情報処理学会論文誌　プログラミング　Vol. 3　No. 4　1–15 (Sep. 2010)

サーバ/クライアント自動分割を備えたWebフレームワークの設計と実装

稲津和磨[†1]　岩崎英哉[†2]

近年 Google Maps などに代表される，Ajax と呼ばれる手法を用いた Web アプリケーションが増加している．Ajax を用いた Web アプリケーションはサーバ側とクライアント側で動作する 2 つのプログラムにより構成され，それらが協調して動作する．そのため，煩雑な通信処理を記述する必要があり，さらには，それぞれの実装言語が多くの場合は異なっているため，プログラミングが非常に煩雑になる．そこで本論文では，開発効率の向上を目的として，JavaScript に基づくプログラム言語で Web アプリケーションを 1 つのプログラムとして記述し，処理の柔軟な分割を可能とするフレームワークを提案する．また，そのプログラム言語で記述されたプログラムを読み込み，サーバ側で動作するソースコードとクライアント側で動作するソースコードを出力するような機構を設計し，実装した．提案機構は，与えられたプログラムを解析し各構文要素がサーバとクライアントのどちらで実行するべきかを決定する．その際，分割方針を指定することで，同じソースコードから異なる分割を得ることができる．同じ動作をする Web アプリケーションを，提案機構と既存の手法のそれぞれを用いて記述し，プログラムを比較して記述性が向上していることを確認した．また，提案機構によるオーバヘッドは，Web アプリケーションで行われる一般的な処理については問題ない程度の小さなものであることを確認した．

Design and Implementation of Web Framework with Auto Partitioning into Server and Client

Kazuma Inazu[†1] and Hideya Iwasaki[†2]

Ajax based web applications such as Google Maps have increased in recent years. An Ajax application is composed of the server side program and the client side program. Usually these programs are written in different programming languages. Furthermore, communications between the server and the client have to be explicitly described. For this reason, development of web applications is a very complicated task. To solve this problem, we proposes a framework that enables the user to write a web application as a single program. We have implemented a system that partition such a program into both the server side program and the client side one. It decides which parts of a source program should be executed at the client side or at the server side based on a policy given by the user. We confirmed that our framework makes the development of web applications easier, by comparing a program using our framework with a program using an existing method for the same application. Although our framework causes overhead in execution time, it is acceptable for general web applications.

1. はじめに

近年，Google Maps [1] のような，Web アプリケーションと呼ばれる Web ページが広く認知されるようになった．Web アプリケーションとは，一般的なアプリケーションのように動作する Web ページであり，Web ブラウザがあるだけで実行が可能であるという特徴を持っている．

また，携帯端末の高性能化により，一般の PC で見ることができるページと同じ Web ページを携帯端末からも閲覧することができるようになってきた．そのため，Web ページ開発において携帯端末からの利用を考慮しなければならなくなってきた．

しかし，Web アプリケーションの開発は非常に煩雑である．第 1 の理由として，サーバ側とクライアント側の両方のプログラムを記述する必要があることがあげられる．それらのプログラムは，クライアント側は JavaScript，サーバ側は Perl などというように，一般的には異なったプログラミング言語で記述されるため，開発効率が悪い．また，たとえそれらを同じ JavaScript で記述したとしても，それらが協調して動作するような通信コードを陽に記述する必要があり，開発が煩雑になってしまう．

第 2 の理由として，動作環境や機器構成などの変更への追随が煩雑である点があげられる．前述した携帯端末などは一般的な PC と比べ性能が低いため，一部の処理をサーバ側で肩代わりするなどの対応が必要である．しかし，現状の方法では 1 度記述したサーバとクライアントのプログラムを変更し，サーバあるいはクライアントでの処理の一部を他方で処理するようにすることは，容易ではない．

そこで本論文では Web アプリケーションを 1 つのプログラムとして記述することを提案

†1 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
†2 電気通信大学大学院情報理工学研究科
Graduate School of Informatics and Engineering, The University of Electro-Communications

する．さらにそのプログラムを解析し，実際にサーバ側およびクライアント側で動作する 2 つのプログラムに分割する機構を設計し実装した．

1 つのプログラムとして記述することで，次の利点を得ることができる．

- 2 つの異なる言語のプログラムを作成するという開発者の負担を減らし，ロジックの再利用性を向上させことができる．
- サーバとクライアントの両方の処理を一体として記述できるようにすることで，サーバ/クライアント間の通信を陽に記述する必要がなくなり，協調動作を意識することなく開発ができる．
- 分割を自動的に行うことで，後から簡単に分割点を変更することができ，プログラムの保守性が向上する．
- サーバとクライアントにまたがる処理の最適化の可能性を示すことができる．

本論文ではまず，2 章で近年の Web アプリケーションについて説明し，開発における問題点を明らかにする．続いて 3 章では，提案機構の動作の概要と，提案機構に与えるプログラムを記述する言語である Babel の説明をする．次に 4 章で提案機構の実現方法について述べ，実際にどのように提案機構が動作しているかを説明する．5 章で実装した提案機構を用いて実際に Web アプリケーションを記述し，その有効性について議論する．6 章では，本研究の提案手法と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. Web アプリケーション

2.1 近年の Web アプリケーション

Web アプリケーションとは，Web ブラウザ上で閲覧することのできる Web ページの一種で，あたかもアプリケーションのように動作するものを指す．画面の構成は通常の Web ページと同様に HTML で記述され，そのデザインは CSS 等で記述される．ファイルの保存や，ユーザ認証などの処理はサーバ側にあるプログラムによって実行され，クライアント側での動的な処理については，ブラウザから標準的に利用できる JavaScript で記述されたプログラムによって実行される．このような構成にすることで，応答性の高いアプリケーションを実現している．

2.2 Web アプリケーション開発における問題点

Web アプリケーションの開発は，煩雑であることが知られている．その理由は Web アプリケーションの動作原理に由来するものである．前述のように，Web アプリケーションは，サーバ側で動作するプログラムと，クライアント側の Web ブラウザで動作するプログラムの 2 つからなる．しかし，このように全体で 1 つの処理を行うプログラムを別々のプログラムに記述することにより，様々な問題が発生する．

2.2.1 プログラミング言語の問題

2 つのプログラムは，一般的には異なるプログラミング言語で記述される．具体的にはサーバ側は Perl や Java，PHP などの言語，クライアント側は多くのブラウザで動作する JavaScript で記述される．つまり，Web アプリケーションの開発において，開発者は 2 つの言語について精通している必要がある．2 つの別々の言語を用いることにより，バグを招きやすくなったり，ほぼ同じコードをそれぞれで実装しなければならないなどの問題が発生する．このようなプログラムの整合性をとるのは煩雑で，プログラムの保守性の低下を招く．

このようなプログラミング言語の問題は，サーバ側を JavaScript で記述することである程度和らげることができるが，次に述べる問題点を避けることはできない．

2.2.2 プログラムの協調動作の問題

2 つのプログラムは，互いに通信を行いながら協調して動作する．そのためにデータを直列化したり，通信の処理を行うなどの煩雑な処理を記述しなければならない．

このような状況で，性能に制限のある携帯端末などへの対応を想定し，たとえばクライアント側で行っている処理の一部をサーバ側で行うようにプログラムを変更することを考える．変更はクライアント側の処理のコードの一部を切り取りサーバ側に貼り付ける，というような簡単な方法で行うことはできない．なぜならサーバ，クライアントの協調処理を行うためのインタフェースが変化するためである．

3. 提 案 手 法

3.1 概　　要

提案機構の概要を**図 1** に示す．利用者はサーバの処理とクライアントの処理を一体としてプログラムを記述する．このプログラムは JavaScript [2] を模した言語である Babel で記述する．このプログラムと分割方針を提案機構に与えると，実際にサーバ側で動作するプログラムとクライアントのブラウザ上で動作するプログラムに自動分割する．自動分割の際には，場所に依存した処理などを考慮しつつ，分割方針に基づきプログラムの各処理の実行場所を決定する．

分割方針としては，現在のところ以下の 3 種類のいずれかを与えることができる．

- なるべくサーバで実行されるように分割する．

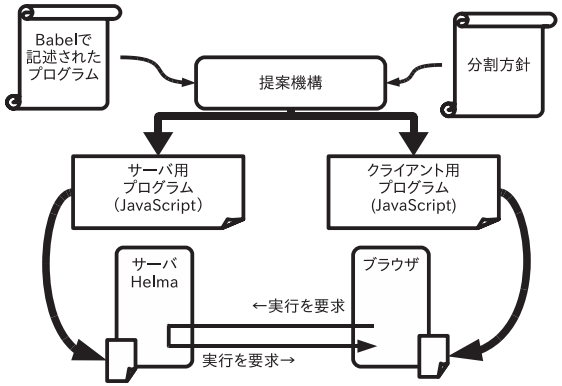

図 1　提案機構の概要
Fig. 1　System overview.

