

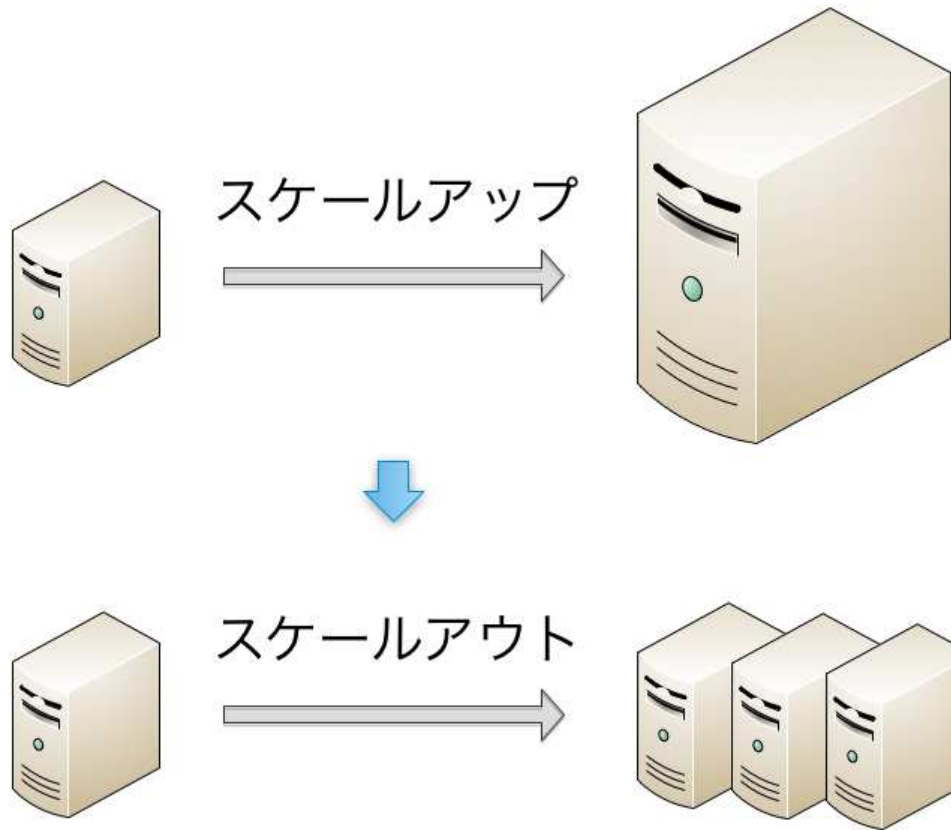
# マルチコア時代の サーバプログラミング とHaskell

2011.11.9

IJ-II  
山本和彦

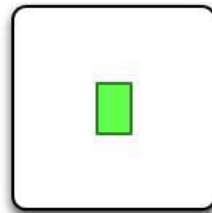
# サーバのはやり

---

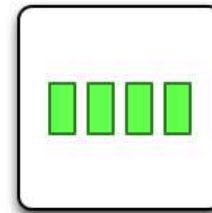


# CPU のはやり

---



マルチコア化



あなたのサーバは  
マルチコアを活用できていますか？

お題

マルチコア時代に  
使える技術を再検証してみる

## 内容

### 同時接続処理技術

都度プロセス生成

都度スレッド生成

プロセスプール

スレッドプール

イベント駆動

マルチプロセス+イベント駆動

マルチスレッド+イベント駆動

### プログラミング技術

コールバック

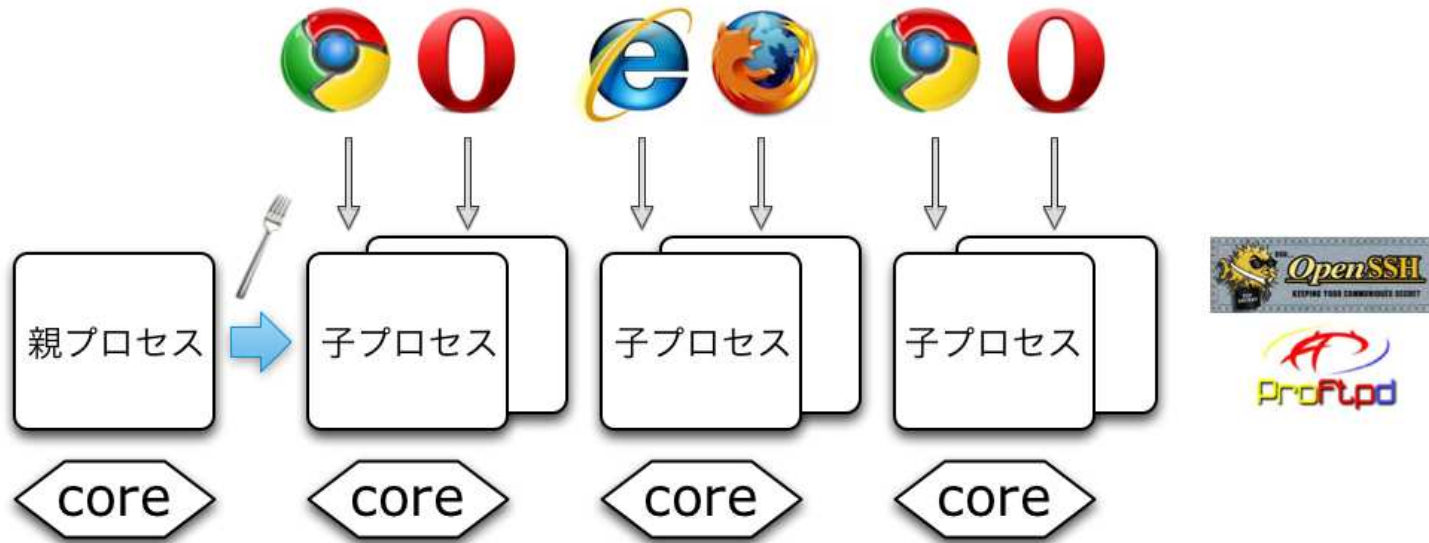
コルーチン

軽量プロセス

軽量スレッド

# 同時接續處理技術

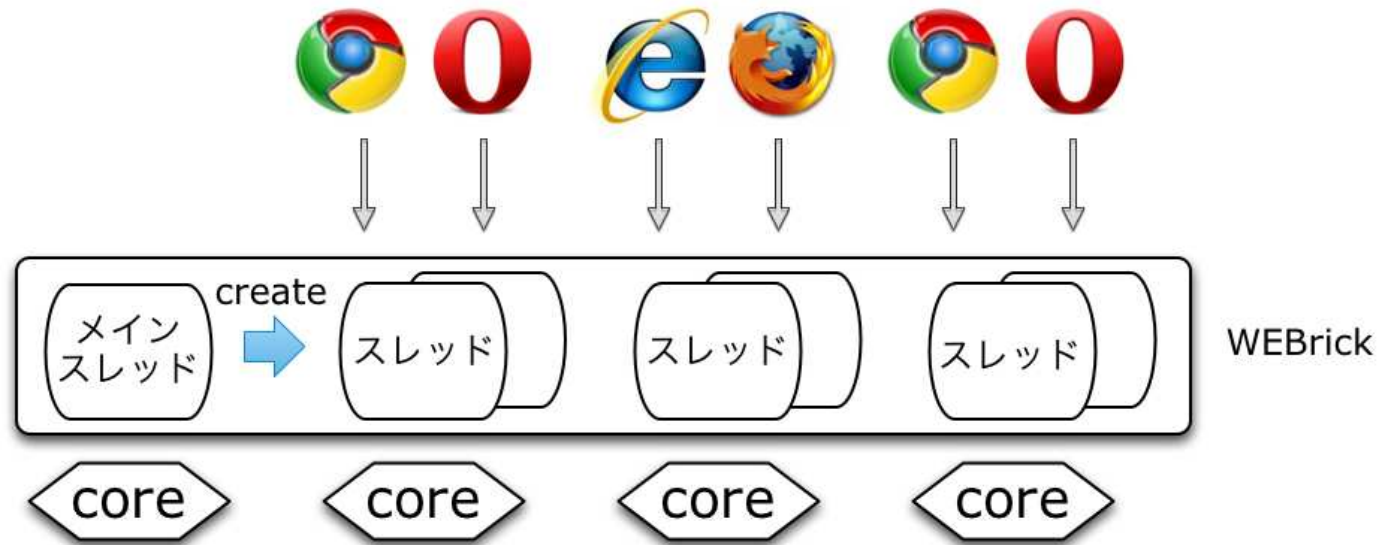
## 都度プロセス生成



- 接続を受けるたびにプロセスを fork する
- 利点
  - コードが簡単で見通しがよい
  - セキュアである
- 欠点
  - fork は遅い
  - コンテキストスイッチは重い

