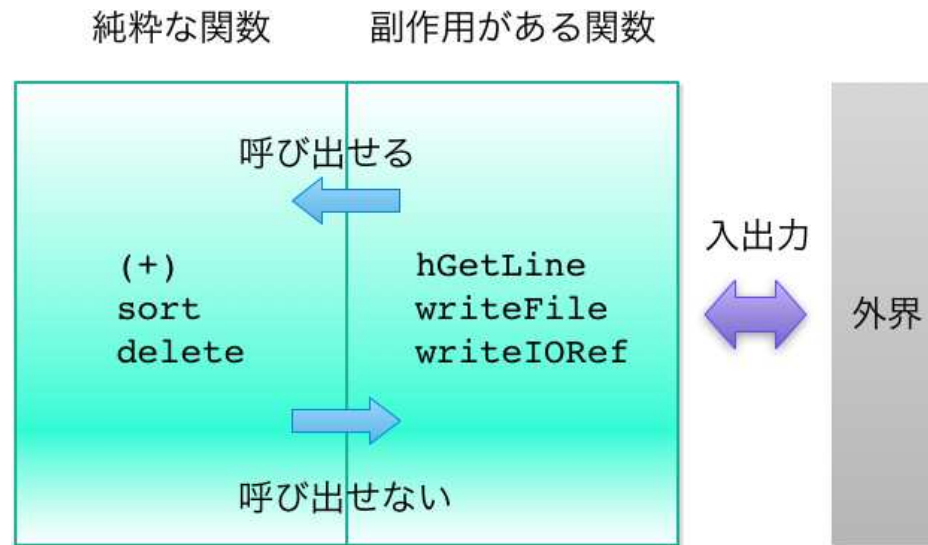
```
catch (
  処理1
) (\_ ->
  catch (
    処理2
  ) (\_ ->
    処理3
  )
)
```

命令書のリストを直列化

- これまでに作った命令書のリスト
 - `ios :: [IO ()]`
 - `ios = map sendFile files`
- `||>` でつなぐには畳み込む
 - `foldr1 (||>) ios`

