Vol.2020-DPS-185 No.6
2020/12/21

Web WorkerにDOM操作機能を提供するフレームワークの設計と実装

宇佐見 優一郎[1,a)]　早川 智一[2,b)]

概要：本論文では，Web Worker（以下，Worker）に DOM（Document Object Model）操作機能を提供するフレームワークを提案する．Worker は，Web アプリケーションのバックグラウンド処理を実現するが，DOM 操作を行えないという仕様上の課題がある．この課題の解決を試みたフレームワークはあるが，DOM 操作機能の制限・独自の記法の強制・メモリのリークなどの制約を持つ．提案フレームワークは，これらの制約を緩和するために，独自のランタイムと利用者のコードの解析・変換機構を提供する．これによって利用者は，他のフレームワークよりも前述の制約を意識せずに，Worker 上のコード内に DOM 操作を記述できるようになる．我々はプロトタイプを設計・実装し，提案フレームワークが前述の制約を緩和することを確認した．

1. はじめに

Web Worker [1]（以下，Worker）は，Web アプリケーションの処理を効率化するための HTML 5 [2] の新技術である．従来の（Worker を用いない）Web アプリケーションでは，すべての処理が単一スレッド上で行われるため，実行に時間を要する処理が応答性を低下させる要因となっていた．Worker を用いると並列処理が可能になり，長時間継続する処理が応答性を維持したまま実現可能となった．

しかし，Worker には DOM（Document Object Model）操作を直接は行えないという課題がある．これは，Worker（を含む HTML 5 の仕様）が，マルチスレッドによる複雑さを避けるように設計されているためである．これによって，開発者は Worker 固有の仕様を意識した開発が必要となり，開発工数の増大等の問題を引き起こす．

この課題の解決を目的としたフレームワークはいくつか存在するが（3 章），それらのフレームワークを用いる場合，DOM 操作が可能になることと引き換えに以下の制約を受ける：（1）独自の記法の強制や利用可能な機能の制限（以下，実装制約）；（2）フレームワークの設計に起因するメモリリークの可能性（以下，メモリ制約）．これらの制約から，前述の課題が完全に解決されたとは言い難い．

[1] 明治大学大学院理工学研究科情報科学専攻
Graduate School of Science and Technology, Meiji University
[2] 明治大学理工学部情報科学科
School of Science and Technology, Meiji University
[a)] y_usami@cs.meiji.ac.jp
[b)] t_haya@cs.meiji.ac.jp

そこで本論文では，これらの制約を緩和するフレームワークを提案する．提案フレームワークは，（1）実装制約を緩和するためのランタイムと，（2）メモリ制約を緩和するためのソースコードの解析・変換機構と――を提供する．これによって開発者は，他のフレームワークよりも少ない制約で Worker からの DOM 操作が可能となる．

本論文の構成は次のとおりである．2 章では，本研究の技術的背景について概説する．3 章では，先行研究・関連技術を紹介する．4 章では，提案フレームワークを概説する．5 章と 6 章では，提案フレームワークの設計と実装を詳説する．7 章では，提案フレームワークの評価結果を報告する．8 章では，まとめと今後の予定を述べる．

2. 技術的背景

本章では，本研究の技術的背景として，Web アプリケーションの実行モデルと Worker との関連について概説する．

2.1 従来の Web アプリケーションの実行モデル

図 **1** に，従来の Web アプリケーションの実行モデルを示す．従来の Web アプリケーションの場合，すべての処理が，Main スレッドや UI スレッドと呼ばれる単一のスレッド（以下，Main スレッド）上で行われる．Main スレッドでは，すべての処理（画面の更新・ネットワーク通信・タイマー処理など）をイベントとして扱い，イベントを 1 つの実行単位として処理が進む．イベントは次の流れで処理される：（1）メッセージキューからイベントを取り出す；（2）イベントに紐付いた処理を実行する；（3）実行完了後，

