一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
MSS2018-79,SS2018-50(2019-01)

JavaScript フレームワーク比較支援のための プレイグラウンド型ツールの試作

中島　　望† 　柗本　真佑† 　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565–0871 大阪府吹田市山田丘 1–5
E-mail: †{n-nakajm,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　JavaScript フレームワークとは，Web のフロントエンド開発を効率化するための枠組みである．大規模な Web アプリケーションを開発する際にはフレームワークを使用することが一般的であり，開発者は使用するフレームワークを適切に選択する必要がある．しかし，適切なフレームワークの選択は開発者にとって容易ではない．フレームワークはアーキテクチャパターンや記法といった実装面及び処理時間等の性能面でそれぞれ特徴がある．比較の際にはこれらの特徴を把握し，総合的に判断する必要がある．またフレームワークは日々新しく誕生しており，改良されたものが継続的にリリースされている．そのため，実装や性能の単純な比較情報はすぐに古くなってしまう．本研究ではソースコードの比較と処理時間の計測が可能なプレイグラウンド型のフレームワーク比較ツールを試作した．提案ツールはブラウザ上で動作し，DOM 操作や Ajax 通信といった機能の実装例を実装面と性能面の両方で比較できる．また予備実験として，提案ツールを利用し，複数のフレームワークによる実装のソースコード及び処理時間を比較した．実験の結果，本ツールを用いてフレームワークの実装面および性能面を比較できることが示された．

キーワード　JavaScript フレームワーク，フロントエンド開発，プレイグラウンド

1. はじめに

ブラウザ上で動作する JavaScript は，Web のフロントエンド開発における中心言語となっている [1]．世界中の全 Web サイトのうち，95%が JavaScript を使用しているという調査結果がある [2]．Web 周辺技術の進化および爆発的な需要の増加に伴い，Web アプリケーションのソースコードは複雑化の一途を辿っている [3]．ソースコードの複雑化を改善し，開発効率や保守性を向上するために，JavaScript フレームワーク（以下，JSF）を用いた Web 開発が広く行われている [3] [4]．JSF の具体例としては Vue.js や Angular が知られている．各 JSF は MVC や MVVM といった様々なアーキテクチャパターンを採用しており，JSF のパターンに基づいてアプリケーションを構造化する [5]．またアプリケーションの開発時に必要とされる機能が予め関数として用意されており，それを利用することでソースコードを簡略化できる [3]．開発者は JSF を適用することで，高品質なソースコードを効率的に開発できる．

JSF は開発面での品質向上を目的としているため，パフォーマンスとのトレードオフが存在することも知られている [6]．そして適用する JSF の選択，すなわちアーキテクチャパターンの選択においては，プログラミング言語やパラダイムの選択と同様に正解は存在しない．開発するアプリケーションの特性や，開発者達の好み，スキル等の条件に合うものを選ぶ必要がある [3]．また機能を提供するライブラリは特定の機能を別ライブラリと容易に差し替えられるが，全体の構造を規定する JSF は開発途中での変更が容易ではない．よって高品質なソースコードと高機能なアプリケーションを実現するためには，JSF の実装面及び性能面の特徴を把握し，開発するアプリケーションに応じた適切な JSF の選択が重要である [3]．

一方で適切な JSF の選択は容易ではない．JSF の選択を難しくする要因として，以下の点が挙げられる．一つ目は，JSF の選択肢の豊富さである．同じアーキテクチャパターンを採用している JSF が複数存在しているため，採用したいアーキテクチャパターンから一つの JSF を選び出すことは難しい．例えば MVVM というアーキテクチャパターンを採用している JSF には Vue.js や Knockout.js 等が挙げられる．同じアーキテクチャパターンを採用する JSF でもそれぞれ記法は異なるため，比較時には各 JSF の記法の違いを把握する必要がある．

二つ目は，実行環境の多様さである．Web アプリケーションの特性上，Google Chrome や Firefox といったブラウザ，スマートフォンや PC といったデバイスの組み合わせにより，多数の実行環境が想定される．JSF は実行環境によらず一様に動作させるための Polyfill(注1) を含んで実装されているが [3]，JSF

