HTML/CSS/JavaScript に対するサイズ最適化リテラシの醸成に向けて

柗本 真佑[†1,a)]　佐伯 幸郎[†1]　中村 匡秀[†1]

概要：Web のリソースはインターネットを経由して転送されるという特性上，そのファイルサイズは極力小さいことが望ましい．画像や音声，静止画等のメディア系リソースは，単体でそのファイルサイズが大きい．そのため，Web 上に公開する前に JPEG や MP3 等の圧縮フォーマットに変換することが，Web のリテラシ（あるいは常識）として広く知られている．一方で，HTML や CSS，JavaScript 等のテキストベースの Web リソースは，ファイルサイズが小さいため，圧縮処理や最適化処理の適用が軽視されてきた．しかしながら昨今の Web は，JavaScript を中心とした巨大な処理がクライアント側で行われることも珍しくなく，そのファイルサイズは肥大化の一途をたどっている．本稿では，テキストベースの Web リソースに対する最適化リテラシの醸成を目的とし，いくつかの最適化技術を紹介するとともに，Web 全体に存在する JavaScript の最適化効果について報告する．

1. はじめに

本稿では，インターネット全体に対するトラフィック量削減のための，**Web リソースのサイズ最適化**について考える．ここでの Web リソースとは，Web を構成するファイル群のことであり，大別してメディア系リソース（画像や動画，音声）とテキスト系リソース（HTML や CSS，JavaScript）に分類することができる．

このうち，メディア系リソースは単体でそのファイルサイズが大きく，圧縮フォーマットによる配布が Web のリテラシとしてすでに醸成されている．これらのメディア系リソースは，基本的にはその品質（画質や音質）とデータ量とのトレードオフで最適化されており，コンテンツの性質に応じて可逆フォーマット（png 等），あるいは不可逆圧縮フォーマット（jpeg/gif，mp4/flv/ogg，mp3 など）が選ばれることが一般的である．

一方でテキスト系リソースに対するサイズ削減手法の適用（サイズ最適化と呼ぶ）は，Web 開発者のリテラシとして醸成されているとは言い切れない．テキスト系リソースは，サーバがソースコードをそのままクライアントに送信し，クライアント側で適宜実行するコードオンデマンドというアーキテクチャをとる．そのため，バイナリ配布のような形態とは異なり，コンパイラによる最適化が必ずしも適用されるとは限らない．例えば，可読性確保のためのインデントや改行，コメントはそもそも実行に不要であり，また到達不能コードや利用されない変数，デバッグ用のコードなども無駄な記述であるといえる．

ネットワーク通信量の削減という観点では，これらのコードは事前に何かしらの手段でもって最適化されるべきである．Ihm と Pai のトラフィック調査 [1] によると，Web 上のデータ通信量の割合はメディア系リソースが約 50%と支配的ではあるが，前述の通りすでにリテラシとして浸透している．一方で，テキスト系リソースの占める割合は 20%ではあるが，現在は Ajax や HTML5 等のクライアントサイド処理技術が普及しており，特に JavaScript のサイズは今後も肥大化すると考えられる．

さらに，テキスト系リソースに対するサイズ最適化リテラシが広まっていない理由として，Web リソース開発に対する障壁の低さが挙げられる．Web リソースの開発は，統合開発環境やコンパイラが不要であり，テキストエディタとブラウザさえあれば開発が可能である．そのため，「動けば良い」という思想で作られたコードも Web 上に数多く存在している．またテキスト系リソースは単体でそのサイズが小さいことから，サイズ最適化処理が軽視されていることも考えられる．実際に，Web 全体におけるアクセスランキング [*1] のトップ 10 のうち，徹底してサイズ最適化を施している Web サイトは Google と Facebook のみであり，その他の Yahoo や Wikipedia，Amazon では不要なインデ

[†1] 現在，神戸大学大学院システム情報学研究科
Presently with Graduate School of System Infomatics, Kobe University
[a)] shinsuke@cs.kobe-u.ac.jp