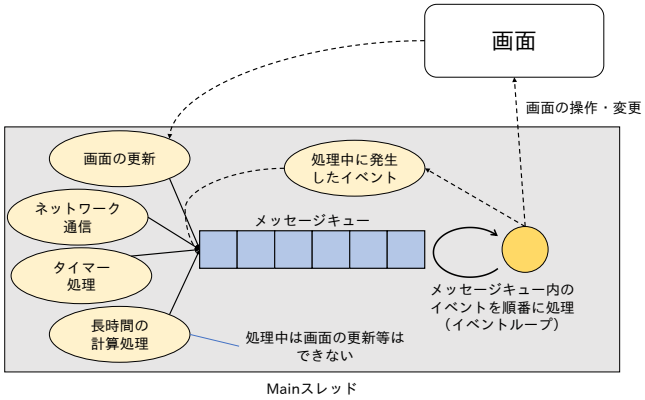

図 1　従来の Web アプリケーションの実行モデル

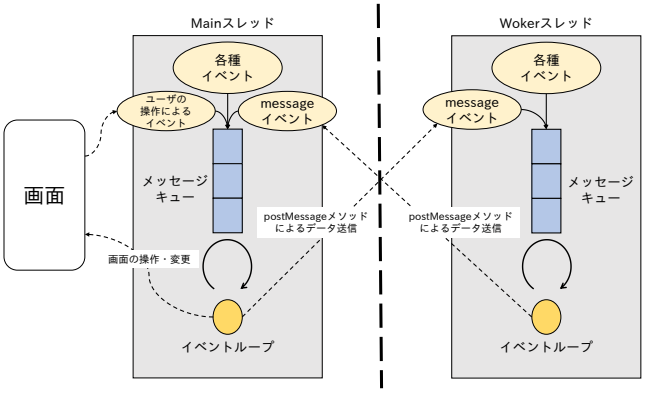

図 2　Worker を用いた Web アプリケーションの実行モデル

メッセージキューに未処理のイベントが存在するか確認する；(4-a) 存在すればそのイベントを処理する；(4-b) 存在しなければ新しいイベントが発生するまでイベントを待ち続ける．この一連の繰り返しをイベントループと呼ぶ．

Web アプリケーションにおいて，長時間実行される処理は応答性を低下させる要因となる．これは，長時間実行される処理によって他のイベント処理が滞ってしまうためである．これによって Web アプリケーションの利用者からは，Web ブラウザがフリーズしたように見えてしまう．

2.2　Worker を用いた Web アプリケーションの実行モデル

図 2 に，Worker を用いた Web アプリケーションの実行モデルを示す．Worker を用いることで，独立したスレッドでの並列処理が可能となる．各々のスレッドは，独立したメモリ空間とイベントループを持つため，スレッド間でのデータのやり取りはメッセージパッシングを用いる．具体的には，一方のスレッドから `postMessage` メソッドを用いてデータを送信し，他方のスレッドはそれを message イベントとして受け取ることでデータの授受を実現する．

Worker の特徴に，(1) Main スレッド上で動作するソースコードと Worker スレッド上で動作するソースコードを（一般的には別ファイルとして）分割して用意する必要があること，(2) Worker スレッド上で動作するソースコード内で DOM 操作ができないこと——がある．これによって開発者は，従来とは本質的に異なる開発が必要となる．

以降，本論文中では，2 つのスレッドを区別する際に，Main 側・Worker 側と呼ぶことで区別する．

3.　先行研究・関連技術

本章では，提案フレームワークに類似する（Worker 内での DOM 操作の実現を主眼とする）フレームワークを紹介し，提案フレームワークとの類似性・差異について述べる．

3.1　先行研究

我々はこれまで，フレームワーク独自の記法の強制を緩和するため，それを隠蔽するフレームワークを提案してきた [3]．同フレームワークでは，ソースコードを自動変換する機構を開発者に提供することで，独自の記法の隠蔽を行う．これによって開発者は，Worker 上で動作するソースコードに対しても，ブラウザの提供する DOM API（Application Programming Interface）と同様の形式で DOM 操作を記述可能となる恩恵が得られる．

しかし，このフレームワークはフレームワーク独自の記法の隠蔽を目的としており，メモリリークの可能性（メモリ制約）には対処していなかった．本提案フレームワークでは，メモリリークの可能性までを考慮した点で異なる．

3.2　関連技術

Via.js [4] は，DOM 等の API を Worker 内で模倣するフレームワークである．Via.js と提案フレームワークは，Proxy オブジェクト [5] を利用して Worker に DOM 操作機能を提供する点で共通性がある．しかし Via.js では，DOM 要素のプロパティ値取得に独自の記法が必要となり（実装制約），リソースの同期機構を原因としたメモリリーク（3.4.1 節）の可能性がある（メモリ制約）．提案フレームワークでは，独自の機構（5.4 節）でこれらの制約を緩和する点で異なる．

DOM-Proxy [6] は，Via.js を発展させ，より透過的な DOM API 記述を可能にする実験的なフレームワークである．DOM-Proxy では，SharedArrayBuffer [7] や WeakRefs [8] によるメモリ管理，Atomics API [9] によるスレッド間の同期制御によって，Worker 内での透過的な DOM API の利用を実現している．しかし，DOM-Proxy は実験的実装のため，一部の最新ブラウザでしか動作せず，現時点で広く利用されているブラウザ環境での動作は想定されていないようである．提案フレームワークは，広く利用されているブラウザ環境で利用可能な機能のみで制約を緩和する点で異なる．

WorkerDOM [10] は，DOM API の独自の実装を Worker に提供するフレームワークである．WorkerDOM と提案フレームワークは，Worker 内に DOM 操作を記述できる点で共通性がある．しかし WorkerDOM では，DOM API を独自実装しているため利用可能な機能が限定的であり（実

表 1　3 つの要件で見た関連技術の比較

	再現性	堅牢性	汎用性
Via.js	△	△	○
WorkerDOM	×	×	○
DOM-Proxy	○	○	×

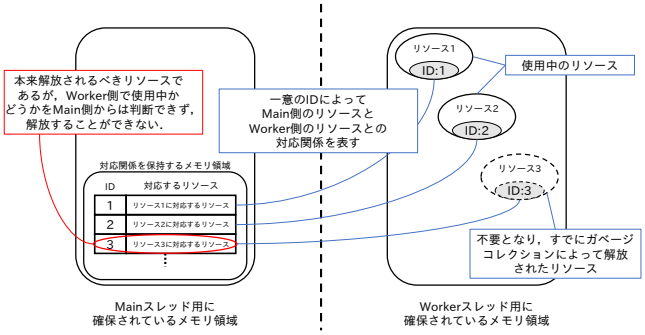

図 3　メモリリークを引き起こすリソース同期の模式図

装制約），リソースの同期機構を原因としたメモリリーク（3.4.1 節）が発生する可能性がある（メモリ制約）．提案フレームワークでは，Proxy オブジェクトを用いた設計と変換機構の導入によってこれらの制約を緩和する点で異なる．

