JavaScript マルチスレッドライブラリの実装と応用

牧　大介[†]　岩崎英哉[††]

Ajax 開発は，従来の Web 開発に比べて，複雑な非同期処理を 1 つのスレッドの上にすべて記述しなければならない点，JavaScript では非同期通信をイベント駆動型でしか記述できないため，制御フローの記述が困難である点が挙げられる．この問題を解決するため，我々は JavaScript のためのマルチスレッドライブラリを開発した．開発したライブラリは，代表的な複数の Web ブラウザで可搬性を維持しつつ，プリエンプティブなスレッド切り替えが可能である，さらにオブジェクト指向で API を提供している．このライブラリは，マルチスレッドスタイルで記述された JavaScript プログラムを継続ベースの並行処理プログラムに変換することで実装している．現在，JavaScript のフルセット言語の変換をサポートしている．この実装を用いて実際にいくつかのアプリケーションを記述すること，本ライブラリの記述性について評価を行った．提案機構を用いた場合には，非同期通信を含むプログラムでも制御フローを明示的に記述できる点，サーバ側手続きを JavaScript の関数を用いて抽象化できる点などによって，Ajax アプリケーションの記述について有効であることを確かめた．

An Implementation and Applications of JavaScript Multithread Library

Daisuke Maki[†] and Hideya Iwasaki[††]

1. はじめに

Ajax を利用した Web アプリケーションが広く普及したが，Ajax アプリケーションの開発が難しいという問題は依然として残されたままである．我々は Ajax アプリケーションの開発に必須である非同期通信の記述が煩雑になってしまう点に着目し，この問題を解決するために JavaScript にマルチスレッドフレームワークを実装した[9]．現在当フレームワークは実装をさらに進め，JavaScript のフルセット言語をサポートをするに至っている．

本発表では，提案機構を応用した事例について，具体的なコードを紹介しながら，従来と比較し実際にどのような利点があるのかを報告する．主として，従来の JavaScript で主流であったイベント駆動型のプログラミングに起因する問題を取り上げ，それらの問題がどのように解決されているかを述べる．

本発表における JavaScript は，Web ブラウザに組み込まれている ECMAScript[2] の互換言語を指すものとする．

† 電気通信大学電気通信学研究科
Graduate School of Electro-Communications, The University of Electro-Communications
†† 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications

本研究の一部は，情報処理推進機構 (IPA) 2006 年度下期未踏ソフトウェア創造事業の支援を受けている．

2. JavaScript におけるスレッド

2.1 JavaScript のスレッドモデル

本来 JavaScript は UI スレッドと呼ばれる 1 つのスレッドしか持っておらず，その UI スレッドの上ですべての処理を行う．UI スレッドはその名の通り Web ブラウザのユーザインタフェースを担当するスレッドであり，ユーザの入力に対する応答や画面の再描画などの GUI の処理もこのスレッドの上で実行される．GUI と JavaScript が同じ 1 つのスレッドの上で実行されるため，JavaScript が時間のかかる処理を行うとその間 GUI に関わる処理を行うことができず，ユーザからは Web ブラウザがフリーズしたように見えてしまう．ブラウザがフリーズしてしまうというのはユーザ応答性の面で問題なので，できるだけ避けなければならない．従って，JavaScript でのアプリケーション開発の難しさのひとつは，ブラウザがフリーズしたようにユーザに思わせることなく，長時間に渡って UI スレッドを占有し続けないようにプログラミングすることである．

従来の JavaScript の主な用途は，簡単なアニメーションやフォームの入力チェックなど，比較的軽量なものに限られていたため，UI スレッドを考慮したプログラミングが必要となる機会はほとんどなかった．しかし次で述べる Ajax の登場により，JavaScript がサーバとの通信を含む，より複雑な処理を行うようになったことで，これは無視することのできない問題となった．

