

多言語に対応した自己記述をもったバイナリデータ形式

大谷 桂 介^{†1} 脇 田 建^{†1,†2}

データ形式はテキスト形式とバイナリ形式の2種類に分けられるが、テキスト形式には処理性能の問題、バイナリ形式には人間に扱いづらいという問題がある。また共通する問題としてデータ構造をプログラムで適切に扱うための仕組みが必要である。本稿では多言語に対応した自己記述情報を持ったバイナリ形式を提案することでこれらの問題を解決する。処理性能に優れたバイナリ形式にデータ構造などの自己記述情報を持たせ、データ構造に対応したAPIを自動生成することができる。応用分野として科学技術計算とその可視化を想定している。

A Self-descriptive Binary Format Available in Multi-language

KEISUKE OHTANI^{†1} and KEN WAKITA^{†1,†2}

Data formats can be classified in text formats or binary formats. Text formats are not suitable for processing, and binary formats are difficult to use for human. We propose a self-descriptive binary format available in multi-language and solve these issues in this paper. This format is high performance binary format and has self-descriptive data which includes its data structure. You can generate APIs from structure data. Our format can be used in scientific computation and its visualization as application.

1. はじめに

複数のプログラムやユーザー間でデータを共有するときやプログラムで使ったデータを後で使うために二次記憶装置に保存するとき、そのデータの形式をあらかじめ決めておく必要

がある。普及しているデータ形式は多種多様だが、大きく分けると印字可能な文字列のみでデータを表す**テキスト形式**と、任意のバイト列でデータを表す**バイナリ形式**の2種類になる。テキスト形式のデータは人間が読めるため扱いやすいが、プログラムで処理するときの性能はバイナリ形式に劣る。バイナリ形式は処理性能に優れるが、人間には読むのが困難なデータのため何のデータなのか理解しにくい。またどちらの形式でもデータの構造をプログラムで適切に扱うための入出力APIが必要となる。

本稿では多言語に対応した自己記述を持ったバイナリデータ形式を提案する。処理性能に優れたバイナリ形式を採用し、自己記述を持つことでバイナリ形式の欠点であるデータの扱いにくさの問題を解消している。またデータ構造に対応したAPIを自動生成する機能を提供することでAPIを用意する負担も取り除いている。これらの特徴によって提案形式は様々な言語や環境で利用可能なデータ形式となっている。

本稿の構成は以下の通りである。第2節では様々なデータ形式とその問題点を説明する。第3節ではこれらの問題を解消する新たなデータ形式を提案する。そして第4節で提案形式の能力について評価する。最後に第5節では本稿をまとめる。

2. 関連研究

メモリに入りきらないデータを二次記憶装置に保存するときや、複数のユーザーまたはプログラム間でデータを共有するときに、一定の形式に従ってデータを記録する必要がある。例えば科学技術データはテラバイト単位のデータを共同研究で利用することが容易に想定できる。データ形式は印字可能な文字列でデータを表すテキスト形式と、任意のバイト列でデータを表すバイナリ形式の2種類に分けられる。

2.1 テキスト形式

テキスト形式は一般のテキストエディタで簡便に編集可能であり、たとえそのデータ形式が不明な場合であっても、容易に表示し、その内容から形式を憶測できる場合も多い。しかしテキスト形式は印字可能文字に限定されるため、バイナリ形式と比べてデータ圧縮には向かない。この圧縮問題に対しては1)のようにテキストデータを圧縮しバイナリデータにしたものをbase64でASCII文字列に変換しテキスト形式にする手法がある。しかしこのように生成した文字列は自然言語として無意味な文字列であり、ユーザーが直接読み書きできる利点が失われる。構造を持ったテキスト形式にはXML^{2),3)}やJSON⁴⁾がある。どちらも通信プロトコルにおけるデータ表現や、複合的データの表現形式として広く利用されている。しかしこれらのデータ形式を含めてテキスト形式は各要素のデータ長が不定のためデー

^{†1} 東京工業大学大学院 数理・計算科学専攻

Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

^{†2} 独立行政法人科学技術振興機構, CREST

Japan Science and Technology Agency, CREST

タ全体を読み込まなければランダムアクセスができない。またテキストデータの扱いにおいては、メモリ中のバイナリ表現との相互変換のために、エンコーディングの処理、構文解析、型変換、シリアライズなどの多くのオーバーヘッドが要求される。

2.2 バイナリ形式

一方バイナリ形式の利点は全てのバイト列を使えること、ランダムアクセスが可能なことである。しかしバイナリデータはユーザーが直接読むのが困難なため、データだけではその内容や利用方法が分かりにくい。これはデータを複数人で共同利用する場合や古いデータを利用するとき障害となる。またバイナリデータは同じバイト列でもそのデータ構造（スキーマ）によって表す意味が全く異なる。すなわちバイナリデータを利用するにはそのスキーマに対応したデータ読み書きプログラムをスキーマごとに用意しなければならない。異なるソフトウェア環境で利用するときは、それぞれの環境に対応したプログラムを作らなければならない。そしてスキーマが変更されたらそれぞれのプログラムにも更新作業が必要となる。ユーザーにとってはデータを読み込んだ後に行う解析などの処理が本来の目的であるにも関わらず、このような煩末なプログラミングに一定の労力をかけられている。

2.2.1 バイナリ表現された構造化テキストデータ

テキスト形式の欠点を解消しバイナリ形式の利点を取り入れた形式として XML をバイナリ形式にした EXI⁵⁾、Xebu⁶⁾、BXSA⁷⁾ や JSON をバイナリ形式にした BSON⁸⁾、BinaryParser⁹⁾ が挙げられる。

