非同期処理のためのJavaScriptマルチスレッドフレームワーク

牧 大介[†] 岩崎英哉[††]

Ajax は Web 開発の世界で普及したが，一方で Ajax 開発が従来の Web 開発に比べて非常に困難であることがよく知られている．その理由として，Ajax 開発においては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あげられる．上の問題を解決するため，本論文では JavaScript のマルチスレッドライブラリを提案する．提供するライブラリの特徴としては，(1) 代表的な複数の Web ブラウザで可搬性があること，(2) プリエンプティブなスレッド切替えが可能であること，(3) オブジェクト指向で API を提供すること，がある．提案機構では，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を応用して既存の処理系で実行可能な JavaScript プログラムへと変換し，この変換済みプログラムを実行時ライブラリであるスレッド・スケジューラの上で並行実行する．そして実際に Ajax アプリケーションを記述することで，提案機構の有効性を確かめた．提案機構にはオーバヘッドがあるが，Ajax アプリケーションにおける通信遅延に比べると十分に小さいため，実用上は大きな問題にはならないと考えられる．

JavaScript Multithread Framework for Asynchronous Processing

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

Although Ajax is widely used in the development of Web applications, it is well known that Ajax development is much more difficult than traditional Web development. There are two reasons: (1) Ajax developers have to write complex asynchronous program on a single thread; (2) asynchronous communication on JavaScript can be programmed only in event driven style, which causes control-flow difficulty. To resolve this problem, we provide multithread library to JavaScript programmers. The proposed library has the following features: (1) it is portable among popular Web browsers; (2) it provides preemptive scheduling; (3) it provides object-oriented API. The proposed system converts JavaScript programs written in multithreaded-style into those in continuation-based style that are executable on existing systems, and then executes them concurrently on a runtime-library called thread-scheduler. To see the effectiveness of the library, we implemented an Ajax application using the library. The overhead of the converted programs is not a serious problem in practice because the overhead is smaller enough than communication delay of Ajax applications.

1. はじめに

Google Maps に代表される Ajax アプリケーションが広く認知されるようになったが，一方で Ajax アプリケーションの開発は非常に困難であることがよく知られている．その理由として第 1 に，1 つしかスレッドがない実行環境の上で高度なユーザインタラクションを求められるアプリケーションを実装しなければならないという問題がある．第 2 の理由として，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もあげられる．これらはどちらも，ノンプリエンプティブなシングルスレッド環境であるという JavaScript のスレッドモデルに起因している．

本研究ではこの問題を解決するために，JavaScript でマルチスレッドプログラミングを利用することを提案し，JavaScript にマルチスレッドプログラミング環境を提供するライブラリを実装した．このライブラリは次のような特徴を持っている．

- JavaScript の処理系にはいっさい手を加えておらず，JavaScript で記述したライブラリである．

† 電気通信大学大学院電気通信学研究科
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開発を行っている．

- 代表的な Web ブラウザ間での可搬性を有する．
- プリエンプティブなスレッド切替えが可能である．
- オブジェクト指向の API を持つ．

Ajax アプリケーションが普及した背景として，ブラウザ間の互換性が高まったことで様々なブラウザの上で実行可能になったということがある．これをふまえ，本提案機構の設計では，実際に Ajax 開発に利用できるようにするためにブラウザ間での可搬性に重点をおいた．その結果，特定の処理系に手を入れることはせず，コード変換を用いることで，既存の処理系で利用可能なライブラリとして実装している．

コード変換ではマルチスレッドのプログラミングスタイルで記述された JavaScript プログラムを，既存の処理系で実行できる通常の JavaScript プログラムへと書き換える．この書き換えでは継続ベースの並行処理[11),14)] を応用している．具体的な作業はプログラムを細かく分割してその間にコンテキストスイッチをするようなコードを挿入することであるが，その際，制御フローを明示的に保存するために継続渡し形式[1)] を，プリエンプティブなスケジューリングを実現するためにトランポリン[4)] を利用している．こうして変換されたプログラムは，複数のスレッドの上で並行に動作しているように振る舞う．また，変換されたプログラム自体が JavaScript の標準仕様[3)] に準じたプログラムであるため，既存の JavaScript 処理系で実行可能である．これにより特定のブラウザに依存しない可搬性を達成した．

本論文の構成は以下のとおり．まず 2 章で Ajax とその中核技術である JavaScript について簡単に説明しながら，Ajax 開発を困難にしている原因を明らかにする．続いて，3 章で提案機構であるマルチスレッドライブラリが満たすべき設計目標とその機能を，想定している使用法に基づきながら概説する．次に 4 章でその目標をどのようにして達成するのか，提案機構の実現方法について述べる．5 章で，実装した提案機構のプロトタイプを用いて実際に Ajax アプリケーションを記述することで，その有効性について議論する．6 章では，本研究の提案手法が参考にしている既存研究について，また他分野での関連する研究を紹介する．最後に，7 章で本論文をまとめ，今後の課題について述べる．

2. JavaScript と Ajax

2.1 JavaScript

JavaScript は Web ページ中で動的なコンテンツを記述するために開発されたオブジェクト指向言語である．そもそもは Netscape Navigator に組み込まれたスクリプト言語を指すものだが，本論文では Web ブラウザに組み込まれている ECMAScript 3rd edition[3)] に準ずる言語を指すものとする．

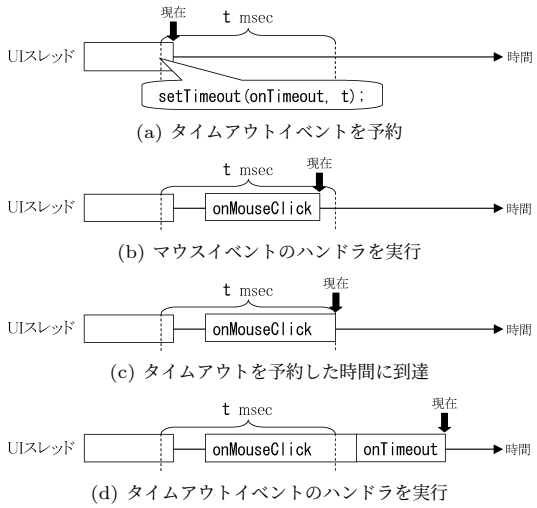

図 1 実行時の UI スレッドイメージ
Fig. 1 Image of runtime UI-thread.

Web ブラウザにおいては，すべての JavaScript の処理を 1 本のスレッドの上で行う．このスレッドは Web ブラウザのユーザインタフェースを構築しているスレッドで，UI スレッドと呼ばれる．通常 JavaScript プログラムは，マウスクリックなどのイベントに対応するイベントハンドラとして呼び出され，1 つずつ UI スレッドの上でノンプリエンプティブに実行される．また，イベントハンドラが実行されている間は他のイベントが割り込むことはできないような処理系となっている．

たとえば，タイムアウトイベントとマウスイベントが起こるような JavaScript プログラムを実行した場合の UI スレッドの状態を考えてみる．まずプログラムがタイムアウトイベントを予約していったん終了したところが **図 1** (a) である．JavaScript においてタイムアウトを設定するには組み込み関数 `setTimeout` を用いる．これは第 1 引数にイベントハンドラとして呼び出すべき関数，第 2 引数にタイムアウトまでの時間をミリ秒単位で指定し，予約されたタイマの識別子を表す整数値を返す．ここでタイムアウトの前に，ユーザのマウスクリックによってマウスクリックのイベントハンドラである `onMouseClick` が実行されると図 1 (b) のようになる．この `onMouseClick` が処理を終了するまでに，図 1 (c) のように，先に予約したタイムアウトの時間になってしまったとしても，本来

イベントが発生すべき時間にイベントは発行されない．これは上で述べたとおり，イベントハンドラが実行されている間は他のイベントを発行できないからである．実際にタイムアウトイベントが発行されるのは，図 1 (d) のように `onMouseClick` が処理を終えた後になってしまう．この例から分かるように，JavaScript でのイベントは Web ブラウザの内部的なキューに登録され，UI スレッドが空いているときに発行できるイベントを順次キューから取り出して発行させるような仕組みになっている．

UI スレッドがイベントハンドラを実行している間は他の処理が割り込むことができないという制約は，ユーザインタフェースの描画にも適用される．したがって，時間のかかる処理などが長時間にわたって UI スレッドを占有し続けていると，ユーザからはブラウザ自身がフリーズしたように見えてしまう．典型的な例は次のような無限ループである．

```
while ( 1 ) do_stuff();
```

