怠惰なRubyistへの道
Enumerator::Lazyの使いかた

Chikanaga Tomoyuki
(@nagachika)
Yokohama.rb

Agenda

- Enumerable
- Enumerator
- Enumerator::Lazy

WARNING

今日の話は Ruby 2.0 の 新機能について

Ruby 2.0

2013年リリース予定

[ruby-dev:44691] より

Aug. 24 2012: big-feature freeze

Oct. 24 2012: feature freeze

Feb. 24 2013: 2.0 release

Ruby の生誕 20 周年

Ruby 2.0

未来の話

2.0 is in development

※注※

仕様は開発中のものであり
リリース版では変更される場合があります

Install Ruby 2.0

未来を手中に

rvm, rbenv を使っていれば

```
rvm install ruby-head
rbenv install 2.0-devel
```

Ruby 2.0 for Windows

Ruby環境構築講座
Windows編

arton

Self Introduction

twitter id: @nagachika

ruby trunk changes
(http://d.hatena.ne.jp/nagachika)

CRuby committer

Yokohama.rb → 福岡遠征中

Sound.rb(ruby-coreaudio, ruby-puredata)

Enumerator::Lazy

Enumerator::Lazy

の前に

Enumerable

Enumerable

もちろん使ってますよね?

Array, Hash, IO...

Array や Hash, IO は
Enumerable を
include している

Enumerable

【形容詞】 数えられる [日本語 WordNet(英和)]

可算；可付番 [クロスランゲージ 37分野専門語辞書]

Capable of being enumerated; countable [Wikitionary]

Enumerable

「each メソッドを呼ぶと、何かが順に(N回) yield で渡される」というルール(N >= 0)

Enumerable

each メソッドが定義されているクラスに include して使う(Mix-in)

Enumerable

each だけ用意しておけば
map, select, reject, grep, etc...
といったメソッドが使えるようになる

(これらのメソッドが each を使って
実装されている)

An Example

```
class A
  include Enumerable
  def each
    yield 0
    yield 1
    yield 2
  end
end
A.new.map{|i| i * 2 } # => [ 0, 2, 4 ]
```

ここまでのまとめ

Enumerable は each メソッドを使って
多彩なメソッドを追加する
なにが「数え上げられる」のかは
each メソッドがどのように
yield を呼び出すかで決まる

Enumerator

使ってます?

Enumerator

each に渡すブロックを保留した
状態でオブジェクトとして持ち回す

to_enum, enum_for

Enumerator の作りかた

```
Enumerator.new(obj, method=:each, *args)
obj.to_enum(method=:each, *args)
obj.enum_for(method=:each, *args)
Enumerator.new{|y| ...}
```

↑これだけちょっと特殊

Object#enum_for

each 以外の任意のメソッドの呼び出しを
Enumerator にできる
(実は Enumerable でなくてもいい)

```
e_byte = $stdin.enum_for(:each_byte)
  => バイトごとに yield

e_line = $stdin.enum_for(:each_line)
  => 行ごとに yield
```

Enumerator.new{|y|}

```
e = Enumerator.new{|y|
  y << 0
  y << :foo
  y << 2
}
```

ブロックが each の代わり。
ブロックパラメータ y に << すると
yield することになる。

Enumerator.new{|y|}

```
class A
  def each
    yield 0
    yield :bar
    yield 2
  end
end
A.new.to_enum
```

と同じ

Enumerator as External Iterator

外部イテレータとして使える

```
e = (0..2).to_enum
e.next    # => 0
e.next    # => 1
e.next    # => 2
e.next
=> StopIteration: iteration reached an end
```

each が yield する値が順に next で取り出せる

Method Chain

map, select など一部のメソッドはブロックを省略すると Enumerator を返すので複数のメソッドを組み合わせて使う

```
ary.map.with_index { |v,i| ... }
```

Method Chain

```
[:a, :b, :c, :d].map.with_index do |sym, i|
  i.even? ? sym : i
end
=> [:a, 1, :c, 3]
```

Enumerator
独自のメソッド

with_index

[ruby-dev:31953] Matz wrote...

“mapがenumeratorを返すと

obj.map.with_index{|x,i|....}

ができるのが嬉しいので導入したのでした。というわけで、

* eachにもmapにもwith_indexが付けられて嬉しい

というのが本当の理由です。”

Secret Story of Enumerator.

Enumerator は with_index の為に生まれた!

というのは嘘ですが
主な用途は with_index の
ためのような気がします

ここまでのまとめ

- Enumerator
  - Object#to_enum, enum_for
  - 外部イテレータ化
- Method Chain

Enumerator::Lazy

2.0の新機能

Enumerator::Lazy

Enumeratorのサブクラス

Enumerable#lazy

Enumerator::Lazy の作りかた

Enumerable#lazy を呼ぶ

```
>> (0..5).lazy
=> #<Enumerator::Lazy: 0..5>
```

Enumerator::Lazy

Lazyのmap, select, zipなど一部のメソッドがさらにLazyを返す

```
>> (0..5).lazy.map{|i| i * 2}

=> #<Enumerator::Lazy:
#<Enumerator::Lazy: 0..5>:map>
```

Enumerator::Lazy

> Enumerator::Lazy.instance_methods(false)

map	collect	flat_map
collect_concat	select	find_all
reject	grep	zip
take	take_while	drop
drop_wihle	cycle	
lazy	force	

[ruby 2.0.0dev (2012-05-30 trunk 35844)]

force

Enumrator::Lazy#force

to_a の alias

遅延されているイテレータの評価を実行

A Lazy Demo

```
>> l = (0..5).lazy
=> <Enumerator::Lazy: 0..5>
>> l_map = l.map{|i| p i; i * 2 }
=> <Enumerator::Lazy: #<Enumerator::Lazy: 0..5>:map>
# この時点ではまだブロックの内容は実行されていない!
>> l_map.force
0
1
2
3
4
5
=> [0, 2, 4, 6, 8, 10]
# force を呼ぶと実行される
```