テキスト形式をバイナリ形式にすることでランダムアクセスが可能になり、型情報を持つことで数値データなどを適切なバイナリ表現で扱える。これらの形式の利点は元のテキスト形式と同じように利用できる点である。広く普及している XML や JSON に対応したプログラムやライブラリは多く、それらと容易に連携ができる。特にブラウザと連携できる点が大きな利点であり、様々なプラットフォームに対応したインターフェースを構築するためのデータ形式として利用できる。

これらの形式の欠点は多数の抽象化層を導入することに伴うデータ処理速度の劣化である。つまり、バイナリ表現されたテキストデータのアクセスのためには、従来の構造化テキストデータと同様にエンコーディングや複雑な構文解析を要する上に、低レベルにおけるテキスト、バイナリ間の相互変換を要する。また、構文解析されたのちも、高速なアクセスが求められる場合には、さらに計算機に自然な表現との相互変換が求められる。このため、バイナリ形式を利用した通信料の削減の利はあるものの、大規模数値計算のように高速なデータ処理が求められる応用には、より効率的なデータ表現形式が求められる。

2.2.2 バイト列へのシリアライズ

プログラミング言語にはオブジェクトの内容をバイト列へシリアライズする手段を提供するものがある。これで生成されたデータもバイナリ形式のひとつである。Java の ObjectOutputStream クラスや .NET Framework の BinaryFormatter クラス、PHP の serialize 関数がバイト列へのシリアライズ方法として挙げられる。

この方法の利点は、バイナリデータとプログラミング言語のオブジェクトが完全に対応していることである。バイナリ形式に合わせたオブジェクトをプログラミング言語で定義したり、逆に言語中のオブジェクトに合わせたバイナリ形式を設計する必要がない。特別な変換が必要ないため、オブジェクトをそのままデータとして出力したり、データをオブジェクトへ直接読み込むことができ、プログラムコードを簡潔に記述することもできる。また BinaryFormatter クラスや serialize 関数のようにオブジェクトの内容を自動的に全て出力できる方式だと、新しいオブジェクトに対応した入出力機能を作る必要がなくユーザー独自のオブジェクトを導入しやすい。

シリアライズ方式の欠点は特定のプログラミング言語に依存していることである。シリアライズしてできたデータを他の言語で利用する場合、他のバイナリ形式と同様に読み込んだデータをその言語での適切なオブジェクトに変換することになる。これではシリアライズの恩恵を全く受けられない。ひとつの言語のみを使えばシリアライズを活用できるが、データの用途に合わせて最適な言語を選択することができない。そのため他の言語なら標準でサポートしている機能を実装する負担が生じる。またシリアライズは常にオブジェクト単位でデータを扱うため、オブジェクトをひとつずつ読み書きする処理が基本となる。そのため多数のオブジェクトデータの中から必要な部分だけを抜き出すランダムアクセスが難しく、大量のデータを扱うのには向かない。

2.2.3 通信プロトコル

プログラム間の通信のプロトコルとして Protocol Buffer¹⁰⁾ や MessagePack¹¹⁾、Thrift¹²⁾、Avro¹³⁾ のようなバイナリ形式がある。

これらの形式の共通点としてデータがスキーマを持っていることが挙げられる。スキーマを持つことによってデータの中に想定外の形式のデータが紛れてもそれを判別することができる。またスキーマが既知であることからその構造専用のデータアクセス方法を提供することができ、スキーマ特有の性質を利用してデータサイズを圧縮することも可能である。Remote Procedure Call (RPC) で利用できるのも共通点のひとつである。ひとつのサーバーにライブラリをインストールしておけば各クライアントは RPC で必要な関数を呼び出

すことで通信プロトコルを利用することができる。プロトコルを使う各クライアントにはライブラリなどを個別にインストールする必要がないことは多数のマシンとの通信に利用される通信プロトコル形式にとって利点となる。

ただしこれらの形式はプログラム間のデータ通信を主な用途としていることから、想定するデータはひとつのマシンのメモリに格納できる程度のサイズであり、大規模なデータを想定していない。メモリに格納できるサイズなら利用するデータが全体のうちの一部分であっても、データ全体をメモリに読み込んで利用すれば簡単でアクセス効率も良い。しかしデータサイズが増えればメモリにデータ全体を読み込まず、データファイルから必要な部分を探して読み込むことになる。データファイルが格納されている二次記憶装置はメモリよりアクセス速度が遅いため、データ形式と連携した探索が求められる。通信プロトコル用データ形式はこのような場合での利用を想定しておらず、大規模なデータを効率よく利用することができない。

2.2.4 分散データストレージシステム

大規模なデータを効率よく処理するためのシステムとして分散データストレージの BigTable¹⁴⁾ や Cassandra¹⁵⁾ が挙げられる。

これらのシステムの特徴は多数のマシンを使ってデータを管理することである。データへのアクセスや操作のリクエストを同時に多数受け取っても、それぞれのデータを別のマシンで管理していればその処理を分散させることができる。同時に利用することが多いデータ群などを考慮してデータを配置することで局所性を利用してアクセス速度を上げることもできる。また新たなマシンを管理システムに追加することで記憶領域や処理能力を向上させることができる。これらのシステムのデータモデルは複数のキーからひとつの値へのマッピングを基本としている。同じキーを持つデータ群をグループ分けして管理単位とすることで局所性を利用できる。キーによってプログラムからのアクセス制限を設定することもできる。

このようなシステムの欠点はデータモデルが制限されていることである。プログラムで利用した構造を持ったデータをこれらのシステムに保存する場合、その構造体をキーから値へのマッピング関係に変換しなければならない。そしてデータの読み込み時はこの逆の変換が必要である。この変換の過程で失われるデータがあってはならない。またスキーマの変更に対応できる柔軟な設計が求められる。

