本書は、小社刊の以下の刊行物をもとに、大幅に加筆と修正を行い書籍化したものです。

・『WEB+DB PRESS』Vol.120 特集「[自作 OS ×自作ブラウザで学ぶ] Web ページが表示されるまで―― HTML を運ぶプロトコルとシステムコールの裏側」

はじめに

本書では、自分の手を動かして簡単なブラウザを作ってみることにより、ブラウザの裏側で何が起きているかを学びます。普段からブラウザを使用していて、その裏側に少しでも興味を持った方を対象としています。また、Web アプリケーションを開発している方も、本書によって Web の知識を深めて、普段書いているコードをさらに深く理解する手助けにもなれば幸いです。

また、本書のもう一つの特徴として、今回開発するブラウザは、関連書『[作って学ぶ] OS のしくみ』で解説されている簡単な OS の上で動くようになっています。つまり、本書と合わせて『[作って学ぶ] OS のしくみ』も読むことによって、ブラウザと OS のどちらも作ることができます。それにより、ユーザーとのやりとりを行うユーザーインタフェースから、HTML、CSS、JavaScript の言語の解釈、ネットワークの裏側、そして、コンピュータの根幹のしくみまで知ることも可能です。本書だけでブラウザは完成するので、ブラウザのみに興味がある方は本書だけを読み進めることもできます。

ブラウザは、開発者にとってもユーザーにとっても、もはや日常の一部と言えるほど身近なソフトウェアです。しかし近年のブラウザはあまりにも高機能かつ巨大になってしまったため、そのしくみを詳しく理解することは難しくなっています。

たとえば 2024 年 7 月時点で、Chromium ブラウザのソースコードは約 3,270 万行[注1]もあります。また、ほかのオープンソースプロジェクトである Firefox のソースコードは約 3,100 万行[注2]あると言われています。この規模のソースコードをすべて読んで、しくみを理解するのは容易ではありません。

本書は、もはやブラックボックスと化してしまったブラウザのコアの機能を自分の手を動かして実装することで、ブラウザの裏側を少しでも理解することを目的としています。コアの機能とは、具体的にはサーバとやりとりをして目的の Web サイトを表示することです。本書で実装するブラウザは、ユーザーからのインプットである URL を解釈し、サーバと HTTP リクエスト／レスポンスをやりとりし、サーバから返ってきた HTML や CSS や JavaScript などのリソースをユーザーに見やすく表示します。

本書によって、複雑なソフトウェアであるブラウザに対して少しでも「わかった」という気持ちを抱いていただけることを願っています。

注 1　https://openhub.net/p/chrome/analyses/latest/languages_summary
注 2　https://openhub.net/p/firefox/analyses/latest/languages_summary

本書を読む前の準備

■環境構築

本書のサンプルプログラムは macOS と Ubuntu でテストされており、macOS や Ubuntu や Debian GNU/Linux などの Linux ディストリビューション上で開発することを想定しています。Windows では、WSL（*Windows Subsystem for Linux*）などを使用して Windows 上で仮想的に Linux 環境を作ることで対応が可能です。

● Rust のインストール

本書のサンプルプログラムは、プログラミング言語の一つである Rust[注1] で書かれています。Rust は、複数のツールを使ってプログラムの管理をします。プログラムをコンパイルしたり実行したりするために必要な一連のツール群をツールチェインと呼びます。

ツールチェインをインストールするために、ターミナルを開いて以下のコマンドを実行してください。このコマンドは公式ページ[注2] に記載されているものと同等です。

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

ツールチェインをインストールすると、以下のコマンドが使用できるようになっているはずです。

- rustup
  Rust のツールチェイン管理ツール。Rust のツールチェインのインストール、アップデート、管理に使用する
- rustc
  Rust のコンパイラ。Rust のソースコードをバイナリコードに変換する
- cargo
  Rust のビルドツール。Rust のプロジェクトをビルド、テスト、デプロイするために使用する

● 本書で使用している Rust のバージョン

本書のサンプルプログラムは、関連書『[作って学ぶ] OS のしくみ』で説明されているゼロから作成した WasabiOS[注3] という自作 OS のアプリケーションとして動作します。その特性上、nightly というバージョンのツールチェインを使用する必要があります。Rust は、nightly、beta、stable という 3 段階のリリースサイクルを経てユーザーのもとに届きます。デフォルトで使用されているのが stable バージョンで、その名のとおり、一番安定した機能を含んでいます。対して nightly は、最も実験的でかつ最新の機能を含むバージョンです。nightly は毎日リリースされるのに対し、stable は 6 週間ごとにリリースされます。本書執筆時点（2024 年 7 月）で、OS 開発のためは nightly でしか採用されていない機能を使う必要があるので、OS で使用しているツールチェインに合わせて本書でも nightly を使用しています。

注 1　https://www.rust-lang.org/
注 2　https://rustup.rs/
注 3　https://github.com/hikalium/wasabi

ツールチェインのバージョンを指定できる rust-toolchain.toml をプロジェクトのトップディレクトリに追加しましょう。

rust-toolchain.toml

```
[toolchain]
channel = "nightly-2024-01-01"
components = [ "rustfmt", "rust-src" ]
targets = [ "x86_64-unknown-linux-gnu" ]
profile = "default"
```

現在インストールされている Rust コンパイラのバージョンを rustup show コマンドで確認すると、nightly-2024-01-01 の日付のものになっているはずです。

```
$ rustup show
(省略)
active toolchain
----------------

nightly-2024-01-01-aarch64-apple-darwin (overridden by '/Users/.../rust-toolchain.toml')
```

● QEMU のインストール

QEMU とは、オープンソースのエミュレータです。エミュレータとは、あるコンピュータシステムが別のコンピュータシステムの機能を模倣するソフトウェアまたはハードウェアです。普通、アプリケーションを開発しているときにエミュレータは必要ありません。しかし本書のブラウザは、WasabiOS の上で動かすためにエミュレータが必要です。

QEMU をインストールするために、Mac を使用している方は以下を実行してください。

```
$ brew install qemu
```

Debian GNU/Linux や Ubuntu を使っている方は、以下を実行してください。

```
$ apt install qemu-system
```

ほかの環境で開発している方は、公式のページ[注4]を参考にダウンロードとインストールをしてください。

● Git のインストール

Git はプログラムのバージョン管理を行うツールです。ソースコードをダウンロードするときに Git を使用するので、もし今まで使用したことがなければインストールしてください。

Git をインストールするために、Mac を使用している方は以下を実行してください。

```
$ brew install git
```

Debian GNU/Linux や Ubuntu を使っている方は、以下を実行してください。

```
$ apt install git-all
```

注 4　https://www.qemu.org/download/

ほかの環境で開発している方は、公式のページ[注5]を参考にダウンロードとインストールをしてください。

■ サンプルプログラム

本書で解説して実装するブラウザのプログラムは、2つのGitHubリポジトリに掲載されています。サンプルブラウザアプリケーション（**Sa**mple **B**rowser **A**pplication）を略してSaBAという名前です。もし本書を参考にして自分でプログラムを書いているときに思ったように動かなければ、これらのリポジトリのコードと見比べてみてください。

- github.com/d0iasm/saba
  最新の変更／修正を含むリポジトリ。本書で書かれていること以上の実装を含む
- github.com/d0iasm/sababook
  本書とまったく同じコードのリポジトリ。章ごとでディレクトリが分けられている

● SaBA ブラウザの構成

SaBAの大まかなディレクトリ構造は以下のようになっています。`build/` ディレクトリや便利スクリプトなどは省略しています。

```
saba $ tree -L 2
.
├── Cargo.toml
├── README.md
├── saba_core
│   ├── Cargo.toml
│   └── src
├── src
│   └── main.rs
├── net
│   ├── std
│   └── wasabi
└── ui
    ├── cui
    └── wasabi
```

一番のメインとなる実装は `saba_core/` ディレクトリ以下に存在します。`src/` ディレクトリは `main` 関数を含むアプリケーションのエントリポイントになります。それ以外の `net/`、`ui/` ディレクトリは、アプリケーションを動かすOSによって実装を変える必要があるため、ディレクトリが細分化されています。ただし、本書ではWasabiOS上で動かす実装のみを紹介します。

- saba_core
  HTML、CSS、JavaScriptを解釈してページをレンダリングする機能の実装。外部クレートへの依存関係を持たない。第2章、第4章、第5章、第7章で実装
- src
  アプリケーションのエントリポイントとなるメイン関数の実装。各章で少しずつ実装

注5　https://git-scm.com/downloads

- net
  ネットワークに関する機能の実装。第 3 章で実装
- ui
  ユーザーインタフェースに関する機能の実装。第 6 章で実装

WasabiOS の構成

SaBA を動かす OS は、関連書で解説されている WasabiOS[注 6] を使用します。さらに深掘りして、ネットワークの根幹や OS がどのようにリソースを管理しているかまで理解したい場合は、こちらのリポジトリと関連書『[作って学ぶ] OS のしくみ』も参考にしてください。

WasabiOS 上でアプリケーションを開発する際に特に重要なのは WasabiOS リポジトリの noli/ ディレクトリです。これは OS とアプリケーションをつなぐライブラリ群で、文字や図形の描画などの機能をアプリケーションに提供しています。