[*1] http://www.alexa.com/topsites

© 2015 Information Processing Society of Japan

ントやコメントが残されたまま，無駄なデータ転送として我々の端末にダウンロードされているのが現状である．

本稿の目的は，テキスト系の Web リソースに対する最適化リテラシの醸成にある．そのためには，まずはサイズ最適化の効果を定量的に調査し，Web エンジニアに対してその効果を広く知らしめること，およびツールや Web サーバへのプラグイン機能としてサイズ最適化手法を提供し，容易に利用できるようにすることが必要であると考える．本稿では，この目的の第一歩として，まず既存のサイズ最適化手法を整理し，実際の Web リソースに対する最適化効果に関する実験を行う．さらに最適化手法のツール化方針について検討する．

2. テキスト系 Web リソースに対するサイズ最適化

すでに様々なサイズ最適化手法が提案されている．本節では，まずこれらの手法を以下の 2.1 節から 2.6 節の 6 種類に分類し，それぞれの性質とその副作用について考察する．

2.1 非論理的表現の削除

プログラムロジックに関係のないコメントや改行，インデント等の表現を削除する方法である [2][3]．テキスト系 Web リソースに対する最も単純なサイズ最適化手法といえる．例えば，以下のような 1 から 10 までの総和を表示する JavaScript コードが与えられた場合，

```
// initialization
var sum = 0;
// summation
for (var i = 0; i <= 10; i++) {
  sum += i;
}
// output
alert(sum);
```

本手法を適用した際の結果は以下のようなコードが得られる．

```
var sum=0;for(var i=0;i<=10;i++){sum+=i;}
alert(sum);
```

コメントやインデント，改行，変数代入式のスペースなどが削除されている．最適化効果としては，102 バイトのコードが 52 バイトとなっており，約 50%の削減率が得られている．JavaScript に限らず，HTML や CSS でもコメントやインデントが多く含まれており，本手法の適用が可能である．

2.2 論理的表現の省略

プログラム中で利用される変数名や関数名などの論理的な表現を可能な限り短く省略する，あるいは削除する方法である [2][3]．2.1 節の手法で最適化されたコードに対して，本手法を適用すると以下のようなコードが得られる．

```
var s=0;for(var i=0;i<=10;i++){s+=i}alert(s)
```

変数名 sum が s に省略されているほか，省略可能な行末のセミコロンが削除されている．元のコード 102 バイトに対して，44 バイトの削減効果（削減率 57%）が得られている．他にも，JavaScript では以下のような様々な省略記述が可能であり，ショートハンドコーディング [4] やコンデンスドコードとして知られている．

```
var a = void 0;    // var a = undefined;
var b = !1;        // var b = false;
var c = {};        // var c = new Object();
var d = [];        // var d = new Array();
```

この論理的表現の省略は HTML リソースと CSS リソースに対しても有効である．HTML の場合，XML の属性要素（id 名や class 名など）の省略といった形で適用できる．CSS の場合は色コードの省略指定や，プロパティをまとめて指定できるショートハンドプロパティ [5] などが提供されている．その一例を挙げる．以下は生の CSS コードであり，

```
#container {
  margin: 0px 0px 0px 0px;
  color: #EEEEEE;
  font-style: italic;
  font-size: 1.2em;
  font-family: Arial, sans-serif;
}
```

ショートハンドプロパティを利用すると以下のような省略が可能である．

```
#container {
  margin: 0;
  color: #EEE;
  font: italic 1.2em Arial, sans-serif;
}
```

2.3 論理的構造の最適化

プログラムの論理構造を考慮したサイズ最適化手法であり，様々な方法が存在する．未使用変数の宣言部の削除や，未到達コードの削除，冗長コードの削除のほか，アルゴリズム自体を変更する方法などが存在する．一般的にこれらの方法は，メモリ使用量の節約や実行速度の向上などの観点からの最適化によって実施されることが多い．

2.1 節の総和計算コードに対して，最適化を行うと以下のような結果が得られる．

```
alert(55)
```

© 2015 Information Processing Society of Japan

表 1　テキスト系 Web リソースに対するサイズ最適化手法の比較
Table 1　Comparison of size optimization techniques for text-based web resources.

