DEWS2005 5A-o2

非XMLデータに対するXPath検索のための ラッパーのインターフェイスの設計

渡谷　賢治† 　　田島　敬史†

† 北陸先端科学技術大学院大学 石川県辰口町旭台 1-1
E-mail: †{watatani,tajima}@jaist.ac.jp

あらまし　本論文では各種の非 XML データに対して XPath による検索を行うシステムを提案する．このシステムは，ある共通インターフェイスとなる操作群を各データ形式上で実現するラッパーと，ラッパーが提供する操作群を使って XPath を評価する共通モジュールから構成される．ラッパーが提供する操作群が低レベル過ぎると実行効率が悪くなり，逆に高レベル過ぎると各データ形式毎にラッパーを用意するコストが高くなる．本論文では，どのような操作群が XPath の効率的評価に必要か考察し，また，低レベルなインターフェイスの例である SAX インターフェイスを用いる場合や，より高レベルなインターフェイスを用いる場合との性能比較を行う．

キーワード　XML, 問合せ処理, 最適化

Interface Design of Wrappers for XPath Queries on Non-XML Data

Kenji WATATANI† and Keishi TAJIMA†

† Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatunokuti, Isikawa, Japan
E-mail: †{watatani,tajima}@jaist.ac.jp

Abstract　In this paper, we develop a system for querying non-XML data by XPath. The system consists of wrappers that provide a interface to each data format consisting of common primitive operations and a common module that evaluate XPath by using operations provided by wrappers. If the wrappers provide very low-level operations, it is hard to evaluate XPath efficiently, and if they are very high-level, it is costly to develop wrappers for many data formats. The purpose of this paper is to design an operation set that balances those two factors, and compare it with more low-level interfaces, such as SAX, and more high-level interfaces.

Key words　XML, Query Processing, Optimization

1. はじめに

現在，汎用の標準データ形式として XML が広く普及しつつある．XML 形式を採用する利点としては，様々なデータに対して XML 用のツールを用いて統一的な処理を行えるという点がある．例えば，自分のアドレス帳，スケジュール表，メイルボックス等を XML 形式でで保存しておけば，これらのデータに対して XML 用の検索言語である XPath [3] で一括して検索を行うなどの処理が可能となる．しかし，各アプリケーション専用の様々なデータ形式や画像等の様々なデータ種別毎の標準データ形式も依然として多く使用されており，データ量や性能上の問題から，今後も全てのデータが XML 形式になるとは考えにくい [4]．そこで，本研究では，XML データと非 XML データの両方を統一的に処理するためのシステム，特に XPath で統一的に検索するためのシステムの開発を行う．

非 XML データに対して XPath を用いた問い合わせを実現する手法としては, [4] でも述べられているように，以下のような手法が考えられる．

（1） データを XML に変換してファイルに保存し，検索時にそのファイルを用いる．

（2） データを XML 形式に変換し，それを DB に格納し，この DB に対して問い合わせを発行する．

（3） データを検索時にメモリ上に XML に変換して展開し，そのデータに対して検索処理を行う．

（4） 与えられた検索式の評価に必要な操作を，対象となるデータ形式上の操作に変換し，元データ上で検索処理を行う．

上記 (1)，(2) では，XML に変換済みのデータと元データとの二つが存在するため，データに変更があった場合の整合性の問題と，データ量の増大の問題がある．上記 (3) ではメモリ上に XML 形式でデータを展開すると，メモリ領域を大量に消費す

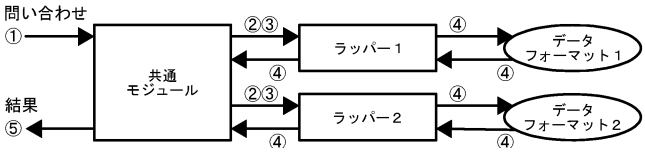

図 1　共通モジュールとラッパーの構成

る問題がある．そこで，本研究では上記 (4) の方式を採用する．この方式では，データの重複は存在せず，またデータを XML としてメモリ上に展開しないため，メモリを大量に消費するという問題もない．

本研究で提案するシステムは，より詳しくは，データ形式に依存しない処理を行う共通モジュールと，データ形式依存の処理を行うラッパーからなる．ラッパーは，各データ形式毎に用意され，各データ形式のデータに対して共通モジュールが統一的にアクセスできるように，ある共通の操作群からなるインターフェイスを提供する．共通モジュールとラッパーが協調して動作する手順はより具体的には，以下のようになる．