アプリケーションを WasabiOS で動かす

アプリケーションを WasabiOS で動かすためには、cargo build コマンドでビルドしたアプリケーションのバイナリを、WasabiOS が提供する run_with_app.sh というスクリプトを使用して走らせる必要があります。これらを自動的に行ってくれる便利なシェルスクリプトを用意したので、以下のスクリプトを自分のプロジェクトに追加してください。または、d0iasm/saba/run_on_wasabi.sh[注 7] からコピーすることもできます。

run_on_wasabi.sh

```
#!/bin/bash -xe

HOME_PATH=$PWD
TARGET_PATH=$PWD"/build"
OS_PATH=$TARGET_PATH"/wasabi"
# アプリケーションの名前が saba とは異なるとき、次の行を変更する
APP_NAME="saba"
MAKEFILE_PATH=$HOME_PATH"/Makefile"

# build ディレクトリを作成する
if [ -d $TARGET_PATH ]
then
  echo $TARGET_PATH" exists"
else
  echo $TARGET_PATH" doesn't exist"
  mkdir $TARGET_PATH
fi

# WasabiOS をダウンロードする（https://github.com/hikalium/wasabi）
# もしスクリプトが失敗する場合は、`rm -rf build/wasabi` などで
# ダウンロードした OS を削除する必要がある
if [ -d $OS_PATH ]
then
  echo $OS_PATH" exists"
  echo "pulling new changes..."
  cd $OS_PATH
```

注 6　https://github.com/hikalium/wasabi
注 7　https://github.com/d0iasm/saba/blob/main/run_on_wasabi.sh

```
  git pull origin for_saba
else
  echo $OS_PATH" doesn't exist"
  echo "cloning wasabi project..."
  cd $TARGET_PATH
  git clone --branch for_saba git@github.com:hikalium/wasabi.git
fi

# アプリケーションのトップディレクトリに移動する
cd $HOME_PATH

# Makefile をダウンロードする
if [ ! -f $MAKEFILE_PATH ]; then
  echo "downloading Makefile..."
  wget https://raw.githubusercontent.com/hikalium/wasabi/main/external_app_template/
Makefile
fi

make build
$OS_PATH/scripts/run_with_app.sh ./target/x86_64-unknown-none/release/$APP_NAME
```

シェルスクリプトをトップディレクトリに追加したら、chmod コマンドを使用してシェルスクリプトに実行権限を与えましょう。

```
$ chmod +x run_on_wasabi.sh
```

アプリケーションのトップディレクトリで run_on_wasabi.sh のスクリプトを走らせると、アプリケーションが WasabiOS の上で開始します。もしアプリケーションの名前を独自のものにした場合は、スクリプトの APP_NAME を変更してください。

また、もしスクリプトが途中で失敗したら、rm -rf build などによりダウンロードした WasabiOS のソースコードを削除してみてください。

● プロジェクトの作成

プロジェクトを作成してみましょう。cargo コマンドを使用することによって簡単に新しいプロジェクトを作成できます。cargo new コマンドとそれに続いてプロジェクト名を入力することで新しいディレクトリを作成します。ディレクトリの配下には Cargo.toml と src ディレクトリが自動的に作成されます。

```
$ cargo new saba
```

Cargo.toml はプロジェクトの設定を管理するための設定ファイルです。ライブラリの依存関係などをここに書きます。WasabiOS とやりとりするための noli ライブラリを使えるように Cargo.toml を書き換えてみましょう。

Cargo.toml

```
[package]
name = "saba"
version = "0.1.0"
edition = "2021"

[dependencies]
noli = { git = "https://github.com/hikalium/wasabi.git", branch = "for_saba" }
```

src ディレクトリ以下の main.rs を変更してみましょう。WasabiOS は、スタンダードライブラリに依存せずに書かれています。スタンダードライブラリとは、Rust では std によってインポートできるライブラリ群のことです。OS の制約上、アプリケーションも同じくスタンダードライブラリに依存せずに書く必要があります。よって、ファイルの最初には #![no_std] と書いてください。

noli ライブラリの API を使用するために、use noli::prelude::*; も必要です。これで、文字を出力したり図形を描画したりできます。

src/main.rs

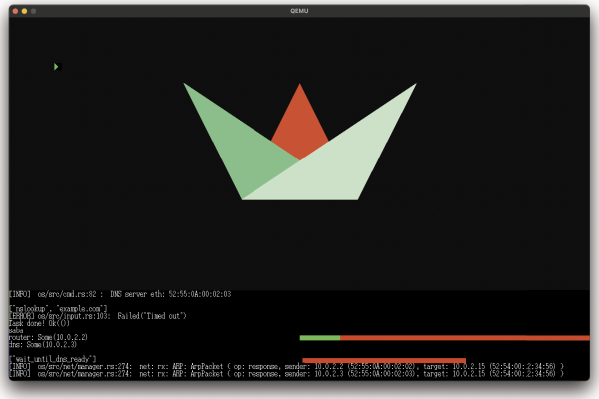
```
#![no_std]
#![cfg_attr(not(target_os = "linux"), no_main)]

use noli::prelude::*;

fn main() {
    Api::write_string("Hello World\n");
    println!("Hello from println!");
    Api::exit(42);
}

entry_point!(main);
```

run_on_wasabi.sh スクリプトを使用して OS 上でアプリケーションを動かしてみましょう。

```
$ ./run_on_wasabi.sh
```

スクリプトを走らせると、QEMU のアプリケーションが開始します（**図 0-1**）。その画面上またはターミナル上でアプリケーションの名前（saba）を入力して Enter キーを押すと、そのアプリケーションが開始します。

図 0-1　QEMU のスタート直後の画面



WasabiOS は、QEMU の画面の下部にログが出力します。上記の "Hello World!" の文字列を出力するアプリケーションを開始すると、QEMU とターミナル上のどちらでもログの出力が確認できます（**図 0-2**）。