手法名	適用対象			クライアントに対する副作用
	HTML	CSS	JavaScript	
非論理的表現の削除	✓	✓	✓	可読性の低下
論理的表現の省略	✓	✓	✓	可読性の低下
論理的構造の最適化	n/a	n/a	✓	可読性の低下，（メモリ量低下，実行速度向上）
動的自己復元	n/a[*1]	n/a[*1]	✓	可読性の低下，実行時負荷の増加
リソース結合	✓	✓	✓	可読性の低下
HTTP 圧縮	✓[*2]	✓[*2]	✓[*2]	実行時負荷の増加

[*1] HTML/CSS 単体での適用は不可能だが，JavaScript 上で HTML/CSS を復元するようなコードを書けば適用可能．
[*2] HTTP プロトコルレベルの圧縮であるためリソースの種別を問わず適用可能．ただしテキスト系以外には効果は低い．

この場合，for ループによる加算処理は，変数の初期値やループ回数が外部要因に全く影響を受けないため，定数 55 の出力として省略可能である．なおこの場合は，データサイズとメモリ使用量，速度のいずれの観点でも肯定的な最適化効果が得られている．

jQuery をはじめとする JavaScript ライブラリの多くは，この最適化処理が事前に加えられた状態で配布されており，*.min.js という名称で CDN 等のコンテンツ配信用サーバ等にデプロイすることがリテラシとなっている．

2.4　動的自己復元

ハフマン符号や LZW 等の可逆圧縮手法により圧縮化されたソースコード文字列を，サーバ側であらかじめ生成しておき，クライアント側で動的にソースコードを復元し実行する手法である [6]．具体的な処理の流れは以下の通りである．

(1) 開発者は生ソースコード JS_{raw} を圧縮化した文字列 JS_{comprd} を生成する．
(2) 開発者は解凍処理用のコード $decode()$ と JS_{comprd} をセットにした JavaScript コードを生成し，HTML リソースに埋め込んでサーバにデプロイする．
(3) クライアントはサーバに HTTP リクエストを送信しサーバから Web リソースを受け取る．
(4) クライアントは $decode(JS_{comprd})$ を動的に実行し，JS_{raw} を生成してそのまま実行する．

具体的なソースコードを以下に示す．2.1 節で述べた総和計算の生ソースコード（コメント含む）を対象に，可逆圧縮手法としてハフマン符号化を用いた場合，クライアント側に送信される JavaScript コードは以下の通りとなる．

```
1 var decode = function(a) {
2   var d = ...;  // huffman decoding
3   return d;
4 }
5 var a='1B1b1e1f2J2Y1c3K3_1I1X1[1]3V3a1Y2R2S
  1P1S1T3O3P1Q2[2g1L1M3f3h1J3Z3e1E1F3]3^1C2h3
  Y181;1>3'1<2Z3X19242U163S142f3W';
6 eval(decode(a));
```

関数 decode はハフマン圧縮の伸張化関数であり，5 行目の変数 a が圧縮化文字列 JS_{comprd} である．なお，この変数 a にはコメントも全て含まれている．6 行目で伸張化処理を行うと同時に，eval() によって JS_{raw} を動的に実行している．

この手法は，ソースコードをただのテキストと見なしてクライアントに送信し，クライアント側で実行するコードオンデマンドという特性を利用した手法ともいえる．ソースコード中に利用される文字列は，for や if 等の予約語が多くその文字列分布の偏りが多いため，高い圧縮効果が見込める．また，この手法はソースコードをただのテキスト文字列と見なして圧縮化する方法であるため，これまでに述べた最適化手法との併用が可能という利点がある．ただし，復元用コード $decode()$ を追加する必要があるため，短いコードには逆効果である．先のコード例の場合，対象コードが短すぎるため約 5 倍のデータサイズになっている．また，復元用の処理が必要であるため，クライアント側での負荷の増加という副作用がある．