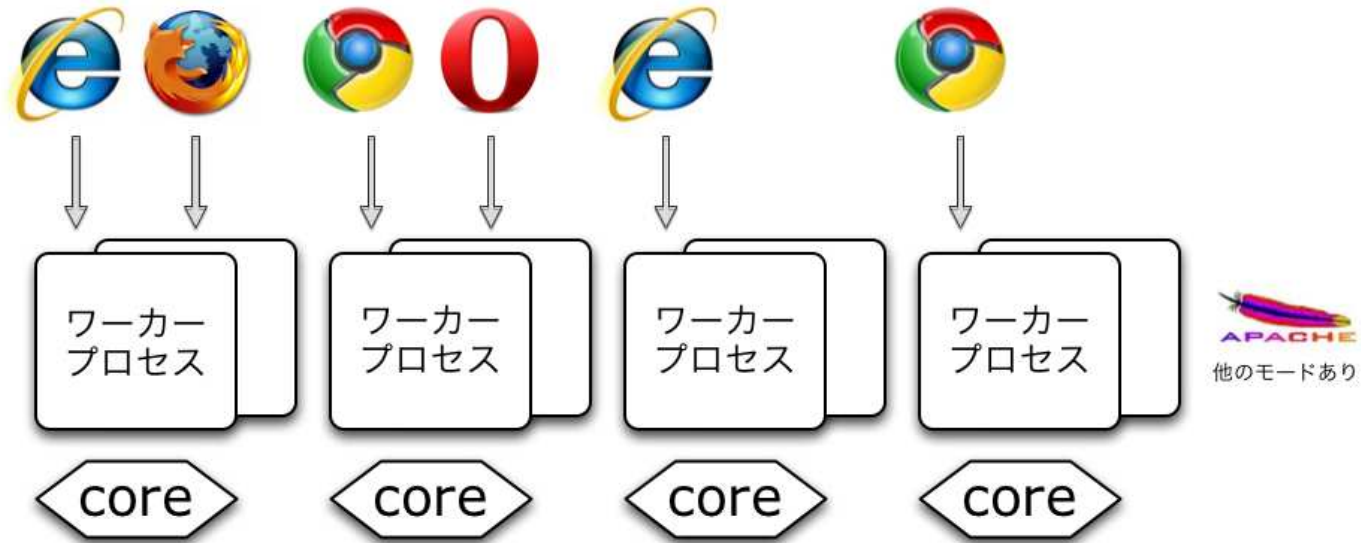


## 都度スレッド生成



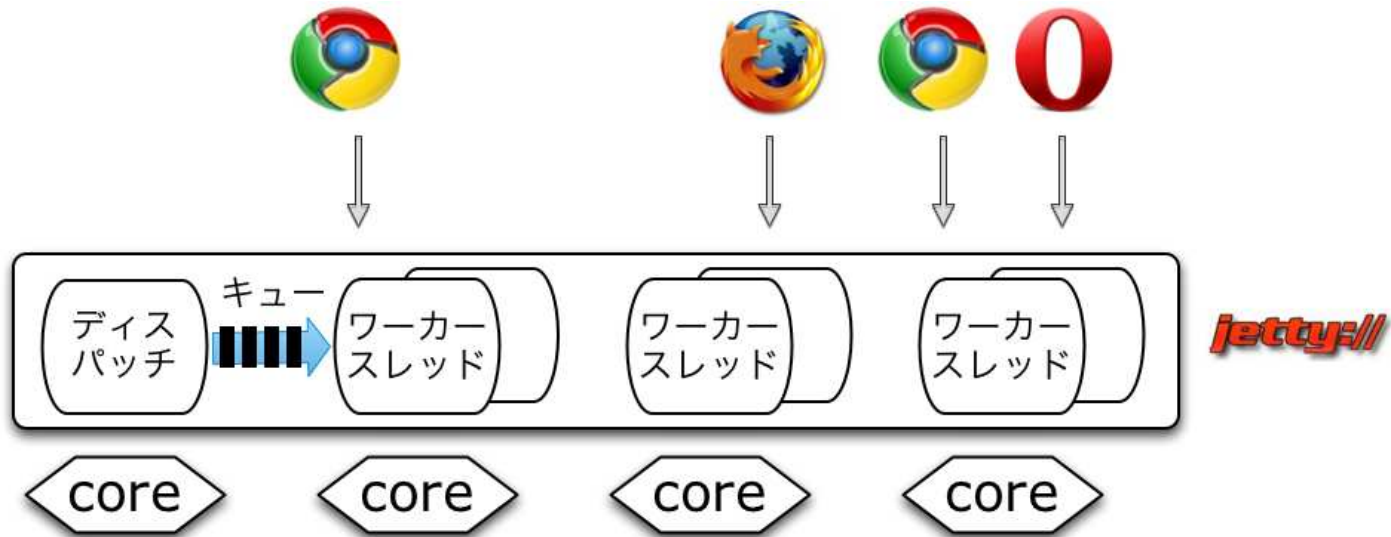
- 接続を受けるたびにスレッドを作る
- 利点
  - 共有資源が使える
- 欠点
  - ほとんど使われてない

## プロセスプール (prefork)



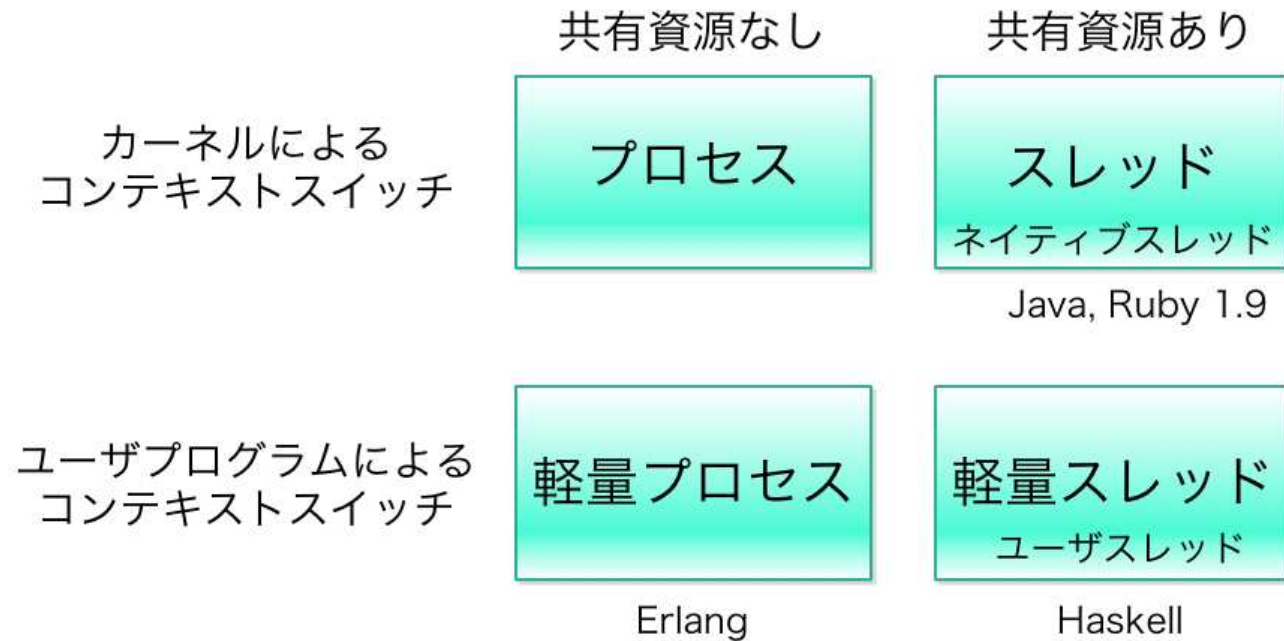
- プロセスをあらかじめ fork しておく
- 利点
  - コードの見通しがよい、セキュアである
  - fork の遅さがなくなる
- 欠点
  - ブロックしてもよいが、プロセスが足らなくなる可能性がある
  - コンテキストスイッチは重い

# スレッドプール



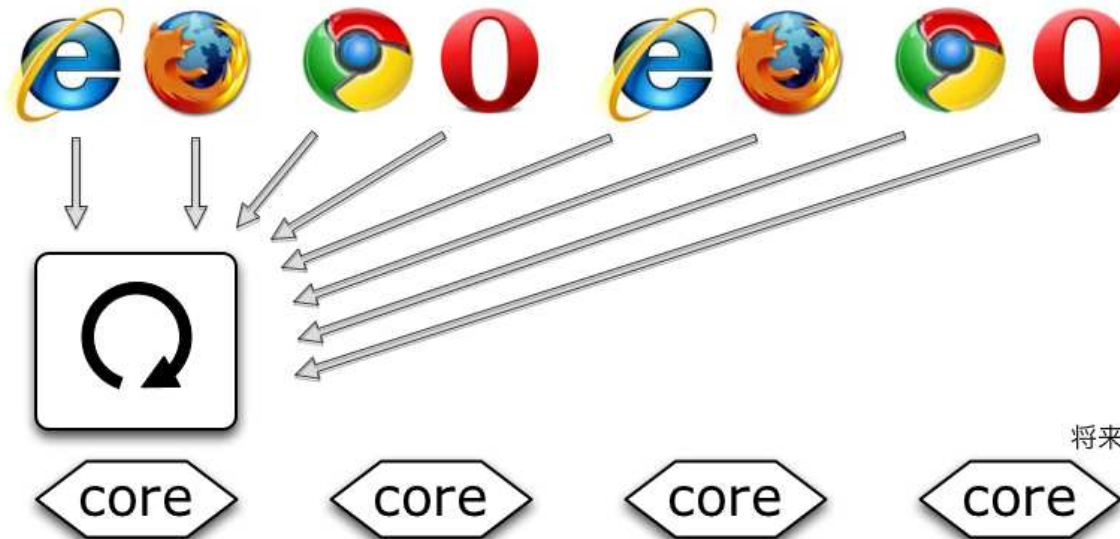
- 接続がキューを使ってワーカースレッドにディスパッチされる
- 利点
  - 都度プロセス生成/プロセスプールより資源を消費しない
- 欠点
  - ブロックしてもよいが、スレッドが足らなくなる可能性がある
  - ディスパッチがボトルネックになりうる