図 0-2　ログの出力結果

● 本書のコードの読み方

第 2 章から実装していくブラウザのコードは、本書で以下のように書かれています。

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Token {
    (省略)
    /// https://262.ecma-international.org/#sec-identifier-names
    Identifier(String),
    /// https://262.ecma-international.org/#sec-keywords-and-reserved-words
    Keyword(String),
    /// https://262.ecma-international.org/#sec-literals-string-literals
    StringLiteral(String),
}
```

新しく実装する箇所は太字で書かれているので、書籍を読みながら実装していく方は太字の箇所を自分のプログラムに随時追加してください。もしコード中に太字がまったくない場合は、すべてのコード追加する必要があるという意味です。また、「(省略)」と書かれている部分はすでに実装を紹介した箇所です。

コード中に時折出てくる URL は、その実装に対応する仕様書への URL を表します。もし仕様書ではどのように書かれているのか気になる方は自分で確かめてみてください。

● 注意事項

本書で解説・実装をするブラウザのアプリケーションは自作 OS 上で動いているため、さまざまな制約があります。たとえば、アプリケーションが使用できるメモリには限りがあります。なので、もしページ遷移を繰り返すと、アウトオブメモリ、つまり必要なメモリ容量をこれ以上確保できず実行が中断してしまうなどの問題があります。

また､OS にはアプリケーションを中断する機能がありません。もしブラウザのアプリケーションを終了したいときは、QEMU のアプリケーション自体を閉じて、OS の実行自体を終了させてください。

さらに、アクセスできる Web サイトは HTTP から始まるページのみです。通信が暗号化されている HTTPS から始まるページにはアクセスできないことに注意してください。

ブラウザも OS もとても巨大なプログラムで、かつ、さまざまな使い方が存在します。本書で明示的に解説されている使い方以外は、バグを含んでいる可能性が大いにあることにご注意ください。もし明らかなバグを見つけた場合は、saba リポジトリの issue に報告していただけるとうれしいです。

目 次

第1章

ブラウザを知る

Web サイトを表示するアプリケーション

WWW（World Wide Web）、通称 Web が 1989 年に欧州原子核研究機構（CERN）で開発されて以来、インターネットを通じて情報を得ることは私たちの日常から切り離せないものになりました。Web ブラウザ、または単にブラウザは、私たちがインターネットとやりとりするときの窓口の役割を担ってくれます。世界中のさまざまな情報にアクセスするために、ブラウザは時間とともに機能をどんどん増やし、今では一人の人間がブラウザを完全に理解するのは困難なくらい大規模なプログラムになってしまいました。はたしてブラウザとは、いったい全体何なのでしょうか？

筆者は、ブラウザにはコアとなる役割が 3 つあると考えています。

❶インターネットを通してサーバと通信するクライアント
❷ HTML と CSS を描画するレンダリングエンジン
❸ JavaScript を実行する JavaScript エンジン

そして、そのコアとなる役割を支えるために、ブラウザはストレージとしてデータを保存したり、ユーザーの使い勝手が良くなるような UI も提供したりします。すべての機能においてセキュリティを担保することも忘れてはいけません。

さらに、現代のブラウザでは、これらを実装する設計アプローチとして、マルチプロセスアーキテクチャを採用しています。

本章では、コアとなる 3 つの機能とそれらを支えるさまざまな機能、それらを実装するマルチプロセスアーキテクチャ、そしてセキュリティ機構について解説します。

ブラウザの役割❶ ── Web クライアントとしてのブラウザ

私たちが情報をブラウザで閲覧するために関わってくる要素は、大きく分けて以下の 3 つあります。

- Web クライアント
- Web サーバ
- インターネット

Web クライアントは、ユーザーと実際にやりとりするソフトウェアで、本書では Web クライアントとブラウザを同等のものとして扱います。ブラウザ以外

の Web クライアントとしては cURL というターミナルのコマンドラインから使用できるソフトウェアなどがあります。

Web サーバは、情報を格納しているソフトウェア、ハードウェアです。私たちがブラウザを使用して見ている情報は Web サーバ内に存在します。情報とは、たとえば、HTML、CSS、JavaScript などのファイルとそれらに存在する文字列を指します。それぞれの役割については本章の「Web サイトの構成」で後述します。

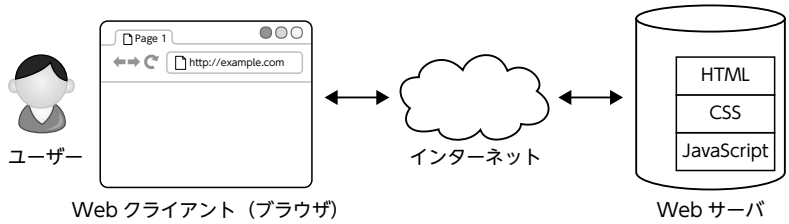
ブラウザのコアとなる役割の 1 つ目が Web クライアントです。ブラウザは、どのページの情報が欲しいかのリクエストを Web サーバに送って、その返事をもらうことで情報を得ます。そのときの通り道がインターネットです。

クライアント／サーバモデル

情報を提供するサーバと、インターネットを通じてそれにアクセスして利用するクライアントから成り立つソフトウェアの構成をクライアント／サーバモデルと言います。サーバとクライアントは異なる目的を持って運用されていることが特徴で、サーバが中心的な役割を担っていて負荷が高いのに対し、クライアントは利用者がサーバにアクセスするための限定的な役割のみを行うため、クライアントの負荷は低いことが多いです。

Web クライアントと Web サーバは、クライアント／サーバモデルの Web に特化したシステムになります（**図 1-1**）。

図 1-1　クライアント／サーバモデル



クライアント／サーバモデルではないソフトウェアの構成としては、ピア・ツー・ピアモデルなどがあります。これはすべてのコンピュータが同等の機能を有しており、中心的な存在はありません。

■Web クライアント

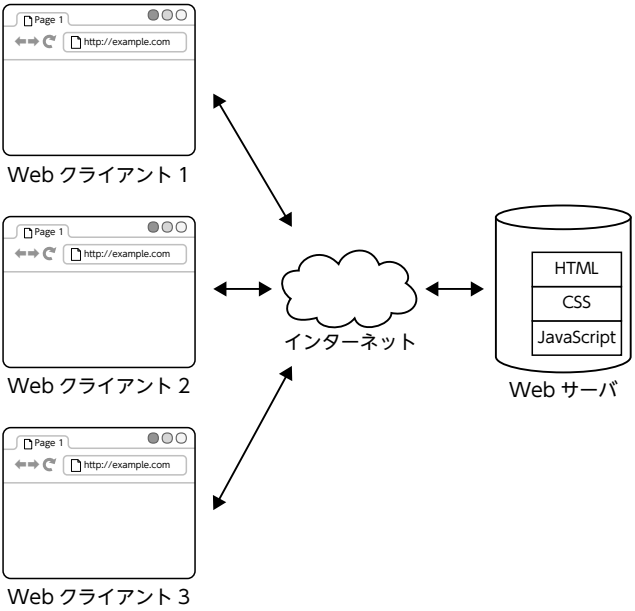
Web クライアントとは、Web サーバに対して、どんな情報が欲しいのかを記述した HTTP を送信し、その返事として受信したリソースを解釈してユーザーに見やすく表示したりするソフトウェアのことを指します。厳密に言えばブラウザ以外の Web クライアントも存在するので、Web クライアントとブラウザは同じ意味を表す用語ではないですが、前述したように、本書では Web クライアントとはブラウザのことを指すことにします。

■Web サーバ

Web サーバとは、Web クライアントに対して、HTML や画像などを提供するソフトウェア、およびそのソフトウェアが動くハードウェアのことを指します。本書では、ソフトウェアとしての Web サーバのみを取り扱います。

Web サーバが動くハードウェアは、基本的には高性能なことが多いです。なぜならクライアント／サーバモデルでは、クライアントが多数いるのに対し、サーバ側は圧倒的に少ない数で対応していることが多いからです（**図 1-2**）。

図 1-2　複数のクライアントとサーバの例



たくさんの Web クライアントから接続があったときに、より効率良く情報を提供できるようにするさまざまな手法が存在します。

たとえば、複数のサーバに負荷を均等に分散するロードバランサというしくみが存在します。これによりサーバへの負荷を分散できます。また、障害時には自動的にトラフィックを別のサーバに転送することも可能です。

また、CDN（*Contents Delivery Network*）は、複数の地理的に分散されたサーバを使用してコンテンツを配信するネットワークです。CDN は、画像、HTML、CSS、JavaScript ファイルなどのコンテンツをキャッシュし、クライアントからのリクエストに対して最も近いサーバからコンテンツを提供することで、レスポンス時間を短縮します。

ほかにも手法はあるのですが、本書ではこれ以上のサーバ側の解説は行いません。

インターネットと Web

上記で紹介した Web クライアントと Web サーバの情報のやりとりを支えるのがインターネットです。本書ではこれまで Web（WWW）とインターネットを同様のものとして書いていましたが、本来は少し異なるものです。

インターネットは、コンピュータがつながり相互に通信しているネットワークのことを指します。TCP/IP と呼ばれる通信プロトコルを用いて、さまざまな機器やサービスが相互接続されています。インターネット上では、メールの送受信、ファイルの共有、オンラインゲームや動画配信など、さまざまな活動が行われています。

対して Web は、インターネット上で提供されている情報共有システムの名前です。HTML で記述されるハイパーテキストと呼ばれる文書形式とハイパーリンクと呼ばれるしくみを用いて Web ページ間をつなぎ、ユーザーがさまざまな情報に簡単にアクセスできるようにしています。Web サイトを閲覧するときにはインターネット上の Web のシステムを利用していることになります。技術的には、HTTP というプロトコルを用いてインターネット上でやりとりするシステムのみを Web と呼びます。最近では、より安全に通信を行えるように、HTTP の内容に暗号化を行って内容を秘匿することのできる HTTPS というプロトコルのほうが主流です。

インターネット上でやり取りするが Web ではないものの例として、電子メールの送受信やファイルの転送などがあります。電子メールは SMTP、POP3、IMAP などのプロトコルを用いて、ファイル転送は FTP や SFTP などのプロトコルを用いて通信を行うためです。

■通信プロトコル

インターネット上の通信はプロトコルに従って行われます。プロトコルとは人間でいう言語のようなもので、やりとりをする際の手順や規則を定めている規格です。プロトコルは複数の層から成り立つ多層構造になっており、それぞれの層で異なる役割を持ちます。この多層構造を表現するモデルが TCP/IP モデルや OSI 参照モデルです。

TCP/IP は、IETF[注1] という組織によって策定され、プロトコルを 4 層に分割しています。物理的な現象から遠い層、つまり、ユーザーに近い層から、以下のように分けられています。

❶アプリケーション層
❷トランスポート層
❸インターネット層
❹リンク層

OSI 参照モデルは、国際標準化機構（ISO）[注2] という組織によって策定され、プロトコルを 7 層に分割しています。物理的な現象から遠い層、つまり、ユーザーに近い層から、以下のように分けられています。

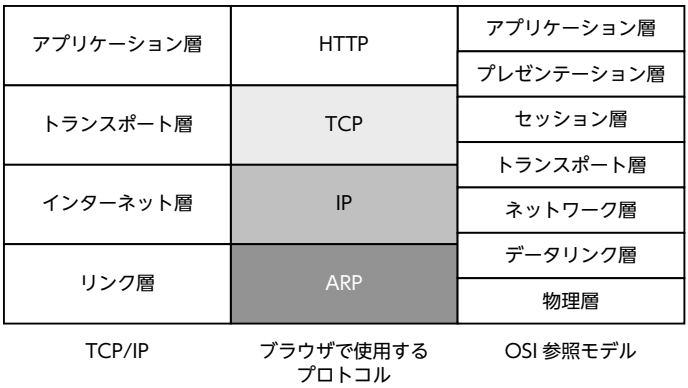
❶アプリケーション層
❷プレゼンテーション層
❸セッション層
❹トランスポート層
❺ネットワーク層
❻データリンク層
❼物理層

注 1　https://www.ietf.org/
注 2　https://www.iso.org/home.html

本書では TCP/IP の分け方に準拠することにします。

Web サイトを表示するためには、アプリケーション層からリンク層までのすべてのプロトコルを使用します。具体的には、アプリケーション層では HTTP、トランスポート層では TCP や UDP、インターネット層では IP、リンク層では ARP を使用します（**図 1-3**）。

図 1-3　TCP/IP モデル・OSI 参照モデル



アプリケーション層より下の階層にあるプロトコルは OS によって管理されています。関連書の『[作って学ぶ] OS のしくみ』にてトランスポート層以下のプロトコルについても解説と実装をしているので、そちらを参照してください。

■HTTP

TCP/IP と OSI 参照モデルのアプリケーション層に位置する、Web 上の情報をやりとりする際に使用するプロトコルが HTTP です。ハイパーテキスト転送プロトコル（*Hypertext Transfer Protocol*）の略です。ハイパーテキストとは複数の文書を相互に関連付けるしくみで、HTML を使用して作成できます。HTTP はハイパーテキストを Web クライアントと Web サーバ間で送受信するためのプロトコルです。

プロトコルはテキストで情報が表現されているテキストベースのものと、0 と 1 の 2 進数で情報が表現されているバイナリベースの 2 種類があります。HTTP はテキストベースのプロトコルなので、人間が簡単に読むことができます。

HTTP はブラウザがどのようなリソースを取得したいのかを記述した HTTP

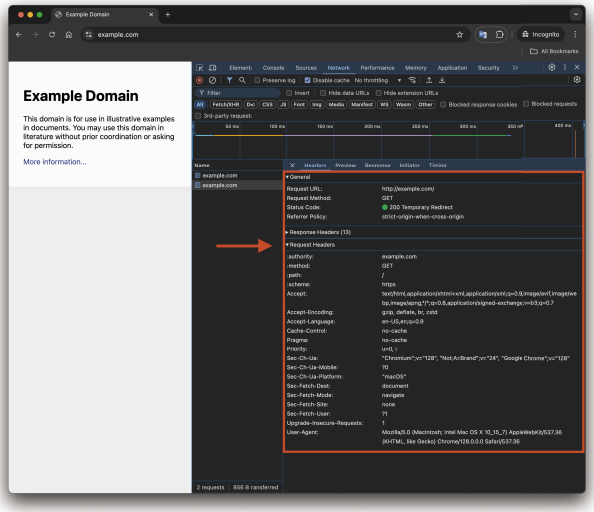
リクエストを Web サーバに送信し、Web サーバがそのリクエストに対しての返信となる HTTP レスポンスを返すことでやりとりします。

たとえば、Mac 上の Chrome ブラウザを使用して、http://example.com にアクセスしたときの HTTP リクエストは以下のようになります。

