Ruby による XML 処理

大場 光一郎（伊藤忠テクノソリューションズ株式会社）
大場 寧子（株式会社 万葉）
須藤 功平（株式会社 クリアコード）

概要 XML 処理を Ruby を使って行うときの第一の課題は，XML を正しく処理し目的を達成することであるが，もう一つの大きな課題は，Ruby の特性を活かし，オブジェクト指向的に美しい設計を行い，プログラムのインターフェイスを直感的で使いやすくすることである．本稿では，ライブラリ RSS Parser の開発を通じてこれら 2 つの課題にどのように取り組んだかについての経験を通して，広く Ruby における XML 処理の設計・実装について考慮すべき事柄について論じる．ことに，現在国内では開発者が最も多く，実際の開発に頻繁に使われている Java との対比を通し，Ruby の特性を活かした設計についての知見を述べる．

1. はじめに

1.1 Ruby の利用の広がり

近年，プログラミング言語 Ruby の国内外での利用が拡大している． Ruby はコンパイルをせずに実行できるスクリプト言語であり，すべてのデータがオブジェクトとして表現されるオブジェクト指向の言語である．Ruby は柔軟性に富み，簡潔で直感的なコードを書きやすいと開発者の支持を集めている[1]． 2005 年に David Heinemeier Hansson 氏によって Web アプリケーションフレームワーク Ruby on Rails が Ruby によって開発され，大きな注目を集めたことで Ruby 利用の普及拡大の大きなきっかけとなった． Ruby の柔軟性やコードの書きやすさをもたらす特徴としては、単一継承ながらモジュールを利用した多重継承と同様の実装も行えること、強力なイントロスペクション[1]を備えていること，オペレータもメソッドとして実現されておりオーバーライドできること，クラスの作成やメソッドの追加などが動的に行われるため実行時にプログラム自体を変更できること，基本的な記法が簡素であることがある[2].

こうした特徴のある Ruby により XML を処理する際には，例えば、コンパイルを必要とし静的型付けの言語である Java における処理とは別のアプローチが適する場合がある．本稿では，須藤による RSS Parser ライブラリの開発を通じて考究された，XML 処理のための Ruby ならではの，あるいは Ruby に適したアプローチについて述べる．

[1] イントロスペクションとは，実行時にオブジェクトの詳細な情報を調査・参照でき，時にそれを通じてオブジェクトを変更したり影響を与えたりできる機能をいう．

1.2 フィード情報の処理

近年，ブログや Web サイトの情報を集約して XML で情報を配信するフィード技術が浸透している．フィード技術は，ニュースサイトやブログの更新情報を配信するのみならず，iTunes などで使われる PodCast といったマルチメディア情報の配信をするのにも応用されている．

フィード情報は，特定の語彙からなる XML で記述されるため，コンピュータで処理するのに適している．フィードの利用により，システム間で情報を連携させることが容易になる．

フィードの利用における重要な課題は、配信に使われる XML 応用言語の書式が統一されていないことである．代表的な言語は RSS であるが，バージョンの差異によって書式が異なる．RSS 0.9 は，W3C で策定された RDF 書式を利用する．RSS 0.91 は RDF を使わず，独自の XML 構文で記述される．RSS 1.0 は，RSS 0.9 と同様に RDF を利用するが，名前空間を利用して拡張可能になっている．RSS 2.0 は，RSS 0.91 互換性を維持した拡張である．さらに，近年では似通った目的で Atom もよく使われている．Atom の書式は，RSS とはまったく異なっている．以上ようにのフィードを配信するための XML 応用言語は複数の種類が存在し，複数のサービスのフィード情報を利用するには，いずれの XML 応用言語にも対応する複雑な処理が必要である．

こうした問題を踏まえ，フィード情報を Ruby から簡単に利用できるようにすることを目指し，RSS Parser を開発した．RSS Parser は 2004 年 4 月から Ruby に同梱され，配布・メンテナンスされている．本稿では，まず次節において RSS Parser で適用した XML パーサである REXML の Ruby 的な特徴について延べ，これに基づく

RSS Parser の設計について考察する.

2. REXML による XML 解析

フィード情報の解析には XML の解析が必要となる. RSS Parser は, 特定の XML 処理ライブラリに依存しないように構成しているが, デフォルトでは REXML[3]を利用する[2]. REXML は, XML 標準に概ね準拠し[3], XPath をサポートしたライブラリであり, DOM (Document Object Model)風のツリー解析やシンプルなストリーム解析, より SAX (Simple API for XML)に似せた SAX2 ストリーム解析, および XML プル解析を備えている. REXML は, 2003 年 8 月にリリースされた Ruby 1.8.0 から標準添付されるようになり, XML 処理の基本ライブラリとして広く利用されている[4].

