

JavaScript 入門

Web アプリケーション開発のための基礎

第 2 版 (プレビュー 24)

Copyright © 2024, Katsunori Nakamura

中村勝則

2024 年 8 月 20 日

免責事項

本書の内容は参考資料であり、掲載したプログラムリストは全て試作品である。本書の使用に伴って発生した不利益、損害の一切の責任を筆者ならびに IDEJ 出版は負わない。

目次

1	はじめに	1
1.1	本書の内容	1
1.2	前提事項	1
1.3	サンプルコードの提示に関すること	1
2	HTML と CSS	2
2.1	HTML	2
2.1.1	HTML の要素	2
2.1.1.1	HTML 要素の属性	2
2.1.2	HTML の基本的な構成	2
2.1.2.1	文書型宣言	2
2.1.2.2	html 要素	3
2.1.2.3	head 要素	3
2.1.2.4	body 要素：文書本体要素	3
2.1.3	HTML 文書の論理的構造	3
2.1.3.1	文書のレイアウトのための要素	4
2.1.3.2	記事の論理構造	4
2.1.3.3	まとめ	4
2.1.4	HTML 文書内のコメント	4
2.2	CSS	4
2.2.1	セレクタ	5
2.2.2	サイズや色の値の表現	5
2.2.3	CSS のコメント	6
2.3	HTML, CSS 各論	6
2.3.1	見出しと段落：h 要素, hgroup 要素, p 要素	6
2.3.1.1	span 要素	7
2.3.1.2	hgroup 要素	7
2.3.1.3	文字, テキストに関する設定	7
2.3.1.4	文字の書体（フォント）に関する設定	9
2.3.1.5	ルビ	11
2.3.2	ハイパーリンクの設置	11
2.3.2.1	a 要素に対する特殊なセレクタ	12
2.3.3	HTML 要素のレイアウト	12
2.3.3.1	HTML 要素をレイアウトするためのボックス構造	14
2.3.3.2	div 要素	15
2.3.3.3	レイアウトと包含ブロック	15
2.3.3.4	要素の位置について	16
2.3.3.5	ボックス構造の位置とサイズに関すること	17
2.3.3.6	コンテナ内外のレイアウト制御	19
2.3.4	箇条書き：ul 要素, ol 要素, li 要素	19
2.3.5	表：table 要素	20
2.3.5.1	thead, tbody, ifoot 要素	21
2.3.6	画像の埋め込み：img 要素	21
2.3.7	不透明度	22
2.3.8	可視属性	22
2.3.9	form 要素	22

2.3.9.1	form 要素の記述形式と Web サーバとの連携	23
2.3.9.2	form 要素に依らない UI 構築	23
2.3.9.3	ラベル	23
2.3.9.4	テキスト入力 (1): 単一行の入力	23
2.3.9.5	テキスト入力 (2): 複数行の入力	24
2.3.9.6	パスワード入力	24
2.3.9.7	ボタン	24
2.3.9.8	スライダー	24
2.3.9.9	データリスト要素	25
2.3.9.10	選択要素	26
2.3.9.11	チェックボックス	26
2.3.9.12	ラジオボタン	27
2.3.9.13	フィールドセット要素	27
2.3.9.14	メータと進捗インジケータ (プログレスバー)	28
2.3.10	外部リソースへのリンク要素	29
2.3.11	スタイルシートの形態	29
3	JavaScript	30
3.1	前提事項	30
3.1.1	コンソール出力	30
3.1.2	文を複数の行に分割して記述する方法	31
3.1.3	プログラム中のコメント	31
3.2	文とブロック	31
3.3	関数, メソッド, プロパティ	31
3.4	変数	32
3.4.1	変数のスコープ	32
3.4.2	変数の使用における良くない例	33
3.4.2.1	更に注意すべき事柄	34
3.4.3	分割代入	34
3.4.3.1	分割代入における「オブジェクトのパターン」	36
3.4.3.2	分割代入の高度な応用	36
3.4.4	変数の廃棄	36
3.5	基本的なデータ型	37
3.5.1	数値	37
3.5.1.1	基本的な演算	37
3.5.1.2	累算的な代入演算子	38
3.5.1.3	特殊な値	38
3.5.1.4	長整数: BigInt	39
3.5.1.5	数値リテラル	39
3.5.1.6	数学関数	40
3.5.2	論理値	41
3.5.3	文字列	41
3.5.3.1	エスケープシーケンス	42
3.5.3.2	複数行に渡る文字列の記述	42
3.5.3.3	文字列の長さ (文字数)	42
3.5.3.4	文字列の連結と分解	43
3.5.3.5	文字列の繰り返し	44
3.5.3.6	部分文字列	44

3.5.3.7	文字列の含有検査	44
3.5.3.8	文字列の探索	44
3.5.3.9	テンプレートリテラル (テンプレート文字列)	45
3.5.3.10	文字列を数値に変換する方法	46
3.5.4	シンボル	47
3.5.5	値の比較	47
3.5.6	値の型に関すること	48
3.6	データ構造	50
3.6.1	配列: Array	50
3.6.1.1	空配列	51
3.6.1.2	要素の追加	51
3.6.1.3	要素の削除	52
3.6.1.4	配列の連結	53
3.6.1.5	部分配列	53
3.6.1.6	配列の編集	53
3.6.1.7	配列の反転	54
3.6.1.8	要素の整列	54
3.6.1.9	指定した要素を配列内に並べる方法	55
3.6.1.10	要素の存在や位置の調査	55
3.6.1.11	配列の並びを表す文字列の取得	56
3.6.1.12	全要素に対する条件検査	56
3.6.1.13	条件を満たす要素の探索	57
3.6.1.14	特殊なプロパティ	58
3.6.2	オブジェクト: Object	58
3.6.2.1	要素の削除	59
3.6.2.2	プロパティのキーに使用できるデータ	59
3.6.2.3	空オブジェクト	60
3.6.2.4	オブジェクトから配列への変換	60
3.6.2.5	オブジェクトの要素の個数を求める方法	60
3.6.2.6	計算されたプロパティ名	60
3.6.3	Map オブジェクト	61
3.6.3.1	要素へのアクセス	61
3.6.3.2	要素の存在確認	61
3.6.3.3	要素の個数の調査	62
3.6.3.4	全要素の取出し	62
3.6.3.5	要素の削除	62
3.6.3.6	ミュータブルなデータをキーに使用する際の注意	63
3.6.4	Set オブジェクト	64
3.6.4.1	要素の存在確認	64
3.6.4.2	要素の追加と削除	64
3.6.4.3	要素の個数の調査	64
3.6.4.4	配列への変換	65
3.6.4.5	全要素の削除	65
3.6.4.6	ミュータブルなデータを要素にする際の注意	65
3.6.5	スプレッド構文	66
3.7	特殊なデータ構造	67
3.7.1	型付き配列 (TypedArray) と ArrayBuffer	67
3.7.1.1	ArrayBuffer の複数のビューによる共有	68

3.7.1.2	バイトオーダー (リトルエンディアン/ビッグエンディアン)	69
3.7.2	データビュー (DataView)	70
3.7.3	文字列⇄バイナリデータの変換	72
3.7.3.1	Node.js での方法	72
3.7.4	Blob	73
3.7.4.1	Blob 作成の例	73
3.7.4.2	Blob のデータの取り出し	74
3.7.5	BinaryString	74
3.7.5.1	Blob との間での変換	74
3.8	条件分岐	75
3.8.1	if ... else 文	75
3.8.2	switch 文	76
3.8.3	値を選択する 3 項演算子	77
3.8.4	論理値以外を条件式に用いるケース	77
3.9	反復制御	79
3.9.1	while 文	79
3.9.2	do ... while 文	79
3.9.3	for 文	80
3.9.3.1	for ... of 文	80
3.9.3.2	for ... in 文	81
3.9.4	break と continue	82
3.10	関数の定義	83
3.10.1	変数のスコープ	83
3.10.1.1	変数の使用における安全でない例	84
3.10.2	引数に関する事柄	85
3.10.2.1	仮引数と実引数	85
3.10.2.2	任意の個数の引数を扱う方法	85
3.10.2.3	仮引数にデフォルト値を設定する方法	85
3.10.3	関数自体のスコープ	86
3.10.4	関数式	87
3.10.4.1	即時実行関数	87
3.10.4.2	アロー関数式	87
3.10.5	関数の再帰的呼び出し	88
3.10.6	ネストされた関数定義 (内部関数)	89
3.10.6.1	クロージャ	89
3.10.7	オブジェクトとメソッドの考え方	91
3.10.8	ジェネレータ関数	92
3.10.8.1	ジェネレータ関数の入れ子	93
3.11	イテラブルオブジェクトとイテレータ	94
3.11.1	イテラブルオブジェクト	94
3.12	オブジェクト指向プログラミング (OOP)	96
3.12.1	メソッドの実装	97
3.12.1.1	prototype プロパティ	97
3.12.2	インスタンスのコンストラクタを調べる方法	98
3.12.3	Object.create によるオブジェクトの生成	98
3.12.4	プロパティ継承の仕組み	98
3.12.5	class 構文による方法	101
3.12.5.1	コンストラクタ, メソッドの記述	102

3.12.5.2	パブリックフィールド	102
3.12.5.3	静的メソッド	103
3.12.5.4	クラスの継承	104
3.12.5.5	オブジェクトのクラスを調べる方法	105
3.12.5.6	プロパティのカプセル化	106
3.12.5.7	class 式	106
3.13	日付, 時刻の扱い	108
3.13.1	現在時刻の取得	108
3.13.2	値の取得と設定	108
3.13.3	データ形式の変換	109
3.13.3.1	Date オブジェクト→文字列	109
3.13.3.2	文字列→Date オブジェクト	110
3.14	正規表現	111
3.14.1	文字列探索 (検索)	111
3.14.1.1	RegExp オブジェクト	112
3.14.2	パターンマッチ	112
3.14.2.1	マッチした部分の抽出	112
3.14.2.2	行頭, 行末でのパターンマッチ	113
3.14.2.3	パターンマッチの繰り返し実行	113
3.14.2.4	複数行のテキストに対するパターンマッチ	113
3.14.3	置換処理	114
3.15	例外処理	116
3.15.1	エラーオブジェクト	116
3.15.1.1	例外 (エラー) の種類	117
3.15.2	例外を発生させる方法	118
3.16	その他の便利な機能	120
3.16.1	forEach による反復処理	120
3.16.2	配列の全要素に対する一斉処理	121
3.16.2.1	対象要素のインデックスを取得する方法	121
3.16.3	配列に対する二項演算の連鎖的実行	122
4	Web アプリケーション開発のための基礎	123
4.1	HTML と JavaScript	123
4.2	DOM に基づいたプログラミング	124
4.2.1	DOM の最上位のオブジェクト	124
4.2.1.1	グローバル変数の所在	124
4.2.2	HTML の最上位のオブジェクト	124
4.2.3	HTML 要素のクラス	124
4.2.4	HTML の木構造 (階層構造)	125
4.2.4.1	子要素, 親要素を取得する方法	125
4.2.4.2	document 直下の子要素	126
4.2.4.3	childNodes プロパティ	126
4.2.5	HTML 要素の id 属性から要素ノードを取得する方法	127
4.2.6	HTML 要素の要素名 (タグ名) の取得	127
4.2.7	HTML 要素の生成	127
4.2.7.1	HTML 要素の属性を設定する方法	128
4.2.7.2	HTML 要素の属性を取得する方法	128
4.2.7.3	HTML 要素のデータ属性	128

4.2.8	テキストノードの作成	129
4.2.9	子ノードの追加	129
4.2.9.1	子ノードの追加位置の指定	129
4.2.10	子ノードの削除	131
4.2.11	HTML 要素にテキストコンテンツを与える方法	131
4.2.12	HTML 要素の class 属性から要素ノードを取得する方法	131
4.2.13	DOMParser	133
4.2.14	CSS のセレクタから HTML 要素を取得する方法	134
4.3	JavaScript で CSS にアクセスする方法	135
4.3.1	HTML 要素のスタイルの取得	135
4.3.2	スタイルの動的な変更	135
4.3.2.1	style プロパティ配下の CSS 属性名に関する注意	136
4.4	イベント駆動型プログラミング	137
4.4.1	イベントハンドリングの仕組み	137
4.4.2	イベントの種類	138
4.4.2.1	ページの読み込み, 離脱	138
4.4.2.2	マウスの操作	138
4.4.2.3	キーボードの操作	138
4.4.2.4	変化や選択によって発生するイベント	138
4.4.2.5	その他のイベント	139
4.4.3	addEventListener によるイベントハンドラの登録	139
4.4.4	イベント名のプロパティにイベントハンドラを登録する方法	140
4.4.5	イベントオブジェクト	140
4.4.5.1	イベントオブジェクトの使用例 (マウス関連)	141
4.4.5.2	イベントオブジェクトの使用例 (キーボード関連)	142
4.4.6	タイマー	143
4.4.6.1	タイマー処理の反復	143
4.4.6.2	タイマーの反復処理の応用例	144
4.5	表示環境に関する事柄	145
4.5.1	ディスプレイ (スクリーン) 関連	145
4.5.2	ブラウザウィンドウ, コンテンツページ関連	145
4.5.3	表示の解像度を求める方法	146
4.5.4	px (ピクセル) から pt (ポイント) への単位の変換	147
4.5.5	ビューポート内での HTML 要素の位置とサイズ	148
4.6	ローカル環境でのデータの保存	150
4.6.1	localStorage	150
4.6.1.1	データの保存	150
4.6.1.2	データの取得	151
4.6.1.3	キーの取得	151
4.6.1.4	サンプルプログラム: 全データの取得	152
4.6.1.5	データの消去	152
4.6.2	sessionStorage	152
4.7	画像, 音声の扱い	153
4.7.1	静止画像の扱い	153
4.7.1.1	静止画像を動的に切り替える方法	153
4.7.1.2	img 要素の生成と追加	153
4.7.2	音声, 動画の扱い	155
4.7.2.1	HTMLMediaElement クラス	155

4.7.2.2	再生と一時停止	155
4.7.2.3	各種のプロパティ	155
4.7.2.4	各種のイベント	156
4.7.2.5	動画：HTMLVideoElement オブジェクト	156
4.7.2.6	応用例	157
4.8	非同期処理と Promise	158
4.8.1	Promise オブジェクト	158
4.8.2	実装例	159
4.9	JSON	163
4.9.1	JSON データの作成	163
4.9.2	JSON データの展開	163
4.10	通信のためのデータ変換	164
4.10.1	URL エンコーディング	164
4.10.2	Base64	164
4.10.2.1	base64-js ライブラリ	165
4.11	ファイルの入出力	167
4.11.1	ファイルの読み込み	167
4.11.1.1	input 要素によるファイル選択ダイアログ	167
4.11.1.2	FileReader によるファイルの読み込み	168
4.11.2	ファイルの保存	170
4.12	ドラッグアンドドロップ	172
4.12.1	ドラッグアンドドロップに伴う処理	172
4.12.1.1	dragstart イベントを受けて行う処理	173
4.12.1.2	dragover イベントを受けて行う処理	173
4.12.1.3	drop イベントを受けて行う処理	173
4.12.1.4	イベントオブジェクトの target 属性	173
4.13	プログラムの分割開発	175
4.13.1	単純な分割開発	175
4.13.2	モジュールによる分割開発	175
4.13.2.1	モジュール内のオブジェクトを公開する方法 (1)：DOM を介する	176
4.13.2.2	モジュール内のオブジェクトを公開する方法 (2)：export で公開する	176
4.13.2.3	モジュール毎の名前空間を実現するための工夫	177
5	Web Workers	180
5.1	基本的な考え方	180
5.1.1	ワーカーのプログラミングの考え方	180
5.2	ワーカーの並行実行	182
5.3	メッセージ送信におけるデータの複製と移転	184
6	通信のための機能	186
6.1	Fetch API	186
6.1.1	基本的な通信の手順	186
6.1.2	Promise による非同期処理	186
6.1.3	サーバからの応答の解析	187
6.1.4	応答からのデータ本体の取り出し	187
6.1.5	例外処理	187
6.1.6	Fetch API の使用例	188

7	グラフィックスの作成 (1) : canvas 要素	190
7.1	座標と角度の考え方	190
7.2	描画領域のサイズ	190
7.3	描画コンテキスト	191
7.3.1	フィルに関する描画の設定	191
7.3.2	輪郭線に関する描画の設定	191
7.3.2.1	線幅 (線の太さ) と始点, 終点の座標の考え方	191
7.3.2.2	線の端の形状の設定	191
7.3.2.3	線の結合部の形状の設定	191
7.3.2.4	破線の設定	192
7.4	描画メソッド	193
7.4.1	矩形 (四角形) の描画	193
7.4.2	テキストの描画	193
7.4.2.1	矩形とテキストを描画するサンプル	193
7.4.3	パスの描画	194
7.4.3.1	折れ線の描画	194
7.4.3.2	円弧の描画	196
7.4.3.3	楕円弧の描画	197
7.4.4	画像の描画	198
7.4.5	画素 (ピクセル) の描画と取得	199
7.4.5.1	ImageData オブジェクトの画素の構造	199
7.4.5.2	ImageData オブジェクトの描画	199
7.4.5.3	canvas から画素を取得する方法	201
7.5	スケール, 傾き, 原点の変更	201
7.5.1	描画コンテキストの状態の保存と復元	202
8	グラフィックスの作成 (2) : SVG	203
8.1	基礎事項	203
8.1.1	座標系と大きさ, 角度の単位	203
8.1.2	記述の形式	203
8.1.3	ビューポートとビューボックス	203
8.1.4	HTML への SVG の配置	204
8.1.4.1	img 要素として配置する方法	204
8.1.4.2	object 要素として配置する方法	205
8.1.5	defs 要素による定義の再利用	206
8.2	SVG の図形要素	207
8.2.1	多角形, 折れ線	207
8.2.2	輪郭線に関する設定事項	208
8.2.2.1	破線の設定	208
8.2.2.2	線の端の形状の設定	209
8.2.2.3	線の結合部の形状の設定	209
8.2.3	矩形: rect 要素	210
8.2.4	円: circle 要素	210
8.2.5	楕円: ellipse 要素	211
8.2.6	直線: line 要素	211
8.2.7	テキスト (文字列): text 要素	211
8.2.7.1	テキストの描画位置の基準	212
8.2.7.2	テキストの強調	212

8.2.7.3	フォントスタイル	212
8.2.7.4	下線, 上線, 打消し線	212
8.2.7.5	文字の回転	213
8.2.8	パス: path 要素	213
8.2.8.1	直線の描画	213
8.2.8.2	楕円弧	214
8.2.9	画像データの配置	216
8.2.10	transform 属性	217
8.2.10.1	変換の重ね合わせ	218
8.2.10.2	use 要素による定義の展開	218
8.2.10.3	図形要素のグループ化	219
9	プログラムライブラリ (1): jQuery	220
9.1	jQuery の導入	220
9.2	jQuery オブジェクト	220
9.2.1	関数の起動	221
9.2.2	HTML 要素へのアクセス	221
9.2.3	イベントハンドリングの登録	222
9.2.4	HTML 要素の生成と DOM への追加	222
9.2.5	jQuery オブジェクトが保持する HTML 要素	222
9.2.6	jQuery で扱う HTML 要素	223
9.2.7	jQuery における window, document	223
9.3	CSS 属性へのアクセス	223
9.4	イベントハンドリングのためのメソッド	224
9.5	HTML 要素の属性へのアクセス	224
9.5.1	属性値の動的な扱い	225
9.5.1.1	チェックボックスの扱い	225
9.5.1.2	ラジオボタンの扱い	226
9.5.1.3	データ属性の扱い	227
9.5.2	class 属性の扱い	227
9.6	DOM の操作	229
9.6.1	要素の追加	229
9.6.2	要素の削除	231
9.6.3	子要素, 親要素, 内部要素の取得	231
9.7	要素の選択	233
9.8	アニメーション効果	235
9.9	Web アプリケーション実装に関すること	236
10	プログラムライブラリ (2): MathJax	238
11	プログラムライブラリ (3): React	240
11.1	基礎事項	240
11.1.1	基本的なライブラリ	240
11.1.2	React アプリのレンダリング	240
11.1.3	React 要素とコンポーネント	241
11.1.3.1	JSX	241
11.1.3.2	クラスコンポーネント, 関数コンポーネント	242
11.1.4	コンポーネントに値を渡す方法: Props	243
11.1.5	コンポーネントの State とライフサイクル	244

11.1.5.1	State	244
11.1.5.2	ライフサイクル	245
11.1.5.3	コンポーネントのスタイルの設定	245
11.1.5.4	条件付きレンダリング	245
11.1.5.5	実装例に沿った解説	246
11.1.6	コンポーネントのイベントハンドリング	250
11.1.7	複数コンポーネントの一括デプロイ：リストとキー	251
11.2	実用的な使用方法	253
11.2.1	npm による開発作業の概観	253
11.2.1.1	必要なソフトウェアのインストール	253
11.2.1.2	ソースコードの用意	254
11.2.1.3	ビルド作業のための設定ファイルの準備	254
11.2.1.4	ビルド作業の実行	255
11.2.2	ソースコードの分割開発	256
11.2.3	ビルド作業時のリソース複製の設定：copy-webpack-plugin	258
11.2.4	コンポーネント実装のための簡便な方法	258
A	Web サーバ関連	260
A.1	Node.js による HTTP サーバ	260
A.1.1	ライブラリの読み込み	260
A.1.2	サーバの作成と起動	260
A.1.3	サーバの処理の実装	261
A.1.3.1	URL の解析	261
A.1.3.2	クエリ文字列の解析	262
A.1.3.3	リクエストオブジェクトの解析	262
A.1.3.4	クライアントへの応答のための処理	262
A.1.3.5	POST メソッドで受け取ったデータの受理	263
A.1.3.6	エラーハンドリング（例外処理）	264
A.1.4	サーバの実装例	264
A.1.4.1	単純な例	264
A.1.4.2	POST されたデータを受理する例	265
A.1.5	Express ライブラリによる HTTP サーバ	266
A.1.5.1	サーバの作成	266
A.1.5.2	リクエストへの応答処理の実装	266
A.1.5.3	サーバの起動	268
A.1.5.4	サーバの実装例（1）：リクエストのルーティング	268
A.1.5.5	サーバの実装例（2）：リクエストの解析	269
A.1.5.6	サーバの実装例（3）：ルートパラメータの取得	270
A.2	Python による HTTP サーバ	272
A.2.1	http モジュールによる素朴な HTTP サーバ	272
A.2.1.1	HTTPServer クラス	272
A.2.1.2	BaseHTTPRequestHandler クラス	272
A.2.1.3	サーバの実装例	273
A.2.2	Flask	276
A.2.2.1	最も簡単な HTTP サーバの実装例	276
A.2.2.2	リクエストの情報や POST されたデータの取得	277
A.3	Apache HTTP Server	281
A.3.1	ソフトウェアの入手	281

A.3.2	ディレクトリの構成	281
A.3.3	インストール作業と設定	281
A.3.4	CGI 関連	283
A.3.4.1	CGI を有効にする設定	283
A.3.4.2	CGI の実装	283
A.3.4.3	CGI のサンプル (1): 単純な例	284
A.3.4.4	CGI のサンプル (2): 環境変数と POST された内容の表示	286
A.3.5	WSGI 関連	288
A.3.5.1	WSGI を有効にする設定	288
A.3.5.2	WSGI スクリプトの実装	289
A.3.5.3	WSGI のサンプル (1): 単純な例	290
A.3.5.4	WSGI のサンプル (2): 環境変数と POST された内容の表示	291
A.4	ローカルの計算機環境で HTTP サーバを起動する簡易な方法	293
A.4.1	Node.js による簡易な方法	293
A.4.2	Python による簡易な方法	293
A.5	WebSocket	294
A.5.1	Node.js による WebSocket	294
A.5.1.1	サーバインスタンスの生成と設定	294
A.5.1.2	クライアントとの通信に関する処理の実装	295
A.5.1.3	エラーハンドリング	295
A.5.1.4	通信相手に対するメッセージの送信	296
A.5.1.5	接続中のクライアントの一覧	296
A.5.1.6	実装例	296
A.5.1.7	URL のパスに基づくルーティング	298
A.5.2	Python による WebSocket	302
A.5.2.1	基本的な考え方	302
A.5.2.2	サーバの作成と実行の流れ	302
A.5.2.3	ハンドラ関数	302
A.5.2.4	サーバの実装例 (1): 非同期のイテレーションによる方法	303
A.5.2.5	サーバの実装例 (2): 通常の反復による方法	305
B	Web ブラウザ関連	306
B.1	スクリプトのキャッシュの抑止	306
C	Node.js	307
C.1	REPL による対話的な利用方法	307
C.1.1	REPL の終了	307
C.1.2	プログラムの読み込み	307
C.2	標準入力からの同期入力	308

1 はじめに

本書は主に JavaScript 言語の基礎的な事柄について解説する。また、Web アプリケーションの開発に必要なとなる最小限の範囲で HTML や CSS といった事柄についても解説する。

JavaScript は ECMAScript として標準化されており、その言語処理系は各種 Web ブラウザや Web サーバを始めとするシステムに実装されている。また、JavaScript で記述されたプログラムの実行速度は概して大きく、この言語の応用範囲が広がっている。

JavaScript 言語処理系は Web ブラウザに搭載されたもの¹ と、サーバサイド用に開発されたもの² があり、前者は Web アプリケーションのフロントエンドを支え、後者は Web サーバ側の処理を実現する。どちらも ECMAScript に基づく言語処理系であるが、それぞれの目的に合わせた API が ECMAScript から独立した形で言語処理系に実装されている。

1.1 本書の内容

本書は Web アプリケーション開発に必要なとなる最小限度の内容を学ぶための入門書であり、技術項目の包括的なリファレンスではないことを予め了承されたい。

本書の第 2 章では、Web アプリケーションの開発に必要なとなる最小限の範囲で HTML や CSS について解説する。これらに関する知識を習得済みの読者の方には第 2 章の内容が不要な場合もあるので、読者の知識状態によって判断されたい。

本書の第 3 章は JavaScript の基礎事項についての内容である。プログラミング言語としての基礎事項について解説しているので、JavaScript 言語を学ぶ読者にとっては基盤となる内容である。

本書の第 4 章は、Web ブラウザ上で動作する Web アプリケーションを構築するための基礎事項についての内容である。HTML の論理構造を表現するモデルである DOM に関する基礎事項や、Web アプリケーションのための基本的なプログラミングモデルであるイベント駆動型プログラミングをはじめとする技術的な内容について取り扱う。

本書の第 5 章では、Web アプリケーションでプログラムの並行実行を実現する Web Workers について解説する。

本書の第 6 章、第 7 章は Web ブラウザ上でグラフィックスの描画を実現するための基礎事項についての内容である。また第 8 章、第 9 章は、Web アプリケーション構築に役立つプログラムライブラリに関する内容である。

1.2 前提事項

本書で解説する HTML, CSS, JavaScript に関する事柄は Web ブラウザ上で表示、実行されることを前提とする。特に断らない限り、使用する Web ブラウザは Google Chrome とする。Microsoft 社の Edge や Mozilla Firefox, Apple 社の Safari といったブラウザも使用可能であるが、HTML のレンダリングや JavaScript プログラムからの出力に若干の違いが起こる可能性があることを了承されたい。

使用する計算機環境は、上記 Web ブラウザを実行できるシステムであれば良い。計算機の機種、OS などは特に限定しない。また、HTML, CSS, JavaScript のコードを記述するためのソフトウェアツールも特に限定せず、基本的にはテキストエディタと Web ブラウザがあれば本書で挙げるサンプルコードは実行可能である。

本書の巻末付録に紹介する事柄や、参考的内容を紹介する際に、Node.js や Python 言語処理系を使用するケースを挙げるがあるので、それらを利用可能にしておくことを推奨する。

1.3 サンプルコードの提示に関すること

本書で提示する各種のサンプルコードは、実行と結果の確認を簡便な形で行えることを意識して作られている。従って、実用的な Web システムの構築における局面とは異なる形式、作法で記述されることもあることをあらかじめ了承されたい。

¹Google Chrome の V8 や、Mozilla Firefox の SpiderMonkey, macOS の Safari に搭載された JavaScriptCore などが有名。

²Node.js (<https://nodejs.org/>) が有名。

2 HTML と CSS

Web アプリケーションのユーザインターフェース（UI : User Interface）は HTML と CSS で構成される。ここでは、HTML に関する基礎事項について解説する。

2.1 HTML

HTML (HyperText Markup Language) はハイパーテキストを記述するためのマークアップ言語³ であり、最新の言語仕様は WHATWG (Web Hypertext Application Technology Working Group)⁴ が W3C (World Wide Web Consortium)⁵ と共に HTML Living Standard として⁶ 定めている。

HTML 文書はテキストデータ（可読な文字で構成されるデータ）として記述し、テキストファイルとしてストレージに保存する。

2.1.1 HTML の要素

HTML の文書を構成する要素（HTML 要素）は次のような形式である。

書き方： <要素名>コンテンツ</要素名>

<…> のような記述をタグ (Tag) と呼び、<要素名> を開始タグ、</要素名> を終了タグと呼ぶ。「コンテンツ」の部分にはテキストデータや別の要素を子要素として含むことができる。終了タグは省略できる場合があるが、特に空要素 (Void element) に分類される要素は子要素を持たず、終了タグを記述しない。(開始タグのみの記述) 空要素のタグ内の末尾にスラッシュ「/」を記述する場合があるが、これは必須ではない。

例. 空要素 input タグの末尾のスラッシュ

```
<input type="range" />
```

これは、XHTML⁷ の仕様に従ったものであるが、いくつかのプログラミング用のツール（開発環境内のエディタなど）では XHTML を踏襲し、空要素のタグ内の末尾のスラッシュを自動的に補完して挿入する場合もある。ただし本書では、このような末尾のスラッシュは基本的には書かないこととする。

2.1.1.1 HTML 要素の属性

開始タグの中には各種の属性を記述することができる。

書き方： <要素名 属性名 1=値 1 属性名 2=値 2 … >

特に重要な属性名として「id」と「class」がある。これらは文書内の各要素を識別するための名前を与えるものである。id 属性は HTML 要素に一意名を与える。また、複数の HTML 要素に同じ class 属性の値を設定することができ、その値によって要素のグループ化ができる。

id, class 属性は、CSS や JavaScript から HTML 要素を識別する際に必要となる。

要素名や属性名の具体的なもの（重要なもの）は以後順次解説する。

2.1.2 HTML の基本的な構成

2.1.2.1 文書型宣言

HTML 文書を記述する際には文書型宣言を記述する。

書き方： <!DOCTYPE html>

これにより、標準的な HTML の規格に沿った形（標準モード）で HTML 文書を表示することを Web ブラウザに指示することができる。この記述を省略すると Web ブラウザは互換モードで HTML 文書を扱うこととなり、表示の際の体裁が予期しないものになることがあるので注意すること。

³HTML は SGML を基に開発された。SGML を基にしたものとして他に XML がある。

⁴<https://whatwg.org/>

⁵<https://www.w3.org/>

⁶HTML5 という仕様は 2021 年に廃止された。ただし、HTML5 の仕様の多くは HTML Living Standard に踏襲されている。

⁷XML の仕様に基づき HTML を再定義したものであるが、2009 年に開発は終了した。

2.1.2.2 html 要素

文書型宣言に続いて html 要素を記述する。

書き方： `<html lang="言語コード"> コンテンツ </html>`

html 要素は HTML 文書の最上位の要素であり **ルート要素**とも呼ばれる。lang 属性には ISO 639-1 に規定された言語コードを指定⁸する。例えば、日本語の HTML 文書を作成する場合は `<html lang="ja">`、英語の HTML 文書を作成する場合は `<html lang="en">` と記述する。lang 属性は **グローバル属性**であり、全ての HTML 要素に適用⁹される。lang 属性は省略することが可能であり、その場合は Web ブラウザが文書の言語を適宜判断する。

html 要素は子要素として次に説明する head 要素と body 要素を持つ。

2.1.2.3 head 要素

head 要素は文書内容として Web ブラウザには表示されないものであり、文書のタイトル情報や各種のメタ情報、スタイルの設定などを記述する部分である。head 要素の子要素となる重要なものについて以下に解説する。

■ meta 要素

HTML 文書に関する各種のメタ情報を meta 要素の属性として記述する。meta 要素に記述した情報は当該 HTML 文書を閲覧するシステム¹⁰が使用することがある。特に charset 属性は HTML 文書を記述する文字の **エンコーディング**を指定するものとして重要である。

例. 当該 HTML 文書のエンコーディングが UTF-8 であることを示す場合

```
<meta charset="utf-8">
```

また、name 属性と content 属性も重要となる場合がある。(下の例参照)

```
著者      : <meta name="author" content="中村勝則">
概要      : <meta name="description" content="この文書の概要">
関連キーワード : <meta name="keywords" content="語 1, 語 2, 語 3">
```

■ title 要素

title 要素は HTML 文書に題名情報を与える。

書き方： `<title>題名情報</title>`

「題名情報」にはテキストデータのみを記述する。この要素はテキスト以外の子要素（他の HTML 要素など）を持たない。この題名情報は当該 HTML 文書を Web ブラウザで表示した際にタブなどの部分に表示されるので簡潔に（短く）記述することが望ましい。

HTML の仕様では title 要素は必須とされるが、これを省略しても Web ブラウザ上では当該 HTML 文書は問題なく表示される。ただし、ユーザビリティや SEO¹¹の観点からも title 要素は省略するべきではない。

2.1.2.4 body 要素：文書本体要素

HTML 文書の本体部分（Web ブラウザに実際に表示される部分）は body 要素として記述する。この要素は html 要素の子要素として 1 つだけ記述する。

以下、各種の HTML 要素を配置して HTML 文書を作る。

2.1.3 HTML 文書の論理的構造

HTML は文書を記述することが元来の目的の 1 つであり、その意味ではいわゆる **ページ記述言語**¹²とは立場が異なる。ページ記述言語は、印刷や表示のための出力装置の制御が目的であるのに対し、文書は論理的な構造に基づいて内容を読み手に伝えるものである。例えば、日本語の文書表現においても、「章→節→項→段落」といった階層的な論理構造に従って内容を整然と記述している。HTML においてもこれと類似の文書構造を **h 要素**や **p 要素**など（後

⁸ロケール (locale) を指定する場合もある。

⁹html 要素の子要素に個別に lang 属性を与えることも可能である。

¹⁰Web ブラウザやクローラなど。

¹¹検索エンジン最適化 (Search Engine Optimization)

¹²印刷出力装置のための PostScript が有名。

述)で実現することができる。

ここでは、body 要素の配下で文書構造を作り上げるための各種の要素について解説する。

2.1.3.1 文書のレイアウトのための要素

HTML では body 要素の内容をナビゲーション部、メイン部、フッター部に分けてレイアウトすることができ、そのための要素 nav, main, footer が用意されている、それら要素は body 要素の直下に配置する。

nav 要素はコンテンツページの閲覧の際のナビゲーションに関する内容などを記述する。例えば、nav 要素にページ内の各部へのリンクをまとめ、コンテンツページのユーザビリティを高めるといった応用が可能である。

main 要素は具体的な記事内容を記述する部分であり、body 要素の子要素として1つ配置する。

footer 要素はコンテンツページの終了部分に記述する内容を持つもので、通常はコンテンツページの最下部に配置する。

以上の要素を、後に説明する CSS を用いて適切な位置に配置して Web ブラウザに表示することができる。

2.1.3.2 記事の論理構造

main 要素の配下で具体的な記事内容を article 要素として記述する。すなわち、1つの記事を1つの article 要素として記述し、必要に応じて複数の記事を複数の article 要素として記述する。

article 要素の子要素として記事を記述する際、適宜 h 要素で見出しを記述し、それに続いて段落を p 要素で必要だけ記述する。h 要素は h1 ~ h6 までの6段階があり¹³、それらを使い分けることで記事の文書レベルを表現できる。また、h 要素を含む見出し部を hgroup 要素にまとめることができる。すなわち、hgroup 要素により見出し行(h 要素)とそれに付随する記述を p 要素で追記することができ、見出しに高い表現力を持たせることができる。

記事の記述の中に aside 要素(余談要素)を記述することで、記事の内容に間接的に関連する事柄を挿入することができる。また、記事の内容は必要に応じて section 要素でまとめることができ、記事内容のまとまりを表現(グループ化)することができる。

2.1.3.3 まとめ

以上に解説した要素を使用することで、文書レイアウトや記事の論理構造を表現することができるが、それらのこととは無関係に body 要素の配下には自由に HTML 要素を記述することができ、Web アプリケーションの UI を簡潔な形で実現することも可能である。ただし、文書資産として HTML コンテンツを構築する際には、記事の論理構造などを整然と設計するべき¹⁴である。

2.1.4 HTML 文書内のコメント

HTML 文書内にはコメントを記述することができる。コメントは HTML の文書内容とは見なされない。

書き方: `<!--コメント内容-->`

「コメント内容」は複数行に渡って記述しても良い。ただし、コメントの入れ子はできず、他の HTML 要素内にコメントを記述することもできない。

厳密には、コメントは1つの HTML 要素(コメント要素、コメントノード)である。

2.2 CSS

CSS (Cascading Style Sheets) は、HTML や XML をブラウザで表示する際に体裁の制御を行うためのスタイルシート言語であり、W3C が仕様を取りまとめて勧告している。

基本的に CSS は次のような形式で記述する。

書き方: `セレクタ { 属性1: 値1; 属性2: 値2; ... }`

この形式の記述を CSS の規則集合(ルールセット)という。「セレクタ」はスタイルを設定する対象(HTML 要素など)であり、これが持つ属性1, 属性2,... に対してそれぞれ値1, 値2,... を設定する。またこの場合の「属性:値;」が個々の宣言の記述であり、それらを { } で括った部分を宣言ブロックという。

¹³1つのコンテンツページには h1 要素が1つだけ存在する形式(h2~h6 は必要に応じて複数記述して良い)が推奨されている。

¹⁴セマンティックウェブの考え方。

CSS の規則集合を HTML 文書内に配置するには style 要素を使用する。

書き方： <style> CSS の規則集合 </style>

style 要素は head 要素、body 要素のどちらの子要素にも含めることができる¹⁵。また、CSS の記述をまとめてテキストファイルとして保存しておき、それを HTML 文書閲覧時に読み込んで適用するという使用方法¹⁶もある。このことに関しては後の「2.3.10 外部リソースへのリンク要素」(p.29)で解説する。

2.2.1 セレクタ

CSS のセレクタは、対象とする HTML 要素を指定するものである。具体的には HTML 要素の要素名（要素型セレクタ）、あるいは id 属性の値（ID セレクタ）、class 属性の値（クラスセレクタ）などである。

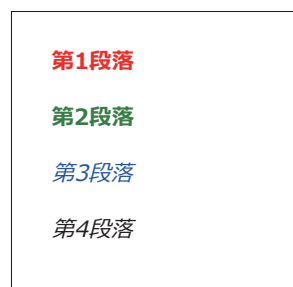
ID セレクタ： #id 値

クラスセレクタ： .class 値

セレクタによって選択的に HTML 要素にスタイルを施す例を次に示す。

記述例：

```
1 <style>
2   p { font-size: 14pt; }
3   #I1 { color: red; }
4   #I2 { color: green; }
5   #I3 { color: blue; }
6   #I4 { color: black; }
7   .C1 { font-weight: bold; }
8   .C2 { font-style: italic; }
9 </style>
10 <p id="I1" class="C1">第1段落</p>
11 <p id="I2" class="C1">第2段落</p>
12 <p id="I3" class="C2">第3段落</p>
13 <p id="I4" class="C2">第4段落</p>
```



左の記述を Web ブラウザで表示した例

記述例の 2 行目のルールセットにより p 要素全てのフォントサイズ (font-size 属性) が 14pt (14 ポイント：p.5 の表 1 参照) に設定される。また、10～13 行目の p 要素に一意の id 名 "I1"～"I4" が与えられており、3～6 行目のルールセットが個別に文字の色 (color 属性) を設定している。

10～11 行目の p 要素には同じ class 名 "C1" が、12～13 行目の p 要素には同じ class 名 "C2" が与えられており、それらにより p 要素がグループ化されている。これにより class 名 "C1" の p 要素の font-weight (強調の度合い) が bold (太字) に、class 名 "C2" の p 要素の font-style (フォントスタイル) が italic (斜体) に設定されている。

対象となる複数のセレクタをコンマ区切りで並べ (セレクタリスト)、それらに共通の CSS 設定を与えることもできる。セレクタの記述方法、CSS の属性やそれらに与える値について具体的な (重要な) ものを順次解説する。

2.2.2 サイズや色の値の表現

HTML 要素の各部のサイズを指定する際の単位の表現を表 1 に示す。

表 1: CSS のサイズ指定における単位 (一部)

単位	解 説	単位	解 説
pt	ポイント (1/72 インチ)	px	ディスプレイの画素
mm	ミリメートル	cm	センチメートル
in	インチ (2.54cm)	pc	パイカ (1/6 インチ=4.233mm)
em	フォントサイズに対する比率	%	基準の値*に対する百分率

* 後の「2.3.3.3 レイアウトと包含ブロック」(p.15)で解説する。

参考) 1px は実際の表示デバイス上の 1 画素であるとは限らない。CSS の仕様では 1px = 0.75pt とされている。ただし実際の表示デバイス上ではこの比率が異なることがあるので注意すること。

¹⁵head 要素内に style 要素を記述することが推奨される。

¹⁶CSS の記述を外部のファイルにしておき、それを HTML 文書内の link 要素によって読み込むことができる。link 要素は head 要素の子要素として記述することが推奨される。

HTML 要素の各部の色を指定する際の表記には様々なものがあるが、特に色名称、16 進カラーコード、rgb 関数記法の 3 種類の表記が多く用いられる。色名称は CSS の仕様として定められたもの¹⁷ がある。

■ 16 進カラーコード

書き方： #赤緑青

「赤」、「緑」、「青」の各部にはそれぞれの色チャンネルの値を 2 桁の 16 進数 (00~ff) で指定する。

■ rgb 関数記法

書き方： rgb(赤, 緑, 青)

「赤」、「緑」、「青」の各部にはそれぞれの色チャンネルの値を 0~255 の範囲の値で指定する。あるいは百分率の形式 (0%~100%) で指定しても良い。また、第 4 の引数としてアルファ値 (不透明度) を 0~1.0 の範囲 (1.0 が不透明) で与えることもできる。

各表記と対応する色の一部を図 1 に示す。

red #ff0000 rgb(255,0,0)	green #00ff00 rgb(0,255,0)	blue #0000ff rgb(0,0,255)	black #000000 rgb(0,0,0)
cyan #00ffff rgb(0,255,255)	magenta #ff00ff rgb(255,0,255)	yellow #ffff00 rgb(255,255,0)	white #ffffff rgb(255,255,255)

図 1: 色と各表記の対応 (一部)

本書で紹介する表記法以外にも多くの表記法がある。また色モデルも RGB 以外に HSL (色相、彩度、明度) など也可以使用。詳しくは W3C の公式インターネットサイトを参照のこと。

2.2.3 CSS のコメント

CSS の記述にコメントを挿入することができる。

書き方： /* コメント */

「コメント」の部分は複数行に渡って記述しても良い。

2.3 HTML, CSS 各論

ここでは HTML 要素と CSS 属性について使用頻度の高いものについて解説する。

2.3.1 見出しと段落：h 要素, hgroup 要素, p 要素

見出し行を実現する h1~h6 要素は、末尾の数値が大きくなる程、表示の際のサイズが小さくなる。段落は p 要素で実現する。例えば次のような HTML (一部) の記述は Web ブラウザでは図 2 のように表示される。

記述例：

```
1 <style>#sp1 { color: red;}</style>
2 <h1>文書構造の例</h1>
3   <p>これは文書の最上位(h1)のレベルの内容です。 </p>
4   <h2>見出し</h2>
5     <p>これは文書の2番目(h2)のレベルの内容 (1つ目) です。 </p>
6     <h3>小見出し</h3>
7       <p>これは文書の3番目(h3)のレベルの内容です。 </p>
8     <h2>見出し</h2>
9       <p>これは文書の2番目(h2)のレベルの内容 (<span id="sp1">2つ目</span>) です。 </p>
```

¹⁷W3C の公式インターネットサイトを参照のこと。

文書構造の例

これは文書の最上位(h1)のレベルの内容です。

見出し

これは文書の2番目(h2)のレベルの内容（1つ目）です。

小見出し

これは文書の3番目(h3)のレベルの内容です。

見出し

これは文書の2番目(h2)のレベルの内容（2つ目）です。

図 2: Web ブラウザによる表示
注) 実際の表示とは若干異なることがある

2.3.1.1 span 要素

上記記述例の 9 行目にある span 要素は、テキストの文字の並びなどの一部を括ることができ、部分的な CSS 設定のために用いられることが多い。

2.3.1.2 hgroup 要素

hgroup 要素は h 要素と p 要素をまとめて見出し部を実現する。例えば次のような HTML（一部）の記述は Web ブラウザでは図 3 のように表示される。

記述例：

```
1 <hgroup>
2   <h1>hgroup要素内のh1レベルの見出し行 </h1>
3   <p>このようにp要素で付加的記述を見出し部に追記できます。 <br>
4     h1要素とp要素で見出し部を構成しています。 </p>
5 </hgroup>
6 <p>上記見出し部に続く段落の記述内容です。 </p>
```

※ 3 行目の
 は改行処理の要素である。

hgroup要素内のh1レベルの見出し行

このようにp要素で付加的記述を見出し部に追記できます。
h1要素とp要素で見出し部を構成しています。

上記見出し部に続く段落の記述内容です。

図 3: Web ブラウザによる表示
注) 実際の表示とは若干異なることがある

2.3.1.3 文字、テキストに関する設定

テキストの表示に関する CSS 属性の一部を表 2 に、フォント（書体）に関する CSS 属性の一部を表 3 に示す。

表 2: テキストの表示に関する CSS 属性（一部）

CSS 属性	解説
letter-spacing	文字同士の間隔
line-height	1 行の高さ
text-indent	段落開始のインデント幅の大きさ
text-align	テキストのアラインメント。left（左寄せ）、right（右寄せ）、center（中央揃え）などが指定できる。
text-decoration	overline（上線）、underline（下線）、line-through（打ち消し線）などを設定する。線種、線幅なども設定可能。

表 3: フォントに関する CSS 属性 (一部)

CSS 属性	解説
font-family	フォントの種類
font-size	フォントの大きさ
font-style	フォントのスタイル. normal (デフォルト), oblique (斜体), italic (筆記体)
font-weight	強調の度合い (太さ). normal (デフォルト), bold (強調), lighter (細め) 100~900 まで 100 刻みの値でも指定できる. (推奨)

Web ブラウザで標準的に使用できるフォントによる表示の例を示す.

記述例:

```

1 <style>
2   p {
3     font-size: 24pt;
4     line-height: 20pt;
5   }
6   #t1 { font-family: serif; }
7   #t2 { font-family: sans-serif; }
8   #t3 { font-family: monospace; }
9   #t4 { font-family: cursive; }
10  #t5 { font-family: system-ui; }
11 </style>
12 <p id="t1">(serif) 01234abcdABCD日本語</p>
13 <p id="t2">(sans-serif) 01234abcdABCD日本語</p>
14 <p id="t3">(monospace) 01234abcdABCD日本語</p>
15 <p id="t4">(cursive) 01234abcdABCD日本語</p>
16 <p id="t5">(system-ui) 01234abcdABCD日本語</p>

```

(serif) 01234abcdABCD日本語
(sans-serif) 01234abcdABCD日本語
(monospace) 01234abcdABCD日本語
(cursive) 01234abcdABCD日本語
(system-ui) 01234abcdABCD日本語

左の記述を Web ブラウザで表示した例

CSS の text-decoration 属性の設定によって、テキストに各種の線で装飾することができる. 各種属性と設定値を表 4 に示す.

表 4: text-decoration に関する属性と設定値 (一部)

属 性	値	解 説
text-decoration-line	none	線なし (デフォルト)
	underline	下線
	overline	上線
	line-through	中央を横切る線
text-decoration-style	solid	実線
	double	二重線
	dotted	点線
	dashed	破線
	wavy	波線
text-decoration-color	色	線の色
text-decoration-thickness	太さ	線の太さ

span 要素内のテキストに下線を施す例を次に示す. 下記のような HTML (一部) の記述は Web ブラウザでは図 4 の様に表示される.

記述例:

```

1 <style>
2   #tx1 { text-decoration-line: underline; }
3   #tx2 { text-decoration-line: overline; }
4   #tx3 { text-decoration-line: line-through; }
5   #tx4 { text-decoration-line: underline overline; }
6   #tx5 { text-decoration: underline wavy red; }
7 </style>
8 <p>これは<span id="tx1">実線の下線による装飾</span>です. </p>
9 <p>これは<span id="tx2">実線の上線による装飾</span>です. </p>

```

```

10 <p>これは<span id="tx3">実線の中線による装飾</span>です。</p>
11 <p>これは<span id="tx4">実線の上線と下線による装飾</span>です。</p>
12 <p>これは<span id="tx5">波線の下線による装飾</span>です。</p>

```

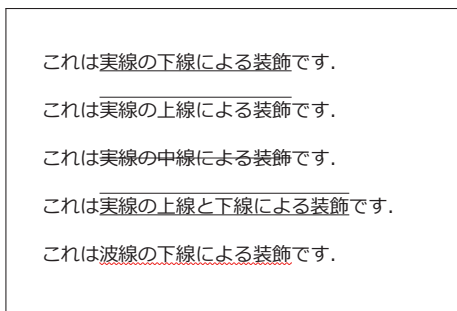


図 4: text-decoration の設定による下線の付与

上記の記述例の 6 行目の様に、text-decoration 属性に複数のものを一括して設定することができる。

2.3.1.4 文字の書体（フォント）に関する設定

ブラウザにテキストを表示する際の書体（フォント）としては、Web ブラウザで標準的に使用されるもの以外に、特定のフォントファイルを読み込んで使用することもできる。

Web デザインにおいて用いられるフォントファイルの形式を表 5 に示す。

表 5: フォントファイルの形式（一部）		
形 式	解 説	ファイル名の拡張子
TrueType (truetype)	多くの計算機環境でテキストを表示する際に使用される。 OS 間で互換性が無いことがある。また、解像度に制限がある場合がある。	ttf, ttc
OpenType (opentype)	多くの計算機環境でテキストを表示する際に使用される。 TrueType を発展させたもので、いくつかの問題が解決されている。	otf, otc, ttf, ttc
woff, woff2.0 (woff, woff2)	Web 専用のフォントである。（Web Open Font Format） TrueType, OpenType を応用して開発されたフォント形式。	woff, woff2

() の内の記述は CSS における形式名

フォントファイルを CSS のフォント関連の属性に関連させるには CSS の @font-face ルール¹⁸ を記述する。

《フォント関連の CSS 属性の登録》

```

@font-face {
  font-family: フォント名;
  src: url( フォントファイルの URL ) format( フォント形式 );
  font-style: フォントスタイル;
  font-weight: 強調の度合い;
}

```

「フォントファイルの URL」から「フォント形式」のフォントデータを読み込み、それを「フォント名」として使用可能にする。また、「フォントスタイル」、「強調の度合い」を登録することもできるが、これらは省略可能である。

フォントによっては、異なるスタイルのものが個別のフォントファイルとして提供されることがある。その場合は同じ font-family の @font-face ルールをフォントファイルの数だけ記述して、それぞれのフォントファイルに対する src の値、font-style の値（normal, bold, lighter など）を記述する。

以下に示す例は、Google Fonts サービス¹⁹ から入手した Dela Gothic One というフォントを用いてテキストを表示するものである。

¹⁸アットルールと呼ばれるものの 1 つ。

¹⁹<https://fonts.google.com/>

記述例：

```
1 <style>
2   @font-face {
3     font-family: "DelaGothicOne";
4     src: url("../fonts/DelaGothicOne-Regular.ttf") format("truetype");
5   }
6   p {
7     font-family: DelaGothicOne;
8     font-size: 24pt;
9   }
10 </style>
11 <p>フォント 書体 Font</p>
```

この例では、カレントディレクトリの fonts フォルダにあるフォントファイル DelaGothicOne-Regular.ttf を "truetype" の形式で読み込んで（4 行目）おり、それを CSS の font-family 属性 "DelaGothicOne" として使用可能に（3 行目）している。これを Web ブラウザで表示すると図 5 の様になる。

フォント 書体 Font

図 5: フォント Dela Gothic One によるテキストの表示

記述例：

```
1 <style>
2   @font-face {
3     font-family: "NotoSansJP";
4     src: url("../fonts/NotoSansJP-Light.otf") format("opentype");
5     font-weight: 300;
6   }
7   @font-face {
8     font-family: "NotoSansJP";
9     src: url("../fonts/NotoSansJP-Regular.otf") format("opentype");
10    font-weight: 400;
11  }
12  @font-face {
13    font-family: "NotoSansJP";
14    src: url("../fonts/NotoSansJP-Bold.otf") format("opentype");
15    font-weight: 700;
16  }
17  p { margin: 0pt; }
18  #p1 {
19    font-family: NotoSansJP;
20    font-weight: 300;
21    font-size: 24pt;
22  }
23  #p2 {
24    font-family: NotoSansJP;
25    font-weight: 400;
26    font-size: 24pt;
27  }
28  #p3 {
29    font-family: NotoSansJP;
30    font-weight: 700;
31    font-size: 24pt;
32  }
33 </style>
34 <p id="p1">フォント 書体 Font (weight:300) </p>
35 <p id="p2">フォント 書体 Font (weight:400) </p>
36 <p id="p3">フォント 書体 Font (weight:700) </p>
```

この例では、カレントディレクトリの fonts フォルダにある 3 つのフォントファイル（Google Fonts サービスから入手）

NotoSansJP-Light.otf, NotoSansJP-Regular.otf, NotoSansJP-Bold.otf

を "opentype" の形式で読み込んで（2～16 行目）おり、それらを CSS の同一の font-family 属性 "NotoSansJP" として使用可能にしている。これは、1 つのフォントシリーズの重みの異なるフォントを font-weight の値で区別する例である。これを Web ブラウザで表示すると図 6 の様になる。

フォント 書体 Font (weight:300)
フォント 書体 Font (weight:400)
フォント 書体 Font (weight:700)

図 6: フォント NotoSansJP によるテキストの表示

2.3.1.5 ルビ

テキストにルビを付けるには ruby 要素を記述する。

書き方 (1): `<ruby>対象のテキスト<rt>ルビ</rt></ruby>`

「対象のテキスト」に「ルビ」を付ける。また、ルビのレイアウトに対応していない Web ブラウザでの表示のために次のような記述をすると良い。

書き方 (2): `<ruby>対象のテキスト<rp>(</rp><rt>ルビ</rt><rp>)</rp></ruby>`

この様に rp 要素を rt 要素の前後に置くことで、ルビのレイアウトに対応していない Web ブラウザでの表示において、ルビの部分を括弧で括った形で表示することができる。

テキストにルビを付ける例を次に示す。

記述例:

```
1 <style>
2   p { font-size: 24pt; }
3 </style>
4 <p>それは<ruby>言語道断<rp>(</rp><rt>ごんごどうだん</rt><rp>)</rp></ruby>です。 </p>
```

これを Web ブラウザで表示すると図 7 の (a) の様になる。

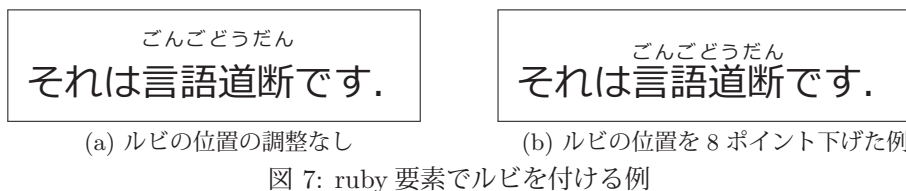


図 7: ruby 要素でルビを付ける例

ルビの位置の上下を調整する例を次に示す。

記述例:

```
1 <style>
2   p { font-size: 24pt; }
3   rt {
4     position: relative;
5     top: 8pt;
6   }
7 </style>
8 <p>それは<ruby>言語道断<rp>(</rp><rt>ごんごどうだん</rt><rp>)</rp></ruby>です。 </p>
```

これを Web ブラウザで表示すると図 7 の (b) の様になる。

2.3.2 ハイパーリンクの設置

HTML の主要な機能の 1 つがハイパーリンクであり、HTML 文書中の指定した部分に他の HTML 文書などのリソースを接続することができる。これには a 要素 (アンカー要素) を使用する。

書き方: `コンテンツ`

「コンテンツ」の部分に「接続先の URL」へのハイパーリンクを設置する。

例. WHATWG のホームページへのハイパーリンク

`<p>これはWHATWG へのハイパーリンクの例です。 </p>`

これは p 要素の内容の一部に WHATWG のホームページへのハイパーリンクを設置する記述例であり、これを Web ブラウザで表示すると次の様に表示される。

これは[WHATWG へのハイパーリンク](https://whatwg.org/)の例です。

このような表示に対して、青い下線部分をポインティングデバイス（マウスなど）でクリック（あるいはタップ）するとリンク先の Web コンテンツが表示される。

a 要素の開始タグの中に target 属性を記述することで、リンク先の表示方法が制御できる。（次の例参照）

例. Web ブラウザの新規タブを開いてリンク先を表示する

<p>これはWHATWG へのハイパーリンクの例です. </p>

この様に target 属性に "_blank" を設定すると、リンク先を新規タブに表示する。

標準的には、ハイパーリンクを設置した部分は青い下線部として表示されるが、この表現を CSS の設定で変更することができる。例えば、下線の表示を無効にするには、当該 a 要素の CSS 属性 text-decoration に none を設定する。この text-decoration 属性はテキストに装飾的な線を付加するためのものである。

2.3.2.1 a 要素に対する特殊なセレクタ

a 要素はマウスなどのポインティングデバイスで操作する対象であり、ユーザのマウスの挙動（ホバリング、ボタンの押下など）によって表示の体裁を動的に変更することができる。具体的には、マウスの挙動に対応したセレクタ（表 6）にそれぞれルールセットを記述する。

表 6: マウスの挙動に応じたセレクタ（一部）

セレクタ	解説
a:hover	a 要素の上にマウスが翳されている状態
a:active	a 要素の上でマウスボタンを押下した状態
a:visited	a 要素のリンク先を既に閲覧し終わった状態

セレクタ先頭の a の部分は #id 値 を指定しても良い。

表 6 にあるセレクタのコロン「:」以下の部分は擬似クラスと呼ばれるもので、対象の HTML 要素の状態を指定するものである。

例えば次のような HTML（一部）の記述について考える。

記述例：

```
1 <style>
2   a {text-decoration: none;}
3   a:hover {color: green;}
4   a:active {color: red;}
5   a:visited {color: gray;}
6 </style>
7 <p>これは<a href="https://whatwg.org/">WHATWG へのハイパーリンク </a>の例です. </p>
```

これを Web ブラウザで表示した際のマウス操作に対する反応は次の様になる。

状 態	表 示
閲覧前	→ これは WHATWG へのハイパーリンク の例です.
リンク部にマウスを翳している間	→ これは WHATWG へのハイパーリンク の例です.
リンク部でマウスボタンを押している間	→ これは WHATWG へのハイパーリンク の例です.
閲覧後	→ これは WHATWG へのハイパーリンク の例です.

2.3.3 HTML 要素のレイアウト

Web ブラウザは記述された HTML 要素を通常フロー（normal flow）と呼ばれる制御でレイアウトして表示する。通常フローの制御下では、これまでに解説してきた h 要素や p 要素は、その前後で改行が施されて垂直方向（下方

向)に順次表示される。このような形式で表示されるものを**ブロックレイアウト**の要素であるという。また、先に解説した `span` 要素や `a` 要素などはテキストの中に配置して、テキストの並びの中(水平方向:右方向)に表示される。このような形式で表示されるものを**インラインレイアウト**の要素であるという。HTML 文書の中の改行コードは、Web ブラウザでの表示においては基本的には無視され、上のようなレイアウトの制御に従って表示される。表示の際に改行を強制する場合は `br` 要素を記述する。

HTML の古い仕様では各種の HTML 要素を「インラインレベル要素」と「ブロックレベル要素」と呼ばれるカテゴリに分類しており²⁰、現在でもそのことに由来した性質が CSS の `display` 属性として残っている。`display` に `block`、`inline` などの値を設定することで対象となる HTML 要素のレイアウトを変更できる。

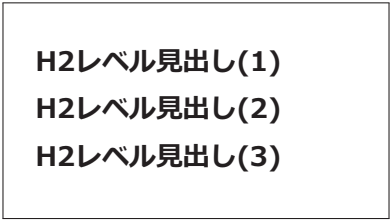
■ **ブロックレイアウトの要素をインラインレイアウトにする例**

`h2` 要素はブロックレイアウトであり、次のような記述例の内容を Web ブラウザで表示すると右側のような表示となる。

記述例：

```
1 <h2>H2レベル見出し(1)</h2>
2 <h2>H2レベル見出し(2)</h2>
3 <h2>H2レベル見出し(3)</h2>
```

注) `h2` 要素の CSS 属性 `display` は変更していないものとする。



左の記述を Web ブラウザで表示した例

次に、先の記述例に「`<style> h2 { display:inline; } </style>`」という要素を加えて `h2` 要素の CSS 属性 `display` を `inline` に変更した場合の表示例を図 8 に示す。



図 8: `h2` 要素の `display` 属性 (CSS) を `inline` に変更した際の表示

■ **インラインレイアウトの要素をブロックレイアウトにする例**

`span` 要素はインラインレイアウトであり、次のような記述例の内容を Web ブラウザで表示する場合について考える。

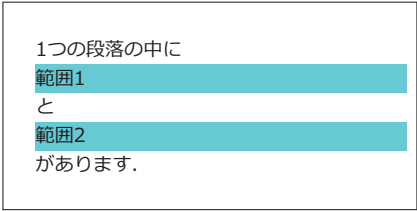
記述例：

```
1 <style>span {background-color: cyan;}</style>
2 <p>1つの段落の中に<span>範囲1</span>と<span>範囲2</span>があります。</p>
```

`span` 要素の CSS 属性 `display` を変更せずに Web ブラウザで表示すると、

1つの段落の中に **範囲1** と **範囲2** があります。

と表示される。次に「`<style>span {display:block;}</style>`」という要素を加えて `span` 要素の CSS 属性 `display` を `block` に変更した場合は次のような表示となる。



`span` 要素の部分がブロックレイアウトとなり、前後に改行が施されている。またこの部分の CSS 属性 `background-color` (背景色) を `cyan` にしているが、デフォルトではブロックレイアウトの横幅 (CSS の `width` 属性) は表示領域と同

²⁰このような分類は HTML5 で廃止され現在ではより多彩なカテゴリに分類されているが、「インラインレベル要素」、「ブロックレベル要素」という言葉は現在でも便宜上使用されることが多い。

じである。

CSS の display 属性には他にも設定できるレイアウト用の値がある。詳しくは W3C をはじめとする公式の情報源を参照のこと。

2.3.3.1 HTML 要素をレイアウトするためのボックス構造

Web ブラウザが HTML 文書をレイアウトする際、個々の要素のために図 9 のようなボックスを生成する。

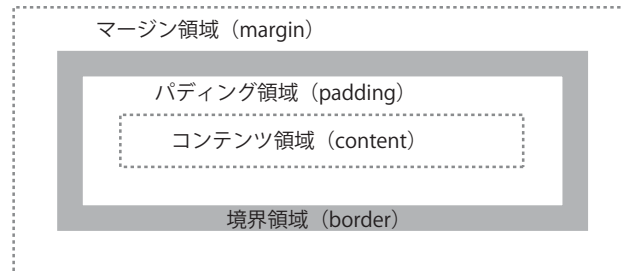


図 9: HTML 要素をレイアウトするためのボックス構造

このボックスは、**コンテンツ領域**、**パディング領域**、**境界領域**、**マージン領域**の4つの領域から成る。HTML 文書の各要素が持つコンテンツはコンテンツ領域にレイアウトされ、その外側をパディング領域が覆う。更にその外側を境界領域が覆っており、これを枠線として表示することが可能である。更にその外側をマージン領域が覆っている。

外観としてこのボックスは、コンテンツを枠で囲んで表示するものと見ることができ、枠の外側の余白（マージン領域）と内側の余白（パディング領域）があると考えると理解しやすい。

以上のことを具体的に確かめる例として次の Container01.html を示す。

ファイル：Container01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>Container</title>
6     <style>
7       body {
8         width: 400pt;
9         padding: 10pt;
10        border: solid 1pt black;
11      }
12      section {
13        margin-top: 10pt;
14        padding: 10pt;
15        border: solid 1pt black;
16        background-color: #e0e0e0;
17      }
18      p {
19        padding: 10pt;
20        border: solid 1pt black;
21        background-color: lightyellow;
22      }
23      span {
24        padding: 5pt;
25        border: solid 1pt black;
26        background-color: lightpink;
27      }
28    </style>
29  </head>
30  <body>
31    body要素
32    <section>
33      section要素
34      <p>これはsection内の段落です。 <span>インラインの要素</span>を含みます
35        . </p>
36    </section>
37  </body>
</html>
```

これを Web ブラウザで表示した例を図 10 に示す。

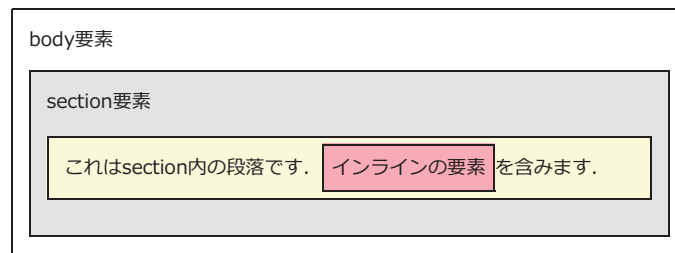


図 10: Web ブラウザによる表示の例

body, section, p, span の各要素が枠で囲まれている様子がわかる。

コンテンツ領域の横幅は CSS 属性 width に、高さは height に設定する。

マージン領域のサイズは CSS 属性 margin に設定する。またこの際、margin-top, margin-bottom, margin-left, margin-right, のように margin の直後に **-部分**の接尾辞を付けることで、マージンの上下左右の各部を個別に設定することができる。

パディング領域のサイズは CSS 属性 padding に設定する。また margin の場合と同様の方法で上下左右の各部のパディングを設定することができる。

境界領域の設定は CSS 属性 border に対して行う。更に border には接尾辞 -style, -width, -color を付けて、それぞれ線種、太さ、色を設定することができる。border-style に設定する値と枠の外観を図 11 に示す。

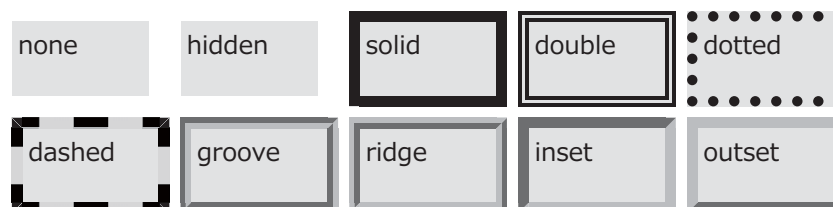


図 11: border-style に設定する値と枠の外観

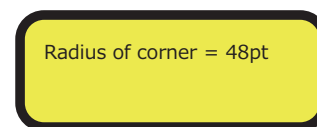
2.3.3.2 div 要素

文書構造としての意味を持たない HTML 要素として div 要素があり、これは、その内部に HTML 要素を配置するためのコンテナである。div 要素はブロックレイアウトの要素であり、先に解説したボックス構造を実現できる。

div 要素に関する記述例を次に示す。

記述例：

```
1 <style>
2   div {
3     width: 160pt;
4     height: 40pt;
5     padding: 12pt;
6     border: solid 8px black;
7     border-radius: 24px;
8     background-color: yellow;
9   }
10 </style>
11 <div>Radius of corner = 48pt</div>
```



左の記述を Web ブラウザで表示した例

この記述例の 6 行目のように border を一括設定することができる。また 7 行目にある border-radius 属性はボックスの角の半径を設定するものである。

2.3.3.3 レイアウトと包含ブロック

HTML 要素のレイアウトは、それを含む**包含ブロック**に対して行われる。すなわち、「HTML 要素はそれを含んでいる上位のブロックレイアウト要素の内部でレイアウトされる」ということである。ここで注意すべきこととして、包含ブロックは単なる親要素を意味しないことがある。このことに関して例 Container02.html を示して解説する。

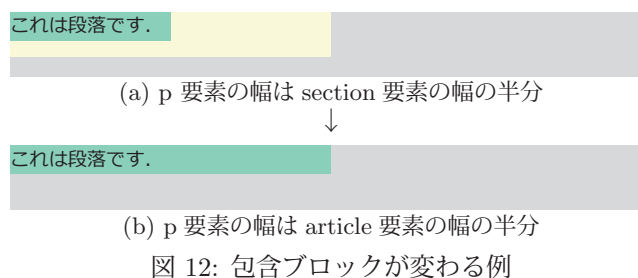
```

1  <!DOCTYPE html>
2  <html lang="ja">
3    <head>
4      <meta charset="utf-8">
5      <title>Container</title>
6      <style>
7        article {
8          display: block;
9          width: 400pt;
10         height: 40pt;
11         background-color: lightgray;
12       }
13       section {
14         display: block;
15         width: 50%;
16         height: 28pt;
17         background-color: lightyellow;
18       }
19       p {
20         width: 50%;
21         background-color: aquamarine;
22       }
23     </style>
24   </head>
25   <body>
26     <article>
27       <section>
28         <p>これは段落です. </p>
29       </section>
30     </article>
31   </body>
32 </html>

```

この例では、article → section → p という要素の階層構造となっており、section 要素の幅は article 要素の幅の半分 (50%)、p 要素の幅は section 要素の幅の半分 (50%) (図 12 の (a)) となる。これは、各要素の階層関係がそのまま包含ブロックの階層関係となっていることによる。

次に、上記ファイルの 14 行目にある display 属性の設定を inline に変更して表示した例を図 12 の (b) に示す。



section 要素の display 属性を inline にするとブロックレイアウトの要素ではなくなるので、p 要素の包含ブロックではなくなる。従って、p 要素の包含ブロックは section 要素の上位の article 要素となり、p 要素の幅は article 要素の幅の半分になる。

2.3.3.4 要素の位置について

HTML 要素の表示のレイアウトは、Web ブラウザが適切に決定するが、CSS の position 属性の設定によって変更することができる。position 属性のデフォルト値は static であるが、それ以外の値に設定すると表 7 のような CSS 属性によって上下左右の位置を指定することができる。

表 7: HTML 要素の位置を決める CSS 属性

属 性	解 説	属 性	解 説	属 性	解 説	属 性	解 説
top	上端の位置	bottom	下端の位置	left	左端の位置	right	右端の位置

position 属性の設定値ごとに表 7 の属性の意味合いが変わる。(表 8 参照)

表 8: position 要素の値に対するレイアウトの意味合い

値	解 説
static	top, bottom, left, right の設定は無効
absolute	当該 HTML 要素の包含ブロックからの相対位置
relative	display 属性が static である場合のレイアウトからの相対位置
fixed	ビューポートによって定められた初期の包含ブロックからの位置
sticky	直近のスクロールする祖先および包含ブロックからの位置 ただし、表示範囲によっては fixed と同じ効果がある。

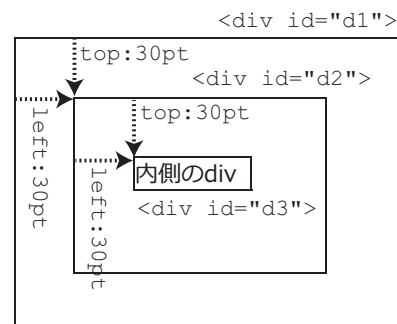
position の値が absolute である場合を例に挙げてレイアウトの様子を示す。

記述例：CSS

```
1 <style>
2   div {
3     position: absolute;
4     border: solid 1pt black;
5   }
6   #d1 {
7     width: 200pt;
8     height: 150pt;
9   }
10  #d2 {
11    top: 30pt;
12    left: 30pt;
13    width: 130pt;
14    height: 90pt;
15  }
16  #d3 {
17    top: 30pt;
18    left: 30pt;
19    width: 60pt;
20    height: 16pt;
21  }
22 </style>
```

記述例：HTML

```
1 <div id="d1">
2   <div id="d2">
3     <div id="d3">
4       内側のdiv
5     </div>
6   </div>
7 </div>
```



Web ブラウザで表示した例

div 要素が包含ブロックに対して、CSS の top, left の値によって指定された位置にレイアウトされていることがわかる。

次に、包含ブロックに対して CSS の bottom, right の値によって div 要素をレイアウトする例を示す。

記述例：CSS

```
1 <style>
2   div {
3     position: absolute;
4     border: solid 1pt black;
5   }
6   #d1 {
7     width: 160pt;
8     height: 80pt;
9   }
10  #d2 {
11    bottom: 20pt;
12    right: 30pt;
13  }
14 </style>
```

記述例：HTML

```
1 <div id="d1">
2   <div id="d2">
3     内側のdiv
4   </div>
5 </div>
```



Web ブラウザで表示した例

2.3.3.5 ボックス構造の位置とサイズに関すること

ボックスレイアウトの HTML 要素の各部と CSS 属性との対応を図 13 に示す。

幅と高さを指定する CSS 属性は width と height であるが、要素全体のサイズは padding, border, margin の大きさを加えたものとなる。また top, bottom, left, right による位置の設定は margin の外周に対するものとなる。

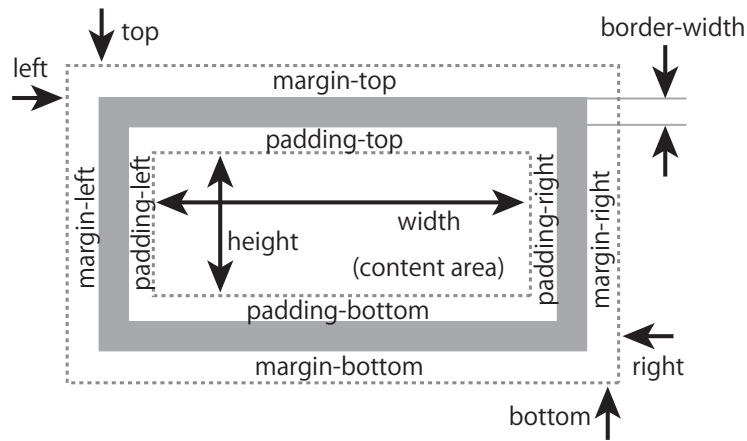


図 13: ボックスレイアウトの要素の各部

ボックスレイアウトの HTML 要素の CSS 属性 `box-sizing` に `border-box` を設定すると図 14 のように `width` と `height` の解釈が変わる. (`box-sizing` のデフォルト値は `content-box` である)

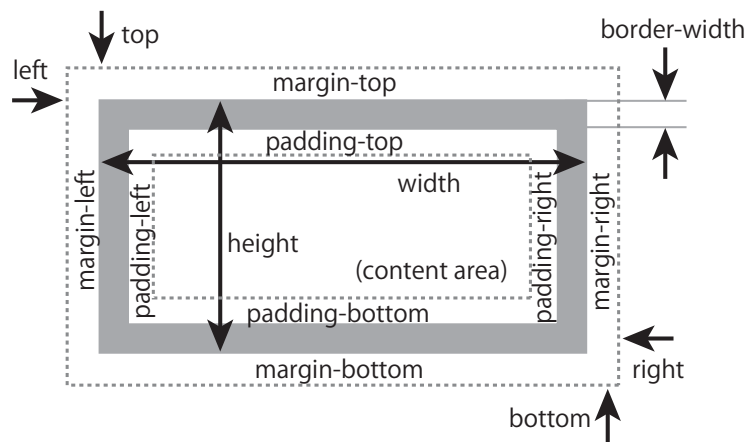


図 14: `box-sizing` を `border-box` にした場合の各部

`box-sizing` の設定によって `div` 要素のサイズが変わる様子を例示する. 次に示す CSS の記述例では, 2~11 行目の設定を 2 つの `div` 要素に共通して施しているが, それぞれの `div` 要素には異なる `box-sizing` の値を与えている.

記述例: CSS

```

1 <style>
2   div {
3     position: absolute;
4     font-size: 22pt;
5     border: solid 20pt black;
6     background-color: yellow;
7     width: 110pt;
8     height: 90pt;
9     padding: 10pt;
10    margin: 10pt;
11  }
12  #d1 {
13    top: 0pt;
14    left: 0pt;
15    box-sizing: content-box;
16  }
17  #d2 {
18    top: 160pt;
19    left: 0pt;
20    box-sizing: border-box;
21  }
22 </style>

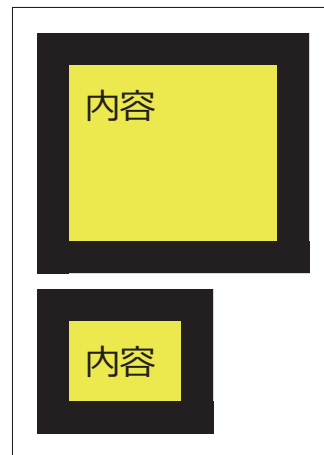
```

記述例: HTML

```

1 <div id="d1">内容</div>
2 <div id="d2">内容</div>

```



Web ブラウザで表示した例

結果として表示された `div` 要素のサイズが異なることがわかる.

実際に Web コンテンツや UI を構築する際には, レイアウトの方針によって `box-sizing` の設定を選択すると良い.

2.3.3.6 コンテナ内外のレイアウト制御

div を始めとするコンテナ要素は内部のレイアウト制御を CSS の display 属性で設定することができる。

書き方： display: 外側のレイアウト 内側のレイアウト;

「外側のレイアウト」は当該コンテナ要素自身のレイアウトを指定するもので、先に解説したように block, inline などが設定できる。「内側のレイアウト」には表 9 に示すようなものを使用できる。

表 9: コンテナの内側のレイアウト (一部)

レイアウト	解説
flow	通常フロー (normal flow)
flex	要素を水平もしくは垂直の一次元に配置する。
grid	グリッドに沿った形で要素を配置する。

■ flex レイアウトの応用例

一次元配置における表示位置 (始点, 中央, 終点) を CSS の align-items 属性で設定する例を示す。

記述例:

```
1 <style>
2   div {
3     display: inline flex;
4     height: 50pt;
5     border: solid 4pt black;
6     align-items: start;
7   }
8 </style>
9 <div>div要素</div>
```



左の記述を Web ブラウザで表示した例

この例は div 要素内でのコンテンツの上下位置の設定に応用できる。

■ インラインレイアウトにおける位置関係

インラインレイアウトにおける垂直の位置を CSS の vertical-align 要素で指定することができる。次に示す例は, div 要素の垂直位置を指定するものである。

記述例:

```
1 <style>
2   p { display: inline; }
3   div {
4     display: inline flex;
5     height: 50pt;
6     border: solid 4pt black;
7     vertical-align: middle;
8     align-items: center;
9   }
10 </style>
11 <p>インラインの</p><div>div要素</div><p>です。</p>
```

7 行目の vertical-align の設定値を様々に変えて表示した例を図 15 に示す。



図 15: Web ブラウザで表示した例

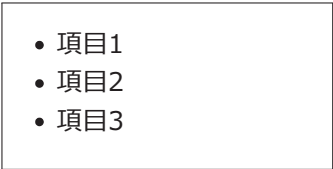
2.3.4 箇条書き: ul 要素, ol 要素, li 要素

順序なしの箇条書きは ul 要素で, 順序付きの箇条書きは ol 要素で記述する。箇条書きの各項目要素はそれらの子要素として li 要素で記述する。例えば次のような HTML (一部) の記述は Web ブラウザでは図 16 のように表示さ

れる。

記述例：順序なし箇条書き

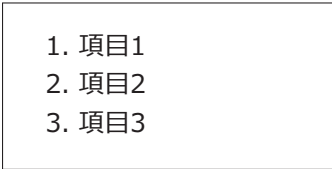
```
1 <ul>
2   <li>項目1</li>
3   <li>項目2</li>
4   <li>項目3</li>
5 </ul>
```



(a) 順序なし

記述例：順序つき箇条書き

```
1 <ol>
2   <li>項目1</li>
3   <li>項目2</li>
4   <li>項目3</li>
5 </ol>
```



(b) 順序つき

図 16: 箇条書き

このように、li 要素に**マーカー**（先頭の記号）が付く形式の箇条書きが表示される。マーカーの種類は CSS の list-style-type 属性に指定することができる。（表 10）

表 10: 箇条書きのマーカー（一部）

値	マーカー	値	マーカー	値	マーカー	値	マーカー
none	（なし）	disc	●	square	■	circle	○
decimal	1,2,3...	lower-alpha	a,b,c...	upper-alpha	A,B,C...	lower-greek	α, β, γ ...
lower-roman	i, ii, iii...	upper-roman	I, II, III				

2.3.5 表：table 要素

table 要素は表を構成するものであり、UI 構築の際にも各種の UI 要素のレイアウトに応用できる。table 要素を構成するための内部の要素には多様なものがあるが、ここでは基本的なものに限定して解説する。

table 要素の子要素にはキャプションを挿入するための caption 要素があるがこれは省略可能である。表は複数の行（tr 要素）から構成され、各行は複数の列項目（td 要素）から構成される。

table 要素で表を構成する例を次の Table01.html に示す。

ファイル：Table01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>Table01</title>
6   <style>
7     table {
8       background-color: gray;
9       border-spacing: 3pt;
10    }
11    th { background-color: lightgreen; }
12    td { background-color: lightgoldenrodyellow; }
13  </style>
14 </head>
15 <body>
16   <table>
17     <caption>キャプション</caption>
18     <tr><th></th><th>列1</th><th>列2</th><th>列3</th></tr>
19     <tr><th>行1</th><td>セル1-1</td><td>セル1-2</td><td>セル1-3</td></tr>
20     <tr><th>行2</th><td>セル2-1</td><td>セル2-2</td><td>セル2-3</td></tr>
21   </table>
22 </body>
23 </html>
```

9 行目で table 要素の CSS 属性 border-spacing の設定があるが、これは、表の各セルの間の距離を指定するものである。これを 3pt として表示した例が図 17 の (a)、0pt として表示した例が (b) である。

キャプション

	列1	列2	列3
行1	セル1-1	セル1-2	セル1-3
行2	セル2-1	セル2-2	セル2-3

(a) border-spacing が 3pt の場合

キャプション

	列1	列2	列3
行1	セル1-1セル1-2セル1-3		
行2	セル2-1セル2-2セル2-3		

(b) border-spacing が 0pt の場合

図 17: table 要素による表の作成

table, th, td 要素には CSS の border などの設定ができる。

2.3.5.1 thead, tbody, ifoot 要素

table 要素の内容は thead, tbody, ifoot 要素によって、それぞれヘッダー部、本体部、フッター部としてまとめることができる。それによって、各部毎に CSS の設定を個別に行うことができる。

2.3.6 画像の埋め込み：img 要素

img 要素は画像オブジェクトをインラインレイアウトで配置する。

書き方：

「画像のパスや URI」で指定された画像データを読み込んで配置する。その際、CSS 属性の width, height でサイズを指定できる。「テキスト」には当該画像に関する簡潔な説明を記述するが、alt= の部分は省略可能である。

img 要素によってコンテンツに画像を挿入する例を ImgElement01.html に示す。

ファイル：ImgElement01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>ImgElement</title>
6     <style>img { width:48pt; }</style>
7   </head>
8   <body>
9     <p>テキストの途中にを配置する例です。</p>
10  </body>
11 </html>
```

9 行目の img 要素で図 18 のような内容の画像ファイル IMGobj01.png を挿入する。（画像ファイルはコンテンツと同じディレクトリにあるものとする）



図 18: 画像ファイル IMGobj01.png

上記 ImgElement01.html を Web ブラウザで表示した例を図 19 に示す。



図 19: Web ブラウザで表示した例

ファイル ImgElement01.html の 6 行目の CSS の記述により、画像の表示サイズが横幅 48pt になっている。

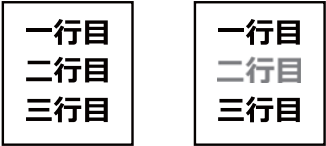
2.3.7 不透明度

HTML 要素には**不透明度**を設定する CSS 属性 `opacity` があり、0（透明）～1.0（不透明）の範囲で値を設定する。`opacity` の設定の例を次の `OpacityVisivility01.html` を用いて示す。

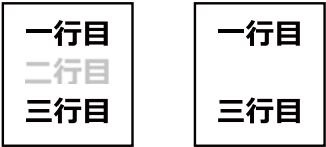
ファイル：OpacityVisivility01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>OpacityVisivility01</title>
6     <style>
7       p {
8         line-height: 14pt;
9         margin: 2pt;
10        font-size: 12pt;
11        font-weight: bold;
12      }
13      #p2 {
14        opacity: 1.0;
15        visibility: visible;
16      }
17    </style>
18  </head>
19  <body>
20    <p id="p1">一行目</p>
21    <p id="p2">二行目</p>
22    <p id="p3">三行目</p>
23  </body>
24 </html>
```

これを Web ブラウザで表示すると下図の (a) ようになる。



(a) opacity: 1.0 (b) opacity: 0.5



(c) opacity: 0.2 (d) opacity: 0.0

右の例では「二行目」を表示する p 要素（id="p2"）の不透明度を 14 行目の記述で変更する。その値を変えて表示したものが (b)～(d) の図である。

2.3.8 可視属性

HTML 要素には不透明度よりも優先する**可視属性**がある。これは CSS の `visibility` 属性で、これを `hidden` に設定すると当該 HTML 要素が表示されなくなるので、先の例 `OpacityVisivility01.html` の 15 行目を変更して試されたい。デフォルト値は `visible`（可視）である。

2.3.9 form 要素

form 要素は、ユーザが Web ページに対して入力したデータを Web サーバ側のプログラムに送信するためのものである。form 要素は UI（ユーザインターフェース）を構成するための各種の要素（表 11）を配下に持つ。

表 11: form 要素配下の UI 要素

要素	解 説
input	type 属性の値によって様々な UI 要素となる。
textarea	複数行のテキストを入力するための領域
select	複数の項目を選択するためのリスト
button	ボタン要素。テキスト以外のものもボタンにできる。

form 要素に入力された内容は

`<input type="submit" name="名称" value="ボタントップの表示">`

のような UI 要素（送信ボタン）の操作によって Web サーバに送信される。これは GUI²¹ のボタンであり、`value` に指定した内容をそのボタントップに表示する。また「名称」は任意の（一意の）テキストを指定できる。

例えば `value="以上の内容を送信します"` のようにすると当該要素は Web ブラウザでは図 20 のようなボタンとして表示される。
上記の方法とは別に `button` 要素でも送信ボタンを実現できる。

書き方： `<button name="名称" value="値">コンテンツ</button>`

²¹Graphical User Interface

図 20: 送信ボタン

form 要素配下の UI 要素には name 属性の値を与えおき、ユーザの操作による入力値は value 属性に得られる。そして form 要素の内容は各 UI 要素の name 属性の値と value 属性の値をペアにした形のテキストデータ（下記）として Web サーバに送信される。

2.3.9.1 form 要素の記述形式と Web サーバとの連携

クライアント側からの送信データのサイズが十分に小さい場合は GET メソッドでも良いが、POST メソッドで送信することが推奨される。

form 要素配下で使用する UI 要素は form 要素の外部でも使用することができる。すなわち、Web サーバにデータを送信せずに、ローカルの計算機環境における UI 要素として使用することができる。

label 要素は必須のものではない.

左の記述を Web ブラウザで表示した例

23

例えば、上の例において「入力欄 1」のフィールドに "abc", 「入力欄 2」のフィールドに "xyz" と入力して Web サーバに送信すると

```
"tx1=abc&tx2=xyz"
```

という文字列がサーバ側プログラムに渡される。

2.3.9.5 テキスト入力 (2)：複数行の入力

textarea 要素は複数行のテキストを入力、編集するための領域を実現する。

書き方： `<textarea name="名称">初期テキスト</textarea>`

入力されたテキスト（文字列）がこの要素の value 属性の値となる。「初期テキスト」を予め与えておくことができる。この要素はインラインレイアウトである。

例. textarea

```
<p>テキスト編集<textarea name="ta1">初期テキスト</textarea></p>
```

これを Web ブラウザで表示すると図 21 のようになる。

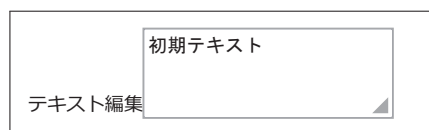


図 21: テキスト編集領域

2.3.9.6 パスワード入力

input 要素に属性 type="password" を設定することでパスワード入力フィールドとなる。

書き方： `<input type="password" name="名称">`

入力されたパスワード（文字列）がこの要素の value 属性の値となる。

例. パスワード入力フィールド

```
<p>パスワード:<input type="password" name="名称"></p>
```

これを Web ブラウザで表示すると図 22 のようになる。

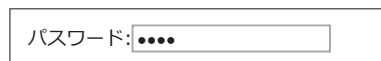


図 22: パスワードフィールド

文字を入力するとダミー文字列が表示される。

2.3.9.7 ボタン

input 要素に属性 type="button" を設定することでボタンとなる。

書き方： `<input type="button" name="名称" value="ボタントップの文字">`

この形式で実現したボタンは送信ボタンではなく、入力操作（クリック、タップ）をしても form 要素の内容は Web サーバに送信されない。この形式のボタンは JavaScript などによるクライアント側の処理に主として用いられる。

例. ボタン

```
<p>クリックしてください→<input type="button" name="b1" value="ボタン"></p>
```

これを Web ブラウザで表示すると図 23 のようになる。

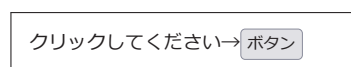


図 23: ボタン

2.3.9.8 スライダー

input 要素に属性 type="range" を設定することでスライダーとなる。

書き方： `<input type="range" min="最小値" max="最大値" step="刻み幅" name="名称">`

min, max, step の記述は省略可能で、デフォルトでは min=0, max=100, step=1 である。スライダーのつまみの位

置がこの要素の value 属性の値となる。

例. スライダー

`<input type="range" name="s1">` これを Web ブラウザで表示すると図 24 のようになる。



図 24: スライダー

■ スライダーに目盛りを表示する方法

input 要素に list 属性を与えることで、スライダーに目盛りを表示することができる。このとき、datalist 要素（後述）で目盛り位置の値のリストを作成しておいたものを label 属性に設定する方法がある。これについて例を挙げて解説する。

記述例：

```
1 <input type="range" name="s2" list="tick">
2 <datalist id="tick">
3   <option value="0"></option>
4   <option value="25"></option>
5   <option value="50"></option>
6   <option value="75"></option>
7   <option value="100"></option>
8 </datalist>
```



左の記述を Web ブラウザで表示した例

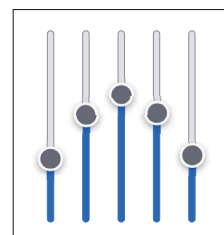
記述例の 2～8 行目に datalist 要素²³ が定義されており、値 0,25,50,75,100 のリストができている。この要素の id 値を input 要素の label 属性に与えている。

■ スライダーを垂直方向にする方法

input 要素に属性 orient="vertical" を与えると、スライダーが垂直方向になる。（次の例参照）

記述例：

```
1 <input type="range" orient="vertical" name="s11">
2 <input type="range" orient="vertical" name="s12">
3 <input type="range" orient="vertical" name="s13">
4 <input type="range" orient="vertical" name="s14">
5 <input type="range" orient="vertical" name="s15">
```



これを Web ブラウザで表示すると右図のようになる。

2.3.9.9 データリスト要素

datalist 要素はデータとしての値のリストを定義する。この要素は本来 UI としてレイアウトするためのものではなく、他の HTML 要素に値の集合を提供することを主な目的とする。

書き方： `<datalist id="一意名">`
 `<option value="値 1">コンテンツ 1</option>`
 `<option value="値 2">コンテンツ 2</option>`
 `</datalist>`

このように記述した datalist 要素の id 値（上記「一意名」の部分）を input 要素の list 属性に与えると、その input 要素の入力操作において、項目選択の入力が可能になる。例えば、次のような記述を Web ブラウザで表示して入力作業を行った例を図 25 に示す。

記述例：

```
1 <datalist id="dlst">
2   <option value="dog">犬</option>
3   <option value="cat">猫</option>
4   <option value="bird">鳥</option>
5   <option value="turtle">亀</option>
6 </datalist>
```

²³ 「2.3.9.9 データリスト要素」（p.25）で解説する。

```
7 <input type="text" list="dlst">
```

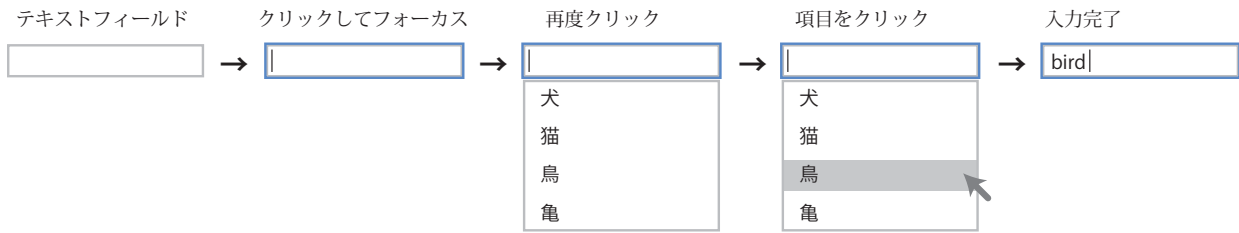


図 25: datalist をテキストフィールドに応用した例

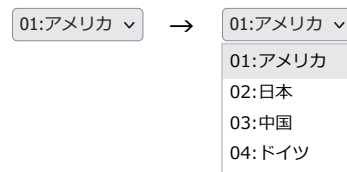
このように、option 要素のコンテンツが選択肢として表示され、選択すると value 属性の値が入力値となる。

2.3.9.10 選択要素

select 要素は、定義された値のリストから必要なものを選択するための UI を実現する。

記述例：

```
1 <select>
2   <option value="1">01:アメリカ</option>
3   <option value="2">02:日本</option>
4   <option value="3">03:中国</option>
5   <option value="4">04:ドイツ</option>
6 </select>
```



クリックによって選択肢が現れる。

これを Web ブラウザで表示すると右図のようになる。

select 要素の子要素には datalist の子要素と同様のものを与える。選択肢の個々の項目を option 要素として記述する。UI には option のコンテンツが表示され、選択結果はそれに対応する value の値となる。

select 要素の開始タグの中に multiple を記述すると、複数項目を同時に選択する²⁴ ことができる。(図 26)

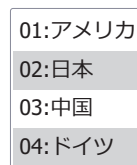


図 26: 複数項目の同時選択の例

select 要素配下の option 要素の開始タグに selected を記述²⁵ すると、その選択肢を予め選択した状態にできる。

2.3.9.11 チェックボックス

input 要素に属性 type="checkbox" を設定することでチェックボックスとなる。

書き方： <input type="checkbox" name="名称" value="値">

このタグ内に checked²⁵ を記述すると、予めチェックされた状態となる。当該チェックボックスがチェックされていない状態では Web サーバに「値」は送信されない。また、value 属性を持たないチェックボックスがチェックされていると、当該チェックボックスの値として"on" が Web サーバに送信される。

記述例：

```
1 <input type="checkbox" name="c1" id="c1" checked>
2 <label for="c1">項目1</label><br>
3 <input type="checkbox" name="c2" id="c2">
4 <label for="c2">項目2</label>
```



チェックボックス

これを Web ブラウザで表示すると右図のようになる。

²⁴ Shift キーや Ctrl キーなどを押しながらクリックする。

²⁵ selected, checked は論理属性の 1 つであり、JavaScript 言語で値を設定する際は true を与える。

2.3.9.12 ラジオボタン

input 要素に属性 type="radio" を設定することでラジオボタンとなる。

書き方： `<input type="radio" name="名称" value="値 1">`
`<input type="radio" name="名称" value="値 2">`
`<input type="radio" name="名称" value="値 3">`
 ⋮
 (必要なだけ記述する)
 ⋮

ラジオボタンは複数の input 要素を同一の name 属性でグループ化して構成し、その内の 1 つがチェックできる仕組みである。同一の name 属性をもつラジオボタンの内どれか 1 つのタグ内に **checked** を記述して予めチェックされた状態にしておくのが良い。

記述例：

```
1 <input type="radio" name="r1" value="値1" id="r11" checked>
2 <label for="r11">項目1</label>
3 <input type="radio" name="r1" value="値2" id="r12">
4 <label for="r12">項目2</label>
5 <input type="radio" name="r1" value="値3" id="r13">
6 <label for="r13">項目3</label>
```

☒ 項目1 ☐ 項目2 ☐ 項目3

ラジオボタン

これを Web ブラウザで表示すると右図のようになる。

2.3.9.13 フィールドセット要素

form 配下の UI 要素をグループ化してレイアウトする際に fieldset 要素が役立つ。

書き方： `<fieldset id="一意名">`
 `<legend>キャプション</legend>`
 ⋮
 (UI の記述)
 ⋮
 `</fieldset>`

fieldset の使用例を示す。

記述例：

```
1 <style>#f1 { width: 200pt; }</style>
2 <fieldset id="f1">
3   <input type="checkbox" name="c1" id="c1" checked>
4   <legend>グループ見出し</legend>
5   <label for="c1">項目11 </label>
6   <input type="checkbox" name="c2" id="c2">
7   <label for="c2">項目12</label>
8   <fieldset id="f2">
9     <input type="radio" name="r1" value="値1" id="r11" checked>
10    <label for="r11">項目21 </label>
11    <input type="radio" name="r1" value="値2" id="r12">
12    <label for="r12">項目22 </label>
13  </fieldset>
14 </fieldset>
```

これを Web ブラウザで表示すると図 27 のようになる。

グループ見出し

☒ 項目11 ☐ 項目12

☒ 項目21 ☐ 項目22

図 27: fieldset によるグループ化

2.3.9.14 メータと進捗インジケータ（プログレスバー）

■ メータ

一次元の数値を可視化する UI 要素の 1 つに meter がある。

書き方： `<meter min="最小値" max="最大値" value="値"
low="境界値 1" high="境界値 2" optimum="最適値">コンテンツ</meter>`

min から max の範囲にある value の値を可視化する。min のデフォルトは 0, max のデフォルトは 1.0 である。low, high に値を設定すると min ~ max の範囲を区分けして可視化する。この際に optimum の値を設定することで、どの区間を最適値として可視化するかを指定できる。デフォルトでは最適の区間は low~high である。

この要素による可視化ができない Web ブラウザでは「コンテンツ」が表示される。

■ 進捗インジケータ（プログレスバー）

一次元の数値を可視化する UI 要素の 1 つに progress がある。

書き方： `<progress max="最大値" value="値">コンテンツ</progress>`

0 から max の範囲にある value の値を可視化する。max のデフォルトは 1.0 である。

この要素による可視化ができない Web ブラウザでは「コンテンツ」が表示される。

《メータと進捗インジケータの例》

`<meter min="0" max="1.0" low="0.2" high="0.8">メーター</meter>` と

`<progress max="1.0">進捗インジケータ</progress>`

の value 属性の値に応じた表示の変化の例を図 28 に示す。

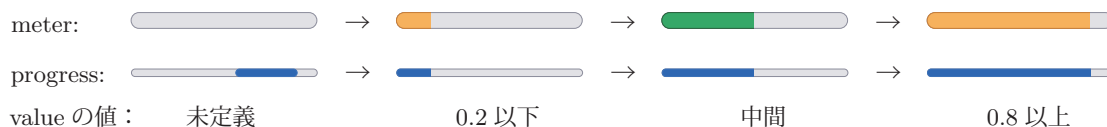


図 28: meter と progress

progress 要素は value が未設定（未定義）の場合、インジケータが左右に浮動する。

2.3.10 外部リソースへのリンク要素

現実的に HTML 文書を作成する際、CSS は HTML とは別のファイルに作成して使用することが多い。また、画像データを Web サイトのアイコン²⁶ として使用することもでき、HTML 文書は当該文書ファイル (*.html) 以外のリソース（外部リソース）を使用することが多い。

HTML 文書に外部リソースを関連付けるには link 要素を用いる方法がある²⁷。

書き方： <link rel="関係" href="外部リソース" type="メディアタイプ">

「関係」には、外部リソースと当該 HTML 文書との関係を記述する。具体的には "stylesheet" (CSS), "icon" (アイコン) など様々なものがある。また「外部リソース」には外部リソースの URL やパスを記述する。

■ メディアタイプについて

type= の部分は多くの場合省略できるが、メディアタイプ (MIME タイプとも呼ぶ) を明示する場合に記述する。メディアタイプは、インターネット上でファイルなどのリソースを配信する際に、そのリソースの種類を明示するために記述する情報であり、

”タイプ名/サブタイプ名”

の形式で表記する。

タイプ名は表 12 に挙げるようなもので、リソースのデータとしての種類を示す。また、サブタイプ名は、フォーマット形式やベンダー名といった更に詳細な情報を示す。

表 12: メディアタイプのタイプ名 (一部)

タイプ名	種 類	タイプ名	種 類
text	テキストデータ	application	各種アプリケーション用データ
font	フォントデータ	image	静止画像データ
audio	音声データ	video	動画画像データ

メディアタイプは IANA (Internet Assigned Numbers Authority)²⁸ が取りまとめて管理している。

2.3.11 スタイルシートの形態

HTML 文書内に style 要素として記述されたスタイルシートは内部スタイルシート²⁹ と呼ぶ。また、外部リソースとして読み込まれたスタイルシートを外部スタイルシート³⁰ と呼ぶ。

この他にもインラインスタイルと呼ばれるスタイルの記述方法もある。これは、対象となる HTML 要素の開始タグの中に style 属性として記述するものである。

記述例： <p style="font-size:24pt;">Sample Text</p>

スタイルシートの各種形態の優先順位は、

1. インラインスタイル
2. 内部スタイルシート
3. 外部スタイルシート

の順であり、同一の HTML 要素に対する同じ CSS 属性の設定がそれぞれの形態のスタイルシートに重複して記述されている場合は上記の順で優先される。

²⁶ サイトアイコン、ファビコン、favicon (favorite icon) などと呼ばれる。

²⁷ JavaScript のプログラムのファイルは link 要素ではなく、script 要素で読み込む。

²⁸ <https://www.iana.org/>

²⁹ 内的スタイルシートとも呼ばれる。

³⁰ 外的スタイルシートとも呼ばれる。

3 JavaScript

3.1 前提事項

本書では JavaScript の言語処理系（JavaScript エンジン）として Google 社の V8 を前提とする。ただし、V8 以外であっても広く普及している Web ブラウザに搭載されている JavaScript エンジン³¹ を使用する場合は特に問題は発生しないと思われる。

本書では JavaScript プログラムの実行例を示す際、HTML ファイルとして記述したものを Web ブラウザで実行する形か、もしくは短いプログラムを直接的に言語処理系に与えて実行する形を取る。後者の形でプログラムを実行するには Web ブラウザに付属の開発者用のツールを使用すると良い。また、HTML 要素にアクセスしない範囲のプログラムであれば Node.js³² を使用して JavaScript のプログラムを実行することも可能である。（下記の実行例を参照）

例. Google Chrome ブラウザのデベロッパーツールのコンソールでの実行

```
> 1+2  ←計算式の入力とエンターキーの押下
< 3      ←システムからの出力
```

これは Google Chrome ブラウザのデベロッパーツールのコンソールで計算処理を実行した例である。コンソールには入力プロンプト「>」が表示され、それに続いて JavaScript の文や式を記述してエンターキーを押す。その後、出力を意味する記号「<」と計算結果が表示される。

例. コマンドシェルから起動した Node.js での実行

```
> 1+2  ←計算式の入力とエンターキーの押下
3      ←システムからの出力
```

これは、OS のコマンドシェル（ターミナル環境）から起動した Node.js で計算処理を実行した例である。ディスプレイには入力プロンプト「>」が表示され、それに続いて JavaScript の文や式を記述してエンターキーを押す。その後、直下に計算結果が表示される。

上に示した 1+2 は処理結果として値をもたらす**式**であり、式を単体で実行するとその値が表示される。

他の処理系（他の Web ブラウザ）でも、表示の体裁に多少の差異はあるが、基本的には同様の作業が可能である。

本書ではサンプルプログラムをコンソールで実行する例を示す際、簡単のため「<」の表記は基本的に省くものとする。

3.1.1 コンソール出力

Web ブラウザに搭載された JavaScript 言語処理系の API³³（Application Programming Interface）には、コンソールそのものを表すオブジェクト `console` があり、これに対して各種のメソッドを実行することができる。特に本書では、提示するサンプルプログラムにおいてコンソールへの出力処理を行う `log` メソッドを多用する。

例. コンソールへの出力処理

```
> console.log(123)  ←数値の値 123 をコンソールに出力する処理の実行
123                ←（出力 1）上記の処理による値 123 の出力
undefined          ←（出力 2）log メソッドからの戻り値
```

このように、サンプルプログラムの実行結果として出力が 2 つ得られているように見える。1 つ目は `log` メソッドによる値の出力がコンソールに表示されたもので、2 つ目は `log` メソッド自身が返す値である。`log` メソッドのように実行結果としての値を返さないものは `undefined`³⁴ となる。

Node.js にも `console` オブジェクトがあり、`log` メソッドはターミナルウィンドウ（**標準出力**）に値を出力する。

³¹Mozilla Firefox の SpiderMonkey や macOS 用 Safari の JavaScriptCore など。

³²<https://nodejs.org/>

³³プログラミングに使用するために用意されたクラスや関数、各種オブジェクトなどのこと。

³⁴`undefined` も 1 つの値である。

3.1.2 文を複数の行に分割して記述する方法

長い文を複数の行に分割して記述することができる。

例. 式の分割入力 (Web ブラウザのコンソールの場合)

```
> 1+2+ Enter    ←「+」で行を終える
      3+4 Enter    ←続く式を入力 (ここで式は終了)
      10                                ←計算結果
```

このように「+」で行を終えると、続きの行の記述が必要となるので、式の入力が継続される。これはターミナル環境における Node.js においても同様である。(次の例)

例. 式の分割入力 (Node.js の場合)

```
> 1+2+ Enter    ←「+」で行を終える
... 3+4 Enter    ←続く式を入力 (ここで式は終了)
      10                                ←計算結果
```

このように Node.js ではターミナル環境における入力継続の際には、継続入力のプロンプト「...」が表示される。

3.1.3 プログラム中のコメント

`/*` と `*/` で括った範囲は**コメント**と見なされ実行の対象とはならない。またこの形式のコメントは改行を含めて複数行に渡って記述することができるが、入れ子の構造にすることはできない。

プログラムの中に「`//`」を記述すると、それ以降行末までがコメントと見なされる。

3.2 文とブロック

JavaScript のプログラムを構成する個々の文の終端には**セミコロン**「`;`」を記述する³⁵。また、0 個以上の文を波括弧 `{ }` で括ったものを**ブロック**と呼ぶ。0 個の文は**空文**と呼ばれ、セミコロン 1 つで表記する。

値を与える**式**を文として記述することが可能であり、それを**式文**と呼ぶ。式文の値は捨てられる。

ブロックは条件分岐や反復といった制御構文、関数定義などにおいて、実行するプログラムの範囲を指定する際に用いられる。またブロックは値ではなく、変数に代入することはできない。(次の例)

例. ブロックの誤った使い方

```
> a = { console.log(1); console.log(2); } Enter    ←ブロックを変数 a に代入する試み
a = { console.log(1); console.log(2); }
      ^
Uncaught SyntaxError: Unexpected token '.'    ←文法エラーとなる
```

代入ではなく

```
{ console.log(1); console.log(2); }
```

をそのまま実行するとブロックとして扱われ、正しく動作する。波括弧 `{ }` は後の「3.6.2 オブジェクト: Object」(p.58) で解説する**オブジェクト**でも使用し、その場合は値として扱われ、変数への代入などができるので若干の注意が必要である。

3.3 関数, メソッド, プロパティ

関数は「**関数 (引数並び)**」と記述して実行するもので、引数に対して何らかの処理を行い、その結果の値を返すものである。関数は入れ子の記述 (ネスト) が可能であり、

関数 1(..., 関数 2(...), ...)