このようなプログラムを実行すると，UI スレッドが解放されなくなり，ブラウザのフリーズを引き起こしてしまう．したがって JavaScript プログラミングの上では，長時間にわたり UI スレッドを占有しないように注意を払いながらプログラミングを行う必要がある．

しかし JavaScript は現在の処理を中断・再開するような命令を持っていない．そのため，UI スレッドを解放する方法は実行中のイベントハンドラからリターンして，完全に処理を終わらせることだけである．また，実行の再開はつねにイベントハンドラが呼び出されることによって，関数の先頭から始まる．このために，UI スレッドを適宜解放しながら実行する JavaScript プログラムを作成することは容易ではない．実際，途中で UI スレッドを解放する必要があるような処理ではループ文や関数呼び出しのような多くの制御構文を用いることができないため，プログラマは非常に複雑な記述を強いられることになる．この制御フロー記述の問題は，2.2.2 項で詳しく述べる．

2.2 Ajax

Ajax（Asynchronous JavaScript + XML）[5] アプリケーションとは主にサーバと非同期に通信を行うような Web アプリケーションのことを指す．

従来の Web アプリケーションでは，サーバとの通信ができるのは，基本的にページを遷移するときだけであり，ユーザによる入力の結果サーバとの通信が必要となれば，他のページへ遷移するか現在のページをリロードするなりして，ページ全体を読み直す必要があった．この手法では，ページを読み込んでいる間ユーザができることは，ただ待つことだけである．そのため，頻繁に通信を必要とする種のアプリケーションではユーザ応答性が著しく低くならざるをえなかった．

```
 1: var req = new XMLHttpRequest();
 2: req.open("GET", url, true);
 3: req.onreadystatechange = function () {
 4:   if (req.readyState == 4) {
 5:     if(req.status == 200)
 6:       document.write(req.responseText);
 7:     else
 8:       document.write("ERROR");
 9:   }
10: };
11: req.send(null);
```

図 2 XMLHttpRequest の使用例
Fig. 2 Example of use of XMLHttpRequest.

これに対し Ajax アプリケーションでは JavaScript プログラム中から Web サーバと通信する．このことで，ページ遷移を起こすことなく新たにデータを取得したり，送信したりすることが可能となり，高いユーザ応答性を持つ Web アプリケーションを作成することが可能となった．

2.2.1 JavaScript でのサーバとの通信

JavaScript から Web サーバと通信を行うには組み込みオブジェクトである XMLHttpRequest を用いる．基本的な使用法を**図 2** に示す．

2 行目の `open` メソッドの引数の意味はそれぞれ，HTTP リクエストのメソッド，URL，非同期通信を行うか否かを示すブール値である．ここで，第 3 引数は原則的に `true` しか使用されない．なぜならば 2.1 節で述べたように，JavaScript を実行している間は UI スレッドが占有されてしまうため，同期通信ではサーバからのレスポンスを待っている間，ブラウザがフリーズしたようになってしまうからである．そのようなアプリケーションはユーザビリティの観点から好ましくないので，実際には非同期通信しか使用されない．

次に 3 行目で `onreadystatechange` プロパティに関数を代入している．これにより XMLHttpRequest オブジェクトの readystatechange イベントに対応するハンドラが登録される．このイベントは非同期通信を行っている間に，オブジェクトの状態が変わるたびに発行される．オブジェクトの現在の状態は `readyState` プロパティで知ることができ，その値は以下のように割り当てられている[9]．

0：Uninitialized
1：Open
2：Sent
3：Receiving
4：Loaded

最後に，11 行目にあるように send メソッドを呼び出すことでサーバにリクエストを送信する．send の引数は HTTP リクエストのボディだが，HTTP プロトコルの GET メソッドではリクエストボディが存在しないため，この場合は null を渡している．この後，オブジェクトの状態が変わるたびに onreadystatechange に設定したハンドラ関数が呼び出されることになる．上のプログラムではオブジェクトの状態が 4，すなわち読み込み終了（Loaded）になるまで待ち続け，読み込みが終了したらレスポンスステータスの値を確認し，正常なレスポンスであれば読み込んだ内容を，さもなければエラーを示す文字列を表示するようになっている．

2.2.2 Ajax プログラミングの問題点

図 2 に示したように，Ajax アプリケーションにおける非同期通信はイベント駆動型で記述されている．これは，通信の結果を得るためには一度実行中のイベントハンドラを終了しなければならないということを意味する．実用的なアプリケーションプログラムでは 1 つの仕事を完了するために何度も通信を行う必要があるため，この方法では通信の前後でプログラムが分断されてしまうことになる．

簡単な例を使って説明する．サーバからデータを取得して，それに何らかの処理を施したうえで，サーバに送り返すプログラムを考える．これは一連の処理であるが，実装すると**図 3** (a) のようなコードになってしまう．このコードを見ても，どのような順番でコードが実行されるのか，また何をするプログラムなのかをすぐに理解するのは難しい．実際にはまず 1〜4 行目を実行して一度プログラムを終了（UI スレッドを解放）．イベント通知によって 6 行目の handler1 が実行され，13 行目でまた終了．最後に 20 行目の handler2 が実行される．

しかし，同じプログラムを同期通信を用いると図 3 (b) のようにはるかに簡潔に記述でき，またその内容は容易に理解できるであろう．これは，通信の前後でプログラムが分断されていないことと，JavaScript 言語が提供している構文による恩恵が直接的に得られているからである．しかし 2.1 節で述べたように，JavaScript のスレッドモデルの制約のために，このように書くことは現実の Ajax 開発では許容されない．

```
 1: var req = new XMLHttpRequest();
 2: req.open("GET", url, true);
 3: req.onreadystatechange=handler1;
 4: req.send(null);
 5:
 6: function handler1 () {
 7:   if (req.readyState == 4) {
 8:     if(req.status == 200) {
 9:       var text = req.responseText;
10:       text に変更を加える
11:       req.open("POST", url, true);
12:       req.onreadystatechange=handler2;
13:       req.send(text);
14:     } else {
15:       onError("ERROR: can't GET");
16:     }
17:   }
18: }
19:
20: function handler2 () {
21:   if (req.readyState == 4) {
22:     if (req.status == 200) {
23:       document.write("OK");
24:     } else {
25:       onError("ERROR: can't POST");
26:     }
27:   }
28: }
29:
30: function onError () {
31:   document.write("ERROR");
32: }
```

(a) 非同期通信を用いた記述

```
try {
  var req = new XMLHttpRequest();
  req.open("GET", url, false);
  req.send(null);
  if (req.status != 200) throw "ERROR";
  text に変更を加える
  req.open("POST", url, false);
  req.send(text);
  if (req.status != 200) throw "ERROR";
  document.write("OK");
} catch (e) {
    document.write(e);
}
```

(b) 同期通信を用いた記述

図 3 サーバとデータをやりとりするプログラム
Fig. 3 Program that communicates with a server.

また，Ajax 開発ではモジュール化の一般的な方法である関数による抽象化が難しいという問題をかかえている．Ajax を利用したユーザ認証を行う手続き authenticate を考えてみる．これはユーザ名とパスワードを引数にとり，正しい組合せであれば true，そうでなければ false を返すような関数として抽象化で

きると考えられよう．しかし，いざ認証の処理をサーバ側で行おうとすると，このインタフェースでは不都合が生じる．認証手続きの中でサーバとの通信が必要ということは，イベント駆動型による非同期通信を行うということである．つまり，認証の結果をサーバから受け取るためには一度プログラムを終了する必要があるので，結果を `authenticate` の戻り値として返すことはできない．代わりに，非同期通信が終わった後に呼び出されるべき関数 `callback` を指定して，認証の結果はこの関数に渡されるようにしなければならない．したがって，`authenticate` のインタフェースは次のようにしなければならない．

```
authenticate (name, pass, callback)
```

このように，内部の実装の変更が外から見えるインタフェースに影響を及ぼしてしまうということは，モジュール化のうえで大きな問題である．

これらの問題はどちらも，通信の前後でプログラムが分断されてしまうことによって引き起こされている．通信のたびにプログラムを終了する必要があるということは，ループ文，`try-catch` などの構文構造，関数呼び出しなど，JavaScript 言語で提供されている機能を用いて制御フローを記述することができないということである．この制御フロー問題が，Ajax 開発の複雑化を招いている．これは JavaScript 上の通信がイベント駆動型であること，ひいては JavaScript のスレッドモデルに原因がある．重要なのは，この問題がイベント駆動型のプログラミングによっている限り回避できないということである．イベント駆動型である非同期通信機能のプリミティブの上にどのような巧妙なラッパをかぶせようとも，この本質的な問題の解決にはならないのである．