〔1〕 共通モジュールは，ユーザから XPath 式を受取り，これを解析する．

〔2〕 問い合わせの対象となる各データ毎に，そのデータ形式に応じたラッパーが起動される．

〔3〕 共通モジュールは，与えられた問い合わせ式の解析結果に基づいて，各ラッパーに対して，各データに対する操作命令を出す．

〔4〕 各ラッパーは操作命令を各データ形式上の操作に変換し，各データのファイルにアクセスして必要な部分読み出して共通モジュールに返す．

〔5〕 共通モジュールは読み出したデータの内容に基づいて，さらにラッパーに対する操作命令を出す．以下，最終結果が得られるまで，これを繰り返す．

また，以上の協調動作の様子を図示したものを図 1 に示す．

共通モジュールとラッパーは上記のような手順で協調動作を行うが，その役割の分担の仕方には，両者の間の共通インターフェイスとなる操作命令のレベルに応じて様々な形が考えられる．一般に，ラッパーが提供するインターフェイスが高レベルであるほど，共通モジュールの作成は容易になるが，ラッパーの作成コストは高くなり，逆に，ラッパーが提供するインターフェイスが低レベルであるほど，共通モジュールの作成コストは高くなるが，ラッパーの作成コストは低くなる．共通モジュールは一つ作成すればよいのに対し，ラッパーは各データ形式毎に作成する必要があることを考えれば，ラッパーの作成コストが低い方が望ましく，よって，共通インターフェイスの操作群は低レベルである方が望ましい．

一方，問合わせの実行効率を考えた場合，ラッパーの提供する共通インターフェイスのレベルが高いほど，各データ形式依存の最適化をラッパー内に組み込みやすくなり，実行効率を上げることが容易になるが，逆に，共通インターフェイスのレベルが低いほど，各データ形式依存の最適化をラッパー内に組み込みにくくなり，実行効率を上げることが困難になることが考えられる．別の言い方をすると，共通モジュールとラッパーの役割分担において，ラッパーの分担が大きいほど，様々なデータ形式依存の効率化を組み込める余地が大きくなり，逆に，共通モジュールの分担が大きいほど，そのような効率化を組み込める余地が小さくなる．これは，共通モジュール側で実現できる処理は，必ず同じことを，生データに直接触れるラッパー側でも実現可能だが，逆に，生データに直接触れるラッパー側で実現できる処理は，ある共通インターフェイスを通してのみデータに触れる共通モジュール側で実現できるとは限らないためである．

そこで，本研究では，ラッパーの作成コスト削減と問合わせ処理の効率化という，二つの相反する要素をバランス良く両立できるような，ラッパーのインターフェイスの設計を目的とする．より具体的には，本論文では，後述する 4 つの方式について考案，検討し，その一部については，実装を行って比較実験を行った．まだ実装を行っていない，残る方式についても，今後，実装と比較実験を進める予定である．

2. 関 連 研 究

バイナリデータの XML ビューを実現する枠組みに関する関連研究としては，既に [1] や [4] などがある．しかし，これらの研究は，データ形式の記述から自動的に XML ビューを生成する機構に関する研究であり，本研究での，各データ形式毎にラッパーを人手で作成するというアプローチとは異なる．また，これらの既存の研究では，汎用の XML ビューを提供することを目的としているが，本研究では，目的を XPath による統一的な検索に絞り，その目的に特化した，効率的な処理を行えるラッパーを作成する場合に適切なインターフェイスの設計を目的としており，目的が異なる．例えば, [4] では，XPath の効率的評価についても述べられているが，XML ビューのインターフェイスとしては汎用性を重視して DOM を採用している．

3. XML と XPath

XML 形式のデータはノードにラベルのついた木構造で表現することができる．XPath はこの木構造に対して問い合わせをする木パターン言語であり，例えば，図 2 のような木構造を持つ XML データに対して，root の下の person の下に現れる name の値を取得するには/root/person/name という XPath 式を使用し，また person 以下全体を取得するなら/root/person となる．また，“[” と “]” で囲まれた条件式 (predicate と呼ぶ) を使用することができ，name が “Alan” である person の age を取得する場合は，/root/person[name=”Alan”]/age となる．

XPath 式は “/” で分割することができ，“/” で分割された各部分式をロケーションステップと呼ぶが，本研究ではさらに predicate を表す “[” と “]” でも分割したものを「ステップ」と呼ぶこととする．例えば，/root/person[name=”Alan”]/age という問合わせを本研究で言うステップに分割した場合，/root, person，name=”Alan”，age の四つの部分式に分割される．

一方，“/” では分割せずに，“[” と “]” でのみ分割した物を，本研究では simple path 式と呼ぶことにする．例えば，

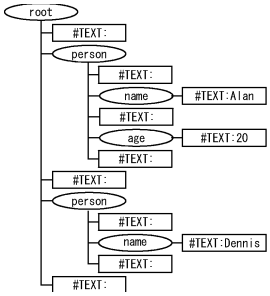

図 2　XML の木構造

/root/person[name="Alan"]/age という問合わせを本研究で言う simple path 式に分割した場合，/root/person，name="Alan"，age の三つの部分式に分割される．

4. 本論文で提案・検討する方式

本論文では，具体的には，以下の 4 つの方式について，提案，検討を行う．

方式 1: 問合わせ依存の処理を全てラッパー側で行う

共通モジュールは具体的な XPath の処理は行わなず，単に，対象データに応じたラッパーを起動して，ラッパーに XPath 式を渡す．ラッパーは XPath 式を処理してデータを操作し，その結果を共通モジュールに返す．この方式ではラッパーに各データ形式固有の処理以外の処理（XPath 式の評価に関する，各データ形式で共通な処理）も含むため，ラッパーのコーディング量が増えるが，もし，ラッパーの作成コストが問題にならない程度であれば，問合わせ処理の効率化の余地は最も大きい．