この手法は，HTML や CSS に対しては直接適用することはできない．ただし，HTML/CSS の文字列を，JavaScript 側で動的にレンダリングするような JavaScript コードを生成すれば適用可能ではある．この場合，全ての JavaScript コードをダウンロードし復元してからブラウザへのレンダリングが開始されるため，利用者の大幅な体感速度の低下が発生すると考えられる．

2.5　リソース結合

個別の.js や.css として提供される複数の Web リソースファイルを一つのファイルにまとめることで，HTTP リクエストの数とリソースロード時の冗長な記述を削減できる．例えば以下のような 4 つの JavaScript ファイルをロードする HTML があった場合，

```
1 <script src="jquery.min.js">
2 <script src="jquery-ui.min.js">
3 <script src="jquery-mobile.min.js">
4 <script src="application.js">
```

© 2015 Information Processing Society of Japan

全てを結合した unified.js をロードするように変更することでサイズを削減できる．

```
<script src="unified.js">
```

トラフィックサイズに対する削減効果は小さいものの，クライアント側への副作用はほとんどないという利点を持つ．

2.6 HTTP 圧縮

HTTP/1.1 のオプショナル機能として規定されている通信量削減のための手法である [7][8]．HTTP プロトコルに対するサイズ削減方法であり，これまで紹介した Web リソースそのものに対する処理とは異なるが，高い通信量削減効果を持つ．HTTP のレスポンスボディをサーバ側で gzip や deflate などの可逆圧縮アルゴリズムを用いて圧縮し，クライアント側で伸張化する．

具体的な HTTP ネゴシエーションの流れは以下の通りである．まずクライアントが Accept-Encoding フィールドを加えて HTTP リクエストを送信する．

```
GET /index.html HTTP/1.1
Host: example.com
Acccept-Encoding: gzip
```

この場合，クライアント側では gzip による伸張処理が可能であることを表す．このリクエストに対し，Web サーバは自身が gzip による圧縮処理をサポートしているかを判断する．圧縮処理が可能であった場合，以下のようなレスポンスヘッダと，gzip により圧縮されたレスポンスボディを送信する．

