はじめに

あなたは「HTML5」とは何かと問われたら何と答えるだろう。恐らく人によってその答えは違ってくるのではないだろうか。もし、あなたがウェブ開発者でなければ、答えに窮するのではないだろうか。もちろん、ウェブ開発者であっても、自信をもって答えられる人はいないはずだ。

HTML5という言葉だけが独り歩きし、今となっては、バズワード化している。HTML5を使ったサービスを考える企画マン、HTML5を使ってアプリを開発する開発者、HTML5を使ったソリューションを提案する営業マンやコンサルタント、さまざまな人々がHTML5という言葉を使って会話している。

よく考えると、HTML5を使ったサービス、HTML5を使ったアプリ、HTML5を使ったソリューションという表現は、非常に曖昧で実態がないように聞こえるのは私だけではないだろう。それは、世間一般に使われるHTML5という言葉の定義がないからだ。また、HTML5というバズワードをきっかけに、ウェブ技術がウェブ業界以外の業界で注目されていることも、混乱に拍車をかけている要因の1つだろう。

私はここで「HTML5」という言葉を定義するつもりはない。しかし、HTML5という漠然とした言葉を、実態を伴った技術をベースとして説明したいと思ったのが、本書を執筆するきっかけだった。

多くの人々は、HTML5で具体的に何ができるのかをよくわからないままに、HTML5という言葉を使っている。これまで実現できなかったことをできるようにする魔法の杖かのように、HTML5に期待を抱く人もいる。一方、実際にウェブ開発に携わった人であれば、まだ使えない、と失望感を抱く人もいる。ここまで両極端な感じ方が存在するのは、HTML5が何を意味し、そして、HTML5に何を期待しているかが違うからだ。同じ言葉にもかかわらず、そこから思い浮かべるイメージがここまで異なる言葉はほかにないだろう。

この両極端な捉え方は、どちらも間違っていない。あえて間違いを指摘するなら、現在と未来を区別せずに語っている点だろう。

まず、HTML5を魔法の杖と期待する方へ言おう。失望させてしまうかもしれないが、率直に言えば、HTML5は魔法の杖ではない。技術的にも最先端ではない。HTML5ではグラフィックス、ビデオ、オーディオ、リアルタイム通信、などなど、さまざまな新たなAPIが開発された。

しかし、すべては既存の技術の代替に過ぎない。多くの機能は、何年も前からAdobe FlashやMicrosoft SilverLightで実現できる。そのほかの機能も、ウェブから一歩離れれば、同様の機能は実現できた。

次に、HTML5は使えないと失望している方へ言おう。いま、あらゆるデバイスでワンソースマルチユースが実現できるはずもない。HTML5という標準化技術がある時点ですべて完成し、突然、すべてのブラウザーに実装されるはずもないからだ。標準化技術というものは、それがどこでも当たり前に使えるようになるには、それなりに時間がかかるものだ。そして、世間が言うHTML5が、いつかの時点で完成するわけでもない。これから永久に進化し続けるのだ。最先端の機能だけを見つめれば、いつの時代も、ワンソースマルチユースは実現しない。こなれた機能から徐々にワンソースマルチユースになるのだ。

いずれにせよ、ウェブの標準化は、アプリケーション基盤を目指していることは事実だ。従って、将来的には、これまでOSがネイティブアプリ向けに提供してきた機能が、ブラウザーから利用できる、または、ウェブ技術を使ったパッケージアプリとして利用できるようになるだろう。しかし、現時点では、まだアプリケーション基盤としては成熟しきっていない。

HTML5とうまく付き合うには、技術的な裏付けが必要だ。やみくもに最先端を求めても、そのプロジェクトは失敗することは目に見えている。HTML5にはどんな機能があるのか、どの機能がどのデバイスやブラウザーで動作するのか、パフォーマンスは実用レベルなのか、といった目利きが必要だ。

このような目利きができるようになるには、最低限、ウェブで実現できる機能を網羅的に知っておく必要がある。では、世間で言われるHTML5には、どれほどの機能があるのだろうか。結論から言うと、アプリケーション基盤として不足があるとはいえ、本書では説明しきれないほどに膨大だ。

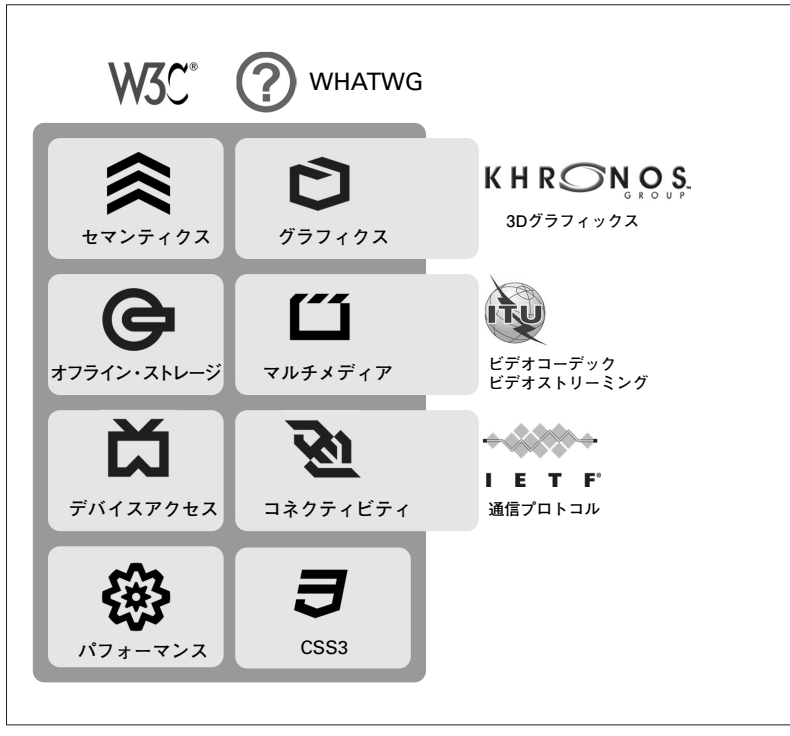
ここでは、おおざっぱに機能を分類してみよう。実は、ウェブの標準化団体であるW3Cが、HTML5をうまくカテゴリーに分類して、カテゴリーごとのロゴを用意している。そこに、HTML5と関連が深いほかの標準化団体を重ねてみよう。

この図はウェブ関連技術を機能面で分類している。基本的にウェブ技術の標準化はW3Cにて行われているが、正確には、HTML5の生みの親は「WHATWG」と呼ばれるブラウザーベンダーのコミュニティーだ。現在は、この両者の団体が協力しながら標準化作業を行っている。しかし、WHATWGがすべての機能を新たに生み出したわけではない。WHATWGが近年のウェブの進化の礎を築いたのは事実だが、W3C発のAPIも数多くある。

さらに、機能ごとに見ていくと、世間で言われているHTML5は、W3CやWHATWGだけでなく、そのほかさまざまな標準化団体と深く関連している。

3Dグラフィックスを扱うWebGLと呼ばれるAPIは、KHRONOSと呼ばれるグループが策定している。KHRONOSでは、3Dグラフィックスを扱うために、型付き配列をJavaScriptから扱う

図 HTML5の全体像



機能も策定している。これはバイナリーデータを効率的に扱うことができるため、W3Cが開発しているいくつかのAPIでも採用されている。WebGLに関しては本書では解説しないが、型付き配列については本書でも取り上げる。

H.264などのビデオコーデックはITU-T（国際電気通信連合 電気通信標準化部門）が策定したものだ。図には掲載していないが、ビデオコーデックには、WebMと呼ばれるGoogleが提供しているオープンな動画規格も存在する。さらにITU-Tでは、ビデオストリーミングの技術も標準化しており、本書で紹介するAPIの話と深く関わる。

ウェブは基本的にHTTPと呼ばれる通信プロトコルに依存しているが、これを策定しているのはIETFだ。さらに、本書でも紹介するWebSocketやWebRTCと呼ばれるリアルタイム双方向通信に関連する機能は、W3CにてAPIを開発しているものの、通信部分に関してはIETFが策定している。

このように、HTML5とは、さまざまな標準化団体、そして、それに関わる企業やディベロッパーコミュニティーが作り上げてきたウェブ関連の標準技術全体、そして、その進化そのものを指し示している。

これだけ多岐にわたる機能を持ったHTML5を理解するには、それだけハードルが高いのも事実だ。さまざまな標準化技術が絡み合っているゆえに、仕様書が1つにまとまっておらず、散在

しているのが、より状況を分かりにくくしている。さらに、W3Cに限定したとしても、非常に数多くの仕様に分散しており、必要な情報がどこに存在するのかを知るのも難しいだろう。もちろん、それらすべてが英語ゆえに、言葉の壁もある。そして、標準化団体で仕様が存在しているからといって、それが今すぐに使えるとも限らない。

本書では、主にW3Cが開発する機能にフォーカスを当てる。W3Cが開発している仕様は、大まかに宣言型の機能と命令型の機能に分類できる。宣言型とは、ウェブページの構造を作り出すマークアップ、スタイリングを宣言するCSS、ベクターグラフィックスを扱うSVGなどだ。一方、命令型の機能とは、分かりやすく言えば、プログラミングのことだ。特にJavaScriptから操作するための機能を指す。この機能は「API」という形で標準化されている。本書では、このAPIをできる限り網羅的に紹介する。

HTML5を理解する上で、通信プロトコルやコーデックなどの関連技術が重要なことは言うまでもない。しかし、HTML5で何ができるのかを理解したいのであれば、どんなAPIが存在するのかを知るのが最も分かりやすい。なぜなら、通信プロトコルやコーデックなどの関連技術は、APIを通して使うからだ。