本章では, REXML の解説を通して, Ruby による XML 解析が, Java のような静的型づけの言語による XML 解析とはどのように異なるアプローチとなるか示し, 第 3 章 RSS Parser 開発の基本的な方向性を示す.

2.1 DOM の解析

XML のパース方式としてよく知られている DOM についてまず, 述べる. REXML を用いて XML から DOM ツリーを得るには, まずライブラリ'rexml/document'をロードし, 次に XML データを引数として REXML::Document.new を呼ぶ. なお, new を呼ぶのは Ruby でインスタンスを作る際の共通の作法である. コード例は次のようになる.

```
require 'rexml/document'
xml = <<EOX
<text>
  <p>Hello, World.</p>
</text>
EOX
doc = REXML::Document.new xml
```

文字列ではなく, XML ファイル file.xml を解析する場合に必要なコードは次の２行のみでよい.

```
require 'rexml/document'
doc = REXML::Document.new File.new('file.xml')
```

REXML::Document は DOM におけるドキュメントに相当するクラスである. このコードの特徴は, 「パーサがデータをパースする」ではなく, 「DOM ツリーをデータを基に作る」という表現にになっていることである. このようなインターフェイス設計では, コードにおける必須の「登場人物」が少なくなり, コードがシンプルになる効果がある. シンプルなコードを追求することは Ruby の言語的, 文化的な特徴のひとつである[5]. 以上のコードと比較として, 次に, Java による同等の XML 解析の実行コードを示す.

```
t org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.parsers.*;
import java.io.*;

class FileRead {
    public FileRead() {
        DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
        DocumentBuilder builder = null;
        try {
            builder =
factory.newDocumentBuilder();
        } catch (ParserConfigurationException e) {
            e.printStackTrace();
        }
        File file = new File("file.xml");
        try {
            Document doc = builder.parse(file);
        } catch (SAXException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    static public void main(String[] args) {
        FileRead xml = new FileRead();
    }
}
```

これほどのコード量の差が生じるのは, Java ではパーサ, パーサのファクトリ, ドキュメントをライブラリのユーザに意識させるインターフェイス設計であること, 検査例外をキャッチしなければならない, コードを実行

[2] このほか xmlscan と xmlpaser を利用できる
[3] OASIS による W3C の XML 適合性試験スイートの中の non-validating の試験すべてが通る
[4] REXML を利用しているアプリケーションについては REXML のサイト[3]でも言及されている.
[5] まつもとゆきひろ氏は「コードの世界」[4]で Ruby の設計原則の第一に「簡潔さ」を挙げている.

するためにクラス（この例では FileRead）が必要であるといった言語上の特性に依るものである．

REXML の DOM の構造は W3C の DOM と同様であり，内部の要素やテキストにアクセスする方法も Java の場合と似ている．なお，getAttributes ではなく attributes といった具合に，メソッド名に get を使わない短いメソッド名が付けられているが，これは Ruby の一般的なスタイルである．

要素を表す REXML::Element は Enumerable を include しており，それ自体が子要素の集合としても振る舞うので，さらに簡略に記述することもできる．例えば，変数 doc にドキュメント（REXML::Element の派生クラス）オブジェクトが入っているとき，次の２つはどちらもルート要素を返す．

```
doc.children.first
```

```
doc.first
```

また，独自の便利な走査メソッドも用意されている．例えば以下のコードは，ドキュメント doc 内の要素を再帰的にたどって，要素名を順に出力する．

```
doc. each_recursive do |element|
  puts element.name
end
```

このように，REXML を使って DOM を得る処理は，基本的には Java における処理に似ているが，より簡潔に，Ruby のスタイルに沿って記述できる． ただし，REXML には RELAX NG による検証機能があるが，XML Schema による検証機能がない．また，エラーメッセージが不親切であるなど，Java に比べて使い辛い点もある．

2.2 ストリーム解析

REXML では DOM 解析だけでなく，SAX 的なストリーム解析も行える．ストリーム解析は，DOM 解析と比べ，軽量で省メモリとなるため，RSS Parser の開発においては，REXML のストリーム解析の利用を選択した．