3.3　関連技術の比較

1 章に挙げた課題点を解消するために，フレームワークに求められる要件を次のように定義した：

(1) 再現性：ブラウザの提供する DOM API と同様の形式で記述できること；
(2) 堅牢性：安全に（フレームワークを原因としたメモリリークを起こすことなく）利用可能であること；
(3) 汎用性：広い環境で利用可能な機能のみを用いて実装されていること．

これらの要件について，上述の関連技術をまとめたものが表 **1** である．どのフレームワークにも課題があり，すべての要件を満たすフレームワークはないことがわかる．

3.4　関連技術での制限とその原因

3.4.1　リソースの同期によるメモリリーク

関連技術では，スレッド間でのリソースの同期を原因としたメモリリークが発生する可能性がある．これは，(1) 各スレッドが独立したメモリ空間をもつという Worker の仕様，(2) 不要なメモリが解放（ガベージコレクション）されたか否かを言語的に取得する方法がないという JavaScript の仕様 ——によるものである．

メモリリークを引き起こすリソース同期の仕組みを模式化したものが図 **3** である．(1) の特徴により，フレームワーク内部では，Worker 側から利用されたリソースと，それに対応する実際のリソースとの対応関係を Main 側で保持する必要がある．Worker 側で利用される各リソースに対して，一意となるように ID をもたせることでこれを実現できる．これによって，メッセージパッシングを利用して，Worker 側から Main 側のリソースを指定し利用することができるようになる．Worker 側で利用されたリソースが不要となると，JavaScript のランタイム（ガベージコレクタ）によって自動的にメモリが解放される．しかし (2) の特徴から，不要となったリソースが解放されたことをフレームワークが知ることができず，Worker 側で解放されたリソースに対応する Main 側のリソースが保持し続けられてしまう．これは，Main 側の対応関係から明示的に削除しなければ，ガベージコレクションの対象にならないためである．

以上より，不要なリソースが解放されずに蓄積し続けてしまう可能性がある．Via.js・DOM-Proxy では不要なリソースとして DOM 要素が蓄積し，WorkerDOM では DOM 要素とそれに付随する文字列が蓄積し続ける．

このメモリリークの問題に対し，Via.js や DOM-Proxy では WeakRefs を用いた改善が提案されている．WeakRefs は，JavaScript での弱参照[*1]を実現する新機能であり，`FinalizationRegistry` クラスを利用することでリソース解放時の処理を定義することが可能となる．しかし本論文執筆時点で，WeakRefs は提案段階（Stage 4 Proposal）の機能であり，ECMAScript の次期バージョン（ECMAScript 2021）までは正式な仕様とはいえない．したがって，この機能が利用可能な Web ブラウザは限定的である．

3.4.2　同期呼び出しが必要な API の利用制限

Via.js と WorkerDOM では，`preventDefault` 等の同期的な呼び出しが必要な API は Worker 内で利用できない．これは，Main 側でのイベントの発火に対し，それを処理するための関数が Worker 側で定義されているためである．これらの API を模倣するには，JavaScript の処理を一時的に停止・再開させる（言語レベルの）機能が必要となる．

この問題に対し，DOM-Proxy では実験的実装として Atomics API を用いた方法が試みられているが，スレッド停止によるパフォーマンスの低下が示されている．これは，Atomics API のうち，提案段階（Stage 3 Proposal）の機能である `Atomics.waitAsync` [11] が必要であり，現状では `Atomics.waitAsync` の動作を模倣した実装（ポリフィル）を利用せざるをえないためである．

提案フレームワークにおいても，これらの API の利用はできず，広い環境で動作する機能のみを用いてこれらの API の利用を実現することは今後の課題である．

4.　提案手法

4.1　概要

図 **4** に，提案フレームワークの概要を示す．提案フレーム

[*1] 弱参照は，ガベージコレクタによるオブジェクトの回収を妨げない参照である．オブジェクトに対して，通常の参照が 1 つでも残っている場合は回収の対象とならないが，弱参照のみが残っている場合は，回収の対象となる．

ワークは，Main 側フレームワーク・Worker 側フレームワーク・変換機構で構成される．処理の流れは次のとおりである：(1) 開発者は Main 側ソースコードとともに，Worker 側ソースコードを生の（ブラウザの提供する）DOM API と同じ形式で記述する；(2) 変換機構で Worker 側ソースコードを提案フレームワーク固有の形式へ変形する；(3) Main 側ソースコードを Web ブラウザが読み込み，Main 側ソースコード内で Worker が起動する；(4) Worker 側ソースコード内で DOM API の呼び出しが発生する；(5) Worker 側フレームワークが DOM API 呼び出しの内容を Main 側フレームワークへ送信する；(6) Main 側フレームワークが，Worker 側フレームワークから送られてきた DOM API 呼び出しを実行する；(7–8) DOM API 呼び出しに戻り値が存在する場合は，Worker 側フレームワークへ通知する．