3. 提 案

2 節ではテキスト形式とバイナリ形式が持つ問題点を述べた。テキスト形式は全てのデー

タを文字列に変換する必要がある。アクセスが自在に行えない。バイナリ形式は直接読めないため単体では利用方法がわからない。そしてデータのスキーマとプログラミング言語のデータ型を適切に対応付ける入出力プログラムを用意する負担がある。

本稿では複数のプログラミング言語に対応した自己記述情報を持ったバイナリデータ形式を提案する。この形式のデータは性能に優れたバイナリ形式を多数のプログラミング言語やソフトウェア環境で利用するために必要な情報を自己記述情報として内包している。この設計によって上述したデータ形式の問題点を解決し、様々な目的で統一的に利用可能な柔軟で効率のよいデータ形式となっている。

提案形式においては各バイナリデータは、そのデータのメタ情報である自己記述情報を内包する。自己記述情報は、バイナリデータの構造とエンコーディングを機械的に処理可能な形式で表現したスキーマ、データの内容、作られた目的と手段などを自然言語で記述した情報、作成日時、そのデータが二次的データである場合には一次データの記述子などをプラットフォームに独立した形式で記述したものである。ユーザーは、データの自己記述情報を閲覧することで、データの種別や利用方法を容易に知ることができる。閲覧手段はプラットフォームに独立した形で提供する。

バイナリデータのスキーマをプログラミング言語で適切に利用するために、スキーマの入出力 API を自動生成する **スキーマコンパイラ** を用意する。ユーザーは生成した API を利用するだけで、読み込んだデータをプログラミング言語でのデータ型として利用できる。そのために、生成する API にはデータ入出力機能の他に、スキーマで定義した構造を言語のデータ型に変換した型定義が含まれる。スキーマは特定のプログラミング言語に依存しないため、生成する API は様々な言語に対応させることができる。

提案形式は実際には JAR 形式を取っている。JAR 形式は ZIP アーカイブであり、Java クラスファイルと、マニフェストファイルと呼ばれるクラスファイルのセキュリティやメインクラス指定を記述したファイルを含む。JAR ファイルは Java 実行環境があれば内包するクラスのメソッドをスタンドアロンに起動することができる。そのためデータファイルにスキーマコンパイラのクラスファイルを添付することで、データファイルだけでスキーマコンパイラを実行することができる。自己記述閲覧手段も同様の形式で提供し、ユーザーは Java 実行環境さえあれば外部アプリケーションを用意せずに提案形式の機能を利用できる。

提案形式はバイナリ形式でありながらテキスト形式のように扱いやすいデータ形式となっている。特に科学技術データのようなデータを扱うのに向いている。このようなデータは複数人で共同利用することがあるが、自己記述情報によってデータに関する情報を容易に共有

できる。研究グループの新規メンバーが複雑なデータを処理することになっても、そのメンバーはデータファイルに添付された自己記述を参照できる。他のメンバーの連絡不備によって利用方法が分からなくなることがなく、すぐに作業を始めることができる。そして科学技術データは種類や目的によって様々な構造をとるが、それらはスキーマを定義するだけで対応できる。定義したスキーマからスキーマコンパイラで使用言語向けに API を生成できるため、データの性質や処理目的に合わせて最適な言語や環境を選択できる。グループ作業の場合メンバーによって使用言語や環境が異なることも考えられるが、それぞれに合わせた API を生成できるためメンバー各々が使い慣れた環境のまま共同作業ができる。また提案形式のデータはバイナリ形式のため、効率よくデータ処理を行える。

3.1 利 用 例

本形式の利用例として、ウェブから収集した大規模社会ネットワークシステムのデータをスーパーコンピュータを用いて解析しその結果を可視化することを考える。まずは一連の手法が正しく動作するか検証するために、ローカルマシン上で小規模のサンプルデータを作りテスト解析を行うとする。スキーマを決定するまでには、スキーマを仮決めしてそれに対応した収集・解析プログラムでテスト実行し、その結果を受けてスキーマを修正し、再度テストを行うというサイクルが必要になる。本提案形式はスキーマから API を自動的に生成し、その API を通してバイナリデータをプログラミング言語に自然な形式を通してアクセスできるため、スキーマの変更に伴う解析プログラムの修正は限定的である。このため、プログラマは従来のような煩末で面倒な低レベル入出力の処理から開放される。

次にデータ収集は Java を用いたクロールロボットを使うとする。テストプログラムで Java 以外の言語を使っていたことも考えられるが、API の自動生成によって入出力部分は直ちに完成するので収集処理本体の実装に集中できる。収集されたデータはランダムアクセスができるように構成され、スキーマなどの自己記述情報が自動的に添付される。これにデータの説明などの情報を手動で追加することも可能である。

続く解析作業はスーパーコンピュータ上で並列言語 X10 を用いる。これまでと異なるソフトウェア環境でもそのまま利用できるのが提案形式の利点のひとつである。データを利用するために特別なインストール作業は必要なく、データファイルを設置するのみでよい。また解析作業をデータ収集担当とは別の人が担当することも考えられる。通常ならばデータの仕様が伝えるためのドキュメントなどを作り共有する必要があるが、提案形式はデータ自体がそのドキュメントとなるため情報の共有をし損なう恐れがなくスムーズに作業を引き継ぐことができる。提案形式はバイナリ形式なのでテキスト形式のようなオーバーヘッドはな

く解析ができる。

解析結果の膨大なデータはウェブ上で実装された対話的なブラウザを用いて閲覧することにする。この場合、バイナリ形式をサポートしない JavaScript にバイナリデータを提供するために、スキーマコンパイラは JSON 形式を解した API を生成する。JSON は主要なスクリプト言語がサポートしているため、この API を利用すればそれらの言語で記述されたプログラムの間のデータ交換も容易である。