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
```

HTTP はテキストベースプロトコルであるため，gzip や deflate 等による圧縮効果が高い．現在では多くの Web サーバ（Apache HTTP server や IIS，Nginx 等）で標準で搭載されている機能である．しかしながら，これらの Web サーバでは標準で HTTP 圧縮機能が無効化されており，Web サーバ管理者の手によって有効化されないと利用できない．HTTP プロトコルレベルで圧縮化する方法であるため，これまでに述べた Web リソース自体の最適化方法と併用しても，サイズ削減効果が失われにくいという特性がある．サーバとクライアント両者での圧縮処理/伸張処理の負荷が増加するという副作用がある．

2.7 6 つの最適化手法の比較

2.1 節から 2.6 節までの最適化手法の比較を表 1 に示す．HTTP 圧縮以外の全ては，リソースに対して直接変更を加える手法である．そのため基本的にソースコードの可読性が低下するという副作用が発生する．特に論理的構造の最適化と動的自己復元は，元コードの構造が大幅に変更されるため，人が全く理解できなくなるケースが多い．これらの可読性の低下は，いつでも Web リソースのソースコードを確認し参考にできるという，Web のオープンな思想が失われると言い換えることもできる．

HTTP 圧縮と動的自己復元は，クライアント側での負荷の増加という副作用が発生する．ただし動的自己復元は極端なサイズ削減手段であり，実用性は低い．一方で，現在 HTTP 圧縮は多くのサーバで利用されており，またクライアント端末の処理速度の向上もあってその負荷増加はほとんど無視できる程度である．

結論として，ほとんどの最適化手法は可読性の低下という副作用を除けば大きな悪影響はなく，前向きに利用を検討するべきであるといえる．サイズ最適化のツール化に当たっては，この可読性の低下をいかに回避するかが重要となる．

3. サイズ最適化効果の調査

実験では，3 つの著名な JavaScript ライブラリを対象に，3 つのサイズ最適化手法（非論理的表現の削除，論理的表現の省略，および HTTP 圧縮）の適用の効果を確かめた．非論理的表現の削除と論理的表現の省略を行うツールとして YUI Compressor[9] を用いた．なお，各種ライブラリのバージョンは以下の通りである．

- jQuery: ver. 2.1.1
- prototype.js: ver. 1.7.2
- backbone.js: ver.1.1.2

実験の結果を表 2 に示す．生サイズは各種ライブラリの非最適化状態でのファイルサイズであり，YUI，HTTP 圧縮，およびその両方を適用した際のファイルサイズが示されている．また括弧は生のファイルサイズと比べた際のサイズ削減率を示す．

いずれのライブラリに対しても，高い最適化効果が得られることが確認できる．例えば jQuery に対するファイル削減率は，YUI Compressor の適用により 47%，HTTP 圧縮の適用により 70%と，半分以上のサイズ削減が可能である．また両方の最適化を組み合わせた場合は 85%と大幅にサイズが削減されている．

なお，いずれのライブラリも，*.min.js という名前の最適化済みコードを配布してる．さらに，ライブラリ配布用の CDN サーバは既に HTTP 圧縮が有効化されている．すなわち，いずれのライブラリも既にいくつかのサイズ最適化方法を実践しており，ネットワーク全体での無駄なデータ転送の節約に貢献していると言い換えることもできる．

4. サイズ最適化処理の自動化に向けて

本節では，これまでに説明したサイズ最適化手法をいか

© 2015 Information Processing Society of Japan

表 2　JavaScript ライブラリに対するサイズ最適化の効果
Table 2　Effects of size optimization for popular JavaScript libraries

対象 lib	生サイズ	YUI[*1]	HTTP 圧縮	両方
jQuery	241.6 KB	128.2 KB	72.9 KB	37.0 KB
		(47%)	(70%)	(85%)
prototype	193.1 KB	102.3 KB	45.3 KB	33.0 KB
		(47%)	(77%)	(83%)
backbone.js	59.6 KB	19.7 KB	17.3 KB	6.9 KB
		(67%)	(71%)	(88%)

括弧は生サイズと比べた際のサイズ削減率
[*1] YUI Compressor による最適化を適用（非論理的表現の削除，および論理的表現の削除）

にツール化・自動化するかについてのアイデアについて検討する．

4.1　既存ツール

サイズ最適化を行うための様々なツールが Web 上に提供されている．

YUI Compressor[9]: 最もよく知られた JavaScript 最適化ツールである．2.1 節と 2.2 節の最適化（非論理的表現の削除，および論理表現の削除）をサポートする．副作用の少ない最も基本的なサイズ最適化を支援するツールであるといえる．

