

iOS 11 Programming.



堤 修一
@shu223

池田 翔
@ikesyo

加藤 尋樹
@cockscomb

岸川 克己
@k_katsumi

吉田 悠一
@sonson_twit

所 友太
@tokorom

坂田 晃一
@huin

川邊 雄介
@jeffsuke

永野 哲久
@7gano

話題の ARKit, CoreML などの新フレームワークや、Drag and Drop, Metal 2, HomeKit, MusicKit, AirPlay 2, Swift 4, Xcode 9 の新機能、UIKit のアップデートなど iOS 11 の主要トピックを、国内の第一線級の開発者陣が詳しく解説！

それ知りたい！が本になる、技術書クラウドファンディング



iOS 11 Programming

堤 修一、吉田 悠一、池田 翔、坂田 晃一、加藤 尋樹、川邊

2017-07-12 版 発行

はじめに

「(ユーザーインターフェイス一新した) iOS 7 以来のインパクトがあるアップデートだなあ」これが iOS 11 の最初の印象でした。しかし、beta 版をインストールしてみても変更点が分からなかつたので、更に興味をそそられました。iOS 11 のアップデートを扱う本書は 500 ページ近くあります。にも関わらず、ぱっと触ってもやはり変更点が分からないです。簡単に言えば、目に見える部分である GUI の変更点が少なく、それ以外の変更点が多くあるからです。たとえば、Core ML という機械学習のためのフレームワークが新設されましたが、これは目に見えるものではなく、ユーザーの選択をサポートすることなどに使われます。また、こちらも新設された ARKit は、Apple の標準アプリではまだ使われていません（これは iOS では珍しいことです）。以上のことから、iOS 11 は「これから iOS の起点となる OS」なのではないか、と思いました。つまり GUI の洗練が一段落した今、これからのコンピューティングの方向性を模索するような新機能が「ひっそりと」搭載された OS であると。それらがいつ花開くのか、また開かないのかは分かりません。しかし、iOS 11 の全体像を把握し、各トピックを掘り下げるようなものあれば、今後（5 年～程度）のアプリやサービス開発のヒントとなるのではないか、と考えました。そこで、第一線の開発者の方々に声を掛け、本という形にまとめることにしました。

そうやって本書は作られました。以上のような発起人の動機は抜きにして、iOS 11 の決定版とも言うべき本になったと自負しております。アプリやサービス開発に少しでもお役に立てれば幸いです。

本書について

本書の目的は、iOS 11 のその全容を把握できるようにすることです。しかし、新機能・変更点は大小多岐に渡るため、すべてを解説することはできません。そこで、iOS 11 の中から特に重要と考えたトピックを掘り下げて提供することといたしました。

本書の構成は次の 4 部構成となっています。

- 第 1 部 Augmented Reality, Machine learning
- 第 2 部 Swift 4, Xcode 9
- 第 3 部 UIKit の新機能とアップデート
- 第 4 部 その他の新フレームワーク、アップデート

第 1 部では、iOS の新機能の中でも特に大きくとりあげられた、AR と機械学習を解説します。「第 2 章 ARKit」「第 3 章 Core ML」は共に基礎から応用までを深く解説しております。

第 2 部では、iOS 11 の土台となる「Swift 4」と「Xcode 9」を解説します。

第3部では、ユーザーインターフェイスの変更点を解説します。「Drag And Drop」「Document-Based Application」「AutoLayout 関連の変更点」を解説します。

第4部では「PDFKit」「Core NFC」「SiriKit」「HomeKit」「Metal」「MusicKit」「AirPlay2」を解説します。iOS 11 の新フレームワークではない SiriKit、HomeKit、Metal は、それぞれ入門からはじまり、iOS 11 におけるアップデートを解説する構成となっています。

本書の読み方

本書はごく一部を除いて、各章は互いに関連していません。そのため特に興味のある章から読む進めることができます。

サンプルコード

本書のサンプルコードは [iOS11samplecode リポジトリ](https://github.com/peaks-cc/iOS11samplecode/) (<https://github.com/peaks-cc/iOS11samplecode/>) にあります。MIT ライセンスとなっています。各章でプロジェクトファイルの指定がある場合を除き、chapter_ + 各章番号 が章ごとのサンプルコードとなっています。サンプルコードの誤りを見つけられた場合などは、issues にご投稿いただけすると幸いです。

謝辞

第3章の技術的なチェックをしてくださったデンソーアイティーラボラトリの関川雄介氏にお礼申し上げます。また、mlmodel ファイルのコンパイルについて、極めて重要な質問を送ってください、結果、筆者が誤解していたところを修正できたことを@taniguche@氏にもお礼申し上げます。第2章（ARKit）、第13章（Metal）の査読をしてくださった Fyusion Inc. の登本悠介氏、第13章の査読をしてくださった後藤年宏 (@x67x6fx74x6f) 氏へ、この場を借りて、お礼を申し上げます。日高正博さん (@mhidaka) には、本書の制作全般についてアドバイス・サポートいただきました。熊谷友宏さん (@es{kumagai}) には、Tex の数式レンダリングでサポートいただきました。

クラウドファンディングと PEAKS

本書は技術書クラウドファンディング・サービスである「PEAKS」のプロジェクトとして開始され、645人の支援者のサポートによって作られました。出資者特典である「アーリーアクセス」でいただいたご意見も反映されております。PEAKS ではこんな本を作りたい！ という方を募集しています。<https://peaks.cc/request> からご連絡板だKればさいわいです。

目次

はじめに	i
本書について	i
本書の読み方	ii
サンプルコード	ii
謝辞	ii
クラウドファンディングと PEAKS	ii
第1章 第1章 iOS 11 の概要	1
1.0.1 新しい API	1
1.0.2 Core Technology	1
1.0.3 iPad の GUI が大幅に変更	2
1.0.4 アプリの 64bit 対応が必須に	2
1.1 Xcode 9	2
1.2 Swift 4	2
1.3 UI の変更点	2
1.4 Siri	3
1.5 Depth Maps	3
1.6 Vision	3
1.7 その他のフレームワーク	3
1.8 まとめ	3
第1部 Augmented Reality, Machine learning	5
第2章 ARKit	6
2.1 はじめに	6
2.2 ARKit 入門その1 - 最小実装で体験してみる	6
2.2.1 手順1: プロジェクトの準備	7
2.2.2 手順2: ViewController の実装	7
2.2.3 基本クラスの解説	9
2.3 ARKit 入門その2 - 水平面を検出する	12
2.3.1 水平面を検出するためのコンフィギュレーション	12

2.3.2	平面検出に関するイベントをフックする - ARSessionDelegate	13
2.3.3	平面検出に関するイベントをフックする - ARSCNViewDelegate	15
2.3.4	検出した平面を可視化する	16
2.4	ARKit 入門その 3 - 検出した水平面に仮想オブジェクトを置く	18
2.4.1	3D モデルを読み込む	18
2.4.2	仮想オブジェクトとして検出した平面に置く	19
2.5	ARKit 開発に必須の機能	20
2.5.1	トラッキング状態を監視する	20
2.5.2	デバッグオプションを利用する	21
2.5.3	トラッキング状態をリセットする / 検出済みアンカーを削除する	25
2.6	特徴点 (Feature Points) を利用する	26
2.6.1	特徴点を可視化する	27
2.6.2	特徴点の ID や座標を取得する	28
2.7	AR 空間におけるインタラクションを実現する	29
2.7.1	ヒットテスト (当たり判定) を行う	29
2.7.2	デバイスの移動に対するインタラクション	34
	SCNBillboardConstraint を用いる方法	37
2.8	アプリケーション実装例 1 : 現実空間の長さを測る	38
2.8.1	ARKit における座標と現実のスケール	38
2.8.2	現実空間における二点間の距離	39
2.9	アプリケーション実装例 2 : 空中に絵や文字を描く	41
2.9.1	実装方針	42
2.9.2	スクリーンの中心座標をワールド座標に変換する	42
2.9.3	頂点座標の配列から、線としてのカスタムジオメトリを構成する	43
2.9.4	その他の実装のポイント	44
2.10	アプリケーション実装例 3 : Core ML + Vision + ARKit	46
2.10.1	実装方針	47
2.10.2	Core ML・Vision・ARKit 連携のポイント	47
2.11	Metal + ARKit	50
第 3 章	Core ML	52
3.1	はじめに	52
3.1.1	ユースケース	52
3.2	本章の構成	53
3.3	Core ML のために学ぶ機械学習	54
3.4	機械学習の分類	54
	独立同分布 (independent identically distributed, i.i.d.)	56
3.4.1	データとモデルとパラメータ	57
3.4.2	学習と推論	57
3.4.3	学習性能と汎化性能	62

3.4.4	過学習	63
3.4.5	データとモデル	68
3.5	まとめ	68
3.6	Core ML	68
3.6.1	概要	68
3.6.2	サポートされるモデル	69
3.6.3	Core ML のスタック	69
3.6.4	開発の流れ	70
3.6.5	開発環境	71
3.6.6	Core ML Tools	71
3.7	実装	74
3.7.1	回帰～scikit-learn	75
3.7.2	モデルとパラメータ	76
3.7.3	学習	77
3.7.4	予測と評価	78
3.7.5	モデルデータの変換	80
3.7.6	Xcode 上での開発	81
3.7.7	CNN～Keras	82
3.7.8	ニューラルネットワーク	84
3.7.9	活性化関数	85
3.7.10	プーリング	87
3.7.11	畳み込みニューラルネットワーク	87
3.7.12	最適化	88
3.7.13	過学習と歴史	88
3.7.14	設計指針	89
3.7.15	モデルファイルの書き出し	90
3.8	Core ML の短所	94
3.9	まとめ	97
3.10	参考文献	98
第 II 部	Swift 4, Xcode 9	100
第 4 章	Swift 4 の新機能とアップデート	101
4.1	はじめに	101
	Swift 4 コンパイラの言語モード	101
4.2	Codable プロトコル	101
4.2.1	Codable プロトコルの基本的な使い方	102
4.2.2	コンパイラによる実装の自動生成	104
4.2.3	CodingKeys	105
4.2.4	Decodable と Decoder	106

4.2.5	JSONDecoder	113
4.2.6	Encodable と Encoder	116
4.2.7	JSONEncoder	118
4.2.8	継承関係の表現（superDecoder() と superEncoder()）	119
4.2.9	自動生成による実装と手動実装の混在	122
4.3	Smart KeyPaths	123
4.3.1	KeyPath 構文	124
4.3.2	KeyPath による subscript	124
4.3.3	KeyPath のクラス階層	125
4.3.4	Foundation の追加 API	128
4.4	参考文献	130
 第 5 章 Xcode 9 の新機能		
5.1	はじめに	131
5.1.1	本章の構成	131
5.2	開発フェーズの新機能	132
5.2.1	Github の統合とソースコード管理機能の改善	132
5.2.2	ソースコードエディタの改善	138
5.2.3	シミュレータの新機能	143
5.3	デバッグフェーズの新機能	149
5.3.1	ワイヤレスデバッグ機能	149
5.3.2	ビューデバッグの改善	151
5.4	テストフェーズの新機能	154
5.4.1	XCTest の新 API	154
5.4.2	UI テストの新機能	161
5.4.3	コマンドラインサポートの強化	163
5.5	Xcode サーバーの利用	167
5.5.1	Xcode サーバーの概要	167
5.5.2	サーバーのセットアップと Bot の作成・実行	167
5.5.3	トリガーを利用したスクリプトの実行	173
5.5.4	アーカイブと iOS デバイスへの配布	177
5.6	まとめ	182
 第 III 部 UIKit の新機能とアップデート		184
 第 6 章 Drag and Drop		
6.1	ドラッグ＆ドロップによるデータのやり取り	185
6.1.1	NSItemProvider	186
6.1.2	Uniform Type Identifier (UTI)	186
6.1.3	データをやり取りする	187

6.1.4	非同期的にデータを提供する	188
6.1.5	データやファイルでの受け渡し	189
6.1.6	複数のデータ形式を扱う	190
6.1.7	NSItemProvider に対応したクラスを作る	190
6.1.8	ペーストボードで NSItemProvider を利用する	191
6.2	ドラッグ	193
6.2.1	ドラッグのプレビュー	195
6.2.2	ドラッグのアニメーション	198
6.2.3	複数アイテムのドラッグ	199
6.2.4	データの移動	201
6.3	ドロップ	201
6.3.1	ペースト	201
6.3.2	UIDropInteraction	202
6.3.3	ドロップのプレビューとアニメーション	204
6.3.4	ドロップの進捗表示	205
6.3.5	同一アプリ内でのドラッグ&ドロップ	206
6.4	スプリングローディング	207
6.5	UITableView と UICollectionView	208
6.5.1	UITableView と UICollectionView のドラッグ	208
6.5.2	ドロップ	209
6.5.3	ドロップのプレースホルダ	213
6.5.4	同一アプリ内でのドラッグ&ドロップ	214
6.5.5	UITableView と UICollectionView のスプリングローディング	216
6.6	UITextView と UITextField	217
6.6.1	ドラッグのカスタマイズ	217
6.6.2	ドロップのカスタマイズ	218
6.6.3	ドロップとペーストのカスタマイズ	218
6.6.4	受け付けるデータの種類	219
第 7 章	Files と Document Based Application	221
7.1	はじめに	221
7.2	Files アプリ	221
7.3	ドキュメントブラウザ API	222
7.4	Document-Based App の実装	224
7.4.1	Documents ディレクトリの公開	224
7.4.2	サポートするドキュメント形式の宣言	224
7.4.3	ルート View Controller の設定	226
7.4.4	UIDocumentBrowserViewController のカスタマイズ	227
7.4.5	ドキュメントの選択	229
7.4.6	ドキュメントの新規作成	229

7.4.7	Open-in-place に対応する	230
7.4.8	ドキュメントのオープンと編集	231
7.4.9	UIDocumentBrowserViewController へのアクションの追加	232
7.4.10	UIDocumentBrowserViewController のトランジション	233
7.4.11	Spotlight	234
7.5	Thumbnail Extension	235
7.6	Quick Look Preview Extension	237
7.7	おわりに	238
第 8 章	レイアウト関連の新機能及び変更点	239
8.1	ラージタイトルと UINavigationBar	239
8.1.1	ラージタイトルをナビゲーションバーに追加	240
8.1.2	検索フィールドをナビゲーションバーに追加	242
8.2	Auto Layout とレイアウト手法のアップデート	245
8.2.1	レイアウトマージンの独自定義	245
8.2.2	セーフエリア	245
8.2.3	UIScrollView の変更点	249
8.2.4	UITableView の変更点	251
8.2.5	UIStackView の変更点	254
8.3	iOS 11 におけるアクセシビリティ、ダイナミックタイプ関連のアップデート	257
8.3.1	ダイナミックタイプでカスタムフォントを使う	257
8.3.2	ダイナミックタイプと行間スペースの調整	257
8.3.3	アクセシビリティコンテントサイズに合わせた画像の拡大縮小	258
8.3.4	Accessibility Inspector によるデバッグ	260
8.4	参考文献	262
第 IV 部	その他の新フレームワーク、アップデート	264
第 9 章	Core NFC	265
9.1	はじめに	265
第 10 章	PDF Kit	266
10.1	はじめに	266
10.2	PDF Kit とは	266
10.3	基本的な使い方	266
10.4	PDFView	269
10.4.1	表示形式のカスタマイズ	270
10.4.2	ページの移動	274
10.5	PDFThumbnailView	276
10.6	PDFDocument	276
10.6.1	ページ情報の取得	276

10.6.2 PDFPage	276
10.6.3 目次情報の取得	277
10.6.4 PDFOutline	277
10.6.5 テキストの抽出	281
10.6.6 PDF を検索する	284
10.7 PDFSelection	291
10.8 PDFAnnotation	291
 第 11 章 SiriKit	292
11.1 SiriKit とは	292
11.2 iOS 11 の変更点	293
11.2.1 To-Do 管理とメモ帳 (List and Notes) ドメイン	293
11.2.2 QR コード (Visual Code) ドメイン	294
11.3 動作のしくみ	295
11.3.1 SiriKit とアプリの関係	295
11.3.2 Intents Extension での動作	296
11.4 アプリ実装の準備	299
11.4.1 収容アプリ	299
11.4.2 Intents Extension	301
11.5 サンプルプロジェクト : To-Do 管理とメモ帳 (List and Notes)	306
11.5.1 IntentHandler (ハンドラの生成)	306
11.5.2 タスクリストのタイトルを検証する (リゾルブ)	310
11.5.3 Intent を実行する前の最終確認 (コンファーム)	311
11.5.4 INCreateTaskListIntent を処理する (ハンドル)	311
11.6 サンプルプロジェクト : QR コード表示	313
11.6.1 IntentHandler (ハンドラの生成)	313
11.6.2 QR コードの種類を確認する (リゾルブ)	314
11.6.3 Intent を実行する前の最終確認 (コンファーム)	315
11.6.4 INGetVisualCodeIntent を処理する (ハンドル)	316
11.6.5 連絡先から QR コードを生成する	318
11.7 Extension のデバッグ	321
11.8 まとめ	325
 第 12 章 HomeKit 入門と iOS 11 のアップデート	326
12.1 はじめに	326
12.1.1 本章の構成	326
12.2 HomeKit 入門	327
12.2.1 HomeKit でできること	327
12.2.2 HomeKit の構成	327
12.2.3 アクセサリへのアクセス	334
12.2.4 アクションとシーンの利用	340

12.2.5 トリガによるオートメーション	341
12.2.6 ユーザの管理	345
12.2.7 変更の監視	347
12.2.8 カメラの利用	347
12.2.9 Siri の利用	354
12.3 iOS 11 でのアップデートまとめ	356
12.3.1 新しいイベントの追加	356
12.3.2 HMEventTrigger のアップデート	359
12.3.3 HMHome のアップデート	362
12.3.4 HMAccessory と HMCharacteristic のアップデート	362
12.3.5 HMHomeDelegate のアップデート	363
12.3.6 HMAccessoryDelegate のアップデート	363
12.4 HomeKit 実践	363
12.4.1 HomeKit Accessory Simulator の利用	363
12.4.2 HomeKit 対応製品利用実例	365
12.5 まとめ	386
12.6 HomeKit お役立ちリファレンス	386
12.6.1 HMAccessoryCategory.categoryType 一覧	386
12.6.2 HMService.serviceType 一覧	386
12.6.3 HMCharacteristic.characteristicType 一覧	386
12.6.4 HMCharacteristicMetadata.format 一覧	386
12.6.5 HMCharacteristicMetadata.units 一覧	386
第 13 章 Metal	391
13.1 Metal の概要	391
13.1.1 Metal とは	391
13.1.2 OpenGL ES と Metal	391
13.1.3 Metal の用途	392
13.2 Metal の基礎	393
13.2.1 Metal の基礎概念	393
13.2.2 Metal の基本クラス	394
13.3 MetalKit	397
13.3.1 MTKView	397
13.3.2 MTKTextureLoader	399
13.4 Metal 入門その 1 - 画像を描画する	400
13.4.1 1. 描画処理のためのセットアップを行う	400
Metal 非対応デバイスの判定	400
13.4.2 2. 画像をテクスチャとしてロードする	401
13.4.3 3. 描画処理を実行する	402
13.5 Metal 入門その 2 - シェーダを利用する	408

13.5.1	Metal シェーダの基礎	409
13.5.2	「画面全体を一色に塗る」シェーダの実装	411
13.5.3	シェーダにデータを渡すためのバッファを準備する	413
13.5.4	MTLRenderPipelineDescriptor を作成する	414
13.5.5	MTLRenderPipelineState を生成する	415
13.5.6	レンダリングコマンドの作成	416
13.6	Metal 入門その3 - シェーダでテクスチャを描画する	419
13.6.1	テクスチャを扱うシェーダの実装	419
サンプラーを引数から渡す場合	421	
13.6.2	テクスチャ内座標データをシェーダに渡す	421
13.6.3	テクスチャをシェーダに渡す	423
13.6.4	ピクセルフォーマットを合わせる	423
13.7	ARKit+Metal その1 - マテリアルを Metal で描画する	424
13.7.1	SCNProgram	425
13.7.2	SCNProgram を利用する場合のシェーダの実装	426
13.8	ARKit+Metal その2 - Metal による ARKit のカスタムレンダリング	428
13.9	Metal 2	429
13.9.1	6つのキーフィーチャー	430
13.9.2	Argument Buffers	431
13.10	Metal を動作させるためのハードウェア要件	436
第 14 章	Audio 関連アップデート	438
14.1	はじめに	438
14.2	MusicKit	438
14.2.1	MusicKit とは	438
14.2.2	Apple Music API	439
14.2.3	StoreKit	452
14.2.4	MediaPlayer	461
14.2.5	Apple Music API の他のトピック	465
14.3	AirPlay 2	468
14.3.1	AirPlay 2 とは	468
14.3.2	AirPlay 2 に対応する	468
14.3.3	Long-Form Audio	468
14.3.4	AirPlay Picker を表示する	469
14.3.5	Enhanced Buffering に対応する	472
14.3.6	AVPlayer/AVQueuePlayer を使って再生する	472
14.3.7	AVSampleBuffer クラスを使って再生する	472
14.4	AVAudioEngine のアップデート	475
14.4.1	AVAudioEngine とは	475
14.4.2	AVAudioEngine のマニュアル・レンダリング	478

14.4.3 マニュアル・レンダリングをエフェクト機能として組み込む	480
14.4.4 まとめ	481

第 1 章

第 1 章 iOS 11 の概要

2017 年 6/5 より開催された **WWDC 2017** はこれまでになく意欲的な新機能やアップデートが多く「今年の WWDC は面白い」という声がよく聞かれました。Keynote ではハードウェアの発表も多かったのですが、開発者向けの概要である Platforms State of the Union^{*1} の内容にそって、iOS 11 関連の概要をまとめます。<https://developer.apple.com/videos/play/wwdc2017/102/>

1.0.1 新しい API

代表として ARKit、Core ML、MusicKit のアイコンが登場します。iOS 11 の一番目立つフィーチャーはなんといっても ARKit と Core ML でしょう。**Augmented Reality, Machine Learning** として紹介されるこの二つは本書の目玉でもあります。ともに入門から解説してありますので、第一部を参照ください。14 章で解説する MusicKit は、Apple Music ヘフルアクセスできるもので、iOS 標準のミュージック App における Apple Music に関するほとんどの機能を実装できます。

1.0.2 Core Technology

次に、APFS、Metal、HEVC/HEIF によるメディアのアップデートを表すアイコンが登場します。APFS (Apple File System)^{*2} はモダンでパワフルな新しいファイルシステムだと紹介されます。iOS は 10.3 から APFS となりました。

https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/APFS_Guide/Introduction.html
13 章で解説する Metal はローレベルなグラフィックス API です。GPU ヘアクセスするための API と言ってもよく、Core ML を支えるものもあり、ARKit とも関連するため iOS 11 でのアップデートと合わせて入門から解説しています。

HEVC/HEIF は新しい動画/画像のフォーマットです。ともにこれまでの 2 倍の高压縮率と言われており、iOS 11 以降の標準フォーマットとなります。

*1

*2

1.0.3 iPad の GUI が大幅に変更

次に、iPad がこれまでになく大きく変わり、生産性が向上することが述べられます。iPad の変更を支えるものとして、Dock, マルチタスキングの改善などが紹介されますが、開発者が注目すべきは Drag and Drop と Document-Based Apps です。Drag and Drop はその名のとおり、ドラッグとドロップを iPad ではアプリケーション間で、iPhone ではアプリケーション内でサポートします。Document-Based Apps は Files の登場と合わせてドキュメントを中心としたアプリが簡単に開発できるようになります。6 章、7 章で詳細に解説します。

つついで、macOS, watchOS, tvOS の変更が述べられたあと、アプリの審査が早くなったり、ユーザーレビューに返信できるようになったことなどが語られます。こちらも開発者には影響の大きい改善です。

1.0.4 アプリの 64bit 対応が必須に

次に、iOS 11 では **64bit** 対応が必須になることが発表されます。これは注意しなければならないでしょう。

1.1 Xcode 9

次に、Swift Playgrounds のアップデートと Xcode 9 の変更点が語られます。Xcode 9 で大きく発表されたのは、エディター部分のリニューアルによる改善です。Swift のリファクタリングがやっとできるようになり、ファイルの開く速度、フレームレートが速くなったと発表されています。Xcode 9 については 5 章を参照ください。

1.2 Swift 4

Swift 4 の特徴として

- 文字列処理の改善
- Codable
- Building Large Project(40% 向上)

が語られます。特に Codable はデータのシリアル化、デシリアル化を扱いやすくする機能で、特別な要件がなければ、積極的に採用すべきものといえるでしょう。Swift 4 については 4 章で解説します。

Xcode 9 と Swift 4 でビルトがとにかく速くなったことが繰り返し述べられます。Swift だとビルトが遅いという声がよく聞かれたので、これはよい改善です。

1.3 UI の変更点

- Large Title

- Dynamic Type

が紹介されます。

iOS 11 では、ナビゲーションバーのタイトルが大きくなり、スクロール時の挙動がこれまでと変わります。これを Large Title と呼びます。Dynamic Type は、設定で文字サイズを変更できる機能で、iOS 10 から搭載されていますが、より簡単に対応できるようになりました。AutoLayout の変更と合わせて 8 章で解説します。

1.4 Siri

Siri に対応したアプリケーションを作成できる SiriKit に、新たに以下のドメインが追加されました。

- Payment accounts (送金)
- Lists (タスク)
- Notes
- QRCodes

11 章で入門から追加された新しいドメインまで解説しています。

1.5 Depth Maps

二つのカメラを使って深度を取り、それが開発者にも利用可能なことが解説されます。たとえば、前方にいる人と後方にいる人が区別し、それぞれにエフェクトをかける、といったことができるようになります。機種に制限がありますが、API も公開されています^{*3}。

1.6 Vision

Vision^{*4}は画像認識のための新しいフレームワークです。Vision と Core ML が統合されており、Core ML の上に Vision が構築されていることも解説されます。

1.7 その他のフレームワーク

本書では上記以外に新設されたフレームワークとして iOS で NFC を Core NFC (9 章)、PDFKit (10 章)、アップデートと合わせて入門から解説する Home Kit (12 章)、Air Play2, AVAudioEngine (ともに 14 章) を扱います。

1.8 まとめ

- Core ML、ARKit といった新しいフレームワーク

^{*3} AVFoundation の機能として提供されています。 <https://developer.apple.com/documentation/avfoundation/avdepthdata>

^{*4} <https://developer.apple.com/documentation/vision>

- Core Technology の改善
- iPad の大幅な更新
- Large Title, Dynamic Type、Drag And Drop, Document-Based Apps による UIKit の更新
- その他のフレームワークの追加
- これらを支える Swift 4 と Xcode 9

という構図が見えてきます。

第 I 部

Augmented Reality, Machine learning

第2章

ARKit

2.1 はじめに

ARKit は、**AR (Augmented reality／拡張現実)** を構築するためのフレームワークです。iOS 11 の新機能の中でも非常に期待度が高く、正式リリースを待たずしてネットではすでに多くの画期的なデモが公開され、開発者の間だけではなく一般ユーザーの間でも話題になっています。本章では、そういった **ARKit** を使った画期的なアイデアをいち早く具現化できるようになることを主眼に、ARKit と関連フレームワークを用いた「具体的な実装方法」を解説します。

最初の「ARKit 入門」では、はじめの一歩として、最小実装で ARKit を体験します。ARKit を用いれば、実にシンプルな実装で強力な AR 機能が利用できることを実感していただけることでしょう。

その後は平面を検出する方法、その平面に仮想オブジェクトを設置する方法、そしてその仮想オブジェクトとインタラクションできるようにする方法…と、読み進めるにつれて「作りながら」引き出しが増えていき、最終的には ARKit を用いた巻尺（メジャー）や、空間に絵や文字を描くといった、話題になったようなアプリケーションの実装ができるよう構成しています。

ARKit は現実の空間を利用するフレームワークなので、自分の手元（スマホ）で実際に動かし体感しながら学んでいくのがベストです。本章のサンプルプロジェクトは [iOS11_samplecode リポジトリ](#) の `chapter_02` フォルダにありますので、ぜひ Xcode を開いてサンプルを動かし、デバイスを持って歩き回ったりするなど、楽しみながら学んでください。

2.2 ARKit 入門その1 - 最小実装で体験してみる

非常に高機能・高性能な ARKit ですが、シンプルな API で簡単に扱えるようになっています。各クラスの役割や詳細な使い方は後述するとして、まずは（約）3行でできる **ARKit** の最小実装を示しますので、「こんなに簡単にできるのか」と実感してもらいつつ、そこから ARKit の API デザインの大枠を掴んでください。

実は Xcode 9 の「Augmented Reality App」テンプレートからプロジェクトを生成すれば、一行もコードを書かずにそのままビルドして AR 機能が動作するアプリが新規作成できてしまいます。ですが、「非常に簡単に実装できる」という点を体感してもらうためにも、本節では既存プロジェクトに数行のコードを追加して **AR** 機能を構築する、というところから始めます。

(サンプルコード: 01_FirstAR)

2.2.1 手順1: プロジェクトの準備

プロジェクトの設定やアセットの追加を行います。

まず、ARKit ではカメラを利用するので、Info.plist に NSCameraUsageDescription キー^{*1}を追加しておきます。

Interface Builder（以降"IB"と表記します）から、ARSCNView オブジェクトを追加します。ViewController 直下の UIView オブジェクトのクラスを ARSCNView に変更するだけでも OK です。

次に、ARKit を用いて「現実世界にオーバーレイするシーン」のデータをプロジェクトに追加します。ここでは Xcode の「Augmented Reality App」テンプレートに含まれているシーンファイル ship.scn (図 2.1)^{*2}とそこから参照している texture.png を追加^{*3}することにします。

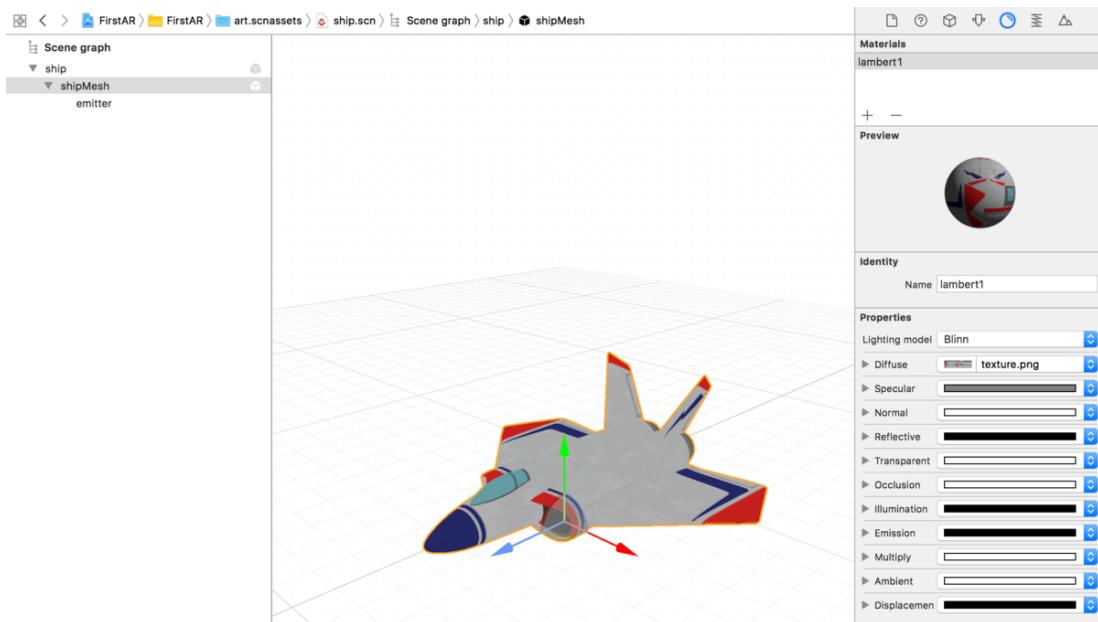


図 2.1: ship.scn

2.2.2 手順2: ViewController の実装

ARKit をインポートします。

^{*1} Xcode の Info.plist エディタでは「Privacy - Camera Usage Description」と表示されます。

^{*2} .scn は SceneKit 用のシーンのデータを扱うファイルの拡張子です。

^{*3} サンプルではこれらのファイルを、.scnassets という拡張子のついた SceneKit のアセット管理用フォルダに入れています。

```
import ARKit
```

ARSCNView を保持するプロパティを用意し、手順 1 で IB 上に用意した同クラスのオブジェクトと接続しておきます。

```
@IBOutlet var sceneView: ARSCNView!
```

viewDidLoad() に次の 3 行を追加します。

```
// シーンを生成して ARSCNView にセット
sceneView.scene = SCNScene(named: "art.scnassets/ship.scn")!

// セッションのコンフィギュレーションを生成
let configuration = ARWorldTrackingConfiguration()

// セッション開始
sceneView.session.run(configuration)
```

さあ、これでもうビルドして実機で動かせるようになりました。

アプリを起動してすぐにカメラ入力がスタートするので、周りをグルッと見てください。すると、飛行機が設置されていることがわかります。デバイスを持って移動してみても、飛行機は最初の位置にしっかりと固定され、色々な角度から見ることができます。

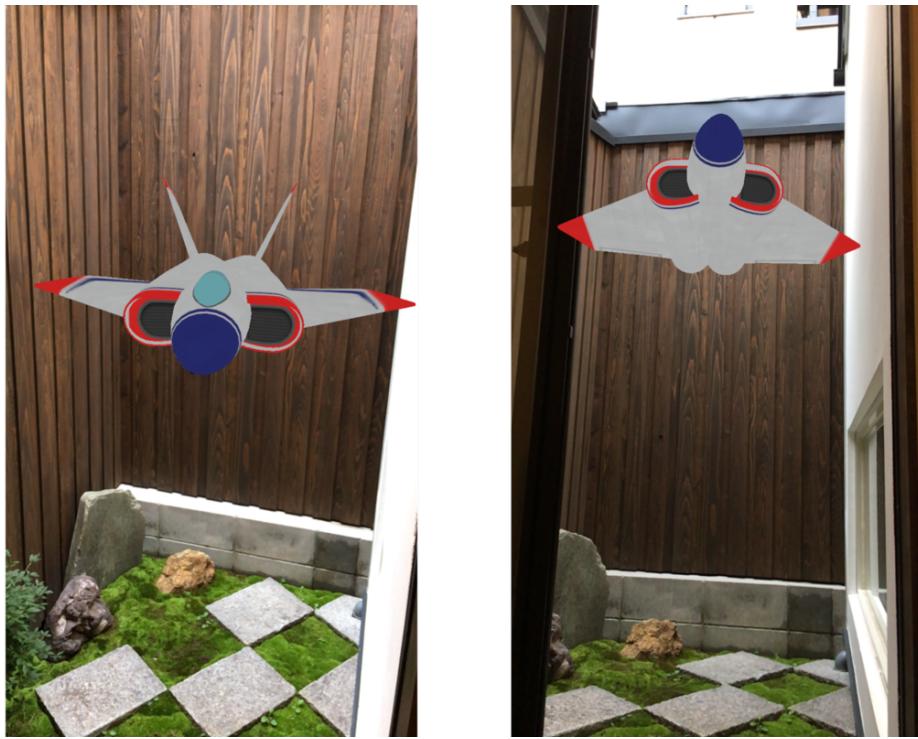


図 2.2: 仮想オブジェクトが現実世界に設置され、色々な角度から見ることができる

つまり、わずか 3 行程度の実装によって、現実世界に仮想シーンをオーバーレイし、デバイスの位置や角度の変化に追従する AR の機能が構築できることになります。

2.2.3 基本クラスの解説

最小実装での挙動を確認できたところで、上で書いたコードが何をしていたのかを把握するため、出てきたクラスについて簡単に見ていきましょう。

ARSession

ARKit では、その AR 機能全体を `ARSession` クラスを用いてセッション単位で管理します。

`ARSession` の `run(_:)` メソッドを呼ぶと、セッションが動作開始します。セッションが開始すると、ARKit 内部でカメラからの入力の画像解析や、デバイスのモーション情報の取得・解析が開始され、現実空間の認識とデバイスのトラッキング（現実空間におけるデバイス＝自分の位置や向きを追跡）が始まります。

ARKit のない時代に AR を実現しようとした場合、AVFoundation でカメラ入力画像データの取得、Core Motion でモーション情報の取得、かつそれらを「リアルタイムに」解析・統合するという、非常に複雑で難易度の高い実装が必要でした。

それが、次のように `run(_:)` メソッドを呼ぶだけで実現できるようになったのです。

```
session.run(configuration)
```

`run` メソッドの定義は次のようになっており、第1引数には次項で解説する `ARConfiguration` を指定します。

```
func run(_ configuration: ARConfiguration, options: ARSession.RunOptions = [])
```

第2引数の `options` はデフォルト値（オプションなし）が与えられており省略可能です。本節でもできるだけコードをシンプルにするために省略しています。^{*4}

ARKit のセッションを中断したい場合は、`pause()` メソッドを呼びます。

```
session.pause()
```

なお、セッションにはもう1つ「リセット」という概念が存在します。この「リセット」については、追加の概念や実装についての説明が必要となるので、後の「2.5.3 トラッキング状態をリセットする / 検出済みアンカーを削除する」で解説します。

ARConfiguration

`ARConfiguration` クラスは、ARKit のセッションの様々なコンフィギュレーションを管理するためのクラスです。`ARSession` を `run` するときの引数に渡します。

上の実装で出てきた `ARWorldTrackingConfiguration` は `ARConfiguration` のサブクラスです。デバイスの向きや位置をトラッキングし、かつデバイスのカメラから見える現実世界の「面」^{*5} を検出する、というコンフィギュレーションを表すクラスです。

このクラスを用いて、たとえば「どんな種類のトラッキングを行いたいか」といったことが設定できます。上の最小実装のように、何も指定しない、クラスを初期化しただけのデフォルト設定のままでも使用できます。

ARSCNView

ARKit はコアな処理のみを行い、実際の描画は SceneKit や SpriteKit、Metal が担当します（図 2.3）。

^{*4} 第2引数の `options` に指定できる `ARSession.RunOptions` は、「2.5.3 トラッキング状態をリセットする / 検出済みアンカーを削除する」で解説しています。

^{*5} 公式リファレンスでは「Surface」と表現されています。平面（Plane）だけではなく曲面も含めるという意味での「Surface」なので、本書では「面」と訳しました。

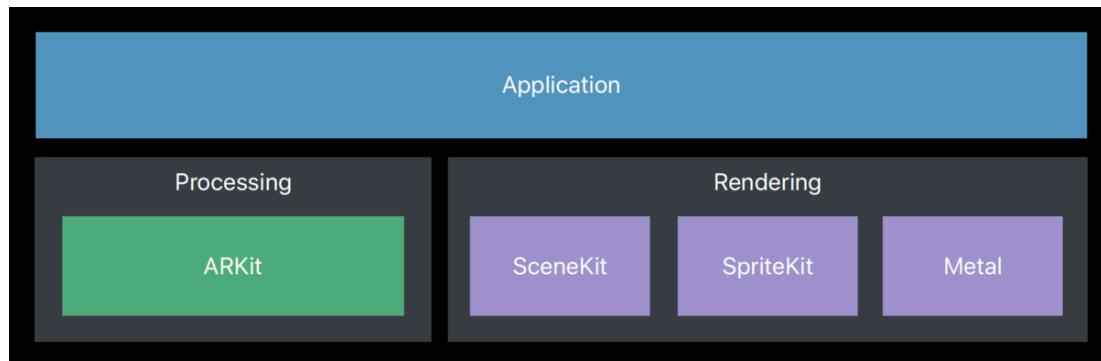


図 2.3: ARKit と他のフレームワークの関係（本リリースで差し替え）

TODO: 再作図: テキストはそのままで、配色はおまかせします。

特に、3D 空間の描画は基本的に SceneKit が担当します^{*6}。

SceneKit には、描画する 3 次元のシーン（`SCNScene`）を、UIKit ベースの UI 階層内で共存して描画するための `SCNView` という `UIView` を継承したクラスがあります。

```
class SCNView : UIView, SCNSceneRenderer, SCNTechniqueSupport
```

その `SCNView` を継承して ARKit 関連の機能を持たせたものが、`ARSCNView` クラスです。

```
class ARSCNView : SCNView
```

`ARSCNView` は、上述した `ARSession` オブジェクトをプロパティで保持しており、また「2.7 AR 空間におけるインタラクションを実現する」で後述するヒットテストの機能も持っています。すなわち `ARSCNView` は、「ARKit 関連の機能を取り扱いつつ、UIKit ベースの UI 階層内で 3 次元シーンを描画するためのクラス」です。

ちなみに、先ほど書いたコードには `ARSession` オブジェクトを生成する処理はありませんでした。この理由は、`ARSCNView` オブジェクトを生成した時点で、その `session` プロパティにはすでに `ARSession` オブジェクトが生成されて入っているためです。そのおかげで、`ARSCNView` オブジェクトを用意すればあとはセッションを `run` するだけという手軽さで AR が実現できるようになっています。

```
sceneView.session.run(configuration)
```

^{*6} Metal を用いたカスタムレンダリングについては「2.11 Metal + ARKit」で後述します。

2.3 ARKit 入門その2 - 水平面を検出する

前節では最小実装で ARKit を体験しました。次のステップでは現実世界の「水平面」を検出してみましょう。現実世界における水平面とはたとえば、地面や床、テーブル等のことです。カメラからの入力という2次元の画像から、水平な平面を「3次元的に」検出し、それをデバイスの動きに追従して認識し続けます。

平面検出の内部処理は非公開ですが、カメラから得られる毎フレームの画像から「特徴点」を取得しており、それらをシーン解析に用いていることは公式ドキュメントでも明記^{*7}されていますし、WWDC のセッション^{*8}で「より高品質なトラッキング結果を得るためにには、テクスチャのある環境で実行しましょう」と明言されてもいたので、特徴点を利用した次のような処理を行っていると考えられます。

1. 特徴点を検出する^{*9}
2. 時間的に連続した2フレーム間で特徴点のマッチングを行い対応点を得る
3. 複数の対応点の動きが平面上の動きとみなせる場合に平面として検出する

この機能をゼロから自分で実装すると大変ですが、ARKit だととても簡単です。基本的にはコンフィギュレーションにひとつ設定を追加するだけです。具体的に見ていきましょう。

(サンプルコード: 02_PlaneDetection)

2.3.1 水平面を検出するためのコンフィギュレーション

床・テーブル等の、現実世界にある「水平な平面」を検出するには、`ARWorldTrackingConfiguration` クラスの `planeDetection` プロパティに `.horizontal` をセットします。

```
worldConfig.planeDetection = .horizontal
```

そして、このコンフィギュレーションを `run(_:)` メソッドの引数に渡してセッションを開始します。

```
session.run(worldConfig)
```

`planeDetection` プロパティは `ARWorldTrackingConfiguration.PlaneDetection` 型

^{*7} たとえば `ARSCNDebugOptions` の `showFeaturePoints` のリファレンスページで、特徴点群がシーン解析の中間結果であることが明記されています。特徴点および `showFeaturePoints` については「2.6.1 特徴点を可視化する」を参照。

^{*8} "602 Introducing ARKit - Augmented Reality for iOS"

^{*9} たとえば「コーナー」として検出された点を特徴点とみなす方法があります。局所的な「見え」がユニークであり視点が変わっても見え方が変わりづらいことが良い特徴点の条件となります。

(`OptionSet` に準拠) です。平面検出はデフォルトでは無効になっているので、明示的にこのプロパティに値を指定しないと平面検出は行われません。

なお、iOS 11.0 時点では `ARWorldTrackingConfiguration.PlaneDetection` 型のオプションとしては `horizontal` のみが定義されています。垂直平面を認識するためのオプション等はまだ提供されていません。

2.3.2 平面検出に関するイベントをフックする - `ARSessionDelegate`

`ARWorldTrackingConfiguration` の `planeDetection` に値をセットしセッションを開始すると、そのセッションが有効な間は ARKit はその平面を検出し続け、更新し続けます。

ここで、`ARSessionDelegate` プロトコルへの準拠を宣言しておき、

```
class ViewController: UIViewController, ARSessionDelegate
```

`ARSession` オブジェクトの `delegate` プロパティに値をセットしておくことで、

```
sceneView.session.delegate = self
```

平面検出に関する次のタイミングで、`ARSessionDelegate` の各デリゲートメソッドが呼ばれるようになります（ここで登場する「アンカー」については次項で解説します）。

- 平面が新たに検出された（1つまたは複数のアンカーがセッションに追加された）

```
func session(_ session: ARSession, didAdd anchors: [ARAnchor])
```

- 検出済みの平面が更新された（1つまたは複数のアンカーが更新された）

```
func session(_ session: ARSession, didUpdate anchors: [ARAnchor])
```

- 検出済みの平面が削除された（1つまたは複数のアンカーがセッションから削除された）

```
func session(_ session: ARSession, didRemove anchors: [ARAnchor])
```

ARAnchor

`ARSessionDelegate` プロトコルの各メソッドの引数にある `ARAnchor` は、**AR** シーンの**3D**空間内に何らかのオブジェクトを設置するための位置・向きを表すクラスです。「アンカー (anchor)」は日本語で「錨 (いかり)」のことですが、まさに**3D**空間に物体を固定する錨としての役割を担います。

`ARAnchor` は、そのアンカーを一意に決める `UUID`型の `identifier` プロパティと、

```
var identifier: UUID { get }
```

`3D`空間における位置と回転(向き)を決める `matrix_float4x4`型(`4x4`の変換行列)の `transform` プロパティを持ちます。

```
var transform: matrix_float4x4 { get }
```

ARPlaneAnchor

`ARPlaneAnchor` は、`ARAnchor` のサブクラスです。コンフィギュレーションで平面検出を指定して `AR` のセッションを開始している場合、各デリゲートメソッドで得られる「アンカー」は、平面の位置・向きを特定するための `ARPlaneAnchor` 型で届きます。

```
func session(_ session: ARSession, didAdd anchors: [ARAnchor]) {
    // 平面検出時は ARPlaneAnchor 型のアンカーが得られる
    guard let planeAnchors = anchors as? [ARPlaneAnchor] else { return }
    print("平面を検出: \(planeAnchors)")
}
```

`ARPlaneAnchor` は、平面を表すための中心位置を示す `center` プロパティと、

```
var center: vector_float3 { get }
```

大きさ(幅・奥行き)を示す `extent` プロパティを持ちます。

```
var extent: vector_float3 { get }
```

`extent` は3次元ベクトルの `vector_float3` 型となっていますが、平面のアンカーは必ず2次元ですので、大きさは `x,z` だけで決まり、`y` の値は常にゼロとなっています。

なお、`ARPlaneAnchor` は平面のアラインメントを示す `alignment` プロパティも持ちますが、

```
var alignment: ARPlaneAnchor.Alignment { get }
```

その型である `ARPlaneAnchor.Alignment` には現状では `horizontal` しか定義されていないので、将来的にさまざまなアラインメントがサポートされるまで用途はなさそうです。

2.3.3 平面検出に関するイベントをフックする - ARSCNViewDelegate

`ARSCNView` では、AR セッションでアンカーが追加・更新・削除される際、そのアンカーに対応するノード (`SCNNode`) も追加・更新・削除されます。そのイベントをハンドリングするために `ARSCNViewDelegate` プロトコルが用意されています。

`ARSCNViewDelegate` プロトコルへの準拠を宣言しておき、

```
class ViewController: UIViewController, ARSCNViewDelegate
```

`ARSCNView` の `delegate` プロパティに値をセットしておくことで、

```
sceneView.delegate = self
```

次に示すそれぞれのタイミングで `ARSCNViewDelegate` の各デリゲートメソッドが呼ばれるようになります。

- 新しいアンカーに対応するノードがシーンに追加された

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor: ARAnchor)
```

- 対応するアンカーの現在の状態に合うようにノードが（これから）更新される

```
func renderer(_ renderer: SCNSceneRenderer, willUpdate node: SCNNode, for anchor: ARAnchor)
```

- 対応するアンカーの現在の状態に合うようにノードが更新された

```
func renderer(_ renderer: SCNSceneRenderer, didUpdate node: SCNNNode, for anchor: ARAnchor)
```

- 削除されたアンカーに対応するノードがシーンから削除された

```
func renderer(_ renderer: SCNSceneRenderer, didRemove node: SCNNNode, for anchor: ARAnchor)
```

これらのデリゲートメソッドには、アンカーに対応する `SCNNNode` オブジェクトが渡されるので、次項で解説する検出した平面の可視化や、あとで解説する仮想オブジェクトの設置など、平面検出イベントに合わせて `SceneKit` で何らかの描画を行いたい場面で役立ちます。

2.3.4 検出した平面を可視化する

平面検出時に `ARSCNViewDelegate` の `renderer(_:didAdd:for:)` の引数に渡される `SCNNNode` オブジェクトには、ジオメトリが割り当てられていません。つまり、そのままでは検出した平面は可視化されません。

しかし、検出した平面を可視化することは開発中は非常に重宝しますし、ARKit によって実現したい機能によっては任意のテクスチャで表示したいこともあるかもしれません。

検出した平面アンカーに対応する `SCNNNode` オブジェクトは、次のような手段で可視化できます。

- ジオメトリを割り当てる
- ジオメトリを持つ子ノードを追加する

平面ジオメトリを持つ子ノードを追加する実装例は次のとおりです。

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNNode, for anchor: ARAnchor) {  
    guard let planeAnchor = anchor as? ARPlaneAnchor else {fatalError()}  
  
    // 平面ジオメトリを作成……(1)  
    let geometry = SCNPlane(width: CGFloat(planeAnchor.extent.x),  
                           height: CGFloat(planeAnchor.extent.z))  
    geometry.materials.first?.diffuse.contents = UIColor.yellow  
  
    // 平面ジオメトリを持つノードを作成  
    let planeNode = SCNNNode(geometry: geometry)  
  
    // 平面ジオメトリを持つノードを x 軸まわりに 90 度回転……(2)  
    planeNode.transform = SCNMatrix4MakeRotation(-Float.pi / 2.0, 1, 0, 0)  
  
    DispatchQueue.main.async(execute: {
```

```
// 検出したアンカーに対応するノードに子ノードとして持たせる
node.addChildNode(planeNode)
})
}
```



図 2.4: 検出した平面を可視化する

コード内コメント (1) で示した行で、平面ジオメトリを作成する際に、そのサイズを `ARPlaneAnchor` の `extent` プロパティの `x` と `z` だけから決定しています。この理由については「2.3.2 平面検出に関するイベントをフックする - `ARSessionDelegate`」で解説しました。

(2) で示した行では、「平面ジオメトリを持つノード」を、`x` 軸まわりに 90 度回転させています。これは `SCNPlane` ジオメトリはデフォルトでは `x-y` 平面に対して作成されるので、それを `x-z` 平面 (= ARKit/SceneKit の座標系における水平面) に重なるようにするためです。

2.4 ARKit 入門その3 - 検出した水平面に仮想オブジェクトを置く

前節で現実世界の水平面を検出できるようになりました。次は、その検出した水平面に3Dモデルを仮想オブジェクトとして設置してみましょう。

ここまで説明で、ARSCNView はシーン (SCNScene) を持っていて、ARSCNViewDelegate で検出した平面アンカーに対応する SCNNNode まで取得できることが分かっています。ですので、あとは仮想オブジェクトとしての SCNNNode を子ノードとして載せるだけです。

仮想オブジェクトとしてどのような3Dモデルを利用するか、そのノードをどのように扱うかはすべて SceneKit の役割であり、ARKit には依存しません。言い換えると、従来からある実装方法やアセットを ARKit に利用できる、ということになります。

(サンプルコード: 03_VirtualObject)

2.4.1 3D モデルを読み込む

まず、仮想オブジェクトとして設置する3Dモデルを読み込みます。

.scn ファイルから読み込むコードは次のとおりです。

```
// シーンファイルからシーンを生成
let scene = SCNScene(named: "filename.scn", inDirectory: "directoryname")!

// 生成したシーンの rootNode 配下の子ノードを別のノードに載せ替える
let virtualObjectNode = SCNNNode()
for child in scene.rootNode.childNodes {
    virtualObjectNode.addChildNode(child)
}
```

シーンファイルからいったん SCNScene オブジェクトを生成したあとに、rootNode 配下の子ノードを別のノードに載せ替えています。これは、ARSCNView が保持しているシーンのノード階層下に置けるようにするための、従来どおりの（ARKit 登場以前からある）手法です。

SCNSceneSource クラスを利用すれば、.dae など他のファイルフォーマットから3Dモデルを読み込むこともできます。

```
let url = Bundle.main.url(forResource: "filename", withExtension: "dae")!
let sceneSource = SCNSceneSource(url: url, options: nil)!
let virtualObjectNode = sceneSource.entryWithIdentifier("modelId", withClass: SCNNNode.self)!
```

2.4.2 仮想オブジェクトとして検出した平面に置く

前項で作成した仮想オブジェクトのノードを、検出した平面アンカーに対応するノードに追加するだけです。

たとえばこの操作を、平面を新たに検出したときに呼ばれる `ARSCNViewDelegate` の `renderer(_:didAdd:for:)` で行う場合の実装は次のようにになります。

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNNode, for anchor: ARAnchor) {  
    // 平面アンカーに対応するノード上に仮想オブジェクトのノードを載せる  
    node.addChildNode(virtualObjectNode)  
}
```

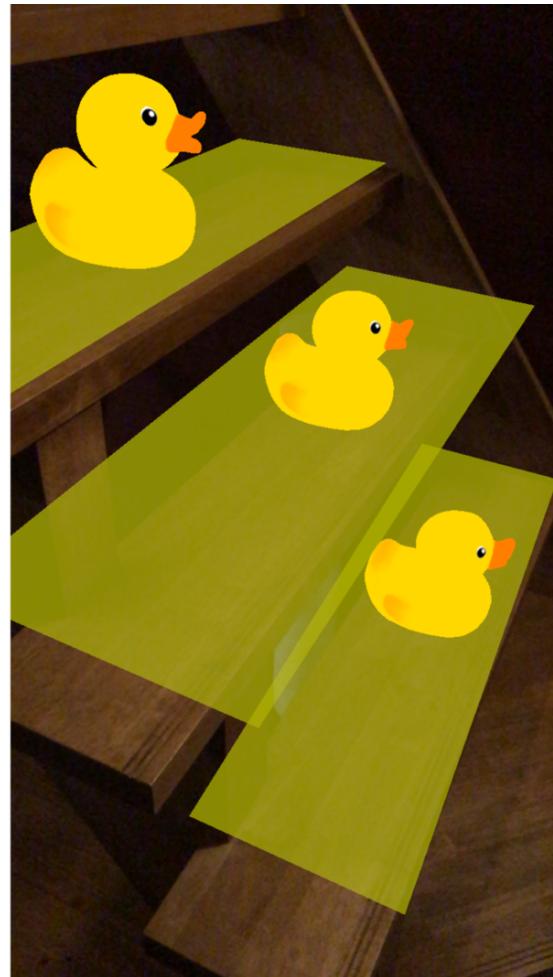


図 2.5: 現実世界の平面を検出し、仮想オブジェクトを設置

2.5 ARKit 開発に必須の機能

本節では、さらに ARKit を奥深く触っていくための、開発に便利な、あるいは適切なユーザーエクスペリエンスを提供するために不可欠な機能について解説します。

(サンプルコード: 04_DevMisc)

2.5.1 トラッキング状態を監視する

基礎編で、ARKit では ARSession を run するだけで、カメラやモーションセンサによるデータ取得が始まり、現実空間の認識とデバイスのトラッキングが始まると説明しました。

しかし、ARKit もあらゆる状況下で最高の精度を発揮できるわけではありません。何らかの理由でトラッキングが利用できなかったり、精度が下がっていることもあります。適切なユーザーエクスペリエンスを提供するためには、そういう状況も適切にハンドリングする必要があるでしょう。

そのために、ARSessionObserver プロトコルには、トラッキング状態に変化があると呼ばれるメソッドが用意されています。

```
func session(_ session: ARSession, cameraDidChangeTrackingState camera: ARCamera)
```

トラッキング情報は、第2引数 ARCamera オブジェクトの trackingState プロパティに入っています。型は ARCamera.TrackingState の enum で、次の種類が定義されています。

表 2.1: ARCamera.TrackingState の種類

case	概要
notAvailable	トラッキングを利用できない
limited(ARCamera.TrackingState.Reason)	トラッキングは利用できるが、結果の品質は疑わしい
normal	トラッキングは最適な結果を出せている

値が limited(ARCamera.TrackingState.Reason) の場合、トラッキング精度が下がっている理由を取得できます。次の種類が定義されています。

表 2.2: ARCamera.TrackingState.Reason の種類

case	概要
initializing	初期化処理中のためトラッキングが制限されている
excessiveMotion	画像ベースのトラッキングを正確に行うにはデバイスの動きが速すぎる
insufficientFeatures	カメラに映るシーン内に、画像ベースのトラッキングを行うために十分な識別可能な特徴が含まれていない

これらの情報はユーザーに適切なアクションをとってもらうためにはとても有用です。たとえば次のような活用例が考えられます。

- **initializing**
 - 初期化中であることをユーザーに伝えて初期化完了まで低精度であることを許容してもらう
 - 初期化中は AR 機能をまだ提供しない（たとえば高精度でないとユーザーエクスペリエンスを大きく損なうアプリの場合）
- **excessiveMotion**
 - ユーザーにもっとゆっくりカメラを動かすよう促す
- **insufficientFeatures**
 - 別のもっとテクスチャのある平面で試してもらうよう促す（「2.3 ARKit 入門その2 - 水平面を検出する」の序文を参照。）

2.5.2 デバッグオプションを利用する

`ARSCNView` の `debugOptions` プロパティを利用すると、デバッグに便利な様々な機能を利用できます。

実はこのプロパティは、正確には `ARSCNView` の親クラスである `SCNView` が準拠している `SCNSceneRenderer` プロトコルで定義されているもので、型は `SCNDebugOptions` です。すなわち、`ARKit` の機能ではなく、`SceneKit` の機能です。

`debugOptions` が持つオプションの一覧を示します（表 2.3）。

表 2.3: SCNDebugOptions 一覧

オプション	<code>@available</code> ^{*10}	概要
<code>showPhysicsShapes</code>	iOS 9.0	各ノードに追加されている <code>SCNPhysicsBody</code> を可視化する
<code>showBoundingBoxes</code>	iOS 9.0	各ノードのバウンディングボックスを可視化する
<code>showLightInfluences</code>	iOS 9.0	シーン内の <code>SCNLight</code> オブジェクトの位置を可視化する
<code>showLightExtents</code>	iOS 9.0	シーン内の <code>SCNLight</code> オブジェクトに影響を受ける領域を可視化する
<code>showPhysicsFields</code>	iOS 9.0	シーン内の <code>SCNPhysicsField</code> オブジェクトに影響を受ける領域を可視化する
<code>showWireframe</code>	iOS 9.0	シーン内のジオメトリのワイヤーフレームを描画する
<code>renderAsWireframe</code>	iOS 11.0	シーン内のオブジェクトをワイヤーフレームで描画する
<code>showSkeletons</code>	iOS 11.0	show skinning bones ^{*11}
<code>showCreases</code>	iOS 11.0	show subdivision creases ^{*12}
<code>showConstraints</code>	iOS 11.0	show slider constraint ^{*13}
<code>showCameras</code>	iOS 11.0	カメラを可視化する

`SCNDebugOptions` は `OptionSet` 型なので、次のように複数のオプションを同時に指定できます。

```
sceneView.debugOptions = [.showBoundingBoxes, .showWireframe]
```

これらに加えて、`ARKit` では `ARSCNView` 専用のデバッグオプション `ARSCNDebugOptions` が

定義されています。

表 2.4: ARSCNDebugOptions 一覧

オプション	概要
showWorldOrigin	ワールド座標の原点を可視化する
showFeaturePoints	ARKit が検出した 3D 特徴点群を可視化する

ARSCNDebugOptions のそれぞれのオプションは、結局のところ SCNDebugOptions 型なので、debugOptions プロパティに他の（ARSCNDebugOptions ではない）オプションと一緒に指定できます。

```
sceneView.debugOptions = [ARSCNDebugOptions.showFeaturePoints,
                           ARSCNDebugOptions.showWorldOrigin,
                           .showBoundingBoxes]
```

本節では SCNDebugOptions のうち、ARKit 利用アプリ開発でもよく使いそうなオプションをいくつか紹介します^{*14}。

showBoundingBoxes

各ノードのバウンディングボックス（境界を示す直方体）を表示します。ノードの親子関係がどうなっているか、どういう境界を持っているか等を把握するのに便利です。

^{*14} ARSCNDebugOptions.showFeaturePoints については「2.6.1 特徴点を可視化する」で解説します。

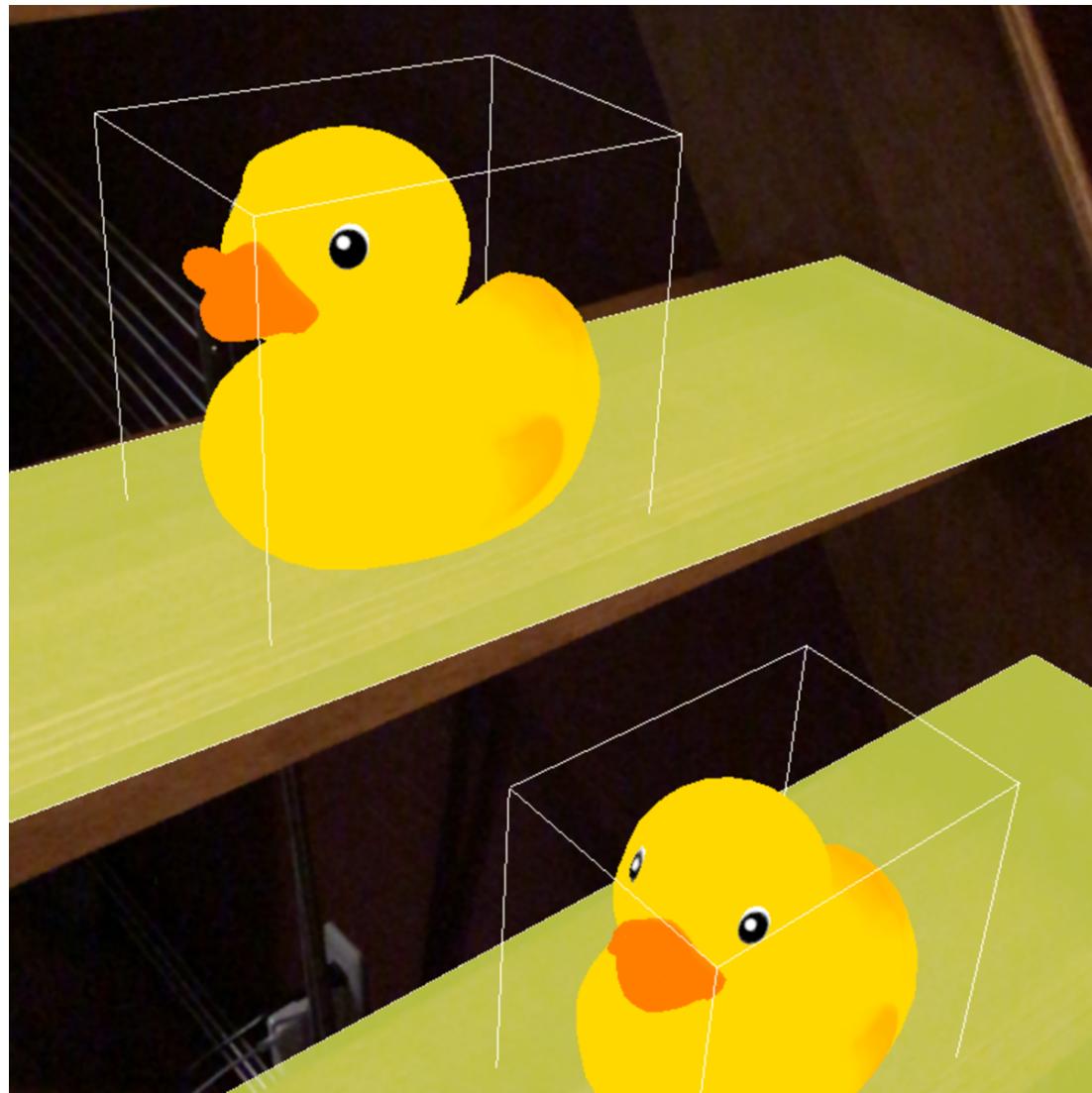


図 2.6: デバッグオプション ‘showBoundingBoxes’を利用

showWireframe

シーン内のジオメトリのワイヤーフレームを描画します。

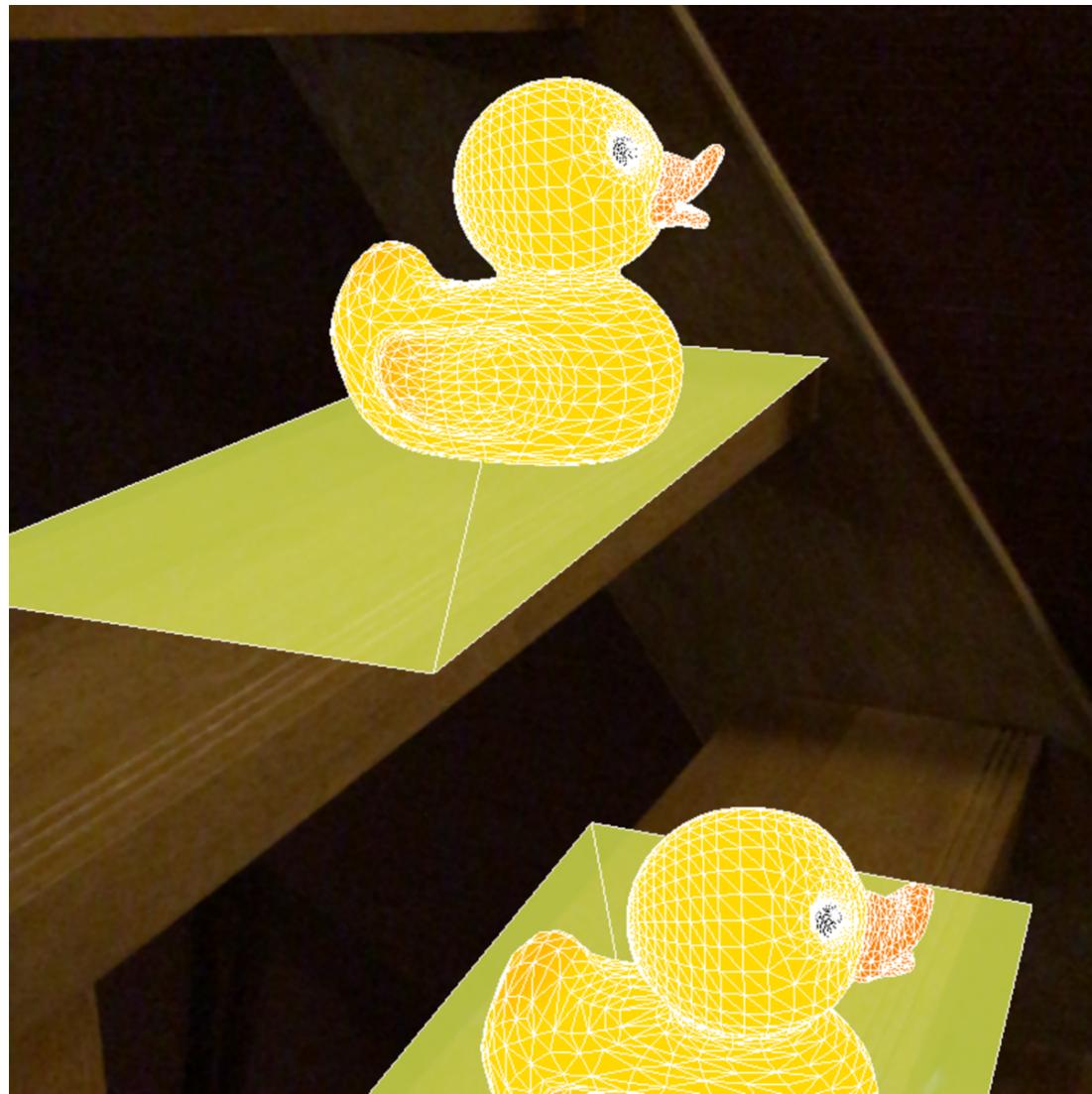


図 2.7: デバッグオプション ‘showWireframe’を利用

renderAsWireframe

シーンをすべてワイヤーフレームで描画します。`showWireframe` はジオメトリを通常どおり描画した上でワイヤーフレームを描画していたのに対し、こちらはワイヤーフレームだけで描画されます。

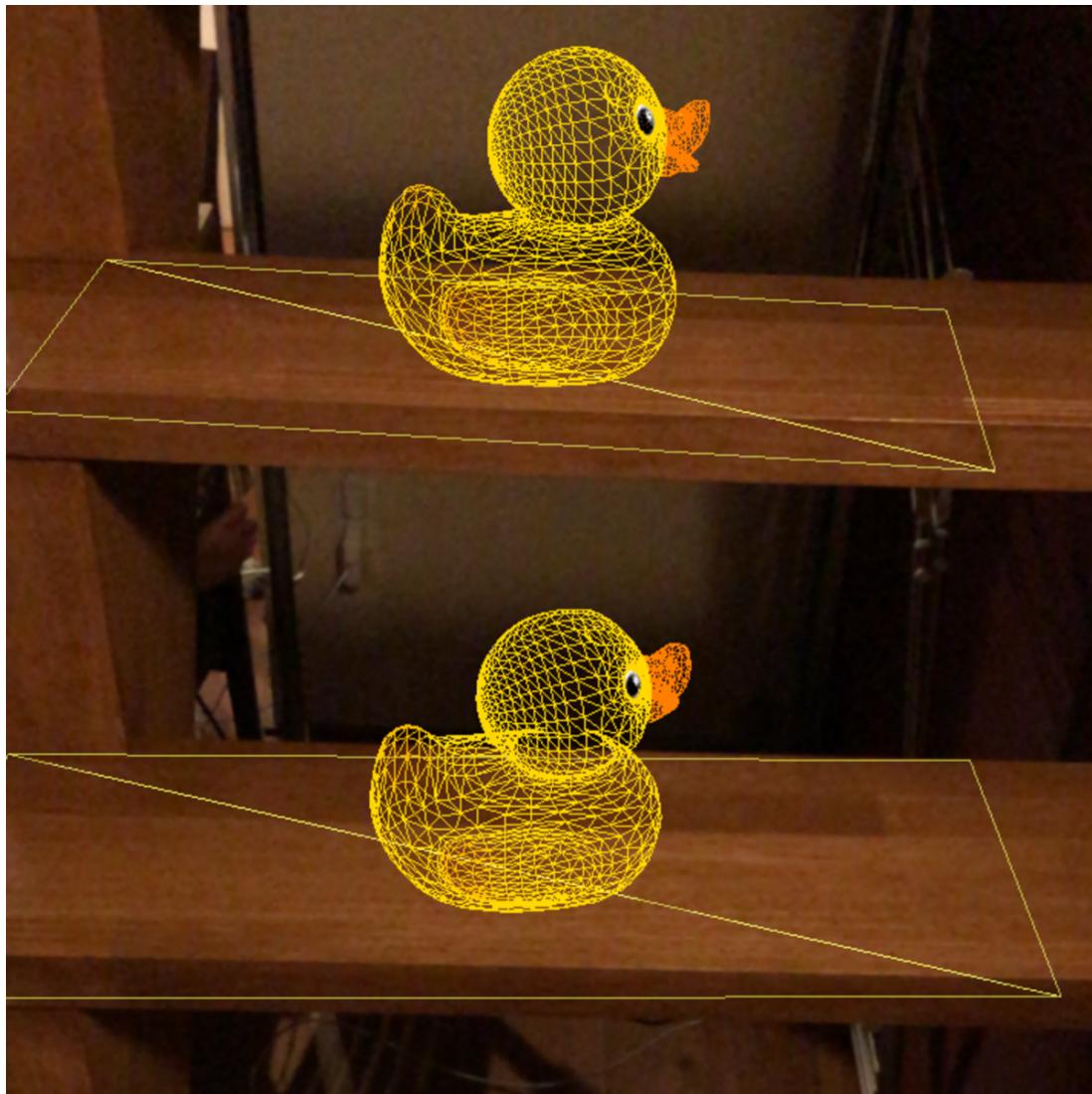


図 2.8: デバッグオプション ‘renderAsWireframe’を利用

2.5.3 トラッキング状態をリセットする / 検出済みアンカーを削除する

ARKit のトラッキング状態をリセットしたり、検出済みアンカーを削除したりしたい場合、`ARSession` の `run(_:options:)` メソッドの第 2 引数 `options` に渡す `ARSession.RunOptions` を利用します。

`resetTracking`

すでに `run` している、あるいは以前に `run` したセッションに対して `run(_:options:)` を呼んだ場合、デフォルトでは (`resetTracking` オプションを渡さなければ)、セッションはトラッキング

状態をそのまま引き継ぎます^{*15}。

逆に、前回と同様のコンフィギュレーションで `run(_:options:)` を呼ぶ際に `resetTracking` オプションを指定すると、トラッキング状態をリセットして（初期状態にして）開始できます。

```
session.run(worldConfig, options: [.resetTracking])
```

ただし、前回セッションと違うタイプのコンフィギュレーションを設定して `run(_:options:)` を呼ぶ場合、トラッキング状態は必ずリセットされた上で開始します^{*16}。

removeExistingAnchor

すでに `run` している、あるいは以前に `run` したセッションに対して `run(_:options:)` を呼んだ場合、デフォルトでは（`removeExistingAnchor` オプションを渡さなければ）、セッションは検出済みアンカーをそのまま維持します。

全ての検出済みアンカーを削除して平面認識をやりなおしたい場合は、`removeExistingAnchor` オプションを指定して `run(_:options:)` を呼びます。

```
sceneView.session.run(configuration, options: [.removeExistingAnchors])
```

これで検出済みアンカーのすべてが削除されます。アンカーが削除される際、`ARSessionDelegate` の `session(_:didRemove:)`^{*17} や、それに付随する `ARSCNViewDelegate` の `renderer(_:didRemove:for:)`^{*18} といったデリゲートメソッドも呼ばれます。

なお、`ARSession.RunOptions` は `OptionSet` に準拠しているため、次のように複数オプションの同時指定ができます。

```
sceneView.session.run(configuration, options: [.resetTracking, .removeExistingAnchors])
```

2.6 特徴点（Feature Points）を利用する

ARKit がデバイスの向きや位置をトラッキングしたり、現実世界の「面」を検出したりする際、ARKit はカメラから得られる画像から「特徴点（Feature Points）」を抽出し、その情報をトラッキ

^{*15} トラッキングの具体的な処理・アルゴリズムはブラックボックスであるため、この維持される／リセットされる「トラッキング状態」の詳細はわかりません。しかし、公式リファレンスでは `ARAnchor` オブジェクトはカメラに対する位置を維持すると書かれています。

^{*16} 公式リファレンスには、暗黙的に `resetTracking` オプションが有効になると書かれています。

^{*17} 「2.3.2 平面検出に関するイベントをフックする - `ARSessionDelegate`」参照

^{*18} 「2.3.3 平面検出に関するイベントをフックする - `ARSCNViewDelegate`」参照

ングや面の検出に使っています^{*19}。

現状では `ARWorldTrackingConfiguration` の `planeDetection` プロパティに指定できる値は `.horizontal` しかないため、現状の ARKit では垂直面や平面以外の任意の面は検出してくれません。しかし、特徴点は水平面に限らず抽出され、さらに特徴点に対してのヒットテストも行えるので、特徴点が得られた3次元座標上に仮想オブジェクトを設置するといった使い方もできます。

また、実際に ARKit を実機で動作させるとわかりますが、水平面の検出まで意外と時間がかかります。前述したとおり、特徴点の抽出は水平面の検出の前段となる処理ですので、水平面が検出されるよりもずっと早い段階で特徴点は検出され始めます。また、水平面よりも簡単に・多く検出されます。そこでこれをを利用して、水平面がなかなか検出されない場合にも、特徴点を可視化したり、特徴点の検出状況に応じてユーザーに何らかの情報を提示することで待ち時間のユーザー体験を改善するという活用方法も考えられます。

つまり、特徴点を活用することで、より柔軟に、広範に **ARKit** を応用することが可能になります。本節では、この特徴点の利用方法を解説します。

(サンプルコード: 05_FeaturePoints)

2.6.1 特徴点を可視化する

まずは「特徴点」がどのようなものか可視化してみましょう。

`ARSCNView` の `debugOptions` プロパティ^{*20}に指定できる `ARSCNDebugOptions` 型のデバッグオプションとして `showFeaturePoints` を指定します。

```
sceneView.debugOptions = [ARSCNDebugOptions.showFeaturePoints]
```

すると、ARKit が抽出した特徴点が可視化されます（図 2.9）。

*19 「2.3 ARKit 入門その2 - 水平面を検出する」の序文を参照。

*20 「2.5.2 デバッグオプションを利用する」参照



図 2.9: デバッグオプション ‘showFeaturePoints’で特徴点を可視化

なお、本デバッグオプションは、`ARWorldTrackingConfiguration` 利用時のみ有効です。

2.6.2 特徴点の ID や座標を取得する

`ARSessionDelegate` の `session(_:didUpdate:)` というデリゲートメソッドは、フレームの更新があるたびに呼ばれます。この第2引数で渡される `ARFrame` オブジェクトが、最新フレームにおけるさまざまな情報を保持しています。

```
func session(_ session: ARSession, didUpdate frame: ARFrame)
```

`ARFrame` クラスの `rawFeaturePoints` プロパティには、`ARWorldTrackingConfiguration` を利用したときの、当該フレームに含まれる特徴点の情報を保持する点群（Point Cloud）のデータが入っています。型は `ARPointCloud` です。

```
var rawFeaturePoints: ARPointCloud? { get }
```

ARPointCloud はプロパティを 3つ持つだけの非常にシンプルなクラスです。

表 2.5: ARPointCloud のプロパティ一覧

プロパティ	型	説明
<code>--count</code>	<code>Int</code>	点の数
<code>points</code>	<code>[vector_float3]</code>	各点の 3 次元座標
<code>identifiers</code>	<code>[UInt64]</code>	各点の ID

点群情報から ID が得られることからもわかるように、各特徴点はフレーム毎に別々のものとして抽出されるのではなく、フレームをまたがって同じ特徴点には同じ **ID** が割り振られます。

2.7 AR 空間におけるインタラクションを実現する

これまでの節で、現実世界の「平面」を検出し、仮想オブジェクトを設置する方法を学びました。これでカメラに映る現実世界の中に、仮想のオブジェクトを「表示する」ことができるようになったわけですが、次のステップとしては、それらの仮想オブジェクトに対してユーザーが何らかの「操作」を行えるようにしたくなるのではないかでしょうか。

たとえば買い物ができる AR アプリであれば、商品としての仮想オブジェクトをタップで選択できるようにしたいでしょう。現実の部屋に仮想の家具を設置できるアプリであれば、その家具をドラッグして動かしたり、ピンチイン／アウトで縮小や拡大ができるようにしたくなるでしょう。

このように、ユーザーのアクションに対して AR 側に何らかの応答をさせる、すなわち「インタラクション」を実現するための ARKit の機能や実装方法について解説します。

(サンプルコード: 06_ARInteraction)

2.7.1 ヒットテスト（当たり判定）を行う

ユーザーからの操作を受けて何らかの応答を行うには、「その操作がシーン内のどのオブジェクトに向けられているものなのか」を知ることは不可欠です。そのために「ヒットテスト」（当たり判定）を行います。

ユーザーが画面内のある位置をタップしたとします。UI が 2D であれば、デバイスのスクリーンも、描画されている UI もどちらも 2 次元座標で扱えるのでヒットテストのロジックは非常にシンプルですが、対象が 3D 空間である場合は、**2D** のスクリーンに対する操作を **3D** 空間にに対する操作として考える必要があるため、ヒットテストの計算は 2D のときほどシンプルではありません。

本項ではこのヒットテストの実装方法について解説します。

ARSCNView の `hitTest(_:types:)` メソッドによるヒットテスト

ARKit では 3D 空間にに対するヒットテストを簡単に行うための仕組みが用意されています。そのうちのひとつが、ARSCNView の `hitTest(_:types:)` メソッドです。

```
func hitTest(_ point: CGPoint, types: ARHitTestResult.ResultType) -> [ARHitTestResult]
```

第1引数に 2D 座標（ここではスクリーン内のタップされた座標）、第2引数には「ヒットテスト結果のタイプ」を指定します。

この「ヒットテスト結果のタイプ」の型は `ARHitTestResult.ResultType` で、次の4つのタイプが定義されています。

表 2.6: `ARHitTestResult.ResultType` のタイプと概要説明

タイプ名	結果のタイプ
<code>featurePoint</code>	ARKit によって抽出される特徴点
<code>existingPlane</code>	検出済みの平面アンカー (<code>planeDetection</code> 利用時)、ただし平面のサイズはヒットテストに考慮されません
<code>existingPlaneUsingExtent</code>	検出済みの平面アンカー (<code>planeDetection</code> 利用時)、平面のサイズが考慮される
<code>estimatedHorizontalPlane</code>	推定の水平面 ^{*21}

ここでは「検出済みの平面」をヒットテストの対象にするため、第2引数に `existingPlaneUsingExtent` を指定します。

```
let results = sceneView.hitTest(pos, types: .existingPlaneUsingExtent)
```

ここで `existingPlane` を指定しても、検出済みの平面をヒット対象にしたヒットテストを実施できます。ただしこの `existingPlane` は、平面アンカーの `extent`（大きさの情報が入っている）を使用しない、つまり無限にその平面が広がっているものとしてヒットテストが行われます。

また `ARHitTestResult.ResultType` は `OptionSet` に準拠しているので、複数タイプの指定も可能です。

```
let results = sceneView.hitTest(pos, types: [.existingPlane, .estimatedHorizontalPlane])
```

ヒットテストの結果は `ARHitTestResult` 型の配列となっています。配列になっている理由は、タッチした先には奥行きがあるため、奥行きの異なる複数のオブジェクトが同時にヒットする可能性があるためです。配列に格納された `ARHitTestResult` オブジェクトは、カメラに近い順でソートされています。

ヒットテストのタイプとして `existingPlaneUsingExtent` または `existingPlane` を指定した場合、`ARHitTestResult` の `anchor` プロパティに、ヒットした面のアンカーが入ってきます。

```
var anchor: ARAnchor? { get }
```

具体的な実装を見てみましょう。画面のタップ位置からヒットテストを行い、ヒットした平面のうちもっともカメラに近い平面に反応させるコードは次のようになります。

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    // タップ位置のスクリーン座標を取得
    guard let touch = touches.first else {return}
    let pos = touch.location(in: sceneView)

    // 平面を対象にヒットテストを実行
    let results = sceneView.hitTest(pos, types: .existingPlaneUsingExtent)

    // ヒット結果のうち、もっともカメラに近いものを取り出す
    if let result = results.first {

        // ヒットした平面のアンカーを取り出す……(1)
        guard let anchor = result.anchor else {return}

        // アンカーに対応するノードを取得
        guard let node = sceneView.node(for: anchor) else {return}

        // 平面ジオメトリを持つ子ノードを探して反応させる（略）
    }
}
```

見てのとおり、タップイベントの取得処理は UIKit を用いているだけです。平面ジオメトリを持つ子ノードの探索部分は従来からある SceneKit の実装ですし、ARKit として新たに出てきたのも、`ARSCNView` の `hitTest(_:types:)` メソッドと `ARHitTestResult` の `anchor` プロパティ以外では、コード内コメント (1) の、アンカーに対応するノードを `ARSCNView` の `node(for:)` メソッドを利用して取得している部分ぐらいです。

サンプル「ARInteraction」を実行して、画面内に水平面が検出されている状態で水平面をタップすると、水平面の色が一時的に変わります。

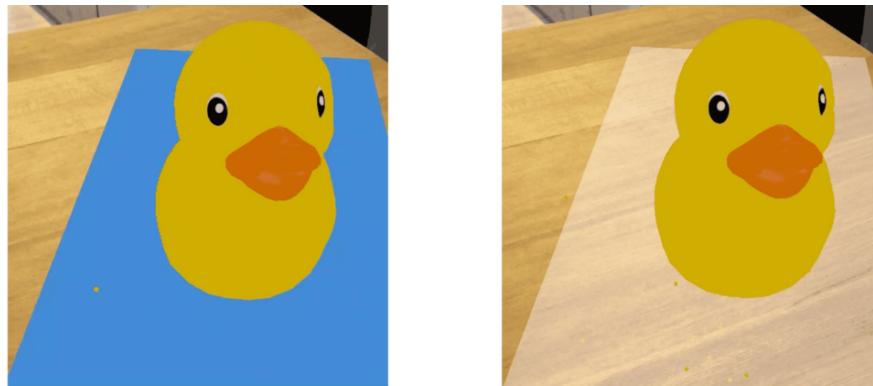


図 2.10: ARKit で検出された水平面のインタラクション

SCNSceneRenderer の hitTest(_:options:) メソッドによるヒットテスト

前項で解説した ARSCNView が備えているヒットテストの機能は、ARKit によって検出できる特徴点や平面が判定の対象となります。実は SceneKit の SCNSceneRenderer プロトコルにもヒットテストのメソッドが定義されています。ARSCNView は SCNSceneRenderer プロトコルに準拠している SCNView を継承しているので、こちらのメソッドも利用できます。

```
func hitTest(_ point: CGPoint, options: [SCNHitTestOption : Any]? = nil) -> [SCNHitTestResult]
```

SceneKit 版ヒットテストの ARKit 版との大きな違いは、SceneKit のシーディングラフ内にあるノードがヒットテスト対象になる、という点です。検出した水平面ではなく、その水平面上に設置した仮想オブジェクトのノードに対してインタラクションを可能にしたい場合には、こちらの SceneKit のヒットテスト機能の方が向いているといえます。

実装は次のようにになります。

```
// ヒットテストのオプション設定
let hitTestOptions = [SCNHitTestOption: Any]()

// ヒットテスト実行
let results: [SCNHitTestResult] = sceneView.hitTest(pos, options: hitTestOptions)

// ヒットしたノードに合致する仮想オブジェクトはあるか、再帰的に探す
guard let virtualNode = virtualObjectNode else {return}
for result in results {
    for child in virtualNode.childNodes {
        guard child == result.node else {continue}
        virtualNode.react()           // 該当するノードに反応させる
        return
    }
}
```

上のコードでは、画面のタップ位置からヒットテストを実行し、ヒットしたノードに合致する仮想オブジェクトに反応させる、という処理を行っています。

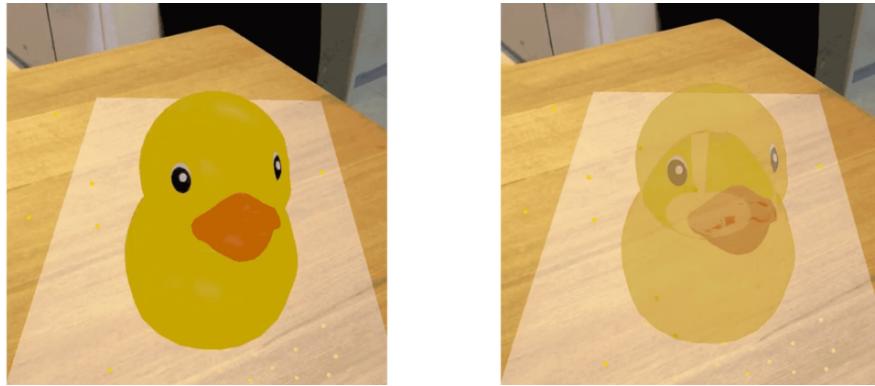


図 2.11: SceneKit ノードのインタラクション

特徴点に対するヒットテスト

ARSCNView の `hitTest(_:types:)` の第 2 引数に指定できる「ヒットテスト結果のタイプ」に `featurePoint` を指定すると、特徴点へのヒットテストが行われるようになります。

```
let results = sceneView.hitTest(pos, types: [.featurePoint])
```

結果の型は `ARHitTestResult` ですが、特徴点に対してヒットした場合の結果には `anchor` プロパティに値が入りません。結果タイプに `existingPlaneUsingExtent` や `existingPlane` を指定した場合とは、その点が異なります。

また、ヒットテスト実行時に複数の結果タイプを指定した場合は、

```
let results = sceneView.hitTest(pos, types: [.featurePoint, .existingPlane])
```

次のように `ARHitTestResult` の `type` プロパティを見て「どのタイプのオブジェクトにヒットした結果か」(平面なのか特徴点なのか) を判別できます。

```
// 結果のタイプの判定
switch result.type {
case .featurePoint:
```

```
// 特徴点へのヒット
print(result.anchor as Any)      // 特徴点にはアンカーはないので nil になる
case .existingPlane:
    // 検出済み平面へのヒット
    print(result.anchor as Any)      // ヒットした平面のアンカーが得られる
default:
    break
}
```

2.7.2 デバイスの移動に対するインタラクション

ARKit は現実空間におけるデバイスの位置や向きをトラッキングします。現実空間内におけるデバイスの移動や向きの変化に追従して、ARKit の空間内におけるカメラの位置や向きを更新します。

この情報を利用すると、ユーザーが iPhone を持って移動したり、iPhone を動かしたりといったデバイスの動きに合わせて仮想オブジェクトにアクションを取らせることができます。

たとえば、WWDC17 のデモにあったように、空間内に設置した仮想のキャラクターにずっとユーザーの方を向かせる、といったことが可能です。

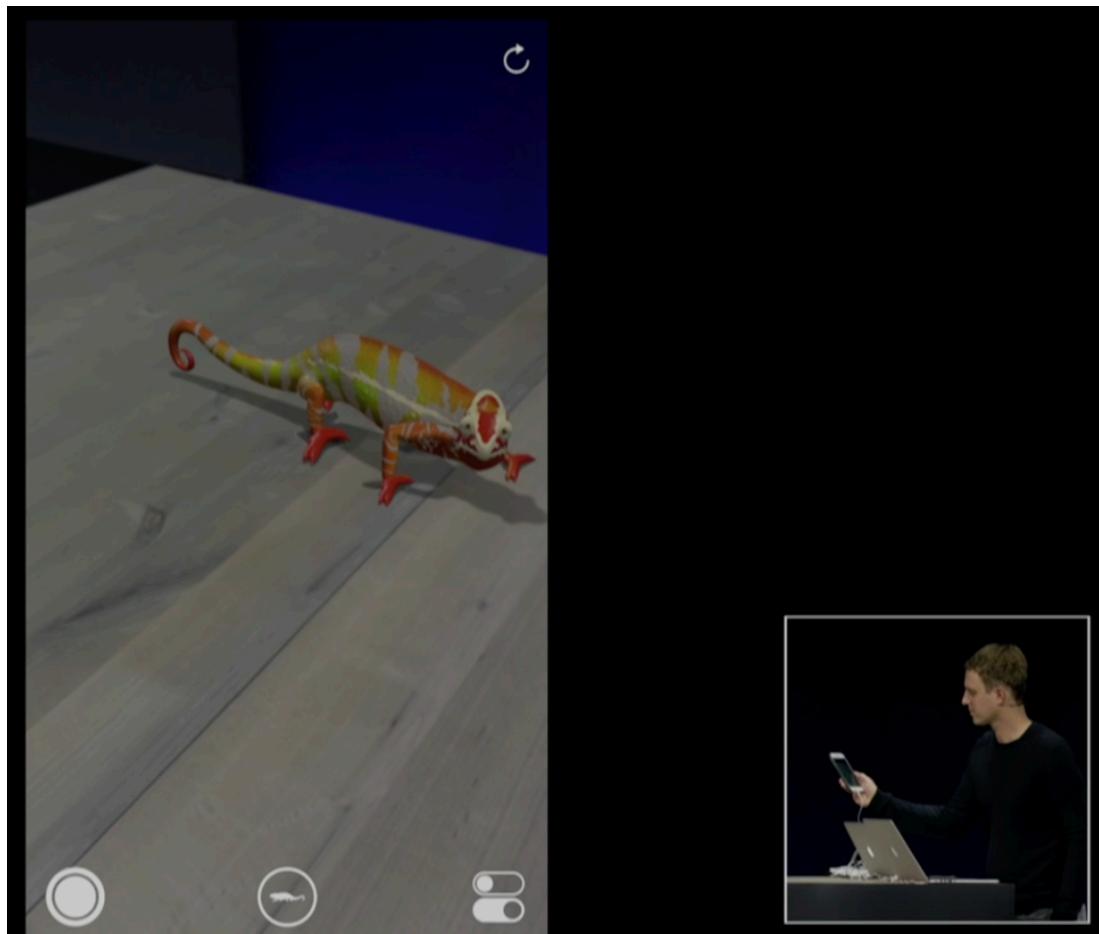


図 2.12: WWDC17 のデモより。仮想のカメレオンが常にユーザーの方を向く

最新フレームにおけるカメラの情報を取得する

「2.6.2 特徴点の ID や座標を取得する」でもすでに触れましたが、ARKit ではカメラから新しい入力があるたびに `ARSessionDelegate` の `session(_, didUpdate:)` メソッドが呼ばれます。その第2引数に渡される `ARFrame` を調べると、最新フレームが持つさまざまな情報を取得できます。

```
func session(ARSession, didUpdate: ARFrame)
```

たとえば `camera` プロパティだと、そのフレームにおけるカメラ情報を保持する `ARCamera` 型のオブジェクトにアクセスできます。

```
let camera = frame.camera // ARCamera
```

カメラのワールド変換行列から、カメラの位置・向きを取得する

ARCamera は matrix_float4x4 型の transform プロパティを持っており、この 4x4 の行列がワールド座標系^{*22}におけるカメラの変位と回転（つまり位置と向き）を表します。

```
let transform = camera.transform // matrix_float4x4
```

この 4x4 行列は、ワールド座標系における変換行列という意味で「ワールド変換行列」と呼ばれます。このワールド変換行列から、ARKit の空間内における位置を得るには、次のように行列の要素を取り出して 3 次元ベクトルを作成します。

```
let pos = SCNVector3Make(transform.columns.3.x, transform.columns.3.y, transform.columns.3.z)
```

もしくは SceneKit に用意されている 4x4 行列を表す型 SCNMatrix4 にいったん変換してから、位置を表す 3 次元ベクトルを得る方法もあります。

```
let mat = SCNMatrix4(transform)
let pos = SCNVector3(mat.m41, mat.m42, mat.m43)
```

ワールド変換行列から向きを計算することができますが、もっと手軽に、ARCamera にはカメラの向きをオイラー角（Euler angles）として取得できる eulerAngles プロパティが用意されています。型は vector_float3 です。

```
let angles = camera.eulerAngles // vector_float3
```

仮想オブジェクトが常にカメラの方を向くようにする

さて、ここで「デバイスの移動や向きの変化に対するインタラクション」の例として、現実空間内に設置した仮想オブジェクトが常にユーザーの方を向くように実装してみましょう。

iOS 11 で SCNNNode に look(at:) メソッドが追加されました。このメソッドを使うと、引数に渡したワールド座標に向くように、SCNNNode オブジェクトの orientation を更新してくれます。

^{*22} 3D プログラミングの世界では、ワールド座標以外に、ローカル座標、カメラ座標といった座標系が存在するためこのような呼び方がされています。聞き慣れない方は単純に「3D 空間内における x,y,z の 3 軸で表される座標」として読んでください。

```
// 仮想オブジェクトをカメラの方に向ける  
virtualNode.look(at: pos)
```

便利なメソッドですが、単純にこれを使用するだけだと仮想オブジェクトが x, y, z の 3 軸すべてについて回転が起きる可能性があり、重力方向について傾いてしまう等、水平面に対して設置しているオブジェクトにとっては不自然な結果になってしまう場合があります。

そこで、重力方向に対しては回転を加えないよう、y 軸に沿ってだけ回転させることにします。仮想オブジェクトのワールド座標とカメラのワールド座標の成すベクトルの、x-z 平面における角度を計算し、その角度を打ち消すように y 軸に対する回転を与えます。

```
// カメラの位置を計算  
let mat = SCNMatrix4(frame.camera.transform)  
let cameraPos = SCNVector3(mat.m41, mat.m42, mat.m43)  
  
// 仮想オブジェクトとカメラの成すベクトルの、x-z 平面における角度を計算  
let vec = virtualNode.position - cameraPos  
let angle = atan2f(vec.x, vec.z)  
  
// 仮想オブジェクトの rotation に反映  
virtualNode.rotation = SCNVector4Make(0, 1, 0, angle + Float.pi)
```

これでデバイスを持って移動しても、仮想オブジェクトがずっとユーザーの方を向くようになります。

■コラム: SCNBillboardConstraint を用いる方法

ARKit におけるカメラの取り扱い (ARCamera) の解説のため、「2.7.2 デバイスの移動に対するインタラクション」ではベクトルや角度の計算を行いましたが、実は「特定軸の回転だけを許可しつつ、ノード (SCNNode) を常にカメラの方に向かせる」という目的だけでいえば、SCNBillboardConstraint を用いてもっと簡単に実現できます。

```
let billboardConstraint = SCNBillboardConstraint()  
billboardConstraint.freeAxes = SCNBillboardAxis.Y  
virtualNode.constraints = [billboardConstraint]
```

これで仮想オブジェクトのノードに、Y 軸に対する回転だけを自由にして、常にカメラの方を向くような制約が付加されます。

2.8 アプリケーション実装例1：現実空間の長さを測る

ここまでで、AR機能を実現するための要素となる実装方法を解説してきました。ここからはARKitを用いた具体的なアプリケーションの実装手順を解説します。

ARKitが発表されて間もない2017年6月25日、YouTubeにARKitを使ったアプリのデモ動画が投稿され、話題となりました。

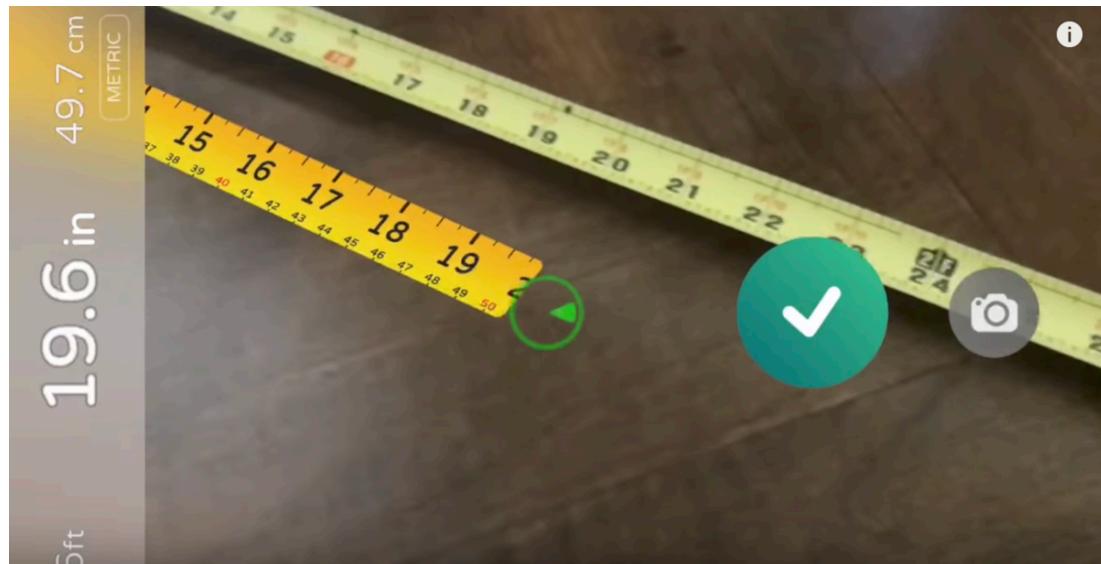


図2.13: AR Measure App Demo - Augmented reality tape measure

ARKitを用いて現実空間の長さを測定している動画です。静止画では何をしているか分かりづらいかもしれません、図2.13において上側にあるのが現実のメジャー（巻尺）で、下側にあるのがARKitで計測した長さに合わせて描画している仮想メジャーです。数値がほぼ一致しており、かなり良い精度で測定できていることがわかります。

このようなアプリが実現可能なのは、ARKitは現実のスケール（距離や大きさ）も推定しているためです。本節では、ARKitの関連する仕様について解説するとともに、このような現実空間の長さを測るアプリの実現方法を検討してみます。

（サンプルコード: 07_ARMeasure）

2.8.1 ARKitにおける座標と現実のスケール

ARKitは現実空間の平面を認識して、3D空間における座標に落とし込みます。前述したとおり現実のスケールも推定するため、そのARKitの3次元空間における座標のスケールは、推定した現実のスケールを反映しています。その単位はメートルです。

たとえば、座標(0, 0, 0)にあるアンカーAと、座標(0, 0, 1)にあるアンカーBがARKitによって検出されたとすると、ARKitは、アンカーBはアンカーAに対してz方向に1メートル離れた位置にある、と推定していることになります。

同様に、「ARSCNView の hitTest(_:types:) メソッドによるヒットテスト」で解説したヒットテストで得られる ARHitTestResult の distance プロパティの値も、ヒットしたオブジェクトのカメラからの距離をメートルで表しています。

2.8.2 現実空間における二点間の距離

さて、冒頭で紹介したメジャーのアプリは、距離を計測したい始点と終点をそれぞれ画面タップで指定するという操作方法になっているようです。

ということは、ARKit のヒットテストの機能を用いれば、**3D** 空間における**2** 点間の座標の距離を計算する方法で同様のことが行えそうです。

具体的なコードは次のとおりです。

```
// プロパティとして定義
var startNode: SCNNNode? // 計測したい距離の始点を示すノード
var endNode: SCNNNode? // 計測したい距離の終点を示すノード
```

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    // タップ位置のスクリーン座標を取得
    guard let touch = touches.first else {return}
    let pos = touch.location(in: sceneView)

    // 平面を対象にヒットテストを実行
    let results = sceneView.hitTest(pos, types: [.existingPlane])

    // 最も近い（手前にある）結果を取得
    guard let result = results.first else {return}

    // ヒットした位置を計算する
    let hitPos = result.worldTransform.position()

    // 始点はもう決まっているか？
    if let startNode = startNode {
        // 終点を決定する（終点ノードを追加）
        endNode = putSphere(at: hitPos, color: UIColor.green)
        guard let endNode = endNode else {fatalError()}
    }

    // 始点と終点の距離を計算する
    let distance = (endNode.position - startNode.position).length()

    // 始点と終点を結ぶ線を描画する
    lineNode = drawLine(from: startNode, to: endNode, length: distance)

    // ラベルに表示
    statusLabel.text = String(format: "Distance: %.2f [m]", distance)
} else {
    // 始点を決定する（始点ノードを追加）
    startNode = putSphere(at: hitPos, color: UIColor.blue)
```

```
    }
}
```

可視化のための処理が入っているため一見複雑に見えるかもしれません。しかし「2点間の距離計算」に関する本質的な実装だけを見ると、とてもシンプルになっています。

- 画面タップ位置についてヒットテストを行う：ヒットテスト結果の3次元座標を「始点」とする
- 画面タップ位置についてヒットテストを行う：ヒットテスト結果の3次元座標を「終点」とする
- 始点・終点の2点間の距離を計算する

たったこれだけです。「ARSCNView の hitTest(_:types:) メソッドによるヒットテスト」でのヒットテストと違う点は、hitTest メソッドの types 引数に .existingPlaneUsingExtent ではなく、.existingPlane を指定している点です。認識した平面の extent 範囲よりも広い範囲で計測できるようにするためにです。

ちなみに3次元座標を表す SCNVector3 型の計算をシンプルに記述するために、サンプルでは次のように extension および演算子を定義しています。

```
extension SCNVector3 {

    func length() -> Float {
        return sqrtf(x * x + y * y + z * z)
    }

    func - (left: SCNVector3, right: SCNVector3) -> SCNVector3 {
        return SCNVector3Make(left.x - right.x, left.y - right.y, left.z - right.z)
    }
}
```

アプリの実行画面を図 2.14 に示します。

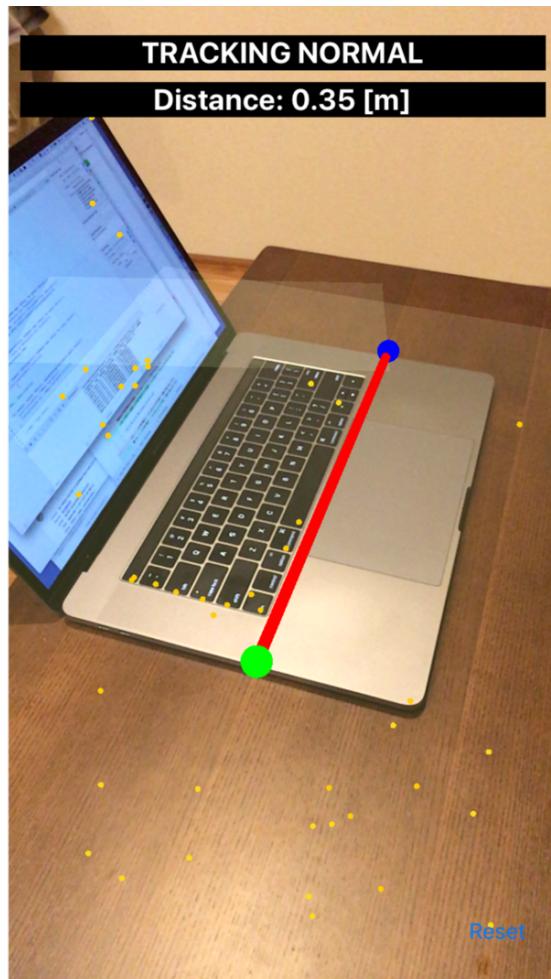


図 2.14: ARKit を利用した距離の計測結果

ARKit による 2 点間の距離の推定結果は「0.35m」と出ています。計測対象である Macbook Pro (15-inch,2016) の横幅は、公式情報によると「34.93cm」とのことですので、正確に計測できていることが分かります。

2.9 アプリケーション実装例 2：空中に絵や文字を描く

「空中に絵や文字を描く」というのは AR の分野でも典型的なアプリケーションのひとつではあります、デモとしては非常にわかりやすくインパクトがあります。トラッキングの精度が優れていないと描いた線が意図どおりに配置されずバラバラになってしまないので、ARKit の精度（トラッキング品質）を示す格好の題材でもあります。

本節では「iOS デバイスを動かして空中に線を描く」アプリケーションの実装方法を示します。
(サンプルコード: 08_ARDrawing)

2.9.1 実装方針

デバイスを動かして空中に線を書くためには、シンプルに「各時点でのデバイスの位置（ワールド座標）を繋いで線を構築する」という方針が考えられます。

この方針を具体的な実装に落とし込むと次のようになります。

- ARSessionDelegate の session(_:didUpdate:) または SCNSceneRendererDelegate (ARSCNViewDelegate が継承している) の renderer(_:updateAtTime:) をデバイスの位置を取得するタイミングとする
- 「各時点でのデバイスの位置」としてスクリーンの中心座標をワールド座標に変換したもの要用いる
- 各時点でのワールド座標を頂点として SCNGeometrySource、SCNGeometryElement を用いて線としてのカスタムジオメトリを構築する

何をデバイス位置としてどう計算するか、またそれらを頂点として線をどう描画するかは他にも多くの方法が考えられますが^{*23}、本節ではこの方針に沿って実装していきます。

2.9.2 スクリーンの中心座標をワールド座標に変換する

前述の方針通り、デバイスの位置は「スクリーンの中心座標をワールド座標に変換したもの」を用います。ここではその変換方法として、SCNSceneRenderer プロトコル (ARSCNView の親クラス SCNView が準拠している) に定義されている、2 次元のスクリーン座標をワールド座標に変換するための unprojectPoint(_:) メソッドを利用します。

```
func unprojectPoint(_ point: SCNVector3) -> SCNVector3
```

引数にはスクリーンにおける座標を渡すのですが、その型は 3 次元ベクトルになっています。これは、3 次元空間における座標を決めるためには、スクリーンに対する奥行き方向も指定しないと候補が無限に存在してしまうからです。

引数に渡す SCNVector3 型の z 値には 0.0~1.0 を指定します。0.0 を指定するとレンダラの視錐台における手前側のクリッピング平面上にある座標が得られ、1.0 だと遠方側のクリッピング平面にある座標が得されることになります。

スクリーンの中心座標を取得して、ワールド座標に変換するまでの実装を次に示します。

```
// スクリーンの中心座標を取得
let screenBounds = UIScreen.main.bounds
```

^{*23} たとえばデバイス位置として各フレームにおけるカメラのワールド座標を利用する、線の描画に Metal のシェーダを用いる等。

```
let centerPos = CGPoint(x: screenBounds.midX, y: screenBounds.midY)

// 3次元ベクトルにする
let centerVec3 = SCNVector3Make(Float(centerPos.x), Float(centerPos.y), 0.99)

// unproject (ワールド座標に変換) する
let worldPos = sceneView.unprojectPoint(centerVec3)
```

2.9.3 頂点座標の配列から、線としてのカスタムジオメトリを構成する

ここで「カスタム」なジオメトリと呼んでいるのは、SceneKit が提供している SCNGeometry サブクラス一たとえば SCNPlane（平面）や SCNSphere（球）以外の、自分で定義する形状のことです。

本項では各時点でのデバイス位置（ワールド座標）を頂点として蓄積していく、動的に線としてのカスタムジオメトリを構成する実装方法を示します。

たとえば頂点座標の SCNVector3 とそのインデックスを蓄積していくための配列を次のように定義しておき、

```
private var vertices: [SCNVector3] = []
private var indices: [Int32] = []
```

頂点とそのインデックスをいくつか格納した後、次のように SCNGeometrySource、SCNGeometryElement を生成します。

```
let source = SCNGeometrySource(vertices: vertices)
let element = SCNGeometryElement(indices: indices, primitiveType: .line)
```

ここで、SCNGeometryElement の初期化の際に第2引数に渡しているプリミティブタイプ.line は、第1引数に渡されたインデックスの配列が、線を構成するものであることを指定するものです。

これら2つのオブジェクトを用いて、SCNGeometry を生成できます。

```
let geometry = SCNGeometry(sources: [source], elements: [element])
```

ただし、頂点を結んで線としてのカスタムジオメトリを構成する場合、その線は「太さ」を持ちません。そこで、三角形を2つ組み合わせて太い線を構成するようにします。

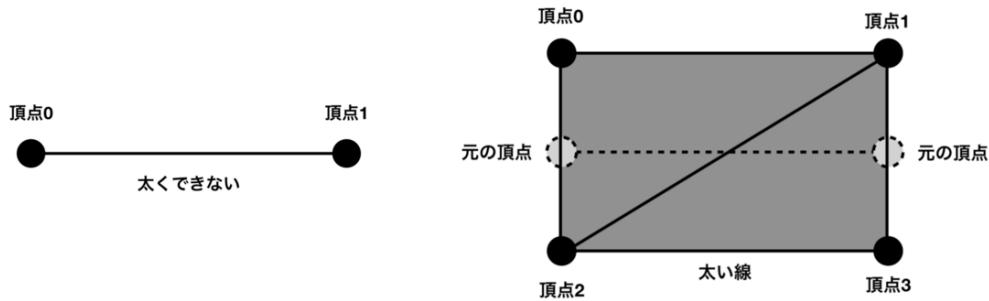


図 2.15: (左) 2 頂点で線を構成 (右) 三角形を 2 つ組み合わせて線を構成 (本リリースで差し替え)

TODO: 再作図: 本書全体の作図のトーンで清書していただければと存じます。

この場合は `SCNGeometryElement` の初期化の際に第 2 引数に `.triangleStrip` を渡します。

```
let element = SCNGeometryElement(indices: indices, primitiveType: .triangleStrip)
```

2.9.4 他の実装のポイント

デバイス位置の取得タイミング

`ARSCNViewDelegate` プロトコルは `SCNSceneRendererDelegate` プロトコルを継承しています。

```
ARSCNViewDelegate <SCNSceneRendererDelegate, ARSessionObserver>
```

デバイス位置の取得タイミングとして、本節ではこの `SCNSceneRendererDelegate` プロトコルに定義されている `renderer(_:updateAtTime:)` メソッドを用います。このデリゲートメソッドは `ARSCNView` において毎フレーム、アニメーションや物理計算の評価前に呼ばれるため、公式リファレンスではレンダリンググループに対する更新処理をここで行うよう教示しています。

```
func renderer(_ renderer: SCNSceneRenderer, updateAtTime time: TimeInterval)
```

精度を安定させる

描いた線が空間に安定するように、トラッキング品質がベストではない間は描画を許可しないようになります^{*24}。

```
private func isReadyForDrawing(trackingState: ARCamera.TrackingState) -> Bool {  
    switch trackingState {  
        case .normal:  
            return true  
        default:  
            return false  
    }  
}
```

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
    guard let frame = sceneView.session.currentFrame else {return}  
    guard isReadyForDrawing(trackingState: frame.camera.trackingState) else {return}
```

以上のように実装したアプリの実行結果を図 2.16 に示します。各時点でのデバイス位置から動的に線としてのカスタムジオメトリを構成し、現実の 3D 空間に設置できていることがわかります。

^{*24} トラッキング状態については「2.5.1 トラッキング状態を監視する」で解説しています。

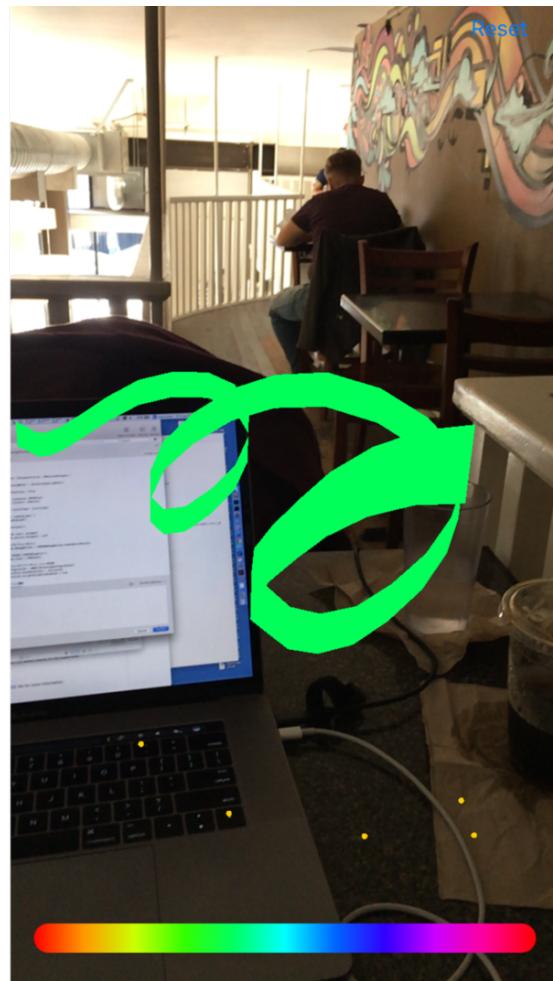


図 2.16: ARKit を利用して空中に文字を描く

2.10 アプリケーション実装例3: Core ML + Vision + ARKit

iOS に AR フレームワークが標準搭載されることの大きな魅力のひとつは、これまでの 10 年の iOS の歴史の中で脈々と整えられてきた豊富な他のフレームワークと連携させられることでしょう。その中でも、ARKit と同様に iOS デバイスのカメラから得た画像を入力として利用できる Core ML・Vision とは非常に組み合わせやすく、なおかつ AR & 機械学習というホットな技術の組み合せだけにデモとしてのインパクトもあります。

本節では、「Core ML+Vision で物体認識を行い、その物体に ARKit で仮想のタグを付ける」アプリケーションの実装方法を解説します。

(サンプルコード: 09_ARObjectDetection)

2.10.1 実装方針

本例で実現したいことは、ARKit の 3D 空間に Core ML による物体認識結果をテキストノードとして設置することです。この際、「Core ML による認識結果を、どうやって 3D 空間の座標に紐付けるか」がポイントとなります。

たとえば物体認識の結果に対象の物体を囲む矩形の座標（2D 画像内の座標）が入っている場合、簡単な方法だとその中心座標を求め、ARKit でヒットテストを行いヒットした位置（3D 空間内の座標）にテキストノードを設置する、という方法が考えられます。

本節では Core ML のモデルとして Apple が配布している `Inceptionv3.mlmodel` を利用します。このモデルではその結果に認識した物体の座標が含まれません。

そこで、`VNCoreMLRequest` の `imageCropAndScaleOption` プロパティを用いて、入力画像の中心からクロップしたものを見出し、認識に利用するよう指定します。

```
request.imageCropAndScaleOption = .centerCrop
```

認識結果を設置する 3D 空間における座標には、「スクリーン座標の中心に対するヒットテスト結果」を用いることにします。画面中心にあるものを認識したはずなので、画面中心に見えているオブジェクト（実際には特徴点または平面）上に認識結果を示すテキストノードを設置しよう、という方針です。

2.10.2 Core ML・Vision・ARKit 連携のポイント

Core ML の詳細は本書の「第3章 Core ML」を参照しつつ、ここでは ARKit との連携のポイントとなる実装のみ解説します。

毎フレームのカメラからの入力画像データへのアクセス

Core ML へ入力画像として渡すために、ARKit がカメラから毎フレーム取得している画像データを取得します。この画像データは `ARFrame` の `capturedImage` プロパティより得られます。

```
var capturedImage: CVPixelBuffer { get }
```

型は `CVPixelBuffer` で、このバッファにカメラから得られたピクセルデータが入っています。

`ARFrame` オブジェクトは ARKit のセッションにおいてフレームの更新があるたびに呼ばれる `ARSessionDelegate` の `session(_:didUpdate:)` メソッドの第2引数から得られるので、ここからピクセルバッファを取得することで毎フレームの処理を行えます。

```
func session(_ session: ARSession, didUpdate frame: ARFrame) {  
    let pixelBuffer = frame.capturedImage
```

ARSCNView を利用している場合は、ARSCNViewDelegate の（実際に SCNSceneRendererDelegate の）renderer(_:updateAtTime:) メソッドから最新フレームの ARFrame オブジェクトを取得し、ピクセルバッファにアクセスする方法もあります。

```
func renderer(_ renderer: SCNSceneRenderer, updateAtTime time: TimeInterval) {  
    guard let frame = sceneView.session.currentFrame else {return}  
    let pixelBuffer = frame.capturedImage
```

Core ML の認識を実行する

ARKit から取得した毎フレームの画像データを、Core ML+Vision に渡します。

VNImageRequestHandler の初期化時に、引数に入力画像データを渡す必要がありますが、前項で解説した方法で得た CVPixelBuffer オブジェクトをそのまま渡せます。

```
let handler = VNImageRequestHandler(cvPixelBuffer: imageBuffer)
```

VNImageRequestHandler が作成できたら、あとは perform() メソッドを呼ぶだけです。

```
try? handler.perform([request]) // request は VNCoreMLRequest オブジェクト
```

Core ML の認識結果をテキストノードとして設置する

Inceptionv3 モデルを利用した場合、認識結果は VNClassificationObservation オブジェクトとして得られます。

結果が得られたら、ARSCNView の hitTest(_:types:) メソッドによるヒットテスト^{*25}を、スクリーンの中心座標を対象に行います。そして、ヒットテストの結果を元に SCNText をジオメトリに持つノードを設置します。

```
// 平面、特徴点を対象にヒットテストを実行
```

^{*25} 「2.7.1 ヒットテスト（当たり判定）を行う」参照。

```
let results = sceneView.hitTest(screenCenterPos, types:  
    [.existingPlaneUsingExtent, .featurePoint])  
  
// 手前にある結果を取り出す  
if let result = results.first {  
    // SCNText ジオメトリを持つノード  
    let tagNode = TagNode()  
    // ヒットテスト結果のワールド変換行列をセット  
    tagNode.transform = SCNMatrix4(hitTestResult.worldTransform)  
    // シーンに設置  
    sceneView.scene.rootNode.addChildNode(tagNode)  
}
```

実際のサンプルではもう少し細かい制御をしていますが、ポイントは以上です。実行すると、図 2.17 のように Core ML による認識結果が ARKit により現実空間にタグ付けされます。

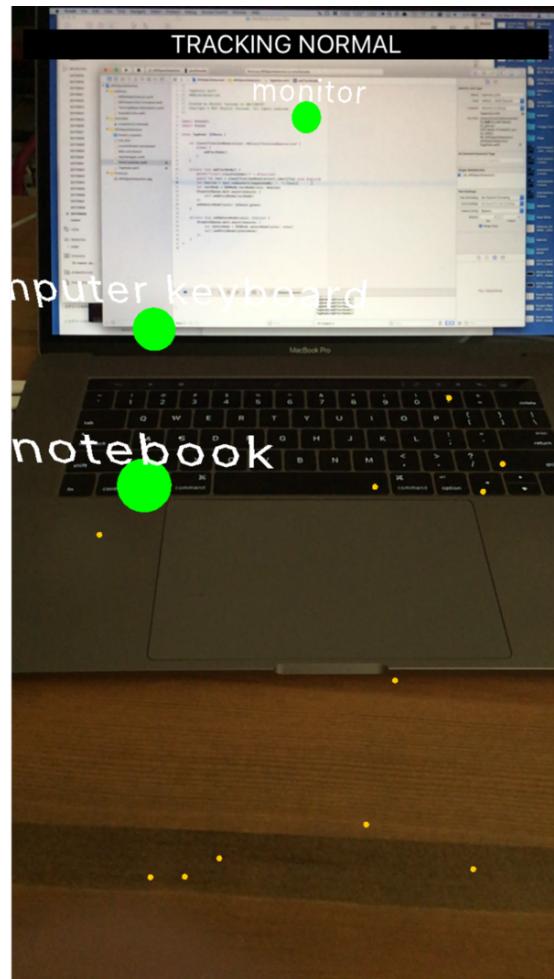


図 2.17: 認識結果のタグ付け

2.11 Metal + ARKit

『ARKit はコアな処理のみを行い、実際の描画は SceneKit や SpriteKit、Metal が担当する』ということを本章の冒頭で述べました。Metal を用いたカスタムレンダリングについては Apple からも "Displaying an AR Experience with Metal"^{*26} というドキュメントが公開されています。

しかしこのドキュメントでいう「Metal を用いたカスタムレンダリング」は、「SceneKit や SpriteKit を用いる代わりに」、自前で **Metal** を用いてレンダリング処理を実装することもできる」という話で、つまり SceneKit/SpriteKit に相当する 3D/2D レンダリングエンジンを自作することを意味します。たとえば SceneKit では実現できない機能やパフォーマンスを達成したいような場合には有効な選択肢ですが、実装の量も難度も非常に高くなってしまいます。

これとは別に、すべてを Metal で代替するのではなく、**3D 空間の扱い**は基本的に **SceneKit** を利用し、シーン内ノードのマテリアルの描画に **Metal** を利用する、という選択肢もあります。

たとえば図 2.18 のような表現が可能です。この例では ARKit を用いて現実空間の水平面上に設置した仮想オブジェクト（のノードのマテリアル）を、Metal のシェーダを用いて描画しています。

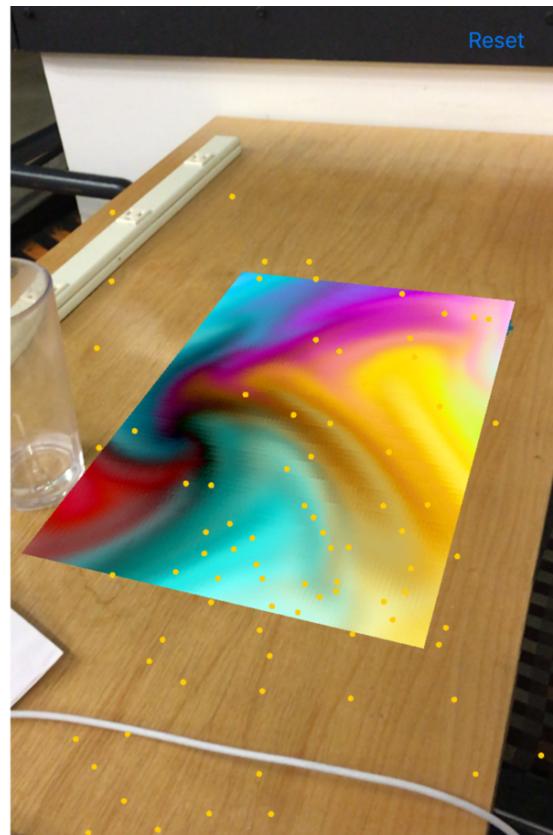


図 2.18: SCNNode のマテリアルの描画に Metal を利用

^{*26} Displaying an AR Experience with Metal

もっと AR らしく現実空間と仮想オブジェクトを融合させた図 2.19 のような表現も、Metal を用いることで実現できます。

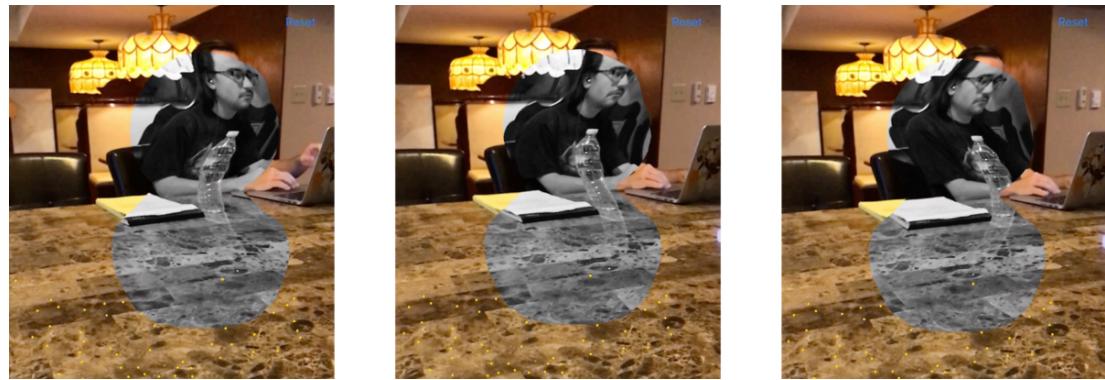


図 2.19: 仮想オブジェクトの描画に現実空間を反映

これらの実装方法は Metal の知識を必要とするため、詳細は「Metal」の章にて解説します。

第3章

Core ML

3.1 はじめに

Core ML は、機械学習における推論（予測）を SIMD と GPU で高速化するフレームワークです。開発者は、Keras、scikit-learn、libsvmなどの有名な機械学習のツールでシリアル化されたモデルとパラメータを Core ML で直接利用することができます。

Apple は、今日の AI ブームの火つけ役となった研究や最先端の研究成果のモデル、パラメータを Core ML で利用できる形にして自社サイトで公開しています。ゆえに、Apple が強く想定する Core ML のターゲットは、アプリケーションの開発者が学術的な成果やオープンソースとして公開されているモデルとパラメータを使って、アプリケーションを開発することなのかも知れません。しかし、すでに公開されている研究成果だけでは、開発者が解決したい問題をすべて解決できるわけではありません。私は、開発者自身で設計・学習したモデル・パラメータで問題を解き、それをアプリケーションに組み込むことこそが Core ML を使う価値であると考えます。Core ML は、そのときのアプリケーションへの機械学習の組み込みを容易にするフレームワークです。そのためには、そのフレームワークの単純な使い方だけではなく、機械学習の基礎と、解きたい問題に関する深い知識や洞察などが開発者に求められます。

3.1.1 ユースケース

まず、最初に Core ML のユースケースについて考えてみます。現在、使われている主な機械学習を応用したサービスのほとんどは、ネットワークを通じて提供されています。たとえば、Google Drive の写真へのタグ付けサービスや自動翻訳サービスなどがあります。リモートのサーバ上で機械学習を運用した方が、管理・運用コストが低いと考えられます。機械学習によるサービスと、学習に必要なデータの収集が同時に実行できるからです。そのような観点から、Apple は、ローカルで機械学習を利用する利点の一つにプライバシー保護をあげています。しかし、リモートで機械学習を運用することが、常にプライバシーを侵害を意味するわけではありません。これらの議論を踏まえ、ローカルのアプリケーションに Core ML を使って機械学習を使った機能を組み込む必要性や利点は、どういったときに生じるのでしょうか。

既存の研究成果やオープンなモデルを手身近に使いたいケース

機械学習で解決したい課題が、写真中の文字認識、自動タグ付けなど、一般的な課題であり、学会、オープンソースのコミュニティでその最新の成果が公開されているケースです。これは、Appleが想定する使い方にもっとも近いケースだと私は考えます。物体認識、文字認識等は、多くの研究などの成果物があり、オープンソースで公開されているものも少なくありません。リモートでこれを実装する場合は、開発者がサーバで動作する機械学習のアプリケーションを実装し、多くのトランザクションを処理する必要があります。それと比較し、公開されている手法をそのまま利用し、ローカルで運用するCore MLの使い方は、導入のコストパフォーマンスが高いと言えます。ただし、研究成果等は、商用利用を禁じているものもありますので、ライセンス等に注意して利用する必要があります。

瑣末な機能を機械学習で実現したいケース

これは、ユーザに提示するメニューの順番を並び替えるといった、ちょっとした機能を機械学習で実現したいようなケースです。数年前までは、自前でサーバを持たないデベロッパは、ユーザの挙動のログを収集できないため、こういった機能を実現することはシステム的に不可能でした。しかし、近年では、Google firebaseといったユーザの挙動を収集してくれるサービスがあり、自前のサーバをまったく持たずに、開発者は、ユーザの挙動を収集できます。こうして収集したログに基づいて、各ユーザがよく選ぶメニューを予測するモデルとパラメータをオフラインで学習することができます。Core MLを使えば、そのモデルとパラメータを使って、ローカルのデバイス上でユーザがよく選ぶメニューを予測し、メニューの順番を自動的に並び替える機能を実現できます。こういったサードパーティのサービスで集めたユーザログから、ちょっとした利便性をユーザに機械学習で提供したい場合などもCore MLのユースケースになると考えられます。

即時性が求められるケース

AR アプリケーションにおける物体認識、カメラアプリの顔検出、指で書いた文字でテキストを入力する機能をiOSで実装したいというケースです。認識等に高速なレスポンスが求められる場合、サーバで機械学習を実行するのは現実的な実装とは言えません。また、オフラインでも実行できる必要があるものは、サーバ上で機械学習を実行するような構成にはできません。

こういった場合に、Core MLで認識のためのモデルとパラメータをアプリケーションに内蔵しておけば、高速に画像を処理できます。

3.2 本章の構成

本章は、開発者が解きたい課題のために、機械学習のモデルを設計し、そのパラメータを計算し、それらをCore ML用にシリアル化し、アプリケーションに組み込むまでの流れを説明します。本章の第二節では、Core MLを使うにあたって必要な機械学習の大まかな知識を解説し、第3節では、Core MLの概要と、重要なピースであるCore ML toolsについて説明します。第4節では、上記の3つのユースケースのうち「既存の研究成果やオープンなモデルを手身近に使いたいケース」と「即時性が求められるケース」を取り上げ、Core MLを使って、機械学習を利用したアプ

リケーションを実装する具体的な手順を解説します。最後に、Core ML を注意深く考察すると見つかる短所について議論し、本章をまとめたいと思います。

3.3 Core ML のために学ぶ機械学習

Core ML を利用するにあたって、最低限必要と考えられる機械学習の概念について解説します。機械学習を支える領域は、線形代数、解析、数理統計など多くの領域に渡ります [1][2][3]。本節では、Core ML を利用するにあたって、読者が、機械学習のツールやの Core ML の内部で行われていること（隠蔽されていること）のイメージを大まかにつかめるようになることを目的にそれらのダイジェストを解説します。また、回帰、モデル、パラメータ、最適化、推論、学習誤差（誤差）、汎化誤差（性能）など、これらの用語を聞いて、何を説明するか、それらが何を意味するかをイメージできる読者諸氏は、本節を読む必要はありません。以降の数式の表記を以下にまとめます。

$$\begin{cases} v, \theta \dots \text{スカラー} \\ \mathbf{v}, \Theta \dots \text{ベクトル, 行列} \\ v_i \dots \text{ベクトル } \mathbf{v} \text{ の第 } i \text{ 成分} \\ W_{i,j} \dots \text{行列 } \mathbf{W} \text{ の } i \text{ 行 } j \text{ 列成分} \end{cases}$$

本節のサンプルプロジェクトは samplecode/内にある python/scikit-learn/scikit-learn.ipynb です。

3.4 機械学習の分類

機械学習は、識別モデルと生成モデルの二つに大別されます。識別モデルには、入力から何らかの出力を推定する回帰、入力を分類するクラス識別などが含まれます。識別モデルは、端的に言うと、ある入力に対して、一つの出力値（多次元のベクトルも含む）を返すモデルです。生成モデルは、モデルから任意の新しいデータを生成できるモデルです。例えば、識別モデルの場合、機械学習によって得られるものは、ある成人男性の身長を入力するとその人の体重を予測する機能です。一方で生成モデルの場合は、何度も出力を繰り返すと、その出力の履歴が学習に使ったデータと似た分布になるような、ある成人男性の身長と体重のペアを出力する機能が得られます。

さらに、機械学習の中には、教師あり学習と教師なし学習があります。教師あり学習では、何らかの入力に対する正しい出力のペアが正解データとして、人によって与えられます。そのデータを使って、任意の入力に対する出力を推定する機能を学習することが教師あり学習です。一般的な機械学習のほとんどはこれです。

一方で、何が正解かは与えられていないのですが、手元にあるデータの傾向から、正しそうな出力を推定する何かを学習することを教師なし学習と言います。まず、具体的な教師なし学習の一例として、図 3.1 の点をクラスタリングする（グループにわける）ことを考えてください。図 3.1 には、各データがどのクラスタに属しているかという正解ラベルがまったく与えられていません。しかし、人が見ると、図 3.1 の点の集まりは、なんとなく、ふたつのクラスタを形成しているように見えます。これは、教師なし学習の典型的な例で、実際にこのデータは、k-means と呼ばれる手法でクラスタリングすることができます。図 3.2 に k-means によるクラスタリングの結果を示します。k-means は、あらかじめデータが幾つのクラスタに別れるかを事前に情報として与える必要があり

ますが、クラスタ数自体もパラメータと考え、データから学習する手法もあります [4][5]。

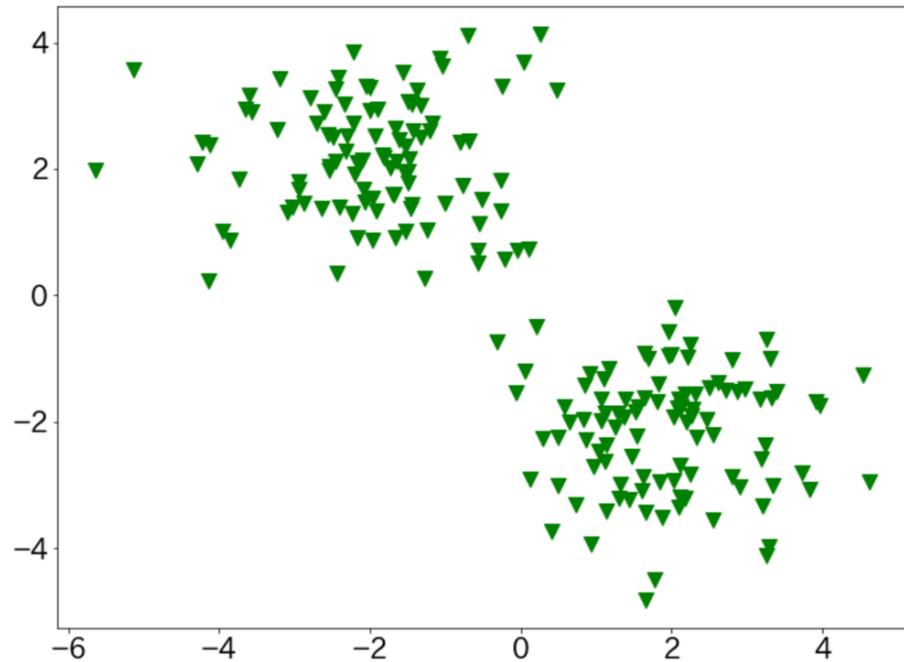


図 3.1: 教師なし学習の例（データ）

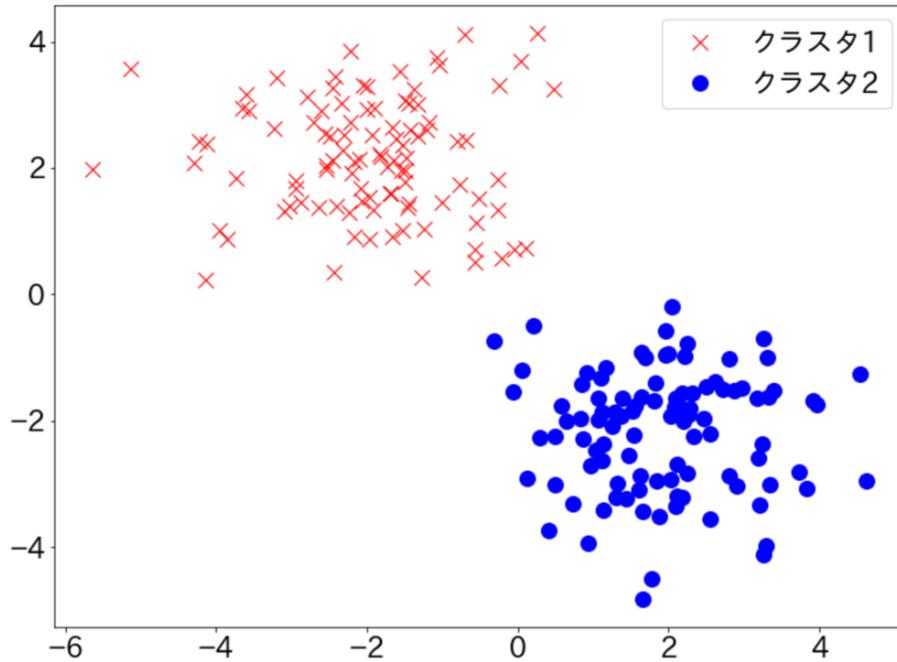


図 3.2: 教師なしデータを、教師なし学習の手法によって、クラスタリングした結果の例（データ）

機械学習の分野の中には、広告収益を最適化する目的で一時注目されたバンディット問題や、囲碁や自律制御ロボットなどで使われる強化学習などいろいろなものがあり [6][7]、ここですべてを解説することはできません。一般的に AI と呼ばれている、機械学習を使った技術やアプリケーションのほとんどは、教師あり学習かつ識別モデルです。本節は、教師あり学習かつ識別モデルの中でも、回帰モデルを例として、機械学習の概要をさらに詳しく説明していきます。

■コラム: 独立同分布 (independent identically distributed, i.i.d.)

