Red Data Tools

楽しく実装すればいいじゃんねー

須藤功平

株式会社クリアコード

沖縄Ruby会議02

2018-03-10

Red Data Tools

プロジェクト

実現したいこと

Rubyで
データ処理

やっていること

データ処理用の
ツールの開発

開発例

- ✓ 各種フォーマットを扱うgem
  - ✓ Apache Arrow, Apache Parquet, CSV, ...
- ✓ 各種パッケージの用意
  - ✓ deb, RPM, Homebrew, ...
- ✓ ...

この話の目的

勧誘

一緒に開発しようぜ！

勧誘方法

- ✓ プロジェクトのポリシーを紹介
  - ✓ 一緒に活動したくなる！
- ✓ 開発の具体例を紹介
  - ✓ 一緒に開発したくなる！

ポリシー1

Rubyコミュニティーを超えて協力する

もちろんRubyコミュニティーとも協力する

Rubyに閉じずに協力

- 他の言語は敵ではない
  - 他の言語がよくなることは
    Rubyがなにかを失うことではない
- みんなよくなったらいいじゃん
  - Rubyも他の言語も

協力例
Apache Arrow

- Pythonの人達他と一緒に C/C++のライブラリーを開発
  - それぞれでバインディングを開発
  - それぞれで同じライブラリーを活用

ポリシー2

非難することよりも
手を動かすことが大事

非難しない

そんなことを
している
時間はない

手を動かす

- ✓ これ、よくないなー
  - ✓ よくすればいいじゃんねー
- ✓ これがないからなー
  - ✓ 作ればいいじゃんねー

ポリシー3

一回だけの活発な活動より
小さくても
継続的に活動することが
大事

一回だけの活発な活動

- ✓ ○○作ったよ！どーん！
  - ✓ すごい！
  - ✓ …数年後…今動かないんだよね…

継続的な活動

- ちまちま〇〇作ってるんだー！
  - がんばってね！今後に期待！
  - …数年後…これは…使える！

継続的な活動のために

- ✓ がんばり過ぎない
  - ✓ 短距離走ではなくマラソン
  - ✓ 途中で休んだっていい
- ✓ 1人で抱え込まない
  - ✓ みんなでやれば途中で休みやすい

ポリシー4

現時点での知識不足は
問題ではない

知識不足？

- ✓ 高速な実装の実現
  - ✓ プログラミング・数学などの高度な知識は便利
- ✓ 今すごい人でも最初はなにも知らなかった
  - ✓ 「今知らないこと」は「始めない理由」にはならない

私たちは学べる

- ✓ 知識は身につく
  - ✓ 活動していく中で自然と
- ✓ 学び方
  - ✓ OSSの既存実装から学習
  - ✓ ドキュメントを読む
  - ✓ 他の人から教えてもらう

ポリシー5

部外者からの非難は
気にしない

部外者からの非難

- ✓ Rubyで頑張ってもアレだよねーとか
  - ✓ 無視する
- ✓ 対応している時間はない

ポリシー6

楽しくやろう！

楽しむ

使っているのは
Rubyだから！

ポリシー

- ✓ Ruby外とも協力
- ✓ 非難するより手を動かす
- ✓ 継続的な活動
- ✓ 知識不足は問題ない
- ✓ 外からの非難は気にしない
- ✓ 楽しくやろう！

開発例

- ✓ ツールの紹介**ではない**
- ✓ 実装の紹介
  - ✓ 開発中に**おっ**と思ったやつとか
  - ✓ 紹介したいやつとか

開発例1

CSV

CSV

- ✓ CSVの読み書きライブラリー
- ✓ 2003年から標準添付
- ✓ 2018年からメンテナンスを引取

digを追加したとき

https://github.com/ruby/csv/pull/15

- ✓ CSV::Table#dig
- ✓ CSV::Row#dig

digの作り方

```
def dig(index, *indexes)
  # ここだけ違う
  value = find_value(index)
  # ↓は共通
  return nil if value.nil?
  return value if indexes.empty?
  value.dig(*indexes)
end
```

CSV::Rowでのfind_value

```
row[index] # => field value
```

digの中ではselfの[]を呼べばよい

[]の呼び方

```
def dig(index, *indexes)
  # インスタンスメソッドでは
  # selfを省略できるからこう？
  value = [index]
  # ↑は配列リテラル
  # ...
end
```

