iOSアプリ
モダン開発入門

髙橋 凌 [著]

Swift5.0 + Xcode10.2 対応!
もう一歩先へ進みたいiOSアプリ開発初心者向け！

Xcodeテクニックから便利ライブラリ、Firebase、テスト、CI/CD等
モダンな開発環境周辺を解説！

iOS
アプリ開発
がんばるぞ!!

iOS アプリ モダン開発入門

k0uhashi　著

2019-04-14 版　　iOS アプリ開発がんばるぞ！！ の会 発行

はじめに

こんにちは、@k0uhashi です。本書を手にとって頂き、誠にありがとうございます。

本書では、現場で使われているモダンな環境が知りたい！ という iOS アプリ開発初心者に向けて、よく使われるツールやライブラリ、サービスをまとめてわかりやすく解説していきます。少しでも読んでいる方の開発の手助けになれたら嬉しいです。

対象読者

本書では、次の読者を対象としています。

- 簡単な iOS アプリなら開発できる
  - カウンターアプリ、おみくじアプリ、何らかの API を取得して表示する等
- もっと便利なサービスやライブラリを使って、開発効率を上げたいと思っている

推奨知識

本書を読むにあたって知っておくとより理解が進む概念について挙げていきます。

- ライブラリ
- フレームワーク
- ライブラリとフレームワークの違い
- SaaS, PaaS, mBaaS とは？

想定環境

本書では、以下の環境をターゲットとしています。

- Xcode10.2 もしくは 10.1
- Swift5.0 もしくは 4.2

- Ruby 2.3.7p456

基本 Swift5.0 を基に解説しますが、執筆時点で対応していないライブラリもあるので、一部 Swift4.2 で動作させている部分があります。

本書に関するお問い合わせ

- twitter: https://twitter.com/k0uhashi

免責事項

本書は有志によって作成されているもので、米 Apple 社とは一切関係がありません。また、掲載されている内容は情報の提供のみを目的にしていて、本書を用いた開発・運用は必ず読者の責任と判断によって行ってください。著者は本書の内容による開発、運用の結果によっての結果について、いかなる責任も負いません。

目次

第 1 章

iOS アプリ開発において読んでおくべきもの

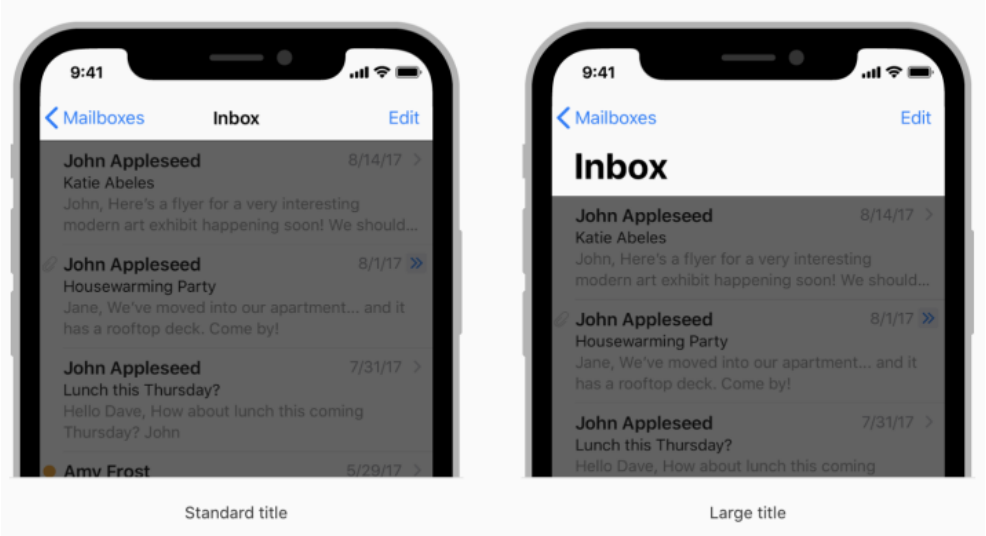
本章では、これからの iOS アプリ開発において、また筆者のこれまでの iOS アプリ開発をしてきた中で読んでおくべきだと**強く感じた**リファレンスや書物、サイトを紹介していきます。

1.1 HumanInterfaceGuidelines

- Human Interface Guidelines
- https://developer.apple.com/design/human-interface-guidelines/
- 以下、HIG

Apple による UI 設計の原則を定めたガイドライン。iOS、macOS、watchOS、tvOS アプリ開発における UI の原則や思想等が書かれています。

例えば、NavigationController の TitleBar に LargeTitles を使うのはどういうときか？（Clock アプリのようなタブ付きのわかりやすいレイアウトがある場合はいらない）とか



▲図 1.1　TitleBar の扱い

ステータスバー、NavigationBar はどういうときに隠すのか？（ex. ユーザーに没入感をもたせたい時）等、なぜその UI がそうなのか？ その UI はこういうときに使うといったガイドラインが詳しく書かれています。

▲図 1.2　ステータスバーの扱い

作ったアプリが「なんかイマイチな UI だな〜、使い心地良くないな〜・・・」と感じるなら、一度見てみましょう。

1.2　iOS アプリ開発ベストプラクティス

- iOS-factor (https://ios-factor.com/ja/)
- リポジトリ (https://github.com/ios-factor/ios-factor.com)

これまでの iOS アプリ開発プラクティスの歴史や現代の iOS アプリ開発の開発環境の設定やデプロイ、開発における考え方（すべてのユーザーが最新バージョンにアップデートすることを前提にしない...）など現代の iOS アプリ開発者にとってとても役に立つプラクティス集です。オープンソースプロジェクトで有志によって更新されています。

例えば、次のようなことが書かれています。

> V. Prefer local over remote できるだけバックエンドなしで動作するように iOS アプリをスマートに保つ
> -- （中略)

さまざまな理由でできるだけ多くのビジネスロジックと計算をデバイス上で実行すべきです。
プライバシー：リモートサーバーにデータを送信しないようにするスピード：サーバーへのデータ送信やレスポンス待ちには時間がかかり失敗するかもしれない（不安定な Wi-Fi など) データ利用：ユーザーには毎月のデータ制限があることが多いスケーリング：アプリが流行った場合は、バックエンドサービスをスケールアップする責任があるバッテリ寿命：モバイルデータを使うとバッテリを浪費する信頼性：まだ信頼性の低い LTE/3G 回線を持つ国がある

1.3 Swift 実践入門 —— 直感的な文法と安全性を兼ね備えた言語 (WEB+DB PRESS plus)

- 技術評論社より出版

Swift とはなんぞや？ というところから、言語仕様解説、実際の業務でどう使うか？まで解説していて、初心者〜上級者までこれ 1 冊でいけるといっても過言ではないくらい良い本です。必須。実際、著者（僕）も iOS アプリ開発に入門するときに一番最初に購入していて、今でも時々参照しています。

第 2 章

Xcode 超活用術

本章では、Xcode の便利ショートカットから便利なデバッグの方法、コードスニペット等についてを紹介しています。これらの知識があるとなしの場合ではかなり開発効率が違ってくるので、ぜひ覚えてみましょう！

2.1 ショートカット

Xcode で iOS アプリを効率的に開発する上で習得必須な便利ショートカットをここで紹介していきます。

2.1.1 Build 周り

- Command + B
  - ビルド
- Command + R
  - ビルド後にアプリを実行
- Command + Control + R
  - 直近でビルドした成果物を利用してアプリを実行
  - 開発中にアプリがクラッシュしたときや何らかの影響でアプリが落ちた時に、デバッガを有効にした状態で起動したい場合に便利！
- Command + U
  - テスト実行
- Command + Control + R
  - 直近でビルドした成果物を利用してテスト実行
- Command + . (ドット)