```
GET / HTTP/1.1
Host: example.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,ima
ge/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,ja-JP;q=0.8,ja;q=0.7
Cache-Control: max-age=0
Connection: keep-alive
If-Modified-Since: Thu, 17 Oct 2019 07:18:26 GMT
If-None-Match: "3147526947+gzip"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.
36 (KHTML, like Gecko) Chrome/97.0.4692.71 Safari/537.36
```

送受信する HTTP の情報はブラウザ上で確認もできます（**図 1-4**）。

図 1-4　ブラウザで確認できる HTTP リクエストのヘッダ



このHTTPリクエストに対するWebサーバからのHTTPレスポンスは以下のようになります。一部のヘッダは省略してあります。空白行のあとに、<!doctype>から始まるHTMLの情報が見えますね。

```
HTTP/1.1 200 OK
Age: 268965
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
(省略)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1256

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
(省略、以下HTMLが続く)
```

HTTPの詳しい解説と実装は第3章の「HTTPを実装する」で行います。

■URLによるリソースの指定

ブラウザでは、ユーザーが入力するURL（*Uniform Resource Locator*）をもとに、HTTPリクエストを構築します。URLはWeb上にあるリソースがどこにあるかを示す識別子で、ブラウザを使用してどのWebサイトを閲覧したいかを指定するために使用します。たいていの場合、ブラウザの上部または下部にあるツールバーでURLを入力したり、現在のURLを確認したりできます。

URLは、リソースがどこにあるかと、そのリソースにどのようにアクセスするかを示します。通常、URLにはプロトコル（`http://`や`https://`）、ホスト、パスなどが含まれます。

本章で扱う部分のURLの構文は以下のとおりです。

```
<scheme>:://<host>:<port>/<path>?<searchpart>
```

それぞれの要素は以下のような意味を持ちます。

- scheme
  プロトコルの名前。`http`、`https`など

- host
  ホスト。ネットワークに接続された機器を識別するための名前。example.com など
- port
  ポート番号。80、8888 など
- path
  サーバ内の階層化されたリソースを指定するパス。/、/index.html、/foo/bar/index.html など
- searchpart
  Web サイトに提供するキーと値のペアで表される追加の情報。a=123、a=123&b=456 など

たとえば、ユーザーが http://example.com/test.html の URL をブラウザに入力したとします。ブラウザはこの URL を以下のようにそれぞれの要素に分解します。

- scheme → http
- host → example.com
- port →該当なし
- path → /test.html
- searchpart →該当なし

まず、scheme が http であるため、HTTP で通信する準備をします。そして host の値を見て、HTTP リクエストを作成します。

URL の詳しい解説と実装は第 2 章の「URL を分解する」で行います。

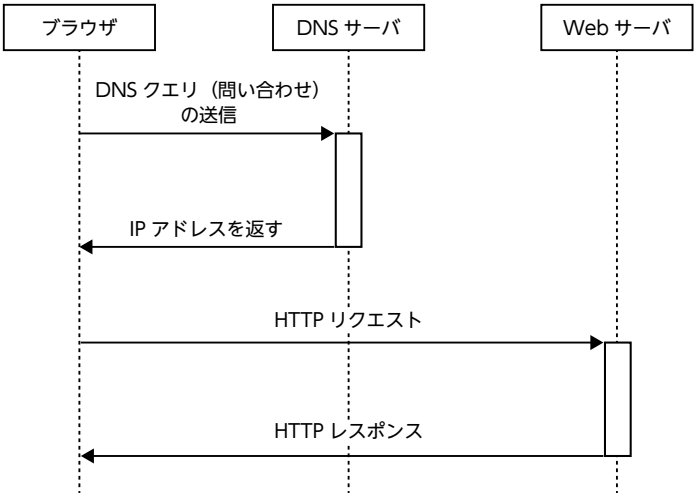
■DNS

DNS（*Domain Name System*）とは、ドメイン名と IP アドレスの対応付けを行うシステムです。DNS の情報を管理する DNS サーバはインターネット上に存在しており、ユーザーが Web サイトにアクセスする直前に使用されます。

たとえば、ユーザーが example.com のサイトにアクセスしたいとき、example.com の情報だけでは Web サーバを特定できません。Web サーバにアクセスするためには、そのサーバの IP アドレスを知る必要があります。DNS サーバは、example.com などの文字列で表されるドメイン名と IP アドレスの対応を管理しており、DNS サーバに対して問い合わせをすることで IP アドレスを知

ることができます。この IP アドレスを知る工程のことを名前解決と呼びます。そしてその IP アドレスの情報のおかげで、どこに HTTP リクエストを送信すればよいのかがわかります（**図 1-5**）。

図 1-5　ブラウザと DNS と Web サーバの関係



本書では DNS の詳しい説明と実装は行いませんが、『[作って学ぶ] OS のしくみ』で紹介を行っているのでぜひ参照してみてください。本書での実装は、OS が提供する DNS の API を使用することで名前解決を行います。

ブラウザの役割❷ ――レンダリングエンジンとしてのブラウザ

レンダリングとは、コンピュータのプログラムを用いて画像や映像などを生成することを指します。ブラウザの文脈でレンダリングという言葉を使う際は、HTML、CSS、および画像などのリソースを解釈して描画する、という意味として使われます。

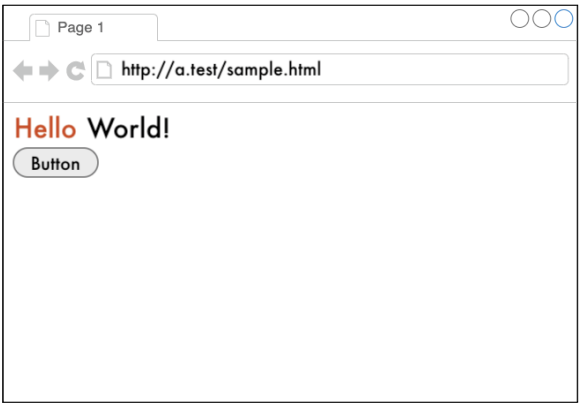
ブラウザのコアとなる役割の 2 つ目がレンダリングを行うことです。レンダリングを行うソフトウェアのことをレンダリングエンジンと呼びます。

Web サイトの構成

ブラウザのレンダリングによって描画されるものが Web サイトです。Web サイトは、Web サーバが提供する情報によって構成されています。具体的には、HTML、CSS、JavaScript と呼ばれる言語で記述されている文字列や、PNG、JPEG のような画像などです。

たとえば、**図 1-6** のページで「Hello World!」やボタンを記述しているのが HTML で、「Hello」の色を変更しているのが CSS です。そして、JavaScript はボタンをクリックするなどのユーザーの行動によって Web サイトの内容を動的に書き換えることができます。

図 1-6 Web サイトの例



HTML

HTML は Web サイトを記述するために使用されるマークアップ言語の一つです。山括弧（<>）で囲まれたタグによって成り立ち、見出しや段落といった文書の構造を表現できます。また、HTML のハイパーリンクのためのタグを使用することで、文書間を行き来することもできます。

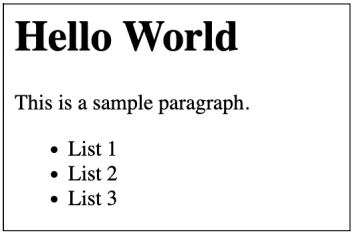
HTML が記述されたファイルの拡張子には `.html` が使用されます。たとえば、以下のような文章が HTML です。

sample.html

```
<html>
  <body>
    <h1>Hello World</h1>
    <div>
       <p>This is a sample paragraph.</p>
       <ul>
           <li>List 1</li>
           <li>List 2</li>
           <li>List 3</li>
       </ul>
   </div>
 </body>
</html>
```

この HTML は**図 1-7** のように描画されます。

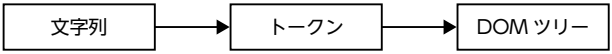
図 1-7　sample.html によるページの例



■HTML トークン

HTML は、文字列であるソースコードから、いくつかの手順を踏んでデータ構造を変化させながらブラウザによって解析されます（**図 1-8**）。

図 1-8　HTML のデータ構造の変化



まず、文字列からトークンへと分割されます。トークンとは記号・象徴を意味する一般的な英単語ですが、ここでは、HTML を細かく区切ったときに意味の

ある文字列の最小単位を表します。

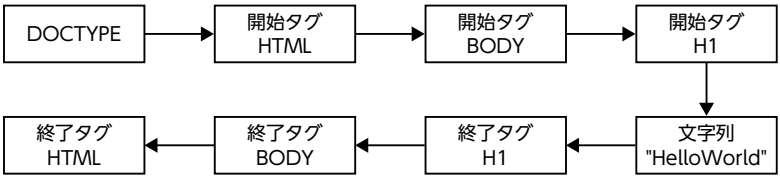
HTML のトークンには以下の 6 つの種類があります。

- DOCTYPE
  文書がどのバージョンの HTML を使用するのかを表す。<!DOCTYPE html>
- **開始タグ**
  終了タグをともに使用され、ある要素の開始を表す。 など
- **終了タグ**
  開始タグをともに使用され、ある要素の終了を表す。 など
- **コメント**
  文書の結果に影響はしないコメントを残すことができる
- **文字列**
  タグを含まない純粋な文字列
- EOF（*End Of File*）
  ファイルの終了を表す

たとえば、以下のような HTML の文字列をトークンに分割すると、**図 1-9** のようになります。

