

オブジェクト直列化技法の比較検討

Comparative Study of Object Serialization Techniques

前田 和昭

Kazuaki Maeda

本稿では、平成 22 年度に研究活動を開始したプロジェクト 52e の概要を説明し、その成果の一つとして、XML で表現可能な構造化データを対象とした場合のオブジェクト直列化技法について概説したあとで、定量的・定性的な比較について述べる。さらに、構造化データ処理に対するアスペクト指向プログラミングの適用可能性について考察する。

キーワード: オブジェクト直列化, XML, スキーマ定義, アスペクト指向プログラミング

1 はじめに

1.1 プロジェクトの目的と構成員

コンパイラは、プレーンテキストで保持されるソースコードを読み込み、字句と構文を解析し、ソースコードを表現するための木を作り上げる。この木のことを AST (抽象構文木, Abstract Syntax Tree) と呼ぶ。現在主流のプログラミング言語である Java や C# の AST を、研究代表者の前田が構築した経験から、AST は 100~200 種類のノードが絡まった複雑な構造となることが分かっている。

本プロジェクトでは、これまでコンパイラ開発で何度も AST を構築してきた経験を一般化し、他分野へ適用することを狙い、データ表現のための DSL (ドメイン特化言語, Domain Specific Language) を設計し、DSL からアプリケーションの一部を生成するソフトウェアツールを開発し、新しいアプリケーション開発手法の提案を試みる。

本プロジェクトの構成員は下記の 3 名である。

- 代表者: 前田和昭 (中部大学 経営情報学部 経営情報学科 教授)

役割: これまで行ってきた言語処理システム、データベース、ソフトウェア工学に関する経験をもとにして、データ表現のための言語の開発と、アスペクト指向プログラミングが可能となる拡張を試みる。

- 研究協力者: 竹下 亨 (元) 中部大学 大学院 経営情報学専攻 教授)

役割: これまでの豊富な経験と XML に関する最新の情報収集により、本プロジェクトの活動を支援する。

- 研究協力者: 福田浩章 (芝浦工業大学 工学部 准教授)

役割: Web アプリケーション開発の豊富な経験をもとにして、Web アプリケーションの理想的な姿を追い求めながら共同のアイデアを作り上げ、本プロジェクトの研究活動を支援する。

1.2 プロジェクトの概要と成果

現在、データ表現のために XML が広く使われている。XML は、複数のプラットフォーム (コンピュータ、オペレーティングシステム、プログラミング言語) で利用可能であり汎用性が高い。教科書などでは、XML の利点として、人間が判読可能であることがあげられている。しかし、コンピュータで解析するための設計となっているため、人間が判読し易いとは言えない。また、多数の仕様が標準化され汎用的な表現であるように言われているが、仕様が複雑すぎて全てを理解することは難しい。さらに、名前空間を利用して異なる種類のデータを一つの XML に混在させることができるが、複雑な XML をさらに判読困難にしている。

そこで本プロジェクトでは、XML の欠点を克服することを目標にして、XML に似ているものの、汎用的であることと単純で人間が理解しやすいことの両方を実現するためのデータ表現を提案することにした。具体的には、スクリプト言語として最近脚光を浴びている Ruby をデータ表現のために利用して、データ表現を Ruby 構文で記述するための DSL を設計した。さらには、DSL を定義するための言語を準備し、ソフトウェアツールを使ってデータ操作のためのプログラムを生成可能にした。この生成されたプログラムをうまく使いこなすことで、データ中心のアプリケーションが容易に作成できる。また、アスペクト指向プログラミングの成果とコンパイラ開発の豊富な経験から、アプリケーション開発のための新しい手法を検討した。

これまで約 3 年間にわたるプロジェクト活動の結果として発表した論文のうち代表的なもの（国際会議論文 7 編と論文誌論文 1 編）を以下に記す。

1. A Scala-Based Domain Specific Language for Structured Data Representation, 5th International Conference on Software and Data Technologies (2010 年 7 月).
2. Executable Representation for Structured Data Using Ruby and Scala, 10th International Symposium on Communications and Information Technologies (2010 年 10 月).
3. Data Processing on Executable Data Representation with an Aspect Weaver, The 14th IASTED International Conference on Software Engineering and Applications (2010 年 11 月).
4. Visualization of Code Clone Detection Results and the Implementation with Structured Data, International Conference on Software and Data Engineering (2011 年 3 月).
5. A Domain Specific Language to Represent Structured Data for Multiple Programming Languages, IADIS International Conferences Informatics 2011 (2011 年 7 月).
6. Structured Data Representation for Multiple Programming Languages on Java VM, International Conference on Computational Techniques and Artificial Intelligence (2011 年 10 月).
7. Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats,

The 2nd International Conference on Digital Information and Communication Technology and its Applications (2012 年 5 月).

8. Comparative Survey of Object Serialization Techniques and the Programming Supports, Journal of Communication and Computer, Vol.9, No.8 (2012 年 8 月).