Enumeratorとの違い

- Method Chain の使いかたがより強力に
- 中間データの抑制

Pitfall of Enumerator (1)

map, select のようにブロックの戻り値を利用するメソッドを複数繋げることができない(しても意味がなくなる)

```
(0..5).map.select{|i| i % 2 == 0 }
 => [ 0, 2, 4 ]
```

↑ map の意味がなくなっている

True Method Chain

map や select といったメソッドもいくつでも
Method Chain できる。

そう、Lazy ならね。

```
# 1 から 1000 のうち任意の桁に7を含む数を返す
(1..1000).lazy.map(&:to_s).select{|s|
  /7/ =~ s
}.force
```

Pitfall of Enumerator (2)

map, select のようにブロックの戻り値を利用するメソッドを外部イテレータ化しても意味がない

```
e = (0..5).select
e.next # => 0
e.next # => 1
e.next # => 2
e.next # => 3
```

↑ select の意味がなくなっている

True External Iterator

map などにブロックを渡せるので
適用した結果を順に取り出せる

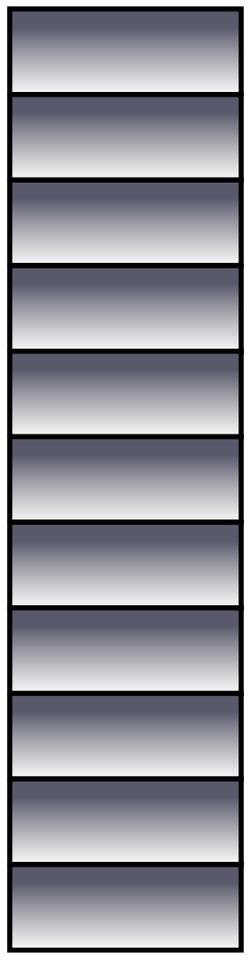
```
l = (0..5).lazy.select{|i| i.even? }
l.next # => 0
l.next # => 2
l.next # => 4
l.next # => StopIteration: iteration reached an end
```

