Webブラウザから利用できるSAXパーサ “Freddy” の実装と評価

横山　昌平†

† 産業技術総合研究所グリッド研究センター
〒 305–8568 茨城県つくば市梅園 1–1–1 中央第二
E-mail: †dews2007@yokoyama.ac

あらまし　本研究の目的は Web ブラウザ上で半構造データを処理する為の軽量システムの実現である．JavaScript から XML のような半構造データを扱う仕組みとしては JSON がある．JSON は文書全体を一旦メモリ上に読み込み木を構築する必要があり，大きなデータを扱う為には適していない．そのような用途において XML 処理では SAX を利用する．SAX は高速でかつ低消費メモリの特徴を持つが，ブラウザ上で JavaScript による SAX パーサを構築する議論はほとんどされていない．本論文では Web ブラウザ上で動作する SAX を提案し，その性能を評価する．
キーワード　SAX, JavaScript, XML

An implementation and evaluation of a SAX parser “Freddy” on Web browsers

Shohei YOKOYAMA†

† National Institute of Advanced Industrial Science and Technology, Grid Technology Research Center
Tsukuba Central2, Umezono, 1–1–1, Tsukuba, Ibaraki, 300-8568, Japan
E-mail: †dews2007@yokoyama.ac

Abstract　The purpose of this research is to materialize the lightweight system which handles semi-structured data on Web browsers. In this context, it has been proposed JSON. However, JSON is not suitable for handling large amount of semi-structured data, because it must build a tree structure of the whole document in memory. In this case, SAX is memory-efficient and fast way, but I can find almost no SAX Parses of JavaScript on Web browsers. In this paper, I propose and evaluate a new SAX Parser for JavaScript applications on Web browsers.
Key words　SAX, JavaScript, XML

1. はじめに

近年 Web ブラウザは，以前のように Web サーバが出力したデータを単に表示するだけでなく，高度なユーザインタラクションを有するアプリケーションのプラットフォームとして注目されてきている．

従来，Web ブラウザにおけるユーザインタラクションは CGI(Common Gateway Interface) やサーバサイドスクリプト等に基づいていた．ユーザの操作が CGI プログラムに伝えられると，ユーザの入力に従い動的な出力を返す．そしてその出力は新しい Web ページとしてブラウザ上に表示される．つまり，ユーザの操作をサーバに伝える度に，新しい Web ページのデータを取得しなければならなかった．ページ遷移には通信時間や通信量などのコストがかかる．またブラウザが表示する Web ページは HTML というデザインに特化したマークアップ言語で記述する必要があり，データのみの通信を行う事ができず非効率である．

そのような従来手法の欠点を解消する手法として Ajax(AsynchronousJavaScript and XML) [1] が注目されている．Ajax は Web ブラウザ上で非同期通信を行うプログラミングモデルの総称であり，JavaScript や XML，HTTP 等の技術に基づいている．Ajax を利用して実装すれば，Web ページはページの遷移無しに Web サーバとの通信を行う事が可能となる．この仕組みによって Web ページは HTML によるデザインと XML 等によるデータを分離する事ができ，高度なアプリケーションが様々に開発されている．例えば Google Map や Windows Live Local ではスクロール可能な地図・衛星画像を提供している．また Google は表計算ソフトやワードプロセッサソフト等を Web ブラウザを通じて提供している．このような Ajax によって引き起こされた Web 利用環境の新しいパラダイムは Web2.0 [2] と呼ばれ，非常に注目されている．

本論文はこの Web2.0 における Web ブラウザ・サーバ間の

データ通信に注目する．Web ブラウザ・サーバ間の通信に用いるデータ形式として，現在，ブラウザによるパースを必要としない軽量な半構造データ記述フォーマット JSON(JavaScript Object Notation) [3] が注目されている．JSON は JavaScript のオブジェクトリテラルに従い，オブジェクトや配列やそれらの入れ子構造を記述でき，XML 形式と互換性が高く，また XML 形式に比べよりオープンであり，単純さや相互運用性などでは勝っているとされている [4].

JSON によって記述された半構造データは JavaScript からそのまま読み込む事ができる．しかしながら，この手法では XML 処理における DOM のように読み込む半構造データ全体を Web ブラウザ上のメモリ領域に読み込む必要があるため，大規模なデータを扱うには適していない．XML 処理ではそのような用途として SAX を用いる．SAX は XML 文書を先頭から走査し，構成要素をイベントとしてユーザに通知するイベントドリブン形式の XML 処理手法であり，DOM と共に広く普及している．DOM はデータの再利用性の高さが利点とされており，SAX は実行速度および消費メモリが少ない事が利点とされ，用途によって使い分けられている．

一方 JavaScript による半構造データ処理では SAX に相当するシステムが著者の調べた限りでは存在しない．今後 Web2.0 と呼ばれる新しい Web の発展と共に，あるいは情報爆発と呼ばれる Web 上のデータ総量の爆発的な増加に伴い，Web ブラウザ上で扱う必要のあるデータ量は漸増すると考えられる．Web ブラウザを搭載するコンピュータはサーバ用の大規模計算機から，非常に限られた計算資源しか持たないシンクライアントまで多種多様である．そこで，搭載しているメモリの量等によらず，大規模なデータを扱う仕組みを Web ブラウザ上で実現することは非常に有用であると考える．