[]の呼び方

```
def dig(index, *indexes)
  # こういうときこそsendで
  # メソッド呼び出し
  value = send(:[], index)
  # 動く
  # ...
end
```

[]の呼び方

```
def dig(index, *indexes)
  # row[index]みたいに書けば
  # よかった
  value = self[index]
  # もちろん動く
  # ...
end
```

selfの[]の呼び方

- ✓ 自分も昔悩んだ気がする
- ✓ 懐かしかったので紹介
- ✓ みんなで開発➔気づきやすい

開発例2

Red Datasets

Red Datasets

- ✓ データを簡単に使えるように！
- ✓ 実験・開発に便利

例：日本語データ欲しい！

```
require "datasets"
# 日本語版Wikipediaの記事データ
options = {language: :ja, type: :articles}
dataset = Datasets::Wikipedia.new(options)
dataset.each do |page| # 全ページを順に処理
  p page.title          # タイトル
  p page.revision.text # 本文
end
# インターフェイスがeachなのがカッコいいんだよ！
```

実装方法

1. データのダウンロード
2. データのパース
3. 順にyield

ダウンロード

```
require "open-uri"
open("https://...") do |input|
  File.open("...", "wb") do |output|
    IO.copy_stream(input, output)
  end
end
```

途中でエラーになったら？

やり直し

または

再開

再開

HTTP
range request

HTTP range request

- リクエスト
  - Range: bytes=#{start}-
- レスポンス
  - 206 Partial Content
  - Content-Range: bytes ...

open-uriとrange request

- ✓ open-uriはrange request未対応
  - ✓ 出力無関係のAPIだからしょうがない
- ✓ どうなるのがいいだろう？
  - ✓ 今度田中さんに相談しよう
  - ✓ Red Data ToolsはRubyもよくしたい
    ポリシー1：コミュニティーを超えて協力

open-uriでrange request

```ruby
# 文字列キーはHTTPヘッダーになる
options = {"Range" => "bytes=#{start}-"}
open("...", options) do |input|
  # レスポンスが200か206かはわからない
  File.open("...", "ab") do |output|
    IO.copy_stream(input, output)
  end
end
```

大きなデータの扱い

- ✓ 時間がかかる
- ✓ あとどのくらいか気になる
  - ✓ ちゃんと動いているよね…？

あとどのくらい？

プログレスバー

プログレスバーの実装

```
n = 40
1.upto(n) do |i|
  print("\r|%-*s|" % [n, "*" * i])
  sleep(0.1)
end
puts
```

sprintfフォーマット

%-*s

- %：フォーマット開始
- -：左詰め
- *：引数で幅を指定
- s：対象は文字列

sprintfフォーマット

```
"%-*s" % [n, "*" * i]
```

- ✓ 幅はn桁
- ✓ "*" * iを左詰め

open-uriでプログレスバー

```
length = nil
progress = lambda do |current|
  ratio = current / length.to_f
  print("\r|%-10s|" % ["*" * (ratio * 10).ceil])
end
open(uri,
     content_length_proc: ->(l) {length = l},
     progress_proc: progress) do |input|
  # ...
end
puts
```

もっとプログレスバー

- バックグランド化したら？
  - 表示して欲しくない
    ヒント：プロセスグループ
- リダイレクトしているときは？
  - 表示して欲しくない
    ヒント：IO#tty?
- プログレスバーの表示幅は？
  ヒント：io/console/size

開発例3

Apache Arrow
Red Arrow

Apache Arrow

- ✓ インメモリーデータ分析用データフォーマット
  ほぼ固まってきた
- ✓ インメモリーデータ分析用高速なデータ操作実装
  徐々に実装が始まっている
- ✓ 今、すごくアツい！

Apache Arrowの特徴

データ交換コストが低い

低データ交換コスト

- ✓ 複数システムで協力しやすい
  - ✓ Rubyでデータ取得➔Pythonで分析
- ✓ 徐々にRubyを使えるところを増やせる

Apache Arrowの利用例

- ✓ Scala🤝Python
  - ✓ Apache Spark
- ✓ CPU🤝GPU
  - ✓ https://github.com/gpuopenanalytics/libgdf
- ✓ CPU🤝FPGA
  - ✓ https://github.com/johanpel/fletcher