Efficient Memory Usage

Lazy は yield 毎に Method Chain の最後まで処理される

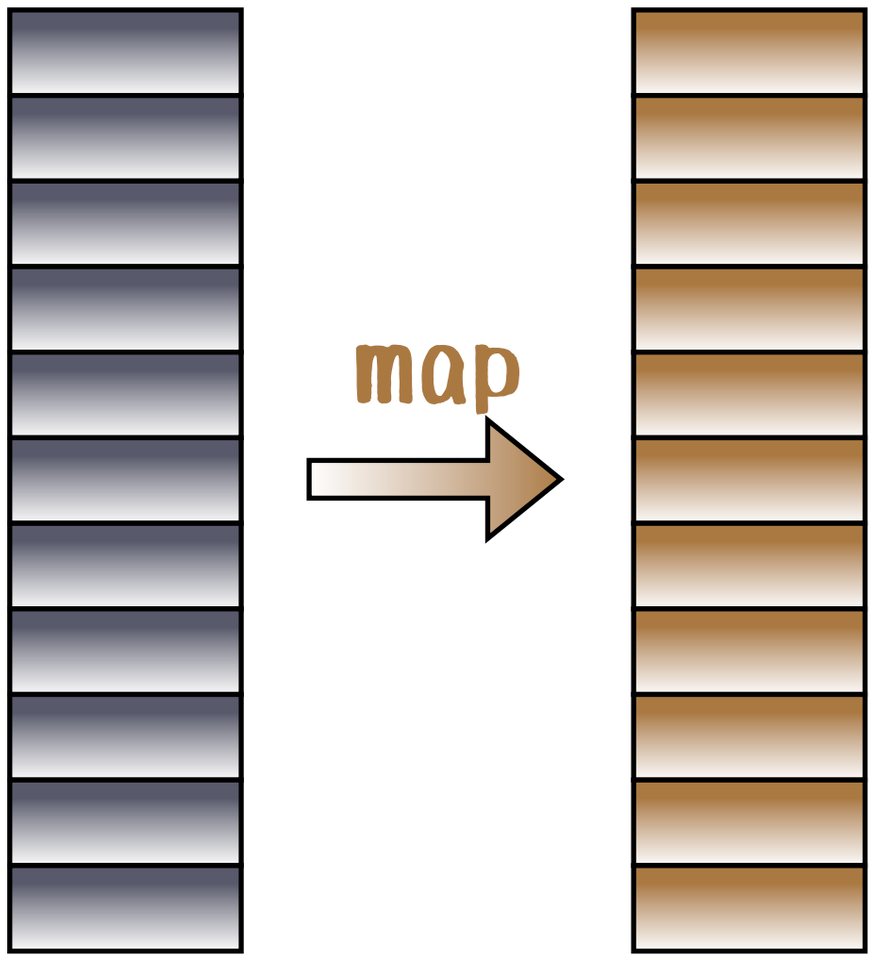
Enumerable version

large_array.map{...}.select{...}



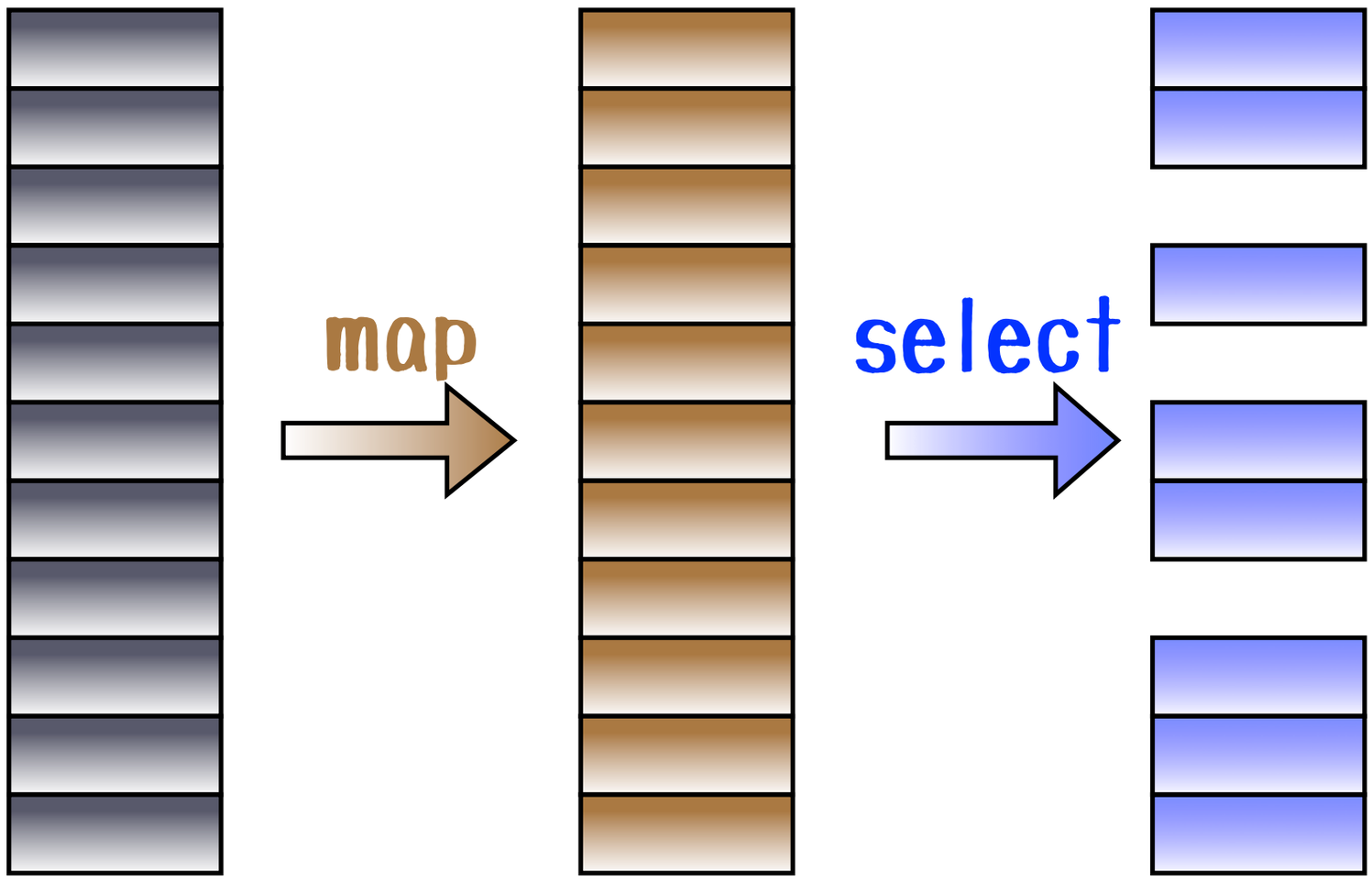
Enumerable version

large_array.map{...}.select{...}



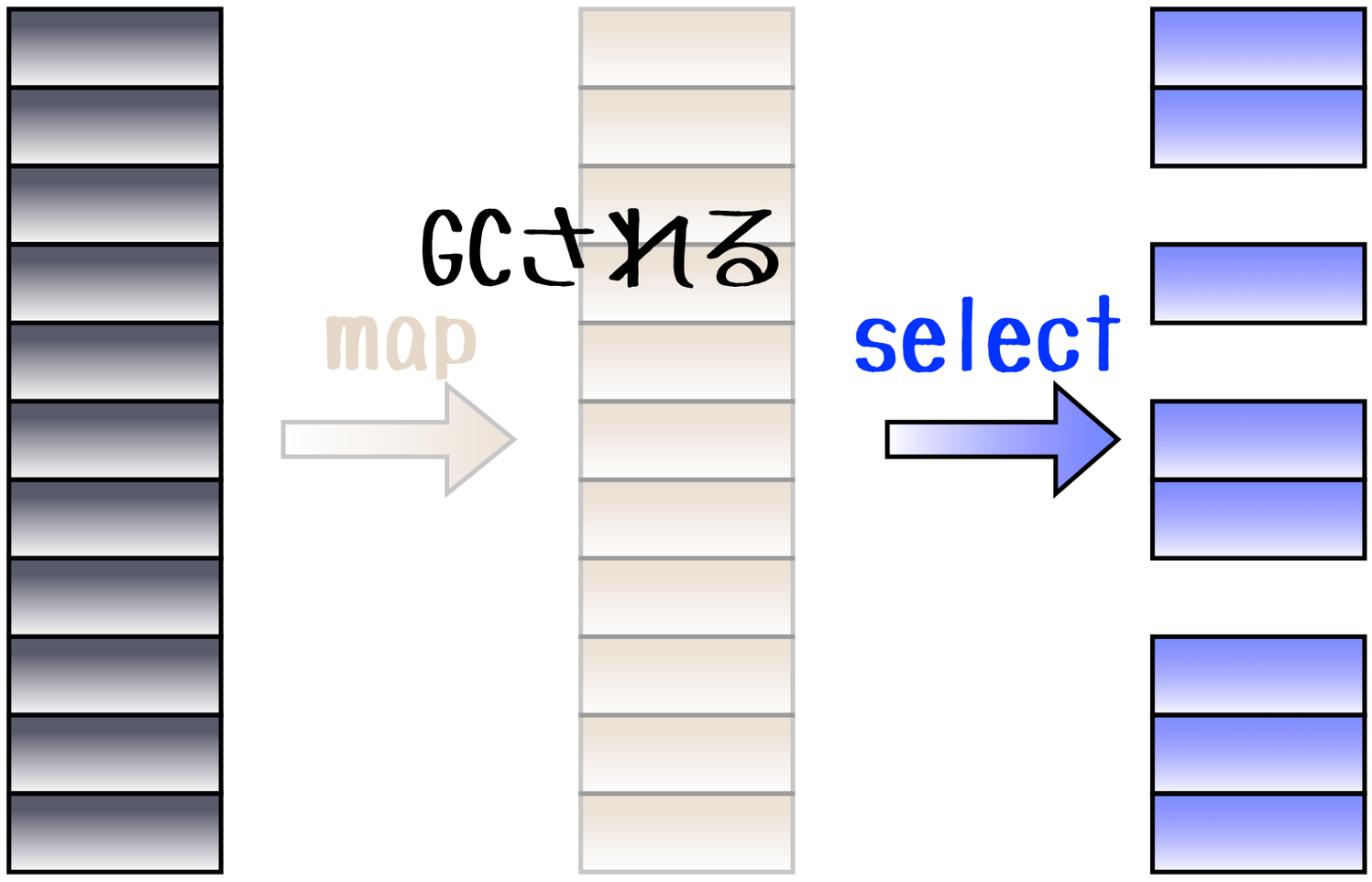
Enumerable version

large_array.map{...}.select{...}



