第2章 API とライブラリー

この章では、JavaScript で標準的に提供される（DOM 以外の）API と、さまざまなライブラリーなどについて紹介する。

2.1 jQuery について

JavaScript のライブラリー jQuery は、ブラウザー間の振舞いの違いを吸収するなどの、ユーティリティー関数を集めたものである。「$」という名前の変数を多用するので、jQuery を使っているプログラムはたいてい一目でわかる。

しかし ES6 くらいからライブラリーを使わない JavaScript （＿＿＿＿と呼ぶことがある）でもブラウザー間の振舞いの違いがかなり改善されており、jQuery の必要性は減ってきている。

この資料では jQuery は使っていない。

2.2 Web APIs

DOM 以外にも ブラウザー上で動作する JavaScript のプログラムからは多くの API (Application Programming Interface) を利用することができる。API はブラウザーや OS の提供する様々なサービスをプログラムから利用するための接続口である。

以下の節では、良く使われる Web API をいくつか紹介する。

Canvas

Web ページでグラフィックス（図形）を描画するための API である。HTML に＿＿＿＿というタグでキャンバス（描画のための領域）を確保する。

```
<canvas id="canvas" width="340" height="210">
</canvas>
```

このキャンバスから、描画のためのオブジェクト Context を取得し、そのメソッドを呼び出して描画する。

ファイル canvas.js

```
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext('2d');
const w = 340, h = 210;
const num = 24;
const xspan = w / num, yspan = h / num;
for (let i = 0; i < num; i++) {
  const x = xspan * i, y = yspan * i;
  console.log(x, y);
  ctx.beginPath();
  ctx.strokeStyle = `hsl(${i * 15}, 100%, 50%)`;
  ctx.lineWidth = 2;
```

```
    ctx.moveTo(x, 0);
    ctx.lineTo(0, h - y);
    ctx.stroke();
  }
```

JSON と Base64

JSON (JavaScript Object Notation) と Base64 は Web アプリケーションで通信やデータの保存によく使われるので、ここで紹介する。

名前	説明
`JSON.parse(`*str*`)`	文字列をパースしてオブジェクトへ
`JSON.stringify(`*obj*`)`	オブジェクトからそれを表す文字列へ
`window.atob(`*b64*`)`	Base64 表現から元の生データへ
`window.btoa(`*str*`)`	生データから Base64 表現へ

データ URL スキーム

データ URL は `http:`, `https:` などの外部へのリンクとは異なり、データをファイル中にインラインで埋め込むことができる URL で `data:` という接頭辞から始まる。例えば `<img src='`*Base64*`エンコードしたデータ'>` のように使うと、  のような画像のデータを HTML 中に埋め込むことができる。

次の例では `data` という配列に計算したビットマップ画像のデータ（計算部分は割愛）から、 `btoa` 関数を利用してデータ URL スキームを生成し、`img` タグの `src` 属性に設定している。

ファイル bmp.js

```
const bstr = data.map(c =>
                String.fromCharCode(c)).join("");
const astr = window.btoa(bstr);
const para = document.getElementById("para");
const img  = document.createElement("img");
img.src = `${astr}`;
para.appendChild(img);
```

BigInt

BigInt は通常の数値では表せないような大きな整数値を表すためのデータ型である。（通常の数値は倍精度の浮動小数点数なので、安全に表せる最大の整数は $2^{53}-1$ でだいたい 9000 兆である。）

BigInt は、整数リテラルに接尾辞 `n` をつけるか、`BigInt` 関数に整数または文字列の値を渡すことで生成することができる。下の例は 100! を計算している。

ファイル bigint.js

```
const n = 100;
let fact = 1n;
for (let i = 1; i < n; i++) {
    fact *= BigInt(i);
}
```

```
  document.getElementById('output').innerHTML = fact;
```