2.2 Ajax 開発における問題

Ajax[3] の中心となっている技術は，JavaScript によるサーバとの非同期通信である．従来の Web アプリケーションでは，サーバと通信する機会はページのデータを読み込む際しかなかった．つまりサーバとデータをやり取りするためには，ページ全体を新たに読み直すしか方法がなかったのである．ページを読み込んでいる間アプリケーションのユーザは待つことしかできないため，従来の Web アプリケーションではユーザ応答性が難点であった．しかし，JavaScript プログラムからサーバと通信する方法が確立されたことで，通信のたびにページ全体を読み直す必要がなくなり，高いユーザ応答性を持つ Web アプリケーションを実現することができるようになった．

JavaScript からのサーバとの通信は，原則として非同期通信によって行われる．これは先に述べた JavaScript のスレッドモデルに起因している．同期通信は通信が完了するまでプログラムの実行を待機させる方式であるため，JavaScript でこれを行うと UI スレッドをブロックさせてしまい，通信の待ち状態の間ブラウザがフリーズしてしまうためである．一方，非同期通信はイベント駆動型のプログラムによって記述されるが，イベント駆動型のプログラムでは制御フローの記述が困難になってしまうことがよく知られている[5),8)]．Ajax プログラミングにおいてもこの制御フローの問題は同様であり，Ajax 開発を困難にしている主な要因のひとつとなっている[6)]．

3. マルチスレッドフレームワーク

前章で述べたユーザ応答性の問題は，JavaScript が UI スレッドという 1 つのスレッドしか持っていないことが原因であった．そこで我々はこの問題を解決するため，JavaScript の上にマルチスレッド環境を実現するためのライブラリを提案した[9)]．提案機構の設計にあたり，当初に掲げた目標は次の通りである:

(1) 代表的な Web ブラウザの間で可搬性がある．
(2) プリエンプティブなスレッド切り替えが可能である．
(3) オブジェクト指向で API を提供する．

目標 1 については，Ajax が普及した背景の 1 つとしてブラウザ間の互換性が高まったことがあるため，特に重要視した．この目標は，提案機構をすべて JavaScript で実装することにより，ブラウザ拡張やプラグインを必要とすることなく達成されている．また前章で述べたように，JavaScript プログラミングでの大きな問題点は，ひとつの処理が長時間にわたって UI スレッドを占有し続けないように実装することであった．提案機構はこの問題からプログラマを解放するために，プリエンプティブなスレッドスケジューリングを目標とした．

現在提案機構は JavaScript のフルセット言語と一部の拡張構文をサポートしており，おおむね ECMAScript 3rd Editon[2)] に準拠している．実装は非同期通信ライブラリなどの上位ライブラリと共にオープンソースとして公開されており，Web サイト☆ よりダウンロードすることができる．

3.1 使 用 例

提案機構はライブラリとして実装されているため，基本的には読み込むだけで利用可能な状態となる．**図 1** に簡単な使用例を示す．図 1(a) の最初の行が提案機構を読み込んでいる箇所である．ここでは提案機構を thread.js というファイル名で保存してあるとしている．4 行目と 10 行目の `Concurrent.Thread.create` は提案機構が提供するメソッドで，新しくスレッドを作成し，引数として渡された関数をそのスレッドの上で実行する．このプログラムを実行すると，いくつか「Java」と表示されたあとに続けていくつか「Script」と表示され，またいくつか「Java」が表示され，また「Script」と表示される，というように，2 つの実行がプリエンプティブに実行を切り替えながら交互に実行される様子がわかる．

提案機構のもうひとつの利用方法として，HTML の script 要素の type 属性を用いることもできる．例えば 図 1(a) のプログラムは図 1(b) のように書くこともできる．下 2 つの script 要素の type 属性が「`text/x-script.multithreaded-js`」となっているところに注目されたい．この値を指定しておくと，提案機構がそれぞれの要素の内容を新しく作成したスレッドの上で実行するようになっている．