という書き方ができる。

メソッドはドット表記により「**対象. メソッド (引数並び)**」と記述して実行するもので、引数と対象を用いて何らかの処理を行い、その結果の値を返すものである。また、メソッドは処理対象に対して何らかの処理を施す (変更を加える) こともある。

³⁵ASI (自動セミコロン挿入) の機能により文の終端にセミコロンを記述しなくても問題が発生しない場合もある。

対象が何らかの属性を持ち、その値を設定もしくは参照する場合もドット表記を用いて「**対象. プロパティ**」と記述する。

メソッドやプロパティの記述においては次のようにドット表記を連鎖させることができる。

対象. プロパティ1. プロパティ2...

広い意味では、メソッドも処理機能を持ったプロパティであり、実際にメソッドのことを単にプロパティと呼ぶことも多い。また、複数のメソッドをドット表記で連鎖する記述のことを**メソッドチェーン**と呼ぶことがある。

3.4 変数

変数とは各種の値に与えられる記号的な名前であり、変数を介してそれらの値にアクセスすることができる。変数には代入演算子「**=**」（イコール）を用いて値を代入することができ、代入処理自体はその値を返す³⁶。初期値を与えずに宣言された変数の初期値は `undefined` となる。

3.4.1 変数のスコープ

変数には、それが有効である範囲（**スコープ**）があり、変数はその使用に際して**宣言**する必要がある。（表 13）また、変化しない値は**定数**として名前を与えて宣言することができる。

表 13: 変数、定数の宣言のための文

宣 言 文	解 説
<code>let</code> 変数並び	ブロックスコープの変数を宣言する。
<code>const</code> 変数並び	ブロックスコープの 定数 を宣言する。
<code>var</code> 変数並び	関数スコープあるいはグローバルスコープの変数を宣言する。
（宣言せず）	グローバルスコープの変数（ グローバル変数 ）となる。（非推奨）

変数並びは記号をコンマで区切って並べたものである。

宣言や代入が全くなされていない未使用の記号を参照するとエラー（例外）が発生する。

例. 未使用の記号を参照する試み

```
> x Enter      ←記号 x が未使用の場合に参照を試みると…
Uncaught ReferenceError: x is not defined      ←エラーとなる
```

グローバルスコープの変数（**グローバル変数**）は、特定のブロックや関数に限定されず、プログラムの任意の場所からアクセスできる³⁷。

`let` や `var` で変数を宣言する際に初期値を代入することができる。また、`const` で定数を宣言する際には必ず初期値を代入しなければならない。

`let` 宣言によって、変数のスコープ（有効範囲）が限定される様子を次に示す。

例. ブロック外での変数宣言

```
> let a = 2 Enter      ←ブロック外で変数 a を宣言して値 2 を代入
undefined      ←変数宣言の処理自体は値を返さない
```

続いて次のようにブロックを記述して実行する。

例. ブロックの実行（先の例の続き）

```
> { Enter      ←ブロックの開始
  let a = 3; Enter      ←ブロック内に限定された変数 a
  console.log(a); Enter      ←それを出力
} Enter      ←ブロックの終了
3      ← console.log による出力：ブロック内で代入した値
undefined      ←ブロックの実行結果（console.log の実行結果）
```

³⁶これを応用すると `a = b = 12` といった、代入の連鎖が実現できる。

³⁷実際のプログラミングにおいて、グローバル変数の使用は安全ではなく、あまり推奨されない。

この例でわかるように、ブロック外の変数 `a` ではなく、ブロック内の変数 `a` の値が表示されている。この直後に変数 `a` の値を確認する。(次の例)

例. ブロック外で変数 `a` の値を確認 (先の例の続き)

```
> a  ←変数 a の値を確認
2      ←ブロック外で宣言した変数 a の値
```

以上のことから、同じ名前 `a` を持つ変数がブロックの内外で別のものになっていることがわかる。ただし、入れ子になったブロックにおいて、内側のブロックから外側のブロックのスコープの変数にアクセスすることはできる。

例. ブロックの内側から外側のブロックの変数にアクセスする

```
> let a = 3;  ←外側のスコープの変数
undefined      ←上の文の戻り値
> {  ←ブロックの開始
  console.log(a);  ←変数の値の確認 (1)
  a = 5;          ←変数に値を代入
  console.log(a);  ←変数の値の確認 (2)
... }  ←ブロックの終了
3      ← (1) による出力
5      ← (2) による出力
undefined      ← (2) の処理の戻り値
```

ブロックの内側から外側の変数 `a` の値を参照できていることがわかる。またブロックの内側で `a = 5`; として値を設定しており、これがブロックの外側でも確認できる (次の例)

例. ブロック外で変数の値を確認 (先の例の続き)

```
> a  ←変数の値を確認
5      ←ブロック内でされた値
```

ブロック内で `let` 宣言されていない変数は、上位のブロックで `let` 宣言された変数と見做される。

上位のブロックにおいても `let` で宣言されていない変数をブロック内部で `let` 宣言せずに使用するべきではない。これについては次に解説する。

3.4.2 変数の使用における良くない例

変数の宣言をせずに記号に値を代入すると、それは**グローバルスコープ**の変数となり、特定のブロックや関数に限定されずプログラムの任意の場所からアクセスできる。このような形での変数の使用はプログラム開発において安全ではなく、処理系の実行モードによってはエラーとなる場合もあり、推奨されない。

次に示す例は、変数宣言をせずに未使用の記号 `g` に値を代入するものである。

例. 宣言せずに変数に値を代入する例

```
> {  ←ブロックの開始
  g = 5;  ←宣言されていない変数 g への代入
  console.log(g);  ←それを出力
}  ←ブロックの終了
5      ← console.log による出力：ブロック内で代入した値
undefined      ←ブロックの実行結果 (console.log の実行結果)
> g  ←ブロック外で変数 g の値を確認
5      ←参照できている
```

ブロック内で値を代入した `g` がブロック外部でも参照できている。

重要)

本書で短いサンプルプログラムを提示する際、簡便のために変数宣言の記述を省略することが多いが、実際のプログラム開発においては、変数は宣言して使用すべきである。また、`var` による宣言よりも、ブロックごとにスコープを限定できる `let` による変数の宣言をするべきである。

プログラム内で値の変更が発生しない記号に関しては、変数よりも `const` によって定数として宣言すべきである。

3.4.2.1 更に注意すべき事柄

記号 `a`, `b` 共に未使用である場合に次のようなプログラムを実行する。

例. 変数宣言時の初期値の代入

```
> { Enter      ←ブロックの開始
    let a=b=24; Enter    ←変数宣言時に初期値を代入する
  } Enter      ←ブロックの終了
undefined      ←ブロックの実行結果 (let による宣言の結果)
```

これは、ブロックスコープの変数を宣言して初期値を与える例である。ただし、ブロック内の変数として宣言されたのは `a` のみであることに注意しなければならない。このことを確かめるために次の例を実行する。

例. ブロック外での変数 `a` の確認 (先の例の続き)

```
> a Enter    ←変数 a はブロック内のスコープなのでブロック外で参照を試みると…
Uncaught ReferenceError: a is not defined    ←エラーとなる
```

続けて、変数 `b` について同様に確認する。(次の例)

例. ブロック外での変数 `b` の確認 (先の例の続き)

```
> b Enter    ←変数 b を確認すると…
24      ←参照できている
```

変数 `b` はブロックスコープではないことがわかる。このようなことが起こった理由について考える。

最初に `let a=b=24;` としているが、これを詳細に捉えると、

```
let a=(b=24);
```

という記述と同じであることがわかる。これは `b=24` の部分が先に実行されており、記号 `b` が `let` による宣言から外れていることからグローバル変数となっていることを意味する。従って、グローバル変数 `b` に対する代入結果の値である `24` が `let` 宣言で記号 `a` に代入されることとなる。

これは、プログラム中の見つけにくいバグの原因になることがあるので注意が必要である。ブロックスコープの複数の変数に「`=`」の連鎖で値を代入するには次のように記述するべきである。

例. 変数の初期値の設定の工夫

```
> { Enter      ←ブロックの開始
    let a,b; Enter    ←変数の宣言のみ
    a = b = 24; Enter    ←代入の連鎖
  } Enter      ←ブロックの終了
24      ←最後の文 (代入処理) の結果の値
```

この処理では変数 `a`, `b` 共にブロック内のスコープとなり、ブロック外では参照できない。(次の例)

例. ブロック外での変数の確認 (先の例の続き)

```
> a Enter    ←変数 a はブロック内のスコープなのでブロック外で参照を試みると…
Uncaught ReferenceError: a is not defined    ←エラーとなる
> b Enter    ←変数 b はブロック内のスコープなのでブロック外で参照を試みると…
Uncaught ReferenceError: b is not defined    ←エラーとなる
```

3.4.3 分割代入

後の「3.6 データ構造」(p.50) で解説する **配列**や**オブジェクト** (Object クラス) を代入処理に応用すると、高度な分割代入ができる。

例. 配列を用いた分割代入

```
> [a,b] = [3,5]  ←要素を対応させる形で代入する
[ 3, 5 ]        ←左辺の値が代入処理の結果として表示される
> a  ← a の値の確認
3
> b  ← b の値の確認
5
```

このように、データ構造の要素の位置に対応する形で代入ができる。このとき、左辺のデータ構造の要素には基本的には変数記号もしくはアンダースコア「_」のみを持つ形にしなければならない。

例. 良くない例

```
> [a,5] = [3,5]  ←左辺の要素に具体的な値（数値）があると
[a,5] = [3,5]    ←↓ エラーとなる
Uncaught SyntaxError: Invalid destructuring assignment target
```

var, let, const といった宣言で分割代入することもできる。その場合、代入処理の結果はコンソール上では undefined となる。

例. 宣言付きの分割代入

```
> var [a,b] = [7,9]  ← var 宣言で分割代入
undefined          ←代入処理の結果
> a  ←正常に
7
> b  ←代入されている
9
```

左辺の要素数より右辺の要素数が大きい場合は、変数に対応しない右辺の値が無視される。左辺の要素の方が多い場合は、値に対応しない変数は undefined となる。

例. 左辺の要素数が大きい場合

```
> [a,b,c] = [1,2]  ← c に対応する値がない
[ 1, 2 ]
> c  ← c の値を確認
undefined
```

上の例の場合、c に既に値が代入されていても、上記の分割代入の後は c は undefined となることに注意しなければならない。

オブジェクト (Object)³⁸ を用いた分割代入では、要素の順序に依らずプロパティの名前に対応する形で代入が行われる。

例. オブジェクトを用いた分割代入

```
> obj = { a:6, b:7 }  ←オブジェクトを作成
{ a: 6, b: 7 }
> { b:x, a:y } = obj  ←分割代入
{ a: 6, b: 7 }
> x  ←変数 x の値の確認
7
> y  ←変数 y の値の確認
6
```

この例のように、分割代入の左辺のオブジェクトの中の値を記述する箇所に変数の記号を記述する。これにより、代入の対象（変数と値）がキーで対応付けられる。

³⁸ 「3.6.2 オブジェクト：Object」(p.58) で詳しく解説する。

3.4.3.1 分割代入における「オブジェクトのパターン」

オブジェクトを用いた分割代入において、右辺のオブジェクトのキーと同じ表記の変数に代入する場合は、左辺の記述をより簡単にすることができる。

例. オブジェクトを用いた分割代入の簡易な表記（先の例の続き）

```
> {a,b} = obj  ←変数 a, b に対する代入
{ a: 6, b: 7 } ←右辺の値が表示される
> a  ←変数 a の値を確認
6
> b  ←変数 b の値を確認
7
```

このように、右辺のキー a の値が左辺の変数 a に代入され、同様に右辺のキー b の値が左辺の変数 b に代入される。

この例の {a,b} ような表記はオブジェクトのリテラル表記ではなく、分割代入の際に有効な**オブジェクトのパターン**³⁹ と呼ばれるものである。

3.4.3.2 分割代入の高度な応用

分割代入を応用すると、複雑なデータ構造内の指定した部分を選択的に抽出することができる。

例. 分割代入の高度な応用

```
> [ , , [a, , [b, ]] ] = [1,2,[3,4,[5,6],7],8]  ←部分要素の抽出
[ 1, 2, [ 3, 4, [ 5, 6 ], 7 ], 8 ]
> a  ←変数 a の値を確認
3
> b  ←変数 b の値を確認
5
```

またこの例のように、左辺の変数を部分的に省略することもできる。

3.4.4 変数の廃棄

var, let で宣言された変数を未使用の状態にするには undefined を代入する。この場合、当該変数は引き続き存在する。var, let の宣言なしで作成されたグローバル変数を完全に廃棄するには delete 演算子を用いる。

例. グローバル変数の廃棄

```
> x = 123  ←グローバル変数 x の作成
123 ←設定された値
> x  ←グローバル変数 x の値の確認
123 ←値が参照できている
> delete x  ←グローバル変数 x の廃棄
true ←処理結果
> x  ←変数 x の値の確認
Uncaught ReferenceError: x is not defined ← x というものが存在しない旨のエラー
```

作成したグローバル変数 x が delete 演算子によって廃棄されており、存在しない状態になっていることがわかる。

³⁹オブジェクトのデストラクチャリングパターンとも言う。

3.5 基本的なデータ型

3.5.1 数値

JavaScript の数値 (number 型) は IEEE 754 の倍精度浮動小数点数である。扱うことができる最小の正の数値は `Number.MIN_VALUE` に、最大の正の数値は `Number.MAX_VALUE` に定義されている。整数値も浮動小数点数と同じ型 (number 型) の値として扱われる。整数として正確に扱うことのできる値の範囲の最小値は `Number.MIN_SAFE_INTEGER` に、最大値は `Number.MAX_SAFE_INTEGER` に定義されている。(表 14)

表 14: 扱える数の限界

定 義	値	備 考
1) <code>Number.MIN_VALUE</code>	5×10^{-324}	正の最小値
2) <code>Number.MAX_VALUE</code>	$1.7976931348623157 \times 10^{308}$	正の最大値
3) <code>Number.MIN_SAFE_INTEGER</code>	$-(2^{53}-1) = -9,007,199,254,740,991$	安全に扱える最小の整数値
4) <code>Number.MAX_SAFE_INTEGER</code>	$2^{53}-1 = 9,007,199,254,740,991$	安全に扱える最大の整数値

表 14 の 3)~4) の範囲を超える整数値を表現することもできるが、そのような値は正しく扱えないことがある。(次の例)

例. 安全でない整数値

```
> Number.MAX_SAFE_INTEGER + 1  Enter  ←安全な整数の最大値に 1 を加える
9007199254740992                ←値が正しく得られるように見えるが,
> Number.MAX_SAFE_INTEGER + 8  Enter  ← 8 を加えると…
9007199254741000                ←正確な計算結果は得られない
```

表 14 の 3)~4) の範囲を超える整数値を扱う場合は後に説明する `bigint` 型の整数を使用する。

参考) 数値は `Number` コンストラクタを明に用いて生成することもできる。

例. `Number` コンストラクタ

```
> Number(2.718281828)  Enter  ← Number コンストラクタ
2.718281828             ←数値
```

3.5.1.1 基本的な演算

算術演算など基本的な演算を表 15 に示す。

表 15: 基本的な演算

式	解 説	式	解 説
<code>x + y</code>	和	<code>x - y</code>	差
<code>x * x</code>	積	<code>x / y</code>	除算
<code>x ** y</code>	べき乗 x^y	<code>x % y</code>	剰余 ($x \div y$ の余り)

■ インクリメント／デクリメント演算子

変数に代入されている値を 1 増やす (インクリメントする)、あるいは減らす (デクリメントする) 演算子がある。

インクリメント演算 (1): 変数++ 「変数」の値を参照後にインクリメントする。

インクリメント演算 (2): ++変数 「変数」をインクリメントした後に値を参照する。

例. 末尾の++

```
> a = 3; console.log(a++)  Enter  ←値を参照した後にインクリメント
3                            ←参照時の値が log メソッドで出力されている
undefined                  ← (log メソッドの戻り値)
> a  Enter                  ←上の処理の後で値を確認すると
4                            ←インクリメントされている
```

例. 前置型の++（先の例の続き）

```
> console.log(++a) Enter ←値をインクリメントした後で参照
5 ←インクリメント済みの値が log メソッドで出力されている
undefined ← (log メソッドの戻り値)
> a Enter ←上の処理の後で値を確認すると
5 ←そのまま
```

同様の書き方で、デクリメント演算子「--」によって変数の値を 1 減らすことができる。

3.5.1.2 累算的な代入演算子

既に値が代入されている変数には累算的な代入が実行できる。（次の例）

例. 累算的な代入

```
> a = 3; a = a + 5; a Enter ←変数 a の最終的な値は
8 ←このようになる
```

この例の `a = a + 5` という処理では、先に右辺の `a + 5` を行った後で、その値を左辺に記述した変数に代入している。このように、既存の変数の値を用いた演算結果を当該変数に上書きして代入する処理（累算的な代入⁴⁰）は実際のプログラミングでは多く行われ、`a = a + 5` という処理を `a += 5` と簡潔に記述する（表 16）ことができる。

表 16:

記 述	解 釈	記 述	解 釈	記 述	解 釈
<code>v += x</code>	<code>v = v + x</code>	<code>v -= x</code>	<code>v = v - x</code>	<code>v *= x</code>	<code>v = v * x</code>
<code>v /= x</code>	<code>v = v / x</code>	<code>v %= x</code>	<code>v = v % x</code>		

例. 累算的な代入（その 2）

```
> a = 3; a += 5; a Enter
8 ←先の例と同じ
```

3.5.1.3 特殊な値

`number` 型のいかなる値よりも大きな値が `Number.POSITIVE_INFINITY`（無限大）として、いかなる値よりも小さな値が `Number.NEGATIVE_INFINITY`（負の無限大）として定義されている。またそれらは、`Infinity`、`-Infinity` とそれぞれ同じ値である。

例. 正負の無限大

```
> Number.MAX_VALUE < Infinity Enter ← Number の最大値より無限大が大きい?
true ←真（論理値:Boolean）
> Number.MIN_VALUE > -Infinity Enter ← Number の最小値より負の無限大が小さい?
true ←真（論理値:Boolean）
```

無限大を意味する値はあくまで便宜的なものであり、数値計算に使用する場合は注意すること。

例. 無限大が関わる計算

```
> 1/0 Enter ←この計算結果は
Infinity ←無限大になる
> 1 / Infinity Enter ←有限な正の値を無限大で割ると
0 ←0 になる
> Infinity + Infinity Enter ←無限大同士の和
Infinity ←無限大
```

注) JavaScript では 0 による除算がエラーにならない。

計算の結果が不明な場合は非数となることがある。

⁴⁰再帰的な代入と表現することもある。

例. 非数となる計算

```
> Infinity - Infinity  Enter    ←無限大の差は
NaN                      ←非数
> Infinity / Infinity  Enter    ←無限大の除算は
NaN                      ←非数
```

非数は数値計算に使用すべきではない。

■ 有限値の判定

値が有限かどうかを判定するには `isFinite` 関数を使用する。

例. 有限値かどうかの判定

```
> isFinite( 10 )  Enter    ← 10 は有限値か?
true              ←真 (論理値:Boolean)
> isFinite( Infinity )  Enter    ← Infinity は有限値か?
false            ←偽 (論理値:Boolean)
> isFinite( NaN )  Enter    ← NaN は有限値か?
false            ←偽 (論理値:Boolean)
```

■ 非数の判定

値が NaN かどうかを判定するには `isNaN` 関数を使用する。

例. NaN かどうかの判定

```
> isNaN( NaN )  Enter    ← NaN は NaN か?
true            ←真 (論理値:Boolean)
> isNaN( Infinity )  Enter    ← Infinity は NaN か?
false          ←偽 (論理値:Boolean)
> isNaN( 10 )  Enter    ← 10 は NaN か?
false          ←偽 (論理値:Boolean)
```

3.5.1.4 長整数: BigInt

`number` 型の値として扱えないほど絶対値が大きな整数は `bigint` 型の値として扱う。このクラスの値を作るには `BigInt` コンストラクタを用いるか、整数値表現の末尾に「`n`」を付ける。

書き方: `BigInt(値)`

「値」には文字列による数値表現を与えることができる。また整数値を与えても良い。

例. BigInt の値

```
> BigInt("1234567890987654321")  Enter    ← BigInt コンストラクタ
1234567890987654321n              ←得られた値
> 36n  Enter                      ←小さな値も BigInt として扱える
36n                               ←得られた値
```

例. BigInt の計算

```
> 2n ** 100n  Enter    ← 2100 の計算
1267650600228229401496703205376n    ←得られた値
```

3.5.1.5 数値リテラル

直接的に数値を表記したものを**数値リテラル**という。10進数の数値を記述する際には先頭に0を書かない。0で始まる接頭辞を付けると様々な基数で(表17)で数値を表現できる。

表 17: 数値リテラルの接頭辞

接 頭 辞	解 説	接 頭 辞	解 説
0 (ゼロ)	8 進数	0o (ゼロオー)	8 進数
0x (ゼロエックス)	16 進数	0b (ゼロビー)	2 進数

例. 8 進数 $77_8 = 63_{10}$

```
> 077 Enter ←ゼロ 77
63 ← 10 進数の値
> 0o77 Enter ←ゼロオー 77
63 ←先と同じ値
```

例. 16 進数, 2 進数

```
> 0xff Enter ← ff16
255 ← 25510
> 0b1111 Enter ← 11112
15 ← 1510
```

注意) 浮動小数点数の直前に 0 を付けるとエラーとなる。(次の例)

例. 浮動小数点数の前に 0 を付けることで起こるエラー

```
> 012.345 ←このような記述を試みる
012.345
Uncaught SyntaxError: Unexpected number ←文法エラー
```

表 17 の接頭辞と BigInt の接尾辞「n」を併用することができる。

例. 16 進数表現の BigInt

```
> 0xfffffffffffffffffffffffffffffn Enter ← f が 30 個
1329227995784915872903807060280344575n ←このような BigInt の値
> 2n ** 120n - 1n Enter ← 10 進数で上と同じ値を求める (2120 - 1)
1329227995784915872903807060280344575n ←上と同じ値
```

浮動小数点数の値を**指数表現**で記述することができる。指数表現は

$$n \times 10^m$$

のように、仮数部 n と指数部 m で表現する形式で、JavaScript では

仮数部 e 指数部

と記述する。

例. 指数表現

```
> 0.031415926535e2 Enter ←この指数表現は
3.1415926535 ←このような値である
```

3.5.1.6 数学関数

JavaScript の組み込みオブジェクト Math には多くの数学関数や定数がある。(表 18)

表 18: 数学関数と定数 (一部)

式	解 説	式	解 説
Math.pow(a,b)	べき乗 a^b	Math.sqrt(x)	平方根 \sqrt{x}
Math.E	自然対数の底 e	Math.exp(x)	指数関数 $\exp[x]$
Math.log(x)	対数関数 $\ln(x)$, $\log_e(x)$	Math.log10(x)	対数関数 $\log_{10}(x)$
Math.log2(x)	対数関数 $\log_2(x)$	Math.PI	円周率 π
Math.sin(x)	正弦関数 $\sin(x)$	Math.cos(x)	余弦関数 $\cos(x)$
Math.tan(x)	正接関数 $\tan(x)$	Math.asin(x)	逆正弦関数 $\sin^{-1}(x)$
Math.acos(x)	逆余弦関数 $\cos^{-1}(x)$	Math.atan(x)	逆正接関数 $\tan^{-1}(x)$
Math.round(x)	小数第一位で四捨五入	Math.ceil(x)	x 以上の最小の整数値
Math.floor(x)	x 以下の最大の整数値	Math.random()	乱数生成* (0 以上 1 未満)
Math.abs(x)	絶対値 $ x $	Math.sign(x)	正負の符号 $x < 0 \rightarrow (-1)$, $x = 0 \rightarrow 0$, $0 < x \rightarrow 1$

* 一様乱数. seed は選択できない。

例. π と正弦関数

```
> Math.PI Enter ←  $\pi$ 
3.141592653589793 ←値
> Math.sin( Math.PI/2 ) Enter ← sin( $\pi/2$ )
1 ←値
```

例. e と対数関数

```
> Math.E Enter ←  $e$ 
2.718281828459045 ←値
> Math.log( Math.E ) Enter ← ln( $e$ )
1 ←値
```

3.5.2 論理値

論理値は論理型の値であり、真もしくは偽をそれぞれ `true` , `false` の 2 つの値で表現する。論理値には表 19 に挙げるような論理演算ができる。

表 19: 論理演算

演算	書き方	解 説
否定	<code>!p</code>	<code>p</code> が <code>true</code> ならば <code>false</code> を、 <code>false</code> ならば <code>true</code> を返す。
積	<code>p && q</code>	<code>p, q</code> 共に <code>true</code> の場合に <code>true</code> を、それ以外の場合は <code>false</code> を返す。
和	<code>p q</code>	<code>p, q</code> どちらか 1 つでも <code>true</code> の場合に <code>true</code> を、共に <code>false</code> の場合は <code>false</code> を返す。

3.5.3 文字列

文字列（文字の並び）は二重引用符「`"`」もしくは単引用符「`'`」で括って記述⁴¹ する。

例. `"abc123"`, `'This is a pen.'`, `"日本語の文字列"`

2 種類の引用符を使い分けると、引用符自体を文字列の中に記述することが容易になる。（次の例）

例. `'a"bc"d'`, `"あ'いうえ'お"`

また、**バックスラッシュ**「`\`」や日本の通貨記号「`¥`」⁴² を引用符記号の直前に記述すると、文字列を括る引用符と同じ記号をその文字列の中に記述することができる。

例. `'a¥'bc¥'d'`, `"a¥"bc¥"d"`

以上のように、引用符を用いて文字列を記述したものを**文字列リテラル**という。

文字列は `string` 型の値であり、`String` コンストラクタを用いて生成することもできる。

例. `String` コンストラクタ

```
> String("abcde") Enter    ← String コンストラクタ
'abcde'           ← 得られた文字列
```

文字列を構成する個々の文字には角括弧「`[]`」で**インデックス**を括ったものを付けて参照することができる。

書き方： 文字列「`インデックス`」

「文字列」中の「インデックス」で指定した位置の 1 文字を返す。インデックスは先頭を 0 とする番号である。

例. インデックスで指定した文字を取得する

```
> s = "abcdefghijklmn" Enter    ← 文字列を作成
'abcdefghijklmn'
> s[1] Enter    ← インデックス位置 1 の文字を参照
'b'           ← 得られた文字
> s[100] Enter    ← 文字が存在しない位置を参照すると…
undefined     ← このようになる
```

文字列は作成した後で変更ができない。このようなデータを指して「**イミュータブル**なデータ⁴³ である」と言う。

例. 文字列を事後に変更する試み（先の例の続き）

```
> s[1] = "B" Enter    ← インデックス位置 1 の文字を変更する試み
'B'           ← 変更できたように見えるが…
> s Enter    ← 内容を確認すると
'abcdefghijklmn' ← 変更されていない
```

文字列が事後に変更できないことがわかるが、変更を試みてもエラーが発生しないことに特に注意しなければならない。

⁴¹V8 エンジンでは基本的に単引用符を、SpiderMonkey では二重引用符を使用するが、開始と終了の引用符が同じであればどちらを使用しても問題ない。

⁴²「`\`」,「`¥`」共に同じアスキーコード `0x5c` (`=9210`) であり、使用するフォントによって表示が異なる。

⁴³これに対して、後で説明する配列やオブジェクトなどは作成後に変更可能な**ミュータブル**なデータである。

3.5.3.1 エスケープシーケンス

エスケープシーケンスは印刷や表示のためのデバイスを制御する様々な文字であり、バックスラッシュ「\」や日本の通貨記号「¥」に続く文字で構成される。(表 20)

表 20: 代表的なエスケープシーケンス

ESC	機 能	ESC	機 能	ESC	機 能
¥n	改行	¥t	タブ	¥r	行頭にカーソルを復帰
¥v	垂直タブ	¥f	フォームフィード	¥b	バックスペース
¥¥	'¥' そのもの	¥"	ダブルクオート文字	¥'	シングルクオート文字

例. エスケープシーケンス (改行)

```
> "a¥nb"  ←このような文字列をコンソールにエコーバックしても
'a¥nb' ←そのまま表示されるが
> console.log( "a¥nb" )  ←出力用のメソッドでコンソールに送出すると
a ←エスケープシーケンスの位置で
b ←改行される
undefined ←log メソッドの戻り値
```

例. エスケープシーケンス (タブ)

```
> "a¥tb"  ←これもエコーバックの場合は
'a¥tb' ←そのまま表示されるが
> console.log( "a¥tb" )  ←出力用のメソッド送出すると
a b ←エスケープシーケンスの位置にタブが入る
undefined ←log メソッドの戻り値
```

■ 文字コードのエスケープシーケンス

Unicode のマルチバイト文字を文字コードのエスケープシーケンス (¥u) として入力することができる。例えば日本語「あ」の文字コードは Unicode の 0x3042 なので次のように入力することができる。

例. 文字コードのエスケープシーケンス

```
> "¥u3042"  ←この入力はいずれに
'あ' ←このような文字列となる
```

3.5.3.2 複数行に渡る文字列の記述

逆引用符「```」⁴⁴ を用いると複数行に渡る文字列のリテラルを記述することができる。

例. 複数行に渡る文字列リテラル

```
> s = `一行目`  ←文字列リテラルの記述開始
... 二行目 
... 三行目 `  ←文字列リテラルの記述終了
'一行目¥n二行目¥n三行目' ←得られた文字列
```

改行のエスケープシーケンスを含む形で 3 行に渡る文字列ができています。バッククオートによる文字列リテラルの記述は、後に説明するテンプレートリテラルのためのものである。

3.5.3.3 文字列の長さ (文字数)

文字列の長さ (文字数) は `length` プロパティから得られる。

例. 文字列の長さ

```
> s = "a あ い う え お"  ←文字列を用意
'a あ い う え お' ←代入結果
> s.length  ←長さを取得
10 ←10 文字
```

⁴⁴バッククオート、バックティックと呼ばれることもある。

3.5.3.4 文字列の連結と分解

文字列は加算演算子「+」で連結することができる。

例. 文字列の連結

```
> "abc" + "日本語" + "123" Enter    ← 3つの文字列の連結  
'abc 日本語 123'          ← 連結結果
```

■ +で文字列を連結する際の注意

+ で文字列と数値を連結することもできる。

例. +で文字列と数値を連結する

```
> 1 + "2" Enter    ← 数値と文字列の連結  
'12'  
  
> "1" + 2 Enter    ← 文字列と数値の連結  
'12'
```

このように、結果は文字列として得られるが、注意すべき点がある。(次の例)

例. 注意すべき連結処理

```
> 1 + "2" + 3 Enter    ← 第2項が文字列  
'123'          ← 結果(1)  
  
> 1 + 2 + "3" Enter    ← 第3項が文字列  
'33'          ← 結果(2)
```

+ で連結する際の数値と文字列の順序によって異なる結果が得られている。これは、+ 演算子が左の項から順番に処理することが原因となっている。すなわち、文字列の項が現れた時点で処理結果が文字列となり、それ以降は結果が文字列となるので、結果(2)は先頭2項の演算までは結果が数値となり、第3項の処理の時点で文字列となる。

■ split メソッド

指定した文字列を境界にして文字列を分解するには split メソッドを使用する。

書き方: 文字列.split(境界文字列)

「境界文字列」を境界にして「文字列」を分解して配列 (Array)⁴⁵ として返す。

例. 文字列の分解

```
> "a:b:c:d:e".split(":") Enter    ← コロンを境界にして文字列を分解  
[ 'a', 'b', 'c', 'd', 'e' ]    ← 分解したものの配列  
  
> a = "a境界b境界c境界d境界e".split("境界") Enter    ← 「境界」を境界にして文字列を分解  
[ 'a', 'b', 'c', 'd', 'e' ]    ← 分解したものの配列
```

■ join メソッド

配列に対する join メソッドによって配列要素の文字列を連結することができる。

書き方: 配列.join(境界文字列)

「配列」の要素を連結する。その際「境界文字列」を挿入する。引数を省略するとコンマ「,」を挿入する。

例. 配列要素の連結 (先の例の続き)

```
> a.join("/") Enter    ← 配列要素の連結  
'a/b/c/d/e'          ← 連結結果  
  
> a.join() Enter    ← 引数を省略して連結  
'a,b,c,d,e'          ← コンマを境界に挿入して連結
```

配列の要素が文字列でなくても連結結果は文字列となる。

例. 文字列以外の要素を join で連結

```
> [1, "二", 3.4, "五"].join(":") Enter    ← 数値要素を含む配列の連結  
'1:二:3.4:五'          ← 連結結果
```

⁴⁵後の「3.6.1 配列: Array」(p.50)で詳しく解説する。

3.5.3.5 文字列の繰り返し

repeat メソッドで文字列を指定した回数だけ繰り返すことができる。

書き方： 文字列.repeat(回数)

指定した「回数」だけ「文字列」を繰り返したものを返す。「回数」には 0 以上の整数を与える。

例. 文字列の繰り返し

```
> s = "Abc" Enter ←この文字列を
'Abc'
> s.repeat(3) Enter ←3回繰り返したものを作る
'AbcAbcAbc' ←得られた文字列
> s.repeat(0) Enter ←0を指定すると
'' ←空文字列が得られる
```

3.5.3.6 部分文字列

substring, substr メソッドで文字列の指定した部分を取得できる。

書き方： 文字列.substring(n, m)

「文字列」のインデックス位置「n」から「m-1」までの部分を取得する。先頭の文字のインデックスは 0 である。

書き方： 文字列.substr(n, 長さ)

「文字列」のインデックス位置「n」から「長さ」で指定した部分を取得する。

例. 文字列の部分の取得

```
> "abcdefghijklmn".substring(8,11) Enter ←インデックス位置 8 から 10 (=11-1)
'ijk' ←得られた部分
> "abcdefghijklmn".substr(3,6) Enter ←インデックス位置 3 から 6 文字分
'defghi' ←得られた部分
```

3.5.3.7 文字列の含有検査

includes メソッドを用いると、文字列の含有検査が行える。

書き方： 対象文字列.includes(部分文字列)

「対象文字列」の中に「部分文字列」が含まれていれば true を、含まれていなければ false を返す。

例. 文字列の含有検査

```
> "abcdefg".includes( "cd" ) Enter ←対象文字列中に "cd" があるか
true ←判定結果
> "abcdefg".includes( "xy" ) Enter ←対象文字列中に "xy" があるか
false ←判定結果
```

3.5.3.8 文字列の探索

対象文字列の中で、指定した文字列が最初に現れるインデックス位置を知るには indexOf メソッドを用いる。行末から行頭にかけて探索する場合は lastIndexOf メソッドを用いる。

書き方： 対象文字列.indexOf(探索文字列)

書き方： 対象文字列.lastIndexOf(探索文字列)

「対象文字列」の中から「探索文字列」を探し、そのインデックス位置を返す。見つからない場合は -1 を返す。

例. 文字列の探索

```
> "abcdabcd".indexOf("bc") Enter ←対象文字列中で "bc" の位置を探す
1 ←インデックス位置 1 に検出
> "abcdabcd".lastIndexOf("bc") Enter ←末尾から逆順に探す
5 ←インデックス位置 5 に検出
> "abcdabcd".indexOf("bx") Enter ←存在しない文字列を探すと
-1 ←見つからなかった
```

3.5.3.9 テンプレートリテラル（テンプレート文字列）

テンプレートリテラル（テンプレート文字列）はバッククオート「```」で括った文字列であり、式や値をプレースホルダを使用して埋め込むことができる。

プレースホルダ： `${ 式 }`

これをバッククオートで括った範囲内に記述する。

例. テンプレートリテラルへの値の埋め込み

```
> a = 12; b = 34; Enter    ← a, b の値を用意
34                                ← 2つ目の代入式の値
> `a+b=a+b` Enter    ← a, b の値を埋め込む
`12+34=46`                        ← 得られた文字列
```

テンプレートリテラルは改行を含めて複数行に渡って記述することができる。

例. 複数行に渡るテンプレートリテラルの記述（先の例の続き）

```
> s = `一行目` Enter    ← 記述開始
二行目: a+b=a+b Enter
三行目 ` Enter    ← 記述終了
`一行目\n二行目: 12+34=46\n三行目`    ← 得られた文字列
```

入力時の改行がエスケープシーケンスとして文字列の中に含まれていることがわかる。もちろんこれを `console.log` でディスプレイに送出すると改行効果が得られる。（次の例）

例. 整形出力（先の例の続き）

```
> console.log(s) Enter
一行目
二行目: 12+34=46
三行目
undefined    ← log の戻り値
```

■ タグ付きテンプレートリテラル

テンプレートリテラルの記述の先頭に、文字列編集用の関数（**タグ関数**）を記述（**タグ付きテンプレートリテラル**）すると高度な変換処理が実現できる。これについて例を挙げて解説する。

次のような日付編集のテンプレートリテラルを考える。

例. 日付の編集

```
> y = 2024; m = 4; d = 30; `日付:${y} 年${m} 月${d} 日` Enter    ← 日付の編集
`日付:2024 年 4 月 30 日`    ← 編集結果
```

この例は先に解説した通りプレースホルダへの値の埋め込みである。ここで次のようなタグ関数 `tg` を定義⁴⁶ する。

例. タグ関数 `tg` の定義（先の例の続き）

```
> function tg( s, ...v ) { Enter    ← 関数定義の開始
...     return `Date:${v[0]}/${v[1]}/${v[2]}` Enter
... } Enter    ← 関数定義の終了
undefined    ← 定義処理の結果
```

この関数 `tg` をテンプレートリテラルの先頭に記述する例（タグ付きテンプレートリテラル）を次に示す。

例. タグ付きテンプレートリテラル（先の例の続き）

```
> y = 2024; m = 4; d = 30; tg`日付:${y} 年${m} 月${d} 日` Enter    ← 日付の編集
`Date:2024/4/30`    ← 編集結果
```

このように、関数 `tg` の第2引数以降にプレースホルダの値が配列⁴⁷ として受け取られ、関数内ではそれらを使って自由に文字列を編集することができる。

⁴⁶関数の定義に関しては、後の「3.10 関数の定義」（p.83）で解説する。

⁴⁷後の「3.6.1 配列：Array」（p.50）で解説する。

関数 `tg` の定義を次のように書き換えると、その働きがよく分かる。

例. タグ関数 `tg` の改造 (先の例の続き)

```
> function tg( s, ...v ) { Enter    ←関数定義の開始
...     console.log("文字列部分      :",s); Enter    ←第 1 引数の内容確認
...     console.log("プレースホルダ :",v); Enter    ←第 2 引数以降の内容確認
...     return `Date:${v[0]}/${v[1]}/${v[2]}`; Enter
... } Enter    ←関数定義の終了
undefined    ←定義処理の結果
```

この関数でタグ付きテンプレートリテラルを評価する。(次の例)

例. タグ付きテンプレートリテラル (先の例の続き)

```
> y = 2024; m = 4; d = 30; tg`日付:${y} 年${m} 月${d} 日` Enter    ←日付の編集
文字列部分      :   [ '日付:', '年', '月', '日' ]    ←関数 tg の第 1 引数の内容 (配列)
プレースホルダ  :   [ 2024, 4, 30 ]                  ←関数 tg の第 2 引数以降の内容 (配列)
'Date:2024/4/30'    ←編集結果
```

タグ関数の第 1 引数には元の文字列の純粋な文字列の部分の配列が、第 2 引数以降にはプレースホルダーの値の配列が受け取られていることがわかる。タグ関数内ではそれらを用いて文字列を編集し、それを戻り値とする。

3.5.3.10 文字列を数値に変換する方法

文字列として表現された数値を実際の数値 (number) に変換する方法について解説する。

書き方: `parseInt(文字列)`

「文字列」の内容を整数として解釈 (小数点以下は切り捨て) して数値に変換したものを返す。整数に変換できない文字列を与えた場合は `NaN` を返す。(空白文字も `NaN` となる)「文字列」には基数表現の接頭辞が許される。ただし指数表現は許されず、「e」以降の記述が無視される。

例. 文字列表現の整数を数値に変換する

```
> parseInt("255") Enter    ←変換処理
255              ←変換結果
> parseInt("0xff") Enter    ←基数表現の接頭辞あり
255              ←変換結果
> parseInt("3e2") Enter    ←指数表現は…
3                ←「e」以降が無視される (正しくない値)
> parseInt("abc") Enter    ←整数として解釈できないものは…
NaN
```

文字列として表現された浮動小数点数を実際の数値 (number) に変換するには `parseFloat` を用いる。

書き方: `parseFloat(文字列)`

「文字列」の内容を数値として解釈 (小数点以下も有効) して数値に変換したものを返す。数値に変換できない文字列を与えた場合は `NaN` を返す。(空白文字も `NaN` となる)「文字列」には指数表現が許される。ただし、基数表現の接頭辞は許されない。

例. 文字列表現の数値を数値に変換する

```
> parseFloat("3.14") Enter    ←変換処理
3.14              ←変換結果
> parseFloat("0.314e1") Enter    ←指数表現も
3.14              ←正しく変換処理される
> parseFloat("0xff") Enter    ←基数表現の接頭辞を付けると…
0                ←正しく変換されない
> parseFloat("abc") Enter    ←数値として解釈できないものは…
NaN
```

`Number` コンストラクタは文字列表現の整数、浮動小数点数を柔軟に変換する。

書き方： `Number(文字列)`

「文字列」の内容を数値として解釈して数値に変換したものを返す。数値に変換できない文字列を与えた場合は NaN を返す。上記 `parseInt`, `parseFloat` の機能を概ね含んでいるが、空白文字は 0 に変換する。

例. `Number` コンストラクタによる変換

```
> Number("255") 
255
> Number("0xff") 
255
> Number("3.14") 
3.14
> Number("0.314e1") 
3.14
> Number("abc")  数値として解釈できないものは…
NaN
```

例. 空白文字を数値に変換する試み

```
> parseInt(" ") 
NaN
> parseFloat(" ") 
NaN
> Number(" ") 
0      ← Number コンストラクタの場合のみ 0 となる
```

3.5.4 シンボル

シンボルは `Symbol` 関数が生成する一意の値である。シンボルはオブジェクトのキーなどに使用される特殊な値であり、文字列とは全く異なる。

書き方： `Symbol(シンボル名)`

「シンボル名」は省略できる。また、同じシンボル名で複数のシンボルを生成できるが、それらは互いに別のものである。

例. シンボルの生成と比較（シンボル名なし）

```
> s1 = Symbol(); s2 = Symbol(); s1 == s2 
false
```

例. シンボルの生成と比較（シンボル名あり）

```
> s1 = Symbol("sn"); s2 = Symbol("sn"); s1 == s2 
false
```

比較演算子 `==` は両辺の値が同じかどうかを調べるもので、この例からわかるように、シンボル名の有無に限らず、生成されたシンボルはそれぞれ別のものであることがわかる。

3.5.5 値の比較

値の比較のための演算子を表 21 に示す。

文字列の大小関係は文字コード⁴⁸に基づく。また `==` による比較では両辺の値の型が異なっても `true` となる場合がある。例えば、数値（`number`）、`bigint` 型、文字列を `==` で比較する場合、両辺が同じ値であると解釈できる場合に `true` となることがある。

⁴⁸Unicode のコードポイントの値

表 21: 2つの値の比較

記述	解説
<code>x < y</code>	y が x より大きい場合に true, それ以外の場合は false.
<code>x <= y</code>	y が x 以上の場合に true, それ以外の場合は false.
<code>x > y</code>	y が x より小さい場合に true, それ以外の場合は false.
<code>x >= y</code>	y が x 以下の場合に true, それ以外の場合は false.
<code>x == y</code>	x と y が等しい場合に true, それ以外の場合は false. x,y の型が互いに異なっている場合でも「同じ値」と見なせる場合は true となることがある.
<code>x === y</code>	x と y が等しい場合に true, それ以外の場合は false. x,y の型が互いに異なる場合は false.

例. ==による比較で true となるケース (一部)

```
> 123 == 123n Enter
true
> 123 == "123" Enter
true
> 123n == "123" Enter
true
> 0 == "" Enter
true
> 0n == "" Enter
true
```

```
> 1 == true Enter
true
> false == 0 Enter
true
> false == "0" Enter
true
> false == "" Enter
true
```

このように == による比較は安全でない場合もあるので, === を用いる方が良い.

例. ===による比較

```
> 123 === 123n Enter
false
> 123 === "123" Enter
false
> 0 === false Enter
false
```

3.5.6 値の型に関すること

値の型は typeof 演算子で調べることができる.

書き方: `typeof 値`

「値」の型を文字列の形で返す. これまでに解説した数値, 長整数, 論理値, 文字列, シンボルなどは JavaScript における最も基本的なデータでありプリミティブ型と呼ばれる範疇に分類される. (表 22)

表 22: プリミティブ型 (主なもの)

値	型	値	型	値	型
数値	'number'	長整数	'bigint'	論理値	'boolean'
文字列	'string'	シンボル	'symbol'	未定義	'undefined'

値の型を typeof 演算子で調べる様子を示す.

例. 各種の値の型の調査

```
> typeof 3 Enter
'number'
> typeof 3n Enter
'bigint'
> typeof true Enter
'boolean'
```

```
> typeof "abc" Enter
'string'
> s = Symbol(); typeof s Enter
'symbol'
> typeof undefined Enter
'undefined'
```


プリミティブ型の値には、は上に挙げたもの以外にも `null` がある。実際のプログラミングにおいて `null` を用いるケースは多くはないが、「データが無い」という状態を明示的に表現する場合に用いることがある。

例. `null`

```
> a = null  ← null は値として  
null      ←扱うことができる  
  
> typeof null  ← null の型を調べる  
'object'  ←このような型
```

データの型をより詳細に調べるための別の方法があるが、それに関しては後述する。

3.6 データ構造

ここでは、複数の値をまとめて保持することができる**データ構造**について解説する。

3.6.1 配列：Array

複数の値を 1 次元の並びとして保持するデータ構造に**配列**（Array）がある。配列は要素をコンマ「,」で区切って並べたものを角括弧「[]」で括って記述⁴⁹するか、Array コンストラクタで生成する。

書き方 (1)： `Array(要素 1, 要素 2, …)`

書き方 (2)： `Array(長さ)`

「要素 1」、「要素 2」… を要素とする配列を生成する。要素の順序は生成時に与えた順序となる。与える要素の型に特に制限はなく、別の配列を要素として与えることも可能⁵⁰である。コンストラクタの引数に整数値を 1 つだけ与えると、その値が示す個数の**空要素**を持つ配列を生成する。

例. 配列の生成

```
> a = [1,2,3] Enter    ←配列リテラルとして直接記述
[ 1, 2, 3 ]      ←配列が得られている
> a = Array(1,2,3) Enter    ← Array コンストラクタによる生成
[ 1, 2, 3 ]      ←上と同じ結果
```

《出力の形式に関する留意事項》

Web ブラウザのコンソールで上記の配列を表示すると若干体裁が異なる。V8 エンジン搭載の Web ブラウザのコンソールの場合は「(3) [1, 2, 3]」と、Mozilla SpiderMonkey の場合は「Array(3) [1, 2, 3]」と表示される。

ターミナル環境下の Node.js で長い配列（要素数の多い配列）を表示すると自動的に改行が挿入される。例えば [1,2,3,4,5,6,7,8,9,10,11,12] は

```
[
  1,   2, 3, 4,   5,
  6,   7, 8, 9, 10,
  11, 12
]
```

などと表示される。

配列の要素には**インデックス**を指定してアクセスする。その際、インデックスを角括弧「[]」で括ったものを配列の直後に記述する。

書き方： `配列 [インデックス]`

「インデックス」で指定した位置の要素にアクセスする。先頭要素のインデックスは 0である。

例. 配列の要素の参照（先の例の続き）

```
> a[0] Enter    ←配列の先頭要素の参照
1      ←値が得られている
> a[2] Enter    ←先頭から 3 番目（インデックス=2）の要素の参照
3      ←値が得られている
```

配列の要素は事後に変更することができる。（次の例）

例. 既存の配列の要素の変更（先の例の続き）

```
> a[2] = "新たな値" Enter    ←先頭から 3 番目（インデックス=2）の要素の変更
'新たな値'          ←新たな値
> a Enter      ←内容確認
[ 1, 2, '新たな値' ]    ←変更された配列
```

⁴⁹配列リテラル

⁵⁰他のプログラミング言語におけるリストと同等の取り扱いが可能である。

要素の存在しない位置のインデックスを指定して配列にアクセスすることができる。

例. 要素の存在しない位置にアクセス（先の例の続き）

```
> a[3]  ←要素の存在しない位置を参照すると…
undefined          ←エラーは発生しない
> a[3] = 4  ←要素の存在しない位置に代入すると…
4                  ←代入された値
> a  ←内容確認
[ 1, 2, '新たな値', 4 ] ←変更された配列（配列が拡張されている）
> a.length  ←要素数の確認
4                  ←増えている
```

この例にあるように、**配列の要素数**は length プロパティから得られる。

次に、具体的な要素を与えずに長さのみを与えて配列を生成するケースについて考える。

例. 長さのみを指定して配列を生成

```
> a = Array(10)  ←長さ 10 の配列を生成
[ <10 empty items> ] ←このような内容の配列が得られる
> a.length  ←要素数の確認
10                  ←長さは 10
```

具体的な要素が与えられていない状態ではこの例のように配列の内容が

<長さ empty items>

となる。この形式は Google 製 V8 エンジンの場合で、Mozilla の SpiderMonkey の場合は

<長さ empty slots>

となる。これは要素が存在しないのではなく、指定された個数の空要素が存在していることを意味する。先の例で得られた配列に具体的な要素を与える例を次に示す。

例. 空要素の部分に要素を与える試み（先の例の続き）

```
> a[0] = 0; a  ←インデックス 0 の位置に要素を与える
[ 0, <9 empty items> ] ←配列の内容
> a[3] = 3; a  ←インデックス 3 の位置に要素を与える
[ 0, <2 empty items>, 3, <6 empty items> ] ←配列の内容
> a[9] = 9; a  ←インデックス 9 の位置に要素を与える
[ 0, <2 empty items>, 3, <5 empty items>, 9 ] ←配列の内容
```

3.6.1.1 空配列

空配列（長さ 0 の配列）は [] もしくは Array() で生成できる。

例. 空配列の生成

```
> a = []  ←空配列
[]
> a[100] = 100; a  ←もちろん拡張可能
[ <100 empty items>, 100 ]
```

3.6.1.2 要素の追加

push メソッドで配列の末尾に要素を追加することができる。

書き方： 配列.push(要素 1, 要素 2, …)

「配列」の末尾に「要素 1」, 「要素 2」, … を順番に追加する。このメソッドは「配列」自体を変更する。

例. 配列末尾への要素の追加

```
> a = [1]  ←配列の作成  
[ 1 ]  
> a.push(2,3,4)  ←上記配列の末尾に 2,3,4 を順番に追加  
4 ←追加後の元の配列の要素数が返される  
> a  ←元の配列を確認  
[ 1, 2, 3, 4 ] ←元の配列が変更されている
```

unshift メソッドで配列の先頭に要素を追加することができる.

書き方: 配列.unshift(要素 1, 要素 2, ...)

「配列」の先頭に「要素 1」, 「要素 2」, ... を追加する. このメソッドは「配列」自体を変更する.

例. 配列先頭への要素の追加 (追加順序に注意すること)

```
> a = [4]  ←配列の作成  
[ 4 ]  
> a.unshift(1,2,3)  ←上記配列の先頭に 2,3,4 を追加  
4 ←追加後の元の配列の要素数が返される  
> a  ←元の配列を確認  
[ 1, 2, 3, 4 ] ←元の配列が変更されている
```

3.6.1.3 要素の削除

pop メソッドで配列の末尾の要素を削除することができる.

書き方: 配列.pop()

「配列」の末尾の要素を削除して, その要素を返す. 空配列に対してこのメソッドを実行すると undefined を返す.

例. 配列の末尾の要素の取り出しと削除

```
> a = [1,2]  ←配列の作成  
[ 1, 2 ]  
> a.pop()  ←配列の末尾の要素の取り出しと削除  
2 ←取り出した要素が返される  
> a.pop()  ←再度 pop  
1 ←戻り値  
> a  ←2度 pop した後の配列の確認  
[] ←空配列  
> a.pop()  ←それに対して pop すると…  
undefined ←戻り値
```

shift メソッドで配列の先頭の要素を削除することができる.

書き方: 配列.shift()

「配列」の先頭の要素を削除して, その要素を返す. 空配列に対してこのメソッドを実行すると undefined を返す.

例. 配列の末尾の要素の取り出しと削除

```
> a = [1,2]  ←配列の作成  
[ 1, 2 ]  
> a.shift()  ←配列の先頭の要素の取り出しと削除  
1 ←取り出した要素が返される  
> a.shift()  ←再度 shift  
2 ←戻り値  
> a  ←2度 shift した後の配列の確認  
[] ←空配列  
> a.shift()  ←それに対して shift すると…  
undefined ←戻り値
```

応用) push, pop, unshift, shift を用いると FIFO や LIFO が実現できる.

3.6.1.4 配列の連結

concat メソッドで配列を連結することができる。

書き方： 配列.concat(配列 1, 配列 2, …)

「配列」の後ろに「配列 1」, 「配列 2」, … を連結したものを返す。元の配列は変更されない。

例. 配列の連結

```
> a = [1,2,3] Enter    ←配列の作成
[ 1, 2, 3 ]
> a.concat( [4,5,6], [7,8,9] ) Enter    ←上記配列に引数の配列を連結
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]    ←連結結果が返される
> a Enter    ←元の配列は
[ 1, 2, 3 ]    ←変更されない
```

3.6.1.5 部分配列

slice メソッドで部分配列を取得することができる。

書き方： 配列.slice(s, e)

「配列」のインデックス位置「s」から「e-1」までの範囲を返す。元の配列は変更されない。

例. 部分配列

```
> a = [0,1,2,3,4,5,6,7] Enter    ←配列の作成
[ 0, 1, 2, 3, 4, 5, 6, 7 ]
> a.slice(2,6) Enter    ←インデックス位置の2から6未満(5)までの部分を取得
[ 2, 3, 4, 5 ]    ←得られた部分配列
```

slice メソッドの第2引数を省略すると末尾までの部分配列が得られる。また引数を全く与えない場合は元の配列の複製を返す。

例. slice メソッドの引数の省略（先の例の続き）

```
> a.slice(5) Enter    ←第2引数の省略
[ 5, 6, 7 ]    ←インデックス位置の5から末尾まで
> a.slice() Enter    ←引数なし
[ 0, 1, 2, 3, 4, 5, 6, 7 ]    ←元の配列の複製が得られる
```

3.6.1.6 配列の編集

splice メソッドで配列の内容を編集することができる。

書き方： 配列.splice(位置, 削除長, 要素 1, 要素 2, …)

「配列」の指定したインデックスの「位置」から指定した「削除長」の範囲を削除して、「位置」以降に「要素 1」, 「要素 2」, … を挿入する。このメソッドは元の「配列」を変更し、削除した範囲の配列を返す。

■ 配列の指定範囲の削除

splice メソッドの引数に挿入要素を与えずに（第3引数以降を省略して）実行すると、配列の指定した範囲を削除するのみである。

例. 指定範囲の削除

```
> a = [0,1,2,3,4,5,6,7,8,9,"a","b","c"] Enter    ←配列の作成
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'a', 'b', 'c' ]
> a.splice(7,3) Enter    ←インデックス位置7から要素を3つ削除する
[ 7, 8, 9 ]    ←削除したものが返される
> a Enter    ←元の配列を確認
[ 0, 1, 2, 3, 4, 5, 6, 'a', 'b', 'c' ]    ←削除済み
```

■ 削除と挿入

splice メソッドの引数に挿入要素を与えて実行する例を示す。

例. 削除と挿入（先の例の続き）

```
> a.splice(5,2,"five","six","seven","eight") Enter    ←削除と挿入挿入
[ 5, 6 ]                                     ←削除したものが返される
> a Enter    ←元の配列を確認
[ 0, 1, 2, 3, 4, 'five', 'six', 'seven', 'eight', 'a', 'b', 'c' ]    ←編集済み
```

3.6.1.7 配列の反転

reverse メソッドで配列の要素の順序を反転することができる。

書き方： 配列.reverse()

「配列」の要素の順序を反転する。このメソッドは元の配列を変更し、その配列自体を返す。

例. 配列の反転

```
> a = [1,2,3,4,5] Enter    ←配列の作成
[ 1, 2, 3, 4, 5 ]
> a.reverse() Enter    ←反転処理
[ 5, 4, 3, 2, 1 ]    ←反転結果
> a Enter    ←元の配列を確認
[ 5, 4, 3, 2, 1 ]    ←反転されている
```

3.6.1.8 要素の整列

sort メソッドで配列の要素の順序を整列することができる。

書き方 (1)： 配列.sort()

「配列」の要素の順序を整列する。このメソッドは元の配列を変更し、その配列自体を返す。引数を省略した場合は昇順に整列される。

例. 要素の順序の整列

```
> a = [3,1,5,2,4] Enter    ←配列の作成
[ 3, 1, 5, 2, 4 ]
> a.sort() Enter    ←整列処理
[ 1, 2, 3, 4, 5 ]    ←整列結果
> a Enter    ←元の配列を確認
[ 1, 2, 3, 4, 5 ]    ←整列されている
```

▲注意▲

sort メソッドは、要素を文字列（UTF-16 のコードの並び）と見做して大小関係を判定する。従って、要素が数値の場合は注意すること。

例. 数値要素の場合の整列

```
> a = [1000,2,400] Enter    ←数値配列の作成
[ 1000, 2, 400 ]
> a.sort() Enter    ←整列処理
[ 1000, 2, 400 ]    ←整列されていないように見える
```

この例では、配列の要素が整列されていないように見えるが、要素が文字列

"1000", "2", "400"

と見做されたため、配列は元々昇順である。

要素の大小関係の判定方法を明に指示するには、sort メソッドの引数に大小関係を判定する関数を与える。

書き方 (2)： 配列.sort(判定関数)

「判定関数」の戻り値に従って「配列」を整列する。判定関数は、配列内の隣接する要素 x, y から次のような戻り値を返すもの（表 23）として定義する。

表 23: 整列順の判定関数

大小関係の定義	整列処理	戻り値
$x < y$	$x \rightarrow y$	負
$x === y$	実行せず	0
$x > y$	$y \rightarrow x$	正

表 23 に従って、数値の大小から整列処理を制御する関数 `CmpFn` を定義し、それを用いて、数値（number 型）の大きさの昇順に整列処理を行う例を次に示す。

例. 数値要素の整列（先の例の続き）

```
> function CmpFn(x,y) { return x-y; }  ←判定関数の定義
undefined                               ←上の文の実行結果
> a.sort(CmpFn)  ←判定関数を用いて整列処理
[ 2, 400, 1000 ] ←数値の昇順として整列された
```

関数の定義に関しては、後の「3.10 関数の定義」（p.83）で解説する。

3.6.1.9 指定した要素を配列内に並べる方法

`fill` メソッドで配列内に指定した要素を並べることができる。

書き方： 配列.`fill`(値)

既存の「配列」の全ての要素を「値」にする。このメソッドは対象の「配列」の内容を変更し、その配列を返す。

例. 配列の要素を全て 7 にする

```
> a = new Array(10)  ←長さ 10 の配列の生成
[ <10 empty items> ] ←得られた配列
> a.fill(7); a  ←その配列を 7 で満たして確認
[ 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 ] ←結果
```

配列内に要素を並べる際の開始位置と終了位置を指定することができる。

書き方： 配列.`fill`(値, s, e)

「配列」内のインデックス位置 `s` から `e-1` までの範囲を「値」で満たす。第 3 引数 `e` を省略するとインデックス `s` から末尾までの範囲を「値」で満たす。

例. インデックス範囲を指定した `fill` の処理

```
> a = [0,1,2,3,4,5,6,7,8,9]  ←長さ 10 の配列の生成
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
> a.fill( 99, 3, 6 )  ←インデックス位置 3 ~ 6-1 の範囲を 99 にする
[ 0, 1, 2, 99, 99, 99, 6, 7, 8, 9 ] ←結果
```

例. インデックス範囲を指定した `fill` の処理（終了位置の省略）

```
> a = [0,1,2,3,4,5,6,7,8,9]  ←長さ 10 の配列の生成
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
> a.fill( 99, 6 )  ←インデックス位置 6 以降を 99 にする
[ 0, 1, 2, 3, 4, 5, 99, 99, 99, 99 ] ←結果
```

3.6.1.10 要素の存在や位置の調査

`includes` メソッドによって、配列の中に指定した要素が存在するかどうかを調べることができる。

書き方： 配列.`includes`(要素)

「配列」に「要素」が存在すれば `true` を、無ければ `false` を返す。

例. 要素の存在検査

```
> a = ["a","b","c","d","c","b","a","z"] Enter    ←配列の生成
[ 'a', 'b', 'c', 'd', 'c', 'b', 'a', 'z' ]
> a.includes( "d" ) Enter    ←配列の要素に "d" があるか?
true                                     ←ある
> a.includes( "x" ) Enter    ←配列の要素に "x" があるか?
false                                    ←ない
```

indexOf メソッドによって、配列内の指定した要素のインデックス位置を求めることができる。存在しない要素の位置を求めようとすると負値 (-1) を返す。

書き方： 配列.indexOf(要素)

「配列」の先頭から「要素」を探し、最初にそれが見つかったインデックス位置を返す。

例. 要素の位置を調べる (先の例の続き)

```
> a.indexOf("c") Enter    ←配列内の要素 "c" のインデックス位置を求める
2
> a.indexOf("x") Enter    ←配列内に存在しない要素 "x" のインデックス位置を求める試み
-1                                     ←存在しないことを意味する
```

配列 a の中で最初に "c" が現れるインデックス位置である 2 が得られていることがわかる。要素 "c" はインデックスが 4 の位置にも存在するがそれは無視される。

indexOf メソッドの第 2 引数に要素の探索の開始位置を指定することもできる。

例. 探索開始位置の指定 (先の例の続き)

```
> a.indexOf("c",3) Enter    ←インデックス位置 3 以降で要素 "c" の位置を求める
4                                     ←2 番目の "c" のインデックス位置
```

要素の探索を逆順に行う lastIndexOf もある。

書き方： 配列.lastIndexOf(要素)

「配列」の末尾から先頭にかけて「要素」の位置を調べ、最初に見つかったインデックスを返す。要素が見つからなければ負値 (-1) を返す。また、第 2 引数に探索開始位置のインデックスを与えることもできる。

例. 逆順の探索 (先の例の続き)

```
> a.lastIndexOf("c") Enter    ←配列の末尾から逆順に要素 "c" を探す
4
> a.lastIndexOf("c",3) Enter    ←インデックス位置 3 から逆順に要素 "c" を探す
2
```

3.6.1.11 配列の並びを表す文字列の取得

toString メソッドによって、配列の要素の並びを表す文字列を取得することができる。

書き方： 配列.toString()

例. 配列の並びを表す文字列 (先の例の続き)

```
> a.toString() Enter    ←配列の要素の並びを文字列に変換
'a,b,c,d,c,b,a,z'      ←結果
> [].toString() Enter    ←空配列を文字列に変換
''                       ←結果 (空文字列)
```

3.6.1.12 全要素に対する条件検査

every メソッドを用いると、配列の全ての要素が指定した条件を満たすかどうかを検査することができる。

書き方： 配列.every(関数)

与えられた 1 つの値に対して true か false を返す「関数」を「配列」の全要素に対して実行し、全ての場合において true となる場合に true を、1 つでも false となる場合は false を返す。

次に示すような、偶数かどうかを判定する関数 isEven を考える。

例. 与えられ多数が偶数かどうかを判定する関数の定義

```
> function isEven(n) { return n%2==0 ? true : false } Enter    ←関数の定義
undefined      ←関数定義の完了
```

この関数 isEven は、第 1 引数に与えられた値が偶数ならば true を、そうでなければ false を返す。

例. 関数 isEven による判定処理（先の例の続き）

```
> isEven(2) Enter    ← 2 は
true        ←偶数である
> isEven(3) Enter    ← 3 は
false       ←偶数ではない
```

関数の定義に関しては、後の「3.10 関数の定義」（p.83）で解説する。また、上記関数の内部における条件判定の方法に関しては「3.8 条件分岐」（p.75）で解説する。

上で定義した関数 isEven を配列の全ての要素に対して実行し、全て偶数かどうかを every メソッドで検査する例を次に示す。

例. every による検査（先の例の続き）

```
> [0,2,4,6,8].every( isEven ) Enter    ←全ての要素は偶数か？
true
> [0,2,4,6,7].every( isEven ) Enter    ←全ての要素は偶数か？
false
```

■ 条件を満たす要素が 1 つでも含まれるかどうかの検査

配列の中に、指定した条件を満たす要素が少なくとも 1 つあるかを検査するには some メソッドを用いる。

書き方： 配列.some(関数)

与えられた 1 つの値に対して true か false を返す「関数」を「配列」の要素に対して順次実行し、true となる要素がある場合に true を、true となるものが全く無ければ false を返す。

例. 配列の中に 1 つでも偶数があるかどうかを検査（先の例の続き）

```
> [1,3,4,7,9].some( isEven ) Enter    ← 1 つでも偶数の要素があるか？
true
> [1,3,5,7,9].some( isEven ) Enter    ← 1 つでも偶数の要素があるか？
false
```

3.6.1.13 条件を満たす要素の探索

find メソッドを用いると、指定した条件を満たす要素を配列の中から探索することができる。

書き方： 配列.find(関数)

与えられた 1 つの値に対して true か false を返す「関数」を「配列」の要素に対して順次実行し、true となる最初の要素を返す。true となる要素が 1 つも無ければ undefined を返す。

先に例示した関数 isEven を用いて、最初に見つかる偶数の要素を取得する例を示す。

例. 最初の偶数要素を取得する（先の例の続き）

```
> [1,3,4,7,9].find( isEven ) Enter    ←最初の偶数要素を求める
4
> [1,3,5,7,9].find( isEven ) Enter    ←奇数ばかりの配列に対して実行する試み
undefined
```

条件を満たす最初の要素のインデックス位置を取得するには findIndex メソッドを使用する。

例. 最初の偶数要素の位置を取得する（先の例の続き）

```
> [1,3,4,7,9].findIndex( isEven ) Enter    ←最初の偶数要素のインデックスを求める
2
> [1,3,5,7,9].findIndex( isEven ) Enter    ←奇数ばかりの配列に対して実行する試み
-1      ←条件を満たす要素がなければ負値 (-1) を返す
```

3.6.1.14 特殊なプロパティ

配列は、**要素ではない**プロパティを保持することができる。(次の例)

例. 要素ではないプロパティ

```
> a = [1,2,3,4,5];  ←配列の作成
[ 1, 2, 3, 4, 5 ]
> a["name"] = "田中 三郎";  ←"name" プロパティの追加
'田中 三郎'
> a  ←内容確認
[ 1, 2, 3, 4, 5, name: '田中 三郎' ] ←"name" プロパティが保持されている
```

ドット表記でも同様のことができる。(次の例)

例. ドット表記でプロパティにアクセスする(先の例の続き)

```
> a.age = 23;  ←"name" プロパティの追加
23
> a  ←内容確認
[ 1, 2, 3, 4, 5, name: '田中 三郎', age: 23 ] ←内容
```

この例のように、配列に追加された name, age といったものは配列の要素ではない。そのことを次の例で確かめる。

例. 要素の確認(先の例の続き)

```
> a.length  ← name, age は要素数として
5 ←カウントされない
> a[5]  ←インデックスを付けてアクセス
undefined ←することができない
> a.name  ←しかし、値は
'田中 三郎' ←保持されている
> a["age"]  ←このプロパティも
23 ←同様
```

このようなプロパティは後に説明する**オブジェクト**の性質による。(配列はオブジェクトを基に実装されている)

3.6.2 オブジェクト：Object

オブジェクトの要素は**名前(キー)**と**値**をコロン「:」で結び付けた組(**プロパティ**)であり、オブジェクトはそれら要素を波括弧 { } で括ったデータ構造である。また、1つのオブジェクトにおいて要素の名前(プロパティのキー)は一意である。

次に波括弧 { } の表記(リテラル)でオブジェクトを作る例を示す。

例. 要素の名前(プロパティ名)は一意

```
> d = {"apple":"りんご", "orange":"みかん", "orange":"蜜柑"}  ←重複する名前があると
{ apple: 'りんご', orange: '蜜柑' } ←後から記述されたものが有効である
```

上の例において、キーに使用するものが数値の表記や特殊なものでなければ引用符を省略することができる。(次の例)

例. 上と同じオブジェクトの作成

```
> d = {apple:"りんご", orange:"みかん", orange:"蜜柑"}  ←引用符を付けずにキーを記述
{ apple: 'りんご', orange: '蜜柑' }
```

空白を含んだキーを使用する場合は引用符を付けること。また特殊記号を含んだキーの場合も引用符を付けるべきである。

オブジェクトは**連想配列**と呼ばれることもあり、名前を基に値にアクセスすることができる⁵¹。その場合、オブジェクトの後ろに角括弧 [] で名前を括ったものを付ける。

⁵¹ 他のプログラミング言語でも同様のデータ構造を持つものがあり、**辞書**と呼ぶことがある。

例. 名前を指定したオブジェクトへのアクセス（先の例の続き）

```
> d["orange"]  ←名前を指定して値を参照
'蜜柑' ←値
> d["orange"] = "みかん"  ←代入により値を変更可能
'みかん' ←変更後の値
> d  ←内容確認
{ apple: 'りんご', orange: 'みかん' } ←オブジェクトの中身
```

この場合のキーは文字列型で与えること.

代入によって新たなプロパティを登録することもできる.

例. 新たな要素の登録（先の例の続き）

```
> d["grape"] = "ぶどう"  ←代入により新たなものを追加可能
'ぶどう' ←値
> d  ←内容確認
{ apple: 'りんご', orange: 'みかん', grape: 'ぶどう' } ←オブジェクトの中身
```

存在しないプロパティを参照すると, 値は `undefined` となる.

例. 存在しないプロパティの参照（先の例の続き）

```
> d["banana"]  ←"banana"の名前は存在しない
undefined
```

要素へのアクセスには角括弧以外にもドット表記が使用できる.

例. ドット表記によるプロパティへのアクセス（先の例の続き）

```
> d.apple  ←ドット表記で apple の名前を参照
'りんご' ←値が得られている
> d.banana = "バナナ"  ←ドット表記で要素の新規追加も可能
'バナナ' ←追加した要素の値
> d  ←内容確認
{ apple: 'りんご', orange: 'みかん', grape: 'ぶどう', banana: 'バナナ' }
```

このように, ドット表記でプロパティのキーを指定する場合は引用符で括らない.

ドット表記でキーを指定してプロパティにアクセスする方法は記述上の簡便性があるが, 特殊な名前のキーを使用する場合は角括弧の形式を取る必要がある. また, 変数に格納された文字列型のキーを使う場合はやはり角括弧の形式を取る必要がある.

3.6.2.1 要素の削除

オブジェクトの要素の削除（プロパティの削除）には `delete` 演算子を使用する.

例. プロパティの削除（先の例の続き）

```
> delete d.grape  ←grape のプロパティの削除
true
> d  ←内容確認
{ apple: 'りんご', orange: 'みかん', banana: 'バナナ' } ←オブジェクトの中身
```

3.6.2.2 プロパティのキーに使用できるデータ

プロパティのキーには文字列かシンボルが使用できる. 次に示す例は, キーに数値を使用するケースである.

例. キーに数値を使用するケース

```
> d = 1:2, 2:4, 3:6  ←キーに数値を与えると
{ '1': 2, '2': 4, '3': 6 } ←文字列に変換される
```

値の部分は文字列以外の型のデータを与えることができる.

3.6.2.3 空オブジェクト

空オブジェクトはリテラルとして と記述する。また Object コンストラクタで生成することもできる。

例. 空オブジェクト

```
> a = {} Enter    ←空オブジェクトの作成
{}
> a = new Object() Enter    ← Object コンストラクタでも生成可能 (new 演算子を付ける)
{}

```

3.6.2.4 オブジェクトから配列への変換

Object.entries でオブジェクトを配列に変換することができる。

書き方： Object.entries(オブジェクト)

「オブジェクト」の全プロパティのキーと値を組にした配列を要素とする配列を返す。

例. オブジェクトを配列に変換

```
> d = {"apple": "りんご", "orange": "みかん", "grape": "ぶどう"} Enter ←オブジェクトの作成
{ apple: 'りんご', orange: 'みかん', grape: 'ぶどう' }
> Object.entries( d ) Enter    ←上のオブジェクトを配列に変換
[[ 'apple', 'りんご' ], [ 'orange', 'みかん' ], [ 'grape', 'ぶどう' ]] ←変換結果

```

オブジェクトのキーのみの配列は Object.keys で取得できる。

書き方： Object.keys(オブジェクト)

「オブジェクト」の全てのキーを要素とする配列を返す。

例. オブジェクトの全キーの配列を取得する (先の例の続き)

```
> Object.keys( d ) Enter    ←キーの配列を取得
[ 'apple', 'orange', 'grape' ]

```

3.6.2.5 オブジェクトの要素の個数を求める方法

オブジェクトの要素の個数 (プロパティの個数) を直接求めるメソッドはないが、次のような方法で簡単に算出することができる。

算出方法： Object.keys(オブジェクト).length

これは、オブジェクトのキーの配列を取得する方法の応用である。

例. オブジェクトの要素の個数を求める (先の例の続き)

```
> d Enter    ←先の例で作成したオブジェクト
{ apple: 'りんご', orange: 'みかん', grape: 'ぶどう' }
> Object.keys(d).length Enter    ←要素の個数を算出
3                                ←要素数

```

3.6.2.6 計算されたプロパティ名

オブジェクトのリテラルを記述する際、プロパティ名には固定されたリテラル以外にも、**計算されたプロパティ名** (computed property name) を使用することができる。

例. 計算されたプロパティ名

```
> var key1 = "apple", key2 = "orange"; Enter    ←このような変数を用意
undefined                                ← var による代入処理の結果
> d = { [key1]: "りんご", [key2]: "みかん" } Enter    ← [] で括ってキーを与える
{ apple: 'りんご', orange: 'みかん' }    ←変数名がキーとして展開されている

```

このように、角括弧 [] で括ったキーを与えると、その中の式の値が結果的なキーとなる。

重要)

オブジェクトには独自のメソッドを付与することができる。これに関しては、「3.10 関数の定義」(p.83) 内の「3.10.7 オブジェクトとメソッドの考え方」(p.91) で解説する。

3.6.3 Map オブジェクト

Map オブジェクトは先のオブジェクト (Object) と同様に**連想配列**として機能するものであり、**キー**に対する**値**を関連付けるデータ構造である。また、Map オブジェクトはキーに文字列やシンボル以外のデータを使用することができる。しかし、オブジェクトの場合と同様にキーは一意でなければならない。

Map オブジェクトは Map コンストラクタで生成する。

書き方： `new Map([[キー 1, 値 1], [キー 2, 値 2], …])`

「キー 1」、「キー 2」… にそれぞれ「値 1」、「値 2」… を関連付けた Map オブジェクトを生成する。コンストラクタの引数を省略すると空の Map オブジェクトを生成する。

例. Map オブジェクトの生成

```
> m = new Map([[1,2],[2,4],[3,6],[4,8]])  Enter    ← Map オブジェクトの生成
Map(4) { 1 => 2, 2 => 4, 3 => 6, 4 => 8 }      ←得られた Map オブジェクト
```

《出力の形式に関する留意事項》

処理系ごとに Map オブジェクトを表示する際の体裁が若干異なる。V8 エンジン搭載の Web ブラウザのコンソールやターミナル環境下の Node.js の場合は上記の例のような表示

Map(要素数) { キー 1 => 値 1, キー 2 => 値 2, … }

となる。また、Mozilla SpiderMonkey の場合は

Map { キー 1 → 値 1, キー 2 → 値 2, … }

となる。また、このような表記の Map オブジェクトのリテラルはない。

3.6.3.1 要素へのアクセス

get メソッドでキーに対する値を参照することができる。

書き方： `Map オブジェクト.get(キー)`

「Map オブジェクト」の「キー」に対する値を返す。「キー」に対応する値がない場合は undefined を返す。

例. キーに対する値の参照 (先の例の続き)

```
> m.get(1)  Enter    ←キー 1 に対する値を参照
2          ←値
> m.get(5)  Enter    ←存在しないキー 5 に対する値を参照
undefined   ←値
```

set メソッドでキーに対する値を設定することができる。

書き方： `Map オブジェクト.set(キー, 値)`

「Map オブジェクト」の「キー」に対する値を設定する。処理結果の「Map オブジェクト」を返す。

例. キーに対する値の設定 (先の例の続き)

```
> m.set(5,10)  Enter    ←キー 5 に対して値 10 を設定
Map(5) { 1 => 2, 2 => 4, 3 => 6, 4 => 8, 5 => 10 }  ←設定後の Map オブジェクト
```

3.6.3.2 要素の存在確認

has メソッドで要素の存在を確認することができる。

書き方： `Map オブジェクト.has(キー)`

「Map オブジェクト」に「キー」を持つ要素が存在すれば true を、存在しなければ false を返す。

例. 要素の存在確認 (先の例の続き)

```
> m.has(5)  Enter    ← 5 をキーに持つ要素があるか?
true        ←ある
> m.has(6)  Enter    ← 6 をキーに持つ要素があるか?
false       ←ない
```


3.6.3.3 要素の個数の調査

size プロパティから要素の個数が参照できる。

書き方： Map オブジェクト.size

「Map オブジェクト」の要素の個数が得られる。

例. 要素の存在確認（先の例の続き）

```
> m.size  Enter    ←要素の個数の参照  
5        ←現在 5 個
```

3.6.3.4 全要素の取出し

entries メソッドで全要素を取り出すことができる。

書き方： Map オブジェクト.entries()

「Map オブジェクト」の全要素を取り出し、イテレータとして返す。

例. 全要素の取出し（先の例の続き）

```
> e = m.entries()  Enter    ←全要素の取出し  
[Map Entries] { [ 1, 2 ], [ 2, 4 ], [ 3, 6 ], [ 4, 8 ] } ←イテレータ
```

イテレータは反復制御に使用するものであるが、Array.from で配列に変換することができる。

例. イテレータを配列に変換する（先の例の続き）

```
> Array.from(e)  Enter    ←配列に変換  
[ [ 1, 2 ], [ 2, 4 ], [ 3, 6 ], [ 4, 8 ] ]
```

Map オブジェクトの全要素のキーのみをイテレータとして取り出すには keys を、全要素の値のみをイテレータとして取り出すには values を使用する。

例. キーのみ、値のみの取出し（先の例の続き）

```
> k = m.keys()  Enter    ←全てのキーの取出し  
[Map Iterator] { 1, 2, 3, 4 } ←イテレータ  
> v = m.values()  Enter    ←全ての値の取出し  
[Map Iterator] { 2, 4, 6, 8 } ←イテレータ
```

これらも Array.from で配列に変換することができる。

イテレータを用いた反復処理に関しては後の「3.9.3.1 for ... of 文」(p.80) で解説する。

3.6.3.5 要素の削除

delete メソッドで要素を削除することができる。

書き方： Map オブジェクト.delete(キー)

「Map オブジェクト」に「キー」を持つ要素が存在すればそれを削除して true を返す。要素が存在しなければ false を返す。

例. 要素の削除（先の例の続き）

```
> m.delete(5)  Enter    ← 5 をキーに持つ要素を削除  
true          ←削除成功  
> m  Enter    ← Map オブジェクトの内容確認  
Map(4) { 1 => 2, 2 => 4, 3 => 6, 4 => 8 } ←現在の内容  
> m.delete(5)  Enter    ←存在しない要素を削除する試み  
false         ←削除できず
```

clear メソッドで全ての要素を削除することができる。

書き方： Map オブジェクト.clear()

「Map オブジェクト」を空にして undefined を返す。

例. 全ての要素の削除（先の例の続き）

```
> m.clear()  ← m を空にする
undefined
> m  ←内容確認
Map(0) {} ←空になった
```

3.6.3.6 ミュータブルなデータをキーに使用する際の注意

Map オブジェクトはキーとしてミュータブルなデータを使用することができるが、これに関してはいくつか注意しなければならないことがある。以下にいくつか例を示す。

例. 配列を Map のキーにする

```
> a = [1,2,3]; b = [1,2,4]; m = new Map([[a,11],[b,12]])  ← Map オブジェクトの作成
Map(2) { [ 1, 2, 3 ] => 11, [ 1, 2, 4 ] => 12 } ←得られた Map オブジェクト
```

これは配列をキーに使用して Map オブジェクトを作成する例である。得られた Map オブジェクトに対してキーを指定して値を取得することを試みる。（次の例）

例. キーを指定して値を参照する試み（先の例の続き）

```
> m.get([1,2,3])  ←配列をキーにして参照を試みると…
undefined ←意外な結果に見える
> m.get([1,2,4])  ←このキーでも…
undefined ←意外な結果に見える
```

正しく値が得られていないように見えるが次の例では正しく値が得られる。

例. 正しく値が得られるケース（先の例の続き）

```
> m.get(a)  ←配列作成時の変数 a をキーにして参照を試みると
11 ←正しく値が得られる
> m.get(b)  ←配列作成時の変数 b をキーにして参照を試みると
12 ←正しく値が得られる
```

このことから、Map オブジェクト作成時に与えたキー（変数 a）と、参照時に与えたキー（配列リテラル [1,2,3]）は異なるものであることがわかる。従って、Map オブジェクトを参照する際には、生成時と**同一のデータである**キーを使用しなければならない。また、数値や文字列といった**プリミティブ型**の値はリテラル表記のキーを与えても良い。

演習課題.

上の例の実行後に `b[2] = 3` として変数 b の配列を変更すると、b の内容は `[1, 2, 3]` となる。その状態で変数 m の値を参照すると、

```
Map(2) { [ 1, 2, 3 ] => 11, [ 1, 2, 3 ] => 12 }
```

となり、重複するキーを持つように見える。これは何を意味するのか考えよ。

またこの状態で、変数名をキーにして Map オブジェクト m を参照すると次のように正しく値が得られる。

```
> m.get(a) 
11
> m.get(b) 
12
```

この理由についても考えよ。

3.6.4 Set オブジェクト

Set オブジェクトは一意の値を要素として持つデータ構造であり、Set コンストラクタで生成することができる。

書き方： `new Set(初期値)`

「初期値」には配列を始めとする様々なデータ構造⁵²を与えることができる。

例. Set オブジェクトの生成

```
> s = new Set( [1,2,3,4,3,2,1] ) Enter    ←重複する要素は
Set(4) { 1, 2, 3, 4 }           ←排除される
```

このように Set オブジェクトは

`Set(要素数) { 要素の並び }`

の形式で表示される。ただし、このような形式のリテラルはない。

3.6.4.1 要素の存在確認

`has` メソッドで要素が Set オブジェクトに存在するかを確認できる。

書き方： `Set オブジェクト.has(要素)`

「Set オブジェクト」に「要素」が存在すれば `true` を、存在しなければ `false` を返す。

例. 要素の存在確認（先の例の続き）

```
> s.has( 3 ) Enter    ←要素 3 は
true        ←存在する
> s.has( 30 ) Enter   ←要素 30 は
false       ←存在しない
```

3.6.4.2 要素の追加と削除

`add` メソッドで要素を Set オブジェクトに追加することができる。

書き方： `Set オブジェクト.add(要素)`

「Set オブジェクト」に「要素」を追加し、追加後の Set オブジェクトを返す。

例. 要素の追加（先の例の続き）

```
> s.add( 5 ) Enter    ←新たな要素 5 の追加
Set(5) { 1, 2, 3, 4, 5 }      ←追加後の Set オブジェクト
```

`delete` メソッドで指定した要素を Set オブジェクトから削除することができる。

書き方： `Set オブジェクト.delete(要素)`

「Set オブジェクト」から「要素」を削除する。削除が成功した場合は `true` を、失敗した場合は `false` を返す。

例. 要素の削除（先の例の続き）

```
> s.delete( 5 ) Enter    ←既存の要素 5 の削除
true          ←削除成功
> s Enter        ←内容確認
Set(4) { 1, 2, 3, 4 }      ←要素 5 が削除されている
> s.delete( 5 ) Enter   ←存在しない要素 5 の削除を試みると…
false          ←削除失敗
```

3.6.4.3 要素の個数の調査

`size` プロパティから Set オブジェクトの要素数を参照することができる。

書き方： `Set オブジェクト.size`

例. 要素数の調査（先の例の続き）

```
> s.size Enter    ← s の要素数は
4          ← 4 個
```

⁵²イテラブルなオブジェクト（反復可能オブジェクト）

3.6.4.4 配列への変換

Array.from によって Set オブジェクトを配列に変換することができる。

書き方: `Array.from(Set オブジェクト)`

例. 配列への変換 (先の例の続き)

```
> Array.from( s )  ←配列への変換
[ 1, 2, 3, 4 ]      ←変換結果
```

3.6.4.5 全要素の削除

clear メソッドで Set オブジェクトを空にすることができる。

書き方: `Set オブジェクト.clear()`

戻り値は undefined である。

例. 全要素の削除 (先の例の続き)

```
> s.clear()  ←全要素の削除
undefined
> s  ←内容確認
Set(0) {}      ←空
```

3.6.4.6 ミュータブルなデータを要素にする際の注意

Set オブジェクトはミュータブルなデータを要素にすることができるが、これに関してはいくつか注意しなければならないことがある。以下にいくつか例を示す。

例. 配列を Set の要素にする

```
> a = [1,2,3]; b = [1,2,4]; s = new Set( [a,b] )  ← Set オブジェクトの作成
Set(2) { [ 1, 2, 3 ], [ 1, 2, 4 ] }      ←得られた Set オブジェクト
```

これは配列を要素に使用して Set オブジェクトを作成する例である。得られた Set オブジェクトに、指定した配列 (リテラル) が存在するか検査する例を示す。(次の例)

例. 指定した配列が要素として存在するか検査する (先の例の続き)

```
> s.has([1,2,3])  ←配列が要素にあるか調べてみると…
false           ←要素に含まれない
> s.has([1,2,4])  ←この配列も…
false           ←要素に含まれない
```

検査が正しくできていないように見えるが、次の例では正しい結果が得られる。

例. 正しい結果が得られるケース (先の例の続き)

```
> s.has(a)  ←変数 a の値が要素に含まれるかを検査
true      ←含まれる
> s.has(b)  ←変数 b の値が要素に含まれるかを検査
true      ←含まれる
```

このことから、Set オブジェクト作成時に与えた要素 (変数 a) と、検査時に与えた配列リテラル [1,2,3] は異なるものであることがわかる。従って、Set オブジェクトの要素を検査する際には、生成時と**同一のデータであるもの**を使用しなければならない。また、数値や文字列といった**プリミティブ型**の値の存在検査の場合はリテラル表記の値を与えても良い。

演習課題.

上の例の実行後に `b[2] = 3` として変数 b の配列を変更すると、b の内容は [1, 2, 3] となる。その状態で変数 s の値を参照すると、

```
Set(2) { [ 1, 2, 3 ], [ 1, 2, 3 ] }
```

となり、重複する要素を持つように見える。これは何を意味するのか考えよ。

またこの状態で、変数の値が Set オブジェクト s に含まれるか検査すると次のように正しい結果が得られる。

```
> s.has(a) 
true
> s.has(b) 
true
```

この理由についても考えよ.

3.6.5 スプレッド構文

配列 (Array) やオブジェクト (Object) を始めとする**反復可能なデータ構造**⁵³ は他のデータ構造のリテラルの中に展開することができる.

書き方: ... データ構造

ドット 3 つの後ろにデータ構造を記述することで、それを他のデータ構造のリテラルの中に展開することができる.

例. 配列を他の配列の中に展開する

```
> a1 = [4,5,6]     ←この配列 a1 を
[ 4, 5, 6 ]
> a2 = [1,2,3, ...a1, 7,8,9]     ←この配列の中に展開する
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]    ←展開結果の配列 a2
```

例. 文字列を配列の中に展開する

```
> s = "文字列"     ←この配列 s を
' 文字列 '
> a = [1,2,3, ...s, 4,5,6]     ←この配列の中に展開する
[ 1, 2, 3, ' 文', ' 字', ' 列', 4, 5, 6 ]    ←展開結果の配列 a
```

例. オブジェクトを他のオブジェクトの中に展開する

```
> ob1 = {c:3, d:4}     ←このオブジェクト ob1 を
{ c: 3, d: 4 }
> ob2 = {a:1, b:2, ...ob1, e:5, f:6}     ←このオブジェクトの中に展開する
{ a: 1, b: 2, c: 3, d: 4, e: 5, f: 6 }    ←展開結果のオブジェクト ob2
```

⁵³反復処理の構文の中で、反復の制御に使用できるデータ構造.

3.7 特殊なデータ構造

ここでは、ファイル入出力⁵⁴ や通信の際に使用される特殊なデータ構造について解説する。

3.7.1 型付き配列 (TypedArray) と ArrayBuffer

計算機の記憶資源はバイト (8 ビット) 単位のアクセスが基本であり、そのような形式でのデータの扱いに適した数値配列が JavaScript には用意されている。具体的には、8 ビット、16 ビット、32 ビット、64 ビットの各サイズにビット長が定められた数値の配列構造があり、それらを表 24 に示す。

表 24: 固定ビット長の数値の配列 (一部)

配列の種類	解説	1 つの要素が扱える値の範囲
Int8Array	符号付き 8 ビット整数の配列	-128 ~ 127
Uint8Array	符号なし 8 ビット整数の配列	0 ~ 255
Int16Array	符号付き 16 ビット整数の配列	-32,768 ~ 32,767
Uint16Array	符号なし 16 ビット整数の配列	0 ~ 65,535
Int32Array	符号付き 32 ビット整数の配列	-2,147,483,648 ~ 2,147,483,647
Uint32Array	符号なし 32 ビット整数の配列	0 ~ 4,294,967,295
BigInt64Array	符号付き 64 ビット長整数の配列	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
BigUint64Array	符号なし 64 ビット長整数の配列	0 ~ 18,446,744,073,709,551,615
Float32Array	32 ビット浮動小数点数の配列	正負ともに絶対値の範囲が約 1.2E-38 ~ 3.4E+38 IEEE 754 単精度浮動小数点数
Float64Array	64 ビット浮動小数点数の配列	正負ともに絶対値の範囲が約 5.0E-324 ~ 1.8E+308 IEEE 754 倍精度浮動小数点数

表 24 に挙げた配列は**型付き配列** (TypedArray) と呼ばれるもので、ArrayBuffer と呼ばれるデータ構造にアクセスするための**ビュー**である。すなわち、実際のデータは ArrayBuffer の形式で記憶上に保持されており、それに対して表 24 に挙げた配列の形式で要素にアクセスする仕組みとなっている。これによって**固定ビット長の数値の配列**が実現できる。

■ ArrayBuffer について

ArrayBuffer は、バイトデータの並びを表現する記憶域のオブジェクトである。また ArrayBuffer は**ミュータブル** (変更可能) なオブジェクトであり、各種のビューを介してバイナリデータの作成、編集を可能にする。

■ 型付き配列の作成

書き方 (1): `new 配列の種類 (要素の個数)`

書き方 (2): `new 配列の種類 (初期値のデータ)`

「配列の種類」は表 24 に示すものである。「要素の個数」を与えて作成した場合は、各要素の初期値はゼロである。「初期値のデータ」は配列の要素の初期値を決めるもので、通常の配列 (Array) などイテラブル (反復可能) なものを与えることができる。「初期値のデータ」に ArrayBuffer オブジェクトを与えると、それをそのまま参照するビューを作成するが、それ以外の型のデータを初期値として与えると、初期値のデータを格納する ArrayBuffer を新規に生成してそのビューを返す。

例. Int8Array の作成

```
> a1 = new Int8Array( 3 )   [Enter]   ←要素数 3 の配列を作成
Int8Array(3) [ 0, 0, 0 ]     ←得られた配列

> a2 = new Int8Array( [5,4,3,2,1] ) [Enter]   ← Array から Int8Array を作成
Int8Array(5) [ 5, 4, 3, 2, 1 ]   ←得られた配列
```

Int8Array の要素へのアクセスは、基本的には通常の配列 (Array) と同様である。

⁵⁴ 「4.11 ファイルの入出力」 (p.167) で解説する。

例. 要素へのアクセス（先の例の続き）

```
> a1[0] = 10; a1[1] = 9; a1[2] = 8; a1  ←要素に値を設定して配列を確認
Int8Array(3) [ 10, 9, 8 ] ←要素の値が変更されている
> a2[1] * 6  ←要素を参照して計算を実行
24 ←計算結果
```

符号なし整数の配列の要素に負の値を与えると **2 の補数** の解釈がなされる。

例. Uint8Array に負の値の要素を与える試み（先の例の続き）

```
> u1 = new Uint8Array( [-1] )  ←要素に -1 を与えると
Uint8Array(1) [ 255 ] ←それは 2 の補数では 255 となる
```

■ 型付き配列から基の ArrayBuffer を得る方法

型付き配列の buffer プロパティから ArrayBuffer を参照することができる。

例. Int8Array から ArrayBuffer を得る（先の例の続き）

```
> b2 = a2.buffer  ←ArrayBuffer を参照する
ArrayBuffer { [Uint8Contents]: <05 04 03 02 01>, byteLength: 5 } ←ArrayBuffer
```

■ 既存の配列のサイズを調べる方法

型付き配列や ArrayBuffer のバイトサイズは byteLength プロパティから参照できる。また型付き配列の要素数は length から参照できる。例えば次の例の様に 1 つの要素が 32 ビット（4 バイト）である整数配列を考える。

例. 32 ビット整数の配列（先の例の続き）

```
> a3 = new Int32Array([5,4,3,2,1])  ←32 ビット（4 バイト）整数の配列
Int32Array(5) [ 5, 4, 3, 2, 1 ]
```

当然であるが、この配列の要素数とバイトサイズは異なる。（次の例）

例. 配列のサイズに関する調査（先の例の続き）

```
> a3.length  ←要素数を調べる
5
> a3.byteLength  ←バイトサイズを調べる
20
> a3.buffer.byteLength  ←基の ArrayBuffer のバイトサイズを調べる
20
```

■ 型付き配列と通常の Array で共通する機能

通常の配列（Array）に対するメソッドの多くが、型付き配列にも同名のメソッドとして実装されているので試されたい。参考までに、Array と TypedArray で共通するメソッド名を表に挙げる。

表 25: 参考：Array と TypedArray で共通のメソッド名（一部）

copyWithin()	entries()	every()	fill()
filter()	find()	findIndex()	forEach()
includes()	indexOf()	join()	keys()
lastIndexOf()	map()	reduce()	reduceRight()
reverse()	slice()	some()	sort()
toLocaleString()	toString()	values()	

3.7.1.1 ArrayBuffer の複数のビューによる共有

型付き配列（TypedArray）は ArrayBuffer のビューであり、実際のデータは ArrayBuffer として保持されている。また、既存の ArrayBuffer から型付き配列のコンストラクタによって別のビューを作成することができる。これにより、同一の ArrayBuffer を共有する複数の型付き配列を作成することができる。これに関して例を挙げて説明する。

まず次のような処理で、32 ビット（4 バイト）長の符号付き整数を要素として 1 つ持った配列を作成する。

例. Int32Array の作成

```
> a32 = new Int32Array( [16909060] ) Enter    ←要素を 1 つ持った符号付き 4 バイト整数の配列  
Int32Array(1) [ 16909060 ]
```

この配列が要素として持つ 16909060 は、

$$16909060 = 1 \times 256^3 + 2 \times 256^2 + 3 \times 256^1 + 4 \times 256^0$$

であるので、1 バイトの位 (くらい)ごとに「1, 2, 3, 4」の値を持つ 4 つのバイト列で表現される。(256ⁿ の基数による表現)

ここで得られた配列 a32 の内容を持つ ArrayBuffer を buffer プロパティから参照して、Int8Array コンストラクタで 8 ビット (1 バイト) 符号付き整数配列のビューを取得してその内容を確認する。(次の例)

例. 上記配列の内容を共有する Int8Array の作成 (先の例の続き)

```
> a8 = new Int8Array( a32.buffer ) Enter    ←符号付き 1 バイト整数のビューを取得  
Int8Array(4) [ 4, 3, 2, 1 ]
```

a32 の値を 1 バイトごとに分解した形の要素を持っていることがわかるが、各バイト値の格納順位に注意すること。

上に示した処理は、Intel の Core i7 を CPU として持つ計算機環境で実行した例である。この場合、a32 の値は 256ⁿ の基数の表現では高い位から順に「1, 2, 3, 4」となるが、Int8Array では逆の順位で格納されていることがわかる。すなわち、配列の格納順位の低い方に 256ⁿ の基数表現における高い位の値が保持されることとなる。このことを指して「**バイトオーダーがリトルエンディアンである**」と表現する。これに関しては後の「3.7.1.2 バイトオーダー (リトルエンディアン/ビッグエンディアン)」(p.69) で解説する。

■ TypedArray のコンストラクタの振る舞いの違い

TypedArray のコンストラクタの引数に ArrayBuffer 以外の初期値 (通常の Array など) を与えると、そのデータを保持する ArrayBuffer を新規に生成し、それを buffer プロパティとして持つ TypedArray を作成して返す。しかし TypedArray のコンストラクタの引数に ArrayBuffer を初期値として与えると、その ArrayBuffer を buffer プロパティとして持つ TypedArray を作成して返す。すなわち、先に示した例における a32 と a8 は共通の ArrayBuffer を持つ。これに関して例を挙げて示す。

先の例における a8 の各要素の値を次のようにして変更する。

例. 先の例の a8 の変更 (先の例の続き)

```
> a8[0] = 1; a8[1] = 2; a8[2] = 3; a8[3] = 4; a8 Enter    ←要素の値を変更して a8 全体を表示  
Int8Array(4) [ 1, 2, 3, 4 ]    ←値が変更されている
```

この処理によって、同じ ArrayBuffer を共有する a32 の値も変更される。(次の例)

例. 先の例の a32 の値の確認 (先の例の続き)

```
> a32 Enter    ←値の確認  
Int32Array(1) [ 67305985 ]    ←変更されている
```

この処理の結果として得られている a32 の要素の値 67305985 は

$$67305985 = 4 \times 256^3 + 3 \times 256^2 + 2 \times 256^1 + 1 \times 256^0$$

であり、リトルエンディアンのアーキテクチャの格納形式での表現に合致することがわかる。

3.7.1.2 バイトオーダー (リトルエンディアン/ビッグエンディアン)

2 バイト以上の長さで表現される数値データに関しては、記憶資源上でのバイト毎の格納順番に注意しなければならない。計算機が値をバイト列として記憶資源に格納する際の順序を**バイトオーダー**⁵⁵ (図 29) という。バイトオーダーは計算機のアーキテクチャによって異なり、Intel の x86 系とその互換 CPU では**リトルエンディアン**である。

このような事情から、バイナリデータを異なるアーキテクチャの計算機環境の間で流用する際にはバイトオーダーに関して注意しなければならない。

⁵⁵リトルエンディアン、ビッグエンディアンの他にもミドルエンディアンなるバイトオーダーを採用している CPU も存在する。

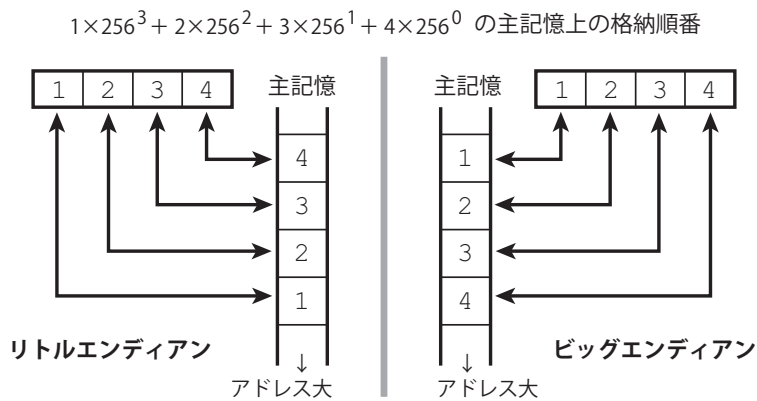


図 29: バイトオーダー

参考)

モトローラ社の MC68000 系列の CPU ではビッグエンディアンである。これとは別に JVM の場合は独自のアーキテクチャを実現しており、実行環境の CPU の種類に依らず常にビッグエンディアンである。

▲注意▲

ARM, PowerPC の系列の CPU ではリトルエンディアン / ビッグエンディアンを切り替えることができる⁵⁶ ので、これら CPU の計算機環境との間でバイナリデータを交換する場合は特にバイトオーダーを意識すること。

3.7.2 データビュー (DataView)

型付き配列 (TypedArray) とは別に ArrayBuffer のビューとなる DataView があり、TypedArray よりも柔軟に ArrayBuffer の内容を編集することができる。DataView を作成するには同名のコンストラクタを使用する。

書き方: `new DataView(ArrayBuffer オブジェクト)`

「ArrayBuffer オブジェクト」のビューとなる DataView オブジェクトを生成して返す。

表 26 に、値を参照するための DataView のメソッドを挙げる。

表 26: ArrayBuffer のバイト位置 Offset の内容を参照するメソッド (一部)

メソッド	解説
<code>getInt8(Offset)</code>	Offset のバイト位置のバイト値を符号付き整数として返す。
<code>getUint8(Offset)</code>	Offset のバイト位置のバイト値を符号無し整数として返す。
<code>getInt16(Offset, BiteOrder)</code>	Offset のバイト位置から 2 バイト (16 ビット) 分を符号付き整数として返す。
<code>getUint16(Offset, BiteOrder)</code>	Offset のバイト位置から 2 バイト (16 ビット) 分を符号無し整数として返す。
<code>getInt32(Offset, BiteOrder)</code>	Offset のバイト位置から 4 バイト (32 ビット) 分を符号付き整数として返す。
<code>getUint32(Offset, BiteOrder)</code>	Offset のバイト位置から 4 バイト (32 ビット) 分を符号無し整数として返す。
<code>getBigInt64(Offset, BiteOrder)</code>	Offset のバイト位置から 8 バイト (64 ビット) 分を符号付き整数として返す。
<code>getBigUint64(Offset, BiteOrder)</code>	Offset のバイト位置から 8 バイト (64 ビット) 分を符号無し整数として返す。
<code>getFloat32(Offset, BiteOrder)</code>	Offset のバイト位置から単精度浮動小数点数 (32 ビット) を読取って返す。
<code>getFloat64(Offset, BiteOrder)</code>	Offset のバイト位置から倍精度浮動小数点数 (64 ビット) を読取って返す。

BiteOrder に true を与えるとリトルエンディアン, false を与えるとビッグエンディアンとなる。BiteOrder を省略すると false と解釈されるが、処理系によって仕様が異なることがある。

表 27 に、値を設定するための DataView のメソッドを挙げる。
表 27 に挙げたメソッドは、処理後に undefined を返す。

■ DataView の使用例

ArrayBuffer を生成した後、それに対するビュー DataView, Int32Array を作成して値の設定と参照を行う例を示す。ArrayBuffer を単独で作成するにはコンストラクタを実行する。

⁵⁶ バイエンディアン

表 27: ArrayBuffer のバイト位置 Offset に値 Value を設定するメソッド (一部)

メソッド	解 説
setInt8(Offset, Value)	8 ビット符号付き整数を設定する.
setUint8(Offset, Value)	8 ビット符号無し整数を設定する.
setInt16(Offset, Value, BiteOrder)	16 ビット符号付き整数を設定する.
setUint16(Offset, Value, BiteOrder)	16 ビット符号無し整数を設定する.
setInt32(Offset, Value, BiteOrder)	32 ビット符号付き整数を設定する.
setUint32(Offset, Value, BiteOrder)	32 ビット符号無し整数を設定する.
setBigInt64(Offset, Value, BiteOrder)	64 ビット符号付き整数を設定する.
setBigUint64(Offset, Value, BiteOrder)	64 ビット符号無し整数を設定する.
setFloat32(Offset, Value, BiteOrder)	単精度浮動小数点数 (32 ビット) を設定する.
setFloat64(Offset, Value, BiteOrder)	倍精度浮動小数点数 (64 ビット) を設定する.

BiteOrder に true を与えるとリトルエンディアン, false を与えるとビッグエンディアンとなる.
BiteOrder を省略すると false と解釈されるが, 処理系によって仕様が異なることがある.

書き方: new ArrayBuffer(バイト長)

「バイト長」の長さの ArrayBuffer を作成して返す.

下の例は, ArrayBuffer を作成して, そのビューを 2 種類作成する処理である.

例. ArrayBuffer とそれに対するビューの作成

```
> ab = new ArrayBuffer( 4 )   [Enter]   ←長さ 4 バイトの ArrayBuffer を作成
ArrayBuffer { [Uint8Contents]: <00 00 00 00>, byteLength: 4 }
> dv = new DataView( ab )     [Enter]   ←上記 ArrayBuffer のデータビューを作成
DataView { ... }
> a32 = new Int32Array( ab )   [Enter]   ←上記 ArrayBuffer の型付き配列のビューを作成
Int32Array(1) [ 0 ]
```

この様にして得られた ArrayBuffer にデータビューを通して「4,3,2,1」をこの順で設定して Int32Array のビューで内容を確認する処理を次に示す.

例. データビューで値を設定して Int32Array のビューで内容を確認 (先の例の続き)

```
> dv.setInt8(0,4); dv.setInt8(1,3); dv.setInt8(2,2); dv.setInt8(3,1); a32[Enter] ←設定と確認
Int32Array(1) [ 16909060 ]   ←符号付き 4 バイト整数としての表現が得られている
```

この処理は Intel Corei7 CPU (リトルエンディアン) での実行例であり, その解釈での整数が得られていることがわかる. 次に, バイトオーダーを明に指定して値を取り出す例を示す.

