Web閲覧時におけるJavaScriptライブラリ使用による副作用の調査

山本 将弘[1,a)]　柗本 真佑[1,b)]　楠本 真二[1,c)]

概要：JavaScript には jQuery をはじめとする様々なライブラリが存在している．これらを利用することで，開発者はコードの可読性や再利用性の向上，ブラウザ互換処理の省略といった利点が得られる．その一方で，ユーザに対しては実行時間やネットワーク通信量の増加などの副作用が発生する．本稿では jQuery を題材として，JavaScript ライブラリ使用時に発生する実行パフォーマンスへの副作用について，API の実行時間の観点から調査を行う．調査では，まず jQuery の API について，出現頻度の高い使用方法を調べる．そして，それらの使用方法に対して，同じ動作をする実装を JavaScript ネイティブで作成して実行時間の変化を計測する．また，調査で得られた結果をもとに考察を行う．

キーワード：JavaScript ライブラリ，副作用，jQuery，API，実行時間

1. はじめに

JavaScript は Web ブラウザ上で動作するクライアントサイド言語として，必要不可欠な技術となりつつある．一方で，その記法や文法に対して一定の批判も存在しており，決して開発が容易な言語であるとはいえない．具体的な批判としては，型の比較が直感的でなくバグの要因になりやすい [1]，名前空間の機能がなくグローバルオブジェクトを多用しがちである [2]，複数のコールバック処理が入れ子になり可読性を低下させるコールバック地獄 [3] などが挙げられる．さらに，JavaScript の処理や HTML/CSS の描画がブラウザごとに異なるというクロスブラウザ問題 [4], [5] は，Web の開発における重大な障壁となっている．

これらの問題に対し，様々な JavaScript ライブラリやフレームワークが提供されるようになった．中でも jQuery[6] は Web 全体の内，約 72%で利用されているという報告もあり [7]，高い人気を持つライブラリであるといえる．jQuery は “*write less, do more*” という標語を掲げており，文字通り Web 開発者に対して少ない記述で多くの処理を実現することが可能である．その基本的なアイデアは，JavaScript で多用される典型的な処理（HTML 要素の選択や属性の追加，イベント処理の追加など）をライブラリの提供する API とし，その API の中でブラウザ互換処理などの付随する処理を隠蔽化することにある．例えば，ある HTML 要素（`element`）の背景色を赤に変更する際は，JavaScript ネイティブの記法では `element.style.background-color="red"` という代入式の形で記述する必要があるが，jQuery では `element.css("background-color", "red")` というセッター系 API を呼び出す形で記述可能となる．さらにこの `css(key, value)` の中では，変数 `key` に対して，ブラウザごとの表現の違い（ベンダプレフィックス）を吸収する処理が行われており，容易にブラウザ互換処理が実現できる．近年では，このようなライブラリ以外にも，MVC フレームワーク [8], [9] や，JavaScript を生成する代替言語（altJS）なども登場しており，Web の開発者を支援する技術が浸透しつつある．

このように，JavaScript のライブラリやフレームワークは開発者に様々な恩恵をもたらす一方で，Web の利用者（あるいはブラウザの実行環境）に対しては一定の悪影響を及ぼす．例えば，ライブラリ読み込み時には一定のネットワーク通信が発生し，さらに読み込み後には JIT（Just-in-Time）コンパイラによる処理が発生する．よって，ライブラリに強く依存した Web ページでは，実際に HTML の結果がレンダリングされるまでに一定の遅延が発生することになる．さらには，当然ながら全てのページがライブラリの提供する全ての API を利用するわけではない．すなわち，読み込み時に発生したネットワーク通信とコンパイル処理には無駄が含まれていたといえる．また先の例にも挙げたブラウザ互換処理は，様々なプラット

[1] 大阪大学大学院情報科学研究科
[a)] m-yamamt@ist.osaka-u.ac.jp
[b)] shinsuke@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

フォーム・ブラウザからの利用を想定する開発者にとっては必要不可欠であるものの，単一のブラウザを使うほとんどの利用者にとっては直接的には恩恵がない処理である．

このようなライブラリが生み出す悪影響は，Java や C，Ruby などの他のプログラミング言語でも発生するが，JavaScript では特に発生しやすい問題だと考えられる．これは，Web がコードオンデマンド [10] と呼ばれるアーキテクチャを取るためである．コードオンデマンドは，クライアントの要求に応じてサーバがソースコードを配布し，クライアント側でコンパイル・実行するというアーキテクチャのことを指す．このソースコード配布型という特徴から，Web 上のソースコードはコンパイラによる事前の最適化処理が保障されない．そのため，不要な処理や利用されない API もクライアント側に転送・実行されてしまう．一般的な JavaScript ライブラリは最適化を施した `min.js` という形式で配布されているケースが多いものの，個々の Web ページに記述された JavaScript ソースコードの 9 割以上は最適化されていない [11] のが現状である．

