FUJITSU
shaping tomorrow with you

SYNCDOT®

大量データ処理時における Rubyメモリ使用量削減対策

RubyWorld Conference 2012

株式会社富士通システムズ・イースト
中坊 誠秀

自己紹介

自己紹介

FUJITSU

- 株式会社富士通システムズ・イースト所属
  - http://jp.fujitsu.com/group/feast/
- 2007年より SyncDOT® を**Ruby**で開発
  - http://jp.fujitsu.com/group/feast/services/packages/webmailer/
- **Ruby**歴は2007年からの約5年

FUJITSU

SyncDOT®

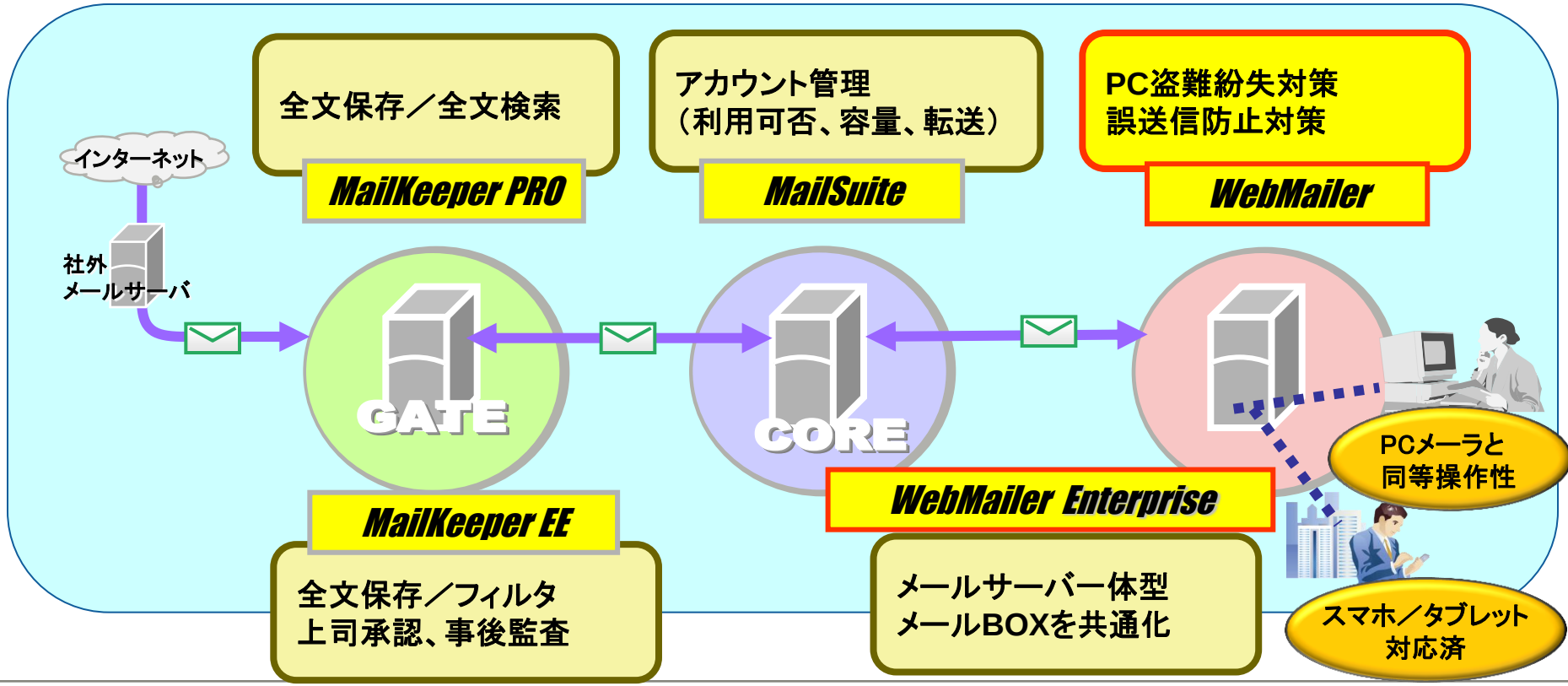
カタログコーナで頒布中

FUJITSU

セキュリティ対策を一気通貫で実現

メールは「組織の神経」、情報リスクは信用に直結

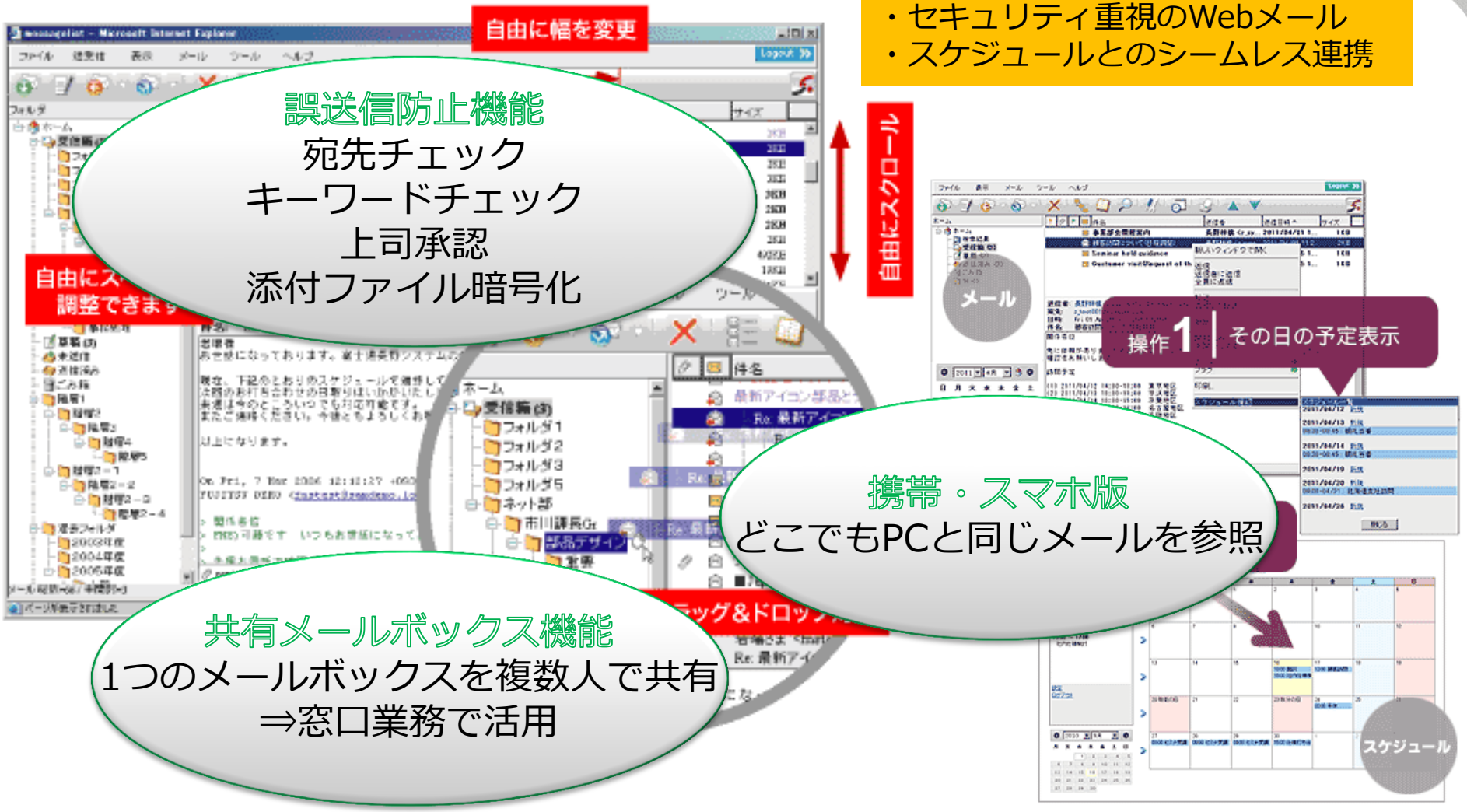
- 情報漏洩対策、誤配信防止、コンプライアンス、フォレンジック対応…



SyncDOT® Web Mailer

FUJITSU

・セキュリティ重視のWebメール
・スケジュールとのシームレス連携



- WebMail機能
- POPサーバ機能
- IMAPサーバ機能
- SMPTサーバ機能

メモリを大量消費する課題がある

まずは、メモリ使用状況の確認が必要

メモリ使用状況調査

メモリ使用状況確認

FUJITSU

```
p `egrep "VmSize|VmPeak" /proc/#$$/status`; p `free`
```

IMAPサーバ接続
login
select
append
fetch
:
close
logout
コネクション解放

増加量が一定ではない

予想通り増加

予想通り増加

予想通り増加

GCのタイミングによって違う？

メモリ使用状況確認

```
p `egrep "VmSize|VmPeak" /proc/#$$/status`; p `free`
GC.start
p `egrep "VmSize|VmPeak" /proc/#$$/status`; p `free`
```