本章でのデータは、すべて独立同分布 (independent identically distributed, 以降 i.i.d.) を仮定しています。i.i.d. は、データが同じ分布から独立に生成されるということを意味します。具体的な i.i.d. の例として、サイコロを何回か振るとき、サイコロの出目は、過去のサイコロの出目に影響されないといったケースが挙げられます。つまり、サイコロは、過去の出目の履歴に関わらず、常に $1/6$ の確率でそれぞれの目が出るということです。チンチロチンでそろそろ ○○な目が出るはずだといった浅はかなギャンブラーの考えは根本的に間違っているということです。一方で、i.i.d. でない例として、英文をキーボードで打つときに、次に打たれるキーは、直前に打たれたキーに依存するというケースが挙げられます。これは、英単語の綴りには明らかに統計的な偏りがあるためです。例えば、"q" のあとに "z" が打たれる可能性はほぼありません。一方で、"a" や "e" のキーは、常に高い確率で打たれる可能性を持つと予想できます。

つまり、キーボードで次に打たれるキーを予測する時は、明らかに過去のキー入力を考慮して予測する必要があるということです。こういったケースは、ユーザが次に購入する商品の予想など様々なものに当てはまります。むしろ、現実問題には、i.i.d. を仮定できるデータの方が少ないとも考えられます。もし、機械学習に使おうと思っているデータが明らかに i.i.d. でない場合は、時系列モデル、状態遷移モデルの利用を考える必要があります。

3.4.1 データとモデルとパラメータ

機械学習のシンプルな例として、ある1次元の入力 x に対し、ある1次元の出力 y を出力する課題を考えます。ここでは例として、「ユーザーがアプリを起動する時刻」を入力とし、「アプリ起動直後にユーザーが購入することの多い商品の価格」を出力として推定するケースを考えてみましょう。ありえない例ですが、多くのユーザーが起動したタイミングで買う可能性が高い製品をアプリのトップ画面に大きく表示できれば、売上の向上を見込むことができると仮説を立てます。

この課題は、前述の識別モデルの典型的な回帰問題に帰着できます。回帰とは、ある入力に対してある出力を返すことです。機械学習の文脈では、手元にあるデータの入力と出力の関係を学習し、入力に対して正しい出力を返す関数を作ることを意味します。

この回帰関数を一般化すると、

$$y = f(x, \Theta) \cdots (3.1)$$

と書くことができます。この(3.1)の意味するところは、入力 x に対して出力 y を返す関数 f があり、その関数 f は、パラメータ Θ を持つということです。この例で言えば、 x がユーザの起動時刻であり、 y がユーザがそのタイミングで購入することが多い商品の価格となります。 $\Theta = \{a, b\}$ とし、関数 f の形状を1次の線形モデルとすれば、

$$y = a \cdot x + b \cdots (3.2)$$

として中学校で習った1次関数を使って回帰関数を表現することができます。細部を省いて議論すると、 f は式の形を定義し、 Θ はその式の中の入力変数以外の定数と考えることができます。機械学習において、 f のような式の定義をモデルと呼び、 f の中有る Θ をパラメータと呼びます。任意の入力に対して適切な出力を出せるように、この f と Θ を適切に選ぶことが回帰問題の要です。

3.4.2 学習と推論

今ここに、実際のユーザのログから得た n 個の入出力データ $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ があるとします。これは、実際に収集したユーザがアプリケーションを起動した時刻 x_i と、そのときに購入したアイテムの価格 y_i のペアのリストです。開発者がこのデータを正しく再現できるモデルとパラメータを用意することができれば、この課題を解決できると考えられます。それでは、どのようなモデルを用意し、どう学習すればよいのでしょうか。

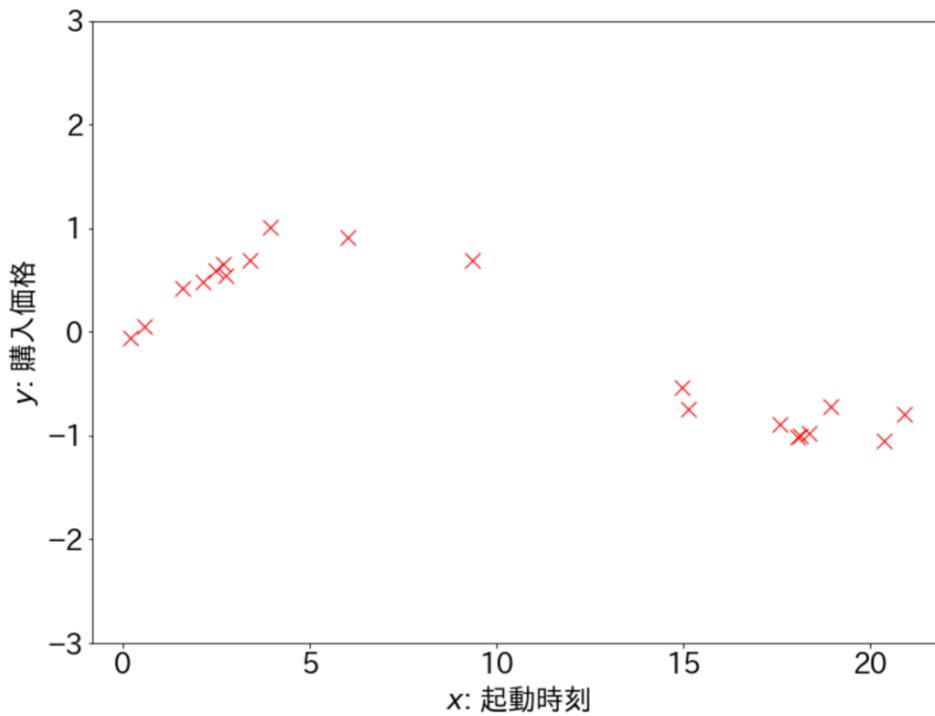


図 3.3: 学習に使うデータ

図 3.3 に学習に使うデータを示します。図 3.3 の横軸 x にユーザがアプリケーションを起動した時刻をとり、縦軸 y にユーザがそのときに購入したアイテムの価格をとり、データをプロットしたと考えてください。このデータを、1 次の回帰モデル（直線当てはめ）で表現することを考えます。

1 次の回帰モデルは、

$$y = a_1 \cdot x + a_0 \cdots (3.3)$$

で表されます。後述する学習によって得たパラメータをモデルに代入し、任意の入力に対する出力を計算することを、予測 あるいは 推論 と言います。この例では、推論は、学習によって得た a_0 と a_1 を代入した (3.3) に予測したい x の値を代入し、予測値 y の値を出力することを意味します。

機械学習でパラメータを計算するということは、収集したデータにもっとも適当なパラメータ a_i を求めるということです。機械学習では、このデータに対して適当なパラメータを求めるこれを「学習する」と言います。この「適当な」パラメータとはどういった特徴を持つものでしょうか。「適当な」パラメータの定義は複数あります。今回の例では、もっとも単純な推論誤差の絶対値を最小にするパラメータを「適当な」パラメータと考えることにします。

推論誤差の絶対値とは、推論結果 ($a_1 \cdot x + a_0$) とデータ y の差分の絶対値です。つまり、 i 番目のデータに対する誤差は、

$$e_i = |y_i - (a_1 \cdot x_i + a_0)| \cdots (3.4)$$

と表現できます。そして、学習に使うデータセットが、 n 個あるとすると、データセット全体に対する誤差は、

$$e = \frac{1}{2} \cdot \sum_{i=1}^n |y_i - (a_1 \cdot x_i + a_0)|^2 \cdots (3.5)$$

と表現できます。^{*1}

この(3.5)のような予測値と正解データとの差を定義する関数を誤差関数(**cost function**)と呼びます。また、この誤差 e を最小にする a_0 と a_1 を求めることができがパラメータを求めるこことなり、これを最適化と呼びます^{*2}。この最適化とは、 e のような関数を最小化あるいは最大化するようなパラメータを求ることです。scikit-learnなどのCore MLがサポートする機械学習のツールは、最適化がすでに実装されています。このため、それらを使う場合は、最適化の詳細について完全に理解する必要はありません。最適化の速度など、それ自体を改善したい場合は、ツールを使うのではなく、自分でそのアルゴリズムを実装する必要があります。最適化に興味のある方は、[8][9]などの書籍を参考してください。

データから学習したパラメータを式に代入し、 x に 0~5 の値を代入し、計算結果を描画すると（実際に直線なので 2 点のみで描画できます）、図 3.4 のような結果が得られます。

^{*1} 一般に推論誤差の絶対値は 2 乗し、 $\frac{1}{2}$ を係数にして定義することが多いです。これは最適化の計算において x について微分すると 2 次の係数が 1 になり計算しやすいためです。また $\frac{1}{2}$ を係数にしても最小あるいは最大になるときの x の値は変わりません。

^{*2} プログラミングの分野では、実行環境に応じたパフォーマンスを引き出すようなコーディングをすること最適化と呼びますが、それと混同しないように注意してください。

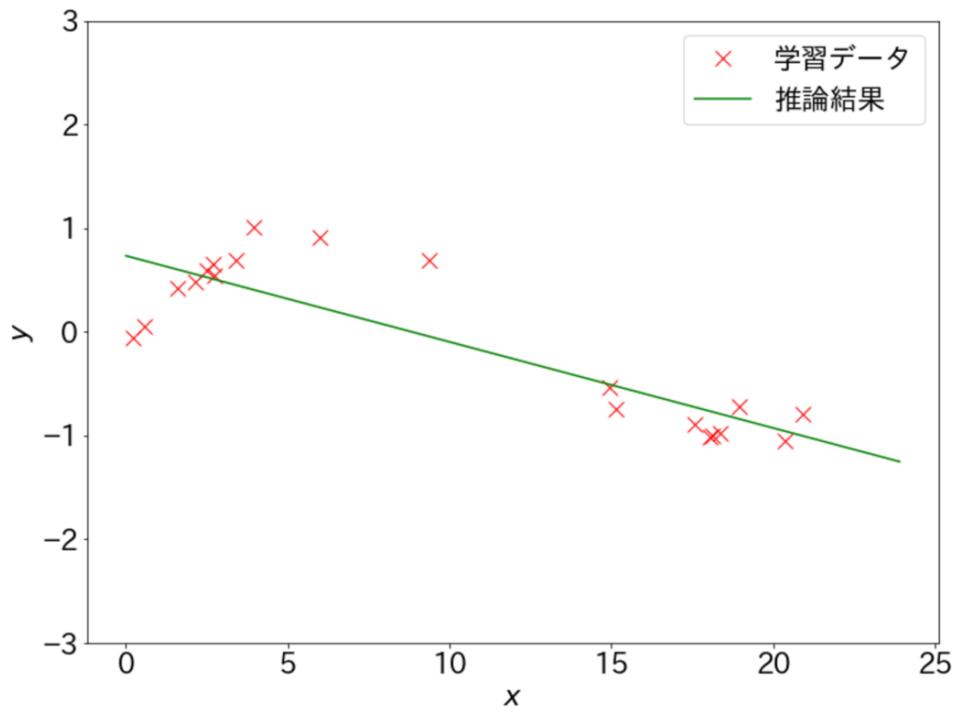


図 3.4: 直線で回帰した例

図 3.4 を見ると、データをうまく表現できているように見えません。それでは、2次の回帰モデルではどうでしょうか。2次の回帰モデルは、

$$y = a_2 \cdot x^2 + a_1 \cdot x + a_0 \cdots (3.6)$$

と表現されます。この a_0 、 a_1 、 a_2 を1次の回帰モデルと同様に計算し、その曲線を描画すると、図 3.5 の結果が得られます。

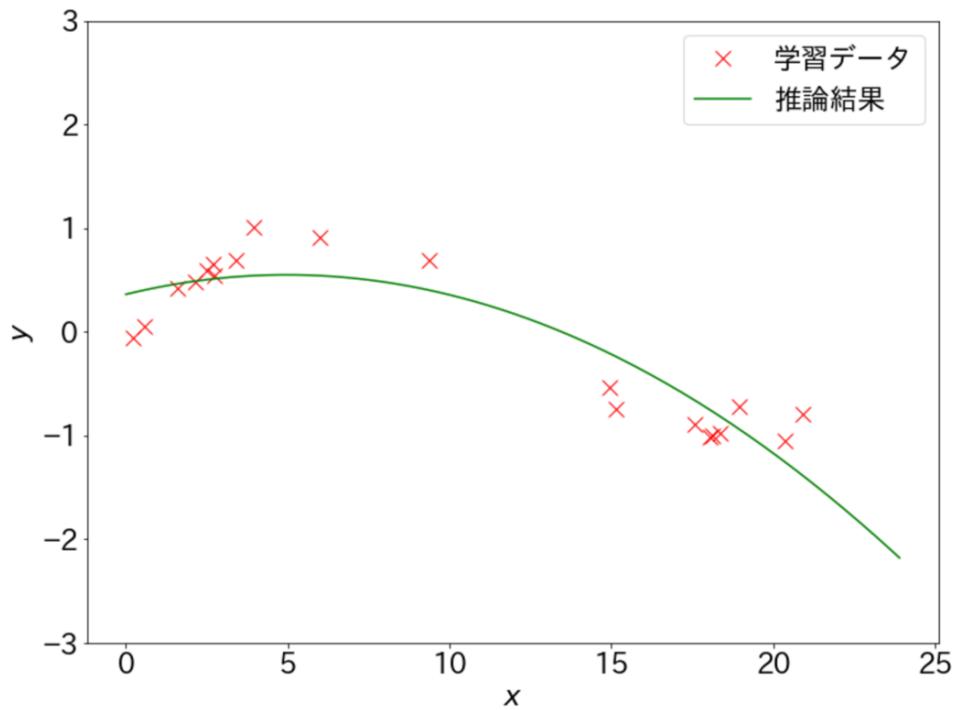


図 3.5: 2 次の曲線で回帰した例

図 3.5 を見ると、2 次のモデルは、1 次のモデルよりデータをうまく表現できているように見えます。単純に予想すると、モデルの次数をあげていくと、表現力が上がり、性能が向上していくように考えられます。そこで、実際に、このデータに 9 次の線形回帰モデルを適応してみると、図 3.6 に示すような推論結果が得られます。図 3.6 から、9 次のモデルは、他の次数の低いモデルよりも高い精度でデータを表現できているように見えますが、何か正しくないようにも見えます。

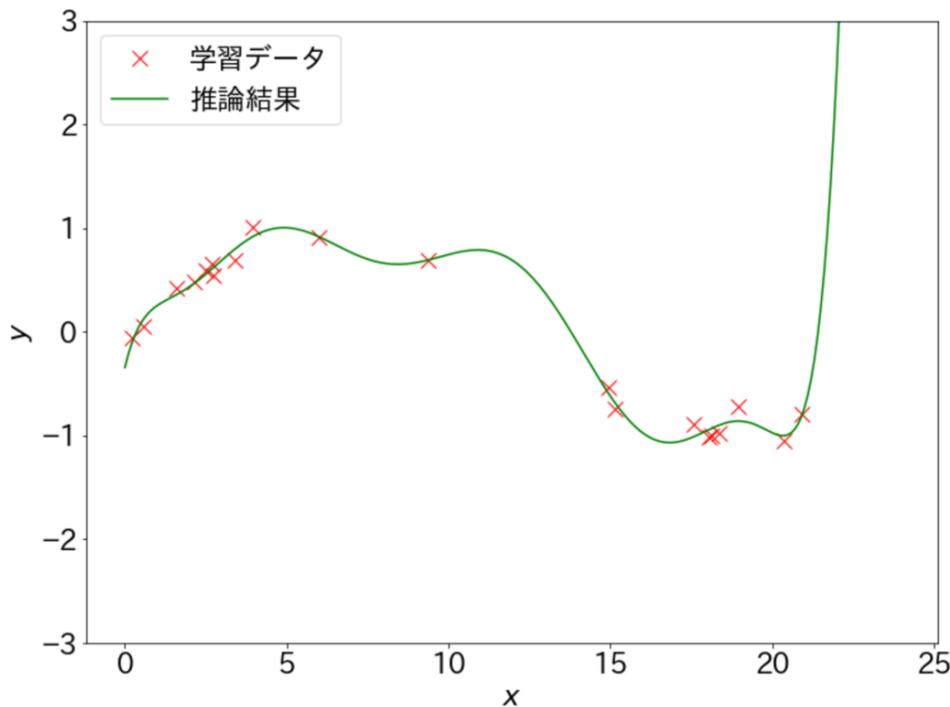


図 3.6: 図 4.9 次の多項式で回帰した例

3.4.3 学習性能と汎化性能

機械学習において、もっとも大切なことのひとつが性能評価の考え方です。先ほどのデータに対する誤差 e を最小化できるモデルやパラメータがもっとも高い性能をもつと考えていいのでしょうか。残念ながら、そうではありません。なぜならば、機械学習で本質的に求められる性能は、パラメータを計算するときに手元にないデータに対する、汎化性能と呼ばれる性能であって、手元にあるデータに対する性能ではないからです。現実的には、この収集不可能なデータを必要とする汎化性能は、評価できません。このため、一般的に擬似的に手元にないデータがあるように見せかけて性能評価を行います。具体的に、今手元にあるデータをモデルやパラメータを決めるための 学習データ と、性能を検証するための テストデータ の二つに分け、学習に使っていないデータで性能を評価します。このとき、学習データを使って評価した性能を 学習誤差 と呼び、テストデータを使って評価した性能を テスト誤差 と呼びます^{*3}。

この例では、パラメータを学習するときに最小化した誤差、つまり (3.5) に学習データを代入し、

^{*3} 厳密に議論すると、この汎化誤差は"真の汎化誤差"ではありません。なぜならば、"真の汎化誤差"は、想定されるすべてのデータが無限になければ、評価できないからです。しかし、"想定されるすべてのデータが無限に"あれば、学習誤差を最小化することと汎化誤差を最小化することが同値になるので、もはや汎化誤差を議論する必要がなくなってしまいます。このことを忘れずに機械学習の性能について考えてください。テスト誤差(性能)を汎化誤差(性能)と呼ぶこともあるようです。

計算した結果が学習誤差に相当します。また、(3.5) にテストデータを代入し、計算したものがテスト誤差に相当します。学習誤差が小さくても、学習時に利用していないデータに対して、小さい誤差で回帰できなければ（つまりテスト誤差が大きい）、そのパラメータとモデルは「適当ではない」と評価します。それでは、実際に学習誤差とテスト誤差が乖離するようなケースは存在するのでしょうか。この議論は、機械学習において、極めて重要な問題を提起するため、具体的に説明していきます。

3.4.4 過学習

さきほどの例と同じデータがあるとします。そして、そのデータを学習データとテストデータに二つに分けます。この学習データを使って、1次、2次、4次、9次、16次の多項式を使って回帰し、得られたパラメータをつかって、推論した結果を曲線で描画し、学習データを X でプロットし、テストデータを丸でプロットした結果を図 3.7、図 3.8、図 3.9、図 3.10、図 3.11 に示します。

特に、図 3.7 には、学習データと推論結果との差を破線の垂線で、テストデータと推論結果との差を実線の垂線で描画しました。この破線の垂線の長さの総和が学習誤差を示し、実線の垂線の長さの総和がテスト誤差を示します。さらに、(3.5) に学習したパラメータと学習データおよびテストデータを代入し計算した、これらの 5 つのケースの学習誤差とテスト誤差を表 3.1 にまとめます。

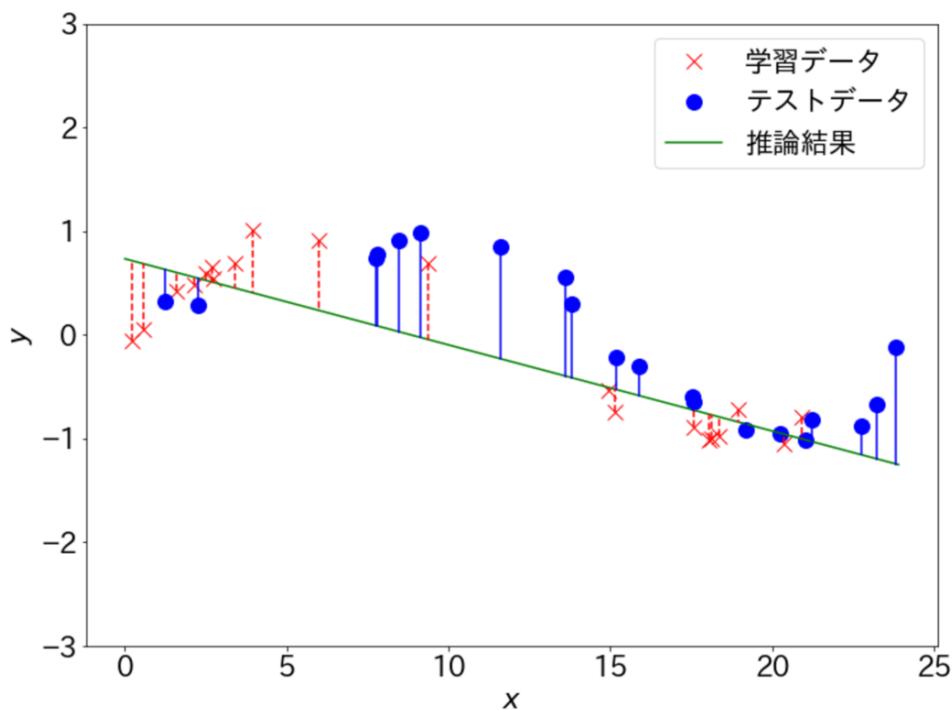


図 3.7: 1次の多項式で回帰し、推論とテストデータと比較した図

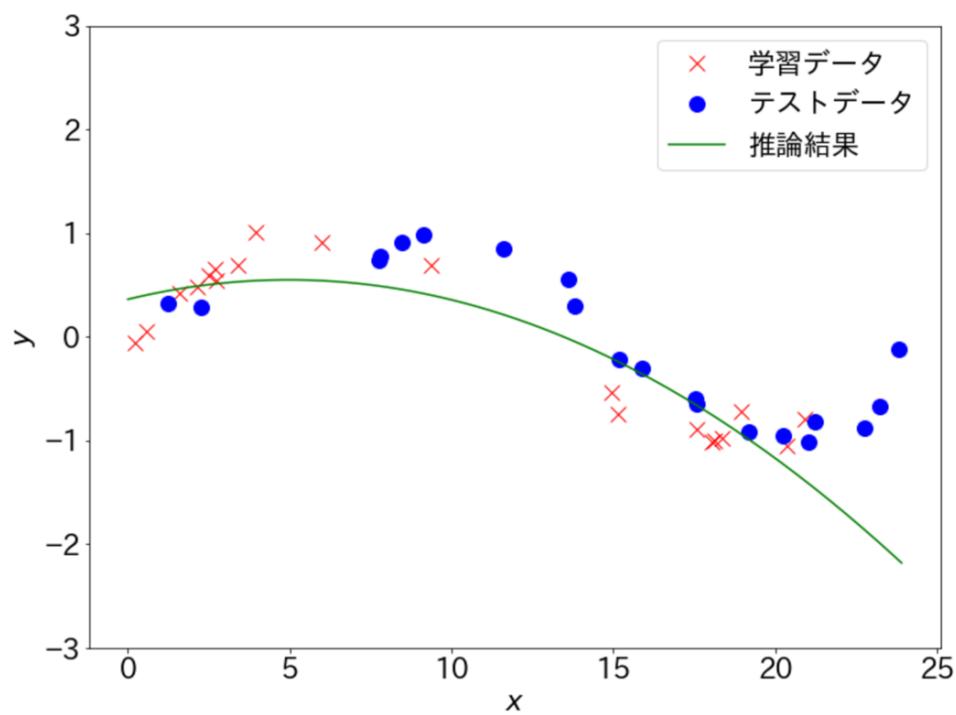


図 3.8: 2次の多項式で回帰し、推論とテストデータと比較した図

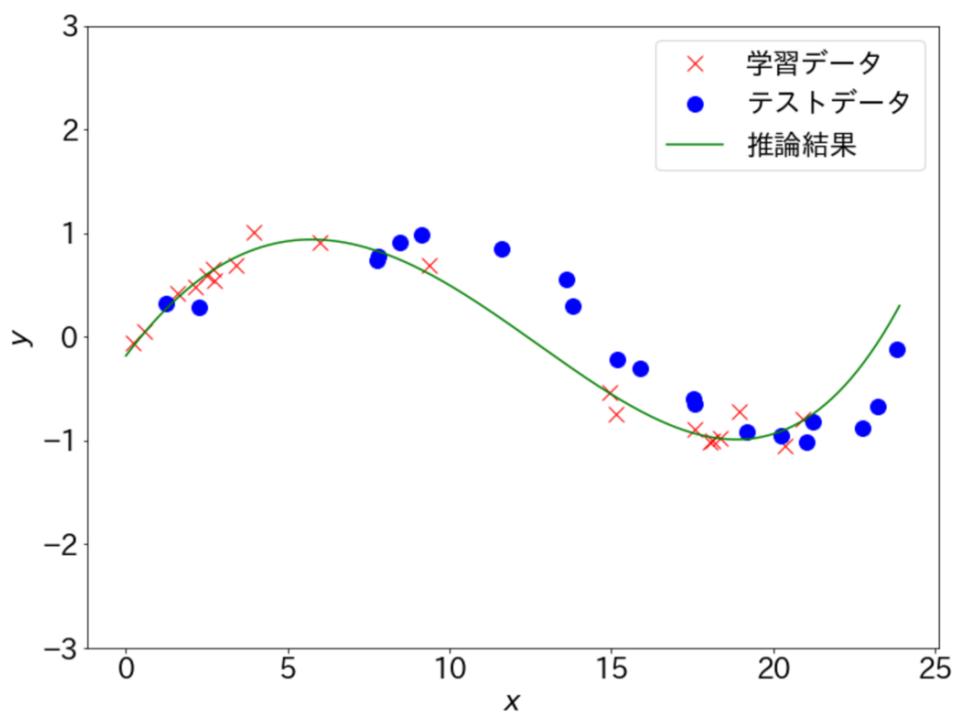


図 3.9: 4次の多項式で回帰し、推論とテストデータと比較した図

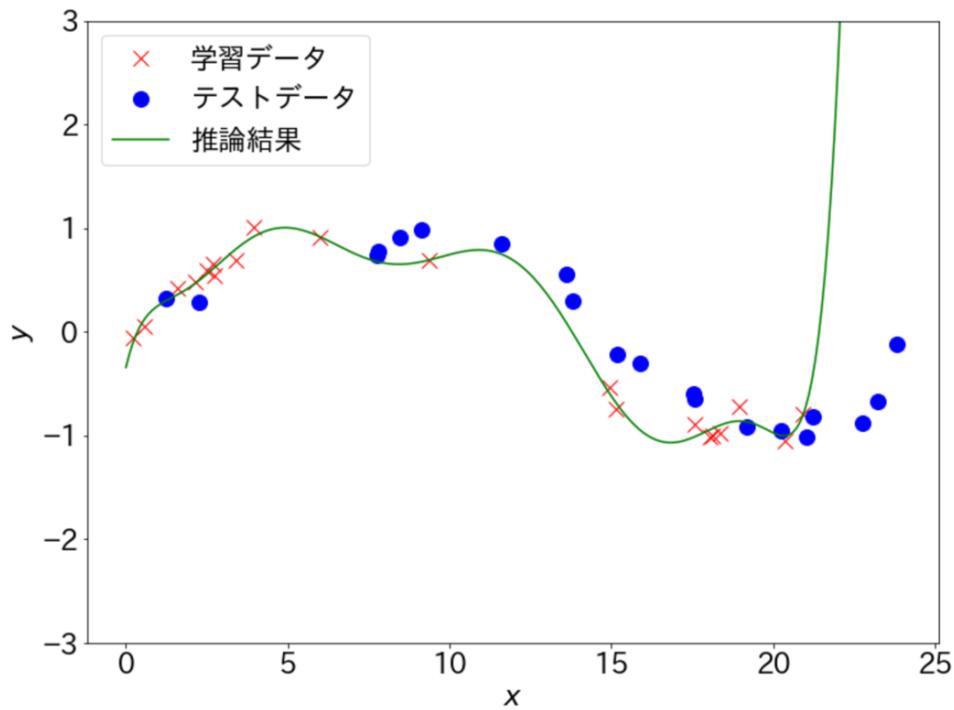


図 3.10: 9 次の多項式で回帰し、推論とテストデータと比較した図

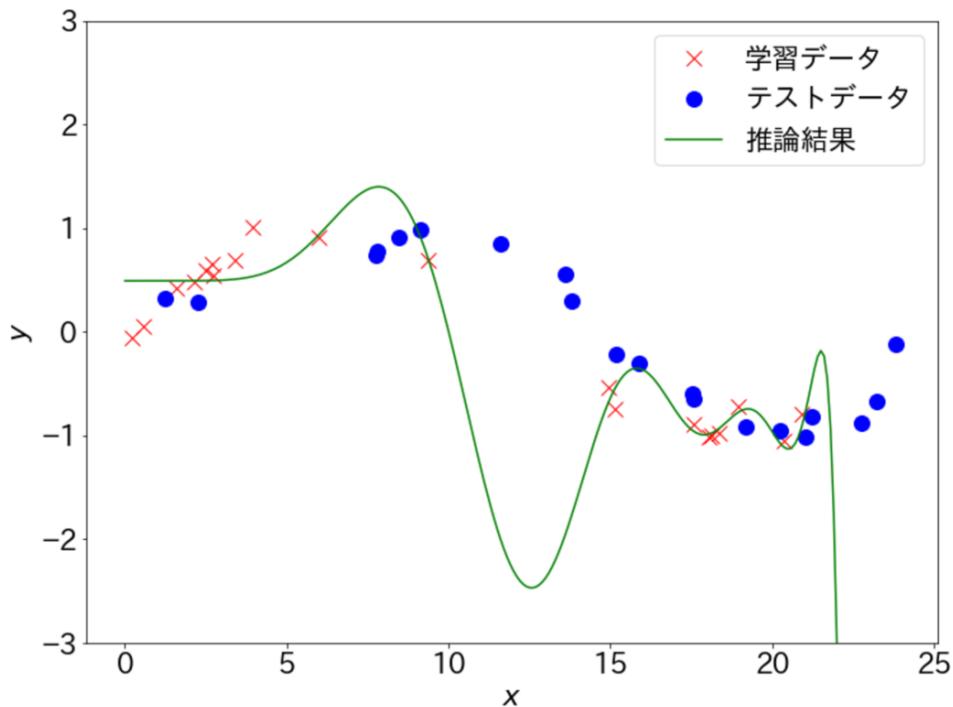


図 3.11: 16次の多項式で回帰し、推論とテストデータと比較した図

表 3.1: それぞれの性能の値は (3.5) で求めた誤差の二乗の総和

多項式の次数	1次の多項式	2次の多項式	4次の多項式	9次の多項式	16次の多項式
学習誤差	1.66	1.35	0.43	0.31	0.93
テスト誤差	1.94	2.53	0.99	34.65	382.17

これらの図 3.9, 図 3.9, 図 3.9 を見ると、1次から4次までモデルが複雑になっていくに連れて、曲線は X と丸の両方の近辺を辿るようになります。一方、図 3.11, 図 3.11 のより複雑なモデルでは、推論結果の曲線は、学習データを示す X との距離が小さくなっていますが、テストデータである丸の周辺をまったく通らず、テストデータを無視したような推論結果を示していることもわかります。

表 3.1 を見ると、モデルを複雑にしていくと学習誤差は小さくなるが、テスト誤差が大きくなることが数値的にわかります。これは、データの本質的な複雑さよりも、モデルの方が複雑になってしまい、データに含まれるノイズまで、モデルが説明してしまっていることに起因しています。機械学習では、こういった現象を過学習と呼びます。モデルを複雑にすれば、学習誤差を改善できますが、テスト誤差は悪化します。また、この過学習が起こる理由を、データ数に対してモデルが複

雑すぎると解釈することもできます。

表3.1から、この実験結果の中では、学習誤差が小さく、またテスト誤差が最も小さい4次のモデルがもっとも適当なモデルであると言えます。機械学習において、性能が高いモデルとパラメータを得る王道はありません。データが意味する本質を理解し、適切な複雑性を持ったモデルを探し、大量のデータを使ってパラメータを学習することが肝要です。

3.4.5 データとモデル

一方、学習に必要なデータ量の観点からもモデルをただ複雑にすればいいわけではないことがわかります。グラフで任意の直線を引くためには、少なくとも2点必要であるように（傾きと切片、二つの定数を計算する必要があります）、2次のモデルのパラメータを求めるには、少なくとも3つ以上のデータが必要です（2次関数の定数は3つあります）。このため、モデルが複雑になると学習に必要なデータは増加します。もし、パラメータが600,000個ほどあるモデルがあった場合、莫大な学習データがないとパラメータをうまく学習できません^{*4}。

3.5 まとめ

議論や説明が大変ではありましたが、多項式による回帰を例に機械学習の概要を解説しました。ここで解説した、特に評価についての概念や考え方は、単純な線形回帰、ベイズ推定、深層学習など、どんな機械学習の手法についても共通するものです。機械学習を応用したアプリケーションを作るには、そもそも課題やデータの性質を理解し、それをうまく説明できそうなモデルを選ぶ必要があります。そのモデルはパラメータを持ち、そのパラメータはデータから学習する必要があります。開発者は、集めたデータを、パラメータを学習する学習データと、評価を行うテストデータのふたつのグループに分け、学習データでパラメータを学習し、テストデータで学習したパラメータとモデルの性能を評価します。モデルを複雑にすると、より難しいデータを表現できるようになりますが、過学習しやすく、またパラメータの学習に必要なデータ量も増加するという性質があります^{*5}。開発者は、自分が用意できるデータの数と問題に対する知識や考察に基づき、モデルとパラメータを設計・学習し、慎重に開発する必要があります。

3.6 Core ML

3.6.1 概要

ここまで機械学習の概要について説明してきました。本節では、Core MLについて説明していきます。

単純に機械学習のタスクを分類すると、次の3つに分けられます。

1. 設計・データ収集 データを集め、モデルを考える
2. 学習・評価 パラメータを学習し、その性能を評価する

^{*4} 実際には正則化などの手法でうまく学習する方法があります。

^{*5} 過学習と性能の評価は重要かつ難しいものです。正しく理解するには、[1][2][3]の文献や機械学習、学習理論の書籍を参考にしてください。

3. 推論

モデルとパラメータで与えられた入力に対する出力を回帰する

この中で、1と2は、オフラインで開発時に実行するものであり、多少時間がかかってもユーザエクスペリエンスに直接影響を与えません。一方で、3はユーザのデバイス上あるいはネットワークサービスの場合、サーバ上で実行するものであり、即時性が求められます。Core MLは、この3つの中の推論の実装を肩代わりしてくれるフレームワークです。つまり、Core MLには、よく使われる機械学習のモデルが（プログラミングの意味で）最適化されて実装されており、開発者は、Core MLに実装されたモデルをランタイム時にロードし、そのモデルにパラメータをセットし、推論を実行します。Core MLは、どんな機械学習のモデルもサポートするわけではありません。これは、機械学習のモデルの自由度の高さ（つまりはどんな関数でもいい）を考えると当然です。

3.6.2 サポートされるモデル

Core MLがサポートする機械学習のモデルとツールの一覧を表に示します。

表 3.2: Core ML がサポートするツールとモデル [10]

モデルの種類	詳細	ツール
ニューラルネットワーク	一般的なニューラルネットワーク、畳み込み層、リカレント構造	Caffe, Keras 1.2.2 later
木構造	決定木、ランダムフォレスト、ブースティング木	scikit-learn 0.18, XGBoost 0.6
サポートベクターマシン	単純回帰、多クラス分類	scikit-learn 0.18, LIBSVM 3.22
線形回帰	線形回帰、ロジスティック回帰	scikit-learn 0.18
特徴量処理	疎ベクトル、密ベクトル、カテゴリ分類特徴への変換	scikit-learn 0.18
パイプライン	モデルの組み合わせ	scikit-learn 0.18

表 3.2 にあるツールによって書き出されたモデルを利用する限り、Core MLに任せれば、GPUや SIMD を利用した実装で、推論を（プログラミングの意味で）最適化してくれます。ただし、繰り返しになりますが、Core ML がそれぞれのツールから生成したすべてのモデルをサポートするわけではないことにも注意してください。Core ML は、この表以外のツール等で書き出したモデルと、デベロッパが自由に設計した機械学習のモデルをサポートしません。そういう場合は、Core ML の枠組みから外れ、開発者自ら、推論部分を実装する必要があります。

3.6.3 Core ML のスタック

図 3.12 に Apple が公開しているドキュメントにある Core ML のスタックを引用します [11]。Core ML が、Accelerate.framework と BNNS、Metal Performance Shaders に支えられていることがわかります。Accelerate.framework は、SIMD を使った一般的な数値計算の高速化フレームワークや BLAS や LAPACK と呼ばれる線形代数演算のライブラリも含みます。BNNS は、SIMD で最適化されたニューラルネットワークのためのフレームワークで、Metal Performance Shaders は、GPU を使うためのフレームワークです。以上のことから、Core ML が、SIMD と GPU の恩恵を受けていることがわかります。

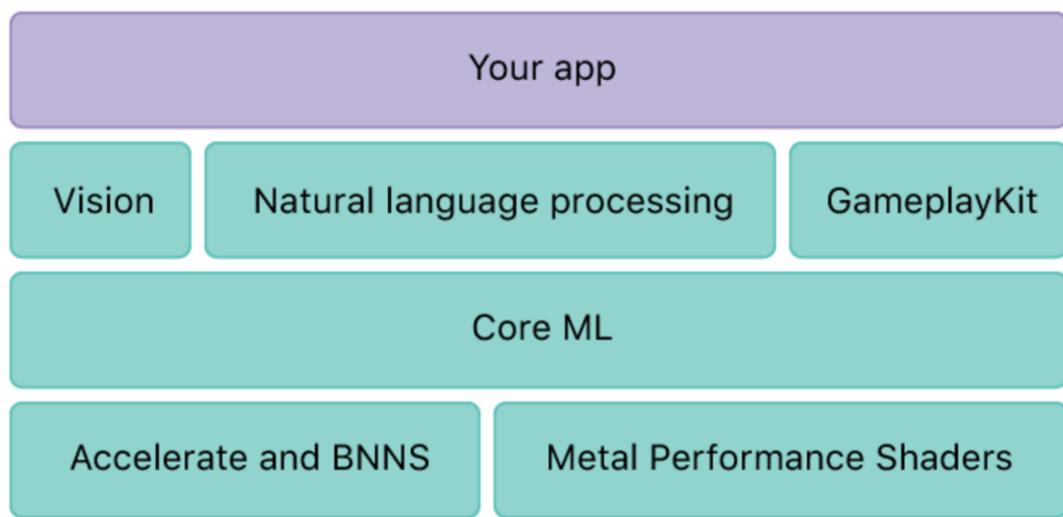


図 3.12: Apple が公開している Core ML のスタック

3.6.4 開発の流れ

Core ML を使った開発をもう少し具体的にしたアウトラインを図 3.13 に示します。まず、開発者は、機械学習ツールを使ってモデルの設計およびパラメータの学習を行い、そのモデルとパラメータを書き出します。その後、次に説明する Core ML Tools を使い、このモデルとパラメータのデータを Core ML のフレームワークで読み込めるように mlmodel ファイルにエクスポートします。最後に、Xcode に mlmodel ファイルをインポートし、アプリケーションに組み込みます。



図 3.13: アプリケーションのパッケージ内の Core ML のモデルデータ

3.6.5 開発環境

図 3.13 にある 3 つのステップの中で、Xcode を使わなければならない最後の実装ステップ以外は、Mac 以外の環境でも実行可能です。この特徴を生かし、機械学習の設計とパラメータの学習は、Linux で実行した方がコストが低いかもしれません。Linux の場合、Docker などのツールで環境を容易にセットアップしやすく、パッケージマネージャ等のメンテナンス性も高くなります。Mac と異なり、Linux はハードウェアの構成を自由に変えることができるため、ニューラルネットワークのパラメータの学習等に GPU を利用しやすい環境です（機械学習に GPU を使おうとすると、事実上 nVidia の GPU 以外は選択肢となりえません。Mac の場合、nVidia のビデオカードを内蔵していない機種がほとんどなので、外付けの PCIe 拡張ボックスなどを導入する必要があります）。また、Linux を使う場合、マシンとして、Amazon Web Service の EC2 のインスタンスを利用することもできます。

3.6.6 Core ML Tools

Core ML の重要なツール Core ML Tools について説明します。Core ML Tools の役割は、次の 3 つです。

1. モデルとパラメータを Xcode にインポートするために mlmodel ファイル (Core ML のモデルデータを保存するファイル) に変換する
2. 推論の入出力変数の名前を定義する
3. 作者名や著作権、ライセンスなどの情報を追加する

Core ML Tools の開発は銳意行われており、WWDC2017 で発表当時、version 0.30 でしたが、すでに、version 0.6.3^{*6} にアップデートされています。

Core ML Tools は Python で書かれており、オープンソースです。動作環境は macOS に限らず、Linux 環境でも動作します。Linux でパラメータを学習し、Linux で Core ML Tools を使って、モデルファイルを作り、macOS に送るといったスキームも実行可能です。ゆえに、Core ML では、機械学習のモデルやパラメータを計算する環境と、アプリケーションを開発する環境と、それをデプロイする環境が分離され、デベロッパは、それぞれの作業を分業することができるわけです。

現在、Core ML Tools は、Python version 2.7 で動作します。macOS のデフォルトの Python のバージョンが 2.7 なので、macOS の環境下で素直に Python を使っている人は、そのままインストールできます。しかし、pyenv や anaconda などで、それ以外のバージョンをインストールし、使っている場合は、バージョン 2.7 の環境を構築する必要があります。Core ML Tools のインストールは、python のパッケージマネージャ pip を使うのが便利です。

```
pip install -U coremltools
```

WWDC2017 の Core ML の紹介セッションで、Apple のエンジニアは、jupyter notebook を使い、Safari 上で学習したモデルデータの書き出しやコメント等の追加を Core ML Tools を使って実行するデモを披露しました [13]。機械学習の部分と mlmodel への変換のステップは、この jupyter notebook を使うことをオススメします [12]。jupyter notebook は、ノートのようにコードとその実行結果を保存しておける対話型の Python の実行環境です。jupyter notebook を使えば、scikit-learn, Keras などのツールによる機械学習のコーディングをブラウザで実行できます。コードと実行結果やグラフの内容をブラウザ上で編集、保存しておけるので、パラメータの学習などの作業するとして最適です。jupyter notebook の詳細については、jupyter.org のサイトを参照してください。

Linux マシンに jupyter notebook をインストールすれば、Mac からブラウザ経由でコードを書き、Linux マシン上で機械学習を実行することもできます。また、この構成の場合、jupyter notebook で書いたコードから Linux マシンの GPU を利用することもできます。jupyter notebook は、以下のように pip を使って、簡単にインストールできます。ただし、Mac に jupyter notebook をセットアップする場合、scipy が依存する gfortran、matplotlib が依存する freetype を、pip で jupyter notebook をインストールする前に、別途インストールする必要があります。gfortran は、GFortranBinaries からダウンロードしてセットアップできます^{*7}。freetype は、brew を使って、`brew install freetype` として、インストールできます。

^{*6} 2017年10月6日現在。

^{*7} <https://gcc.gnu.org/wiki/GFortranBinaries>

```
pip install --upgrade pip
pip install scikit-learn numpy scipy matplotlib cython jupyter
```

個別の実装例は、次節で解説しますが、Core ML Tools のコード例は以下のようになります。`scikitLearnModel` オブジェクトは、python で、scikit-learn を使って作ったモデルやパラメータを含んだオブジェクトです。変換自体は、`convert` メソッドを呼ぶだけなので、簡単です。`convert` メソッドの引数に入出力の型や名前を指定することができます。また、ライセンスや製作者名も簡単に指定できます。ここで指定した情報は、mlmodel ファイルを Xcode で選択して確認できます(図 3.14)。

```
### CoreML 用にモデルとパラメータを書き出す
from coremltools.converters import sklearn
coreml_model = sklearn.convert(scikitLearnModel, ["x1", "x2", "x3", "x4"])
coreml_model.author = 'Yuichi Yoshida'
coreml_model.license = 'BSD'
coreml_model.short_description = 'CoreML Test'
coreml_model.save('./output/linear_model.mlmodel')
```

以下のように文字列を `input_description` プロパティに設定し、それぞれの変数や出力値にコメントを追加できます。

```
coreml_model.input_description['x1'] = u'1 次の入力値。推定したい値を入力してください。',
coreml_model.input_description['x2'] = u'2 次の入力値。推定したい値の 2 乗を入力してください。',
coreml_model.input_description['x3'] = u'3 次の入力値。推定したい値を 3 乗を入力してください。',
coreml_model.input_description['x4'] = u'4 次の入力値。推定したい値を 4 乗を入力してください。',
coreml_model.output_description['prediction'] = u' 推定した結果'
```

▼ Machine Learning Model

Name linear_model
Type Pipeline Regressor
Size 660 bytes
Author Yuichi Yoshida
Description CoreML Test
License MIT

▼ Model Class

C linear_model ⓘ
Swift generated interface for model

▼ Model Evaluation Parameters

Name	Type	Description
▼ inputs		
x1	Double	1次の入力値、推定したい値を入力してください。
x2	Double	2次の入力値、推定したい値の2乗を入力してください。
x3	Double	3次の入力値、推定したい値を3乗を入力してください。
x4	Double	4次の入力値、推定したい値を4乗を入力してください。
▼ outputs		
prediction	Double	推定した結果

図 3.14: Xcode で mlmodel ファイルを開いたときのビュー

mlmodel ファイルは、Python の機械学習のモデルのインスタンスから、convert メソッドから書き出すことができます。あるいは、以下のように各ツールでシリアル化されたファイルから直接変換して mlmodel ファイルを作成することもできます。例えば、以下のように convert メソッドの引数に直接ファイルをパスを指定することで、直接ファイルから mlmodel ファイルを作成できます。

```
import coremltools.converters.keras as keras_converters
coreml_model = keras_converters.convert('./KerasMNIST.h5', input_names='image',
                                         output_names='digit')
```

3.7 実装

本節では、二つのケースを想定し、実際に機械学習のツールで設計・学習したモデルとパラメータを利用したアプリケーション開発の概要について説明します。例は、以下の二つです。それぞれ、異なる機械学習ツールを使って、モデルを作り、パラメータを学習し、まったく異なる目的で機械学習を応用します。

3.7.1 回帰～scikit-learn

まず、一つ目の実装例は、"自前でサーバを立てるまでもないような、頃末な機能を機械学習で実現したいケース"として、ユーザの起動時刻に応じて、3つのメニューの表示項目の順番を動的に変えるアプリケーションを取り上げます。まず、手元に表3.3のようなデータがあったとします。

本節のサンプルプロジェクトはsamplecode/内にあるiOS/RecChar/MenuPrediction.xcodeprojです。本節のサンプルプロジェクトはsamplecode/内にあるpython/scikit-learn/Softmax.ipynbです。

表3.3: 変換したデータ

時刻	ユーザ選んだ項目
2:03	メニュー3
2:14	メニュー2
2:28	メニュー2
2:32	メニュー3
2:34	メニュー2
2:50	メニュー3
4:04	メニュー3
4:23	メニュー3
4:26	メニュー3
4:50	メニュー3

このデータの場合、時刻や項目がとる値が直接的な数字ではありません。それぞれのデータを機械学習を行うのに適した形、以下の表3.4のように、時刻は時+分/60で変換し、メニュー項目は、メニュー1に0を、メニュー2に1を、メニュー3に2を割り当てて変換します。

表3.4: オリジナルのデータ

時刻	ユーザ選んだ項目
2.240572	1
2.473243	1
2.548051	2
2.571376	1
2.849114	2
4.070842	2
4.390852	2
4.436798	2
4.833840	2

変換の結果、このデータを2次元の数値として取り扱うことができます。横軸に時刻を取り、縦軸にメニューに選んだ項目を取り、例として使うデータをプロットしたものを図3.15に示します。図3.15をみると、出力も3パターンしかありませんし、データの分布を見ても、そこまで無秩序な

データには見えません。データを観察すると、どうやら、時刻に応じて、ユーザが選ぶメニューの項目に偏りがあるように見えます。時刻に応じて、よく選ばれるメニューが初めの方に表示されるように自動的に並び替えられそうです。

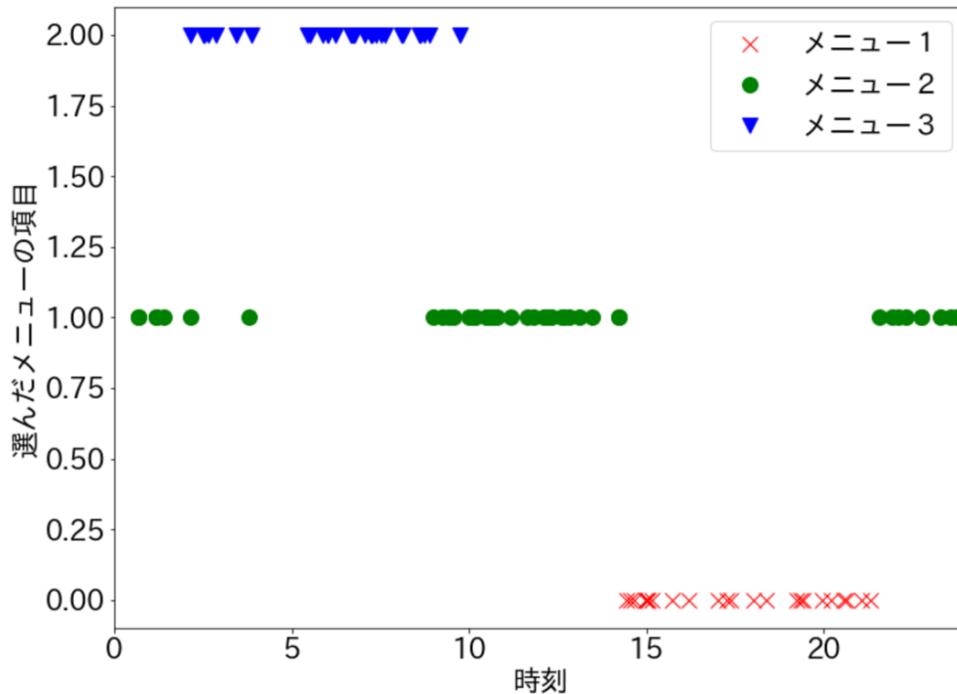


図 3.15: 時刻と選択したメニューのデータ

3.7.2 モデルとパラメータ

図 3.15 に示したデータから、各時刻において選択されやすいメニュー項目を推定するために、前節で紹介した線形回帰モデルをベースに、ロジスティック回帰と呼ばれるモデルを使うことにします。ロジスティック回帰自体の説明を始めると、それだけで大変な分量になってしまうので、シンプルに説明します。ロジスティック回帰は、ラベルなどの離散の選択肢を推定するときに使うモデルのひとつです。ロジスティック回帰では、ある入力に対して、選択肢と同じ個数の実数値を返す関数を考え、その関数をソフトマックス関数と呼ばれる関数に入力し、無理やり確率のような値にしてしまうことができます。ソフトマックス関数は、入力を $\mathbf{x} = \{x_1, x_2, x_3, \dots, x_n\}$ 、出力を $\mathbf{y} = \{y_1, y_2, y_3, \dots, y_n\}$ とすると、

$$y_k = \frac{e^{x_k}}{e^{x_1} + e^{x_2} + e^{x_3} + \dots + e^{x_n}} \cdots (3.7)$$

として定義されます。 (3.7) の入力 x_i に回帰関数の出力を入れると、ロジスティック回帰のモ

ルになります。この例では、回帰関数 $f_i(x)$ を3次の多項式として、それぞれの3つのメニューが選択される確率 p_i は、

$$p_i = \frac{e^{f_i(x)}}{\sum_{k=1}^3 e^{f_k(x)}} \cdots \quad (3.8)$$

と定義され、 $f_i(x)$ は、

$$\begin{cases} f_1(x) = a_{1,3} \cdot x^3 + a_{1,2} \cdot x^2 + a_{1,1} \cdot x + a_{1,0} \\ f_2(x) = a_{2,3} \cdot x^3 + a_{2,2} \cdot x^2 + a_{2,1} \cdot x + a_{2,0} \\ f_3(x) = a_{3,3} \cdot x^3 + a_{3,2} \cdot x^2 + a_{3,1} \cdot x + a_{3,0} \end{cases} \cdots \quad (3.9)$$

と表現されます。このモデルで学習するパラメータは、 $a_{i,j}$ で、合計12個ということになります。

まとめると、ソフトマックス関数に入力される $f_i(x)$ は、3つのメニュー項目の確からしさを意味し、ソフトマックス関数が出力する3つの値 p_i は、3つのメニューであるそれぞれの確率ということになります。

3.7.3 学習

Core MLは、scikit-learnの回帰のモデルをサポートするので、scikit-learnを使い、Pythonで機械学習のコードを書くことにします[14]。ここからは、Pythonのコードを使って説明していきます。まずデータを学習データとテストデータに分けます。

```
# データ
x = データの読み込み
label = データの読み込み
# データが100個あり、50個を学習データに、50個をテストデータと仮定します
dataCount = 100;

# ここまでがデータの生成。データを学習データとテストデータに、ランダムで切り分ける。
# テストデータは、学習に使ってはならない。
learnIndex = np.random.permutation(dataCount)[0:dataCount/2]
testIndex = np.random.permutation(dataCount)[dataCount/2:dataCount]

# 3次の多項式の基底を考えて、xの値を拡張する
X = np.c_[x, x * x, x * x * x]

# データを実際に切り分ける
learnX = X[learnIndex]
learnY = y[learnIndex]
learnLabel = label[learnIndex]

testX = X[testIndex]
testY = y[testIndex]
testLabel = label[testIndex]
```

scikit-learnには、ロジスティック回帰のモデルが実装されているので、それを使って学習するこ

とにします。scikit-learn を使った学習のコードは、これで終わりです。

```
from sklearn.linear_model import LogisticRegression
log_model = LogisticRegression()
log_model.fit(learnX, learnLabel)
```

`fit` メソッドがデータにモデルを"フィッティング"させるパラメータを求める、という意味を持つと考えるとよいでしょう。次に実際に、学習したパラメータの学習誤差とテスト誤差を計算してみましょう。`score` メソッドに、推論したい入力と、その正解データを引数として与えると、精度を得ることができます。このコードを使って、モデルや得られたパラメータの性能を評価するとよいでしょう。

```
# 性能評価
print(u' 学習性能（誤差） = %f' % log_model.score(learnX, learnLabel))
print(u' テスト性能（誤差） = %f' % log_model.score(testX, testLabel))
```

性能評価の出力結果は、次のようになります。

```
->学習性能（誤差） = 0.800000
->テスト性能（誤差） = 0.820000
```

また、推論は以下のコードで実行できます。

```
# scikit-learn で推論
predictByScikit = log_model.predict(testX)
```

3.7.4 予測と評価

あまりにもツールが強力すぎて、実際の学習や推論時に行われていることがさっぱりわかりません。そこで、実際に scikit-learn で学習して得られたパラメータを用いて、自分のコードで推論してみることにします。scikit-learn の `predict` メソッドを実行すると、ソフトマックス関数の中身がわかりません。そこで、その中身の可視化を目的に推論を自前で計算することにします。学習したパラメータは、`LogisticRegression` クラスのプロパティとしてアクセスすることができます。

```
print(log_model.intercept_)
print(log_model.coef_)
```

パラメータの出力結果は、次のようにになります。

```
[-0.82938622  0.28770332 -1.02881563]
[[ -2.48389756e+00   3.04420169e-01  -8.74362497e-03]
 [ 1.72626709e-01  -4.30660554e-02   1.67552070e-03]
 [-7.51704027e-01   6.67002983e-01  -5.84902671e-02]]
```

`intercept_` プロパティは、オフセット、つまり入力に依存しない固定の値を意味し、上で設計したモデルにおける、 $a_{1,0}, a_{2,0}, a_{3,0}$ に該当します。オフセットがないと、出力値は常に原点を通ることになり（入力が0ベクトルのとき、常に0を出力する）、表現能力が落ちてしまいます。また、`coef_` プロパティは、残りの回帰パラメータであり、 $a_{1,1}, a_{2,1}, a_{3,1}, a_{1,2}, a_{2,2}, a_{3,2}, a_{1,3}, a_{2,3}, a_{3,3}$ に該当します。上で言及したように、合計 12 個のパラメータが学習によって得られたことがわかります。

推論時には、得られたパラメータを (3.8) に代入し、ソフトマックス関数の 3 つの出力値を計算します。それぞれの出力値がメニュー 1 ~ 3 の確からしさを表します。

```
# 自前のコードで推論
# 入力に対して係数を乗算し、オフセットを足す
f = np.dot(log_model.coef_, testX.transpose())
    + np.matlib.repmat(log_model.intercept_, 50, 1).transpose()
# exponential をとる
numerator = np.exp(f)
# 式(?)の分母を計算する
denominator = np.sum(numerator, axis=0)
# 確率を計算する
p = numerator / denominator
```

図 3.16 に出力した p をプロットします。図 3.15 と比べると、ソフトマックス関数の出力値が元のメニューを再現できていることがわかります。

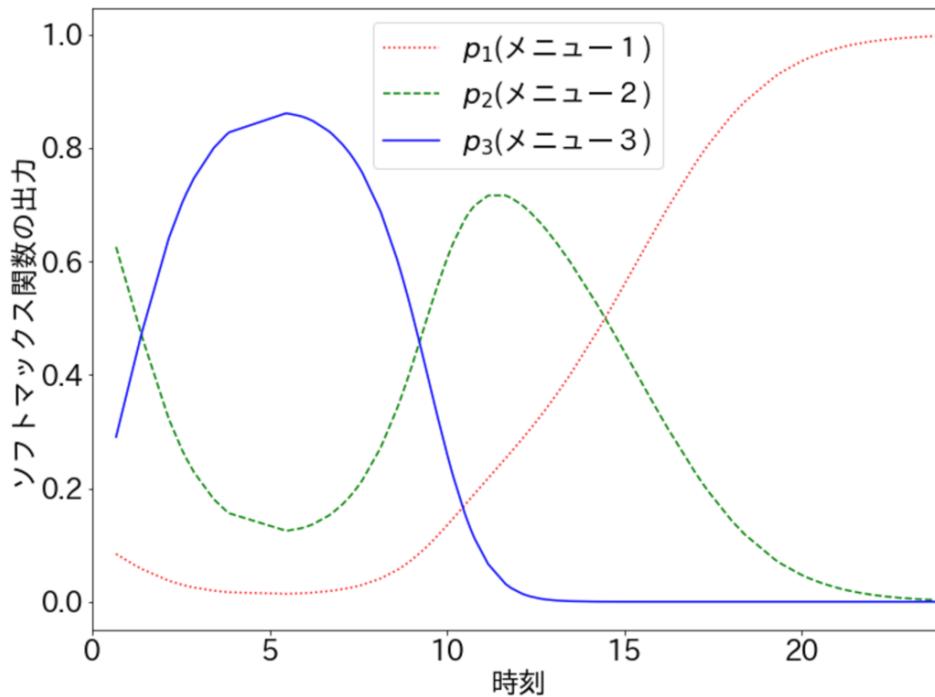


図 3.16: 推論結果

3.7.5 モデルデータの変換

次に、設計したモデルと学習したモデルを Core ML で扱えるように変換します。scikit-learn で作ったモデルを Core ML Tools のコンバータに渡すだけです。以下のコードは、書き出す際に付加情報を追加しています。入力変数名や出力変数名は、理解しやすいように定義しておいた方がよいでしょう。デプロイ時に `convert` メソッドで指定した変数名は、iOS で実装する時の変数名として利用します。さらに、作者名やライセンス、全体および変数ごとの説明文などを追記できます。

```
# CoreML 用にモデルとパラメータを書き出す
from coremltools.converters import sklearn
coreml_model = sklearn.convert(log_model, ["x1", "x2", "x3"], "label")
coreml_model.author = 'Yuichi Yoshida'
coreml_model.license = 'MIT'
coreml_model.short_description = '時刻からよく選ばれるメニューの項目を推定します。'
coreml_model.input_description['x1'] = u'1次の入力値。推定したい値を入力してください。'
coreml_model.input_description['x2'] = u'2次の入力値。推定したい値の2乗を入力してください。'
coreml_model.input_description['x3'] = u'3次の入力値。推定したい値の3乗を入力してください。'
```

```
coreml_model.output_description['label'] = u'推定した結果'
coreml_model.save('./output/MenuPrediction.mlmodel')
```

この Python のコードを実行すると、指定したパスに Core ML のモデルファイルが書き出されます。次にこのファイルを Xcode にインポートします。

プロジェクト内で mlmodel ファイルを選択すると、図 3.17 のようなプレビューが表示されます。ここで、変換時に設定した情報が表示されます。

さらに Xcode 上で mlmodel ファイルが、プロジェクトのターゲットに追加されていることを確認してください。ファイルがターゲットに追加されていないと、mlmodel ファイルがリソースに追加されず、モデルがコンパイル時にロードされないため、エラーが発生するので注意してください。

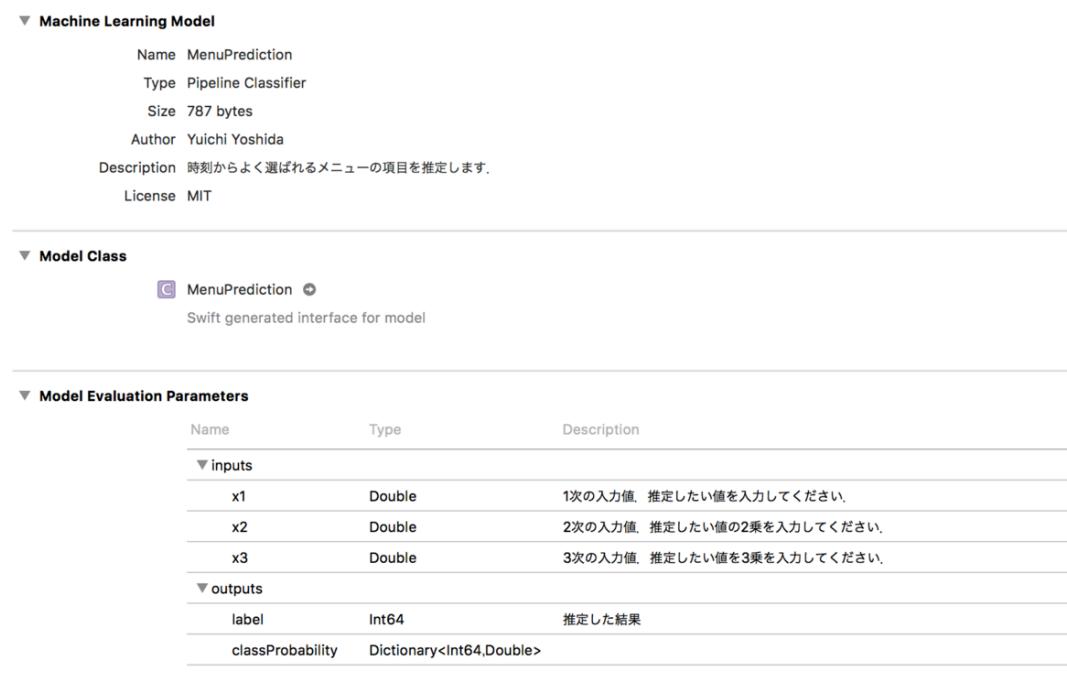


図 3.17: Xcode 上での mlmodel ファイルのプレビュー

3.7.6 Xcode 上での開発

ここから、Core ML を使った開発に移ります。Core ML を使うと、ネイティブアプリケーション向けに機械学習のためのコードをほとんど書く必要がありません。これは、Core ML Tools のモデルデータの変換機能と、Core ML のインターフェースが綺麗に設計されているためです。まず、モデルとパラメータをロードするところですが、Xcode が自動的にバイナリを mlmodel ファイルから生成してくれるため、

```
let model = MenuPrediction()
```

この一行だけで、モデルをロードすることができます。Core ML が強力であることが分かります。次に、モデルを使って推論してみます。

```
do {
    let x1 = Double(18);
    let x2 = x1 * x1;
    let x3 = x1 * x1 * x1;
    let result = try model.prediction(x1: x1, x2: x2, x3: x3)
    print(result)
    print(result.label)
    print(result.classProbability)
} catch {
    print(error)
}
```

推論は、`prediction` メソッドを使って実行します。`prediction` メソッドは、例外を投げる
ので、`do-catch` で実装します。`prediction` メソッドの引数は、`mlmodel` ファイルを書き出すとき
に設定した入力変数の名前がそのまま引き継がれます。推論の結果として、`MLFeatureProvider`
型のオブジェクトが返されます。その中身には、`mlmodel` ファイルを書き出すときに設定した出力
変数の名前を使ってアクセスすることができます。また、今回のロジスティック回帰のケースでは、
3 つの選択肢それぞれの推定された確率にもアクセスすることができます。

以上、scikit-learn のロジスティック回帰を Core ML で利用するアプリケーションの実装例を紹
介しました。機械学習のツール（ここでは scikit-learn）でモデル設計とパラメータの学習を実行し、
Core ML Tools でモデルを変換し、Xcode で実際の利用のためのコードを書くという一連の実装の
流れをご理解いただけたのではないかと思います。

3.7.7 CNN～Keras

二つ目の実装例では、"高い即時性が求められるケース"として、ユーザが指で画面に書いた数字を
認識するアプリケーションを取り上げます。実装するアプリケーションは、ニューラルネットワー
クを使って、数字を認識することにします。図 3.17 に、アプリケーションの画面を示します。ユー
ザが画面上の黒い領域に指で数字を書くと、Core ML のモデルがそれを入力として、書かれた数字
の種類を推論します。

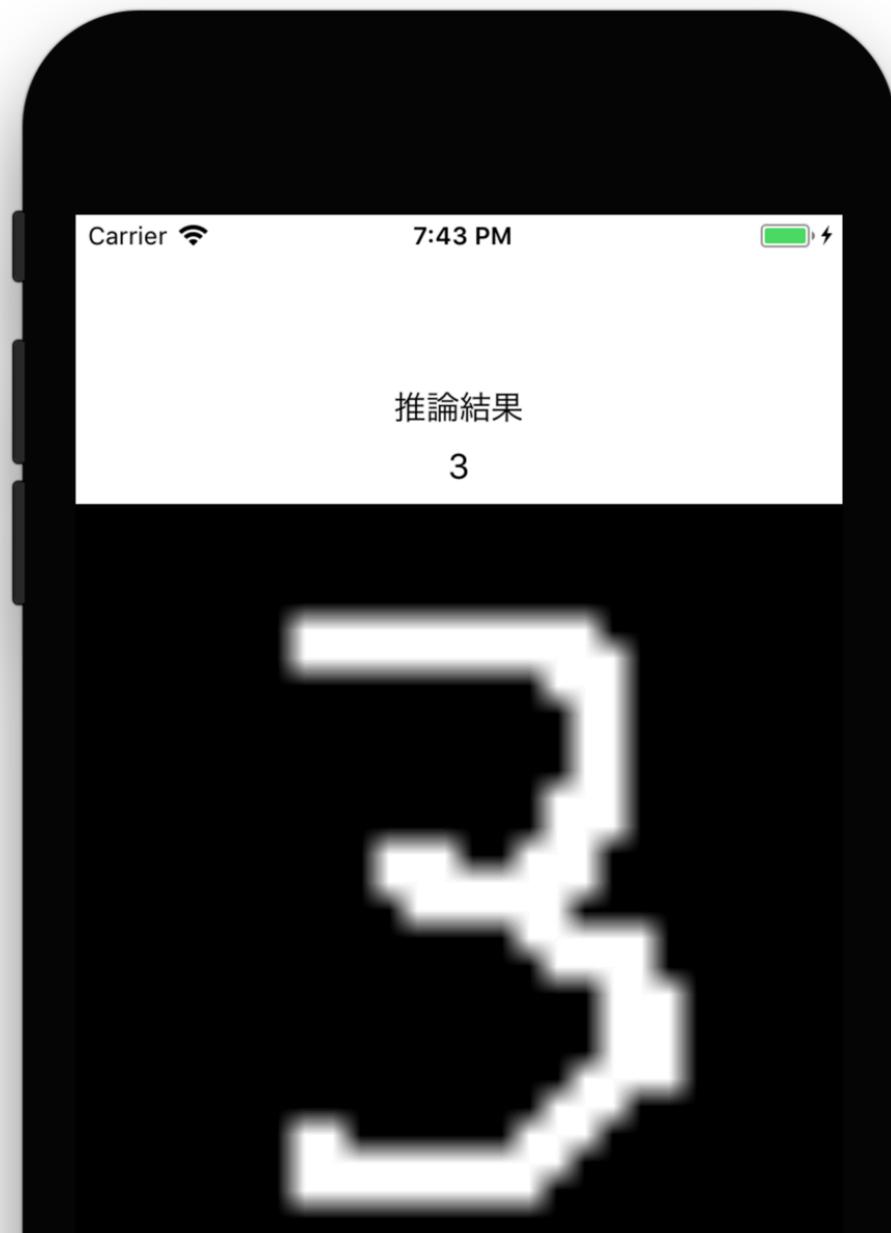


図 3.18: アプリケーション

本節のサンプルプロジェクトは samplecode/内にある iOS/RecChar/RecChar.xcodeproj です。本節のサンプルプロジェクトは samplecode/内にある python/keras/ActivationFunction.ipynb です。本節のサンプルプロジェクトは samplecode/内にある python/keras/keras.ipynb です。

ニューラルネットワークのモデルの設計とパラメータの学習には、Keras を使います [15]。Core ML は、ニューラルネットワークのツールの中でも、Keras と Caffe をサポートします。Keras は、ニューラルネットワークのツールをバックエンドとして動く、ニューラルネットワークを設計、パラメータを学習するためのツールです。Keras は、既存のツールより簡単にニューラルネットワークを実装したり、実験したりできるように開発されたラッパーのようなものと考えてください。Google

のTensorFlowやTheanoなどがKerasのバックエンドとして利用されています。

MNISTと呼ばれる手書きの数字のデータセットを使って、数字認識ができるニューラルネットワークのパラメータを学習します[16]。MNISTとは、アメリカ国立標準技術研究所(NIST)によって作成・公開されている手書き数字のデータセットです。このデータセットは、実際に人に書いてもらった60000枚の学習用の数字の画像(学習データ)と、10000枚のテスト用の画像(テストデータ)から構成されます。

3.7.8 ニューラルネットワーク

ニューラルネットワークの簡単な説明から始めます。ニューラルネットワークのモデルは、線形演算と活性化関数と呼ばれる非線形関数(乱暴に表現すると単純な1次方程式では表現できない関数)を多段かつ並行につなげたものです。シンプルな2層のニューラルネットワークを図3.19に示します。この例では、入力を \mathbf{x} 、中間層(隠れ層)の値を \mathbf{z} 、1層目の重みを行列 $\mathbf{W}^{(1)}$ 、2層目の重みを行列 $\mathbf{W}^{(2)}$ 、出力を \mathbf{y} 、1層目の活性化関数を f_1 、2層目の活性化関数を f_2 としています。

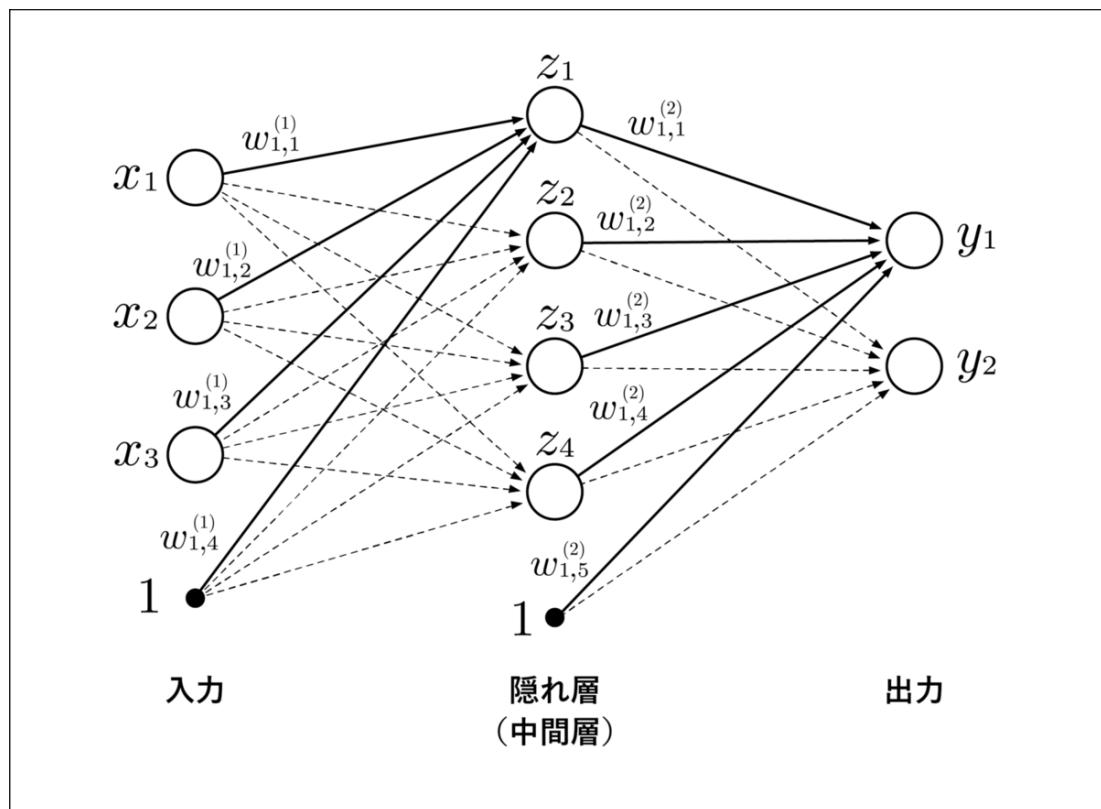


図3.19: 2層のシンプルなニューラルネットワーク

図3.19の例のように、ニューラルネットワークの構成を、入力が3次元のベクトル \mathbf{x} 、中間層 \mathbf{z} が4つの値を持ち(4次元)、出力が2次元のベクトル \mathbf{y} とすると、1層目の計算は、

$$\begin{cases} z_1 = f_1(W_{1,1}^{(1)} \cdot x_1 + W_{1,2}^{(1)} \cdot x_2 + W_{1,3}^{(1)} \cdot x_3 + W_{1,4}^{(1)}) \\ z_2 = f_1(W_{2,1}^{(1)} \cdot x_1 + W_{2,2}^{(1)} \cdot x_2 + W_{2,3}^{(1)} \cdot x_3 + W_{2,4}^{(1)}) \\ z_3 = f_1(W_{3,1}^{(1)} \cdot x_1 + W_{3,2}^{(1)} \cdot x_2 + W_{3,3}^{(1)} \cdot x_3 + W_{3,4}^{(1)}) \\ z_4 = f_1(W_{4,1}^{(1)} \cdot x_1 + W_{4,2}^{(1)} \cdot x_2 + W_{4,3}^{(1)} \cdot x_3 + W_{4,4}^{(1)}) \end{cases} \cdots (3.10)$$

と書き下すことができます。また、2層目の出力層の計算は、

$$\begin{cases} y_1 = f_2(W_{1,1}^{(2)} \cdot z_1 + W_{1,2}^{(2)} \cdot z_2 + W_{1,3}^{(2)} \cdot z_3 + W_{1,4}^{(2)} \cdot z_4 + W_{1,5}^{(2)}) \\ y_2 = f_2(W_{2,1}^{(2)} \cdot z_1 + W_{2,2}^{(2)} \cdot z_2 + W_{2,3}^{(2)} \cdot z_3 + W_{2,4}^{(2)} \cdot z_4 + W_{2,5}^{(2)}) \end{cases} \cdots (3.11)$$

と展開できます。このとき、1層目の重み行列 $\mathbf{W}^{(1)}$ は、 5×4 の行列となり、2層目の重み行列 $\mathbf{W}^{(2)}$ は、 2×5 の行列となります。それぞれの層の変数の数よりも重み行列のサイズが一つ大きいのは、オフセットがあるためです。ニューラルネットワークでは、データを使って、この重み行列の値を学習することになります。複雑なニューラルネットワークは、この中間層の数や変数の数が多くなっていきます。後述する畳み込み層に対して、(3.10) や (3.11) のようにすべての入力と出力の間に重みを割り当てる層を全結合層 (fully connected layer) と呼びます。

3.7.9 活性化関数

活性化関数は、ニューラルネットワークが非線形なデータを表現できるようにするために重要なものです。活性化関数は、非線形関数であり、複雑な入出力関係を実現するために必要であると考えてください。活性化関数をなくしてしまったり、活性化関数に線形関数を採用すると、ニューラルネットワークは、ただの線形変換と同じになってしまいます。活性化関数には、シグモイド関数 ((3.12), 図 3.21)、ReLU((3.13), 図 3.21) などの関数が使われます。

$$y = \frac{1}{1 + e^{-a \cdot x}} \cdots (3.12)$$

$$y = \begin{cases} x & x \geq 0 \\ 0 & \text{else} \end{cases} \cdots (3.13)$$

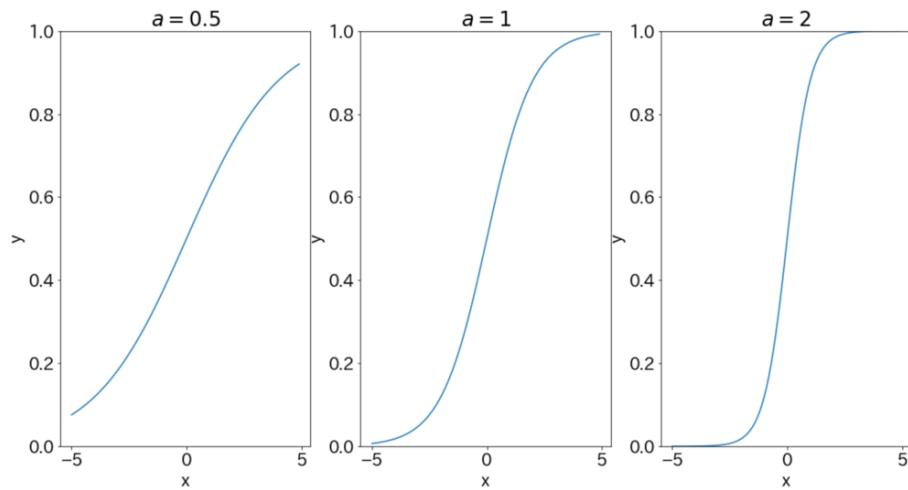


図 3.20: シグモイド関数

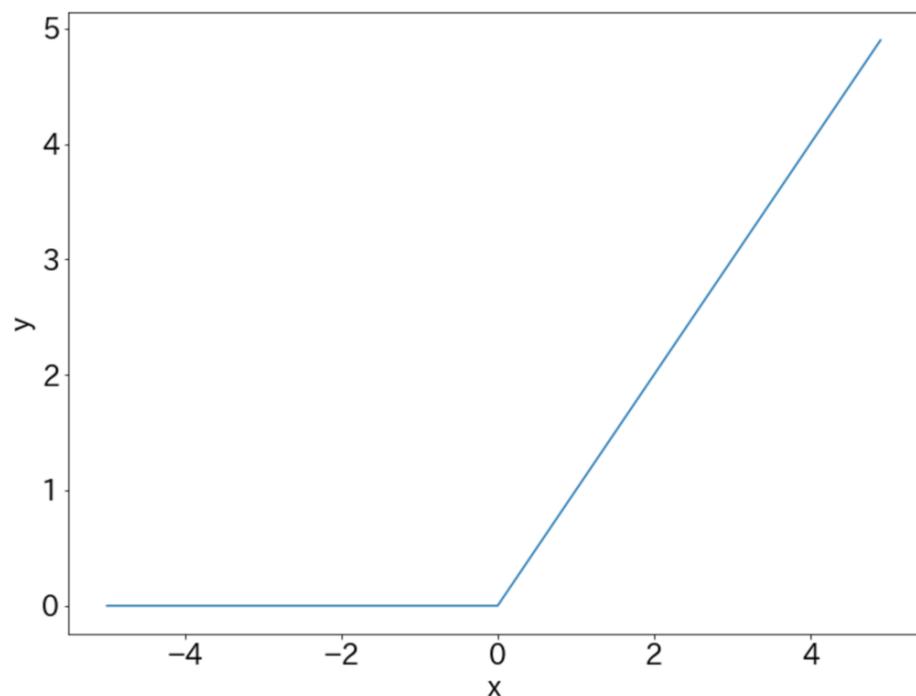


図 3.21: RELU

3.7.10 プーリング

プーリング (pooling) と呼ばれる、入力ベクトルの中の最大値 (maxpooling) や最小値を抽出する処理も使われます。プーリングは、画像の特定の領域の大まかな情報を表現するために使われます。

3.7.11 畳み込みニューラルネットワーク

畳み込み層 (convolutional layer) は、すべての入力と出力の間に異なる重みを割り当てるのではなく、画像処理のフィルタのような入力と出力の間に共通するフィルタで重み付けするものです。そして、この畳み込み層を持つニューラルネットワークを畳み込みニューラルネットワーク (convolutional neural network) と呼ばれることもあります。畳み込みニューラルネットワークでは、この重みを、データから計算します。畳み込みの計算の端のフィルタがはみ出る部分の処理の仕方によって、畳み込みで出力されるベクトルの次元は異なります。畳み込みが 3×3 のフィルタであり、入力が 4×4 の画像、つまり 16 次元のベクトルであり、畳み込みではみ出す領域を計算しない場合、次の層（隠れ層）の次元は、自動的に 2×2 の画像、つまり 4 次元ベクトルになります。

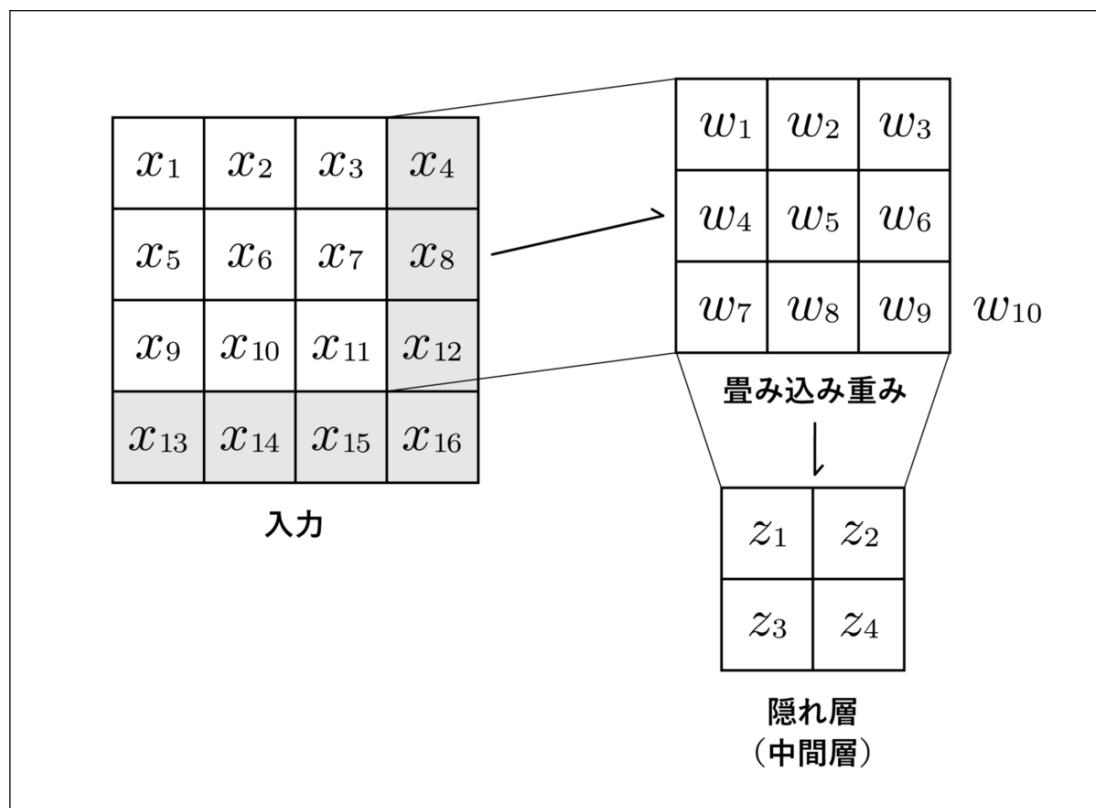


図 3.22: 畳み込み計算の概念図

図 3.22 を例に具体的な畳み込み計算を説明します。 z_1 の計算方法は、

$$\begin{aligned}
 z_1 = & W_1^{(1)} \cdot x_1 + W_2^{(1)} \cdot x_2 + W_3^{(1)} \cdot x_2 \\
 & + W_4^{(1)} \cdot x_5 + W_5^{(1)} \cdot x_6 + W_6^{(1)} \cdot x_7 \quad \cdots (3.15) \\
 & + W_7^{(1)} \cdot x_9 + W_8^{(1)} \cdot x_{10} + W_9^{(1)} \cdot x_{11} \\
 & + W_{10}^{(1)}
 \end{aligned}$$

となり、一つ隣の出力 z_2 の計算方法は、畳み込むフィルタをひとつずらして、

$$\begin{aligned}
 z_2 = & W_1^{(1)} \cdot x_2 + W_2^{(1)} \cdot x_3 + W_3^{(1)} \cdot x_4 \\
 & + W_4^{(1)} \cdot x_6 + W_5^{(1)} \cdot x_7 + W_6^{(1)} \cdot x_8 \quad \cdots (3.16) \\
 & + W_7^{(1)} \cdot x_{10} + W_8^{(1)} \cdot x_{11} + W_9^{(1)} \cdot x_{12} \\
 & + W_{10}^{(1)}
 \end{aligned}$$

となります。畳み込み層で、フィルタ 5×5 と設計すると、これにオフセットの数を加えて、パラメータの数は $25+1$ になります。また、 3×3 のフィルタを畳み込み層に使った場合、パラメータ数は 10 個になります。一方、全結合層を適応した場合、入力が 4×4 の画像、つまり 16 次元のベクトルで、 2×2 の画像、つまり 4 次元ベクトルであるとき、 17×4 の行列が重み行列となり、68 個のパラメータを持つことになります。畳み込み層のパラメータ数は、全結合層と比較すると少なく、過学習を起こしにくいため、有効に働くことが知られています。

3.7.12 最適化

ニューラルネットワークの全体構成と活性化関数が決まると、次にパラメータである各層の重み行列をデータを使って学習します。ニューラルネットワークでは、誤差逆伝搬法と呼ばれるアルゴリズムでパラメータを学習します。scikit-learn と同じく、ニューラルネットワークのツールを使う場合、開発者がパラメータの最適化に関する実装する必要はありません。ただ、誤差逆伝搬法自体は難しいものではないので、具体的な最適化方法について、興味のある方は、参考文献を読みながら、自分で実装して見るとよいと思います [1][17][18]。

3.7.13 過学習と歴史

前節で、機械学習では、モデルが複雑になると過学習が起こりやすくなると説明しました。これは、モデルがニューラルネットワークであっても同様です。1 層のニューラルネットワークを考えて、入力を画像とし、出力を $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ の 10 種類の数字の確からしさ、とする例を考えます。MNIST を使う場合、入力が $784 (= 28 \times 28)$ 次元のベクトル、出力が 10 次元のベクトルということになります。その 1 層目が全結合層のとき、重み行列 \mathbf{W} は、 $(784+1) \times 10$ の行列になります。学習で計算しなければならないパラメータの数は 7850 個に及びます。

最近までニューラルネットワークは、規模を大きくし、パラメータが多くなりすぎると、計算量の観点からそれらのパラメータを計算することも、学習理論の観点から汎化性能を下げないように学習することも極めて難しいと考えられてきました。しかし、2006 年に Hinton らが Deep belief network を提案したことをきっかけに、ニューラルネットワークの研究は爆発的に進展しました [17][18]。従来ではうまく学習できないとされた多層構造を持つネットワークのパラメータを学習するテクニックが多く提案されました。また、同時にニューラルネットワークのパラメータの学習に GPU を使うことで、計算量的にも不可能と言われていた規模のネットワークが現実的な時間で学

習できるようになってきました。驚くべきことに、2014年にGoogleが発表した画像認識のためのニューラルネットワークGoogleNetは、40層の深さを持ち、およそ700万個のパラメータを持ちます[19]。さらに、現在では、1000層を超えるようなネットワークの構成方法や学習方法が提案されています[20]。

今日のニューラルネットワークの研究が爆発的に進化した、もうひとつの理由はソフトウェアにあります。多くの研究機関、大学、企業が、ニューラルネットワークの学習・推論のためのツールを開発し、オープンソースで公開しました。それらのツールは、ネットワークの設計や実験を容易にするために開発されたものであり、ニューラルネットワークの学習に共通するパラメータの学習の部分を隠蔽した作りになっています。結果、雨後の筍のように提案される手法の追実験や改善が容易になり、ニューラルネットワークの研究は進展してきました。ニューラルネットワークの研究は、研究者、GPU、そして、オープンソースの力を得て、今日の発展を遂げたと私は考えています。

3.7.14 設計指針

今回の例では、なるべく小さなニューラルネットワークを設計することにします。小さなネットワークは過学習しにくく、推論も高速に実行でき、メモリの消費も小さいという利点があります。

Kerasには、MNISTのデータを管理するクラスが用意されているので、それを使います。本来ならば、自分で画像と、その画像のラベルを用意する必要があります。今回のモデルは、畳み込み層(ReLU)、畳み込み層(ReLU)、MaxPooling(小ブロックごとの最大の値を出力する層)、畳み込み層(ReLU)、全結合層(ReLU)、最後はソフトマックス関数で出力するという構成をとります。ソフトマックス関数がここでも出てくるのは、最後の出力値として、10個の値を出力するソフトマックス関数を使うことで、どの文字が相応しいかを確率のように(10個の値の合計が1になる)出力することができます。この構成をKerasで書くと、以下のようになります。

```
### ネットワーク設計
model = Sequential()
model.add(Conv2D(32, kernel_size=[3, 3], padding='same', input_shape=(img_height, img_width, 1)))
model.add(Activation('relu'))
model.add(Conv2D(32, kernel_size=[3, 3], padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(16, kernel_size=[3, 3], padding='same'))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
# num_classes = 10 識別対象の種類
model.add(Dense(num_classes, activation='softmax'))
```

これでニューラルネットワークの設計が終わりました。Kerasのようなツールを使うと、(3.11)～(3.16)に示したニューラルネットワークを構成するために必要な計算をまったく実装する必要がありません。また、ツールがない場合、パラメータを学習するためには、これらの式の微分を自分で実装する必要がありますが、それもすべて自動で行われます。この上のコードだけですべてが完結してしまうことに、CaffeやKerasに代表されるニューラルネットワークのツールの生産性の高さ

が如実に現れていると思います。

`summary` メソッドを実行すると、ネットワークの内容が出力されます。実際の構成とパラメータ数を確認してください。次に、学習のための誤差関数、パラメータの更新アルゴリズム、テスト時の評価指標をセットします。今回は、ソフトマックス関数を最終層に使っているため、クロスエンタロピーと呼ばれる誤差関数を用います。パラメータの更新には、RMSProp と呼ばれる更新方法を使っています。パラメータの更新アルゴリズムはいろいろあるので、論文や Keras のドキュメントを参考にしてください。また、テスト時の指標は、当然、画像に書かれた文字を当てられるかが重要なので、正解できるかの精度を評価指標としてセットします。

```
### ネットワークの構成を出力する
model.summary()
# 誤差関数、パラメータの更新アルゴリズム、テスト時の評価指標をセット
model.compile(loss='categorical_crossentropy',
               optimizer=RMSprop(),
               metrics=['accuracy'])
```

最後に学習し、得られたパラメータの性能を評価します。

```
model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           verbose=1,
           validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

3.7.15 モデルファイルの書き出し

Xcode にインポートできるように、完成したモデルを Core ML Tools を使って、`mlmodel` ファイルに書き出します。前節のサンプルと同様に、変数名や説明文、ライセンスなどの情報をセットします。

```
import coremltools

# モデル変換
# 入力と出力の変数名もセットする
coreml_model = coremltools.converters.keras.convert(
    model,
    input_names='image',
    output_names='digit')
```

```
# モデルの付加情報をセット
coreml_model.author = u'Yuichi Yoshida'
coreml_model.license = 'MIT'
coreml_model.short_description = u'iOS11 Programming のサンプル'

# 入力と出力の説明文をセットする
coreml_model.input_description['image'] = u'入力画像',
coreml_model.output_description['digit'] = u'推定した数字の確率',

# mlmodel ファイルへの書き出し
coreml_model.save('KerasMNIST.mlmodel')
```

Core ML Tools で書き出した mlmodel ファイルを Xcode にインポートします。モデルの変換等が正しく実行されていれば、以下のようなプレビュー (図 3.23) が Xcode 上で確認できるはずです。

▼ Machine Learning Model

Name	KerasMNIST
Type	Neural Network
Size	1.2 MB
Author	Yuichi Yoshida
Description	iOS11 Programmingのサンプル
License	MIT

▼ Model Class

 KerasMNIST ⓘ
Automatically generated Swift model class

▼ Model Evaluation Parameters

Name	Type	Description
▼ inputs		
image	MultiArray (Double 1 x 28 x 28)	入力画像
▼ outputs		
digit	MultiArray (Double 10)	推定した数字の確率

図 3.23: Keras のモデルから、変換後エクスポートされた mlmodel ファイル

アプリへの組み込み～テストコード

インポートしたモデルを使い、手書き文字認識アプリを実装していきます。推論については、Core ML が肩代わりしてくれるので、まずは、Core ML で MNIST のテストデータを読み込み、汎化性能を iOS 上で評価する仕組みを実装します。

Xcode で確認すると、入力する形式が、MLMultiArray 型の 1x28x28 のサイズになっています。この定義に合わせて、入力データを Core ML のモデルの predict メソッドに渡す必要があります。

MNIST のデータは、バイナリファイルで提供されているためファイルから直接読み込みます。バイナリファイルの読み込みに関する実装の詳細は、サンプルコードを参考してください。

MNIST のバイナリデータは、行ごとにデータが保存されています。今回のモデルへの入力は、1 チャンネル（グレースケールの画像）、行（画像の縦方向）、列（画像の横方向）の順に並んだデータを入力します。次のように `MLMultiArray` 型にデータを渡します。実装するときは、画像データの並びに注意してください。

```
do {
    let input = try MLMultiArray(
        shape: [1, NSNumber(value: height), NSNumber(value: width)],
        dataType: .double)
    for row in 0..<height {
        for column in 0..<width {
            let label: UInt8 = try read(from: handle).bigEndian
            let value = Double(label) / 255.0
            input[[0, NSNumber(value: row),
                   NSNumber(value: column)]] = NSNumber(value: value)
        }
    }
} catch {
    print(error)
}
```

ニューラルネットワークのモデルを生成し、画像中の文字を推論し、正解率を集計するコードは次のとおりです。`loadLabel` と `loadImage` は、指定されたインデックスのテストデータを MNIST のバイナリファイルから読み出すメソッドです。ここまでコードが正しく実装されていれば、Keras で `model.evaluate` メソッドで測定したときと同じ精度が、このコードで得られるはずです。

```
do {
    let model = KerasMNIST()
    let count = 1000
    var trueCount = 0
    for i in 0..<count {
        let label = try loadLabel(index: i)
        let image = try loadImage(index: i)
        let result = try model.prediction(image: image)
        if label == result.digit.maxIndex {
            trueCount += 1
        }
    }
    let precision = Double(trueCount) / Double(count) * 100
    print(precision)
} catch {
    print(error)
}
```

アプリへの組み込み～ユーザー入力

次にユーザーの入力を prediction メソッドに渡す部分のコードを紹介します。一筆書きを前提として、ユーザーは、画面をタップし、ドラッグして、画面上に数字を描き、画面から指を離すと、画面に描かれた文字を推論させるようにします。

サンプルでは、ドラッグ中の指で描かれた文字をメモリに保存するコードを UIViewController の touchesMoved メソッドで実装しました。UIViewController のサブクラスにユーザーが描いた文字を保存するための配列 pixelBuffer32bit を用意しておきます。そして、タップの位置を UITouch から取得し、配列 pixelBuffer32bit の対応する場所に値をセットします。下のコードで、配列 pixelBuffer32bit に二重ループで値をセットしているのは、ユーザが書いた線を太くするためにです。Core ML に引き渡して推論するとき、ユーザがタップした中心の場所だけだと、その線が細すぎて認識しにくいためです。

```
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let touch = touches.first else { return }

    let location = touch.location(in: self.view)
    if imageView.frame.contains(location) {
        let locationInImageView = imageView.convert(location, from: self.view)

        let xInView = locationInImageView.x / imageView.frame.size.width
        let yInView = locationInImageView.y / imageView.frame.size.height

        let xIndex = Int(xInView * CGFloat(ViewController.width))
        let yIndex = Int(yInView * CGFloat(ViewController.height))

        let blockSize = 1
        let left = xIndex - blockSize > 0 ? xIndex - blockSize : 0
        let right = xIndex + blockSize < ViewController.width
            ? xIndex + blockSize : ViewController.width - 1
        let top = yIndex - blockSize > 0 ? yIndex - blockSize : 0
        let bottom = yIndex + blockSize < ViewController.height
            ? yIndex + blockSize : ViewController.height - 1

        for x in left..
```

次のコードは、タップ&ドラッグ（つまり文字を画面に書く操作）が終わった時に呼ばれます。ここでは、配列 pixelBuffer32bit に保存したユーザが書いた文字を、MLMultiArray 型の行列である input にコピーし、KerasMNIST の prediction メソッドに引き渡します。Core ML のランタ

イムは、KerasMNIST のモデルとパラメータを用いて、入力された配列 `input` に対する推論を計算し、結果を返します。アプリケーションは、その推論結果に、Core ML Tools で指定した名前(図図 3.23)でアクセスできます。

```
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    do {
        let input = try MLMultiArray(
            shape: [1, NSNumber(value: ViewController.height),
                    NSNumber(value: ViewController.width)],
            dataType: .double)
        for x in 0..
```

もし、カメラで撮影した画像を使いたい場合は、AVFoundation の `AVCaptureVideoDataOutput` クラスを使って、カメラ画像をキャプチャし、`CVImageBuffer` クラスからピクセルデータを読み取り、`MLMultiArray` 型へ変換すればよいでしょう。

以上で、Keras で設計・学習したニューラルネットワークを Core ML で利用する方法を紹介しました。Core ML を使えば、scikit-learn を使ったアプリと同様に、ニューラルネットワークの設計・学習がアプリ実装と分離されるため、簡単にその推論能力をアプリに取り込むことができることを紹介しました。一方で、すべてが簡単にいくわけではなく、画像などの多次元データを入力に使う場合には、開発者は、Core ML のモデルで推論を実行する時に入力を `MLMultiArray` 型にデータを変換する必要があります。特に画像の場合は、RGB の順番、1 行ずつデータが入っているのか、1 列ずつデータが入っているのかなどに注意する必要があります。

3.8 Core ML の短所

ここまで概要と実装例の説明で、機械学習を応用する際に Core ML を使わない手はないように感じられますが、実際にそうなのでしょうか？

Core ML の短所はあるのでしょうか。残念ながら、Core ML にはいくつかの課題、問題点があります。ここでは、それらについて説明します。

モデルを自由に構成できない

Core ML がサポートする機械学習のモデルは、前節で述べたモデルだけに限られます。ゆえに、Core ML の枠組みの中で、数多くある機械学習のモデルを開発者が自由に使えるわけではありません。mlmodel ファイルの仕様は、Protocol buffer の定義ファイルとして公開されていますが、Core ML によってサポートされる機械学習のモデルの範囲を超えるような mlmodel ファイルを開発者が作ることはできません。つまり、Apple が実装した機械学習のランタイムに、開発者がパラメータを流し込むのが、現状の Core ML の本質です。これは、Core ML の実装上の大好きな課題です。例えば、Apple が Core ML 内で実装していない活性化関数を開発者がモデルの中で使うことはできません。

機械学習は、現在、日進月歩の進化を見せてています。これを応用していくためには、Core ML の内部を Apple が逐次更新、追加していくか、あるいは Core ML 自体をオープンソースにしてしまい、新機能の追加をコミュニティと共同で実装していくのかしかありません。実際、今日大ブームになっているニューラルネットワークの研究分野では数ヶ月単位のペースで新しい手法が発表されると言っても過言ではない状態です。Apple が Core ML のランタイムをプライエタリに開発していくスタイルで、新技術に追従していくのは、非現実的と言わざるを得ません。私は、Core ML tools だけではなく、Core ML 自体も同時にオープンソースにしなければ、長いスパンで Core ML が開発者に支持されることは難しいと考えます。

モデルとパラメータを秘匿できない

これは、機械学習を運用する上でもっとも重要な課題です。Core ML ではモデルやパラメータを隠蔽することができません。Core ML で扱うモデルは、アプリケーション内部にリソースとして保存されます。ビルドしたアプリケーションパッケージの中身を確認すると、モデルやパラメータが内包されていることがわかります。実際に前節で説明した Keras を使った数字認識のサンプルの中身を確認すると、ネットワークの構成等が JSON ファイルがリソースとして保存されていることがわかります(図 3.24)。model.espresso.net や model.espresso.shape ファイルをテキストエディタでひらけば、ニューラルネットワークの各層の情報を確認することができます。また、model.espresso.weight には、バイナリ形式でパラメータが格納されています。

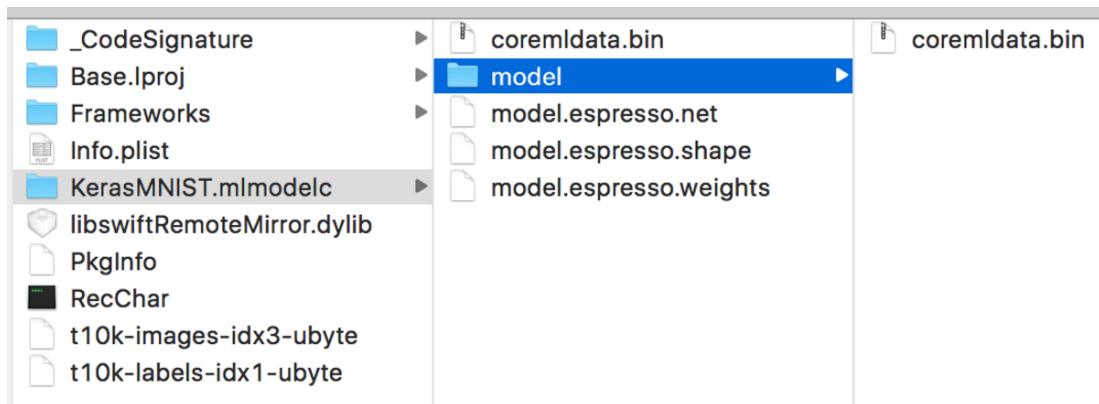


図 3.24: アプリケーションのパッケージ内の Core ML のモデルデータ

モデルやパラメータは、開発者の貴重な知的財産です。Core ML を使って配布する限り、これを隠すことはできません。この問題は、ローカルデバイスにバンドルされるデータ（OAuth のアプリケーションキーや、本章の場合、mlmodel ファイル）の秘匿に関するものであり、未だに解決されない重要な課題のひとつです。ユーザの端末に配信したデータを例え暗号化しても、データを利用する瞬間は復号化しなければなりません。今のところ、アプリケーションにバンドルしたデータを悪意のあるユーザが覗き見ることを防ぐ手段はありません。

以上の観点から、モデルやパラメータを完全に秘匿したい場合は、開発者は Core ML を使うべきではありません^{*8}。Core ML を使うときは、誰が設計・学習しても似たようなモデル、パラメータが得られそうな問題に限定すべきかもしれません。

モデルの動的な更新が難しい

Core ML は、モデルとパラメータを事前に設計・学習し、運用時は、それらを用いて推論をするフレームワークです。それゆえ、アプリケーション実行中にバンドルした mlmodel のモデルやパラメータを更新できません。しかしながら、実装を工夫し、mlmodel ファイル自体をネットワーク越しに配信することで、アプリケーションバイナリをアップデートせずにモデルとパラメータを更新できます。

本節のサンプルプロジェクトは samplecode/内にある iOS/UpdateModel/UpdateModel.xcodeproj です。

まず、通常のバンドルされた mlmodel ファイルの扱いについて説明します。Xcode は、拡張子が mlmodel であるファイルがプロジェクトに追加されると、自動的にバックグラウンドで、mlmodelc パッケージにコンパイルします。続いて、Xcode は、アプリケーションをビルドするときに、mlmodelc パッケージをリソースとして追加します。アプリケーションが起動されると、Core ML のランタイムが、mlmodelc ファイルから、MLModel クラスのサブクラスを作成します。これが、Core ML の基本的なランタイム時の動作の流れです。

一方で、ビルド時にバンドルする以外の方法で mlmodel ファイルをアプリケーションにロードさ

^{*8} これは、Core ML に限った話ではなく、ローカルデバイスに機械学習に関するコードを自前で実装したときも同様です。

せる手段もあります。それは、ドキュメントディレクトリに保存した mlmodel ファイルをコンパイルし、モデルオブジェクトを生成する方法です。まず、新しい mlmodel ファイルを、ネットワークのどこかに置いておき、iOS のアプリケーションにそれをダウンロードし、ドキュメントディレクトリに保存します。このとき、mlmodel ファイルはまだコンパイルされていないので、Core ML のランタイムで開くことができません。Core ML には、mlmodel ファイルを mlmodelc にコンパイルするために、`MLModel` クラスに `compileModel(at:)` というクラスメソッドが用意されています。このメソッドを使うと、mlmodel ファイルと同じパスに、mlmodelc という拡張子を持つフォルダが作成され、そこに mlmodel ファイルのコンパイル結果がコピーされます。そして、`MLModel` クラスの `init(contentsOf:)` メソッドで、その mlmodelc フォルダのパスを指定すると、Core ML の `MLModel` クラスのサブクラスのインスタンスを生成できます。サンプルコードは、以下になります。mlmodel ファイルの元ファイルの拡張子が bin になっているのは、Xcode が自動的に mlmodel ファイルをコンパイルすることを実験的に阻止するためです。

```
do {
    guard let url = Bundle.main.url(
        forResource: "KerasMNIST", withExtension: "bin") else { return }
    let compiledURL = try MLModel.compileModel(at: url)
    let model = try MLModel(contentsOf: compiledURL)
} catch {
    print(error)
}
```

以上のように mlmodel ファイルのコンパイルと実行環境で行うことで、アプリケーションバイナリを更新せずに Core ML のモデルを更新することができます。

以上、本節で議論したことと同様のことが [21] でも議論されています。Core ML を導入することの是非は、極めて、慎重かつ精密に議論されるべきです。コストやノウハウの秘匿を含めて、慎重に Core ML を利用するかどうかを検討してください。

3.9 まとめ

本章では、Core ML の概要、Core ML のための機械学習の基礎、Core ML を使ったアプリケーションの実装例について説明しました。Core ML は、機械学習のモデルを設計し、そのパラメータを学習するタスク・環境と、実際にアプリケーションに組み込むタスク・環境を分離してくれます。さらに、Core ML を使うと、開発者は、機械学習の推論を実装する必要がありません。一方で、Core ML には、モデルを自由に組めないという課題、モデルやパラメータを秘匿できない課題などの短所があります。Core ML のために学ぶ機械学習で解説したように、機械学習をアプリケーションに応用するために重要なのは、Core ML を使いこなすことではなく、機械学習で解決したい問題に対する知識、そのデータをたくさん集めて、性能のよいモデル・パラメータを得ることです。これらのことが適切に実施されなければ、Core ML の完成度がいくら高くても、開発者は何の成果も得ることはできません。

推論の実装の一切を肩代わりし、かつ高速化してくれる Core ML は、極めて、強力なフレーム

ワークです。もし、あなたの開発しているアプリケーションに機械学習で解決できそうな問題があるならば、Core ML の短所も考慮しながら、それを取り入れてみていはいかがでしょうか。

3.10 参考文献

- [1] C.M. ビショップ (2012) 『パターン認識と機械学習 上』 (元田浩 (監訳), 栗田多喜夫 (監訳) · 他訳) 丸善出版.
- [2] C.M. ビショップ (2012) 『パターン認識と機械学習 下』 (元田浩 (監訳), 栗田多喜夫 (監訳) · 他訳) 丸善出版.
- [3] Richard O. Duda, Peter E. Hart, David G. Stork (2001) 『Pattern Classification』 Wiley-Interscience.
- [4] 佐藤一誠 (2016) 「ノンパラメトリックベイズ 点過程と統計的機械学習の数理 (機械学習プロフェッショナルシリーズ)」講談社。
- [5] 石井 健一郎、上田 修功 (2014) 「続・わかりやすいパターン認識—教師なし学習入門」 オーム社。
- [6] 牧野貴樹、他 (2016) 「これから学習」 森北出版。
- [7] 本多淳也、中村篤祥 (2016) 「バンディット問題の理論とアルゴリズム (機械学習プロフェッショナルシリーズ)」 講談社。
- [8] 金谷健一 (2005) 『これなら分かる最適化数学—基礎原理から計算手法まで』 共立出版.
- [9] 金谷健一 (2003) 『これなら分かる応用数学教室—最小二乗法からウェーブレットまで』 共立出版.
- [10] Apple (2017) 「Converting Trained Models to Core ML」 , https://developer.apple.com/documentation/coreml/converting_trained_models_to_core_ml 2017年7月23日アクセス.
- [11] Apple (2017) 「Core ML」 , <https://developer.apple.com/documentation/coreml/> 2017年7月23日アクセス.
- [12] jupyter, "http://jupyter.org"
- [13] Apple(2017,) "Core ML in depth" <https://developer.apple.com/videos/play/wwdc2017/710/> "WWDC 2017 - Session 710 - iOS, macOS, tvOS, watchOS"
- [14] scikit-learn, "http://scikit-learn.org"
- [15] Keras, "<https://keras.io>"
- [16] MNIST, "<http://yann.lecun.com/exdb/mnist/>"
- [17] 岡谷貴之 (2015) 『深層学習 (機械学習プロフェッショナルシリーズ)』 講談社.
- [18] Ian Goodfellow, Yoshua Bengio, Aaron Courville (2016) 『Deep Learning (Adaptive Computation and Machine Learning series)』 The MIT Press.
- [19] C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9.
- [20] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 770-778.

- [21] Alex Sosnovshchenko (2017) 「Why Core ML will not work for your app (most likely)」 ,
<http://alexsosn.github.io/ml/2017/06/09/Core-ML-will-not-Work-for-Your-App.html> 2017年8月24日アクセス.

第 II 部

Swift 4, Xcode 9

第4章

Swift 4 の新機能とアップデート

4.1 はじめに

本章では、Swift 4 で追加された機能のうち、Foundation フレームワークに深く関係する、`Codable` プロトコルと Smart KeyPaths について解説します。

■コラム: Swift 4 コンパイラの言語モード

Swift 4 のコンパイラは Swift 3.2 モードと、Swift 4 モードという 2 つの言語モードをサポートしています。前者は Swift 3.x と基本的に互換性があるモードで、後者が Swift 4 の全ての機能と API 変更が有効になるモードです。Swift 4 モードでビルドしたフレームワークを、Swift 3.2 モードでビルドするアプリで使用する、といった混在利用が可能です。言語モードはプロジェクト単位、もしくはターゲット単位で、“Swift Language Version” という Build Settings から変更できます。

実は Swift 4 の新機能のほとんどは Swift 3.2 モードでも利用でき、これから紹介する `Codable` プロトコルと Smart KeyPaths もそれに含まれます。Swift 4 の構文に移行せずとも導入できるこれらの機能をぜひ活用していきましょう。

4.2 `Codable` プロトコル

Swift 4 で Foundation フレームワーク向けの新機能として導入されたものの 1 つが、`Codable` プロトコルと、それに付随する各種のエンコーダ／デコーダです。

これまで Swift でデータをシリализ、デシリализしようとする場合、以下の選択肢がありました。

- `NSCoding`
- `JSONSerialization`
- `PropertyListSerialization`

`NSCoding` は Objective-C 由来のプロトコルであるため、`struct` や `enum` といった値型では使用

できません。また残りの2つのクラスでも、`Any`型を配列や辞書にキャストしながら文字列のキーを使ってデータを保存・取得する必要があります。そのため、型安全性を重視したSwiftらしいAPIといえるものではありません。

こうした問題をSwiftネイティブなAPIで解消することを目指して導入されたのが、`Codable`プロトコルと各種のエンコーダ／デコーダです。

これらは『SE-0166: Swift Archival & Serialization』^{*1}と『SE-0167: Swift Encoders』^{*2}の2つのproposalとしてswift-evolutionで提案、採択されました。

`Codable`プロトコルは、シリализацию目的とした`Encodable`プロトコルと、デシリализацию目的とした`Decodable`プロトコルの組み合わせからできています。次のように定義されています。

```
public typealias Codable = Decodable & Encodable
```

`Encodable`をエンコード（シリализую）する役割は`Encoder`プロトコル、`Decodable`をデコード（デシリализую）する役割は`Decoder`プロトコルが担当します。

エンコーダ／デコーダの標準ライブラリの実装として、`JSONSerialization`に対する`JSONEncoder`と`JSONDecoder`、`PropertyListSerialization`に対する`PropertyListEncoder`と`PropertyListDecoder`が用意されています。

次項では`Codable`の基本的な使い方を見ていきましょう。

4.2.1 `Codable`プロトコルの基本的な使い方

`Codable`プロトコルを使い始めるのは非常に簡単です。もっとも単純な例では、次のように`Codable`に準拠するだけです。たったこれだけで、コンパイラが必要な実装を自動生成してくれます。

```
struct User: Codable {
    let id: String
    let name: String
    let age: Int
}
```

この`struct`をJSONからデコード、JSONにエンコードするのが次の例です。

```
let json = """
{
    "id": "123",
```

^{*1} <https://github.com/apple/swift-evolution/blob/master/proposals/0166-swift-archival-serialization.md>

^{*2} <https://github.com/apple/swift-evolution/blob/master/proposals/0167-swift-encoders.md>

```
"name": "Foo Bar",
"age": 24
}
""".data(using: .utf8)!

let decoder = JSONDecoder()
let decodedUser = try decoder.decode(User.self, from: json)

let encoder = JSONEncoder()
let encodedUserData: Data = try encoder.encode(decodedUser)
```

また `Codable` に準拠する型同士をネストさせることも可能です。

```
struct Foo: Codable {
    let name: String
}

struct Bar: Codable {
    let name: String
}

struct FooBar: Codable {
    let foo: Foo
    let bar: Bar
}

let foo = Foo(name: "foo")
let bar = Bar(name: "bar")
let fooBar = FooBar(foo: foo, bar: bar)

let json = try JSONEncoder().encode(fooBar)
let string = String(data: json, encoding: .utf8)!
// {"foo":{"name":"foo"}, "bar":{"name":"bar"}}
```

以下の組み込みデータ型が `Codable` に準拠していて、`Codable` 準拠の型のプロパティとして使用できます。

- 標準ライブラリ
 - Bool, Int, Int8~Int64, UInt, UInt8~UInt64, Float, Double, String, RawRepresentable, Optional, Array, Set, Dictionary
- Foundation
 - CGAffineTransform, Calendar, CharacterSet, Data, Date, DateComponents, DateInterval, Decimal, IndexPath, IndexSet, Locale, Measurement, NSRange, PersonNameComponents, TimeZone, URL, UUID
- CoreGraphics
 - CGAffineTransform, CGFloat, CGPoint, CGRect, CGSize, CGVector

`Optional` や `Array` などのジェネリック型が `Codable` に準拠すると記したことについて補足すると、ジェネリックパラメータの型（要素型）が `Codable` に準拠していない場合、包含する型は本来 `Codable` として振る舞うことはできないはずです。これは Swift 4.0 時点でのジェネリクス機能の制限によるもので、将来的に `Conditonal conformances`^{*3} と呼ばれる機能が実装されると、要素型が `Codable` である時だけ `Array` も `Codable` である、などと定義できます。このように、現時点では要素型が `Codable` 非準拠である場合はランタイムクラッシュが発生するため、若干の注意が必要です。

4.2.2 コンパイラによる実装の自動生成

先の例では、`Codable` に準拠した型のプロパティの名前と、JSON のキーの名前が一致しているため、全てを自動生成に任せることができました。

先ほどからコンパイラによる実装の自動生成の話をしていますが、これは一体どのようなものでしょうか？ 実際には以下のような実装が生成されています。

```
struct User: Codable {
    let id: String
    let name: String
    let age: Int

    private enum CodingKeys: String, CodingKey {
        case id
        case name
        case age
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        id = try container.decode(String.self, forKey: .id)
        name = try container.decode(String.self, forKey: .name)
        age = try container.decode(Int.self, forKey: .age)
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(id, forKey: .id)
        try container.encode(name, forKey: .name)
        try container.encode(age, forKey: .age)
    }
}
```

`Codable` プロトコルへの準拠による実装の自動生成には、大きく以下の3つがあります。

1. `enum CodingKeys (CodingKey プロトコル)`
2. `init(from decoder: Decoder) throws (Decodable プロトコル)`
3. `func encode(to encoder: Encoder) throws (Encodable プロトコル)`

^{*3} <https://github.com/apple/swift-evolution/blob/master/proposals/0143-conditional-conformances.md>

単純なケースでは、これらの自動生成を活用することで、実装やメンテナンスの手間を大きく省略できます。一方で、これらの要素を自分で実装することで、`Codable` の挙動をカスタマイズできます。

Swift 4.0 時点では、`struct NewType: Codable` のように、型の宣言時に `Codable` 準拠を記述した時しか自動生成は行われません。`extension NewType: Codable {}` では自動生成は行われないため、注意してください。もちろん実装をカスタマイズする場合は extension による宣言と実装でも問題ありません。

4.2.3 CodingKeys

まずは `CodingKey` プロトコルに準拠した、`CodingKeys` というネストした enum について見ていくましょう。この enum によってプロパティ名とキー名を静的に関連付ける仕組みとすることで、デコードとエンコードを型安全に行えるようになっています。

自動生成時には、対象になる型の全てのストアドプロパティ（`lazy` などを除く）が列挙の対象になります。この時、プロパティ名がそのまま case 名、つまりキー名として使用されます。

ストアドプロパティのうち 1 つでも `Codable` に準拠しない型がある、もしくは要素型が `Codable` に準拠しないジェネリック型がある場合、`CodingKeys` の自動生成と `Codable` への自動準拠に失敗し、コンパイルエラーになります。こうしたケースでは、`CodingKeys` という名前のネストした enum を明示的に定義し、以下のルールにしたがうことで、`Decodable` と `Encodable` の実装は自動生成に任せたままにできます。

- `Codable` に準拠するプロパティの名前を case として列挙する
- `Codable` に準拠しないプロパティにはデフォルト値を用意する

```
struct A {
    let int: Int
}

struct B: Codable {
    let string: String
    let a: A = A(int: 0) // Codable に準拠していないのでデフォルト値を用意する

    // a は列挙から除外する
    enum CodingKeys: String, CodingKey {
        case string
    }

    // init(from decoder: Decoder) throws と
    // func encode(to encoder: Encoder) throws は自動生成される
}
```

`CodingKeys` の手動定義はプロパティ名とキー名が一致しない場合にも利用できます。Swift のプロパティ名は `lowerCamelCase`、JSON のキー名は `snake_case`、というケースで有用です。

```
// sneak_case の JSON
let json = """
{
    "full_name": "Sho Ikeda"
}
"""

struct Author: Codable {
    let fullName: String

    enum CodingKeys: String, CodingKey {
        case fullName = "full_name"
    }
}
```

ここでは `CodingKeys` という enum の名前も重要です。というのも、`Decodable.init(from:)` と、`Encodable.encode(to:)` の自動生成では、`CodingKey` 準拠の enum が `CodingKeys` という名前で自動生成されている前提で実装されているため、名前を揃えておく必要があります。逆に言うと、それらの実装をカスタマイズする場合は `CodingKey` 準拠の型を扱う箇所も自分で実装することになるため、`AuthorCodingKeys` のような自由な名前で構いません。

また `CodingKey` プロトコルの定義は以下のようになっています。これらの実装もコンパイラにより自動生成されますが、その挙動をカスタマイズするケースはそれほど多くはないでしょうから、あまり気にする必要はないでしょう。