  - 実行中のビルドやアプリを終了
- Command + K
  - クリーン
- Command + \
  - ブレークポイントの追加や削除
- Command + Control + Y
  - ブレークポイントのところから再開

2.1.2 検索周り

- Command + F
  - エディタ内の検索
- Command + Option + F
  - エディタ内の置換
- Command + Shift + F
  - プロジェクト内の検索
- Command + Shift + Option + F
  - プロジェクト内の置換
- Command + Control + Y
  - ファイルの検索
- Command + Shift + O（オー）
  - プロジェクト内のファイル名やメソッドの検索
  - これを覚えるとでかい！

2.1.3 コードエディタ周り

- Command + Shift + J
  - プロジェクトナビゲータで現在編集しているファイルの位置をフォーカスさせる
- Option + Click
  - 軽くリファレンスを表示

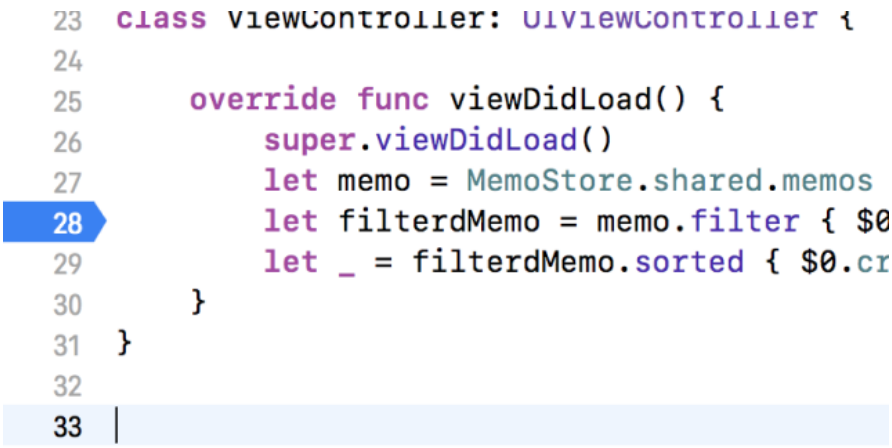
2.1.4 シミュレータ

- Command + S

  - スクリーンショットの撮影
- Command + K
  - キーボードの表示、非表示の切り替え
- Command + Shift + H
  - ホーム画面に戻る

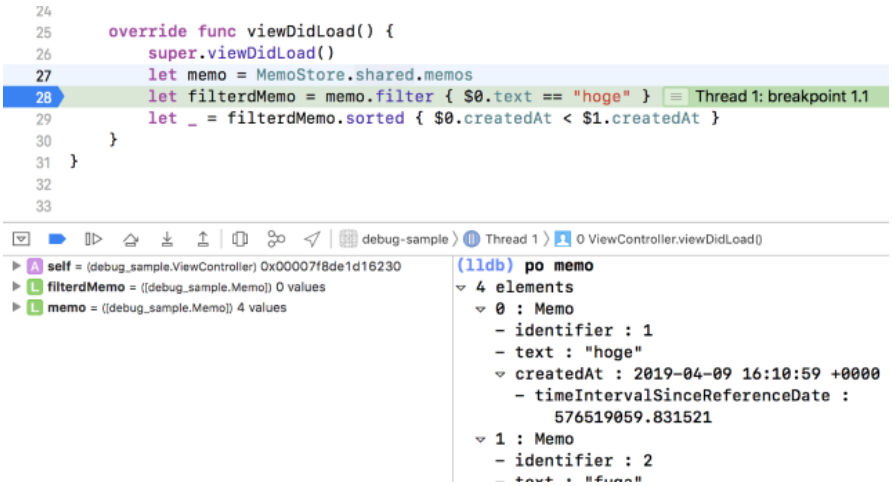
2.2 デバッグ術

Xcode では、デバッグするときに print 文を使って値を追っていく方法の他に、ソースコードエディタの行数の部分をクリックして、**ブレークポイント**を設定して値を方法もあります。（外すときは青い矢印を外にドラッグ）



▲図 2.1　ブレークポイントの設定

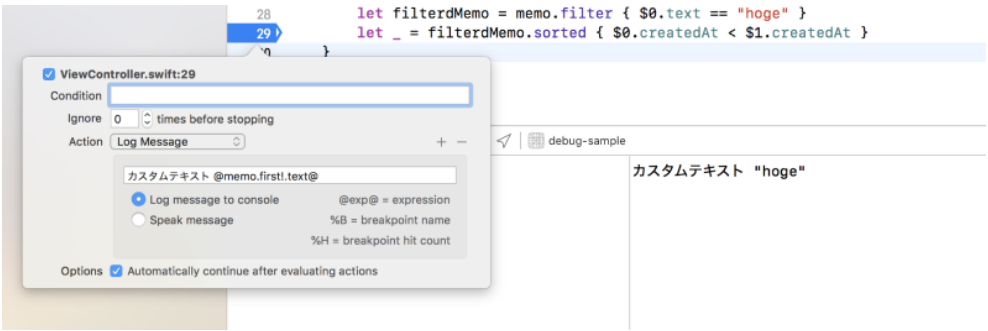
Xcode でビルド＆実行中にこの設定したブレークポイントの部分の処理が実行されると、そこで一時停止ができ変数の中身を確認できます。（右下で `po 変数名` コマンドを入力すると同じように確認できます。



▲図 2.2　ブレークポイントで一時停止

しかし、for 文の中の値を見たい場合繰り返されるたびに一時停止して値を確認するのは面倒です。Xcode にはブレークポイントで一時停止させずに変数の結果だけ出力させる方法もあるのでそれを使ってみます。

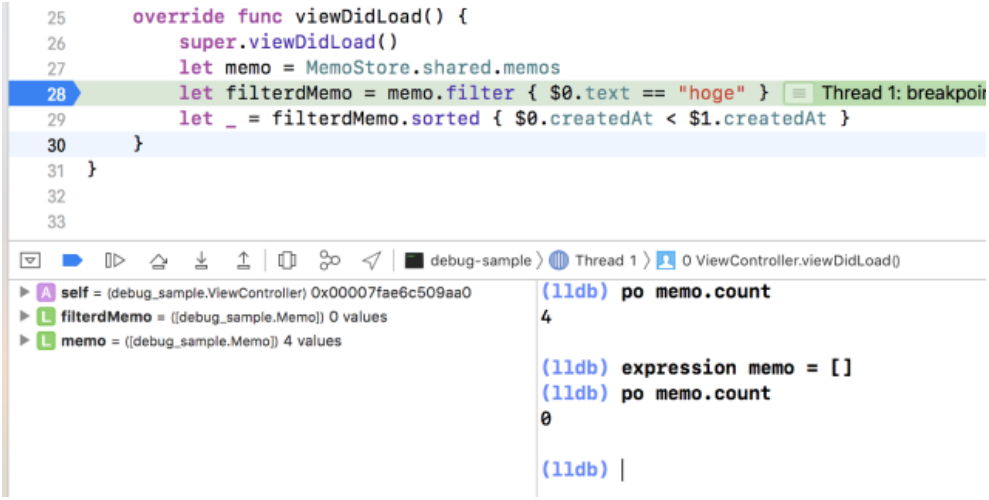
次のように設定してみてください



▲図 2.3　ブレークポイントの詳細設定

expression

また、ブレークポイントで一時停止させている時に変数へ値を代入できます。一時停止したら (lldb) のところで `expression 変数 = 新しい変数`とやってみてください