本研究では Web ブラウザ上で JavaScript から利用できる SAX パーサ Freddy を開発した．Freddy は SAX の特徴でもある高速かつ低消費メモリで動作し，さらに通信や Web ブラウザによる制約などを隠蔽し，言語に関わらず SAX を利用したことのあるプログラマにとって習得が容易な構造を持っている．本論文ではその実装手法および得られた知見を報告する．

本論文の構成は，続く 2 章では提案手法に関連する技術の説明と，提案手法の概要を述べる．3 章では提案手法について詳述し，4 章にて実験を行い提案手法の性能に関して考察する．最後に 5 章で本論文をまとめる．

2. 関連技術と提案手法の概要

本章では Web ブラウザから半構造データを扱う際の既存技術を紹介し，本研究に至る動機，背景を述べる．

2.1 Ajax

Ajax はブラウザ上で非同期通信を行う仕組みであり，ダイナミック HTML 技術と組み合わせる事により，ページ遷移を伴わずにバックグラウンドで Web サーバと通信を行い，表示されているコンテンツの一部を書き換える事ができる．XML 文書を出力する CGI 等と組み合わせて，Web サーバ・クライアント間の動的かつ非同期のデータ通信を実現できる．しかしながら読み込んだ HTML ファイルの存在するサーバとしか通信できないなど制約が多い．

2.2 JavaScript ファイルのダイナミックロード

この Ajax の問題を回避する手法として，Ajax に代わり JavaScript ファイルのダイナミックロードが注目されている．これは表示されている Web ページのブラウザ上に展開されている DOM ツリーを操作して，URL で指定することのできる任意の JavaScript プログラムを実行する為の仕組みで，URL でアクセスできる全ての JavaScript リソースにアクセスすることができる．例えば `http://yokoyama.ac/example.js` を読み込む為の JavaScript プログラムを次に記す．

```
01:  var script = document.createElement('script');
02:  script.charset = 'UTF-8';
03:  script.src = 'http://yokoyama.ac/example.js';
04:  document.body.appendChild(script);
```

01 行目で読み込む JavaScript コードを入れる <script> 要素を作成しており，03 行目で読み込む JavaScript コードの URL を src 属性として指定している．04 行目で <body> 要素直下に <script> 要素を挿入することにより，ブラウザは src 属性に指定された URL を読み込み，そこに書かれた JavaScript プログラムを実行する．

src 属性にクエリストリングを含んだサーバサイドスクリプトの URL を指定することにより，Web サーバから動的な結果を得ることができ，Web ブラウザ・サーバ間の通信手段として利用することができる．

しかしながら，この手法では読み込むデータ形式はあくまで JavaScript のプログラムコードであり，XML 等汎用のデータ記述形式を直接扱えない事は，ブラウザ上で効率の良いデータ処理を考える際には問題となる．

2.3 JSON

JSON は JavaScript の文法に従った半構造データ記述形式として注目を浴びている．JSON を利用すれば，前節で述べた JavaScript ファイルのダイナミックロードを利用して容易に半構造データを読み込みメモリ上に展開することができる．JSON は XML と互換性が高く，任意の XML 文書を JSON 形式に変換するゲートウェイも存在する [5]．このゲートウェイを利用すれば任意の Web サービスをブラウザ上から容易に利用することができる [6].

JSON は非常に単純な構造で構成されている．文字列，数字，真偽値，null という四つのプリミティブ型を持ち，配列もしくはオブジェクトを構成することができる．また入れ子にすることも可能であり，文法は JavaScript の完全なサブセットとなっている．JSON で記述された JavaScript ファイルが前節で述べた方法により Web ページに読み込まれた場合，ブラウザの利用しているメモリ内に展開され，その Web ページ内の JavaScript コードから半構造データにアクセスすることができる．

この方法はメモリ内に半構造データを展開し，そのオブジェクトの木構造にプログラムからアクセスするという点で DOM と類似した手法であると考えることができる．この手法の利点

は，メモリ上に展開された木が更新可能であり，また任意の要素にアクセスするのが容易であるという点である．一方で欠点は，処理を行うコンピュータが搭載しているメモリ量によって，利用できる XML 文書のサイズが制限されてしまう事である．Web クライアントの計算資源は多種多様であり，これら全てをターゲットとするサービスを考える際，この制約は非常に重要な問題であると考えられる．

2.4 SAX

XML 処理において搭載メモリ量に左右されない XML 処理手法として SAX が利用されている．SAX は XML 文書を先頭から走査し，要素開始，文字列，要素の終了等を発見する度にイベントとしてユーザに通知する順次アクセス型の手法で，読み込みと同時に処理を開始でき，メモリ負担も低い等の特徴を持っている．DOM と SAX の比較を表 1 に記す．

表 1　DOM と SAX
Table 1　DOM vs. SAX

比較項目	DOM	SAX
モデル	木	イベント列
処理方向	ランダムアクセス可	順次アクセスのみ
処理方式	対話的	バッチ式
更新	可	不可
処理のオーバヘッド	大きい	小さい
メモリ消費	大きい	少ない