本稿は、上記の論文 8 を邦訳し、それを土台として内容を充実させながら研究報告書として書き改めたもので、オブジェクト直列化の代表例 6 つ（Java オブジェクト直列化, IDL, XStream, Protocol Buffers, Apache Avro, MessagePack）を取り上げ、定性的な観点と定量的な観点から比較する。また、アスペクト指向プログラミングの適用可能性について言及する。

2 オブジェクト直列化技法の比較

ここでは、共通の例題を設定して 6 つの技法を概観し、定性的・定量的観点からそれらを比較する。定性的な比較では、スキーマ定義が必要かどうか、スキーマコンパイラが必要かどうか、直列化の形式がアスキーかバイナリか、どのプログラミング言語をサポートしているかについて検討する。定量的な比較では、各技法を使って共通例題を直列化してファイルに書き出し、そのサイズを測定して検討する。

2.1 背景

古くから研究開発が進められ、成熟したソフトウェアとして、コンパイラがある。コンパイラは、開発者が記述したソースコードを読み込み、オブジェクトコードを生成する。このコンパイラは、フロントエンドとバックエンドに分割することができ、フロントエンドでは、プレインテキストを解析し、ソースコードの構造を表現する AST を作り出す。AST には、ソースコードの構文情報と意味情報が含まれている。コンパイラのバックエンドは、フロントエンドから AST を受け取り、オブジェクトコードを生成する。

著者は、Java コンパイラのフロントエンドをゼロから作成した経験がある。このとき、AST を構築するために、約 200 のクラス、1,000 以上のメソッドとフィールドを定義する必要があり、非常に厄介な作業であった。もし、逆エンジニアリングツールやソース

コード検索エンジンのように、ソースコード解析を必要とするソフトウェアツールを作ろうとすると、このようなフロントエンドの構築という厄介な作業が必要になる。

書籍 1) を読んでみると、

XML is a buzzword you will see everywhere on the Internet, but it's also a rapidly maturing technology with powerful real-world applications, particularly for the management, display, and organization of data.

という記述があり、XML が多くの分野でデータ記述のために使われていることが分かる。

XML 文書には、文書中心 XML 文書とデータ中心 XML 文書の 2 種類があると言われている^{2, 3)}。文書中心 XML 文書は、表示することを主目的としている。これに対して、データ中心 XML 文書では、粒の細かいデータの集まりを表現する。本稿では、プログラム内部のオブジェクトを直列化することを扱うため、データ中心 XML 文書を対象とする。以降、XML によるデータを扱う場面では、データ中心 XML 文書に限定する。

XML をコンパイラに応用した例として、JavaML⁴⁾がある。JavaML は、Java ソースコードを解析した結果、その構文情報と意味情報を出力するために XML を使って表現したものである。JavaML コンパイラがあれば、ソースコードから AST が構築され、その AST を XML 文書として書き出してくれるため、ソフトウェアツールを構築するときにフロントエンドを新しく作成する必要がなくなる。ソフトウェアツールを作りたいときには、XML ライブラリを利用して、JavaML で表現されたデータを読み込めば、そこから Java ソースコードに関する情報（クラス宣言、メソッド定義、メソッド呼び出しなど）を簡単に取得することができる。

XML が標準化され、いろいろなソフトウェアツールが開発されることになり、また多様なプログラミング言語向けに多くのライブラリが開発されている。その結果として、異なるプラットフォーム（異なる OS やコンピュータ）間でのデータ交換が楽になった。このような XML の利点に対して欠点も見えてきている。

XML は、人間が読めるようにテキスト形式で書かれているが、XML 文書には多くの冗長的なタグ（開始タグと終了タグ）が書き込まれ複雑になりがちで、その理解が難しくなることが多い。この反省をふまえて、

最近では、XML の冗長性を省き簡潔な記述を目指した JSON の人気が高まっている^{5, 6)}。JSON による記述では、JavaScript のオブジェクト記法をそのまま使う。実際に JSON でデータを記述してみると、XML で記述した場合に比べて小さくなることが分かる。

Java の標準ライブラリには、オブジェクト直列化⁷⁾が含まれ、オブジェクト（とそれから到達可能なオブジェクト）をバイトストリームに変換する機能（一般的に直列化（serialization）と呼ばれている）を提供する。また、直列化したバイトストリームを元のオブジェクトに戻す逆直列化（deserialization）と呼ばれる機能も提供されている。Java のオブジェクト直列化は、オブジェクトをファイルに保管するのに便利である。オブジェクトの各フィールドの値はバイナリ形式で保存されるので、直列化後のサイズは、アスキー形式で保存するよりも小さくなるように思えるものの、これまでの経験では、JSON などのアスキー形式よりも大きくなることがほとんどであった。これは、単にバイナリ形式であればサイズが常に小さくなるわけではないことを示している。これらのことから、代表的なオブジェクト直列化の技法を取り上げ、実際に直列化したときの大きさを実験により調査する必要があると考えた。

以下の節では、6つのオブジェクト直列化技法を概観し、それらを定量的・定性的に評価する。そのために、共通例題を使って6つの直列化技法でファイルに書き出し、その大きさを比較する。定性的な面では、以下の4つ

- スキーマ定義が必要かどうか
- スキーマコンパイラが必要かどうか
- 直列化はアスキー形式かバイナリ形式か
- どのプログラミング言語をサポートしているかで比較する。

2.2 オブジェクト直列化の技法

オブジェクト直列化のための6つの技法を比較する共通例題として、図1に示すような（ここではJavaで記述）、注文のための Order クラスと、注文に追加可能なオプションのための Option クラスを考える。また、共通例題に設定される値を示すために、オプションをいくつか追加したときの注文をXML文書で図2に示す。

```

public class Order {
    String product;
    CupSize size;
    String date;
    int price;
    List<Option> options;
}
public class Option {
    String name;
    int price;
}