# プロセスとスレッド



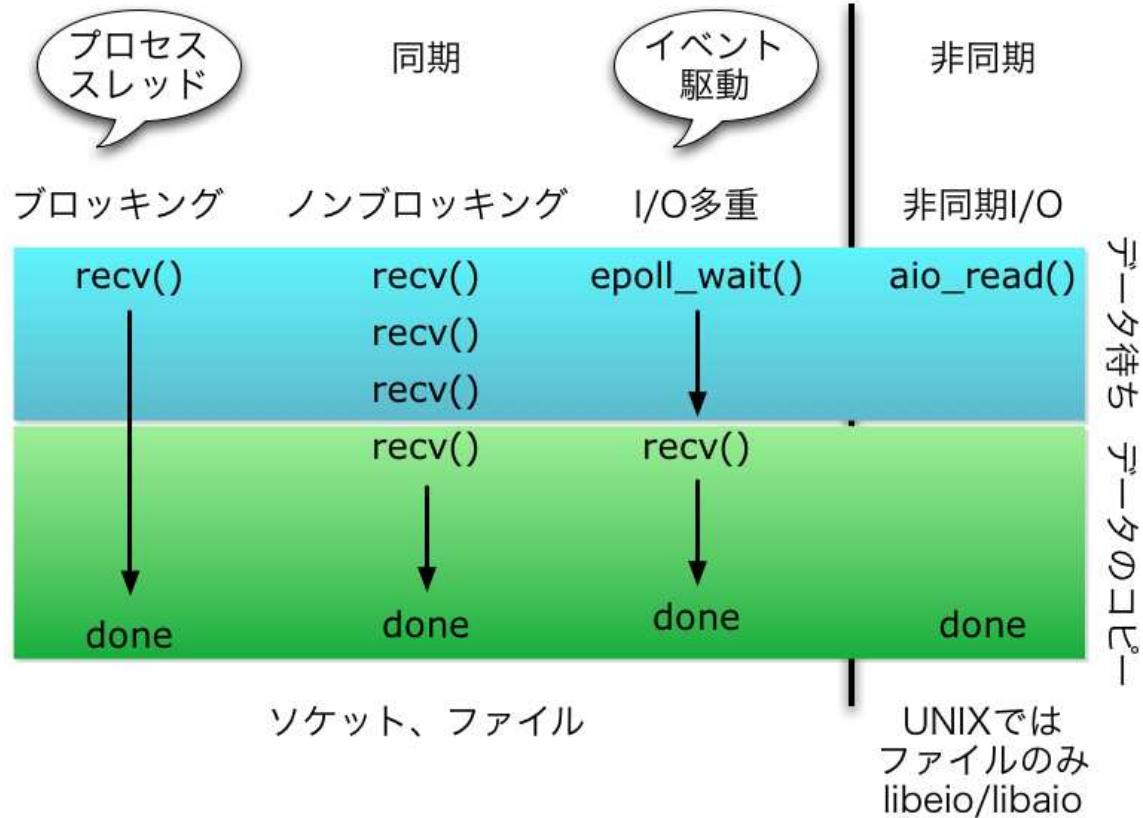
軽量 = 10万個上げても大丈夫

## イベント駆動



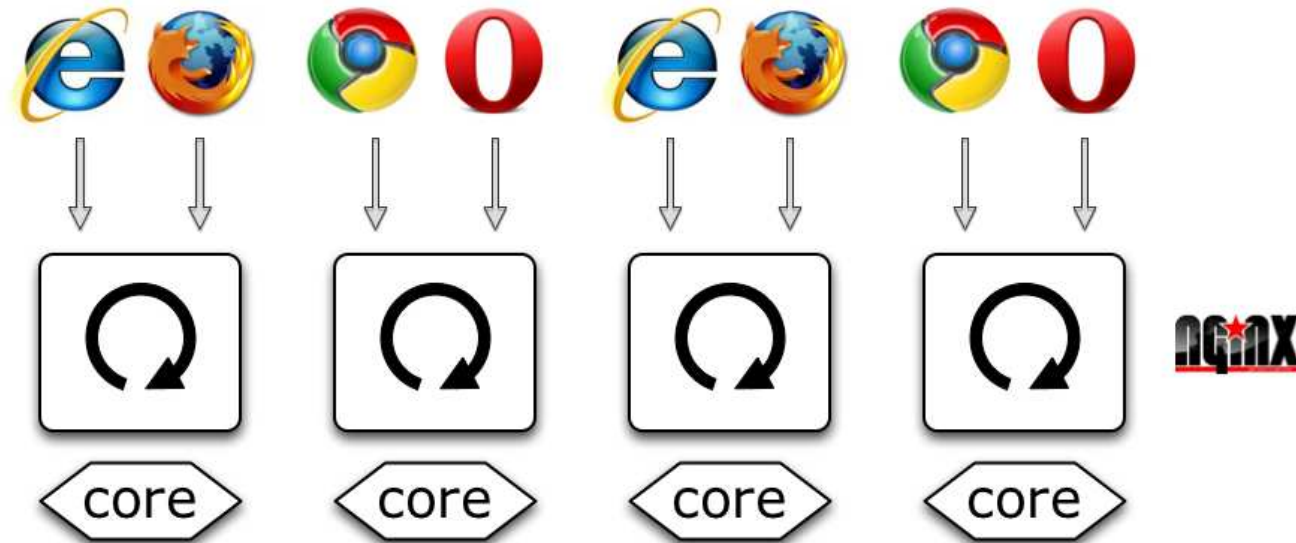
- I/O 多重を使ってハンドラ(コールバック)を駆動する
- 利点
  - 資源をあまり消費しない
- 欠点
  - ハンドラではブロックさせてはいけない
  - コードの見通しは悪い
  - 複数のコアを活用できない