Apache ArrowをRubyでも！

- ✓ バインディングを本体で開発
  - ✓ 本体の開発チームに入った
  - ✓ GObject Introspection(GI)を利用
- ✓ GIを使うとRuby以外のバインディングも自動生成できる

ポリシー1：コミュニティーを超えて協力

Red Arrow

- ✓ Apache Arrowの
  Rubyバインディング
- ✓ GIベースのバインディングに
  Ruby特有の機能をプラス

Ruby特有の機能例

スライスAPI

スライス

データの一部を切り出す

```
(0..10).to_a.slice(2)
# => 2
(0..10).to_a.slice(2..4)
# => [2, 3, 4]
(0..10).to_a.slice(2, 4)
# => [2, 3, 4, 5]
```

Red Arrowのスライス

```
table.slice(2)
# 2行目だけのテーブル
# Array#sliceと違う
```

Red Arrowのスライス

```
table.slice(2..4)
# 2,3,4行目だけのテーブル
# Array#sliceと同じ
```

Red Arrowのスライス

```
table.slice(2, 4)
# 2,4行目だけのテーブル
# Array#sliceと違う
table.slice(2, 4, 6, 8)
# 2,4,6,8行目だけのテーブル
# Array#sliceと違う
```

Red Arrowのスライス

```
table.slice([2, 4])
# 2,3,4,5行目だけのテーブル
# Array#slice(2, 4)と同じ
```

Red Arrowのスライス

```
table.slice([true, false] * 5)
# 0,2,4,6,8行目だけのテーブル
```

Red Arrowのスライス

```
table.slice do |slicer|
  slicer.price >= 500
end
# priceカラムの値が500以上の
# 行だけのテーブル
```

Red Arrowのスライス

```
table.slice do |slicer|
  (slicer.price >= 500) &
    (slicer.is_published)
end
# price カラムの値が500以上かつ
# is_published カラムの値がtrueの
# 行だけのテーブル
```

Red Arrowのスライス

Active Recordみたいなfluent inteface
もブロック内で式を書く方がRubyっぽいん
じゃないかと思うんだよねー
fluent interfaceはORを書きにくいからさー

Fluent interface:
http://bliki-ja.github.io/FluentInterface/

ブロック内で条件式

```ruby
class Arrow::Table
  # table.slice {|slicer| ...}の実現
  def slice(*slicers)
    if block_given?
      slicer = yield(Slicer.new(self))
      slicers << slicer.evaluate
    end
    # ...
  end
end
```

ブロック内で条件式

```
class Arrow::Slicer
  def initialize(table)
    @table = table
  end

  # slicer.priceの実現
  def method_missing(name, *args, &block)
    ColumnCondition.new(@table[name])
  end
end
```

ブロック内で条件式

```
class ColumnCondition < Condition
  def initialize(column)
    @column = column
  end

  # slicer.price >= 500の実現
  def >=(value)
    GreaterEqualCondition.new(@column, value)
  end
end
```

ブロック内で条件式

```
class GreaterEqualCondition < Condition
  def initialize(column, value)
    @column = column
    @value = value
  end

  # slicer.price >= 500を評価
  def evaluate
    # ホントはC++で実装
    @column.collect {|value| value >= @value}
  end
end
```

ブロック内で条件式

```
class Condition
  # (slicer.price >= 500) & (...)の実現
  def &(condition)
    AndCondition.new(self, condition)
  end
end
```

条件式のポイント

- ✓ 遅延評価
  - ✓ ❌各要素毎にブロックを評価
  - ✓ ブロックは一回だけ評価
  - ✓ ブロックで指定した条件は
    コンパイルしてC++実装で実行

Red Arrowと外の世界

- ハブになるといいかも！
  - 各種データと変換可能に
  - 各種オブジェクトと変換可能に
- Ruby間の連携を推進
  - 今は互換性がないライブラリー
    ➔連携できるように！

データ変換

- ✓ Arrow::Table.load
  - ✓ データの読み込み
- ✓ Arrow::Table#save
  - ✓ データの書き出し

データ変換例

```
# CSVを読み込んで
table = Arrow::Table.load("a.csv")
# Arrowで保存
table.save("a.arrow")
# Parquetで保存
table.save("a.parquet")
```