```

図 1 Java による Order と Option クラスの一部

2.2.1 Java のオブジェクト直列化

Java パッケージに標準で含まれているオブジェクト直列化では、オブジェクトをコード化してバイトストリーム (ObjectOutputStream) へ書き出す。また、バイトストリーム (ObjectInputStream) から読み込み、書き込みのときと等価なオブジェクトを再構築する逆直列化機能も提供する。ObjectOutputStream クラスの writeObject メソッドが直列化を担当し、ObjectInputStream クラスの readObject メソッドが逆直列化を担当する。例えば、図 3 に示すように、オブジェクトの状態を ObjectOutputStream に書き出し、ObjectInputStream からオブジェクトを読み込んでオブジェクトを再構築することができる。Java のオブジェクト直列化では、新たにスキーマを定義する必要がない。直列化したいクラスに “implements Serializable” を追記し、直列化したいときには writeObject メソッドを、逆直列化したいときには readObject メソッドを呼び出すだけでよい。

Java のオブジェクト直列化を使うと、Java Object Serialization Specification⁸⁾に記載されているバイナリ形式で書き出される。しかし、その仕様は Java に依存しているため、他のプログラミング言語で使えない場合がある。Java はアプリケーション開発では人気のあるプログラミング言語の一つであるが、分散環境を考えると、他のプログラミング言語で書かれたプログラムと、データ交換が必要になる場合が多分にあり得る。また、計算資源が乏しい場合、時間制約が厳しい場合、レガシーシステムとの連結が必要な場合などの状況もあり得る。したがって、オブジェクト直列化の機能を設計するときには、複数のプログラミング言語をサポートすることを考慮することが望ましい。

```

<order>
  <product>Caffeinless Coffee</product>
  <size>Grande</size>
  <date>December 21,2012</date>
  <price>460</price>
  <options>
    <option>
      <name>Shot</name>
      <price>50</price>
    </option>
    <option>
      <name>Extra whip</name>
      <price>50</price>
    </option>
    <option>
      <name>Hazelnut</name>
      <price>0</price>
    </option>
    <option>
      <name>Caramel sauce</name>
      <price>0</price>
    </option>
    <option>
      <name>Chocolate sauce</name>
      <price>50</price>
    </option>
    <option>
      <name>Chocolate chip</name>
      <price>50</price>
    </option>
  </options>
</order>

```

図 2 オプションを 6 つ追加した注文のための XML 文書

```

Order anOrder = new Order();
anOrder.setProduct("Caffeinless Coffee");
anOrder.setSize(CupSize.Grande);
anOrder.setDate("December 21,2012");
anOrder.setPrice(460);
List list = new ArrayList();
anOrder.setOptions(list);
Option thisOption = new Option();
thisOption.setName("Shot");
thisOption.setPrice(50);
list.add(thisOption);
// オプションを 5 つ追加するコードは省略する。

FileOutputStream fos
    = new FileOutputStream("order.dat");
ObjectOutputStream oos
    = new ObjectOutputStream(fos);
oos.writeObject(theOrder);

FileInputStream fis
    = new FileInputStream("order.dat");
ObjectInputStream ois
    = new ObjectInputStream(fis);
Order theOrder = (Order)ois.readObject();

```

図 3 writeObject メソッドと readObject メソッドを使った Java プログラムの一部

2.2.2 IDL

1980 年代中頃、コンパイラ開発を支援するために、Interface Description Language (IDL)⁹⁾ が設計され、

```

Structure StarbucksOrder Root Order Is
Order => product : String,
        size    : CupSize,
        date    : String,
        price   : Integer,
        options : Seq Of Option;

For CupSize Use Representation Enumerated;
CupSize ::= Small | Tall | Grande | Venti;
Small =>; Tall =>; Grande =>; Venti =>;

Option => name : String,
        price : Integer;
End

```

図 4 IDL によるスキーマ定義

その開発ツールが作られた¹。IDL は、構造化データを定義するための言語である。共通例題を IDL で定義した例を図 4 に示す。本稿では、このような構造化データの定義のことを「スキーマ定義」(Schema definition)と呼ぶことにする。

IDL のためのソフトウェアツール群 Scorpion Toolkit は、保守が終了しているものの現在でも利用可能である。このソフトウェアツールの中でも重要なものとして、IDL で記述されたスキーマ定義を読み込み、指定されたプログラミング言語のためのコードを生成する IDL コンパイラがある。IDL コンパイラは、AST を直列化して書き出したり、書き出したデータを逆直列化するためのコードを生成する。このように、スキーマ定義を読み込んで、オブジェクト直列化に関係するコードを生成するコンパイラのことを「スキーマコンパイラ」と呼ぶことにする。IDL を利用する開発者は、スキーマコンパイラが生成するコードを利用して、指定されたプログラミング言語でプログラムを記述する。

Scorpion Toolkit のスキーマコンパイラが生成するコードは、アスキー形式かバイナリ形式のどちらかで構造化データを直列化して書き出す機能を持つ。もし、C 言語をターゲットとして選んだとき、スキーマコンパイラは C 言語のためのデータ構造宣言とマクロ定義と関数定義を生成する。図 5 に生成されたマクロ NOrder, productOfOrder(), sizeOfOrder(), dateOfOrder(), priceOfOrder(), 関数 order_out(), order_in() を利用したプログラムを示す。NOrder は、メモリ割り当てのためのマクロである。productOfOrder(), sizeOfOrder(), dateOf-

¹本稿における IDL とは、OMG IDL¹⁰⁾とは別のシステムである。

```

Order anOrder = NOrder;
productOfOrder(anOrder) = "Caffeinless Coffee";
sizeOfOrder(anOrder) = GrandeToCupSize(NGrande);
dateOfOrder(anOrder) = "December 21,2012";
priceOfOrder(anOrder) = 460;

FILE *ofp = fopen("order-idl.txt","w");
order_out(ofp,theOrder);

FILE *ifp = fopen("order-idl.txt","r");
Order theOrder = order_in(ifp);