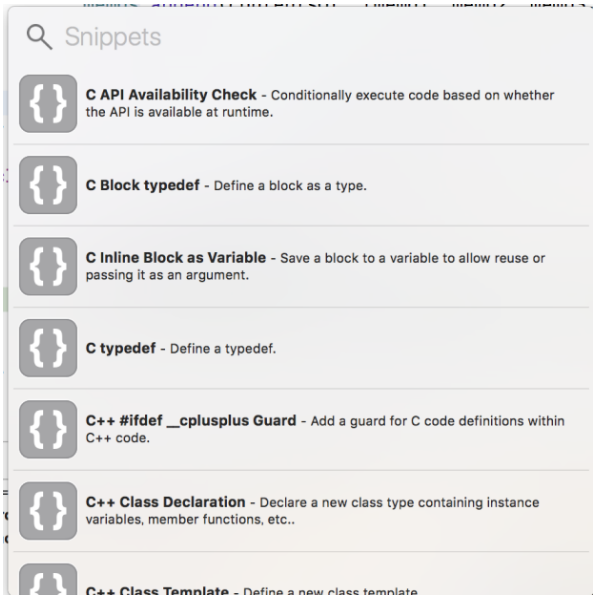
▲図 2.4　expression の使用

let で定義されている変数でも代入できました！

2.3　魔法のコードスニペット

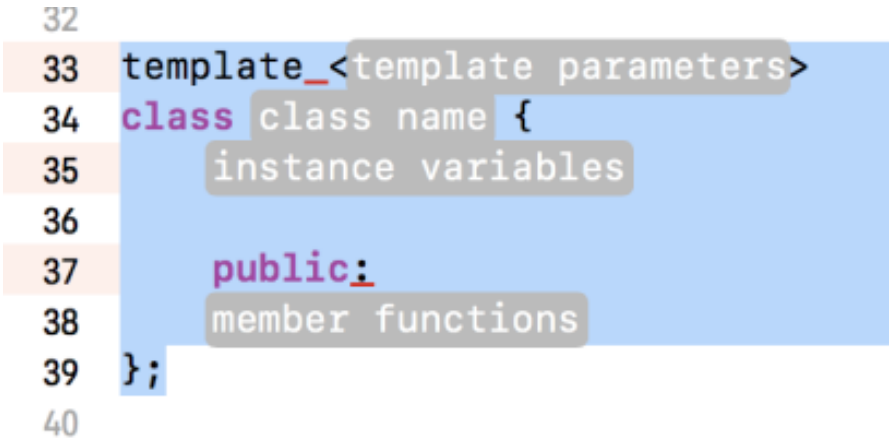
Xcode には、似たコードをいちいち書かなくても良いように、**Code Snippet** という機能が搭載されています。試しに、Xcode 上で **Command** + **Shift** + **L** を押して呼び出してみてください。

何も設定を変更していなければ、次の画面がでてきます。



▲図 2.5　Code Snippet

それを Enter かエディタ上へドラッグアンドドロップすると、指定したスニペットがエディタ上に貼り付けられます。



▲図 2.6　サンプル

ここの灰色の部分は<# #>で文字を囲むと作れます。

新しく snippet を作る場合は、上部メニューバーの Editor > Create Code Snippet から作成します。ちなみに、筆者は次のようなものを作って使っています。

▼

```
// テスト用のテンプレ（後に紹介する Quick というテストフレームワーク用）
import XCTest
import Nimble
import Quick
@testable import <# Product Name #>

class <# test class name ex. LocalNotificationTests #>: QuickSpec {

    override func spec() {

        beforeEach {
        }

        describe("<# test title ex. Test Location Manager #>") {

            <# local variables #>

            context("<# context name ex. Append Notification #>", {
                it("<# it name ex. Simple Append #>", closure: {

                })
            })
        }
    }
}
```

▼

```
// R.swift を用いて CustomView を作る時
import UIKit

class <# View Class Name #>: UIView {

    override init(frame: CGRect) {
        super.init(frame: frame)
        commonInit()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        commonInit()
    }

    private func commonInit() {
        if let view = R.nib.<# View Class Name #>.firstView(owner: self) {
            view.frame = self.bounds
            view.autoresizingMask = [.flexibleHeight, .flexibleWidth]
            self.addSubview(view)
        }
```

```
    }
}
```

▼

```
@IBOutlet private weak var <# variable name #>: <# class name #>
```

第 3 章

ライブラリ管理ツールと便利ライブラリ

3.1 CocoaPods

Ruby で作られている iOS/MacOS 向けのライブラリ管理ツール。オープンソースプロジェクトであり、GitHub 上で有志によって開発されている

- CocoaPods
- https://github.com/CocoaPods/CocoaPods

GitHub からライブラリをいちいちダウンロードして導入しなくても、これを使うことによって自動でライブラリ間の依存を解決しつつ、ライブラリをプロジェクトに簡単に導入できます。

3.1.1 導入

前提として、PC に Ruby がインストールされている必要がありますが、MacOS ではデフォルトで導入されているので、今回は Ruby のインストールは省略します。

CocoaPods を PC に導入するには terminal を開いて、次のコマンドを実行して下さい。

```
sudo gem install cocoapods
pod setup
```

その後、terminal で Xcode プロジェクトの階層へ移動し、次のコマンドを実行

```
cd YOUR_PROJECT_DIR
pod init
```

これでライブラリ管理する準備は完了！ あとは生成された **Podfile** ファイルにどのライブラリをインストールするか記述していきます。

例えば、後に本書で紹介する PKHUD (https://github.com/pkluz/PKHUD) というライブラリを導入したい場合、Podfile を次のように編集します。

▼

```
source 'https://github.com/CocoaPods/Specs.git'

target 'YOUR_PROJECT_TARGET_NAME' do
  use_frameworks!

  pod 'PKHUD'
end
```

その後、次のコマンドでライブラリの更新とインストールを行います。

▼

```
pod update
```

このコマンドを打つと、インストールするライブラリのバージョンと、それをどこからとってくるか・・・等が **Podfile.lock** に記録されて出力されます。
pod install コマンドを実行すると、この **Podfile.lock** の中身を参照して、書かれているバージョンにあわせてライブラリを自動でインストールします。

使い分けの例

- **pod install**
  - 新たにチームメンバーが増えたときに環境を揃えられる
  - プロジェクトを **git clone** して **pod install**
- **pod update**
  - ライブラリのバージョンをアップデートしたい

3.2 Carthage

前節で説明したものと同じようなライブラリ管理ツールとして Carthage（カーセッジ、カルタゴ）があります。こちらは Swift で作られていて、同じようにオープンソースプロジェクトとして GitHub で公開されています。

CocoaPods との違いを**大まかに**すると、CocoaPods でライブラリ管理をするより、Carthage でライブラリ管理をしたほうが**コンパイル時間が短縮できる**というメリットがあります。というのも、CocoaPods はビルド時に毎回ライブラリもビルドしますが、Carthage はライブラリの導入時にしかビルドしません。なのでその分ビルド時間が速くなります。（※ CocoaPods でも一応毎回ビルドしないように設定はできる）
ただし、導入が少し難しかったり、一部のライブラリは対応していなかったりするので CocoaPods と共存することが多いです。

ライブラリ導入時に変にハマる可能性があるので、CocoaPods でライブラリ管理をするのが最初は良さそうです。

3.3 R.swift - リソースをタイプセーフに扱う

- https://github.com/mac-cain13/R.swift
- プロジェクトにインポートした画像や ViewController の nib 名を安全に取得できるようになるライブラリ