(注1)：ブラウザ間の仕様の違いに対応するために埋め込まれるソースコードを指す．

の性能は実行環境によって異なるという指摘も存在する [4]．比較時には各 JSF の実行環境による性能面の違いを把握する必要がある．

三つ目は，情報入手の難しさである．Web 技術は成長が早く，JSF のみならず実行環境であるブラウザもアップデートが頻繁にリリースされる．例えば Google を中心に開発されている Angular は，2018 年 5 月にバージョン 6.0.0 がリリースされた後，5 ヶ月後の 11 月にバージョン 7.0.0 がリリースされている [7]．バージョンの更新の際には JSF の記法の変更やパフォーマンスの改善が行われることが多く，開発者は利用したい条件での情報を得ることが難しい．

本稿では開発者に対する JSF の選択の支援を目的として，JSF を実装面及び性能面で比較するプレイグラウンド型ツール jact を試作する．jact は一つ目の課題に対応するためにタスクベースのコード比較機能を，二つ目の課題に対応するためにオンデマンドな性能計測機能を持つ．そして三つ目の課題に対応するためにツールをプレイグラウンド化し，JSF についての情報を共有可能にする．また予備実験として jact を用いて 3 つの JSF を 6 つの環境下で比較し，ソースコード及び処理速度の比較によってどのような情報を得ることができるのかを確かめた．Web 開発において汎用的な機能を実現するソースコードを JSF を用いて実装し，比較した．実験の結果，jact を用いて実装面及び性能面の違いを比較できることを確認できた．

2. 準　　備

2.1 プレイグラウンド

ブラウザ上でソースコードを編集し，実行できるサービスをプレイグラウンドと呼ぶ．代表的な JavaScript のプレイグラウンドとして，JSFiddle や CodePen が挙げられる．利用者は実行環境を準備する必要がないため，ソースコードの動作をブラウザ上で手軽に確かめることができる．またプレイグラウンドを利用してソースコードを共有することで，ソースコードだけでなくその実行環境も共有できる．

2.2 JavaScript フレームワーク

JavaScript フレームワーク（JSF）とは，Web のフロントエンド開発を効率化するために使用されるソースコードの枠組みである．一般にソースコードに対して部分的に機能を付与する目的で利用されているライブラリとは異なり，ソースコード全体に MVC 等のアーキテクチャを付与する目的で利用されている [8]．開発者は選択した JSF の記法に従ってアプリケーションを作成する．JSF はアーキテクチャパターンや記法，利用できる関数にそれぞれ特徴がある．JSF の利用によってソースコードを構造化したり [5]，あらかじめ定義されている様々な関数を利用したりすることができるため [3]，コード品質の向上や開発の効率化が期待できる．

例として Vue.js と Knockout.js のソースコードを図 1 に示す．図 1 は Web ページに文字列"Hello!"を出力するソースコードである．Vue.js と Knockout.js は MVVM というアーキテクチャパターンを採用している．MVVM は画面上の表示（以下，View）とアプリケーションのデータ（以下，Model）を紐付ける．通常 JavaScript では View もしくは Model の一方が変更された際，もう一方に変更を反映させる処理が必要になる．MVVM は View と Model の紐付けによって，変更の反映処理が不要になる．これによって，開発者はインタラクティブなインターフェースの作成を容易に行うことができる．

図 1: "Hello!"を表示するソースコード

(a) Vue.js

```
<div id="app">
  {{ msg }}
</div>
<script>
  var app = new Vue({
    el: '#app',
    data: {
      msg: 'Hello!'
    }
  })
</script>
```

(b) Knockout.js

```
<span data-bind="text:m"></span>
<script>
  ko.applyBindings({
    m: 'Hello!'
  });
</script>
```