本書は技術書であるが、完全解説本ではない。エッセンスだけをできる限りシンプルに解説する。ウェブ開発者でない方は、プログラムコードを理解できなくても、コメント文や本文を読めば、ある程度は何ができるかを把握できるようにしたつもりだ。開発者の方には、新たなAPIの利用イメージがつかめるよう、できる限りシンプルに必要なコードのみに限定して解説した。

HTML5は、ウェブに新たな機能を追加しただけでなく、旧来よりブラウザーに独自に実装されていたAPIも標準化している。しかし、利用できるブラウザーが限られていたこともあり、あまり知られていないAPIがいくつもあった。近年のウェブ標準化の流れのなかで、こういったAPIが標準化され、多くのブラウザーが実装した。このような古くて新しいAPIも紹介する。

ご存じのとおり、多くの標準化団体はデジュール標準を採用している。デジュール標準とは、標準策定を先に行い、それに従ってベンダーがそれを製品に実装する方式のことだ。特にハードウェアが関連する技術においては、デジュール標準でないと問題が起こってしまう。そのため、デジュール標準の世界では、いつにその仕様が標準化（確定）されるのかが重要になる。

一方、ウェブ関連技術のうち、特にW3Cが策定するAPIは、デジュール標準ではない。ブラウザーへの実装が先だ。もちろん、実装の前に仕様の提案が行われるが、標準化（勧告）に至るのは実装の後だ。先に実装されるということは、当然、標準化されるまでの間は互換性が維持されない可能性がある。さらに、標準化されたからといって、すべてのブラウザーにそれが実装されるとも限らないし、仮に実装されるにしても、その時期はバラバラだ。

よく、HTML5がいつ完成するのか、という話題を耳にする。はっきり言おう。それは愚問だ。

HTML5仕様が2014年に勧告になる予定だと聞いたことがある人も多いだろう。すでに紹介したとおり、HTML5とは、さまざまな仕様が関係するウェブ技術の総称だ。2014年に勧告の予定であるHTML5仕様というのは、世間が言うHTML5全体を表しているわけではない。全体の一部でしかないのだ。

確かにHTML5仕様の完成度が高まるということは喜ばしいことだが、実はすでにHTML5.1仕様の策定が始まっている。また、そのほかの仕様の勧告予定の時期はバラバラだ。そして、新たなAPIが雨後のタケノコのごとくわいてくるのだ。世間一般に言われるHTML5の標準化策定がいつ終わるのかと問われれば、永久に終わらないとしか言えない。

仕様策定に関わる人やブラウザーベンダーでない限り、いつ勧告になるのかは重要ではないはずだ。ブラウザーにどの機能が実装されているかを知る方がよっぽど重要なはずだ。もし気にするなら、いま、W3Cでどんな提案があるのか、そして、それがいつごろブラウザーに実装されそうか、だろう。

W3Cの標準化は、ソフトウェア開発の考え方に近づいている。それぞれの仕様はいずれ勧告に至るが、すぐにバージョンアップされる仕様も多い。まだ勧告に至っていないにもかかわらず、次のバージョンの仕様も検討しているくらいだ。HTML5の生みの親であるWHATWGは、すでにバージョンという考え方からすら脱却している。常に改良し続ける策定モデルを採用しているのだ。これをリビングスタンダードと呼ぶ。一方、W3Cは、ある段階で区切りを付け、それにバージョン番号を付ける。これをスナップショットモデルと呼ぶ。

W3Cの標準化は、ほかの標準化団体の方式とは異なり、非常に流動的な世界なのだ。デジュール標準の世界にいる人にとっては、理解に苦しむところがあるかもしれない。ウェブの世界では、どの技術（API）がどのブラウザーに実装されているか、いつ実装されそうか、すでに互換性を持った状態で実装されているのか、を常に意識する必要がある。ビジネス的には、その機能を実装したブラウザーの市場シェアがどれくらいなのかについても意識する必要があるだろう。

ウェブの世界においては、このような目利きが重要になってくる。本書も書籍である以上、執筆時点のスナップショットに過ぎない。本書の一部の記載は、あっという間に陳腐化してしまうだろう。しかし、本書がHTML5関連のAPIの理解の基礎となり、その後のキャッチアップのための土台なれば、著者としてこの上ない喜びだ。

最後に、本書を手にとって頂いた読者のみなさまに、この場を借りて厚く御礼申し上げたい。本書が少しでもみなさまの発展に貢献できれば幸いだ。

平成25年12月吉日

有限会社futomi 代表取締役

株式会社ニューフォリア 取締役 最高技術責任者

羽田野 太巳

羽田野 太巳（はたの ふとみ）

有限会社futomi 代表取締役

株式会社ニューフォリア 取締役 最高技術責任者

1993年 名古屋大学理学部数学科卒、日本電信電話（株）（NTT）入社。伝送系エンジニアとして通信系インフラの保守運用を経て、通信系SIとして企業通信系システム設計に従事。1999年 NTTぷららに出向、インターネット接続サービスおよびサーバーのシステム運用、サービス企画に従事し、IPTVサービスの立ち上げに携わる。

2004年独立後、（有）futomiを設立し、ウェブ・システム開発の傍ら、ウェブコンサルティングも手がける。

HTML5の気運が高まる以前からHTML5の探求を始め、HTML5専門サイト「HTML5.JP」を立ち上げ、HTML5の普及啓蒙に関わり、HTML5関連書籍や雑誌記事執筆も行う。

2011年（株）ニューフォリア 取締役（最高技術責任者）に就任し、HTML5を使ったサービスの開発やウェブベースのデジタルサイネージ・システムの研究開発の指揮も執る。また、W3CにてWeb-based Signage Business Groupの共同議長を務める。

主な著書

- **HTMLとJavaScriptではじめるWindowsストアアプリ開発入門（株式会社秀和システム）**
- **HTML5ガイドブック 増補改訂版（株式会社インプレスジャパン）**
- **徹底解説 HTML5 APIガイドブック ビジュアル系API編（株式会社秀和システム）**
- **徹底解説HTML5マークアップガイドブック 最終草案対応版（株式会社秀和システム）**

ほか

目次

Chapter

1

DOM

HTML5をはじめとしたウェブAPIを使うにあたり、DOMの理解は避けて通れない。HTML5といえば、派手で最新のAPIばかりが注目される。しかし、地味ではあるがDOMも時代とともに進化している。本章では、近年に新たに作られたDOM関連のAPI、そして、古くから仕様策定は行われていたものの、近年になってブラウザーに実装され、利用可能になったAPIにフォーカスを当てる。

1-1 DOMの最新版 DOM4

標準化団体 W3C-DOM4
API仕様書 http://www.w3.org/TR/dom/

JavaScriptを使ってリッチなウェブコンテンツを作る上で、DOM（Document Object Model）はその基礎となる最も重要なAPIと言えるだろう。DOMとはドキュメントの内容や構造やスタイルにアクセスしたりアップデートしたりできるAPIのことだ。DOMを知らずしてウェブアプリケーションを作ることはできない。ここではDOMの最新の仕様であるDOM4のAPIについて解説する。

1-1-1 DOM4の全体像

DOM4は、ドキュメントや要素に共通のDOM APIを規定している。table要素やvideo要素など個別のHTML要素のみに使えるAPIはHTML5仕様で規定している。DOM4は、すでに策定済みのDOM関連仕様を、HTMLでの利用を前提にECMAScript向けにアップデートし、1つの仕様として統合したものだ。

DOM4のベースとなる仕様

- DOM Level 3 Core
  http://www.w3.org/TR/DOM-Level-3-Core/
- DOM Level 3 Events
  http://www.w3.org/TR/DOM-Level-3-Events/
- DOM Level 2 Traversal and Range
  http://www.w3.org/TR/DOM-Level-2-Traversal-Range/
- Element Traversal
  http://www.w3.org/TR/ElementTraversal/

Coreには、ドキュメントやHTML要素などの情報を取得したり操作したりするためのAPIが規定されている。

Eventsには、マウスクリックやキーボードのキー押下といったイベントを捕捉したり、発生したイベントの詳細情報を取得したりするための枠組みと関連APIが規定されている。

Traversalは、名前の通りドキュメントのコンテンツを上から順に走査し、必要となるHTML要素などを見つけながら、何かしらの処理を行う枠組みを提供する。

Rangeは、ドキュメント内のコンテンツの特定の範囲を扱うAPIだ。要素の開始タグと終了タグで

区切った範囲だけでなく、テキストの途中から始めて、要素をまたいで、別の要素のテキストの途中で終わるような範囲も扱うことができる。これは主にウェブページのコンテンツのテキスト選択範囲を操作するために使われる。

前述のとおり、DOM4はいくつかの古い仕様がベースになっているため、すべてが最新のAPIというわけではない。いくつかのAPIはInternet Explorer 8まで実装されていなかったため、近年まであまり注目されることはなかった。しかしInternet Explorer 9以降、これらのAPIが実装されたこともあり、ようやく実用レベルになったと言えるだろう。

ここでは、近年に新たに作られたDOM関連のAPI、そして、古くから仕様策定は行われていたものの、近年になってブラウザーに実装され利用可能になったAPIや近年になって注目されてきたAPIにフォーカスを当てる。

1-1-2 class属性のトークンから要素を検索── getElementsByClassName()メソッド

ドキュメントツリーからHTML要素を検索する手段として、id属性の値から検索するgetElementById()メソッド、タグの名前から検索するgetElementsByTagName()メソッド、name属性の値から検索するgetElementsByName()メソッドが以前からよく使われてきた。

しかし、実際のHTMLのマークアップでは、class属性にトークンを入れることが多く、CSSでは、このトークンをフックにすることが多い。JavaScriptからもCSSと同様にclass属性のトークンをフックに検索したい要望はかねてから存在してきた。もちろん、ドキュメントツリーを総検索して、class属性のトークンから該当する要素を検索するスクリプトを自前で書くことも可能だが、ドキュメントのサイズが大きいとパフォーマンスがネックになる。