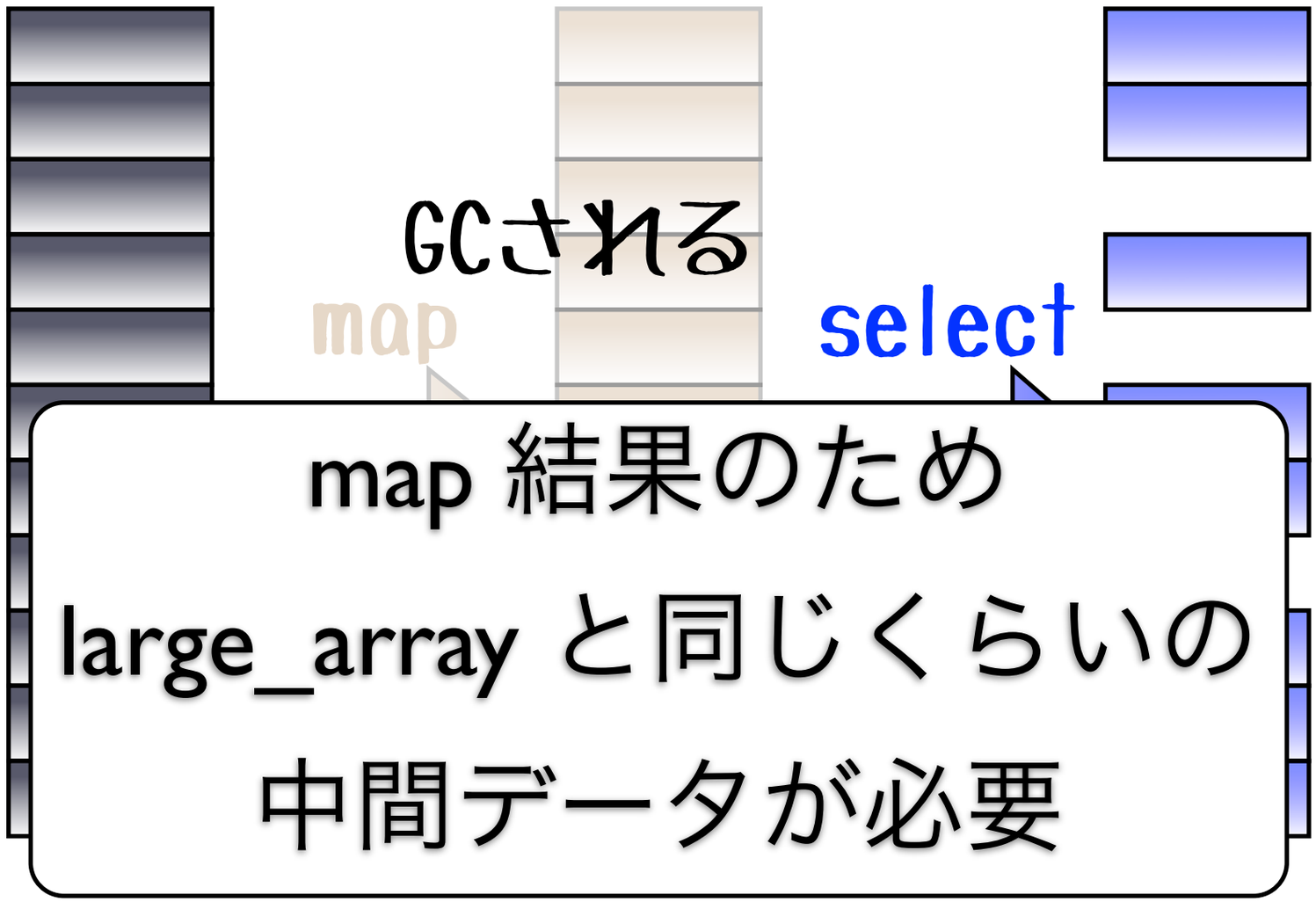
Enumerable version

large_array.map{...}.select{...}



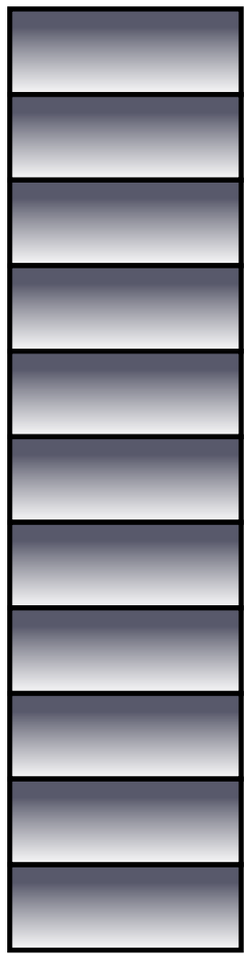
Enumerable version

large_array.map{...}.select{...}



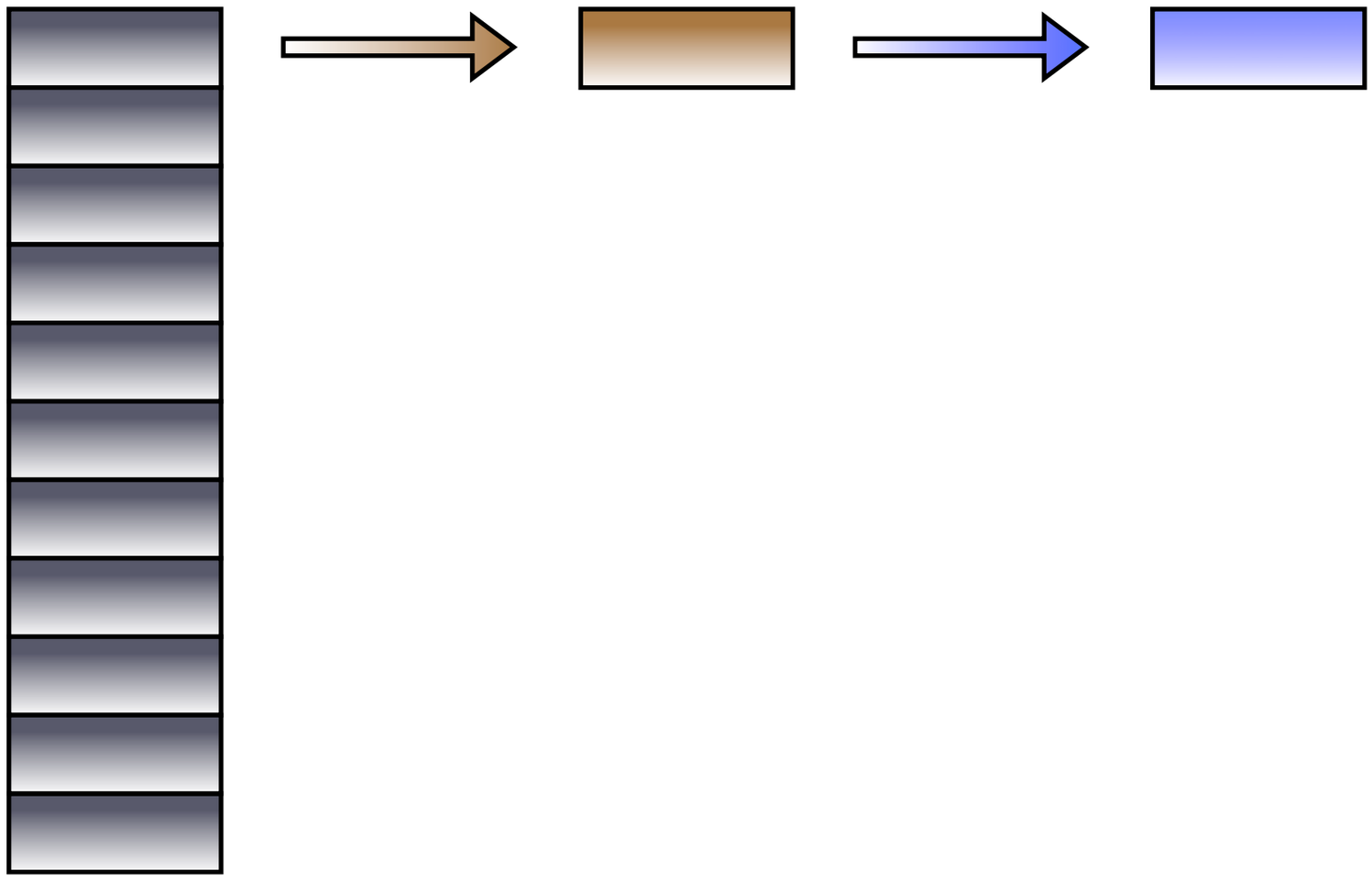
Lazy version

large_array.lazy.map{...}.select{...}.force



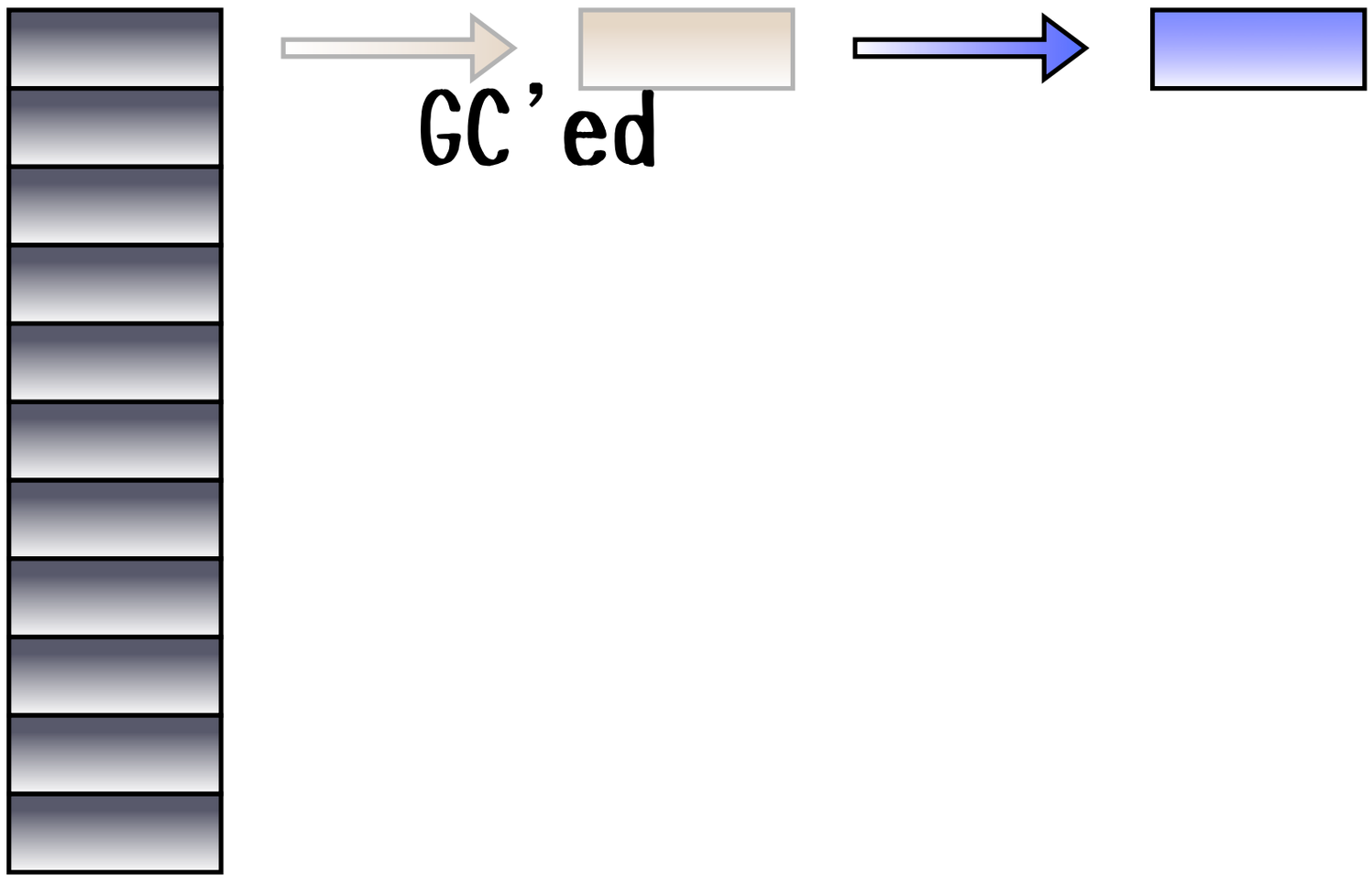
Lazy version

