会誌「情報処理」Vol.63 No.2（Feb. 2022）「デジタルプラクティスコーナー」

特集号招待論文

Apache ArrowによるRubyのデータ処理対応の可能性

村田賢太[1, 3]　須藤功平[2, 3]

[1]（株）Speee　[2]（株）クリアコード　[3]Red Data Tools

RubyはWebシステムの記述言語として高い生産性を発揮し，Web業界では広く浸透している．一方，分析的データ処理への対応が弱いため，データ処理分野ではほとんど利用されていない．昨今のDX推進などの流れから，Rubyで書かれた既存システムのデータ処理への対応が近い将来必要となるだろう．そのような要求に対応するためには，前もってRubyを分析的データ処理に対応させる必要がある．本稿では，Rubyを分析的データ処理に対応させる手段としてApache Arrowが有効であることを示す．Apache Arrowは，既存のデータ処理コンポーネント間のデータ連携の非効率性を解消するために提案された，データフォーマットとAPIである．RubyをApache Arrowに対応させることで，分析的データ処理に対応できるだけでなく，データ処理分野における先進的な取り組みにRubyからアクセスできるようになる．

1. Rubyを分析的データ処理に対応させる目的

1.1 データ処理分野におけるRubyの位置付けと課題

プログラミング言語Ruby[1]は，Ruby on Rails[2]によるWebアプリケーションの記述言語として高い生産性を発揮できるとして世界中で人気を集め，これまでに数多くの利用実績がある（[2]に数十万という数字が記載されている）．ここ数年のWebフロントエンド技術の進化によってWebアプリケーションの作り方が変わったことで，Ruby on Railsが新規開発で採用される機会は減ってきているものの，過去に開発されたRailsアプリケーションがいまも現役稼働している例は少なくないだろう．Railsアプリケーションの例としてよく名前が挙がるCookpad[3]，GitHub[4]，Shopify[5]のような多数のユーザを獲得している有名サービス以外でも，これまでのRuby on Railsの人気を考慮すると，無名の企業の社内システムとして稼働しているRailsアプリケーションが多数存在することが想像できる．さらに，最近になってRailsアプリケーションに対して有効なRuby用のJITコンパイラ[6]が試作されており，今後もRubyとRuby on Railsの利用が続く可能性は高い．

一方，データ処理分野においてRubyはまったくと言ってよいほど取り上げられることがない．Googleなどで“Best Programming Language for Data Science”のようなキーワードを検索して出てくる[7]のようなサイトにはRubyはまったく出てこない．このようなサイトで必ず名前が挙がるPython [8]やR [9]と比べると，Rubyは特に分析的なデータ処理への対応が弱いことに気づくだろう．第一に，Apache Spark [10]などの分析用のデータ処理コンポーネントをRubyから利用するためには，データの受け渡しに対応する必要がある．追加の開発コストを避けるためにJSON[11]のような共通フォーマットを利用すると，データのシリアライズとデシリアライズの処理コストが発生してしまい，扱うデータ量が大きい場合には無視できない問題となる．

企業のDX推進が望まれている昨今の潮流[12]により，既存の業務アプリケーションを分析的データ処理コンポーネントと連携させることで機械学習の利用やビッグデータの活用に対応していくことが必要になるだろう．この流れは，既存のRailsアプリケーションにも同様に当てはまる．そのため，Rubyを分析的データ処理に適応させていくことが急務である．

1.2 Apache ArrowによるRubyの分析的データ処理への対応

Rubyが分析的データ処理に適応していくためには次の2つの取組みの両方が必要である．

（1）Rubyで利用可能なデータ分析ツールを拡充させる
（2）Ruby以外の言語で実装されたデータ処理コンポーネントとの連携を強化する

ここで問題となるのが両者に対応するための開発リソースである．Rubyを分析的データ処理で利用する事例がある程度増えないと，恒常的に確保できる開発者の人数や費用が限られてしまうため，対応作業を少人数で進めていかなければならない．可能な限り少ない手間でRuby用のツールの拡充と多数のデータ処理コンポーネントとの連携を実現し，それらが高いパフォーマンスを発揮できることが求められる．そして，少数の開発者であっても機能開発を継続できることが望ましい．

本稿では，このような制約のもとでRubyを分析的データ処理に対応させていく手段として，RubyをApache Arrow[13]に対応させることが有効であることを述べる．本稿は次のように構成される．第2章では分析的データ処理におけるApache Arrowの役割について説明する．第3章ではRubyをApache Arrowに対応させる取り組みとしてRed Arrow[14]を紹介し，その有効性を示す．第4章で関連技術について述べ，第5章でRubyのデータ処理対応についての今後の展望を述べて論文をまとめる．

2. 分析的データ処理においてApache Arrowが果たす役割

2.1 Apache Arrowが登場した背景

一般に，データ処理システムは複数のコンポーネントを組み合わせて構成され，分散システムになる場合も多い[15]．ここでコンポーネントと呼んでいる対象は，データ処理の一部の役割を担うライブラリやミドルウェアである．たとえば，データベースは，データを検索可能な状態で保存し，問合せに対して適切なデータを取捨選択して返す役割を担うデータ処理コンポーネント

である．データベースにはトランザクション処理に向いているものと分析的な処理に向いているものとがあり，どちらか一方に特化している場合が多い．たとえばリレーショナルデータベースはトランザクション処理向けである．分析的な処理に向いているデータベースの例としてはApache HBase[16]やApache Kudu[17]がある．

データ処理コンポーネントは，自身の目的に対して適切な内部データ表現を使用して作られている．このデータ構造はほかのコンポーネントの内部データ表現とは無関係に設計され，内部データをそのままほかのコンポーネントに連携することは不可能である．