このような要望を受け、HTML5仕様ではgetElementsByClassName()メソッドが新たに規定された。その後、HTML5仕様から分離され、DOM4に移された。なお、おなじみのgetElementById()などのメソッドもDOM4で規定されている。

getElementsByClassName()メソッドは、近年のすべてのブラウザーに実装された。Internet Explorerはバージョン9から実装されている。getElementsByClassName()メソッドは、引数にclass属性のトークンを指定することで、該当する要素のリストを返す。

サンプルコード DOM4_getElementsByClassName.html

```
<div id="A" class="a w"></div>
<div id="B" class="b w"></div>

<script>
(function () {

showId( document.getElementsByClassName("a") ); // A
```

```
showId( document.getElementsByClassName("b") ); // B
showId( document.getElementsByClassName("w") ); // A B

function showId(el_list) {
    var id_list = [];
    for( var i=0, len=el_list.length; i<len; i++ ) {
        id_list.push(el_list.item(i).id);
    }
    console.log(id_list.join(" "));
}

})();
</script>
```

1-1-3 class属性のトークンへのアクセス ── classList

HTMLでは、数多くのHTML要素のclass属性にトークンがマークアップされている。トークンはスペース区切りでいくつでもセットすることができる。

```
<div class="a b c"></div>
```

JavaScriptからclass属性の値を文字列として取得することができる。しかし、個々のトークンを操作しようとすると、文字列操作を行う必要があり、やや面倒であった。たとえば、トークンbを削除するなら、"a b c"という文字列を半角スペースで分離して配列に入れ、"b"だけを除外した上で、半角スペースで区切った文字列として連結して、最後にdiv要素のclass属性の値としてセットする。この処理が多くなれば、パフォーマンスにも影響が出るだろう。

DOM4では、このclass属性のトークンをJavaScriptからリストとしてアクセスできるよう、要素のDOMオブジェクトにclassListプロパティを新たに規定した。もともとはHTML5仕様で規定されたものだが、現在は、DOM4に移されている。

要素のノードオブジェクトのclassListプロパティから、DOMTokensオブジェクトが得られる。このオブジェクトを通して、該当の要素のclass属性のトークンにアクセスする。

```
// div要素のノードオブジェクト
var div_el = document.getElementsByClassName("a").item(0);
// div要素のDOMTokenListオブジェクト
var tokens = div_el.classList;
```

このオブジェクトのプロパティとメソッドは次のとおりだ。

表 DOMTokensオブジェクトのメソッドとプロパティ

メソッドとプロパティ	説明
length	トークンの数を返す
item(index)	引数indexのインデックス位置に相当するトークンを返す
contains(token)	引数tokenに指定した文字列をトークンとして含んでいればtrueを、そうでなければfalseを返す
add(token)	引数tokenに指定した文字列をトークンとして追加する
remove(token)	引数tokenに指定したトークンを削除する
toggle(token)	引数tokenに指定したトークンが存在すれば削除し、存在しなければ追加する
toString()	トークンのリストを半角スペース区切りで連結した文字列を返す

サンプルコード DOM4_classList.html

```
<div id="A" class="a"></div>

<script>
(function () {

// div要素のノードオブジェクト
var div_el = document.getElementsByClassName("a").item(0);
// div要素のDOMTokenListオブジェクト
var tokens = div_el.classList;
// トークンbを追加
tokens.add("b");
console.log(tokens.toString()); // "a b"
// トークンaを削除
div_el.classList.remove("a");
console.log(tokens.toString()); // "b"
// トークンcを切り替え（実際には追加）
div_el.classList.toggle("c");
console.log(tokens.toString()); // "b c"
// トークンcの存在をチェック
console.log(tokens.contains("c")); // true

})();
</script>
```

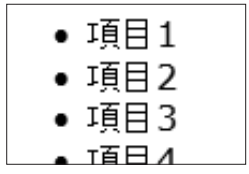
このように、DOMTokensオブジェクトにはトークン操作に求められるメソッドやプロパティが用意されているため、面倒な文字列操作をせずに、スマートにトークンが操作できるようになる。

1-1-4 ドキュメント断片の生成 ── DocumentFragment

DocumentFragmentとは、ドキュメントの断片を1つのオブジェクトとして扱う枠組みだ。通常、DOMといえば、要素が単位になる。しかし、JavaScriptからコンテンツを生成してドキュメントに挿入することを考えたとき、要素を1つずつ挿入するのは、非常にパフォーマンスが悪い。挿入したいコンテンツを1つの塊として扱い、それをまとめてドキュメントに挿入するのだ。

次のサンプルは、100個のli要素を生成し、それをまとめてul要素の中に挿入している。

サンプル表示例



サンプルコード DOM4_DocumentFragment.html

```
<ul id="list"></ul>

<script>
(function () {

// DocumentFragmentを生成
var fragment = document.createDocumentFragment();
// li要素を生成してDocumentFragmentに挿入
for( var i=0; i<100; i++ ) {
    // li要素を生成
    var li_el = document.createElement("li");
    li_el.textContent = "項目" + (i+1);
    // li要素をDocumentFragmentに挿入
    fragment.appendChild(li_el);
}
// DocumentFragmentをul要素に挿入
var ul_el = document.getElementById("list");
ul_el.appendChild(fragment);

})();
</script>
```

li要素を生成する都度に、それをul要素に挿入することもできる。しかし、このサンプルでは、事前にDocumentFragmentオブジェクトを生成し、その中に生成したli要素を挿入している。そして、最後にul要素にDocumentFragmentオブジェクトを挿入している。1つずつ要素ノードをドキュメントツリーに挿入すると、その都度にブラウザーではレンダリング処理が発生するため、パフォーマン

スが悪くなる。DocumentFragmentは、そのパフォーマンス問題を解決してくれる。

DocumentFragmentはかなり以前からどのブラウザーにも実装されていたが、あまり注目を浴びることはなかった。しかし、近年はスマートフォンなど、パソコンより非力な端末で、高度かつ複雑な処理をJavaScriptに頼るようになり、そのパフォーマンスは常に意識しなければならない。このような状況のなか、改めてDocumentFragmentが注目されはじめたと言えるだろう。

1-1-5 ノードの変化の捕捉── MutationObserver

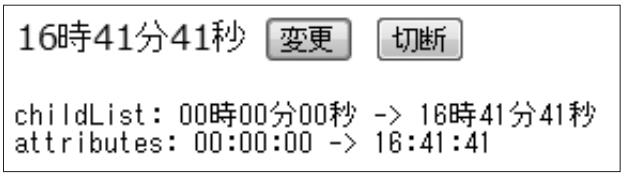
MutationObserverは、ドキュメントツリー上で発生したノードの変化を知らせてくれる枠組みだ。知ることができる変化は、子ノードの追加や削除、属性の変更、子となるテキストノードの値の変更だ。

同様の機能はDOM Level 3 EventsでMutation Eventとして策定され、多くのブラウザーに実装されていた。しかし、パフォーマンスなどの問題が存在したため、DOM Level 3 EventsのMutation Eventは非推奨（すでに存在する実装のためだけに仕様に残されているが、事実上の廃止）となり、新たに設計し直されたDOM4のMutationObserverに置き換えられた。

DOM4のMutationObserverは、すべてのメジャーブラウザーの最新版に実装されている。Internet Explorerはバージョン11から実装されている。

次のサンプルでは、time要素を使って時刻が表示されている。変更ボタンを押すと、現在の時刻に差し替わるが、time要素のテキストを更新するだけでなく、datetime属性の値も更新している。time要素に変更が発生すると、変更前の値と変更後の値が追記されていく。切断ボタンを押すと、time要素の変化の監視を終了し、それ以降は変更の状況の追記が行われなくなる。

サンプル表示例



サンプルコード DOM4_MutationObserver.html

```
<p>
    <time datetime="00:00:00">00時00分00秒</time>
    <button id="mod" type="button">変更</button>
    <button id="dis" type="button">切断</button>
</p>
<pre></pre>

<script>
```

```
(function () {

// time要素のノードオブジェクト
var time_el = document.getElementsByTagName("time").item(0);
// pre要素のノードオブジェクト
var pre_el = document.getElementsByTagName("pre").item(0);

// MutationObserverオブジェクトをを生成
// 引数には変化を検知したときに実行したいコールバック関数を指定
var observer = new MutationObserver(mutationCallback);

// 監視内容の定義(MutationObserverInitオブジェクト)
var opts = {
    childList: true,              // 子要素の追加および削除の監視
    subtree: true,                // 子孫ノードの変化の監視
    characterData: true,          // テキストの変更の監視
    characterDataOldValue: true,  // 変更前のテキストを記録
    attributes: true,             // 属性の変化の監視
    attributeOldValue: true,      // 変更前の属性値を記録
    attributeFilter: ["datetime"] // 監視対象の属性名
};

// 監視の開始
// MutationObserverオブジェクト(変数observer)のobserve()メソッドを呼び出す
//  * 第一引数:開始対象の要素ノードオブジェクト
//  * 第二引数:監視内容を定義したMutationObserverInitオブジェクト
observer.observe(time_el, opts);

// 変更ボタンを押したときの処理
document.getElementById("mod").onclick = function() {
    // 現在の時刻
    var dt = new Date();
    var h = ("0"+dt.getHours()).slice(-2);
    var m = ("0"+dt.getMinutes()).slice(-2);
    var s = ("0"+dt.getSeconds()).slice(-2);
    // time要素のコンテンツとして現在の時刻を挿入
    time_el.textContent = h + "時" + m + "分" + s + "秒";
    // time要素のdatetime属性の値を更新
    time_el.setAttribute("datetime", h + ":" + m + ":" + s);
};

// ノードに変化が発生したときの処理(MutationObserverのコールバック)
function mutationCallback(redords) {
    // 第一引数(変数redords)にはMutationRecordオブジェクトのリストが与えられる。
    // このリストから発生した変化をひとつずつ処理
    redords.forEach(function(rec) {
        // MutationRecordオブジェクト(変数 rec)
        var type    = rec.type;          // 発生した変化の種類
        var target  = rec.target;        // 変化があったノードのオブジェクト
        var removed = rec.removedNodes;  // 削除されたノードのリスト
        var added   = rec.addedNodes;    // 追加されたノードのリスト
        var attr    = rec.attributeName; // 変化があった属性の名前
```

```
        var old     = rec.oldValue;      // 変化前の値
        // 変化の内容をレポート
        var msg = type + ": ";
        if(type === "childList") {
            msg += removed.item(0).nodeValue + " -> " + added.item(0).nodeValue + "¥n";
        } else if(type === "characterData") {
            msg += old + " -> " + target.nodeValue + "¥n";
        } else if(type === "attributes") {
            msg += old + " -> " + target.getAttribute(attr) + "¥n";;
        }
        pre_el.appendChild(document.createTextNode(msg));
    });
}

// 切断ボタンを押したときの処理
document.getElementById("dis").onclick = function() {
    // MutationObserverを切断（監視の終了）
    observer.disconnect();
};

})();
</script>
```