4.2 利用例

図 **5** と図 **6** に，提案フレームワークの利用例を示す．この例は，文献 [1] 内の素数を探索して画面に表示するプログラムを提案フレームワークで書き換えたものである．

Main 側ソースコード（図 5）では，Web ブラウザが提供する `Worker` クラスの代わりに，フレームワークが提供するクラス（`ViceWorker`）をインスタンス化する．その後 `start` メソッドを呼び出して Worker スレッドを起動する．

Worker 側ソースコード（図 6）では，先頭に提案フレームワークを読み込む 1 行（`importScripts`）を記述する．これによって，この行以降で DOM API が利用可能となる．

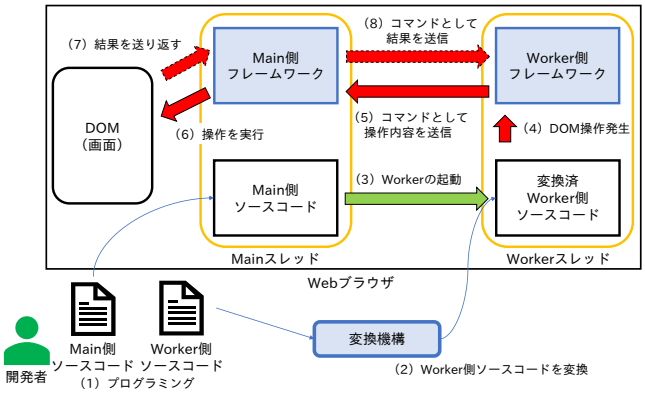

図 **4** 提案フレームワークの概要

```
const worker = new ViceWorker("worker.js");
worker.start(); /* 提案フレームワークを初期化して起動する */
```

図 **5** 提案フレームワークの利用例（Main 側）

```
/* 提案フレームワークの読み込み */
importScripts("./worker_bundle.js");

let n = 1, isPrime;
while (true) {
    n++, isPrime = true;
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime) {
        /* 提案フレームワークが提供するオブジェクト（独自のdocument）経由でのDOM API呼び出し */
        document.getElementById("result").textContent = n;
    }
}
```

図 **6** 提案フレームワークの利用例（Worker 側）

```
/* 右辺の関数を送信できない */
button.onclick = function() { console.log("clicked!"); };

/* 第二引数の関数を送信できない */
button.addEventListener("click", (e) => {
  console.log("clicked!");
});
```

図 **7** `postMessage` メソッドを使用して送信できないデータの例

5. 設計

提案フレームワークは，アプリケーション実行時のランタイム（Main 側フレームワーク・Worker 側フレームワーク）と，アプリケーション実行前に必要な変換機構からなる．

5.1 DOM 操作の実現

提案フレームワークを用いた Worker からの DOM 操作は，DOM 操作の内容を抽象化したコマンドをスレッド間で送受信することで実現する．具体的には，(1) Worker 側ソースコードで発生した DOM 操作を Worker 側フレームワークが受け取り，(2) 操作内容に対応する 1 つのコマンドを生成し，(3) 生成したコマンドを Main 側フレームワークへ送信し，(4) Main 側フレームワークが受け取ったコマンドを処理する——ことで DOM 操作を実現する．これによって開発者は，Worker 特有のメッセージパッシングを用いた DOM 操作を意識せずに DOM 操作が可能となる．

コマンドを用いた DOM 操作を実現するには，ユーザからの DOM 操作を検知し，それに対応するコマンドを生成する方法が必要となる．これを実現するために，提案フレームワークでは Proxy オブジェクトを利用する．Proxy オブジェクトは ECMAScript 2015 で導入された，オブジェクトに対する基本的な操作（値の取得，代入，関数呼び出しなど）に割り込んで独自の操作を定義できる機能である．

一方のスレッドのフレームワークから他方のスレッドのフレームワークへコマンドを送信する際に，Worker が提供する `postMessage` メソッドを用いるため，送信可能なデータは制限される．これは，`postMessage` メソッドに渡されるデータは structured clone algorithm [12] でコピーされるからであり，コピー不可能なデータは送信することができないからである．送信不可能なデータの一例として関数オブジェクトが挙げられる． 図 **7** に示すような DOM 操作において，関数をデータとして渡すことができない．

`postMessage` メソッドで送信不可能なデータを送信する必要がある時，提案フレームワークでは Proxy オブジェクトを用いた変換（以下，データ変換）を行う．具体的には，(1) 送信側で，送信不可能なデータを判別可能にする一意の ID を生成し，その ID と送信不可能なデータの対応関係を保存し，(2) 送信不可能なデータを送信する代わり

に，そのデータに対応した ID を送信し，(3) 受信側で，受け取った ID を用いて Proxy オブジェクトを生成し，送信不可能なデータの代わりとして扱う——という操作を行う．