方式 2: インターフェイスとして simpla path 式を用いる

共通モジュールは適切なラッパーを起動するとともに，与えられた XPath 式を解析し，より単純な XPath のサブクラスである simple path 式の集合に分割する．当初の XPath 式を分割して得られた各 simpl path 式は，それぞれ，それに対応する，ラッパーが提供する操作命令に変換されてラッパーに渡される．ラッパーは渡された命令に従い操作を行い，操作が完了すると共通モジュールに通知し，次の命令を待つ．この場合，一つの操作命令が，XML 木上での単純なナビゲーションの複数ステップをまとめたものに相当する．

方式 3: インターフェイスとしてステップを用いる

方式 3 は方式 2 とほぼ同様だが，インターフェイスが提供する操作の単位が simple path 単位ではなくステップ単位となる．この場合，一つの操作命令が，XML 木上での単純なナビゲーションの一ステップに相当する．

方式 4: 問合わせ依存の処理を全て共通モジュール側で行う

共通モジュールは適切なラッパーを起動するとともに，XPath を評価する．ラッパーは，与えられた XPath 式依存の処理は一切行わず，常に同じ動作を行う．よって，全ての問合わせに対応できるためには，ラッパーは，常にデータの先頭から最後までの全体を処理して，データ全体の仮想的な XML ビューを共通モジュールに提供することになる．

インターフェイスとなる操作群のレベルで見た場合，方式 1 が最も高レベルであり，以下，方式 2，方式 3，方式 4 の順で低レベルとなる．そのため，共通モジュールの作製は方式 1 に近づくほど簡単となるが，ラッパーの作成は方式 4 に近づくほど簡単となる．一方，検索効率は，方式 4 が最も悪く，方式 1 に近づくほど，効率化の余地がある．特に，方式 4 は，常にデータ全体を処理するため，データが大きくなると，問合わせ効率が非常に悪くなることになる．

5. 方式 2 の詳細

この章では，方式 2 における共通インターフェイスと，共通モジュールおよびラッパーそれぞれの動作について，さらに詳しく説明する．

5.1 方式 2 におけるインターフェイス

方式 2 で用いるラッパーが提供するインターフェイスは対象データを引数として受け取って起動され，その後 simple path を受け取り，その処理結果を返すインターフェイスとなる．

ラッパー内部でのデータアクセスには，効率性の面から，ランダムアクセスが求められる．また，メモリを大量に消費しないことも重要である．そこで，ここでは，SAX のように先頭からデータを見ていくが，現在いる要素からある関係にある要素に対応する位置まで移動してスキップしたり，戻ったりすることができるインターフェイスを用いる．使用することができる具体的な命令群について，以下に示す．

- **regist** x y fp: simple path 式 x と，そのコンテキストノード y と，そのファイルポインタ fp のセットをラッパーに登録する．登録されたらラッパーからクエリディスクリプタ qd が返される．クエリディスクリプタは，以降の操作命令で引数として渡すために使用される．
- **go** (**next**)? qd: qd で識別される simple path 式とコンテキストノードの組み合わせについて，式の解となる要素を探索し，そのような要素が存在する場合，ファイルポインタを最初に現れるそのような要素の開始地点に移動する．オプション **next** が指定された場合は，問合わせ qd について最後に返した解の次の解を探し，その開始位置にファイルポインタを移動する．どちらの場合も，そのような解がない場合は，ファイルポインタは移動せず，返り値として false が返される．
- **test** (**next**)? qd: **go** とほぼ同様だが，そのような要素が存在するかどうかを調べ true または false を返すだけで，ファイルポインタは移動しない．
- **get** (**next**)? qd: **go** とほぼ同様だが，そのような要素が存在する場合は，その要素以下の部分木の XML ビューを生成し，共通モジュールに返す．

5.2 方式 2 の動作概要

方式 2 の共通モジュールは，XPath 式を simple path のリストに分解し，これらをラッパーが提供する操作のうちの対応するものに変換してラッパーに渡す．この XPath 式から，操作命令のリストへの変換アルゴリズムを表 1 に示す．

例として，ユーザから XPath 式として /a[b/c]/d が提示されたとする．これを表 1 に示すアルゴリズムで命令リストに変換する例を以下に挙げる，

- リスト X は { /a, [b/c], /d } となる

- 命令リストに **go /** を追加する
- /a は “/” から始まり，最終ステップでは無いため，**go a** を命令リストに追加
- [b/c] は”[”から始まり，最終ステップでは無いため，**test b/c** を命令リストに追加
- /d は ”/”から始まり，最終ステップであるため，命令リストには **get d** を追加

