JavaScript が好きになる記事

いーはちえる

はじめに

先日，インプレスジャパンの『改定 6 版 TrueType フォントパーフェクトコレクション[*1]』を購入して「こころぴょんぴょん[*2]」している，いーはちえるです．

皆さんは **JavaScript** というスクリプト言語をご存知ですか．JavaScript は主に Web ページの表現を高めるために使われるものです．近年では Web アプリケーションの普及などに伴い，Flash に代わってより多くの場面で使われるようになりました．

しかしながら，JavaScript は**良くない言語**であるとの評価を良く耳にします．また，そのような評価により，普段 JavaScript を使うことのない人にも悪い印象を与えてしまっているように感じます．

そこで，皆さんが JavaScript を好きになってくれるような事実をいくつかまとめてみました．この記事によって，少しでも JavaScript のことを好きになって貰えれば幸いです．

なお，以下では何かしらのプログラミング経験がある方に向けた文章を記述します．ですので，経験がない方などには，分からない専門用語や謎の英数字の羅列のようなものがたくさん出てきますので，内容が伝わらないかも知れません．けれども，文面から表面的な印象を感じ取っていただければ幸いです．

[*1] 非常に良い本です．皆さんも是非一冊．
http://www.impressjapan.jp/books/3361

[*2] http://www.gochiusa.com/

1　型

1.1　数値型について

JavaScript には，変数宣言の際に型の指定をしませんが，文字列・真偽値・配列などの型が存在します．もちろん，数値を表す型もあるのですが，JavaScript の数値型はただ 1 つのみです．**倍精度浮動小数点型のみ**です．

ところで，JavaScript にも当然ビット演算はあります．どうしているのかというと，小数点以下を切り捨てて整数にして演算を行います．また，このときの整数が 32bit で表現できる範囲外にある場合は，下位 32bit 部分が演算に使われます．さらに，演算結果は符号付きの結果になります．サンプルコードをリスト 1 に示します．

1.2　typeof

変数の型を調べる typeof は存在します．typeof は変数の型を表す文字列を返します．

▼ リスト 1　ビット演算

```
var x = 3.3;
var y = 5.9;

// 整数部分でビット演算
var and = x & y; // => 1
var or  = x | y; // => 7
var xor = x ^ y; // => 6

// 下位32ビットのみ使われる
var hugeV = 0x6800000068; // = 446676598888
var mask  = 0xff000000ff; // = 1095216660735
var res   = hugeV & mask; // => 0x68 = 104

// 左シフト演算は32ビットでローテートする
var ls = 1 << 33; // => 2
// 右シフト演算も32ビットでローテートする！
var rs = 0x40000000; >> 62 // => 1

// ビット演算の結果は符号付き！
var plus = 1 << 31 // => -2147483648
```

▼リスト 2　有効値のチェック

```
var print = function(arg) {
  //null, undefined, 無効な数値の順にチェック
  if( arg === null      ||
      arg === undefined ||
      (typeof arg === "number"
         && !isFinite(arg))
    ) {
    return;
  }
  console.log(arg);
}
```

例えば数値型の場合は ‘number’，文字列の場合は ‘string’ と文字列が返ってきます．

JavaScript の面白いところは，想像もつかないような結果を返してくれることです．例えば，配列を指す変数について，普通は ‘array’ が返ってくることを期待します．しかし，JavaScript は違います．‘**object**’ が返ってきます．JavaScript にはオブジェクト型というものがあるので，これだけでは判別できません．

また，null や NaN についても typeof は有効です．ただし，null の場合は ‘**object**’，NaN の場合は ‘**number**’ となります．

なお，変数を初期化されていないときに入っている undefined という値に対しては，typeof は ‘**undefined**’ を返します．

1.3　偽となる値

C 言語では 0 が偽，それ以外が全て真として評価されます．Ruby では nil と false が偽で，それ以外が真となります．

JavaScript では，**0**，**NaN**，**””**（空文字列），**false**，**null**，**undefined** が偽となり，それ以外が真となります．偽となる値がすごく多いです．

なので，例えば関数の引数などで NaN や null，undefined だけを弾きたい場合，リスト 2 のような記述をしなければなりません．さらに付け加えるとすれば，NaN は **NaN** と等しくなりませんし，利用可能ではない数値である無限大を表す Infinity は真であるということです．

▼リスト 3　==と===

```
if ( 1 == "1" ) {
  console.log("true"); // こちらが実行
}
else {
  console.log("false");
}

if ( 1 === "1" ) {
  console.log("true");
}
else {
  console.log("false"); // こちらが実行
}
```