本研究の目的は，JavaScript ライブラリが与えるクライアントへの副作用の実情を明らかにすることである．ここでの副作用とは，JavaScript の開発を容易にするという開発者に対する（主たる）作用に対して，クライアント環境で発生してしまう意図されない作用と定義する．本稿ではこの調査の第一歩として調査対象ライブラリに jQuery を，調査対象の副作用に API の実行時間を選択する．

調査の手順として，まず jQuery の API について出現頻度の高い使用方法を調べる．そして，それらの使用方法に対して同じ動作をする実装を JavaScript ネイティブで作成して実行時間を比較する．調査の結果，jQuery を用いることで実行時間が平均で約 4 倍になることなどがわかった．さらに，今回実施した調査の結果について考察および議論を行う．

2. jQuery

この節では，本稿で調査の題材として選んだ jQuery について，以降の節の説明のために概要と使用例を述べる．

2.1 jQuery の概要

jQuery[6] は HTML の構成要素の検索や操作，アニメーションや非同期通信などの処理を，API を用いて簡単に記述できるようになる JavaScript ライブラリである．W3Techs の調査によると，2017 年 2 月現在，Web サイトのうち 71.9%のサイトにおいて jQuery が使用されている [7]．この使用されている割合は，JavaScript ライブラリの中で最も高い数値である．これは本稿の調査の題材として jQuery を選択した理由である．

jQuery は現在大きく分けて 1.x 系，2.x 系，3.x 系の 3 つのバージョンが存在する．各バージョンが Web サイトにおいて使用されている割合は図 1 の通りである．ほとんどのサイトにおいて 1.x 系が使用されている．それぞれのバージョン間には差異がある．1.x 系と 2.x 系の間，2.x 系と 3.x 系の間それぞれの主な変化をまとめたものが表 1 である．共通する変化の流れとして，ブラウザの古いバージョンに対する互換処理が削除されていくという点がある．

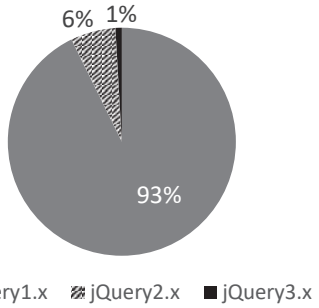

図 1　使用されているバージョンの割合 [7]

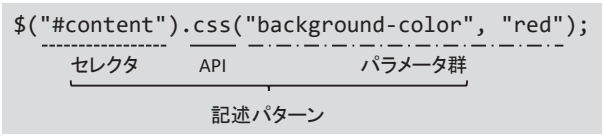

図 2　API 呼び出しの例

2.2 jQuery の使用例

jQuery のライブラリを読み込むことでライブラリ内の実装は「`jQuery`」または「`$`」で参照できるオブジェクトに格納される． このオブジェクトに対して API を呼び出した例が図 2 である．この呼び出しは HTML の構成要素のうち id が「`content`」であるものの背景色を赤色にするというものである．「`$`」に対して引数を与えると，引数に対応する HTML の構成要素を表す jQuery オブジェクトを返す．この部分の記述をセレクタ部とし，引数のことをセレクタと呼ぶことにする．図の例ではセレクタは「`"#content"`」という文字列である．文字列 1 文字目の「`#`」は id を表す記号である．class を表すのは「.」であり，何も記号を付けない場合は構成要素のタグ名を表す．また，引数として与えられるのは文字列ではなく jQuery オブジェクト自体である場合もある．例のセレクタ部では id が「`content`」である構成要素を表す jQuery オブジェクトが返されることになる．この jQuery オブジェクトに対して，引数のパラメータ群とともに API を呼び出す．例では，構成要素のスタイルを指定する API である「`css`」に引数として背

表 1　jQuery のバージョン間の主要な変化

2.x 系での主な変化	• IE6，7，8 の互換処理が削除される • カスタムビルドができるようになる
3.x 系での主な変化	• IE 以外のブラウザで新しいバージョン以外の互換処理が削除される • API の実装の見直しが行われる

景色を表す「background-color」と赤色を表す「red」を与えて呼び出している．jQuery オブジェクトに対しては「$("#aaa").xxx("bbb").yyy("ccc");」のように連続して記述して複数の API を呼び出すことも可能である．