5.2 Main 側フレームワーク

Main 側フレームワークは，実行時に Main 側ソースコードによって読み込まれる．

5.2.1 役割

Main 側フレームワークの役割は次のとおりである：(1) Worker 側フレームワークから送られてくるコマンドを処理する；(2) Worker 側で利用された DOM 要素を保持する；(3) Worker 側フレームワークへ通知が必要なイベントが発生した場合に，Worker 側フレームワークへ通知する．

5.2.2 送信するコマンド

表 **2** に，Main 側フレームワークから Worker 側フレームワークへ送信されるコマンドを示す．Start コマンドは，Worker スレッドでの処理を開始するためのコマンドである．Callback コマンドは，Worker 側からの操作によってリスナが登録された場合に，そのリスナが呼び出されたことを通知するコマンドである．Done コマンドは，Worker 側からの Get コマンドの結果を通知するコマンドである．

各コマンドは Worker 側での処理時に必要なデータを含んでいる．structured clone algorithm によって Worker 側への送信が不可能なものは，前述のデータ変換が行われる．

5.3 Worker 側フレームワーク

Worker 側フレームワークは，実行時に Worker 側ソースコードによって読み込まれる．

5.3.1 役割

Worker 側フレームワークの役割は次のとおりである：(1) Worker 側ソースコードに Proxy オブジェクトで表されたグローバル変数 `document` を提供する；(2) DOM API 呼び出しが発生するたびに，Main 側フレームワークへコマンドを送信する；(3) 不要となった Proxy オブジェクトを Main 側フレームワークへコマンドで通知する；(4) Main 側フレームワークから送られてくるコマンドを処理する．

また，Worker 側フレームワークは Worker 側ソースコードに対して `get` メソッドと `clearReferences` メソッドを提供する．これらのメソッドは，開発者が直接使用することはなく，適切な場所でこれらのメソッドが呼び出されるように変換機構が Worker 側ソースコードを変換する．`get` メソッドは，Proxy オブジェクトで表された DOM 要素のプロパティ値を取得する際に呼び出される非同期メソッドであり，後述する Get コマンドを生成する．`clearReferences` メソッドは，不要となった Proxy オブジェクトを Main 側フレームワークへ通知する際に呼び出され，後述する Clear コマンドを生成する．

5.3.2 送信するコマンド

表 **3** に，Worker 側フレームワークから Main 側フレームワークへ送信されるコマンドを示す．Get コマンドは，Proxy オブジェクトによって表されたプロパティの実際の値を取得するためのコマンドである．Set コマンドは，Proxy オブジェクトによって表されたプロパティへ値を代入するためのコマンドである．Call コマンドは，メソッドの呼び出し時に生成されるコマンドである．Construct コマンドは，new 演算子によってインスタンスを作成するときに生成されるコマンドである．Clear コマンドは，Worker 側で不必要となった Proxy オブジェクトを Main 側に伝えるコマンドである．

Main 側から送信されるコマンドと同様に，各コマンドは Main 側で処理される際に必要なデータを含んでいる．structured clone algorithm によって Main 側への送信が不可能なものに関しては，前述のデータ変換が行われる．

5.4 変換機構

5.4.1 役割

変換機構の役割は，Worker 側で実行されるソースコードをフレームワーク固有の形式へ変換することである．

ソースコードの変換の流れは次のとおりである：(1) 開発者が作成した Worker 側ソースコードから抽象構文木（以下，AST : Abstract Syntax Tree）を生成する；(2) AST を解析し，後述の変形を施す；(3) 変形した AST から Worker 側ソースコードを生成する．これにより，開発者は Worker 側ソースコードがどう変換されたかを意識する必要はない．

5.4.2 Proxy の判定

変換機構では，プログラム内の各変数に対して，実行時に Proxy オブジェクトとなるかどうかの判定を行う．ある変数が Proxy と判定されるのは，(1) `const` で宣言された変数で[*2]，(2) Proxy オブジェクトに対するメソッド呼び出しの戻り値——の場合である．フレームワークが提供するグローバル変数 `document` が Proxy オブジェクトであり，同 Proxy オブジェクトに対するメソッド呼び出しの戻り値も，提案フレームワークが提供する Proxy オブジェクトとなることから判定が行われる．

図 **8** に，変数が Proxy オブジェクトかどうかの判定例を示す．この例のコード片は，新たな p 要素を DOM API を用いて生成して，変数 `p` に代入する処理を行うものである．このコードに対応する AST から，代入の右辺が Proxy オブジェクト（この例では `document`）に対するメソッド呼び出し（この例では `createElement` の呼び出し）であることがわかる．これによって，新たに宣言された変数 `p` は実行時に Proxy オブジェクトとなると判定される．

[*2] JavaScript には，変数を宣言する方法が 3 種類（`var`, `let`, `const`）存在する．`var` と `let` では再代入が可能だが，`const` では再代入ができない．`var` と `let` への対応は今後の課題である．

表 2　Main 側フレームワークから Worker 側フレームワークへ送信されるコマンドの一覧

コマンド	説明	同時に送信されるデータ
Start	Worker スレッド内処理の開始	なし
Callback	イベント発生の通知	funcId，args
Done	値の取得の操作の結果通知	flushId，results