ストリーム解析を行うには，リスナーオブジェクトを作り，REXML::Document の parse_stream メソッドに渡す．次に挙げるコード例は， Twitter.com 上で「Ruby」を検索した結果の Atom データをストリーム解析し，Twitter で最近「Ruby」とつぶやいたユーザを取得する．

```
require 'rexml/document'
require 'rexml/streamlistener'
require 'open-uri'

class TwitterListener
  include REXML::StreamListener
  attr_reader :users
  def initialize
    @users = []
  end
  def tag_start(name, attrs)
    @name_started = true if name == "name"
  end
  def text(text)
    if @name_started
      @users << text
      @name_started = false
    end
  end
end

list = TwitterListener.new
src=open("http://search.twitter.com/search.atom?q=ruby
")
REXML::Document.parse_stream(src, list)
```

上記コードでは REXML::StreamListener を include しているが，これは Java におけるインターフェイスのようにリスナーの型を指定するためのものではなく，テンプレート実装を include した上で必要なメソッドだけオーバーライドするためのものである．Ruby は動的型付けの言語なので，リスナーオブジェクトが期待された振る舞い（メソッド）を備えていれば，そのクラスは何でも構わない．

3. RSS Parser の開発

3.1 基本構造

RSS Parser では，REXML のストリーム解析のような SAX 系のパーサを内部で利用し，フィードオブジェクトを構築し，ユーザに返す．また，解析時には，ニーズに応じて検証も行う．そのため，プログラムは主に以下のパートに分かれる．

- XML 解析時に呼び出されるイベントハンドラ
- イベントハンドラから作られるフィードオブジェクト
- イベントハンドラから実行される検証

このほか，ユーザがシンプルに解析を実行するために，これらをラップするパーサクラス，エラークラスも必要である．

3.2 XML パーサの抽象化

RSS Parser では, REXML のみに依存するのではなく, XML 解析処理を抽象化し, 図 1 に示すように, BaseParser, BaseListener という抽象クラスを設けた．これらりより, ほかの XML 処理ライブラリも簡単に扱える．

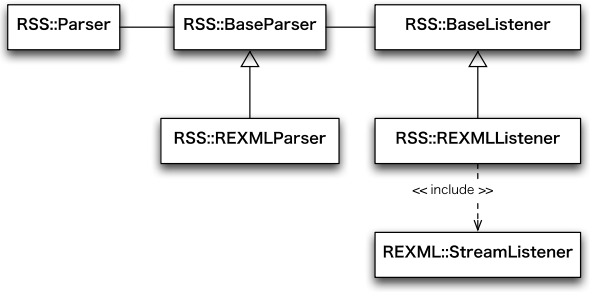

図 1.RSS Parser の UML クラス図

3.3 異なるフィード規格への対応

RSS Parser では, 広く使われている RSS 0.91, RSS 0.92, RSS 1.0，RSS 2.0，Atom に対応することとした．解析結果となるフィードオブジェクトを，規格ごとに別のものとするか，共通にするかは設計上の重大な問題である．

共通にするアプローチでは，ユーザは 1 種類のデータ構造を覚えればよく，将来の規格の追加の際にも学習コストが低いというメリットがある. しかし，すべてのフィード規格が表現できるすべての情報をフィードオブジェクトに盛り込むためには，各フィード規格単体での直感的な使用感を犠牲にしなければならない．

例えば，RSS と Atom では，タイトルという概念に対して，データ構造が異なる．RSS のタイトル情報は単なる文字列だが，Atom では HTML や XHTML の場合もある．そのため，共通のフィードオブジェクトを設計する際には，タイトル情報の形式をユーザに提供する必要が生じる．オブジェクト指向的な設計をすれば，タイトル情報の形式をタイトルオブジェクト自身に持たせることになるので，タイトルの形式と値はそれぞれ次のように参照させるインターフェイスが有力となる．

```
tem.title.type   # タイトル情報の形式
item.title.value   # タイトル情報の値
```

つまり，RSS しか扱わないユーザにとっては item.title で済んだものを, item.title.value と書いて参照しなければならなくなることになる．

規格別に異なるフィードオブジェクトを用意するアプローチでは，このような問題は，発生しない．ユーザがある規格を知っていれば，直感的に利用することができる．しかし，複数の規格のフィードを，違いを意識せずに同じように利用したいニーズを満足できない．