また，セレクタ・API・パラメータ群の組み合わせのことを記述パターンと呼ぶことにする．記述パターンは複数の API を組み合わせたものである．jQuery ではこの記述パターンの形で呼び出されることが多い．そのため，以降の調査はライブラリの API の例として，jQuery の記述パターンを扱う．

3. API 使用による副作用の調査

3.1 調査概要

本調査の目的は，ライブラリの API を呼び出すことによる実行時間の変化を調べることである．1 つの記述パターンについての計測につき，比較するための実装を用意する必要がある．この実装は jQuery の API に対して同等の機能を，ブラウザ互換などの処理を行わずに JavaScript の標準の関数群を用いた実装である．以降，この実装をネイティブの実装と呼ぶことにする．jQuery の記述パターンの呼び出し方は多様であるので，全ての記述パターンに対してネイティブの実装を用意するのは困難である．計測する記述パターンの数を限定するのであれば，使用頻度の高い記述パターンについて計測するのが妥当である．そこでまず，ユーザのアクセス数の多いサイトにおいて使用頻度の高い記述パターンを調査する．そして，使用頻度の高い 10 個の記述パターンに対してネイティブの実装を用意し，実行時間を比較する．比較する jQuery のバージョンについては 1.x 系，2.x 系，3.x 系から 1 つずつバージョンを選択して計測を行う．

以上で述べたことを調査項目にまとめる．本調査では 2 つの調査項目を設けて回答を行う．

RQ1: 使用頻度の高い jQuery の記述パターンは何であるか

RQ2: 使用頻度の高い jQuery の記述パターンを用いることで実行時間がどの程度変化するか

これらの調査項目のそれぞれについて，調査方法と結果を各小節で述べる．

3.2 RQ1: 使用頻度の高い記述パターンの調査

この小節では，ユーザのアクセス数の多いサイトにおいて，使用頻度の高い jQuery の記述パターンを調べる方法とその結果について述べる．調査の流れを図 3 に表す．

3.2.1 調査方法

ユーザのアクセス数の多いサイトは，Alexa 社の提供しているアクセス数上位 100 万サイトのリストを用いる．各サイトのページはサイトに属する全てのページを考慮すると膨大なページ数になるため，本調査では各サイトのトップページに対して集計を行うこととする．100 万サイトのうちアクセスできるサイトのトップページのソースコードをデータベースに格納する．以降の集計作業はデータベース上のコマンドで行う．

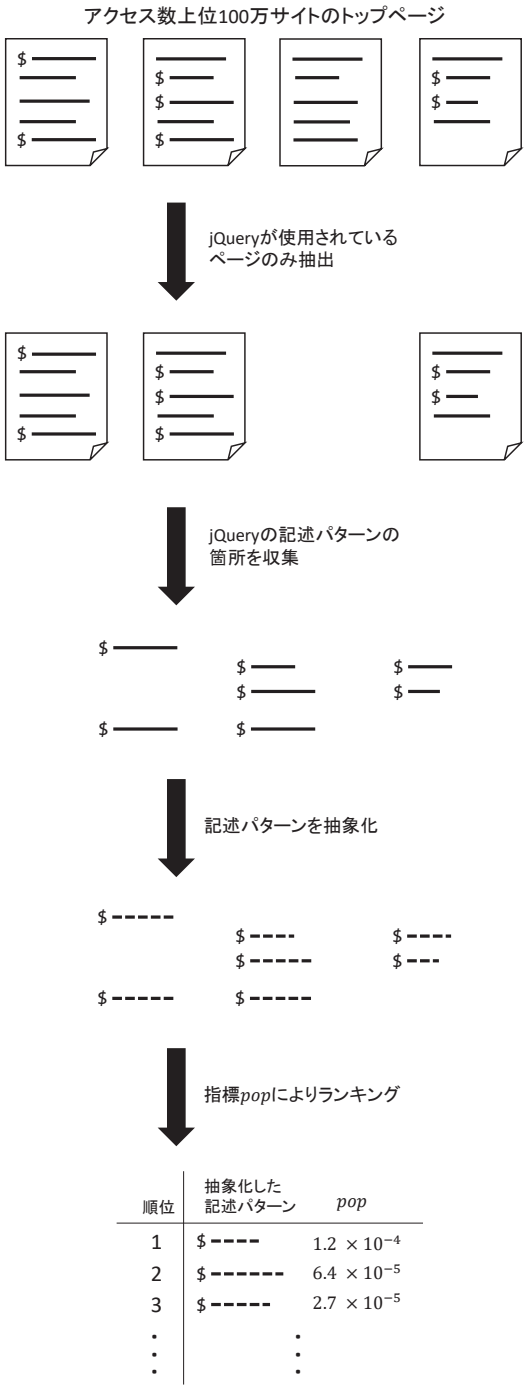

図 3 使用頻度の高い記述パターンの調査の流れ

アクセス数上位 100 万サイトのうち，jQuery を使用しているかどうかでフィルタリングを行う．これはトップページのソースコードに「jquery」と「.js」という記述があるかどうかにより判断する．このフィルタリングを行ったところ，jQuery を使用しているサイトは全体の約 67 %の 665,558 サイトであった．