# ブロッキングとI/O多重



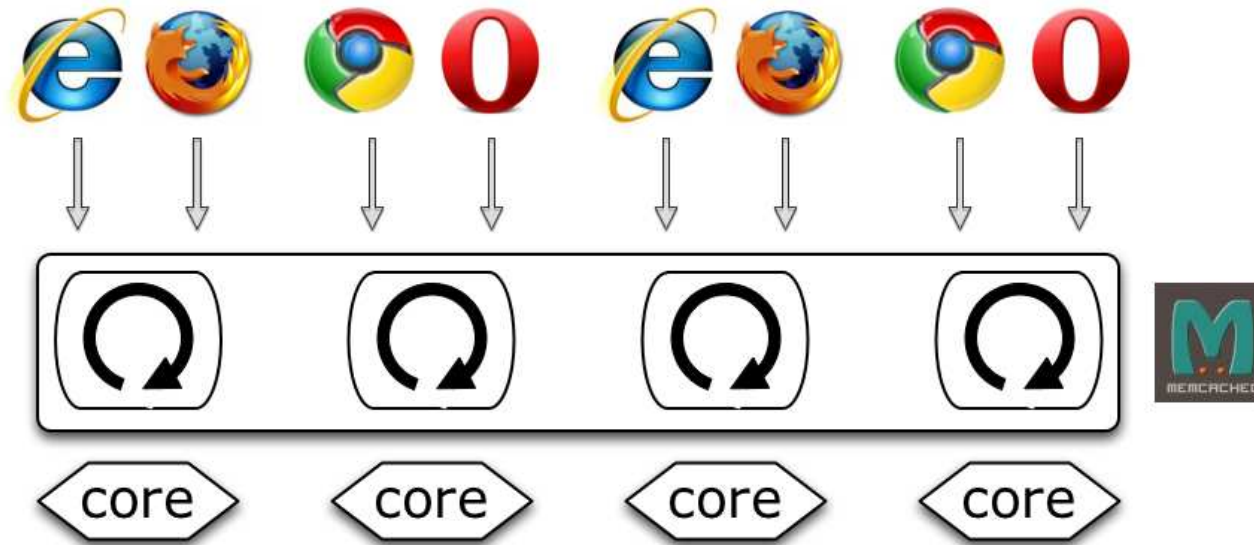
- I/O多重は同期だが、上位層からは非同期に見える

## マルチプロセス+イベント駆動



- 複数のプロセスを立ち上げたイベント駆動
- 利点
  - 資源をあまり消費せずにコアの性能を引き出せる
- 欠点
  - ハンドラではブロックさせてはいけない
  - コードの見通しは悪い

## マルチスレッド+イベント駆動



- 複数のスレッドを立ち上げたイベント駆動
- 利点
  - 資源をあまり消費せずにコアの性能を引き出せる
- 欠点
  - ハンドラではブロックさせてはいけない
  - コードの見通しは悪い