そこで，RSS Parser では，規格別に異なるフィードオブジェクトを提供するが，フィードオブジェクト間でデータ構造を変換できるようにすることにした．変換ができれば，ユーザは複数のフィード規格の解析結果を，専ら自分の利用しやすい規格の構造に変換し，これを統一的に扱うことが可能である．

変換後のフィード規格が対応する情報を持たない規格であった場合，情報が欠落するという問題があるが，実用上は問題にならない．変換による情報の欠落が問題となる場合には, 本来のフィードのまま利用すれることで, 変換に依る情報の欠落の問題を回避できる

3.4 規格ごとのパース処理の分離

フィードの規格については，異なる規格に対応するだけでなく，ある規格が別の規格を取り込むといった展開にも耐えられることが求められる. 具体的には, RSS 1.0, RSS 2.0，Atom が DublinCore のモジュールをサポートしたことがある．RSS Parser の設計にあたっても，このような変化に対応できる柔軟な構造が望ましい．

> あるフィード規格が別の規格を取り込むケースにも対応

複数の規格の解析を行うための設計としては，Java などの静的な言語では，規格ごとに異なるパーサクラスを用意して入力データに応じて差し替えるのが一般的である．しかしこれでは，ある規格の中に別の規格を取り込んで使いたいといったニーズの変化が生じた際に，既存のパーサクラスのコードを変更せずに対応するのは困難である．

そこで，RSS Parser ではパーサの実装について，Ruby ならではの戦略を採った．すなわち次である．

- 規格ごとにコードファイルを分離する
- 各コードが実行される時点で, 共通の XML 解析時のイベントハンドラに，それぞれの規格専用の解析処理を動的に追加する

当然ながら，共通に使える解析処理はあらかじめイベントハンドラに持たせておく. 3.3 節で触れた設計方針により，解析結果を格納するフィードオブジェクトクラスを規格ごとに作ることにした．そして，このフィードオブジェクトクラスの書かれたファイルをロードする際に, イベントハンドラに解析処理を追加するコードも実行することとした．これらにより，規格ごとのコードが分離されるとともに，ある規格が別の規格をとりこむといっ

た変化に対しても，新たなフィードオブジェクトクラスに対応するファイルの追加だけで対応できる．

フィードオブジェクトクラスの分離と，ファイルロード時の処理イメージを図 2 に示す．

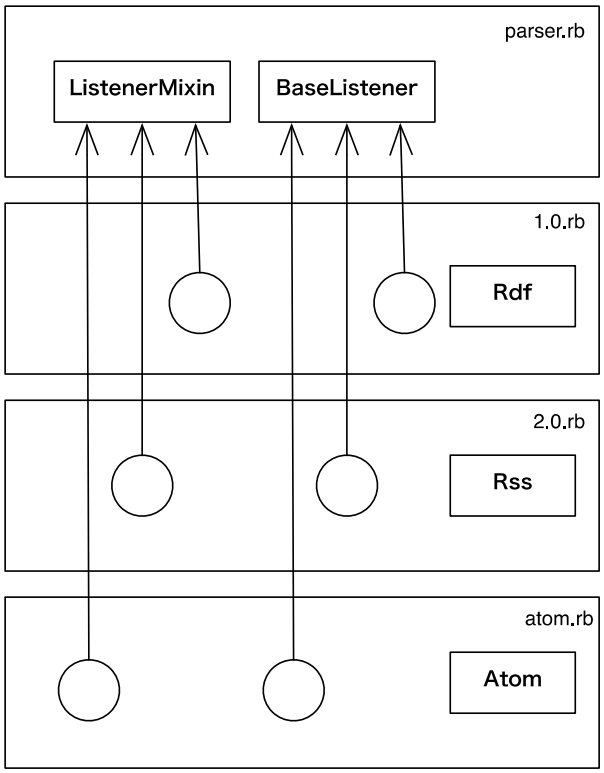

図 2.リスナークラスの構造

図２において，BaseListener クラスと ListnerMixin モジュールは，いずれも最終的に XML 解析時のイベントハンドラに組み込まれる．REXML のストリーム解析を行う場合のイベントハンドラは図 3 のように構成される．

