# iOS描画および印刷ガイド

**iPhone**

**2011-09-26**

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/

# 目次

# 図、表、リスト

第 6 章 印刷 59

# iOSでの描画と印刷

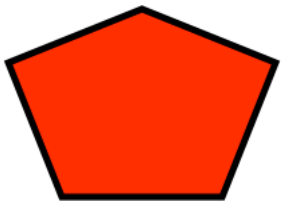
iOSにはアプリケーションで使用できる次の2つの基本的な描画方法があります。

- **ネイティブの描画テクノロジーを使用する。**Core GraphicsフレームワークやUIKitフレームワークを含むネイティブの描画テクノロジーは、2Dの描画をサポートしています。
- **OpenGL ESを使用する。**このオープンスタンダードのグラフィックライブラリは、ゲーム、仮想プロトタイプ、機械設計および建築設計など、高いフレームレートを必要とするアプリケーションによる2Dおよび3Dのデジタルコンテンツ作成をサポートします。

**注：** この文書の一部には、『*iPhone Application Programming Guide*』と『*iPad Programming Guide*』で使用された情報が含まれています。

ネイティブの描画テクノロジーは、ビューやウインドウによって提供されるインフラストラクチャを利用してカスタムコンテンツのレンダリングと表示を行います。ビューが最初に表示されると、システムはビューにそのコンテンツの描画を依頼します。システムビューは自身のコンテンツを自動的に描画しますが、カスタムビューの場合は、システムがカスタムビューの`drawRect:`メソッドを呼び出します。このメソッド内で、ネイティブの描画テクノロジーを使用して、図形、テキスト、画像、グラデーション、またはその他の必要なビジュアルコンテンツを描画します。ビューのコンテンツを変更したい場合に、デベロッパ側で`drawRect:`メソッドを呼び出すことはしません。代わりに、`setNeedsDisplay`または`setNeedsDisplayInRect:`メソッドを呼び出して、ビューの全部または一部を再描画するようシステムに依頼します。アプリケーションの存続期間中、このサイクルが繰り返され続けます。

**図 I-1** `UIBezierPath`クラスのメソッドを利用して描画した図形



この文書では、ネイティブの描画テクノロジー、特に、UIKitフレームワークが提供するテクノロジーについて説明します。OpenGL ESを使用してアプリケーションのコンテンツを描画する方法の詳細については、『*OpenGL ES Programming Guide for iOS*』を参照してください。

**関連する章：** 「iOSでのグラフィックスと描画」（13 ページ）

# 概要

## 描画の基礎を支える重要な概念

UIKitとCore Graphicsを利用してコンテンツを描画する場合は、ビューの描画サイクルのほかに、いくつかの概念に精通していなければなりません。

- `drawRect:`メソッドでは、UIKitはディスプレイにレンダリングするための**グラフィックスコンテキスト**を作成します。このグラフィックスコンテキストには、描画システムが描画コマンドを実行するために必要な情報（塗りつぶし色、描画色、フォント、クリッピング領域、線の幅などの属性を含む）が含まれています。また、ビットマップ画像やPDFコンテンツの描画用にカスタムグラフィックスコンテキストを作成して描画することもできます。
- UIKitは、ビューの左上角を描画の原点とする**デフォルト座標系**を持っています。正の座標値は、左上角の原点から下および右の方向に伸びます。現在の変換行列（ビューの座標空間をデバイスの画面にマップする）を修正することによって、サイズ、向き、および基盤となるビューやウインドウを基準にしたデフォルト座標系の位置を変更できます。
- iOSでは、**論理座標空間**（ポイント単位で距離を表す）は、デバイスの座標空間（単位はピクセル）と同じではありません。精度を高めるために、ポイント数は浮動小数点値で表します。

**関連する章：** 「iOSでのグラフィックスと描画」（13 ページ）

## UIKit、Core Graphics、Core Animationが提供する多数の描画ツール

UIKitとCore Graphicsには、グラフィックスコンテキスト、ベジェパス、画像、ビットマップ、透過レイヤ、色、フォント、PDFコンテンツ、描画矩形とクリッピング領域など、 多数の補足的なグラフィックス機能が揃っています。さらに、Core Graphicsには線の属性、色空間、パターン色、グラデーション、影、および画像マスクに関連する関数があります。Core Animationフレームワークを利用すると、ほかのテクノロジーで作成したコンテンツを操作したり表示したりして、滑らかなアニメーションを作成できます。

**関連する章：** 「iOSでのグラフィックスと描画」（13 ページ）、「ベジェパスの基礎」（35 ページ）、「PDFコンテキストの作成と設定」（44 ページ）、「画像」（51 ページ）

## 高解像度画面対応への変更は容易

iOSデバイスの中には、高解像度画面を備えているものがあるため、アプリケーションは高解像度画面のデバイスと低解像度画面のデバイスの両方で動作するように準備する必要があります。さまざまな解像度に対応するために必要な作業のほとんどはiOSが処理しますが、アプリケーション側でできる作業もいくつかあります。たとえば、特別に指定された高解像度の画像を提供したり、現在の倍率を反映するためにレイヤ関連や画像関連のコードを修正したりする作業などがあります。

**関連する章：** 「高解像度画面のサポート」（27 ページ）

## 印刷に関する一連のオプション

iOS 4.2では、アプリケーションはワイヤレスでコンテンツを印刷できます。印刷ジョブを組み立てる際に、アプリケーションは次の3つの方法でUIKitに印刷対象コンテンツを提供できます。

- 直接印刷可能な1つまたは複数のオブジェクトをフレームワークに渡す。この場合、アプリケーションの関与は最小限しか必要としません。画像データやPDFコンテンツを保持または参照するオブジェクトには、`NSData`、`NSURL`、`UIImage`、または`ALAsset`があります。
- 印刷ジョブに印刷フォーマッタを割り当てる。印刷フォーマッタは、特定のタイプのコンテンツ（プレーンテキスト、HTMLなど）を複数のページにレイアウトできるオブジェクトです。
- 印刷ジョブにページレンダラを割り当てる。通常、ページレンダラは印刷するコンテンツの一部または全部を描画する`UIPrintPageRenderer`のカスタムサブクラスのインスタンスです。ページレンダラは、1つ以上の印刷フォーマッタを使用して、描画に利用したり、印刷可能なコンテンツを整形します。

**関連する章：** 「印刷」（59 ページ）

# iOSでのグラフィックスと描画

高品質のグラフィックスは、アプリケーションのユーザインターフェイスのなかで重要な部分を占めます。高品質のグラフィックスを提供することでアプリケーションの外観がよくなるだけでなく、アプリケーションをシステムのほかの部分の自然な延長のように見せることができます。iOSには、高品質のグラフィックスを作成するために、Open GLによるレンダリングと、QuartzやCore Animation、UIKitを使用したネイティブレンダリングという2つの手段が用意されています。

Open GLフレームワークは主に、ゲームの開発や、高いフレームレートを必要とするアプリケーションを対象にしています。Open GLは、デスクトップコンピュータ上で2Dや3Dのコンテンツを作成するために使用する、C言語ベースのインターフェイスです。iOSは、OpenGL ESフレームワークを通じてOpenGLの描画をサポートしています。このフレームワークは、 OpenGL ES 2.0およびOpenGL ES v1.1のどちらの仕様もサポートしています。OpenGL ESは、特に組み込みハードウェアシステム上での使用を念頭において設計されており、デスクトップバージョンのOpenGLとは多くの点で異なります。

よりオブジェクト指向の描画手法を望むデベロッパ向けに、iOSは、Quartz、Core Animation、およびUIKitのグラフィックスサポート機能を提供しています。Quartzはメインの描画インターフェイスで、パスベースの描画、アンチエイリアスを施したレンダリング、グラデーション付き塗りつぶしパターン、画像、カラー、座標空間変換、PDF文書の作成、表示、解析をサポートします。UIKitは、線画、Quartz画像、およびカラー操作用のObjective-Cラッパーを提供します。Core Animationは、UIKitの多くのビュープロパティについて変更をアニメーション化するための基盤をサポートします。カスタムアニメーションの実装に使用することもできます。

この章では、iOSアプリケーションの描画プロセスの概要を取り上げます。また、サポートする描画テクノロジーごとに、描画テクニックも紹介します。さらに、iOSプラットフォームの描画コードを最適化するためのヒントやガイダンスも示します。

**重要：** UIKitクラスは、一般にスレッドセーフではありません。すべての描画関連操作は、アプリケーションのメインスレッドで実行されなければなりません。

## UIKitのグラフィックスシステム

iOSでは、OpenGL、Quartz、UIKit、またはCore Animationのいずれによるものかに関係なく、すべての描画が`UIView`オブジェクトの領域内で実行されます。ビューは、描画が実行される、画面内の領域を定義します。システムに用意されているビューを使用する場合、ビューの描画は自動的に処理されます。しかし、カスタムビューを定義する場合は、自ら描画コードを用意する必要があります。OpenGLを使用して描画するアプリケーションでは、レンダリングサーフェスをセットアップしたら、OpenGLが指定する描画モデルを使用します。

Quartz、Core Animation、およびUIKitの場合、以降の各セクションで説明する描画概念を用います。

# ビューの描画サイクル

UIViewオブジェクトの基本的な描画モデルでは、要求に応じてコンテンツを更新します。ただし、UIViewクラスでは、更新要求を集め、最も適切なタイミングで描画コードに引き渡すことにより、更新プロセスをより簡単かつ効率的に実行できます。

ビューの一部を再描画するときはいつでも、UIViewオブジェクトに組み込まれている描画コードが、このオブジェクトのdrawRect:メソッドを呼び出します。描画コードは、ビューのうち再描画が必要な部分を示す矩形をこのメソッドに渡します。カスタムビューサブクラスではこのメソッドをオーバーライドして、ビューのコンテンツを描画します。ビューが初めて描画されるときは、drawRect:メソッドに渡される矩形にはビューの表示領域全体が含まれています。その後の呼び出しでは、この矩形はビューのなかで実際に再描画が必要な部分だけを表します。次に示すアクションにより、ビューの更新が引き起こされます。

- ビューの一部を隠している別のビューの移動または除去
- 非表示になっていたビューの再表示（hiddenプロパティをNOに設定）
- ビューを画面外までスクロールし、スクロールし戻す
- ビューのsetNeedsDisplayメソッドまたはsetNeedsDisplayInRect:メソッドの明示的な呼び出し

drawRect:メソッドを呼び出した後、ビューは自らを更新済みとしてマークを付け、新たなアクションが到着して別の更新サイクルがトリガされるのを待ちます。ビューに静的コンテンツが表示されている場合、必要となるのは、スクロールやほかのビューの存在によって生じる、ビューの見え方の変化に対応することだけです。しかし、ビューのコンテンツを定期的に更新する場合は、更新をトリガするsetNeedsDisplayメソッドまたはsetNeedsDisplayInRect:メソッドを呼び出すタイミングを指定しなければなりません。たとえば、1秒あたり数回コンテンツを更新する場合、タイマーを設定してビューを更新することができます。また、ユーザの操作やビュー内での新規コンテンツの作成に応答して、ビューを更新する場合もあります。

# 座標と座標変換

UIKitフレームワークでは、ウインドウまたはビューの原点は左上角に置かれ、正の座標値は原点から下および右の方向に伸びます。これがUIKitのデフォルトの座標系です。UIKitのメソッドや関数を使用して描画コードを記述する場合は、この座標系を使用して、描画対象のコンテンツの個々の点の位置を指定します。

デフォルトの座標系を変更する必要がある場合は、現在の変換行列を変更します。**現在の変換行列(CTM：Current Transformation Matrix)**は数学的行列で、ビューの座標系上の点をデバイスの画面上の点に対応付けます。ビューのdrawRect:メソッドが最初に呼び出されるときに、座標系の原点がビューの原点と一致し、正の座標軸が下および右の方向に伸びるようにCTMが設定されます。ただし、CTMに、拡大縮小、回転、変換などの係数を追加して変更を加えて、サイズ、向き、そして基盤になるビューやウインドウを基準にしたデフォルトの座標系の位置を変更することができます。

CTMに変更を加えるやり方は、手間が大幅に省けるため、ビューにコンテンツを描画する際の標準的な手法として用いられています。現行描画システムにおいて、座標点(20, 20)から始まる10×10の正方形を描画する場合、(20, 20)に移動するパスを作成した後、必要な直線をひととおり描画して正方形を完成させます。ただし、この正方形を後で点(10, 10)まで移動することにした場合は、新た

な開始点を指定してパスを作成し直す必要があります。実際、原点を変更するたびにパスを作成し直す必要が生じます。パスの作成は負担の多い操作です。これに比べて、原点が(0, 0)の正方形を作成し、望みの描画原点に一致するようにCTMに変更を加える処理は少ない負担で済みます。

Core Graphicsフレームワークでは、CTMに変更を加える方法が2通りあります。1つは、『*CGContext Reference*』で定義されているCTM操作関数を使用して直接変更する方法です。もう1つは、`CGAffineTransform`構造体を作成し、望みの変換を適用してその変換をCTMに反映させる方法です。アフィン変換を使用して変換をグループとしてひとまとめにし、それらの変換をCTMに一度に適用することができます。また、アフィン変換を評価、反転し、その結果を用いてコード内の点、サイズ、矩形の値を変更することもできます。アフィン変換については、『*Quartz 2D Programming Guide*』および『*CGAffineTransform Reference*』を参照してください。

## グラフィックスコンテキスト

カスタムの`drawRect:`メソッドを呼び出す前に、ビューオブジェクトは自動的に描画環境を設定し、コードがただちに描画を開始できるようにします。描画環境の設定の一環として、`UIView`オブジェクトは、現在の描画環境に対応するグラフィックスコンテキスト（`CGContextRef`不透過型）を作成します。このグラフィックスコンテキストには、描画システムがその後の描画コマンド実行に必要とする情報が格納されます。グラフィックスコンテキストは、描画時に使用する色、クリッピング領域、線の幅とスタイルの情報、フォント情報、合成オプションなど、基本的な描画属性を定義します。

ビュー以外の場所に描画したい場合は、カスタムのグラフィックスコンテキストオブジェクトを作成できます。Quartzでは、この作業は主に、一連の描画コマンドを取得して、画像やPDFファイルの作成にこれらのコマンドを使用したい場合に行います。グラフィックスコンテキストを作成するには、`CGBitmapContextCreate`関数または`CGPDFContextCreate`関数を使用します。グラフィックスコンテキストを作成したら、それをコンテンツの作成に必要な描画関数に渡すことができます。

グラフィックスコンテキスト、グラフィックスの状態情報の修正、グラフィックスコンテキストを使用したカスタムコンテンツの作成の詳細については、『*Quartz 2D Programming Guide*』を参照してください。グラフィックスコンテキストと一緒に使用する関数の一覧については、『*CGContext Reference*』、『*CGBitmapContext Reference*』および『*CGPDFContext Reference*』を参照してください。

## iOSでのデフォルト座標系と描画

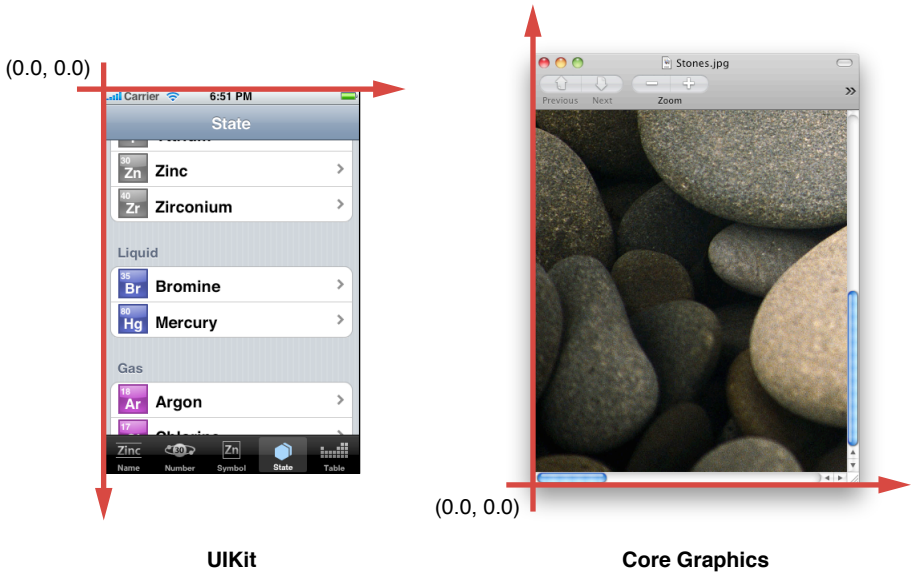
iOSではアプリケーションで何かを描画する場合は、座標系で定義される2次元の空間に描画済みのコンテンツを配置する必要があります。この概念は、一見簡単そうに思えますが、そうでもありません。iOSのアプリケーションは、何かを描画するときに異なる座標系を扱わなければならないこともあります。

グラフィックスコンテキストは、それぞれ1つの座標系を持っています。グラフィックスコンテキストは、描画操作の描画先に合わせて環境を設定します。この環境には、色、フォント、線の幅などの状態属性が含まれています。また、現在の変換行例(CTM)も含まれています。CTMについては、「座標と座標変換」（14 ページ）で説明します。CTMには現在の座標系（描画操作の参照先となる2次元のグリッド）が定義されています。iOSの描画フレームワークは、特定の描画先（画面、ビットマップ、PDFコンテンツ）に描画するためのグラフィックスコンテキストを作成します。そして、これらのグラフィックスコンテキストが、その描画先の初期の座標系を作成します。この初期の座標系は、**デフォルト座標系**と呼ばれます。

iOSの描画フレームワークは、現在のグラフィックスコンテキストに基づいて、それぞれ1つのデフォルト座標系を作成します。デフォルト座標系には、一般に次の2種類があります。

- 描画操作の原点は描画領域の左上角にあり、正の座標値は下と右の方向に伸びます（便宜上、左上角を原点とする座標系をULO(upper-left-origin)と呼びます）。UIKitフレームワークとCore Animationフレームワークは、このデフォルト座標系を持っています。
- 描画操作の原点が描画領域の左下角にあり、正の座標値は上と右の方向に伸びます（便宜上、左下角を原点とする座標系をLLO(lower-left-origin)と呼びます）。Core Graphicsフレームワークは、このデフォルト座標系を持っています。

**図 1-1**　iOSのデフォルト座標系




**注：** Mac OS Xでは、任意の描画先に描画するためのデフォルト座標系は、描画領域の左下角を原点とし、正の座標値は上と右の方向に伸びます。つまり、LLOのデフォルト座標系です。Core Graphicsフレームワークと AppKitフレームワークの描画関数や描画メソッドは、完全にこのデフォルト座標系に最適化されています。ただし、AppKitでは、この座標系を左上角原点に変更するために、プログラムでCTMを反転させる機能を提供しています。

## 画面への描画

iOSでは、UIKitが描画操作用のグラフィックスコンテキストを提供することによって、画面に描画するためのデフォルト座標系を作成します。ビューの`drawRect:`メソッドを呼び出す前に、UIKitは、ディスプレイへのレンダリング用の現在のグラフィックスコンテキストを作成します。`drawRect:`内では、アプリケーションは、グラフィックス状態パラメータ（塗りつぶしの色など）を設定して、明示的にそのパラメータを参照することなく、現在のグラフィックスコンテキストに描画できます。この暗黙のグラフィックスコンテキストは、ULOのデフォルト座標系を作成します。

`drawRect:`、またはその他のメソッド内で、Core Graphics関数を使用してビューに描画する場合は、描画用のグラフィックスコンテキストが必要になります（これらの関数のほとんどで、第1パラメータは`CGContextRef`オブジェクトでなければなりません）。`UIGraphicsGetCurrentContext`関数を呼び出すと、`drawRect:`内で暗黙的に作成されたものと同じグラフィックスコンテキストの明示的なバージョンを取得できます。同じグラフィックスコンテキストであるため、描画関数もULOのデフォルト座標系を参照しなければなりません。