```
public protocol CodingKey {
    var stringValue: String { get }
    init?(stringValue: String)

    var intValue: Int? { get }
    init?(intValue: Int)
}
```

4.2.4 Decodable と Decoder

続いて `Decodable` プロトコルと `Decoder` プロトコルを見ていきましょう。`Decodable` の定義は次のようにになっています。

```
public protocol Decodable {
    init(from decoder: Decoder) throws
}
```

`Decoder` プロトコルに準拠した `decoder` を引数に取るイニシャライザが定義されています。

`throws`により、デコード中に発生したエラーを `throw`できます。標準ライブラリの実装では `DecodingError`というenumが使用され、失敗した理由やどの箇所でデコードが失敗したのかを知ることができます。

```
public enum DecodingError : Error {
    public struct Context {
        public let codingPath: [CodingKey]
        public let debugDescription: String
        public let underlyingError: Error?

        public init(codingPath: [CodingKey],
                   debugDescription: String, underlyingError: Error? = nil) {
            self.codingPath = codingPath
            self.debugDescription = debugDescription
            self.underlyingError = underlyingError
        }
    }

    case typeMismatch(Any.Type, Context)
    case valueNotFound(Any.Type, Context)
    case keyNotFound(CodingKey, Context)
    case dataCorrupted(Context)
}
```

`Decodable`が受け取る `Decoder`の定義は次のようになっています。

```
public protocol Decoder {
    var codingPath: [CodingKey] { get }
    var userInfo: [CodingUserInfoKey : Any] { get }

    func container<Key>(keyedBy type: Key.Type) throws -> KeyedDecodingContainer<Key>
    func unkeyedContainer() throws -> UnkeyedDecodingContainer
    func singleValueContainer() throws -> SingleValueDecodingContainer
}
```

Decodable の実装

次に、これらを利用した `Decodable`の典型的な実装を確認してみます。

```
let inputJSON = """
{
    "string": "String",
    "array": [1,2,3],
    "dictionary": ["A": "AAA", "B": "BBB"],
    "date": "1503325273.0830579",
    "maybeNil1": null,
```

```
"point": [10, 20],
"id": "peaks-cc"
}
"""

struct PleaseDecode: Decodable {
    let string: String
    let array: [Int]
    let dictionary: [String: Double]
    let date: Date
    let maybeNil1: Int?
    let maybeNil2: Bool?

    let point: Point
    let id: ID

    enum CodingKeys: String, CodingKey {
        case string
        case array
        case dictionary
        case date
        case maybeNil1
        case maybeNil2

        case point
        case id
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)

        // プロパティが必須であれば‘decode’
        string = try container.decode(String.self, forKey: .string)
        array = try container.decode([Int].self, forKey: .array)
        dictionary = try container.decode([String: Double].self, forKey: .dictionary)
        date = try container.decode(Date.self, forKey: .date)
        // プロパティが必須でなければ‘decodeIfPresent’
        maybeNil1 = try container.decodeIfPresent(Int.self, forKey: .maybeNil1)
        maybeNil2 = try container.decodeIfPresent(Bool.self, forKey: .maybeNil2)

        point = try container.decode(Point.self, forKey: .point)
        id = try container.decode(ID.self, forKey: .id)
    }
}

struct Point: Decodable {
    let x: Int
    let y: Int

    init(from decoder: Decoder) throws {
        let container = try decoder.unkeyedContainer()
        x = container.decode(Int.self)
        y = container.decode(Int.self)
    }
}

struct ID: Decodable {
```

```
let value: String

init(from decoder: Decoder) throws {
    let container = try decoder.singleValueContainer()
    x = container.decode(String.self)
}
```

`Decodable.init(from:)` を自分で実装すると `CodingKeys` の自動生成が行われなくなるため、`CodingKeys` も自分で定義する必要があることに注意してください。

デコード処理には、まず `Decoder` が持つ以下のメソッドから各種のコンテナを取り出し、実際のデコード処理はそのコンテナが行います。

- `func container<Key>(keyedBy type: Key.Type) throws -> KeyedDecodingContainer<Key>`
- `func unkeyedContainer() throws -> UnkeyedDecodingContainer`
- `func singleValueContainer() throws -> SingleValueDecodingContainer`

`KeyedDecodingContainer` は JSON でいうオブジェクト（Key-Value のストレージ）に相当し、キーに対応する値を取り出せます。`CodingKey` が使われていることから、コンテナ毎に特定のキーからしか値を取り出すことができないようになっています。

`UnkeyedDecodingContainer` は JSON でいう配列に相当し、キーではなく要素の順序で値を取り出します。上記の例では `[10, 20]` といった配列から `x = 10, y = 20` のように順序に対応して値を取り出しています。

`SingleValueDecodingContainer` は JSON でいうプリミティブに相当し、単一の値からの変換を行います。上記の例では "peaks-cc" という文字列から ID 型のインスタンスを生成しています。

これらのコンテナに対して以下の各種メソッドを使用してデコードしていきます。

- `KeyedDecodingContainer`
 - `decode(_:_forKey:)`、`decodeIfPresent(_:_forKey:)`、`decodeNil(forKey:)`
- `UnkeyedDecodingContainer`
 - `decode(_:_)`、`decodeIfPresent(_:_)`、`decodeNil()`
- `SingleValueDecodingContainer`
 - `decode(_:_)`、`decodeNil()`

デコード中のバリデーション

`Decodable` の実装をカスタマイズすることで、独自のバリデーション処理を組み込むことも可能です。次の例では月を表現する数字を 1~12 の範囲に制限しています。

```
struct MonthHolder: Decodable {
    let month: Int // 1~12 のいずれか
```

```
enum CodingKeys: String, CodingKey {
    case month
}

init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy: CodingKeys.self)
    let month = try container.decode(Int.self, forKey: .month)
    guard (1...12).contains(month) else {
        throw DecodingError.dataCorruptedError(
            forKey: CodingKeys.month,
            in: container,
            debugDescription: "month value must be from 1 to 12"
        )
    }
    self.month = month
}
```

ここでは月の値を取得した後に、その値が1～12の範囲に収まっているかを確認し、収まっていない場合にはエラーをthrowしています。このようなバリデーションエラーにはDecodingErrorのdataCorruptedErrorメソッドを使用するとよいでしょう（戻り値はDecodingErrorのcaseの.dataCorruptedです）。

複数のキーから1つのプロパティを導出する

複雑なデコード処理の例として、複数のキーから1つのプロパティの値を導出するケースを考えてみましょう。ここでは2種類の方法を紹介します。

```
let json = """
{
    "location": "Kyoto",
    "first_name": "Sho",
    "family_name": "Ikeda"
}
"""

// パターン1
struct Author_1: Decodable {
    let location: String
    let fullName: String

    enum CodingKeys: String, CodingKey {
        case location
        case firstName = "first_name"
        case familyName = "family_name"
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.location = try container.decode(String.self, forKey: .location)
```

```
        let firstName = try container.decode(String.self, forKey: .firstName)
        let familyName = try container.decode(String.self, forKey: .familyName)
        self.fullName = "\(firstName) \(familyName)"
    }
}

// パターン 2
struct Author_2: Decodable {
    let location: String
    let fullName: String

    enum CodingKeys: String, CodingKey {
        case location
    }

    enum NameCodingKeys: String, CodingKey {
        case firstName = "first_name"
        case familyName = "family_name"
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.location = try container.decode(String.self, forKey: .location)

        let nameContainer = try decoder.container(keyedBy: NameCodingKeys.self)
        let firstName = try nameContainer.decode(String.self, forKey: .firstName)
        let familyName = try nameContainer.decode(String.self, forKey: .familyName)
        self.fullName = "\(firstName) \(familyName)"
    }
}
```

パターン1では、1つの CodingKeys という enum に全てのキーを列挙しています。case の数とストアドプロパティの数が異なるのが気になるかもしれません、問題ありません。先述したように、Decodable.init(from:) を自分で実装する場合は CodingKeys が自動生成されません。したがって、CodingKeys の自動生成時に行われる、case とプロパティが1対1で対応しているかどうかのチェック自体がないため、正常にコンパイルできます。

パターン2では、CodingKey の enum を複数に分割し、1つの decoder から異なるキー群の複数の KeyedDecodingContainer を得ることで、コンテナ毎に値を取得や合成しています。1つのオブジェクトの中で多くのキーがフラットに並んでいる場合は、このように CodingKey の enum を一定のグループ毎に分割すると見通しをよくできるかもしれません。

ネストした要素をフラットにする

先程はフラットに並んだ複数のキーから1つのプロパティを導出するパターンを見ましたが、逆にネストしたJSONのオブジェクトをフラットにするケースはどうなるでしょうか？

```
let json = """
{
```

```
"id": "123",
"meta": {
    "foo": "Foo",
    "bar": "Bar"
}
"""

struct Flattening: Decodable {
    let id: String
    let foo: String
    let bar: String

    enum CodingKeys: String, CodingKey {
        case id
        case meta
    }

    enum MetaCodingKeys: String, CodingKey {
        case foo
        case bar
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.id = try container.decode(String.self, forKey: .id)

        let metaContainer = try container.nestedContainer(
            forKeyedSubitem: MetaCodingKeys.self, forKey: .meta)
        self.foo = try metaContainer.decode(String.self, forKey: .foo)
        self.bar = try metaContainer.decode(String.self, forKey: .bar)
    }
}
```

このように `nestedContainer(forKeyedSubitem:forKey:)` メソッドを使用すると、あるキーに対するオブジェクトを `KeyedDecodingContainer` として得ることができます。あとはそのコンテナから値を取得すればよいでしょう。

こうしたネストした要素を対象とするコンテナを取得するメソッドには他に以下のものがあるので、場面によって使い分けましょう。

- `KeyedDecodingContainer.nestedUnkeyedContainer(forKey:)`
- `UnkeyedDecodingContainer.nestedContainer(forKeyedSubitem:)`
- `UnkeyedDecodingContainer.nestedUnkeyedContainer()`

codingPath

`Decoder` と `DecodingError.Context` が持つ `codingPath` についても説明しておきましょう。これはデコーディング中のデータ階層において、どこを処理しているかを示します。

```
let json = """
{
    "root": {
        "parent": {
            "child": {
                "name": "Child",
                "age": "12"
            }
        },
        "foo": "Foo",
        "bar": "Bar"
    }
}"""
""""
```

上記のような JSON をデコード中に "age" でエラーが発生した場合、その時の `codingPath` は、[`Root.CodingKeys.parent`, `Parent.CodingKeys.child`, `Child.CodingKeys.age`] のようになるでしょう。

`codingPath` は、後述する `Encoder` と `EncodingError` にもほぼ同様の定義で登場して使われています。

4.2.5 JSONDecoder

標準ライブラリが提供するデコーダの実装として、`JSONDecoder` と `PropertyListDecoder` の 2 つが用意されています。本項では `JSONDecoder` の使い方を紹介していきます。

デコーダと言っても、`JSONDecoder` 自体は `Decoder` プロトコルには準拠せず、デコード用には `open func decode<T : Decodable>(_ type: T.Type, from data: Data) throws -> T` という 1 つのメソッドだけが用意されています。こうすることで、`Decoder` に関する実装詳細をカプセル化し、`Decodable` をデコードする利用者には隠蔽されています。`Decoder` プロトコルの利用方法を意識するのは `Decodable` の実装者だけでよくなるというわけです。

`JSONDecoder.decode(_:from:)` には、デコードする型のメタタイプと、JSON 文字列のバイナリデータを渡します。これまでのサンプルコードでも使用していましたが、次のような使い方です。

```
let json = """
{
    "id": "123",
    "name": "Foo Bar",
    "age": 24
}"""
"""".data(using: .utf8)

let decoder = JSONDecoder()
let decodedUser = try decoder.decode(User.self, from: json)
```

このメソッド内では `JSONSerialization.jsonObject(with:options)` を使い、引数の

data を JSON の中間表現として Any 型の値に変換しています。この時の options は空で、`JSONSerialization.ReadingOptions.allowFragments` が指定されていません。そのため JSON のトップレベルはオブジェクトか配列である必要があり、トップレベルに文字列や数値、真偽値を使用できないことに注意してください。

```
// OK
let object = """
{
    "a": "A",
    "b": 10,
    "c": true
}
"""

// OK
let array = """
[
    10, 20, 30
]
"""

// NG
let string = """
    "top level string"
"""
```

また、`JSONDecoder` にはいくつかの `DecodingStrategy` と呼ばれる設定が存在します。例えば日付を JSON 上で表現するには、エポック秒として数字を使用する、ISO 8601 形式の文字列を使用する、などいくつかのパターンが考えられます。こうした特定の型をデコードする際にどのパターンを使用するかを設定するのが `DecodingStrategy` です。`JSONDecoder` には 3 種類の `DecodingStrategy` が用意されています。

- `DateDecodingStrategy`
- `DataDecodingStrategy`
- `NonConformingFloatDecodingStrategy`

DateDecodingStrategy

`Date` 型のデコードに使用される `DateDecodingStrategy` の定義は次のとおりです。

```
public enum DateDecodingStrategy {
    case deferredToDate
    case secondsSince1970
    case millisecondsSince1970
    @available(OSX 10.12, iOS 10.0, watchOS 3.0, tvOS 10.0, *)
    case iso8601
    case formatted(DateFormatter)
    case custom((_ decoder: Decoder) throws -> Date)
```

```
}
```

エポック秒や ISO 8601 形式の文字列、 `DateFormatter` を使用したカスタムフォーマットの文字列、またクロージャを使用した任意のデコード処理が使用できます。この設定は `dataDecodingStrategy` プロパティで変更でき、デフォルト値は `.deferredToDate` です。 `.deferredToDate` では `Date.init(from decoder: Decoder)` の実装にデコード処理が委譲されます。この場合には `init(timeIntervalSinceReferenceDate:)` のイニシャライザが使用されます。

`case iso8601` を使ったデコード処理には Foundation の `ISO8601DateFormatter` が使用されるため、使用できる OS バージョンには限りがあります。

DataDecodingStrategy

`Data` 型のデコードに使用される `DataDecodingStrategy` の定義は次のとおりです。

```
public enum DataDecodingStrategy {  
    case deferredToDate  
    case base64  
    case custom((_ decoder: Decoder) throws -> Data)  
}
```

Base64 でエンコードされた文字列やクロージャを使用した任意のデコード処理が使用できます。この設定は `dataDecodingStrategy` プロパティで変更でき、デフォルト値は `.base64` です。

NonConformingFloatDecodingStrategy

IEEE 754^{*4}で規定される正負の無限大と NaN の値は JSON の数値型では表現できず、文字列で表現する必要があります。それらの値をそれぞれ特定の文字列表現から変換可能にするのが `NonConformingFloatDecodingStrategy` です。定義は次のとおりです。

```
public enum NonConformingFloatDecodingStrategy {  
    case 'throw'  
    case convertFromString(positiveInfinity: String,  
                           negativeInfinity: String, nan: String)  
}
```

`.convertFromString` を使用すると、文字列表現から `Double.infinity` 、 `-Double.infinity` 、 `Double.nan` をデコード結果として返せます(`Float` も同様です)。この設定は `nonConformingFloatDecodingStrategy` プロパティで変更できます。デフォルト値は `.throw` です。

^{*4} <http://ieeexplore.ieee.org/document/5976968/>

4.2.6 Encodable と Encoder

`Encodable` プロトコルと `Encoder` プロトコルも、`Decodable` と `Decoder` に非常に似た構造、関係を持っています。`Encodable` の定義は次のようにになっています。

```
public protocol Encodable {
    func encode(to encoder: Encoder) throws
}
```

`Encoder` プロトコルに準拠した `encoder` を引数に取る `encode(to:)` メソッドが定義されています。`throws` により、エンコード中に発生したエラーを `throw` できます。標準ライブラリの実装では `EncodingError` という `enum` が使用され、失敗した理由やどの箇所でエンコードが失敗したのかを知ることができます。

```
public enum EncodingError : Error {
    public struct Context {
        public let codingPath: [CodingKey]
        public let debugDescription: String
        public let underlyingError: Error?

        public init(codingPath: [CodingKey],
                   debugDescription: String, underlyingError: Error? = nil) {
            self.codingPath = codingPath
            self.debugDescription = debugDescription
            self.underlyingError = underlyingError
        }
    }

    case invalidValue(Any, Context)
}
```

次に `Encodable` が受け取る `Encoder` の定義は次のようにになっています。

```
public protocol Encoder {
    var codingPath: [CodingKey] { get }
    var userInfo: [CodingUserInfoKey : Any] { get }

    func container<Key>(keyedBy type: Key.Type) -> KeyedEncodingContainer<Key>
    func unkeyedContainer() -> UnkeyedEncodingContainer
    func singleValueContainer() -> SingleValueEncodingContainer
}
```

Encodable の実装

続けてこれらを利用した `Encodable` の典型的な実装を確認してみます。

```
let outputJSON = """
{
    "string": "String",
    "array": [1,2,3],
    "dictionary": ["A": "AAA", "B": "BBB"],
    "date": "1503325273.0830579",
    "point": [10, 20],
    "id": "peaks-cc"
}
"""

struct PleaseEncode: Encodable {
    let string: String
    let array: [Int]
    let dictionary: [String: Double]
    let date: Date
    let maybeNil1: Int?
    let maybeNil2: Bool?

    let point: Point
    let id: ID

    enum CodingKeys: String, CodingKey {
        case string
        case array
        case dictionary
        case date
        case maybeNil1
        case maybeNil2

        case point
        case id
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)

        // プロパティが必須であれば‘encode’
        try container.encode(string, forKey: .string)
        try container.encode(array, forKey: .array)
        try container.encode(dictionary, forKey: .dictionary)
        try container.encode(date, forKey: .date)
        // プロパティが必須でなければ‘encodeIfPresent’
        try container.encodeIfPresent(maybeNil1, forKey: .maybeNil1)
        try container.encodeIfPresent(maybeNil2, forKey: .maybeNil2)

        try container.encode(point, forKey: .point)
        try container.encode(id, forKey: .id)
    }
}
```

```
struct Point: Encodable {
    let x: Int
    let y: Int

    func encode(to encoder: Encoder) throws {
        var container = encoder.unkeyedContainer()
        try container.encode(x)
        try container.encode(y)
    }
}

struct ID: Encodable {
    let value: String

    func encode(to encoder: Encoder) throws {
        var container = encoder.singleValueContainer()
        try container.encode(value)
    }
}
```

`Decodable` と同様に、`Encodable.encode(to:)` を自分で実装すると `CodingKeys` の自動生成が行われなくなります。`CodingKeys` も自分で定義する必要があることに注意してください。

エンコード処理にはまず `Encoder` が持つ以下のメソッドから各種のコンテナを取り出し、実際のエンコード処理はそのコンテナが行います。

- `func container<Key>(keyedBy type: Key.Type) -> KeyedEncodingContainer<Key>`
- `func unkeyedContainer() -> UnkeyedEncodingContainer`
- `func singleValueContainer() -> SingleValueEncodingContainer`

これらは `Decoder` の各種コンテナと非常に似通っていることが分かると思います。各コンテナの `encode` メソッドは `mutating` なメソッドになっていて、`Encoder` から取得するコンテナは `var container = encoder.container(keyedBy: CodingKeys.self)` のように、定数ではなく変数を入れる必要があります。

これらのコンテナに対して以下の各種メソッドを使用してエンコードしていきます。

- `KeyedEncodingContainer`
 - `encode(_:_forKey:)`、`encodeIfPresent(_:_forKey:)`、`encodeNil(forKey:)`
- `UnkeyedEncodingContainer`
 - `encode(_:_)`、`encode(contentsOf:)`、`encodeNil()`
- `SingleValueEncodingContainer`
 - `encode(_:_)`、`encodeNil()`

4.2.7 JSONEncoder

標準ライブラリが提供するエンコーダの実装として、`JSONEncoder` と `PropertyListEncoder` の2つが用意されています。本項では `JSONEncoder` の使い方を紹介していきます。

`JSONDecoder` と同様に、`JSONEncoder` 自体は `Encoder` プロトコルには準拠しておらず、エンコード用には `open func encode<T : Encodable>(_ value: T) throws -> Data` というメソッドが 1 つだけ用意されています。

`JSONEncoder.encode(_:_)` には、エンコードする `Encodable` 準拠の型の値を渡します。これまでのサンプルコードでも使用していましたが、次のような使い方をします。

```
let user = User(id: "123", name: "Foo Bar", age: 24)
let encoder = JSONEncoder()
let encodedUserData: Data = try encoder.encode(user)
```

このメソッド内では最終的に `JSONSerialization.data(withJSONObject:options)` を使用して JSON 文字列のバイナリデータを返します。この時、JSON のトップレベルはオブジェクトか配列である必要があるのも `JSONDecoder` と同様です。

`JSONSerialization.WritingOptions` に対応する `JSONEncoder.OutputFormatting` という `OptionSet` を使用することで、JSON 出力時のフォーマットも設定できます。用意されているのは `prettyPrinted` と `sortedKeys` の 2 つです。この設定は `outputFormatting` プロパティで変数でき、デフォルト値は空です。

また `JSONEncoder` には以下の `EncodingStrategy` と呼ばれる設定が存在します。それぞれ同種の `JSONDecoder` の `DecodingStrategy` に対応します。

- `DateEncodingStrategy`
- `DataEncodingStrategy`
- `NonConformingFloatEncodingStrategy`

4.2.8 繙承関係の表現 (`superDecoder()` と `superEncoder()`)

これまで全て `struct` を使った例を見てきましたが、本項では `class` と継承を `Codable` ではどのように扱うかを説明します。

```
class Super: Codable {
    var int: Int?
}

class Child: Super {
    var double: Double?
}

let child = Child()
child.int = 1
child.double = 10

let encoded = try! String(data: JSONEncoder().encode(child), encoding: .utf8)!
print(encoded)
```

上記のコードの出力結果はどうなるでしょうか？ 結果は{"int":1}となり、スーパークラスのデータしか含まれていないことが分かります。このようにスーパークラスで `Codable` に準拠し、サブクラスでプロパティを追加している場合では、`Codable` の自動生成はサブクラス毎に新たな実装を生成してくれるわけではないようです。そのため `Codable` で `class` と継承を扱う場合はスーパークラス、サブクラスともに自分で実装を提供する必要があります。実装例は以下のとおりです。

```
class Super: Codable {
    var int: Int?

    init() {}

    enum CodingKeys: String, CodingKey {
        case int
    }

    required init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.int = try container.decode(Int.self, forKey: .int)
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encodeIfPresent(int, forKey: .int)
    }
}

class Child: Super, CustomStringConvertible {
    var double: Double?

    override init() { super.init() }

    enum ChildCodingKeys: String, CodingKey {
        case double
    }

    required init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: ChildCodingKeys.self)
        self.double = try container.decode(Double.self, forKey: .double)
        try super.init(from: container.superDecoder())
    }

    override func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: ChildCodingKeys.self)
        try container.encodeIfPresent(double, forKey: .double)
        try super.encode(to: container.superEncoder())
    }

    var description: String {
        return "int: \(String(describing:int)), double: \(String(describing:double))"
    }
}
```

```
let child = Child()
child.int = 1
child.double = 10

let encoded = try! String(data: JSONEncoder().encode(child), encoding: .utf8)!
print(encoded) // {"super":{"int":1}, "double":10}

let decoded = try! JSONDecoder().decode(Child.self, from: encoded.data(using: .utf8)!)
print(decoded) // int: Optional(1), double: Optional(10.0)
```

今度はスーパークラス、サブクラス両方のデータが正常にエンコード、デコードされています。

ところでデコードされた結果を見てみると、"super"というキーが気になります。これは `container.superEncoder()` および `container.superDecoder()` によって用意されたキーになっていて、`Codable` は"super"というキーを経由してスーパークラスのデータをエンコード／デコードすることを推奨しています。`superEncoder(forKey:)` および `superDecoder(forKey:)` によって対象となるキーを"super"以外にも変更できます。

また、次のようにスーパークラス用のコンテナを使わずに、サブクラスと同じコンテナも使用できます。しかし、これはスーパークラスの処理でサブクラスの処理結果も上書きできてしまう（またはその逆）ため、推奨されていません^{*5}。

If a shared container is desired, it is still possible to call `super.encode(to: encoder)` and `super.init(from: decoder)`, but we recommend the safer containerized option.

```
class Child: Super, CustomStringConvertible {
    ...

    required init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: ChildCodingKeys.self)
        self.double = try container.decode(Double.self, forKey: .double)
        try super.init(from: decoder)
    }

    override func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: ChildCodingKeys.self)
        try container.encodeIfPresent(double, forKey: .double)
        try super.encode(to: encoder)
    }

    ...
}
```

^{*5} <https://github.com/apple/swift-evolution/blob/master/proposals/0166-swift-archival-serialization.md>

4.2.9 自動生成による実装と手動実装の混在

これまで見てきたように、`Codable` の実装は自動生成に任せることも、手動で実装することもどちらも可能です。一方で、`Encodable` の実装は自動生成、`Decodable` の実装は手動、またはその逆というように、自動生成による実装と手動実装は混在できます。しかしこれには注意が必要です。自動生成による実装と手動実装の間に一貫性がないと、エンコードしたオブジェクトがデコードできなくなってしまう可能性があるからです。

```
struct MixedImplementation: Codable {
    let b: Bool
    let i: Int
    let s: String?

    enum CodingKeys: String, CodingKey {
        case b
        case i
        case s
    }

    // 'Decodable'は手動実装
    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)

        b = try container.decode(Bool.self, forKey: .b)
        i = try container.decode(Int.self, forKey: .i)

        let strings = try container.decodeIfPresent([String].self, forKey: .s)
        s = strings?.first
    }

    // 'Encodable'は自動生成
}

let json = """
{
    "b": true,
    "i": 5000,
    "s": [ "foo", "bar" ]
}"""
.data(using: .utf8)

let decoded = try JSONDecoder().decode(MixedImplementation.self, from: json)
let encoded = try JSONEncoder().encode(decoded)
do {
    let reEncoded = try JSONDecoder().decode(MixedImplementation.self,
                                              from: encoded) // ここで失敗
} catch {
    print(error)
}
```

このような事態を避けるためにも、原則的には混在をさせずに両方自動生成か、両方手動実装

のどちらかに寄せるべきでしょう。`Encodable` か `Decodable` のどちらかしか実装しない場合は気にする必要はありませんが、例えば外部モジュールの型で `Decodable` なものを、extension で `Encodable` にも対応させたい、というようなケースでは必然的に後者が手動実装になるため、十分に注意する必要があるでしょう。

4.3 Smart KeyPaths

Swift 4 での Foundation に関する機能強化のもう 1 つは、Smart KeyPaths と呼ばれる KVC (Key-Value Coding) / KVO (Key-Value Observing) のための機能です。

Swift ではこれまで、Swift 3 から導入された`#keyPath()` 構文^{*6}を使用することで、文字列リテラルを使用しない型安全な方法でプロパティへの一種の参照を得ることができました。しかしその戻り値は結局 `String` であり、下記のようにいくつかの欠点がありました。

- どの型のプロパティか、そのプロパティの戻り値は何か、という型情報が失われている
- `NSObject` を継承したクラスにしか使用できない
 - `value(forKey:)`
 - `value(forKeyPath:)`
 - `setValue(_:,forKey:)`
 - `setValue(_:,forKeyPath:)`
- macOS や iOS などの Darwin プラットフォーム以外で使用できない（Linux で使用できない）

```
class Person: NSObject {
    @objc var name: String = ""
}

let nameKeyPath: String = #keyPath(Person.name)
// これは存在しないプロパティなのでコンパイルエラー
// let fooKeyPath: String = #keyPath(Person.foo)

let person = Person()
person.name = "ikesyo"
let name: Any? = person.value(forKey: #keyPath(Person.name))
```

また Swift のメソッドでは `let f = foo.bar` のように、メソッドを実行することなく参照のみを取得し、後でそのメソッド参照を `let result = f()` と実行できます。一方プロパティには参照を取得して後で実行する機能はありませんでした。

Smart KeyPaths はこれらの欠点を解消するために SE-0161^{*7}での提案、採択を経て導入されました。具体的には、プロパティへの参照を表す `KeyPath` クラス群と、`KeyPath` を返す `KeyPath` 構文が追加されています。

^{*6} <https://github.com/apple/swift-evolution/blob/master/proposals/0062-objc-keypaths.md>

^{*7} <https://github.com/apple/swift-evolution/blob/master/proposals/0161-key-paths.md>

4.3.1 KeyPath 構文

KeyPath 構文は\<Type>.〈path〉という構造をしていて、〈Type〉は型名を、〈path〉はドット区切りの1つ以上のプロパティ名からなるチェーンを示しています。つまりネストした構造も一気に表現できるということです。^{*8} プロパティのチェーン途中に `Optional` が含まれる場合、通常のプロパティアクセス同様にオプショナルチェーンが利用できます。

KeyPath 構文でバックスラッシュによるエスケープが用いられているのは、`Foo.classVar` のように表現される、タイププロパティへのアクセスとの曖昧さを解消するためです。

```
// タイププロパティとの区別
struct Foo {
    static let defaultValue: Int = 645
    var value: Int
}

Foo.defaultValue // タイププロパティ
\Foo.value // KeyPath 構文

// KeyPath のチェーン
struct Foo {
    let bar: Bar?
}
struct Bar {
    let baz: Baz
}
struct Baz {
    let value: Int
}

\Foo.bar?.baz.value
```

4.3.2 KeyPath による subscript

KeyPath 構文によって得られる `KeyPath` を使ってプロパティの読み書きをする `subscript` が `Any` 型、つまり Swift の全ての型に追加されています。`subscript` の定義は以下のとおりです。^{*9}

```
extension Any {
    subscript(keyPath path: AnyKeyPath) -> Any? { get }
    subscript<Root: Self>(keyPath path: PartialKeyPath<Root>) -> Any { get }
    subscript<Root: Self, Value>(keyPath
```

^{*8} SE-0161 では`\Person.friends\[0].name`のような KeyPath 構文中での `subscript` も記載されていますが、Swift 4.0 時点では未実装です。

^{*9} SE-0161 での記載に合わせて `extension Any { ... }`としていますが、これは説明の便宜上の疑似コードです。実際には `Any` 型に `extension` は追加できません。

```
        path: KeyPath<Root, Value>) -> Value { get }
subscript<Root: Self, Value>(keyPath
    path: WritableKeyPath<Root, Value>) -> Value { set, get }
}
```

[keyPath: ...] という名前付き引数の subscript と、ジェネリクスの活用により、既存の subscript と衝突しないよう設計されています。この subscript を使った値の取得や更新は次のとおりです。

```
var foo = Foo(value: 1)
let value = foo[keyPath: \Foo.value] // 1
foo[keyPath: \Foo.value] = 645
let updatedValue = foo[keyPath: \Foo.value] // 645
```

subscript の引数に見られる AnyKeyPath や PartialKeyPath などの型について、詳しくは次項で説明します。

4.3.3 KeyPath のクラス階層

KeyPath 構文や subscript で利用される KeyPath は具体的には次の 5 つの型のクラス階層でできています。継承階層が下がるにつれて、より具体的な型情報を持ち、リードオンリーからリードライト可能な性質を持ちます。

- AnyKeyPath
- PartialKeyPath
- KeyPath
- WritableKeyPath
- ReferenceWritableKeyPath

AnyKeyPath

AnyKeyPath は、パスが始まるルートの型と、パスの終わりとなる値の型の両方とも持たない、型消去されたものです。型消去により、どの型のプロパティであるかやプロパティの戻り値型を無視して混在して扱えます。ただ、通常はこのクラスを使うことはあまりないでしょう。

```
struct Foo {
    let foo: Int
}
struct Bar {
    let bar: String
}
```

```
let fooOfFoo: AnyKeyPath = \Foo.foo
let barOfBar: AnyKeyPath = \Bar.bar

// 別の型の KeyPath と混ぜて扱えるが、
// 別の型のものであれば当然互換性はないので戻り値型は 'Any?' 

let foo = Foo(foo: 645)
let _: Any? = foo[keyPath: fooOfFoo] // 645
let _: Any? = foo[keyPath: barOfBar] // nil

let bar = Bar(bar: "bar")
let _: Any? = bar[keyPath: fooOfFoo] // nil
let _: Any? = bar[keyPath: barOfBar] // "bar"
```

PartialKeyPath

`PartialKeyPath<Root>` は 1 つの要素型を持つジェネリッククラスで、ルートの型を知っています。このクラスを使うと、ある型が持つ、異なる戻り値型の複数のプロパティをまとめて扱えます。一方で、ルートの型は区別されるため、別のルートの型の `PartialKeyPath` とは混在できません。

```
struct Holder {
    let bool: Bool
    let int: Int
    let double: Double
    let string: String
}

var allKeyPaths: [PartialKeyPath<Holder>] = [\Holder.bool,
                                             \Holder.int, \Holder.double, \Holder.string]
let holder = Holder(bool: true, int: 1, double: 1, string: "string")
for keyPath in allKeyPaths {
    // 自分の型のプロパティなので何かしらの値は取れるため、戻り値型は 'Any'
    let value: Any = holder[keyPath: keyPath]
    print(value)
    // true
    // 1
    // 1.0
    // string
}

struct Other {
    let something: String
}

// 異なるルートの型なのでコンパイルエラー
allKeyPaths.append(\Other.something)
```

KeyPath

`KeyPath<Root, Value>`は2つの要素型を持つジェネリッククラスで、ルートの型と、値の型の両方を知っています。これこそがSwiftらしい型安全なKVCを実現してくれるもので、もっとも基本的なKeyPathの型と言えるでしょう。ただしこまでの3つのクラスはリードオンリーで、値は変更できません。

```
struct Foo {
    let foo: Int
    let another: String
}
struct Bar {
    let bar: String
}

let fooOfFoo: KeyPath<Foo, Int> = \Foo.foo
let barOfBar: KeyPath<Bar, String> = \Bar.bar

let foo = Foo(foo: 645, another: "another")
let bar = Bar(bar: "bar")

// 値の型が分かっている
let int: Int = foo[keyPath: fooOfFoo] // 645
let string: String = bar[keyPath: barOfBar] // "bar"

// 値の型が異なればコンパイルエラー
let another: KeyPath<Root, String> = \Foo.another
let int: Int = foo[keyPath: \Foo.another]

// ルートの型が異なればコンパイルエラー
foo[keyPath: barOfBar]
bar[keyPath: fooOfFoo]
```

WritableKeyPath と ReferenceWritableKeyPath

`WritableKeyPath<Root, Value>`と`ReferenceWritableKeyPath<Root, Value>`は、`KeyPath`同様に2つの要素型を持つジェネリッククラスです。これら2つのKeyPathはリードライトの性質を持っていますが、前者は値型のセマンティクス、後者は参照型のセマンティクスを持つという違いがあります。これは参照型のプロパティを変更するには変数(`var`)でなく定数(`let`)でよく、値型のプロパティを変更するのに定数ではなく変数である必要があるのと同じ関係性です。

```
struct ValueMutation {
    var value: Int
}

// 値型のセマンティクス
let keyPathi: WritableKeyPath<ValueMutation, Int> = \ValueMutation.value
```

```
// 値型のプロパティは変数で変更できる
var v1 = ValueMutation(value: 0)
v1[keyPath: keyPath1] = 645

// 値型のプロパティは定数では変更できず、コンパイルエラー
let v2 = ValueMutation(value: 0)
v2[keyPath: keyPath1] = 645

class ReferenceMutation {
    var value: Int = 0
}

// 参照型のセマンティクス
let keyPath2: ReferenceWritableKeyPath<ReferenceMutation,
                                         Int> = \ReferenceMutation.value

// 参照型のプロパティは変数で変更できる
var r1 = ReferenceMutation()
r1[keyPath: keyPath2] = 645

// 参照型のプロパティは定数でも変更できる
let r2 = ReferenceMutation()
r2[keyPath: keyPath2] = 645

// ‘WritableKeyPath’だと定数では変更できなくなり、コンパイルエラー
let keyPath3: WritableKeyPath<ReferenceMutation, Int> = \ReferenceMutation.value
let r3 = ReferenceMutation()
r3[keyPath: keyPath3] = 645
```

4.3.4 Foundation の追加 API

iOS 11 では Foundation にも Smart KeyPaths を活用した API が追加されています。最もよく使われるには `NSObject.observe(_:options:changeHandler:)` でしょう。

これまで KVO を行う場合、`addObserver(_:forKeyPath:options:context:)` と `NSObject.observeValue(forKeyPath:of:change:context:)` を組み合わせて使っていましたが、変更があったプロパティのオブジェクトは `Any?`として渡され、自分で型チェックやキャストを行ったり、`context` を使用して適切に監視対象を区別したりする必要がありました。

一方 `NSObject.observe(_:options:changeHandler:)` では `KeyPath<Root, Value>`により、監視対象のオブジェクトと監視先のプロパティの型が静的に決定され、型安全に KVO を行えるようになりました。またクロージャベースの API となったことも使い勝手を向上させています。戻り値の `NSKeyValueObservation` クラスの `invalidate()` メソッドを呼ぶか、そのインスタンスが破棄されると監視を解除できます。

```
@objc class Observable: NSObject {
    @objc dynamic var name: String = "default"
```

```
}

let instance = Observable()

// これまでの API
var kvoContext: UInt8 = 0
@objc class Observer: NSObject {
    override func observeValue(forKeyPath keyPath: String?,
                                of object: Any?, change: [NSKeyValueChangeKey : Any]?,
                                context: UnsafeMutableRawPointer?) {
        if context == &kvoContext && keyPath == "name",
            let newValue = object as? String {
            print(newValue)
        } else {
            super.observeValue(forKeyPath: keyPath,
                               of: object, change: change, context: context)
        }
    }
}
let observer = Observer()
let keyPath1 = #keyPath(Observable.name)
instance.addObserver(observer, forKeyPath: keyPath1,
                     options: [.new], context: &kvoContext)
instance.name = "updated1"
instance.removeObserver(observer, forKeyPath: keyPath1)

// 新しい API
let keyPath2: KeyPath<Observable, String> = \Observable.name
let observation = instance.observe(\Observable.name, options: [.new]) {
    (receiver: Observable, change: NSKeyValueObservedChange<String>) in
    let newValue = receiver.name
    print(newValue)
}
instance.name = "updated2"
observation.invalidate()
```

その他に `NSEExpression` や `NSSortDescriptor` にも、`KeyPath<Root, Value>`を引数に取るイニシャライザのオーバーロードが追加されています。

```
extension NSEExpression {
    public convenience init<Root, Value>(forKeyPath keyPath: KeyPath<Root, Value>)
}

extension NSSortDescriptor {
    public convenience init<Root, Value>(keyPath: KeyPath<Root, Value>,
                                         ascending: Bool)
    public convenience init<Root, Value>(keyPath: KeyPath<Root, Value>,
                                         ascending: Bool,
                                         comparator cmptr: @escaping Foundation.Comparator)
    public var keyPath: AnyKeyPath?
}
```

4.4 参考文献

- What's New in Foundation
- Encoding and Decoding Custom Types
- Using JSON with Custom Types
- Swift Archival & Serialization
- Swift Encoders
- Swift 4 から実装される Codable
- Swift 4 Codable
- Codable の init(from:) をどう書くか
- Ultimate Guide to JSON Parsing With Swift 4
- Smart KeyPaths: Better Key-Value Coding for Swift
- KeyPath の便利な使い道を研究する

第5章

Xcode 9 の新機能

5.1 はじめに

2017年6月に開催されたWWDC 2017では、ドラッグアンドドロップのようなUIKitの新機能に加え、MusicKit、PDFKit、ARKitやCoreMLなど多くの新フレームワークが発表されました。^{*1} どれもアプリ自体の可能性や価値を高めることに直結する内容で、期間中に数多くのセッションが行われました。

一方で、WWDCは開発者環境やツールに関する発表が行われる場でもあります。ツール系での今年のトピックは、Xcodeのメジャーアップデートである「Xcode 9」が発表されたことでした。Xcodeのメジャーアップデートは毎年の恒例ではありますが、今回発表されたXcode 9には、ワイヤレスデバッグやSwift向けのリファクタリング機能など、アプリの開発速度を確実に改善してくれる嬉しい機能がたくさん盛り込まれました。筆者個人としては、ここ数年のアップデートの中でも非常に充実した内容だと感じています。

そこで本章では、Xcode 9の新機能や改善点について解説します。私たち開発者にとって、Xcodeをはじめとする開発ツール・APIは、新しいAPI以上に長い時間利用するものです。これらを十分に使いこなすことで、より効率的に開発を進められるようになることでしょう。

ぜひ本章を読んで、開発環境改善の参考にしてください。

5.1.1 本章の構成

Xcodeは長年利用されているツールであり、すでに多くの機能が存在しています。Xcode 9は新機能の追加だけでなく、これらの既存機能も多く改善されています。そこで本章では、新機能を「開発」「デバッグ」「テスト」の3つのフェーズに分類して、それぞれのフェーズで便利な新機能を紹介します。また、3つフェーズに加えて、Xcodeに統合されたXcodeサーバー機能についても解説します。

本章の構成は次のとおりです。

- 開発フェーズの新機能
 - Githubの統合とソースコード管理機能の改善
 - エディタ機能の改善

^{*1} <https://developer.apple.com/videos/play/wwdc2017/102/>

- iOS シミュレータの変更点
- デバッグフェーズの新機能
 - ワイヤレスデバッグ機能
 - ビューデバッグ機能の改善
- テストフェーズでの新機能
 - XCTest の新 API
 - XCUITest の新機能
 - コマンドラインのサポート強化
- Xcode サーバー
 - 機能の概要
 - アプリのインテグレーションテスト
 - アーカイブと配布

Xcode サーバーとは、2014 年の WWDC で発表された継続的インテグレーションのための機能です。^{*2} これまで macOS Server^{*3}に含まれていた機能ですが、Xcode 9 で Xcode アプリケーション本体へと統合されました。

5.2 開発フェーズの新機能

本節では、アプリの開発フェーズにおいて役立つ新機能を紹介します。ソースコードの取得やバージョン管理、コーディング、シミュレータでの動作確認などといった、開発中に必要な作業で多くの機能追加や改善が施されています。

5.2.1 Github の統合とソースコード管理機能の改善

Xcode は、古くから Git と Subversion の 2 つのバージョン管理システム (VCS) をサポートしています。^{*4} しかし、これまでコードのコミットやブランチ操作など一通りの作業はできたものの、機能の多くは「Source Control」メニュー や コンテキストメニューから利用できる程度で、あまり使い勝手の良いものではありませんでした。

Subversion については将来のバージョンでサポート停止することが公式に発表されています。

Xcode 9 ではこの使い勝手が大きく改善されました。VCS の UI がナビゲーター ペインの 1 つとして用意され、ブランチの状況やコミットログを見やすく表示します。また、ソースコード共有サービスである Github^{*5} とも連携するようになり、Xcode から Github のリポジトリを検索してクロールしたり、反対に Xcode から Github リポジトリを作成してコードを公開したりできます。iOS アプリ開発で利用するライブラリの多くは Github にホストされているため、これらの改善によって、いっそうライブラリを扱いやすくなりました。

^{*2} <https://developer.apple.com/videos/play/wwdc2014/415/>

^{*3} 旧称は「OS X Server」です。

^{*4}

^{*5} <https://github.com>

Github アカウントの設定とリポジトリのクローン

Xcode から Github を利用するには、まず Xcode に Github のアカウントを追加します。Xcode 9 の Preferences を開き、Accounts タブ左下にある「+」ボタンをクリックしてアカウントを追加します。github.com だけでなく Github Enterprise にも対応していて、その場合はサーバーの URL も入力します。

アカウントを追加できたら、認証方法も忘れないで設定しておきましょう。追加したアカウントを選び、クローンで利用するプロトコルと秘密鍵を選択します（図 5.1）。SSH 鍵の作成は Github のヘルプを参考にしてください。^{*6}

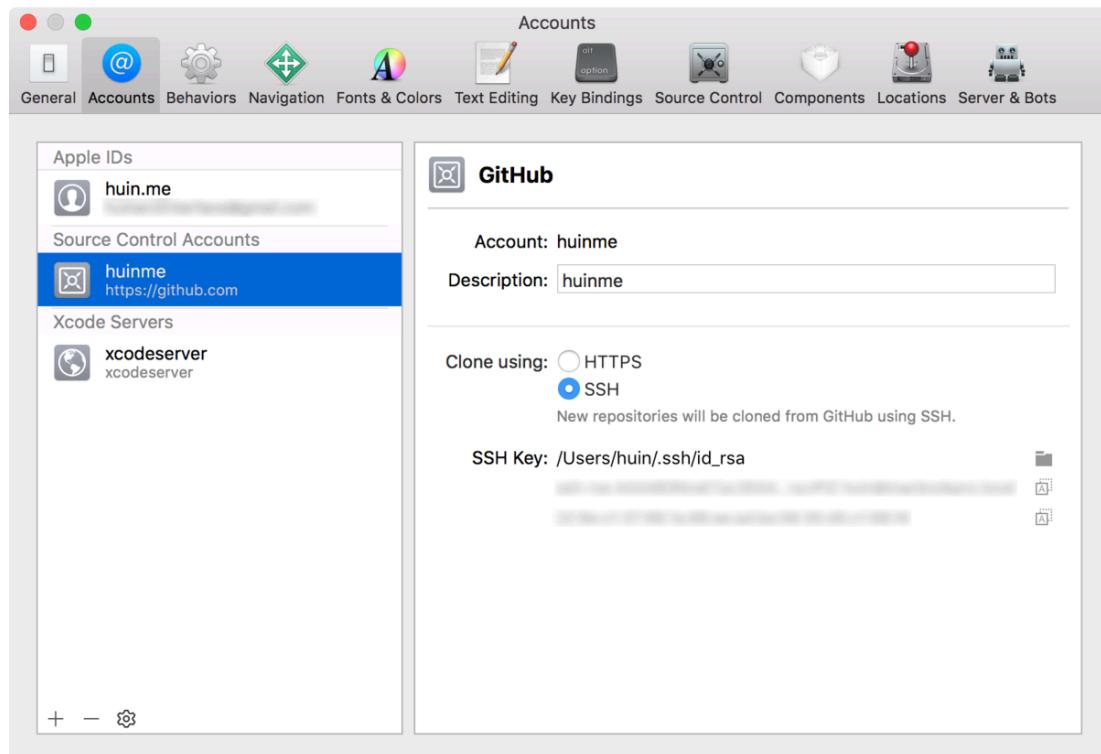


図 5.1: Github アカウントの追加と鍵の設定

アカウント設定が完了したら、Github からリポジトリをクローンしてみましょう。メニューから「Source Control」→「Clone」を選ぶと、リポジトリ選択画面が表示されます。初期状態では自分が関わっているリポジトリやスターをつけたリポジトリが表示されるので、そこから選択するか、キーワード検索をしてヒットしたリポジトリをクローンします（@{devgitclonerepositories}）。リポジトリの選択画面では、最終更新日時を確認したり、README も表示できたりするので、リポジトリを選択する際の参考にできます。

^{*6} <https://help.github.com/articles/connecting-to-github-with-ssh/>

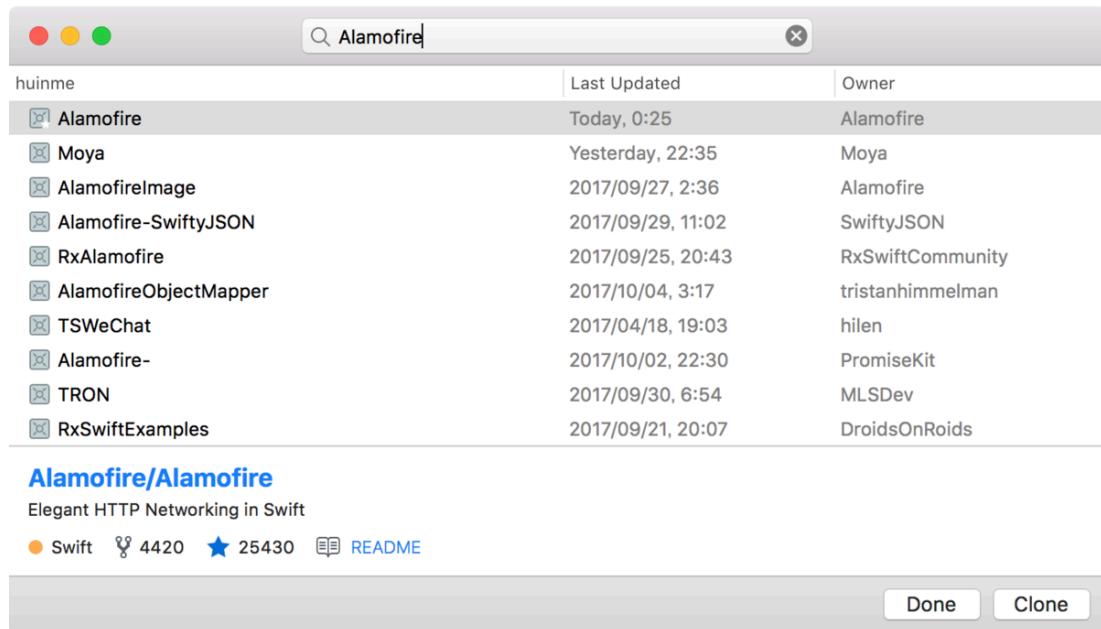


図 5.2: リポジトリの検索

Xcodeとの連携はGithubのサイトにも及んでいて、リポジトリのクローンをWebブラウザからも行えます。リポジトリのルートディレクトリにXcodeのプロジェクトファイルが置いてさえあれば、図5.3のようにGithubのページ上に「Open in Xcode」メニューが表示されるので、ここからクローンできます。ルートディレクトリにプロジェクトファイルがない場合、メニューは表示されないので注意してください。

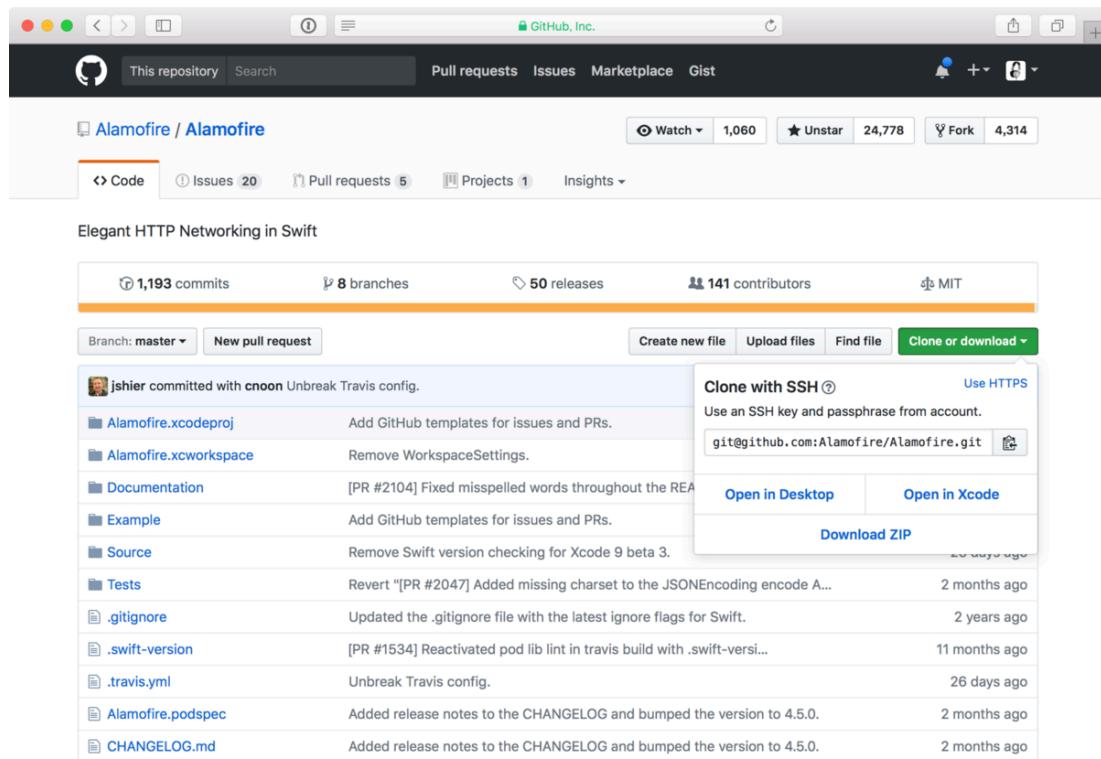


図 5.3: Web ブラウザからのクローン

コミットログの確認とブランチ・タグの操作

Xcode 9 からはリポジトリに関する情報を、ナビゲーターペインから簡単に確認できます。ナビゲーターペインの2番目のタブから、ブランチ、タグ、リモートブランチを一覧で確認できます（図5.4）。これまでのXcodeもVersion Editorからコミットログを確認できていましたが、その時に開いているブランチのログしか見られませんでした。

Xcode 9 では、ブランチを切り替えることなく各ブランチのコミットログを確認できるようになっています。各コミットにはコミットしたユーザーのGitHubアイコンも表示されるので、チームで開発している場合には誰がコミットしたのか分かりやすくなっています。

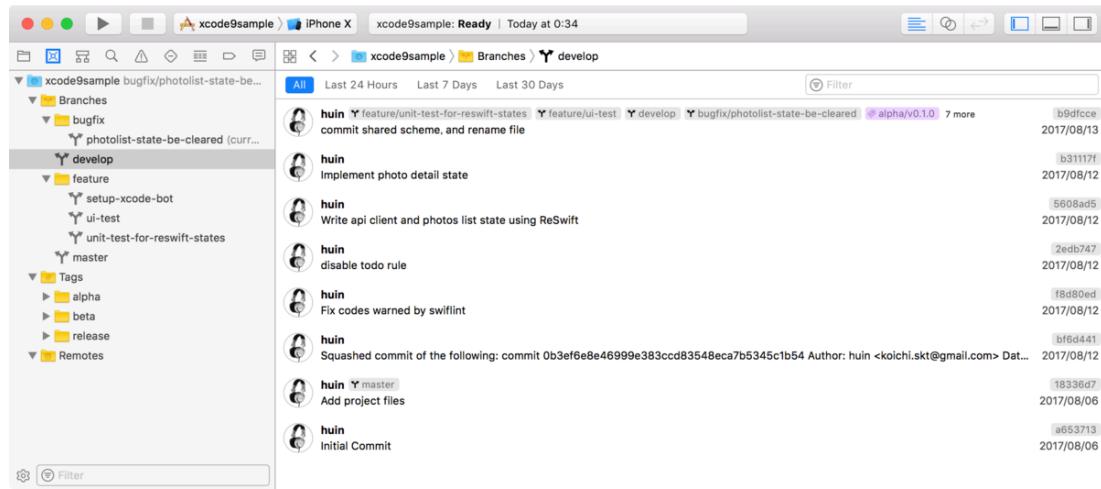


図 5.4: ナビゲータペインからのコミットログの確認

コミットログは指定したキーワードで検索できます。図 5.5 では `message: reswift` と入力していて、コミットメッセージだけを対象に `reswift` というキーワードで検索しています。これ以外にも `author:` や `revision:` (コミットハッシュ) でも検索できます。必要に応じてフィールドを指定すると良いでしょう。

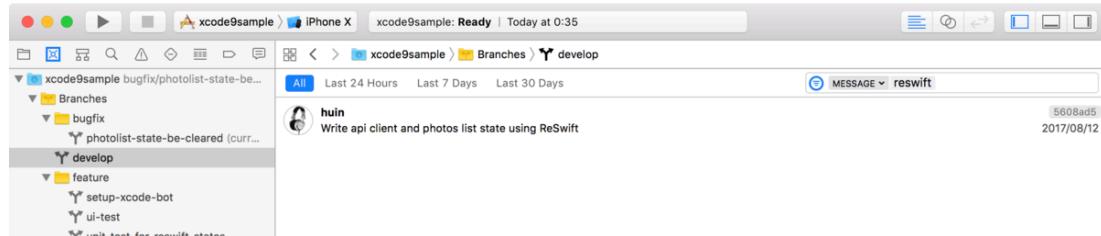


図 5.5: コミットログのフィルタ

ブランチの切り替えやマージ、タグの作成・削除といった操作は、ブランチのコンテキストメニューから行えます(図 5.6)。ブランチやタグを新しく作成する場合には、スラッシュ (/) で単語を区切っておくと、Xcode 上でグループにして表示してくれます(同図)。新規機能の開発ブランチに `feature/`、バグフィックスに `bugfix/` とつけるなど、命名規則を決めておけばブランチやタグを見やすく整理できます。

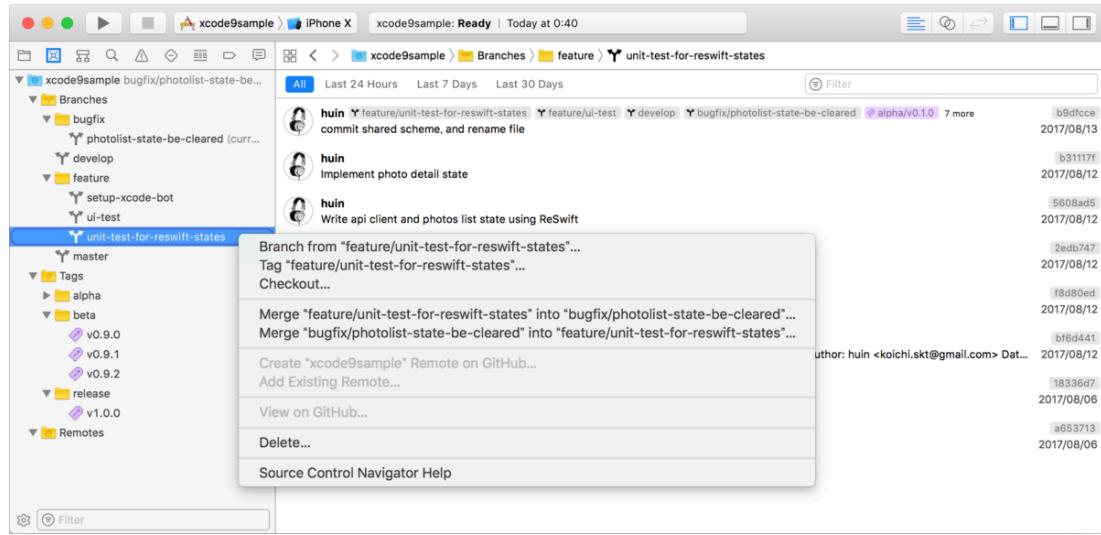


図 5.6: ブランチ操作

リポジトリの作成と公開

ここまででは、すでにあるリポジトリが存在する前提で、コミットログやブランチを確認するための方法を説明しました。自分で新たにプロジェクトを作成した場合や、ライブラリを公開したい場合はどうすればいいでしょうか。Xcode 9 では、リモートリポジトリの作成も直接行えます。

リポジトリの作成はソースコントロールナビゲーターで行います。「Remotes」グループのコンテキストメニューから、「Create "Project Name" Remote on Github」を選ぶと図 5.7 のようなダイアログが表示されます。アカウントや公開・非公開の設定も含めて、このダイアログからリポジトリを作成できます。

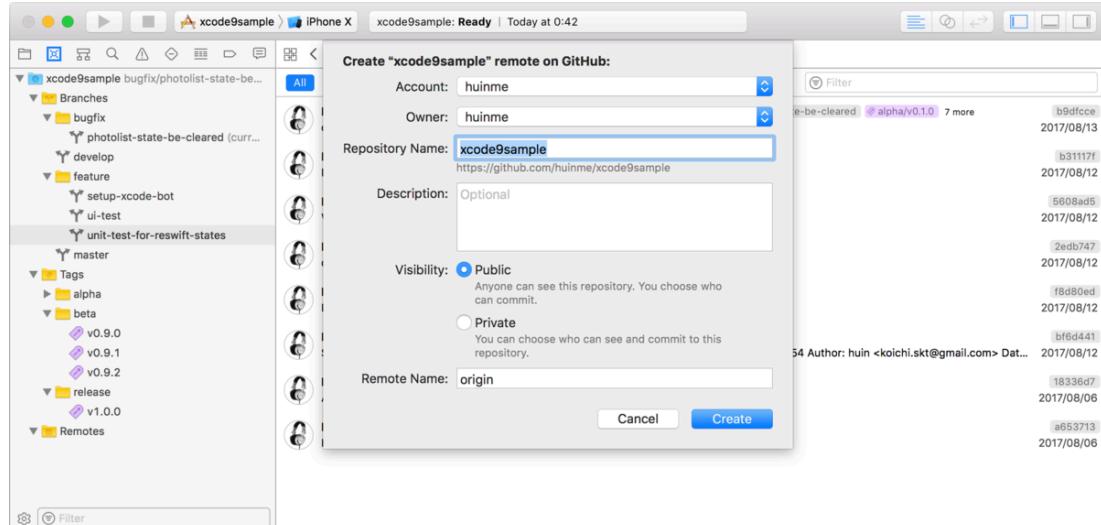


図 5.7: リポジトリの作成

.gitignore ファイルやライセンスファイルの生成が自動生成されないなど、Github の Web ページで作成する場合と比べていくつかの制限はあります。しかし、ブラウザと Xcode を切り替えながら公開するよりもはるかに早く、リポジトリを作成できます。

5.2.2 ソースコードエディタの改善

ソースコードエディタ関連での新機能の中で、一番の目玉は、なんといっても Swift のリファクタリングに対応したことです。

プログラミングにおいてリファクタリングは頻繁に発生する作業の一つですが、Swift のリファクタリングは、Swift 言語の登場以来ずっとサポートされていませんでした。Objective-C では問題なく利用できる機能でも、Swift のコードでリファクタリングをしようとするとき「Can't refactor Swift code.」と表示され、ため息を漏らした開発者は多いはずです。Xcode 9 ではこの状況がついに解消されました。

リファクタリング以外にも、コンテキストに応じたコード修正機能などの新機能が追加されています。これらの新機能も、iOS アプリ開発においてコーディングの効率を改善してくれる嬉しい機能になっています。

リファクタリング

リファクタリング実行時の操作手順は、これまでとほぼ同様です。リファクタリングしたい箇所でコンテキストメニューを表示し、一覧から「Refactor」を選びます（図 5.8）。メニューバーからだと「Editor」メニューの下にあります。^{*7}

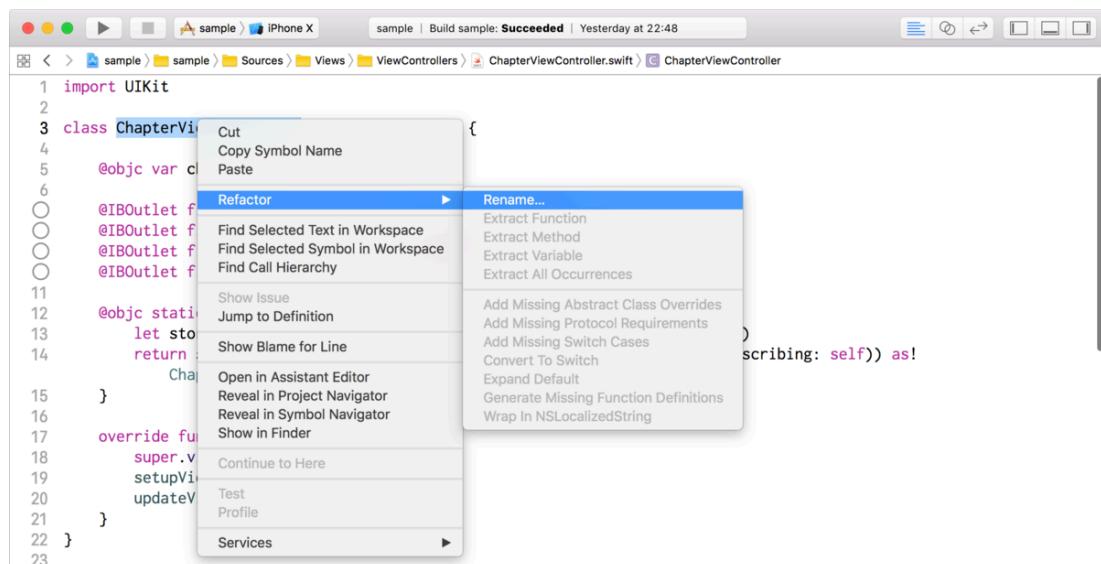


図 5.8: リファクタリングの実行

Xcode 9 では全 12 種類のリファクタリング操作が用意されています。表 5.1 に一覧を示します。

*7 従来の「Edit」メニュー下から移動しました。

編集中のソースの言語や選択中のコードの内容など、コンテキストに応じて利用可能な項目は変化します。

表 5.1: 利用可能なリファクタリング機能

項目名	内容
Rename...	変数やクラス、メソッド名などを変更する
Extract Function	選択範囲を関数として抽出する
Extract Method	選択範囲をメソッドとして抽出する
Extract Variable	ローカル変数を抽出する
Extract All Occurrences	ローカル変数を抽出する
Add Missing Abstract Class Overrides	未実装の仮想関数をオーバーライド（コードを挿入）する ^{*8}
Add Missing Protocol Requirements	未実装のプロトコルメソッドを挿入する
Add Missing Switch Cases	列挙型の値に対して <code>switch</code> 文を利用している場合に、不足している <code>case</code> 節をすべて挿入する
Convert To Switch	同じ値に対して <code>if else</code> を繰り返している場合に、 <code>switch</code> 文に置き換える ^{*10}
Expand Default	列挙型の値に対して <code>switch</code> 文を利用している場合に、 <code>switch</code> 文の <code>default</code> 節を定義する
Generate Missing Function Definitions	Objective-C などで、ヘッダーファイルで宣言されている未実装メソッドを実装ファイルで生成する
Wrap In NSLocalizedString	選択中の文字列リテラルを <code>NSLocalizedString()</code> で置き換える

例として、ビューコントローラのクラス名をリネームする状況を考えてみましょう。図 5.9 は、Storyboard から初期可能なビューコントローラクラス `ChapterViewController` のクラス名を変更するときのプレビュー画面です。ビューコントローラのクラス名を変更する場合、Storyboard のクラス指定などソースコード以外の部分まで考慮する必要があります。このプレビューから、Swift 内のコードはもちろん、Storyboard、Objective-C のコードも含めてリネーム対象として正しく検出できていることがわかります。

The screenshot shows the Xcode 9 interface with the 'Rename' refactoring dialog open. The file 'ChapterViewController.swift' is selected. The code editor displays the following Swift code:

```
File Name: ChapterViewController.swift
1 import UIKit
2
3 class ChapterViewController: UIViewController {
4
5     @objc static func instantiateFromStoryboard() -> ChapterViewController! {
6         let storyboard = UIStoryboard(name: "Main", bundle: Bundle(for: self))
7         return storyboard.instantiateViewController(withIdentifier: String(describing: self)) as!
8             ChapterViewController
9     }
10
11 }
12
13 extension ChapterViewController {
14
15 }
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47 }
```

The 'Main.storyboard' file is also open, showing a 'Chapter View Controller' with its 'Class Name' set to 'ChapterViewController'. Below the storyboard, the 'ChaptersListViewController.m' file is open, showing Objective-C code that creates an instance of 'ChapterViewController'.

図 5.9: クラス名のリネーム

もうひとつ注意が必要なケースとして、メソッド名をセレクター経由で呼び出している状況下で、そのメソッド名をリネームするケースがあります。このケースについても、セレクターで呼び出しているコード部分をリネーム対象として正常に検出できていました。また Objective-C のメソッドを Swift からセレクターで呼び出している場合や、その逆の場合においても、セレクター呼び出しのコードを正常に検出できていました。図 5.10 は Objective-C で実装されたメソッドをリネームするときのプレビューです。Swift でセレクター呼び出しをしている部分もリネーム対象として検出されていることがわかります。

このように Xcode 9 のリファクタリング機能は、Swift と Objective-C が混在しているプロジェクトでも利用できるくらいの品質となっています。

The screenshot shows the Xcode interface with the following details:

- Project Navigator:** Shows three files: `ChaptersListViewController.h`, `ChaptersListViewController.m`, and `SampleViewController.swift`.
- Editor Area:** Displays the `ChaptersListViewController.h` file content:

```
1 #import <UIKit/UIKit.h>
2
3 @interface ChaptersListViewController : UIViewController
4
5 - (void)doSomethingWithValue:(NSString *)string;
6
7 @end
8
```
- Editor Area:** Displays the `ChaptersListViewController.m` file content:

```
50
51 - (void)doSomethingWithValue:(NSString *)string
52 {
```
- Editor Area:** Displays the `SampleViewController.swift` file content:

```
9     let viewController = ChaptersListViewController()
10    viewController.perform(#selector(ChaptersListViewController.doSomethingWithValue:)), with: "Example
11        Text")
```

図 5.10: メソッド名のリネーム：Objective-C のメソッドを Swift から呼び出している場合でも検出できます

アクションメニュー

リファクタリング機能に関連して、Xcode 9 からはアクションメニュー^{*11}と呼ばれる機能が追加されています。これはコマンドキーを押しながらコードをクリックしたときに表示されるメニューのことです。Xcode 8 までは、同操作は単にコードの定義箇所へジャンプする機能に割り当てられていました。Xcode 9 からは、従来と同じ定義箇所へのジャンプも含めて、クリックした場所に応じたリファクタリング項目や、コードの修正メニューが表示されます。

図 5.11 は、クラス定義の末端の}の部分でアクションメニューを表示させた時の様子です。メニューに表示されているアクションは、次のとおりです。

- **Jump to Definition** : 定義箇所へのジャンプ
 - **Show Quick Help** : クイックヘルプ(ドキュメントコメント)の表示
 - **Edit All in Scope** : 現在のスコープ内での一括編集
 - **Fold** : コードの折りたたみ
 - **Add Method** : メソッドの追加
 - **Add Property** : プロパティの追加
 - **Rename...** : リネーム(この場合はクラス名)

たとえばここで **Rename...** を選択すると、リファクタリング機能のリネームが実行されます。このように、Xcode 9 ではクリックアクションからコードの修正やリファクタリングを開始できます。

*¹¹ Release Notesによると"Direct manipulation of code structure such as tokens and blocks."と説明されていますが、Plastforms State of the Union のデモでは"Actions menu"と呼んでいたためこの名前で説明しています。

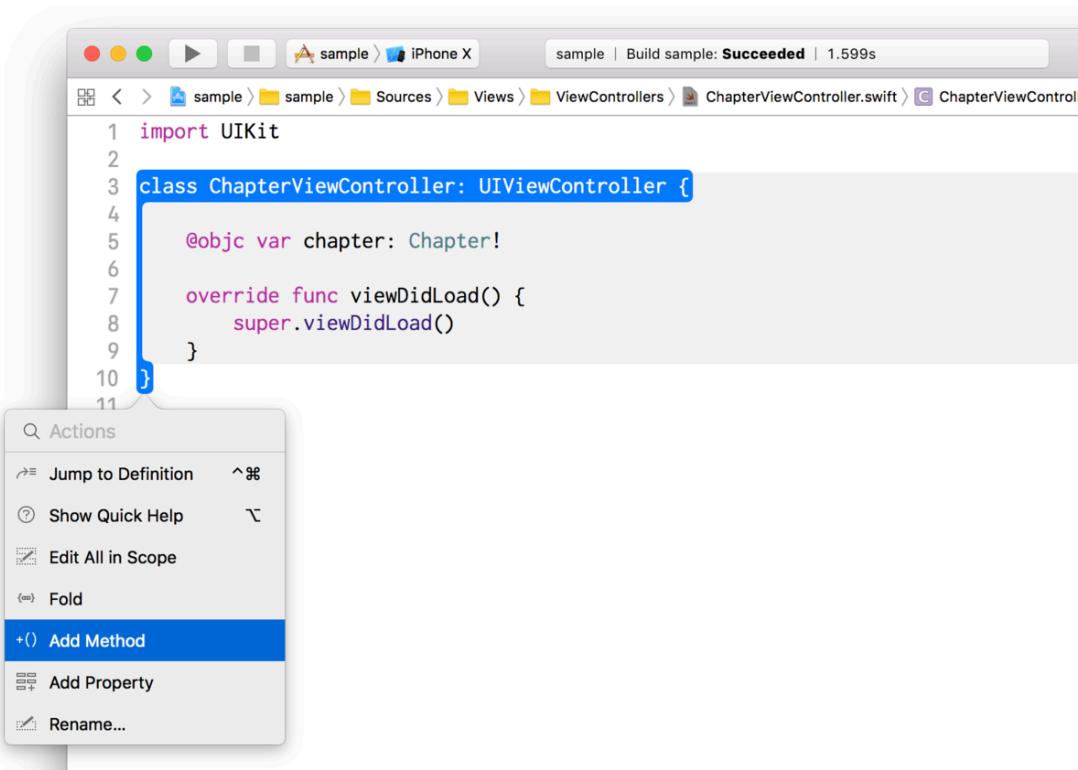


図 5.11: クラスへのアクション

クリックアクションに表示されるメニューは、表示させるコードのコンテキストによって変化します。クラスではなくメソッド定義部分で表示させた場合、引数や返り値の追加アクションが表示されます。if や switchなどの分岐処理では、else や case など別の分岐ブロックを追加するアクションが表示されます（図 5.12）。

The screenshot shows the Xcode 9 interface with a code editor window. The file is ChapterViewController.swift and the function is updateViews(). A cursor is positioned inside an if block. A context menu is open, titled 'Actions', with the 'Fold' option highlighted. Other options include 'Add "else" Statement', 'Add "else if" Statement', and 'Extract Method'. The code in the editor is as follows:

```
31 func updateViews() {
32     if let chapter = chapter {
33         let format = NSLocalizedString("chapter-number-and-title", comment: "第%ld章 %@", ...
34         titleLabel.text = String(format: format, chapter.number, chapter.title)
35         authorNameLabel.text = chapter.author.name
36         leadTextView.text = chapter.lead
37         authorProfileImageView.image = UIImage(named: chapter.author.profileImageName)
38     }
39 }
40
41
42
43
44
45
46
47
48
49
50
51
```

図 5.12: 'if'文の分岐追加

Xcode 9 のクリックアクションを利用すると、状況に応じた適切な編集作業が実行できます。

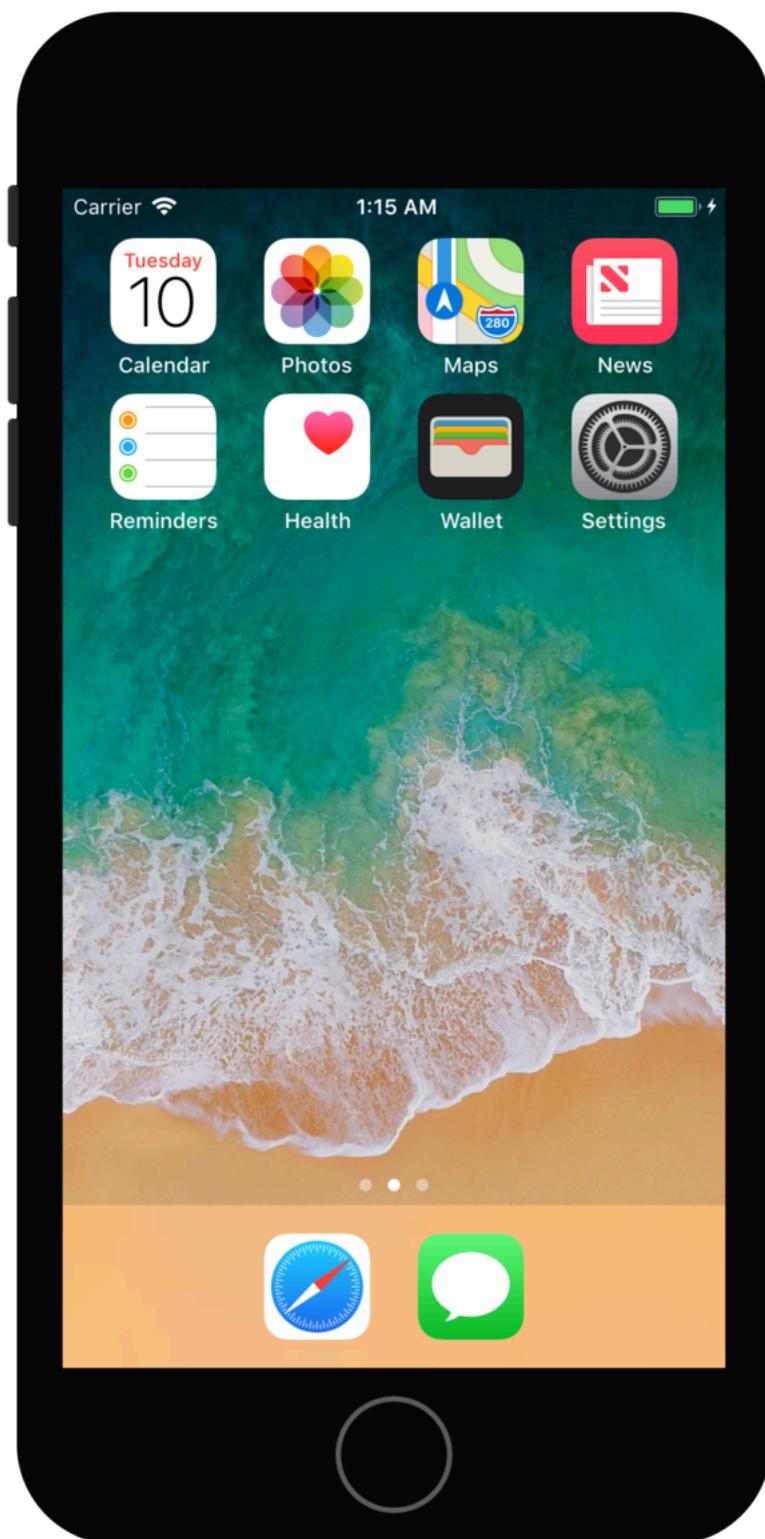
5.2.3 シミュレータの新機能

Xcode 9 で大きく改善された機能を一つ上げるとすれば、筆者は「iOS Simulator」を挙げます。これまで味気なくスクリーンだけの表示だったインターフェイスが、Xcode 9 では筐体部分も含めた端末全体が表示されるように変更され、より直感的にハードウェア部分を操作できます。また、シミュレータの同時起動や Mac からのデータ共有など、開発スピードを上げてくれる新機能も追加されています。

新しくなった UI とハードウェア機能のシミュレーション

Xcode 9 のシミュレータを起動したときに、何より最初に気づくのが新しくなったインターフェイスです。これまでデバイスのスクリーン部分だけが表示されていましたが、Xcode 9 からは筐体も含めた端末全体が表示されるようになりました（図 5.13）*12

*12 「Window」メニューの「Show Device Bezels」で従来の UI に戻せます。



iPhone 8 - iOS 11.0

図 5.13: 新しくなった iOS シミュレータ

これにともなって、端末のハードウェア操作もシミュレータ上で実行できるようになっています。たとえば、これまでホームボタンのクリックは⌘ + シフト + H のショートカットで操作していました。新しい iOS シミュレータでは、画面上のホームボタンをクリックすることで操作できます。ホームボタンだけではなく、シミュレータの両側にあるロックボタン、サイレントスイッチ、ボリュームボタンもクリックによって反応するため、より直感的に端末の操作を再現できます。

また、これまでショートカット (⌘ + 数字) で固定のウインドウサイズにリサイズしていましたが、新しい iOS シミュレータでは通常のアプリケーションと同様に、ウインドウの角部分にポインタをあわせることで自由なサイズにリサイズできます。標準サイズへは⌘ + 2 で戻せます。

複数シミュレータの起動

iOS 11 がサポートする端末は、iPad も含めると 8 種類もあります。^{*13} このような状況でアプリ開発時にもっとも苦労することの一つが、各画面サイズでの動作確認です。これまでの iOS シミュレータは一度に 1 種類の端末しか起動できませんでした。そのため、違うサイズ、あるいは違う OS バージョンのシミュレータで動作確認をしようと思うと、そのたびにシミュレータが起動するのを待つ必要があり、とても不便でした。実際、多くの開発者にとっても不便な状況だったようで、Facebook 社はシミュレータを複数起動するための独自のツールをオープンソースでリリースしています。^{*14}

Xcode 9 ではこの制限からついに解放され、複数のシミュレータを同時に起動できます（図 5.14）。



図 5.14: 複数シミュレータの起動

シミュレータの複数起動は、「Hardware」メニューの「Device」から非表示の新しいデバイスを選択するだけです。これまでだと動作中のシミュレータが終了して新たなシミュレータが起動していましたが、Xcode 9 からは起動中のシミュレータが増えています。

^{*13} <https://www.apple.com/jp/ios/ios-11/>

^{*14} <https://github.com/facebook/FBSimulatorControl>

今回の複数起動サポートによって、1つのシミュレータを終了するときのショートカットは⌘ + Wに変更されています。これまでのように⌘ + Qを押してしまうと、すべてのシミュレータが同時に終了してしまうので注意してください。なお、⌘ + コントロール + Wを押すと終了オプションを選べます（図 5.15）。ここで「Keep Running」を選ぶと、ウインドウが閉じるだけで裏側では状態が保持されるため、次回また起動したい時に高速に復帰できます。複数のシミュレータを頻繁に入れ替えたい場合には、こちらの終了モードにしておくと便利かもしれません。

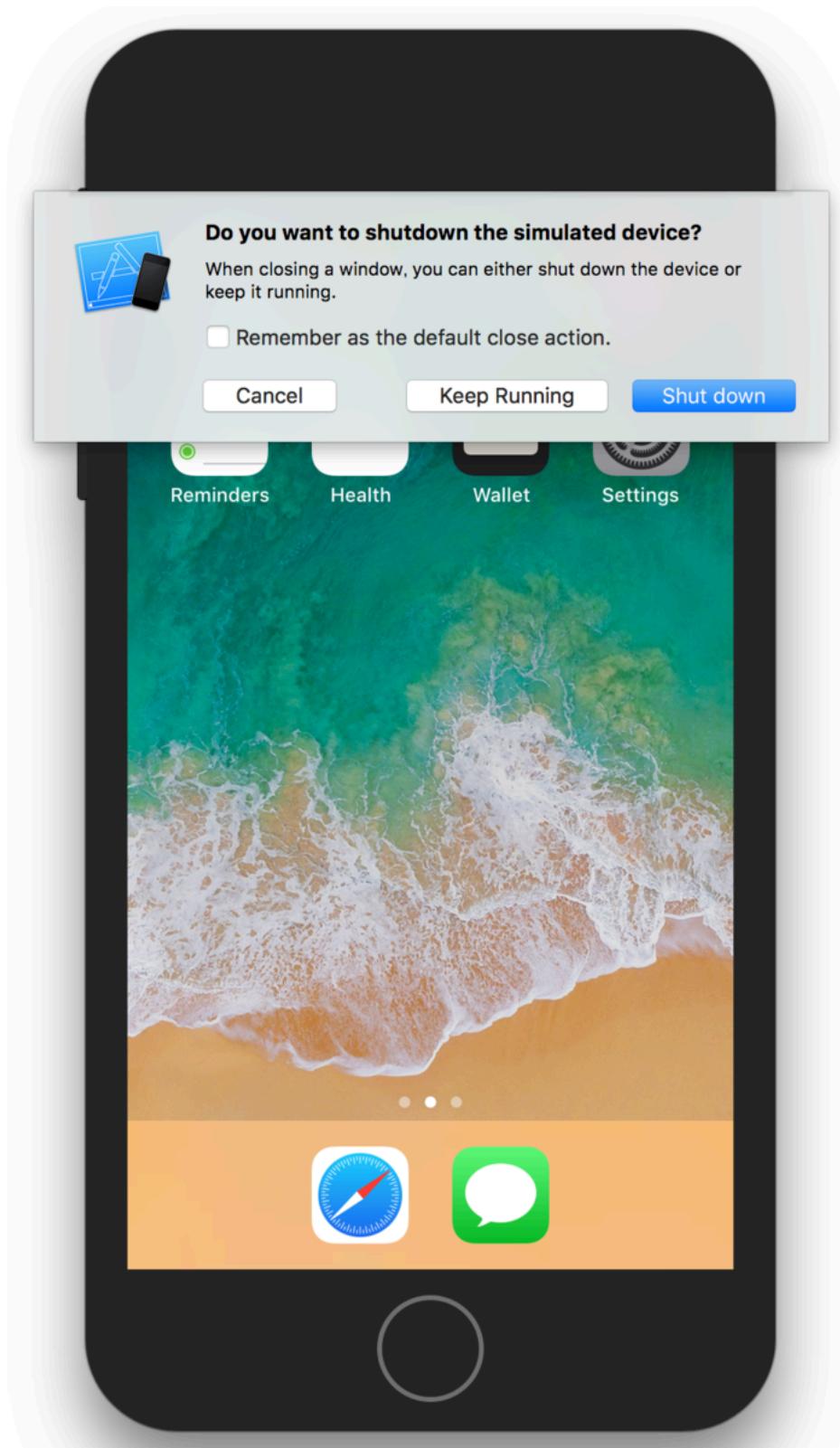


図 5.15: シミュレータの終了オプション

現在のところ、Xcode から実行するときに複数のシミュレータで同時に実行させることはできないようですが、今回の複数起動サポートによって動作確認が格段に楽になっています。

Mac 本体とのデータ共有機能

アプリの開発をしていると、Mac 側からシミュレータに対してデータを渡したいことがあります。写真の加工アプリであればテスト用の画像を使いたい場合があるでしょうし、位置情報を利用したアプリであれば、あらかじめ用意されたロケーションではなく、日本国内のロケーションを手軽に利用したい場合があると思います。Xcode 9 の iOS シミュレータでは、これらのデータを Mac の共有メニューを通して簡単に渡せます。

シミュレータに渡せるデータの種類は写真、位置情報、Web URL の3種類です。

写真については、これまで画像のドラッグ＆ドロップによってシミュレータに渡していたので、共有の方法が1つ増えたことになります。2つめの位置情報についても、任意の位置情報を渡すことはこれまで出来ていました。しかし緯度経度を直接指定する方式だったため、おせじにも簡単とは言えない方法でした。

Xcode 9 からは Mac 付属のマップアプリケーションを使って位置情報を渡せます。シミュレータを起動した状態にしておき、マップアプリケーションの共有ボタンをクリックします。すると共有先にシミュレータが表示されるので、それを選べば渡せます。シミュレータ側でもマップを開き、現在地表示モードにしておけば、共有のタイミングで Mac 側と同じ位置を表示してくれます。また、開発中のアプリでも `CLLocationManager` で位置情報を受け取れるようにしておけば、マップ同様に位置情報を受け取れます。従来のように、緯度経度を直接入力するよりもはるかに簡単に共有できます。



図 5.16: 位置情報の共有

3つ目の Web URL の共有も、マップと同じ要領で Safari の共有メニューからシミュレータを選ぶだけです。シミュレータ側では Mobile Safari が起動して、共有した Web ページを開いてくれます。モバイルサイトの開発中であれば、この方法で Web ページを開きつつ、さらに Mac 側の Safari で Web インスペクタを開くと効率的にデバッグを行えます。

なお、複数のシミュレータを起動している場合には、すべてのシミュレータに共有したり、1つだけを選んで共有したりできます。

5.3 デバッグフェーズの新機能

本節では、デバッグフェーズで便利な新機能について解説します。

新機能の開発中はもちろん、リリース後に不具合が見つかった場合など、デバッグ作業はアプリ開発では様々なところで発生します。コードを書いている時間と同じくらい、デバッグ作業に時間をかけることもあります。そのため、効率的にデバッグを行い問題の修正を行うことは、コードを効率的に書くことと同じくらい重要です。

Xcode 9 では、ワイヤレスデバッグをはじめとして、デバッグ作業をより効率的に行うための新機能が追加されています。

5.3.1 ワイヤレスデバッグ機能

2008 年に iOS SDK^{*15}がリリースされて以来、開発中のアプリを実機で動かすためには必ずケーブルで Mac と iOS デバイスを接続した状態で行う必要がありました。Xcode 9 からは、ケーブルで接続しなくても WiFi 経由でデバッグ実行ができる「ワイヤレスデバッグ機能」が追加されています。

ワイヤレスデバッグを行うには、簡単なセットアップを済ませておく必要があります。まず iOS 11 以降が動作する iOS デバイスをケーブルで Mac と接続し、Xcode 9 にデバイスを認識させます。そして「Window」メニューから「Devices and Simulators」を選ぶと、Xcode で認識しているデバイス一覧が表示されます。接続したデバイスがここに表示されたら、「Connect via network」にチェックを入れます（図 5.17）。チェックを入れたらケーブルを外します。

Mac と iOS デバイスが同じ WiFi ネットワークに接続できていれば、ケーブルを外しても「Connected」の一覧にデバイスが表示されているはずです。もし「Disconnected」が表示されてしまうようであれば、同じ WiFi ネットワークに接続できているか確認してください。

```
XXX: BLOCK_HTML: YOU SHOULD REWRITE IT
<!-- NOTE: 画像加工まとめて行うなら debug-wireless-setup-original.png を利用してください
Serial Number と Identifier が見えないようになっていれば OK です。
-->
```

*15 旧称は「iPhone SDK for iPhone OS」です。

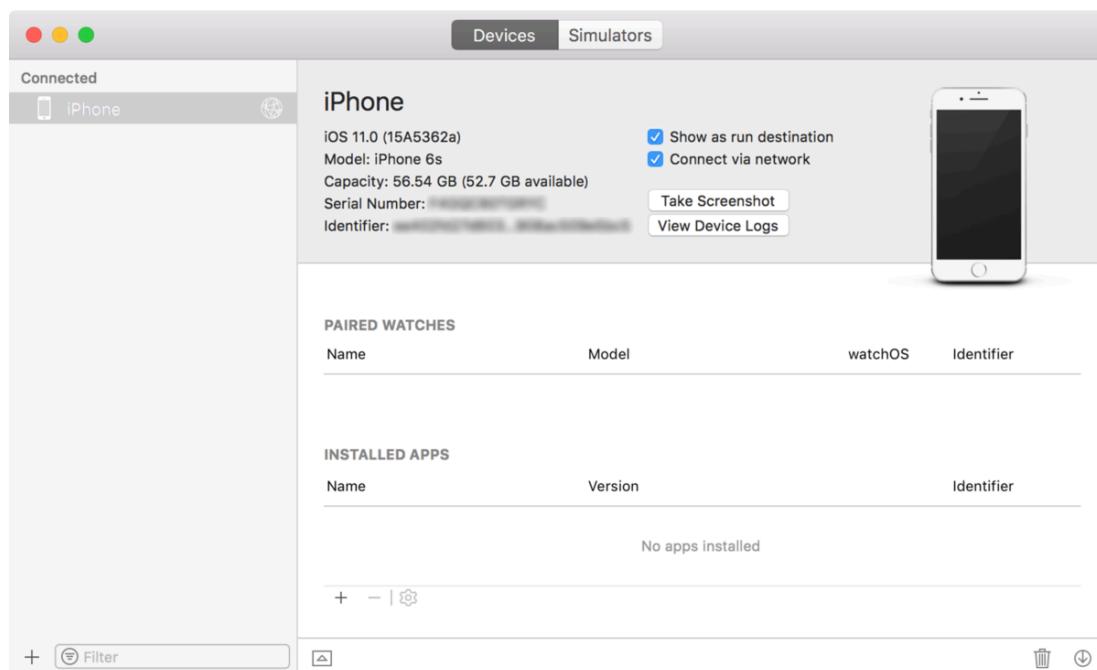


図 5.17: ワイヤレスデバッグの設定

セットアップさえ済ませてしまえば、あとはこれまでのように Xcode からアプリを実行し、デバッガ作業を行えます。ワイヤレスデバッグのために何かを意識する必要はありません。

もし、なんらかの理由で WiFi ネットワークが切れてしまい、デバイスとの接続が切れてしまった場合も慌てる必要はありません。デバイス側でアプリが実行中のままであれば、Xcode と接続状態に戻った後にデバッグを再開できます。図 5.18 のように、Xcode の「Debug」メニューから「Attach to Process」を選ぶとデバイスで稼働中のプロセス一覧が表示されるので、一番上に表示されるアプリのプロセスを選択すればデバッグを再開できます。

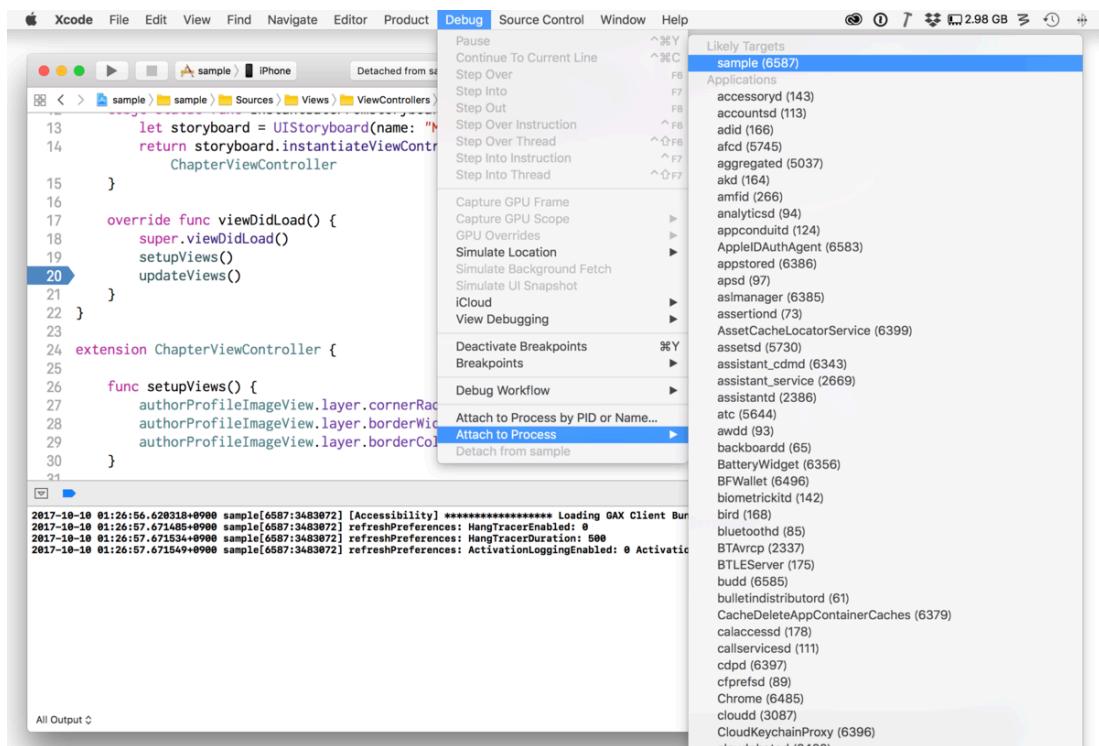


図 5.18: デバッグの再開

ここまでで説明したように、ワイヤレスデバッグはとても簡単に利用できます。強いて制限をあげるのであれば、デバイスが iOS 11 以降であることと、Mac とデバイスが同じ WiFi に接続されていることの 2 つくらいです。

ワイヤレスデバッグでは単にアプリを実行するだけなく、以下のような作業も行えます。

- ユニットテストの実行
- UI レコーディング・UI テストの実行
- Instruments によるプロファイリング

このようにこれまでケーブル接続で行っていた多くの作業を、Xcode 9 ではワイヤレスで行えるようになっています。

5.3.2 ビューデバッグの改善

次は、ビューデバッグの変更点について紹介します。

iOS アプリ開発では、画面表示されているビューオブジェクトをデバッグしたいことが頻繁にあります。特に Auto Layout や、Size Class を駆使した Adaptive UI を持つアプリの場合には、デバイスに応じてどのレイアウト制約が有効になっているかなど、多くの情報を確認しながら開発を進める必要があります。また最近の iOS アプリはビューコントローラをネストした複雑な UI を構築することも多いため、その場合にはビューだけでなくビューコントローラまで確認したいシチュエーションも増えてきました。

Xcode 9 では、ビューデバッグのための機能である **Debug View Hierarchy** が強化され、ビュー コントローラまでアクセスできるようになりました。アプリ実行中に「Debug View Hierarchy」ボタンをクリックすると、ビューコントローラまで含めた階層構造がデバッグナビゲーターに表示されます（図 5.19）。また、中央のプレビュー部分にもビューコントローラのクラス名が表示されます。

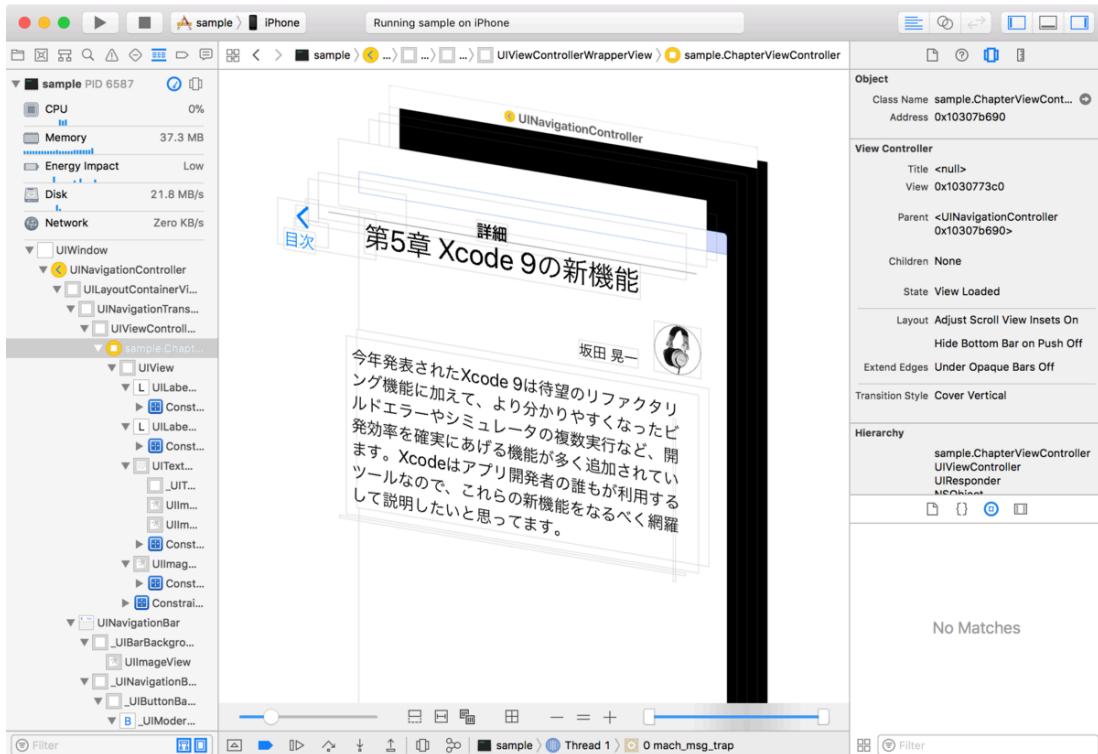


図 5.19: Debug View Hierarchy の改善点

ちなみに、ビューデバッグの状態では LLDB からビューコントローラにアクセスできないため、アドレスからコンビニエンス変数にキャストする必要があります。図 5.19 は、`ChapterViewController` という名前のビューコントローラクラスが、自身のもつ `chapter` プロパティに基づいてビューを表示している時のデバッグ画面です。この時、ビューコントローラのアドレスが `0x10307b690` だとすると、LLDB で `chapter` プロパティの内容を見るためには、リスト 5.1 のように LLDB コマンドを実行します。

ポイントは `unsafeBitCast(to:)` でアドレスから指定の型にキャストし、その結果をコンビニエンス変数`$vc` にアサインしていることです（`vc` は任意の名前を使用します）。これでビューコントローラの各メソッドを呼び出せます。

リスト 5.1: アドレスからコンビニエンス変数へのキャスト

```
(lldb) expr -l Swift -- $vc.chapter
(sample.Chapter?) $R0 = 0x00000001c009e640 {
    ObjectiveC.NSObject = {}
    number = 5
    title = "Xcode 9 の新機能"
    lead = "今年発表された Xcode 9 は待望のリファクタリング機能に加えて、より分かりやすくなったビルドエラーやシミュレータの複数実行など、開発効率を確実にあげる機能が多く追加されています。Xcode はアプリ開発者の誰もが利用するツールなので、これらの新機能になるべく網羅して説明したいと思ってます。"
    author = 0x00000001c0468280 {
        ObjectiveC.NSObject = {}
        name = "坂田 晃一"
        profileImageName = "huin"
    }
}
```

UIKit だけでなく、SpriteKit と SceneKit 向けにも強化されています。図 5.20 は、WWDC 2017 のサンプルコード^{*16}の実行中にビューデバッグを起動したときの画面です。SCNView（あるいは SKView）内の各ノードまで詳細に表示されていることがわかります。また、この状態で画面をドラッグするとカメラ位置を自由に移動でき、各ノードのレイアウトなどを直感的に確認できます。

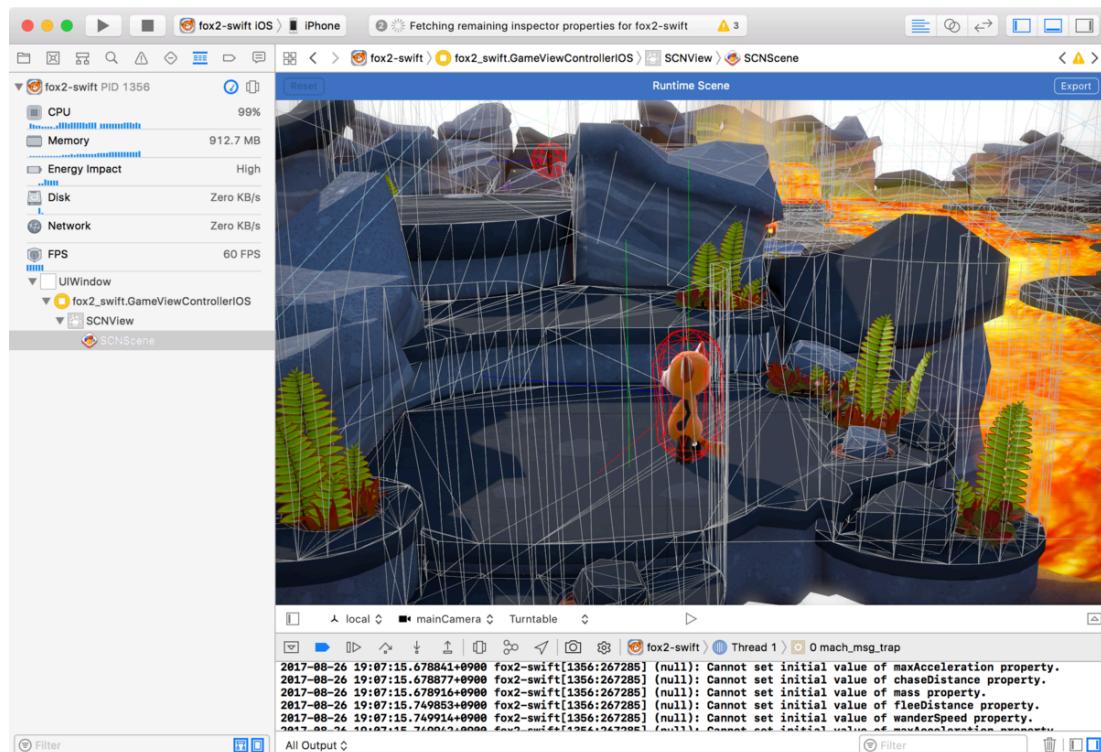


図 5.20: SceneKit のビューデバッグ

^{*16} <https://developer.apple.com/library/content/samplecode/scenekit-2017/Introduction/Intro.html>

このように、Xcode 9 ではビューデバッグ機能が大きく強化されていて、UIKit はもちろん、SpriteKit や SceneKit でもビュー情報を確認しやすくなっています。

5.4 テストフェーズの新機能

開発、デバッグときて、次はテストフェーズで便利な新機能を紹介します。Xcode のテスト関連の機能は、2013 年に Xcode 5 に XCTest フレームワークが登場して以来、Xcode 6 ではパフォーマンステストや非同期テストのサポート、Xcode 7 で UI テストのサポートと毎年着実にその機能が強化されてきました。Xcode 9 でも、テストアタッチメントのサポートやテストの並列実行など多くの改善が施されています。

5.4.1 XCTest の新 API

ここでは、UI テストに限らず iOS アプリのテスト全体において役立つ新 API を紹介します。

テストアタッチメント

アプリのテストをしていると、テスト中に利用したデータを成果物として保存しておきたいことがあります。たとえば、ある Web API のレスポンスをパースするテストがあるとして、パース失敗時のデータを分析用に保存する必要があるかもしれません。写真の加工アプリであれば、加工後の写真を保存できれば UI テストと組み合わせることで機能テストが検証しやすくなるでしょう。

Xcode 9 ではこういった成果物をテストアタッチメントとして保存し、レポート画面から確認できます。

具体的なコードをもとに説明します。まずはもっとも簡単な例として、文字列をテストアタッチメントとして保存してみます。リスト 5.2 では、UUID 文字列を使って XCTAttachment オブジェクトを初期化し、それを XCTestCase クラスの add(_:) メソッドで保存しています。XCTAttachment クラスは Xcode 9 で追加された新しいクラスです。

リスト 5.2: UUID を保存する処理

```
"""
class sampleTests: XCTestCase {

    func testSaveUUID() {
        let uuid = UUID().uuidString

        let attachment = XCTAttachment(string: "UUID : \(uuid)")
        attachment.lifetime = .keepAlways
        add(attachment)
    }
}
```

レポートナビゲーターからテスト結果を見ると、確かに生成した文字列がテキストファイルとして保存されていることがわかります（図 5.21）。テストアタッチメントは Xcode 内で確認するだけではなく、クリップのアイコンをクリックして任意の場所へも保存できます。これがテストアタッ

チメントの基本的な使い方です。

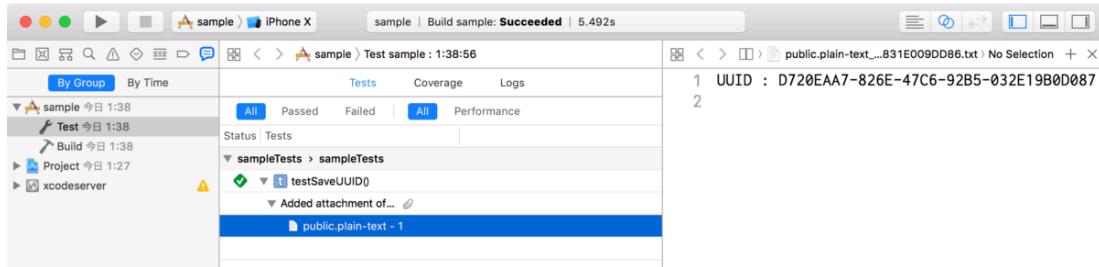


図 5.21: テストアタッチメントの確認

もう少し詳しく見ていきましょう。このクラスを利用して成果物を保存します。生成したアタッチメントオブジェクトを渡している `add(_:)` メソッドは、こちらも Xcode 9 で新しく追加された `XCTActivity` プロトコルのメソッドです。`XCTestCase` がこのプロトコルに適合しているので、インスタンスマソッドとして呼び出し、アタッチメントを保存しています。

アタッチメントとしてデータを残すポイントは 7 行目です。`XCTAttachment` の `lifetime` プロパティは、テストアタッチメントの保存期間を表すプロパティです。実行中のテストが成功した場合に、アタッチメントを破棄するかそれとも保持しつづけるかを指定します。プロパティの型は `XCTAttachment.Lifetime` で、`.keepAlways` か `.deleteOnSuccess` のどちらかの値をとります。このコードでは `.keepAlways` を指定しているため、成功したテストのレポート画面で確認できます。

テスト成功時にテストアタッチメントを削除するか保持するか、デフォルトの挙動はスキームの設定で変更できます。Xcode のスキーム編集画面で、テストアクションのオプションタブを開くと、「Delete when each test succeeds」という項目があります（図 5.22）。この項目のチェックを外せば、すべてのテストアタッチメントについて成功後も保持するようになります。Xcode 9 で新規作成したプロジェクトはデフォルトでチェックが入っているので、その状態でアタッチメントを保持するためには、リスト 5.2 のようにアタッチメントごとに保持期間を指定する必要があります。どのような目的でテストアタッチメントを利用したいかによって設定を変更するといいでしょう。

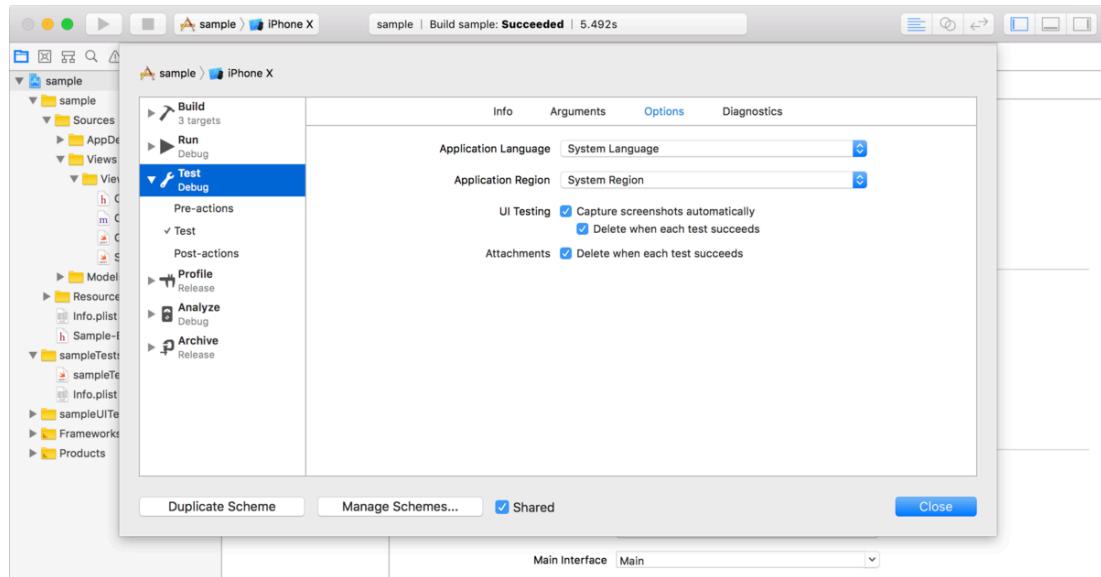


図 5.22: テストアタッチメントの保存期間の設定

さて、リスト 5.2 では話を簡単にするために文字列をそのまま保存しましたが、`XCTAttachment` は文字列以外のデータでも初期化できます。`XCTAttachment` クラスのイニシャライザを見てみると、多くのイニシャライザが用意されていることがわかります。たとえば `init(contentsOfFile:)` は任意のファイルの URL を渡してファイルを成果物として保存します。また、`init(image:quality:)` を利用すると画像を、品質を指定した上で保存できます。

なお、テストアタッチメントはリスト 5.3 のパスに保存されています。レポート画面から 1 つずつ保存するのが面倒な場合は、このディレクトリをまるごとコピーすることであとから簡単に参照できます。

リスト 5.3: テストアタッチメントの保存場所

```
"""
~/Library/Developer/Xcode/DerivedData/{プロジェクト固有名}/Logs/Test/Attachments
```

このように、Xcode 9 では `XCTAttachment` クラスを利用して任意のデータを成果物として自由に保存し、テストやアプリの品質を改善するために利用できます。

UI テストとスクリーンショット API

UI テストの際にはテスト中の画面を保存するための API も用意されています。UI テストでスクリーンショットを保存できると、各端末でのレイアウトチェックも効率的に行えて、とても便利です。

ここでは図 5.23 のようなアプリを想定して解説します。本書の目次と各章の執筆者およびリード文を表示するだけの簡単なアプリです。

```
--[[path = (not exist)]]--
```

本書の目次とリード文を表示するアプリ

```
XXX: BLOCK_HTML: YOU SHOULD REWRITE IT
<!--![本書の目次とリード文を表示するアプリ](images/test-unit-test-screenshot-sampleapp-01.png)
![本書の目次とリード文を表示するアプリ](images/05testunittestscreenshotsampleapp02.png)
-->
```

まずは、単純に起動直後の画面を保存するコードをリスト 5.4 に示します。難しいところはなく、アプリケーションプロキシの `screenshot()` を呼んで `XCUIScreenshot` オブジェクトを取得して、それをアタッチメントとして保存しているだけです。`XCTAttachment` にはスクリーンショットオブジェクトから初期化するためのイニシャライザが用意されているため、他のデータ形式と同様にアタッチメントを作成できます。11 行目の `name` プロパティは必須ではありませんが、アタッチメントの名前を指定しておくことで、あとで確認しやすくしています。

リスト 5.4: ‘`screenshot()`’を利用したスクリーンショットの保存

```
"""
class sampleUITests: XCTestCase {

    func testCaptureChapterListView() {
        let app = XCUIApplication()
        app.launch()

        // 起動直後の画面
        let screenshot: XCUIScreenshot = app.screenshot()
        let attachment = XCTAttachment(screenshot: screenshot)
        attachment.lifetime = .keepAlways
        attachment.name = "Chapter List"
        add(attachment)
    }
}
```

このテストの実行結果が図 5.24 です。指定した名前で、起動直後の目次画面が正常に保存できていることがわかります。

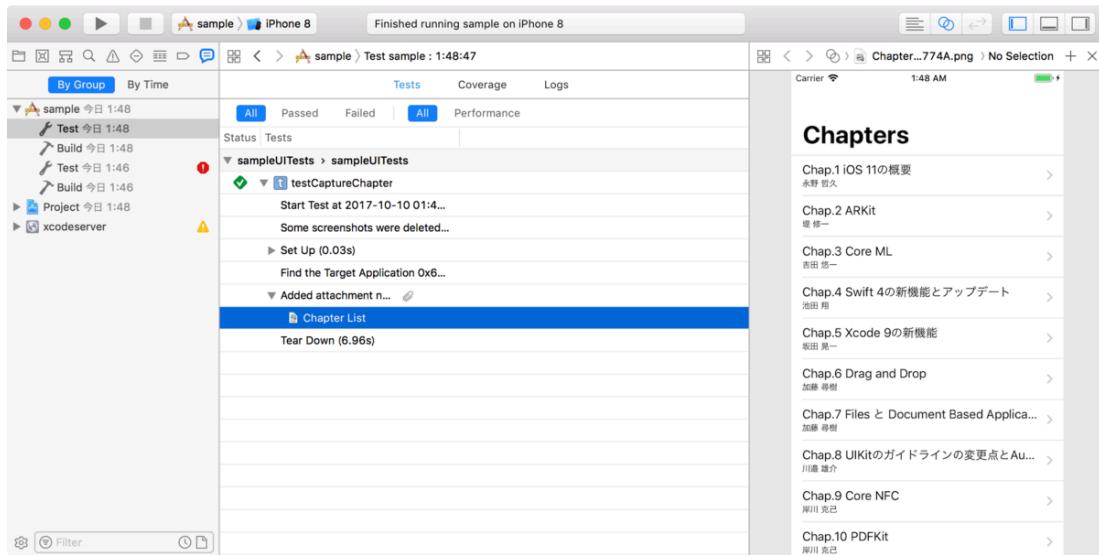


図 5.23: 保存されたスクリーンショットの確認

API の説明をしましょう。スクリーンショットオブジェクトを取得した `screenshot()` メソッドは、Xcode 9 で追加された `XCUIScreenshotProviding` プロトコルのメソッドです。`XCUIApplication` クラスはこのプロトコルに適合しているため、スクリーンショットを取得できます。この `XCUIScreenshotProviding` プロトコルは、実は `XCUIElement` クラスも適合しています。そのため画面全体ではなく特定のビューをルートにスクリーンショットを取得したい場合には、UI 要素オブジェクトに対して `screenshot()` メソッドを呼びます。

この他に `XCUIScreenshotProviding` に適合しているクラスには、Xcode 9 で追加された `XCUIScreen` クラスがあります。こちらは `UIScreen` と同様に物理スクリーンを表現するクラスで、画面全体のスクリーンショットを取得できます。そのためリスト 5.4 は、`XCUIScreen.main` からスクリーンショットを取得することでも同じ結果を得られます。

```
XXX: BLOCK_HTML: YOU SHOULD REWRITE IT
<!--
NOTE: XCUIScreen.screens でセカンドディスプレイのスクリーンショットを撮ろうとすると落ちる。
ドキュメントの記述とズれてるので Xcode GM でたら再検証。
有線で物理接続した画面しかサポートしていないか、
セカンドディスプレイの準備が整うまでスリープが必要かもしれない。
-->
```

スクリーンショットの自動取得

UI テストでは `XCUIScreenshot` オブジェクトを取得することで簡単にスクリーンショットを保存できます。しかしこの方法の欠点は、取得タイミングを自分で記述しなければならないことです。画面遷移ごとにすべてのスクリーンショットを取得したい場合を考えてみると想像できるように、あらゆる場所にスクリーンショット保存のコードを書くのはとても面倒です。

このような手間を削減するために、Xcode 9 では UI テスト中に自動でスクリーンショットを保存してくれる機能が追加されています。自動保存を有効にするには、スキーム編集画面にある、`test-unit-test-enable-auto-screenshots` がない

取得したスクリーンショットをテスト成功時にも削除せず保存したい場合には、「Delete when each test succeeds」のチェックを外しておきます。

この状態で全画面を表示するように UI テストを書いておけば、すべての画面のスクリーンショットがテスト後に手に入ります。この自動取得機能を利用すれば、冗長なスクリーンショットの保存コードが消え、UI テストだけを実施するキレイなテストコードにできます。

筆者が確認した範囲では、おおよそすべての UI 操作ごとにスクリーンショットが保存されました。テスト内容によっては希望の状態のスクリーンショットが手に入らないかもしれません、手動でコードを書くことに比べると、遙かに簡単にスクリーンショットを取得できます。

アクティビティによるログのグループ化

UI テストのレポート画面では、すべての UI 操作ごとにログが出力されます。たとえば全画面を表示するようなテストでは、図 5.25 のように、似た操作のログが大量に出力されることになります。この多くのログの中からスクリーンショットを確認しようと思うと、とても面倒です。

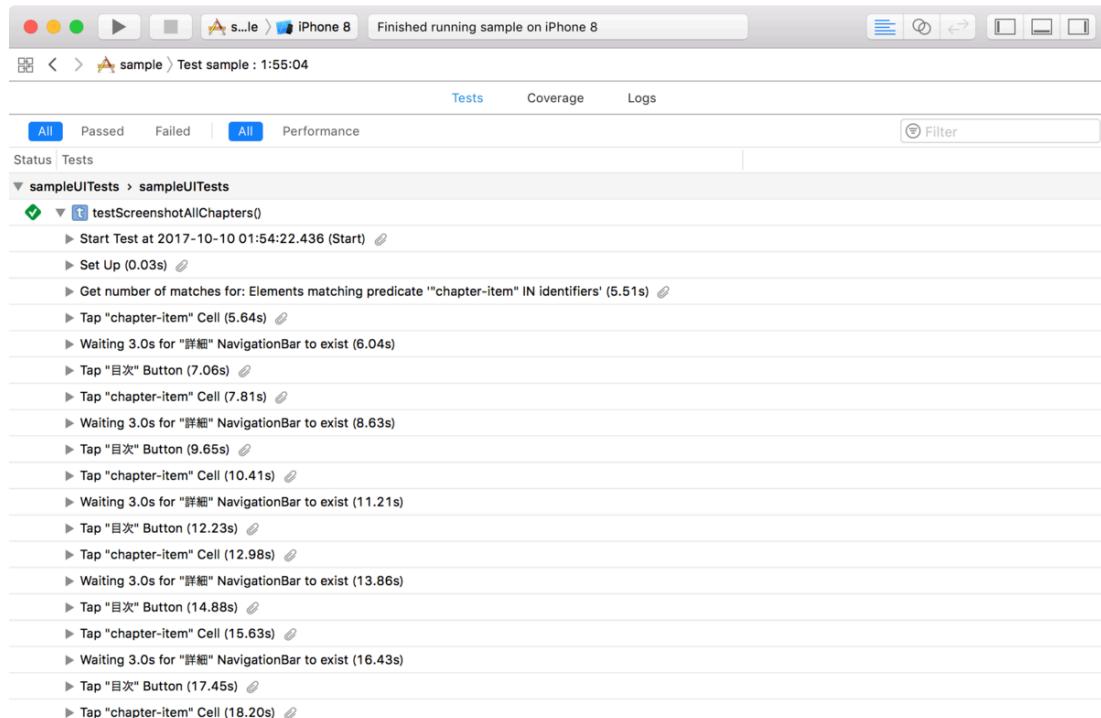


図 5.24: 通常の UI テストログ

しかし `XCTContext` というクラスを利用すれば、一連の操作を「アクティビティ」としてグループ化し、ログを階層表示にできます。

使い方は簡単で、`XCTContext.runActivity(named:block:)` メソッドにグループ名と UI 操作

を含んだブロックを渡すだけです。この API を利用したテストコードがリスト 5.5 です。

リスト 5.5: ‘XCTContext’を使ったテストコード

```
"""
class sampleUITests: XCTestCase {

    func testScreenshotAllChapters() {
        let app = XCUIApplication()
        app.launch()

        let cells = app.tables
            .children(matching: .cell)
            .matching(identifier: "chapter-item")

        for i in 0..<cells.count {
            XCTContext.runActivity(
                named: "Capture Screenshot : Chapter \\"(i + 1)"") { _ in
                    cells.element(boundBy: i).tap()
                    XCTAssertTrue(app.navigationBars["詳細"].waitForExistence(timeout: 3))
                    app.navigationBars.element.buttons["目次"].tap()
                }
            }
        }
    }
}
```

このテストを実行すると、詳細画面での操作が章ごとにグループ化されたレポート画面が得られます（図 5.26）。

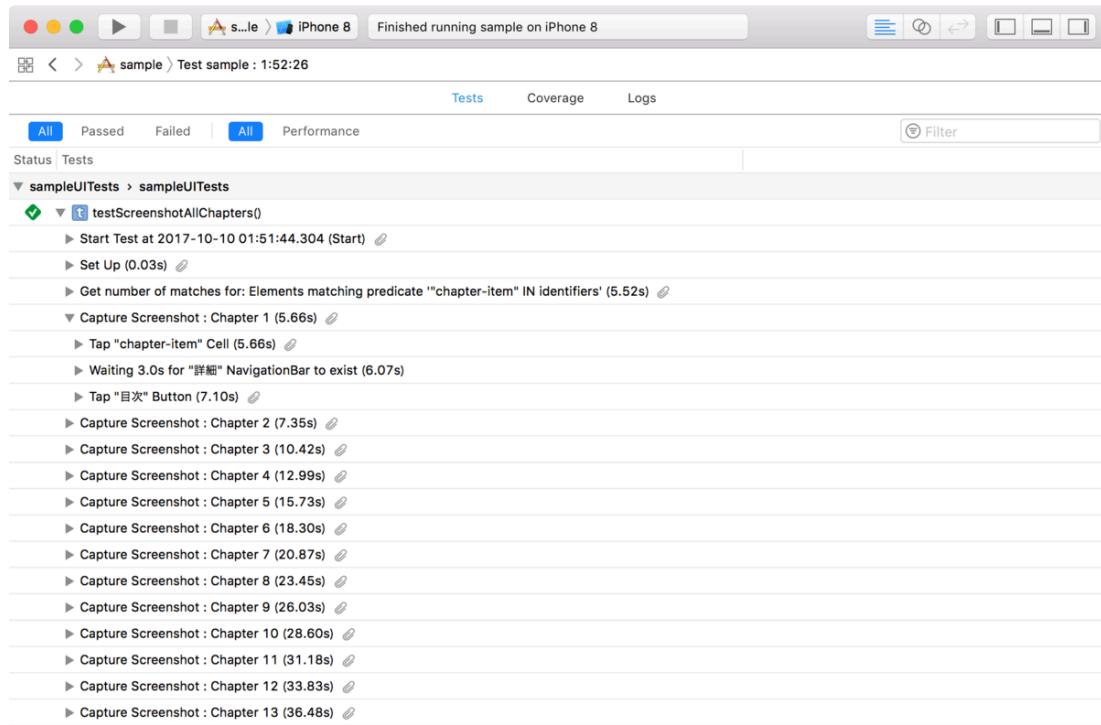


図 5.25: ‘XCTContext’を使ったテストのレポート

このように、Xcode 9 では `XCTContext` クラスを利用してことで、テストログをグループ化して見やすくなっています。

5.4.2 UI テストの新機能

Xcode 7 で追加された UI テストにおいて、Xcode 9 では複数のアプリケーションを同時実行できる機能が追加されました。

iOS 8 から追加された「App Groups」によって、アプリ間でデータを共有できるようになっています。App Groups はアプリ利用時では便利ですが、テストにおいては、アプリの挙動が他のアプリの状態に依存する環境ができたとも言えます。もちろんテスト用のデータを事前に用意することでも対処できますが、シナリオテストのような一連の操作をテストする場合には難しいところもありました。この課題を解決するために、Xcode 9 ではテストターゲット以外のアプリケーションも操作できる API が追加されています。

具体的な例をみてみましょう。UI テストで他のアプリケーションを起動するには、テストメソッド内でリストリスト 5.6 のように書きます。

リスト 5.6: キャプション

```
// テストターゲットアプリの起動
let app = XCUIApplication()
app.launch()
```

```
// 他のアプリの起動
let otherApp = XCUIApplication(bundleIdentifier: "com.yourcompany.apps.otherapp")
otherApp.activate()
```

XCUIApplication クラスの `init()` および `init(bundleIdentifier:)` イニシャライザは、Xcode 9 から追加されたメソッドです。前者はテストターゲットのアプリに対するプロキシオブジェクトを生成し、後者では任意のアプリケーションへのプロキシオブジェクトを生成します。プロキシオブジェクトを生成した後は、`activate()` メソッドを呼ぶことでフォアグラウンドに切り替えられます。

`activate()` メソッドは同期処理を行うので、後続の処理では待ち時間をとる必要はありません。起動したいアプリケーションが同じプロジェクト内の別ターゲットであれば、起動前にインストールもしてくれるので、事前にインストールされているかを気にする必要もありません。

もし、アプリケーションの起動状態をチェックしたいのであれば、XCUIApplication クラスの `state` プロパティを利用します。このプロパティは `UIApplicationState` 同様にアプリケーションの実行状態を表します。ただし、非同期で変化するプロパティなので、リストリスト 5.7 のように非同期テスト用の `Expectation` クラスを利用してアサートします。

リスト 5.7: キャプション

```
let app = // initialize and activate application

let predicate = NSPredicate(format: "state == \(\(XCUIApplication.State.runningForeground.rawValue))")
let expectation = XCTNSPredicateExpectation(predicate: predicate, object: reader)
wait(for: [expectation], timeout: 10)
```

XCTNSPredicateExpectation クラスは `XCTestExpectation` を継承している非同期テスト用のクラスで、Xcode 8.3 で追加されたクラスです。

このクラスをはじめとする、非同期テスト用のクラスやメソッドについては、WWDC 2017 のセッション^{*17} やドキュメント^{*18} を確認してください。

UI テストパフォーマンスの改善

UI テスト API を利用したシナリオテストでは、表示中のすべての UI 要素 (`XCUIElement` オブジェクト) から操作対象の要素を探査し、タップやスワイプといった操作を呼び出してテストを実行していきます。UI 要素はボタンやセルといった要素のカテゴリを指定した上で、そこからインデックスや Accessibility 用の識別子を指定して探索するため、操作対象の UI 要素を探すために毎回全探索が実行されていました。

Xcode 9 では、要素探索を効率的に行う API が追加されています。具体的には `XCUIElementTypeQueryProvider` プロトコルに追加された `firstMatch` プロパティです。このプロパティを使った探索はリストリス

*17 <https://developer.apple.com/videos/play/wwdc2017/409/>

*18 https://developer.apple.com/documentation/xctest/asynchronoustestsandexpectations/testingasynchronousoperationswith_

ト 5.8 のように書きます。

リスト 5.8: キャプション

```
let app = XCUIApplication()  
  
let button = app.buttons["accessibility-identifier"].firstMatch
```

`firstMatch` プロパティは最初の 1 つが見つかった時点で探索を終了するため、高速に動作すると WWDC 2017 のセッションで説明されています。^{*19} 一方で、もしその要素がただ 1 つであることを保証する場合には、従来からある `XCUIElementQuery` クラスの `element` プロパティを利用するようにドキュメントに書かれています。こちらのプロパティは該当する要素が複数あった場合には、テストを失敗としてマークしてくれます。

UI テストを書く場合には、どちらの方法がテストの意図により沿っているかを意識しながら書くと良いでしょう。

5.4.3 コマンドラインサポートの強化

ここからは、`xcodebuild` コマンドについてテスト関連の機能強化を紹介します。`xcodebuild` コマンドは、Xcode で行える様々な作業を CLI から行うためのコマンドです。Xcode 9 では、テストの並列実行をはじめとするテスト関連機能が強化されています。

iOS アプリ開発のタスクランナーである fastlane^{*20} の登場により、いまや `xcodebuild` コマンドを直接触ることは少なくなりました。しかし、fastlane 自体も内部で `xcodebuild` を利用していますし、`xcodebuild` が CI の中核を担うコマンドであることに変わりはありません。

コマンドラインからのテストの実行

まず、`xcodebuild` コマンドを使ってテストを実行する方法を説明します。

コマンドラインからテストを実行するには、Xcode プロジェクトがあるディレクトリでリスト 5.9 のようにコマンドを実行します。

リスト 5.9: ‘xcodebuild’を使ったテストの実行