このサンプルでは、変更ボタンを押すと属性値とテキストを更新する。しかし、MutationObserverは、監視対象のノードオブジェクトの変化を1つずつ報告するのではなく、まとめて報告する。そのため、変更ボタンを押してもmutationCallback関数は一度しか呼び出されない。その代わり、mutationCallback関数の第一引数（変数records）には、すべての変化を格納したリストが与えられるため、それを1つずつ処理する。

変化を格納したリスト（変数records）には、MutationRecordオブジェクト（変数rec）が格納される。このオブジェクトから変化に関する情報を取得することができる。

このサンプルでは、time要素のテキストが変更されると、MutationRecordオブジェクト（変数rec）のtypeプロパティの値は"characterData"だと期待するだろう。しかし実際には"childList"となる。これは、変更ボタンを押したときに実行される次のコードが原因だ。

```
time_el.textContent = h + "時" + m + "分" + s + "秒";
```

要素ノードオブジェクトのtextContentプロパティに値をセットすると、該当の要素の子ノードであるTextノードを削除してから、新たなTextノードを挿入しているからだ。では、このコードを次のコードに差し替えてみよう。

```
time_el.firstChild.nodeValue = h + "時" + m + "分" + s + "秒";
```

このコードは、Textノードを差し替えるのではなく、その値を直接変更することになる。この場合は、MutationRecordオブジェクト（変数rec）のtypeプロパティの値は"characterData"となる。

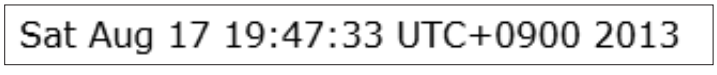
1-1-6 独自イベントの生成── CustomeEvent

通常、イベントはブラウザーが発生させるため、スクリプト側ではそれを受け取るだけになる。さらに、そのイベントとは、さまざまなウェブ標準仕様で規定されたイベント、または、ブラウザーが独自に実装したイベントに限られる。

しかし、場合によっては、スクリプト側で独自のイベントを作り出し、好きなタイミングでそのイベントを発生させたいこともあるだろう。それを実現するのが独自イベントだ。

次のサンプルは、myclockという独自のイベントを生成し、それをp要素で1秒おきに発生させる。イベントオブジェクトのdetailプロパティに現在日時を表すDateオブジェクトをセットし、それをイベントリスナー側が受け取れるようにする。

サンプル表示例



サンプルコード DOM4_CustomEvent_DOM3.html

```
<p id="now">-</p>

<script>
(function () {

// 1秒ごとにp要素でmyclockイベントを発生させる
var p_el = document.getElementById("now");
window.setInterval(function() {
    // Eventオブジェクトを生成
    // createEvent()メソッド
    // * 第一引数：イベントのインタフェース名（独自イベントなら"event"）
    var custom_event = document.createEvent("Event");
    // Eventオブジェクトを初期化
    // * 第一引数：イベント名を表す文字列
    // * 第二引数：バブリングさせるかどうかの真偽値（trueまたはfalse）
    // * 第三引数：キャンセル可能かどうかの真偽値（trueまたはfalse）
    custom_event.initEvent("myclock", true, false);
    // detailプロパティに現在の日時を表すDateオブジェクトをセット
    custom_event.detail = new Date();
    // p要素でmyclockイベントを発生させる
    // dispatch()メソッドの引数にはCustomeEventオブジェクトを指定する
    p_el.dispatchEvent(custom_event);
}, 1000);

// p要素にイベントリスナーをセット
```

```
p_el.addEventListener("myclock", function(event) {
    // Eventオブジェクトのdetailプロパティから現在日時を取得
    var dt = event.detail;
    // p要素のテキストとして現在日時を表示
    this.textContent = dt.toString();
}, false);

})();
</script>
```

このサンプルでは、documentオブジェクトのcreateEvent()メソッドを使って汎用的なイベントのオブジェクトを生成した。そして、initEvent()メソッドを使ってイベントの情報をセットした。この方法は、独自イベントだけでなく、マウス関連のイベントなど、規定のイベントの生成にも対応している。

```
var custom_event = document.createEvent("event");
custom_event.initEvent("myclock", true, false);
custom_event.detail = new Date();
```

この方法は、DOM Level 3 Eventsで規定されており、DOM4に引き継がれたものだ。この方法は、Firefoxの最新版、Internet Explorer 9以上に実装されている。2013年11月時点でChrome、Opera、Safariの最新版には実装されていない。

一方、DOM4では、独自イベント専用のCustomEventと呼ばれるAPIを新たに追加した。前述の3行のコードは、CustomEventであれば以下のシンプルなコードに書き換えられる。

サンプルコード DOM4_CustomEvent_DOM4.html

```
// CustomEventオブジェクトを生成
// * 第一引数：イベント名を表す文字列
// * 第二引数：イベントの初期化情報を格納したオブジェクト
var custom_event = new CustomEvent("myclock", {
    bubbles   : true,      // バブリングさせるかどうかの真偽値（trueまたはfalse）
    cancelable: false,     // キャンセル可能かどうかの真偽値（trueまたはfalse）
    detail    : new Date() // 独自データ
})
```

CustomEventインタフェースは、Chrome、Firefox、Opera、Safariの最新版には実装されているが、Internet Explorer 11には実装されていない。

1-1-7　コンテンツの範囲── Range

Rangeとは、名前のとおり、ドキュメント上のコンテンツの範囲を扱う枠組みだ。これまで見てきたDOM APIとは違って、HTML要素のテキストの途中を区切りとして扱うことができる点が大きな特徴だ。

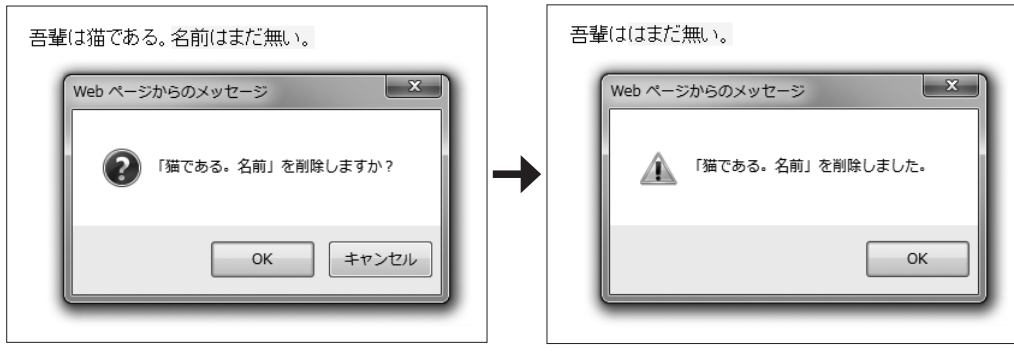
次のサンプルでは、特定のテキストの範囲を取り出し、それをドキュメントから削除する。当初は次のコンテンツがマークアップされている。

```
<p>吾輩は猫である。<mark>名前はまだ無い。</mark></p>
```

ここから「猫である。名前」のテキスト範囲を抽出する。そして、このテキスト範囲を削除して、次のような状態にする。

```
<p>吾輩は<mark>はまだ無い。</mark></p>
```

サンプル表示例



サンプルコード DOM4_Range.html

```
<p>吾輩は猫である。<mark>名前はまだ無い。</mark></p>

<script>
(function () {

// p要素のノードオブジェクト
var p_el = document.getElementsByTagName("p").item(0);
// mark要素のノードオブジェクト
var mark_el = p_el.getElementsByTagName("mark").item(0);
// Rangeオブジェクトを生成
var range = document.createRange();
// 範囲の開始位置を指定
// setStart()メソッド
// * 第一引数：対象のノードオブジェクト
```

```
// * 第二引数：オフセット
range.setStart(p_el.firstChild, 3);
// 範囲の終了位置を指定
// setEnd()メソッド
// * 第一引数：対象のノードオブジェクト
// * 第二引数：オフセット
range.setEnd(mark_el.firstChild, 2);
// 範囲のDocumentFragmentをコピーして取得
var fragment = range.cloneContents();
// DocumentFragmentのテキストを抽出
var text = fragment.textContent;
// DocumentFragmentのテキストを確認表示
var res = window.confirm("「" + text + "」を削除しますか？");
if(res === true) {
    // 該当の範囲を削除して削除した範囲を取得
    var fgmt = range.extractContents();
    // 削除した範囲をアラート表示
    window.alert("「" + fgmt.textContent + "」を削除しました。");
}
})();
</script>
```