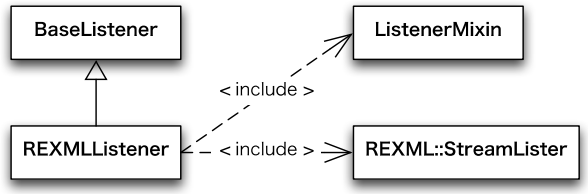

図 3.リスナーのクラス図

このように，先にクラスやモジュールの基本部分を作成し，後から柔軟にメソッドを追加するという戦略は，Ruby の動的な処理によって可能となる．Ruby では，クラスやモジュールの定義も動的に実行されるコードであり[9]，その中で様々な動的な処理を行える[6]．実際の規格ごとのソースファイル内の具体的な構造は次のようになっており，結果オブジェクトクラスの作成，BaseListener へのメソッド追加，ListenerMixin へのメソッド追加[7]を行う．

```
module RSS
  class RDF < Element
    # 結果オブジェクトのルート要素
    # さらに子要素クラスを定義
    …
  end

  # 特定の要素を処理するためのメソッドを BaseListener に追加
  RSS10::ELEMENTS.each do |name|
    BaseListener.install_get_text_element(URI, name, name)

  module ListenerMixin
    # メソッドを追加
    …
  end
end
```

BaseListener クラスのクラスメソッドである install_get_text_element は最終的に次に示すクラスメソッド def_get_text_element を呼び, Ruby の define_method メソッドを用いて動的に BaseListener にインスタンスメソッド start_xxx （xxx には要素名が入る）を追加していく．

```
def def_get_text_element(uri, element_name, file, line)
  register_uri(uri, element_name)
  method_name = "start_#{element_name}"
  unless private_method_defined?(method_name)
    define_method(method_name) do |name, prefix, attrs, ns|
      uri = _ns(ns, prefix)
      if self.class.uri_registered?(uri, element_name)
        start_get_text_element(name, prefix, ns, uri)
      else
        start_else_element(name, prefix, attrs, ns)
      end
    end
```

[6] なお，クラスやモジュールの定義の外側にも自由にコードを書くことができる．

[7] ListenerMixin モジュールへのメソッドの追加には，クラスやモジュールの定義をどこからでも書き換えられる Ruby のオープンクラスという特性を利用している．オープンクラスについては「Metaprogramming Ruby」[7]p4 に言及がある．

```
    private(method_name)
  end
end
```

なお，規格ごとに同じクラスやモジュールに独自の処理メソッドを追加するアプローチでは，異なる規格の間で同名の別メソッドを追加する可能性がある．RSS Parser では，すでにメソッドが追加済ならば新たに追加せず，メソッドを複数の規格で共有する方法を採った．規格ごとの処理の違いは内部で分岐される．

3.5 バリデーション機能

Java のように，コンパイル時に型チェックを行う言語では，解析結果を入れるクラスはあらかじめ完全に定義されるため，入力 XML が期待する構造になっていないことはチェックされやすい．例えば，対応するクラスのない未知の要素が現れた場合，それをどこかに格納するよりも，例外を出すことが一般的である[8]．これに対して Ruby は，メソッドの引数や返り値が特定の型でなければコンパイルが通らないといった静的型づけの言語ではなく，ダックタイピングができる言語である．ダックタイピングでは，オブジェクトが最初にコードを記述した時点で想定した型かどうかではなく，オブジェクトがその場面で必要な振る舞いを備えているかが重要となる．むしろ，型をチェックしてしまうと，将来のダックタイピングによる拡張性を阻害する危険がある．このような背景から，Ruby では 100%想定通りであるかにこだわるよりも，主要な部分が動作することが重視される．このような文化的な特徴を一因として，Ruby では一般に Java に比べて入力データの検証の優先度が低くなりがちである．

しかし，RSS Parser の開発にあたっては，バリデーションを重視し[9]，デフォルトでバリデーション機能を有効とした．バリデーションを意識することは XML 処理で最も重要なことであると考えている．

バリデーションは，適切なエラーを適切な個所に通知することにつながる．適切なエラー通知は，ソフトウェアの開発，デバッグ，利用において非常に重要である．通知されるエラーが，単に「不正な入力です」では何をもって不正な入力であるのか問題が不明確なため，対応することが難しい．「1 番目の項目のタイトルがありません」ならば対応することができる．また，できるだけ問題が発生した段階でエラーが通知されることが望ましい．処理が進んだ後での通知では，エラーが発生したときからの状態が変化していることがあるため，エラーの源となった処理がわかり辛くなってしまう．

バリデーションはフィードを利用する前に行うため，フィードを利用する処理の途中であるはずの情報がないという問題に遭遇することを回避できる．また，RSS Parser は，利用者がバリデーションにおける問題を特定するために必要な情報をできる限りエラーメッセージに盛り込む方針で開発した．フィードに問題がある場合でも解決しやすい．

