報道各位

2020 年 6 月 4 日
株式会社インプレスR&D
https://nextpublishing.jp/

Swift と連携してカスタムスクリプト環境を提供！

# 『JavaScriptCore で遊ぼう』発行

## 技術の泉シリーズ、6 月の新刊

インプレスグループで電子出版事業を手がける株式会社インプレス R&D は、『JavaScriptCore で遊ぼう』(著者:熊谷友宏)を発行いたします。

最新の知見を発信する『技術の泉シリーズ』は、「技術書典」や「技術書同人誌博覧会」をはじめとした各種即売会や、勉強会・LT 会などで頒布された技術同人誌を底本とした商業書籍を刊行し、技術同人誌の普及と発展に貢献することを目指します。

**『JavaScriptCoreで遊ぼう』**
https://nextpublishing.jp/isbn/9784844378839


著者:熊谷 友宏
小売希望価格:電子書籍版 1600 円(税別)／印刷書籍版 2000 円(税別)
電子書籍版フォーマット:EPUB3／Kindle Format8
印刷書籍版仕様:B5 判／カラー／本文 114 ページ
ISBN:978-4-8443-7883-9
発行:インプレス R&D

**＜＜発行主旨・内容紹介＞＞**

iOS や macOS アプリの SDK には、JavaScriptCore フレームワークという、アプリに JavaScript を処理する機能を簡単に組み込めるライブラリーが標準搭載されています。これを使うと Swift ネイティブコードと JavaScript コードとを密接に連携して、自由度の高いプログラミングが可能になります。

本書では JavaScriptCore フレームワークの基礎から、JavaScript と Swift の連携についての詳細まで、徹底的に解説します。

(本書は、次世代出版メソッド「NextPublishing」を使用し、出版されています。)

## JavaScriptCore の基本的な使い方を解説

# 2. 基本的な使い方

それでは、JavaScriptCore を Swift で使う方法について見ていきます。この章をひととおり眺めればJavaScriptCore の使い方が掴めるはずです。

### 2.1. 実装の流れ

まずはJavaScriptCore の雰囲気をざっくり掴むことから始めましょう。JavaScriptCore を使うときの基本的な流れは、次のようになります。

1. 最初にJavaScriptCore コンテキストを生成する。
2. コンテキスト上でJavaScript コードを実行する。
3. コンテキストから実行結果を取得する。

JavaScriptCore では、これらの操作をとても手軽に行えます。もう少し詳しく分けて整理してみましょう。

### 2.2. JavaScriptCoreを使えるようにする

iOS アプリや macOS アプリであれば、Swift でJavaScriptCore を使えるようにするのは簡単です。JavaScriptCore を使いたいソースファイル内に次の 1 行を記載するだけで、JavaScriptCore を利用する準備が整います。

```
import JavaScriptCore
```

JavaScriptCore モジュールは、macOS アプリや iOS アプリの SDK[1] に標準で組み込まれています。そのため、複雑な準備をしなくても、すぐに JavaScript 実行環境をアプリ内に作ることが可能です。

### 2.3. コンテキストを生成する

JavaScriptCore モジュールをインポートしたら、`JSContext` クラスをインスタンス化します。これだけで JavaScript 仮想マシンが用意され、このコンテキストを通して JavaScript コードを実行できます。

#### 2.3.1. IUO 属性の扱いについて

このとき、コンテキストは `JSContext!` 型、つまり **IUO 属性付きのオプショナル型**で取得できるところに注意しながら扱うようにします。

```
let context = JSContext()!
```

戻り値はオプショナル型ですが、普通は `nil` が得られることはなさそうなので、コンテキストの取得時に強制アンラップしておくと扱いやすいと思います。このように速やかに強制アンラップすることで、もしコンテキストが取得できなかった場合でも**この行で強制終了してくれる**ので、後になってから問題が発覚する可能性が少なくなります。

#### 2.3.2. 速やかに nil を解消する

ここでアンラップしなかった場合、コンテキストを格納した変数は IUO 属性が外れて `JSContext?` 型になります。これによって明示的なアンラップが必要になり、そのおかげで、それ以降は `nil` をもれなく考慮したコードが書けるようになります。