直和型と例外

- 純粋な関数は全域関数であるべき
 - 失敗は直和型で表す
- 副作用のある関数は例外を投げてよい



式で構成する 文相当の式で構成する
失敗は直和型で表現 例外を投げてよい

自己紹介

関数プログラミング

外部言語呼び出し

直和型

ファーストクラスIO



軽量スレッド

軽量スレッド

並列と並行

並列
Parallel

一つの仕事を複数に分割して
それぞれを同時にこなすこと

例：シュミレーション

並行
Concurrent

複数の仕事同時にこなすこと

例：Web アプリ

注：「疑似並列が並行だ」という人もいます

破壊的な代入が制限されている
関数型言語は並行/並列処理に
向くと言われていました

しかし、すべての関数型言語が
並行/並列処理に取り組んでいる
訳ではありません

Haskell は並行/並列処理に
取り組んでいます
僕は並列処理のことはよく分かりません

今日は
Haskell の並行(concurrent)
の話



アーキテクチャの比較

コードの見通し

ネイティブ・スレッド	○	×	○
イベント駆動	×	○	×
1コア1プロセス マッピング	×	○	○
軽量スレッド	○	○	○

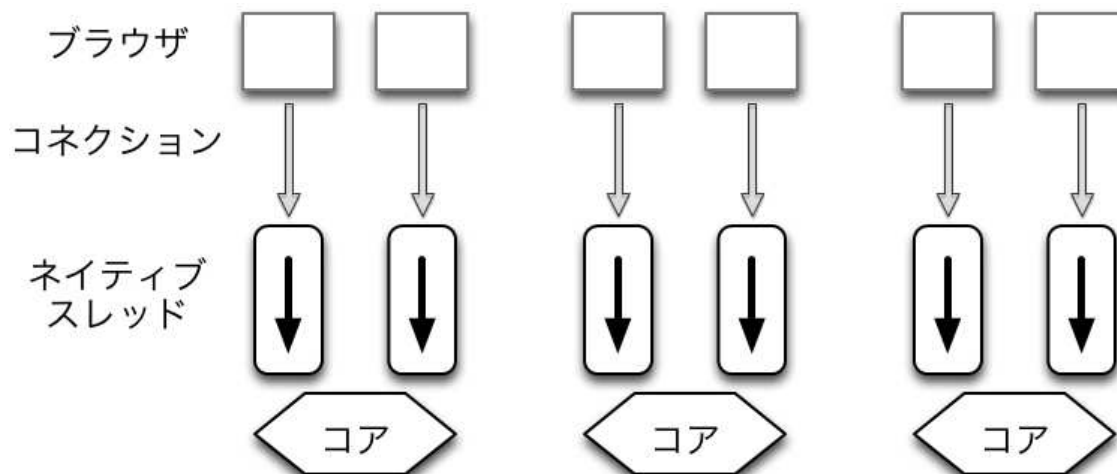
コア当たりの性能

マルチコアの活用

- 今日是最後の軽量スレッドを改良する話

ネイティブスレッド

- コネクションごとにネイティブスレッドを生成する
 - 古くはプロセスを生成する
 - 生成の仕方にも種類がある
 - 新しいコネクションが来る度に生成
 - あらかじめ生成しておいて使い回す (プール)
 - コードの見通し ○
 - 制御を占有できる
 - コア当たりの性能 ×
 - ネイティブスレッド/プロセスのコンテキストスイッチが多い
 - マルチコアの活用 ○