例. バイトオーダーを指定して符号付き 4 バイト整数を取り出す (先の例の続き)

```
> dv.getInt32(0,true) [Enter]   ←符号付き 4 バイト整数 (リトルエンディアン) を取り出す
16909060                     ←得られた値 (リトルエンディアン)
> dv.getInt32(0,false) [Enter]  ←符号付き 4 バイト整数 (ビッグエンディアン) を取り出す
67305985                     ←得られた値 (ビッグエンディアン)
```

■ DataView の用途

DataView は「指定したバイト位置に, 指定した型, 指定したバイトオーダーで値を読み書きする」処理を実現するものである. この機能は, バイナリデータの編集を実現するものであり, ファイル入出力や通信におけるデータの取扱いのための基本的な手段となる.

各種の数値型のデータをバイナリデータに変換する場合にも DataView が応用できる.

3.7.3 文字列⇄バイナリデータの変換

TextEncoder クラスを用いるとマルチバイト文字を含んだ文字列を Uint8Array の配列に変換することができる。

書き方： `new TextEncoder()`

utf-8 エンコーディングの文字列を扱うための変換器（エンコーダ）を生成して返す。これによって得られたエンコーダに `encode` メソッドを用いることで変換処理が実行される。

書き方： `エンコーダ.encode(変換対象文字列)`

「変換対象文字列」（utf-8）を Uint8Array 形式配列に変換したものを返す。

例. 文字列を Uint8Array に変換する

```
> E = new TextEncoder() Enter    ←エンコーダの生成
TextEncoder {encoding: 'utf-8'}    ←得られたエンコーダ
> u1 = E.encode( "科学" ) Enter    ←"科学" の文字を Uint8Array 配列に変換
Uint8Array(6) [ ... ]    ←得られた配列
> u2 = E.encode( "技術" ) Enter    ←"技術" の文字を Uint8Array 配列に変換
Uint8Array(6) [ ... ]    ←得られた配列
```

TextDecoder クラスを用いると Uint8Array 形式の配列を文字列に変換することができる。

書き方： `new TextDecoder(エンコーディング)`

Uint8Array 形式の配列を扱うための変換器（デコーダ）を生成して返す。変換処理は「エンコーディング」に従う。「エンコーディング」を省略すると utf-8 が採用される。これによって得られたデコーダに `decode` メソッドを用いることで変換処理が実行される。

書き方： `デコーダ.decode(Uint8Array 配列)`

「Uint8Array 配列」を文字列に変換したものを返す。

例. Uint8Array を文字列に変換する（先の例の続き）

```
> D = new TextDecoder() Enter    ←デコーダの生成
TextDecoder {encoding: 'utf-8', fatal: false, ignoreBOM: false} ←得られたデコーダ
> D.decode( u1 ) Enter    ←先の例の u1 を文字列に変換
'科学'    ←得られた文字列
> D.decode( u2 ) Enter    ←先の例の u2 を文字列に変換
'技術'    ←得られた文字列
```

3.7.3.1 Node.js での方法

Node.js にも同様の API が `util` ライブラリに提供されており、これを読み込むことで `TextEncoder`、`TextDecoder` が使用できる。具体的には、Node.js で実行するプログラムの冒頭で、

```
const util = require('util');
const TextEncoder = util.TextEncoder;
const TextDecoder = util.TextDecoder;
```

と記述することで、Web ブラウザで実行する場合と同様の方法で変換処理が実行できる。

3.7.4 Blob

先に解説した `ArrayBuffer` はバイナリデータを編集するための **ミュータブル** (変更可能) なオブジェクトであるが、**イミュータブル** (変更不可能) なものとして、ここで解説する `Blob` (Binary Large Object) がある。Blob は元来、データベース管理システム (DBMS) においてバイナリデータを格納する場合のデータ型として考案されたものであり、JavaScript においても同様のものとして `Blob` を扱うための型 (クラス) が実装された。

`Blob` オブジェクトは、ファイル入出力や通信においてバイナリデータを効率的に扱うのに適しており、Web ブラウザ側でのプログラミングにおいてよく用いられる。すなわち、先に解説した `ArrayBuffer` とそれに付随するオブジェクトの主な目的はバイナリデータの作成と編集であるが、出来上がったバイナリデータを通信路に送出する、あるいはファイルに出力する際には `Blob` オブジェクトに変換するという流れを取るのが一般的である。Node.js でもバージョン 18 以降で `Blob` クラスが標準実装された。

後の「4.11 ファイルの入出力」(p.167) ではファイル入出力のための `File` クラスについて解説するが、`File` クラスは `Blob` クラスの拡張であり、ファイルと `Blob` は同様の扱いができることが多い。

`Blob` オブジェクトは `Blob` コンストラクタで生成することができる。

書き方： `new Blob(データ, オプション)`

「データ」を「オプション」で指定した形式で `Blob` オブジェクトにして返す。「データ」には `Array` を始めとするシーケンスを与える。「オプション」には `Object` クラスのオブジェクトを与える。「オプション」に指定するプロパティとして `type` が特に重要であり、このプロパティに **MIME タイプ** (メディアタイプ) を与えることで、作成する `Blob` オブジェクトの形式が決定される。

`Blob` オブジェクトが持つデータのサイズ (バイト単位) は `size` プロパティが、MIME タイプは `type` プロパティが保持している。

3.7.4.1 Blob 作成の例

■ テキスト (文字列データ) を Blob にする例

文字列データを `Blob` オブジェクトにする例を次に示す。

例. テキストデータを `Blob` に変換する

```
> s = "一行目¥n 二行目¥n 三行目" Enter    ← 3 行分の文字列データ
'一行目¥n 二行目¥n 三行目'

> b = new Blob( [s], {type:"text/plain"} ) Enter    ← 上の文字列 s を Blob に変換
Blob { size: 29, type: 'text/plain' }      ← 得られた Blob オブジェクト
```

これは変数 `s` が持つ 3 行分のテキストを単純なテキスト (`text/plain`) として `Blob` に変換して変数 `b` に与える例である。`Blob` のデータサイズと MIME タイプを参照する例を次に示す。

例. データサイズと MIME タイプの参照 (先の例の続き)

```
> b.size Enter    ← データサイズの参照
29

> b.type Enter    ← MIME タイプの参照
'text/plain'
```

■ 型付き配列 (TypedArray) を Blob にする例

`TypedArray` を `Blob` オブジェクトにする例を次に示す。

例. `Int8Array` を `Blob` に変換する (先の例の続き)

```
> a8 = new Int8Array([1,2,3,4,5]) Enter    ← 型付き配列の作成
Int8Array(5) [ 1, 2, 3, 4, 5 ]

> b8 = new Blob( [a8], {type:"application/octet-stream"} ) Enter    ← 上の配列を Blob に変換
Blob { size: 5, type: 'application/octet-stream' }          ← 得られた Blob オブジェクト
```

データフォーマットや使用するアプリケーションを意識しない形でバイナリデータを作成する場合、MIME タイプは `"application/octet-stream"` とする。

3.7.4.2 Blob のデータの取り出し

Blob オブジェクトの内容を ArrayBuffer に変換して取得するには `arrayBuffer` メソッドを用いる。また、テキストを保持する Blob オブジェクトからテキストデータを取得するには `text` メソッドを使用する。これらメソッドは非同期処理であり、変換処理の **プロミス** (Promise) オブジェクトを返す。

書き方 (ArrayBuffer): `Blob オブジェクト.arrayBuffer()`

書き方 (テキスト) : `Blob オブジェクト.text()`

これらメソッドは非同期処理なので、Promise に対する適切なハンドリングを行うこと。次に示す例は `await` によってメソッド実行の完了を同期するものである。

例. Blob の内容を取り出す処理 (先の例の続き)

```
> ab = await b.arrayBuffer() Enter    ← ArrayBuffer への変換
ArrayBuffer { ... byteLength: 29}
> s2 = await b.text() Enter    ←テキストデータ (文字列) への変換
'一行目¥n 二行目¥n 三行目'
```

これは非同期処理の終了を同期するものであり、処理に時間がかかる場合においては Promise の非同期処理として変換を行うべきである。

上の例で得られた `ab`, `s2` は Blob オブジェクト `b` の内容を参照するものではなく、新たに生成されたものである。

3.7.5 BinaryString

バイナリデータ (バイト列) を手軽にリテラル表現する方法に `BinaryString` がある。これは `"¥x"` による **エスケープシーケンス** である。

例. 3 バイトのバイナリデータ (`BinaryString`) を 16 進数の 00, 01, 02 として作成

```
> bs = "¥x00¥x01¥x02" Enter    ← BinaryString の作成
'¥x00¥x01¥x02'
```

バイナリデータの列が文字列 (`BinaryString`) `bs` として作成されている。

表現する値が **アスキーコード** (ASCII) の場合は、通常の文字列としてデータが得られる。

例. エスケープシーケンスで文字列 "ABC" を作成

```
> bs = "¥x41¥x42¥x43" Enter    ← BinaryString の作成
'ABC'    ←通常の文字列が得られる
```

以上のことからわかるように、`BinaryString` はそのデータ型が存在するのではなく、エスケープシーケンスで表現された 文字列 である。

3.7.5.1 Blob との間での変換

`BinaryString` を Blob に変換する例を示す。

例. `BinaryString` を Blob に変換する

```
> bs = "¥x03¥x04¥x05" Enter    ← BinaryString の作成
'¥x03¥x04¥x05'
> b = new Blob( [bs], {type:"application/octet-stream"}) Enter    ← Blob に変換
Blob {size: 3, type: 'application/octet-stream'}    ←得られた Blob
```

この例のように MIME タイプ (メディアタイプ) として `"application/octet-stream"` を指定する。

次に、`FileReader` を利用して Blob を `BinaryString` に変換する例を示す。

例. 変換処理の準備 (先の例の続き)

```
> fr = new FileReader(); fr.onload = function() { window.bs2 = fr.result; } Enter
f () { window.bs2 = fr.result; }    ←戻り値の関数式
```

これで変換処理用の `FileReader` が `fr` に作成された⁵⁷。これを用いて Blob を `BinaryString` に変換する。

⁵⁷詳しくは後の「4.11.1.2 `FileReader` によるファイルの読み込み」(p.168) で解説する。

例. Blob を BinaryString に変換する (先の例の続き)

```
> fr.readAsBinaryString( b )  ←変換処理の実行
undefined
> bs2  ←得られた BinaryString の確認
'¥x03¥x04¥x05'
```

BinaryString が得られていることがわかる。

3.8 条件分岐

3.8.1 if … else 文

条件式や論理値に従って実行するプログラムの部分を選択するための if … else 文がある。

書き方： if (条件) 実行部 1 else 実行部 2

「条件」の部分には論理値を与えるもの (式, 変数など) を記述する。「条件」の値が true ならば「実行部 1」を, false ならば「実行部 2」を実行する。実行部には 1 つの文あるいはブロックを与える。また実行部には別の if … else 文を与えることができる。else 以降は省略することができる。

例. 単純な if 文

```
> p = true  ←論理値 (真)
true
> if ( p ) console.log('真です. ');  ← if 文
真です.          ← console.log による出力
undefined        ← if 文自体の実行結果
```

この形式の記述では p が偽 (false) の場合は何も実行されない。(次の例)

例. 偽の場合の動作

```
> p = false  ←論理値 (偽)
false
> if ( p ) console.log('真です. ');  ← if 文
undefined          ← if 文自体の実行結果
```

else 以降を記述すると p が偽 (false) の場合の処理ができる。(次の例)

例. else の記述

```
> p = false  ←論理値 (偽)
false
> if ( p ) console.log('真です. '); else console.log('偽です. '); 
偽です.          ← console.log による出力
undefined        ← if 文自体の実行結果
```

実際のプログラミングにおいては、実行部は「短文;」の形式よりもブロック「{ プログラム }」の形式で与えることが多い。(次の例)

例. 実行部はブロックで記述するのが一般的

```
> p = true  ←論理値 (真)
true
> if ( p ) {  ←真の場合のブロックの開始
    console.log('真です. ');  ←実行部 1
} else {  ←真の場合のブロックの終了と偽の場合のブロックの開始
    console.log('偽です. ');  ←実行部 2
}  ←偽の場合のブロックの終了
真です.          ←実行部 1 による出力
undefined        ← if 文自体の実行結果
```

else 以降に更に if 文を記述することで、条件を段階的に判定する処理が実現できる。(次の例)

例. 段階的な条件判定

```
> x = 3 [Enter]    ←この値を以下で判定処理する
3
> if ( x < 0 ) { [Enter]
    console.log("負の値です. "); [Enter]    ←実行部 1
} else if ( x > 5 ) { [Enter]
    console.log("5 より大きいです. "); [Enter]    ←実行部 2
} else { [Enter]
    console.log("0 以上かつ 5 以下です. "); [Enter]    ←実行部 3
} [Enter]
0 以上かつ 5 以下です.    ←実行部 3 による出力
undefined    ← if 文自体の実行結果
```

x の値を様々に変更して実行を試みられたい。

3.8.2 switch 文

与えた式の値によって実行する部分を選択するための switch 文がある。

書き方：

```
switch ( 式 ) {
    case 値 1 :
        (式が値 1 と同じ場合の実行部)
        break;
    case 値 2 :
        (式が値 2 と同じ場合の実行部)
        break;
    :
    default :
        (どれにも該当しなかった場合の実行部)
}
```

「式」の値が「値 1」,「値 2」, … のどれに合致するかによって実行部を選択する。各 case 節の最後には break を記述する。break を記述しない場合は、プログラムの実行が下の case 節に継続（落下：fall-through）する。

default 節は省略することができる。

例. switch 文による実行部の選択

```
> x = 2 [Enter]    ←値の設定
2
> switch ( x ) { [Enter]    ← switch 文の開始
    case 1: [Enter]    ← x が 1 の場合
        console.log("1 です. "); [Enter]    ←実行部 (1)
        break;
    case 2: [Enter]    ← x が 2 の場合
        console.log("2 です. "); [Enter]    ←実行部 (2)
        break;
    case 3: [Enter]    ← x が 3 の場合
        console.log("3 です. "); [Enter]    ←実行部 (3)
        break;
    default: [Enter]    ←どれにも合致しない場合
        console.log("どれもありません. "); [Enter]    ←実行部 (4)
} [Enter]    ← switch 文の終了
2 です.    ←実行部 (2) による出力
undefined    ← switch 文の実行結果
```

変数 x の値によって実行部が選択される様子がわかる。次に、break の記述がないことによる「落下」(fall-through) の例を示す。

例. 落下の例

```
> x = 2  ←値の設定
2
> switch ( x ) {  ← switch 文の開始
  case 1:  ← x が 1 の場合
    console.log("1 です. ");  ←実行部 (1)
  case 2:  ← x が 2 の場合
    console.log("2 です. ");  ←実行部 (2)
  case 3:  ← x が 3 の場合
    console.log("3 です. ");  ←実行部 (3)
  default:  ←どれも合致しない場合
    console.log("どれもありません. ");  ←実行部 (4)
}  ← switch 文の終了
2 です.                ←実行部 (2) による出力
3 です.                ←実行部 (3) による出力 (落下)
どれもありません.     ←実行部 (4) による出力 (落下)
undefined              ← switch 文の実行結果
```

「case 2:」の部分に合致した後、switch 文の最後までが実行されている。

3.8.3 値を選択する 3 項演算子

条件式や論理値に従って値を選択するための式がある。

書き方： 条件 ? 値 1 : 値 2

「条件」の部分には論理値を与えるもの（式、変数など）を記述する。「条件」の値が true ならば「値 1」を、false ならば「値 2」を返す。

例. 奇数/偶数の判定

```
> x = 1  ←奇数の設定
1
> a = x%2==0 ? "even" : "odd"  ←判定
'odd'                                     ←変数 a には 'odd' が得られる
> x = 2  ←偶数の設定
2
> a = x%2==0 ? "even" : "odd"  ←判定
'even'                                    ←変数 a には 'even' が得られる
```

変数 x の値を 2 で割った余りが 0 かどうかで変数 a に得られる値が変わっている。

3.8.4 論理値以外を条件式に用いるケース

if 文の条件式には論理値（true か false）以外のものを用いることができる。（次の例）

例. 整数を条件式に用いる（その 1）

```
> c = 1  ←値を 1 に設定
1
> if ( c ) { console.log("真です"); } else { console.log("偽です"); }  ←条件判定
真です                ← true の判定
undefined              ← if 文の実行結果
```

if 文の条件式として 1 を与えると「真」と判定される。

例. 整数を条件式に用いる（その 2）

```
> c = 0;  ←値を 0 に設定
0
> if ( c ) { console.log("真です"); } else { console.log("偽です"); }  ←条件判定
偽です                ← false の判定
undefined
```

if 文の条件式として 0 を与えると「偽」と判定される。すなわち、JavaScript では 0 は false 相当、それ以外の数値は true 相当の判定となる。

配列やオブジェクトは、要素が空のものであっても真の判定⁵⁸ となる。(次の例)

例. 空配列を条件式に用いる

```
> c = [] Enter    ←値を空配列 [] に設定
[]
> if ( c ) { console.log("真です"); } else { console.log("偽です"); } Enter ←条件判定
真です      ← true の判定
undefined
```

論理値の true のように真となることを **truthy**、論理値の false のように偽となることを **falsy** と表現する。falsy なものを表 28 に挙げる。

表 28: 条件判定における false 相当 (falsy) のもの		
値	型	解説
null	null	値が存在しないことを示す。
undefined	undefined	プリミティブ値の undefined
NaN	数値	数値ではないことを示す。
0	数値	数値 0.0 や 0x0 など含む。
-0	数値	マイナスゼロ -0.0 や -0x0 など含む。
0n	長整数	長整数型のゼロ 0x0n も含む。なお、長整数型にはマイナスゼロはない。0n の負の数は 0n。
""	文字列	空文字列 "" や “ ” も含む。

⁵⁸他の言語 (Python など) とは異なる特徴であるので注意すること。

3.9 反復制御

3.9.1 while 文

書き方： **while (条件) 実行部**

「条件」（論理値）が true である場合に「実行部」を実行し、これを繰り返す。「実行部」には1つの文あるいはブロックを与える。

例. while による反復（その1）

```
> x = 3  ←変数 x の値の設定
3
> while ( x > 0 ) console.log( x-- );  ← 1 つの文を繰り返す記述
3          ←反復処理 1 回目
2          ←反復処理 2 回目
1          ←反復処理 3 回目
undefined ← while 文の実行結果
```

これは while 文で1つの文の実行を繰り返す例である。

例. while による反復（その2）

```
> x = 3  ←変数 x の値の設定
3
> while ( x > 0 ) {  ← while 文の開始
  console.log( x ); 
  x--; 
}  ← while 文の記述の終了
3          ←反復処理 1 回目
2          ←反復処理 2 回目
1          ←反復処理 3 回目
1          ← while 文の実行結果
```

これはブロックの実行を反復する例である。

演習課題.

上記の実行例で while 文の実行結果が 1 となっている理由について考えよ。

3.9.2 do … while 文

書き方： **do 実行部 while (条件);**

まず「実行部」を実行し、「条件」（論理値）が true である場合にこの処理を繰り返す。「条件」が false ならば do … while 文を終了する。

先の while 文と違い、do … while 文では「実行部」は「条件」の記述に無関係に必ず1回は実行される。

例. do … while による反復（その1）

```
> x = 3  ←変数 x の値の設定
3
> do console.log(x--); while ( x > 0 );  ← 1 つの文を繰り返す記述
3          ←反復処理 1 回目
2          ←反復処理 2 回目
1          ←反復処理 3 回目
undefined ← do … while 文の実行結果
```

これは do … while 文で1つの文の実行を繰り返す例である。

例. do … while による反復（その2）

```
> x = 3  ←変数 x の値の設定
3
> do {  ← do … while 文の開始
  console.log(x); 
  x--; 
} while ( x > 0 );  ← do … while 文の終了
3      ←反復処理 1 回目
2      ←反復処理 2 回目
1      ←反復処理 3 回目
1      ← do … while 文の実行結果
```

これはブロックの実行を反復する例である。

条件とは無関係に実行部は必ず 1 回は実行される。このことを次の例で示す。

例. 必ず 1 回は実行される実行部

```
> x = -1  ←変数 x の値の設定
-1      ←負の値
> do console.log(x); while( x > 0 );  ← do … while 文では
-1      ←必ず 1 回は実行される
undefined ← do … while 文の実行結果
```

3.9.3 for 文

書き方： for (初期化処理; 条件; 各回の後処理) 実行部

まず「初期化処理」を 1 度だけ実行する。「条件」が true であれば「実行部」を実行した後「各回の後処理」を行い、これを繰り返す。「実行部」には 1 つの文あるいはブロックを与える。

例. 1 つの文を繰り返す処理

```
> for ( let n=0; n<3; n++ ) console.log(n);  ← 1 つの文の反復
0      ←反復処理 1 回目
1      ←反復処理 2 回目
2      ←反復処理 3 回目
undefined ← for 文の実行結果
```

この例における変数 n のスコープは関数内部のブロックに限定されており、事後に参照できない。(次の例)

例. 変数 n の確認（先の例の続き）

```
> n  ←変数 n の確認
Uncaught ReferenceError: n is not defined ←エラーとなる
```

例. ブロックを繰り返す処理

```
> for ( let n=0; n<3; n++ ) {  ← for 文の開始
  console.log("n="+n); 
}  ← for 文の記述の終了
n=0      ←反復処理 1 回目
n=1      ←反復処理 2 回目
n=2      ←反復処理 3 回目
undefined ← for 文の実行結果
```

この例においても変数 n は関数の実行部のブロックのスコープであり、終了後は参照できない。

3.9.3.1 for … of 文

for … of 文を用いると、反復可能（イテラブル）なデータ構造から要素を 1 つずつ取り出す形で反復処理が実行できる。反復可能なものには文字列、配列、Map オブジェクト、Set オブジェクトや各種のイテレータなどがある。ただし、オブジェクト（Object）はこの構文では使用できないので、後に解説する for … in 文で使用する。

書き方： for (要素 of イテラブル) 実行部

「イテラブル」から要素を1つずつ「要素」に受け取りながら「実行部」を実行する。

例. for ... of 文 (文字列)

```
> s = "ABC" [Enter] ←文字列
'ABC'

> for ( e of s ) console.log(e); [Enter] ← 1文字ずつ出力
A
B
C
undefined ← for ... of 文の実行結果
```

例. for ... of 文 (Map オブジェクト)

```
> m = new Map( [ ["apple","りんご"], ["orange","みかん"], ["grape","ぶどう"] ] ) [Enter] ← Map
Map(3) { 'apple' => 'りんご', 'orange' => 'みかん', 'grape' => 'ぶどう' }

> for ( e of m ) { [Enter] ← for ... of の開始
  console.log("key: "+e[0]); [Enter] ← キーの出力
  console.log("value: "+e[1]); [Enter] ← 値の出力
} [Enter] ← for ... of の記述の終了
key: apple
value: りんご
key: orange
value: みかん
key: grape
value: ぶどう
undefined ← for ... of 文の実行結果
```

3.9.3.2 for ... in 文

for ... in 文を用いると、オブジェクト (Object) を始めとする**列挙可能** (enumerable) なデータ構造からキーを1つずつ取り出す形で反復処理が実行できる。

書き方: for (キー in 列挙可能なデータ) 実行部

「列挙可能なデータ」からキーを1つずつ「キー」に受け取りながら「実行部」を実行する。

例. for ... in 文 (オブジェクト)

```
> obj = {"apple":"りんご", "orange":"みかん", "grape":"ぶどう"} [Enter] ←オブジェクト
{ apple: 'りんご', orange: 'みかん', grape: 'ぶどう' }

> for ( k in obj ) { [Enter] ← for ... in の開始
  let v = obj[k]; [Enter]
  console.log("key:"+k+", ¥tvalue:"+v); [Enter]
} [Enter] ← for ... in の記述の終了
key:apple, value:りんご
key:orange, value:みかん
key:grape, value:ぶどう
undefined ← for ... in 文の実行結果
```

注意) シンボル (Symbol) のキーは無視される。

例. シンボルのキーを含む場合 (先の例の続き)

```
> obj[Symbol("sm")] = "シンボル"; obj [Enter] ←プロパティの追加
{ apple: 'りんご', orange: 'みかん', grape: 'ぶどう', [Symbol(sm)]: 'シンボル' }

> for ( k in obj ) { [Enter] ← for ... in の開始
  let v = obj[k]; [Enter]
  console.log("key:"+k+", ¥tvalue:"+v); [Enter]
} [Enter] ← for ... in の記述の終了
key:apple, value:りんご
key:orange, value:みかん
key:grape, value:ぶどう
undefined ← for ... in 文の実行結果
```

シンボルのキーは無視されていることがわかる。

3.9.4 break と continue

反復制御の処理の途中で break 文があると、その時点で反復の処理を終え、当該反復構文の外へ出る。

例. 反復の中断

```
> for ( let n=0; n < 10; n++ ) { Enter    ← for 文の開始
    console.log(n); Enter
    if ( n==2 ) break; Enter    ← n の値が 2 に達したら強制終了
} Enter    ← for 文の記述の終了
0
1
2
undefined    ← for 文の実行結果
```

この例の for 文は、n の値が 0 から 9 まで反復する記述であるが、n が 2 に達した時点で break が実行されて、for 文の外へ出る。

反復制御の処理の途中で continue 文があると、その時点で反復の処理を次の回にスキップする。すなわち、実行中のブロックの処理を打ち切り、for や while の次の条件判定処理に移行する。

例. 反復を次の回にスキップ

```
> for ( let n=1; n < 10; n++ ) { Enter    ← for 文の開始
    if ( n%3==0 ) continue; Enter    ← n が 3 の倍数なら次回にスキップ
    console.log(n); Enter
} Enter    ← for 文の記述の終了
1
2
4                ← 3 の倍数がスキップされている
5
7
8
undefined    ← for 文の実行結果
```

3.10 関数の定義

関数は、受け取った引数を用いて処理を行った結果、何らかの値（戻り値）を呼び出し元に返すものである⁵⁹。関数は function 宣言文で定義する。

書き方： **function 関数名 (引数並び) { 実行部 }**

「関数名」の関数を定義する。「引数並び」に受け取った値を用いて「実行部」を実行する。「引数並び」は記号をコンマで区切って並べたものであり、省略することも可能である。「実行部」は具体的な処理を記述したブロックである。ブロック内に return 文があると処理を終了して値を返す。

書き方： **return 戻り値**

「戻り値」を関数の値として呼び出し元に返す。戻り値や return 文自体を省略するとその関数の戻り値は undefined となる。

function 宣言文自体は実行結果として undefined を返す。定義した関数に引数を渡して呼び出すには

関数名 (引数並び)

とする。

例. 加算を実行する関数 kasan

```
> function kasan(x,y) { return x+y; } Enter    ←関数 kasan の定義
undefined                               ←宣言文の戻り値
> kasan(1,2) Enter    ←上で定義した kasan に引数を与えて呼び出す
3                                         ← kasan(1,2) の戻り値
```

戻り値や return 文を省略する例を次に示す。

例. 戻り値が undefined となる関数 (1)

```
> function kasan2(x,y) { x+y; } Enter    ←関数 kasan2 の定義 (return 文がない)
undefined                       ←上の文の戻り値
> kasan2(1,2) Enter    ←呼び出す
undefined                       ←関数の戻り値
```

例. 戻り値が undefined となる関数 (2)

```
> function kasan3(x,y) { Enter    ←関数 kasan3 の定義の記述の開始
  let z = x+y; Enter
  return; Enter    ← return の後の戻り値がない
} Enter    ←定義の記述の終了
undefined
> kasan3(1,2) Enter    ←呼び出す
undefined     ←関数の戻り値
```

関数定義において、意図して戻り値や return 文を省略することはあるが、注意が必要である。

3.10.1 変数のスコープ

関数定義の記述内で var, let, const で宣言された変数のスコープは当該関数内である。すなわち、そのような変数は当該関数内でのみ使用可能である。また、関数定義の記述の外部で宣言された変数と同じ名前の変数の使用を関数内部で宣言すると、その変数のスコープは当該関数の内部となる。

⁵⁹それ以外に、オブジェクトのコンストラクタとしての用途がある。詳しくは後の「3.12 オブジェクト指向プログラミング」(p.96)で解説する。

例. 関数内外での変数のスコープ

```
> var x = 123; [Enter] ←関数外部での変数 x の宣言
undefined
> function fnc() { [Enter] ←関数定義の記述の開始
...     var x = 456; [Enter] ←関数内部での変数 x の宣言
...     console.log( x ); [Enter] ←変数 x の出力
... } [Enter] ←関数定義の記述の終了
undefined ←関数宣言の処理結果
```

これは、関数定義の記述の内外で同じ名前の変数を宣言する例である。次に変数 x の値を確認する例を示す。

例. 関数の実行と変数 x の確認（先の例の続き）

```
> fnc() [Enter] ←関数を実行して
456 ←関数内スコープの変数 x の値を確認
undefined ← fnc の戻り値
> x [Enter] ←関数 fnc の外部で変数 x の値を確認すると
123 ←当初の値のまま
```

この例から、同じ名前の変数 x が関数の内外で異なるものとして扱われることがわかる。

3.10.1.1 変数の使用における安全でない例

関数内部で var, let, const で宣言されていない変数については関数外部のスコープの変数として扱われる。

例. 関数内部で宣言なしで変数を使用する例

```
> var x = 123; [Enter] ←関数外部での変数 x の宣言
undefined
> function fnc() { [Enter] ←関数定義の記述の開始
...     console.log( x ); [Enter] ←変数 x の出力
...     x = 456; [Enter] ←変数 x に変更を加える
... } [Enter] ←関数定義の記述の終了
undefined ←関数宣言の処理結果
```

これは、関数 fnc 外部の変数 x を関数内で使用する例である。このような定義の後で動作を確認する。（次の例）

例. 関数内で関数外部の変数を変更する例（先の例の続き）

```
> fnc() [Enter] ←関数を実行
123 ←関数外部の変数 x の値が確認できる
undefined ← fnc の戻り値
> x [Enter] ←関数 fnc の外部で変数 x の値を確認すると
456 ←値が変更されている
```

関数内部で関数外部のスコープの変数 x の値にアクセスできることがわかる。これは実際のプログラミングにおいては安全ではない。関数内部の変数は var, let, const で宣言して使用するべきである。

■ 更に危険な例

未使用の変数に対して関数内部で var, let, const 宣言なしで値を割り当てると、その変数はグローバルスコープの変数となる。

例. 関数内部でグローバルスコープの変数を作ってしまうケース

```
> function fnc() { [Enter] ←関数定義の記述の開始
...     y = 789; [Enter] ←宣言せずに変数 y に代入
...     console.log( y ); [Enter] ←変数 y の出力
... } [Enter] ←関数定義の記述の終了
undefined ←関数宣言の処理結果
```

このように定義された関数 fnc の内部では宣言文なしで変数 y に値が割り当てられており、fnc の実行によって y はグローバル変数となる。（次の例）

例. 関数実行後の変数 `y` の確認 (先の例の続き)

```
> fnc()  ←関数を実行
789      ←変数 y の値が確認できる
undefined ← fnc の戻り値

> y  ←変数 y の値を確認すると
789      ←グローバル変数として存在している
```

これも不用意にグローバル変数を作ってしまう原因となるので注意すべきである.

3.10.2 引数に関する事柄

3.10.2.1 仮引数と実引数

`function` 文などで関数を定義する際に記述する引数 (未だ具体的な値を持っていない) を**仮引数**と呼び, 実際に関数を呼び出す際に与える引数 (具体的な値) を**実引数**と呼ぶ.

3.10.2.2 任意の個数の引数を扱う方法

引数に**スプレッド構文**⁶⁰を応用すると, 関数の定義や呼び出しの際に任意の個数の引数を扱うことができる.

例. 任意の個数の引数を受け取って合計を求める関数

```
> function goukei( ...a ) {  ←スプレッド構文による仮引数
  let r = 0; 
  for ( n of a ) r += n; 
  return r 
}  ←関数定義の記述の終了
undefined      ←関数定義の処理の結果

> goukei(1,2)  ← 2 つの実引数を与えて呼び出す
3             ←関数の戻り値

> goukei(1,2,3,4,5)  ← 5 つの実引数を与えて呼び出す
15            ←関数の戻り値
```

この例の関数 `goukei` のようにスプレッド構文で受け取った引数は, 関数内部では配列 (Array) として扱われる.

関数呼び出しの際の実引数に配列をスプレッド構文で与えることができる.

例. 実引数に配列をスプレッド構文で渡す (先の例の続き)

```
> a = [1,2,3,4,5,6,7,8,9,10]  ←この配列を
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

> goukei( ...a )  ←スプレッド構文で実引数に渡す
55              ←関数の戻り値
```

3.10.2.3 仮引数にデフォルト値を設定する方法

関数の引数にはデフォルト値を設定することができる. このためには, 関数定義の記述の際に, 仮引数に値の代入を記述する.

例. デフォルト値を持つ関数の定義

```
> function kasan4( a=1, b=2 ) {  ←仮引数にデフォルト値を設定
  return a+b; 
} 
undefined      ←関数定義の処理の結果
```

このように定義することで, 関数を呼び出す際に実引数の省略が可能になる. (次の例)

⁶⁰ 「3.6.5 スプレッド構文」(p.66) で解説した.

例. 呼び出し時の実引数の省略（先の例の続き）

```
> kasan4() [Enter]    ←実引数を完全に省略
3          ←デフォルト値による計算結果
> kasan4(5) [Enter]   ← 2 番目の実引数を省略
7          ←デフォルト値によって 5+2 となる
> kasan4(undefined,5) [Enter] ← 1 番目の実引数を省略したことになる
6          ←デフォルト値によって 1+5 となる
```

3.10.3 関数自体のスコープ

function 宣言文で定義された関数は、その定義を記述する関数の内部でのみ有効である。次に示す例は関数 fout の定義の内部で関数 fin の定義を記述するものである。

例. 関数 fout の中で定義された関数 fin

```
> function fout() { [Enter]    ←関数 fout の定義の記述の開始
  function fin() { [Enter]    ←関数 fin の定義の記述の開始
    console.log("message from fin"); [Enter]
  } [Enter]    ←関数 fin の定義の記述の終了
  fin(); [Enter]    ← fout の内部で fin を呼び出す記述 (1)
} [Enter]    ←関数 fout の定義の記述の終了
undefined    ← fout の宣言の戻り値
```

この関数 fout を呼び出すと、fout はその内側の関数 fin を呼び出す。(次の例)

例. 関数 fout の呼び出し（先の例の続き）

```
> fout() [Enter]    ←外側の関数の呼び出し
message from fin [Enter] ←先の例の (1) の記述による出力
undefined
```

しかし、内部の関数 fin を直接的に呼び出す試みは失敗する。(次の例)

例. fout 内の fin を呼び出す試み（先の例の続き）

```
> fin() [Enter]    ←内側の関数 fin の呼び出しを試みると…
Uncaught ReferenceError: fin is not defined ←エラーとなる
```

以上のことから、fin は fout 内に限定された関数であることがわかる。ただし、function 宣言で定義された関数はブロックスコープではない。(次の例)

例. ブロック内で定義された関数 fin

```
> { [Enter]    ← function 宣言ではない通常のブロックの開始
  function fin() { [Enter]    ←関数 fin の定義の開始
    console.log("message from fin"); [Enter]    ←関数 fin の処理 (1)
  } [Enter]    ←関数 fin の定義の終了
  fin(); [Enter]    ←ブロック内で関数 fin を呼び出す
} [Enter]    ←ブロックの終了
message from fin    ←関数 fin による出力
undefined           ← fin からの戻り値
```

これは、ブロックをそのまま実行する処理なので、直ちに関数 fin が呼び出されている。

続いて、関数 fin を直接呼び出す。(次の例)

例. ブロック外で関数 fin を呼び出す試み（先の例の続き）

```
> fin() [Enter]    ←関数 fin の呼び出し
message from fin    ←関数 fin による出力
undefined           ← fin からの戻り値
```

ブロック外で fin を呼び出すことができています。これにより、function 宣言された関数はブロックスコープではないことがわかる。

3.10.4 関数式

関数を定義する別の方法に**関数式**の記述がある。関数式は先の function 宣言文と同様の記述形式であるが、式として扱われる。(次の例)

例. 関数式 (1)

```
> f = function kasan(x,y) { return x+y; } [Enter] ←関数の記述を式として変数 f に代入
[Function: kasan] ← undefined ではない
> f(3,4) [Enter] ←関数の呼び出し
7 ←関数 f の戻り値
```

関数 f が実行できることがわかる。ただしこの場合は kasan という関数名は使えない。

例. 注意すべきこと (先の例の続き)

```
> kasan(3,4) [Enter] ←これを試みると…
Uncaught ReferenceError: kasan is not defined ←エラーとなる
```

関数式の記述の際は関数名を省略することができる。関数名の無い関数式を**無名関数**あるいは**匿名関数**と呼ぶことがある。関数式は **Function オブジェクト** である。

例. 関数式 (2)

```
> f = function(x,y) { return x+y; } [Enter] ←関数名を省略
[Function: f] ← Function オブジェクトである
> f(3,4) [Enter] ←関数の呼び出し
7 ←関数 f の戻り値
```

関数式を let で変数に代入すると、その関数はブロックスコープとなる。(次の例)

例. ブロックスコープの関数

```
> { [Enter] ←ブロックの開始
  let fblk = function() { [Enter] ←関数式を let で変数に代入
    console.log("message from fblk"); [Enter] ←出力処理 (1)
  } [Enter] ←関数式の記述の終了
  fblk(); [Enter] ←関数の呼び出し
} [Enter] ←ブロックの終了
message from fblk ←関数 fblk からの出力
undefined ←関数 fblk の戻り値
```

この例の関数 fblk はブロックスコープとなり、ブロックの外では無効である。(次の例)

例. ブロック外での fblk の呼び出し (先の例の続き)

```
> fblk() [Enter] ←関数の呼び出しを試みると…
Uncaught ReferenceError: fblk is not defined ←エラーとなる
```

3.10.4.1 即時実行関数

関数式の直後に引数並びを記述し、それを即時実行することができる。このような形式の関数を**即時実行関数** (IIFE : Immediately Invoked Function Expression) と言う。

例. 即時実行関数

```
> (function(x,y) { return x+y; })(3,4) [Enter]
7
```

3.10.4.2 アロー関数式

function による関数式よりも機能が少し限定されるが、より簡潔な表現ができる**アロー関数式**が使用できる。

書き方： (引数並び)=>実行部

「引数並び」に受け取った値を用いて「実行部」を実行する。「実行部」には式を 1 つ、あるいはブロックを与える。以下にアロー関数式で関数を定義する例を示す。

例. 加算関数 f1 の定義と実行

```
> f1 = (x,y) => { return x+y; } [Enter]    ←アロー関数式
[Function:  f1]
> f1(3,4) [Enter]    ←呼び出し
7          ←戻り値 (計算結果)
```

この例の f1 と同等の関数をもう少し簡単に記述することができる。(次の例)

例. 加算関数 f2 の定義と実行

```
> f2 = (x,y) => x+y [Enter]    ← return を使わずに戻り値の式を直接記述
[Function:  f2]
> f2(3,4) [Enter]    ←呼び出し
7          ←戻り値 (計算結果)
```

引数の個数が 1 個の場合は更に簡単な記述が可能である。次の例は、与えられた 1 つの数値を 2 倍する関数 dbl を実装する例である。

例. 値を 2 倍する関数 dbl

```
> dbl = x=>2*x [Enter]    ←仮引数に括弧を付けずに記述できる
[Function:  dbl]
> dbl(4) [Enter]    ←呼び出し
8          ←戻り値 (計算結果)
```

この例では、2 倍する関数式を $x \Rightarrow 2x$ と記述しており、値の変換規則のような表現ができる。

アロー関数式で即時実行関数を実現することもできる。

3.10.5 関数の再帰的呼び出し

自分自身を再帰的に呼び出す関数を定義することができる。例えば次のような関数の定義について考える。

$$\text{sum}(n) = \begin{cases} n < 1 & \rightarrow 0 \\ n \geq 1 & \rightarrow n + \text{sum}(n-1) \end{cases}$$

これは 0～n までの整数の合計を求める関数 $\text{sum}(n)$ を数学的帰納法に基づいて定義するものである。この形式の定義は関数の再帰的定義でそのまま実装できる。(次の例)

例. $\text{sum}(n)$ を再帰的に定義する

```
> function sum(n) { [Enter]    ←関数定義の記述開始
  if ( n < 1 ) { [Enter]
    return 0; [Enter]
  } else { [Enter]
    return n+sum(n-1); [Enter]
  } [Enter]
} [Enter]    ←関数定義の記述終了
```

これを実行する例を次に示す。

例. 関数 sum の実行 (先の例の続き)

```
> sum(0) [Enter]    ← 0 のみの合計
0
> sum(10) [Enter]    ← 0～10 の合計
55
> sum(1000) [Enter]    ← 0～1000 の合計
500500
```

正しく計算できていることがわかる。ただし関数の呼び出しにおいては、呼び出す前のその関数の内部状態（ローカル変数など）がシステムのスタックに保存されるため、呼び出し回数が大きすぎる場合は言語処理系に例外（エラー）が発生する。(次の例)

例. 多すぎる再帰的呼び出し（先の例の続き）

```
> sum(10000) [Enter]    ← 0~10000 の合計の試み
Uncaught RangeError: Maximum call stack size exceeded    ←エラーとなる
    at sum (<anonymous>:2:5)
    at sum (<anonymous>:5:18)
    ⋮
    (以降省略)
    ⋮
```

演習課題. 再帰的定義に依らない形で関数 sum を実装せよ.

記憶資源を急激に消費する問題に対しては再帰的関数定義を用いるべきではない. 例えば次のような 0 以上の整数に対するフィボナッチ数を求める関数 fib を考える.

$$fib(n) = \begin{cases} n < 2 & \rightarrow 1 \\ n \geq 2 & \rightarrow fib(n-1) + fib(n-2) \end{cases}$$

この関数も再帰的な関数定義によって実現できる. (次の例)

例. fib(n) を再帰的に定義する

```
> function fib(n) { [Enter]    ←関数定義の記述開始
  if ( n < 2 ) { [Enter]
    return 1; [Enter]
  } else { [Enter]
    return fib(n-1) + fib(n-2); [Enter]
  } [Enter]
} [Enter]    ←関数定義の記述終了
```

この関数 fib は, 再帰的呼び出しの度に自分自身を呼び出す回数が 2 倍になり, 処理時間が急速に大きくなる. また, 状態を保存するスタックが急激に大きくなり, 言語処理系に例外 (エラー) が発生することがある. (次の例)

例. 関数 fib の実行 (先の例の続き)

```
> fib(0) [Enter]
1
> fib(10) [Enter]
89
> fib(30) [Enter]    ←このあたりから応答時間が目立つ
1346269
> fib(60) [Enter]    ←応答が無くなる…
```

関数の再帰的定義は, 複雑な問題を扱う場合の理解を平易にするが, 以上の例で見たように, 計算時間や記憶資源の消費が大きくなるので十分に注意すること.

演習課題. 再帰的定義に依らない形で関数 fib を実装せよ.

3.10.6 ネストされた関数定義 (内部関数)

関数定義の内部で更に別の関数を定義することができる. 関数の内部で定義される関数を**内部関数**と呼び, それを包含する関数を**外部関数**と呼ぶ. 内部関数は外部関数のスコープにおいて有効であり, 通常は外部関数の処理が終了すると消滅する⁶¹. 内部関数の内側では更に別の関数を定義することができ, 必要なだけ関数定義を入れ子にする (ネストする) ことができる. 関数定義の入れ子は, 関数内に一時的もしくは局在的な関数を作成することを可能にする.

3.10.6.1 クロージャ

ネストされた関数定義の応用に**クロージャ**がある. 通常の場合, 関数は実行が終了するとその内部状態 (ローカル変数をはじめとするオブジェクト) は全て失われる. 従って関数は通常の場合, 実行の終了後に持続する状態を保持することができない. このことを次の例で確認する.

⁶¹後に解説するクロージャとしての内部関数は, 外部関数が終了した後も存続する.

例. 試み 1-1

```
> function f1() { Enter ←関数 f1 の定義の開始
...     let c = 0; Enter
...     return ++c; Enter
... } Enter ←関数 f1 の定義の終了
undefined ←関数定義処理の結果
```

これは、関数内部の変数 *c* をインクリメントしてそれを返す関数 *f1* の定義である。これを実行すると次のようになる。

例. 試み 1-2 (先の例の続き)

```
> f1() Enter ←関数 f1 の実行
1 ←戻り値
> f1() Enter ←関数 f1 の再実行
1 ←前回と同じ戻り値
```

関数 *f1* の実行の度にその内部の変数 *c* の値が 0 にリセットされるので、戻り値は毎回同じになっている。これに対して次の例のような関数定義を考える。

例. 試み 2-1

```
> function extFunc() { Enter ←外部関数の定義の開始
...     let c = 0; Enter
...     function f1() { Enter ←内部関数の定義の開始
...         return ++c; Enter
...     } Enter ←内部関数の定義の終了
...     return f1; Enter ←上記の内部関数自体を戻り値とする
... } Enter ←外部関数の定義の終了
undefined ←関数定義処理の結果
```

この例では、関数 *extFunc* の内部で別の関数 *f1* が定義されている。関数 *f1* はその外部関数のスコープにある変数 *c* の値をインクリメントしたものを返す。

ここで重要な点が、関数 *extFunc* は内部関数 *f1* の戻り値ではなく、*f1* そのものを戻り値として返していることである。すなわち、関数 *extFunc* の戻り値は単なる数値ではなく、それ自体が実行可能な関数である。(次の例)

例. 試み 2-2 (先の例の続き)

```
> f1Closure = extFunc() Enter ←関数 extFunc の実行
[Function: f1] ← extFunc の内部関数 f1 そのものが得られている
```

ここで得られた *f1Closure* は内部関数 *f1* の戻り値ではなく、*f1* そのものである。従ってこの *f1Closure* は実行可能である。(次の例)

例. 試み 2-3 (先の例の続き)

```
> f1Closure() Enter ←関数 f1Closure の初回の実行
1 ← extFunc のスコープの変数 c の値が得られた
> f1Closure() Enter ←関数 f1Closure の 2 回目の実行
2 ← extFunc のスコープの変数 c の値が得られた (前回の値をインクリメントした値)
> f1Closure() Enter ←関数 f1Closure の 3 回目の実行
3 ← extFunc のスコープの変数 c の値が得られた (前回の値をインクリメントした値)
```

この実行結果からわかることのポイントは下記の通り。

- 外部関数 *extFunc* は既に実行終了している。
- 内部関数 *f1* は既に実行終了した *extFunc* のスコープの変数 *c* を使っている。
- 外部関数 *extFunc* のスコープの内部関数 *f1* が *extFunc* 実行終了後も存在している。
- 外部関数 *extFunc* のスコープの変数 *c* が *extFunc* 実行終了後も存在している。

通常の場合、関数の実行が終了すると、その関数内の変数や内部関数は消滅するが、今回の例の *f1Closure* ように、外部関数の更に外部から内部関数が参照されていると、その内部関数は外部関数の終了後も存続する。また、内部関数 *f1* が外部関数のスコープの変数 *c* を使用しているため、外部関数が終了後もその変数 *c* は存続する。

今回の例のように、スコープの内部-外部の参照関係によって内部状態を維持する関数をクロージャと呼ぶ。また、クロージャを包む外部関数をエンクロージャと呼ぶことがある。

クロージャは、後で解説するオブジェクト指向プログラミングとは別の方法で、機能とデータを保持する実体を実装する方法となる。

3.10.7 オブジェクトとメソッドの考え方

オブジェクトのプロパティには関数式を与えることもでき、これにより、オブジェクトに対するメソッドの概念が生まれる。

例. メソッドを持つオブジェクト

```
> a = { v:3, f:function(){console.log("メソッドによる出力");} } Enter ←オブジェクトの作成
{ v: 3, f: [Function: f] } ←プロパティfが Function オブジェクトを持っている
> a.v Enter ←プロパティvは
3 ←数値
> a.f() Enter ←メソッド
メソッドによる出力 ←その実行結果
undefined ←メソッドの戻り値
```

オブジェクト a が、実行可能なメソッド f を持っていることがわかる。オブジェクトにメソッドを与える方法として、**メソッドの短縮記法**がある。次の例は上の例と同等の機能を持ったオブジェクトを作成するものである。

例. メソッドを持つオブジェクト：メソッドの短縮記法による

```
> a = { v:3, f(){console.log("メソッドによる出力");} } Enter ←オブジェクトの作成
{ v: 3, f: [Function: f] } ←プロパティfが Function オブジェクトを持っている
> a.v Enter ←プロパティvは
3 ←数値
> a.f() Enter ←メソッド
メソッドによる出力 ←その実行結果
undefined ←メソッドの戻り値
```

このように、メソッドをプロパティとして記述する際に、コロン「:」による表記を使わずに

プロパティ名 (引数並び) { 実行する処理 }

と書くことができる。

メソッド内では、それが所属するオブジェクト自体は this キーワードで参照できる。

例. this キーワードによる自身の参照

```
> a = { v:3, f(){console.log("メソッドによる出力:v="+String(this.v));} } Enter ← this を使用
{ v: 3, f: [Function: f] }
> a.f() Enter ←メソッドの実行
メソッドによる出力:v=3 ←その実行結果：自身のプロパティvが参照できている
undefined ←メソッドの戻り値
```

3.10.8 ジェネレータ関数

通常関数は、`return` が実行されると呼び出し元に値を返して終了し、ローカル変数など関数内部の状態が破棄される。これに対してジェネレータ関数は `yield` によって値を返し、その後も実行が継続される。ジェネレータ関数は次のような形で定義する。

function* 関数名 (仮引数の並び) { 処理の記述 }

次に 0 ～ 2 の数値を返すジェネレータ関数の例を示す。

例. 0 ～ 2 の数値を返すジェネレータ関数 `gnrf1`

```
> function* gnrf1() { Enter
...     let n; Enter
...     for ( n = 0; n < 3; n++ ) { Enter
...         yield n; Enter
...     } Enter
... } Enter
undefined      ←関数定義の処理結果
```

この関数 `gnrf1` の中では `for` による反復処理が記述されており、その中の `yield` で `n` の値が呼び出し元に返される。ただし、`yield` によって値が返されてもこの関数の処理は終了せず、次の呼び出し時も `for` の反復がそのまま継続される。

通常関数が `Function` オブジェクトであるのに対し、`function*` で定義されたものは `GeneratorFunction` オブジェクトであり、これを実行すると `Generator` オブジェクトが得られる。このオブジェクトに対して `next` メソッドを実行することで `function*` に記述した処理が実際に実行される。また `next` メソッドによって得られるものは

{ value:値, done:状態 }

の形式のオブジェクトである。すなわち、`yield` で返された値が `value` プロパティの「値」である。また、`Generator` の処理が終了していれば `done` プロパティの値は `true`、実行の途中であれば `false` となる。

例. `GeneratorFunction` と `Generator` (先の例の続き)

```
> gnrf1.constructor.name Enter      ← gnrf1 の型を調べる
'GeneratorFunction'      ←判明した型
> var g1 = gnrf1() Enter      ← Generator を生成
undefined                 ← var による代入処理の結果
> Object.getPrototypeOf(g1) Enter    ← g1 の型を調べる
Object [Generator] {}    ← Generator である
```

これで、実際に実行できる `g1` が得られた。次に、これに対して `next` メソッドを用いて処理を実行する。

例. `Generator` オブジェクト `g1` の実行 (先の例の続き)

```
> g1.next() Enter      ← g1 の実行
{ value: 0, done: false }    ←最初の状態
> g1.next() Enter      ← g1 の実行
{ value: 1, done: false }    ←次の状態
> g1.next() Enter      ← g1 の実行
{ value: 2, done: false }    ←次の状態
> g1.next() Enter      ← g1 の実行
{ value: undefined, done: true } ←終了した状態
```

`for` 文が中断されずに実行され、順番に値が得られている様子がわかる。また、最終状態では `done` プロパティが `true` になり、`value` プロパティが `undefined` になっているのがわかる。

■ ジェネレータ関数の用途

ジェネレータ関数は、動的にデータを生成するオブジェクトとして用いることができる。「3.6 データ構造」(p.50) で解説した各種のデータ構造では、各要素が既に存在しているものとして保持されているが、ジェネレータ関数では、要求に応じて (`next` メソッドによって) 値を順次生成するオブジェクトとして用いることができる。

ジェネレータ関数は for...of 構文やスプレッド構文で利用できる。

例. for...of 構文でのジェネレータ関数の応用（先の例の続き）

```
> for ( r of gnrf1() ) { 
...     console.log( r ); 
... } 
0      ← console.log による出力
1
2
undefined ← for 文の実行結果
```

例. スプレッド構文でのジェネレータ関数の応用（先の例の続き）

```
> [-1, ...gnrf1(), 3]  ←スプレッド構文
[-1, 0, 1, 2, 3 ]    ←得られた配列
```

3.10.8.1 ジェネレータ関数の入れ子

yield* を使うとジェネレータ関数を入れ子にすることができる。また、ジェネレータ関数以外のイテラブルなオブジェクトの要素を戻り値にすることもできる。

例. yield* の使用例（先の例の続き）

```
> function* gnrf2() { 
...     yield -1; 
...     yield* gnrf1(); 
...     yield* "ab"; 
... } 
undefined ←関数定義の処理結果
```

最後の yield* は文字列の各文字を戻り値として順次返す。

例. 上記のジェネレータ関数の実行（先の例の続き）

```
> [ ...gnrf2() ]  ←スプレッド構文に gnrf2 を応用
[-1, 0, 1, 2, 'a', 'b' ]    ←得られた配列
```

3.11 イテラブルオブジェクトとイテレータ

next メソッドを持ち、それが呼び出されると { value: 値, done: ブール値 } を返すものをイテレータという。ジェネレータ関数が返す Generator オブジェクトもイテレータである。Generator オブジェクトでなくてもイテレータを作ることができる。

例. カスタムイテレータ

```
1 var L = {
2   start:0, end:2, pos:0,
3   next: function() {
4     if ( this.pos > this.end ) {
5       return {value:undefined,done:true}
6     } else {
7       const r = {value:this.pos,done:false}
8       this.pos++;
9       return r
10    }
11  }
12 }
```

この例のコードを Web ブラウザのコンソール、もしくは Node.js のコンソールで実行すると L にイテレータが得られる。

例. next メソッドの実行（先の例の続き）

```
> L.next() Enter
{ value: 0, done: false } ←最初の値
> L.next() Enter
{ value: 1, done: false } ←次の値
> L.next() Enter
{ value: 2, done: false } ←最後の値
> L.next() Enter ←終了後は
{ value: undefined, done: true } ←このような値
```

next メソッドが機能している様子がわかる。

上記 L は後の例でも使用するので下記のようにして初期化しておく。

例. L の初期化（先の例の続き）

```
> L.start = 0; L.pos = 0; L Enter ← L の初期化と内容確認
{ start: 0, end: 2, pos: 0, next: [Function: next] } ←初期化された
```

3.11.1 イテラブルオブジェクト

イテラブルオブジェクト（反復可能オブジェクト）は、Symbol.iterator の名前のメソッドを持ち、そのメソッドを呼び出すとイテレータを返すオブジェクトである。イテラブルオブジェクトは for...of 文で使うことができる。

先の例で使ったイテレータ L を返すイテラブルオブジェクトを作る例を次に示す。

例イテラブルオブジェクトを作る（先の例の続き）

```
1 var itr = {
2   [Symbol.iterator]: function() {
3     return L;
4   }
5 }
```

この例で作成したオブジェクト itr は Symbol.iterator の名前のメソッドが定義されており、そのメソッドはイテレータ L を返す。

注意) Symbol.iterator はリテラルではなく式である。従って、この名前のメソッドを定義する場合、メソッド名は計算されたプロパティ名⁶²を与えるため角括弧 [] で括っている。

⁶² 「3.6.2.6 計算されたプロパティ名」(p.60) で解説している。

上の例で作成したイテラブルオブジェクト itr を for...of 文で使用する例を次に示す。

例. itr を for...of 文で使用する (先の例の続き)

```
> for (const value of itr) { 
...     console.log(value); 
... } 
```

0 ← console.log による出力
1
2
undefined ← for 文の実行結果

重要)

文字列や配列, Map, Set などイテラブルオブジェクトであり, Symbol.iterator の名前のメソッドを持つ。

例. 各種データ構造がイテラブルオブジェクトであることを確認する

```
> "abc"[Symbol.iterator]       ←文字列の Symbol.iterator プロパティは  
[Function: [Symbol.iterator]]      ← Function オブジェクトである  
> a = [1,2,3]; a[Symbol.iterator]       ←配列の Symbol.iterator プロパティは  
[Function: values]                      ← Function オブジェクトである  
> m = new Map([[1,2],[2,4]]); m[Symbol.iterator]       ← Map オブジェクトの場合は  
[Function: entries]                      ← Function オブジェクトである  
> s = new Set([1,2,3]); s[Symbol.iterator]       ← Set オブジェクトの場合は  
[Function: values]                      ← Function オブジェクトである
```

ジェネレータ関数が生成する Generator オブジェクトもイテラブルオブジェクトである。(確認されたい)

3.12 オブジェクト指向プログラミング (OOP)

JavaScript では、プログラマが自由に独自のオブジェクト⁶³を新たなデータ型であるかのように定義することができる。それを行うためには**コンストラクタ**を function 宣言文で定義する。すなわち、定義されたコンストラクタを用いてプログラマが設計した独自のオブジェクトを生成する。コンストラクタから生成された独自のオブジェクトを**インスタンス**と言う。

以下に例を挙げてコンストラクタの定義とインスタンス生成の流れについて解説する。

【サンプル事例】

name (名前), weight (体重), height (身長) の 3 つの値を保持する Human オブジェクトを定義する。

例. Human オブジェクトのコンストラクタ

```
> function Human(name,weight,height) { Enter    ←コンストラクタの定義開始
  this.name = name; Enter    ← name プロパティの設定
  this.weight = weight; Enter    ← weight プロパティの設定
  this.height = height; Enter    ← height プロパティの設定
} Enter    ←コンストラクタの定義終了
undefined          ← function 文の実行結果
```

これで Human オブジェクトのコンストラクタが定義できた。コンストラクタ内の this は生成されるインスタンスを意味する。コンストラクタでは基本的には return は書かない。上の例では、引数に与えられた値を当該インスタンスの name, weight, height に設定する。

実際にインスタンスを生成するにはコンストラクタに対して new 演算子を使用する。

例. インスタンスの生成 (先の例の続き)

```
> h1 = new Human("佐藤 太郎",70,1.71) Enter    ←インスタンスの生成
Human { name: '佐藤 太郎', weight: 70, height: 1.71 }    ←得られたインスタンス
```

これで変数 h1 に Human オブジェクトのインスタンスが得られた。インスタンスの表示は V8 エンジンでは

コンストラクタ名 { プロパティの並び }

のように表示され、SpiderMonkey では

object { プロパティの並び }

のように表示される。

Human オブジェクトも通常のオブジェクトと同様の方法でプロパティにアクセスできる。

例. プロパティの参照 (先の例の続き)

```
> h1["name"] Enter    ←角括弧でプロパティを参照
'佐藤 太郎'          ←値
> h1.name Enter    ←ドット表記でプロパティを参照
'佐藤 太郎'          ←値
```

例. プロパティへの値の設定 (先の例の続き)

```
> h1.name = "佐藤 次郎" Enter    ←値の設定 (変更)
'佐藤 次郎'
> h1 Enter    ←内容確認
Human { name: '佐藤 次郎', weight: 70, height: 1.71 }    ←変更されている
```

new 演算子で別のインスタンスを次々と生成することができる。

例. 同じ構造を持った別のインスタンスを生成 (先の例の続き)

```
> h2 = new Human("鈴木 花子",65,1.64) Enter    ← new 演算子でインスタンス生成
Human { name: '鈴木 花子', weight: 65, height: 1.64 }    ←先の h1 とは別のインスタンス
```

⁶³ 「3.6.2 オブジェクト: Object」(p.58) で解説した、JavaScript に組み込みのオブジェクトとは異なるもの。

3.12.1 メソッドの実装

プログラマが定義するコンストラクタにメソッドを実装するには、コンストラクタの prototype プロパティに関数式を与える。

書き方： コンストラクタ.prototype.メソッド名 = 関数式

「関数式」で記述されたメソッドを「メソッド名」の名前で「コンストラクタ」から生成されるインスタンスで利用可能にする。先に示した Human オブジェクトのコンストラクタに bmi メソッドを実装する例を次に示す。

例. bmi メソッドの定義（先の例の続き）

```
> Human.prototype.bmi = function() { Enter      ← new 演算子でインスタンス生成
    return this.weight / this.height ** 2; Enter      ← BMI 値を算出して返す
  } Enter      ←関数式の記述の終了
    [Function (anonymous)]      ←無名関数としてメソッドが登録された
```

この後、各 Human インスタンスに対して bmi メソッドが実行できる。（次の例）

例. bmi メソッドの実行（先の例の続き）

```
> h1.bmi() Enter      ← h1 に対して実行
23.938989774631512      ← BMI 値
> h2.bmi() Enter      ← h2 に対して実行
24.167162403331353      ← BMI 値
```

3.12.1.1 prototype プロパティ

prototype プロパティはコンストラクタの関数オブジェクトに与えるものである。このプロパティは、当該コンストラクタから生成されたインスタンスが共有するものである。実際に、先の例で生成したインスタンス h1, h2 においてもその内部に bmi という名前のプロパティは確認できない。

例. インスタンスの確認（先の例の続き）

```
> h1 Enter      ←内容確認
Human { name: '佐藤 次郎', weight: 70, height: 1.71 }      ← bmi というプロパティは無い
> h2 Enter      ←内容確認
Human { name: '鈴木 花子', weight: 65, height: 1.64 }      ← bmi というプロパティは無い
```

従って、h1.bmi() を実行すると、システムはまず当該インスタンスである h1 のプロパティ bmi を探し、そこに見つからなければコンストラクタ側の prototype プロパティに bmi というものがあるかを探す。このような機能により、同一のコンストラクタから生成された全てのインスタンスが共有するプロパティが実現できる。

次の例は、先の Human コンストラクタの prototype プロパティ内に更に kind なる共有プロパティを追加するものである。

例. 更なる共有プロパティの登録（先の例の続き）

```
> Human.prototype.kind = "primate" Enter      ←更なる共有プロパティ kind の追加
'primate'
> h1.kind Enter      ← h1 から kind を
'primate'      ←参照できる
> h2.kind Enter      ← h2 から kind を
'primate'      ←参照できる
```

例. 共有可能なプロパティの再確認（先の例の続き）

```
> Human.prototype Enter      ←共有プロパティの確認
{ bmi: [Function (anonymous)], kind: 'primate' }
```

この仕組みを応用すると、他のプログラミング言語におけるクラス変数と同等のものが実現できる。

■ constructor プロパティ

コンストラクタの prototype プロパティは constructor プロパティを持ち、当該コンストラクタの関数オブジェクトを保持している。

例. Human オブジェクトのコンストラクタ（先の例の続き）

```
> Human.prototype.constructor [Enter]    ←コンストラクタを調べる
[Function: Human]              ← Human コンストラクタの関数オブジェクト
```

3.12.2 インスタンスのコンストラクタを調べる方法

関数オブジェクトには name プロパティがあり、その関数名を保持している。従って、インスタンスのコンストラクタの関数名を調べることで、あるインスタンスを生成したコンストラクタを調べることができる。

例. インスタンスのコンストラクタを調べる（先の例の続き）

```
> h1.constructor.name [Enter]    ← h1 のコンストラクタ名を調べる
'Human'                    ← Human コンストラクタから生成されたことがわかる
```

3.12.3 Object.create によるオブジェクトの生成

new 演算子とコンストラクタによる方法とは別に、Object.create によってオブジェクトを生成する方法がある。

書き方： **Object.create(オブジェクト)**

「オブジェクト」に与えたオブジェクトをプロトタイプとする新たなオブジェクトを生成して返す。これに関して例を挙げて説明する。

先に例示した Human コンストラクタでは、生成されるインスタンスが共有するプロパティを Human.prototype プロパティが保持している。これを Object.create に与えて実行すると Human オブジェクトが生成される。ただしこの場合はコンストラクタによる初期化処理（Human 関数内の処理）は実行されない。

例. Object.create による Human オブジェクトの生成（先の例の続き）

```
> h0 = Object.create( Human.prototype ) [Enter]    ← Human オブジェクト生成
Human {}                                       ←得られた Human オブジェクト
```

この例からわかるように得られた Human オブジェクト h0 は name, weight, height といったプロパティを持っていない。ただし、Human.prototype によって共有されているものは参照できる。（次の例）

例. kind プロパティの確認（先の例の続き）

```
> h0.kind [Enter]    ←共有プロパティを確認
'primate'            ←参照できている
```

以上のことを応用すると、OOP におけるプロパティの**継承**の仕組みが実現できる。

3.12.4 プロパティ継承の仕組み

オブジェクト指向プログラミングでは、先に定義されたデータ構造の性質を引き継いだ（**継承**した）、更に高機能な別のデータ構造を定義するという流れでシステム開発を行う。JavaScript では、プログラマが独自に定義するデータ構造のコンストラクタに prototype プロパティを持たせ、生成されるインスタンスに共通する機能（共通するプロパティ）を与えているが、これを継承する形で別のデータ構造のコンストラクタを定義することで発展的なシステム開発が可能になる。

ここでは、データ構造の発展的な実装について例を挙げて解説する。先に Human オブジェクト（親）のコンストラクタを定義する例を挙げたが、そのオブジェクトの機能を引き継いだ American オブジェクト（子）と Japanese オブジェクト（子）のコンストラクタを実装する。

例. American オブジェクトのコンストラクタ（先の例の続き）

```
> function American(name,weight,height) { [Enter]    ←コンストラクタの定義の開始
  Human.call(this,name,weight,height); [Enter]    ← Human コンストラクタの呼び出し
} [Enter]    ←コンストラクタの定義の記述の終了
undefined [Enter]    ← function 文の実行結果

> American.prototype = Object.create(Human.prototype) [Enter]    ←プロパティの継承
Human {} [Enter]    ←継承されたプロトタイプ
```

ここで注意すべきこととして、American.prototype.constructor は Human コンストラクタのもので上書きされ

ていることがある。(次の例)

例. American.prototype.constructor の確認 (先の例の続き)

```
> American.prototype.constructor Enter    ←確認すると  
[Function: Human]    ← Human コンストラクタになっている
```

これを次のようにして修正する必要がある。

例. American コンストラクタの修正 (先の例の続き)

```
> American.prototype.constructor = American Enter    ←修正  
[Function: American]
```

これで、Human オブジェクトの性質を受け継いだ American オブジェクトのコンストラクタが定義できた。ここまでの例で重要なポイントが2つある。1つ目は call メソッドによって元になった Human オブジェクト (親) のコンストラクタを American オブジェクト (子) のコンストラクタから呼び出している点である。American オブジェクトは Human オブジェクトの一種でもあることから、American のインスタンスを生成する際は、その親となる Human オブジェクトとしての初期化処理をする必要があるのがこのような書き方となる。

書き方： 親のコンストラクタ.call(this, 引数並び)

「this」は、当該オブジェクト (上の例の American) が生成するインスタンスを意味するもので、このように call メソッドを使用することで「親のコンストラクタ」を this に対して実行することができる。

上の例における重要なポイントの2つ目は、

```
American.prototype = Object.create(Human.prototype)
```

という記述によって、Human オブジェクトのプロトタイプ (共有機能) を American オブジェクトのプロトタイプに引き継いでいる (継承している) 点である。(この際に、先に説明した constructor プロパティの上書きが起ってしまうので修正処理を行った) そして Human オブジェクトには無い American オブジェクト独特の機能を American.prototype に追加することができる。すなわち、親である Human オブジェクトの機能を引き継いだ上、子である American オブジェクト独自の機能を追加するという流れで発展的な機能拡張が実現できる。このように、親のコンストラクタの prototype プロパティを子のコンストラクタの prototype プロパティに連鎖的に次々と継承する構造を **プロトタイプチェーン**と呼ぶ。

上の例で定義された American オブジェクトのコンストラクタに独自のメソッドを追加する例を次に示す。

例. American オブジェクト独自の greet メソッドの実装 (先の例の続き)

```
> American.prototype.greet = function() { Enter    ← greet メソッドの定義の開始  
    console.log("Hello."); Enter  
} Enter    ← greet メソッドの定義の終了  
[Function (anonymous)] Enter    ←関数式の代入処理の戻り値
```

これは、American コンストラクタのプロトタイプに greet というメソッドを実装するものである。この後で American オブジェクトのインスタンスを生成する例を次に示す。

例. American のインスタンス生成 (先の例の続き)

```
> a1 = new American( "John", 90, 1.87 ) Enter    ← American インスタンスの生成  
American { name: 'John', weight: 90, height: 1.87 }    ←得られたインスタンス  
> a1.constructor.name Enter    ← a1 のコンストラクタの確認  
'American'    ← American オブジェクトであることがわかる
```

例. American のインスタンスのメソッド、プロパティ (先の例の続き)

```
> a1.bmi() Enter    ← Human (親オブジェクト) のメソッドが使える  
25.737081414967538  
> a1.kind Enter    ← Human (親オブジェクト) のプロパティが参照できる  
'primate'  
> a1.greet() Enter    ← American 独自の greet メソッドを実行  
Hello.    ←機能している  
undefined    ← greet メソッドの戻り値
```

American 独自の greet メソッドが機能している様子がわかる。勿論ではあるが、greet メソッドは Human オブジェクト（親）のインスタンスには使えない。（次の例）

例. Human インスタンスに greet メソッドを実行する試み（先の例の続き）

```
> h1.greet() Enter    ← Human インスタンスに greet メソッドは
Uncaught TypeError: h1.greet is not a function    ←使えない
```

同様の流れで、Human オブジェクト（親）の機能を引き継いだ（継承した）更に別の Japanese オブジェクト（子）のコンストラクタを定義して使用する例を次に示す。（次の例）

例. Human オブジェクトを継承した Japanese オブジェクトのコンストラクタ（先の例の続き）

```
> function Japanese(name,weight,height) { Enter    ← Japanese コンストラクタ
    Human.call(this,name,weight,height); Enter    ←親のコンストラクタの呼び出し
} Enter    ← Japanese コンストラクタの定義の記述の終了
undefined    ← function 文の実行結果

> Japanese.prototype = Object.create(Human.prototype) Enter    ←プロトタイプ継承
Human {}

> Japanese.prototype.constructor = Japanese Enter    ← constructor プロパティの修正
[Function: Japanese]

> Japanese.prototype.greet = function() { Enter    ← Japanese 独特の greet メソッド
    console.log("こんにちは。"); Enter
} Enter    ← greet メソッドの記述の終了
[Function (anonymous)]

> Japanese.prototype.omotenashi = function() { Enter    ← Japanese 独特の omotenashi メソッド
    console.log("いらっしやいませ。"); Enter
} Enter    ← omotenashi メソッドの記述の終了
[Function (anonymous)]
```

このように実装された Japanese は Human の子であり、American の兄弟である。また、Japanese は American も持たない omotenashi メソッドを備えている。（次の例）

例. Japanese のインスタンス生成（先の例の続き）

```
> j1 = new Japanese( "日本 太郎", 75, 1.76 ) Enter    ← Japanese インスタンスの生成
Japanese { name: '日本 太郎', weight: 75, height: 1.76 }    ←得られたインスタンス

> j1.constructor.name Enter    ← j1 のコンストラクタの確認
'Japanese'    ← Japanese オブジェクトであることがわかる
```

例. Japanese のインスタンスのメソッド、プロパティ（先の例の続き）

```
> j1.bmi() Enter    ← Human（親オブジェクト）のメソッドが使える
24.212293388429753

> j1.kind Enter    ← Human（親オブジェクト）のプロパティが参照できる
'primate'

> j1.greet() Enter    ← Japanese 独自の greet メソッドを実行
こんにちは。    ←機能している
undefined    ← greet メソッドの戻り値

> j1.omotenashi() Enter    ← Japanese 独自の omotenashi メソッドを実行
いらっしやいませ。    ←機能している
undefined    ← omotenashi メソッドの戻り値
```

Japanese 独自の greet, omotenashi メソッドが機能している様子がわかる。勿論ではあるが、omotenashi メソッドは Human オブジェクト（親）や American オブジェクト（兄弟）のインスタンスには使えない。（次の例）

例. Human, American インスタンスに omotenashi メソッドを実行する試み（先の例の続き）

```
> h1.omotenashi() Enter    ← Human インスタンスに omotenashi メソッドは
Uncaught TypeError: h1.omotenashi is not a function    ←使えない

> a1.omotenashi() Enter    ← American インスタンスに omotenashi メソッドは
Uncaught TypeError: a1.omotenashi is not a function    ←使えない
```

このように、コンストラクタのプロトタイプを継承し機能拡張をしながら次々と子孫のコンストラクタを実装することができる。上の例で実装した Japanese コンストラクタを更に継承した Kyotoite コンストラクタを実装する例を次に示す。

例. Japanese オブジェクトを継承した Kyotoite オブジェクトのコンストラクタ（先の例の続き）

```
> function Kyotoite(name,weight,height) { Enter      ← Kyotoite コンストラクタ
    Japanese.call(this,name,weight,height); Enter      ← 親のコンストラクタの呼び出し
} Enter      ← Kyotoite コンストラクタの定義の記述の終了
undefined      ← function 文の実行結果

> Kyotoite.prototype = Object.create(Japanese.prototype) Enter      ←プロトタイプの継承
Human {}

> Kyotoite.prototype.constructor = Kyotoite Enter      ← constructor プロパティの修正
[Function: Kyotoite]

> Kyotoite.prototype.omotenashi = function() { Enter      ← Kyotoite 独特の omotenashi メソッド
    console.log("おいでやす. "); Enter
} Enter      ← omotenashi メソッドの記述の終了
[Function (anonymous)]
```

このように実装される Kyotoite コンストラクタは、Human から始まる代々のプロトタイプを引き継いでいる。また、omotenashi メソッドは Japanese のものとは異なる機能を実行するものとして置き換えられている。(オーバーライド)

例. Kyotoite のインスタンス生成（先の例の続き）

```
> k1 = new Kyotoite( "祇園 彩乃", 58, 1.62 ) Enter      ← Kyotoite インスタンスの生成
Kyotoite { name: '祇園 彩乃', weight: 58, height: 1.62 }      ←得られたインスタンス

> k1.constructor.name      ← k1 のコンストラクタの確認
'Kyotoite'
```

例. Kyotoite のインスタンスのメソッド、プロパティ（先の例の続き）

```
> k1.bmi() Enter
22.10028959000152

> k1.kind Enter
'primate'

> k1.greet() Enter
こんにちは.
undefined

> k1.omotenashi() Enter      ← Kyotoite 独自の omotenashi メソッドの実行
おいでやす.
undefined
```

このように代々のプロパティが有効である。また Kyotoite の omotenashi メソッドのように、先祖のプロパティを子孫のプロパティで新たにオーバーライドすることができる。

3.12.5 class 構文による方法

これまでに解説したオブジェクト指向プログラミング（OOP）は**プロトタイプベース**⁶⁴の考え方に基づくものであり、既存のオブジェクトを**雛形**（原型：prototype）として参照し、それを複製して拡張するという考え方である。JavaScript の OOP はプロトタイプベースであり、コンストラクタの prototype プロパティを別のコンストラクタの prototype プロパティに複製して継承する形（**プロトタイプチェーン**）で新たなデータ構造（独自のオブジェクト）を発展的に実装する。

プロトタイプベース以外にも**クラスベース**の考え方に基づく OOP のスタイルもあり、理解が平易であり記述が簡便であることから、多くのプログラミング言語でクラスベースの OOP が採用されている。JavaScript においても、元来のプロトタイプベースの OOP の機能の上に**糖衣構文**（syntactic sugar）を導入することでクラスベースの OOP を

⁶⁴ インスタンスベースと呼ばれることもある。

実現している。それが、ここで解説する class 構文である。

クラスベースの OOP では、プログラマが実装するデータ構造とそれに付随する機能を**クラス**として定義する。クラスは抽象化されたデータ型であり、それを具体化した個別のオブジェクトを**インスタンス**として生成し、実際の処理に使用する。class 宣言文の書き方を次に示す。

書き方： class クラス名 { 定義内容 }

「クラス名」のクラス（型）を定義する。「定義内容」にはコンストラクタや当該クラスのインスタンスに対するメソッドなどを記述する。クラス定義内では、当該クラスのインスタンスを `this` と表記して扱う。

既存のクラスを継承した形で新たなクラスを定義することもできる。

書き方： class 新クラス名 extends 既存のクラス名 { 定義内容 }

3.12.5.1 コンストラクタ、メソッドの記述

クラスの定義内容としてコンストラクタを実装するには次のように記述する。

書き方： constructor(引数並び) { 定義内容 }

また、同様の形式でメソッドを実装することができる。

書き方： メソッド名 (引数並び) { 定義内容 }

プロトタイプベースの事例として先に示した Human, Japanese, American, Kyotoite オブジェクトをクラスとして定義する例を示す。

Human クラスを宣言する class 宣言文を次に示す。

記述例： Human クラスの宣言

```
1 class Human {
2     constructor(name,weight,height) {
3         this.name = name;
4         this.weight = weight;
5         this.height = height;
6     }
7     bmi() {
8         return this.weight / this.height ** 2;
9     }
10 }
```

コンストラクタ（2～6 行目）と bmi メソッド（7～9 行目）を定義している。

この宣言を行った後、インスタンスを生成する例を次に示す。

例. Human クラスのインスタンスの生成（先の例の続き）

```
> h1 = new Human("佐藤 太郎",70,1.71) 
Human { name: '佐藤 太郎', weight: 70, height: 1.71 } ←インスタンス
> h2 = new Human("鈴木 花子",65,1.64) 
Human { name: '鈴木 花子', weight: 65, height: 1.64 } ←インスタンス
```

コンストラクタが実行されてインスタンスが生成されている。これらに対して bmi メソッドを実行する例を次に示す。

例. bmi メソッドの実行（先の例の続き）

```
> h1.bmi() 
23.938989774631512
> h2.bmi() 
24.167162403331353
```

3.12.5.2 パブリックフィールド

class 宣言の中に**パブリックフィールド**⁶⁵を設置することで、生成されるインスタンスに共通のプロパティを与えることができる。また、static キーワードを付加してパブリックフィールドを設置すると**静的なフィールド**となり、それはインスタンスには属さず、クラスに属する。

パブリックフィールドに関する事例を後に示す。

⁶⁵パブリッククラスフィールドとも呼ばれる。

3.12.5.3 静的メソッド

static キーワードを付加してメソッドを実装すると、そのメソッドはインスタンスには属さず、クラスに属する。

■ パブリックフィールド、静的メソッドの実装例

先に例示した Human クラスの定義を改変し、パブリックフィールド、静的メソッドを実装したものを次に示す。

記述例：Human クラスの宣言（改）

```
1 class Human {
2     kind = "primate";
3     static total = 8e10;
4     constructor(name, weight, height) {
5         this.name = name;
6         this.weight = weight;
7         this.height = height;
8     }
9     bmi() {
10        return this.weight / this.height ** 2;
11    }
12    static abst() {
13        return "mammal";
14    }
15 }
```

2行目に kind への代入の記述があるが、これが Human クラスにおけるパブリックフィールドの1つとなる。この kind は生成されるインスタンスにプロパティとして組み込まれる。特にこのようなものをパブリックインスタンスフィールドと呼ぶことがある。また、3行目の total への代入の記述は静的なフィールドである total の設置であり、これは、生成されるインスタンスには属さず Human クラスに属する。

static キーワードの付いたパブリックフィールドをパブリック静的フィールドと呼ぶことがある。12~14行目に記述されている abst メソッドは静的メソッドであり、生成されるインスタンスに対しては実行できず、Human.abst() として Human クラスのメソッドとして実行される。

この宣言を行った後、インスタンスを生成する例を次に示す。

例. Human クラス（改）のインスタンスの生成（先の例の続き）

```
> h1 = new Human("佐藤 太郎",70,1.71) Enter
Human { kind: 'primate', name: '佐藤 太郎', weight: 70, height: 1.71 } ←インスタンス
> h2 = new Human("鈴木 花子",65,1.64) Enter
Human { kind: 'primate', name: '鈴木 花子', weight: 65, height: 1.64 } ←インスタンス
```

生成されたインスタンス全てにパブリックインスタンスフィールドである kind が組み込まれていることがわかる。この例からもわかるように、kind フィールドの設置はコンストラクタ内で

```
this.kind = "primate";
```

と記述した場合と同等の結果となる。このように、生成されるインスタンスのプロパティをパブリックインスタンスフィールドの形で記述すると、クラス定義の文書としての明示性を高めることにもなる。

パブリックフィールドの宣言においては、初期値の設定を必要としない場合はフィールド名のみの記述をしても良い。例えば上の「Human クラスの宣言（改）」の2,3行目において「kind;」、「static total;」と宣言することが文法的に可能である。

パブリック静的フィールド total はインスタンスのプロパティではないことを次に示す。

例. total フィールドの確認（先の例の続き）

```
> Human.total Enter ← Human クラスに属する total フィールド
800000000000 ←値が参照できている
> h1.total Enter ←インスタンス h1 に total フィールドがあるか確認
undefined ←インスタンスには total フィールドはない
```

静的メソッド abst はインスタンスに対しては実行できず、Human クラスに対して実行されることを次に示す。

例. abst メソッドの実行（先の例の続き）

```
> Human.abst() Enter ← Human クラスに対して abst メソッドを実行
'mammal' ←実行できている
> h1.abst() Enter ←インスタンス h1 に対して abst メソッドの実行を試みる
Uncaught TypeError: h1.abst is not a function ←実行できない（エラー）
```

3.12.5.4 クラスの継承

class 宣言によって既に定義されているクラス（スーパークラスあるいは**基底クラス**と呼ぶ）の機能を引き継いだ（継承した）別のクラス（**サブクラス**あるいは**拡張クラス**と呼ぶ）を定義する方法について例を挙げて解説する。

先に例示した Human クラスを継承した American, Japanese クラスを定義する例を示す。まず American クラスの定義を次に示す。

記述例：American クラスの宣言

```
1 class American extends Human {
2     static total = 3.319e8;
3     greet() {
4         console.log("Hello.");
5     }
6 }
```

この宣言により、Human クラスのコンストラクタ、メソッド、パブリックフィールドを全て継承した American クラスが定義される。

3～5 行目の greet メソッドは American クラス独自のものとして定義される。

ただし、2 行目の記述により、total フィールドに関しては Human クラスの定義ではなく American クラス独自の定義が優先される。（定義の**オーバーライド**）2 行目の記述がもしも無ければ total フィールドは Human クラスのものが継承される。

American クラスのインスタンスを生成する例を次に示す。

例. American クラスのインスタンス a1 の生成（先の例の続き）

```
> a1 = new American( "John", 90, 1.87 ) Enter    ←インスタンス a1 の生成
American { kind: 'primate', name: 'John', weight: 90, height: 1.87 }
> a1.bmi() Enter    ← Human クラスから継承した bmi メソッドが使える
25.737081414967538    ←実行結果
> a1.greet() Enter    ← American クラス独自の greet メソッドの実行
Hello.                ←実行結果
undefined              ← greet メソッドの戻り値
```

American クラスの total フィールドは Human クラスのものとは別のものとしてオーバーライドされているが、abst メソッドは Human クラスのものがそのまま継承されている。（次の例）

例. オーバーライドと継承（先の例の続き）

```
> American.total Enter    ← American クラスの total フィールドの参照
331900000              ← American クラス独自の値となっている
> American.abst() Enter    ← abst メソッドは Human クラスのものが継承されている
'mammal'               ← Human クラスの場合と同じ
```

同様の方法で、Japanese, Kyotoite クラスを実装する例を示す。

記述例：Japanese クラスの宣言

```
1 class Japanese extends Human {
2     static total = 1.257e8;
3     greet() {
4         console.log("こんにちは
5         . ");
6     }
7 }
```

この宣言により、Human クラスのコンストラクタ、メソッド、パブリックフィールドを全て継承した Japanese クラスが定義される。

3～5 行目の greet メソッドは Japanese クラス独自のものとして定義される。

ただし、2 行目の記述により、total フィールドに関しては Human クラスの定義ではなく Japanese クラス独自の定義が優先される。（定義の**オーバーライド**）2 行目の記述がもしも無ければ total フィールドは Human クラスのものが継承される。

Japanese クラスのインスタンスを生成する例を次に示す。

例. Japanese クラスのインスタンス j1 の生成 (先の例の続き)

```
> j1 = new Japanese( "日本 太郎", 75, 1.76 ) Enter    ←インスタンス j1 の生成
Japanese { kind: 'primate', name: '日本 太郎', weight: 75, height: 1.76 }
> j1.bmi() Enter    ← Human クラスから継承した bmi メソッドが使える
24.212293388429753    ←実行結果
> j1.greet() Enter    ← Japanese クラス独自の greet メソッドの実行
こんにちは.    ←実行結果
undefined    ← greet メソッドの戻り値
```

例. オーバーライドと継承 (先の例の続き)

```
> Japanese.total Enter    ← Japanese クラスの total フィールドの参照
125700000    ← Japanese クラス独自の値となっている
> Japanese.abst() Enter    ← abst メソッドは Human クラスのものが継承されている
'mammal'    ← Human クラスの場合と同じ
```

Japanese クラスを継承した Kyotoite クラスの定義の例を次に示す.

記述例: Kyotoite クラスの宣言

```
1 class Kyotoite extends Japanese {
2     static total = 2.5e5;
3     omotenashi() {
4         console.log("おいでやす.");
5     }
6 }
```

この宣言により, Japanese クラスのコンストラクタ, メソッド, パブリックフィールドを全て継承した Kyotoite クラスが定義される.

3~5 行目の omotenashi メソッドは Kyotoite クラス独自のものとして定義される.

ただし, 2 行目の記述により, total フィールドに関しては Japanese クラスの定義ではなく Kyotoite クラス独自の定義が優先される. (定義のオーバーライド)

Kyotoite クラスのインスタンスを生成する例を次に示す.

例. Kyotoite クラスのインスタンス k1 の生成 (先の例の続き)

```
> k1 = new Kyotoite( "祇園 彩乃", 58, 1.62 ) Enter    ←インスタンス k1 の生成
Kyotoite { kind: 'primate', name: '祇園 彩乃', weight: 58, height: 1.62 }
> k1.bmi() Enter    ← Japanese クラスから継承した bmi メソッドが使える
22.10028959000152    ←実行結果
> k1.greet() Enter    ← Japanese クラスから継承した greet メソッドが使える
こんにちは.    ←実行結果
undefined    ← greet メソッドの戻り値
> k1.omotenashi() Enter    ← Kyotoite クラス独自の omotenashi メソッドの実行
おいでやす.    ←実行結果
undefined    ← omotenashi メソッドの戻り値
```

3.12.5.5 オブジェクトのクラスを調べる方法

プロトタイプベースの OOP の場合と同様の方法でインスタンスのクラスを調べることができる.

例. インスタンスのクラスを調べる (先の例の続き)

```
> h1.constructor.name Enter    ← h1 のクラスを調べる
'Human'
> a1.constructor.name Enter    ← a1 のクラスを調べる
'American'
> j1.constructor.name Enter    ← j1 のクラスを調べる
'Japanese'
> k1.constructor.name Enter    ← k1 のクラスを調べる
'Kyotoite'
```

3.12.5.6 プロパティのカプセル化

オブジェクトのプロパティは秘匿化して外部からのアクセスを拒むことができる。このことをプロパティの**カプセル化**と言う。プロパティやメソッドをカプセル化するには、それら名称の先頭に「#」を付ける。カプセル化について例を挙げて解説する。

記述例：プロパティのカプセル化

```
1 class MyClass {
2     #Vprv = 100;
3     Vpub = 200;
4     #Mprv() {
5         console.log("プライベートメソッド");
6     }
7     Mpub() {
8         console.log("通常のメソッド");
9         this.#Mprv();
10        console.log(this.#Vprv);
11    }
12 }
```

2行目の #Vprv フィールドと 4~6 行目の #Mprv メソッドはカプセル化（秘匿化）されており、インスタンスの外部からはアクセスできない。ただし、9,10行目のように、このクラス内部ではアクセスできる。

MyClass の定義をした後、インスタンスを生成してプロパティ、メソッドにアクセスするを示す。

例. プロパティのカプセル化の検証

```
> m = new MyClass() [Enter]    ←インスタンスの生成
MyClass { Vpub: 200 }
> m.Vpub [Enter]               ←通常のプロパティを参照
200                            ←参照できている。
> m.#Vprv; [Enter]             ←カプセル化されたプロパティの参照を試みると…
m.#Vprv;                       ←↓エラーとなる
^
Uncaught SyntaxError: Private field '#Vprv' must be declared in an enclosing class
```

例. メソッドのカプセル化の検証

```
> m.Mpub() [Enter]             ←通常のメソッドを実行
通常のメソッド
プライベートメソッド          ← Mpub メソッド内部から #Mprv メソッドを呼び出すことは可能
100                            ← Mpub メソッド内部から #Vprv プロパティにアクセスすることは可能
undefined                      ← Mpub メソッドの戻り値
> m.#Mprv(); [Enter]           ←カプセル化されたメソッドを直接実行することは…
m.#Mprv();                     ←できない（エラーとなる↓）
^
Uncaught SyntaxError: Private field '#Mprv' must be declared in an enclosing class
```

接頭辞「#」を持つインスタンスフィールドを**プライベートインスタンスフィールド**⁶⁶と呼ぶ、また、接頭辞「#」を持つメソッドを**プライベートメソッド**と呼ぶ。

3.12.5.7 class 式

先に解説した class 構文はクラス定義を宣言する文であるが、式としての class 構文（**class 式**）もある。

書き方 (1)： 変数 = class クラス名 { 定義内容 }

書き方 (2)： 変数 = class { 定義内容 }

クラスの定義内容を表す式を「変数」に代入する。

例. class 式 (1)

```
> C = class CExpr { prv = 100; } [Enter]    ← class 式を C に代入
[class CExpr]
> m = new C() [Enter]                 ←インスタンスの生成
CExpr { prv: 100 }                  ←指定したクラス名のインスタンスが得られる
```

⁶⁶ プライベートデータプロパティと呼ぶこともある。

例. class 式 (2)

```
> C2 = class { prv = 100; }     ←無名の class 式を C2 に代入  
[class C2]  
> m2 = new C2()     ←インスタンスの生成  
C2 { prv: 100 }    ← class 式の変数名がクラス名となっている
```

3.13 日付、時刻の扱い

JavaScript には日付と時刻の値を取り扱う **Date** コンストラクタが提供されており、システムの現在時刻を取得する機能やタイムスタンプ情報を扱うための機能が利用できる。

JavaScript では時刻を **UNIX エポック**⁶⁷ を基準とする **UNIX 時間**⁶⁸ として取り扱う。

3.13.1 現在時刻の取得

Date コンストラクタを関数として引数なしで実行することで、現在時刻を表す文字列が得られる。また、new 演算子によってインスタンスを生成（引数なしで）すると **Date オブジェクト** が得られる。

例. 現在時刻を文字列として取得

```
> s = Date() [Enter] ← Date 関数の実行
'Sun Oct 08 2023 13:54:17 GMT+0900 (日本標準時)' ← 現在時刻の文字列
```

次に、Date オブジェクトの形式で現在時刻を取得する例を示す。

例. 現在時刻を Date オブジェクトとして取得

```
> d = new Date() [Enter] ← Date インスタンスの生成
2023-10-08T04:54:17.093Z ← Date オブジェクトを ISO8601 形式で表示
```

この例のように、Date オブジェクトは ISO 8601 形式⁶⁹ で表示される。また、この例は Node.js を OS のターミナルウィンドウで実行したものであり、UTC（協定世界時）のタイムゾーンにおける表示（末尾の Z が UTC を意味する）である。（本書では基本的に Node.js での実行例を示す）Date オブジェクトの表示は JavaScript の実行環境で異なることがあり、Google Chrome ブラウザのコンソールでは

Sun Oct 08 2023 13:54:17 GMT+0900 (日本標準時)

といった形式のローカルのタイムゾーンで表示される。

3.13.2 値の取得と設定

Date オブジェクトから各種の情報（ローカルタイム）を取り出すためのメソッドを表 29 に示す。

表 29: Date オブジェクトから値を取得するメソッド（一部）

メソッド	解 説	メソッド	解 説
getFullYear	西暦年の取得	getHours	「時」の値の取得
getMonth	月-1 の値の取得	getMinutes	「分」の値の取得
getDate	月の日の値の取得	getSeconds	「秒」の値の取得
getDay	曜日のインデックス*の取得	getMilliseconds	「ミリ秒」の値の取得

*日曜日を 0 とするインデックス

例. 各値の取出し（先の例の続き）

```
> d.getFullYear() [Enter]
2023 ← 西暦年
> d.getMonth() [Enter]
9 ← 月-1
> d.getDate() [Enter]
8 ← 月の日
> d.getDay() [Enter]
0 ← 日曜日
```

```
> d.getHours() [Enter]
13 ← 時
> d.getMinutes() [Enter]
54 ← 分
> d.getSeconds() [Enter]
17 ← 秒
> d.getMilliseconds() [Enter]
93 ← ミリ秒
```

Date オブジェクトに各種の情報（ローカルタイム）を設定するためのメソッドを表 30 に示す。

⁶⁷西暦 1970 年 1 月 1 日 0 時 0 分 0 秒の時刻

⁶⁸UNIX エポックからの経過秒数

⁶⁹日付と時刻の表記に関する ISO の国際規格

表 30: Date オブジェクトに値を設定するメソッド (一部)

メソッド	解 説	メソッド	解 説
setFullYear	西暦年の設定	setHours	「時」の値の設定
setMonth	月-1 の値の設定	setMinutes	「分」の値の設定
setDate	月の日の値の設定	setSeconds	「秒」の値の設定
		setMilliseconds	「ミリ秒」の値の設定

表 30 の各関数は、当該 Date オブジェクトの UNIX 時間 (ミリ秒) の値を返す。

例. 各値の設定 (先の例の続き)

```
> d.setFullYear(2021) 
1633668857093
> d.setMonth(6) 
1625720057093
> d.setDate(23) 
1627016057093
> d.setHours(20) 
1627041257093
```

```
> d.setMinutes(0) 
1627038017093
> d.setSeconds(0) 
1627038000093
> d.setMilliseconds(0) 
1627038000000
> d.toLocaleString() 
'2021/7/23 20:00:00' ←得られた日付と時刻
```

3.13.3 データ形式の変換

3.13.3.1 Date オブジェクト→文字列

Date オブジェクトに対して toLocaleString メソッドを実行することで、指定したロケールにおける表現の文字列が得られる。

書き方: Date オブジェクト.toLocaleString(ロケール)

「Date オブジェクト」を指定した「ロケール」の文字列に変換する。「ロケール」を省略すると、計算機環境のロケールが採用される。

例. Date オブジェクトを文字列に変換する

```
> d = new Date()  ← Date インスタンスの生成
2023-10-08T06:29:26.590Z
> s1 = d.toLocaleString('ja-JP')  ← Date オブジェクトを文字列に変換
'2023/10/8 15:29:26' ←得られた文字列
```

得られた文字列にはミリ秒の情報が欠落していることに注意すること。

toLocaleString の第 2 引数には様々なオプションをオブジェクトの形で与えることができる。例えば、指定したタイムゾーンでの時刻表示の文字列を得るには次のようにする。

例. UTC での時刻表示の文字列を取得する (先の例の続き)

```
> d.toLocaleString('ja-JP', { timeZone: 'UTC' })  ←タイムゾーンを指定
'2023/10/8 6:29:26' ←得られた文字列 (UTC)
```

このように、第 2 引数のオブジェクトのプロパティ timeZone にタイムゾーンを与える。この場合、IANA の Time Zone Database⁷⁰ の表記でタイムゾーンを記述する。この例では 'UTC' を与えている。

toLocaleString の第 2 引数に与えるオブジェクトのプロパティにはこの他にも多くのものがある。詳しくは他の文献 (文献 [1] など) を参照のこと。

⁷⁰<https://www.iana.org/time-zones> : 例えば日本のタイムゾーンは 'Asia/Tokyo'。

Date オブジェクトを ISO 8601 形式の文字列に変換するには toISOString メソッドを使用する。

書き方： Date オブジェクト.toISOString()

例. Date オブジェクトを ISO8601 形式の文字列に変換する（先の例の続き）

```
> s2 = d.toISOString()     ← Date オブジェクトを ISO8601 形式の文字列に変換  
'2023-10-08T06:29:26.590Z'    ←得られた文字列
```

得られた文字列にはミリ秒の情報も含まれている。

3.13.3.2 文字列→Date オブジェクト

日付と時刻を表す文字列を Date コンストラクタに与えると、それに対応する Date インスタンスが生成される。

例. 文字列を Date オブジェクトに変換する（先の例の続き）

```
> new Date( s1 )     ←文字列 s1 を Date オブジェクトに変換  
2023-10-08T06:29:26.000Z    ←ミリ秒の情報はない  
  
> new Date( s2 )     ←文字列 s2 を Date オブジェクトに変換  
2023-10-08T06:29:26.590Z    ←ミリ秒の情報が含まれている
```

3.14 正規表現

JavaScript では文字列のパターンを表現する**正規表現**を扱うことができる。正規表現は表 31 の表記で文字の種類を、表 32 の表記で反復を表現する形式である。

表 31: 文字の種類を表す正規表現（一部）

パターン	解説
<code>[c₁c₂c₃...]</code>	特定の文字の集合 $c_1c_2c_3\cdots$ （これらの内のどれかに該当）
<code>[c₁-c₂]</code>	$c_1\sim c_2$ の間に含まれる文字。文字の範囲をハイフン '-' でつなげる。例: <code>[a-d]</code> → 'a', 'b', 'c', 'd' のどれか
<code>[^文字の集合]</code>	「文字の集合」にマッチしないもの
<code>.</code> （ドット）	任意の 1 文字
<code>\d</code>	数字（ <code>[0-9]</code> と同じ）
<code>\D</code>	数字以外
<code>\s</code>	空白文字（タブ、改行文字なども含む）
<code>\S</code>	空白文字以外（ <code>\s</code> でないもの）
<code>\w</code>	アンダースコアを含む半角英数字（ <code>[a-zA-Z0-9_]</code> と同じ）
<code>\W</code>	アンダースコア、半角英数字以外（ <code>\w</code> でないもの）

注意） 日本の通貨記号「¥」は、使用するフォントや端末装置によってはバックスラッシュ「\」として表示される。

表 32: 繰り返しの表記（一部）

表記	解説	表記	解説
<code>+</code>	1 回以上	<code>*</code>	0 回以上
<code>?</code>	0 回かもしくは 1 回	<code>{m,n}</code>	m 回以上 n 回以下（回数が大きい方を優先）
<code>{n}</code>	n 回	<code>{m,}</code>	m 回以上

正規表現を応用して文字列の探索やパターンマッチ、置換処理などを行うことができる。それらの処理を行う方法を示しながら正規表現の具体的な使用方法について解説する。

3.14.1 文字列探索（検索）

文字列に対して `search` メソッドを用いることで、指定したパターンが存在する位置を特定することができる。

書き方： `テキスト.search(パターン)`

「テキスト」の文字列の中に「パターン」が最初に現れる場所（インデックス）を返す。「パターン」は正規表現で記述する。探索が失敗すると -1 を返す。

例. 正規表現のパターンを探索する

```
> txt = '0123abcdEFGHxyz4567'; Enter    ←文字列を用意
'0123abcdEFGHxyz4567'
> txt.search( /[a-z]+/ ); Enter    ←英小文字が 1 つ以上並んでいる場所を探す
4                                ←検出した位置のインデックス
> txt.search( /[A-Z]+/ ); Enter    ←英大文字が 1 つ以上並んでいる場所を探す
8                                ←検出した位置のインデックス
```

特定の文字列を探索することもできる。

例. 特定の文字列を探索する（先の例の続き）

```
> txt.search( /xyz/ ); Enter    ←文字列 "xyz" の場所を探す
12                                ←検出した位置のインデックス
> txt.search( /ijk/ ); Enter    ←文字列 "ijk" の場所を探す試み
-1                                ←見つからなかった
```

この例のように、正規表現はスラッシュで括る。スラッシュで括った正規表現は `RegExp` オブジェクトである。

3.14.1.1 RegExp オブジェクト

正規表現は RegExp オブジェクトとして扱う。

書き方 (1): `new RegExp(正規表現を記述した文字列)` (コンストラクタ)

書き方 (2): `/正規表現の記述/` (リテラル)

参考) RegExp コンストラクタの第 2 引数にはオプションを与えることができる。またリテラル表記の場合も最後のスラッシュの後ろにオプションを記述することができる。オプションに関しては本書では必要に応じて紹介するが、詳しくは他の文献 (文献 [1] など) を参照のこと。

3.14.2 パターンマッチ

パターンマッチは単なる探索処理とは異なり、文字列と正規表現を照合する処理である。具体的には `match` メソッドで実行する。

書き方: `テキスト.match(正規表現)`

「テキスト」の文字列と「正規表現」を照合する。照合結果の情報を持つ配列を返す。照合が失敗すると `null` を返す。

例. 単純なパターンマッチ

```
> txt = '012abc345xyz678'; Enter    ←テキストの用意
'012abc345xyz678'

> r = txt.match( /abc/ ); Enter    ←パターンマッチの処理
[ 'abc', index: 3, input: '012abc345xyz678', groups: undefined ]    ←結果

> txt.match( /ijk/ ); Enter    ←照合が失敗するケース
null                        ←照合失敗
```

この例では「abc」のパターンの照合が成功しており、照合結果の配列が `r` に得られている。照合結果の配列は、

[検出した語, index:照合位置, input:照合対象, ...]