large_array.lazy.map{...}.select{...}.force



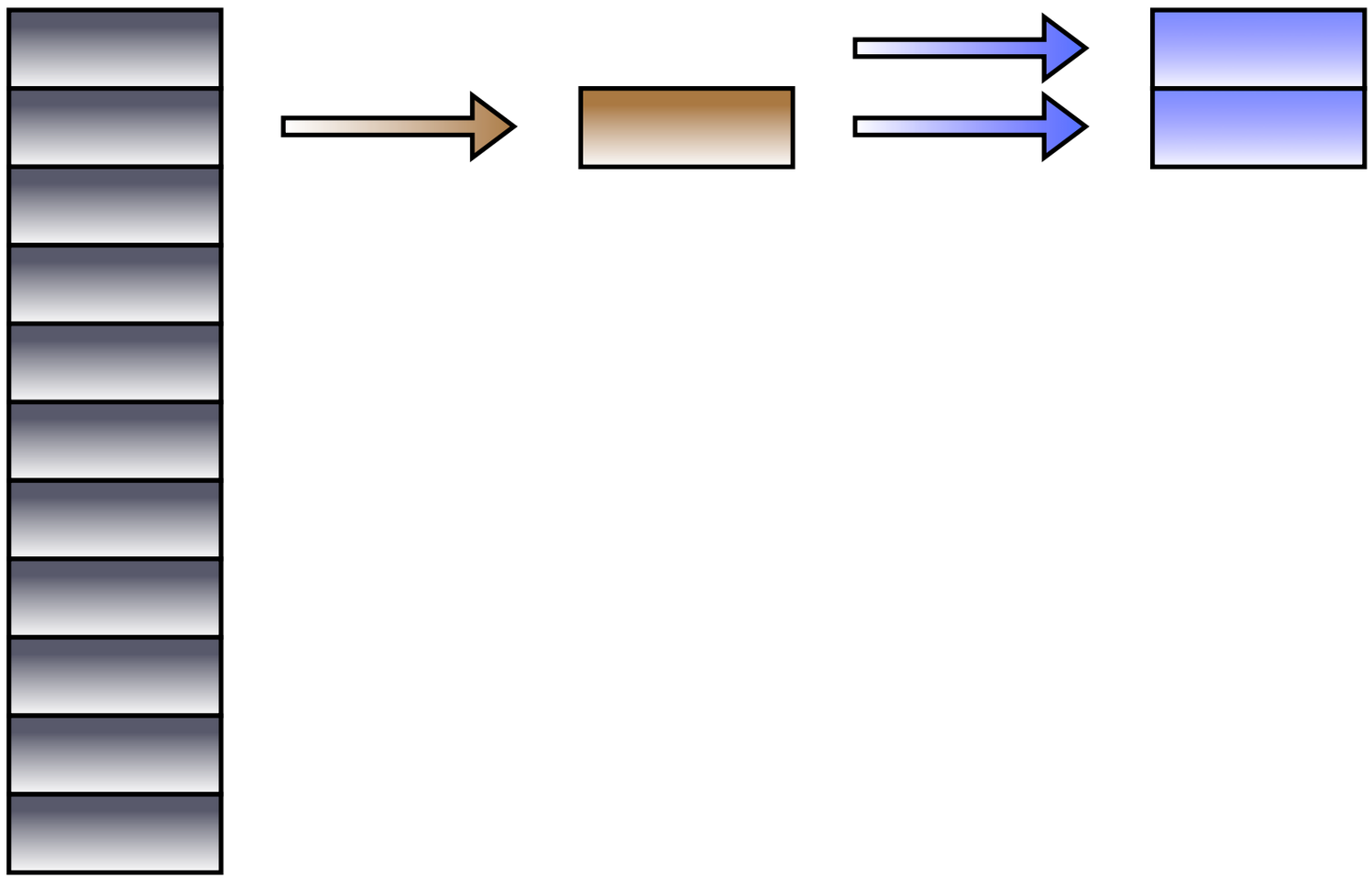
Lazy version

large_array.lazy.map{...}.select{...}.force



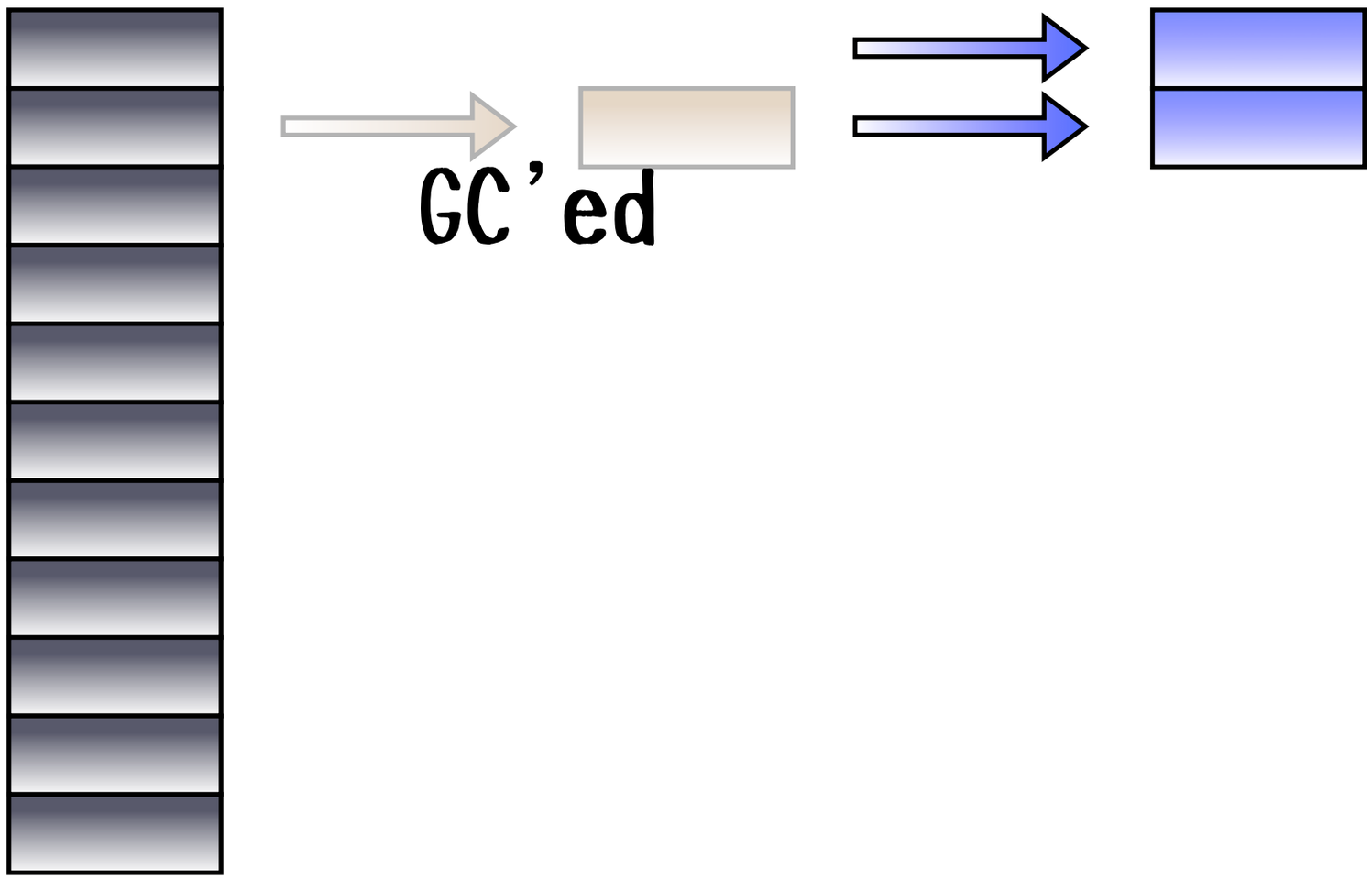
Lazy version

large_array.lazy.map{...}.select{...}.force



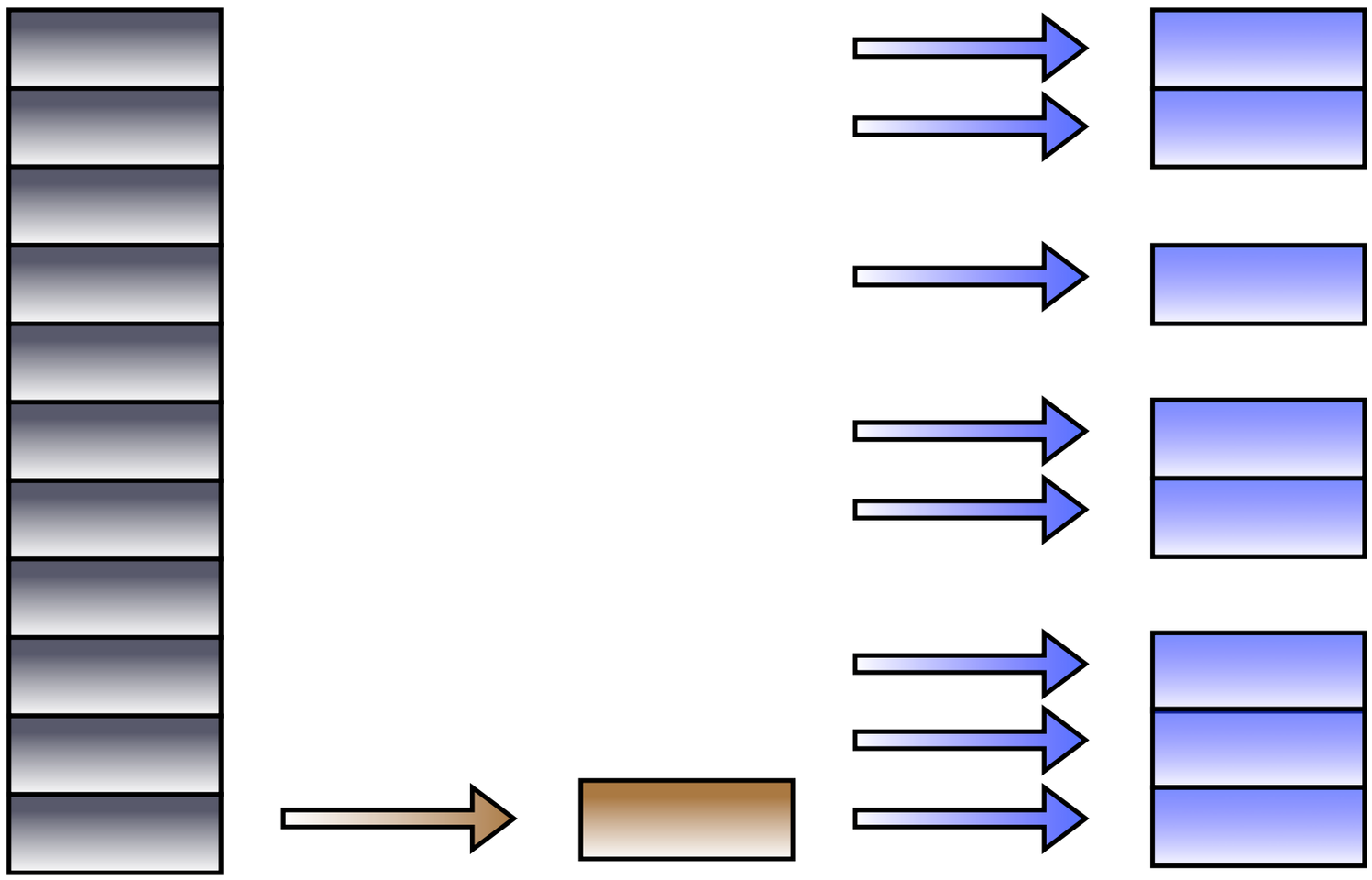
Lazy version

large_array.lazy.map{...}.select{...}.force