Core Graphics関数を使用してUIKitビューに描画する場合は、描画操作にはUIKitのULO座標系を使用しなければなりません。あるいは、CTMに反転変換を適用して、Core GraphicsのネイティブのLLO座標系を使用して、UIKitビューにオブジェクトを描画することもできます。反転変換の詳細については、「デフォルト座標系の反転」（18 ページ）で説明します。

`UIGraphicsGetCurrentContext`関数は、必ず有効なグラフィックスコンテキストを返します。たとえば、PDFコンテキストを作成して`UIGraphicsGetCurrentContext`を呼び出すと、PDFコンテキストが返されます。Core Graphics関数を使用してビューに描画する場合は、`UIGraphicsGetCurrentContext`によって返されたグラフィックスコンテキストを使用しなければなりません。

**注：** `UIPrintPageRenderer`クラスには、印刷可能なコンテンツを描画するためのメソッドがいくつか宣言されています。`drawRect:`と同様に、UIKitには、これらのメソッドの実装用に暗黙のグラフィックスコンテキストが含まれています。このグラフィックスコンテキストは、ULOのデフォルト座標系を作成します。

## ビットマップコンテキストやPDFコンテキストへの描画

UIKitは、ビットマップグラフィックスコンテキストに画像をレンダリングする関数と、PDFグラフィックスコンテキストに描画することによって、PDFコンテンツを生成する関数を提供しています。どちらのアプローチの場合も、最初にグラフィックスコンテキスト（ビットマップコンテキスト、またはPDFコンテキスト）を作成する関数を呼び出す必要があります。返されるオブジェクトは、その後の描画や状態設定呼び出しのための、現在の（暗黙の）グラフィックスコンテキストとしての役割を果たします。グラフィックスコンテキストでの描画が終了したら、別の関数を呼び出してコンテキストを閉じます。

**ビットマップコンテキスト：**

```
UIGraphicsBeginImageContextWithOptions
ここでコンテンツを描画する
UIGraphicsEndImageContext
```

**PDFコンテキスト：**

```
UIGraphicsBeginPDFContextToFile または UIGraphicsBeginPDFContextToData
ここでコンテンツを描画する
image = UIGraphicsGetImageFromCurrentImageContext
UIGraphicsEndPDFContext
```

**注：** これらの関数の詳細については、「画像」（51 ページ）（ビットマップコンテキストへの描画の場合）と「PDFコンテンツの生成」（43 ページ）を参照してください。

UIKitが提供するビットマップコンテキストもPDFコンテキストも、ULOのデフォルト座標系を作成します。Core Graphicsには、ビットマップグラフィックスコンテキストに描画する関数に相当する関数と、PDFグラフィックスコンテキストに描画する関数に相当する関数があります。しかし、Core Graphicsを通じてアプリケーションが直接作成したコンテキストは、LLOのデフォルト座標系を作成します。

iOSでは、ビットマップコンテキストやPDFコンテキストへの描画には、UIKitの関数を使用することをお勧めします。ただし、Core Graphicsの代替関数を使用し、描画結果を表示することを意図している場合は、デフォルト座標系の違いを補正するようにコードを調整する必要が生じます。詳細については、「デフォルト座標系の反転」（18 ページ）を参照してください。

## パスの描画

UIKitとCore Graphicsには、パス（円弧と線分で作成されたベクトルベースの図形）を描画するためのAPIもあります。UIKitは`UIBezierPath`クラスを提供し、Core Graphicsは`CGPathRef`不透過型を提供しています。どちらのAPIを使用してもグラフィックスコンテキストなしでパスを作成できますが、パス内の点は現在の座標系（ULOまたはLLOのいずれか）を参照している必要があります。また、パスのレンダリングにはグラフィックスコンテキストが必要です。

レンダリング先は、ビュー、ビットマップ画像、またはPDFファイルが可能です。パスの点の配置に使用する座標系によって、パスのレンダリング方法が決まります。つまり、LLO座標系を想定してパスの点を指定した後で、UIKitビュー（デフォルト座標系はULO）にそのパスをレンダリングすると、レンダリングされた図形は意図したものとは違って見えます。最良の結果を得るには、常に、レンダリングに使用するグラフィックコンテキストの現在の座標系の原点に基づいて点を指定する必要があります。

**注：** 円弧のパスの場合は、この「ルール」に従っていても追加作業が必要です。ULO座標系に点を配置するCore Graphic関数を使用してパスを作成し、その後でパスをUIKitビューに描画すると、円弧が「指す」向きが異なります。詳細については、「異なる座標系での描画による副作用」（19 ページ）を参照してください。

iOSでパスを作成する場合は、Core Graphicsだけが提供する機能（パスへの楕円の追加など）が必要でない限り、`CGPath`関数ではなく、`UIBezierPath`を使用することをお勧めします。UIKitでパスを作成して描画する方法の詳細については、「ベジェパスを使用した図形の描画」（35 ページ）を参照してください。

## デフォルト座標系の反転

UIKitによる描画を反転させるには、LLO座標系の描画環境を、UIKitのデフォルト座標系に調整するように現在の変換行列(CTM)を変更します。描画用にUIKitのメソッドと関数のみを使用する場合は、CTMを反転させる必要はありません。しかし、UIKitの呼び出しとCore GraphicsやImage I/Oの関数呼び出しが混在する場合は、CTMの反転が必要になることがあります。

特に、Core Graphics関数を呼び出して画像やPDFコンテンツを作成した場合は、その画像やPDFページをUIKitのビューに描画すると、逆さまに表示されます。画像やページを正しく表示するには、CTMを反転させる必要があります。

Core Graphicsのコンテキストに描画されたオブジェクトを、UIKitビューに適切に表示されるように反転させるには、次の2つの手順でCTMを修正しなければなりません。原点を描画領域の左上角に変換します。次に、倍率変換を適用して、y軸に-1を掛けます。これを実行するコードは、次のようになります。

```
CGContextTranslateCTM(graphicsContext, 0.0, drawingRect.size.height);
CGContextScaleCTM(graphicsContext, 1.0, -1.0);
```

Core Graphicsの画像オブジェクトで初期化されたUIImageオブジェクトを作成すると、UIKitが反転変換を実行します。どのUIImageオブジェクトもCGImageRef不透過型に基づいています。CGImageプロパティを利用してCore Graphicsオブジェクトにアクセスして、その画像に対して何らかの処理を行うことができます（Core Graphicsには、UIKitでは利用できない画像関連機能があります）。処理が終了したら、変更後のCGImageRefオブジェクトからUIImageオブジェクトを再作成できます。

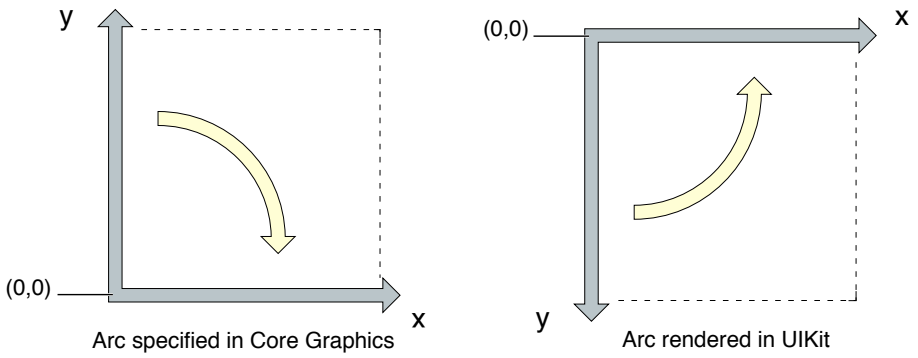
**注：** Core GraphicsのCGContextDrawImage関数を使用すると、任意のレンダリング先に画像を描画できます。この関数には2つのパラメータがあります。1つはグラフィックスコンテキスト、もう1つは、画像のサイズと、ビューなどの描画サーフェスでの位置の両方を定義した矩形です。CGContextDrawImageを利用して画像を描画している場合は、現在の座標系をLLOの向きに調整しないと、UIKitビューでは画像が逆さまに表示されます。また、この関数に渡す矩形の原点は、この関数が呼び出されたときの現在の座標系の原点になります。

## 異なる座標系での描画による副作用

ある描画テクノロジーのデフォルト座標系でオブジェクトを描画してから、別の描画テクノロジーのグラフィックスコンテキストでそれをレンダリングしたときに、描画の異常が発覚することもあります。このような副作用を考慮してコードを調整できます。

### 円弧と回転

CGContextAddArcやCGPathAddArcなどの関数を利用してパスを描画したときに、LLO座標系を想定している場合、その円弧をUIKitビューに正しくレンダリングするには、CTMを反転させる必要があります。しかし同じ関数を使用して、ULO座標系に点が配置された円弧を作成した後でそのパスをUIKitビューにレンダリングすると、元の円弧から変更されていることが分かります。円弧の終点は、UIBezierPathクラスを使用して作成した円弧の終点とは反対の向きを指しているはずです。たとえば、下向きの矢印は上を指し、弧の曲がりの向きも異なっています。Core Graphicsで描画した円弧の向きを、ULO座標系に合わせて変更しなければなりません。この向きは、これらの関数のstartAngleパラメータとendAngleパラメータによって制御します。




オブジェクトを回転した場合も、同様の鏡像効果が生じる場合があります（たとえば、CGContextRotateCTMを呼び出した場合）。ULO座標系を参照するCore Graphics関数呼び出しを使用してオブジェクトを回転すると、UIKitにレンダリングしたときのオブジェクトの向きは逆になります。コードでは、回転による向きの違いを考慮しなければなりません。それには、CGContextRotateCTMを利用して、angleパラメータの符号を反転させます（たとえば、負の値を正の値にします）。

### 影

オブジェクトから影が落ちる方向はオフセット値によって指定され、そのオフセット値の解釈は描画フレームワークの規約によります。UIKitでは、xとyの正のオフセット値によって、オブジェクトの右下に影が作成されます。Core Graphicsでは、xとyの正のオフセット値によって、オブジェクトの右上に影が作成されます。UIKitのデフォルト座標系を持つオブジェクトの位置を揃えるためにCTMを反転しても、オブジェクトの影には作用しません。このため、影は正しくオブジェクトを追跡しません。正しく追跡できるようにするには、オフセット値を現在の座標系に合うように変更する必要があります。

**注：** iOS 3.2より前は、Core GraphicsとUIKitは、影の方向に関して同じ規約（正のオフセット値によってオブジェクトの右下に影が作成される）を使用していました。

## ポイントとピクセル

iOSでは、描画コードで指定する座標と、デバイス本来のピクセルの間に差異があります。Quartz、UIKit、Core Animationなどネイティブの描画テクノロジーを使う場合は、**ポイント**で距離を表す**論理座標空間**を使用して座標値を指定します。この論理座標系は、システムフレームワークによって画面上のピクセルを管理するために使われる**デバイス座標空間**とは切り離されています。システムは論理座標空間上のポイントを自動的にデバイス座標空間上のピクセルにマップします。ただし必ずしも1対1でマップされるとは限りません。この動作の結果として、常に留意すべき、ある重要な事実が生じます。

**1ポイントが必ずしも画面上の1ピクセルに対応しているとは限りません。**

ポイント（および論理座標系）を使用する目的は、デバイスに依存しない一貫性のある出力サイズを提供することです。ポイントの実際のサイズは重要ではありません。ポイントの目的は、ビューやレンダリングしたコンテンツのサイズと位置を指定するために、コードの中で使用できる比較的一貫した尺度を提供することです。ポイントがピクセルに実際にマップされる仕組みは、システムフレームワークによって処理される詳細部分です。たとえば、高解像度画面のデバイスでは、1ポイント幅の線は、画面上では実際には2ピクセル幅の線になります。その結果、類似の2つのデバイス（そのうちの一方だけが高解像度画面を持つ）に同じコンテンツを描画した場合、コンテンツはどちらのデバイス上でもほとんど同じサイズのように見えます。

アプリケーションの描画コードにおいては、ほとんどの場合はポイントを使用しますが、ポイントがピクセルにどのようにマップされているかを知る必要が生じることもあります。たとえば、高解像度画面では、増えたピクセルを使用してコンテンツにさらにディテールを追加したり、コンテンツの位置やサイズをわずかに調整したりしたくなることがあります。iOS 4以降では、`UIScreen`、`UIView`、`UIImage`、`CALayer`の各クラスを使用して、特定のオブジェクトのポイントとピクセルの関係を示す**倍率**を明らかにすることができます。iOS 4より前のバージョンでは、この倍率は1.0であることが前提でしたが、iOS 4以降では、デバイスの解像度に応じて1.0または2.0のどちらかになります。今後は、その他の倍率も使用される可能性があります。

## 色および色空間

iOSは、Quartzで利用可能な最大限の色空間をサポートしていますが、ほとんどのアプリケーションで必要となるのはRGB色空間だけです。iOSは組み込みハードウェアで実行され、画面にグラフィックスを表示するよう設計されているため、RGB色空間が最適な色空間です。

UIColorオブジェクトは、RGB、HSB、グレイスケールの値を使用してカラー値を指定できる簡易メソッドを備えています。この方法で色を作成する場合、色空間を指定する必要はありません。色空間は、UIColorオブジェクトが自動的に決定してくれます。

Core Graphicsフレームワークでは、CGContextSetRGBStrokeColor関数およびCGContextSetRGBFillColor関数を使用して色を作成し、設定することもできます。Core Graphicsフレームワークには、ほかの色空間を使用して色を作成したり、カスタム色空間を作成する機能もありますが、描画コードでこれらの機能を使用することはお勧めしていません。描画コードでは常にRGBカラーを使用してください。

# 描画に関するヒント

以降の各セクションでは、エンドユーザから見たアプリケーションの外観を魅力的なものにしながら、質の高い描画コードを作成するためのヒントを紹介します。

## カスタム描画コードを使用する場面の判断

作成するアプリケーションのタイプに応じて、カスタム描画コードをほとんど、あるいはまったく使用しないで済ませることができます。没入型のアプリケーションでは通常、カスタム描画コードを惜しみなく使用しますが、ユーティリティ型アプリケーションや生産性型アプリケーションであれば、標準のビューやコントロールを使用してコンテンツを表示できることが少なくありません。

カスタム描画コードは、表示するコンテンツを動的に変化させる必要のある場合に限定するべきです。たとえば、描画アプリケーションであれば、カスタム描画コードを使用してユーザの描画コマンドを追跡する必要がありますし、ゲームであれば、常に変化するゲームの状況を反映するため絶えず画面を更新することになります。これらの状況では、適切な描画テクノロジーを選択し、カスタムビュークラスを作成してイベントを処理し、表示を適切に更新する必要があります。

一方、アプリケーションのインターフェイスの大半が固定されている場合、あらかじめインターフェイスを1つ以上の画像ファイルにレンダリングしておき、UIImageViewオブジェクトを使用して、それらの画像を実行時に表示することができます。インターフェイスを構築する際に、必要に応じてほかのコンテンツとImage View群を重ねることができます。たとえば、UILabelオブジェクトを使用して、設定変更可能なテキストを表示し、ボタンなどのコントロールを追加して対話機能を提供できます。

## 描画パフォーマンスの改善

描画は、どのプラットフォームでも比較的負荷の高い処理です。描画コードを最適化することは、開発プロセスにおいて常に重要です。表 1-1では、描画コードをできる限り最適化するためのヒントを紹介します。これらのヒントに加え、コードのテスト、ホットスポットや冗長部分の除去を行うためのパフォーマンスツールを必ず使用してください。

**表 1-1** 描画パフォーマンス改善のヒント

| ヒント | Action |
| --- | --- |
| 描画を最小限にする | 各更新サイクルの間は、ビューのなかで実際に変更された部分だけを更新します。`UIView`の`drawRect:`メソッドを使用して描画を実行する場合、そのメソッドに渡された更新矩形を使用して、描画の範囲を制限します。OpenGLによる描画の場合、自ら更新を追跡する必要があります。 |
| 不透過なビューは不透過としてマークする | 不透過なコンテンツを持つビューの合成は、部分的に透過なビューを合成する場合よりもはるかに負担が少なくて済みます。ビューを不透過にするためには、ビューコンテンツに透明効果が含まれないようにし、ビューの`opaque`プロパティを`YES`に設定します。 |
| スクロール時に表のセルとビューを再利用する | スクロール時に新たなビューを作成することはいかなる場合でも避けてください。新しいビューを作成するのに時間を費やすと、画面の更新に使用できる時間が少なくなり、スクロール動作が滑らかでなくなります。 |
| スクロール時に、前のコンテンツのクリアを避ける | デフォルトでは、UIKitはビューの現在のコンテキストバッファをクリアしてから`drawRect:`メソッドを呼び出して、その同じ領域を更新します。ビュー内のスクロールイベントに応答する場合、スクロールによる更新の間に繰り返しこの領域をクリアすると、負荷が大きくなる可能性があります。この動作を無効にするには、`clearsContextBeforeDrawing`プロパティの値を`NO`に変更します。 |
| 描画中のグラフィックス状態の変更を最小限に抑える | グラフィックス状態を変更すると、ウインドウサーバに負担がかかります。類似の状態情報を利用するコンテンツを描画する必要のある場合、そうしたコンテンツをまとめて描画し、必要な状態変更の回数を減らすことを試みてください。 |

# QuartzとUIKitを使用した描画

Quartzとは、iOSにおけるネイティブのウインドウサーバと描画テクノロジーを表す総称です。Core Graphicsフレームワークは、Quartzの核を構成し、コンテンツの描画に使用される主要インターフェイスです。このフレームワークは、以下を操作するためのデータ型と関数を提供します。

- グラフィックスコンテキスト
- パス
- 画像とビットマップ
- 透明レイヤ
- 色、パターン色、色空間
- グラデーションと陰影
- フォント

- PDFコンテンツ

UIKitは、グラフィックス関連の操作に焦点を合わせた各種クラスを提供し、Quartzの基本機能に基づいて構成されています。UIKitのグラフィックスクラスは、描画ツールの総合セットを意図したものではありません。その役割はCore Graphicsがすでに担っています。代わりに、UIKitグラフィックスクラスは、ほかのUIKitクラスの描画をサポートします。UIKitのサポート機能には、次のクラスと関数が含まれます。

- `UIImage`。画像を表示する不変クラスを実装します。
- `UIColor`。デバイスカラーの基本サポートを提供します。
- `UIFont`。フォント情報を必要とするクラスにフォント情報を提供します。
- `UIScreen`。画面に関する基本情報を提供します。
- `UIBezierPath`。線、円弧、楕円、その他の図形を描画できるようにします。
- `UIImage`オブジェクトをJPEG形式またはPNG形式で生成する関数
- ビットマップグラフィックスコンテキストに描画する関数
- PDFグラフィックスコンテキストに描画することによって、PDFデータを生成する関数
- 矩形を描画する関数および描画領域をクリッピングする関数
- 現在のグラフィックスコンテキストを変更したり、取得する関数

UIKitを構成するクラスとメソッドの詳細については、『*UIKit Framework Reference*』を参照してください。Core Graphicsフレームワークを構成する不透過型と関数の詳細については、『*Core Graphics Framework Reference*』を参照してください。

## グラフィックスコンテキストの設定

`drawRect:`メソッドが呼び出されるまでに、オペレーティングシステムはすでに、デフォルトのグラフィックスコンテキストの作成と設定を完了しています。このグラフィックスコンテキストへのポインタは、`UIGraphicsGetCurrentContext`関数を呼び出して取得できます。この関数は、`CGContextRef`型への参照を返します。この参照を、Core Graphics関数に渡して現在のグラフィックス状態を変更します。図 4-1に、グラフィックス状態のさまざまな側面を設定するために使用する主要な関数を示します。すべての関数の一覧については、『*CGContext Reference*』を参照してください。この表は、UIKitに代替できるものがあれば、それも示しています。