3. 提 案 手 法

3.1 マルチスレッドプログラミング

前章で述べたように，非同期通信の記述にイベント駆動型のプログラミングを用いている限り，Ajax プログラミングの複雑さを避けて通ることはできない．そもそも Ajax の複雑さは 1 本しかない UI スレッドの上ですべての非同期処理を記述する点にその原因があった．サーバからの返答を待っている間にも処理を進めることができるよう複数のスレッドがあれば，記述性とユーザ応答性の両方を損なうことなくプログラミングを行うことができる．

そこで本論文では，Ajax プログラミングの複雑さを解消するため，イベント駆動型プログラミングではなくマルチスレッドプログラミングにより非同期処理を記述する方法を提案する．そのため，JavaScript にマルチスレッドプログラミング環境を提供することを目指した．

3.2 設 計

マルチスレッドを提供するにあたっての基本的方針として，本提案機構を JavaScript 上のライブラリとして提供することとした．つまり，Web ブラウザの実装に手を入れず，代表的な Web ブラウザ間での可搬性を維持することを目標として掲げている．なぜならば，本研究が対象としている Ajax が普及した背景の 1 つに，Web ブラウザ間の互換性が高まったことがあるからである．特定の Web ブラウザだけでマルチスレッドが利用可能になったとしても，実際に Ajax 開発に活かせないのでは意味がない．

ライブラリとして実現するために，基本的には UI スレッドをタイムシェアリングすることでマルチスレッドを提供することを考える．機能としては，プリエンプティブなスレッド切替えと，オブジェクト指向 API を提供することを目指した．また，4 章で詳しく述べるが，本提案機構はマルチスレッドを実現するためにコード変換を用いる．JavaScript では実行時に生成したコードやサーバから非同期通信で取得したコード片を組み込み関数 `eval` で評価することが行われるため，これら実行時に得られたコードもマルチスレッドで実行できるよう，コード変換を実行時に行えるように設計を行った．

JavaScript において関数は Function オブジェクトとして表現されるが，あらゆるオブジェクトがそうであるように Function オブジェクトもまた文字列化，つまり `toString` メソッドを実装している．Function オブジェクトの `toString` メソッドのデフォルトの実装は，その関数を実装しているコードを返すように標準仕様[3] で定められている．そのため実行時の JavaScript プログラムは，自分自身のソースコードを関数単位で文字列として手に入れることができる．これを利用し，コード変換は関数を単位として行うこととした．

実行時のコード変換は，次節で述べる `create` メソッド，あるいは `compile` メソッドに引数として渡された関数のみを行い，コード変換対象の関数が呼んでいる関数をさらに芋づる式に変換することはしない．その第 1 の理由としてはまず，組み込み関数として提供される関数（たとえば入出力関数など）は JavaScript では記述することができないため，そもそもコード変換ができないからである．また第 2 の理由として，5 章で詳述するように，本提案機構のコード変換には

表 1 公開メソッド一覧
Table 1 List of public methods.

static メソッド	
`compile(f)`	関数 f をコード変換する.
`create(f, a1, a2, ...)`	関数 f を新しいスレッドの上で実行する.
`self()`	現在のスレッドオブジェクトへの参照を返す.
`sleep(t)`	現在のスレッドを t ミリ秒休眠させる.
`stop()`	現在のスレッドを無期限で中断させる.
`yield()`	他のスレッドへ処理を切り替える.
`Http.get(u)`	URL u の内容を HTTP プロトコルの GET メソッドで取得する.
`Http.post(u, b)`	URL u に対して HTTP プロトコルの POST メソッドで文字列 b を送信し，レスポンスボディを返す.
instance メソッド	
`join()`	対象スレッドが終了するまで現在のスレッドを中断させる.
`kill()`	対象スレッドを終了させる.
`notify(e)`	対象スレッドの上に例外 e を発生させる.

大きなオーバヘッドが存在するため，必要な箇所のみをコード変換することでプログラム全体のオーバヘッドを抑えることができるからである.

実行時コード変換だけではなく，実行前にあらかじめ必要箇所のコードを変換しておくこともできる．変換の内容は実行時の変換とまったく同じなので，コード変換が必要な箇所があらかじめ分かっているのであれば，実行前に変換しておくことで実行時の CPU 時間を節約できる.

3.3 API

本提案機構がライブラリとして提供する API の一覧を**表 1** に示す．これらはすべて Thread コンストラクタに定義されている．ここで `Thread.Http.get` と `Thread.Http.post` は UI スレッドをブロックしない通信用 API で，2.2.1 項で述べた XMLHttpRequest のかわりに使うものである.

以下に実際に提案ライブラリを用いて記述したプログラムを見ながら，API と提供している機能について簡単に述べる．例として**図 4** のプログラムを用いる．これは指定した URL からデータを取得して，その内容を表示するだけの単純なプログラムである．ただし，サーバと通信を行っている間，ブラウザがフリーズしていないことを示すために，ステータスバーに簡単なアニメーションを表示する.

このプログラムではまず，2 つの関数を定義している．1 つめの `load` はサーバからデータを取得して表示する関数，もう一方の `nowLoading` はアニメーションを表示する関数で，それぞれが別々のスレッドの上で実行される.

22 行目が，関数 `load` を別のスレッドで実行するように呼び出している部分である．`Thread.create` は第 1 引数に指定された関数をコード変換し，新しく作成したスレッドの上で変換された関数を実行するメソッドである．残りの引数はそのまま変換された関数に渡される．`load` も最初に `Thread.create` を用いて，`nowLoading` を別スレッドで呼び出している（2 行目）．これで `load` と `nowLoading` が並行に動作するようになる.

```
 1: function load (url) {
 2:   var th = Thread.create(nowLoading);
 3:   try {
 4:     var res = Thread.Http.get(url);
 5:     document.write(res.responseText);
 6:   } catch (e) {
 7:     document.write("ERROR: " + e);
 8:   }
 9:   th.kill();
10: }
11:
12: function nowLoading () {
13:   var bar = ["|", "/", "-", "\\"];
14:   var i = 0;
15:   while ( true ) {
16:     window.status = "Now loading..."
17:                   + bar[i=(i+1)%4];
18:     Thread.sleep(125);
19:   }
20: }
21:
22: Thread.create(load, "http://...");
```

図 4 提案ライブラリを用いた記述例
Fig. 4 Example of use of the proposed system.

`Thread.create` の戻り値は新しく作成されたスレッドを表すスレッドオブジェクトで，以降このスレッドに対する操作はこのオブジェクトを通じて行う．たとえば，任意のスレッドに例外を発生させるメソッド `notify` がある．例外は発生したスレッドで捕捉し処理することもできるが，捕捉されなかった場合にはそのスレッドの実行を終了させる．9 行目で使用されている `kill` メソッドは，次のコードの省略形である.

```
th.notify(new Thread.KillException());
```

4 行目の Thread.Http.get によりサーバからデータを取得しているが，Thread.Http.get 関数から処理が帰ってくるのは通信がすべて終わったあとである．通信の間このスレッドの処理は一時的に停止したように見えるが，その間も他のスレッドは動き続けることができる．そのため，ユーザ応答性を下げる心配はない．また，通信中に発生した例外（通信エラーなど）は 6 行目の catch 節で捕捉されるようになっている．このように JavaScript が提供する基本的な構文構造を用いて制御フローが記述できるため，2.2.2 項であげた複雑さを緩和することができている．

関数 nowLoading の内容は，基本的に while 文による無限ループで，その中でアニメーションを描画している．この関数は自身では処理を終了することができないので，他のスレッドから notify メソッドにより例外を発生させられることで終了させられることを期待している．

アニメーションの再描画の間隔をあけるために，18 行目では Thread.sleep メソッドを使用している．これは現在のスレッドを一時的に休眠させるメソッドで，休眠時間をミリ秒単位で指定する．休眠されている間にも，notify メソッドを用いて例外を発生させることができる．この場合には，タイムアウトを待たずに即座に実行が再開される．