IMAPサーバ接続
login
select
append
fetch
:
close
logout
コネクション解放

メモリ使用量が増加

ある程度減少

ある程度減少

ある程度減少

GCでメモリ使用量が増える？？

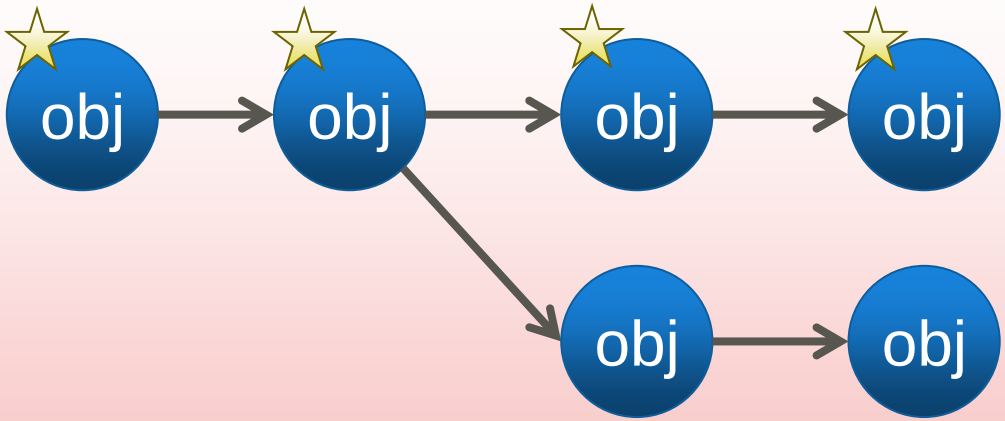
FUJITSU

RubyのGC

RubyのGC

FUJITSU

- RubyのGCは マーク＆スイープ



スイープ処理

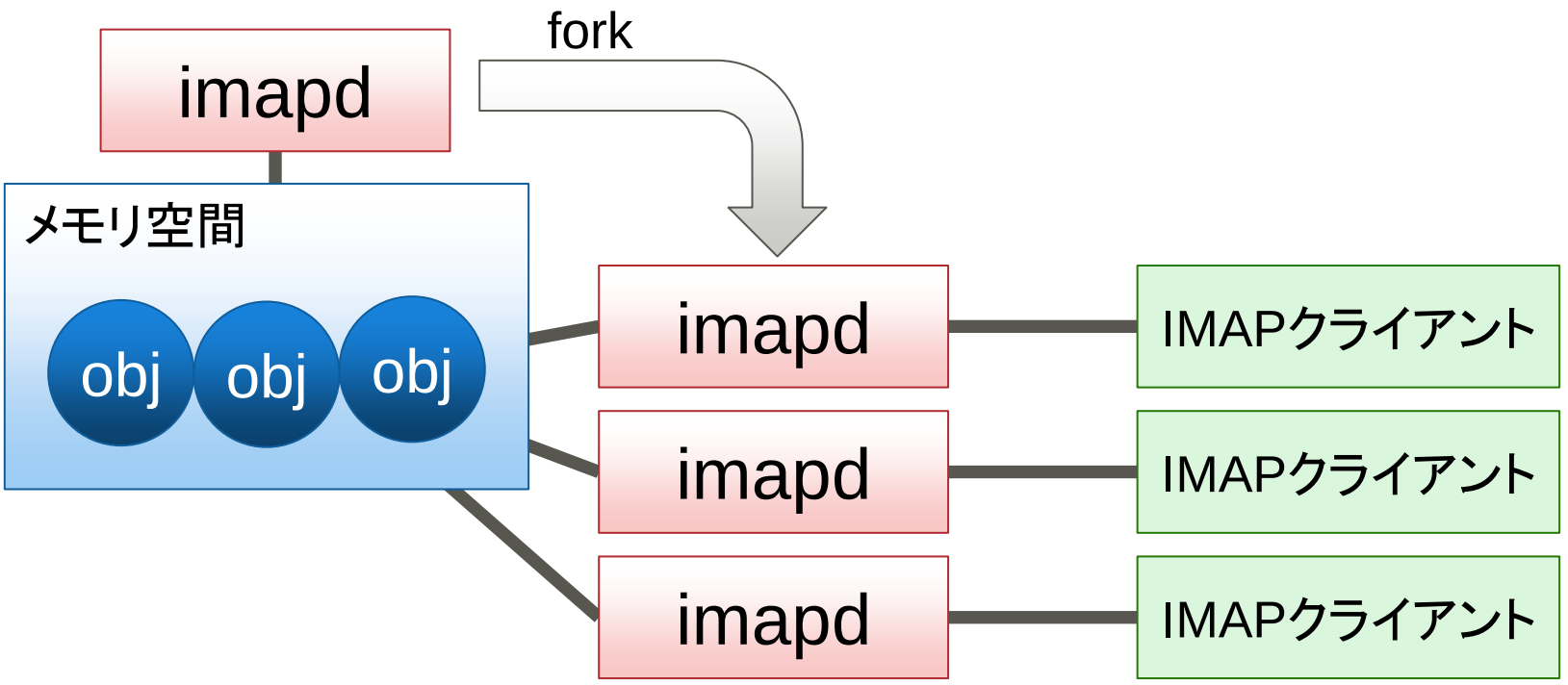
使用していないオブジェクト領域を解放

FUJITSU

IMAPのloginで何してるか？

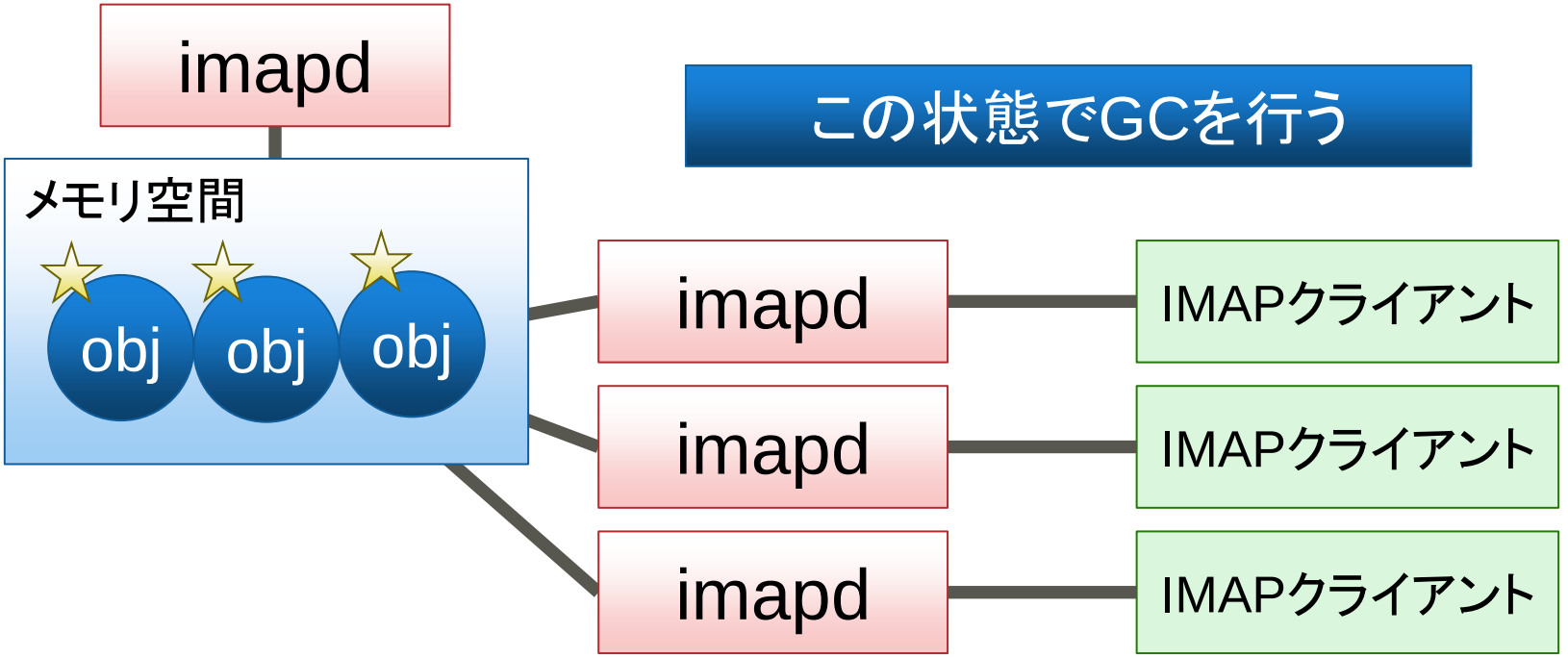