最終的なリストの要素は {**go /**, **go a**, **test b/c**, **get d**} となる．

ラッパーは，基本的にはこれらの操作命令を順にラッパーに渡し，ラッパーから返される結果に基づき次の動作を決定していく．この処理のアルゴリズムを表 2 に示す．共通モジュールはラッパーから返された値が真，位置情報，または文字列の場合，命令リスト中の次の命令を登録する regist 命令をまずラッパーに渡し，返されたクエリディスクリプタを次の命令の相対パスに置き換えてラッパーに渡す．ラッパーから返された値が偽の場合はリストの一つ前の命令に戻り，移動命令の場合は go next 命令，テスト命令の場合は test next 命令，取得命令の場合は get next 命令を発する．このときのクエリディスクリプタは偽と判別されたクエリディスクリプタの一つ前のクエリディスクリプタを使用する．リストの先頭の命令に対しラッパーが偽を返すと，動作を終える．以下に，具体的な動作例を挙げる（regist 命令の部分は省く）．

/x/y[@index=3]/text() という XPath 式の場合はリストには {**go /**, **go x/y**, **test @index=3**, **get text()** } が格納される．**test @index=3** により ラッパーは非 XML データの /x/y ノードの attribute である index の値を取得し，値 3 と比較され，同値なら共通モジュールに true を返し，違うなら false を返す．共通モジュール側では true の場合次の命令 **get text()** を発行し，false なら **go next x/y** 命令を発行する． false だった場合，この処理は **go next** x/y 命令の結果が false となるまで繰り返される．

/a/b//c という XPath 式の場合，リストには {**go /**, **get a/b//c**} という命令が格納され先頭から処理される．**get /a/b//c** は最初に出された後，その結果が true なら false が返るまで **get next a/b//c** の発行を繰り返す.

6. 方式 3 の詳細

この章では，方式 2 について説明した前章と同様，方式 3 における共通インターフェイスと，共通モジュールおよびラッパーそれぞれの動作について，詳しく説明する．

6.1 方式 3 におけるインターフェイス

方式 3 で用いるラッパーが提供するインターフェイスは方式 2 とよく似た物になるが，各命令は，XPath の一ステップに相当する物になっている．使用することができる命令群について，以下に示す.

- **go** (**AnyChild**|**Next**)? x: 現在ファイルポインタがいるノードの子ノードで要素名 x を持つ物が存在するかどうか調べる．**AnyChild** が付くと子孫ノードから，**Next** が付くと現在位置のノードの，現在位置以降の兄弟ノードから要素 x を探す．そのような要素が存在する場合，ファイルポインタは最初に現れるそのような要素の開始地点に移動する．

```
XPath 式を predicate で分割しリスト X にそれぞれを加える;
空のリスト "命令リスト" を作成;
"go /"を命令リストに追加;
for(i=0; i < X の要素数; i++) {
  x = X の要素 [i];
  if(x が"/"から始まる){
    x を "/" "相対ステップ" に分割;
    if(x が X の最終要素である){
      命令 "get 相対ステップ" を命令リストに追加;
    } else {
      命令 "go 相対ステップ" を命令リストに追加;
    }
  } else (x が"["から始まる){
    x を "[" "predicate 文字列" "]" に分割;
    if(x が X の最終要素である){
      命令 "get predicate 文字列" を命令リストに追加;
    } else {
      命令 "test predicate 文字列" を命令リストに追加;
    }
  }
}
```

表 1　方式 2 の XPath を命令へ変換するアルゴリズム

```
tr(命令){
  "名" = "命令" からロケーションステップ名を取得;
  qd = "regist 名"をラッパーに渡す;
  result ="命令"の名を qd に置き換えラッパーに渡す;
  if(result が 真 もしくは偽以外の文字列){
    tr(次の命令)
    各"命令"種別の next 命令を作成
  } else if (result が 偽){
    処理を終了し，元のプログラムに戻る;
  }
  result = "命令"をラッパーに渡す;
  while(真){
    if(result が 真 もしくは偽以外の文字列){
    tr(次の命令)
      result = "命令"をラッパーに渡す;
    } else if (result が 偽){
      処理を終了し，元のプログラムに戻る;
    }
  }
}
```

表 2　方式 2 の命令リストをラッパーに渡す時のアルゴリズム

- **test** (**AnyChild**|**Next**)? x: **go** とほぼ同様だが，そのような要素が存在するかどうかを調べ true または false を返すだけで，ファイルポインタは移動しない.
- **get** (**AnyChild**|**Next**)? x: **go** とほぼ同様だが，そのような要素が存在する場合は，その要素以下の部分木の XML ビューを生成し，共通モジュールに返す.