Thread.sleep と同種のメソッドとして，Thread.stop もある．これは現在のスレッドの実行を中断させるが，こちらの場合にはタイムアウトにより実行が再開されることはなく，もっぱら例外が発生するのを待つのみである．したがってこれは，次のような使用法を想定している．

```
try {
  Thread.stop();
} catch (e) {
  ここから実行が再開される
}
```

現在のところ，本提案機構は排他制御機能を提供してはいない．その理由は，多くの Ajax アプリケーションでは JavaScript はユーザインタフェースのみを担当しており，排他制御を必要とするような主たる処理のほとんどはサーバ側で実行されているのが現状であることを考えると，実際の利用においては大きな問題になるケースは少ないと判断したためである．そのため，プログラマは必要ならばユーザレベルで排他制御を記述しなければならない．しかし，排他制御機能の提供は本ライブラリの趣旨として必須であり，今後の課題である．

4. 実現方法

4.1 概要と構成

本提案手法を，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する．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ごと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．これを実現するにあたって，関数型言語の世界で利用されてきた，継続ベースのマルチスレッディングを応用する．継続[1)]とはいわば「残りの計算」を表す関数で，プログラムのある時点で，そこから後に実行されるべき処理を関数の形で抽象化したものである．非常に簡単な例を示すと，**図 5** (a) のプログラムはその意味を変えずに図 5 (b) のように書き換えることができる☆．

図 5 (a) のプログラムの (*) の時点における継続は，書き換え後の関数 rest に相当する．ここで (*) の時点で実行を中断し，後で再開したいとしよう．すると，rest を保存しておいて，再開したいときにそれを呼び出せばよいことが分かる．つまり，スレッドのコンテキストの保存は継続の保存，コンテキストスイッチは保存しておいた継続の呼び出しに対応する．あとは，適切な方法で継続（この場合は rest）を呼び出し元に返して保存する方法，また保存しておいた継続を適切なタイミングと適切な方法で呼び出すことを考えればよい．本提案手法では前者を主にコード変

```
function f ( ) {
  var c = getc();
  // (*)
  document.write(c, "\n");
}
```

(a) 元のプログラム

```
function f ( ) {
  var c = getc();
  function rest ( ) {
    document.write(c, "\n");
  }
  rest();
}
```

(b) 書き換えた結果

図 5 継続を用いた書き換えのイメージ
Fig. 5 Rewriting with continuation.

☆ 図 5 は書き換えのイメージを示すための例であり，実際の書き換えは以降で詳述するように，もっと複雑である．

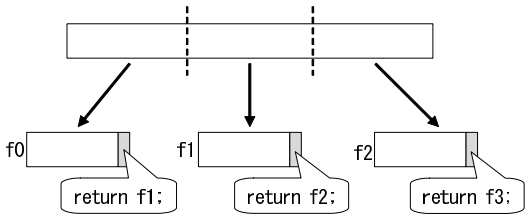

図 6 コード変換イメージ
Fig. 6 Image of code conversion.

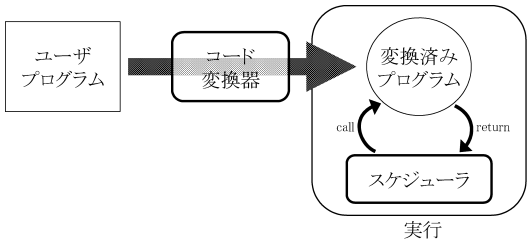

図 7 提案機構概観
Fig. 7 System overview.

換器を，後者をスレッド・スケジューラを実装することで実現する．コード変換を利用してマルチスレッドを実現することで，JavaScript 処理系にはまったく手を入れることなく，すなわち既存の処理系とも互換性を保ったままマルチスレッド・プログラミングを利用可能にするという目的を達成することができる．

コード変換器は通常の JavaScript プログラムを基本ブロック単位の複数の関数に分割し，少し処理を進めては継続を返すという形に書き換える．いい換えれば，**図 6** のようにプログラムを細切れにしてそのそれぞれを関数とし，関数の最後（すなわち，細切れに切った境目）に次に実行する関数（実際は 4.2 節で述べるコンテキストオブジェクト）を返すようなコードを挿入するのである．こうして細切れにされた関数はスケジューラによって呼び出される．スケジューラは細切れになった関数を呼び出し，返された関数を再び呼び出す，といった処理を繰り返し，プリエンプションなどによりコンテキストを切り替えるときには，返された関数をあとで呼び出すようなイベントハンドラを登録して，UI スレッドを解放する．こうすることで，上で述べたような，基本ブロックの実行を進めて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ことが実現される．コード変換器はプログラムをその意味を変えずに，スケジューラが期待しているような形へと変形させるのである．提案機構全体は，**図 7** に示したような構成となっている．

4.2 コンテキストオブジェクトと継続オブジェクト

実行中のプログラムのコンテキストとしてスケジューラが受け取るべきものを考えると，細切れにされた関数が単に次に実行すべき関数を返すだけでは，3.2 節で述べた設計目標を達成することはできない．このほかに，`sleep` メソッドのようにスレッドの中断を通知する手段，また，次の継続を呼び出す際に引数として渡す値が必要である．このため，関数の戻り値としてつねに次のような 3 つのプロパティを持つオブジェクトを返すこととする．

```
{
  continuation: 継続,
  timeout     : 再開までの待ち時間,
  ret_val     : 継続の実引数
}
```

以降このオブジェクトを「コンテキストオブジェクト」と呼ぶ．`timeout` プロパティには中断する時間をミリ秒単位で指定するが，`stop` メソッドのように時間により再開されることがない場合には，負数でそれを表す．また，タイムスライスという一定の時間を導入し，スレッドを切り替える必要がない場合にその時間内であれば，UI スレッドを解放せずに実行を続けることとした．スレッドを切り替える必要がないことは，`timeout` プロパティに undefined 値を設定することで示される．

一方で，JavaScript における「残りの計算」，すなわち継続を表現するにも関数だけでは足りない点がある．これは次の 2 つである．

- `this` の値
- 例外ハンドラ

JavaScript での関数（メソッド）コード実行中の `this` 値は，関数を呼び出すごとに決定される．そのため継続を表す関数を呼び出す際に `this` が参照する値を覚えておかなければならない．また，通常の継続とは別に例外発生時の継続を記憶しておく必要がある．それは 3.3 節で述べたように，`notify` メソッドを用いることで，スレッドの外部から例外を発生させることができるため，例外を発生させたスレッドが例外ハンドラにアクセスできるようにしておかなければならないためである．

以上の検討により，JavaScript における継続を，次のようなプロパティを持つオブジェクトとして表現する．

```
{
  procedure: 残りの計算処理を表す関数,
  this_val : 関数の中での this 値,
  exception: 例外発生時の継続
}
```

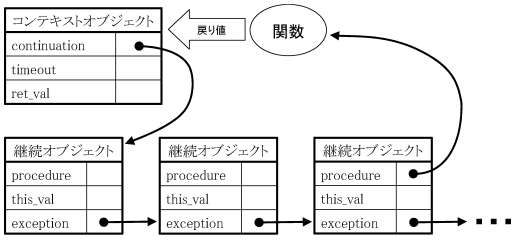

図 8 オブジェクトの関係
Fig. 8 Relationship of objects.