- なるべくクライアントで実行されるように分割する．
- 通信回数がなるべく少なくなるように分割する．

ユーザはこれらの方針の中から，状況に応じて最も適切なものを選択できる．さらに同一のプログラムに対して，目的に応じて異なる分割方針を与え，異なるサーバ/クライアントのプログラムを得ることもできる．いずれにしても上の分割方針は場合に応じて使い分けられるべきであり，どれが最も良いというものではない．

分割されたサーバ側のプログラムは，サーバサイド JavaScript エンジン Helma[3] で動作する JavaScript のプログラムである．クライアント側のプログラムは，一般的なブラウザ上で動作する JavaScript のプログラムである．どちらのプログラムもそれぞれの実行時ライブラリ（ランタイム）から呼び出される形で実行され，必要に応じて相手側へ動作を切り替えながら処理を進める．

3.2　Babel

入力として与えるプログラムは Babel（Babel: A Both-side Execution Language）で記述する．Babel は本研究で用いるために設計したものであり，JavaScript のサブセットに，`s_`，`c_`など処理する場所を指示する接頭辞を付加した変数や，変数の処理する場所を変更するキャスト関数を導入した言語である．JavaScript の主要な機能を提供するサブセットではあるが，`with`，`try...catch` などはサポートしていない．

表 1　キャスト関数
Table 1　Cast functions.

関数	説明
`$A`	どちらでもアクセス可能な値に変換する
`$S`	サーバでのみアクセス可能な値に変換する
`$C`	クライアントでのみアクセス可能な値に変換する

3.2.1　場所に依存した処理

Web アプリケーションには，片方の側でしか利用できない資源にアクセスするような処理や，外部に漏れると問題のある資源にアクセスする処理などが含まれる．たとえば，データベースへのアクセスなどはサーバ側でしか行うことができない．対して，HTML 要素へはクライアント側でのみアクセスできる．また，パスワード等は外部に漏れると問題のあるデータであるため，サーバ側でのみアクセスすべき資源である．

Babel では，サーバ/クライアントの一方でしかアクセスが許されない資源を格納する変数名に特別な接頭辞をつける．接頭辞 `s_`はサーバでのみ処理することができる資源を格納する変数，接頭辞 `c_`はクライアントでのみ処理することができる資源を格納する変数を意味する．このような変数に関する処理は，指示された側で行われるように分割する．接頭辞のない変数は，どちらでも処理できる資源と見なし，分割方針に基づき実行場所を決定する．

特別な接頭辞の付いた変数の値は，別の接頭辞の付いた，あるいは接頭辞のないの変数に代入することができない．これにより，場所に依存した資源が，期待される場所でのみアクセスされるようになっている．また，ひとまとまりの処理の中に異なる接頭辞の付いた変数が混在する場合は，その処理を実行する場所を定めることができないため，分割不可能というエラーになる．

依存する場所を実行中に変えたいような場合も考えられる．たとえばユーザの入力した文字列と，サーバ側に格納された本当のパスワードとの照合の処理などを行う場合である．ユーザの入力した文字列は HTML 要素を持つ値であるため，クライアント側の資源に依存した値である．しかしサーバ側にある正しいパスワードのデータはサーバから外に漏らしてはならないため，サーバ側に依存した資源である．そのため，そのまま比較を行おうとすると，比較の演算を行う場所が定まらずにエラーとなる．

このような場合は，ユーザの入力した文字列のデータの依存場所を，サーバ側の資源に変更するか，あるいはどちらで処理しても問題のない資源に変更する必要がある．そのために，Babel はキャストの機構を提供している．具体的には**表 1** に示したキャスト関数によ

```
function load(page) {
  if (page != null) {                        // ファイル名が有効か？
    s_file = open(page+".txt");              // サーバに依存した変数
    s_str = s_file.read();                   // サーバに依存した変数
    dat = $A(s_str);                         // どちらでも処理できるようにキャスト
    html = dat.replace("\n","<br>");
    c_elm.innerHTML = html;                  // クライアントに依存した変数
    size = dat.length;
    c_size.innerHTML = size + "byte";        // クライアントに依存した変数
  }
}
```

図 2　Babel による記述の例
Fig. 2　Example code of Babel.

```
function load(page){
 if(page!=null){                                  A (1)
 s_file = open(page + ".txt");                    S (2)
 s_str = s_file.read();                           S (3)
 dat = $A( s_str );                               S (4)
                                                  A (5)
 html=dat.replace("¥n","<br>");                   A (6)
 c_elm.innerHTML = html;                          C (7)
 size = dat.length;                               A (8)
 c_size.innerHTML=size+"byte";                    C (9)
 }
}
```

図 3　タグ付与の例
Fig. 3　Example of tagging.

り，依存関係を変更することができる．

3.2.2　記　述　例

Babel を用いた具体的な記述例を**図 2** に示す．この記述例は，与えられたファイル名のファイルの内容とそのバイト数を HTML 要素として表示する関数である．

ファイルオブジェクトはサーバに依存した資源，HTML の要素はクライアントに依存した資源なので，格納する変数に `s_file`，`s_str`，`c_elm`，`c_size` のように接頭辞を付加している．またサーバ側の資源であるファイルオブジェクトが返す文字列を格納する変数を，どちらでもアクセス可能な変数にキャストすることで，クライアント側にある HTML 要素に表示することを実現している．

3.3　分割方針と自動分割

3.1 節で述べたように，提案機構は Babel で記述されたプログラムを，分割方針に基づいて 2 つのプログラムに分割する．具体的には，分割を行うコマンドに分割方針をオプションで与える．

分割の際に，提案機構は Babel で記述されたプログラムの構文要素に

- サーバで実行（S で表す）
- クライアントで実行（C で表す）
- どちらでも実行可能（A で表す）

という 3 つの属性のいずれかを示すタグを付与する．タグ付与の詳細は 4.1 節で述べる．図 2 で示したプログラムのタグ付けを**図 3** に示す．分割は，このタグを考慮して行う．実行場所が定まっている，すなわち S か C のタグが付けられた処理は，それぞれの場所で処理を行うようにする．どちらでも実行可能，すなわち A というタグが付けられた処理は，分割方針に基づいて以下のように実行場所を決定する．なるべくサーバ上で実行するという方針の場合は，そのような処理をサーバ側に，なるべくクライアントという方針の場合は，そのような処理はクライアントに配置する．通信回数をなるべく少なくするという方針の場合は，そのような処理を適切な方（場合によっては両方）に配置し，少ない通信回数で実行できるようにする．

図 3 のようにタグのついたプログラムを，それぞれの分割方針で分割した場合の，サーバ，クライアント双方への各構文要素の配置および動作の流れを**図 4** に示す．

各要素の数字は図 3 の各行の数字と対応している．また，図の左半分はサーバからこの処理が呼び出された場合の動作の流れを表しており，右半分はクライアントからこの処理が呼ばれた場合の動作の流れである．さらに，それぞれの呼び出しに対して，なるべくサーバで実行するような分割，なるべくクライアントで実行するような分割，通信回数が少なくなるような分割の 3 通りを示している．合計 6 通りの動作例のぞれぞれにおいて，左側はクライアントのプログラム，右側はサーバのプログラムであり，矢印は実行時にサーバ/クライアントで動作する場所が移動することを表す．いずれの分割方法でも，タグ S の要素はサーバ側のプログラムに，タグ C の要素はクライアント側のプログラムに配置する．またタグ A の要素は，なるべくサーバ/クライアントで実行するような分割の場合にはサーバ

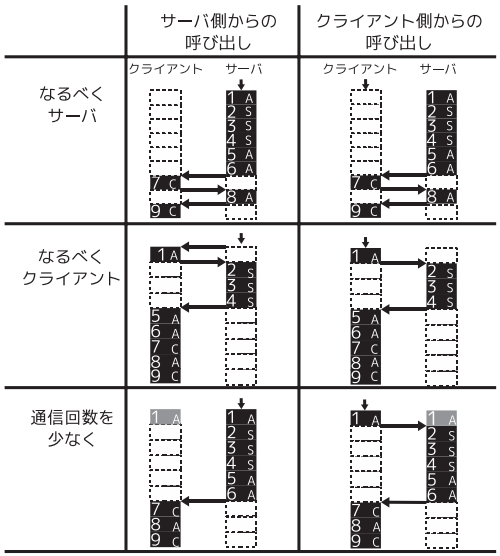

図 4　分割と動作の例
Fig. 4　Examples of partition and execution process.