3.2 通信ライブラリ

提案機構は専用の通信ライブラリを提供している．これは UI スレッドをブロックすることなしに同期通

☆ http://jsthread.sourceforge.net

```
<script type="text/javascript"
 src="thread.js"></script>
<script type="text/javascript">
  Concurrent.Thread.create(function(){
    while ( true ) {
      document.body.innerHTML += "Java"
                              +  "<br>";
      Concurrent.Thread.yield();
    }
  });
  Concurrent.Thread.create(function(){
    while ( true ) {
      document.body.innerHTML += "Script"
                              +  "<br>";
      Concurrent.Thread.yield();
    }
  });
</script>
```

(a) create メソッドを用いた例

```
<script type="text/javascript"
 src="thread.js"></script>
<script
 type="text/x-script.multithreaded-js">
  while ( true ) {
    document.body.innerHTML += "Java"
                            +  "<br>";
  }
</script>
<script
 type="text/x-script.multithreaded-js">
  while ( true ) {
    document.body.innerHTML += "Script"
                            +  "<br>";
  }
</script>
```

(b) type 属性を用いた例

図 1　提案機構の使用例

信を可能とする．これによりイベント駆動型で記述する煩雑さを避けることができる．

通信ライブラリは `Concurrent.Thread.Http` という名前空間の下にまとめられており，HTTP の GET および POST メソッドを行うための関数 `get` と `post`，それ以外の HTTP メソッドのための関数 `send` を提供している．それぞれの関数は戻り値として `XMLHttpRequest` オブジェクトを返すが，呼出し元に返った時点で通信は完了しているため，通信の結果をコールバック関数を用いてイベント駆動で受け取る必要はない．

図 2(a) に通信ライブラリを利用する例を示す．これはサーバからデータを取得し，それに操作を加えてサーバに送り返す簡単なプログラムである．同じプログラムを提案機構を用いずに実装したものが図 2(b)

```
var req1, req2, result;
req1=Concurrent.Thread.Http.get(URL);
result=doSomething(req1);
req2=Concurrent.Thread.post(URL, result);
aleret(req2.responseText);
```

(a) 提案機構を用いた記述

```
var req1, req2, result;

req1 = new XMLHttpRequest();
req1.open("GET", URL, true);
req1.onreadystatechange = callback1;
send(null);

function callback1 ( ) {
  if ( req1.readyState == 4 ) {
    result = doSomething(req1);
    req2 = new XMLHttpRequest();
    req2.onreadystatechange = callback2;
    req2.send(result);
  }
};

function callback2 ( ) {
  aleret(req2.responseText);
}
```

(b) 従来記述

図 2　通信を行うプログラムの例

である．これだけのことをするためにも複数のコールバック関数を記述せねばならず，複雑な記述が要求されることがわかる．

3.3 実現方法

提案機構は，時分割によって UI スレッドの上に擬似的なマルチスレッド環境を構築することにより実現されている．基本的なイメージとしては，JavaScript プログラムを基本ブロック毎に細切れにして実行を進め，適切なタイミングに基本ブロックの切れ目で UI スレッドを解放するというものである．細切れの実行を実現するための中心的な手法はコード変換によっている．つまり，マルチスレッドスタイルに書かれた JavaScript プログラムを基本ブロック単位に分割し，各基本ブロックを関数にした形へと変換する．変換後のプログラムは細かい関数をくり返し呼び出すことで変換前のプログラムと同じ内容を実行できるようになっており，各関数の間で実行を中断できるようにしておく．この手法をライブラリとして実現するために，提案機構ではこのコード変換を行うプログラムも JavaScript 自身で実装されている．

変換したプログラムはスケジューラと呼ばれる実行部分によって呼び出される．**図 3** に提案機構の全体図を示す．スケジューラは，基本的には前述の細切れに

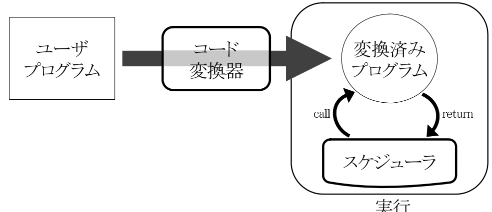

図 3 提案機構概観
Fig. 3 System overview

された関数をくり返し呼び出すだけのものである．ただし，各スレッドが順番に実行されるように，また，スレッドの処理の合間にも JavaScript が起こすイベント処理 (クリックやタイムアウトなどに対応するイベントハンドラ) が実行されるようスレッドの実行の予約を行う．基本的には，JavaScript の組み込み関数である `setTimeout` を用いてスレッドの実行を予約することで，スレッドとその他のイベントのスケジューリングを同じ Web ブラウザの内部で行わせることで，これを実現している．