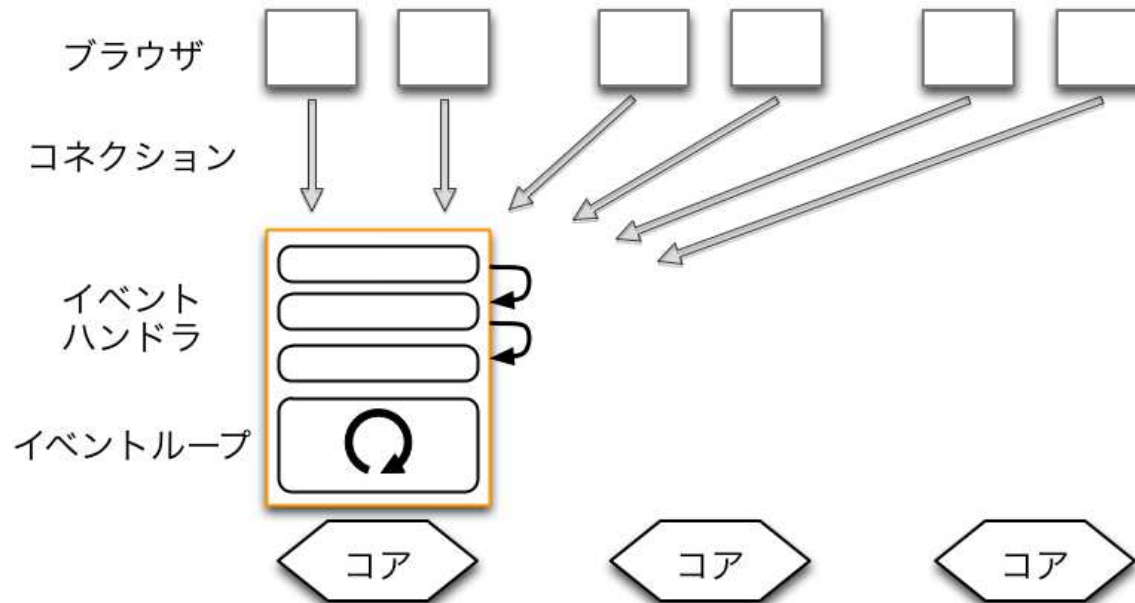
プロセスとスレッド



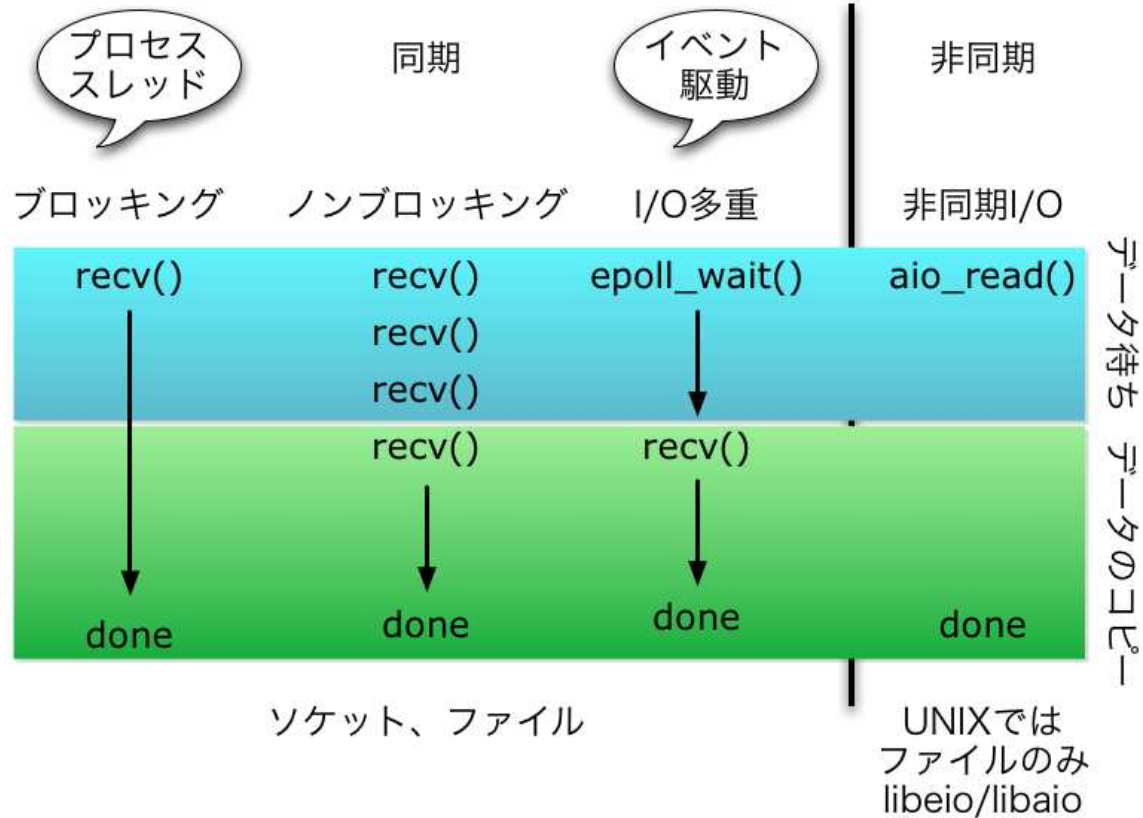
軽量 = 10万個上げても大丈夫

イベント駆動

- epoll/kqueue を使ったイベントループがハンドラ(コールバック)を駆動する
 - コードの見通し ×
 - コールバック地獄：コードがぶつ切りになる
 - コア当たりの性能 ○
 - マルチコアの活用 ×