側/クライアント側に配置するが，通信回数が少なくなるような分割の場合には，その（タグ A の要素）の前に S あるいは C のいずれの要素があるかに従って，サーバあるいはクライアントに配置する．それは，S/C の要素はサーバ/クライアントで実行されるため，それに続く A の要素はそのままサーバ/クライアントで実行するのが通信回数が少なくてすむからである．ただし，関数の先頭にある A の要素（図 3 では (1)）など，前のタグが定められない場合には，その要素がサーバとクライアントのどちらでも実行される可能性があるため，サーバとクライアントの双方に重複して配置する．このような配置方法をとることで，なるべくサーバ/クライアントで実行されるように分割した場合は，大半の処理がサーバ/クライアントで実行されることが，また，通信回数が少なくなるように分割した場合に，少ない通信回数で処理が完了していることを確認できる．

4. 実　　装

提案機構は，Babel の解析のために，Java で記述された JavaScript の処理系である Rhino[4] のパーサ部分を利用した．

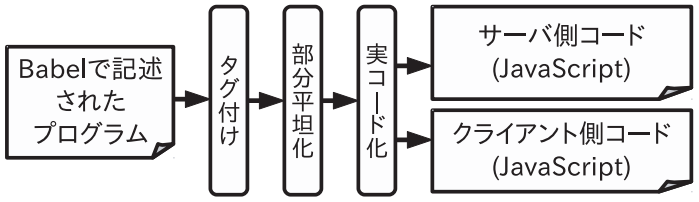

図 5　処理の流れ
Fig. 5　Processing flow.

分割処理は**図 5** に示すように，タグ付け，部分的平坦化を行い，最後に実際に動作するコードにするための実コード化の処理を行い，サーバとクライアントで動作する 2 つのプログラムを生成する．以降これらの処理を詳しく説明する．**図 6** に本章における説明のために用いる，簡略化した Babel の文法を示す．

4.1 タグ付け

与えられた Babel のプログラムを分割するためには，どの部分をどちらで実行するかを決定する必要がある．そのため 3.3 節に示したようなタグを付与する．

タグ付けは，構文要素の *ExprSite* の単位で，変数名に付けられた接頭辞に基づいて行う．接頭辞 `s_`あるいは `c_`が付いた変数を含む処理は，場所に依存した処理なので，その処理全体が接頭辞によって指定された場所になるようにタグ S あるいは C を付ける．接頭辞の付いた変数を含まない処理については，タグ A を付与する．タグ付けは抽象構文木をボトムアップにたどりながら行う．また，そのようなタグ付けが行えない場合は，エラーを返す．各 *ExprSite* に付けるタグを決定したら，その *ExprSite* に含まれる（部分木になっている *ExprSite* を除く）構文要素すべてに，求まったタグを伝播させる．

タグ付けの規則を**図 7** に示す．ただしこの規則には，決定したタグを伝播させる操作は含まれていない．$\mathcal{T}_X$ は，非終端記号 X に属す構文要素を引数にとり，その構文要素のタグを返す．その際，接頭辞 `s_` の付いた変数への代入値にタグ C が付くなど，矛盾が生じてタグ付けが不可能になれば，タグ付けは失敗し，分割不可能というエラーとなる．

図 7 の規則に出てくる演算`@`と#は，タグに関する演算である．これらの定義を**表 2** と**表 3** にそれぞれ示す．ここで，– とした部分はエラーを表しており，この組合せが現れた場合，タグ付けはエラーとなり，提案機構は分割を中止する．

`@`は代入の左辺と右辺のタグから代入全体のタグを決定する規則を表す．この規則は左辺値のタグを右辺の構文要素にも付けるというものである．

Program	::=	*SourceElement*∗
SourceElement	::=	*FuncDecl*
	|	*Statement*
FuncDecl	::=	`function (`*Var*`) {` *SourceElement* ∗ `}`
Statement	::=	*IfStatement*
	|	*ForStatement*
	|	*ExprStatement*
	|	*Block*
IfStatement	::=	`if (`*ExprSite*`)` *Statement* `else` *Statement*
ForStatement	::=	`for (`*ExprSite*`;` *ExprSite*`;` *ExprSite*`)` *Statement*
ExprStatement	::=	*ExprSite*`;`
ExprSite	::=	*Expr*
Expr	::=	*Identifier* `=` *Expr*
	|	*Expr Binop Factor*
	|	*Expr*`(`*ExprSite*`)`
Block	::=	`{` *Statement* ∗ `}`
Factor	::=	*Number*
	|	*Identifier*
	|	*CastFunc*`(`*ExprSite*`)`
	|	`(`*Expr*`)`
Identifier	::=	`s_` で始まる名前
	|	`c_` で始まる名前
	|	上記以外の名前
CastFunc	::=	`$S` \| `$C` | `$A`
Binop	::=	`+` | `-` | `*` | `/`

図 6　簡略化 Babel の文法
Fig. 6　Syntax of simplified Babel.

#は二項演算式のタグを決定する．この規則は，両オペランドのタグを求めた後，それらが S と C という組合せであれば矛盾が生じているのでエラーとするが，それ以外の組合せの場合には S あるいは C を優先させ全体のタグとする．

図 8 にタグ付与の例を示す．各構文要素の右上に付与されたタグを示し，全体の四角形の右上に，最終的に決定するタグを示した．図 8 (a) のように，左辺値がサーバに依存した `s_` の付く変数であれば，全体にタグ S が付き，右辺の数値がサーバで生成されるようになる．図 8 (b) では，数値 `120` のタグははじめは A であるが，タグ C の付けられた変数 `s_a` と加算が行われているので，#により全体にはタグ S が付けられる．図 8 (c) のようにタグ

$$
\begin{aligned}
\mathcal{T}_{ExprSite}\,[\![e]\!] &= \mathcal{T}_{Expr}\,[\![e]\!] \\
\mathcal{T}_{Expr}\,[\![v = e]\!] &= \mathcal{T}_{Identifier}\,[\![v]\!]\ @\ \mathcal{T}_{Expr}\,[\![e]\!] \\
\mathcal{T}_{Expr}\,[\![e\ op\ u]\!] &= \mathcal{T}_{Expr}\,[\![e]\!]\ \#\ \mathcal{T}_{Factor}\,[\![u]\!] \\
\mathcal{T}_{Expr}\,[\![e_1(e_2)]\!] &= \mathcal{T}_{Expr}\,[\![e_1]\!] \\
\mathcal{T}_{Factor}\,[\![n]\!] &= \mathsf{A} \\
\mathcal{T}_{Factor}\,[\![v]\!] &= \mathcal{T}_{Identifier}\,[\![v]\!] \\
\mathcal{T}_{Factor}\,[\![\texttt{\$S}(e)]\!] &= \mathsf{S} \\
\mathcal{T}_{Factor}\,[\![\texttt{\$C}(e)]\!] &= \mathsf{C} \\
\mathcal{T}_{Factor}\,[\![\texttt{\$A}(e)]\!] &= \mathsf{A} \\
\mathcal{T}_{Factor}\,[\![(e)]\!] &= \mathcal{T}_{Expr}\,[\![e]\!] \\
\mathcal{T}_{Identifier}\,[\![\texttt{s\_}v]\!] &= \mathsf{S} \\
\mathcal{T}_{Identifier}\,[\![\texttt{c\_}v]\!] &= \mathsf{C} \\
\mathcal{T}_{Identifier}\,[\![v]\!] &= \mathsf{A}
\end{aligned}
$$

図 7　タグ付けの規則
Fig. 7　Tagging rule.