```

図 5 IDL を使った C 言語によるプログラムの一部

fOrder(), priceOfOrder() はフィールドへのアクセスのためのマクロであり、order_out() と order_in() は直列化と逆直列化のために生成された関数である。

2.2.3 XStream

XStream¹¹⁾ は、オブジェクトを XML で直列化・逆直列化する Java のためのライブラリである。XStream では、Java のオブジェクト直列化のように Serializable インタフェースを実装する必要はなく、IDL のようにスキーマ定義を書く必要もない。実行時に、直列化すべきオブジェクト（とそこから到達可能なオブジェクト）をリフレクションを使って調べ上げ、その結果を使って直列化する。

XStream を使ってオブジェクトを直列化するには、図 6 に示すように、XMLStream オブジェクトを生成し、toXML メソッドを呼ぶだけでよい。逆直列化す

```

Order anOrder = new Order();
anOrder.setProduct("Caffeinless Coffee");
anOrder.setSize(CupSize.Grande);
anOrder.setDate("December 21,2012");
anOrder.setPrice(460);

XStream xstream = new XStream();
xstream.alias("order", Order.class);
xstream.alias("option", Option.class);
FileOutputStream fos
    = new FileOutputStream("order.xml");
xstream.toXML(anOrder, fos);

FileInputStream fis
    = new FileInputStream("order.xml");
xstream = new XStream();
Order theOrder =(Order)xstream.fromXML(fis);

JsonHierarchicalStreamDriver json
    = new JsonHierarchicalStreamDriver();
xstream = new XStream(json);
xstream.alias("order", Order.class);
fos = new FileOutputStream("order.json");
xstream.toXML(theOrder, fos);

```

図 6 XStream を使った Java プログラムの一部

るには、fromXML メソッドを呼び出し、キャストによりタイプを合わせるだけである。XStream の現在のバージョンでは、JsonHierarchicalStreamDriver クラスを使って JSON による直列化ができる。

デフォルトの設定では、クラス名とフィールド名が直列化後の XML 文書の要素名となる。図 2 に示した、オプションを追加した注文は、XStream を使って出力したものである。

2.2.4 Protocol Buffers

Protocol Buffers ¹²⁾ は、構造化データを効率的にコード化することを目標に Google のメンバが開発したものである。小さく速く直列化できるように、バイナリ形式を使って直列化する。Protocol Buffers を使ったプログラムを作成するには、最初に、図 7 に示すようなスキーマ定義を準備する必要がある。ユーザは、メッセージ型を定義する。メッセージ型の詳細定義では、データ型、フィールド名、唯一の番号を記述する。データ型は、整数型、浮動小数点型、論理型、文字列型、列挙型、他のメッセージ型のどれかである。Protocol Buffers では、スキーマコンパイラがスキーマ定義を読み、各フィールドへのアクセッサ（例えば、getProduct() と setProduct()）や、直列化・逆直列化するためのメソッド、内部データをカプセル化するための Protocol Buffers 専用のビルダクラスなどを生成する。図 8 に、生成されたアクセッサを使い、直列化・逆直列化する Java プログラムを示す。Protocol Buffers の現在のバージョンは、3つのプログラミング言語 Java, C++, Python をサポートする。

```
option java_package = "protobuf";
message Order {
  enum CupSize {
    Small = 1; Tall = 2;
    Grande = 3; Venti = 4;
  }
  required string product = 1;
  required CupSize size = 2;
  required string date = 3;
  required int32 price = 4;
  message Option {
    required string name = 1;
    required int32 price = 2;
  }
  repeated Option options = 5;
}
```

図 7 Protocol Buffers のためのスキーマ定義

```
Order.Builder anOrder = Order.newBuilder();

anOrder.setProduct("Caffeinless Coffee");
anOrder.setSize(Order.CupSize.Grande);
anOrder.setDate("December 21, 2012");
anOrder.setPrice(460);

FileOutputStream fos
    = new FileOutputStream("order.dat");
anOrder.build().writeTo(fos);

FileInputStream fis
    = new FileInputStream("order.dat");
Order theOrder = Order.parseFrom(fis);
```

図 8 Protocol Buffers を使った Java プログラムの一部

2.2.5 Apache Avro

Apache Avro ¹³⁾ は、Hadoop サブプロジェクトとして開発された直列化フレームワークであり、データの永続化と Hadoop ノード間通信のための直列化データ形式のために開発された。Apache Avro を使うには、図 9 に示すような、JSON で記述されたスキーマ定義が必要となる。動的プログラミング言語と統合するために、Apache Avro ではコードを生成することなく、実行時にスキーマ定義を読み込み、それからすぐに直列化を実行できる。

Apache Avro を使って注文オブジェクトを直列化・逆直列化する Java プログラムを図 10 に示す。コード生成が必要ないかわりに、put メソッドを使って名前-値ペアを設定する。このプログラムは、最初の Encoder がオブジェクトをバイナリ形式でファイルに書き出し、Decoder がそのファイルからオブジェクトを逆直列化する。

```
{
  "namespace": "avro",
  "name": "Order",
  "type": "record",
  "fields": [
    {"name": "product", "type": "string"},
    {"name": "size", "type": "enum",
     "symbols": [
       "Small", "Tall", "Grande", "Venti"
     ]
    },
    {"name": "date", "type": "string"},
    {"name": "price", "type": "int"},
    {"name": "options", "type": {
      "type": "array",
      "items": "avro.Option"
    }}
  ]
}
```