このように提案形式は様々なプログラムや環境を通して行われる処理の中で統一して利用できるデータ形式である。複数のデータ形式を使う場合プログラマはそれぞれの形式を理解するだけでなく、各形式間との変換を実装することにもなる。データ形式によっては導入作業も必要になることも考えられる。提案形式のように統一したデータ形式があればそれらの問題は解消できる。

3.2 ス キ ー マ

バイナリ形式は単体では単なるバイト列に過ぎない。そのバイト列をどのように解釈するか意味を与えるのがデータ構造を表すスキーマである。バイナリデータの読み書きにはスキーマのみを頼りにすることができる。スキーマをユーザー側で定義することができれば、使用目的に合わせて最適な構造でデータを扱うことができる。提案形式の特徴としてスキーマがデータファイルに添付されていることがある。広く普及している画像や音声、動画などのバイナリ形式ではデータ形式の仕様がデータの外部で文章化されていることが多い。この定義方法はデータの種類が明らかな場合には有効だが、提案形式のようにユーザーが独自の構造を定義できるようにしたい場合では未知のデータから構造定義を探す手段がないため使えない。スキーマがデータと一体になっていれば、データを利用するとき簡単にスキーマを参照することができる。またスキーマによる自己記述が保証されているため、構造が不明になることなく独自のデータ形式を定義し利用できる。ここで問題になるのはスキーマを記述する方法である。自由なデータ形式が使える基盤があっても、使いたい構造をスキーマで記述できなければ意味が無い。

スキーマではデータを**データ要素**の集合として考える。データ要素は名前と型を持ち、プログラミング言語で変数として操作できる単位である。スキーマはデータ要素の名前と型の記述を基本とする。データ要素に名前をつけるのは同じ型の要素を名前で区別できるようにするためである。基本的な型として整数型と実数型、文字列型が指定できる。整数型と実数型はバイト長も指定する。また整数型は符号付きか符号なしのどちらかも指定する。値域や必要な精度などに応じてこれらを設定することでサイズ効率のよいスキーマを定義できる。

文字列型は可変長で任意の文字列を扱える。スキーマで長さを固定すると単純に扱える文字列が減ってしまう。そして指定した長さを超える文字列が渡されたときに例外処理としてその文字列を拒否するか規定の長さまで切り詰めるかなど複数の処理が考えられ、その例外処理は自動生成する API の仕様とするかユーザーが指定できるようにするか、という問題が生じる。任意の文字列が使えるようにすればこれらの問題は起こらない。そして構造を持ったデータを表現するための型として配列型とオブジェクト型がある。配列型は任意の型をひとつ取って可変長配列にすることができる。可変長とした理由は文字列型の場合と同様である。オブジェクト型は子要素として内部に別のデータ要素を複数定義することができる。この型によって関連のあるデータ要素をひとつにまとめられる。子要素としてオブジェクト型の要素を取ることもでき、複数の階層構造を形成するスキーマも定義できる。

これらの特徴を持ったスキーマは JSON 形式に従って記述する。JSON を採用した理由はスキーマがデータ要素の名前と型の組を基本とするからである。JSON のオブジェクトメンバは名前と値の組を列挙して記述する。この値としてデータ要素の型を記述すればデータ要素の定義が表現できる。また JSON のオブジェクトはオブジェクトをメンバにして階層構造を取ることができ、スキーマのオブジェクト型として利用できる。これらの類似点から JSON を採用した。JSON を利用することで多数のプログラミング言語が対応しているという特徴を活用できるのも大きな利点である。C や Java, JavaScript など 40 以上の言語で利用できるようにするため、スキーマの読み書きや他のプログラムとの連携も容易に実装できる。テキスト形式のためユーザーが直接読み書きできるのも利点のひとつである。同様の形式には XML があるが、それに比べて軽量な JSON のほうがスキーマ記述には向いていると判断した。2.1 節で述べたようにテキスト形式には様々な欠点があるものの、スキーマデータは軽量であるためこの欠点は問題にならない。

以上を踏まえ、スキーマ記法の BNF は以下の通りとなった。

```

<object> ::= "{" <members> "}"
<members> ::= <member> | <member> "," <members>
<member> ::= <name> ":" <type>
<type> ::= "string" | <int> | <float> | <array> | <object>
<int> ::= <sign> "int" <int-bits>
<sign> ::= "s" | "u"
<int-bits> ::= "8" | "16" | "32" | "64"
<float> ::= "float" <float-bits>

```

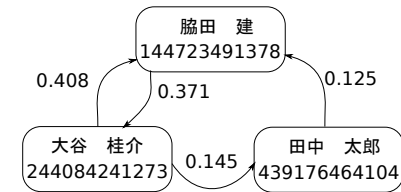


図 1 Twitter のフォロー・フォロワー関係ネットワークの例
Fig.1 Example of follows and followers network in Twitter.

```
<float-bits> ::= "32" | "64"
```

```
<array> ::= "[" <type> "]"
```

ただし非終端記号 $\langle name \rangle$ は正規表現 $[_a-zA-Z][_a-zA-Z0-9]^*$ で表される文字列を示す。使用できる文字列を制限しているのは、この文字列をプログラムコード中で変数名や関数名として使用するためである。また開始記号は $\langle object \rangle$ である。

スキーマの記述例として図 1 のような Twitter のフォロー・フォロワー関係ネットワークを解析するためのユーザーデータスキーマを示す。

```

{ "users": [ { "id": "uint32",
               "name": "string",
               "follows": [ { "id": "uint32",
                              "weight": "float64" } ],
               "followers": [ { "id": "uint32",
                                "weight": "float64" } ] } ] }