2つの異なるデータ処理コンポーネント間でデータの連携が必要になった場合，次のいずれかの方法でデータの受け渡しを実現することになる．

（1）2つのコンポーネント間で専用のデータ連携の仕組みを実装する．たとえば，多くのリレーショナルデータベースシステムは，C言語で実装されるアプリケーション用のクライアントライブラリを提供している．このクライアントライブラリとデータベースサーバの間でデータ連携のプロトコルを定め実装している．アプリケーションは，クライアントライブラリが提供するデータ構造にアクセスすることでデータを操作できる．
（2）JSONのような共通のデータフォーマットを仲介してデータ連携を実現する．この場合，各コンポーネントが内部データ表現と共通フォーマットの間のデータ変換を実装する．さらに，コンポーネント間で共通フォーマット上のスキーマを示し合わせることも必要となる．

近年，コンピュータの性能向上に伴いデータ処理システムへの要求は高度化し，複数のコンポーネントを組み合わせることで生じる問題が目立つようになった[18]．データ処理システムが複雑化することで，使用されるコンポーネントの数が増え，データ連携が必要となるコンポーネントの組合せが増大した．データ連携のパスが増えると実装しなければならないデータ変換処理も増えてしまう．専用のデータ連携パスが存在しない経路でJSONのような共通フォーマットを利用すると，今度は大きなデータ量を連携する際のデータ変換処理が大きなCPU時間を消費してしまう．

このようなデータ処理コンポーネントが抱える問題を解決する手段としてApache Arrowが提案された．

2.2　Apache Arrowが解決を目指す問題と，解決のためのアプローチ

Apache Arrowは分析的データ処理コンポーネントが持つ次の2つの問題を解決することを目指している．

（1）データ連携時のコスト
（2）コンポーネント間の機能差に起因する問題

2.2.1　データ連携時のコスト

異なる内部データ表現を採用する2つのコンポーネント間でデータを連携する方法は2つあった．すなわち（1）専用のデータ変換を実装すること，および（2）JSONのような共通フォーマットを採用する方法である．専用のデータ変換を実装する方法は，データ連携の経路が増えるに従って実装すべきデータ変換の数が増え，コードのメンテナンスコストが増大する．一方，共通

データフォーマットを仲介する場合は，シリアライズとデシリアライズでデータ変換が2回走るため連携したいデータ量が増えると実行時の処理速度が増大してしまう．このようなメンテナンスコストや実行時のコストはないほうが望ましい．

Apache Arrowは，コンポーネント間のデータ表現のハブとなる新たなデータフォーマットを定めることで，この問題の解決を目指している．**図1**はApache Arrowの公式サイト内[19]に掲載されているもので，Apache Arrowによってコンポーネント間のデータ連携が効率化される様子を示した図である．図の左側はApache Arrowがない状態であり，上述の（1）に相当する状態である．そして，図の右側の状態がApache Arrowによってデータ連携が効率化された状態である．

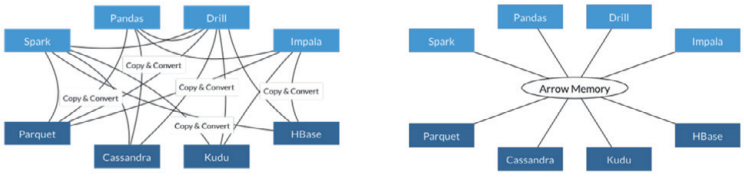

図1　Apache Arrowによってデータ連携が効率化される以前（左）と，効率化された後（右）の様子（[19]より引用）

Apache Arrowのデータフォーマットはメモリ上のデータの配置に関する仕様でありArrowフォーマット[20]と呼ぶ．Arrowフォーマットの詳細については次項で述べる．

Apache Arrowによるデータ連携では，メモリ上のArrowフォーマットをコンポーネント間でそのまま受け渡すことを想定している．そのため，データ連携のためのデータ変換処理を実装する必要はなく，実行時のデータ変換コストも発生しない．ネットワークを介したデータ連携で必要な場合にデータの圧縮をサポートするが，メモリ上で値が連続的に配置されている配列の単位で圧縮を行うため，データの受信側では受け取った圧縮データを展開してメモリ上にそのまま配列データとして配置するだけである．

JSONのようなデータフォーマットとArrowフォーマットを比較すると，Arrowフォーマットでは構文解析のコストが生じず，非常に高速にデータの受け渡しが可能となる．**図2**は，データフォーマットごとにデータの読み込み時間を計測した結果である．読み込み対象のファイルは，長さ10バイトの無作為な文字列で構成される列が1列と，倍精度浮動小数点数の値を持つ列が10列で構成されるデータフレームを各フォーマットで保存して作成した．この結果から，一般によく利用されるCSVやJSONと比較して，Arrowフォーマットが圧倒的に高速であることが分かる．

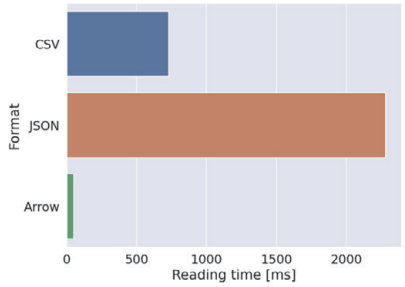

図2　データフォーマットごとの読み込み時間（シングルスレッド）

2.2.2　現代的なハードウェアの特徴を考慮したデータレイアウト

Apache Arrowが定めるArrowフォーマットは，2次元のテーブルデータと多次元数値配列データを対象とするもので，分析的データ処理で最もコンピュータの性能が発揮されることを狙って設計されている．