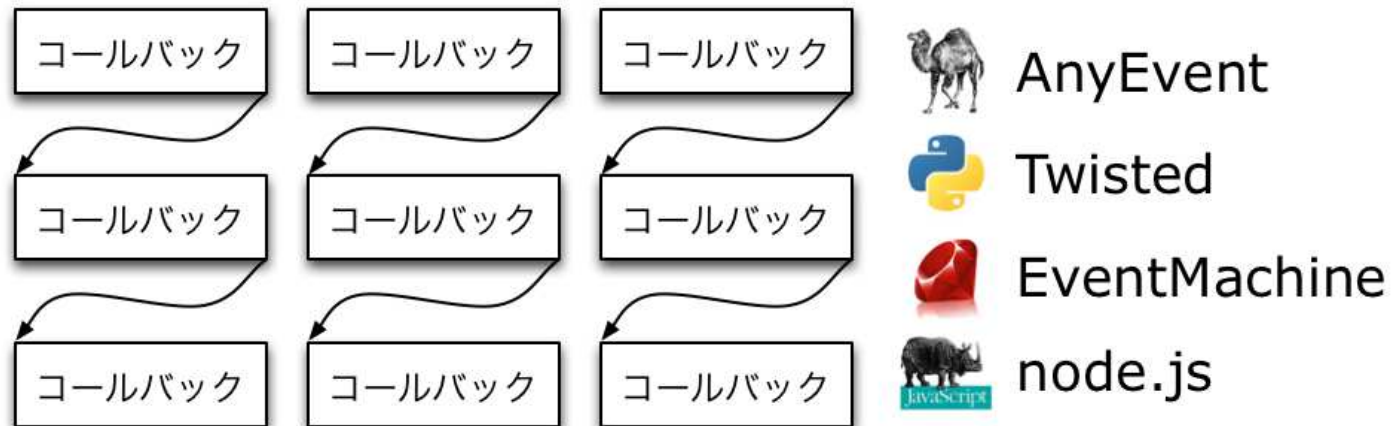
ブロッキングとI/O多重



- I/O多重は同期だが、上位層からは非同期に見える

コールバック

■ イベント駆動をプログラマに見せる



- 各種コールバック(ハンドラ)を登録する
 - あるいはコールバックを渡して行く (継続渡しスタイル)
- コードがぶつ切りとなり見通しが悪い
- コールバック地獄が発生する

コールバック地獄

- 通常のスタイル (同期)

```
var x = f();  
var y = g(x);  
var z = h(y);
```

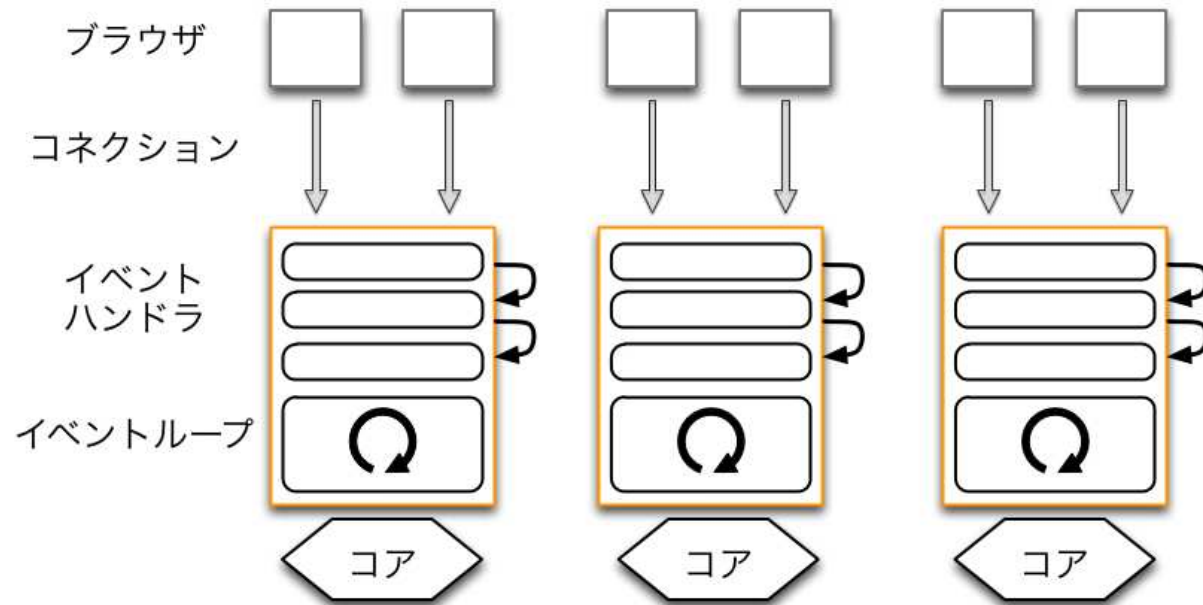
- コールバック・スタイル (非同期)

```
f(function(x) {  
  g(x,function(y) {  
    h(y,function(z) {  
      ...  
    });  
  });  
});
```