Fetch

Ajax は _____ JavaScript + XML の略で、JavaScript からページの遷移を伴わないサーバーとの通信を利用するプログラミング手法である。（名前にXMLが入っているが、必ずしも XML を使うとは限らない。）従来 `XMLHttpRequest` オブジェクトが使われていたが、2010 年代後半以降 Fetch という新しい API が Internet Explorer を除くほとんどのモダンブラウザーで利用可能になってきている。（利用できないブラウザーに対しても Polyfill が存在する。）ここでは Fetch を紹介する。 この API の中心となるのは `fetch` メソッドである。このメソッドは _____ というオブジェクトを返す。

Polyfill とは古いブラウザーで、新しい（JavaScript の）機能を使えるようにするためのコードである。

以下の例はサーバー側としては「`/find?name=`色名」のような URL でアクセスされると、その色のコードを 16 進 の 6 桁の整数で返すような振る舞いを想定している。また、サーバーが `localhost` の 8888 番ポートで稼働していると仮定している。

ファイル ajax.html

```
  <body>
    <input type="text" value="黒" id="name" />
    <p id="para"></p>
  </body>
```

ファイル ajax.js

```
const url  = "http://localhost:8888/find";
const name = document.getElementById("name");
const para = document.getElementById("para");
name.addEventListener("change", ev => {
  const n = ev.target.value;
  fetch(url + "?name=" + n, {
    method: "GET", mode: "cors"
  }).then(res => {
    if (!res.ok) {
      throw new Error(`HTTP error: ${res.status}`);
    }
    return res.text();
  }).catch(reason => {
    para.append(`request failed by ${reason}`);
  }).then(text => {
    const span = document.createElement("span");
    span.style.color = "#" + text;
    span.innerHTML = n;
    para.append(span);
  });
});
```

ここで `fetch` の第 1 引数はサーバーの URL である。第 2 引数のオブジェクトには次のようなプロパティーを指定する。

`method`
: `"GET"`, `"POST"` などリクエストのメソッドを指定する。

`headers`
: リクエストに追加するヘッダーを指定する

`mode`
: リクエストのモードを指定する。 クロスオリジンリクエストのときは、ここに `"cors"` を指定する。

`body`
: リクエストの本体を指定する。このデータを作るときはサーバー側が受け取れる形式に合わせて FormData（`multipart/form-data` の場合）、 URLSearchParams（`application/x-www-form-urlencoded` の場合）、 あるいは前述の `JSON.stringify`（`application/json` の場合）などを使うと便利である。

> クロスオリジンリクエストとは、大雑把に言えば Web ページのあるホストとは別のホストへのリソースの要求である。セキュリティー上の理由で、通常はクロスオリジンリクエストはブラウザーによってブロックされる。要求されたホスト側の許可がある場合のみ、アクセスが可能になる。

戻り値の Promise は非同期処理に使われるオブジェクトである。サーバーとの通信のように、待たされる可能性のある処理を表している。 `Promise` オブジェクトの `then` メソッドは非同期処理が成功したときに呼び出されるコールバックを登録する。また、`catch`メソッドは失敗したときに呼び出されるコールバックを登録する。この 2 つのメソッドは、また `Promise` を返すので、さらに、`then` メソッドや`catch`メソッドを連鎖させることができる。

さらに、ECMAScript 2017 から `Promise` を使ったプログラムは `async` と `await` というキーワードを使って、コールバックを使わない、自然なスタイルで記述できるようになった。詳細の説明は割愛するが、上記のプログラムを次のように書き換えることができる。

ファイル async.js

```
2  const url = "http://localhost:8888/find";
3  const name = document.getElementById("name");
4  const para = document.getElementById("para");
5  name.addEventListener("change", async ev => {
6    try {
7      const n = ev.target.value;
8      const res = await fetch(url + "?name=" + n, {
9        method: "GET", mode: "cors"
10     });
11     if (!res.ok) {
12       throw new Error(`HTTP error: ${res.status}`);
13     }
14     const text = await res.text();
15     const span = document.createElement("span");
16     span.style.color = "#" + text;
17     span.innerHTML = n;
18     para.appendChild(span);
19   } catch (reason) {
20     para.append(`request failed by ${reason}`);
21   }
22 });
```