図 9 Apache Avro のためのスキーマ定義

```
File file = new File("schema-order.avro");
Schema schema =
    AvroUtils.parseSchema(file);
GenericRecord r = new
    GenericData.Record(schema);
r.put("product", "Caffeinless Coffee");
r.put("size", "Grande");
r.put("date", "December 21,2012");
r.put("price", 460);

FileOutputStream fos
    = new FileOutputStream("order.dat");
EncoderFactory ef = EncoderFactory.get();
Encoder enc = ef.binaryEncoder(fos,null);
GenericDatumWriter<GenericRecord> writer
    = new GenericDatumWriter<GenericRecord>(schema);
writer.write(r,enc);

FileInputStream fis
    = new FileInputStream("order.dat");
DecoderFactory df = DecoderFactory.get();
Decoder dec = df.binaryDecoder(fis,null);
GenericDatumReader<GenericRecord> reader
    = new GenericDatumReader<GenericRecord>(schema);
GenericRecord theOrder = reader.read(null,dec);

fos = new FileOutputStream("order.json");
enc = ef.jsonEncoder(schema,fos);
w = new GenericDatumWriter<GenericRecord>(schema);
writer.write(r,enc);
```

図 10 Apache Avro を使った Java プログラムの一部

EncoderFactory クラスは、2 種類（バイナリ形式と JSON 形式）の Encoder を提供する。もし、JSON 形式で直列化したいときには、Encoder の実装を

```
Encoder enc = ef.binaryEncoder(fos,null);
```

から、

```
Encoder enc = ef.jsonEncoder(schema, fos);
```

へ変更するだけでよい。Encoder を JsonEncoder に変更して直列化した後、図 11 に示す JSON データがファイルに書き出される。

```
{"product":"Caffeinless Coffee", "size":"Grande", "
  date":"December 21,2012", "price":460, "
  options":[]}
```

図 11 Apache Avro を使ったときの JSON による注
文オブジェクトの出力

2.2.6 Message Pack

MessagePack¹⁴⁾は、バイナリ形式を使って効率的にオブジェクト直列化するためのライブラリである。スキーマ定義を準備する・しないは、利用者が選択できる。もし、スキーマ定義を使うことなく、Java をプロ

```
@MessagePackMessage
public class Order {
    public String product;
    public String size;
    public String date;
    public int price;
    public List<Option> options;
}
```

図 12 MesasgePack のための Order クラス（スキーマ定義を書かずに使う場合）

```
namespace mpack
enum CupSize {
    0: Small
    1: Tall
    2: Grande
    3: Venti
}
message Option {
    1: string name
    2: int price
}
message Order {
    1: string name
    2: CupSize size
    3: string date
    4: int price
    5: list<Option> options
}
```

図 13 MessagePack のためのスキーマ定義

グラミング言語として選択するなら、図 12 にしめすように、@Message を public フィールドに書き込むことで、直列化を指示することになる。

MessagePack のためのスキーマ定義を図 13 に示す。このスキーマ定義の構文は、各フィールドに唯一となる番号を記述する点で、Protocol Buffers のスキーマ定義に似ている。現在の実装では、スキーマ定義のサポートは Java だけであるが、仕様書には、複数のプログラミング言語をサポートすると記載されている。

プログラムがオブジェクトを直列化したいときには、図 14 に示すように、MessagePack クラスの pack メソッドを呼び出し、バイト列からオブジェクトを逆直列化するときには、MessagePack クラスの unpack メソッドを呼び出す。現在の実装では、C, C++, C#, Java, D, Go, Node.JS, JavaScript, Perl, Python, Ruby, Scala, Lua, PHP, Erlang, Haskell, OCaml の 17 種類のプログラミング言語がサポートされている。

```

Order anOrder = new Order();
anOrder.setProduct("Caffeinless Coffee");
anOrder.setSize("Grande");
anOrder.setDate("Decmber 21,2012");
anOrder.setPrice(460);
anOrder.setOptions(new ArrayList<Option>());

MessagePack mpack = new MessagePack();
FileOutputStream fos
    = new FileOutputStream("order.dat");
Packer packer = mpack.createPacker(fos);
packer.write(anOrder);

mpack = new MessagePack();
FileInputStream fos
    = new FileInputStream("order.dat");
Unpacker unpacker = mpack.createUnpacker(fos);
Order thisOrder = unpacker.read(Order.class);

```

図 14 MessagePack を使った Java プログラムの一部

2.2.7 JSON ライブラリ

6 つの直列化技法の他に、JSON のためのオブジェクト直列化のライブラリとして、Flexjson¹⁵⁾、Gson¹⁶⁾、Jackson¹⁷⁾、Json-lib¹⁸⁾、json-smart¹⁹⁾、JsonMarshaller²⁰⁾ を紹介する。

Flexjson は、Java オブジェクトをなるべく簡単に JSON に直列化・逆直列化する軽量ライブラリである。

Gson は、もともと Google 社内で使うために開発された。Java のソースコードが手元になく、バイトコードのみの場合でも直列化できるように、単に toJson() と fromJson() と呼び出すだけで良い。

Jackson は、構造化データを扱うために 3 種類の方法（ストリーミング API、木モデル、データバインディング）を提供する。ストリーミング API では、StAX²¹⁾ と同じようにデータを扱う。木モデルでは、構造化データをメモリ内で木構造で表現し、DOM²²⁾ と似た方法でアクセスできるようになっている。データバインディングでは、Java ビーンズと JSON との間でのマッピングを提供する。