```
```sh  
$ xcodebuild \
-scheme "schemeName" \
-destination "platform=iOS Simulator,name=iPhone 7,OS=11.0" \
test
```

`schemeName` というスキーム名のテストアクションに指定されているターゲットを、iOS 11 を搭載した iPhone 7 シミュレータで実行します。ちなみにスキームやテストターゲットの一覧は

<sup>\*19</sup> <https://developer.apple.com/videos/play/wwdc2017/409/>

<sup>\*20</sup> <https://github.com/fastlane/fastlane>

`xcodebuild -list`、利用可能なデバイス・シミュレータの一覧は `xcrun simctl list` で確認できます。

以降では、このコマンドをベースとした変更点や新機能を解説します。

### テストの並列実行サポート

コマンドラインからのテストで一番大きく進歩した点がテストの並列実行をサポートしたことです。「5.2.3 シミュレータの新機能」でも述べたように、Xcode 9 が iOS シミュレータの複数同時起動をサポートしたことで、効率的に動作を確認できるようになりました。同様に、テストにおいても複数のシミュレータ・実機での並列実行をサポートしたことで、テストの実行時間を飛躍的に削減できます。

テストを並列実行する方法は非常に簡単で、`-destination` オプションを複数渡すだけです。たとえば iPhone 5s, iPhone 7, iPhone 7 Plus と 3 つのシミュレータで同時にテストを行いたい場合はリスト 5.10 のようにコマンドを実行します。

リスト 5.10: キャプション

```
$ xcodebuild \
-scheme "schemeName" \
-destination "platform=iOS Simulator,name=iPhone 5s,OS=11.0" \
-destination "platform=iOS Simulator,name=iPhone 7,OS=11.0" \
-destination "platform=iOS Simulator,name=iPhone 7 Plus,OS=11.0" \
test
```

このコードではすべて iOS 11.0 のシミュレータを指定していますが、実行可能な OS バージョンであれば複数の OS バージョンを混在して指定することもできます。また、実機が接続状態にあれば"platform=iOS,id={UDID}"を指定することで、シミュレータと実機のテストも同時に実行できます。もちろん実機はワイヤレスでテストできます！

ここで気なるのが、最大いくつまで同時にテストを実行できるのか？ ということです。試しにユニットテストを 1 つだけ実行するプロジェクトを作り、筆者の環境（MacBook Pro Late 2013, メモリ 16GB, macOS High Sierra）で検証してみました。その結果、最大 15 種類のシミュレータで同時にテストを実行することができましたが、数回に一度はエラーで失敗扱いとなってしまいました。そこで新しく追加された`-maximum-concurrent-test-simulator-destinations` オプションを利用して並列数を 4 にしてみたところ、安定して成功するようになりました。どうやら同時に指定できるシミュレータの数や並列実行数は、テストの内容や Mac 本体の処理能力に依存するようです。そのため、これらの値は環境に応じてチューニングするのが良いでしょう。なお、シミュレータではなく実機でテストする場合には`-maximum-concurrent-test-device-destinations` で並列数を制御できます。

このように開発者側で気をつける部分はあるものの、Xcode 9 では並列実行によってテストの実行時間を短縮できます。

```
XXX: BLOCK_HTML: YOU SHOULD REWRITE IT
<!--
NOTE: 2017-10-01
環境: MacBook Pro(Late 2013), macOS Sierra, Xcode 9
```

同時実行数をあげすぎるとシミュレーターの起動に失敗して、起動成功したシミュレーターのテストだけが実行される。  
(1つでも起動エラーがあるとテスト全体としては FAILED になる)

ログをみるとテスト開始時にすべてのシミュレーターが起動する仕様らしく、  
テスト自体の並列数実行数ではなくて指定するシミュレーターの数が成功するかどうかを分けそう。

なので‘-maximum-concurrent-test-simulator-destinations’を指定しても、  
トータルで実行したいシミュレーター数が多いとテストに失敗する。  
逆に言うと指定したシミュレーターの数が少なければ並列数多くても成功する。

対策として‘-disable-concurrent-testing’を指定するとシミュレーターの起動が1つずつになるので  
確実にテストを実行できる（もちろん並列数1になるので並列実行の意味なくなる）

-->

```
XXX: BLOCK_HTML: YOU SHOULD REWRITE IT
<!--
NOTE: 2017-10-09
環境: MacBook Pro(Late 2013), macOS High Sierra, Xcode 9
```

macOS High Sierra にアップグレードして再度検証。  
こちらでも指定するシミュレーターの数が多いと失敗する模様。  
特に UI テストが入ると、シミュレーターの指定数を8未満にしないと高確率で失敗する。

-->

### プロビジョニングプロファイルの更新およびデバイスの自動追加オプション

実機でアプリケーションを実行する際に避けて通れないのが、Apple Developer Center へのデバイスの登録とプロビジョニングプロファイル（以下プロファイル）の準備です。従来はブラウザから手動で行っていたこれらの作業は、Xcode 8 で追加された Automatic Signing 機能によって、ほとんど意識する必要がないくらいに簡単なものになりました。しかしこマンドラインからテストを実行する場合にはこの機能はサポートされておらず、Xcode アプリケーションからこれらの作業を済ませておく必要がありました。Xcode 9 では、コマンドラインからでもこれらの作業を行ってくれるオプションが追加されています。

1つ目のオプションが`-allowProvisioningUpdates` です。このオプションを利用すると、チムプロビジョニングプロファイルを更新したうえで実機テストを実行してくれます。

このオプションの効果を説明する前に、まずはオプションを追加しない場合のログをリスト 5.11 に示します。これは Xcode 9 で必要なプロファイルをダウンロードせずにテストを実行した時のログの一部です。プロファイルが見つからないためにテストに失敗したことがメッセージからわかり

ます。また`-allowProvisioningUpdates` オプションを指定することで、Automatic Signing が利用できることも示されています。

リスト 5.11: プロファイルがない場合の実行ログ

```
```
Testing failed:
No profiles for 'me.huin.apps.sample' were found: Xcode couldn't find any iOS App Development provisioning profiles on the system.
Code signing is required for product type 'Application' in SDK 'iOS 11.0'
** TEST FAILED **
```

そこでメッセージに従って`-allowProvisioningUpdates` オプションを指定してテストを実行します。途中キーチェーンのパスワードを求めるウィンドウが2回表示されるので、どちらもログインパスワードを入力して「許可」ボタンをクリックします。また、以降の実行でパスワード入力を求められないようにするには「常に許可」ボタンをクリックします。パスワードを入力するとあとはいつも通りテストが実行され、問題がなければ`TEST SUCCEEDED` と表示されて終了します。このように`-allowProvisioningUpdates` オプションを指定することで、コマンドラインから実行した場合でもプロファイルを自動で更新してくれます。

もう一つ、実機でのテストを簡単にしてくれるオプションが`-allowProvisioningDeviceRegistration` です。前述の`-allowProvisioningUpdates` に加えてこのオプションを指定すると、未登録のデバイスを開発者アカウントに登録した上でプロファイルの更新を行ってくれます。事前に Xcode から開発者アカウントを追加しておくことと、デバイスを Mac に接続した際にコンピューターを信頼しておくことが必要ですが、デバイスの登録作業を意識することなくテストを実行できます。

これら2つのオプションは、個人で開発している場合にはあまりメリットがないかもしれません。しかし、例えば社内で独自にテスト環境を構築している場合には、デバイスの登録やプロファイルの管理と行った煩雑な作業を簡略化してくれるため、非常にうれしい新機能といえます。

言語と地域を指定したテスト実行

多言語対応しているアプリ向けのオプションも追加されました。Xcode 8までは、Xcode からアプリを実行する際の言語・地域設定は、スキーム編集画面で指定していました。Xcode 9ではテストアクションの設定画面で言語・地域設定も指定できるようになっていて、さらに`xcodetool` にも地域・言語を指定するオプションが追加されています。

追加されたオプションは2つで、言語が`-testLanguage`、地域が`-testRegion` です。それぞれ、ISO 639-1 の言語コードと ISO 3166-1 の国名コードで指定します。

テストを英語と日本語それぞれで実行する場合のコマンドをリスト 5.12 に示します。

リスト 5.12: 言語及び地域を指定したテスト

```
```sh
日本語
$ xcocdebuild \
-scheme "schemeName" \
-destination "platform=iOS Simulator,name=iPhone 7,OS=11.0" \
```

```
-testLanguage "ja"
-testRegion "JP"
test

英語
$ xcodebuild \
-scheme "schemeName" \
-destination "platform=iOS Simulator,name=iPhone 7,OS=11.0" \
-testLanguage "en"
-testRegion "US"
test
```

これにより、指定した言語・地域設定でテストを実行できます。実機でテストを実行する場合にも有効です。

注意しなければならないのは、前述の並列実行を行った場合には、すべてのシミュレータで同じ地域・言語になることです。そのため、並列実行と言語・地域指定を組み合わせたい場合は、地域・言語を入れ替えながら並列実行を複数回実行するという手順が必要になります。

## 5.5 Xcode サーバーの利用

本章の最後に、Xcode 9 で統合されたサーバー機能について解説します。

### 5.5.1 Xcode サーバーの概要

Xcode サーバーは iOS/Mac アプリケーション向けの継続的インテグレーション機能です。アプリケーションの静的解析、テスト、アーカイブ (.ipa ファイル作成) を定期的に実行でき、問題の早期発見や品質の改善に役立てられます。Xcode サーバー機能は 2013 年の WWDC で Xcode 5 とともに発表された機能です。登場以来、毎年改善が行われてきたものの、有料の macOS X Server の機能として提供されていたせいか、広く利用されている機能とはいえない状況でした。

Xcode 9 では、このサーバー機能が Xcode アプリケーション本体に統合されました。そのおかげで、Xcode 本体をインストールするだけで継続的インテグレーションを実現できるようになっています。また、これまでに紹介したテストアタッチメントやテストの並列実行サポートによって、実行時間やレポーティングも改善されています。

本項ではこのサーバー機能の使い方について解説します。

### 5.5.2 サーバーのセットアップと Bot の作成・実行

#### サーバーのセットアップ

Xcode サーバー機能を利用するためには、まずは機能を有効にする必要があります。Xcode の Preferences から「Server & Bots」タブを開き、右上のスイッチを切り替えて有効にします（図 5.27）。設定は Xcode が自動でやってくれますが、途中でテストを実行するユーザーを選ぶよう求められるので、ユーザー一覧から選びます。インテグレーションの実行中にサードパーティのツールを使いたい場合には、ここで選んだユーザーで利用できるようにインストールを行います。以上

でサーバーのセットアップは完了です。

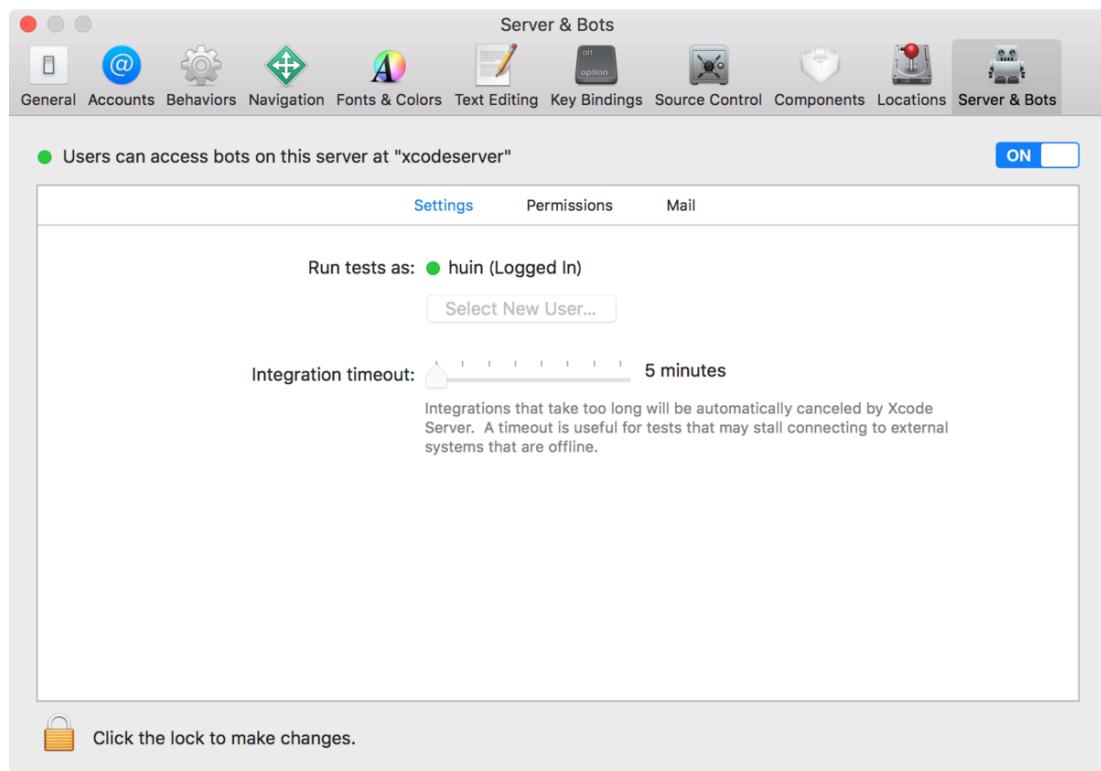


図 5.26: サーバー機能の有効化

### Bot の作成

サーバーの準備が終わったら、次は実行するインテグレーションタスクを作成します。Xcode サーバーでは、このタスクのことを「Bot」と呼んでいます。

Bot を作成するにあたって、まずプロジェクト側の準備をします。必要なことは「スキームを共有すること」と「ソースコードをリモートに置くこと」の2つです。前者は、スキームの編集画面から Shared にチェックを入れるだけです。プロジェクトファイルの下に xcshareddata フォルダが作成されるので、コミットした上でリモートにプッシュしておきます。後者に関しては、もしローカルで新規に作成したプロジェクトであれば、「リポジトリの作成と公開」を参考にしてソースコードをリモートにプッシュしておきます。すでにリモートから取得したプロジェクトであれば作業は必要ありません。

なお、以降ソースコードのホストは Github で行われていると仮定して説明をします。

プロジェクトの準備が終わったら、実際に Bot を作成します。Xcode でプロジェクトを開いた後に、「Product」メニューから「Create Bot...」を選択します。設定ウィザードが始まるので、指示に従って設定を進めていきます。いくつか設定の補足をしておきます。2ステップ目の「Configure source control for this bot」では、リモートからソースコードをチェックアウトする際の認証方法を選べます（図 5.28）。「Personal SSH Keys」を選んでユーザー個人の SSH 鍵を利用していい

のですが、セキュリティ上の懸念がある場合は「Bot-Specific SSH Keys」を利用して対象リポジトリにだけ有効な SSH 鍵を発行できます。こちらを選ぶ場合には、Github のヘルプ<sup>\*21</sup>を参考に Deploy Keys の設定を行ってください。

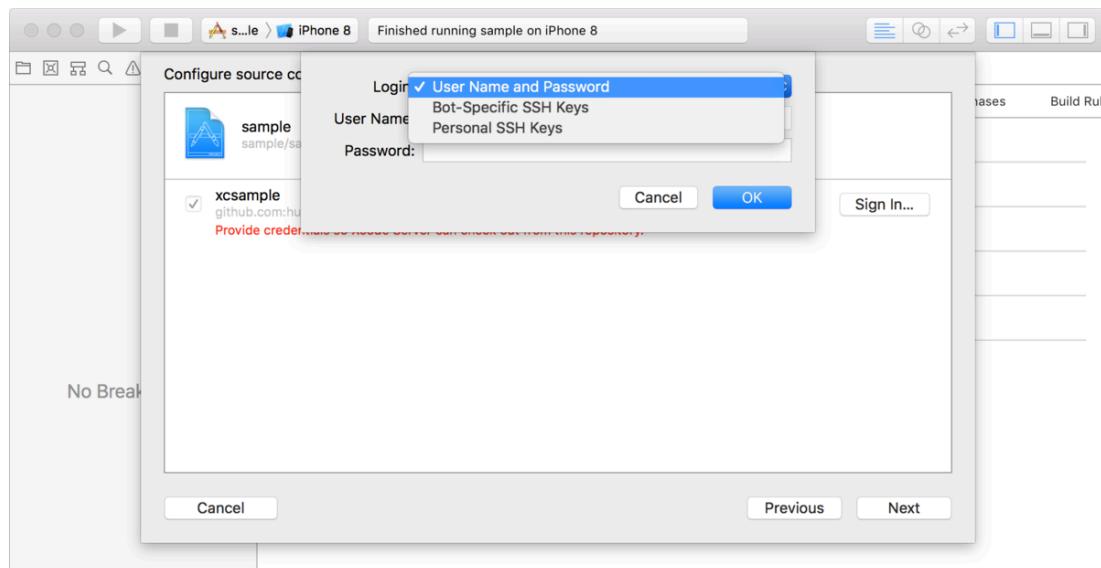


図 5.27: ソースコード取得の認証方法設定

3ステップ目では、Bot が行うアクションを選択します（図 5.29）。Xcode サーバーでは1回のインテグレーションの中で「Analytics（コードの静的解析）」、「テスト」、「アーカイブ（xcarchive の作成と ipa ファイルの作成）」の3つを実行できます。ひとまずビルドが通ることを確認するために、ここではテストだけにチェックを入れます。

<sup>\*21</sup> <https://developer.github.com/v3/guides/managing-deploy-keys/>

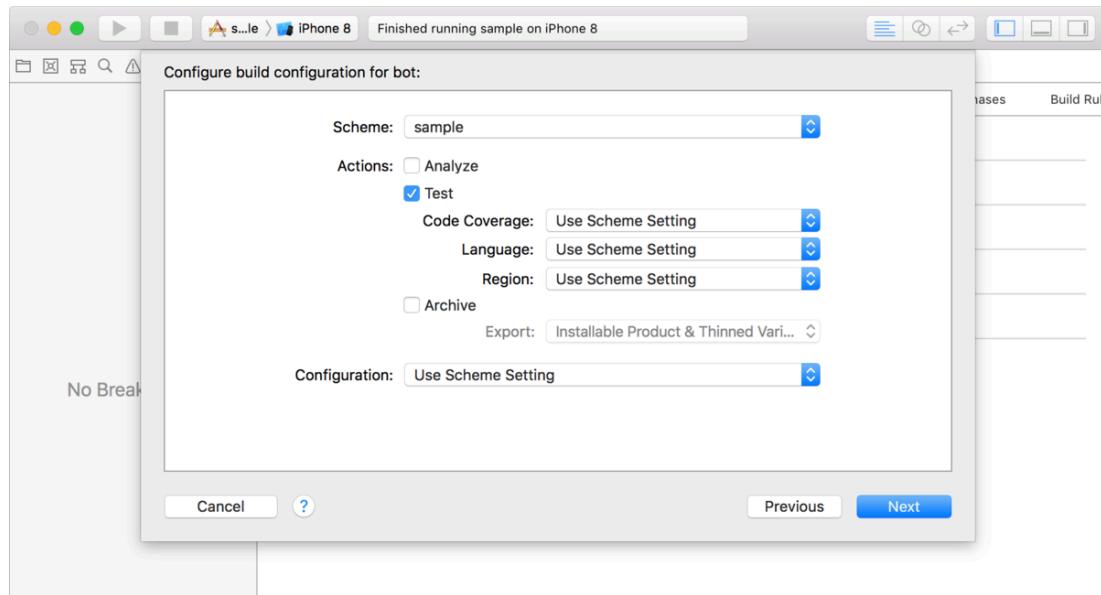


図 5.28: アクションとビルド設定の確認

4ステップ目では、実行スケジュールを設定します（図 5.30）。間隔もしくは日時を指定して定期実行する「Periodically」、ソースコードに変更があった時に検知して実行する「On Commit」、もしくは手動で実行する「Manually」の3つのスケジュール方法が選べます。

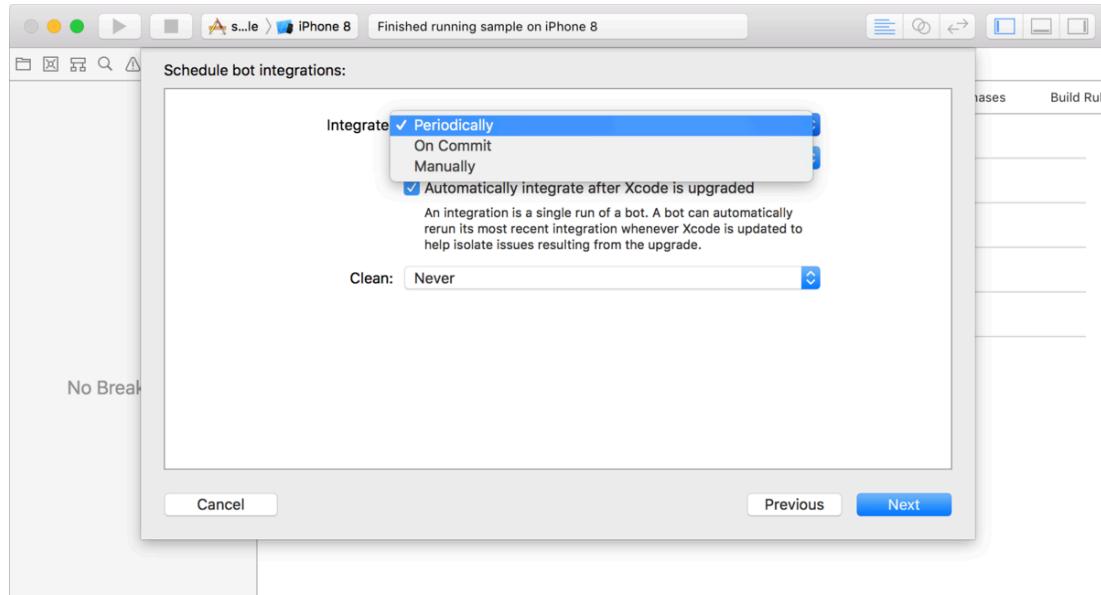


図 5.29: 実行スケジュールの設定

5ステップ目ではテストを実行するデバイス（実機あるいはシミュレータ）を選択します。すべての実機やシミュレータでテストを実行してもいいのですが、Bot が正常に実行できることを確認

したいので、ここでは3つのシミュレータのみを選択しています（図5.31）。

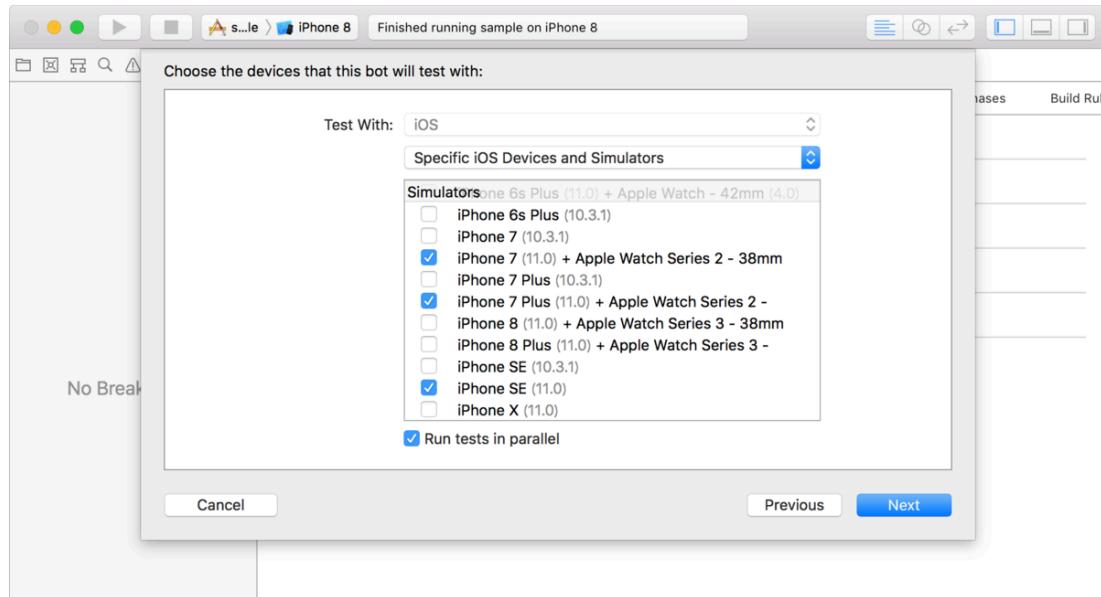


図5.30: 実行シミュレータの選択

テストデバイスの選択以降は、実機テストとアーカイブに利用するプロビジョニングプロファイルの追加、実行時引数及び環境変数の設定、トリガーの設定と続きます。これらの設定はあとから編集できるので、今回はシミュレータ上でテストだけを行う設定で作成しています。

#### インテグレーションの実行と結果の確認

Botの作成が完了すると、XcodeのレポートナビゲーターにサーバーとBotが表示されます。この状態でBotを選択して、画面右上の「Integrate」ボタンまたはコンテキストメニューからインテグレーションを実行できます。実行履歴は同じレポートナビゲーターに表示されるので、選択して詳細な結果を確認できます。図5.32では設定どおり3つのシミュレータでテストが実行され、すべてで成功していることがわかります。

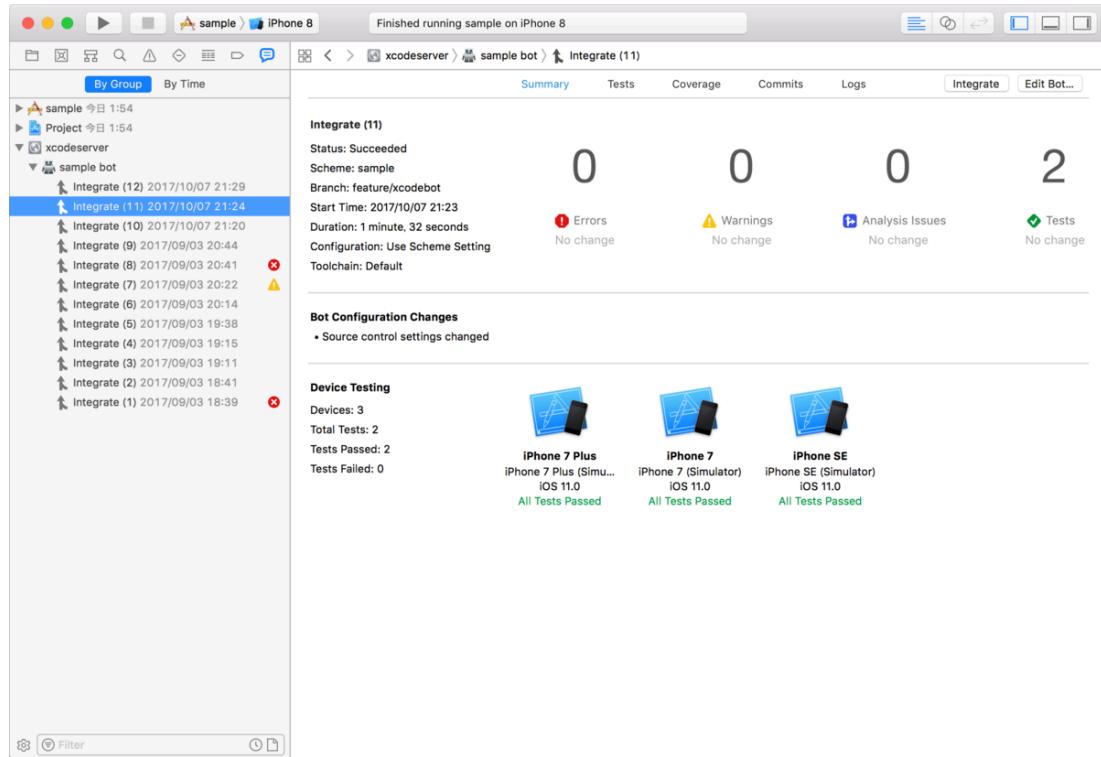


図 5.31: 実行結果の確認

図 5.32 はサマリーですが、テストタブでは、テストメソッドごとの実行結果やテストアタッチメントの確認など、より詳しい情報を確認できます。

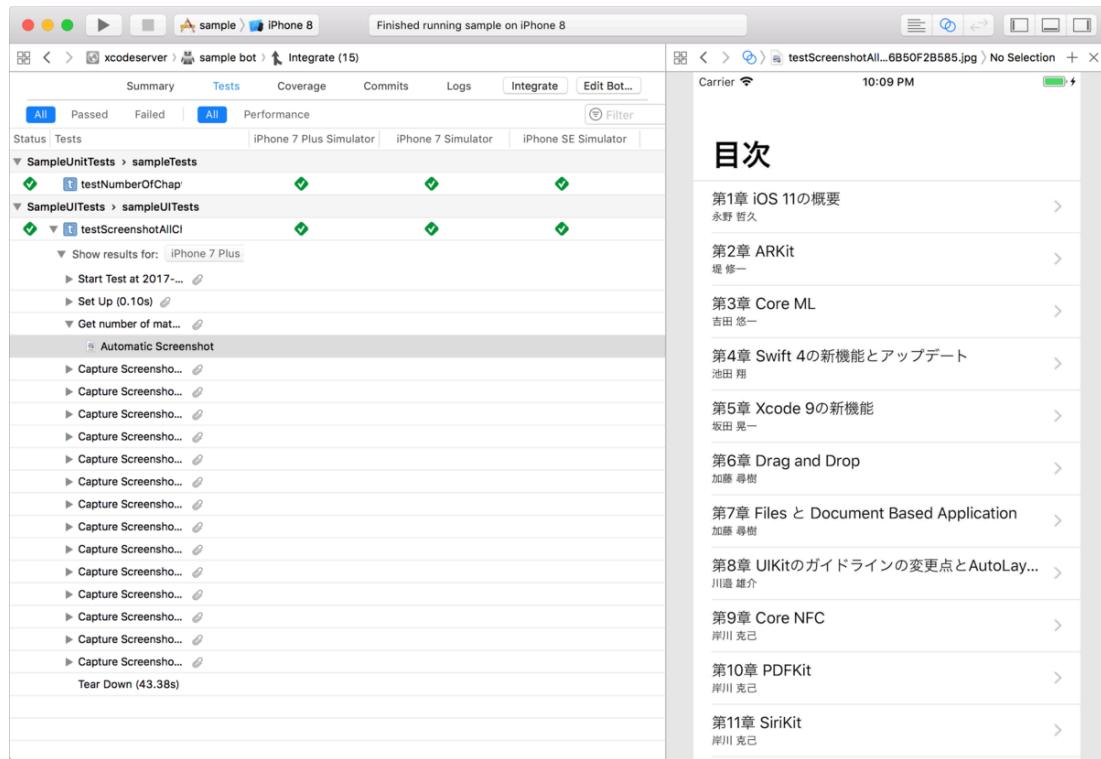


図 5.32: テストアタッチメントの確認

### 5.5.3 トリガーを利用したスクリプトの実行

ここまでにサーバーの設定から Bot の作成・実行まで、ひととおりの使い方を説明してきました。プロジェクトが外部ライブラリを利用してない場合や、手動で依存関係を解決できている場合にはいいのですが、Cocoapods<sup>\*22</sup>や Carthage<sup>\*23</sup>を利用している場合には上記の方法だとビルドに失敗します。これらのツールを使っている場合には、ビルド前にいくつか処理を実行する必要があるためです。ここでは Xcode サーバーのトリガー機能を利用して、依存性の解決や任意の処理を実行する方法を紹介します。

#### プロジェクトの準備

Cocoapods 経由で Lint ツールである SwiftLint<sup>\*24</sup>を利用するケースを考えます。また Cocoapods 自体も、システム領域ではなく Bundler<sup>\*25</sup>を利用して、リポジトリ以下にインストールしたものを利用するものとします。

この環境を実現するには、Gemfile（リスト 5.13）と Podfile（リスト 5.14）を用意した上で、リスト 5.15 のようにコマンドを実行する必要があります。SwiftLint 自体の設定は公式サイトを参

<sup>\*22</sup> <https://cocoapods.org/>

<sup>\*23</sup> <https://github.com/Carthage/Carthage>

<sup>\*24</sup> <https://github.com/realm/SwiftLint>

<sup>\*25</sup> <http://bundler.io/>

考にしてください。

設定ファイルさえ用意してしまえば難しい手順ではありません。これらのファイルをリポジトリにプッシュしたら、次は Bot の設定を編集します。

リスト 5.13: Gemfile の内容

```
"""
source "https://rubygems.org"

gem "cocoapods"
```

リスト 5.14: Podfile の内容

```
"""
platform :ios, '11.0'

target 'sample' do
 use_frameworks!
 pod 'SwiftLint', '~> 0.18.1'
end
```

リスト 5.15: Cocoapods および SwiftLint のインストール

```
'''sh
$ cd /path/to/project
$ bundle install --path .bundle
$ bundle exec pod install
```

## Bot の編集

Xcode サーバーには「トリガー」という機能があります。これはインテグレーションの前後で任意の処理を実行したり、インテグレーションで問題を検出した時にメールで通知したりするよう設定できます。今回はインテグレーション開始前に処理を実行する「Pre-Integration Script」を利用して、Cocoapods や SwiftLint のインストールを行います。

トリガーの設定は Bot の編集画面から行います。編集画面はレポートナビゲーターで Bot を選択し、ウインドウ右上の「Edit Bot...」ボタンをクリックすることで表示できます。トリガーを設定する前に、まずリポジトリ設定を変更します。Cocoapods を利用する場合、ビルトの対象がプロジェクトファイルからワークスペースファイルになるため、Repositories タブで **Replace Repositories** ボタンをクリックして、ワークスペースファイルに切り替えておきます。

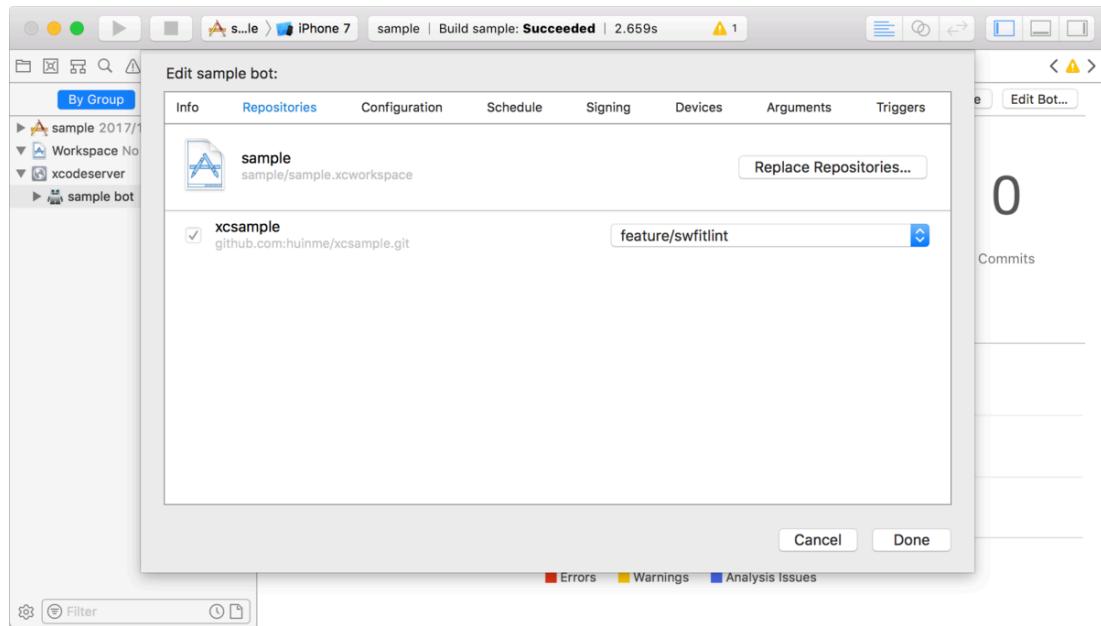


図 5.33: リポジトリ設定の更新

リポジトリ設定を更新できたら「Triggers」タブを選択し、左下の+ボタンから「Pre-Integration Script」を追加します（図 5.35）。スクリプトの内容にはリスト 5.16 を入力します。

リスト 5.16: Pre-Integration Script の内容

```
```sh
#!/bin/sh
source "$HOME/.bashrc"

cd $XCS_PRIMARY_REPO_DIR

bundle install --path .bundle
bundle exec pod install
```

スクリプトの2行目では、bash の設定ファイルを読み込んでいます。インテグレーションの実行自体はサーバーのセットアップ時に選んだユーザーで行われますが、シェルスクリプトの実行環境はユーザーがターミナルを利用する時と異なるため、手動で設定ファイルを読み込んで環境変数などの設定が必要です。bash 以外のシェルを利用している場合は、適宜設定ファイルを書き換えてください。

4行目ではカレントディレクトリを変更しています。環境変数 XCS_PRIMARY_REPO_DIR はチェックアウトしてきたリポジトリのパスを表している^{*26}ので、この位置に移動しています。

残りの2行は手動で行う場合と同様に、Bundler を利用して Cocoapods と SwiftLint をインス

^{*26} <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/xcodeguide-continuousintegration/EnvironmentVariableReference.html>

トールしています。

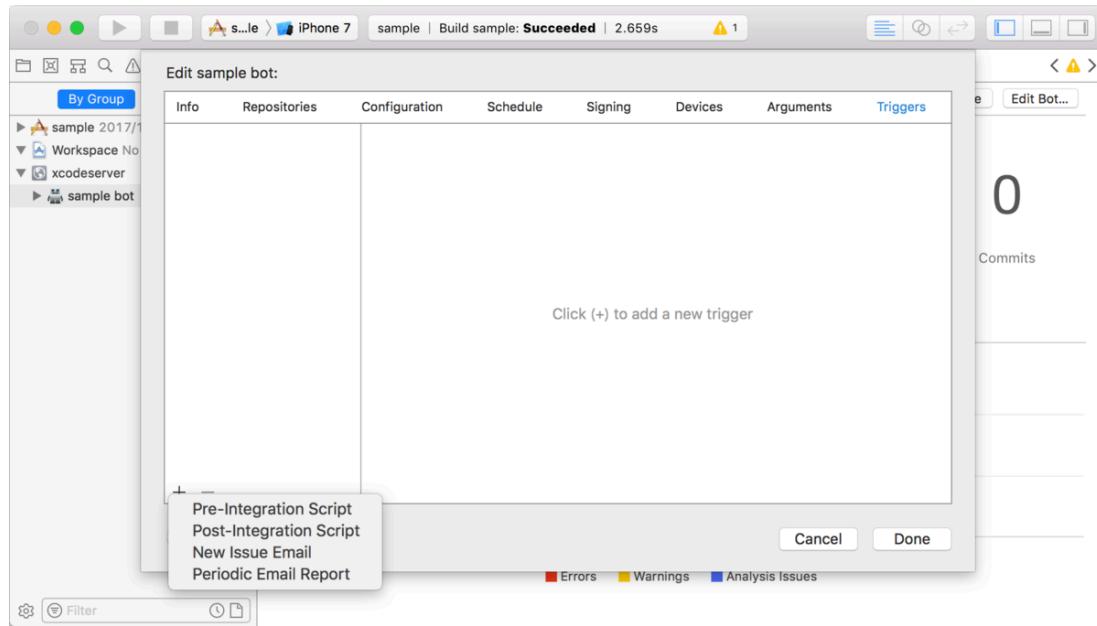


図 5.34: トリガースクリプトの追加

実行結果の確認

スクリプトを追加したら、編集を完了してインテグレーションを実行します。SwiftLint でエラー やワーニングが検出されていれば、図 5.36 のように、Summary に Issues として表示されます。もしうまく設定できずにエラーになる場合には、Logs タブからトリガーのログを表示して原因を取り除きます。

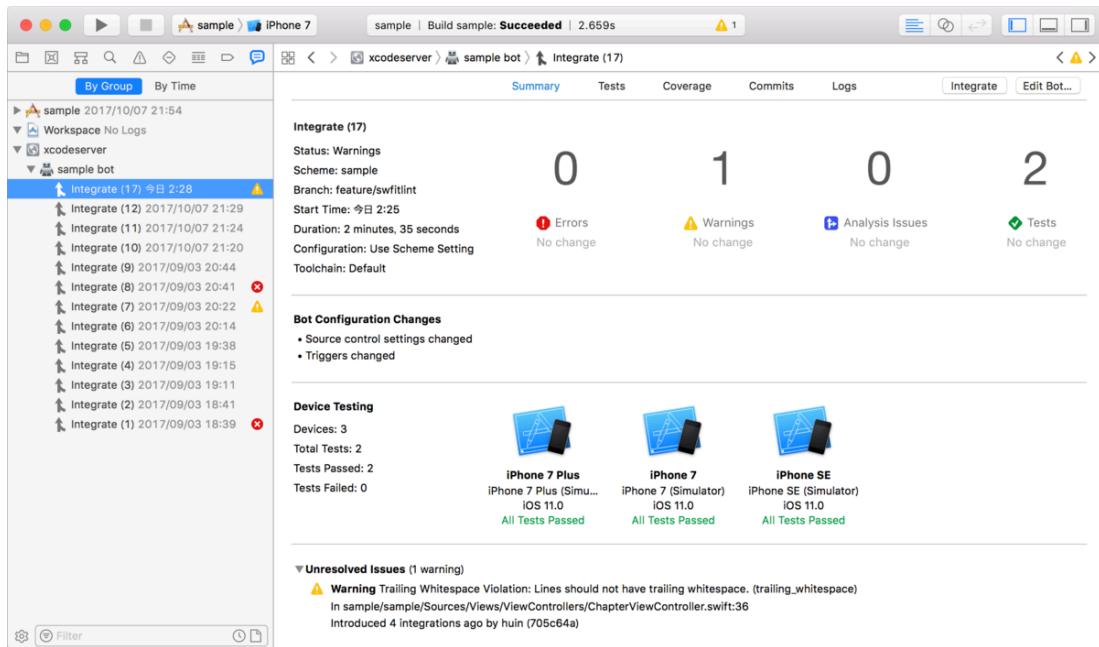


図 5.35: SwiftLint の実行結果の確認

このように、トリガーを利用することでインテグレーションの前後に任意の処理を実行できます。外部ライブラリの依存性解決だけでなく、Slackへの通知といった外部システムとの連携も実現できます。

なお、Xcode サーバーでテストを実行した場合、テストアタッチメントはリスト 5.17 のパスに保存されます。このパスと「Post-Integration Scripts」トリガーを利用すれば、Xcode サーバーから実行した時の成果物もまとめて回収できます。

リスト 5.17: テストアタッチメントの保存場所

```
"""
$XCS_DERIVED_DATA_DIR/Logs/Test/Attachments/
```

5.5.4 アーカイブと iOS デバイスへの配布

ここでは Xcode サーバーを利用してアプリをベータ配布する方法を説明します。

Xcode サーバーでは静的解析やテストの実行だけでなく、アーカイブを実行して ipa ファイルを配布することもできます。ベータ配布機能は Apple 自身が TestFlight を提供していますが、審査不要な内部テスターでは 25 人までしか配布できません。また、開発環境などを用意しているプロジェクトの場合には、iTunes Connect を経由することに抵抗のある場合もあると思います。Xcode サーバーによる配布機能は、このような場合に有効な選択肢になります。

なお、プロジェクトの用意が済んでいて Xcode からのアーカイブが可能である

ことを前提とします。

Bot の編集とアーカイブの実行

Xcode サーバーでバイナリをアーカイブするための準備として、まず Bot の編集画面で Configuration からアーカイブアクションを有効にしておきます。この時エクスポート方法は Installable Product もしくは Installable Product & Thinned Variatns を選んでおきます。詳細にエクスポート方法を設定したい場合は、「Use Custom Export Options Plist」を選択して、設定用ファイルを指定します。^{*27}

アーカイブアクションを有効にしたら、次は開発者証明書とプロビジョニングプロファイルをサーバーに追加します。Xcode の環境設定で開発者アカウントを追加していれば、Bot の編集画面で Signing タブの「Certificates & Profiles」に証明書とプロビジョニングプロファイルが表示されているはずです。ここでアーカイブに必要な項目の「Add to Server」をクリックします（図 5.37）。以上で設定は完了です。

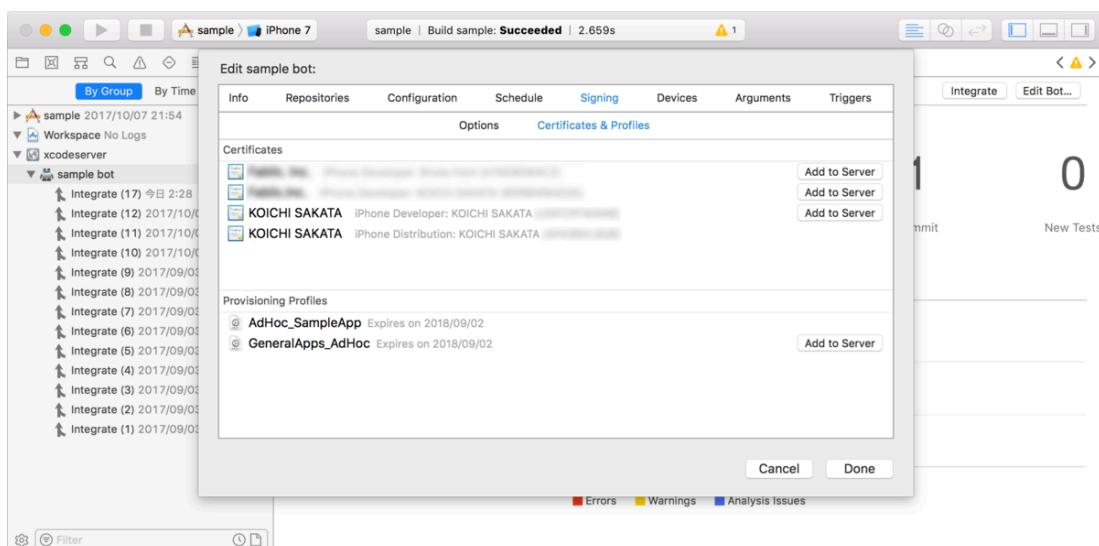


図 5.36: 証明書とプロファイルの追加

アーカイブが正常にできていれば、レポート画面に「Build Results」という項目が表示されます（図 5.38）。ipa ファイルと xcarchive ファイルの 2 つが生成され、必要なものを保存できます。また、インストール可能な実機が Xcode に接続されていれば、「Save」メニューのプルダウンから「Install on Device...」を選択し、デバイスに直接インストールも可能です。

^{*27} https://developer.apple.com/library/content/technotes/tn2339/index.html#//apple_ref/doc/uid/DTS40014588-CH1-WHATKEYSCANIPASSTOTHEEXPORTOPTIONSPLISTFLAG_

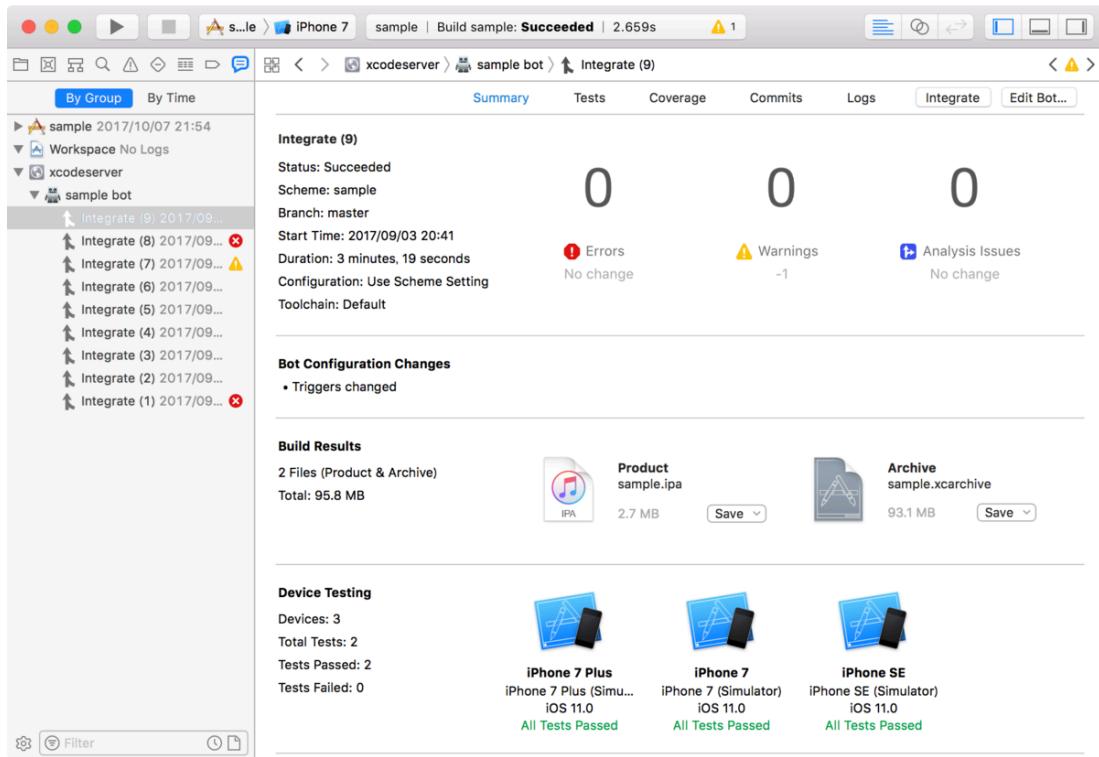


図 5.37: アーカイブ結果の確認

iOS 端末への直接インストール

前述のとおり、アーカイブの成果物を保存しておくことは可能ですが、サーバーを動かしている Mac から実機に直接インストールするのは面倒です。Xcode サーバーは Web インターフェイスも同時に提供しているため、実機からアクセスして成果物を OTA 配布できます。

まず、iOS 端末からサーバーへアクセスします。ローカルネットワーク内であればマシン名でアクセスできるはずなので、マシン名が `xcodeserver` であれば <https://xcodeserver.local/xcode> へアクセスします。この時図 5.39 のように警告が表示されますが、ここでは無視して進んでしまって問題ありません。

Web インターフェイスにアクセスすると、サーバーで稼働している Bot の一覧とインテグレーションの結果を確認できます。インストールしたい Bot をタップして詳細ページへと進みます。インストール可能な成果物がある場合は、図 5.40 のように「PROFILE」と「INSTALL」という 2 つのリンクが表示されます。まずは PROFILE のリンクをタップして、インストールに必要な構成ファイルをインストールします。プロファイルのインストール後、設定アプリで「一般」>「情報」>「証明書信頼設定」を開きます。「Xcode Server Root Certificate Authority」という項目が表示されているので、スイッチを切り替えて有効にします（図 5.41）。

ここまでできたら再度 Safari に戻り、今度は INSTALL リンクをタップします。確認のアラートが表示されるので、インストールをタップすれば完了です。

このように Xcode サーバーを利用すれば、開発版のビルドを簡単にチーム・社内に配布できます。



図 5.38: 接続に関する警告

The screenshot shows the Xcode Organizer interface. At the top, there's a status bar with "圏外 WiFi" (Cellular WiFi), the time "20:45", and battery level "52%". Below the status bar, a header bar displays a lock icon and the URL "xcodeserver.local" with a refresh button. The main content area has a back arrow labeled "Back" and the title "sample bot".

The central part of the screen shows a card for "Sample Bot". It includes:

- An icon of a stylized "A" shape.
- The name "Sample Bot".
- The timestamp "Today at 8:44 PM".
- The file size "2.6 MB".
- The operating system "iOS".
- A blue "PROFILE" button.
- A green "INSTALL" button.

Below this card is a section titled "Summary Results" containing:

Integration 9	Just now
0	2
-1	0% Coverage

At the bottom is a section titled "Contributors".

図 5.39: インテグレーションの詳細ページ

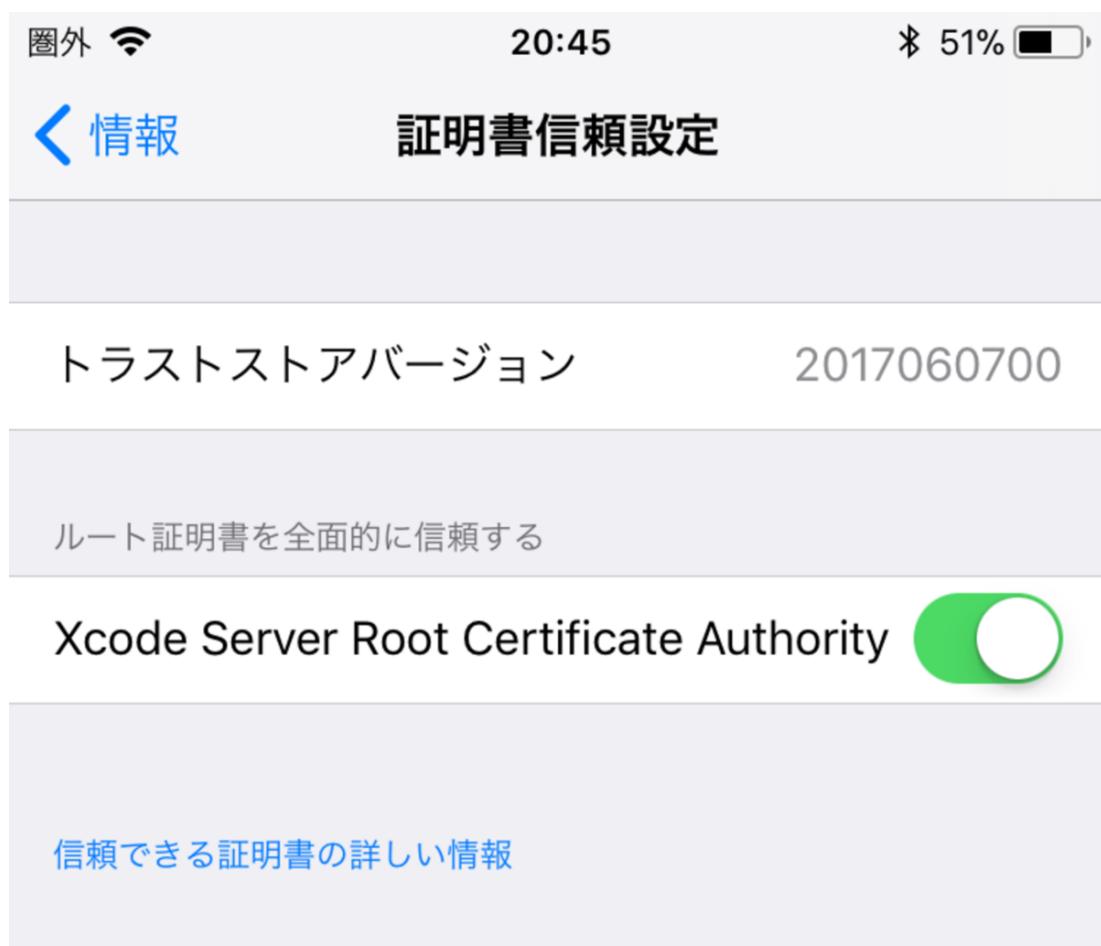


図 5.40: 証明書信頼設定の画面

5.6 まとめ

本章では、Xcode 9 および周辺の開発ツールに関する新機能を紹介しました。

開発フェーズでは、ソースコード管理機能の強化に始まり、リファクタリング機能やシミュレーターの複数実行などこれまで不便に感じていた部分が大きく改善されていることを説明しました。

デバッグフェーズでは、ワイヤレスでバグ機能やビューデバッガでのビューコントローラ参照について紹介しました。これらの新機能によって複数の端末を接続し直すことなく動作確認をしたり、レイアウトの問題なども原因を素早く調査できるようになりました。

テストフェーズでは、テストアタッチメント API や UI テストでの複数アプリケーションの操作について紹介しました。iOS アプリが毎年複雑になっていく中テストフレームワークも合わせて改善されていて、より高度なテストを行えます。

最後に、Xcode アプリケーション本体に統合されたサーバー機能について解説しました。継続的インテグレーションのサービスは多くの選択肢がありますが、Xcode サーバーを利用するとテストからベータ配布まで 1 つのマシンで実現できます。

以上のように、Xcode 9 では非常に多くの新機能・変更点が加えられています。いずれの機能も、アプリを開発する中で便利なものばかりです。ぜひこれらの機能を使いこなして、より高品質なアプリを開発してください。

第 III 部

UIKit の新機能とアップデート

第6章

Drag and Drop

本章ではドラッグ&ドロップを解説します。

ドラッグ&ドロップはアプリ間やアプリ内でデータをやり取りすることができる機能です。iPhone ではアプリ内で、iPad ではアプリ間でもドラッグ&ドロップを利用できます。

指でコンテンツをホールドして浮き上がらせ、そのまま指を動かして移動させることでドラッグできます。ドラッグした先ではコンテンツの挿入位置と、コピーあるいは移動を表すアイコンが表示され、指を離すことでコンテンツが挿入されて、ドロップが完了します。この一連のシーケンスがドラッグ&ドロップです。

ヒューマンインターフェイスガイドラインでは、ドラッグ&ドロップは「直感的」な機能^{*1}とされており、ユーザーの「この要素はドラッグできそう」「ここでドロップできそう」といった期待に応える必要があります。ドラッグ&ドロップはユーザーインターフェイス中のほとんどのコンテンツに期待されるものですので、ほとんどのアプリはドラッグ&ドロップをサポートすべきです。

UIKit のドラッグ&ドロップ API は、`UIView` にドラッグとドロップのインタラクションを加えることで実現します。ドラッグとドロップはそれぞれ別の処理であるため、どちらかだけを実装することもできます。`UITableView` と `UICollectionView` はドラッグ&ドロップを行う主要な部分となるので、特別なサポートが追加されました。`UITextView`、`UITextField`、`WKWebView`、`UIWebView` は自動的にドラッグ&ドロップがサポートされます。このうち `UITextView` と `UITextField` についてはドラッグ&ドロップの挙動をカスタマイズできます。

6.1 ドラッグ&ドロップによるデータのやり取り

ドラッグ&ドロップでは、アプリ間で非同期にデータのやり取りが行われます。具体的にはメモリ上のデータ、もしくはファイルが受け渡されることになります。受け渡しのコンテナとなるのが Foundation フレームワークの `NSItemProvider` クラスです。このクラスは iOS 8 から存在し、App Extension においてデータをやり取りする際に使われてきました。iOS 11 ではこれが大幅に強化され、ドラッグ&ドロップを実現するための役割を担うようになりました。

^{*1} <https://developer.apple.com/ios/human-interface-guidelines/interaction/drag-and-drop/>

6.1.1 NSItemProvider

ドラッグが開始されるとき、ドラッグ元のアプリでは `NSItemProvider` のインスタンスを生成します。`NSItemProvider` はいわゆる「promise」です。インスタンスを作るこの時点では、まだデータやファイルが存在する必要はありません。実際にドロップされてからはじめてデータを読み込みます。

クラウドストレージサービスを例に考えると、もしドラッグを開始する時点で完全なデータが必要であれば、事前に全てのデータをアプリ内にダウンロードしておくか、あるいはドラッグを開始した瞬間にユーザーの操作をブロックしてダウンロードすることになります。また、ユーザーが操作をキャンセルしてドロップが実行されなければ、余計なデータのダウンロードが発生します。`NSItemProvider` では、本当に必要になってからデータをダウンロードすることができるため、効率的です。

`NSItemProvider` には複数の種類のデータをひとまとめにしておく機能があります。たとえばドラッグ元が高度な機能を備えた画像編集のアプリだったとします。同じアプリ内でドロップされるなら、アプリ独自の内部的なフォーマットでやり取りすれば、メタデータも含めてデータのロスをなくすことができます。ドロップ先がその他の一般的なアプリなら、JPEG や PNG などの一般的な画像フォーマットで表現されていると都合がよいでしょう。このように、より詳細な形式から一般的な形式へとフォールバックを行う仕組みを持っています。

6.1.2 Uniform Type Identifier (UTI)

実際のデータのやり取りを見ていく前に、そもそもデータやファイルの種類を特定する仕組みについて知る必要があります。macOS や iOS ではデータの種類を表すために、「Uniform Type Identifier」、略して UTI という識別子が用いられます。

UTI は `com.apple.iwork.keynote.key` のようにリバースドメインで表現されます。ただし `public.jpeg` のように一般的なデータの種類に与えられる `public` は予約済みであり、Apple だけが定義できます。^{*2}

UTI は継承が可能で、たとえば `public.jpeg` は `public.image` を継承しており、さらに `public.image` は `public.data` を継承しています。UTI は物理的なカテゴリーと、機能的なカテゴリーに分けることができ、`public.jpeg` は物理的なカテゴリーです。物理的な UTI は最終的には基底の UTI として `public.item` を持ります。機能的なカテゴリーは、データがどのような機能を持つのかを表します。たとえば `public.image` は基底 UTI である `public.content` も継承しており、機能的な UTI でもあります。UTI はこのように多重継承が可能です。

開発者は、アプリの `Info.plist` ファイルに UTI を宣言することで独自の UTI を定義できます。

`NSItemProvider` では、UTI についていくつかのルールがあります。`NSItemProvider` に複数の種類のデータを持たせる場合、優先順位を決める必要があります。このとき、アプリ独自のフォーマットも含めて、なるべくデータのロスが少ない詳細なフォーマットを優先するように設定し、一般的なフォーマットへだんだんとフォールバックするようにするべきです。画像で例えると、

^{*2} もう1つ `dyn` も予約済みで、動的に作られる UTI に用いられます。

com.example.awesome-photo-editor.image のようにアプリ独自のフォーマットを最優先として、次に可逆圧縮の public.png、そして非可逆圧縮の public.jpeg という順番にします。

またデータを提供する際に NSItemProvider に設定する UTI は、public.image ではなく public.jpeg のように、より具体的かつ物理的な UTI を使用するべきです。データを受け取る側は、そのデータを扱うことができるのか判定する必要があり、UTI を手がかりにしてデコードします。具体的な UTI が指定されていないと、データがなんであるか判別するのが難しくなります。

6.1.3 データをやり取りする

ドラッグの元となるアプリが NSItemProvider を使ってデータを提供するにはいくつかの方法があります。単純なのは初めからデータを持っている同期処理の場合です。 NSItemProvider(object:) イニシャライザで NSItemProvider を作ります。

リスト 6.1: 同期的に ‘NSItemProvider’を作る

```
let image: UIImage = ...
let itemProvider = NSItemProvider(object: image)
```

後述しますが、NSString, NSAttributedString, NSURL, UIColor, UIImage は NSItemProviderWriting プロトコルと NSItemProviderReading プロトコルを実装しているため、read/write に特別な処理が必要なく、簡単に扱えます。

読み込む側では loadObject(ofClass:completionHandler:) メソッドを使います。

リスト 6.2: ‘NSItemProvider’から ‘UIImage’を読み込む

```
let progress = itemProvider.loadObject(ofClass: UIImage.self) { (image, error) in
    if let image = image as? UIImage {
        print(image)
    } else if let error = error {
        print(error)
    }
}
```

UIImage クラスを指定して画像として読み込むことができます。同期的に作った NSItemProvider であっても、読み込む際は非同期的であることに注意してください。また、completionHandler: (NSItemProviderReading?, Error?) -> Void は内部のキュー中で呼び出され、メインスレッドからは呼ばれません。ユーザーインターフェースを操作する場合は DispatchQueue.main.async { /* Do something */ }などを利用して、メインスレッドで実行してください。戻り値として Progress が返るので、進捗を確認できます。

6.1.4 非同期的にデータを提供する

クラウドストレージサービスのように、ドラッグ開始時にはまだメタデータしか持たず、データ本体は後から非同期的にダウンロードする場合、`registerDataRepresentation(forTypeIdentifier:visibility:loadHandler:completionHandler:)` メソッドを用いて、ダウンロードなどの処理を遅延実行します。

リスト 6.3: ‘NSItemProvider’に非同期的にデータを登録する

```
let url: URL = ...
let itemProvider = NSItemProvider()

itemProvider.registerDataRepresentation(forTypeIdentifier: kUTTypePNG as String,
                                         visibility: .all) { completionHandler in
    let urlSession = URLSession.shared
    let task = urlSession.dataTask(with: url) { (data, response, error) in
        completionHandler(data, error)
    }
    task.resume()
    return task.progress
}
```

`loadHandler: @escaping (completionHandler: (Data?, Error?) -> Void) -> Progress?` でデータの取得を行い、`completionHandler` を呼び出します。また `Progress?` を返すことができます。

読み込みに時間がかかるのであれば、`Progress` を返すことで進捗をユーザーに伝えることができます。加えて、`Progress` の `cancellationHandler` を設定するなどして、`Progress` の `cancel()` メソッドによるキャンセル操作にも対応するとよいでしょう。

`URLSession` が通信を行うときに返す `URLSessionTask` は iOS 11 から `ProgressReporting` プロトコルに準拠するようになりました。`progress` プロパティから `Progress` オブジェクトを得ることができます、キャンセル処理にも対応しています。

表 6.1 に示す通り、`visibility: NSItemProviderRepresentationVisibility` に設定した値で、ドロップされる先のアプリを限定できます。特に制限のない `.all` のほか、自分のプロセスのみに限定する `.ownProcess`、同じチーム識別子を持つものに限定する `.team`、macOS にのみアプリグループで限定する `.group` があります。外部のアプリには渡したくないデータがある場合などに利用できます。

表 6.1: `NSItemProviderRepresentationVisibility` の値によるドロップ先の制限

NSItemProviderRepresentationVisibility	ドロップ先の範囲
<code>.all</code>	すべてのアプリ
<code>.ownProcess</code>	ドラッグ元と同一のアプリ
<code>.team</code>	チーム識別子が同一のアプリ
<code>.group</code>	同一のアプリグループ (macOS のみ)

読み込む際には `loadObject(ofClass:completionHandler:)` メソッドで読み込みに使うクラスを指定する他にも、`loadDataRepresentation(forTypeIdentifier:completionHandler:)` メソッドで UTI を指定する方法があります。

リスト 6.4: ‘NSItemProvider’から UTI を指定してデータを読み込む s

```
let progress = itemProvider.loadDataRepresentation(
    forTypeIdentifier: kUTTypeImage as String) { (data, error) in
    if let image = data.flatMap(UIImage.init(data:)) {
        print(image)
    } else if let error = error {
        print(error)
    }
}
```

戻り値の `Progress` は `loadHandler` で返したものです。

6.1.5 データやファイルでの受け渡し

`NSItemProvider` にデータを登録する方法には、`registerDataRepresentation(forTypeIdentifier:visibility:completionHandler:)` メソッドを用いて `Data` として登録する以外にも、`registerFileRepresentation(forTypeIdentifier:fileOptions:visibility:completionHandler:)` メソッドでファイルシステム上のファイルを指す URL を登録することもできます。

また `NSItemProvider(object:)` イニシャライザと同様に `registerObject(_:visibility:completionHandler:)` や `registerObject(ofClass:visibility:loadHandler:)` でもオブジェクトを登録できます。

`NSItemProvider` からデータを読み込む際は、`Data` で受け取る `loadDataRepresentation(forTypeIdentifier:completionHandler:)` ファイルシステム上の一時ファイルとしてその URL を受け取る `loadFileRepresentation(forTypeIdentifier:completionHandler:)` オブジェクトとして受け取る `loadObject(ofClass:completionHandler:)` が使えます。

これらに加えて、ファイルのコピーを受け渡すのではなく、サンドボックス内のファイルを直接、他のアプリに参照させることができます。 `registerFileRepresentation(forTypeIdentifier:fileOptions:visibility:completionHandler:)` の `fileOptions:` `NSItemProviderFileOptions` パラメータに `.openInPlace` を与えることで、他のアプリに直接参照させることができます。

受け取る側は `loadInPlaceFileRepresentation(forTypeIdentifier:completionHandler:)` を利用します。 `completionHandler: @escaping (URL?, Bool, Error?) -> Void` の第二引数の `isInPlace: Bool` は、直接アクセスできたかどうかを示します。またこの方法で渡されたファイルを変更することもできます。

このようにデータをやり取りする際には、データを渡す側も受け取る側も、`Data` や `URL`、オブジェクトなど、複数の選択肢があります。しかし `NSItemProvider` が内部的に変換してくれるので、双方で揃える必要はありません。必要に応じてファイルからデータが読み取られたり、一時ファイルに保存されたりします。

6.1.6 複数のデータ形式を扱う

`NSItemProvider` ヘデータを登録するメソッドを何度も呼び出すことで、複数の種類のデータを登録できます。読み込む際には UTI などの条件に合致する、最も先に登録されたものが利用されます。登録されているデータの種類は `registeredTypeIdentifiers(fileOptions:)` で取得できるほか、`hasItemConforming(toTypeIdentifier:fileOptions:)` や `canLoadObject(ofClass:)` などのメソッドで求めるデータが登録されているか確認できます。

リスト 6.5: ‘`NSItemProvider`’に複数の形式でデータを登録する

```
let image: UIImage = ...
let itemProvider = NSItemProvider()

itemProvider.registerDataRepresentation(
    forTypeIdentifier: kUTTypePNG as String, visibility: .all) { completionHandler in
    completionHandler(UIImagePNGRepresentation(image), nil)
    return nil
}

itemProvider.registerDataRepresentation(
    forTypeIdentifier: kUTTypeJPEG as String, visibility: .all) { completionHandler in
    completionHandler(UIImageJPEGRepresentation(image, 1), nil)
    return nil
}
```

このように複数回登録する以外にも、`NSItemProviderWriting` や `NSItemProviderReading` を実装したクラスを作ることで同様の処理が行えます。

6.1.7 `NSItemProvider` に対応したクラスを作る

`NSString`、`NSAttributedString`、`NSURL` など `NSItemProviderWriting` プロトコルと `NSItemProviderReading` プロトコルを実装しているクラスが、ごく簡単にやり取りできることは先に示しました。ここでは、この 2 つのプロトコルの実装方法を解説します。

`NSItemProviderWriting`

`NSItemProvider` に登録できるクラスは、`NSObject` を継承して、`NSItemProviderWriting` プロトコルを実装している必要があります。

最初に、`writableTypeIdentifiersForItemProvider: [String]` プロパティで UTI の配列を返すことで、そのクラスがどのようなデータの種類を提供するのかを宣言します。

リスト 6.6: ‘`NSItemProviderWriting`’プロトコルでの UTI の宣言

```
protocol NSItemProviderWriting {
    // 必須
    public static var writableTypeIdentifiersForItemProvider: [String] { get }
```

```
// オプション
optional public var writableTypeIdentifiersForItemProvider: [String] { get }
...
}
```

このプロパティは同名で static プロパティとインスタンスプロパティがあり、そのうち static プロパティは必須です。static プロパティでは、そのクラスが必ず提供できるデータの種類だけを返し、インスタンスのプロパティではインスタンス毎に詳細なデータの種類を返します。たとえば URL では、static プロパティだと `["public.url"]` を返しますが、インスタンスプロパティでは URI のスキームが `file://`なら `["public.file-url"]` を返す、といった使い分けができます。

UTI の配列 `writableTypeIdentifiersForItemProvider` では、データのロスの最も少ない形式を先頭にして、順に一般的になるようにデータ形式を並べていきます。たとえば太字や斜体などを含むようなリッチテキストであれば、リッチテキストのほか、互換性のためにプレーンテキストでもデータを提供できます。この場合は `writableTypeIdentifiersForItemProvider` を `["public.rtf", "public.utf8-plain-text"]` としてリッチテキストを先にします。可能な限り先頭の形式が利用されるので、結果として正確なデータが受け渡されやすくなります。

次に `loadData(withTypeIdentifier:forItemProviderCompletionHandler:)` を実装する必要があります。先に宣言した UTI の配列のうち実際に利用されるものが第一引数で渡されるので、実際のデータを取得して第二引数のクロージャに渡します。

`itemProviderVisibilityForRepresentation(withTypeIdentifier:)` という同名のクラスメソッドとインスタンスメソッドでは、UTI ごとに `NSItemProviderRepresentationVisibility` を決めることができます。

NSItemProviderReading

`NSItemProvider` から読み込まれるクラスを作るには、`NSObject` を継承して、`NSItemProviderReading` プロトコルを実装します。

static プロパティの `readableTypeIdentifiersForItemProvider: [String]` を実装し、データのロスの少ないデータ表現から順に、クラスが対応する UTI を返します。次に static メソッドの `object(withItemProviderData:typeIdentifier:)` を実装し、与えられた Data からクラスのインスタンスを作ります。

以上の実装で、`NSString`, `NSAttributedString`, `NSURL`, `UIColor`, `UIImage` のようにオブジェクトのまま読み書きできるようになります。

6.1.8 ペーストボードで `NSItemProvider` を利用する

ドラッグ&ドロップとコピー&ペーストは、アプリ間でのデータのやり取りに使われるという意味において、よく似ています。iOS 11 では `UIPasteboard` が拡張され、`NSItemProvider` に対応しました。これによってコピー&ペーストでも `NSItemProvider` の仕組みを使えるようになりました。

UIPasteboard に追加された `setItemProviders(_:localOnly:expirationDate:)` メソッドでは、`NSItemProvider` の配列をペーストボードに設定することで、ペーストボードにデータをコピーできます。また `setObjects(_:localOnly:expirationDate:)` メソッドでは `NSItemProviderWriting` プロトコルを実装したクラスのオブジェクトをセットすることでコピーできます。セットされたデータを得るには `itemProviders: [NSItemProvider]` プロパティを利用します。これらのメソッドで `NSItemProvider` を利用すれば、ペーストされるまで実データの取得や生成を遅らせることができます。

ペーストされる側では `UIPasteConfigurationSupporting` プロトコルにより、`pasteConfiguration: UIPasteConfiguration?` プロパティでペーストできるデータの種類を宣言します。`UIPasteConfigurationSupporting` は `UIResponder` クラスで実装されており、これを継承している `UIView` や `UIViewController` はペーストに対応できます。

`UIPasteConfiguration` クラスは、受け入れることが可能なデータの種類を UTI や `NSItemProviderReading` プロトコルを実装したクラスとして設定可能です。イニシャライザで設定することも、後から追加することもできます。

`UIPasteConfigurationSupporting` プロトコルには必須の `pasteConfiguration: UIPasteConfiguration?` プロパティのほか、`canPaste(_:)` メソッドと `paste(itemProviders:)` メソッドがあり、これを実装することで実際にペーストを行います。

リスト 6.7: ‘UIPasteConfigurationSupporting’によるペーストのサポート

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        pasteConfiguration = UIPasteConfiguration(forAccepting: UIImage.self)
    }

    override func canPaste(_ itemProviders: [NSItemProvider]) -> Bool {
        return itemProviders.filter(
            { $0.canLoadObject(ofClass: UIImage.self) }).count > 0
    }

    override func paste(itemProviders: [NSItemProvider]) {
        for itemProvider in itemProviders
            where itemProvider.canLoadObject(ofClass: UIImage.self) {
                // ここでペーストの処理を行う
            }
    }
}
```

iOS 10までは、ペーストをサポートするために、`UIResponder` クラスの `canPerformAction(_:withSender:)` メソッドを実装してペーストが可能かどうかを返し、`UIResponderStandardEditActions` プロトコルの `paste(_:)` メソッドを実装することで対応していました。iOS 11からは `UIPasteConfigurationSupporting` プロトコルを実装した `UIView` や `UIViewController` で、`UIPasteConfiguration` を設定し、`canPaste(_:)` メソッドと `paste(itemProviders:)` メソッド

ドを実装することで、ペーストをサポートできます。

このようにしてサポートされるペースト機能は、`UIMenuController` によって表示される「ペースト」メニュー や、キーボードショートカットの「ペースト (Command+V)」などから利用されます。そして後述するドロップからも、このペースト機能が利用されることがあります。

6.2 ドラッグ

ドラッグに対応したアプリでドラッグ可能なアイテムを長押しすると、対象が浮き上がりドラッグ状態になります。この浮き上がることを「リフト」と呼びます。リフト状態で指を動かし適当なところで離すとドロップが実行されます。

アプリをドラッグに対応させるには、対応させる `UIView` に `UIDragInteraction` を追加します。`UIDragInteraction` クラスは `UIInteraction` プロトコルに準拠したクラスの1つです。`UIDragInteraction` には対応する `UIDragInteractionDelegate` があり、ドラッグに関連する多くの処理はこのデリゲートを介して行います。

典型的には `UIViewController` の `viewDidLoad()` メソッドで `UIDragInteraction` を設定します。

リスト 6.8: View に対する ‘`UIDragInteraction`’ の追加でドラッグをサポートする

```
let dragInteraction = UIDragInteraction(delegate: self)
view.addInteraction(dragInteraction)
view.isInteractionEnabled = true
```

`UIDragInteraction` のインスタンスを作るには、イニシャライザに `UIDragInteractionDelegate` を渡す必要があります。ここでは `UIViewController` が実装するものとします。`UIView` の `addInteraction(_:)` メソッドで `UIDragInteraction` インスタンスを追加します。ドラッグ対象の `view` は `isInteractionEnabled: Bool` プロパティが `true` でなければドラッグのジェスチャーに反応しないので、注意が必要です。

`UIDragInteractionDelegate` では少なくとも、ドラッグが開始される時に呼び出される `dragInteraction(_:itemsForBeginning:)` メソッドを実装する必要があります。このメソッドには `UIDragInteraction` と `UIDragSession` が引数として与えられ、ドラッグされるアイテムを表す `UIDragItem` の配列を返します。

リスト 6.9: ドラッグ開始時のアイテムの設定

```
extension ViewController: UIDragInteractionDelegate {

    func dragInteraction(_ interaction: UIDragInteraction,
        itemsForBeginning session: UIDragSession) -> [UIDragItem] {
        // ここでドラッグされた要素に応じてデータを与える
        let itemProvider: NSItemProvider = ...
        return [UIDragItem(itemProvider: itemProvider)]
    }
}
```

UIDragSession プロトコルは UIDragDropSession プロトコルを継承しており、1つのドラッグ&ドロップのセッションを表しています。ドラッグ&ドロップでは、マルチタッチによって複数のドラッグが同時に存在することがあります。その1つひとつをセッションという単位で表しています。

UIDragItem はドラッグされるアイテムを表すクラスで、NSItemProvider を用いて初期化できます。

ドラッグの最小限の実装はこれだけです。これらをひとまとめにした UIViewController の実装を示します。ここでは Storyboard を使って複数の UIImageView が作られているとします。

リスト 6.10: ドラッグをサポートする最小限の実装

```
class ViewController: UIViewController {

    @IBOutlet var imageViews: [UIImageView]!
    let images: [UIImage] = ...

    private func loadImages() {
        for (imageView, image) in zip(imageViews, images) {
            imageView.image = image
        }
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        loadImages()

        for imageView in imageViews {
            let dragInteraction = UIDragInteraction(delegate: self)
            imageView.addInteraction(dragInteraction)
            imageView.isUserInteractionEnabled = true
        }
    }

}

extension ViewController: UIDragInteractionDelegate {

    func dragInteraction(_ interaction: UIDragInteraction,
                        itemsForBeginning session: UIDragSession) -> [UIDragItem] {
        guard let imageView = interaction.view as? UIImageView else {
            return []
        }
        guard let image = imageView.image else {
            return []
        }

        let itemProvider = NSItemProvider(object: image)
        return [UIDragItem(itemProvider: itemProvider)]
    }

}
```

```
}
```

複数の `UIDragInteraction` で同じデリゲートを共有できます。`dragInteraction(_:itemsForBeginning:)` メソッドでは、`UIDragInteraction` の `view: UIView?` プロパティに関連づけられた `view` を取得できます。

もしドラッグを開始できるようなデータがない場合は空の配列を返せます。このときドラッグは発生しません。

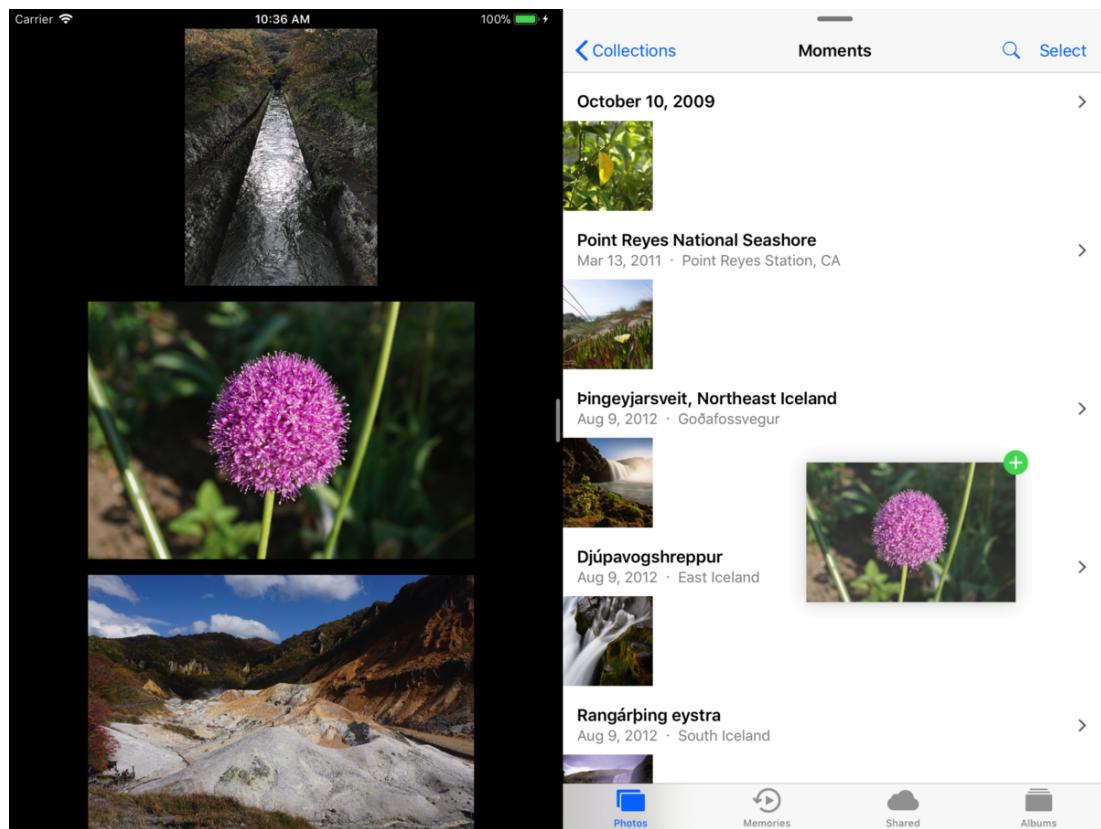


図 6.1: ドラッグを開始できる

6.2.1 ドラッグのプレビュー

ドラッグ中はドラッグしているデータを表すプレビューが表示されます。デフォルトでは `UIDragInteraction` を追加した `view` を元にプレビューが作られます。これで十分なこともありますが、なるべくドラッグ中のデータとプレビューを合わせるために、カスタマイズした方がよい場合も多いでしょう。

リフト中のプレビューをカスタマイズするには `UIDragInteractionDelegate` の `dragInteraction(_:previewForLi` メソッドを実装します。このメソッドで `UITargetedDragPreview` オブジェクト返すことで、リフ

トする時のプレビューをカスタマイズできます。

`UITargetedDragPreview` はドラッグ中のプレビューを表すクラスです。プレビューには任意の `UIView` オブジェクトを設定できます。さらに `UIDragPreviewParameters` クラスを使って、プレビューの表示領域や背景色を設定できます。`UIDragPreviewTarget` クラスを用いることで、プレビューの表示位置を指定できます。

リスト 6.11: リフト中のプレビューのカスタマイズ

```
extension ViewController: UIDragInteractionDelegate {

    func dragInteraction(_ interaction: UIDragInteraction,
                         previewForLifting item:UIDragItem,
                         session: UIDragSession) -> UITargetedDragPreview? {
        guard let imageView = interaction.view as? UIImageView else {
            return nil
        }
        guard let image = imageView.image else {
            return nil
        }

        let preview = UIImageView(image: image)
        // 'AVMakeRect'を使うとアスペクト比を保ったまま任意の'CGRect'に収めることができる
        preview.frame.size = AVMakeRect(aspectRatio: image.size,
                                         insideRect: imageView.bounds).size

        let parameters = UIDragPreviewParameters()

        let center = imageView.convert(imageView.center, from: imageView.superview)
        let target = UIDragPreviewTarget(container: imageView, center: center)

        return UITargetedDragPreview(view: preview,
                                     parameters: parameters,
                                     target: target)
    }
}
```

プレビューでは全く新たな `view` を作ることも、すでに存在している `view` そのものを使うこともできます。すでに存在している `view` を使うときは、`UITargetedDragPreview` のパラメータから `target` 引数を省略できます。`target` 引数を省略すると、`view` 引数に与えた `UIImageView` の `Superview` が `UIDragPreviewTarget` として利用されます。

ドラッグがキャンセルされた時のプレビューのためには `UIDragInteractionDelegate` の `dragInteraction(_:previewForCancelling:withDefault:)` メソッドを実装します。新しく `UITargetedDragPreview` を作って返すこともできますが、`defaultPreview` にリフト時のプレビューが渡ってきます。これをを利用して、`UITargetedDragPreview` の `retargetedPreview(with:)` メソッドで新しい `target` を設定して、キャンセルされた時のプレビューとして再利用できます。

リスト 6.12: ドラッグをキャンセルした時のプレビューのカスタマイズ

```
extension ViewController: UIDragInteractionDelegate {  
  
    func dragInteraction(_ interaction: UIDragInteraction,  
        previewForCancelled item: UIDragItem,  
        withDefault defaultPreview: UITargetedDragPreview) -> UITargetedDragPreview? {  
        guard let imageView = interaction.view as? UIImageView else {  
            return nil  
        }  
  
        let center = imageView.convert(imageView.center, from: imageView.superview)  
        let target = UIDragPreviewTarget(container: imageView, center: center)  
  
        return defaultPreview.retargetedPreview(with: target)  
    }  
}
```

プレビューを実装すると、ドラッグ開始時には要素の位置から浮き上がり、キャンセル時には元の位置に戻っていくように見せることができます。アニメーションはiOSによって管理されているので、プレビューとなるviewを作ることと、要素の位置を指定することで、プレビューの実装は十分です。

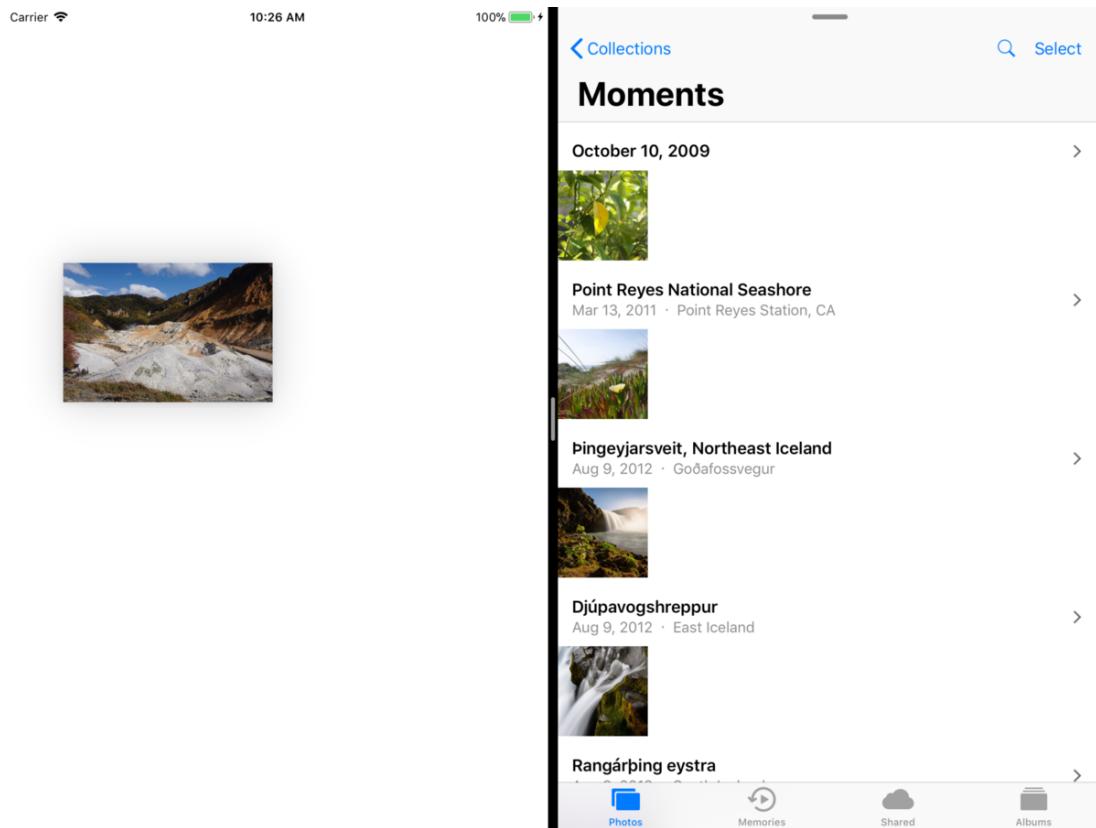


図 6.2: ドラッグ開始時にプレビューが浮き上がる

プレビューはドラッグの間、少し小さなサイズで表示されます。プレビューが画面の広い範囲を覆ってしまうと、ユーザーがドロップしたい場所を見つけられなくなってしまうことから、iOS によってサイズが調節されます。しかし UITableView で要素の並べ替えに使うような場合は、サイズが変わらない方が自然です。UIDragInteractionDelegate の dragInteraction(_:prefersFullSizePreviewsFor:) メソッドで true を返すことで、同じアプリの中でドラッグしている間は元のサイズを維持できます。

ドラッグ中のプレビューを変更するには、UIDragItem の previewProvider: ((_) -> UIDragPreview?)? プロパティに UIDragPreview オブジェクトを返すクロージャを設定します。previewProvider プロパティが nil ならリフト中と同じデフォルトのプレビューが利用されます。

6.2.2 ドラッグのアニメーション

ドラッグのアニメーションに合わせて、他の部分にもアニメーションを加えることもできます。アニメーションを付け加えるには UIDragInteractionDelegate の dragInteraction(_:willAnimateLiftWith:session:) を実装します。このメソッドに引数として渡される UIDragAnimating オブジェクトを利用し、addAnimations(_:) メソッドでリフトの開始に合わせたアニメーションを追加し、addCompletion(_:) でリフトが終了するのに合わせたアニメーションを追加します。addCompletion(_:) のクロージャには UIViewAnimatingPosition が渡され、この値が.start

ならリフトはキャンセルされ、.end ならリフトが正常に行われたことを示します。

リスト 6.13: ドラッグのアニメーションに合わせて他の要素もアニメーションさせる

```
extension ViewController: UIDragInteractionDelegate {

    func dragInteraction(_ interaction: UIDragInteraction,
        willAnimateLiftWith animator: UIDragAnimating, session: UIDragSession) {
        guard let imageView = interaction.view as? UIImageView else {
            return
        }
        animator.addAnimations {
            imageView.alpha = 0.8
        }
        animator.addCompletion { position in
            switch position {
            case .start, .end:
                imageView.alpha = 1
            case .current:
                break
            }
        }
    }
}
```

同様に dragInteraction(_:item:willAnimateCancelWith:) メソッドを実装することで、ドラッグがキャンセルされた場合のアニメーションを設定できます。

6.2.3 複数アイテムのドラッグ

ドラッグ&ドロップでは、複数のアイテムを同時にドラッグできます。これをサポートするには UIDragInteractionDelegate の dragInteraction(_:itemsForAddingTo:withTouchAt:) メソッドを実装します。ドラッグに追加できるアイテムに制約を設けることもできます。

リスト 6.14: ドラッグセッションへのアイテムの追加

```
extension ViewController: UIDragInteractionDelegate {

    func dragInteraction(_ interaction: UIDragInteraction,
        itemsForBeginning session: UIDragSession) -> [UIDragItem] {
        guard let imageView = interaction.view as? UIImageView else {
            return []
        }
        guard let image = imageView.image else {
            return []
        }

        let itemProvider = NSItemProvider(object: image)

        let dragItem = UIDragItem(itemProvider: itemProvider)
        dragItem.localObject = image
    }
}
```

```
        return [dragItem]
    }

func dragInteraction(_ interaction: UIDragInteraction,
                     itemsForAddingTo session: UIDragSession,
                     withTouchAt point: CGPoint) -> [UIDragItem] {
    guard let imageView = interaction.view as? UIImageView else {
        return []
    }
    guard let image = imageView.image else {
        return []
    }

    for item in session.items {
        // ドラッグセッションに異なる種類のアイテムが含まれていたら追加しない
        guard item.itemProvider.hasItemConformingToTypeIdentifier(
            kUTTypeImage as String) else {
            return []
        }
        // ドラッグセッションにすでに含まれている時は追加しない
        guard (item.localObject as? UIImage) != image else {
            return []
        }
    }

    let itemProvider = NSItemProvider(object: image)

    let dragItem = UIDragItem(itemProvider: itemProvider)
    dragItem.localObject = image

    return [dragItem]
}
}
```

複数アイテムのドラッグでは、デフォルトでは全く同じアイテムを複数ドラッグできます。例えば、同じ写真を何枚もひとつのメールに添付したい、ということはないでしょう。そこで同一のアイテムが複数回ドラッグセッションに追加されることがないように、アイテムを追加する時に取り除きます。UIDragItem の localObject: Any? プロパティに、ドラッグされているアイテムを示すオブジェクトを設定しておけば、すでに追加されたアイテムと同一のアイテムかどうかを簡単に調べることができます。

またドラッグに様々な種類のデータが混ざらないように、NSItemProvider の hasItemConformingToTypeIdentifier メソッドを使ってすでにドラッグに含まれるデータの種類を確かめることもできます。

このほか、同時に複数のドラッグのセッションが存在している場合に、そのうちどれにアイテムを追加するかを決めるための dragInteraction(_:sessionForAddingItems:withTouchAt:) メソッドがあります。

6.2.4 データの移動

ドラッグ＆ドロップでは、基本的にデータがコピーされます。しかし同一のアプリ内では、コピーではなくデータを移動することもできます。そのためにはドラッグとドロップの双方の処理が、データの移動をサポートする必要があります。

ドラッグ元がムーブをサポートするためには、`UIDragInteractionDelegate` の `dragInteraction(_:sessionAllowsMove:)メソッドで true を返します。ドロップ先では、後述しますが UIDropInteractionDelegate の dropInteraction(_:sessionDidUpdate:) メソッドで返す UIDropProposal に UIDropOperation.move を設定します。`

6.3 ドロップ

アプリがドロップをサポートするためには追加の実装が必要です。`UIPasteConfigurationSupporting` によってペーストに対応している場合は、自動的にドロップにも対応します。ドロップについてさらにカスタマイズを行うには `UIDropInteraction` を用います。

6.3.1 ペースト

`NSItemProvider` がペーストボードでも利用できることはすでに説明しました。`UIPasteConfigurationSupporting` プロトコルを実装した view や `UIViewController` は、自動的にドロップに対応します。

`UIPasteConfigurationSupporting` の `paste(itemProviders:)` は、ペーストだけでなくドロップの際にも呼び出されます。1つの実装でペーストとドロップの両方をサポートできます。

リスト 6.15: ‘UIPasteConfigurationSupporting’の実装によるペーストやドロップのサポート

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        pasteConfiguration = UIPasteConfiguration(forAccepting: UIImage.self)
    }

    override func paste(itemProviders: [NSItemProvider]) {
        for itemProvider in itemProviders {
            itemProvider.loadObject(ofClass: UIImage.self, completionHandler:
                { [weak self] (image, error) in
                    guard let strongSelf = self else {
                        return
                    }
                    if let image = image as? UIImage {
                        DispatchQueue.main.async {
                            strongSelf.insert(image: image,
                                at: strongSelf.view.center)
                        }
                    } else if let error = error {
                        print(error)
                    }
                }
            )
        }
    }
}
```

```
        })
    }

private func insert(image: UIImage, at center: CGPoint) {
    let imageView = UIImageView(image: image)
    imageView.frame.size = AVMakeRect(
        aspectRatio: image.size,
        insideRect: CGRect(x: 0, y: 0, width: 200, height: 200)
    ).size
    imageView.center = center
    view.addSubview(imageView)
}

}
```

6.3.2 UIDropInteraction

`UIDropInteraction` を用いるドロップの実装では、ドロップについて細かくコントロールできます。`UIDropInteraction` も `UIDragInteraction` と同様に、`UIDropInteraction` クラスのイニシャライザに `UIDropInteractionDelegate` を設定し、view に `UIView` の `addInteraction(_:)` メソッドで追加します。

`UIDropInteractionDelegate` では、ドラッグが view の領域に入っているときに呼び出される `dropInteraction(_:sessionDidUpdate:)` メソッドと、実際にドロップするときに呼び出される `dropInteraction(_:performDrop:)` メソッドのふたつを実装する必要があります。

`dropInteraction(_:sessionDidUpdate:)` メソッドでは `UIDropProposal` オブジェクトを返します。このオブジェクトにはイニシャライザで `UIDropOperation` を渡します。

`UIDropOperation` が取り得る値は表 6.2 で示す通り `.cancel`、`.forbidden`、`.copy`、`.move` のいずれかです。`.cancel` もしくは `.forbidden` を返すと、ドロップはキャンセルされます。`.forbidden` は、本来ならドロップできるが何らかの理由で一時的にドロップできないことを示すのに使います。`.copy` と `.move` を返すと実際にドロップされます。ただし `.move` は、引数として渡される `UIDropSession` の `allowsMoveOperation: Bool` プロパティが `true` でなければ、`.cancel` と同じように扱われます。

表 6.2: `UIDropOperation` の値によってドロップ時の挙動が変わる

<code>UIDropOperation</code>	ドロップ	備考
<code>.cancel</code>	キャンセルされる	
<code>.forbidden</code>	キャンセルされる	理由があってドロップできないときに使う
<code>.copy</code>	ドロップされる	
<code>.move</code>	データの移動が許可されていればドロップされる	<code>UIDropSession</code> の <code>allowsMoveOperation</code> が <code>true</code> のとき

`UIDropOperation` の値は、ドラッグのプレビュー右上に表示されるアイコンにも反映されます。

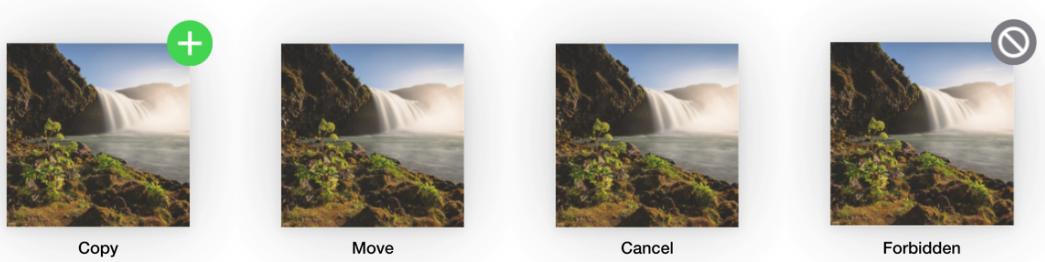


図 6.3: ‘UIDropOperation’の値によるアイコンの変化

`dropInteraction(_:performDrop:)` メソッドではドロップされた際の処理を行います。`UIDropSession` の `loadObjects(ofClass:completion:)` メソッドを用いてドラッグされていたデータを取得できます。データは `completion` クロージャで非同期に得られますが、このクロージャは `NSItemProvider` の場合とは異なり、メインスレッドで実行されます。

これらに加えて、`dropInteraction(_:canHandle:)` を実装し、`UIDropSession` の `canLoadObjects(ofClass:)` メソッドをなどを利用してドロップ可能かどうかを先に判定できます。

リスト 6.16: ドロップをサポートするための実装例

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let dropInteraction = UIDropInteraction(delegate: self)
        view.addInteraction(dropInteraction)
    }

    private func insert(image: UIImage, at center: CGPoint) {
        let imageView = UIImageView(image: image)
        imageView.frame.size = AVMakeRect(
            aspectRatio: image.size,
            insideRect: CGRect(x: 0, y: 0, width: 200, height: 200)
        ).size
        imageView.center = center
        view.addSubview(imageView)
    }

    extension ViewController: UIDropInteractionDelegate {

        func dropInteraction(_ interaction: UIDropInteraction,
                            canHandle session: UIDropSession) -> Bool {
            return session.canLoadObjects(ofClass: UIImage.self)
        } func dropInteraction(_ interaction: UIDropInteraction, canHandle
                            session: UIDropSession) -> Bool {
            return session.canLoadObjects(ofClass: UIImage.self)
        }
    }
}
```

```
func dropInteraction(_ interaction: UIDropInteraction, sessionDidUpdate
                     session: UIDropSession) -> UIDropProposal {
    guard session.canLoadObjects(ofClass: UIImage.self) else {
        return UIDropProposal(operation: .cancel)
    }
    return UIDropProposal(operation: .copy)
}

func dropInteraction(_ interaction: UIDropInteraction,
                     performDrop session: UIDropSession) {
    guard session.canLoadObjects(ofClass: UIImage.self) else {
        return
    }
    session.loadObjects(ofClass: UIImage.self) { [weak self] (images) in
        guard let strongSelf = self else {
            return
        }
        for image in images {
            strongSelf.insert(image: image as! UIImage,
                             at: session.location(in: strongSelf.view))
        }
    }
}
```

6.3.3 ドロップのプレビューとアニメーション

ドラッグするときと同じように、`UIDropInteractionDelegate` のメソッドを実装することで、ドロップ時のプレビューやアニメーションをカスタマイズできます。

`UIDropInteractionDelegate` の `dropInteraction(_:previewForDropping:withDefault:)` メソッドを実装することで、ドロップ時のプレビューをカスタマイズできます。`nil` を返せばデフォルトのドロップアニメーションが実行されます。`defaultPreview` として渡ってくる `UITargetedDragPreview` に `retargetedPreview(with:)` メソッドを呼び出すことで、特定の位置にドロップするアニメーションを作成できます。まったく新しく `UITargetedDropPreview` オブジェクトを作って返すこともできます。このようにドロップのプレビューをカスタマイズするためには、ドロップによって追加される要素の座標を事前に計算し、`UIDragPreviewTarget` を作る必要があります。また、一度にドロップされるアイテムの数が多いときは、カスタマイズに関わらずデフォルトのドロップアニメーションになります。

ドロップに合わせてアプリのユーザーインターフェースをアニメーションさせるためには、`UIDropInteractionDelegate` の `dropInteraction(_:item:willAnimateDropWith:)` メソッドを実装します。`animator` として渡される `UIDragAnimating` の `addAnimations(_:)` メソッドや `addCompletion(_:)` メソッドで、それぞれドロップ開始時とドロップ完了時に実行したいアニメーションを追加します。

6.3.4 ドロップの進捗表示

ドラッグ&ドロップでは `NSItemProvider` の機能を用いており、ドロップされた後に必要なデータを読み込むことを冒頭で説明しました。そのため、ユーザーがドロップしてからそれが画面に反映されるまでに時間がかかる場合があります。iOSは、データの読み出しに時間がかかると自動的にアラートを表示し、読み込みの進捗と「キャンセル」ボタンを表示します。このとき表示される進捗は、`NSItemProvider` から返された `Progress` を反映しています。`ProgressReporting` プロトコルに準拠している `UIDropSession` から `progress: Progress` プロパティで取得できるものと同じです。またこのアラートの「キャンセル」ボタンでは、ユーザーがドロップをキャンセルできます。このとき `Progress` オブジェクトの `cancel()` メソッドが呼び出されます。

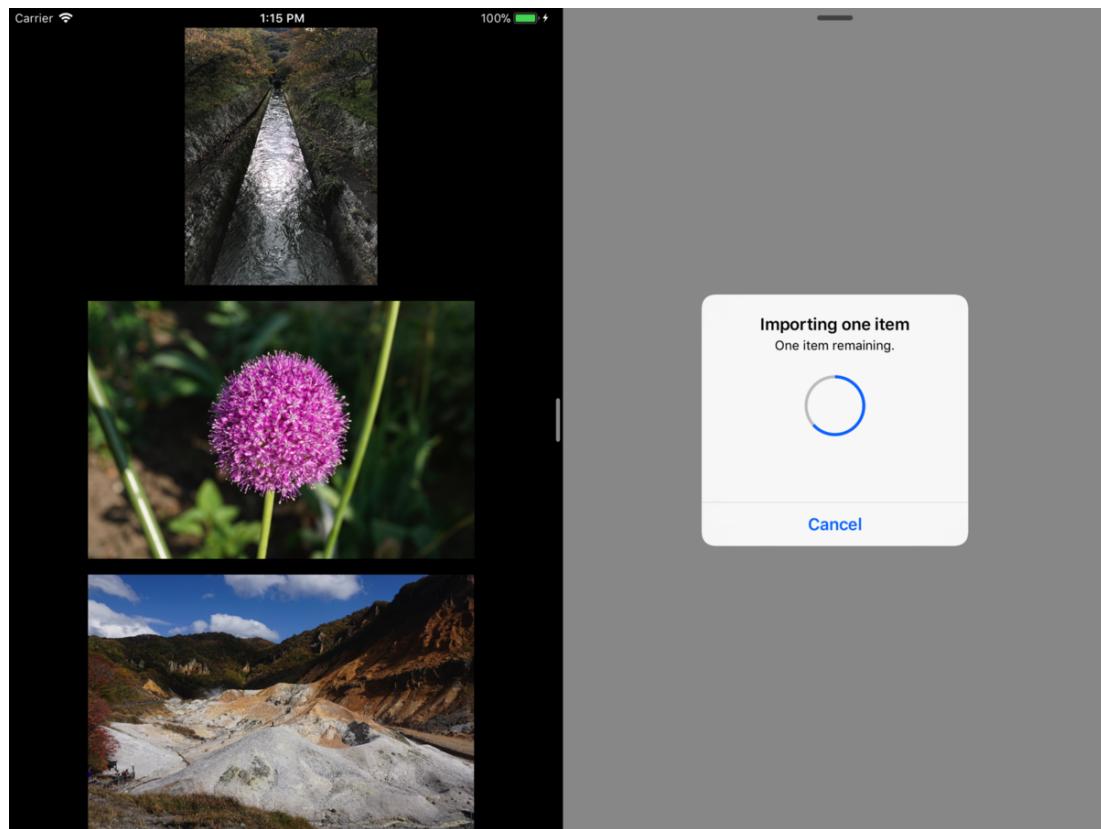


図 6.4: ドロップの進捗を表示するアラート

このようにドロップに時間がかかるってしまう場合にも、自動的に表示されるアラートによって、ユーザーがドロップをキャンセルすることができます。

このデフォルトの動作をカスタマイズする方法も用意されています。`UIDropSession` の `progressIndicatorStyle: UIDropSessionProgressIndicatorStyle` プロパティに `.none` を設定することで、iOSのデフォルトの表示を抑制できます。そして `UIDropSession` の `progress: Progress` プロパティか、または個々の `NSItemProvider` のメソッドがデータの読み込みの際に返

す Progress オブジェクトを利用して、読み込みの進捗を表示します。

リスト 6.17: ドロップ時の進捗表示を自分で実装する

```
extension ViewController: UIDropInteractionDelegate {

    func dropInteraction(_ interaction: UIDropInteraction,
                         performDrop session: UIDropSession) {
        guard session.canLoadObjects(ofClass: UIImage.self) else {
            return
        }

        let placeholder = UIView(frame: CGRect(x: 0, y: 0, width: 200, height: 200))
        placeholder.backgroundColor = UIColor.lightGray
        let indicator = UIProgressView(progressViewStyle: .default)
        indicator.center = placeholder.center
        placeholder.addSubview(indicator)
        placeholder.center = session.location(in: view)
        view.addSubview(placeholder)

        let progress = session.loadObjects(ofClass: UIImage.self)
                                    { [weak self] (images) in
            guard let strongSelf = self else {
                return
            }
            placeholder.removeFromSuperview()
            for image in images {
                strongSelf.insert(image: image as! UIImage,
                                  at: session.location(in: strongSelf.view))
            }
        }
        indicator.observedProgress = progress
    }

}
```

6.3.5 同一アプリ内でのドラッグ＆ドロップ

ドラッグ元とドロップ先が同一のアプリである場合には、UIDropSession の localDragSession: UIDragSession? プロパティから UIDragSession が得られます。UIDragSession には localContext: Any? プロパティがあり、アプリ内に限定した何らかのデータを持つことができます。

同一のアプリ内のドラッグ＆ドロップに限っては、iPad だけでなく iPhone でも有効にできます。そのためには UIDragInteraction の isEnabled プロパティを true にします。この値は iPad ではデフォルトで true ですが、iPhone の場合は異なります。デバイス毎のデフォルト値は UIDragInteraction のクラスプロパティ isEnabledByDefault から取得できます。

6.4 スプリングローディング

macOS の Finder で、ファイルをドラッグ中にフォルダーの上で少しホバーすると、そのフォルダーが開きます。この動作をスプリングローディングと呼びます。ドラッグ中はクリックができないので、スプリングローディングを利用してナビゲーションを行えるようになっています。iOS のドラッグ＆ドロップにもスプリングローディングがあります。

`UISpringLoadedInteractionSupporting` プロトコルに準拠した UIKit のクラスは、`isSpringLoaded: Bool` プロパティを `true` にするだけでスプリングローディングが有効になります。たとえば `UIButton` や `UIBarButtonItem`、`UITabBar` や `UITableView`、`UICollectionView` が準拠しています。ドラッグ中にこれらの要素の上でホバーすると、要素に応じてアクションが実行されます。たとえば `UIButton` などのボタンであれば、ボタンを押下したのと同じアクションが引き起こされます。`UITableView` や `UICollectionView` は、cell を選択したのと同じアクションが発生します。

リスト 6.18: ‘`UISpringLoadedInteractionSupporting`’プロトコルに準拠した ‘`UIButton`’でのスプリングローディングの対応

```
class ViewController: UIViewController {

    @IBOutlet weak var button: UIButton!
    didSet {
        button.isSpringLoaded = true
    }
}
```

任意の view でスプリングローディングをサポートするためには `UISpringLoadedInteraction` を使います。

`UISpringLoadedInteraction` の `init(activationHandler:)` イニシャライザにスプリングローディング時の処理をクロージャで渡して初期化します。作成した `UISpringLoadedInteraction` オブジェクトを `UIView` に `addInteraction(_:)` することで、スプリングローディングに対応できます。

`UISpringLoadedInteraction` のイニシャライザは、この他に `UISpringLoadedInteractionBehavior` プロトコルに準拠したオブジェクトと、`UISpringLoadedInteractionEffect` プロトコルに準拠したオブジェクトをそれぞれ設定することができ、スプリングローディングの挙動をカスタマイズできます。`UISpringLoadedInteractionBehavior` プロトコルは、スプリングローディングを実行するか決める `shouldAllow(_:with:)` メソッドと、スプリングローディングが実行された後に呼ばれる `interactionDidFinish(_:)` メソッドを持ちます。`UISpringLoadedInteractionEffect` プロトコルは、`interaction(_:didChangeWith:)` メソッドで、状態に応じてスプリングローディング中の view のエフェクトを設定します。

6.5 UITableView と UICollectionView

ここまでに、任意の view でドラッグ&ドロップをサポートするには `UIDragInteraction` と `UIDropInteraction` を用いることを説明しました。

ところで、多くのアプリではドラッグ&ドロップの対象となるデータは、`UITableView` や `UICollectionView` といった、コレクションを扱う view で表示しているのではないでしょうか。このふたつの view について、`IndexPath` を用いたドラッグ&ドロップを簡単に実装できるよう特別な API が用意されています。

ここからの説明では `UITableView` を例に進めますが、対応する API は `UICollectionView` にも存在します。`UITableView` と `UICollectionView` の API はよく似ているので、ドラッグ&ドロップに関する API も非常によく似ています。

6.5.1 UITableView と UICollectionView のドラッグ

`UITableView` と `UICollectionView` に、新しく `dragDelegate` プロパティが追加されました。それぞれ `UITableViewDragDelegate` プロトコルと `UICollectionViewDragDelegate` プロトコルに準拠したオブジェクトを設定できます。

基本は `UIDragInteractionDelegate` を実装するときと同じで、ドラッグ開始時に呼ばれるデリゲートメソッドで `UIDragItem` の配列を返します。`UITableViewDragDelegate` プロトコルの場合、`dragInteraction(_:itemsForBeginning:)` の代わりに `tableView(_:itemsForBeginning:at:)` メソッドが呼ばれます。この引数に `IndexPath` が含まれるので、関連する `UIDragItem` の配列を特定しやすくなっています。

ドラッグをサポートするために最低限必要なのはこのメソッドだけです。

リスト 6.19: ‘UITableView’でドラッグをサポートする

```
class ViewController: UITableViewController {

    static let BasicCellIdentifier: String = "BasicCell"

    var names: [String] = [
        "First",
        "Second",
        "Third",
    ]

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView.dragDelegate = self
    }

    override func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int {
        return names.count
    }
}
```

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: ViewController.BasicCellIdentifier, for: indexPath)
    cell.textLabel?.text = names[indexPath.row]
    return cell
}

extension ViewController: UITableViewDragDelegate {

    func tableView(_ tableView: UITableView,
        itemsForBeginning session: UIDragSession,
        at indexPath: IndexPath) -> [UIDragItem] {
        let itemProvider = NSItemProvider(object: names[indexPath.row] as NSString)
        let dragItem = UIDragItem(itemProvider: itemProvider)
        return [dragItem]
    }

}
```

複数アイテムのドラッグをサポートするには `tableView(_:itemsForAddingTo:at:point:)` メソッドを実装します。これも `UIDragInteractionDelegate` の `dragInteraction(_:itemsForAddingTo:withTouchAt:)` メソッドと対応しています。

`UITableView` や `UICollectionView` の項目の選択状態は、ドラッグにも影響します。ドラッグ開始時や要素の追加の際に、選択状態の項目をドラッグしようとすると、選択状態になっている要素すべてがドラッグ対象の要素に加わります。

ドラッグ中は、デフォルトではドラッグ中のアイテムの `cell` がプレビューに使われます。`cell` の一部分だけをプレビューとして使うなど、カスタマイズしたい場合は `tableView(_:dragPreviewParametersForRowAt:)` メソッドで `UIDragPreviewParameters` オブジェクトを返します。たとえば `cell` に画像のサムネイルが表示されていて、その画像のデータだけがドラッグされるような場合に使えます。

ドラッグ中の `cell` の見た目を変えたい場合は、`UITableViewCell` や `UICollectionView` に追加された `dragStateDidChange(_:)` メソッドを `override` します。このメソッドに渡される `UITableViewCellDragState` または `UICollectionViewCellDragState` の値が、ドラッグ中の状態に応じて、通常時の `.none` からドラッグのリフト中である `.lifting`、ドラッグ中の `.dragging` と、順に変化します。

6.5.2 ドロップ

ドラッグと同様にドロップについても、`UITableView` と `UICollectionView` に新しく `dropDelegate` プロパティが追加されました。`UITableView` と `UICollectionView` で、それぞれ `UITableViewDropDelegate` プロトコルまたは `UICollectionViewDropDelegate` プロトコルに準拠したオブジェクトを設定します。

最低限実装が必要なのは `tableView(_:performDropWith:)` メソッドです。引数として渡ってくる `UITableViewDropCoordinator` オブジェクトを利用して、ドロップ時の挙動を実装します。

リスト 6.20: ‘UITableView’でドロップをサポートする

```
extension ViewController: UITableViewDropDelegate {

    func tableView(_ tableView: UITableView,
                  performDropWith coordinator: UITableViewDropCoordinator) {
        let destination: IndexPath
        if let indexPath = coordinator.destinationIndexPath {
            destination = indexPath
        } else {
            let section = tableView.numberOfSections - 1
            let row = tableView.numberOfRows(inSection: section)
            destination = IndexPath(row: row, section: section)
        }

        _ = coordinator.session.loadObjects(ofClass: String.self) { items in
            var indexPaths: [IndexPath] = []
            for (index, item) in items.enumerated() {
                let indexPath = IndexPath(row: destination.row + index,
                                         section: destination.section)
                self.names.insert(item, at: indexPath.row)
                indexPaths.append(indexPath)
            }
            self.tableView.insertRows(at: indexPaths, with: .automatic)
        }
    }
}
```

ドロップに対応するには、`UITableViewDropCoordinator` の `destinationIndexPath: IndexPath` プロパティからドロップ先となる `IndexPath` を取得します。`UITableView` の余白部分にドロップしたときは `nil` になっているので、その場合は `IndexPath` を作ります。次に `session: UIDropSession` プロパティからデータの読み出しを行います。そしてデータソースへのデータの追加や `UITableView` に対して `insertRows(at:with:)` メソッドの呼び出しを行い、実際に `UITableView` を更新します。

事前に対応可能なデータかどうかを確かめたい場合、`UITableViewDropDelegate` の `tableView(_:canHandle:)` メソッドを実装します。

リスト 6.21: ‘UITableView’におけるドロップのカスタマイズ

```
extension ViewController: UITableViewDropDelegate {

    func tableView(_ tableView: UITableView, canHandle
                  session: UIDropSession) -> Bool {
        return session.hasItemsConforming(
            toTypeIdentifiers: [kUTTypeUTF8PlainText as String])
    }
}
```

```
func tableView(_ tableView: UITableView, dropSessionDidUpdate
    session: UIDropSession, withDestinationIndexPath
    destinationIndexPath: IndexPath?) -> UITableViewDropProposal {
    return UITableViewDropProposal(operation: .copy,
                                    intent: .insertAtDestinationIndexPath)
}
```

ドロップされる位置にすき間を作るなど、ドロップ中の動作をユーザーインターフェースに反映させたい場合、UITableViewDropDelegate の tableView(_:dropSessionDidUpdate:withDestinationIndexPath:) メソッドを実装し、UITableViewDropProposal を返します。

UITableViewDropProposal の init(operation:intent:) イニシャライザで、第一引数の UIDropOperation で .copy か .move かを示せるほか、第二引数の UITableViewDropIntent によってドロップされる予定の位置を示せます。UITableViewDropIntent の .insertAtDestinationIndexPath は、IndexPath の位置に挿入されることを示し、.insertIntoDestinationIndexPath は IndexPath の位置にある既存の cell の中に取り込まれることを意味します。.automatic は位置に応じて、それらのうち適した方が選ばれます。

これらの値によって、UITableView や UICollectionView は、ドロップされる位置をユーザーインターフェースに反映します。たとえば .insertAtDestinationIndexPath のときは、ドロップされる位置を表すために、cell ひとつ分の空間を作ります。

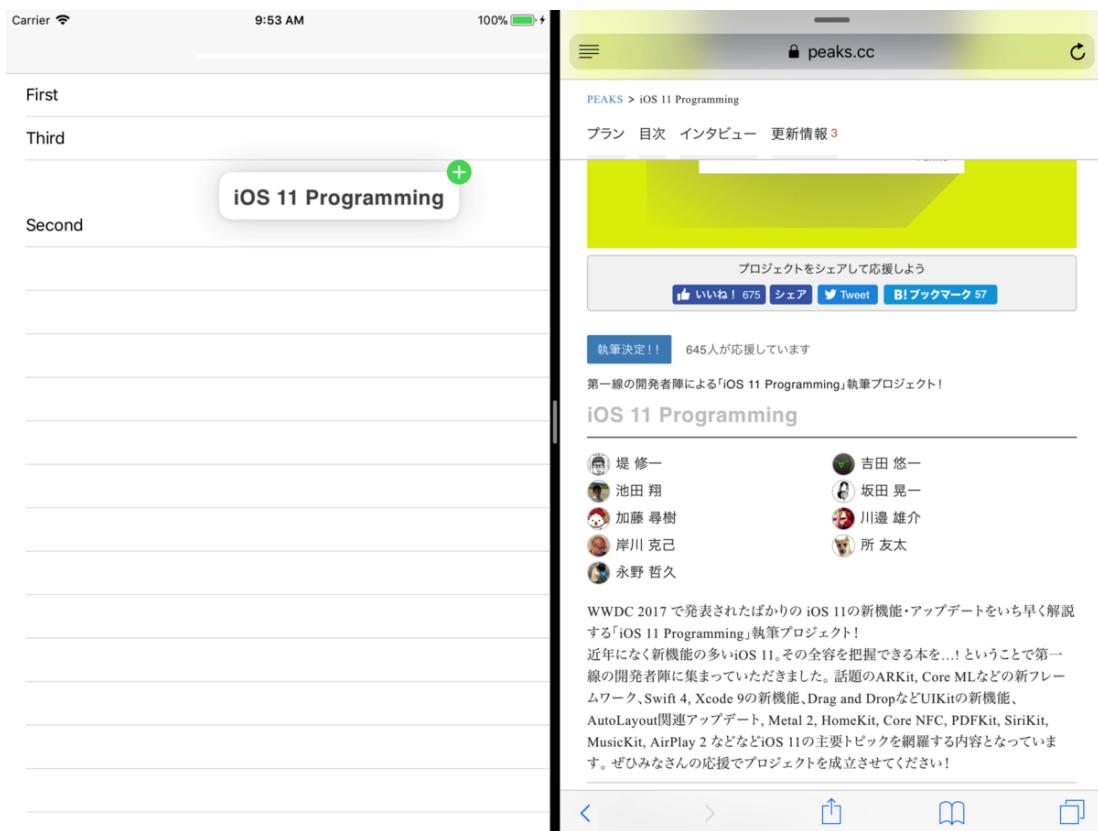


図 6.5: UITableViewDropIntent.insertAtDestinationIndexPath を指定すると cell ひとつ分の空間が作られる

表 6.3: UITableViewDropIntent とドロップされる位置

UITableViewDropIntent	ドロップされる位置
.unspecified	指定されない
.insertAtDestinationIndexPath	index path の位置
.insertIntoDestinationIndexPath	index path に存在する cell の内側
.automatic	ドロップの位置に応じて index path の位置か、または既存の cell の内側

表 6.4: UICollectionViewDropIntent とドロップされる位置

UICollectionViewDropIntent	ドロップされる位置
.unspecified	指定されない
.insertAtDestinationIndexPath	index path の位置
.insertIntoDestinationIndexPath	index path に存在する cell の内側

6.5.3 ドロップのプレースホルダ

ドロップ実行後、読み込みに時間がかかると、ユーザーインターフェースになかなか反映されず、ドロップしたことが分かりづらい状態になってしまいます。これを緩和するために、UITableView や UICollectionView にはプレースホルダを表示するための機能が追加されました。

UITableViewDropPlaceholder は、UITableViewPlaceholder を継承した、ドロップのプレースホルダの機能を提供するためのクラスです。`init(insertionIndexPath:reuseIdentifier:rowHeight:)` イニシャライザを使って、挿入位置の IndexPath とプレースホルダに使う cell の reuseIdentifier、そして cell の高さを与え、初期化します。`cellUpdateHandler:((UITableViewCell) -> Void)?` プロパティにクロージャを設定し、プレースホルダの cell の表示を変更できます。

UITableViewDropCoordinator の `drop(_:to:)` メソッドに UITableViewDropPlaceholder を渡すと、プレースホルダを使ったドロップアニメーションを実行できます。このメソッドを呼び出すとドロップアニメーションとともにプレースホルダが UITableView へ挿入されます。このメソッドは UITableViewDropPlaceholderContext オブジェクトを返します。NSItemProvider を使ってデータの取得に成功したら、UITableViewDropPlaceholderContext の `commitInsertion(dataSourceUpdates:)` メソッドを呼び出します。このとき dataSourceUpdates に、データソースにもデータを追加するクロージャを渡します。もしもデータの取得に失敗したら、`deletePlaceholder()` メソッドでプレースホルダを消去します。

リスト 6.22: ‘UITableViewDropPlaceholder’を用いてドロップ時にプレースホルダを表示する

```
extension ViewController: UITableViewDropDelegate {

    func tableView(_ tableView: UITableView,
                  performDropWith coordinator: UITableViewDropCoordinator) {
        let destination: IndexPath
        if let indexPath = coordinator.destinationIndexPath {
            destination = indexPath
        } else {
            let section = tableView.numberOfSections - 1
            let row = tableView.numberOfRows(inSection: section)
            destination = IndexPath(row: row, section: section)
        }

        for (index, item) in coordinator.items.enumerated() {
            let indexPath = IndexPath(row: destination.row + index,
                                      section: destination.section)

            let placeholder = UITableViewDropPlaceholder(
                insertionIndexPath: indexPath,
                reuseIdentifier: PlaceholderCellIdentifier,
                rowHeight: UITableViewAutomaticDimension)
            placeholder.cellUpdateHandler = { cell in
                cell.textLabel?.text = "..."
            }

            let placeholderContext = coordinator.drop(item.dragItem, to: placeholder)
        }
    }
}
```

```
        _ = item.dragItem.itemProvider.loadObject(
            ofType: String.self) { (item, error) in
        DispatchQueue.main.async {
            if let item = item {
                placeholderContext.commitInsertion(
                    dataSourceUpdates: { indexPath in
                        self.names.insert(item, at: indexPath.row)
                })
            } else {
                placeholderContext.deletePlaceholder()
            }
        }
    }
}
```

プレースホルダの仕組みによって、非同期的なデータ読み込みがあっても挿入位置を見失わずに済むようになっています。しかしプレースホルダは、データソースには存在しない要素です。UITableView の reloadData() などのメソッドを呼び出してしまうと、挿入したプレースホルダは失われてしまいます。プレースホルダを利用するときは reloadData() などのメソッドを呼び出さずに、UITableview に新たに追加された performBatchUpdates(_:completion:) メソッドのクロージャ内部で、insertRows(at:with:) などのメソッドを利用してください。

プレースホルダは、UITableView の API からは存在しないかのように扱われます。内部的には UITableView や UICollectionView が新たに準拠するようになった UITableViewDataSourceTranslating プロトコルによって、IndexPath の変換が行われています。プレースホルダも含めて実際に表示されている位置の IndexPath を Presentation Index Path と呼び、データソースの IndexPath を Data Source Index Path と呼びます。通常の API からは Data Source Index Path としてやり取りされます。ただし UICollectionView のレイアウトのために Presentation Index Path を得たい場合は、dataSourceIndexPath(forPresentationIndexPath:) メソッドや presentationSectionIndex(forDataSourceSectionIndex:) メソッドを通して、Presentation Index Path と Data Source Index Path を相互に変換できます。

6.5.4 同一アプリ内でのドラッグ&ドロップ

同一アプリ内でのドラッグ&ドロップでは、データを他のアプリと同じように非同期でやり取りするのは無駄があります。この場合、UIDragItem の localObject プロパティを使って直接データを参照させることで、非同期的なデータのやり取りが不要になり、高速に動作します。

リスト 6.23: アプリ内でのドラッグ&ドロップは ‘localObject’を利用できる

```
extension ViewController: UITableViewDropDelegate {

    func tableView(_ tableView: UITableView,
                  performDropWith coordinator: UITableViewDropCoordinator) {
```

```
let destination: IndexPath
if let indexPath = coordinator.destinationIndexPath {
    destination = indexPath
} else {
    let section = tableView.numberOfSections - 1
    let row = tableView.numberOfRows(inSection: section)
    destination = IndexPath(row: row, section: section)
}

for (index, item) in coordinator.items.enumerated() {
    let indexPath = IndexPath(row: destination.row + index,
                             section: destination.section)

    if let localObject = item.dragItem.localObject as? String {
        tableView.performBatchUpdates({
            names.insert(localObject, at: indexPath.row)
            tableView.insertRows(at: [indexPath], with: .automatic)
        }, completion: nil)
        coordinator.drop(item.dragItem, toRowAt: indexPath)
    } else {
        ...
    }
}
}
```

UITableView の `performBatchUpdates(_:completion:)` メソッドを呼び出した後に、
UITableViewDropCoordinator の `drop(_:toRowAt:)` などのメソッドを呼び出すと、アニメーションを実行させることができます。

ドロップアニメーションは UITableViewDropCoordinator の 3 つのメソッドを使い分けることで違ったものになります。IndexPath への挿入を示す `drop(_:toRowAt:)` メソッド、IndexPath の位置にある既存の cell の内側に取り込まれる `drop(_:intoRowAt:rect:)` メソッド、そしてまったくの任意の位置へのドロップを表す `drop(_:to:)` メソッドがあります。

さらに、同一の UITableView や UICollectionView の中でドラッグ&ドロップすると、ドラッグ&ドロップによる並び替えが可能です。そのためにはまず、`tableView(_:dropSessionDidUpdate:withDestinationIndexPath:)` メソッドで UITableViewDropProposal を返すときに、`UIDropOperation.move` を設定します。あとはドロップの対応箇所で、実際に並び替えを行います。ただし UITableView の場合、UITableViewDataSource の `tableView(_:moveRowAt:to:)` メソッドが実装されていれば、こちらが利用されます。

リスト 6.24: 並び替えにドラッグ&ドロップを利用すると ‘UITableViewDataSource’ の ‘tableView(_:moveRowAt:to:)’ メソッドが呼び出される