コールバック関数 `then` を使って受け取っていた部分が、 `await` というキーワードを付加するだけで、普通の値のように扱えていることがわかる。コールバック関数 `catch` を使っていた部分は、 try ～ catch 文を使って表されている。

WebSocket

サーバーとの双方向通信、特にサーバーからの＿＿＿＿（クライアントからの要求によらず、サーバー側の都合とタイミングで、サーバーからクライアントに送る通信）を扱うための API である。

ファイル ws.html

```
<body>
  <input id="inp" type="text" value="Hello!" />
  <p id="para"></p>
</body>
```

ファイル ws.js

```
  const conn = new WebSocket("ws://127.0.0.1:8889");
  const para = document.getElementById("para");
  const inp  = document.getElementById("inp");

  conn.addEventListener("open",
       e => console.log("open"));
  conn.addEventListener("close",
       e => console.log("close"));
  conn.addEventListener("error",
       e => console.log("error"));
  conn.addEventListener("message", e => {
    console.log(e.data);
    const n = document.createElement("span");
    n.innerHTML = e.data;
    para.append(n);
    para.appendChild(document.createElement("br"));
  });

  inp.addEventListener("change", e => {
    conn.send(e.target.value);
  });
```

サーバー側との接続を確立 (`new WebSocket(…)`) したあとは、その接続からのメッセージの到着などのイベントに対して `addEventListener` メソッドでリスナーを登録する。また、接続に対して＿＿＿メソッドで、サーバー側にデータを送信する。

サーバー側は、送られたメッセージに応じてメッセージを返し、また、ときどき別のメッセージをプッシュする、というものを想定している。上の例ではサーバーが `localhost` の 8889 番ポートで稼働していると仮定している。

WebStorage

ページの再読み込みやページ遷移を行っても持続するオブジェクトで、＿＿＿＿＿＿と＿＿＿＿＿＿の 2 つがある。前者は、同じオリジン（プロトコルとホストとポートの組）で共有され、ブラウザーを一旦閉じても続けて存在する。後者は、一つのタブが開いている間に持続し、同じオリジン

のページの間で共有することができる。どちらも、通常のオブジェクトとしてプロパティーにアクセスすることができるが、`getItem`, `setItem`, `removeItem` などの専用のメソッドを使うことが推奨されている。

次の例では、背景色をユーザーに入力してもらい、それを次に開いたときも覚えているようにするために `localStorage` を使っている。

ファイル webstorage.html

```
<body>
  <p>Hello!</p>
  <input type="text" id="cinput" value="#00FF00" />
</body>
```

ファイル webstorage.js

```
  const cinput = document.getElementById("cinput");

  if (!localStorage.getItem("color")) {
    localStorage.setItem("color", "#00FF00")
  }
  const color = localStorage.getItem("color");
  cinput.value = color;
  document.body.setAttribute("bgcolor", color);

  cinput.addEventListener("change", ev => {
    const c = ev.target.value;
    localStorage.setItem("color", c);
    document.body.setAttribute("bgcolor", c);
  });
```

File

JavaScript がファイルを操作するための API である。ファイルは `<input type="file" …>` などの HTML 要素、ドラッグアンドドロップなどを通して取得される。

ファイル file.html

```
<body>
  <input id="inputf" type="file" multiple />
  <p id="para"></p>
</body>
```