Arrowフォーマットでは，テーブルデータをRecordBatchと呼ばれる構造で表現する．RecordBatchは，1つのSchemaおよび列と同数の配列で構成されるオブジェクトである（**図3**）．Schemaはテーブルの論理的な構造を表すメタデータであり，テーブルを構成する各列の名前と値のデータ型の情報を持つ．列を表す配列には値が先頭から順に連続的に並んでいる．このように列単位でデータを配置する列指向のデータレイアウトを採用することで，同時にアクセスされやすいデータのキャッシュローカリティが高くなる．列指向のデータレイアウトと行指向のデータレイアウトの違いを**図4**に示す．また，配列の先頭要素はCPUのキャッシュラインやSIMD命令用レジスタのワードサイズに沿うようにアライメントされる．アライメントによって，メモリとCPU間のデータ転送効率が高くなる．

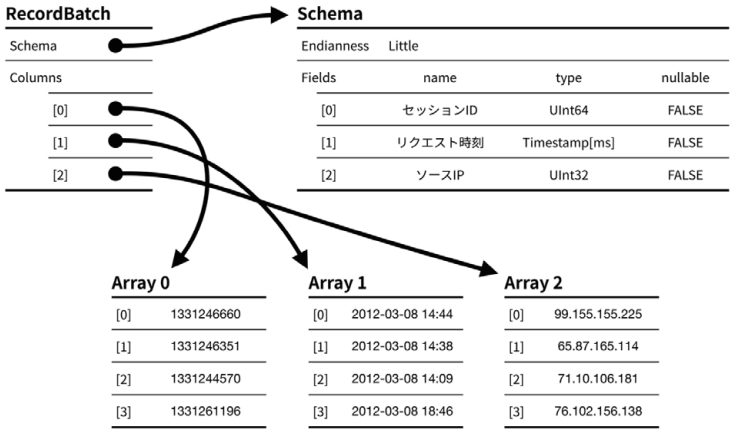

図3　RecordBatchのデータ構造

(a) 論理的なテーブル構造

	セッションID	リクエスト時刻	ソースIP
1行目	1331246660	2012-03-08 14:44	99.155.155.225
2行目	1331246351	2012-03-08 14:38	65.87.165.114
3行目	1331244570	2012-03-08 14:09	71.10.106.181
4行目	1331261196	2012-03-08 18:46	76.102.156.138

(b) 行指向のデータ配置

1行目	1331246660
	2012-03-08 14:44
	99.155.155.225
2行目	1331246351
	2012-03-08 14:38
	99.155.155.225
3行目	1331244570
	2012-03-08 14:09
	99.155.155.225
4行目	1331261196
	2012-03-08 18:46
	99.155.155.225

(c) 列指向のデータ配置

セッションID	1331246660
	1331246351
	1331244570
	1331261196
リクエスト時刻	2012-03-08 14:44
	2012-03-08 14:38
	2012-03-08 14:09
	2012-03-08 18:46
ソースIP	99.155.155.225
	65.87.165.114
	71.10.106.181
	76.102.156.138

図4　テーブルデータのメモリ上の配置方法の違い

このように，Arrowフォーマットは列の単位でメモリ上に値を連続配置する列指向レイアウトであり，キャッシュメモリとSIMD命令という現代のCPUの特徴を考慮してレイアウトが設計されているため，分析的データ処理でよく実行される列のスキャンが必要なデータ処理が効率よく実行できる[21].

Arrowフォーマットは2016年に最初の案が発表提案された．それから現在までまだ数年しか経過していないが，このフォーマットの特徴を活かした先進的なデータ処理コンポーネントがすでにいくつか存在する．たとえば，PySpark[22]，HeteroDB社のPG-Strom[23]，そしてNVIDIAのRAPIDS[24]である．また，研究段階のものとしてはFletcher[25],[26]がある．

2.2.3　コンポーネント間の機能差に起因する問題

複数のコンポーネントが類似した機能を持っている場合の機能差が問題となる場合がある．ここでは，データフレームと呼ばれるテーブルデータの分析のためのAPIを例にこの問題について説明する．

データフレームのAPIはプログラミング言語が異なっていてもおおよそ共通の機能セットを提供することになる．そのため，各プログラミング言語でデータフレームAPIを提供するライブラリ群は，多くの共通するアルゴリズムや機能を提供している．たとえば，配列の平均や標準偏差を求める機能，CSVファイルを読み込んでデータフレームオブジェクトを生成する機能などである．

既存のデータフレームライブラリは，内部データ表現と実装言語が異なるため，このような共通機能を互いに再利用できない実装として各々で独自に行っている．開発プロジェクトが分離しているため，ノウハウの共有も活発ではない[27]．

機能の実装が共有されないことで，共通機能の微妙な仕様の違いが発生している．たとえば，pandasとRでは，NaNをカテゴリ型配列のカテゴリとして扱えるかどうかの違い[28]や，データフレームのマージ処理のパフォーマンスの違い[29]が存在する．このような微妙な機能差は予測不可能なタイミングでアプリケーションにとっての問題に発展することがあり，そうなると非常に厄介である．

これらの問題は，データフレームライブラリが内部データ表現を共有し，コア機能を共通の言語で実装することで回避可能である．

Apache Arrowでは，データフレームAPI，クエリAPI，データ入出力APIについて，次の方法でこれらの問題の解決を目指している．その方法は，C++やJavaのようにハードウェアの性能を引き出せる少数のプログラミング言語でコアライブラリを実装し，それ以外のPython，R，Rubyのような言語はコアライブラリのバインディングを作成する方法である．こうすることで，異なる言語で同じ機能を実装する回数を最小限に抑えられ，どのプログラミング言語でも最高のパフォーマンスを発揮させることが可能となる．アルゴリズムで操作する対象となるデータ構造が共通であるため，実装言語が異なっていてもアプローチを共有できる可能性があり，異なるプログラミング言語の開発者同士でノウハウを共有し合い，開発の労力を減らせる可能性もある．