例えば、いま開発中のプロジェクト内で画像や ViewController.xib を扱うときに次のような書き方をしている場合

▼

```
let icon = UIImage(named: "settings-icon")
let font = UIFont(name: "San Francisco", size: 42)
let color = UIColor(named: "indictator highlight")
let viewController = CustomViewController(nibName: "CustomView", bundle: nil)
let string = String(format: NSLocalizedString("welcome.withName", comment: ""),
      locale: NSLocale.current, "Arthur Dent")
```

R.swift を使うことで、次のようにタイプセーフに扱うことができるようになります。

▼

```
let icon = R.image.settingsIcon()
let font = R.font.sanFrancisco(size: 42)
let color = R.color.indicatorHighlight()
let viewController = CustomViewController(nib: R.nib.customView)
let string = R.string.localizable.welcomeWithName("Arthur Dent")
```

ライブラリの導入に少し手順が必要ですが、導入するとタイプセーフにリソースを扱えるようになり、typo を減らせてとても便利です！

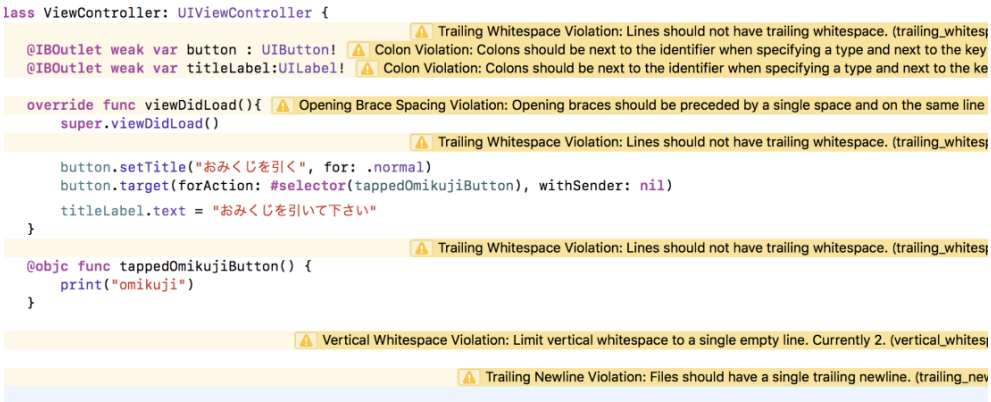
- インストール詳細はコチラ
- (https://github.com/mac-cain13/R.swift#cocoapods-recommended)

3.4 SwiftLint - 静的解析ツール

- オープンソースの静的解析ツール
- https://github.com/realm/SwiftLint

Github の Swift Style Guide (https://github.com/github/swift-style-guide) に基づいて Swift のコードスタイルをフォーマットしてくれるツールです。

例えば、次のようにコードスタイルガイドに反したコードを書いたときに自動的に指摘をしてくれます。



▲図 3.1　SwiftLint を実行した時の指摘

チーム開発では必須ですね！

デフォルトだと、自分のプロダクトのコード以外にも CocoaPod で導入したライブラリのコードに対しても Lint をかけてしまうので、プロジェクトの配下に**.swiftlint.yml** という名前で設定ファイルを作り、中身を次のように書き換えて下さい。

▼

```
# lint 対象とするディレクトリ
included:
  - /YOUR/PROJECT/SOURCE/PATH
# デフォルトの lint 設定は、force cast はコンパイルエラーレベルにしているが、warning（黄色
いやつ）レベルまでに変更したいときにはこう書く
force_cast: warning
# 上と同じような感じ
force_try:
  severity: warning
# 変数名の lint の除外パターン指定
```

```
variable_name:
  excluded:
    - e
    - id
# 1つの関数の行数制限
# 80 ー 100 までは warning(黄色)
# 100 以上はコンパイルエラー
function_body_length:
  - 80
  - 100
```

3.5 SwiftDate - 日付操作ライブラリ

- オープンソースの日付パース、フォーマット、カスタム日付作成が Swift で簡単にできるようになるライブラリ
- https://github.com/malcommac/SwiftDate

次のように扱えます。

▼

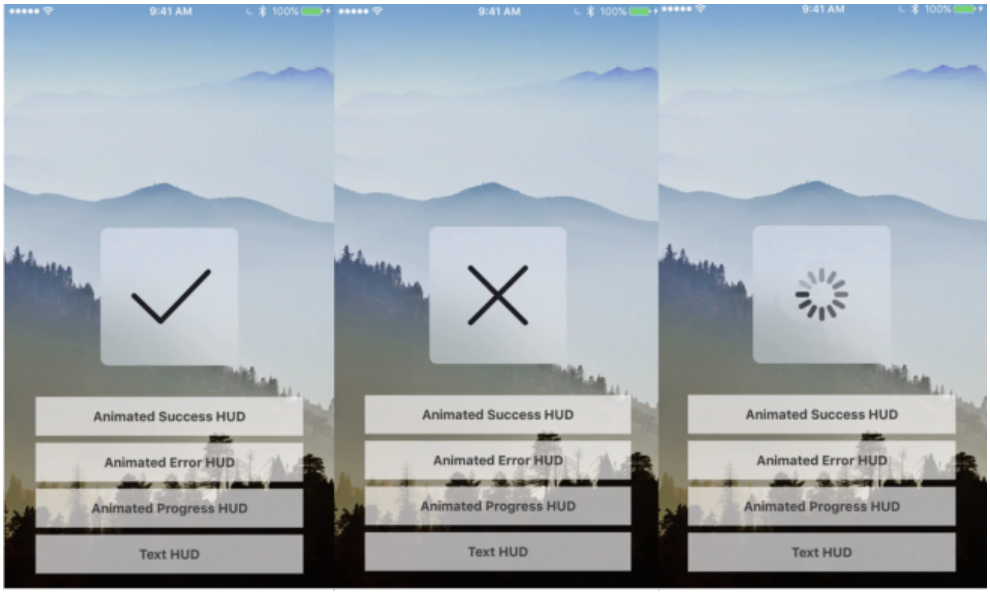
```
// 例えば、このようにテキストから日付に設定できたり
let _ = "2010-05-20 15:30".toDate("yyyy-MM-dd HH:mm")
// 次のように日付計算できたり
let _ = ("2010-05-20 15:30:00".toDate() + 3.months - 2.days)
let _ = Date() + 3.hours
let _ = date1 + [.year:1, .month:2, .hour:5]
let _ = date1 + date2
// 次のように判定できたりします
let _ = date.compare(.isToday)
let _ = date.compare(.isNight)
let _ = date.compare(.isNextWeek)
let _ = date.compare(.isThisMonth)
let _ = date.compare(.startOfWeek)
let _ = date.compare(.isNextYear)
```

日付によって XX したい、N 時間後に X したい等、日付に関わる処理をしたいときに便利！

3.6 PKHUD - HUD ライブラリ

- 簡単にいろいろな HUD を表示できるライブラリ
- https://github.com/pkluz/PKHUD

処理中、成功、失敗等の HUD を簡単に表示できるライブラリ。
例えば、次のように表示させることができます。(アニメーション付き)



▲図 3.2　PKHUD で使える HUD 集

3.7　Nuke - 画像読み込み・キャッシュライブラリ

- 画像読み込み、キャッシュライブラリ
- https://github.com/kean/Nuke

ImagePipeline という仕組みがとても便利で、画像取得の優先度を設定できたりタスクのキャンセルが簡単に行えます。

筆者が扱っているプロダクトでは、以前は SDWebImage を使っていたのですが、それよりも Nuke を使ったほうが画像表示が早く扱いやすかったので乗り換えました。