> Ruby での XML 処理では
> 意識して
> バリデーションを行う

バリデーション機能の実装にも，Ruby 的な方法を用いた．具体的には，要素タグの開始時に検証を実行する Proc オブジェクトを作成してスタックに積み，終了時にスタックに積んだ Proc オブジェクトを実行している．Proc オブジェクトは，クロージャをオブジェクト化したものである．

3.6 パーサのインターフェイス

RSS Parser はフィードの処理を行うことに特化するので，ごく少ないコードで解析を行い，必要な値を参照できることが望ましい．そこで，フィードを解析するためには RSS::Parser クラスの parse メソッドに入力データを渡すというインターフェイスとした．次に， Ruby 公式サイトのフィードを解析するコード例を示す．

```
require "rss"
require "open-uri"

feed_uri = "http://www.ruby-lang.org/ja/feeds/news.rss"
feed = open(feed_uri) do |input|
  RSS::Parser.parse(input)
end
```

解析した結果は，前述のとおり，規格ごとに異なるデータ構造となり，ルート要素にあたるオブジェクトが返される．そこからフィードの構造に沿ったメソッド呼び出しで内部のデータにアクセスできる．例えば，RSS 1.0

[8] 「Effective Java 第二版」[8]ではメソッドにおける強力なパラメータの正当性検査が推奨されている．Ruby でも同種のことを行えるが，Ruby のプログラミング指南本ではこういった手法の重要性はあまり強調されない．

[9] RSS Parser は Ruby 用のフィードパーサー実装で唯一バリデーション機能を実装している

に含まれる各項目のタイトルを出力する場合は次のように記述する．

```
rss10 = RSS::Parser.parse(input)
rss10.items.each do |item|
  puts item.title
end
```

3.7 フィード間の変換インターフェイス

あるフィード規格での解析結果を，異なるフィード規格に変換するために，to_rss メソッドと to_atom メソッドを用意した．次に示すのは，RSS 1.0，RSS 2.0，Atom のフィードを順に解析し，すべてのフィードを RSS 2.0 と同じ構造を持つオブジェクトにしてからタイトル情報を表示するコードである．

```
sources = [rss10_url, rss20_url, atom_url]
rss20_feeds = sources.collect do |source|
  feed = open(source) do |input|
    RSS::Parser.parse(input)
  end
  feed.to_rss("2.0")
end
rss20_feeds.each do |rss20|
  rss20.channel.items.each do |item|
    puts item.title
  end
end
```

4. 実践の結果得られた知見

4.1 Ruby による XML 処理の利点

RSS Parser の開発を通じて，Ruby による XML 処理の利点を見いだすことができた．まず，クラスやモジュールのメソッドを動的に変更できるため，同じパーサ（イベントハンドラ）クラスを異なるフィード規格で共有しつつ，規格ごとの専用処理を順次加えるアプローチをとることができた．これには，複数の規格に対応したり，ある規格が別の規格を取り組むといった変化のある XML 応用言語の解析処理においては，コードの量を少なく保てる[10]，新しいファイルを追加するだけで規格の追加や，拡張に対応できるといった効果がある．フィードの規格は今後も変更や拡張が行われることが予想されるので，変化を前提とした設計の重要性は高い．

次に，検証処理で役立ったように，Proc オブジェクトを使うと，処理を動的に作成した上で保存しておき，別のタイミングで実行することができる．XML を SAX 的なやり方で解析する場合には，要素の開始時に処理を作成し，終了時に実行することができ，便利である．

RSS Parser では対象となるフィードの構造にあわせてあらかじめクラスをモデリングしており，解析時は，適切なクラスを見つけてオブジェクトを作り，構造化していた．Ruby の動的な側面を利用すれば，スキーマファイルから対応するデータクラスを動的に作成すること[11]や，XML インスタンスそのものから XML の内容に沿ったインターフェイスを備えたオブジェクトを作成することも可能である．[12]

4.2 Ruby らしい設計

Ruby を使った XML 処理では，Ruby の言語的特性に適した設計が有効と考えられる．モジュールの include によってクラスに機能を追加でき，多重継承に近いことが行えるという特性は，設計に大きなインパクトを与える[13]．Java にはモジュールによる構造化の機能がないので，コードの共通部分を抽出するためには専ら継承が利用される．しかし，継承は一系統しかできないため，すでに継承をある側面の抽象化に利用していると，別の側面で実装を抽出するのが困難になり，結果として一部の実装が複数箇所に重複して記述されやすいという問題があった．これに対して，Ruby ではモジュールの include で，いくつでも異なる側面ごとに実装を切り出して構造化することができる．RSS Parser でも，XML 解析クラスに include されるモジュールを規格ごとに拡張しているのをはじめ，モジュールを多用した．