**表 1-2**　グラフィックスの状態を変更するCore Graphics関数

| グラフィックスの状態 | Core Graphics関数 | UIKitの代替 |
|---|---|---|
| 現在の変換行列(CTM) | `CGContextRotateCTM`<br>`CGContextScaleCTM`<br>`CGContextTranslateCTM`<br>`CGContextConcatCTM` | なし |

| グラフィックスの状態 | Core Graphics関数 | UIKitの代替 |
|---|---|---|
| クリッピング領域 | `CGContextClipToRect` | なし |
| 線：幅、結合、キャップ、ダッシュ、マイター上限 | `CGContextSetLineWidth`<br>`CGContextSetLineJoin`<br>`CGContextSetLineCap`<br>`CGContextSetLineDash`<br>`CGContextSetMiterLimit` | なし |
| 曲線推定の精度 | `CGContextSetFlatness` | なし |
| アンチエイリアスの設定 | `CGContextSetAllowsAntialiasing` | なし |
| 色：塗りつぶしとストロークの設定 | `CGContextSetRGBFillColor`<br>`CGContextSetRGBStrokeColor` | `UIColor`クラス |
| グローバルなアルファ値（透明度） | `CGContextSetAlpha` | なし |
| レンダリングインテント | `CGContextSetRenderingIntent` | なし |
| 色空間：塗りつぶしとストロークの設定 | `CGContextSetFillColorSpace`<br>`CGContextSetStrokeColorSpace` | `UIColor`クラス |
| テキスト：フォント、フォントサイズ、文字間のスペース、テキスト描画モード | `CGContextSetFont`<br>`CGContextSetFontSize`<br>`CGContextSetCharacterSpacing` | `UIFont`クラス |
| ブレンドモード | `CGContextSetBlendMode` | `UIImage`クラスと各種描画関数を使用して、使用するブレンドモードを指定することができます。 |

グラフィックスコンテキストには、保存されたグラフィックス状態のスタックがあります。Quartzがグラフィックスコンテキストを作成する時点では、スタックは空です。`CGContextSaveGState`関数を使用すると、現在のグラフィックス状態のコピーがスタックにプッシュされます。以後、グラフィックス状態に加えた変更はその後の描画操作に影響を与えますが、スタックに格納されたコピーには影響しません。変更が完了した時点で、`CGContextRestoreGState`関数を使用して、保存されている状態をスタックの一番上からポップすることにより、前のグラフィックス状態に戻ることができます。このようにグラフィックス状態をプッシュしたり、ポップしたりする方法は、前の状態にすばやく戻る方法であり、状態の変更を個別に取り消す必要をなくします。この方法は、状態の特定の側面（クリッピングパスなど）を元の設定に復元するただ1つの方法でもあります。

グラフィックスコンテキストの概要およびグラフィックスコンテキストを使用した描画環境の設定については、『*Quartz 2D Programming Guide*』の「Graphics Contexts」を参照してください。

## パスの作成と描画

パスとは、連続する直線とベジェ曲線を使用して表される2Dのジオメトリシーンを記述したものです。UIKitには、ビュー内に矩形などの単純なパスを描画するための`UIRectFrame`関数と`UIRectFill`関数が含まれています。Core Graphicsにも、矩形や楕円といった単純なパスを作成するための簡易関数が含まれています。

より複雑なパスの場合は、UIKitの`UIBezierPath`クラスか、Core Graphicsフレームワークのパス関数を使用して、自らパスを作成する必要があります。`UIBezierPath`を使用してパスを描く方法の詳細については、「ベジェパスを使用した図形の描画」（35 ページ）を参照してください。複雑なパス要素を構成する点を指定する方法を含め、Core Graphicsを使用してパスを描く方法の詳細については、『*Quartz 2D Programming Guide*』の「Paths」を参照してください。パスの作成に使用する関数については、『*CGContext Reference*』および『*CGPath Reference*』を参照してください。

## パターン、グラデーション、陰影の作成

Core Graphicsフレームワークには、パターン、グラデーション、陰影を作成する関数も用意されています。これらはモノクロ以外の色の作成や、作成したパスの塗りつぶしに使用します。パターンは画像やコンテンツの繰り返しを元に作成します。グラデーションと陰影はそれぞれ、色から色への滑らかな移り変わりを生み出すことができます。

パターン、グラデーション、陰影の作成と使用の詳細については、『*Quartz 2D Programming Guide*』を参照してください。

# Core Animation効果の適用

Core AnimationはObjective-Cフレームワークで、滑らかなリアルタイムアニメーションをすばやく簡単に作成する基盤を提供します。Core Animation自体は、形状や画像、その他のタイプのコンテンツを作成する基本ルーチンを提供するわけではないという点で、描画テクノロジーではありません。その代わり、別のテクノロジーを使用して作成したコンテンツを操作したり、表示したりするためのテクノロジーであるといえます。

ほとんどのアプリケーションでは、iOSで何らかの形でCore Animationを利用することによってメリットがあります。アニメーションは、今起きていることをユーザに伝えるフィードバックの働きがあります。たとえば、ユーザが「設定(Settings)」アプリケーションを操作するとき、ユーザが環境設定の下位階層に進んだり、ルート階層に戻ったりするのに合わせて、画面が視野にすべり込んだり、視野からすべり出たりします。この種のフィードバックは重要で、ユーザに対して文脈情報を提供します。アニメーションは、アプリケーションの見た目を充実させる働きもあります。

ほとんどの場合、ごくわずかな労力でCore Animationのメリットを享受することができます。たとえば、`UIView`クラスのいくつかのプロパティ（ビューのフレーム、中心、色、不透明度など）について、値が変化するとアニメーションが開始するように設定できます。UIKitにこれらのアニメーションを実行するように伝えるために、一定の作業が必要となりますが、アニメーション自体は自動的に作成され、実行されます。組み込みのビューアニメーションをトリガする方法については、『*UIView Class Reference*』の「Animating Views」 in *UIView Class Reference*を参照してください。

基本アニメーションからさらに充実させるには、Core Animationのクラスやメソッドを直接扱う必要があります。以降の各セクションでは、Core Animationに関する情報を紹介し、iOSにおいて、Core Animationのクラスやメソッドを使用して典型的なアニメーションを作成する方法について説明します。Core Animationの詳細と使用方法については、『*Core Animation Programming Guide*』を参照してください。

## レイヤについて

Core Animationにおける重要なテクノロジーがレイヤオブジェクトです。レイヤは、ビューに性質がよく似た軽量オブジェクトですが、実際には、表示するコンテンツのジオメトリやタイミング、視覚的プロパティをカプセル化する、モデルオブジェクトです。コンテンツは、次に示す3つのうち、いずれかの方法で提供されます。

- `CGImageRef`をレイヤオブジェクトの `contents`プロパティに割り当てることができます。
- デリゲートをレイヤに割り当てて、デリゲートに描画を処理させることができます。
- `CALayer`をサブクラス化し、表示メソッドの1つをオーバーライドすることができます。

レイヤオブジェクトのプロパティを操作すると、実際に操作されるのはモデルレベルのデータで、このデータによって、関連付けられているコンテンツの表示方法が決まります。そのコンテンツの実際のレンダリングはコードとは別に処理され、高速にレンダリングされるよう徹底して最適化されます。デベロッパは、レイヤコンテンツとアニメーションプロパティを設定するだけでよく、あとはCore Animationが引き継いで処理してくれます。

レイヤおよびその使い方については、『*Core Animation Programming Guide*』を参照してください。

## アニメーションについて

レイヤのアニメーション化では、Core Animationは独立したアニメーションオブジェクトを使用してアニメーションのタイミングと動作を制御します。`CAAnimation`クラスとそのサブクラスにより、コード内で使用可能なさまざまな種類のアニメーション動作が提供されます。ある値から別の値へとプロパティを移行させるシンプルなアニメーションを作成することも、指定した値とタイミング関数を通じてアニメーションを追跡する、複雑なキーフレームアニメーションを作成することもできます。

Core Animationでは、複数のアニメーションをトランザクションと呼ばれる単位にまとめることもできます。`CATransaction`オブジェクトは、アニメーションのグループを1つの単位として扱います。このクラスのメソッドを使用して、アニメーションの再生時間を設定することもできます。

カスタムアニメーションを作成する方法については、『*Animation Types and Timing Programming Guide*』を参照してください。

# 高解像度画面のサポート

iOS SDK 4.0以降を対象にビルドされるアプリケーションは、さまざまな画面解像度のデバイス上で実行できるように備えなければなりません。幸い、iOSでは複数の画面解像度を容易にサポートすることができます。さまざまな種類の画面に対処するための作業の大部分は、システムフレームワークが行います。しかし、アプリケーション側でもラスタ画像を更新するための作業がいくらか発生し、さらにアプリケーションによっては、増えたピクセルを生かすためにさらに作業が必要になることもあります。

この話題に関連する重要な背景情報については、「ポイントとピクセル」（20 ページ）を参照してください。

## 高解像度画面のサポートのためのチェックリスト

高解像度画面のデバイスに合わせてアプリケーションを更新するには、次の作業を行う必要があります。

- アプリケーションバンドル内の画像リソースごとに、高解像度の画像を用意します。これについては「画像リソースファイルの更新」（28 ページ）で説明します。
- 高解像度のアプリケーションアイコンとドキュメントアイコンを用意します。これについては「アプリケーションのアイコンと起動画像の更新」（30 ページ）で説明します。
- ベクトルベースの図形やコンテンツに対しては、従来と同様に、引き続きカスタムのCore GraphicsおよびUIKit描画コードを使用します。描画されるコンテンツにさらにディテールを追加する場合は、「カスタム描画コードの更新」（30 ページ）を参照してください。
- Core Animationレイヤを直接使用する場合は、描画前にレイヤの倍率を調整する必要が生じることもあります。これについては「Core Animationレイヤの倍率の考慮」（31 ページ）で説明します。
- 描画にOpenGL ESを使用している場合は、高解像の描画を行うかどうかを決め、それに応じてレイヤの倍率を設定します。これについては「OpenGL ESを使った高解像度コンテンツの描画」（32 ページ）で説明します。
- 作成するカスタム画像に対しては、現在の倍率を考慮に入れるように画像作成コードを修正します。これについては「プログラミングによる高解像度ビットマップ画像の作成」（30 ページ）で説明します。
- Core Animationレイヤに直接コンテンツを提供する場合は、必要に応じてアプリケーションのコードを調整して倍率の補正を行います。これについては「Core Animationレイヤの倍率の考慮」（31 ページ）で説明します。

# 無償で手に入る描画機能の向上

iOSの描画テクノロジーは、画面本来の解像度に関わらず、レンダリングされたコンテンツの見栄えをよくするための多数のサポート機能を備えています。

- 標準のUIKitビュー（テキストビュー、ボタン、テーブルビューなど）は、どのような解像度でも自動的に正しくレンダリングします。
- ベクトルベースのコンテンツ（`UIBezierPath`、`CGPathRef`、PDF）は、増えたピクセルを自動的に活用して、より明確な線で図形をレンダリングします。
- テキストは自動的により高い解像度で鮮明にレンダリングされます。
- UIKitは、アプリケーションで用意した画像の高解像度のバリエーション（`@2x`）を自動的に読み込むことができます。

既存の描画コードの大部分がうまく動作する理由は、Core Graphicsなどのネイティブの描画テクノロジーが現在の倍率を考慮してくれるからです。たとえば、ビューの1つが`drawRect:`メソッドを実装していれば、UIKitは自動的にそのビューの倍率を画面の倍率に設定します。さらにUIKitは、ビューの倍率に対応するために、描画中に使用されるあらゆるグラフィックスコンテキストの**現在の変換行列(CTM)**を自動的に変更します。このため、`drawRect:`メソッドで描画するコンテンツはすべて、デバイス本来の画面に合うように拡大縮小されます。

アプリケーションがネイティブの描画テクノロジーだけを使用してレンダリングを行っているのであれば、必要な作業は高解像度バージョンの画像を用意することだけです。システムビュー以外を使用しない、あるいはベクトルベースのコンテンツのみを使用するアプリケーションは、修正する必要はありません。しかし、画像を使用するアプリケーションは、新しいバージョンの画像をより高い解像度で提供する必要があります。具体的には、画像を2倍に拡大する必要があり、その結果、横方向と縦方向のピクセル数が2倍に増え、全体のピクセル数は4倍に増えます。画像リソースの更新に関する詳細については、「画像リソースファイルの更新」（28 ページ）を参照してください。

# 画像リソースファイルの更新

iOS 4で実行するアプリケーションは、画像リソースごとに2つのファイルを持つ必要があります。1つは、与えられた画像の標準解像度のバージョンを提供するファイル、もう1つは、それと同じ画像の高解像度バージョンを提供するファイルです。この一組の画像ファイルの命名規則は次の通りです。

- 標準：<*画像名*><*デバイス修飾子*>.<*ファイル名拡張子*>
- 高解像度：<*画像名*>`@2x`<*デバイス修飾子*>.<*ファイル名拡張子*>

各名前の<*画像名*>と<*ファイル名拡張子*>の部分には、そのファイルの通常の名前と拡張子を指定します。<*デバイス修飾子*>の部分は省略可能であり、`~ipad`または`~iphone`の文字列が入ります。これらの修飾子は、iPadとiPhone用に異なるバージョンの画像を指定する場合に追加します。高解像度画像のファイルに修飾子`@2x`を付けるのは新しい命名方法で、これによりシステムは、その画像は標準画像の高解像度のバリエーションであると認識できます。

**重要：** アプリケーションの画像の高解像度バージョンを作成する際、新しいバージョンはアプリケーションバンドル内でオリジナルと同じ場所に配置します。

## アプリケーションへの画像の読み込み

`UIImage`クラスは、アプリケーションに高解像度画像を読み込むために必要なすべての作業に対応します。新しい画像オブジェクトを作成するときには、同じ名前を使って標準バージョンと高解像度バージョンの両方の画像を要求します。たとえば、`Button.png`と`Button@2x.png`という2つの画像ファイルがある場合は、次のコードを使用してボタン画像を要求します。

```
UIImage* anImage = [UIImage imageNamed:@"Button"];
```

**注：** iOS 4以降では、画像ファイルを指定する際にファイル名拡張子を省略することもできます。

高解像度画面のデバイスでは、`imageNamed:`メソッド、`imageWithContentsOfFile:`メソッド、`initWithContentsOfFile:`メソッドは自動的に、要求された画像のうち名前に`@2x`の修飾子が付いているバージョンを探します。該当する画像が見つかったら、その画像を読み込みます。特定の画像の高解像度バージョンを提供しなかった場合、画像オブジェクトは標準解像度の画像を（もしあれば）読み込み、描画時にそれを拡大します。

`UIImage`オブジェクトは、画像を読み込む際に、その画像ファイルの識別子に基づいて自動的に`size`および`scale`プロパティを適切な値に設定します。標準解像度画像に対しては、`scale`プロパティを1.0に設定し、画像のサイズを画像のピクセル寸法に設定します。ファイル名に`@2x`の識別子が付く画像に対しては、`scale`プロパティを2.0にし、幅と高さの値を2分の1にしてこの倍率を補正します。2分の1の値は、その画像のレンダリングに論理座標空間で使う必要があるポイントによる寸法と正確に一致します。

**注：** Core Graphicsを使用して画像を作成する場合、Quartz画像には明示的な倍率がないため、倍率は1.0であるとみなされます。`CGImageRef`データ型から`UIImage`オブジェクトを作成する場合は、`initWithCGImage:scale:orientation:`を使用します。このメソッドを使用すると、特定の倍率をQuartz画像データと関連付けることができます。

`UIImage`オブジェクトは、描画時に自動的にその倍率を考慮します。したがって、アプリケーションバンドル内で正しい画像リソースを提供している限り、画像をレンダリングするコードはすべて同じ動作をするはずです。

## Image Viewを使用した画像の表示

アプリケーションが画像の表示に`UIImageView`オブジェクトを使用している場合、そのビューに割り当てる画像はすべて同じ倍率を使用しなければなりません。Image Viewを使用して、1枚の画像を表示したり複数の画像をアニメーション化したりできます。また、ハイライト画像を提供することもできます。そのため、これらの画像の1つに対して高解像度バージョンを提供する場合は、すべての画像にも同様に高解像度バージョンを用意する必要があります。

## アプリケーションのアイコンと起動画像の更新

アプリケーションのカスタム画像リソースを更新するほか、アプリケーションのアイコンと起動画像用に新しい高解像度アイコンも提供する必要があります。これらの画像リソースを更新する手順は、ほかのすべての画像リソースの場合と同様です。対象画像の新しいバージョンを作成し、その画像ファイル名に修飾文字列`@2x`を追加し、その画像をオリジナルと同じように扱います。たとえば、アプリケーションアイコンの場合、高解像度画像のファイル名をアプリケーションの`Info.plist`ファイルの`CFBundleIconFiles`キーに追加します。

アプリケーション用のアイコンの指定に関する詳細については、『*iOS App Programming Guide*』の「アプリケーションアイコン」 in *iOS App Programming Guide*を参照してください。起動画像の指定方法に関する詳細については、同じ文書の「アプリケーションの起動画像」 in *iOS App Programming Guide*を参照してください。

# カスタム描画コードの更新

アプリケーションで何らかのカスタム描画を行うときには、ほとんどの場合は画面本来の解像度について気にする必要はありません。ネイティブの描画テクノロジーが自動的に、論理座標空間で指定された座標を、画面本来のピクセルに正しくマップされるようにします。しかし、場合によってはコンテンツを正しくレンダリングするために現在の倍率を知る必要があります。このような場合に対して、UIKit、Core Animation、その他のシステムフレームワークは、描画を正しく行えるように必要な支援を提供します。

## プログラミングによる高解像度ビットマップ画像の作成

現在`UIGraphicsBeginImageContext`関数を使用してビットマップを作成している場合は、倍率を考慮するようにコードを調整する必要が生じることもあります。`UIGraphicsBeginImageContext`関数は常に倍率1.0で画像を作成します。デバイスの画面が高解像度画面の場合、この関数で作成した画像はレンダリングの時点では滑らかに見えないことがあります。倍率が1.0以外の画像を作成するには、代わりに`UIGraphicsBeginImageContextWithOptions`を使用します。この関数の使用手順は、`UIGraphicsBeginImageContext`関数の場合と同じです。

1. `UIGraphicsBeginImageContextWithOptions`を呼び出してビットマップコンテキストを（適切な倍率で）作成し、それをグラフィックススタックにプッシュします。
2. UIKitまたはCore Graphicsのルーチンを使用してその画像の内容を描画します。
3. `UIGraphicsGetImageFromCurrentImageContext`を呼び出してビットマップの内容を取得します。
4. `UIGraphicsEndImageContext`を呼び出してスタックからコンテキストをポップします。

たとえば、次のコードは200×200ピクセルのビットマップを作成します（ピクセルの数は、画像のサイズと倍率の掛け算によって求められます）。

```
UIGraphicsBeginImageContextWithOptions(CGSizeMake(100.0,100.0), NO, 2.0);
```

**注：** 常にデバイスのメイン画面に合うようにビットマップを拡大したい場合は、`UIGraphicsBeginImageContextWithOptions`関数を呼び出す際に倍率を0.0に設定します。

## 高解像度画面用にネイティブのコンテンツを調整する

高解像度画面上では異なるコンテンツを描画する場合、現在の倍率を使用して描画コードを修正することができます。たとえば、縁に沿って1ピクセル幅の境界縁が描画されているビューがあるとします。倍率が2.0のデバイス上では、`UIBezierPath`オブジェクトを使用して幅が1.0の線を描画したとすると、2ピクセル幅の線が描かれます。この場合、線幅を倍率で割ると、正しい1ピクセル幅の線が得られます。

もちろん、倍率に応じて描画特性を変更すると、予期せぬ結果が生じる可能性があります。1ピクセル幅の線は、デバイスによってはきれいに表示されるかもしれませんが、高解像度デバイス上では細すぎてはっきりと見えにくくなる可能性があります。このような変更を行うかどうかの判断はデベロッパに任されています。