コンテンツの範囲を作り出すためには、まずRangeオブジェクト（変数range）を生成する。次にRangeオブジェクトのsetStart()とsetEnd()メソッドを使って、範囲の開始位置と終了位置を定義する。ここで注意すべき点が2つある。

1つ目は、第一引数に指定する対象のノードオブジェクトだ。テキストの範囲を指定する場合、この第一引数に指定するのは、要素ノードオブジェクトではなく、その子となるTextノードオブジェクトだ。そのため、p要素およびmark要素のノードオブジェクトのfirstChildプロパティを使って、Textノードを指定している。

2つ目は、第二引数に指定するオフセットだ。オフセットとは、対象ノードがTextノードであれば、最初の文字の左側が0、最初の文字と2つ目の文字の間が1と数える。

Rangeは、後に紹介するText field selection APIと組み合わせて使うことが多い。実際の応用例は、そちらを参照してほしい。

1-1-8 ノードの走査── Traversal

通常、DOMツリー上の特定のノードにアクセスするためには、getElementById()、getElementsByTagName()などのメソッドや、要素ノードオブジェクトのparentNode、childNodes、firstChildなどのプロパティを使う。しかし、ドキュメント上の特定の要素の中にある子孫要素を1つずつ何かしらの処理をしたい場合、DOM Level 3で規定されたAPIだけでは非常に煩雑になることがある。特に要素の階層が深ければ、それだけ煩雑さも増す。

一方、Traversalは、より柔軟なノードの検索条件をフィルターとして定義して、見つかった都度に

何かしらの処理を行うことができる枠組みを提供する。そのため、前述のケースにおいては、非常に効率的かつ高速にDOMツリーの操作が行える。

Traversalは、メジャーブラウザーの最新版ですでに実装されている。Internet Explorerにはバージョン9から実装されている。

Traversalは、Element TraversalとDOM Traversalの2つに分類される。それぞれについて解説する。

要素だけを走査する── Element Traversal

要素トラバーサルは、Element Traversalという仕様で規定されたAPIだが、現在は、DOM4に置き換えられている。Element Traversalは、非常にシンプルなプロパティを要素ノードオブジェクトに追加しただけだ。

DOM Level 3 Coreで規定された要素ノードオブジェクトのchildNodes、firstChild、lastChild、previousSibling、nextSiblingプロパティはよく知られているが、使い勝手が悪いと感じた読者も多いだろう。それは、ホワイトスペースノードも対象になってしまうからだ。ホワイトスペースノードとは、マークアップにおいて終了タグと開始タグの間に入れられる改行やインデントを表す。

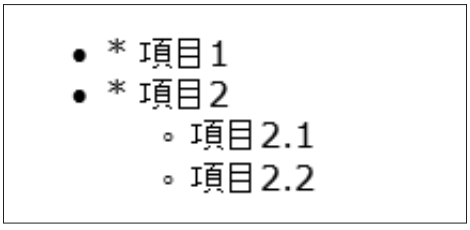
DOMはマークアップを忠実に再現するため、ホワイトスペースまでもがDOMツリーに含まれてしまう。しかし、JavaScriptからDOM操作を行う場合、ホワイトスペースを考慮することは皆無と言ってもよいだろう。Element Traversalは、要素ノードだけを探索するプロパティを追加した。

表 要素トラバーサルのプロパティ

プロパティ	説明
children	子要素のリストを返す
firstElementChild	子要素のうち最初の要素を返す
lastElementChild	子要素のうち最後の要素を返す
previousElementSibling	直前の兄弟要素を返す
nextElementSibling	直後の兄弟要素を返す

次のサンプルは、ul要素からli要素を探索し、それぞれのli要素のテキストの最後に文字を追加する。ここでは、firstElementChildとnextElementSiblingプロパティを使ってul要素の子ノードのうち、要素ノードのみを対象にしている。

サンプル表示例



サンプルコード **DOM4_Traversal_Element.html**

```
<ul id="list">
    <li>項目1</li>
    <li>項目2
        <ul>
            <li>項目2.1</li>
            <li>項目2.2</li>
        </ul>
    </li>
</ul>

<script>
(function () {

// ul要素の最初の子要素のノードオブジェクト
var child = document.getElementById("list").firstElementChild;
// ul要素の子要素すべてに対して反復処理
while( child ) {
    // li要素のテキストの最初に「* 」を追加
    child.firstChild.nodeValue = "* " + child.firstChild.nodeValue;
    // 次の兄弟要素を検索
    child = child.nextElementSibling;
}

})();
</script>
```

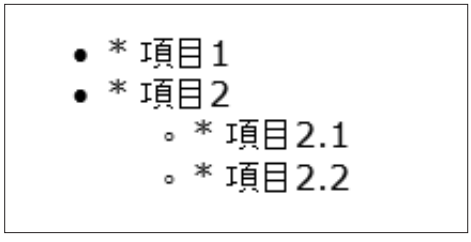
注意して欲しいのは、Element Traversalは、あくまでも子要素のみが対象だ。このサンプルでは、li要素の中にul要素がさらに存在しているが、このul要素の中にあるli要素は対象外だ。基本的に、Element Traversalは、ネストレベル（階層の深さ）が同じ要素に対して反復処理を行う際に役に立つ。

独自のフィルターで走査する── DOM Traversal

DOM Traversalは、前述の要素トラバーサルと比べて、より柔軟な走査を実現する。特に、要素のネストレベル（階層の深さ）に関係なく、DOMツリー順に該当の要素すべてに対して処理したい場合に役に立つ。フィルターは関数によって定義するため、より柔軟な抽出処理を作ることができる。

次のサンプルは、前述のサンプルとよく似ているが、前述のサンプルとは異なり、ネストレベルに関係なく、すべてのli要素に対して、テキストの始めに「*」を挿入する。

サンプル表示例



サンプルコード DOM4_Traversal_DOM.html

```
<ul id="list">
    <li>項目1</li>
    <li>項目2
        <ul>
            <li>項目2.1</li>
            <li>項目2.2</li>
        </ul>
    </li>
</ul>

<script>
(function () {

// NodeIteratorオブジェクトを生成
// * 第一引数：走査のルートとなる要素ノードオブジェクト
// * 第二引数：対象とするノードの種類
// * 第三引数：フィルター関数
// * 第四引数：null固定で指定する（IE対策）
var iter = document.createNodeIterator(
    document.getElementById("list"),
    NodeFilter.SHOW_ELEMENT,
    function(node) {
        // li要素のみに一致するフィルター
        if(node.nodeName === "LI") {
            return NodeFilter.FILTER_ACCEPT;
        } else {
            return NodeFilter.FILTER_REJECT;
        }
    },
    null
);

// ul要素の子要素すべてに対して反復処理
var child = null;
while( child = iter.nextNode() ) {
```

```
    // li要素のテキストの最初に「* 」を追加
    child.firstChild.nodeValue = "* " + child.firstChild.nodeValue;
}

})();
</script>
```

NodeIteratorオブジェクトを生成するdocument.createNodeIterator()メソッドには4つの引数を与える。

document.createNodeIterator()メソッドの第一引数には、操作のルートとなる要素ノードオブジェクトを指定する。つまり、ここで指定する要素の配下のコンテンツが走査の対象になる。

document.createNodeIterator()メソッドの第二引数には、検索対象となるノードの種類を数値で指定するが、数値では分かりにくいため、NodeFilterオブジェクトに規定された定数プロパティを使うとよいだろう。前述のサンプルでは要素のみを抽出したいため、NodeFilter.SHOW_ELEMENTを指定した。そのほか、すべてのノードを対象とするNodeFilter.SHOW_ALL、テキストノードを対象とするNodeFilter.SHOW_TEXTなどが利用できる。

document.createNodeIterator()メソッドの第三引数には、フィルター処理を行う関数を指定する。第二引数で対象とするノードの種類に関してはフィルタリングされるが、さらに何かしらの条件でフィルタリングしたい場合は、関数の中にフィルタリングの処理を入れる。この場合、検索対象であれば定数NodeFilter.FILTER_ACCEPTを、検索対象でなければ定数NodeFilter.FILTER_REJECTを返すようにする。

document.createNodeIterator()メソッドの第四引数には、固定でfalseを指定する。DOMトラバーサルは、もともとはDOM Level 2 Traversal and Range仕様で規定されたAPIで、この時点では4つの引数が規定されていた。しかし、DOM4で第四引数は廃止となった。ところが、Internet ExplorerはDOM Level 2 Traversal and Range仕様に基づいて実装されており、4つ目の引数を指定しないとエラーになってしまう。そのため、実際のコーディングにおいては、4つ目の引数にfalseを固定で指定する。

1-2 CSSセレクタによる要素の検索 Selectors API

標準化団体 W3C-Selectors API Level 1
API仕様書 http://www.w3.org/TR/selectors-api/

これまでHTMLドキュメント上のHTML要素にアクセスするために、getElementById()、getElementsByTagName()、getElementsByClassName()メソッドなどを使ってきたが、階層が複雑なDOMツリー上の特定の要素にアクセスする場合は、コードが複雑になりがちだった。特に、id属性、class属性がセットされていないHTML要素へのアクセスは、非常に面倒であった。

Selectors APIは、CSSセレクタを使って特定の要素にアクセスする手段を提供する。Selectors APIは、2つのメソッドを規定している。

表 Selectors APIのメソッド

メソッド	説明
querySelector(selectors)	引数selectorsに指定したCSSセレクタに該当する要素のうち、ドキュメントツリー上で最初の要素のノードオブジェクトを返す。該当の要素が見つからなければnullを返す
querySelectorAll(selectors)	引数selectorsに指定したCSSセレクタに該当するすべての要素をNodeListオブジェクトとして返す

これらメソッドは、documentオブジェクトでも要素ノードオブジェクトでも利用可能だ。さらに、DocumentFragmentオブジェクトでも利用できる。documentオブジェクトで使えば、ドキュメント全体から検索を行い、要素ノードオブジェクトで使えば、そのHTML要素の中から検索を行う。