```
<html>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

図 1-9　HTML トークンの例



■DOM ツリー

トークンに分割された HTML のデータは、次に DOM ツリーと呼ばれる木構

造に変換されます。DOM とは Document Object Model の略で、HTML 文書の構造とコンテンツを表すデータ構造です。これにより JavaScript などのスクリプト言語から HTML 文書を操作／利用することが可能になります。

DOM ツリーは、ノードによって構成されています。ノードは Node インタフェース[注3]を実装するオブジェクトです。ノードの定義は DOM Living Standard の仕様書で IDL（*Interface Description Language*）によって記述されています。IDL とは、オブジェクトが持つデータや提供する関数を定義する言語です。

```
[Exposed=Window]
interface Node : EventTarget {
  const unsigned short ELEMENT_NODE = 1;
  (省略)
  readonly attribute Node? firstChild;
  readonly attribute Node? lastChild;
  readonly attribute Node? previousSibling;
  readonly attribute Node? nextSibling;
  (省略)
  [CEReactions] Node appendChild(Node node);
  [CEReactions] Node replaceChild(Node node, Node child);
  [CEReactions] Node removeChild(Node child);
}
```

インタフェースを実装したオブジェクトでは、インタフェースで定義されているデータにアクセスしたり、操作を行ったりできます。たとえば、Node インタフェースでは firstChild のフィールドが存在するので、DOM ツリーのすべてのノードは firstChild メンバを持ちます。また、appendChild という操作によって、現在のノードの配下に新しいノード（子ノード）を追加できます。

インタフェースは継承できます。たとえば、インタフェース A がインタフェース B を継承したとすると、インタフェース A は B の定義をすべて引き継ぎます。さらに、A に独自の定義を追加することも可能です。これにより IDL の定義の再利用と構造化を行うことができます。

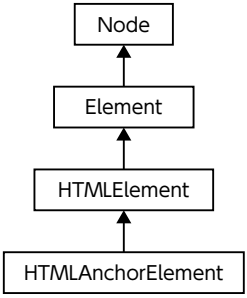
ノードの種類、つまり、Node インタフェースを継承するインタフェースは 2024 年の現時点で以下の 9 種類があります。

注 3　https://dom.spec.whatwg.org/#interface-node

- Element[注4]
- Attr[注5]
- Text[注6]
- CATASection[注7]
- ProcessingInstruction[注8]
- Comment[注9]
- Document[注10]
- DocumentType[注11]
- DocumentFragment[注12]

今回私たちが実装するブラウザで特に重要なのが Element インタフェースです。それぞれのタグに対応したインタフェースがこの Element インタフェースを継承される形で実装されます。たとえば、<a> タグで表されるほかのページへのリンクは**図 1-10** のように `Node`、`Element`、`HTMLElement` インタフェースを継承した `HTMLAnchorElement` を実装します。

図 1-10 HTMLAnchorElement

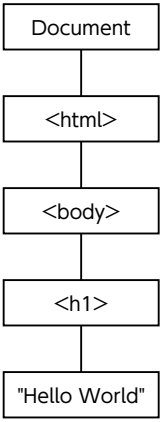


注 4 https://dom.spec.whatwg.org/#element
注 5 https://dom.spec.whatwg.org/#attr
注 6 https://dom.spec.whatwg.org/#interface-text
注 7 https://dom.spec.whatwg.org/#interface-cdatasection
注 8 https://dom.spec.whatwg.org/#interface-processinginstruction
注 9 https://dom.spec.whatwg.org/#interface-comment
注 10 https://dom.spec.whatwg.org/#document
注 11 https://dom.spec.whatwg.org/#documenttype
注 12 https://dom.spec.whatwg.org/#documentfragment

HTML トークンのときと同様の HTML ページを DOM ツリーに変換すると**図 1-11** のようになります。

```
<html>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

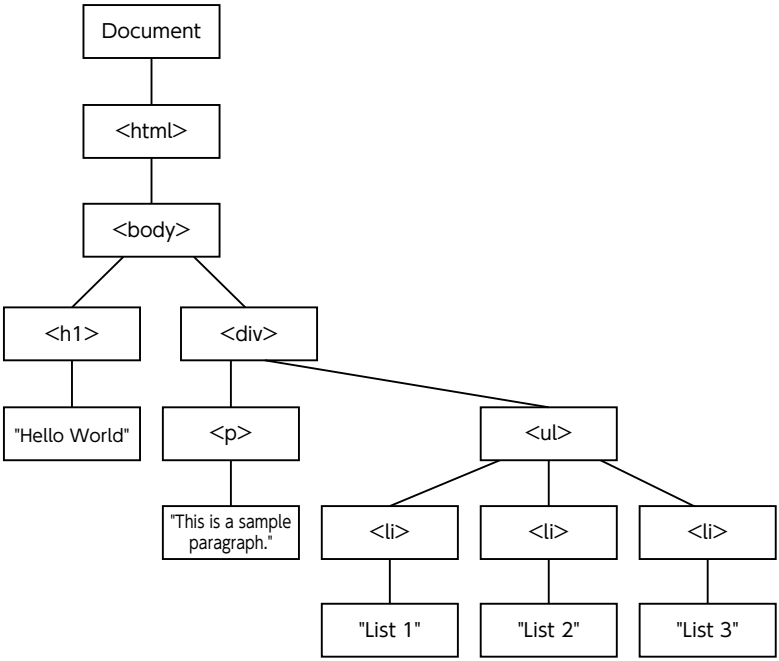
図 1-11　簡単な DOM ツリーの例



次はもう少し複雑な例です（**図 1-12**）。

```
<html>
  <body>
    <h1>Hello World</h1>
    <div>
       <p>This is a sample paragraph.</p>
       <ul>
           <li>List 1</li>
           <li>List 2</li>
           <li>List 3</li>
       </ul>
   </div>
 </body>
</html>
```

図 1-12　複雑な DOM ツリーの例



HTML の詳しい解説と実装は第 4 章の「HTML を解析する」で行います。

CSS

CSS（*Cascading Style Sheets*）は、HTML などの文書に装飾を加えるための言語です。たとえば、以下のような CSS は、HTML の段落要素に対し、文字の色を指定しています。

```
p {
  color: red;
}
```

CSS は、文字列をトークンに分割し、そのトークン列から CSSOM という木構造を作成することで解釈されます。

■CSS トークン

CSS も、HTML と同じくまずは文字列からトークンへと分割されます。HTML と同様に、CSS を細かく区切ったときに意味のある文字列の最小単位を表します。HTML との混同を避けるために明示的に CSS トークンと呼ぶことにします。

CSS Syntax Module Level 3[注13] で定義されている CSS トークンは 24 種類あります。本書ではそのうちの特に重要なものを紹介します。

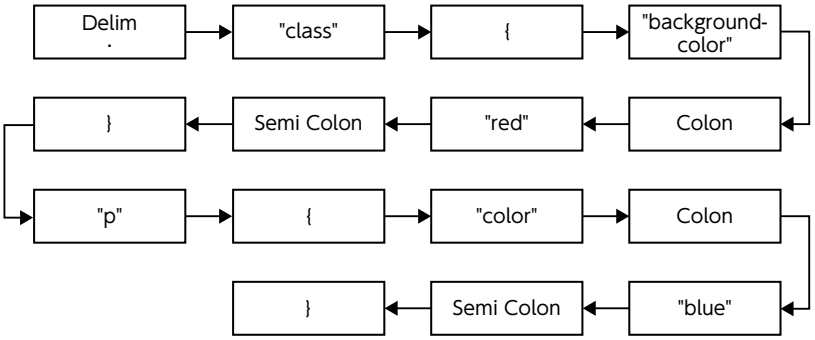
- ident-token
  変数などの識別子を表すトークン
- number-token
  数字を表すトークン
- hash-token
  ハッシュ記号を表すトークン
- delim-token
  区切り文字を表すトークン
- colon-token
  コロン（:）を表すトークン
- semicolon-token
  セミコロン（;）を表すトークン
- {-token
  始め波括弧を表すトークン
- }-token
  終わり波括弧を表すトークン

たとえば、以下の CSS をトークンに分割すると**図 1-13** のようになります。

```
.class {
  background-color: red;
}

p {
  color: blue;
}
```

注 13　https://www.w3.org/TR/css-syntax-3/#tokenization

図 1-13 CSS のトークン例



■CSSOM

CSS トークンに分割された CSS の文字列は、次に CSSOM（*CSS Object Model*）と呼ばれる木構造に変換されます。CSSOM は、ブラウザが CSS の情報を解析しツリーの形で表現したデータ構造です。CSSOM の情報に基づいて、DOM ツリーの各ノードにスタイルを適用したり、各ノードの大きさや位置を決定したりします。

CSSOM も DOM ツリーのようにノードによって構成されています。しかし、すべてのノードが `Node` インタフェースを実装していた DOM ツリーのときとは異なり、CSSOM にはさまざまの種類のノードが存在します。

たとえば、一つの CSS 文書のルートノードを表す `CSSStyleSheet` インタフェース[注14]は以下のような IDL で定義されます。`CSSRuleList` はどのセレクタに対してどのようなスタイルを適用するかを表す CSS ルールのリストで、`cssRules` によってアクセスできます。

```
[Exposed=Window]
interface CSSStyleSheet : StyleSheet {
  constructor(optional CSSStyleSheetInit options = {});

  readonly attribute CSSRule? ownerRule;
  [SameObject] readonly attribute CSSRuleList cssRules;
  unsigned long insertRule(CSSOMString rule, optional unsigned long index = 0);
  undefined deleteRule(unsigned long index);
```

注 14　https://www.w3.org/TR/cssom-1/#the-cssstylesheet-interface