表 2　代入の場合の場所の決定
Table 2　Decision rule of assignment.

		右項		
左項 @ 右項		S	C	A
左項	S	S	–	S
	C	–	C	C
	A	–	–	A

表 3　演算の場合の場所の決定
Table 3　Decision rule of binary operation.

		右項		
左項 # 右項		S	C	A
左項	S	S	–	S
	C	–	C	C
	A	S	C	A

が異なる変数どうしを演算する場合は，キャスト関数を用いる．この場合，`$A` の中（変数 `c_foo`）は構文要素 *ExprSite* なので，全体とは別個にタグ C が付けられる．これが `$A` でタグ A にキャストされ，代入の右辺全体のタグは A になる．最後に代入の左辺は S，右辺は A になり，演算子`@`により全体のタグは S に決定する．図 8 (d) は，キャストを行わずエ

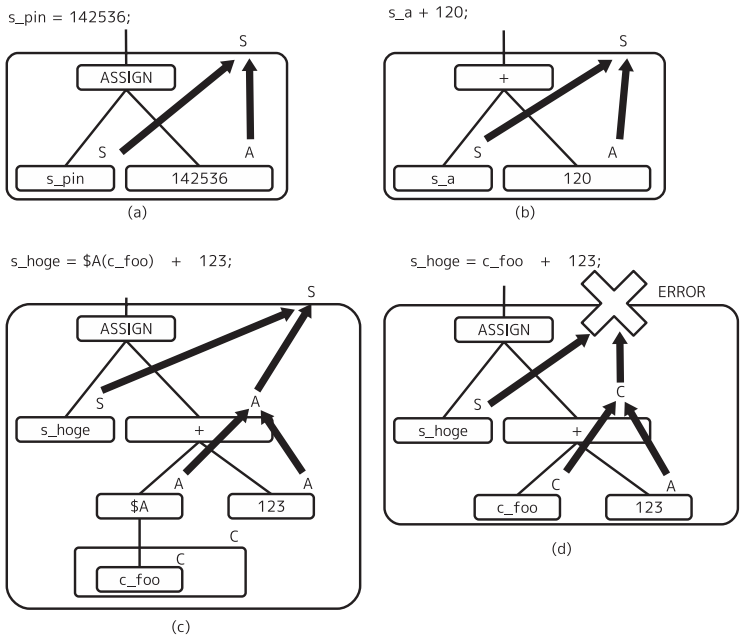

図 8　実行場所の決定
Fig. 8　Example of deciding execution place.

ラーになる例である．

4.2　部分的平坦化

コード上の 1 つの式文（*ExprStatement*），if 文（*IfStatement*），for 文（*ForStatement*）の中では，実行場所を変え相手方に制御を移す可能性がある．その可能性のある地点，具体的にはこれらの文中で異なるタグが付加された地点，あるいは関数呼び出しを行っている地点でコードを分割しておくと，4.3.1 項で後述する制御の移動（中断と再開）を実現しやすい．そのために，タグを付けた後，プログラムを部分的に平坦化して簡単にする．ここで「部分的に」とは，コード分割の必要なものに対してだけ平坦化を行う，という意味である．

部分的平坦化の処理の例を以下に示す．

- 演算の入れ子をなくし，2 項演算と代入の形にする．
- 関数呼び出しの引数を変数にする．

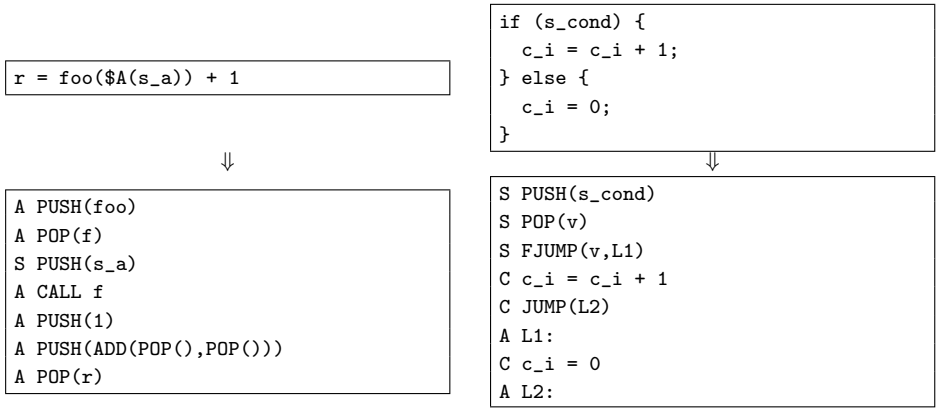

図 9　平坦化の例
Fig. 9　Example of flattening.

- if 文などの制御構造をジャンプとラベルに変換する．

部分的平坦化の簡単な例を**図 9** に示す．変換後のプログラムの左のカラムに伝播したタグを示した．右の例で if 文の then 節や else 節の中の代入文は，実行場所がすべてクライアントで，関数呼び出しを含まないため，部分的平坦化が行われていない．以降，より詳細な変換方法について説明する．

部分的平坦化の結果，1 つの式文，if 文，for 文は，複数の要素（*FlatStatement* とする）の並びから構成されるようになる可能性がある．*FlatStatement* の定義を，**図 10** に示す．*FlatStatement* は，従来の *Statement* に加え，平坦化のために新たに導入する *PushStatement*，*PopStatement*，*JumpStatement*，*CallStatement*，*Label* からなる．

`PUSH` と `POP` は，プログラムの実行時に唯一存在するスタックに関する操作である．このようなスタックが必要な理由は，Babel が出力するサーバ側/クライアント側のプログラムが互いに制御を移しながら動作する際には，関数スタックも相手側に伝える必要があるので，スタックをプログラム中で陽に操作するためである．また，`JUMP` は無条件ジャンプを行い，`FJUMP` は続く引数が偽の場合に続く引数に示したラベルにジャンプする．`CALL` は，関数呼び出しを行う．その際，実引数はスタックに事前に積んでおく．後の部分的平坦化の規則では示していないが，呼び出された関数の先頭で，実引数をスタックから取り出し，仮引数に代入する処理を行う．また，関数から戻る際には，返り値をスタックに積む．関数呼

$$
\begin{array}{rcl}
\mathit{FlatStatement} & ::= & \mathit{Statement} \\
& | & \mathit{PushStatement} \\
& | & \mathit{PopStatement} \\
& | & \mathit{JumpStatement} \\
& | & \mathit{CallStatement} \\
& | & \mathit{Label} \\
\mathit{PushStatement} & ::= & \texttt{PUSH(}\mathit{Expr}\texttt{)} \\
\mathit{PopStatement} & ::= & \texttt{POP(}\mathit{Identifier}\texttt{)} \\
\mathit{JumpStatement} & ::= & \texttt{JUMP(}\mathit{Label}\texttt{)} \\
& | & \texttt{FJUMP(}\mathit{Identifier}, \mathit{Label}\texttt{)} \\
\mathit{CallStatement} & ::= & \texttt{CALL(}\mathit{Identifier}\texttt{)} \\
\mathit{Label} & ::= & \text{大文字で始まり:で終わる名前}
\end{array}
$$

図 10　*FlatStatement* の定義
Fig. 10　Definition of *FlatStatement*.