ファイル file.js

```
  const inputf = document.getElementById("inputf");
  const p      = document.getElementById("para");
  inputf.addEventListener("change", ev => {
    const files = ev.target.files;
    for (let i = 0; i < files.length; i++) {
      const f = files[i];
      p.append(f.name);
      p.appendChild(document.createElement("br"));
      if (f.type.match(/text.*/)) {
        const el = document.createElement("pre");
        el.style.borderStyle = "solid";
        p.appendChild(el);
        const reader = new FileReader();
        reader.onload = ev => {
          el.innerHTML = ev.target.result;
        };
```

```
        reader.readAsText(f);
      } else if (f.type.match(/image.*/)) {
        const el = document.createElement("img");
        p.appendChild(el);
        p.appendChild(document.createElement("br"));
        const reader = new FileReader();
        reader.onload = ev => {
          el.setAttribute("src", ev.target.result);
        };
        reader.readAsDataURL(f)
      } else {
        const el = document.createElement("span");
        el.style.color = "red";
        el.innerHTML = "ファイルの種類が判定できません。";
        p.appendChild(el);
        p.appendChild(document.createElement("br"));
      }
    }
  });
```

ファイルの読み込みは、待ち時間が発生するので、非同期に実行する。つまり FileReader の onload に読み込んだときに呼び出されるコールバックを登録して、取得したい形式に応じて readAsText, readAsDataURL, readAsArrayBuffer などのメソッドを呼出す。

Worker

時間がかかってしまう計算をバックグラウンドで実行するための仕組みである。通常の JavaScript のプログラムはシングルスレッドで実行されるので、時間がかかる部分があると他の必要な処理が滞ってしまう。時間のかかる処理を Worker に任せておけば、待たされずに必要な処理を行うことができる。ただし、Worker ができることには制限があり、例えば Worker の中で＿＿＿＿＿＿することはできない。

Worker は、それが実行する JavaScript ソースファイルを指定することで生成する。Worker と Worker を生成したスクリプトの間ではメッセージをやり取りする。Worker にメッセージを送るには＿＿＿＿＿＿＿＿というメソッドを用いる。逆に Worker からメッセージを受取るには＿＿＿＿＿＿というイベントにリスナーをセットしておく。

逆に Worker 側のプログラムでは、＿＿＿＿＿＿＿＿という変数にメッセージを受け取ったときの処理を記述しておく、呼出し側にメッセージを送るのは、同じく postMessage というメソッドである。

以下の例は、時間のかかるかもしれない計算—素因数分解を Worker を使って記述している。

ファイル worker.html

```
<body>
  <input id="number" type="text" size="16">
<p>例: <code>10000007600001443</code>,
<code>100000001600000063</code></p>
  <input id="send" type="button" value="send!" />
```

```
  <p id="para" style="white-space:pre;"></p>
</body>
```

ファイル workerCreate.js

```
  const num  = document.getElementById("number");
  const p    = document.getElementById("para");
  const send = document.getElementById("send");
  const myWorker = new Worker("worker.js");

  send.addEventListener("click", ev => {
    const n = num.value.trim();
    if (!(/^[0-9]+$/.test(n))) {
      p.append(num.value + "は正の整数ではありません\n");
      return;
    }
    myWorker.postMessage(n);
    p.append(n, " = ");
  });

  myWorker.addEventListener("message", e => {
    p.append(e.data);
  })
```

ファイル worker.js

```
onmessage = e => {
  let n = BigInt(e.data);
  let sep = "";
  for (let p = 2n; ; p++) {
    if (p * p > n) {
      if (n > 1) {
        postMessage(sep);
        postMessage(n);
      }
      postMessage("\n");
      return;
    }
    while (n % p == 0) {
      postMessage(sep);
      postMessage(p);
      sep = " * ";
      n /= p;
    }
  }
}
```

関連して ServiceWorker という、特定のオリジンとパスに関連付けられた Worker がある。例えば、オフラインのときにも Web ページが最低限の動作をできるように使われる。

その他

スマートフォンやタブレットなどのモバイルデバイス向けのページを作るときは DeviceMotion, DeviceOrientation, Touch, Drag and Drop, などが必要になるかもしれない。 MDN の Web API のページにウェブブラウザーから利用できる API の一覧がある。