方式 3 では共通モジュール側で XPath 式をステップ単位に分解し，それらを方式 3 のラッパーが提供するインターフェイスの命令に変換する必要があるが，その変換アルゴリズムを表 3

```
XPath 式をステップで分割しリスト X にそれぞれを加える;
真偽値 flag を 偽にセット
空のリスト "命令リスト" を作成;
for(i=0; i < X の要素数; i++) {
  x = X の要素 [i];
  if(x が"/"から始まる){
    x を "/" "名" に分割;
    if(ステップ名が空文字列 "" である){
      x は"//"のため，flaq を真にする;
    } else {
      if(flag が真){
        if(x が X の最終要素){
          命令 "get AnyChild 名" を命令リストに追加;
        } else {
          命令 "go AnyChild 名" を命令リストに追加;
        }
        flaq に偽をセット;
      } else {
        if(x が X の最終要素){
          命令 "get 名" を命令リストに追加;
        } else {
          命令 "go 名" を命令リストに追加;
        }
      }
    }
  } else (x が"["から始まる){
    x を "[" "predicate" "]" に分割;
    if("predicate"に比較演算子を含む){
      "predicate" を "名前" "比較演算子" "値" に分割;
      命令 "get 名前" を命令リストに追加;
    } else {
      命令 "test predicate" を命令リストに追加;
    }
    if(x が X の最終要素である){
      命令 "get ."を命令リストに追加;
    }
  }
}
```

表 3　方式 3 の XPath を命令リストへ変換するアルゴリズム

に示す.

方式 3 における共通モジュールの動作例として，ユーザから XPath 式として/a[b]/c が提示されたとする．これを表 3 に示すアルゴリズムでステップ単位に対応する命令リストに変換すると，

- リスト X は { /a, [b], /c } となる．真偽値 flag は偽である.
- /a は “/” から始まる文字列であり，flag は偽であるので **go a** を追加
- [b] は”[”から始まる文字列であり，flag は偽であるので **test b** を追加
- /c は “/” から始まる文字列であり，flag は偽で最終要素のため **get c** を追加

となり，最終的なリストの要素は {**go a**, **test b**, **get c**} となる.

共通モジュールはラッパーに対しこれらの命令を送り，その結果に基づいて次の動作を決定する．共通モジュール側はラッパーから返された値が真の場合，リストの次の命令をラッパーに渡す．偽の場合はリストの一つ前の命令に戻り，移動命令の場合は go Next 命令，テスト命令の場合は test Next 命令，取得命令の場合は get Next 命令を発する．リストの先頭の命令に対しラッパーが偽を返すと，動作を終える．動作例を次に挙げる.

/x/y[@index=3]/text() という XPath 式の場合はリストには **go x**，**go y**，**get @index**，**get text()** が格納される．**get @index** によりラッパーは非 XML データの/x/y ノードの attribute である index の値を取得する．取得した値は共通モジュールで 3 と比較され，同値なら次の **text()** を発行し，違うなら **go Next y** 命令を発効して次の/x/y ノードの index 値の取得をラッパーに指示する．この処理は **go Next y** 命令の結果が false となるまで繰り返される.

/a/b//c という XPath 式の場合，リストには **go x**，**go y**，**getAnyChild c** という命令が格納され先頭から処理される．**getAnyChild c** は，/a/b の子孫である最初の c ノードに出会うと一度 true を返すが，共通モジュールはここでさらに **getAnyChild c** 命令を出し，これをラッパーから false が返るまで繰り返す.

7.　方式 4 の詳細

本研究では，方式 4 における，常にデータ全体の XML ビューを提供するインターフェイスとして，標準的な XML アクセスのための API の一つである，SAX を用いた．すなわち，ラッパーは与えられたデータを XML に仮想的に変換し，この XML データを表す SAX イベントの列を生成する．共通モジュールは，このイベント列を受け取って，与えられた XPath 式を評価し，解となる部分を抜き出してユーザに返す．このように，方式 4 では，大きなデータからほんの一部を抜き出す問合わせが与えられた場合においても，ラッパーがデータの XML ビュー中の全ての要素に関するイベント列を生成していくので，問合わせ効率が非常に悪くなる.

8.　システムの実装

実装として方式 2，方式 3，方式 4 の共通モジュール及びラッパーを Java 5.0 で実装した．ラッパーの例としては，メールボックス形式の一つである MBX 形式を対象とした物を実装した．方式 1 についても，現在，実装を行っている.

8.1　各方式でサポートする XPath の範囲

現時点では各方式の実装においてサポートしている XPath の範囲は以下の通りである.

方式 1 でサポートする XPath の範囲

- 全ての XPath 命令.

方式 2 でサポートする XPath の範囲

- **axis**: よく使用される child，attribute，parent，descendant-or-self のみ.
- **NodeTest**: 要素名によるノードテスト及び text() のみ.
- **predicate**: [80] や [@x=”z”]，[x/y/z/@a=”b”] といった，数字のみの物，値を調べるのみの物，相対ロケーションパス，

表 4　MBX 構造の文法

MBX	:=	(MAIL)*
MAIL	:=	HEADERS, BODY
HEADERS	:=	(HEADER)+
HEADER	:=	HEADERNAME ":" HEADERVALUE
HEADERNAME	:=	#STRING
HEADERVALUE	:=	#STRING
BODY	:=	#STRING

相対ロケーションパスと値を調べる物を複数合成した物をサポート．入れ子になっている predicate はサポートしない．

方式 3 でサポートする XPath の範囲

- **axis**: よく使用される child，attribute，parent，descendant-or-self のみ．
- **NodeTest**: 要素名によるノードテスト及び text() のみ．
- **predicate**: [80] や [@x=”y”]，[x] といった，数字のみの物，値を調べるのみの物，一ステップの物をサポート．入れ子なっている predicate はサポートしない．