▼　サンプルコード

```
Nuke.loadImage(with: url, into: imageView)
```

▼　サンプルコード 2

```
Nuke.loadImage(
    with: url,
    options: ImageLoadingOptions(
        placeholder: UIImage(named: "placeholder"),
```

```
        transition: .fadeIn(duration: 0.33)
    ),
    into: imageView
)
```

▼ サンプルコード 3

```
  let options = ImageLoadingOptions(
      placeholder: UIImage(named: "placeholder"),
      failureImage: UIImage(named: "failure_image"),
      contentModes: .init(
          success: .scaleAspectFill,
          failure: .center,
          placeholder: .center
      )
  )

  Nuke.loadImage(with: url, options: options, into: imageView)
}
```

第 4 章

Firebase と連携してアプリの品質を上げる

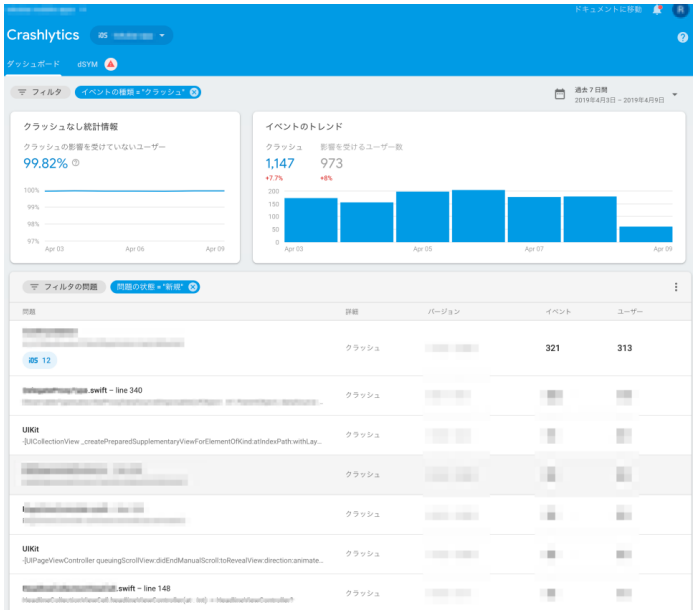
この章では、iOS アプリ開発においてもはや無くてはならぬ手軽で強力なモバイルおよび Web アプリのバックエンドサービスである、「Firebase」の機能について紹介します。Firebase のサービスはいくつかありますが、その中でもモバイルアプリに関わり、本書と趣旨があうものだけを抜粋して紹介します。

4.1 Firebase Crashlytics

Firebase Crashlytics はリアルタイムなクラッシュレポートツールで、アプリの品質を低下する要因（クラッシュや非重大なエラー）を追跡することができ、修正するのに役立ちます。

- Firebase Crashlytics
- https://firebase.google.com/docs/crashlytics/?hl=ja

例えば、これを導入してビルド、インストールしたアプリにクラッシュが起きると自動でレポートが Firebase に送信され、蓄積されていきます。次のように Firebase の Crashlytics 項目を選択すると確認ができます。



▲図 4.1　Firebase Crashlytics の画面

どのクラスのどの行でクラッシュしたか、iOS の特定のバージョンの問題なのか、どれくらい起きているのか等を確認できます。何らかのタイミングでクラッシュが急増した場合にも、メール等で自動通知してくれます。

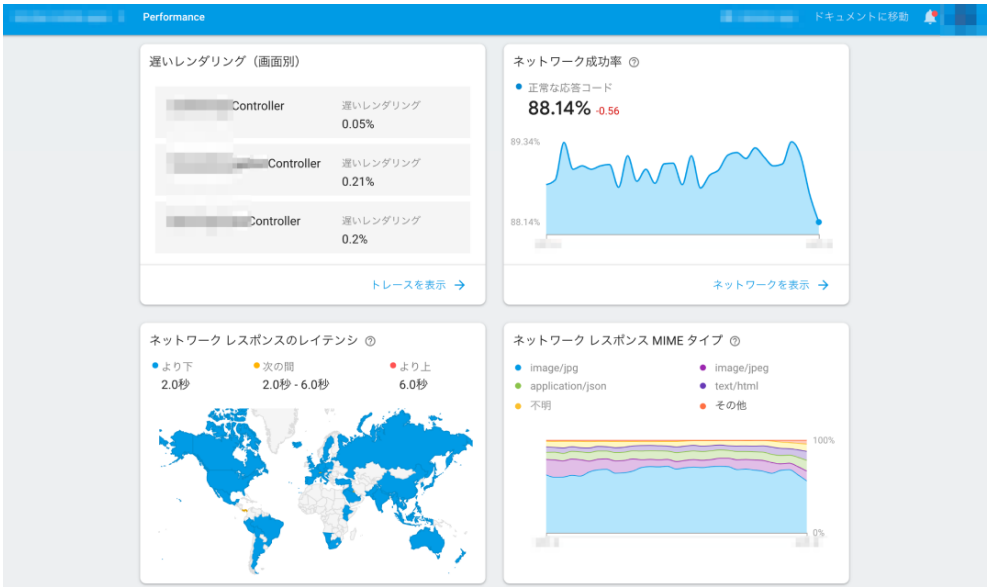
4.2　Firebase Performance Monitoring

iOS/Android アプリのパフォーマンスを自動で収集し、Firebase にレポートしてくれる便利なサービス

- Firebase Performance Monitoring
- https://firebase.google.com/docs/perf-mon/?hl=ja

主に、次のような指標を収集し、閲覧できます。

- レンダリングの速さ
- ネットワークレスポンス
- 起動時の所要時間 (ex. 中央値)



▲図 4.2　Firebase Performance の画面

ここからさらに端末別、OS 別、国別等でフィルタリングして表示することができます。また、このツールを使ってパフォーマンス改善したという記事を筆者が現在所属している会社のテックブログで公開しています。興味がある方はぜひ見てみてください！

- エンジニアドリブンでユーザー体験を約 250 %改善した話
- https://techblog.tokubai.co.jp/entry/kaizen-ux-lead-by-engineer

4.3　Firebase RemoteConfig

アプリのアップデートをしなくてもリアルタイムでアプリ内の動作や外観を変えることができる便利なツール

- https://firebase.google.com/docs/remote-config/?hl=ja
- Firebase RemoteConfig

主に機能をユーザ属性別（性別、国、年代等）に出したり、新機能を段階公開できたり、AB テストをやりたいときに使用します。次のように書きます。

▼

```
// 前提: RemoteConfig で取得したデータを管理する RemoteConfigManager というシングルトン
インスタンスを作っておく
// やりたいこと: 性別ごとに一部レイアウトの色が変わるようにしてみたい
navigationController.backgroundColor = RemoteConfigManager.shared.appThemeColor

// やりたいこと: 購入ボタンのテキストとカラーの AB テストをしたい
purchaseButton.setTitle(RemoteConfig.shared.purchaseButtonTitle, for: .normal)
purchaseButton.backgroundColor = RemoteConfig.shared.purchaseButtonColor

// API キーを誤って削除したときに即座に差し替えできるように RemoteConfig にセットする
let apikey = RemoteConfig.shared.apikey
```

RemoteConfig には、次のように値を追加します。

▲図 4.3　RemoteConfig の画面

4.4　Firebase Authentication

サービス内で匿名ユーザーログイン、メールアドレス＋パスワードログイン、Google アカウント連携、Twitter アカウント連携等といった仕組みを簡単に実装できる便利ライブラリ・サービス