```

まず複数のユーザーデータを扱うために、ユーザーひとり分のデータオブジェクトを配列にし、トップレベルオブジェクトの子要素として “users” と定義している。ユーザーオブジェクトは id, name, follows, followers の 4 つの子要素を持つ。このうち follows と followers は他のユーザーとのつながりを示す配列データである。配列の各要素は対象のユーザー ID と解析用のつながりの重みを持っている。

スキーマは JSON 形式で記述しているが、定義されるデータ構造は JSON や JavaScript などの言語から独立した構造であることに注意していただきたい。

3.3 スキーマのコンパイル

3.2 節ではユーザーが独自のデータ構造を定義するためのスキーマについて説明した。この独自のデータ構造をプログラムで利用するには、データの入出力の他にその構造をプログ

ラミング言語でのデータ型定義に変換する必要がある。この変換を手作業で設計する場合、変換が正しいか検証しなければならない。またスキーマが変更されるたびに変換処理もスキーマに合わせて修正することになる。これらの実装を自動化するのがスキーマコンパイラである。スキーマコンパイラはスキーマの構造に対応した入出力 API を生成する。API はデータをプログラミング言語のデータ型として読み込む機能を提供する。この API 自動生成によってスキーマで定義したデータ構造をすぐにプログラムに取り込んで利用できるようになる。

スキーマコンパイラのもうひとつの利点は複数のプログラミング言語に対応できることである。通常、あるデータ形式を複数の言語で利用する場合は、そのデータ形式のための入出力機能やデータオブジェクトをそれぞれの言語に対して作らなければならない。提案形式のスキーマは言語から独立した構造であり、複数の言語に対応させることができる。さらに API は自動生成できるため、ユーザーは利用する言語向けに API を生成するだけで好きな言語で提案形式のデータを利用できる。プログラミング言語にはそれぞれ得手不得手があり、ユーザーによって使いやすい言語も異なる。状況に合わせて言語を使い分けられることで実装スピードや処理能力の向上につながる。現在スキーマコンパイラは Java, C++ をサポートしており、今後、分散並列言語 X10 や JSON をサポートする予定である。

ここからは主に Java 向けのコンパイラについて説明する。自動生成する API は入出力部分とオブジェクト部分に分かれる。入出力部分を担当するのは DataManager クラスである。データ読み込みでは DataManager はデータファイルからバイナリデータの構成を読み出し、それをオブジェクトとして取り出す。データ書き出しでは出力するオブジェクトを DataManager が受け取ってバイナリデータを作り、最後にデータを自己記述情報やスキーマコンパイラと共にまとめてひとつのファイルを生成する。オブジェクト部分はスキーマの構造が反映されてクラスやメソッドが生成される。スキーマで定義したオブジェクト型がクラスとなり、オブジェクトの子要素がクラスのメンバとなる。

例えば 3.2 節で挙げたスキーマ例からは図 2 に示すクラス構成の API が生成される。生成されるクラスは入出力部分を担う DataManager とオブジェクト部分の RootObject, UsersUnit, FollowsUnit, FollowersUnit である。DataManager にはデータファイルを読み込み RootObject にアクセスするメソッドと、各オブジェクトクラスを出力するためのメソッドが定義されている。RootObject はトップレベルオブジェクトを表すクラスである。スキーマでは users 配列を子要素に持つオブジェクトとして定義されていた。UsersUnit は users 配列の 1 要素を表すクラスである。すなわち UsersUnit の配列が users 配列に相当す

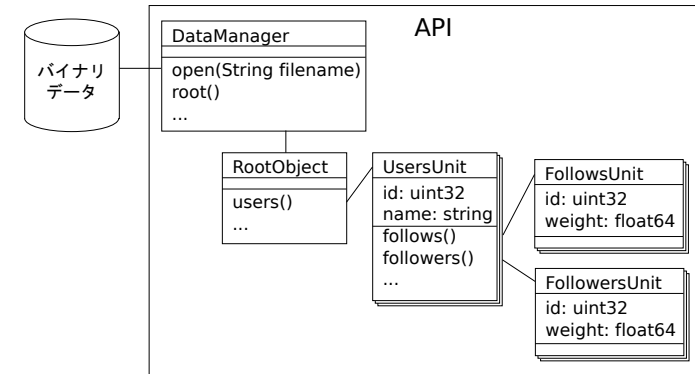


図 2 スキーマ例から生成した Java API のクラス構成

Fig. 2 Java API class structure generated from example of schema.

表 1 スキーマの型と変換後の Java の型の対応

Table 1 Types in schema and transformed types in Java.

スキーマの型	Java の型
sint8	byte
sint16, uint8	short
sint32, uint16	int
sint64, uint32	long
uint64	BigInteger
float32	float
float64	double
string	String
配列	アクセスメソッド
オブジェクト	Object/Unit クラス

る。このクラスは id と name をメンバフィールドに持ち、follows 配列や followers 配列にアクセスするためのメンバメソッドが定義されている。FollowsUnit と FollowersUnit はそれぞれ follows 配列と followers 配列の 1 要素を表すクラスである。それぞれ id と weight というメンバフィールドを持っている。

次にスキーマでのデータ型がどのように Java コードとして変換されるか見ていく。スキーマの型と変換後の Java の型の対応をまとめたのが表 1 である。まず float32/64 と string は単純に Java の float, double, String となる。符号付き整数 (sint8/16/32/64) も単純

表 2 スキーマ例の users 配列から定義されるメソッド
Table 2 Methods defined from "users" array in example of schema.