また，Ruby には簡潔に記述できることを重視する文化がある．RSS Parser においても，フィードの規格ごとに別の解析結果クラスを用意して，フィードの構造に対して直感的で簡潔なインターフェイスを実現することを重視した．

Ruby では処理ロジック自体もブロックや Proc オブジ

[10] 例えば，規格ごとに異なるイベントハンドラクラスを作り，共通項を抽象クラスに抽出するというアプローチに比べてコード量が少なくなる．

[11] 実際に，SOAP4R の wsdl2ruby では WSDL をもとに stub クラスを生成できる．ただし解析のたびにデータクラスを作成しなおすのではもちろん効率が悪い．

[12] James Gray 氏は 2010 年の RubyConf での発表「unblocked」の中で，XML インスタンスを解析しながら動的にクラスを作ってバインディングをする例を挙げている[12]．

[13] Yugui の文献 6)によれば，このようなモジュールの利用のしかたは Ruby では Mix-in と呼ばれ，制限された多重継承として働くとある．

ェクトとして簡単に流通させることができる.そのため,RSS Parser の検証で利用したように,動的にロジックを生成して後で使うという設計を行うことができる.

Ruby のクラスは,実行時に動的に作成,変更される.これを念頭に置くと,Java とは異なる設計が可能である.RSS Parser では,複数の異なるフィード規格が同じイベントハンドラクラスを共通で使うというモデルを採用した.これは,規格ごとに分けて記述したコードを順次実行することで共通クラスに機能を追加できる Ruby だからこそとり得る戦略といえる.もし Java で実装するのであれば,クラスに動的に機能を追加できないので,あらかじめ規格ごとのパーサクラスを用意して,入力データの規格を判定していずれかのパーサに渡すという方法が一般的である.例えば Java の RSS パーサライブラリの Rome[11]は,この様な戦略をとっており,RSS 0.9,RSS 1.0,Atom 1.0 といった書式毎に異なったパーサクラスがある.そして,このような設計の場合,ある規格がある規格を取り込むような変更にあたり,既存のコードの修正を回避することは難しくなる.

> Ruby 的な設計では
> Mix-in,ブロック,
> 動的な性質を活用≈

Ruby で XML 処理を行うためには,以上のような言語的特性の違いを意識し,Ruby に適した設計を行うことが重要である.

4.3 Ruby を使った XML 処理の課題

Ruby を使った XML 処理には,注意を要する事項もある.Ruby 1.8 は,言語としては文字エンコーディングに関知していないので,ライブラリ作者や利用者がエンコーディングを意識し,必要に応じて自分で変換を行う必要がある.例えば REXML は,XML 内の文字列を UTF8 に変換して処理するが,REXML を使うソースコードが UTF8 で書かれていないと,理由が示されないまま文字列の比較がうまくいかない問題を生ずる.なお,Ruby 1.9 ではエンコーディングを String オブジェクトが持ち,ソースコードも自らのエンコーディングを記述するように改善されたので,意図しないエンコーディングのミスマッチを解決しやすくなっている.しかし,Java のようにプログラム中の String が常に同一のエンコードであるわけではないということには注意を払う必要がある.

また,RSS Parser でも用いた include や,メソッドの追加にも注意が必要である.というのは,Ruby はメソッド名の衝突に関して特に警告を出さないからである.これは,Ruby にはそれがプログラマの意図していないものか,意図したものか(既存のものを上書きする意図であるか)判別できないためである.RSS Parser の例のように,異なるドメインの機能が共通オブジェクトをシェアし,拡張するようなモデルでは,特に注意が必要と思われる.

4.4 最近の動向

最後に,最近の動向について補足する.最近人気を集めている Ruby の XML ライブラリに,Nokogiri[14]がある.

REXML には,エラー処理が不親切であったり,名前空間の対応が完全でない問題がある.Nokogiri は,これらを解決するより強力な XML ライブラリとして 2008 年に登場した.技術的な特徴として,C 言語で実装された libxml2 という XML 処理ライブラリを使っており,XML だけでなく HTML の解析機能も備えている.

