IOクラスの入出力メソッドについて

cuzic

自己紹介

- cuzic
  - 「きゅーじっく」と読みます。
    - × quzic　× cusic
  - 「リファクタリング Ruby エディション」の読書会を計画しています。
    - ＠ JR 尼崎駅徒歩２分（小田公民館）
    - 日程未定
    - Kent Beck 著
  - 最近 JavaScript の勉強中
    - 関数が First-Class Object でいいね。

IO クラス

- みんな知っているよね？

puts "Hello, World!"

  - STDOUT や STDIN は IO のインスタンス
  - open("/tmp/tempfile") do |io| ~ end
  - ファイル入出力だけでなく、
    ネットワーク処理などにも IO クラスを利用。

Ruby の IO はかしこい

- Ruby のスレッドとうまく連携
  - ブロックする IO#read / IO#write は内部的に select(2) を利用。上手に他のスレッドを実行
- 高水準 IO と低水準 IO の両方が利用可能
  - 高水準IO ：
FILE構造体（stdio）を使う printf(3) や puts(3)
  - 低水準IO ：
file descriptor を使う read(2) など
- エラーが生じると例外を raise。
  - EAGAIN、EINTR、EWOULDBLOCK とか
- 低レベルのシステムコールも数多く利用可能
  - fcntl (2) 、fsync (2) などに加え、BasicSocket では getsockopt(2) や recv(2) なども利用可能

IO#read (1)

- IO#read *length*
  - 長さを指定したいときはふつう これを使う。

```
buffer = nil
open("/dev/urandom") do |io|
  buffer = io.read 10
end
p buffer
#=> "¥216X¥222¥300¥217uT¥236¥232t"
```

IO#read (2)

- IO#read *length, buffer*
  - 実は 第二引数を指定可能。
事前にメモリ割り当てすることで高速化。

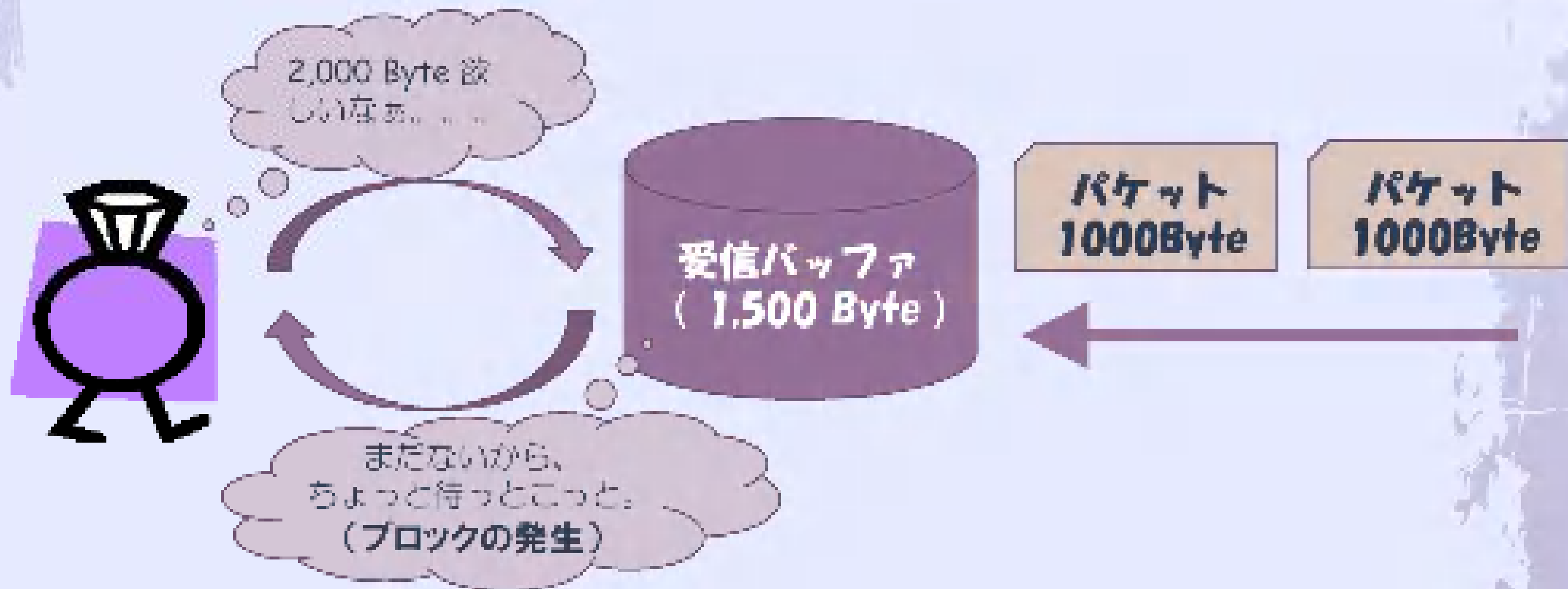
```
require 'benchmark'
buffer = " " * 1000
Benchmark.measure do
  open("/dev/zero") do |io|
    1000_000.times do
      io.read 1000, buffer
    end
  end
end.display
#=>  1.5000   0.0000   1.5000 (  1.3980)
（比較） 58.8900   4.3440  63.2340 ( 62.6950)
```