Json-lib は、JSON の普及に貢献した Douglas Crockford が作ったライブラリである。これは、Apache Commons²³⁾ (Commons Lang, Commons BeanUtils, Commons Collections, Commons Logging) と ezmorph²⁴⁾ を必要とする。Json-lib の利用者は、これら 5 つの jar ファイルを準備する必要がある。

json-smart は、Java のジェネリックを使ったカスタマイズ性を重視したライブラリである。外部ライブラリは必要としない。

JsonMarshaller は、簡素化と効率化を狙ったライブ

リである。Java のクラスを定義するときに @Entity と @Value アノテーションを書き込む。したがって、直列化したいクラスの全てのソースコードが必要となる。

2.3 オブジェクト直列化の比較

前節では、オブジェクト直列化のための 6 つの技法を説明した。それらを定性的に、

- スキーマ定義が必要か不要か選択可能か
- スキーマコンパイラが必要か不要か選択可能か
- バイナリ形式で直列化するか、アスキー形式で直列化するか。アスキー形式の場合にサポートされているのは、XML か JSON か
- どのプログラミング言語がサポートされているか

の観点から比較した結果を表 1 に示す。もし、複数のプログラミング言語をサポートするなら、IDL や Protocol Buffers のように、スキーマ定義を記述して、そこからプログラミング言語に依存したコードを生成する方が開発者にとって望ましい。このとき、スキーマ定義は利用したいプログラミング言語とは独立した記述になることがほとんどである。MessagePack は、多くの開発者の貢献によって複数のプログラミング言語をサポートするようになった。これは、オープンソースとして世界中に公開したからで、オープンソース活動はプロジェクトの活性化のために重要である。

直列化してファイルに書き出したデータのサイズを測定し、定量的な観点から比較したものを表 2 に示す。IDL, Protocol Buffers, Apache Avro, Message Pack を使ったバイナリ形式での直列化の場合、アスキー形式の場合と比べてサイズが小さくなる。ところが、Java のオブジェクト直列化の場合、バイナリ形式で直列化するものの、直列化したデータのサイズは小さい。内部データを調べてみると、直列化したデータに Java のタイプ情報が多く含まれることが分かり、これがデータサイズを大きくしている。XStream の場合、XML で直列化する場合に比べ、JSON で直列化の方が約 24% 小さくなっている。XML で直列化したときの図 2 と、JSON で直列化したときの図 11 を比べてみると、JSON で直列化した場合の方が簡潔に表現されていることが分かり、これがサイズの差につながっている。6 つの技法の中で、Apache Avro を使ってバイナリ形式で直列化する場合が一番小さくなっている。また、XStream で XML を使って直列化する場合が一番大きくなっている。

表 1 オブジェクト直列化の比較

	スキーマ定義は必要か	コンパイラは必要か	バイナリ形式か アスキー形式か	サポートされる プログラミング言語
Java 直列化	不要	不要	バイナリ	Java
IDL	必要	必要	アスキー, バイナリ	C, Pascal
XStream	不要	不要	XML, JSON	Java
Protocol Buffers	必要	必要	バイナリ	Java, C++, Python
Apache Avro	必要	不要	バイナリ, JSON	C, C++, C#, Java, PHP, Python
MessagePack	選択可能	選択可能	バイナリ	C, Java, Ruby を含む 17 言語

表 2 オブジェクト直列化後のデータサイズ

技法	アスキー か バイナリ	サイズ (bytes)
Java 直列化	バイナリ	516
IDL	アスキー バイナリ	339 135
XStream	XML でアスキー JSON でアスキー	644 489
Protocol Buffers	バイナリ	143
Apache Avro	バイナリ JSON でアスキー	123 298
MessagePack	バイナリ	130

3 XMLとアスペクト指向プログラミング

前節の比較では、スキーマ定義を記述し、そこからプログラミング言語に依存したコードを生成する技法が開発者にとって望ましいと述べたものの、実際には問題があることが分かった。以下では、XML を使ったプログラミングに限定して議論する。

3.1 イベント駆動アプローチと木構造ベースアプローチ

開発者が XML 文書処理するプログラムを作るには、2つのアプローチがある。それが、イベント駆動アプローチと木構造ベースアプローチである。イベント駆動アプローチの代表例が SAX²⁵⁾ と StAX²¹⁾ であり、木構造アプローチの代表例が DOM²²⁾ である。

SAX は、イベント駆動アプローチの API を提供する。それは、XML 文書を読み込み、基本的なコンポーネント（例えば、要素の開始と終了など）を認識して、図 15 の左上端に示すように、SAX ハンドラにイベントを送る。XML 文書の部分処理するためのイベントは一つずつ処理され、もし必要ないものがあれば、無視することで、プログラミング作業は容易になる。開発者がイベントハンドラを駆使して、複数のイベントの結果を保管することもできる。図 16 に、要素の個数を数えるための Java プログラムの一部を示す。SAX を使ってみると、現在の状態を保持するために、状態マシンを作ることがよくある。このプログラミングスタイルは、一般的に、データ処理を複雑にする。