という形式である。

この例では単なる探索処理を実行したように見えるが、以降に示す「部分の抽出」などでパターンマッチの理解を深める。

3.14.2.1 マッチした部分の抽出

パターンマッチの処理によってテキストの部分抽出をすることができる。具体的には、抽出したい部分の正規表現を括弧で括る。次の例は、テキスト中の数字列でない部分 (正規表現の `¥D+`) を抽出する処理である。

例. パターンマッチによる部分抽出 (先の例の続き)

```
> r = txt.match( /¥d+(¥D+)¥d+(¥D+)¥d+/ ); Enter    ←パターンマッチによる部分の抽出
[
  '012abc345xyz678',          ←処理結果
  'abc',
  'xyz',
  index: 0,
  input: '012abc345xyz678',
  groups: undefined
]
```

この例では正規表現の中に 2 箇所の抽出対象の部分の括弧があり、得られた抽出部分が `r` の要素として含まれている。得られた配列の最初の要素は正規表現にマッチしたテキストの部分、それ以降は抽出されたテキストの部分である。(次の例)

例. 抽出結果の取出し (先の例の続き)

```
> r[0] Enter    ←インデックス 0 の要素は
'012abc345xyz678'    ←正規表現にマッチしたテキストの部分

> r[1] Enter    ←以降の要素は
'abc'                ←抽出された部分である

> r[2] Enter
'xyz'
```

3.14.2.2 行頭、行末でのパターンマッチ

正規表現の先頭に「^」を記述することで、テキストの先頭部分のパターンマッチができる。また、正規表現の末尾に「\$」を記述することで、テキストの終端部分のパターンマッチができる。以下に実行例を示す。

例. テキストの先頭部分でのマッチ

```
> txt = "abcd0123"  Enter ←テキストの用意
'abcd0123'

> r = txt.match( /^(¥d+)/ )  Enter ←テキストの先頭部分の数字列にマッチさせる試み
null ←マッチしない

> r = txt.match( /^[a-z]+/ )  Enter ←テキストの先頭部分の英小文字列にマッチさせる
[ 'abcd', 'abcd', index: 0, input: 'abcd0123', groups: undefined ] ←マッチの結果
```

テキスト txt の先頭には数字列はなく、`/(¥d+)/` はマッチしないので null が得られているが、先頭には英小文字列があり、これが `/[a-z]+/` にマッチしている。

次に、テキストの終端部分でのパターンマッチの例を示す。

例. テキストの終端部分でのマッチ（先の例の続き）

```
> r = txt.match( /(¥d+)$/ )  Enter ←テキストの終端部分の数字列にマッチさせる
[ '0123', '0123', index: 4, input: 'abcd0123', groups: undefined ] ←マッチの結果

> r = txt.match( /[a-z]+$/ )  Enter ←テキストの終端部分の英小文字列にマッチさせる試み
null ←マッチしない
```

3.14.2.3 パターンマッチの繰り返し実行

テキスト全体に対してパターンマッチを繰り返すには `matchAll` メソッドを使用する。

書き方: `テキスト.matchAll(/正規表現の記述/g)`

「テキスト」に対して「正規表現の記述」のパターンマッチを繰り返し、マッチした複数の結果をイテレータとして返す。正規表現にはオプション「g」(global) を付ける必要がある。

例. パターンマッチの繰り返し

```
> txt = "012ABC345DEF678"  Enter ←テキストの用意
'012ABC345DEF678'

> r = txt.matchAll( /¥d+/g )  Enter ←数字列のマッチを繰り返し実行
Object [RegExp String Iterator] {} ←イテレータが得られる
```

これは、テキストに対して数字列 `¥d+` のパターンマッチを繰り返す例である。結果として r にイテレータが得られている。この r から処理結果の要素を順次取り出す例を次に示す。

例. パターンマッチの結果を順次取り出す（先の例の続き）

```
> for ( m of r ) {  Enter ←反復処理の記述
  console.log(m);  Enter ←処理結果の要素の出力
}  Enter ←反復処理の記述の終了

[ '012', index: 0, input: '012ABC345DEF678', groups: undefined ] ←1つ目
[ '345', index: 6, input: '012ABC345DEF678', groups: undefined ] ←2つ目
[ '678', index: 12, input: '012ABC345DEF678', groups: undefined ] ←3つ目
undefined ←for 文の実行結果
```

この方法を応用すると、探索や部分抽出の繰り返しの処理が実現できる。

3.14.2.4 複数行のテキストに対するパターンマッチ

エスケープシーケンス「`¥n`」(改行) を含んだテキストは複数の行から構成されたものと見ることができる。(次の例)

例. 複数行から成るテキスト

```
> txt = "1abc\n02de\n003f"; console.log(txt); Enter ←複数行のテキストの出力  
1abc      ← 1 行目  
02de      ← 2 行目  
003f      ← 3 行目  
undefined ← log メソッドの実行結果
```

「\n」で区切られた文字列をそれぞれ別の行と見なしてパターンマッチを繰り返すには、matchAll に与える正規表現にオプション「g」と「m」(multiline)を与える。

例. 複数行に対するパターンマッチ (先の例の続き)

```
> r = txt.matchAll( /^(?d+)([a-z]+)$/gm ) Enter ←オプション g, m を与える  
Object [RegExp String Iterator] {}      ←イテレータが得られる
```

これは、テキストの各行を数字列と英小文字列のフィールドから成るものと見なして、それらデータを行毎に取り出す処理の例である。この例で得られたイテレータから抽出部分を出力する例を次に示す。

例. 抽出部分の出力 (先の例の続き)

```
> for ( m of r ) { Enter ←反復処理の記述  
    n = m[1]; a = m[2]; Enter ←抽出部分の取出し  
    console.log('number:',n,'letter:',a); Enter ←出力  
} Enter ←反復処理の記述の終了  
number: 1 letter: abc      ← 1 件目  
number: 02 letter: de      ← 2 件目  
number: 003 letter: f      ← 3 件目  
undefined                  ← for 文の実行結果
```

3.14.3 置換処理

特定の文字列を別の文字列に置き換える、あるいは、正規表現に合致する文字列を別の文字列に置き換える処理を行うには replace メソッドを用いる。

書き方： テキスト.replace(置換対象, 新しい文字列)

「テキスト」中の最初に見つかった「置換対象」を「新しい文字列」に置き換える。「置換対象」には特定の文字列、あるいは正規表現 (RegExp オブジェクト) が使用できる。この処理においては元の「テキスト」は変化せず、置換処理を施した結果の新たな文字列を返す。

例. 単純な置換処理

```
> txt = "012abc345DEF678GHI" Enter ←テキストの用意  
'012abc345DEF678GHI'  
> txt2 = txt.replace( "abc", "ABC" ) Enter ←"abc"を"ABC"に置き換える  
'012ABC345DEF678GHI'      ←置換結果
```

文字列 "abc" が "ABC" に置き換えられていることがわかる。

置換対象に正規表現を与える例を次に示す。

例. 正規表現を用いた置換処理 (先の例の続き)

```
> txt3 = txt2.replace( /\d+/, "数値" ) Enter ←数字列に該当するものを置き換える  
'数値 ABC345DEF678GHI'      ←置換結果
```

正規表現に該当する最初のものが置換されている。全ての該当箇所を置換するには正規表現にオプション「g」を与える。(次の例)

例. 全ての該当箇所の置換処理 (先の例の続き)

```
> txt3 = txt2.replace( /\d+/g, "数値" ) Enter ←正規表現にオプション「g」を与える  
'数値 ABC 数値 DEF 数値 GHI'      ←全ての該当箇所が置換されている
```

正規表現ではない特定の文字列を別の文字列に置換する際、全ての該当箇所を置換するには replaceAll メソッドを使用する。(次の例)

例. replaceAll による置換処理（先の例の続き）

```
> txt = "catdogcatdotcat"  ←このテキストを  
'catdogcatdotcat'  
  
> txt.replace( "cat", "猫" )  ← replace で置換すると  
'猫 dogcatdotcat' ←最初に見つかるものだけが置換されるが  
  
> txt.replaceAll( "cat", "猫" )  ← replaceAll で置換すると  
'猫 dog 猫 dot 猫' ←全ての該当箇所が置換される
```

3.15 例外処理

JavaScript 処理系では、実行不可能なプログラムを読み込むとエラー（例外：Exception）が発生する。（次の例）

例. 例外が発生するケース

```
> v = undefined [Enter]    ←変数 v に undefined を代入
undefined        ←代入結果
> v.prop1 [Enter]          ←存在しないプロパティを敢えて参照
Uncaught TypeError: Cannot read properties of undefined (reading 'prop1')    ←エラー (例外)
```

これは「未定義」を意味する値 `undefined` のプロパティ `prop1` を参照する試みである。当然ではあるが、`undefined` には `prop1` なるプロパティは存在せず、この試みは実行不可能であり、エラー（例外）が発生する。

実際の Web アプリケーションにおいて例外が発生すると、プログラムの実行はその時点で停止する。このことを異常終了（crash）と表現する。次の例 `ErrorTest01.html` はプログラムの異常終了を示すものである。

記述例：ErrorTest01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>ErrorTest01</title>
6   <script>
7     let v = undefined;
8     console.log("v.prop1 :", v.prop1);
9     console.log(" 処理完了");
10  </script>
11 </head>
12 <body></body>
13 </html>
```

これは先の例と同様の試みを行う Web アプリケーションの例であり、`v.prop1` にアクセスする行で例外が発生する。その時点で JavaScript のプログラムは異常終了（停止）し、次の行の

```
console.log("処理完了");
```

は実行されない。これについては、Web ブラウザのコンソールを開くと先の例と同様のエラーメッセージが出力されているのが確認できる。

プログラムの異常終了は Web アプリケーションとしては致命的であり、適切に例外処理を施してプログラムが停止しないようにしなければならない。例外処理を施すには `try ... catch` の構文を用いる。

《例外処理》

```
try {
  (例外を起こす可能性のある処理)
} catch( エラーオブジェクトを受け取る仮引数 ) {
  (例外発生時の処理)
} finally {
  (try ... catch 構文の終了時に実行する処理)
}
```

`try` ブロックの「例外を起こす可能性のある処理」を実行し、例外が発生すると異常終了せずに `catch` ブロックの「例外発生時の処理」を実行する。この際に、発生した例外に関する情報を保持するエラーオブジェクトが「エラーオブジェクトを受け取る仮引数」に渡される。

`finally` ブロックは例外処理の終了時に実行される。またこの部分は、例外発生の有無に係わらず必ず実行される。

3.15.1 エラーオブジェクト

エラーオブジェクト（Error オブジェクト）は例外発生時に生成され、`catch` の引数に渡される。このオブジェクトの重要なプロパティに `name` と `message` があり、`name` にはエラーの種類（後述）が、`message` にはエラーに関する詳細のメッセージが文字列として保持されている。

先の `ErrorTest01.html` に例外処理を施して異常終了しない形にしたものを次の `ErrorTest02.html` に示す。

記述例：ErrorTest02.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>ErrorTest02</title>
6   <script>
7     let v = undefined;
8     try {
9       console.log("v.prop1 :",v.prop1);
10    } catch( e ) {
11      console.log("v.prop1は参照不可能！");
12      console.log("name      :",e.name);
13      console.log("message   :",e.message);
14    } finally {
15      console.log("例外処理を実施した。");
16    }
17    console.log("処理完了");
18  }</script>
19 </head>
20 <body></body>
21 </html>
```

この例では、v.prop1 を参照する部分を try ブロックで実行する。例外が発生した際には異常終了せずに、例外に関する情報を持ったエラーオブジェクト e が catch ブロックに渡される。catch ブロック内では例外が発生した旨を報告する文字列と e.name, e.message をコンソールに出力する。

例外処理の最後に「例外処理を実施した。」という文字列をコンソールに出力する。

この例では例外処理を施しているのので、例外が発生しても異常終了せずに最後の

console.log("処理完了");
が実行される。

ErrorTest02.html を Web ブラウザで表示すると、コンソールに次のように出力されることが確認できる。

コンソールに出力される内容：

```
v.prop1 は参照不可能！
name      : TypeError
message   : Cannot read properties of undefined (reading 'prop1')
例外処理を実施した。
処理完了
```

3.15.1.1 例外（エラー）の種類

表 33 に重要な例外（エラー）の一覧を示す。

表 33: 例外（エラー）の一覧（一部）

例 外	解 説
TypeError	変数または引数の型が有効でない場合に発生する例外。
ReferenceError	不正な参照から参照先の値を取得した時に発生する例外。
RangeError	数値変数または引数が、その有効範囲外である場合に発生する例外。
InternalError	JavaScript エンジンで内部エラー*が発生した時に発生する例外。
URIError	encodeURIComponent() または decodeURI() に不正な引数が渡された時に発生する例外。
SyntaxError	構文エラー

* "too much recursion" (深すぎる再帰) など。

表 33 に示す例外は、当該例外を表すエラーオブジェクトのクラスであり、それをコンストラクタとしてエラーオブジェクトを作成することができる。

書き方： new 例外 (エラーメッセージ)

「エラーメッセージ」を message プロパティに持つ「例外」のクラスのエラーオブジェクトを生成する。(次の例)

例. TypeError のエラーオブジェクトを作る

```
> e = new TypeError("データ型に関するエラーが発生しました。"); e.name
'TypeError'          ← 例外の種類
> e.message           ← message プロパティの確認
'データ型に関するエラーが発生しました。'
```

e.name, e.message の値が確認できている。

■ エラーの種類に応じた例外処理

例外処理の catch ブロック内で、受け取ったエラーオブジェクトの name 属性に従ってエラーの種類に応じた個別の処理を実装することができる。また、instanceof 演算子でエラーオブジェクトの種類を判定することもできる。(次の例)

例. エラーの種類判定 (先の例の続き)

```
> e instanceof TypeError      ← TypeError か?
true
> e instanceof SyntaxError     ← SyntaxError か?
false
```

各種エラーは Error クラスの派生クラス (拡張クラス) である。(次の例)

例. エラーの種類判定 (先の例の続き)

```
> e instanceof Error          ← Error か?
true
```

エラーオブジェクト e の種類は TypeError であり、更に Error でもあることがわかる。

3.15.2 例外を発生させる方法

throw 文で例外を発生させることができる。

書き方: **throw エラーオブジェクト**

与えた「エラーオブジェクト」の例外を発生させる。

先の例で作成したエラーオブジェクト e の例外を発生させる例を示す。

例. throw で例外を発生させる (先の例の続き)

```
> throw e  
Uncaught TypeError: データ型に関するエラーが発生しました。    ←例外が発生した
(付随するメッセージなど)
```

参考) throw の後に文字列などを与えても例外が発生するが、あまり推奨されない。

【throw の応用例】

JavaScript では 0 による除算はエラーとならず⁷¹ 演算結果が Infinity (もしくは -Infinity) や NaN となる。

例. JavaScript での除算

```
> 3 / 2      ← 正常な除算
1.5
> 3 / 0      ← 0 による除算 (1)
Infinity
> -3 / 0     ← 0 による除算 (2)
-Infinity
> 0 / 0      ← 0 による除算 (3)
NaN
```

他の言語処理系と同様に 0 による除算でエラーを発生させる仕組みを考える。次に示す例は throw を応用して 0 による除算をエラーにする関数 warizan の定義である。

記述例: x / y を算出する関数 warizan (ファイル: errorTest04.js)

```
1 function warizan(x,y) {
2   if ( y==0 ) {
3     let e = new Error("0 による 除 算 !");
4     e.name = "ZeroDivisionError";
5     throw e;
6   } else {
7     return x/y;
```

⁷¹他の言語処理系では 0 による除算はエラーとなる。

```
8 |   }  
9 | }
```

この例の3～5行目で例外を起こしている。この場合の例外は JavaScript 処理系には元来備わっていない "ZeroDivisionError" というものであるが、実際にはこのエラーのクラスは Error である。

JavaScript 処理系でこの関数を定義した後の動作を次に示す。

例. 関数 warizan の実行

```
> warizan(3,2)     ←正常な除算  
1.5  
> warizan(3,0)     ← 0 による除算  
Uncaught ZeroDivisionError: 0 による除算！    ←エラーとなる  
    (付随するメッセージなど)
```

0 による除算でエラーが発生している様子がわかる。

3.16 その他の便利な機能

3.16.1 forEach による反復処理

各種データ構造には、反復処理のための `forEach` メソッドが使用できる。

書き方： `データ構造.forEach(関数)`

「データ構造」から要素を1つずつ取り出して「関数」を実行する。戻り値はない (`undefined`)。「データ構造」には配列をはじめとするいくつかのクラスのものを与えることができる。「関数」には関数名や関数式を与えることができる。

以下に例を示して解説する。

次のような、引数を1つ取ってそれを出力する関数 `f` を考える。

例. 関数 `f` の定義

```
> function f(x) { console.log(x); } Enter    ←関数 f の定義
undefined                        ←上記処理の結果
> f("データ") Enter    ←関数 f の評価
データ                            ←与えた引数が出力された
undefined                        ←関数 f の戻り値
```

次に、関数 `f` を様々なクラスのデータ構造に対して実行する例を示す。

例. 配列に対する `forEach` メソッド (先の例の続き)

```
> a = ["x","y","z"] Enter    ←配列の作成
[ 'x', 'y', 'z' ]
> a.forEach( f ) Enter    ←配列の各要素に f を実行する
x
y
z
undefined                ← forEach の実行結果
```

通常のオブジェクトに対しては `forEach` は実行できない。(次の例)

例. オブジェクトに対する `forEach` メソッド実行の試み (先の例の続き)

```
> obj = { "x":1, "y":2, "z":3 } Enter    ←オブジェクトの作成
{ x: 1, y: 2, z: 3 }
> obj.forEach( f ) Enter    ← forEach を実行しようとする…
Uncaught TypeError: obj.forEach is not a function    ←エラーとなる
```

ただし、`Object.keys` や `Object.values` などで作成した配列には `forEach` を実行することができる。

例. オブジェクトのキーの配列に対して `forEach` を実行する (先の例の続き)

```
> k = Object.keys(obj) Enter    ←オブジェクトのキー配列の取得
[ 'x', 'y', 'z' ]
> k.forEach( f ) Enter    ←キー配列の各要素に f を実行する
x
y
z
undefined                ← forEach の実行結果
```

Set オブジェクトに対する `forEach` メソッドの実行例を示す。

例. Set オブジェクトに対する `forEach` メソッド (先の例の続き)

```
> s = new Set(["x","y","z"]) Enter    ← Set の作成
Set(3) { 'x', 'y', 'z' }
> s.forEach( f ) Enter    ← Set の各要素に f を実行する
x
y
z
undefined                ← forEach の実行結果
```

Map オブジェクトに対する forEach メソッドの実行例を示す。

例. Map オブジェクトに対する forEach メソッド (先の例の続き)

```
> m = new Map([["x",1],["y",2],["z",3]]) Enter    ← Map の作成
Map(3) { 'x' => 1, 'y' => 2, 'z' => 3 }
> m.forEach( f ) Enter    ← Map の各要素に f を実行すると…
1
2      ←キーではなく値の方が f に渡される
3
undefined    ← forEach の実行結果
```

Map の forEach メソッドに与える関数は次のように 3 つの引数を取ることができる。

関数の引数： `function(値, キー, Map) { … }`

このような形の関数を forEach メソッドに与えることでキー、値の両方を受け取ることができる。「Map」には、forEach を実行する対象の Map オブジェクトが渡される。

この形の関数を定義して、Map オブジェクトの forEach メソッドで実行する例を示す。

例. 3 つの引数を持つ関数を forEach メソッドで実行する (先の例の続き)

```
> function f3(v,k,mp) { console.log(k,"=>",v,":",mp); } Enter    ← 3 つの引数を取る関数 f3
undefined
> m.forEach( f3 ) Enter    ← Map の各要素に f3 を実行する
x => 1 : Map(3) { 'x' => 1, 'y' => 2, 'z' => 3 }
y => 2 : Map(3) { 'x' => 1, 'y' => 2, 'z' => 3 }    ←キー、値、元の Map が得られている
z => 3 : Map(3) { 'x' => 1, 'y' => 2, 'z' => 3 }
undefined    ← forEach の実行結果
```

ここで示したデータ構造以外でも forEach メソッドが使えるクラスは多く、特に DOM 関連のクラスのインスタンスに forEach メソッドを用いることで HTML 文書の操作を効率的に行うことができる。

3.16.2 配列の全要素に対する一斉処理

配列に対する map メソッドを使用すると、与えられた関数を全ての要素に対して実行した結果を配列として取得することができる。

書き方： `配列.map(関数)`

配列の全要素を順次「関数」の第 1 引数に与えて実行し、得られた値を要素とする配列を返す。

例えば、与えられた値を 2 倍する次のような関数 dbl を考える。

例. 値を 2 倍する関数 dbl の定義

```
> function dbl(n) { return 2*n; } Enter    ←関数の定義
undefined    ←定義処理の完了
> dbl(3) Enter    ← 3 を 2 倍する処理
6      ←結果
```

次に、配列に対して map メソッドを用いてこの関数を一斉に実行する処理を示す。

例. map による一斉処理

```
> [0,1,2,3].map( dbl ) Enter    ←配列の全ての要素に対して dbl を実行する
[ 0, 2, 4, 6 ]      ←結果
```

3.16.2.1 対象要素のインデックスを取得する方法

map メソッドの引数に与える関数として 2 つの引数を取るものを与えると、第 1 引数に配列の要素が、第 2 引数にその要素のインデックスが渡される。(次の例)

例. 2つの引数を取る関数を map に与える

```
> function sqr(x,i) { return [i,x**2]; } Enter ← 2 引数の関数の定義
undefined ← 定義処理の完了
> [2,3,4].map( sqr ) Enter ← 配列の全ての要素に対して sqr を実行する
[ [ 0, 4 ], [ 1, 9 ], [ 2, 16 ] ] ← 結果
```

この例における関数 sqr は,

[第2引数の値, 第1引数の値の2乗]

を返す. 従って, sqr を map メソッドに与えて実行すると, それらの配列が返される.

3.16.3 配列に対する二項演算の連鎖的実行

reduce, reduceRight を使用すると配列の要素に対して順番に二項演算を施した結果を得ることができる.

書き方: 配列.reduce(関数)

「配列」の先頭2つの要素を「関数」の第1, 2引数に与えて実行し, その結果と第3要素を関数に与えて実行するという過程を最終の要素に至るまで繰り返す. reduce は最終的に得られた値を返す. reduceRight メソッドは, 同様の処理を配列の末尾の要素から先頭にかけて実行する.

例えば, 与えられた2つの値の差を求める関数 diff を考える.

例. 2つの値の差を求める関数 diff の定義

```
> function diff(x,y) { return x-y; } Enter ← 関数の定義
undefined ← 定義処理の完了
> diff(5,2) Enter ← 5 - 2 の計算
3
```

次に, reduce, reduceRight メソッドを用いてこの関数を配列に対して実行する処理を示す.

例. reduce, reduceRight による処理 (先の例の続き)

```
> [10,5,2,1].reduce( diff ) Enter ← ((10 - 5) - 2) - 1 の処理
2 ← 結果
> [10,5,2,1].reduceRight( diff ) Enter ← ((1 - 2) - 5) - 10 の処理
-16 ← 結果
```


4 Webアプリケーション開発のための基礎

ここでは、JavaScript を用いて Web アプリケーションを開発するために必要となる基本的な事柄に関して解説する。

4.1 HTML と JavaScript

Web アプリケーションは全体としては HTML 文書として構築し、その中に JavaScript で記述したプログラムを含める。具体的には HTML の script 要素として JavaScript のプログラムを含める。

書き方 (1): `<script>JavaScript のプログラム</script>`

書き方 (2): `<script src="JavaScript のプログラムの URL"></script>`

HTML 文書とは別に JavaScript のプログラムを用意する場合は (2) の書き方に従う。この場合、JavaScript プログラムは HTML 文書と同じエンコーディングで作成する必要がある。HTML 文書と異なるエンコーディングで作成した JavaScript プログラムを読み込む際は、script の開始タグに次のような charset 属性を与える。

`charset="JavaScript プログラムのエンコーディング"`

上記 (1), (2) それぞれの書き方で作成した Web アプリケーションの例を次に示す。

記述例: JShtml00.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>JS実行の例 </title>
6   <script>
7     alert("「OK」をクリックしてください。");
8   </script>
9 </head>
10 <body></body>
11 </html>
```

このページの内容

「OK」をクリックしてください。

OK

右の HTML 文書を Web ブラウザ (Google Chrome) で表示するとこのようなダイアログが表示される。

これは JavaScript プログラムを含んだ HTML 文書の例であり、5~7行目がアラートダイアログを表示するプログラムを含んだ script 要素である。script 要素は head 要素内に限らず、body 要素内の任意の箇所に必要な数だけ配置することができる。

上記のものと同じ働きをするものを、HTML 文書と JavaScript プログラムを別々のファイルの形式で実現したものを次に示す。

記述例: JShtml00-1.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>JS実行の例 </title>
6   <script src="JShtml00-1.js">
7   </script>
8 </head>
9 <body></body>
10 </html>
```

記述例: JShtml00-1.js

```
1 alert("「OK」をクリックしてください。");
```

右の HTML 文書に上の JavaScript プログラムを script 要素で組み込んでいる。2つのファイル JShtml00-1.html と JShtml00-1.js は同じディレクトリにあるものとする。

JShtml00.html, JShtml00-1.js で使用した alert メソッドは、Web ブラウザにおける JavaScript のグローバルオブジェクト window に対して実行するもので、正確には

`window.alert("「OK」をクリックしてください。");`

と記述する。window オブジェクトに関しては後で詳しく解説する。

4.2 DOM に基づいたプログラミング

DOM (Document Object Model) とは、HTML や XML の文書を各種のオブジェクトの木構造（階層構造）で表現したモデルである。Web ブラウザに搭載された JavaScript 言語処理系には HTML 文書を DOM として扱うための API が提供されており、JavaScript のプログラムで HTML 文書の取り扱い（編集など）ができる。

4.2.1 DOM の最上位のオブジェクト

Web ブラウザのウィンドウを表すオブジェクトとして window があり、DOM の全てのオブジェクトはこの window 配下に存在する。window はグローバルオブジェクトであり、JavaScript のプログラムの任意の場所でアクセスできる。window に対するメソッドを実行する際に、基本的には「window. メソッド」と記述するが、この時「window」の記述を省略することもできる。例えば、先に示した例の中に

```
alert("「OK」をクリックしてください。");
```

というプログラムの部分があったが、これは

```
window.alert("「OK」をクリックしてください。");
```

と記述したのと同じである。

window オブジェクトは **Window クラス** のインスタンスである。

4.2.1.1 グローバル変数の所在

Web ブラウザの JavaScript においては、**グローバル変数**（大域変数）は window オブジェクトのプロパティである。このことを Web ブラウザのコンソールで確認する（次の例）

例. グローバル変数の所在確認 1（コンソールでの実行）

```
> window.gv = 3 [Enter]    ← window オブジェクトのプロパティ gv に値を設定
3
> gv [Enter]               ← グローバル変数として値を確認
3                           ← 参照できている
```

これは var 宣言でグローバル変数を作成したのと同じである。（次の例）

例. グローバル変数の所在確認 2（コンソールでの実行）

```
> var gv2 = 4 [Enter]      ← グローバル変数 gv2 を作成
undefined                 ← var 宣言の処理結果
> window.gv2 [Enter]      ← window オブジェクトのプロパティ gv2 を確認
4                           ← 値が得られている
```

4.2.2 HTML の最上位のオブジェクト

HTML 文書自体を表すオブジェクトとして document があり、HTML の全ての要素はこの document 配下のオブジェクトとして存在する。document はグローバルオブジェクトであり、JavaScript のプログラムの任意の場所でアクセスできる。document は window オブジェクトの直下に属しており、正確には

```
window.document
```

と記述する。

document オブジェクトは **HTMLDocument クラス** のインスタンスである。

参考)

HTML ドキュメント (HTMLDocument) や XML ドキュメント (XMLDocument) の上位のクラスとして Document クラスがある。

4.2.3 HTML 要素のクラス

DOM における HTML 要素は、その種類を意味する OOP のクラスのインスタンスである。全ての HTML 要素のクラスは HTMLElement クラスのサブクラス（拡張クラス）であり、「HTML 要素の種類 Element」というクラス名を

持つ。ただし、HTML 要素のインスタンスを生成するには、これらのコンストラクタと new 演算子を用いるのではなく、document に対する createElement メソッドを使用する。

書き方： document.createElement(タグの名前)

「タグの名前」を文字列として与えると、HTML 要素のインスタンスを生成して返す。これに関することは、後の「4.2.7 HTML 要素の生成」(p.127) で詳しく解説する。

注)

システムが提供する既存の HTML 関連のクラスを拡張したクラスをプログラマが独自に定義して使用する場合は、その独自のクラスのコンストラクトと new 演算子によってインスタンスを生成することができる。

4.2.4 HTML の木構造（階層構造）

DOM においては document オブジェクトを頂点とする木構造（階層構造）で HTML が表現されており、DOM 内における HTML 要素を**要素ノード**と呼ぶ。また、頂点となる document は**ドキュメントノード**と呼ぶ。

DOM では HTML 要素内の属性やテキストを扱うことができ、それぞれ**属性ノード**、**テキストノード**と呼ぶ。

HTML の head 要素、body 要素はそれぞれ

document.head	(HTMLHeadElement クラスのオブジェクト)
document.body	(HTMLBodyElement クラスのオブジェクト)

として参照できる。更に、これらの要素の子要素として各種の HTML 要素が階層的に配置されており、要素ノードは親子関係を構成している。

4.2.4.1 子要素、親要素を取得する方法

DOM の中のある要素ノードの配下にあるノード（子ノード）は、children プロパティから参照できる。

書き方： 要素ノード.children

「要素ノード」の全ての子ノード⁷²を持つ HTMLCollection オブジェクトが得られる。HTMLCollection オブジェクトは HTML の要素ノードを複数持つデータ構造であり、イテラブルである。

HTMLCollection オブジェクトが持つ要素数は length プロパティから参照できる。また、保持する各要素は item メソッドで参照することができる。

書き方： HTMLCollection オブジェクト.length

書き方： HTMLCollection オブジェクト.item(インデックス)

item メソッドは「インデックス」が示す要素を返す。

次に示す DOMtest01.html を用いて DOM の扱いを例示する。

記述例：DOMtest01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>DOMtest01</title>
6 </head>
7 <body>
8   <h1 id="hd1">DOM階層のサンプル </h1>
9   <p id="p1">テキスト 1 </p>
10  <div id="d1">
11    <div id="d2">テキスト 2 </div>
12  </div>
13 </body>
14 </html>
```

DOM階層のサンプル

テキスト1
テキスト2

右の HTML 文書を Web ブラウザで表示したところ。
この HTML を基に DOM について解説する。

DOMtest01.html を Web ブラウザ⁷³ で表示して、開発者ツール（デベロッパーツール）のコンソールから JavaScript の実行を試みる。

⁷²テキストノードやコメントは含まない。それらを含めて取得するには childNodes プロパティ（p.126 で解説）を参照する。

⁷³Google Chrome ブラウザを推奨する。

例. head 要素の子要素を取得する

```
> cn = document.head.children  ← head 要素の子要素を取得
HTMLCollection(2) [meta, title] ← cn に HTMLCollection が得られた

> cn.length  ← 要素の個数を調べる
2 ← 2 個
```

例. HTMLCollection オブジェクトの要素を参照する (先の例の続き)

```
> cn0 = cn.item(0)  ← インデックス 0 の要素は
<meta charset="utf-8"> ← HTML の meta 要素

> cn1 = cn.item(1)  ← インデックス 1 の要素は
<title>DOMtest01</title> ← HTML の title 要素
```

head 要素配下の子ノードが得られていることがわかる。この例のように、Web ブラウザの開発者ツールでの HTML 要素の表示は「<title>…」のように見やすく表示されているが、実際には HTMLElement のサブクラスのオブジェクトである。(次の例)

例. 得られた子ノードのクラスを調べる (先の例の続き)

```
> cn0.constructor.name  ← cn0 のクラスを調べる
'HTMLMetaElement'

> cn1.constructor.name  ← cn1 のクラスを調べる
'HTMLTitleElement'
```

上に示した方法で得られた要素ノードは、それが所属する直上の要素ノード (親ノード) と、それが従える子ノードとの連結関係を保持している。従って、得られた要素ノードから更に再帰的に children プロパティを参照して次々と子孫のノードを参照することができる。

ある要素ノードが所属する親ノードは、その要素ノードの parentNode プロパティから参照できる。

書き方: 要素ノード.parentNode

例. 親ノードを調べる (先の例の続き)

```
> cn0.parentNode  ← cn0 の親ノードを調べる
<head>…</head> ← head 要素

> cn1.parentNode  ← cn1 の親ノードを調べる
<head>…</head> ← 上と同じ head 要素
```

4.2.4.2 document 直下の子要素

<html>…</html> で表される html 要素は document の子要素である。

例. <html>…</html> の在り処の確認 (先の例の続き)

```
> c = document.children  ← document 直下の子ノードを取得
HTMLCollection [html] ← 最上位の html 要素を持つ

> c.item(0)  ← 子ノード群の先頭要素 (html 要素) を参照
<html lang="ja"> ← HTML 文書
  <head>…</head>
  <body>…</body>
</html>
```

4.2.4.3 childNodes プロパティ

先に解説した children プロパティはテキストやコメントを含まない。それらを含んだ子ノードは childNodes から取得できる。

書き方: ノード.childNodes

このプロパティから得られるのは NodeList クラスのオブジェクトである。

例. children と childNodes の違い (先の例の続き)

```
> p1.children Enter    ← id="p1" の要素の children は  
HTMLCollection []    ←空であるが  
  
> nd = p1.childNodes Enter    ← childNodes は  
NodeList [text]      ←要素を 1 つ (テキストノード) 持つ  
  
> nd[0].data Enter    ←テキストノードのデータを確認  
'テキスト 1'          ←文字列で得られた
```

children プロパティからテキストノード⁷⁴ を参照することはできないが, childNodes からは参照できている.

4.2.5 HTML 要素の id 属性から要素ノードを取得する方法

HTML の id 属性の値はそのまま, それが示す HTML 要素を表す JavaScript の グローバルオブジェクト として扱うことができる. (次の例)

例. id 属性の値をオブジェクトとして参照する (先の例の続き)

```
> d1 Enter    ← d1 を参照する  
<div id="d1">...</div>    ←要素ノード  
  
> d2 Enter    ← d2 を参照する  
<div id="d2">テキスト 2</div>    ←要素ノード
```

このこととは別に, document オブジェクトに対する getElementById メソッドを使用することで, id 属性の値から要素ノードを取得することができる.

書き方: document.getElementById(id 属性値)

「id 属性値」を文字列として与えると, それに対する要素ノードを返す.

例. getElementById メソッドによる要素ノードの取得 (先の例の続き)

```
> document.getElementById("d1") Enter    ← id="d1" のノードを取得する  
<div id="d1">...</div>    ←要素ノード
```

この方法で要素ノードを取得し, let や const. var など宣言された変数に代入すると, スコープを限定した形で要素ノードを取り扱うことができる.

4.2.6 HTML 要素の要素名 (タグ名) の取得

HTML の要素オブジェクト (要素ノード) の要素名 (タグ名) は nodeName プロパティから参照できる.

書き方: 要素ノード.nodeName

「要素ノード」の要素名 (タグ名) が得られる.

例. 要素名 (タグ名) と id 属性の参照 (先の例の続き)

```
> hd1.nodeName Enter    ← hd1 の要素名 (タグ名) を参照する  
'H1'          ←要素名 (タグ名) が得られている
```

4.2.7 HTML 要素の生成

先の「4.2.3 HTML 要素のクラス」(p.124) で解説したとおり, createElement メソッドで新たな HTML 要素のオブジェクトを生成することができる.

例. 新たな p 要素の生成 (先の例の続き)

```
> ne = document.createElement("p") Enter    ← p 要素を新たに生成  
<p></p>        ←得られた p 要素
```

この方法で生成された直後の HTML 要素はまだ DOM の中には配置されていない.

⁷⁴ 「4.2.8 テキストノードの作成」(p.129) で解説する.

例. 生成直後の HTML 要素の親と子を確認する（先の例の続き）

```
> ne.children  ←子要素を確認
HTMLCollection [] ←まだ子要素は無い
> ne.parentNode  ←親要素を確認
null ←まだどの親にも属していない
```

4.2.7.1 HTML 要素の属性を設定する方法

先の処理で得られた p 要素はまだ属性もテキストも持っていない。HTML 要素のオブジェクトに属性を与えるには `setAttribute` メソッドを使用する。

書き方: `HTML 要素オブジェクト.setAttribute(属性, 値)`

「HTML 要素オブジェクト」の「属性」に「値」を設定する。「属性」と「値」は文字列として与える。このメソッドの戻り値は無い (undefined)。

例. p 要素に `id="p2"` という属性を与える（先の例の続き）

```
> ne.setAttribute("id","p2")  ← id 属性を与える
undefined
> ne  ←処理後の HTML オブジェクトを確認
<p id="p2"></p> ← id 属性を持っている
```

参考) 特に既存の属性に対してはドット表記で

`HTML 要素オブジェクト.属性名 = 値`

と記述して属性に値を設定することもできる。ただし、その属性が論理属性である場合は論理値 (true か false) を代入すること。またドット表記による論理属性の参照の場合は論理値が得られる。

4.2.7.2 HTML 要素の属性を取得する方法

HTML 要素のオブジェクトの属性を取得するには `getAttribute` メソッドを使用する。

書き方: `HTML 要素オブジェクト.getAttribute(属性)`

「HTML 要素オブジェクト」の「属性」を取得して文字列として返す。「属性」は文字列として与える。

例. p 要素の `id` 属性を取得する（先の例の続き）

```
> ne.getAttribute("id")  ← id 属性を取得する
'p2' ←値
```

更に簡単に、

書き方: `HTML 要素オブジェクト.属性名`

と記述して属性の値を参照することもできる。(次の例)

例. p 要素の `id` 属性を参照する（先の例の続き）

```
> ne.id  ← id 属性を参照する
'p2' ←値
```

4.2.7.3 HTML 要素のデータ属性

HTML 要素には各種のデータを保持するためのデータ属性 (Data Attributes) を設定することができる。これは、HTML 要素にプログラマが自由に値を与えることを可能にするもので、`data-` の接頭辞の後に自由に命名ができる属性である。

例. p 要素に独自のデータ属性 `"data-mydata"` を設定する（先の例の続き）

```
> ne.setAttribute("data-mydata","MyValue")  ←データ属性の設定
undefined ←上記の処理結果
> ne  ← p 要素の確認
<p id="p2" data-mydata="MyValue"></p> ←属性が設定されている
```


4.2.8 テキストノードの作成

HTML 要素のオブジェクトにテキスト要素を与えるには**テキストノードオブジェクト**を作成し、それを子要素 (HTML のテキスト要素) として当該 HTML 要素オブジェクトに与える⁷⁵。

テキストノードは `createTextNode` メソッドを用いて生成する。

書き方： `document.createTextNode(テキスト)`

文字列として与えられた「テキスト」を持つテキストノードを生成して返す。

例. テキストノードの生成 (先の例の続き)

```
> tn = document.createTextNode("テキスト 3")  Enter  ←テキストノードの生成
"テキスト 3"                                ←得られたテキストノード
```

テキストノードは `Text` クラスのインスタンスである。テキストノードが持つテキストデータは `data` プロパティ (もしくは `nodeValue` プロパティ) が文字列の形で保持している。

例. テキストノードが持つテキストデータ (先の例の続き)

```
> tn.data  Enter  ←テキストデータの参照
'テキスト 3'      ←文字列
> tn.nodeValue  Enter  ←この場合も
'テキスト 3'      ←同様
```

テキストノードの `data` プロパティ, `nodeValue` プロパティには新たな文字列を代入して更新することができる。

4.2.9 子ノードの追加

要素ノードに子ノードを追加するには `appendChild` メソッドを使用する。

書き方： `要素ノード.appendChild(子ノード)`

「要素ノード」の最後の子ノードとして「子ノード」を追加する。このメソッドは追加した子ノードを返す。

例. `p` 要素のオブジェクトにテキストノードを追加する (先の例の続き)

```
> ne.appendChild( tn )  Enter  ←子ノードの追加
"テキスト 3"            ←追加した子ノードが戻り値
> ne  Enter  ←処理後の p 要素を確認
<p id="p2" data-mydata="MyValue">テキスト 3</p>  ←テキストを持っている
```

更に同じ方法で, `p` 要素を HTML の `body` 要素に子要素として追加する例を示す。

例. `body` 要素に `p` 要素を追加する (先の例の続き)

```
document.body.appendChild( ne )  Enter  ←子ノードの追加
<p id="p2" data-mydata="MyValue">テキスト 3</p>  ←追加された子ノード
```

この処理の後, 直ちに Web ブラウザに HTML の内容が反映され, 右のような表示となる。

DOM階層のサンプル

テキスト1
テキスト2
テキスト3

4.2.9.1 子ノードの追加位置の指定

既存の子ノードの前に新たな子ノードを挿入するには `insertBefore` メソッドを使用する。

書き方： `親のノード.insertBefore(新規ノード, 既存の子ノード)`

「親ノード」が持つ子ノードの内, 「既存の子ノード」の位置に「新規ノード」を挿入する。以下に例を挙げて新規ノードの挿入の処理を示す。

⁷⁵ここでは, 親ノードに対して子ノードを追加する方法について解説する関係上この方法を挙げるが, 単にテキストコンテンツを HTML ノードに与える場合はもっと現実的で簡便な方法がある。それについては後の「4.2.11 HTML 要素にテキストコンテンツを与える方法」(p.131)で解説する。

例. 新規ノードの作成（先の例の続き）

```
> ne = document.createElement("p")  ← p 要素の生成
<p></p>
> ne.setAttribute("id","p0")  ← id 属性の設定
undefined
> tn = document.createTextNode("テキスト 0 ")  ←テキストノードの生成
"テキスト 0 "
> ne.appendChild( tn )  ← p 要素にテキストノードを与える
"テキスト 0 "
> ne  ← p 要素の確認
<p id="p0">テキスト 0 </p> ←出来上がった p 要素
```

この処理で得られた p 要素を既存の p 要素（id="p1"）の位置に挿入する。（次の例）

例. 新規ノードの挿入（先の例の続き）

```
> ch0ld = document.getElementById("p1")  ←既存の p 要素を取得
<p id="p1">テキスト 1 </p>
> ch0ld.parentNode.insertBefore( ne, ch0ld )  ←その位置に ne の値を挿入
<p id="p0">テキスト 0 </p> ←挿入された p 要素
```

この処理によって、既存の p 要素（id="1"）の位置に新規に作成した p 要素（id="0"）が挿入される。この後、直ちに Web ブラウザに HTML の内容が反映され、右のような表示となる。

DOM階層のサンプル

テキスト0
テキスト1
テキスト2
テキスト3

親ノードを意識せずに新規の子ノードを挿入する before, after メソッドもある。

書き方： 既存の子ノード.before(新規ノード)

書き方： 既存の子ノード.after(新規ノード)

before メソッドは「既存の子ノード」の位置に、after メソッドは直後に「新規ノード」を挿入する。戻り値は無い（undefined）。

次のような 2 つの p 要素 ne05, ne15 を作成し、それらを before, after メソッドで DOM に挿入する例を示す。

例. p 要素を 2 つ作成（先の例の続き）

```
> ne05 = document.createElement("p");
ne05.setAttribute("id","p05");
tn = document.createTextNode("テキスト 0.5");
ne05.appendChild( tn );
```

```
> ne15 = document.createElement("p");
ne15.setAttribute("id","p15");
tn = document.createTextNode("テキスト 1.5");
ne15.appendChild( tn );
```

ne05, ne15 をそれぞれ id="1" の p 要素の前と後に挿入する例を示す。

例. 上で作成した ne05, ne15 を挿入する（先の例の続き）

```
> ch0ld.before( ne05 )  ←前に挿入
undefined
> ch0ld.after( ne15 )  ←後に挿入
undefined
```

この処理によって、既存の p 要素 (id="1") の位置の前と後に ne05, ne15 がそれぞれ挿入される。この後、直ちに Web ブラウザに HTML の内容が反映され、右のような表示となる。

DOM階層のサンプル

テキスト0
テキスト0.5
テキスト1
テキスト1.5
テキスト2
テキスト3

4.2.10 子ノードの削除

要素ノードから子ノードを削除するには `removeChild` メソッドを使用する。

書き方： `要素ノード.removeChild(子ノード)`

「要素ノード」から「子ノード」を取り除き、それを返す。

更に簡単な `remove` メソッドもある。

書き方： `要素ノード.remove ()`

「要素ノード」を DOM から取り除く。戻り値は無い (undefined)。

例。 DOM から ne05, ne15 を取り除く (先の例の続き)

```
> document.body.removeChild( ne05 )  ← ne05 を取り除く
<p id="p05">テキスト 0.5</p>
> ne15.remove()  ← ne15 を取り除く
undefined
```

この処理の後、直ちに Web ブラウザに HTML の内容が反映され、右のような表示となる。

DOM階層のサンプル

テキスト0
テキスト1
テキスト2
テキスト3

4.2.11 HTML 要素にテキストコンテンツを与える方法

HTML 要素の `textContent` 属性にテキストを与えることで当該 HTML 要素にテキストコンテンツを与えることができる。またこの属性からテキストコンテンツを参照することもできる。

書き方： `要素ノード.textContent = テキスト`

例。 p 要素にテキストコンテンツを与える

```
> np = document.createElement("p")  ← p 要素の作成
<p></p>
> np.textContent = "サンプルテキスト"  ←それに対してテキストを与える
'サンプルテキスト'
> np  ←確認
<p>サンプルテキスト</p> ←テキストが設定されている
> np.textContent  ←テキストコンテンツを参照
'サンプルテキスト' ←値が得られている
```

作成したテキストノードを別の HTML 要素の子要素として追加する方法について先に解説したが、特にテキストコンテンツの追加処理に関しては、この方法の方が簡便である。

4.2.12 HTML 要素の class 属性から要素ノードを取得する方法

HTML 要素の `class` 属性によって複数の HTML 要素をグループ化することができる。指定した `class` 属性を持つ HTML 要素のオブジェクトをまとめて取得するには `document` の `getElementsByClassName` メソッドを用いる。

書き方： `document.getElementsByClassName(class 名)`

「class 名」の class 属性の値を持つ HTML 要素をまとめて取得し、HTMLCollection オブジェクトとして返す。これに関して次の DOMtest02.html を例に上げて解説する。

記述例：DOMtest02.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>DOMtest02</title>
6 </head>
7 <body>
8   <p id="tx1" class="tx">テキスト1</p>
9   <p id="tx2" class="tx">テキスト2</p>
10  <p id="tx3" class="tx">テキスト3</p>
11 </body>
12 </html>
```

左の HTML 文書を Web ブラウザで表示すると次のようになる。

テキスト1

テキスト2

テキスト3

この状態で Web ブラウザのコンソール上で `getElementsByClassName` メソッドを実行する例を示す。(Google Chrome ブラウザでの実行例)

例. `getElementsByClassName` メソッドの実行

```
> hc = document.getElementsByClassName("tx") Enter ↓得られた要素群
HTMLCollection(3) [p#tx1.tx,p#tx2.tx,p#tx3.tx, tx1:p#tx1.tx, tx2:p#tx2.tx, tx3:p#tx3.tx]
> hc.length Enter ←得られた HTML 要素の個数を調べる
3
```

ここで得られた HTML 要素を確認する。(次の例)

例. 個々の HTML 要素を確認する (先の例の続き)

```
> hc[0] Enter
<p id="tx1" class="tx">テキスト 1</p>
> hc[1] Enter
<p id="tx2" class="tx">テキスト 2</p>
> hc[2] Enter
<p id="tx3" class="tx">テキスト 3</p>
```

注) HTMLCollection オブジェクトには `forEach` メソッドは使えない。(次の例)

例. `forEach` メソッドを使用する試み (先の例の続き)

```
> hc.forEach( e => console.log(e) ) Enter ← HTMLCollection に forEach を試みると…
Uncaught TypeError: hc.forEach is not a function ←エラーとなる
    at <anonymous>:1:4
```

HTMLCollection オブジェクトを配列に変換すると `forEach` メソッドが使える。

例. 配列に変換して `forEach` を実行する (先の例の続き)

```
Array.from(hc).forEach( e => console.log(e) ) Enter
<p id="tx1" class="tx">テキスト 1</p>
<p id="tx2" class="tx">テキスト 2</p>
<p id="tx3" class="tx">テキスト 3</p>
undefined
```

スプレッド構文を使用して同様の処理をすることもできる。

例. スプレッド構文で配列に変換 (先の例の続き)

```
[...hc].forEach( e => console.log(e) ) Enter
<p id="tx1" class="tx">テキスト 1</p>
<p id="tx2" class="tx">テキスト 2</p>
<p id="tx3" class="tx">テキスト 3</p>
undefined
```

参考)

document.getElementsByClassName によって得られた HTMLCollection オブジェクトが保持する要素は元の HTML 文書の要素の参照であり、文書中の HTML 要素が動的に変更されるとそれが直ちに HTMLCollection オブジェクトの要素にも反映される。

4.2.13 DOMParser

DOMParser インターフェースは、文字列として記述された HTML (XML) を解析して Document クラスのオブジェクトとして解釈するための機能を提供する。DOMParser のインスタンスは HTML (XML) のパーサ (解析器) となる。

書き方: new DOMParser()

パーサ (解析器) を生成して返す。

例. パーサの生成

```
> p = new DOMParser() Enter    ←パーサの生成
DOMParser { }          ←得られたパーサ
```

このようにして得られたパーサに対して parseFromString メソッドを実行することで文字列を Document クラスのオブジェクトに変換することができる。

書き方: パーサ.parseFromString(文字列, メディアタイプ)

「パーサ」を使用して「文字列」を「メディアタイプ」で識別し、Document オブジェクトに変換したものを返す。例えばメディアタイプとして "text/html" を与えると HTMLDocument クラスのオブジェクトを返す。

例. 文字列を HTMLDocument に変換する (先の例の続き)

```
> s = "<h2>文字列</h2>" Enter    ←文字列表現の HTML
"<h2>文字列</h2>"

> d = p.parseFromString( s, "text/html" ) Enter    ←上記文字列を HTMLDocument に変換
HTMLDocument{ ... }
```

この例で得られた d は <html> の要素を含む HTMLDocument オブジェクトである。

例. 得られた HTMLDocument を調べる (先の例の続き)

```
> d.childNodes Enter    ←子ノードを調べる
NodeList [ html ]    ←<html> を含んでいる

> d.childNodes[0] Enter    ←先頭の子ノードを調べる
<html>

> d.body.childNodes[0] Enter    ←更に内部を調べる
<h2>                  ←<h2> 要素を持っていることがわかる
```

重要)

parseFromString メソッドによって得られたオブジェクト (上記の例における d) は、window.document と同じ HTMLDocument クラスのオブジェクトである。すなわち、document とは別のインスタンスであるが、document に対して行う各種の処理が同様に適用できる。

4.2.14 CSS のセレクタから HTML 要素を取得する方法

document のメソッド `querySelector`, `querySelectorAll` を使用することで, CSS のセレクタに対応する HTML 要素を取得することができる.

書き方: `document.querySelector(セレクタの記述)`

「セレクタの記述」に合致する HTML 要素の内, 最初の要素オブジェクトを返す. 合致するものがない場合は `null` を返す.

先に示した DOMtest02.html を Web ブラウザで表示してコンソールで `querySelector` メソッドを実行する例を示す.

例. `querySelector` メソッドによる HTML 要素の取得 (先の例の続き)

```
> nd = document.querySelector(".tx") Enter    ← class="tx" に合致するものを取得する
<p id="tx1" class="tx">テキスト 1</p>    ←合致する最初のものが得られる
```

「セレクタの記述」に合致する全ての HTML 要素を取得するには `querySelectorAll` を使用する.

書き方: `document.querySelectorAll(セレクタの記述)`

得られた HTML 要素群が `NodeList` オブジェクトとして得られる.

これの実行例を次に示す.

例. `querySelectorAll` メソッドによる HTML 要素の取得 (先の例の続き)

```
> ndList = document.querySelectorAll(".tx") Enter    ← class="tx" に合致するものを取得する
NodeList(3) [p#tx1.tx, p#tx2.tx, p#tx3.tx]    ←合致する全てのものが得られる
```

変数 `ndList` に 3 つの HTML 要素を持つ `NodeList` オブジェクトが得られており, 通常の添字 [] で要素にアクセスができる. また `forEach` メソッドが使える.

例. `NodeList` オブジェクトの要素の参照 (先の例の続き)

```
> ndList[0] Enter
<p id="tx1" class="tx">テキスト 1</p>
> ndList[1] Enter
<p id="tx2" class="tx">テキスト 2</p>
> ndList[2] Enter
<p id="tx3" class="tx">テキスト 3</p>
```

例. `forEach` メソッドの実行 (先の例の続き)

```
> ndList.forEach( e => console.log(e) ) Enter
<p id="tx1" class="tx">テキスト 1</p>
<p id="tx2" class="tx">テキスト 2</p>
<p id="tx3" class="tx">テキスト 3</p>
undefined
```

参考)

`document.querySelectorAll` で得られた `NodeList` の要素は静的であり, DOM の HTML を動的に変更してもそれは `document.querySelectorAll` で得られた要素オブジェクトには反映されない.

4.3 JavaScript で CSS にアクセスする方法

JavaScript から CSS の属性にアクセスするにはいくつかの方法がある。ここでは例を挙げて具体的な方法について解説する。

記述例：CSStest01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>CSStest01</title>
6   <style>
7     #p1 {
8       font-size: 24pt;
9       color: black;
10    }
11  </style>
12 </head>
13 <body>
14   <p id="p1">文字列</p>
15 </body>
16 </html>
```

文字列

左の記述を Web ブラウザで表示した例

Web ブラウザ（Google Chrome を推奨）でこの HTML を表示した状態で開発者ツール（デベロッパーツール）のコンソールを開き、JavaScript のコードを実行する形で解説する。

4.3.1 HTML 要素のスタイルの取得

window オブジェクトに対するメソッド `getComputedStyle` を使用することで、DOM の要素ノードに施されたスタイルを取得することができる。

書き方： `window.getComputedStyle(要素ノード)`

「要素ノード」のスタイルを `CSSStyleDeclaration` オブジェクトとして返す。得られた `CSSStyleDeclaration` から各種 CSS 属性の値を取得するには、通常のオブジェクトのプロパティに対するのと同様の方法を取る。

例. CSStest01.html の id="p1" の要素の色を調べる

```
> h = document.getElementById("p1") Enter    ←要素ノードの取得
<p id="p1">文字列</p>
> cs = window.getComputedStyle( h ); cs["color"] Enter    ← CSS の color 属性の値を調べる
'rgb(0, 0, 0)'                                ←色の値が得られた
```

CSStest01.html の id="p1" の HTML 要素の色は black と記述されているが、その値が rgb 関数表記の形式で得られていることがわかる。

スタイルシートはインラインスタイル、内部スタイルシート、外部スタイルシートといった複数の形態で与えられる。また、HTML 文書を実際に表示するデバイスの仕様などの事情もあることから、`window.getComputedStyle` によって得られる CSS 属性は結果的に有効になったものであり、スタイルシートに記述したものとは別の形式で得られることがある。（次の例）

例. CSStest01.html の id="p1" の要素のフォントサイズを調べる（先の例の続き）

```
> cs["font-size"] Enter    ←フォントサイズの取得
'32px'                ← 32 ピクセル
```

CSStest01.html の中では 24pt（24 ポイント）と記述されているフォントサイズが、表示デバイス上では 32 px（32 ピクセル）であることがわかる。

4.3.2 スタイルの動的な変更

HTML 要素のオブジェクト（要素ノード）のスタイルを動的に変更するには、当該要素ノードの `style` プロパティの設定を変更する。

書き方： 要素ノード.style.CSS 属性 = 値

「要素ノード」の「CSS 属性」に「値」を設定する。以下に例を挙げて解説する。

例. CSStest01.html の id="p1" の要素の色を変更する（先の例の続き）

```
> h.style.color = "red"  ←色の変更  
'red'
```

この処理の後、直ちに Web ブラウザに HTML の内容が反映され、右のような表示となる。

文字列

Web ブラウザで表示した例

HTML の要素ノードの style プロパティの設定の対象となるのはインラインスタイルであり、内部、外部のスタイルシートは変更されない。

4.3.2.1 style プロパティ配下の CSS 属性名に関する注意

CSS 属性の名称はハイフン「-」を含むものがある。例えば CSStest01.html では font-size という CSS 属性名の記述があるが、ハイフンは JavaScript ではマイナス記号としての役割がある。従って、style プロパティ配下のフォントサイズのプロパティに値を設定するには

要素ノード.style.fontSize = "36pt"

などと記述する。すなわち、「font-size」のハイフンを取り除き、続く最初の文字を大文字（キャメルケース）にする。また、CSS 属性の「float」は「cssFloat」と記述する。

例. CSStest01.html の id="p1" の要素のフォントサイズを変更する（先の例の続き）

```
> h.style.fontSize = "36pt"  ←フォントサイズの変更  
'36pt'
```

この処理の後、直ちに Web ブラウザに HTML の内容が反映され、右のような表示となる。

文字列

Web ブラウザで表示した例

参考) 先の処理と同じことを

```
h.style["font-size"] = "36pt"
```

と記述して実行することもでき、この場合は CSS の属性名は通常通りの記述が可能である。

4.4 イベント駆動型プログラミング

Web アプリケーションは **GUI** (Graphical User Interface) を介してユーザと対話するものである。GUI の形式のアプリケーションにおいては、ユーザが行った UI 要素に対する操作を起点として、対応するプログラム (関数など) を起動して処理を実行する。すなわち、UI 要素に対するユーザの操作はシステムに対する**イベント**であり、それを受けて対応する**イベントハンドラ** (イベントによって開始するプログラム) を起動する。このような仕組みを**イベントハンドリング**と呼ぶ。またこのようなスタイルのプログラミングを**イベント駆動型プログラミング**と呼ぶ。

イベント駆動型プログラミングのスタイルで構築した単純な Web アプリケーションの例を Event01.html に示す。

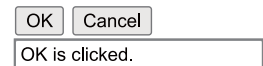
記述例: Event01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>Event01</title>
6   <script>
7     console.log("出力1");
8     console.log("出力2");
9     function f1() {
10      const t1 = document.getElementById("t1");
11      t1.value = "OK is clicked.";
12    }
13    function f2() {
14      const t1 = document.getElementById("t1");
15      t1.value = "";
16    }
17  </script>
18 </head>
19 <body>
20   <input type="button" value="OK" onClick="f1()">
21   <input type="button" value="Cancel" onClick="f2()">
22   <br>
23   <input type="text" id="t1">
24 </body>
25 </html>
```

これを Web ブラウザで表示すると次のような UI が現れる。



この際、7~16 行目の JavaScript プログラムが実行される。7,8 行目のプログラムにより、Web ブラウザのコンソールに「出力 1」「出力 2」というメッセージが表示される。「OK」ボタンをクリックすると次のような表示となる。



また、「Cancel」ボタンをクリックするとテキストフィールド (id="t1") に空文字列が設定されて初期の表示となる。

このシステムでは起動時に 7~16 行目 (<script>...</script>の部分) のプログラムが一通り実行されて関数 f1, f2 が定義され、その後はイベントを待つ状態となる。20,21 行目に「OK」ボタンと「Cancel」ボタンが実装されており、input 要素の開始タグの中に「onClick=」という属性が記述されている。これは、当該ボタンがクリックされた際に起動するプログラムを指定するもので、「OK」ボタンがクリックされると関数 f1 が起動される。同様に「Cancel」ボタンがクリックされると関数 f2 が起動される。この例のように

onイベント = イベントハンドラ

という記述によって、当該 UI 要素に「イベント」が発生した際に「イベントハンドラ」を起動する仕組みが実装される。またこの例では「onClick」というキャメルケースの属性でマウスのクリックのイベント処理を実現しているが「onclick」と記述しても良い。

対象にイベントハンドラを登録する方法は他にもあり、特に後の「4.4.3 addEventListener によるイベントハンドラの登録」(p.139) で解説する方法が推奨される。

4.4.1 イベントハンドリングの仕組み

ユーザからの UI に対する操作を始めとするイベントが発生すると、それらは順番に**イベントキュー**と呼ばれる機構⁷⁶に登録される。また、JavaScript 言語処理系は**イベントループ**と呼ばれる反復処理を常に実行しており、イベントキューに登録されているイベントを順番に取り出して、それに対応するイベントハンドラを起動する処理を繰り返す。(イベントキューが空の場合はイベントの受信を待機する)

⁷⁶FIFO (First In, First Out) の待ち行列。

4.4.2 イベントの種類

ここでは基本的な（使用頻度の高い）イベントを挙げる。

4.4.2.1 ページの読み込み、離脱

当該 Web アプリケーションが Web ブラウザに読み込まれ、それが完了したときに load イベント（表 34）が発生する。

表 34: ページの読み込み、離脱など

イベント	説 明
load	ページの全ての内容の読み込みが完了した時に発生
unload	ウィンドウを閉じた時や他のページに切り替えた時、あるいはページをリロード（更新）した時に発生（ページのリソースをアンロードした時に発生）

4.4.2.2 マウスの操作

マウスの操作の際に発生する各種のイベントの一部を表 35 に示す。

表 35: マウスの操作（代表的なもの）

イベント	説 明
click	対象をクリック（シングルクリック）した時に発生
dblclick	対象をダブルクリックした時に発生
mouseover	対象にマウスポインタが重なった時に発生
mouseout	対象からマウスポインタが離れた時に発生
mousedown	対象でマウスボタンを押下した時に発生
mouseup	対象でマウスボタン押下した後、ボタンを開放した時に発生
mousemove	対象の上でマウスを動かしている時に発生
contextmenu	対象の上でマウスの右ボタンがクリックされた時に発生

4.4.2.3 キーボードの操作

キーボードの操作の際に発生する各種のイベントの一部を表 36 に示す。

表 36: キーボードの操作（代表的なもの）

イベント	説 明
keydown	キーを押した時に発生
keyup	押していたキーを放した時に発生

4.4.2.4 変化や選択によって発生するイベント

対象の状態が変化した際や対象を選択した際に発生する各種のイベントの一部を表 37 に示す。

表 37: 変更・選択（代表的なもの）

イベント	説 明
resize	window のサイズが変更された時に発生
scroll	対象要素にスクロールが起こった時に発生
select	対象要素のテキストが選択された時に発生
focus	対象要素（ページやフォーム要素など）がフォーカスされた時に発生
blur	対象要素（ページやフォーム要素など）のフォーカスが外れた時に発生
change	対象の value 属性の値が変化した時に発生
input	change は value 属性の値の変化が完全に終わった段階で発生するが、input は「変わりつつある」段階で発生する。 (例えば、スライダーをマウスで変更している最中など)

4.4.2.5 その他のイベント

先に挙げたものの以外に表 38 のようなイベントを挙げる。

表 38: その他

イベント	説明
submit	フォームを送信しようとした時に発生
reset	フォームがリセットされた時に発生
abort	画像の読み込みを中断した時に発生
error	画像の読み込み中にエラーが発生した時に発生

4.4.3 addEventListener によるイベントハンドラの登録

先の Event01.html の例で示した方法以外にも addEventListener メソッドでイベントハンドラを対象に登録する方法があり、その方法が推奨される。

書き方： 対象オブジェクト.addEventListener(イベント, イベントハンドラ)

「対象オブジェクト」に「イベント」が発生した際の「イベントハンドラ」を登録する。またこの処理を複数回実行することで、同じ「対象オブジェクト」の同じ「イベント」に異なる「イベントハンドラ」を複数登録することもできる。その場合は「イベント」の発生を受けた場合に登録された複数の「イベントハンドラ」を登録された順で実行する。

addEventListener メソッドを用いる形で Event01.html と同様の機能を実現する Event01-2.html を示す。

記述例：Event01-2.html

```
1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>Event01-2</title>
6    <script>
7      function f1() {
8        const t1 = document.getElementById("t1");
9        t1.value = "OK is clicked.";
10     }
11     function f2() {
12       const t1 = document.getElementById("t1");
13       t1.value = "";
14     }
15     function f3() {
16       console.log("Canceled.");
17     }
18     function f0() {
19       const b1 = document.getElementById("b1");
20       const b2 = document.getElementById("b2");
21       b1.addEventListener("click",f1)
22       b2.addEventListener("click",f2)
23       b2.addEventListener("click",f3)
24     }
25     window.addEventListener("load",f0);
26   </script>
27 </head>
28 <body>
29   <input type="button" value="OK" id="b1">
30   <input type="button" value="Cancel" id="b2">
31   <br>
32   <input type="text" id="t1">
33 </body>
34 </html>
```

これを Web ブラウザで表示すると script 要素（6～26 行目）が実行され、各種のイベントハンドラが関数として定義され、初期化のためのイベントハンドラ f0 が登録（25 行目）される。

「OK」ボタンと「Cancel」ボタンのクリックのためのイベントハンドラ f1, f2, f3 を 7～17 行目で定義している。実際にそれらをボタンのクリックに対するイベントハンドラとして登録するのは f0（18～24 行目）である。この f0 は、コンテンツの読み込みが完了したことを受けて起動されることが 25 行目に記述されている。

このシステムでは「Cancel」ボタンがクリックされた際に f2, f3 の 2 つが順番に起動するように設定（22～23 行目）されている。イベントハンドラ f3 の働きにより、「Cancel」ボタンがクリックされると、Web ブラウザのコンソールに「Canceled.」と出力される。

4.4.4 イベント名のプロパティにイベントハンドラを登録する方法

イベントを受信する対象のプロパティにイベントハンドラを登録する方法もある。

書き方： 対象オブジェクト.**on**イベント名 = イベントハンドラ

「対象オブジェクト」に「イベント名」のイベントが発生した場合に「イベントハンドラ」を起動する。

この方法で Event01.html と同様の機能を実現する Event01-3.html を示す。

記述例：Event01-3.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>Event01-3</title>
6   <script>
7     function f1() {
8       const t1 = document.getElementById("t1");
9       t1.value = "OK is clicked.";
10    }
11    function f2() {
12      const t1 = document.getElementById("t1");
13      t1.value = "";
14    }
15    function f0() {
16      const b1 = document.getElementById("b1");
17      const b2 = document.getElementById("b2");
18      b1.onclick = f1;
19      b2.onclick = f2;
20    }
21    window.onload = f0;
22  </script>
23 </head>
24 <body>
25   <input type="button" value="OK" id="b1">
26   <input type="button" value="Cancel" id="b2">
27   <br>
28   <input type="text" id="t1">
29 </body>
30 </html>
```

これを Web ブラウザで表示すると script 要素（6～22 行目）が実行され、各種のイベントハンドラが関数として定義され、初期化のためのイベントハンドラ f0 が登録（21 行目）される。

「OK」ボタンと「Cancel」ボタンのクリックのためのイベントハンドラ f1, f2 を 7～14 行目で定義している。実際にそれらをボタンのクリックに対するイベントハンドラとして登録するのは f0（15～20 行目）である。この f0 は、コンテンツの読み込みが完了したことを受けて起動されることが 21 行目に記述されている。

このシステムでは、ボタン要素の onclick プロパティにイベントハンドラ f1, f2 を登録している。また、コンテンツ読み込みが完了したことを受けて起動する f0 は window オブジェクトの onload プロパティに設定されている。

この方法では、同じイベントに対して複数のイベントハンドラを登録することができない。従って、addEventListener メソッドによる方法が推奨される。

4.4.5 イベントオブジェクト

JavaScript のイベント処理においては、発生したイベントに関する様々な情報をイベントオブジェクトとして取得することができる。イベントオブジェクトは、当該イベント発生時にイベントハンドラの第 1 引数に渡される。従ってイベントオブジェクトを取得するには、イベントハンドラを定義する際にそのための仮引数を設置する必要がある。

イベントオブジェクトの各種プロパティ（表 39～41）からイベントに関する各種の情報が参照できる。

表 39: イベントオブジェクトの基本的なプロパティ（一部）

プロパティ	解 説
type	イベントの種類を表す文字列
timeStamp	要素が作成されてからイベント発生までの経過時間（ミリ秒）
currentTarget	イベントハンドラを登録した要素への参照
target	イベントが発生した要素への参照

4.4.5.1 イベントオブジェクトの使用例（マウス関連）

マウス操作によって発生したイベントオブジェクトのプロパティの一部を表 40 に挙げる。

表 40: イベントオブジェクトのマウスに関するプロパティ（一部）

プロパティ	解 説
offsetX, offsetY	イベントが発生した要素の左上を原点とする座標 (px)
pageX, pageY	ページ（コンテンツ）の左上を原点とする座標 (px)
clientX, clientY	ブラウザの表示エリアの左上を原点とする座標 (px)
screenX, screenY	ディスプレイの左上を原点とする座標 (px)
altKey	Alt キーが押されている場合 true, それ以外は false

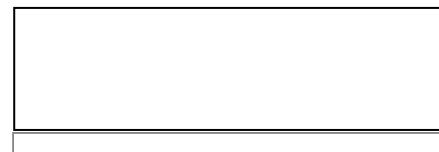
マウス関連のプロパティをイベントオブジェクトから取得する例を Event02.html に示す。

記述例：Event02.html

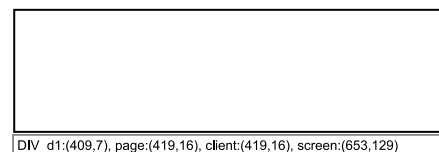
```
1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>Event02</title>
6    <style>
7      #d1 {
8        width: 324pt;
9        height: 90pt;
10       border: solid 2px black;
11     }
12     #tx1 {
13       width: 320pt;
14     }
15   </style>
16   <script>
17     function f1(e) {
18       const tx1 = document.getElementById("tx1");
19       const tg = e.target.nodeName;
20       const id = e.target.id;
21       const x = e.offsetX, y = e.offsetY;
22       const msg1 = tg+"_"+id+":("+x+","+y+")";
23       const px = e.pageX, py = e.pageY;
24       const msg2 = "page:("+px+","+py+")";
25       const cx = e.clientX, cy = e.clientY;
26       const msg3 = "client:("+cx+","+cy+")";
27       const sx = e.screenX, sy = e.screenY;
28       const msg4 = "screen:("+sx+","+sy+")";
29       tx1.value = msg1+msg2+msg3+msg4;
30     }
31     function f0() {
32       const d1 = document.getElementById("d1");
33       d1.addEventListener("mousemove", f1);
34     }
35     window.addEventListener("load", f0);
36   </script>
37 </head>
38 <body>
39   <div id="d1"></div>
40   <input type="text" id="tx1">
41 </body>
42 </html>
```

これを Web ブラウザで表示すると script 要素（16～36 行目）が実行され、各種のイベントハンドラが関数として定義され、初期化のためのイベントハンドラ f0 が登録（35 行目）される。

表示した直後は次のように四角い枠とテキストフィールドが表示される。



この表示の中の四角い枠（div 要素）の上にマウスポインタを重ねると、マウスに関する各種の情報がテキストフィールド（input 要素）に表示される。（次の例）








div 要素はマウスイベント mousemove を受信するとイベントハンドラ f1 を起動する。この際に第 1 引数に当該イベントの情報を保持するイベントオブジェクトを渡す。

その後 f1 はイベントが発生した要素に関する情報（19～20 行目）とマウスポインタの位置に関する情報（21～28 行目）を編纂してテキストフィールド（input 要素）に表示（29 行目）する。

4.4.5.2 イベントオブジェクトの使用例（キーボード関連）

キーボード操作によって発生したイベントオブジェクトのプロパティの一部を表 41 に挙げる。

表 41: イベントオブジェクトのキーボードに関するプロパティ（一部）	
プロパティ	解 説
key	押されたキーの文字列
code	押されたボタンのキーボード上の役割を表現する文字列
altKey	 キーが押されていれば true, それ以外は false
ctrlKey	 キーが押されていれば true, それ以外は false
shiftKey	 キーが押されていれば true, それ以外は false
metaKey	メタキー* が押されていれば true, それ以外は false

*Apple の mac では , Windows では 

キーボード関連のプロパティをイベントオブジェクトから取得する例を Event03.html に示す。

記述例：Event03.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>Event03</title>
6   <style>
7     input { width: 500pt; }
8   </style>
9   <script>
10    function f1(e) {
11      const tx2 = document.getElementById("tx2");
12      const tp = e.type;
13      const tm = Math.round(e.timeStamp);
14      const key = e.key;
15      const code = e.code;
16      const alt = e.altKey;
17      const ctrl = e.ctrlKey;
18      const shift = e.shiftKey;
19      const meta = e.metaKey;
20      const msg = "time:"+tm+", type:"+tp+
21        ", key:"+key+", code:"+code+
22        ", alt:"+alt+", ctrl:"+ctrl+
23        ", shift:"+shift+", meta:"+meta;
24      tx2.value = msg;
25    }
26    function f0() {
27      const tx1 = document.getElementById("tx1");
28      tx1.addEventListener("keydown",f1);
29    }
30    window.addEventListener("load",f0);
31  </script>
32 </head>
33 <body>
34   入力テキスト : <input type="text" id="tx1"><br>
35   イベントの 情報 : <input type="text" id="tx2">
36 </body>
37 </html>
```

これを Web ブラウザで表示すると script 要素（9～31 行目）が実行され、各種のイベントハンドラが関数として定義され、初期化のためのイベントハンドラ f0 が登録（30 行目）される。

表示した直後は図 30 の (a) のようにテキストフィールドが上下に 2 つ表示される。

上側のテキストフィールドを選択してキーボードのボタンを押すと、押したボタンに関する各種の情報が図 30 の (b) のように下側のテキストフィールドに表示される。

上側のテキストフィールドはキーボードイベント keydown を受信するとイベントハンドラ f1 を起動する。この際に第 1 引数に当該イベントの情報を保持するイベントオブジェクトを渡す。その後 f1 はイベントが発生した時刻とイベントの種類に関する情報、押したボタンに関する情報を編纂（20～23 行目）して下側のテキストフィールドに表示（24 行目）する。

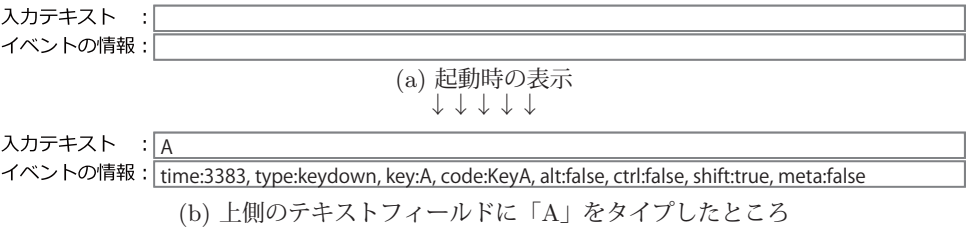


図 30: Event03.html を Web ブラウザで表示して操作した例

4.4.6 タイマー

setTimeout 関数を用いることで、時間の経過に対するイベントハンドリングができる。

書き方： `setTimeout(関数, 経過時間, 引数並び)`

setTimeout が実行された後「経過時間」が経過したことを意味するイベントをイベントキューに登録する。これによりタイマーが経過時間を監視し、経過時間満了のイベントを受けて「関数」に「引数並び」を与えて起動する。setTimeout はタイマー ID (timeoutID) を意味する正の整数値を返す⁷⁷。

setTimeout で登録されたタイマーは clearTimeout で取り消すことができる。

書き方： `clearTimeout(タイマー ID)`

「タイマー ID」が示すタイマーを取り消す。戻り値はない (undefined)。

例. 3 秒 (3,000ms) 後に "test" をコンソールに表示する (Google Chrome のコンソールでの実行)

```
> tid = setTimeout( console.log, 3000, "test" ) Enter    ←タイマーの登録
4          ← timeoutID を意味する整数値
test       ←上記の処理から 3 秒 (3,000ms) 後に表示される
```

例. 3 秒 (3,000ms) 後に "test" をコンソールに表示する (コマンドウィンドウ下の Node.js での実行)

```
> tid = setTimeout( console.log, 3000, "test" ) Enter    ←タイマーの登録
Timeout {                                     ← Timeout オブジェクト
  _idleTimeout: 3000,
  _idlePrev: [TimersList],
  _idleNext: [TimersList],
    (途中省略)
  [Symbol(asyncId)]: 246,
  [Symbol(triggerId)]: 6
}
> test    ←上記の処理から 3 秒 (3,000ms) 後に表示される
```

設定された経過時間の後で console.log が実行されていることがわかる。

例. 投入されたタイマーの取り消し

```
> tid = setTimeout( console.log, 3000, "test" ); clearTimeout( tid ) Enter
undefined
```

この例では、タイマーを投入した直後にそれを取り消しており、"test" は表示されない。

4.4.6.1 タイマー処理の反復

先の setTimeout では登録された処理は 1 度だけ実行されるが、setInterval を使用するとタイマーによる関数の起動を繰り返すことができる。

書き方： `setInterval(関数, 経過時間, 引数並び)`

setTimeout と同様のタイマー処理を際限なく反復する。setInterval はインターバル ID (intervalID) を意味する正の整数値を返す⁷⁷。

setInterval で登録されたタイマーは clearInterval で取り消すことができる。

書き方： `clearInterval(インターバル ID)`

「インターバル ID」が示すタイマーを取り消す。戻り値はない (undefined)。

参考)

setTimeout の戻り値を clearInterval の引数に与えてタイマーを取り消すことができる。同様に setInterval の戻り値を clearInterval の引数に与えてタイマーを取り消すことができる。(あまり推奨されない)

⁷⁷ Node.js では Timeout オブジェクトを返す。

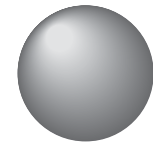
4.4.6.2 タイマーの反復処理の応用例

setInterval を応用した視覚効果の繰り返しの例を Timer01.html に示す。

記述例：Timer01.html

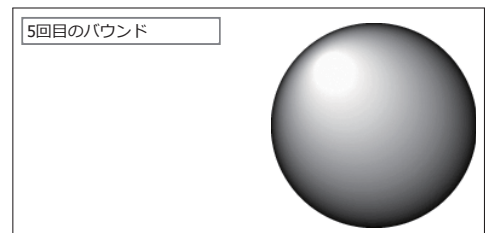
```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>Timer01</title>
6   <style>
7     #ball {
8       position: absolute;
9       top: 10pt;
10      left: 0pt;
11    }
12  </style>
13  <script>
14    let x = 0, dx = 2, n = 0, tid;
15    function f1() {
16      x += dx;
17      ball.style.left = x + "pt";
18      if ( x > 200 || x < 0 ) {
19        dx *= -1;
20        n++;
21        tx.value = n + "回目のバウンド";
22      }
23    }
24    function f0() {
25      tid = setInterval(f1,10);
26    }
27    window.addEventListener("load",f0);
28  </script>
29 </head>
30 <body>
31   <input type="text" id="tx">
32   
33 </body>
34 </html>
```

右の HTML 文書を Web ブラウザで表示すると、
下記のような画像が左右に移動を繰り返す。



BallGray01.gif

コンテンツの表示直後に 24～26 行目の関数 f0 が実行される。これにより関数 f1（15～23 行目）が 10ms 間隔で際限なく実行される。f1 では変数 x の値を変数 dx だけ変化させる。変数 x の値はボールの横位置を決定するものであり、17 行目の記述によって id="ball" の img 要素の CSS の left 属性に与えられる。また、x の値が 0～200 の範囲を超えると x の増分である dx の値の正負が反転（19 行目）し、ボールの移動方向が変わる。



Web ブラウザで表示したところ

4.5 表示環境に関する事柄

4.5.1 ディスプレイ（スクリーン）関連

表示に使用するディスプレイを表すオブジェクトに `window.screen` がある。これはグローバルオブジェクト `screen` としてアクセスできる。`screen` オブジェクトからスクリーンのピクセルサイズを取得することができる。(表 42)

表 42: `screen` オブジェクトのプロパティ（一部）

プロパティ	解 説
<code>width</code>	ディスプレイの横幅（単位:px）
<code>height</code>	ディスプレイの高さ（単位:px）
<code>availWidth</code>	ディスプレイの有効表示幅（単位:px）
<code>availHeight</code>	ディスプレイの有効表示高（単位:px）

`width`, `height` からは表示領域のサイズが得られるが、実際に表示に使用できる領域はこれより小さい場合が多い。例えば OS が UI として専有している領域⁷⁸ は表示可能な領域ではない。表示に使用できる領域のピクセルサイズは `availWidth`, `availHeight` から参照する。(次の例)

例. `screen` オブジェクトから表示領域のサイズを取得する

<pre>> screen.width 1600</pre>	<pre>> screen.availWidth 1600</pre>
<pre>> screen.height 1200</pre>	<pre>> screen.availHeight 1152</pre>

←左の値より少し小さい

これは Windows 環境において Web ブラウザのコンソールで実行した例であり、タスクバーの存在により有効表示高が少し小さくなっていることがわかる。

4.5.2 ブラウザウィンドウ、コンテンツページ関連

Web ブラウザのウィンドウの大きさに関する情報は `window` オブジェクトのプロパティ（表 43）から得られる。

表 43: `window` オブジェクトのプロパティ（一部）

プロパティ	解 説
<code>outerWidth</code>	ウィンドウの横幅
<code>outerHeight</code>	ウィンドウの高さ
<code>innerWidth</code>	コンテンツの表示領域の横幅
<code>innerHeight</code>	コンテンツの表示領域の高さ

Web ブラウザのコンテンツを表示する領域はビューポートと呼ばれ、その大きさは `innerWidth`, `innerHeight` から得られる。`outerWidth`, `outerHeight` は Web ブラウザのウィンドウ全体のサイズである。

例. `window` オブジェクトから Web ブラウザの各種サイズを取得する（Web ブラウザのコンソールでの実行）

<pre>> window.outerWidth 1370</pre>	<pre>> window.innerWidth 1354</pre>
<pre>> window.outerHeight 1052</pre>	<pre>> window.innerHeight 319</pre>

これは Web ブラウザのコンソールで実行した例である。コンテンツの表示領域はコンソールの表示領域を除外した部分であり、右側の実行結果の値は左側の実行結果の値よりも小さいことがわかる。

Web ブラウザに表示される HTML コンテンツの表示サイズに関する情報は `document.documentElement` のプロパティ（表 44）から得られる。

⁷⁸Windows の場合はタスクバーなど、macOS の場合は Dock などがある。スマートフォンなどの携帯情報端末においてはカメラやセンサーが配置される部分（ノッチ）などがある。

表 44: コンテンツの表示サイズに関する document.documentElement のプロパティ

プロパティ	解 説
clientWidth	ウィンドウ内に表示されているコンテンツの横幅（スクロールバーを除く）（単位:px）
clientHeight	ウィンドウ内に表示されているコンテンツの高さ（スクロールバーを除く）（単位:px）
scrollWidth	コンテンツ全体の横幅（隠れている部分を含む）（単位:px）
scrollHeight	コンテンツ全体の高さ（隠れている部分を含む）（単位:px）

表 44 を用いて HTML コンテンツの表示サイズを調べる例を ContentSize01.html を用いて示す。

記述例：ContentSize01.html

```

1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>ContentSize01</title>
6   <style>
7     div {
8       margin: 0pt;
9       padding: 0pt;
10      width: 3990pt;
11      height: 2990pt;
12      border: solid 5pt blue;
13      background-color: yellow;
14    }
15  </style>
16 </head>
17 <body>
18   <div></div>
19 </body>
20 </html>

```

左のコンテンツを Web ブラウザで表示すると次のようになる。



これは 4,000pt × 3,000pt のサイズ（border 含む）の div 要素であり、Web ブラウザの表示領域よりも大きい。この状態で Web ブラウザのコンソールでコンテンツサイズを調べる。（次の例）

例. コンテンツサイズの調査

```
> document.documentElement.clientWidth Enter
778 ←スクロールバーを除く表示部分の横幅
```

```
> document.documentElement.clientHeight Enter
198 ←スクロールバーを除く表示部分の高さ
```

```
> document.documentElement.scrollWidth Enter
5340 ←コンテンツ全体の横幅
```

```
> document.documentElement.scrollHeight Enter
4015 ←コンテンツ全体の高さ
```

4.5.3 表示の解像度を求める方法

コンテンツの表示の**解像度**は「1 インチの幅に描くことができる画素の数」で定義され、単位は dpi（dots per inch）または ppi（pixels per inch）である。また、表示のためのデバイスによっては画素の横幅と高さが異なる場合もあり、より厳密には、横方向の解像度と縦方向の解像度を別のものとして扱う。

使用中の Web ブラウザの解像度を直接的に取得する関数やメソッドは存在しない⁷⁹ が、1 インチの大きさの HTML 要素の横幅や高さを調査することで当該 Web ブラウザの解像度を知ることができる。この方法で解像度を調べる例を ScreenRatio01.html に示す。

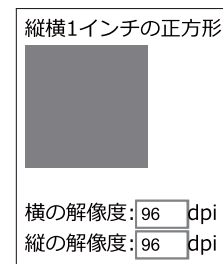
⁷⁹Web ブラウザや JavaScript エンジンの実装系の中にはそのような機能を提供しているものもある。


```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>ScrrRatio01</title>
6    <style>
7      #d1 {
8        margin: 0pt; padding: 0pt;
9        width: 1in; height: 1in;
10       border: 0pt;
11       background-color: gray;
12     }
13     #t1, #t2 { width: 24pt; }
14   </style>
15   <script>
16     function f1() {
17       const d1 = document.getElementById("d1");
18       const t1 = document.getElementById("t1");
19       const t2 = document.getElementById("t2");
20       const cs = window.getComputedStyle( d1 );
21       t1.value = cs['width'].replace("px","");
22       t2.value = cs['height'].replace("px","");
23     }
24     window.addEventListener("load",f1);
25   </script>
26 </head>
27 <body>
28   縦横1インチの正方形<br>
29   <div id="d1"></div><br>
30   横の解像度:<input type="text" id="t1">dpi<br>
31   縦の解像度:<input type="text" id="t2">dpi
32 </body>
33 </html>

```

左のコンテンツを Web ブラウザで表示すると次のようになる。



29 行目に記述した div 要素 (id="d1") の縦横のサイズを 7～12 行目の CSS の記述によって 1 インチにしており、20～22 行目の処理によって縦横のサイズをピクセル値で取得し、それを解像度の値としている。

注) ここで得られた解像度の値は文字列型である。

4.5.4 px (ピクセル) から pt (ポイント) への単位の変換

表示の解像度は使用するデバイスによって異なることがあり、Web デザインにおける HTML 要素のサイズ設定をピクセル単位で行うと、同じコンテンツを解像度の異なるデバイスで表示すると異なった大きさで表示される⁸⁰。従って、異なる解像度のデバイスにおいても同じ大きさで HTML コンテンツを表示するには、HTML 要素のサイズ設定はポイント単位で指定する必要がある。

HTML 要素の各種のサイズは基本的にはピクセル単位の値として取得されるが、ここでは、ピクセル単位 (単位:px) で得られた数値をポイント単位 (単位:pt) に変換する方法について解説する。

ポイント単位のサイズの考え方においては、1 ポイントは 1/72 インチである。従って、インチ単位の大きさの値 L_{in} をポイント単位の値 L_{pt} に変換するには、

$$L_{pt} = L_{in} \times 72$$

とすれば良い。また、ピクセル単位の大きさの値 L_{px} をインチ単位の値 L_{in} に変換するには、解像度の値 R_{dpi} を用いて、

$$L_{in} = \frac{L_{px}}{R_{dpi}}$$

とする。以上のことから、ピクセル単位の大きさの値 L_{px} をポイント単位の値 L_{pt} に変換するには、

$$L_{pt} = \frac{L_{px}}{R_{dpi}} \times 72$$

とすれば良い。この計算方法を用いて、解像度 R_{dpi} の下でピクセル単位の値 L_{px} 、ポイント単位の値 L_{pt} 、インチ単位の値 L_{in} を相互に変換する関数群が次のように実装できる。

⁸⁰もちろんこのことを意図してコンテンツを設計する場合もあり、事情に応じてサイズ設定に対する考え方は異なる。

実装例：単位変換の関数群

```

1  /* px -> pt */
2  function px2pt(Lpx,Rdpi) {
3      return Lpx*72/Rdpi;
4  }
5
6  /* pt -> px */
7  function pt2px(Lpt,Rdpi) {
8      return Lpt*Rdpi/72;
9  }
10
11 /* インチ -> px */
12 function in2px(Lin,Rdpi) {
13     return Lin*Rdpi;
14 }
15
16 /* px -> インチ */
17 function px2in(Lpx,Rdpi) {
18     return Lpx/Rdpi;
19 }
20
21 /* インチ -> pt */
22 function in2pt(Lin) {
23     return Lin*72;
24 }
25
26 /* pt -> インチ */
27 function pt2in(Lpt) {
28     return Lpt/72;
29 }

```

例. 解像度 96dpi 下で 192px のサイズをポイント値に変換

```
> px2pt(192,96) Enter
144    ←ポイント値
```

例. 解像度 96dpi 下で 144pt のサイズをピクセル値に変換

```
> pt2px(144,96) Enter
192    ←ピクセル値
```

例. 解像度 96dpi 下で 2 インチをピクセル値に変換

```
> in2px(2,96) Enter
192    ←ピクセル値
```

例. 解像度 96dpi 下で 192px をインチに変換

```
> px2in(192,96) Enter
2      ←インチ
```

例. 2 インチをポイント値に変換

```
> in2pt(2) Enter
144    ←ポイント値
```

例. 144pt をインチに変換

```
> pt2in(144) Enter
2      ←インチ
```

4.5.5 ビューポート内での HTML 要素の位置とサイズ

HTML 要素が Web ブラウザのビューポート内に表示されている位置を取得するには `getBoundingClientRect` メソッドを使用する。

書き方： `HTML 要素.getBoundingClientRect()`

ビューポート内での「HTML 要素」の位置とサイズを保持した `DOMRect` オブジェクトを返す。

`DOMRect` オブジェクトは当該 HTML 要素のビューポート上での位置とサイズを表現したものであり、border の外周を包む矩形領域を表す。(margin 領域は含まない) このオブジェクトには表 45 に示すようなプロパティがあり、各種の値を参照できる。

表 45: `DOMRect` オブジェクトのプロパティ

プロパティ	解 説
<code>x</code>	<code>DOMRect</code> オブジェクトの原点（左上隅）の横位置
<code>y</code>	<code>DOMRect</code> オブジェクトの原点（左上隅）の縦位置
<code>width</code>	<code>DOMRect</code> オブジェクトの横幅
<code>height</code>	<code>DOMRect</code> オブジェクトの高さ
<code>left</code>	上記 <code>x</code> と同じ。ただし <code>width</code> が負の場合は <code>x+width</code>
<code>top</code>	上記 <code>y</code> と同じ。ただし <code>height</code> が負の場合は <code>y+height</code>
<code>right</code>	<code>DOMRect</code> オブジェクトの右端の横位置。ただし <code>width</code> が負の場合は <code>x</code> と同じ
<code>bottom</code>	<code>DOMRect</code> オブジェクトの下端の縦位置。ただし <code>height</code> が負の場合は <code>y</code> と同じ

単位は全て px

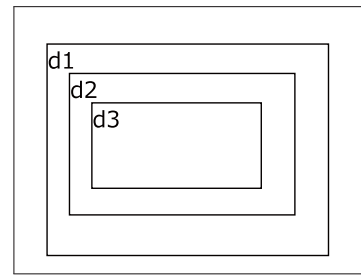
表 45 に示したプロパティ名には CSS の属性名と同じものもあるが、CSS の場合とは意味が異なるので注意が必要である。

`DOMRect` オブジェクトの使用例を `PosAndSize01.html` に示す。

記述例：PosAndSize01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>PosAndSize01</title>
6     <style>
7       div {
8         position: absolute;
9         top: 20px;    left: 15px;
10        border: solid 1px black;
11      }
12      #d1 { width: 200px; height: 150px; }
13      #d2 { width: 160px; height: 100px; }
14      #d3 { width: 120px; height: 60px; }
15    </style>
16  </head>
17  <body>
18    <div id="d1">d1
19      <div id="d2">d2
20        <div id="d3">d3</div>
21      </div>
22    </div>
23  </body>
24 </html>
```

左のコンテンツを Web ブラウザで表示すると次のようになる。



id="d2" の div 要素は id="d1" の div 要素の中にある。id="d3" の div 要素は id="d2" の div 要素の中にある。また 3 つの div 要素はその包含要素の左上から top:20px, left:15px の位置にある。

PosAndSize01.html を Web ブラウザで開き、コンソールの操作で DOMRect オブジェクトを取得していくつかのプロパティを参照する例を示す。

例. div 要素の DOMRect オブジェクトを取得する

```
> d1 = document.getElementById("d1") Enter
<div id="d1">...</div>

> d2 = document.getElementById("d2") Enter
<div id="d2">...</div>

> d3 = document.getElementById("d3") Enter
<div id="d3">...</div>
```

次に、得られた DOMRect オブジェクトのプロパティの値と CSS の同名のプロパティの値を比較する例を示す。

例. DOMRect オブジェクトのプロパティの値の参照

```
> d1.getBoundingClientRect()["top"] Enter
20
> d1.getBoundingClientRect()["left"] Enter
15
> d2.getBoundingClientRect()["top"] Enter
41
> d2.getBoundingClientRect()["left"] Enter
31
> d3.getBoundingClientRect()["top"] Enter
62
> d3.getBoundingClientRect()["left"] Enter
47
```

例. CSS のプロパティの値の参照

```
> window.getComputedStyle( d1 )["top"] Enter
'20px'
> window.getComputedStyle( d1 )["left"] Enter
'15px'
> window.getComputedStyle( d2 )["top"] Enter
'20px'
> window.getComputedStyle( d2 )["left"] Enter
'15px'
> window.getComputedStyle( d3 )["top"] Enter
'20px'
> window.getComputedStyle( d3 )["left"] Enter
'15px'
```

DOMRect オブジェクトのプロパティの値は Web ブラウザに実際に表示した場合の位置を意味していることがわかる。

次に、HTML 要素の横幅と高さに関するプロパティを参照する例を示す。

例. DOMRect オブジェクトのプロパティの値の参照

```
> d1.getBoundingClientRect()["width"] Enter
202
> d1.getBoundingClientRect()["height"] Enter
152
```

例. CSS のプロパティの値の参照

```
> window.getComputedStyle( d1 )["width"] Enter
'200px'
> window.getComputedStyle( d1 )["height"] Enter
'150px'
```

DOMRect オブジェクトのプロパティから得られた横幅と高さは border の太さを加算したものになっていることがわかる。

4.6 ローカル環境でのデータの保存

多くの Web ブラウザにはローカル環境にデータを保存する機能が備わっている。これは、Web ブラウザで表示するコンテンツリソースの**オリジン**毎にデータを保存する機能である。ここで言うオリジンとは、当該コンテンツにアクセスするためのホスト（ドメイン名や IP アドレス）と URL のスキーム（http, https など）、ポート番号の組み合わせとして定義される。

オリジン毎に隔離されたデータ保持の機能は、当該コンテンツの閲覧作業に固有の情報を保持することを可能とし、更に、異なる Web サイト間での情報漏洩を防ぐためのセキュリティにもなる。またこのような、データの保持をオリジン毎に隔離する考え方を**同一オリジンポリシー**⁸¹（SOP：Same-Origin Policy）という。

ここでは Web ブラウザに提供されている localStorage と sessionStorage について解説する。

4.6.1 localStorage

window オブジェクトのプロパティに localStorage があり、これにローカル環境の情報をオリジン毎に保存することができる。localStorage はそのままグローバルオブジェクトとして扱うことができる。1つのオリジンに対して保存可能なデータサイズは約 5MB 程度である。

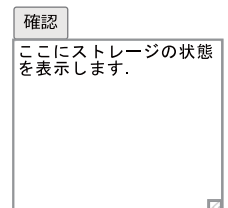
localStorage は Storage クラスのオブジェクトである。次に示すサンプル StorageTest01.html を用いて localStorage について解説する。

記述例：StorageTest01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>StorageTest01</title>
6     <script>
7       function f0() {
8         ta1.value = "ここにストレージの状態を表示します。";
9       }
10      function f1() {
11        let k, v, buf="";
12        for ( let i=0; i<localStorage.length; i++ ) {
13          k = localStorage.key(i);
14          v = localStorage.getItem(k);
15          if ( i != 0 ) { buf += "\n"; }
16          buf += k + " : " + v;
17        }
18        ta1.value = buf;
19      }
20    </script>
21  </head>
22  <body onLoad="f0()">
23    <input type="button" value="確認" id="b1" onClick="f1()">
24    <br>
25    <textarea col="22" rows="10" id="ta1"></textarea>
26  </body>
27 </html>
```

これを Web ブラウザで表示すると右のような表示となる。「確認」ボタンをクリックすると下部のテキストエリアに localStorage に保存されているデータの内容がキーと値のペアとして表示される。

記述例の各部に関しては後に解説する。



4.6.1.1 データの保存

setItem メソッドで localStorage にデータを保存することができる。

書き方： localStorage.setItem(キー, 値)

⁸¹同一生成元ポリシー、同一源泉ポリシーとも呼ばれる。また、異なるオリジン間でリソースを共有することを、オリジン間リソース共有（CORS：Cross-Origin Resource Sharing）という。

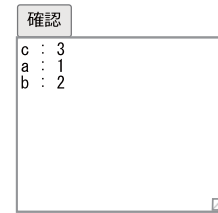
「キー」に対する「値」を保存する。戻り値はない (undefined)。「キー」,「値」は文字列として与える。また保存できない場合⁸² にはエラー (例外) が発生する。

以下に `setItem` メソッドの実行例を示す。

例. データの保存

```
> localStorage.setItem( "a", "1" ) Enter  
undefined  
> localStorage.setItem( "b", "2" ) Enter  
undefined  
> localStorage.setItem( "c", "3" ) Enter  
undefined
```

左のように実行した後,「確認」ボタンをクリックすると下のように `localStorage` の内容が表示される。



4.6.1.2 データの取得

`getItem` メソッドで `localStorage` に保存されたデータを取得することができる。

書き方: `localStorage.getItem(キー)`

「キー」に対する「値」を取得して文字列として返す。「キー」は文字列として与える。また「キー」が示すデータが存在しない場合には `null` が返される。

例. データの取得 (先の例の続き)

```
> localStorage.getItem( "a" ) Enter ←キー "a" に対する値を参照  
'1' ←得られた値  
> localStorage.getItem( "b" ) Enter ←キー "b" に対する値を参照  
'2' ←得られた値  
> localStorage.getItem( "c" ) Enter ←キー "c" に対する値を参照  
'3' ←得られた値
```

4.6.1.3 キーの取得

`key` メソッドで `localStorage` に保存されたデータのキーを取得することができる。

書き方: `localStorage.key(インデックス)`

非負の整数で与えられた「インデックス」に対するキーを取得して文字列として返す。ただし、保存されているデータのキーの順序は保存された順ではないことに注意すること。実際にデータが存在する範囲外や負の「インデックス」を与えると `null` を返す。

例. インデックスを指定してキーを参照する (先の例の続き)

```
> localStorage.key( 0 ) Enter ←インデックス 0 のキー  
'c'  
> localStorage.key( 1 ) Enter ←インデックス 1 のキー  
'b'  
> localStorage.key( 2 ) Enter ←インデックス 2 のキー  
'a'  
> localStorage.key( 3 ) Enter ←範囲外のインデックス  
null
```

全てのキーの配列を取得するには `Object` の `keys` メソッドを用いて次のようにする。

書き方: `Object.keys(localStorage)`

例. 全キーの配列を取得する (先の例の続き)

```
> Object.keys( localStorage ) Enter  
(3) ['c', 'a', 'b'] ←得られた配列
```

⁸²Web ブラウザの設定によってはデータの保存が許可されない場合がある。また、`localStorage` が満杯である場合もデータの保存ができない。

4.6.1.4 サンプルプログラム：全データの取得

先に解説した `Object.keys` を用いて `localStorage` の全データを取得することができる。(次の例)

記述例：

```
1 let keys = Object.keys( localStorage );
2 let allDat = {};
3 for ( key of keys ) {
4     allDat[key] = localStorage.getItem(key);
5 }
```

このコードを実行することで、`localStorage` の全データがオブジェクト `allDat` に得られる。次の例は、上のコードを実行した後の確認である。

例. `localStorage` の全データの取得

```
> allDat  ←内容確認
{c: '3', a: '1', b: '2'} ←全データが得られている。
```

4.6.1.5 データの消去

`removeItem` メソッドで `localStorage` に保存されたデータを削除することができる。

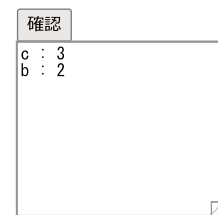
書き方： `localStorage.removeItem(キー)`

「キー」が示すデータを消去する。戻り値はない (`undefined`)。存在しない「キー」が与えられた場合は `localStorage` に変化はない。

例. キー "a" のデータの削除 (先の例の続き)

```
> localStorage.removeItem( "a" )
undefined
```

この処理の後、「確認」ボタンをクリックすると右のように表示され、キー "a" のデータが削除されていることがわかる。



当該オリジンに対する `localStorage` を空にするには `clear` メソッドを次のような形式で使用する。

書き方： `localStorage.clear()`

戻り値はない (`undefined`)。

例. `localStorage` を空にする (先の例の続き)

```
> localStorage.clear()
undefined
```

この処理の後、「確認」ボタンをクリックすると右のように表示され、`localStorage` が空になっていることがわかる。



4.6.2 sessionStorage

`sessionStorage` は `localStorage` と同様の扱いができるが、データの保存が Web ブラウザのセッションに限られるという制約がある。すなわち、Web ブラウザのタブやウィンドウを閉じると `sessionStorage` の内容は消滅する。ただし、同一のタブ内でページの表示を切り替えても `sessionStorage` は保持される。

4.7 画像, 音声の扱い

4.7.1 静止画像の扱い

4.7.1.1 静止画像を動的に切り替える方法

HTML の `img` 要素の `src` 属性を変更することにより, 動的に画像を切替えることができる. これに関するサンプルを `ImageTest01.html` に示す.

記述例: `ImageTest01.html`

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>ImageTest01</title>
6     <script>
7       let n = 0, imgs = new Array();
8       function f1() {
9         im.src = imgs[n];
10        n++;
11        if ( n > 2 ) {
12          n = 0;
13        }
14      }
15      function f0() {
16        document.getElementById("bt").addEventListener("click",f1);
17        for ( let i=1; i<4; i++ ) {
18          fname = "ImageTest01_"+ i + ".gif";
19          imgs.push(fname);
20        }
21      }
22      window.addEventListener("load",f0);
23    </script>
24  </head>
25  <body>
26    <input type="button" value="次" id="bt"><br>
27    <img alt="画像" id="im">
28  </body>
29 </html>
```

このサンプルは, 当該ファイルと同じディレクトリに図 31 に示す画像ファイルがあることを前提とする. このサンプルを Web ブラウザで表示すると, その直後に関数 `f0` が実行され, 配列 `imgs` に図 31 の 3 つのファイルの名前が格納される.



図 31: 使用する画像ファイル

コンテンツの表示の先頭には「次」ボタンが表示され, それをクリックすると, 図 31 の 3 つの画像が順番に表示 (図 32) される.

「次」ボタンをクリックされると関数 `f1` が起動し, サンプルの 9 行目で `img` 要素の `src` 属性に異なる画像ファイルの名前が設定され, 画像の表示が切り替わる.

4.7.1.2 `img` 要素の生成と追加

`Image` コンストラクタで `img` 要素を生成することができる.

書き方: `new Image(横幅, 高さ)`

「横幅」と「高さ」(単位: `px`) で指定したサイズの `img` 要素 (`HTMLImageElement` オブジェクト) を作成して返す. 「横幅」と「高さ」は省略可能で, その場合は `document.createElement("img")` と同等の処理となる.

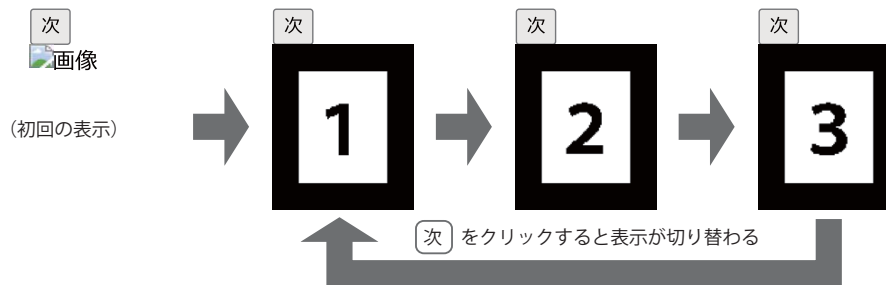


図 32: ImageTest01.html の Web ブラウザ上での動作

Image コンストラクタの使用例を ImageTest01-0.html に示す.

記述例: ImageTest01-0.html

```

1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>ImageTest01-0</title>
6   </head>
7   <body></body>
8 </html>

```

このコンテンツの body 要素は空であり, Web ブラウザで表示すると空白のページが表示される. その状態で Web ブラウザのコンソールを開き, img 要素の生成 (Image コンストラクタ) と body 要素への追加 (appendChild メソッド) を行う例を示す.

例. img 要素の生成と追加

```

> im = new Image() Enter
<img>
> im.src = "ImageTest01.1.gif" Enter
'ImageTest01.1.gif'
> document.body.appendChild( im ) Enter


```

左の処理の結果, Web ブラウザの表示が下ようになる.



```

> im = new Image() Enter
<img>
> im.src = "ImageTest01.2.gif" Enter
'ImageTest01.2.gif'
> document.body.appendChild( im ) Enter


```

左の処理の結果, Web ブラウザの表示が下ようになる.