以降これを「継続オブジェクト」と呼ぶ．procedure プロパティの値は 1 引数関数で，呼び出すとコンテキストオブジェクトを返さなければならない．exception プロパティは継続オブジェクトを指しているので，exception プロパティによる継続オブジェクトの連鎖はリスト構造をなす．実行時の視点から見ると，このリストによって動的スコープを持つ try-catch の入れ子構造が表現されていることとなる．したがって，継続オブジェクトとコンテキストオブジェクトの関係は，**図 8** のようになっている．

4.3 スケジューラ

4.3.1 トランポリンの利用

スケジューラの主な仕事はコンテキストオブジェクトを受け取って，そこに保存されている継続を繰り返し呼び出すことであるため，処理の中心となるのは次のようなループである．

```
do {
  try {
    context =
    context.continuation.procedure.call(
      context.continuation.this_val,
      context.ret_val
    );
  } catch (e) {
    例外処理
  }
} while (context.timeout == undefined
         && タイムスライスが残っている);
```

これはトランポリン[4] と呼ばれ☆，コンパイラの実装でたびたび用いられる手法である[6),7),13),15)]．コンテキストスイッチのためにこのトランポリンを抜けた後もなお呼び出すべき継続が残っている場合には，そのスレッドの実行はまだ終了していないので，実行再開に必要なコンテキストをどこかに保存しなければならない．スレッドのコンテキストはコンテキストオブジェクトにまとめられているので，コンテキストの保存はコンテキストオブジェクトを保持しておくだけでよい．問題は，あとで再度トランポリンを実行するように実行再開を予約する方法である．コンテキストスイッチの際には一度，このトランポリンを実行していたイベントを完全に終了させ UI スレッドを解放しなくてはならないため，単純にトランポリンをさらにループ文でくるんで，ラウンドロビンを実装するような方法をとることはできない．

提案機構では，スケジューラの実行を再開するのに JavaScript の組み込み関数である setTimeout を用いている．これはユーザからのインタラクションなしにイベントを発生させることができるため，今回の用途に適している．2.1 節で述べたように，setTimeout によるイベントはブラウザ内部のイベントキューに蓄積され，UI スレッドが解放されるたびにキューの先頭から発行できるイベントが 1 つ取り出され発行される（その結果，発行されたイベントのイベントハンドラが UI スレッド上で実行される）ので，スケジューリングを行っていることに相当する．そのため実際のスケジューリング処理は，各スレッドに 1 つずつ対応するようイベントをキューに登録することで，setTimeout 関数の内部処理にすべて任せている．したがってスレッドオブジェクトは，setTimeout 関数の戻り値であるタイマを識別する整数値とコンテキストオブジェクトからなる，次のようなデータ構造である．

```
{
  timerID: タイマ識別子,
  context: コンテキストオブジェクト
}
```

イベントの予約は，次に示すスレッドオブジェクトのユーザ非公開メソッド standBy によって行われる．

```
function standBy ( t ) {
  // すでにイベントがキューに登録されていたら
  // 重複しないようにクリアする
  if (this.timerID !== undefined)
    clearTimeout(this.timerID);
  var self = this;
  this.timerID = setTimeout(
    function(){ self.doNext(); },
    t
  );
}
```

ここで doNext は，トランポリンを実装しているメソッドである．したがって現在の実装におけるスレッ

☆ ディスパッチ・ループ[13)]，あるいは UUO ハンドラ[6)] とも呼ばれる．

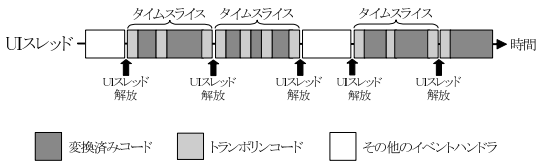

図 9 提案機構実行時の UI スレッドイメージ
Fig. 9 Image of UI-thread when using provided system.

ドの実体は，スレッドの残りの計算処理を保存しているコンテキストオブジェクトと，スレッドの再開イベントを予約しているタイマの組である．タイマがタイムアウトイベントを通知すると，イベントハンドラでは対応するコンテキストオブジェクトを用いてトランポリンのループを開始する．

たとえば，タイムスライスの消費によるプリエンプションの際には，トランポリンの繰返しを抜けた後に `standBy(0)` を実行する．ここで `standBy` の実引数が 0 ということは，0 ミリ秒後，すなわちなるべく早くタイマイベントを起こすようイベントのキューに登録することを意味する．実際には前述のように，キュー中のイベントは UI スレッドが空き次第順番に 1 つずつしか取り出されないので，これによりラウンドロビン方式のスケジューリングが達成されることになる．

提案機構における UI スレッドの実行イメージを**図 9** に示す．図のように，タイムスライスが消費されるたびにトランポリンコードが UI スレッドを明示的に解放することで，マウスクリックなどのその他のイベントハンドラはタイムスライスとタイムスライスの隙間で実行される．これにより，大きな遅延を起こさずにユーザに応答することができる．

4.3.2 休眠と例外発生

現在のスレッドを休眠させるメソッド `sleep` は，単に引数で与えられた休眠時間をそのまま `timeout` プロパティに設定したコンテキストオブジェクトを返すようなメソッドとして，次のように定義できる．

```
function sleep ( $this, $args, $cont ) {
  return {continuation: $cont,
          ret_val     : undefined,
          timeout     : $args[0]  };
}
```

この関数の戻り値をスケジューラが受け取ると，トランポリンループを抜けて `timeout` プロパティに設定された休眠時間を実引数として `standBy` を呼び出す．これによりスレッドの休眠を実現している．ここで現れた `$cont` や `$args` といった特殊な引数については，次の 4.4 節で詳述する．

他スレッドに例外を発生させる `notify` メソッドでも，同様に `standBy` を用いる．`notify` の場合には `sleep` や `stop` メソッドで休眠状態にあるスレッドをただちに起こす必要があるので，予約されているタイマがあればそれを中止して，スレッドが保存しているコンテキストオブジェクトの `continuation` プロパティの値を例外発生時の継続に設定し直してから，`standBy` を 0 を引数として与えて呼び出す．

```
function notify ( e ) {
  this.context.continuation
    = this.context.continuation.exception;
  this.context.ret_val = e;
  this.standBy(0);
}
```

これで対象スレッド（`this`）の実行が，例外ハンドラ（catch 節）に対応する継続から再開されるようにイベントが予約される．発生した例外の値は `ret_val` プロパティを経て，継続オブジェクトの `procedure` プロパティが指している関数へと引数として渡される．

以上のように処理系で提供されているイベント処理機構を積極的に利用することにより，スレッドを再開するタイミングやスケジューリングの問題は，タイムアウトイベントを通じてすべて処理系のイベント処理の上に転嫁することができる．また，処理系のイベント処理機構を利用するので，スレッド実行中にユーザのインタラクションにより起こされたマウスイベントやキーイベントの割当ても，同じ枠組みの中でスケジューリングすることができるという利点がある．このような実装は，簡潔さと処理系の機能を利用できるという利点があるが，今後スレッドに優先順位をつけるなどより細かいスケジューリングが必要になれば再検討が必要であろう．

繰返し継続を呼び出していく過程でスレッドの仕事がすべて終了した場合，NoContinuationException の発生によってこのことが通知される．したがって，それ以上計算すべきことが残っていないような状況では，継続オブジェクトの `procedure` プロパティには次のような関数が設定されている．

```
function ( r ) {
  throw new NoContinuationException(r);
}
```

こうして投げられた例外をトランポリンの catch 節が捕捉することで，そのスレッドの仕事は完了する．

4.4 コード変換器

コード変換器は JavaScript プログラムのソースを入力とし，スケジューラの期待するような形のプログラムのソースを出力するプログラムである．3.2 節で

述べたように，コード変換は Function オブジェクトの性質を利用して関数単位で行う．そして変換した結果のソースコードを eval で評価することで，スケジューラが期待するような動作をする関数を実行時に得ることができる．

変換された関数は，スケジューラの期待に沿って現在のコンテキストとしてコンテキストオブジェクトを返さなければならない．そのため，この関数を呼び出す際には継続オブジェクトを渡す必要がある．これは関数からの戻り先を保持しておくためで，このようなスタイルを継続渡し形式[1] と呼ぶ．さらに，this 値も引数として明示的に渡すこととする．引数として this 値と継続オブジェクトが増えるため，本来の引数がそれらと混ざってしまわないよう，本来の引数は 1 つの配列にまとめて別個に渡すこととする．

コード変換における JavaScript 特有の問題として，with 文を用いることで変数のスコープを実行時に変更することができる点がある．こうなると，ある名前が指している実体を確定できるのは実行時だけになってしまうので，変数名が本質的に重要な役目を果たすようになる．そのためたとえ局所変数であっても，コード変換の前後で変数名を書き換えることができない．この制約の中で変数名の衝突を避けるために，コード変換の過程で新たに現れる変数には，変数名の先頭に「$」を付すこととする☆．このような命名法による衝突の回避策は，入力として与えられるプログラムによっては完全な解法とはならないが，現実的には十分に許容されると考えられる．

もう 1 つの JavaScript 特有の問題として，Arguments オブジェクトの存在がある．これは関数呼び出しの際に生成されるオブジェクトで，実引数の数や値などを保持している配列のようなオブジェクトである．関数の中では変数 arguments を通してこれらを参照でき，可変長引数関数を記述する際に用いられる．しかし変換後のプログラムでは基本ブロックごとに別々の関数になっているため，このままでは基本ブロックごとに arguments の値が変わってしまう．また，Arguments オブジェクトの特殊な性質として，この配列要素に代入を行うと対応する仮引数の値も更新されるということがある．たとえば，次のコードを実行すると，「overwritten」と表示される．

```
function f (x) {
  arguments[0] = "overwritten";
  document.write(x);
}
f("argument");
```