## お勧めの同時接続処理技術

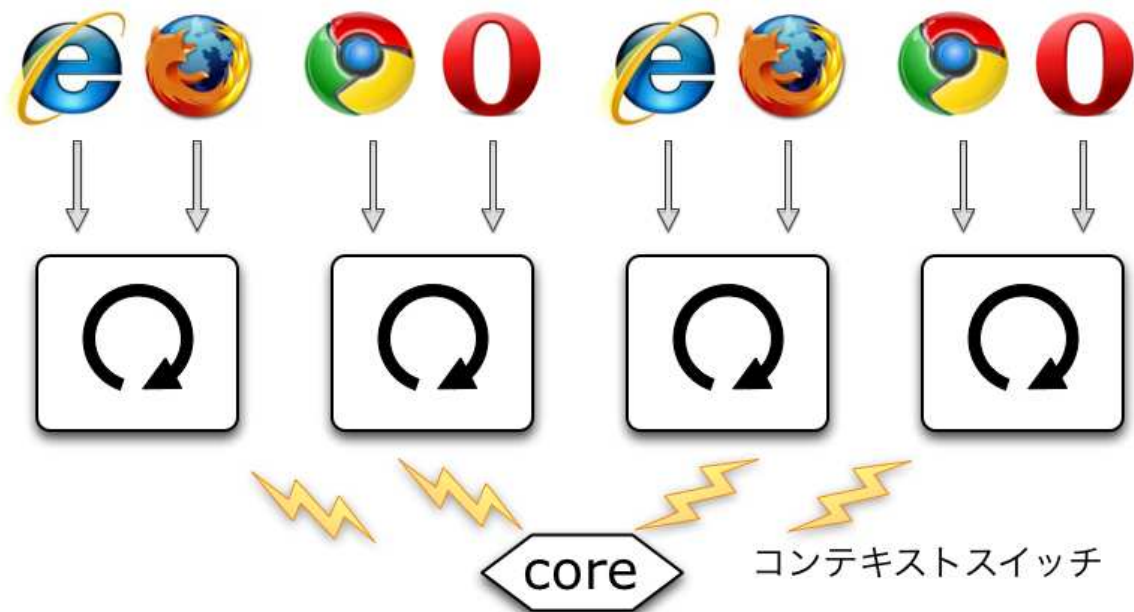
---

マルチプロセス  
+ イベント駆動

マルチスレッド  
+ イベント駆動

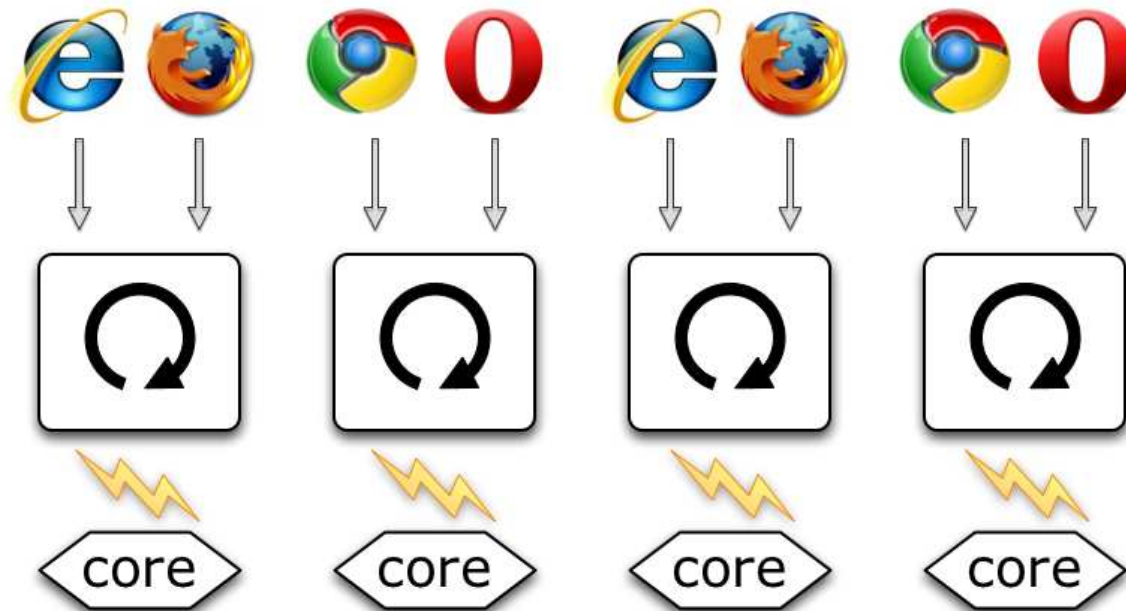
- イベント駆動をマルチプロセス+イベント駆動に変更するのは簡単
- 共有資源が必要ならマルチスレッド+イベント駆動
  - 例) memcached の「キャッシュ」