2　構文

2.1　予約語

JavaScript には，表 1 に示すような予約語があります．驚くべきことではありますが，これらの予約語は予約されているだけで使うことができません．

2.2　==と===

JavaScript に限った話ではないのですが，2 つの値が等しいかどうかを調べる演算子に ‘==’ と ‘===’ があります．他言語における等値演算子と同じことがしたい場合は ‘===’ の方を使います．

では ‘==’ の方はというと，左辺と右辺で型が異なるときには，同じ型になるように変換した上で値を比較します．結構なおせっかいです（リスト 3）．

2.3　関数の引数

JavaScript の関数呼び出しにおいて，引数の数は一切考慮されません．明らかに引数が必要な関数に引数を 1 つも渡さなくても，過剰に引数を渡しても，一切エラーになりません．

表 1　JavaScript の予約語の一部
（MDN[*4] より引用）

class	enum	export	extends
implements	import	interface	let
package	private	protected	public
static	super	yield	

[*4] https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Reserved_Words

▼リスト 4　関数と引数

```
var sample = function(a, b, c) {
  console.log(a,b,c);
}

// 以下は全部有効
sample(1, 2, 3);
sample("a", "b");
sample(true);
sample();
sample(1, 2, 3, 4, 5, 6, 7, 8)
```

▼リスト 5　例外処理の概形

```
try {
  // 例外が発生するかも知れない処理
}
catch(e) {
  if (e.name === " ... ") {
    ...
  }
  ...
}
```

足りない分の引数には undefined が設定されます（リスト 4）．過剰な引数は argument という配列状の隠し変数を用いてアクセスすることができます．JavaScript ではいとも簡単に可変長引数をとる関数を作ることが可能です．

2.4　例外処理

JavaScript にも，例外処理があります．使い方も，C++ や Java などと同等です．

しかし，1 つの try につき 1 つの catch しか使えません．なので，全ての例外を一括して受け取るようにした後，if 文を使って例外の種類ごとの処理を書く必要があります（リスト 5）．

3　スコープ

3.1　変数のスコープ

見た目が C 言語や C++ に似ている JavaScript ですが，変数のスコープは全く異なっています．

C/C++ では，ブロックスコープと言って，ある変数はそれが宣言されたブロックとその内側のブロック内において有効です．しかし，JavaScript では関数スコープが採用されています．変数は宣言された関数とその関数内で定義された関数において有効です．つまり，ブロック内で宣言した変数がそのブロックの外からアクセスできてしまうのです（リスト 6）．幸いなことに，宣言を上書きすることはできますが，注意しておきましょう．

▼リスト 6　変数のスコープ

```
var scope = function(arg) {
  {
    var hoge = true;
    ...
  }

  for (var i = 0; i < 4; ++i) {
    ...
  }

  // C++だとhogeやiにアクセスできないが……
  console.log(hoge); // => true
  console.log(i); // => 4

  // 内側の関数から，変数にアクセス可能です
  var a = "foo";
  var fun = function() {
    console.log(a); // aは宣言してないが……
  };
  fun(); // => fooが出力される
}
```

▼リスト 7　this の値あれこれ

```
a = "foo"; // グローバル変数はvarが不要
var obj = {
  a : "hoge",
  fun1 : function(){
    console.log(this.a);
  }
};

var fun2 = function() {
  console.log(this.a);
};

var Fun3 = function() {
  console.log(this.a);
};

var obj2 = {
  a : "bar"
};

obj.fun1(); // => hoge
fun2(); // => foo
new Fun3(); // => undefined
obj.fun1.apply(obj2); // => bar
```

3.2　this のスコープ

JavaScript には this というキーワードがあります．this というキーワードは，使われる場所によって何を指すかが変わります（リスト 7）．

あるオブジェクトのメソッド内で使った場合，this はそのオブジェクト自身を指します（リスト 7 の fun1）．

これは，C++ や Java における this と同じ感じです．関数の中で this を使うと，グローバルなオブジェクトを指します（リスト 7 の fun2）．ブラウザなどでは window オブジェクトを指すようになります．コンストラクタとして呼び出した関数では，this は新しいオブジェクトを指します（リスト 7 の Fun3）．コンストラクタでは this に要素を追加して，最後に this を返すことで新しいオブジェクトを作ります．関数を **apply** や **call** と言ったメソッドを通じて呼び出すと，それらのメソッドに渡した引数を this として参照します（リスト 7 の obj.fun1.apply）．