表 3　Worker 側フレームワークから Main 側フレームワークへ送信されるコマンドの一覧

コマンド	説明	同時に送信されるデータ
Get	プロパティの値の取得操作	getId，objectId，path
Set	プロパティへの値の代入操作	objectId，path，value
Call	メソッドの呼び出し	objectId，path，args，returnObjectId
Construct	インスタンスの生成	objectId，path，args，returnObjectId
Clear	不要リソースの解放	objectIds

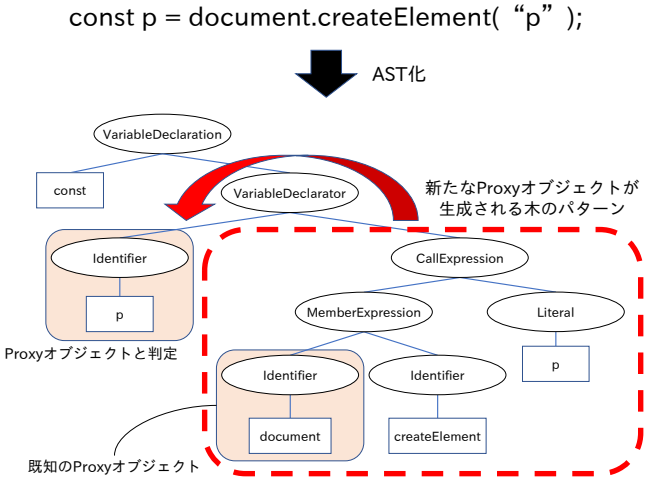

図 8　変数が Proxy オブジェクトかどうかの判定の例

5.4.3　変形処理

変換機構では，AST に 4 つの変形を行う．例として，ソースコードの変形前（図 **9**）と過程（図 **10** から図 **13**）を示す．

1 つ目は，Worker 側ソースコード全体を 1 つの関数とする変形である．変形適用後のソースコードを図 10 に示す．Worker 側フレームワークでは，Main 側フレームワークから Start コマンドを受け取ると，この関数を実行してアプリケーションの処理を開始する．なお，ここでの関数名 `main` は便宜的なものであり，実際にはソースコード内の他の変数名と異なる（衝突しない）名前を割り当てる．

2 つ目は，Proxy オブジェクトに対するメソッド呼び出しの戻り値を利用していない場合に，その戻り値を一時変数へ代入する変形である．変形適用後のソースコードを図 **11** に示す．一時変数に用いる変数名は，ソースコード内の他の変数名と異なるものをフレームワークが生成して割り当てる．図 11 では便宜的に `__tmp__1` としている．この変形により，不要となる Proxy オブジェクトを適切に Main 側フレームワークへ通知できるようになる．

3 つ目は，DOM 要素のプロパティ値取得にあたる操作を探索し，`get` メソッドを呼び出す形への変形である．変形適用後のソースコードを図 **12** に示す．これは，プロパティ値そのものを Proxy オブジェクトで表すことができず，Main 側へ実際の値を取得する必要があるためである．

4 つ目は，変数スコープごとに Proxy オブジェクトを探索し，不要となることが明確な Proxy オブジェクトに対して `clearReferences` メソッドを呼び出す形への変形で

```
const p = document.createElement("p");
p.textContent = document.title;
document.body.appendChild(p);
```

図 9　変形適用前のソースコード

```
async function main() {
    const p = document.createElement("p");
    p.textContent = document.title;
    document.body.appendChild(p);
}
```

図 10　1 つ目の変形適用後のソースコード

```
async function main() {
    const p = document.createElement("p");
    p.textContent = document.title;
    const __tmp__1 = document.body.appendChild(p);
}
```

図 11　2 つ目の変形適用後のソースコード

```
async function main() {
    const p = document.createElement("p");
    p.textContent = await get(document.title);
    const __tmp__1 = document.body.appendChild(p);
}
```

図 12　3 つ目の変形適用後のソースコード

```
async function main() {
    (() => {
        return new Promise(async (resolve) => {
            const p = document.createElement("p");
            p.textContent = await get(document.title);
            const __tmp__1 = document.body.appendChild(p);
            resolve([p, __tmp__1]);
        });
    })().then((proxies) => {
        clearReferences(proxies);
    });
}
```

図 13　4 つ目の変形適用後のソースコード

表 4　変換機構の実装に用いたソフトウェア

ソフトウェア	バージョン	備考
Node.js	14.15.0	JavaScript ランタイム
Esprima	4.0.1	AST の生成
Estraverse	4.2.0	AST の変形
Escope	3.6.0	スコープ木の生成
Escodegen	1.11.0	ソースコードの生成

```
setInterval(() => {
    const div = document.createElement("div");
    document.body.appendChild(div);
    document.body.removeChild(div);
}, 1);
```

図 14　評価対象のコード片

ある．変形適用後のソースコードを図 13 に示す．これによって，Main 側が Worker 側の変数の生存期間を知れないが故のメモリリークを緩和する．

6. 実装

表 4 に，変換機構のプロトタイプ実装に用いたソフトウェアを示す．変換機構は AST を扱うことができれば実装言語を問わないが，今回は Node.js 上で利用可能な CLI（Command Line Interface）ツールとして実装した．これは，Web アプリケーション開発では JavaScript が主流の開発言語であり，言語の統一性から Node.js が最適であると考えたためである．ソースコードから AST への変換に Esprima [13] を用いた．AST の変形に Estraverse [14], Escope [15] を用いた．変形した AST からソースコードへの変換に Escodegen [16] を用いた．