4. 応 用 事 例

本章では提案機構の応用について，具体的なコード例を取り上げながら紹介する．

4.1 非同期通信

2.2 節で述べたように，Ajax 開発において従来の JavaScript のスレッドモデルが問題となるのは，主にサーバとの非同期通信を行うときであった．そのためまずは，提案機構が提供する通信ライブラリが既存の問題をどのように解決できるのかを紹介する．

4.1.1 ツリートラバーサル

Ajax アプリケーションの簡単なサンプルとして，ツリー型の電子掲示板システムを考える．これは最初にすべての記事データを取得してしまうのではなく，ユーザの要求に応じて要求駆動でサーバと通信を行い，順次データを取得する．しかしユーザが記事を読んでいる間にもネットワーク帯域を有効活用し，同時にバックグラウンドで記事の先読みを行うこととする．

提案機構を用いると先読みの記述は，**図 4**(a) のように書くことができる．簡単のため，記事データの読み込みを行う一引数の関数，`getArticle` があることとする．`getArticle` は記事の ID を引数に取り，結果として 1 つの記事を表すオブジェクトを返す．この関数を用いるとこのコードではまず，`getArticle` にキャッシュ機能を追加した `getArticleCache` を実装している．`getArticleCache` は内部で `getArticle` を使っており，ここではキャッシュは単純にグローバル変数で保持しておくこととする．続いて，この `getArticleCache` を用いて先読みを行う関数 `backgroundLoad` を実装する．これは記事 ID の配列を引数に取り，各 ID に対して `getArticleCache` を適用し，その結果の子記事を引数として `backgroundLoad` を再帰的に呼び出す．アプリケーションの電子掲示板における記事データはツリー構造になっているため，再帰的にツリーを辿ることですべての記事データをキャッシュすることができる．

```
var cache = {};

function getArticleCache (id) {
  if (!cache[id]) {
    cache = getArticle(id)
  }
  return cache[id];
}

function backgroundLoad (ids) {
  for (var i=0; i < ids.length; i++) {
    var a = getArticleCache(ids[i]);
    backgroundLoad(a.children);
  }
}
```

(a) 提案機構を用いた記述

```
var cache = {};

function getArticleCache (id, callback) {
  if (cache[id]) {
    callback(cache[id]);
  } else {
    getArticle(id, function(a){
      cache[id] = a;
      callback(a);
    })
  }
}

function backgroundLoad (ids, callback) {
  var i = 0;
  function l ( ) {
    if (i < ids.length) {
      getArticleCache(ids[i], function(a){
        backgroundLoad(a.children, l)
      });
    } else {
      callback();
    }
  }
  l();
}
```

(b) 従来記述

図 4 先読み処理の記述

上のように，提案機構を用いた場合には先読み処理を直感的に表現することができたが，一方で提案機構を用いないで同じ処理を実装すると図 4(b) のようになる．処理の内容としてはまったく同じことを行っているのだが，概観だけを見るとかなり異なっており，また単純に行数を比較すると提案機構を用いた場合のおおむね 2 倍の量になっている．この違いの原因は，記事を取得する関数 `getArticle` がコールバック関数の呼び出しによって結果を返すようになっていることである．2.2 節で述べたとおり，従来のシングルスレッド環境の JavaScript プログラムではサーバとの通信をイベント駆動型で行う．そのため基本的に関数呼出しは，通信のあとで行うべき処理をコールバック関数にまとめ，それを引数として渡す形をとる．これは `getArticle` だけではなく，それを利用している `getArticleCache` や `backgroundLoad`，またそれらを利用する他の関数も同様である．`backgroundLoad` の実装を一見すると，図 4(a) で for 文で表現されていた配列に対する繰り返しが見て取れないが，こちらでは内部で定義した関数 `l` を `backgroundLoad` へコールバック関数として渡して再帰的に呼び出すことで，ループ構造をハンドコーディングしている．これは従来の JavaScript ではループ文の途中で実行を中断するようなことができないためである．このように制御フローを自分でコーディングしなくてはならないため，非常に複雑な記述を要求される．