```
class ViewController: UITableViewController {

    override func tableView(_ tableView: UITableView,
```

```
        moveRowAt sourceIndexPath: IndexPath,
        to destinationIndexPath: IndexPath) {
    let name = names.remove(at: sourceIndexPath.row)
    names.insert(name, at: destinationIndexPath.row)
}

extension ViewController: UITableViewDropDelegate {

    func tableView(_ tableView: UITableView,
                  dropSessionDidUpdate session: UIDropSession,
                  withDestinationIndexPath destinationIndexPath: IndexPath?) -> UITableViewDropProposal {
        if tableView.hasActiveDrag {
            if session.items.count > 1 {
                return UITableViewDropProposal(operation: .cancel)
            } else {
                return UITableViewDropProposal(operation: .move,
                                               intent: .insertAtDestinationIndexPath)
            }
        } else {
            return UITableViewDropProposal(operation: .copy,
                                           intent: .insertAtDestinationIndexPath)
        }
    }
}
```

また UICollectionView には、reorderingCadence プロパティが追加されました。

```
var reorderingCadence: UICollectionViewReorderingCadence { get set }
```

.immediate、.fast、.slow の値があり、UICollectionView の並び替えの際に、ドロップされる位置をどれくらいの早さで認識するかに影響します。

6.5.5 UITableView と UICollectionView のスプリングローディング

UITableView や UICollectionView でもスプリングローディングがサポートされています。UISpringLoadedInteractionSupporting プロトコルに準拠しているため、isSpringLoaded: Bool プロパティでスプリングローディングを有効にできます。さらに要素ごとでスプリングローディングのサポートを変えたければ、UITableViewDelegate の tableView(_:shouldSpringLoadRowAt:with:) メソッドを実装することで個別設定できます。

6.6 UITextView と UITextField

UITextView、UITextField、WKWebView、UIWebView は、デフォルトでドラッグ&ドロップをサポートしています。UITextView と UITextField については、ドラッグ&ドロップの挙動をカスタマイズできます。

UITextView と UITextField は、新たに UITextDraggable プロトコルおよび UITextDroppable プロトコルに準拠するようになりました。textDragDelegate: UITextDragDelegate? プロパティと textDropDelegate: UITextDropDelegate? プロパティが追加され、これらのデリゲートを介してドラッグとドロップをカスタマイズできます。

UITextDroppable プロトコルは UITextPasteConfigurationSupporting プロトコルを継承しています。UITextPasteConfigurationSupporting プロトコルには pasteDelegate: UITextPasteDelegate? プロパティが存在し、このデリゲートでドロップとペーストをカスタマイズできます。

6.6.1 ドラッグのカスタマイズ

UITextDragDelegate では、textDraggableView(_:itemsForDrag:) メソッドを実装して、ドラッグするデータをカスタマイズできます。引数に UITextDragRequest が渡されるので、これを利用してドラッグするデータを決めます。デフォルトでドラッグされるデータは suggestedItems プロパティに格納されています。

リスト 6.25: ‘UITextDragDelegate’の実装

```
class ViewController: UIViewController {

    @IBOutlet weak var textView: UITextView!
    didSet {
        textView.textDragDelegate = self
        textView.textDropDelegate = self
        textView.pasteDelegate = self
    }
}

extension ViewController: UITextDragDelegate {

    func textDraggableView(_ textDraggableView: UIView & UITextDraggable,
                           itemsForDrag dragRequest: UITextDragRequest) -> [UIDragItem] {
        return dragRequest.suggestedItems
    }
}
```

6.6.2 ドロップのカスタマイズ

UITextView の `isEditable` プロパティが `false` で編集できないときでもドロップのみ許可したい場合、UITextDropDelegate の `textDroppableView(_:willBecomeEditableForDrop:)` メソッドを使います。

ここで `UITextDropEditability` の値として `.temporary` を返すと、編集できない UITextView でもドロップできます。さらに `.yes` を返せば、ドロップ以降は編集可能な状態になります。デフォルトは `.no` です。

また `textDroppableView(_:proposalForDrop:)` メソッドで UIDropProposal クラスを継承した UITextDropProposal クラスのインスタンスを返すことで、ドロップできるかどうかを含めたカスタマイズが可能です。UITextDropProposal の `dropAction: UITextDropAction` プロパティでは、デフォルトの `.insert` ならキャレットの位置に挿入されますが、これを `.replaceAll` にすると、UITextView や UITextField の中身のすべてが置き替わります。また `.replaceSelection` であれば、選択範囲があるときは選択範囲の文字列を置き換え、選択範囲がなければ `.insert` と同様になります。

リスト 6.26: ‘UITextDropProposal’や‘UITextDropAction’によってドロップ時の動作を変える

```
extension ViewController: UITextDropDelegate {

    func textDroppableView(_ textDroppableView: UIView & UITextDroppable,
                           proposalForDrop drop: UITextDropRequest) -> UITextDropProposal {
        let proposal = UITextDropProposal(operation: .copy)
        proposal.dropAction = .replaceAll
        return proposal
    }

}
```

6.6.3 ドロップとペーストのカスタマイズ

ドロップされるテキストの属性付きテキストからスタイル情報を除去したプレーンテキストにしたい場合、UITextPasteDelegate の `textPasteConfigurationSupporting(_:transform:)` メソッドで、ドロップされるデータを書き換えて実現します。引数で渡される UITextPasteItem オブジェクトの `itemProvider: NSItemProvider` からデータを取得し、UITextPasteItem の `setResult(string:)` メソッドや `setResult(attributedString:)` メソッド、あるいは `setResult(attachment:)` メソッドで、UITextView などに反映させます。

データをハンドリングできない場合には `setDefaultResult()` メソッドを呼び出すことでデフォルトのドロップを実行できます。`setNoResult()` メソッドでドロップが実行されないようにもできます。

リスト 6.27: ‘UITextPasteDelegate’でペースト時の挙動をカスタマイズする

```
extension ViewController: UITextPasteDelegate {

    func textPasteConfigurationSupporting(
        _ textPasteConfigurationSupporting: UITextPasteConfigurationSupporting,
        transform item: UITextPasteItem) {
        _ = item.itemProvider.loadObject(ofClass: String.self) {
            (string, error) in
            DispatchQueue.main.async {
                if let string = string {
                    item.setResult(string: string)
                } else {
                    item.setNoResult()
                }
            }
        }
    }
}
```

6.6.4 受け付けるデータの種類

UITextView と UITextField がペーストやドロップによって受け付けられるデータは、デフォルトでは allowsEditingTextAttributes: Bool プロパティの値によって決まります。allowsEditingTextAttributes が true であればプレーンテキストのほか、属性付きのテキストも受け付けます。UITextView なら画像も受け付けるようになっています。allowsEditingTextAttributes が false なら、UITextView も UITextField もプレーンテキストだけを受け付けます。

この挙動は UITextView と UITextField の pasteConfiguration: UIPasteConfiguration? プロパティを変更することでカスタマイズできます。たとえば UIImage を受け付けるような UIPasteConfiguration を設定すれば、allowsEditingTextAttributes が false でも画像をドロップしたりペーストしたりすることが可能になります。このとき、追加したデータの種類は UITextPasteDelegate を使って独自にハンドリングする必要があります。

リスト 6.28: ‘UITextView’で画像のドロップに対応する

```
class ViewController: UIViewController {

    @IBOutlet weak var textView: UITextView! {
        didSet {
            textView.pasteDelegate = self
            let pasteConfiguration = UIPasteConfiguration()
            pasteConfiguration.addTypeIdentifiers(forAccepting: String.self)
            pasteConfiguration.addTypeIdentifiers(forAccepting: UIImage.self)
            textView.pasteConfiguration = pasteConfiguration
        }
    }
}
```

```
extension ViewController: UITextPasteDelegate {  
  
    func textPasteConfigurationSupporting(  
        _ textPasteConfigurationSupporting: UITextPasteConfigurationSupporting,  
        transform item: UITextPasteItem) {  
        if item.itemProvider.canLoadObject(ofClass: UIImage.self) {  
            _ = item.itemProvider.loadObject(ofClass: UIImage.self) {  
                (image, error) in  
                DispatchQueue.main.async {  
                    if let image = image as? UIImage {  
                        item.setResult(attachment: NSTextAttachment(  
                            data: UIImagePNGRepresentation(image),  
                            ofType: kUTTypePNG as String  
                        ))  
                    } else {  
                        item.setNoResult()  
                    }  
                }  
            }  
        } else {  
            item.setDefaultResult()  
        }  
    }  
}
```

第7章

Files と Document Based Application

7.1 はじめに

本章では、iOS 11 の Files アプリと Document-Based App について紹介します。

Files アプリは、iOS 11 から追加された新しいシステムアプリです。iOS 10 までの iCloud Drive アプリを置き換え、さらに機能拡張する形で導入されました。Files アプリでは iCloud Drive だけでなく、アプリのコンテナ内にある Documents ディレクトリも含む、iOS 上のあらゆるドキュメントを一覧できます。

Document-Based App とは、もともとある種の macOS のアプリについて使われている呼称です。アプリの主要な機能がドキュメントを中心とした構成になっていて、ドキュメントを作成・編集・保存することが主目的のアプリです。

典型的には、macOS だとひとつのウインドウ／タブがひとつのドキュメントに対応します。代表的なアプリとして、macOS／iOS における Pages、Numbers、Keynote などがあります。

iOS 11 では、ドキュメントを中心としたユーザーのワークフローが、より実現しやすくなりました。これまでのアプリ中心の考え方だけでなく、ドキュメント（ファイル）を中心に据えてアプリを使うことが、今まで以上に大きな意味を持ちます。

7.2 Files アプリ

Files アプリには iCloud Drive とサードパーティのクラウドストレージサービス、そしてアプリの Documents ディレクトリが統合されています。

Files には iCloud Drive 上のファイルや Application Cloud Container が並びます。これらはサードパーティのクラウドストレージサービスと同様に、File Provider extension として実装されています。

Files では、クラウドストレージサービスやアプリ内の Documents ディレクトリなど、ファイルの物理的な位置による整理だけでなく、「最近使った項目（Recents）」や「タグ（Tags）」のように、横断的に整理する仕組みがあります。

自分が作ったアプリのコンテナの中にある Documents ディレクトリも、オプトインで Files アプリに表示させられます。

7.3 ドキュメントブラウザ API

Document-Based App の基本的な利用フローでは、アプリを起動すると最初にドキュメントブラウザが表示されます。そこから既存のドキュメントを開くか、あるいは新規作成するかを選んで、アプリを利用します。

iOS における Document-Based App で中心的な役割を担うのは、`UIDocumentBrowserViewController` という新しく追加された View Controller です。ドキュメントブラウザの標準的なユーザーインターフェイスを提供します（図 7.1）。

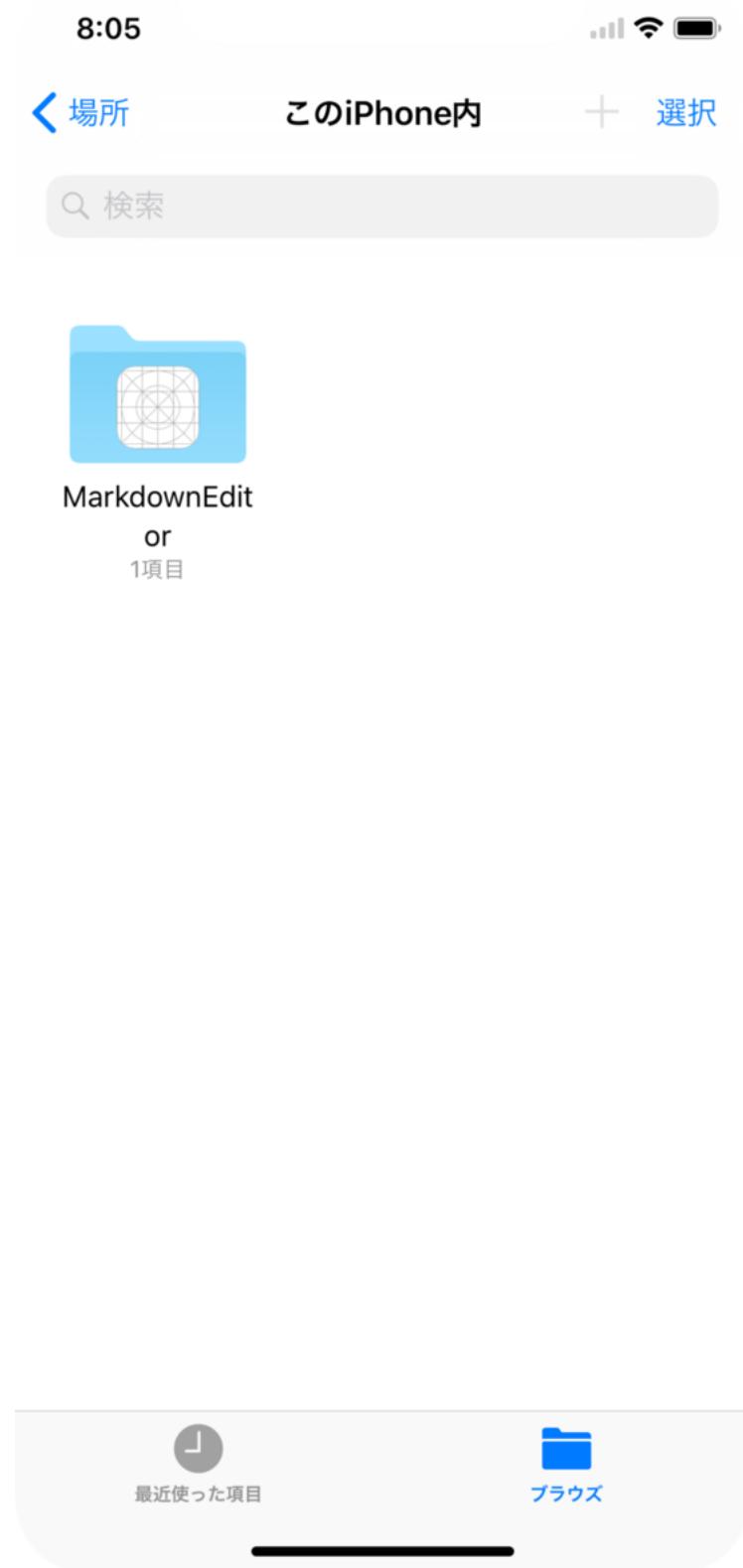


図 7.1: ‘UIDocumentBrowserViewController’

`UIDocumentBrowserViewController` は Files アプリと同様の機能を持っていて、さらにアピアランスも変更できるので、アプリに組み込みやすくなっています。

iOS 10 までは、ドキュメントブラウザを実現するために Foundation Framework の機能を直接的に利用せざるを得ませんでした。`NSMetadataQuery` クラスを利用して、iCloud Drive や他のアプリから共有されているファイルを取得する必要がありました。

`UIDocumentBrowserViewController` を使うと、アプリのコンテナ内のファイルにアクセスできるだけでなく、他のアプリのコンテナやクラウドストレージサービスに置かれているファイルにもアクセスできます。

なお、この `UIDocumentBrowserViewController` の導入によって、iOS 8 から存在していた `UIDocumentMenuViewController` は非推奨になっています。

Document-Based App ではほかにも、ドキュメントの Open in Place と呼ばれる機能や、複数のプロセスから適切な方法でファイルシステムにアクセスするための仕組みなどが互いに連携して、アプリの機能を構成します。iOS 11 では新しく「Thumbnail Extension」や「Quick Look Preview Extension」も追加され、ドキュメントを中心としたワークフローをより使いやすいものにしてくれます。

7.4 Document-Based App の実装

`UIDocumentBrowserViewController` を使って Document-Based App を実装してみます。

7.4.1 Documents ディレクトリの公開

Files アプリで自作アプリのドキュメントを一覧できるようにするには、Info.plist ファイルで `UISupportsDocumentBrowser` キーに YES を設定するか、もしくは `UIFileSharingEnabled` と `LSSupportsOpeningDocumentsInPlace` の両方に YES を設定します。

`UISupportsDocumentBrowser` は、iOS 11 で新しく追加された設定で、アプリが Document-Based App であり、`UIDocumentBrowserViewController` を使用していることを示します。`UISupportsDocumentBrowser` が有効になっていると、サンドボックス内の Documents ディレクトリが Files アプリに公開されます。

`UIFileSharingEnabled` は iTunes から Documents ディレクトリ内のファイルにアクセスするための設定です。`LSSupportsOpeningDocumentsInPlace` は、他のアプリからコンテナ内のファイルに直接アクセスさせるために使います。`UIFileSharingEnabled` と `LSSupportsOpeningDocumentsInPlace` の 2 つが有効になっているとき、コンテナ内の Documents ディレクトリが Files アプリに公開されます。

7.4.2 サポートするドキュメント形式の宣言

アプリが取り扱うドキュメントの形式も Info.plist ファイルで宣言します。Info.plist の `CFBundleDocumentTypes` キーに、アプリが対応するドキュメントの形式を配列として列挙します。ドキュメントの種類は辞書で、表 7.1 の値をそれぞれ設定します。

`CFBundleTypeIconFiles` に指定するアイコンのサイズは、iPad では 64 × 64px と 320 × 320px、

表 7.1: CFBundledocumenttypes に設定されるドキュメントの形式

キー	説明
CFBundletypename	ドキュメントの形式の名前を指定
CFBundletypeiconfiles	ドキュメントの形式を表すアイコン画像の名前を配列で指定
Lsitemcontenttypes	ドキュメントの形式を表す UTI の配列を指定
Lshandlerrank	ドキュメントに対するアプリの優先順位を指定
CFBundletyperole	ドキュメントに対してアプリが何ができるのか指定

iPhone では $22 \times 29\text{px}$ と $44 \times 58\text{px}$ です。

Lshandlerrank は表 7.2 の値を持ちます。

表 7.2: Lshandlerrank

Lshandlerrank の値	説明
Owner	ドキュメントを作り出す主要なアプリであることを示す
Default	ドキュメントを開くのに使われるアプリであることを示す
Alternate	ドキュメントを他のアプリの代替として閲覧するアプリであることを示す
None	ドキュメントを開くときには利用されないがドロップを受け付けるアプリであることを示す

CFBundletyperole は表 7.3 の値を持ちます。

表 7.3: CFBundletyperole

CFBundletyperole の値	説明
Editor	ファイルを表示・編集できることを示す
Viewer	ドキュメントを表示できることを示す
Shell	実行可能なドキュメントのランタイムサービスとなるアプリであることを示す
None	ドキュメントに対してなんの操作もできないことを示す

たとえば、Markdown エディタを作ろうとした場合、CFBundledocumenttypes 内は表 7.4 のように設定します。

表 7.4: Markdown エディタの CFBundledocumenttypes 設定例

キー	値
CFBundletypeiconfiles	markdown-document-320.png markdown-document-64.png markdown-document-512.png
CFBundletypename	Markdown
CFBundletyperole	Editor
Lshandlerrank	Alternate
Lsitemcontenttypes	net.daringfireball.markdown

iOS が定義していない UTI を使いたい場合、Info.plist に UTI の定義が必要です。

`UTEportedTypeDeclarations` もしくは `UTImportedTypeDeclarations` キーに、UTI を定義する辞書を配列で指定します。2つのキーは、UTI の定義が「exported」か「imported」のどちらに属するかで使い分けます。

- `exported`: アプリが独自に UTI を定義する → `UTEportedTypeDeclarations` キー
- `imported`: 他のアプリやデベロッパによって定義される UTI に対応する → `UTImportedTypeDeclarations` キー

たとえば、全く新しい画像フォーマットを策定したのであれば「exported」で、他社の策定している画像フォーマットに対応したのであれば「imported」となります。

表 7.5 は「exported」のときに使う `UTEportedTypeDeclarations` キーの内容です。

表 7.5: `UTEportedTypeDeclarations` キーに指定する項目

キー	説明
<code>UTTypeConformsTo</code>	UTI が継承する他の UTI の配列
<code>UTTypeDescription</code>	UTI の説明。InfoPlist.strings ファイルでローカライズできる
<code>UTTypeIdentifier</code>	UTI 文字列
<code>UTTypeSize64IconFile</code>	64 × 64px のアイコン画像。「exported」の UTI にのみ含める
<code>UTTypeSize320IconFile</code>	320 × 320px のアイコン画像。「exported」の UTI にのみ含める
<code>UTTypeTagSpecification</code>	拡張子や MIME タイプなど UTI 以外での表現

Markdown の UTI を定義する場合は「imported」となり、その定義は表 7.6 のようになります。

表 7.6: Markdown の UTI 定義 (UTImportedTypeDeclarations)

キー	値
<code>UTTypeConformsTo</code>	<code>public.plain-text</code>
<code>UTTypeDescription</code>	Markdown
<code>UTTypeIdentifier</code>	<code>net.daringfireball.markdown</code>
<code>UTTypeTagSpecification
/public.filename-extension</code>	<code>md,mkd,mkdn,mdown,mkdown,markdown</code>
<code>UTTypeTagSpecification
/public.mime-type</code>	<code>text/markdown</code>

7.4.3 ルート View Controller の設定

Info.plist の設定ができたら、`UIDocumentBrowserViewController` のサブクラスを作って、アプリのルート View Controller として設定します。`UIDocumentBrowserViewController` はルート View Controller となるように設計されていて、Document-Based App は、アプリの最初の画面が `UIDocumentBrowserViewController` になっているのが標準的なふるまいです。

`UIDocumentBrowserViewControllerDelegate` もこのサブクラスで実装するとよいでしょう。

他のナビゲーションフローでドキュメントを選択する必要があれば、`UIDocumentPickerController` を利用します。

7.4.4 UIDocumentBrowserViewController のカスタマイズ

`UIDocumentBrowserViewController` はアプリに合わせてカスタマイズできます。

表示ドキュメント形式の絞り込み

表示されるドキュメントを絞り込むには、イニシャライザ `init(forOpeningFilesWithContentTypes:)` で、サポートするドキュメントの UTI を指定します。渡した UTI を持つドキュメント形式だけを表示するようになるので、アプリで扱えないファイルを隠すのに使えます。

`UIDocumentBrowserViewController` が Storyboard から初期化される場合はイニシャライザを使うことができませんが、このときは Info.plist で指定した `CFBundleDocumentTypes` を元に自動で設定されます。

サポートするドキュメントの一覧は、`allowedContentTypes` プロパティを使って取得できます。

新規ドキュメント作成の可否

ドキュメントの新規作成を行えるかどうかを決めるのは、`allowsDocumentCreation` プロパティです。`allowsDocumentCreation` が `true` の時、navigation bar に「追加 (+)」ボタンが表示され、このボタンからドキュメントの新規作成ができます。デフォルト値は `true` です。



図 7.2: ‘`UIDocumentBrowserViewController`’の新規作成ボタン

複数のドキュメントを一度に選択できるようにするには `allowsPickingMultipleItems: Bool` プロパティを `true` にします。デフォルト値は `false` です。

アピアランスの変更

アプリのテーマに合うアピアランスにカスタマイズもできます。

背景は `browserInterfaceStyle` プロパティによって変更できます。`.dark`、`.light`、`.white` の3つの値から設定できます。

ボタンなど各ユーザーインターフェイス要素の色は、`view` プロパティから `UIView` を取り出し、その `tintColor` を設定することで変更できます。

7.4.5 ドキュメントの選択

ユーザーがドキュメントを選択したら、`UIDocumentBrowserViewControllerDelegate` の `documentBrowser(_:didPickDocumentURLs:)` メソッドが呼ばれます。一般的なアプリだと、ここでドキュメントを表す新たな View Controller を作成することになります。

`allowsPickingMultipleItems` プロパティを `true` にしている場合は、引数の URL が複数になることがあります。

ところで、このメソッドに渡ってくる URL は、別なアプリの Documents ディレクトリ中のファイルかもしれませんし、クラウドストレージサービスに置かれたファイルかもしれません。ユーザーがファイルを整理したような場合は URL が変わることがあります。

このため得られた URL は、そのままでは State Restoration などで長期間保持しておくことには向きません。URL の `bookmarkData(options:includingResourceValuesForKeys:relativeTo:)` メソッドを利用してブックマークを作成・保持するようにしましょう。URL のブックマークであれば、ファイルのファイルシステム上の位置が変わった場合も追跡してくれます。

7.4.6 ドキュメントの新規作成

ユーザーがドキュメントを新規作成できるようにするには、`allowsDocumentCreation` プロパティを `true` にした上で、`UIDocumentBrowserViewControllerDelegate` の `documentBrowser(_:didRequestDocumentCreationWithHandler:)` メソッドを実装します。

`documentBrowser(_:didRequestDocumentCreationWithHandler:)` メソッドでは、新しいドキュメントを作成して一時ファイルとして保存し、その URL と `ImportMode` を引数にして `importHandler` クロージャを呼びます。

`ImportMode` の値は `.copy`、`.move`、`.none` のいずれかです。

- `.move`: URL が示すファイルを、ユーザーがドキュメントを作成したい場所へと移動
- `.copy`: URL が示すファイルのコピーを作成
- `.none`: ドキュメントを作成できない

`documentBrowser(_:didRequestDocumentCreationWithHandler:)` メソッドでは、空のドキュメントを用意してすぐに返すことも、テンプレートを元に新しいドキュメントを作ることもできます。

.copy は、たとえばテンプレートファイルから新しいドキュメントを作成するために、テンプレートファイルを新しいドキュメントのためにコピーする、という場合に使います。テンプレートを元にドキュメントを作成するには、新しい View Controller を表示させてテンプレートをいくつか示し、そこからユーザーに選択させます。そしてテンプレートの URL を .copy と共に importHandler で返します。Document-Based App では、外部のアプリからドキュメントを渡される場合もあります。

リスト 7.1: テンプレートファイルを用いたドキュメントの新規作成

```
```swift
class DocumentBrowserViewController: UIDocumentBrowserViewController {

 func documentBrowser(_ controller: UIDocumentBrowserViewController,
 didRequestDocumentCreationWithHandler
 importHandler: @escaping (URL?, UIDocumentBrowserViewController.ImportMode)
 -> Void) {
 let templateURL = Bundle.main.url(forResource: "Untitled",
 withExtension: "md")
 importHandler(templateURL, .copy)
 }

}
```

```

ドキュメントが作成される位置は、システムの「設定」アプリから、アプリごとの「書類ストレージ」設定により設定できるようになっています。iCloud Drive が有効であれば「iCloud Drive」がデフォルトの保存場所として設定されています。「この iPhone 内」または「この iPad 内」を保存場所に設定すると、アプリのコンテナの Documents ディレクトリに保存されます。

7.4.7 Open-in-place に対応する

アプリが対応するドキュメントの形式を、Info.plist の CFBundleDocumentTypes で宣言していると、システムによってアプリが開かれることができます。

このとき UIApplicationDelegate の application(_:open:options:) が呼び出され、ファイルの URL が渡されます。

外部から渡ってきたドキュメントを開くには、UIDocumentBrowserViewController の revealDocument(at:importIfNeeded:completion:) メソッドを利用します。ファイルを必要に応じて UIDocumentBrowserViewController の管理する領域に取り込むために、importIfNeeded 引数を true に設定します。ドキュメントにアクセスできるようになると、completion クロージャにファイルの URL が渡ってきます。この URL を使ってドキュメントを開きます。

基本的な流れは iOS 9 以降で変わっていませんが、UIDocumentBrowserViewController の登場によってより単純になりました。

7.4.8 ドキュメントのオープンと編集

ドキュメントの URL を得たら、データを開いて、編集・保存することになります。

ドキュメントの URL は、他アプリのコンテナ内のファイルの場合は Security-scoped URL になっています。

Security-scoped URL が指すファイルにアクセスを得るには、URL の `startAccessingSecurityScopedResource()` メソッドを呼び、ファイルのアクセス開始を宣言しなければいけません。そうしないとファイルにアクセスできないためです。さらに、ファイルのアクセスが必要なくなったら、`stopAccessingSecurityScopedResource()` メソッドを呼び出し、アクセス権を放棄しなければなりません。このメソッドが呼び出されないとカーネルのリソースがリークし、リークが多いと、アプリは再起動するまでアクセス権が得られなくなります。

また、複数のプロセスが同時にドキュメントを参照する可能性についても考慮が必要です。複数のアプリやアプリ拡張が、同一のドキュメントファイルを同時に操作しようとすると、問題が発生します。たとえば2つのアプリが同時にドキュメントへ書き込もうとすると、ドキュメントファイルのデータが不正な状態になる恐れがあります。あるいは、ドキュメントを開こうとしている途中で、別のアプリがドキュメントを書き換えてしまうと、読み込まれたデータは破損している可能性があります。このような競合の問題を解決するために、`NSFilePresenter` プロトコルと `NSFileCoordinator` クラスが用意されています。

`NSFilePresenter` プロトコルはファイルの状態を監視するためのものです。ファイル管理を行うクラスは、このプロトコルに準拠して、ファイルの状態に応じた通知を受け取り、適切な処理を行う必要があります。

`NSFileCoordinator` クラスはファイルに対する排他的なアクセスを担うクラスで、本クラスを介してファイルの書き込み・読み込みが行えます。内部的にロックを取ることで複数のプロセスによるアクセスを調整してくれるため、同時アクセスに伴う問題を防げます。

UIDocument

前項の Security-scoped URL、`NSFilePresenter`、`NSFileCoordinator` は、必要な処理である反面正しく使うのは面倒です。しかし `UIDocument` を利用すると、内部的にはこれらの仕組みを使っていますが、実装がずっと簡単になります。`UIDocument` クラスは Document-Based App のドキュメントを抽象化するクラスです。`UIDocument` のサブクラスを作って、必要なメソッドをオーバーライドして使います。

最低限オーバーライドが必要なのは `contents(forType:)` メソッドと `load(fromContents:ofType:)` メソッドです。`contents(forType:)` メソッドは、その時点でのドキュメントの内容を `Data` または `FileWrapper` で返します。`load(fromContents:ofType:)` メソッドは `Data` または `FileWrapper` からドキュメントの内容を読み出します。

リスト 7.2: ‘`UIDocument`’を継承した Markdown ファイルのドキュメント

```
```swift
class MarkdownDocument: UIDocument {
```

```
enum DocumentError: Error {
 case incompatibleType(String?)
 case invalidContents
}

var markdown: String?

override func contents(forType typeName: String) throws -> Any {
 guard typeName == MarkdownUTI else {
 throw DocumentError.incompatibleType(typeName)
 }
 guard let string = markdown else {
 throw DocumentError.invalidContents
 }
 guard let data = string.data(using: .utf8) else {
 throw DocumentError.invalidContents
 }
 return data
}

override func load(fromContents contents: Any,
 ofType typeName: String?) throws {
 guard typeName == MarkdownUTI else {
 throw DocumentError.incompatibleType(typeName)
 }
 guard let data = contents as? Data else {
 throw DocumentError.invalidContents
 }
 guard let string = String(data: data, encoding: .utf8) else {
 throw DocumentError.invalidContents
 }
 markdown = string
}
```

UIDocument のサブクラスは `init(fileURL:)` イニシャライザでファイルの URL から初期化できます。ドキュメントを開くときは `open(completionHandler:)` メソッドを利用し、保存するときは `save(to:for:completionHandler:)` メソッドを呼び出します。

たったこれだけで、様々な問題を気にすることなくドキュメントファイルを扱うことができます。この他に、ドキュメントに変更があった時のアンドゥや自動保存にも簡単に対応できます。

#### 7.4.9 UIDocumentBrowserViewController へのアクションの追加

`UIDocumentBrowserViewController` は、`customActions` プロパティを利用すると独自のアクションを追加できます。

`UIDocumentBrowserAction` クラスで独自のアクションを作るには、アクションを一意に表す `identifier` 文字列と、アクション名の `localizedTitle` 文字列、そしてアクションが表示される場所を示す `availability`、アクションの本体となる `handler` クロージャをそれぞれ用意し、イニシャライザに渡します。`availability` には `UIDocumentBrowserAction.Availability` から

.menu と .navigationBar のどちらかまたは両方を選びます。availability の値によって、アクションはドキュメントを選択した時に表示されるメニュー や navigation bar 内に表示されます。

UIDocumentBrowserViewController でシェアボタンが押された時に表示される UIActivity もカスタマイズできます。UIDocumentBrowserViewControllerDelegate の documentBrowser(\_:applicationActivities:) メソッドで、追加したい UIActivity の配列を返すことができます。さらに documentBrowser(\_:willPresent:) メソッドでは UIActivityController のインスタンスも操作できるので、たとえば表示したくない UIActivity も指定できます。

この他、ナビゲーションバーに、追加でボタンを表示できます。ボタンを追加するには additionalLeadingNavigationBarButtonItems プロパティと additionalTrailingNavigationBarButtonItems プロパティに UIBarButtonItem を設定します。

#### 7.4.10 UIDocumentBrowserViewController のトランジション

ドキュメントが選択されたときのトランジションを、ドキュメントのプレビューが拡大していくようなアニメーションにできます。

transitionController(forDocumentURL:) メソッドで得られる UIDocumentBrowserTransitionController オブジェクトを保持しておきます。UIDocumentBrowserTransitionController は UIViewControllerAnimatedTransitioning プロトコルに準拠しています。DocumentBrowserViewController のサブクラスで UIViewControllerTransitioningDelegate プロトコルを実装し、animationController(forPresented:presenting:source:) メソッドと animationController(forDismissed:) メソッドから、保持しておいた UIDocumentBrowserTransitionController オブジェクトを返します。これだけの実装で、トランジションのアニメーションを Document-Based App らしいものにできます。

リスト 7.3: ドキュメントを開く時のトランジションを変更する

```
```swift
class DocumentBrowserViewController: UIDocumentBrowserViewController {

    private var transitionController: UIViewControllerAnimatedTransitioning?

    func presentDocument(at documentURL: URL) {
        transitionController = transitionController(forDocumentURL: documentURL)

        let storyboard = UIStoryboard(name: "Main", bundle: nil)
        let documentViewController = storyboard.instantiateViewController(
           (withIdentifier: "DocumentViewController") as! DocumentViewController

        documentViewController.transitioningDelegate = self

        documentViewController.document = Document(fileURL: documentURL)

        present(documentViewController, animated: true, completion: nil)
    }

    extension DocumentBrowserViewController: UIViewControllerTransitioningDelegate {
```

```
func animationController(forPresented presented: UIViewController,
    presenting: UIViewController,
    source: UIViewController) -> UIViewControllerAnimatedTransitioning? {
    return transitionController
}

func animationController(forDismissed dismissed: UIViewController)
    -> UIViewControllerAnimatedTransitioning? {
    return transitionController
}
```

Document-Based App を実装するのは、ここまでで説明してきたとおりですが、Xcode 9 から新しく「Document-Based App」テンプレートが用意されています。このテンプレートを利用すれば簡単に Document-Based App を作り始めることができることでしょう。

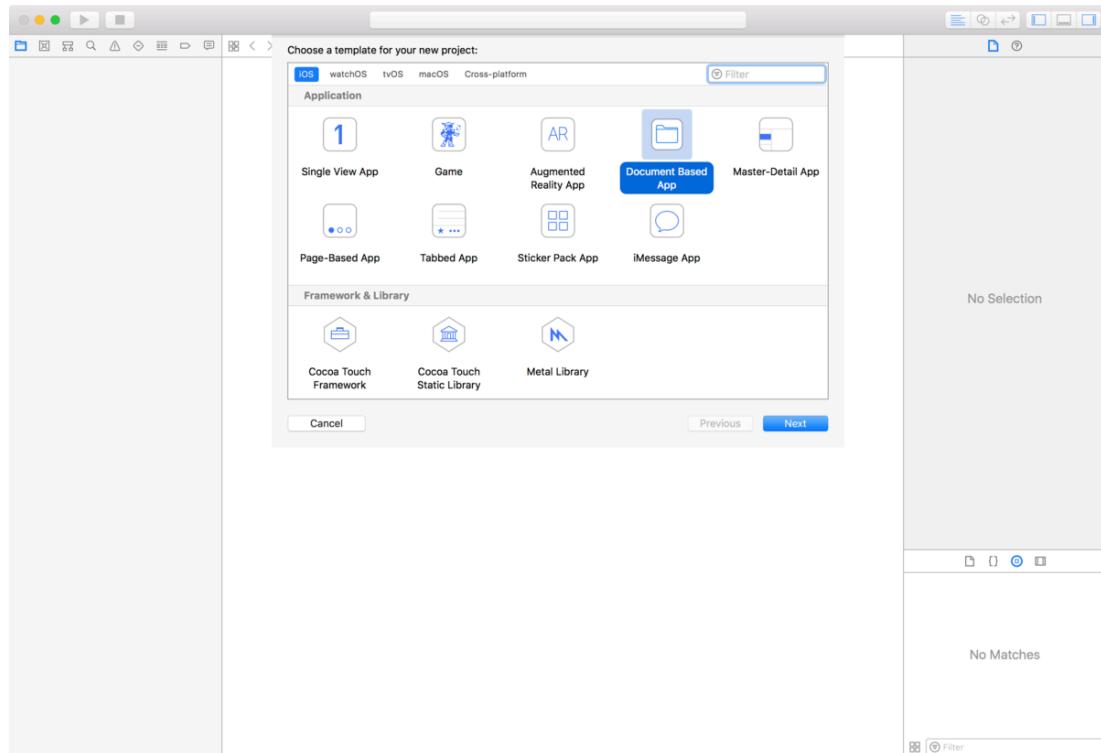


図 7.3: 「Document-Based App」テンプレート

7.4.11 Spotlight

`UIDocumentBrowserViewController` では、ドキュメントのメタデータについては自動的に Core Spotlight にインデキシングしてくれます。しかし、ドキュメントの内容は自分で Core Spotlight

にインデキシングさせる必要があります。

重複を防ぐために、独自にインデキシングする場合には、自動でインデキシングされるメタデータと同一のドキュメントを指していることを表すように、`CSSearchableItem` の `attributeSet:CSSearchableItemAttributeSet` に、`contentsURL` プロパティとしてドキュメントの URL を指定します。

7.5 Thumbnail Extension

Files アプリや `UIDocumentBrowserViewController` でサムネイルを表示する方法はいくつかあります。

古典的な方法だと、`UIDocument` のサブクラスでサムネイルを作る方法があります。`UIDocument` の `fileAttributesToWrite(to:for:)` メソッドで `URLResourceKey.thumbnailDictionaryKey` に画像を設定して返します。この方法では、特定のアプリから作られるドキュメントにのみサムネイルが生成されます。

より一般的にサムネイルを提供するために、iOS 11 から Extension の一つとして `Thumbnail Extension` が新しく加わりました。Files アプリや `UIDocumentBrowserViewController` で利用されます。

`Thumbnail Extension` は `QuickLook Framework` によって提供されます。実装にあたっては Xcode の「Thumbnail Extension」Template から始めるのが簡単です。最初に、`Thumbnail Extension` の `Info.plist` に必要な情報を加えます。

リスト 7.4: `Thumbnail Extension` の `Info.plist`

```
<key>NSExtension</key>
<dict>
    <key>NSExtensionAttributes</key>
    <dict>
        <key>QLSupportedContentTypes</key>
        <array>
            <string>com.example myfile</string>
        </array>
    </dict>
    <key>NSExtensionPointIdentifier</key>
    <string>com.apple.quicklook.thumbnail</string>
    <key>NSExtensionPrincipalClass</key>
    <string>$(PRODUCT_MODULE_NAME).ThumbnailProvider</string>
</dict>
```

`Thumbnail Extension` の extension point は `com.apple.quicklook.thumbnail` です。

`Thumbnail Extension` では、`QLSupportedContentTypes` キーに UTI の配列を設定して、サムネイルを生成する対象のドキュメントの形式を指定します。指定できる UTI は、アプリが独自に定義して `export` している UTI のみです。iOS は UTI が一致しているかどうかで `Thumbnail Extension` を選択し、UTI の継承関係は影響しません。

実際にサムネイルを提供するのは `NSExtensionPrincipalClass` に指定された `QLThumbnailProvider`

を継承したクラスです。このクラスは `provideThumbnail(for:_:)` メソッドでサムネイルを生成して返します。引数に `QLFileThumbnailRequest` が渡ってくるので、ここからサムネイルを生成するのに必要な情報を得ます。

`fileURL` プロパティにはサムネイルを生成するドキュメントの URL が格納されています。`maximumSize` と `minimumSize` プロパティにサムネイルの望ましいサイズが格納されています。サムネイルのサイズは `maximumSize` になるべく近づけ、幅か高さのどちらかが一致するようにします。`scale` プロパティは画面の scale factor に応じます。`maximumSize` に `scale` を掛け合わせたものが、スクリーン上のピクセル数と一致します。

生成したサムネイルは `QLThumbnailReply` クラスを用いて、`provideThumbnail(for:_:)` メソッドの `handler` に渡します。`QLThumbnailReply` クラスは3通りの方法で初期化できます。

`init(contextSize:currentContextDrawing:)` イニシャライザでは、第一引数 `contextSize` でコンテキストのサイズを指定し、第二引数のクロージャ `drawingBlock` で描画を行います。`drawingBlock` 中は UIKit のグラフィックコンテキストが有効なので、UIKit を使ってサムネイルを描画します。`contextSize` に `QLFileThumbnailRequest` の `scale` を掛け合わせたものが、実際に描画されるサムネイルのサイズです。UIKit の描画システムは暗黙的に `scale` を利用するので、特に意識しなくともちょうどよいサイズで描画できます。

リスト 7.5: UIKit でサムネイルを描画する

```
handler(QLThumbnailReply(contextSize: request.maximumSize,
                         currentContextDrawing: { () -> Bool in
    UIColor.black.set()
    UIBezierPath(rect: CGRect(origin: .zero,
                               size: request.maximumSize)
    ).fill()
    return true
}), nil)
```

`init(contextSize:drawing:)` イニシャライザも同様に、第一引数 `contextSize` でコンテキストのサイズを指定し、第二引数のクロージャ `drawingBlock` で描画を行います。`drawingBlock` に Core Graphics の `CGContext` が渡ってくるので、Core Graphics でサムネイルを描画します。`contextSize` に `QLFileThumbnailRequest` の `scale` を掛け合わせたものが、実際に描画されるサムネイルのサイズです。Core Graphics の描画システムでは `scale` を意識して利用する必要があります。

リスト 7.6: Core Graphics でサムネイルを描画する

```
```swift
handler(QLThumbnailReply(contextSize: request.maximumSize,
 drawing: { (context: CGContext) -> Bool in
 context.setFillColor(UIColor.black.cgColor)
 context.fill(
 CGRect(
 origin: .zero,
 size: CGSize(
```

```
 width: request.maximumSize.width * request.scale,
 height: request.maximumSize.height * request.scale
)
)
)
return true
}, nil)
```

`init(imageFileURL:)` イニシャライザでサムネイル画像の URL も指定できます。

## 7.6 Quick Look Preview Extension

Document-Based App では Quick Look Preview Extension を用意することで Quick Look にも対応できます。

ドキュメントの Quick Look は、iOS 標準アプリやサードパーティ製のアプリから、QuickLook Framework の `QLPreviewController` を介して表示されます。またドキュメントが Spotlight の検索結果として表示されるとき、3D Touch の peek を行うことでも表示されます。

実装は Xcode の「Quick Look Preview Extension」テンプレートから始めるといいでしょう。最初に Quick Look Preview Extension の Info.plist に必要な情報を加えます。

リスト 7.7: Thumbnail Extension の Info.plist

```
<!--xml
<key>NSExtension</key>
<dict>
 <key>NSExtensionAttributes</key>
 <dict>
 <key>QLSupportedContentTypes</key>
 <array>
 <string>com.example myfile</string>
 </array>
 <key>QLSupportsSearchableItems</key>
 <true/>
 </dict>
 <key>NSExtensionMainStoryboard</key>
 <string>MainInterface</string>
 <key>NSExtensionPointIdentifier</key>
 <string>com.apple.quicklook.preview</string>
</dict>
```

Quick Look Preview Extension の extension point は `com.apple.quicklook.preview` です。

Quick Look Preview Extension でも Thumbnail Extension と同様に、`QLSupportedContentTypes` キーに UTI の配列を設定して、プレビューできるドキュメントの形式を指定します。指定できる UTI は、アプリが独自に定義して export している UTI のみです。iOS は UTI が一致しているかどうかで Quick Look Preview Extension を選択します。

さらに Spotlight からの Quick Look に対応する場合は `QLSupportsSearchableItems` を YES に

します。

Quick Look Preview Extension の機能を提供するのは `QLPreviewingController` プロトコルに準拠した View Controller です。`NSExtensionPrincipalClass` にクラス名を設定するか、もしくは `NSExtensionMainStoryboard` に Storyboard 名を設定して、`QLPreviewingController` に準拠した View Controller を指定します。

`QLPreviewController` を介した Quick Look のためには `preparePreviewOfFile(at:completionHandler:)` メソッドを実装します。引数の URL からドキュメントを取得して View Controller の view に Quick Look を表示させ、`handler` クロージャを評価して処理の終了を iOS に伝えます。

Spotlight の検索結果からの Quick Look のためには `preparePreviewOfSearchableItem(identifier:queryString)` メソッドを実装します。引数の `identifier` はユーザーが Quick Look を行なった検索結果を指しているので、これを用いて CoreSpotlight Framework の `CSSearchQuery` を初期化し、検索結果を取得します。

リスト 7.8: ‘identifier’を使って ‘CSSearchQuery’を初期化する

```
let query = CSSearchQuery(queryString: "identifier == '\(identifier)'", attributes: nil)
```

検索結果をもとに View Controller の view に Quick Look を表示させ、`handler` クロージャを評価して終了します。引数の `queryString` に、ユーザーが Spotlight 検索を行った時の検索クエリが渡ってきてるので、ドキュメント中の関連する位置をハイライトするなどするとよいでしょう。

## 7.7 おわりに

ファイルやドキュメントは、これまでの iOS にも存在していた概念です。しかし伝統的に iOS ではファイルシステムが隠蔽され、ユーザーがその複雑さに触れることのないように制御されてきました。ユーザーが触れるのはアプリであってファイルではない、というのが iOS のパラダイムだと言えます。

iOS 11 からは、ファイルという概念が今まで以上に大きな役割を持ちます。ただしそれはファイルシステム上のアイテムという意味でのファイルではなく、ユーザーが作成したドキュメントという形態を取ります。iOS 11 の Files アプリや Document-Based App は、ドキュメントを中心にアプリを利用する新しいパラダイムをもたらすことになるでしょう。

## 第8章

# レイアウト関連の新機能及び変更点

本章では、iOS 11におけるレイアウト関連の新機能、変更点を解説します。レイアウトに影響を与えるナビゲーションバーの変更、セーフエリアレイアウトガイドやスクロールビューのインセットといったAuto Layout関連の変更点、ダイナミックタイプの変更点の三点をカバーしています。iOS 11でのレイアウト関連の変更点は、派手な機能変更はほぼないものの、開発者にとって痒いところに手の届く知っておきたい変更点が多くなっています。

本章のサンプルプロジェクトは*iOS11\_samplecode*リポジトリ内にある/*chapter\_08/iOS11ProgrammingAuthors.xcodeproj*です。

### 8.1 ラージタイトルとUINavigationBar

iOS 11では、UIの改善が行われました。その中でも大きな変化の一つが、ナビゲーションバーに追加されたラージタイトルです（図8.1）。WWDC 2017のWhat's New in Cocoa Touchによると、ラージタイトルによりユーザーが今どこにいるかを視覚化できます。近年、Airbnb等でも採用されているスタイルです。さらに、検索フィールドもナビゲーションバーに統合され、デザインも変更されました。

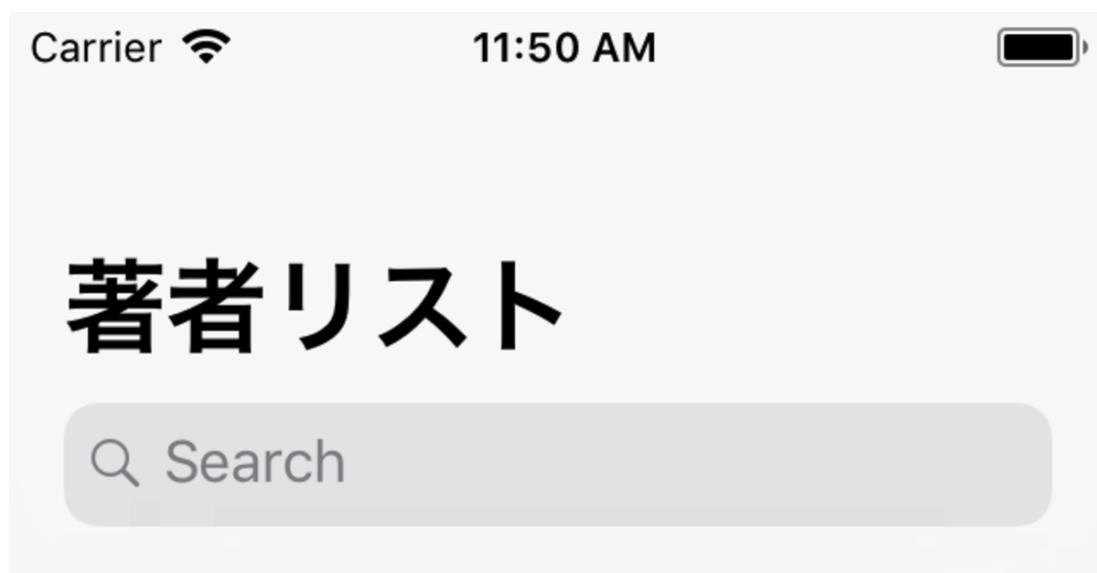


図 8.1: ナビゲーションバーに大きなタイトルを表示できる

表示領域を最大化するために、コンテンツのスクロールに合わせて、ナビゲーションバーの高さを変更できます。その場合、上むきにコンテンツをスクロールすると、ラージタイトルは過去のナビゲーションバーと似たタイトルになり、検索フィールドはナビゲーションバーに収納され、ナビゲーションバーの高さは縮小し、これまでと同様の表示領域を確保できます。



図 8.2: ラージタイトルは、スクロール挙動によりサイズが変更される

### 8.1.1 ラージタイトルをナビゲーションバーに追加

ナビゲーションバーのタイトルをラージタイトルに変更するには、`UINavigationBar` に追加された以下のプロパティを使います。

```
open var prefersLargeTitles: Bool
```

サンプルプロジェクト `AuthorListViewController.swift` では、`viewDidLoad()` 内でラージタイトルの設定がされています。

```
override func viewDidLoad() {
 navigationController?.navigationBar.prefersLargeTitles = true
}
```

プロジェクトをビルドすると、ナビゲーションバーのタイトルが（図 8.1）のように表示されます。

さらに、ナビゲーションバーは `largeTitleTextAttributes` プロパティを持ち、ここから開発者はこのラージタイトルのテキストアトリビュートディクショナリにアクセスできます。

```
var largeTitleTextAttributes: [NSAttributedStringEncoding : Any]? { get set }
```

`NSAttributedAttributedString.h` で定義されるキーを使って、ラージタイトルのシャドウやテキストカラー、フォント等を変更できます。たとえば、以下のように設定すると、ラージタイトルに指定したスタイルが適用されます（図 8.3）。

```
navigationController?.navigationBar.largeTitleTextAttributes = [
 NSAttributedStringEncoding.font: UIFont(name: "SnellRoundhand-Black", size: 30)!,

 NSAttributedStringEncoding.foregroundColor: UIColor.blue
]
```

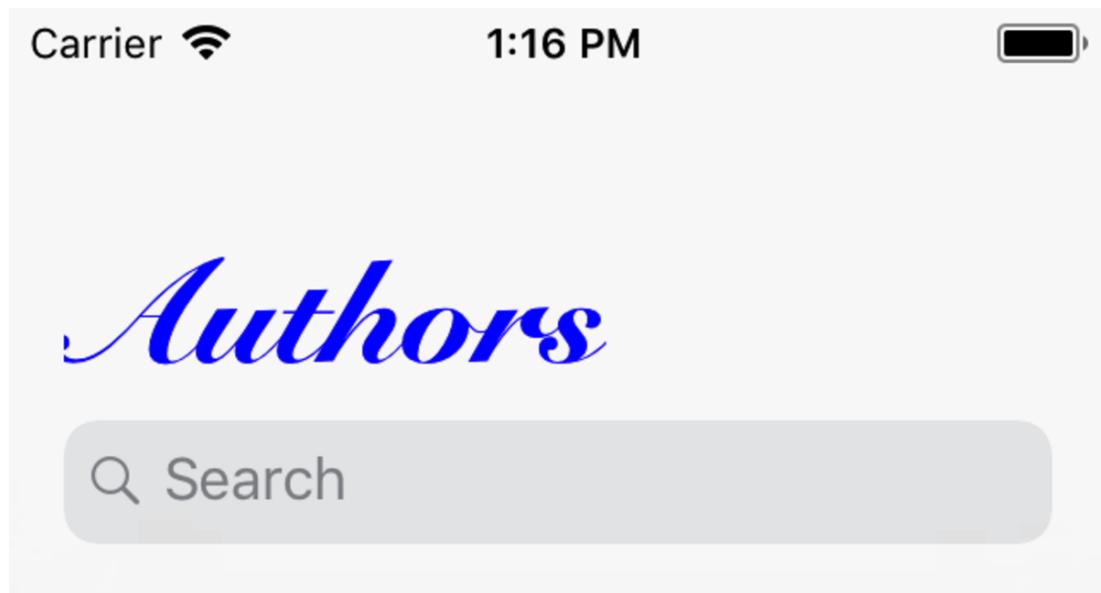


図 8.3: ラージタイトルのテキストアトリビュートを変更できる

ただし、ラージタイトルがユーザーのスクロールによって縮小モードになっているときは、この

テキストアトリビュートは適用されません。これを変更するためには、`titleTextAttributes` プロパティから別途変更する必要があります。

### 8.1.2 検索フィールドをナビゲーションバーに追加

iOS 11 では、検索フィールドをナビゲーションバーに追加できます（図 8.4）。デザインも変更されていて、iOS 10 以前と比較すると、角丸の半径が大きくなっています。



図 8.4: iOS 11 では検索フィールドをナビゲーションバーの一部として表示できる

検索フィールドを追加するには、`UINavigationItem` に追加された `searchController` に `UISearchController` オブジェクトをセットします。

```
open var searchController: UISearchController?
```

デフォルトの設定では、`searchController` の検索フィールドはスクロールによって現れたり隠れたりしますが、`hidesSearchBarWhenScrolling` で挙動を変更できます。

```
open var hidesSearchBarWhenScrolling: Bool
```

`false` にすると、検索フィールドは常にナビゲーションバー下部に固定して表示されます。

`viewDidLoad()` 内で、`UISearchController` インスタンスを作成し、ナビゲーションバーの `searchController` にセットします。次に、テキストフィールドへの入力内容の変更を受け取るために `searchResultsUpdater` にオブジェクトをセットします。

```
override func viewDidLoad() {
 let search = UISearchController(searchResultsController: nil)
 search.searchResultsUpdater = self
 searchdimsBackgroundDuringPresentation = false
 navigationItem.searchController = search
}
```

ビルトしてリストを下にスクロールすると、検索フィールドがナビゲーションバー上に表示されます。

次に、検索内容と `UITableView` 上の表示内容を関連づけましょう。`searchResultsUpdater` として指定したオブジェクトは、`searchResultsController` の入力内容に応じた変更を行います。このオブジェクトは `UISearchResultsUpdating` プロトコルに準拠している必要があります、`updateSearchResults(for:)` メソッドでコールバックを受け取ります。

```
func updateSearchResults(for searchController: UISearchController) {
 if let text = searchController.searchBar.text,
 text.count > 0 {
 filteredAuthors = authors.filter {
 $0.name.contains(text.lowercased()) ||
 $0.twitter.contains(text.lowercased())
 }
 } else {
 filteredAuthors = authors
 }
}
```

```
 tableView.reloadData()
}
```

ここでは、入力されたテキストを元に `Author` オブジェクトをフィルタリングし、表示の更新を行っています。

## 8.2 Auto Layout とレイアウト手法のアップデート

### 8.2.1 レイアウトマージンの独自定義

iOS 8 で導入されたレイアウトマージンは、システムが設定した統一的なマージンです。画面サイズに合わせて、システム側で適切な値が設定されていました。`UIViewController.view` であれば、サイズクラスによって水平方向マージンとして 16pt もしくは 20pt が設定されています。

iOS 11 では、`UIViewController.view` プロパティの `layoutMargins` を上書きできるようになりました。`UIView` クラスオブジェクトの `directionalLayoutMargins` プロパティにリスト 8.1 のように値を割り当てます<sup>\*1</sup>。

リスト 8.1: `directionalLayoutMargins` を指定しビューのマージンを調整する

```
view.directionalLayoutMargins = NSDirectionalEdgeInsets(
 top: 10, leading: 50, bottom: 10, trailing: 50)
```

この例では、上下に 10pt、leading と trailing に 50pt のマージンを設定しています。

注意点としては、システムマージンよりも小さい値を設定した場合、この設定のみではレイアウトに適用されません。`viewRespectsSystemMinimumLayoutMargins` プロパティを `false` することで、システム側が設定しているレイアウトマージンの値を無視できます（リスト 8.2）。

リスト 8.2: `viewRespectsSystemMinimumLayoutMargins` を無効にすると、システムマージンよりも小さい値が扱える

```
viewRespectsSystemMinimumLayoutMargins = false
```

### 8.2.2 セーフエリア

iOS 11 でラージタイトルナビゲーションバーが登場したことにより、回転挙動以外でもナビゲーションバーの高さが状態によって変化するようになりました。

これまで `UIViewController` の `topLayoutGuide` と `bottomLayoutGuide` プロパティに対し

<sup>\*1</sup> `layoutMargins` に `UIEdgeInsets` オブジェクトを割り当てるこことはできますが、Apple は `directionalLayoutMargins` を使うよう推奨しています。

て制約を与えることで、ナビゲーションバー やタブバーなどがコンテンツに重なる場合のレイアウトに対応してきました。一方、iOS 11 では、セーフエリアという概念が `UIView` に追加され（図 8.5）、`UIViewController` から独立してレイアウトを定義できます。このセーフエリアは、その名の通り「安全な表示領域」を示し、ナビゲーションバー やタブバーなどに隠れていないエリアを取得できます。地味な変更ながらも、不必要に複雑化していたレイアウトを平易にしました。



図 8.5: 黒枠で示された部分がセーフエリア

セーフエリアはレイアウトマージンよりも役割が明確という利点の他に、`UIView` のプロパティになったため `UIScrollView` のサブクラスが自らの表示可能領域を知ることができるようにになったという利点があります。たとえば、`UIScrollView` がナビゲーションバーの下に部分的に重なっている場合、これまで `UINavigationController` が重なっている領域を計算し、`UIScrollView` の `contentInset` プロパティに割り当てていました。しかし、iOS 11 以降では、`UIScrollView` が自らのセーフエリアを知っているため、`UINavigationController` がその値を変更する必要がなくなりました。これにより、パディングを設定するための `contentInset` が外部から自動的に変更されることはありません、開発者にとってレイアウトがよりフレンドリーになったと言えるでしょう。

### セーフエリアを有効にする

Xcode 9 でセーフエリアを使うために行う移行作業は、非常に簡単です。既存の Storyboard を開き、Interface Builder で File Inspector を開くと、Use Safe Area Layout Guides というオプションがあります。このオプションを有効にすると、Xcode が自動的にセーフエリアをビューオブジェクトに追加します。

```
--[[path = (not exist)]]--
```

### 「Use Safe Area Layout Guides」にチェックを入れる

Xcode はこの設定変更時に、`topLayoutGuide` と `bottomLayoutGuide` に対して定義していたレイアウトを `safeAreaLayoutGuide` の対応する箇所へと自動的に置き換えてくれます。また、Use Safe Area Layout Guides オプションは後方互換性があるため、ビルドターゲットが iOS 10 以前でも、このオプションを有効にできます。

### セーフエリアの特徴

`UIViewController` のプロパティである `topLayoutGuide`、`bottomLayoutGuide` とは違い、セーフエリアを表す `safeAreaLayoutGuide` は `UIView` のプロパティです。そのため、レイアウトを `UIView` の内部で完結させることができます。シーンに紐付いたビューオブジェクトでも同様にセーフエリアを設定できるため、このビューオブジェクトを `UIViewController` のルートビューに貼り付けた場合も、セーフエリアは同様の挙動をします。つまり、状態によってビューを出し分けるために、複数のビューをシーンに紐づけているような場合でも、これまで `UIViewController` でレイアウト定義をしていたのと同様の方法でレイアウトの定義ができるようになります。



図 8.6: シーンに紐付いたオブジェクトでもセーフエリアを設定可能

そのほか、safeAreaLayoutGuide の優れている点として、topLayoutGuide や bottomLayoutGuide では定義上取得できなかった高さ、幅、水平垂直方向の中心を取得できるようになりました。これにより、追加でビューを定義することなくビュー可視領域の中央に、ビューオブジェクトを配置でき、追加作業なしに画面の回転にも対応できます。

### 8.2.3 UIScrollView の変更点

iOS 11 では、UIScrollView のレイアウト指定方法が改善されました。この変更は、UIScrollView のサブクラスである UITableView、UICollectionView や WKWebView にも影響があります。破壊的な変更はありませんが、開発者をサポートしてくれる、より洗練された機能が追加されています。

#### iOS による UIScrollView のコンテンツインセット自動調整

iOS 11 では、UIViewController の automaticallyAdjustsScrollViewInsets プロパティ<sup>\*2</sup>が廃止されました。このフラグがオンの時、UIViewController はコンテンツである UIScrollView がナビゲーションバー等と重なっている場合に、contentInset の値を変更することでスクロールの位置を自動的に調整していました。

iOS 11 では、その代わりに、UIScrollView に adjustedContentInset という UIEdgeInsets の変数が追加されました。結果的に、パディングを調整するために用意されている contentInset は、その目的のためだけに使われるようになりました。

iOS 10 以前での UIScrollView の contentInset の挙動を考慮し計算している場合を除いては、開発者が特別な変更をする必要はありません。なぜならば、セーフエリア内にコンテンツが表示される

<sup>\*2</sup> Interface Builder では、Attribute Inspector 内、Layout 下にある Adjust scroll view insets から値を変更できました

ように `adjustedContentInset` が調整されるため、`automaticallyAdjustsScrollViewInsets` の値によらず、UIScrollView の表示可能領域を自動的に考慮してくれるからです。UIScrollView の `contentInset` を計算し代入している場合は、期待した挙動を得られない場合があります。そういったケースを防ぐために、以下のプロパティが追加されています。

```
var contentInsetAdjustmentBehavior: UIScrollViewContentInsetAdjustmentBehavior
```

この値は `.automatic`、`.scrollableAxes`、`.never`、`.always` を取り、デフォルトでは `.automatic` となっています。`.automatic` は、iOS 11 ではコンテンツが隠されている場合 `contentInset` を自動調整し、iOS 10 以前では `automaticallyAdjustsScrollViewInsets = true` の時と同様に上部下部の `contentInset` を自動的に調整します。`.scrollableAxes` は、スクロール可能な方向のインセットを調整します。つまり、スクロール不可能なときには調整されません。`.always` はコンテンツが隠されている場合は必ず `contentInset` を調整、`.never` はどのような場合でも調整をしません。

#### スクロール可能領域の指定するレイアウトガイド

Auto Layout を使うことで、UIScrollView のコンテンツサイズをそのサブビューのレイアウトに合わせて動的に変更できます。iOS 10 以前では、UIScrollView とそのサブビューの間に制約を与えることで、コンテンツサイズを決定できました<sup>\*3</sup>。しかし、サブビューとの制約を与える対象はあくまで UIScrollView であったため、UIScrollView の内部から制約を与えるとそのコンテンツサイズが決定されるということは明示的だとは言い難いものでした。

iOS 11 では、`contentLayoutGuide` という UILayoutGuide オブジェクトが、UIScrollView に追加されました<sup>\*4</sup>。`contentLayoutGuide` は、UIScrollView のコンテンツエリアを示すレイアウトガイドです。このレイアウトガイドにより、開発者はより明示的にコンテンツサイズの指定を記述できます。以下のように UIScrollView のサブビューと UIScrollView の `contentLayoutGuide` との間に制約を指定することでスクロール可能領域を指定できます。

```
scrollView.addSubview(label)
label.translatesAutoresizingMaskIntoConstraints = false
NSLayoutConstraint.activate([
 scrollView.layoutMarginsGuide.leadingAnchor.constraint(equalTo: label.leadingAnchor),
 scrollView.layoutMarginsGuide.trailingAnchor.constraint(equalTo: label.trailingAnchor),
 scrollView.contentLayoutGuide.topAnchor.constraint(equalTo: label.topAnchor),
 scrollView.contentLayoutGuide.bottomAnchor.constraint(equalTo: label.bottomAnchor)
])
```

<sup>\*3</sup> 実装の詳細は、拙著「よくわかる Auto Layout」の「6.5 動的なスクロール領域を持つ UIScrollView のパターン」を参照ください。

<sup>\*4</sup> iOS 11 でも、これまでと同様に UIScrollView とそのサブビューの間に制約を与える方法で、スクロール領域を指定できます。

垂直方向へのスクロールを実現するため、水平方向は UIScrollView の layoutMarginsGuide とサブビューを固定することで、これらのビューの幅を等しくしています。

垂直方向は、UIScrollView の contentLayoutGuide とサブビューを固定することで、スクロール可能領域を指定しています。注意点としては、UILayoutGuide はインターフェースビルダー上で指定できません。

#### スクロールビューのフレームを指定するレイアウトガイド

iOS 11 では、frameLayoutGuide という UILayoutGuide オブジェクトが追加されました。このレイアウトガイドは、UIScrollView のサブビューに、スクロールの影響を受けないレイアウトを与える場合に使えるレイアウトガイドです。iOS 10 でも、スクロールの影響を受けないが UIScrollView の座標内にビューを表示したい場合、スクロールのサブビューにせず、同じ階層に配置する等で対応できました。frameLayoutGuide を使うと、UIScrollView のサブビューであっても、コンテンツエリアではなく UIScrollView のフレームに対して制約を与えることができます。これにより、より明示的なレイアウト定義が可能になりました。

以下は、スクロール領域の右上にラベルを固定する例です。たとえば、ページ番号や、文章のタイトルなどの用途が考えられます。

```
NSLayoutConstraint.activate([
 scrollView.frameLayoutGuide.trailingAnchor.constraint(equalTo: titleLabel.trailingAnchor),
 scrollView.frameLayoutGuide.topAnchor.constraint(equalTo: titleLabel.topAnchor)
])
```

#### 8.2.4 UITableView の変更点

##### Self-Sizing Cell がデフォルト設定に

iOS 11 では tableView.rowHeight の初期値が UITableViewAutomaticDimension となり、動的にセルの高さが計算される Self-Sizing Cell が使われることになります。既存のプロジェクトを Xcode 9 で開く場合、特に変更を加える必要はありませんが、新たに UITableView を定義するときにはその影響を受けます。Self-Sizing Cell を無効にするためには、セル、ヘッダー、フッターそれぞれの estimatedHeight に 0 を代入する必要があります。

```
override func viewDidLoad() {
 tableView.estimatedRowHeight = 0
 tableView.estimatedSectionHeaderHeight = 0
 tableView.estimatedSectionFooterHeight = 0
}
```

Self-Sizing Cell によって決定されるセルの高さは、iOS 10 以前と同様に、セルの contentView に与えられた制約によって決定されます。

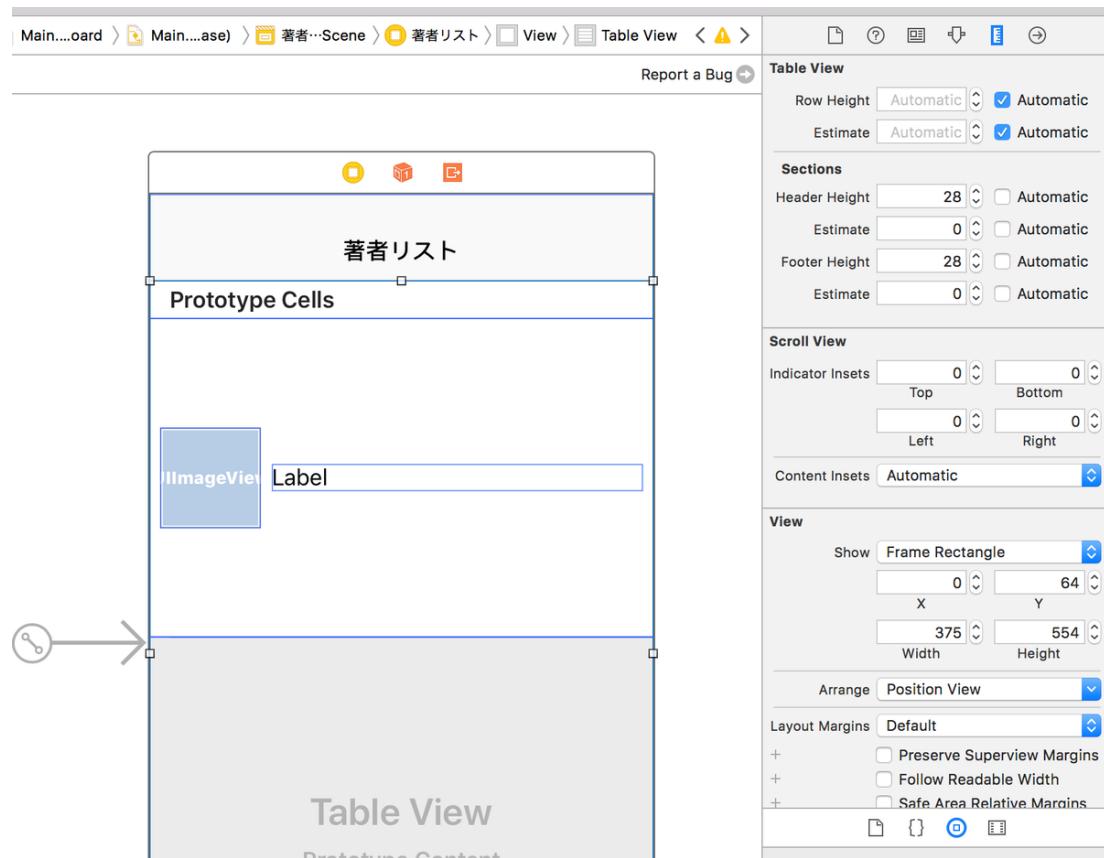


図 8.7: Interface Builder からも Self-Sizing Cell の設定が可能になった

Xcode 9 では Interface Builder からも図 8.8 のように、`rowHeight`、`estimatedRowHeight` を変更できるようになりました。セクションヘッダー、フッターに関しても同様です。これにより、UIViewController にレイアウトに関する余計な情報を記述する必要がなくなりました。なお、Interface Builder 上でも `estimatedRowHeight` が 0、もしくは `rowHeight` が Automatic 以外に設定されていると、Self-Sizing が無効になってしまふので注意が必要です\*5。

#### UITableView のインセットとセパレーター

UITableView はセパレーターのインセットを変更できる `separatorInset` プロパティを持っています（図 8.9）。iOS 10 以前では、セパレーターの位置は、リーダブルコンテンツガイド\*6 を基準に、`separatorInset` の値によって、相対的に決定されます。

\*5 著者は Xcode 8 で作成したプロジェクトをインポートし、デフォルト設定を使うためにコードでの Self-Sizing Cell 実行指定を削除した際、Self-Sizing Cell が自動で有効にならずハマりました。

\*6 リーダブルコンテンツガイドは UIView の持つレイアウトガイド変数の一つで、ユーザーが読みやすいとされる領域を定義している。



図 8.8: `separatorInset.left` はテーブルビューのセパレーター左部のインセットを示している

iOS 11 からはセーフエリアからの相対距離によってレイアウトが決定します。つまり、初期設定では `separatorInset` の値を明示的に定義した場合、セパレータはセルの端から設定した距離の位置に表示されることになります。そのため、リーダブルコンテンツガイドの値に依存することなく、レイアウトを定義できるようになりました<sup>\*7</sup>。

また、`separatorInsetReference` プロパティを使うと、`separatorInset` をシステムがどのようにレイアウトに反映させるかを変更できます。

```
var separatorInsetReference: UITableViewSeparatorInsetReference
```

初期値である `.fromCellEdges` を指定すればセルの端から、`.fromAutomaticInsets` を指定すればデフォルトのセパレータの位置から、`separatorInset` で指定した値だけセパレータが移動されます。

また、`UITableViewCell` のインセットもセーフエリアの影響を受け、セーフエリア内にコンテンツが収まるように調整されます。この時、`UITableViewCell` 自身ではなく、その `contentView` の表示領域が変更されることでインセットが調整されます。`UIView` をセクションヘッダーや `UITableView` ヘッダーに利用する例をたまに見かけますが、これらの恩恵を受けるには `UITableViewCell` もしくは `UITableViewHeaderFooterView` を使う必要があり、注意が必要です。

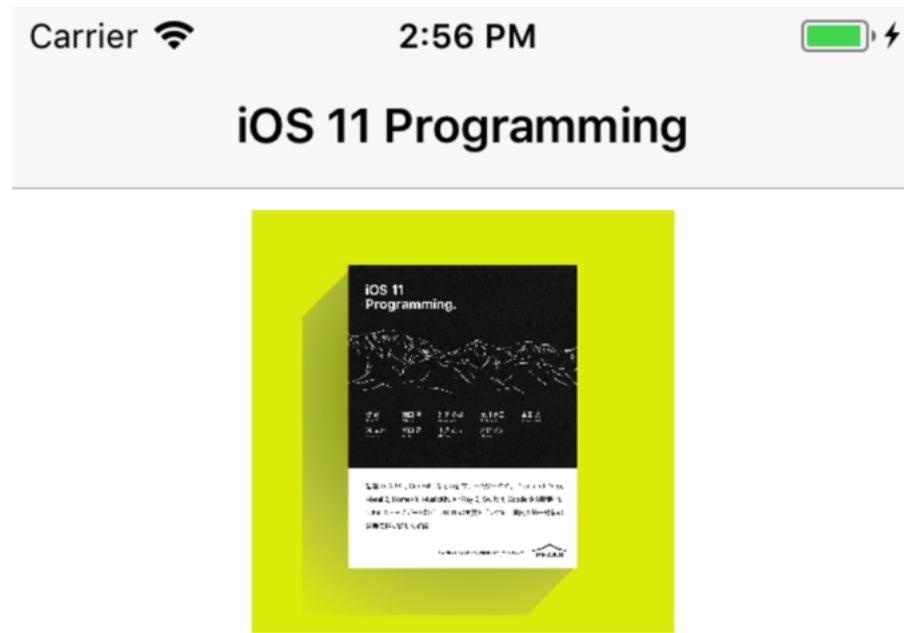
<sup>\*7</sup> セルのセパレーターの位置を調整するために `separatorInset` に負の値を割り当てたことがある人も多いのではないかでしょうか。

### 8.2.5 UIStackView の変更点

iOS 9 で追加された `UIStackView` は、縦もしくは横方向にサブビューを並べることができるビューです。`UIStackView` に並べられたビューは、設定された変数を元に Auto Layout を使ってレイアウトされます。その変数の一つの中に、`spacing` という `CGFloat` の変数があり、これは並べられたビュー間の距離変更するのに使われます。このスペースプロパティはレイアウトを定義する上で非常に便利なのですが、図 8.10 のように同一のスタックビュー内で複数のスペースを指定することはできず、ビューをネストすることで対応するのが一般的でした<sup>\*8</sup>。

---

<sup>\*8</sup> Auto Layout は、複数の連立方程式をレイヤーごとに解決していくため、ビューの階層を浅くする方が計算コストが低くなります。また、可読性も下がります。そのため、できれば不必要なネストは避けたいです。



## 第一線の開発者陣による「iOS 11 Programming」執筆プロジェクト！

購入締切日 2017年07月28日 23:59

リリース予定日 2017年10月23日

フォーマット PDF (300ページ～)



図 8.9: 1 つの ‘UIStackView’内で複数のスペースを指定する例

iOS 11 では、リスト 8.3 のようにスタックビュー内の特定のスペース幅を指定できるようになりました。以下の例では、`bookImageView` というビューの後のスペースを 30pt に指定しています。

リスト 8.3: ‘bookImageView’の直後のスペースに 30pt 追加しています。

```
stackView.setCustomSpacing(30, after: bookImageView)
```

執筆時点では `UIStackView` のカスタムスペースは Interface Builder 上からは指定できません。また、あるビューの直後のスペースしか指定できないので、直前のビューのスペースを指定する場合は、リスト 8.4 のようにビューオブジェクトを取得し、距離を指定する必要があります。

リスト 8.4: ある特定のビューの直前のスペースを調整するメソッド

```
extension UIStackView {
 func setCustomSpacing(_ spacing: CGFloat, before arrangedSubview: UIView) {
 let index = self.subviews.index(of: arrangedSubview)
 if let index = index,
 index > 1 {
 let view = self.subviews[index - 1]
 self.setCustomSpacing(12, after: view)
 }
 }
}
```

また、`setCustomSpacing(_:,after:)` メソッドで使うスペースとして、リスト 8.5 の 2 つの値が追加されました。

リスト 8.5: ‘UIStackView’に追加されたスペース指定用プロパティ

```
class let spacingUseDefault: CGFloat
class let spacingUseSystem: CGFloat
```

`spacingUseDefault` は、`UIStackView` インスタンスのスペーシングプロパティの値を指定したいときに使います。たとえば、一度スペースを変更した後、スタックビューのデフォルト値に戻したいときなどに使えます。`spacingUseSystem` は、システムで設定されたスペースを取得します。iPhone では 8pt となっていますが、OS が指定できる値なので今後変更される可能性もあります。

## 8.3 iOS 11におけるアクセサビリティ、ダイナミックタイプ関連のアップデート

iOS 7で導入されたダイナミックタイプは、ユーザーの設定に応じた文字サイズの変更を可能にする機能です。カスタムフォントが気軽に使えない<sup>\*9</sup>、レイアウトが崩れがち等の理由で敬遠されがちでした。

iOS 11では、ダイナミックタイプでカスタムフォントをサポートしたり、アクセサビリティ設定に応じた画像の拡大縮小を実行できるようになったり、アクセサビリティ対応時にデバッグをサポートするツールが整備されたりと、開発者がアクセサビリティをサポートするまでの敷居が低くなつたと言えるでしょう。

### 8.3.1 ダイナミックタイプでカスタムフォントを使う

ダイナミックタイプを使うと、システムの設定アプリで指定したフォントサイズに合わせ、アプリ内のフォントサイズを変更できます。iOS 10までは、事前に定義されたシステムフォントを使ったテキストスタイルのみ使用できましたが、iOS 11からはカスタムフォントを指定できます。

リスト 8.6: iOS 10までは、事前に定義されたテキストスタイルのフォントのみが使用可能

```
let font = UIFont.preferredFont(forTextStyle: .headline)
```

iOS 11では、UIFontMetricsが追加されました。このクラスは、@<list>(dynamicTypeios11)のように、基準となるフォントサイズを元に、設定アプリで現在選択されているフォントサイズに合わせてスケールするオブジェクトを生成します。

リスト 8.7: iOS 11では、'UIFontMetrics'クラスを使って開発者が独自定義したフォントでもダイナミックタイプに対応可能

```
let bodyFont = UIFontMetrics.default.scaledFont(for: UIFont(name: "HiraKakuProN-W6", size: 14)!)
detailLabel.font = bodyFont
```

### 8.3.2 ダイナミックタイプと行間スペースの調整

文字サイズの変更など、外的要因に対応できる柔軟なレイアウトを定義するのは、モバイルアプリエンジニアにとって大きなチャレンジです。図 8.11のようにラベル間の距離を定数で定義し、かつ、ダイナミックタイプに対応している場合、文字サイズが変更になったときにラベル間の距離を調整するには、独自の処理が必要がありました。

<sup>\*9</sup> iOS 10以前、ダイナミックタイプ自体はカスタムフォントをサポートしていませんでしたが、システムフォントサイズの変化を取得し、それをカスタムフォントに適用することで、カスタムフォントを使うことはできました。

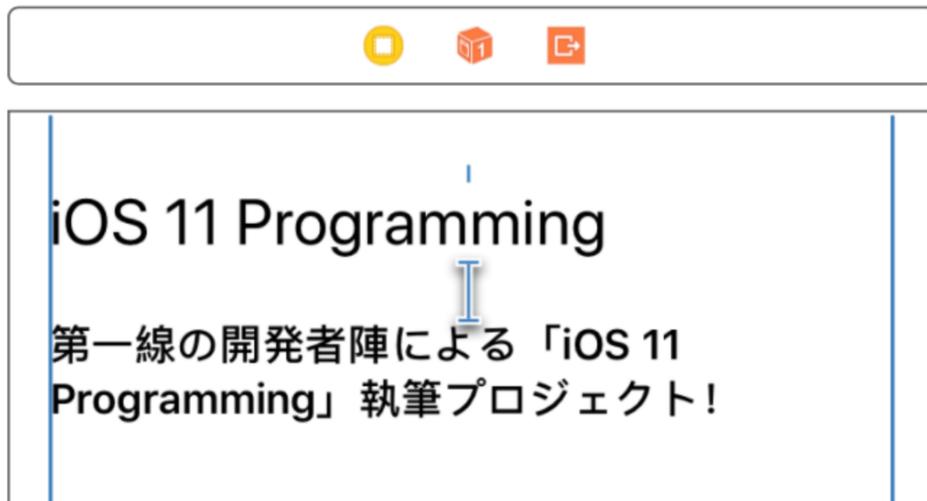


図 8.10: ラベル間の距離に定数を定義する例

iOS 11 からは、`NSLayoutYAxisAnchor` の以下のメソッドを使うと、システムのスペースを元にレイアウトの定義が可能になりました。

```
func constraintEqualToSystemSpacingBelow(_ anchor: NSLayoutYAxisAnchor,
 multiplier: CGFloat) -> NSLayoutConstraint
```

```
subTitleLabel.firstBaselineAnchor.constraintEqualToSystemSpacingBelow(
 titleLabel.firstBaselineAnchor, multiplier: 1.0)
```

`multiplier` を変更することで距離を調整できます。

### 8.3.3 アクセサビリティコンテンツサイズに合わせた画像の拡大縮小

Xcode 9 からは、ベクトル画像をラスタライズせずにプロジェクトに含めることができます。図 8.12 のように、イメージアセットの Attributes inspector から、Preserve Vector Data を選択するとベクトル情報が保存されます。WWDC2017 の Building Apps with Dynamic Type セッションによると、使用している画像が PDF の場合、画像がイメージアセットのコンパイル時にラスタライズされずに PDF のままビルドに保存されます。UIImageView や UIButton にこの画像を使用するときには PDF として保存されたベクトル情報を元に描画されるため、拡大縮小してもスムーズな画像描画が可能です。

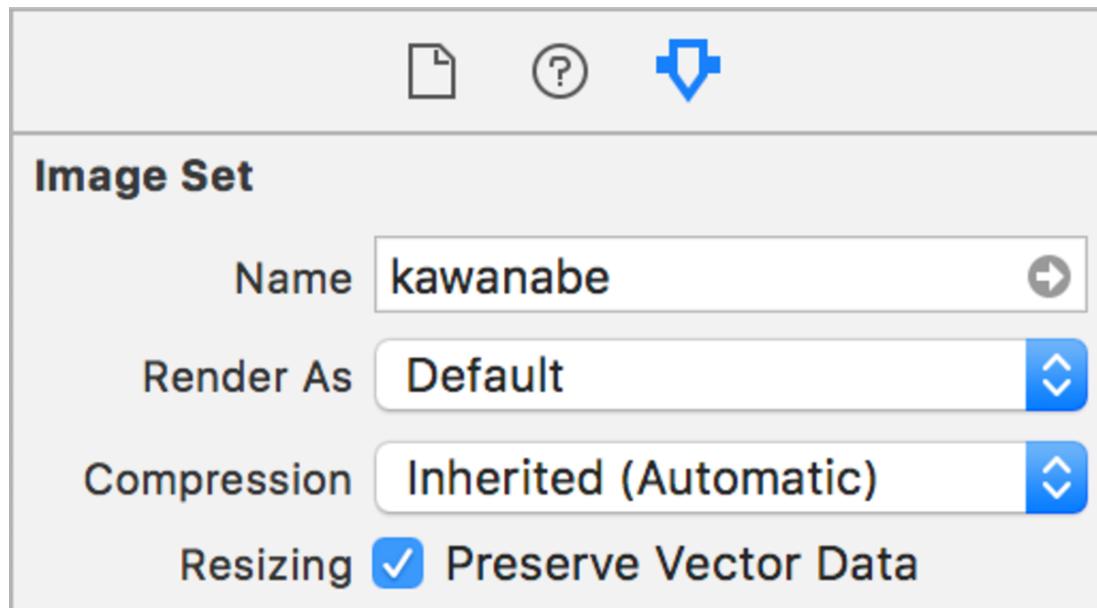


図 8.11: Preserve Vector Data を選択するとラスタライズされていない画像を使える

iOS 11 からは、ベクトル情報を保存できるようになっただけでなく、アクセシビリティのフォントサイズ設定に合わせて画像サイズを変更できるようになりました。画像は、フォントとは異なり `UIContentSizeCategory` に合わせて、五段階でサイズが変化します。この設定は Interface Builder から、もしくはコードで行うことができます。Interface Builder の場合、ビューを選択し、Attributes Inspector から設定できます。図 8.13 のように、画像を使用するビューで `Adjust Image Size` オプションを有効にすることで、画像サイズが自動的にスケールするようになります<sup>\*10</sup>。

<sup>\*10</sup> 指定した画像がベクトル画像でなくてもスケール自体は有効になりますが、もちろん拡大時の描画結果はスムーズではなくなります。

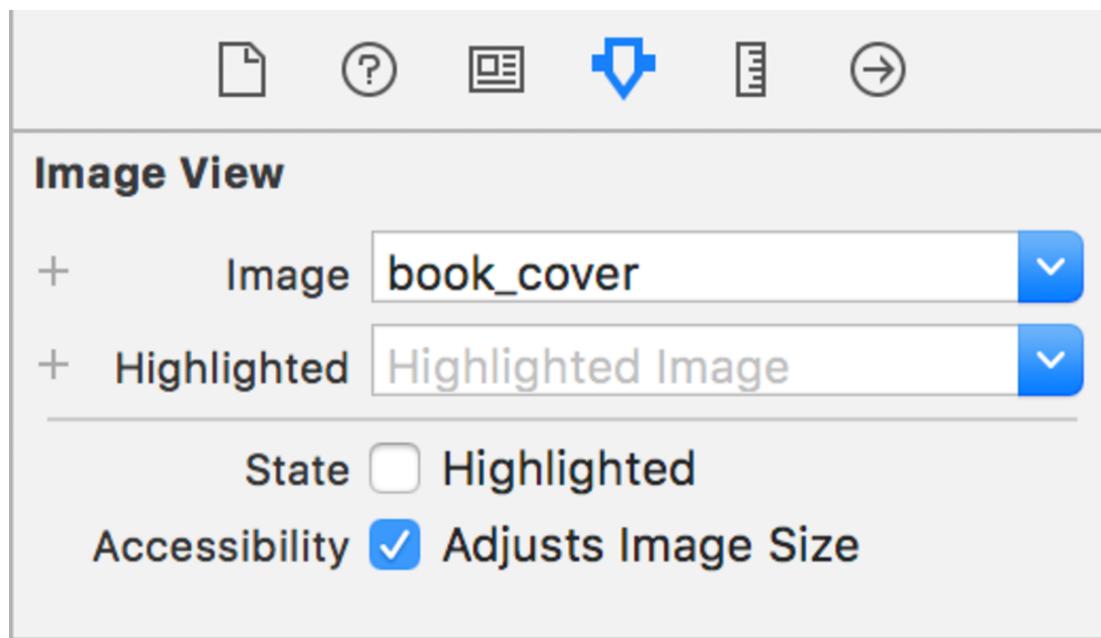


図 8.12: Adjust Image Size を有効化するとアクセサビリティコンテントサイズにあわせて画像描画サイズが変更される

コードの場合は、`adjustsImageSizeForAccessibilityContentSizeCategory` フラグを `true` にします。

```
image.adjustsImageSizeForAccessibilityContentSizeCategory = true
```

このフラグは、`UIAccessibilityContentSizeCategoryImageAdjusting` プロトコルで定義されており、このプロトコルに準拠しているオブジェクトであれば、アクセサビリティコンテントサイズに合わせた画像の縮小拡大に適応できます。執筆時点では、`NSTextAttachment`、`PKAddPassButton`、`PKPaymentButton`、`UIButton`、`UIImageView` が、このプロトコルに準拠しています。

#### 8.3.4 Accessibility Inspectorによるデバッグ

Xcode 9 では開発環境が大きく改善しました。その改善の一つである Accessibility Inspector はレイアウト関連のデバッグをサポートします。特に、フォントサイズの変更確認が非常に容易になりました。Accessibility Inspector は、Xcode > Open Developer Tool > Accessibility Inspector から開くことができます。図 8.14 のように、対象とするターゲットを選択し、右上のギアアイコンを選択すると、フォントサイズ設定を変更できるスライダーが表示されます。

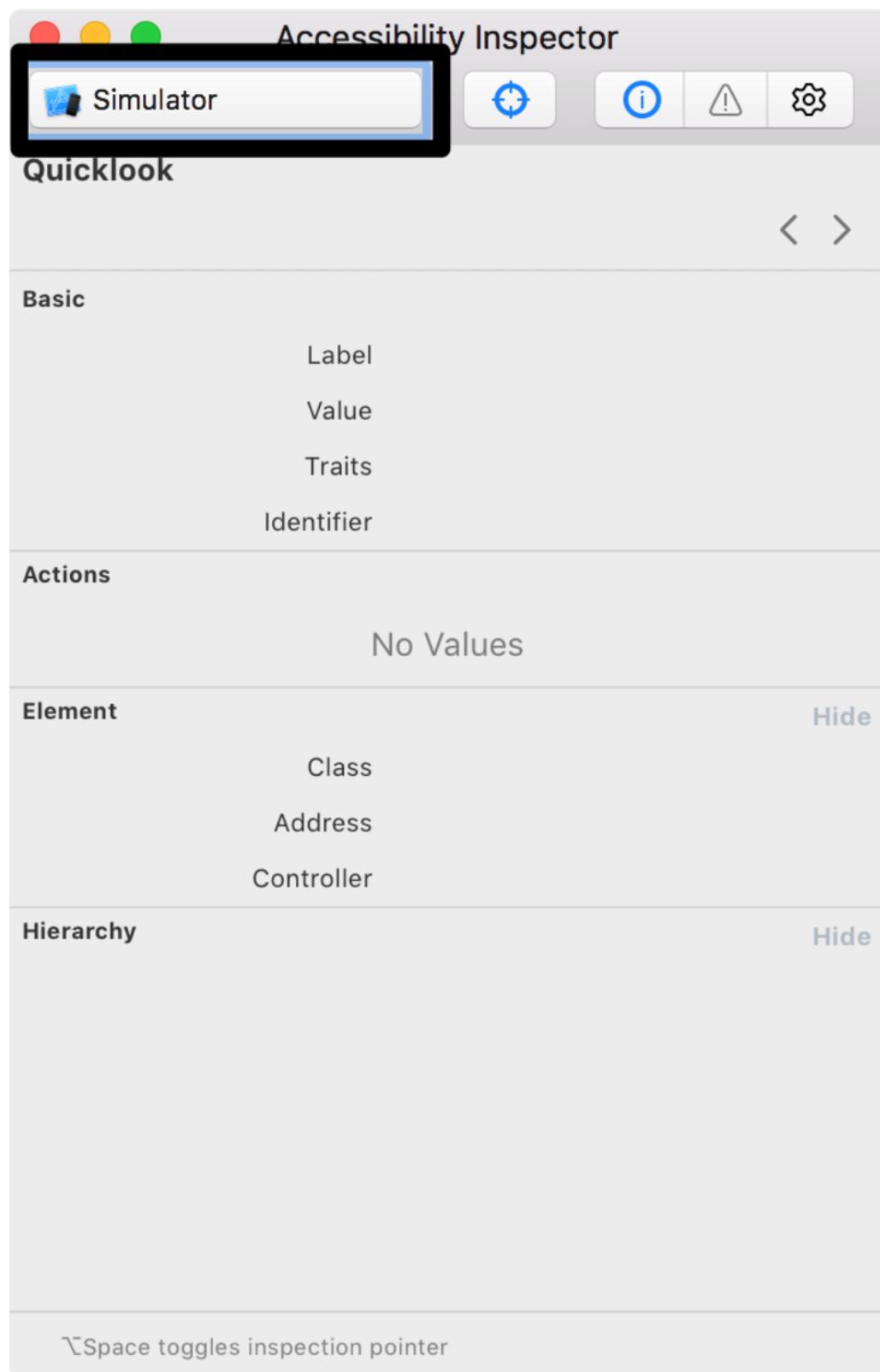


図 8.13: Accessibility Inspector を使いたいターゲットを指定する

レイアウト時に、フォントサイズ変更に動的に対応するよう設定していると、図 8.16 や図 8.16 のように、Accessibility の設定に応じてシミュレータ上の表示結果も更新されます。Xcode 8 以前だと、シミュレータでフォントサイズを変更するには、実機と同様設定アプリを開き値を変更する必要があります。しかし、Accessibility Inspector によりデバッグが非常に容易になりました。



図 8.14: Accessibility Inspector でフォントサイズを変更した例 1

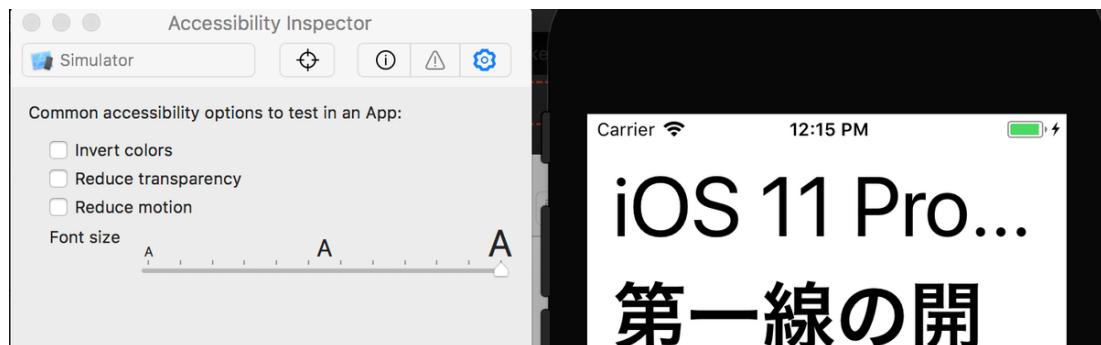


図 8.15: Accessibility Inspector でフォントサイズを変更した例 2

さらに、iOS 10 で導入された `adjustsFontForContentSizeCategory` プロパティを使えば、ユーザー設定に合わせて自動でラベルのサイズを変更できます<sup>\*11</sup>。ダイナミックタイプを便利に使うための下地が徐々に整ってきたのではないでしょうか。

## 8.4 参考文献

[22] What's New in Cocoa Touch, <https://developer.apple.com/videos/play/wwdc2017/201/>

[23] Updating Your App for iOS 11 <https://developer.apple.com/videos/play/wwdc2017/204/>

<sup>\*11</sup> iOS 9 以前でも、バックグラウンドにあるアプリがフォント設定の変更に合わせてフォントサイズを変更することは可能でした。ただし、UIViewController で通知を受け取り、ビューの更新を定義する必要がありました。

[24] Building Apps with Dynamic Type <https://developer.apple.com/videos/play/wwdc2017/245/>

[25] Asset Catalog Changes in Xcode 9 <http://martiancraft.com/blog/2017/06/xcode9-assets/>

[26] Size Classes And Core Components <https://developer.apple.com/videos/play/wwdc2017/812/>

[27] WWDC 2017: Large Titles and Safe Area Layout Guides <https://www.bignerdranch.com/blog/wwdc-2017-large-titles-and-safe-area-layout-guides/>

## 第Ⅳ部

その他の新フレームワーク、アップ  
デート

## 第9章

### Core NFC

#### 9.1 はじめに

## 第 10 章

# PDF Kit

### 10.1 はじめに

本章では、iOS 11 から新しく使えるようになった PDF Kit フレームワークについて解説します。

### 10.2 PDF Kit とは

PDF Kit は PDF (Portable Document Format<sup>\*1</sup>) の表示、データやアクセス権限の取得、簡単な編集を可能にするフレームワークです。macOS ではバージョン 10.4 (Tiger) から、iOS ではバージョン 11 から利用できます。PDF を扱うための API はもう一つ Core Graphics フレームワークの一部として CGPDF から始まるオブジェクト・関数群が提供されています。PDF Kit も内部では Core Graphics の CGPDF API を利用しています。iOS では長らく CGPDF API しか利用できませんでしたが、iOS 11 からは Mac と同様に PDF Kit も利用できます。

PDF Kit を利用できるメリットとして、テキスト選択や検索といったインタラクションを伴うリッチな表示を非常に簡単に実現できることと、データの取得が容易であることが挙げられます。CGPDF を利用して同じことを行うためには、前提条件として PDF の仕様や仕様を理解するための周辺知識（組版、フォント、Unicode、幾何学、など）を詳しく理解している必要がありました。PDF Kit では少なくとも iBooks のビューアを作るくらいであれば、PDF の仕様を意識することはありません。

表 10.1 PDFKit / CGPDF 比較

	PDFKit Music	CGPDF API
表示	○	○
編集	△	△

### 10.3 基本的な使い方

PDF Kit フレームワークの主要なクラスは `PDFView` と `PDFDocument` です。PDF を表示するには PDF ファイルの URL もしくは `Data` から `PDFDocument` のインスタンスを生成し、`PDFView` の `document` プロパティに設定するだけです。

---

<sup>\*1</sup> アドビシステムズが開発および提唱する、ソフトウェア、ハードウェア、オペレーティングシステムに関係なく、文書を確実に表示および交換するために使用されるファイル形式です。

```
import UIKit
import PDFKit

class ViewController: UIViewController {
 override func viewDidLoad() {
 super.viewDidLoad()

 let pdfView = PDFView(frame: view.bounds)

 pdfView.autoresizingMask = [.flexibleWidth, .flexibleHeight]

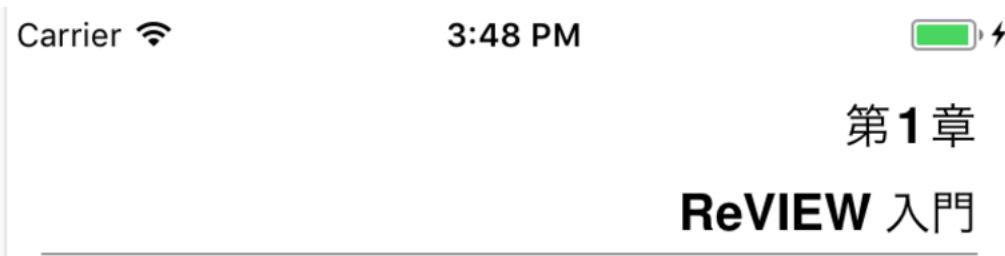
 pdfView.displayMode = .singlePageContinuous
 pdfView.displayDirection = .vertical

 let pdfDocument = PDFDocument(url: Bundle.main.url(forResource: "Sample", withExtension: "pdf")!)
 pdfView.document = pdfDocument

 pdfView.autoScales = true

 view.addSubview(pdfView)
 }
}
```

これだけのコードで図 10.1 の表示が得られます。(注意: 例のように `autoScales` プロパティを変更するのは、`PDFDocument` を代入した後に行わなければ意図した通りに動作しません)



本章では ReVIEW とは何かについて説明します。ReVIEW 記法の特徴や、それを文書の作成に用いる上での利点・注意点を紹介していきます。

### ReVIEW を用いた原稿の例

ReVIEW（「れびゅー」と読みます）の定義を説明する前にまず雰囲気を感じてもらうため、ReVIEW を用いた実例を見てみましょう。リスト 1.1 は書籍『Effective Android』<sup>1</sup>の筆者担当分の冒頭です。紙面の都合で原文にはない改行などが含まれていますが、原稿ほぼそのままです。

リスト 1.1: 『Effective Android』38 章の冒頭

```
= Google Drive API を使ってファイルをダウンロードする
```

本章では、Python スクリプトから Google Drive に保存されているファイルを取得する方法を紹介します。

Android とは直接関係ありませんが、  
同様の API を Android 上で利用する際に参考になるかもしれません。

```
== 収録されている背景
```

『Effective Android』同人誌版の執筆が行われていた頃、  
原稿は Google Drive 上で管理されていました。  
一部の技術者の希望により git も用いる運用に途中から切り替えたのですが、  
全員が git を利用できるわけではないため、  
原則 Google Drive にファイルをアップロードとし、  
git は希望者が選択して用いる、という体制になりました<sup>2</sup>。

```
//footnote[not_used] [本稿執筆時点ではすでにこの体制は終了しています。]
```

<sup>1</sup> <http://tatsu-zine.com/books/effective-android>

### 第1章 ReVIEW 入門

このとき、管理者が Google Drive 上のファイルを手動でダウンロードして  
git プロジェクトに取り込んでいると聞き、  
私はその作業を自動化できるかもしれないと考えました。

図 10.1: 基本的な使い方

PDFView は何も設定しなくても、テキストの選択やコピー、複数ページのスクロール、目次などに使われるリンクをタップすると該当のページにジャンプするといった PDF の基本的なインターフェンスは一通りサポートされています（図 10.2）。

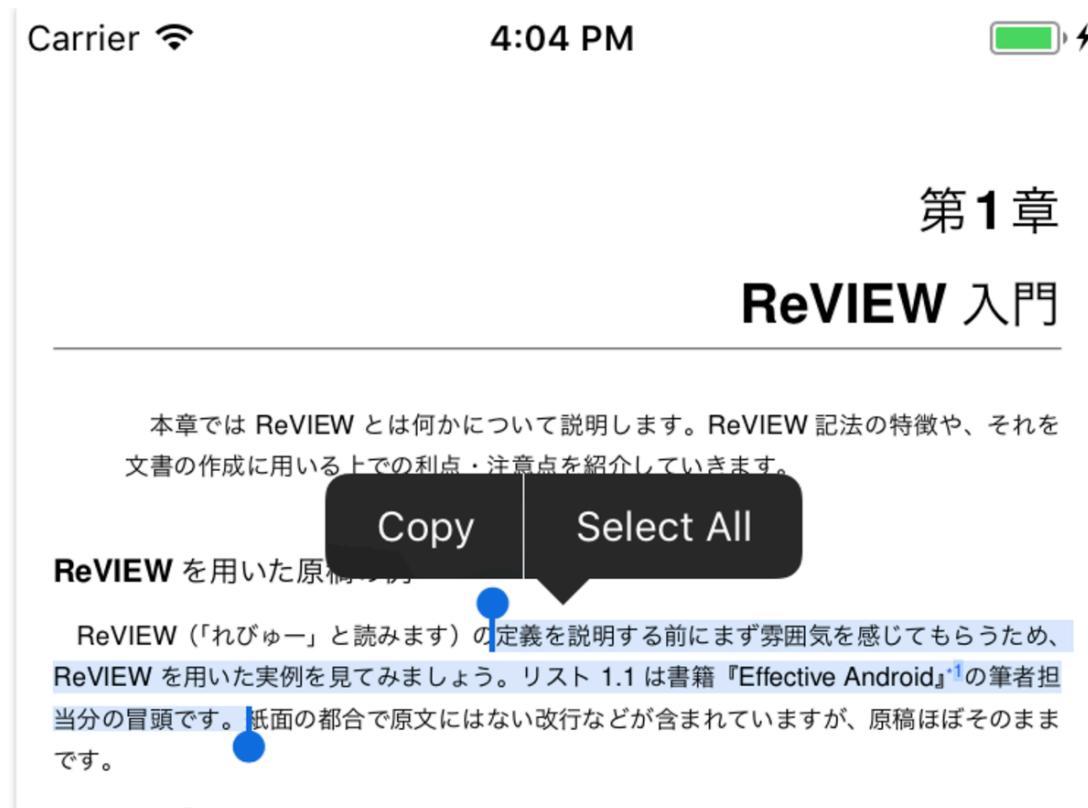


図 10.2: テキストの選択

## 10.4 PDFView

PDFView は、PDFDocument とともに PDF Kit の中心となるクラスです。PDF の表示に関する機能のすべてを担います。PDFView 使うことで任意の PDF をアプリケーションに表示できます。テキスト選択やコピー、リンククリックによるジャンプ、ページ遷移の履歴といった PDF ビューアとしての基本機能をサポートしています。PDFView を使わずに、PDF Kit の API を直接使用してより高機能な UI を実装することもできますが、ほとんどのケースでは PDFView の機能で十分でしょう。

#### 10.4.1 表示形式のカスタマイズ

displayDirection

```
var displayDirection: PDFDisplayDirection { get set }
```

タテ方向・ヨコ方向の表示を切り替えます。設定できる値は.`.vertical` または.`.horizontal` のいずれかです。デフォルトの値は.`.vertical` です。

```
@available(iOS 11.0, *)
public enum PDFDisplayDirection : Int {
 case vertical
 case horizontal
}
```

The screenshot shows a mobile device screen with a PDF document open. At the top, there are status icons: 'Carrier' with signal strength, '3:48 PM' in the center, and a battery icon with a lightning bolt to the right. The PDF page has a light gray header section containing the text 'Carrier' and '3:48 PM'. The main content area has a white background. At the top of this area, the title '第1章' (Chapter 1) is centered in a large, bold, black font. Below it, the subtitle 'ReVIEW 入門' is also centered in a bold, black font. A horizontal line separates the title from the main text. The main text is in a standard black font and reads: '本章では ReVIEW とは何かについて説明します。ReVIEW 記法の特徴や、それを文書の作成に用いる上での利点・注意点を紹介していきます。' (In this chapter, we will explain what ReVIEW is. We will introduce the features of the ReVIEW syntax and its advantages and points to note when using it for document creation.)

**ReVIEW を用いた原稿の例**

ReVIEW（「れびゅー」と読みます）の定義を説明する前にまず雰囲気を感じてもらうため、ReVIEW を用いた実例を見てみましょう。リスト 1.1 は書籍『Effective Android』<sup>1</sup>の筆者担当分の冒頭です。紙面の都合で原文にはない改行などが含まれていますが、原稿ほぼそのままです。

リスト 1.1: 『Effective Android』38 章の冒頭

```
= Google Drive API を使ってファイルをダウンロードする

本章では、Python スクリプトから Google Drive に保存されているファイルを取得する方法を紹介します。
Android とは直接関係ありませんが、
同様の API を Android 上で利用する際に参考になるかもしれません。

== 収録されている背景

『Effective Android』同人誌版の執筆が行われていた頃、
原稿は Google Drive 上で管理されていました。
一部の技術者の希望により git も用いる運用に途中から切り替えたのですが、
全員が git を利用できるわけではないため、
原則 Google Drive にファイルをアップロードとし、
git は希望者が選択して用いる、という体制になりました2。

//footnote[not_used] [本稿執筆時点ではすでにこの体制は終了しています。]
```

<sup>1</sup> <http://tatsu-zine.com/books/effective-android>

図 10.3: ‘.vertical’

Carrier ⌂ 4:01 PM



163	
164	
165	

**第1章 ReVIEW 入門**

本章では ReVIEW とは何かについて説明します。ReVIEW 記法の特徴や、それを文書の作成に用いる上での利点・注意点を紹介しています。

**ReVIEW を用いた原稿の例**

ReVIEW（「れいひゅー」と読みます）の定義を説明する前にまず雰囲気を感じてもらうため、ReVIEW を用いた実例を見てみましょう。リスト 1.1 は書籍『Effective Android』の筆者担当部分の冒頭です。紙面の都合で原文にはない改行などが含まれていますが、原稿はそのままです。

```

リスト 1.1. 「Effective Android」38 篇の冒頭
+ Google Drive API を使ってファイルをダウンロードする
本章では、Python スクリプトから Google Drive に保存されているファイルを取得する方法を紹介します。
Android とは直訳開源ありませんが、
開発の API を Android 上で利用する際に参考になるかもしれません。
-- 国語されている言葉
Effective Android は人間語版の執筆が行われていた頃、
原稿は Google のドライブ上で保管されていました。
一方で日本語の筆者たる立場で、実際に途中から切り替えたのですが、
全員が git + GitHub で開発するわけではありませんでした。
最初 Google Drive にファイルをアップロードとし、
git は再び者が選択して開く、という体制になりました。
// footnote[note_wanted] [本稿執筆時点ではすでにこの体制は終了しています。]
-- http://taku-zini.com/effective-android

```

**第1章 ReVIEW 入門**

このとき、管理脚が Google Drive 上のファイルを手動でダウンロードして git プロジェクトに取り込んでいると聞きました。私はその作業を自動化できるかもしれないと考えました。

Google Drive API との連携は Google から解説で提供されていました。  
少し調べた結果、今回の目標のためにそれを使えばよいことが分かりました。

```

* What Can You Do with the Drive SDK? sheet-migration

```

これが ReVIEW のコンパイラによって図 1.1 のような PDF に変換されます。

**1.1 ReVIEW とは何か**

この例により「ReVIEW」には実は二つの異なる側面があることがわかります。「ReVIEW 記法」と「ReVIEW ザーク」の側面です。

まず「ReVIEW 記法」について。例で紹介した文章では、いくつか日本語では一般的でない記号が含まれています。それらが、以下のように解釈されています。

- 行頭に「+」を並べると、PDF でその行は章や節と解釈される。
- 空行で区切ると段落になる。
- // footnote という命令で脚注を作ることが出来る。

人が読む文章の中に、機械が解析するための命令を含めることで、見出しや段落、フォントサイズといった情報を埋め込んでいます。このような言語は、一般に「軽量マークアップ言語」と呼ばれます。「ReVIEW 記法」はそういった軽量マークアップ言語の一つです<sup>4</sup>。

次に「ReVIEW ザーク」について。「ReVIEW 記法」で書かれた上記の文章を PDF に変換したザークもまた ReVIEW と呼ばれます。今回は PDF へ変換ましたが、PDF の他にも様々な形式のファイルを出力できます。

**4. HTML**

4 「軽量」でない「マークアップ言語」もあります。専門は専門ですが、例えば実行する HTML や LaTeX は「軽量」とはあまり呼ばれないようです。人にとって記述しやすく読みやすいと思えるのが一つのポイントのようですね。

図 10.4: ‘.horizontal’

**displayMode**

```
var displayMode: PDFDisplayMode { get set }
```

ページの表示方法を変更します。単一ページ表示.`singlePage` と見開き表示.`twoUp` があり、それぞれに特定の 1 ページ（見開きなら 2 ページ）のみの表示と、すべてのページをスクロールして表示する`singlePageContinuous`、`twoUpContinuous` があります。デフォルトの値は`singlePageContinuous` です。図 10.5 は`twoUpContinuous` を指定したときの表示です。

```
@available(iOS 11.0, *)
public enum PDFDisplayMode : Int {
 case singlePage
 case singlePageContinuous
 case twoUp
 case twoUpContinuous
}
```

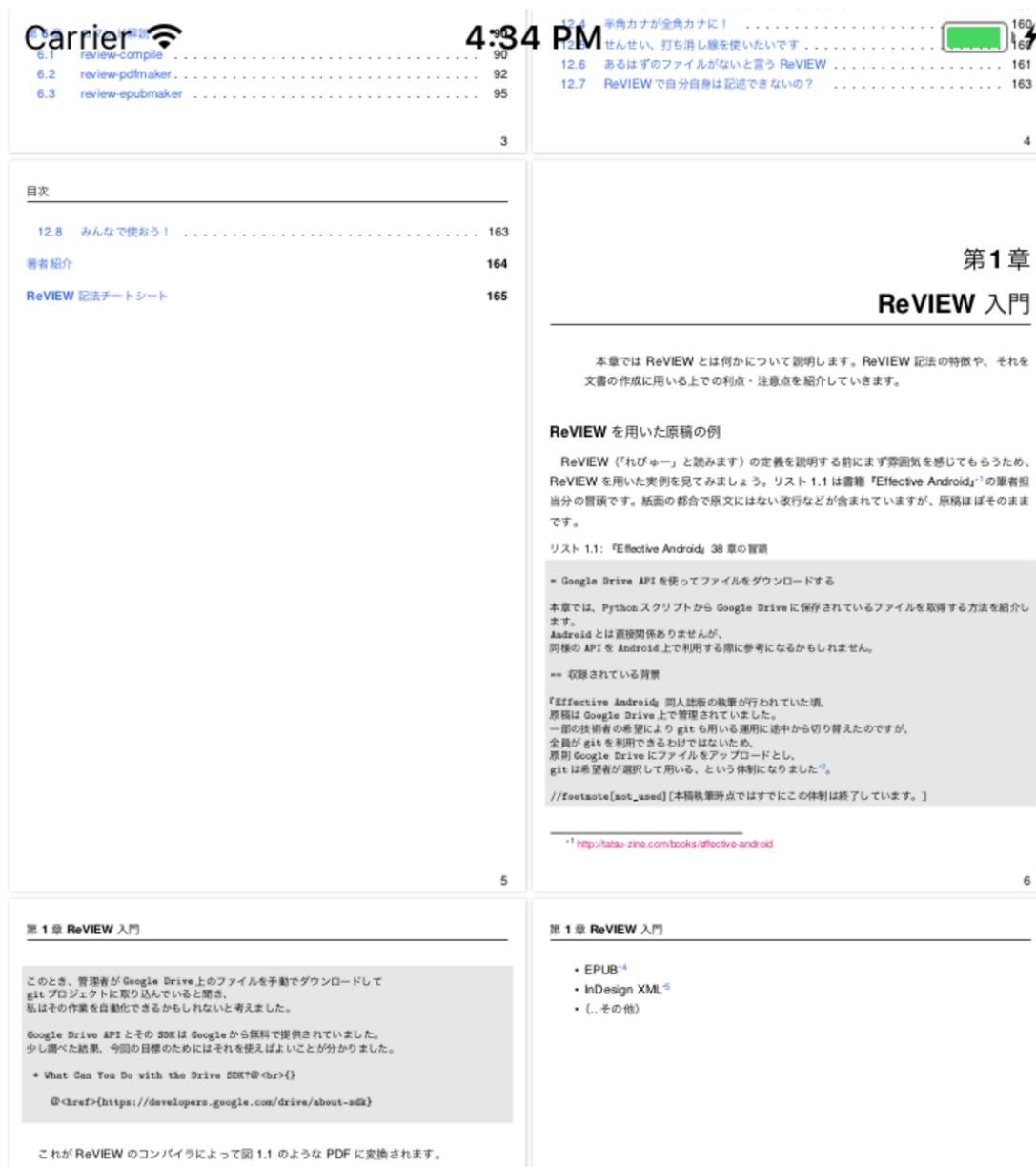


図 10.5: ‘.twoUpContinuous’

.singlePage または.twoUp を指定した場合は、自動でスクロール可能にはなりません。そのため、最初に表示したページ以外のページを表示するにはページをめくるジェスチャーやボタンを提供するなど、プログラムから何らかの手段で表示ページを変更することになります。

#### 10.4.2 ページの移動

ページをめくる機能を提供する以外にも、目次や検索結果から特定のページにジャンプしたり、Web ブラウザのようにジャンプした先から直前に表示していたページに戻りたい場合があります。そのような PDF ファイルの任意の位置に移動するためのメソッドが用意されています。

```
func go(to page: PDFPage)
func go(to destination: PDFDestination)
func go(to selection: PDFSelection)
func go(to rect: CGRect, on page: PDFPage)
```

移動先を示す引数には、任意のページを表す `PDFPage` オブジェクトや、選択範囲を表す `PDFSelection` オブジェクト、目次やアノテーションに使われる `PDFDestination` が渡せます。ページ内をさらに細かく `CGRect` を使って特定の位置に移動することもできます。

ページを前後にめくるボタンやジェスチャーを実装するための、シンプルなメソッドが用意されています。

```
func goToFirstPage(_ sender: Any?)
func goToLastPage(_ sender: Any?)
func goToNextPage(_ sender: Any?)
func goToPreviousPage(_ sender: Any?)
```

引数に `sender: Any?` を取るのは、macOS の古い形式の API によります。ボタンなどのアクションに Interface Builder を使って、そのままバインドすることを想定したシグネチャになっています。

下記はすでに先頭のページや最後のページにいるときにボタンを非アクティブにするために、前後や先頭、最後のページに移動できるかどうかを判定するメソッドです。

```
open func canGoToFirstPage() -> Bool
open func canGoToLastPage() -> Bool
open func canGoToNextPage() -> Bool
open func canGoToPreviousPage() -> Bool
```

ページを基準に指定する代わりに、Web ブラウザのように履歴を使って移動するためのメソッドも提供されています。

```
func goBack(_ sender: Any?)
func goForward(_ sender: Any?)
```

```
func canGoBack() -> Bool
func canGoForward() -> Bool
```

```
func canGoToFirstPage() -> Bool
func canGoToLastPage() -> Bool
func canGoToNextPage() -> Bool
func canGoToPreviousPage() -> Bool
```

例えば、目次のリンクやアノテーションを使って移動した場合に、ページの履歴が更新されます。直前に見ていたページに戻りたいという場合に利用します。

## 10.5 PDFThumbnailView

## 10.6 PDFDocument

PDF のファイルやデータ構造そのものを表すクラスです。PDFView と同じく、PDF Kit の中心的存在のクラスです。PDF のデータ構造にアクセスするためのさまざまなメソッドやプロパティを提供します。このクラスを用いて、PDF ファイルからテキストや目次の情報を取り出したり、テキストの検索、アクセス権限の取得、ページの追加・削除・入れ替え、といった PDF データに対する操作をします。

PDFDocument をインスタンス化するには PDFDocument のイニシャライザに PDF ファイルの URL か Data オブジェクトを渡します。

```
let pdfDocument = PDFDocument(url: Bundle.main.url(forResource: "Sample", withExtension: "pdf")!)
```

### 10.6.1 ページ情報の取得

#### 10.6.2 PDFPage

通常の PDF は 1 つ以上のページから構成されます。PDFDocument は各ページの情報を PDFPage のインスタンスとして保持しています。ページ数や PDFPage の情報は PDFDocument のメソッドやプロパティを使って取得します。

##### ページ数の取得

```
var pageCount: Int { get }
```

ページの取得

```
func page(at index: Int) -> PDFPage?
```

ページの追加

```
func insert(_ page: PDFPage, at index: Int)
```

ページの削除

```
func removePage(at index: Int)
```

ページの入れ替え

```
func exchangePage(at indexA: Int, withPageAt indexB: Int)
```

### 10.6.3 目次情報の取得

#### 10.6.4 PDFOutline

PDF ファイルの中には目次情報を持っているものがあります（図 10.6）。PDF の目次は本の目次と同様に、「章」「節」「項」といった階層構造を持ちます。目次情報は `PDFOutline` オブジェクトによって表されます。`PDFOutline` オブジェクトは、目次の階層構造を表すためにツリー構造をしています。`PDFDocument` の `outlineRoot` プロパティを使ってツリーのルートノードを取得できます。



図 10.6: ‘目次情報の例 (iBooks)’

```
var outlineRoot: PDFOutline? { get set }
```

通常、目次は複数の項目を持つので、ルートの `PDFOutline` オブジェクトは、複数の子ノードを持っています。子ノードの数は `numberOfChildren` プロパティ、

```
var numberOfChildren: Int { get }
```

子ノードそのもののオブジェクトは `child(at:)` メソッドを用いて取得します。インデックスの数字はゼロから始まります。

```
func child(at index: Int) -> PDFOutline
```

先ほどの図 10.6 に示したように、目次情報をツリー形式で表示したいことはよくあります。iBooks と同じ表示を実現するには、下記のようにツリー構造を深さ優先で探索するコードを書きます。

```
func printOutlineTree(outline: PDFOutline, depth: Int) {
 print("\(String(repeating: " ", count: depth))\((outline.label!))")
 for i in (0..
```

このメソッドに `PDFOutline` オブジェクトを渡すと、オブジェクトを起点とした目次情報の一覧を出力します。

```
if let root = pdfDocument?.outlineRoot {
 printOutlineTree(outline: root, depth: 0)
}
```

ルートオブジェクトを渡した場合、次の出力例のように先頭から順になった目次情報が得られます。

はじめに  
本書の内容について  
TechBooster とは  
お問い合わせ先  
第 1 章 ReVIEW 入門  
1.1 ReVIEW とは何か  
1.2 ReVIEW の特色  
1.3 ReVIEW の向いている分野、向いていない分野  
1.4 ReVIEW の課題  
1.5 まとめ

第2章 環境構築  
2.1 ReVIEW 環境の構成  
2.2 Mac での環境構築  
2.3 Linux での環境構築  
2.4 Windows での環境構築

第3章 執筆を始める  
3.1 プロジェクトを作成する  
3.2 文を修飾する  
3.3 PDF に変換する  
3.4 EPUB ファイルを作る  
3.5 おわりに

第4章 記法をおぼえよう  
4.1 ReVIEW 記法  
4.2 インライン命令とブロック命令  
4.3 見出し  
4.4 リード文  
4.5 段落と改行  
4.6 コメント  
4.7 箇条書き  
4.8 リスト  
4.9 コマンドライン  
4.10 引用  
4.11 リンク  
4.12 脚注  
4.13 図  
4.14 表  
4.15 文字の装飾

第5章 スタイル解説  
5.1 スタイルファイルの種類  
5.2 ページサイズを変更する  
5.3 目次に表示する項目をカスタマイズする  
5.4 余白を調節する  
5.5 章のページ起こしを指定する  
5.6 ヘッダとフッタをカスタマイズする  
5.7 既存のスタイルを変更する  
5.8 本の構成を変更する  
5.9 まとめ

第6章 コマンド解説  
6.1 review-compile  
6.2 review-pdfmaker  
6.3 review-epubmaker  
6.4 review-preproc  
6.5 review-init  
6.6 その他のコマンド

第7章 Sublime Text 2 で ReVIEW を書く  
7.1 Sublime Text 2 用 ReVIEW プラグインをインストールする  
7.2 ReVIEW プラグインのシンタックスハイライトを使う  
7.3 ReVIEW プラグインの入力補完を使う

第8章 仕様書を作ろう  
8.1 Suica Reader の仕様書を ReVIEW で書いてみる  
8.2 まとめ

第9章 ReVIEW.js で学ぶ ReVIEW 記法のお約束  
9.1 ReVIEW.js ってなに？  
9.2 ReVIEW 記法を調べよう！

第10章 同人誌を作ろう  
10.1 企画と募集  
10.2 執筆 -しんちょくどーですかー？

```
10.3 編集 ーしめきりそこですよ？
10.4 入稿と頒布
第11章 同人誌的 ReVIEW 関連用語開発
第12章 トラブル集
12.1 review-init が RubyGems 版で壊れているとか古いとか
12.2 「==タイトル」と原稿に書いたら ReVIEW のコンパイルが止まらなくなったり
12.3 TeX Live をインストールしたのにフォントがない！
12.4 半角カナが全角カナに！
12.5 せんせい、打ち消し線を使いたいです
12.6 あるはずのファイルがないと言う ReVIEW
12.7 ReVIEW で自分自身は記述できないの？
12.8 みんなで使おう！
著者紹介
ReVIEW 記法チートシート
```

それぞれの目次のタイトル文字列は `label` プロパティで取得します。目次は対応するページの情報を持っています。

```
var destination: PDFDestination? { get set }
```

目次の参照先は `PDFPage` ではなく、`PDFDestination` という `PDFPage` の特定の座標を示すオブジェクトとして表されます。目次の参照先はページだけでなく、「節」や「項」などページ内の特定の箇所や特定の図表を指す場合もあるためです。`PDFDestination` は参照先の `PDFPage` オブジェクトに加え、ページ内の特定の位置を示す `point` プロパティを保持しています。

```
weak var page: PDFPage? { get }
var point: CGPoint { get }
```

テーブルビューの特定の目次をタップすると対応するページにジャンプするといった動作は、`PDFDestination` オブジェクトを `PDFView` の `go(to: PDFDestination)` メソッドに渡すだけで実現できます。

```
func go(to destination: PDFDestination)
```

### 10.6.5 テキストの抽出

PDF Kit によって提供されるもっとも強力な機能の1つが、テキストの抽出です。`PDFDocument` の `string` プロパティから、PDF ファイルに含まれるすべてのテキストを取得できます。

```
var string: String? { get }
```

ただし、このプロパティの取得は PDF に含まれるテキストの量によっては非常に長い時間がかかります。そのため、ほんの数ページしかないことがわかっている場合を除き、このプロパティを使うことは少ないでしょう。その代わりに `PDFPage` にも同様に `string` プロパティを持っているので、こちらを使います。こちらは PDF ファイル全体ではなく、ページごとのテキストが取得できます。電子書籍ビューアのように、不特定多数の PDF ファイルを扱う場合は、こちらを利用する方が適しています。

また、`PDFPage` には `attributedString` プロパティが提供されています。こちらは PDF として表示したときとほとんど同様の表示を実現できるような `NSAttributedString` オブジェクトを返します。PDF として表示したときのフォントやスタイル情報を属性として含んでいます。

```
var attributedString: NSAttributedString? { get }
```

`attributedString` を使う場合、取得されるテキストの品質は PDF のデータに依存することに注意してください。これは、PDF というファイルフォーマットがそもそも段落や行といった文書構造の概念を持たないことによります。PDF はページを見た目通りに印刷するためのフォーマットに過ぎず、文字の位置は座標として表されます。文書構造は存在しないので、たいていの文書は見た目の通りに上から下、左から右に向かって出現するという前提をもとに、ある程度のまとまりを行や段落として扱っているだけです。

このことは、例えば行をまたぐ単語をコピーする場合に問題となることがあります。

The screenshot shows an iPhone 7 simulator running iOS 11.0. The top status bar displays the time as 6:38 PM and battery level. The main screen is a document titled "第1章 ReVIEW 入門". The content discusses the history of ReVIEW and its use in commercial publishing. It highlights the software's extensibility and its ability to handle various output formats like HTML, EPUB, and InDesign XML. The text also mentions the challenges of working with legacy markup languages like Markdown and TeX. A callout bubble points to a section about TeX. At the bottom of the page, there is a note about the pronunciation of "TeX". The footer of the page includes a "Copy" button and a "Select All" button.