2.3 フレームワーク選択における課題

JSF の利用は開発者にとって様々なメリットがある一方で，適切な JSF の選択は容易ではない．JSF の選択における課題点として，以下の点が挙げられる．

2.3.1 P_1：選択肢の豊富さ

JSF によって採用しているアーキテクチャパターンは様々であり，例として MVC，MVVM が挙げられる．同じアーキテクチャパターンを採用している JSF は複数存在するが，ソースコードの記法は JSF によって異なる．例として同一のアーキテクチャパターンを採用している Vue.js と Knockout.js を挙げる．図 1 に示した通り，Vue.js は HTML 中に変数を埋め込むが，Knockout.js では HTML 中に変数を埋め込まずに HTML タグ内に紐付けるデータを明示する．このように，開発者はアーキテクチャパターンを選択すれば JSF を決定できるのではなく，それぞれの JSF の記法を把握して比較する必要がある．

2.3.2 P_2：実行環境の多様さ

Web アプリケーションの特性上，実行環境はユーザによって様々である．ユーザがどのブラウザを使用するのか，どのデバイスで使用するのかという 2 つの要素によって実行環境が構成される．Gizas らの研究 [4] や Mariano の研究 [5] で示される通り，実行環境によって JSF が発揮できる性能は異なる．

2.3.3 P_3：情報入手の難しさ

JSF そのもの，そして実行環境であるブラウザも頻繁にアップデートがリリースされる．例えば JSF の一つである Angular はわずか 5 ヶ月でバージョンが更新され，新機能の追加やパフォーマンスの改善が行われた [7]．JSF の比較研究は Gizas らの研究 [4] 等で実施されているが，実際に開発者が JSF の比較を行う際には比較情報が古くなっている場合が多く，自分が利用したいバージョンの情報を得ることが難しい．

3. 提案手法：jact

3.1 概　　要

本稿では前章で述べた課題の解決を目的とした JSF 比較ツー

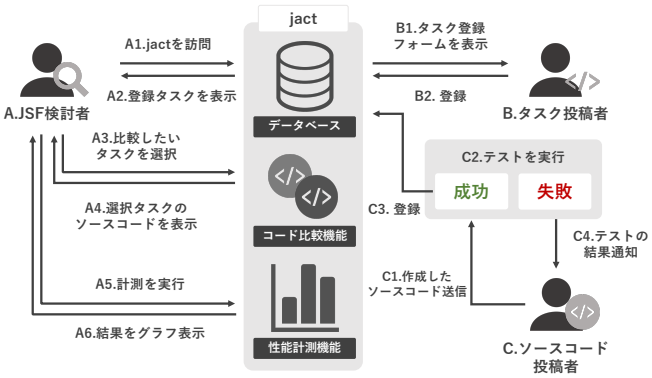

図 2: jact の利用の流れ

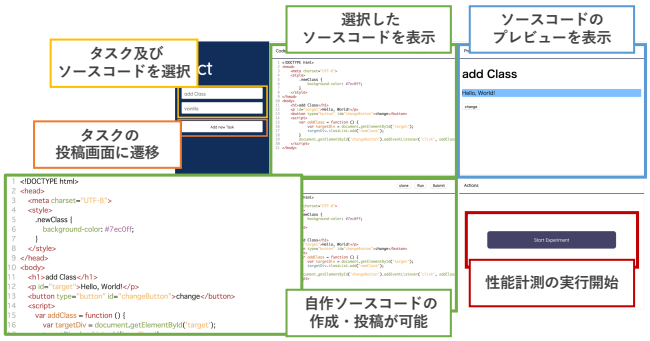

図 3: コード比較画面

ル jact を提案する．提案手法は課題 P_1 に対応するためにタスクベースのコード比較機能を，課題 P_2 に対応するためにオンデマンドな性能計測機能を持つ．また課題 P_3 に対応するためにツールをプレイグラウンド化する．プレイグラウンド化のためにアクタによる投稿機能とソースコードのテスト機能を持つ．jact の利用によって JSF の実装面及び性能面の特徴が比較可能になり，開発者の JSF 選択の支援が期待できる．