2.3　RubyがApache Arrowに対応するメリット

RubyがApache Arrowに対応することで，具体的にどのようなメリットがあるのだろうか．本節では，筆者がメリットであると考えている2つの要素について述べる．

2.3.1　Apache Arrowをハブとするデータ処理コンポーネントのネットワークに容易に入れるようになる

RubyがApache Arrowに対応することによって，Apache Arrowに対応したデータ処理コンポーネントのエコシステムに容易に参入できるようになる．つまり，図1の右側に示したApache Arrowによってコンポーネント間のデータ連携を効率化した後の世界にRubyが参加できるようになる．これは非常に大きなメリットであろう．

2.3.2　Apache Arrowが持つ将来性を直接Rubyの将来性に繋げられる

最近になってApache Arrowを応用した製品や研究プロジェクトが次々と登場してきている．

HeteroDB社が開発したPG-Strom[23]は，PostgreSQLから効率よくGPUを利用できるようにした拡張モジュールである．SQLクエリから自動的にGPU用のコードを生成し，クエリの実行をGPU上で非同期かつ並列に実行できる．PG-StromはもともとApache Arrowとは無関係であったが，2019年にリリースされたバージョン2.2でArrowフォーマットに対応し，SSDとGPUの間でArrowフォーマットのデータを直接転送することができるようになった．その結果，PG-Stromによって4倍以上に改善れていたクエリ実行のスループットが，Arrowフォーマットを利用することでさらに3倍以上改善された[30]．

Apache Arrowを応用する研究例としてはFletcher[25],[26]がある．Fletcherは，データ処理でFPGAアクセラレータを利用するためのフレームワークであり，FPGAにおけるデータ表現としてArrowフォーマットを採用している．Arrowフォーマットを利用したことで，演算効率が向上し，正規表現マッチで9〜49倍，文字列の書き出しで1.3倍，K-meansクラスタリングで最大2.7倍の速度向上を達成している[25]．

RubyがApache Arrowに対応するもう1つのメリットは，Apache Arrowの周囲で行われるこのような先進的な取り組みをRubyの将来性に直接繋げられることにある．

3. Red Arrow - RubyのApache Arrow対応

3.1 Rubyにとっての適切なアプローチとは

RubyをApache Arrowに対応させる方法には2つの道が存在する．すなわち，（1）C++やJavaのようにApache Arrowのデータフォーマットを採用したデータ処理ライブラリをRubyで実装する方法，そして（2）C++で実装されたライブラリのバインディングを開発する方法である．

2つの方法を比較すると，明らかに後者のバインディング開発がRubyには適している．RubyはC++やJavaと異なり遅いプログラミング言語であり，実用的な速度を発揮するには多くの機能をC++言語で拡張ライブラリとして実装しなければならないだろう．そもそも，前者の方法は前述のApache Arrowが目指している方向からブレてしまう．したがって，C++ライブラリのバインディングを開発することがRubyのApache Arrow対応の唯一の最適な道である．

次に検討すべきことはバインディングの実装方法である．バインディングは，すべてを手書きする方法，自動生成の仕組みを使う方法，そして実行時に動的に生成する方法のいずれかで作成できる．

すべてを手書きする方法は，PythonとRのバインディングで採用されている実装方法である．この方法では，C++で拡張ライブラリを手書きすることになる．その拡張ライブラリでは，C++ライブラリにおけるクラスに対応するRubyのクラスを定義し，そのクラスのインスタンスがC++ライブラリのオブジェクトをラップできるようにRubyオブジェクトの構造体を定義する．

バインディングの自動生成は，SWIG[31]が広く普及している．SWIGを採用する場合，C++ライブラリとRubyライブラリの間の対応関係を記述するSWIG独自の定義ファイルを作り，それを維持していくことになる．この定義ファイルから，SWIGが拡張ライブラリのC++コードを自動生成する．つまり，自動生成の方法でも拡張ライブラリが作られる．しかし，現行のSWIGの定義ファイルのみでは，異なる言語で共有できる部分と共有出来ない部分があり，Ruby用の定義ファイルをそのまま流用してほかの言語のバインディングを生成できるわけではない．

実行時にバインディングを動的生成するにはForeign Function Interface（FFI）を利用する．Rubyでは標準ライブラリFiddleがFFIの機能を提供しているため，Fiddleを利用して実行時に共有ライブラリを動的ロードし，オブジェクトとして取り出した関数を呼び出すようなAPIをRubyで記述する．このような仕組みは存在するが，残念ながらApache ArrowのC++ライブラリを対象とする場合は適切な方法ではない．その理由は（1）C++ライブラリの関数名を表すシンボルがマングルされてしまうこと，そして（2）C++ライブラリの一部の機能がテンプレート機能を使って実装されているため共有ライブラリではなくヘッダファイルに存在することである．C++ライブラリに対してバインディングの動的生成を行うには，C言語用のバインディングを作成し，そのCライブラリをFFIで動的ロードして利用する必要がある．

このように，バインディングの作成方法は3つあり，それぞれに一長一短の特徴がある．C++ライブラリが対象の場合，バインディングを手書きする方法か，SWIGのような仕組みで自動生成するのが現実的な選択肢として残るだろう．

しかし，Apache ArrowのRubyバインディングであるRed Arrowは，そのどちらの選択肢も採用しなかった．実際に採用された方法は，C言語用のバインディングを作成し，Rubyバインディングを動的生成する方法である．ただし，C言語用のバインディングはGLib[32]を用いて実装すること，そしてRubyバインディングの動的生成にはGObject Introspection[33]を利用する点が特徴的である．その理由は3.3節で後述するように，GLibとGObject Introspectionの利用が汎用的なバインディング開発手法になり得るからである．