次にどのような記述パターンが呼び出されてい

るかを集計する．API の記述パターンは正規表現で「`$\(.+?\)\..+?\(.*?\)`」と表される．正規表現に該当する記述パターンの中で，セレクタ，API，パラメータ群はそれぞれ変化する．特にページごとに変化をするのはセレクタとパラメータ群である．ページごとに変化がある状態では，各ページで同じ用途で使用されている記述パターンを同一視することができない．そこで集計の前にセレクタとパラメータ群に対して抽象化を行う．セレクタについては「`"#"`」で始まるセレクタを `id`，「`".`」で始まるセレクタを `class`，その他「`"`」で始まるセレクタを `tag`，「`"`」以外で始まるセレクタを `variable` として抽象化する．パラメータ群についてはコンマ（`,`）で区切った文字列それぞれをパラメータとする．そして，「`"`」で始まるパラメータを `string`，「`function`」で始まるパラメータを `function`，それ以外のパラメータを `variable` として抽象化する．API についてはそのまま使用する．抽象化したセレクタ，API，抽象化したパラメータの組み合わせを抽象化した記述パターンと呼ぶことにする．そして記述パターンに対して使用頻度が高い順にランキングをつける．

出現回数の合計だけで順位付けを行うと，特定のページでのみ何度も使用される記述パターンが上位に来る可能性がある．そのため，多くのページで使われているほど大きな値を取るような指標で順位付けを行う必要がある．そこで，情報検索の統計値である tf-idf[12] を参考にした *pop* という指標を提案する．*pop* および *pop* の導出に必要な変数の定義は以下である．

記述パターンの集合を $\{pattern_i\}$ とする．ある記述パターン $pattern_i$ を使用しているページの数 $upage_{pattern_i}$ は以下で表される．

$$upage_{pattern_i} = |\{page : page \ni pattern_i\}| \quad (1)$$

jQuery が使われているページの集合を $jpage$ とする．jQuery が使われているページのうち，記述パターン $pattern_i$ が出現するページの割合 $df_{pattern_i}$ は以下で表される．

$$df_{pattern_i} = \frac{upage_{pattern_i}}{|jpage|} \quad (2)$$

ページ $page$ における記述パターン $pattern_i$ の出現回数を $n_{page,pattern_i}$ とする．ページ $page$ における全記述パターンのうちの $pattern_i$ の割合 $rate_{page,pattern_i}$ は以下で表される．

$$rate_{page,pattern_i} = \frac{n_{page,pattern_i}}{\sum_k n_{page,pattern_k}} \quad (3)$$

jQuery が使われているページにおける $rate_{page,pattern_i}$ の平均値 $tf_{pattern_i}$ は以下で表される．

$$tf_{pattern_i} = \frac{1}{|jpage|} \sum_{page \in jpage} rate_{page,pattern_i} \quad (4)$$

tf-idf の場合，あるドキュメント（ここではページ）での特徴的な語（記述パターン）を抽出するために，*df* の逆数を用いる．一方で本稿では多くのページで出現する記述パターンを抽出するために，逆数を取らないこととする．よって，$pop_{pattern_i}$ を以下の計算式で定義する．

$$pop_{pattern_i} = df_{pattern_i} \cdot tf_{pattern_i} \quad (5)$$

以上の定義により，*pop* の値は呼び出される回数の合計が多く，さらに多くのページで出現するほど大きな値になる．これにより特定のページでのみ何度も呼び出されている記述パターンのランキングへの影響を抑えることができる．抽象化したそれぞれの記述パターンに対して *pop* の値を計算し，降順にソートして順位付けを行う．

3.2.2 調査結果

データベース上で集計をして，*pop* の値が降順になるようにランキング付けを行った結果が表 2 である．最も *pop* の値が大きいのは「`$(variable).attr(string)`」であった．これは `variable` が表す HTML の構成要素に対して，ある属性を取得する記述パターンである．実際のソースコードを見てみると，この記述パターンにより「`this`」が表す構成要素の「`value`」という属性の値を取得する際などに使用されていた．