方式 4 でサポートする XPath の範囲

- 全ての XPath 命令

8.2　MBX 形式

MBX 形式のファイルには一つ以上の電子メールが保存され，一つの電子メールは「From -」で始まり，一回の空行と「From -」で始まる文字列までである．メールの構造を BNF 式で表したものを表 4 に示す．ここで，#STRING とは制御文字以外の ISO-8859-1 の文字セットである．

8.3　方式 2 用ラッパーの実装

ラッパーは，共通モジュールからの命令により MBX データ上でファイルポインタを移動し，成功するとファイルポインタもしくは真偽値，あるいは取得データを返す．内部処理のアルゴリズムを表 5 に示す．

MBX データを木構造で表した図が図 3 である．ここで各ノードの左上の数字はノード番号であり，右上の数字はファイルポインタの位置である．また，ノード番号 4 のテキストノードの値は”fail”で，ノード番号 8 のテキストノードの値は”success”であるとする．このとき，共通モジュールに XPath 式 /mbx/mail[headers/header=’success’]/body が与えられたときのラッパーの処理を以下に示す．

（1） ラッパーは ”regist / / 0”命令が渡されると命令リストに追加し，クエリディスクリプタ qd1 を返す．

（2） 次に ”go qd1”を受け取りると，qd1 に対応する移動先”/”を命令リストから取得，移動し位置情報 0 を返す．

（3） 次に ”regist mbx/mail / 0”命令を受けとると命令リストに追加し，qd2 を返す．

（4） 次に ”go qd2”を受け取ると，qd2 に対応する移動先”mbx/mail”を命令リストから取得し，移動し位置情報 0 を返す．

（5） 次に ”regist headers/header=’success’ mbx/mail 0” を受け取ると命令リストに追加し，qd3 を返す．

（6） 次に ”test qd3”を受け取ると，qd3 から対応するテスト

```
空のスタックである命令スタック を作成;
int counter = 0; //クエリディスクリプタ用のカウンタ
while(真){
  命令 = 共通モジュールからの命令を読み込む;
  真偽値 result = 偽;
  switch(命令){
  case 登録命令;
    result = regist(命令);
    break;
  case 移動系命令:
    result = go(命令);
    break;
  case テスト系命令:
    result = test(命令);
    break;
  case 取得系命令:
    result = get(命令);
    break;
  }
  if(result が真で，命令がテスト系命令である){
    共通モジュールに true を入力;
  } else {
    共通モジュールに false を入力;
  }
}
```

表 5　方式 2 ラッパー内部の疑似プログラム

”headers/header=’success’”を取得する．その後 headers/header の値を取得し，’success’ と同値であるかを比較する．今回はノード番号 4 の値が”fail”であり，その他に”headers/header”に対応する要素がないため，qd3 から位置情報を読み取り，一つ前の移動命令”go qd2”によって返された位置情報である 0 に移動し，命令リストから qd3 削除し共通モジュールに false を渡す．

（7） 次に ”next qd2”を受け取ると，ノード 2 より後ろにある”mbx/mail”に移動し，位置情報 100 を返す．

（8） 次に ”regist headers/header=’success’ mbx/mail 100” を受け取ると命令リストに追加し，qd3 を共通モジュールに渡す．

（9） 次に ”test qd3”を受け取ると，qd3 から対応するテスト ”headers/header=’success’”を取得する．その後 headers/header の値を取得し，’success’ と同値であるかを比較する．今回はノード番号 8 の値が”success”であり共通モジュールに true を渡す．

（10） 次に ”regist body mbx/mail 100”を受け取ると命令リストに追加し，qd4 を共通モジュールに渡す．

（11） 次に ”get qd4”を受け取ると，body 要素を取得し，共通モジュールにその文字列を渡す．

8.4　方式 3 用ラッパーの実装

ラッパーは共通モジュールから起動され，共通モジュールからの停止命令により終了する．MBX データ用のラッパーは，共通モジュールからの命令により MBX データ上でファイルポインタを移動し，成功すると true もしくは取得データを返し，失敗すると false を返す．追加の処理として内部ではファイルの先頭から現在読み込んだ位置までに go 命令で移動したノードの

```
真偽値 regist(命令){
  命令スタックに命令を追加して counter ++;;
  共通モジュールに QD ("qd"+counter) を渡す;
}

真偽値 go(命令){
  fp = 現在のファイルポインタ;
  index = 命令から QD を抜き出し，その番号を取得;
  path = 命令リスト [index] の 相対パス x を取得;
  真偽値 result = path に移動を行った成否;
  //result が真なら現在のファイルポインタの位置が移動
  if(result が真){
    共通モジュールに fp の値を渡す;
  } else {
    現在のファイルポインタを fp にセット;
  }
  return result;
}
```