このように，DOM と SAX 共に異なる利点を持ち，用途によって使い分けられている．

2.5 XML for <SCRIPT>

ブラウザ上において，DOM のようにメモリに半構造データの木を展開する手法としては先述の JSON がある．一方で SAX のようなメモリ負担が少ないイベントドリブン型の XML 処理手法はあまり議論されていない．

SAX のようにイベントハンドラを利用して XML 文書を JavaScript で読み込む手法としては XML for <SCRIPT> [7] がある．JavaScript によってイベントハンドラが記述でき，URL でアクセス可能な任意の XML 文書を読み込む点では SAX と同等であるが，XML 文書を一旦ブラウザ内に読み込んだあと，そのデータに対してパースを行っている為，SAX の特徴であるオーバヘッドの小ささと消費メモリの少なさが実現できていない．この手法は SAX のプログラミングモデルを継承しているが，SAX と同等であるとは言えない．SAX は DOM と補完しあう関係であるが，XML for <SCRIPT> を利用して，DOM の利用が適さない大きな XML 文書を読み込む事は現実的ではない．

Web ブラウザで表計算やワードプロセッサなど高度なアプリケーションが提供されはじめている現状を鑑みると，Web ブラウザ・サーバ間での密接なデータ連携がより重要になると考えられる．サーバ用コンピュータからシンクライアントまで存在する Web において通信するデータ量のスケーラビリティー確保は重要な課題であると考える．SAX は計算資源の大小に関わらず大きな XML 文書を処理する仕組みである．Web ブラウザ上においてオーバヘッドが少なく，消費メモリも小さい SAX を実現する事は重要な課題であると考えた．

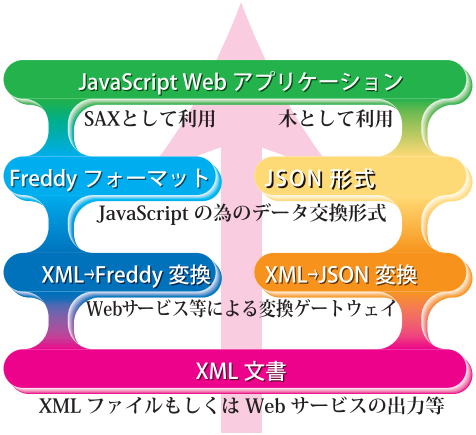

図 1　Freddy と JSON の比較
Fig. 1　Freddy Vs. Json

2.6 提案手法の概要

本論文は，メモリ上に木構造を展開する JSON の欠点である高メモリ負荷を補完する，イベントドリブン型のツール Freddy の提案である．Freddy もしくは JSON を利用して XML 文書を扱う手法の概要を図 1 に示す．

提案手法ではサーバ側において XML をイベントの順序に従い細かく分割し，複数のファイルに保存する．それらのファイルは JavaScript のサブセットである独自のフォーマット (Freddy フォーマット) で記述され，イベント毎に SAX イベントハンドラを呼び出すコードが定義されている．分割された複数のファイルを順番にクライアントである Web ブラウザにダイナミックロードする事により，クライアント側で定義された SAX イベントハンドラが呼び出されて，SAX イベントがクライアントプログラムに通知される．読み込んで実行されたファイルを順次メモリから開放する事により，XML 文書全体を読み込む必要なく効率的に動作する．

これらの処理はバックグラウンドで行われ，XML ファイルの分割，取得，統合は利用者から隠蔽されている．利用者は Freddy へ読み込みたい XML 文書の URL とイベントハンドラを渡すだけで動作させることができる．

この手法ではサーバ側で XML ファイルを分割しており，その為のオーバヘッドが発生する．その影響を最低限にする為，転送データの圧縮を行い通信量を削減している．圧縮には先行研究である要素名圧縮手法 [8] を用いている．

本研究は既存の SAX を改良する訳では無い事には注意が必要である．本研究は Web ブラウザという非常に制限された環境において，通常のスタンドアロンアプリケーションにおける SAX 環境をどのように実現するかに着目しており，性能的な優位性を主張するものではない．しかしながら，提案手法のスケーラビリティーを検証する為，後述する実験にてスタンドアプリケーションの SAX 環境との比較を行っている．

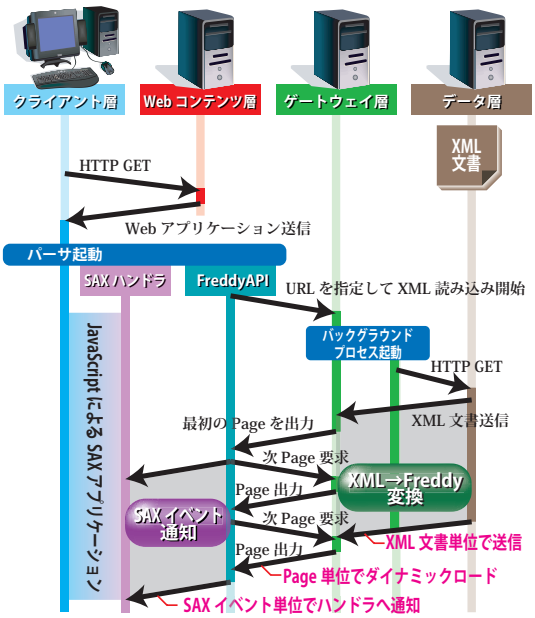

図 2　Freddy の概要

Fig. 2　Overview of the Freddy

3. 提 案 手 法

本章では Freddy の仕様とその実装手法を詳述する．図 2 は Freddy が SAX パーサとして動作する仕組みを表している．Freddy を用いた XML 文書の呼び出しは Web ブラウザであるクライアント層，ブラウザが表示している HTML ファイルを持つ Web サーバ (Web コンテンツ層)，XML 文書を Freddy フォーマットに変換し規定されたイベント数毎に一つのファイルとして保存するゲートウェイ層，そして XML 文書が存在する Web サーバ (データ層) という四層からなる．