次のサンプルは、table要素の中のtbody要素の中から見て偶数行のtr要素の中にあるtdのみを検索し、その行の背景色をグレーにする。table要素は次のようにマークアップされている。class属性に"t"が指定されている。そして、thead要素が1つ、tbody要素が2つ、tfoot要素が1つある。

```
<table class="t">
    <thead>
        <tr><th>A</th><th>B</th><th>C</th></tr>
    </thead>
    <tbody>
        <tr><td>1</td><td>2</td><td>3</td></tr>
        <tr><td>4</td><td>5</td><td>6</td></tr>
        <tr><td>7</td><td>8</td><td>9</td></tr>
        <tr><td>10</td><td>11</td><td>12</td></tr>
        <tr><td>13</td><td>14</td><td>15</td></tr>
    </tbody>
    <tbody>
```

```
        <tr><td>16</td><td>17</td><td>18</td></tr>
        <tr><td>19</td><td>20</td><td>21</td></tr>
        <tr><td>22</td><td>23</td><td>24</td></tr>
        <tr><td>25</td><td>26</td><td>27</td></tr>
        <tr><td>28</td><td>29</td><td>30</td></tr>
    </tbody>
    <tfoot>
        <tr><td>D</td><td>E</td><td>F</td></tr>
    </tfoot>
</table>
```

スクリプトを実行する前と後のレンダリング結果は次のとおりだ。

スクリプト実行前

A	B	C
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
19	20	21
22	23	24
25	26	27
28	29	30
D	E	F

スクリプト実行後

A	B	C
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
19	20	21
22	23	24
25	26	27
28	29	30
D	E	F

ここでは、Selectors APIを使わない場合と使う場合の両方を比べてみよう。まずはSelectors APIを使わない場合の一例だ。

サンプルコード Selectors_1.html

```
<script>
(function () {

var table_el_list = document.getElementsByClassName("t");
var table_el_num = table_el_list.length;
for( var i=0; i<table_el_num; i++ ) {
    var table_el = table_el_list.item(i);
    if(table_el.nodeName !== "TABLE") { continue; }
    var tbody_el_list = table_el.getElementsByTagName("tbody");
    var tbody_el_num = tbody_el_list.length;
    for( var j=0; j<tbody_el_num; j++ ) {
        var tbody_el = tbody_el_list.item(j);
        var tr_el_list = tbody_el.getElementsByTagName("tr");
        var tr_el_num = tr_el_list.length;
        for( var k=1; k<tr_el_num; k+=2 ) {
```

```
            var tr_el = tr_el_list.item(k);
            var td_el_list = tr_el.getElementsByTagName("td");
            var td_el_num = td_el_list.length;
            for( var m=0; m<td_el_num; m++ ) {
                var td_el = td_el_list.item(m);
                td_el.style.backgroundColor = "#aaaaaa";
            }
        }
    }
}

})();
</script>
```

まず、class属性からgetElementsByClassName()メソッドでtable要素を抜き出す。しかし、class属性から取り出した要素は必ずしもtable要素とは限らない。そのため、それが本当にtable要素かどうかをチェックする必要がある。それから、tbody要素をgetElementsByTagName()メソッドを使って取り出し、さらに、同様の方法でtr要素、td要素を取り出していく。

td要素を取り出すだけなら、table要素からgetElementsByTagName()を一度だけ使えば事足りるのだが、このサンプルでは、tbody要素に限定し、さらに、tbody要素の中で偶数行のtr要素に限定する必要がある。そのため、これだけ面倒なコードを書かざる得ない。

では次に、同じ処理をSelectors APIを使って書き直してみよう。

サンプルコード Selectors_2.html

```
<script>
(function () {

var td_el_list = document.querySelectorAll("table.t>tbody>tr:nth-of-type(even)>td");
var td_el_num = td_el_list.length;
for( var i=0; i<td_el_num; i++ ) {
    var td_el = td_el_list.item(i);
    td_el.style.backgroundColor = "#aaaaaa";
}

})();
</script>
```

ここでは、querySelectorAll()メソッドを使って必要とするtd要素を一発で取り出している。Selectors APIを使わないサンプルと比べて、圧倒的にコードがシンプルになる。さらに、Selectors APIは、DOM検索においてパフォーマンス向上にも寄与する。

Selectors APIは、すでに最新のメジャーブラウザーには実装されている。Internet Explorerにはバージョン8から実装されている。

1-3 XML文字列とXMLオブジェクトの相互変換
DOM Parsing and Serialization

標準化団体 W3C-DOM Parsing and Serialization
API仕様書 http://www.w3.org/TR/DOM-Parsing/

DOM Parsing and Serializationは、HTMLドキュメントやXMLドキュメントのマークアップ文字列をDOMオブジェクトに変換するAPIだ。XMLに関しては、DOMオブジェクトを文字列に変換することも可能だ。

すでにメジャーブラウザーすべてに実装されており、Internet Explorerはバージョン9から実装している。

次のサンプルは、XMLフォーマットの文字列をDOMオブジェクトに変換する。そして、そのDOMオブジェクトからXMLフォーマットの文字列を取り出し、それをブラウザーのコンソールに出力する。

サンプルコード DOMParser.html

```
// DOMParserオブジェクトを生成
var parser = new DOMParser();

// XMLドキュメントの文字列をパースしてDOMオブジェクトに変換
// parseFromString()メソッド
// * 第一引数：XMLまたはHTMLの文字列
// * 第二引数：ドキュメントタイプを表す文字列
var xml = parser.parseFromString("<a><b>テキスト</b></a>", "application/xml");

// XMLSerializerオブジェクトを生成
var serializer = new XMLSerializer();

// DOMオブジェクトをテキストに変換
// serializeToString()メソッド
// * 第一引数：XMLの文字列
var serialized = serializer.serializeToString(xml);
console.log(serialized); // <a><b>テキスト</b></a>
```

XMLフォーマットの文字列をDOMオブジェクトに変換するためには、まず、DOMParserオブジェクト（変数parser）を生成する。そして、DOMParserオブジェクトのparseFromString()メソッドを呼び出す。

このメソッドの第一引数にはXMLの文字列を指定し、第二引数にはドキュメントタイプを表す文字列を指定する。ドキュメントタイプを表す文字列はあらかじめ決められており、以下のタイプを指定することができる。

表 変換可能なドキュメントタイプ

ドキュメントタイプを表す文字列	説明
"text/html"	HTMLドキュメント
"text/xml"	XMLドキュメント
"application/xml"	XMLドキュメント
"application/xhtml+xml"	XHTMLドキュメント
"image/svg+xml"	SVGドキュメント

XMLのDOMオブジェクトを文字列に変換するためには、まず、XMLSerializerオブジェクトを生成（変数serializer）する。そして、XMLSerializerオブジェクトのserializeToString()メソッドを呼び出す。このメソッドの第一引数には、XMLのDOMオブジェクトを指定する。

1-4 スタイルへのアクセス CSSOM

標準化団体 W3C-CSSOM
API仕様書 http://www.w3.org/TR/cssom/

CSSOMとはCSS Object Modelのことを表し、DOM（Document Object Model）と同様に、CSSにアクセスするAPIを規定している。HTML要素に直接的に指定されたインラインスタイルだけでなく、link要素やstyle要素で組み込んだスタイルシートにもアクセスすることが可能だ。さらに、Media Queriesの情報にもアクセスすることができる。

もともと、CSSOMで規定されているAPIは、DOM Level 2 Styleという仕様で規定されたものだ。これを改良したり、不必要な機能を削除したり、求められてきた機能を追加したりして、置き換えられたのがCSSOMだ。

■ DOM Level 2 Style
http://www.w3.org/TR/DOM-Level-2-Style/

ここでは、CSSOMで規定されているAPIのうち、Media Queries、インラインスタイル、スタイルシート、コンピューティッドスタイルにアクセスするAPIについて紹介する。

1-4-1 メディアクエリ定義の取得── MediaList

HTMLドキュメントのスタイリングは、必ずしも、あらゆるデバイスに最適とは限らない。そのため、デバイスに応じてスタイルを切り替える手法が以前から存在した。最も古くから使われているのは、デバイスのタイプに応じてスタイルを切り替える手法だ。link要素を使って、CSSファイルを分離するなら、次のようにマークアップする。

```
<link rel="stylesheet" href="common.css" media="all">
<link rel="stylesheet" href="screen.css" media="screen">
<link rel="stylesheet" href="print.css" media="print">
```

また、スタイルシートの中に直接記述することも可能だ。

```
@media screen {
  * { font-family: sans-serif }
}
```

近年、スマートフォンやタブレットの登場により、デバイスのディスプレーのサイズが多様化したため、スタイルをスクリーンサイズなどでも切り替える枠組みも、メディアクエリに追加された。

■ Media Queries
http://www.w3.org/TR/css3-mediaqueries/

メディアクエリといえば、ディスプレーサイズに応じてスタイルを切り替える枠組みを指すことが多いが、実際には、旧来から使われているmedia属性の使い方も含めて仕様は定義されている。まずは、メディアクエリの基礎を簡単に紹介しよう。

メディアクエリの基礎

メディアクエリを使って、たとえば、スクリーンの横幅が480px以下の場合にだけ、特定のCSSファイルを適用したいなら、次のようにマークアップできる。

```
<link rel="stylesheet" media="screen and (max-width: 480px)" href="sp.css">
```

スクリーンのサイズだけではなく、スクリーンオリエンテーションに応じたスタイルの切り替えも可能だ。

```
<link rel="stylesheet" media="screen and (orientation:portrait)" href="portrait.css">
<link rel="stylesheet" media="screen and (orientation:landscape)" href=" landscape.css">
```