このように，JavaScript の上で変数と配列要素との間に特殊な関係が存在するのは，Arguments オブジェクトだけである．しかし上で述べたように，変換後の関数は引数の渡し方が変わっているため，Arguments オブジェクトの配列要素に代入をしても，ユーザが期待するように仮引数の値が同時に更新されることはない．Arguments オブジェクトを作成できるのは関数呼び出しのときだけなので，この問題に対処するため，変換後の関数の中では組み込みメソッドである apply を用いてもう一度関数適用することで，この特殊な関係を作り上げることとした．

したがって，変換後の関数は次のような形をとることとなる．

```
function (
  $this,  // this 値
  $args,  // 引数配列
  $cont   // 継続オブジェクト
) {
  return function ( 仮引数リスト ) {
    $args = arguments;
    局所変数の宣言
    継続オブジェクトの定義
    …
    return コンテキストオブジェクト;
  }.apply($this, $args);
}
```

ここで「仮引数リスト」は，変換前の元の関数の仮引数リストとまったく同じものを使用する．こうすることで，仮引数と配列要素の間の特殊な関係が，期待どおりに成り立っている引数配列（Arguments オブジェクト）を作り上げることができる．こうしてできた Arguments オブジェクトを，$args 変数で保持しておくことで，以下の変換後の関数本体で利用可能にしておく．

変換後の関数の主な内容は，図 6 で見たような細切れにされたプログラムが継続オブジェクトの並びとして表現されたものである．継続オブジェクトの procedure プロパティが指す関数の内部で arguments 変数の値を適切に復帰させるために，procedure プロパティが指す関数は基本的には次のような形をとることになる．

```
function ($ret_val) {
  arguments = $args;
  この継続で進める処理
```

☆ JavaScript では変数名に「$」を使用することができる．ただし標準仕様[3] では，これは機械的に生成される変数にのみ使用されるべきとしている．

```
function fib (n) {
  var x, y, z;
  x = y = 1;
  while ( n != 0 ) {
    z = x + y;
    y = x;
    x = z;
    n--;
  }
  return x;
}
```

(a) 元のプログラム

```
function fib ( n ) {
    var x, y, z;
  $LABEL0:
    x = y = 1;
  $LABEL1:
    if ( !(n != 0) ) goto $LABEL2;
    z = x + y;
    y = x;
    x = z;
    n--;
    goto $LABEL1;
  $LABEL2:
    return x;
}
```

(b) goto 文とラベルを用いた形に書き換えた結果

```
 1: function fib ($this, $args, $cont) {
 2:   return function (n) {
 3:     $args = arguments;
 4:     var x, y, z;
 5:     var $LABEL0 = {
 6:       procedure: function ($ret_val) {
 7:         arguments = $args;
 8:         x = y = 1;
 9:         return {continuation:$LABEL1,
10:                 timeout     :undefined,
11:                 ret_val     :undefined};
12:       },
13:       this_val : this,
14:       exception: $cont.exception
15:     };
16:     var $LABEL1 = {
17:       procedure: function ($ret_val) {
18:         arguments = $args;
19:         if ( !(n != 0) )
20:           return {continuation:$LABEL2,
21:                   timeout     :undefined,
22:                   ret_val     :undefined};
23:         z = x + y;
24:         y = x;
25:         x = z;
26:         n--;
27:         return {continuation: $LABEL1,
28:                 timeout     : undefined,
29:                 ret_val     : undefined};
30:       },
31:       this_val : this,
32:       exception: $cont.exception
33:     };
34:     var $LABEL2 = {
35:       procedure: function ($ret_val) {
36:         arguments = $args;
37:         return {continuation:$cont,
38:                 timeout     :undefined,
39:                 ret_val     :x        };
40:       },
41:       this_val : this,
42:       exception: $cont.exception
43:     };
44:     return $LABEL0;
45:   }.apply($this, $args);
46: }
```

(c) コード変換の最終結果

図 10 コード変換の例
Fig. 10 Example of code conversion.

return コンテキストオブジェクト;
}

コンテキストオブジェクトの continuation プロパティは次に実行するべき処理，つまりジャンプ先を示していると考えることができる．その意味で，コンテキストオブジェクトを返すことと goto 文の間にはアナロジが存在する．これを利用して，コード変換の第 1 歩は入力プログラムを goto 文とラベルを用いた形に書き換えることである．すると，書き換え後の基本ブロックが継続の手続きに，goto 文が return 文にほとんどそのまま対応する．ただし，継続が必ずコンテキストオブジェクトを返すように注意しなければならない．

4.5 コード変換例

図 10 にコード変換の例を示す．まずは図 10 (a) のプログラムを goto とラベルを用いた形に書き換える．その結果が図 10 (b) である．そして，基本ブロックを継続の手続きに，goto 文をコンテキストオブジェク

トを返すように書き換えて，上で定めた変換後の関数の型にあてはめると最終的に図 10 (c) のようになる．

ここで，基本ブロック $LABEL0 の書き換えで，書き換え後にはコンテキストオブジェクトを返すようにgoto に相当するコードを補足している（9〜11 行目）ことに注意されたい．

また，return 文の書き換え方法についてはこれまで詳しく述べていなかったが，コンテキストオブジェクトの ret_val プロパティを用いて次の継続へ引数として渡している．それ以外はこれまでに述べてきた置き換えを忠実に行っただけである．

関数呼び出しの変換は少し複雑である．3.2 節で述べたように，本提案機構は変換対象コードが利用する関数を芋づる式に変換することはしない．そのような変換されない関数の代表例が組み込み関数であった．しかし，組み込み関数を利用しないで実際的なJavaScript プログラムを作成するのはほとんど不可能であるため，コード変換された関数からはコード変換された関数だけでなく，コード変換されていない関数も呼び出せるようにしなければならない．これは，変換済みの関数を呼び出す場合と変換されていない関数を呼び出す場合との 2 通りに，変換結果のコードで条件分岐することで対応している．たとえば，次のコード片：

```
r = o.m(x, y);
```

は，以下のように変換される．

```
var $LABELa = {
  procedure: function ($ret_val) {
    …
    if (o.m がコード変換済みの関数である) {
      return o.m(o, [x, y], $LABELb);
    } else {
      return { continuation: $LABELb,
               timeout     : undefined,
               ret_val     : o.m(x, y) };
    }
  },
  …
};
var $LABELb = {
  procedure: function ($ret_val) {
    arguments = $args;
    r = $ret_val;
    …
  },
  …
};
```

上のように，コード変換されている関数の場合には this 値，引数リスト，継続を引数にして呼び出すとコンテキストオブジェクトが返されるため，それをそのまま返せばよい．変換されていない関数の場合には，その場で通常どおりに呼び出して，その戻り値をコンテキストオブジェクトの ret_val プロパティの値として返す．関数呼び出しを変換する際には，実引数リストの中に関数呼び出し式が入れ子になっていないように，あらかじめ前へ括り出しておかなければならない．これは関数呼び出しを含むほとんどすべての式についても同様である．

5. 評　　価

JavaScript のサブセット言語をサポートしたプロトタイプを実装し，提案機構の評価を行った．現在プロトタイプがサポートしている構文は，if 文，while 文，try-catch 文であるが，これだけで基本的なプログラムは記述できる．

まず定性的評価として実際に簡単な Ajax アプリケーションを実装し，提案機構の記述性について評価を行った．次に，提案機構を用いることで生じるオーバヘッドについて，定量的評価を行う．

5.1 記 述 性

従来手法との記述性を比較するため，具体的なアプリケーションとして**図 11** のようなツリー掲示板を Ajax アプリケーションとして 2 種類実装して評価を行った．1 つは提案機構のプロトタイプを用いたもの，もう 1 つは Prototype [12] を用いて実装したものである．Prototype は Ajax 用の代表的ライブラリとして広く普及しており，非同期通信だけでなく GUI 操作用のライブラリなど多数のライブラリの集合である．Prototype の非同期通信ライブラリは XMLHttpRequest のラッパとして構築されており，イベント駆動型のインタフェースとなっている．そのため，2.2.2 項で述べた問題をかかえている．