3.2 利用の流れ

jact の利用の流れを図 2 に示す．jact は図 2 に示す三種類のアクタを想定しており，アクタによって利用の流れは異なる．

JSF 検討者：jact を利用することで自分が使用する JSF を検討するアクタである．JSF 検討者は図 2 において A1～A6 の矢印で示されるように，jact に示されたタスクから自分の利用したいタスクを選択し，そのタスクを実現するソースコードを比較することができる．また性能計測を実行することで，自分の環境下での各 JSF の性能を計測できる．

タスク投稿者：jact に用意されていないタスクを追加で投稿するアクタである．タスク投稿者は図 2 において B1，B2 の矢印で示されるように，必要なタスクの情報を jact に投稿する．

ソースコード投稿者：jact で用意されているタスクに対して，自作のソースコードを投稿するアクタである．ソースコード投稿者は図 2 において C1～C4 の矢印で示されるように，自作のソースコードを jact に投稿する．ソースコードに対してテストが行われ，成功すればソースコードとして登録される．

3.3 特　　徴

3.3.1 タスクベースのコード比較

課題 P_1：選択肢の豊富さに対応するための特徴であり，あるタスクを実現する各 JSF の様々なソースコードの比較を実現する．この特徴は図 2 に示された矢印 A3，A4 に対応する．タスクとして登録されている DOM 操作や Ajax 通信は，アプリケーションの開発時に頻繁に実装される機能である [4]．JSF 検討者はソースコードを比較することで，JSF の実装面を比較できる．

3.3.2 オンデマンドな性能計測

また jact は課題 P_2：実行環境の多様さに対応するために，各 JSF の性能面の比較を実現する性能計測機能を持つ．この特徴は図 2 に示された矢印 A5，A6 に対応する．選択したタスクに登録されているソースコードを実行して処理時間を計測し，グラフに表示する．想定される実行環境で本ツールを用いることで，JSF の性能面を比較できる．

3.3.3 プレイグラウンド

課題 P_3：情報入手の難しさに対応するために，jact をプレイグラウンド化している．タスク及びソースコードを共有することで，JSF がアップデートされた場合には新しい情報を共有でき，最新の情報を用いた JSF の比較が可能になる．

3.3.4 アクタによる投稿

jact のプレイグラウンド化を実現するための特徴であり，ソースコードの投稿は図 2 に示された矢印 C1～C4 に，タスクの投稿は矢印 B1，B2 に対応する．タスク及びソースコードを管理者が用意するだけでは，新しい情報への対応や十分な情報の提供は難しい．そこで，登録されていないが開発時に必要となるタスクや，同じ JSF を用いた別のソースコード，そして別の JSF を用いたソースコードの投稿を可能にした．

3.3.5 ソースコードのテスト

ソースコードの投稿機能を実現するためにソースコードのテスト機能が追加されており，この機能は図 2 に示された C2 に対応する．ソースコード投稿者が投稿したソースコードが，タスクを実現するための条件を満たしているかテストを行う．この機能により，タスクを実現するための条件を全て満たしたソースコードのみの登録を実現する．

4. 実　　装

4.1 概　　要

jact は JavaScript によって実装された Web アプリケーションである．ユーザインタフェースの実現のために，Vue.js を一部使用している．またエディタのシンタックスハイライトのために CodeMirror を使用している．jact は REST API を経由してサーバからデータベースに保存されたタスク，ソースコード及びテストの情報を取得している．API サーバには Node.js を，REST API の作成には Node.js のフレームワークである Express を使用している．タスク及びソースコードの情報を保持するデータベースとして MongoDB を使用している．開発期間は約 6 ヶ月，コード行数は約 1,200 行である．

4.2 コード比較

図 3 に jact のコード比較画面を示す．JSF 検討者は図 3 に

示される画面でソースコードの比較を行う．画面は左側のバーと右側のエリアに分割されている．左側のバーでは**タスク及びソースコードの選択，タスク投稿画面への遷移**が可能である．右側の 4 分割されたエリアでは，それぞれ**選択したソースコードの表示，自作ソースコードの作成・投稿，ソースコードのプレビュー，性能計測の実行開始**が可能である．