IO#sysread

- IO#sysread は IO#read とどう違うの？
  - IO#read は **受信バッファ** を経由**する**
  - IO#sysread は **受信バッファ** を経由**しない**
- 受信バッファ って？
  - IO はファイル以外にもいろいろ扱う
  - ネットワーク（ TCP/UDP 等）とか
  - 欲しい length のデータがそろうまで、内部的にデータを貯めておいたり
  - 取り出されるのに先立って、データを保存しておいたり

受信バッファ

- 受信バッファのおかげで詳細な入出力の処理を気にせず、プログラムを書ける
- 反面、ブロック（待ち）が発生してしまう。高速化の障壁になることも・・・。
  - ・待たずに別の処理をしたいのに・・・。
  - ・ある分だけでいいからすぐ欲しいのに・・・。



IO#read と IO#sysread

- IO#read を使うと欲しい長さだけ取得可能
  IO#sysread を使うと待ちが生じない

```
r, w = IO.pipe

str = "0123456789"
Thread.start do
  w.write str
  sleep 10
  w.write str
end

puts Time.now.strftime("%H:%M:%S")
buffer = r.read 15
# ここで、ブロック（待ちが発生）する
puts Time.now.strftime("%H:%M:%S")
puts buffer #=> "012345678901234"
```

```
r, w = IO.pipe

str = "0123456789"
Thread.start do
  w.write str
  sleep 10
  w.write str
end

puts Time.now.strftime("%H:%M:%S")
buffer = r.sysread 15
# すぐに次の行に移る
puts Time.now.strftime("%H:%M:%S")
puts buffer #=> "0123456789"
```

混ぜるな危険

- 待ちをできるかぎり少なくしたいけど、、、
- IO#read と IO#sysread は同時利用不可
  - 受信バッファの都合で、同時に使うとバグの元
  - IO#gets 、IO#each_line など、みんな IO#sysread とは同時利用できません・・・。
- けど、どうしても同時に使いたい！！！
  - そんなあなたに耳よりな話があります！！
    - **IO#read_nonblock**
    - **IO#readpartial**

IO#readpartial

- Ruby 1.8.3 以降で利用可能
- 受信バッファにデータがある場合は、とりにいかない

	返り値	受信バッファ	Pipe Content
r, w = IO.pipe		empty	empty
w.write str		empty	"0123456789"
r.read 5	"01234"	"56789"	empty
w.write str		"56789"	"0123456789"
r.readpartial 10	"56789"	empty	"0123456789"
r.readpartial 10	"0123456789"	empty	empty
r.readpartial 10	ブロック		
w.write str （別スレッド）		empty	"0123456789"
ブロック戻る	"0123456789"	empty	empty