IMAPのログイン処理

FUJITSU



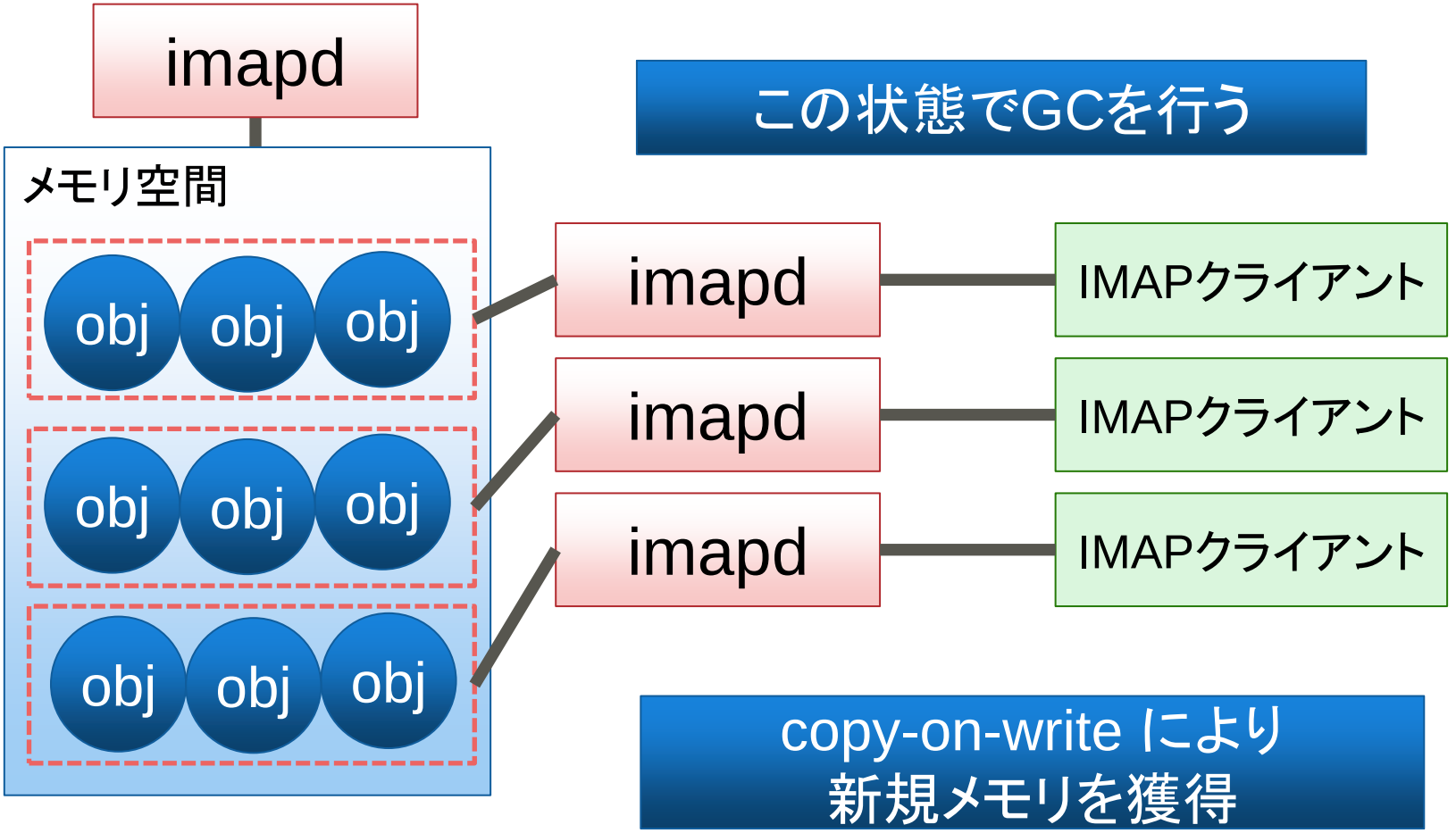
メモリ空間を共有して、子プロセス毎に
クライアントとコネクションを生成する。

IMAPのログイン処理



IMAPのログイン処理

FUJITSU



smapsによるメモリ調査

FUJITSU

- **親imapd起動**

PID,	Size,	Rss,	Shared_Clean,	Shared_Dirty,	Private_Clean,	Private_Dirty,	Swap,	Pss
11816,	22596,	15176,	724,	0,	0,	14452,	0,	14484