他に表から読み取れることとして，「`Class`」が付いた API が多いことがわかる．「`addClass`」は構成要素に class を追加する API であり，「`removeClass`」は class を削除する API である．

ランキング表において 5 番目に *pop* の値が大きい「`$(variable).ready(function)`」はブラウザが構成要素の読み込みを完了したときに呼び出す関数を指定する API である． 指定する関数で行う処理はページごとに大きく異なる．RQ2 では使用頻度の高い記述パターンに対して実行時間の計測を行うが，この API は関数の内容により実行時間が大きく変わるため計測が困難である．よって「`$(variable).ready(function)`」を除いた 9 つの記述パターンに対して，RQ2 で実行時間への影響の調査を行う．

表 2 抽象化した記述パターンのランキング

順位	抽象化した記述パターン	*pop*	計測対象
1	`$(variable).attr(string)`	1.3×10^{-4}	✓
2	`$(variable).find(string)`	7.4×10^{-5}	✓
3	`$(variable).addClass(string)`	5.9×10^{-5}	✓
4	`$(id).addClass(string)`	5.1×10^{-5}	✓
5	`$(variable).ready(function)`	4.8×10^{-5}	
6	`$(id).html(variable)`	4.5×10^{-5}	✓
7	`$(id).css(string, string)`	4.2×10^{-5}	✓
8	`$(class).removeClass(string)`	3.9×10^{-5}	✓
9	`$(class).addClass(string)`	3.5×10^{-5}	✓
10	`$(class).css(string, string)`	2.9×10^{-5}	✓

RQ1 への回答をまとめると以下のようになる．

RQ1「使用頻度の高い jQuery の記述パターンは何であるか」への回答

- 使用頻度の高い 10 の記述パターンは表 2 の通り
- 最も使用頻度の高い記述パターンは「`$(variable).attr(string)`」
- class に関する API が多い

3.3 RQ2: 記述パターンの実行時間への影響の調査

この小節では，RQ1 で調べた使用頻度の高い記述パターンについて，実行時間がどの程度変化するかを調べる方法とその結果について述べる．

3.3.1 調査方法

記述パターンの実行にかかる時間を計測するのに必要なものとして，記述パターンを実行する HTML がある．実在の Web ページを計測のために使用するとすると，抽象化したセレクタやパラメータを再度具象化する必要があり，その作業は困難である．そこで本調査では多数の構成要素からなる計測用 HTML を作成する．HTML の構成要素は木構造であり，2 つの要素のみからなるとき，図 4 のように入れ子構造となる場合と並列構造となる場合がある．そこで用意する計測用 HTML として入れ子構造と並列構造をそれぞれ繰り返した 2 種類の HTML を作成する．また，記述パターンのセレクタやパラメータによって構成要素を指定できるように，それぞれの構成要素に id や class を付与する．id や class の付与をランダムに行った HTML を入れ子構造と並列構造それぞれで 10 個ずつ，計 20 個の HTML を用意する．そして，それぞれの HTML に対して前節で得られた使用頻度の高い記述パターンを実行する．

記述パターン 1 回あたりの実行時間は非常に短いため，比較に十分な値になるまで繰り返し実行して時間を計測する．予備実験の結果，計測用 HTML の要素数を 5 万にすれば記述パターンの実行時間が比較に必要な値になることがわかった．この HTML の構成要素に対して id や class を指定して繰り返し記述パターンを実行することを 1 セットの実行とする．ブラウザで JavaScript の実行を開始した直後は様々な要因によって多く時間がかかることがある．そこで，要因による影響を減らすために初めに 5 セット計測をしない実行をしたのち，10 セット実行して計測を行う．そして，10 セットの中央値を実行時間の計測結果とする．この計測手法は JavaScript のパフォーマンスに関する調査論文 [13] を参考にしている．

表 3　比較する jQuery のバージョン

バージョン	ファイルサイズ	リリース日
1.12.4	287KB	2016 年 5 月 20 日
2.2.4	252KB	2016 年 5 月 20 日
3.1.1	261KB	2016 年 9 月 22 日

```
<div>
  <div>
  </div>
</div>
```

(a) 入れ子構造

```
<div></div>

<div></div>
```

(b) 並列構造

図 4　2 つの要素の関係

```
for (var i = 1; i <= 25000; i++) {
    $("#i" + i).addClass("c" + (i + 5000));
}
```

図 5　記述パターンに対する jQuery の実装

```
for (var i = 1; i <= 25000; i++) {
    document.getElementById("i" + i)
        .classList.add("c" + (i + 5000));
}
```