このように，this の値はころころと変わってしまい，ときには予期せぬバグを引き起こします．例えば，コンストラクタとして用意した関数を間違って new を付けずに呼び出してしまうと，グローバルなオブジェクトに変数と関数を定義して，グローバルなオブジェクトを返してしまいます．そこで，このような間違いを減らすため，コンストラクタとする関数は名前を大文字で始めるという慣習が存在します．

また，あるオブジェクトのメソッドを DOM のイベントリスナーなどに，コールバック関数として登録するときには注意が必要です．this のやっかいな点は，関数を作ったときではなく，呼び出されるときに this の値が決定することです．コールバック関数を呼び出す関数は，コールバックに登録したメソッドの属するオブジェクトのメンバではないので，**this** の値がグローバルなオブジェクトを指すものとしてメソッドが実行されます（リスト 8）．this を用いてオブジェクト内のプロパティやメソッドにアクセスする処理を記述していると，ここでエラーが発生するか分かりにくいバグが発生します．しかも，その原因が this の値が知らないところで変わっていることによるものなので，分かりにくいことこの上ないです．この問題の対処方法はいくつかあり，bind を用いて this をそのメソッドが属するオブジェクトに固定した関数を作成し，それをコールバック関数として登録するものが一般的です．

似たような問題として，メソッド内で定義した関数を呼び出したときに，その関数内から this を通じてオブジェクトにアクセスできないというものもあります．これも，内部の関数がオブジェクトに（直接）属していないために発生します（リスト 9）．

4 その他

4.1 記号プログラミング

[b] jjencode[*5]というものがあります．これは JavaScript のコードを記号だけで記述するように変換します．どのように実現しているかは，参考文献 [2][3] を参照してください．リスト 10 は jjencode でエンコードしたコードです．

▼ リスト 8　this とコールバック

```
var obj = {
  uniqueValue : "x68",
  callback : function() {
    console.log(this.uniqueValue);
  }
};

// 上手くいかない例
setTimeout(obj.callback, 2000);

// 対処法
setTimeout(obj.callback.bind(obj), 2000);
```

▼ リスト 9　this と内部関数

```
var obj = {
  circle : "x68",
  outer : function() {
    var hoge = "foo bar";
    var inner = function() {
      console.log(hoge); // => foo bar
      console.log(this.circle); // =>
        undefined
    }
    inner();
  }
};

obj.outer();
```

[*5] http://utf-8.jp/public/jjencode.html

▼ リスト 10　jjencode

```
// console.log("x68"); を変換したコード
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,
   $_$_:(![]+"")[$],_$_:++$,$_$$:({}+"")[$],
   $$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],
   $__:++$,$_$:++$,$$__:({}+"")[$],$$_:++$,$$$
   :++$,$___:++$,$__$:++$};$.$_=($.$_=$+"")[$.
   $_$]+($._$=$.$_[$.__$])+($.$$=($.$+"")[$.
   __$])+((!$)+"")[$._$$]+($.__=$.$_[$.$$_])+(
   $.$=(!""+"")[$.__$])+($._=(!""+"")[$._$_])+
   $.$_[$.$_$]+$.__+$._$+$.$;$.$$=$.$+(!""+""
   )[$._$$]+$.__+$._+$.$+$.$$;$.$=($.___)[$.$_
   ][$.$_];$.$($.$($.$$+"\""+$.$$__+$._$+"\\"+
   $.__$+$.$_$+$.$$_+"\\"+$.__$+$.$$_+$._$$+$.
   _$+(![]+"")[$._$_]+$.$$$_+"."+(![]+"")[$.
   _$_]+$._$+"\\"+$.__$+$.$__+$.$$$+"(\\\"\\"+
   $.__$+$.$$$+$.___+$.$$_+$.$___+"\\\");"+"\"
   "())());
```

4.2　強制オープンソース

JavaScript は，半ば強制的にオープンソースとなります．先の記号プログラミングや，minify と呼ばれる最適・難読化によって，ある程度は実装を隠すことができます．

しかし，容易にデコードが可能です．ツールを使えば簡単に，元の実装と同等な，ある程度読みやすいコードに変換することができます．

そのため，他の Web サービスと連携するようなコードを JavaScript で記述するとき，そこに API キーなどが含まれていると，簡単に発見され解読されてしまいます．

まとめ

皆さんいかがでしたか．JavaScript のことが**好きになりましたか**．

この記事は参考文献 [1] の内容をもとに作成しました．この本を読む前から，ここに上げたような**かわいいポイント**のいくつかには気づいていました．しかし，この本を読むことによって，自分の知らなかった部分やさらに深い部分について知ることができました．この本は，JavaScript を使っていく上でとても役に立つアドバイスにあふれていますので，是非ともご一読されることをオススメします．