コンテキストを**取得できなかったかもしれない状態のまま、処理を続行し続けないといけないことはほとんどない**と思うので、何らかの方法で速やかに「取得できなかったかもしれない」状況を解消するようにします。

#### 2.3.3. 強制アンラップだと都合が悪いとき

強制アンラップでアプリが落ちることが致命的になる場合は、次のように、別の方法でエラーを外側に伝えると良いかもしれません。たとえば、エラーハンドリングを使うことで呼び出し元にエラーを通知することができます。

```
guard let context = JSContext() else {

    throw NSError(domain: "MyError", code: 1)
}
```

### 2.4. スクリプトを実行する

コンテキストを取得できたら、それに対して JavaScript コードを文字列で添え `evaluateScript` を実行すれば、そのコードを実行できます。

```
context.evaluateScript("var v1 = 10")
context.evaluateScript("var v2 = v1 + 10")
```

実行結果は**コンテキストに蓄積されていく**ので、前に実行した結果を踏まえてさらに次の処理を進めていくことが可能です。

[1] Software Development Kit ― アプリを開発するためのプログラムモジュール郡です。





## 変数の扱い方や扱う際の注意点も紹介

# 3. 変数の扱い方

基本的な流れを押さえたところで、続いて JavaScript 変数を Swift で扱う方法について見ていきます。

Swift 側と JavaScript 側とでふるまいが異なることがある[1]ので、そのあたりは注意しないといけませんが、普通に使う分にはとても自然に扱えるようになっています。

### 3.1. 変数を表現する型

JavaScriptCore を使うと、JavaScript 側の値を Swift 側では `JSValue` 型で扱うことになります。この型はいわゆる**バリアント型**で、JavaScript で扱う全ての値がこのひとつの型に収められます。

```
let n: JSValue = context.objectForKeyedSubscript("n")!
```

なお、この値を表現する `JSValue` 型は Objective-C 由来の型であるためか、Swift でも構造体ではなくクラスで定義されています。

### 3.2. JSValue型から値を取得する

JavaScript を実行して得られた値は `JSValue` 型で得られます。そのため、その値を Swift で使うときには、値を Swift の型として取り出す必要があります。

具体的には、次の `JSValue` 型が備えている **Swift 型に変換するためのメソッド**を使って、ネイティブな型に変換します。

| | |
|---|---|
| `toDouble()` | `Double` 型に変換 |
| `toString()` | `String!` 型に変換 |
| `toBool()` | `Bool` 型に変換 |
| `toObject()` | `Any!` 型に変換 |
| `toObjectOf(_:)` | `AnyClass!` 型を持つ `Any!` 型に変換 |
| `toNumber()` | `NSNumber!` 型に変換 |
| `toInt32()` | `Int32` 型に変換 |
| `toUInt32()` | `UInt32` 型に変換 |
| `toDate()` | `Date!` 型に変換 |
| `toArray()` | `[Any]!` 型に変換 |
| `toPoint()` | `NSPoint` 型に変換 |
| `toRect()` | `NSRect` 型に変換 |
| `toSize()` | `NSSize` 型に変換 |
| `toDictionary()` | `[AnyHashable:Any]!` 型に変換 |

#### 3.2.1. 値を数値として扱いたいとき

たとえば、JavaScript 側に用意した変数 `value` の値を、Swift 側で `Int32` 型として使いたいときには、次のように `toInt32()` メソッドを使います。

```
let n: Int32 = context.objectForKeyedSubscript("n")!.toInt32()
```

ところで、Swift では数値を `Int` 型で表すのが一般的です。ただし `JSValue` 型を直接的に Swift の数値型へ変換できるメソッドは `toInt32()` と `toUInt32()` だけしかないので、値を `Int` 型として扱いたいときには、いったん JavaScript における数値の内部表現である `Double` 型に変換してから、Swift 側で `Int` 型に再変換するようにします。

```
let n: Int = Int(context.objectForKeyedSubscript("n")!.toDouble())
```

#### 3.2.2. 値を文字列として扱いたいとき

文字列型の扱いは簡単です。`JSValue` 型に用意されている `toString()` メソッドを使うことで Swift の `String` 型に変換できます。

```
let n: String = context.objectForKeyedSubscript("n")!.toString()
```

ただし、変数に `null` や `undefined` といった値が入れられているときには、JavaScript のときと同じ変換が行われて `"null"` や `"undefined"` といった文字列が得られるため注意が必要です。