```

> im = new Image() Enter
<img>
> im.src = "ImageTest01.3.gif" Enter
'ImageTest01.3.gif'
> document.body.appendChild( im ) Enter


```

左の処理の結果, Web ブラウザの表示が下ようになる.



Image オブジェクトの横幅と高さはそれぞれ width, height プロパティから参照できる. (次の例)

例. Image オブジェクトの横幅と高さの参照 (先の例の続き)

```

> im.width Enter ←横幅を求める
100
> im.height Enter ←高さを求める
117

```

4.7.2 音声、動画の扱い

HTML の audio 要素を用いて HTML 文書に音声データを組み込むことができる。

書き方： `<audio src=音声データ>テキスト</audio>`

「音声データ」には対象リソースのパスや URI を与える。この要素に非対応のブラウザの場合は「テキスト」が表示される。この要素は音声であり、Web ブラウザのビューポート内には表示されないが、audio 要素の開始タグ内に controls を記述すると音声の再生の制御に関するコントロールパネル（制御パネル）が表示される。（下の例）

記述例	Web ブラウザでの表示（Google Chrome の場合）
<code><audio src="sound01.mp3" controls></audio></code>	

controls は audio 要素の**論理属性**である。論理属性にはこの他にも繰り返し再生（ループ再生）を指定する loop、自動再生を指定する autoplay、音消しを指定する muted などがある。

Web ブラウザによっては再生できない音声フォーマットがある。その場合は複数の音声データを audio 要素の配下に source 要素として書き並べると良い。

書き方： `<audio>
 <source src=音声データ 1 type=メディアタイプ>
 <source src=音声データ 2 type=メディアタイプ>
 :
</audio>`

このように記述することで、「音声データ 1」、「音声データ 2」…の中から再生可能な音声データを探して再生することができる。ここで注意すべき点として、あくまでも 1 つの audio 要素は 1 つの音源を表すものであり、「音声データ 1」、「音声データ 2」…を順番に再生するものではなく、source 要素は再生可能な選択肢であるということがある。

4.7.2.1 HTMLMediaElement クラス

DOM において audio 要素は HTMLAudioElement クラスのオブジェクトである。このクラスは HTMLMediaElement クラスを継承しており、後で解説する video 要素の HTMLVideoElement クラスも HTMLMediaElement を継承している。従って、audio オブジェクトも video オブジェクトも共通のプロパティを持っており、基本的には同じ方法で扱うことができる。

4.7.2.2 再生と一時停止

HTMLMediaElement オブジェクトに play メソッドを実行すると再生が始まる。また、pause メソッドを実行すると一時停止となる。

書き方： `HTMLMediaElement オブジェクト.play()`

書き方： `HTMLMediaElement オブジェクト.pause()`

「HTMLMediaElement オブジェクト」の再生、一時停止を行う。引数はない。play メソッドは当該メディアの再生処理に関する**プロミスオブジェクト**⁸³（Promise）を返す。pause メソッドは値を返さない。（undefined）

pause メソッドによって一時停止した場合はその再生位置が保持され、play メソッドを実行するとその再生位置から再生が始まる。

4.7.2.3 各種のプロパティ

HTMLMediaElement オブジェクトには表 46 に挙げるようなプロパティがある。

⁸³非同期処理の最終的な完了もしくは失敗を表すオブジェクト。

表 46: HTMLMediaElement オブジェクトのプロパティ (一部)

プロパティ	意 味	備 考
src	メディアデータ	メディアデータの URL の設定と参照ができる。
volume	音量	0～1.0 の値で設定と参照ができる。
currentTime	再生位置	秒単位の値で設定と参照ができる。
duration	再生時間	秒単位で全再生時間が参照ができる。ライブストリーミングメディアのように予め全長がわからない場合は +Infinity となる。
playbackRate	再生速度	通常の再生速度との比率の値として設定と参照ができる。
paused	停止状態	一時停止中の場合は true, それ以外は false となる。(参照のみ)
muted	音消し状態	ミュートされている状態を true, それ以外を false として, 設定と参照ができる。
autoplay	自動再生	自動再生する状態を true, しない状態を false として, 設定と参照ができる。
loop	繰り返し再生	繰り返し再生する状態を true, しない状態を false として, 設定と参照ができる。
controls	制御パネル	制御パネルを表示する状態を true, しない状態を false として, 設定と参照ができる。

これらプロパティから各種の値を参照することができる。また、値の設定（変更）ができるものもあり、メディアデータの再生に関する制御ができる。

4.7.2.4 各種のイベント

HTMLMediaElement オブジェクトに関する各種のイベントを表 47 に挙げる。

表 47: HTMLMediaElement オブジェクトに関するイベント (一部)

イベント	解 説
play	メディアの再生が始まった時に発生する。
pause	メディアの再生が一時停止した時に発生する。
ended	メディアの再生が終了した時に発生する。
error	メディアの読み込み中もしくは再生中にエラーが起こった時に発生する。
ratechange	メディアの再生速度が変化した時に発生する。
timeupdate	メディアの再生位置（時刻）が変化した時に発生する。
volumechange	メディアの音量やミュート状態が変化した時に発生する。
loadstart	メディアデータの読み込みが開始した時に発生する。

4.7.2.5 動画：HTMLVideoElement オブジェクト

HTML の video 要素で動画を扱うことができる。video 要素は HTMLVideoElement クラスのオブジェクトであり、これは HTMLMediaElement クラスを継承している。従って基本的な扱いに関しては先の audio 要素と共通している。ここでは、HTMLVideoElement オブジェクト固有のプロパティを表 48 に挙げる。

表 48: HTMLVideoElement オブジェクトのプロパティ (一部)

プロパティ	解 説
videoWidth	ビデオリソースの内在的な横幅（単位:px）（参照のみ）
videoHeight	ビデオリソースの内在的な高さ（単位:px）（参照のみ）
width	HTML 要素としての表示領域の横幅（単位:px）
height	HTML 要素としての表示領域の高さ（単位:px）
poster	動画再生前に表示するポスター画像（画像の URL）

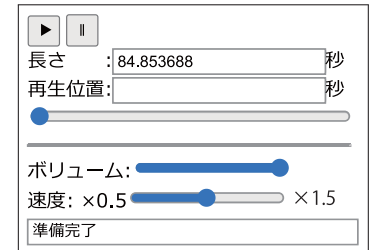
4.7.2.6 応用例

サウンドのプレーヤーを実現するサンプル MediaPlayer01.html を示す。

記述例：MediaPlayer01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>MediaPlayer01</title>
6     <style>
7       #b1,#b2 { width:20pt; height:20pt; }
8       hr, #r0, #t3 { width:200pt; }
9       hr { margin-left:0pt; }
10    </style>
11    <script>
12      function f1() {
13        a1.play(); t3.value = "再生中";
14      }
15      function f2() {
16        a1.pause(); t3.value = "一時停止";
17      }
18      function f3() { a1.volume = r1.value; }
19      function f4() { a1.playbackRate = r2.value; }
20      function f5() {
21        t2.value = a1.currentTime;
22        r0.value = a1.currentTime/a1.duration;
23      }
24      function f6() {
25        a1.currentTime = r0.value*a1.duration;
26      }
27      function f7() { t3.value = "再生終了"; }
28      function f0() {
29        b1.addEventListener("click",f1);
30        b2.addEventListener("click",f2);
31        r1.addEventListener("input",f3);
32        r2.addEventListener("input",f4);
33        a1.addEventListener("timeupdate",f5);
34        a1.addEventListener("ended",f7);
35        r0.addEventListener("input",f6);
36        t1.value = a1.duration;
37        t2.value = "";
38        t3.value = "準備完了";
39        r0.value = 0; r1.value = 1.0; r2.value=1.0;
40      }
41      window.addEventListener("load",f0);
42    </script>
43  </head>
44  <body>
45    <audio src="sound01.mp3" id="a1"></audio>
46    <input type="button" value="▶" id="b1">
47    <input type="button" value="||" id="b2"><br>
48    長さ :<input type="text" id="t1">秒<br>
49    再生位置:<input type="text" id="t2">秒<br>
50    <input type="range" id="r0" min="0" max="1.0"
51      step="0.001"><hr>
52    ボリューム:<input type="range" id="r1"
53      min="0" max="1.0" step="0.001"><br>
54    速度: ×0.5<input type="range" id="r2"
55      min="0.5" max="1.5" step="0.001">×1.5<br>
56    <input type="text" id="t3">
57  </body>
58 </html>
```

左のコンテンツを Web ブラウザで表示すると次のように表示される。



最下段のテキストフィールドには状態が表示され、コンテンツを読み込んだ直後は「準備完了」と表示され、「長さ」のフィールドにサウンドの再生時間（長さ）が表示される。

「▶」ボタンをクリックすると音声の再生が始まり、最下段のテキストフィールドに「再生中」と表示される。また、再生中は「再生位置」のフィールドに再生位置（時刻）が表示される。

「||」ボタンをクリックすると一時停止となり、最下段のテキストフィールドに「一時停止」と表示される。

再生が終了すると最下段のフィールドに「再生終了」と表示される。

「ボリューム」と「速度」は随時調整可能である。

「再生位置」のフィールドの下のスライダは再生位置を反映するもので、再生中は自動的に移動する。またこのスライダをマウスで動かすと、該当の位置に再生位置が変更される。

サウンドデータは 45 行目の audio 要素で読み込んでいる。

UI は 46～56 行目で実装しており、それらに対するイベント処理の設定を関数 f0 で行っている。f0 はコンテンツが Web ブラウザに読み込まれた直後に実行される。

4.8 非同期処理と Promise

Web ブラウザや Node.js の JavaScript 言語処理系は基本的にはシングルスレッドの処理系であり、複数の処理を複数のスレッドとして同時に並行して実行することができない。したがって、通信や入出力をはじめとする時間のかかる処理を実行する場合、それら処理が完了するまでの間は他の実行可能な処理を行うべきである。

JavaScript 言語処理系はイベント駆動型プログラミング⁸⁴ の形で複数の処理の実行を管理している。この形の制御では、複数の処理をイベントループと呼ばれる機構で管理し、そこに登録された処理を順次監視する。そして、イベント発生などによって実行可能となった処理に制御を移し、それが終了あるいは一時停止した場合などに再び制御をイベントループに戻して、次の実行可能な処理を探す。

終了までに時間のかかる処理を非同期タスクとしてイベントループ上で管理してその終了を監視する形にすれば、当該処理の実行中に他のタスクの実行はブロックされない。非同期タスクとして重要なものにタイマータスク⁸⁵、I/O タスク⁸⁶、Promise（後述）を解決するマイクロタスクなどがある。

4.8.1 Promise オブジェクト

Promise オブジェクトは非同期処理を管理するものであり、その状態を保持する。直感的にわかりやすい例として、先の「4.7.2 音声、動画の扱い」（p.155）で解説したマルチメディアの再生における Promise オブジェクトがある。動画や音声といったマルチメディアである HTMLMediaElement オブジェクトに play メソッドを実行するとそれが再生されるが、再生中は他の処理をブロックしない。また play メソッドは実行した時点で Promise オブジェクトを返し、それがマルチメディアの状態（再生中、再生終了、再生失敗といった状態）を保持している。すなわち、マルチメディアの再生は通常の JavaScript プログラムの実行とは別に Promise オブジェクトの管理下で実行されていると見ることができる。実際にシステム内部で行われるマルチメディアの再生やファイル入出力、通信の処理自体は非同期 I/O 操作と呼ばれるもので、JavaScript プログラムの解釈と実行の処理とは別の範疇のこととして行われている。

非同期 I/O 操作ではないプログラム（関数）の実行を Promise オブジェクトの下で管理することもできる。これを応用すると、システム全体の動作をブロックすることなく、時間のかかる処理を実行することができる。

《関数の実行を管理する Promise オブジェクトの作成》

```
new Promise( async function( resolve, reject ) {  
    try {  
        await 処理に時間のかかる関数;  
        resolve( 戻り値 );  
    } catch( error ) {  
        エラー発生時の処理;  
        reject( 失敗時のメッセージ );  
    }  
})
```

「処理に時間のかかる関数」を管理する Promise オブジェクトを生成し、その実行を開始する。この関数のことを非同期関数と呼ぶ。この非同期関数が成功裡に終了した場合に Promise オブジェクトは「戻り値」を持つ。また非同期関数の実行においてエラーが発生した場合は「エラー発生時の処理」を実行し、Promise オブジェクトは「失敗時のメッセージ」を持つ。「error」はエラー発生時の Error オブジェクトである。「resolve」、「reject」はシステムから自動的に与えられる関数であり、非同期関数が正しく終了した場合に resolve 関数を、エラー発生時には reject 関数を実行する形にする。

Promise コンストラクタに与える引数はアロー関数式

```
async (resolve, reject) => { 実行部 }
```

でもよい。

⁸⁴ 「4.4 イベント駆動型プログラミング」（p.137）参照のこと。

⁸⁵ 「4.4.6 タイマー」（p.143）を参照のこと。

⁸⁶ 後で解説する Fetch API や FileReader API、付録で解説する Web サーバ関連の API などがこれに該当する。

非同期関数（上記「処理に時間のかかる関数」）は次のように記述する。

《非同期関数》

```
async function( 引数 ) {  
    (以下のような処理を含む反復処理)  
    処理  
    実行の制御をイベントループに戻す処理  
}
```

このような形式で構成された非同期関数は「処理」を行った後に「実行の制御をイベントループに戻す処理」を実行する流れを繰り返す。この「処理」の部分はシステムのスレッドを専有するが、その後の「実行の制御をイベントループに戻す処理」によってプログラムの実行制御を一旦イベントループに戻す。そして再びこの非同期関数に実行制御が戻され、実行が継続される。

「実行の制御をイベントループに戻す処理」の部分は次のような「非同期の実行待ち関数」として実装することができる。

■ 非同期の実行待ち関数

```
function sleep(ms) { return new Promise( resolve => setTimeout(resolve,ms) ); }
```

このように定義された sleep 関数を

```
await sleep(0)
```

として実行することで実行の制御をイベントループに戻すことができる。sleep 関数の引数に 0 を与えているが、これはプログラムの実行を休止するのが目的ではなく、await によって現行の処理を一時停止して実行の制御をイベントループに戻すことを目的としていることを意味する。

4.8.2 実装例

ここでは、時間のかかる処理を Promise オブジェクトとしてイベントループ上で管理する例を挙げて、非同期処理について解説する。

■ 終了まで時間がかかる処理の事例

次の longTask01.js に示す関数 longTask について考える。

記述例：longTask01.js

```
1 function longTask() {  
2     const u = Math.pow(2,-29); // 完了の値  
3     const g = 1.0 - u;        // 失敗の値  
4     while ( true ) {  
5         let r = Math.random();  
6         if ( r < u ) {  
7             console.log( 'Complete' );  
8             break;  
9         } else if ( r > g ) {  
10            let e = new Error( "大きすぎる乱数" );  
11            e.name = "TooBigRandomError";  
12            throw e;  
13        }  
14    }  
15 }
```

上に示した関数 longTask は Math.random 関数を使って一様乱数（0 以上 1.0 未満）を次々と生成するもので、 2^{-29} 未満の非常に小さな乱数が得られた場合に処理を終了し、 $1 - 2^{-29}$ より大きな乱数が得られた場合にエラー（"TooBigRandomError/大きすぎる乱数"）を発生する。ただし、 2^{-29} 未満の乱数や $1 - 2^{-29}$ より大きな乱数が得られる確率は極めて小さく、longTask が終了（あるいはエラーが発生）するまでに時間がかかる。（読者の計算機環境では関数定義内の 2^{-29} の部分を適宜変更して試されたい）このことを Web ブラウザのコンソールで確認する。

上記 longTask01.js の function 文をそのまま Web ブラウザのコンソールに入力して関数 longTask を定義した後でそれを実行する例を示す。

例. 関数 longTask を定義した後でそれを実行

```
> function longTask() { Enter    ←関数定義
  ⋮
  (中略)
  ⋮
undefined    ←関数定義処理の結果
> longTask(); console.log("次の処理") Enter    ←関数の実行と「次の処理」
Complete      ←実行開始から時間をおいて表示される
次の処理      ←上の表示の後で表示される
```

あるいは、実行開始から時間をおいて次のようなエラーが表示される.

```
VM49:12 Uncaught TooBigRandomError: 大きすぎる乱数    ← Google Chrome ブラウザの場合のエラー表示
    at longTask (<anonymous>:10:12)
    at <anonymous>:1:1
```

longTask の実行が終わるまで他の処理はブロックされて実行されない. 従って「Complete」の表示の後に「次の処理」が表示される.

上の例のように、時間のかかる処理が他の処理の実行をブロックすることは、実用的なプログラミングにおいては避けねばならない. 例えば先のような longTask 関数を使用した Promise01-1.html では Web ブラウザ上での動作に問題が生じる.

記述例: Promise01-1.html (良くない例)

```
1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>Promise01-1</title>
6    <style>
7      #t1 { width: 320px; }
8      #t2 { width: 270px; }
9    </style>
10   <script>
11     // 時間のかかる処理
12     function longTask() {
13       const u = Math.pow(2,-31); // 完了の値
14       const g = 1.0 - u;         // 失敗の値
15       while ( true ) {
16         let r = Math.random();
17         if ( r < u ) {
18           console.log( 'Complete' );
19           break;
20         } else if ( r > g ) {
21           let e = new Error( "大きすぎる乱数" );
22           e.name = "TooBigRandomError";
23           throw e;
24         }
25       }
26     }
27     // 上記の処理を実行するする関数
28     function exeLongTask() {
29       try {
30         longTask();
31         t2.value = "処理完了";
32       } catch( error ) {
33         t2.value = "失敗: "+error.name+"/"+error.message;
34       }
35     }
36     // 実行
37     function f0() {
38       t1.value = "";
39       t2.value = "実行中";
40       exeLongTask(); // 時間のかかる処理
41       // 日付時刻の表示
42       let t = setInterval( "t1.value = String( new Date() )", 100 );
43     }
44   </script>
```

```

45 </head>
46 <body onLoad="f0()">
47   日付時刻:
48   <input type="text" id="t1"><br>
49   exeLongTask():
50   <input type="text" id="t2">
51 </body>
52 </html>

```

(更に処理時間がかかる形に乱数の大きさの判定の部分を変更してある)

Promise01-1.html を Web ブラウザで表示すると図 33 のようになる。

日付時刻: Sat Aug 03 2024 18:40:56 GMT+0900 (日本標準時)
 exeLongTask(): 処理完了

図 33: Promise01-2.html を Web ブラウザで開いたところ

「exeLongTask()」の部分は「処理完了」もしくは「失敗: TooBigRandomError/大きすぎる乱数」と表示されるが、Web ブラウザの更新ボタンをクリックすると、表示の更新までに時間がかかることがわかる。これは longTask 関数が処理を終えるまで Web ブラウザの動作をブロックしていることが原因であり、この関数の処理が終わった段階で「日付時刻」のフィールドも変化し始める。

次に、longTask 関数が Web ブラウザの他の処理をブロックしないように改善した Promise01-2.html を示す。

記述例：Promise01-2.html

```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>Promise01-2</title>
6    <style>
7      #t1 { width: 320px; }
8      #t2 { width: 273px; }
9    </style>
10   <script>
11     // 非同期sleep関数
12     function sleep(ms) {
13       return new Promise( resolve => setTimeout(resolve,ms) );
14     }
15     // 時間のかかる処理
16     async function longTask() {
17       const u = Math.pow(2,-10); // 完了の値
18       const g = 1.0 - u;        // 失敗の値
19       while ( true ) {
20         let r = Math.random();
21         if ( r < u ) {
22           console.log( 'Complete' );
23           break;
24         } else if ( r > g ) {
25           let e = new Error( "大きすぎる乱数" );
26           e.name = "TooBigRandomError";
27           throw e;
28         }
29         await sleep(0);
30       }
31     }
32     // 上記の処理のPromiseオブジェクトを生成する関数
33     function exeLongTask() {
34       return new Promise( async function(resolve,reject) {
35         try {
36           await longTask();
37           resolve( "処理完了" );
38         } catch( error ) {
39           reject( "失敗: "+error.name+"/"+error.message );
40         }
41       } );
42     }
43     // 実行
44     function f0() {

```

```

45         t1.value = "";
46         t2.value = "実行中";
47         // Promiseの処理
48         exeLongTask().then( function(msg) {
49             t2.value = msg;
50         }).catch( function(e) {
51             t2.value = e;
52         });
53         // 日付時刻の表示
54         let t = setInterval( "t1.value = String( new Date() )", 100 );
55     }
56 </script>
57 </head>
58 <body onLoad="f0()">
59     日付時刻:
60     <input type="text" id="t1"><br>
61     Promiseの状態:
62     <input type="text" id="t2">
63 </body>
64 </html>

```

(乱数の大きさの判定の部分を変更してある)

このサンプルでは、関数 longTask 内の while ループで毎回

```
await sleep(0);
```

を実行しており、このタイミングでプログラムの実行の制御がシステムのイベントループに移されるので、while ループが他のイベント処理をブロックしない。またこの関数 longTask の実行は、exeLongTask 関数が生成する Promise オブジェクトによって非同期処理として管理され、await によってその終了（完了もしくはエラー）が検知される。

exeLongTask が生成する Promise オブジェクトの完了のイベント（もしくはエラーの発生）を受けて起動する処理は、関数 f0 内において then メソッドと catch メソッドによって登録されている。

4.9 JSON

JavaScript のオブジェクト (Object) や配列 (Array) のリテラル表記は柔軟でしかも汎用性が高く、多くのプログラミング言語やアプリケーションプログラムにおけるデータの表現として採用されており、**JSON** (JavaScript Object Notation) として標準化⁸⁷ されている。

JSON は**データ表現言語**の1つとして普及しており、テキストデータ (文字列) の形式で記述され、異なるシステム間でのデータの交換に使用される。その際のエンコーディングはBOM なし UTF-8とする。JavaScript 言語処理系でも、データとしてのオブジェクト (Object のインスタンス) や配列 (Array のインスタンス) を JSON 形式の文字列に変換して通信やファイル入出力に用いる。

JSON の仕様は JavaScript のリテラル表記法と完全には同じではなく、概ね次のようなものである。

- 文字列データは二重引用符「"」で括る。
- オブジェクトのキーの部分は二重引用符「"」で括る。
- 数値は整数、浮動小数点数、指数表現が使えるが、10 進数表現に限る。
- 数値は二重引用符「"」で括らない。
- 論理値 true, false は小文字で記述する。(二重引用符「"」で括らない)
- null は小文字で記述する。(二重引用符「"」で括らない)

4.9.1 JSON データの作成

JSON.stringify を使用して JavaScript のオブジェクト (Object のインスタンス) を JSON 形式の文字列に変換することができる。

書き方： JSON.stringify(データ)

JavaScript の「データ」を JSON の文字列にして返す。

例. JavaScript のオブジェクトを JSON の文字列に変換する

```
> obj = { name:"Taro", age:32, height:171, weight:70, favorite:["apple","orange"] } Enter  
{name: 'Taro', age: 32, height: 171, weight: 70, favorite: Array(2)}  
> jsn = JSON.stringify( obj ) Enter ←上記 obj を JSON に変換  
'{"name":"Taro","age":32,"height":171,"weight":70,"favorite":["apple","orange"]}'
```

この例で得られた文字列 jsn は更に適切に変換 (Blob などへの変換) することで、通信路に向けて送信したり、ファイルに保存することができる。

参考) JSON.stringify には、「データ」内の変換対象を指定するためのオプションや、JSON 生成時のテキスト整形に関するオプションも指定できる。

4.9.2 JSON データの展開

JSON.parse を使用して、JSON の文字列を JavaScript のオブジェクトや配列に変換することができる。

書き方： JSON.parse(JSON 文字列)

「JSON 文字列」を展開した結果の JavaScript のデータを返す。

例. JSON の文字列を展開する (先の例の続き)

```
> obj2 = JSON.parse( jsn ) Enter ←先の jsn を展開する  
{name: 'Taro', age: 32, height: 171, weight: 70, favorite: Array(2)}
```

この例の処理によって、先の例で作成した jsn の内容を展開した JavaScript のデータが obj2 に得られる。

⁸⁷RFC 8259, ECMA-404 など。

4.10 通信のためのデータ変換

4.10.1 URL エンコーディング

HTTP 通信では、URL として扱えない文字を使用する場合、それらを **URL エンコーディング** (URI エンコーディング、パーセントエンコーディングと呼ぶこともある)⁸⁸ によって使用可能な文字に変換して扱う。実際の Web アプリケーション開発においては、クライアント (Web ブラウザなど) と HTTP サーバ (Web サーバ) の通信が基本的な部分であり、両者の通信においては URL エンコーディングされたデータを取り扱う。

JavaScript には、特殊文字や Unicode 文字などを URL エンコーディングする処理と逆変換 (デコーディング) する処理を行うための関数が備わっている。

書き方: `encodeURIComponent(変換対象文字列)`

書き方: `decodeURIComponent(変換対象文字列)`

`encodeURIComponent` 関数は「変換対象文字列」をエンコーディングしたものを返す。`decodeURIComponent` 関数はエンコーディングされている「変換対象文字列」をデコーディングして元の文字列に復元したものを返す。

例. 文字列をエンコーディング/デコーディングする処理

```
> r = encodeURIComponent( "JavaScript 言語" )  [Enter]    ←文字列のエンコーディング
'JavaScript%E8%A8%80%E8%AA%9E'                ←変換された文字列
> decodeURIComponent( r )  [Enter]    ←逆変換 (復元) の処理
'JavaScript 言語'                                ←元に戻った
```

上記の関数は URL のクエリストリング (クエリパラメータ) の部分を変換対象とする。すなわち、URL の表記全体を取り扱うものではない。(次の例)

例. URL の表記全体を変換する試み

```
> u = "http://localhost/cgi01?名 1=値 1&名 2=値 2"  [Enter]    ← URL 表記の文字列
'http://localhost/cgi01?名 1=値 1&名 2=値 2'
> encodeURIComponent( u )  [Enter]    ←それをエンコーディングすると…
'http%3A%2F%2Flocalhost%2Fcgi01%3F%E5%90%8D1%3D%E5%80%A41%26%E5%90%8D2%3D%E5%80%A42'
```

この例からわかるように、URL の表記全体をエンコーディングすると、その結果は URL の表記ではなくなる。従って、クエリストリングなどを含めた URL 表記全体を正しくエンコーディング/デコーディングするには次の関数を使用する。

書き方: `encodeURI(変換対象文字列)`

書き方: `decodeURI(変換対象文字列)`

使用方法は先の関数に準じる。

例. URL の表記全体を正しく変換する (先の例の続き)

```
> r = encodeURI( u )  [Enter]    ← URL 表記の文字列をエンコーディング
'http://localhost/cgi01?%E5%90%8D1=%E5%80%A41&%E5%90%8D2=%E5%80%A42'  ←正常
> decodeURI( r )  [Enter]    ←デコーディング
'http://localhost/cgi01?名 1=値 1&名 2=値 2'                                ←元に戻った
```

4.10.2 Base64

ASCII コードの範囲にないバイナリデータを 64 個の可読文字に変換する手法に Base64 がある⁸⁹。ASCII 文字列を Base64 形式に変換、逆変換する下記の関数がある。

書き方: `btoa(変換対象文字列)`

書き方: `atob(変換対象文字列)`

`btoa` 関数は ASCII 文字から成る「変換対象文字列」を Base64 形式 (文字列) に変換して返す。`atob` 関数は `btoa` 関数によって作成された Base64 形式の文字列を変換前の文字列に復元したものを返す。

⁸⁸RFC 3986 として標準化されている。

⁸⁹RFC 4648 として標準化されている。

例. ASCII の文字列を Base64 に変換し、復元する処理

```
> a = btoa( "This is a pen." )  ← Base64 に変換
'VGhpcyBpcyBhIHBlbi4='      ←変換後の文字列
> atob( a )  ← Base64 形式を元の文字列に復元
'This is a pen.'              ←元の文字列
```

参考) Base64 に変換されたデータは元のデータに比べて 30%以上サイズが大きくなる。

btoa, atob 関数で扱えるのは ASCII 文字列に限られる。一般的なバイナリデータを扱うには別の方法が必要となる。次に紹介する base64-js ライブラリは Uint8Array オブジェクトと Base64 形式文字列を相互に変換する機能を提供する。

4.10.2.1 base64-js ライブラリ

base64-js は有志のエンジニア達が開発して GitHub⁹⁰ で公開しているプログラムライブラリである。このライブラリが提供する API を利用することで、バイナリデータと Base64 形式文字列の相互の変換が簡便な方法で実現できる。このライブラリは、Web ブラウザ上の JavaScript で利用できるのもので、Web アプリケーションに組み込んで使用できる。また、Node.js 上でも利用可能である。

■ Web アプリケーションでの利用

base64-js は公式インターネットサイトから入手して使うことができる。具体的には、当該ライブラリである base64js.min.js を入手して、HTML コンテンツの head 要素内に

```
<script src="base64js.min.js"></script>
```

として読み込む。

Uint8Array 形式のバイナリデータを Base64 形式文字列に変換するには base64js.fromByteArray を用いる。

書き方： base64js.fromByteArray(バイナリデータ)

「バイナリデータ」を Base64 形式文字列に変換したものを返す。

Base64 形式文字列を Uint8Array 形式のバイナリデータに変換するには base64js.toByteArray を用いる。

書き方： base64js.toByteArray(Base64 データ)

「Base64 データ」を Uint8Array 形式のバイナリデータに変換したものを返す。

次に示す HTML コンテンツ base64jsTest01.html を用いて、マルチバイト文字を Base64 形式の文字に変換（エンコード）し、それを復元（デコード）する例を示す。

記述例：base64jsTest01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>base64jsTest01</title>
6   <script src="base64js.min.js"></script>
7 </head>
8 <body></body>
9 </html>
```

これを Web ブラウザで表示して、コンソールで変換処理を行う例を次に示す。

例. マルチバイト文字を Base64 に変換（エンコード）する処理

```
> u8a = new TextEncoder().encode( "言語" )  ← "言語" を Uint8Array に変換
Uint8Array(6) [ ... ]                      ← Uint8Array のデータになった
> b = base64js.fromByteArray( u8a )  ←上記データを Base64 形式文字列に変換
'6KiA6Kqe'                                  ←変換結果
```

文字列 "言語" が Base64 形式文字列に変換できている。次にこれを元の文字列に復元する処理を示す。

⁹⁰<https://github.com/beatgammit/base64-js>

例. Base64 から文字列を復元（デコード）する処理（先の例の続き）

```
> u8a = base64js.toByteArray( b )  [Enter]    ←上記 Base64 形式文字列を Uint8Array に変換
Uint8Array(6) [ ... ]              ← Uint8Array のデータになった
> new TextDecoder().decode( u8a )  [Enter]    ←上記 Uint8Array データを文字列に変換
' 言語'                             ←変換結果
```

■ Node.js での利用

Node.js で base64-js を使用するには、パッケージ管理ツール npm（Node Package Manager）で必要なものをシステムにインストールする。

インストール方法： `npm install base64-js`

システムのコマンドシェルで上記コマンドを発行すると `node_modules` ディレクトリ下に `base64-js` ディレクトリとして `base64-js` がインストールされる。

Node.js で実行するプログラムの冒頭で、

```
var base64js = require('base64-js')
```

と記述することで、Web ブラウザの JavaScript の場合と同様の記述方法で `base64-js` の機能を使うことができる。

4.11 ファイルの入出力

Web ブラウザではセキュリティの観点から、ローカルの計算機環境の資源へのアクセスは制限されている。ただし、ユーザとの対話処理を通じた形でのファイルの入出力は可能である。

4.11.1 ファイルの読み込み

HTML の input 要素には、ローカルの計算機環境のファイルシステムの内容を選択するダイアログを表示する機能があり、これと File API を組み合わせることで、ファイルの読み込みが実現できる。

4.11.1.1 input 要素によるファイル選択ダイアログ

書き方： `<input type="file">`

この形式の input 要素はファイル選択機能となる。また、次のような属性を与えることができる。

■ accept=ファイル形式

「ファイル形式」には、読み込み対象のファイルの種類を意味する拡張子を文字列の形式で与える。例えば HTML ファイルの場合は「`accept=".html"`」と記述する。複数のファイル形式を指定する場合は、それぞれの拡張子をコンマ「,」で区切って書き連ねた文字列を与える。例えば、HTML ファイルとテキストファイルを読み込み対象とする場合は「`accept=".html,.txt"`」などと記述する。

読み込み対象のファイルの種類は拡張子以外にも **メディアタイプ** (MIME タイプ)⁹¹ を記述しても良い。この accept 属性に記述するものを **固有ファイル型指定子** と呼ぶ。

■ multiple (論理属性)

ファイル選択ダイアログで複数のファイルを同時に選択する場合にこの属性を記述する。

【ファイル選択ダイアログのサンプル】

次に示すサンプル InputTypeFile01.html を用いてファイル選択ダイアログについて解説する。

記述例：InputTypeFile01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>InputTypeFile01</title>
6 </head>
7 <body>
8   <input type="file" accept=".html" multiple id="t1">
9 </body>
10</html>
```

左のコンテンツを Web ブラウザで表示すると次のように表示される。

ファイル選択 選択されていません

8 行目の input 要素は Web ブラウザ上では「ファイル選択」と表示されたボタンとして現れる。

このサンプル InputTypeFile01.html の 8 行目に記述されている input 要素は HTMLInputElement クラスのインスタンスであり、t1 というオブジェクトとして参照できる。「ファイルを選択」をクリックするとファイル選択のためのダイアログが表示される。ダイアログの操作で選択されたファイルは、t1 のプロパティ value, files から参照できる。すなわち、value プロパティからは選択したファイルのパスを示す文字列が、files プロパティからは選択したファイル（複数選択可能）のリストを意味する FileList オブジェクトが得られる。これらプロパティを開発者ツールのコンソールで確認する。

例. ファイル選択ダイアログの処理をする前の確認

```
> t1.value    ← value プロパティの確認
''          ←空文字列

> t1.files    ← files プロパティの確認
FileList {length: 0}    ←選択されたファイルは 0 個
```

次に、ダイアログでファイル選択をした後に同様の確認を行う。「ファイルを選択」ボタンをクリックしてファイルを選択 (図 34) する。

⁹¹p.29 の「メディアタイプについて」を参照のこと。

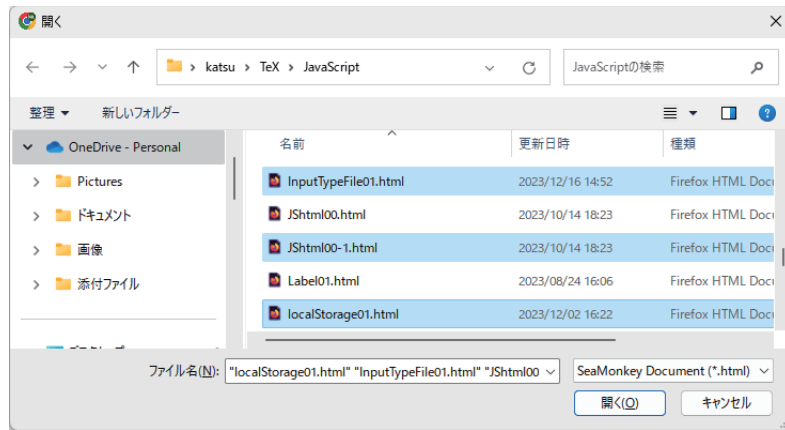


図 34: 「ファイル選択」をクリックしてダイアログを表示したところ

図 34 は HTML ファイルを 3 つ選択する例である。この状態で「開く」をクリックするとダイアログが閉じられ、Web ブラウザでの input 要素の表示が次のようになる。

ファイル選択 3 ファイル

ファイル選択の結果、オブジェクト t1 にそれらの情報が得られる。(次の例)

例. ファイル選択ダイアログの処理の後の確認 (先の例の続き)

```
> t1.value Enter ← value プロパティの確認
'C:\fakepath\InputTypeFile01.html' ← 選択されたファイル群の最初のもののパス
> t1.files Enter ← files プロパティの確認
FileList {0: File, 1: File, 2: File, length: 3} ← 選択されたファイルは 3 個
```

この処理で注意すべき点として、value プロパティに得られたファイルのパスが実際のものとは異なる偽装されたものとなることがある。これは、システム外部からの不正な情報収集に対するセキュリティである。

選択したファイルに実際にアクセスするには、files プロパティの要素を用いる。

例. 選択したファイルを確認する (先の例の続き)

```
> t1.files[0] Enter ← 得られたファイルの先頭のものの確認
File {name: 'InputTypeFile01.html', lastModified: 1702705976311,...} ← 参照結果
```

FileList オブジェクトの要素は選択された個々のファイルを示す File オブジェクトであり、上の例では選択したファイル群の最初の File オブジェクトを参照している。

4.11.1.2 FileReader によるファイルの読み込み

ファイル選択ダイアログの処理によって得られた File オブジェクトを開くには FileReader クラスのオブジェクトを使用する。

書き方: `FileReader()`

これによって FileReader クラスのインスタンスが生成され返される。以後はそれを用いてファイル内容の読み込みができる。ファイル読み込みのための FileReader オブジェクトのメソッドを表 49 に挙げる。

表 49: ファイルの読み込みのためのメソッド (一部)

メソッド	解 説
<code>readAsText(f)</code>	File オブジェクト (もしくは Blob) の f からテキストデータを読み込む。第 2 引数にエンコーディングを文字列形式で与えることもできる。
<code>readAsBinaryString(f)</code>	File オブジェクト (もしくは Blob) の f からバイナリデータを読み込む。得られたバイナリデータは文字列として解釈される。
<code>readAsArrayBuffer(f)</code>	File オブジェクト (もしくは Blob) の f からバイナリデータを読み込む。得られたバイナリデータは <code>ArrayBuffer</code> として得られる。

表 49 に挙げたメソッドはファイルの読み込みをメインのプログラムとは別に非同期に実行し `undefined` を返す、読み込みが完了すると FileReader オブジェクトに `loadend` イベントが発生する。従って、読み取ったファイルの内容の処理

は、このイベントの発生を待って行うことになる。また、ファイルから読み取った内容は、FileReader オブジェクトの result プロパティから参照できる。

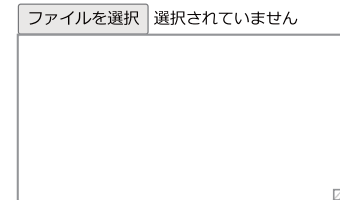
【テキストデータを読み込む例】

ローカルの計算機環境のファイルシステムからテキストファイルを選んで読み込む例を FileReader01.html に示す。

記述例：FileReader01.html

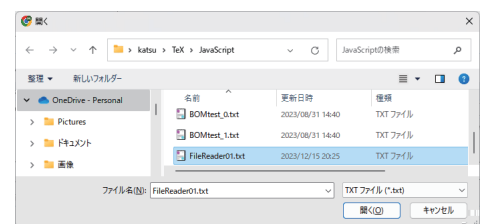
```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>FileReader01</title>
6   <style>
7     #ta1 {
8       width: 200pt; height: 100pt;
9       white-space: nowrap;
10      overflow: auto;
11    }
12  </style>
13  <script>
14    const fr = new FileReader();
15    function f1() {
16      fr.readAsText(t1.files[0]);
17    }
18    function f2() {
19      ta1.value = fr.result;
20    }
21    function f0() {
22      t1.addEventListener("change", f1);
23      fr.addEventListener("loadend", f2);
24    }
25    window.addEventListener("load", f0);
26  </script>
27 </head>
28 <body>
29   <input type="file" accept=".txt" id="t1"><br>
30   <textarea id="ta1"></textarea>
31 </body>
32 </html>
```

左のコンテンツを Web ブラウザで表示すると次のように表示される。



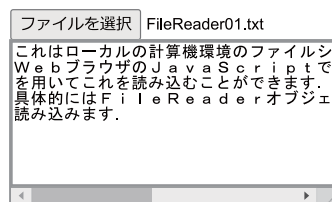
29 行目の input 要素は Web ブラウザ上では「ファイルを選択」と表示されたボタンとして現れる。

「ファイルを選択」のボタンをクリックするとファイル選択ダイアログが表示され、テキストファイルを選ぶ（下記）ことができる。



上記の 23 行目で FileReader オブジェクト fr に loadend イベントのハンドラ f2 を登録している。これにより、ファイルの読み込みが終了すると、関数 f2 が起動し、ファイルから読み取った内容を textarea 要素の value プロパティに与える。

ファイル選択ダイアログでテキストファイルを選択して「開く」ボタンをクリックすると下図のように、そのファイルを開いて textarea 要素に表示する。



読み取ったテキストファイルの内容を textarea に表示している。

【バイナリデータを読み込む例】

p.21 で取り上げた画像ファイル IMGobj01.png（図 18）の内容をバイナリデータとして読み取る例を FileReader02.html に示す。



図 18: 画像ファイル IMGobj01.png

記述例：FileReader02.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>FileReader02 </title>
6   <style>
7     hr {
8       width: 180pt;
9       margin-left: 0pt;
10    }
11  </style>
12  <script>
13    const fr = new FileReader();
14    function f1_1() {
15      fr.readAsBinaryString(t1.files[0]);
16    }
17    function f1_2() {
18      fr.readAsArrayBuffer(t2.files[0]);
19    }
20    function f2() {
21      console.log("読み終了");
22    }
23    function f0() {
24      t1.addEventListener("change", f1_1);
25      t2.addEventListener("change", f1_2);
26      fr.addEventListener("loadend", f2);
27    }
28    window.addEventListener("load", f0);
29  </script>
30 </head>
31 <body>
32   読み方法1：BinaryString形式 <br>
33   <input type="file" accept=".png" id="t1">
34   <hr>
35   読み用法2：ArrayBuffer形式 <br>
36   <input type="file" accept=".png" id="t2">
37 </body>
38 </html>
```

左のコンテンツを Web ブラウザで表示すると次のようになる。

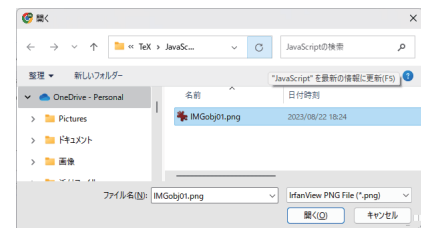
読み方法1：BinaryString形式

ファイルを選択 選択されていません

読み用法2：ArrayBuffer形式

ファイルを選択 選択されていません

このプログラムは PNG 形式の画像ファイルを読み込むものである。ファイルの選択と読み込みの流れは先の FileReader01.html と概ね同じであり、「ファイルを選択」のボタンをクリックするとファイル選択のダイアログ（下記）が表示され、ファイルを選択して「開く」をクリックするとファイルの内容が読み込まれる。



“読み方法1”の側の「ファイルを選択」のボタンをクリックすると readAsBinaryString メソッドで、“読み方法2”の側の「ファイルを選択」のボタンをクリックすると readAsArrayBuffer メソッドでファイルが読み込まれる。読み込み後は Web ブラウザのコンソールに「読み終了」と表示される。

ファイルを読み込んだ後で Web ブラウザのコンソールにおいて読み込んだ内容を確認する。

例. “読み方法1”の処理後の作業

> fr.result ← 読み結果の確認
(エスケープシーケンスで表現されたバイナリデータの表示…)

例. “読み方法2”の処理後の作業

> fr.result ← 読み結果の確認
ArrayBuffer(4763)… ← ArrayBuffer オブジェクト

4.11.2 ファイルの保存

Web ブラウザには HTML の a 要素のリンク先のリソースをローカルの計算機環境に安全にダウンロードする機能が備わっている。これを応用すると、JavaScript のプログラムとしてファイルを保存する処理が実現できる。

Blob をはじめとするオブジェクトには、それを指し示す URL を一時的に割り当てることができ、その URL を HTML の a 要素の href 属性に与えることができる。これにより、HTML の a 要素介してそれが指し示すリソースをローカルの計算機環境のファイルシステムにダウンロードすることができる。

Web ブラウザにはグローバルの URL オブジェクト⁹²があり、この静的メソッド createObjectURL を用いると Blob などのオブジェクトを表す URL を生成することができる。

書き方： URL.createObjectURL(対象オブジェクト)

「対象オブジェクト」の URL を生成して返す。

⁹²URL オブジェクトは window オブジェクトの配下のものである。

createObjectURL で得られた URL を HTML の a 要素の href 属性に、ダウンロードの際のファイルの名前を download 属性に与え、当該 a 要素のオブジェクトに対して click メソッドを実行することで URL が指し示すリソースをダウンロードすることができる。ダウンロード先のディレクトリは Web ブラウザのダウンロード用ディレクトリである。

書き方： a 要素.click()

「a 要素」に対してマウスでクリックしたのと同じ処理を実行する。

上の一連の処理を応用して、textarea 要素の内容（value 属性の値）をファイル「Untitled.txt」としてダウンロードする例を、次の Download01.html で示す。

実装例：Download01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>Download01</title>
6     <style>
7       #ta1 { width: 200pt; height: 60pt; }
8       #dwn { visibility: hidden;}
9     </style>
10    <script>
11      function f1() {
12        const b = new Blob( [ta1.value], {type:"text/plain"} );
13        const u = window.URL.createObjectURL(b);
14        console.log(u);
15        dwn.download = "Untitled.txt";
16        dwn.target = "_blank";
17        dwn.href = u;
18        dwn.click();
19      }
20    </script>
21  </head>
22  <body>
23    <textarea id="ta1"></textarea><br>
24    <input type="button" value="保存" id="b1" onClick="f1()">
25    <a id="dwn"></a>
26  </body>
27 </html>
```

これを Web ブラウザで表示すると図 35 の (a) のように表示される。表示された textarea に文字を入力して「保存」ボタンをクリックする（図 35 の (b)）と、その内容がテキストファイル「Untitled.txt」として Web ブラウザに設定されたダウンロードディレクトリに保存される。

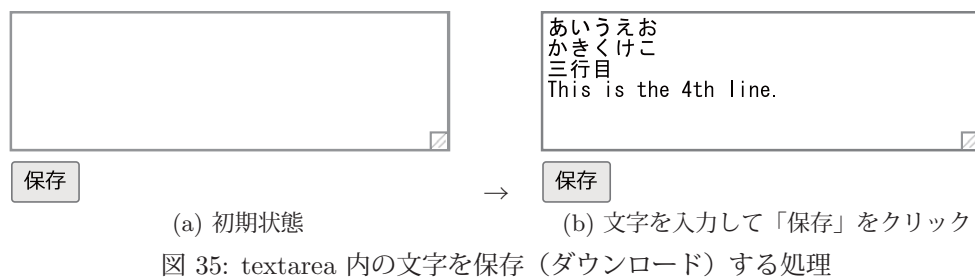


図 35: textarea 内の文字を保存（ダウンロード）する処理

ファイルを保存する際、下記のような URL が Web ブラウザのコンソールに出力される。

例. textarea の内容を持つ Blob の URL

blob:null/c285260e-3203-476f-bba6-dcaa2580d8f4

コンソールに表示される URL は一時的なものであり、処理の度に異なるものが得られる。

4.12 ドラッグアンドドロップ

多くの Web ブラウザはドラッグアンドドロップに対応しており、コンテンツ内の要素をマウスでドラッグ (drag) することができる。マウスで HTML 要素を他の要素の上にドラッグし、そこでマウスボタンを放すことでドロップが起こる。

多くの HTML 要素はデフォルトの状態ではドラッグアンドドロップには対応しておらず⁹³，マウスによる移動ができない。そのため、ドラッグ操作の対象とする HTML 要素に対してドラッグ操作を可能とするための設定が必要となる。このことを次のサンプル DnD01_0.html を例に上げて解説する。

記述例：DnD01_0.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>DnD01_0</title>
6   <style>
7     div {
8       position: absolute;
9       width: 200px; height: 80px;
10      border: solid 3px;
11      font-size: 24px;
12      text-align: center;
13      line-height: 80px;
14    }
15    #dv1 {
16      top: 5px; left: 5px;
17      border-color: blue;
18    }
19    #dv2 {
20      top: 5px; left: 240px;
21      border-color: red;
22    }
23  </style>
24 </head>
25 <body>
26   <div id="dv1">これをドラッグ</div>
27   <div id="dv2">ここへドラッグ</div>
28 </body>
29 </html>
```

このサンプルでは 2 つの div 要素で実現した四角形（青と赤で色分け）を表示する。div 要素は最初はドラッグ処理に対応しておらず、青の枠を赤の枠の上にドラッグしようとする、div 内のテキストをドラッグすることになる。（図 36）



枠は動かず、文字をドラッグ（選択）することになる
図 36: ドラッグの試み

HTML 要素をドラッグ可能にするには、その要素の `draggable` 属性に `"true"` を設定する。すなわち、青い枠の div 要素を

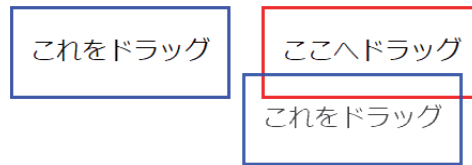
```
<div id="dv1" draggable="true">これをドラッグ</div>
```

に修正するとドラッグが可能になる。（図 37）

4.12.1 ドラッグアンドドロップに伴う処理

ドラッグアンドドロップにおいてはドラッグされる対象とドロップされる対象があり、前者にドラッグが起こると `dragstart` イベントが、後者にドロップが起こると `drop` イベントが発生する。また、前者をドラッグして後者の上に重ねている間は `dragover` イベントが連続して発生する。

⁹³一部の要素（img 要素や a 要素など）はデフォルトでドラッグに対応している。



青の枠がドラッグに反応して移動する

図 37: draggable="true" と設定した場合

4.12.1.1 dragstart イベントを受けて行う処理

dragstart イベントはマウスによってドラッグされた要素に対して発生する。このイベントを受けて行うべき処理は、当該イベントオブジェクトの `dataTransfer` 属性に対する値の設定である。

マウスによるドラッグの操作を行うと、対象の要素が視覚的に移動し、それをドロップした際に、ドロップ先の要素に対して事前に設定しておいた値を渡す。すなわち、その渡すべき値を dragstart のイベントオブジェクトの `dataTransfer` 属性に設定しておく。

先のサンプル `DnD01_0.html` において、ドラッグする青い枠 (`id="dv1"` の div 要素) のドラッグ開始時に `dataTransfer` の値を設定する例を示す。

```
let dv1 = document.getElementById("dv1");
dv1.addEventListener("dragstart", function(e) {
  e.dataTransfer.setData("text/plain", 文字データ );
});
```

この例では、`id="dv1"` の div 要素のドラッグ開始時に、イベントオブジェクトに「文字データ」を設定している。この処理は、当該イベントオブジェクトの `dataTransfer` 属性に対して `setData` メソッドによって行う。 `setData` メソッドの第1引数にはデータのメディアタイプ (MIME タイプ) を、第2引数には設定するデータを与える。上の例はテキストの「文字データ」を与えるためのもので、メディアタイプとして `"text/plain"`⁹⁴ を指定している。

4.12.1.2 dragover イベントを受けて行う処理

dragover は、ドロップを受け付ける側の要素の上に何かドラッグ操作で重ねられた (さしかかった) ときに発生する。このときの重要な処理として、ドラッグによってもたらされるデータが元のデータの複製 (コピー) であるか、あるいは元のものに移転したものかを設定することがある。このために、イベントオブジェクトの `dataTransfer.dropEffect` 属性に `"copy"` もしくは `"move"` を設定する。

ドラッグアンドドロップのためのイベント処理は、他のイベント処理と混乱が起こることが多く、dragover のイベントを扱う場合は `preventDefault` メソッドで他のイベント処理を抑止しておく安全である。

4.12.1.3 drop イベントを受けて行う処理

drop は、ドロップを受け付ける側の要素の上に何かドロップされたときに発生する。この場合も `preventDefault` メソッドで他のイベント処理を抑止しておく安全である。

4.12.1.4 イベントオブジェクトの target 属性

イベントオブジェクトの `target` 属性には、そのイベントが発生した HTML 要素が保持されている。イベント処理の際、この部分から対象の HTML 要素に関する様々な値 (子要素、属性など) を取得することができる。

【サンプルに沿った解説】

先に挙げたサンプル `DnD01_0.html` をドラッグアンドドロップに対応する形に改変した `DnD01.2.html` を示す。

記述例: `DnD01.2.html`

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
```

⁹⁴ 単純なテキストデータを意味する MIME タイプ。

```

5   <title>DnD01_2</title>
6   <style>
7     div {
8       position: absolute;
9       width: 200px;      height: 80px;
10      border: solid 3px;
11      font-size: 24px;    text-align: center; line-height: 80px;
12    }
13    #dv1 {
14      top: 5px;    left: 5px;
15      border-color: blue;
16    }
17    #dv2 {
18      top: 5px;    left: 240px;
19      border-color: red;
20    }
21  </style>
22  <script>
23    function f0() {
24      //--- dv1 のドラッグ操作に関する設定 ---
25      let dv1 = document.getElementById("dv1");
26      dv1.addEventListener("dragstart", function(e) {
27        e.dataTransfer.setData("text/plain",e.target.textContent);
28        console.log("ドラッグ開始："+e.target.id);
29        console.log("    内容="+e.target.textContent);
30      });
31      //--- dv2 のドラッグ操作の受け付けに関する設定 ---
32      let dv2 = document.getElementById("dv2");
33      // ドラッグがさしかかった際の処理
34      dv2.addEventListener("dragover", function(e) {
35        e.dataTransfer.dropEffect = "copy";
36        e.preventDefault();
37        console.log("ドラッグ受け付け："+e.target.id);
38      });
39      // ドロップされた際の処理
40      dv2.addEventListener("drop", function(e) {
41        e.preventDefault();
42        console.log("ドロップ受け付け："+e.target.id);
43        console.log("    受信データ："+e.dataTransfer.getData("text/plain"));
44      });
45    }
46    window.addEventListener("load",f0);
47  </script>
48 </head>
49 <body>
50   <div id="dv1" draggable="true">これをドラッグ</div>
51   <div id="dv2" >ここへドラッグ</div>
52 </body>
53 </html>

```

このサンプルはドラッグアンドドロップの処理を行う。Web ブラウザに表示された青い枠をドラッグするとドラッグ開始時に

```

ドラッグ開始：dv1
    内容=これをドラッグ

```

と Web ブラウザのコンソールに表示される。このとき表示される「内容=」の後の文字列は、当該イベントオブジェクトの `target` 属性からテキスト要素 (`textContent` 属性の値) を参照したものである。

青い枠を赤い枠の上に重ねると

```

ドラッグ受け付け：dv2

```

が複数連続して Web ブラウザのコンソールに表示される。

青い枠を赤い枠の上にドロップすると

```

ドロップ受け付け：dv2
    受信データ：これをドラッグ

```

と Web ブラウザのコンソールに表示される。

4.13 プログラムの分割開発

規模の大きな Web アプリケーションを構築する際、JavaScript のプログラムを複数のファイルに分割して作成することが一般的である。ここでは、プログラムの分割開発に関することを解説する。

4.13.1 単純な分割開発

最も簡単な方法として、プログラムを複数のファイルとして作成して読み込むという形がある。例えば次のサンプル multiScript01.html, multiScript01-1.js, multiScript01-2.js を例に挙げて考える。

記述例：multiScript01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>multiScript01</title>
6   <script src="multiScript01-1.js"></script>    <!-- 1つ目のファイルの読み込み -->
7   <script src="multiScript01-2.js"></script>    <!-- 2つ目のファイルの読み込み -->
8   <script>
9     console.log( c );                          // 計算結果をコンソールに出力
10  </script>
11 </head>
12 <body></body>
13 </html>
```

記述例：multiScript01-1.js

```
1 let a = 1, b = 2;
2 function f1(x,y) { return x+y; }
```

記述例：multiScript01-2.js

```
1 let c = f1(a,b);
```

主たる HTML ファイル multiScript01.html は2つの JavaScript ファイル multiScript01-1.js, multiScript01-2.js をこの順番で読み込み、計算結果の変数 c を Web ブラウザのコンソールに出力するものである。この HTML ファイルを Web ブラウザで表示すると、コンソールに「3」と表示される。

この例からもわかるように、複数の script 要素に分割して与えられた JavaScript プログラムが、この順番で1つのプログラムとして連結されて実行されている様子がわかる。

この方法では、複数の script 要素のプログラムが同じスコープで扱われ、クラスや関数、変数の名前が共通のものとして扱われる。このことを Web ブラウザのコンソールで確認する例を次に示す。

例. multiScript01-1.js で定義された変数、関数をコンソールで確認

```
> a Enter    ←変数 a の値が
1          ←参照できている
> b Enter    ←変数 b の値が
2          ←参照できている
> f1(5,6) Enter ←関数 f1 が
11         ←実行できている
```

これは当然のことのように思われるかもしれないが、規模の大きなシステムを分割開発する際に名前の衝突をはじめとする問題の原因となることもあるので、次に解説するモジュールの形式で分割開発することが推奨される。

4.13.2 モジュールによる分割開発

モジュールは独立したコンテキスト（スコープ）を持つプログラムであり、次のような記述で HTML に組み込む。

```
<script src=モジュールのパス type="module"></script>
```

「モジュールのパス」には読み込むモジュールファイルのパスを指定する。このとき URL は使えないことに注意すること。

このようにして読み込まれたモジュールは独自のコンテキストとして扱われ、モジュール内で定義されたクラス、関数、変数はそのモジュール内でのみ有効である。ただし、モジュールから DOM にはアクセスできる。これらのことを次のサンプル moduleTest01.html, moduleTest01.js で確かめる。

記述例：moduleTest01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>moduleTest01</title>
6   <style>
7     input[type="text"] {
8       width: 50px;
9       border: solid 1.5px #888888;
10    }
11  </style>
12  <script src="moduleTest01.js"
13    type="module"></script>
14 </head>
15 <body>
16   a=<input type="text" id="t1"><br>
17   b=<input type="text" id="t2"><br>
18   c=<input type="text" id="t3">
19 </body>
20 </html>
```

記述例：moduleTest01.js

```
1 let a = 1, b = 2;
2 function f1(x,y) { return x+y; }
3 let c = f1(a,b);
4 // DOMへのアクセス
5 document.getElementById("t1").value = a;
6 document.getElementById("t2").value = b;
7 document.getElementById("t3").value = c;
```

上のモジュールファイルは左の HTML ファイルと同じディレクトリにあるとする。この HTML を Web ブラウザで表示すると次のようになる。

a=
b=
c=

モジュールから DOM の input 要素に値を設定できていることがわかる。

注意) モジュールスクリプトの読み込みについて

moduleTest01.html を Web ブラウザで表示する場合、セキュリティの事情から http プロトコルで開くこと。
(例. <http://127.0.0.1:8080/moduleTest01.html>)
これには Web サーバが必要となる。ローカルの計算機環境で Web サーバを実現する方法に関しては巻末付録「A.4 ローカルの計算機環境で Web サーバを起動する簡易な方法」(p.293)を参照のこと。

注意) モジュールスクリプトのキャッシュについて

モジュールスクリプトは HTML の内容とは異なる仕組みで Web ブラウザにキャッシュ（一時保存）されることがあり、その場合は、モジュールスクリプトに加えた変更は Web ブラウザの更新ボタンだけでは反映されないことがある。これに関しては「B.1 スクリプトのキャッシュの抑止」(p.306)を参照のこと。

上のサンプルでは、moduleTest01.js をモジュールとして HTML に読み込んでいるので、定義された変数 a, b, c や関数 f1 はこのモジュールスクリプト内でのみ有効である。このことを Web ブラウザのコンソールで確認する（次の例）

例. モジュール内で定義された変数、関数の確認（コンソールでの作業）

```
> a       ←変数 a にアクセス
Uncaught ReferenceError: a is not defined  ←できない
    at <anonymous>:1:1

> f1(3,4)   ←関数 f1 にアクセス
Uncaught ReferenceError: f1 is not defined ←できない
    at <anonymous>:1:1
```

4.13.2.1 モジュール内のオブジェクトを公開する方法 (1)：DOM を介する

モジュールから DOM へのアクセスができるので「4.2.1.1 グローバル変数の所在」(p.124)で解説した方法によって、モジュール内のオブジェクトの値を window オブジェクトのプロパティに設定し、グローバル変数として公開することができる。ただし、不用意なグローバル変数の使用は、名前の衝突の原因となるので十分に注意すること。

4.13.2.2 モジュール内のオブジェクトを公開する方法 (2)：export で公開する

export 文の記述により、モジュール内のオブジェクトを他のモジュールに対して公開することができ、他のモジュールはそれらを import 文で受け取ることができる。

オブジェクトの公開 : export { オブジェクトの並び }
オブジェクトの受取り : import { オブジェクトの並び } from 提供側モジュールのパス

「オブジェクトの並び」はコンマで区切って記述する。また、関数や変数の宣言時に接頭辞の様に export, import を記述することもできる。import でオブジェクトを受け取る際には「提供側モジュールのパス」を指定する。このとき、

パスの記述の先頭には、絶対パス／相対パスを表す記述（"/", "./", "../"など）を必ず付けること。

export, import によるモジュール間でのオブジェクトの公開と受取りについて、次のサンプル moduleTest02.js, moduleTest02-2.js で例を示す。

記述例：moduleTest02.js

```
1 import {c,b,a} from "../moduleTest02-2.js";
2 console.log("a =",a);
3 console.log("b =",b);
4 console.log("c =",c);
```

記述例：moduleTest02-2.js

```
1 const a = 1, c = 3, d = 4;
2 export const b = 2;
3 export {a,c};
```

この例では右側のスクリプト moduleTest02-2.js が公開するオブジェクト a, b, c を左側のスクリプト moduleTest02.js が受け取る。またこの例からわかるように、オブジェクトを export する順序, import する順序は意味を持たず、オブジェクトの名前に基づいた受け渡しとなる。

記述例：moduleTest02.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>moduleTest02</title>
6   <script src="moduleTest02.js"
7     type="module"></script>
8 </head>
9 <body></body>
10 </html>
```

上のモジュールスクリプトを読み込んで実行する HTML を左に示す。これを Web ブラウザで表示すると、ブラウザのコンソールに次のように出力される。

コンソールの表示：

```
a = 1
b = 2
c = 3
```

4.13.2.3 モジュール毎の名前空間を実現するための工夫

モジュールスクリプトの形式でシステムを構築する際、各モジュールスクリプトを代表する記号を定め、当該モジュールが公開するオブジェクトを全てその記号（名前空間）の配下で統括するという方法がよく採用される。すなわち、公開するオブジェクト全てを

モジュール名. 公開オブジェクト名

という形式で「モジュール名」のオブジェクトのプロパティとして扱うという方法である。これは簡単に実現できる方法であり、当該モジュールスクリプトの冒頭部分で、

export const モジュール名 = {};

と記述しておき、公開用の関数は、

モジュール名. 関数名 = function(引数並び) { プログラム };

として実装する。同様に、公開用の変数は、

モジュール名. 変数名 = 値;

として実装する。そして「モジュール名」を export すると、受取り側のモジュールでは提供側のモジュールと同様に

モジュール名. 公開オブジェクト名

の形式で公開されたオブジェクトが使用できる。

以上のような形で名前空間を実現する例を次のサンプル moduleTest03.html, moduleTest03.js, nsTest03_1.js, nsTest03_2.js で示す。

記述例：moduleTest03.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>moduleTest03</title>
6   <script src="moduleTest03.js" type="module"></script>
7 </head>
8 <body></body>
9 </html>
```

左の HTML が読み込む
モジュールファイル
moduleTest03.js
を次に示す。

記述例：moduleTest03.js

```
1 import {nsTest03_1} from "./nsTest03_1.js";
2 import {nsTest03_2} from "./nsTest03_2.js";
3 console.log( "nsTest03_1.a :", nsTest03_1.a );
4 console.log( "nsTest03_1.b :", nsTest03_1.b );
5 console.log( "nsTest03_2.a :", nsTest03_2.a );
6 console.log( "nsTest03_2.b :", nsTest03_2.b );
```

このモジュールファイルは更に次の 2 つのモジュール nsTest03_1.js. nsTest03_2.js を読み込み、それらモジュールから受け取ったオブジェクトの値をコンソールに出力する。

記述例：nsTest03_1.js

```
1 export const nsTest03_1 = {};
2 nsTest03_1.a = 11;
3 nsTest03_1.b = 12;
```

記述例：nsTest03_2.js

```
1 export const nsTest03_2 = {};
2 nsTest03_2.a = 21;
3 nsTest03_2.b = 22;
```

これら 2 つのモジュールはそれぞれ記号（名前空間） nsTest03_1, nsTest03_2 を持ち、それらにオブジェクト a, b がプロパティとして登録されている。Web ブラウザで moduleTest03.html を表示すると、ブラウザのコンソールに次の様に表示される。

コンソールの表示：

```
nsTest03_1.a : 11
nsTest03_1.b : 12
nsTest03_2.a : 21
nsTest03_2.b : 22
```

それぞれの名前空間に属するオブジェクト a, b は提供側のものと同じであることがわかる。

■ デフォルトエクスポート

上で示した例のように、只 1 つのオブジェクトを export する場合は default 宣言すると更に便利な扱いができる。すなわち、default 宣言で export されたオブジェクトは、受取り側で任意の記号に割り当てることができる。

オブジェクトの公開 : export default 公開オブジェクト

オブジェクトの受取り : import 名前 from 提供側モジュールのパス

「公開オブジェクト」（公開側）と「名前」（受取り側）は波括弧 { } で括らない。受取り側の「名前」は自由に与えることができ、提供側の「公開オブジェクト」を受取り側では異なる「名前」として扱うことができる。

default 宣言を応用した形で先の例を書き直した例を次の moduleTest04.html, moduleTest04.js, nsTest04_1.js, nsTest04_2.js で示す。

記述例：moduleTest04.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>moduleTest04</title>
6   <script src="moduleTest04.js" type="module"></script>
7 </head>
8 <body></body>
9 </html>
```

左の HTML が読み込む
モジュールファイル
moduleTest04.js
を次に示す。

記述例：moduleTest04.js

```
1 import NS1 from "./nsTest04_1.js";
2 import NS2 from "./nsTest04_2.js";
3 console.log( "NS1.a :", NS1.a );
4 console.log( "NS1.b :", NS1.b );
5 console.log( "NS2.a :", NS2.a );
6 console.log( "NS2.b :", NS2.b );
```

このモジュールファイルは更に次の 2 つのモジュール nsTest04_1.js. nsTest04_2.js （提供側）を読み込み、それらモジュールから受け取ったオブジェクトの値をコンソールに出力する。

提供側の公開オブジェクトは次に示すように nsTest04_1, nsTest04_2 であるが、上のスクリプト（受取り側）ではそれらを NS1, NS2 という名前で受け取っている。

記述例：nsTest04_1.js

```
1  const nsTest04_1 = {};  
2  nsTest04_1.a = 11;  
3  nsTest04_1.b = 12;  
4  export default nsTest04_1
```

記述例：nsTest04_2.js

```
1  const nsTest04_2 = {};  
2  nsTest04_2.a = 21;  
3  nsTest04_2.b = 22;  
4  export default nsTest04_2
```

moduleTest04.html を Web ブラウザで表示すると、コンソールに次のように表示される。

コンソールの表示：

```
NS1.a : 11  
NS1.b : 12  
NS2.a : 21  
NS2.b : 22
```

受取側で決めた名前 NS1, NS2 として提供側のオブジェクトが扱えることがわかる。

5 Web Workers

JavaScript のプログラミングモデルは基本的にはシングルスレッドであり、本当の意味で複数のスレッドを並行して実行することはできない。従って、時間のかかる計算処理を別のスレッドとして並行実行するには Web Workers を使用する。

5.1 基本的な考え方

Web Workers ではワーカー (worker) と呼ばれるプログラムをメインスレッド (Web ブラウザで実行される JavaScript プログラムのスレッド) とは別に起動して実行する。このスレッドをワーカースレッドと呼ぶ。

ワーカーは通常、メインスレッドとなる JavaScript プログラムとは別のファイルなどのリソースとして作成される。本書ではこれをワーカースクリプトと呼ぶ。

ワーカーは Worker クラスのオブジェクトとして扱う。Worker オブジェクトのコンストラクタは次のようにして呼び出す。

書き方: `new Worker(ワーカースクリプトの URL)`

引数に与えた URL (ファイルのパスでも良い) からワーカースクリプトを読み込み、メインスレッドとは別に独自のワーカースレッドとして起動する。

ワーカーは複数起動することが可能である。また、Worker オブジェクトの作成 (ワーカーの起動) はメインスレッドだけでなく、ワーカースレッド内でも可能である。すなわち、ワーカーが別のワーカーを起動⁹⁵ することも可能である。本書では、ワーカーのネストにおいて Worker オブジェクト作成側を親スレッド、作成されたワーカースレッド側を子スレッドと呼ぶことにする。

ワーカーは独自のメモリ空間を持ち、メインスレッドや他のワーカースレッドとはオブジェクトの共有ができない。スレッド同士がデータをやり取りするには、文字列のメッセージの送受信を応用する。これについては後で説明する。また当然であるが、DOM にアクセスできるのはメインスレッドのみであり、ワーカースレッドは DOM にアクセスすることができない点に注意すること。また、ワーカースクリプト側では HTMLElement クラスのオブジェクトも扱うことができない。

5.1.1 ワーカーのプログラミングの考え方

メインスレッド、ワーカースレッド共に、基本的にイベントループを実行するものであり、親子の両スレッドは共に相手からのメッセージの受信をイベントとして待機している。すなわち、ワーカーを用いたプログラミングは、相手スレッドからのメッセージ受信を起点とするイベントハンドリングの形となる。

Worker オブジェクトを作成した親スレッドでは、ワーカーはその Worker オブジェクトであり、それに対して `postMessage` メソッドを実行することでワーカーにメッセージを送信することができる。ワーカースレッド側でも同様のメソッドで親スレッドに向けてメッセージを送信する。

ワーカースクリプトでは `self` というオブジェクト (`DedicatedWorkerGlobalScope` クラスのオブジェクト) を介して親スレッドとの通信を行う。すなわち、`self` オブジェクトにメッセージ受信のイベントハンドリングを登録して親スレッドからのメッセージ受信の処理を行う。また、`self` オブジェクトに対して `postMessage` メソッドを実行して親スレッド側にメッセージを送信する。

書き方: `Worker オブジェクト.postMessage(メッセージ)` (親スレッド→子スレッド)
`self.postMessage(メッセージ)` (子スレッド→親スレッド)

「メッセージ」を相手スレッドに送信する。「メッセージ」として文字列以外にも、配列や Object を与えることができる⁹⁶。戻り値は無い。(undefined)

子スレッドからのメッセージ送信は、親スレッド側 Worker オブジェクトの `message` イベントとしてハンドリングする。また、親スレッドからのメッセージ送信は、子スレッド側 `self` オブジェクトの `message` イベントとしてハンドリングする。(図 38)

⁹⁵これを「ワーカーのネスト」(入れ子)と表現する。

⁹⁶関数やプロトタイプチェーンなどはデータとして送信できないので注意すること。

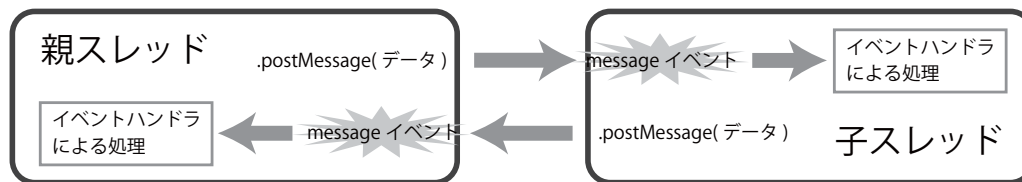


図 38: 親スレッドと子スレッドの連携の概略

■ サンプル実装を用いた解説

サンプル WebWorker00.html は同一ディレクトリ下のワーカースクリプト Worker00.js を起動するものである。

記述例：WebWorker00.html

```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>WebWorker00</title>
6    <style>
7      input[type="text"] { border: solid 1px #777777; }
8    </style>
9    <script>
10     let w;
11     function f1(e) {                                // ワーカーからのメッセージ受信後の処理
12       t2.value = e.data;
13     }
14     function f2(e) {                                // 送信ボタンクリック時の処理
15       w.postMessage(t1.value);                      // ワーカーにメッセージ送信
16     }
17     function f0() {
18       w = new Worker("Worker00.js");                // ワーカーを作成
19       w.addEventListener("message",f1);              // ワーカーにイベント登録（受信待機）
20       b1.addEventListener("click",f2);
21     }
22     window.addEventListener("load",f0);
23   </script>
24 </head>
25 <body>
26   <input type="text" value="" id="t1">
27   <input type="button" value="送信" id="b1"><br>
28   <input type="text" value="" id="t2">
29 </body>
30 </html>

```

ワーカースクリプト：Worker00.js

```

1  self.onmessage = function(e) { // 親スレッドからのメッセージ受信時の処理
2    let msg = "Worker received:" + e.data;
3    self.postMessage( msg );    // 親スレッドにメッセージ送信
4  };

```

※ 1 行目の self.onmessage の self は省略可能である。

WebWorker00.html を Web ブラウザで表示すると図 39 の (a) の様に表示される。

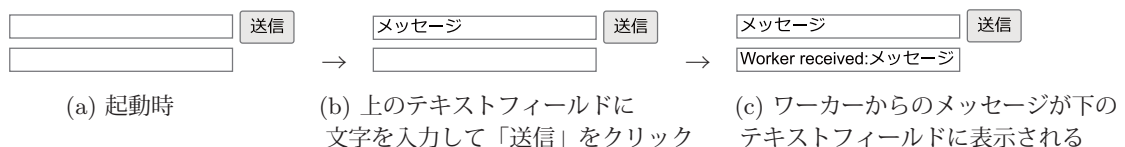


図 39: WebWorker00.html の動作

このサンプルコンテンツは上下2つのテキストフィールドを持つ。上のフィールドに文字を入力して **送信** ボタンをクリックすると、そのフィールドの内容がワーカーに送信される（図 39 の (b)）。それを受信したワーカーは受け取ったメッセージの先頭に「Worker received:」を付けたものをメインスレッド側に送信する。そのメッセージを受け取ったメインスレッドの処理により図 39 の (c) のような表示となる。

以上の動作から、メインスレッドとワーカースレッドがメッセージを交換している様子がわかる。

注意 ワーカースクリプトの実行について

WebWorker00.html を Web ブラウザで表示する場合、セキュリティの事情から http プロトコルで開くこと。
(例. `http://127.0.0.1:8080/WebWorker00.html`)
これには Web サーバが必要となる。ローカルの計算機環境で Web サーバを実現する方法に関しては巻末付録
「A.4 ローカルの計算機環境で Web サーバを起動する簡易な方法」(p.293)を参照のこと。

注意 ワーカースクリプトのキャッシュについて

ワーカースクリプトに加えた変更は、Web ブラウザのページ更新ボタンだけでは即座に有効にならないことがある。
その場合の対処法については「B.1 スクリプトのキャッシュの抑止」(p.306)を参照のこと。

次に、メインスレッドとワーカースレッドが同時に並行して動作することを示す。

5.2 ワーカーの並行実行

メインスレッドとワーカースレッドが互いにブロックせずに並行して動作することを示す。まず準備として、次のような関数 `f1` を考える。

例. 処理に時間のかかる関数 `f1`

```
> function f1( msg ) {      Enter      ←関数定義の開始
  let n = 0;                Enter
  while ( true ) {          Enter
    if ( Math.random() < 2**(-27) ) {      Enter
      console.log("hit:",msg,++n);        Enter
      if ( n >= 4 ) break;                Enter
    }                                     Enter
  }                                     Enter
  return msg;                    Enter
}                                Enter      ←関数定義の終了
undefined                       ←関数定義の処理結果
```

この関数 `f1` は乱数を立て続けて生成し、極小の値 (2^{-27} 未満の値) の乱数が得られたときにコンソールに「hit:」で始まる文字列を表示する。また極小の値が 4 回出た時点で実行を終了する。

この関数において極小の乱数が出る確率は極めて小さく、そのような値が 4 回出るまでには相当の時間を要するだけでなく、関数の実行が終了するまで当該スレッドはこの処理によってブロックされる。

この関数 `f1` を Web ブラウザのコンソールで実行する例を次に示す。

例. `f1` の実行 (先の例の続き)

```
> f1("job-1"); f1("job-2")      Enter      ← f1 を 2 つ連続して実行
hit: job-1 1                    ← 1 つ目の f1 の実行 (ここから)
hit: job-1 2
hit: job-1 3
hit: job-1 4                    ← 1 つ目の f1 の実行 (ここまで)
hit: job-2 1                    ← 2 つ目の f1 の実行 (ここから)
hit: job-2 2
hit: job-2 3
hit: job-2 4                    ← 2 つ目の f1 の実行 (ここまで)
'job-2'                         ← 2 つ目の f1 の戻り値
```

これは関数 `f1` を 2 つ起動する例である。実行の結果から、1 つ目の `f1` の実行が完了した後に 2 つ目の `f1` が起動されることがわかる。しかも、`f1` の実行には数秒の時間を要する⁹⁷ ことがわかる。また、上記の例の実行が完了するまで Web ブラウザの当該ページのスレッドがブロックされる。

次に、この関数 `f1` をワーカースレッドで実行すると他のスレッドがブロックされないことを、サンプル Web-Worker01.html, Worker01.js を用いて示す。

⁹⁷ 使用する計算機環境により処理時間が異なるので、極小値の判定を適宜変更して試されたい。

記述例：WebWorker01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>WebWorker01</title>
6   <style>
7     input[type="text"] {
8       width: 320px;
9       border: solid 1px #777777;
10    }
11  </style>
12  <script>
13    function f2(e) {
14      t1.value = new Date();
15    }
16    function f1(e) {    // ワーカーからのメッセージ受信後の処理
17      console.log(e.data,"finished.");
18    }
19    function f0() {
20      // ワーカーを2つ作成してイベントハンドリングを登録
21      let w1 = new Worker("Worker01.js");    let w2 = new Worker("Worker01.js");
22      w1.addEventListener("message",f1);    w2.addEventListener("message",f1);
23      // 2つのワーカーにメッセージを送信（ワーカー側関数 f1 の起動）
24      w1.postMessage("job-1");                w2.postMessage("job-2");
25      // メインスレッドの時計表示
26      let t = setInterval(f2,10);
27    }
28    window.addEventListener("load",f0);
29  </script>
30 </head>
31 <body><input type="text" id="t1"></body>
32 </html>
```

ワーカースクリプト：Worker01.js

```
1 function f1( msg ) {
2   let n = 0;
3   while ( true ) {    // 極小の値が4回出るまで繰り返す
4     if ( Math.random() < 2*(-27) ) {    // 極小の値が出たら
5       console.log("hit:",msg,++n);    // コンソールにメッセージ表示
6       if ( n >= 4 ) break;            // 4回目なら終了
7     }
8   }
9   return msg;
10 }
11
12 self.onmessage = function(e) {    // 親スレッドからのメッセージ受信時の処理
13   let msg = f1( e.data );    // 関数 f1 を起動して終了後に
14   self.postMessage( msg );    // 親スレッドにメッセージを送信
15 };
```

WebWorker01.html を Web ブラウザで表示すると図 40 の様に表示される。
この際、p.182 の**注意**）を踏まえること。

Fri Mar 22 2024 14:04:26 GMT+0900 (日本標準時)

日時の表示が時々刻々と変化する。
図 40: WebWorker01.html の表示

このサンプルでは、関数 f1 が2つのワーカーとしてほぼ同時に起動される。そして、メインスレッドによる日時の変化と、2つのワーカーの動作が互いにブロックされことなく並行実行されることが Web ブラウザのコンソールで確認できる。

Web ブラウザのコンソールの表示：

```
hit: job-1 1      ← 2つのワーカーが同時に並行して実行され、
hit: job-1 2
hit: job-2 1      それらのメッセージが入り混じっている
hit: job-2 2
hit: job-1 3      図 40 の時計表示も同時に進捗する
hit: job-2 3
hit: job-2 4
job-2 finished.   ← 2番目のワーカーの関数 f1 が終了した
hit: job-1 4
job-1 finished.   ← 1番目のワーカーの関数 f1 が終了した
```

5.3 メッセージ送信におけるデータの複製と移転

postMessage メソッドによってスレッド間でデータを送信する場合、基本的には送信元のデータの複製を相手スレッドに送信する。このとき、**移転**⁹⁸（transfer）する形でデータを送信すると、当該データの所有権は元のスレッドから相手スレッドに移り、元のスレッドでは使用できなくなる。すなわち、元のスレッドからそのデータはアクセスできなくなり、相手スレッドのものとなる。

postMessage メソッドで移転できるオブジェクトは表 50 に挙げるようなクラスのものであり、それらは **Transferable**（移転可能）であると表現する。

表 50: Transferable なクラス（一部）

ArrayBuffer	ImageBitmap	AudioData	VideoFrame	OffscreenCanvas
MessagePort	ReadableStream	WritableStream	TransformStream	RTCDDataChannel

特にサイズの大きなデータを相手スレッドに送信する場合、移転する形を取ることによって複製にかかる時間や消費する記憶資源の問題が解決できる。相手スレッドにオブジェクトを移転する場合は次のような形式で postMessage メソッドを記述する。

書き方： 対象.postMessage(データ, [移転対象オブジェクトの配列])

「データ」を送信する際、それに含まれる「移転対象オブジェクト」を相手スレッドに移転する。

型付き配列（ArrayBuffer）を相手スレッドに移転する処理を、次の WebWorker02.html, Worker02.js で例示する。

記述例：WebWorker02.html

```
1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>WebWorker02</title>
6    <script>
7      function f0() {
8        let w = new Worker("Worker02.js");
9        let d1 = "文字列データ";           // 文字列
10       let d2 = new Int8Array([1,2,3,4,5,6,7,8,9,10]); // 型付き配列
11       console.log( "[親] 送信前の d1 の長さ:", d1.length );
12       console.log( "[親] 送信前の d2 の長さ:", d2.length );
13       w.postMessage(d1);
14       w.postMessage(d2.buffer,[d2.buffer]);
15       console.log( "[親] 送信後の d1 の長さ:", d1.length );
16       console.log( "[親] 送信後の d2 の長さ:", d2.length );
17     }
18     window.addEventListener("load",f0);
19   </script>
20 </head>
21 <body></body>
22 </html>
```

⁹⁸移譲と表現する場合もある。

ワーカースクリプト：Worker02.js

```
1 self.onmessage = function(e) { // 親スレッドからのメッセージ受信時の処理
2   let dat = e.data;
3   let tp = dat.constructor.name;
4   console.log( "[子] 型 :", tp );
5   if ( tp == "ArrayBuffer" ) {
6     let sum = 0;
7     for ( n of new Int8Array(dat) ) { sum += n; }
8     console.log( "[子] 長さ:", dat.byteLength, ", 合計:", sum );
9   } else {
10    console.log( "[子] 長さ:", dat.length, ", 内容:", dat );
11  }
12 };
```

WebWorker02.html では、相手スレッド Worker02.js に文字列のデータと ArrayBuffer のオブジェクトを送信する。また、ワーカースレッド Worker02.js 側では、受け取ったデータの型によって表示方法を変える処理を行っている。また、親スレッド側では送信の前後でデータの長さがどの様に変化するかを表示し、移転したデータは親スレッド側では使用できなくなっている様子を示す。

WebWorker02.html を Web ブラウザで表示すると、ブラウザのコンソールには次の様に表示される。

Web ブラウザのコンソールの表示：

[親] 送信前の d1 の長さ： 6	← 6 文字の文字列をワーカーに送信
[親] 送信前の d2 の長さ： 10	← 要素数 10 の配列をワーカーに送信
[親] 送信後の d1 の長さ： 6	← 文字列は送信後も変化なし
[親] 送信後の d2 の長さ： 0	← Transferable なオブジェクトは内容が失われる
[子] 型 : String	
[子] 長さ： 6 , 内容： 文字列データ	← 子スレッドが受信した文字列
[子] 型 : ArrayBuffer	
[子] 長さ： 10 , 合計： 55	← 子スレッドが配列を受信して合計を算出

[親] の表示は親スレッド（メインスレッド）、[子] の表示は子スレッド（ワーカースレッド）によるものである。

文字列データを送信する際は複製して、Int8Array 内の ArrayBuffer（buffer プロパティ）を送信する場合は移転によってデータを送信している。結果として、型付き配列（d2 のオブジェクト）は送信後にサイズが 0 になっていることがわかる。このデータはワーカースレッド側のものになっている。

6 通信のための機能

6.1 Fetch API

Fetch API は Web サーバとの通信のための機能を提供する。具体的には、HTTP サーバに対して http, https のプロトコルでリクエストを送信し、応答を受診するための機能を提供する。クライアント側の JavaScript プログラムと Web サーバとの間の通信には待ち時間が発生するので、リクエストの送信処理とサーバからの応答の受信処理は Promise クラスのオブジェクトを用いた**非同期処理**の形を取る。

6.1.1 基本的な通信の手順

Fetch API では、fetch 関数でリクエストを Web サーバに送信する。

書き方： `fetch(リクエスト)`

「リクエスト」にはサーバの URL（対象の HTML リソースや CGI プログラムなどの URL）を表した文字列、あるいは Request オブジェクトを与える。この関数を実行するとリクエストが送信され、それに対する応答を待機するための Promise オブジェクトが返される。その後は戻り値の Promise オブジェクトを用いたサーバからの応答の解析を行う。更にその後、応答内容のデータを取得する処理を行う。

6.1.2 Promise による非同期処理

fetch 関数が実行されてリクエストが Web サーバに送信された後、Web サーバは応答処理を行い、その結果をクライアントに返送する。この間、通信とサーバ側の処理に多少の時間がかかり、fetch 関数の呼び出しが終わった時点ではまだサーバからの応答は得られていない。すなわち、fetch 関数の戻り値である Promise オブジェクトはまだ**待機状態**（pending）にあり、サーバからの応答を待っている。また、Promise オブジェクトが待機状態である間も JavaScript のプログラムは**ブロック**されることなく、fetch 関数呼び出しの次のステップに進む。

Promise オブジェクトは、それが監視している処理が正常に終了すると**完了状態**⁹⁹（fulfilled）となり、処理結果に関する解析処理が可能となる。処理結果を解析する処理は当該 Promise オブジェクトに対する then メソッドで登録しておく。すなわち、fetch 関数の呼び出し後も JavaScript のイベント処理が Promise オブジェクトの状態を監視しており、それが完了の状態になった時点で then で登録された関数が呼び出される。

例. fetch 関数実行後の応答の監視

```
var p1 = fetch( リクエスト );
var p2 = p1.then( 関数 1 );
    (次の処理)
```

ここに例示した「関数 1」は Web サーバからの応答が完了したことを受けて起動するもので、応答の内容を解析するための処理を行う。この例のような処理はブロックされることなく進み、Promise オブジェクト p1 が完了状態になった時点で「関数 1」が起動される。

then メソッドもその処理結果として Promise オブジェクト p2 を返し、「関数 1」の処理が完了するのを待つ。当然、p2 も当初は pending の状態になり、プログラムの実行は直ちに「次の処理」に進む。p2 に対しても p1 の場合と同様に then メソッドを実行することになり、その段階では応答内容のデータを取り出す処理を行う。（次の例）

例. 応答内容の解析（先の例の続き）

```
var p3 = p2.then( 関数 2 );
p3.catch( 例外処理 );
    (次の処理)
```

ここに示した「関数 2」は先の例の「関数 1」の実行完了（p2 が完了状態になったこと）を受けて起動するもので、応答内容のデータを取り出す処理を行う。この例のような処理はブロックされることなく進み「次の処理」進む。この例の then メソッドの戻り値 p3 も Promise オブジェクトである。

⁹⁹ 「履行の状態」と表現されることもある。

Promise が待機している処理が失敗する（拒否される）場合もあり，その場合は当該 Promise オブジェクトは**拒否状態**（rejected）となる．拒否状態を受けて起動する処理は catch メソッドで登録しておく．上の例では「関数 2」の実行が失敗した場合の処理を「例外処理」として catch メソッドで登録している．また，拒否状態に対して then メソッドが返す Promise オブジェクトも拒否状態となるので，上記 2 つの例を通して，p1, p2, p3 の拒否状態は次々と伝搬し，最後の catch でハンドリングされることになる．

以上のことから，fetch 関数実行後の処理（応答解析，応答内容の取得，エラー処理）を次のような**メソッドチェーン**の形で実装することができる．

```
fetch( リクエスト ).then( 関数 1 ).then( 関数 2 ).catch( 例外処理 )
```

サーバに「リクエスト」を送信し，応答解析を「関数 1」で，応答内容の取得を「関数 2」で実行し，各段階でエラーが発生した場合の処理を「例外処理」でハンドリングする．これら一連の処理は JavaScript 処理系のイベント処理として管理され，プログラムの実行をブロックしない．

6.1.3 サーバからの応答の解析

先の例における「関数 1」は，Response オブジェクトの内容を解析する関数として実装する．具体的には，第 1 引数に Response オブジェクトを受け取り，それを解析した結果の Promise オブジェクトを返す形の関数として実装する．

```
書き方の例 1： function( res ) { 解析結果の Promise を返す処理 }
```

```
書き方の例 2： res => 解析結果の Promise を返す処理
```

「res」には Response オブジェクトが与えられる．このような関数の内部では，応答のメディアタイプ（MIME タイプ）を調べて，各種の解析用のメソッドを実行し，その結果を戻り値として返す．

サーバからの応答のメディアタイプは Response オブジェクト res の headers に対して

```
res.headers.get("Content-Type")
```

とすることで文字列の形で得られる．この値から判断して表 51 に挙げる解析用のメソッドの内 1 つを実行してその値を返す．

表 51: Response オブジェクト res 解析用のメソッド

res. メソッド	解説
res.text()	応答本体をテキストとして解析する．
res.json()	応答本体を JSON として解析する．
res.formData()	応答本体を FormData オブジェクトとして解析する．
res.blob()	応答本体を Blob（バイナリデータ）として解析する．
res.arrayBuffer()	応答本体を ArrayBuffer（バイト配列）として解析する．

6.1.4 応答からのデータ本体の取り出し

先の例における「関数 2」は，サーバからの応答内容のデータ本体を第 1 引数に受け取る関数として実装する．

```
書き方の例 1： function( dat ) { dat を使用する処理 }
```

```
書き方の例 2： dat => dat を使用する処理
```

「dat」にはサーバからの応答内容のデータ本体が与えられる．

6.1.5 例外処理

Promise オブジェクトの catch メソッドの引数に与える例外処理は，**エラーオブジェクト**¹⁰⁰（Error クラス）を第 1 引数に受け取る関数として実装する．

```
書き方の例 1： function( err ) { err を使用する処理 }
```

```
書き方の例 2： err => err を使用する処理
```

「err」にはエラーオブジェクトが与えられる．

¹⁰⁰ 「3.15.1 エラーオブジェクト」（p.116）参照のこと．

6.1.6 Fetch API の使用例

Web サーバのコンテンツディレクトリにある次のようなリソースを取得するシステムの例を示す。

記述例：FetchTest01.txt

```
1 This is a sample of text data.
2 This contains 3 lines.
3 Hello, World.
```

記述例：FetchTest01.json

```
1 {"a":1,
2  "b":2,
3  "c":3}
```

これらが `http://localhost/` にある場合にそれらを Fetch API で取得するサンプル `FetchTest01.html` を示す¹⁰¹。

記述例：FetchTest01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>FetchTest01</title>
6   <style>
7     #t1, #t2, #t3, #t4 { width: 260px; border: solid 1px gray; }
8     #t3 {
9       height: 120px;
10      overflow: auto;      white-space: pre;
11    }
12  </style>
13  <script>
14    var ct;
15    // JSONデータの整形表示
16    function f2(dat) {
17      let k, tx="キー\t値\n-----\n";
18      const ks = Object.keys(dat).sort();
19      for ( k of ks ) {
20        tx += k + "\t" + String(dat[k]) + "\n";
21      }
22      t3.value = tx;
23    }
24    // リクエスト, 解析, 内容表示
25    function f1() {
26      const u = t1.value;
27      t2.value = t3.value = t4.value = "";
28      fetch(u).then( // リクエストの発行
29        res => { // 応答の解析
30          ct = res.headers.get("Content-Type");
31          t2.value = ct;
32          if ( ct=="application/json" ) {
33            return res.json(); // JSONの解析
34          } else {
35            return res.text(); // テキストの解析
36          }
37        }
38      ).then( // 内容表示
39        dat => {
40          if ( ct=="application/json" ) {
41            f2(dat); // JSONの内容を表示
42          } else {
43            t3.value = dat; // テキストを表示
44          }
45          t4.value = "正常終了";
46        }
47      ).catch( // 例外 (エラー) 処理
48        er => {
49          t4.value = "異常終了";
50        }
51      );
52    }
53  </script>
54 </head>
55 <body>
56   <input type="text" value="http://localhost/FetchTest01.txt" id="t1"><br>
```

¹⁰¹同一オリジンポリシーに従って、この HTML ファイルも `http://localhost/` にあるとする。


```

57 <input type="button" value="リクエスト" id="b1" onClick="f1()"><br>
58 Content-Type:<br>
59 <input type="text" id="t2"><br>
60 Data:<br>
61 <textarea id="t3" wrap="off"></textarea><br>
62 Message:<br>
63 <input type="text" id="t4">
64 </body>
65 </html>

```

これを Web ブラウザで表示すると右のような表示となる。

最上部のテキストフィールドに取得するリソースの URL を入力し、**リクエスト** ボタンをクリックすると、「Content-Type」の欄にリソースのメディアタイプが、「Data」の欄にコンテンツの内容が表示される。

リソースの取得が正常に終了すると「Message」欄に「正常終了」、取得が失敗すると「異常終了」と表示される。

リクエスト ボタンをクリックした際の実行例を次に示す。

例. http://localhost/FetchTest01.txt の要求

例. http://localhost/FetchTest01.json の要求

7 グラフィックスの作成 (1) : canvas 要素

HTML の canvas 要素には JavaScript のプログラムで線画や画像を描画することができる。canvas 要素はビットマップの描画領域、すなわち画素（ピクセル:pixel）の格子配列として画像を構成するもの（ラスタグラフィックス）である。従って、座標やサイズの単位はピクセル（px）である。

canvas 要素に描画する線画図形は基本的に輪郭線や、それが囲む内部の領域といったものである。（図 41）

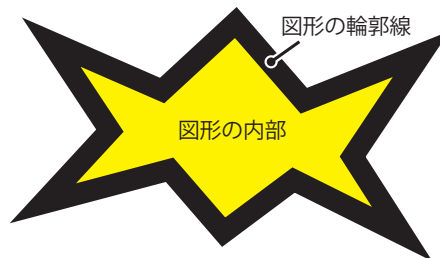


図 41: 線画図形の基本構造

本書では以降の解説の中で、この「図形の内部」のことを塗りつぶしもしくはフィルと呼ぶ。また、輪郭線のことをストロークと呼ぶことがある。輪郭線とフィルは別の実体として描画される。

本書では、canvas グラフィックスの最も基本的な事柄について解説する。

7.1 座標と角度の考え方

canvas 要素における座標は左上の角を原点とし、横を x 、縦を y とする直交座標系であり、右に行くほど x の値は大きくなり、下に行くほど y の値は大きくなる。また、回転の角度は図 42 に示すように、時計回り（右回り）を正とし、単位は弧度（ラジアン:rad）である。図形描画に関する傾きは水平を 0rad とする。

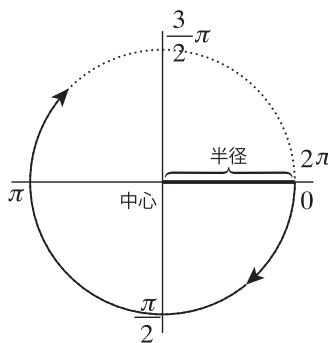


図 42: canvas における角度の考え方

7.2 描画領域のサイズ

canvas 要素の描画領域のサイズは HTML タグ内に記述することができる。

書き方： `<canvas width=横幅 height=高さ> 代替コンテンツ </canvas>`

描画領域のピクセル構成を「横幅」×「高さ」に設定する。また、JavaScript のプログラムで canvas 要素の描画領域サイズを動的に変更することもできる。例えば canvas 要素 cvs がある場合、

```
cvs.width = 横幅;  
cvs.height = 高さ;
```

などと記述することができる。

注意)

canvas 要素に対する CSS の属性 width, height はブラウザ上での表示サイズを設定するものであり、表示領域のピクセルサイズを設定するものではない。すなわち、CSS の width, height 属性の設定は、canvas の描画領域の表示をそのサイズに拡張する。

7.3 描画コンテキスト

canvas 要素に対する描画には**描画コンテキスト**（グラフィックコンテキストまたは単にコンテキストと呼ぶこともある）を用いる。描画コンテキストは canvas 要素から `getContext` メソッドで取得する。

書き方： `canvas オブジェクト.getContext("2d")`

引数に与える "2d" は 2 次元の描画のための描画コンテキストを取得することを指定するものである。例えば canvas オブジェクト `cvs` がある場合、その描画コンテキストを取得するには、

```
ctx = cvs.getContext("2d");
```

などと記述する。これにより `cvs` の描画コンテキストが `ctx` に得られる。

`getContext` の引数にはこの他にも、3 次元描画のための "webgl", "webgl2" といったものも指定できるが本書では 2 次元描画に限って解説する。

7.3.1 フィルに関する描画の設定

描画コンテキストの `fillStyle` プロパティに色やグラデーション、画像のテクスチャを与えることで、フィルの設定ができる。

書き方： `描画コンテキスト.fillStyle = フィルの設定`

「描画コンテキスト」のフィルに「フィルの設定」を与える。フィルの色を設定する場合は色名や `rgb` 関数表記の文字列を与える。

次の例は描画コンテキスト `ctx` に色を設定するものである。

例. 赤を設定

```
ctx.fillStyle = "red";
```

例. 青を設定

```
ctx.fillStyle = "rgb(0,0,255)";
```

7.3.2 輪郭線に関する描画の設定

描く輪郭線の線幅（線の太さ）は描画コンテキストの `lineWidth` プロパティに設定する。また、`strokeStyle` プロパティに色やグラデーション、画像のテクスチャを与えることで、輪郭線の設定ができる。これに関しては先の `fillStyle` の設定の場合と同様の扱いができる。

7.3.2.1 線幅（線の太さ）と始点、終点の座標の考え方

canvas 要素内での輪郭線の線幅と位置の考え方の概略を図 43 に示す。これは、HTML 要素に対して CSS の属性で位置やサイズを設定する場合のものとは異なることに注意しなければならない。

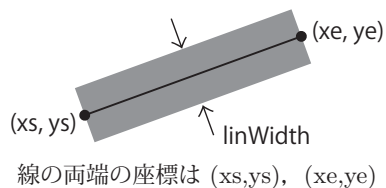


図 43: 線の幅と両端の座標の関係

7.3.2.2 線の端の形状の設定

描画コンテキストの `lineCap` プロパティに設定する値によって、描画される線の端の形状を変えることができる。

書き方： `描画コンテキスト.lineCap = 形状`

「形状」として "butt", "round", "square" を指定することで線の端が図 44 のように変わる。("butt" がデフォルト)

7.3.2.3 線の結合部の形状の設定

描画コンテキストの `lineJoin` プロパティに設定する値によって、描画される線の結合部の形状を変えることができる。

書き方： `描画コンテキスト.lineJoin = 形状`

「形状」として "bevel", "round", "miter" を指定することで線の結合部が図 45 のように変わる。("miter" がデフォ

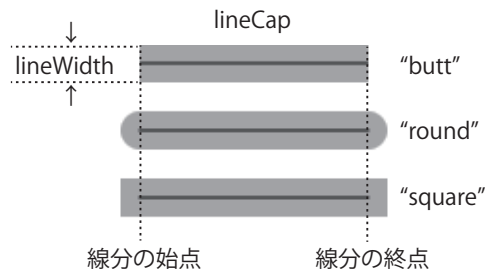


図 44: 線の端の形状

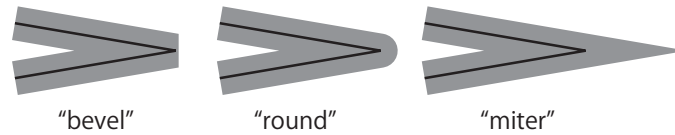


図 45: 線の結合部の形状

ルト)

`lineJoin` のデフォルト値は "miter" (留め継ぎ) であるが、結合部の線同士の角度が鋭い場合は結合部の突出が大きくなる。この突出が大きくなり過ぎると、結合部が自動的に "bevel" となる。

■ miter の制限について

"miter" の結合における突出の長さの制限には、描画コンテキストの `miterLimit` プロパティが関係する。すなわち、突出部の長さが線幅 `lineWidth` の半分と `miterLimit` の積よりも大きくなると結合部が "bevel" となる。(図 46 参照) `miterLimit` のデフォルト値は 10.0 である。

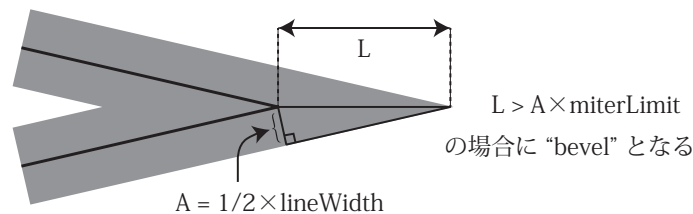


図 46: miter の突出の制限

7.3.2.4 破線の設定

描画コンテキストに `setLineDash` メソッドを実行することで輪郭線を破線に設定することができる。

書き方： 描画コンテキスト.`setLineDash`(配列)

「配列」に線の長さと空白の長さを意味する数値を並べたものを与える。例えば、描画コンテキスト `ctx` に、

```
ctx.setLineDash([20,4,5,4]);
```

と実行して線を描画すると図 47 のような破線となる。



図 47: 破線の例

図 47 のような破線を描く HTML 文書全体を `Canvas04.html` に示す。

記述例：`Canvas04.html`

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>Canvas04</title>
6   <script>
7     let cvs, ctx;
8     function f1() {
9       ctx.strokeStyle = "rgb(0,0,0)";
10      ctx.lineWidth = 5;
11      ctx.setLineDash([20,4,5,4]);
```

```

12         ctx.beginPath();
13         ctx.moveTo(10,10);
14         ctx.lineTo(190,10);
15         ctx.stroke();
16     }
17     function f0() {
18         cvs = document.getElementById("cvs");    // canvas要素の取得
19         cvs.width = 200;                        // canvas要素のwidth属性を設定
20         cvs.height = 50;                        // canvas要素のheight属性を設定
21         ctx = cvs.getContext("2d"); // 描画コンテキストの取得
22         f1();    // 描画ルーチン
23     }
24     window.addEventListener("load",f0);
25 </script>
26 </head>
27 <body>
28     <canvas id="cvs"></canvas>
29 </body>
30 </html>

```

Canvas04.html の中に記述された beginPath, moveTo,.lineTo, stroke の各メソッドはパスを描くもので、これらについては後の「7.4.3 パスの描画」(p.194)で解説する。

7.4 描画メソッド

canvas 要素に図形や画像を描くには、描画コンテキストに対して各種のメソッドを実行する。ここでは、特に基本的な描画メソッドについて解説する。

7.4.1 矩形（四角形）の描画

輪郭線の矩形は strokeRect, フィルの矩形は fillRect を用いて描く。

書き方： 描画コンテキスト.strokeRect(x, y, w, h)

書き方： 描画コンテキスト.fillRect(x, y, w, h)

座標 (x,y) の位置に横幅 w, 高さ h の矩形を描く。戻り値はない。(undefined)

7.4.2 テキストの描画

フィルによるテキストの描画は fillText, 輪郭線によるテキスト（袋文字）の描画は strokeText で行う。

書き方： 描画コンテキスト.fillText(文字列, x, y)

書き方： 描画コンテキスト.strokeText(文字列, x, y)

座標 (x,y) の位置に「文字列」を描く。戻り値はない。(undefined)

テキストの位置の基準は図 48 に示すように、テキストを収めるボックス領域の左上である。



図 48: テキストの位置の基準

描画する文字のフォントに関することは font プロパティに設定する。

書き方： 描画コンテキスト.font = "太さ 大きさ フォント名"

このように、フォントに関する設定を記述した文字列を与える。例えば、描画コンテキスト ctx がある場合、

```
ctx.font = "bold 42px serif";
```

と記述するとテキストの描画が、太字、42 ピクセルの serif（明朝体）に設定される。

7.4.2.1 矩形とテキストを描画するサンプル

次の Canvas01.html は矩形とテキストを描画するサンプルである。

```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>Canvas01</title>
6    <script>
7      let cvs, ctx;
8      function f1() {
9        // 黄色いフィルの矩形
10       ctx.fillStyle = "yellow";
11       ctx.fillRect(15,10,270,150);
12       // 黒い輪郭線の矩形
13       ctx.lineWidth = 18
14       ctx.strokeStyle = "black";
15       ctx.strokeRect(15,10,270,150);
16       // 文字の描画（フィル）
17       ctx.fillStyle = "blue";
18       ctx.font = "bold 42px serif";
19       ctx.fillText("文字列描画",40,70);
20       // 文字の描画（輪郭線：袋文字）
21       ctx.lineWidth = 2;
22       ctx.strokeStyle = "red";
23       ctx.font = "bold 42px sans-serif";
24       ctx.strokeText("文字列描画",40,125);
25     }
26     function f0() {
27       cvs = document.getElementById("cvs"); // canvas要素の取得
28       cvs.width = 300; // canvas要素のwidth属性を設定
29       cvs.height = 170; // canvas要素のheight属性を設定
30       ctx = cvs.getContext("2d"); // 描画コンテキストの取得
31       f1(); // 描画ルーチン
32     }
33     window.addEventListener("load",f0);
34   </script>
35 </head>
36 <body>
37   <canvas id="cvs"></canvas>
38 </body>
39 </html>

```

これを Web ブラウザで表示すると、図 49 のようになる。



図 49: 矩形とテキストの描画

7.4.3 パスの描画

パスは折れ線や円弧を含んだ複雑な形状の輪郭線やフィルを描画するためのものである。パスの構築は `beginPath` メソッドの実行から始めて順次図形を構築する手順を踏み、最後に `stroke` や `fill` といったメソッドで輪郭線やフィルを完成する。

7.4.3.1 折れ線の描画

`beginPath`¹⁰² によってパスの描画が開始された状態で、`moveTo` メソッドを実行することで折れ線の始点を決定する。

書き方： 描画コンテキスト.`moveTo`(`x`, `y`)

座標 (`x`,`y`) を折線の始点とし、パスの現在の座標を (`x`,`y`) とする。戻り値はない (`undefined`)。

¹⁰²`beginPath` メソッドは値を返さない。 (`undefined`)

パスの現在の座標から指定した座標 (xn, yn) に向けて直線を描く (パスを拡張する) には lineTo メソッドを実行する。

書き方： 描画コンテキスト.lineTo(xn, yn)

lineTo の戻り値はない (undefined) 。その後、次々と lineTo メソッドを実行することで折れ線を構築する。

パスの現在の座標からパスの始点に向けて直線を描いてパスを閉じるには closePath メソッドを実行する。

書き方： 描画コンテキスト.closePath()

closePath の戻り値はない (undefined) 。このメソッドを実行していない状態では、パスは開いた状態 (始点と現在の座標が繋がっていない状態) となる。

現在の時点までに構築されたパスを輪郭線として描画するには stroke メソッドを、フィルとして描画するには fill メソッドを実行する。

書き方： 描画コンテキスト.stroke()

書き方： 描画コンテキスト.fill()

これらメソッドの戻り値はない (undefined) 。

以上のことを応用して折れ線のパスを描画するサンプル Canvas05.html を示す。

記述例： Canvas05.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>Canvas05</title>
6   <script>
7     let cvs, ctx;
8     function f1() {
9       ctx.fillStyle = "yellow";
10      ctx.strokeStyle = "black";
11      ctx.lineWidth = 5;
12      ctx.beginPath();
13      ctx.moveTo(10,10);
14      ctx.lineTo(190,30);
15      ctx.lineTo(10,50);
16      // ctx.closePath();           // (1)
17      // ctx.fill();                 // (2)
18      ctx.stroke();                 // (3)
19    }
20    function f0() {
21      cvs = document.getElementById("cvs"); // canvas要素の取得
22      cvs.width = 200;                     // canvas要素のwidth属性を設定
23      cvs.height = 60;                     // canvas要素のheight属性を設定
24      ctx = cvs.getContext("2d"); // 描画コンテキストの取得
25      f1();                                // 描画ルーチン
26    }
27    window.addEventListener("load",f0);
28  </script>
29 </head>
30 <body>
31   <canvas id="cvs"></canvas>
32 </body>
33 </html>
```

このサンプル中のコメント (1)~(3) の行を選択的に有効にして Web ブラウザで表示した例を図 50 に示す。



図 50: Canvas05.html でのパスの描画

■ 構築されるパスの範囲

パスは前回 `beginPath` が実行された時点から現在の時点までのものが1つのまとまりであり、次回 `beginPath` が実行されると、それ以降は新たなパスの取り扱いとなる。従って、1つのパスの構築中に `moveTo` を実行することで、連続しない複数の折れ線を1つのパスとして扱うことができる。例えば先のサンプル `Canvas05.html` で描画用の関数 `f1` を次のように変更する。

記述例：描画関数 `f1` の変更

```
1 function f1() {
2     ctx.fillStyle = "yellow";
3     ctx.strokeStyle = "black";
4     ctx.lineWidth = 5;
5     ctx.beginPath();
6     // 1つ目の折れ線
7     ctx.moveTo(10,10); ctx.lineTo(170,30); ctx.lineTo(10,50);
8     // 2つ目の折れ線
9     ctx.moveTo(210,10); ctx.lineTo(370,30); ctx.lineTo(210,50);
10    ctx.closePath(); // これは2つ目の折れ線に対して有効
11    ctx.fill(); ctx.stroke(); // これらは両方の折れ線に対して働く
12 }
```

この関数 `f1` では `beginPath` の後 `moveTo` が2つ実行されており、それぞれの位置から開始する2つの折れ線が存在する。また、`closePath` は2つ目の折れ線にのみ有効となる。ただし、2つの折れ線は1つのパスとして扱われ、`fill` と `stroke` は両方の折れ線に対して働く。関数 `f1` をこのように変更して Web ブラウザで表示したものを図 51 に示す。



図 51: 関数 `f1` の変更後の表示

7.4.3.2 円弧の描画

`arc` メソッドによって円弧を描くことができる。またこれを応用すると円を描くことができる。

書き方： 描画コンテキスト.`arc(x, y, 半径, 開始角, 終了角, 反時計回り)`

座標 (x,y) を中心とする「半径」の円弧を描く、描く円弧の範囲は「開始角」から「終了角」までで、「反時計回り」に `true` を与えると反時計回り、`false` を与えると時計回りに弧を描く。「反時計回り」の引数は省略可能で、デフォルトは `false` である。

`arc` メソッドで作成する円弧はパスの一部である。直線のパスと円弧を組み合わせて描画する例を `Canvas06.html` に示す。

記述例：`Canvas06.html`

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4     <meta charset="utf-8">
5     <title>Canvas06</title>
6     <script>
7         const pi = Math.PI;
8         let cvs, ctx;
9         function f1() {
10             ctx.fillStyle = "yellow"; ctx.strokeStyle = "blue";
11             ctx.lineWidth = 18;
12             ctx.beginPath();
13             ctx.moveTo(383.2, 310); // A
14             ctx.lineTo(210,210); // B
15             ctx.lineTo(383.2, 110); // C
16             ctx.arc(210,210,201,11*pi/6,pi/6,true); // (1)
17             // ctx.arc(210,210,201,11*pi/6,pi/6,false); // (2)
18             ctx.closePath();
19             ctx.fill(); ctx.stroke();
20         }
21         function f0() {
22             cvs = document.getElementById("cvs"); // canvas要素の取得
23             cvs.width = 420; // canvas要素のwidth属性を設定
```

```

24         cvs.height = 420;                // canvas要素のheight属性を設定
25         ctx = cvs.getContext("2d"); // 描画コンテキストの取得
26         f1();                // 描画ルーチン
27     }
28     window.addEventListener("load",f0);
29 </script>
30 </head>
31 <body>
32     <canvas id="cvs"></canvas>
33 </body>
34 </html>

```

このサンプル中のコメント (1),(2) の内, (1) の行のみを有効にして Web ブラウザで表示した例を図 52 の左側 (輪郭線とフィルのみ) に示す。

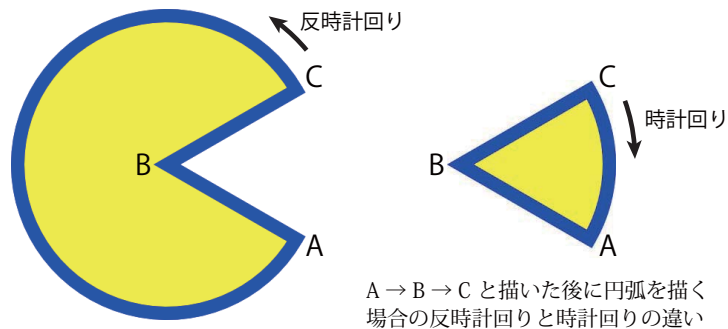


図 52: Canvas06.html でのパス (円弧を含む) の描画

Canvas06.html のコメント (2) の行のみを有効にすると図 52 の右側のように (輪郭線とフィルのみ) 表示される。

参考) 先の Canvas06.html の描画関数 f1 を次のようにすると円弧ではなく円が描画される。

記述例：

```

1 function f1() {
2     ctx.strokeStyle = "blue";
3     ctx.lineWidth = 5;
4     ctx.beginPath();
5     for ( let r=7; r<50; r+=10 ) {
6         ctx.moveTo(r+50,50);
7         ctx.arc(50,50,r,0,2*pi);
8         ctx.closePath();
9     }
10    ctx.stroke();
11 }

```



関数 f1 を左のように変更して Web ブラウザで表示した例

7.4.3.3 楕円弧の描画

ellipse メソッドによって楕円弧を描くことができる。またこれを応用すると楕円を描くことができる。

書き方： 描画コンテキスト.ellipse(x, y, 横半径, 縦半径, 傾き, 開始角, 終了角, 反時計回り)