Freddy は XML 文書の URL と SAX イベントハンドラが渡されると以下の手順で動作する．

（1） **クライアント層:** ゲートウェイに URL を通知，(JavaScript ファイルのダイナミックロードを利用してゲートウェイにアクセスする)

（2） **ゲートウェイ層:** URL のファイルを読み込む為のバックグラウンドプロセスを起動，最初の Freddy フォーマットファイルの完成を監視する

（3） **バックグラウンドプロセス:** URL のファイルを読み込み Freddy フォーマットへ変換，ファイルへ保存する

（4） **ゲートウェイ層:** 最初の Freddy フォーマットファイルの完成したら，それをクライアントへ出力する

（5） **クライアント層:** ゲートウェイの出力結果である Freddy フォーマットのファイルがダイナミックロードされ，利用者が実装した SAX イベントハンドラにイベントを通知する

（6） **クライアント層:** 次の Freddy フォーマットファイルを読み込む (後述するが読み込むデータは Volume と Page という値の組で識別される)

（7） **ゲートウェイ層:** 指定された Volume に該当するファイルの完成を監視し，完成していたら指定された Page をクライアントへ出力する

（8） **クライアント層:** 読み込んだファイル内の SAX イベントを利用者へ通知した後，データは即時開放される

（9） XML 文書が終わりを示す endDocument イベントが通知されるまで手順 6 から手順 8 を繰り返す

手順 3 はバックグラウンドで行われる．提案手法では一つの XML 文書を規定された SAX イベント数毎に断片化し，それぞれ独立したファイルに保存している．デフォルトでは 5000 イベント毎に断片化しているが，この値はクライアントから指定できる．

この仕組みにより，ゲートウェイ層で XML 文書全体を読み終わらなくとも，断片化されたファイル毎に出力する事が可能となり，処理のオーバヘッドを軽減している．利用者へ最初の SAX イベントが通知されるのは，XML 文書の大きさに関わらず，ゲートウェイが XML 文書から 5000 個の SAX イベントを取り出して Freddy ファイルに保存した時点である．

3.1 Freddy フォーマット

3.1.1 基本フォーマット

本論文で提案する Freddy フォーマットは，SAX のイベント列を基にし，JavaScript の文法を持つ，XML 形式と互換性のある半構造データ記述手法である．本論文では説明の簡単の為，文書開始，要素開始，文字列，要素終了，文書終了の 5 つのイベントのみを扱う事とする．本節ではこれらのイベントに基づいてた XML 互換の Freddy フォーマットを説明する．また次節では，大きな Freddy フォーマットファイルの効率的な分割に関して述べる．

まずは，以下の簡単な XML 文書を考える．

```
01:  <list>
02:      <cocktail name="Martini">
03:          $10-
04:      </cocktail>
05:      <cocktail name="Gibson">
06:          $10-
07:      </cocktail>
08:  </list>
```

この文書から得られた SAX イベント列を表 2 に示す．

表 2　SAX イベント列の例

Table 2　An example of SAX event sequence

順序	イベント　プロパティ
1	文書開始
2	要素開始　要素名:list
3	要素開始　要素名:cocktail, 属性:name=Martini
4	文字列　$10-
5	要素終了　要素名:cocktail
6	要素開始　要素名:cocktail, 属性:name=Gibson
7	文字列　$10-
8	要素終了　要素名:cokctail
9	要素終了　要素名:list
10	文書終了

Freddy フォーマットはこの SAX イベント列の各イベントを対応する JavaScript のコードに置き換えたものである．

```
01:  h.sd();
02:  h.s('list');
03:  h.s('cocktail','{"name":"Martini"}');
04:  h.c('$10-');
05:  h.e();
06:  h.S('b','{"name":"Gibson"}');
07:  h.c('$10-');
08:  h.e();
09:  h.ed();
```

先頭の変数 h はイベントハンドラオブジェクトを参照している．すなわち，それぞれの行は直接イベントハンドラ上のメソッドを呼び出す JavaScript コードになっている．提案手法ではこのメソッド呼び出しを総称してイベントコンテナと呼んでいる．このファイルを前述した方式で動的に Web ページ内に読み込む事で実行され Web ページ上で定義されたイベントハンドラに SAX イベントを通知することができる．

Freddy フォーマットでは複数回現れる要素は全て短い名前の要素名に符号化される．Freddy フォーマットは XML と比べ，要素名・属性名を含む全ての文字列を引用符で括る必要がある等，冗長な文法である．そこで要素名を符号化する事により，サイズの増加を防いでいる．符号化後の要素名は圧縮要素名と呼ぶ．圧縮要素名は以下の正規表現を満たす値で構築され，ゲートウェイとクライアントで同じ符号化手法を用いることにより，辞書ファイル無しで動的な復号を実現している．