3.2　GObject IntrospectionによるRubyバインディング生成の仕組み

GObject Introspectionによってバインディングが生成される際に使用されるファイルとその関係を**図5**に示す．

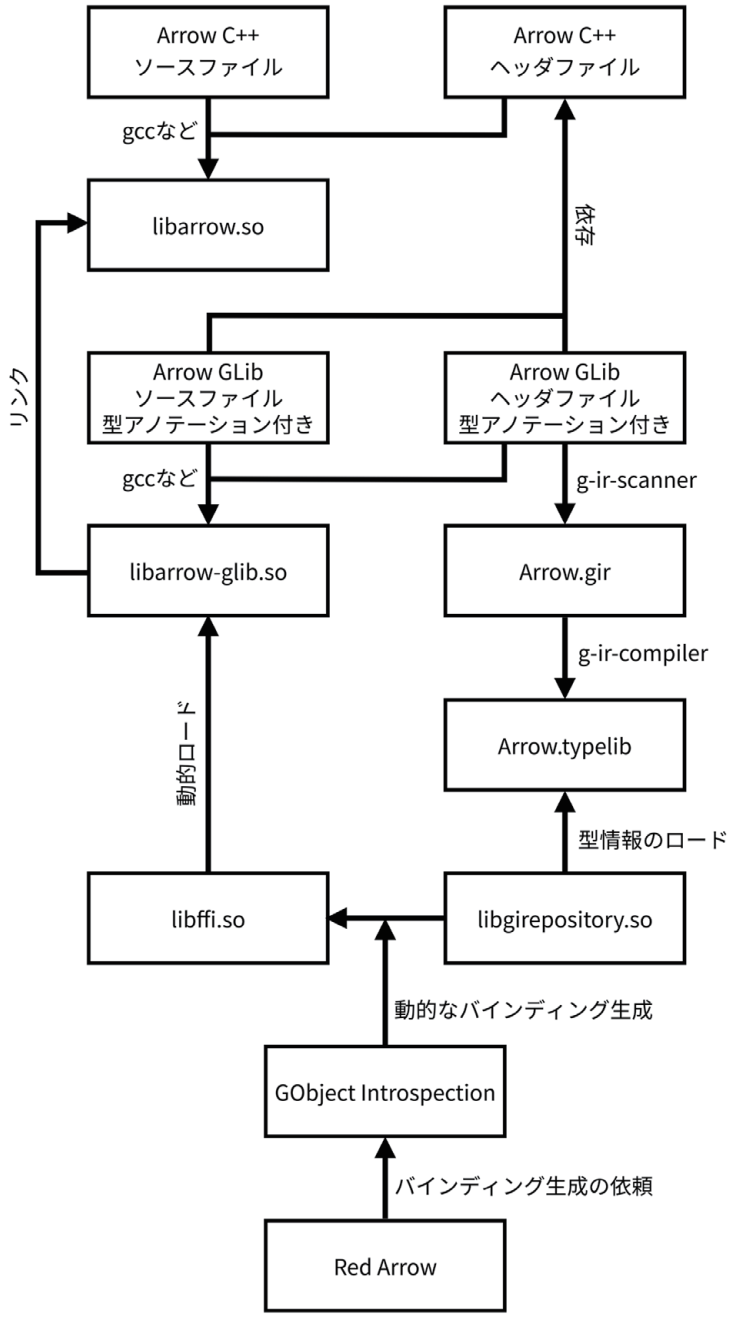

図5　GObject Introspectionによるバインディング生成

GObject Introspectionを利用するには，GObject Introspection用の型アノテーションが記述されたソースコードが必要となる．Red Arrowの場合，それはArrow GLibライブラリのソースコードである．型アノテーションつきソースコードから，GObject Introspectionのツールによって型情報データベースであるtypelibファイルArrow.typelibがArrow GLibライブラリのビルド時にlibarrow-glib.soと一緒に生成される．

実行時には，Red ArrowがGObject Introspectionにライブラリ名"Arrow"を指定してバインディング生成を依頼する．するとGObject IntrospectionがArrow.typelibをロードして型情報と依存ライブラリの一覧を取得，依存ライブラリのロードとlibarrow-glib.soのlibffi.soによる動的ロードを行う．そして，GObject IntrospectionのRubyバインディングが型情報に基づいてArrow GLibのクラスや関数に対応するRubyのクラスやメソッドなどを定義する．

RubyからArrow C++の機能を呼び出すには，Rubyバインディングのオブジェクトに対して対応するメソッドを呼び出せばよい．すると，Rubyオブジェクトが保持しているGLibオブジェクトを取り出してArrow GLibの関数を呼び出す．Arrow GLibの関数は，GLibオブジェクトの中からArrow C++のオブジェクトを取り出し，そのオブジェクトのメンバ関数を呼び出す．**図6**に，RubyのArrow::RecordBatchオブジェクトのschemaメソッドの呼び出しがArrow C++の関数呼び出しに至る流れを示す．

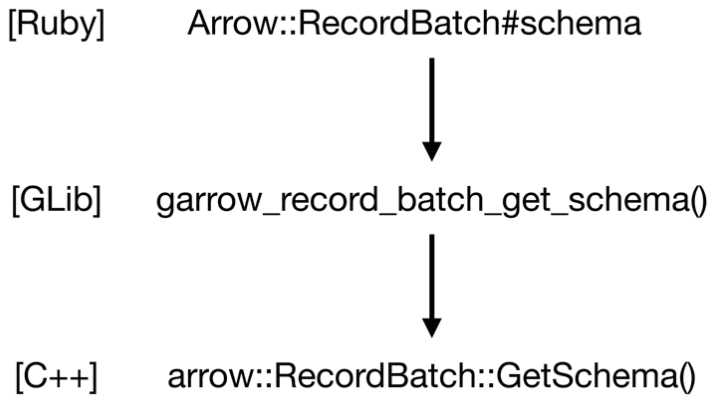

図6　RubyからArrow C++の関数を呼び出すときの流れ

3.3　GObject Introspectionを採用する利点

Red ArrowでGObject Introspectionを採用するメリットはあるのだろうか．Red Arrowが目的とするArrow C++がC++ライブラリであるため，一見するとGObject IntrospectionのためにArrow GLibをわざわざ作らねばならなくなっているように見える．つまり，バインディングをC++で手書きする，もしくはSWIGで自動生成するよりも開発コストが増えているように見える．