座標 (x,y) を中心とする「横半径」「縦半径」の楕円弧を描く。描く楕円弧の範囲は「開始角」から「終了角」までで、「反時計回り」に true を与えると反時計回り、false を与えると時計回りに楕円弧を描く。「反時計回り」の引数は省略可能で、デフォルトは false である。「傾き」には楕円弧の中心を軸にして回転させる角度を与える。

ellipse メソッドで作成する楕円弧はパスの一部である。複数の楕円を描く例を Canvas07.html に示す。

記述例： Canvas07.html

```

1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4     <meta charset="utf-8">
5     <title>Canvas07 </title>
6     <script>
7         const pi = Math.PI;
8         let cvs, ctx;

```

```

9      function f1() {
10         ctx.strokeStyle = "blue";
11         ctx.lineWidth = 5;
12         for ( let n=0; n<12; n++ ) {
13             ctx.beginPath();
14             ctx.ellipse(150,150,140,20,n/12*pi,0,2*pi);
15             ctx.closePath();
16             ctx.stroke();
17         }
18     }
19     function f0() {
20         cvs = document.getElementById("cvs");    // canvas要素の取得
21         cvs.width = 300;                        // canvas要素のwidth属性を設定
22         cvs.height = 300;                       // canvas要素のheight属性を設定
23         ctx = cvs.getContext("2d"); // 描画コンテキストの取得
24         f1();    // 描画ルーチン
25     }
26     window.addEventListener("load",f0);
27 </script>
28 </head>
29 <body>
30     <canvas id="cvs"></canvas>
31 </body>
32 </html>

```

これは 12 個の楕円を 30° ずつ回転させながら描画するもので、これを Web ブラウザで表示した例を図 53 に示す。

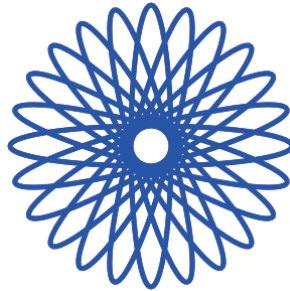


図 53: Canvas07.html による描画

7.4.4 画像の描画

drawImage メソッドを用いることで canvas 上に画像を描画することができる。

書き方： 描画コンテキスト.drawImage(画像, x, y)

「画像」の左上を canvas の座標 (x,y) に配置して描画する。戻り値はない (undefined)。

図 54 に示す画像 JavaScript_11st.png を canvas 上に描画するサンプル Canvas08.html を示す。

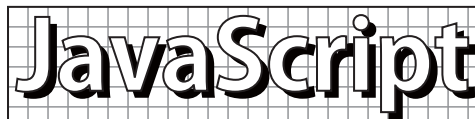


図 54: サンプル画像：JavaScript_11st.png

記述例：Canvas08.html

```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4      <meta charset="utf-8">
5      <title>Canvas08</title>
6      <style>
7          #cvs { border: solid 1px black; }
8      </style>
9      <script>
10         let cvs, ctx, im = new Image();
11         function f1() {
12             ctx.drawImage(im,10,10);
13         }

```

```

14     function f0() {
15         im.src = "JavaScript_11st.png";
16         cvs = document.getElementById("cvs");    // canvas要素の取得
17         cvs.width = 500;                        // canvas要素のwidth属性を設定
18         cvs.height = 140;                       // canvas要素のheight属性を設定
19         ctx = cvs.getContext("2d");             // 描画コンテキストの取得
20         f1();                                    // 描画ルーチン
21     }
22     window.addEventListener("load",f0);
23 </script>
24 </head>
25 <body>
26     <canvas id="cvs"></canvas>
27 </body>
28 </html>

```

参考) 画像を拡張して canvas に描画することも可能である。

書き方： 描画コンテキスト.drawImage(画像, x, y, w, h)

「画像」の左上を canvas の座標 (x,y) に配置して描画する。この場合「画像」の横幅を w, 高さを h に拡張して描画する。戻り値はない (undefined)。

7.4.5 画素 (ピクセル) の描画と取得

ImageData オブジェクトは汎用の画素配列として扱うことができ、このオブジェクトの画素 (ピクセル) は直接編集することができる。ImageData オブジェクトの色空間はデフォルトで RGBA¹⁰³ である¹⁰⁴。

ImageData オブジェクトを作成するには canvas 要素の描画コンテキストに対して createImageData メソッドを実行する。

書き方： 描画コンテキスト.createImageData(w, h)

横幅「w」、高さ「h」の ImageData オブジェクトを生成して返す。

参考) ImageData オブジェクトは当該クラスのコンストラクタを使用して作成することもできる。

書き方： new ImageData(w, h)

7.4.5.1 ImageData オブジェクトの画素の構造

ImageData オブジェクトの data プロパティは画素の RGBA の各値 (0~255) を並べた 1 次元の配列 (表 52 参照) である。

この他にも、ImageData オブジェクトは横幅を与える width, 高さを与える height の各プロパティを持つ。

ImageData オブジェクトの data プロパティは 1 次元の配列なので、座標 (x,y) を指定して RGBA の値を設定するには次に示すような関数 setPixel を実装しておくと便利である。

記述例： ImageData オブジェクトの画素を設定する setPixel 関数の実装

```

1 function setPixel(img,x,y,r,g,b,a) {
2     img.data[(img.width * y + x)*4] = r;
3     img.data[(img.width * y + x)*4 + 1] = g;
4     img.data[(img.width * y + x)*4 + 2] = b;
5     img.data[(img.width * y + x)*4 + 3] = a;
6 }

```

7.4.5.2 ImageData オブジェクトの描画

ImageData オブジェクトを canvas 要素に描画するには putImageData メソッドを使用する。

書き方： 描画コンテキスト.putImageData(ImageData オブジェクト, x, y)

canvas の座標 (x,y) の位置に「ImageData オブジェクト」の左上の角を配置する形で描画する。このメソッドの戻り値はない (undefined)。

¹⁰³RGB の 3 色に加えてアルファ値 (不透明度) を持つ。

¹⁰⁴他の色空間を指定することも可能である。

表 52: 例. ImageData オブジェクト img の画素の構造

記 述	対象画素 (n 番目)	意味
img.data[0]	第 0 番目の画素	R 値
img.data[1]	第 0 番目の画素	G 値
img.data[2]	第 0 番目の画素	B 値
img.data[3]	第 0 番目の画素	アルファ値
img.data[4]	第 1 番目の画素	R 値
img.data[5]	第 1 番目の画素	G 値
img.data[6]	第 1 番目の画素	B 値
img.data[7]	第 1 番目の画素	アルファ値
img.data[8]	第 2 番目の画素	R 値
img.data[9]	第 2 番目の画素	G 値
img.data[10]	第 2 番目の画素	B 値
img.data[11]	第 2 番目の画素	アルファ値
:	:	:

先に解説した drawImage メソッドでも ImageData オブジェクトを canvas に描画することが可能である。putImageData メソッドが、ImageData オブジェクトの画素の配列で canvas 内の指定した場所の画素を上書きするのに対して、drawImage は画像に拡大縮小などの処理を施して canvas 上に描画するという違いがある。

画素を直接操作する形で画像を作成して canvas に描画する例を Canvas09.html に示す。

記述例：Canvas09.html

```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>Canvas09</title>
6    <style>
7      #cvs { border: solid 1px black; }
8    </style>
9    <script>
10     let cvs, ctx, im;
11     function setPixel(img,x,y,r,g,b,a) {
12       img.data[(img.width * y + x)*4] = r;
13       img.data[(img.width * y + x)*4 + 1] = g;
14       img.data[(img.width * y + x)*4 + 2] = b;
15       img.data[(img.width * y + x)*4 + 3] = a;
16     }
17     function f1() {
18       let x, y, r, g, b, xs = 0;
19       for ( r = 0; r < 256; r+=255 ) {
20         for ( g = 0; g < 256; g+=255 ) {
21           for ( b = 0; b < 256; b+=255 ) {
22             for ( x = xs; x < xs+50; x++ ) {
23               for ( y = 0; y < 150; y++ ) {
24                 setPixel(im,x,y,r,g,b,255);
25               }
26             }
27             xs += 50;
28           }
29         }
30       }
31       ctx.putImageData(im,0,0);
32     }
33     function f0() {
34       cvs = document.getElementById("cvs"); // canvas要素の取得
35       cvs.width = 400; // canvas要素のwidth属性を設定
36       cvs.height = 150; // canvas要素のheight属性を設定
37       ctx = cvs.getContext("2d"); // 描画コンテキストの取得

```



```

38         im = ctx.createImageData(400,150);
39         f1();          // 描画ルーチン
40     }
41     window.addEventListener("load",f0);
42 </script>
43 </head>
44 <body>
45     <canvas id="cvs"></canvas>
46 </body>
47 </html>

```

このサンプルを Web ブラウザで表示すると図 55 のようになる。

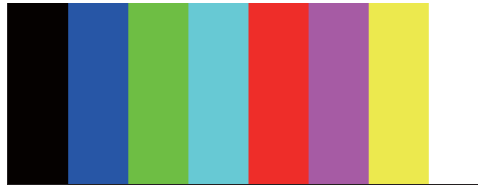


図 55: Canvas09.html を Web ブラウザで表示したところ

7.4.5.3 canvas から画素を取得する方法

canvas から画素を取得するには getImageData メソッドを使用する。

書き方： 描画コンテキスト.getImageData(x, y, w, h)

canvas 要素から指定した矩形領域の画素を取り出して ImageData オブジェクトとして返す。矩形領域の左上の位置は (x,y)、横幅は w、高さは h で指定する。得られる ImageData オブジェクトは対象の canvas 要素の部分の複製であり、参照ではない。

7.5 スケール、傾き、原点の変更

描画コンテキストに対して表 53 に示すメソッドを用いると、描画コンテキストの状態を変更することができる。

表 53: 描画コンテキストの状態を変更するメソッド（一部）

メソッド	解 説
scale(Mx, My)	描画のスケールを横の比率 Mx, 縦の比率 My に設定する。原点は変更しない。
rotate(A)	原点を中心に座標系を A(rad) 傾ける。原点は変更しない。
translate(dx, dy)	原点を現在の位置から横に dx, 縦に dy ずらす。

次のサンプル Canvas10.html は表 53 に示したメソッドによって画像（JavaScript_11st.png¹⁰⁵）の描画がどのように変化するかを示すものである。

記述例：Canvas10.html

```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4      <meta charset="utf-8">
5      <title>Canvas10</title>
6      <style>
7          #cvs { border: solid 1px black; }
8      </style>
9      <script>
10         let cvs, ctx, pi = Math.PI, im = new Image();
11         function f1() {
12             // ctx.scale(2.0,3.0);          // (1)
13             // ctx.translate(100,50);      // (2)
14             // ctx.rotate(pi/6);           // (3)
15             ctx.drawImage(im,10,10);
16         }
17         function f0() {

```

¹⁰⁵p.198 の図 54 に示したもの。

```

18         im.src = "JavaScript_11st.png";
19         im.onload = function() {
20             cvs = document.getElementById("cvs");    // canvas要素の取得
21             cvs.width = 500;                        // canvas要素のwidth属性を設定
22             cvs.height = 300;                       // canvas要素のheight属性を設定
23             ctx = cvs.getContext("2d");             // 描画コンテキストの取得
24             f1();                                    // 描画ルーチン
25         }
26     }
27     window.addEventListener("load",f0);
28 </script>
29 </head>
30 <body>
31     <canvas id="cvs"></canvas>
32 </body>
33 </html>

```

このサンプルを Web ブラウザで表示すると図 56 の a ようになる。

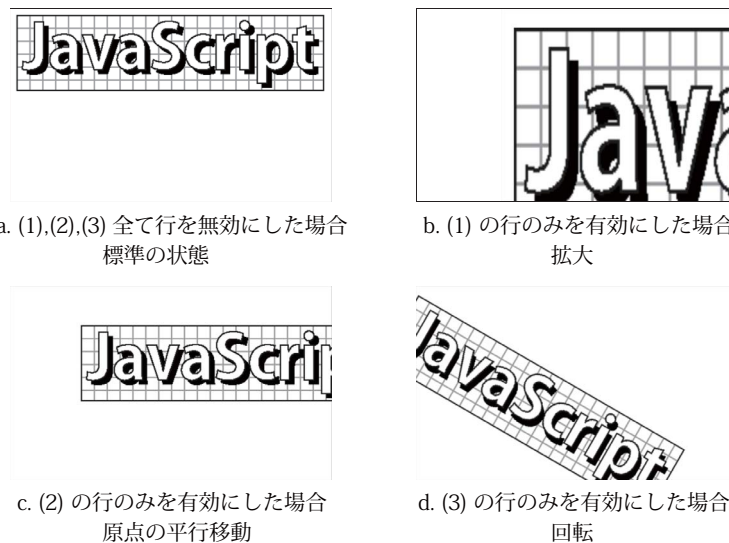


図 56: Canvas10.html を Web ブラウザで表示したところ

Canvas10.html のコメント行 (1)~(3) の内、(1) の部分のみを有効にした場合は図 56 の b、(2) の部分のみを有効にした場合は c、(3) の部分のみを有効にした場合は d のように表示される。

注意)

上の例のケースでは、画像の読み込みの完了を待つため、im のイベント load を受けて画像の描画を行うという流れにしている。

7.5.1 描画コンテキストの状態の保存と復元

表 53 (p.201) に示したメソッド群で描画コンテキストの状態を変更した後、逆の状態変更を行って再度元の状態に戻すのは煩わしい作業である。ただし、描画コンテキストには現在の状態を保存する機能 (save メソッド) と復元する機能 (restore メソッド) があり、それらを利用すると、描画コンテキストの状態を変更する前に戻ることができる。

書き方: 描画コンテキスト.save()

書き方: 描画コンテキスト.restore()

save メソッドを実行すると、その時点での描画コンテキストの状態がシステム内部のスタックに保存される。restore メソッドを実行すると、描画コンテキストの状態が先に save メソッドによって保存された状態となる。

save メソッドで保存した状態を restore メソッドで復元する前に更に save メソッドを実行すると、その時点の状態がスタックに積み上げられ、restore 実行時には最近 save を実行した状態に復元される。

save, restore どちらのメソッドも値を返さない (undefined)。

8 グラフィックスの作成 (2) : SVG

Scalable Vector Graphics (SVG) は XML で記述される 2 次元のベクターグラフィックス¹⁰⁶ である。これは抽象化された図形要素から成るもので、画素で構成されたビットマップグラフィックスと異なり、拡大表示の際にジャギーなどの表示の崩れが発生しない。ビットマップグラフィックスのことをペイント系グラフィックス、ベクターグラフィックスのことをドロー系グラフィックスと呼ぶこともある。

厳密には SVG の仕様は HTML と異なるが、両者とも XML の拡張であり、HTML 文書の中に SVG によるグラフィックスを直接的に埋め込むことが可能である。また、多くの主要な Web ブラウザが SVG グラフィックスをレンダリングする機能を持っており、SVG で記述されたグラフィックスの要素に対して、CSS による装飾や JavaScript による操作が可能である。以上のような理由から、SVG は Web コンテンツ中にベクターグラフィックスを埋め込むための標準的な方法となっている。

SVG は W3C が仕様を取りまとめて勧告している。本書執筆時点での版は 1.1 である。

8.1 基礎事項

8.1.1 座標系と大きさ、角度の単位

SVG の座標系は x 軸（横軸）と y 軸（縦軸）からなる 2 次元であり、右に行くほど x の値が大きくなり、下に行くほど y の値が大きくなる。大きさの表現に関しては特定の単位系に依存しない抽象的なものであるが、Web ブラウザ上での表示の際にはピクセルサイズ (px) として解釈するのが一般的である。

SVG の座標系においては角度の単位は度数法の「度」(1 周が 360°) で時計回りを正、反時計回りを負とする。また図形要素の傾きを表す際は水平を 0° とする。

8.1.2 記述の形式

SVG は **svg 要素** (svg タグ) として記述する。

《svg 要素》

```
<svg xmlns="http://www.w3.org/2000/svg">
  ⋮
  (図形描画の記述)
  ⋮
</svg>
```

xmlns 属性に与えているものは特定の Web コンテンツのための URL ではなく、SVG のためのキーワードなどを定義する XML の名前空間を意味する文字列である。この名前空間は、SVG 要素と属性を正しく解釈するために必要な情報を提供する。SVG を作成する場合は、特殊なケースや拡張要素を使用する場合を除いて、この文字列 "http://www.w3.org/2000/svg" を固定値として与える。

svg 要素内には HTML の場合と同様に title 要素を記述することができる。ただしこれは推奨事項であり必須ではない。

8.1.3 ビューポートとビューボックス

SVG の座標系の中の、表示対象の矩形領域を**ビューボックス**と呼ぶ。この領域の位置と大きさは svg 要素の viewBox 属性に与える。

書き方: `viewBox = "x y w h"`

SVG の座標における (x,y) を左上とする横幅 w 高さ h の矩形領域を表示対象とする。

ビューボックスの、ブラウザ上における表示領域を**ビューポート**と呼ぶ。ビューポートの横幅は svg 要素の width 属性に、高さは height 属性に与える。

書き方: `width = "W" height="H"`

¹⁰⁶ 「ベクトルグラフィックス」とも呼ばれることもある。

ビューボックスをブラウザ上の矩形領域（横幅 W、高さ H のビューポート）に対応（図 57）させる。ブラウザ上でのビューポートの位置とサイズは、当該 SVG を含む HTML 文書の記述や CSS の設定によって制御される。

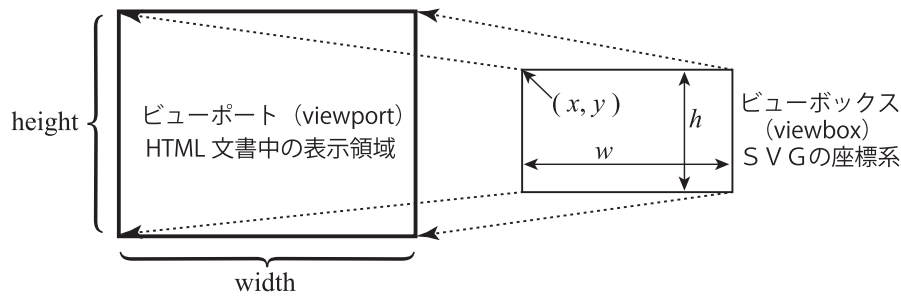


図 57: ビューポートとビューボックスの対応

注意)

ビューポートとビューボックスのアスペクト比（縦横のサイズの比率）は同じにしておくべきであるが、両者のアスペクト比が異なる場合は、ビューボックスの内容がそのアスペクト比を保った形でビューポートに収まるように自動的に調整（縮小）される。また、svg 要素に属性「`preserveAspectRatio="none"`」を与えると、ビューボックスのアスペクト比を変更してビューポートに強制的に一致（図形を変形）させる。

svg 要素の `preserveAspectRatio` 属性に関する詳細は W3C の公式インターネットサイトなどを参照のこと。

8.1.4 HTML への SVG の配置

SVG を HTML 文書中に配置するには、基本的には 3 つの異なる手法がある。1 つは SVG を単純に画像データとして（img 要素として）HTML 中に配置する方法である。この場合、配置された SVG はあくまで「画像データ」であり、SVG 内の各要素は HTML の DOM の体系の中での要素としては扱えない。

2 つ目の方法は、SVG 要素を HTML 文書の中に直接的に記述することである。SVG は HTML と同様に「タグ」で構成されるので、そのまま HTML 文書中に記述することができる。この方法では、記述された SVG 内の各要素が HTML の DOM の階層の中に組み込まれるので、それらは CSS のスタイル設定や JavaScript の処理対象となる。従って最も直感的な実装方法とすることができるが、SVG は画像データとして独立したファイルの形で扱われることが多く、HTML 文書中に直接的に記述するという手法は、ファイル管理の観点から好ましくない場合もある。

3 つ目の方法は object 要素（あるいは embed 要素）として HTML コンテンツの DOM 階層の中に SVG を組み込むことである。この方法では、独立したファイルとして与えられた SVG 要素（とその配下の要素）を HTML の DOM 階層の中に組み込むことができ、CSS の設定対象、あるいは JavaScript の処理対象とすることができる。

8.1.4.1 img 要素として配置する方法

SVG として作成したグラフィックスはファイルとして保存（拡張子：`*.svg`）しておき、それを HTML の img 要素として読み込んで表示することができる。以下に具体的な例を挙げて解説する。

次のようなファイル SVG00.svg として作成された SVG について考える。

記述例：SVG00.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg"
2   width="162" height="85" viewBox="0 0 162 85">
3   <title>SVG00</title>
4   <rect x="5" y="5" width="152" height="75"
5     stroke="black" stroke-width="10" fill="yellow"/>
6   <text x="15" y="65" fill="red"
7     font-size="60" font-family="sans-serif" font-weight="bold"
8   >S</text>
9   <text x="56" y="65" fill="green"
10    font-size="60" font-family="sans-serif" font-weight="bold"
11  >V</text>
12  <text x="98" y="65" fill="blue"
13    font-size="60" font-family="sans-serif" font-weight="bold"
14  >G</text>
15 </svg>
```

この SVG は図 59 のようなもので、直接 Web ブラウザで表示することも可能である。SVG 内の図形要素に関する詳細は後の「8.2 SVG の図形要素」(p.207) で解説する。



図 58: SVG00.svg を Web ブラウザで表示したところ

上に示した SVG を HTML 内に配置する例を次の SVG00.html に示す。

記述例：SVG00.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>SVG00</title>
6   <style>
7     div {
8       width: 170px; height: 93px; margin: 0px;
9       padding-top: 8px; padding-left: 8px;
10      border: solid 1px #7f7f7f;
11    }
12  </style>
13 </head>
14 <body>
15   <div></div>
16 </body>
17 </html>
```

これを Web ブラウザで表示すると、図 59 のようになる。この例においては img 要素が SVG の結果的なビューポートとなる。

勿論、SVG の内容自体を HTML コンテンツ中に記述しても良いが、保守管理の目的から SVG は画像データとして別のファイルにしておくことが一般的である。



図 59: SVG00.html を Web ブラウザで表示したところ

8.1.4.2 object 要素として配置する方法

下記のように object 要素の形式で SVG を HTML 内に組み込むことができる。

《 object 要素》

```
<object data=SVG リソース type="image/svg+xml">
  ⋮
  (代替テキスト)
  ⋮
</object>
```

「SVG リソース」は読み込む SVG のファイルのパスや URI である。SVG を読み込む場合の MIME タイプ（メディアタイプ）として type 属性に "image/svg+xml" を与える。

この形式で SVG を HTML 内に組み込むと、Web ブラウザ上で表示されるだけでなく、SVG 内部の要素が HTML の DOM 階層の中に組み込まれ、CSS の設定や JavaScript の処理の対象となる。

先に示した SVG00.svg を object 要素として HTML に組み込む例 SVG00-2.html を示す。

記述例：SVG00-2.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>SVG00</title>
6 </head>
7 <body>
8   <object data="SVG00.svg" type="image/svg+xml">
9     これは SVG00.svg です。
10  </object>
11 </body>
12 </html>
```

注意) object 要素として読み込むことができる SVG は当該 HTML と同じオリジン内のものでなければならない¹⁰⁷。

8.1.5 defs 要素による定義の再利用

SVG では defs 要素を配置することで、その内部に記述した定義事項を再利用することができる。defs 要素は HTML の場合と同じように、開始タグ <defs> から始まり、終了タグ </defs> で終わる。

SVG では HTML の場合と同様に CSS でスタイルを記述することができ、defs 要素内に style 要素 (<style>～</style>) を配置することができる。

SVG で要素の個数が多い複雑なグラフィックスを作成する際に defs 要素は有用である。先に示したサンプル SVG00.svg を defs 要素を用いて書き直した SVG00-2.svg を示す。

記述例：SVG00-2.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg"
2   width="162" height="85" viewBox="0 0 162 85">
3   <defs>
4     <style>
5       .txtElements {
6         font-size: 60px;
7         font-family: sans-serif;
8         font-weight: bold;
9       }
10    </style>
11  </defs>
12  <title>SVG00-2</title>
13  <rect x="5" y="5" width="152" height="75"
14    stroke="black" stroke-width="10" fill="yellow"/>
15  <text class="txtElements" x="15" y="65" fill="red">S</text>
16  <text class="txtElements" x="56" y="65" fill="green">V</text>
17  <text class="txtElements" x="98" y="65" fill="blue">G</text>
18 </svg>
```

全ての text 要素にクラス名 txtElements が与えられており、それに対して CSS でスタイルを与えている。また CSS 利用の都合上、font-size 属性の値に単位 px が記述されている。

注意) SVG を Web コンテンツの画像データとして扱う場合、SVG 内に記述された CSS は Web ブラウザによって解釈されるが、SVG を一般的な画像データとしてドロー系ソフトウェアで扱う場合は、内部に記述された CSS が正しく解釈されない場合があるので注意すること。

defs 要素として記述された定義内容は use 要素で展開することができる。これに関しては、図形要素と合せて解説する。

¹⁰⁷同一オリジンポリシー (p.150)

8.2 SVG の図形要素

SVG の図形要素は Canvas グラフィックス¹⁰⁸ の場合と基本的な考え方が似ており、輪郭線（ストローク）と、それによって囲まれる塗りつぶしの領域から成る（図 41）。

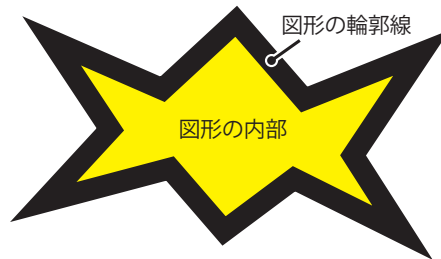
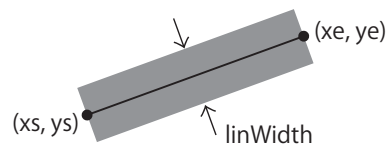


図 41: 線画図形の基本構造 (p.190 より)

本書では以降の解説の中で、この「図形の内部」のことを塗りつぶしもしくはフィルと呼ぶ。また、輪郭線のことをストロークと呼ぶことがある。輪郭線とフィルは別の実体ではなく、1つの図形要素に統合されている点が Canvas グラフィックスの場合と異なる。

線を描く際の座標と線幅の考え方も Canvas グラフィックスのそれと基本的には同じ（図 43）である。



線の両端の座標は (xs,ys), (xe,ye)

図 43: 線の幅と両端の座標の関係 (p.191 より)

本書では SVG の図形要素の内、使用頻度の高いものを選んで解説する。

8.2.1 多角形, 折れ線

polygon 要素は多角形を実現する。

《多角形》

```
<polygon fill=フィルの設定 stroke=輪郭線の設定 stroke-width=線幅  
points=頂点の並び />
```

「頂点の並び」から成る多角形を実現する。「フィルの設定」,「輪郭線の設定」には色やグラデーション, パターン (テクスチャ) が指定できるが, "none" を指定すると透明になる。

● fill, stroke, stroke-width といった属性の設定に関しては, 多角形, 折れ線以外の図形要素においても同様である。

points 属性には頂点の座標の並びを次のような書式で記述して与える。

書き方: "x1,y1 x2,y2 ... xn,yn"

頂点の座標の並び (x1,y1)-(x2,y2)- ... -(xn,yn) を結んだ多角形となる。

polygon 要素で多角形を描画する例 SVG01-1.svg を示す。

記述例: SVG01-1.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">  
2   <title>多角形</title>  
3   <!-- 多角形 (塗りつぶし) -->  
4   <polygon fill="#FFFF00" stroke="#000000" stroke-width="6"  
5     points="89.316,33.427 100.164,60.902 128.408,52.252 113.689,77.864  
6       138.063,94.553 108.862,99.014 111.011,128.475 89.316,108.427  
7       67.622,128.475 69.771,99.014 40.57,94.553 64.943,77.864  
8       50.225,52.252 78.469,60.902"/>
```

¹⁰⁸ 「7 グラフィックスの作成 (1): canvas 要素」(p.190) で解説。

```

9      <!-- 多角形（塗りなし） -->
10     <polygon fill="none" stroke="#000000" stroke-width="6"
11         points="219.316,128.476 230.164,101 258.408,109.65 243.689,84.039
12             268.063,67.35 238.862,62.888 241.011,33.427 219.316,53.476
13             197.622,33.427 199.771,62.888 170.57,67.35 194.943,84.039
14             180.225,109.65 208.469,101"/>
15 </svg>

```

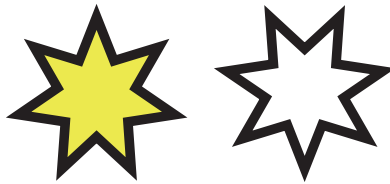


図 60: SVG01-1.svg を Web ブラウザで表示したところ

polygon 要素と類似の書き方で折れ線を実現する polyline 要素がある。polygon 要素では points 属性に与えた座標並びの始点と終点を結んで閉じるが、polyline 要素は折れ線なので始点と終点を結ばない。

8.2.2 輪郭線に関する設定事項

8.2.2.1 破線の設定

図形要素に属性 stroke-dasharray を与えることで輪郭線を破線にすることができる。この属性に与える設定は次の通りである。

書き方： ”線の長さ 1 間隔 1 線の長さ 2 間隔 2 …”

このように、線の長さと間隔の大きさを交互に記述する。

破線を描く例を SVG02-1.svg に示す。

記述例：SVG02-1.svg

```

1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>折れ線</title>
3   <polyline fill="none" stroke="#000000" stroke-width="10"
4       points="41,42 41,142 71,42 71,142 "/>
5   <polyline fill="none" stroke="#000000" stroke-width="3"
6       stroke-dasharray="2,3"
7       points="141,42 141,142 171,42 171,142 "/>
8 </svg>

```

これを Web ブラウザで表示すると図 61 のようになる。



図 61: SVG02-1.svg を Web ブラウザで表示したところ

1 つ目の polyline（実線）が図 61 の左側に、2 つ目の polyline（破線）が右側に表示される。

輪郭線の設定を CSS で与えることもでき、先の例と同じものを SVG02-2.svg のように記述することができる。

記述例：SVG02-2.svg

```

1 <svg xmlns="http://www.w3.org/2000/svg">
2   <defs>
3     <style>
4       #solidLine {
5         fill: none;
6         stroke: #000000;
7         stroke-width: 10;
8       }

```

```

9      #dashedLine {
10         fill: none;
11         stroke: #000000;
12         stroke-width: 3;
13         stroke-dasharray: 2,3;
14     }
15     </style>
16 </defs>
17 <title>折れ線</title>
18 <polyline id="solidLine" points="41,42 41,142 71,42 71,142 "/>
19 <polyline id="dashedLine" points="141,42 141,142 171,42 171,142 "/>
20 </svg>

```

勿論この場合も、p.206 の注意事項を意識すること。

8.2.2.2 線の端の形状の設定

図形要素に属性 `stroke-linecap` を与えることで輪郭線の端の形状を制御することができる。この属性に "butt", "round", "square" の値を与えた場合の様子を図 62 に示す。

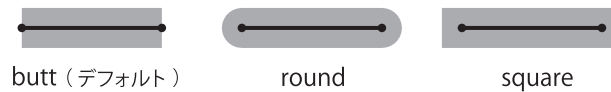


図 62: 線の端の形状

8.2.2.3 線の結合部の形状の設定

図形要素に属性 `stroke-linejoin` を与えることで線の結合部の形状を制御することができる。この属性に "miter", "round", "bevel" の値を与えた場合の様子を図 63 に示す。



図 63: 線の結合部の形状

`stroke-linejoin` 属性の設定例を SVG02-3.svg に示す。

記述例: SVG02-3.svg

```

1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>折れ線（曲がり角の指定）</title>
3   <polyline fill="none" stroke="#000000" stroke-width="14"
4     stroke-linejoin="miter" stroke-miterlimit="10"
5     points="33,41 33,111 73,41 73,111 "/>
6   <polyline fill="none" stroke="#000000" stroke-width="14"
7     stroke-linejoin="round" stroke-miterlimit="10"
8     points="113,41 113,111 153,41 153,111 "/>
9   <polyline fill="none" stroke="#000000" stroke-width="14"
10    stroke-linejoin="bevel" stroke-miterlimit="10"
11    points="193,41 193,111 233,41 233,111 "/>
12 </svg>

```

これを Web ブラウザで表示すると図 64 のようになる。



図 64: SVG02-3.svg を Web ブラウザで表示したところ

■ miter の制限について

”miter” の結合における突出の長さの制限には、図形要素の属性 `stroke-miterlimit` が関係する。(図 65 参照)

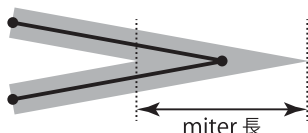


図 65: $\text{miterlimit} = \text{miter 長} / \text{線幅}$

線の幅に対する線の結合部の **miter 長**の比率を `stroke-miterlimit` に与えておくと、実際の miter 長がこの制限を超える場合に結合部の形状が ”bevel” となる¹⁰⁹。このことを次のサンプル SVG02-4.svg で示す。

記述例：SVG02-4.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>折れ線（留幅の指定）</title>
3   <polyline fill="none" stroke="#000000" stroke-width="10"
4     stroke-miterlimit="100"
5     points="35,79 35,179 105,79 105,179 145,79
6       145,179 165,79 165,179 "/>
7   <polyline fill="none" stroke="#000000" stroke-width="10"
8     stroke-miterlimit="10"
9     points="245,79 245,179 315,79 315,179 355,79
10      355,179 375,79 375,179 "/>
11 </svg>
```

これを Web ブラウザで表示すると図 66 のようになる。



図 66: SVG02-4.svg を Web ブラウザで表示したところ

この例では、`polyline` 要素に `stroke-linejoin` 属性を設定していないため線の結合部は ”miter” となる。ただし、線の結語部の突出が大きくなりすぎる部分が自動的に ”bevel” となっていることがわかる。1 つ目の `polyline` では `stroke-miterlimit` 属性に十分大きい値 (100) を与えているので全ての結合部が ”miter” となっているが、2 つ目の `polyline` では右側の 2 つの結合部が自動的に ”bevel” となっていることがわかる。

8.2.3 矩形：rect 要素

`rect` 要素は矩形（四角形）を描画する。`rect` 要素は輪郭線とフィルに関する属性に加えて次のような属性を持つ。

属性	意味	属性	意味
<code>x</code>	矩形の左上の座標の横位置	<code>y</code>	矩形の左上の座標の縦位置
<code>width</code>	矩形の横幅	<code>height</code>	矩形の高さ
<code>rx</code>	角の丸みの横半径	<code>ry</code>	角の丸みの縦半径

8.2.4 円：circle 要素

`circle` 要素は円を描画する。`circle` 要素は輪郭線とフィルに関する属性に加えて次のような属性を持つ。

属性	意味	属性	意味	属性	意味
<code>cx</code>	中心の座標の横位置	<code>cy</code>	中心の座標の縦位置	<code>r</code>	半径

¹⁰⁹この制限は Canvas グラフィックスの場合 (p.192) と数学的には同義である。

8.2.5 楕円：ellipse 要素

ellipse 要素は楕円を描画する。ellipse 要素は輪郭線とフィルに関する属性に加えて次のような属性を持つ。

属性	意味	属性	意味
cx	中心の座標の横位置	cy	中心の座標の縦位置
rx	横方向の半径	ry	縦方向の半径

8.2.6 直線：line 要素

line 要素は直線を描画する。line 要素は輪郭線とフィルに関する属性に加えて次のような属性を持つ。

属性	意味	属性	意味	属性	意味	属性	意味
x1	始点の横位置	y1	始点の縦位置	x2	終点の横位置	y2	終点の縦位置

■ サンプル

円、楕円、矩形、直線を描画するサンプル SVG03-1.svg を示す。

記述例：SVG03-1.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>円, 楕円, 直線, 矩形</title>
3   <circle fill="none" stroke="#000000" stroke-width="10"
4     cx="80" cy="75" r="45"/>
5   <ellipse fill="none" stroke="#000000" stroke-width="10"
6     cx="260" cy="75" rx="120" ry="45"/>
7   <rect fill="none" stroke="#000000" stroke-width="10"
8     x="400" y="30" width="200" height="90"/>
9   <line stroke="#000000" stroke-width="10"
10    x1="30" y1="145" x2="610" y2="145"/>
11 </svg>
```

これを Web ブラウザで表示すると図 67 のようになる。



図 67: SVG03-1.svg を Web ブラウザで表示したところ

8.2.7 テキスト（文字列）：text 要素

text 要素はテキスト（文字列）を描画する。

書き方： <text>描画対象テキスト</text>

text 要素は輪郭線とフィルに関する属性に加えて次のような属性を持つ。

属性	意味	属性	意味
x	始点の横位置	y	始点の縦位置
rotate	個々の文字の角度	font-size	フォントの大きさ
font-family	フォント名	font-weight	フォントの太さ lighter / normal / bold / bolder (注 1)
font-style	フォントのスタイル normal / italic / oblique (注 2)	text-decoration	文字の装飾 underline / overline / line-through

(注 2) italic と oblique は同じになることがある。 (注 1) bold と bolder は同じになることがある。

8.2.7.1 テキストの描画位置の基準

テキストを描画する座標位置 (x,y) は、テキストのベースラインの左端¹¹⁰ を基準とする。(図 68)



図 68: テキストの位置

8.2.7.2 テキストの強調

font-weight 属性を指定して、テキストの強調を制御する例 SVG03-2.svg を示す。

記述例: SVG03-2.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>テキスト</title>
3   <text x="10" y="30" font-family="serif" font-size="24" font-weight="lighter">
4     >1. 文字列描画ABCabcgjpqy</text>
5   <text x="10" y="70" font-family="serif" font-size="24" font-weight="normal">
6     >2. 文字列描画ABCabcgjpqy</text>
7   <text x="10" y="110" font-family="serif" font-size="24" font-weight="bold">
8     >3. 文字列描画ABCabcgjpqy</text>
9   <text x="10" y="150" font-family="serif" font-size="24" font-weight="bolder">
10    >4. 文字列描画ABCabcgjpqy</text>
11 </svg>
```

これを Web ブラウザで表示すると右のようになる。この例では font-weight 属性が "bold" の場合と "bolder" の場合で同じ結果となっている。

- 1.文字列描画ABCabcgjpqy
- 2.文字列描画ABCabcgjpqy
- 3.文字列描画ABCabcgjpqy
- 4.文字列描画ABCabcgjpqy

8.2.7.3 フォントスタイル

font-style 属性を指定してフォントスタイルを制御する例 SVG03-3.svg を示す。

記述例: SVG03-3.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>テキスト</title>
3   <text x="10" y="30" font-family="serif" font-size="24" font-style="normal">
4     >1. 文字列描画ABCabcgjpqy</text>
5   <text x="10" y="70" font-family="serif" font-size="24" font-style="italic">
6     >2. 文字列描画ABCabcgjpqy</text>
7   <text x="10" y="110" font-family="serif" font-size="24" font-style="oblique">
8     >3. 文字列描画ABCabcgjpqy</text>
9 </svg>
```

これを Web ブラウザで表示すると右のようになる。この例では font-style 属性が "italic" の場合と "oblique" の場合で同じ結果となっている。

- 1.文字列描画ABCabcgjpqy
- 2.文字列描画ABCabcgjpqy
- 3.文字列描画ABCabcgjpqy

8.2.7.4 下線, 上線, 打消し線

text-decoration 属性を指定して、下線, 上線, 打消し線を表示する例 SVG03-4.svg を示す。

記述例: SVG03-4.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>テキスト</title>
3   <text x="10" y="30" font-family="serif" font-size="24"
4     text-decoration="underline">1. 文字列描画ABCabcgjpqy</text>
5   <text x="10" y="70" font-family="serif" font-size="24"
6     text-decoration="overline">2. 文字列描画ABCabcgjpqy</text>
7   <text x="10" y="110" font-family="serif" font-size="24">
```

¹¹⁰テキストの基準の位置を変更することも可能である。


```
8      text-decoration="line-through">3. 文字列描画ABCabcgjpqy</text>
9  </svg>
```

これを Web ブラウザで表示すると右のようになる。

1.文字列描画ABCabcgjpqy
2.文字列描画ABCabcgjpqy
3.文字列描画ABCabcgjpqy

8.2.7.5 文字の回転

rotate 属性を指定して文字を回転表示する例 SVG03-5.svg を示す。

記述例：SVG03-5.svg

```
1  <svg xmlns="http://www.w3.org/2000/svg">
2    <title>テキスト</title>
3    <text x="10" y="30" font-family="serif" font-size="24" rotate="0">
4      >1. 文字列描画ABCabcgjpqy</text>
5    <text x="10" y="70" font-family="serif" font-size="24" rotate="15">
6      >2. 文字列描画ABCabcgjpqy</text>
7  </svg>
```

これを Web ブラウザで表示すると右のようになる。

1.文字列描画ABCabcgjpqy
2.文字列描画ABCabcgjpqy

8.2.8 パス：path 要素

path 要素を用いると、折れ線だけでなく楕円弧、ベジェ曲線¹¹¹ などを含んだ複雑な図形を描くことができる。

《 path 要素》

<path fill=フィルの設定 stroke=輪郭線の設定 stroke-width=線幅
d=描画コマンド/>

「描画コマンド」の通りにパスを描く。

「描画コマンド」はコマンドと引数を空白で区切って並べたものである。

書き方： ”コマンド 引数1 引数2 …”

「コマンド」と「引数」から成る並びは複数記述することもできる。

以下に各種の描画コマンドについて解説する。

8.2.8.1 直線の描画

path 要素の M コマンド (MoveTo) で座標位置を設定し、L コマンド (LineTo) で直線を描画する。

書き方： M x y

描画位置を座標 (x,y) に移動する。

書き方： L x y

現在の座標位置から (x,y) の位置に向けて直線を描き、描画位置をその座標に設定する。

L コマンドを複数実行した後、Z コマンドによって現在の座標と最初の座標を結び、閉じたパスにすることができる。Z コマンドには引数はない。

これらコマンドを使用して直線のパスを描くサンプル SVG02-5.svg を示す。

¹¹¹Paul de Casteljau (仏), Pierre Bézier (仏) らによって考案された曲線である。多くのベクターグラフィックス用ソフトウェアの曲線描画機能で採用されている。本書ではベジェ曲線についての解説は割愛する。

記述例：SVG02-5.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>パス1</title>
3   <!-- 開いたパス -->
4   <path d="M 20 20 L 100 20 L 100 100"
5         stroke="#000000" stroke-width="8" fill="#ffff00" />
6   <!-- 閉じたパス -->
7   <path d="M 120 20 L 200 20 L 200 100 Z"
8         stroke="#000000" stroke-width="8" fill="#ffff00" />
9   <!-- 相対座標指定で描くパス -->
10  <path d="M 220 20 l 80 0 l 0 80 Z"
11        stroke="#000000" stroke-width="8" fill="#ffff00" />
12 </svg>
```

これを Web ブラウザで表示すると図 69 のようになる。



図 69: SVG02-5.svg を Web ブラウザで表示したところ

■ コマンド記述の大文字／小文字の違い

path 要素の描画コマンドを記述する際、大文字のコマンドの場合は引数に与える座標は**絶対座標**、小文字のコマンドの場合は**相対座標**となる。ここで言う相対座標とは、現在の描画位置を基準にした横方向（x 座標）の変化量と縦方向（y 座標）の変化量を意味する。

未だ何も描画を実行していない段階で「m コマンド」（小文字）で描画開始位置を指定すると、原点 (0,0) に対する相対位置に設定される。

■ 水平／垂直の直線

L コマンド以外にも直線描画のコマンドがあり、H コマンドで水平線、V コマンドで垂直線が描画できる。

水平線： "H x"

垂直線： "V y"

H コマンドは現在座標から垂直位置を保って x に水平線を、V コマンドは水平位置を保って y に垂直線を描く。

先の SVG02-5.svg と同様の図形を H、V コマンドによって作図するサンプル SVG02-5-2.svg を示す。

記述例：SVG02-5-2.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>パス1（水平，垂直）</title>
3   <!-- 開いたパス -->
4   <path d="M 20 20 H 100 V 100"
5         stroke="#000000" stroke-width="8" fill="#ffff00" />
6   <!-- 閉じたパス -->
7   <path d="M 120 20 H 200 V 100 Z"
8         stroke="#000000" stroke-width="8" fill="#ffff00" />
9   <!-- 相対座標指定で描くパス -->
10  <path d="M 220 20 h 80 v 80 Z"
11        stroke="#000000" stroke-width="8" fill="#ffff00" />
12 </svg>
```

8.2.8.2 楕円弧

A コマンドによって楕円弧を描くことができる。

書き方： A 水平半径 垂直半径 傾き 弧の大小フラグ 回転方向フラグ x y

現在の座標を始点として、終点 (x,y) までを結ぶ楕円弧を描く。楕円弧は「水平半径」、「垂直半径」で大きさが指定された楕円の一部である。

始点と終点を結ぶ楕円弧は図 70 に示すように 4 通り存在し「弧の大小フラグ」、「回転方向フラグ」によってどの楕円弧を描くかが決定される。

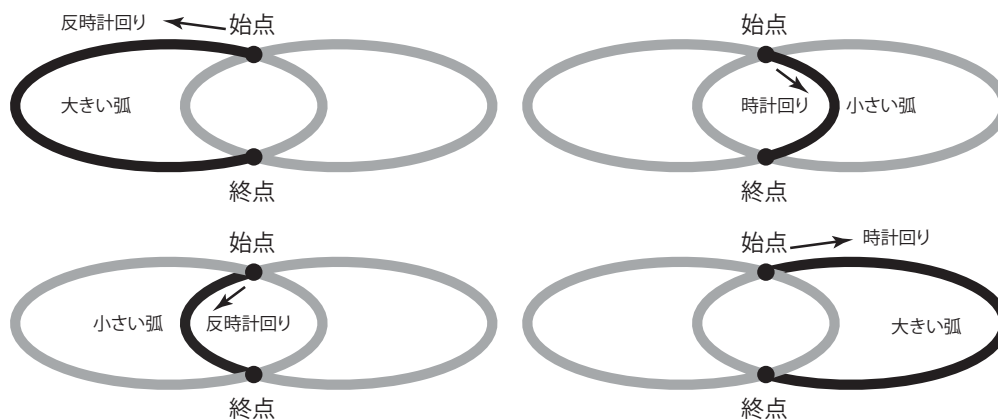


図 70: 楕円弧を決定する 4 つの条件 (弧の大小と回転方向)

「弧の大小フラグ」が 1 の場合に大きい方の楕円弧が、0 の場合に小さい方が選択される。また「回転方向フラグ」が 1 の場合に時計回りが、0 の場合に反時計回りが選択される。このことを示すサンプル SVG02-6.svg を示す。

記述例：SVG02-6.svg

```

1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>パス2</title>
3   <!-- 楕円弧：始点-終点(250,30)-(250,150)，水平半径150，垂直半径60 -->
4   <!-- 大きい方の楕円弧，反時計回り -->
5   <path d="M 250 50 A 150 60 0 1 0 250 150"
6         stroke="#000000" stroke-width="8" fill="#ffff00" />
7   <!-- 小さい方の楕円弧，時計回り -->
8   <path d="M 250 50 A 150 60 0 0 1 250 150"
9         stroke="#ff0000" stroke-width="8" fill="#00ff00" />
10 </svg>

```

これを Web ブラウザで表示すると図 71 のようになる。

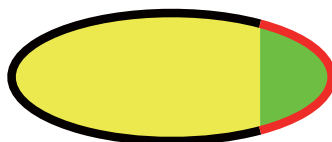


図 71: SVG02-6.svg を Web ブラウザで表示したところ

図 71 中の黒い楕円弧が「反時計回り」の「大きい方」の楕円弧，赤い楕円弧が「時計回り」の「小さい方」の楕円弧である。

小文字の「a コマンド」の場合は終点の指定が相対座標となる。先の SVG02-6.svg と同じ図形を「a コマンド」で実現するサンプルを SVG02-6-2.svg に示す。

記述例：SVG02-6-2.svg

```

1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>パス2</title>
3   <!-- 楕円弧：始点-終点(250,30)-(250,150)，水平半径150，垂直半径60 -->
4   <!-- 大きい方の楕円弧，反時計回り -->
5   <path d="M 250 50 a 150 60 0 1 0 0 100"
6         stroke="#000000" stroke-width="8" fill="#ffff00" />
7   <!-- 小さい方の楕円弧，時計回り -->
8   <path d="M 250 50 a 150 60 0 0 1 0 100"
9         stroke="#ff0000" stroke-width="8" fill="#00ff00" />
10 </svg>

```

■ 楕円弧の傾きについて

楕円弧を描く際の「傾き」は、当該楕円弧の元になる楕円の傾きの角度を意味する。10° 傾けた楕円の弧を描く例を SVG02-6-3.svg に示す。

記述例：SVG02-6-3.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>パス2</title>
3   <!-- 楕円弧：始点-終点(250,30)-(250,150)，水平半径150，垂直半径60 -->
4   <!-- 大きい方の楕円弧，反時計回り -->
5   <path d="M 250 50 a 150 60 10 1 0 0 100"
6         stroke="#000000" stroke-width="8" fill="#ffff00" />
7   <!-- 小さい方の楕円弧，時計回り -->
8   <path d="M 250 50 a 150 60 10 0 1 0 100"
9         stroke="#ff0000" stroke-width="8" fill="#00ff00" />
10 </svg>
```

これを Web ブラウザで表示すると図 72 のようになる。

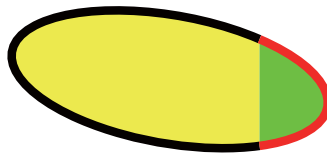


図 72: SVG02-6-3.svg を Web ブラウザで表示したところ

■ 始点、終点の設定が正しくない場合について

始点と終点が離れ過ぎて楕円弧が結べない場合は、適切に拡大された楕円弧が描画される。例えば SVG02-6-4.svg に示すような楕円弧について考える。

記述例：SVG02-6-4.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg">
2   <title>パス2</title>
3   <!-- 楕円弧：始点-終点(250,30)-(250,150)，水平半径150，垂直半径60 -->
4   <!-- 大きい方の楕円弧，反時計回り -->
5   <path d="M 250 50 a 150 60 0 1 0 0 150"
6         stroke="#000000" stroke-width="8" fill="#ffff00" />
7   <!-- 小さい方の楕円弧，時計回り -->
8   <path d="M 250 50 a 150 60 0 0 1 0 150"
9         stroke="#ff0000" stroke-width="8" fill="#00ff00" />
10 </svg>
```

この例では、始点と終点の距離が 150 である。しかし楕円弧の垂直半径が 120 であるので終点に届かない。この場合は図 72 のように、自動的に楕円弧が拡大されて描画される。

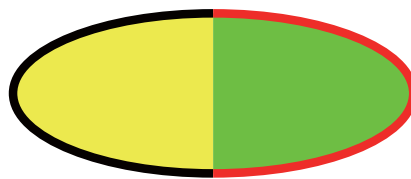


図 73: SVG02-6-4.svg を Web ブラウザで表示したところ

始点と終点が楕円の垂直半径に一致する形になっている。

※ 楕円弧の始点と終点が一致する場合は楕円弧が決定できないので描画されない。

8.2.9 画像データの配置

画像データを図形要素として SVG 中に配置するには image 要素を使用する。

書き方： <image x=横位置 y=縦位置 width=横幅 height=高さ xlink:href=画像リソース />

「画像リソース」（URL やパス）で示される画像データを、「横位置」と「縦位置」で表される座標に配置する。画像データは「横幅」と「高さ」に合う形で配置¹¹²される。

厳密には、image 要素は SVG 中に画像リソースのリンクを配置するものであり、対象の画像データそのものを SVG 中に組み込むものではない。

¹¹²画像のアスペクト比は保たれる。

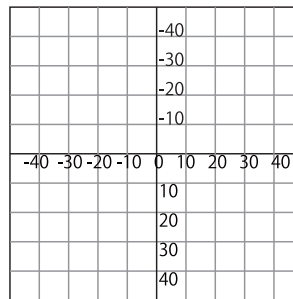
注意)

xlink:href 属性を使用するためには、XML の名前空間 <http://www.w3.org/1999/xlink> が必要となるので、svg 要素の開始タグに

```
xmlns:xlink="http://www.w3.org/1999/xlink"
```

と記述すること。

右の図のような PNG 画像データを配置する SVG を SVG04-03.svg に示す。



AxisPlane.png (200 × 200px)

記述例：SVG04-03.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
2   width="200" height="200" viewBox="-100 -100 200 200">
3   <title>画像の配置</title>
4   <image x="-100" y="-100" width="200" height="200" xlink:href="AxisPlane.png"/>
5 </svg>
```

8.2.10 transform 属性

図形要素に transform 属性を与えることで、当該図形要素を平行移動、回転、拡大縮小することができる。この属性には関数形式で各種の変換処理（表 54）を指定する。

表 54: transform 属性に指定する変換処理（一部）

記 述	解 説
translate(tx, ty)	図形要素を横方向に tx、縦方向に ty 並行移動する。ty は省略可能で、その場合は 0 と見做される。
rotate(a, cx, cy)	図形要素を (cx,cy) を中心に a° 回転する。cx, cy は省略可能で、その場合は原点中心の回転となる。
scale(mx, my)	原点を中心にして図形要素を横方向に mx 倍、縦方向に my 倍する。my は省略可能で、その場合は mx と同じと見做される。

次の SVG04-01.svg は、text 要素を平行移動、回転する例である。

記述例：SVG04-01.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg"
2   width="300" height="300" viewBox="0 0 300 300">
3   <title>平行移動と回転</title>
4   <text x="0" y="0" font-family="serif" font-weight="bold" font-size="24"
5     transform="translate(10,24)"
6   >文字の平行移動</text>
7   <text x="10" y="80" font-family="serif" font-weight="bold" font-size="24"
8     transform="rotate(30,70,70)"
9   >文字の傾斜</text>
10 </svg>
```

これを Web ブラウザで表示すると右のようになる。

文字の平行移動

文字の傾斜

次の SVG04-02.svg は、text 要素を拡大する例である。

記述例：SVG04-02.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg"
2   width="700" height="150" viewBox="0 0 700 150">
3   <title>拡大</title>
4   <text x="0" y="12" font-family="sans-serif" font-weight="bold" font-size="12"
```

```

5      fill="white" stroke="black" stroke-width="0.25" transform="scale(1,1)"
6    >文字の拡大表示</text>
7    <text x="0" y="12" font-family="sans-serif" font-weight="bold" font-size="12"
8      fill="white" stroke="black" stroke-width="0.25" transform="scale(2,2)"
9    >文字の拡大表示</text>
10   <text x="0" y="12" font-family="sans-serif" font-weight="bold" font-size="12"
11     fill="white" stroke="black" stroke-width="0.25" transform="scale(4,4)"
12   >文字の拡大表示</text>
13   <text x="0" y="12" font-family="sans-serif" font-weight="bold" font-size="12"
14     fill="white" stroke="black" stroke-width="0.25" transform="scale(8,8)"
15   >文字の拡大表示</text>
16 </svg>

```

これを Web ブラウザで表示すると右
ようになる。

文字の拡大表示

8.2.10.1 変換の重ね合わせ

transform 属性に与える変換は複数のものを重ね合わせることができる。次の SVG04-04.svg は並行移動を連鎖する例である。

記述例：SVG04-04.svg

```

1 <svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
2   width="200" height="200" viewBox="-50 -50 100 100">
3   <title>変換の重ね合わせ</title>
4   <image x="-50" y="-50" width="100" height="100" xlink:href="AxisPlane.png"/>
5   <circle cx="0" cy="0" r="5" fill="red"/>
6   <circle cx="0" cy="0" r="5" fill="green"
7     transform="translate(30,0)"/>
8   <circle cx="0" cy="0" r="5" fill="blue"
9     transform="translate(30,0) translate(0,30)"/>
10 </svg>

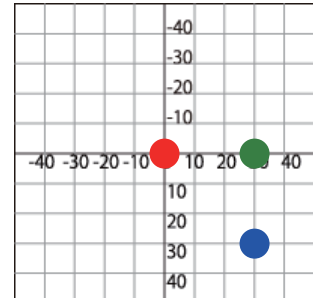
```

これは、中心の位置、半径とも同じ 3 つの circle 要素を描画する例である。translate による変換が連鎖することを確認できる。

これを Web ブラウザで表示すると右ようになる。

座標系の原点の解釈が変更される様子がわかる。

使用した画像は p.217 で取り上げた AxisPlane.png である。



8.2.10.2 use 要素による定義の展開

defs 要素の中に定義した図形要素の id 属性を指定して use 要素で展開することができる。

書き方：<use href="#"定義の id 属性" x=横位置のオフセット y=縦位置のオフセット (他の属性...) />

「x」「y」のオフセットを指定して defs 要素内に定義した図形要素を展開する。transform 属性を与えることもできる。

次の SVG05-1.svg は defs 要素内に定義した circle 要素を use 要素で展開する例である。

記述例：SVG05-1.svg

```

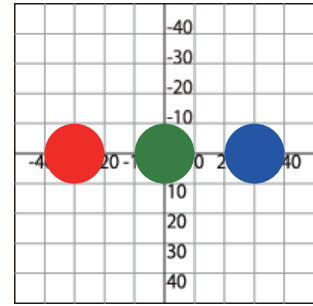
1 <svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
2   width="200" height="200" viewBox="-50 -50 100 100">
3   <defs>
4     <circle cx="0" cy="0" r="10" id="C"/>
5   </defs>
6   <title>図形要素の定義と再利用</title>
7   <image x="-50" y="-50" width="100" height="100" xlink:href="AxisPlane.png"/>
8   <use href="#C" x="-30" y="0" fill="red"/>
9   <use href="#C" fill="green"/>
10  <use href="#C" fill="blue" transform="translate(30,0)"/>
11 </svg>

```


これは、中心の座標 (0,0)、半径 10 の円に fill 属性と位置のオフセットを与えて描画するものである。

これを Web ブラウザで表示すると右のようになる。

使用した画像は p.217 で取り上げた AxisPlane.png である。



8.2.10.3 図形要素のグループ化

複数の図形要素は g 要素としてグループ化することができる。すなわち

```
<g id="名前">~</g>
```

の子要素として複数の図形要素を束ね、それを #名前 にて引用することができ、1つの図形要素のように扱うことができる。

次の SVG05-2.svg は defs 要素内に定義した g 要素を use 要素で展開する例である。

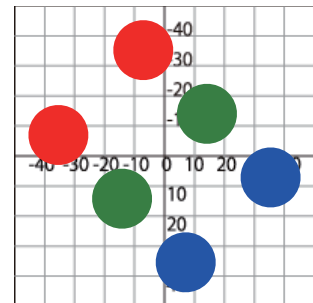
記述例：SVG05-2.svg

```
1 <svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
2   width="200" height="200" viewBox="-50 -50 100 100">
3   <defs>
4     <g id="g1">
5       <circle cx="-30" cy="0" r="10" fill="red"/>
6       <circle cx="0" cy="0" r="10" fill="green"/>
7       <circle cx="30" cy="0" r="10" fill="blue"/>
8     </g>
9   </defs>
10  <title>図形要素のグループ化</title>
11  <image x="-50" y="-50" width="100" height="100" xlink:href="AxisPlane.png"/>
12  <use href="#g1" transform="rotate(45) translate(0,20)"/>
13  <use href="#g1" transform="rotate(45) translate(0,-20)"/>
14 </svg>
```

これは、色の異なる3つの円を g 要素で束ねたものを defs 内に定義し、それを use で描画する例である。g 要素を描画する際に transform 属性で角度と位置を変換している。

これを Web ブラウザで表示すると右のようになる。

使用した画像は p.217 で取り上げた AxisPlane.png である。



9 プログラムライブラリ (1) : jQuery

jQuery は米国のプログラマであるジョン・レシグによって開発された JavaScript 用のプログラムライブラリである。jQuery は、CSS を操作する機能、イベントハンドリングに関する機能、DOM を操作する機能、グラフィカルな効果を実現する機能など、様々な機能を提供する。jQuery を HTML コンテンツに導入することによって、高度なコンテンツ操作の機能を簡便な形で記述できるようになる。

jQuery の本体と関連の情報は、下記の公式インターネットサイトから入手できる。

<https://jquery.com/>

本書では jQuery に関する最も初歩的な事柄について解説する。

9.1 jQuery の導入

jQuery は JavaScript で記述されたプログラムファイルとして提供されており、それを公式インターネットサイトからダウンロードして HTML に組み込んで使用することができる。

jQuery 本体のファイルは

jquery-3.7.1.js

のようなファイル名を持ち、ハイフンの後ろにバージョン番号が付いている。実際に jQuery を使用する際はもっと簡単な名前（例えば jquery.js）に変更しても良い。

jQuery は HTML に script 要素として読み込むことができる。

書き方： `<script type="text/javascript" src="jquery.js"></script>`

この例では、jQuery 本体のファイル jquery.js を読み込んでいる。

jQuery は、公式インターネットサイトの CDN ¹¹³ からネットワーク経由で読み込むこともでき、その場合は取得した jQuery の URL を script 要素の src 属性に与える。CDN の具体的な URL は公式サイトを参照のこと。

9.2 jQuery オブジェクト

jQuery ライブラリが提供する主たるオブジェクトは **jQuery オブジェクト** であり、それは HTML の各種要素を独自の形式として扱うためのものである。jQuery オブジェクトを生成するための関数（**jQuery オブジェクト生成関数**¹¹⁴）はライブラリと同名の「jQuery」であり、

jQuery(引数)

のように記述する。この「引数」の部分に HTML の記述やセレクタの記述などを与えるとそれに対応する jQuery オブジェクトを生成して返す。この関数はエイリアスとして \$ という記号も使うことができ、この記号を記述することの方が一般的である。

jQuery オブジェクト生成関数は **Function オブジェクト** であり、様々なプロパティを持つ。例えば、\$.fn.jquery から当該ライブラリのバージョン番号を取得することができる。

jQuery ライブラリが提供する機能は大まかに次のような 5 つのカテゴリに分類することができる。

1. 関数の起動
2. HTML 要素の新規作成
3. DOM の編集
4. イベントハンドリングの登録
5. Ajax 関連のリクエスト処理

ここでは例を挙げて上記 1~4 について概観する。

次のようなサンプルを Web ブラウザで表示し、コンソールで jQuery オブジェクトの動作を確認する。

¹¹³コンテンツデリバリーネットワーク（content delivery network）。情報のリソースをネットワーク経由で配信する最適化されたシステム。

¹¹⁴単に **jQuery 関数**と呼ぶこともある。

記述例：jQuery00.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>jQuery00</title>
6     <script type="text/javascript" src="jquery.js"></script>
7   </head>
8   <body>
9     <p id="tx1">jQueryを読み込むサンプルです. </p>
10  </body>
11 </html>
```

このコンテンツを Web ブラウザで読み込むと「jQueryを読み込むサンプルです。」と表示される。

次のようにして、Web ブラウザのコンソールで jQuery のバージョン番号を調べることができる。

例. jQuery のバージョン番号の確認

```
> $.fn.jquery 
'3.7.1' ←使用中の jQuery のバージョン番号
> jQuery.fn.jquery  ←このように記述しても良い
'3.7.1'
```

9.2.1 関数の起動

jQuery オブジェクト生成関数の引数に関数を与えるとそれを起動する。

例. jQuery オブジェクトによる関数の起動（先の例の続き）

```
> function f1() { console.log("サンプル"); }  ←関数 f1 を定義
undefined ←上記の処理結果
> r = $( f1 )  ←それを実行
jQuery.fn.init {0: document, length: 1} ←上の式の戻り値 (jQuery オブジェクト)
サンプル ←関数 f1 の実行による出力
```

処理結果として戻り値が r に得られている。この場合は、1つの要素を持つ配列のような形式のデータ構造であり、その要素を参照する例を次に示す。

例. 戻り値の jQuery オブジェクトの要素を参照する（先の例の続き）

```
> r[0] 
#document ( ... ) ←HTML 文書を表すオブジェクト
```

得られた要素は jQuery00.html の内容全体を意味する HTMLDocument オブジェクトである。

勿論、jQuery オブジェクト生成関数の引数に関数式を与えても良い。

例. 上と同じ処理（先の例の続き）

```
> $( function() { console.log("サンプル"); } ) 
jQuery.fn.init {0: document, length: 1} ←上の式の戻り値
サンプル ←関数式の実行による出力
```

この方法による関数の起動は、当該 HTML 文書が Web ブラウザに読み込まれて DOM の操作が可能になった後になる。すなわち、Web ブラウザが HTML を読み込み DOM の構築をしている最中は \$(関数) で関数が起動されず、DOM の構築が完了した後で起動される。このことに関しては後の「9.9 Web アプリケーション実装に関すること」(p.236) で更に解説する。

9.2.2 HTML 要素へのアクセス

\$(セレクタ) で得られるオブジェクトは「セレクタ」が示す DOM 内のオブジェクトを指すものであり、それに対して各種の jQuery メソッドを実行することができる。

例. p 要素のオブジェクトを参照 (先の例の続き)

```
> $( "#tx1" ) Enter    ← p 要素のオブジェクト  
jQuery.fn.init {0: p#tx1, length: 1}    ← jQuery オブジェクト
```

このように HTML の要素が jQuery オブジェクトとして扱われることがわかる。これに対して jQuery のメソッドが実行できる。(次の例)

例. p 要素のテキストを取得する (先の例の続き)

```
> $( "#tx1" ).text() Enter  
jQuery を読み込むサンプルです。    ←得られたテキスト
```

これは jQuery のメソッド text で id="tx1" の要素のテキストを取得する例である。

text メソッドは取得したテキスト (文字列) を返すが、多くの jQuery メソッドは jQuery オブジェクトを返す。これを応用すると、同一のオブジェクトに異なるメソッドを次々と実行することができる。このようなメソッドチェーンにより、たくさんの処理を簡潔に記述することが可能になる。

9.2.3 イベントハンドリングの登録

jQuery には、簡便な記述でイベントハンドリングを実現する各種のメソッドが提供されている。次に示す例は、jQuery00.html の p 要素にマウスクリックのイベントハンドリングを登録する例である。

例. マウスクリックのイベントハンドリングの登録 (先の例の続き)

```
> $( "#tx1" ).click( function() { console.log("クリックされました"); } ) Enter  
jQuery.fn.init {0: p#tx1, length: 1}    ←上の式の戻り値
```

このように click メソッドで、セクタで指定した HTML 要素にマウスクリックのイベントハンドリングを登録できる。この後、当該要素をクリックすると Web ブラウザのコンソールに「クリックされました」と表示される。

9.2.4 HTML 要素の生成と DOM への追加

jQuery では \$(HTML 要素の記述) と記述すると、その HTML 要素を表す jQuery オブジェクトを生成する。また append メソッドでそれを DOM の要素として追加することができる。

例. p 要素を新規作成して DOM に追加する (先の例の続き)

```
> x = $( "<p id='tx2' class='c1'>2つ目の p 要素です. </p>" ) Enter    ←要素の作成  
jQuery.fn.init {0: p#tx2, length: 1}    ←上の式の戻り値  
> $( "body" ).append(x) Enter    ← body 要素の最後の子要素として追加  
jQuery.fn.init {0: body, length: 1, prevObject: j...y.fn.init}    ←戻り値 (body 要素)
```

この処理の直後、Web ブラウザの 2 行目に「2つ目の p 要素です。」と表示される。append メソッドの引数には直接的に文字列として HTML 要素を与えることもできる。

例. 更に簡単な記述で DOM に HTML 要素を追加する (先の例の続き)

```
> $( "body" ).append( "<p id='tx3' class='c1'>The third P element</p>" ) Enter    ←要素の追加  
jQuery.fn.init {0: body, length: 1, prevObject: j...y.fn.init}    ←戻り値 (body 要素)
```

この処理の直後、Web ブラウザの 3 行目に「The third P element」と表示される。

9.2.5 jQuery オブジェクトが保持する HTML 要素

jQuery オブジェクトは 1 つもしくは複数の HTML 要素を保持することができる。先の例では変数 x に

```
jQuery.fn.init {0: p#tx2, length: 1}
```

という jQuery オブジェクトが得られているが、このインデックス 0 の位置の要素はセクタ #tx2 が示す HTML 要素であることがわかる。しかも jQuery オブジェクトの要素としての HTML 要素のクラスは HTMLElement である。

先の例では jQuery00.html の body 要素に 2 つの p 要素を追加した。その場合に class='c1' の属性が与えられているが、このクラス属性で要素を取得する場合について考える。

例. クラスを指定して複数の HTML 要素を取得する (先の例の続き)

```
> e = $( ".c1" ) [Enter] ←複数要素の取得          ↓得られたjQuery オブジェクト
jQuery.fn.init {0: p#tx2.c1, 1: p#tx3.c1, length: 2, prevObject: j...y.fn.init}
```

e に得られた jQuery オブジェクトは 2 つの HTML 要素を保持している. それら個々の要素を取り出すには eq メソッドを使用する.

書き方; jQuery オブジェクト.eq(インデックス)

「jQuery オブジェクト」内の「インデックス」で示される要素を返す.

例. e の要素を調べる (先の例の続き)

```
> e.length [Enter] ←要素の個数を調べる
2          ← 2 個
> e.eq(0) [Enter] ←インデックス位置 0 の要素を調べる
jQuery.fn.init {0: p#tx2.c1, length: 1, prevObject: j...y.fn.init}
> e.eq(1) [Enter] ←インデックス位置 1 の要素を調べる
jQuery.fn.init {0: p#tx3.c1, length: 1, prevObject: j...y.fn.init}
```

eq メソッドとは別に, 添字 [n] で参照すると HTMLElement オブジェクトの形で得られる. (次の例)

例. e の要素を HTMLElement オブジェクト として参照 (先の例の続き)

```
> e[0] [Enter] ←インデックス位置 0 の要素を調べる
<p id="tx2" class="c1">2 目目の p 要素です. </p> ← HTMLElement オブジェクト
> e[1] [Enter] ←インデックス位置 1 の要素を調べる
<p id="tx3" class="c1">The third P element</p> ← HTMLElement オブジェクト
```

9.2.6 jQuery で扱う HTML 要素

jQuery の各種メソッドに HTML の記述を与える場合は, jQuery オブジェクト, 文字列表現の他に HTMLElement クラスのオブジェクトを与えても良い.

9.2.7 jQuery における window, document

jQuery では \$(window), \$(document) として window オブジェクト, document オブジェクトを扱うことができる.

9.3 CSS 属性へのアクセス

jQuery では css メソッドによって, HTML 要素の CSS 属性への値の設定や参照ができる.

書き方; \$(セレクタ).css(CSS 属性, 値)

「セレクタ」で示される対象の「CSS 属性」に「値」を設定する.

例. #id="tx1" の要素の色を赤に設定する (先の例の続き)

```
> $( "#tx1" ).css( "color", "red" ) [Enter]
jQuery.fn.init {0: p#tx1, length: 1} ←上の式の戻り値
```

この処理の後, #id="tx1" の p 要素の色が赤に設定され, 当該部分の表示が

jQuery を読み込むサンプルです.

となる.

セレクタで指定した要素の CSS 属性の値を取得する場合にも css メソッドが使える. この場合は css メソッドに属性項目の文字列を 1 つだけ与える.

書き方; \$(セレクタ).css(CSS 属性)

「セレクタ」で示される対象の「CSS 属性」を返す.

例. CSS の "color" 属性の値を参照する (先の例の続き)

```
> $( "#tx1" ).css( "color" ) [Enter] ← CSS の属性 "color" の値を参照
'rgb(255, 0, 0)'          ←参照結果
```

9.4 イベントハンドリングのためのメソッド

先の「9.2.3 イベントハンドリングの登録」(p.222) でマウスクリックイベントを登録する方法を解説したが、他のイベントに関しても表 55 にあるようなメソッドが存在する。

表 55: イベントハンドリング登録のためのメソッド (一部)

メソッド	解 説	メソッド	解 説
click	クリックされた	dblclick	ダブルクリックされた
mouseenter	マウスが要素の上に入った	mouseleave	マウスが要素から出た
mousedown	マウスボタンが押された	mouseup	マウスボタンが離された
hover	マウスが要素の上で動いた	scroll	スクロールされた
focus	要素がフォーカスを得た	blur	要素がフォーカスを失った
keydown	キーが押された	keyup	キーが離された
keypress	キーが押されている		
change	値が変更された	submit	送信された (form 要素)

注) input イベントに対応するメソッドは無い。

表 55 にあるようなメソッドを次のように記述して実行する。

書き方: \$(セレクトラ). イベントに対応するメソッド (関数)

表 55 のメソッドとは別に、on メソッドでもイベントハンドリングの登録ができる。

書き方: \$(セレクトラ).on(イベント, 関数)

「イベント」には表 55 のメソッド名が指定できる他、input イベントなど、より多くのイベント名が指定できる。

例. on メソッドによるイベントハンドリングの登録 (先の例の続き)

```
> $( "#tx1" ).on( "dblclick", function() { console.log("ダブルクリック"); } ) Enter
jQuery.fn.init {0: p#tx1, length: 1} ←上の式の戻り値
```

この後、id="tx1" の p 要素をダブルクリックすると Web ブラウザのコンソールに「ダブルクリック」と表示される。

注意) on メソッドで正常にハンドリングできないケースもある。その場合は表 55 のメソッドを使うこと。

9.5 HTML 要素の属性へのアクセス

jQuery の attr メソッドによって HTML 要素の属性にアクセスできる。

書き方: \$(セレクトラ).attr(属性, 値)

「セレクトラ」で示される対象の「属性」に「値」を設定し、その対象を返す。第 2 引数を省略すると「属性」の値を返す。

注意) 動的に変化する属性にアクセスするには後に解説する val, prop, data などのメソッドを使用すること。

attr メソッドによる処理を、次のサンプル jQuery04.html を用いて例示する。

記述例: jQuery04.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>jQuery04</title>
6     <script type="text/javascript" src="jquery.js"></script>
7   </head>
8   <body><input id="e1"></body>
9 </html>
```

これを Web ブラウザで表示するとテキストフィールド が表示¹¹⁵ される。この状態で Web ブラウザのコンソールを開き次のような処理を行う。

¹¹⁵input 要素の type 属性のデフォルトは "text" である。

例. id="e1" の input 要素の各種属性の設定¹¹⁶

```
> $("#e1").attr("type","range").attr("min","0").attr("max","1000") Enter ←各種属性の設定
jQuery.fn.init {0: input#e1, length: 1}
> $("#e1").attr("step","5").attr("value","700") Enter ←同上
jQuery.fn.init {0: input#e1, length: 1}
> $("#e1")[0] Enter ←input 要素を確認
<input id="e1" type="range" min="0" max="1000" step="5" value="700"> ←変更された input 要素
```

この処理により input 要素がテキストフィールドからスライダーに変わり、Web ブラウザの表示が



となる。

9.5.1 属性値の動的な扱い

attr メソッドで HTML 要素の属性値を読み取ると、attr メソッドで設定した際の値が得られる。ユーザが UI を操作して変更された値（最新の値）を読み取るには val メソッドを使用すると良い。（次の例）

例. つまみを動かした後で input 要素の value 属性の値を取得する（先の例の続き）

```
> $("#e1").attr("value") Enter ← value 属性値の参照 (1)
'700' ←設定時のまま
> $("#e1").val() Enter ← value 属性値の参照 (2)
'310' ←最新の値
```

以上のことから、attr メソッドは HTML の文書としての構成要素の属性を設定するもの（初期設定するもの）であり、事後の動的な変更とその値の参照には別の方法が必要であるということが理解できる。

val メソッドは input 要素を含む form 要素の値を動的に設定、参照する場合に使用する。

書き方： \$(セレクトア).val(値)

form 要素に「値」を設定し、その要素を返す。引数を省略した場合は値を参照して返す。

val メソッドとは別に、prop メソッドも存在する。これは、HTML 要素の開始タグに記述する属性にアクセスするためのもので、これを用いて動的に属性値を設定、参照することもできる。

書き方： \$(セレクトア).prop(属性, 値)

「セレクトア」で示される対象の「属性」に「値」を設定し、その対象を返す。第 2 引数を省略すると「属性」の値を返す。

例. prop メソッドによる属性値の設定と参照（先の例の続き）

```
> $("#e1").prop("value","100") Enter ← value 属性値の設定
jQuery.fn.init {0: input#e1, length: 1}
> $("#e1").prop("value") Enter ← value 属性値の参照
'100'
```

9.5.1.1 チェックボックスの扱い

チェックボックス（<input type="checkbox">）の状態は論理属性 checked プロパティの論理値（true / false）で決まる。このことを次のサンプル jQuery05-1.html を用いて確認する。

記述例：jQuery05-1.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>jQuery05-1</title>
6     <script type="text/javascript" src="jquery.js"></script>
7   </head>
8   <body>
9     <input type="checkbox" name="c1" id="c1" checked>
10    <label for="c1">項目 c1</label>
```

¹¹⁶この処理を 1 行で行っても良い。

```

11     <input type="checkbox" name="c2" id="c2">
12     <label for="c2">項目 c2</label>
13 </body>
14 </html>

```

これを Web ブラウザで表示すると ☒ 項目c1 ☐ 項目c2 と表示される。この状態で Web ブラウザのコンソールを開いていくつかの処理を行う。

例. チェックボックスの状態の確認

```

> $( "#c1" ).prop("checked")  ←左のチェックボックスは
true                          ←チェックされている
> $( "#c2" ).prop("checked")  ←右のチェックボックスは
false                         ←チェックされていない

```

ここで注意すべきことは、得られる値は文字列ではなく論理値であることである。チェックボックスの状態を設定する際も論理値を与える。

例. チェックボックスの状態の設定（先の例の続き）

```

> $( "#c1" ).prop("checked",false)  ←左のチェックを外す
jQuery.fn.init {0: input#c1, length: 1}
> $( "#c2" ).prop("checked",true)  ←右にチェックする
jQuery.fn.init {0: input#c2, length: 1}

```

この処理の結果 Web ブラウザの表示は ☐ 項目c1 ☒ 項目c2 となる。

9.5.1.2 ラジオボタンの扱い

ラジオボタン（<input type="radio">）の状態は**論理属性** checked プロパティの論理値（true / false）で決まる。このことを次のサンプル jQuery05-2.html を用いて確認する。

記述例：jQuery05-2.html

```

1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>jQuery05-2</title>
6     <script type="text/javascript" src="jquery.js"></script>
7   </head>
8   <body>
9     <input type="radio" name="r1" value="値1" id="r11" checked>
10    <label for="r11">項目 r11</label>
11    <input type="radio" name="r1" value="値2" id="r12">
12    <label for="r12">項目 r12</label>
13    <input type="radio" name="r1" value="値3" id="r13">
14    <label for="r13">項目 r13</label>
15  </body>
16 </html>

```

これを Web ブラウザで表示すると ☒ 項目r11 ☐ 項目r12 ☐ 項目r13 と表示される。この状態で Web ブラウザのコンソールを開いていくつかの処理を行う。

ラジオボタンが押されているかどうかを調べる処理を示す。

例. ラジオボタンの状態の調査

```

> $( "#r11" ).prop("checked")  ←左のボタンの状態を調べる
true                          ← on
> $( "#r12" ).prop("checked")  ←中央のボタンの状態を調べる
false                         ← off
> $( "#r13" ).prop("checked")  ←右のボタンの状態を調べる
false                         ← off

```

これはチェックボックスの場合と同じ手法である。ラジオボタンは複数の要素から成るもので、その内 1 つのみが押される UI である。次の方法を用いると、押されているボタンの値（value 属性）を 1 行で求めることができる。

例. 押されているボタンの値を求める (先の例の続き)

```
> $( "input[name='r1']:checked" ).val() ☐ Enter ←チェックされているボタンの値を調べる  
' 値 1'
```

UI の操作に依らず, JavaScript のプログラムでラジオボタンを切り替える方法の 1 例を次に示す.

例. ラジオボタンの切り替え (先の例の続き)

```
> $( "input[name='r1']" ).prop("checked",false) ☐ Enter ←一旦全てのボタンを off にして  
jQuery.fn.init {0: input#r11, 1: input#r12, 2: input#r13, length: 3,  
  prevObject: j...y.fn.init}  
> $( "#r13" ).prop("checked",true) ☒ Enter ←右のボタンを on にする  
jQuery.fn.init {0: input#r13, length: 1}
```

これは, ラジオボタンの全要素の状態を一旦 off にした後で, 特定の要素を on にするという手法である. この処理の結果 Web ブラウザの表示は ☐ 項目r11 ☐ 項目r12 ☒ 項目r13 となる.

9.5.1.3 データ属性の扱い

jQuery では, 属性名が「data-」ではじまるデータ属性には data メソッドでアクセスすることが推奨されている.

書き方: `$(セレクタ).data(キー, 値)`

データ属性名「data-**キー**」に「**値**」を設定し, 対象要素を返す.「**値**」を省略すると値を参照して返す.

data メソッドを用いた処理をサンプル jQuery05-4.html を用いて例示する.

記述例: jQuery05-4.html

```
1 <!DOCTYPE html>  
2 <html lang="ja">  
3   <head>  
4     <meta charset="utf-8">  
5     <title>jQuery05-4</title>  
6     <script type="text/javascript" src="jquery.js"></script>  
7   </head>  
8   <body><p data-v1=" 値 1" id="p1">p要素</p></body>  
9 </html>
```

このサンプルでは p 要素のデータ属性 data-v1 に " 値 1" が設定されている. このサンプルを Web ブラウザで表示すると「p 要素」と表示される. その状態で Web ブラウザのコンソールで次のような処理を行う.

例. データ属性を prop メソッドで参照する試み

```
> $( "#p1" ).prop("data-v1") ☐ Enter  
undefined ←値が得られない
```

これは Google Chrome ブラウザによる実行例であるが, データ属性の値が得られていないことがわかる. 次に data メソッドによる実行例を示す.

例. data メソッドによるデータ属性へのアクセス (先の例の続き)

```
> $( "#p1" ).data("v1") ☐ Enter ←値の参照が  
' 値 1' ←できる  
> $( "#p1" ).data("v1","新しい値 1") ☐ Enter ←値の変更  
jQuery.fn.init {0: p#p1, length: 1}  
> $( "#p1" ).data("v1") ☐ Enter ←新たな値の確認  
' 新しい値 1' ←変更できている
```

ただしこれは動的な変更なので, 元の p 要素自体は変化しない. (次の例)

例. 元の p 要素の確認 (先の例の続き)

```
> $( "#p1" )[0] ☐ Enter ←確認  
<p data-v1="値 1" id="p1">p要素</p> ←変化していない
```

9.5.2 class 属性の扱い

jQuery は HTML 要素の class 属性を操作するためのメソッドを提供する. (表 56)

表 56: class 属性を操作するためのメソッド (一部)

メソッド	解 説	戻り値
addClass(C)	対象の class 属性に C を加える.	対象の要素
removeClass(C)	対象 class 属性から C を除去する.	対象の要素
toggleClass(C)	対象の class 属性に C があれば除去し、無ければ加える. (クラス名の切り替え)	対象の要素

次のサンプル jQuery05-3.html を用いて表 56 のメソッドの実行を例示する.

記述例: jQuery05-3.html

```

1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>jQuery05-3</title>
6     <script type="text/javascript" src="jquery.js"></script>
7   </head>
8   <body><p>p要素</p></body>
9 </html>

```

これは body 要素内に p 要素が1つ存在するコンテンツで、これを Web ブラウザで表示すると「p 要素」と表示される. この状態で Web ブラウザのコンソールを開いて処理の例を示す.

例. p 要素にクラス名を加える

```

> $( "p" ).addClass("c1") [Enter]   ←クラス名の追加
jQuery.fn.init {0: p.c1, length: 1, prevObject: j...y.fn.init}
> $( "p" )[0] [Enter]               ←処理後の p 要素の確認
<p class="c1">p要素</p>             ←クラス名が設定されている

```

これは p 要素にクラス名 "c1" を与える例である. 対象要素が既に持っているクラス名を与えようとした場合は対象は変化しない.

次に, p 要素のクラス名を除去する例を示す.

例. p 要素のクラス名を除去する (先の例の続き)

```

> $( "p" ).removeClass("c1") [Enter] ←クラス名の除去
jQuery.fn.init {0: p, length: 1, prevObject: j...y.fn.init}
> $( "p" )[0] [Enter]               ←処理後の p 要素の確認
<p class>p要素</p>                  ←クラス名が除去されている

```

除去の結果, クラス名を全く持たない状態になっても class 属性自体は残る. 存在しないクラス名を除去しようとした場合は対象は変化しない.

例. 空のクラス名 (先の例の続き)

```

> $( "p" ).prop("class") [Enter]   ←クラス属性の確認
''                                  ←空である

```

次に, toggleClass メソッドの実行例を示す.

例. toggleClass の実行 (先の例の続き)

```

> $( "p" ).toggleClass("c1") [Enter]   ←toggleClass 実行
jQuery.fn.init {0: p.c1, length: 1, prevObject: j...y.fn.init}
> $( "p" )[0] [Enter]                 ←処理後の p 要素の確認
<p class="c1">p要素</p>                ←クラス名が追加されている
> $( "p" ).toggleClass("c1"); [Enter] ←再度 toggleClass 実行
  $( "p" )[0] [Enter]                 ←処理後の p 要素の確認
<p class>p要素</p>                    ←クラス名が除去されている

```

HTML 要素は複数のクラス名を class 属性に保持することができる.

例. 複数のクラス名を与える (先の例の続き)

```
> $( "p" ).addClass("c1").addClass("c2").addClass("c3").addClass("c4") Enter ←クラス名の追加
jQuery.fn.init {0: p.c1.c2.c3.c4, length: 1, prevObject: j...y.fn.init}
> $( "p" )[0] Enter ←処理後の p 要素の確認
<p class="c1 c2 c3 c4">p 要素</p> ← 4 つのクラス名を持つ
```

class 属性が複数のクラス名を保つ場合も addClass, removeClass, toggleClass メソッドは機能する. (次の例)

例. 複数のクラス名がある場合の処理 (先の例の続き)

```
> $( "p" ).removeClass("c4") Enter ←クラス名 c4 の除去
jQuery.fn.init {0: p.c1.c2.c3, length: 1, prevObject: j...y.fn.init}
> $( "p" )[0] Enter ←処理後の p 要素の確認
<p class="c1 c2 c3">p 要素</p> ←除去されている
> $( "p" ).toggleClass("c2") Enter ← c2 を切り替え
jQuery.fn.init {0: p.c1.c3, length: 1, prevObject: j...y.fn.init}
> $( "p" )[0] Enter ←処理後の p 要素の確認
<p class="c1 c3">p 要素</p> ← c2 が除去されている
```

9.6 DOM の操作

DOM の操作のためのメソッドの内, 特に基本的なものを表 57 に示す.

表 57: DOM 操作のためのメソッド (一部)

メソッド	解 説	戻り値
append(E)	要素 E を対象の子要素の最後に追加する.	対象のオブジェクト
prepend(E)	要素 E を対象の子要素の最初に追加する.	対象のオブジェクト
before(E)	要素 E を対象の要素の直前に挿入する.	対象のオブジェクト
after(E)	要素 E を対象の要素の直後に挿入する.	対象のオブジェクト
remove()	対象の要素を削除する.	削除したオブジェクト
empty()	対象の要素の子要素を全て削除する.	対象のオブジェクト

jQuery による DOM の操作の方法を, サンプル jQuery02.html を用いて例示する.

記述例: jQuery02.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>jQuery02</title>
6     <style>
7       div, p { margin:2px; padding:2px;
8                 font-size:14px; line-height:16px;
9                 border:solid 1px black; }
10      div { width:100px; }
11    </style>
12    <script type="text/javascript" src="jquery.js"></script>
13  </head>
14  <body>
15  </body>
16 </html>
```

これは body 要素が空なので Web ブラウザで表示しても空白ページとなる. 以下で, Web ブラウザのコンソールによる作業を通して DOM が操作される様子を確認する.

9.6.1 要素の追加

例. append による要素の追加

```
> $( "body" ).append( "<div id='d3'>div 要素 3</div>" ) Enter div要素3
jQuery.fn.init {0: body, length: 1, prevObject: j...y.fn.init}
```

この処理によって、Web ブラウザの表示が右に示すようなものとなる。append メソッドは引数に与えた要素を対象要素の子要素の最後に追加する。従って再度同様の処理を行うと id="d3" の要素の次に追加される。(次の例)

例. 再度 append で要素を追加 (先の例の続き)

```
> $( "body" ).append( "<div id='d5'>div 要素 5</div>" )   
jQuery.fn.init {0: body, length: 1, prevObject: j...y.fn.init}
```

div要素3
div要素5

これを実行すると右のような表示となる。

prepend メソッドは引数に与えた要素を対象要素の子要素の最初に追加する。(次の例)

例. prepend による要素の追加 (先の例の続き)

```
> $( "body" ).prepend( "<div id='d1'>div 要素 1</div>" )   
jQuery.fn.init {0: body, length: 1, prevObject: j...y.fn.init}
```

div要素1
div要素3
div要素5

この処理によって、Web ブラウザの表示が右に示すようなものとなる。

before メソッドは引数に与えた要素を対象要素の直前に追加する。(次の例)

例. before による要素の追加 (先の例の続き)

```
> $( "#d3" ).before( "<div id='d2'>div 要素 2</div>" )   
jQuery.fn.init {0: div#d3, length: 1}
```

div要素1
div要素2
div要素3
div要素5

この処理によって、Web ブラウザの表示が右に示すようなものとなる。

after メソッドは引数に与えた要素を対象要素の直後に追加する。(次の例)

例. after による要素の追加 (先の例の続き)

```
> $( "#d3" ).after( "<div id='d4'>div 要素 4</div>" )   
jQuery.fn.init {0: div#d3, length: 1}
```

div要素1
div要素2
div要素3
div要素4
div要素5

この処理によって、Web ブラウザの表示が右に示すようなものとなる。

これまでに解説したメソッドを使って、id="d3" の div 要素に対して次のように要素の追加を行う。

例. div 要素への追加 (先の例の続き)

```
> $( "#d3" ).append( "<p id='p32'>p 要素 32</p>" );   
$( "#d3" ).prepend( "<p id='p31'>p 要素 31</p>" );   
$( "#d3" ).append( "テキスト" )   
jQuery.fn.init {0: div#d3, length: 1}
```

div要素1
div要素2
p要素31
div要素3
p要素32
テキスト
div要素4
div要素5

この処理によって、Web ブラウザの表示が右に示すようなものとなる。

表 57 の append, prepend, before, after と類似のメソッド (表 58) がある。

表 58: DOM 操作のためのメソッド (一部)

メソッド	解 説	戻り値
appendTo(E)	対象を要素 E の子要素の最後に追加する。	対象のオブジェクト
prependTo(E)	対象を要素 E の子要素の最初に追加する。	対象のオブジェクト
insertBefore(E)	対象を要素 E の直前に挿入する。	対象のオブジェクト
insertAfter(E)	対象を要素 E の直後に挿入する。	対象のオブジェクト

これらのメソッドは表 57 のメソッドと比較すると、実行対象と引数の関係が逆になっている。ただし注意するべき点もあり、先の例で示した

```
$( "#d3" ).append( "テキスト" )
```

と同じ処理を

```
$( "テキスト" ).appendTo( "#d3" )
```

として実行することはできない。

9.6.2 要素の削除

remove メソッドを使用して要素を削除する例を示す。

例. 指定した要素の削除（先の例の続き）

```
> r = $( "#p32" ).remove() 
jQuery.fn.init {0: p#p32, length: 1}
```

この処理によって id="#p32" の p 要素が削除され、Web ブラウザの表示が右に示すようなものとなる。remove メソッドは削除した要素を返す。上の例では削除された要素は r に受け取られている。（次の例）

div要素1
div要素2
p要素31
div要素3テキスト
div要素4
div要素5

例. remove メソッドの戻り値（先の例の続き）

```
> r  ←内容確認
jQuery.fn.init {0: p#p32, length: 1} ←削除された要素
> r[0]  ←更に内部
<p id="p32">p 要素 32</p> ← HTMLElement
empty メソッドを使用して、指定した要素の子要素を全て削除する例を示す。
```

例. 指定した要素を空にして新たな内容を与える（先の例の続き）

```
> $( "#d3" ).empty().append( "新しいテキスト" ) 
jQuery.fn.init {0: div#d3, length: 1}
```

この処理によって id="#d3" の div 要素が一旦空にされた後「新しいテキスト」というテキストが子要素に追加される。この処理の結果、Web ブラウザの表示を右のようになる。

div要素1
div要素2
新しいテキスト
div要素4
div要素5

9.6.3 子要素、親要素、内部要素の取得

指定した要素の子要素、親要素を取得するメソッドを表 59 に示す。

表 59: 子要素、親要素、内部要素の操作のためのメソッド（一部）

メソッド	解 説	戻り値
children()	対象の直下の子要素から全ての HTML 要素を取得する。	取得した子要素
contents()	対象の直下の全ての子要素を取得する。	取得した子要素
parent()	対象の親要素を取得する。	取得した親要素
html()	対象のコンテンツを HTML 文書形式の文字列として取得する。	取得したコンテンツ
html(H)	文字列で表現された HTML 文書 H で対象要素のコンテンツを上書きする。	対象のオブジェクト
text()	対象のコンテンツをテキスト形式（文字列）で取得する。	取得したコンテンツ
text(T)	対象要素のコンテンツをテキスト（文字列）T で上書きする。	対象のオブジェクト

次のサンプル jQuery03.html を使用して表 59 のメソッドの使用例を示す。

記述例：jQuery03.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>jQuery03</title>
6     <script type="text/javascript" src="jquery.js"></script>
7   </head>
8   <body>
9     テキスト1
10    <p id="p1">p 要素1 </p>
11    <!-- コメント -->
12    <p id="p2">p 要素2 </p>
13    テキスト2
14  </body>
15 </html>
```

これを Web ブラウザで表示すると右のようになる。その状態で Web ブラウザのコンソールで各種メソッドを実行する例を示す。

テキスト1
p要素1
p要素2
テキスト2

例. body の直下の子要素の内, HTML 要素を全て取得する

```
> c1 = $( "body" ).children() [Enter] ←子要素の内, HTML 要素を取得
jQuery.fn.init {0: p#p1, 1: p#p2, length: 2, prevObject: j...y.fn.init} ←得られたもの
> c1.length [Enter] ←得られた要素数を確認
2 ← 2 個
> c1[0] [Enter] ←インデックス 0 の要素を確認
<p id="p1">p 要素 1</p>
> c1[1] [Enter] ←インデックス 1 の要素を確認
<p id="p2">p 要素 2</p>
```

body 要素の直下の要素の内, HTML 要素が得られている。当然, テキスト要素, コメント要素は得られていない。それらを含めて全ての子要素を取得するには contents メソッドを使用する。(次の例)

例. body の直下の全ての子要素を取得する (先の例の続き)

```
> c2 = $( "body" ).contents() [Enter] ←全ての子要素を取得
jQuery.fn.init {0: text, 1: p#p1, 2: text, 3: comment, 4: text, 5: p#p2, 6: text,
length: 7, prevObject: j...y.fn.init}
> c2.length [Enter] ←要素の個数を確認
7
```

一見すると, 得られた要素の個数が jQuery03.html の body の子要素の個数よりも多く見える。これは, body 要素内の子要素の間に空白文字や改行文字が含まれていることによる。以下にそれを確認する。

例. テキスト要素の部分を確認 (先の例の続き)

```
> c2[0] [Enter] ←最初のテキスト要素を確認
" テキスト 1 "
> c2[0].nodeValue [Enter] ←内部のテキストデータを確認
'¥n テキスト 1¥n ' ←改行や空白が含まれている
```

従って p 要素やコメント要素の間にも空白や改行が含まれており, それらがテキスト要素として扱われる。

例. 親要素を求める (先の例の続き)

```
> b = $( "#p1" ).parent() [Enter] ← id="#p1" の p 要素の親要素を求める
jQuery.fn.init {0: body, length: 1, prevObject: j...y.fn.init}
> b[0] [Enter] ←親要素を参照
<body>...</body> ← body 要素
```

例. body 要素内のコンテンツの取得 (先の例の続き)

```
> $( "body" ).html() [Enter] ← body 内の HTML を取得
'¥n テキスト 1¥n <p id="p1">p 要素 1</p>¥n ¥x3C!-- コメント -->
¥n <p id="p2">p 要素 2</p>¥n テキスト 2¥n ¥n'
> $( "body" ).text() [Enter] ← body 内のコンテンツをテキスト形式で取得
'¥n テキスト 1¥n p 要素 1¥n ¥n p 要素 2¥n テキスト 2¥n ¥n'
```

これは body 内のコンテンツを HTML 形式, テキスト形式で取得する例である。次に, 同じメソッドによって body 内のコンテンツを上書きする例を示す。

例. body 要素内のコンテンツの上書き 1 (先の例の続き)

```
> $( "body" ).html( "<p>新しいコンテンツ</p>" ) [Enter]
jQuery.fn.init {0: body, length: 1, prevObject: j...y.fn.init}
```

この処理によって body 要素内のコンテンツが 1 つの p 要素となり, Web ブラウザの表示が「新しいコンテンツ」となる。

例. body 要素内のコンテンツの上書き 2 (先の例の続き)

```
> $( "body" ).text( "新しいテキスト" ) 
jQuery.fn.init {0: body, length: 1, prevObject: j...y.fn.init}
```

この処理によって body 要素内のコンテンツが 1 つのテキスト要素となり、Web ブラウザの表示が「新しいテキスト」となる。

9.7 要素の選択

jQuery では \$(セレクタ) としてセレクタの条件を満たす HTML 要素を選択することができるが、filter メソッドを使用することで更に高度な選択処理ができる。

書き方: \$(セレクタ).filter(条件)

「セレクタ」で選択した要素群から更に「条件」を満たすものを絞り込んだ要素群を返す。「条件」の部分にはセレクタを記述した文字列や、論理値を返す関数などを与えることができる。

次のようなサンプル jQuery06.html を使用して filter メソッドの処理を例示する。

記述例: jQuery06.html

<pre>1 <!DOCTYPE html> 2 <html lang="ja"> 3 <head> 4 <meta charset="utf-8"> 5 <title>jQuery06</title> 6 <script type="text/javascript" src="jquery.js"></script> 7 </head> 8 <body> 9 10 <li class="中村 human" data-adr="大阪">太郎 11 <li class="中村 dog" data-adr="大阪">ポチ 12 <li class="佐藤 human" data-adr="大阪">二郎 13 <li class="佐藤 cat" data-adr="大阪">タマ 14 <li class="鈴木 human" data-adr="東京">三郎 15 <li class="鈴木 dog" data-adr="東京">シロ 16 <li class="田中 human" data-adr="東京">四郎 17 <li class="田中 cat" data-adr="東京">ミケ 18 19 </body> 20 </html></pre>	<pre>1. 太郎 2. ポチ 3. 二郎 4. タマ 5. 三郎 6. シロ 7. 四郎 8. ミケ</pre>
---	--

これを Web ブラウザで表示すると右のようになる。この状態で Web ブラウザのコンソールで各種処理を例示する。次の例は、CSS のセレクタで対象の要素を選択する処理である。

例. CSS のセレクタによる要素の選択

```
> x = $( "li.human" )  ←要素の選択
jQuery.fn.init {0: li.中村.human, 1: li.佐藤.human, 2: li.鈴木.human, 3:
li.田中.human, length: 4, prevObject: j...y.fn.init}

> for ( e of x ) console.log( e.textContent )  ←選択された要素のテキストを表示
太郎      ←選択された要素のテキスト (ここから)
二郎
三郎
四郎      ←選択された要素のテキスト (ここまで)
undefined ← for 文の戻り値
```

次に、これと同じ処理を filter メソッドを用いて行う例を示す。

例. filter メソッド (先の例の続き)

```
> x = $( "li" ).filter( ".human" ) [Enter] ←要素の選択
jQuery.fn.init {0: li. 中村.human, 1: li. 佐藤.human, 2: li. 鈴木.human, 3:
li. 田中.human, length: 4, prevObject: j...y.fn.init}
> for ( e of x ) console.log( e.textContent ) [Enter] ←選択された要素のテキストを表示
太郎          ←選択された要素のテキスト (ここから)
二郎
三郎
四郎          ←選択された要素のテキスト (ここまで)
undefined     ← for 文の戻り値
```

このように、CSS のセレクタによる選択を filter メソッドで行うこともできる。

filter メソッドの引数には関数を与えることもでき、より高度な選択処理ができる。この場合の関数は論理値を返すもので、それによって要素が選択対象か否かを判別できる。関数内部では判別の対象の要素を `$(this)` と記述して扱うことができる。例えば、要素を判別するための次のような関数 f1 を定義する。

例. 要素判別関数 f1 (先の例の続き)

```
> function f1() { [Enter] ←関数定義の開始
    return $(this).data("adr") == "大阪"; [Enter]
} [Enter] ←関数定義の終了
undefined     ←関数定義の処理結果
```

この関数を filter メソッドの引数に与えると、対象要素が1つずつ関数 f1 の this に渡されて判別処理され、data-adr 属性が "大阪" である要素のみが選択される。(次の例)

例. 上記関数 f1 を filter メソッドに与える (先の例の続き)

```
> x = $( "li" ).filter( f1 ) [Enter] ←要素の選択処理
jQuery.fn.init {0: li. 中村.human, 1: li. 中村.dog, 2: li. 佐藤.human, 3:
li. 佐藤.cat, length: 4, prevObject: j...y.fn.init}
> for ( e of x ) console.log( e.textContent ) [Enter] ←選択された要素のテキストを表示
太郎          ←選択された要素のテキスト (ここから)
ポチ
二郎
タマ          ←選択された要素のテキスト (ここまで)
undefined     ← for 文の戻り値
```

filter メソッドに渡す関数の引数には、処理対象の要素のインデックスが渡される。例えば次のような関数 f2 を考える。

例. 要素判別関数 f2 (先の例の続き)

```
> function f2( n ) { [Enter] ←関数定義の開始
    return n > 3; [Enter]
} [Enter] ←関数定義の終了
undefined     ←関数定義の処理結果
```

この関数を filter メソッドの引数に与えると、対象要素が1つずつ関数 f1 の this に渡されて判別処理され、そのインデックスが 3 より大きい要素のみが選択される。(次の例)

例. 上記関数 f2 を filter メソッドに与える (先の例の続き)

```
> x = $( "li" ).filter( f2 ) [Enter] ←要素の選択処理
jQuery.fn.init {0: li. 鈴木.human, 1: li. 鈴木.dog, 2: li. 田中.human, 3:
li. 田中.cat, length: 4, prevObject: j...y.fn.init}
> for ( e of x ) console.log( e.textContent ) [Enter] ←選択された要素のテキストを表示
三郎          ←選択された要素のテキスト (ここから)
シロ
四郎
ミケ          ←選択された要素のテキスト (ここまで)
undefined     ← for 文の戻り値
```

9.8 アニメーション効果

表 60 に示すメソッドを使用すると、対象要素にアニメーション効果を与えることができる。

表 60: アニメーション用メソッド（一部）

メソッド	解説
<code>show(時間)</code> <code>hide(時間)</code>	対象の要素を、与えた「時間」（ミリ秒）をかけて表示する。 対象の要素を、与えた「時間」（ミリ秒）をかけて隠す。 「時間」のデフォルトは 0ms である。
<code>fadeIn(時間)</code> <code>fadeOut(時間)</code>	対象の要素を、与えた「時間」（ミリ秒）をかけてフェードインして表示する。 対象の要素を、与えた「時間」（ミリ秒）をかけてフェードアウトして隠す。 「時間」のデフォルトは 400ms（0.4 秒）である。
<code>slideDown(時間)</code> <code>slideUp(時間)</code>	対象の要素をスライドダウン（拡大しながら表示）する。 対象の要素をスライドアップ（縮小しながら隠匿）する。 効果に要する時間をミリ秒で指定する。デフォルトは 400ms（0.4 秒）である。

これらのメソッドは、処理結果として対象のオブジェクトを返す。

次のサンプル を用いてアニメーション効果を例示する。

記述例：jQuery07.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   <head>
4     <meta charset="utf-8">
5     <title>jQuery07</title>
6     <script type="text/javascript" src="jquery.js"></script>
7   </head>
8   <body></body>
9 </html>
```

これを Web ブラウザで表示すると図 74 のようになる。

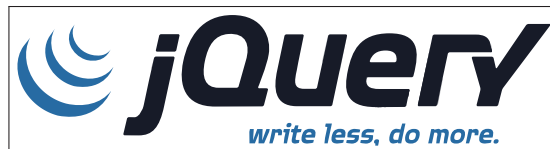


図 74: jQuery07.html の表示

この状態で Web ブラウザのコンソールを開き、アニメーション効果を実行する例を示す。

例. 画像を隠す／表示する

```
> $( "#im1" ).hide() [Enter]    ←画像を隠す処理
jQuery.fn.init {0:  img#im1, length:  1}
> $( "#im1" ).show() [Enter]    ←画像を表示する処理
jQuery.fn.init {0:  img#im1, length:  1}
```

これらの処理で画像を隠す／表示する効果が得られる。（確認されたい）

メソッドの引数にミリ秒単位で時間を与えると、隠す／表示する処理の時間を制御できる。

0 より大きい時間を与えて同様の処理を行うと、hide では左上方向への縮小と透明化によって画像を隠し、show では右下方向への拡大と不透明化によって表示する。（確認されたい）

slideDown, slideUp も show, hide と似た効果をもたらすが、不透明化、透明化の効果は無く、拡大、縮小による表示と隠匿である。また、アニメーションの対象の横幅のサイズを明に設定するとそれが固定され、高さの拡大による表示、高さの縮小による隠匿の効果（垂直のスライドイン／スライドアップ）が実現できる。

例. 垂直方向の slideUp, slideDown (先の例の続き)

```
> w = $( "#im1" ).css( "width" ) Enter    ←対象要素の横幅を取得して
'1022px'

> $( "#im1" ).css( "width", w ) Enter    ←それを明に再設定
jQuery.fn.init {0: img#im1, length: 1}

> $( "#im1" ).slideUp() Enter    ←垂直のスライドアップ
jQuery.fn.init {0: img#im1, length: 1}

> $( "#im1" ).slideDown() Enter    ←垂直のスライドダウン
jQuery.fn.init {0: img#im1, length: 1}
```

実際の効果について確認されたい。

9.9 Web アプリケーション実装に関すること

jQuery を応用した Web アプリケーションを構築する場合は、jQuery 自体の読み込みの開始と終了、イベントハンドリングの登録をはじめとする初期化処理のタイミングについて考える必要がある。

Web アプリケーションのプログラミングのためのライブラリには jQuery の他にも多くのものがある。特にサイズの大きなライブラリは読み込みに時間がかかることがあり、head 要素内でそれを読み込む処理を行うと、コンテンツ全体のレンダリングをブロックすることがある。そのような場合は body 要素内の最終の位置にライブラリを読み込む処理を記述することが推奨される。ただし、jQuery はそのサイズが 300KB に満たないものであり、読み込み処理を head 要素内で行っても問題が起こることはあまり無い。そのような理由で本書では、jQuery を script 要素の src 属性に指定して head 要素内で読み込む形式を基本としている。

Web アプリケーションは**イベント駆動型プログラミング**の形で実装することが基本的であり、HTML で構築された各種の UI に対してイベントハンドリングを設定する。Web アプリケーションは起動時にイベントハンドリングの設定をはじめとする各種の初期化の処理を実行するが、それは必要とするライブラリの読み込みが完了した後で行う。このような初期化の処理は、1つの関数として実装しておき、HTML の DOM が完成した直後、あるいは全てのリソースの読み込みが完了した直後に実行するというのが、アプリケーション開発における一般的な形である。

Web アプリケーションの初期化処理の実装方法について次のサンプル jQuery08.html を用いて解説する。

記述例：jQuery08.html

```
1  <!DOCTYPE html>
2  <html lang="ja">
3    <head>
4      <meta charset="utf-8">
5      <title>jQuery08</title>
6      <style>
7        #im1 {
8          width: 530px; margin-top: 10px;
9        }
10     </style>
11     <script type="text/javascript" src="jquery.js"></script>
12     <script>
13       function f0() {
14         $("#im1").hide();
15         $("#b11").click( function() { $("#im1").show(); } );
16         $("#b12").click( function() { $("#im1").hide(); } );
17         $("#b21").click( function() { $("#im1").fadeIn(); } );
18         $("#b22").click( function() { $("#im1").fadeOut(); } );
19         $("#b31").click( function() { $("#im1").slideDown(); } );
20         $("#b32").click( function() { $("#im1").slideUp(); } );
21       }
22       $(f0); // (1)
23       $(document).ready(f0); // (2)
24       $(window).on("load",f0); // (3)
25     </script>
26   </head>
27   <body>
28     <input type="button" value="表示" id="b11">
29     <input type="button" value="隠す" id="b12">
30     <input type="button" value="フェードイン" id="b21">
```

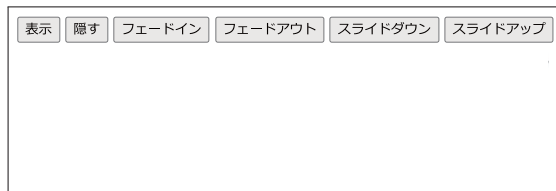


```

31 <input type="button" value="フェードアウト" id="b22">
32 <input type="button" value="スライドダウン" id="b31">
33 <input type="button" value="スライドアップ" id="b32"><br>
34 
35 </body>
36 </html>

```

これを Web ブラウザで表示すると図 75 の (a) のようになる。



(a) 起動時



(b) 画像表示

図 75: jQuery08.html の表示

ウィンドウ上部にはいくつかのボタンが設置されており、それらのクリックによって画像の表示と隠匿の操作ができる。

jQuery08.html では、イベントハンドリングの登録などの処理化処理を関数 f0 として定義している。このコンテンツを Web ブラウザに読み込むと head 要素内の script 要素によって jQuery が読み込まれ、続いて関数 f0 の定義処理と、その実行をスケジュールする処理をコメント (1)~(3) のある行によって行っている。

実際に f0 を起動して初期化処理を行うのはコメント (1)~(3) の行の内の 1 行であり、3 種類の方法から選ぶことができる。(1) の行と (2) の行の形式

```

$( 関数 )
$(document).ready( 関数 )

```

は基本的に同じであり、DOM の構築が完了した後で「関数」が実行される。(3) の行の形式

```

$(window).on("load", 関数 )

```

では、DOM 構築と全てのリソース（画像など）の読み込みが完了した後で「関数」が実行される。

10 プログラムライブラリ (2) : MathJax

MathJax ライブラリを使用することで、HTML コンテンツ内に記述した $\text{T}_{\text{E}}\text{X}$ の数式を整形表示することができる。
MathJax に関する情報は公式インターネットサイト

<https://www.mathjax.org/>

から入手できる。MathJax ライブラリは、公式サイトがオンラインで公開しているもの (CDN) を直接利用することもできるが、ライブラリ本体をローカルの環境にダウンロードしてオフラインで利用することもできる。

ここでは、サンプルを示しながら MathJax の利用方法について解説する。

■ サンプル 1 (オンラインでの利用) : mathjax01-1.html

```
1 <html>
2 <head>
3   <meta charset="utf-8">
4   <title>MathJaxによる数式の表示 (CDN) </title>
5   <style>
6     p {
7       margin-top: 10pt;
8       margin-left: 10pt;
9       font-size: 12pt;
10    }
11  </style>
12  <script id="MathJax-script" async
13    src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-cthtml.js"></script>
14 </head>
15 <body>
16   <p>\(\displaystyle \frac{d}{dx}\sin(x)=\cos(x)\)</p>
17   <p>\(\displaystyle \int_{-\infty}^{\infty}f(x)dx=1\)</p>
18 </body>
19 </html>
```

サンプル中の 12~13 行目にあるように、公式サイトの JavaScript を利用するために

```
<script id="MathJax-script" async
  src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-cthtml.js"></script>
```

と記述することで MathJax が有効になる。これを Web ブラウザで表示すると図 76 のようになる。

$$\frac{d}{dx}\sin(x) = \cos(x)$$
$$\int_{-\infty}^{\infty} f(x) dx = 1$$

図 76: Web ブラウザで表示した例

mathjax01-1.html に示した例は、公式サイトが公開するライブラリをオンラインで利用した例であるが、MathJax ライブラリ本体をローカルの環境に配置することで、インターネット接続が無い状態 (オフライン) でも同様の処理を行うことができる。

■ サンプル 2 (オフラインでの利用) : mathjax01-2.html

```
1 <html>
2 <head>
3   <meta charset="utf-8">
4   <title>MathJaxによる数式の表示 (ローカル) </title>
5   <style>
6     p {
7       margin-top: 10pt;
8       margin-left: 10pt;
9       font-size: 12pt;
10    }
```

```

11 |     </style>
12 |     <script id="MathJax-script" async
13 |         src="mathjax/es5/tex-cthtml.js"></script>
14 | </head>
15 | <body>
16 |     <p>\(\displaystyle \frac{d}{dx}\sin(x)\,,=\,,\cos(x)\)</p>
17 |     <p>\(\displaystyle \int^{\infty}_{-\infty}f(x)\,,dx\,,=\,,1\)</p>
18 | </body>
19 | </html>

```

この例は、MathJax ライブラリを含むディレクトリ `mathjax` が当該コンテンツと同じディレクトリに配置されていることを前提としている。サンプル中の 12～13 行目にあるように

```

<script id="MathJax-script" async
    src="mathjax/es5/tex-cthtml.js"></script>

```

と記述することで MathJax が有効になり、インターネット接続が無い状態でも数式が整形表示される。

11 プログラムライブラリ (3) : React

React は米 Meta 社（旧 Facebook）が開発した、ユーザインターフェース（UI）構築用の JavaScript ライブラリである。React はオープンソースソフトウェアである¹¹⁷。React による Web ページのレンダリングは独自の**仮想 DOM**を基本としており、Web ページの更新において高い効率を実現する。また、JavaScript に独自の拡張を加えた JSX 言語の導入によって簡便な UI 設計を可能にする。

React は高度な UI 構築を可能にするだけでなく、画面遷移や状態管理のための優れた機能を提供しており、高度な Web アプリケーションを実現するための手段を簡便な形で提供する。React を用いて構築した Web アプリケーションのことを本書では「React アプリ」と呼ぶこととする。

React に関しては、Meta 社が最も基本的な機能を支えるライブラリを提供しているが、それを応用した高機能なライブラリを多くの団体、個人（サードパーティ）が公開しており、全体として UI ライブラリのエコシステムを成している。本書では React の最も基本的な部分について解説する。

11.1 基礎事項

11.1.1 基本的なライブラリ

React の最も基本的な機能を提供するものに次の 2 つのものがある。

- 1) `react.development.js` 開発作業用
- 2) `react.production.min.js` 本番用

1) の方は、エラーメッセージなどが詳細で、デバックのために有利であるので、Web アプリケーションを開発する段階で使用される。2) の方は Web アプリケーションの本番運用に適した形に最適化されている。従って上記 1), 2) のどちらかを選択して使用する。このライブラリは 最も基本的な **React オブジェクト**を提供する。

React が DOM を扱うための機能を提供するものに次の 2 つのものがある。

- 1) `react-dom.development.js` 開発作業用
- 2) `react-dom.production.min.js` 本番用

先の場合と同様に、開発作業の段階か本番運用の段階かで上記 1), 2) のどちらかを選択して使用する。このライブラリは DOM 構築に関する **ReactDOM オブジェクト**を提供する。

JSX のトランスパイルに Babel と呼ばれるライブラリ¹¹⁸が必要で、これには次の 2 つのものがある。

- 1) `babel.js` 開発作業用
- 2) `babel.min.js` 本番用

開発作業の段階か本番運用の段階かで上記 1), 2) のどちらかを選択して使用する。

JSX は、XML 要素を簡便な形でデータとして扱うための JavaScript の拡張である。Web ブラウザに備わった通常の JavaScript のエンジンは JSX をそのままの形で扱うことができない。従って、JSX で記述されたプログラムは一旦通常の JavaScript のプログラムに変換する必要がある。Babel はこれを行う機能を提供する。

参考)

高水準言語のプログラムを低水準言語（機械語など）のプログラムに変換する処理は「コンパイル」と呼ばれるが、高水準言語のプログラムを別の高水準言語のプログラムに変換する処理は「トランスパイル」と呼ばれる。

11.1.2 React アプリのレンダリング

React は、HTML 文書中の指定した要素に対して構築した React アプリをレンダリングする。すなわち、指定した HTML 要素を頂点（root）として、そこに独自の DOM を構成する。例えば HTML 中の `<div id="root"></div>` という div 要素がある場合、

```
const domRoot = ReactDOM.createRoot(root);
```

とすると、当該 div 要素が domRoot として React アプリのレンダリング対象となる。

¹¹⁷MIT ライセンス

¹¹⁸Babel の開発元は React の開発元とは異なる。（<https://babeljs.io/>）

レンダリングの対象に **React 要素**や**コンポーネント**（後述）などをレンダリングするには、

レンダリング対象.render(要素やコンポーネントなど)

と render メソッドを用いる。

この方法以外にも

ReactDOM.render(要素やコンポーネントなど, document.getElementById("root"))

としてレンダリングすることもできる。ただし、先に解説した方法は React 18 版から導入された新しいものであり、並行レンダリングなどの高度な機能が有効になるので、先に解説した方法を取ることが推奨される。

11.1.3 React 要素とコンポーネント

React アプリを構成する要素（**React 要素**）を次のようにして作成することができる。

React.createElement(HTML 要素名, プロパティ, 子要素)

例えば、`<input type="button" value="OK">` という HTML 要素に該当する React 要素は次のようにして作成することができる。

React.createElement("input", type: "button", value: "OK")

本書では、React アプリ構築に関する文脈では React 要素のことを単に「要素」と呼ぶことがあるので了解されたい。

React 要素として HTML の h1 要素をレンダリングするサンプル `reactTest01.html` を示す。

記述例：reactTest01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>reactTest01</title>
6   <!-- Reactのライブラリを読み込む -->
7   <script src="react.development.js"></script>
8   <script src="react-dom.development.js"></script>
9 </head>
10 <body>
11   <div id="root"></div> <!-- Reactアプリの最上位要素 -->
12   <script>
13     // h1要素を作成
14     const e = React.createElement("h1", {}, "Hello, world!");
15     // 要素をDOMにレンダリング
16     const domRoot = ReactDOM.createRoot(root);
17     domRoot.render(e);
18   </script>
19 </body>
20 </html>
```

これを Web ブラウザで開くと **Hello, world!** と表示される。

11.1.3.1 JSX

JSX では XML を直接的にデータとして扱うことができる。また JSX の XML の記述の中には { } で式を括って記述ことができ、式の値を XML の中に埋め込むことができる。JSX のスクリプトを script 要素で読み込んで実行するには属性 `type="text/babel"` を与える。

JSX を用いると先の `reactTest01.html` において h1 要素を作成する部分をタグの表記で

`const e = <h1>Hello, world!</h1>;`

と記述することができる。この形で `reactTest01.html` を書き換えた `reactTest02.html`¹¹⁹ を示す。

記述例：reactTest02.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
```

¹¹⁹後の「11.2 実用的な使用方法」(p.253) では、npm でビルドするアプリケーションプロジェクト `buildTest02` としての版を示す。

```

4   <meta charset="utf-8">
5   <title>reactTest02</title>
6   <!-- Reactのライブラリを読み込む -->
7   <script src="react.development.js"></script>
8   <script src="react-dom.development.js"></script>
9   <script src="babel.min.js"></script>
10  </head>
11  <body>
12    <div id="root"></div>
13    <script type="text/babel">
14      // h1要素を作成
15      const e = <h1>Hello, world!</h1>;
16      // React要素をDOMにレンダリング
17      const domRoot = ReactDOM.createRoot(root);
18      domRoot.render(e);
19    </script>
20  </body>
21 </html>

```

head 要素内で `<script src="babel.min.js"></script>` として Babel を読み込み、
body 要素内で `<script type="text/babel">` として JSX を有効にしている。

11.1.3.2 クラスコンポーネント、関数コンポーネント

コンポーネントは UI を構成するためのものである。これは単なる React 要素とは異なり、React 要素やそれらから構成される部分的な DOM、それらに対する処理（イベントハンドリングなど）を含んだものである。またコンポーネントは階層的に構築することができる。コンポーネントには関数として定義されるものと、クラスとして定義されるものがあり、それぞれ**関数コンポーネント**、**クラスコンポーネント**と呼ぶ。

クラスコンポーネントは `React.Component` クラスの拡張クラスとして定義し、そのインスタンスメソッド `render` 内で構築した UI を戻り値として返す。関数コンポーネントは更に簡単なものであり、構築した UI を戻り値として返す。コンポーネントはクラスあるいは関数であるので、必要なだけ生成することができる。

クラスコンポーネント、関数コンポーネントとして HTML の h1 要素、h2 要素を実装する例を `reactTest03.html` に示す。

記述例：reactTest03.html

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="UTF-8" />
5    <title>reactTest03</title>
6    <script src="react.development.js"></script>
7    <script src="react-dom.development.js"></script>
8    <script src="babel.min.js"></script>
9  </head>
10 <body>
11   <div id="root"></div>
12   <script type="text/babel">
13     // クラスコンポーネントによる h1要素の定義
14     class HelloWorld extends React.Component {
15       render() {
16         return ( <h1>Hello, world!</h1> );
17       }
18     }
19     // 関数コンポーネントによる h2要素の定義
20     function Nice2MeetU() {
21       return ( <h2>Nice to meet you.</h2> );
22     }
23     // DOMにレンダリング
24     const domRoot = ReactDOM.createRoot(root);
25     domRoot.render( <><HelloWorld /><Nice2MeetU /></> );
26   </script>
27 </body>
28 </html>

```


この例では h1 要素を HelloWorld クラスとして、h2 要素を Nice2MeetU 関数として定義している。定義されたコンポーネントを JSX によって DOM に配置するには

```
<コンポーネント名 />
```

と記述する。

reactTest03.html を Web ブラウザで開くと右のように表示される。

Hello, world!