- 工夫はできるが、やはり見通しが悪い
- コールバックごとの例外処理もつきまとう

1コア1プロセス・マッピング

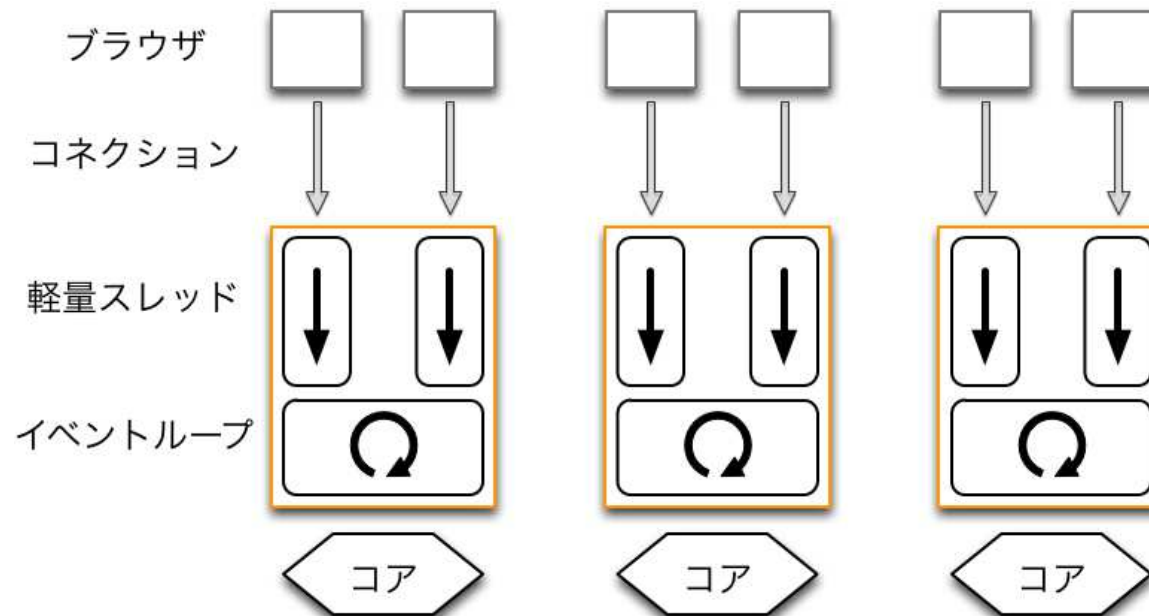
- コアごとにプロセスを起動し、ポートを共有する
 - コードの見通し ×
 - コア当たりの性能 ○
 - マルチコアの活用 ○