7. 評価

本章では，提案フレームワークの評価結果を報告する．我々は，フレームワークごとのメモリ制約を調べるために，実行時のヒープ使用量と DOM ノード数を測定した．

7.1 評価対象

測定対象のコード片を図 14 に示す．DOM の変更が頻繁に発生する状況を再現するために，測定対象のアプリケーションを div 要素の生成と削除を無限に繰り返すものとした．これに相当する機能を Worker を用いない場合[*3]・Via.js・WorkerDOM・DOM-Proxy・提案フレームワークのそれぞれで実装し，ヒープ使用量と DOM ノード数の変化を測定した．Via.js に関しては，WeakRefs が利用可能か否かで結果が異なるため，それぞれについて測定した．各フレームワークは本論文執筆時点での最新版を用いた．

[*3] 今回測定に用いたコード片は DOM 操作のみを含むため，生の（フレームワークを用いない）Worker 上での実行ができない．そのため，生の Worker との比較ではなく，Main スレッドのみで実行する場合との比較を行なった．より現実性の高いアプリケーションを対象とした評価は今後の課題である．

表 5　評価に用いたソフトウェア

ソフトウェア	バージョン	備考
Puppeteer	5.3.1	ブラウザテスト用ライブラリ
Google Chrome	86.0.4240.198	Web ブラウザ

7.2 評価手順

測定の手順は次のとおりである：(1) アプリケーションの読み込みが完了した時点でのヒープ使用量と DOM ノード数を測定する；(2) GC（Garbage Collection）を実行し（以下，開始時 GC），ヒープ使用量と DOM ノード数を測定する；(3) アプリケーションの実行を開始する；(4) 1 秒ごとにその時点でのヒープ使用量と DOM ノード数を測定する；(5) 60 秒経過後，GC を実行し（以下，終了時 GC），ヒープ使用量と DOM ノード数を測定する．

7.3 環境

評価を行った環境は以下のとおりである．

- CPU：Intel Core i7-8550U CPU @ 1.80GHz
- ホスト OS：Windows 10 Pro 1909
- 仮想環境：Oracle VirtualBox 6.0
- ゲスト OS：Ubuntu 20.04.1 LTS
  - コア数・スレッド数：2 コア 2 スレッド

評価に用いたソフトウェアを表 5 に示す．アプリケーションを動作させるブラウザとして Google Chrome を用いた．これは，開始時 GC と終了時 GC を実行するために，プログラム内から任意のタイミングで GC を実行する必要があるためである[*4]．メモリ使用量と DOM ノード数を測定するために Puppeteer [17] を利用した．Puppeteer が提供する `page.metrics` メソッドを用いて，実行時のヒープ使用量と DOM ノード数を取得することが可能である．

7.4 結果

測定結果をグラフ化したものを図 15, 図 16 に示す．図 15 のグラフにおいて，縦軸はヒープ使用量（MiB），横軸は経過時間（秒）である．図 16 のグラフにおいて，縦軸は DOM ノード数，横軸は経過時間（秒）である．アプリケーションの読み込みが完了した時点での測定結果を 0 秒の値，開始時 GC 実行後の測定結果を 1 秒の値，アプリケーション実行中の測定結果を 2–61 秒の値，終了時 GC 実行後の測定結果を 62 秒の値としている．

測定の際，DOM-Proxy を用いて実装したアプリケーションでは実行時エラーが発生し，正常な測定を行うことができなかった．これは，DOM-Proxy が古いブラウザ API 仕様に対して実装されていることが原因であった．実行時エラーの解消には，フレームワーク内部を書き換える必要があるため，今回は図 15 と図 16 の測定結果から除外した．

[*4] 起動時にオプション `--js-flags=--expose-gc` を指定することでグローバル関数 `gc` が提供され，動作中の Web アプリケーションが任意のタイミングで GC を実行できるようになる．