従来記述のもうひとつの問題は，関数の内部で通信を行うと，それが関数のインタフェースにコールバック関数という形で現れてしまうことである．ちょうど `getArticle`，`getArticleCache`，そして `backgroundLoad` がそうであったように，内部で通信を行う関数を使用すると，使用している関数もコールバック関数を受け取るように変更しなければならない．これはプログラムのモジュール化の上で問題となるのは言うまでもない．

提案機構とその通信ライブラリを併せて用いてプログラムを記述することにより，これらの問題を解消することができている．

4.1.2 JSON-RPC

JavaScript とサーバで通信を行うために広く使われているプロトコルとして JSON-RPC[4] がある．JSON-RPC は遠隔手続き呼出し (RPC) の仕様のひとつであるが，通信のデータフォーマットとして JavaScript で使うのに都合の良いよう設計されている JSON[1] を用いるのが特徴である．JSON-RPC の実装にはサーバとの通信が不可欠なため，通常は JSON-RPC 用のライブラリはイベント駆動型のインターフェースを持つように設計せざるを得ない．しかし本来 RPC が，通信を手続き呼出しの形で抽象化したものであることを考えれば，引数を与えると結果が戻り値として返ってくるという形が自然である．またイベント駆動型であると，前節で述べた問題も避けられない．

```
rproc = Concurrent.Thread.JSON.RPC.bind({
  url   : "srv/JSON-RPC",
  method: "rproc"
});

try {
  var r = rproc();
  alert(r);
} catch ( e ) {
  alert("ERROR: " + e);
}
```

(a) 提案機構を用いた記述

```
jsonrpc=new JSONRpcClient("srv/JSON-RPC");

jsonrpc.srv.rproc(function(r, e){
  if (e) {
    alert("ERROR: " + e.message);
  } else {
    alert(r);
  }
});
```

(b) JSON-RPC-Java を用いた記述

図 5 JSON-RPC の使用例

提案機構は専用の通信ライブラリの上に組み上げられた上位ライブラリとして，JSON-RPC ライブラリを提供している．提案機構を用いた場合には，JSON-RPC でアクセスできるサーバ上の手続きを JavaScript の関数にそのまま対応付けさせることができる．**図 5**(a) に使用例を示す．まず 1 行目から 3 行目にかけて，提案機構の提供する `Concurrent.Thread.JSON.RPC.bind` によって RPC の対応付けを行う．`bind` に RPC を提供する URL や手続き名の情報を引数として渡すと，結果として関数が返る．あとはこの返された関数を呼び出すことで，RPC によってサーバ上の手続きをあたかも通常の JavaScript 関数であるかのように呼び出すことができる．サーバ側で手続きを実行した結果例外が発生した場合には JavaScript の上でも例外の発生によって通知されるため，try-catch 構文によって例外を捕捉することができる．一方，同様の処理を既存の JSON-RPC 用フレームワークである JSON-RPC-Java[7] を用いて記述すると図 5(b) のようになる．1 行目で RPC

```
function getUserName ( ) {
  if ( !login_info )
    login_info = loginPrompt();
  return login_info.userName;
}
```