今回紹介した内容は，氷山の一角です．泣く泣く紹介することができなかったことがたくさんあります．プライベートなメソッドやプロパティをオブジェクトに追加する方法や parseInt の挙動など，興味がある方は是非とも調べてみて下さい．

それでは，JavaScript とより良い関わりを．

付録 A：JavaScript の実行環境

JavaScript を実行するには，昔は HTML のスケルトンコードに直接記述するか，別ファイルに記述したスクリプトを実行するようにしていました．

しかし，ここで挙げたコードのようなちょっとしたスクリプトを動かすのであれば，Google Chrome[*6] や Firefox[*7]をはじめとするモダンブラウザに搭載されている開発ツールを用いて実行するのがお手軽でオススメです．Chrome では「JavaScript コンソール」，Firefox では「Web コンソール」という名前で提供されています．どちらのブラウザでも，F12 キーを押すことで開くことができます．

なお，Internet Explorer[*8]でも，比較的新しめのバージョンならば利用可能なようです（筆者の環境に入っている IE11 にはありました）．

付録 B：代替言語

最近は，コンパイルすることで JavaScript を生成するような JavaScript の代替言語が数多く登場しています．有名なものでは，CoffeeScript[*9]や TypeScript[*10]，Google の Dart[*11] などが挙げられます．

これらの言語は，好みはあるでしょうが，より効率的に開発を進めるのに役立ちます．
是非とも，これら代替言語についても学ばれることをオススメします．

[*6] https://www.google.co.jp/chrome/browser/
[*7] https://www.mozilla.org/ja/firefox/new/
[*8] http://windows.microsoft.com/ja-jp/internet-explorer/download-ie
[*9] http://coffeescript.org/
[*10] http://www.typescriptlang.org/
[*11] https://www.dartlang.org/

付録 C：JSLint，JSHint

lint というソフトウェアをご存知でしょうか？これは，C 言語のソースコードを解析して，良くない記述をしている部分を指摘するソフトウェアです．これを用いることで，より綺麗で間違いの少ないソースコードを書くことができます．

JavaScript にも，これと同じように，良くない記述を指摘する JSLint[*12] というソフトウェアがあります．しかし，この JSLint は判定が厳しすぎたり，ソースコードの書き方を押し付けてくるような面があり，海外では不人気だったりします．そのため，間違いの原因となりうる部分だけを指摘するようにした JSHint[*13]が開発され，こちらの方がより広く使われているようです．

どちらのソフトウェアも，エラーチェックにとても役立つツールなので，JavaScript を直接書くような方は，事前にこれらのツールを用いて間違いをチェックすることをオススメします．

付録 D：minify

記述した JavaScript のサイズを小さくし，実装の内容を解析されにくくするために，コメントと不要なスペースや改行を削除し，変数名を a や b などの短いものに変換するなどの書き換えを行うことを minify と言います．

JavaScript ライブラリとして公開したり，自身のサイトや Web サービスなどで JavaScript を用いるときには，ダウンロードサイズを減らすために，積極的に minify することをオススメします．

minify を行うツールは Web アプリケーションとしてたくさん公開されています．もし，それらの Web アプリケーションを（ソースコードの流出などを防ぐために）使いたくない場合は，Google の Closure Compiler[*14]の利用をオススメします．

注意

この記事では JavaScript1.6 を対象としています．今後，多くのブラウザでより新しいバージョンのものが使用可能になった場合，ここで挙げた内容について，実際とは異なる状況になる場合があります．

謝辞

草稿の提出締め切りを当初の予定から 1 週間延ばしてもらった上，その締め切りからも 2 日ほど遅れてしまったにも関わらず，仏のような慈悲深さで受け取ってくれた会誌編集長にこの場をお借りして感謝の気持ちを表明します．

参考文献

[1] Douglas Crockford，“JavaScript: The Good Parts —「良いパーツ」によるベストプラクティス”，オライリー・ジャパン，2008

[2] “記号だけの JavaScript プログラミングの基本原理”，
http://perl-users.jp/articles/advent-calendar/2010/sym/3

[3] “記号だけの JavaScript プログラミングの原理 その 2 ”，
http://perl-users.jp/articles/advent-calendar/2010/sym/15

[*12] http://www.jslint.com/

[*13] http://www.jshint.com/

[*14] https://developers.google.com/closure/compiler/