```
  Promise<CSSStyleSheet> replace(USVString text);
  undefined replaceSync(USVString text);
};
```

また、一つの CSS ルールのノードを表す CSSStyleRule インタフェース[注15]では、セレクタを表す selectorText と各プロパティに対する値を保持する style を持ちます。

```
[Exposed=Window]
interface CSSStyleRule : CSSRule {
  attribute CSSOMString selectorText;
  [SameObject, PutForwards=cssText] readonly attribute CSSStyleDeclaration style;
};
```

また先ほどと同じ CSS の例を見てみましょう。この CSS には 2 つのルールが存在し、各ルールは .class セレクタと p セレクタを持ちます。これを CSSOM で表すと**図 1-14** のようになります。

```
.class {
  background-color: red;
}

p {
  color: blue;
}
```

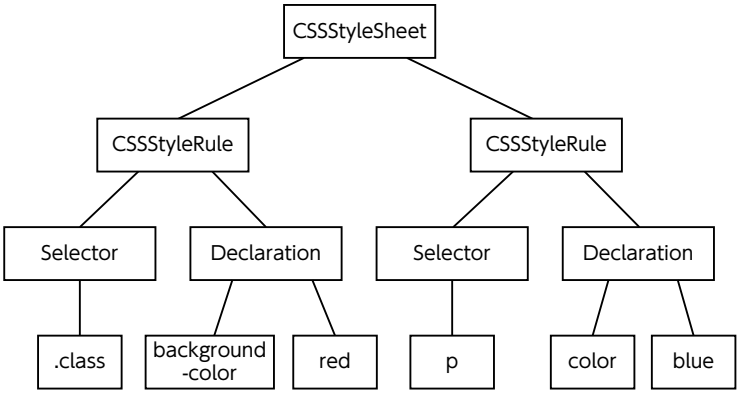
■ レイアウトツリー／レンダーツリー

レイアウトツリーまたはレンダーツリーとは、ブラウザが Web ページを表示するために、HTML と CSS をもとに作成する内部的なデータ構造です。名称はブラウザの実装によって異なりますが、本書ではレイアウトツリーの表記を使用します。

レイアウトツリーは、DOM ツリーと CSSOM の情報を使用して作成されます。レイアウトツリーは、各要素が画面上のどこに配置されるか、各要素の幅や高さ、どの要素がほかの要素の子要素であるか、兄弟要素であるかなどを決定します。

注 15　https://www.w3.org/TR/cssom-1/#the-cssstylerule-interface

図 1-14 CSSOM の例

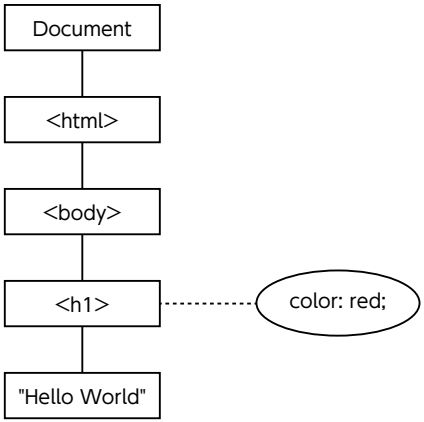


たとえば、以下のような CSS を含む HTML だと、**図 1-15** のようなレイアウトツリーになります。レイアウトツリーは描画のためのデータ構造のため、`display: none;` で指定された描画されない要素は含まないことに注意してください。

```
<html>
  <style>
    h1 {
      color: red;
    }
    .foo {
      display: none;
    }
  </style>
  <body>
    <h1>Hello World</h1>
    <p class="foo">This is sample text.</p>
  </body>
</html>
```

CSS とレイアウトツリーの詳しい解説と実装は第 5 章の「CSS で装飾する」で行います。

図 1-15 レイアウトツリーの例



ブラウザの役割❸ ── JavaScript エンジンとしてのブラウザ

JavaScript は、ブラウザ上で動かすことができるプログラミング言語です。ブラウザ以外でも動かすことができるのですが、ブラウザは JavaScript を動かすための主要な実行環境（ランタイム）の一つです。

ブラウザのコアとなる最後の役割が JavaScript の解釈と実行を行うことです。JavaScript の解釈と実行を行うソフトウェアのことを JavaScript エンジンと呼びます。

JavaScript

JavaScript は、一般的なプログラミング言語と同じく、四則演算、変数の定義、ループ文、if 文などの実行の制御、関数の定義と呼び出しなどが可能です。たとえば、変数を定義するのは以下のように書くことができます。

```
var a = "foo";
```

JavaScript の仕様書である ECMAScript 2024 Language Specification で

は、これは VariableStatement[注16] と定義されており、日本語では変数宣言文と呼びます。var というキーワードで始まり、変数宣言のリストがあり、セミコロン（;）で終了します。

```
VariableStatement : var VariableDeclarationList ;
```

JavaScript も、HTML や CSS と同じく、文字列をトークン列に分割し、抽象構文木（AST）と呼ばれる木構造を作成することによって解釈します。

■ JavaScript トークン

JavaScript でも、HTML や CSS と同じく、まず文字列からトークンへと分割します。HTML や CSS と同様に、トークンは、JavaScript を細かく区切ったときに意味のある文字列の最小単位を表します。HTML や CSS との混同を避けるために JavaScript トークンと明示的に呼ぶことにします。トークンについて説明するのはもう 3 回目なので、慣れてきましたね。

2024 年の現時点で最新の JavaScript の仕様書である ECMAScript 2024 Language Specification では、12 ECMAScript Language: Lexical Grammar[注17] でさまざまなトークンを定義しています。本書ではそのうちの特に重要なものを紹介します。

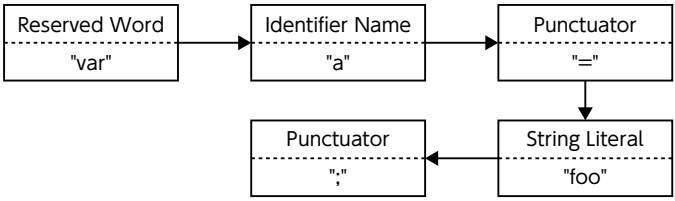
- IdentifierName
  変数などの識別子を表すトークン
- ReservedWord
  await、var、const などの予約語を表すトークン
- Punctuator
  記号を表すトークン
- NumericLiteral
  数字を表すトークン
- StringLiteral
  文字列を表すトークン

注 16　https://262.ecma-international.org/#prod-VariableStatement
注 17　https://262.ecma-international.org/#sec-ecmascript-language-lexical-grammar

以下のような JavaScript のプログラムを JavaScript トークンに分割すると**図 1-16** のようになります。

```
var a = "foo";
```

図 1-16 JavaScript トークンの例



■抽象構文木（AST）

JavaScript トークンに分割された JavaScript のデータは、次に AST（*Abstract Syntax Tree*、抽象構文木）と呼ばれる木構造の形に変換されます。AST はプログラミング言語において一般的に使用される用語で、JavaScript 以外でも使用されます。

JavaScript の AST も、ノードによって構成されており、CSSOM のようにさまざまの種類のノードが存在します。

たとえば、足し算または引き算のノードを表す `AdditiveExpression`[注 18] は以下のような BNF で定義されます。BNF とはバッカス・ナウア記法（*Backus-Naur Form*）の略で、プログラミング言語の構文や文法を定義するために使用されます[注 19]。`AdditiveExpression` は `MultiplicativeExpression`、または `AdditiveExpression` と `MultiplicativeExpression` をプラス記号またはマイナス記号によってつなげたものと置換できます。

```
AdditiveExpression :
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression – MultiplicativeExpression
```

注 18　https://262.ecma-international.org/#prod-AdditiveExpression
注 19　厳密には、BNF を独自に拡張した EBNF を使用しています。

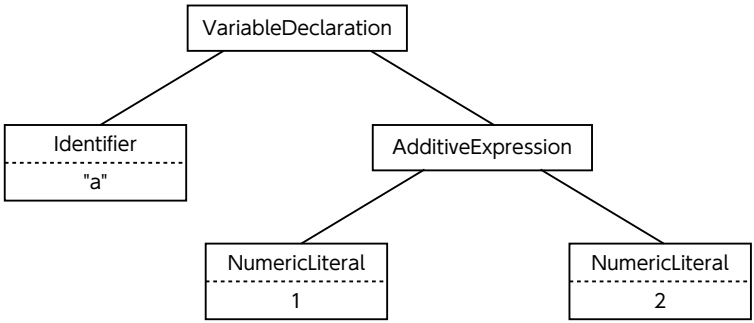
また、変数宣言のノードを表す VariableDeclaration[注20] は、変数の名前を表す BindingIdentifier または var [a, b] = [1, 2]; のようなパターンによる複数への変数を表す BindingPattern と初期化式によって構成されます。

```
VariableDeclaration :
  BindingIdentifier Initializer
  BindingPattern Initializer
```

以下のような JavaScript のプログラムの AST は、**図 1-17** のようになります。

```
var a = 1+2;
```

図 1-17 AST の例



本書の実装では、ノードの種類が簡略化されているため、仕様書の名称とは少し異なります。詳しくは第 7 章で解説します。

ブラウザ API

JavaScript がブラウザ上でしか行えないことの一つとして、HTML や CSS の操作があります。たとえば、JavaScript によって、HTML で記述されている文字列を動的に変更することが可能です。この機能は DOM を操作するための API、DOM API と呼ばれます。DOM API の仕様は、JavaScript 言語自体の

注 20　https://262.ecma-international.org/#prod-VariableDeclaration

仕様とは無関係のため、独立して存在します。

たとえば、JavaScript で特定のタグのノードを取得したいとき、以下のように書くことができます。