もし、図 15 の右上端に示すように、木構造アプローチでデータを処理したいときは、DOM が使われる。

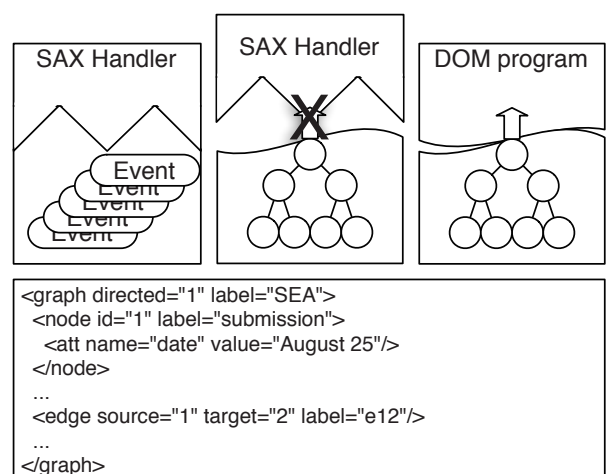


図 15 XML 文書処理する 2つのアプローチ

```

public class SaxModel extends DefaultHandler{
    public void startDocument() {...}
    public void endDocument() {...}
    public void startElement(String nsURI,
        String lName,String qName,Attributes atts)
    {    count++;    }
    public void endElement(String nsURI,
        String lName,String qName) {...}
    private int count=0;
}

```

図 16 Java による SAX ハンドラの一部

```

public int countChildElements(Node n) {
    int count = 0;
    if (n != null) {
        if (n instanceof Element)
            count++;
        NodeList list = n.getChildNodes();
        int length = list.getLength();
        for (int i = 0; i < length; i++) {
            Node cn = list.item(i);
            count += countChildElements(cn);
        }
    }
    return count;
}

```

図 17 Java による DOM プログラムの一部

XML 文書进行处理するために、まず最初に XML 文書を読み込み、パージングして、主メモリ内の木構造として保持する。木を構築したあと、各ノードは、木構造巡回のための API でアクセスできる。DOM API の標準仕様は、W3C²²⁾で定義されている。図 17 に、要素の個数を数える Java プログラムの一部を示す。DOM API を使えば、XML 文書を読み、木を巡回し、ノードを追加・削除するプログラムを開発することができる。

イベント駆動アプローチは、対象外の要素を捨て、特定の要素だけに絞って処理するときにより便利である。もし、イベント駆動アプローチのプログラムに問題がでたとき、木構造アプローチに書き換えることが必要になる。しかし、図 15 の上部中央に示すように、木構造アプローチは、イベント駆動アプローチと全く異なるため、イベント駆動アプローチで記述した部分を全て書き換えなければならない。

3.2 XML バインディング

JAXB²⁶⁾ や XMLBeans²⁷⁾に代表される XML バインディングは、SAX や DOM に関する知識なしで XML 文書にアクセスするための機能を提供する。Java XML バインディングの場合、図 18 に示すように、スキーマ

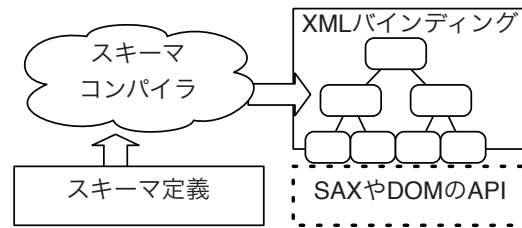


図 18 XML バインディングのためのスキーマコンパイラ

マコンパイラがスキーマ定義を読みこみ、Java クラスを生成する。この生成された Java クラスは、XML 文書中の要素と属性の各々に対応づけられる。したがって、SAX や DOM を使うことなく、オブジェクト指向プログラミングスタイルで、XML 文書内のデータにアクセスできる。

ところが、スキーマコンパイラによって生成されたコードは、アクセッサメソッドとそれに付随するメソッドが提供されるだけで、中身はブラックボックスになっている。もし、生成されるクラスの中に何らかのメソッド（例えば、グラフ表現の巡回など）を追加したい場合は、生成後のコードを手作業で修正するしかない。このプログラミングスタイルは、保守のときに問題となる。

3.3 アスペクト指向プログラミングの検討

アスペクト指向プログラミング²⁸⁾は、過去 20 年における技術進歩の一つである。アスペクト指向プログラミングの目的は、モジュール化を進めるために、関心事を分離するために、アスペクトを個別に定義するところにある。全てのアスペクトを定義した後、アスペクトウィーバーが全てのアスペクトのコード片を回収し、まとめたコードを生成する。この技術の進歩を、XML 文書処理のような構造化データ処理に反映すべきである。

本プロジェクトでは、Ruby 構文を使ったアスキー形式のデータ表現 RibON² を設計し、それを使ったデータ処理プログラムを開発するための新しい手法について検討を進めた。RibON は、木構造を表現することに特化しているわけではなく、一般的なデータ表

²RibON stands for “Ruby instructions become Object Notation.”

現が可能なように設計された言語である。RibON の重要な特徴は、プログラミング言語の構文の一部に基づいており、その記述が実行可能なところにある。

XML で記述されたデータ表現は、静的なデータにすぎないが、RibON で記述されたデータ表現は、プログラムと同様に、動的で実行可能である。実行可能な表現は、RibON による一連のメソッド呼び出しであり、その結果、アスペクト指向プログラミングを適用してデータ処理のモジュール化を進めることができる。RibON とアスペクト指向プログラミングの組み合わせは、構造化データ表現の可読性と簡潔性を向上させることができ、かつ、保守性に富んだプログラムの開発が可能になると信じている。

4 おわりに

本稿では、オブジェクト直列化のための6つの技法を概説し、定性的観点と定量的観点からの比較を行った。オブジェクト直列化のための各技法は、開発現場の必要性から現場に適した技法として出てきたものであり、全ての点で最善となる技法はないというのは明らかである。さらに、構造化データを処理するときに、アスペクト指向プログラミングが適用可能であることについて言及した。