2.3 JavaScript 上のライブラリー

Web アプリケーションを作成しやすくために、多くの JavaScript ライブラリーが作成されている。以下では、その中のいくつかをごく簡単に紹介する。

Vue.js

Web アプリケーションのユーザーインターフェイスの構築を対象とするライブラリーである。

React

React もユーザーインターフェイス構築のためのライブラリーである。 という JavaScript の構文の拡張を使用する。

Electron

Node.js などに基いて、Web アプリケーションと同じ技術を使って、 を構築するためのフレームワークである。Visual Studio Code も Electron フレームワークを使っている。

その他

Angular, Next.js, Svelte, Nuxt.js, Aurelia, Ember.js などのライブラリーあるいはフレームワークが、クライアント側のプログラムで広く使われているようである。このようなライブラリーやフレームワークはプログラミング言語自体と比べると、流行り廃りが激しい。

2.4 JavaScript のツール

Node.js

主にサーバー側でネットワークアプリケーションを構築するために使われる JavaScript 実行環境である。

Deno

Node.js と同じく、主にサーバー側で使われる JavaScript 実行環境である。Node.js の作者自身が Node.js の反省に基づき設計した。

npm

npm は Node.js のパッケージ管理システムである。 JavaScript のパッケージは、いくつかの JavaScript ソースファイル（ES6 の場合はモジュール）とその他のリソースから構成され、パッケージ間の依存関係が記述されたものである。また npm パッケージを集めたレポジトリーが運営されている。もともと npm という名前は Node Package Manager の頭文字に由来していた。つまり、Node.js 用のパッケージ管理システムであった。しかし、Node.js 以外の（クライアント側の）JavaScript のパッケージ管理にも使用されるようになってきている。特に ECMAScript 6 のモジュールを使う場合は、その必要性が高い。その場合、webpack や Vite などのバンドラーと呼ばれるツールと一緒に使われるのが

普通である。モジュールを使っていると JavaScript が小さなソースファイルに細分化されて、そのままでは読み込みに時間がかかるようになるため、実運用時に一つの大きな JavaScript ファイルにまとめて（バンドルして）配布したいからである。（開発時には、頻繁に変更される部分をいちいちバンドルしないようになっている。）

JavaScript 用のバンドラーとしては、 webpack, Browserify, Rollup, Vite, esbuild, Turbopack などが知られている。

2.5 AltJS

Alternative JavaScript のことで、JavaScript の代替となるプログラミング言語のことである。ほとんどが JavaScript にコンパイルされて実行される。ここでは主な AltJS を紹介する。

TypeScript

マイクロソフト社によってオープンソースプロジェクトとして開発されているプログラミング言語である。JavaScript にコンパイル時（実行前）の
を導入している。

ClojureScript

の方言である Clojure を JavaScript にコンパイルして実行する処理系である。もともと Clojure は Java 仮想機械 (JVM) 上で動作する、並行プログラミングのために設計されたプログラミング言語である。（なお Closure と混同しないこと。 Google Closure Tools は JavaScript 用のツールキットである。）

Kotlin

2011年、ジェットブレインズ (JetBrains) 社の Andrey Breslav, Dmitry Jemerov らによる。 Kotlin は、もともと JVM で動作するように設計され、 など現代的な特徴を取り入れ、“より良い” Java を目指している言語である。JavaScript にコンパイルすることも可能になっている。

WebAssembly

と省略されることがある。 WebAssembly は JavaScript と違って、
形式でブラウザー上で実行される。C や Rust のような高級言語で書かれたプログラムをコンパイルすることを想定している。

Emscripten

C, C++ などを WebAssembly にコンパイルするためのツール群である。

Pyodide

WebAssembly 上の Python の処理系である。 HTML に直接 Python のソースを記述できるフレームワークの PyScript も Pyodide を使用している。