返り値の型	メソッド	用途
UsersUnit	getUsersUnit(int pos)	ランダムアクセス
Iterator<UsersUnit>	usersIterator()	シーケンシャルアクセス
Iterable<UsersUnit>	users()	シーケンシャルアクセス
int	getUsersLength()	配列の長さ取得

な変換だが, Java には符号なし整数型がないため uint8/16/32 は 2 倍のサイズの符号付き整数で代用する. それでも uint64 には適当なプリミティブ型がないため, BigInteger を利用する. 配列はアクセスメソッドを定義する. 例えば先程の例の users 配列の場合は表 2 の 4 つのメソッドが定義される. 一部分のみにアクセスする場合はランダムアクセスメソッドを使い, シーケンシャルアクセスする場合はイテレータを使う. オブジェクトは基本的に Object クラスとなる. 例えば要素名が edge というオブジェクト型は EdgeObject というクラスが生成される. また例の users 配列のように配列の要素にオブジェクト型がある場合は Unit クラスになる. Object と Unit に本質的な違いはないが, 名前を変えることで配列の 1 要素のオブジェクトであることが分かるようになっている.

提案形式のデータを処理するプログラマは, スキーマコンパイラで生成した API を用いて, データ処理のアプリケーションプログラムを作成する. その具体的な方法を 3.2 節で示したスキーマ例を使って説明する. まずスキーマコンパイラでスキーマから API を生成する. スキーマコンパイラは JAR 形式のスタンドアロンアプリケーションであり, 次のように実行する.

```
java -jar compiler.jar -generate twitter_network.json twitterNetwork.jar
```

これで図 2 に示した 5 クラスを含んだ twitterNetwork.jar という API が生成される. 次に API を使ってデータ処理プログラムを作る. ここでは例としてデータに含まれる各ユーザーのフォロワーとのつながりの重みを和が 1 になるように正規化し新たなデータファイルを作るメソッドを示す (例外処理などは省略している).

```
void normalizeFollowers() {
    DataManager dm = new DataManager("input.jar");
    RootObject root = dm.root();
    for (UsersUnit user : root.users()) {
        double weightsum = 0;
```

```
        for (FollowersUnit follower : user.followers())
            weightsum += follower.weight;
    }
    for (FollowersUnit follower : user.followers()) {
        follower.weight /= weightsum;
        dm.writeFollowersUnit(follower);
    }
    dm.writeUsersUnit(user);
}
dm.writeRootObject(root);
dm.writeFile("output.jar");
}
```

まず DataManager を用いてデータファイルを読み込む. 次にトップレベルオブジェクト RootObject にアクセスし, Iterable<UsersUnit> を返す users メソッドを使い各ユーザーデータにアクセスする. そしてフォロワーとのつながりの重みの和を求め, 正規化を行う. RootObject.users メソッドと同様に UsersUnit.followers メソッドで全ての FollowersUnit に対して処理を行なっている. 変更を行った FollowersUnit は DataManager の書き込みメソッドに渡し新規データファイル作成の準備をする. UsersUnit や RootObject でも同様の書き込みメソッドを適用している. これらのオブジェクトには直接変更を行ってはいないが, 新規データを構成するために必要なデータなので出力が必要である. 最後に新規ファイルに準備したデータをまとめて書き出すことで処理が完了する. ここに示したプログラムから利用している API は, バイナリファイルのフォーマットやエンディアンに関わる低レベルな処理を完全に隠蔽し, 計算機アーキテクチャ間, プログラミング言語間の差異を吸収する抽象レイヤーを形成している.

4. 評価

3 節ではテキスト形式にあったアクセス性能の問題, バイナリ形式の単体では利用方法が分からない問題, スキーマに対応した API を用意しなければならない問題を解決する手法を提案した. 本節ではこれらの提案手法がそれぞれの問題をどの程度解決できているか評価する.

4.1 アクセス性能

アクセス速度を検証するために, 1000 万個の倍精度浮動小数点数配列データへ回数を変

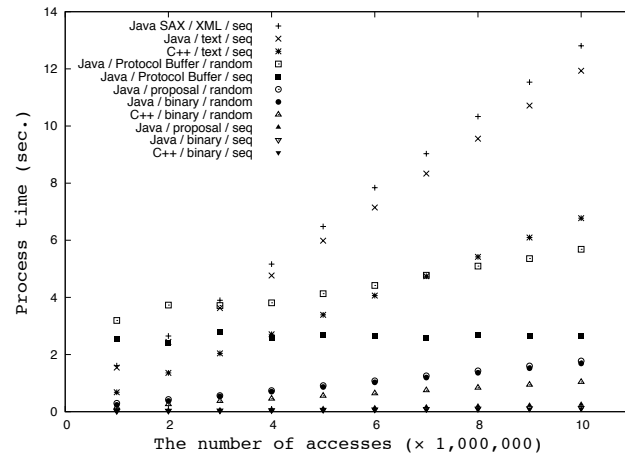


図3 倍精度浮動小数点数データの読み込み時間
Fig. 3 Process time of loading double datas.