開発開始時から現在に至るまで、書籍の編集・出版で実際に使われることを意識して作られてきたわけです。

商業出版の実績も多数あります。武藤氏の所属するトップスタジオでは、すでに 50 冊程度の書籍の編集で ReVIEW を使用しているそうです。また、高橋氏の主宰する達人出版会では、自社で出版している PDF・EPUB の制作用フォーマットとして ReVIEW を採用しています。

拡張性に優れている Copy Select All

商業出版の現場に対応するため、ReVIEW は拡張性の高い柔軟な構文を採用しています。出版の現場では、HTML のみならず、EPUB や InDesign XML といった複数種類のフォーマットに向けて出力を行う必要があります。そういう状況に 1 種類の原稿データで対応したり、それも原稿の書きやすさをなるべく落とさないように……そういう要求に ReVIEW は応えられます。

ReVIEW は、出力形式の特長を引き出すための独自構文を追加できる上、そうした追加を行なっても他のケースへの悪影響を抑えることができます。書いた後から出力先が増えた場合にも、元の原稿に新しい出力についての情報を書き足せば良いのです。

一方、他の軽量マークアップ言語では、しばしばアウトプットとして HTML フォーマット 1 種類に特化して作られているため、こういった形で拡張を行うことがしばしば難しくなります。軽量でないマークアップ言語は習熟に時間がかかるため、そもそも原稿を書くのが大変です。

既存のマークアップ言語で原稿を書くとこうなる

軽量マークアップ言語の Markdown を用いて技術ブログを書いていたとします。この時点では、記事の執筆は快適なはずです。

記事を書きためて人気もそこそこ、というところで、ブログ上の複数の記事をまとめて、一冊の技術者向け同人誌にしようと思い立ったとします。

Markdown は紙の出力を得意としないので、紙媒体に強い TeX<sup>14</sup>などを介して PDF を生

\*14 読み方は多様らしいですが、筆者は「テフ」と読んでいます。記述は比較的面倒なため、マークアップ言語ながら「軽量」と思う人は稀な気がします。

11

第1章 ReVIEW 入門

成することになるでしょう<sup>15</sup>。TeX は HTML 出力向けではないため、TeX と Markdown の両方でデータを管理するか、一方からもう一方へ変換することで原稿を管理することになります<sup>16</sup>。

この時、少なくとも二つ、回避できない問題が発生します。

図 10.7: 行をまたぐ単語のコピー

図 10.7 のように、行をまたぐ単語を選択してコピーすると、実際にコピーされる文字列は期待に反して次のようにになります。

フォー マット

間に不自然な空白が含まれてしまっています。これは行が変わっているために、改行扱いになっているためです。iBooks や Safari、Mac の Preview アプリなどほとんどの PDF ビューアでは同様の挙動を示します。みなさんも PDF をコピーした際に、このような不自然な空白や、改行が多く含まれてしまったり、逆にすべての行が繋がってしまったり、といったことを経験したことがあるかもしれません。テキスト検索の解説でも述べますが、PDF のテキストデータに対して何らかの処理を行う場合は、この挙動が問題になることがあります。その場合は、あらかじめ不自然な空白や改行をクリーニングしておくなどの処理が必要です。

#### 10.6.6 PDF を検索する

テキストの抽出と並ぶ強力な機能として、検索があります。検索を行うメソッドは同期的に実行される `findString()` と、非同期に実行される `beginFindString()` の 2 種類があります。

```
func findString(_ string: String, withOptions options: NSString.CompareOptions = []) -> [PDFSelection]
func beginFindString(_ string: String, withOptions options: NSString.CompareOptions = [])
func beginFindStrings(_ strings: [String], withOptions options: NSString.CompareOptions = [])
```

`findString()` は PDF ファイル全体を同期的に検索するため、PDF ファイルのサイズによっては非常に長い時間がかかります。通常の利用では非同期メソッドである `beginFindString()` を利用するか、選択範囲に対してのみ検索を実行する下記のメソッドを使用します。

```
func findString(_ string: String, fromSelection selection: PDFSelection?, withOptions options: NSString.Com-
```

`beginFindString()` メソッドは、バックグラウンドスレッドで検索を実行し、進捗や結果をデリゲートメソッドによって通知します。検索結果や進捗を取得するには、必要に応じて `PDFDocumentDelegate` のメソッドを実装します。

```
optional func didMatchString(_ instance: PDFSelection)
optional func documentDidBeginDocumentFind(_ notification: Notification)
optional func documentDidBeginPageFind(_ notification: Notification)
optional func documentDidEndDocumentFind(_ notification: Notification)
```

```
optional func documentDidEndPageFind(_ notification: Notification)
optional func documentDidFindMatch(_ notification: Notification)
```

検索対象の文字列が見つかるたびに、`didMatchString(_: PDFSelection)` または `documentDidFindMatch(_: Notification)` が呼ばれます。どちらを使っても取得できる情報は変わりませんが、マッチした箇所が `PDFSelection` 型のオブジェクトとして渡される `didMatchString(_:)` の方が使い勝手が良いです。`Notification` 型のオブジェクトを引数に取るようになっているのは、macOS の API の設計が古いことが理由です。

試しに、iBooks のようにテキストを検索し、見つかった文字列が含まれるページにジャンプ、さらには見つかった文字列をハイライトする、という処理を実装してみましょう（図 10.8）。



図 10.8: 文字列の検索

テキストの検索を開始するには、これまで説明したように `PDFDocument.beginFindString(_:withOptions:)` メソッドに検索したい文字列を渡します。2番目の引数には必要に応じて `NSString.CompareOptions` を渡します。大文字小文字の無視する検索など、通常の文字列比較で使用できるオプションが渡せます。

メソッドを実行すると、文字列が見つかるたびに `didMatchString(_:)` メソッドが呼ばれます。検索結果をテーブルビューに表示するためには、見つかった文字列の場所の情報を保持している `PDFSelection` オブジェクトを `Array` などを使って保持します。

ここで得られる `PDFSelection` オブジェクトは、検索対象の文字列とピッタリ一致する位置情報を持っているわけですが、検索結果に検索結果の画面には、iBooks のように見つかった検索語の周辺の行や、どのページやセクションに含まれているのかを利便性のために表示したいことがあります（図 10.9）。



図 10.9: 文字列の検索

PDF Kit には、そのようなときに必要に応じて選択範囲を拡げることができる `extendForLineBoundaries()` メソッドが用意されています。

```
func extendForLineBoundaries()
```

このメソッドを使うと、レシーバの `PDFSelection` オブジェクトが変更されることに注意してください。

PDF ファイルが目次情報を持っている場合、`PDFSelection` オブジェクトがどの目次に含まれるのかを知るには `PDFDocument.outlineItem(for:)` を使用します。

```
func outlineItem(for selection: PDFSelection) -> PDFOutline?
```

検索結果の一つをタップして検索結果が含まれるページにジャンプするには、`PDFView.go(to:)` に `PDFSelection` オブジェクトを渡します。このとき、`PDFSelection` の `color` プロパティを設定し、`PDFView` の `currentSelection` プロパティに代入すると、検索対象の文字列をハイライトして表示します。

```
selection.color = .yellow
pdfView.currentSelection = selection
pdfView.go(to: selection)
```

「テキストの抽出」でも説明したように、PDF には文書構造が存在しないので、単語が行をまたぐような場合に意図した通りに検索されないことがあります。



図 10.10: 行をまたぐ単語にマッチしない例

このような検索の問題は iBooks や、Mac の Preview アプリ、DropBox の PDF ビューアなどほとんどのアプリで発生しています。ですので、この挙動を受け入れるということも選択肢の一つではあります。もし、もっと厳密に正確な検索を提供したい場合は、「テキストの抽出」セクションで説明しているようにあらかじめクリーニングしたテキストデータに対して検索するなど、別の方針を検討してください。

## 10.7 PDFSelection

## 10.8 PDFAnnotation

## 第 11 章

# SiriKit

本章では SiriKit の基本と、iOS 11 で SiriKit から使えるようになった「To-Do 管理とメモ帳」と「QR コード」の 2 つのドメインについて解説します。

### 11.1 SiriKit とは

SiriKit は自分のアプリを Siri と連携させるための API です。Siri に話しかけることでアプリの機能を利用できます。SiriKit を Intents Extension 上で使うことによって、アプリを起動せず音声だけでアプリを操作できます。例えば、メッセージアプリなら Siri に「〇〇でメッセージを送って」と言うことで、メッセージを送信できます。

ただし、どんな内容でもアプリを操作できるわけではなく、Siri と連携するためには、使用目的が表 11.1 に示すドメインに関係した内容でなければいけません。

表 11.1: SiriKit をアプリで利用できるドメイン一覧

ドメイン	補足
メッセージ (Messaging)	
電話をかける (VoIP Calling)	
送金および支払い (Payments)	
写真の検索 (Photos)	
ワークアウトの開始 (Workouts)	
タクシーの乗車予約 (Ride Booking)	
レストランの予約 (Restaurant Reservations)	Apple のマップ・データチームのサポートが必要
CarPlay (Car Commands, CarPlay)	自動車メーカーでなければ利用できない
To-Do 管理およびリマインダーとメモ帳 (List and Notes)	iOS 11 で追加
QR コード (Visual Codes)	iOS 11 で追加

自分の提供しているアプリがこのドメインの領域に含まれているなら、SiriKit を利用して自分のアプリを Siri と連携できます。

## 11.2 iOS 11 の変更点

iOS 11 では利用可能なドメインに、To-Do 管理およびリマインダーとメモ帳 (List and Notes)、QR コード (Visual Codes) という 2 つのドメインが新たに追加されました。

To-Do アプリやノートアプリでは、SiriKit に対応すればアプリを起動せずに、Siri から直接 To-Do タスクやメモを追加・編集できます。アドレス帳を扱うアプリや送金・支払いを扱うアプリでは、Siri に話しかけることで QR コードを Siri の画面に直接表示できます。

また既存のドメインに対する機能追加として、タクシーの乗車予約 (Ride Booking) ドメインはに「予約をキャンセル」するタスクと、「乗車サービスにフィードバックを送る」タスクが追加されました。送金および支払い (Payments) ドメインには、「送金」のタスクと「口座残高を確認する」タスクが追加されました。

### 11.2.1 To-Do 管理とメモ帳 (List and Notes) ドメイン

To-Do 管理とメモ帳 (List and Notes) ドメインでは、To-Do タスクの作成、タスクを完了にする、メモの追加と編集、検索に対応しています。To-Do タスクを作成する際には、必要に応じて期限や場所など、リマインダーの情報を追加できます。アプリでは、Siri から必要な情報を受け取り、To-Do タスクやメモのデータをアプリに保存します。

To-Do 管理とメモ帳ドメインには処理できるタスクとして次の 6 つの Intent が用意されています。

1. メモの作成 (Create Note)
2. メモにテキストを追加 (Append to Note)
3. メモ帳とタスクリストおよびリマインダーからメモやタスクを検索 (Search for Notebook Items)
4. タスクリストの作成 (Create Task List)
5. タスクリストにタスクを追加 (Add Task)
6. タスクの状態 (完了、未完了) を変更 (Set Task Attributes)

タスク	Intent
メモの作成	INCreateNoteIntent
メモにテキストを追加	INAppendToNoteIntent
メモ帳／タスク／リマインダーからメモやタスクを検索	INSearchForNotebookItemsIntent
タスクリストの作成	INCreateTaskListIntent
タスクリストにタスクを追加	INAddTasksIntent
タスクの状態変更	INSetTaskAttributeIntent

エクステンションで Siri から伝えられるタスクを扱うには、その Intent を扱うためのプロトコルを実装します。各 Intent に対して～IntentHandling というプロトコルが用意されています。エクステンションで扱いたいタスクの Intent に対応するプロトコルを実装します。例えば、Siri からメモを作成できるようにしたい場合は、メモの作成の Intent INCreateNoteIntent に対応する INCreateNoteIntentHandling プロトコルを実装します。

タスク	プロトコル
メモの作成	INCreateNoteIntentHandling
メモにテキストを追加	INAppendToNoteIntentHandling
メモ帳／タスク／リマインダーからメモやタスクを検索	INSearchForNotebookItemsIntentHandling
タスクリストの作成	INCreateTaskListIntentHandling
タスクリストにタスクを追加	INAddTasksIntentHandling
タスクの状態変更	INSetTaskAttributeIntentHandling

SiriKit のドキュメントには SiriKit で扱えるドメインの一覧が載っています。<sup>\*1</sup> さらに各ドメインの詳細ページ（例: List and Notes ドメインの詳細ページ<sup>\*2</sup>）には用意されている Intent の一覧がありますので、まずはこちらを参照し、自分のアプリが SiriKit のドメインと合致しているかどうか、Intent は何に対応すれば良いかを検討しましょう。各 Intent クラスのリファレンスページ（例: INCreateTaskListIntent クラスのリファレンス<sup>\*3</sup>）では、Intent とクラスの解説に加え、この Intent を Siri から呼び出すには何と話しかければ良いのかがわかるサンプルフレーズが各言語ごとに掲載されています。タスクリストを新しく作成する Intent (INCreateTaskListIntent) では「(《appName》で) ショッピングリストを作成してバナナと書いて」と「結婚式の準備というリストを作成」です。英語なら「Create a shopping list with bananas on <appName>」、「Start a list called Wedding Prep」と掲載されています。後述しますが、Intents Extension をデバッグする際には、テキストで Siri に与えるフレーズを記述することができます。そのため母国語以外のフレーズを確認する場合でも、ここに載っているサンプルを元に少し書き換えるだけで簡単に確認できます。詳しくは「Extension のデバッグ」セクションをご覧ください。

#### ドキュメントに記載のサンプルフレーズ例

地域|例 1|例 2|---|---| en|Create a shopping list with bananas on <appName>|Start a list called Wedding Prep zh\_CN|建立一个▣物列表添加香蕉 (在 <appName> 上)|▣建一个列表叫做婚礼准▣ ja|(<appName>で) ショッピングリストを作成してバナナと書いて|结婚式の準備というリストを作成 ko|(<appName> ) |

### 11.2.2 QR コード (Visual Code) ドメイン

QR コードのドメインでは QR コードを使った支払いと請求、または連絡先の交換ができます。アプリで QR コードが利用できる場合は、このドメインに対応することで、収容アプリ<sup>\*4</sup>を起動することなく QR コードを直接 Siri から呼び出せ、Siri の画面に表示できます。さらに、QR コードを Siri の画面から直接読み取ることもできます。読み取った QR コードの情報は、利用できるアプリに Siri から直接受け渡されます。

QR コードのタスクに対応する Intent は、iOS 11 だと INGetVisualCodeIntent の 1 つだけが用意されています。

<sup>\*1</sup> <https://developer.apple.com/documentation/sirikit/>

<sup>\*2</sup> <https://developer.apple.com/documentation/sirikit/listsandnotes>

<sup>\*3</sup> <https://developer.apple.com/documentation/sirikit/increatetasklistintent>

<sup>\*4</sup> エクステンションはアプリケーションに付属して AppStore を通じて配布されます。エクステンションを含むアプリケーションを、収容アプリケーション (Containing App) と呼びます。

タスク	Intent
QR コードによる連絡先交換や支払いの請求	INGetVisualCodeIntent

QR コードの種類として次の 4 種類が定義されています。

- `.unknown` (不明)
- `.contact` (連絡先)
- `.requestPayment` (請求)
- `.sendPayment` (送金)

例えば「自分の連絡先 QR コードを表示」と話しかけると、種類は`.contact` (連絡先) と解釈されます。「自分の支払いコードを表示」と話しかけると、`.sendPayment` (送金) と解釈されます。「自分の QR コードを表示」の場合は`.unknown` (不明) と解釈されます。

`INGetVisualCodeIntent` に対応するには、この Intent の処理に対応するプロトコル `INGetVisualCodeIntentHandling` を実装します。ドメインの Intent すべてに対応するプロトコル `INVisualCodeDomainHandling` でも構いません。現在このドメインに用意されている Intent は 1 つなので、実質的にどちらを利用しても同じです。

タスク	プロトコル
QR コードによる連絡先交換や支払いの請求	<code>INGetVisualCodeIntentHandling</code>

Intent のリファレンス<sup>\*5</sup> には、Siri によってタスクとして解釈される言葉の例が載っています。デバッグする際の参考にしてください。

ドキュメントに記載のサンプルフレーズ例

地域|例 1|例 2|---|---| en>Show my personal QR code|Display my payment code zh\_CN|示我的个人二|示我的支付| ja|自分の QR コードを表示|自分の支払いコードを表示 ko|QR |

### 11.3 動作のしくみ

SiriKit の実装に触れる前に、Siri とアプリとがどのように連携して動作するかを簡単に説明します。

#### 11.3.1 SiriKit とアプリの関係

SiriKit は `SiriKit.framework` というフレームワークがあるわけではなく、アプリと Siri を連携するための API 群の総称を指します。具体的には、SiriKit は `Intents.framework` と `IntentsUI.framework` の 2 つのフレームワークによって構成されています。

`Intents.framework` は、アプリと Siri とで相互にデータを取り取りするための橋渡しとなる仕組みを提供します。`IntentsUI.framework` は、あらかじめ用意された標準の UI ではない独自の UI を表示したいときに、必要に応じて利用します。そして、これらのフレームワークは「Intents Extension」という App Extension の中で使用します。

Siri に話しかけると、Siri が話された内容を解釈し、ドメインとタスクを決定します。アプリのケ

---

<sup>\*5</sup> <https://developer.apple.com/documentation/sirikit/ingetvisualcodeintent>

イパビリティを適切に設定し、Intents Extension の中で連携するための API 群を記述することで、ドメインとタスクが合致する Intents Extension に Siri からの呼び出しが届くようになります。

Intents Extension と収容アプリ<sup>\*6</sup>とは、アプリケーショングループ（App Group）を有効にし、共有コンテナを使ってユーザーデフォルトや Core Data によりデータを共有します（図 11.1）。

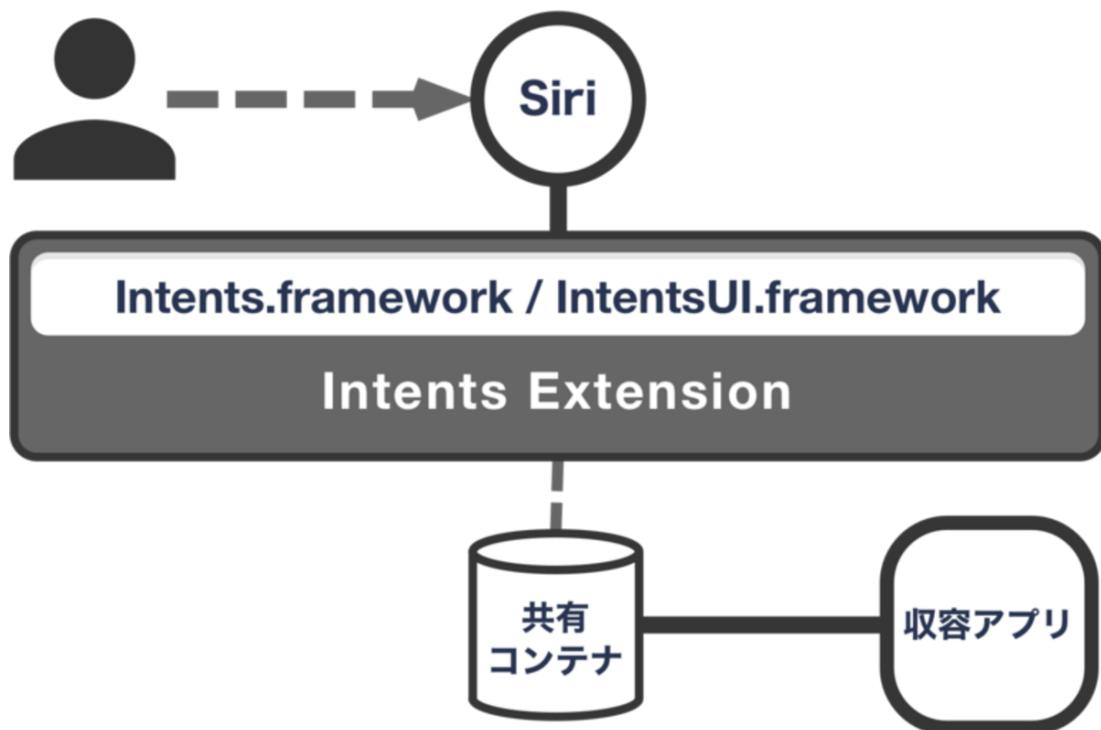


図 11.1: Siri とアプリの関係

この仕組みは、Today ウィジェットエクステンションや、キーボードエクステンションなどの App Extension と同様です。つまり、Siri と連携させるためには、アプリ本体（収容アプリ）に加え、App Extension を開発することになります。

Siri に話しかけると、Siri が話された内容を解釈し、ドメインとタスクを決定します。エクステンションが扱えるドメインとタスクであれば、そのエクステンションが呼び出されます。

### 11.3.2 Intents Extension での動作

Intents Extension で Siri からの情報を受取り処理を実行する手順は、大きく分けると次の 4 ステップに分かれます。

1. データを受け取るオブジェクト（ハンドラ）の生成
2. 不足している情報の解決（リゾルブ）

<sup>\*6</sup> エクステンションはアプリケーションに付属して App Store を通じて配布されます。エクステンションを含むアプリケーションを、収容アプリケーション（Containing App）と呼びます。

3. 処理開始前の最終確認（コンファーム）
4. 処理の実行（ハンドル）

それぞれを説明します。

### ハンドラオブジェクトの生成

Siri からの情報は、Intent オブジェクトとして Intents Extension に届きます。そのときのエントリポイントで、以降の処理を担当するオブジェクトを決定します。このオブジェクトのことを「ハンドラオブジェクト」といいます。SiriKit を使った動作は、まずこのハンドラオブジェクトの生成から始まります。

Intents Extension のエントリポイントになるのは、INExtension を継承したクラスの `handler(for:)` メソッドです。Siri によって解釈されたタスクを示す Intent オブジェクトと共に呼び出されます。

この `handler(for:)` メソッドで、Intent インスタンスの型や内容を調べ、適切なハンドラオブジェクトをインスタンス化して返すことで、以降の処理が実行されます。

ハンドラオブジェクトは、利用する Intent の末尾に～IntentHandling が付与された名前のプロトコルに準拠している必要があります。タスク生成の Intent (INCreateTaskListIntent) であれば、ハンドラオブジェクトは INCreateTaskListIntentHandling プロトコルに準拠しなければいけません。

一方で、各ドメインには用意されているすべての Intent に対応するための～DomainHandling というプロトコルも定義されています。このプロトコルは同一ドメイン内のすべてのプロトコルへ準拠したプロトコルです。用意されている Intent が多いドメインでも、たくさんのプロトコルを羅列する必要がなくなります。ただし、SDK のバージョンアップに伴って、ドメインに Intent が増えた場合は互換性がなくなることになります。

～IntentHandling プロトコルには `handle(intent:completion:)` の他に、関する 1 つ以上の `resolve～(for:with:)` と、1 つの `confirm(intent:completion:)` メソッドがあります。それぞれは次に説明するリゾルブ・コンファームで使用するメソッドです。

### 不足情報の解決（リゾルブ）

Siri に話しかけた内容は、必要な全ての情報が揃っているとは限りません。そこで、タスクを実行するために必要な情報が正しく揃っているかどうかの検証と、不足がある場合は追加の情報をユーザーに求めることができます。このフェーズをリゾルブと呼びます。

各 Intent クラスには 1 つ以上の `resolve～(for:with:)` というメソッドが用意されていて、処理がハンドルされる前に 1 回以上呼ばれます。

各メソッドで値をチェックしたり、ユーザーに確認を求めるなどの処理を行い、結果を返します。返す結果によって、次のステップに進まずに失敗扱いとすることもあります。

`resolve～(for:with:)` メソッドの実装は任意です。

### 処理開始前の最終確認（コンファーム）

リゾルブで確定した情報をもとに、処理をハンドルするかどうか、最終的な確認を行うフェーズです。ここで行うべき検証とは、たとえば複数の情報によって判断されるバリデーションや、サーバーに問い合わせる必要がある処理などです。

確認には `confirm(intent:completion:)` メソッドを使い、ここでは Intent の処理を実行できるかどうか、具体的には `handle(intent:completion:)` メソッドに進めるかどうかの最終確認を行います。

次のコードは、コンファームのフェーズで記すコードの一例です。

```
func confirm(intent: INCreateTaskListIntent,
 completion: @escaping (INCreateTaskListIntentResponse) -> Void) {
 if ... {
 let response = INCreateTaskListIntentResponse(code: .inProgress, userActivity: nil)
 completion(response)

 URLSession.shared.dataTask(with: URL(string: "...")!) { (data, response, error) in
 let response = INCreateTaskListIntentResponse(code: .ready, userActivity: nil)
 completion(response)
 }
 } else {
 let response = INCreateTaskListIntentResponse(code: .ready, userActivity: nil)
 completion(response)
 }
}
```

`intent` オブジェクトのプロパティを確認し、タスクが実行できる状態になっていればレスポンスに `.ready` をセットして `completion` ハンドラを呼びます。サーバーに問い合わせる必要がある場合は、いったん `.inProgress` を返し、準備ができた時点で `.ready` を返すなどとします。

何らかの理由によってタスクが実行できないことが判明した場合には、この時点で失敗のレスポンスを返します。

`confirm(intent:completion:)` メソッドの実装は任意です。

### 処理の実行（ハンドル）

リゾルブ、コンファームの段階で揃った情報を使って実際に処理を行うフェーズです。処理の後で、成功か失敗か、あるいは収容アプリを起動する必要があるかをレスポンスとして返します。

ハンドルフェーズで使用するのは `handle(intent:completion:)` メソッドです。このメソッドは必ず実装しなければいけません。

以上のように、ハンドラオブジェクトの作成後に `resolve~(for:with:)` メソッドによって必要な情報を収集・検証し、`confirm(intent:completion:)` メソッドでタスクを達成できるかどうかを確認し、`handle(intent:completion:)` メソッドで実際にタスクを処理するという手順は、SiriKit に共通する手順です。

## 11.4 アプリ実装の準備

SiriKit を使ったアプリを作るためには、App Extension とその収容アプリの 2 つを開発することになります。

- 収容アプリ側：Siri ケイパビリティの有効化 & Siri の利用許可取得
- Intents Extension 側：Siri との連携処理を実装

Siri から渡される情報は Intents Extension 内で扱いますが、収容アプリ側でも Siri を利用するための作業が必要です。この作業が済んでいなければ、たとえ App Extension を正しく実装して Siri に正しく話しかけていても、Siri からエクステンションが呼び出されません。

そこで本節では、収容アプリの処理や Intents Extension での Info.plist 設定など、SiriKit を使ったアプリを作るときの準備に相当する部分を説明します。

### 11.4.1 収容アプリ

#### Siri ケイパビリティを有効にする

SiriKit を使うためには、収容アプリの「Siri」ケイパビリティを有効にする必要があります。手順は iCloud・プッシュ通知・In-App Purchase などの他ケイパビリティと同様で、Xcode の GUI からケイパビリティのスイッチを ON にすると、Xcode によってエンタイトルメントファイル (~.entitlements) がプロジェクトに追加され、このエンタイトルメントを使用するようビルド設定が更新されます。OS および App Store はこのエンタイトルメントを参照し、アプリが Siri に対応していることを認識します。

具体的な手順は次のとおりです。

収容アプリの Siri ケイパビリティを有効にするには、Xcode のプロジェクト設定画面で収容アプリのターゲットを選択し、「Capabilities」タブに切り替え「Siri」ケイパビリティを ON にします。自動的にエンタイトルメントファイル (<アプリ名>.entitlements) がプロジェクトに追加されます。App Store はこのエンタイトルメントが存在することで、後述する Intents Extension がアプリに組み込まれていることを認識します。

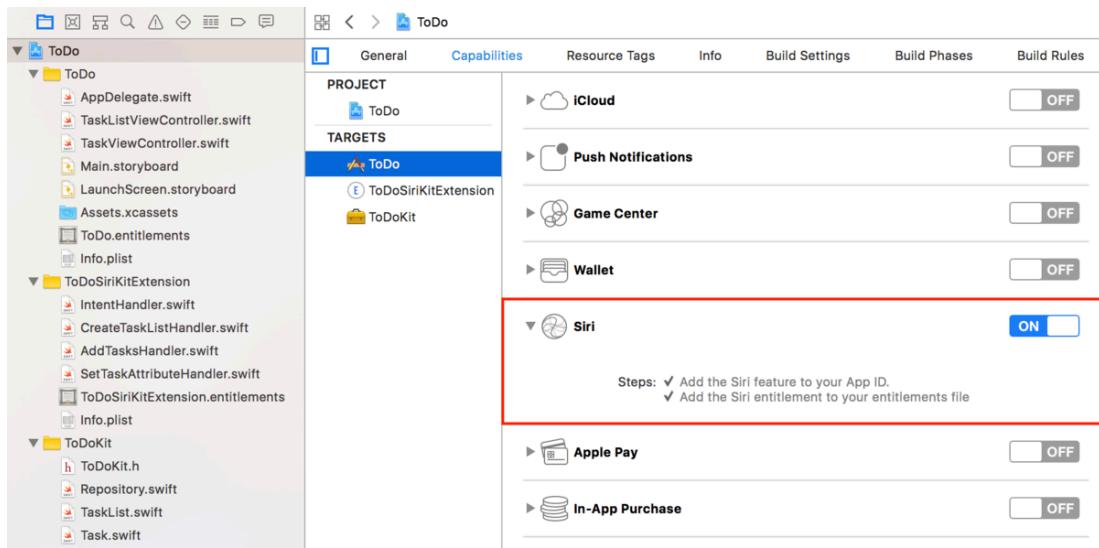


図 11.2: Siri ケイパビリティを有効にする

### Siri の利用許可を取得する

Siri がアプリを呼び出せるようにするには、ユーザーデータにアクセスするために、Siri の利用許可をユーザーに確認する必要があります。この手順はカメラや写真へのアクセス、位置情報の取得、マイクの使用などユーザーの許可が必要になる他の操作と共通です。

まず、収容アプリの Info.plist ファイルに NSSiriUsageDescription キーを追加し、アプリが Siri を利用するための目的を記述します。記述した文章は、許可を求めるダイアログ中に表示されます。

利用許可の要求は、収容アプリの実行時におこないます。つまり、アプリのダウンロード後に一度も起動されてない場合は、Siri を使ったアプリのタスク実行はできません。

許可を求めるには Intents.framework の INPreferences の requestSiriAuthorization(\_:) クラスメソッドを使います（リスト 11.1）。

リスト 11.1: requestSiriAuthorization の使用例

```
func application(_ application: UIApplication,
 didFinishLaunchingWithOptions launchOptions:
 [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
 if case INPreferences.siriAuthorizationStatus() =
 INSiriAuthorizationStatus.notDetermined {
 INPreferences.requestSiriAuthorization { status in
 switch status {
 case .authorized:
 print("authorized")
 case .denied, .restricted, .notDetermined:
 print("not authorized")
 }
 }
 }
}
```

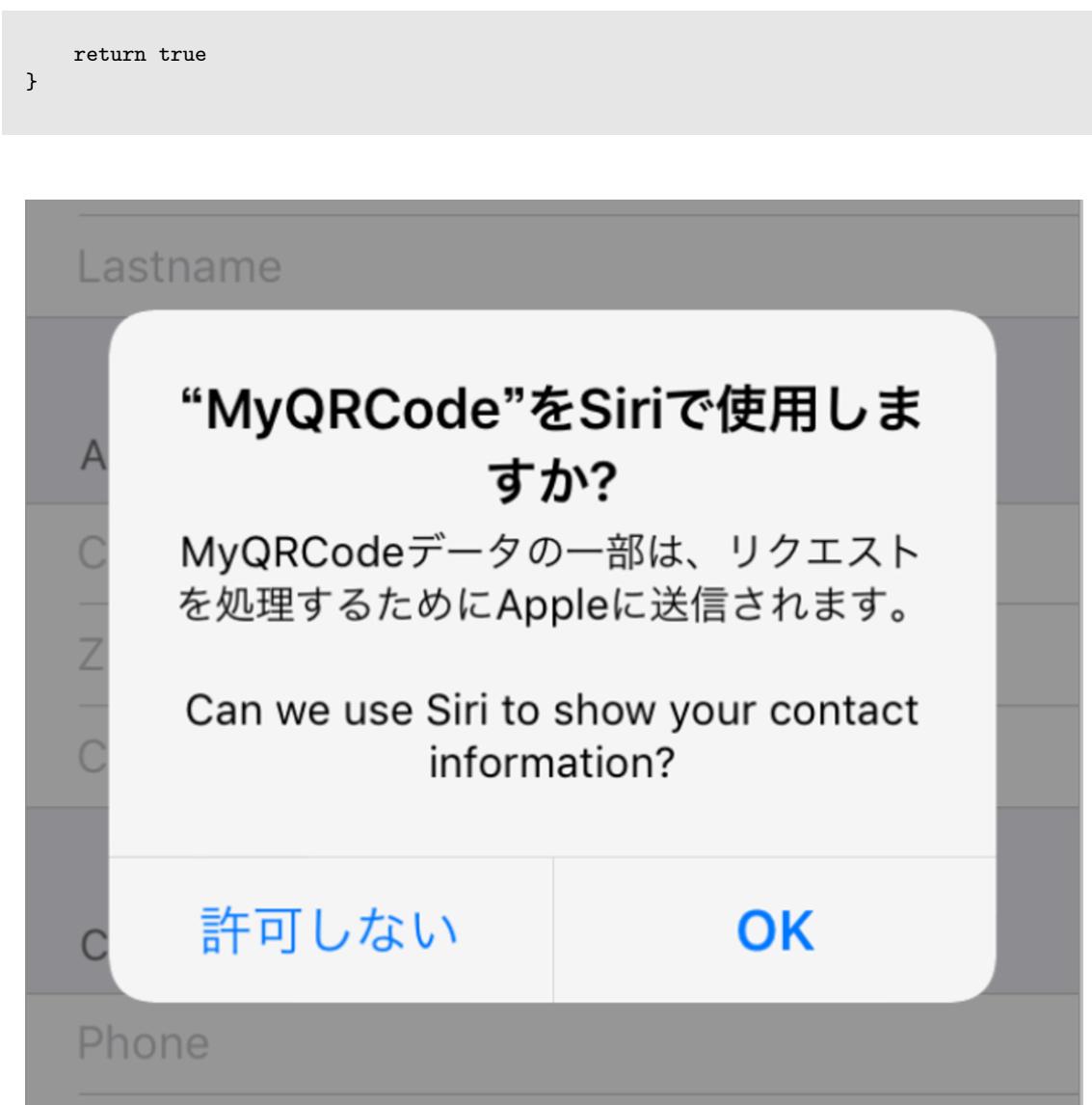


図 11.3: Siri の利用許可を取得する

上の例では AppDelegate におけるアプリの起動直後に呼び出していますが、必要に応じて、ヘルプ画面をはさむなど、ユーザーが許可を出しやすい工夫をすると良いでしょう。INPreferences.siriAuthorizationStatus() で得られる利用許可の値が.authorized になったときだけ、Siri はアプリに処理を依頼します。

SiriKit に対応するための、収容アプリ側に必要な手順は以上です。

#### 11.4.2 Intents Extension

Intents Extension 側の、SiriKit を使うために必要な作業は、Intents Extension をプロジェクトのターゲットとして追加し、その Extension で対応するタスクを Info.plist に設定することです。

この 2 作業以外については、利用する SiriKit のドメイン・Intent によって使用する型や情報が若干異なるものの、基本的な部分は同じです。たとえば Intent の種類によって適合するプロトコル (~IntentHandling)、渡される Intent オブジェクト、および completion ハンドラに渡すレスポンスオブジェクト (~IntentResponse) が異なるだけです。

### Intents Extension をプロジェクトに追加する

Intents Extension は、収容アプリのターゲットとして追加します。このとき「Include UI Extension」のチェックを有効にした場合、「Intents UI Extension」のターゲットも同時に追加されます。これは Siri の画面をカスタマイズする場合に使用します。Siri と連携するだけなら必須ではなく、本書でも説明しませんが、必要に応じて分かりやすい UI を構築すると使い勝手が良くなります。

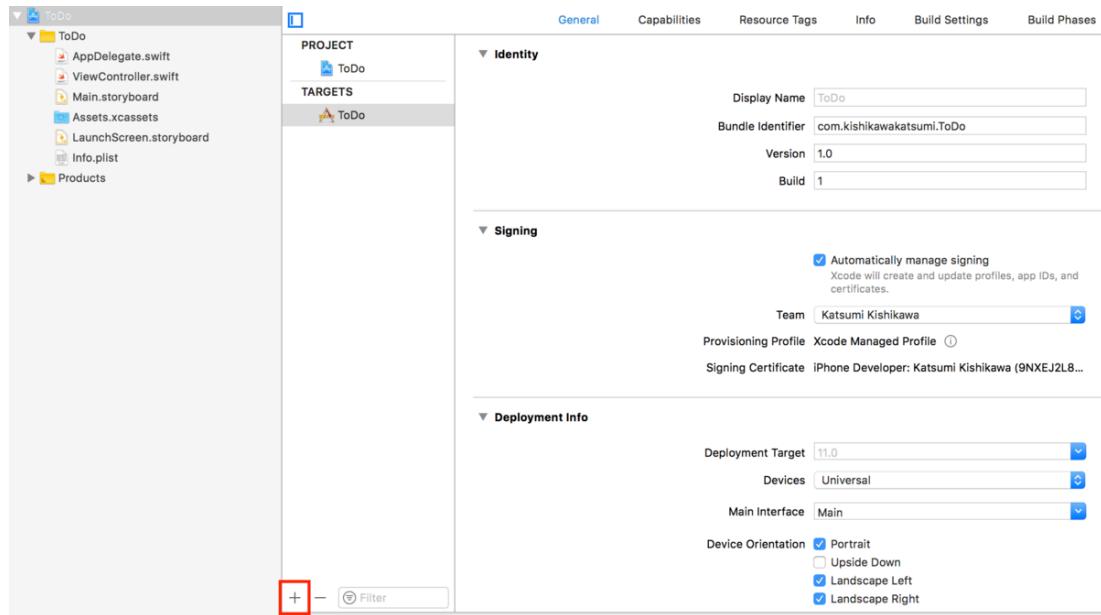


図 11.4: Extension をターゲットに追加

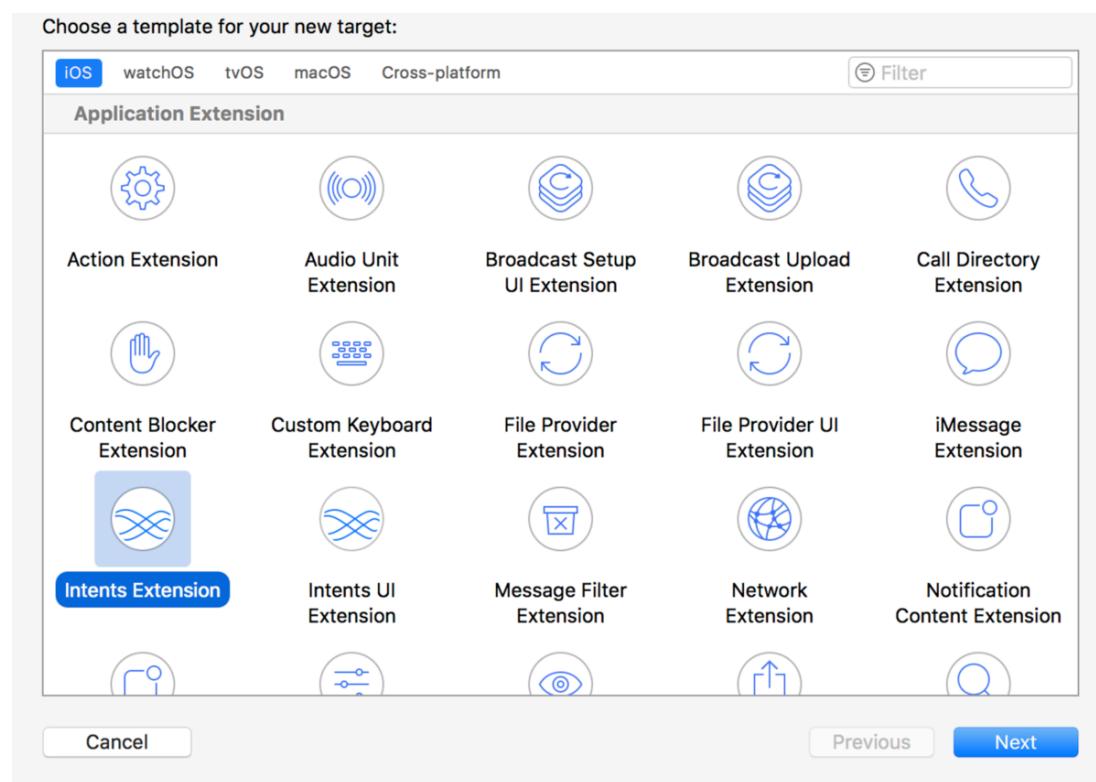


図 11.5: Extension をターゲットに追加

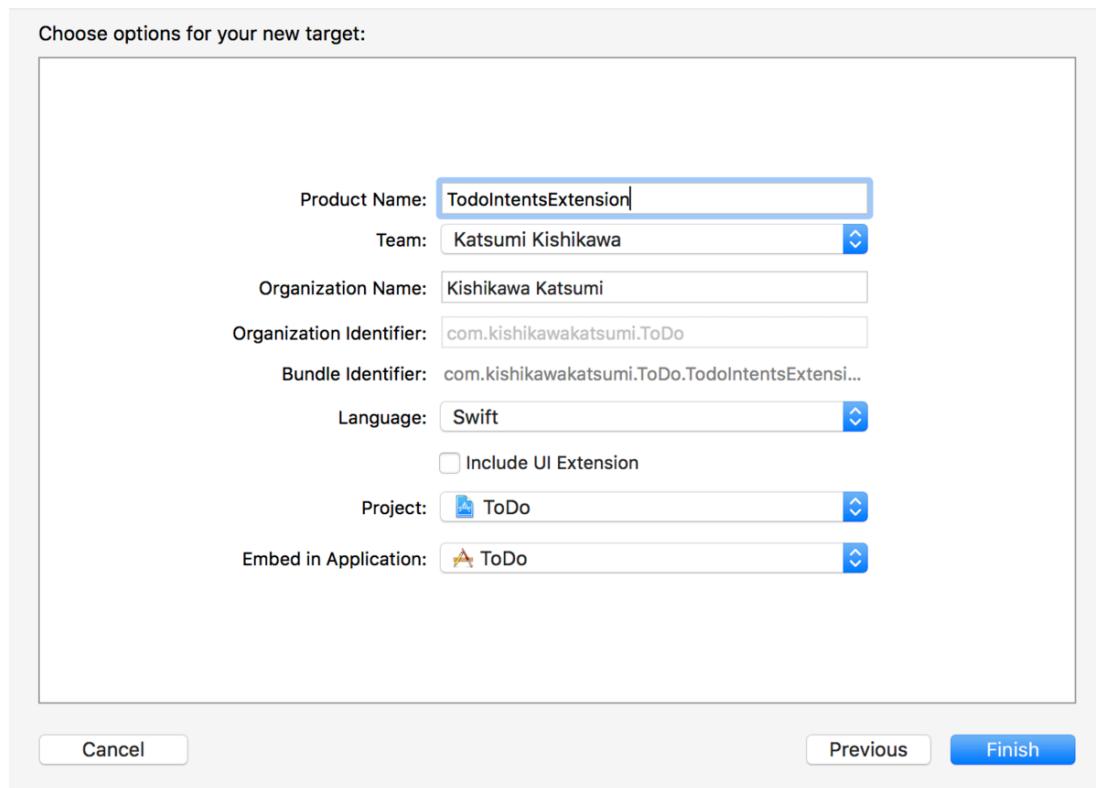


図 11.6: Extension をターゲットに追加

Extension の追加時、スキーマを有効にするかどうかを選択するダイアログが表示されるので「Activate」をクリックします。「Intents Extension」をデバッグする場合にはこのスキーマを指定して起動します。

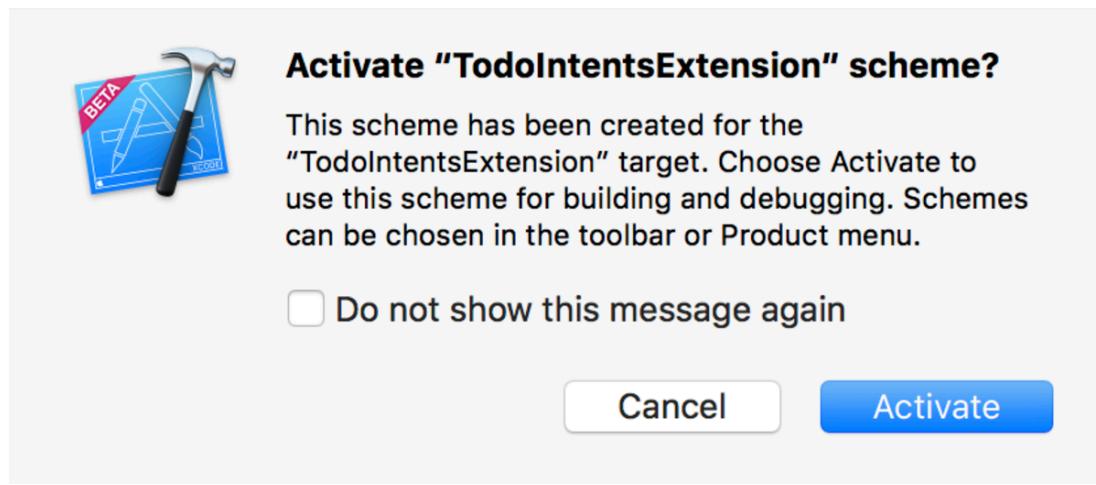


図 11.7: Extension のスキーマを有効にする

### 対応するタスクを Info.plist に設定

プロジェクトのターゲットとして Intents Extension を追加した後、Xcode が用意したデフォルトの Info.plist ファイルを編集して、どの Intent を扱うかを指定します。

Siri からタスクの処理を受け取るために、この Intents Extension が処理できるタスクを Info.plist に登録する必要があります。SiriKit はこの Info.plist ファイルの記述にもとづき、どの Intent を Extension に渡すのかを判断します。

Info.plist のキーの一覧は、表 11.6 のとおりです。

表 11.2: Info.plist のキーの一覧

NSExtension-NSExtensionAttributes	説明
NSExtensionAttributes	Dictionary
IntentsRestrictedWhileLocked	処理にデバイスのロック解除が必要かどうか。Intent のクラス名で指定
IntentsSupported	サポートする Intent のクラス名を配列で指定。定義順は Siri への問い合わせの解釈の優先度
NSExtensionPointIdentifier	エクステンションの種類 (Share や Today など) を示す識別子を指定。Siri エクステンション
NSExtensionPrincipalClass	エントリポイントとなるクラスの名前

**IntentsSupported** キーの配列には、使用する Intent (INIntent) のクラス名を設定します。Siri に話しかけた内容からドメインと Intent が決まると、システムは Info.plist の **IntentsSupported** キーに記された情報を調べ、タスクを処理できるアプリを判断します。もし一致していれば、**NSExtensionPrincipalClass** キーに記述されているクラスのインスタンスを生成し、Extension の `handler(for:)` メソッドを呼び出します。

各ドメインには 1 つ以上の Intent があらかじめ用意されています。**IntentsSupported** キーに Intent を複数記述することで、複数の Intent を 1 つのエクステンションで扱うこと、複数のドメインの Intent を扱うこともできます。また、ドメインに用意されている Intent のうちの一部だけに対応しても構いません。たとえば、To-Do 管理とメモ帳 (List and Notes) のドメインには 6 つの Intent が用意されていますが、後述するサンプル作成のときはタスクリスト作成の Intent (INCreateTaskListIntent) だけを扱います。

**IntentsSupported** キーに複数の Intent を記述した場合は、オブジェクトがどの Intent のインスタンスなのかを判断して、Intent の種類に応じて適切なオブジェクトを返さなければなりません。

**IntentsSupported** キーに設定される配列の順序は、優先度に影響します。「電話」と「メッセージ」といったよく似た Intent を扱うアプリの場合、Siri への問い合わせがあいまいであるとき「メッセージ」を優先したい場合は、「メッセージ」の Intent を先頭に記述します。

**IntentsRestrictedWhileLocked** キーは Intent の処理にデバイスのロック解除が必要となる Intent の場合に設定します。支払い (Payments) の Intent など金銭を扱うようなタスクの場合、デバイスのロック解除が必須です。また、デフォルトではロック解除が必要ない Intent について、何らかの理由でロック解除を求めるようにしたい場合にもこのキーを設定します。キーの値には Intent のクラス名を設定します。

なお、Xcode が用意する Info.plist ファイルにはデフォルトでメッセージ (Messaging) に関する Intent が 3 つ (INSendMessageIntent、INSearchForMessagesIntent、

`INSetMessageAttributeIntent`) 設定されています。メッセージ以外のタスクに対応したい場合、最初にこれらの Intent を削除する必要があります。

Key	Type	Value
▼ Information Property List	Dictionary	(10 items)
Localization native development r...	String	<code>\$(DEVELOPMENT_LANGUAGE)</code>
Bundle display name	String	<code>TodoIntentsExtension</code>
Executable file	String	<code>\$(EXECUTABLE_NAME)</code>
Bundle identifier	String	<code>\$(PRODUCT_BUNDLE_IDENTIFIER)</code>
InfoDictionary version	String	6.0
Bundle name	String	<code>\$(PRODUCT_NAME)</code>
Bundle OS Type code	String	XPC!
Bundle versions string, short	String	1.0
Bundle version	String	1
▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(2 items)
► IntentsRestrictedWhileLocked	Array	(0 items)
▼ IntentsSupported	Array	(3 items)
Item 0	String	<code>INSendMessageIntent</code>
Item 1	String	<code>INSearchForMessagesIntent</code>
Item 2	String	<code>INSetMessageAttributeIntent</code>
NSExtensionPointIdentifier	String	<code>com.apple.intents-service</code>
NSExtensionPrincipalClass	String	<code>\$(PRODUCT_MODULE_NAME).IntentHandler</code>

図 11.8: Extension に追加されるデフォルトの Info.plist

あとは各 Intent に沿った処理を記述することで、アプリから Siri を利用できるようになります。

## 11.5 サンプルプロジェクト：To-Do 管理とメモ帳（List and Notes）

iOS 11 で追加された To-Do 管理とメモ帳（List and Notes）のドメインを使って、アプリを作成してみましょう。

本節では、ドメインに用意されている 6 つの Intent のうち、次の Intent を扱います。

- タスクリストの作成（`INCreateTaskListIntent`）

サンプルプロジェクトでは、さらに次の 2 つの Intent にも対応したコードになっています。複数の Intent をどう扱うかの参考にしてください。

- タスクリストにタスクを追加（`INAddTasksIntent`）
- タスクの状態（完了、未完了）を変更（`INSetTaskAttributeIntent`）

### 11.5.1 IntentHandler（ハンドラの生成）

プロジェクトに Intents Extension のターゲットを追加します。

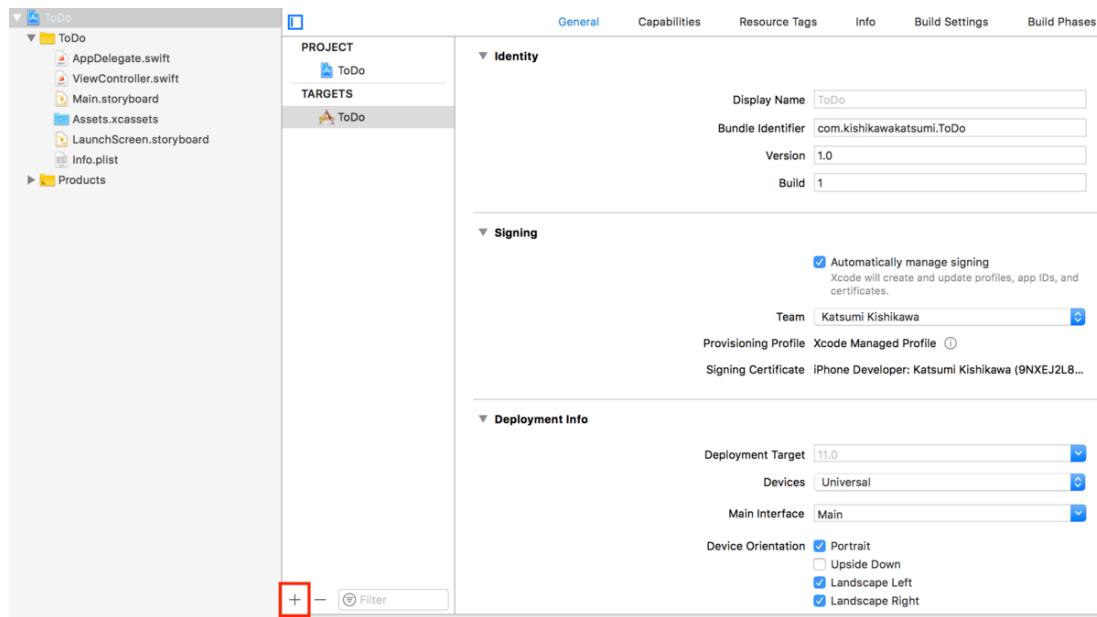


図 11.9: Extension をターゲットに追加

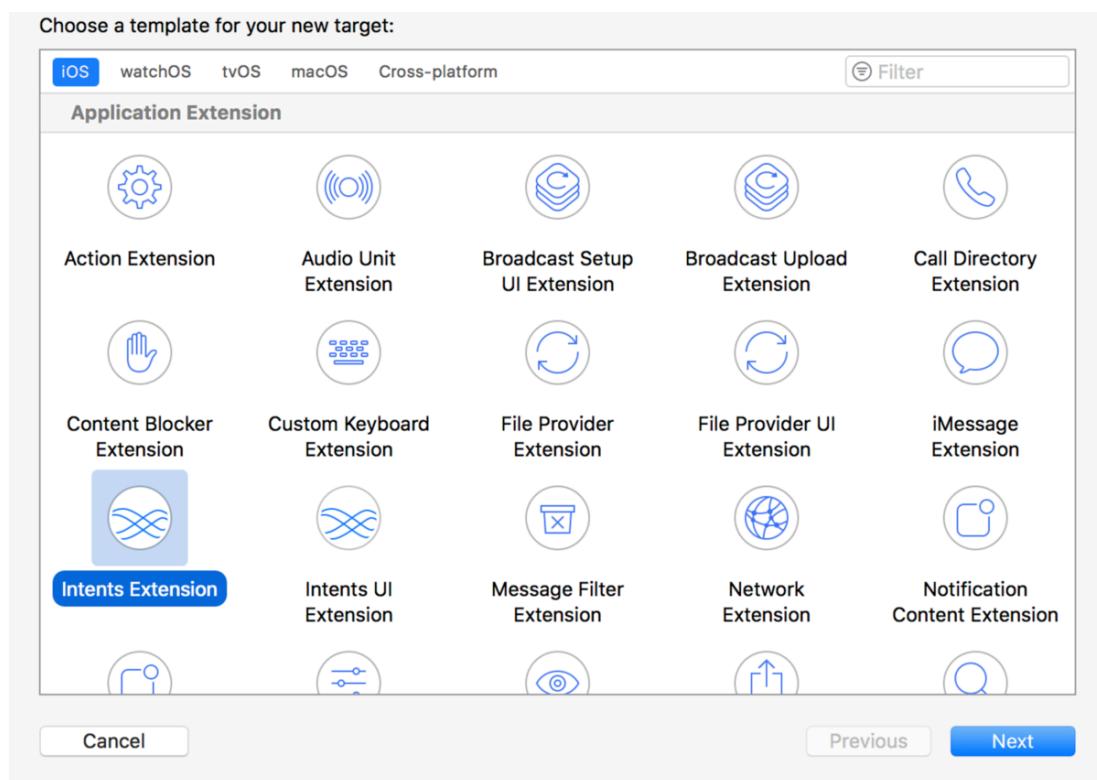


図 11.10: Extension をターゲットに追加

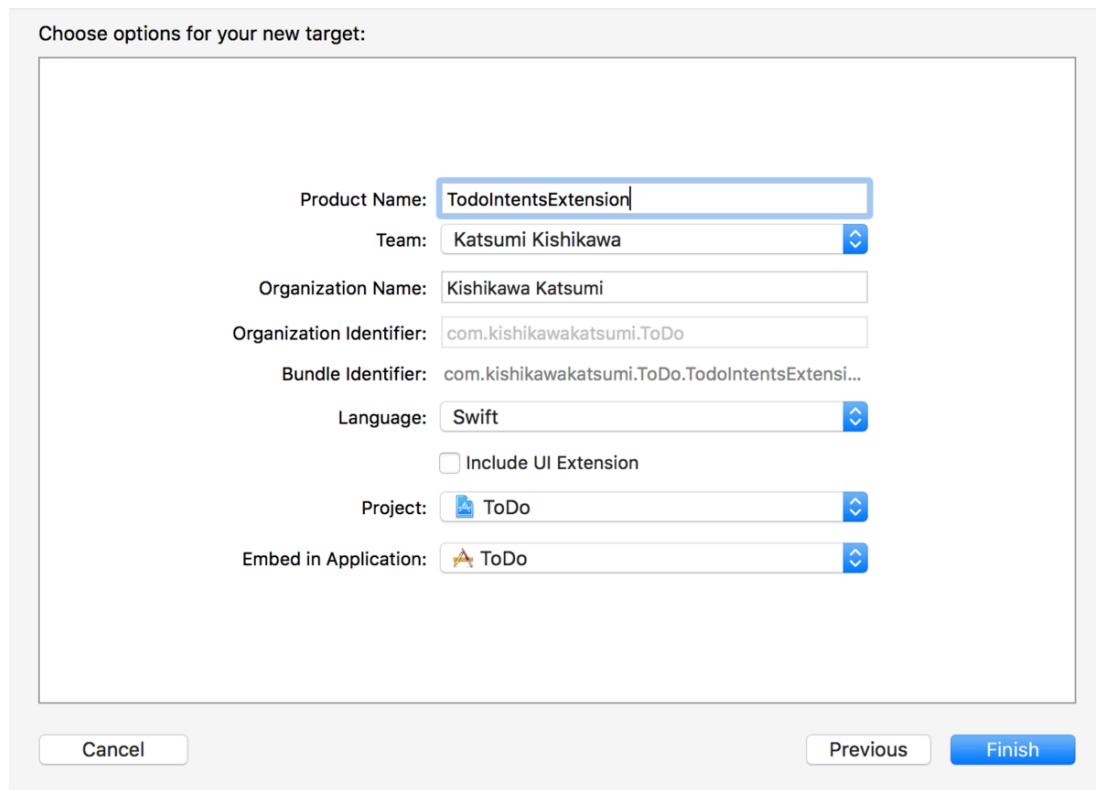


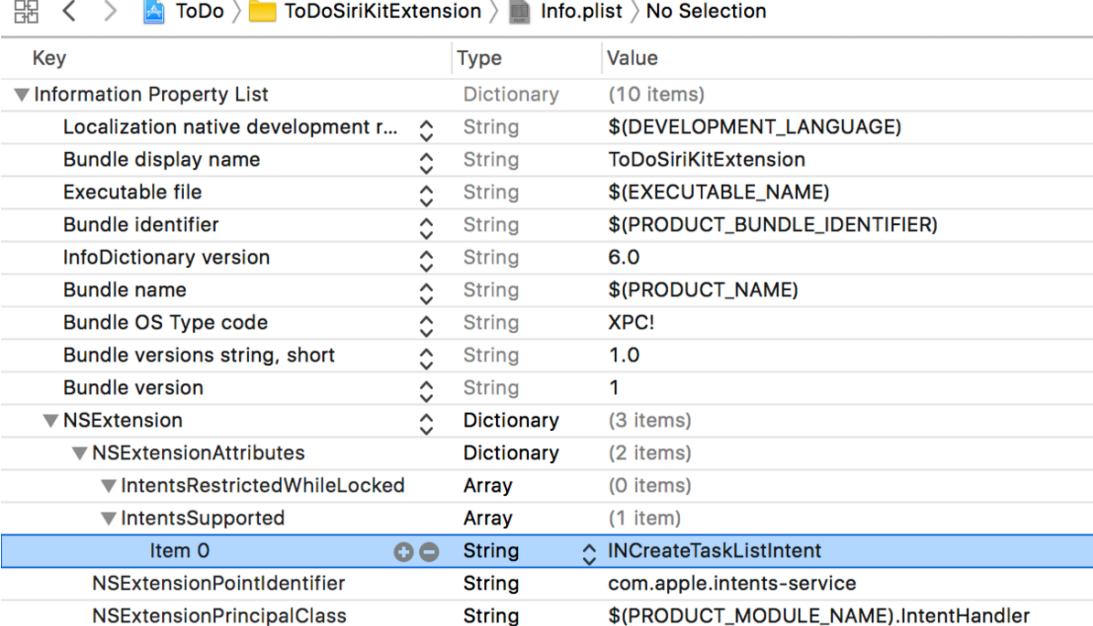
図 11.11: Extension をターゲットに追加

Info.plist にデフォルトで記述されている、メッセージ (Messaging) ドメインに対応するための設定は必要ないので削除しておきます。

Key	Type	Value
▼ Information Property List	Dictionary	(10 items)
Localization native development r...	String	\$(DEVELOPMENT_LANGUAGE)
Bundle display name	String	TodoIntentsExtension
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	XPC!
Bundle versions string, short	String	1.0
Bundle version	String	1
▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(2 items)
▼ IntentsRestrictedWhileLocked	Array	(0 items)
▼ IntentsSupported	Array	(0 items)
NSExtensionPointIdentifier	String	com.apple.intents-service
NSExtensionPrincipalClass	String	\$(PRODUCT_MODULE_NAME).IntentHandler

図 11.12: Info.plist

Siri からのタスクリスト作成に対応するには、Intents Extension の Info.plist に設定する IntentsSupported キーの配列に、Intent のクラス名である INCreateTaskListIntent を追加します。



Key	Type	Value
▼ Information Property List	Dictionary	(10 items)
Localization native development r...	String	\$(DEVELOPMENT_LANGUAGE)
Bundle display name	String	ToDoSiriKitExtension
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	XPC!
Bundle versions string, short	String	1.0
Bundle version	String	1
▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(2 items)
▼ IntentsRestrictedWhileLocked	Array	(0 items)
▼ IntentsSupported	Array	(1 item)
Item 0	String	INCreateTaskListIntent
NSExtensionPointIdentifier	String	com.apple.intents-service
NSExtensionPrincipalClass	String	\$(PRODUCT_MODULE_NAME).IntentHandler

図 11.13: Info.plist

自動生成された IntentHandler.swift にも、メッセージドメインに関する各種処理がすでに記述されています。Info.plist と同様にそれらを取り除きます。そして、handler(for:) メソッドで、INCreateTaskListIntent を処理するクラスを作成し、返します。

```
override func handler(for intent: INIntent) -> Any? {
 switch intent {
 case is INCreateTaskListIntent:
 return CreateTaskListHandler()
 default:
 return nil
 }
}
```

上のコードでは、intent オブジェクトの種類を switch 文によって割り振り、適切なインスタンスを返しています。今回は、INCreateTaskListIntent の場合に、この後で作成する CreateTaskListHandler クラスのインスタンスを返すようにします。CreateTaskListHandler クラスは INCreateTaskListIntentHandling プロトコルに準拠したクラスで、以降のフェーズで実装します。

CreateTaskListHandler.swift を追加して、下のように記述します。

```
import Foundation
import Intents
import ToDoKit

class CreateTaskListHandler: NSObject, INCreateTaskListIntentHandling {
```

INCreateTaskListIntentHandling プロトコルは、INCreateTaskListIntent に対応するハンドラオブジェクトのプロトコルです。

ところで、Xcode が自動的に作成するコードでは `self` を返し、同じクラスですべてのプロトコルを実装するようなコードになっています。簡単なものであればこれで十分でしょうが、複数の Intent に対応する場合だとコードがわかりにくくなってしまいます。このようにプロトコルごとに専用のクラスを用意し、`handler(for:)` メソッドでは Intent に対応する専用のクラスを返すだけにした方が読みやすくなります。

### 11.5.2 タスクリストのタイトルを検証する（リゾルブ）

今回の例では、タスクリストの生成に最低限必要な情報はタスクリストのタイトルだと仮定します。サンプルプロジェクトでは Intent から受け取ったタイトルをそのまま使用していますが、実際のアプリでは、文字数の制限があったり、すでに存在する名前と同じ名前のタスクリストは作成できなかったり、あるいはサーバーと通信する必要がある場合にはすぐに結果を返すことができない場合などがあります。そのような場合、リゾルブフェーズのメソッドを使って不足している情報をユーザーに求めたり、受け取った情報をチェックして訂正を求めたりするなど、エクステンション内でユーザーと対話できます。

リゾルブの一例として、タスクリストのタイトルを確認する `resolveTitle(for:with:)` を実装してみましょう。次のコードのとおり、同メソッドを `CreateTaskListHandler` クラス内に記述します。

```
func resolveTitle(for intent: INCreateTaskListIntent, with completion: @escaping (INSpeakableStringResolutionResult) -> Void) {
 if let title = intent.title {
 completion(.success(with: title))
 } else {
 completion(.needsValue())
 }
}
```

タイトルが存在すれば `.success(with: title)` を返し次のステップへ、存在しなければ `.needsValue()` を返し、ユーザーに再度タイトルを話しかけてもらいます。

`completion` ハンドラのパラメータである `IN~ResolutionResult` は `INIIntentResolutionResult` のサブクラスです。返せる値はサブクラスとスーパークラスの両方に定義されていて、`INIIntentResolutionResult` には `needsValue()`、`notRequired()`、`unsupported()` の共通で

使用する3つの状態が定義されています。サブクラスにはIntentによって変わる状態が定義されているので、サブクラスのメソッドだけを見ていると適切な値が見つからないことがあるので注意してください。

すでに同じ名前のタイトルが存在する場合には、`unsupported()`を返して失敗させたり、`needsValue()`を返して再度入力してもらったり、`disambiguation(with:)`を返して代わりの選択肢を提示したりするなどが考えられます。アプリの実装やユーザーの使い勝手を考慮して適切なものを選択しましょう。

応用として、タイトルが与えられていなければデフォルトのタイトルを自動的に付けても良いですし、「(デフォルトのタイトル)でタスクリストを作成します」のようにユーザーに確認を求める事もできます。タイトルがあいまいだったり、既存のタイトルと衝突する場合に、ユーザーに確認を求めたり、代わりの選択肢を提示したりもできます。

「動作のしくみ」セクションで説明したように、`resolve~(for:with:)`メソッドの実装は任意です。

### 11.5.3 Intentを実行する前の最終確認（コンファーム）

リゾルブの終了後、実行前の最終確認を行うのがコンファームです。

コンファームの段階で呼ばれる`confirm(intent:completion:)`メソッドでは、Intentの処理を実行できるかどうか、具体的にはハンドルフェーズの`handle(intent:completion:)`メソッドに進めるかどうかの最終確認を行います。

サンプルプロジェクトでのコードは次のとおりです。

```
func confirm(intent: INCreateTaskListIntent,
 completion: @escaping (INCreateTaskListIntentResponse) -> Swift.Void) {
 // ここで最終確認を行い、その結果をレスポンスとして返す
 let response = INCreateTaskListIntentResponse(code: .ready,
 userActivity: nil)
 completion(response)
}
```

このメソッドで検証することは特にないので、いかなる場合でも`.ready`を返すだけにしています。<sup>\*7</sup>

### 11.5.4 INCreateTaskListIntentを処理する（ハンドル）

リゾルブ・コンファームの後、いよいよハンドルフェーズで処理を行います。

`INCreateTaskListIntentHandling`プロトコルに準拠しているため、実装しなければならないメソッドは`handle(intent:completion:)`です。`handle(intent:completion:)`メソッドの1番目の引数には、`INCreateTaskListIntent`のインスタンスが渡されます。

`CreateTaskListHandler.swift`の`handle(intent:completion:)`メソッドにタスクの処理

<sup>\*7</sup> `confirm(intent:completion:)`メソッドの実装は任意ですので、`.ready`を返すだけなら実装は不要です。

を記述します。INCreateTaskListIntent で行うべきタスクは、タスクリストの新規作成、（あれば）同時にいくつかのタスクを追加する、（必要に応じて）タスクリストをグループに分類する、の3つです。最低限期待されていることは、タスクリストの新規作成です。

CreateTaskListHandler クラスに、次のように handle(intent:completion:) メソッドを追加します。

```
func handle(intent: INCreateTaskListIntent,
 completion: @escaping (INCreateTaskListIntentResponse) -> Void) {
 guard let title = intent.title else {
 let response = INCreateTaskListIntentResponse(code: .failureRequiringAppLaunch,
 userActivity: nil)
 completion(response)
 return
 }

 if let taskTitles = intent.taskTitles, !taskTitles.isEmpty {
 Repository.shared.addTaskList(title: title.spokenPhrase,
 tasks: taskTitles.map { Task(title: $0.spokenPhrase) })
 } else {
 Repository.shared.addTaskList(title: title.spokenPhrase)
 }

 let response = INCreateTaskListIntentResponse(code: .success,
 userActivity: nil)
 completion(response)
}
```

このメソッドが呼ばれるときには、処理すべき Intent の種類は決まっています。Intent オブジェクトには Siri が解釈した情報が格納されており、これを使ってタスクを実行します。最後にタスクの実行結果にもとづいて completion ハンドラを呼び出します。Siri から与えられる Intent オブジェクトの内容は、ユーザーが話した内容によるので、必要な情報が不足していたり、間違っている場合があります。その場合は completion ハンドラに与える引数によって、タスクが完了しなかったことや、追加情報を入力するために収容アプリを起動する必要があることなどをシステムに伝えます。

上のコードは、まず intent オブジェクトからタスクリストのタイトルがあるかどうかをチェックします。タイトルが提供されていなければ Extension だけで処理をするには情報が不足しているので、失敗のステータスをレスポンスとして返し、続きの処理をするためにユーザーに収容アプリの起動を促します。

タスクリストのタイトルがあれば、さらに同時に追加するタスクが存在するかどうかをチェックします。タスクのタイトルがあれば、タスクリストの作成と同時にタスクも作成します。タスクのタイトルがなければタスクリストだけを作成します。

タスクリストの作成には、タスクのタイトルがあれば十分ですので、成功のステータス.success をレスポンスにセットします。

エクステンションで実行する処理は基本的には収容アプリと同じ処理になります。そのため、できるだけ共通のコードを利用したいところです。コードを共通化するために、独立したフレームワー

クとして切り出し、収容アプリとエクステンションでそのフレームワークを利用するようにします。

上記のコード例では、共通のコードを `ToDoKit` というフレームワークにまとめています。`addTaskList(title:)`などのメソッドは `ToDoKit` に実装されているメソッドで、収容アプリでもタスクリストやタスクの作成には同じメソッドを使用しています。

サンプルプロジェクトでは、さらに `INAddTasksIntent` と `INSetTaskAttributeIntent` をサポートし、各タスクリストにタスクを追加することと、タスクを完了済みにマークすることができるようになっています。プロジェクトの完全なコードはサンプルプロジェクトをご覧ください。

## 11.6 サンプルプロジェクト：QR コード表示

タスクリスト生成のアプリに続いて、本節では QR コード（Visual Codes）のドメインを使用したアプリを作ります。自分の連絡先（名前、住所、電話番号、メールアドレスなど）を QR コードにエンコードして表示するプログラムです。

??

### 11.6.1 IntentHandler（ハンドラの生成）

プロジェクトに Intents Extension のターゲットを追加します。

Siri の画面に QR コードを表示できるようにするには、Intent Extension の `Info.plist` に設定する `IntentsSupported` キーの配列に、Intent のクラス名である `INGetVisualCodeIntent` を追加します。

Intent Extension に対し、Intent に対応するプロトコルを実装します。`INGetVisualCodeIntent` に対応するプロトコルは `INGetVisualCodeIntentHandling` です。

`IntentHandler.swift` のデフォルトで作成されたコードを次のコードに置き換えて、`INGetVisualCodeIntent` を処理するクラスとして `self` を返してください。

```
import Intents

class IntentHandler: INExtension {
 override func handler(for intent: INIntent) -> Any {
 return self
 }
}
```

先程のタスクリスト作成の処理では、`IntentHandler` とは別のオブジェクトがハンドラオブジェクトになっていました。QR コードのドメインは、用意されている Intent が 1 種類しかないので、Intent に対する処理もこのクラスに書くことにします。Intent を処理するオブジェクトは自分自身なので、`handler(for:)` メソッドで `self` を返しています。

まず `INGetVisualCodeIntent` を処理するコードを追加します。`IntentHandler.swift` のコードを次のように変更してください。

```
class IntentHandler: INExtension, INGetVisualCodeIntentHandling {
 override func handler(for intent: INIntent) -> Any? {
 return self
 }

 func handle(intent: INGetVisualCodeIntent,
 completion: @escaping (INGetVisualCodeIntentResponse) -> Void) {
 let response = INGetVisualCodeIntentResponse(code: .failure, userActivity: nil)
 completion(response)
 }
}
```

IntentHandler クラスが INGetVisualCodeIntentHandling プロトコルに適合するようになりました。～IntentHandling プロトコルで最低限実装の必要なメソッドは handle(intent:completion:) です。これはどの Intent を実装する場合でも共通です。

名前が似ていて紛らわしいですが、handler(for:) メソッドは INExtension が適合している INIntentHandlerProviding プロトコルの必須メソッドです。handler(for:) からエクステンションの処理が始まり、適切なオブジェクトに処理を振り分ける役割を担っています。引数の intent オブジェクトの種類に応じて、Intent を処理する適切なオブジェクトを返します。

### 11.6.2 QR コードの種類を確認する（リゾルブ）

INGetVisualCodeIntentHandling には、リゾルブで使用するメソッドとして resolveVisualCodeType(for:with:) メソッドが用意されています。このメソッドですることは、何の QR コードを表示すべきかを決定することです。

現在のバージョンでは、QR コードの種類として次の 4 種類が定義されています。この値は引数に渡された Intent の visualCodeType プロパティに格納されています。

- .unknown (不明)
- .contact (連絡先)
- .requestPayment (請求)
- .sendPayment (送金)

今回のサンプルプロジェクトは連絡先データを表示するアプリなので、.contact が指定されていた場合は目的に合致しています。.requestPayment、.sendPayment が指定されていた場合は、明らかにユーザーの期待している動作は達成できません。その場合、この時点で処理を失敗させることが望ましいでしょう。

.unknown については、Intent のリファレンス<sup>\*8</sup>に載っている「自分の QR コードを表示」というフレーズ例の場合、visualCodeType は.unknown になります。.contact となるフレーズは「自分の連絡先コードを表示」などです。このことから、今回の場合は.unknown の場合は連絡先を表示すると解釈した方が、使い勝手が良さそうという判断で話を進めます。

<sup>\*8</sup> <https://developer.apple.com/documentation/sirikit/ingetvisualcodeintent>

`IntentHandler.swift` に、次のように `resolveVisualCodeType(for:with:)` メソッドを追加します。

```
func resolveVisualCodeType(for intent: INGetVisualCodeIntent,
 with completion: @escaping (INVisualCodeTypeResolutionResult) -> Void) {
 switch intent.visualCodeType {
 case .unknown, .contact:
 completion(.success(with: .contact))
 default:
 completion(.unsupported())
 }
}
```

引数として渡される `intent` オブジェクトの `visualCodeType` プロパティをチェックします。`.unknown` または `.contact` が指定されている場合は、処理を継続します。それ以外の `visualCodeType` なら `.unsupported()` を返して未サポートということを伝え、処理を中断します。

### 11.6.3 Intent を実行する前の最終確認（コンファーム）

コンファームのタイミングで、`handle(intent:completion:)` メソッドに進めるかどうかの最終確認を行います。

サンプルプロジェクトでは QR コードの生成はローカルで行なっていて、必要な情報もすべてローカルのデバイス上に存在するので、サーバーと通信する必要はありません。しかし、QR コードの生成をサーバーにリクエストしたり、さらに追加の情報をサーバーから取得するような場合は、通信が完了するまでの間、`.inProgress` を返しておき、準備が整うまで待つことができます。

次のコードはその一例です。`confirm(intent:completion:)` メソッドを追加します。

```
func confirm(intent: INGetVisualCodeIntent,
 completion: @escaping (INGetVisualCodeIntentResponse) -> Void) {
 let response = INGetVisualCodeIntentResponse(code: .ready,
 userActivity: nil)
 completion(response)
}
```

今回はこのメソッドで検証することは特ないので、いかなる場合でも `.ready` を返します。

例えば、QR コードの生成や追加の情報にネットワーク通信が必要な場合は、いったん `.inProgress` を返しておき通信を開始します。そして通信が終り準備ができた時点で `.ready` を返すようにできます。何らかの理由により、タスクが実行できないことが判明した場合には、この時点での失敗のレスポンスを返します。

#### 11.6.4 INGetVisualCodeIntent を処理する（ハンドル）

`handle(intent:completion:)` メソッドにタスクの処理を記述します。`INGetVisualCodeIntent` で行うべきタスクは、QR コードに代表される Visual Code の作成です。

次のコード例では共通のコードを `MyQRCodeKit` というフレームワークにまとめています。`Repository` や `generateVisualCode()` などのメソッドを `MyQRCodeKit` に実装しておくことで、収容アプリで QR コードを表示する場合でも同じメソッドを使えるようにしています。

`IntentHandler.swift` を次のように変更します。

```
(import MyQRCodeKit)

func handle(intent: INGetVisualCodeIntent,
 completion: @escaping (INGetVisualCodeIntentResponse) -> Void) {
 guard let me = Repository.shared.load() else {
 let response =
 INGetVisualCodeIntentResponse(code: .failureAppConfigurationRequired,
 userActivity: nil)
 completion(response)
 return
 }
 guard let image = me.generateVisualCode(),
 let imageData = UIImagePNGRepresentation(image) else {
 let response = INGetVisualCodeIntentResponse(code: .failure,
 userActivity: nil)
 completion(response)
 return
 }

 let response = INGetVisualCodeIntentResponse(code: .success, userActivity: nil)
 response.visualCodeImage = INImage(imageData: imageData)
 completion(response)
}
```

上のコードでは、まず自分の連絡先データが登録されているかどうかをチェックしています。このサンプルプロジェクトでは自分の連絡先データは収容アプリを起動して登録する必要があります。他にもデバイスのアドレス帳から取得したり、サーバーの API から取得する場合もあるでしょう。

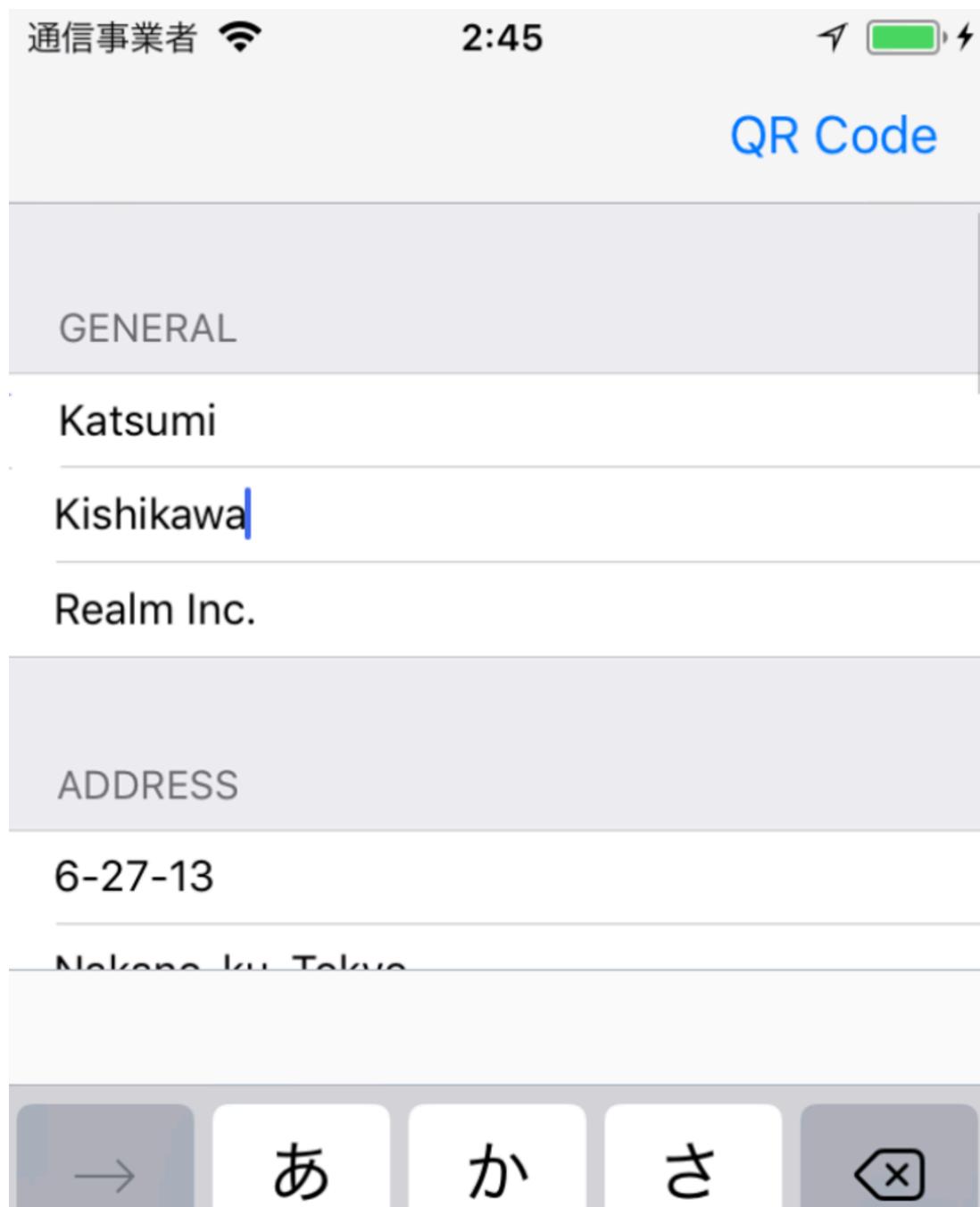


図 11.14: QR コード収容アプリ

連絡先データが登録されていなければ、処理ステータスを失敗として続きの処理はアプリを起動する必要があることを伝えます (.failureAppConfigurationRequired)。連絡先データが登録されていれば、連絡先データを QR コードにエンコードし、QR コードの画像を生成します。何らかの理由により、QR コードの生成に失敗した場合は、処理ステータスを失敗にして処理を中断します (.failure)。

QRコードが正しく生成された場合は、QRコードのデータから `INImage` オブジェクトを作成してレスポンスにセットします (`.success`)。`INImage` は Intent に画像を渡すときに使われるラッパークラスです。QRコードのドメイン以外でも画像が必要になる場合に利用されます。Intents Extension の UI をカスタマイズする際にも利用されます。

### 11.6.5 連絡先から QR コードを生成する

CoreImage フレームワークを使うことで QR コードを生成できます。次のコードを見てください。

```
public func generateVisualCode() -> UIImage? {
 let parameters: [String : Any] = [
 "inputMessage": vCardRepresentation(),
 "inputCorrectionLevel": "L"
]
 let filter = CIFilter(name: "CIQRCodeGenerator", withInputParameters: parameters)

 guard let outputImage = filter?.outputImage else {
 return nil
 }

 let scaledImage = outputImage.transformed(by: CGAffineTransform(scaleX: 6, y: 6))
 guard let cgImage = CIContext().createCGImage(scaledImage, from: scaledImage.extent) else {
 return nil
 }

 return UIImage(cgImage: cgImage)
}
```

QRコードの生成には `CIQRCodeGenerator` という CoreImage のフィルタ (`CIFilter`) を使用します。必要なパラメータは `inputMessage` と `inputCorrectionLevel` です。

- `inputMessage` : QRコードにエンコードするデータ。Data型に変換して渡す
- `inputCorrectionLevel` : 誤り訂正能力。L、M、Q、Hのいずれかを指定する

誤り訂正能力とは、コードの一部が汚れたり破れたりして欠損したとしても、コード自身でデータを復元できる能力のことです。Lがもっとも低く、Hがもっとも高い誤り訂正能力を持ちます。それぞれ L: 7%、M: 15%、Q: 25%、H: 30% の欠損に耐えられます。レベルを上げれば誤り訂正能力は向上しますが、データが増えるため、コードのサイズは大きくなります。

サンプルプロジェクトだと、コードを画面表示するため欠損するケースは稀でしょうから、誤り訂正能力にはもっとも低いレベルを指定しています。

QRコードにはデータ長の上限を超えない限りは自由に文字列をエンコードできます。ただせつかく iPhone に表示する以上は、iPhoneを使って簡単に読み取れ、アドレス帳に登録できた方が使い勝手がいいでしょう。そのために独自フォーマットではなく連絡先の交換に用いられる汎用のフォーマットを使用することにします。

サンプルプロジェクトでは iPhone のアドレス帳でも使われている vCard 形式を使用していま

す。vCard フォーマットにはいくつかバージョンがあり、例ではバージョン 3.0 を使用しています。vCard 3.0 のフォーマットは次のとおりです。

```
BEGIN:VCARD
VERSION:3.0
N:Lastname;Surname
FN:Displayname
ORG:EVENX
URL:http://www.evenx.com/
EMAIL:info@evenx.com
TEL;TYPE=voice,work,pref:+49 1234 56788
ADR;TYPE=intl,work,postal,parcel:;Wallstr. 1;Berlin;;12345;Germany
END:VCARD
```

このほかにも NTT DoCoMo の携帯電話で QR コードを利用した連絡先交換フォーマットとして利用されていた MECARD 形式も利用できます。

```
MECARD:N:Doe, John;TEL:13035551212;EMAIL:john.doe@example.com;;
```

どちらの形式でも iPhone のカメラアプリで読み取ることができ、アドレス帳に自動的に登録されます。

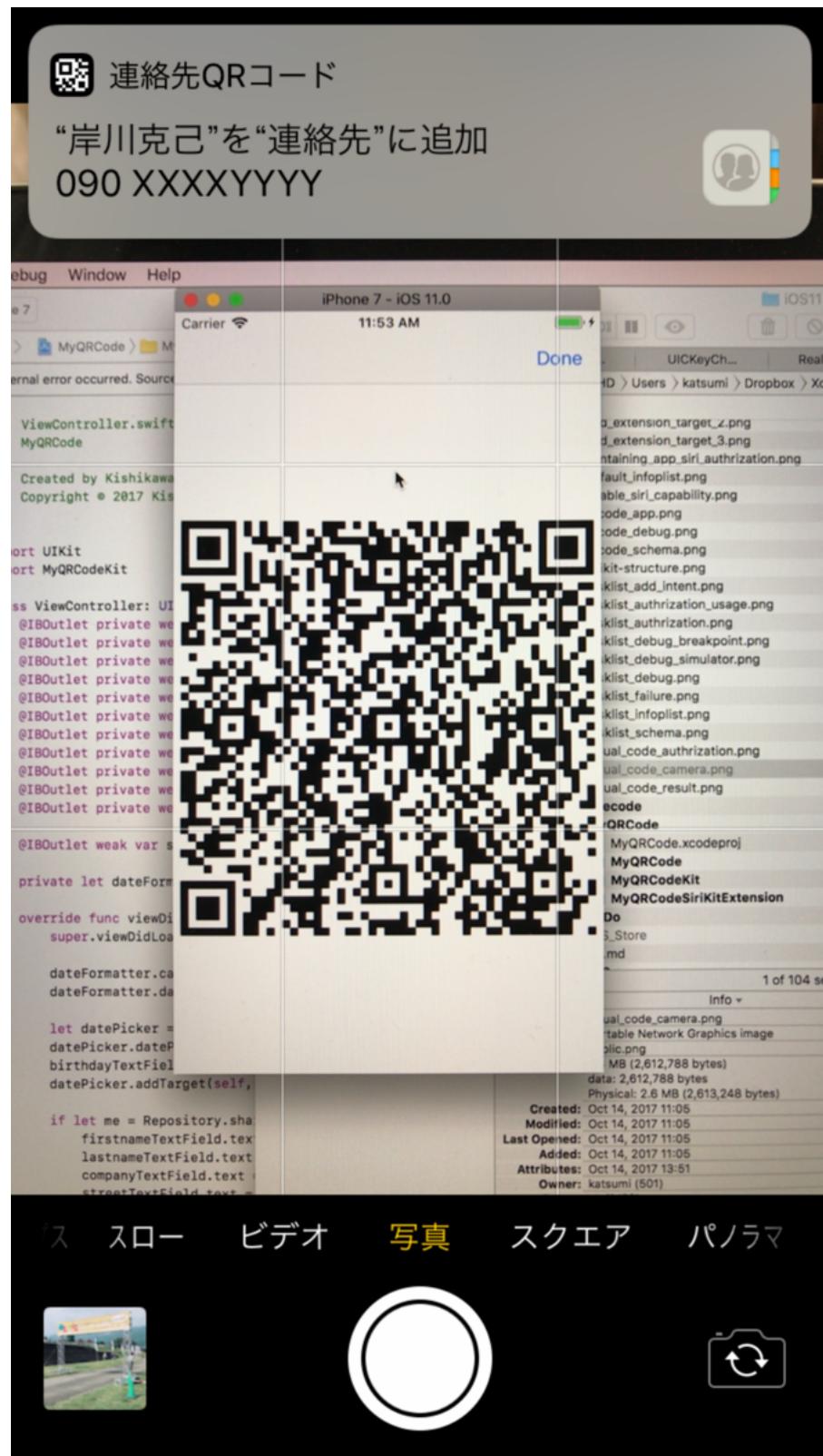


図 11.15: 標準カメラで QR コードを読み取る

CIQRCodeGenerator で生成した QR コードはそのままだと画面に表示するには小さいので、アフィン変換をかけて 6 倍に拡大しています。

```
let scaledImage = outputImage.transformed(by: CGAffineTransform(scaleX: 6, y: 6))
```

## 11.7 Extension のデバッグ

Siri を起点に Extension の処理が呼び出されるかどうか、タスクリスト生成のサンプルを使って確認してみましょう。

Intents Extension の動作を確認・デバッグするには、まず Xcode が起動するスキーマを Extension のスキーマに変更します。

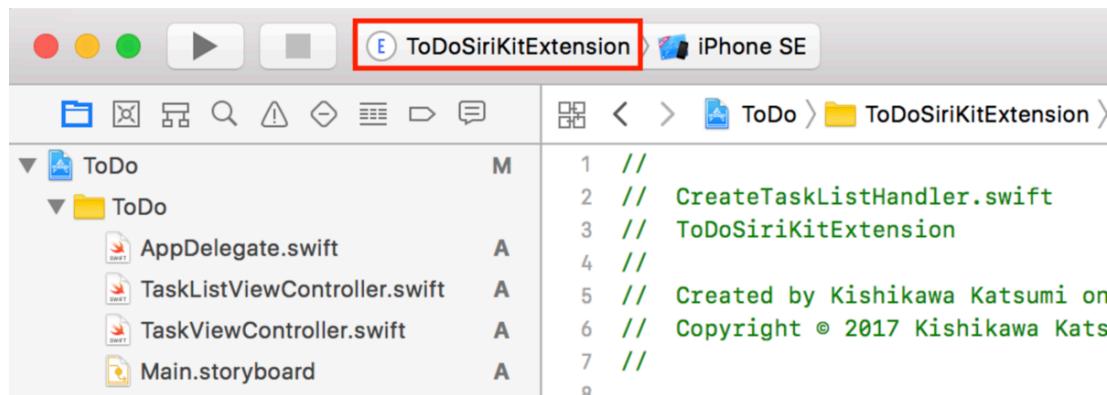


図 11.16: Extension のスキーマを選択する

確認はデバイスを使って実際に Siri に話しかけても良いですが、ここでは簡単にシミュレータと「Siri Intent Query」の機能を使いましょう。今回は日本語のフレーズを利用るので、シミュレータの Siri の設定で言語 (Language) を日本語 (Japanese) にあらかじめ変更しておきます。シミュレータ、またはデバイス自体の言語や地域の設定にかかわらず、Siri が解釈する言語はこの設定が使用されます。



図 11.17: Siri Intent Query

Extension のスキーマを選択した状態で、「Edit Scheme...」を選択します。「Run」セクションの「Info」タブを選択します。画面の中ほどに「Siri Intent Query」のテキストボックスがあります。ここに入力したテキストは、Xcode から実行した際に自動的に Siri に話しかけるフレーズとして使われるため、Extension のデバッグで実際に声を出す必要はありません。またこの機能を使って、どのようなフレーズが Siri に必要な情報として解釈されるのか試行錯誤できます。フレーズによっては期待した通りに呼び出されないこともあるので、必要に応じて Siri の解釈にヒントを与える「独自の語彙を追加する」ことも検討します。

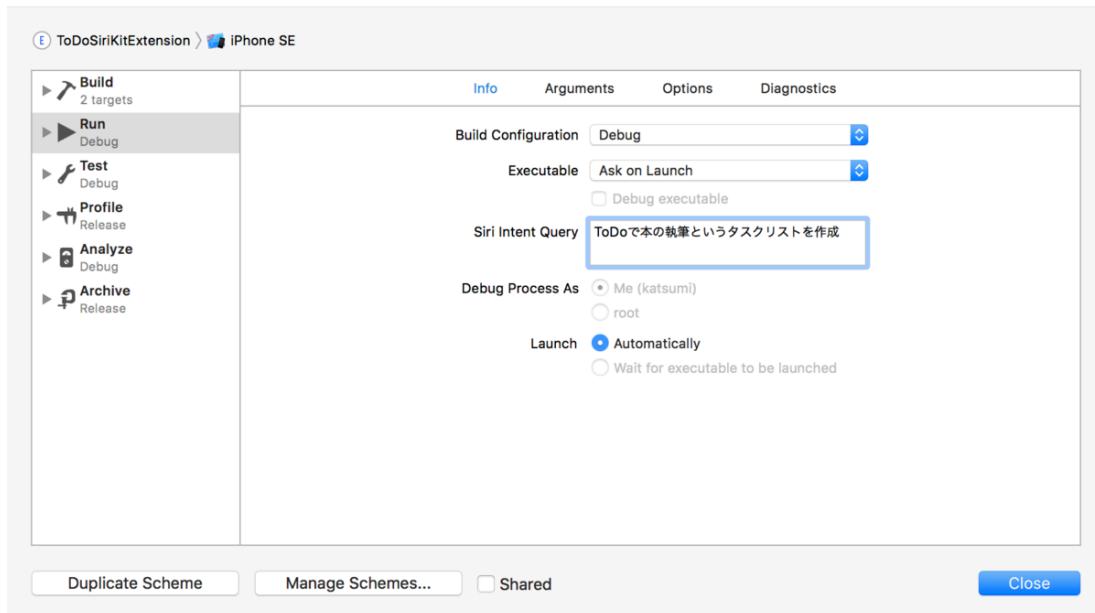


図 11.18: Siri Intent Query

Extension が Siri から呼び出されることを確認するため、`IntentHandler.swift` の `handler(for:)` メソッドと、`CreateTaskListHandler.swift` の `handle(intent:completion:)` メソッドにブレークポイントを置いてください。そして「Siri Intent Query」に「ToDo で本の執筆というタスクリストを作成」と入力し、実行してみてください。

Siri がフレーズを解釈し、Extension が Siri から起動されるはずです。そして、`IntentHandler.swift` の `handler(for:)` メソッドのブレークポイントで処理が止まります。



図 11.19: Extension のデバッグ（シミュレータの表示）

ブレークポイントで `handler(for:)` メソッドに渡される `intent` オブジェクトの内容を確認してみましょう。デバッガを使ってプリントすると、`INCreateTaskListIntent` クラスのオブジェクトで、`title` プロパティに「本の執筆」と設定されていることが確認できます。フレーズは Siri によって正しく解釈され、Extension が呼び出されていることがわかります。

```
11 class IntentHandler: INExtension {
12 override func handler(for intent: INIntent) -> Any? {
13 switch intent {
14 case is INCreatetaskListIntent:
15 return CreateTaskListHandler()
16 default:
17 return nil
18 }
19 }
20 }
21
22
```



図 11.20: Extension のデバッグ（ブレークポイント）

もし、Extension の処理が呼び出されない場合は、収容アプリや Extension での設定が行われているかを確認してくださいまた、Siri の許可は収容アプリの実行時に行うので、少なくとも一度は収容アプリを起動する必要があります。

エクステンションの処理が呼び出されているが、intent オブジェクトが `INCreatetaskListIntent` クラスのオブジェクトでなかったり、フレーズの情報がプロパティに期待した通りに設定されていないなどの場合は、「Siri Intent Query」に設定したフレーズを見直してください。

## 11.8 まとめ

これまで見てきたように、アプリから Siri を利用するには収容アプリとエクステンションの両方について、コードや Info.plist、エンタイトルメントと多岐にわたる作業が必要になるため、敷居が高いと思われるかもしれません。しかし、SiriKit の API はわかりやすく、1つ1つの実装は単一のタスクに集中できるように作られており、とてもシンプルです。また、ドメインや Intent が異なっても基本的な手順は共通です。

まだ利用可能なドメインは決して多くありませんが、これからさらに増加していくことは間違いません。もし、リリースしているアプリがドメインに含まれるのであれば、良い機会ですのでぜひチャレンジしてください。

## 第 12 章

# HomeKit 入門と iOS 11 のアップデート

### 12.1 はじめに

HomeKit は「HomeKit Accessory Protocol に対応したホームオートメーション機器と通信し、これを制御するためのフレームワーク」です<sup>\*1</sup>。

具体的には、HomeKit を利用することで、アプリから照明を操作したり、温度センサーやモーションセンサーをトリガとしてさらに別の機器を操作する設定をしたり、といったことが実現できます。

HomeKit を使ったアプリを開発して App Store に公開する開発者は多くないかもしれません。他方で、Apple は HomeKit を個人の趣味の範囲で楽しむことも許可しています。2016 年までは HomeKit Accessory Protocol<sup>\*2</sup>は一部メーカーにしか公開されていませんでしたが、2017 年からは Apple Developer アカウントさえ持っていれば誰でも参照できるようになりました<sup>\*3</sup>。

自宅やオフィスのオートメーションを実現するための専用アプリを自分で作ることができる、というの、なんともプログラマ冥利に尽きる体験ではないでしょうか。

#### 12.1.1 本章の構成

まず「12.2 HomeKit 入門」で、HomeKit に具備されている機能と利用方法をソースコードを交えて具体的に解説します。

次に「12.3 iOS 11 でのアップデートまとめ」で、iOS 11 でアップデートされた内容について一通り紹介します。

最後に「12.4 HomeKit 実践」では、HomeKit Accessory Simulator の利用方法と、実際に HomeKit 対応製品を購入して利用した実例を紹介します。

オマケとして「12.6 HomeKit お役立ちリファレンス」には、「12.6.2 HMService.serviceType 一覧」など、役立つ一覧表を Apple の API リファレンス +  $\alpha$  で掲載しています。

---

<sup>\*1</sup> Apple の HomeKit Developer Guide の概説より

<sup>\*2</sup> HomeKit 対応製品を作るための仕様

<sup>\*3</sup> <https://developer.apple.com//homekit/specification/>

## 12.2 HomeKit 入門

### 12.2.1 HomeKit でできること

HomeKit でできることは次のとおりです。

- ホームやルームなどの構成の管理
- HomeKit 対応アクセサリの管理と操作
- 多数のアクセサリを一度に操作するシーンの管理
- オートメーションのためのトリガの管理
- ホームを利用するユーザの管理
- ホームやアクセサリの変更監視

### 12.2.2 HomeKit の構成

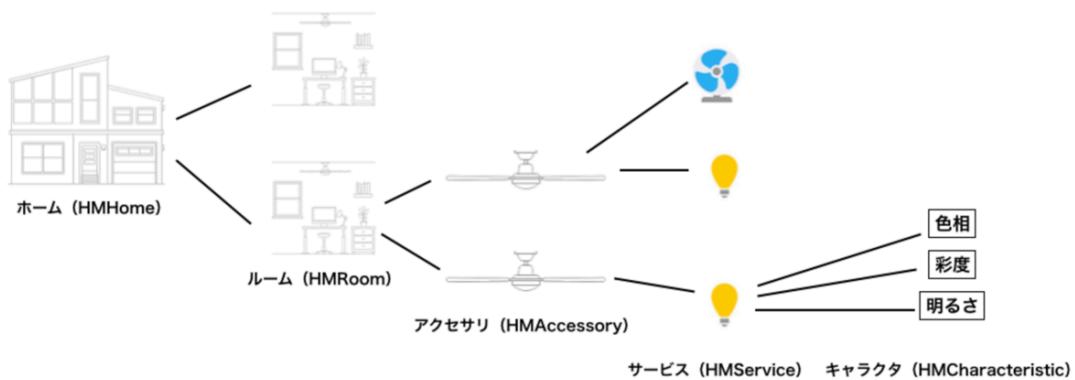


図 12.1: HomeKit 構成全体像

本節では、HomeKit を構成する要素について解説します。

## ホームとアクセサリ

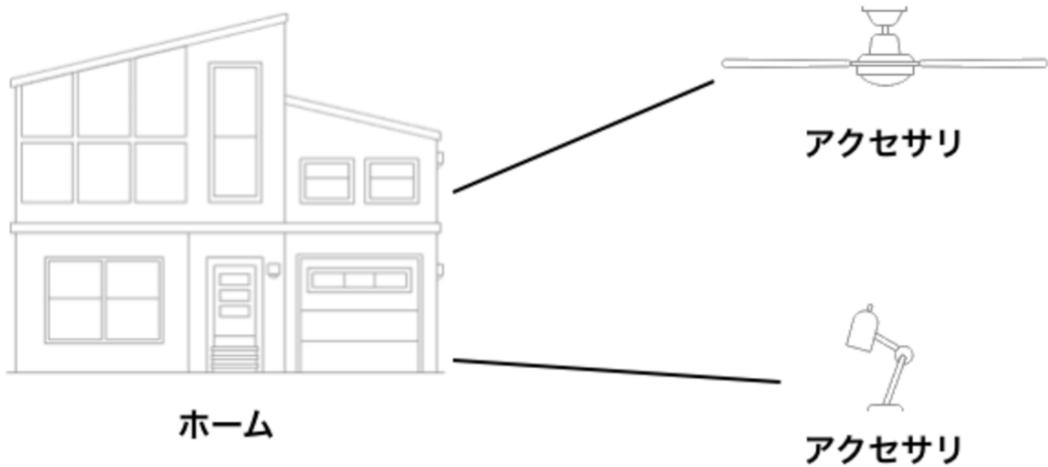


図 12.2: ホームとアクセサリ

HomeKit の最小構成は自宅などのホーム (`HMHome`) に、HomeKit に対応したライトやセンサーなどのアクセサリ (`HMAccessory`) が保持された状態です。例えば、自宅のリビングにライトを 1 つ設置して利用しているだけなら、以降の項で紹介するルームやゾーンなどは一切気にしなくとも利用できます。

ホームにアクセサリを簡単に追加するためのメソッドが `HMHome` に用意されています（リスト 12.1）。

## リスト 12.1: アクセサリの追加

```
home.addAndSetupAccessories { error in
}
```

この 1 つのメソッドを呼び出すだけで HomeKit に用意されたアクセサリ追加の UI が表示され、細かな設定まで全ての面倒を見てくれます（図 12.3）。

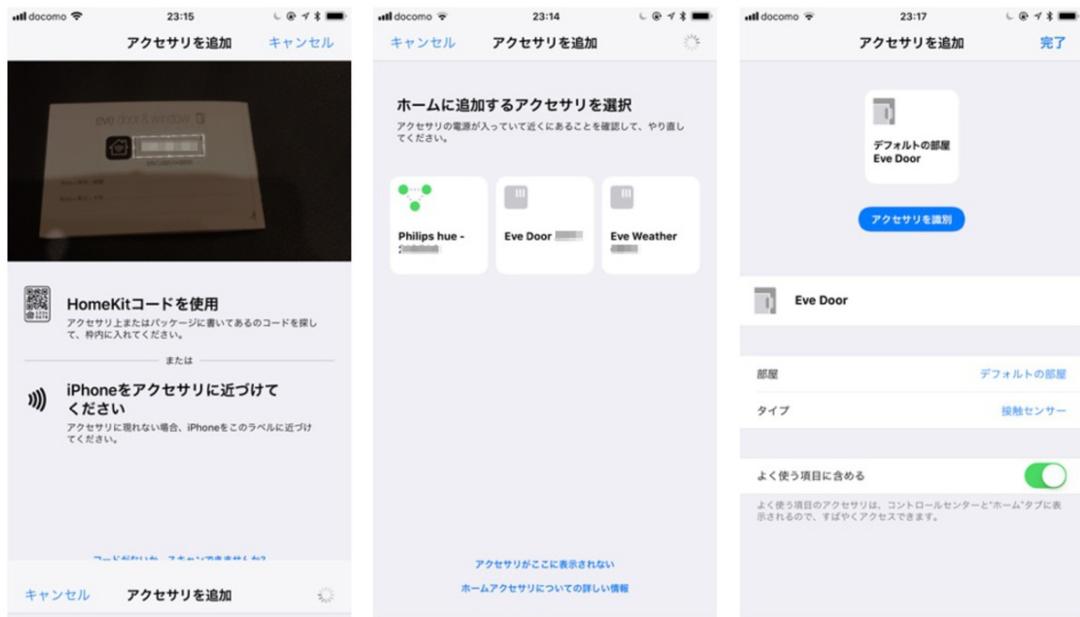


図 12.3: アクセサリ追加の UI

### ホームマネージャとホーム

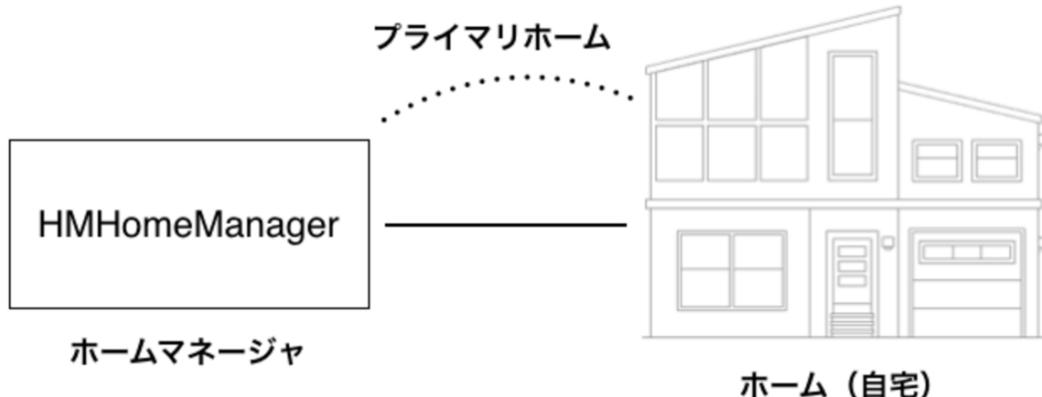


図 12.4: ホームマネージャとホーム

前項のとおり最小構成はホームとアクセサリだけですが、ホームを管理するホームマネージャ (HMHomeManager) が存在します。利用するホームが自宅1つであっても、ホームマネージャを経由してホームにアクセスする必要があります。この場合、そのホームがホームマネージャのプライマリホームとして設定されています（リスト 12.2）。

リスト 12.2: プライマリホームのアクセサリを参照

```
// HMHomeManagerDelegate の homeManagerDidUpdateHomes(_ manager: HMHomeManager)
// が呼ばれるのを待つ必要がある
// これが呼ばれるまで primaryHome は nil のまま

let home = homeManager.primaryHome
let accessories = home?.accessories
```

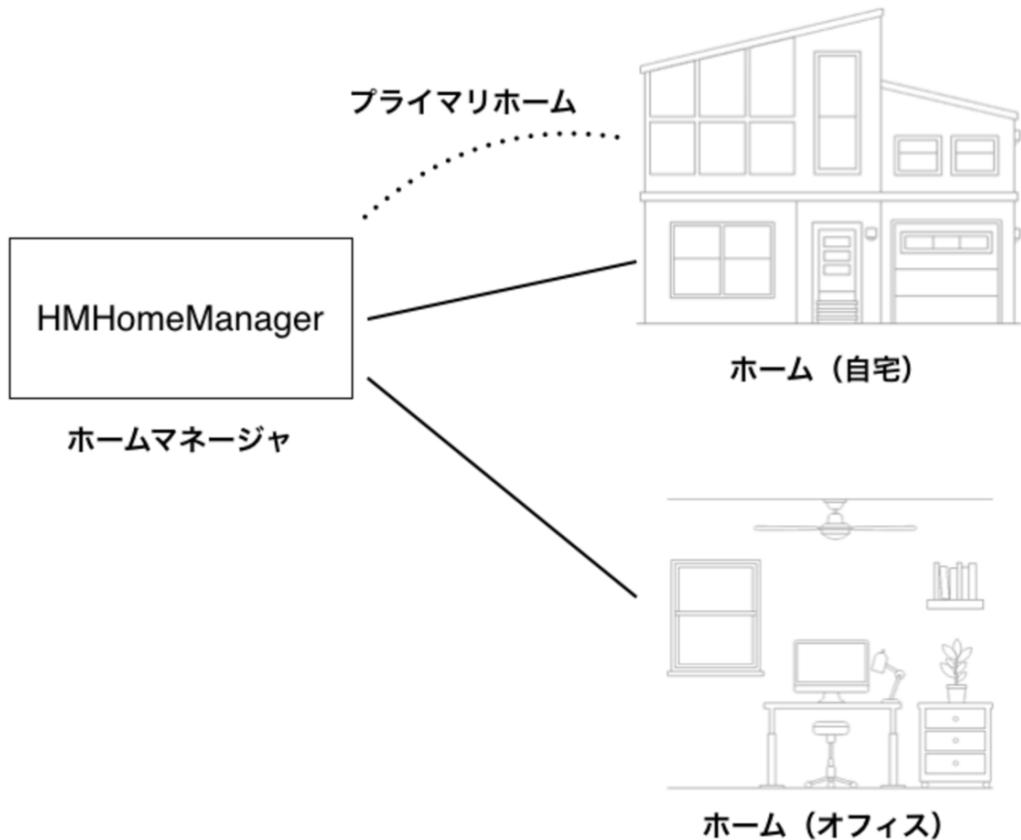


図 12.5: ホームマネージャと複数のホーム

自宅とオフィスなど複数のホームを管理したい場合にはホームマネージャへのホームの追加や削除ができます（リスト 12.3）。ホームを削除するとそのホームに所属するアクセサリなどもまとめて削除されてしまうため注意が必要です。

## リスト 12.3: ホームの追加と削除

```
// 名前を指定してホームを追加
homeManager.addHome(withName: "オフィス") { home, error in
```

```
if let home = home {
 // home が実際に追加されたホーム
}
}

// 任意のホームを削除
homeManager.removeHome(office) { error in
}
```

### プライマリホームの管理

ホームマネージャにはプライマリホームが1つ存在します。ホームマネージャに1つ目のホームを追加したとき、そのホームは自動的にプライマリホームとなります。また、プライマリホームを削除すると、残ったホームのいずれかが自動的にプライマリホームとなります。

もちろん、任意のホームをプライマリホームに指定することもできます（リスト12.4）。

リスト12.4: プライマリホームを指定

```
homeManager.updatePrimaryHome(myHome) { error in
}
```

### ホームとルーム

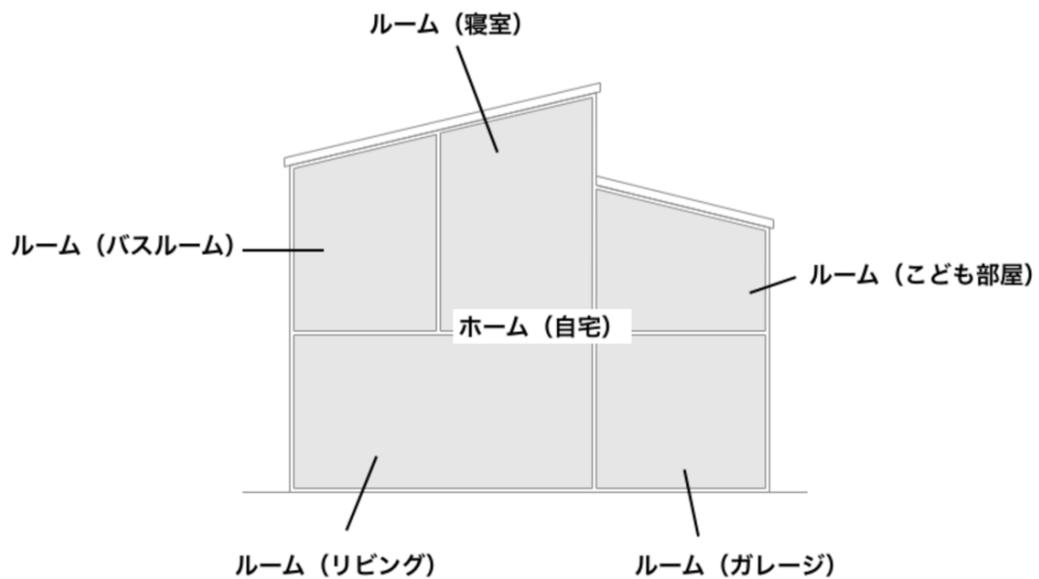


図12.6: ホームとルーム

ホームにはリビング、寝室など複数のルーム (HMRoom) を持たせることができます。「ホームとアクセサリ」のとおりアクセサリを保持するのはホーム (HMHome) ですが、アクセサリをいずれかのルームに割り当てるすることができます（リスト 12.5）。

リスト 12.5: アクセサリをルームに割り当てる

```
// ルームに任意のアクセサリを割り当てる
home.assignAccessory(accessory, to: room) { error in
}
```

ホームにルームを追加するときは必ずルームの名前を指定して追加する必要があります。既存のルームを別のホームに移動することはできません。また、ルームをホームから削除すると、そのルームの存在自体が削除されてしまいます（リスト 12.6）。

リスト 12.6: ホームにルームを追加・削除する

```
// 名前を指定してホームにルームを追加
home.addRoom(withName: name) { room, error in
 if let room = room {
 // room が実際に追加されたルーム
 }
}

// ホームから任意のルームを削除
home.removeRoom(room) { error in
}
```

## デフォルトの部屋

「アクセサリをいずれかのルームに割り当てることができる」と書きましたが、正確にはアクセサリは常にいずれか 1 つのルームに割り当てられた状態になっています。ホームにはデフォルトの部屋 (HMHome の `roomForEntireHome()`) というものがあり、明示的にルームに割り当てられていないアクセサリは全てデフォルトの部屋に割り当てられます。そのアクセサリを他のルームに割り当てるとき、デフォルトの部屋からは自動的に外されます。また、アクセサリが割り当てられているルームが削除された場合、そのアクセサリは再度デフォルトの部屋に戻ります。

なお、いずれかのルームに割り当てられたアクセサリを割り当てるから解除したい場合は、デフォルトの部屋にそのアクセサリを割り当てます<sup>\*4</sup>（リスト 12.7）。デフォルトの部屋にも残さず完全に削除したい場合は HMHome の `removeAccessory(_:completionHandler:)` を使います。

リスト 12.7: アクセサリの割り当てる解除

```
// アクセサリの割り当てる解除
```

<sup>\*4</sup> 割り当てる解除するための専用のメソッドは存在しない

```
home.assignAccessory(accessory, to: home.roomForEntireHome()) { error in
}

// アクセサリをホームから削除
home.removeAccessory(accessory) { error in
}
```

## ルームとゾーン

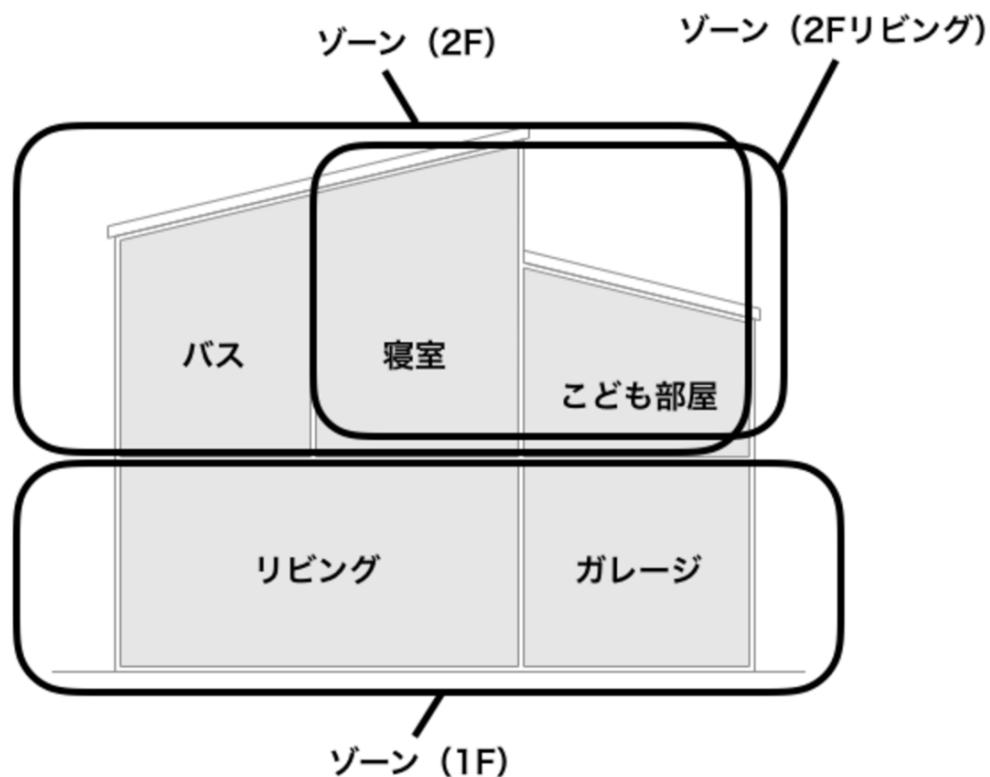


図 12.7: ルームとゾーン

必要であればルーム (HMRoom) をグルーピングするためのゾーン (HMZone) が利用できます。ゾーンもホームに追加・削除して管理します (リスト 12.8)。

## リスト 12.8: ゾーンの追加・削除

```
// 名前を指定してゾーンを追加
home.addZone(withName: name) { zone, error in
 if let zone = zone {
 // zone が実際に追加されたゾーン
 }
}
```

```
}
```

```
// 任意のゾーンを削除
home.removeZone(zone) { error in
}
```

ゾーンには複数のルームを追加することができ、ルームは複数のゾーンに所属することができます（リスト12.9）。

リスト12.9：ゾーンにルームを追加・削除

```
// ゾーンに任意のルームを削除
zone.addRoom(room) { error in
}

// ゾーンから任意のルームを削除
zone.removeRoom(room) { error in
}
```

### 12.2.3 アクセサリへのアクセス

本節では、アクセサリの機能や特性にアクセスし、操作する具体的な方法を解説します。

#### アクセサリとサービス

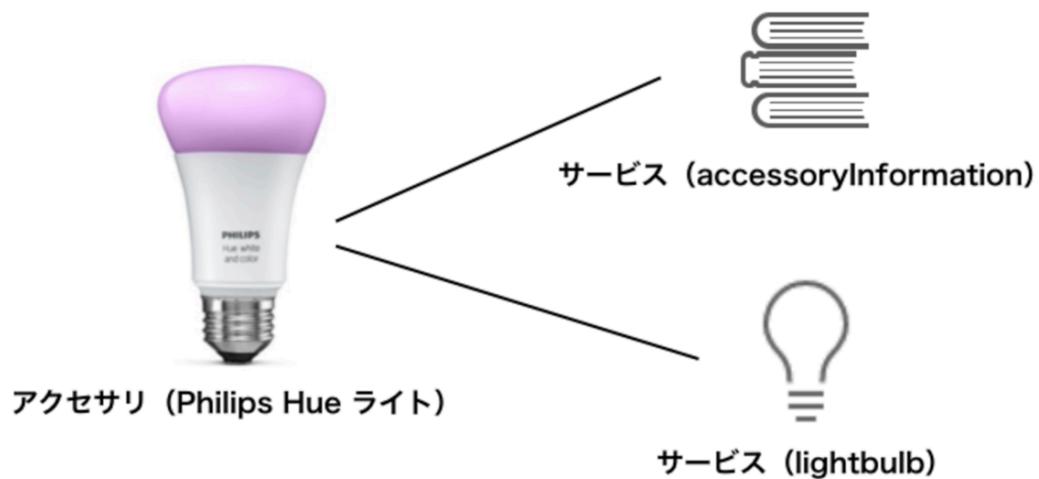


図12.8：アクセサリとサービス

1 つのアクセサリ (**HMAccessory**) は複数の機能を持っています。これらアクセサリの機能のこととを HomeKit ではサービス (**HMServices**) と呼びます。サービスには様々な種類があり、iOS 11 に定義されている全サービスを「12.6.2 HMServices.serviceType 一覧」で紹介しています。

各アクセサリのサービスは **services** プロパティで取得できます（リスト 12.10）。

リスト 12.10: アクセサリのサービスを参照

```
let services = accessory.services
```

例えば Philips Hue のライトには表 12.1 の 2 種のサービスがあります。

表 12.1: Philips Hue ライトのサービス

HMServiceType～	説明
Lightbulb	電球
AccessoryInformation	アクセサリ情報サービス

**AccessoryInformation** は単にファームウェアバージョンやシリアル番号などのメタ情報を保持するだけのサービスですので、電球の明るさなどを取得・変更するには **Lightbulb** を利用します。

## サービスとキャラクタ

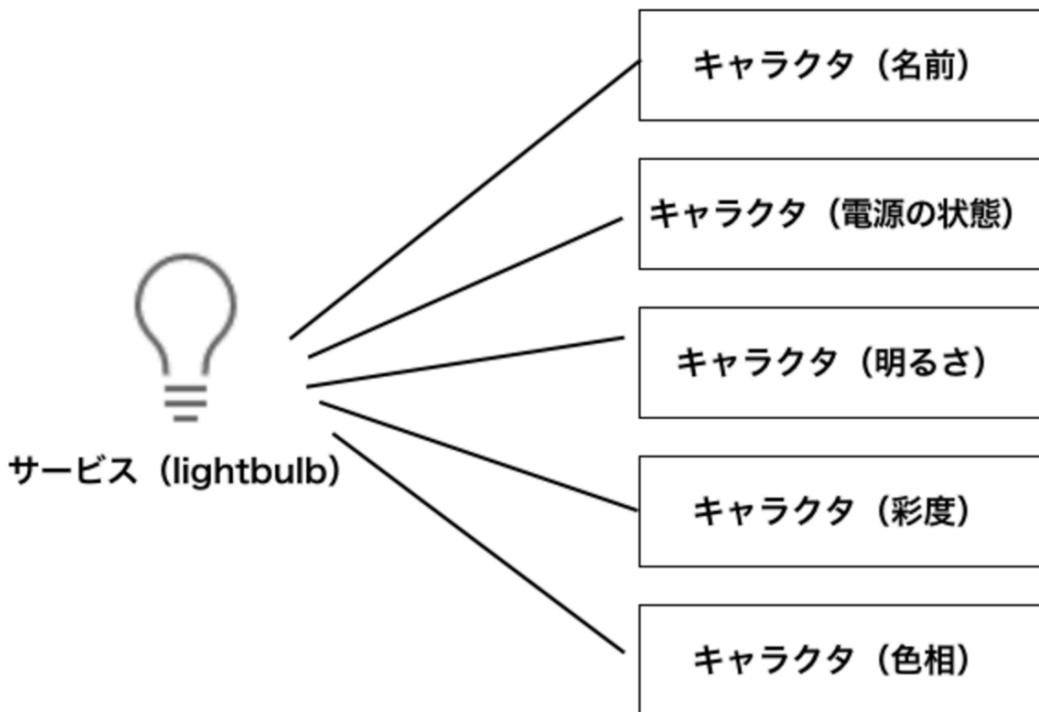


図 12.9: サービスとキャラクタ

1つのサービス(HMService)は複数の特性を持ちます。これらサービスの特性のことをHomeKitではキャラクタ(HMCharacteristic)と呼びます。キャラクタにも様々な種類があり、iOS 11に定義されている全キャラクタを「12.6.3 HMCharacteristic.characteristicType一覧」で紹介しています。

各サービスのキャラクタは characteristics プロパティで取得できます（リスト 12.11）。

リスト 12.11: サービスのキャラクタを参照

```
let characteristics = service.characteristics
```

一例として Lightbulb サービスには表 12.2 の 5 種のキャラクタがあります<sup>\*5</sup>。

例えばライトが点灯しているかどうかを知るためには PowerState のキャラクタを取得します（リスト 12.12）。キャラクタの種類の判別には characteristicType プロパティを利用し、キャラクタの種類に応じて value プロパティを変換して参照する必要があります。value プロパティが取

<sup>\*5</sup> 正確には未定義（カスタム）の特性をもう1つ持っています

表12.2: Lightbulbのキャラクタ

HMCharacteristicType～	説明
Name	名前
Hue	色相
Saturation	彩度
Brightness	明るさ
PowerState	電源の状態

り得る値の詳細については metadata プロパティで知ることができます。metadata プロパティについては次節の「キャラクタのメタデータ」で紹介します。

リスト12.12: PowerStateの参照

```
// characteristicsの中からPowerStateを抽出
let service = home.servicesWithTypes([HMSERVICETypeLightbulb])?.first
let candidates = service?.characteristics
 .filter { $0.characteristicType == HMCharacteristicTypePowerState }

guard let powerState = candidates?.first else {
 return
}

// powerState.valueに取得済みのvalueが入っているが
// readValueでデバイスから最新のvalueを再読み込み可能
powerState.readValue { error in
 // PowerStateはBool(NSNumber)でvalueが返ってくる
 guard let value = powerState.value as? Bool else {
 return
 }
 // ライトが点灯中ならtrue
 // ライトが消灯中ならfalse
 print("# powerState: \(value)")
}
```

もちろんライトの点灯・消灯を切り替えることもできます（リスト12.13）。

リスト12.13: PowerStateの上書き

```
// 点灯したい場合はtrue
// 消灯したい場合はfalse
let value = true
powerState.writeValue(value) { error in
}
```

### キャラクタのメタデータ

キャラクタ(HMCharacteristic)の`metadata`プロパティに入っているメタデータ(HMCharacteristicMetadata)を参照することで、キャラクタの`value`プロパティが取り得る値の詳細を知ることができます。メタデータには表12.3に示すプロパティがあります。

表12.3: HMCharacteristicMetadataのプロパティ一覧

プロパティ	型	説明
<code>format</code>	<code>String?</code>	<code>value</code> のフォーマット
<code>units</code>	<code>String?</code>	<code>value</code> の単位
<code>minimumValue</code>	<code>NSNumber?</code>	最小値
<code>maximumValue</code>	<code>NSNumber?</code>	最大値
<code>stepValue</code>	<code>NSNumber?</code>	最小単位
<code>validValues</code>	<code>[NSNumber]?</code>	<code>value</code> が取り得る値
<code>maxLength</code>	<code>NSNumber?</code>	<code>value</code> が許容するStringの長さ
<code>manufacturerDescription</code>	<code>String?</code>	製造元による説明

まず見るべきプロパティは`format`です。`format`には"int"、"bool"といった`value`プロパティに入る値のフォーマットを示す文字列が入っています。`units`プロパティは`value`に単位がある場合にその単位を示す文字列を持ちます。`format`と`units`の値の一覧をそれぞれ「12.6.4 HMCharacteristicMetadata.format一覧」と「12.6.5 HMCharacteristicMetadata.units一覧」で紹介しています。`minimumValue`、`maximumValue`、`stepValue`は、それぞれ`value`が数値型の場合の最小値、最大値、最小単位を表します。

一例としてPhilips Hueのライトの`Brightness`キャラクタ(明度・明るさ)の実際の値を表12.4に示します。

表12.4: Philips Hue ライトの明るさのメタデータ

プロパティ	実際の値
<code>format</code>	int
<code>units</code>	percentage
<code>minimumValue</code>	0
<code>maximumValue</code>	100
<code>stepValue</code>	1
<code>validValues</code>	
<code>maxLength</code>	
<code>manufacturerDescription</code>	Brightness

これにより、`Brightness`キャラクタの`value`は0~100までの数値で1単位で値の上げ下げができることがわかります。

`validValues`プロパティには`value`が取り得る数値が限定される場合に、サポートされる数値が列挙されます。例えば`AirQuality`キャラクタ(空気質)が取り得る値として、表12.5に示す

値が定義されています。このうち数個の値しかサポートしていないケースでは`[0,1,2,4]`といった`NSNumber`の配列が`validValues`プロパティにセットされています。

表 12.5: AirQuality の値一覧

値	rawValue
<code>unknown</code>	0
<code>excellent</code>	1
<code>good</code>	2
<code>fair</code>	3
<code>inferior</code>	4
<code>poor</code>	5

最後に、`maxLength`プロパティは`value`に入る文字列の最大長を示し、`format`が`"string"`の場合のみ使われます。

#### サービスとサービスグループ

必要であれば、複数のサービス（`HMServices`）をサービスグループ（`HMServicesGroup`）としてまとめることができます。これにより、複数サービスの取り扱いが簡単になります<sup>\*6</sup>。例えば、自宅の間接照明だけをまとめた「間接照明すべて」というサービスグループを作ることができます（リスト 12.14）。

リスト 12.14: サービスグループの作成・削除

```
// 間接照明をまとめたサービスグループを作る
home.addServiceGroup(withName: "間接照明すべて") { serviceGroup, error in
 guard let serviceGroup = serviceGroup else {
 return
 }

 // すべての間接照明をサービスグループに追加
 for service in allIndirectLights {
 serviceGroup.addService(service) { error in
 }
 }
}

// サービスグループから任意のサービスを削除
serviceGroup.removeService(service) { error in
}

// ホームからサービスグループを削除
home.removeServiceGroup(serviceGroup) { error in
}
```

<sup>\*6</sup> `HMServicesTypeAccessoryInformation`など`serviceType`によってはサービスグループに追加できないものがあります

サービスグループ (`HMServiceGroup`) の `name` プロパティはそのまま Siri で利用できます。例えば、Siri に「間接照明すべて消す」という命令をすることで、サービスグループにまとめた全ての間接照明を一度に消灯することができます。

#### 12.2.4 アクションとシーンの利用

本節では、アクセサリを制御するアクションや、複数のアクションをまとめるシーンについて解説します。

##### アクション

HomeKit の要素を更新するアクション (`HMAction`) という概念があります。

iOS 11 でサポートされているアクションは、`HMCharacteristicWriteAction` という各アクセサリ (`HMAccessory`) のキャラクタ (`HMCharacteristic`) の `value` プロパティを更新するアクション1つだけです。例えば、照明の `Saturation` キャラクタ (彩度) を最も大きく (100) するアクションはリスト 12.15 のように作ります。

リスト 12.15: 照明の明るさを最大にするアクション

```
// 照明のキャラクタの中から Saturation を抽出
let candidates = lightbulb.characteristics
 .filter { $0.characteristicType == HMCharacteristicTypeSaturation }

guard let saturation = candidates.first else {
 return
}

// アクションの作成
let action = HMCharacteristicWriteAction(
 characteristic: saturation,
 targetValue: NSNumber(value: 100)
)
```

##### シーン

また、複数のアクションをまとめるシーン (`HMActionSet`) を作り (リスト 12.16)、任意に実行することができます (リスト 12.17)。

リスト 12.16: シーンを作成・削除

```
// シーンを作成
home.addActionSet(withName: "サンプル") { actionSet, error in
 guard let actionSet = actionSet else {
 return
 }
```

```
// 作成したシーンにアクションを追加
actionSet.addAction(action) { error in
}
}

// シーンを削除
home.removeActionSet(actionSet) { error in
}
```

リスト 12.17: シーンを実行

```
home.executeActionSet(actionSet) { error in
}
```

シーンを利用することで、例えば、帰宅したときに

- リビングの照明を点灯する
- 玄関外の照明を消灯する
- ヒーターをつける

といった一連のアクションをまとめて実行できます。

シーンは自分で作成できるほか、HomeKit に4つビルトインされています（表 12.6）。

表 12.6: ビルトインされているシーン一覧

HMAccountType～	name
WakeUp	おはよう
Sleep	おやすみ
HomeArrival	ただいま
HomeDeparture	行ってきます

### 12.2.5 トリガによるオートメーション

HomeKit にはシーン（HMAccount）を実行するためのトリガ（HMTrigger）という機能があります。トリガには、

- タイマートリガ（HMTimerTrigger）
- イベントトリガ（HMEventTrigger）

の2種があります。これらのトリガを活用することで、

- 日の入りと同時にクリスマスツリーのライトを点灯させる
- リビングの気温が28度を超えたらファンを回す

といったオートメーションを実現できます。

### タイマートリガ

タイマートリガ (HMTimerTrigger) は主に、指定した日時にシーンを実行するためのトリガです。HMTimerTrigger は、発火する日時を指定して作成します（リスト 12.18）。