Nice to meet you.

■ React.Fragment

React の JSX では、複数の要素やコンポーネントを React.Fragment によってまとめることができる。reactTest03.html では

```
<><HelloWorld /><Nice2MeetU /></>
```

として <>, </> で 2 つのコンポーネントをまとめている。<> は <React.Fragment> の省略形、</> は </React.Fragment> の省略形である。

■ コンポーネントの名称に関する注意

JSX では基本的に、小文字で始まる要素名は HTML 要素と見做される。従って、コンポーネントの名称は大文字で始まるものにしなければならない。

11.1.4 コンポーネントに値を渡す方法：Props

コンポーネントには Props を介して値を渡すことができる。この際、HTML 要素への属性の設定と同様の形式で記述する。

書き方： <コンポーネント名 名前 1={ 式 1 } 名前 2={ 式 2 } />

これによって「名前」に対する「値」が割り当てられ、それが Props としてコンポーネントに渡される。クラスコンポーネントが Props を受け取ると、当該クラスのインスタンスはそれを this.props として受取り、

```
this.props.名前
```

として値を参照することができる。

関数コンポーネントは引数を介して Props を受け取る。すなわち、引数 p に Props を受け取った関数はその内部で

```
p.名前
```

として値を参照することができる。

JSX による HTML (XML) の記述の中に Props として受け取った値を埋め込むにはそれを { } で括る。例えば p という Props の prp という名前の値を h1 要素のコンテンツとするには、

```
<h1>{p.prp}</h1>
```

などと記述する。

Props を用いてコンポーネントに値を渡す例をサンプル reactTest04.html で示す。

記述例：reactTest04.html

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8" />
5   <title>reactTest04</title>
6   <script src="react.development.js"></script>
7   <script src="react-dom.development.js"></script>
8   <script src="babel.min.js"></script>
9 </head>
10 <body>
11   <div id="root"></div>
12   <script type="text/babel">
13     // クラスコンポーネントによる h1 要素の定義
14     class HelloWorld extends React.Component {
15       render() {
16         return ( <h1>{this.props.prp1}</h1> );
```

```

17     }
18   }
19   // 関数コンポーネントによる h2 要素の定義
20   function Nice2MeetU(p) {
21     return ( <h2>{p.prp2}</h2> );
22   }
23   // DOMにレンダリング
24   const s1 = "Hello, world!";
25   const s2 = "Nice to meet you.";
26   const domRoot = ReactDOM.createRoot(root);
27   domRoot.render(
28     <><HelloWorld prp1={s1}/>
29     <Nice2MeetU prp2={s2}/></>);
30   </script>
31 </body>
32 </html>

```

この例は先のサンプル reactTest03.html を改変したもので、h1、h2 要素に Props を介してコンテンツを渡す形にしている。すなわち、変数 s1 の文字列を HelloWorld コンポーネントは prp1 という名前で、変数 s2 の文字列を Nice2MeetU コンポーネントは prp2 という名前で受け取っている。

11.1.5 コンポーネントの State とライフサイクル

11.1.5.1 State

コンポーネントは独自に状態を保持して管理するための State (ステート) を持つことができる。State は通常の変数とは異なるもので、React がその変化を監視しており、State の変化を起点にして当該コンポーネントのレンダリングを発動する。

クラスコンポーネントは、そのインスタンスの state プロパティとして State を保持し、setState メソッドで State を変更する。クラスコンポーネントに State を与えるには、そのコンストラクタ内で

```
this.state = { キー 1:値 1, キー 2:値 2, … }
```

と記述すると良い。各キーに対する値を変更するには次のようにして setState メソッドを実行する。

```
this.setState( { キー 1:新しい値 1, キー 2:新しい値 2, … } )
```

このとき、キーと値のペアは変更対象となるもののみで良い。また、キーに対する値を累積的に変更する¹²⁰ 場合は setState の引数に、値を変更するための関数を与える。例えば、State のキー c の値を 1 増加させる処理を実装するには次のように記述すると良い。

```
this.setState( s => ( { c:s.c + 1 } ) )
```

このような記述の場合、関数の処理対象の s には当該コンストラクタの State である this.state が渡される。また、関数の戻り値として更新後の State 中の値を表すオブジェクト (変更対象となるもの) を返す形にする。

関数コンポーネントに State を持たせるには useState フックを使用する。すなわち、コンポーネントの定義の冒頭で State を使用するための宣言を

```
const [ ステート変数, 更新用関数 ] = React.useState( 初期値 )
```

と記述する。これにより State を構成するためのステート変数とそれを変更するために使用する更新用関数が得られる。また、ステート変数に「初期値」が与えられる。

更新用関数を実行すると第 1 引数に与えた値がステート変数に設定される。また、クラスコンポーネントの場合と同様に、ステート変数を累積的に変更する¹²⁰ 場合はそれを実行する関数を更新用関数の引数に与える。例えば、ステート変数の値を 1 増加させる処理を実装するには更新用関数の引数に

```
x => x + 1
```

といった関数式を与えると良い。

useState フックを用いた宣言は、必要に応じて複数回実行可能で、複数のステート変数を作成する¹²¹ ことができる。

¹²⁰ 前の値から新しい値を算出する処理など。カウンタの値を増やす処理などがこれに該当する。

¹²¹ React の内部事情から、同一の関数コンポーネント内では、ステート変数取得の記述の順序は一定している必要がある。

11.1.5.2 ライフサイクル

コンポーネントは次のようなライフサイクルを持つ。

1) 生成

関数コンポーネントは `return` で戻り値が返された時点で、クラスコンポーネントはそのインスタンスがコンストラクタによって作られた時点で当該コンポーネントが生まれる。

2) DOM へのマウント

生成されたコンポーネントは `React` の仮想 DOM の一部として組み込まれる。クラスコンポーネントの場合は `componentDidMount` メソッドがこのフェーズを捉えており、この時点で実行する処理は `componentDidMount` メソッドとして実装する。

3) 更新

コンポーネントに渡される `Props` の値が変更された、あるいは `State` が変更された場合などに当該コンポーネントは自動的に再度レンダリング¹²² される。

4) DOM からのアンマウント

`State` の変更などを起点にして、既に DOM に存在するコンポーネントが DOM から外されることがある。クラスコンポーネントの場合は `componentWillUnmount` メソッドがこのフェーズを捉えており、この直前に実行する処理は `componentWillUnmount` メソッドとして実装する。

関数コンポーネントにおいて上記 2),4) のフェーズを捉えるには `useEffect` フックを用いてマウント時の処理とアンマウント時の処理を登録する。

```
書き方： React.useEffect( () => {  
    (マウント直後に実行する処理)  
    return アンマウント時に実行する関数; }  
    , 依存配列)
```

「アンマウント時に実行する関数」のことをクリーンアップ関数と呼ぶことがある。

11.1.5.3 コンポーネントのスタイルの設定

コンポーネントのスタイルを設定するには、スタイル設定を保持するオブジェクトをコンポーネントの `style` 属性に与える。

```
書き方： <コンポーネント style={スタイル設定オブジェクト} />
```

例. コンポーネントへのスタイルの設定

```
const s = { position:"absolute", left:"10px", top:"60px" };  
return ( <コンポーネント style={s}/> );
```

`React` 要素も同様の形式でスタイル設定ができる。

11.1.5.4 条件付きレンダリング

条件式によってレンダリングの制御が可能である。

```
書き方： { 条件式 && <コンポーネント /> }
```

「条件式」が真 (`true`) の場合に「コンポーネント」をレンダリングする。この形式で既にレンダリングされているコンポーネントも再レンダリングの際に「条件式」が偽 (`false`) になると、そのコンポーネントはアンマウントされる。

また、条件判定のための三項演算子を応用することもできる。

```
書き方： { 条件式 ? <コンポーネント 1 /> : <コンポーネント 2 /> }
```

「条件式」が真 (`true`) の場合に「コンポーネント 1」を、偽 (`false`) の場合に「コンポーネント 2」をレンダリングする。

¹²²このとき `React` の最新の仮想 DOM が Web ブラウザの実際の DOM に反映される。

11.1.5.5 実装例に沿った解説

ここでは実装例を通してコンポーネントの State とライフサイクルについて考える。右に示すような画像を img 要素として表示するコンポーネントを実装して、そのライフサイクルを扱う例を reactTest05.html に示す。



画像：BallGray02.png

記述例：reactTest05.html（コンポーネントがアンマウントされない例）

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8" />
5   <title>reactTest05</title>
6   <script src="react.development.js"></script>
7   <script src="react-dom.development.js"></script>
8   <script src="babel.min.js"></script>
9 </head>
10 <body>
11   <div id="root"></div>
12   <script type="text/babel">
13     // Reactアプリを関数コンポーネントとして実装する
14     function App() {
15       // ボールを表示するクラスコンポーネント
16       class B1 extends React.Component {
17         // コンストラクタ
18         constructor() {
19           super();
20           this.state = { pos: 0 };    // Stateの用意
21         }
22         // DOMマウント時の準備処理
23         componentDidMount() {
24           console.log(" B1がマウントされた");
25           this.timerID = setInterval(() => {
26             this.setState({pos: this.state.pos + 2});
27             console.log(" B1が移動した");
28           }, 20);
29         }
30         // アンマウント時の終了処理（クリーンアップ）
31         componentWillUnmount() {
32           console.log(" B1がアンマウントされた");
33           clearInterval(this.timerID);
34         }
35         // このコンポーネントのレンダリング
36         render() {
37           // ボールが右端に到達するまで移動する処理
38           if ( this.state.pos < 200) {
39             const s = { position:"absolute",
40                         left:this.state.pos+"px", top:"0px" };
41             return (  );
42           }
43         }
44       }
45       // ボールを表示する関数コンポーネント
46       function B2() {
47         const [pos,setPos] = React.useState(0);    // Stateの用意
48         // DOMマウント時の準備処理
49         React.useEffect( () => {
50           console.log(" B2がマウントされた");
51           const timerID = setInterval(() => {
52             setPos( p => p+1 );
53             console.log(" B2が移動した");
54           }, 20);
55           // クリーンアップ関数
56           return () => {
57             clearInterval(timerID);
58             console.log(" B2がアンマウントされた");
59           };
60         }, []);
61         // ボールが右端に到達するまで移動する処理（レンダリング）
62         if ( pos < 200) {
63           const s = { position:"absolute",
```

```

64         left:pos+"px", top:"60px" }];
65     return (  );
66   }
67 }
68 // この React アプリ全体のレダリング
69 return ( <><B1 /><B2 /></> );
70 }
71 const domRoot = ReactDOM.createRoot(root);
72 domRoot.render( <App /> );
73 </script>
74 </body>
75 </html>

```

この例では B1, B2 の2つのコンポーネントが img 要素として画像 BallGray02.png を表示する。また、それら2つのコンポーネントはそれぞれ状態変数 pos を持ち、マウントされた時点で setInterval によってタイマーを作成して pos を変化させ続ける。pos は img 要素の水平位置を決定するので、これを Web ブラウザで表示すると図 77 のようになる。

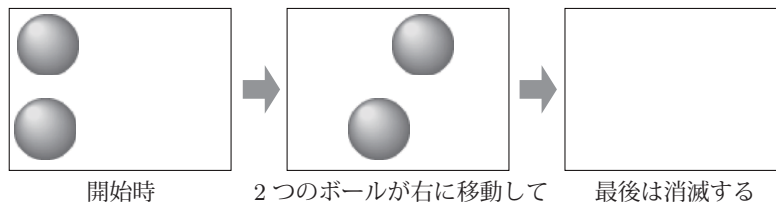


図 77: Web ブラウザでの表示

B1 によるボールは B2 によるボールの2倍の速度で移動する。

reactTest05.html ではコンポーネントの生成、マウント、更新を行うが アンマウントはしないことに注意すること。このことは Web ブラウザのコンソールで確認できる。(次の例)

コンソールの表示.

```

B1 がマウントされた
B2 がマウントされた
B1 が移動した
B2 が移動した
B1 が移動した
B2 が移動した
:

```

B1, B2 がマウントされて表示位置が移動するのが確認できるが、両者が Web ブラウザ上から消滅した後も

```

B1 が移動した
B2 が移動した

```

のメッセージが繰り返され、どちらもアンマウントされることはない。また両者の timerID のタイマーも解除されず動作を続ける。

■ 条件付きレンダリングでコンポーネントをアンマウントする例

条件付きレンダリングを応用するとコンポーネントを明示的にアンマウントすることができる。次に示す reactTest05-2.html は先の reactTest05.html と似ているが、コンポーネント B1, B2 の表示を制御する状態変数 c1, c2 を App コンポーネント内に設置しており、c1 が true の場合に B1 を、c2 が true の場合に B2 をレンダリングする。

記述例: reactTest05-2.html (コンポーネントが不自然にアンマウントされる例)

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="UTF-8" />
5    <title>reactTest05-2</title>
6    <script src="react.development.js"></script>
7    <script src="react-dom.development.js"></script>
8    <script src="babel.min.js"></script>
9  </head>
10 <body>

```

```

11 <div id="root"></div>
12 <script type="text/babel">
13   // Reactアプリを関数コンポーネントとして実装する
14   function App() {
15     const [c1,setC1] = React.useState(true);
16     const [c2,setC2] = React.useState(true);
17     // ボールを表示するクラスコンポーネント
18     class B1 extends React.Component {
19       // コンストラクタ
20       constructor() {
21         super();
22         this.state = { pos: 0 };    // Stateの用意
23       }
24       // DOMマウント時の準備処理
25       componentDidMount() {
26         console.log(" B1がマウントされた");
27         this.timerID = setInterval(() => {
28           this.setState({pos: this.state.pos + 2});
29           console.log(" B1が移動した");
30         }, 20);
31       }
32       // 再レンダリング時点で終了をチェック
33       componentDidUpdate() {
34         if( this.state.pos >= 200 ) this.props.setC(false);
35       }
36       // アンマウント時の終了処理（クリーンアップ）
37       componentWillUnmount() {
38         console.log(" B1がアンマウントされた");
39         clearInterval(this.timerID);
40       }
41       // このコンポーネントのレンダリング
42       render() {
43         // ボールが右端に到達するまで移動する処理
44         if ( this.state.pos < 200) {
45           const s = { position:"absolute",
46                     left:this.state.pos+"px", top:"0px" };
47           return (  );
48         }
49       }
50     }
51     // ボールを表示する関数コンポーネント
52     function B2(p) {
53       const [pos,setPos] = React.useState(0);    // Stateの用意
54       // DOMマウント時の準備処理
55       React.useEffect( () => {
56         console.log(" B2がマウントされた");
57         const timerID = setInterval(() => {
58           setPos( p => p+1 );
59           console.log(" B2が移動した");
60         }, 20);
61         // クリーンアップ関数
62         return () => {
63           clearInterval(timerID);
64           console.log(" B2がアンマウントされた");
65         };
66       }, [p]);
67       // posが変更した時点で終了をチェック
68       React.useEffect( () => {
69         if (pos >= 200) p.setC(false);
70       }, [pos]);
71       // ボールが右端に到達するまで移動する処理（レンダリング）
72       if ( pos < 200) {
73         const s = { position:"absolute",
74                   left:pos+"px", top:"60px" };
75         return (  );
76       }
77     }
78     // このReactアプリ全体のレンダリング（条件付きレンダリング）
79     return (
80       <>
81         { c1 && <B1 c={c1} setC={setC1} /> }
82         { c2 && <B2 c={c2} setC={setC2} /> }

```



```

83     </>
84   );
85 }
86 const domRoot = ReactDOM.createRoot(root);
87 domRoot.render( <App /> );
88 </script>
89 </body>
90 </html>

```

この例では、B1 の変数 pos が 200 を超えた場合に c1 を false に、B2 の変数 pos が 200 を超えた場合に c2 を false にする。これにより、B1、B2 が右端に到達したときにアンマウントし、その際にコンポーネントのタイマーも解除される。ただし、B1 によるボールが右端で消滅した際に B2 によるボールも影響を受けて一旦アンマウントされる。この直後、B2 は再度マウントされ位置が左端に戻る。これは、App コンポーネントが再レダリングされる際に B1、B2 の両者が同時にアンマウントされることによる。

上記のマウント-アンマウントの流れを Web ブラウザのコンソールで確認されたい。

B1、B2 のレダリングを個別に制御するには、それらのステートを個別に保持するためのラッパーとなるコンポーネントで包むと良い。

■ ステート管理のためのラッパー

先の例と似た動作をするサンプル reactTest05-3.html¹²³を示す。この例では、コンポーネント B1、B2 を包むためのラッパーコンポーネント Wrapper1、Wrapper2 を設置している。これにより、B1、B2 のステートを個別に管理して、B1 のアンマウントが B2 に影響を与えないようにしている。

記述例：reactTest05-3.html

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="UTF-8" />
5    <title>reactTest05-3</title>
6    <script src="react.development.js"></script>
7    <script src="react-dom.development.js"></script>
8    <script src="babel.min.js"></script>
9  </head>
10 <body>
11   <div id="root"></div>
12   <script type="text/babel">
13     // Reactアプリを関数コンポーネントとして実装する
14     function App() {
15       // 1つ目のラッパー
16       function Wrapper1() {
17         const [c1,setC1] = React.useState(true);
18         // ボールを表示するクラスコンポーネント
19         class B1 extends React.Component {
20           // コンストラクタ
21           constructor() {
22             super();
23             this.state = { pos: 0 };    // Stateの用意
24           }
25           // DOMマウント時の準備処理
26           componentDidMount() {
27             console.log(" B1がマウントされた");
28             this.timerID = setInterval(() => {
29               this.setState({pos: this.state.pos + 2});
30               console.log(" B1が移動した");
31             }, 20);
32           }
33           // 再レンダリング時点で終了をチェック
34           componentDidUpdate() {
35             if( this.state.pos >= 200 ) this.props.setC(false);
36           }
37           // アンマウント時の終了処理（クリーンアップ）
38           componentWillUnmount() {

```

¹²³後の「11.2.2 ソースコードの分割開発」(p.256) では、npm でビルドするアプリケーションプロジェクト buildTest05 としての版を示す。

```

39         console.log(" B1がアンマウントされた");
40         clearInterval(this.timerID);
41     }
42     // このコンポーネントのレンダリング
43     render() {
44         // ボールが右端に到達するまで移動する処理
45         if ( this.state.pos < 200) {
46             const s = { position:"absolute",
47                         left:this.state.pos+"px", top:"0px" };
48             return (  );
49         }
50     }
51 }
52 // このラッパーのレンダリング
53 return ( c1 && <B1 c={c1} setC={setC1} /> );
54 }
55 // 2つ目のラッパー
56 function Wrapper2() {
57     const [c2,setC2] = React.useState(true);
58     // ボールを表示する関数コンポーネント
59     function B2(p) {
60         const [pos,setPos] = React.useState(0);    // Stateの用意
61         // DOMマウント時の準備処理
62         React.useEffect( () => {
63             console.log(" B2がマウントされた");
64             const timerID = setInterval(() => {
65                 setPos( p => p+1 );
66                 console.log(" B2が移動した");
67             }, 20);
68             // クリーンアップ関数
69             return () => {
70                 clearInterval(timerID);
71                 console.log(" B2がアンマウントされた");
72             };
73         }, []);
74         // posが変更した時点で終了をチェック
75         React.useEffect( () => {
76             if (pos >= 200) p.setC(false);
77         }, [pos]);
78         // ボールが右端に到達するまで移動する処理 (レンダリング)
79         if ( pos < 200) {
80             const s = { position:"absolute",
81                         left:pos+"px", top:"60px" };
82             return (  );
83         }
84     }
85     // このラッパーのレンダリング
86     return ( c2 && <B2 c={c2} setC={setC2} /> );
87 }
88 return (
89     <>
90         <Wrapper1 />
91         <Wrapper2 />
92     </>
93 )
94 }
95 const domRoot = ReactDOM.createRoot(root);
96 domRoot.render( <App /> );
97 </script>
98 </body>
99 </html>

```

これを Web ブラウザで表示すると 2 つのボールが左端から右端に移動して消滅する様子がわかる。コンポーネントのマウント-アンマウントの流れを Web ブラウザのコンソールで確認されたい。

11.1.6 コンポーネントのイベントハンドリング

コンポーネントにイベントハンドラを登録するには、当該コンポーネントの属性

on イベント = { イベントハンドラ }

として登録する。これに関して reactTest06.html に例示する。

記述例：reactTest06.html

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8" />
5   <title>reactTest06</title>
6   <script src="react.development.js"></script>
7   <script src="react-dom.development.js"></script>
8   <script src="babel.min.js"></script>
9 </head>
10 <body>
11   <div id="root"></div>
12   <script type="text/babel">
13     function App() {
14       class B1 extends React.Component {
15         eh1() { // ボタン「B1」のイベントハンドラ
16           console.log("B1がクリックされた。");
17         }
18         render() {
19           return ( <input type="button" value="B1" onClick={this.eh1} /> );
20         }
21       }
22       function B2() {
23         function eh2() { // ボタン「B2」のイベントハンドラ
24           console.log("B2がクリックされた。");
25         }
26         return ( <input type="button" value="B2" onClick={eh2} /> );
27       }
28       // このReactアプリ全体のレダリング
29       return ( <><B1 /><B2 /></> );
30     }
31     const domRoot = ReactDOM.createRoot(root);
32     domRoot.render( <App /> );
33   </script>
34 </body>
35 </html>
```

これを Web ブラウザで表示すると のようにボタンが2つ表示される。それらボタンをクリックすると Web ブラウザのコンソールにそれぞれ、

B1 がクリックされた。
B2 がクリックされた。

と表示される。

11.1.7 複数コンポーネントの一括デプロイ：リストとキー

箇条書き (ul, ol, li 要素) や表 (table, tr, td 要素) は同じ要素を複数配置して構成するが、それらは要素やコンポーネントの配列としてレンダリングすることができる。例えば、ul 要素中に複数の li 要素を配列としてレンダリングする例 reactTest07.html を次に示す。

記述例：reactTest07.html

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8" />
5   <title>reactTest07</title>
6   <script src="react.development.js"></script>
7   <script src="react-dom.development.js"></script>
8   <script src="babel.min.js"></script>
9 </head>
10 <body>
11   <div id="root"></div>
12   <script type="text/babel">
13     function App() {
14       const fruits = ["りんご", "みかん", "ブドウ", "バナナ"];
```

```

15     return (
16       <ul>
17         { fruits.map( (e,i)=>{ return ( <li key={i}>{e}</li> ); } ) }
18       </ul>
19     );
20   }
21   // DOMにレンダリング
22   const domRoot = ReactDOM.createRoot(root);
23   domRoot.render( <App /> );
24 </script>
25 </body>
26 </html>

```

これは、配列 fruits から map メソッドで li 要素を展開する例で、これを Web ブラウザで表示すると右のような箇条書きが表示される。

各 li 要素には識別のための key 属性を付与している。

● りんご
● みかん
● ブドウ
● バナナ

配列から表をレンダリングする例 reactTest08.html を次に示す。

記述例：reactTest08.html

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="UTF-8" />
5    <title>reactTest08</title>
6    <style>
7      td { border: 1px solid gray;}
8    </style>
9    <script src="react.development.js"></script>
10   <script src="react-dom.development.js"></script>
11   <script src="babel.min.js"></script>
12 </head>
13 <body>
14   <div id="root"></div>
15   <script type="text/babel">
16     function App() {
17       const tbl = [[11,12,13],
18                   [21,22,23],
19                   [31,32,33]];
20
21       return (
22         <table><tbody>
23           {tbl.map((row, rowIndex) => (
24             <tr key={rowIndex}>
25               {row.map((cell, cellIndex) => (
26                 <td key={cellIndex}>{cell}</td>
27               ))}
28             </tr>
29           ))}
30         </tbody></table>
31       );
32     }
33     // DOMにレンダリング
34     const domRoot = ReactDOM.createRoot(root);
35     domRoot.render( <App /> );
36   </script>
37 </body>
</html>

```

これは、配列 tbl から map メソッドで tr, td 要素を展開する例で、これを Web ブラウザで表示すると右のような表が表示される。

各 tr, td 要素には識別のための key 属性を付与している。

11	12	13
21	22	23
31	32	33

11.2 実用的な使用方法

先に解説した React の使用方法¹²⁴ は最も素朴なものであるが、実際の Web アプリケーションの開発作業においては、多数のライブラリの依存関係の管理やソースコードの分割管理に関する煩雑さが問題となる。従って、現実的には npm などの開発用のツールを用いて Web アプリケーションを開発することが一般的である。

npm は npm, Inc. が開発して管理するソフトウェアツールであり Node.js に基づいている。計算機環境に Node.js をインストールすると npm も同時にインストールされる。npm は JavaScript プログラムの開発作業を支える標準的なツールとして普及しており、本書ではこれを前提として開発作業について解説する。

11.2.1 npm による開発作業の概観

React による Web アプリケーション開発はプロジェクトとして 1 つのディレクトリで管理するのが一般的である。すなわち、

- Web アプリケーションの全体となる HTML ファイル（*.html）
- JavaScript のプログラムファイル（*.js）
- 必要となるライブラリ群のファイル

を 1 つのプロジェクトディレクトリにまとめて開発作業を管理する。

HTML ファイルや JavaScript ファイルとして記述された Web アプリケーションは npm コマンドによるビルド作業によって最終的に実行可能なものとなる。このビルド作業に先立って、必要なライブラリをインストールし、ビルド作業に必要な設定ファイルを作成する。また、開発に関する各種の作業は基本的にコマンドシェル（ターミナルウィンドウ、コマンドプロンプトウィンドウ）で実行する。

11.2.1.1 必要なソフトウェアのインストール

Web アプリケーションのプロジェクトディレクトリで次のようにコマンドを発行することで react, react-dom が当該プロジェクト配下にインストールされる。

コマンド： `npm install react react-dom`

この作業により、インストール作業の報告が表示された後、当該プロジェクト配下に `node_modules` ディレクトリが作成され、その中に必要なソフトウェア群が配置される。また、開発作業に必要な設定ファイル（*.json）も作成される。

この他にも JSX のトランスパイルのための Babel と、ビルド作業のための webpack 関連ツール（下記）が必要となる。

■ Babel 関連ツール

babel/core:

Babel のコアパッケージ。

babel/preset-env:

ブラウザごとの対応バージョンを考慮して必要なトランスパイルを実行するためのツール。

babel/preset-react:

React の JSX 構文をブラウザが実行できる形式に変換するために必要。

babel-loader:

Webpack で Babel を使用するためのローダー。

■ webpack 関連ツール

webpack:

複数の JavaScript ファイルやリソースを 1 つのバンドルファイルにまとめるモジュールバンドラ。

webpack-cli:

webpack のコマンドラインインターフェースで、ビルド作業や開発サーバーの起動などを行う。

¹²⁴ ローカル環境や CDN から script 要素でライブラリを読み込んで使用する方法。

webpack-dev-server:

開発用のローカルサーバーを提供し、ファイルの変更を監視して自動的に再ビルドする機能を実現する。

html-webpack-plugin:

webpack で HTML ファイルを生成するためのプラグイン。ビルドした JavaScript ファイルを自動的に HTML に挿入する。

具体的には次のようにして必要なソフトウェアをプロジェクトにインストールする。

```
コマンド: npm install --save-dev @babel/core @babel/preset-env @babel/preset-react
          babel-loader webpack webpack-cli webpack-dev-server html-webpack-plugin
```

このコマンドを1行で入力すること。この処理によって、node_modules ディレクトリ配下に必要なものがインストールされる。

11.2.1.2 ソースコードの用意

Web アプリケーションを構成する HTML ファイルと JavaScript ファイルを用意する。HTML ファイルはプロジェクトディレクトリに index.html というファイル名で作成することが推奨される。また HTML ファイルに読み込まれる JavaScript ファイルはプロジェクトディレクトリのサブディレクトリ src に index.js というファイル名で作成することが推奨される。

次に示す例は、プロジェクトディレクトリ buildTest02 下に index.html と src/index.js を作成するものである。

記述例: buildTest02/index.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>buildTest02</title>
6 </head>
7 <body>
8   <div id="root"></div>
9   <script src="src/index.js"></script>
10 </body>
11 </html>
```

記述例: buildTest02/src/index.js

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 const e = <h1>Hello, world!</h1>; // React要素
5
6 // React要素をDOMにレンダリング
7 const domRoot = ReactDOM.createRoot(root);
8 domRoot.render(e);
```

この例は先の p.241 で示した reactTest02.html を npm でビルドするアプリケーションの形式として改変したものである。buildTest02/index.html が Web アプリケーションの最上位の HTML であり、これが buildTest02/src/index.js を読み込んで実行する。

通常の場合であれば、import 文を使う JavaScript はモジュールの形式で扱う必要があるが、npm で React アプリケーションとしてビルドする場合は、Babel によるトランスパイル過程でこの問題は解消される。

11.2.1.3 ビルド作業のための設定ファイルの準備

Babel のための設定ファイル .babelrc を次のような内容を与えてプロジェクトのディレクトリに作成する。

記述例: buildTest02/.babelrc

```
1 {
2   "presets": ["@babel/preset-env", "@babel/preset-react"]
3 }
```

参考) babel.config.js という設定ファイルを作成する方法もある。

ビルド作業を行う webpack のための設定ファイル webpack.config.js を次のような内容を与えてプロジェクトのディレクトリに作成する。

記述例：buildTest02/webpack.config.js

```
1 // 必要なモジュールの読み込み
2 const path = require('path');
3 const HtmlWebpackPlugin = require('html-webpack-plugin');
4
5 module.exports = {
6   mode: 'development',
7   entry: './src/index.js',    // エントリとなる最上位の JavaScript ファイル
8   output: {
9     path: path.resolve(__dirname, 'dist'), // ビルド結果を収めるディレクトリ
10    filename: 'bundle.js',         // ビルド結果の JavaScript ファイル
11  },
12  module: {
13    rules: [
14      {
15        test: /\.js$/,           // ファイルのタイプ（拡張子）
16        exclude: /node_modules/, // ルールの適用除外
17        use: {
18          loader: 'babel-loader', // 使用するローダー
19        },
20      },
21    ],
22  },
23  plugins: [                      // 使用するプラグイン
24    new HtmlWebpackPlugin({
25      template: './index.html',   // ソースの HTML
26      filename: 'index.html',     // ビルド結果の HTML のファイル名
27      inject: 'head'              // ビルド結果の JavaScript を収める要素
28    }),
29  ],
30  devServer: {                    // 開発用 Web サーバに関する設定
31    contentBase: path.join(__dirname, 'dist'), // ビルド結果を収めるディレクトリ
32    compress: true,                // 圧縮指定
33    port: 9000,                   // 開発用 Web サーバのポート番号
34  },
35 };
```

この設定で、プロジェクトディレクトリ下の HTML ファイル index.html と src ディレクトリ下の JavaScript ファイル index.js からビルドした Web アプリケーションを dist ディレクトリ下に出力する。また「mode: 'production',」とするとビルド後の JavaScript ファイルが最適化される。

先のインストール作業で作成された package.json を編集し、次のようなエントリを追加する。

記述例：（追加部分）

```
1 "scripts": {
2   "test": "echo \"Error: no test specified\" && exit 1",
3   "start": "webpack serve",
4   "build": "webpack"
5 }
```

11.2.1.4 ビルド作業の実行

設定ファイルの用意が完了した後、次のようなコマンドをプロジェクトディレクトリで発行してビルド作業を実行する。

コマンド： npm run build

コマンドを発行するとビルド作業に関する報告が出力され、エラーがなければ dist ディレクトリにビルド結果（index.html, bundle.js）が出力される。dist/index.html を Web ブラウザで開くことで当該 Web アプリケーションが起動する。

先に例示したプロジェクト buildTest02 では buildTest02/dist/index.html と buildTest02/dist/bundle.js がビルド作業によって作成される。

Hello, world!

図 78: プロジェクト buildTest02 でビルドされた Web アプリケーション

11.2.2 ソースコードの分割開発

実際の Web アプリケーションの開発作業においては、各種コンポーネントの定義などを別々のソースコードとして分割して作成することが一般的である。ここでは、先の p.249 で示した reactTest05-3.html を各パートに分離した形で実装する例を示す。

記述例：buildTest05/index.html （Web アプリケーションの全体を成す HTML）

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8" />
5   <title>buildTest05</title>
6 </head>
7 <body>
8   <div id="root"></div>
9   <script src="src/index.js"></script>
10 </body>
11 </html>
```

記述例：buildTest05/src/index.js （上記 HTML から読み込む JavaScript プログラム）

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 import Wrapper1 from './comp1.js'; // 別ファイルからコンポーネント
5 import Wrapper2 from './comp2.js'; // を読み込む
6
7 // Reactアプリを関数コンポーネントとして実装する
8 function App() {
9   return (
10     <>
11       <Wrapper1 />
12       <Wrapper2 />
13     </>
14   )
15 }
16
17 const domRoot = ReactDOM.createRoot(root);
18 domRoot.render( <App /> );
```

上の例では、移動するボールを表示するコンポーネントを包含するラッパーが Wrapper1, Wrapper2 としてレンダリングされる。各ラッパーとその内部の定義は次の comp1.js, comp2.js として個別のソースプログラム（モジュール形式）として作成した。

記述例：buildTest05/src/comp1.js

```
1 import React from 'react';
2
3 function Wrapper1() {
4   const [c1, setC1] = React.useState(true);
5   // ボールを表示するクラスコンポーネント
6   class B1 extends React.Component {
7     // コンストラクタ
8     constructor() {
9       super();
10      this.state = { pos: 0 }; // Stateの用意
11    }
12    // DOMマウント時の準備処理
13    componentDidMount() {
14      console.log(" B1がマウントされた");
15      this.timerID = setInterval(() => {
16        this.setState({pos: this.state.pos + 2});
17        console.log(" B1が移動した");
```

```

18         }, 20);
19     }
20     // 再レンダリング時点で終了をチェック
21     componentDidMount() {
22         if( this.state.pos >= 200 ) this.props.setC(false);
23     }
24     // アンマウント時の終了処理（クリーンアップ）
25     componentWillUnmount() {
26         console.log(" B1がアンマウントされた");
27         clearInterval(this.timerID);
28     }
29     // このコンポーネントのレンダリング
30     render() {
31         // ボールが右端に到達するまで移動する処理
32         if ( this.state.pos < 200) {
33             const s = { position:"absolute",
34                         left:this.state.pos+"px", top:"0px" };
35             return (  );
36         }
37     }
38 }
39 // コンポーネント B1のレダリング
40 return ( c1 && <B1 c={c1} setC={setC1} /> );
41 }
42
43 export default Wrapper1

```

記述例：buildTest05/src/comp2.js

```

1  import React from 'react';
2
3  function Wrapper2() {
4      const [c2,setC2] = React.useState(true);
5      // ボールを表示する関数コンポーネント
6      function B2(p) {
7          const [pos,setPos] = React.useState(0);    // Stateの用意
8          // DOMマウント時の準備処理
9          React.useEffect( () => {
10              console.log(" B2がマウントされた");
11              const timerID = setInterval(() => {
12                  setPos( p => p+1 );
13                  console.log(" B2が移動した");
14              }, 20);
15              // クリーンアップ関数
16              return () => {
17                  clearInterval(timerID);
18                  console.log(" B2がアンマウントされた");
19              };
20          }, []);
21          // posが変更した時点で終了をチェック
22          React.useEffect( () => {
23              if (pos >= 200) p.setC(false);
24          }, [pos]);
25          // ボールが右端に到達するまで移動する処理（レンダリング）
26          if ( pos < 200) {
27              const s = { position:"absolute",
28                          left:pos+"px", top:"60px" };
29              return (  );
30          }
31      }
32      // コンポーネント B2のレダリング
33      return ( c2 && <B2 c={c2} setC={setC2} /> );
34  }
35
36  export default Wrapper2

```

これらモジュールが提供する Wrapper1, Wrapper2 を先の index.js が読み込んでレンダリングする。使用する画像ファイル BallGray02.png は JavaScript ファイルと同じ src ディレクトリに配置して開発作業を行う。

アプリケーションのビルド作業は先に解説した buildTest02 の場合と概ね同じであるが、今回のケースでは HTML,

JavaScript 以外にも画像ファイル BallGray02.png も扱っており、ビルド作業で全てのリソースを目的のディレクトリ dist に複製するには次に解説する copy-webpack-plugin を使用する。

11.2.3 ビルド作業時のリソース複製の設定：copy-webpack-plugin

実際の Web アプリケーション開発においては、画像データをはじめとする様々なリソースが用いられる。それら必要なリソースはプロジェクトの src ディレクトリに収めて開発作業を進めるのが一般的であるが、ビルド作業において目的のディレクトリ（dist）にもそれらの複製を配置する必要がある。HTML、JavaScript 以外のファイルを手作業で目的のディレクトリに複製しても良いが、copy-webpack-plugin を使用するとその作業を自動化することができる。

copy-webpack-plugin をプロジェクトにインストールするには次のような、プロジェクトのディレクトリで次のようなコマンドを発行する。

コマンド： npm install --save-dev copy-webpack-plugin

この処理によって node_modules ディレクトリ配下に copy-webpack-plugin がインストールされる。

copy-webpack-plugin を使用するには、ビルド作業を行う webpack のための設定ファイル webpack.config.js の冒頭文に次の 1 行を加える。

追加する行： const CopyWebpackPlugin = require('copy-webpack-plugin');

必要なリソースを開発用ディレクトリからビルド先のディレクトリに複製する設定は「plugins:」の部分に次のような要素を追加する。

《リソース複製のための設定》

```
new CopyWebpackPlugin(  
  patterns: [  
    { from: 複製元のリソースのパス, to: 複製先のリソースのファイル名 }  
  ]  
)
```

from, to にはワイルドカードが使える。

「patterns:」の部分の記述例を次に示す。

```
{ from: 'src/img01.png', to: 'img01.png' }, // src/img01.png を dist ディレクトリにコピー  
{ from: 'src/img02.png', to: 'img02.png' }, // src/img02.png を dist ディレクトリにコピー  
{ from: 'src/*.png', to: '[name][ext]' }    // src 配下の全ての png 画像を dist ディレクトリにコピー
```

11.2.4 コンポーネント実装のための簡便な方法

先に解説したように、コンポーネントの実装にはクラス定義の形式（クラスコンポーネント）と関数定義の形式（関数コンポーネント）がある。クラスコンポーネントはオブジェクト指向プログラミングの考え方が応用できる一方で、記述自体が長く複雑になりがちである。これに対してアロー関数式に基づく関数コンポーネントは記述が簡潔になるので React アプリの開発者に好まれる傾向があり、実際にこの形式による実装が主流になりつつある。（次の例）

記述例：buildTest02-2/index.html

```
1 <!DOCTYPE html>  
2 <html lang="ja">  
3 <head>  
4   <meta charset="utf-8">  
5   <title>buildTest02-2</title>  
6 </head>  
7 <body>  
8   <div id="root"></div>  
9   <script src="src/index.js">  
10     </script>  
11 </body>  
12 </html>
```

記述例：buildTest02-2/src/index.js

```
1 import React from 'react';  
2 import ReactDOM from 'react-dom';  
3  
4 // アロー関数式による簡単な  
5 // 関数コンポーネントの実装  
6 const App = ()=>  
7   <>  
8     <h1>Hello, world!</h1>  
9     <p>This is a React application.</p>  
10   </>  
11  
12 // コンポーネント App を DOM にレンダリング  
13 const domRoot = ReactDOM.createRoot(root);  
14 domRoot.render( <App /> );
```

このプロジェクト buildTest02-2 では、関数コンポーネント App が純粋な HTML に近い形式で実現できていることがわかる。

Hello, world!

This is a React application.

図 79: buildTest02-2 をビルドして Web ブラウザで表示した様子

付録

A Webサーバ関連

A.1 Node.js による HTTP サーバ

Node.js は V8 JavaScript エンジン上に構築された JavaScript 実行環境の 1 つであり、ファイル入出力や通信のための機能を持ち、Web サーバなどのサーバサイドの情報処理機能を実装することができる。ここでは主に、表 61 に示すライブラリを用いて HTTP サーバの機能を Node.js で実現するための最も基本的な方法について解説する。

表 61: 使用するライブラリ

ライブラリ	解 説
http	HTTP 通信に関する機能を提供する。
url	URL を解析するための機能を提供する。
querystring	URL に含まれるクエリ文字列を解析するための機能を提供する。
express	Web アプリケーションのためのフレームワーク

表 61 の内、http、url、querystring が最も基本的なものである。本書ではまずこれらライブラリによる HTTP サーバ実装の方法について解説し、後の「A.1.5 Express ライブラリによる HTTP サーバ」(p.266) で express について解説する。

A.1.1 ライブラリの読み込み

次のように記述して上記のライブラリを Node.js のプログラムに読み込んで使用するのが一般的である。

```
const http = require('http');
const url = require('url');
const querystring = require('querystring');
```

これによって、それぞれのライブラリをその名称のオブジェクトとして扱うことができる。本書でもこの慣例に従うこととする。

A.1.2 サーバの作成と起動

http ライブラリの createServer によって HTTP サーバを作成することができる。

書き方： http.createServer(関数)

サーバの機能を実現する「関数」を与えてサーバ (Server クラスのインスタンス) を作成して返す。ここで与える「関数」は 2 つの引数を取るもので、

関数 (リクエストオブジェクト, 応答オブジェクト)

の形式で実行されるものである。「リクエストオブジェクト」 (IncomingMessage クラス) はクライアントから受信したリクエストに関する情報を保持するもので、「応答オブジェクト」 (ServerResponse クラス) はクライアントに対する応答処理に関するものである。この「関数」は特にリクエストハンドラ¹²⁵と呼ばれる。

上の処理で得られたサーバを起動するには listen メソッドを実行する。

書き方： サーバ.listen(ポート番号, ホスト名, 関数)

「ホスト名」の「ポート番号」からのリクエストを受信して処理する「サーバ」を起動する。また、起動時の処理を実行する「関数」を与える。この「関数」は特にリスニングハンドラ¹²⁶と呼ばれる。「ホスト名」には当該サーバシステムのネットワークインターフェースのホスト名もしくは IP アドレスを与える。

¹²⁵ リクエストリスナと呼ばれることもある。

¹²⁶ リスニングリスナとも呼ばれる。

A.1.3 サーバの処理の実装

クライアントからのリクエストに応える処理は `http.createServer` でサーバを作成する際に与える関数で行う。具体的にはその関数で、

1. リクエストオブジェクトの解析
2. 応答オブジェクトに対する各種の処理

の処理を行う。

上記 1 の処理で特に重要なものとして、クライアントが送信したリクエストから URL を取り出し、それに含まれるクエリ文字列を取得することがある。この処理には `url`, `querystring` ライブラリの機能を用いる。

A.1.3.1 URL の解析

文字列として表現された URL は `url` ライブラリの `parse` メソッドで解析することができる。

書き方： `url.parse(URL 文字列)`

「URL 文字列」を解析した結果を `Url` オブジェクトとして返す。

コマンドシェル上で `Node.js` を起動して `url` ライブラリを読み込み、URL を解析する処理を次に示す。(簡単のため変数の宣言は省略する)

例. ライブラリの読み込みと URL の解析

```
> url = require("url")  [Enter]    ← url ライブラリの読み込み
{ (省略) }

> u = "http://www.mysite.com:3000/Welcome.cgi?名 1=値 1&名 2=値 2" [Enter] ← URL (文字列) の作成
'http://www.mysite.com:3000/Welcome.cgi?名 1=値 1&名 2=値 2'

> U = url.parse( u )  [Enter]    ← Url オブジェクトに変換
Url { (省略) }
```

`Url` オブジェクトが得られている。このオブジェクトは表 62 に示すようなプロパティを持ち、URL の各部の値を文字列の形で保持している。

表 62: `Url` オブジェクトのプロパティ (一部)

プロパティ	解説
<code>href</code>	解析後の URL
<code>protocol</code>	URL のプロトコル部分 (例: 'http:')
<code>hostname</code>	URL のホスト名の部分 (例: 'www.mysite.com')
<code>port</code>	URL のポート番号の部分
<code>pathname</code>	URL のパスの部分
<code>search</code>	URL のクエリ文字列の部分 ('?' を含む)
<code>query</code>	URL のクエリ文字列の部分 ('?' を除く)
<code>hash</code>	URL のフラグメント識別子の部分 ('#' を含む)

例. `Url` オブジェクトのプロパティ (先の例の続き)

```
> U.href  [Enter]
'http://www.mysite.com:3000/Welcome.cgi?名 1=値 1&名 2=値 2'

> U.protocol  [Enter]
'http:'

> U.hostname  [Enter]
'www.mysite.com'

> U.port  [Enter]
'3000'

> U.pathname  [Enter]
'/Welcome.cgi'

> U.search  [Enter]
```

```
'?名 1=値 1&名 2=値 2'  
> U.query   
' 名 1=値 1&名 2=値 2'
```

フラグメント識別子（html 要素の id 属性の値）を持つ URL を解析する例を次に示す。

例. フラグメント識別子を持つ URL の解析（先の例の続き）

```
> u2 = "http://www.mysite.com:3000/HomePage.html#section5"   
'http://www.mysite.com:3000/HomePage.html#section5'  
> U2 = url.parse( u2 )   
Url { (省略) }  
> U2.pathname   
'/HomePage.html'  
> U2.hash   
'#section5' ←フラグメント識別子が得られている
```

A.1.3.2 クエリ文字列の解析

文字列として表現された**クエリ文字列**は querystring ライブラリの parse メソッドで Object クラスのインスタンスにすることができる。

書き方: `querystring.parse(クエリ文字列)`

「クエリ文字列」を Object クラスに変換したものを返す。

コマンドシェル上で Node.js を起動してこれらライブラリを読み込み、URL を解析する処理を次に示す。（簡単のため変数の宣言は省略する）

例. ライブラリの読み込みと URL の解析

```
> querystring = require("querystring")  ← querystring ライブラリの読み込み  
{ (省略) }  
> Q = querystring.parse(U.query)  ←クエリ文字列の取り出しと変換  
[Object: null prototype] { '名 1': '値 1', '名 2': '値 2' } ←クエリのオブジェクト
```

A.1.3.3 リクエストオブジェクトの解析

リクエストハンドラ内では、応答処理を始める前にクライアントからのリクエストの情報を持つ**リクエストオブジェクト**を解析する。リクエストオブジェクトのプロパティを表 63 に挙げる。

表 63: リクエストオブジェクト req のプロパティ（一部）

プロパティ	解 説
req.url	クライアントが送信したリクエストの中の URL ※ プロトコル、ホスト、ポートなどに関する情報は削除されていることに注意
req.method	リクエストメソッド（GET、POST など）の文字列
req.headers	リクエストのヘッダ情報を持つオブジェクト（Object）

上の表の内、req.headers も有用なプロパティ（表 64）を持つ。特に上の表の req.url はプロトコル、ホスト、ポートなどの情報が削除されているので、それらに関する情報は下の表 64 に示すプロパティから取得する。

A.1.3.4 クライアントへの応答のための処理

リクエストハンドラ内では**応答オブジェクト**を用いてクライアントに対する応答を送信する。この場合の処理手順を次に示す。

1. 応答ヘッダ（Response Header）の送信
2. 応答データの送信
3. 応答終了の通知

表 64: req.headers のプロパティ (一部)

プロパティ	解 説
req.headers.host	ホスト名とポート番号 (例: localhost:3000)
req.headers["user-agent"]	リクエストを送信したエージェントの情報
req.headers.accept	クライアントが扱うことができるコンテンツタイプ
req.headers["accept-language"]	クライアントが解釈できる言語
req.headers.authorization	クライアントの認証情報
req.headers.cookie	クライアントから送信されたクッキー

応答ヘッダ (Response Header) とは、サーバからの応答内容に関する情報をクライアントに通知するもので、**応答ステータス**¹²⁷ やメディアタイプ (MIME タイプ) などから成る。応答ヘッダの送信には、応答オブジェクトに対して `writeHead` メソッドを実行する。

書き方: 応答オブジェクト.writeHead(ステータスコード, ヘッダオブジェクト)

「ステータスコード」には応答ステータスを意味する整数値を、「ヘッダオブジェクト」にはヘッダのキーに対する値を組にしたオブジェクト (Object インスタンス) を与える。ヘッダのキーとして特に重要なものが "Content-Type" で、これにメディアタイプ (MIME タイプ) を指定する。

例. 応答オブジェクト `res` を介した応答ヘッダの送信

```
res.writeHead(200, {'Content-Type': 'text/plain; charset=utf-8'});
```

これは、utf-8 エンコーディングのテキストコンテンツを成功裡 (ステータスコード 200) に作成してクライアントに送信する場合の例である。

応答ヘッダを送信した後は、応答オブジェクトに対して `write` メソッドを実行して応答内容のデータ (応答ボディ) をクライアントに送信する。

書き方: 応答オブジェクト.write(チャンク)

「チャンク」には文字列あるいは Buffer オブジェクトを与える。このメソッドは複数回に渡って実行することができ、応答ボディを段階的に送信することができる。

応答ボディの送信が終われば、応答の終了をクライアントに通知する必要がある。これには、応答オブジェクトに対して `end` メソッドを実行する。

書き方: 応答オブジェクト.end(チャンク)

`write` メソッドと同様に「チャンク」を応答ボディとして送信することができる。全体として小さな応答ボディを送信する場合は `write` メソッドを使用せずに `end` メソッドだけで送信を完結することもできる。

A.1.3.5 POST メソッドで受け取ったデータの受理

クライアントから POST メソッドで送信されたデータは、サーバ側の受信時には小さな**チャンク**に分割されていることが一般的である。それらデータのチャンクはリクエストオブジェクトの `data` イベントを受けて逐次受理する。また全てのチャンクの受理が完了するとリクエストオブジェクトに `end` イベントが発生する。

書き方: リクエストオブジェクト.on("data", 関数)

POST されたデータの次のチャンクが利用可能になった時に "data" イベントが起こり「関数」(イベントハンドラ) を起動する。このとき「関数」の第 1 引数には次のチャンクが与えられる。関数に与えられるチャンクは Buffer オブジェクトであり、これに対して `toString` メソッドを実行することで文字列に変換することができる。

参考)

リクエストオブジェクト `.setEncoding('utf-8')`

を実行しておくと「関数」に utf-8 エンコーディングされた文字列が渡される。

¹²⁷HTTP ステータスコードとも呼ばれる。特に重要なものを p.290 の表 76 に掲載している。

■ 全チャンクの読み取り完了時の処理

書き方： リクエストオブジェクト.on("end", 関数)

POST されたデータを全て受理し終えた時に "end" イベントが起こり「関数」（イベントハンドラ）を起動する。「関数」の引数は省略することができる。

A.1.3.6 エラーハンドリング（例外処理）

リクエストオブジェクトは error イベントをハンドリングすることができ、何らかのエラーが発生した場合の例外処理が可能である。

書き方： リクエストオブジェクト.on("error", 関数)

リクエストオブジェクトの処理にエラーが発生した時に "error" イベントが起こり「関数」（イベントハンドラ）を起動する。このとき「関数」の第1引数には Error オブジェクトが与えられる。

A.1.4 サーバの実装例

A.1.4.1 単純な例

次に示す NodeHTTPsv01.js はクライアントからのリクエストの内容をテキストデータとして応答する単純なサンプルである。

記述例：NodeHTTPsv01.js（サーバスクリプト）

```
1  const http = require('http');
2  const url = require('url');
3  const querystring = require('querystring');
4
5  const server = http.createServer((req, res) => {
6      const U = url.parse(req.url);
7      const Q = querystring.parse(U.query);
8      let msg = 'Query:\t\t' + decodeURIComponent(U.query) + '\n' +
9              '\t\t--> ' + JSON.stringify(Q) + '\n' +
10             'Host:\t\t' + req.headers.host + '\n' +
11             'Method:\t\t' + req.method + '\n' +
12             'UA:\t\t' + req.headers['user-agent'] + '\n' +
13             'Accept:\t\t' + req.headers.accept + '\n' +
14             'AcceptLang:\t\t' + req.headers['accept-language'] + '\n' +
15             'Authorization:\t\t' + req.headers.authorization + '\n' +
16             'Cookie:\t\t' + req.headers.cookie + '\n';
17      res.writeHead(200, {'Content-Type': 'text/plain; charset=utf-8'});
18      res.write( msg );
19      res.end( '--- 以上 ---' );
20  });
21
22  server.listen(3000, '127.0.0.1', () => {
23      console.log('サーバが http://127.0.0.1:3000 で起動した。');
24  });
```

上記のサーバスクリプトをローカルの計算機環境（localhost）で実行し、クライアント（Web ブラウザ）からアクセスする例を示す。

ローカルの計算機環境のコマンドシェルで次のように操作して Node.js で上記サーバスクリプト実行する。

コマンド： node NodeHTTPsv01.js

この結果、標準出力（コマンドウィンドウ）に「サーバが http://127.0.0.1:3000 で起動した。」と出力され、サーバが起動してリクエストを受け付ける状態となる。次に、同じ計算機で Web ブラウザを起動して下記 URL にアクセスする。

URL： http://localhost:3000/?名1=値1&名2=値2

これにより、Web ブラウザからのリクエストを先のサーバスクリプトが受け付けて応答し、Web ブラウザの表示が次のようになる。

```

Query:      名 1=値 1&名 2=値 2
            --> {"名 1":"値 1","名 2":"値 2"}
Host:       localhost:3000
Method:     GET
UA:         Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:127.0) Gecko/20100101 Firefox/127.0
Accept:     text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
AcceptLang: ja,en-US;q=0.7,en;q=0.3
Authorization: undefined
Cookie:     ... (Cookie 情報) ...
--- 以上 ---

```

A.1.4.2 POST されたデータを受理する例

クライアントから POST メソッドで受け取ったデータをテキストデータにして返信するサンプル NodeHTTPsv02.js を示す。

記述例：NodeHTTPsv02.js (サーバスクリプト)

```

1  const http = require('http');
2  const url = require('url');
3  const querystring = require('querystring');
4
5  const server = http.createServer((req, res) => {
6      const U = url.parse(req.url);
7      const Q = querystring.parse(U.query);
8      let msg = 'Host:\t' + req.headers.host + '\n' +
9              'Method:\t' + req.method + '\n';
10     res.writeHead(200, {'Content-Type': 'text/plain; charset=utf-8'});
11     res.write( msg );
12     // POSTメソッドの場合の処理
13     if ( req.method == "POST" ) {
14         res.write( '--- POSTされたデータ ---\n' );
15         let dat = ''; // データを受け取る変数
16         // dataイベントでPOSTされたデータを逐次受け取る
17         req.on('data', chunk => {
18             dat += chunk.toString();
19         });
20         // endイベントでPOSTデータの終了を検知
21         req.on('end', () => {
22             dat += '\n';
23             res.write( decodeURIComponent(dat) );
24             res.end( '--- 以上 ---' );
25         });
26     } else { // POSTメソッドではない場合
27         res.end( '--- 以上 ---' );
28     }
29 });
30
31 server.listen(3001, '127.0.0.1', () => {
32     console.log('サーバが http://127.0.0.1:3001 で起動した。');
33 });

```

このサーバスクリプトを Node.js に与えてサーバとして起動する。次に同じ計算機環境の Web ブラウザで下記の NodeHTTPsv02.html を開いて上記サーバにリクエスト（POST メソッド）を送信する。

記述例：NodeHTTPsv02.html

```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4      <meta charset="utf-8">
5      <title>NodeHTTPsv02</title>
6      <style>
7          textarea {
8              width:200px; height: 70px;
9              border: solid 1px #555555;
10         }
11     </style>
12 </head>
13 <body>
14     <form action="http://localhost:3001/" method="POST">

```

```

15     <input type="submit" name="FormName1" value="SUBMIT(POST)"><br>
16     <textarea name="TextData1"></textarea>
17 </form>
18 </body>
19 </html>

```

これを Web ブラウザで開いた直後は図 80 の (a) のように表示される。

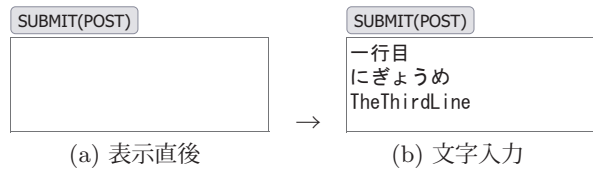


図 80: Web ブラウザでの表示

図 80 の (b) のようにテキストエリアに文字を入力して **SUBMIT(POST)** ボタンをクリックするとテキストエリアの内容が POST メソッドでサーバに送信される。その結果、サーバからそれを含んだテキストデータが返信され、Web ブラウザに次のように表示される。

```

Host:    localhost:3001
Method:  POST
--- POST されたデータ ---
FormName1=SUBMIT(POST)&TextData1=一行目
にぎょうめ
TheThirdLine
--- 以上 ---

```

A.1.5 Express ライブラリによる HTTP サーバ

Express.js は、TJ Holowaychuk 氏によって開発された、大規模な Web サービス開発にも耐えうる Web アプリケーションフレームワーク¹²⁸ である。Express.js 自体は小型で軽量であるが、各種のミドルウェアを組み込むことで様々な機能を追加することができる。本書では Express.js で HTTP サーバを実現するための最も基本的な方法について解説する。

Node.js のシステムに Express.js を導入するには、パッケージ管理ツールである npm を使用する。

インストール作業: `npm install express`

Express.js は、次のように記述して Node.js のプログラムに読み込んで使用するのが一般的である。

```
const express = require('express');
```

このようにして読み込まれた `express` は関数（Function クラスのインスタンス）である。

A.1.5.1 サーバの作成

`express` 関数を実行すると、HTTP サーバのインスタンスが得られる。

書き方: `express()`

これにより、サーバのインスタンス（EventEmitter クラス）が生成されて返される。

例. サーバのインスタンスの生成

```
const express = require('express');    ← Express.js ライブラリの読み込み
const app = express();                ← サーバのインスタンス app の生成
```

A.1.5.2 リクエストへの応答処理の実装

クライアントからのリクエストに対する処理（ルートハンドラ¹²⁹）は、表 65 に示すような、サーバのインスタンスに対するメソッド（ルーティングメソッド）で実装する。

表 65 に示すメソッドは 2 つの引数を与えて実行する。すなわち、

書き方: `メソッド (URL のパス, ルートハンドラ関数)`

¹²⁸<http://expressjs.com/>

¹²⁹HTTP 通信におけるリクエストハンドラのことである。ルーティングハンドラと呼ばれることもある。

表 65: ルーティングメソッド（ルートハンドラを実装するメソッド）

メソッド	解説
get	GET メソッドのリクエストに対する応答処理
post	POST メソッドのリクエストに対する応答処理
put	PUT メソッドのリクエストに対する応答処理
delete	DELETE メソッドのリクエストに対する応答処理
patch	PATCH メソッドのリクエストに対する応答処理
options	OPTIONS メソッドのリクエストに対する応答処理
all	全てのリクエストメソッドに対する応答処理

の形式で実行する。これにより「URL のパス」に対するリクエストを受けた際に起動する「ルートハンドラ関数」を登録する。またこの関数は 2 つの引数を取り、

関数 (リクエストオブジェクト, 応答オブジェクト)

という形で呼び出されるものとして実装する。以上のことを踏まえて、GET メソッドのリクエストに応えるルートハンドラを登録する例を次に示す。

例. GET メソッドのリクエストのハンドリング

```
app.get("/home", (req, res) => {           ←サーバインスタンス app によるハンドリングの実装
  res.send("Hello World!");               ←クライアントへの応答処理
});
```

これは URL のパス `/home` に対して GET メソッドのリクエストを受けた際のハンドリング（ルートハンドラ）をサーバインスタンス `app` に対して実装する例である。クライアントからのリクエストに関する情報は **リクエストオブジェクト** `req` が保持している。

この方法で、異なる複数の URL パスに対するルートハンドラをそれぞれ実装できる。すなわち、必要なだけ `get` メソッドを実行すれば良い。

クライアントへの応答に関する処理は **応答オブジェクト** `res` に対して行う。

■ リクエストオブジェクト

リクエストオブジェクトの重要なプロパティを表 66 に示す。

表 66: リクエストオブジェクト `req` のプロパティ（一部）

プロパティ	解説
<code>req.method</code>	リクエストメソッド（GET, POST など）を表す文字列
<code>req.headers</code>	リクエストの HTTP ヘッダーを含むオブジェクト
<code>req.url</code>	リクエストの URL を表す文字列
<code>req.path</code>	リクエストの URL のパスを表す文字列
<code>req.cookies</code>	クライアントから送信されたクッキーを含むオブジェクト (<code>cookie-parser</code> ミドルウェアが必要)
<code>req.body</code>	リクエストボディのデータ
<code>req.query</code>	クエリ文字列のデータを <code>Object</code> インスタンスにしたもの
<code>req.params</code>	ルートパラメータのデータを <code>Object</code> インスタンスにしたもの

■ 応答オブジェクト

応答オブジェクトの重要なメソッドを表 67 に示す。

表 67 の `send` メソッドは引数に与えられた「データ」の内容から、応答ボディのメディアタイプを自動的に設定（下記）して送信する。

表 67: 応答オブジェクト res のメソッド (一部)

プロパティ	解説
res.setHeader(項目, 値)	HTTP レスポンスヘッダの「項目」に「値」を設定する。
res.status(値)	HTTP ステータスコードとして「値」を設定する。
res.send(データ)	レスポンスボディに「データ」を設定してクライアントに送信する。
res.json(データ)	レスポンスボディに「データ」(JSON)を設定してクライアントに送信する。

- ・「データ」が文字列の場合は "text/html"
- ・「データ」が JavaScript のオブジェクト (Object クラス) や配列の場合は "application/json"
- ・「データ」がバッファ (Buffer クラス) または Uint8Array の場合は "application/octet-stream"

特にメディアタイプやステータスコードを意図して設定する場合は `setHeader`, `status` メソッドを用いる。これらメソッドは実行対象 (表の中の `res`) に設定処理を施した後、元の応答オブジェクトを返すので **メソッドチェーン** の形式が適用できる。(次の例)

例. 応答ボディのメディアタイプを "text/plain" にしてテキストを送信する

```
res.setHeader("Content-Type", "text/plain; charset=utf-8").send("テキストデータ");
```

表 67 の `json` メソッドで JSON を送信できるが、`send` メソッドを使用しても良い。

A.1.5.3 サーバの起動

サーバのインスタンスに対して `listen` メソッドを実行することでそれが HTTP サーバとして起動し、クライアントからのリクエストを受け付ける。

書き方: サーバのインスタンス.`listen`(ポート番号, ホスト名, 関数)

「ホスト名」に指定した NIC の「ポート番号」で受信するリクエストを受け付けるサーバとして起動する。「関数」はサーバ起動時に 1 回だけ実行されるもので、サーバの初期化に関する処理を行うものとして実装する。「関数」には引数を与えなくても良い。第 2 引数の「ホスト名」は省略可能で、その場合は当該計算機の全ての NIC が対象となる。

A.1.5.4 サーバの実装例 (1): リクエストのルーティング

表 65 に示す `get` メソッドを用いて、異なる URL パスへのリクエストをハンドリングする例 `ExpressSv01.js` を示す。

記述例: `ExpressSv01.js`

```
1  const express = require("express");           // Express.js のインポート
2  const app = express();                         // サーバインスタンスの生成
3
4  // URLパス "/" への GET リクエストに対するハンドラ
5  app.get("/", (req, res) => {
6      res.send("<html><body><h2>Welcome to my home page!</h2></body></html>");
7  });
8
9  // URLパス "/About" への GET リクエストに対するハンドラ
10 app.get("/About", (req, res) => {
11     res.send(
12         "<html><body><h2>This is a sample of request-routing.</h2></body></html>");
13 });
14
15 // ローカルホストの3000番ポートでサーバを起動する
16 app.listen( 3000, () => {
17     console.log("サーバが http://localhost:3000 で起動した");
18 });
```

このサーバスクリプトを Node.js で実行すると、2つの異なる URL パス "/" と "/About" への HTTP リクエストに対してそれぞれ異なるルートハンドラを起動する。このように、異なる URL パスに対してリクエストのハンドリングを選択することを **リクエストのルーティング** と表現する。

サーバと同じ計算機で Web ブラウザを起動して、それら 2つの URL パスにアクセスする例 (ブラウザの表示結果) を次に示す。

例. "http://localhost:3000/" へのリクエスト

→ Welcome to my home page!

例. "http://localhost:3000/About" へのリクエスト

→ This is a sample of request-routing.

A.1.5.5 サーバの実装例 (2)：リクエストの解析

クライアントから受信したリクエストに関する情報を解析し、その内容をテキストデータとしてクライアントに返信するサーバスクリプト ExpressSv02.js を次に示す。

記述例：ExpressSv02.js

```
1  const express = require("express");      // Express.js のインポート
2  const app = express();                  // サーバインスタンスの生成
3
4  // JSONデータ解析用ミドルウェア
5  app.use(express.json());
6
7  // URLエンコードされたデータの解析用ミドルウェア
8  app.use(express.urlencoded({ extended: true }));
9
10 // URLパス "/" へのリクエストに対するハンドラ
11 app.all("/", (req, res) => {
12     let r = "";
13     r += "Method:\t\t"      + req.method + "\n";
14     r += "URL:\t\t"         + decodeURIComponent(req.url) + "\n";
15     r += "Path:\t\t"        + req.path + "\n";
16     r += "QueryString:\t"   + JSON.stringify(req.query) + "\n";
17     r += "Host:\t\t"        + req.headers["host"] + "\n";
18     r += "UA:\t\t"          + req.headers["user-agent"] + "\n";
19     r += "Accept:\t\t"      + req.headers["accept"] + "\n";
20     r += "AcceptLanguage:\t" + req.headers["accept-language"] + "\n";
21     if ( req.method == "POST" ) {
22         r += "--- Posted Body ---\n" + JSON.stringify(req.body) + "\n";
23     }
24     res.setHeader("Content-Type", "text/plain; charset=utf-8");
25     res.send(r);
26 });
27
28 // ローカルホストの3000番ポートでサーバを起動する
29 app.listen( 3000, () => {
30     console.log("サーバが http://localhost:3000 で起動した");
31 });
```

これは、表 66 に示したリクエストオブジェクトのプロパティから各種の情報を取得してそれらをテキストデータにして返送するものである。このサーバスクリプトでは、クライアントから受信したデータボディを JSON として解釈するために express.json というミドルウェアを読み込んで使用している。また、URL エンコードされたデータボディを復元するために express.urlencoded ミドルウェアを使用している。

このスクリプトを Node.js で起動し、同じ計算機で起動した Web ブラウザから GET リクエストを送信する例を次に示す。

例. "http://localhost:3000/?名1=値1&名2=値2" へのリクエスト

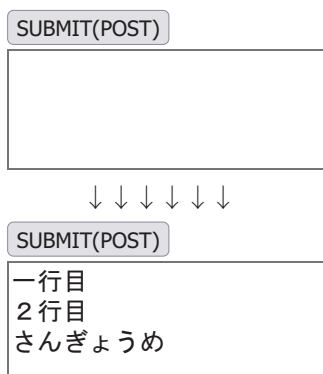
```
Method:      GET
URL:         /?名1=値1&名2=値2
Path:        /
QueryString: {"名1":"値1","名2":"値2"}
Host:        localhost:3000
UA:          Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept:      text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,⋯
AcceptLanguage: ja,en-US;q=0.7,en;q=0.3
```

POST メソッドによるリクエストに関して、次に示す ExpressSv02.html を Web ブラウザで表示して動作を確認する。

記述例：ExpressSv02.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>ExpressSv02</title>
6   <style>
7     textarea {
8       width:200px; height: 70px;
9       border: solid 1px #555555;
10    }
11  </style>
12 </head>
13 <body>
14   <form action="http://localhost:3000/" method="POST">
15     <input type="submit" name="FormName1" value="SUBMIT (POST)"><br>
16     <textarea name="TextData1"></textarea>
17   </form>
18 </body>
19 </html>
```

これを Web ブラウザで開くと右のように表示される。



コンテンツの textarea 要素の中に文字を入力して

SUBMIT(POST) ボタンをクリックすると form 要素の内容が POST メソッドで HTTP サーバに送信される。

textarea に文字を入力して SUBMIT する

その結果、サーバからの次のようなテキストデータが返信されて Web ブラウザに表示される。

例. 上記 HTML の SUBMIT 後の表示

```
Method:      POST
URL:         /
Path:        /
QueryString: {}
Host:        localhost:3000
UA:          Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept:      text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp, ...
AcceptLanguage: ja,en-US;q=0.7,en;q=0.3
--- Posted Body ---
{"FormName1":"SUBMIT(POST)","TextData1":"一行目¥r¥n 2行目¥r¥n さんぎょうめ"}
```

クライアントから受信した form データが JSON として受け取られ、”--- Posted Body ---” の下に表示されているのが確認できる。

A.1.5.6 サーバの実装例 (3)：ルートパラメータの取得

リクエストのルーティングにおいては URL のパスの一部をパラメータとして扱うことができる。例えば URL のパスに次のような記述あるとする。

/katsu

このような URL へのリクエストをハンドリングするために、ルーティングメソッド (p.267 の表 65) の第 1 引数に

```
"/:username"
```

という文字列を与えてルートハンドラを実装すると、リクエストオブジェクト req の params プロパティに

```
{ "username":"katsu" }
```

といったオブジェクトが得られる。すなわち、ルーティングメソッドに設定する URL パスの中に

" : パラメタ名 "

という記述があると、URL パス中の当該位置の文字列を「パラメタ名」のキーに対する値とするオブジェクトが req.params から得られる。実際にこれを応用したサーバスクリプト ExpressSv03.js を示す。

記述例：ExpressSv03.js

```
1  const express = require("express");      // Express.js のインポート
2  const app = express();                    // サーバインスタンスの生成
3
4  // URLパス "/" でのルートパラメータ「username」を取得
5  app.all("/:username", (req, res) => {
6      let r = "";
7      r += "Method:\t"      + req.method + "\n";
8      r += "URL:\t"         + decodeURIComponent(req.url) + "\n";
9      r += "Path:\t"        + req.path + "\n";
10     r += "User:\t"         + JSON.stringify(req.params) + "\n";
11     res.setHeader("Content-Type", "text/plain; charset=utf-8");
12     res.send(r);
13 });
14
15 // ローカルホストの3000番ポートでサーバを起動する
16 app.listen( 3000, () => {
17     console.log("サーバが http://localhost:3000 で起動した");
18 });
```

このスクリプトを Node.js で起動して、同じ計算機の Web ブラウザからリクエストを送信した場合の処理（Web ブラウザの表示）の例を次に示す。

例. "http://localhost:3000/katsu" のリクエスト

```
Method:  GET
URL:     /katsu
Path:    /katsu
User:    {"username": "katsu"}
```

例. "http://localhost:3000/taro" のリクエスト

```
Method:  GET
URL:     /taro
Path:    /taro
User:    {"username": "taro"}
```

A.2 Python による HTTP サーバ

ここでは、Python 言語処理系による HTTP サーバの構築方法についての最も基本的な事柄に関して解説する。

※ 本書では Python 言語の詳細に関する解説はしない¹³⁰。

A.2.1 http モジュールによる素朴な HTTP サーバ

公式の Python 言語処理系¹³¹ に標準添付されている http ライブラリは HTTP 通信に関する基本的な機能を提供する。また http.server モジュールは HTTP サーバを実現するための基本的な機能を提供する。

注意)

Python の http ライブラリで構築される HTTP サーバは、小規模のサービス構築、あるいはプロトタイピングのためのものであり、大規模な Web サービスの公開には適していない。本格的な HTTP サーバの機能を Python で実装するには「A.3 Apache HTTP Server」(p.281) で解説する CGI や WSGI の形を取るのが良い。

http.server モジュールが提供する重要なクラスとして HTTPServer, BaseHTTPRequestHandler があり、前者のインスタンスが HTTP サーバとなり、後者のインスタンス（リクエストハンドラ）がクライアントからのリクエストに対する応答処理を行う。BaseHTTPRequestHandler は抽象的なクラスであり、これを拡張したクラスに具体的な機能を実装する。

A.2.1.1 HTTPServer クラス

コンストラクタ： HTTPServer(サーバアドレス情報, リクエストハンドラのクラス)

「サーバアドレス情報」には次のような 2 要素のタプルを与える。

(NIC のアドレス, ポート番号)

「NIC のアドレス」にはクライアントからのリクエストを受け付ける NIC のホスト名もしくは IP アドレスを文字列で、「ポート番号」には整数値を与える。「リクエストハンドラのクラス」には BaseHTTPRequestHandler 継承した拡張クラスを与える。

HTTPServer クラスのインスタンスに serve_forever メソッドを実行することで当該 HTTP サーバが起動して、クライアントからのリクエストを受け付ける。

A.2.1.2 BaseHTTPRequestHandler クラス

BaseHTTPRequestHandler は抽象的なクラスであり、これを継承した拡張クラス（リクエストハンドラのクラス）の側に具体的なメソッドを実装する。クライアントからのリクエストに応えるためのメソッドは

do_リクエスト

という名前で実装する。(次の例)

例. リクエストハンドラ MyHandler クラス

```
class MyReqHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        (リクエストの解析と応答の処理)
        :
```

これは、リクエストハンドラ MyHandler クラスを定義し、POST メソッドによるリクエストをクライアントから受けた場合の処理を行うメソッド do_POST を実装する例である。

リクエストハンドラ（上の例の self）の属性やメソッドを表 68, 69 に示す。

表 68 の self.headers は辞書であり、表 69 に示すようなキーを持つ。

POST メソッドでクライアントから受信したデータは表 68 の self.rfile に対して read メソッドを用いて読み取る。この場合、read メソッドの引数には受信したデータのサイズを整数値で与える。具体的には、表 69 の 'Content-Length' の値を整数に変換したものを read メソッドの引数に与える¹³²。

¹³⁰Python 言語に関しては拙書「Python3 入門 -Kivy による GUI アプリケーション開発, サウンド入出力, ウェブスクレイピング-」でも解説しています。

¹³¹PSF が公開しているリファレンス実装としての CPython

¹³²HTTP チャンクエンコーディング（ヘッダ：Transfer-Encoding: chunked）でデータが POST される場合は別の方法を取る。

表 68: リクエストハンドラの属性：リクエスト解析用（一部）

属 性	解 説
self.command	クライアントからのリクエストメソッド（'GET', 'POST' など）を表す文字列
self.path	クライアントからのリクエスト URL のパス部分を表す文字列
self.headers	クライアントからのリクエストヘッダーを表すオブジェクト（http.client.HTTPMessage クラス）
self.rfile	クライアントからのリクエストボディを読み取るためのファイルのようなオブジェクト（io.BufferedReader クラス）

表 69: self.headers のキー（一部）

キー	解説
'User-Agent'	クライアント（Web ブラウザ）に関する情報を表す文字列
'Accept'	クライアントが扱うことのできるコンテンツタイプを表す文字列
'Host'	リクエストが送信されたホスト名とポート番号を表す文字列
'Content-Type'	リクエストボディのメディアタイプを表す文字列
'Content-Length'	リクエストボディのサイズを表す文字列

■ クライアントへの応答の送信

クライアントに応答する場合、最初にヘッダ情報を送信する。具体的には、リクエストハンドラに対して `send_header` でヘッダ情報を必要なだけ設定する。

書き方： `send_header(キー, 値)`

ヘッダ情報の「キー」（ヘッダ名とも言う）に対して「値」を設定する。「キー」と「値」は共に文字列で与える。`send_header` メソッドは応答前の設定を行うものであり、実際に応答としてヘッダを送信するには `end_headers` を用いる。

例. 応答前のヘッダの設定とクライアントへの送信

```
self.send_header('Content-type', 'text/html')
self.end_headers()
```

この後、クライアントに対する応答データを送信することができる。

リクエストハンドラには表 68 に挙げたもの以外にも、応答を送信するため属性 `wfile` も備えている。これはファイルのようなオブジェクト（`io.BufferedWriter` クラス）であり、`write` メソッドを用いることでクライアントに応答データを送信することができる。

書き方： `リクエストハンドラ.wfile.write(バイト列)`

「バイト列」をクライアントに送信する。

A.2.1.3 サーバの実装例

次に示す `PythonHTTPsv01.py` は GET, POST の 2 種類のリクエストに対して応答する単純な HTTP サーバの例である。このサーバはクライアントからのリクエストの情報をテキストの形式で応答するものである。

記述例：`PythonHTTPsv01.py`

```
1 # coding: utf-8
2 from http.server import BaseHTTPRequestHandler, HTTPServer
3 from urllib.parse import urlparse, parse_qs, unquote
4
5 # 受信したリクエストの解析
6 def DispReq(r):
7     # クエリ文字列の取得
8     parsed_path = urlparse(r.path)
9     query = parse_qs(parsed_path.query)
10    return \
11        'Host:\t\t' + r.headers.get('Host') + '\n' + \
12        'Path:\t\t' + unquote(r.path) + '\n' + \
13        'Query:\t\t' + str(query) + '\n' + \
```

```

14         'Method:\t\t' + r.command + '\n' + \
15         'Accept:\t\t' + r.headers.get('Accept') + '\n' + \
16         'UA:\t\t' + r.headers.get('User-Agent') + '\n' + \
17         'Content-Type:\t' + str(r.headers.get('Content-Type')) + '\n'
18
19 # 応答処理
20 def SendRes(r,d):
21     r.send_response(200)
22     r.send_header('Content-type', 'text/plain; charset=utf-8')
23     r.end_headers()
24     r.wfile.write( d.encode('utf-8') )
25
26 # 受信したリクエストを取り扱うクラス
27 class MyReqHandler(BaseHTTPRequestHandler):
28     # GETメソッドの場合の処理
29     def do_GET(self):
30         SendRes(self,DispReq(self))
31     # POSTメソッドの場合の処理
32     def do_POST(self):
33         c_len = int(self.headers.get('Content-Length')) # 受信したデータの長さ
34         dat = self.rfile.read(c_len) # 受信したデータの取得
35         msg = DispReq(self) + '--- POSTされたデータ ---\n' + \
36             unquote(dat.decode('utf-8')) + '\n--- 以上 ---'
37         SendRes(self,msg)
38
39 server_address = ('127.0.0.1', 8000)
40 httpd = HTTPServer(server_address, MyReqHandler)
41 print(f'http://{server_address[0]}:{server_address[1]}/...')
42 httpd.serve_forever()

```

このサーバでは、BaseHTTPRequestHandler クラスを拡張した MyReqHandler クラスで具体的な処理 (do.GET, do.POST メソッド) を実現している。クライアントへの応答内容の作成は DispReq 関数で行い、その結果得られたものを SendRes 関数でクライアントに送信する。クライアントがリクエストした URL の解析は urllib.parse モジュールの urlparse 関数で行い、更に parse_qs 関数でクエリ文字列の解析を行っている。また、URL エンコーディングされた文字列の復元には unquote 関数を用いている。

このサーバをローカルの計算機で起動し、同じ計算機上の Web ブラウザから次のような URL (クエリ文字列付き) をリクエストする。

`http://localhost:8000/?名 1=値 1&名 2=値 2`

これを受信したサーバはリクエストの内容を解析して次のようなテキストをクライアント (Web ブラウザ) に返送する。

例. サーバからの応答

```

Host:          localhost:8000
Path:          /?名 1=値 1&名 2=値 2
Query:         {'名 1': ['値 1'], '名 2': ['値 2']}
Method:        GET
Accept:        text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
UA:            Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:127.0) Gecko/20100101 Firefox/127.0
Content-Type:  None

```

次に、次のような HTML を Web ブラウザで開き、同じサーバに対して POST メソッドで form 要素の内容を送信する例を示す。

記述例: PythonHTTPsv01.html

```

1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4      <meta charset="utf-8">
5      <title>PythonHTTPsv01</title>
6      <style>
7          textarea {
8              width:200px; height: 70px;
9              border: solid 1px #555555;
10         }

```

```

11     </style>
12 </head>
13 <body>
14     <form action="http://localhost:8000/" method="POST">
15         <input type="submit" name="FormName1" value="SUBMIT (POST)"><br>
16         <textarea name="TextData1"></textarea>
17     </form>
18 </body>
19 </html>

```

この HTML を Web ブラウザで開いた直後は図 81 の (a) のような表示となる。

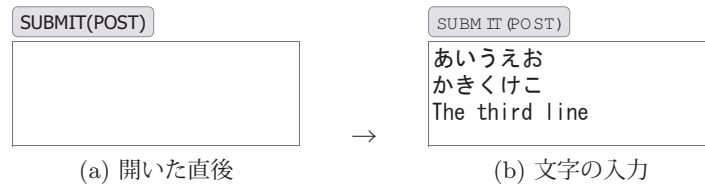


図 81: PythonHTTPsv01.html を Web ブラウザで表示した様子

(b) のように textarea 要素に文字を入力して **SUBMIT(POST)** ボタンをクリックすると、form 要素の内容がサーバに送信され、それらを解析した結果のテキストが Web ブラウザに表示される。(次の例)

例. サーバからの応答

```

Host:          localhost:8000
Path:          /
Query:         {}
Method:        POST
Accept:        text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
UA:           Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:127.0) Gecko/20100101 Firefox/127.0
Content-Type:  application/x-www-form-urlencoded
--- POST されたデータ ---
FormName1=SUBMIT(POST)&TextData1=あいうえお
かきくけこ
The+third+line
--- 以上 ---

```

A.2.2 Flask

Flask は Armin Ronacher 氏によって開発されたコンパクトな Web アプリケーションフレームワークであり、HTTP サーバの機能をはじめ、ルーティング、テンプレートエンジン、セッション管理、フォームバリデーションなどの機能を提供する。Flask は「マイクロフレームワーク」とも表現され、IoT やエッジコンピューティングの分野においても高度な Web アプリケーションを実現するための基盤となる。また、サーバ実装のためのコーディングも極めて簡便な形式であることが Flask の特徴である。

Flask は Python の標準ライブラリではなく、使用に際しては予め pip など Python 環境にインストールしておく必要がある。

本書では、Flask で HTTP サーバを実現するための最も基本的な方法について解説する。Flask の詳細に関しては公式インターネットサイト

<https://palletsprojects.com/p/flask/>

を参照のこと。

A.2.2.1 最も簡単な HTTP サーバの実装例

クライアントからの GET リクエストに対して 'Hello World' という内容で応答する単純な HTTP サーバの例 FlaskSv01.py を示す。

記述例：FlaskSv01.py

```
1 # coding: utf-8
2 from flask import Flask      # Flaskライブラリの読み込み
3 app = Flask( __name__ )      # サーバのインスタンスの生成
4
5 @app.route('/')              # サーバURLのパス「/」に対するリクエストに応える
6 def hello():                  # 関数 hello の定義
7     return 'Hello World'
8
9 app.run()                     # サーバの起動
```

これは1つのソースファイルとして実装された HTTP サーバであり、OS のコマンドから Python 処理系と共に起動するものである。この例からわかるように、flask ライブラリが提供する Flask クラスのインスタンス app が HTTP サーバ本体であり、それに対して run メソッドを実行することでそれが HTTP サーバとして起動する。

■ Flask コンストラクタ

Flask コンストラクタは HTTP サーバを生成する。

書き方： `Flask(モジュール名)`

第1引数「モジュール名」(import_name 引数)には当該 HTTP サーバのモジュール名を与える。上の例 FlaskSv01.py は1つのソースファイルなので、これを実行するとグローバル変数 __name__ の値である '.__main__' がコンストラクタに与えられる。これは、1つのスクリプトファイルとして実行された場合の当該スクリプトのモジュール名である。

Flask による HTTP サーバは、1つのディレクトリに複数のソースファイルをまとめたパッケージ形式で実装することもあり、コンストラクタに与える import_name 引数は重要である。

■ リクエストハンドラ

クライアントからのリクエストを受けて作動するリクエストハンドラは関数として定義される。このとき、当該関数定義に

`@Flask インスタンス.route(URL のパス, methods=[メソッドの並び])`

という形式のデコレータを前置する。これによりその関数は、「URL のパス」に対するリクエストを「メソッドの並び」に該当するメソッドで受信した際のリクエストハンドラとなる。デコレータの methods の記述を省略した場合のデフォルトは「methods=['GET']」(GET リクエストに対する処理)となる。リクエストハンドラの戻り値が応答ボディ (デフォルトのメディアタイプは text/html) としてクライアントに送信される。

メディアタイプを指定して応答ボディを送信するには Response オブジェクトを用いる。これを使用するには、事前に flask ライブラリからこのクラスを読み込んでおくこと。

書き方： `Response(データ, mimetype=メディアタイプ)`

「メディアタイプ」で「データ」を送信するための `Response` オブジェクトを返す。

■ サーバの起動

Flask インスタンスに対して `run` メソッドを実行することでそれが HTTP サーバとして起動する。

書き方： `Flask インスタンス.run(ホスト名, ポート番号)`

「ホスト名」(host 引数) にはリクエストを受け付ける NIC のホスト名 (あるいは IP アドレス) を文字列で、「ポート番号」(port 引数) には整数値を与える。「ホスト名」のデフォルトは '127.0.0.1', 「ポート番号」のデフォルトは 5000 である。

先のサーバ `FlaskSv01.py` を OS のコマンド

`python FlaskSv01.py` (B シェル, Anaconda プロンプトなどの場合)

あるいは

`py FlaskSv01.py` (Windows 用 PSF 版 Python の場合)

として実行することで当該計算機で HTTP サーバが起動する。その後、同じ計算機で Web ブラウザを起動して

`http://localhost:5000/`

の URL をリクエストすると「Hello World」が表示される。

各種設定を省略しない形で先のサーバ `FlaskSv01.py` を書き換え、更に応答ボディのメディアタイプを 'text/plain' として送信する `FlaskSv02.py` を示す。

記述例： `FlaskSv02.py`

```
1 # coding: utf-8
2 from flask import Flask, Response # Flaskライブラリの読み込み
3 app = Flask( __name__ )           # サーバのインスタンスの生成
4
5 @app.route('/', methods=['GET'])   # URLのパス「/」に対するリクエストに応える
6 def hello():                       # 関数 hello の定義
7     return Response('Hello World',mimetype='text/plain; charset=utf-8')
8
9 app.run('127.0.0.1',5000)          # サーバの起動
```

A.2.2.2 リクエストの情報や POST されたデータの取得

Flask ライブラリが提供する `request` オブジェクトは、リクエスト受信後の情報 (表 70) を保持する。

表 70: request オブジェクトの属性 (一部)

属性	解 説
method	受信したリクエストのメソッド (GET, POST など) の文字列
mimetype	リクエストボディのメディアタイプ
args	URL のクエリ文字列を辞書に変換したもの
form	フォームデータを辞書に変換したもの
headers	リクエストのヘッダ情報 (EnvironHeaders クラス)
user_agent	クライアント情報 (UserAgent クラス)
cookies	クライアントのクッキー情報の辞書
files	アップロードされたファイルを保持する辞書
stream	POST されたデータを読み込むためのストリーム

クライアントからのリクエストを受けて表 70 のいくつかの属性をテキストデータにして返送するサーバの例 `FlaskSv03.py` を示す。

記述例： `FlaskSv03.py`

```
1 # coding: utf-8
2 from flask import Flask, Response, request # Flaskライブラリの読み込み
3 app = Flask( __name__ )                   # サーバのインスタンスの生成
```

```

4
5 def getReq():          # リクエスト情報の解析
6     rqDat = '-----\n'
7     rqDat += 'Method:\t\t' + request.method + '\n'
8     rqDat += 'ContentType:\t' + request.mimetype + '\n'
9     rqDat += 'QueryStr:\t' + str(dict(request.args)) + '\n'
10    rqDat += 'form:\t\t' + str(dict(request.form)) + '\n'
11    rqDat += 'UA:\t\t' + str(request.user_agent) + '\n'
12    rqDat += 'Cookies:\t' + str(dict(request.cookies)) + '\n'
13    rqDat += 'Headers:\t' + str(type(request.headers)) + '\n'
14    rqDat += 'Files:\t\t' + str(dict(request.files)) + '\n'
15    return rqDat
16
17 # GETメソッドのリクエストの処理
18 @app.route('/get-method', methods=['GET'])
19 def doGet():
20     return Response(f'GETメソッドで受信\n{getReq()}',
21                     mimetype='text/plain; charset=utf-8')
22
23 # POSTメソッドのリクエストの処理
24 @app.route('/post-method', methods=['POST'])
25 def doPost():
26     return Response(f'POSTメソッドで受信\n{getReq()}',
27                     mimetype='text/plain; charset=utf-8')
28
29 app.run('127.0.0.1', 5000)          # サーバの起動

```

このサーバをローカルの計算機環境で起動し、同じ計算機の Web ブラウザから

`http://localhost:5000/get-method?名1=値1&名2=値2&名1=値3`

の URL をリクエスト (GET) すると次のようなテキストが表示される。

例. サーバ FlaskSv03.py からの応答

GET メソッドで受信

```

-----
Method:      GET
ContentType:
QueryStr:    {'名1': '値1', '名2': '値2'}
form:        {}
UA:          Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:127.0) Gecko/20100101 Firefox/127.0
Cookies:     ... (Cookie 情報) ...
Headers:     <class 'werkzeug.datastructures.headers.EnvironHeaders'>
Files:       {}

```

先のサーバ FlaskSv03.py は POST リクエストを受け付けることもできる。次のような HTML を Web ブラウザで開いてそれを確認する。

注意)

クエリ文字列として同一のキーを複数記述した場合、はじめに記述したものが有効になることに注意。

記述例: FlaskSv03-04.html

```

1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4     <meta charset="utf-8">
5     <title>FlaskSv03-04</title>
6     <style>
7         textarea {
8             width:200px; height: 70px;
9             border: solid 1px #555555;
10        }
11    </style>
12 </head>
13 <body>
14     <form action="http://localhost:5000/post-method" method="POST">
15         <input type="submit" name="FormName1" value="SUBMIT (POST)"><br>
16         <textarea name="TextData1"></textarea>
17     </form>
18 </body>

```


これを Web ブラウザで開いた直後は図 82 の (a) のように表示される。

SUBMIT(POST)

(a) 開いた直後

→

SUBMIT(POST)

一行目
2行目
さんぎょうめ

(b) 文字の入力

図 82: FlaskSv03-04.html を Web ブラウザで表示した様子

この状態で (b) のように textarea 要素に文字を入力して **SUBMIT(POST)** をクリックすると、次のようなテキストが表示される。

例. サーバ FlaskSv03.py からの応答

POST メソッドで受信

```
-----
Method:      POST
ContentType: application/x-www-form-urlencoded
QueryStr:    {}
form:        {'FormName1': 'SUBMIT(POST)', 'TextData1': '一行目¥r¥n2 行目¥r¥nさんぎょうめ'}
UA:          Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:127.0) Gecko/20100101 Firefox/127.0
Cookies:     ... (Cookie 情報) ...
Headers:     <class 'werkzeug.datastructures.headers EnvironHeaders'>
Files:       {}
```

先の例では POST リクエストで送信された form データを request.form から取得している。request オブジェクトからデータを取得するメソッドは表 71 に挙げるようなものもあり、バイナリデータ (バイト列)、テキスト、JSON といった形式でリクエストボディを取得することができる。

表 71: request オブジェクトに対するメソッド (一部)

メソッド	解 説
get.data(as_text=True)	POST されたテキストデータの取得
get.data()	POST されたバイナリデータ (バイト列) の取得
get.json()	POST された JSON データの取得 (辞書形式)

次に示すサンプル FlaskSv04.py は get_data メソッドでリクエストボディを取得するものである。

記述例: FlaskSv04.py

```
1  # coding: utf-8
2  from flask import Flask, Response, request  # Flaskライブラリの読み込み
3  from urllib.parse import unquote           # URLエンコーディングの復元機能
4  app = Flask(__name__)                      # サーバのインスタンスの生成
5
6  def getReq():                               # リクエスト情報の解析
7      rqDat = '-----\n'
8      rqDat += 'Method:\t\t' + request.method + '\n'
9      rqDat += 'ContentType:\t' + request.mimetype + '\n'
10     rqDat += 'UA:\t\t' + str(request.user_agent) + '\n'
11     return rqDat
12
13 # GETメソッドのリクエストの処理
14 @app.route('/get-method', methods=['GET'])
15 def doGet():
16     return Response(f'GETメソッドで受信\n{getReq()}',
17                     mimetype='text/plain; charset=utf-8')
18
19 # POSTメソッドのリクエストの処理
20 @app.route('/post-method', methods=['POST'])
21 def doPost():
22     r = f'POSTメソッドで受信\n{getReq()}'
23     r += '-----\n'
```

```

24     r += unquote(request.get_data(as_text=True))
25     return Response(r,mimetype='text/plain; charset=utf-8')
26
27 app.run('127.0.0.1',5000)           # サーバの起動