- **子プロセスfork**

PID,	Size,	Rss,	Shared_Clean,	Shared_Dirty,	Private_Clean	Private_Dirty,	Swap,	Pss
11816,	22596,	15324,	872,	14020,	0	432,	0,	7477
11849,	22596,	15432,	880,	14020,	0	532,	0,	7582

- **ログイン処理**

PID,	Size,	Rss,	Shared_Clean,	Shared_Dirty,	Private_Clean	Private_Dirty,	Swap,	Pss
11816,	22596,	15324,	872,	2452,	0	12000,	0,	13261
11849,	22632,	16528,	1920,	2452,	4	12152,	0,	13508

Copy-on-write friendly path がある Ruby 1.9系を使う？

Ruby Enterprise Edition を使う？

親プロセスで確保するメモリ利用を最小限にする

メモリを消費している箇所

メモリ消費箇所の特定

FUJITSU

- procfs / free を各メソッド単位で発行して調査

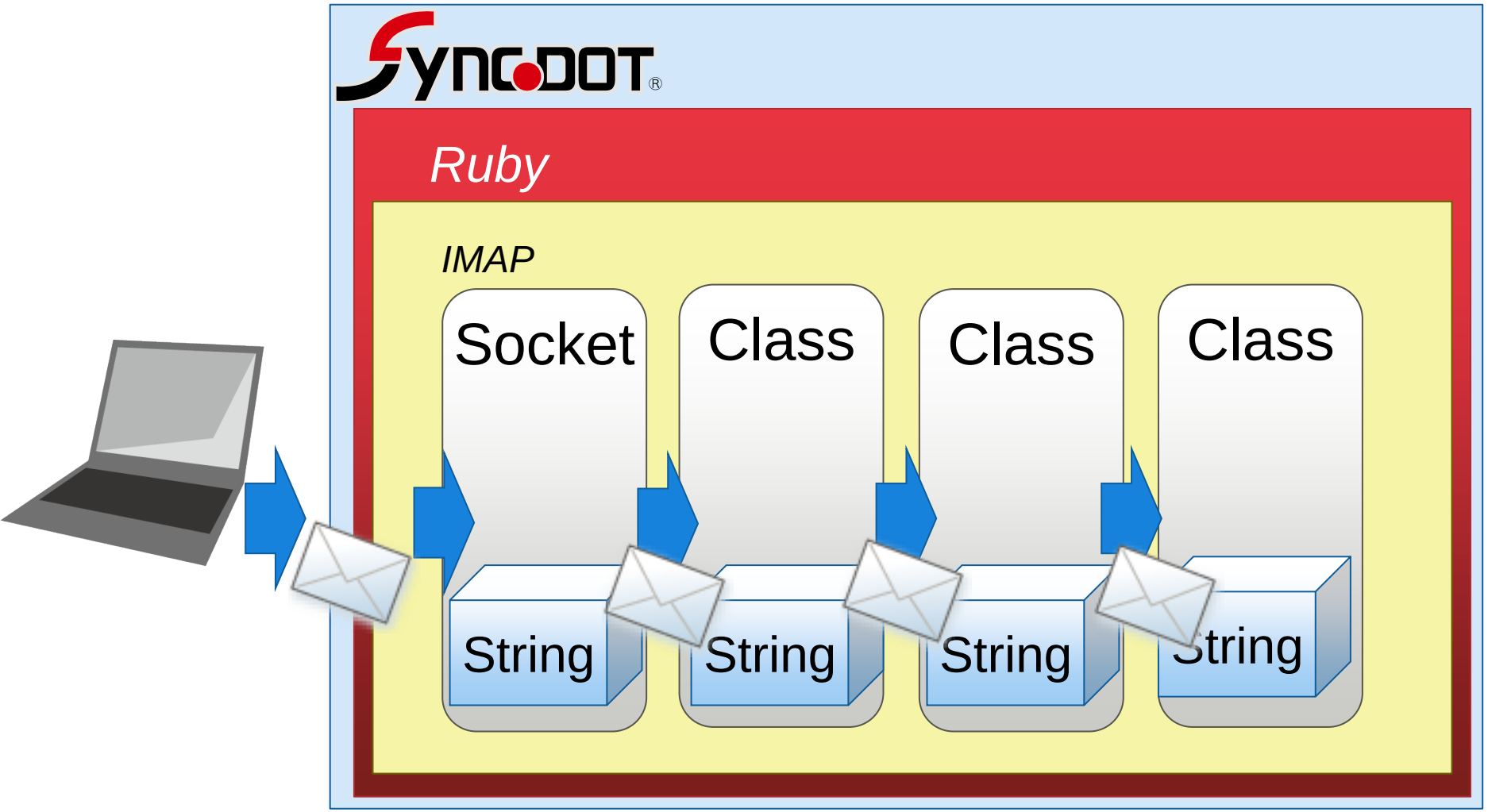
各クラス毎に最適化したデータを持ちまわっていた

ファイルオブジェクトや、大きなStringデータの
ブロック処理後に、メモリ使用量が増加

大量のクラス生成にてメモリ使用量が増加

各クラスに最適化したデータ

FUJITSU

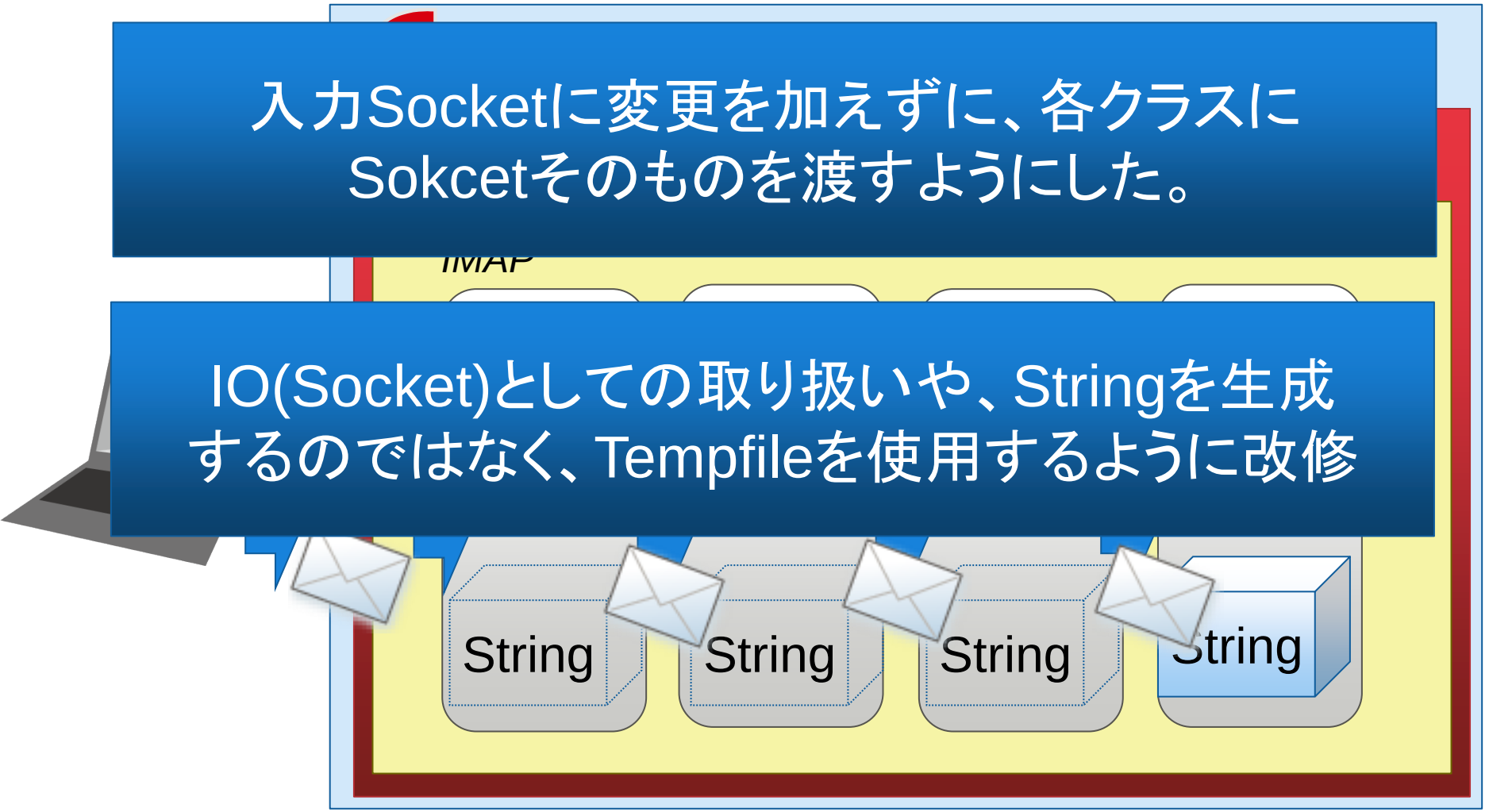


各クラスに最適化したデータ

FUJITSU

入力Socketに変更を加えずに、各クラスに
Sokcetそのものを渡すようにした。

IO(Socket)としての取り扱いや、Stringを生成
するのではなく、Tempfileを使用するように改修



各クラスに最適化したデータ

FUJITSU

- メリット
  - メモリ使用量低下
- デメリット
  - 処理時間が伸びる
  - Disk I/Oが増える

ブロック処理(File)