リスト 12.18: ‘HMTimerTrigger’を作成

```
// UIDatePicker の日付を利用
guard let date = datePicker?.date else {
 return nil
}

let calendar = Calendar(identifier: .gregorian)

// 秒が 0 以外だと HMTimerTrigger の作成に失敗するため注意
let comp = calendar.dateComponents(
 [.minute, .hour, .day, .month, .year, .era],
 from: date
)
let fireDate = calendar.date(from: comp)

// HMTimerTrigger を作成
let trigger = HMTimerTrigger(
 name: "明日の 15 時",
 fireDate: fireDate,
 timeZone: datePicker?.timeZone,
 recurrence: nil,
 recurrenceCalendar: nil
)
```

`fireDate` に指定する日付は、将来の日付で、秒は 0 でなければならないことに注意が必要です<sup>\*7</sup>。ここで作成したトリガはホームに追加して利用します（リスト 12.19）。

リスト 12.19: トリガをホームに追加・削除

```
// トリガを追加
home.addTrigger(trigger) { error in
}

// トリガを削除
home.removeTrigger(trigger) { error in
}
```

なお、HMTimerTrigger の `isEnabled` プロパティはデフォルトで `false` になっており、トリガは無効な状態です。トリガを有効にするには、シーンを追加したうえで `enable(_:completionHandler:)` を呼んで `isEnabled` を `true` に更新します（リスト 12.20）。

<sup>\*7</sup> UIDatePicker の `date` プロパティをそのまま利用すると秒が 0 以外の場合があります

また、繰り返し設定を指定していない場合、`isEnabled` プロパティは一度実行された時点で `false` に戻されます。

リスト 12.20: トリガを有効にする

```
trigger.enable(true) { error in
}
```

### イベントトリガ

イベントトリガ (`HMEventTrigger`) は、特定のイベント (`HMEvent`) が発生したときにシーンを実行するためのトリガです。イベントには、iOS 10 以前から利用できた

- `HMCharacteristicEvent`
- `HMLocationEvent`

の 2 つと、iOS 11 で追加された

- `HMCalendarEvent`
- `HMSignificantTimeEvent`
- `HMCharacteristicThresholdRangeEvent`
- `HMPresenceEvent`
- `HMDurationEvent`

5 つの計 7 つがあります。iOS 11 で加えられたものについては「12.3 iOS 11 でのアップデートまとめ」で紹介しますので、ここでは `HMCharacteristicEvent` と `HMLocationEvent` の 2 つについて解説します。

#### HMCharacteristicEvent

`HMCharacteristicEvent` はキャラクタ (`HMCharacteristic`) の `value` プロパティが指定の値になったときに発生するイベントです。このイベントをトリガ (`HMEventTrigger`) に設定することで、例えば、ドアが開いた時に電球を点灯するといったオートメーションが実現できます。

「ドアが開いたとき」をトリガとする `HMEventTrigger` を作成するコードをリスト 12.21 に示します。

リスト 12.21: ドアが開いたときに発火するトリガ

```
let contactState = //< ドアの開閉状態を示す ContactState キャラクタ
// .none なら非接触（ドアが開いている状態）
let value = NSNumber(value: HMCharacteristicValueContactState.none.rawValue)

// HMCharacteristicEvent を使ってトリガ作成
let event = HMCharacteristicEvent(characteristic: contactState, triggerValue: value)
let trigger = HMEventTrigger(name: "ドアが開いたら", events: [event], predicate: nil)
```

### HMLocationEvent

`HMLocationEvent` は自分が指定したエリアに入った（もしくは出た）タイミングで発生するイベントです。このイベントをトリガ (`HMEventTrigger`) に設定することで、例えば、自宅から半径 1000 メートル以内に入ったときに玄関の電灯を点ける、といったオートメーションが実現できます（リスト 12.22）。

リスト 12.22: 自宅から半径 1000 メートル以内に入ったときに発火するトリガ

```
// 自宅から半径 1000 メートルの region
let center = CLLocationCoordinate2D(latitude: 35.70206910, longitude: 139.77532690)
let radius = 1000.0
let region = CLCircularRegion(center: center, radius: radius, identifier: "MyHome")

// region に入ったときのみ対象
region.notifyOnEntry = true
region.notifyOnExit = false

// HMLocationEvent を使ってトリガ作成
let event = HMLocationEvent(region: region)
let trigger = HMEventTrigger(name: "半径 1km 以内", events: [event], predicate: nil)
```

### `predicate` プロパティ

イベントトリガ (`HMEventTrigger`) には `predicate` プロパティがあり、ここにシーンを実行するための条件を加えることができます。例えば、イベントトリガに `HMLocationEvent` を設定し、指定したエリアに自分が入ったとしても、この `predicate` プロパティに追加した条件を満たしていないければシーンは実行されません。

HomeKit には、この条件 (`NSPredicate`) を作るためのメソッドがいくつか用意されています（このうち iOS 11 で追加されたものについては「`predicate` 用メソッドのアップデート」で紹介します）。

まず、「指定した時分まで」「指定した時分より後」という条件を作るための

- `predicateForEvaluatingTrigger(occurringBefore:)`
- `predicateForEvaluatingTrigger(occurringAfter:)`

があります。それぞれ指定したい時分を `DateComponents` 型で指定するだけです。具体的に「18 時以降」という条件を指定するコードの例をリスト 12.23 に示します。

リスト 12.23: 18 時以降という条件を指定

```
// 18 時以降
```

```
let comp = DateComponents(hour: 18)
let predicate = HMEventTrigger.predicateForEvaluatingTrigger(occurringBefore: comp)

let trigger = HMEventTrigger(name: "18 時以降", events: [event], predicate: predicate)
```

次に、特定のキャラクタ (HMCharacteristic) の value プロパティが任意の範囲かどうかという条件を作るための

- `predicateForEvaluatingTrigger(_:relatedBy:toValue:)`

があります。値の範囲の指定 (`relatedBy`) には `NSComparisonPredicate.Operator` 型を利用し、`.greaterThan`、`.lessThanOrEqualTo` など柔軟な範囲指定ができます。具体的に「気温が 28 度を超える」という条件を指定するコードの例をリスト 12.24 に示します。

リスト 12.24: 気温が 28 度を超えるという条件を指定

```
let temperature = //< 現在の気温のキャラクタ

// 気温が 28 度を超える場合
let predicate = HMEventTrigger.predicateForEvaluatingTrigger(
 temperature,
 relatedBy: .greaterThan,
 toValue: 28.0
)

let trigger = HMEventTrigger(name: "28 度超", events: [event], predicate: predicate)
```

## 12.2.6 ユーザの管理

自分が作成・管理しているホームにゲストユーザを追加することもできます。`HMHome` の `manageUsers(completionHandler:)` を呼ぶとユーザ管理用の画面が表示されます（リスト 12.25）。

リスト 12.25: ユーザを管理する

```
home.manageUsers { error in
}
```

この画面でユーザの追加や削除など全ての操作ができます（図 12.10）。



図 12.10: ユーザの管理画面

現在利用中のユーザ（`HMUser`）は `HMHome` の `currentUser` プロパティで参照でき、`HMUser` の `name` プロパティでユーザ名を、`isAdministrator` プロパティで管理者かどうかをそれぞれ取得できます。

### 12.2.7 変更の監視

HomeKit が取り扱う情報は常に他のアプリから変更される可能性があります。そのため、HomeKit が持つデリゲートメソッドを利用し、外部からの変更を検知してアプリの状態を正しく保つ必要があります。

なお、これらのデリゲートメソッドは変更を行ったプログラム自体では呼び出されません。例えば、プログラムでホーム（`HMHome`）にルーム（`HMRom`）を追加したとしても、そのプログラム自身の `HMHomeDelegate.home(_ home: HMHome, didAdd room: HMRom)` が呼ばれることがありません。

### 12.2.8 カメラの利用

HomeKit に対応したカメラデバイスがあれば、カメラで撮影している動画をアプリの画面に表示したり、静止画を撮ったり、マイクやスピーカーの設定を更新したり、といったことが簡単に実現できます。

これらの操作をするには、カメラ機能を持ったアクセサリ（`HMAccessory`）の `cameraProfiles` プロパティから `HMCameraProfile` のインスタンスを取得し、表 12.7 に示すプロパティ群を利用します。

表 12.7: `HMCameraProfile` のプロパティ一覧

プロパティ名	型	説明
<code>streamControl</code>	<code>HMCameraStreamControl</code>	ビデオストリーム（動画）
<code>snapshotControl</code>	<code>HMCameraSnapshotControl</code>	スナップショット（静止画）
<code>microphoneControl</code>	<code>HMCameraAudioControl</code>	マイク
<code>speakerControl</code>	<code>HMCameraAudioControl</code>	スピーカー
<code>settingsControl</code>	<code>HMCameraSettingsControl</code>	カメラ設定

以降の節では、各プロパティの利用方法について紹介していきます。

#### `streamControl`

`HMCameraProfile` の `streamControl` プロパティ（`HMCameraStreamControl`）を使うことで、カメラで撮影中の動画をアプリ内で表示することができます。

具体的には、

1. `streamControl` の `delegate` を設定
2. `HMCameraView` (`UIView` のサブクラス) を適当な場所に追加する
3. `HMCameraStreamControl` の `startStream()` メソッドで撮影を開始する

4. HMCameraStreamControlDelegate の cameraStreamControl(\_:) を受けて、HMCameraView の cameraSource プロパティに、HMCameraStreamControl の cameraStream プロパティをセットする
  5. 必要なくなったら HMCameraStreamControl の stopStream() メソッドで撮影を停止する
- とするだけです（リスト 12.26、リスト 12.27）。

リスト 12.26: HMCameraView を設置し撮影開始

```
cameraProfile.streamControl?.delegate = self

let view = //< CameraView を add する UIView

// HMCameraView を設置
let cameraView = HMCameraView()
cameraView.frame = view.bounds
cameraView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
view.addSubview(cameraView)
self.cameraView = cameraView

// 撮影を開始
cameraProfile.streamControl?.startStream()
```

リスト 12.27: cameraSource を設定

```
extension SampleViewController: HMCameraStreamControlDelegate {
 func cameraStreamControlDidStartStream(_ cameraStreamControl: HMCameraStreamControl) {
 cameraView?.cameraSource = cameraStreamControl.cameraStream
 }
}
```

これだけで、設置した HMCameraView の枠内に、撮影中の動画がリアルタイムに描画されます（図 12.11）。

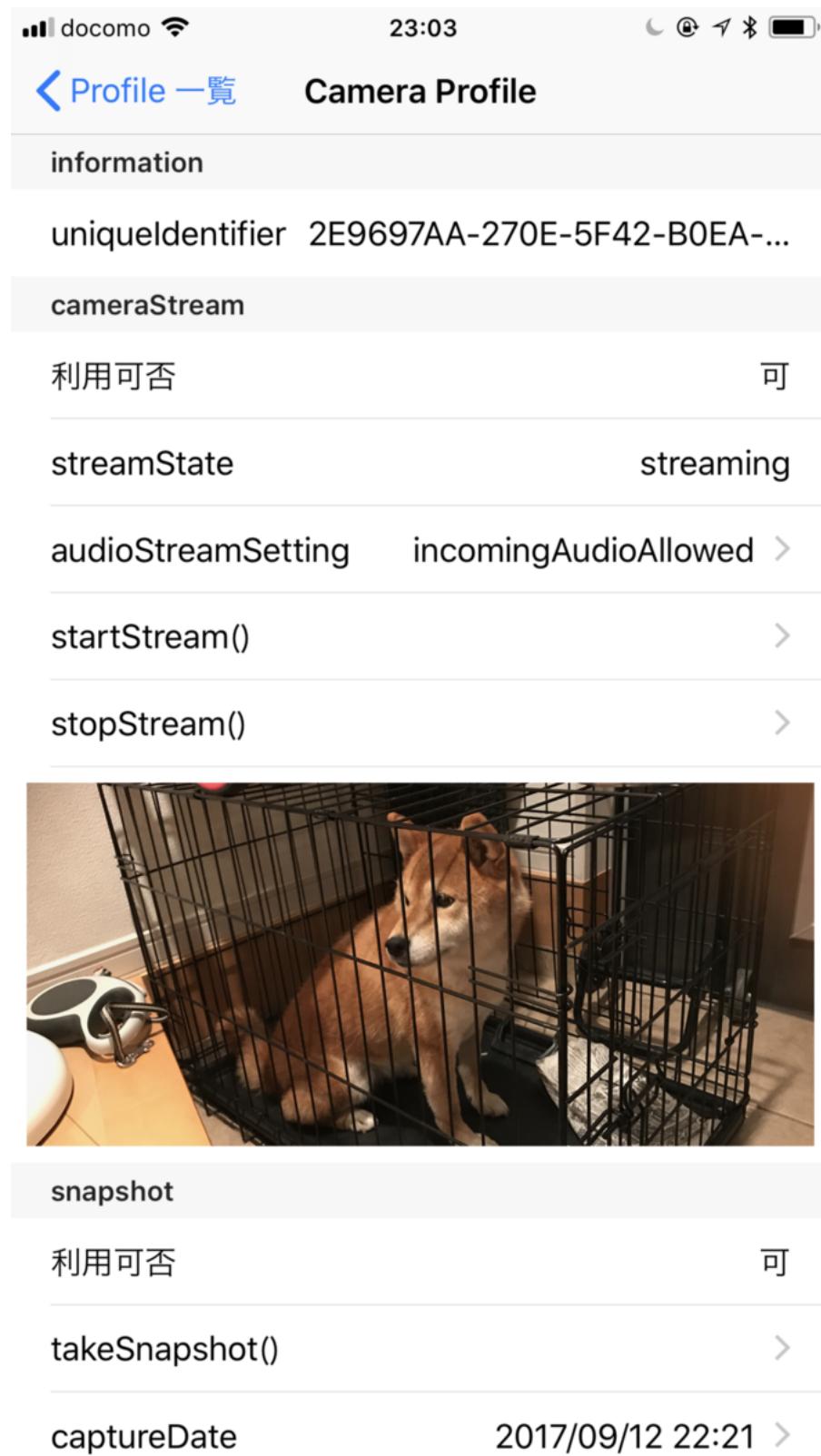


図 12.11: 撮影中の動画を画面に表示

また、動画撮影の状態は `HMCameraStreamControl` の `streamState` プロパティで参照できます。`streamState` プロパティが取り得る値は `HMCameraStreamState` という `enum` として定義されています（表 12.8）。

表 12.8: `HMCameraStreamState` の値一覧

値	説明
<code>notStreaming</code>	撮影中ではない
<code>starting</code>	撮影開始中
<code>stopping</code>	撮影停止中
<code>streaming</code>	撮影中

この他、撮影中の音声の設定を `HMCameraStreamControl` の `cameraStream` の `audioStreamSetting` プロパティで設定できます。`audioStreamSetting` プロパティが取り得る値は `HMCameraAudioStreamSetting` という `enum` として定義されています（表 12.9）。

表 12.9: `HMCameraAudioStreamSetting` の値一覧

値	説明
<code>muted</code>	音声は利用しない
<code>incomingAudioAllowed</code>	カメラ側のマイクによる収音のみ許可
<code>bidirectionalAudioAllowed</code>	カメラ側のマイクによる収音も、カメラのスピーカーからこちらの音声を再生することも許可 <sup>*8</sup>

`audioStreamSetting` プロパティの更新には、`updateAudioStreamSetting(_:completionHandler:)` メソッドを利用します（リスト 12.28）。

リスト 12.28: `audioStreamSetting` の更新

```
guard let cameraStream = cameraProfile.streamControl?.cameraStream else {
 return
}

cameraStream.updateAudioStreamSetting(.incomingAudioAllowed) { error in
}
```

### snapshotControl

`HMCameraProfile` の `snapshotControl` プロパティ (`HMCameraSnapshotControl`) を使うことで、カメラで静止画を撮影し、アプリ内で表示することができます。

具体的には、

1. `snapshotControl` の `delegate` を設定
2. `HMCameraView` (`UIView` のサブクラス) を適当な場所に追加する

3. HMCameraSnapshotControl の takeSnapshot() メソッドで静止画を撮影する
4. HMCameraSnapshotControlDelegate の cameraSnapshotControl(\_:didTake:error) を受けて、引数の HMCameraSnapshot を HMCameraView の cameraSource プロパティにセットする

とするだけです（リスト 12.29、リスト 12.30）。streamControl の利用方法とほぼ同じです。

リスト 12.29: HMCameraView を設置し静止画撮影

```
cameraProfile.snapshotControl?.delegate = self

let view = //< CameraView を add する UIView

// HMCameraView を設置
let cameraView = HMCameraView()
cameraView.frame = view.bounds
cameraView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
view.addSubview(cameraView)
self.cameraView = cameraView

// 静止画撮影
cameraProfile.snapshotControl?.takeSnapshot()
```

リスト 12.30: cameraSource を設定

```
extension SampleViewController: HMCameraSnapshotControlDelegate {
 func cameraSnapshotControl(_ cameraSnapshotControl: HMCameraSnapshotControl,
 didTake snapshot: HMCameraSnapshot?, error: Error?) {
 cameraView?.cameraSource = snapshot
 }
}
```

これにより、HMCameraView の枠内に、撮影した静止画が描画されます（図 12.12）。

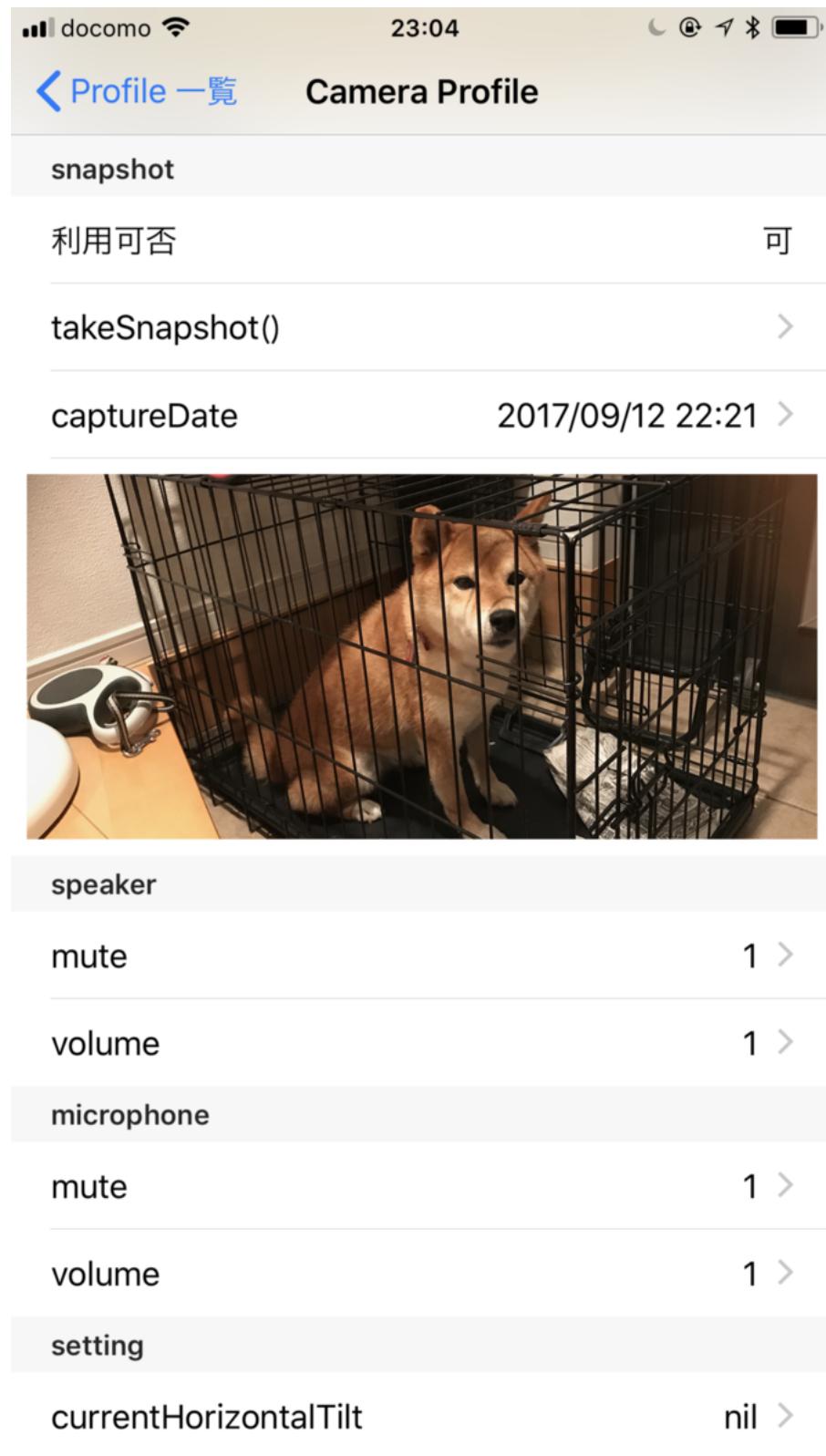


図 12.12: 静止画を撮影して表示

もう 1 つ、`HMCameraSnapshotControl` には、最後に撮影した静止画を参照できる `mostRecentSnapshot` プロパティがあります。ここに `HMCameraSnapshot` のインスタンスが入っている場合、これを `HMCameraView` の `cameraSource` プロパティにセットして、最後に撮影した静止画を画面に表示することができます。対象の静止画が撮影された日時は、`HMCameraSnapshot` の `captureDate` プロパティで参照できます。

また、他のアプリなどから静止画が撮影された場合、それを `HMCameraSnapshotControlDelegate` の `cameraSnapshotControlDidUpdateMostRecentSnapshot(_:)` メソッドで検知することができます。

#### microphoneControl/speakerControl

`HMCameraProfile` には、マイクの設定をするための `microphoneControl` プロパティと、スピーカーの設定をするための `speakerControl` プロパティがあります。

それぞれに `HMCameraAudioControl` のインスタンスがセットされており、`HMCameraAudioControl` のプロパティから、消音と音量のキャラクタ (`HMCharacteristic`) を参照できます (表 12.10)。

表 12.10: `HMCameraAudioControlProperties` のプロパティ一覧

プロパティ名	<code>HMCharacteristicType</code> ~	説明
<code>mute</code>	<code>Mute</code>	消音
<code>volume</code>	<code>Volume</code>	音量

例えば、スピーカーの音量を変更するにはリスト 12.31 のようにします。

リスト 12.31: スピーカーの音量を変更

```
guard let volume = cameraProfile.speakerControl?.volume else {
 return
}

volume.writeValue(50) { error in }
```

#### settingsControl

`settingsControl` には `HMCameraSettingsControl` のインスタンスがセットされています。`HMCameraSettingsControl` は表 12.11 に示す 9 つのキャラクタ (`HMCharacteristic`) をプロパティとして持ちます。

デバイスごとに設定可能なキャラクタのインスタンスが保持されており、サポート外のキャラクタは `nil` になっています。例えば、カメラの画像を反転させるにはリスト 12.32 のようにします。

リスト 12.32: カメラの画像を反転させる

表 12.11: HMCameraSettingsControl のプロパティ一覧

プロパティ名	HMCharacteristicType～	説明
currentHorizontalTilt	CurrentHorizontalTilt	現在の横方向の傾斜角度
currentVerticalTilt	CurrentVerticalTilt	現在の縦方向の傾斜角度
targetHorizontalTilt	TargetHorizontalTilt	横方向の傾斜角度の目標値
targetVerticalTilt	TargetVerticalTilt	縦方向の傾斜角度の目標値
digitalZoom	DigitalZoom	デジタルズーム
opticalZoom	OpticalZoom	光学ズーム
imageMirroring	ImageMirroring	画像反転
imageRotation	ImageRotation	画像回転
nightVision	NightVision	ナイトビジョン

```
guard let imageMirroring = cameraProfile.settingsControl?.imageMirroring else {
 return
}

imageMirroring.writeValue(true) { error in
}
```

### 12.2.9 Siri の利用

HomeKit 構成要素の `name` プロパティに設定した名前を利用し、Siri へ各種操作を命令することができます。現在サポートされているのは以下 6 つです。

- ホーム（`HMHome`）
- ルーム（`HMRoom`）
- ゾーン（`HMZone`）
- サービス（`HMService`）
- サービスグループ（`HMServiceGroup`）
- シーン（`HMActionSet`）

#### ホーム指定での操作

例えばホーム（`HMHome`）の `name` プロパティに「オフィス」と設定している場合、Siri に

「オフィス」の電気をつけて

と命令することでそのホームの全ての電気を点けることができます。なお、

電気をつけて

と名前を省略して命令をした場合、プライマリホーム（primaryHome）のすべての電気が点灯します。

#### ルーム・ゾーン指定での操作

ルーム（HMRoom）の name プロパティを使って、

リビングの電気をつけて

と命令することができます。また、複数のルームを束ねたゾーン（HMZone）の name プロパティを使って、

「2階」の電気をつけて

と命令することもできます。

#### アクセサリ指定での操作

現在のところ、アクセサリ（HMAccessory）の name プロパティは Siri に認識されないようです。

#### サービス・サービスグループ指定での操作

サービス（HMService）の name プロパティを指定して、

「間接照明」をつけて

と命令することも、もちろんできます。なお、サービス（HMService）の name プロパティは重複可能で、複数のサービスに同じ名前をつけ、1回の命令でそれら全てを操作することもできます。

また、複数のサービスをサービスグループ（HMServiceGroup）にまとめ、サービスグループの name プロパティ指定で、

「関節照明すべて」をつけて

と命令することもできます。

### シーン指定での操作

シーン (`HMAbstractAction`) の `name` プロパティを指定して、

「スペシャル設定」にして

と命令し、複数のアクセサリの複数のキャラクタを、シーンに登録した状態にまとめて変更することができます。

### トリガ指定での操作

現在のところ、トリガ (`HMTimerTrigger`) の `name` プロパティは Siri に認識されないようです。

## 12.3 iOS 11 でのアップデートまとめ

### 12.3.1 新しいイベントの追加

iOS 11 ではイベントトリガ (`HMEventTrigger`) に設定可能なイベント (`HMEvent`) がいくつか追加されました。

#### `HMCalendarEvent`

`HMCalendarEvent` の追加により日時指定での繰り返しのオートメーションが簡単に実現できるようになります<sup>\*9</sup>。例えば、午前 8:30 に発火するイベントトリガを作成するにはリスト 12.33 のようにします。

リスト 12.33: `HMCalendarEvent` の利用

```
let comp = DateComponents(hour: 8, minute: 30)
let event = HMCalendarEvent(fire: comp)
let trigger = HMEventTrigger(name: "午前 8:30", events: [event], predicate: nil)
```

#### `HMSignificantTimeEvent`

日の出、日の入りを発火条件とするための `HMSignificantTimeEvent` も加わりました。例えば、日の入り 30 分前に発火するイベントトリガを作成するにはリスト 12.34 のようにします。

リスト 12.34: `HMSignificantTimeEvent` の利用

<sup>\*9</sup> これまで `HMTimerTrigger` を利用することで可能でしたが `HMCalendarEvent` には `HMTimerTrigger` がない付加機能がいくつかあります

```
let offset = DateComponents(minute: -30)
let event = HMSignificantTimeEvent(significantEvent: .sunset, offset: offset)
let trigger = HMEventTrigger(name: "日の入り 30 分前", events: [event], predicate: nil)
```

HMSignificantTimeEvent のイニシャライザの significantEvent に指定する HMSignificantEvent には表 12.12 の 2 種があります。

表 12.12: HMSignificantEvent

HMSignificantEvent	説明
sunrise	日の出
sunset	日の入り

#### HMCharacteristicThresholdRangeEvent

特定キャラクタ (HMCharacteristic) の value プロパティの数値の範囲を発火条件とする HMCharacteristicThresholdRangeEvent も加わりました。例えば、リビングの現在の温度が 29 度以上になった時に発火するイベントトリガを作成するにはリスト 12.35 のようにします。

リスト 12.35: HMCharacteristicThresholdRangeEvent の利用

```
let temperature = //< 現在の温度の characteristic

let range = HMNumberRange(minValue: 29.0)
let event = HMCharacteristicThresholdRangeEvent(
 characteristic: temperature,
 thresholdRange: range
)
let trigger = HMEventTrigger(name: "気温 29 度以上", events: [event], predicate: nil)
```

数値の範囲を指定する HMNumberRange には表 12.13 に示す 3 種の使いかたがあります。

表 12.13: HMNumberRange の例

例	説明
HMNumberRange(minValue: 29)	29 以上
HMNumberRange(maxValue: 19)	19 以下
HMNumberRange(minValue: 20, maxValue: 28)	20 以上 28 以下

### HMPresenceEvent

`HMPresenceEvent` が追加され、ホームに誰もいなくなった時や、誰か一人が帰ってきた時など、ユーザの存在状態を発火条件とすることもできるようになりました。例えば、ホームに誰もいなくなった時に発火するイベントトリガを作成するにはリスト 12.36 のようにします。

リスト 12.36: `HMPresenceEvent` の利用

```
let event = HMPresenceEvent(presenceEventType: .lastExit, presenceUserType: .homeUsers)
let trigger = HMEventTrigger(name: "全員が出発", events: [event], predicate: nil)
```

`HMPresenceEvent` のイニシャライザの `presenceEventType` に指定できる値には表 12.14 の 6 種があります。

表 12.14: `HMPresenceEventType` 一覧

<code>HMPresenceEventType</code>	説明
<code>everyEntry</code>	対象のユーザいずれかが到着
<code>everyExit</code>	対象のユーザいずれかが出発
<code>firstEntry</code>	最初のユーザが到着
<code>lastExit</code>	最後のユーザが出発（全員が出発）
<code>atHome</code>	<code>firstEntry</code> と同じ
<code>notAtHome</code>	<code>lastExit</code> と同じ

`firstEntry` と `atHome`、`lastExit` と `notAtHome` は、それぞれどちらを指定しても全く同じ挙動になりますが、文脈によって使い分けるのが良さそうです。例えば「家に誰もいない時」という文脈なら `notAtHome` を利用するのが直感的でしょう。

また、`presenceUserType` に指定できる値には表 12.15 の 3 種があります。

表 12.15: `HMPresenceEventUserType` 一覧

<code>HMPresenceEventUserType</code>	説明
<code>currentUser</code>	自分
<code>homeUsers</code>	ホームを利用するユーザ全員
<code>customUsers</code>	指定したユーザ <sup>*10</sup>

### HMDurationEvent

`HMDurationEvent` はイベントトリガに直接設定できない特殊なイベントです。`HMDurationEvent` については「`endEvents` プロパティ」で紹介します。

### 12.3.2 HMEventTrigger のアップデート

HMEventTrigger 自体にも、多くの機能追加と変更があります。

#### recurrences プロパティ

HMEventTrigger の `recurrences` プロパティを利用することで、イベントトリガを特定の曜日にだけ発火させることができます。例えば、平日（月曜～金曜）の日の出に発火するイベントトリガを作成するにはリスト 12.37 のようにします。

リスト 12.37: `recurrences` の利用

```
let event = HMSignificantTimeEvent(significantEvent: .sunrise, offset: nil)

// 月曜日から金曜日までを示す5つのDateComponentsを作成
let weekday = Array(2...6).map { DateComponents(weekday: $0) }

// recurrences を指定して HMEventTrigger を作成
let trigger = HMEventTrigger(
 name: "Weekday",
 events: [event],
 end: nil,
 recurrences: weekday,
 predicate: nil
)
```

`recurrences` プロパティには `DateComponents` の配列を指定しますが、iOS 11 時点で対応している `DateComponents` のプロパティは曜日 (`weekday`) のみです。

#### executeOnce プロパティ

HMEventTrigger に追加された `executeOnce` プロパティにより、そのイベントトリガが一度のみ発火されるものなのか繰り返し発火されるものなのかを、明示できるようになりました。具体的には `executeOnce` プロパティが `true` の場合、そのイベントトリガが発火した際、`isEnabled` プロパティが自動的に `false` に変更されます。`executeOnce` を更新するコードをリスト 12.38 に示します。

リスト 12.38: `executeOnce` を更新

```
trigger.updateExecuteOnce(true) { error in }
```

#### endEvents プロパティ

HMEventTrigger が発火してシーンが実行された後に、特定の条件によって発火前の状態に戻すための `endEvents` プロパティも追加されました。具体的には、HMEventTrigger の発火により電

球を点灯させた 5 分後に、自動的に電球の状態を元に戻す（つまり消灯する）といったことができます（リスト 12.39）。

リスト 12.39: endEvents を設定

```
let trigger = //< 電球を点灯させる HMEventTrigger
// 5 分後
let endEvent = HMDurationEvent(duration: 300)

// endEvents を設定
trigger.updateEndEvents([endEvent]) { error in
}
```

HMDurationEvent は、iOS 11 で追加された endEvents プロパティ専用のイベントです。endEvents プロパティには、HMDurationEvent の他、HMCharacteristicEvent、HMCharacteristicThresholdRangeEvents を設定できます。

#### predicate 用メソッドのアップデート

predicate プロパティに指定する NSPredicate を作るためのメソッドもいくつか追加されました。

これまであった String 指定で日の出・日の入りを条件として指定するメソッドが Deprecated (非推奨) となり、代わりに HMSignificantTimeEvent で指定できるメソッドが追加されました。

- predicateForEvaluatingTriggerOccurring(afterSignificantEvent:)
- predicateForEvaluatingTriggerOccurring(beforeSignificantEvent:)
- predicate(forEvaluatingTriggerOccurringBetweenSignificantEvent:, secondSignificantEvent:)

例えば、日中のみ（日の出から日の入りまで）という条件で predicate プロパティを指定するにはリスト 12.40 のようにします。

リスト 12.40: HMSignificantTimeEvent 指定で predicate

```
let sunrise = HMSignificantTimeEvent(significantEvent: .sunrise, offset: nil)
let sunset = HMSignificantTimeEvent(significantEvent: .sunset, offset: nil)

// 日の出から日の入りまでという条件
let predicate = HMEventTrigger.predicate(
 forEvaluatingTriggerOccurringBetweenSignificantEvent: sunrise,
 secondSignificantEvent: sunset
)

// 作成した predicate を指定してトリガ構築
let trigger = HMEventTrigger(
 name: "日の出から日の入り",
 events: [event],
 predicate: predicate
)
```

HMPresenceEvent を指定するメソッドも追加されました。

- `predicateForEvaluatingTrigger(withPresence:)`

これにより、例えば、誰も家にいないときだけ日の入りとともに自動でライトを点ける、といった条件指定ができます（リスト 12.41）。

リスト 12.41: HMPresenceEvent 指定で predicate

```
// 誰も家にいないときだけという条件
let presence = HMPresenceEvent(
 presenceEventType: .notAtHome,
 presenceUserType: .homeUsers
)
let predicate = HMEventTrigger.predicateForEvaluatingTrigger(withPresence: presence)

// 作成した predicate を指定してトリガ構築
let trigger = HMEventTrigger(name: "不在", events: [event], predicate: predicate)
```

最後にもう 1 つ、時刻を条件として指定するメソッドに、時刻の範囲を一度に指定できるものが追加されました。

- `predicateForEvaluatingTriggerOccurringBetweenDate(with:, secondDateWith:)`

例えば、10:00 から 18:30 という条件で `predicate` プロパティを指定するにはリスト 12.42 のようになります。

リスト 12.42: 開始時刻と終了時刻指定で predicate

```
// 10:00 から 18:30 という条件
let start = DateComponents(hour: 10, minute: 0)
let end = DateComponents(hour: 18, minute: 30)
let predicate = HMEventTrigger.predicateForEvaluatingTriggerOccurringBetweenDate(
 with: start,
 secondDateWith: end
)

// 作成した predicate を指定してトリガ構築
let trigger = HMEventTrigger(name: "日中", events: [event], predicate: predicate)
```

#### triggerActivationState プロパティ

`HMEventTrigger` に新しく追加された `triggerActivationState` プロパティにより、`HMEventTrigger` の利用可否状態を参照できるようになりました。iOS 10 までは `isEnabled` プロパティを参照し、単純にトリガ自体が有効か無効かを知ることしかできませんでした。しかし

`triggerActivationState` プロパティでは表 12.16 のように、利用できない理由を含め知ることができます。

表 12.16: HMEventTriggerActivationState 一覧

値	説明
<code>disabled</code>	トリガ自体が無効になっている
<code>disabledNoHomeHub</code>	ホームハブがないため利用不可
<code>disabledNoCompatibleHomeHub</code>	互換性のあるホームハブがないため利用不可
<code>disabledNoLocationServicesAuthorization</code>	位置情報の利用が許可されていないため利用不可
<code>enabled</code>	利用可能

### 12.3.3 HMHome のアップデート

`HMHome` には `homeHubState` プロパティが追加されました。これによりホームハブの状態を参照できるようになりました。`homeHubState` プロパティに入る `HMHomeHubState` の値の一覧を表 12.17 にまとめます。

表 12.17: HMHomeHubState

値	説明
<code>notAvailable</code>	ホームハブが設定されていない
<code>connected</code>	ホームハブに接続されている
<code>disconnected</code>	ホームハブに接続されていない

### 12.3.4 HMAccessory と HMCharacteristic のアップデート

`characteristicType` として `HMCharacteristicTypeColorTemperature` が追加されました（表 12.18）。

表 12.18: HMCharacteristicTypeColorTemperature

HMCharacteristicType～	説明	フォーマット
<code>ColorTemperature</code>	色温度	int

一方で表 12.19 の 4 つが Deprecated になっています。

これらの `characteristicType` が Deprecated になったのは `HMAccessory` に

- `manufacturer` プロパティ
- `model` プロパティ
- `profiles` プロパティ

表 12.19: Deprecated になった characteristicType 一覧

HMCharacteristicType～	説明	フォーマット
Manufacturer	製造元	string
Model	モデル	string
SerialNumber	シリアル番号	string
FirmwareVersion	ファームウェアのバージョン	string

- `firmwareVersion` プロパティ

が追加されたためです。iOS 11 以降はアクセサリ > サービス > キャラクタと辿らなくとも `HMAccessory` のインスタンスから直接これらの情報を参照できるようになりました。

### 12.3.5 HMHomeDelegate のアップデート

`HMHomeDelegate` には以下 2 つのメソッドが追加されました。

- `home(_ home: HMHome, didUpdate homeHubState: HMHomeHubState)`
  - `homeHubState` の更新を監視
- `homeDidUpdateAccessControl(forCurrentUser home: HMHome)`
  - `currentUser` の権限の更新を監視

### 12.3.6 HMAccessoryDelegate のアップデート

`HMAccessoryDelegate` には以下 3 つのメソッドが追加されました。

- `accessory(_ accessory: HMAccessory, didAdd profile: HMAccessoryProfile)`
  - アクセサリに `profile` が追加されたことを監視
- `accessory(_ accessory: HMAccessory, didRemove profile: HMAccessoryProfile)`
  - アクセサリから `profile` が削除されたことを監視
- `accessory(_ accessory: HMAccessory, didUpdateFirmwareVersion firmwareVersion: String)`
  - アクセサリの `firmwareVersion` の更新を監視

## 12.4 HomeKit 実践

### 12.4.1 HomeKit Accessory Simulator の利用

HomeKit を利用したアプリを開発するには、まず Xcode プロジェクトの Capabilities で HomeKit の項目を ON にする必要があります<sup>\*11</sup>。このとき「Download HomeKit Simulator...」という

---

<sup>\*11</sup> もう 1 点、アプリの Info.plist に `NSHomeKitUsageDescription` を設定しておく必要があります

ボタンが表示されますので、ここから Homekit Accessory Simulator を取得可能です（図 12.13）。

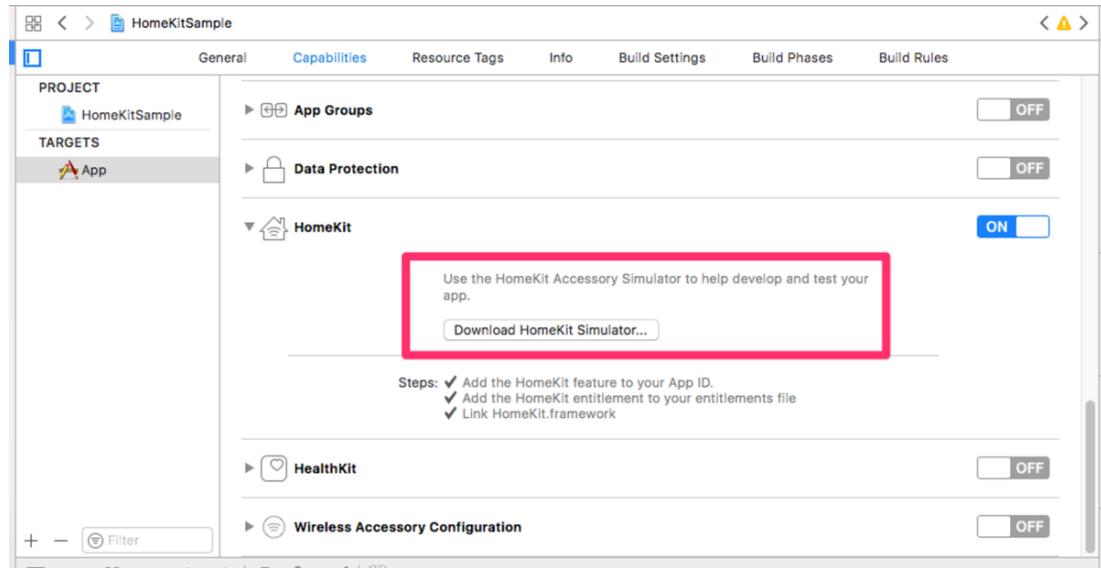


図 12.13: Additional Tools for Xcode のダウンロード

正確には、この導線から Additional Tools for Xcode をダウンロードすると、その中のツールの1つに HomeKit Accessory Simulator があります（図 12.14）。

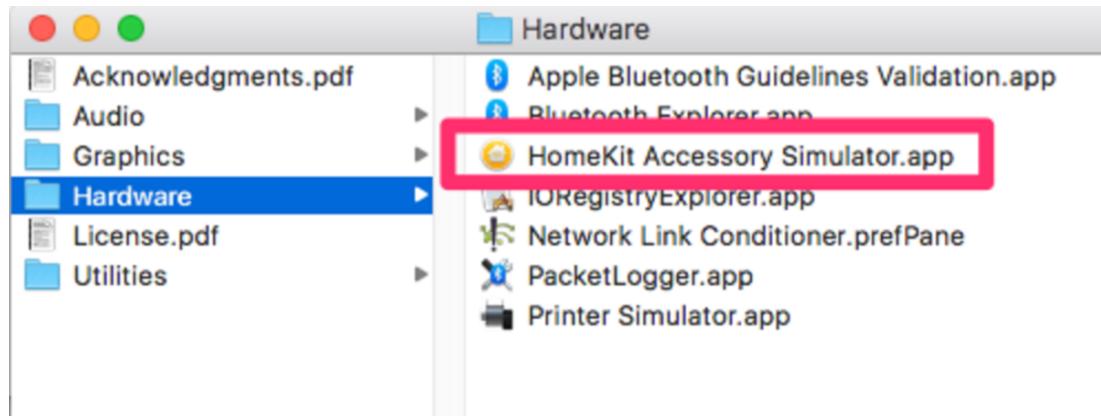


図 12.14: HomeKit Accessory Simulator の場所

HomeKit Accessory Simulator を利用すると、任意のアクセサリ（HMAccessory）、サービス（HMServices）、キャラクタ（HMCharacteristic）を追加可能で、まだ日本では手に入らないアクセサリの挙動もシミュレート可能です（図 12.15）。ここで作成したアクセサリは、シミュレータ上だけでなく実機のホーム（HMHome）に追加することができ、Apple 純正のホームアプリでも、自作の HomeKit 対応アプリでも利用することができます。例えば、彩度（Saturation）などの数値もこのツールの GUI で変更でき、実際の HomeKit 対応デバイスを利用するよりもデバッグやテストの効率が上がるケースがあります。

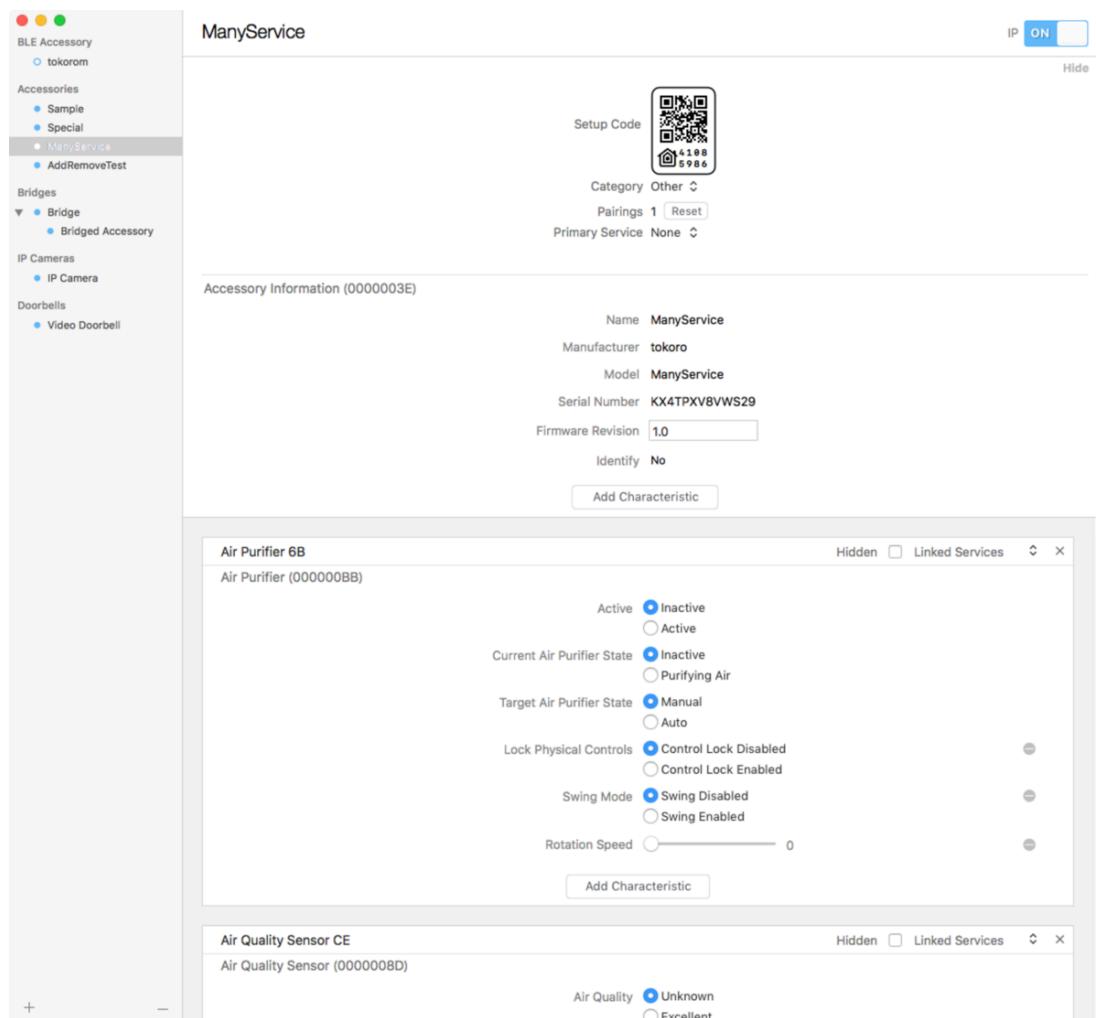


図 12.15: HomeKit Accessory Simulator

#### 12.4.2 HomeKit 対応製品利用実例

「12.4.1 HomeKit Accessory Simulator の利用」ではシミュレータの有用性について紹介しましたが、やはり実際に HomeKit に対応したデバイスを利用しての開発は魅力的です。この節では、日本で購入できる HomeKit 対応製品のうち 7 つを実際に購入し、取得できた情報をまとめます。

## Philips Hue



図 12.16: Philips Hue

表 12.20: 主なサービス

HMSERVICEType～	説明
Lightbulb	電球

表 12.21: 主なキャラクタ

HMCharacteristicType～	説明	フォーマット	書き込み
PowerState	電源の状態	bool	可
Hue	色相	float	可
Saturation	彩度	float	可
Brightness	明るさ	int	可

スマート IoT 照明のパイオニアです。PowerState キャラクタで点灯/消灯を、Hue、Saturation、Brightness キャラクタで電球の色相、彩度、明るさをそれぞれ変更できます（リスト 12.43）。

リスト 12.43: Hue の電球の色を変える

```
// Hue、Saturation、Brightness に対応したキャラクタを取得
let hue = //< HMCharacteristicTypeHue
let saturation = //< HMCharacteristicTypeSaturation
let brightness = //< HMCharacteristicTypeBrightness
```

```
// 電球をピンク色に更新
hue.writeValue(355) { error in
}
saturation.writeValue(38) { error in
}
brightness.writeValue(91) { error in
}
```

キャラクタの `value` の更新が視覚的にわかりやすく、また、`true/false` だけでなく数値の `value` を書き込み可能な製品はまだ珍しいです。そのため、HomeKit プログラミングにトライするうえで最初に購入するものとして、おすすめしやすい製品の1つです。

また、今回紹介する製品の中では唯一、ブリッジの役割を担うアクセサリを含み、ブリッジにライトが連なるというアクセサリの多段構成も確認できます。

Elgato Eve Wireless Room Sensor



図 12.17: Elgato Eve Wireless Room Sensor

表 12.22: 主なサービス

HMServicesType~	説明
TemperatureSensor	温度センサー
HumiditySensor	湿度センサー
AirQualitySensor	清浄度センサー
Battery	バッテリー

表12.23: 主なキャラクタ

HMCharacteristicType~	説明	フォーマット	書き込み
CurrentTemperature	現在の温度	float	-
CurrentRelativeHumidity	現在の相対湿度	float	-
AirQuality	空気質	uint8	-
BatteryLevel	電池残量	uint8	-
ChargingState	充電の状態	uint8	-
StatusLowBattery	状況（電池残量低下）	uint8	-

部屋に設置するだけで部屋の現在の温度(CurrentTemperature)、相対湿度(CurrentRelativeHumidity)、空気質(AirQuality)が参照できます。空気質(AirQuality)キャラクタが確認できる製品は、今回紹介する製品の中ではこれだけです\*12。部屋の温度をトリガとしてエアコンを操作するなどの利用方法が考えられます。

空気質(AirQuality)キャラクタのvalueのフォーマットはuint8ですが、この数値の意味はHMCharacteristicValueAirQualityというenumとして定義されています（表12.24）。

値 | rawValue | 説明 | --- | ---- | unknown | 0 | 未定義 | excellent | 1 | とても良い | good | 2 | 良い | fair | 3 | 普通 | inferior | 4 | 良くない | poor | 5 | とても良くない |

動作には単3電池3本が必要で、電池の残量などもBatteryLevel、ChargingState、StatusLowBatteryキャラクタで把握可能です。

\*12 空気質の参照が不要なら、もう少しだけ低価格な「Elgato Eve Degree Indoor Sensor」も選択肢にあがります

## Elgato Eve Weather Wireless Outdoor Sensor



図 12.18: Elgato Eve Weather Wireless Outdoor Sensor

table[eveWeatherService] 主なサービス

表 12.24: HMCharacteristicValueAirQuality 値一覧

HMSERVICEType~	説明
TemperatureSensor	温度センサー
HumiditySensor	湿度センサー
Battery	バッテリー

「Elgato Eve Wireless Room Sensor」の屋外用です。こちらで取得できるのは現在の温度

表12.25: 主なキャラクタ

HMCharacteristicType~	説明	フォーマット	書き込み
CurrentTemperature	現在の温度	float	-
CurrentRelativeHumidity	現在の相対湿度	float	-
BatteryLevel	電池残量	uint8	-
ChargingState	充電の状態	uint8	-
StatusLowBattery	状況（電池残量低下）	uint8	-

(CurrentTemperature)、相対湿度 (CurrentRelativeHumidity) です。

この製品自体は気圧の計測もサポートしていますが、iOS 11 時点の HomeKit は気圧のキャラクタをサポートしていません。ただ、HomeKit は未定義のサービス (HMService) やキャラクタ (HMCharacteristic) を許容していますので、気圧の情報も未定義のサービス、キャラクタから無理やり参照することはできます（リスト 12.44）。

リスト 12.44: 気圧を取得

```
// Eve Weather の気圧を示す serviceType
let serviceType = "E863F00A-079E-48FF-8F27-9C2605A29F52"
// Eve Weather の気圧を示す characteristicType
let characteristicType = "E863F10F-079E-48FF-8F27-9C2605A29F52"

// 上記 serviceType/characteristicType と合致するキャラクタを取得
let service = home.servicesWithTypes([serviceType])?.first
let candidates = service?.characteristics
 .filter { $0.characteristicType == characteristicType }

guard let airPressure = candidates?.first else {
 return
}

airPressure.readValue { error in
 guard let value = airPressure.value as? Float else {
 return
 }
 // 取得した気圧を print
 print("airPressure: \(value)")
}
```

動作には単3電池2本が必要で、電池残量関連のキャラクタも参照可能です。



図 12.19: Eve Weather を外壁に設置

屋外用ということで外壁等に設置することになります（図 12.19）。両面テープなど付属していませんので別途購入が必要です。防水等級は IPX3（防雨形）です。完全防水ではなく生活防水レベル

とのことですので、強い雨の日にも故障しないかはわかりません（設置してから3ヶ月の範囲では雨の日でも問題なく利用できています）。

#### Elgato Eve Door & Window Wireless Contact Sensor



図 12.20: Elgato Eve Door & Window Wireless Contact Sensor

表 12.26: 主なサービス

HMServiceType～	説明
ContactSensor	接触センサー
Battery	バッテリー

表 12.27: 主なキャラクタ

HMCharacteristicType～	説明	フォーマット	書き込み
ContactState	接触センサーの状態	uint8	-
BatteryLevel	電池残量	uint8	-
ChargingState	充電の状態	uint8	-
StatusLowBattery	状況（電池残量低下）	uint8	-

ドアや窓の開閉状態を感知することができる接触センサーです。この製品は2つの部品に分かれしており、この2つが近接しているかどうかを判定できます。

接触センサーの状態 (ContactState) キャラクタの value のフォーマットは uint8 ですが、この数値の意味は HMCharacteristicValueContactState という enum として定義されています（表 12.28）。

表12.28: HMCharacteristicValueContactState 値一覧

値	rawValue	説明
detected	0	接触
none	1	非接触

動作には単3電池1本が必要で、電池残量関連のキャラクタも参照可能です。



図 12.21: Eve Door & Window を開き戸に設置

設置用の両面テープも付属しています。図 12.21 は自宅の玄関の開き戸にこの製品を設置した例ですが、設置場所さえ確保できれば引き戸にも設置できます。

Elgato Eve Motion Wireless Motion Sensor



図 12.22: Elgato Eve Motion Wireless Motion Sensor

表 12.29: 主なサービス

HMSERVICEType～	説明
MotionSensor	モーションセンサー
Battery	バッテリー

モーションセンサーです。動きを検知 (`MotionDetected`) している状態かどうかを `true/false` で取得できます。Apple 純正のホームアプリには「センサーが何かを検知したとき」をトリガとしたオートメーションを作成する機能がありますが、この `MotionDetected` キャラクタはその用途に使えます（図 12.23）。

表 12.30: 主なキャラクタ

HMCharacteristicType~	説明	フォーマット	書き込み
MotionDetected	動きを検知	bool	-
BatteryLevel	電池残量	uint8	-
ChargingState	充電の状態	uint8	-
StatusLowBattery	状況（電池残量低下）	uint8	-



図 12.23: ホームアプリの新規オートメーション画面

動作には単3電池2本が必要で、電池残量関連のキャラクタも参照可能です。

なお、センサーの感度なども変更することができ、変更したい場合はApp Storeから「Elgato Eve」をダウンロードして設定します（図12.24）。



図 12.24: Elgato Eve の設定画面

D-Link Omna 180 Cam HD カメラ



図 12.25: D-Link Omna 180 Cam HD カメラ

表 12.31: 対応するカメラコントロール

プロパティ名	型	説明
streamControl	HMCameraStreamControl	ビデオストリーム（動画）
snapshotControl	HMCameraSnapshotControl	スナップショット（静止画）
microphoneControl	HMCameraAudioControl	マイク
speakerControl	HMCameraAudioControl	スピーカー
settingsControl	HMCameraSettingsControl	カメラ設定

表 12.32: 主なサービス

HMSERVICEType～	説明
CameraControl	カメラコントロール
Microphone	マイク
Speaker	スピーカー
MotionSensor	モーションセンサー

表 12.33: 主なキャラクタ

HMCharacteristicType～	説明	フォーマット	書き込み
ImageMirroring	画像反転	bool	可
Mute	消音	bool	可
Volume	音量	int	可
MotionDetected	動きを検知	bool	-

2017 年 9 月時点で日本で購入可能な唯一の HomeKit 対応 IP カメラです。

ビデオストリーム（動画）の利用、スナップショット（静止画）の撮影、マイクやスピーカーの利用と、HomeKit がサポートするカメラコントロールとしては一通りのことができます。また、カメラを利用したモーション検知（MotionDetected）も可能です。

App Store から「OMNA」アプリをダウンロードして利用することで、HomeKit がサポートしていない部分の設定変更もできます。

カメラで撮影している動画をアプリで表示する具体的な方法、マイクやスピーカーの設定方法などについては「12.2.8 カメラの利用」で紹介します。

Koogeek Smart Plug P1



図 12.26: Koogeek Smart Plug P1

表 12.34: 主なサービス

HMServiceType～	説明
Outlet	コンセント

いわゆるスマートコンセントです。このデバイスを経由してコンセントとファン（扇風機）などの機器を接続し、接続した機器の電源の On/Off を更新することができます。このデバイスのプラグは3ピンのため、日本で一般的な2ピン用のコンセントで利用する場合は、別途3ピン→2ピン変換プラグが必要です（図 12.27）。

表 12.35: 主なキャラクタ

HMCharacteristicType～	説明	フォーマット	書き込み
PowerState	電源の状態	bool	可
OutletInUse	コンセント使用中	bool	-



図 12.27: プラグの形状

具体的なコントロール方法ですが、このアクセサリが持つ PowerState キャラクタに対して `writeValue(true)` で電源 On、`writeValue(false)` で電源 Off ができます。また、OutletInUse キャラクタにより、コンセントに機器が接続されているかを `true/false` で参照することもできます。

なお、Koogeek Smart Plug P1 は、今回紹介する製品の中では `associatedServiceType` プロパティを更新できる唯一のデバイスです。

`associatedServiceType` プロパティを更新するには、Apple 純正のホームアプリ<sup>\*13</sup>でこのデバイスの詳細画面を開き、「タイプ」を表 12.36 の 2 種<sup>\*14</sup>の中からコンセントに接続した機器に応じて

<sup>\*13</sup> その他、App Store で公開されている Koogeek Home アプリなどからも変更が可能です

<sup>\*14</sup> 選択項目としてはもう1つ「コンセント」がありますが、これを選択すると `associatedServiceType` に `nil` が設定されます

最適なものを選択します。

表12.36: 選択できる associatedServiceType

HMServicetType～	説明
Fan	ファン
Lightbulb	電球

例えば、ここで「ファン」を選択すると、ホームアプリ上でこのサービスを示すアイコンもファンになり、電源をオンにすると勢いよくファンが回るようなギミックも見られます（図12.28）。

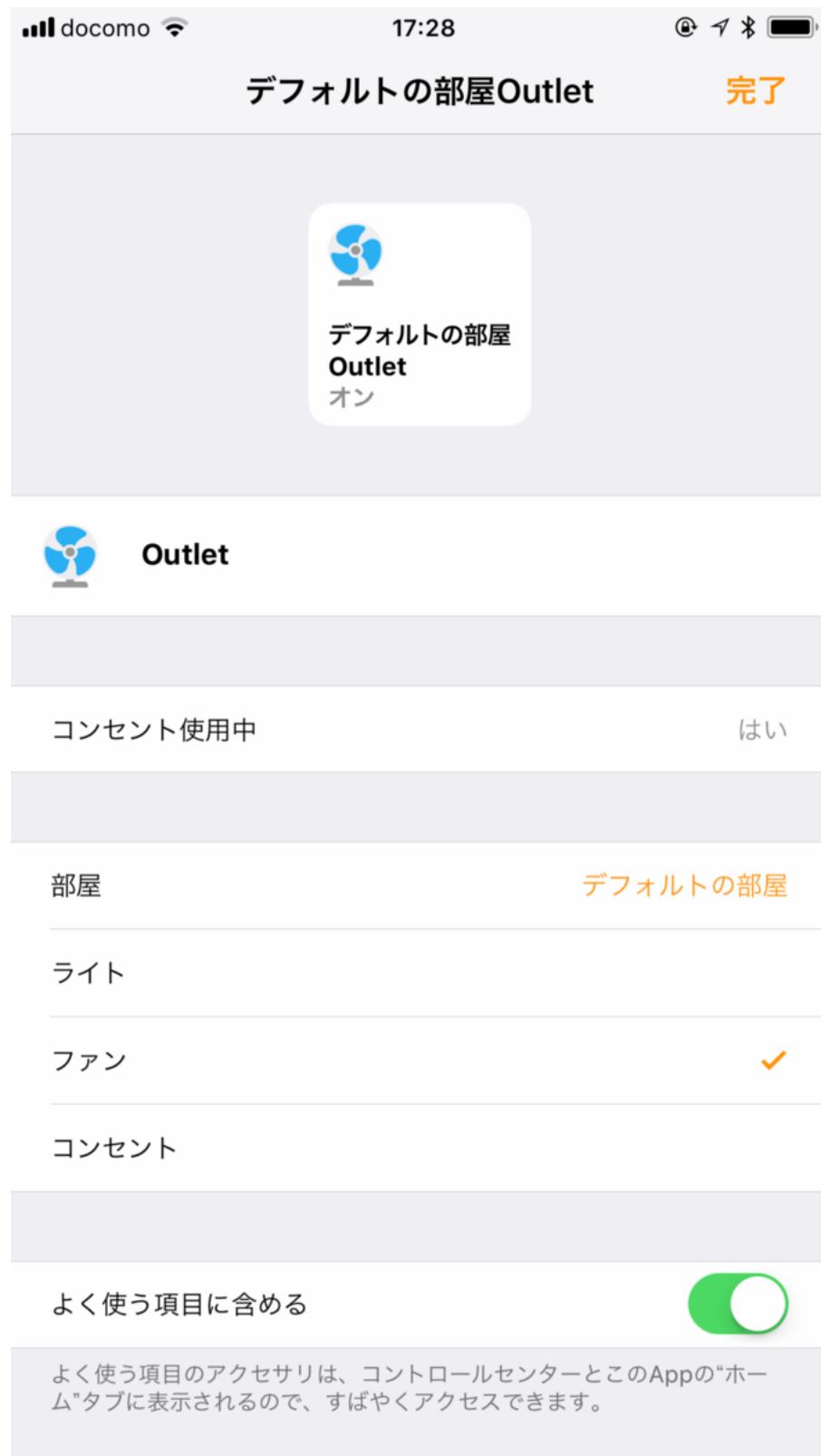


図 12.28: associatedServiceType の変更

## 12.5 まとめ

「12.2 HomeKit 入門」で紹介したように、HomeKitには1つのアクセサリの特性を細かく操作する機能から、ホーム、ルーム、ゾーンなどアクセサリをグルーピングするための構成を管理する機能まで、自宅のオートメーションのための機能を広くサポートしています。

また「12.3 iOS 11でのアップデートまとめ」で解説したように、iOS 11でも多くの機能が追加されました。特にはイベントトリガ(HMEventTrigger)に関連する追加のボリュームが大きく、より細やかで実用的なオートメーションを実現できるように、というAppleの意気込みが感じられます。

私は「12.4 HomeKit 実践」のHomeKit対応製品利用実例をまとめるために、実際に7つの対応製品を購入して利用しましたが、自宅の様々な情報が手元のiPhoneに集約されることに感動しました。今現在はまだ、日本で購入できるHomeKit対応製品の数が少ないこともあり、HomeKitが普及しているとは言えません。しかし、対応製品が潤沢になり、HomeKitがバージョンアップし続けることにより、遠くない将来、多くのiOSプログラマが自宅をカスタマイズするためにHomeKitを利用したコードを書くようになるでしょう。

ぜひHomeKitを活用し、自分の書いたコードでマイホームやオフィスが進化する感動を味わってみてください。

## 12.6 HomeKit お役立ちリファレンス

12.6.1 HMAccessoryCategory.categoryType一覧

12.6.2 HMService.serviceType一覧

12.6.3 HMCharacteristic.characteristicType一覧

12.6.4 HMCharacteristicMetadata.format一覧

12.6.5 HMCharacteristicMetadata.units一覧

表 12.37: categoryType 一覧

HMAccessoryCategoryType～	説明
Other	その他
SecuritySystem	セキュリティシステム
Bridge	ブリッジ
Door	ドア
DoorLock	ドアロック
Fan	ファン
GarageDoorOpener	ガレージドアオープナー
IPCamera	IP カメラ
Lightbulb	電球
Outlet	コンセント
ProgrammableSwitch	プログラマブルスイッチ
RangeExtender	中継器
Sensor	センサー
Switch	スイッチ
Thermostat	サーモスタット
VideoDoorbell	ビデオドアベル
Window	窓
WindowCovering	ブラインド
AirPurifier	空気清浄機
AirHeater	ヒーター
AirConditioner	エアコン
AirHumidifier	加湿器
AirDehumidifier	除湿機

表 12.38: serviceType 一覧

HMServiceType～	説明
AccessoryInformation	アクセサリ情報サービス
AirPurifier	空気清浄機
AirQualitySensor	清浄度センサー
Battery	バッテリー
CarbonDioxideSensor	二酸化炭素センサー
CarbonMonoxideSensor	一酸化炭素センサー
ContactSensor	接触センサー
Door	ドア
Doorbell	ドアベル
Fan	ファン
VentilationFan	ファン
FilterMaintenance	フィルタ点検
GarageDoorOpener	ガレージドアオープナー
HeaterCooler	冷暖房機
HumidifierDehumidifier	加湿機・除湿機
HumiditySensor	湿度センサー
LeakSensor	漏れセンサー
LightSensor	光センサー
Lightbulb	電球
LockManagement	施錠管理
LockMechanism	施錠機構
Microphone	マイク
MotionSensor	モーションセンサー
OccupancySensor	人感センサー
Outlet	コンセント
SecuritySystem	セキュリティシステム
Label	ラベル
Slats	スラット（羽板）
SmokeSensor	煙センサー
Speaker	スピーカー
StatelessProgrammableSwitch	ステートレス・プログラマブル・スイッチ
StatefulProgrammableSwitch	ステートフル・プログラマブル・スイッチ
Switch	スイッチ
TemperatureSensor	温度センサー
Thermostat	サーモスタット
Window	窓
WindowCovering	ブラインド
CameraControl	カメラコントロール
CameraRTPStreamManagement	RTPストリームマネジメント

表 12.39: characteristicType 一覧

HMCharacteristicType～	説明	フォーマット	単位
Active	動作中	uint8	
AdminOnlyAccess	管理者のみアクセス可能	bool	
AirParticulateDensity	空気中の微粒子の濃度	float	micrograms/m^3
AirParticulateSize	空気中の微粒子のサイズ	uint8	
AirQuality	空気質	uint8	
AudioFeedback	オーディオフィードバック	bool	
BatteryLevel	電池残量	uint8	percentage
Brightness	明るさ	int	percentage
CarbonDioxideDetected	二酸化炭素を検知	uint8	
CarbonDioxideLevel	二酸化炭素レベル	float	ppm
CarbonDioxidePeakLevel	二酸化炭素ピークレベル	float	ppm
CarbonMonoxideDetected	一酸化炭素を検知	uint8	
CarbonMonoxideLevel	一酸化炭素レベル	float	ppm
CarbonMonoxidePeakLevel	一酸化炭素ピークレベル	float	ppm
ChargingState	充電の状態	uint8	
ColorTemperature	色温度	int	
ContactState	接触センサーの状態	uint8	
CoolingThreshold	冷房のしきい値温度	float	celsius
CurrentAirPurifierState	現在の空気清浄機の状態	uint8	
CurrentDoorState	現在のドアの状態	uint8	
CurrentFanState	現在のファンの状態	uint8	
CurrentHeaterCoolerState	現在の冷暖房機の状態	uint8	
CurrentHeatingCooling	現在の冷暖房の状態	uint8	
CurrentHorizontalTilt	現在の横方向の傾斜角度	int	arcdegrees
CurrentHumidifierDehumidifierState	現在の加湿機・除湿機の状態	uint8	
CurrentLightLevel	現在の光量	float	lux
CurrentLockMechanismState	施錠機構の現在の状態	uint8	
CurrentPosition	現在の位置	uint8	percentage
CurrentRelativeHumidity	現在の相対湿度	float	percentage
CurrentSecuritySystemState	現在のセキュリティシステムの状態	uint8	
CurrentSlatState	現在のスラット（羽板）の状態	uint8	
CurrentTemperature	現在の温度	float	celsius
CurrentTilt	現在の傾斜角度	int	arcdegrees
CurrentVerticalTilt	現在の縦方向の傾斜角度	int	arcdegrees
DehumidifierThreshold	除湿機のしきい値	float	percentage
FilterChangeIndication	フィルタ交換表示	uint8	
FilterLifeLevel	フィルタ寿命レベル	float	
FilterResetChangeIndication	フィルタ交換表示をリセット	uint8	
HeatingThreshold	暖房のしきい値温度	float	celsius
HoldPosition	固定位置	bool	
Hue	色相	float	arcdegrees
HumidifierThreshold	加湿機のしきい値	float	percentage
InputEvent	プログラマブル・スイッチ・イベント	uint8	
LabelIndex	ラベルの索引	uint8	
LabelNamespace	ラベルの名前空間	uint8	
LeakDetected	漏れを検知	uint8	
LockManagementAutoSecureTimeout	施錠管理の自動セキュリティタイムアウト	uint32	seconds
LockManagementControlPoint	施錠管理のコントロールポイント	tlv8 <sup>*15</sup>	
LockMechanismLastKnownAction	施錠機構の最後の既知の動作	uint8	
LockPhysicalControls	コントロールボタンをロック	uint8	
Logs	ログ	tlv8 <sup>*16</sup>	
MotionDetected	動きを検知	bool	
Name	名前	string	

表 12.40: format 一覧

HMCharacteristicMetadataFormat~	値
Bool	bool
Int	int
Float	float
String	string
Array	array
Dictionary	dict
UInt8	uint8
UInt16	uint16
UInt32	uint32
UInt64	uint64
Data	data
TLV8	tlv8

表 12.41: units 一覧

HMCharacteristicMetadataUnits~	値
Celsius	celsius
Fahrenheit	fahrenheit
Percentage	percentage
ArcDegree	arcdegrees
Seconds	seconds
Lux	lux
PartsPerMillion	ppm
MicrogramsPerCubicMeter	micrograms/m^3

## 第 13 章

# Metal

iOS 11 で Metal は「Metal 2」となり、多くの新機能・改善が加えられました。また、新たに登場した ARKit が Metal でのカスタムレンダリングをサポートしていたり、Core Image のカスタムカーネルを Metal のシェーダで書けるようになる等、他フレームワークとの連携も強化されました。

ただ、これまで Metal は、ゲームアプリの派手な 3D グラフィックスや、写真加工アプリの特殊なエフェクト等に用いられることが多く、まだまだほとんどの iOS 開発者にとって馴染みのある API だとは言い難いのではないのでしょうか。

本章では、まず「そもそも Metal とは何か」というところから始めて、その基本概念や実装方法を解説し、その上で iOS 11 で加わった新機能や強化された機能について解説します。

本章のサンプルプロジェクトは [iOS11\\_samplecode リポジトリ](#) の chapter\_13 フォルダにあります。なお、Metal はシミュレータではサポートされていないため、動作確認の際には必ず実機を用いてください。

### 13.1 Metal の概要

#### 13.1.1 Metal とは

Metal は、Apple によって提供されているローレベルなグラフィックス API です。別の言い方をすると、GPU へのアクセスを提供する API です。GPU というハードウェア層を直接たたく API (=ローレベル) であり、GPU は"Graphics Processing Unit"の略であるとおり、そもそもは描画処理を担う演算装置なので、そのインターフェースである Metal は「ローレベルなグラフィックス API」というわけです。

近年では GPU の並列演算処理能力に着目し、描画以外の汎用的な計算に用いられるケースも増えてきています。たとえばディープラーニングは非常に計算量が多いアルゴリズムですが、その大部分を占める行列演算は並列化しやすく GPU での処理に向いています。この用途では「汎用」の意味から GPGPU (General Purpose GPU) という呼び方をすることもあります。

#### 13.1.2 OpenGL ES と Metal

Metal 登場以前から、iOS ではローレベルなグラフィックス API として OpenGL ES が利用できました。ただ OpenGL ES は Apple が独自に策定しているものではなく、様々なプラットフォーム

における様々なハードウェアをサポートしているため、Apple のハードウェアに最適化されているわけではありません。

そこで Apple は新たに Metal というグラフィックス API を開発しました。Apple が独自に策定しているものですから、Apple のハードウェアに最適化し、性能を限界まで引き出すことができます。登場時には OpenGL ES と比較して「最大 10 倍速い」と謳っていました。

### 13.1.3 Metal の用途

Metal が発表された、2014 年の WWDC の基調講演におけるデモはこのようなものでした。

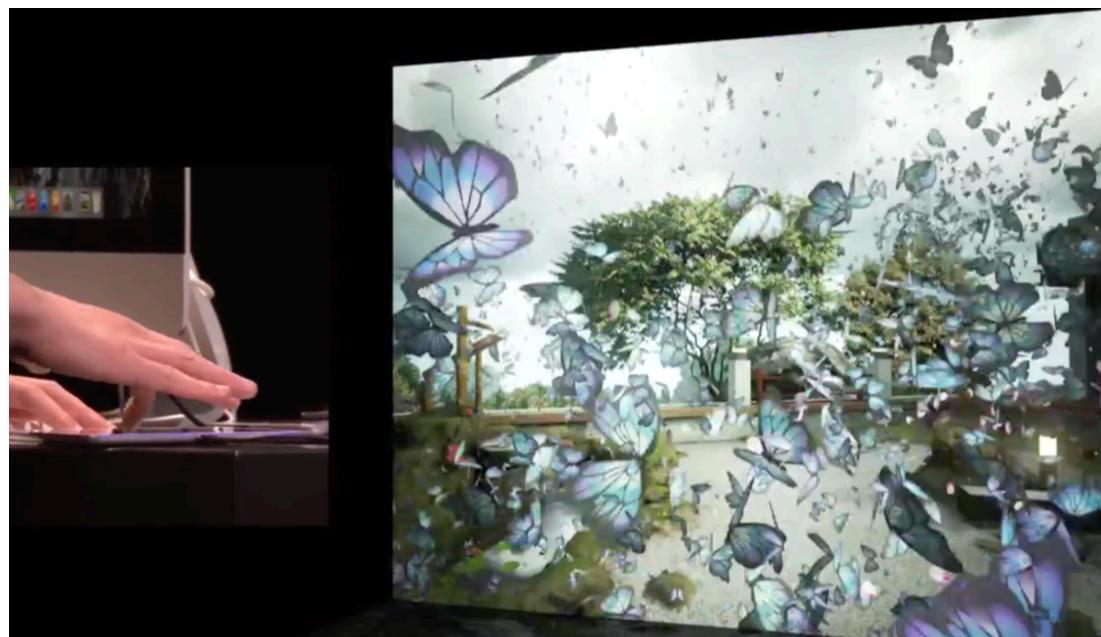


図 13.1: Metal 初登場時の WWDC 基調講演でのデモ

日本庭園風の 3D シーンの中で、桜の花びらが舞い散り、鯉の群れが池の波紋を追いかけ、砂紋を動的に描き、蝶の大群を舞わせる、というデモでした。

Metal の能力を活かした素晴らしいデモでしたが、逆にこういったデモを見ると、「Metal=重いグラフィックス処理を扱う本格的なゲーム向けの API」と考えてしまい、興味の対象外としてしまった開発者も多かったことでしょう。

実際には、Metal は多くの場所で陰ながら活躍しています。WWDC17 の "Platforms State of the Union" では次のような図が使われていました。

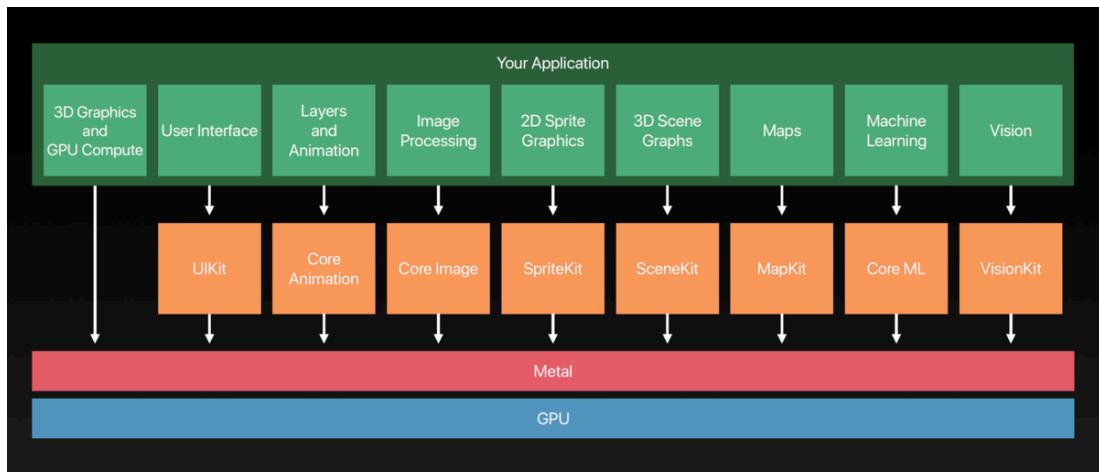


図 13.2: Metal を利用しているフレームワーク

TODO: 再作図: テキストは全てこのままで。配色はおまかせします。

画像処理を行う Core Image や、3D 空間の計算・描画を行う SceneKit はもちろんのこと、UIKit や Core Animation という iOS の超定番フレームワークの下回りでも実は Metal が利用されているのです。

多くのケースでは Metal は暗黙的に使用されているため、開発者が実際に Metal の API を直接利用する必要があるわけではないのですが、下位レイヤの概念を知っておくことは決して損はありません。

また、従来は重い 3D 描画を扱うアプリや特殊な写真加工を行うアプリの開発者でもない限り、OpenGL や Metal といった低レイヤのグラフィックス API を直接利用することは稀でしたが、近年ではその用途が拡大してきています。

たとえば今もっともホットなトピックである機械学習の、Convolutional Neural Network（畳み込みニューラルネットワーク）の計算を Metal で行う API が Metal Performance Shaders フレームワークで利用できますし、iOS 11 では RNN（Recurrent Neural Network）の API も追加されました。同じくホットな分野である AR を扱う ARKit でも、より自在なレンダリングを行うために Metal が利用できるようになっています。

## 13.2 Metal の基礎

### 13.2.1 Metal の基礎概念

Metal を扱う場合、たとえば「画像を描画する」というもっともシンプルな実装でも、多くのクラスが登場します。「最小実装」が UIKit のように小さくはないのです。GPU 周りを扱うプログラミングに慣れていないと、「たったこれだけのことをやるのに、なぜこんなに複雑になるのか」と不可解に思えてしまうかもしれません。

しかし、Metal 利用時に CPU と GPU がどう振る舞うか、その処理の流れや概念を理解しておけば、Metal の API は各段に理解しやすくなります。

まずはもっとも大まかな構成として、図 13.3 を見てください。

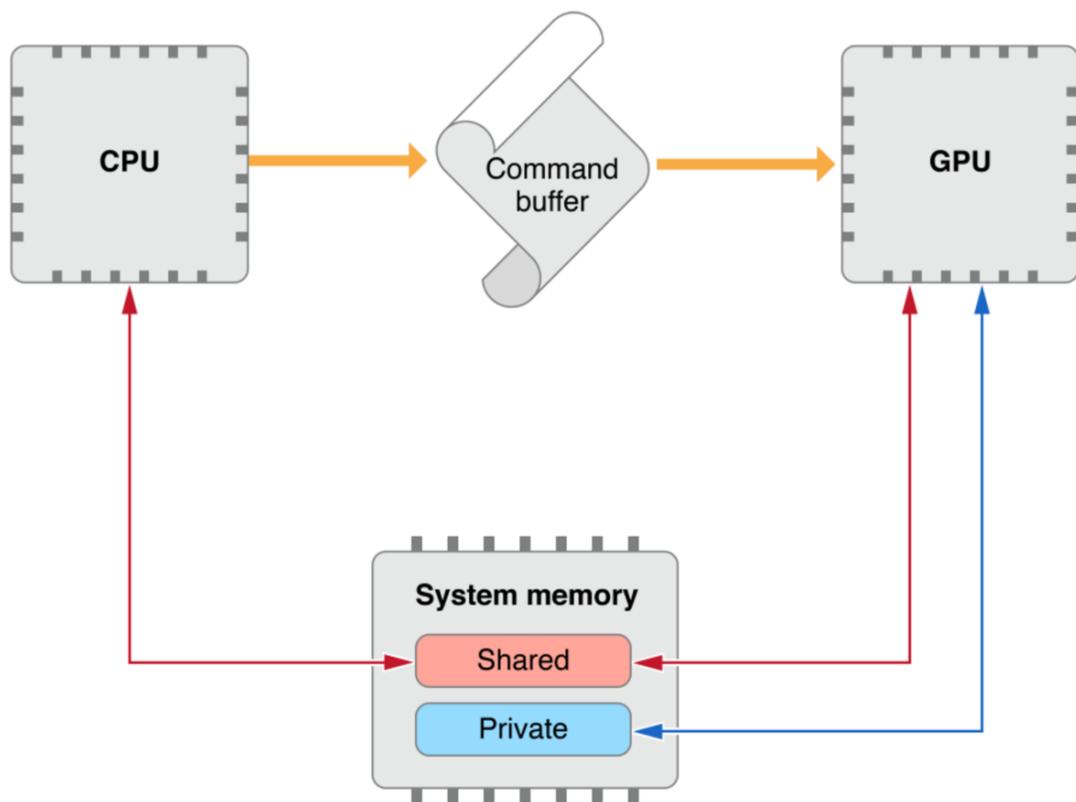


図 13.3: CPU・GPU とコマンドバッファ

TODO: 再作図: - shared/Private の四角はなくす- 青い矢印なくす  
この図は次のことを表現しています。

- CPU と GPU は両方からアクセスできるメモリ領域を持つ
- CPU は、GPU への命令（コマンド）をコマンドバッファという形で作成し、GPU に渡す
- GPU は渡されたコマンドを実行する

これが大前提となる概念です。これぐらいであればシンプルで直観的ではないでしょうか。このシンプルな概念を踏まえておくだけで、次から解説する Metal の基本クラスの役割がグッと理解しやすくなります。

### 13.2.2 Metal の基本クラス

#### MTLDevice

ハードウェアとしての GPU を抽象化したプロトコルです。

MTLDevice オブジェクトへの参照を取得するには、`MTLCREATESYSTEMDEFAULTDEVICE()` 関数を利用します。

```
func MTLCreateSystemDefaultDevice() -> MTLDevice?
```

クラスには属さない関数なので、そのまま利用できます。

```
let device = MTLCreateSystemDefaultDevice()
```

### MTLCommandBuffer

CPU で作成され、GPU で実行されるコマンドを格納するコンテナです。図 13.3 にあったとおり、このコマンドバッファ単位で **GPU** に渡され、実行されます。

コマンドバッファへのコマンド追加等が完了したら、コマンドバッファを「コミット」することで、このコマンドバッファが GPU によって実行されます（キューと絡めたより詳細な挙動は次項で解説します）。

いったんコミットしたコマンドバッファに対しては、登録しておいたスケジュールハンドラや処理完了ハンドラが実行されるのを待つか、その実行ステータスをチェックすることしかできません。これはつまり、コマンドバッファのオブジェクトは再利用はできないということでもあります。したがってプロパティ等にオブジェクトを保持するのではなく、都度生成して使います。

### MTLCommandQueue

コマンドバッファの実行順を管理するキューです。次のように `MTLDevice` から作成します。

```
let commandQueue = device.makeCommandQueue()
```

基本的にアプリに 1 つだけ作成し、Metal デバイス生存中はオブジェクトを維持しておきます。スレッドセーフです。

コマンドバッファ (`MTLCommandBuffer` オブジェクト) の作成は、このコマンドキューの役割です。次のように生成したコマンドバッファを、

```
let commandBuffer = commandQueue.makeCommandBuffer()
```

`commit()` メソッドを呼びコミットすると、生成元のコマンドキューにコマンドバッファが追加（エンキュー<sup>\*1</sup>）されます。

<sup>\*1</sup> `MTLCommandBuffer` には `enqueue()` というメソッドもあります。`enqueue()` を呼んだタイミングで、当該コマンドバッファのキューにおける実行順が確保されますが、コミット自体はされません（`commit()` を呼ばない限り実行され

```
commandBuffer.commit()
```

コマンドキューはエンキューされた順に GPU にコマンドバッファを送り、GPU はそのコマンドを実行します。こうしてコマンドキューによりコマンドバッファの実行順が保証されるというわけです。

### MTLCommandEncoder

コマンドを作成し、コマンドバッファに追加（エンコード）します。

コマンドエンコーダの種類を表 13.1 に示します。

表 13.1: コマンドエンコーダの種類

コマンドエンコーダ	概要
MTLRenderCommandEncoder	グラフィックスレンダリングコマンドをエンコードする
MTLComputeCommandEncoder	並列演算処理をエンコードする
MTLBlitCommandEncoder	バッファ・テクスチャ間のコピー処理をエンコードする
MTLParallelRenderCommandEncoder	並列実行する複数のグラフィックスレンダリングコマンドをエンコードする

コマンドエンコーダがコマンドをコマンドバッファに追加し、コマンドバッファがコマンドキューにエンキューされ、GPU で実行される、という関係を図示すると、次のようにになります。

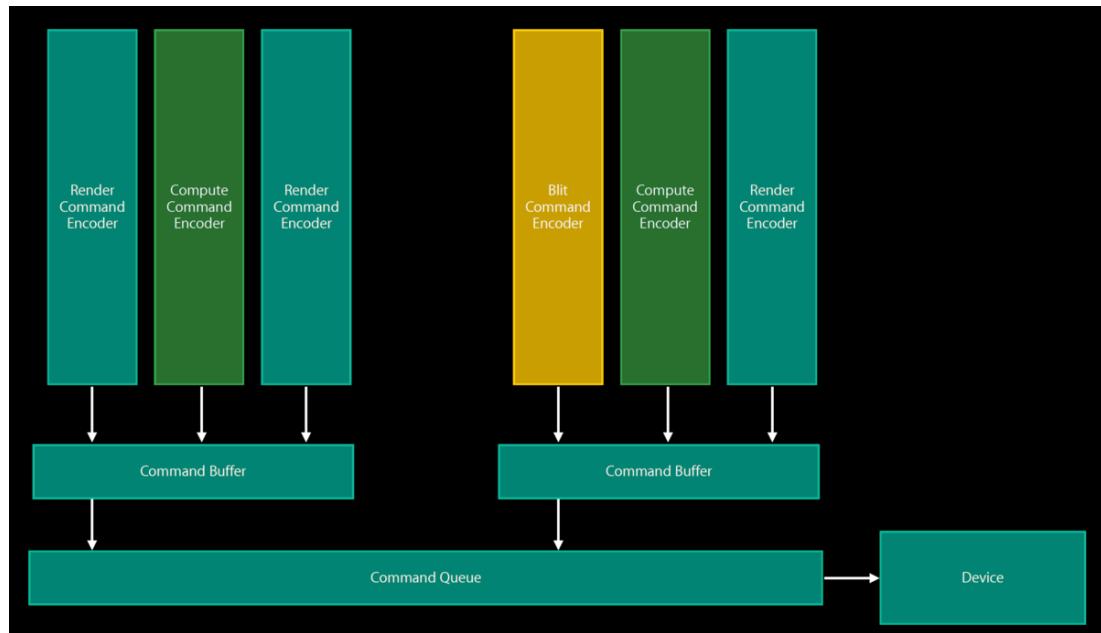


図 13.4: コマンドキュー、コマンドバッファ、コマンドエンコーダの関係

---

ません)。

TODO: 再作図: - 各テキストを以下のように変更してください（指示のないテキストはそのまま） - Command Buffer -> コマンドバッファ- Command Queue -> コマンドキュー- 上段の"XXX Command Encoder"すべて -> コマンドエンコーダ- Device -> GPU

### MTLBuffer, MTLTexture

`MTLTexture` はテクスチャデータを、`MTLBuffer` は頂点データ等、レンダリングに必要なパラメータを保持します。どちらのプロトコルも `MTLResource` に準拠し、図 13.3 における **GPU** からアクセス可能なメモリ領域上に確保されます。

## 13.3 MetalKit

前節では Metal の背景にある概念と、基本クラスについて解説しました。これらを使って実際に動くアプリケーションをつくっていく「実践」に入りたいところですが、その前にもう一点だけ、MetalKit というフレームワークを紹介します。

MetalKit を利用することで Metal によるグラフィックス描画の実装が簡単になります<sup>\*2</sup>。本節ではその中でも主要クラスである `MTKView` と `MTKTextureLoader` という 2 つのクラスについて解説します。これらは次の節「13.4 Metal 入門その 1 - 画像を描画する」の実装でも使用します。

### 13.3.1 MTKView

`MTKView` は Metal によって描画を行うビュークラスです。`UIView` を継承しているので、UIKit の他のビューと同じように扱えます。

初期化は IB から `MTKView` オブジェクトを選択して追加してもよいですし、イニシャライザから初期化して `addSubview(_:)` することもできます。

イニシャライザから初期化する場合は、第 2 引数に `MTLDevice` オブジェクトを渡します。

```
init(frame frameRect: CGRect, device: MTLDevice?)
```

IB から初期化した場合は、コードから明示的に `device` プロパティに `MTLDevice` オブジェクトをセットします。

```
var device: MTLDevice?
```

これらイニシャライザの第 2 引数、および `device` プロパティはオプショナル型になっていますが、`MTLDevice` オブジェクトを渡さないと描画ができません。Metal 非対応端末でもこのビューが存在できるようにこのような仕様（オプショナル型）になっていると予想されますが、Metal 対応

<sup>\*2</sup> MetalKit は、Metal 登場時の iOS 8 時点ではまだ存在せず、iOS 9 で新規追加されました。

端末にも関わらず何も描画されない場合等は、この `device` をセットし忘れていたり、何らかのミスで `nil` を渡そうとしていたりしていないか疑ってみましょう。

### MTKViewDelegate

`MTKView` では、何をどのように Metal で描画するかの実装を与える必要があります。`MTKViewDelegate` はその「`MTKView` で何を描画するのか」の実装を受け持つプロトコルです。

`MTKViewDelegate` プロトコルに準拠したクラスを用意し、そのオブジェクトを `MTKView` の `delegate` プロパティにセットします。

```
var delegate: MTKViewDelegate?
```

このプロトコルは以下 2 つのメソッドを持ちます。どちらも実装が必須です。

- 描画サイズに変更がある前に呼ばれるメソッド

```
func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize)
```

- ビューへの描画が依頼されると呼ばれるメソッド

```
func draw(in view: MTKView)
```

この `draw(in:)` に、Metal での描画処理を実装することで、Metal を用いたビューへの描画が実現できます。

### MTKViewDelegate の `draw(in:)` が呼ばれるタイミング

前述した通り、実際の描画処理は `MTKViewDelegate` の `draw(in:)` メソッドに実装するわけですから、このメソッドがいつ呼ばれるのか、その仕様について知っておくことは重要です。

デフォルトでは内部のタイマによって自動的に再描画が行われ、毎フレーム `draw(in:)` が呼びられます。

しかし、`isPaused` または `enableSetNeedsDisplay` プロパティに `true` がセットされている場合は、自動的な再描画は行われず、`setNeedsDisplay()` メソッドが呼ばれたタイミングで `MTKViewDelegate` の `draw(in:)` が呼びれます。

また一時停止中(`isPaused` が `true`)であり、`setNeedsDisplay` が無効(`enableSetNeedsDisplay` が `false`)の場合は、親クラスである `UIView` の `draw(_:)` メソッドを明示的に呼ぶことでこの `draw(in:)` を発火させることができます。

### 13.3.2 MTKTextureLoader

`MTKTextureLoader` は、GPU からアクセスできるメモリ領域<sup>\*3</sup>にテクスチャデータをロードし、`MTLTexture` オブジェクトを作成するためのクラスです。

初期化時は `MTLDevice` オブジェクトを渡します。

```
init(device: MTLDevice)
```

様々なソースから `MTLTexture` を作成するメソッドを備えており、たとえば、次のメソッドはアセットカタログにある `name` で指定された画像アセットから画像データをテクスチャとしてロードします。

```
func newTexture(name: String, scaleFactor: CGFloat, bundle: Bundle?, options: [MTKTextureLoader.Option : Any]?)
```

第2引数の `scaleFactor` にはアセットカタログから取得するテクスチャのスケールファクター(2x や 3x)を `CGFloat` 型で指定します。

他にも、URL、`CGImage`、`Data` 等からロードするメソッドも備えています。

```
func newTexture(URL: URL, options: [MTKTextureLoader.Option : Any]?)
```

```
func newTexture(cgImage: CGImage, options: [MTKTextureLoader.Option : Any]?)
```

```
func newTexture(data: Data, options: [MTKTextureLoader.Option : Any]?)
```

また、非同期でテクスチャをロードするメソッドも用意されています。

```
func newTexture(name: String, scaleFactor: CGFloat, bundle: Bundle?, options: [MTKTextureLoader.Option : Any]?, completion: ((MTLTexture?) -> Void))
```

<sup>\*3</sup> 「13.2 Metal の基礎」を参照。

## 13.4 Metal 入門その1 - 画像を描画する

Metal の背景にある概念や基本クラス、そして Metal による実装を簡単にしてくれる MetalKit について学んだところで、いよいよここから実践に入っていきましょう。

まずははじめの一歩として、「Metal の API を使って画像を描画」してみます。

画像を1枚だけ描画するというのは、並列演算という GPU の強みがあまり活きる題材ではありません。ですが、後述するシェーダを利用せずに実現できるため、新しく出てくる概念が比較的少なく済みます。また結果をデバイス上で明確に・簡単に確認できるので、「Metal 実装の第一歩目」には最適です。

本節では、MTKView を利用して、Metal で画像を描画します。処理の大まかな流れとしては次の通りです。

1. 描画処理のためのセットアップを行う
2. 画像をテクスチャとしてロードする
3. 描画処理を実行する

それでは具体的に見てきましょう。

(サンプルコード: 01\_MetalImageRender)

### 13.4.1 1. 描画処理のためのセットアップを行う

Metal で描画処理を行うために、各種オブジェクトの初期化等、セットアップ処理を実装します。ここで実装する処理は、基本的にはフレーム毎に変化する要因がないので、一度だけ行えば OK です。描画ループの各フレームごとに行う必要はありません。

まず、MTLDevice オブジェクトを取得します。これは Metal を扱う場合は必ず行う処理です。

```
private let device = MTLCreateSystemDefaultDevice()!
```

#### ■コラム: Metal 非対応デバイスの判定

MTLCreateSystemDefaultDevice() 関数は、Metal 非対応デバイスでは nil を返します。したがって通常はこの関数の結果が nil かどうかを判定して、Metal 非対応デバイス向けの処理を書きます。

本章ではコードをシンプルにするために、デバイスが Metal に対応していることを前提にしています。

Metal 対応デバイスについては章末の「13.10 Metal を動作させるためのハードウェア要件」を参照してください。

作成した `MTLDevice` オブジェクトから、`MTLCommandQueue` を初期化します。`MTLCommandQueue` オブジェクトは再利用可能なので、プロパティに保持することにします。

```
private var commandQueue: MTLCommandQueue!
```

```
commandQueue = device.makeCommandQueue()
```

`MTKView` を初期化します。前述した通りプログラムから初期化してもよいですが、ここでは IB からオブジェクトを追加することにします。

```
@IBOutlet private weak var mtkView: MTKView!
```

`MTKView` の `device` プロパティと `delegate` プロパティをセットします。<sup>\*4</sup>

```
mtkView.device = device
mtkView.delegate = self
```

### 13.4.2 2. 画像をテクスチャとしてロードする

描画したい画像を `MTLTexture` としてロードします。`MTKTextureLoader` のメソッドを 1 つ呼ぶだけです。

`MTLTexture` を保持するプロパティを用意しておき、

```
private var texture: MTLTexture!
```

`MTKTextureLoader` を初期化し、

```
let textureLoader = MTKTextureLoader(device: device)
```

---

<sup>\*4</sup> 「13.3.1 MTKView」を参照

テクスチャをロードするメソッドを呼びます。<sup>5</sup>

```
texture = try! textureLoader.newTexture(
 name: "filename",
 scaleFactor: view.contentScaleFactor,
 bundle: nil)
```

MTKView のピクセルフォーマット (MTLPixelFormat 型) を、表示するテクスチャのピクセルフォーマットに合わせます。

```
mtkView.colorPixelFormat = texture.pixelFormat
```

このピクセルフォーマットが合っていないと、実行時にエラーが発生し、クラッシュしてしまいます。

### 13.4.3 3. 描画処理を実行する

「MTKViewDelegate」で解説したとおり、MTKView への描画の依頼があるたびに、MTKViewDelegate の draw(in:) メソッドが呼ばれるので、ここに Metal での描画処理を実装します。

```
func draw(in view: MTKView) {
 // 描画処理
}
```

Metal では、GPU への描画命令をコマンドバッファという形で作成し、GPU に渡す、ということを「13.2 Metal の基礎」で述べました。

コマンドバッファ生成～コミット（エンキュー）までの流れは「MTLCommandQueue」で解説した通りなので、ここで考えるべきは「どのように『画像を描画する』というコマンドを作成し、コマンドバッファに追加するか」です。

```
// コマンドバッファを作成
let commandBuffer = commandQueue.makeCommandBuffer()

// コマンドを作成
```

<sup>5</sup> MTKTextureLoader の newTexture(name:scaleFactor:bundle:options:) メソッドは「13.3.2 MTKTextureLoader」を参照。

```
...
// コマンドを追加（エンコード）
...
// コマンドバッファをコミット（エンキュー）→GPUに送られる
commandBuffer.commit()
```

これを考えるために、まず、「ドローアブル」について解説します。

### ドローアブル

「ドローアブル」（Metalにおいては `CAMetalDrawable` プロトコルに準拠するオブジェクト）とは、「Metalによってレンダリング、あるいは書き込まれる、表示可能なリソース」を表します。

この `CAMetalDrawable` はさらに `MTLDrawable` プロトコルに準拠しており、さまざまなプロパティやメソッドを持ちますが、ここで重要なのは次の `texture` プロパティです。

```
var texture: MTLTexture { get }
```

この `MTLTexture` オブジェクトが、`CAMetalDrawable` オブジェクトが表す「表示可能なリソース」のコンテンツの実体となります。

そしてこの `CAMetalDrawable` オブジェクト（`MTLDrawable` を継承）を `MTLCommandBuffer` の `present(_:)` メソッドの引数に渡すことで、Metalは与えられた `drawable` を画面に表示します。

```
func present(_ drawable: MTLDrawable)
```

すなわち、`drawable.texture` に与えたテクスチャが画面に表示されます。

表示のタイミングをコントロールするためのメソッドも用意されています。

```
func present(_ drawable: MTLDrawable,
 afterMinimumDuration duration: CFTimeInterval)
```

```
func present(_ drawable: MTLDrawable,
 atTime presentationTime: CFTimeInterval)
```

ですが、このメソッドを呼ぶこと自体が画面表示の実行を意味するわけではありません。このメ

ソッドはあくまでこのコマンドバッファではこの `drawable` を表示に用いる、ということを登録しているだけであって、先に説明したとおり、当該コマンドバッファをコミットすることで、キューの順番に従って GPU に送られ、処理されるようになります。

### MTKView の `currentDrawable`

`CAMetalDrawable` は、イニシャライザを持ちません。つまり、開発者はオブジェクトを作成できません。

```
let drawable = CAMetalDrawable() // ビルドエラー
```

MTKView を利用する場合は、`currentDrawable` という `CAMetalDrawable` 型のプロパティからドローアブルを取得できます。

```
var currentDrawable: CAMetalDrawable?
```

このプロパティはその名のとおり「現在のフレームで用いられるドローアブル」を保持しているので、次のように `draw(in:)` の中で取得することで、毎フレームのドローアブルにアクセスできます。

```
func draw(in view: MTKView) {
 guard let drawable = view.currentDrawable else {return}
```

さて、ここで本節の目的である「画像を描画する」という本題に戻りましょう。画像を描画するシンプルな方針としては、以下のことができればよいわけです。

- `currentDrawable` の `texture` に表示したい画像のデータを渡す
- `currentDrawable` を引数に渡して、`MTLCommandBuffer` の `present(_:)` メソッドを呼ぶ

ただし、次のように `CAMetalDrawable` の `texture` プロパティに直接 `MTLTexture` をセットすることはできません。

```
guard let drawable = view.currentDrawable else {return}
drawable.texture = texture // ビルドエラー
```

次のようなビルトエラーになってしまいます。`texture` プロパティは外部からは `get` しかできないように定義されているためです。

```
Cannot assign to property: 'texture' is a get-only property
```

しかしこの `texture` プロパティが保持する `MTLTexture` オブジェクトに、テクスチャデータをコピーすることは可能です。ここで登場するのが `MTLBlitCommandEncoder` です。

### ブリットコマンドエンコーダ

コマンドを作成し、コマンドバッファに追加（エンコード）する役割を担うコマンドエンコーダにはいくつかの種類があります<sup>\*6</sup>。この中でも `MTLBlitCommandEncoder` はバッファやテクスチャをコピーするコマンドを扱うエンコーダ<sup>\*7</sup>です。

テクスチャをコピーするというタスクは、他のコマンドエンコーダでも行えるのですが、`MTLBlitCommandEncoder` は、他のエンコーダと違いシェーダを書く必要がないため、非常に簡単に扱えます。

これを用いてドローアブルが持つテクスチャにデータをコピーする処理を実装しましょう。

`MTLBlitCommandEncoder` は、次のように `MTLCommandBuffer` から作成できます。

```
let blitEncoder = commandBuffer.makeBlitCommandEncoder()!
```

テクスチャからテクスチャへデータをコピーするメソッドは次のように定義されています。

```
func copy(from sourceTexture: MTLTexture,
 sourceSlice: Int,
 sourceLevel: Int,
 sourceOrigin: MTLOrigin,
 sourceSize: MTLSize,
 to destinationTexture: MTLTexture,
 destinationSlice: Int,
 destinationLevel: Int,
 destinationOrigin: MTLOrigin)
```

引数が多く一見使用方法が難しく見えますが、コピー元・コピー先の情報を指定しているだけです。

第1引数の `from` にコピー元の `MTLTexture` オブジェクトを、第6引数の `to` にコピー先の `MTLTexture` オブジェクトを渡します。

第2～第5引数は、それぞれコピー元テクスチャの、コピーするスライス、ミップマップレベル、原点、サイズを指定します。

第7～第9引数は、それぞれコピー先テクスチャの、コピー先となるスライス、ミップマップレベル、原点を指定します。

<sup>\*6</sup> 表13.1

<sup>\*7</sup> コンピュータグラフィックスの世界でビットマップデータ等を転送する操作のことを"Bit blit"と呼びます。

表示に用いる `CAMetalDrawable` オブジェクトの `texture` プロパティに、アセットカタログからロードしてきた `MTLTexture` オブジェクトをコピーする場合の実装は次のようにになります。

```
blitEncoder.copy(from: texture, // コピー元テクスチャ
 sourceSlice: 0,
 sourceLevel: 0,
 sourceOrigin: MTLOrigin(x: 0, y: 0, z: 0),
 sourceSize: MTLSizeMake(w, h, texture.depth),
 to: drawable.texture, // コピー先テクスチャ
 destinationSlice: 0,
 destinationLevel: 0,
 destinationOrigin: MTLOrigin(x: 0, y: 0, z: 0))
```

これでコピー処理を行うコマンドがコマンドバッファにエンコードされます。  
エンコードが完了したら、`endEncoding()` メソッドを呼びます。

```
blitEncoder.endEncoding()
```

このメソッドは親プロトコルである `MTLCommandEncoder` で定義されているもので、「エンコードが完了したらこのメソッドを呼ぶ」という手順はどの種類のコマンドエンコーダにも共通です。

### 描画処理の全体

さて、ここまで多くのことを解説したので、描画処理の実装の全体を見てみましょう。

```
func draw(in view: MTKView) {
 // ドローアブルを取得
 guard let drawable = view.currentDrawable else {return}

 // コマンドバッファを作成
 let commandBuffer = commandQueue.makeCommandBuffer()!

 // コピーするサイズを計算
 let w = min(texture.width, drawable.texture.width)
 let h = min(texture.height, drawable.texture.height)

 // MTLBlitCommandEncoder を作成
 let blitEncoder = commandBuffer.makeBlitCommandEncoder()!

 // コピーコマンドをエンコード
 blitEncoder.copy(from: texture, // コピー元テクスチャ
 sourceSlice: 0,
 sourceLevel: 0,
 sourceOrigin: MTLOrigin(x: 0, y: 0, z: 0),
 sourceSize: MTLSizeMake(w, h, texture.depth),
 to: drawable.texture, // コピー先テクスチャ
```

```
 destinationSlice: 0,
 destinationLevel: 0,
 destinationOrigin: MTLOrigin(x: 0, y: 0, z: 0))

 // エンコード完了
 blitEncoder.endEncoding()

 // 表示するドローアブルを登録
 commandBuffer.present(drawable)

 // コマンドバッファをコミット（→エンキュー）
 commandBuffer.commit()
}
```

複雑に見えますが、全体としては「コマンドバッファを作成し、コミットする」のが描画ループで行うべき処理で、そのコマンドバッファ作成過程において、「画像をドローアブルの持つテクスチャ領域にコピーする」コマンドをエンコードしている、という構成です。意外とシンプルに見えてこないでしょうか。

これで、次のようにアセットカタログにある画像を表示することができました。



### 13.5 Metal 入門その 2 - シェーダを利用する

「入門その 1」では、プリットコマンドエンコーダを利用して画像を Metal で描画する機能を実現しました。そこからスタートした理由は、プリットコマンドエンコーダはシェーダを使わずに済み、シンプルな形で Metal の基本実装を理解できるためです。

ここからは「レンダーコマンドエンコーダ」<sup>\*8</sup>を利用して、いよいよシェーダによるグラフィックスレンダリング処理を実装していきます。シェーダを使うためには理解すべき概念が増えますが、シェーダを扱ってこそその GPU プログラミングでもあります。まずはというシンプルなシェーダの実装から始めて、基本的な概念や実装方法を理解しましょう。

本節では次のように「画面を一色に塗る」実装を行います。

---

<sup>\*8</sup> 表 13.1

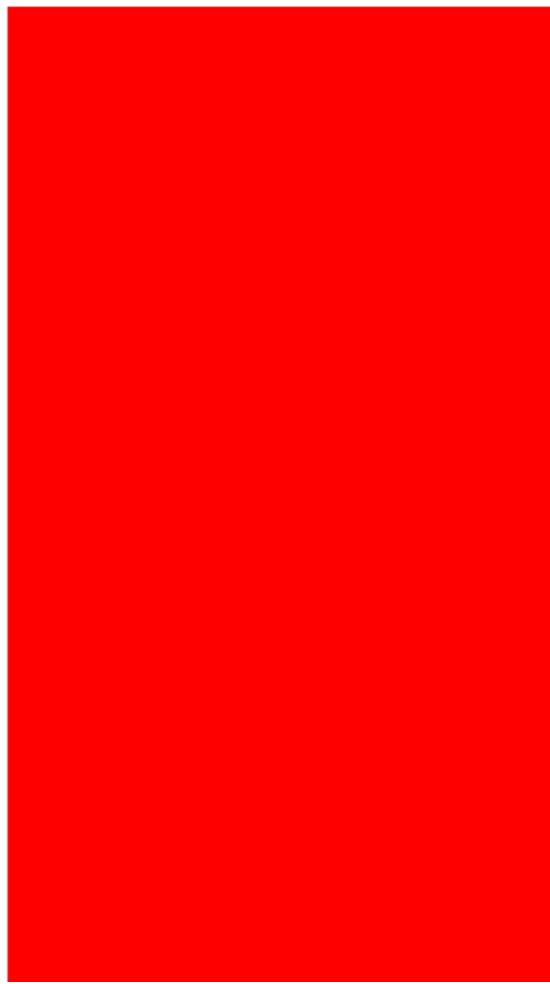


図 13.5: 画面全体をシェーダで赤に描画

これ自体は面白みのあるサンプルではありませんが、Metal でシェーダを扱うにあたって必要な概念や実装方法が最小限のコードで理解できますし（Xcode のプロジェクトテンプレートで生成されるコードはもっと複雑）、ここに出てくる事項はシェーダでどんな処理を行うにしても不可欠です。ぜひ押さえておきましょう。

（サンプルコード: 02\_MetalShaderColorFill）

### 13.5.1 Metal シェーダの基礎

#### シェーダとは

シェーダは、**GPU** 上で並列に実行されるプログラムです。

本節では 2 種類のシェーダが登場します。

- 頂点シェーダ・・・頂点（vertex）に関する計算を行うシェーダ。たとえば頂点の座標変換を行う。
- フラグメントシェーダ・・・ピクセルに関する計算を行うシェーダ。当該ピクセルの最終的

な「色」を決める。

ここで明確に頭に留めておくべきことは、「このプログラムに書いた処理は、複数の入力に対して並列実行される」という点です。

たとえば頂点シェーダであれば、GPUに渡した頂点データそれぞれに対して頂点シェーダのプログラムが実行されますし、フラグメントシェーダであれば最終的に表示するディスプレイのピクセル数だけ、それぞれのピクセルに対して同じプログラムが実行されます。

ここがこれまで書いてきたCPU向けのプログラムとは大きく違う点でしょう。

### Metal Shading Language

Metalのシェーダは"**Metal Shading Language**"（以下MSLと略します）というC++をベースとした独自言語で記述します。言語仕様は"**Metal Shading Language Specification**"とタイトルでAppleから公開されています<sup>\*9</sup>。

### MetalシェーダファイルとMetalライブラリファイル

Metalのシェーダプログラムを記述するファイルは、拡張子を.**.metal**にします。この拡張子を持つファイルがビルド時にプリコンパイルされ、拡張子.**.metallib**の**Metal**ライブラリファイルが生成されます<sup>\*10</sup>。

### MTLLibraryとMTLFunction

**MTLLibrary**はMetalライブラリを表すプロトコルで、**MTLDevice**の**makeDefaultLibrary()**メソッドを呼ぶことでオブジェクトを作成できます。

```
let library = device.makeDefaultLibrary()
```

メソッド名に「デフォルトライブラリ」とありますが、これは、「アプリケーションのメインバンドルに含まれるMetalライブラリファイルから生成された**MTLLibrary**オブジェクト」を意味します。

**MTLLibrary**の**makeFunction(name:)**メソッドにシェーダ関数名を**String**型で指定することで、各シェーダ関数を表す**MTLFunction**オブジェクトを生成できます。

```
let vertexFunction = library.makeFunction(name: "vertexShader")
let fragmentFunction = library.makeFunction(name: "fragmentShader")
```

<sup>\*9</sup> <https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf>

<sup>\*10</sup> OpenGLのシェーダ言語GLSLは実行時にコンパイルされる方式のため、ここは大きくMetalとOpenGL ESとで違う点です。Appleはこのプリコンパイル方式もMetalがOpenGL ESよりも高速な理由のひとつとして挙げています。

### 13.5.2 「画面全体を一色に塗る」シェーダの実装

シェーダプログラムを実装していきましょう。手順としては次の3ステップです。

1. Metal シェーダファイルの作成
2. 頂点シェーダ関数の実装
3. フラグメントシェーダ関数の実装

#### 1. Metal シェーダファイルの作成

.metal 拡張子を持つファイルを作成します。この Metal シェーダファイルは、Xcode の [File]>[New]>[File] から [Metal File] テンプレートを選択して作成することもできます。  
まず"metal\_stdlib"をインポートし、ネームスペース `metal` の利用を宣言します。

```
#include <metal_stdlib>
using namespace metal;
```

ネームスペースの宣言をしなくても Metal シェーダを書くことはできますが、その場合、たとえば `float4` という型をひとつ使うにしても `metal::float4` と書く必要があります。

#### 2. 頂点シェーダ関数の実装

前述した通り、頂点シェーダは「頂点（vertex）に関する計算を行うシェーダ」です。たとえば頂点の座標変換が頂点シェーダで行うべき処理の代表例ですが、本節の目的は「画面全体を一色に塗る」ことなので、実はこの頂点シェーダで処理すべきことは特にありません。

そのため本項では「何もしない」（受け取った頂点データをそのまま出力する）頂点シェーダ関数を実装します。

まず、頂点シェーダの出力となる構造体を定義します。

```
struct ColorInOut
{
 float4 position [[position]];
};
```

この構造体がフラグメントシェーダの入力の型にもなるので、フラグメントシェーダでの処理で使いたいデータも考慮して構造体を定義することになります。

さて、`[[ position ]]` という見慣れない記法が出てきましたが、このように `[[ ... ]]` で囲まれたものは"**Attribute Qualifier**"（属性修飾子）と呼ばれます。それぞれの属性修飾子は、MSL の言語仕様で決められた意味や仕様が定められています。

Metal の頂点シェーダが出力として構造体を返す場合、その構造体はこの `[[ position ]]` で

修飾された要素を持つ必要があります。もしくは頂点シェーダが `float4` を返す場合はその値は暗黙的にその頂点の座標を示すことになり、修飾子は不要です。また `[[ position ]]` で修飾される要素の型は `float4` で、座標 (`x, y, z, w`) を示します<sup>\*11</sup>。

そして、「何もしない」頂点シェーダ関数の実装は次のようにになります。

```
vertex ColorInOut vertexShader(device float4 *positions [[buffer(0)]],
 uint vid [[vertex_id]])
{
 ColorInOut out;
 out.position = positions[vid];
 return out;
}
```

まず、冒頭の `vertex` によって、頂点シェーダの関数であることを宣言しています。また戻り値では先に定義した構造体 `ColorInOut` を型として指定しています。

第1引数にある修飾子 `[[ buffer(n) ]]` で示された引数へは、Swift プログラム側から渡したバッファの内容が入ってきます。（「13.5.3 シェーダにデータを渡すためのバッファを準備する」にて後述）

第2引数にある属性修飾子 `[[ vertex_id ]]` は、型は `ushort` または `uint` で、頂点のインデックスを示します。このインデックスを用いて、修飾子 `[[ buffer(n) ]]` で示された引数に入ってくる頂点データに `positions[vid]` のようにしてアクセスできます。

また第1引数についている修飾子 `device` は、“Address Space Qualifiers”（アドレス空間修飾子）と呼ばれるもののひとつで、変数や引数を割り当てるメモリ領域を特定するためのものです。ポインタ型・参照型の引数や変数に対しては必ず指定する必要があります。

グラフィックスレンダリングのためのシェーダ関数（頂点シェーダ・フラグメントシェーダ）の場合は、`device` および `constant` が指定できます。どちらもデバイスマトリプルに確保されますが、`constant` は `read-only` で、`device` は読み出し・書き込み両方可能です。

### 3. フラグメントシェーダ関数の実装

フラグメントシェーダ関数を実装します。「画面全体を一色に塗る」ためには、フラグメントシェーダでは各ピクセルについてただ一色だけを出力します。

コードは次のようになります。

```
fragment float4 fragmentShader(ColorInOut in [[stage_in]])
{
 return float4(1,0,0,1);
}
```

冒頭の `fragment` によって、フラグメントシェーダの関数であることを宣言しています。

---

<sup>\*11</sup> `float4` は4つの `float` からなる4次元ベクトル。`[[ position ]]` では3次元座標を扱うので `w` は1。

フラグメントシェーダは`[[ stage_in ]]`で修飾される引数を持ち、ここにはラスタライズされたフラグメントデータ（ディスプレイの1ピクセル分を描画するために用いられる構造体）が入ってきます。型は頂点シェーダの出力と同じになりますが、頂点シェーダで出力した頂点データそのものが入ってくるわけではありません。繰り返しになりますが、頂点シェーダでの処理後にラスタライズが行われ、その出力であるフラグメントデータが入ってきます。

つまり、ディスプレイの各ピクセルに対応する情報が、頂点シェーダで使用したのと同じ型で入ってくるということです。本節では構造体の要素として座標を示す`position`を持たせたので、`[[ stage_in ]]`で修飾された引数`in`から「処理中のピクセルの座標値」が得られることになります。

最終的に`float4`型<sup>\*12</sup>で`(r, g, b, a)`の値を返すことで、該当するフラグメント（ピクセル）の「色」を決定することになります。

本実装では引数に入ってきた値が何であれ、戻り値として一律`float4(1, 0, 0, 1)`を返しているので、すべてのピクセルに対して`float4(1, 0, 0, 1)`が返されることになり、画面全体が赤色に塗られます。

### 13.5.3 シェーダにデータを渡すためのバッファを準備する

ここからはCPUプログラム側の実装を見ていきます。

まずは、シェーダにデータを渡すためのバッファを作成しましょう。

前項で実装した頂点シェーダでは、

```
vertex ColorInOut vertexShader(device float4 *positions [[buffer(0)]],
 uint vid [[vertex_id]])
```

のように、バッファから受け取る引数を1つだけ（第1引数）持っていました。

ここに渡すための頂点座標のデータを次のように用意します。

```
private let vertexData: [Float] = [
 -1, -1, 0, 1,
 1, -1, 0, 1,
 -1, 1, 0, 1,
 1, 1, 0, 1
]
```

`Float`型の配列で、各列の4つの値が`(x, y, z, w)`で1つの頂点座標を表しています。ここで各軸の値域は $-1 \sim 1$ で、 $x$ - $y$ 平面は左下から右上に向かって座標の値が大きくなります。すなわちこの4頂点は「画面いっぱいの四角形」を表していることになります。

`MTLBuffer`<sup>\*13</sup>オブジェクトを保持するプロパティを用意し、

\*12 半精度の`half4`も利用可能。

\*13 「`MTLBuffer`, `MTLTexture`」

```
private var vertexBuffer: MTLBuffer!
```

MTLDevice の makeBuffer(bytes:length:options:) メソッドに、上で作成した頂点データのポインタとそのサイズを渡して、MTLBuffer オブジェクトを作成します。・

```
let size = vertexData.count * MemoryLayout<Float>.size
vertexBuffer = device.makeBuffer(bytes: vertexData, length: size, options: [])
```

これで頂点座標データの入ったバッファが作成されました。

#### 13.5.4 MTLRenderPipelineDescriptor を作成する

レンダリングのための一連の処理をレンダリングパイプラインと呼びます。MTLRenderPipelineState はグラフィックスレンダリングパイプラインを表すプロトコルで、頂点シェーダやフラグメントシェーダの関数、および設定を保持します。

MTLRenderPipelineState オブジェクトを生成するにあたって、まずはそのレンダリングパイプラインの設定を記述したレンダリングパイプラインディスクリプタ (MTLRenderPipelineDescriptor) を用意する必要があります。

MTLRenderPipelineDescriptor はプロトコルではなくクラスで、通常のクラスと同じ方法で初期化できます。

```
let descriptor = MTLRenderPipelineDescriptor()
```

レンダリングパイプラインディスクリプタにはさまざまな設定項目がありますが、本節のケースでは次の2つの設定を行います。

- レンダリングパイプラインで用いるシェーダ関数を指定する
- レンダリング先のピクセルフォーマットを指定する

##### レンダリングパイプラインで用いるシェーダ関数を指定する

次のように MTLRenderPipelineDescriptor の vertexFunction、fragmentFunction プロパティに、それぞれ頂点シェーダとフラグメントシェーダの MTLFunction オブジェクト<sup>\*14</sup>をセットします。

<sup>\*14</sup> MTLLibrary オブジェクトを生成し MTLFunction オブジェクトを取得する方法については、「MTLLibrary と MTLFunction」を参照してください。

```
descriptor.vertexFunction = library.makeFunction(name: "vertexShader")
descriptor.fragmentFunction = library.makeFunction(name: "fragmentShader")
```

レンダリング先のピクセルフォーマットを指定する

`MTLRenderPipelineDescriptor` は `MTLRenderPipelineColorAttachmentDescriptorArray` 型の `colorAttachments` プロパティを持ちます。

`MTLRenderPipelineColorAttachmentDescriptorArray` 型は `Array` と違って `RandomAccessCollection` や `MutableCollection` に準拠していないのですが、次のように `MTLRenderPipelineColorAttachmentDescriptor` 型の値を返す `subscript` が定義されています。

```
open subscript(attachmentIndex: Int) -> MTLRenderPipelineColorAttachmentDescriptor!
```

そのため、`descriptor.colorAttachments[0]` のようにインデックスを用いて `MTLRenderPipelineColorAttachmentDescriptor` オブジェクトにアクセスできます。

`MTLRenderPipelineColorAttachmentDescriptor` は、レンダリング先のカラー関連の設定を指定するためのものです。その中でも `pixelFormat` プロパティは必ず指定する必要があります。

本節では一色に塗りたいだけなので、動的に何かのピクセルフォーマットに合わせる必要も、特殊なフォーマットを指定する必要もありません。`bgra8Unorm` を指定しておくことにします。

```
descriptor.colorAttachments[0].pixelFormat = .bgra8Unorm
```

### 13.5.5 MTLRenderPipelineState を生成する

`MTLRenderPipelineDescriptor` が用意できたら、`MTLDevice` の `makeRenderPipelineState(descriptor:)` メソッドを用いて `MTLRenderPipelineState` オブジェクトを生成します。

```
private var renderPipeline: MTLRenderPipelineState!
```

実装は次の通りです。

```
renderPipeline = try! device.makeRenderPipelineState(descriptor: descriptor)
```

なお、Appleによる公式ドキュメント「Metal Programming Guide」<sup>\*15</sup>の"Creating a Render Pipeline State from a Descriptor"という項<sup>\*16</sup>にて、「`MTLRenderPipelineState`の生成は高コストなので、できるだけ再利用すること」と明記されています。同じレンダリングパイプラインが使いまわせる場合は毎フレーム生成したりしないようにしましょう。

### 13.5.6 レンダリングコマンドの作成

冒頭での説明の繰り返しになりますが、グラフィックスレンダリングコマンドをエンコードするには「レンダーコマンドエンコーダ」(`MTLRenderCommandEncoder`)を使用します。

レンダーコマンドエンコーダを用いてレンダリングコマンドを作成するにあたって、以下の手順が必要です。

1. レンダーパスディスクリプタを作成する
2. 1を用いてレンダーコマンドエンコーダを生成する
3. 2を用いてレンダリングコマンドのさまざまな設定を行う
4. プリミティブを描画する

#### 1. レンダーパスディスクリプタを作成する

レンダーパスディスクリプタ (`MTLRenderPassDescriptor`) はレンダリング先のテクスチャ等の「アタッチメント」を指定するためのものです。`MTLRenderCommandEncoder` を生成する際に引数に渡すので、必ず作成する必要があります。

ただし、コマンドエンコーダは（コミットした後は再利用できないため）毎フレーム作成する必要がありますが、`MTLRenderPassDescriptor` オブジェクトは再利用可能なので、一度だけ作成しておいて使いまわすことができます。

このディスクリプタ自体は設定を記述するものなので、(`MTLDevice` から作成したりする必要はなく) 通常のクラスと同様の初期化方法で作成できます。

```
let renderPassDescriptor = MTLRenderPassDescriptor()
```

もしくは、`MTKView`を利用している場合は、`currentRenderPassDescriptor` プロパティから取得することも可能です。

```
guard let renderPassDescriptor = currentRenderPassDescriptor else {return}
```

`MTLRenderPassDescriptor` は `MTLRenderPassColorAttachmentDescriptorArray` 型の

<sup>\*15</sup> <https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html>

<sup>\*16</sup> <https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Render-Ctx/Render-Ctx.html>

`colorAttachments` プロパティを持ちます。

`MTLRenderPassColorAttachmentDescriptorArray` 型は `MTLRenderPassColorAttachmentDescriptor` 型の値を返す `subscript` が定義されているので、`renderPassDescriptor.colorAttachments[0]` のようにインデックスを用いて `MTLRenderPassColorAttachmentDescriptor` オブジェクトにアクセスできます。

`MTLRenderPipelineColorAttachmentDescriptor` クラスは `MTLRenderPassAttachmentDescriptor` のサブクラスで、グラフィックスレンダリングにより生成されるピクセルデータの色値の出力先を記述するためのディスクリプタです。

`MTLRenderPipelineColorAttachmentDescriptor` およびその親クラスの `MTLRenderPassAttachmentDescriptor` は様々なプロパティを持ちますが、ここでは `texture` プロパティに出力先となる `MTLTexture` オブジェクトを指定しておきます。

```
renderPassDescriptor.colorAttachments[0].texture = drawable.texture
```

本節では `MTKView` にシェーダを用いたグラフィックスレンダリング結果を描画しようとしているので、`MTKView` の `currentDrawable` プロパティから取得したドローアブルのテクスチャを渡しています。

## 2. レンダーコマンドエンコーダを生成する

`MTLRenderPassDescriptor` が用意できたら、これを引数に渡して `MTLCommandBuffer` の `makeRenderCommandEncoder(descriptor:)` メソッド呼び出し、`MTLRenderCommandEncoder` オブジェクトを作成します。

```
let renderEncoder = commandBuffer.makeRenderCommandEncoder(descriptor: renderPassDescriptor)
```

## 3. レンダリングコマンドのさまざまな設定を行う

`MTLRenderCommandEncoder` のさまざまなメソッドを使って、必要な設定を行います。

まずは `setRenderPipelineState(_:)` メソッドを使って、「13.5.5 `MTLRenderPipelineState` を生成する」で作成しておいた `MTLRenderPipelineState` オブジェクトをセットします。

```
renderEncoder.setRenderPipelineState(renderPipeline)
```

そして、「13.5.3 シェーダにデータを渡すためのバッファを準備する」で作成しておいた `MTLBuffer` オブジェクトを、`setVertexBuffer(_:offset:index:)` メソッドを使ってセットします。

```
renderEncoder.setVertexBuffer(vertexBuffer, offset: 0, index: 0)
```

ここで第3引数に渡しているインデックスは、「2. 頂点シェーダ関数の実装」で該当する引数の属性修飾子 `[[ buffer(n) ]]` に与えたインデックスと対応しています。

#### 4. プリミティブを描画する

レンダーコマンドエンコーダによるレンダリングコマンド作成の仕上げとして、「プリミティブ」(点・直線・三角形といった、頂点からなる基本的な図形)の描画を行うための `MTLRenderCommandEncoder` のメソッドを呼びます。

いくつか種類がありますが、ここでは `drawPrimitives(type:vertexStart:vertexCount:)` を使用しましょう。

```
func drawPrimitives(type primitiveType: MTLPrimitiveType, vertexStart: Int, vertexCount: Int)
```

引数にはそれぞれ

- プリミティブのタイプ (`MTLPrimitiveType`)
- 描画する頂点の先頭インデックス
- 描画する頂点の数

を指定します。

`MTLPrimitiveType` は頂点座標の配列をどのように使用して図形として描画するかを指定するものです。

たとえば今回の例では、頂点座標データとして4つの点を用意しました。それぞれ頂点 A(-1, -1, 0, 1)、頂点 B(1, -1, 0, 1)、頂点 C(-1, 1, 0, 1)、頂点 D(1, 1, 0, 1) とすると、選ぶ `MTLPrimitiveType` の値によって描画される図形は次のように変わってきます。

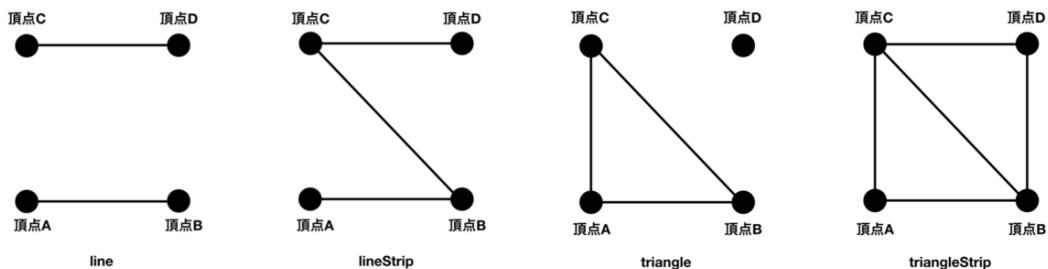


図 13.6: 4 つの頂点と `MTLPrimitiveType`

TODO: 再作図: 本書全体に共通する作図のトーンで再作図していただければと。

本節ではスクリーン全体に描画したいので、4つの頂点を矩形として描画するよう、`triangleStrip`を使用します。

```
renderEncoder.drawPrimitives(type: .triangleStrip, vertexStart: 0, vertexCount: 4)
```