ここでポイントとなるのは，GObject IntrospectionがRuby専用ではないことである．つまり，Arrow GLibはRuby以外の言語でも，GObject Introspectionに対応している言語であれば理論的には利用可能である．

これは，Red Arrowにとってのメリットになっていないように見えるが，実はそうではない．Arrow GLibは現時点ではRed Arrowの開発者によって開発されている．しかし，将来的にRuby以外の言語のバインディングの開発者がArrow GLibの開発に参加してくれる可能性がある．そうなると，Red Arrowの開発者がArrow GLibの開発に割く労力が減るため，結果的にRed Arrowの開発コストが減る．これがRed ArrowでGObject Introspectionを採用するメリットである．

3.4　GObject Introspectionによるオーバーヘッド

Red Arrowを用いると必ずArrow GLibの関数を経由してArrow C++の関数が呼び出される．そのため，C++ライブラリへのバインディングを直接記述する場合には発生しない2種類のオーバーヘッドが生じる．garrow_record_batch_get_schema関数（**図7**）を例にこれらのオーバーヘッドについて説明する．

```
GArrowSchema *
Garrow_record_batch_get_schema(GArrowRecordBatch *record_batch)
{
  // 引数で受け取ったGLibオブジェクトから
  // C++オブジェクトを取り出す
  const auto arrow_record_batch =
    garrow_record_batch_get_raw(record_batch);

  // Rubyから呼び出したかった目的のC++関数の呼び出し
  auto arrow_schema = arrow_record_batch->schema();

  // C++オブジェクトをラップするGLibオブジェクトを
  // 生成して戻り値とする
  return garrow_schema_new_raw(arrow_schema);
}
```

図7　garrow_record_batch_get_schema関数の定義

1つ目のオーバーヘッドは関数呼び出しの増加である．これはArrow GLibの関数を経由することで生じる．Arrow GLib関数の内部では，第1引数で渡されたGLIbオブジェクトからC++オブジェクトを取り出し，目的のC++関数を呼び出し，そして必要な場合は戻り値をGLibオブジェクトでラップする処理も行う．garrow_record_batch_get_schema関数はこの3つの処理をすべて実行しているため，C++関数を直接呼ぶ場合と比較すると，見えている部分だけで3回の関数呼び出しが追加されている．

2つ目のオーバヘッドはメモリ割り当ての増加である．これは，C++オブジェクトをGLibオブジェクトでラップするために発生する．garrow_record_batch_get_schema関数では，戻り値となるarrow::SchemaオブジェクトをGArrowSchemaオブジェクトでラップするためにメモリの割当が発生している．

このようなオーバーヘッドは，たった数回のメソッド呼び出しで発生する程度なら問題にならない．また，メソッドが何度も呼ばれる場合であっても，メソッド内で呼ばれるC++関数の処理に時間がかかる場合はオーバーヘッド自体がほとんど気にならなくなる．

オーバーヘッドが問題となるのは，Rubyでループを回し，その中でバインディングのメソッド呼び出しを行う場合である．たとえば，RecordBatchを行方向にスキャンし，各行をRubyの配列に変換したものを集めて1つの配列にする処理をRubyで書くと**図8**のようになる．この図のraw_recordsメソッドでは，get_column_dataメソッドの呼び出しがgarrow_record_batch_get_column_data関数の呼び出しに対応し，その中でGArrowArrayオブジェクトが毎回割り当てられる．したがって，行数×列数分のGArrowArrayオブジェクトが割り当てられ，捨てられることになる．このようなオーバーヘッドを回避するには，**図9**のように各列の配列オブジェクトを事前に作成してキャッシュしておく必要がある．Red Arrowでは，RecordBatchの行と列のアクセスで無駄なオブジェクトが生成されないように，このような工夫がすでに実装されている．

```
class Arrow::RecordBatch
  def raw_records
    n_rows.times do |i|
      n_columns.times.map do |j|
        # 第 j 列目の配列オブジェクトを取得
        array = get_column_data(j)
        # 配列オブジェクトから i 番目の値を取得
        array[i]
      end
    end
  end
end
```

図8 RecordBatchを行の配列に変換するraw_recordsメソッドのRubyによる実装

```
class Arrow::RecordBatch
  def raw_records
    # 先に配列オブジェクトを作っておく
    arrays = n_columns.times.map do |j|
      get_column_data(j)  # 第 j 列の配列オブジェクト
    end
    n_rows.times do |i|
      n_columns.times.map do |j|
        # 事前にキャッシュした配列オブジェクトから値を取得
        arrays[j][i]
      end
    end
  end
end
```