- https://firebase.google.com/docs/auth/?hl=ja

一時的な匿名ユーザーでのログインは、次のような書き方で行うことができます

▼

```
Auth.auth().signInAnonymously() { (authResult, error) in
  let user = authResult.user
  let isAnonymous = user.isAnonymous  // true
  let uid = user.uid
}
```

メールアドレスとパスワードを使ってサインアップしたいときは次のような書き方でできます。（※匿名ユーザからの引き継ぎではないので注意）

▼

```
Auth.auth()
  .createUser(withEmail: email, password: password) { (authResult, error) in
    guard let user = authResult?.user else { return }
    let user = authResult.user
    let isAnonymous = user.isAnonymous  // false
    let uid = user.uid
  }
```

同様に、Google 連携や Twitter 連携も簡単にできるのでドキュメントを参照してみてください。

第 5 章

テストテクニック

5.1 XCTest

ユニットテスト、パフォーマンステスト、UI テストなどを作成、実行できるフレームワーク

- Apple - XCTest
- https://developer.apple.com/documentation/xctest

特に何もライブラリを導入しなくても、標準の Xcode でも実行でき、次のようにテストを書いていきます。

```
import XCTest
@testable import YOUR_PACKAGE_NAME

class test_sampleTests: XCTestCase {

    override func setUp() {
        // テスト実行前の準備
        // ex. UserDefaults の値の削除など
    }

    override func tearDown() {
        // test ○○の終了後の掃除
    }

    func testExample() {
        // サンプルテストケース
    }

    func testPerformanceExample() {
        // パフォーマンス計測用
        self.measure {
            // 計測したいコードを書く
```

```
        }
    }
}
```

試しにいくつかテスト対象のサンプルを書いてみます

▼

```
// Int+Common.swift
extension Int {
    func increment() -> Int {
        return self + 1
    }
    func decrement() -> Int {
        return self - 1
    }
}
// Array+Common.swift
extension Array {
    func isNotEmpty() -> Bool {
        return !self.isEmpty
    }
}
```

この場合、テストは次のように実装できます。

▼

```
// Int+CommonTest.swift
import XCTest
@testable import test_sample

class IntCommonTest: XCTestCase {
    func testIncrement() {
        let number = 0
        XCTAssertEqual(number.increment(), 1)
        XCTAssertEqual(number.increment().increment(), 2)
    }

    func testDecrement() {
        let number = 0
        XCTAssertEqual(number.decrement(), -1)
        XCTAssertEqual(number.decrement().decrement(), -2)
    }
}

// Array+CommonTest.swift
import XCTest
@testable import test_sample

class ArrayCommonTest: XCTestCase {
    func testIsNotEmpty() {
        let emptyArray: [Int] = []
        let oneArray: [Int] = [0]
        let twoArray: [Int] = [1,2]
```

```
        XCTAssertFalse(emptyArray.isNotEmpty())
        XCTAssertTrue(oneArray.isNotEmpty())
        XCTAssertTrue(twoArray.isNotEmpty())
    }
}
```

5.2 Quick

RSpec や Specta 等のライブラリに似たテスト用ライブラリ

- Quick
- https://github.com/Quick/Quick

まずは、ライブラリを導入します

▼

```
# Podfile
target 'test-sample' do
  use_frameworks!

  target 'test-sampleTests' do
    inherit! :search_paths
    pod 'Quick'
    pod 'Nimble'
  end

  target 'test-sampleUITests' do
    inherit! :search_paths
  end

end
```

▼

```
pod install
```

さきほどの XCTest を使って書いたテストコードを、Quick を使って書き直してみます。

▼

```
// Int+CommonTest.swift
import XCTest
import Quick
import Nimble

@testable import test_sample
```

```
class IntCommonTest: QuickSpec {
    override func spec() {
        describe("Test Extension") {
            let number = 0
            it("Test Increment") {
                expect(number.increment()).to(equal(1))
                expect(number.increment().increment()).to(equal(2))
            }
            it("Test Decrement") {
                expect(number.decrement()).to(equal(-1))
                expect(number.decrement().decrement()).to(equal(-2))
            }
        }
    }
}

// Array+CommonTest.swift
import XCTest
import Quick
import Nimble
@testable import test_sample

class ArrayCommonTest: QuickSpec {
    override func spec() {
        describe("Test Extension") {
            it("Test isNotEmpty") {
                let emptyArray: [Int] = []
                let oneArray: [Int] = [0]
                let twoArray: [Int] = [1,2]

                expect(emptyArray.isNotEmpty()).to(equal(false))
                expect(oneArray.isNotEmpty()).to(equal(true))
                expect(twoArray.isNotEmpty()).to(equal(true))
            }
        }
    }
}
```

Test Success!!

第 6 章

CI/CD - 継続的インテグレーションと継続的デリバリー

6.1 用語解説

- CI (Continuous Integration) 継続的インテグレーション
  - 例: PullRequest の作成時や master ブランチ変更時に自動ビルドとテストを実行し、品質の安定を図る
- CD (Continuous Delivery) 継続的デリバリー
  - 例: master ブランチが変更されるたびに自動でビルドし、自動でテスターに配布する

6.2 Bitrise

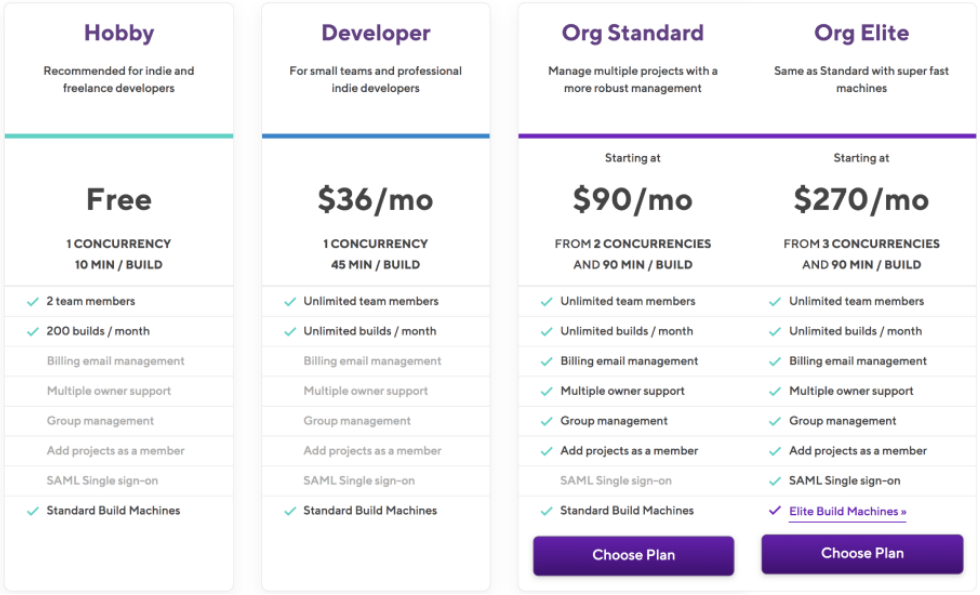
モバイルアプリケーション用の CI/CD サービス

- Bitrise
- https://www.bitrise.io/

サービス登録してワークフロー（github からの clone、ビルド、テスト、デプロイ、Slack 通知等）をポチポチと設定するだけで CI/CD 環境が構築できる便利なサービス