## マルチとはいくつ？



- カーネルのコンテキストスイッチは重い
  - コンテキストスイッチが多発すると負け

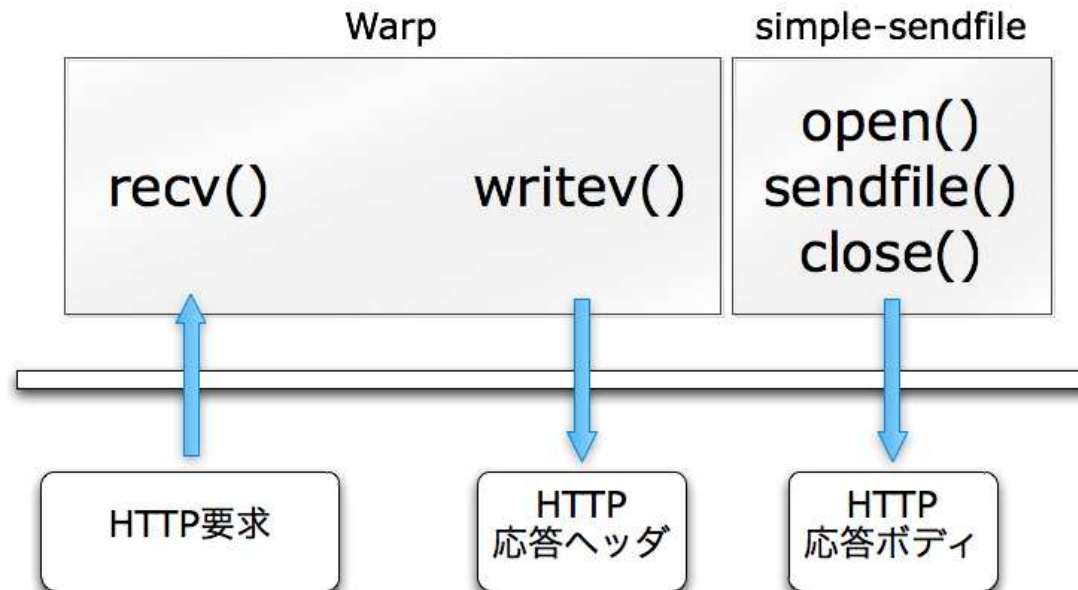
## コア数だけ上げる



- しかも、できるだけコンテキストスイッチが発生しない工夫をすべき
  - システムコールをなるべく呼ばない
    - システムコールの結果をキャッシュ

# Haskell の HTTP エンジン Warp

- Warp が発行するシステムコール

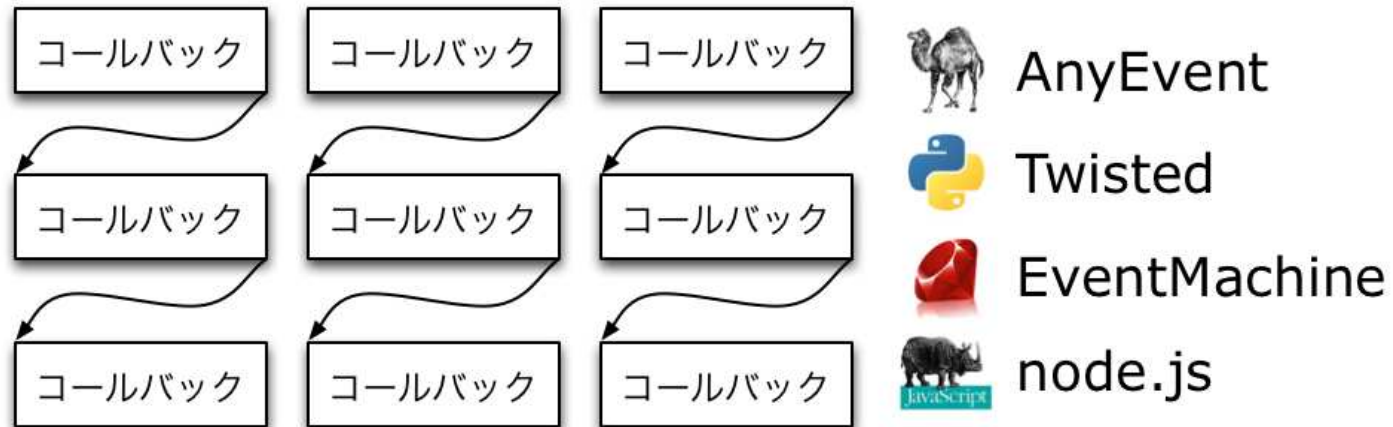


- システムコールを減らす工夫
  - lseek() は使わない
  - stat() の結果は、上位でキャッシュする

# プログラミング技術

## 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);
```

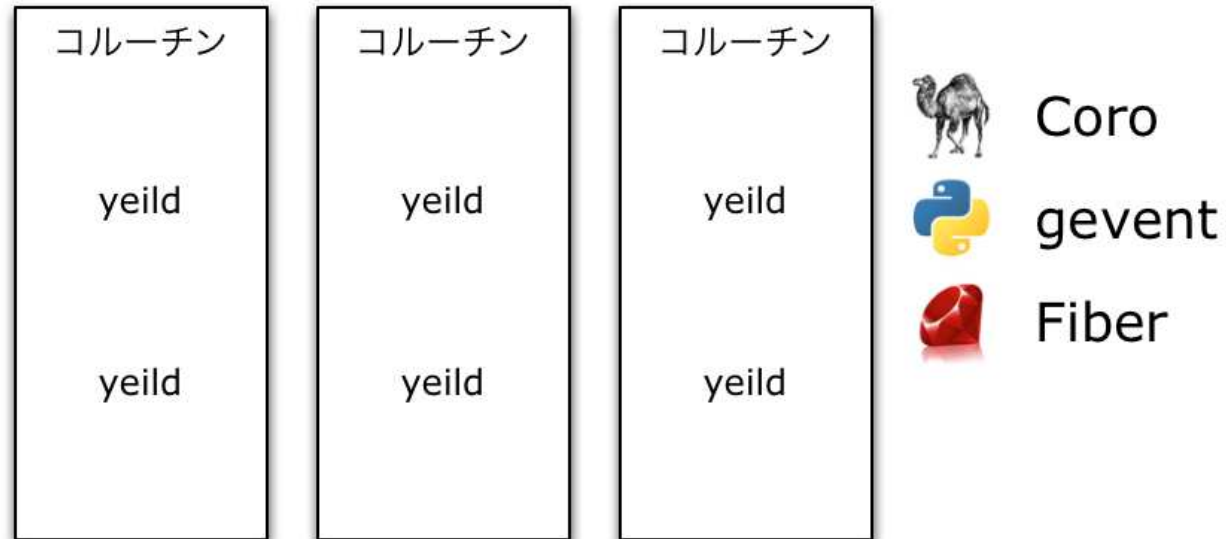
- コールバック・スタイル (非同期)

```
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

## コルーチン

- イベント駆動の上にコルーチン層をかぶせる