び出しは，呼び出し先の関数の中で制御を相手側に移す可能性があることに注意されたい．

具体的な部分的平坦化規則を**図 11** に示す．部分的平坦化の規則 F は，*IfStatement*，*ForStatement*，*ExprStatement* を受け取り，タグ付けされた *FlatStatement*（タグと *FlatStatement* の組で表現する）の並びを生成する．ここで，構文要素 x にタグ t が付けられていることを x^t という表記法で表している．また，関数型言語で用いられるリスト記法で並びを表現し，並びを連結する操作を $+\!\!+$ で表している．$+\!\!+$ を where における等式の左辺で用いている場合，この等式はパターンマッチを表す．すなわち，$cs +\!\!+ [(t', z)] = \mathcal{F}\,[\![ss_1]\!]$ とある場合，$\mathcal{F}\,[\![ss_1]\!]$ の結果の列の最後の要素が (t', z) に，それ以前の部分が cs にマッチする．図中で「部分的平坦化不要の場合」とあるのは，対象とする文や式の全構成要素が同じタグを持ち，かつ，関数呼び出し式を含まない場合である．規則中で v_1，v_2 といった変数，および L_1，L_2 といったラベルが導入されているが，これらは他とは重複しない新規の変数，ラベルとする．ラベルは，実行対象ではなく場所を指し示すものなので，ラベルに対して実行場所を示すタグを付与することにはあまり意味がないが，規則中では便宜的に A を与えている．

4.3　実コード化

実コード化では，タグの付いた平坦化されたコードと分割方針から，実際に動作するサーバ側とクライアント側の JavaScript のコードを生成する．S というタグの付いた構文要素はサーバ側に，C というタグの付いた構文要素はクライアント側に配置する．A というタグ

$$
\begin{array}{l}
\mathcal{F} \;::\; \mathit{Statement} \to [(\mathit{Tag}, \mathit{FlatStatement})] \\
\mathcal{F}\,[\![\texttt{if (}e^t\texttt{) then }ss_1\texttt{ else }ss_2]\!] \\
\quad = \begin{cases}
[(t,\ \texttt{if (}e\texttt{) then }ss_1\texttt{ else }ss_2)] & \text{部分的平坦化不要の場合} \\
\mathcal{F}'_{\mathit{Expr}}\,[\![e^t]\!] +\!\!+ [(t,\ \texttt{POP(}v\texttt{)}), (t,\ \texttt{FJUMP(}v, L_1\texttt{)}] & \\
\quad +\!\!+ cs +\!\!+ [(t', z), (t',\ \texttt{JUMP(}L_2\texttt{)}), (\mathsf{A}, L_1)] & \\
\quad +\!\!+ \mathcal{F}\,[\![ss_2]\!] +\!\!+ [(\mathsf{A}, L_2)] & \text{それ以外} \\
\quad \text{where } cs +\!\!+ [(t', z)] = \mathcal{F}\,[\![ss_1]\!] &
\end{cases} \\
\mathcal{F}\,[\![\texttt{for (}e_1^{t_1}\texttt{; }e_2^{t_2}\texttt{; }e_3^{t_3}\texttt{) }ss]\!] \text{ も } \texttt{if} \text{ と同様にする} \\
\mathcal{F}\,[\![e^t\texttt{;}]\!] = \mathcal{F}_{\mathit{Expr}}\,[\![e^t]\!] \\
\mathcal{F}\,[\![\{s_1 \ldots s_n\}]\!] = \mathcal{F}\,[\![s_1]\!] +\!\!+ \cdots +\!\!+ \mathcal{F}\,[\![s_n]\!] \\
\mathcal{F}_{\mathit{Expr}}\,[\![e^t]\!] = \begin{cases}
[(t, e)] & \text{部分的平坦化不要の場合} \\
\mathcal{F}'_{\mathit{Expr}}\,[\![e^t]\!] & \text{それ以外}
\end{cases} \\
\mathcal{F}'_{\mathit{Expr}}\,[\![(v = e)^t]\!] = \mathcal{F}'_{\mathit{Expr}}\,[\![e^t]\!] +\!\!+ [(t, \texttt{POP(}v\texttt{)})] \\
\mathcal{F}'_{\mathit{Expr}}\,[\![(e\ op\ u)^t]\!] \\
\quad = \mathcal{F}'_{\mathit{Expr}}\,[\![e^t]\!] +\!\!+ \mathcal{F}'_{\mathit{Factor}}\,[\![u^t]\!] +\!\!+ [(t, \texttt{POP(}v_2\texttt{)}), (t, \texttt{POP(}v_1\texttt{)}), (t, \texttt{PUSH(}v_1\ op\ v_2\texttt{)})] \\
\mathcal{F}'_{\mathit{Expr}}\,[\![(e_1(e_2^{t_2}))^{t_1}]\!] \\
\quad = \mathcal{F}'_{\mathit{Expr}}\,[\![e_1^{t_1}]\!] +\!\!+ [(t_1, \texttt{POP(}v_1\texttt{)})] +\!\!+ \mathcal{F}'_{\mathit{Expr}}\,[\![e_2^{t_2}]\!] +\!\!+ [(t_1, \texttt{CALL(}v_1\texttt{)})] \\
\mathcal{F}'_{\mathit{Factor}}\,[\![n^t]\!] = [(t, \texttt{PUSH(}n\texttt{)})] \\
\mathcal{F}'_{\mathit{Factor}}\,[\![v^t]\!] = [(t, \texttt{PUSH(}v\texttt{)})] \\
\mathcal{F}'_{\mathit{Factor}}\,[\![\mathit{castfunc}(e^t)]\!] = \mathcal{F}'_{\mathit{Expr}}\,[\![e_t]\!] \\
\mathcal{F}'_{\mathit{Factor}}\,[\![(e^t)]\!] = \mathcal{F}'_{\mathit{Expr}}\,[\![e^t]\!]
\end{array}
$$

図 11　部分的平坦化の規則
Fig. 11　Flattening rule.

の付いた構文要素の配置場所は，3.3 節で述べたように，分割方針に基づいて決定する．実行場所を反対側に移動するようなコードは，配置場所が相手側に変化する地点に挿入する．

4.4 節で後述するように，場所を移動するときに，ランタイムはグローバル変数やローカル変数の値などを相手側に送信する．また相手側のランタイムは，送信されてきた情報を，グローバル変数やローカル変数の値に反映させる．このため，実コード化では，すべての変数への代入と参照に関して，ランタイムから内容を参照したり設定したりすることを可能とする処置を施している．

4.3.1　中断と再開の実現

部分的平坦化されたコードに含まれるジャンプや実行場所を移動するための処理を行うには，実行中のプログラムの継続を取得して，中断/再開しなければならない．本機構で

```
1   c_a = 1;                  // タグ C
2   s_b = 2;                  // タグ S
3   c_c = c_a + 3             // タグ C
```

(a) 中断/再開を含む Babel コード

```
1   swith (GOTO) {
2     case 1: c_a = 1;
3             throw(new Error({line:2}));
4     case 3: c_c = c_a + 3
5   }
```

(b) 実コード化を施したクライアント側コード

```
1   swith (GOTO) {
2     case 2: s_b = 2;
3             throw(new Error({line:3}));
4   }
```

(c) 実コード化を施したサーバ側コード

図 12　中断/再開の例
Fig. 12　Example of suspending/resuming.

はそのような処理を JavaScript で実現するために，例外と `switch`，`case` を用いている．**図 12** (a) のような Babel コードを例に用いて説明する．このコードを本機構で分割した結果を図 12 (b) および (c) に示す．

分割前のプログラムは，処理を行う場所が 2 行目でクライアントからサーバへと変化している．そのため，クライアントの処理を中断しサーバに制御を移し，サーバプログラムが 2 行目の処理を実行するように動作しなければならない．同様に，2 行目から 3 行目に関しても，処理の流れをサーバからクライアントに移す必要がある．

処理を中断する場合は，図 12 (b) の 3 行目のような例外を発行して，中断地点をランタイムに伝える．ランタイムは，例外を受け取り中断の原因を調べ，反対側に制御を移すなどの処理や，中断されたときの変数の保存などを行う．処理の再開は，`switch` で用いる変数 `GOTO` に再開場所を示す値を設定してすることによって行う．JavaScript の `switch` 文は，C 言語と同様に，`break` を記述しない限り以降の処理が実行される．この仕様を利用し，任意の位置からの処理の再開を実現している．ここでの説明は単純な例しか示さなかったが，実際の処理の再開には，実行位置の設定だけではなく変数の値の復元などが必要である．本機構では，ランタイムがこれらの値の設定，復元を行っている．

```
try {
  実コード化によって生成されたプログラムを呼び出す
  独自の関数スタックから呼び出し元の関数を調べ処理を移す
  スタックが空の場合は終了
} catch(e) {
   e を調べる
   関数呼び出しの場合，独自の関数スタックに情報を保存し関数へ処理を移す
   場所の移動の場合，必要な情報を反対側に送信する
}
```

図 13　ランタイムの擬似コード
Fig. 13　Pseude code of runtime.