## Core Animationレイヤの倍率の考慮

Core Animationレイヤを直接使用してコンテンツを提供するアプリケーションは、倍率を計算に入れるために描画コードを調整しなければならないことがあります。通常、ビューの`drawRect:`メソッド、またはレイヤのデリゲートの`drawLayer:inContext:`メソッドで描画する際、システムは自動的に倍率を計算に入れるためにグラフィックスコンテキストを調整します。しかし、ビューが次のいずれかを行う場合、倍率を認識したり変更したりすることが必要になることもあります。

- 倍率の異なる追加のCore Animationレイヤを作成し、それらをビューのコンテンツに合成する
- Core Animationレイヤの`contents`プロパティを直接設定する

Core Animationの合成エンジンは、各レイヤの`contentsScale`プロパティを見てそのレイヤのコンテンツを合成時に拡大縮小する必要があるかを判断します。アプリケーションがビューに関連付けられていないレイヤを作成する場合、新しいレイヤオブジェクトの倍率は最初はそれぞれ1.0に設定されます。倍率を変更しないまま、高解像度画面上でその後にレイヤを描画した場合、レイヤのコンテンツは倍率の相違を補うために自動的に拡大縮小されます。コンテンツを拡大縮小したくない場合は、レイヤの倍率を2.0に変更することができますが、高解像度のコンテンツを指定せずに変更すると、既存のコンテンツは期待していたよりも小さく見える可能性があります。このような問題を修正するには、そのレイヤに対して高解像度のコンテンツを用意する必要があります。

**重要：** レイヤの`contentsGravity`プロパティには、標準解像度のレイヤコンテンツが高解像度画面で拡大縮小されるかどうかを決める際に使われます。このプロパティの値は、デフォルトでは`kCAGravityResize`に設定され、レイヤコンテンツはレイヤの境界内に納まるように拡大縮小されます。これをサイズ変更しない選択肢に変更すると、自動的な拡大縮小は行われなくなります。このような状況では、それに応じてアプリケーションのコンテンツまたは倍率を調整する必要があります。

レイヤのコンテンツを異なる倍率に合うように調整することは、レイヤの`contents`プロパティを直接設定する場合に行うのが最も適切です。Quartz画像には倍率の概念はないため、直接ピクセルを操作します。このため、レイヤのコンテンツ用の`CGImageRef`オブジェクトを作成する前に、倍率を確認して、それに応じて画像サイズを調整します。具体的には、アプリケーションバンドルから適

切なサイズの画像を読み込むか、または、UIGraphicsBeginImageContextWithOptions関数を使用して倍率がレイヤの倍率と一致する画像を作成します。高解像度画像を作成しない場合は、先述のとおり既存のビットマップを拡大縮小してもかまいません。

高解像度画像の指定とロードの方法に関する詳細については、「アプリケーションへの画像の読み込み」（29 ページ）と図 4-2（48 ページ）を参照してください。高解像度画像の作成方法に関する詳細については「プログラミングによる高解像度ビットマップ画像の作成」（30 ページ）を参照してください。

## OpenGL ESを使った高解像度コンテンツの描画

アプリケーションがレンダリングにOpenGL ESを使用している場合、既存の描画コードは変更なしでそのまま動作するはずです。しかし高解像度画面に描画すると、それに応じてコンテンツが拡大縮小され、ジャギー（ギザギザ）が見られることがあります。ジャギーの原因は、OpenGL ESレンダバッファを支えるCAEAGLLayerクラスのデフォルトの動作が、その他のCore Animationレイヤオブジェクトと同じであることにあります。つまり、倍率は最初は1.0に設定されるため、Core Animationが高解像度画面上ではレイヤのコンテンツを拡大縮小してしまうということです。ジャギーを避けるには、OpenGL ESレンダバッファのサイズを画面のサイズに合わせて大きくする必要があります（ピクセルが増えると、コンテンツに提供するディテールの量を増やすことができます）。しかし、レンダバッファのピクセル数を増やすとパフォーマンスに影響があるため、高解像度画面のサポートを明示的に選択する必要があります。

高解像度の描画を行うには、OpenGL ESコンテンツの表示に使用するビューの倍率を変更しなければなりません。ビューのcontentScaleFactorプロパティを1.0から2.0に変更すると、CAEAGLLayerオブジェクトの倍率に対して相応の変更が行われます。レイヤオブジェクトをレンダバッファにバインドするrenderbufferStorage:fromDrawable:メソッドは、レイヤの境界とその倍率を掛け算することでレンダバッファのサイズを計算します。そのため、倍率を2倍にすると、レンダバッファの幅と高さが2倍になり、コンテンツに使用できるピクセル数が増えます。その後、これらの追加ピクセルに合わせたコンテンツを提供するのはデベロッパの仕事です。

リスト 2-1に、レイヤオブジェクトをレンダバッファにバインドし、結果のサイズ情報を取得するための正しいやりかたを示します。OpenGL ESアプリケーションテンプレートを使用してコードを作成した場合は、この手順はすでに行われているため、残りの作業はビューの倍率を適切に設定するだけです。OpenGL ESアプリケーションテンプレートを使用しなかった場合は、以下のようなコードを使用してレンダバッファのサイズを取得する必要があります。レンダバッファのサイズは、デバイスのタイプに対して固定であると想定してはなりません。

**リスト 2-1**　レンダバッファのストレージの初期化と実際の寸法の取得

```
GLuint colorRenderbuffer;
glGenRenderbuffersOES(1, &colorRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
[myContext renderbufferStorage:GL_RENDERBUFFER_OES fromDrawable:myEAGLLayer];

// レンダバッファのサイズを取得する。
GLint width;
GLint height;
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_WIDTH_OES,
 &width);
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_HEIGHT_OES,
 &height);
```

**重要：** `CAEAGLLayer`オブジェクトを基盤とするビューは、カスタムの`drawRect:`メソッドを実装してはなりません。`drawRect:`メソッドを実装すると、システムはビューのデフォルトの倍率を、画面の倍率と一致するように変更することになります。アプリケーションの描画コードがこの動作を想定していない場合、アプリケーションのコンテンツは正しくレンダリングされません。

高解像度の描画を採用する場合は、それに応じてアプリケーションのモデルおよびテクスチャアセットも調整する必要があります。たとえば、iPadまたは高解像度デバイスで実行するときには、ピクセル数の増加を生かして、より大きなモデルやより詳細なテクスチャを選ぶことができます。逆に、標準解像度のiPhone上では、小さなモデルとテクスチャを使い続けることができます。

高解像度のコンテンツをサポートするかどうかを決める際の重要な要素として、パフォーマンスがあります。レイヤの倍率を1.0から2.0に変更したときにピクセル数が4倍になることで、フラグメントプロセッサの負担が増えます。アプリケーションがフラグメント単位の計算を多く実行している場合、ピクセル数の増加によりアプリケーションのフレームレートが下がる可能性があります。高い倍率ではアプリケーションの実行が極端に遅くなるようであれば、次の選択肢を検討してください。

- 『*OpenGL ES Programming Guide for iOS*』のパフォーマンスチューニングガイドラインを使用して、フラグメントシェーダのパフォーマンスを最適化する。
- よりシンプルなアルゴリズムを採用してフラグメントシェーダに実装する。これにより、個々のピクセルの品質を下げて、画像全体をより高い解像度でレンダリングします。
- 1.0から2.0の間で倍率を設定する。1.5の倍率は、1.0の倍率よりも品質は高くなりますが、2.0に拡大された画像よりも塗りつぶしに必要なピクセル数は少なくて済みます。
- iOS 4以降では、OpenGL ESはオプションとしてマルチサンプル機能を提供する。アプリケーションでより小さい倍率を使用できる場合は（1.0でもかまいません）、マルチサンプルを実装してください。さらなる利点として、このテクニックは高解像度表示をサポートしていないデバイス上で、より高い品質がもたらされる点があります。

最適な解決策はOpenGL ESアプリケーションのニーズによって異なります。上記の選択肢を何回か試して、パフォーマンスと画像品質のバランスが最も良いアプローチを採用してください。

# ベジェパスを使用した図形の描画

iOS 3.2から、UIBezierPathクラスを使用してベクトルベースのパスを作成できるようになりました。このクラスは、Core Graphicsフレームワークのパス関連機能をラップしたObjective-Cのラッパーです。このクラスを使用すると、楕円や矩形などの単純な図形だけでなく、複数の直線や曲線のセグメントを組み込んだ複雑な図形も定義できます。

アプリケーションのユーザインターフェイスに図形を描画するには、パスオブジェクトを使用します。パスの輪郭を描いたり、それで囲まれた領域を塗りつぶしたり、その両方を実行したりできます。また、パスを使用して現在のグラフィックスコンテキストにクリッピング領域を定義し、それを使用してそのコンテキストでのその後の描画操作を変更できます。

## ベジエパスの基礎

UIBezierPathオブジェクトは、CGPathRefデータ型のラッパーです。**パス**は、直線や曲線のセグメントを使用して作成されるベクトルベースの図形です。矩形や多角形を作成するには、直線のセグメントを使用します。一方、円弧、円、複雑な曲線を持つ図形を作成するには、曲線のセグメントを使用します。各セグメントは、（現在の座標系での）1つ以上の点と、これらの点の解釈の仕方を定義した描画コマンドから構成されます。直線や曲線のセグメントの終端は、次のセグメントの始点になります。いくつかの直線や曲線のセグメントが接続されると、**サブパス**と呼ばれる図形を形成します。そして1つのUIBezierPathオブジェクトには、パス全体を定義する1つ以上のサブパスが含まれることもあります。

パスオブジェクトを作成する処理と使用する処理は独立です。パスの作成は最初の処理です。これには次の手順が含まれます。

1. パスオブジェクトを作成する。
2. moveToPoint:メソッドを使用して、最初のセグメントの始点を設定する。
3. 直線や曲線のセグメントを追加して、1つ以上のサブパスを定義する。
4. UIBezierPathオブジェクトの関連する描画属性を変更する。たとえば、ストロークパスのlineWidthプロパティやlineJoinStyleプロパティを設定したり、塗りつぶしパスのusesEvenOddFillRuleプロパティを設定したりします。これらの値は、必要に応じて後からいつでも変更できます。

パスを作成する際には、原点(0, 0)を起点としてパスの点を構成するようにします。そうしておくと、後でパスを移動する操作が簡単になります。描画中は、パスの点には、現在のグラフィックスコンテキストの座標系がそのまま適用されます。パスが原点を起点として構成されていれば、そのパスの位置を変更するのに、変換係数を指定したアフィン変換を現在のグラフィックスコンテキストに適用するだけで済みます。（パスオブジェクト自体を変更する代わりに）グラフィックスコンテキストを変更することの利点は、グラフィックスの状態を保存しておけば、それを復元することで、簡単に変換を元に戻せる点です。

パスオブジェクトを描画するには、strokeメソッドとfillメソッドを使用します。これらのメソッドは、現在のグラフィックスコンテキストに、パスの直線や曲線のセグメントをレンダリングします。このレンダリング処理には、パスオブジェクトの属性を使用して直線や曲線のセグメントをラスタ化することが含まれます。ラスタ化処理を行っても、パスオブジェクトそのものは変更されません。その結果、同じパスオブジェクトを現在のコンテキストや任意のコンテキストに何度でもレンダリングできます。

# パスへの直線と多角形の追加

直線と多角形は、moveToPoint:メソッドとaddLineToPoint:メソッドを使用して、点を並べて作成する単純な図形です。moveToPoint:メソッドは、作成する図形の始点を設定します。その点から、addLineToPoint:メソッドを使用して図形の直線部分を作成します。直前に指定した点と新たに指定した点の間に直線を形成し、直線を連続的に作成していきます。

リスト 3-1に、別々の直線セグメントを使用して五角形を作成するために必要なコードを示します(図 I-1 （9 ページ）は、このコードの実行結果です）。このコードでは、図形の最初の点を設定した後に、接続された4つの直線セグメントを追加します。5番目のセグメントは、closePathメソッドの呼び出しによって追加されます。このメソッドは、最後の点(0, 40)と最初の点(100, 0)を接続します。

**リスト 3-1**　五角形の作成

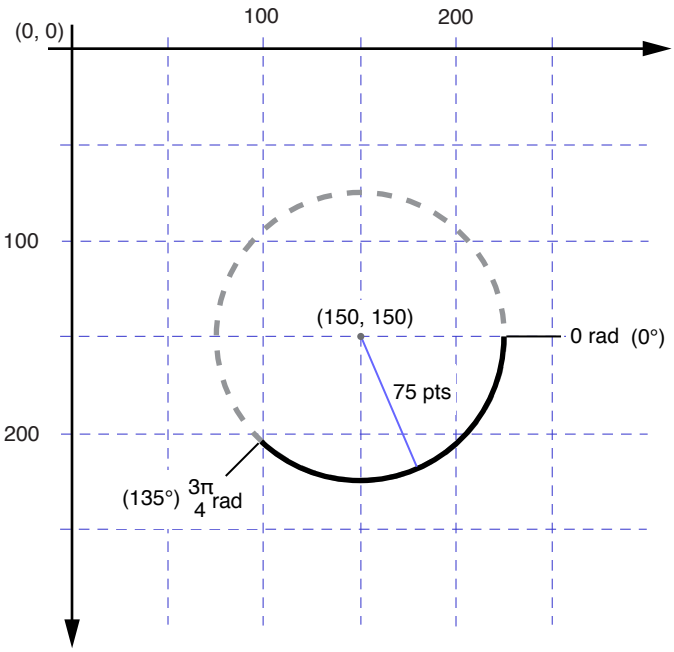
```
UIBezierPath*    aPath = [UIBezierPath bezierPath];

// 図形の始点を設定する
[aPath moveToPoint:CGPointMake(100.0, 0.0)];

// 直線を描画する
[aPath addLineToPoint:CGPointMake(200.0, 40.0)];
[aPath addLineToPoint:CGPointMake(160, 140)];
[aPath addLineToPoint:CGPointMake(40.0, 140)];
[aPath addLineToPoint:CGPointMake(0.0, 40.0)];
[aPath closePath];
```

closePathメソッドを使用すると、図形を記述したサブパスを終了させるだけでなく、最初の点と最後の点の間に直線を引くことができます。これは、最後の直線を描画する必要なしに多角形を完成させる便利な方法です。

# パスへの弧の追加

UIBezierPathクラスでは、新しいパスオブジェクトを弧セグメントで初期化できます。bezierPathWithArcCenter:radius:startAngle:endAngle:clockwise:メソッドのパラメータは、希望する弧と、その弧の始点と終点を含む円を定義します。図 3-1に、弧の作成に必要な構成要素を示します。これには、この弧を定義する円と、それを指定するために使用する角度も含まれています。ここでは、弧を時計回りに作成しています（代わりに、反時計回りに弧を描画すると、この円の破線部分が描画されます）。この弧を作成するためのコードをリスト 3-2 （37 ページ）に示します。

**図 3-1** デフォルトの座標系での弧




**リスト 3-2** 弧パスの新規作成

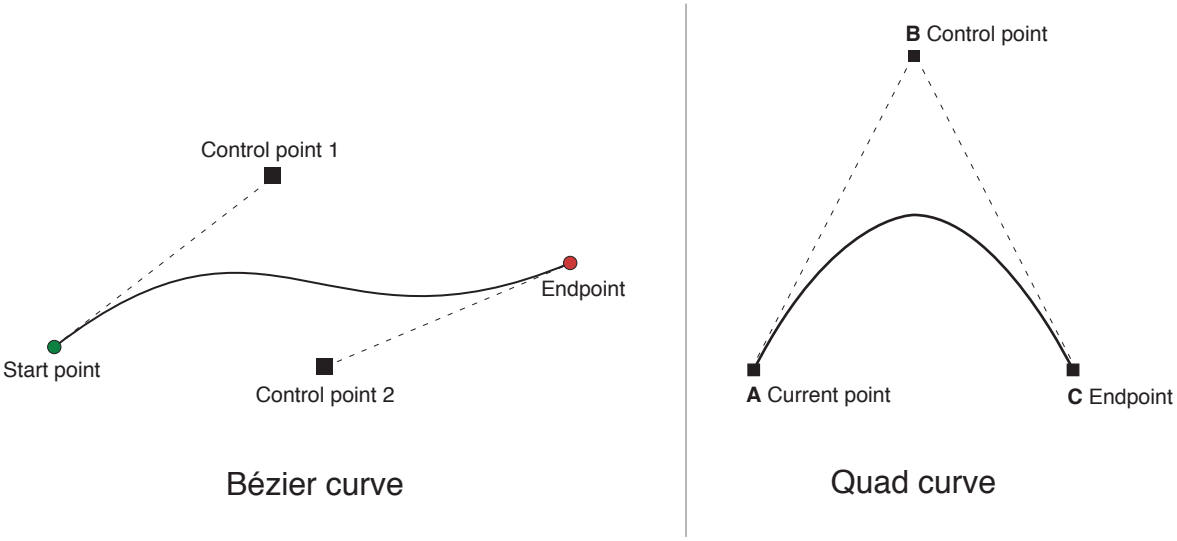
```
// piは、ほぼ3.14159265359に等しい
#define   DEGREES_TO_RADIANS(degrees)  ((pi * degrees)/ 180)

- (UIBezierPath*)createArcPath
{
   UIBezierPath* aPath = [UIBezierPath bezierPathWithArcCenter:CGPointMake(150,
 150)
                          radius:75
                          startAngle:0
                          endAngle:DEGREES_TO_RADIANS(135)
                          clockwise:YES];
   return aPath;
}
```

パスの途中に弧セグメントを組み込みたい場合は、そのパスオブジェクトの`CGPathRef`データ型を直接変更しなければなりません。Core Graphicsの関数を使用してパスを変更する方法の詳細については、「Core Graphicsの関数を使用したパスの変更」（39 ページ）を参照してください。

# パスへの曲線の追加

`UIBezierPath`クラスは、2次および3次のベジェ曲線をパスに追加する機能をサポートしています。曲線セグメントは、現在の点から始まり、指定した点で終了します。曲線の形状は、始点と終点と1つ以上の制御点の間の接線を使用して定義されます。図 3-2に、2種類の曲線の近似値と、制御点と曲線形状の関係を示します。各セグメントの正確な曲率には、すべての点の間の複雑な数学的関係が関与しています。これについては、オンラインドキュメントおよびWikipediaに詳しく記述されています。

**図 3-2** パス内の曲線セグメント




曲線をパスに追加するには、次のメソッドを使用します。

- **三次曲線：** `addCurveToPoint:controlPoint1:controlPoint2:`
- **二次曲線：** `addQuadCurveToPoint:controlPoint:`

曲線は、パスの現在の点によって異なるため、上記のいずれかのメソッドを呼び出す前に、現在の点を設定しなければなりません。曲線が完成すると、現在の点は、新たに指定した終点に更新されます。

# 楕円パスと矩形パスの作成

楕円と矩形は、曲線と直線のセグメントを組み合わせて作成される一般的なタイプのパスです。`UIBezierPath`クラスには、楕円や矩形のパスを作成するための簡易メソッドとして`bezierPathWithRect:`と`bezierPathWithOvalInRect:`があります。これらのメソッドはどちらも、パスオブジェクトを新規に作成して、それを指定された形状で初期化します。メソッドから返されたパスオブジェクトをすぐに使用することも、必要に応じてそれにいくつかの形状を追加することもできます。

既存のパスオブジェクトに矩形を追加するには、ほかの多角形を追加するときと同様に、`moveToPoint:`、`addLineToPoint:`、および`closePath`の各メソッドを使用しなければなりません。矩形の最後の辺に`closePath`メソッドを使用すると、パスの最後の直線を追加するのと同時に、この矩形サブパスの終了を示すことができるので便利です。

既存のパスに楕円を追加する場合は、Core Graphicsを使用する方法が最も簡単です。`addQuadCurveToPoint:controlPoint:`を使用して楕円の曲線を近似することもできますが、`CGPathAddEllipseInRect`関数の方が、はるかに簡単に使用できて正確です。詳細については、「Core Graphicsの関数を使用したパスの変更」（39 ページ）を参照してください。