```
var nodes = document.getElementsByTagName("h1");
```

これは W3C が管理する DOM Living Standard の `Document` インタフェース[注21]で定義されています。タグの名前を意味する `qualifiedName` を引数にとり、戻り値としてノードのコレクションである `HTMLCollection` を返します。

```
[Exposed=Window]
interface Document : Node {
  (省略)
  HTMLCollection getElementsByTagName(DOMString qualifiedName);
  (省略)
}
```

DOM API のほかにも、ネットワークからリソースを取得する `fetch` 関数を定義する Fetch API など、さまざまな API がブラウザ内には存在します。これらの API を総称してブラウザ API と呼びます。

JavaScript の詳しい解説と実装は第７章の「JavaScript を動かす」で行います。

コアの役割を支えるためのさらなる機能

現代のブラウザには、今まで紹介した本来ブラウザが担うコアの役割に加え、ユーザーが使いやすくなるような機能がたくさん含まれています。すべてを紹介することは難しいので、そのうちのいくつかを紹介します。これらの機能は本書で実装するブラウザではサポートしません。

ストレージとキャッシュ

ブラウザは、ストレージやキャッシュとしての機能も持ちます。ユーザーのデー

注 21　https://dom.spec.whatwg.org/#interface-document

タを保存したり、Web サイトの読み込み速度を速くしたりするためなどに使用されます。

ストレージとキャッシュはどちらもデータを保存する機能を持ちますが、それぞれ異なる目的で使用されます。ストレージは、Web アプリケーションが明示的に管理する必要があり、長期的にデータを保存することが多いです。キャッシュは HTTP ヘッダによって制御され、ブラウザが自動的に管理し、期限が切れると削除されます。

■ストレージ

Cookie はユーザーの認証情報やサイトの設定などを保存するために使用されます。ブラウザが保存できる Cookie の数とサイズにはある程度の制限がありますが、RFC 6265[注22] では、ブラウザが提供するべき Cookie の数とサイズに関する最低限の要件が書かれています。ブラウザは、1 つのドメインで少なくとも 50 個の Cookie を保存でき、少なくとも合計で 3,000 個までの Cookie を保存できるようにするべきです。また、各 Cookie は少なくとも 4KB（4,096 バイト）以上保存できるべきです。実際の制約はブラウザの実装によって異なります。Cookie は、サーバとクライアント間で自動的に送受信され、主にセッション管理、ユーザーの設定の保持、トラッキングに使用されます。

ローカルストレージ（Local Storage）は、ブラウザに永続的にキー／バリュー形式でデータを保存し、ブラウザを閉じてもデータを保持し続けます。データは、同一オリジン（プロトコル、ホスト、ポートが同じ）のすべてのページで共有されます。オリジンについては後述します。Local Storage は Web Storage API[注23] の一部として定義されています。

セッションストレージ（Session Storage）は Local Storage と同じく、Web Storage API[注24] の一部として定義されているストレージです。Session Storage も Local Storage と同じくキー／バリュー形式でデータを保存します。ただ、Local Storage と異なり、ウィンドウやタブが閉じられるとデータが削除

注 22 https://datatracker.ietf.org/doc/html/rfc6265#section-6.1
注 23 https://html.spec.whatwg.org/multipage/webstorage.html#the-localstorage-attribute
注 24 https://html.spec.whatwg.org/multipage/webstorage.html#the-sessionstorage-attribute

されます。同一オリジン内の同一タブのすべてのページで共有されますが、異なるタブやウィンドウでは共有されません。

■ キャッシュ

ブラウザのキャッシュは、HTML、CSS、JavaScript、画像、動画などのリソースをキャッシュします。リソースが再度リクエストされた際、キャッシュに保存されている場合はブラウザから提供され、ネットワーク経由のダウンロードが省略されます。これにより、Web サイトの読み込み速度が向上します。HTTP ヘッダ（Cache-Control、Expires、ETag、Last-Modified など）によってキャッシュの制御が可能です。キャッシュに保存されるデータは有効期限が切れると削除されます。

拡張機能

拡張機能は誰でも開発でき、その機能を公開することでさまざまな人にブラウザに追加の機能やカスタマイズを提供できます。たとえば、Chrome ブラウザでは Chrome Web Store[注25] で、Firefox では Firefox Add-ons[注26] で、世界中のデベロッパーが開発した拡張機能を選ぶことができます。ユーザーはこれをインストールすることで、ブラウザの使い勝手を向上させたり、自分のニーズに合わせてブラウザをカスタマイズしたりできます。

PWA

HTML、CSS、JavaScript などの Web の技術を使用して、Android や iOS で提供されるモバイルアプリケーションのような体験を提供できるのが PWA（*Progressive Web App*）です。モバイルアプリのようにスマートフォンのホーム画面に追加できます。

PWA を支える重要な技術が Service Worker[注27] です。Service Worker は、JavaScript によって制御され、Web サーバへのリクエストをキャプチャし、リ

注 25　https://chromewebstore.google.com/
注 26　https://addons.mozilla.org/
注 27　https://w3c.github.io/ServiceWorker/

クエストに対する返事をキャッシュから提供するか、ネットワークから取得するかを決定できます。Service Worker によって、ネットワーク接続がない場合でもリソースをキャッシュから提供することで、オフラインの状況でもアプリを動かし、モバイルアプリケーションのような体験をユーザーに提供することが可能です。

UI にまつわる機能

ブラウザの UI（*User Interface*）にまつわる機能は列挙したらキリがないでしょう。たとえば、タブのグループ化、タブのピン留め、ブックマーク、履歴、プッシュ通知、音声検索・音声操作、シークレットモード、パスワードの管理などがあります。

マルチプロセスアーキテクチャ

今まで紹介した役割や機能は、現代のブラウザでは、複数のプロセス上で動いていることが多いです。このようなブラウザの設計方法をマルチプロセスアーキテクチャと呼びます。

本書で実装するブラウザはとてもシンプルなため、プロセスやスレッドを複数使用することはありませんが、現代のブラウザを支える重要な技術であるため紹介します。

プロセス

プロセスとは、実行中のプログラムのインスタンスのことです。実行可能なプログラムのコード、プログラム中で使用されている変数、Program Counter（PC）や Instruction Pointer（IP）と呼ばれる実行に関する状態、CPU が内蔵する記憶回路であるレジスタなどによって構成されています。

プロセスは、メモリ内で独立した領域を持ち、ほかのプロセスのメモリにアクセスすることはできません。プロセス間で情報をやりとりしたい場合は、IPC（*Inter-Process Communication*、プロセス間通信）によって行います。

■ブラウザプロセス／レンダラプロセス

仕様などで決められているわけではないのですが、ブラウザでは主にブラウザプロセスとレンダラプロセスという最低でも 2 種類のプロセスが存在します。

ブラウザプロセスは、大切なデータの保管やネットワークの通信などの重要な機能を扱うためのプロセスです。レンダラプロセスは、Web サイトのコアとなる HTML、CSS、JavaScript を解釈し、描画するためのプロセスです。

ブラウザのアプリケーションを開くと、まずブラウザプロセスが起動します。そしてブラウザの 1 つのタブにざっくりと 1 つのレンダラプロセスが作成されます。つまり、ユーザーがブラウザ上で 10 個のタブを開いているなら、1 つのブラウザプロセスと、少なくとも 10 個以上のレンダラプロセスが存在することになります。厳密には `<iframe>` のコンテンツも別のレンダラプロセスで実行されるなど、必ずしも 1 対 1 対応ではありません。

なぜレンダラプロセスがブラウザプロセスから分離されているかというと、HTML、CSS、JavaScript などのリソースは誰が書いたものかわからないため、信用できないからです。プロセスが分離されているとメモリ空間も分離されます。もし悪意のある Web サイトにアクセスしてしまったとしても、攻撃者のコードはレンダラプロセスで動いているため、重要な情報を扱うブラウザプロセスのメモリにはアクセスできません。プロセスをたくさん使用するとメモリ使用量が増えるというデメリットがありますが、セキュリティを高めるためにプロセスを分けているのです。

スレッド

プロセスに似た概念にスレッドというものも存在します。スレッドを使用すると、異なるコンピュータプログラムを擬似的に並行処理できます。プロセスとは異なり、メモリ空間を共有するため、異なるスレッド間で同じデータにアクセスできます。しかし、複数のスレッドで動くプログラムから同じデータにアクセスするときにはどのような順序でアクセスするかに気を遣わなければなりません。

たとえば、とあるデータを初期化し、再度同じデータを読んで値を確認するプログラムがあるとします。そしてこのプログラムをスレッド A で実行したとします。しかし、異なるプログラムが異なるスレッド B で動いていたとして、スレッド B からデータを書き換えたとします。このデータの書き換えがスレッド

A のデータの初期化と読み込みの間で起こっていたら、スレッド A のプログラムから見ればデータが勝手に書き換わっているように見えます。このような問題を ABA 問題と言います。

複数のスレッドを扱うときは、ロックという機構を使って ABA 問題のような予期せぬ問題が起きないようにします。ロックは日本語で鍵を表す英語で、家に鍵をかけるような感覚で、触れてほしくないデータに対して「ロックを取る」というような使い方をします。先ほどの例では、スレッド A でデータを初期化した際にロックを取り、このデータにはほかのスレッドから変更できないようにします。そして、スレッド A 上で必要な実行を終えたときにロックを解除してほかのスレッドからも扱えるようにします。しかし、これもお互いがお互いのロックの解除を待つデッドロック問題などが起こり得るので、複数のスレッドを使用してプログラムを書くときには細心の注意が必要です。

■UI スレッド／メインスレッド

ブラウザでは、複数のスレッドでプログラムが実行されていることが一般的です。これは仕様などで決められているわけではないため、各ブラウザベンダーの実装によって詳細は異なります。しかし一般的には UI スレッドまたはメインスレッドと呼ばれる、ユーザーとのインタラクションを扱ったり画面の描画を行ったりするコアとなるスレッドと、ネットワークの処理などを行う複数のバックグラウンドスレッドから成り立っていることが多いです。

■ワーカースレッド

ユーザーがコントロールできるスレッドでワーカースレッドというものがあります。ワーカースレッドとは、バックグラウンドで実行されるスレッドで、Web アプリケーションのパフォーマンスと応答性を向上させるために使用されます。重い計算や非同期処理をワーカースレッド上で実行することで、UI スレッドをブロックすることがなくなるためスムーズさを維持します。

ワーカースレッドは JavaScript によって起動できます。