図 6　記述パターンに対するネイティブの実装

先の実験で得られた使用頻度の高い記述パターンそれぞれについて，計測用 HTML の構成要素に対して記述パターンを実行するプログラムを作成する．記述パターン「`$(id).addClass(string)`」に対して jQuery を使用して実装したプログラムとネイティブの実装の例が図 5，6 である． どちらも id が付与された HTML の構成要素に新たな class を追加するという処理を繰り返している．図 5 の jQuery の実装では記述パターンをそのまま使用している．図 6 のネイティブの実装では getElementById などの標準の関数を組み合わせて目的の処理を実装している．

比較する jQuery のバージョンは表 3 の通りである．1.x 系，2.x 系，3.x 系のうちそれぞれの最新のバージョンを使用する．また，RQ2 の調査を行う環境を表 4 にまとめる．

3.3.2 調査結果

各バージョンごとに記述パターンの実行時間を計測してネイティブの実装の実行時間と比較した結果をまとめたものが図 7，8 である．棒グラフの縦軸が表す値は，ネイティブと jQuery の 3 バージョンそれぞれの実装の実行時間をネイティブの実装の実行時間で割った値，すなわち倍率である．ネイティブの値は常に 1 になるが，比較のためにグラフに追加している．また，横軸の数字は RQ1 で調査した使用頻度の高い記述パターンの順位と対応している．

入れ子・並列の両構造それぞれのグラフにおいて，値が

表 4　RQ2 の調査環境

Web ブラウザ	Google Chrome 55.0.2883.87
OS	Windows 10 Pro
CPU	Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz

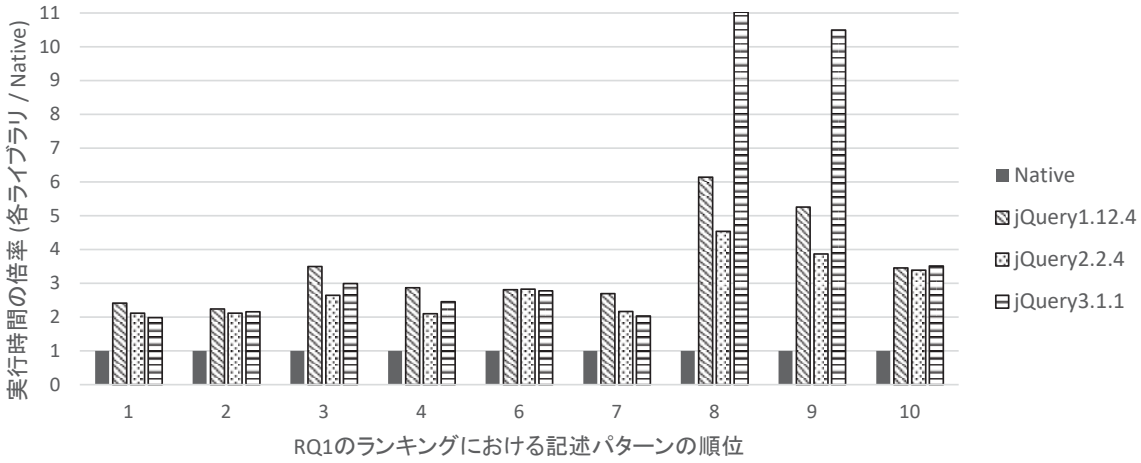

図 7 入れ子構造の計測用 HTML に対する各記述パターンの実行時間

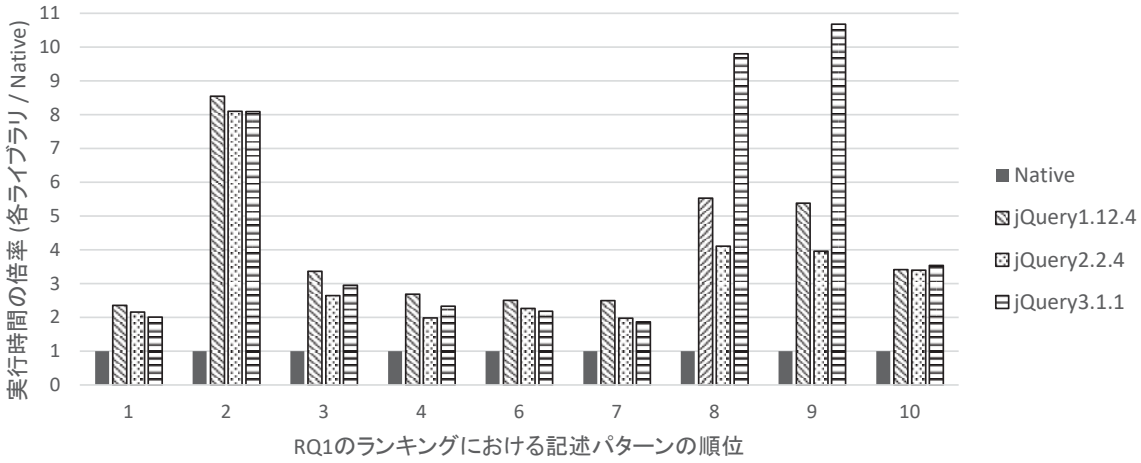

図 8 並列構造の計測用 HTML に対する各記述パターンの実行時間

1 を下回ることはなかった．これは，jQuery の記述パターンを用いるとネイティブの実装に比べて常に実行時間が増加することを意味する．

グラフのうち，最大の値を取ったのは記述パターン「`$(class).removeClass(string)`」を入れ子構造の計測用 HTML に対して jQuery3.1.1 から呼び出したときである．ネイティブの実装と比べて 11 倍の実行時間を要した．同様に jQuery3.1.1 から呼び出したときに高い倍率を取る記述パターン「`$(class).addClass(string)`」との共通点として，セレクタが class であり，API も class に対する処理であることが挙げられる．

また，同じ記述パターンに対する入れ子構造と並列構造のグラフを比較すると，「`$(variable).find(string)`」の場合を除いてほぼ等しい値を取ることが分かる．

全体の平均として，jQuery から記述パターンを呼び出した場合，ネイティブの実装と比べて 3.9 倍の実行時間がかかることがわかった．jQurey の各バージョンで比較すると，jQuery2.2.4 の実行時間が最も短いことがわかった．

RQ2 への回答をまとめると以下のようになる．

RQ2「使用頻度の高い jQuery の記述パターンを用いることで実行時間がどの程度変化するか」への回答

- ネイティブと比較して平均で約 4 倍に増加した
- ネイティブよりも実行時間が短くなることはなかった
- jQuery の各バージョン間では 2.2.4 の実行時間が最も短かった

4. 考察

前節で得られた結果について，グラフをもとに整理して，考えうることについて述べる．

4.1 API の実行時間の観点における jQuery 利用の是非

入れ子・並列の両構造それぞれのグラフから，jQuery の記述パターンを用いることで実行時間が常に増加することがわかる．実験を始める前の段階で予想した通り，ブラウザ互換などの処理によりネイティブの実装と比べて多くの実行時間を要しているものと思われる．

また，ネイティブの実装と比較して jQuery の記述パターンを用いた場合の実行時間が平均で 3.9 倍となった．これは，Web サイトを作成する際ネイティブの実装で事足りるならば，jQuery から API を呼び出す場合と比べて実行時間を 4 分の 1 にすることができる可能性があることを表す．Web ブラウザでページが表示されるまでの時間やクリック時などのアクションにかかる時間を短縮したいのであれば，ネイティブの実装を行う検討の余地があるといえる．

4.2 class 関係の API において jQuery3.x 系が最も大きい値を取る理由

全てのグラフの中で入れ子構造の計測用 HTML に対して記述パターン「`$(class).addClass(string)`」を jQuery3.1.1 から呼び出したときに最大の値となった．バージョンが jQuery2.2.4 の場合は半分以下の 4.5 倍である．jQuery3.x 系は jQuery2.x 系に対して実装の見直しを行っているため，実行時間は同程度か少し短くなると考えていただけに，予想に反する結果となった．この結果について調べるために，`addClass` と `removeClass` の実装を jQuery の各バージョンごとに比較した．jQuery1.12.4 と jQuery2.2.4 間では大きな変化はなかったのに対し，jQuery2.2.4 と jQuery3.1.1 間では正規表現などの処理が変更されていることがわかった．セレクタが class の場合において大きな差が生じているため，複数の構成要素に対する実行に対して変更の影響が出たのではないかと考えられる．

4.3 一部の記述パターンにおいて構造により大きく結果が異なる理由

同じ記述パターンに対する入れ子構造と並列構造のグラフを比較すると，他がほぼ等しい値を取るのに対して「`$(variable).find(string)`」のみ大きな差が生じた．「`find`」は HTML の構成要素を検索する API である．そのため，実装方法と構造の関係により差が生じたものと考えられる．

5. 妥当性への脅威

この節では本調査の結果の妥当性に影響を及ぼす恐れのある問題について述べる．

5.1 内的妥当性

本調査ではサイトで jQuery が使用されているか確認や記述パターン収集に正規表現を用いている．本来は調査の対象にすべき記述が正規表現にマッチしないことにより対象にできていない可能性がある．