# Core Graphicsの関数を使用したパスの変更

UIBezierPathクラスは、実際にはCGPathRefデータ型とそのパスに関連付けられている描画属性をラップしたラッパーに過ぎません。通常は、UIBezierPathクラスのメソッドを使用して直線や曲線のセグメントを追加しますが、このクラスは、基盤となるパスデータ型を直接変更できるCGPathプロパティも公開しています。Core Graphicsフレームワークの関数を使用してパスを作成したい場合は、このプロパティを使用します。

UIBezierPathオブジェクトに関連付けられているパスを変更するには2つの方法があります。Core Graphicsの関数を使用してパス全体を変更する方法、およびCore Graphicsの関数とUIBezierPathのメソッドを組み合わせて使用する方法の2つです。Core Graphics呼び出しを使用してパス全体を変更する方が、いくらか簡単です。可変のCGPathRefデータ型を作成し、パス情報の変更に必要な関数を呼び出します。それが終了したら、そのパスオブジェクトを、対応するUIBezierPathオブジェクトに割り当てます。そのコード例をリスト 3-3に示します。

**リスト 3-3**　新規のCGPathRefをUIBezierPathオブジェクトに割り当てる

```
// パスデータを作成する
CGMutablePathRef cgPath = CGPathCreateMutable();
CGPathAddEllipseInRect(cgPath, NULL, CGRectMake(0, 0, 300, 300));
CGPathAddEllipseInRect(cgPath, NULL, CGRectMake(50, 50, 200, 200));

// ここでUIBezierPathオブジェクトを作成する
UIBezierPath* aPath = [UIBezierPath bezierPath];
aPath.CGPath = cgPath;
aPath.usesEvenOddFillRule = YES;

// UIBezierPathオブジェクトに割り当てた後は、
// CGPathRefデータ型を安全に解放できる
CGPathRelease(cgPath);
```

Core Graphicsの関数とUIBezierPathのメソッドを組み合わせる方法を選んだ場合は、これら2つの間でパス情報を注意深くやり取りしなければなりません。UIBezierPathオブジェクトは、基盤となるCGPathRefデータ型を内部に所有しているため、単純にそのデータ型を取得して直接変更を加えることはできません。代わりに、可変のコピーを作成して、そのコピーを変更してから、CGPathプロパティに戻さなければなりません。そのコード例をリスト 3-4に示します。

**リスト 3-4**　Core Graphics呼び出しとUIBezierPath呼び出しの併用

```
UIBezierPath*    aPath = [UIBezierPath bezierPathWithOvalInRect:CGRectMake(0,
0, 300, 300)];

// CGPathRefを取得し、その可変バージョンを作成する
CGPathRef cgPath = aPath.CGPath;
CGMutablePathRef  mutablePath = CGPathCreateMutableCopy(cgPath);

// パスを変更して、それをUIBezierPathオブジェクトに戻す
CGPathAddEllipseInRect(mutablePath, NULL, CGRectMake(50, 50, 200, 200));
aPath.CGPath = mutablePath;

// パスの可変コピーを解放する
CGPathRelease(mutablePath);
```

# ベジエパスオブジェクトのコンテンツのレンダリング

UIBezierPathオブジェクトを作成したら、そのオブジェクトのstrokeメソッドやfillメソッドを使用して、パスを現在のグラフィックスコンテキストにレンダリングします。ただし、これらのメソッドを呼び出す前に、パスが正しく描画されるように、通常は次のような作業を行います。

- UIColorクラスのメソッドを使用して希望のストローク色と塗りつぶし色を設定する。
- ターゲットビュー内の希望の位置に図形を配置する。

  原点(0, 0)を起点としてパスを作成した場合は、現在の描画コンテキストに適切なアフィン変換を適用できます。たとえば、点(10, 10)を始点とする図形を描画するには、CGContextTranslateCTM関数を呼び出して、水平方向と垂直方向の両方の変換値として10を指定します。（パスオブジェクトの点を調整するよりも）グラフィックスコンテキストを調整する方が好まれます。以前のグラフィックスの状態を保存しておけば、それを復元することで簡単に変更を元に戻せるからです。
- パスオブジェクトの描画属性を更新する。UIBezierPathインスタンスの描画属性は、パスをレンダリングするときに、グラフィックスコンテキストに関連付けられている値よりも優先されます。

リスト3-5に、カスタムビューに楕円を描画するdrawRect:メソッドの簡単な実装を示します。この楕円に接する矩形の左上角は、このビューの座標系の点(50, 50)にあります。塗りつぶし操作はパスの境界線上まで塗りつぶしを行うため、このメソッドでは、境界線を描く前に塗りつぶしを実行します。これによって、境界線の半分が塗りつぶし色によって隠れてしまうのを防止できます。

**リスト 3-5**　ビューにパスを描画する

```
- (void)drawRect:(CGRect)rect
{
    // 描画するための楕円を作成する
    UIBezierPath* aPath = [UIBezierPath bezierPathWithOvalInRect:
                                CGRectMake(0, 0, 200, 100)];

    // レンダリング色を設定する
    [[UIColor blackColor] setStroke];
    [[UIColor redColor] setFill];

    CGContextRef aRef = UIGraphicsGetCurrentContext();

    // 図形の後に描画するコンテンツがある場合は、
    // 変換を行う前に、現在の状態を保存する
    //CGContextSaveGState(aRef);

    // ビューの原点を一時的に調整する。この楕円は
    // 新しい原点に基づいて描画される
    CGContextTranslateCTM(aRef, 50, 50);

    // 必要であれば、描画オプションを調整する
    aPath.lineWidth = 5;

    // 塗りつぶし色によって境界線が隠れてしまわないように
    // 境界線を描く前にパスを塗りつぶす
    [aPath fill];
```

```
    [aPath stroke];

    // その他のコンテンツを描画する前に、グラフィックスの状態を復元する
    //CGContextRestoreGState(aRef);
}
```

# パス上でのヒット検出

パスの塗りつぶし領域でタッチイベントが発生したかどうかを判断するには、`UIBezierPath`の`containsPoint:`メソッドを使用します。このメソッドは、パスオブジェクト内のすべての閉じたサブパスに対して、指定された点をテストし、これらのサブパスの上または内部にその点が存在する場合は`YES`を返します。

**重要：** `containsPoint:`メソッドとCore Graphicsのヒットテスト関数は、閉じたパスに対してのみ動作します。開いたサブパスのヒットに対しては、常に`NO`を返します。開いたサブパスに対してヒット検出を実行したい場合は、そのパスオブジェクトのコピーを作成して、開いたサブパスを閉じてから点をテストします。

パスの（塗りつぶし領域ではなく）ストローク部分のヒットテストを実行したい場合は、Core Graphicsを使用します。`CGContextPathContainsPoint`関数を使用すると、現在グラフィックスコンテキストに割り当てられているパスの塗りつぶし部分、またはストローク部分のいずれかの領域を対象に点をテストできます。リスト 3-6に、指定した点が指定したパスと交わるかどうかをテストするメソッドを示します。*`inFill`*パラメータを使用して、点のテストをパスの塗りつぶし部分に対して行うか、ストローク部分に対して行うかを、呼び出し側で指定できます。ヒット検出が成功するためには、呼び出し側から渡されたパスに、1つ以上の閉じたサブパスが含まれていなければなりません。

**リスト 3-6**　パスオブジェクトに対する点のテスト

```
- (BOOL)containsPoint:(CGPoint)point onPath:(UIBezierPath*)path
inFillArea:(BOOL)inFill
{
   CGContextRef context = UIGraphicsGetCurrentContext();
   CGPathRef cgPath = path.CGPath;
   BOOL    isHit = NO;

   // 使用する描画モードを決定する。デフォルトでは
   // パスのストローク部分に対するヒットが検出される
   CGPathDrawingMode mode = kCGPathStroke;
   if (inFill)
   {
      // 代わりに、パスの塗りつぶし領域でのヒットを検出する
      if (path.usesEvenOddFillRule)
         mode = kCGPathEOFill;
      else
         mode = kCGPathFill;
   }

   // パスを後で削除できるように
   // グラフィックスの状態を保存する
   CGContextSaveGState(context);
   CGContextAddPath(context, cgPath);
```

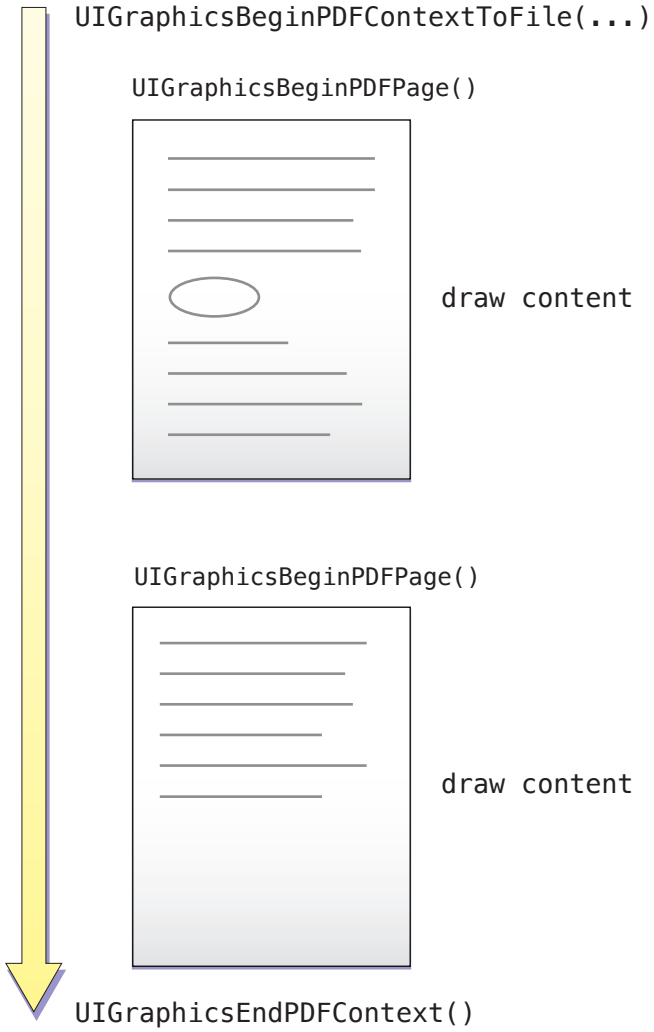
```
    // ヒット検出を実行する
    isHit = CGContextPathContainsPoint(context, point, mode);

    CGContextRestoreGState(context);

    return isHit;
}
```

# PDFコンテンツの生成

UIKitフレームワークでは、ネイティブの描画コードを使用してPDFコンテンツを生成する関数セットを提供しています。これらの関数を利用すると、PDFファイルやPDFデータオブジェクトをターゲットにしたグラフィックスコンテキストを作成できます。そして、画面に描画する際に使用するのと同じUIKitやCore Graphicsの描画ルーチンを使用して、このグラフィックスコンテキストに描画できます。PDFには任意の数のページを作成できます。描画が終了すると、PDFバージョンの描画結果が残ります。

図4-1に、ローカルのファイルシステムにPDFファイルを作成するためのワークフローを示します。`UIGraphicsBeginPDFContextToFile`関数は、PDFコンテキストを作成して、それをファイル名に関連付けます。コンテキストを作成したら、`UIGraphicsBeginPDFPage`関数を使用して、最初のページを開きます。ページを開いたら、そのコンテンツの描画を開始できます。新しいページを作成するには、単に`UIGraphicsBeginPDFPage`を再度呼び出してから、描画を開始します。描画が終了したら、`UIGraphicsEndPDFContext`関数を呼び出して、グラフィックスコンテキストを閉じ、描画結果データをPDFファイルに書き込みます。

**図 4-1** PDF文書を作成するためのワークフロー




以降の各セクションでは、PDF作成の手順について、例を使ってさらに詳しく説明します。PDFコンテンツを作成するために使用する関数の詳細については、『*UIKit Function Reference*』を参照してください。

# PDFコンテキストの作成と設定

PDFグラフィックスコンテキストは、`UIGraphicsBeginPDFContextToData`関数または`UIGraphicsBeginPDFContextToFile`関数のいずれかを使用して作成します。これらの関数は、グラフィックスコンテキストを作成して、それをPDFデータの保存先に関連付けます。`UIGraphicsBeginPDFContextToData`関数の場合、保存先はこの関数に渡される`NSMutableData`オブジェクトです。一方、`UIGraphicsBeginPDFContextToFile`関数の場合、保存先はアプリケーションのホームディレクトリ内のファイルです。

PDF文書は、ページ単位の構造を使用してコンテンツを編成します。この構造には、どのような描画を実行する場合にも次の2つの制限があります。

- 任意の描画コマンドを発行する前に、開いているページが1つ存在する必要がある。
- 各ページのサイズを指定する必要がある。

PDFグラフィックスコンテキストを作成するために使用する関数では、デフォルトのページサイズを指定することはできますが、これらの関数が自動的にページを開くわけではありません。コンテキストを作成したら、`UIGraphicsBeginPDFPage`関数または`UIGraphicsBeginPDFPageWithInfo`関数のいずれかを使用して、明示的に新しいページを開く必要があります。また、新しいページを作成したい場合は、毎回これらの関数のいずれかを再度呼び出して、新しいページの開始を指示しなければなりません。`UIGraphicsBeginPDFPage`関数は、デフォルトのサイズを使用してページを作成します。一方、`UIGraphicsBeginPDFPageWithInfo`関数を利用すると、ページサイズやその他の属性をカスタマイズできます。

描画が終了したら、`UIGraphicsEndPDFContext`を呼び出して、PDFグラフィックスコンテキストを閉じます。この関数は、最後のページを閉じて、コンテキストの作成時に指定されたファイルやデータオブジェクトにPDFコンテンツを書き込みます。また、この関数は、PDFコンテキストをグラフィックスコンテキストのスタックから削除します。

リスト4-1に、テキストビュー内のテキストからPDFファイルを作成するために、アプリケーションが使用する処理ループを示します。PDFコンテキストを設定したり管理したりするための3つの関数呼び出し以外は、ほとんどのコードは、必要なコンテンツの描画に関係するものです。`textView`メンバ変数は、必要なテキストが含まれている`UITextView`オブジェクトを指します。このアプリケーションでは、Core Textフレームワーク（具体的には、`CTFramesetterRef`データ型）を使用して、テキストのレイアウトや連続するページの管理を行います。独自の`renderPageWithTextRange:andFramesetter:`メソッドと`drawPageNumber:`メソッドの実装を、リスト 4-2 （47 ページ）に示します。

**リスト 4-1**　PDFファイルの新規作成

```
- (IBAction)savePDFFile:(id)sender
{
    // Core Text Framesetterを使用してテキストを準備する
    CFAttributedStringRef currentText = CFAttributedStringCreate(NULL,
(CFStringRef)textView.text, NULL);
    if (currentText) {
        CTFramesetterRef framesetter =
CTFramesetterCreateWithAttributedString(currentText);
        if (framesetter) {

            NSString* pdfFileName = [self getPDFFileName];
            // 612&#215;792のデフォルトのページサイズを使用してPDFコンテキストを作成する
            UIGraphicsBeginPDFContextToFile(pdfFileName, CGRectZero, nil);

            CFRange currentRange = CFRangeMake(0, 0);
            NSInteger currentPage = 0;
            BOOL done = NO;

            do {
                // 新規ページの開始を指定する
               UIGraphicsBeginPDFPageWithInfo(CGRectMake(0, 0, 612, 792), nil);

                // 各ページの下端にページ番号を描画する
                currentPage++;
                [self drawPageNumber:currentPage];
```

```
                // 現在のページをレンダリングして、現在の範囲を
                // 次のページの先頭を指すように更新する
                currentRange = [self renderPageWithTextRange:currentRange
andFramesetter:framesetter];

                // テキストの終わりに達したら、ループを終了する
                if (currentRange.location ==
CFAttributedStringGetLength((CFAttributedStringRef)currentText))
                    done = YES;
            } while (!done);

            // PDFコンテキストを閉じて、コンテンツを書き出す
            UIGraphicsEndPDFContext();

            // framesetterを解放する
            CFRelease(framesetter);

        } else {
            NSLog(@"Could not create the framesetter needed to lay out the
atrributed string.");
        }
        // 属性付き文字列を解放する
        CFRelease(currentText);
    } else {
        NSLog(@"Could not create the attributed string for the framesetter");
    }
}
```

# PDFページの描画

PDFの描画はすべて、ページのコンテキスト内で実行されなければなりません。どのPDF文書も少なくとも1つのページがあり、多くの場合、複数のページがあります。新しいページの開始を指定するには、`UIGraphicsBeginPDFPage`関数または`UIGraphicsBeginPDFPageWithInfo`関数を呼び出します。これらの関数は、（すでにページが開いている場合は）前のページを閉じて、新しいページを作成し、そこに描画できるようにします。`UIGraphicsBeginPDFPage`は、デフォルトのサイズを使用して新しいページを作成します。一方、`UIGraphicsBeginPDFPageWithInfo`関数を利用すると、ページサイズや、PDFページのその他の属性をカスタマイズできます。

ページを作成したら、その後のすべての描画コマンドは、PDFグラフィックスコンテキストに捕捉されて、PDFコマンドに変換されます。ページには、アプリケーションのカスタムビューに描画するときと同様に、テキスト、ベクトル図形、画像など、何でも描画できます。発行した描画コマンドは、PDFコンテキストに捕捉され、PDFデータに変換されます。ページ上でのコンテンツの配置は、完全にデベロッパに任されていますが、そのページの境界矩形内で行わなければなりません。

リスト 4-2に、PDFページ内にコンテンツを描画するために使われる2つのカスタムメソッドを示します。`renderPageWithTextRange:andFramesetter:`メソッドは、Core Textを使用して、ページに合ったテキストフレームを作成し、そのフレーム内にテキストを配置します。テキストを配置したら、現在のページの終わりと次のページの始まりを表す新しい範囲を返します。`drawPageNumber:`メソッドは、`NSString`の描画機能を使用して、各PDFページの下端にページ番号文字列を描画します。

**リスト 4-2**　ページ単位のコンテンツの描画

```
// Core Textを使用して、ページ上のフレームにテキストを描画する
- (CFRange)renderPage:(NSInteger)pageNum withTextRange:(CFRange)currentRange
                 andFramesetter:(CTFramesetterRef)framesetter
{
   // グラフィックスコンテキストを取得する
   CGContextRef    currentContext = UIGraphicsGetCurrentContext();

   // テキスト行列を既知の状態にする。これによって
   // 古い拡大縮小倍率が残っていないことが保証される
   CGContextSetTextMatrix(context, CGAffineTransformIdentity);

   // テキストを囲むパスオブジェクトを作成する。テキストの周囲の
   // マージンを72ポイントに設定する
   CGRect    frameRect = CGRectMake(72, 72, 468, 648);
   CGMutablePathRef framePath = CGPathCreateMutable();
   CGPathAddRect(framePath, NULL, frameRect);

   // レンダリングを実行するフレームを取得する
   // currentRange変数は、開始位置だけを指定する。framesetterは
   // フレームに入る分量のテキストを配置する
   CTFrameRef frameRef = CTFramesetterCreateFrame(framesetter, currentRange,
framePath, NULL);
   CGPathRelease(framePath);

   // Core Textは左下角から上に向かって描画する。
   // このため、描画する前に現在の変換を反転する。
   CGContextTranslateCTM(currentContext, 0, 792);
   CGContextScaleCTM(currentContext, 1.0, -1.0);

   // フレームを描画する
   CTFrameDraw(frameRef, currentContext);

   // 描画結果に基づいて、現在の範囲を更新する
   currentRange = CTFrameGetVisibleStringRange(frameRef);
   currentRange.location += currentRange.length;
   currentRange.length = 0;
   CFRelease(frameRef);

   return currentRange;
}

- (void)drawPageNumber:(NSInteger)pageNum
{
   NSString* pageString = [NSString stringWithFormat:@"Page %d", pageNum];
   UIFont* theFont = [UIFont systemFontOfSize:12];
   CGSize maxSize = CGSizeMake(612, 72);

   CGSize pageStringSize = [pageString sizeWithFont:theFont
                                       constrainedToSize:maxSize
                                       lineBreakMode:UILineBreakModeClip];
   CGRect stringRect = CGRectMake(((612.0 - pageStringSize.width) / 2.0),
                                  720.0 + ((72.0 - pageStringSize.height) / 2.0)
 ,
                                    pageStringSize.width,
                                    pageStringSize.height);
```