ツリー掲示板は，記事のデータを最初にすべて読み込むのではなく，子ノードの記事を表示するよう指示を受けてから非同期通信によってデータを取得するよう，要求駆動で動作する．また要求がなくても，バックグラウンドで非同期通信を行い，データの先読みを行う．

先読み処理を行う手続き backgroundLoad の実際の記述は，**図 12** のようになっている．まず図 12 (b) の提案機構を用いたプログラムから説明する．backgroundLoad は読み込みたい記事の ID を配

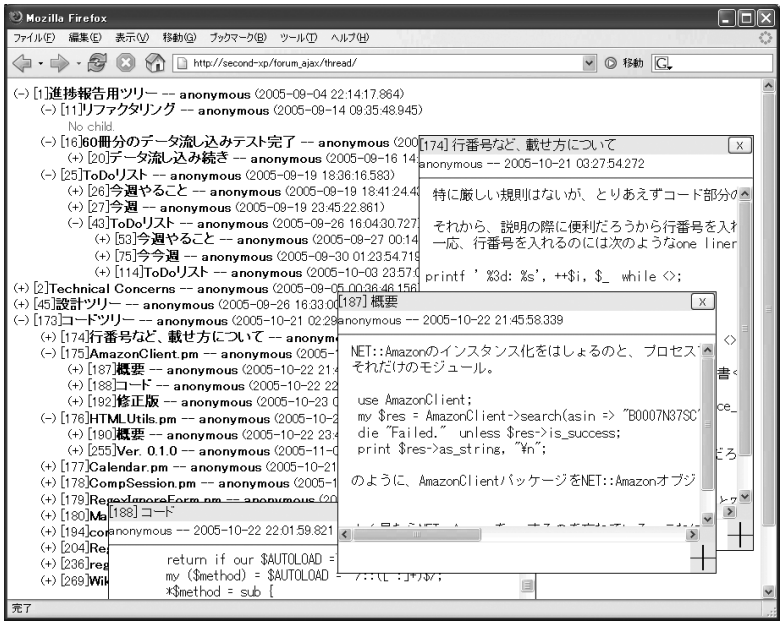

図 11 ツリー掲示板実行画面
Fig. 11 Tree BBS's execution screen.

```
function getArticle (id, cont) {
 if (article[id]) {
  cont(article[id]);
 } else {
  new Ajax.Request(
   "../article/?id=" + id, {
   method    : "GET",
   onSuccess : function (req) {
    var x, e;
    x = req.responseXML;
    e = x.getElementsByTagName("article");
    cont(new Article(e[0]));
   }
  });
 }
}

function backgroundLoad (ids, cont) {
 var i = 0;
 function loop ( ) {
  if (i < ids.length) {
   getArticle(ids[i++], function(a){
    backgroundLoad(a.children, loop);
   });
  } else {
   cont();
  }
 }
 loop();
}
```