そのほか、スクリーンの縦横比や色深度に応じてスタイルを切り替える枠組みまである。

メディア定義にアクセスする── MediaListインタフェース

では、本題のAPIの解説を始めよう。CSSOMで規定されたAPIは、MediaListインタフェースだ。これは、link要素のmedia属性にマークアップされたメディア情報にアクセスするためのAPIだ。次のマークアップを例に説明する。

```
<link rel="stylesheet" media="screen and (max-width: 480px), print" href="sp.css" id="c1">
```

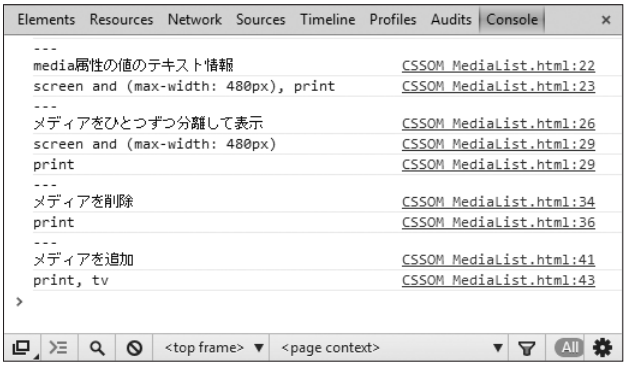
このlink要素のmedia属性には、カンマで区切って、2つのメディアがセットされている。この情報を分離して取り出したり、メディアを削除したり、メディアを追加したりしてみよう。

サンプルコード CSSOM_MediaList.html

```
<script>
(function () {

// MediaListオブジェクトを取得
var media_list = document.getElementById("c1").sheet.media;
// media属性の値のテキスト情報
console.log("---¥nmedia属性の値のテキスト情報");
console.log(media_list.mediaText);

// メディアをひとつずつ分離して表示
console.log("---¥nメディアをひとつずつ分離して表示");
for( var i=0, len=media_list.length; i<len; i++ ) {
    var media = media_list.item(i);
    console.log(media);
}
// メディアを削除
// deleteMedium()メソッド
// * 第一引数：削除したいメディアを表す文字列
console.log("---¥nメディアを削除");
media_list.deleteMedium("screen and (max-width: 480px)");
console.log(media_list.mediaText);

// メディアを追加
// appendMedium()メソッド
// * 第一引数：追加したいメディアを表す文字列
console.log("---¥nメディアを追加");
media_list.appendMedium("tv");
console.log(media_list.mediaText);

})();
</script>
```

処理の状況はコンソールログに表示されるが、その結果は次のとおりだ。

サンプル表示例

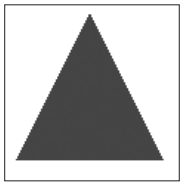


MediaListインタフェースは、定義されたメディアクエリの情報にアクセスするだけだ。実際にどの定義が採用されたかについては、後述のCSSOM View Moduleが扱うので、そちらを参照してほしい。

1-4-2 インラインスタイルの操作 ── CSSStyleDeclaration

インラインスタイルとは、HTML要素のstyle属性にセットされたスタイルのことだ。JavaScriptからは、次のようにインラインスタイルをセットすることは、読者のみなさんもよくご存じのことだろう。

サンプル表示例



サンプルコード CSSOM_ElementCSSInlineStyle.html

```
<div id="triangle"></div>

<script>
(function () {

// div要素のノードオブジェクト
var div_el = document.getElementById("triangle");
// div要素のCSSStyleDeclarationオブジェクト
var s = div_el.style;
// div要素にインラインスタイルをセット
s.width = "0px";
s.height = "0px";
s.borderBottom = "100px solid green";
s.borderLeft = "50px solid white";
s.borderRight = "50px solid white";

})();
</script>
```

HTML要素のノードオブジェクトのstyleプロパティから得られるオブジェクト（変数s）は、CSSStyleDeclarationインタフェースと呼ばれるAPIを実装しており、もう少しさらに詳細な制御が可能だ。

前述のサンプルでは、CSSStyleDeclarationオブジェクトを、あたかも一般的なオブジェクトのように、プロパティに値をセットしていた。しかし、CSSStyleDeclarationオブジェクトは、さまざまなプロパティやメソッドを備えている。

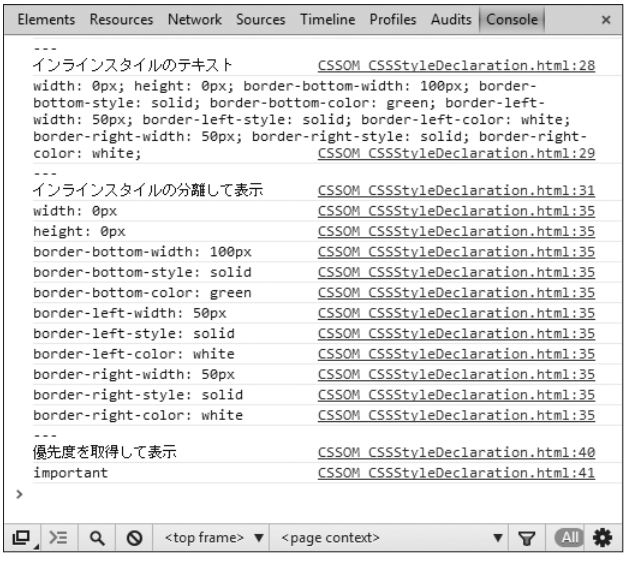
前述のコードに、以下のコードを追加する。

サンプルコード CSSOM_CSSStyleDeclaration.html

```
// インラインスタイルをテキスト表示
console.log("---¥nインラインスタイルのテキスト");
console.log(s.cssText);
// インラインスタイルを分離して表示
console.log("---¥nインラインスタイルの分離して表示");
for( var i=0; i<s.length; i++ ) {
    var name = s.item(i);                          // プロパティ名
    var value = s.getPropertyValue(name);          // 値
    console.log(name + ": " + value);
}
// インラインスタイルをセット
s.setProperty("border-bottom-color", "red", "!important");
console.log(s.getPropertyPriority("border-bottom-color"));
```

コンソールログには、次のような結果が表示される。

サンプル表示例



CSSStyleDeclarationオブジェクト（変数s）のlengthプロパティから、実際にセットされているスタイルの数が得られる。しかし、事前にセットした数より多いことにお気づきだろうか。たとえば、border-bottomプロパティに値が指定されると、ブラウザー内部では、border-bottom-width、border-bottom-style、border-bottom-colorプロパティに分離して処理する。CSSStyleDeclarationオブジェクトから操作するスタイル情報は、すべて、このように分離した形で扱うことになる点に注意してほしい。

CSSStyleDeclarationオブジェクトのgetPropertyValue()メソッドの第一引数にプロパティ名を指定して呼び出すと、その値が得られる。ただし、ここで指定するプロパティ名は、ハイフン付きの値だ。スタイルシートに記述する際に使うプロパティ名をそのまま指定する。たとえば、getPropertyValue()メソッドの第一引数に"borderBottomColor"を指定しても値は得られないので注意してほしい。

値をセットするsetProperty()メソッドも同様だ。第一引数には、ハイフン付きのプロパティ名を指定しなければならない。第二引数には値を指定する。第三引数には優先度を指定することができるが、これはオプションだ。指定する場合は、"!important"、または、"important"を指定する。

優先度は、getPropertyPriority()メソッドを使うと、その値が取得できる。ただし、!は除外される。

なお、CSSStyleDeclarationインタフェースは、すべてのメジャーなブラウザーに実装されている。Internet Explorerではバージョン9から実装されている。

1-4-3 スタイルシートの操作── CSSStyleSheet

style要素やlink要素によって組み込まれたスタイルへのアクセスを、順を追って解説しよう。まず、混乱を避けるために、style要素を例にとって、スタイルシートの用語を整理しよう。

```
<style>
p.msg {
    font-size: 120%;
    color: #444444;
}
</style>
```

このstyle要素の開始タグと終了タグの間の内容全体をスタイルシートと呼ぶ。そして、"p.msg { ... }"を「CSSルール」と呼ぶ。CSSルールのうち、"p.msg"の部分を「セレクタ」と呼び、CSSルールのうち"{ ... }"の部分を「宣言ブロック」と呼ぶ。宣言ブロックのうち、"font-size: 120%"と"color: #444444;"をそれぞれ「宣言」と呼ぶ。宣言のうち、"font-size"や"color"の部分を「プロパティ」と呼び、コロンが続き、その後に来る"120%"や"#444444"をそれぞれ「値」と呼ぶ。

すでに紹介したインラインスタイルを表すCSSStyleDeclarationオブジェクトは、宣言ブロックを表している。スタイルシートも、宣言ブロックを表すCSSStyleDeclarationオブジェクトにたどり着けば、それぞれの宣言を操作することができるようになる。

スタイルシートから宣言ブロックへアクセス

では、スタイルシートからCSSルールを経て、宣言ブロックに到達するまでの手順を見ていこう。

まず、style要素に組み込まれたスタイルそのものにアクセスするには、style要素のノードオブジェ

クトのsheetプロパティを使う。

```
var style_el = document.querySelector("style");
var style_sheet = style_el.sheet;
```

style要素のノードオブジェクトのsheetプロパティは、CSSStyleSheetオブジェクトを返す。これがstyle要素に組み込まれているスタイル全体を表している。

スタイルシートの中のCSSルールを取り出すには、まずCSSStyleSheetオブジェクト（変数style_sheet）のcssRulesプロパティから、CSSルールのリストを取り出す。

```
var rule_list = style_sheet.cssRules;
```

CSSStyleSheetオブジェクト（変数style_sheet）のcssRulesプロパティは、CSSRuleListオブジェクトを返す。これがstyle要素のスタイルシートにあるCSSルールのリストを表している。このリスト（変数rule_list）から個々のCSSルールを取り出す。

```
for( var i=0, len=rule_list.length; i<len; i++ ) {
    var rule = rule_list.item(i);
    ...
}
```