えながらアクセスし、データの合計を求める処理時間を計測する実験を行った。使用したデータは Java の Math.random メソッドで生成したランダムな実数列である。実験環境は Mac OS X 10.6.8, Intel Core 2 Duo 2.53GHz, メモリ 4GB である。実験を行ったデータ形式とプログラムは提案形式の Java API, Java 標準 API を用いたテキストおよびバイナリデータ読み込みプログラム^{*1}, Protocol Buffer を読み込む Java プログラム, XML を読み込む Java SAX プログラム, C++ の標準関数やクラスを用いたテキストおよびバイナリデータ読み込みプログラム^{*2}である。バイナリ形式ではランダムアクセスとシーケンシャルアクセス、テキスト形式ではシーケンシャルアクセスのみで実験した。

結果は図3、図4のようになった。図4はバイナリ形式の結果を拡大表示したものである。ただし性能に劣る Protocol Buffer は除いた。また実験に用いたデータのサイズは表3となった。まず提案形式がテキスト形式に比べて大幅に短い処理時間になっていることが分かる。本実験のようにテキスト形式で実数データを扱う場合、読み込んだ文字列データを数値に変換する必要がある上にデータの文字列表現はバイナリ形式よりサイズが大きくなり、

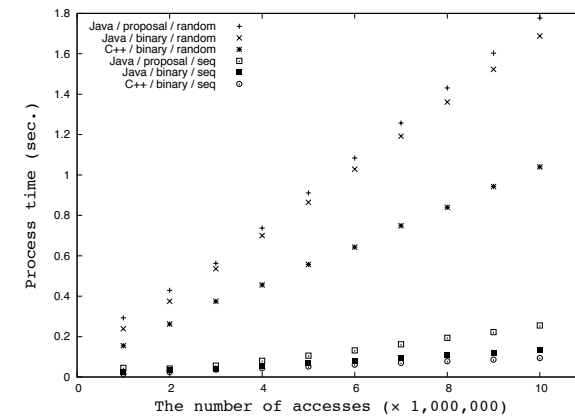


図4 倍精度浮動小数点数バイナリデータの読み込み時間
Fig. 4 Process time of loading double binary datas.

表3 実験データのサイズ
Table 3 Sizes of experimental datas.

データ形式	サイズ
バイナリ形式	76.3MB
提案形式	76.4MB
Protocol Buffer	85.8MB
テキスト形式	183.8MB
XML	250.5MB

読み込みにかかる時間も長くなってしまふ。これらの問題を解決できたことで、バイナリ形式を採用した効果があったと言える。

Protocol Buffer はアクセス回数が増えても処理時間が大きく変化しないが提案形式よりは遅くなっている。この特徴は Protocol Buffer はデータ読み込みの際、データ全体を一度に読み込んでいるためである。また本提案のデータ表現においては、データとスキーマは分離されているが、Protocol Buffer の場合、各データごとに型情報を与えている。このため、同一の型のデータが連鎖する配列のような構造の場合、型情報を保存するために大幅なオーバーヘッドを要することが Protocol Buffer のアクセス性能劣化の原因として考えられ

*1 テキスト形式では BufferedReader クラス、バイナリ形式では FileInputStream クラスを用いた。

*2 テキスト形式では ifstream クラス、バイナリ形式では mmap 関数を用いた。

る。提案形式の性能向上にはメモリマップでファイルアクセスしていることや static オブジェクトを活用しメモリ割り当てを削減していること¹⁶⁾が貢献している。

提案形式と Java 標準 API プログラムを比較すると、ランダムアクセスとシーケンシャルアクセスのどちらでも提案形式が遅くなっている。ランダムアクセスでは提案形式の処理時間は約 1.05 倍でほぼ同じ速度が出ているが、シーケンシャルアクセスでは約 1.92 倍遅くなっている。これは読み込み API でシーケンシャルアクセスを行うとき、内部ではランダムアクセスと同じようにデータの位置を求めてアクセスを行っているためである。シーケンシャルアクセスを位置計算を行わない専用の読み込み処理にすることで改善できる。また C++ のバイナリ形式プログラムは Java 標準 API プログラムより処理速度が速いことから、提案形式のスキーマコンパイラで C++ に対応した API を生成すれば速度向上が見込める。

4.2 スキーマの表現力

提案形式のスキーマは整数型や実数型、文字列型という基本的なデータ型と配列型やオブジェクト型というデータ構造を表す手段を持っている。このため階層構造を取るデータ構造をユーザー独自に定義することができる。例としては表形式のデータや JSON のような構造を持ったデータが表現できる。

このスキーマの欠点はユーザーが定義した構造を再利用できないことである。XML や 2.2.3 節で挙げた通信プロトコル用形式では定義した構造に名前をつけて他の要素の型として利用することができる。これによって例えば循環参照を持つ構造を定義することができる。また異なる箇所でもバラバラに定義された同じ構造をひとつの定義に集約することでスキーマが簡潔になり保守がしやすくなる。提案形式のスキーマでもこのような定義ができるような設計を考えたい。

4.3 スキーマと入出力 API の対応

ここではスキーマコンパイラで自動生成した Java API がどれだけスキーマに対応した構造を取れているか考察する。スキーマの各型と Java コードの対応は表 1 の通りである。実数型や文字列型の対応は自然だが、符号なし整数の対応は Java に符号なし整数型がないため値域が収まる程度の対応しか取れていない。スキーマが C 言語のような符号なし整数型のある言語に依存した型になっていることもあり、スキーマと共に対応関係の設計を検討したい。

オブジェクト型に対応するオブジェクトクラスは、子要素をメンバフィールドに持つ点はスキーマを適切に反映している。しかしオブジェクトクラス同士には、読み込みの機能的には階層があるが構造的には階層を取っていない。3.3 節後半の API を用いたコード例では、

FollowersUnit オブジェクトにアクセスするには UsersUnit オブジェクトのメソッドを利用する必要があり、この点では階層がある。しかし FollowersUnit と UsersUnit はそれぞれ独立したクラスとして定義されており、その構造には階層がない。クラス内クラスとして定義するなどで階層構造を反映できるような対応方式を導入したい。