軽量スレッド

■ 1コア1プロセス・マッピングに、さらに 軽量スレッドを組み合わせる

- コードの見通し ○
 - 制御を占有できる
- コア当たりの性能 ○
- マルチコアの活用 ○



本当は難しい軽量スレッド

標準ライブラリ

標準ライブラリがブロックする

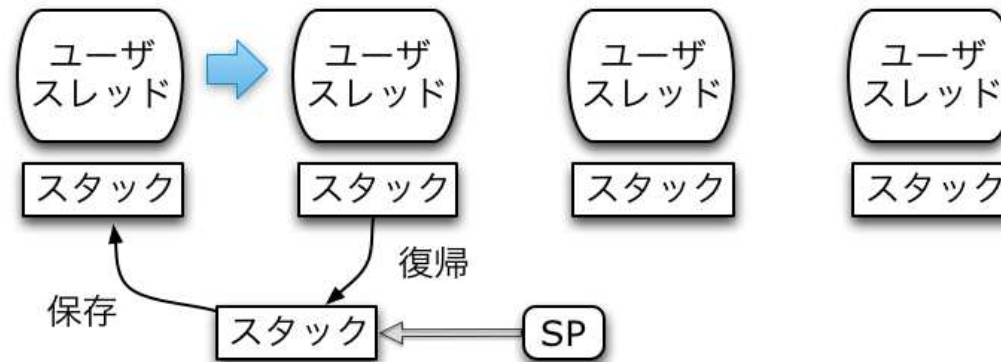
過去の資産のしがらみ

外部言語呼び出し

呼び出した外部言語がブロックする

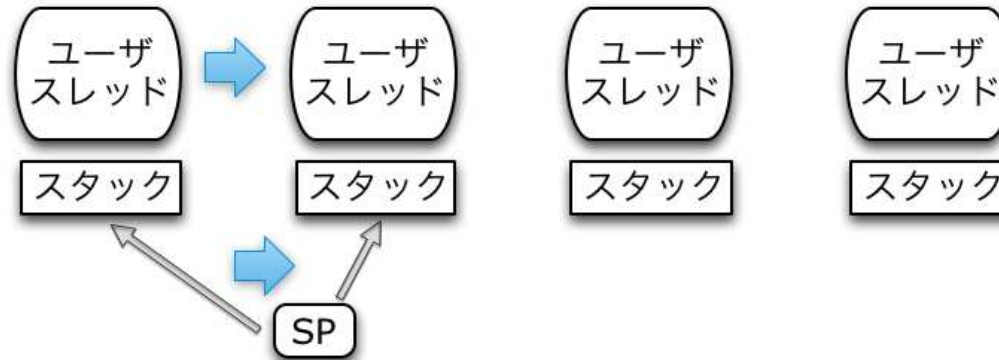
ユーザスレッドがうまくいかなくなる最大の原因

Ruby の場合



- Ruby のユーザスレッドは重かった
 - ユーザスレッドのスタックを
コンテキストスイッチの際にコピー
 - スタックポインタの差し替えにしないのは、
マシン依存にしないため
- Ruby 1.9 からネイティブスレッドへ移行した

Haskell の場合



■ 軽量スレッド

- スタックにはスタックポイントではないレジスタを使う
- コンテキストスイッチの際は、そのレジスタの差し替えで OK
- 小さいスタックから始めて、必要に応じて大きくできる

標準ライブラリは
ブロックしない

ブロックする外部言語
呼び出しはネイティブ
スレッドに任せる

アーキテクチャ比較(再掲)