FUJITSU

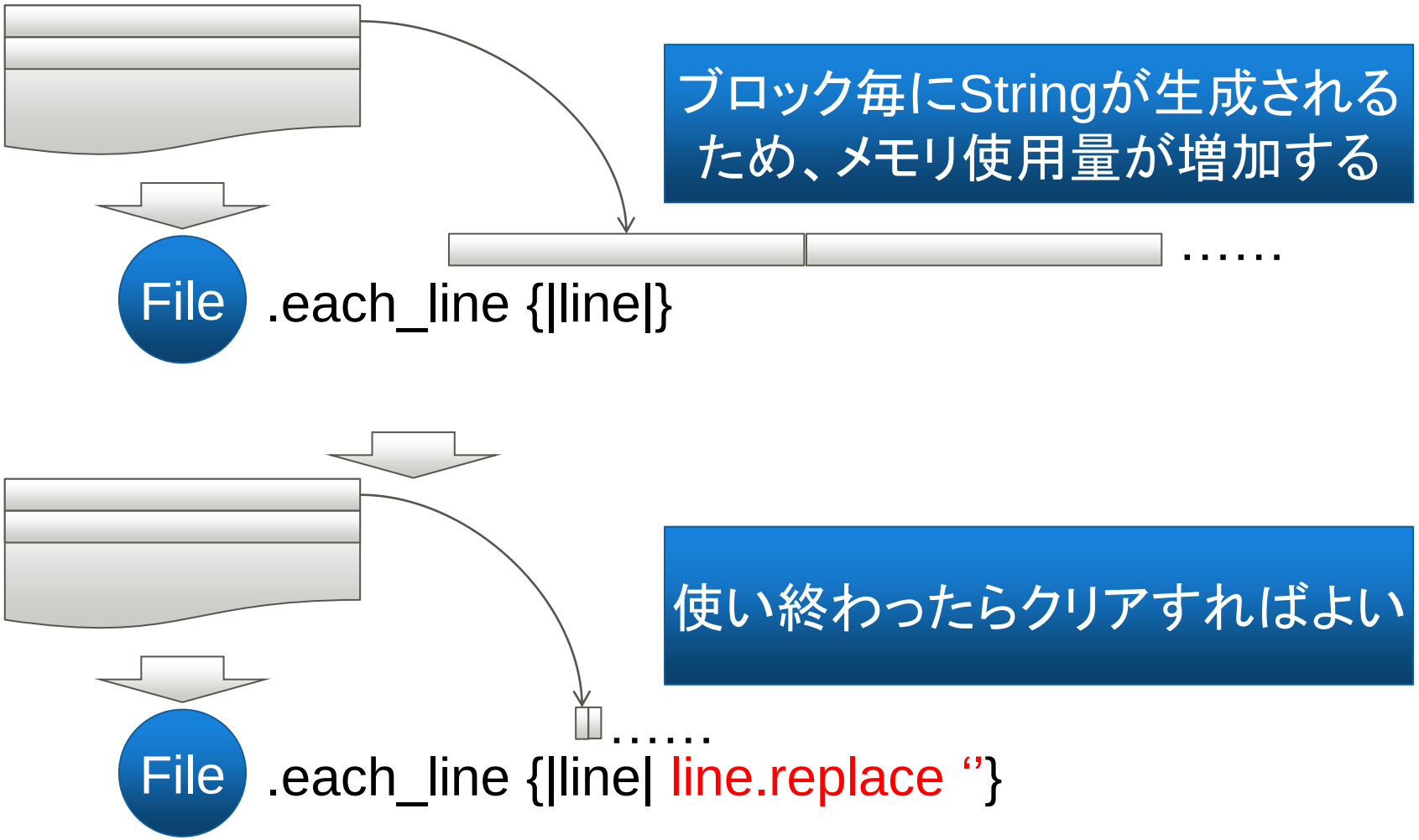
- サンプルプログラム

```
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
File.open(ARGV[0]).each_line{|line|}
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
```

- 実行結果

```
$ ruby1.8 -v
ruby 1.8.7 (2011-06-30 patchlevel 352) [i686-linux]
$ ruby1.8 ./sample1.rb 1m
VmPeak:     3996 kB
VmSize:     3996 kB
VmPeak:     5060 kB
VmSize:     5060 kB
$ ruby1.8 ./sample1.rb 10m
VmPeak:     3996 kB
VmSize:     3996 kB
VmPeak:    10340 kB
VmSize:    10340 kB
```

ブロック処理(File)



ブロック処理(File)

FUJITSU

- サンプルプログラム

```
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
File.open(ARGV[0]).each_line{|line| line.replace ''}
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
```

- 実行結果

```
$ ruby1.8 -v
ruby 1.8.7 (2011-06-30 patchlevel 352) [i686-linux]
$ ruby1.8 ./sample2.rb 1m
VmPeak:     3996 kB
VmSize:     3996 kB
VmPeak:     4004 kB
VmSize:     4004 kB
$ ruby1.8 ./sample2.rb 10m
VmPeak:     3996 kB
VmSize:     3996 kB
VmPeak:     4004 kB
VmSize:     4004 kB
```

ブロック終了時にクリア処理する事により、メモリ使用量を低下させることが出来る

ブロック処理(String)

FUJITSU

- サンプルプログラム

```
str = "#{'a'*1024}¥r¥n"*ARGV[0].to_i
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
str.each_line{|line|}
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
```

- 実行結果

```
$ ruby1.8 -v
ruby 1.8.7 (2011-06-30 patchlevel 352) [i686-linux]
$ ruby1.8 ./sample3.rb 1,000
VmPeak:     5000 kB
VmSize:     5000 kB
VmPeak:     6060 kB
VmSize:     6060 kB
$ ruby1.8 ./sample3.rb 10,000
VmPeak:    14016 kB
VmSize:    14016 kB
VmPeak:    20356 kB
VmSize:    20356 kB
```

ブロック処理(String)

FUJITSU

- **サンプルプログラム**

```
str = "#{'a'*1024}¥r¥n"*ARGV[0].to_i
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
str.each_line{|line| line.replace ''}
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
```

- **実行結果**

```
$ ruby1.8 -v
ruby 1.8.7 (2011-06-30 patchlevel 352) [i686-linux]
$ ruby1.8 ./sample4.rb 1,000
VmPeak:     5000 kB
VmSize:     5000 kB
VmPeak:     5004 kB
VmSize:     5004 kB
$ ruby1.8 ./sample4.rb 10,000
VmPeak:    14016 kB
VmSize:    14016 kB
VmPeak:    14020 kB
VmSize:    14020 kB
```

ブロック終了時にクリア処理する事により、メモリ使用量を低下させることが出来る

ブロック処理後のクリア

FUJITSU

- メリット
  - メモリ使用量低下
- デメリット
  - コードが**Ruby**らしくない

大量のクラス生成

FUJITSU

■ サンプルプログラム

```
class Test
  def initialize(v1,v2,v3)
    @v1 = v1
    @v2 = v2
    @v3 = v3
  end
end
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
Array.new(ARGV[0].to_i).map{Test.new(0,0,0)}
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
```