Closure Compiler[10]: Google が提供する強力なサイズ最適化ツールである．2.1 節から 2.3 節の最適化（非論理的表現の削除，論理表現の削除，および論理的構造の最適化）をサポートする．2.3 節で述べた `alert(55)` のような，アルゴリズムレベルでの強力な最適化を行うことが可能である．ただし副作用が大きく，最適化効果を得るためにはコード記述に対する強い制約を守る必要がある．

packer[6]: JavaScript に対する難読化ツールと記載されているが，その実態は 2.4 節で述べた動的自己復元を行うツールである．ソースコードをただのテキストと見なし，Base62（Base64 文字群から + と − を除いた文字群）を用いた可逆圧縮を適用することによって，難読化とサイズ削減が行われる．

4.2　自動化に向けて

4.1 節で述べたツール類はオンライン用/オフライン用の配布形態をとっており，自由に利用することが可能である．しかしながら，Web 開発者が手作業でソースコードを入力する必要があり，継続的な運用に向いているとは言い難い．また，最適化済みのコードのデプロイは，Web のオープンな思想に反するという側面もある．最適化されたコードは極めて可読性が低く，再利用性は低い．現在の Web はまさにオープンプラットフォームであり，ブラウジングをしながら，いつでも生のソースコードを確認し，その技術を盗むことができる．

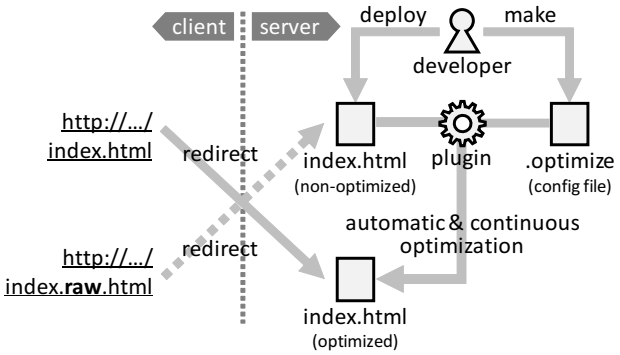

図 1　サイズ最適化プラグインの振る舞い
Fig. 1　Processing flow of size optimization plugin

これらの点からサイズ最適化の自動化には以下の要件が求められると考える．

要件 1: 継続的（手作業を介さず）に処理を施せること．
要件 2: Web のオープンな思想を保つこと．

要件 1 のためには，サイズ最適化処理を Web サーバのプラグインとして提供することが考えられる．プラグインの設定方法としては，`.htaccess` のようなファイルを設置する方式が適する．最適化処理の設定ファイル（`.optimization` 等）を開発者が作成し，コンテンツディレクトリに追加しておくことで，サーバが自動的に処理を施す．これによって，開発者は必要な Web リソースを当該ディレクトリに設置するだけで自動的なサイズ最適化が行われるようになる．

さらに要件 2 に対しては，最適化済みコードと未最適化の生コードを同時に公開するような方法が考えられる．この場合，リソースに対するアクセス URL（あるいはファイル名）をうまくリダイレクトする必要がある．例えば，`index.html` というファイル名は，現在の Web におけるウェルカムページのデファクトとなっており，この名称を変更した `index.min.html` にリダイレクトさせる方法は望ましくない．

要件 1 と要件 2 を満たす，サーバサイドプラグインの振る舞いのアイデアを図 1 に示す．まず，開発者は Web リソースをサーバにデプロイし，さらにサイズ最適化の設定ファイルを当該ディレクトリに設置する．プラグインは自動的

© 2015 Information Processing Society of Japan

にこれら二つのファイルを読み込み，一時的な最適化リソースを生成する．これにより要件 1 の継続的な最適化を達成する．クライアントからの `http://.../index.html` へのアクセスは，この一時コードへリダイレクトさせることで，利用者からは通常の URI アクセスでサイズ最適化済みコードが得られる．さらに，`http://.../index.raw.html` へのアクセスを，生の `index.html` にリダイレクトさせることで，要件 2 のオープンネスを達成できる．結果として，利用者は普段のブラウジング中は常に最適化コードがダウンロードされ，生のコードを確認したい場合には明示的に `*.raw.html` や `*.raw.js` にアクセスすればよい．

5. 議論

5.1 Google が提供する最適化プロキシとの比較

本研究で目的とするリテラシ醸成の対立的な手法として，Google の提供するプロキシサーバを利用する方法 [11] が挙げられる．Android 端末上での Chrome ブラウザでは，Google が提供する最適化プロキシを利用することができる．このプロキシは，Google の提供するサーバを仲介して，2 節で述べたような最適化が自動的に施される．

この方法と比較した際のリテラシ醸成の長所としては，以下のような点が挙げられる．

- Google によるデータ収集を回避できること．
- Web 開発者の意図した柔軟な最適化が可能であること．
- プロキシが不要でありレスポンス速度が速いこと．
- プロキシで最適化不可な SSL 通信にも効果があること．

一方で短所は以下の通りである．

- リテラシ醸成が必要であり即応性がないこと．
- リテラシ醸成が必要であり徹底した最適化が難しいこと．

5.2 サイズ以外の最適化の観点