現在登録されているタスク及びソースコードを表 1 に挙げる．表 1 に挙げられる DOM 操作や Ajax 通信は，Web フロントエンド開発における基本的な操作である．タスクを基本的な操作に設定することで，開発者は自分が実現したい機能に近いタスクを比較できる．

4.3 タスクの投稿

タスクの投稿時に必要となる項目を表 2 に示す．各タスクにはタスク名等の基本的な情報だけでなく，ソースコードのテストを行うために以下の情報も登録されている．

pre：ページ上の要素のうち，変更のターゲットとなる DOM 要素の初期状態を表す．ソースコードの振る舞いを確認する際の事前条件に該当する．

trigger：ボタンのクリック等，ソースコードにおける JavaScript イベントの発火方法を指す．

post：イベント発火後のターゲットの期待値を指す．ソースコードの振る舞いを確認する際の事後条件に該当する．

タスク投稿者は表 2 に示されるすべての項目を送信することで，タスクを投稿できる．

4.4 性 能 計 測

性能計測時には，JSF 検討者が選択したタスクに対して各ソースコードがイベント発火から処理を完了するまでに要する時間を計測する．各ソースコードはそれぞれ 100 回実行され，100 回分の処理時間を合計して出力する．性能計測は，iframe と REST API を用いて各ソースコード毎に以下の手順で行われる．

表 1: 用意されているタスク

タスクカテゴリ	タスク名
DOM 操作	テキストの変更
	テキストの色の変更
	複数テキストの一括変更
	複数テキストの色の一括変更
	クラスの追加
	クラスの削除
Ajax 通信	JSON の取得

表 2: タスクの登録項目

項目名	概要
タスク名	登録するタスクの名称
投稿者名	登録するタスクの投稿者
ソースコード名	サンプルとして登録するソースコードの実装方法
ソースコード	サンプルとして登録するソースコード
pre	ソースコードの事前条件
trigger	イベントの発火方法
post	ソースコードの事後条件

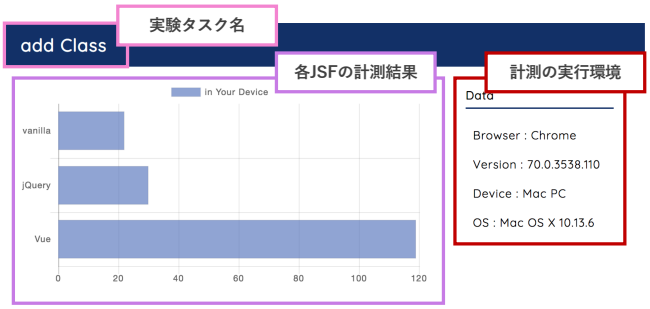

図 4: 性能計測結果の表示画面

(1) iframe を 100 個生成

同じ iframe を使用して実験を行う場合，キャッシュが適用されて正しく計測できない可能性がある．これを回避するために 100 回それぞれ別の iframe を利用して計測を行った．

(2) iframe のソースを API から取得

iframe の src 属性に URL を指定することで，当該フレーム内に Web ページを埋め込むことができる．jact では REST API を使用することでソースコードが取得できるように設計しているため，その URL を iframe の src 属性に指定している．

(3) 監視対象の DOM 要素を指定

イベントの発火による DOM 要素の変更を検知するために，MutationObserver(注2)を用いた．MutationObserver を利用するための準備として，変更を監視すべき DOM 要素を指定する．

(4) 計測を開始

計測開始時の時間を記録し，イベントを発火する．

(5) DOM 要素の変更を検知

イベント発火による DOM 要素の変更が検知された時間を記録する．計測開始時間との差分を取り，処理時間を記録する．

(6) 100 回の処理時間を合算

(3)〜(5) の処理を 100 回直列実行し，処理時間を合算する．

以上の手順をタスクに登録された全てのソースコードに対して直列実行する．全てのソースコードの計測が終了したら，グラフで表示する．計測結果の表示画面を図 4 に示す．計測結果の表示画面では**計測したタスク名，各ソースコードの処理時間のグラフ，実行環境**が表示される．