CompressedElementName = `^[a-zA-z0-9]+$`

現在，提案手法では英数字を用いた六十二進数の整数を初出の要素に順に割り当てる事で符号化を行っている．よって上記 Freddy フォーマット例の *cocktail* という要素名は list に続いて二番目に登場する要素であり，*b* という圧縮要素名に符号化される (06 行目)．

初出の要素開始とそれ以外の要素開始は異なるメソッドを呼び出している．クライアント側で適切に解釈され動的に辞書を作る事で，圧縮要素名から元の要素名を容易に得ることができる．表 3 は Freddy のイベントコンテナの種類とそれに対応した SAX イベントの関係表である．

表 3　SAX イベントと対応するイベントコンテナ
Table 3　SAX events and Event containers

SAX イベント	イベントコンテナ	備考
文書開始	h.sd();	
要素開始 (初回)	h.s(要素名, 属性);	属性は省略可
要素開始	h.S(圧縮要素名, 属性);	属性は省略可
文字列	h.c(文字列);	
要素終了	h.e(要素名);	要素名の省略を推奨
文書終了	h.ed()	

3.1.2　Volume と Page

前節では XML 文書から得られる SAX のイベント列を Freddy のイベントコンテナに変換する手法を述べた．イベントコンテナはそのままクライアント側の SAX イベントハンドラを呼び出す JavaScript のコードであり，Web ページにダイナミックロードすることで利用者に SAX イベントを通知することができる．

ダイナミックロードされたファイルは，Web ブラウザ上のメモリ領域に書き込まれる．つまり，XML 文書全体を Freddy フォーマットとして一つのファイルに保存する場合，結果として XML 文書全体を一旦メモリ上に展開する事になり，SAX の特徴である低消費メモリとは相反する．

そこで提案手法では一つの XML 文書を断片化し，複数のファイルに保存している．クライアント側へ一回のコネクションで送信されるイベントコンテナの単位を Page と呼んでいる．デフォルトでは 5000 イベントで 1Page を構成している．そして，複数の Page をまとめて一つのファイルに保存している．このファイルを Volume と呼んでおり，SAX イベントの順に 0 から始まる整数値の Volume 名で識別される．デフォルト環境において Volume 毎の Page 数は最大で 10Pages となっている．任意の Volume 名 vol に格納する Page 数 $N(vol)$ は以下の式で求められる．

$$N(vol) = \begin{cases} 10 & (vol > 10) \\ vol & (1 \leqq vol \leqq 10) \\ 1 & (vol = 0) \end{cases}$$

なお，Page 内のイベント数および Volume の最大値は計算機環境，通信環境によってクライアント側で値を変更できる．提案手法では XML 文書のイベント数に Page 数が比例する．つまり大きな XML 文書を扱う際は，非常に多くの Page を扱わなければならず，Page 毎にファイルを構成すると，ファイル数が非常に多くなってしまう．現状，多くのファイルシステムでは数百，数千単位のファイルが一つのディレクトリに格納されると，ファイルの処理速度が非常に遅くなってしまう．そこで複数の Page を Volume という単位で束ねて一つのファイルにする事で，ファイル数の過度な増加を防ぐことができる．

さらに，上記式のように 0 以上 10 未満の Volume において Page 数を段階的に増加させる事により，パース初期段階においては比較的小さな大きさの Volume ファイルとなり，ファイルの作成時間が短縮される．クライアント側にはこの Volume ファイルが完成した後にその中の各 Page を送出するため，初期段階における Volume ファイル作成時間の短縮は，利用者への SAX イベント通知速度が向上する事を意味する．一般論として，クライアントマシンよりゲートウェイとなるサーバマシンの方が性能が良いと考えられる為，パース初期段階以降に関してはサーバ側のパース速度が十分に速いものと考えられる．

ダイナミックロードされた各 Page は <script> 要素として Web ブラウザが表示している HTML ファイルの DOM ツリーに挿入される．各 Page の最初には次の Page を読み込む命令が書かれており，ある Page が実行されると同時に次の Page の読み込みを開始し，通信による待ち時間を低減させている．また Page の最後にはその Page 自身をメモリ上から開放する命令が書かれている．一つの Page を読み終わる毎にその Page

を DOM ツリーから削除する事により，読み終わったデータをメモリから開放し，消費メモリを抑えている(注1)．

これらの仕組みにより，高速に動作し消費メモリが少ないという特徴をもった SAX パーサを，Web ブラウザ上の JavaScript から利用することができる．

3.2 Freddy SAX API

これまでに述べたように，提案手法ではゲートウェイと Web ブラウザ間で非常に多くの通信が発生し，また圧縮要素名の復号等の処理をクライアント側で行わなければならない．しかしながら提案手法ではこれらを SAX API 内に隠蔽している為，利用者は通常の SAX を利用したプログラミングモデルに従って JavaScript コードを記述することができる．本節では Freddy が提供する SAX API の利用方法について説明する．