CSVの読み込み

- ✓ CSVは広く使われている
  - ✓ CSVなデータを簡単に使えると捗る
- ✓ 難しいところ
  - ✓ データ定義が緩い
    例：カラムの型情報がない

Red Arrowでの読み込み

- ✓ 2パスで処理
  a. 全部処理して各カラムの型を推定
  b. 推定した型でArrowのデータに変換
- ✓ 😅時間がかかる
  - ✓ 読み込んだデータを別の形式に変換して再利用する使い方を想定
    だから、まぁ、いいかなぁって

型の推定

```
candidate = nil
column.each do |value|
  case value
  when nil; next # ignore
  when "true", "false", true, false; c = :boolean
  when Integer; c = :integer
  # ...
  else; c = :string # わからなかったら文字列
  end
  candidate ||= c
  candidate = :string if candidate != c # 混ざったら文字列
  break if candidate == :string # 文字列なら終わり
end
candidate || :string # わからなかったら文字列
```

オブジェクト変換

- Numo::NArray, NMatrix
  - 既存の多次元配列オブジェクト
- PyCall経由でPyArrow
  - Rubyでデータ作成➡Pythonで処理
- GDK Pixbuf
  - 画像オブジェクト

オブジェクト変換例1

```
# PNG画像を読み込み
pixbuf = GdkPixbuf::Pixbuf.new(file: "a.png")
# Arrow経由でNumo::NArrayに変換
narray = pixbuf.to_arrow.to_narray
# 不透明に（アルファ値（透明度）を0xffに）
narray[true, true, 3] = 0xff
# Arrow経由でGdkPixbuf::Pixbufに変換
no_alpha_pixbuf = narray.to_arrow.to_pixbuf
# GIF画像として保存
no_alpha_pixbuf.save(filename: "a-no-alpha.gif")
```

Pixbuf→Arrow

```
class GdkPixbuf::Pixbuf
  def to_arrow
    bytes = read_pixel_bytes # ピクセル値
    buffer = Arrow::Buffer.new(bytes)
    # 高さ、幅、チャンネル数の3次元配列
    # チャンネル数：RGBAだと4チャンネル
    shape = [height, width, n_channels]
    # バイト列なのでUInt8
    Arrow::Tensor.new(Arrow::UInt8DataType.new,
                      buffer, shape)
  end
end
```

ゼロコピーの実現

- ゼロコピー
  - 同じメモリー領域を参照し、
    コピーせずに同じデータを利用
  - 速い！！！
- データ
  - バイト列へのポインター
  - RubyではRSTRING_PTR(string)

Stringとゼロコピー

```
/* pointerの内容をコピー */
rb_str_new(pointer, size);
/* pointerの内容を参照：ゼロコピー */
rb_str_new_static(pointer, size);
```

StringとゼロコピーとGC

```
arrow_data = /* ... */;
/* ゼロコピー */
rb_str_new_static(arrow_data, size);
/* arrow_dataはいつ、だれが開放？ */
```

Rubyでゼロコピー

- ✓ ❌`rb_str_new_static()`だけ
  - ✓ メモリー管理できない
- ✓ メモリー管理する何かが必要

Red Arrowでゼロコピー

- GBytesを利用
- GBytes
  - GLib提供のバイト列オブジェクト
  - リファレンスカウントあり
- RubyだとGLib::Bytes

GLib::Bytes#to_s

```
static VALUE rgbytes_to_s(VALUE self) {
  GBytes *bytes = RVAL2BOXED(self, G_TYPE_BYTES);
  gsize size;
  gconstpointer data =
    g_bytes_get_data(bytes, &size);
  /* ゼロコピーでASCII-8BITな文字列を生成 */
  VALUE rb_data = rb_enc_str_new_static(
    data, size, rb_ascii8bit_encoding());
  rb_iv_set(rb_data, "@bytes", self); /* GC対策 */
  return rb_data;
}
```