フリー版だと、直列ビルドでビルド時間に MAX10 分等の制限がついているので注意です。

※この本が出版された後に変わる可能性があります。



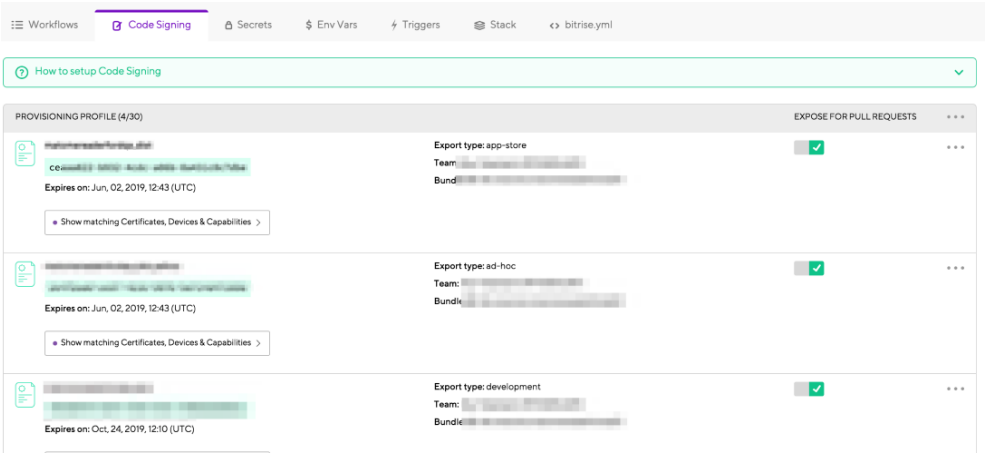
▲図 6.1　Bitrise 課金携帯

初回登録時、次の作業が必要です。

- アカウントの作成
- アプリを登録する（アプリ名とか）
- リポジトリを登録する (optional: Github 連携)
- ブランチを選択 (ex. master)

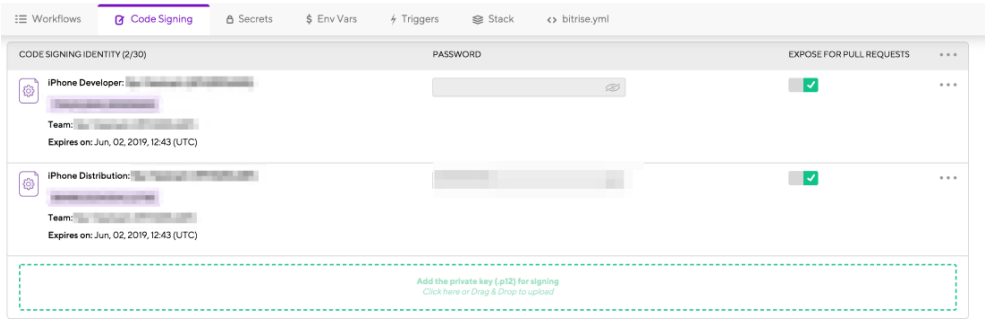
アプリの登録が終わったら、次は自動ビルド時に使う用の証明書を Bitrise サービスに登録します。

- アプリ > Workflow > Code Signing を開く
- PROVISIONING PROFILE セクションに.mobileprovision ファイルをドラッグしてアップロード



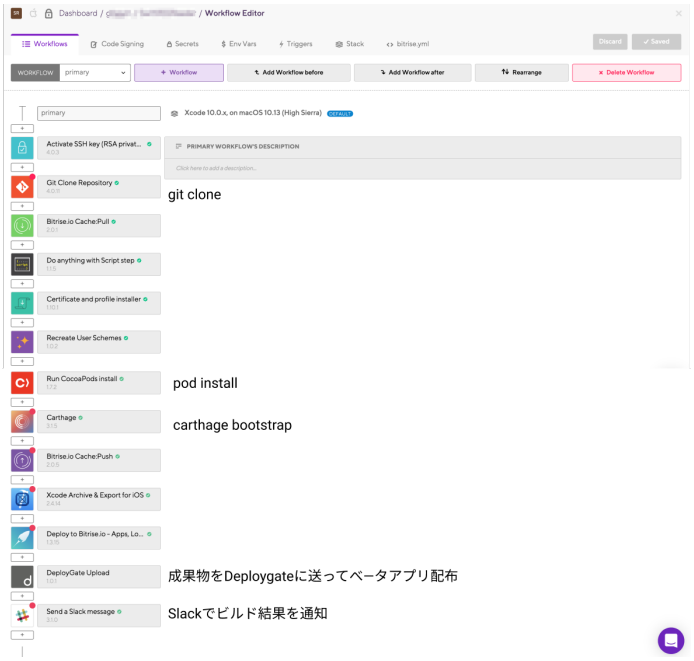
▲図 6.2　Provisioning Profile のアップロード

- CODE SIGNING IDENTITY セクションに.p12 ファイルをドラッグしてアップロード



▲図 6.3　.p12 ファイルのアップロード

次に、workflow を作ります。指定したブランチが更新されたときにどのようなアクションを取るか設定していきます。筆者の個人開発アプリでは次のようにしています。



▲図 6.4　Bitrise workflow

後は、master ブランチを更新したときに作った workflow が動けば成功！

手動で動かす場合は、**Start/Schedule** ボタンを押して branch を **master** にするとワークフローが走ります。

6.3　Fastlane

iOS/Android アプリ開発者向けの CI/CD 用ライブラリ

- Fastlane
- https://github.com/fastlane/fastlane
- https://docs.fastlane.tools/

スクリーンショットの作成、ProvisioningProfile の処理、アプリのデプロイ、リリース等の作業を自動化できます。

6.3.1 導入

今回は、試しに Testflight へコマンド一発でデプロイできる仕組みを作ってみます。まずはプロジェクトに fastlane を導入するため、次のコマンド郡を実行してください。

▼

```
# xcode commandline tool の導入
xcode-select --install
# RubyGems を利用してのインストール
sudo gem install fastlane -NV
# プロジェクトへ移動
cd YOUR_XCODE_PROJECT
# fastlane の導入
fastlane init
```

自動でセットアップすることもできますが、今回はマニュアルでセットアップしていきます。

▼

```
[□]
[□] Looking for iOS and Android projects in current directory...
[23:20:43]: Created new folder './fastlane'.
[23:20:43]: Detected an iOS/macOS project in the current directory: ' あなたのプ
ロジェクト名.xcworkspace'
[23:20:43]: ------------------------------
[23:20:43]: --- Welcome to fastlane    ---
[23:20:43]: ------------------------------
[23:20:43]: fastlane can help you with all kinds of automation for your mobile app
[23:20:43]: We recommend automating one task first, and then gradually
automating more over time
[23:20:43]: What would you like to use fastlane for?
1.     Automate screenshots
2.   □□    Automate beta distribution to TestFlight
3.     Automate App Store distribution
4.     Manual setup - manually setup your project to automate your tasks
?
```

4（マニュアルセットアップ）を入力したあと、各種ファイルの自動生成と fastlane のイロイロについての説明が色々出力されるので基本 Enter 連打で進めて下さい。完了すると、次のようなファイル群が生成されます。

- fastlane/Appfile
  - AppleID やアプリの BundleIdentifier を定義
- fastlane/Fastfile
  - 自動化処理を記述
  - 各種 lane を定義

* lane -> 1つ1つの自動化処理の名前のようなもの
* 自動デプロイする lane、自動スクショ撮る lane、自動テストする lane 等いろいろあります。