あとは他のコマンドエンコーダを利用する場合と同様、コマンドのエンコードを完了し、表示するドローアブルを登録し、コマンドバッファをコミットしてキューに入れます。

以上の実装で、図 13.6 のように画面全体をシェーダで赤色に塗る実装は完成です。

## 13.6 Metal 入門その3 - シェーダでテクスチャを描画する

「入門その2」では、レンダーコマンドエンコーダを用いて、Metalシェーダにより「画面を一色に塗る」実装を行いました。次は同様にレンダーコマンドエンコーダとシェーダを用いつつ、「テクスチャを描画する」実装を行ってみましょう。

シェーダでテクスチャを扱えるようになるということは、カメラから取得したデータをシェーダで処理できるようになるということでもありますし、GPUの並列演算能力を活かした画像処理を行えるようになることもあります。ここから「Metalで実現できること」が一気に広がるので、ぜひとも押さえておきましょう。

基本的には前節で行った実装を踏襲しつつ、テクスチャを扱うための実装が少々加わります。本節では前節の実装との差分を中心に解説します。

(サンプルコード: 03\_MetalShaderImageRender)

### 13.6.1 テクスチャを扱うシェーダの実装

シェーダでテクスチャを扱うにあたり、頂点シェーダの出力かつフラグメントシェーダの入力となる構造体の定義に、これまでの `[[ position ]]` で修飾された 3 次元空間における座標値に加えて、「テクスチャ内における座標」(2 次元) を保持するための要素も追加します。

```
struct ColorInOut
{
 float4 position [[position]];
 float2 texCoords;
};
```

そして、頂点シェーダの関数を次のように実装します。

```
{
 ColorInOut out;
 out.position = positions[vid];
 out.texCoords = texCoords[vid];
 return out;
}
```

第2引数で頂点のテクスチャ内座標を受け取っている点が前節の実装との違いです。特に処理はせず `ColorInOut` 構造体にセットして返している点は前節と同様です。

「テクスチャを描画する」 フラグメントシェーダの実装は次のようにになります。

```
fragment float4 fragmentShader(ColorInOut in [[stage_in]],
 texture2d<float> texture [[texture(0)]])
{
 constexpr sampler colorSampler; // ……(1)
 float4 color = texture.sample(colorSampler, in.texCoords); // ……(2)
 return color;
}
```

第2引数の `texture2d<T>` 型は2次元テクスチャを表し、Tはそれぞれの色値の型を示します。Tとしてはここで用いている `float` の他に、`half`、`short`、`ushort`、`int`、`uint` が指定できます。

そして同引数の修飾子 `[[ texture(n) ]]` に示した引数には、CPU プログラム側で `MTLRenderCommandEncoder` の `setTexture(_:index:)` メソッドでセットした `MTLTexture` オブジェクトの内容が入ってきます<sup>\*17</sup>。

テクスチャは常にデバイスアドレス空間に割り当てられるため、テクスチャ型の引数にはデバイスアドレス修飾子を指定する必要はありません<sup>\*18</sup>。

(1) で新たに `sampler` という型の変数が登場しています。これは「サンプラー」と呼ばれ、どのようにテクスチャデータにアクセスするかを定義します。ここでは特に何も指定せず、デフォルトのまま使用します。

サンプラーは引数から渡すこともできますが、今回のようにシェーダプログラム内で初期化する場合は `constexpr` と共に宣言する必要があります<sup>\*19</sup>。

そして、`texture2d` 型のテクスチャは次のメンバ関数を持ち、第1引数に `sampler` オブジェクトを、第2引数にテクスチャ内座標を渡して、該当するピクセルのデータを取得することができます。第3引数のオフセットは省略可能です。

```
Tv sample(sampler s, float2 coord, int2 offset = int2(0)) const
```

<sup>\*17</sup> MSL 言語仕様書の "4.3.1 AttributeQualifiersToLocateBuffers,TexturesandSamplers" より。

<sup>\*18</sup> MSL 言語仕様書の "4.2.1 device Address Space" より。

<sup>\*19</sup> MSL 言語仕様書の "2.6 Samplers" より。`constexpr` は C++ の機能で、その変数がコンパイル時定数であることを示します。

この関数を使用しているのが(2)です。引数から得たテクスチャデータから、`sampler`オブジェクトと、`ColorInOut`構造体に入っているテクスチャ内座標を使用して、該当するピクセルの色値を取得しています。

そして最後に、そのテクスチャから得た色値をそのまま出力しています。つまりこのフラグメントシェーダは単にテクスチャからピクセルの色値を取り出して画面の当該ピクセルの色として出力しており、その処理が画面全体のピクセルに行われる所以、テクスチャ（画像）が画面に描画される、というわけです。

### ■コラム：サンプラーを引数から渡す場合

サンプラーオブジェクトをCPUプログラム側から作成してシェーダ引数に渡す場合、`MTLSamplerDescriptor`を作成して`MTLSamplerState`を初期化し、

```
private var sampler: MTLSamplerState!
```

```
let samplerDescriptor = MTLSamplerDescriptor()
sampler = device.makeSamplerState(descriptor: samplerDescriptor)!
```

あとは`MTLRenderCommandEncoder`の`setFragmentSamplerState(_:index:)`メソッドを呼ぶことで、該当するインデックスのシェーダ関数の引数にバインドされます。

```
renderEncoder.setFragmentSamplerState(sampler, index: 0)
```

### 13.6.2 テクスチャ内座標データをシェーダに渡す

前節の赤一色に塗る例では頂点の座標データを次のように用意しました。

```
let vertexData: [Float] = [
 -1, -1, 0, 1,
 1, -1, 0, 1,
```

```
-1, 1, 0, 1,
1, 1, 0, 1
]
```

4つの頂点で、x-y 平面でそれぞれ-1～1 の値をとる矩形を構成します。座標の値は左下から右上に向かって大きくなります。

本節ではテクスチャを扱うにあたって、各頂点に対応するテクスチャ内座標も用意します。

```
let textureCoordinateData: [Float] = [
 0, 1,
 1, 1,
 0, 0,
 1, 0
]
```

頂点座標 (-1, -1, 0, 1) に対して (0, 1) が対応していることからわかる通り、こちらは 0～1 の値を取り、左上を原点に、右下に向かって大きくなります。この座標に従ってシェーダ内でサンプラーを用いてテクスチャ内座標に対応するピクセルデータにアクセスすることになります。

このテクスチャ内座標をシェーダに渡すために、`MTLBuffer` を用意します。この実装は「13.5.3 シェーダにデータを渡すためのバッファを準備する」で行った頂点座標のデータを渡す処理と同様なので割愛します。

そして用意したバッファを、`MTLRenderCommandEncoder` の `setVertexBuffer(_:offset:index:)` メソッドでセットします。

```
renderEncoder.setVertexBuffer(texCoordBuffer, offset: 0, index: 1)
```

これも前節の「3. レンダリングコマンドのさまざまな設定を行う」で頂点座標データのバッファをセットした際の手順と同様なのですが、第3引数に渡すインデックスだけが異なります。

ここではインデックス 0 では頂点座標データを渡しているので、テクスチャ内座標データのインデックスは 1 としています。

ここで、頂点シェーダの関数の定義を再掲すると、

```
vertex ColorInOut vertexShader(device float4 *positions [[buffer(0)]],
 device float2 *texCoords [[buffer(1)]],
 uint vid [[vertex_id]])
```

`setVertexBuffer(_:offset:index:)` の第3引数に指定したインデックスと、シェーダ関数の `[[ buffer(n) ]]` 修飾子を持つ引数とが対応していることがわかります。

### 13.6.3 テクスチャをシェーダに渡す

`MTLRenderCommandEncoder` の `setFragmentTexture(_:index:)` メソッドを用いて、描画したいテクスチャの `MTLTexture` オブジェクトもセットします。

```
renderEncoder.setFragmentTexture(texture, index: 0)
```

ここで、フラグメントシェーダ関数の定義を再掲すると、

```
fragment float4 fragmentShader(ColorInOut in [[stage_in]],
 texture2d<float> texture [[texture(0)]])
```

`setFragmentTexture(_:index:)` メソッドの第2引数に渡したインデックスと、シェーダ関数の `[[ texture(n) ]]` 修飾子を持つ引数とが対応していることがわかります。

### 13.6.4 ピクセルフォーマットを合わせる

`MTLRenderPipelineDescriptor` でカラーattachementの設定を行う際に、ピクセルフォーマットを表示したいテクスチャと合わせておきます。

```
descriptor.colorAttachments[0].pixelFormat = texture.pixelFormat
```

実装の差分としては以上です。サンプルを実行すると、次のように画像がシェーダの処理によって画面に描画されます。



### 13.7 ARKit+Metal その 1 - マテリアルを Metal で描画する

ここからがいよいよ iOS 11 からの話になります。第 2 章で解説した iOS 11 の新機能 ARKit と、Metal を組み合わせた実装を解説します。

まずは手軽な組み合わせ方法として、**ARKit** を用いて現実空間の水平面上に設置したノードのマテリアルを、**Metal** を用いて描画する方法を解説します。

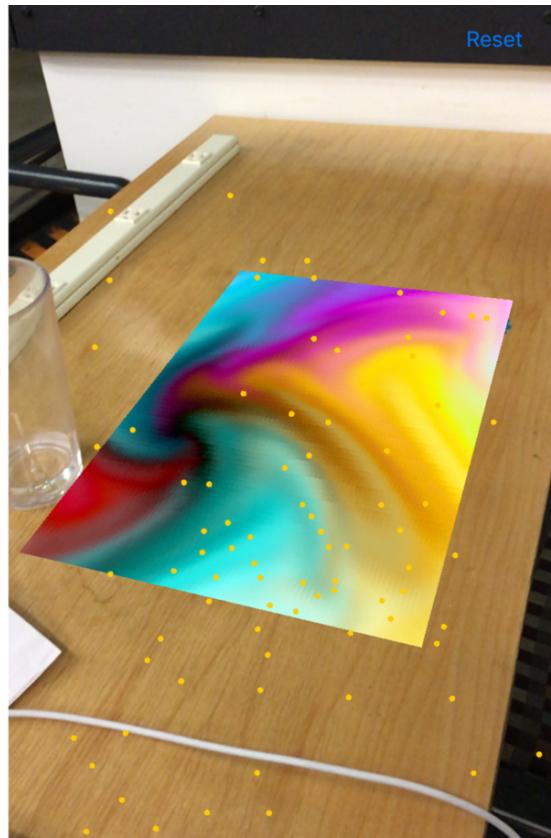


図 13.7: SCNNode のマテリアルの描画に Metal を利用

Metal の基礎はこれまでに、ARKit については第 2 章で解説したので、ここでは連携のポイントに絞って解説します。

(サンプルコード: 04\_ARMetal1)

### 13.7.1 SCNProgram

この実装のポイントとなるのは、SceneKit の `SCNProgram` というクラスです。これを用いることで、`SCNNode` が持つジオメトリ (`SCNGeometry`) のマテリアル (`SCNMaterial`) を、Metal で描画できます。

たとえばマテリアルに色で塗りたい場合は、`SCNMaterial` の `diffuse` 等の `SCNMaterialProperty` 型プロパティに、次のように `UIColor` 等の色を `contents` として指定することで実現できます。

```
material.diffuse.contents = UIColor.red
```

ここで、`SCNMaterial` の `program` プロパティ<sup>\*20</sup>に `SCNProgram` オブジェクトを渡すと、

<sup>\*20</sup> 正確には、`SCNMaterial` が準拠している `SCNShadable` プロトコルのプロパティです。

```
var program: SCNProgram?
```

当該マテリアルのレンダリングをシェーダプログラムによって行うことができます。  
具体的な実装手順は次の通りです。

```
// SCNProgram を初期化
let program = SCNProgram()

// 使用する頂点シェーダの関数名を指定
program.vertexFunctionName = "vertexShader"

// 使用するフラグメントシェーダの関数名を指定
program.fragmentFunctionName = "fragmentShader"

// SCNMaterial(SCNShadable) の program プロパティにセット
node.geometry?.firstMaterial?.program = program
```

`vertexFunctionName`、`fragmentFunctionName` には、アプリケーションバンドル内にある Metal シェーダファイル内に記述されている各シェーダの関数名が指定できます。

### 13.7.2 SCNProgram を利用する場合のシェーダの実装

`SCNProgram` から利用するシェーダを実装する場合は、`<metal_stdlib>` に加え、`<SceneKit/scn_metal>` もインクルードします。

```
#include <SceneKit/scn_metal>
```

#### SCNProgram で用いる頂点シェーダ

SceneKit 自体が 3D レンダリングエンジンであり頂点データの取り扱いの多くを任せられるため、頂点シェーダの書き方はかなり変わってきます。

まず、次のような構造体を定義します。

```
struct VertexInput {
 float3 position [[attribute(SCNVertexSemanticPosition)]];
 float2 texCoords [[attribute(SCNVertexSemanticTexCoord0)]];
};
```

ここでは頂点座標と、テクスチャ内座標を要素として持たせています。

それぞれの要素が持つ `[[ attribute(index) ]]` という修飾子<sup>\*21</sup>のインデックスとして指定されている定数は、"SceneKit Vertex Attribute Qualifiers" (SceneKit 頂点属性修飾子) と呼ばれます<sup>\*22</sup>。

`SCNVertexSemanticPosition` はジオメトリソースによって与えられる頂点の座標を、`SCNVertexSemanticTexCoord0` は第1ジオメトリソースによって与えられるテクスチャ内座標を表します<sup>\*23</sup>。

このようにして定義した構造体を、頂点シェーダで次のように `[[ stage_in ]]` 修飾子を付与した引数の型に指定することで、

```
vertex ColorInOut vertexShader(VertexInput in [[stage_in]],
```

SceneKit は、暗黙的に（明示的にバッファ等を用意してセットするといったことをしなくても）`SCNNode` から必要なデータを頂点シェーダに渡してくれるようになります。

また、SceneKit から「ノード毎の情報」を受け取ることができます。たとえばモデル・ビュー・プロジェクトション変換行列を表す `modelViewProjectionTransform` という名前の要素を持つ構造体を定義し、

```
struct NodeBuffer {
 float4x4 modelViewProjectionTransform;
};
```

次のように、`[[ buffer(n) ]]` 属性修飾子を持つ引数の型に指定することで、この引数を通して、SceneKit からノードの情報（ここでは `modelViewProjectionTransform`）が得られるようになります。

```
vertex ColorInOut vertexShader(VertexInput in [[stage_in]],
 constant NodeBuffer& scn_node [[buffer(0)]])
```

シェーダで受け取れるノード情報としては、`modelViewProjectionTransform` 以外に、次のようなものがあります<sup>\*24</sup>。

- `float4x4 modelTransform;`

<sup>\*21</sup> MSL 言語仕様書の"4.3.3 Attribute Qualifiers to Locate Per-Vertex Inputs"より。

<sup>\*22</sup> SCNProgram の API リファレンス内にある"Table 1 SceneKit Vertex Attribute Qualifiers for Metal Shaders" より。

<sup>\*23</sup> SCNProgram の API リファレンス内にある"Table 1 SceneKit Vertex Attribute Qualifiers for Metal Shaders" より。

<sup>\*24</sup> SCNProgram の API リファレンスページの"Per-Node Data"に一覧があります。

- float4x4 inverseModelTransform;
- float4x4 modelViewTransform;
- float4x4 inverseModelViewTransform;
- float4x4 normalTransform;
- float4x4 inverseModelViewProjectionTransform;
- float2x3 boundingBox;
- float2x3 worldBoundingBox;

SCNProgram で用いる フラグメントシェーダ

SCNProgram で用いる シェーダに CPU プログラム側から 値を渡す場合は、MTLRenderCommandEncoder を用いた前節の方法とは少し作法が違います。

たとえば Float 型の `time` という値を渡したい場合は、Data 型にして、シェーダの該当する引数名 "time" をキーとするプロパティの値としてセット<sup>\*25</sup> します。

```
// Data 型にする
let timeData = Data(bytes: &time, count: MemoryLayout<Float>.size)

// シェーダの引数に対応するキーの値としてセット
material.setValue(timeData, forKey: "time")
```

こうすることで、シェーダ側で次のような関数を定義した場合に、

```
fragment half4 fragmentShader(ColorInOut in [[stage_in]],
 constant float& time [[buffer(0)]])
```

この フラグメントシェーダの 第2引数名が `time` なので、ここに先ほどセットした 値が 入ってきます。

SceneKit のノードのマテリアルを Metal で描画するためのポイントとなる部分の解説は以上です。これで ARKit と Metal を組み合わせた、図 13.9 のような表現が可能になりました。

### 13.8 ARKit+Metal その2 - Metal による ARKit のカスタムレンダリング

これまでに解説したことの組み合わせで、もっと AR らしく 現実空間と仮想オブジェクトを融合させた 図 13.10 のような表現も可能です。

<sup>\*25</sup> ここで用いる `setValue(_:_forKey:)` は、`NSObject` のメソッドです。

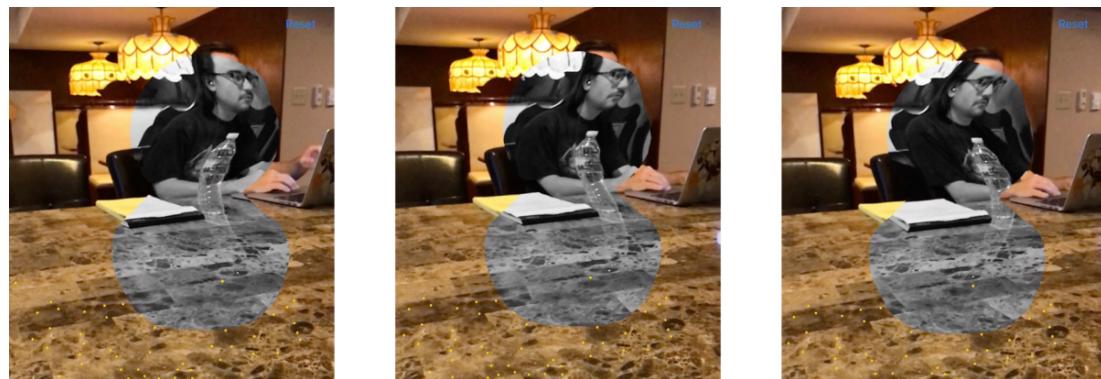


図 13.8: 仮想オブジェクトの描画に現実空間を反映

実装方針としては次のようにになります。

1. ARKit によって検出された平面上に仮想オブジェクトを設置。仮想オブジェクトのマテリアルを特定の色で塗る
2. 1 のシーンのスナップショットを取得する
3. カメラから取得した画像データと、2 の画像データ、経過時間をフラグメントシェーダに渡し、最終的な出力を計算する (→ MTKView 上に描画される)

「シーンのスナップショット」の取得には、ARSCNView の親クラスである SCNView のメソッドが使用できます。

```
func snapshot() -> UIImage
```

他の要素技術としてはこれまでの節で既に解説したものばかりなので、詳細な実装解説はここでは割愛します。ぜひサンプルコードを参照してみてください。

(サンプルコード: 05\_ARMetal2)

## 13.9 Metal 2

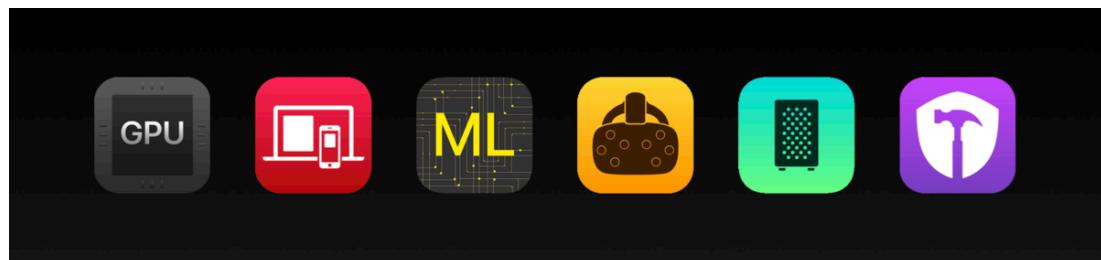
WWDC17 で Metal 2 が発表されました。ロゴも刷新され、大きく生まれ変わった印象がありますが、iOS SDK のフレームワークの観点からみると、Metal2.framework や MetalKit2.framework といったものが登場したわけではありません。Swift に 1 や 2 があるのと同様、大きくバージョンアップしたという意味での 2 です。



図 13.9: Metal 旧ロゴ（右）と、Metal 2 のロゴ（左）

### 13.9.1 6つのキーフィーチャー

WWDC の"Platform State of the Union"では、6 つのアイコンを象徴として Metal 2 の新機能が紹介されました。



それぞれ次のような機能として紹介されました。

- GPU ドリブン・レンダリング
  - CPU 負荷を減らし、GPU をもっと効率的に働かせるための新機能群
  - "Argument Buffers"がその代表例
- macOS/iOS での機能の統合
  - たとえば iOS には以前からあった MTLHeap が macOS でも利用できるように
- 機械学習関連 API の拡張
  - Metal Performance Shaders (以下 MPS) フレームワークの CNN(Convolutional Neural Network) API の追加
  - 新フレームワーク Core ML も内部では MPS を利用しており、Metal 2 による GPU ア

クセラレーションの恩恵を自動的に受けられる

- VR サポート
  - 360°動画編集と 3D コンテンツ作成が Metal により最適化され、Mac での VR コンテンツ制作が可能に
- 外部 GPU のサポート
- ツール類の改善
  - 素早いデバッグや解析、GPU パフォーマンス最適化を可能とするための最も要望の多かった機能群を追加した
  - Xcode と Instruments で利用可能

また最新のチップである A11 Bionic chip 上では、Metal 2 はさらなるパフォーマンスと可用性を発揮し、A10 と比較しても最大 2 倍の数値計算および画像処理速度を行えるとされています<sup>\*26</sup>。

### 13.9.2 Argument Buffers

Metal 2 の新機能の中でも、Metal 利用ケース全般において影響があり、恩恵も大きいのが Argument Buffers です。本項ではその概要・実装方法を解説します。

(サンプルコード: 06\_ARMetalArgumentBuffers)

#### Argument Buffers の概要

Argument Buffer (以下 AB) は、バッファやテクスチャ、サンプラーといったリソースをひとまとめにしてシェーディング関数の引数に渡せるようにするしくみで、CPU 負荷を減らし、アプリケーションのパフォーマンスを改善します。

従来の方法では、レンダリンググループの中で GPU 処理に必要なリソースをそれぞれ API を呼んでセットする必要がありました。必要なバッファをセットする API を呼び、テクスチャをセットする API を呼び、サンプラーをセットする API を呼び、そしてやっと描画 API を呼べたわけです。

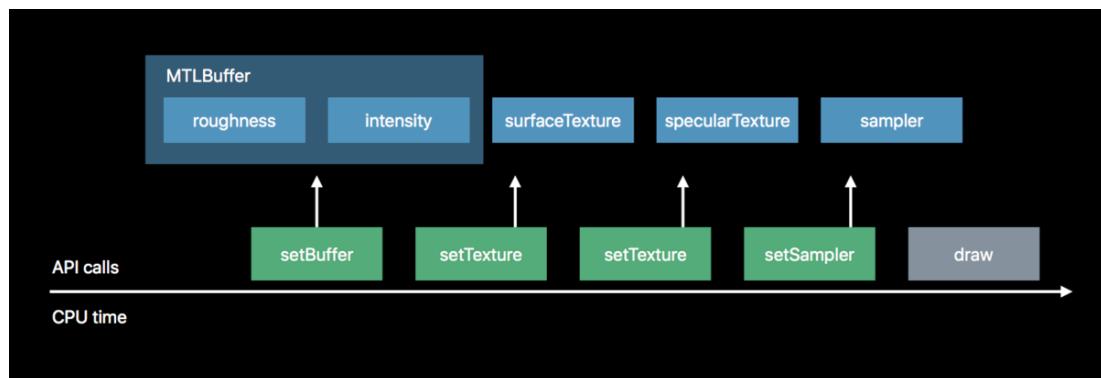


図 13.10: 従来の引数モデル

TODO: 再作図: - 上段の青い四角内にあるテキストを以下のように変更してください（緑の四角

<sup>\*26</sup> Metal 2 on A11 - Overview (<https://developer.apple.com/videos/play/fall2017/602/>)

内のテキストはそのままで) - roughness -> データ 1 - intensity -> データ 2 - surfaceTexture -> テクスチャ 1 - specularTexture -> テクスチャ 2 - sampler -> サンプラー- API calls は setBuffer とかの緑の四角群と中心が合う高さにしてください (つまり横長の矢印に対して"CPU time"と線対称に置く必要はないということです)

上の図の通り、従来の引数モデルでは、バッファ毎、テクスチャ毎、サンプラー毎に API コールが必要で、つまりレンダリングする必要のあるオブジェクトの数に比例して API コール、つまり CPU の負荷が増えます。そして前述したようにレンダリングループの中で行う必要があり、毎フレームの負荷となるため、実質的に画面上に表示するオブジェクトの数の制約にもなっていました。

これが AB を使用することにより、ひとまとめにしたバッファひとつに対して 1 回だけ API を呼ぶだけで済むようになります。

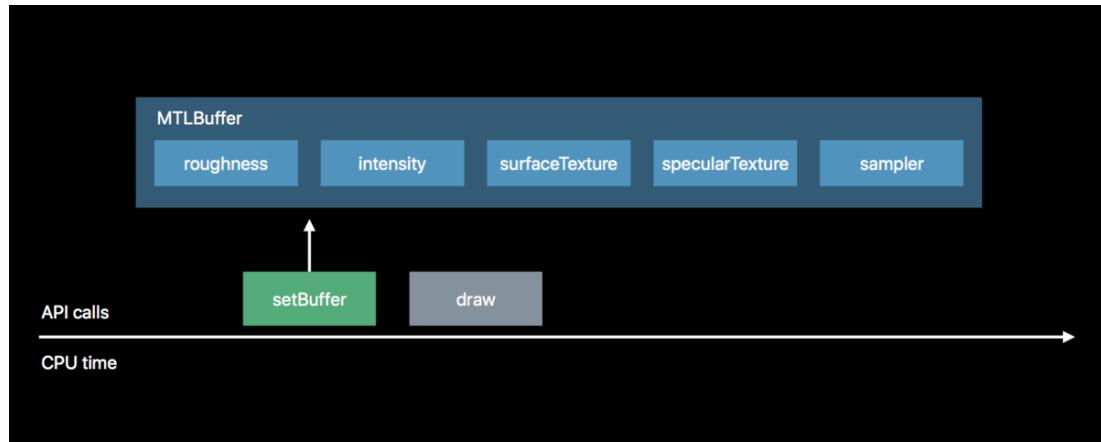


図 13.11: Argument Buffers

TODO: 再作図: - 上段の青い四角内にあるテキストを以下のように変更してください (緑の四角内のテキストはそのまま) - roughness -> データ 1 - intensity -> データ 2 - surfaceTexture -> テクスチャ 1 - specularTexture -> テクスチャ 2 - sampler -> サンプラー- API calls は setBuffer とかの緑の四角群と中心が合う高さにしてください (つまり横長の矢印に対して"CPU time"と線対称に置く必要はないということです)

これにより、GPU に渡すリソース数に応じて増えていた CPU 負荷が、劇的に軽減される、というわけです。

次の図は、WWDC17 の "Introducing Metal 2" セッションで紹介されていたベンチマーク結果です。利用するリソースの数によってどのように処理時間が変わるか、従来の方法と、AB を用いた場合の計測結果が示されています。

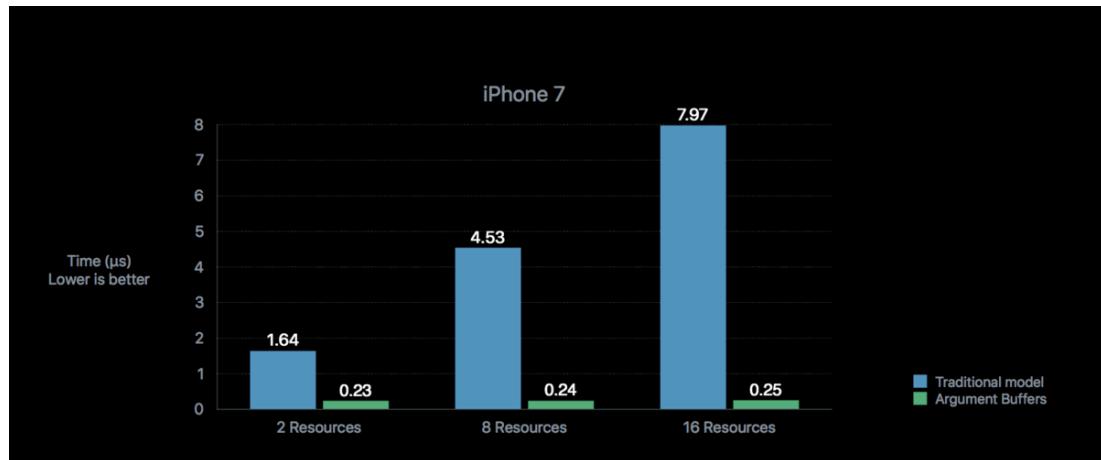


図 13.12: リソース数と処理時間の関係

TODO: 再作図: - 各テキストを以下のように変更してください (指示のないテキストはそのまままで) - Lower is better -> ※低いほうが良い- Traditional model -> 従来モデル

この図によると、iPhone 7 で 16 のリソースを利用した場合には、AB を用いることでなんと 18 倍のパフォーマンス改善が見られたようです。

### Argument Buffers の実装

WWDC のセッションで、利点のひとつに「簡単に使える」ことが挙げられていた程に、AB の実装は簡単です。

まず、シェーダ側で、次のように AB の型としての構造体を定義します。

```
struct ShaderInputs {
 texture2d<float> snapshotTexture;
 texture2d<float> cameraTexture;
 sampler textureSampler;
 float time;
};
```

ここでは `texture2d<float>` 型のテクスチャ 2 つ、`sampler` 型のサンプラー 1 つ、`float` 型のデータ 1 つを要素として持たせています。

AB は以下の種類を要素として含めることができます。

- `half` や `float` 等のスカラ型
- `half4` や `float4x4` のベクトル・行列型
- 基本データ型（上記 2 タイプ）の配列や構造体
- バッファのポインタ
- テクスチャデータ型とその配列
- サンプラー型とその配列

そして、用意した AB を表す構造体を、それを利用したいシェーダ関数で、引数の型に指定します。その際、属性修飾子 `[[ buffer(n) ]]` を付けます。

```
fragment float4 fragmentShader(ColorInOut in [[stage_in]],
 constant ShaderInputs &inputs [[buffer(0)]])
```

ここからは CPU 側のプログラムの作業です。

AB を作成するためのエンコーダ、`MTLArgumentEncoder` オブジェクトを保持するプロパティを用意しておきます。

```
private var argEncoder: MTLArgumentEncoder!
```

AB を利用するシェーダ関数の `MTLFunction` オブジェクトから、`makeArgumentEncoder(bufferIndex:)` を呼び `MTLArgumentEncoder` を初期化します。

```
argEncoder = fragmentFunction.makeArgumentEncoder(bufferIndex: 0)
```

引数のインデックスとしてはこの AB を渡したいシェーダ関数の引数の修飾子 `[[ buffer(n) ]]` のインデックスと一致するように指定します。

ちなみに最もしここでインデックスが合致していないと、実行時にエラーとなります。

```
failed assertion 'Function xxxx does not have a buffer argument with buffer index x'
```

次に、AB を表す `MTLBuffer` オブジェクトを用意します。これは AB 独自のものではなく、これまでに扱ってきたバッファと同様のものです。オブジェクトの生成方法も同じで、`MTLDevice` の `makeBuffer(length:options:)` メソッドを呼ぶのですが、その際に引数に渡すバッファのサイズは、先ほど作成した `MTLArgumentEncoder` オブジェクトの `encodedLength` プロパティから取得できます。

```
let argBuffer = device.makeBuffer(length: argEncoder.encodedLength, options: [])!
```

生成した `MTLBuffer` オブジェクトを、`MTLArgumentEncoder` の `setArgumentBuffer(_:offset:)` メソッドを用いてセットします。

```
argEncoder.setArgumentBuffer(argBuffer, offset: 0)
```

そして、当該 AB で用いるテクスチャの `MTLTexture` オブジェクトや、サンプラーの `MTLSamplerState` オブジェクト、変数のポインタ等を `MTLArgumentEncoder` の `setTexture(_:index:)` や `setSamplerState(_:index:)` といったメソッドを利用してセットします。

```
argEncoder.setTexture(textureSnapshot, index: 0)
argEncoder.setTexture(textureCamera, index: 1)
argEncoder.setSamplerState(sampler, index: 2)
argEncoder.constantData(at: 3).storeBytes(of: time, as: Float.self)
```

ここで引数に指定する各インデックスは、シェーダ側で定義した構造体の要素のインデックスに合わせます。

`MTLArgumentEncoder` は（コマンドエンコーダとは違い）再利用できるので、ここまで処理で動的に変わるものがないのであれば、毎フレームのレンダリングの度に行う必要はありません。

従来の方法であれば下記のように `MTLRenderCommandEncoder` のメソッドを用いてテクスチャやサンプラーをセットする必要があり、しかもコマンドエンコーダオブジェクトは再利用できないため、描画ループの中でこれを行う必要がありました。

```
// >>> 従来の方法です。AB では必要ありません。
renderEncoder.setFragmentTexture(textureSnapshot, index: 0)
renderEncoder.setFragmentTexture(textureCamera, index: 1)
renderEncoder.setFragmentSamplerState(sampler, index: 0)
renderEncoder.setFragmentBuffer(timeBuffer, offset: 0, index: 0)
// <<< 従来の方法です。AB では必要ありません。
```

このために、シェーダで利用するリソースの数、そしてフレームレートに比例して CPU 負荷が増大していましたが、AB ではこの必要がなくなりました。

代わりに、`MTLTexture` や `MTLBuffer` といった `MTLResource` プロトコルに準拠したリソース型のオブジェクトを AB で利用する場合は、`MTLRenderCommandEncoder` の `useResource(_:usage:)` メソッドでその利用を宣言しておく必要があります。

```
func useResource(_ resource: MTLResource, usage: MTLResourceUsage)
```

第1引数には利用するリソースオブジェクト、第2引数には利用方法を `MTLResourceUsage` で指定します。

今回の場合はサンプラーでピクセルデータにアクセスするので、`.sample` を指定しておきます。

```
renderEncoder.useResource(snapshotTexture, usage: .sample)
renderEncoder.useResource(cameraTexture, usage: .sample)
```

指定しない場合、次のような実行時エラーとなります。

```
> Execution of the command buffer was aborted due to an error during execution.
```

### 13.10 Metal を動作させるためのハードウェア要件

Metal はハードウェアと密接した API なので、「iOS xx 以上で利用可能」といった iOS のバージョン以外に、「GPU ファミリー」(GPU Family) というハードウェア面からの区分も考慮して利用可否を判断する必要があります。

まずは以下に歴代 iOS デバイスの SoC (System on a Chip) と、それらが属する GPU ファミリーの一覧を示します。

SoC	GPU ファミリー	iOS デバイス
A11	4	iPhone X, iPhone 8 & 8 Plus
A10X	3	iPad Pro 12.9-inch (2nd) & iPad Pro (10.5-inch)
A10	3	iPhone 7 & 7 Plus
A9X	3	iPad (5th), iPad Pro (12.9-inch, 9.7-inch)
A9	3	iPhone 6s, 6s Plus, SE
A8X	2	iPad Air 2
A8	2	iPhone 6 & 6 Plus, iPad mini 4, iPod Touch (6th)
A7	1	iPhone 5s, iPad Air, iPad mini 2 & 3

このハードウェア面からの区分である GPU ファミリーと、iOS のバージョンを加味して定義された Metal の機能セットの区分を示す `MTLFeatureSet` という型があります。

enum 型で、次の case が定義されています。

このフィーチャーセットと、具体的な Metal の対応表が PDF で公開されています。

- Metal Feature Set Tables (<https://developer.apple.com/metal/Metal-Feature-Set-Tables.pdf>)

各フィーチャーセットについてのサポート状況をプログラムから調べるには、`MTLDevice` の `supportsFeatureSet(_:)` メソッドを利用します。

```
func supportsFeatureSet(_ featureSet: MTLFeatureSet) -> Bool
```

表 13.2: SoC と GPU ファミリーの対応、および搭載デバイスの一覧

case	値	@available	ハードウェア/iOS バージョン
iOS_GPUFamily1_v1	0	iOS 8.0	A7/iOS 8
iOS_GPUFamily2_v1	1	iOS 8.0	A8/iOS 8
iOS_GPUFamily1_v2	2	iOS 9.0	A7/iOS 9
iOS_GPUFamily2_v2	3	iOS 9.0	A8/iOS 9
iOS_GPUFamily3_v1	4	iOS 9.0	A9, A10/iOS 9
iOS_GPUFamily1_v3	5	iOS 10.0	A7/iOS 10
iOS_GPUFamily2_v3	6	iOS 10.0	A8/iOS 10
iOS_GPUFamily3_v2	7	iOS 10.0	A9, A10/iOS 10
iOS_GPUFamily1_v4	8	iOS 11.0	A7/iOS 11
iOS_GPUFamily2_v4	9	iOS 11.0	A8/iOS 11
iOS_GPUFamily3_v3	10	iOS 11.0	A9, A10/iOS 11
iOS_GPUFamily4_v1	11	iOS 10.0 <sup>*27</sup>	A11/iOS 11

## 第 14 章

# Audio 関連アップデート

### 14.1 はじめに

本章では、iOS 11 の Audio 関連のアップデートの中から、MusicKit と AirPlay2、AVAudioEngineについて解説します。

MusicKit は Apple Music が提供する膨大な楽曲群の再生などをあなたのアプリから簡単に行える API 群です。iOS 標準のミュージック App における Apple Music に関するほとんどの機能を実装できます。AirPlay2 は AirPlay の機能を改善したもので、特徴としては複数デバイスのサポートと、より安定した再生があります。AVAudioEngine のアップデートでは、任意のタイミングでオーディオレンダリングが実行できるマニュアル・レンダリング機能が追加されました。

MusicKit と AirPlay2 の API 群を眺めると、私たちの生活をさらに音楽で満たしたいという Apple のメッセージを感じますし、2017 年 12 月にリリース予定の HomePod<sup>\*1</sup>を中心としたホームオーディオ関連の充実が予想されます。また Objective-C/Swift のインターフェイスで複雑なオーディオ処理が行える AVAudioEngine 関連も機能拡張が進んでおり、今後ますます充実することでしょう。これらの最新機能を把握して、ご自身のアプリケーションに活かしていきましょう。

### 14.2 MusicKit

#### 14.2.1 MusicKit とは

MusicKit は Apple の説明<sup>\*2</sup>に *Adding music to your app has never been so easy.* とあるようように、Apple Music 上の音楽の再生はもちろん、検索、ユーザーが Apple Music で再生した曲の履歴やチャートの取得などミュージック App の Apple Music 関連機能のほとんどが実装できるものです。iOS には以前から iPod Library Access<sup>\*3</sup>というフレームワークがあり、(Apple Music 登場以前の) ミュージック App が持つ機能とほぼ同じものを作ることができました。MusicKit はその Apple Music 版と言えるもので、つまり iPod Library Access と合わせると、現在のミュージック App とほぼ同じものを作れます。iOS 9.3 時点で Apple Music の曲を再生できるようにな

---

<sup>\*1</sup> <https://www.apple.com/homepod/>

<sup>\*2</sup> <https://developer.apple.com/musickit/>

<sup>\*3</sup> iPhone OS 3.0 で実装されましたが、iPod Library Access のドキュメントは現在削除されています。この名称は使わなくなったのではないかと思われますが、実際には iOS 11 でも使用できます。Media Player フレームワークの機能です。

り、iOS 11 では後述の Apple Music API、Apple Music へ登録する画面が提供されるようになりました。既存の API 群も含めて、Apple Music 関連機能を一通り揃えたものを MusicKit と呼ぶことにしたと考えていいでしょう。

MusicKit と Apple Music（ミュージック App）の機能を比較したのが表 14.1 です。

表 14.1 MusicKit / Apple Music（ミュージック App）比較

	Apple Music	MusicKit
再生	○	○
ダウンロード（ローカルキャッシュ）	○	×
For You	○	○
Radio	○	○
検索	○	○
ライブラリへの追加	○	○
キューの操作	○	○（制限あり）
ジャケット画像の取得・表示	○	○

曲を事前にダウンロードすること以外は、ミュージック App とほぼ同等の機能を実現できます。

MusicKit は API 群の総称で、具体的には次の 3 つの技術で構成されています。

- Apple Music API
- MediaPlayer フレームワーク
- StoreKit

Apple Music API はリクエストに対して JSON レスポンスが返る一般的な Web API です。曲の検索や詳細情報の取得などに使います。MediaPlayer フレームワークは MusicKit では主に曲の再生・停止を担当します。StoreKit は主にアプリ内課金のためのフレームワークですが、MusicKit では Apple Music の登録状況の取得、登録のための機能を担当します。MusicKit で Apple Music の曲を再生するまでのおおまかな流れは次のとおりです。

1. Apple Music API へアクセスするためのトークンを生成する
2. Apple Music API で再生する曲の情報（StoreID）を取得する
3. StoreKit で Apple Music へのアクセス許可をユーザーから得る
4. StoreID を MediaPlayer フレームワークの MPMusicPlayerController に渡して再生する

### 14.2.2 Apple Music API

Apple Music API は、Twitter/GitHub などのクライアントアプリを作ったことがある方にはなじみ深い、一般的な Web API です。Apple Music の情報はすべてこの API を使って取得します。

#### Developer Token

Apple Music API では認証のために Developer Token と Music User Token を使います。曲の情報を取得するのに必要なのは Developer Token のみで、先頭に Bearer とスペースを付けて HTTP リクエストの Authorization ヘッダーにセットして使用します。URLRequest を使う場合の具体例をリスト 14.1 に示します。

リスト 14.1: URLRequest を使う場合の Developer Token の使用例

```
var urlRequest = URLRequest(url: urlComponents.url!)
urlRequest.httpMethod = "GET"
urlRequest.addValue("Bearer \\"(DEVELOPER_TOKEN)"", forHTTPHeaderField: "Authorization")
```

Developer Token によって API を使用できる開発者であるかどうかの認証が行われます。この Token が間違っていたり期限が切れている場合は、HTTP ステータスの 400 番台が返ります。Music User Token は後述しますが、Apple Music で特定のユーザー、つまりアプリケーションを使用するユーザーと関連する情報を取得・変更する場合に使います。まずは Developer Token を生成するための準備を行っていきます。手順は次のとおりです。

1. Music ID の取得
2. Key の作成
3. Key を使って Developer Token を生成

#### Music ID の作成

はじめに Music ID を作ります。Music ID は後ほど作成する **Key** において、どのアプリケーションのものであるかを特定するための ID です。

Apple の Member Center(<https://developer.apple.com/account/>) にログインして、Certificates, Identifiers & Profiles をクリックします（図 14.1）。

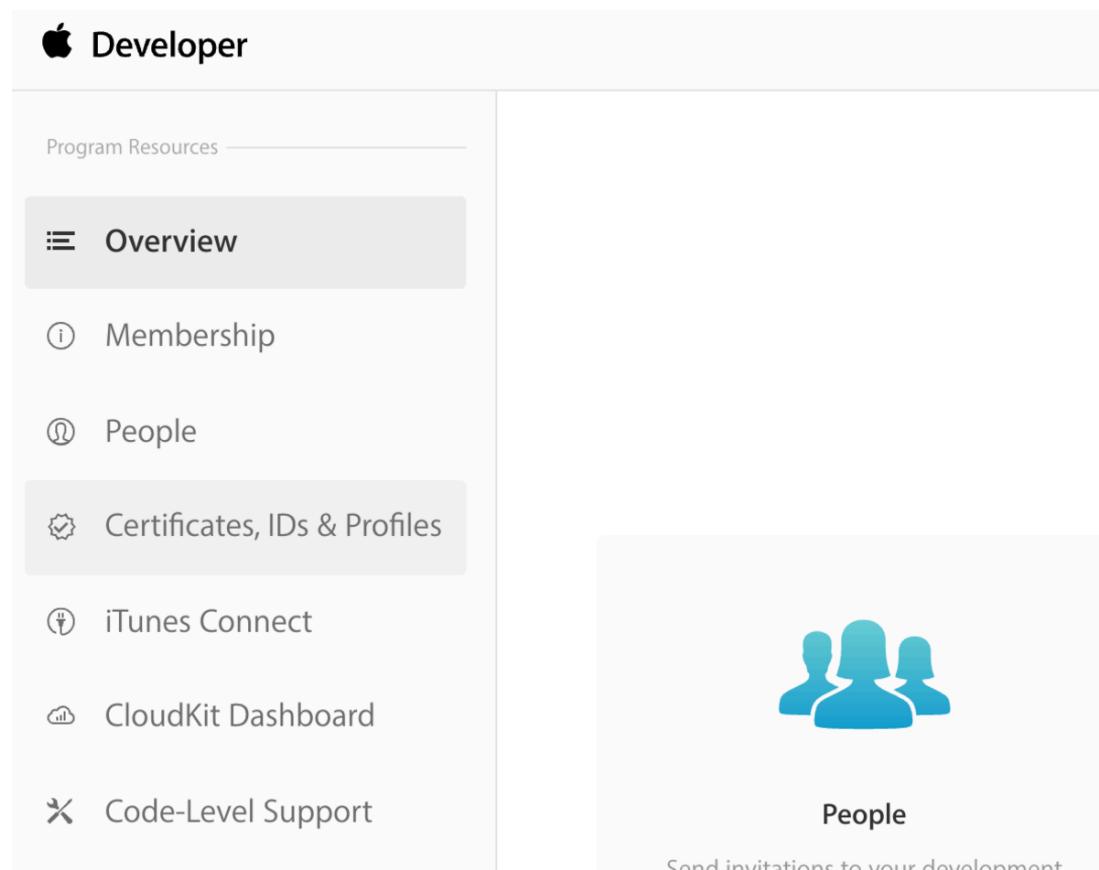


図 14.1: Member Center の Certificates, Identifiers &amp; Profiles

次に左のメニューの Identifiers から Music IDs をクリックして（図 14.3）右上の + ボタンを押して Registering a Music ID のページを開き、Music ID Description と Identifier を入力します（図 14.3）。

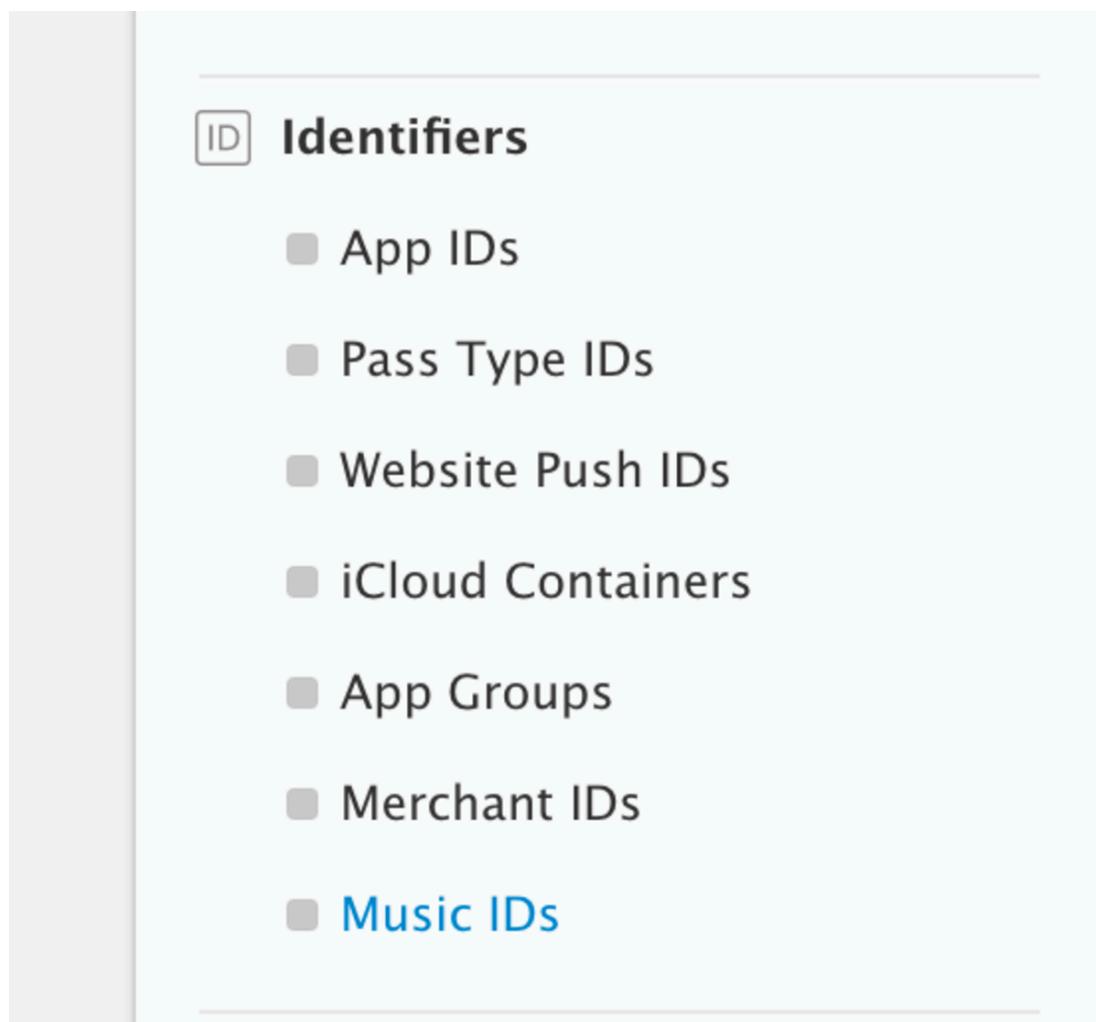


図 14.2: Music ID メニュー

The screenshot shows a user interface for registering a Music ID. At the top left is a large 'ID' icon. To its right, the title 'Registering a Music ID' is displayed. Below this, there is a horizontal line followed by a note: 'For each app that uses the Apple Music API, register a music identifier (Music ID) and then create an associated MusicKit private key.' A second horizontal line follows. Underneath, the section 'Music ID Description' is shown with a 'Description:' label and a text input field containing the placeholder 'Enter a unique identifier for your Music ID, starting with the string 'music''. Below the input field is a note: 'You cannot use special characters such as @, &, \*, ', '''. Another note below says: 'NOTE: Enter the product name of your app that will appear to users.' A third horizontal line follows. The next section is 'Identifier' with a 'ID:' label and a text input field containing the placeholder 'Enter a unique identifier for your Music ID, starting with the string 'music''. Below the input field is a note: 'We recommend using a reverse-domain name style string (i.e., music.com.example).'

図 14.3: Registering a Music ID の画面

Music ID Description にはアプリの説明を、Identifier には `music.` で始まるリバースドメインを入力します。example.com というドメインを持っている開発者の場合、`music.com.example` となります。一意であればよいので、区別するために `music.com.example.applemusic_test` など最後にアプリの名前を入れてもいいでしょう。Description は後で変更できますが、Identifier は変更できませんので、変更したい場合は Music ID を削除するか新規に作成することになります。

### Key の作成

Key は Apple Push Notification Service などでも作成する秘密鍵です。このファイルは他者に渡らないように管理する必要があります。

Key を作っていきましょう。

Certificates, Identifiers & Profiles の左メニューから Keys をクリックします（図 14.4）。

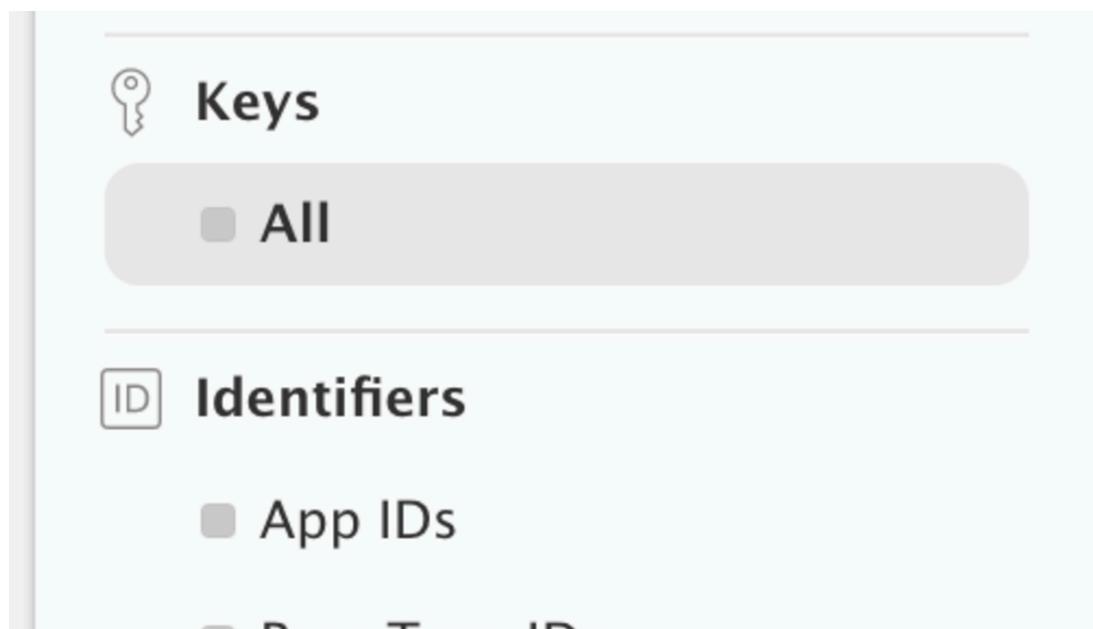


図 14.4: Keys メニュー

右上の + ボタンをクリックして Create a New Key ページを開きます（図 14.5）。

**Create a New Key**

---

**Key Description**

Name:  You cannot use special characters such as @, &, \*, ', "

---

**Key Services**

Enable and configure services for this key. Associating a service with this key allows you to use the service and authenticate communication with the service.

**Enable Service**

**APNs**  
Use the Apple Push Notification service for your notification requests. One key is used for all of your apps. For more information refer to the [Local and Remote Notification Programming Guide](#).

**MusicKit**  
Use the MusicKit service with the Apple Music APIs to access the Apple Music catalog and if authorized by the user, make personalized requests. For more information, refer to the [Apple Music API Reference](#). Configure

**DeviceCheck**  
Use the device check service to enable the persistent storage of two bits of data on a per-device, per-developer basis. One key is used for all of your apps. For more information, refer to the [device check service documentation](#).

図 14.5: Create a New Key 画面

Key Description には何の Key なのか分かりやすい名前を付けておくといいでしょう。下の Key Services からは MusicKit にチェックを付け、右の Configure ボタンをクリックします。Music ID が選択できるので、先ほど作成した Music ID を選んで Continue ボタンをクリックします。確認画面（図 14.7）が出るので、Confirm を押すとダウンロード画面（図 14.7）に変わります。Download ボタンをクリックして AuthKey\_xxxxxxx.p8（以下、AuthKey.p8）というファイルをダウンロードします。このファイルはこの画面でしかダウンロードできないので注意しましょう。図 14.7 中央の **Key ID** は後ほど使うので控えておいてください。

### Confirm your key configuration.

Confirm that you have configured your key with the correct services and information.

Name: Apple Music Test

#### Key Services

Review services for this key. When you associate a service with this key allows you to use the functionality and authenticate with the corresponding service.

##### MusicKit

Use the MusicKit service with the Apple Music APIs to access the Apple Music catalog and if authorized by the user, make personalized requests. For more information, refer to the [Apple Music API Reference](#).

##### MusicKit Configuration Info

Music ID: Apple Music Test API (music.co.rollcake.applemusictest)

Cancel

Back

Confirm

図 14.6: 確認画面

Your key is ready.

#### Download and Back Up

After downloading your key, it cannot be re-downloaded as the server copy is removed. If you are not prepared to download your key at this time, click Done and download it at a later time. Be sure to save a backup of your key in a secure place.

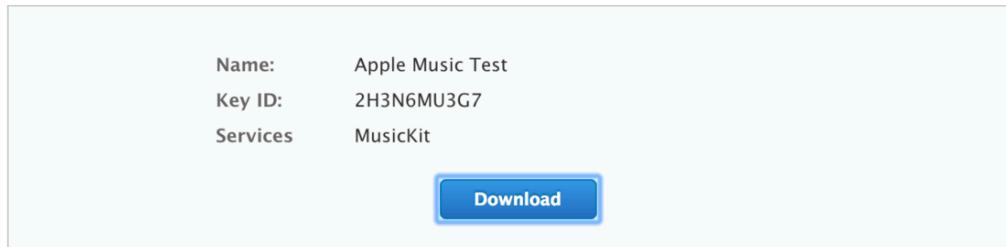


図 14.7: ダウンロード画面

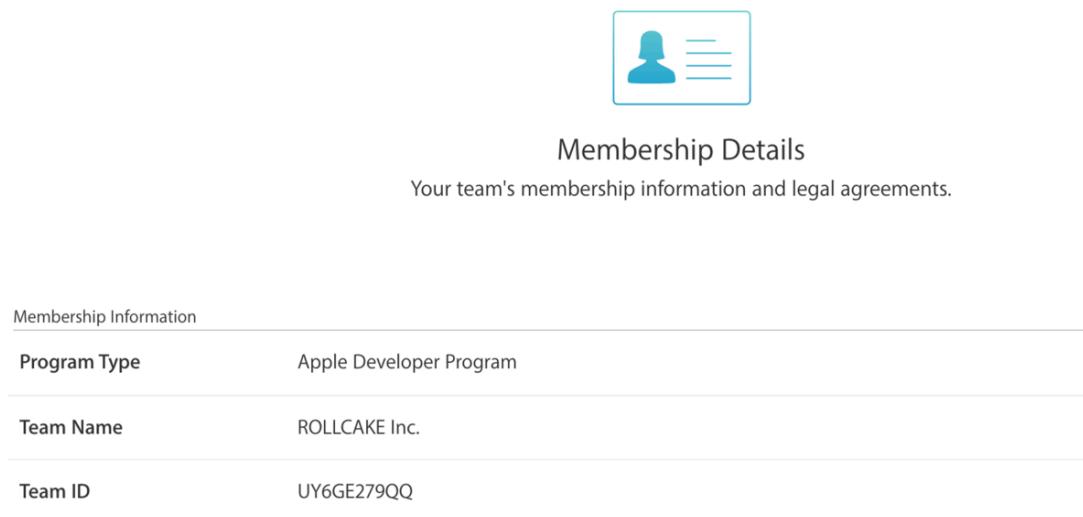
#### Developer Token の生成

Developer Token は JSON Web Token(JWT) 形式で生成します。JWT は公開鍵暗号方式を使って JSON をトーカン化する仕様 (RFC 7519) [^RFC7519] で、主に認証に使われます。

Developer Token の生成には以下の 5 つが必要です。

- AuthKey.p8 内の秘密鍵
- Key ID
- Developer アカウントの Team ID
- 現在日時
- トーカンの期限の日時

Team ID は Member Center の Account -> Membership から確認できます (図 14.8)。



The screenshot shows the 'Membership Details' section of the Apple Developer Program portal. At the top is a blue user icon. Below it is the title 'Membership Details' and a subtitle 'Your team's membership information and legal agreements.' Under the heading 'Membership Information', there are three rows of data:

Program Type	Apple Developer Program
Team Name	ROLLCAKE Inc.
Team ID	UY6GE279QQ

図 14.8: Team ID の確認

これらからリスト 14.2 のような JSON を作り、AuthKey.p8（内の秘密鍵）を使って JWT 形式でトークン化します。

リスト 14.2: JWT でトークン化する JSON の例

```
{
 "alg": "ES256", //Developer Token は ES256 形式のみ対応
 "kid": "ABC123DEFG" //Key ID
}
{
 "iss": "DEF123GHIJ", //Team ID
 "iat": 1437179036, //現在時刻
 "exp" : 1493298100 //トークンの期限の日時の UNIX Time
}
```

JWT を実装するのは大変なので、公開されているライブラリ<sup>\*4</sup>から好みの言語のものを使うとよいでしょう。ここでは、Node.js 用である `apple-music-jwt`<sup>\*5</sup> を使ってみます。JWT の実装ではなく Apple Music の Token 専用になっています。適当なフォルダを作成し、npm でインストールしてコマンドライン版を使います。

```
$ mkdir musikit_token
$ cd musikit_token
$ npm install apple-music-jwt
```

<sup>\*4</sup> <https://jwt.io/>

<sup>\*5</sup> <https://www.npmjs.com/package/apple-music-jwt>

```
$ cd node_modules/apple-music-jwt/bin/
$.apple-music-jwt -k '<Key ID>' -t '<Team ID>' -f 'AuthKey.p8 のパス'
```

実行すると Developer Token が output されます。

## Apple Music API を使う

Miles Davis<sup>\*6</sup>のアルバムを探して、1曲目を再生したい場合を例に説明します。

Miles Davis のアルバムを検索する場合のリクエスト URL は次のとおりです。

<https://api.music.apple.com/v1/catalog/jp/search?types=albums&term=Miles+Davis&limit=1>

URL 先頭の <https://api.music.apple.com/v1/catalog/> までは、どのリクエストを送信する場合も全て共通です。v1 の箇所にはバージョンが入ります。現在は v1 です。次の /jp/ は国を表します。この国を Apple Music API では **Storefront** と呼びます。Storefront はリージョンの意味で使われますが、地域によってヒットチャートが異なり、聞ける曲も違うため、リージョンではなくお店のトップという意味で Storefront という名前になっているものと思われます<sup>\*7</sup>。Storefront を取得する API も用意されています。

残りの `search` とパラメーター部分は「Miles と Davis の両方のキーワードを持つアルバムを 1 件だけ探す」という意味です。

ターミナルで次の curl コマンドを実行してみましょう。

```
DEVELOPER_TOKEN=生成した Developer Token

curl -v -H "Authorization: Bearer $DEVELOPER_TOKEN" \
"https://api.music.apple.com/v1/catalog/jp/search?types=albums&term=Miles+Davis&limit=1"
```

このリクエストを実行すると、次のようなレスポンスが JSON 形式で得られます。実際には改行なしの一行ですが、見やすさのために整形してあります。

```
{
 "results" : {
 "albums" : {
 "href" : "/v1/catalog/jp/search?types=albums&term=Miles+Davis&limit=1",
 "next" : "/v1/catalog/jp/search?types=albums&term=Miles+Davis&offset=1",
 "data" : [
 {
 "id" : "268443092",
 "name" : "Miles Davis - Sketches for My Father"
 }
]
 }
 }
}
```

\*6 ジャズ・ミュージシャン。モードジャズを確立したこと有名。<https://ja.wikipedia.org/wiki/マイルス・デイヴィス>

\*7 ドキュメントには"Apple Music is a worldwide service that operates in many countries and languages. However, the available content varies by region. In Apple Music, a region is called a storefront, and each storefront vends a different music catalog."とあります。

```
"type" : "albums",
"href" : "/v1/catalog/jp/albums/268443092",
"attributes" : {
 "artwork" : {
 "width" : 600,
 "height" : 600,
 "url" : "https://is3-ssl.mzstatic.com/image/thumb/Music/v4/f3/
d5/eb/f3d5eb6b-34f6-69db-1b2c-477c46ee7662/source/{w}x{h}bb.jpg",
 "bgColor" : "161c18",
 ...
 },
 "artistName" : "マイルス・ディヴィス",
 "isSingle" : false,
 "url" : "https://itunes.apple.com/jp/album/Kind-of-Blue/id268443092",
 "isComplete" : true,
 "genreNames" : [
 "ジャズ",
 "ミュージック",
 ...
],
 "trackCount" : 5,
 "releaseDate" : "1959-08-17",
 "name" : "Kind of Blue",
 "copyright" : "Originally released 1959 Sony Music Entertainment Inc.",
 "playParams" : {
 "id" : "268443092",
 "kind" : "album"
 }
}
}
]
}
}
```

types=albums で検索したので、albums が返ってきています。href はリクエストしたエンドポイント、next はページングする場合のエンドポイントです。"data"にアルバムの情報があります。attributes 内の artistName, name, trackCount から、これはマイルス・ディヴィスの Kind of Blue という 5 曲のアルバムであることが分かります。artwork にはアルバムのアートワーク情報があります。url はアートワーク画像の URL です。{w}x{h}bb.jpg 部分の{w}と{h}は、それぞれ画像の横幅と縦幅を意味していて、画像を取得するときにサイズを指定できるようになっています。たとえば 100x100bb.jpg とすると、100 × 100 ピクセルの画像が取得できます。artwork 内の width と height はそれぞれ最大サイズを表しますので、この場合、600 ピクセル以上のサイズは取得できません。他の項目では"playParams"の情報が重要で、id 要素を MediaPlayer フレームワークの MPMusicPlayerController に渡して再生します。以降は playParams の id を StoreID と呼びます。

さて、Kind of Blue の情報を取得できたので、これを元にアルバムを再生することができます。さらにアルバム内の曲=トラックの情報も取得して 1 曲目を再生することにしましょう。

"data"内の"href"を見ると次のように書かれています。

```
"href" : "/v1/catalog/jp/albums/268443092",
```

これがアルバムのエンドポイントです。後ろに tracks を付けるとトラック情報が取得できます。

```
curl -v -H "Authorization: Bearer $DEVELOPER_TOKEN" \
"https://api.music.apple.com/v1/catalog/jp/albums/268443092/tracks"
```

上の curl コマンドを実行した結果は次のとおりです。結果は整形しています。

```
{
 "data" : [
 {
 "id" : "268443097",
 "type" : "songs",
 "href" : "/v1/catalog/jp/songs/268443097",
 "attributes" : {
 ...
 ...albums 取得時と同じ項目...
 ...
 "durationInMillis" : 547983,
 "releaseDate" : "1959-08-17",
 "name" : "So What",
 "playParams" : {
 "id" : "268443097",
 "kind" : "song"
 },
 "trackNumber" : 1,
 "composerName" : "マイルス・デイヴィス"
 },
 "relationships" : {
 "albums" : {
 "data" : [
 {
 "id" : "268443092",
 "type" : "albums",
 "href" : "/v1/catalog/jp/albums/268443092"
 }
],
 "href" : "/v1/catalog/jp/songs/268443097/albums"
 },
 "artists" : {
 "data" : [
 {
 "id" : "44984",
 "type" : "artists",
 "href" : "/v1/catalog/jp/artists/44984"
 }
],
 }
 }
 }
]
}
```

```
 "href" : "/v1/catalog/jp/songs/268443097/artists"
 }
},
{
...以下、track3～track5
]
}
```

albums の場合と同じ部分が多いですが、type が"song"となっていて、attributes では name がトラック名=曲名になっているので、曲名は「So What」だと分かります。また trackNumber と durationInMillis の 2 つの情報から、So What が  $547983 \div (1000 \times 60.0) = 9$  分程度の長さだと分かります。

音楽的な余談ですが、あるアーティスト名を Web で検索したとき最初に出てくるアルバムは、そのアーティストの代表作であることが多いでしょう。著名なアーティストであれば名盤ということになります。実際、Miles Davis の代表作はジャズで最も売れたアルバムであると言われるこの"Kind of Blue"で、「So What」はこのアルバムを象徴するモード・ジャズの名曲です [^m]。名盤の一曲目は必ずと言っていいほど「つかみ」の曲なはずですから（名盤の一曲目がつまらないわけがない）、以上の API を使うだけでも「あるアーティストを代表する曲を再生するアプリ」を作ることができるはずです [^m2]。

[^m]: 和音=コードではなく、モード=スケールを中心として曲を構成・演奏するジャズ。"So What"は D ドリアンと Eb ドリアンという二つのスケールで作られています。[^m2]: albums ではなく songs で検索したときの 1 曲目の方がよいかもしれません。

### 14.2.3 StoreKit

曲を再生するには、アプリから Apple Music へのアクセスをユーザーに許可してもらう必要があります。曲は写真や住所と同じでユーザーのプライベートなデータであるからです。MusicKit ではアクセス許可関係をはじめとする次の機能を StoreKit フレームワークが担当します。

- Apple Music へのアクセス許可
- Apple Music へ登録するための専用画面
- Apple Music のどの機能が使えるかを表すケイパビリティ

具体的には、アクセス許可とケイパビリティは SKCloudServiceController クラス、Apple Music へ登録する専用画面は SKCloudServiceSetupViewController クラスが担当します。

#### アクセス許可

ユーザーにアクセス許可をもらうためには、まず info.plist に NSAppleMusicUsageDescription の項目を追加し、Apple Music へアクセス理由を記述しておく必要があります ((図 14.9))。

Main storyboard file base name	String	Main
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)
▶ Supported interface orientations (iPad)	Array	(4 items)
Privacy - Media Library Usage Description	String	◊ Apple Musicを使うためにアクセスします

図 14.9: Info.plist の記述例

SKCloudServiceController.requestAuthorization を実行すると、NSAppleMusicUsageDescription で設定したメッセージを含むアラートが表示されます（図 14.10）。

```
SKCloudServiceController.requestAuthorization { (status) in
 switch status {
 case .authorized:
 print("authorized")
 case .denied, .restricted, .notDetermined:
 print("not authorized")
 }
}
```

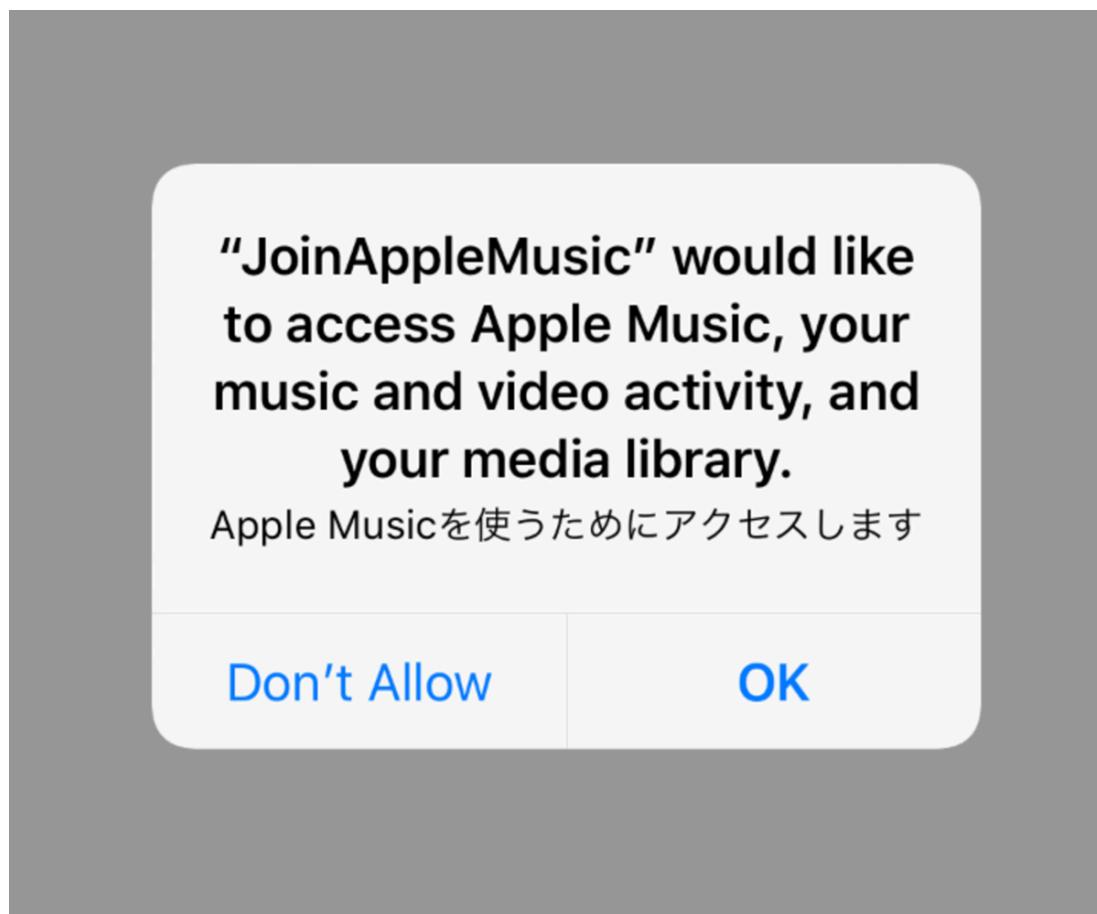


図 14.10: アクセス許可を求めるアラート

アラートの「許可」をタップすると `status` が`.authorized`になります。Apple Music catalog tracks にアクセスできるようになります。その他の場合の `status` が表す状態は次のとおりです。`.authorized`以外はアクセスできません。

表 14.1: SKCloudServiceAuthorizationStatus 一覧

status	状態
notDetermined	許可、拒否がまだ決定されていない
denied	拒否されている
restricted	設定によって制限されている
authorized	許可されている

### ケイパビリティ

アクセス許可とは別に、アプリケーションが利用できる機能を表すケイパビリティがあります。MusicKit におけるケイパビリティは次の 3 種類があります。

表 14.2: SKCloudServiceCapability の一覧

名前	説明
musicCatalogPlayback	Apple Music の曲を再生することができる
musicCatalogSubscriptionEligible	Apple Music への登録が可能な状態にある
addToCloudMusicLibrary	iCloud ミュージックライブラリに Apple Music の曲を追加することができる

アプリ作成においては Apple Music への登録状況を取得し、主に次の目的で使用します、

- 曲が再生可能かを判定
- 登録していなければ登録画面を表示する
- iCloud ミュージックライブラリに曲を追加できるかを判断

アプリのケイパビリティを取得するには、SKCloudServiceController の requestCapabilities メソッドを呼びます。ただし、取得するためには事前にアクセス許可を受けておく必要があります。

```
SKCloudServiceController().requestCapabilities(completionHandler: { (capability, error) in
 print(capability) // [.none, .musicCatalogPlayback, .addToCloudMusicLibrary]
 if capability.contains(.musicCatalogPlayback) {
 print(".musicCatalogPlayback ケイパビリティが利用できる")
 }
})
```

ケイパビリティは SKCloudServiceCapability 型で、OptionSet プロトコルを実装している構造体なので、contains メソッドを使って目的のケイパビリティが含まれているかどうかを確認します。

特にデバイスで制限を設定していない場合には、ケイパビリティは次のようにになります。

表 14.3: Apple Music に登録している/していない場合のケイパビリティ

	Apple Music に登録していない	Apple Music に登録している
musicCatalogPlayback	x	○
musicCatalogSubscriptionEligible	○	x
addToCloudMusicLibrary	x	○

つまり、capability に .musicCatalogSubscriptionEligible があるて、.musicCatalogPlayback がない場合、ユーザーは Apple Music に登録しておらず登録すれば再生可能だということになります。このときに必要に応じて Apple Music への登録画面を表示します。.addToCloudMusicLibrary は設定 -> iCloud ミュージックライブラリの設定の影響を受けます（図 14.11）。

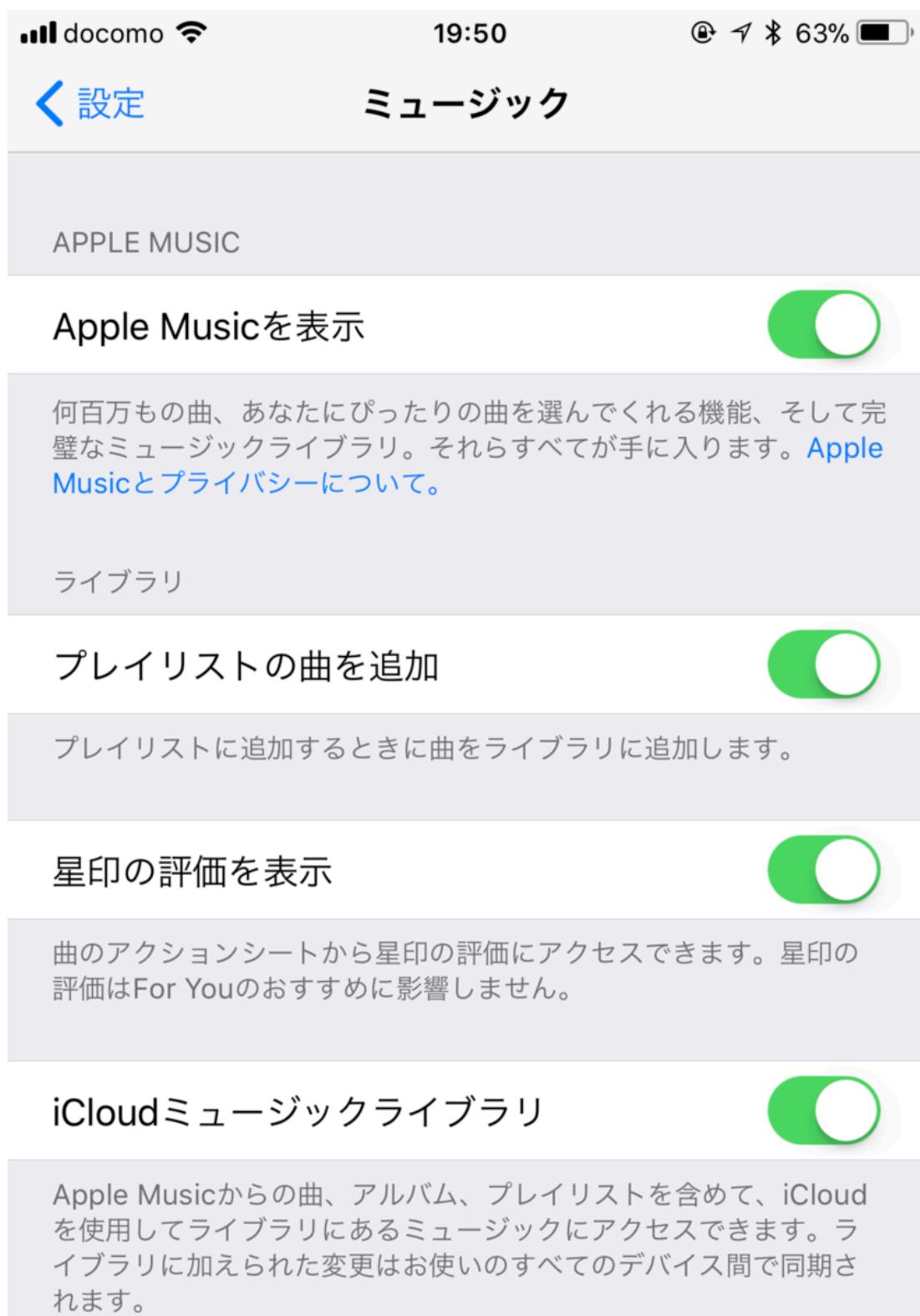


図 14.11: 設定.app の iCloud ミュージックライブラリの設定

### Music User Token

Apple Music API でユーザー固有のデータ、たとえば再生履歴などを取得するには、Music User Token が必要です。

Music User Token を取得するには、`SKCloudServiceController` の `requestUserToken` メソッドを呼びます。

```
requestUserToken(forDeveloperToken:completionHandler:)
```

引数に Developer トークンを渡して呼ぶと、コールバックで Music User Token が得られます。

```
let token = "生成した Developer トークン"
SKCloudServiceController().requestUserToken(forDeveloperToken: token) { (userToken, error) in
 print(userToken) //Music User Token
}
```

Music User Token を使う例として、ユーザーが頻繁に再生（ヘビーローテーション）している曲を取得してみましょう。これまで Authorization ヘッダーに Developer トークンを渡していましたが、さらに Music-User-Token というヘッダを追加します。

Apple Music API ではユーザー固有のデータの場合、URL が次のように `/{{version}}/me` となります。

```
https://api.music.apple.com/v1/me/
```

ヘビーローテーションしている曲を取得するリクエストの curl コマンドは次のとおりです。

```
curl -v -H "Music-User-Token: $USER_TOKEN" -H "Authorization: Bearer $DEVELOPER_TOKEN"
\ "https://api.music.apple.com/v1/me/history/heavy-rotation" ""
```

実行すると、よく聞いている曲がレスポンスとして返ります。

```
{
 "data": [
```

```
{
 "attributes": {
 "artistName": "Salyu",
 "artwork": {
 ...
 },
 "copyright": "© 2005 TOY'S FACTORY",
 "genreNames": [
 "J-Pop",
 "ミュージック"
],
 "isComplete": true,
 "isSingle": false,
 "name": "landmark",
 "playParams": {
 "id": "526002742",
 "kind": "album"
 },
 "recordLabel": "TOY'S FACTORY",
 "releaseDate": "2005-06-15",
 "trackCount": 11,
 "url": "https://itunes.apple.com/jp/album/landmark/id526002742"
 },
 "href": "/v1/catalog/jp/albums/526002742",
 "id": "526002742",
 "type": "albums"
},
],
"href": "/v1/me/history/heavy-rotation"
}
```

### Apple Music の登録画面を表示する

MusicKit では、Apple Music に登録していない（＝利用していない）人に向けて登録を促し、登録開始できる画面も用意されています。SKCloudServiceSetupViewController は通常の ViewController と同様に `present` メソッドで表示しますが、表示する前にどういった表示を行うのかの設定が必要です。現在は登録を促すページのみ表示可能で、これを表すオプションを事前に設定します。設定は `load` メソッドで行います。

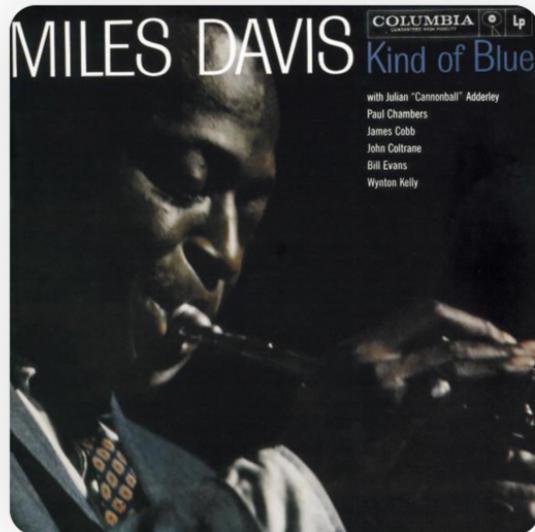
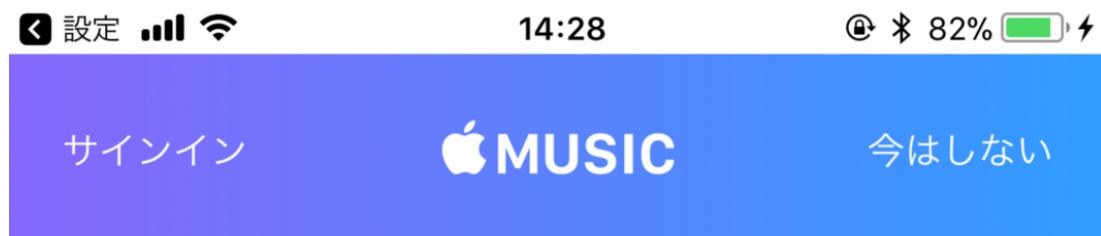
```
let controller = SKCloudServiceSetupViewController()
controller.load(options: [
 .action : SKCloudServiceSetupAction.subscribe
],
 completionHandler: { (flag, error) in
 print(flag)
 print(error)
 })
present(controller, animated: true, completion: nil)
```

`options` には `SKCloudServiceSetupOptionsKey` で表すキーと、`SKCloudServiceSetupAction` 型の値をペアにして渡します。上のコードではキーに `.action`、値には `SKCloudServiceSetupAction.subscribe` を渡していく、Apple Music の登録画面を表示することを指示しています。`.action` は必須のキーで、現在、設定できるのは `.subscribe` のみです。

その他の設定できるキーと値は次のとおりです。

SKCloudServiceSetupOptionsKey	値
<code>.iTunesItemIdentifier</code>	<code>StoreID</code> (例:"268443097")
<code>.affiliateToken</code>	iTunes アフィリエイトプログラムのトークン
<code>.campaignToken</code>	iTunes アフィリエイトプログラムのキャンペーントークン
<code>.messageIdentifier</code>	<code>.addMusic, .connect, .join, .playMusic</code>

`.iTunesItemIdentifier` キーと `StoreID` を値として設定すると、`StoreID` のアイテム (=曲、アルバム) が登録画面に表示されます。「So What」を設定すると(図 14.12) のようになります。`.affiliateToken, .campaignToken` キーは iTunes アフィリエイトプログラムのそれぞれのトークンを設定します。



聴きたいすべての音楽がここに。ダウンロードしてお楽しみください。

登録すると、ミュージックライブラリのすべての音楽やその他莫大な数の曲を お使いのあらゆるデバイス

図 14.12: So What を設定した場合の表示

.messageIdentifier キーを使うと、登録画面で表示されるメッセージのタイプを変更できます。値には SKCloudServiceSetupMessageIdentifier 型の .addMusic, .connect, .join, .playMusic の 4 種類が用意されています。

この登録画面を表示したあと、ユーザーが登録したかどうかはデリゲートメソッドの cloudServiceSetupViewControllerDidDismiss 内で確認するのがいいでしょう。登録したかどうかの確認はケイパビリティで行います。

```
func cloudServiceSetupViewControllerDidDismiss(_
 cloudServiceSetupViewController: SKCloudServiceSetupViewController) {

 SKCloudServiceController().requestCapabilities(
 completionHandler: { (capability, e
 if capability.contains(.musicCatalogPlayback) {
 print(".musicCatalogPlayback ケイパビリティが利用できるので Apple Music へ登録し
た")
 }
})
}
```

#### 14.2.4 MusicPlayer

実際に曲を再生するには、MediaPlayer フレームワークの MPMusicPlayerController を使います。

MPMusicPlayerController にはクラス変数 `systemMusicPlayer`、`applicationMusicPlayer`、`applicationQueuePlayer` の 3 つがあります（表●●）。

表 14.4: MPMusicPlayerController クラスのプレーヤー

クラス変数	機能
<code>systemMusicPlayer</code>	システム共通のプレーヤー
<code>applicationMusicPlayer</code>	アプリケーション固有のプレーヤー
<code>applicationQueuePlayer</code>	アプリケーション固有のプレーヤー + キュー操作機能

`systemMusicPlayer` を使うとミュージック App と共にプレーヤーを使うので、アプリケーションがバックグラウンドに回ってもしても再生され続けます。`applicationMusicPlayer` はアプリケーション専用のプレーヤーなのでバックグラウンドになった場合、再生は停止します。そのかわりミュージック App に影響を与えません。たとえばジョギングアプリの場合、そのアプリを起動しているときだけしか音楽が再生されないと不便なので、`systemMusicPlayer` を使ったほうがいいでしょう。しかし、勝手に曲を再生すると、ユーザーが設定した曲やプレイリストなど、ミュージック App の状態に影響を与えることがありますので、注意が必要です。ゲームなどでバックグラウンドミュージックとして使う場合には、`applicationMusicPlayer` を使ってアプリの終了=音楽の終了にしたほうがいいケースも多いでしょう。`applicationQueuePlayer` は `applicationMusicPlayer` にキューを操作できる機能を追加したもので、後ほどキューの操作で解説します。

##### 曲を再生する

ある曲を再生するには、`setQueue` メソッドで曲をキューに追加して、`play` メソッドを呼びます。MusicKit のキューとは、次に再生される曲のリストのことです。たとえばミュージック App である曲を選んで次に再生を選ぶと図 14.13 のような画面になります。

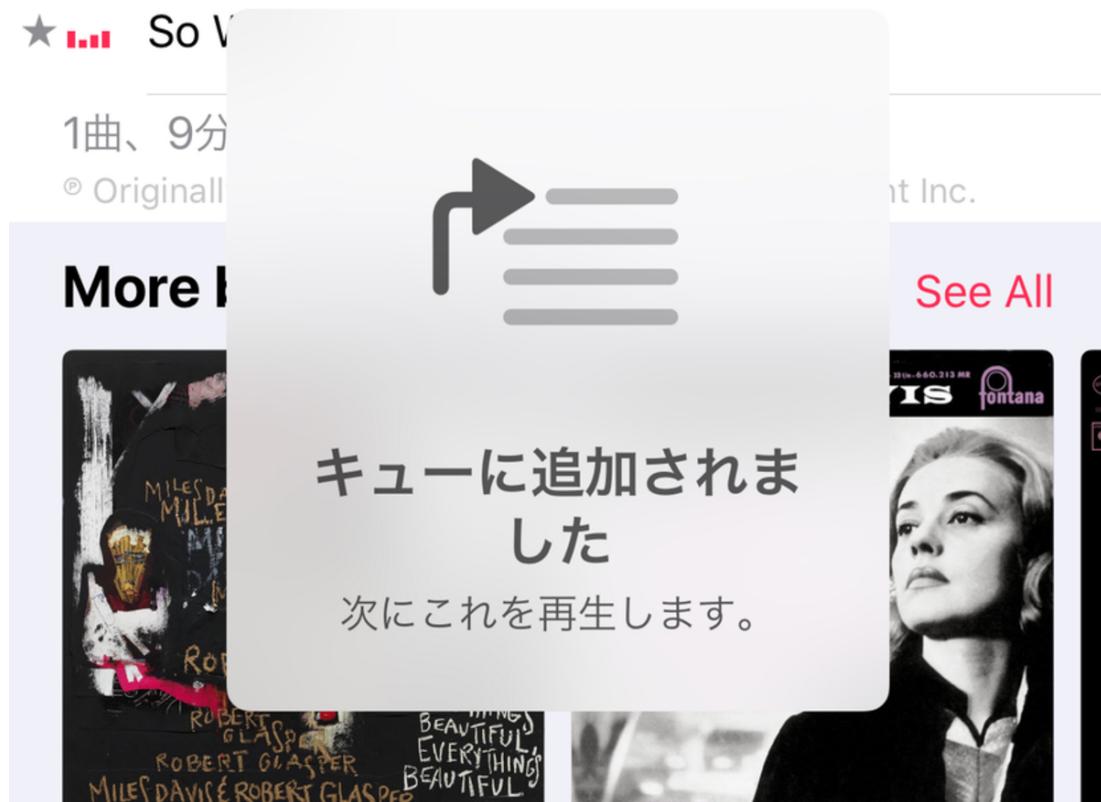


図 14.13: ミュージック App で曲をキューに追加する

Kind of Blue の「So What」の"playParams"は次のようになっていました。

```
"playParams" : {
 "id" : "268443097",
 "kind" : "song"
},
```

これを再生する場合、次のように記述します。`setQueue` メソッドの引数には Apple Music API で取得した StoreID を配列で渡します。

```
let musicPlayerController = MPMusicPlayerController.applicationMusicPlayer
let soWhatId = "268443097"
musicPlayerController.setQueue(with: [soWhatId])
musicPlayerController.play()
```

### アルバムを再生する

キューには曲 (kind=song) だけでなく、アルバム (kind=album) も同じように追加できます。Kind of Blue の"playParams"は次のようにになっていました。

```
"playParams" : {
 "id" : "268443092",
 "kind" : "album"
}
```

アルバムを追加する場合のコードは次のとおりです。曲を再生するときのコードと比べ、`setQueue` メソッドの引数が変わっているだけです。

```
let musicPlayerController = MPMusicPlayerController.applicationMusicPlayer
let kindOfBlueAlbumId = "268443092"
musicPlayerController.setQueue(with: [kindOfBlueAlbumId])
musicPlayerController.play()
```

この場合アルバムの全トラックがキューに追加されるので、次の曲を再生すると Kind of Blue の2曲目「Freddie Freeloader」が再生されます。次の曲を再生するには `skipToNextItem` メソッドを呼びます。

```
musicPlayerController.skipToNextItem()
```

### キューを操作する

`applicationMusicPlayer` を使うと次の曲や前の曲へ移動できます。しかし、たとえば次の曲のタイトルが何であるか、を知ることはできません。キューの状態、つまり次の曲のタイトルは何か、といった情報を事前に取得することができません。キューの情報が必要、もしくは内容を変更する場合には `applicationQueuePlayer` を使います。`systemMusicPlayer` にキュー機能を追加したプレーヤーはありません。

```
class var applicationQueuePlayer: MPMusicPlayerApplicationController { get }
```

`applicationQueuePlayer` の `perform` メソッドでキューの変更と現在の状態が取得できます。

```
func perform(queueTransaction: @escaping (MPMusicPlayerControllerMutableQueue)
 -> Void, completionHandler: @escaping (MPMusicPlayerControllerQueue, Error?) -> Void)
```

引数で渡される MPMusicPlayerControllerMutableQueue は変更可能なキューで、insert、remove メソッドがあります。

たとえば、Kind of Blue のアルバムの最後にもう一度「So What」を追加するには次のようにします。

```
musicPlayerController.setQueue(with: [kindOfBlueAlbumId]) //Kind of Blue をキューに追加
musicPlayerController.play() //再生しないと実際にキューに追加されない

musicPlayerController.perform(queueTransaction: { (mutableQueue:) in
 mutableQueue.insert(
 let soWhatId = "268443097"
 MPMusicPlayerStoreQueueDescriptor(storeIDs: [soWhatId]), //So What を追加
 after: mutableQueue.items.last) //最後に追加
}) { (queue, error) in
 for item in queue.items {
 print(item.title!)
 }
}
```

実行結果は次のとおりです。

```
So What
Freddie Freeloader
Blue In Green
All Blues
Flamenco Sketches
So What
```

completionHandler には変更後のキューが MPMusicPlayerControllerQueue として渡されるので、これを使うとキュー内の曲のタイトルをすべて表示できます。musicPlayerControllerQueue.items は MPMediaItem の配列を返します。MPMediaItem は MediaPlayer フレームワークにおいてライブラリの中のアイテム（＝曲）の情報を複数持つもので、title、artist、albumTitle などのプロパティがあります。ここでは曲のタイトルを取得して表示しました。

Kind of Blue のアルバムから「So What」を取り除く場合は次のようにします。

```
musicPlayerController.setQueue(with: [kindOfBlueAlbumId]) //Kind of Blue をキューに追加
musicPlayerController.play()
musicPlayerController.perform(queueTransaction: { (mutableQueue) in
```

```
 mutableQueue.remove(mutableQueue.items.first!) //最初の item = So What を取り除く
}) { (queue, error) in
 for item in queue.items {
 print(item.title!)
 }
}
```

```
Freddie Freeloader
Blue In Green
All Blues
Flamenco Sketches
```

#### Apple Music の曲を iCloudMusicLibrary に追加する

MPMediaLibrary の addItem メソッドを使うと、iCloud ミュージックライブラリに Apple Music の曲を追加できます。ただし、.addToCloudMusicLibrary ケイパビリティが利用可能である必要があります。

追加するには MPMediaLibrary の addItem メソッドを呼びます。productID には StoreID を渡します。

```
func addItem(withProductID productID: String,
completionHandler: (([MPMediaEntity], Error?) -> Void)? = nil)
```

たとえば「So What」を追加するときのコードは次のとおりです。

```
let soWhatId = "268443097"
MPMediaLibrary.default().addItem(withProductID: soWhatId) { (items, error) in
 if let item:MPMediaItem = items.first as? MPMediaItem {
 print(item.title!) //So What
 }
}
```

ミュージック App を開いてみると、最近追加した項目に Kind of Blue があるはずです。

#### 14.2.5 Apple Music API のその他のトピック

ここまで扱わなかったものの中からいくつか便利なもの、特徴のあるものを紹介します。

2017 年 8 月時点では、以降で紹介する API でユーザー固有なものはすべて反映までに 3 分以上時間がかかる場合がありました。Apple Music API 固有の問題というよりは、ミュージック App

でも同様な現象があるため、即時反映というわけではなさそうです。

### Like / Dislike

曲やアルバムに Like! / Dislike を付けることができます。

```
PUT https://api.music.apple.com/v1/me/ratings/albums/{id}
```

唯一の PUT リクエストで、パラメータとして次の JSON が必要です。

```
{"type": "rating", "attributes": {"value": 1}}
```

attributes の value が 1 だと Like、-1 が Dislike を表します。

```
curl -v -H 'Content-Type: application/json' \
-H "Music-User-Token: $USER_TOKEN" -H "Authorization: Bearer $DEVELOPER_TOKEN" \
-H 'Accept: application/json' -X PUT -d '{"type": "rating", "attributes": {"value": 1}}' \
"https://api.music.apple.com/v1/me/ratings/albums/526002742"
```

### Charts

Apple Music のチャートが取得できます。

```
GET https://api.music.apple.com/v1/catalog/{storefront}/charts?type={types}
```

```
curl -v -H "Authorization: Bearer $DEVELOPER_TOKEN" \
"https://api.music.apple.com/v1/catalog/jp/charts?type=songs"
```

types=songs で検索するとヒット曲のチャートが取得できるように思います。Apple Music のチャートであり、iTunes ストアのランキングではないため、たとえば 2017 年 8 月時点では日本のチャートでは Mr.Children の曲が売り上げ 1 位ですが、Apple Music のチャートであるため別の曲が返ります。

### 検索のオートコンプリート

検索する場合のキーワードを補完する、オートコンプリート機能があります。

```
GET https://api.music.apple.com/v1/catalog/{storefront}/search/hints
```

```
curl -v -H "Authorization: Bearer $DEVELOPER_TOKEN" \
"https://api.music.apple.com/v1/jp/search/hints?term=miles"
```

とすると、次のようにキーワード候補が取得できます。

```
{
 "results": {
 "terms": [
 "a thousand miles",
 "miles away",
 "miles davis",
 "miles davis - so what and greatest hits",
 "milestone/sora～この声が届くまで～ - single",
 "miles ahead",
 "miles davis quintet",
 "milestone",
 "milestones",
 "miles word"
]
 }
}
```

### 最近再生した曲

```
GET https://api.music.apple.com/v1/me/recent/played
```

で取得できます。curl コマンドは次のとおりです。

```
curl -v -H "Music-User-Token: $USER_TOKEN" -H "Authorization: Bearer $DEVELOPER_TOKEN" \
"https://api.music.apple.com/v1/me/recent/played"
```

その他にもいろいろな API がありますので、Apple Music API Reference<sup>\*8</sup>を参照ください。

<sup>\*8</sup> <https://developer.apple.com/library/content/documentation/NetworkingInternetWeb/Conceptual/AppleMusicWebServicesReference/>

## 14.3 AirPlay 2

### 14.3.1 AirPlay 2 とは

AirPlay 2 はオーディオ向け AirPlay（以下、AirPlay 1）の機能改善版で、Apple は以下の特徴を謳っています。

- マルチルーム対応
- Long-Form Audio
- Enhanced Buffering

マルチルーム対応とは、たとえばリビングの Apple TV に AirPlay しつつ、客間の HomePod にも AirPlay することができる機能のことです。マルチ「ルーム」となっていますが、実際にはマルチデバイス対応です。AirPlay 1 では 1 対 1 で接続できるのみでした。Long-Form Audio とは、音楽、ポッドキャストなどの一定の長さのあるオーディオ・コンテンツのことで、ベルやアラートなどで中断されることなく音楽などを再生できるようになりました。Enhanced Buffering（バッファリングの強化）では、AirPlay する際にスピーカー側に転送するデータ（バッファ）のサイズが大きく取られ、より安定した再生ができるようになります。以前の AirPlay では秒単位だったのが AirPlay 2 では分単位になっています。

以上のように、複数デバイスに安定再生できるのが AirPlay 2 の特徴です。

AirPlay 2 をサポートするプラットフォームは iOS/macOS/tvOS で、対応スピーカーは HomePod<sup>\*9</sup>、Apple TV 第 4 世代、あとは今後登場するサードパーティの AirPlay 2 スピーカーです。

### 14.3.2 AirPlay 2 に対応する

アプリを AirPlay 2 に対応する手順は次のとおりです。

1. Audio Session のカテゴリ設定で Long-Form Audio であることを表明する
2. AirPlay Picker を表示して AirPlay 先を選択できるようにする
3. Enhanced Buffering に対応した再生方法で実装する

### 14.3.3 Long-Form Audio

Long-Form Audio とは、Long-From Audio とは、音楽、ポッドキャストなどの、一定の長さがあるオーディオ・コンテンツのことです。アラート音のような効果音は含みません。Long-Form Audio であることを示すには Audio Session で次のようにポリシーを設定します。

```
let audioSession = AVAudioSession.sharedInstance()
audioSession.setCategory(AVAudioSessionCategoryPlayback,
```

<sup>\*9</sup> <https://www.apple.com/homepod/>

```
mode: AVAudioSessionModeDefault,
routeSharingPolicy: .longForm)
```

たとえば iPhone に電話がかかってきた場合を考えてみましょう。iOS には Audio Session という仕組みがあり、オーディオの再生優先度が決定できます。電話の音声は最優先されるので、音楽を聞いていたとしても中断されて電話音声が流れます。しかし Long-Form Audio であることを示しておけば、その再生は別の系統として扱われるため、電話がかかってきても AirPlay が中断されません。

.LongForm を設定できるのは Audio Session カテゴリが `AVAudioSessionCategoryPlayback` で、モードが `AVAudioSessionModeDefault`、`AVAudioSessionModeMoviePlayback`、`AVAudioSessionModeSpokenAudio` のいずれかの場合のみです。

Audio Session のカテゴリ、モードの詳細についてはドキュメント<sup>\*10</sup>を参照してください。

#### 14.3.4 AirPlay Picker を表示する

AirPlay Picker とは、アプリからコンテンツを AirPlay する先を選択するための View のことです。クラスは `AVRoutePickerView` です。このビューを表示すると図 14.15 のボタンが表示され、タップすると AirPlay 先を選択する専用の GUI が表示されます図 14.15。UIView のサブクラスなので、他のビューに `addSubview` して表示します。

リスト 14.3: UIViewController の `self.view` の中央に表示する例

```
let routePickerView = AVRoutePickerView()
self.view.addSubview(routePickerView)
routePickerView.translatesAutoresizingMaskIntoConstraints = false
routePickerView.centerXAnchor.constraint(equalTo: self.view.centerXAnchor).isActive = true
routePickerView.centerYAnchor.constraint(equalTo: self.view.centerYAnchor).isActive = true
```

<sup>\*10</sup> <https://developer.apple.com/documentation/avfoundation/avaudiosession>



図 14.14: AVRoutePickerView

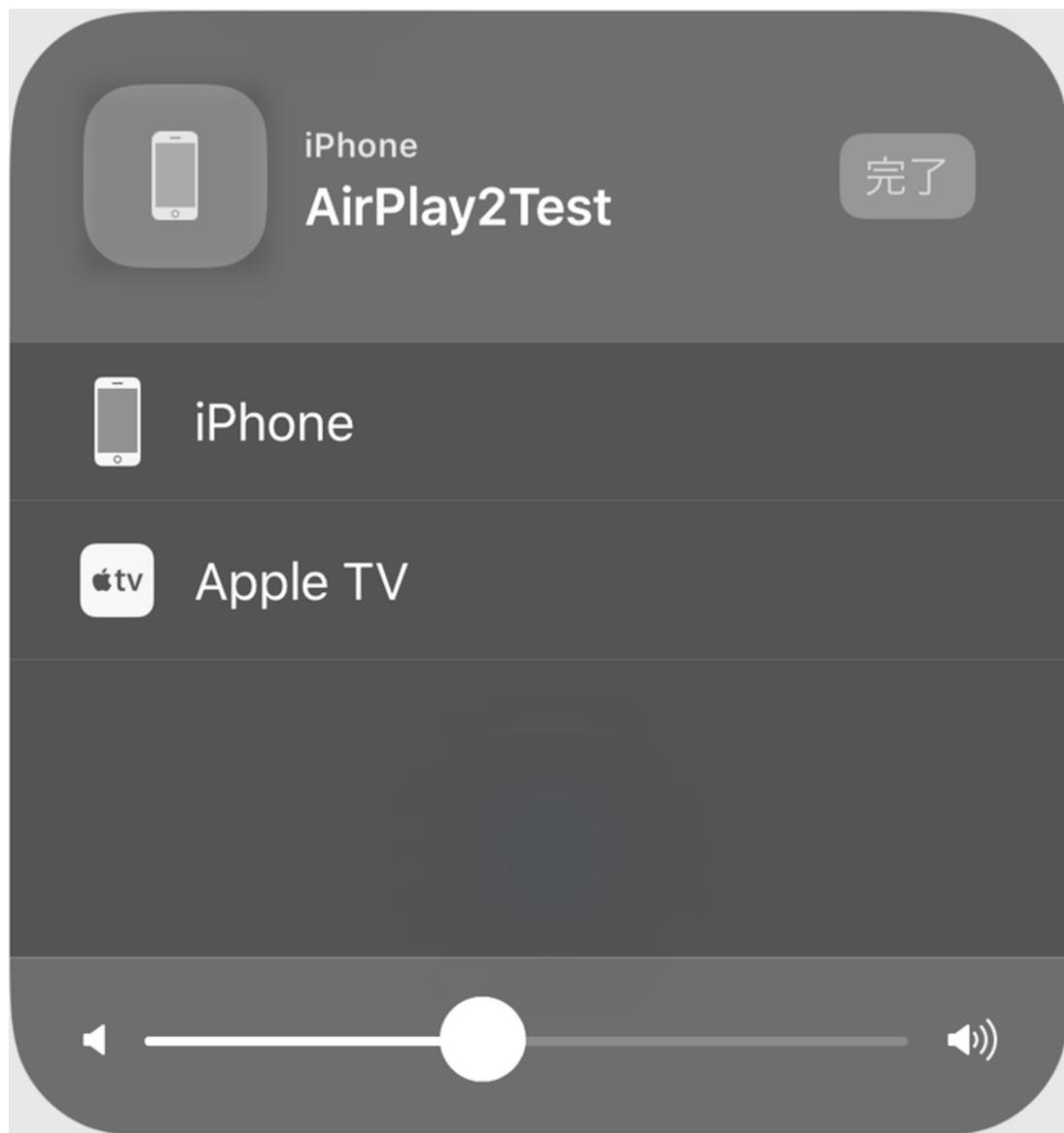


図 14.15: AVRoutePickerView をタップして表示される画面

#### AVRouteDetector

`AVRouteDetector` は AirPlay の状態を管理できるクラスです。AirPlay 先が増えたり減ったりしたときに通知を受け取ることができます。`AVRouteDetector` の `isRouteDetectionEnabled` を `true` にして、`AVRouteDetectorMultipleRoutesDetectedDidChange` 通知を受け取るようにします。

```
let routeDetector = AVRouteDetector()
routeDetector.isRouteDetectionEnabled = true
```

```
NotificationCenter.default.addObserver(
 forName: NSNotification.Name.AVRouteDetectorMultipleRoutesDetectedDidChange,
 object: routeDetector,
 queue: nil) { (notification) in
 print("AVRouteDetectorMultipleRoutesDetectedDidChange")
}
```

### 14.3.5 Enhanced Buffering に対応する

Enhanced Buffering に対応するには、2つの方法があります。1つは `AVPlayer` もしくは `AVQueuePlayer` を使う方法で、もう1つは `AVSampleBufferAudioRenderer` と `AVSampleBufferRenderSynchronizer`（以下、`AVSampleBuffer` クラス）を使う方法です。

### 14.3.6 AVPlayer/AVQueuePlayer を使って再生する

`AVPlayer` を使うとローカル、サーバー上のオーディオファイル、ムービーを再生できます。これまでの再生方法と同じ記述で、自動的に Enhanced Buffering に対応してくれます。

```
let url = Bundle.main.url(forResource: "source", withExtension: "caf")
player = AVPlayer(url: url!)
player.play()
```

`AVQueuePlayer` は `AVPlayer` にキュー機能を追加したものです。

### 14.3.7 AVSampleBuffer クラスを使って再生する

`AVSampleBuffer` クラスを使うと、バッファする内容を自由に操作できます。そのため、リアルタイムに生成した音を再生する場合や、特殊な暗号化がされた動画の再生など、`AVPlayer`/`AVQueuePlayer` では対応できないような目的のときに使います。

具体的には `AVSampleBufferAudioRenderer` がバッファを要求してきた場合に `CMSampleBuffer` をキューに追加することで何を再生するかを決定します。`AVSampleBufferAudioRenderer` はオーディオのレンダリング、`AVSampleBufferRenderSynchronizer` は主に再生スピードの変更を担当します。

`AVSampleBuffer` クラスを使って再生する場合の手順は次の通りです。

1. `AVSampleBufferAudioRenderer` と `AVSampleBufferRenderSynchronizer` のインスタンスを用意する (`audioRenderer`, `renderSynchronizer`)
2. `renderSynchronizer` の再生スピード (`rate`) を 1.0 にする
3. バッファの要求を開始する
4. `audioRenderer` がバッファを要求している場合、再生する `CMSampleBuffer` を順番に返す

5. 曲の終わりに達するなど、バッファするものがなくなった場合には nil を返す

以下でオーディオファイルにエフェクトをかけて再生したい場合の、AVSampleBuffer クラスを使った例を解説します。SampleBufferAudioPlayer クラスとして実装します。

(本章のサンプルコードはなるべく短くなるよう、例外処理などを可能な限り省略しています)

```
class SampleBufferAudioPlayer: NSObject {
 //audioRenderer と renderSynchronizer をインスタンス変数として用意しておく
 let audioRenderer = AVSampleBufferAudioRenderer()
 let renderSynchronizer = AVSampleBufferRenderSynchronizer()

 //オーディオ・レンダリング処理はサブスレッド内で行う必要があるため、
 //専用の serializationQueue を用意しておく
 let serializationQueue = DispatchQueue(label: "serialization queue")

 //AVAudioFile を使ってバッファを読み込む
 var sourceFile: AVAudioFile!
 //再生するオーディオのフォーマット情報。CD クオリティの場合 16bit/44.1kHz のような情報を持つ
 var readingFormat: AVAudioFormat!

 //最終的に AudioRenderer に渡す CMSampleBuffer のオーディオフォーマット
 let sampleBufferformatDescription = createFloat32CMAudioFormatDescription()

 //現在オーディオファイルのどの位置を読み込んでいるかを保持
 var currentFrame: AVAudioFrameCount = 0

 //このファイルを読み込む
 let sourceDataURL = Bundle.main.url(forResource: "source", withExtension: "caf")!

 override init() {
 renderSynchronizer.addRenderer(audioRenderer)

 // AVAudioFile を使う場合、Interleaved でバッファを読み込むと
 // Buffer 数=1, チャンネル数 1 のバッファになってしまふので
 // interleaved: false = Non Interleaved で読み込む

 //pcmFormatFloat32 = オーディオデータを Float32 型で Non Interleaved で読み込む
 sourceFile = try! AVAudioFile(forReading: sourceDataURL,
 commonFormat: .pcmFormatFloat32,
 interleaved: false)
 readingFormat = sourceFile.processingFormat
 }
}
```

```
func play(){
 //audioRenderer 関係の処理は DispatchQueue の async メソッド内で実行します。
 serializationQueue.async {
 self.startEnqueueing()
 self.renderSynchronizer.rate = 1.0
 self.audioRenderer.volume = 1.0
 }
}
```

```

}

func startEnqueueing() {
 //バッファの要求を開始します
 self.audioRenderer.requestMediaDataWhenReady(on: serializationQueue) {
 [weak self] in
 guard let strongSelf = self else { return }
 let audioRenderer = strongSelf.audioRenderer

 // audioRenderer.isReadyForMoreMediaData == true の場合は
 // audioRenderer がバッファを必要としています
 while audioRenderer.isReadyForMoreMediaData {
 if let sampleBuffer = strongSelf.nextSampleBuffer() {
 //バッファをキューに追加します
 audioRenderer.enqueue(sampleBuffer)
 } else {
 //nextSampleBuffer が nil の場合はファイルの末に達しているので再生を終えます
 audioRenderer.stopRequestingMediaData()
 break
 }
 }
 }
}

```

オーディオファイルからバッファを読み込むにはいくつかの方法がありますが、`AVAudioFile` を使うのが簡単です。イニシャライザに必要であれば読み込むフォーマットを指定して、`read` メソッドでバッファを渡すだけでバッファを読み込みます。読み込むバッファには `AVAudioPCMBuffer` を使います。ここでは `Float32` 型で Non Interleaved で読み込むようにしました。Non Interleaved とはオーディオファイルが 2ch ステレオの場合に左右のチャンネルがそれぞれ別のバッファになっている形式のことです。左右のチャンネルのサンプルを LLL…RRR…と持ちます。Interleaved の場合にはバッファは一つで、左右のチャンネルのサンプルを LRLRLR…と持ちます。

`nextSampleBuffer` の処理は少々複雑ですが、オーディオファイルの先頭から順番にバッファを読み込んで返すだけです。

```

func nextSampleBuffer() -> CMSampleBuffer? {
 let buffer = AVAudioPCMBuffer(pcmFormat: readingFormat,
 frameCapacity:processingFrameLength)!

 do{
 // オーディオファイルから AVAudioPCMBuffer にデータを読み込みます。
 try sourceFile.read(into: buffer, frameCount: processingFrameLength)
 }catch {
 return nil
 }

 // audioStreamBasicDescription.mSampleRate = 再生するサンプリングレートと
 // currentFrame = 現在位置を考慮して CMSampleTimingInfo を作成します。
 var timing = CMSampleTimingInfo(
 duration: CMTIMEMake(1, Int32((audioStreamBasicDescription.mSampleRate))),
 presentationTimeStamp: CMTIMEMake(Int64(currentFrame),

```

```
 Int32((audioStreamBasicDescription.mSampleRate)),
 decodeTimeStamp: kCMTimeInvalid)

 // 読み込んだフレーム数だけ位置を進めます。
 currentFrame += buffer.frameLength

 // Non Interleaved で読み込んだが AVSampleBufferAudioRenderer は
 // Interleaved の CMSampleBuffer を渡すとレンダリングエラーになるので
 // Interleaved に変換します
 var audioBufferList = convertToInterleaved(
 audioBufferList: buffer.mutableAudioBufferList,
 frameLength: buffer.frameLength)

 // ここでエフェクト処理を行います
 performEffect(audioBufferList:audioBufferList)

 // CMSampleBuffer を作成します
 var sampleBuffer: CMSampleBuffer? = nil
 CMSampleBufferCreate(nil,nil,false,nil,nil,
 sampleBufferformatDescription,
 CMItemCount(buffer.frameLength),
 1,
 &timing,
 0,
 nil,
 &sampleBuffer)

 // CMSampleBuffer にオーディオファイルから読み込んだバッファをセットします
 CMSampleBufferSetDataBufferFromAudioBufferList(sampleBuffer!,
 kCFAllocatorDefault,
 kCFAllocatorDefault,
 0,
 &audioBufferList)
 return sampleBuffer
}
```

AVPlayer で再生する場合に比べて必要な処理が多いですが、再生するバッファを直接操作できるため自由度が高くなっています。

## 14.4 AVAudioEngine のアップデート

### 14.4.1 AVAudioEngine とは

AVAudioEngine は AVAudioNode を接続して必要なオーディオ処理を実現する仕組みです。 AVAudioNode はオーディオファイルの再生、エフェクト、マイク入力などのオーディオ処理に必要な機能を担当します。

iOS 11 時点で AVAudioNode のサブクラスは、次の 11 個があります。

クラス	機能
AVAudioInputNode	マイク入力
AVAudioPlayerNode	オーディオファイル、バッファの再生
AVAudioMixerNode	ミキサーとアウトプット
AVAudioUnitMIDIInstrument	MIDI 音源
AVAudioUnitDelay	ディレイ
AVAudioUnitDistortion	ディストーション
AVAudioUnitEQ	イコライザー
AVAudioUnitReverb	リバーブ
AVAudioUnitSampler	サンプラー
AVAudioUnitTimePitch	タイムストレッチとピッチシフト
AVAudioUnitVarispeed	再生スピードを変更する

これらを自由に組み合わせてオーディオ処理を実現できます。たとえば、マイク入力とエフェクトのリバーブ、最後にスピーカーを接続すると、カラオケマイクのような処理が実現できます（図14.16）。

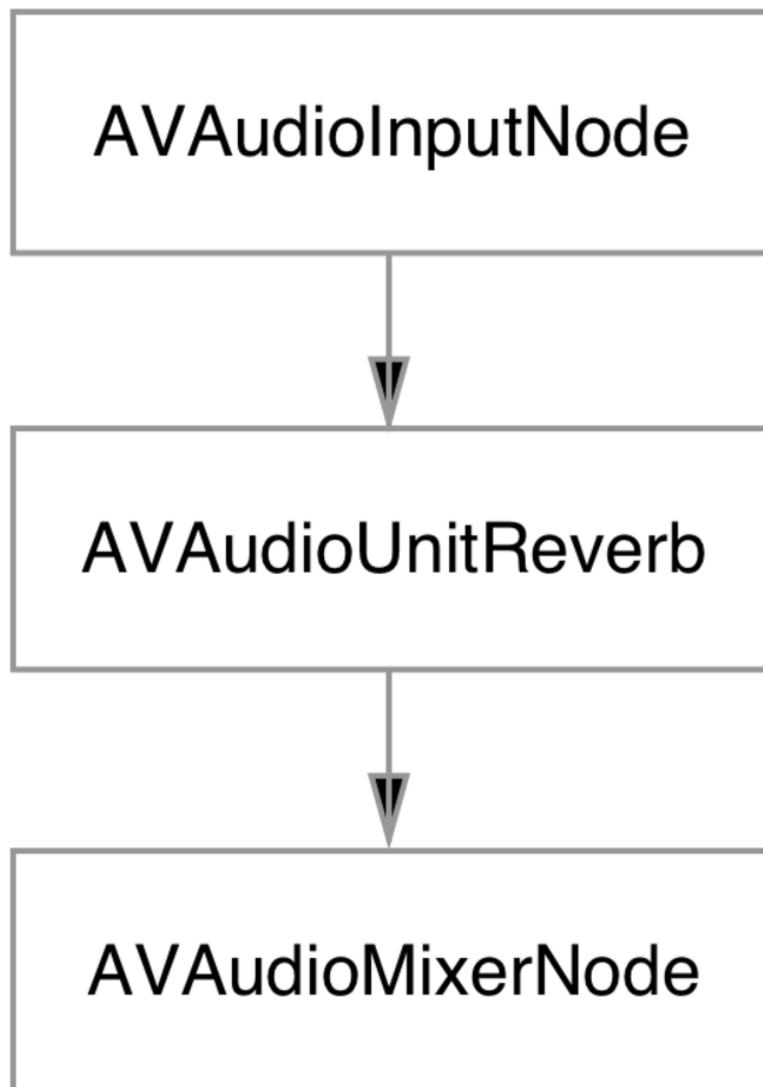


図 14.16: ‘AVAudioNode’の接続例

オーディオファイルにリバーブをかけて再生する場合のコードは次のとおりです。

リスト 14.4: オーディオファイルにリバーブをかけて再生する

```
import AVFoundation
import PlaygroundSupport

let sourceFileURL = Bundle.main.url(forResource: "source", withExtension: "caf")!
let sourceFile = try! AVAudioFile(forReading: sourceFileURL)
let format = sourceFile.processingFormat

let engine = AVAudioEngine()

//Node を準備する
let player = AVAudioPlayerNode()
let reverb = AVAudioUnitReverb()

//AVAudioEngine に関連付けする
engine.attach(player)
engine.attach(reverb)

// リバーブのパラメーターをセットする。ここでは中規模ホールの残響のプリセットを読み込んでいる
reverb.loadFactoryPreset(.mediumHall)
reverb.wetDryMix = 50

// Audio Unit (の Node) を接続する。
// player -> reverb -> mainMixer の順に接続している
engine.connect(player, to: reverb, format: format)
engine.connect(reverb, to: engine.mainMixerNode, format: format)

// 再生するオーディオファイルをセット
player.scheduleFile(sourceFile, at: nil)

// 再生を開始
try! engine.start()
player.play()
```

#### 14.4.2 AVAudioEngine のマニュアル・レンダリング

iOS 11 のアップデートで、AVAudioEngine がマニュアル・レンダリングに対応しました。これまでは実時間で再生することしかできませんでしたが、任意のタイミングで処理を実行できるようになりました。これによって、たとえばあるオーディオファイルにイコライザーをかけて新しいファイルに書きだす、といったことも出来ます。

マニュアルレンダリングを行うには、`enableManualRenderingMode` メソッドで必要な設定を行い、`renderOffline` メソッドでレンダリングを実行します。

`enableManualRenderingMode` メソッドには、どういうオーディオ・フォーマットで処理するか、最大何フレームずつ処理するかを渡します。

```
func enableManualRenderingMode(_ mode: AVAudioEngineManualRenderingMode,
 format pcmFormat: AVAudioFormat,
 maximumFrameCount: AVAudioFrameCount) throws
```

`renderOffline` メソッドには処理するフレーム数とオーディオ・バッファを渡します。

```
func renderOffline(_ number_of_frames: AVAudioFrameCount,
 to buffer: AVAudioPCMBuffer) throws -> AVAudioEngineManualRenderingStatus
```

オーディオファイルにリバーブをかけて書きだす場合の実装例は次のとおりです。

リスト 14.5: オーディオファイルにリバーブをかけて書きだす

```
....

player.scheduleFile(sourceFile, at: nil)

let maxNumber_of_frames: AVAudioFrameCount = 4096 //一度に処理するフレーム数
try! engine.enableManualRenderingMode(.offline, format: format,
 maximumFrameCount: maxNumber_of_frames)

//処理を開始する
try! engine.start()
player.play()

//書きだすファイルを用意する
let documentsPath = NSSearchPathForDirectoriesInDomains(.documentDirectory,
 .userDomainMask,
 true)[0]
let outputURL = URL(fileURLWithPath: documentsPath + "/output.caf")
let outputFile = try! AVAudioFile(forWriting: outputURL,
 settings: sourceFile.fileFormat.settings)

//レンダリング用のバッファを用意する
let buffer = AVAudioPCMBuffer(pcmFormat: engine.manualRenderingFormat,
 frameCapacity: engine.manualRenderingMaximumFrameCount)!

//engine.manualRenderingSampleTime を見て、ファイルの長さに達するまで繰り返し実行する
while engine.manualRenderingSampleTime < sourceFile.length {
 //ファイルの最後では、4096 フレームに満たない場合があるので計算する
 let frames_to_render = min(
 buffer.frameCapacity,
 AVAudioFrameCount(sourceFile.length - engine.manualRenderingSampleTime))

 //レンダリングを実行
 let status = try! engine.renderOffline(frames_to_render, to: buffer)
 switch status {
 case .success:
 //レンダリングが成功したのでファイルにバッファを書き込む
 try outputFile.write(from: buffer)
 default:
 break
 }
}
```

```
player.stop()
engine.stop()
```

実行すると、output.caf にリバーブが反映されたオーディオファイルが書き出されます。

#### 14.4.3 マニュアル・レンダリングをエフェクト機能として組み込む

マニュアル・レンダリングを応用すると、既存のオーディオ・システムにエフェクト機能を組み込むことができます。ただし、再生するバッファを自分で管理するシステムに限ります。

手順は次のとおりです。

1. AVAudioPlayerNode と使用したいエフェクトや AVAudioMixerNode を接続する
2. AVAudioEngine をオフラインレンダリング・モードにする
3. AVAudioEngine の処理を開始する
4. エフェクトをかけたいバッファ（ソースバッファ）を受け取ったら、AVAudioPlayerNode に scheduleBuffer として渡す
5. オフラインレンダリングを実行する
6. ソースバッファにエフェクトがかかったバッファが取得できるのでこれを再生する

AirPlay2 のサンプル（サンプルコード 00000000）に組み込んでみましょう。サンプルコード 00000000 の nextSampleBuffer メソッドで返していたバッファに対してエフェクトをかけなければよいことになります。

```
class SampleBufferAudioPlayer: NSObject {
 //インスタンス変数を追加
 let engine = AVAudioEngine()
 let player = AVAudioPlayerNode()
 let reverb = AVAudioUnitReverb()
 ...

 override init(){
 renderSynchronizer.addRenderer(audioRenderer)
 sourceFile = try! AVAudioFile(forReading: sourceFileURL,
 commonFormat: .pcmFormatFloat32,
 interleaved: false)
 format = sourceFile.processingFormat

 engine.attach(player)
 engine.attach(reverb)

 reverb.loadFactoryPreset(.mediumHall)
 reverb.wetDryMix = 50

 try! engine.enableManualRenderingMode(.offline,
 format: format,
 maximumFrameCount: processingFrameLength)
```

```
engine.connect(player, to: reverb, format: format)
engine.connect(reverb, to: engine.mainMixerNode, format: format)

try! engine.start()
player.play()
}
```

`nextSampleBuffer` では、`AVAudioEngine` にソースとなるバッファを渡してレンダリングし、結果のバッファから `CMSampleBuffer` を返します。

```
func nextSampleBuffer() -> CMSampleBuffer?{
 let sourceBuffer = AVAudioPCMBuffer(pcmFormat: format,
 frameCapacity:processingFrameLength)!

 do{
 try sourceFile.read(into: sourceBuffer, frameCount: processingFrameLength)
 }catch {
 return nil
 }

 //buffer を追加。renderOffline を実行するとこのバッファがレンダリングされる
 player.scheduleBuffer(sourceBuffer, completionHandler: nil)

 //結果を受け取るバッファを用意
 let resultBuffer = AVAudioPCMBuffer(pcmFormat: format,
 frameCapacity:processingFrameLength)!

 //レンダリングを実行
 //resultBuffer は sourceBuffer にリバーブがかったバッファになる
 try! engine.renderOffline(processingFrameLength, to: resultBuffer)

 //以下、resultBuffer を CMSampleBuffer にセットして返す
}
```

#### 14.4.4 まとめ

iOS 11 の Audio 関連のアップデートによって

- Music Kit によって、Apple Music の膨大なライブラリとユーザーデータが自由に扱える
- Air Play2 によってあらゆるオーディオ・ソースがマルチデバイスで安定再生できる
- エフェクト関連も順調に充実してきている

ことを述べました。これらを組み合わせるヒントについても一つ提示しました。HomePod の登場によって、これまで以上にホームオーディオ関連の盛り上がりが予想されます。本章を参考に新しいアプリに挑戦いただければ幸いです。 [1] C.M. ビショップ (2012) 『パターン認識と機械学習上』 (元田浩 (監訳), 栗田多喜夫 (監訳) · 他訳) 丸善出版。

[2] C.M. ビショップ (2012) 『パターン認識と機械学習 下』 (元田浩 (監訳), 栗田多喜夫 (監訳) · 他訳) 丸善出版。

- [3] Richard O. Duda, Peter E. Hart, David G. Stork (2001)『Pattern Classification』Wiley-Interscience.
- [4] 佐藤一誠 (2016)「ノンパラメトリックベイズ 点過程と統計的機械学習の数理 (機械学習プロフェッショナルシリーズ)」講談社。
- [5] 石井 健一郎、上田 修功 (2014)「続・わかりやすいパターン認識—教師なし学習入門」オーム社。
- [6] 牧野貴樹、他 (2016)「これから強化学習」森北出版。
- [7] 本多淳也、中村篤祥 (2016)「バンディット問題の理論とアルゴリズム (機械学習プロフェッショナルシリーズ)」講談社。
- [8] 金谷健一 (2005)『これなら分かる最適化数学—基礎原理から計算手法まで』共立出版。
- [9] 金谷健一 (2003)『これなら分かる応用数学教室—最小二乗法からウェーブレットまで』共立出版。
- [10] Apple(2017)「Converting Trained Models to Core ML」, <<https://developer.apple.com/documentation/coreml>> 2017年7月23日アクセス。
- [11] Apple (2017)「Core ML」, <<https://developer.apple.com/documentation/coreml>> 2017年7月23日アクセス。
- [12] jupyter, "<http://jupyter.org>"
- [13] Apple(2017), "Core ML in depth" <https://developer.apple.com/videos/play/wwdc2017/710/> "WWDC 2017 - Session 710 - iOS, macOS, tvOS, watchOS"
- [14] scikit-learn, "<http://scikit-learn.org>"
- [15] Keras, "<https://keras.io>"
- [16] MNIST, "<http://yann.lecun.com/exdb/mnist/>"
- [17] 岡谷貴之 (2015)『深層学習 (機械学習プロフェッショナルシリーズ)』講談社。
- [18] Ian Goodfellow, Yoshua Bengio, Aaron Courville (2016)『Deep Learning (Adaptive Computation and Machine Learning series)』The MIT Press.
- [19] C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9.
- [20] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 770-778.
- [21] Alex Sosnovshchenko (2017)「Why Core ML will not work for your app (most likely)」, <<http://alexsoasn.github.io/ml/2017/06/09/Core-ML-will-not-Work-for-Your-App.html>> 2017年8月24日アクセス。
- [22] What's New in Cocoa Touch, <https://developer.apple.com/videos/play/wwdc2017/201/>
- [23] Updating Your App for iOS 11 <https://developer.apple.com/videos/play/wwdc2017/204/>
- [24] Building Apps with Dynamic Type <https://developer.apple.com/videos/play/wwdc2017/245/>
- [25] Asset Catalog Changes in Xcode 9 <http://martiancraft.com/blog/2017/06/xcode9-asset-catalog-changes/>

assets/

[26] Size Classes And Core Components <https://developer.apple.com/videos/play/wwdc2017/812/>

[27] WWDC 2017: Large Titles and Safe Area Layout Guides <https://www.bignerdranch.com/blog/wwdc-2017-large-titles-and-safe-area-layout-guides/>

# **iOS 11 Programming**

---

2017年7月12日 初版第1刷 発行

著 者 堤 修一、吉田 悠一、池田 翔、坂田 晃一、加藤 尋樹、川邊 雄介、岸川 克己、所 友太、永野 哲久

---