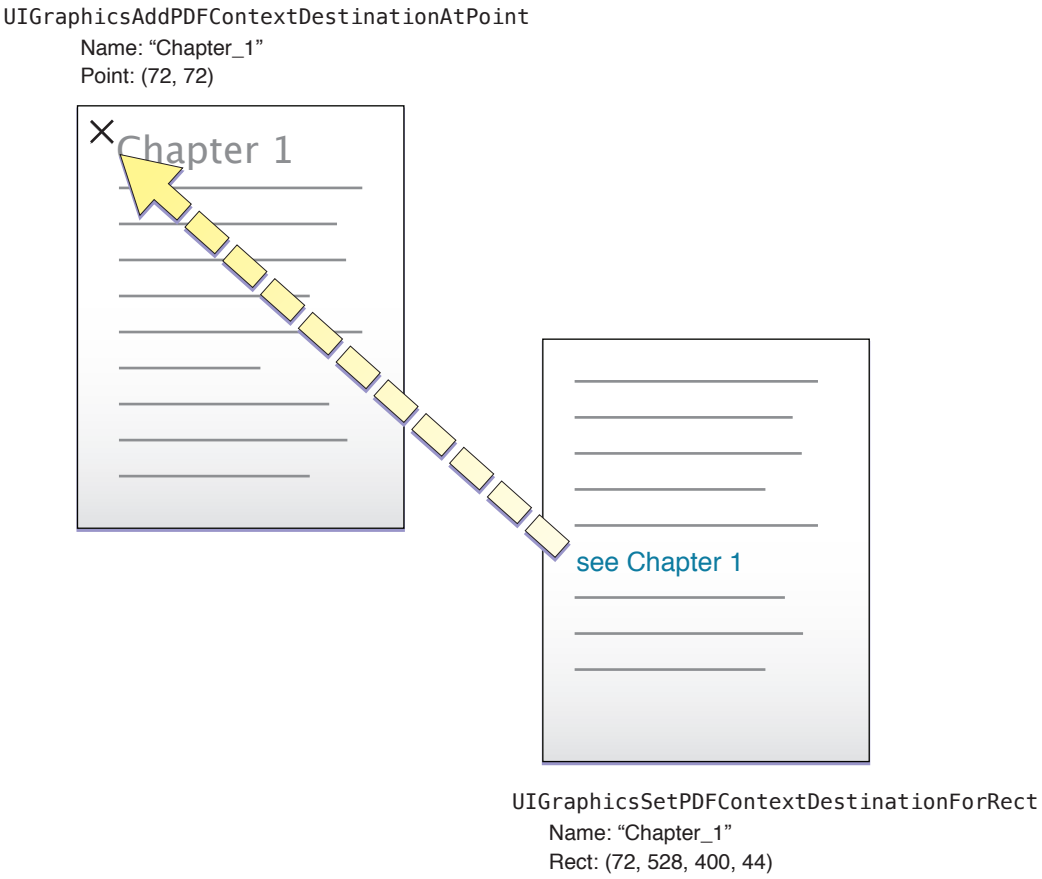
```
    [pageString drawInRect:stringRect withFont:theFont];
}
```

# PDFコンテンツ内でのリンクの作成

コンテンツを描画するだけでなく、同じPDFファイル内の別のページや外部URLへのリンクを含めることもできます。1つのリンクを作成するには、PDFページにソース矩形とリンク先を追加しなければなりません。リンク先の属性の1つは、そのリンクの一意の識別子としての役割を果たす文字列です。特定のリンク先へのリンクを作成するには、ソース矩形を作成するときに、そのリンク先の一意の識別子を指定します。

新しいリンク先をPDFコンテンツに追加するには、`UIGraphicsAddPDFContextDestinationAtPoint`関数を使用します。この関数は、名前付きのリンク先を現在のページの特定の位置に関連付けます。そのリンク先へのリンクを作成するには、`UIGraphicsSetPDFContextDestinationForRect`関数を使用して、リンク用のソース矩形を指定します。図 4-2に、PDF文書のページに適用した場合の、これら2つの関数呼び出しの関係を示します。「see Chapter 1」というテキストを囲む矩形をタップすると、それに対応するリンク先（Chapter 1の先頭）に移動できます。

**図 4-2**　リンク先とジャンプ元の作成



文書内へのリンクを作成できるほかに、`UIGraphicsSetPDFContextURLForRect`関数を使用して、文書の外部にあるコンテンツへのリンクも作成できます。この関数を使用してリンクを作成するときは、あらかじめリンク先を作成しておく必要はありません。必要なのは、この関数を使用して、ターゲットURLと、現在のページ上のソース矩形を指定することだけです。

# 画像

機能と美しさの両方の理由から、画像はアプリケーションのユーザインターフェイスに浸透している要素です。画像は、アプリケーションの重要な差別化要因になり得ます。

アプリケーションで使用する多くの画像（起動画像、アプリケーションアイコンなど）は、アプリケーションのメインバンドルにファイルとして保存されます。デバイスタイプ（iPad、iPhoneおよびiPod touch）に固有の起動画像やアイコンを用意したり、高解像度ディスプレイ用に最適化された画像を用意することができます。これらのバンドルされる画像ファイルの完全な説明については、『*iOS App Programming Guide*』の「Advanced App Tricks」および「Implementing Application Preferences」を参照してください。「画像リソースファイルの更新」（28 ページ）では、画像ファイルに高解像度画面との互換性を持たせるための調整について説明します。

## 画像のためのシステムサポート

iOSのより下位レベルのシステムフレームワークと同様に、UIKitフレームワークも、画像の作成、アクセス、描画、書き込み、および操作のための幅広い機能を提供しています。

### UIKitの画像クラスと関数

UIKitフレームワークには、何らかの形で画像に関わる3つのクラスと1つのプロトコルが含まれています。

`UIImage`
: このクラスのオブジェクトはUIKitフレームワークでの画像を表します。さまざまなソース（ファイルやQuartz画像オブジェクトなど）から画像を作成できます。このクラスのメソッドを利用すると、さまざまなブレンドモードや不透明度を使用して、画像を現在のグラフィックスコンテキストに描画できます。

  `UIImage`は、必要な変換を処理を自動的に行います。たとえば、（高解像度ディスプレイを考慮して）適切な倍率を適用したり、Quartz画像が指定された場合に、（原点が左上角にある）UIKitのデフォルト座標系に合うように画像の座標系を修正するなどの処理です。

`UIImageView`
: このクラスのオブジェクトは、単一の画像を表示するか、一連の画像をアニメーション化するビューです。1つの画像がビューの唯一のコンテンツになる場合は、画像を描画する代わりに`UIImageView`オブジェクトを使用します。

`UIImagePickerController` と `UIImagePickerControllerDelegate`
: このクラスとプロトコルは、ユーザから提供される画像（写真）やムービーを取得する手段をアプリケーションに提供します。このクラスは、写真やムービーを選択したり作成するためのユーザインターフェイスを表示し、管理します。ユーザが写真を選択すると、このクラスは選択された`UIImage`オブジェクトをデリゲートに送ります。デリゲートには、プロトコルメソッドが実装されていなければなりません。

これらのクラスのほかにも、UIKitには、画像に関わるさまざまな作業を実行するために呼び出すことができる関数が定義されています。

- **画像をサポートするグラフィックスコンテキストに描画する。**`UIGraphicsBeginImageContext`関数が、オフスクリーンのビットマップグラフィックスコンテキストを作成します。このグラフィックスコンテキストに描画して、そこから`UIImage`オブジェクトを抽出できます（詳細については、「画像の作成の描画」（54 ページ）を参照）。
- **画像データを取得したりキャッシングする。**各`UIImage`オブジェクトが、ベースとなるCore Graphics画像オブジェクト（`CGImageRef`）を1つ持っており、デベロッパはそれに直接アクセスできます。そして、そのCore GraphicsオブジェクトをImage I/Oフレームワークに渡してデータを保存できます。`UIImagePNGRepresentation`関数や`UIImageJPEGRepresentation`関数を呼び出して、`UIImage`オブジェクトの画像データをPNG形式やJPEG形式に変換することもできます。データオブジェクトのバイトにアクセスして、画像データをファイルに書き出すこともできます。
- **画像をデバイス上のフォトアルバムに書き込む。**画像をデバイス上のフォトアルバムに入れるには、`UIImageWriteToSavedPhotosAlbum`関数を呼び出して、`UIImage`オブジェクトを渡します。

「画像の作成と描画」（54 ページ）では、これらのUIKitクラスと関数を使用する場合のシナリオを示します。

## その他の画像関連フレームワーク

UIKit以外のいくつかのシステムフレームワークを使用して、画像の作成、アクセス、変更、および書き込みを行うことができます。UIKitのメソッドや関数では画像関連の特定の作業を実行できないことがわかった場合は、より下位レベルのフレームワークの関数を利用して、目的の処理を実行できる場合があります。これらの関数の中には、Core Graphicsの画像オブジェクト（`CGImageRef`）を必要とするものもあります。`UIImage`オブジェクトのベースとなる`CGImageRef`オブジェクトには、`CGImage`プロパティを介してアクセスできます。

**注：** 画像関連の特定の作業を実行するUIKitのメソッドや関数が存在する場合は、下位レベルの関数ではなく、それを使用するべきです。

QuartzのCore Graphicsフレームワークは、下位レベルのシステムフレームワークの中で最も重要です。いくつかのCore Graphics関数は、UIKitの関数やメソッドに対応しています。たとえば、ビットマップグラフィックスコンテキストを作成して、そのコンテキストに描画できる関数もあれば、さまざまなソースから画像を作成できる関数もあります。ただし、Core Graphicsの方が画像処理のための選択肢がたくさんあります。Core Graphicsを利用すると、画像マスクの作成と適用、既存の画像の一部からの画像作成、色空間の適用、いくつかの追加の画像属性（1行あたりのバイト数、1ピクセルあたりのビット数、レンダリングインテントなど）へのアクセスができます。

Image I/Oフレームワークは、Core Graphicsと密接に関連しています。これを利用すると、アプリケーションはほとんどの画像ファイル形式（標準的なWeb形式、高ダイナミックレンジの画像、未加工のカメラデータなど）を読み書きできます。また、高速な画像エンコードやデコード、画像メタデータ、および画像キャッシングの機能もあります。

Assets Libraryは、「写真（Photos）」アプリケーションで管理されているアセットにアプリケーションからアクセスできるようにするフレームワークです。アセットは、画像表現（PNG、JPEGなど）またはURLとして取得できます。この画像表現やURLから、Core Graphicsの画像オブジェクトや未加工の画像データを取得できます。また、このフレームワークを利用すると、「保存された写真(Saved Photos)」アルバムに画像を書き込むこともできます。

## サポートされる画像形式

表 5-1に、iOSが直接サポートする画像形式の一覧を示します。これらの形式のうち、PNG形式がアプリケーションで使用する画像形式として最も推奨されます。一般に、UIKitがサポートする画像形式は、Image I/Oフレームワークがサポートする画像形式と同じです。

**表 5-1**　サポートされる画像形式

| 形式 | ファイル名拡張子 |
|---|---|
| Portable Network Graphic (PNG) | `.png` |
| Tagged Image File Format (TIFF) | `.tiff`, `.tif` |
| Joint Photographic Experts Group (JPEG) | `.jpeg`, `.jpg` |
| Graphic Interchange Format (GIF) | `.gif` |
| Windows Bitmap Format (DIB) | `.bmp`, `.BMPf` |
| Windowsアイコン形式 | `.ico` |
| Windowsカーソル | `.cur` |
| +XWindowビットマップ | `.xbm` |

# 画像品質の維持

ユーザインターフェイスに高品質の画像を提供することは、設計における優先事項です。画像は、複雑なグラフィックスを表示するとても効率的な方法です。効果的な場面では積極的に使用するべきです。アプリケーション用の画像を作成するときは、次のガイドラインを念頭に置くようにしてください。

- **PNG形式の画像を使用する。**PNG形式は、データ損失のない画像コンテンツを提供します。つまり、画像データをPNG形式で保存してから、それを読み込むとまったく同じピクセル値に戻すことができます。また、PNGの最適化された保存形式は、画像データをより高速に読み込めるように設計されています。PNGは、iOS用の画像形式としてよく使われています。
- **サイズ変更が不要となるよう画像を作成する。**ある特定サイズの画像を使用する予定のある場合は、対応する画像リソースをそのサイズで作成します。拡大縮小を行うと必要なCPUサイクルが増え、補間処理が必要となるため、大きめの画像を作成してから適切なサイズに縮小する方法はとらないようにしてください。可変サイズの画像を表示する必要のある場合は、その画像を異なるサイズで複数作成しておき、対象のサイズに比較的近い画像を縮小するようにします。

- **不透過なPNGファイルからアルファチャンネルを削除する。**PNG画像の各ピクセルが不透過な場合、アルファチャンネルを削除することにより、その画像を含むレイヤのブレンドが不要となります。これにより、画像の合成が大幅に簡素化され、描画パフォーマンスが改善します。

# 画像の作成と描画

iOSは、UIKitとCore Graphicsの両方のフレームワークを使用した画像のロードと表示をサポートします。画像の描画に使用するクラスと関数をどのように決めるかは、画像の使用目的によって異なります。ただし、コード内で画像を表示するときはできる限りUIKitのクラスを使用することをお勧めします。表 5-2に、考えられるいくつかの画像使用のシナリオと、それぞれの場合に推奨される処理方法を示します。

**表 5-2**　画像使用のシナリオ

| シナリオ | 推奨される処理方法 |
| --- | --- |
| 画像をビューのコンテンツとして表示 | `UIImageView`クラスを使用して画像を表示します。この方法は、ビューのコンテンツが画像だけの場合に使用できます。画像のビューの上にさらにビューのレイヤを追加して、ほかのコントロールやコンテンツを描画することもできます。 |
| ビューの一部分に、装飾として画像を表示 | `UIImage`クラスを使用して画像をロードし、表示します。 |
| ビットマップデータを画像オブジェクトに保存 | それには、UIKit関数やCore Graphics関数を使用します。詳細については、「ビットマップコンテキストへの描画」（55 ページ）を参照してください。 |
| 画像をJPEGファイルまたはPNGファイルとして保存 | `UIImage`オブジェクトを元の画像データから作成します。`UIImageJPEGRepresentation`関数または`UIImage-PNGRepresentation`関数を呼び出して`NSData`オブジェクトを取得し、取得したオブジェクトのメソッドを使用してデータをファイルに保存します。 |

次の例に、アプリケーションバンドルから画像をロードする方法を示します。画像をロードした後、この画像オブジェクトを使用して`UIImageView`オブジェクトを初期化できます。また、この画像を保存しておき、ビューの`drawRect:`メソッドの中で明示的に描画できます。

```
NSString* imagePath = [[NSBundle mainBundle] pathForResource:@"myImage"
ofType:@"png"];
UIImage* myImageObj = [[UIImage alloc] initWithContentsOfFile:imagePath];
```

ビューの`drawRect:`メソッドで画像を明示的に描画するときは、`UIImage`で利用可能ないずれの描画メソッドも使用できます。これらのメソッドでは、ビューのどこに画像を描画するかを指定できるため、描画の前に別途変換を作成し、適用する必要がありません。`anImage`というメンバ変数に、事前にロードした画像を保存したと仮定した場合、次の例ではその画像を、ビューの座標の(10, 10)の位置に描画します。

```
- (void)drawRect:(CGRect)rect
{
    // 描画する
```

```
    [anImage drawAtPoint:CGPointMake(10, 10)];
}
```

**重要：** CGContextDrawImage関数を使用してビットマップ画像を直接描画する場合、デフォルトでは、画像データはY軸に沿って反転します。これは、Quartzの画像が、左下角を原点とし、正の座標軸が原点から上方向と右方向に伸びる座標系を前提としているためです。描画の前に変換を適用することもできますが、より簡単かつ推奨されるQuartz画像の描画方法は、Quartz画像をUIImageオブジェクトでラップすることです。このオブジェクトでは、座標空間の違いが自動的に補正されます。Core Graphicsを使用した画像の作成と描画の詳細については、『*Quartz 2D Programming Guide*』を参照してください。

# ビットマップグラフィックスコンテキストへの描画

ビットマップ画像用のコンテキストを作成してそこに描画した後に、そのコンテキストから画像オブジェクトを取得できます。このテクニックはさまざまな目的で使用できます。たとえば、画像を縮小したり（サムネイルの作成など）、複数の画像を合成したり、描画操作の結果を静的な形式で保持するためなどです。

UIKitフレームワークまたはCore Graphicsフレームワークのいずれかの関数を使用して、ビットマップコンテキストに描画した後で、そこから画像を抽出します。UIKitでは、その手順は以下のようになります。

1. UIGraphicsBeginImageContextWithOptions関数またはUIGraphicsBeginImageContext関数を呼び出して、画像ベースの新しいグラフィックスコンテキストを作成します。

   1つの目の関数（iOS4.0で導入された）は、好んで呼び出される関数です。この関数は、ビットマップのサイズ、ビットマップが不透明かどうかを表すフラグ、ビットマップに適用する倍率の3つのパラメータを取ります。画像の一部を透過ピクセル（アルファチャンネル）にしたい場合は、不透明フラグをNOにしなければなりません。倍率パラメータの詳細については、「プログラミングによる高解像度ビットマップ画像の作成」（30 ページ）を参照してください。

2. 画像コンテンツをコンテキストに描画します。描画には、UIKitのメソッドや関数、またはCore Graphicsの関数を使用できます。

3. UIGraphicsGetImageFromCurrentImageContext関数を呼び出すと、描画した画像に基づくUIImageオブジェクトが生成され、返されます。このメソッドを呼び出した後も描画を継続して、追加画像を生成できます。

4. グラフィックスコンテキストを閉じるには、UIGraphicsEndImageContext関数を呼び出します。

リスト 5-1のメソッドでは、インターネットからダウンロードした画像を取得して、それをアプリケーションアイコンのサイズに縮小した画像ベースのビットマップコンテキストに描画します。次に、このビットマップデータからUIImageオブジェクトを取得して、それをインスタンス変数に代入します。ビットマップのサイズ（UIGraphicsBeginImageContextWithOptionsの第1パラメータ）と描画されるコンテンツのサイズ（imageRectのサイズ）は一致していなければなりません。コンテンツがビットマップより大きい場合は、コンテンツの一部が切り取られ、結果の画像には表示されません。

**リスト 5-1**　縮小画像をビットマップコンテキストに描画し、その結果の画像を取得する

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    UIImage *image = [[UIImage alloc] initWithData:self.activeDownload];
    if (image != nil && image.size.width != kAppIconHeight && image.size.height
 != kAppIconHeight) {
        CGRect imageRect = CGRectMake(0.0, 0.0, kAppIconHeight, kAppIconHeight);
         UIGraphicsBeginImageContextWithOptions(itemSize, NO, [UIScreen
mainScreen].scale);
         [image drawInRect:imageRect];
         self.appRecord.appIcon = UIGraphicsGetImageFromCurrentImageContext();
 // 戻り値のUIImage
         UIGraphicsEndImageContext();
    } else {
        self.appRecord.appIcon = image;
    }
    self.activeDownload = nil;
    [image release];
    self.imageConnection = nil;
    [delegate appImageDidLoad:self.indexPathInTableView];
}
```

Core Graphicsの関数を呼び出して、生成されたビットマップ画像のコンテンツを描画することもできます。その例として、リスト 5-2のコードでは、PDFページの縮小画像を描画します。`CGContextDrawPDFPage`を呼び出す前に、コードでは、描画された画像をUIKitのデフォルト座標系に合わせるために、グラフィックスコンテキストを反転させています。

**リスト 5-2**　Core Graphics関数を使用したビットマップコンテキストへの描画