### 3.3. どの型に変換できるかを調べる

このように、JavaScript には暗黙変換の文化があります。たいていの場面で気軽に値を扱える反面、うっかりすると思いがけない値が得られる場合があります。そのため、扱っている値がどんな値なのかを知った上で処理する必要が出てくるときがあります。

`JSValue` 型で扱っている値の種類を知りたいときは、`JSValue` 型に用意されている `isNumber` などの判定用のプロパティーを使います。

| | |
|---|---|
| `isNumber` | `Number` 型であるかを判定 |
| `isString` | `String` 型であるかを判定 |
| `isBoolean` | `Boolean` 型であるかを判定 |

[1] Swift 側と JavaScript 側とでのふるまいの違いについては、第 5 章で詳しく紹介します。

## コードを書きやすくするためのポイントを紹介

### 9.2. CGSize型などを安全に扱う

JavaScriptCoreでは、Swiftで使える値型のうち、CGSize型とCGRect型、CGPoint型、NSRange型の4つはJavaScript側でも使えるようになっています。ただ、Swift 3.0の仕様変更に伴って、うっかりすると**予期しない動作の原因になるかもしれない**ことは第6章で紹介しました。

具体的には、これらの型をJavaScriptに渡すときにはJSValue型にラップしてから渡さないといけないのですが、ラップしなくても渡すことだけはできてしまいます。そうなると「CGSize型の値をJavaScript側で取得できないことがある」という問題に悩まされてしまうかもしれません。

たとえば、次のようにちょっとした違いで動作が大きく変わってきます。

```
let size = CGSize(width: 5.5, height: 10.0)

context.setObject(size,
                  forKeyedSubscript: "size1" as NSString)

context.setObject(JSValue(size: size, in: context)!,
                  forKeyedSubscript: "size2" as NSString)

context.evaluateScript("size1.width")!       // undefined
context.evaluateScript("size2.width")!       // 5.5
```

これらの値型をJavaScript側で頻繁に使う場合は、**JSContextクラスを拡張して、CGSizeなどの値型を自動でJSValue型にラップする**ようにすると便利になるかもしれません。

たとえばCGSize型を自動的にラップするメソッドは次のようになります。

```
extension JSContext {

    @nonobjc
    func setObject(_ size: CGSize, forKeyedSubscript
                key: (NSCopying & NSObjectProtocol)!) {

        setObject(JSValue(size: size, in: self)!, forKeyedSubscript: key)
    }
}
```

同じようにしてCGRect型、CGPoint型、NSRange型についても実装すれば、これらの型をJSValue型にラップし忘れることがなくなります。

ちなみにここでメソッドに@nonobjcを付けているのは、setObject(_:forKeyedSubscript:)という署名のメソッドがObjective-C側に既に存在しているためです。Objective-CはSwiftと違ってオーバーロードができないため、今回の新たに追加するメソッドは、Objective-C側にはエクスポートしないようにしています。

こうすることで、このメソッドはObjective-C側では使えなくなりますが、Objective-C側ではそもそもid型にCGSize型は渡せないため、間違ってCGSize型をObjective-C側に最初から存在していたメソッドに渡してしまう心配はなくなるはずです。

### 9.3. キーを文字列リテラルで指定する

JSContextクラスのsetObject(_:forKeyedSubscript:)メソッドを使うとき、キー名に文字列リテラルを使えないようになっていました。これはSwift 3.0以降のObjective-C相互運用の機能が、Objective-Cでいう文字列の添字を(NSCopying & NSObjectProtocol)型として解釈することが影響しているようです。

文字列リテラルの後にas NSStringを付与してブリッジすれば済むのでそれほど煩わしくはないのですが、たとえばJavaScriptCoreを扱うライブラリーを作っていたりして、文字列リテラルをほとんどJavaScriptCoreのためだけに使うような極端な場合には少し面倒かもしれません。

そんなときには**文字列リテラルの既定の型を変更する**ことで、ブリッジせずにそのまま文字列リテラルを渡せるようになります。

文字列リテラルの型を変更するには、**型エイリアスStringLiteralTypeをNSStringに書き換え**ます。

```
typealias StringLiteralType = NSString

func prepare(in context: JSContext) {

    // 文字列リテラルの既定の型が NSString として解釈される
    context.setObject(200, forKeyedSubscript: "status")
}
```