4.5 ソースコードのテスト

ソースコードの投稿機能を実現するためにソースコードのテスト機能が実装されている．ソースコードのテスト時には，各タスクに登録された **pre**，**trigger**，**post** を使用する．テストには性能計測時と同様に iframe を使用する．テストの手順は以下の通りである．

(1) iframe の更新

テストを実行するために，iframe のソースコードをテスト対象のソースコードに変更しておく．

(2) iframe に対して pre を実行

ソースコードが事前条件を満たしているか確認するために，

(注2)：JavaScript に用意されている API の 1 つで，監視対象とした DOM 要素が変更された時に指定したコールバック関数を実行することができる．

図 5: 実験タスクの実行例

```
1 <p id="target">Hello, World!</p>
2 <button type="button" id="changeButton">change</button>
3 <script>
4     var addClass = function () {
5         var targetDiv = document.getElementById('target');
6         targetDiv.classList.add('newClass');
7     }
8     document.getElementById('changeButton').addEventListener('click',
              addClass);
9 </script>
```

図 6: vanilla JS による クラス属性の追加 のソースコード

pre として登録されたソースコードを実行する．

(3) iframe に対して trigger を実行

ソースコード内のイベントを発火させるために，trigger として登録されたソースコードを iframe に対して実行する．

(4) iframe に対して post を実行

ソースコードが事後条件を満たしているか確認するために，post として登録されたソースコードを実行する．

(2) と (4) の実行に対して true が得られた場合，テストは成功となる．どちらか一方でも false となった場合や，イベントが正しく発火されなかった場合にはテストは失敗となる．

5. 予 備 実 験

5.1 概　　要

jact を用いて実際に JSF の比較を行う予備実験を実施する．本実験の目的は jact を用いることで JSF の実装面及び性能面の違いが確認できることを示す点にある．jact に登録されているタスクを選択してコード比較と性能計測を行う．性能計測は複数環境下で行い，それぞれの環境下での実行結果を比較する．

5.2 対　　象

対象となるタスクは，DOM 要素へのクラス属性の追加である．実行例を図 5 に示す．初期状態は文字列にクラスが追加されていないため，色の変更もなくテキストのみ表示されている．ボタンを押すことでテキストにクラス属性が追加され，テキストの背景に色が表示される．

比較対象の JSF としては，jQuery と Vue.js の 2 つを採用する．さらに性能計測のベースラインとして JSF を用いない（すなわち JavaScript のみを用いた）実装を比較対象に加える．以降ではこの JSF を用いないソースコードを vanilla JS と表す．

5.3 実 装 比 較

図 6，図 7，図 8 に実験タスク「クラス属性の追加」を実現する各 JSF のソースコードを示す．ただし JSF による違いが見られる，HTML の body タグ内の一部のみ抜粋している．

まず，図 7 に示した jQuery を用いた際のソースコードを vanilla JS と比較する．図 6 に示される vanilla JS の処理は，クラス属性を追加する関数の定義，及びイベントハンドラへの登録の 2 つで構成されている．jQuery でも記述の流れは同様

```
1 <p id="target">Hello, World!</p>
2 <button type="button" id="changeButton">change</button>
3 <script>
4     $('#changeButton').click(() => {
5         $('#target').addClass('newClass');
6     });
7 </script>
```

図 7: jQuery による クラス属性の追加 のソースコード

```
1  <div id="obj">
2      <p id="target" v-bind:class="{newClass:isActive}">Hello, World!</p>
3      <button type="button" id="changeButton" v-on:click="addClass()">
           change</button>
4  </div>
5  <script>
6      var vm = new Vue({
7          el: '#obj',
8          data: {
9              isActive:false
10         },
11         methods: {
12             addClass(){
13                 this.isActive = true;
14             }
15         }
16     })
17 </script>
```