本稿ではサイズ最適化のみに着目したが，JavaScript の場合，他のプログラミング言語と同様にサイズ以外の様々な最適化の観点が存在する．一般的によく知られる観点は，メモリ最適化や速度最適化である．また，JavaScript はモバイル環境で実行されることも多いため，消費電力という観点からの最適化も考えられる．基本的に，これらの最適化は互いにトレードオフの関係が成り立っており，ある性質を犠牲にして別の性質を最適化することになる．

いずれの最適化も，利用者の体感速度や利用環境に一定の改善効果を与えられるものである．そのため，サイズ最適化だけでなく，他の観点でも最適化するようなリテラシが醸成されるべきだといえる．特に，提供する Web リソースの性質に合わせて最適化を施すことで，Web 全体での通信量の削減や体感速度の向上などの効果が期待できる．

6. おわりに

本稿では，テキスト系の Web リソースに対する最適化リテラシの醸成を目的として，6 種類のサイズ最適化手法を整理し，その内のいくつかの手法について最適化効果に関する予備調査を行った．さらにこの処理の自動化に向けての方針について紹介した．

今後は実際の Web サイトに対する大規模な定量的調査を行い，Web 全体でのリテラシの普及度の調査，およびサイズ最適化を徹底した際の効果を調べる予定である．また自動化方針で述べた `.optimization` の記述方法や，より厳密な最適化トレードオフの調査も今後の予定である．さらにリテラシ醸成のためのエンドユーザ向けサービスについても検討中である．

謝辞 この研究の一部は，科学技術研究費（基盤研究 B 26280115，15H02701，若手研究 B 26730155，萌芽研究 15K12020）の研究助成を受けて行われている．

参考文献

[1] Ihm, S. and Pai, V. S.: Towards Understanding Modern Web Traffic, *Internet Measurement Conference*, pp. 295–312 (2011).
[2] Souders, S.: *High Performance Web Sites -Essential Knowledge for Front-End Engineers-*, O'reilly (2007).
[3] Frederick, G. and Lal, R.: *Optimizing Mobile Markup*, pp. 213–238, Apress (2009).
[4] SitePoint: 19+ JavaScript Shorthand Coding Techniques, `http://www.sitepoint.com/shorthand-javascript-techniques/` (last accessed at December 2014).
[5] MDN: Shorthand properties, `https://developer.mozilla.org/en-US/docs/Web/CSS/Shorthand_properties` (last accessed at December 2014).
[6] Edwards, D.: A JavaScript Compressor. version 3.0, `http://dean.edwards.name/packer/` (last accessed at December 2014).
[7] Liu, Z., Saifullah, Y., Greis, M. and Sreemanthula, S.: HTTP Compression Techniques, *Wireless Communications and Networking Conference*, Vol. 4, pp. 2495–2500 (2005).
[8] Nielsen, H. F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. W. and Lilley, C.: Network Performance Effects of HTTP/1.1, CSS1, and PNG, *Conf. Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 155–166 (1997).
[9] Lecomte, J.: YUI Compressor, `http://yui.github.io/yuicompressor/` (last accessed at December 2014).
[10] Google Inc.: Closure Compiler, `https://developers.google.com/closure/compiler/` (last accessed at December 2014).
[11] Google Inc.: Data Compression Proxy, `https://developer.chrome.com/multidevice/data-compression` (last accessed at December 2014).

© 2015 Information Processing Society of Japan