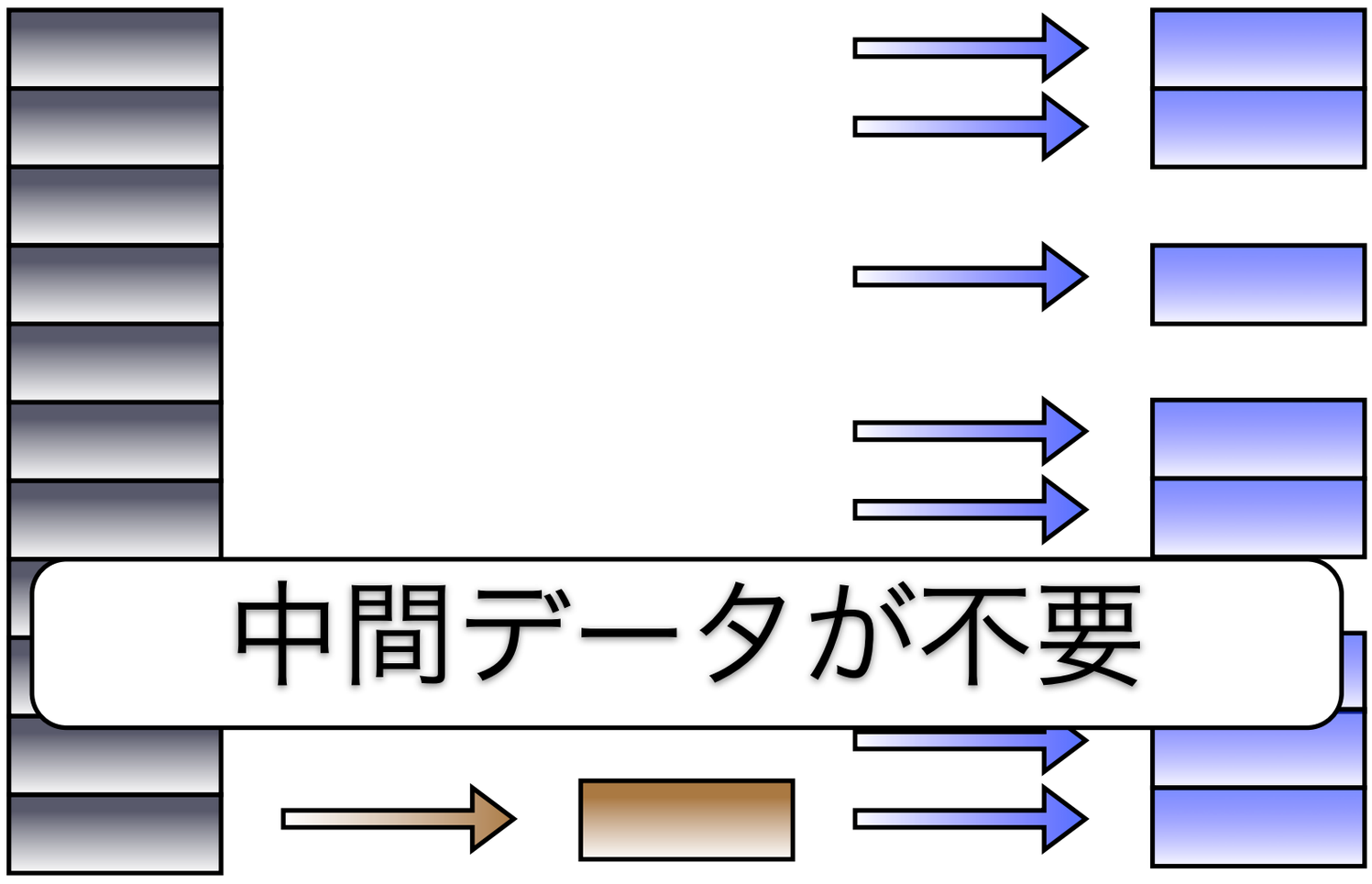
Lazy version

large_array.lazy.map{...}.select{...}.force



Lazy version

large_array.lazy.map{...}.select{...}.force



Lazy Chain Demo

```
>> (0..4).lazy.select{|i|
     p “select:#{i}”; i.event?
   }.map{|i| p “map:#{i}”; i * 2 }.force
“select:0”
“map:0”
“select:1”
“select:2”
“map:2”
“select:3”
“select:4”
“map:4”
=> [0, 4, 8]
```

Lazy with IO

例）$stdin から空行を読むまでの各行の文字数の配列を返す

```
$stdin.lazy.take_while{||
  ! l.chomp.empty?
}.map {|| l.chomp.size }
```

Lazy with IO

Enumerable だと空行まで先に読んでしまう。

Lazy なら1行読むたびに take_while/map のブロックが実行される

→ 逐次処理できる

Lazy List

無限に続く要素を扱える

```
list = (0..Float::INFINITY)
list.map{|i| i * 2}.take(5)
```

=> 返ってこない

```
list.lazy.map{|i| i * 2 }.take(5).force
```

=> [0, 2, 4, 6, 8]

Pitfall of Lazy?

- メモ化されない
  (force を2回呼ぶともう一度 each が呼ばれる)
  - IOなど副作用のある each だと結果が変化するけどいいのかなぁ(いいかも)
- enum_for のように each のかわりになるメソッドを指定できない

Lazy as a Better Enumerator

Lazy は Enumerator の正統進化版

Happy Lazy Programming!

One More Thing!

https://github.com/nagachika/lazy_sample