表 6　方式 2 ラッパー内部の疑似プログラムのサブルーチン群 1

```
真偽値 test(命令){
  index = 命令から QD を抜き出し，その番号を取得;
  path = 命令リスト [index] の 相対パス x を取得;
  真偽値 result = false;
  if(path に比較演算子が入っている){
    result = path に移動し，値の比較結果;
  } else {
    result = path に移動できるかどうか;
  }
  return result;
}

真偽値 get(命令){
  index = 命令から QD を抜き出し，その番号を取得;
  path = 命令リスト [index] の 相対パス x を取得;
  真偽値 result = path の値の取得の成否;
  if(result が真){
    共通モジュールに値を渡す;
  }
  return result;
}
```

表 7　方式 2 ラッパー内部の疑似プログラムのサブルーチン群 2

XPath 式による階層表現とファイルポインタのオフセット，カウンタをセットで保持する。また，共通モジュールから渡された命令が取得命令の場合，その値を共通モジュールに渡す.

共通モジュールに XPath 式 /mbx/mail[2]/body/text() が与えられたときのラッパーの処理を以下に示す.

（1） ラッパーは go mbx を受け取るとラッパー内部のアルゴリズムに則り，命令が移動系命令のため go() に入る．ルートノード以下に要素 mbx があるため，result は真となるのでノード番号 1 の要素 mbx に移動し，スタックに ”/mbx 0 1” を追加する

（2） 次に go mail を受け取るとラッパー内部のアルゴリズムに則り，命令が移動系命令のため go() に入る．現在のノード以下に mail が存在する場合，その mail に移動し，スタックに ”/mbx/mail 0 1” を追加する

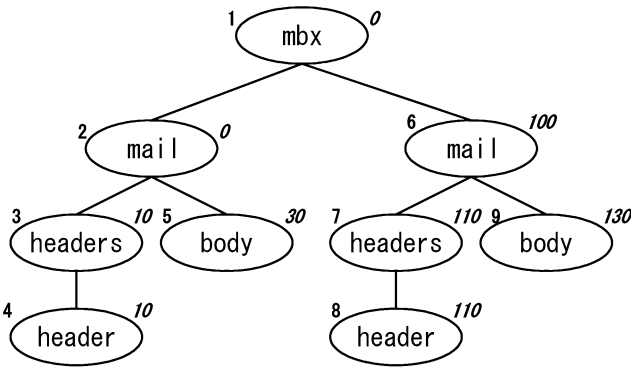

図 3　MBX データの木構造での表現

（3） 次に go Next mail を受け取ると 1 つ次の兄弟要素 mail に移動し，スタックに”/mbx/mail 100 2”を追加する”/mbx/mail 100 2”より，インデックス値が 2 であることが解るため，共通モジュール側では次の命令に移動する

（4） 次に go body を受け取ると 1 つめの body 要素に移動し，スタックに ”/mbx/mail/body 130 0” を追加する

（5） 最後に get text() を受け取るとノード番号 9 のテキスト値を取得して共通モジュールに渡す

8.5　方式 4 の共通モジュールの実装

方式 4 においては，ラッパーからは SAX イベント列が返されるので，共通モジュールは，その SAX イベントを用い XPath 式の評価を行い，結果をユーザに提示する．XPath 式を評価するシステムには XSQ [5] を用いた.

8.6　方式 4 用ラッパーの実装

ラッパーは共通モジュールからインスタンスとして作成され，MBX データファイルを受け取り，SAX イベントを発生させる．MBX データファイルは XML データでは無いため，先頭から順にファイル内を走査し，仮想的に MBX データファイルが XML データであるかのように SAX イベントを発生させる簡易 SAX パーザを作成した.

9.　実験及び評価

非 XML データとして電子メールボックス形式の一つである MBX 形式を用いた評価実験を行った．実験環境は，PC(Pentium M 1.4GHz，RAM 512MB，Windows XP SP2，J2SDK 5.0) 上である．実験には格納されているメール数がそれぞれ違う MBX ファイルを 5 つ用い，それぞれ格納されているメール数は 1000, 2000, 3000, 4000, 5000 である．それぞれのメールデータは自作のメールデータ生成プログラムを使用した．これにより 1 メールにつきヘッダ数 3，本文部 1，本文部を持ったメールデータが生成される．本文部の長さは 0 から 5000 単語までのランダムの数の単語が辞書データから生成される.

9.1　実 験 結 果

上記の条件で，XPath 式/mbx/mail[n]/body/text() を用いて各方式の実験を行った．ここで，n は 0 からそれぞれの格納デー

タ数までの，500 毎の値である．方式 2,3,4 において，各 MBX ファイルに対し上記の XPath 式を用いて問い合わせした実行時間を図 4 に示す．グラフの横軸が，問合わせ式中の n に対応する．また，方式 2,3,4 のメモリ使用量を表 5 に示す．また，比較の対象として，対象 MBX データを事前に，MBX から XML への変換を行うオープンソースのプログラムを用いて XML に変換し，これに対して Xalan [2] を用いて問合わせを行うという事前変換方式についても実験を行い，この場合の値も，方式 0 として，同じ図中に示してある．それぞれ 3 回データをとり，その平均を採用した．また，ここでのメモリ使用量とは，共通モジュールとラッパーのメモリ使用量を合計したものである．