GLib::Bytes#initialize

```
static VALUE rgbytes_initialize(VALUE self, VALUE rb_data) {
  const char *pointer = RSTRING_PTR(rb_data);
  long size = RSTRING_LEN(rb_data);
  GBytes *bytes;
  if (RB_OBJ_FROZEN(rb_data)) { /* ゼロコピー */
    bytes = g_bytes_new_static(pointer, size);
    rb_iv_set(self, "source", rb_data); /* GC対策 */
  } else { /* コピー */
    bytes = g_bytes_new(pointer, size);
  }
  G_INITIALIZE(self, bytes);
  return Qnil;
}
```

GBytesはすでに使っていた

```
class GdkPixbuf::Pixbuf
  def to_arrow
    # ピクセル値はGLib::Bytes
    bytes = read_pixel_bytes
    buffer = Arrow::Buffer.new(bytes)
    shape = [height, width, n_channels]
    Arrow::Tensor.new(Arrow::UInt8DataType.new,
                      buffer,
                      shape)
  end
end
```

オブジェクト変換例2

```
# RelationをArrow形式に
users = User
  .where("age >= ?", 20)
  .to_arrow
```

Relation→Arrow

```ruby
module ArrowActiveRecord::Arrowable
  def to_arrow(batch_size: 10000)
    in_batches(of: batch_size).each do |relation|
      column_values_set = relation.pluck(*column_names).transpose
      data_types.each_with_index do |data_type, i|
        column_values = column_values_set[i]
        arrow_batches[i] << build_arrow_array(column_values, data_type)
      end
    end
    # ...
    Arrow::Table.new(...)
  end
end
```

Red Chainer

ChainerのRuby移植

Chainer

- ✓ 深層学習フレームワーク
  - ✓ Pythonのみで実装
- ✓ 移植しやすい
  - ✓ Pythonのみで実装されているから

Red Chainerサンプル

```
model = Chainer::Links::Model::Classifier.new(MLP.new(args[:unit], 10))
optimizer = Chainer::Optimizers::Adam.new
optimizer.setup(model)
train, test = Chainer::Datasets::Mnist.get_mnist

train_iter = Chainer::Iterators::SerialIterator.new(train, args[:batchsize])
test_iter = Chainer::Iterators::SerialIterator.new(test, args[:batchsize], repeat: false, shuffle: false)

# ...
```

横に長い

Chainerのサンプル

```
import chainer
import chainer.links as L

model = L.Classifier(MLP(args.unit, 10))
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)
train, test = chainer.datasets.get_mnist()
train_iter = chainer.iterators.SerialIterator(train, args.batchsize)
test_iter = chainer.iterators.SerialIterator(test, args.batchsize,
                                             repeat=False, shuffle=False)
```

こっちも横に長い

Pythonでの短く仕方

```
# 通常
import chainer
model = chainer.links.Classifier(MLP(args.unit, 10))

# Lでショートカット
import chainer.links as L
model = L.Classifier(MLP(args.unit, 10))
```

Rubyでの短く仕方

```ruby
# 通常
model = Chainer::Links::Model::Classifier.new(...)

# Lでショートカット
L = Chainer::Links
model = L::Model::Classifier.new(...)
```

PythonとRubyの違い

- Python
  - ファイル内でだけLが有効
  - ファイル単位でネームスペース
- Ruby
  - グローバルにLが有効
  - 微妙！

Rubyらしく短く

```ruby
# Rubyのネームスペースの仕組みはモジュール
Module.new do
  include Chainer::Links
  model = Model::Classifier.new(...)
end
```

開発したくなった？

- ✓ 楽しそう！
  - ✓ 一緒に開発しようぜ！
- ✓ やっぱRubyでデータ扱いたい！
  - ✓ 一緒に開発しようぜ！
- ✓ レベルアップしたい！
  - ✓ 一緒に開発しようぜ！

レベルアップへの道

- ✓ 😅NG集をたくさん覚える
- ✓ 😀よいコードにたくさん触れる

よいコード

Red Data Toolsに
たくさんある！

一緒に開発

- ✓ 🌏どこからでもオンラインで
  - ✓ GitHubとGitter（チャット）を使用
- ✓ ➕東京は月一オフラインで
  - ✓ 🔍「OSS Gate Red Data Tools」

Join Red Data Tools!

- Webサイト
  - https://red-data-tools.github.io/ja/
- Gitter（チャット）
  - https://gitter.im/red-data-tools/ja
- GitHub
  - https://github.com/red-data-tools/