```
const worker = new Worker("worker.js");
```

JavaScript の言語自体にはマルチスレッドの機能は存在しません。ワーカースレッドはブラウザによって提供されている機能です。

COLUMN

iOS 上でのブラウザアプリ

シングルプロセスのブラウザ

つい先ほどブラウザはマルチプロセスで動いていると説明しましたが、iOS で動くブラウザアプリは例外です。Apple の規約により、アプリストアに掲載できる iOS アプリはシングルプロセスで動かさなければいけません。

また、レンダリングエンジンも Apple が開発している WebKit を使用する必要があります。具体的には `WKWebView`[注1] という API を使用して、HTML や CSS の解釈と描画を行う必要があります。`WKWebView` の API は任意の JavaScript を動かすことができるため、ある程度の自由度はありますが、自分で 0 から HTML や CSS を解釈するよりは制限があります。

EU でのデジタル市場法（DMA）

2023 年、iOS アプリに大きな影響を与えるデジタル市場法（DMA）[注2] が欧州連合（*European Union*）によって施行されました。Google、Apple、Microsoft、Amazon などの会社がゲートキーパーとして指定され、市場支配力の乱用を防ぎ新規参入をしやすくするための規制です。

これにより、Apple によって iOS アプリで設けられていた制限が緩和されることになりました。しかし、EU だけにしか適用されないため、日本で使用される iOS のブラウザアプリは引き続き `WKWebView` を使い続ける必要があります。

注 1　https://developer.apple.com/documentation/webkit/wkwebview
注 2　https://ec.europa.eu/commission/presscorner/detail/en/ip_23_4328

ブラウザのセキュリティ対策

本書で作成するブラウザでは、セキュリティの対策は一切しません。しかしブラウザにとって適切なセキュリティ対策を行うことは、ユーザーのプライバシーやデータ保護、Web アプリケーションの安全性を確保するために非常に重要です。適切なセキュリティ対策を講じることで、個人情報の漏洩（ろうえい）や不正アクセス、マルウェア感染といったリスクを最小限に抑え、安全に Web サイトをブラウジングできます。

サイト分離（*Site Isolation*）

サイト分離は、ブラウザが異なる Web サイトのリソースを別々のプロセスで実行することで、異なるサイト間の攻撃を防ぐための機構です。プロセスの解説の節で話したように、異なるプロセスは独立したメモリ空間を持つため、異なるサイトで動くプログラムが互いに干渉することはできません。

現代のブラウザでは、マルチプロセスアーキテクチャの設計において、サイトの単位によってプロセスを分離することが多いです。具体的には、HTTP などのスキームと有効なトップレベルドメイン（.com、.jp など）とその 1 つ上のドメイン（eTLD+1）が同じであれば同サイトとし、同じプロセスで実行されます（**表 1-1**）。同じサイトのことを Same-site、異なるサイトのことを Cross-site と呼びます。

表 1-1 Same-site と Cross-site の例

URL1	URL2	同じサイトかどうか
https://www.example.com:443	https://www.evil.com:443	Cross-site（eTLD+1 が異なる）
https://www.example.com:443	https://abc.example.com:443	Same-site（"abc" は eTLD+2 なので関係ない）
https://www.example.com:443	http://www.example.com:443	Cross-site（スキームが異なる）
https://www.example.com:443	https://www.example.com:80	Same-site（ポート番号は関係ない）

サイト分離は、Spectre と Meltdown の脆弱性（ぜいじゃく）が見つかってから急激に重要性が高まりました。Spectre と Meltdown は、2018 年初頭に発表されたプロセッサのセキュリティ脆弱性で、現代のほぼすべての CPU に影響を及ぼす可能性があります。

Spectre は、CPU の予測実行機能を悪用して、サイドチャネル攻撃によって機密データを漏洩させる攻撃です。CPU は、プログラムの命令を実行するときに、将来の命令を予測して事前に実行できます。この予測が正しい場合にはパフォーマンスが向上しますが、予測が外れた場合には、キャッシュに不正に保存されたデータを攻撃者が利用できます。

Meltdown は、CPU の保護機構をバイパスし、カーネルメモリにアクセスできる脆弱性です。通常、アプリケーションは OS のカーネルメモリにアクセスできないように保護されていますが、Meltdown はこれを回避してカーネルメモ

リを直接読み取ることができます。

同一生成元ポリシー（*Same Origin Policy*）

同一生成元ポリシーは、異なるオリジン間のリソースへのアクセスを制限するセキュリティ機構です。スキーム、ホスト、そしてポート番号が同じ場合、オリジンは同じだとされます（**表 1-2**）。同じオリジンのことを Same-origin、異なるオリジンのことを Cross-origin と呼びます。異なるオリジンからのリソースへのアクセスを制限することにより、クロスサイトスクリプティング（XSS）やクロスサイトリクエストフォージェリ（CSRF）などの攻撃を軽減する、または防ぐことができます。

表 1-2 Same-origin と Cross-origin の例

URL1	URL2	同じ Origin かどうか
https://www.example.com:443	https://www.evil.com:443	Cross-origin（ドメイン名が異なる）
https://www.example.com:443	https://abc.example.com:443	Cross-origin（サブドメインが異なる）
https://www.example.com:443	http://www.example.com:443	Cross-origin（スキームが異なる）
https://www.example.com:443	https://www.example.com:80	Cross-origin（ポート番号が異なる）
https://www.example.com:443	https://www.example.com:443	Same-origin

`window.opener` のような JavaScript の API を使用して、異なるオリジンの情報を参照しようとすると、特定の情報にはアクセスできなかったり書き込みができなかったりする制限が存在します。また、異なるオリジン間で `XMLHttpRequest` や `<img>` によって、ネットワーク越しにデータを読み書きする際は、以下のようなカテゴリに分けられます。

- 異なるオリジンへの書き込みは許可される。たとえば、リンクやフォームの送信などがある
- 異なるオリジンの埋め込みは許可される。たとえば、`<iframe>`、`<script>`、`<link>` などのタグによって異なるオリジンのリソースを自身の Web サイトに埋め込むことができる
- 異なるオリジンからの読み込みは一般に許可されない。もし異なるオリジンのリ

ソースを読み込みたいとき、後述する CORS で明示的に許可する必要がある

ただ、これらはあくまでも同一生成元ポリシーによる制約であって、そのほかのしくみによって許可したり、許可されなくすることも可能です。

■オリジン間リソース共有（*CORS*）

CORS（*Cross-Origin Resource Sharing*）とは、ブラウザが異なるオリジン間でリソースを共有するためのしくみです。CORS は、同一生成元ポリシーによって制限されるクロスオリジンへのリクエストをホワイトリスト形式で許可します。HTTP ヘッダを使用して、サーバがどのオリジンからのリクエストを許可するかを指定します。

CORS に関する HTTP レスポンスで使用できるヘッダは以下のようなものがあります。

- Access-Control-Allow-Origin
  特定のオリジンまたはアスタリスク（*）を指定して、どのオリジンからのリクエストを許可するかを示す
- Access-Control-Allow-Methods
  許可される HTTP メソッド（例：GET、POST、PUT、DELETE）を指定する
- Access-Control-Allow-Headers
  許可されるカスタム HTTP ヘッダを指定する
- Access-Control-Allow-Credentials
  認証情報（Cookie など）を含むリクエストを許可するかを示す

対して、CORS に関する HTTP リクエストで使用できるヘッダは `Origin` があります。

- Origin
  リクエスト元のオリジンを示す

コンテンツセキュリティポリシー（*CSP*）

CSP（*Content Security Policy*）は、Web ページがロードするリソースのソース

を制御するためのセキュリティポリシーです。CSP を使用すると、ブラウザは Web ページが許可されたドメインからのリソースのみを読み込むように強制され、クロスサイトスクリプティング（XSS）やインジェクション攻撃などのセキュリティリスクを軽減できます。

CSP は HTTP レスポンスヘッダまたは HTML のメタタグを使用して定義されます。HTTP レスポンスヘッダでは、`Content-Security-Policy` で定義できます。たとえば、以下のようなレスポンスヘッダがあるとします。

```
Content-Security-Policy: default-src 'self'; img-src *; script-src https://↵
trusted.cdn.com
```

こちらは以下のような意味を持ちます。

- **`default-src 'self'`**
  すべてのタイプのリソースに対して、同一オリジンからのみコンテンツを読み込むことができる
- **`img-src *`**
  画像はどのオリジンからも読み込むことができる
- **`script-src https://trusted.cdn.com`**
  JavaScript のスクリプトは `https://trusted.cdn.com` からのみ読み込むことができる

本書のゴール・注意点

次章から作っていくブラウザのサンプルアプリケーションは、ネットワークのやりとり、レンダリングエンジン、JavaScript エンジン、そして UI を実装します。しかし、マルチプロセスアーキテクチャによるプロセス分離や、そのほかのセキュリティに関する対策は一切実装しません。

あくまでもブラウザの基本動作を理解するためのシンプルなアプリケーションなので、自分の作ったアプリケーションを使用する際には十分に気を付けてください。