9.2 考　　察

図 4 の各方式を比較すると方式 3 が，他方式に比べて高速な結果となり，メモリ使用量も他方式と比べて方式 3 が少なくなることがわかった．また，方式 4 は方式 0 よりも遅くなってしまった．これは，どちらのラッパーも MBX ファイルもしくは XML ファイルを 1 パスで処理することを考えると，方式 0 のラッパーがオープンソースのプログラムを使用していることに対し，方式 4 のラッパーが自作のプログラムを使用することで，速度を重視したチューニングができていない可能性がある．また，方式 3 が一番早くなる大きな理由としては，方式 0 や方式 4 は図 4 によると，総メール数が大きくなるほど処理時間は大きく，同じ方式で，同じ総メール数の場合の XPath 式の処理時間は変わらない．これは，方式 0 や方式 4 では，最初の方のメイルを取り出す問合わせの場合でも，必ずデータを一度最後まで読まなければならないためである．しかし，方式 3 では/mbx/mail[n +1] 以降のノードは処理する必要がないため，その時点で処理を打ち切ることができる．よって，n の値が低い場合は高速な処理を行うことができる．方式 2 も同様の理由により，方式 0 や 4 よりも速くなっている．方式 2，3 を比べると，ほぼ方式 2 は 3 の倍の時間がかかっていることになる．これは，方式 2 は 3 と違い，クエリの登録時間がかかるからである．現在では 1 命令につき 1 命令登録をしなければならないが，命令の再利用を図ることで高速化が図れると思われる．また，方式 0 の結果は MBX から XML への変換時間を含まない．この変換時間を足すと，非 XML データに対して XPath による問い合わせを行う場合，方式 0 を使用するよりも当システムを使用する方が処理時間及びメモリ使用量に関して有効であることがわかる．方式 2，3 のそれぞれのコード量は，全体としては方式 3 が方式 2 より少なくなった，また，共通モジュール部は方式 2 が，ラッパーについては方式 3 が少なくなった．これは，ラッパー部に関しては XPath の処理などを方式 3 はしなくて良いためであるが，その分，共通モジュール側での XPath 処理が多くなるため，方式 3 の共通モジュール部のコード量が大きくなるからである．本研究ではファイル形式毎にラッパー部を作成するため，ラッパー部の作成コスト削減には方式 3 が適していると言えるが，予想したほどはコード量に差が出なかった．

10. まとめと今後の課題

本研究では非 XML データに対する XPath 検索のためのラッパーの提供するインターフェイスのデザインについて考察し，有効と考えられる方式の二つ，すなわち前記の，方式 2 と方式 3 について実装，評価実験を行った．また，より低レベルのインターフェイスを用いる方式 4 や，事前に XML に変換する方式との比較も行った．今回は電子メールボックス形式の一つである MBX について評価を行ったが，今後，他形式の非 XML データについても実験を行う予定である．また，方式 2 や 3 を改良した方式についても検討を行う予定である．

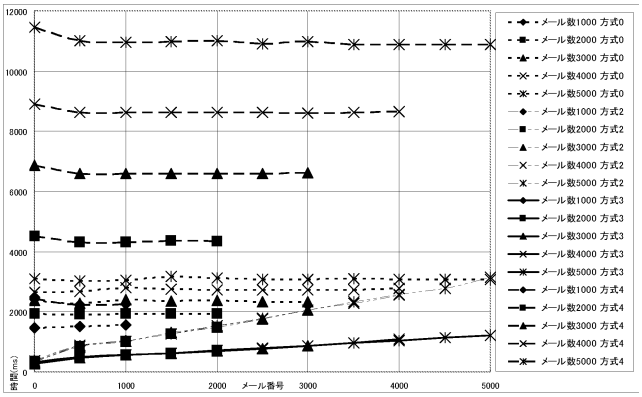

図 4　方式 0,2,3,4 の処理時間

メール総数	方式 0 (K byte)	方式 2 (K byte)	方式 3 (K byte)	方式 4 (K byte)
1000	21442	2106	1617	5004
2000	44250	2809	1839	5004
3000	64213	3466	1998	5005
4000	87598	4210	2282	5005
5000	106109	4863	2439	5006

図 5　方式 0,2,3,4 のメモリ使用量

	共通モジュール	ラッパー	合計
方式 2	217	1189	1406
方式 3	257	1085	1342

図 6　方式 2,3 のコード量

文　　献

[1] eDIKT::BinX. http://www.edikt.org/binx.

[2] Xalan. http://xml.apache.org/xalan-j/index.html.

[3] J. Clark and S. DeRose, eds. *XML Path Language (XPath) Version 1.0 – W3C Recommendation*. http://www.w3.org/TR/xpath, Nov. 1999.

[4] 品川徳秀 and 北川博之. バイナリデータ上の XML ビュー機構と XPath 処理の提案. *DBSJ Letters*, 2(3):49–52, Dec. 2003.

[5] XSQ. http://www.cs.umd.edu/projects/xsq/