試しに、自動でテストを走らせる lane を定義してみます。その場合は、Fastfile を開き次のように編集します

▼

```
default_platform(:ios)

platform :ios do
  desc "test"
  lane :test do
    run_tests # 実行する処理  action と呼ばれている
  end
end
```

その後、次のコマンドを入力すると自動でテストが走ります

▼

```
fastlane test
```

また、action はもっとカスタマイズすることができます。

▼

```
platform :ios do
  desc "test"
  lane :test do
    run_tests(
      devices: ["iPhone 6s", "iPad Air"],
      clean: true
    )
  end
end
```

他にも、AppleStore への自動デプロイ、Testflight への自動デプロイ等いくつか action が用意されています。詳しくはドキュメントをご参照下さい。(https://docs.fastlane.tools/actions/)

最後に、筆者の個人開発アプリの Fastfile の中身を一部ここで紹介します。参考になれば嬉しいです！

▼

```
default_platform(:ios)

platform :ios do

  desc "Submit App to Fabric Beta"
  lane :beta do
    build_app(
      scheme: "Production-Debug",
      configuration: "Release-Adhoc",
      export_method: "ad-hoc"
    )
    crashlytics(
      api_token: "YOUR_TOKEN",
      build_secret: "YOUR_SECRET_KEY",
      groups: "development"
    )
    slack(
      slack_url: "https://hooks.slack.com/services/YOUR_SLACK_INCOMING_HOOK_URL",
      channel: "#notify",
      username: "fastlane"
    )
  end

  lane :release do
    build_app(
      scheme: "Production",
      configuration: "Release-AppStore",
      workspace: "YOUR_PROJECT_NAME.xcworkspace",
      export_method: "app-store",
      include_bitcode: true
    )
    upload_to_app_store
    slack(
      slack_url: "YOUR_SLACK_INCOMING_HOOK_URL",
      channel: "#notify",
      username: "fastlane"
    )
  end

end
```

第 7 章

次のステップへ

ここまででやってきたことを少し振り返ってみます。

- ライブラリ管理ツールの導入 (CocoaPods, Carthage)
- ライブラリの導入 (R.swift, Nuke 等)
- 品質向上のためのツール導入（Crashlytics、Performance Monitorning 等）
- テスト (XCTest, Quick)
- CI/CD (Bitrise, fastlane)

ここまでは、アプリ開発のおけるいわゆる**基盤**といった部分を支える仕組みを導入してきました。では、基盤を整えたら次は何をするか？ そう、グロースです。

アプリを成長させていきましょう！！！！

アプリを成長させるためのツールとして、本書で少し紹介した ‘RemoteConfig‘ の他にも ‘Firebase A/BTesting‘、‘Firebase Cloud Messaging‘等、たくさんあります。・・・が、それの紹介はまた次のお話ということで・・・。（打ち切りの可能性あり）

第 8 章

最後に

ここまで読んで頂き、ありがとうございました。本書は iOS アプリ開発初心者や「チュートリアル終わって、サンプルアプリを作れるようになったけど、次なにすりゃいいの？ というかみんなどんなツール使ってるの？ モダンな環境ってなんだろう？ 教えて〜〜」と過去に思っていた当時の自分に向けて最高の本を作ろうと思い、書き始めました。・・・が、業務が押していたり WWDC 当選したおかげで英語の勉強が挿し込んだり縁があって副業を週 1.5 でやることになったりと技術同人誌執筆に割く時間が減ってしまい、結果、各ツールやライブラリについて深堀りしていくことができず、広く・浅くの紹介・解説になってしまいました。それでも過去の自分がこの本を見て「めっちゃ最高に役立つじゃん！」と思えるような本になんとか頑張って仕上げました。これからのアプリ開発でどこかしらの部分で、本書で得た知識を活かして頂ければとても嬉しいです。

第 9 章

備考

(ページ調整用　メモ欄)

著者について

@k0uhashi

著者略歴

- 1997 年生まれ
  - 秋田県大仙市
- 2005 年〜 小学２年生〜５年生
  - はじめて PC に触れる。WarRock というネトゲにハマり、それを主軸にしつつ色々なネトゲをやりまくる（MapleStory、MixMaster、チョコットランド、アラド戦記等）小学校にあまり行かなくなる。
- 2008 年〜 小学５年生〜中学１年生
  - ネトゲの経験値レートに絶望し、（１日中ずっとレベル上げしてもで次のレベルアップまでの経験値のうち 3% しか稼げない）オンラインゲームのエミュ鯖という存在を知る。
  - MapleStory、Gunz というオンラインゲームのエミュ鯖を立て、友達に接続先を改変したクライアントを配布し、仲間内だけしかいないサーバー内で遊ぶようになる。経験値１００倍にしたり好きな武器や好きなクエストを作って遊ぶようになった。めちゃくちゃおもしろかった。
- 2010 年 中学１年生〜高校１年生
  - Minecraft にハマる。public なサーバーを立てて外の人を誘って自鯖でずっとプレイし続けた。そこそこ賑わった。中学はよく昼から登校していたが、一応そこそこ行った。
  - イベント期間は学校を連続で休むことがよくあり、担任からいじめの心配をされるようになった。
- 2012 年 高校１年生〜高校３年生
  - ドラゴンクエスト 10 にハマる。高校にあまりいかなくなった。（単位はとった）ニコニコ生放送でゲーム配信やったりした。
- 2015 年 専門１年生〜２年生

  - 岩手に移住。
  - 本格的にプログラミングをはじめた。
  - Unity/C#で 3D スクロールアクションゲームを作った。同級生にしかダウンロードされなかった
  - Unity/C#で VR ホラーゲームを作った。公開はしていない
  - Unity/C#でねこつまみという 2D ゲームを作ってニコニコ自作ゲームフェスに投稿した。全然ダウンロードされなかった
  - Unity/C#の講座動画を出した。全然再生されなかった。
  - Android/Java でゲーム攻略アプリを作った。そこそこダウンロードされた。毎月多い時で 6 万くらいの副収入
  - 応用情報技術者試験に合格した。
  - Rails でキュレーションサイトを作った。全然アクセスされなかった
  - Rails でプログラミング学習サービスを作った。全然アクセスされなかった
- 2017 年 エンジニア 1 年目
  - 東京進出。
  - Android/Java のゲーム攻略アプリをリニューアルした。そこそこダウンロードされた。
  - NintendoSwitch が品薄だったので入荷したらアフィ付きリンクで登録した人に即通知が飛ぶような仕組みを作った。170 万くらい売上に貢献した。
  - 月給 21 万
  - 年度末に転職した。
- 2018 年 エンジニア 2 年目
  - iOS/Swift のゲーム攻略アプリを出した。全然ダウンロードされなかった。
  - 本を書いた。技術書典 5 で「比較して学ぶ RxSwift4 入門」を頒布した。そこそこ売れた。
  - 本を書いた。商業版としてインプレス R & D より「比較して学ぶ RxSwift 入門」出版。どんくらい売れてるのかまだわからない。
  - 月給 33 万
- 2019 年 エンジニア 3 年目
  - とあるスタートアップで副業始めた。
  - 本を書いた。（この本）技術書典 6 で頒布した。そこそこ売れてほしい。（願い）
  - 月給 33 万（現在 4 月の執筆時点ではまだ前年度の評価が給料に反映されていないのでたぶん 2 ヶ月後くらいに変わりそう。変わってほしい。）＋ 副業 20 万

iOS アプリ モダン開発入門

2019 年 4 月 14 日　技術書典 6 版 v1.0.0
著　者　k0uhashi
編　集　k0uhashi
発行所　iOS アプリ開発がんばるぞ！！ の会
印刷所　日光企画　様