3.2.1 SAX イベントハンドラ

3.1.1 節にて XML 文書から得られた SAX イベントはイベントコンテナに変換され Freddy フォーマットのファイルへ格納される事を述べた．イベントコンテナはそのまま Web ブラウザ上で対応する SAX イベントハンドラ関数を呼び出す命令として解釈される為，Page が実行されると SAX のイベント順に定義された関数呼び出しを行う．

各イベントコンテナに対応した関数を Web ブラウザ上で予め定義する事によって，ゲートウェイ側で解釈した SAX イベント列が Web ブラウザ上の JavaScript プログラムへ渡される．この仕組みによって，SAX アプリケーションを JavaScript で構築することができる．しかしながら，イベントコンテナはサイズの削減の為に可読性の低い短い関数名を使っている．そこで提案手法ではイベントコンテナをラップすることによって，利用者は Java における SAX イベントハンドラと同じ関数名を使って，ハンドラを定義する事ができる．以下に SAX イベントとイベントコンテナ，イベントハンドラ関数の関係表を記す．

表 4　イベントコンテナと対応するハンドラ関数
Table 4　SAX events and Event containers

SAX イベント	イベントコンテナ	ハンドラ関数
文書開始	h.sd();	h.startDocument();
要素開始	h.s(…);h.S(…);	h.startElement(…);
文字列	h.c(…);	h.Charactors(…);
要素終了	h.e(…);	h.endElement(…);
文書終了	h.ed()	h.endDocument();

利用者は Java における SAX と同じ関数名のハンドラを定義することにより，SAX イベントを受け取ることができる．また，イベントコンテナのメソッド呼び出しはシステムによって自動的に対応するハンドラ関数へリダイレクトされる．またその際に圧縮要素名は自動で展開されるため，利用者は要素名の符号化を意識せずにハンドラを記述することができる．

(注1)：利用している物理メモリ領域が実際に開放されるのは利用している Web ブラウザの判断による．

3.2.2 SAX パーサの起動

Freddy における SAX パーサの起動は，一般的な SAX API と同様である．まずパーサのインスタンスを作成し，イベントハンドラを設定する．そして処理する XML 文書の URL を指定する事で XML 文書の読み込みが開始される．

```
01:  //新しいパーサの作成
02:  p = new freddy.SAXParser();
03:  //イベントハンドラのインスタンスを作成
04:  h = new MySaxHandler();
05:  //パーサにイベントハンドラを設定
06:  p.setSimpleEventHandler(h);
07:  //url の XML 文書をパースする
08:  p.parse("http://yokoyama.ac/freddy/dataS.xml");
```

4. 性能評価

Freddy を利用した XML 文書処理の性能を計測するために 1MB から 100MB までのサイズの異なる三つの XML 文書を作成した．そして，それらの文書に対して Python で実装した SAX アプリケーション及び，Web ブラウザ上で提案手法を用いて実装された SAX アプリケーションがそれぞれインターネットを介して読み込みに要する時間を計測した．

本実験では，SAX を用いてネットワーク越しの XML 文書を処理するシナリオにおいて，スタンドアロンアプリケーションでこれを行う場合と提案手法を利用する場合を比較する事により，Web ブラウザを用いて大規模な XML 処理が可能かどうかを明らかにする．

4.1 実験環境

性能評価の為に図 2 で示した環境を実際に構築した．システムの簡単化の為，Web コンテンツ層，ゲートウェイ層，データ層は同じサーバを利用している．サーバの性能を表 5 に記す．

表 5　サーバーマシンの性能
Table 5　The server machine specificatione

ハードウエア	DELL PowerEdge 2650
CPU	Intel Xeon 2GHz × 2
メモリ	512MB
OS	Linux (Fedora Core 6)
HTTPD	Apache 1.3.37

クライアントは Freddy による SAX アプリケーションが実際に動作するマシンである．性能を表 6 に記す．

表 6　クライアントマシンの性能
Table 6　The client machine specificatione

ハードウエア	IBM ThinkPad X41 Tablet
CPU	Intel Pentium M 1.6GHz
メモリ	1GB
OS	Microsoft Windows XP Tablet Edition
Web ブラウザ	Internet Explorer 6
	Mozilla FireFox2

インターネットにおける一般的な利用環境を想定して，サー

バマシンとクライアントマシンは地理的にもネットワーク的にも異なる場所に配置している．

次に性能評価実験で利用した実装系を記す．クライアントマシンの Web ブラウザ上で動作する SAX アプリケーションは JavaScript で実装されている．ゲートウェイ層とクライアント層との通信は Apache HTTP サーバ上で動作する PHP スクリプトによって実装されている．バックグラウンドで動作する XML 文書を Freddy フォーマットへ変換するプログラムは Python で実装している．

本実験で利用するテストデータは XML 処理の標準的なベンチマークである XMark [9] の xmlgen を用いて，1MB，10MB，100MB の XML 文書を作成した．それぞれ XML ファイルとしてデータ層の Web サーバ上へ配置した．

4.2 実験方法

本実験では Python によって実装されたスタンドアロンアプリケーションと Freddy を利用した JavaScript による Web アプリケーションの比較を行う．Web アプリケーションの実行には Microsoft Internet Explorer 6(以下 IE と略す) と Mozilla FireFox2(以下 FF と略す) の双方を用いた．つまり，本実験では以下の 5 つの場合に関して，SAX によって XML 文書を処理し終わるまでの時間を計測している．