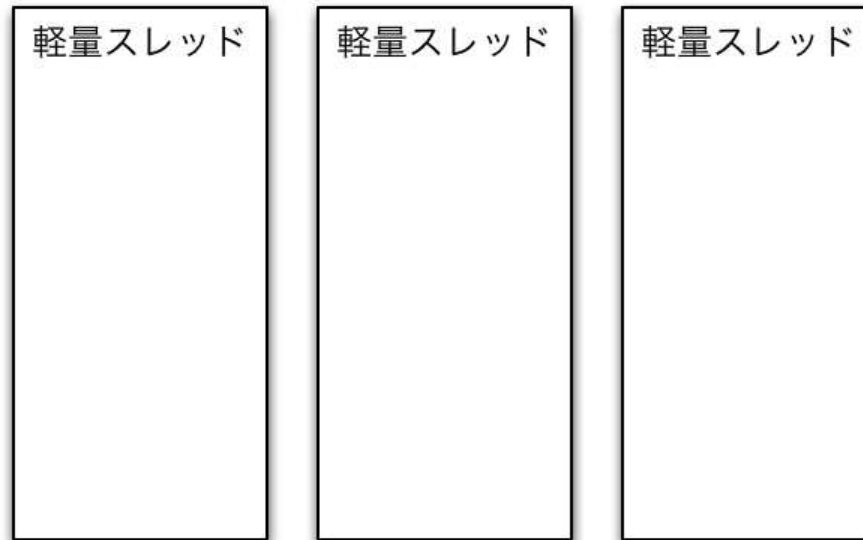
- コードの見通しがよい
- 横取りできないので明示的に「譲る」必要がある



## 軽量プロセス、軽量スレッド

---

- イベント駆動の上に軽量プロセス、軽量スレッド層をかぶせる



- 通常のプログラミングと同じ
- コードの見通しがよい

## 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

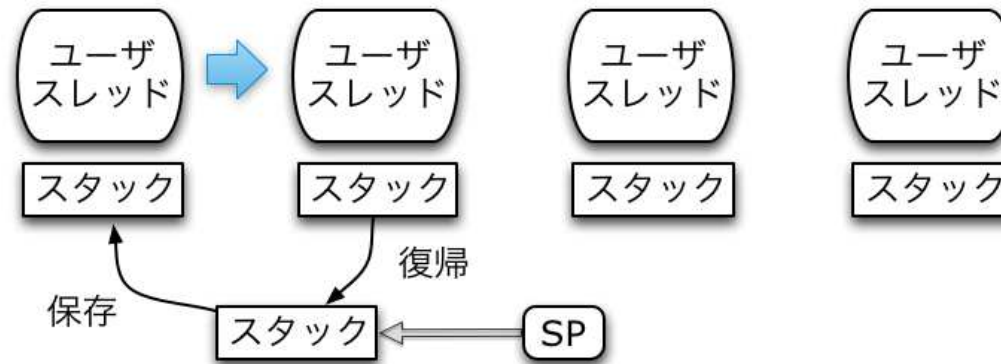
過去の資産のしがらみ

外部言語呼び出し

呼び出した外部言語がブロックする

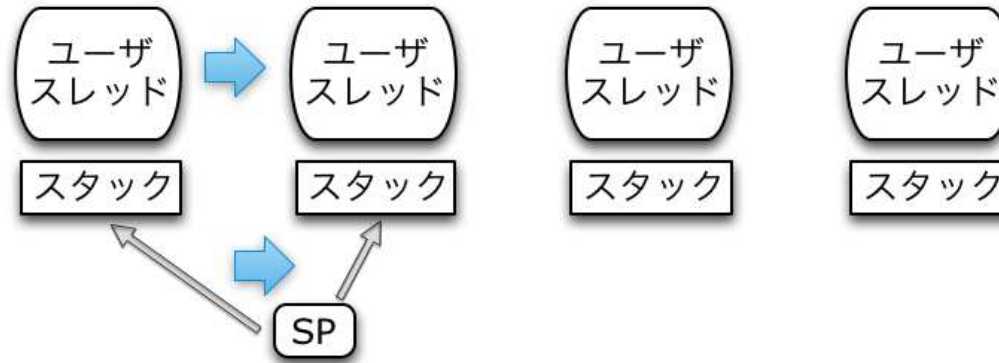
ユーザスレッドがうまくいかなくなる最大の原因

## Ruby の場合



- Ruby のユーザースレッドは重かった
  - ユーザー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 では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タ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