コードの見通し ↙

ネイティブ・スレッド	○	×	○
イベント駆動	×	○	×
1コア1プロセス マッピング	×	○	○
軽量スレッド	○	○	○

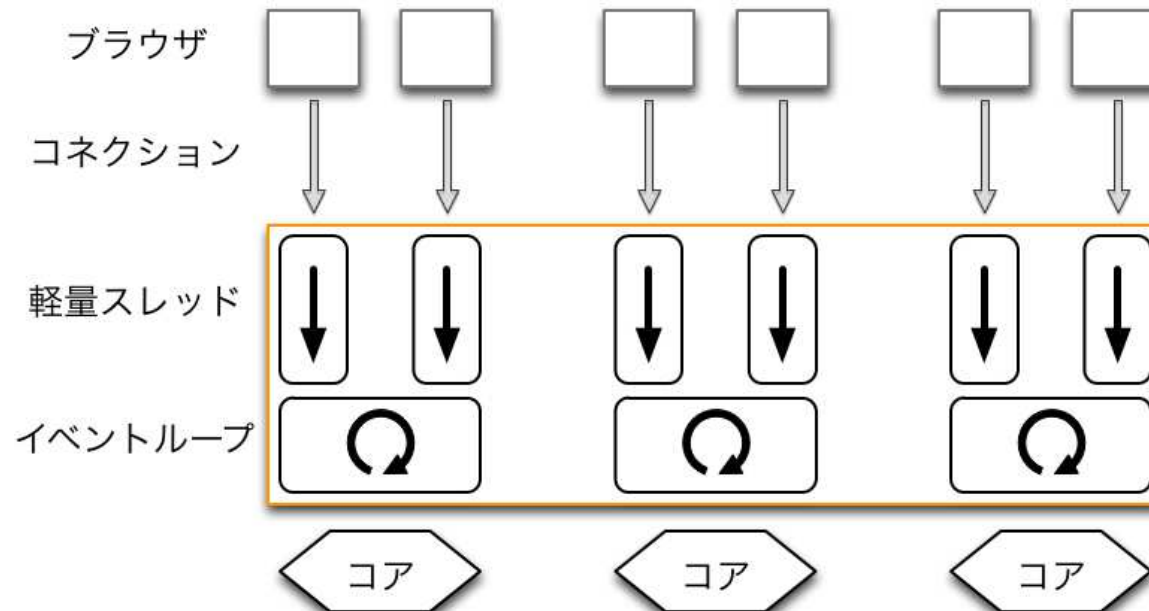
コア当たりの性能 ↗
マルチコアの活用 ↗

軽量スレッド+プロセス/コアの問題点

- 定型コードの再発明
 - コアごとにプロセスを起動し
ポートを共有するコードを毎回書かないといけない
- 共有資源の欠如
 - たとえば設定ファイルをすべてのプロセスで
再読み込みする必要がある。
 - そのためのプロセス間通信のコードも必要
- 新しい獣群の暴走(thundering herd)
 - 新しいコネクションが来るとすべてのプロセスが起こされる
 - Blocking accept() では解決済み
 - Non-blocking accept() で再び現れる