Nokogiri は REXML よりも高速に動作する.筆者の MacBook Core2Duo 2.4GHz メモリ 4GB の環境では,18k byte の XML を解析するのに REXML が 3.85 秒かかったのに対して,Nokogiri は 0.11 秒しかかからなかった.これは,計算量の必要な XML のパース処理を高速な libxml2 で実行しているためである.Nokogiri は,以前であれば処理時間が問題になったような大量の XML 処理をも Ruby から行える道を拓いた.

5.まとめ

本稿ではまず,複数の規格が乱立するフィード情報を近年利用が進む Ruby から利用することの重要性と,そのためのライブラリの必要性を述べた.続いて,フィード情報の解析に使える代表的な XML 処理ライブラリである REXML について説明し,Ruby による XML 解析が Java に比べて簡潔に記述できることを示した.次に,Ruby でフィード情報を利用するための RSS Parser ライブラリの基本構造,特定の XML パーサに依存させないための抽象化,異なる規格と解析結果オブジェクトのクラスについての設計方針,規格ごとの実装の分離,バリデーション,パース及びフィードオブジェクトの変換インターフェイスについて述べた.さらに,RSS Parser の開発を通じて,XML 処理における Ruby ならではの利点,設計上の留意点,注意点について考察した.

---

[14] Nokogiri[10]は,CRuby のコミッタであり Ruby on Rails のコミッタでもある Aaron Patterson 氏によって主に開発されている

Ruby の柔軟で動的な言語的特性を活用すると，複数の異なる規格や，規格の将来の変化に対応しやすい，モジューラビリティの高いプログラム開発が可能になる．動的にメソッドを追加するといったメタプログラミングのコードは概して難しく，読みにくいという欠点もあるが，うまく取り入れることで，少ない労力で大きな仕事をするプログラムを書いたり，メンテナンスをしやすくすることができる．

参考文献

1) Bruce A. Tate: Java から Ruby，オライリー・ジャパン，pp.24-26 (2007)
2) 大場寧子，大場光一郎，久保優子：Ruby on Rails 逆引きクイックリファレンス，毎日コミュニケーションズ(2008)
3) REXML, http://www.germane-software.com/software/rexml/ .
4) まつもとゆきひろ：まつもとゆきひろ コードの世界 スーパープログラマになる 14 の思考法，日経 BP 社，p11 (2009)
5) David Flanagan，まつもとゆきひろ:プログラミング言語 Ruby，オライリー・ジャパン(2009)
6) Yugui: 初めての Ruby，オライリー・ジャパン, p159,(2008)
7) Paolo Perrotta：Metaprogramming Ruby, Pragmatic Bookshelf
8) ジョシュア・ブロック: Effective Java 第二版，ピアソン・エデュケーション，p.175 (2008)
9) 青木峰郎: Ruby ソースコード完全解説，インプレス社，p69 (2002)
10) Nokogiri: http://nokogiri.org
11) Java フィードライブラリの Rome, https://rome.dev.java.net/ .
12) James Gray: Unblocked, http://www.slideshare.net/JamesEdwardGrayII/unblocked .

大場 光一郎（正会員）
E-mail: koichiro.ohba@ctc-g.co.jp
1995 年独立系ソフトウェアハウス入社．2001 年 伊藤忠テクノソリューションズ（株）入社，Java を使った SI 開発，Ruby 開発案件の標準化推進を経て，現在 Ruby や JRuby を活用してクラウドサービス開発に従事．

大場 寧子（非会員）
E-mail: y.ohba@everyleaf.com
東京大学文学部日本語日本文学（国文学）専修課程終了．1996 年(株)ジャストシステム入社，自然言語処理を利用したシステム開発などに従事．2001 よりベンチャー企業にて Java による Web アプリケーション開発を統括．2007 年(株)万葉を設立．代表取締役社長．Ruby on Rails を使った開発や iPhone アプリケーション開発に携わる．

須藤 功平（非会員）
E-mail: kou@clear-code.com
岩手大学大学院工学研究科情報システム工学専攻博士前期課程修了．株式会社クリアコードにて Ruby などフリーソフトウェアを活用したシステム開発に従事．2008 年 8 月より代表取締役に就任．就任後も就任前と同様に活発に開発を行う．

投稿受付：2010 年 9 月 19 日
採録決定：2010 年 11 月 16 日
編集担当：守安 隆（東芝ソリューション）