図 6 getUserName の実装例

の対応付けを行い，3 行目が実際に RPC を行っている箇所である．見てわかるように，引数としてコールバック関数を渡し，遠隔手続きの結果はコールバック関数で受け取るようイベント駆動型で記述されている．コールバック関数には，第 1 引数として遠隔手続きの戻り値が渡され，第 2 引数にはサーバ側で例外が発生したときにのみ例外を表すオブジェクトが渡される．こちらの方法では，例外処理のために JavaScript 言語が提供する try-catch 構文の支援を受けられないため，サーバ側で例外が発生したかどうかを毎回手動でチェックしなければならないことがわかる．また，提案機構では catch 節で捕捉されない例外は呼出し元へと伝播していくため例外処理を呼出し元にまかせることができるが，JSON-RPC-Java を用いた場合には例外を伝播させることができない．しかも，このようなイベント駆動型のプログラムの場合に，呼出し元へ例外の発生を通知することは一般には容易ではない．

4.2 ユーザ入力

前節まで，イベント駆動型の非同期通信が制御フロー記述を複雑にしてしまうという点を問題として取り上げてきたが，同様の問題は ユーザからの入力を受け取る場合についても言える．Ajax アプリケーションは Web ブラウザで動作する GUI アプリケーションであり，これは非同期通信と同様にイベント駆動型で記述される．そのため必然的に，ユーザからの入力はイベント駆動型のプログラミングによって受け付けることになる．多くの場合はこれで問題はないが，やはり問題となるケースもある．

例としてログイン機能のある Web アプリケーションで，ユーザ名を取得する関数 getUserName を考える．ユーザがすでにログインしている場合は現在のユーザ名をすぐに返せばよいが，そうでない場合にはログイン用の入力フォームを表示して，ユーザ名とパスワードを入力させることとしよう．このような関数は**図 6** のように実装できると考えられる．ここで，関数 loginPrompt はログインフォームを表示してユーザに入力を促し，ログイン情報を結果として返すものとする．しかしこの loginPrompt は，どのようにして実装すべきだろうか．イベント駆動型の GUI プ

```
function loginPrompt ( ) {
  var form=document.createElement("FORM");
  form.innerHTML
    = 'Name: '
    + '<input type="text" name="name">'
    + 'Password: '
    + '<input type="password" name="pass">'
    + '<input type="button"'
    + '       name="ok" value="OK">';
  document.body.appendChild(form);
  Concurrent.Thread.waitFor(form.ok,
                            "click");
  return authenticate(form.name.value,
                      form.pass.value);
}
```

図 7 loginPrompt の実装例

ログラミングでは，ユーザが入力できるようにいったん UI スレッドを解放しなければならないため，従来の JavaScript ではこのような関数を実装することはできなかった．提案機構はこのような用途のために，関数 waitFor を提供している．これは指定した GUI 部品に特定のイベントが発生するのを待機するためのメソッドであり，イベントが発生するまで現在のスレッドの実行を中断させる．waitFor を用いて loginPrompt を実装した例が **図 7** である．waitFor は第 1 引数に監視対象にする GUI 部品を，第 2 引数にイベント種別をとる．図 7 では，フォーム中の「OK」と書かれたボタンがクリックされるまで待機することを意味する．また，プログラム中で使われている関数 authenticate は，ユーザ名とパスワードを引数にとり，認証をおこなった上でログイン情報を返す関数であるとする．

このようにマルチスレッドが利用可能になったことで，ある処理が 1 つのスレッドをブロックしたとしても他のスレッドが処理を続行することができ，そのためユーザ入力を途中に含むような処理であっても直感的な形で抽象化・モジュール化が可能となった．これは従来の JavaScript では不可能であったことである．

4.3 ループによる実装

通常，プログラムの多くの部分はループ構造となっているため，ループ構文はプログラミングにおいて重要である．しかしながら，従来の JavaScript プログラミングに関して言えばループは好ましくない存在であるといえる．これは 2.1 節で述べたように，JavaScript では長い時間を必要とする処理はユーザ応答性の妨げとなり，ループ構文はその代表ともいえる存在だからである．提案機構はプリエンプティブなマルチスレッド環境を提供するため，長い時間を必要とするループ

```
function addElement ( el ) {
  while ( !document.body )
    Concurrent.Thread.yield();
  document.body.appendChild(el);
}
```