変数ruleには、CSSルールを表すCSSStyleRuleオブジェクトが格納されることになる。このCSSStyleRuleオブジェクトに組み込まれているプロパティによって、セレクタや宣言ブロックにアクセスする。

CSSStyleRuleオブジェクトには、以下のプロパティがセットされている。

表 CSSStyleRuleオブジェクトのプロパティ

プロパティ	説明
selectorText	セレクタの文字列を返す
style	宣言ブロックを表すCSSStyleDeclarationオブジェクトを返す

CSSStyleDeclarationオブジェクトさえ取り出せれば、あとは、宣言の読み取り、追加、削除は自由自在だ。以上を踏まえて、style要素に定義されたスタイルシートの情報を取り出してみよう。

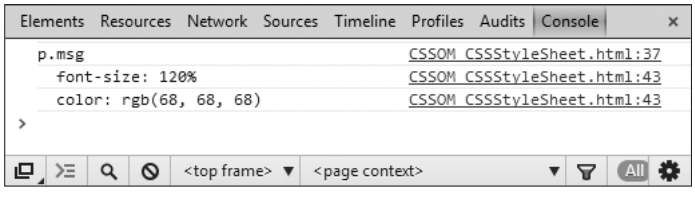
サンプルコード CSSOM_CSSStyleSheet.html

```
<script>
(function () {

// style要素のノードオブジェクト
var style_el = document.querySelector("style");
```

```
// CSSStyleSheetオブジェクトを取得
// ・style要素のスタイルシート全体を表す
var style_sheet = style_el.sheet;
// CSSRuleListオブジェクトを取得
// ・CSSルールをリストとして取得
var rule_list = style_sheet.cssRules;
// CSSStyleRuleオブジェクトを取得
for( var i=0, len=rule_list.length; i<len; i++ ) {
    // CSSStyleRuleオブジェクト
    // ・CSSルールを表す
    var rule = rule_list.item(i);
    // CSSルールのセレクタ
    var selector = rule.selectorText;
    console.log(selector);
    // CSSルールの宣言ブロック（CSSStyleDeclaration）
    var s = rule.style;
    for( var j=0; j<s.length; j++ ) {
        var name = s.item(j);
        var value = s.getPropertyValue(name);
        console.log("  " + name + ": " + value);
    }
}

})();
</script>
```

このスクリプトを実行すると、コンソールログには次のような結果が表示される。

サンプル表示例



このサンプルではstyle要素を例に挙げたが、外部CSSファイルを読み込むlink要素も、まったく同様に処理することが可能だ。

CSSルールの追加と削除

CSSルールの情報は読み取るだけでなく、追加や削除も可能だ。多くのシーンでは、インラインスタイルの操作で事足りるわけだが、スタイルシートを直接操作する方が効率的な場合もある。次のサンプルには、class属性がセットされたtable要素がマークアップされている。

サンプルコード table要素のマークアップ

```
<table class="t">
    <thead>
        <tr><th>A</th><th>B</th><th>C</th></tr>
    </thead>
    <tbody>
        <tr><td>1</td><td>2</td><td>3</td></tr>
        <tr><td>4</td><td>5</td><td>6</td></tr>
        <tr><td>7</td><td>8</td><td>9</td></tr>
        <tr><td>10</td><td>11</td><td>12</td></tr>
        <tr><td>13</td><td>14</td><td>15</td></tr>
        <tr><td>16</td><td>17</td><td>18</td></tr>
        <tr><td>19</td><td>20</td><td>21</td></tr>
        <tr><td>22</td><td>23</td><td>24</td></tr>
        <tr><td>25</td><td>26</td><td>27</td></tr>
        <tr><td>28</td><td>29</td><td>30</td></tr>
    </tbody>
</table>
```

また、style要素には次のスタイルが定義されている。

サンプルコード style要素のマークアップ

```
<style>
table.t { border: 2px solid black; border-collapse: collapse; }
table.t th,td { width: 100px; text-align: center; }
table.t thead th { background-color: #333333; color: white; }
table.t tbody td { border-bottom: 1px solid black; }
table.t>tbody>tr:nth-of-type(odd)>td { background-color: #cccccc; }
</style>
```

このスタイルにより、table要素は次のようにレンダリングされる。

図 レンダリング結果

A	B	C
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
19	20	21
22	23	24
25	26	27
28	29	30

今の段階では、style要素に定義されたスタイルによって奇数行だけ背景色がグレーになっている。

これをスクリプトから偶数行だけ背景色がグレーになるよう変更してみよう。

サンプルコード CSSOM_insertRule.html

```
<script>
(function () {

// style要素のCSSStyleSheetオブジェクトを取得
var style_sheet = document.querySelector("style").sheet;
// CSSルールを削除
// deleteRule()メソッド
// ・第一引数：CSSルールのインデックス番号
style_sheet.deleteRule(4);
// CSSルールを追加
// insertRule()メソッド
// ・第一引数：CSSルールを表す文字列
// ・第二引数：CSSルールのインデックス番号
var rule_text = "table.t>tbody>tr:nth-of-type(even)>td { background-color: #cccccc; }";
style_sheet.insertRule(rule_text, 4);

})();
</script>
```

まず、style要素のノードオブジェクトのsheetプロパティからCSSStyleSheetオブジェクト（変数style_sheet）を取得する。このオブジェクトのdeleteRule()メソッドを呼び出すと、個別のCSSルールを削除することができる。

このメソッドの第一引数にはCSSルールのリストにおけるインデックス番号を指定する。奇数行のtd要素の背景色をセットしているCSSルールは、style要素では5番目に記述されている。従って、CSSルールのリストにおけるインデックス番号は4だ。

CSSルールを追加したい場合は、insertRule()メソッドを呼び出す。このメソッドの第一引数には、CSSルールを表す文字列を指定する。第二引数には、挿入したい位置を表すインデックス番号を指定する。

ここでは、偶数行のtd要素の背景色をグレーにするCSSルールを表す文字列を第一引数に指定する。第二引数には、先ほど削除したCSSルールと同じインデックス番号を指定する。

図 スクリプト実行後のレンダリング結果

A	B	C
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
19	20	21
22	23	24
25	26	27
28	29	30

これで、背景色がグレーになる行が奇数行から偶数行に切り替わる。実質的に数行のコードで実現できる。このサンプルにはtable要素は1つしかなく、その行数も10行程度しかない。しかし、同じclass属性値を持ったtable要素がいくつもマークアップされ、かつ、各テーブルの行数が100行ほどあると仮定してみよう。旧来どおり、DOMアクセスによって、td要素のインラインスタイルを1つずつ操作することも可能だが、非常に効率が悪いだろう。

なお、スタイルシートへのアクセスに関しては、すべてのメジャーなブラウザーに実装されている。Internet Explorerではバージョン9から実装されている。

1-4-4 最終的に適用されたスタイルの取得 ――コンピューティッドスタイル

コンピューティッドスタイルとは、ブラウザーが最終的に適用したスタイルのことだ。ご存じのとおり、CSSは、ブラウザーがデフォルトで持っているデフォルトスタイル、そして、ウェブ制作者がlink要素やstyle要素で組み込むオーサースタイル、そして最後に、オーサースタイルの仲間であるが、インラインスタイルがある。これらの順番でスタイルを重ねながら適用して、最終的なスタイルができあがる。

この最終的なスタイル、つまり、ページ閲覧者が目にしているスタイルをJavaScriptから取得することができる。もちろん、オーサースタイルやインラインスタイルで指定されなかったスタイルもすべて取得可能だ。

次のサンプルは、p要素のコンピューティッドスタイルを取得している。style要素では、該当のp要素に対してfont-sizeのみがセットされている。さらに、疑似要素を使って、最初の一文字のサイズを大きくしている。

サンプルコード CSSOM_getComputedStyle.html

```
<style>
#msg { font-size: 12px; }
#msg::first-letter { font-size: 24px; }
```

```
</style>

<p id="msg">今日は快晴です。</p>

<script>
(function () {

// p要素のノードオブジェクト
var p_el = document.getElementById("msg");
// p要素のコンピューティッドスタイルのCSSStyleDeclarationオブジェクトを取得
// getComputedStyle()メソッド
// * 第一引数：対象の要素のノードオブジェクト
// * 第二引数：疑似要素の文字列（オプション）
var s = window.getComputedStyle(p_el);
console.log(s.getPropertyValue("font-size")); // "12px"
console.log(s.getPropertyValue("color"));     // "rgb(0, 0, 0)"
// 疑似要素
var pseudo_s = window.getComputedStyle(p_el, "::first-letter");
console.log(pseudo_s.getPropertyValue("font-size")); // "24px"
console.log(pseudo_s.getPropertyValue("color"));     // "rgb(0, 0, 0)"

})();
</script>
```

特定の要素に適用されているコンピューティッドスタイルを取得するのはとても簡単だ。windowオブジェクトのgetComputedStyle()メソッドを呼び出すだけだ。第一引数には、要素のノードオブジェクトを指定する。もし疑似要素のスタイルを取得したい場合は、第二引数に疑似要素の文字列を指定する。このサンプルでは、"::first-letter"をセットしている。

getComputedStyle()メソッドから得られるのは、CSSStyleDeclarationオブジェクトだ。要素のノードオブジェクトのstyleプロパティから得られるオブジェクトと同じインタフェースで、使い方はまったく同じだ。これで、あらゆる適用スタイルを取得できるようになる。

getComputedStyle()メソッドは、メジャーブラウザーすべてで実装されている。Internet Explorerではバージョン9から実装されている。ただし、疑似要素のコンピューティッドスタイルの取得に関しては、Chrome、Firefox、Safari、Operaがサポートしているが、Internet Explorer 11は、getComputedStyle()メソッドの第二引数を無視する形で処理される。