図9 RecordBatchを行の配列に変換するraw_recordsメソッドのRubyによる実装

3.5　Rubyバインディング独自機能のC++による実装

図9のように工夫をしたとしても，まだ回避可能なメモリ割り当てが発生する可能性が残っている．それは，文字列型の配列から値を取り出す場合に生じるGBytesオブジェクトの割り当てである．文字列型の配列から値を取り出すときに呼び出される

garrow_binary_array_get_value関数は，**図10**に示した処理を行う．最後の行のreturnで返している値が新たに割り当てられるGBytesオブジェクトである．このGBytesオブジェクトはArrow GLibを経由しなければ必要ないものである．

```
GBytes *
garrow_binary_array_get_value(GArrowBinaryArray *array,
                              gint64 i)
{
  // GLibオブジェクトからC++オブジェクトを取り出す
  // 戻り値はarrow::Array型のポインタ
  auto arrow_array = garrow_array_get_raw(array);
  // arrow::BinaryArray型のポインタにキャストする
  auto arrow_binary_array =
    std::static_pointer_cast<arrow::BinaryArray>(arrow_array);
  // 第i番目の文字列の範囲を表すビューオブジェクトを得る
  auto view = arrow_string_array->GetView(i);
  // ビューと同じ範囲を表現するGBytesオブジェクトを返す
  return g_bytes_new_static(view.data(), view.length());
}
```

図10　garrow_string_array_get_string関数の定義を簡略化したコード

このように，Arrow GLibでは必要だがRubyでは不要となる操作は，GObject Introspectionによって生成されるバインディングを使う以上は回避できない．これを回避するには，C++で機能を実装する必要がある．実際にRed Arrowは，前節で例示したraw_recordsのC++実装を拡張ライブラリとして提供している．

4. Rubyのデータ処理に関する関連技術

Red ArrowによるApache Arrow対応は，共通データフォーマットに対応し，データ処理コンポーネントとの連携の可能性を高めることによってRubyをデータ処理に対応させるアプローチである．これとは異なるアプローチでRubyをデータ処理に対応させる事例がいくつかある．

1つはPyCall[34]である．PyCallはRubyからPython用のライブラリを利用するための言語ブリッジである．PyCallはRubyと同一プロセス下でPython処理系を動かし，Python処理系のC APIを用いてPythonとRubyの間のデータ連携を実現する．

過去に存在し，現在は開発が終了してしまったものもいくつかある．RSRuby[35]とRinRuby[36]は，R言語とのブリッジを実現するライブラリである．RSRubyは，Rubyとは別のプロセスで起動されたR処理系とソケット通信でデータ連携する．RinRubyは，PyCallのようにRubyと同一プロセスでR処理系を動かし，R言語のC APIを利用してデータ連携する．

ruby-spark[37]も開発が終了してしまったライブラリである．これは，Sparkとの連携を実現するものであった．

RSRuby，RinRuby，ruby-sparkの開発が終了してしまった最大の要因は，利用者がそこまで増えなかったことだと筆者は推測している．利用者が増えないとアクティブな開発者も増えない．その状況で，主な開発者がこれらのライブラリを必要としなくなると開発が止まり，開発を引き継ごうとするものも現れず，プロジェクトが自然消滅してしまう．

5. Rubyのデータ処理対応に関する今後の展望

論文の最後にRubyのデータ処理対応についての今後の展望についてまとめる．

5.1 Red Arrowの開発の継続

本稿では，Rubyをデータ処理に対応させていくにあたり，Apache Arrowに対応することが肝心であることを説明した．つまり，Red Arrowの存在がRubyのデータ処理対応にとって非常に重要となる．

Apache Arrowはすでに完成したライブラリではない．むしろ，最近バージョン1がリリースされたばかりで，データフォーマットとAPIの両方について発展途上である．C++ライブラリに限定しても，まだメモリ上のテーブルデータの表現と操作，基本的な演算，ファイルI/Oなどの基本機能，および高次機能に関してはデータソース抽象化の一部のみ提供されている．C++ライブラリの高次機能として計画されているクエリエンジンは本稿執筆時点で開発が始まったばかりであり，さらにデータフレームについては計画があるだけの状態である．

Apache Arrowは今後も速いスピードで開発が進行し，どんどん新しい機能を増やしていくはずである．Red Arrowが開発を止めず，Apache Arrowの進化に追従していくことが今後の展開として期待される．

5.2 Rubyのデータ処理エコシステムの拡充

Apache Arrowへの対応だけが充実していても意味がない．Rubyのデータ処理対応を成功させるには，Rubyプログラマが使用できるデータ処理ツールを拡充しなければならない．

昨今，Rubyコミュニティの一部で，Ruby用のデータ処理ツール開発が盛り上がっている．Red Arrowの開発を推進しているRed Data Tools[38]は，可視化ライブラリCharty[39]も開発している．そのほかにも，機械学習ライブラリRumale[40]や，深層学習用のライブラリであるlibtorchのバインディングTorch.rb[41]が作られたりしている．この盛り上がりを長期に渡り維持していくことも，Apache Arrowへの対応と同程度に重要なことだと筆者は考えている．

5.3 Ruby外のコンポーネントとの連携強化

データ処理システムは多くのコンポーネントを組み合わせて作らなければならない．Rubyで作られたシステムをデータ処理に対応させるには，Ruby外のコンポーネントとの連携強化が必要である．

Red ArrowによってApache Arrowフォーマットを活用できる環境が整備されているので，これを利用して他言語のコンポーネントへのアクセス手段を整備するとよいだろう．たとえば，今度こそPySparkのようなSpark連携を実現し，Sparkによる分散データ処理をRubyでも実装しやすい環境をつくる良いタイミングかもしれない．また，Apache Arrowのための分散計算プラットフォームであるApache Arrow Ballista[42]との連携を実現するのも魅力がある．