図 8 Busy-wait の利用例

であってもユーザ応答性の心配なしに使うことができる．本節ではループを積極的に用いる例を取り上げる．

4.3.1 Busy-wait

長時間に渡るループを用いる例のひとつとして，busy-wait と呼ばれるパターンがある．これはプログラムがある状態になるまで待つというパターンで，通常は目的とする状態の条件式の論理否定を条件とする空の while 文を用いる．従来の JavaScript ではこのような処理は，一定周期でイベントを発生させる組み込み関数 `setInterval` を使うなどして実装することができた．しかしこの方法では一度 UI スレッドを解放する必要があるため，ループの途中や関数呼び出しの深くなった場所など，プログラムの実行が中断できない場所ではまったく利用することができなかった．これに対して提案機構では，ループ文を用いることでどこでもこのようなことが可能である．

例として，現在のページに HTML 要素を追加する単純な関数 `addElement` を考えてみる．HTML の body 要素は `document.body` で参照できるためこの値プロパティの `appendChild` メソッドを使用すれば良いのだが，body 要素は常に存在しているわけではない．つまり，ブラウザがサーバから HTML を読み込んでいる途中の状態で，body 要素に到達する前にはまだ要素を追加することはできないのである．そのため，`addElement` が呼び出されたときに body 要素が存在していなかった場合，body 要素がブラウザによって生成されるまで待つこととする．そのように実装した例が **図 8** である．ここではループの本体は空ではなく，提案機構の提供する `yield` メソッドを呼び出しているが，これは明示的にスレッドの実行権を譲るためのものであり，実質的には何も実行しないことに等しい．ただしループ本体を空にした場合，このスレッドが実行権を持っている一定時間くり返し同じ条件を確認するだけで無駄が多くなるため，くり返しのたびに他のスレッドに仕事をさせるようなっている．

5. おわりに

Ajax 開発のための JavaScript マルチスレッドフレームワークを開発した．提案機構は JavaScript のライブラリとしてすべて JavaScript 自身で実装されており，JavaScript においてプリエンプティブなマルチスレッド環境を実現することができる．提案機構は現在，JavaScript のフルセット言語をサポートしており，その実装はオープンソースとして公開されている．

また本発表では，提案機構の応用事例について調査を行い，それについて報告した．従来の Ajax 開発は多くの部分がイベント駆動型のプログラミングによってしか記述できなかったために，制御フロー記述が煩雑になってしまうという問題があったが，提案機構が提供するを用いた記述ではその問題の多くが解消されている．

今後の課題として，さらに応用範囲を広め，提案機構の上に実装すべき上位ライブラリの拡充が望まれると考えられる．

参 考 文 献

1) Crockford, D. and JSON.org: The application/json Media Type for JavaScript Object Notation (JSON), *Network Working Group RFC 4627* (2006). `http://www.ietf.org/rfc/rfc4627.txt`.
2) ECMA: *ECMAScript Language Specification*, third edition (2000).
3) Garrett, J. J.: Ajax: A New Approach to Web Applications (2005). `http://www.adaptivepath.com/publications/essays/archives/000385.php`.
4) JSON-RPC.ORG: JSON-RPC (2006). `http://json-rpc.org/wiki/`.
5) Li, P. and Zdancewic, S.: Advanced control flow in Java card programming, *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, New York, NY, USA, ACM Press, pp.165–174 (2004).
6) Loitsch, F.: Exceptional Continuations in JavaScript, *Proceedings of the 2007 Workshop on Scheme and Functional Programming*, pp. 37–46 (2007).
7) Metaparadigm Pte Ltd.: JSON-RPC-Java (2006). `http://oss.metaparadigm.com/jsonrpc/`.
8) von Behren, R., Condit, J. and Brewer, E.: Why events are a bad idea (for high-concurrency servers), *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, USA, USENIX Association, pp.19–24 (2003).
9) 牧　大介，岩崎英哉：非同期処理のための

JavaScript マルチスレッドフレームワーク，情報処理学会論文誌: プログラミング，Vol.48, No.SIG 12 (PRO 34), pp.1–18 (2007).