■ 実行結果

```
$ ruby1.8 sample6.rb 100,000
VmPeak:      3996 kB
VmSize:      3996 kB
VmPeak:     21036 kB
VmSize:     21036 kB
```

クラスで保持するインスタンス変数を限りなく減少させることでメモリ使用量を低下できないか？

大量のクラス生成

FUJITSU

- サンプルプログラム

```
class Test
  def initialize(v1,v2,v3)
    @v = 0b000
    @v = @v | v1*0b100
    @v = @v | v2*0b010
    @v = @v | v3*0b001
  end
  def v1; @v[2]; end
  def v2; @v[1]; end
  def v3; @v[0]; end
end
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
Array.new(ARGV[0].to_i).map{Test.new(0,0,0)}
puts `egrep "VmSize|VmPeak" /proc/#$$/status`
```

- 実行結果

```
$ ruby1.8 sample7.rb 100,000
VmPeak:     3996 kB
VmSize:     3996 kB
VmPeak:    16284 kB
VmSize:    16284 kB
```

インスタンス変数を減少させることでメモリ使用量を低下することが出来る

大量のクラス生成

FUJITSU

- メリット
  - メモリ使用量低下
- デメリット
  - コードが読みづらくなる
  - **Ruby**らしくない

FUJITSU

メモリ消費量を抑えたコーディング

省メモリ対策

FUJITSU

- forkするアプリでは、親プロセスを軽くする
- イテレータ処理する場合は、ブロック終了時にイテレータ変数をクリアしてあげる
- クラス内部で保持する変数は少なくする
- 破壊的メソッドを使用する(gsub!など)
- 可能な限り Ruby1.9 を使用する

Ruby1.8 vs Ruby1.9

FUJITSU

	Ruby1.8	Ruby1.9
1MBファイルのeach_line replaceなし	1,064 [KB]	1,056 [KB]
1MBファイルのeach_line replaceあり	8 [KB]	144 [KB]
1MB String の each_line replaceなし	1,060 [KB]	1,056 [KB]
1MB String の each_line replaceあり	4 [KB]	140 [kb]
10万クラス作成 インスタンス変数3つ	**17,040 [KB]**	**2,348 [KB]**
10万クラス作成 インスタンス変数1つ	**12,288 [KB]**	**2,364 [KB]**

Ruby1.9 はプロセスサイズが大きくなるが、
メモリ使用量は抑える事が出来る。

おまけ①(速度改善)

FUJITSU

- Enumerable#each_with_index は遅い

```
$ time ruby1.8 -e 'Array.new(1000000,"1").each_with_index{|v,idx|}'

real    0m2.080s
user    0m2.068s
sys     0m0.008s
```

```
$ time ruby1.8 -e 'idx=0;Array.new(1000000,"1").each{|v| idx+=1}'

real    0m0.292s
user    0m0.280s
sys     0m0.008s
```

```
$ time ruby1.9 -e 'Array.new(1000000,"1").each_with_index{|v,idx|}'

real    0m0.145s
user    0m0.144s
sys     0m0.000s
```

```
$ time ruby1.9 -e 'idx=0;Array.new(1000000,"1").each{|v| idx+=1}'

real    0m0.121s
user    0m0.116s
sys     0m0.004s
```

おまけ②(速度改善)

FUJITSU

- Symbol#to_proc は遅い

```
$ time ruby1.8 -e 'Array.new(1000000,"1").map(&:to_i)'

real    0m1.589s
user    0m1.580s
sys     0m0.008s
```

```
$ time ruby1.8 -e 'Array.new(1000000,"1").map{|v| v.to_i}'

real    0m0.318s
user    0m0.300s
sys     0m0.016s
```

```
$ time ruby1.9 -e 'Array.new(1000000,"1").map(&:to_i)'

real    0m0.184s
user    0m0.172s
sys     0m0.008s
```

```
$ time ruby1.9 -e 'Array.new(1000000,"1").map{|v| v.to_i}'

real    0m0.224s
user    0m0.212s
sys     0m0.008s
```

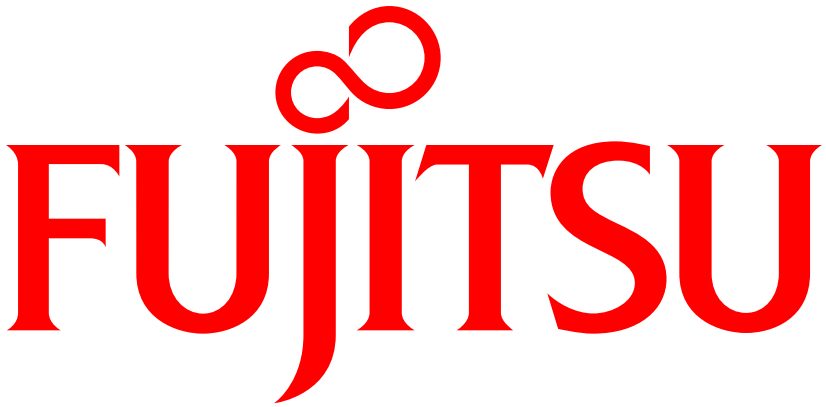
最後に。。。

FUJITSU

今回紹介させていただいた対策は、
独自の調査結果に基づいた対策です。

メモリ削減になる手法がありましたら、
是非ご連絡ください。

ご静聴ありがとうございました。



shaping tomorrow with you