図 8: Vue.js による クラス属性の追加 のソースコード

だが，$関数を使用することで HTML 中の DOM 要素の取得を簡潔に記述できている．また body タグ内の HTML 要素に関する記述は変更されておらず，script タグ内の JavaScript のソースコードのみ編集されている．jQuery の導入の際にはベースとなる HTML を変更する必要がないことがわかる．

Vue.js を用いてタスクを実装したソースコードを図 8 に示す．図 6 と比較すると全体としてコード行数は増加しているが，script タグ内の記述が簡潔であり，Vue インスタンスがデータと関数を持っている構造が把握しやすい．vanilla JS と jQuery が手続き的に DOM 操作を行っているのに対し，Vue.js では HTML 要素と JavaScript を宣言的に紐付けていることが確認できる．一方で図 8 の 1〜4 行目は図 6，7 の 1,2 行目から変更されている．Vue.js を自分のアプリケーションに導入した場合はソースコードを構造的に記述することができる一方，ベースとなる HTML の要素を変更して Vue.js の記述に合わせる必要がある．

5.4 性 能 比 較

3 つのデバイス（iPhone/Android/Windows PC）と 2 つのブラウザ（Chrome/Firefox）を組み合わせた合計 6 種類の環境で，JSF の処理時間の計測を行った．ネットワーク遅延による計測ノイズを回避するために，jact と計測デバイスは同一の LAN に配置した．図 9 に性能計測の結果を示す．縦軸は処理時間であり，各環境下での vanilla JS を基準とした時の処理時間の割合を表す．値が大きいほど処理時間が増加しており，処理速度の低下が発生していることを示している．横軸は使用 JSF と使用ブラウザを表しており，左側は Chrome における各 JSF の計測結果，右側は Firefox における各 JSF の計測結果を表す．各 JSF に対して並ぶ 3 つの縦棒は，左から順に iPhone，Android，Windows PC の結果を表している．

まず JSF による処理時間の違いについて考察する．JSF を含むソースコードは vanilla JS に比べて処理に時間がかかってい

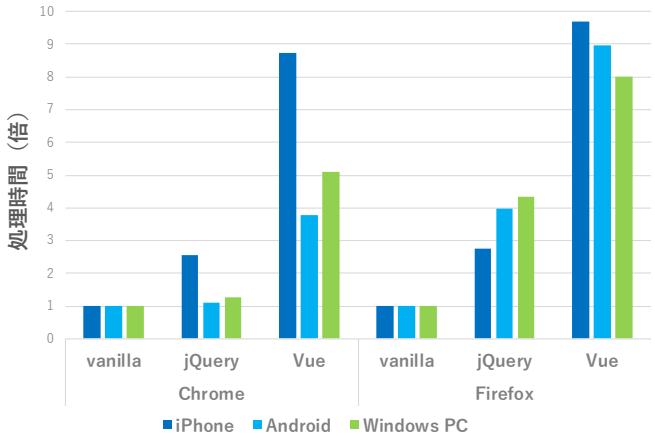

図 9: 実験結果：DOM 要素へのクラス追加

る．Vue.js のソースコードはその傾向が顕著であり，処理時間は最大で 9 倍になる．記述の簡潔化とのトレードオフとして，JSF は処理時間を増加させることがわかる．

次に実行環境による処理時間の違いについて考察する．iPhone は各ブラウザにおけるソースコードの処理時間比が殆ど変わらないのに対し，Android，Windows はブラウザ毎に処理時間比が異なる．jQuery のソースコードは Chrome で実行すると vanilla JS と同等の時間で処理できているのに対し，Firefox で実行すると 4 倍程度の処理時間が必要となる．また Vue.js のソースコードについても Chrome より Firefox の方が vanilla JS と比較して処理に時間がかかる．