定量的な比較から、バイナリ形式で直列化したときのサイズの方が、XML や JSON を使ったアスキー形式を使った直列化よりも小さくなることが分かった。もし、複数のプログラミング言語サポートが必要となるならば、プログラミング言語間の差異を吸収するために、スキーマ定義を記述し、そこからスキーマコンパイラがコードを生成することが必要となる。もし動的言語との相性を重視するならば、Apache Avro が最適な方法であろう。

著者は、複数のプラットフォームで互換性が高まるように、また、人間が読んで理解できるデータ表現として、できる限り簡潔にアスキー形式で表現することが望ましいと信じている。それは、計算機資源の大容量化・低価格化を背景に、オブジェクト直列化後のデータが少しぐらい大きくなっても構わないという判断からである。今後、オブジェクト直列化に関する調査研究を続けつつ、独自の直列化技法を作り上げることを予定している。その結果は、また別の論文にて発表したい。

参考文献

- 1) D. Hunter, J. Rafter, et al., Beginning XML, 4th edition, Wiley (2007).
- 2) A. B. Chaudhri, A. Rashid, and R. Zicari ed., XML Data Management, Addison Wesley, 2003.
- 3) Ronald Bourret, XML and Databases, <http://www.rpbouret.com/xml/XMLAndDatabases.htm> (accessed at Dec. 26, 2012).
- 4) Greg Badros, JavaML: A Markup Language for Java Source Code, 9th International World Wide Web Conference, <http://www9.org/w9cdrom/index.html> (2000).
- 5) JSON, <http://www.json.org/> (accessed at Dec. 26, 2012).
- 6) The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627, <http://www.ietf.org/rfc/rfc4627.txt> (accessed at Dec. 26, 2012).
- 7) Java SE 7 Serialization-related APIs and Developer Guides, <http://download.oracle.com/javase/7/docs/technotes/guides/serialization/> (accessed at Dec. 26, 2012).
- 8) Java Object Serialization Specification: Contents, <http://download.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html> (accessed at Dec. 26, 2012).
- 9) Richard Snodgrass, The Interface Description Language: Definition and Use, Computer Science Press (1989).
- 10) OMG, Common Object Request Broker Architecture (CORBA), OMG Released Versions Of CORBA, <http://www.omg.org/spec/CORBA/3.1/> (accessed at Dec. 26, 2012).
- 11) XStream - About XStream, <http://xstream.codehaus.org/> (accessed at Dec. 26, 2012).

- 12) protobuf - Protocol Buffers,
<http://code.google.com/p/protobuf/>
(accessed at Dec. 26, 2012).
- 13) Welcome to Apache Avro,
<http://avro.apache.org/>
(accessed at Dec. 26, 2012).
- 14) The MessagePack Project,
<http://msgpack.org/> (accessed at Dec. 26, 2012).
- 15) JSON Serialization Usage,
<http://flexjson.sourceforge.net/>
(accessed at Dec. 26, 2012).
- 16) google-gson - A Java library to convert JSON
to Java objects and vice-versa,
<http://code.google.com/p/google-gson/>
(accessed at Dec. 26, 2012).
- 17) Jackson JSON Processor - Home,
<http://jackson.codehaus.org/>
(accessed at Dec. 26, 2012).
- 18) Maven - Json-lib::Welcome,
<http://json-lib.sourceforge.net/>
(accessed at Dec. 26, 2012).
- 19) json-smart - A small and very fast
json parser/generator for java,
<http://code.google.com/p/json-smart/>
(accessed at Dec. 26, 2012).
- 20) jsonmarshaller - Fast, Lightweight,
Easy to Use and Type Safe
JSON marshaling library for Java,
<http://code.google.com/p/jsonmarshaller/>
(accessed at Dec. 26, 2012).
- 21) JSRs: Java Specification Requests,
JSR 173: Streaming API for XML,
<http://www.jcp.org/en/jsr/summary?id=173>
(accessed at Dec. 26, 2012).
- 22) Document Object Model (DOM) Specifications,
<http://www.w3.org/DOM/DOMTR>
(accessed at Dec. 26, 2012).
- 23) Apache Commons, <http://commons.apache.org/>
(accessed at Dec. 26, 2012).
- 24) Maven - Welcome to EZMorph,
<http://ezmorph.sourceforge.net/>
(accessed at Dec. 26, 2012).
- 25) SAX, <http://www.saxproject.org/>
(accessed at Dec. 26, 2012).
- 26) jaxb: JAXB Reference Implementation,
<https://jaxb.dev.java.net/>
(accessed at Dec. 26, 2012).
- 27) Welcome to XMLBeans,
<http://xmlbeans.apache.org/>
(accessed at Dec. 26, 2012).
- 28) Gregor Kiczales, John Lamping, et al., Aspect-
Oriented Programming, ECOOP '97 – Object-
Oriented Programming: 11th European Confer-
ence, LNCS 1241, pp.220–242, Springer, 1997.



前田和昭

1985 年慶応義塾大学理工学
部管理工学科卒業。1990 年慶
応義塾大学大学院博士課程満
期退学。1990 年中部大学経営
情報学部助手。1992 年同学部
講師。2003 年同学部助教授。
2010 年同学部教授 現在に至
る。プログラミング言語処理
系, XML, Web コンピューテ
ィング, オブジェクト指向プロ
グラミング, オープンソース
に興味を持つ。ACM, IEEE
Computer Society, 情報処理
学会, 電子情報通信学会, 情報
システム学会 各会員。