```
// ほかのコードがある...
CGRect pageRect = CGPDFPageGetBoxRect(page, kCGPDFMediaBox);
pdfScale = self.frame.size.width/pageRect.size.width;
pageRect.size = CGSizeMake(pageRect.size.width*pdfScale,
pageRect.size.height*pdfScale);
UIGraphicsBeginImageContextWithOptions(pageRect.size, YES, pdfScale);
CGContextRef context = UIGraphicsGetCurrentContext();
// 最初に背景を白で塗りつぶす
CGContextSetRGBFillColor(context, 1.0,1.0,1.0,1.0);
CGContextFillRect(context,pageRect);
CGContextSaveGState(context);
// PDFページが正しく反転するようにコンテキストを反転する
CGContextTranslateCTM(context, 0.0, pageRect.size.height);
CGContextScaleCTM(context, 1.0, -1.0);
// PDFページが倍率に合った適切なサイズでレンダリングされるように、コンテキストを拡大縮小する
CGContextScaleCTM(context, pdfScale,pdfScale);
CGContextDrawPDFPage(context, page);
CGContextRestoreGState(context);
UIImage *backgroundImage = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
backgroundImageView = [[UIImageView alloc] initWithImage:backgroundImage];
// この後にほかのコードが続く...
```

ビットマップグラフィックスコンテキストでの描画全体にCore Graphicsを使用する場合は、`CGBitmapContextCreate`関数を使用して、コンテキストを作成し、それに画像コンテンツを描画します。描画が完了したら、`CGBitmapContextCreateImage`関数を使用し、そのビットマップコンテ

キストからCGImageRefを作成します。Core Graphicsの画像を直接描画したり、この画像を使用してUIImageオブジェクトを初期化することができます。完了したら、グラフィックスコンテキストに対してCGContextRelease関数を呼び出します。

# 印刷

iPhone、iPad、およびiPod touchデバイスのアプリケーションは、ローカルプリンタにコンテンツを印刷できます。印刷は、すべてのアプリケーションで実装できる重要な機能です。しかも、ほんの数行のコードで済むこともあります。印刷によってアプリケーションの価値が高まったという例は無数にあります。ほんの一例を以下に示します。

- 損害保険調査のアプリケーションで、事故現場から賠償金の支払い請求を印刷する。
- ドローアプリケーションでユーザが自分の作品を印刷する。

**注：** 印刷は、iOS 4.2からシステム機能として導入されました。印刷は、マルチタスクに対応したデバイスでのみサポートされます。

## 簡単かつ直感的に設計されたiOSでの印刷

印刷を行うには、ユーザは、通常、ナビゲーションバーやツールバーにあるボタンや、ユーザが印刷したいビューや選択項目に関連付けられたボタンをタップします。すると、アプリケーションは印刷オプションのビューを表示します。ユーザは、プリンタとさまざまな印刷オプションを選択し、印刷を要求します。アプリケーションは、コンテンツから印刷出力を生成するか、印刷可能なデータやファイルのURLを提供するよう要求されます。要求された印刷ジョブがスプールされ、アプリケーションに制御が戻ります。出力先のプリンタがビジーでない場合は、ただちに印刷が始まります。プリンタが印刷中の場合や、キュー内に印刷待ちのジョブがある場合は、この印刷ジョブは、キューの一番上に移動して印刷されるまで、iOSの印刷キューに留まります。

印刷のアーキテクチャについては、「iOSでの印刷の仕組み」（63 ページ）で詳しく説明します。

**用語に関する補足：** この章では、*印刷ジョブ*という用語が頻繁に登場します。印刷ジョブとは、印刷すべきコンテンツだけでなく、それを印刷する際に使われる情報（プリンタのID、印刷ジョブの名前、印刷の品質や向きなど）を含むひとまとまりの作業のことを言います。

### 印刷ユーザインターフェイス

印刷関連でユーザが最初に目にするのは、印刷ボタンです。印刷ボタンは、多くの場合、ナビゲーションバーやツールバーのバーボタン項目です。この印刷ボタンは、必然的に、アプリケーションが現在表示しているコンテンツに適用されます。つまり、ユーザがこのボタンをタップすると、アプリケーションはそのコンテンツを印刷しなければなりません。印刷ボタンを任意のカスタムボタンにすることもできますが、図 6-1に示すシステム項目アクションボタンを使用することをお勧めします。これは、`UIBarButtonSystemItemAction`定数で指定される`UIBarButtonItem`オブジェクトで、Interface Builderまたは`initWithBarButtonSystemItem:target:action:`を呼び出すことによって作成できます。

**図 6-1** 印刷に使われるシステム項目アクションボタン

ユーザが印刷ボタンをタップすると、アプリケーションのコントローラオブジェクトは、そのアクションメッセージを受信します。コントローラは、それに応答して、印刷の準備を行ってプリンタオプションビューを表示します。このオプションには、必ず、印刷先のプリンタ、印刷枚数、場合によっては印刷するページの範囲が含まれています。選択したプリンタが両面印刷可能な場合は、片面出力にするか両面出力にするかを選択できます。ユーザが印刷をしない場合は、オプションビューの領域外をタップするか（iPadの場合）、「キャンセル」ボタンをタップすると（iPhoneおよびiPod touchの場合）、プリンタオプションビューが消えます。

この種のユーザインターフェイスは、デバイスによって見え方が異なります。iPadの場合、UIKitフレームワークはオプションを含むPopoverビューを表示します（図 6-2を参照）。アプリケーションは、このビューを印刷ボタン、またはアプリケーションのユーザインターフェイスの任意の領域から現れるようにアニメーション化できます。

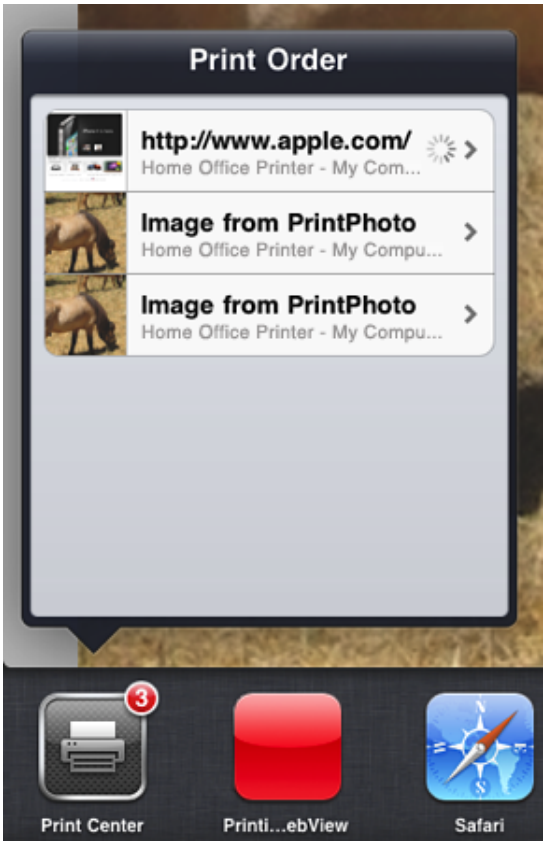
**図 6-2** プリンタオプションのPopoverビュー(iPad)


iPhoneおよびiPod touchデバイスの場合、UIkitは印刷オプションページを表示します。アプリケーションは、このページを画面の下端からスライドアップするようにアニメーション化できます。

**図 6-3** プリンタオプションページ(iPhone)

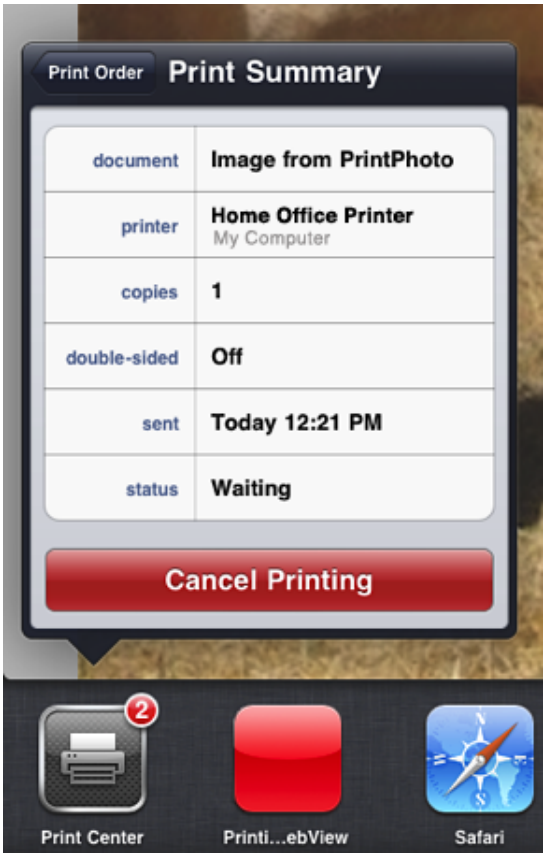

印刷ジョブが発行されて、それが印刷中、または印刷キューで待機中になると、ユーザは、「ホーム」ボタンをダブルタップしてマルチタスクUIの「Print Center」にアクセスすることにより、そのステータスを確認できます。「Print Center」（図6-4を参照）はバックグラウンドのシステムアプリケーションで、印刷キュー内のジョブの順番を表示します（現在印刷中のジョブも含む）。これは、印刷ジョブが実行されている間だけ利用できます。

**図 6-4** Print Center




ユーザは、Print Center内の1つの印刷ジョブをタップして、それについての詳細情報を表示したり（図 6-5）、キュー内の印刷中または待機中のジョブをキャンセルできます。

**図 6-5** Print Center：印刷ジョブの詳細



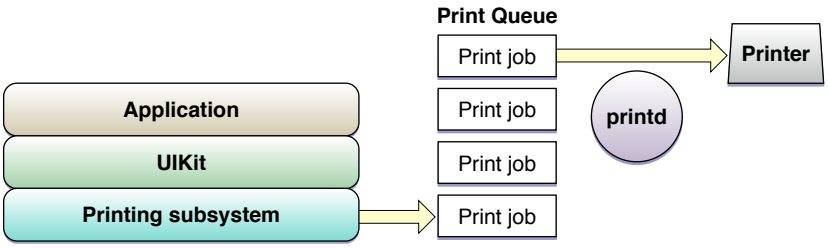

## iOSでの印刷の仕組み

アプリケーションは、UIKitの印刷APIを使用して、印刷ジョブの要素（印刷するコンテンツとその印刷ジョブに関連する情報を含む）を組み立てます。次に、「印刷ユーザインターフェイス」（59ページ）で説明したプリンタオプションビューを表示します。ユーザはオプションを選択して「印刷(Print)」をタップします。UIKitフレームワークが、印刷すべきコンテンツの描画をアプリケーションに依頼することもあります。UIKitは、アプリケーションがPDFデータとして描画した結果を記録します。その後UIKitは印刷用のデータを印刷サブシステムに渡します。

印刷システムは、いくつかの処理を実行します。UIKitが印刷データを印刷サブシステムに渡すと、サブシステムはこのデータをストレージに書き込みます（つまり、データをスプールします）。また、印刷ジョブについての情報を把握します。印刷システムは、印刷ジョブごとに印刷データとメタデータを結合して、FIFO（先入れ先出し）の印刷キューで管理します。1つのデバイス上の複数のアプリケーションが、複数の印刷ジョブを印刷サブシステムに発行することもできます。これらはすべて、印刷キューに配置されます。各デバイスには、発行元のアプリケーションや印刷先のプリンタに関係なく、すべての印刷ジョブ用に1つのキューがあります。

印刷ジョブがこのキューの一番上にくると、システムの印刷デーモン(`printd`)は、印刷先のプリンタの要件を考慮して、必要であれば、印刷データをプリンタが利用可能な形式に変換します。印刷システムは、「用紙なし(Out of Paper)」などのエラー状態を警告としてユーザに報告します。また、印刷ジョブの進捗状況をプログラムによって「Print Center」に報告します。「Print Center」は、この情報を印刷ジョブの「page 2 of 5」のように表示します。

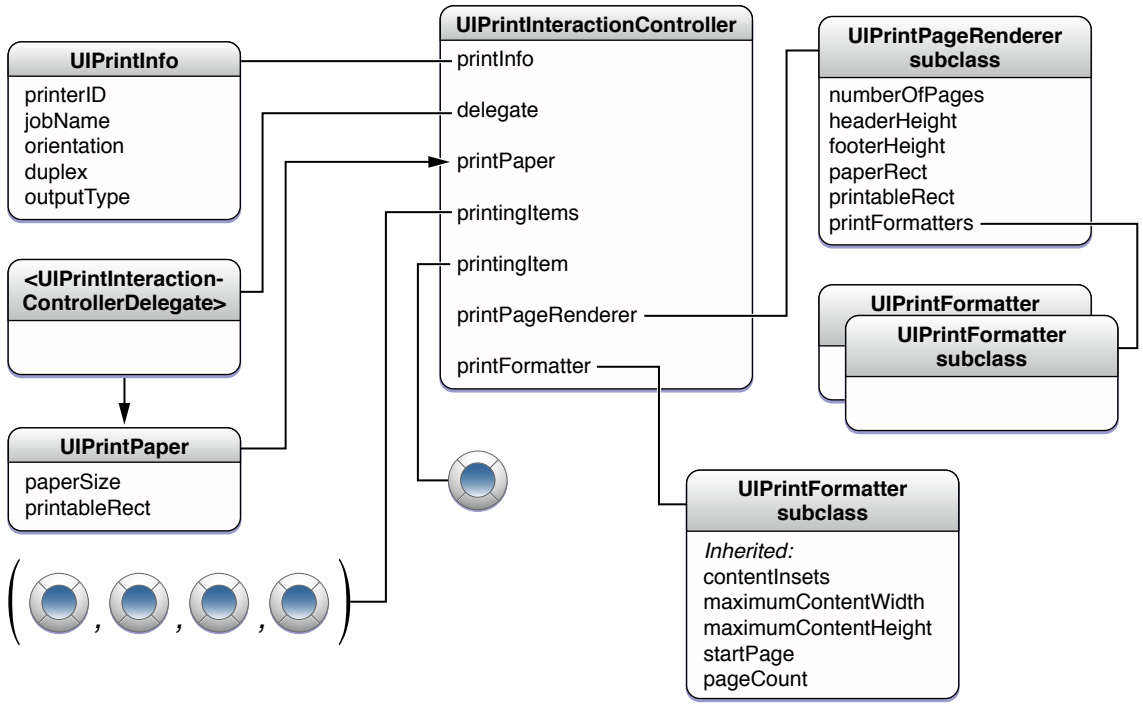
**図 6-6** 印刷のアーキテクチャ




# UIKitの印刷API

UIKitフレームワークは、印刷用のプログラムインターフェイスを発行します。このAPIには、8つのクラスと1つの形式プロトコルがあります。これらのクラスのオブジェクトと、このプロトコルを実装したデリゲートの間には、図 6-7に示すような実行時の関係があります。

**図 6-7** UIKitの印刷オブジェクトの関係




## 印刷用のクラスとプロトコルの概要

印刷用のUIKitクラスのオブジェクトには、特定の役割と仕事があります。

## Print Interaction Controllerとそのデリゲート

UIPrintInteractionControllerクラスの共有インスタンスは、アプリケーションの印刷ジョブを管理する中心的なオブジェクトです。このオブジェクトには、印刷ジョブについての情報(UIPrintInfo)と、用紙サイズと印刷されるコンテンツの領域(UIPrintPaper)が含まれています。また、UIPrintInteractionControllerDelegateプロトコルを採用したデリゲートオブジェクトへの参照も含まれています。印刷すべきコンテンツは、印刷可能な画像やPDF文書の配列、単一の画像やPDF文書、印刷フォーマッタ、またはページレンダラとして含まれています。印刷フォーマッタは、UIPrintFormatterサブクラスのインスタンスです。ページレンダラは、UIPrintPageRendererサブクラスのインスタンスです。

印刷ジョブが印刷可能になると、UIPrintInteractionControllerの3つのメソッドのうちの1つが呼び出されて、プリンタオプションビューが表示されます。詳細については、「印刷オプションの表示」（73 ページ）を参照してください。

UIPrintInteractionControllerDelegateプロトコルを採用するデリゲートは、3つの方法で印刷ジョブにアプリケーション固有の動作を追加できます。1つ目は、より統的な方法で、印刷オプションの表示や消去のときや、印刷ジョブの開始や終了のときに呼び出されるメソッドを実装する方法です。2つ目として、アプリケーションが文書サイズに関して特殊な要件を持つ場合は、デリゲートが希望の用紙サイズや画像領域を表すUIPrintPaperオブジェクトを返します。最後に、デリゲートはプリンタオプションビューを管理するコントローラの親のView Controllerを返すことができます。

## 印刷ジョブと用紙サイズの情報

UIPrintInfoクラスのオブジェクトは、印刷ジョブについての情報をカプセル化します。これには、選択されたプリンタのID、ジョブ名、印刷の向き、出力タイプ、および（プリンタが両面印刷をサポートする場合は）両面印刷モードが含まれています。ユーザは、選択されたプリンタや両面印刷の設定を変更できます。アプリケーションからUIPrintInfoプロパティの値が提供されない場合は、UIKitはデフォルト値を使用します。

UIPrintPaperクラスのインスタンスは、印刷ジョブで使用する用紙サイズと、選択されたプリンタで印刷可能な領域をカプセル化します。用紙サイズとその指定方法は、ロケールとプリンタに固有です。ほとんどのアプリケーションでは、印刷ジョブ用にUIKitによって作成されるデフォルトのUIPrintPaperオブジェクトを使用します。ただし、アプリケーションがコンテンツ領域に関して特殊な要件を持つ場合は、UIPrintInteractionControllerオブジェクトのデリゲートが希望の用紙サイズを表すUIPrintPaperオブジェクトを返すことができます。

UIKitは、印刷先のプリンタに基づく印刷可能なコンテンツのデフォルトの用紙サイズと、印刷ジョブの出力タイプを持っています（UIPrintInfoクラスには、outputTypeプロパティが宣言されており、これはアプリケーションから設定します）。出力タイプがUIPrintInfoOutputPhotoの場合は、デフォルトの用紙サイズは、4x6、A6、またはロケールに応じたその他の標準サイズです。出力タイプがUIPrintInfoOutputGeneralまたはUIPrintInfoOutputGrayscaleの場合は、デフォルトの用紙サイズは、USレター（8.5×11インチ）、A4、またはロケールに応じたその他の標準サイズです。

## ページレンダラ

印刷用に描画するコンテンツをアプリケーションから最大限に制御したい場合は、印刷にページレンダラを使用します。通常、ページレンダラは、UIPrintPageRendererのカスタムサブクラスのインスタンスです。これは、印刷可能なコンテンツの各ページ（または各ページの一部）を描画します。UIKitは、ページレンダラの描画結果を取得して、それをスプールファイルに記録します。この

スプールファイルは印刷システムに渡されます。この基底クラスには、ページ数を表すプロパティと、ページのヘッダとフッタの高さを表すプロパティが含まれています。また、このクラスでは、いくつかのメソッドが宣言されています。これらのメソッドをオーバーライドしてページの特定の部分（ヘッダ、フッタ、コンテンツ自体）を描画したり、ページレンダラと印刷フォーマッタによる描画を組み込むことができます。

**注：** 文書タイトルを繰り返し印刷したり、ページ数をインクリメントするなど、ヘッダ情報やフッタ情報を印刷する場合は、`UIPrintPageRenderer`のカスタムサブクラスを使用する必要があります。

ページレンダラは、1つ以上の印刷フォーマッタを利用して、印刷可能なコンテンツを描画してレイアウトします。特定の範囲のページ用に、ページレンダラに印刷フォーマッタを追加することもできます。UIPrintPageRendererのインスタンスを「そのまま」使用して、複数の印刷フォーマッタを接続できます。

### 印刷フォーマッタ

`UIPrintFormatter`クラスは、印刷フォーマッタの基底クラスです。印刷フォーマッタは、印刷されたコンテンツを複数のページにレイアウトするオブジェクトです。印刷フォーマッタは、ページ番号を割り付ける必要がある複雑なコンテンツに対して使用されます。このクラスには、印刷されるコンテンツの余白と、印刷開始ページを指定するインターフェイスが宣言されています。この情報が与えられると、印刷フォーマッタはレイアウトするページ数を計算します。

`UIViewPrintFormatter`は、複数ページにわたるビューのコンテンツを自動的にレイアウトするインスタンスを持つ`UIPrintFormatter`の具象サブクラスです。このクラスをサポートするために、`UIView`クラスには、それぞれ異なるシナリオを意図した2つのメソッドが追加されています。`UIWebView`、`UITextView`、`MKMapView`の各ビュークラスは、印刷用にコンテンツを描画する方法を認識しています。これらのクラスに対応する`UIViewPrintFormatter`インスタンスを取得するには、そのビューの`viewPrintFormatter`を呼び出します。そして、その印刷フォーマッタを使用してビューを印刷します。

`UISimpleTextPrintFormatter`は、プレーンテキスト文書を自動的に描画してレイアウトするインスタンスを持つ`UIPrintFormatter`の具象サブクラスです。これを使用して、テキストのグローバルプロパティ（フォント、色、配置、改行モードなど）を設定できます。

`UIMarkupTextPrintFormatter`は、HTML文書を自動的に描画してレイアウトするインスタンスを持つ`UIPrintFormatter`の具象サブクラスです。

## 印刷のためのアプローチと戦略

何かを印刷するためにアプリケーションが実行しなければならない処理は、印刷対象によって異なります。画像やPDF文書を印刷する場合は、非常に単純です。ヘッダとフッタを持ち、さまざまな種類のコンテンツから構成された長い文書を印刷する場合は、非常に複雑になる可能性があります。`UIPrintInteractionController`クラスのプログラミングインターフェイスは、そのために利用できるさまざまな手法を整理するのに役立ちます。このクラスには、印刷可能なコンテンツ（または印刷可能なコンテンツのソース）を保持するための4つのプロパティがあり、これらは互いに排他的です。つまり、これらのプロパティの1つに値を割り当てると、UIKitは、ほかのすべてのプロパティが`nil`であると見なします。

表 6-1のプロパティは、複雑さとアプリケーションの制御が少ない順に並べられています。

**表 6-1** オブジェクトまたは印刷可能なコンテンツのソースを表す`UIPrintInteractionController`のプロパティ

| プロパティ | 説明 |
| --- | --- |
| `printingItem` | 単一の`UIImage`、`NSData`、`NSURL`、または画像データやPDFデータを保持または参照する`ALAsset`オブジェクト。このオブジェクトは直接印刷できます。アプリケーションが追加処理を行う必要はありません。`NSURL`オブジェクトは、`file:`スキーム、`assets-library:`スキーム、または登録済みプロトコルを持つ`NSData`オブジェクトを返す任意のスキームを使用しなければなりません。ImageIOフレームワークがサポートする形式を持つ任意の画像を追加できます。詳細については、『*UIImage Class Reference*』の「Supported Image Formats」 in *View Programming Guide for iOS*を参照してください。`ALAsset`オブジェクトは、`ALAssetTypePhoto`型でなければなりません。このプロパティにオブジェクトを割り当てた場合、印刷の向きの設定は無効になります。 |
| `printingItems` | 画像データやPDFデータを保持または参照するオブジェクトの配列。これらのオブジェクトは直接印刷できます。これらは、`UIImage`、`NSData`、または`NSURL`のインスタンスでなければなりません。`NSURL`オブジェクトは、`file:`スキーム、`assets-library:`スキーム、または登録済みプロトコルを持つ`NSData`オブジェクトを返す任意のスキームを使用しなければなりません。ImageIOフレームワークがサポートする形式を持つ任意の画像を追加できます。詳細については、『*UIImage Class Reference*』の「Supported Image Formats」 in *View Programming Guide for iOS*を参照してください。`ALAsset`オブジェクトは、`ALAssetTypePhoto`型でなければなりません。<br>この配列を使用してコンテンツを印刷する場合、ユーザは印刷するページの範囲を設定できません。また、印刷の向きの設定は無効になります。 |
| `printFormatter` | 特定の種類の印刷可能コンテンツをレイアウトする機能を持つ`UIPrintFormatter`の具象クラスのインスタンス。たとえば、長いHTML文書を印刷する場合は（ただし、ヘッダやフッタの印刷は考慮しない）、初期化された`UIMarkupTextPrintFormatter`のインスタンスを割り当てることができます。 |
| `printPageRenderer` | 印刷可能なコンテンツの一部または全部をページごとに描画する`UIPrintPageRenderer`のカスタムクラスのインスタンス。ページレンダラは、印刷可能なコンテンツの特定のページまたは複数のページに対応する、1つ以上の印刷フォーマッタを持つことができます。 |

このようなさまざまな選択肢が利用できる場合、アプリケーションにとってどれが最適でしょうか？表 6-2に、このような決定に関わる要素を示します。

**表 6-2** アプリケーションコンテンツの印刷方法の決定

| 条件 | 決定 |
| --- | --- |
| アプリケーションが直接印刷可能なコンテンツ（画像またはPDF文書）にアクセスできる。 | `printingItem`プロパティまたは`printingItems`を使用する。 |

| 条件 | 決定 |
| --- | --- |
| 単一の画像やPDF文書を印刷したい。ユーザがページの範囲を選択できるようにしたい。 | `printingItem`プロパティを使用する。 |
| プレーンテキストやHTML文書を印刷したい（ただし、ヘッダやフッタなどの付加的なコンテンツは印刷しない）。 | `UISimpleTextPrintFormatter`オブジェクトまたは`UIMarkupTextPrintFormatter`オブジェクトを`printFormatter`プロパティに割り当てる。この印刷フォーマッタオブジェクトは、プレーンテキストまたはHTMLテキストで初期化しなければなりません。 |
| UIKitビューのコンテンツを印刷したい（ただし、ヘッダやフッタなどの付加的なコンテンツは印刷しない）。 | ビューから`UIViewPrintFormatter`オブジェクトを取得して、それを`printFormatter`プロパティに割り当てる。 |
| 印刷されるページにヘッダやフッタを繰り返し印刷したい。ページ番号を付ける可能性がある。 | `UIPrintPageRenderer`のカスタムサブクラスのインスタンスを`printPageRenderer`に割り当てる。このサブクラスは、ヘッダとフッタを描画するために必要なメソッドを実装していなければなりません。 |
| 混合のコンテンツまたはソースを印刷したい（たとえば、HTMLとカスタム描画）。 | `UIPrintPageRenderer`のカスタムサブクラスのインスタンスを`printPageRenderer`に割り当てる。場合によっては、コンテンツの特定ページをレンダリングするために、1つ以上の印刷フォーマッタを印刷レンダラに追加します。 |
| 印刷用の描画を最大限に制御したい。 | `UIPrintPageRenderer`のカスタムサブクラスのインスタンスを`printPageRenderer`に割り当てて、印刷するものすべてを描画する。 |

# 印刷のワークフロー

画像、ドキュメント、その他の印刷可能なアプリケーションコンテンツを印刷する際の一般的なワークフローは次のようになります。

1. `UIPrintInteractionController`の共有インスタンスを取得します。

2. （任意）`UIPrintInfo`オブジェクトを作成して、出力タイプ、ジョブ名、印刷の向きなどの属性を設定します。次に、そのオブジェクトを`UIPrintInteractionController`インスタンスの`printInfo`プロパティに割り当てます（ジョブ名の設定は、強くお勧めします）。

   print-infoオブジェクトを割り当てないと、UIKitは、印刷ジョブにデフォルトの属性を適用します（たとえば、ジョブ名はアプリケーション名になります）。

3. （任意）`delegate`プロパティにカスタムコントローラオブジェクトを割り当てます。このオブジェクトは、`UIPrintInteractionControllerDelegate`プロトコルを採用する必要があります。デリゲートは一連のタスクを実行できます。デリゲートは、印刷オプションが表示または消去されるときや、印刷ジョブが開始または終了するときに適切に応答できます。Print Interaction Controllerの親のビューコントローラを返します。また、このデリゲートは、プリンタ固有の用紙サイズとアプリケーションのニーズに適した印刷可能領域を表す`UIPrintPaper`オブジェクトを返すこともできます。

デリゲートが存在しない場合や、デリゲートが特殊なUIPrintPaperオブジェクトを返さない場合は、UIKitはデフォルトのページサイズと出力タイプ（アプリケーションが印刷するコンテンツの種類を表す）に応じた印刷可能領域を選択します。

4. UIPrintInteractionControllerインスタンスのプロパティに、次のいずれかを割り当てます。

    - 単一のNSData、NSURL、UIImage、またはPDFデータや画像データを保持または参照するALAssetオブジェクト。これはprintingItemプロパティに割り当てます。
    - 直接印刷可能な画像またはPDF文書の配列。これはprintingItemsプロパティに割り当てます。配列の要素は、NSData、NSURL、UIImage、またはPDFデータやサポートされる形式の画像を保持、参照、または表現するALAssetオブジェクトでなければなりません。
    - UIPrintFormatterオブジェクトはprintFormatterプロパティに割り当てます。印刷フォーマッタは、印刷可能なコンテンツのカスタムレイアウトを実行します。
    - UIPrintPageRendererオブジェクトはprintPageRendererプロパティに割り当てます。

    これらのプロパティの1つだけが、任意の印刷ジョブに対してnilでない値になります。これらのプロパティの詳細については、「印刷のためのアプローチと戦略」（66 ページ）を参照してください。

5. 前の手順でページレンダラを割り当てた場合は、通常、そのオブジェクトはUIPrintPageRendererのカスタムサブクラスのインスタンスです。このオブジェクトは、UIKitフレームワークからの要求があると、印刷用にコンテンツのページを描画します。また、印刷されるページのヘッダとフッタのコンテンツも描画します。カスタムページレンダラは、1つ以上の「描画用」メソッドをオーバーライドしなければなりません。また、少なくともコンテンツの一部（ヘッダとフッタを除く）を描画している場合は、その印刷ジョブに対するページ数を計算して返さなければなりません。（UIPrintPageRendererのインスタンスを「そのまま」使用して、一連の印刷フォーマッタを接続できます）。

6. （任意）ページレンダラを使用している場合は、このクラスの具象サブクラスを使用して1つ以上のUIPrintFormatterオブジェクトを作成できます。そして、UIPrintPageRendererのaddPrintFormatter:startingAtPageAtIndex:メソッドを呼び出すか、1つ以上の印刷フォーマッタの配列を作成して、その配列をprintFormattersプロパティに割り当てることによって、UIPrintPageRendererの特定のページ（またはページの範囲）用の印刷フォーマッタをUIPrintPageRendererに追加します。

7. 現在のユーザインターフェイスのイディオムがiPadの場合は、presentFromBarButtonItem:animated:completionHandler:またはpresentFromRect:inView:animated:completionHandler:を呼び出して、印刷インターフェイスをユーザに表示します。イディオムがiPhoneまたはiPod touchの場合は、presentAnimated:completionHandler:を呼び出します。

# 一般的な印刷タスク

以下で説明するすべてのコーディング作業は、印刷要求に応答してアプリケーションが実行する（または実行できる）ことです。ほとんどの作業は、どの順番で実行してもかまいませんが、最初にデバイスに印刷機能があることをチェックする必要があります。また、最後に印刷オプションを

表示しなければなりません。完全な例については、「プリンタ対応のコンテンツの印刷」（74 ページ）、「単一の印刷フォーマッタを使用した印刷」（75 ページ）、および「ページレンダラを使用したカスタムコンテンツの印刷」（79 ページ）を参照してください。

ここで取り上げていない重要な作業に、アプリケーションビューの適切な場所への印刷ボタンの追加、アクションメソッドの宣言、ターゲット／アクション接続の作成、アクションメソッドの実装があります（印刷ボタンを使用する際の推奨事項については「印刷ユーザインターフェイス」（59 ページ）を参照してください）。以降の作業は（「アプリケーションに適した用紙サイズの指定」（71 ページ）を除く）、アクションメソッドの実装の一部です。

## 印刷の可否のテスト

iOSデバイスの中には、印刷に対応していないものもあります。ビューをロードしたら、ただちにそのことを確認する必要があります。デバイスで印刷可能でない場合は、印刷用のユーザインターフェイス要素（ボタン、バーボタン項目など）を追加しないようにプログラミングするか、nibファイルからロードした印刷用の要素をすべて削除しなければなりません。印刷可能かを確認するには、`UIPrintInteractionController`の`isPrintingAvailable`クラスメソッドを呼び出します。リスト 6-1に、その実行方法を示します。ここでは、印刷ボタンのアウトレット（`myPrintButton`）は、nibファイルからロードされるものとしています。

**リスト 6-1**　印刷の可否に基づいて印刷ボタンを有効または無効にする