5.5 考　　察

5.3 節の比較から，jQuery は HTML を変更せずに vanilla JS の冗長なコードを簡潔に記述することが出来ることがわかった．また Vue.js は HTML の変更が必要ではあるが，ソースコードを構造化できることがわかった．一方で 5.4 節の比較から，jQuery は一部の実行環境を除いて処理時間を増加させること，Vue.js は実行環境によらず処理時間を増加させ，その増加量は jQuery よりも大きいことがわかった．以上二点の比較から，JSF の特徴に関して次のようなことが考えられる．

vanilla JS：vanilla JS を使用することでソースコードは冗長になるが，必要な関数のみが実行されるため処理時間が短く性能がよい．性能を重視した機能を実現したい場合には JSF を使わずに JavaScript のみで実装するのが良い．

jQuery：jQuery を使用することで JavaScript のソースコードを HTML の変更を行わずに簡潔化できる．環境によっては多少性能が落ちるが，多少の性能低下が許容範囲内でかつ簡潔なソースコードを好むのであれば，jQuery で実装するのが良い．

Vue.js：Vue.js を使用することでソースコードを構造化することができる．HTML の変更は必要だが，ソースコードと表示する内容の対応を理解しやすくなる．一方どの環境下でも性能の低下が見られるため，アプリケーションの性能よりもソースコードの品質を重視したい場合や，今後の拡張のためにソースコードの可読性をあげたい場合は Vue.js で実装するのが良い．

もちろん一般に実装されるアプリケーションの機能が用意されているタスクのみで実装される可能性は極めて低く，タスク同士の組み合わせやタスクの拡張によって機能が実現される．

6. おわりに

本研究では JavaScript フレームワークの選択における課題を解決するために，フレームワークの実装面及び性能面を最新の情報で比較し，開発者のフレームワーク選択を支援するプレイグラウンド型ツール jact を試作した．また予備実験として jact を用いたフレームワークの実装面及び性能面の比較を行った．実験の結果，提案ツールを利用することでフレームワークを実装面及び性能面から比較できることを確かめられた．

今後の課題として，利用者が jact を用いることでフレームワークの比較をより容易にできるよう，次の二つの改善点が挙げられる．一つ目は比較情報の充実化である．現在 jact は一般公開されておらず，タスク及びソースコードの登録も全て管理者が行っている．そのため，jact の他者による利用や評価は行われていない．jact を一般公開し他者に利用，評価してもらうことで，比較情報の充実化やツールの改善を図ることができる．二つ目は別環境下での性能計測結果の提供である．現在 jact を利用している環境下での性能計測は実行できるが，他の環境下での性能計測結果や別の利用者の性能計測結果を表示して比較させることができない．別環境下での性能計測結果を収集し提供することで，より充実した性能計測を行うことができる．

謝辞　本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：16H02908，18H03222）の助成を得て行われた．

文　　献

[1] P. Ratanaworabhan, B. Livshits, and B.G. Zorn, “JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications,” Proceedings of the 2010 USENIX Conference on Web Application Development, pp.3–3, 2010.

[2] w3Techs, “Usage Statistics of Client-side Programming Languages for Websites, December 2018”. `https://w3techs.com/technologies/overview/client_side_language/all` (accessed at 2018/12/13)

[3] A. Pano, D. Graziotin, and P. Abrahamsson, “Factors and actors leading to the adoption of a JavaScript framework,” Journal on Empirical Software Engineering, vol.23, no.6, pp.3503–3534, 2018.

[4] A. Gizas, S. Christodoulou, and T. Papatheodorou, “Comparative Evaluation of JavaScript Frameworks,” Proceedings of the International Conference on World Wide Web, pp.513–514, 2012.

[5] C.L. Mariano, “Benchmarking JavaScript Frameworks,” Masters dissertation, Dublin Institute of Technology, 2017.

[6] Z. Ladan, “Comparing performance between plain JavaScript and popular JavaScript frameworks,” Bachelor’s thesis, Linnaeus University, Department of Computer Science, 2015.

[7] Angular, “Angular versioning and releases”. `https://angular.io/guide/releases` (accessed at 2018/12/13)

[8] 掌田津耶乃，JavaScript フレームワーク入門，秀和システム，2016.