中断や再開を実現するためには，このように `switch` と `case` を用いる以外にも，同じタグの付いた一連のプログラム部分を関数として分離し，継続渡し形式（Continuation Passing Style）に変換する方法も考えられる．しかしこの方法では，関数分割後のソースコードの可読性が著しく低下すること，分割された関数間で共有される（分割前の元の関数の）局所変数の扱いが煩雑になること，`switch` と `case` を用いる本実装よりも実装コストが大きくなると予想されたという理由により，本機構では `switch` と `case` を用いる方法を利用した．

4.4　ランタイム

前節で説明したように，実コード化によって出力されるコードは，実行場所を移動するときに処理を中断するために特別な例外を発行する．また関数を呼び出す場合も，直接呼び出すのではなく，特別な例外を発行し処理を中断するような動作をする．ランタイムはそのような例外を受け取り，実際に場所を移動したり，関数を呼び出したりする処理を行う．さらに，その後の処理の再開のための準備を行う．ランタイムの構造を**図 13** に示す．

4.4.1　関数スタックの把握

提案機構が出力する 2 つのプログラムが，相手側に制御を移すときは，関数スタックも相手側に伝える必要がある．そのために本機構では，関数呼び出しを特殊な例外を発行することで実現して，関数呼び出しを把握している．関数呼び出しを表す例外は，ランタイムによって受け取られる．関数呼び出し用の例外オブジェクトは，呼び出したい関数の実行位置を表すオブジェクトと，その引数を保持している．ランタイムはその情報をもとに実際に関数を呼び出す．この際ランタイムが独自に持っている関数スタックに，現在の実行位置と現在の場所（サーバかクライアントか）を記録する．関数から戻るときは，ランタイムが関数スタックを調べ，続きの実行位置に制御を移す．

4.4.2　実行場所の移動

実行場所を移動する際は，プログラムは特殊な例外を発行する．ランタイムはこれを受け取り実際に場所の移動を行う．実行場所の移動を示す例外オブジェクトは，現在の実行位置とローカル変数を保持している．ランタイムは，例外に含まれるこれらの情報をもとに，実行位置，グローバル変数やローカル変数の値，ランタイムが独自に持つ関数スタックを相手側に送信する．送信には XML-RPC[5] を用いている．相手側のランタイムはその情報を受け取り，グローバル変数やローカル変数を設定した後に実行位置をセットし続きの処理を実行する．

4.4.3　変数への代入と参照

4.4.1 項で述べたように，本機構は関数スタックを独自に扱うため，変数の参照も独自に行う必要がある．そのため，実行時にスコープ内で有効な変数に関して，名前と値のペアの配列として記憶し，変数の参照時にはその配列を順に調べ，該当する変数の値を書き換えている．また，相手側に制御を移す際にはこの配列を渡すようにしている．現在の実装では，関数オブジェクトの直列化をサポートしていないため，クロージャをサーバクライアント間でやりとりすることはできないが，サーバ側あるいはクライアント側だけで利用するという制限の範囲内ではクロージャも正しく作動する．

5. 評　　価

本機構を評価するために，一般的な PC と携帯端末を用いて実験を行った．一般的な PC と携帯端末を用いることで，クライアントの性能の違いによる実行時間の違いを，確認できると考えた．実験に用いた機器を，**表 4** に示す．クライアントの PC とサーバは，スイッチングハブを介して接続されている．

5.1　基本性能の評価

本機構では，Babel プログラムを部分的に平坦化し，関数呼び出しを例外を用いて実現しているため，サーバ/クライアントの協調動作以外の基本的な部分でかなりの実行オーバヘッドをともなう．この基本的なオーバヘッドを評価するために，次の 3 つのプログラムを用いて測定を行った．

1 つ目のベンチマーク sum は，1 から 100,000 までの和を求めるプログラムで，大半の処理は `for` によるループである．このベンチマークでは基本的な処理時間のオーバヘッドを確認することができる．

2 つ目のベンチマーク fib は，18 番目のフィボナッチ数を求めるものである．こちらは，関数の再帰呼び出しが処理の大半である．そのため，このベンチマークにより，例外を用いた関数呼び出しによるオーバヘッドを測定できる．

3 つ目のベンチマーク str は，文字列処理のプログラムである．行の頭に特殊な記号がある場合は，その行の前後に HTML のタグを挿入する処理を行うプログラムに，800 文字程度の文字列を与えた．Web アプリケーションの一般的な処理の例であり，実際的な処理におけるオーバヘッドの測定を目的としている．

これらのベンチマークを，提案機構を使わずに JavaScript で直接記述したプログラムの実行時間と，提案機構を用いて，なるべくクライアント側で実行するように分割した場合と，なるべくサーバ側で実行した場合の実行時間を測定した．測定はクライアント側で行ったため，サーバ側で実行した場合は，計算要求と結果の送信のための通信時間も含まれている．

結果を**表 5** に示す．一般的な PC で実行した場合のオーバヘッドは，なるべくクライアントで実行した場合で数十倍から数百倍，なるべくサーバで実行した場合では大きい場合で数千倍となっている．特に関数呼び出しが大量に含まれる fib のオーバヘッドが大きいが，

表 4　実験環境
Table 4　Experimental environments.

	PC	携帯端末	サーバ
製品名		HT-03A	
CPU	Intel Cerelon M 1.3 GHz	QUALCOMM MSM7201a	AMD Opteron 2376
	1.3 GHz	528 MHz	2.3 GHz（2 Sockets, 4 Cores）
RAM	768 MB	192 MB	8 GB
OS	WindowsXP Profesional	AndroidOS 1.6	Debian GNU/Linux 5.0.3
LAN	100 Mbps	802.11 g	100 Mbps
ブラウザ	Firefox 3.0.17	標準ブラウザ	

表 5　提案機構を用いた場合の実行時間比較（単位：ミリ秒括弧内は比較対象との比）
Table 5　Comparison of execution times (in miliseconds) and overhead.

PC	sum	fib	str
JavaScript	19 (1.00)	2.67 (1.00)	3.67 (1.00)
なるべくクライアント	513 (27.0)	1,570 (　609)	120 (32.6)
なるべくサーバ	904 (47.6)	19,200 (6,860)	522 (142)

携帯端末	sum	fib	str
JavaScript	330 (1.00)	54.0 (1.00)	93.3 (1.00)
なるべくクライアント	5,080 (15.4)	20,600 (381)	2,110 (22.6)
なるべくサーバ	1,789 (5.43)	19,799 (364)	2,000 (21.4)

表 6　行数の比較
Table 6　Comparison of lines of code.