参考文献

1）Flanagan, D., まつもとゆきひろ（著），卜部昌平（監訳），長尾高弘（訳）：プログラミング言語Ruby，オライリー・ジャパン（2009）．
2）Ruby on Rails : https://rubyonrails.org
3）Cookpad : https://cookpad.com
4）GitHub : https://github.com
5）Shopify, https://shopify.com
6）Chevalier-Boisvert, M., Gibbs, N., Boussier, J., Wu, S. X., Patterson, A., Newton, K. and Hawthorn, J. : YJIT : A Basic Block Versioning JIT Compiler for CRuby, In Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL 2021, pp.25-32 (Oct. 2021).
7）Gallinelli, N. : The 10 Best Data Science Programming Language to Learn in 2021, https://flatironschool.com/blog/data-science-programming-languages (Mar. 2021)
8）Python : https://python.org
9）The R Project for Statistical Computing : https://www.r-project.org
10）Apache Spark : https://spark.apache.org
11）ECMA-404 : The JSON Data Interchange Format (2nd ed.), ECMA International (Dec. 2017).
12）経済産業省：産業界におけるデジタルトランスフォーメーションの推進，https://www.meti.go.jp/policy/it_policy/dx/dx.html
13）Apache Arrow : https://arrow.apache.org
14）Red Arrow - Apache Arrow Ruby : https://github.com/apache/arrow/tree/master/ruby/red-arrow
15）Rodriguez, S. A., Chakraborty, J., Chu, A., Jimenez, I., LeFevre, J., Maltzahn, C. and Uta, A. : Zero-Cost, Arrow-Enabled Data Interface for Apache Spark, arXiv:2106.13020 (cs.DC) (June 2021).
16）Apache HBase : https://hbase.apache.org
17）Apache Kudu - Fast Analytics on Fast Data : https://kudu.apache.org
18）Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S. and Cun, B.-G. : Making Sense of Performance in Data Analytics Frameworks, Proceedings of the 12th USENIX Syposium on Networked Systems Design and Implementation (NSDI 15), pp.293-307 (May 2015).
19）Apache Arrow Overview : https://arrow.apache.org/overview/
20）Arrow Columnar Format Version 1.0 : https://arrow.apache.org/docs/format/Columnar.html
21）Zhang, H., Chen, G. and Tan, K.-L. : In-Memory Big Data Management and Processing : A Survey, IEEE Transactions on Knowledge and Data Engineering, Vol.27, No.7, pp.1920-1948 (July 2015).
22）PySpark Documentation : http://spark.apache.org/docs/latest/api/python/
23）PG-Strom Manual : https://heterodb.github.io/pg-strom/

24）RAPIDS - GPU-Accelerated Data Analytics & Machine Learning : https://developer.nvidia.com/rapids
25）Peltenburg, J.W., Straten, J.V., Wijtemans, L., Leeuwen, L.V., Al-Ars, Z. and Hofstee, H.P. : Fletcher - A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow, Proceedings - 29th International Conference on Field-Programmable Logic and Applications, FPL 2019, pp.270-277 (2019).
26）Peltenburg, J.W., Straten, J.V., Brobbel, M., Hofstee, H. P. and Al-Ars, Z. : Supporting Columnar In-memory Formats on FPGA : The Hardware Design of Fletcher for Apache Arrow, In Hochberger, C., Koch, A., Diniz, P., Woods, R. and Nelson, B. (Eds.), Applied Reconfigurable Computing : 15th International Symposium, ARC 2019, Proceedings, pp.32-47 (2019).
27）McKinney, W. : Apache Arrow and the Future of Data Frame with Wes McKinney, Tech Talk, ACM, https://learning.acm.org/techtalks/apache (July 2020)
28）Differences to R's factor : in pandas User Guide, https://pandas.pydata.org/pandas-docs/stable/user_guide/categorical.html#differences-to-r-s-factor
29）McKinney, W. : Some pandas Database Join (merge) Benchmarks vs. R base::merge, blog post, https://wesmckinney.com/blog/some-pandas-database-join-merge-benchmarks-vs-r-basemerge/ (Jan. 2012)
30）Kohei, K.：PostgreSQLだってビッグデータ処理したい！！〜Apache Arrowが可能にする毎秒10億レコードのデータ処理〜，講演資料 https://www.slideshare.net/kaigai/20191211apachearrowmeetuptokyo (Dec. 2019)
31）SWIG : http://www.swig.org
32）GLib - 2.0 : https://docs.gtk.org/glib/
33）GObject Introspection : https://gi.readthedocs.io/en/latest/
34）PyCall : Calling Python Functions from the Ruby Language, https://github.com/mrkn/pycall.rb
35）Ruby - R bridge : https://github.com/alexgutteridge/rsruby
36）Dahl, D. B. and Crawford, S. : RinRuby : Accessing the R Interpreter from Pure Ruby, Journal of Statistical Software. Vol.29, No.4, pp.1-18 (Nov. 2009).
37）Ruby Wrapper for Apache Spark : https://github.com/ondra-m/ruby-spark
38）Red Data Tools - Data Processing with Ruby! : https://red-data-tools.github.io/
39）Charty - Visualizing Your Data in Ruby : https://github.com/red-data-tools/charty
40）Rumale is a Machine Learning Library in Ruby : https://github.com/yoshoku/rumale
41）Deep Learning for Ruby, powered by LibTorch : https://github.com/ankane/torch.rb
42）Ballista : A Distributed Scheduler for Apache Arrow, https://arrow.apache.org/blog/2021/04/12/ballista-donation/

村田賢太（正会員）muraken@gmail.com

（株）Speeeにてフルタイム OSS開発者に従事．2010年よりRubyコミッタ，2018年よりApache Arrowコミッタも務める．Rubyをデータ処理に対応させることを目指して精力的に活動中．

須藤功平（非会員）kou@clear-code.com

（株）クリアコード代表取締役．2004年よりRubyコミッタ，2017年よりApache Arrow PMCメンバ．2017年よりRed Data Toolsプロジェクトを開始．

受付日：2021年9月1日
採録日：2021年10月27日
編集担当：青木学聡（京都大学）