5. ま と め

データ形式はテキスト形式とバイナリ形式の 2 種類に分類できる。テキスト形式は人間が読んで内容を理解できるため扱いやすいが、全てのデータをテキストにしなければならず、ランダムアクセスができない。バイナリ形式は任意のバイト列を使うことができるが、テキスト形式と違い人間が読むのは困難なので、単体で種類や内容を把握するのが難しい。また使用するデータ構造やプログラミング言語に対応した入出力 API を用意しなければいけなかった。

本稿では多言語に対応した自己記述を持ったバイナリデータ形式を提案することでこれらの問題を解決する。まず処理能力に優れるバイナリ形式を採用した。バイナリ形式はプログラムで使うデータを変換することなくバイト列のまま記録することができるためデータサイズや読み書きの効率が良く、ランダムアクセスも可能である。これによって提案形式で大規模なデータを扱いやすくなった。

提案形式はデータに自己記述を持たせることでバイナリ形式の扱いにくさを解消している。自己記述情報はデータ構造を表すスキーマやデータの種類、利用方法、作成者や作成方法などがある。これらの情報をデータファイルに添付することで、テキスト形式のようにいつでも内容を理解できる。この特徴はデータの内容をよく知らない人とデータファイルを共有したいときなどに有用である。

入出力 API の問題は、スキーマから API を自動生成することによって解決する。スキーマの構造に対応した API を生成し、プログラム中でデータを適切な構造として扱えるようにしている。ユーザーは利用するスキーマやプログラミング言語に対応させた API を用意する負担がなくなる。そのため目的となるデータ処理実装に集中できる。またデータの特性や目的に合わせてスキーマや言語を選ぶことができ、最適な環境でデータ処理を行える。

これらの特徴によって提案形式は科学技術計算データのようなデータが扱いやすくなっている。共同利用や様々なスキーマを持ったデータ、環境や言語の違うプログラムによる処理といった状況で起こりうる負担を提案形式が吸収し、最適な処理を行える。

今後の課題はスキーマの表現力を向上させることや生成する API の改良がある。また API

を多言語に対応させること、特に X10 のような並列言語に対応させることで大規模なデータ解析をサポートする設計も考えたい。

謝辞 本研究は独立行政法人科学技術振興機構より戦略的創造推進事業 CREST の研究助成（課題「ポストペタスケールシステムにおける超大規模グラフ最適化基盤」，研究代表者：藤澤克樹）を受けております。

参 考 文 献

- 1) Isenburg, M. and Snoeyink, J.: Binary compression rates for ASCII formats, *proceedings of the eighth international conference on 3D Web technology (Web3D '03)*, New York, NY, USA, ACM, pp. 173–209 (online), DOI:<http://doi.acm.org/10.1145/636593.636619> (2003).
- 2) Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E. and Yergeau, F.: Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation, W3C (online), available from <http://www.w3.org/TR/2008/REC-xml-20081126/> (accessed 2011-02-16).
- 3) Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F. and Cowan, J.: Extensible Markup Language (XML) 1.1 (Second Edition), W3C Recommendation, W3C (online), available from <http://www.w3.org/TR/2006/REC-xml11-20060816/> (accessed 2011-02-16).
- 4) Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627 (2006).
- 5) Schneider, J. and Kamiya, T.: Efficient XML Interchange (EXI) Format 1.0, W3C (online), available from <http://www.w3.org/TR/2011/PR-exi-20110310/> (accessed 2011-01-25).
- 6) Kangasharju, J., Tarkoma, S. and Lindholm, T.: Xebu: A Binary Format with Schema-Based Optimizations for XML Data, *Web Information Systems Engineering – WISE 2005* (Ngu, A., Kitsuregawa, M., Neuhold, E., Chung, J.-Y. and Sheng, Q., eds.), Lecture Notes in Computer Science, Vol.3806, Springer Berlin / Heidelberg, pp.528–535 (online), DOI:http://dx.doi.org/10.1007/11581062_44 (2005).
- 7) Chiu, K., Devadithya, T., Lu, W. and Slominski, A.: A binary XML for scientific applications, *e-Science and Grid Computing, 2005. First International Conference on*, pp.8 pp. –343 (online), DOI:10.1109/E-SCIENCE.2005.1 (2005).
- 8) BSON Project: BSON - Binary JSON, BSON Project (online), available from <http://bsonspec.org/> (accessed 2011-10-01).
- 9) Chedeau, C.: BinaryParser – Binary File Parsing revisited using Javascript, Curse (online), available from <http://blog.vjeux.com/2011/javascript/binaryparser-unleash-javascript-power.html> (accessed 2011-10-01).
- 10) Google: Protocol Buffers, Google (online), available from <http://code.google.com/apis/protocolbuffers/> (accessed 2011-01-25).
- 11) Furuhashi, S.: The MessagePack Project, The MessagePack Project (online), available from <http://msgpack.org/> (accessed 2011-10-01).
- 12) Slee, M., Agarwal, A. and Kwiatkowski, M.: Thrift: Scalable Cross-Language Services Implementation, Facebook (online), available from <http://incubator.apache.org/thrift/static/thrift-20070401.pdf> (accessed 2011-10-01).
- 13) The Apache Software Foundation: Apache Avro™, The Apache Software Foundation (online), available from <http://avro.apache.org/> (accessed 2011-10-01).
- 14) Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data, *ACM Trans. Comput. Syst.*, Vol.26, pp.4:1–4:26 (online), DOI:<http://doi.acm.org/10.1145/1365815.1365816> (2008).
- 15) The Apache Software Foundation: The Apache Cassandra Project, The Apache Software Foundation (online), available from <http://cassandra.apache.org/> (accessed 2011-10-01).
- 16) 大谷桂介：スキーマを持った実行可能なバイナリデータ形式，東京工業大学理学部情報科学科 (2011).