	言語	行数	
従来方式	JavaScript	44 行	
	Python	18 行	計 62 行
提案方式	Babel	49 行	計 49 行

これは関数呼び出しを，特殊な例外の発行に変換していることによる影響である．しかし，本機構の目的である，Web アプリケーションを記述において，実際に fib のような多くの関数呼び出しを含む処理を記述することは少なく，実際の処理は str のような簡単な文字列処理などが占めるであろう．そのため，多くの場合の処理速度のオーバヘッドは sum や str 程度になると考えられる．一般的な PC で提案機構を使用した場合，クライアントで処理した方が速くなることが確認できた．この結果はクライアントの性能に依存するが，一般的な PC であれば，ほぼ同様の傾向を示すと考えられる．

携帯端末で実行した場合のオーバヘッドも，数十倍から数百倍となっている．fib の実行速度の比が一般的な PC ほど遅くなっていないのは，JavaScript エンジンの違いだと考えられる．本実験においては，携帯端末で提案機構を使用した場合，サーバで処理した方が速くなることが確認できたが，一般的には結果は環境により大きく左右されるであろう．

5.2　実際的なアプリケーションを用いた評価

実際に具体的な Web アプリケーションである Wiki を記述することにより，従来手法との比較を行った．Wiki とはテキストの編集と閲覧を行う Web アプリケーションで，独自の記法で入力されたテキストデータをサーバに保存し，その記法に基づいて整形された HTML 文書を表示するものである．

提案手法として Babel でプログラムを記述したものと，従来手法として，クライアント側は JavaScript，サーバ側は Python で記述したプログラムを用意した．従来手法では，JavaSctipt で行う通信の処理の部分に，`Prototype.js`[6] という一般的に用いられている JavaScript のライブラリを利用した．Wiki の書式変換の処理はクライアント側に記述した．

5.2.1　行数の比較

プログラムの行数の比較の結果を**表 6** に示す．提案機構の方が行数が少なく記述できていることが分かる．この行数の違いは，主にサーバとの通信処理である．本機構ではサーバとの通信処理を自動的に挿入するため，Babel のプログラム上には通信処理を陽に記述する必要がない．

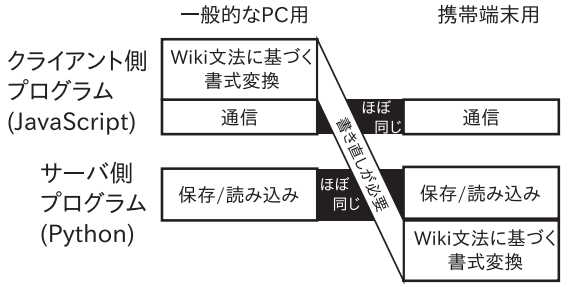

図 14　プログラムの再利用性
Fig. 14　Reusability of code

表 7　再利用性の評価
Table 7　Comprison of reusability.

		変更前	再利用	新たに記述	
従来方式	JavaScript	44 行	18 行	10 行	
	Python	18 行	18 行	20 行	計 30 行
提案方式	Babel	49 行	49 行	0 行	

5.2.2　様々な環境への対応

5.1 節で示したように，携帯端末のような一般的な PC に比べて性能の低いクライアントの場合は，サーバで実行した方が，処理が速くなったり，処理速度は同等でも CPU の使用率が減り，消費電力が抑えられたりするといったメリットが得られる．そこで，同じ動作をするアプリケーションを，一般的な PC 向けに大半の処理をクライアントで実行するものと，携帯端末向けに大半の処理をサーバで実行するものの 2 種類用意することを考える．

従来の手法を用いて，そのような 2 種類のアプリケーションを記述し，プログラムの再利用性について考察する．**図 14** にプログラムの機能と，その再利用性について示した．

一般的な PC 用のプログラムを携帯端末向けのプログラムに書き換えたときに，再利用できた部分の行数と，新たに書き加えた部分の行数を**表 7** に示す．

図 14 に示したように，クライアントからサーバへの通信コードについては，少しの変更で再利用できる．また，サーバ側のファイルの読み込みや，保存の処理についても，ほぼそのまま再利用できる．しかし，Wiki 文法に基づく書式変換の処理については，実行場所が変わるため，プログラミング言語が異なってしまい，ほぼ同じ処理であるにもかかわらず，まったく再利用することができない．そのための移植作業は手間がかかり，開発効率の低下

を招く．また，2 つのソースコード間に不整合が生じやすく，保守性も低下する．

提案機構を用いた場合は，プログラムには変更を加えずに分割方針を変更するだけで，処理の実行される場所が異なるプログラムを得ることができる．同じプログラムを分割して，2 種類のプログラムを出力するため，2 つのプログラムは同じ動作をするものとなる．そのため，保守性が高く開発効率が向上する．

6. 関 連 研 究

提案機構を用いると，通信コードや直列化のコードを意識せずにプログラムすることが可能となる．このような仕組みを提供するものとして Jaxer [7]，GWT [8]，Links [9] があげられる．また，本機構のように 1 つの言語で Web アプリケーションを記述し，それを自動分割するものとして Swift [10] があげられる．

Jaxer は，JavaScript を用いて 1 つのプログラムの中にサーバの処理とクライアントの処理を記述し，それらの処理を協調して動作させるフレームワークである．関数を処理の場所指定の単位としているため，1 つの関数の中にサーバでの処理とクライアントでの処理を混在させることはできず，関数ごとにどの場所で実行するかを利用者が指定する必要がある．これに対し本研究では，Babel を用いて通常の JavaScript のように記述し，場所の指定を変数に付ける接頭辞によって行う．そのため，1 つの関数の処理内容を分割することができ，プログラムのどの部分がどこで実行されるかを，詳細に意識する必要はない．

GWT は，Java を用いて Web アプリケーションを記述するフレームワークである．これを用いることで，サーバのコードもクライアントのコードも Java で記述することができる．あらかじめ用意されたコンポーネントを組み合わせることで，一般的な Java アプリケーションを記述するかのように Web アプリケーションを記述できる．GWT はクラスを場所指定の単位としているため，クラスごとにどの場所で実行するかを利用者が決定することになる．このため Jaxer と同様に，分割の単位が本機構よりも粗い．

Links は，独自の関数型言語でサーバの処理とクライアントの処理を記述し，それらの処理を協調して動作させるフレームワークである．Jaxer と同様に関数単位で実行場所を指定する点，関数型言語を記述言語として採用している点が，本研究との違いである．また，利用者は独自の言語を習得する必要がある．本研究で用いる Babel は，JavaScript を模した言語であるため，JavaScript を習得している利用者は，比較的簡単に習得できる．

上で述べた研究と本研究の大きな違いに，分割する単位があげられる．Jaxer，GWT，Links では，クラスあるいは関数単位で実行場所を指定するのに対し，Babel ではより細かな単位で実行場所を分割できる．Jaxer，GWT，Links でも，細かなクラスや関数を多く定義することにより，Babel と同等の粒度で実行場所を移動させて実行を進めることが可能ではある．しかし，プログラマがそのように細かな関数などを直接記述するのは煩雑であり，また分割方針の変更に追随するのにも手間が必要である．また Babel は，より細かい粒度での自動分割を行っているため，多様な分割方針への発展の余地がある．たとえば，データフローを調べ，よりプログラムが効率良く動作するように分割を行うなどの発展性が考えられる．

Swift は，Java 言語を拡張した言語で記述されたプログラムを，安全に分割することを目的としている．変数や関数にセキュリティアノテーションを付加することで，ユーザに秘密にしたいデータが流出しないような安全な分割を行う．Swift ではデータフローを考慮するため安全な分割をすることができるが，そのために記述しなければならないアノテーションが複雑になる．また，本機構のように分割方針を変更するようなことはできない．

7. お わ り に

本論文では，より簡単に Web アプリケーションを記述できるようにするために，1 つのプログラムとして Web 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 Babel で記述したプログラムを実行できるようにするための変換機構を実装し，その効果を確認した．

実際に提案機構を用いて Web アプリケーションを実装し，一定の有用性を検証したが，実行速度や分割方法については改善の余地がある．

現在の実装では，Babel で記述してある順序で処理を行っているため，何度も通信を繰り返してしまい，処理速度が低下してしまうことがある．このような処理に関して，データフローを解析し，結果が変わらないように処理を並べ替えることができれば，処理速度の大幅な改善が望める．

また，分割方法についても，単にサーバ/クライアントに分けるのではなく，1 度実行してみて，その履歴から実行場所の見当をつける方法や，実行時に負荷を計測し動的に場所を移動するなどの手法が考えられる．今回プログラムの記述言語として設計した Babel に関しても，より変換しやすい言語に再設計することも今後の課題である．

参　考　文　献

1) Google: Google Maps. http://maps.google.com/
2) Flanagan, D.: *JavaScript: The Definitive Guide, 5th edition*, O'Reilly & Associaties Inc. (2006).
3) Helma. http://helma.org/
4) mozilla.org: Rhino: JavaScript for Java. http://www.mozilla.org/rhino/
5) Winer, D.: XML-RPC Specification (1999). http://www.xmlrpc.com/spec
6) Prototype Core Team: *Prototype JavaScript framework*. http://www.prototypejs.org/
7) Aptana Jaxer: The world's first Ajax server. http://www.jaxer.org/
8) Google: Google Web Toolkit. http://code.google.com/intl/en/webtoolkit/
9) Cooper, E., Lindley, S., Wadler, P. and Yallop, J.: Links: Web Programming Without Tiers, *Proc. FMCO 2006*, LNCS 4709, pp.266–296 (2006).
10) Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L. and Zheng, X.: Secure Web Applications via Automatic Partitioning, *Proc. 21st ACM Symposium on Operating Systems Principles* (*SOSP 2007*), pp.31–44 (2007).