本調査では記述パターンを実行するページとして，計測用 HTML を作成した．もし実際の Web サイトの HTML に対して計測を行うと異なる結果になる可能性がある．しかしながら，作成した HTML は構成要素の関係のうち入れ子構造と並列構造を繰り返した両極端の HTML によって再現した．両方の HTML で jQuery を使用している場合と使用していない場合の実行時間に差異が生じているため，結果が大きく変化することはないと考えられる．

本調査では記述パターンの実行時間を計測する際に，1 回あたりの実行時間では短すぎるため，記述パターンを繰り返し実行する際に要する時間を計測している．もしこの繰り返し行う実行が実装方法によってはブラウザにより最適化される場合，計測結果が誤っている可能性がある．これについて Chrome のデベロッパーツールを用いて調べたが，最適化が行われているかどうかを確認することはできなかった．

5.2 外的妥当性

本調査では JavaScript ライブラリを使用したことによる副作用を計測するために jQuery を題材とした．もし jQuery 以外の JavaScript ライブラリを題材として計測したときに異なる結果になる可能性がある．

本調査では使用頻度の高い記述パターンを集計する対象としてアクセス数の多いサイトのトップページのみを選択した．もしトップページ以外も対象とした場合は異なる計測結果になる可能性がある．しかしながら，1 つのサイトには膨大なページが存在するため，それらをすべて対象とするのは困難であるというのは 3 節で述べた通りである．

本調査では実行時間の計測に使用するブラウザとして Google Chrome を用いた．もし他のブラウザを使用して計測を行った場合，異なる計測結果になる可能性がある．

6. おわりに

本稿では，JavaScript ライブラリを使用することによる副作用に関して，jQuery を題材として API の実行時間の変化について調査した．実験の結果，jQuery の記述パターンを使用することにより，ネイティブの実装と比較して平均で約 4 倍の実行時間がかかることなどがわかった．

今後の計画としては，ネットワーク通信量といった，API の実行時間以外の副作用について調査を行う．また，JavaScript ライブラリについては，jQuery 以外のライブラリに対しても調査を行う予定である．複数の調査の結果をもとに明らかになった問題点に対して，解決案を提案することを目標として掲げている．

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究（S）（課題番号：JP25220003），および文部科学省研究費補助金若手研究（B）（課題番号：JP26730155）の助成を得て行われた．

参考文献

[1] Pradel, M. and Sen, K.: The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript, *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015).

[2] Crockford, D.: *JavaScript: The Good Parts: The Good Parts*, " O'Reilly Media, Inc." (2008).

[3] Alimadadi, S., Mesbah, A. and Pattabiraman, K.: Understanding asynchronous interactions in full-stack JavaScript, *Proceedings of the 38th International Conference on Software Engineering*, ACM, pp. 1169–1180 (2016).

[4] Mesbah, A. and Prasad, M. R.: Automated cross-browser compatibility testing, *Proceedings of the 33rd International Conference on Software Engineering*, ACM, pp. 561–570 (2011).

[5] Ochin, J. G.: Cross Browser Incompatibility: Reasons and Solutions, *International Journal of Software Engineering & Applications (IJSEA)*, Vol. 2, No. 3 (2011).

[6] The jQuery Foundation: jQuery, The jQuery Foundation (online), available from ⟨http://jquery.com/⟩ (accessed 2017-02-07).

[7] Q-Success: Usage Statistics and Market Share of jQuery for Websites, February 2017, Q-Success (online), available from ⟨https://w3techs.com/technologies/details/js-jquery/all/all⟩ (accessed 2017-02-07).

[8] Google: AngularJS – Superheroic JavaScript MVW Framework, Google (online), available from ⟨https://angularjs.org/⟩ (accessed 2017-02-10).

[9] Ashkenas, J.: Backbone.js, DocumentCloud (online), available from ⟨http://backbonejs.org/⟩ (accessed 2017-02-10).

[10] Fielding, R. T.: Architectural styles and the design of network-based software architectures, PhD Thesis, University of California, Irvine (2000).

[11] Matsumoto, S. and Nakamura, M.: A Preliminary Study of Size Optimization for Text-Based Web-Resource, *Empirical Software Engineering in Practice (IWESEP), 2016 7th International Workshop on*, IEEE, pp. 36–40 (2016).

[12] Salton, G. and McGill, M. J.: Introduction to modern information retrieval (1986).

[13] Selakovic, M. and Pradel, M.: Performance issues and optimizations in JavaScript: An empirical study, *Proceedings of the 38th International Conference on Software Engineering*, ACM, pp. 61–72 (2016).