1. SAX:
   Python で実装されたスタンドアロンアプリケーション
2. Freddy(FF):
   FF を利用して XML 文書を直接読み込む
3. Freddy(FF) Cache:
   Freddy フォーマットに変換済みのデータを FF で読み込む
4. Freddy(IE):
   IE を利用して XML 文書を直接読み込む
5. Freddy(IE) Cache:
   Freddy フォーマットに変換済みのデータを IE で読み込む

計測は三つのテストデータに対して，上記 5 パターンの SAX の処理終了までにかかる時間を計測した．計測は 100 回ずつ行った．次節にその結果を記す．

4.3 実験結果と考察

各テストデータに対し 100 回行った実験の結果から最大値，最小値，平均値を算出し図 3，図 4，図 5 に示す．

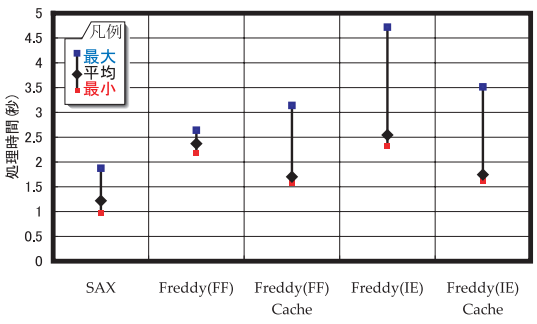

図 3　実験結果 (size=1MB)
Fig. 3　Experiment Result (size=1MB)

これらの結果から，まず IE と FF はほぼ同じ動作速度が得られることが分かった．ただし 100MB のテストセットにおいては若干 IE が勝っている．動作速度に関しては Freddy フォーマットでキャッシュした場合はスタンドアロンアプリケーションによる SAX の処理の 1.5 倍程度，XML 文書を直接 Freddy で読み込む場合はスタンドアロンアプリケーションに比べて 2 倍程度の処理時間を要する事が分かった．文書サイズによる処理時間の関係を図 6 に示す．

Web アプリケーションの利点は全てのインターネットユーザが利用できる Web ブラウザ上でインストール無しに利用できる点である．過去にスタンドアロンアプリケーションとして提供されていたワードプロセッサソフトや表計算ソフトが，現在は Web ブラウザ上で利用することができる．このように多く

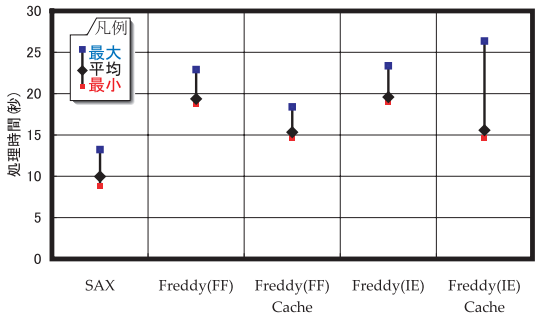

図 4　実験結果 (size=10MB)
Fig. 4　Experiment Result (size=10MB)

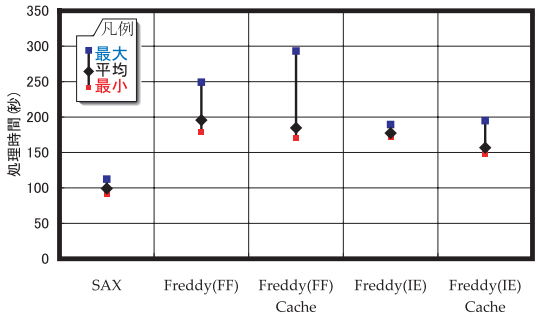

図 5　実験結果 (size=100MB)
Fig. 5　Experiment Result (size=100MB)

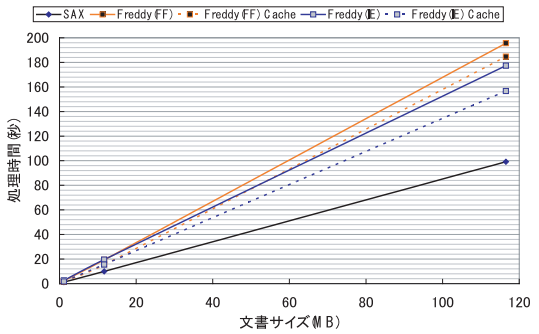

図 6　実験結果 (総合)
Fig. 6　Experiment Result (Summary)

のソフトが Web アプリケーションへ移行している．

提案手法 Freddy はプログラミングに関する制約の多い Web ブラウザ上での実装であり，今回の実験ではスタンドアロンアプリケーションと同等の処理速度とはならないかったが，Freddy の処理時間は文書サイズにのみ比例する事は示せた．Web アプリケーションの利点と急速な普及を考えると，本実験結果は提案手法が Web アプリケーションから利用できる実際的な性能を有している事を表していると考える．

4.4 議　　論

4.4.1 メモリ管理に関する議論

前述の実験の過程で，JavaScript 上でダイナミックロードした Freddy フォーマットの Page を開放をしても，すぐには実際のメモリ領域から開放されない事が判明した．これはブラウザのガベージコレクタがページが遷移した時にのみ動作する為であると考えられる．現在はダイナミックロードした Page を，ブラウザが表示している HTML ファイルの DOM ツリーに <script> タグとしてバインドしており，これを内部的なページ遷移を伴う <iframe> タグに変更することを検討している．