(a) Prototype を用いた記述

```
function getArticle (id) {
  if (article[id]) {
    return article[id];
  } else {
    var r, x, e;
    r = Thread.Http.get(
      "../article/?id=" + id
    );
    x = r.responseXML;
    e = x.getElementsByTagName("article");
    return new Article(e[0]);
  }
}

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

(b) 提案機構を用いた記述

図 12 バックグラウンドで先読みを行う処理の記述
Fig. 12 Implementations of background preloading.

表 2 プログラム行数と関数定義数の比較
Table 2 Comparison of lines and functioins of code.

	コード行数	関数の数
Prototype の利用	156	27
提案機構の利用（変換前）	137	15

列で受け取り，それぞれに対して `getArticle` を適用する．実際の通信処理はこの `getArticle` の中で，`Thread.Http.get` メソッドによって行われる．したがって，各記事のデータを取得するごとに，1 回の HTTP 通信が行われる．`backgroundLoad` では，記事のデータを取得したら，その子ノード記事に対して `backgroundLoad` を再帰的に適用していく．ツリー掲示板なので，再帰的にノードをたどることでツリー全体を深さ優先で順に読み込むことができる．

一方で，Prototype を用いた場合の記述は図 12 (a) のようになる．本質的には提案機構を用いているものと同じことをしているのだが，こちらでは `backgroundLoad` での配列に対する繰返しが把握しにくくなっている．実装では関数 `loop` を間接再帰呼び出しすることでループを作り出しているが，これは制御構文を用いずに制御フローをハンドコーディングしていることに相当し，可読性・保守性の面で難がある．これは通信処理を行っている関数 `getArticle` が，イベント駆動型のインタフェースであることに起因している．また，この手法では，通信の中でエラーが発生した場合にそれを呼び出し側に通知して例外処理させるのが困難である．対して提案機構を用いた実装では，発生した例外は `backgroundLoad` の呼び出し側へと通常どおり通知される．このように，Prototype を用いた実装では 2.2.2 項であげた問題が浮き彫りになっていることが分かる．

次に提案機構の効果を定量的に示す．ツリー掲示板の 2 種類の実装それぞれについて，コードの行数と定義している関数の数を**表 2** に示した☆．これまでに述べたように，イベント駆動型の Ajax プログラムでは各イベントハンドラが関数として定義され，また関数を渡し合うことで制御フローを記述している．そのため，関数の総数がプログラムの複雑度の目安となる．

まず，行数については大きな差異はない．これは本提案機構が実装の再利用をするためのものではないことを考えれば，当然の結果である．つまり，プログラマは自分が実装すべき処理については，自分で面倒を見なくてはならないためである．そのため，どちらもおおむね同じ行数となる．一方で関数定義の数では大きな差が現れている．Prototype を用いた場合に比べて，提案機構を用いた場合にはおよそ半分の関数で記述することができている．このことから，提案機構がプログラムを通して複雑さの解消に寄与していると判断できる．

☆ ただし，GUI のためのコードは両者共通であるため，表の値から除外してある．

5.2 オーバヘッド

2 種類のベンチマークプログラムを，3 つの処理系に対して実行し，提案機構を使用した際のオーバヘッドについて測定を行った．1 つめのベンチマーク prime は 3000 番目の素数を求めるもので，処理の大半は while 文によるループである．もう 1 つの fib は 20 番目のフィボナッチ数を求めるもので，こちらは関数の再帰的呼び出しが処理の大半を占める．測定に用いた環境は，CPU が Pentium 4 3 GHz，メモリは 1 GB，OS は Windows XP Professional SP2 である．

表 3 に測定結果を示す．全体的に 59 倍から 292 倍のオーバヘッドが観測されている．処理系によって結果が異なるため一概にはいえないが，おおむね while ループに対するオーバヘッドが大きいと考えられる．提案機構を使用したプログラムでは，コード変換の結果ジャンプに相当する部分が return と関数呼び出しに書き換えられるため，これによる速度低下が大きく出ていると推測される．

実際は，本提案機構は主に非同期通信が必要となるプログラムで使用されることを想定しており，上のオーバヘッドは通信遅延に比べて十分に小さいことが予想される．5.1 節であげたツリー掲示板を用いて，記事データをバックグラウンドで読み込むのに要する時間を測定することで，そのことを確認した．評価に用いた記事データは全 303 件，計 807 KB で，各記事の大きさは 355 バイトから 30 KB まで様々であり，その平均サイズは 2665 バイトである．5.1 節で述べたように，このツリー掲示板は記事を 1 つ取得するたびに 1 回の HTTP 通信を行うので，合計 303 回の HTTP 通信を行う．クライアント機として前述の PC を，サーバ機として CPU Pentium M 1.2 GHz，メモリ 512 MB，OS Windows XP Professional SP2 の PC を用いて，両者を 100 Mbps Ethernet で接続して測定を行った．使用したサーバソフトウェアは Apache/2.0.59 (Win32) mod_perl/2.0.3 Perl/v5.8.8 である．結果を**表 4** に示す．表のように，提案機構による差は誤差に隠れて見えなくなっている．そのため，実用上は提案機構のオーバヘッドが深刻な問題になることは少ないと考えられる．

表 3 提案機構の使用/未使用での実行時間比較（単位：秒）
Table 3 Comparison of execution times (in seconds).

	Mozilla FireFox 2.0.0.3		Internet Explorer 7.0.5730.11		Opera 9.20	
	prime	fib	prime	fib	prime	fib
提案機構使用	17.72	9.39	10.83	6.53	4.09	1.77
提案機構未使用	0.06	0.15	0.08	0.09	0.07	0.02
使用 / 未使用	292	61	144	69	59	98

表 4 記事データの読み込みに要した時間（単位：秒）
Table 4 Required time to load articles (in seconds).

	Mozilla Firefox 2.0.0.3	Internet Explorer 7.0.5730.11	Opera 9.20
提案機構	16.6 ± 1.6	17.7 ± 1.3	14.8 ± 1.3
Prototype	13.4 ± 1.9	16.3 ± 1.5	14.8 ± 1.0
提案機構 / Prototype	1.2	1.1	1.0

6. 関 連 研 究

本研究では継続ベースの並行処理を応用したが，これは関数型言語，特に継続が第 1 級の対象である言語では以前からよく知られている手法である[11),14)]．このような言語ではプログラムの任意の時点で継続が取得できるため，現在のコンテキストの取得・保存が容易であることが，その理由である．しかしそのような言語の上でも，プリエンプティブなスレッド切替えを実現することは，一般には容易ではない．

JavaScript 処理系にも，継続を第 1 級の計算対象としてしている Rhino[10)] がある．しかしこれは例外的な処理系であり，JavaScript の標準仕様[3)] では継続は第 1 級の計算対象ではない．また現在のところ，Rhino は一般的な Web ブラウザに組み込まれてはいない．本研究は Ajax を主たる対象分野としているため，Web ブラウザに組み込まれていない Rhino は対象外とした．しかし，本提案機構のコード変換器の一部は，Rhino のコードを JavaScript に移植することで利用している．

本研究では JavaScript で継続を扱えるようにするため，プログラムを継続渡し形式（CPS）にコード変換する手法[1),2)] を応用した．継続渡し形式では，関数呼び出しの際に必ず，その関数のあとに実行すべき関数（継続）を引数として渡す．

そして，関数からのリターンは引数として渡された継続を呼び出すことで実現される．これは機械語にたとえていうと，おおむねサブルーチンを呼び出す際に明示的にリターンアドレスを渡していることに相当する．こうすることで関数のコールとリターンによる制御フローをプログラムから操作できるよう顕在化させることができ，それによって継続を値として取得・保存できるようにしているのである．

このようなコード変換を用いた手法は，ある種のマクロアプローチととらえることができる．Steele は RABBIT[6)] において Scheme プログラムを MacLISP にコンパイルしているが，Lisp 方言から Lisp 方言への変換として見れば，ここで Scheme はある種マクロのような役割を果たしている．RABBIT ではこの方法で `call/cc` も実装している．本提案手法は，同様のことを JavaScript から JavaScript への変換として実践している．

関数型以外の言語でコード変換により並行処理を実現した例として Li らの研究[8)] がある．これはスレッドが存在しない Java 環境である JavaCard において，並行動作するスレッドを用いたプログラミングスタイルを可能としたものである．JavaCard アプリケーションはマスタとなるホスト端末とスレーブとなるカードデバイスの上で動作し，両者が非同期で通信するマスタ・スレーブ型プログラミングで記述される．マスタとスレーブの間の非同期通信はイベント駆動型で記述されるため，本研究で Ajax プログラミングの複雑さの要因としてあげた制御フローの問題が，JavaCard プログラミングにおいても存在している．Li らはこの問題を，マスタとスレーブのそれぞれに 1 つずつ存在するスレッドが処理を委譲しあうことで実行を進めるように書かれたプログラムを，JavaCard の規格に合致するマスタ・スレーブ型プログラムにコード変換することで解決を図っている．このように，Li らの研究と本研究には多くの類似点が存在する．しかし，本研究がコード変換で継続を利用したのに対し，彼らはよりアドホックな手段によっている．また，彼らはマスタとスレーブの両方で合わせて 2 つのスレッドが存在する環境を構築したが，本研究ではクライアントである Web ブラウザの上にマルチスレッド環境を構築することで問題の解決を狙っており，Web サーバには

いっさい関与していない点が大きく異なっている．

7. おわりに

本論文では，Ajax アプリケーション開発を複雑・困難なものとしている制御フロー問題を軽減するため，イベント駆動型プログラミングに代わりマルチスレッド・プログラミングで Ajax アプリケーションを記述することを提案した．また実際に JavaScript 上でマルチスレッドを利用可能にするため，マルチスレッド・ライブラリのプロトタイプを実装し，その効果を確認した．本提案機構は可搬性を考慮し，継続ベースの並行処理とコード変換を応用することで JavaScript 処理系にはいっさい手を入れることなくマルチスレッド環境を実現した．

実際に提案機構を用いて Ajax アプリケーションを実装することで一定の有効性を検証できたが，今後いっそうの実例とユースケースの蓄積が必要である．また，提案機構によって変換されたコードには大きなオーバヘッドが存在するため，今後実装の改善により高速化することが望まれる．特に，変換結果のコードを最適化すれば，オーバヘッドの削減に大きな効果を望むことができると考えられる．

参考文献

1) Appel, A.W.: *Compiling with Continuations*, Cambridge University Press, New York, NY, USA (1992).
2) Danvy, O. and Filinski, A.: Abstracting control, *LFP '90: Proc. 1990 ACM conference on LISP and functional programming*, New York, NY, USA, pp.151–160, ACM Press (1990).
3) ECMA: *ECMAScript Language Specification*, 3rd edition (2000).
4) Ganz, S.E., Friedman, D.P. and Wand, M.: Trampolined style, *ICFP '99: Proc. 4th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, pp.18–27, ACM Press (1999).
5) Garrett, J.J.: Ajax: A New Approach to Web Applications (2005). http://www.adaptivepath.com/publications/essays/archives/000385.php
6) Steele, G.L.Jr.: RABBIT: A Compiler for SCHEME, Technical Report AI Lab Technical Report AITR-474, MIT AI Lab (1978).
7) Jones, S.L.P., Hall, C., Hammond, K. and Partain, W.: The Glasgow Haskell compiler: A technical overview, *Proc. Joint Framework for Information Technology Technical Conference*, pp.249–257 (1993).
8) Li, P. and Zdancewic, S.: Advanced control flow in Java card programming, *LCTES '04: Proc. 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, New York, NY, USA, pp.165–174, ACM Press (2004).
9) Microsoft: *XMLHttpRequest Object* (2007). http://msdn2.microsoft.com/en-us/library/ms535874.aspx
10) mozilla.org: Rhino: JavaScript for Java (2006). http://www.mozilla.org/rhino/
11) Shivers, O.: Continuations and threads: Expressing machine concurrency directly in advanced languages, *Proc. 2nd ACM SIGPLAN Workshop on Continuations*, New York, NY, USA, pp.2-1–2-15, ACM Press (1997).
12) Stephenson, S.: Prototype Javascript Library easing the development of dynamic web applications (2006). http://www.prototypejs.org/
13) Tarditi, D., Lee, P. and Acharya, A.: No Assembly Required: Compiling Standard ML to C, *ACM Letters on Programming Languages and Systems*, Vol.1, No.2, pp.161–177 (1992).
14) Wand, M.: Continuation-based multiprocessing, *LFP '80: Proc. 1980 ACM conference on LISP and functional programming*, New York, NY, USA, pp.19–28, ACM Press (1980).
15) 八杉昌宏, 馬谷誠二, 鎌田十三郎, 田畑悠介, 伊藤智一, 小宮常康, 湯淺太一：オブジェクト指向並列言語 OPA のためのコード生成手法，情報処理学会論文誌：プログラミング，Vol.42, No.SIG 11 (PRO 12), pp.1–13 (2001).

(平成 19 年 1 月 22 日受付)
(平成 19 年 5 月 25 日採録)

牧 大介

1982 年生．2005 年国際基督教大学教養学部理学科情報科学専攻卒業．2007 年電気通信大学大学院電気通信学研究科情報工学専攻修士課程修了．2007 年 4 月より同研究科同専攻研究生として在籍中．プログラミング言語とその処理系，特に動的言語に興味を持つ．

岩崎　英哉（正会員）

1960 年生．1983 年東京大学工学部計数工学科卒業．1988 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年同大学計数工学科助手．1993 年同大学教育用計算機センター助教授．その後，東京農工大学工学部電子情報工学科助教授，東京大学大学院工学系研究科情報工学専攻助教授，電気通信大学情報工学科助教授を経て，2004 年より電気通信大学情報工学科教授．工学博士．記号処理言語，関数型言語，システムソフトウェア等の研究に従事．日本ソフトウェア科学会，ACM 各会員．