#### 9.3.1. 既定の型の影響範囲

文字列リテラルの既定の型を変更するとき、アクセスレベルを指定しないでStringLiteralTypeを変更すると、それは**モジュール全体に影響**します。**影響範囲をファイル内にとどめたいとき**は、privateを明記します。

また、StringLiteralTypeを変更して文字列リテラルの動きを変更できるのは**モジュールのグローバルスコープに記載したときだけ**で、関数などのローカルスコープに記載しても効果がありません。そのため、ファイル全体よりも狭い範囲に限って既定の型を変更するということはできません。

アクセスレベルにpublicを指定することもできますが、それを定義したモジュールを取り込んだ先には影響を及ぼしません。たとえばMyJSCoreExtModuleというモジュールでStringLiteralTypeをpublic指定で変更したとしても、それを取り込んだ先で"key" as MyJSCoreExtModule.StringLiteralTypeのように明記しない限りは、文字列リテラルをその型で





## ＜＜目次＞＞



## ＜＜著者紹介＞＞

熊谷 友宏

幼少に MSX-BASIC と出会って以来、プログラミングと戯れる日々を送る。いつしかプログラミングの勉強会と巡り合わせ、楽しさを分かち合える場所の存在に気付く。そんな勉強会で数年間に渡って登壇を重ね、「横浜 iPhone 開発者勉強会」の主催を引き継ぎ、後に念願だった地元での勉強会「カジュアル Swift 勉強会」を開催。さらには勉強会の楽しさを首都圏以外に届けるべく、これまで7都道府県以上で「みんなで Swift 復習会 GO!」を開催する。

そんな勉強会活動の折、技術商業誌を執筆する縁に恵まれ、それを起点に技術同人誌にシフトして今に至る。

その道程でたまたま文化の交差した、アイドルユニット「Pyxis」と「さくらシンデレラ」が見せる世界をきっかけに『勉強会もエンターテインメントなのではないか』、そんな思いを募らせながら模索している。

著書に「Xcode 5 徹底解説」(秀和システム)、「Swift らしい表現を目指そう」、「Swift イニシャライザー大全」、「プログラマーのための新千歳空港入門」、「iOSCon in London 入門」などがある。

＜＜販売ストア＞＞

電子書籍:

Amazon Kindle ストア、楽天 kobo イーブックストア、Apple Books、紀伊國屋書店 Kinoppy、Google Play Store、honto 電子書籍ストア、Sony Reader Store、BookLive!、BOOK☆WALKER

印刷書籍:

Amazon.co.jp、三省堂書店オンデマンド、honto ネットストア、楽天ブックス

※　各ストアでの販売は準備が整いしだい開始されます。

※　全国の一般書店からもご注文いただけます。

【インプレス R&D】 https://nextpublishing.jp/

株式会社インプレス R&D（本社：東京都千代田区、代表取締役社長：井芹昌信）は、デジタルファーストの次世代型電子出版プラットフォーム「NextPublishing」を運営する企業です。また自らも、NextPublishingを使った「インターネット白書」の出版など IT 関連メディア事業を展開しています。

※NextPublishing は、インプレス R&D が開発した電子出版プラットフォーム（またはメソッド）の名称です。電子書籍と印刷書籍の同時制作、プリント・オンデマンド（POD）による品切れ解消などの伝統的出版の課題を解決しています。これにより、伝統的出版では経済的に困難な多品種少部数の出版を可能にし、優秀な個人や組織が持つ多様な知の流通を目指しています。

【インプレスグループ】 https://www.impressholdings.com/

株式会社インプレスホールディングス（本社：東京都千代田区、代表取締役：唐島夏生、証券コード：東証 1 部 9479）を持株会社とするメディアグループ。「IT」「音楽」「デザイン」「山岳・自然」「旅・鉄道」「学術・理工学」を主要テーマに専門性の高いメデイア&サービスおよびソリューション事業を展開しています。さらに、コンテンツビジネスのプラットフォーム開発・運営も手がけています。

【お問い合わせ先】

株式会社インプレス R&D NextPublishing センター
TEL 03-6837-4820
電子メール: np-info@impress.co.jp