<iframe> は表示 Web ページ中の特定領域に別の Web ページを表示する為のタグで，中に HTML でラップした Freddy フォーマットの Page を表示させる事で，<script> タグにバインドした場合と同様に SAX イベントハンドラへ通知できる．

そこで，<iframe> タグを利用する手法を実装し，<script> タグへバインドする手法と比較するの為の予備実験を行った．100MB のデータに対する両手法の CPU 利用率とメモリ消費量を Windows 付属のタスクマネージャで計測し，図 7 に示す．

この結果から，<iframe> を利用すれば，JavaScript 上で Page を開放すると同時に実際のメモリ上からも削除され，メモリの消費量が抑えられる事が分かった．しかしながら，処理時間は <script> タグへバインドする手法よりも多くかかってしまう．そこで，これら両方を利用した手法の開発を現在検討している．これら実装の最適化に関しては今後の課題である．

4.4.2 応用手法に関する議論

本節では提案手法の応用範囲について議論する．提案手法 Freddy は Web サービスの出力等比較的小さな XML 文書をブラウザから扱う為にも適しているが，メモリ上に木を展開する JSON でもそのようなデータは扱う事ができる．SAX と DOM の関係のように，提案手法も JSON で扱えないデータを処理するための手法としての応用を考えている．

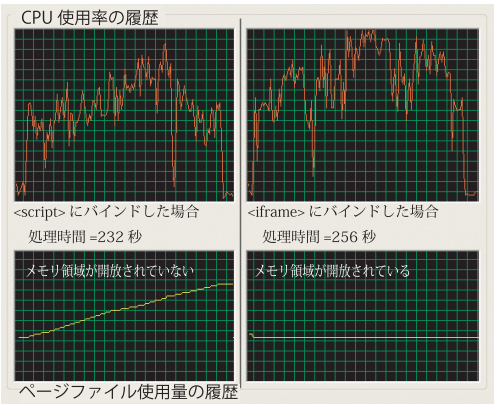

図 7　<script> バインドと <iframe> バインドの比較

Fig. 7　The comparison of a <script> binding and a <iframe> binding

今回の実験では性能評価の為に 100MB の巨大な XML 文書をそのまま利用したが，実際の応用として処理に 3 分かかる XML 文書を Web ブラウザ上で利用する事は考えづらい．しかしながら，Web ブラウザ上で地図ソフトやワードプロセッサソフトあるいはゲームソフト等，様々なソフトが利用されている現在，一つの Web ページに 3 分，あるいはそれ以上留まる事は珍しい事ではなくなった．今後ともそのような Web アプリケーションは爆発的に増加すると考えられる．

提案手法は，そのような Web アプリケーションのサーバ・Web ブラウザ間でのデータ交換手法として応用が可能であると考える．提案手法を利用すれば，Web ブラウザはサーバと長時間コネクションを維持する事が可能である．そして Web アプリケーションは，その目的に応じた SAX イベントハンドラを実装することで動作し，通信に関する詳細は Freddy の提供する API 内に隠蔽される．

5. まとめと今後の課題

本論文では Web ブラウザ上から利用できる SAX パーサ “Freddy” を提案した．XML 文書を Page という単位で断片化し，それぞれの Page を随時読み込む事により，SAX の特徴である高速動作を実現した．また処理の終わったデータを削除する事により消費するメモリ量を考慮した．実験によりスタンドアロンアプリケーションの 50%から 75%のスループットを持つ SAX アプリケーションを Web ブラウザ上に構築できる事を示した．またブラウザのメモリ管理の欠点を指摘し，それを回避する方法を議論した．

今後の課題は前述の項目に加えて，サーバ側の実装の最適化を考えている．

文　　献

[1] J. J. Garrett: “Ajax: A new approach to web applications”, Web. http://adaptivepath.com/publications/essays/archives/000385.php.

[2] T. O’Reilly: “What is web 2.0 design patterns and business models for the next generation of software”, Web. http://www.oreillynet.com/lpt/a/6228.

[3] “Introducing json”, Web. http://www.json.org/.

[4] D. Crockford: “Json: The fat-free alternative to xml”, Web. http://www.json.org/xml.html.

[5] “Xml2json service”, Web. http://www.drk7.jp/MT/archives/001011.html.

[6] 横山, 的野, サイド ミルザ, 小島：“Web2.0 における javascript コードのモジュール化とマッシュアップの枠組み”, 日本データベース学会 Letters, **5**, 3, pp. 1–4 (2006).

[7] “Xml for <script> cross platform xml parsing in javascript”, Web. http://xmljs.sourceforge.net/website/documentation-sax.html.

[8] 横山, 太田, 石川：“要素名圧縮による xml データ圧縮手法の提案— simplified element xml —”, データベースと Web 情報システムに関する IPSJ DBS/ACM SIGMOD Japan Chapter.JSPS-RFTF AMCP 合同シンポジウム (DBWeb2000) (2000).

[9] “Xmark - an xml benchmark project”, Web. http://monetdb.cwi.nl/xml/.