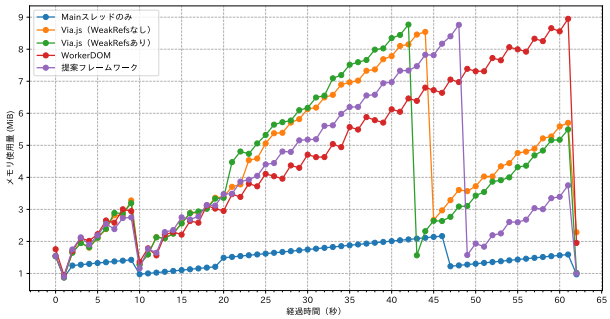

図 **15**　フレームワークごとのヒープ使用量の経時変化

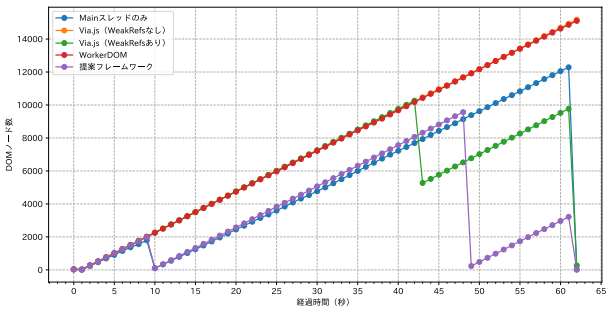

図 **16**　フレームワークごとの DOM ノード数の経時変化

図 15 から，以下のことが分かる：終了時 GC 実行後の値から，(1) Via.js（WeakRefs なし）と WorkerDOM は Main スレッドのみと比較して，1MiB 多くヒープを使用している；(2) Via.js（WeakRefs あり）のヒープ使用量は，Via.js（WeakRefs なし）・WorkerDOM よりも少なく，Main スレッドのみと同程度である；(3) 提案フレームワークのヒープ使用量は，Via.js（WeakRefs なし）・WorkerDOM よりも少なく，Main スレッドのみ・Via.js（WeakRefs あり）と同程度である．これらより，提案フレームワークが他のフレームワーク以上にヒープメモリを解放していることがわかる．

図 16 から，以下のことが分かる：終了時 GC 実行後の値から，(1) Via.js（WeakRefs なし）と WorkerDOM は，不要 DOM ノードを解放していない；(2) Via.js（WeakRefs あり）は，Main スレッドのみと同程度の不要 DOM ノードを解放している；(3) 提案フレームワークは，Via.js（WeakRefs なし）・WorkerDOM よりも多く，Main スレッドのみ・Via.js（WeakRefs あり）と同程度の不要 DOM ノードを解放している．これらより，提案フレームワークが他のフレームワーク以上に DOM ノードを解放していることがわかる．

これらの測定結果から，他のフレームワークと比較して提案フレームワークが，提案段階の機能を用いずに不要なメモリをより多く解放し，メモリリークの可能性を低減していることがわかる．以上より，我々は提案フレームワークが 1 章で述べた制約を緩和しうるという結論を得た．

8. おわりに

本論文では，Worker に DOM 操作機能を提供するフレームワークを提案した．今後は，変換機構の改善と，提案フレームワークのより詳細な評価が課題である．

参考文献

[1] WHATWG: Web Workers, WHATWG (online), available from ⟨https://html.spec.whatwg.org/multipage/workers.html⟩ (accessed 2020-11-03).
[2] WHATWG: HTML 5, WHATWG (online), available from ⟨https://html.spec.whatwg.org/multipage/⟩ (accessed 2020-11-03).
[3] 宇佐見優一郎，早川智一：Web Worker に DOM 操作機能を提供するフレームワークの提案，第 81 回全国大会講演論文集，Vol. 2019, No. 1, pp. 227–228 (2019).
[4] Scirra, A.: Via.js, Ashley Scirra (online), available from ⟨https://github.com/AshleyScirra/via.js⟩ (accessed 2020-11-03).
[5] Ecma International: Proxy Objects, Ecma International (online), available from ⟨https://www.ecma-international.org/ecma-262/11.0/index.html#sec-proxy-objects⟩ (accessed 2020-11-23).
[6] mmis1000: DOM-Proxy, mmis1000 (online), available from ⟨https://github.com/mmis1000/DOM-Proxy⟩ (accessed 2020-11-03).
[7] Ecma International: SharedArrayBuffer Objects, Ecma International (online), available from ⟨https://www.ecma-international.org/ecma-262/11.0/index.html#sec-sharedarraybuffer-objects⟩ (accessed 2020-11-23).
[8] Ecma TC39: WeakRefs TC39 proposal, Ecma TC39 (online), available from ⟨https://github.com/tc39/proposal-weakrefs⟩ (accessed 2020-11-03).
[9] Ecma International: The Atomics Object, Ecma International (online), available from ⟨https://www.ecma-international.org/ecma-262/11.0/index.html#sec-atomics-object⟩ (accessed 2020-11-23).
[10] AMP Project: WorkerDOM, Google (online), available from ⟨https://github.com/ampproject/worker-dom⟩ (accessed 2020-11-03).
[11] ECMA TC39: Asynchronous atomic wait for ECMAScript, Ecma TC39 (online), available from ⟨https://github.com/tc39/proposal-atomics-wait-async⟩ (accessed 2020-11-23).
[12] Mozilla Contributors: The structured clone algorithm, Mozilla (online), available from ⟨https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm⟩ (accessed 2020-11-03).
[13] Hidayat, A.: Esprima, OpenJS Foundation (online), available from ⟨http://esprima.org/⟩ (accessed 2020-11-03).
[14] Suzuki, Y.: Estraverse, ECMAScript Tooling (online), available from ⟨https://github.com/estools/estraverse⟩ (accessed 2020-11-03).
[15] Suzuki, Y.: Escope, ECMAScript Tooling (online), available from ⟨https://github.com/estools/escope⟩ (accessed 2020-11-03).
[16] Suzuki, Y.: Escodegen, ECMAScript Tooling (online), available from ⟨https://github.com/estools/escodegen⟩ (accessed 2020-11-03).
[17] The Chromium Authors: Puppeteer, Google (online), available from ⟨https://pptr.dev/⟩ (accessed 2020-11-03).