付　　録

A.1　行数の比較に用いた wiki のソースコード

A.1.1　Babel で記述したコード

```
function convert(s){          // wiki 文法→ html 変換
    var l = s.split("\\n");  // 改行単位でリストに変換
    var ret = [];
    for(var i = 0;i<l.length;i++){  // 各行ごとに変換
        if(l[i].search("!!!")==0){
            ret[i] = "<h1>"+erase(l[i].slice(3,l[i].length))+"</h1>";
        }else if(l[i].search("!!")==0){
            ret[i] = "<h2>"+erase(l[i].slice(2,l[i].length))+"</h2>";
        }else if(l[i].search("!")==0){
            ret[i] = "<h3>"+erase(l[i].slice(1,l[i].length))+"</h3>";
        }else{
            ret[i] = erase(l[i]) + "<br />";
        }
    }
    return ret.join("\\n");  // 改行で結合
}
function erase(s){            // 危険な文字を除去
    s= s.replace(">","&gt;");
    s= s.replace("<","&lt;");
    return s;
}
function load(){              // サーバ上のファイルの読み込み
    var s_f = new File("out.txt");
    s_f.open();
    var s_s = ""
    while(!s_f.eof()){
        s_s += s_f.readln()+"\\n"
    }
    s_f.close()
    var c_s = $C(s_s);        // クライアントで扱えるように変換
    var c_source = document.getElementById("source");
    c_source.value = c_s;     // テキストエリアに表示
    preview();
}
function save(){              // サーバ上のファイルに書き込み
    var dat = document.getElementById("source").value;
    var s_f = new helma.File("out.txt");
    s_f.remove();
    s_f.open();
    s_f.write(dat);
    s_f.close();
    var c_msg = "save ok";
    alert(c_msg);             // 書き込み完了を通知
}
function preview(){           // wiki 文法文字列を変換し表示
    var c_d1 = document.getElementById("source");
    var c_d2 = document.getElementById("view");
    c_d2.innerHTML = convert(c_d1.value);
}
```

A.1.2　一般の PC 向け Wiki の Python で記述した部分

```
import cgi
if __name__=="__main__":
    f = cgi.FieldStorage();
    s = f.getfirst("text");    # クエリを取得
    m = f.getfirst("method");
    print "Content-Type: text/plain\n"
    if m=="save":              # 保存処理
```

```
        f = open("out.txt","w");
        f.write(s);
        f.close();
        print "ok"
    elif m=="load":             # 読み込み処理
        f = open("out.txt","r");
        s = f.read();
        f.close();
        print s;
    else:
        print "unrecognized method"
```

A.1.3　一般の PC 向け Wiki の JavaScript で記述した部分

```
function convert(s){          // wiki 文法→ html 変換
    var l = s.split("\\n");  // 改行単位でリストに変換
    var ret = [];
    for(var i = 0;i<l.length;i++){  // 各行ごとに変換
            if(l[i].search("!!!!")==0){
                ret[i] = "<h1>"+erase(l[i].slice(3,l[i].length))+"</h1>";
            }else if(l[i].search("!!!")==0){
                ret[i] = "<h2>"+erase(l[i].slice(2,l[i].length))+"</h2>";
            }else if(l[i].search("!")==0){
                ret[i] = "<h3>"+erase(l[i].slice(1,l[i].length))+"</h3>";
            }else{
                ret[i] = erase(l[i]) + "<br />";
            }
    }
    return ret.join("\\n"); // 改行で結合
}
function erase(s){           // 危険な文字を除去
    s= s.replace(">","&gt;");
    s= s.replace("<","&lt;");
    return s;
}
function load(){             // サーバ上のファイルの読み込み
        var source = document.getElementById("source");
        new Ajax.Request("/cgi-bin/cwiki.cgi",{
                method:"get",
                parameters:"method=load",
                onComplete:function(res){
                        source.value = res.responseText
                }});        // prototype.js の API
}
function save(){             // サーバ上のファイルに書き込み
        var dat = document.getElementById("source").value;
        new Ajax.Request("/cgi-bin/cwiki.cgi",{
                method:"get",
                parameters:"method=save&data="+dat,
                onComplete:function(){
                        alert("save ok");
                }});        // prototype.js の API
}
function preview(){          // wiki 文法文字列を変換し表示
    var c_d1 = document.getElementById("source");
    var c_d2 = document.getElementById("view");
    c_d2.innerHTML = convert(c_d1.value);
}
```

A.1.4　携帯端末向け Wiki の Python で記述した部分

```
import cgi
def convert(s):                 # wiki 文法→ html 変換
    l = s.split("\n");          # 改行単位でリストに変換
    ret = [];
    for x in l:                 # 各行ごとに変換
        if x.find("!!!")==0:
            ret.append("<h1>"+erase(x[3:])+"</h1>")
        elif x.find("!!")==0:
            ret.append("<h2>"+erase(x[2:])+"</h2>")
        elif x.find("!")==0:
            ret.append("<h3>"+erase(x[1:])+"</h3>")
        else:
            ret.append(erase(x) + "<br/>")
    return "\n".join(ret);      # 改行で結合
def erase(s):                   # 危険な文字を除去
    s = s.replace(">","&gt;");
    s = s.replace("<","&lt;");
    return s;
if __name__=="__main__":
    f = cgi.FieldStorage();
    s = f.getfirst("text");     # クエリを取得
    m = f.getfirst("method");
    print "Content-Type: text/plain\n"
```

```
    if m=="preview":            # 変換処理
        if s:
            print convert(s);
    elif m=="save":             # 保存処理
        f = open("out.txt","w");
        f.write(s);
        f.close();
        print "ok"
    elif m=="load":             # 読み込み処理
        f = open("out.txt","r");
        s = f.read();
        f.close();
        print s;
    else:
        print "unrecognized method"
```

A.1.5　携帯端末向け Wiki の JavaScript で記述した部分

```
function load(){    // サーバ上のファイルの読み込み要求
        var source = document.getElementById("source");
        new Ajax.Request("/cgi-bin/swiki.cgi",{
                method:"get",
                parameters:"method=load",
                onComplete:function(res){
                        source.value = res.responseText
                }});       // prototype.js の API
}
function save(){    //サーバ上のファイルの保存要求
        var dat = document.getElementById("source").value;
        new Ajax.Request("/cgi-bin/swiki.cgi",{
                method:"get",
                parameters:"method=save&text="+dat,
                onComplete:function(){
                        alert("save ok");
                }});       // prototype.js の API
}
function preview(){ //wiki 文字列の変換要求
        var dat = document.getElementById("source").value;
        var view = document.getElementById("view");
        new Ajax.Request("/cgi-bin/swiki.cgi",{
                method:"get",
                parameters:"method=preview&text="+dat,
                onComplete:function(res){
                        view.innerHTML = res.responseText
                }});       // prototype.js の API
}
```

(平成 22 年 2 月 15 日受付)
(平成 22 年 5 月 6 日採録)

稲津　和磨

1985 年生．2008 年電気通信大学電気通信学部情報工学専攻卒業．2010 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．Web とその周辺技術，特にプログラミング言語に興味を持つ．

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．