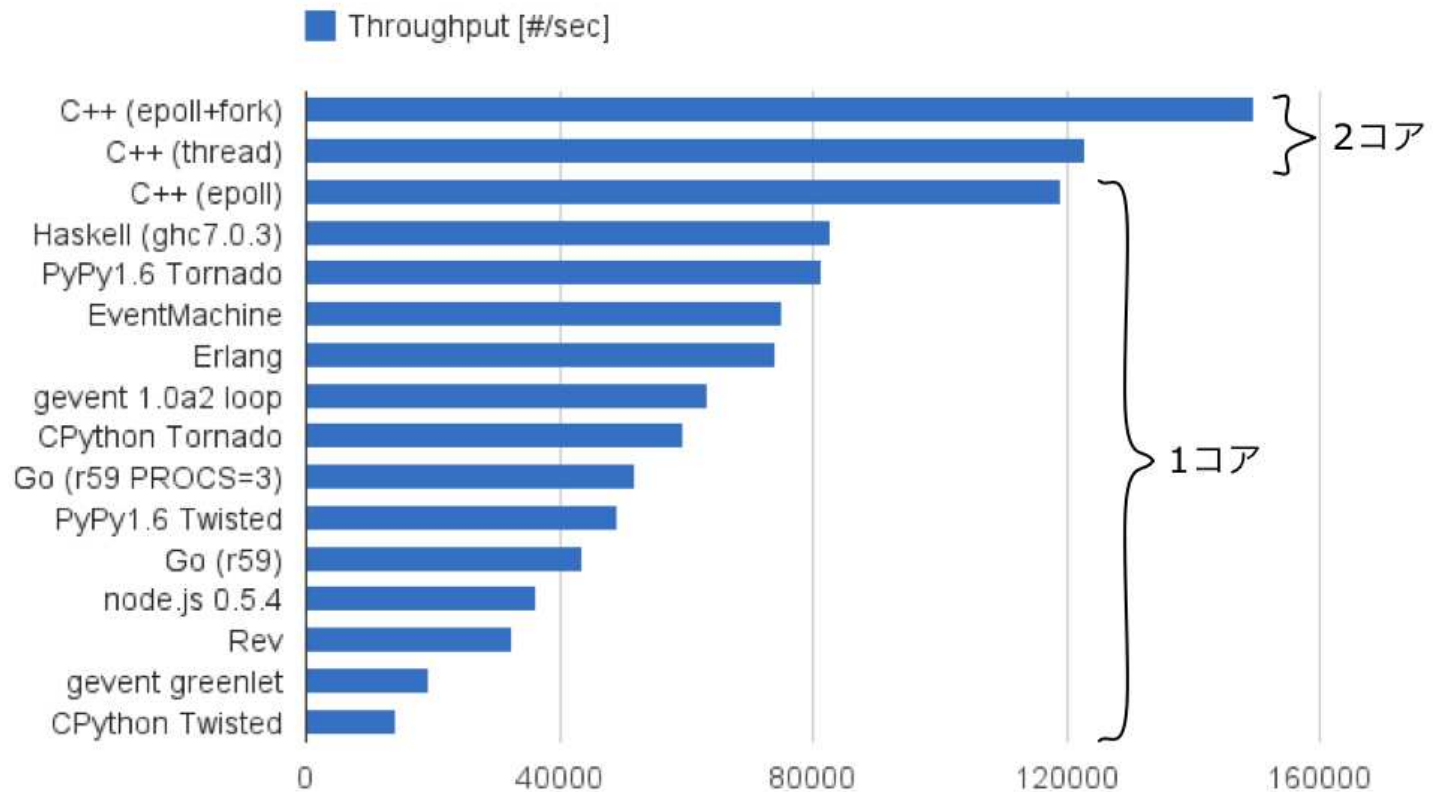
ブロックする外部言語  
呼び出しはネイティブ  
スレッドに任せる

## お勧めのプログラミング技術

---

- 軽量プロセス、軽量スレッド
  - あれば
- コールバック、コルーチン
  - なければ
  - 現実的には軽量プロセス、軽量スレッドがない言語が多い

# エコーサーバのスループット



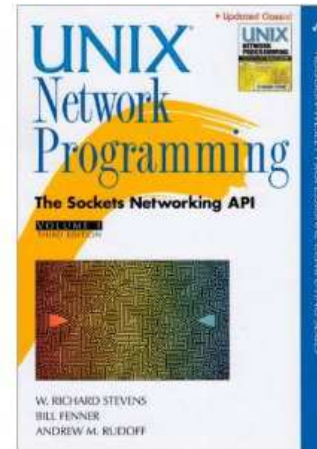
- 最速TCPサーバの条件
  - ～逆襲の Erlang と Haskell の挑戦～

## まとめ

---

- 同時接続処理技術
  - マルチプロセス+イベント駆動
  - マルチスレッド+イベント駆動
  - コアの数だけ起動
  - コンテキストスイッチが多発すると負け
- プログラミング技術
  - 軽量プロセス
  - 軽量スレッド
  - 見通しのよいプログラミング
  - システムコールをなるべく呼ばない

## 参考文献



- モダンネットワークプログラミング入門
  - 古橋貞之
  - WEB+DB PRESS Vol 55 (2010)
- Unix Network Programming Vol 1 (3rd)
  - Stevens, Fenner, Rudoff
  - Addison Wesley
- node.js とは何だったのか？
  - LL Planets



## 参考文献

---

- Mighttpd -- a High Performance Web Server in Haskell
  - Kazu Yamamoto
  - The Monad.Reader, Issue 19
    - <http://themonadreader.files.wordpress.com/2011/10/issue19.pdf>