IO#readpartial

- IO#readpartial では受信バッファに貯めることはしない

	返り値	受信バッファ	Pipe Content
r, w = IO.pipe		empty	empty
w.write str		empty	"0123456789"
r.readpartial 5	"01234"	empty	"56789"
w.write str		empty	"5678901234567 89"
r.readpartial 10	"5678901234"	empty	"56789"
r.readpartial 10	"56789"	empty	empty
r.readpartial 10	ブロック		
w.write str （別スレッド）		empty	"0123456789"
ブロック戻る	"0123456789"	empty	empty

IO#read_nonblock

- Ruby 1.8.5 以降で利用可能
- IO#readpartial と同様に IO#read 等との同時利用が可能
- 取り出せるものが何もないときは、例外が発生する
- 何か取り出せるものがあれば、それを返す

	返り値	受信バッファ	Pipe Content
r, w = IO.pipe		empty	empty
w.write str		empty	"0123456789"
r.read 5	"01234"	"56789"	empty
w.write str		"56789"	"0123456789"
r.read_nonblock 10	"56789"	empty	"0123456789"
r.read_nonblock 10	"0123456789"	empty	empty
r.read_nonblock 10	Errno::EAGAIN 例外が発生する		

各メソッドの動作

- IO#read （ブロッキングモード）
  - 確実に length のデータを返す
  - 足りてなかったら、データの到着を待つ
- IO#readpartial
  - 求める length のデータがそろってなくても、ある分だけ返す
  - 返すべきデータが何もなければブロックする。
  - けど、なんかデータが到着したら、それを返す。
- IO#read_nonblock
  - 返すべきデータがなければ例外を raise する。ブロックしない。
- IO#sysread （ノンブロッキングモード）
  - IO#read 等と混用できない点を除いて、IO#read_nonblock と同じ動作
  - どの Ruby でも問題なく動く。

細かい違いを整理する

	read	readpartial	read_nonblock
受信バッファにデータがあり、長さが十分なとき	受信バッファから取り出す	受信バッファから取り出す	受信バッファから取り出す
受信バッファにデータがあるが、長さが不十分なとき	ブロックする	受信バッファのデータを返す	受信バッファのデータを返す
受信バッファにデータがないが、データは到着している場合。	到着したデータから取り出す。受信バッファに保存する。	到着したデータを取り出す。	到着したデータを取り出す
受信バッファが空でデータも到着していない	ブロックする	ブロックする	EAGAIN などの例外を raise する

いつ使うの？

- IO#readpartial *length*
  - 取り出せるものが何かあれば取り出したいが、何もなければ、ブロックして欲しいとき
  - length を大きな値として、ネットワークプログラミングなどで利用したり。
  - IO#sysread と違い、IO#read等と共存可能
- IO#read_nonblock *length*
  - 一切ブロックさせずに IO を利用したい場合
  - IO#sysread と違い、IO#read等と共存可能
- IO#sysread *length*
  - 文字数がずっと短い
  - IO#sysread 使いこそが真の漢（？）。

具体的な利用例は？

- Mongrel
  - Ruby で書かれた Web サーバ
  - 一部で readpartial を利用している
- SecureRandom
  - Ruby 1.8.7 以降標準添付
  - 安全な乱数生成機
  - /dev/urandom を readpartial で読み込み
    ※ Unix 環境の場合
- EventMachine
  - 非同期サーバを作るフレームワーク
  - 一部で read_nonblock を利用している

まとめ

- 「リファクタリング Ruby エディション」読書会を計画中
- Ruby の IO はかしこい
  - 低水準のことも高水準のこともできちゃう
- IO#read の第２引数は高速化に役立つ
  - こともある。
- IO#readpartial はネットワークプログラミングなどで重宝する

ご静聴ありがとう
ございました。

おまけ（ io/nonblock ）

- Ruby 1.8 / 1.9 には io/nonblock というノンブロッキングモードにするライブラリがあるが・・・
  - 正直、私にはよく分かりません・・・。
  - 期待した動作をしないし、、、
  - 昔のバージョンでは効果があった？？？
  - Cygwin にはない？？？
- open の第２引数は効果があるみたい

open("/tmp/fifo", IO::NONBLOCK | IO::RDONLY)