```

このサーバスクリプトをローカルの計算機環境で起動して、先と同じ FlaskSv03-04.html を同じ計算機の Web ブラウザで開いて form データを POST すると次のような表示となる。

例. サーバ FlaskSv03.py からの応答

POST メソッドで受信

```

-----
Method:      POST
ContentType: application/x-www-form-urlencoded
UA:         Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:127.0) Gecko/20100101 Firefox/127.0
-----

```

FormName1=SUBMIT(POST)&TextData1=一行目

2 行目

さんぎょうめ

get_data メソッドを用いるとデータ形式に依らずリクエストボディを読み込むことができる。

参考)

Flask は WSGI (Web Server Gateway Interface) の仕組みに基づいている。WSGI による Web サービスは Apache HTTP Server でも実装可能で、これに関しては後の「A.3 Apache HTTP Server」(p.281) の「A.3.5 WSGI 関連」(p.288) で解説する。

A.3 Apache HTTP Server

Apache HTTP Server は **Apache ソフトウェア財団**¹³³ が管理し公開するオープンソースの Web サーバ・ソフトウェアである。Apache HTTP Server は歴史も長く、各種 OS 用のものがリリースされており、Web サーバとして広く普及している。本書では Windows 用の Apache HTTP Server について導入と設定の基本的な方法を解説する。

A.3.1 ソフトウェアの入手

Apache HTTP Server は公式インターネットサイト

<https://httpd.apache.org/>

から入手することができる。Windows 用のものは ZIP 書庫の形式（右図）で配布されている。これを展開して Apache24 というディレクトリ（フォルダ）を取り出す。本書ではこれを

C:\¥Apache24

に配置するものとして解説する。



httpd-2.4.59-240404-win64-VS17.zip

Windows 用 Apache HTTP Server (ZIP)

A.3.2 ディレクトリの構成

ディレクトリ Apache24 内には更にいくつかのサブディレクトリがある。その構成を図 72 に示す。

表 72: Apache HTTP Server のディレクトリ群

ディレクトリ	解 説
bin	サーバーの機能を支える実行可能なファイル（プログラム）群。
cgi-bin	CGI（Common Gateway Interface）スクリプト。これらのスクリプトはサーバーによって実行され、動的なコンテンツを生成する。
conf	サーバーの動作に関する設定ファイル群。
error	エラー発生時の表示に関するファイルやテンプレートなど。
htdocs	Web コンテンツのドキュメントルートディレクトリ。HTML ファイル、画像をはじめとする、Web ブラウザからアクセス可能なリソース群。
icons	サーバーが表示する Web ページで使用するアイコン画像。
include	C 言語で開発するモジュールや追加コンポーネントをコンパイルする際に使用するヘッダーファイル（*.h）群。
lib	bin ディレクトリの実行可能プログラムやモジュールに必要なライブラリファイル群。
logs	サーバーの動作報告が記録されるログファイル。アクセスログやエラーログなど。
manual	Apache HTTP Server に関するドキュメントやマニュアルなど。
modules	サーバの機能を拡張するロード可能モジュール。

図 72 のディレクトリの内、特に bin, conf, htdocs, cgi-bin について触れる。

A.3.3 インストール作業と設定

Apache HTTP Server を使用するには、設定ファイル Apache24/conf/httpd.conf を正しく記述する必要がある。このファイル内に記述する重要なディレクティブ¹³⁴について解説する。

■ Web サーバの FQDN とポート番号の設定

設定ファイル Apache24/conf/httpd.conf 内の ServerName ディレクティブを設定する。

書き方：ServerName サーバの FQDN:80

¹³³<https://www.apache.org/>

¹³⁴処理系に対する設定、指示をディレクティブと呼ぶ。

Web サーバをローカルの計算機環境で使用する場合は

```
ServerName localhost:80
```

と記述する。当該ソフトウェアのインストール直後は、設定ファイル内に次のような記述がある。

```
初期状態：#ServerName www.example.com:80
```

これを編集するか、設定の記述を新規に追加する。

■ OS のサービスへの登録と起動

Windows 環境では各種のアプリケーションやサービスがバックグラウンドで動作しており、これは **Windows ツールのサービス**（右図）によって確認できる。これを起動すると当該環境で実行中のソフトウェアの一覧が表示（図 83）される。



サービス

Windows ツールの「サービス」

Apache HTTP Server をインストールするには Apache24/bin のディレクトリでコマンド「`httpd -k install`」を発行する。

```
例. C:¥Apache24¥bin> httpd -k install
```

コマンドの処理が正常に終了すると、サービスとして「Apache2.4」の表示が追加される。

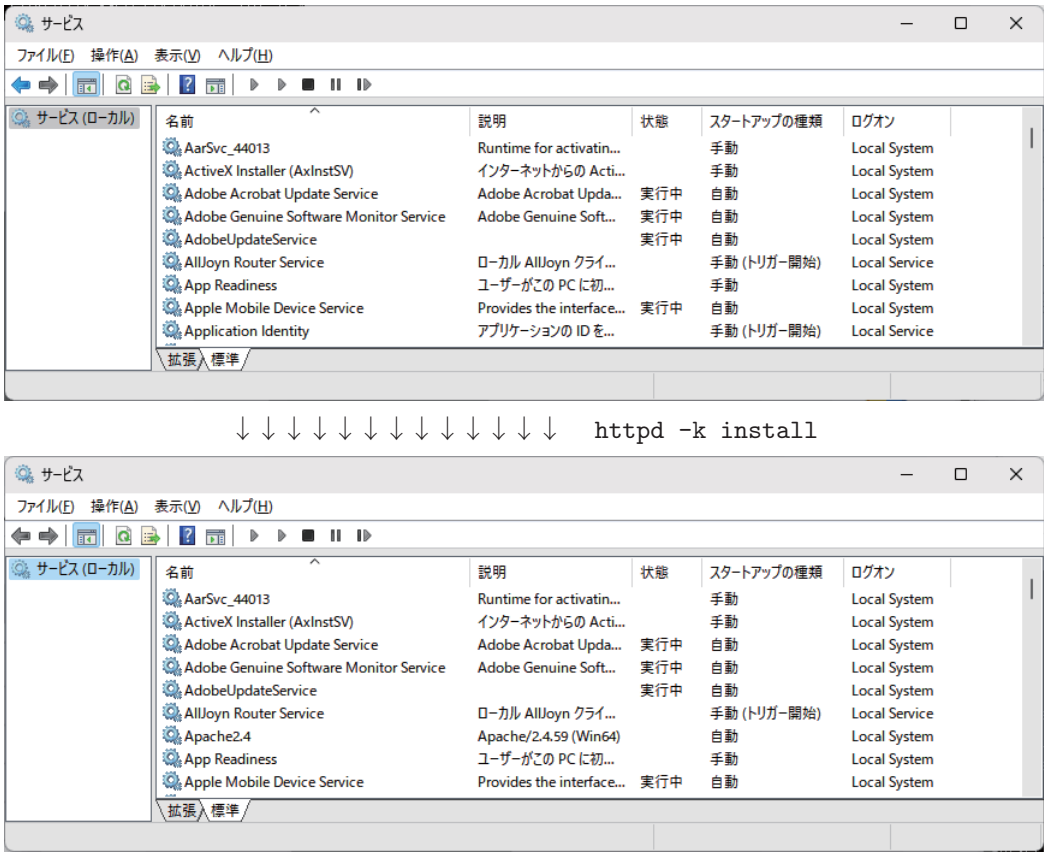


図 83: サービスへの Apache の登録の前後

上の図はインストール前であり、インストールが完了すると下の図のようになる

■ HTML コンテンツの用意

Apache24/htdocs ディレクトリの下に HTML コンテンツ（各種リソース含む）を配置するとそれが当該 Web サーバによって配信される。（次の例）

記述例：Apache24/htdocs/MyHomePage.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>MyHomePage</title>
6 </head>
7 <body>
8   <h1>MyHomePage</h1>
9   <p>Apache HTTP Serverによる配信</p>
10 </body>
11 </html>
```

ローカル環境（localhost）の Web サーバの場合、左のコンテンツが存在する状態で

`http://localhost/MyHomePage.html`

の URL にアクセスすると、Web ブラウザに次のように表示される。

MyHomePage

Apache HTTP Serverによる配信

参考) 当該ソフトウェアをアンインストールするにはコマンド「`httpd -k uninstall`」を発行する。

例. `C:\¥Apache24¥bin> httpd -k uninstall`

この処理により、Windows のサービスから「Apache2.4」が除去される。

A.3.4 CGI 関連

CGI (Common Gateway Interface) は、クライアントシステムからの要求に応じて Web サーバ上でプログラム (CGI プログラム) を起動して実行する仕組みであり、Web アプリケーションを実現するための標準的な方法の 1 つとして広く応用されている。

CGI プログラムは Apache24/cgi-bin ディレクトリに配置し、それを示す URL にクライアント (Web ブラウザ) がアクセスすることで起動する。起動された CGI プログラムはその実行結果をクライアントに返送する。CGI プログラムは各リクエストに対して起動されるものであり、同一の CGI プログラムであっても、複数の異なるリクエストがあると、それぞれ別のプログラムとして起動される。

A.3.4.1 CGI を有効にする設定

Apache HTTP Server で CGI を有効にするには設定ファイル Apache24/conf/httpd.conf 内に必要なディレクティブを記述する。次に基本的なディレクティブの設定を挙げる。設定ファイル内の該当する箇所を確認して編集する。該当箇所がなければ記述を追加する。

■ CGI の機能を支えるモジュールの指定

```
LoadModule cgi_module modules/mod_cgi.so
```

■ CGI プログラムを配置するディレクトリの指定

```
ScriptAlias /cgi-bin/ "${SRVROOT}/cgi-bin/"
```

■ CGI として実行を許可するためのファイル名の拡張子の設定

```
AddHandler cgi-script .cgi .pl .py .js
```

A.3.4.2 CGI の実装

CGI プログラムは、クライアント (Web ブラウザ) からのアクセスに応じて起動され、データを生成して返信するものである。CGI プログラムは様々なプログラミング言語で実装することができる。

■ HTTP リクエストの解析

CGI プログラムが起動直後に行うべき処理として、クライアントからの **HTTP リクエストの種類** (表 73) の判別がある。受信した HTTP リクエストの種類は環境変数 `REQUEST_METHOD` から参照できる。

■ 受信したデータの取得

クライアントは Web サーバの CGI に対してデータを送信することができる。URL の直後に「?」を置き、その直後に

”属性 1=値 1&属性 2=値 2&…”

の形式の文字列を与えて GET リクエストで CGI を起動すると、CGI 側ではそれを環境変数 `QUERY_STRING` から

表 73: クライアントからの HTTP リクエストメソッドの種類

リクエスト	解説
GET	サーバのリソースの要求.
POST	サーバへの情報を送信. フォームの送信やファイルのアップロードなどに使用される.
PUT	サーバ上の特定のリソースを更新または作成するためのリクエスト.
DELETE	サーバ上の特定のリソースを削除するためのリクエスト
HEAD	サーバからヘッダー情報のみを取得するためのリクエスト

参照できる. CGI から参照できる環境変数には表 74 に示すようなものがある.

表 74: CGI 側で利用できる環境変数 (一部)

変数	解説
CONTENT_LENGTH	クライアントから送信されたデータの長さ (単位: バイト)
CONTENT_TYPE	クライアントから送信されたデータのメディアタイプ
QUERY_STRING	クエリ文字列
REQUEST_METHOD	クライアントからのリクエストメソッド (表 73)
REQUEST_SCHEME	リクエストが使用したスキーム (http, https など)
SERVER_ADDR	サーバの IP アドレス
SERVER_PORT	サーバがリクエストを受け取るポート
REMOTE_ADDR	クライアントの IP アドレス
REMOTE_PORT	クライアントのポート番号
HTTP_USER_AGENT	クライアントのユーザーエージェント

クライアントから POST リクエストで送信されたデータは CGI プログラムの標準入力から取得する.

■ 処理の実行と応答メッセージの返信

受信した HTTP リクエストに応じて具体的な処理を実行し, 処理結果の応答メッセージをクライアントに返信する. 応答メッセージの冒頭では

Content-Type: **メディアタイプ**

を送信し, その直後に改行を 2 回送信する. その後は指定したメディアタイプの形式のデータを送信する. 応答メッセージの送出は CGI プログラムから標準出力に出力することで実行する.

A.3.4.3 CGI のサンプル (1): 単純な例

各種のスクリプト言語処理系は利用方法が簡易であり, テキストファイルとして作成したスクリプトプログラムが手軽な方法で実行できる. ここでは, Web サーバにインストールされている Node.js, Python を使用して, それらのスクリプトの形式で CGI プログラムを実装する例を示す.

UNIX 系 OS (Linux, macOS など) のコマンドシェルでは, テキストファイルとして作成したスクリプト言語のプログラムが簡単に実行できる. 具体的には, スクリプト言語で記述したプログラムの先頭行に**シバン** (shebang) と呼ばれる記述 (後述) をして, 当該スクリプトファイルの**アクセス権** (パーミッション) として実行権限を付与すると, そのファイルがそのままコマンドプログラムとしてシェル上で実行できる. Apache の CGI プログラムもこれと同じ方法で実装できる.

■ 実行可能なスクリプトファイルの作成

行頭に**シバン**を持つスクリプトプログラムの書き方は次の通り.

《単体で実行可能なスクリプトファイルの記述》

```
#!/言語処理系（インタプリタ）のパス
（具体的なプログラムの記述）
：
```

←この行がシバン

次に示す Apache24/cgi-bin/nodeCGI01.js は Apache の CGI として作成された Node.js のスクリプトプログラムである。

記述例：Apache24/cgi-bin/nodeCGI01.js

```
1  #!"C:\Program Files\nodejs\node.exe"
2  // NodeによるCGIプログラム
3  process.stdout.write('Content-Type: text/html\n\n');    // コンテンツタイプの送信
4  // ここからHTMLコンテンツの送信
5  process.stdout.write('<!DOCTYPE html>
6  <html lang="ja">
7  <head>
8    <meta charset="utf-8">
9    <title>ローカルのWebページ</title>
10 </head>
11 <body>
12   <h2>ローカルPCのWebページ</h2>
13   <p>CGIによる生成</p>
14 </body>
15 </html>');

```

このプログラムの先頭行にある

```
#!/C:¥Program Files¥nodejs¥node.exe"
```

がシバンであり、C:¥Program Files¥nodejs¥node.exe のパスが示す言語処理系（インタプリタ）を使用して2行目以降のプログラム（JavaScript）を実行する。process.stdout.write は標準出力に出力するためのメソッドであり、その出力がクライアントに送信される。

localhost に Web サーバがある場合、Web ブラウザから次の URL

<http://localhost/cgi-bin/nodeCGI01.js>

にアクセスすると、上記 CGI プログラムが起動し、その出力が Web ブラウザに表示される。（右図）

ローカルPCのWebページ

CGIによる生成

Web ブラウザの表示

次に示す Apache24/cgi-bin/pythonCGI01.py は Apache の CGI として作成された Python のスクリプトプログラムである。

記述例：pythonCGI01.py

```
1  #!"C:\Program Files\Python312\python.exe"
2  # PythonによるCGIプログラム
3  print('Content-Type: text/html\n')    # コンテンツタイプの送信
4  # ここからHTMLコンテンツの送信
5  print(''<!DOCTYPE html>
6  <html lang="ja">
7  <head>
8    <meta charset="sjis">
9    <title>ローカルのWebページ</title>
10 </head>
11 <body>
12   <h2>ローカルPCのWebページ</h2>
13   <p>CGIによる生成</p>
14 </body>
15 </html>''')
```

このプログラムの先頭行にある

```
#!/C:¥Program Files¥Python312¥python.exe"
```

がシバンであり、C:¥Program Files¥Python312¥python.exe のパスが示す言語処理系（インタプリタ）を使用し

て 2 行目以降のプログラム (Python) を実行する。

localhost に Web サーバがある場合、Web ブラウザから次の URL

`http://localhost/cgi-bin/pythonCGI01.py`

にアクセスすると、上記 CGI プログラムが起動し、その出力が Web ブラウザに表示される。この CGI プログラムは先の Apache24/cgi-bin/nodeCGI01.js と同じ内容を送信する。

A.3.4.4 CGI のサンプル (2) : 環境変数と POST された内容の表示

次に示す nodeCGI02.js は Node.js で実装したもので、クライアントからのリクエストに対する応答として、環境変数の内容と POST されたデータをテキスト形式 (text/plain) で返送するものである。

記述例 : nodeCGI02.js

```
1  #!"C:\Program Files\nodejs\node.exe"
2  // POSTされたデータ (標準入力) の読み取り
3  let d = "";
4  process.stdin.on( 'data', chunk => { d += chunk.toString(); });
5
6  // 読み取ったデータの送信
7  process.stdin.on( 'end', () => {
8    // コンテンツタイプの送信
9    process.stdout.write('Content-Type: text/plain; charset=utf-8\n\n');
10   const ks = Object.keys(process.env).sort();      // 環境変数のキーの取得
11   let k="";
12   for ( k of ks ) {                                // 環境変数の内容の送信
13     process.stdout.write( k ); process.stdout.write( "\t");
14     process.stdout.write( decodeURIComponent(String(process.env[k])) );
15     process.stdout.write( "\n" );
16   }
17   // POSTされたコンテンツの送信
18   process.stdout.write( "--- posted contents ---\n" );
19   d = decodeURIComponent(d);
20   process.stdout.write( d );
21 });
```

Node.js では、環境変数はグローバルオブジェクト process の env プロパティが保持している。また、標準入力からデータを受け取るには、process.stdin オブジェクトに起こる data イベントを受けて内容を分割して取得する。

localhost に Web サーバがある場合、Web ブラウザで次の HTML コンテンツ nodeCGI02.html を表示して上記 CGI プログラムを起動する。

記述例 : nodeCGI02.html

```
1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4    <meta charset="utf-8">
5    <title>nodeCGI02</title>
6  </head>
7  <body>
8    <form action="http://localhost/cgi-bin/nodeCGI02.js" method="POST">
9      <input type="submit" name="FormName1" value="SUBMIT(POST)"><br>
10     <textarea name="TextData1"></textarea>
11   </form>
12   <form action="http://localhost/cgi-bin/nodeCGI02.js" method="GET">
13     <input type="submit" name="FormName2" value="SUBMIT(GET)"><br>
14     <textarea name="TextData2"></textarea>
15   </form>
16 </body>
17 </html>
```

この HTML コンテンツでは、form 要素の action 属性で起動する CGI プログラムの URL を指定している。また method 属性でリクエストメソッド (GET, POST) を指定している。

これを Web ブラウザで表示すると、右図のような form 要素が表示される。上部の textarea に文字を入力して **SUBMIT(POST)** ボタンをクリックすると POST メソッドでその内容が Web サーバに送信される。下部の textarea に文字を入力して **SUBMIT(GET)** ボタンをクリックすると GET メソッドでその内容が Web サーバに送信される。

form 要素の送信後はサーバからテキストデータ (text/plain) の形式で

環境変数 値

の一覧が返送され、それが Web ブラウザの表示される。また、POST メソッドで送信した内容はサーバからの返信内容の下部に

```
--- posted contents ---
(POST された内容)
:
```

という形式で表示される。

同様の処理を行う CGI プログラムを Python 言語で実装したものを次の pythonCGI02.py に、それを起動する HTML コンテンツを pythonCGI02.html に示す。

記述例：pythonCGI02.py

```
1  #!"C:\Program Files\Python312\python.exe"
2  import sys, os
3  from urllib.parse import unquote
4
5  # コンテンツタイプの送信
6  print('Content-Type: text/plain; charset=shift-jis\n')
7
8  # 環境変数の出力
9  for k in sorted( list(os.environ.keys()) ):
10     v = unquote(os.environ[k])
11     print( k, '\t', v, sep=' ' )
12
13 # POSTされたデータの出力
14 print( '--- posted contents ---' )
15 v = unquote(sys.stdin.read())
16 print( v )
```

記述例：pythonCGI02.html

```
1  <!DOCTYPE html>
2  <html lang="ja">
3  <head>
4     <meta charset="utf-8">
5     <title>pythonCGI02</title>
6  </head>
7  <body>
8     <form action="http://localhost/cgi-bin/pythonCGI02.py" method="POST">
9         <input type="submit" name="FormName1" value="SUBMIT(POST)"><br>
10        <textarea name="TextData1"></textarea>
11    </form>
12    <form action="http://localhost/cgi-bin/pythonCGI02.py" method="GET">
13        <input type="submit" name="FormName2" value="SUBMIT(GET)"><br>
14        <textarea name="TextData2"></textarea>
15    </form>
16 </body>
17 </html>
```

A.3.5 WSGI 関連

WSGI (Web Server Gateway Interface) は、クライアントシステムからの要求に応じて作動する仕組みであり、サーバ側での情報処理を実現したり、動的にコンテンツを生成してクライアントに返送することができる。特に WSGI は Python 言語で記述されたプログラム (WSGI プログラム) を Web サーバが起動するための仕組みとして標準化¹³⁵されており、各種の Web サーバが WSGI に対応している。

WSGI プログラムにはそれを示す URL が設定され、1 つの WSGI プログラムが複数の異なるリクエストに対応して処理を実行する。これは、1 つの WSGI プログラムが複数のスレッドを生成して各リクエストに応答する形態であり、前述の CGI プログラムのような各リクエスト毎にプログラム起動する形態と比較して効率の高い応答を実現する。

A.3.5.1 WSGI を有効にする設定

Web サーバで WSGI を実現するには、サーバの計算機環境で Python 言語が使用できることが前提となるので、PSF (Python Software Foundation)¹³⁶ の公式インターネットサイトから Python 言語処理系を入手して、サーバ環境に導入しておくこと。

Apache HTTP Server で WSGI を有効にするには設定ファイル `Apache24/conf/httpd.conf` 内に必要なディレクティブを記述する。次に基本的なディレクティブの設定を挙げる。設定ファイル内の該当する箇所を確認して編集する。該当箇所がなければ記述を追加する。

■ Python 言語処理系の指定

`LoadFile` Python 言語処理系 (インタプリタ) のパス

例えば、Windows 環境で HTTP サーバを使用する場合は

```
LoadFile "C:/Program Files/Python312/python312.dll"
```

などと記述する。この例は、Python 処理系が DLL の形式で

```
C:%Program Files%Python312%python312.dll
```

のパスに存在する場合の記述である。

■ WSGI の機能を支えるモジュールの指定

`LoadModule wsgi_module` モジュールのパス

例えば、Windows 環境で HTTP サーバを使用する場合は

```
LoadModule wsgi_module
```

```
"C:/Program Files/Python312/Lib/site-packages/mod_wsgi/server/mod_wsgi.cp312-win_amd64.pyd"
```

などと記述 (1 行で) する。この例は、モジュールが

```
C:%Program Files%Python312%Lib%site-packages%mod_wsgi%server%mod_wsgi.cp312-win_amd64.pyd
```

のパスに存在する場合の記述である。

■ Python 言語処理系のホームディレクトリ

`WSGI PythonHome` Python 言語処理系のホームディレクトリ

例えば、Windows 環境で HTTP サーバを使用する場合は

```
WSGI PythonHome "C:/Program Files/Python312"
```

などと記述する。この例は、Python 処理系のホームディレクトリが

```
C:%Program Files%Python312
```

のパスに存在する場合の記述である。

■ URL に対応する WSGI プログラムの指定

`WSGIScriptAlias` URL のパス 対応する WSGI プログラムのパス

例えば、Windows 環境で HTTP サーバを使用する場合は

¹³⁵特定の団体が規定する規格ではなく、デファクトスタンダードである。

¹³⁶Python 言語処理系を維持、管理し公開している団体。(<https://www.python.org/>)

```
WSGIScriptAlias /wsgiApp01 "C:¥Apache24¥wsgi¥wsgiApp01.wsgi"
```

などと記述する。この例は、URL のパス /wsgiApp01 に対するリクエストに対して

```
C:¥Apache24¥wsgi¥wsgiApp01.wsgi"
```

の WSGI プログラムを起動する設定である。例えば、この設定で ローカル環境のサーバにクライアント（Web ブラウザ）から http スキームで

```
http://localhost/wsgiApp01
```

にリクエストがあると、

```
C:¥Apache24¥wsgi¥wsgiApp01.wsgi"
```

のプログラムが起動する。

■ WSGI スクリプトを実行可能にする設定

WSGI スクリプトとそれを配置するディレクトリに対する設定を Directory, Files ディレクティブとして記述する。次に示す例は、ディレクトリ C:¥Apache24¥wsgi 内に WSGI スクリプト wsgiApp01.wsgi を配置する場合の記述である。

記述例：httpd.conf 内の Directory ディレクティブ

```
1 <Directory "C:\Apache24\wsgi">
2     Require all granted
3     <Files wsgiApp01.wsgi>
4         SetHandler wsgi-script
5     </Files>
6 </Directory>
```

この記述により、wsgi を収めるディレクトリへのアクセスをクライアントに許可し、wsgiApp01.wsgi を wsgi スクリプトとして実行することが可能となる。（Files ディレクティブの記述は省略しても wsgi スクリプトの実行が可能なこともある）

A.3.5.2 WSGI スクリプトの実装

wsgi スクリプトは Python 言語のプログラム（Python スクリプト）として実装する。具体的には、スクリプトのグローバルスコープで記述された application 関数がクライアントからのリクエストに応じて起動する。ここでは、Python 言語に関する知識¹³⁷ を前提として wsgi スクリプトについて解説する。

application 関数（Python 言語）の第 1 引数にはクライアントから受診したデータを含んだ環境変数の辞書オブジェクト（dict オブジェクト）が与えられる。第 2 引数にはクライアントに対する応答を開始する処理のための関数が与えられる。クライアントに対する具体的な応答内容（レスポンスボディ）はバイト列型（bytes）のデータの並び（イテラブルオブジェクト）として application 関数の戻り値とする。

書き方： `def application(環境変数辞書, 応答開始関数):`
 `応答開始関数 (ステータス, ヘッダー情報)`
 `:`
 `(応答内容の生成処理)`
 `:`
 `return 応答内容のイテラブルオブジェクト`

「応答内容のイテラブルオブジェクト」は、バイト列型（bytes）オブジェクトのリストやジェネレータといった形式のデータ構造¹³⁷ である。すなわち、複数のバイト列オブジェクトを次々とクライアントに送信する。

■ 環境変数辞書

application 関数の第 1 引数には、クライアントからリクエストを受けた際の情報が辞書オブジェクトの形式（環境変数辞書）で与えられる。この辞書が持つキーの内、特に重要なものを表 75 に挙げる。

¹³⁷ 拙書「Python3 入門 -Kivy による GUI アプリケーション開発, サウンド入出力, ウェブスクレイピング-」でも解説しています。

表 75: 環境変数辞書の重要なキー（一部）

キー	解 説（対応する値）
REQUEST_METHOD	受診した HTTP リクエストのメソッド（GET, POST など）を表す文字列
QUERY_STRING	URL のクエリストリングを表す文字列
SERVER_NAME	サーバーの名前を表す文字列
SERVER_PORT	サーバーのポートを表す文字列
CONTENT_TYPE	リクエストボディのメディアタイプを表す文字列（PUT, POST の場合）
CONTENT_LENGTH	リクエストボディの長さ（バイト単位）を表す文字列（PUT, POST の場合）
wsgi.errors	エラーメッセージ用ストリーム（デフォルトでは error.log）
wsgi.input	POST, PUT で受信したデータを受け取るための入力ストリーム

参考）基本的なものは p.284 の表 74 と共通である。

■ 応答開始関数

application 関数の第 2 引数には、クライアントに対して応答する際の開始処理を行う関数が与えられる。具体的には、この関数を用いて、応答内容を return で戻り値として送信する前に、ステータスやヘッダー情報などをクライアントに送信する。

応答開始関数を実行する際、第 1 引数には応答の **HTTP ステータスコード**（表 76）を、第 2 引数にはレスポンスボディのヘッダー情報（**HTTP ヘッダー**）（表 77）のリストを与える。

表 76: HTTP ステータスコード（一部）

ステータス	解 説
'200 OK'	リクエストが正常に処理され、レスポンスボディに結果が含まれていることを意味する。
'201 Created'	リソースが正常に作成され、その参照がレスポンスボディに含まれていることを意味する。
'204 No Content'	リクエストが正常に処理されたが、レスポンスボディに返すべき内容がないことを意味する。
'400 Bad Request'	クライアントからのリクエストが不適切な形式であることを意味する。
'403 Forbidden'	クライアントがリソースにアクセスする権限を持っていないことを意味する。
'404 Not Found'	クライアントが要求したリソースが存在しないことを意味する。
'500 Internal Server Error'	サーバー側でエラーが発生したことを意味する。

参考）Python の標準ライブラリ http が提供する HTTPStatu オブジェクトがこれらの情報を提供している。

表 77: HTTP ヘッダー（一部）

ヘッダー	解 説
'Content-Type'	レスポンスボディのメディアタイプ（MIME タイプ）
'Content-Length'	レスポンスボディの長さ（バイト単位）を意味する文字列
'Location'	リダイレクト先の URL の文字列

レスポンスボディのヘッダー情報は表 TblHTTPHeader のヘッダーを含んだ次のようなリストとして記述する。

書き方： [(ヘッダー 1, 値 1), (ヘッダー 2, 値 2), …]

ヘッダー情報としては最低限 'Content-Type' があれば良い。

A.3.5.3 WSGI のサンプル (1)：単純な例

次に示す wsgiApp01.wsgi は、クライアントからのリクエストに対する応答として単純に HTML コンテンツを返送するものである。

記述例：wsgiApp01.wsgi

```
1 # coding: utf-8
2 #--- 応答用のコンテンツ（ここから）---
3 htm = '''
4 <html lang="ja">
5 <head>
6   <meta charset="utf-8">
7   <title>WSGIプログラムからの応答 </title>
8 </head>
9 <body>
10   <h2>Hello,World!</h2>
11   <p>WSGI Sample</p>
12 </body>
13 </html>
14 '''
15 #--- 応答用のコンテンツ（ここまで）---
16
17 # グローバルスコープの application 関数が応答用の関数となる。
18 def application(environ, start_response):
19     '''第1引数 environ に環境変数の内容が得られる
20     第2引数に応答用の設定関数が得られる'''
21     global htm
22     status = '200 OK'
23     headers = [ ( 'Content-type', 'text/html; charset=utf-8' ) ]
24     start_response(status, headers)      # 応答用の設定処理
25     r = htm.encode('utf-8')              # 応答用コンテンツをバイト列に変換して
26     return [r]                           # 送信する
```

この WSGI スクリプトがディレクトリ C:¥Apache24¥wsgi にあり、p.289 の「httpd.conf 内の Directory ディレクティブ」の設定と、p.288 の「URL に対応する WSGI プログラムの指定」の設定がなされているとする。

上記 WSGI スクリプトをローカルのサーバ環境に実装して Web ブラウザから <http://localhost/wsgiApp01> にアクセスした場合の表示を右に示す。

Hello,World!
WSGI Sample

A.3.5.4 WSGI のサンプル (2)：環境変数と POST された内容の表示

次に示す wsgiApp02.wsgi は、クライアントからのリクエストに対する応答として、環境変数の内容と POST されたデータをテキスト形式 (text/plain) で返送するものである。

記述例：wsgiApp02.wsgi

```
1 # coding: utf-8
2 from urllib.parse import unquote
3
4 def application(environ, start_response):
5     txt = ''
6     for x in sorted(list(environ.keys())): # 環境変数の内容をテキストにする処理
7         txt += (unquote(str(x)) + '\t' + unquote(str(environ[x])) +
8                 '\t' + str(type(environ[x])) + '\n')
9     bq = '--- posted contents ---\n'.encode('utf-8')
10    if environ['REQUEST_METHOD'] == 'POST': # POSTされた内容（バイト列）の取得
11        t = unquote(environ['wsgi.input'].read().decode('utf-8'))
12        bq += t.encode('utf-8')
13    status = '200 OK'
14    headers = [ ( 'Content-type', 'text/plain; charset=utf-8' ) ]
15    start_response(status, headers)      # 応答用の設定処理
16    r = txt.encode('utf-8')              # 応答用コンテンツをバイト列に変換して
17    return [r,bq]                        # 送信する
```

この WSGI スクリプトがディレクトリ C:¥Apache24¥wsgi にあり、p.289 の「httpd.conf 内の Directory ディレクティブ」の設定と、次のディレクティブが記述されているとする。

```
WSGIScriptAlias /wsgiApp02 "C:¥Apache24¥wsgi¥wsgiApp02.wsgi"
```

上記 WSGI スクリプトをローカルのサーバ環境に実装して Web ブラウザから

<http://localhost/wsgiApp02>

の URL にアクセスすると、各行が

環境変数名 値 型

の形式のテキストデータが表示される。また次のように、URL にクエリストリング¹³⁸ を付加して

`http://localhost/wsgiApp02?名 1=値 1&名 2=値 2` (?移行がクエリストリング)

としてアクセスすると、それが WSGI スクリプトの環境変数 `QUERY_STRING` に渡され、表示内容に

`QUERY_STRING 名 1=値 1&名 2=値 2 <class 'str'>`

という行として現れる。クエリストリングは GET メソッドで WSGI スクリプトに値を渡すための基本的な方法である。

上記 WSGI スクリプトは POST メソッドで送信されたデータを受け取ることもできる。ここでは次のような HTML コンテンツによって、POST メソッドでローカルの Web サーバにデータを送信する例を示す。

記述例：wsgiApp02.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>wsgiApp02</title>
6 </head>
7 <body>
8   <form action="http://localhost/wsgiApp02" method="POST">
9     <input type="submit" name="FormName1" value="SUBMIT">
10    <textarea name="TextData"></textarea>
11  </form>
12 </body>
13 </html>
```

上記 HTML コンテンツを Web ブラウザで表示してテキストエリアに文字を入力する。(右図)

文字を入力した後 ボタンをクリックすると、テキストエリア内の文字が POST メソッドで送信され WSGI スクリプトに渡される。

この後、Web ブラウザに環境変数の一覧が表示され、POST されたデータが末尾に

```
--- posted contents ---
FormName1=SUBMIT&TextData=一行目の内容
にぎょうめのないよう
```

と表示される。

<input type="button" value="SUBMIT"/>	一行目の内容 にぎょうめのないよう
---------------------------------------	----------------------

¹³⁸クエリパラメータまたは単にパラメータと呼ぶこともある。

A.4 ローカルの計算機環境で HTTP サーバを起動する簡易な方法

A.4.1 Node.js による簡易な方法

Node.js の `http-server` パッケージによって Web サーバ (HTTP サーバ) を起動することができる。このパッケージは Node.js には標準添付されておらず、コマンドウィンドウ (コマンドプロンプトウィンドウ, ターミナルウィンドウ) で `npm` コマンドによってインストールする必要がある。

```
npm install -g http-server
```

この後、`http-server` コマンドを発行すると、カレントディレクトリをコンテンツディレクトリとして HTTP サーバが起動し、ローカルの Web ブラウザから

```
http://127.0.0.1:8080/コンテンツのパス
```

```
http://localhost:8080/コンテンツのパス
```

といった URL で Web コンテンツにアクセスできる。

`http-server` を終了するには、当該コマンドウィンドウで `CTRL` + `C` とキー入力する。

A.4.2 Python による簡易な方法

Python3 の言語処理系には、標準的に Web サーバの機能が含まれている。コマンドウィンドウ (コマンドプロンプトウィンドウ, ターミナルウィンドウ) で次のようにコマンドを発行する。

```
python3 -m http.server
```

```
py -m http.server
```

 (Windows 用の PSF 版 Python の場合)

この操作¹³⁹によって、カレントディレクトリをコンテンツディレクトリとして HTTP サーバが起動し、ローカルの Web ブラウザから

```
http://127.0.0.1:8000/コンテンツのパス
```

```
http://localhost:8000/コンテンツのパス
```

といった URL で Web コンテンツにアクセスできる。

Web サーバを終了するには、当該コマンドウィンドウで `CTRL` + `C` とキー入力する。

¹³⁹ 計算機環境によっては `python` コマンドの場合もある。

A.5 WebSocket

HTTP による通信では、クライアントから送信されたリクエストに対してサーバが応答を返信する形を取る。この形の通信では、サーバが任意のタイミングでクライアントに対してデータをプッシュすることが難しい。これに対して WebSocket プロトコルによる通信では、サーバとクライアント間の双方向の通信が可能である。

WebSocket は HTTP とは異なるプロトコルとして標準化¹⁴⁰ されているが、通信の開始には HTTP プロトコルを使用する。そして、通信が確立されるとその後の通信プロトコルは WebSocket にアップグレード¹⁴¹ される。この場合、通信の開始に行った HTTP 通信と同じポートがそのまま WebSocket のポートとなる。

本書では WebSocket サーバを構築する最も基本的な方法を、Node.js と Python の場合に分けて解説する。

A.5.1 Node.js による WebSocket

Node.js で WebSocket サーバを実現するためのライブラリとして `ws` が利用できる。`ws` は Node.js に標準添付のものではなく、プロジェクトのディレクトリに対して `npm` を用いて導入する必要がある。

インストール方法： `npm install ws`

このコマンド操作で、当該プロジェクトのディレクトリに `ws` が導入される。

`ws` を用いた WebSocket のサーバスクリプトを作成するには、スクリプトの冒頭で

```
const WebSocket = require('ws');
```

と記述する。これにより WebSocket の名前で `ws` ライブラリが利用できる。

A.5.1.1 サーバインスタンスの生成と設定

WebSocket のサーバのインスタンスは `Server` コンストラクタで生成する。

書き方： `new WebSocket.Server(設定情報)`

「設定情報」オブジェクトには、生成するサーバに関する情報（表 78）を与える。

表 78: `WebSocket.Server` コンストラクタに与える設定オブジェクトのプロパティ（一部）

プロパティ	解説
<code>port</code>	サーバが通信に使用するポート番号を与える。（必須）
<code>host</code>	サーバが通信に使用する NIC のホスト名（もしくは IP 青ドレス）を与える。省略時は当該計算機の全ての NIC が対象となる。
<code>path</code>	サーバのルーティングを行う場合に対象となる URL のパスを与える。
<code>noServer</code>	これを <code>true</code> に設定すると、通信開始の際の HTTP サーバは自動的に作成されない。（デフォルトは <code>false</code> ）

例. WebSocket サーバの生成

```
const wss = new WebSocket.Server( { port:8080 } );
```

このように記述することで、ポート番号 8080 で通信するサーバのインスタンス `wss` が生成される。またこのサーバは、引数のオブジェクトの `noServer` プロパティに値を設定していない（暗黙値：`false`）ので、クライアントからの通信を最初に受け付ける際の HTTP サーバを自動的に用意する。その後、この `wss` に対して `on` メソッドを使用して各種のイベントハンドラを登録する。

WebSocket サーバのインスタンスに設定するイベントハンドリングとして重要なものを以下に解説する。

■ WebSocket サーバ起動時のハンドリング

これは、サーバのインスタンスが起動してリクエストの受信を開始したことを意味する“listening” イベントに対するハンドリングである。

書き方： `サーバインスタンス.on("listening", イベントハンドラ)`

「サーバインスタンス」に“listening” イベントが発生した際の処理を行う関数である「イベントハンドラ」を登録す

¹⁴⁰RFC 6455

¹⁴¹HTTP ヘッダの `Upgrade` ヘッダを使用する。

る。この場合のイベントハンドラは特に引数を取らない。この場合の「イベントハンドラ」では主に、クライアントとの通信に先立つ準備に関する処理を行う。

■ クライアントからの接続要求を受けた際のハンドリング

これは、クライアントから当該サーバに対して接続要求を受けたこと¹⁴²を意味する "connection" イベントに対するハンドリングである。

書き方： サーバインスタンス.on("connection", イベントハンドラ)

「サーバインスタンス」に "connection" イベントが発生した際の処理を行う関数である「イベントハンドラ」を登録する。「イベントハンドラ」は次の2つの引数を取る。

引数 1：クライアントとの通信に関する WebSocket オブジェクト

引数 2：クライアントからの接続要求時に得られたリクエストオブジェクト

この内「引数 1」は必須、「引数 2」は省略可能である。特に「引数 1」に得られるオブジェクトはクライアントとの通信に関するものであり、これに対するイベントハンドリングの形で具体的な通信の処理を実装する。すなわち「イベントハンドラ」内で「引数 1」に対するイベントハンドリングを登録する。(これに関しては後述する)

「引数 2」からはクライアントの IP アドレスなどの情報が取得できる。

例. イベントハンドラの記述

```
(ws, req) => {  
  クライアントとの通信に関する WebSocket オブジェクト ws に  
  対するイベントハンドリングの登録処理などの記述  
}
```

■ サーバ終了時のハンドリング

サーバのインスタンスは、クライアントとの接続がない状態で close メソッドを実行するとその動作を終了する。このとき "close" イベント発生するので、それにするハンドリングを行う。

書き方： サーバインスタンス.on("close", イベントハンドラ)

「イベントハンドラ」は特に引数を取らない。この場合の「イベントハンドラ」では主に、サーバが終了する際の処理（クリーンアップなど）を行う。

A.5.1.2 クライアントとの通信に関する処理の実装

先の「■ クライアントからの接続要求を受けた際のハンドリング」(p.295)では、クライアントとの通信に関する WebSocket オブジェクトに対するイベントハンドリングを設定するが、その際の具体的な記述について解説する。

先の例「イベントハンドラの記述」では、WebSocket オブジェクトを ws として取得していたが、それに対するイベントハンドリングを on メソッドで登録する。

■ クライアントからメッセージを受信した際のハンドリング

書き方： WebSocket オブジェクト.on("message", イベントハンドラ)

「イベントハンドラ」の引数にはクライアントからのメッセージが渡されるので、これを受けた処理を実装する。

■ 接続終了時のハンドリング

書き方： WebSocket オブジェクト.on("close", イベントハンドラ)

「イベントハンドラ」は特に引数を取らない。この場合の「イベントハンドラ」では主に、クライアントとの通信を終了する際の処理（クリーンアップなど）を行う。

A.5.1.3 エラーハンドリング

サーバインスタンスや WebSocket オブジェクトの処理にエラーが発生した場合は "error" イベントが発生するので、それらに対してイベントハンドリングを on メソッド

on("error", イベントハンドラ)

で登録することができる。この場合、イベントハンドラの引数には当該エラーに関する Error オブジェクトが渡され

¹⁴²これは、クライアントからの接続要求を HTTP で受信して WebSocket プロトコルにアップグレードする段階である。

る。

A.5.1.4 通信相手に対するメッセージの送信

サーバ、クライアント共に、通信には WebSocket オブジェクトを使用する。これに対して send メソッドを実行することで、任意のタイミングで相手側にメッセージを送信することができる。

書き方： WebSocket オブジェクト.send(メッセージ)

「メッセージ」にはテキストデータ（文字列）あるいは ArrayBuffer を与えることができる。

A.5.1.5 接続中のクライアントの一覧

サーバインスタンスのプロパティ clients には接続中のクライアントの情報（WebSocket オブジェクト）を要素として持つ Set が保持されているので、これによって各クライアントの接続が参照できる。

A.5.1.6 実装例

WebSocket サーバの基本的な機能を理解するためのサーバスクリプトのサンプル WebSocketSV01.js を示す。

記述例：WebSocketSV01.js

```
1  const WebSocket = require('ws');           // モジュールの読み込み
2  const wss = new WebSocket.Server({ port: 8080 }); // WebSocketサーバの作成
3
4  // WebSocketサーバ準備完了時の処理
5  wss.on( "listening", () => {
6      console.log( "準備完了" );
7  });
8
9  // 新規クライアント接続時の処理
10 wss.on( "connection", (ws, req) => {
11     const ip = req.connection.remoteAddress; // クライアントのIPアドレス
12     console.log( "接続開始: '" + ip + "'" );
13
14     // メッセージ受信時の処理
15     ws.on( "message", message => {
16         console.log( "受信したデータ: " + message );
17         // データをエコーバック
18         ws.send( "[エコー] " + message );
19         if ( message == "StopServer" ) { // サーバの終了指示を受けた場合
20             wss.clients.forEach(c => { // すべての接続を閉じて
21                 c.close();
22             });
23             wss.close(); // このサーバを終了させる
24         }
25     });
26     // 接続接続終了時の処理
27     ws.on( "close", (event) => {
28         console.log( "接続終了 (WebSocket) \n" );
29     });
30     // エラー発生時の処理
31     ws.on( "error", (error) => {
32         console.log( "エラー: " + error.name + "/" + error.message );
33     });
34
35     // 接続完了時にクライアントにメッセージを送信
36     ws.send( "サーバ準備完了" );
37 });
38
39 // WebSocketサーバ終了時
40 wss.on( "close", () => {
41     console.log( "サーバが終了しました。" );
42 });
```

このサーバはクライアントからの WebSocket 接続に応えるものである。このサーバを Node.js で起動すると listening イベントが起り、コンソールに「準備完了」と出力される。例えばこのサーバと同じ計算機のクライアントから "ws://localhost:8080/" に接続をリクエストすると connection イベントが起り、コンソールに「接続開始: '::ffff:127.0.0.1」と出力される。あるいは、接続するクライアントによっては「接続開始: '::1」というように出力

される場合もあるが、これは IPv6 のループバックアドレスである。

参考)

サーバスクリプト WebSocketSV01.js がクライアントからの接続要求を受けるとクライアントの IP アドレスが ::ffff:127.0.0.1 のように表示されることがある。これは IPv6 の枠組みで IPv4 の IP アドレスを扱う場合の表現で **IPv4 マップド IPv6 アドレス** (IPv4-mapped (IPv6) address) と呼ばれる。

サーバスクリプト WebSocketSV01.js に対して次のような HTML コンテンツ WebSocketCL01.html によって Web ブラウザから WebSocket 接続する場合について考える。

記述例：WebSocketCL01.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>WebSocketCL01</title>
6   <script>
7     // WebSocketの接続を作成
8     var ws = new WebSocket("ws://localhost:8080/");
9     // 接続開始時の処理
10    ws.onopen = function(event) {
11      console.log( "接続開始" );
12    };
13    // メッセージ受信時の処理
14    ws.onmessage = function(event) {
15      console.log( "受信したデータ: " + event.data );
16    };
17    // 接続接続終了時の処理
18    ws.onclose = function(event) {
19      console.log( "接続終了: " + event.code + "/" + event.reason );
20    };
21    // エラー発生時の処理
22    ws.onerror = function(error) {
23      console.log( "エラー: " + error.name + "/" + error.message );
24    };
25    // サーバへメッセージを送信
26    function f1() {
27      ws.send( ta.value );
28    }
29    // 接続終了
30    function f2() {
31      ws.close(1000,"Normally closed.");
32    }
33    // WebSocketサーバに終了指示を送信
34    function f3() {
35      ws.send( "StopServer" );
36    }
37  </script>
38 </head>
39 <body>
40   <textarea id="ta"></textarea><br>
41   <input type="button" value="送信" onClick="f1()">
42   <input type="button" value="接続終了" onClick="f2()"><br>
43   <input type="button" value="！サーバ自体を終了！" onClick="f3()">
44 </body>
45 </html>
```

■ Web ブラウザの JavaScript の記述

Web ブラウザの JavaScript エンジンにおいても WebSocket 通信のプログラムの記述は Node.js の場合と同様であるが、WebSocket クラスは標準的に提供されており、ライブラリとして読み込む必要はない。ただし、イベントハンドリングの登録方法において若干の違いがある。WebSocketCL01.html のコードからわかるように、WebSocket オブジェクト ws へのイベントハンドリングの登録は

ws.onイベント名 = 関数;

という形式で記述する。

WebSocketCL01.html を Web ブラウザで表示すると図 84 の (a) のようになる。

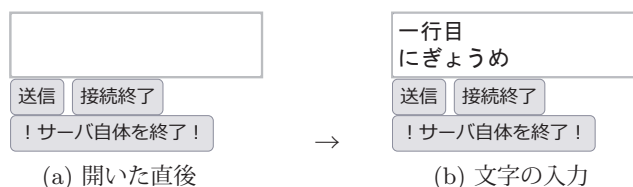


図 84: WebSocketCL01.html を Web ブラウザで開いたところ

WebSocketCL01.html を Web ブラウザ（クライアント）で開いた直後はサーバに対して接続要求が送信され、クライアントに open イベントが起こり、ブラウザのコンソールに「接続開始」と出力される。また接続要求を受けたサーバはクライアントに対して「サーバ準備完了」というメッセージを送信する。これを受けた Web ブラウザはコンソールに「受信したデータ: サーバ準備完了」と出力する。

図 84 の (b) のように textarea 要素に文字を入力して **送信** をクリックするとその内容がサーバに送信され、サーバ側のコンソールに

受信したデータ: 一行目
にぎょうめ

と表示される。この直後、サーバは受け取った内容の先頭に「[エコー]」を付けたものをクライアントに send メソッドで送信（エコーバック）する。これによってクライアント側のコンソールに

受信したデータ: [エコー] 一行目
にぎょうめ

と表示される。

図 84 の **接続終了** をクリックすると、クライアント側の WebSocket オブジェクトに close メソッドが実行されて通信が終了し、クライアント側のコンソールに「接続終了: 1000/Normally closed.」、サーバ側のコンソールに「接続終了 (WebSocket)」と表示される。接続が終了するとコンテンツのボタンをクリックしてもサーバは反応しなくなる。この後、再度接続（ブラウザ側の更新操作など）すると新たにサーバとクライアントが接続されてボタン操作が可能となる。

close メソッドの引数には**終了コード**と**終了メッセージ**を与えることができ、close イベントのハンドリングの際の Event オブジェクトからそれらを参照することができる。終了コードの 1000 は正常終了を意味する。

サーバとクライアントが接続された状態で図 84 の **!サーバ自体を終了!** をクリックすると、サーバにメッセージ "StopServer" が送信され、これを受けたサーバは、全てクライアントからの接続を強制的に終了し、サーバインスタンス自体に close メソッドを実行して終了する。この際、サーバ側のコンソールに

受信したデータ: StopServer
サーバが終了しました。
接続終了 (WebSocket)

と出力され、クライアント側のコンソールには

受信したデータ: [エコー] StopServer
接続終了: 1005/

と出力される。（出力内容と順序に若干の差異が生じることがある）

A.5.1.7 URL のパスに基づくルーティング

URL のパスに基づいて WebSocket のサービスをルーティングするための基本的な方法について例を挙げて解説する。次に示すスクリプト WebSocketSV02.js は 1 つのサーバ上で 2 つの異なる URL パス /path1、/path2 に対する WebSocket 接続を扱う例である。

記述例：WebSocketSV02.js

```
1  const WebSocket = require("ws");      // ライブラリ ws の読み込み
2  const http = require("http");          // ライブラリ http の読み込み
3  const url = require("url");            // ライブラリ url の読み込み
4
```

```

5  const server = http.createServer(); // HTTPサーバの作成
6
7  // URLパスが異なるWebSocketサーバの作成
8  const wss1 = new WebSocket.Server({ noServer: true, path: "/path1" });
9  const wss2 = new WebSocket.Server({ noServer: true, path: "/path2" });
10
11 // wss1に対するイベントハンドリングの登録
12 wss1.on( "connection", (ws, req) => {
13     const ip = req.connection.remoteAddress; // クライアントのIPアドレス
14     console.log( "(path1)接続開始: '"+ ip + "'" );
15     // メッセージ受信時の処理
16     ws.on( "message", message => {
17         console.log( "(path1)受信したデータ: " + message );
18         // データをエコーバック
19         ws.send( "[path1からエコー] " + message );
20     });
21     // 接続完了時にクライアントにメッセージを送信
22     ws.send( "(path1)サーバ準備完了" );
23 });
24
25 // wss2に対するイベントハンドリングの登録
26 wss2.on( "connection", (ws, req) => {
27     const ip = req.connection.remoteAddress; // クライアントのIPアドレス
28     console.log( "(path2)接続開始: '"+ ip + "'" );
29     // メッセージ受信時の処理
30     ws.on( "message", message => {
31         console.log( "(path2)受信したデータ: " + message );
32         // データをエコーバック
33         ws.send( "[path2からエコー] " + message );
34     });
35     // 接続完了時にクライアントにメッセージを送信
36     ws.send( "(path2)サーバ準備完了" );
37 });
38
39 // HTTPサーバに対するハンドリングの登録
40 server.on("upgrade", (request, socket, head) => {
41     // URLのパスを取得
42     const pathname = url.parse(request.url).pathname;
43     // ルーティング
44     if (pathname === "/path1") { // パス /path1 に対するルーティング
45         wss1.handleUpgrade(request, socket, head, (ws) => {
46             wss1.emit("connection", ws, request);
47         });
48     } else if (pathname === "/path2") { // パス /path2 に対するルーティング
49         wss2.handleUpgrade(request, socket, head, (ws) => {
50             wss2.emit("connection", ws, request);
51         });
52     } else {
53         socket.destroy();
54     }
55 });
56
57 // HTTPサーバの起動
58 server.listen(8080);

```

このスクリプトでは2つの WebSocket のサーバインスタンス wss1, wss2 をそれぞれ異なる URL パス /path1, /path2 のために作成している。また、コンストラクタの引数のプロパティ noServer に true を設定しており、最初は HTTP サーバを持たない形にしている。

このサーバでは最初に HTTP サーバである server を 8080 番ポートで起動する。それに対してクライアントから WebSocket 接続要求を受けると server に upgrade イベントが発生し、それを受けて、WebSocket のサーバインスタンス (wss1 または wss2) に handleUpgrade メソッドを実行して通信をアップグレードする。また、この際に接続が要求された URL を解析して URL パスを取り出し、それによって wss1, wss2 どちらのインスタンスを使用するかを選択する。

wss1, wss2 に対する処理の実装は先に解説した方法と同じであるが、今回のスクリプトでは簡素な形にしている。

このサーバに対するクライアント WebSocketCL02-1.html, WebSocketCL02-2.html を次に示す。この2つのクライ

アントはほぼ同じであるが、接続を要求する URL が異なる。

記述例：WebSocketCL02-1.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>WebSocketCL02-1</title>
6   <script>
7     // WebSocketの接続を作成
8     var ws = new WebSocket("ws://localhost:8080/path1");
9     // メッセージ受信時の処理
10    ws.onmessage = function(event) {
11      console.log( "受信したデータ: " + event.data );
12    };
13    // サーバへメッセージを送信
14    function f1() {
15      ws.send( ta.value );
16    }
17  </script>
18 </head>
19 <body>
20   <textarea id="ta"></textarea><br>
21   <input type="button" value="送信" onClick="f1()">
22 </body>
23 </html>
```

記述例：WebSocketCL02-2.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>WebSocketCL02-2</title>
6   <script>
7     // WebSocketの接続を作成
8     var ws = new WebSocket("ws://localhost:8080/path2");
9     // メッセージ受信時の処理
10    ws.onmessage = function(event) {
11      console.log( "受信したデータ: " + event.data );
12    };
13    // サーバへメッセージを送信
14    function f1() {
15      ws.send( ta.value );
16    }
17  </script>
18 </head>
19 <body>
20   <textarea id="ta"></textarea><br>
21   <input type="button" value="送信" onClick="f1()">
22 </body>
23 </html>
```

これら HTML を Web ブラウザで開いた様子を次に示す。

例. WebSocketCL02-1.html

送信

Web ブラウザで開いた直後

↓↓↓↓↓↓

path1へのメッセージ

送信

文字の入力

例. WebSocketCL02-2.html

送信

Web ブラウザで開いた直後

↓↓↓↓↓↓

path2へのメッセージ

送信

文字の入力

それぞれのクライアントにおいて textarea に文字を入力して 送信 をクリックする場合のサーバ、クライアントそれぞれのコンソールの出力を次に示す。

例. WebSocketCL02-1.html の場合

1. 接続要求を受信した際のサーバのコンソール
”(path1) 接続開始: '::ffff:127.0.0.1'”
2. サーバからのメッセージを受信した際のクライアントのコンソール
”受信したデータ: (path1) サーバ準備完了”
3. クライアントからのメッセージを受信した際のサーバコンソール
”(path1) 受信したデータ: path1 へのメッセージ”
4. サーバからのメッセージを受信した際のクライアントのコンソール
”受信したデータ: [path1 からエコー] path1 へのメッセージ”

例. WebSocketCL02-2.html の場合

1. 接続要求を受信した際のサーバのコンソール
”(path2) 接続開始: '::ffff:127.0.0.1'”
2. サーバからのメッセージを受信した際のクライアントのコンソール
”受信したデータ: (path2) サーバ準備完了”
3. クライアントからのメッセージを受信した際のサーバコンソール
”(path2) 受信したデータ: path2 へのメッセージ”
4. サーバからのメッセージを受信した際のクライアントのコンソール
”受信したデータ: [path2 からエコー] path2 へのメッセージ”

上の例は、WebSocketCL02-1.html, WebSocketCL02-2.html の2つのクライアントが同時に1つのサーバ WebSocketSV02.js にアクセスした際のものである。

A.5.2 Python による WebSocket

Python 言語で WebSocket サーバを実現するためのライブラリとして `websockets`¹⁴³ がある。このライブラリは Python 言語処理系に標準添付のもの（標準ライブラリ）ではなく、`pip` コマンドなどで改めて導入する必要がある。

例. OS のシェルで `pip` コマンドを用いて `websockets` をインストールする

```
pip install websockets
```

このライブラリで構築された WebSocket サーバでは、クライアントとの通信を**非同期のイテラブル**として扱うことができるため、セッションの識別が可能であること¹⁴⁴ が特徴である。

本書では、`websockets` を用いて WebSocket サーバを実装する方法に関して最も基本的な事柄について解説する。

A.5.2.1 基本的な考え方

`websockets` ライブラリでは、`asyncio` ライブラリ¹⁴⁵ による非同期処理の考え方に沿って WebSocket サーバを実現する。すなわち、**コルーチン**として作成した WebSocket サーバを**タスク**として `asyncio` のイベントループに投入して実行¹⁴⁶ する。

A.5.2.2 サーバの作成と実行の流れ

WebSocket サーバのコルーチンは `serve` コンストラクタで生成する。

書き方： `websockets.serve(ハンドラ関数, ホスト名, ポート番号)`

「ホスト名」（あるいは IP アドレス）の NIC からの通信を「ポート番号」で受け付けるサーバのコルーチンを生成して返す。得られたサーバは、クライアントからの通信に対する応答処理を「ハンドラ関数」（後述）で行う。

上の処理で得られたコルーチンを `await` で非同期実行すると、サーバのインスタンス（`WebSocketServer` クラス）が得られる。

書き方： `await サーバのコルーチン`

これと同時に、実際のサーバのタスクが `asyncio` のイベントループに投入されて稼働し始める。

A.5.2.3 ハンドラ関数

ハンドラ関数は 2 つの引数を取る関数（コルーチン）として実装する。

書き方： `async 関数名(接続オブジェクト, パス):`
(クライアントへの応答処理)

2 つの引数にはシステムから自動的に値が渡される。「接続オブジェクト」¹⁴⁷（`WebSocketServerProtocol` クラス）はクライアントとの通信に関する機能を司るものであり、各種のメソッドを実行することで具体的な通信の処理を行う。「パス」には、クライアントが要求した URL のパスが与えられる。

「接続オブジェクト」は**非同期のイテラブル**であり、クライアントから順次送信されたメッセージを**非同期のイテレーション**の形で順次受け取ること¹⁴⁸ ができる。また、後に説明する `recv` メソッドを用いてクライアントからのメッセージを受け取ることもできる。

■ メッセージの送受信

クライアントに対してメッセージを送信するには、接続オブジェクトに対して `send` メソッドを実行する。

書き方： `await 接続オブジェクト.send(メッセージ)`

「メッセージ」には文字列やバイト列（`bytes` 型）を与えることができる。

¹⁴³<https://pypi.org/project/websockets/>

¹⁴⁴Node.js で WebSocket サーバを構築する際は、セッションの同一性を識別するための仕組みをプログラマが実装しなければならない。

¹⁴⁵Python 言語処理系に添付されている**標準ライブラリ**で、非同期処理を支えている。

¹⁴⁶詳しくは Python 言語の関連書籍などの資料を参照のこと。拙書「Python3 入門 -Kivy による GUI アプリケーション開発, サウンド入出力, ウェブスクレイピング-」でも解説しています。

¹⁴⁷「プロトコルオブジェクト」と呼ばれることもある。

¹⁴⁸これによってセッションの一意性が保証される。

接続オブジェクトに対して `recv` メソッドを実行することによっても、クライアントからのメッセージを受け取ることができる。

書き方： `await 接続オブジェクト.recv()`

■ クライアントとの接続状態に関する情報

接続オブジェクトの各種プロパティ（表 79）から、クライアントとの接続状態に関する情報を知ることができる。

表 79: 接続状態に関するプロパティ（一部）

プロパティ	解 説
<code>local_address</code>	(サーバの IP アドレス, サーバ側のポート番号)
<code>remote_address</code>	(クライアントの IP アドレス, クライアント側のポート番号)
<code>open</code>	接続されている場合に <code>True</code> , それ以外の場合は <code>False</code> .
<code>closed</code>	接続が閉じられている場合に <code>True</code> , それ以外の場合は <code>False</code> .
<code>close_code</code>	接続が閉じられたときのコード
<code>close_reason</code>	接続が閉じられた理由を示す文字列

■ URL のパスに基づく処理のルーティング

ハンドラ関数の第 2 引数には、クライアントからリクエストされた URL のパスが与えられる。これを応用すると、URL のパスに基づく処理のルーティングが可能である。すなわち、第 2 引数に与えられた値を識別して、それに応じた処理を選択する形でサーバを実装することができる。

A.5.2.4 サーバの実装例（1）：非同期のイテレーションによる方法

クライアントから受信したメッセージをエコーバックする単純な WebSocket サーバの実装例 `PythonWSsv01.py` を示す。

記述例：`PythonWSsv01.py`

```
1  # coding: utf-8
2  import asyncio
3  import websockets
4
5  # ハンドラ関数
6  async def echo( ws, path ):
7      ss = ws.local_address[0]+'('+str(ws.local_address[1])+')'
8      cs = ws.remote_address[0]+'('+str(ws.remote_address[1])+')'
9      print('['+cs+'] セッション開始 ['+ss+'] URLパス="'+path+'"')
10     async for msg in ws:
11         print('['+cs+'] 受信したメッセージ '"+msg+'"')
12         await ws.send( 'Echo: ' + str(msg) )
13     scl = str(ws.close_code)+'/'+ws.close_reason
14     print('['+cs+'] セッション終了',scl)
15
16 # サーバタスク起動用コルーチン
17 async def svMain():
18     # サーバとなるコルーチンの生成
19     sv = websockets.serve( echo, 'localhost', 3000 )
20     # 上記をタスクとしてイベントループに投入
21     svi = await sv # 戻り値はサーバのインスタンス
22     # サーバ起動直後のイベントループ上のタスクを調べる
23     for t in asyncio.all_tasks(asyncio.get_running_loop()):
24         print( t )
25     print()
26     await svi.wait_closed() # サーバが完全に閉じられるまで処理を維持する
27
28 # サーバの起動
29 eloop = asyncio.new_event_loop() # イベントループを新規作成
30 asyncio.set_event_loop( eloop ) # それを現行のスレッドにアタッチ
31 eloop.run_until_complete( svMain() ) # サーバタスク起動用コルーチンを実行
32 eloop.run_forever() # イベントループの永続化
```

このサンプルでは、svMain が非同期処理としてイベントループ上で実行され、それがサーバの生成と起動の処理を行っている。またその際に、イベントループ上のタスクの一覧を出力する。(下記)

サーバスクリプト起動時の出力：

```
<Task pending name='Task-3' coro=<IocpProactor.accept.<locals>.accept_coro() running at ...>>
<Task pending name='Task-4' coro=<IocpProactor.accept.<locals>.accept_coro() running at ...>>
<Task pending name='Task-1' coro=<svMain() running at ...> cb=[_run.until_complete_cb() at ...]>
```

クライアントからのリクエストに対しては、非同期のハンドラ関数 echo が起動して対応する。

参考)

PythonWSsv01.py の末尾 4 行は、サーバタスクをイベントループで実行するための記述である。この 4 行の部分は次の 1 行の記述に替えることも可能であるので試されたい。

```
asyncio.run( svMain() )
```

上のサーバと同じ計算器環境で Web ブラウザを起動し、次の HTML PythonWSc101.html をクライアントとして通信を行う例を示す。

記述例：PythonWSc101.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <title>PythonWSc101</title>
6   <style>
7     #t1, #t2 {
8       width: 200px;
9       border: solid 2px #888888;
10    }
11  </style>
12  <script>
13    // WebSocketの接続を作成
14    var ws = new WebSocket("ws://localhost:3000/");
15    // サーバからメッセージを受信した際の処理
16    ws.onmessage = function(ev) { t2.value = ev.data; }
17    // 送信ボタンをクリックした際の処理
18    function f1() { ws.send( t1.value ); }
19    // セッション終了の処理
20    function f2() { ws.close(); }
21  </script>
22 </head>
23 <body>
24   <input type="text" id="t1"><br>
25   <input type="button" value="送信" onClick="f1()">
26   <input type="button" value="セッション終了" onClick="f2()"><br>
27   <input type="text" id="t2">
28 </body>
29 </html>
```

この HTML を Web ブラウザで開くと図 85 の (a) のように表示される。またこの際にサーバのコンソールに

[127.0.0.1(53448)] セッション開始 [127.0.0.1(3000)] URL パス="/"

と出力される。このメッセージの左端の「[127.0.0.1(53448)]」はクライアントの IP アドレスとポート番号であり、その後の「[127.0.0.1(3000)]」はサーバの IP アドレスとポート番号である。使用する Web ブラウザによっては IP アドレスが IPv6 の形式で出力されることがある。またクライアント側のポート番号はクライアントが適宜決定する。

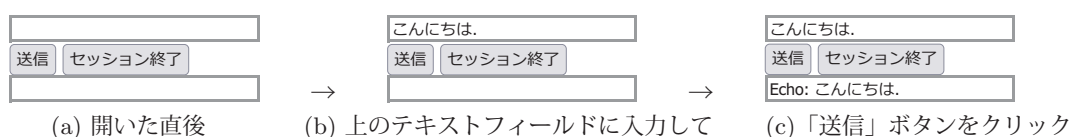


図 85: PythonWSc101.html を Web ブラウザで開いたところ

次に、図 85 の (b) のように、上側のテキストフィールドに文字を入力して「送信」ボタンをクリックすると、それがサーバに送信され、サーバのコンソールに

[127.0.0.1(53448)] 受信したメッセージ "こんにちは. "

と出力される。また、サーバは受信したメッセージをクライアントに送り返し（エコーバック）、これを受信した Web ブラウザは図 85 の (c) のように表示する。

このようなメッセージの入力、送信、エコーバックは何度でも繰り返すことができ、それが一連のセッションとしてハンドラ関数 echo の中の async for の反復処理として扱われる。（試されたい）

ブラウザに表示されている「セッション終了」ボタンをクリックする、あるいはブラウザを閉じるとサーバとの接続が閉じられ、サーバのコンソールに

[127.0.0.1(53448)] セッション終了 1005/

と出力される。

A.5.2.5 サーバの実装例（2）：通常の反復による方法

先の例 PythonWSsv01.py では、クライアントからのメッセージ受信のループを非同期のイテレーション（async for 文）で実現していたが、通常の反復制御（while 文など）で実現することもできる。次の PythonWSsv02.py は先のサーバと同じ機能を別の形で実装したものであり、while 文でクライアントからのメッセージ受信のループを実現している。

記述例：PythonWSsv02.py

```
1  # coding: utf-8
2  import asyncio
3  import websockets
4
5  # ハンドラ関数
6  async def echo( ws, path ):
7      ss = ws.local_address[0]+'('+str(ws.local_address[1])+')'
8      cs = ws.remote_address[0]+'('+str(ws.remote_address[1])+')'
9      print('['+cs+'] セッション開始 ['+ss+'] URLパス="'+path+'"')
10     try:
11         while True:
12             # クライアントからのメッセージを受信するループ
13             msg = await ws.recv()
14             print('['+cs+'] 受信したメッセージ "' + msg + '"')
15             await ws.send( 'Echo: ' + str(msg) )
16         except websockets.exceptions.ConnectionClosed:
17             # セッション終了の検知
18             scl = str(ws.close_code)+'/'+ws.close_reason
19             print('['+cs+'] セッション終了',scl)
20         except Exception as e:
21             # その他の例外の処理
22             print('[ '+ cs + ' ] 予期しないエラー:', e)
23
24 # サーバタスク起動用コルーチン
25 async def svMain():
26     # サーバとなるコルーチンの生成
27     sv = websockets.serve( echo, 'localhost', 3000 )
28     # 上記をタスクとしてイベントループに投入
29     svi = await sv
30     # 戻り値はサーバのインスタンス
31     # サーバ起動直後のイベントループ上のタスクを調べる
32     for t in asyncio.all_tasks(asyncio.get_running_loop()):
33         print( t )
34     print()
35     await svi.wait_closed()
36     # サーバが完全に閉じられるまで処理を維持する
37
38 # サーバの起動
39 asyncio.run( svMain() )
```

非同期のハンドラ関数 echo の中の while ループで、クライアントからのメッセージを順次受信する形になっている。また、セッション終了はそれを意味する websockets.exceptions.ConnectionClosed 例外によって検知している。

B Web ブラウザ関連

B.1 スクリプトのキャッシュの抑止

Web アプリケーションの開発過程では、HTML やそれに付随するスクリプトの内容を変更してその効果を Web ブラウザで確認するという作業を繰り返す。HTML コンテンツの変更はブラウザのページ更新ボタンなどにより表示に反映することができる。しかし、ワーカースクリプトやモジュールスクリプトは Web サーバを介して読み込むことが一般的であり、HTML コンテンツとは異なる仕組みでブラウザにキャッシュ保存（一時保存）されることが多い。その場合、スクリプトに加えた変更はブラウザのページ更新ボタンだけでは即座に有効にならないことがある。

ネットワーク経由のスクリプトファイルの変更を即座にブラウザに反映させるには、Web ブラウザのコンソールの「ネットワーク設定」の項目（コンソールのタブなど）で、キャッシュを無効にすると良い。

C Node.js

Node.js は V8 JavaScript エンジンに基づいて構築された JavaScript 実行環境で、イベントハンドリングによる入出力や通信を実現する。Node.js はサーバサイドの JavaScript 環境であり、Web サーバをはじめとするネットワーク上のシステム開発に多く用いられる。また JavaScript プログラムの実行速度も大きく、汎用のプログラミング環境としても用いることができる。ただし Node.js はサーバサイド指向のプログラミング環境であるため、HTML を扱うための DOM は標準的には搭載されていない¹⁴⁹。

Node.js は広く普及しており、多くのライブラリが標準的に添付されているだけでなく、サードパーティが提供するプログラムライブラリも豊富であり、npm というパッケージ管理機能を用いて導入と管理ができる。本書では Node.js に関する若干の応用例を示す。

C.1 REPL による対話的な利用方法

Node.js は OS のシェルから `node` コマンドで起動して、JavaScript 言語のインタプリタとして対話的に使用することができる。この利用形態では、OS のシェルの UI であるターミナルウィンドウを介して、キーボードから JavaScript の文や式、コマンドを入力すると、処理結果がターミナルウィンドウに出力される。このような利用形態は REPL (Read-Eval-Print Loop) と呼ばれる。

コマンド: `node`

例. `node` コマンドによる Node.js の起動 (Windows の場合)

```
C:\Users\katsu>node  ← node コマンドの発行
Welcome to Node.js v20.15.1.
Type ".help" for more information.
> ← REPL のプロンプト (ここに文や式、コマンドを入力する)
```

C.1.1 REPL の終了

`.exit` コマンドを入力すると REPL を終了して OS のシェルに戻る。

例. REPL の終了 (先の例の続き)

```
> .exit 
C:\Users\katsu> ← OS のシェルに戻った
```

C.1.2 プログラムの読み込み

ファイルからプログラムを読み込むには `.load` コマンドを用いる。

コマンド: `.load` プログラムのファイル名

次のようなプログラムのファイル `hello01.js` を読み込む方法を示す。

記述例: `hello01.js`

```
1 console.log( "Hello, this is Node.js." );
```

例. REPL での上記プログラムファイルの読み込み

```
> .load hello01.js  ← プログラムの読み込み
console.log( "Hello, this is Node.js." ); ← ファイルの内容を表示しながら
Hello, this is Node.js. ← それを実行する
undefined ← console.log の戻り値
```

ただしこの方法では、プログラムの読み込みに伴ってその内容が出力 (ウィンドウに表示) されてしまう。これは、通常はキーボードから入力される文や式が `.load` コマンドではファイルから入力されることが原因となっている。

¹⁴⁹Node.js 上で仮想的に DOM を扱うための `jsdom` というライブラリも公開されている。

.load コマンドとは別に、require 関数でプログラムを読み込む方法がある。この場合は、プログラムはモジュールという形で扱われる。

書き方： require(モジュール名)

「モジュール名」にはファイルのパスを与えることができる。モジュールの読み込み時はその内容は表示されない。

例. require 関数による読み込み

```
> require('./hello01.js') Enter    ←プログラム (モジュール) の読み込み
Hello, this is Node.js.    ←プログラム中の console.log の実行による出力
{}                          ←読み込まれたモジュール
```

先の例 hello01.js を次のようなモジュール hello02.js の形式に書き直す。

記述例：hello02.js

```
1 function greet() {
2     console.log( "Hello, this is Node.js." );
3     return "ReturnValue";
4 }
5
6 module.exports = {greet};
```

これを Node.js の REPL で実行する例を次に示す。

例. hello02.js を REPL で実行する

```
> var hello02 = require("./hello02.js") Enter    ←モジュールの読み込み
undefined
> hello02.greet() Enter    ←モジュール内の関数 greet の実行
Hello, this is Node.js.
'ReturnValue'                  ← greet の戻り値
```

C.2 標準入力からの同期入力

次に示すサンプル NodeInput.js は、Node.js で実行する JavaScript プログラムにおいて文字列入力を実現する例である。

記述例：NodeInput.js

```
1 const readline = require('readline');    // ライブラリの読み込み
2
3 // 入力機能のインスタンスの作成
4 const rl = readline.createInterface({
5     input: process.stdin,
6     output: process.stdout
7 });
8
9 // 入力完了を待つPromiseオブジェクトを返す関数
10 function askQuestion(rl, prompt) {
11     return new Promise((resolve, reject) => {
12         rl.question(prompt, (answer) => {
13             resolve(answer);
14         });
15     });
16 };
17
18 // 入力ループを実現する関数
19 async function inputLoop() {
20     let d;
21     while (true) {
22         d = await askQuestion(rl, '入力: endで終了> ');
23         console.log('-> 取得した値: ${d}');
24         if (d == 'end') {
25             rl.close();
26             break;
27         }
28     }
29 }
```



```
29 | };  
30 |  
31 | inputLoop();    // 入力ループの実行
```

このプログラムで実現されている inputLoop 関数は、イベントループをブロックすることなく Enter キーで完結する文字列の入力を実現するものである。これを node コマンドで実行する例を次に示す。

例. node コマンドによる NodeInput.js の実行

```
C:\¥>node NodeInput.js  
入力:end で終了> サンプルテキストの入力  
-> 取得した値: サンプルテキストの入力  
入力:end で終了> This is a sample text.  
-> 取得した値: This is a sample text.  
入力:end で終了> end  
-> 取得した値: end
```

参考文献

- [1] **Mozilla Developer Network (MDN) Web Docs**,
<https://developer.mozilla.org/ja/>
- [2] **React 公式インターネットサイト（日本語）**,
<https://ja.react.dev/>
- [3] **Apache HTTP Server Documentation**,
<https://httpd.apache.org/docs/>

索引

`*`, 37
`**`, 37
`*=`, 38
`+`, 37
`+=`, 38
`-`, 37
`-=`, 38
`-Infinity`, 38
`.babelrc` (React) , 254
`.exit`, 307
`.load`, 307
`/`, 37
`/=`, 38
`∴`, 58
`<`, 48
`<=`, 48
`==`, 48
`===`, 48
`>`, 48
`>=`, 48
`?`, 77
`$`, 113
`%`, 37
`%=`, 38
`&&`, 41
`_blank`, 12
`\`, 41, 42
`||`, 41
`!`, 41
`$`, 220
`$(document)`, 223
`$(this)`, 234
`$(window)`, 223
`<>`, 243
`</>`, 243
`@font-face`, 9
`¥`, 41, 42
`^`, 113
```, 45  
`xlink:href=`, 217  
16 進カラーコード, 6  
2 の補数, 68  
  
abort, 139  
abs, 40  
absolute, 17  
acos, 40  
add, 64  
addClass, 228  
addEventListener, 139  
after, 130, 229  
alert, 123  
align-items, 19  
altKey, 141, 142  
Apache HTTP Server, 281  
API, 30  
append, 222, 229  
appendChild, 129  
appendTo, 230  
application, 29  
arc, 196  
Array, 43, 50  
Array.from, 62, 65  
ArrayBuffer, 67, 168, 184  
arrayBuffer, 74, 187  
Array コンストラクタ, 50  
article, 4  
ASI, 31  
aside, 4  
asin, 40  
atan, 40  
attr, 224  
audio, 29  
AudioData, 184  
autoplay, 155, 156  
availHeight, 145  
availWidth, 145  
await, 74  
A コマンド (SVG の path 要素) , 214  
a 要素, 11  
  
Babel, 240  
background-color, 13  
Base64, 164  
base64-js, 165  
base64-js (ライブラリ) , 165  
base64js.fromByteArray, 165  
base64js.toByteArray, 165  
BaseHTTPRequestHandler, 272  
before, 130, 229

beginPath, 194  
bgroup, 7  
bigint, 37, 39, 48  
BigInt64Array, 67  
BigInt コンストラクタ, 39  
BigUint64Array, 67  
BinaryString, 74  
Blob, 73  
blob, 187  
block, 19  
blur, 138, 224  
body, 3  
bold, 211  
bolder, 211  
boolean, 48  
border, 14, 15  
border-box, 18  
border-radius, 15  
border-spacing, 21  
bottom, 16, 148  
box-sizing, 18  
br, 7, 13  
brak, 82  
break, 76  
Buffer, 263  
buffer, 68  
button, 22, 24  
byteLength, 68  
  
call, 99  
canvas, 190  
caption, 20  
case, 76  
catch, 116, 187  
CDN, 220  
ceil, 40  
center, 19  
CGI, 23, 283  
change, 138, 224  
charset, 3, 123  
checkbox, 26  
checked, 26, 27, 225, 226  
childNodes, 126  
children, 125, 231  
circle, 20, 210  
class, 2, 101, 102, 131  
class 式, 106  
clear, 62, 65, 152  
clearInterval, 143  
clearTimeout, 143  
click, 138, 171, 222, 224  
clientHeight, 146  
clientWidth, 146  
clientX, 141  
clientY, 141  
closePath, 195  
code, 142  
color, 5  
componentDidMount, 245  
componentWillUnmount, 245  
concat, 53  
console, 30  
const, 32  
constructor プロパティ, 97  
content, 3  
content-box, 18  
Content-Length, 290  
Content-Type, 290  
CONTENT\_LENGTH, 284, 290  
CONTENT\_TYPE, 284, 290  
contents, 231  
contextmenu, 138  
continue, 82  
controls, 155, 156  
copy-webpack-plugin, 258  
CORS, 150  
cos, 40  
crash, 116  
createElement, 125, 127, 153  
createElement (React) , 241  
createImageData, 199  
createObjectURL, 170  
createRoot (React) , 240  
createServer, 260  
createTextNode, 129  
CSS, 2, 4  
cssFloat, 136  
CSSStyleDeclaration, 135  
css メソッド, 223  
ctrlKey, 142  
currentTarget, 140  
currentTime, 156  
  
data, 129, 199, 227  
data-, 128, 227  
datalist, 25

- dataTransfer, 173
- DataView, 70
- Date オブジェクト, 108
- Date コンストラクタ, 108
- dblclick, 138, 224
- decimal, 20
- decode, 72
- decodeURI, 164
- decodeURIComponent, 164
- DedicatedWorkerGlobalScope, 180
- default, 76, 178
- defs, 206
- DELETE, 284
- delete, 36, 59, 62, 64
- disc, 20
- display, 13
- div, 15
- do, 79
- DOCTYPE, 2
- Document, 124
- document, 124, 145
- document.createElement, 125, 127, 153
- document.createTextNode, 129
- document.documentElement, 145
- document.documentElement.clientHeight, 146
- document.documentElement.clientWidth, 146
- document.documentElement.scrollHeight, 146
- document.documentElement.scrollWidth, 146
- document.getElementById, 127
- document.getElementsByClassName, 131
- document.querySelector, 134
- document.querySelectorAll, 134
- documentElement, 145
- DOM, 124
- DOMParser, 133
- DOMRect, 148
- dpi, 146
- drag, 172
- draggable, 172
- dragover, 172, 173
- dragstart, 172, 173
- drawImage, 198
- drop, 172, 173
- dropEffect, 173
- duration, 156
- E, 40
- ECMAScript, 1
- ellipse, 197, 211
- else, 75
- embed, 204
- empty, 229
- encode, 72
- encodeURIComponent, 164
- encodeURIComponent, 164
- end, 19, 263
- end\_headers, 273
- ended, 156
- entries, 62
- enumerable, 81
- eq, 223
- Error, 116, 118
- error, 139, 156
- evey, 56
- exp, 40
- export, 176
- express, 260, 266
- Express.js, 266
- fadeIn, 235
- fadeOut, 235
- false, 41
- falsy, 78
- favicon, 29
- fetch, 186
- Fetch API, 186
- fieldset, 27
- FIFO, 52
- File API, 167
- FileList, 167
- FileReader, 168
- File オブジェクト, 168
- fill, 55, 194, 195
- fillRect, 193
- fillStyle, 191
- fillText, 193
- filter, 233
- finally, 116
- find, 57
- findIndex, 57
- fixed, 17
- Flask, 276
- flex, 19
- Float32Array, 67
- Float64Array, 67
- floor, 40

flow, 19  
focus, 138, 224  
font, 29, 193  
font-family, 8, 211  
font-size, 5, 8, 211  
font-style, 5, 8, 211, 212  
font-weight, 5, 8, 211, 212  
footer, 4  
for, 80  
for ... in, 81  
for ... of, 80  
forEach, 120, 132  
form, 22  
formData, 187  
function, 83  
function\*, 92  
Function オブジェクト, 87, 220  
  
Generator, 92  
GeneratorFunction, 92  
GET, 284  
get, 23, 61  
getAttribute, 128  
getBigInt64, 70  
getBigUint64, 70  
getBoundingClientRect, 148  
getComputedStyle, 135  
getContext, 191  
getDate, 108  
getDay, 108  
getElementById, 127  
getElementsByClassName, 131  
getFloat32, 70  
getFloat64, 70  
getFullYear, 108  
getHours, 108  
getImageData, 201  
getInt16, 70  
getInt32, 70  
getInt8, 70  
getItem, 151  
getMilliseconds, 108  
getMinutes, 108  
getMonth, 108  
getSeconds, 108  
getUint16, 70  
getUint32, 70  
getUint8, 70

grid, 19  
GUI, 22, 137  
g 要素, 219  
  
has, 61, 64  
HEAD, 284  
head, 3  
height, 15, 145, 148, 154, 156, 203  
hgroup, 4  
hidden, 22  
hide, 235  
hover, 224  
HTML, 2  
html, 3, 231  
HTML Living Standard, 2  
HTMLAudioElement, 155  
HTMLBodyElement, 125  
HTMLCollection, 125, 132  
HTMLDocument クラス, 124  
HTMLElement, 124, 222, 223  
HTMLHeadElement, 125  
HTMLImageElement, 153  
HTMLMediaElement, 155  
HTMLVideoElement, 155, 156  
HTML 要素, 2  
HTML 要素の生成, 127  
http-server コマンド, 293  
http-server パッケージ, 293  
http.server モジュール (Python) , 272  
HTTP\_USER\_AGENT, 284  
HTTPServer, 272  
HTTP ステータスコード, 290  
HTTP チャンクエンコーディング, 272  
HTTP ヘッダー, 290  
http ライブラリ (Python) , 272  
HTTP リクエスト, 283  
http (ライブラリ) , 260  
H コマンド (SVG の path 要素) , 214  
h 要素, 4, 6  
  
I/O タスク, 158  
id, 2  
ID セレクタ, 5  
if, 75  
IIFE, 87  
Image, 153  
image, 29, 216  
ImageBitmap, 184  
ImageData, 199



img, 21  
import, 176  
includes, 44, 55  
IncomingMessage クラス, 260  
indexOf, 44, 56  
Infinity, 38  
inline, 19  
innerHeight, 145  
innerWidth, 145  
input, 22, 138, 224  
insertAfter, 230  
insertBefore, 129, 230  
instanceof, 118  
Int16Array, 67  
Int32Array, 67  
Int8Array, 67  
InternalError, 117  
intervalID, 143  
IPv4 マップド IPv6 アドレス, 297  
isFinite, 39  
isNaN, 39  
italic, 5, 211  
item, 125  
  
JavaScript, 1  
JavaScriptCore, 1, 30  
join, 43  
jQuery, 220  
jQuery オブジェクト, 220  
jQuery オブジェクト生成関数, 220  
jQuery 関数, 220  
JSON, 163  
json, 187  
JSON.parse, 163  
JSON.stringify, 163  
JSX, 240, 241  
  
key, 142, 151  
keydown, 138, 224  
keypress, 224  
keys, 62  
keyup, 138, 224  
  
label, 23  
lang, 3  
lastIndexOf, 44, 56  
left, 16, 148  
length, 42, 68, 125  
let, 32  
  
letter-spacing, 7  
li, 19  
LIFO, 52  
lighter, 211  
line, 211  
line-height, 7  
line-through, 211  
lineCap, 191  
lineJoin, 191  
lineTo, 195  
lineWidth, 191  
link, 5, 29  
list, 25  
list-style-type, 20  
listen, 260, 268  
load, 138  
loadend, 168  
loadstart, 156  
localStorage, 150  
Location, 290  
log, 30, 40  
log10, 40  
log2, 40  
loop, 155, 156  
lower-alpha, 20  
lower-greek, 20  
lower-roman, 20  
L コマンド (SVG の path 要素) , 213  
  
main, 4  
map, 121  
Map オブジェクト, 61  
Map コンストラクタ, 61  
margin, 14, 15  
match, 112  
Math, 40  
MathJax, 238  
message, 116, 180  
MessagePort, 184  
meta, 3  
metaKey, 142  
meter, 28  
MIME タイプ, 29, 73, 167  
miterLimit, 192  
miter 長, 210  
mousedown, 138, 224  
mouseenter, 224  
mouseleave, 224

mousemove, 138  
mouseout, 138  
mouseover, 138  
mouseup, 138, 224  
moveTo, 194  
multiple, 26  
muted, 155, 156  
M コマンド (SVG の path 要素) , 213  
  
name, 3, 23, 98, 116  
NaN, 39  
nav, 4  
new, 96  
next, 92  
node, 307  
Node.js, 1, 30, 260, 293, 307  
NodeList, 126, 134  
nodeName, 127  
nodeValue, 129  
none, 12, 207  
normal, 211  
npm, 253  
npm コマンド, 293  
null, 49  
number, 37, 48  
Number.MAX\_SAFE\_INTEGER, 37  
Number.MAX\_VALUE, 37  
Number.MIN\_SAFE\_INTEGER, 37  
Number.MIN\_VALUE, 37  
Number.NEGATIVE\_INFINITY, 38  
Number.POSITIVE\_INFINITY, 38  
Number コンストラクタ, 37, 46  
  
Object, 60  
object, 204, 205  
Object.create, 98  
Object.entries, 60  
Object.keys, 60  
oblique, 211  
OffscreenCanvas, 184  
offsetX, 141  
offsetY, 141  
ol, 19  
on, 224  
OOP, 96  
opacity, 22  
OpenType, 9  
opentype, 9  
option, 25, 26

orient, 25  
otc, 9  
otf, 9  
outerHeight, 145  
outerWidth, 145  
overline, 211  
  
package.json, 255  
padding, 14, 15  
pageX, 141  
pageY, 141  
parent, 231  
parentNode, 126  
parse, 261, 262  
parseFloat, 46  
parseFromString, 133  
parseInt, 46  
password, 24  
path, 213  
patterns:, 258  
pause, 155, 156  
paused, 156  
PI, 40  
play, 155, 156  
playbackRate, 156  
points, 207  
polygon, 207  
polyline, 208  
pop, 52  
position, 16  
POST, 284  
post, 23  
poster, 156  
postMessage, 180  
PostScript, 3  
pow, 40  
ppi, 146  
prepend, 229  
prependTo, 230  
preserveAspectRatio, 204  
preventDefault, 173  
process.stdin, 286  
process.stdout.write, 285  
progress, 28  
Promise, 74, 155, 158, 186  
prop, 225  
Props (React) , 243  
prototype, 97

- push, 51
- PUT, 284
- putImageData, 199
- Python, 272, 293
- p 要素, 4, 6
- QUERY\_STRING, 23, 283, 284, 290
- querySelector, 134
- querySelectorAll, 134
- querystring (ライブラリ) , 260
- radio, 27
- random, 40
- range, 24
- RangeError, 117
- ratechange, 156
- React, 240
- React.Component, 242
- React.Fragment, 243
- ReactDOM オブジェクト, 240
- React オブジェクト, 240
- React 要素, 241
- ReadableStream, 184
- readAsArrayBuffer, 168
- readAsBinaryString, 168
- readAsText, 168
- ready, 237
- rect, 210
- recv, 303
- reduce, 122
- reduceRight, 122
- ReferenceError, 117
- RegExp, 112
- relative, 17
- REMOTE\_ADDR, 284
- REMOTE\_PORT, 284
- remove, 131, 229
- removeChild, 131
- removeClass, 228
- removeItem, 152
- render (React) , 241
- repeat, 44
- REPL, 307
- replace, 114
- replaceAll, 114
- Request, 186
- request, 277
- REQUEST\_METHOD, 283, 284, 290
- REQUEST\_SCHEME, 284
- require, 308
- reset, 139
- resize, 138
- Response, 187, 276
- restore, 202
- result, 169
- return, 83
- reverse, 54
- rgb 関数記法, 6
- right, 16, 148
- rotate, 201, 211, 213, 217
- round, 40
- rp, 11
- rt, 11
- RTCDataChannel, 184
- ruby, 11
- save, 202
- Scalable Vector Graphics, 203
- scale, 201, 217
- screen, 145
- screen.availHeight, 145
- screen.availWidth, 145
- screen.height, 145
- screen.width, 145
- screenX, 141
- screenY, 141
- script, 29, 123
- scroll, 138, 224
- scrollHeight, 146
- scrollWidth, 146
- search, 111
- section, 4
- select, 22, 26, 138
- selected, 26
- self, 180
- send, 302
- send\_header, 273
- SEO, 3
- serve\_forever, 272
- SERVER\_ADDR, 284
- SERVER\_NAME, 290
- SERVER\_PORT, 284, 290
- ServerResponse クラス, 260
- Server クラス, 260
- sessionStorage, 150, 152
- Set, 64
- set, 61

setAttribute, 128  
setBigInt64, 71  
setBigUint64, 71  
setData, 173  
setDate, 109  
setEncoding, 263  
setFloat32, 71  
setFloat64, 71  
setFullYear, 109  
setHours, 109  
setInt16, 71  
setInt32, 71  
setInt8, 71  
setItem, 150  
setLineDash, 192  
setMilliseconds, 109  
setMinutes, 109  
setMonth, 109  
setSeconds, 109  
setState, 244  
setTimeout, 143  
setUint16, 71  
setUint32, 71  
setUint8, 71  
Set オブジェクト, 64  
Set コンストラクタ, 64  
SGML, 2  
shebang, 284  
shift, 52  
shiftKey, 142  
show, 235  
sign, 40  
sin, 40  
size, 62, 64  
slice, 53  
slideDown, 235  
slideUp, 235  
some, 57  
SOP, 150  
sort, 54  
source, 155  
span, 7  
SpiderMonkey, 1, 30  
splice, 53  
split, 43  
sqrt, 40  
square, 20  
src, 156

start, 19  
state, 244  
State (React) , 244, 245  
static, 16, 17, 102, 103  
sticky, 17  
Storage, 150  
string, 41, 48  
String コンストラクタ, 41  
stroke, 194, 195  
stroke-dasharray, 208  
stroke-linecap, 209  
stroke-linejoin, 209  
stroke-miterlimit, 210  
strokeRect, 193  
strokeStyle, 191  
strokeText, 193  
style, 5, 29, 135  
submit, 139, 224  
substr, 44  
substring, 44  
SVG, 203  
svg 要素, 203  
switch, 76  
Symbol, 47  
symbol, 48  
Symbol.iterator, 94  
SyntaxError, 117  
  
table, 20  
tan, 40  
target, 12, 140, 173  
tbody, 21  
td, 20  
Text, 129  
text, 23, 29, 74, 187, 211, 222, 231  
text-align, 7  
text-decoration, 7, 8, 12, 211, 212  
text-indent, 7  
textarea, 22, 24  
textContent, 131, 174  
TextDecoder, 72  
TextEncoder, 72  
tfoot, 21  
thead, 21  
then, 186  
this, 91, 96, 102  
this.props, 243  
throw, 118

- Timeout, 143
- timeoutID, 143
- timeStamp, 140
- timeupdate, 156
- title, 3, 203
- toggleClass, 228
- toISOString, 110
- toLocaleString, 109
- top, 16, 148
- toString, 56
- tr, 20
- Transferable, 184
- transform, 217
- TransformStream, 184
- translate, 201, 217
- true, 41
- TrueType, 9
- truetype, 9
- truthy, 78
- try, 116
- ttc, 9
- ttf, 9
- type, 22, 140
- TypedArray, 67
- TypeError, 117
- typeof, 48
  
- UI, 2
- Uint16Array, 67
- Uint32Array, 67
- Uint8Array, 67
- ul, 19
- undefined, 30, 32, 48
- underline, 211
- Unicode, 42
- UNIX エポック, 108
- UNIX 時間, 108
- unload, 138
- unshift, 52
- upper-alpha, 20
- upper-roman, 20
- URIError, 117
- Url, 261
- URL エンコーディング, 164
- URL オブジェクト, 170
- url (ライブラリ) , 260
- use, 206, 218
- useEffect, 245
- useState, 244
- util, 72
  
- V8, 1, 30
- val, 225
- value, 23
- values, 62
- var, 32
- vertical, 25
- vertical-align, 19
- video, 29
- VideoFrame, 184
- videoHeight, 156
- videoWidth, 156
- viewBox, 203
- visibility, 22
- visible, 22
- volume, 156
- volumechange, 156
- V コマンド (SVG の path 要素) , 214
  
- W3C, 2
- Web Workers, 180
- webpack, 253
- webpack.config.js, 255
- WebSocket, 294
- WebSocket.Server, 294
- websockets, 302
- websockets.exceptions.ConnectionClosed, 305
- wfile, 273
- while, 79
- width, 15, 145, 148, 154, 156, 203
- window, 123, 124, 145
- window.getComputedStyle, 135
- window.innerHeight, 145
- window.innerWidth, 145
- window.outerHeight, 145
- window.outerWidth, 145
- window.screen, 145
- Window クラス, 124
- woff, 9
- woff2, 9
- Worker, 180
- WritableStream, 184
- write, 263
- writeHead, 263
- WSGI, 288
- wsgi.errors, 290
- wsgi.input, 290

x, 148, 211  
XHTML, 2  
XML, 2, 203  
XMLDocument, 124  
  
y, 148, 211  
yield, 92  
yield\*, 93  
  
Z コマンド (SVG の path 要素) , 213  
  
アクセス権, 284  
アスキーコード, 74  
値, 58  
値の比較, 47  
アットルール, 9  
アラートダイアログ, 123  
アルファ値, 6, 199  
アロー関数式, 87, 258  
アンカー要素, 11  
移譲, 184  
異常終了, 116  
イテラブル, 64, 80  
イテラブルオブジェクト, 94  
イテレータ, 62, 80, 94, 113  
移転, 184  
イベント, 137  
イベントオブジェクト, 140  
イベントキュー, 137  
イベント駆動型プログラミング, 137  
イベントハンドラ, 137  
イベントハンドリング, 137  
イベントループ, 137, 158  
イミュータブル, 41, 73  
インクリメント, 37  
インスタンス, 96, 102  
インスタンスのコンストラクタを調べる方法, 98  
インスタンスベース, 101  
インターバル ID, 143  
インデックス, 41, 50  
インラインスタイル, 29, 135, 136  
インラインレイアウト, 13  
エスケープシーケンス, 42, 74  
エラー, 116  
エラーオブジェクト, 116  
エラーの種類, 117  
円, 210  
エンクロージャ, 91  
エンコーディング, 3  
  
円周率, 40  
応答オブジェクト, 260, 262, 267  
応答ステータス, 263  
応答ヘッダ, 263  
応答ボディ, 263  
オブジェクト, 58  
オブジェクトから配列への変換, 60  
オブジェクト指向プログラミング, 96  
オブジェクトのクラスを調べる方法, 105  
オブジェクトのパターン, 36  
オブジェクトの要素の個数, 60  
親ノード, 126  
親要素, 231  
オリジン, 150  
オリジン間リソース共有, 150  
折れ線, 207  
音声, 155  
オーバーライド, 101, 104  
改行, 42  
解像度, 146  
拡張クラス, 104  
可視属性, 22  
仮想 DOM, 240  
型付き配列, 67  
カプセル化, 106  
空オブジェクト, 60  
空配列, 51  
空文, 31  
空要素, 2  
仮引数, 85  
環境変数, 23, 283  
関数, 31, 83  
関数コンポーネント, 242  
関数式, 87  
関数自体のスコープ, 86  
関数の再帰的呼び出し, 88  
外的スタイルシート, 29  
外部関数, 89  
外部スタイルシート, 29, 135  
外部リソース, 29  
基底クラス, 104  
キャメルケース, 136  
キー, 58, 61  
擬似クラス, 12  
逆正弦関数, 40  
逆正接関数, 40  
逆余弦関数, 40  
クエリストリング, 164, 290, 292



クエリパラメータ, 164, 292  
クエリ文字列, 261, 262  
矩形, 210  
クラス, 102  
クラスコンポーネント, 242  
クラスセレクト, 5  
クラスの継承, 104  
クラス変数, 97  
クラスベース, 101  
繰り返しの表記, 111  
クリーンアップ関数, 245  
クロージャ, 89, 91  
グラフィックコンテキスト, 191  
グローバルスコープ, 32, 33  
グローバル属性, 3  
グローバル変数, 32, 124  
グローバル変数の所在, 124  
継承, 98, 104  
検索, 111  
検索エンジン最適化, 3  
現在時刻の取得, 108  
固定ビット長の数値の配列, 67  
子ノード, 125  
コメント (JavaScript), 31  
コメント (CSS), 6  
コメント (HTML), 4  
固有ファイル型指定子, 167  
子要素, 231  
コロンの, 58  
コンストラクタ, 96  
コンテキスト, 191  
コンテナ, 15  
コンテンツデリバリーネットワーク, 220  
コンポーネント, 241, 242  
コンポーネントの名称に関する注意, 243  
互換モード, 2  
再帰的な代入, 38  
再帰的呼び出し, 88  
サイトアイコン, 29  
削除と挿入 (配列), 53  
サブクラス, 104  
サブタイプ名, 29  
四角形, 210  
式, 30, 31  
式文, 31  
四捨五入, 40  
指数関数, 40  
指数表現, 40  
自然対数の底, 40  
シバン, 284  
書体, 9  
シングルクオート, 42  
シングルスレッド, 180  
進捗インジケータ, 28  
シンボル, 47  
ジェネレータ関数, 92  
実引数, 85  
自動セミコロン挿入, 31  
条件付きレンダリング, 245  
条件分岐, 75  
剰余, 37  
垂直タブ, 42  
数値リテラル, 39  
スコープ, 32  
ステート (React) , 244  
ステート変数 (React) , 244  
ストローク, 190, 207  
スプレッド構文, 66, 85  
スライダ, 24  
スライダに目盛りを表示する方法, 25  
スライダを垂直方向にする方法, 25  
スーパークラス, 104  
正規表現, 111  
正弦関数, 40  
正接関数, 40  
静的, 102  
整列 (配列), 54  
セマンティックウェブ, 4  
セミコロン, 31  
セレクト, 5  
セレクトリスト, 5  
宣言, 32  
選択要素, 26  
絶対座標, 214  
絶対値, 40  
全要素の削除 (Set), 65  
全要素の取出し (Map), 62  
相対座標, 214  
即時実行関数, 87  
属性, 2  
属性ノード, 125  
対数関数, 40  
タイプ名, 29  
タイマー, 143  
タイマー ID, 143  
タイマー処理の反復, 143

タイマータスク, 158  
多角形, 207  
タグ付きテンプレートリテラル, 45  
タブ, 42  
単位の変換, 147  
楕円, 211  
ダブルクオート, 42  
チェックボックス, 26  
置換処理, 114  
直線, 211  
通常フロー, 12  
定数, 32  
テキスト, 211  
テキストノード, 125, 129  
テキストフィールド, 23  
テンプレート文字列, 45  
テンプレートリテラル, 45  
ディレクティブ, 281  
デクリメント, 37  
デフォルトエクスポート, 178  
データ構造, 50  
データ属性, 128, 227  
データリスト要素, 25  
糖衣構文, 101  
匿名関数, 87  
トランスパイル, 240  
同一オリジンポリシー, 150  
動画, 156  
ドキュメントノード, 125  
ドラッグ, 172  
ドラッグアンドドロップ, 172  
ドロップ, 172  
ドロー系グラフィックス, 203  
内的スタイルシート, 29  
内部関数, 89  
内部スタイルシート, 29, 135  
名前, 58  
名前空間, 177, 203  
任意の個数の引数, 85  
塗りつぶし, 190, 207  
ネストされた関数定義, 89  
ハイパーテキスト, 2  
ハイパーリンク, 11  
配列, 43, 50  
配列の指定範囲の削除, 53  
配列の反転, 54  
配列の編集, 53  
配列の要素数, 51  
配列の連結, 53  
配列への変換 (Set), 65  
配列リテラル, 50  
破線, 208  
反復可能, 66, 80  
反復可能オブジェクト, 64, 94  
バイエンディアン, 70  
バイトオーダー, 69  
バッククオート, 42, 45  
バックスペース, 42  
バックスラッシュ, 41, 42  
バックティック, 42  
パス, 194  
パスワード, 24  
パターンマッチ, 112  
パブリックインスタンスフィールド, 103  
パブリッククラスフィールド, 102  
パブリック静的フィールド, 103  
パブリックフィールド, 102  
パーサ, 133  
パーセントエンコーディング, 164  
パーミッション, 284  
引数, 83  
非数, 38, 39  
非数の判定, 39  
非同期 I/O 操作, 158  
非同期関数, 158  
非同期処理, 158, 186  
非同期タスク, 158  
標準出力, 30  
標準入力, 23  
標準モード, 2  
ビッグエンディアン, 70  
ビュー (型付き配列), 67  
ビューボックス, 203  
ビューポート, 145, 203  
描画コマンド, 213  
描画コンテキスト, 191  
ビルド作業, 253–255  
ファビコン, 29  
フィル, 190, 207  
フィールドセット要素, 27  
フォント, 8, 9  
フォームフィールド, 42  
袋文字, 193  
符号, 40  
フック, 244, 245  
不透明度, 6, 22, 199

負の無限大, 38  
部分配列, 53  
部分文字列, 44  
ブロック, 31  
ブロックレイアウト, 13  
分割代入, 34  
文書型宣言, 2  
プライベートインスタンスフィールド, 106  
プライベートデータプロパティ, 106  
プライベートメソッド, 106  
プリミティブ型, 48  
プレースホルダ, 45  
プログレスバー, 28  
プロトタイプチェーン, 99, 101  
プロトタイプベース, 101  
プロパティ, 31, 58  
プロパティのカプセル化, 106  
プロパティの削除, 59  
プロミス, 74  
プロミスオブジェクト, 155  
平方根, 40  
変数, 32  
変数の廃棄, 36  
べき乗, 37, 40  
ベクターグラフィックス, 203  
ベースライン, 212  
ペイント系グラフィックス, 203  
ページ記述言語, 3  
包含ブロック, 15  
ボタン, 24  
マイクロタスク, 158  
マーカー, 20  
ミドルエンディアン, 69  
ミュータブル, 41, 67  
無限大, 38  
無名関数, 87  
メインスレッド, 180  
メソッド, 31, 91, 97  
メソッドチェーン, 32, 222  
メソッドの短縮記法, 91  
メディアタイプ, 29, 73, 167  
メータ, 28  
文字コード, 42  
モジュール, 175, 254  
文字列, 41, 211  
文字列探索, 111  
文字列の繰り返し, 44  
文字列の長さ, 42

文字列リテラル, 41  
文字列をバイナリデータに変換, 72  
戻り値, 83  
有限値の判定, 39  
要素型セレクタ, 5  
要素ではないプロパティ (配列), 58  
要素の個数の調査 (Map), 62  
要素の個数の調査 (Set), 64  
要素の削除 (Map), 62  
要素の削除 (オブジェクト), 59  
要素の削除 (配列), 52  
要素の存在確認 (Map), 61  
要素の存在確認 (Set), 64  
要素の追加 (配列), 51  
要素の追加と削除 (Set), 64  
要素ノード, 125  
要素へのアクセス (Map), 61  
余弦関数, 40  
ライフサイクル (React), 245  
ラジオボタン, 27  
ラスターグラフィックス, 190  
ラベル, 23  
乱数, 40  
リクエストオブジェクト, 260, 262, 267  
リクエストのルーティング, 268, 270  
リクエストハンドラ, 260, 266, 272  
リクエストリスナ, 260  
リスト, 50  
リストとキー (React), 251  
リスニングハンドラ, 260  
リスニングリスナ, 260  
リトルエンディアン, 69  
累算的な代入, 38  
ルビ, 11  
ルーティングハンドラ, 266  
ルーティングメソッド, 266  
ルートハンドラ, 266  
ルートパラメータ, 270  
ルート要素, 3  
例外, 116  
例外処理, 116  
例外の種類, 117  
例外を発生させる方法, 118  
連想配列, 58, 61  
論理属性, 26, 155, 225, 226  
論理値, 41  
ワーカー, 180  
ワーカースレッド, 180



## 「JavaScript 入門」

ー Web アプリケーション開発のための基礎

### テキストの最新版と更新情報

本書の最新版と更新情報を，プログラミングに関する情報コミュニティ Qiita で配信しています。

→ <https://qiita.com/KatsunoriNakamura/items/4bb17b0238fcc9c4b6ed>



上記 URL の QR コード

本書はフリーソフトウェアです，著作権は保持していますが，印刷と再配布は自由にさせていただいて結構です。（内容を改変せずをお願いします） 内容に関して不備な点がありましたら，是非ご連絡ください。ご意見，ご要望も受け付けています。

#### ● 連絡先

[nkatsu2012@gmail.com](mailto:nkatsu2012@gmail.com)

中村勝則