新しいアーキテクチャ

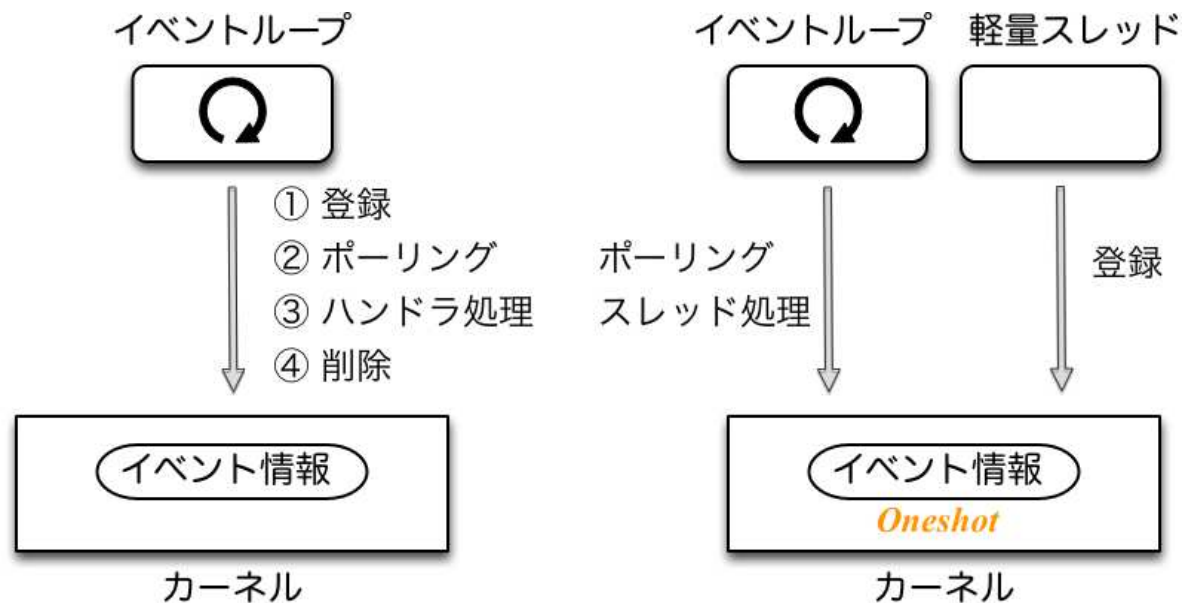
- プロセス内に複数のイベントループを回す



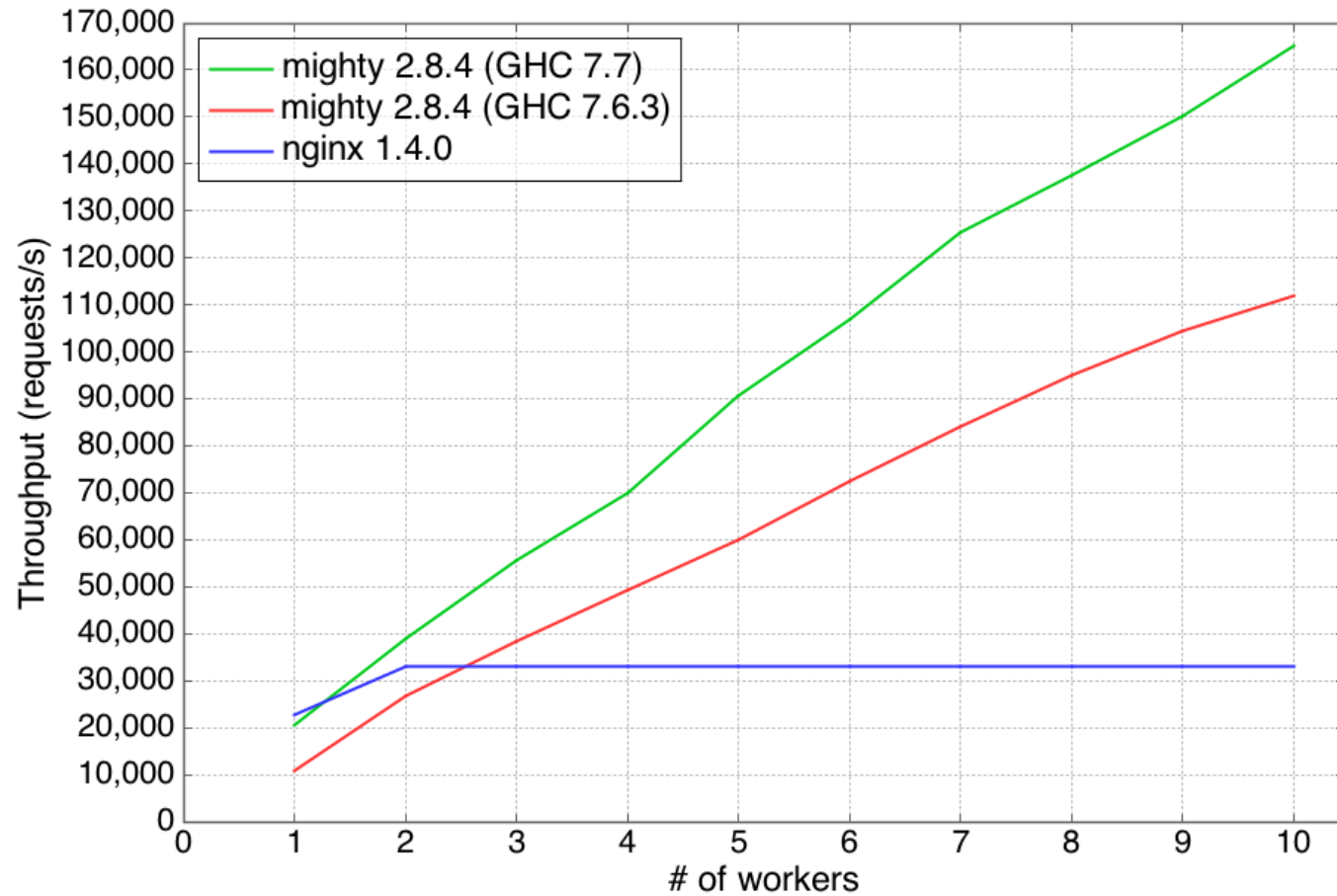
- 同一プロセス
 - 定型コードの再発明は不要
 - 共有資源あり
- 一つのイベントループが新しいコネクションを処理
 - 新しい獣群の暴走はなくなる

新しいイベント処理

- これまでのイベント処理
 - libevent → libev → libuv
 - select() のセマンティクスを継承している
- 我々のイベント処理
 - 登録とポーリングの分離
 - Oneshot の活用
 - あらゆる OS のバグを踏んだ！



ベンチマーク



■ 注：nginxの方がはるかに多機能です

成果物のマージ

再実装



レビューしやすいように
コミットを再構成

レビュー



あっさり通った

自動検証



MacOS のバグを踏んだ。
長い戦いの始まり。

マージ



2013年秋に GHC 7.8.1
に含まれてリリース