```
- (void)viewDidLoad {
    if (![UIPrintInteractionController isPrintingAvailable])
        [myPrintButton removeFromSuperView];
    // その他の処理...
}
```

**注：**「アクション」バーボタン項目などの印刷用の要素を無効にすることもできますが、削除することをお勧めします。`isPrintingAvailable`によって返される値は、印刷が現在可能かではなく、そのデバイスに印刷機能があるかを示しているため、指定されたデバイスに対して不変です。

## 印刷ジョブ情報の指定

`UIPrintInfo`クラスのインスタンスは、印刷ジョブについての情報をカプセル化します。具体的には、次の情報が含まれます。

- 出力タイプ（コンテンツの種類を表す）
- 印刷ジョブ名
- 印刷の向き
- 両面印刷モード
- 選択されているプリンタのID

`UIPrintInfo`のすべてのプロパティに値を割り当てる必要はありません。これらの値のいくつかはユーザが選択します。その他はUIKitがデフォルト値と見なします（それどころか、`UIPrintInfo`のインスタンスを明示的に作成する必要すらありません）。

しかし、ほとんどの場合は、印刷ジョブのいくつかの側面（出力タイプなど）は指定します。printInfoクラスメソッドを呼び出して、UIPrintInfoの共有インスタンスを取得します。そのオブジェクトの設定したいプロパティに値を割り当てます。次に、UIPrintInfoオブジェクトを、UIPrintInteractionControllerの共有インスタンスのprintInfoプロパティに割り当てます。リスト 6-2にこのプロシージャの例を示します。

**リスト 6-2**　UIPrintInfoオブジェクトのプロパティを設定して、それをprintInfoプロパティに割り当てる

```
UIPrintInteractionController *controller = [UIPrintInteractionController
sharedPrintController];
UIPrintInfo *printInfo = [UIPrintInfo printInfo];
printInfo.outputType = UIPrintInfoOutputGeneral;
printInfo.jobName = [self.path lastPathComponent];
printInfo.duplex = UIPrintInfoDuplexLongEdge;
controller.printInfo = printInfo;
```

UIPrintInfoプロパティの1つは印刷の向き（縦長または横長）です。印刷の向きをこれから印刷するオブジェクトの寸法に合わせたい場合もあります。つまり、オブジェクトが大きく、幅が高さより長い場合は、横長が最適な印刷の向きになります。リスト 6-3にその例を示します。

**リスト 6-3**　画像の寸法に合うように印刷の向きを設定する

```
  UIPrintInteractionController *controller = [UIPrintInteractionController
sharedPrintController];
// その他のコードをここに入れる...
  UIPrintInfo *printInfo = [UIPrintInfo printInfo];
  UIImage *image = ((UIImageView *)self.view).image;
  printInfo.outputType = UIPrintInfoOutputPhoto;
  printInfo.jobName = @"Image from PrintPhoto";
  printInfo.duplex = UIPrintInfoDuplexNone;
  // 描画する場合のみ...
  if (!controller.printingItem && image.size.width > image.size.height)
    printInfo.orientation = UIPrintInfoOrientationLandscape;
```

## アプリケーションに適した用紙サイズの指定

デフォルトでは、UIKitフレームワークは、印刷ジョブの出力タイプによって決まるページサイズにコンテンツを印刷します。出力タイプ（UIPrintInfoオブジェクトのoutputTypeプロパティに代入されてられている）がUIPrintInfoOutputPhotoの場合は、デフォルトの用紙サイズは、ロケールに応じて、4×6インチ、A6、またはその他の定型サイズになります。出力タイプがUIPrintInfoOutputGeneralまたはUIPrintInfoOutputGrayscaleの場合は、デフォルトの用紙サイズは、ロケールに応じて、USレター（8.5×11インチ）、A4、またはその他の定型サイズになります。

ほとんどのアプリケーションでは、これらのデフォルトの用紙サイズで十分です。ただし、アプリケーションによっては特殊な用紙サイズが必要になる場合もあります。たとえば、アプリケーションが、それぞれ特殊なコンテンツサイズの文書タイプ（パンフレット、グリーティングカードなど）を持つ場合です。このような場合は、Print Interaction Controllerのデリゲートが UIPrintInteractionControllerDelegateのprintInteractionController:choosePaper:プロトコルメソッドを実装して、UIPrintPaperオブジェクトを返すことができます。このオブジェクトは、利用可能な用紙サイズと特定のコンテンツサイズ用の印刷可能な矩形との最適な組み合わせを表します。

このデリゲートは、2つのアプローチを取ることができます。1つは、渡されたUIPrintPaperオブジェクトの配列を調べて最適な用紙サイズを特定する方法です。もう1つは、UIPrintPaperクラスのbestPaperForPageSize:withPapersFromArray:メソッドを呼び出して、最適なオブジェクトをシステムに選択させる方法です。リスト 6-4に、複数の種類の文書とそれぞれ固有のページサイズをサポートするアプリケーション用のメソッドの実装を示します。

**リスト 6-4**　printInteractionController:choosePaper:メソッドの実装

```
- (UIPrintPaper *)printInteractionController:(UIPrintInteractionController *)pic
    choosePaper:(NSArray *)paperList {
    // カスタムメソッドとプロパティ...
    CGSize pageSize = [self pageSizeForDocumentType:self.document.type];
    return [UIPrintPaper bestPaperForPageSize:pageSize
        withPapersFromArray:paperList];
}
```

通常は、カスタムページレンダラを使用するアプリケーションは、印刷ジョブのためのページ数（numberOfPages）の計算に、用紙サイズを考慮に入れます。

## 印刷ジョブの完了とエラーへの応答

UIPrintInteractionControllerで宣言されている表示メソッドの最後のパラメータは、完了ハンドラです。これについては「印刷オプションの表示」（73 ページ）で説明しています。完了ハンドラは、UIPrintInteractionCompletionHandler型のブロックで、印刷ジョブが正常に完了したときや、エラーのために強制終了したときに呼び出されます。印刷ジョブのためにセットアップした状態をクリーンアップするブロックの実装を、デベロッパ側で提供できます。完了ハンドラはエラーメッセージを記録することもできます。

リスト6-5の例では、完了またはエラーに関する内部プロパティをクリアし、エラーがある場合は、それについての情報を記録します。

**リスト 6-5**　完了ハンドラブロックの実装

```
void (^completionHandler)(UIPrintInteractionController *, BOOL, NSError *) =
        ^(UIPrintInteractionController *pic, BOOL completed, NSError *error) {
    self.content = nil;
    if (!completed && error)
        NSLog(@"FAILED! due to error in domain %@ with error code %u",
            error.domain, error.code);
  };
```

UIKitは、印刷ジョブの終了時にUIPrintInteractionControllerインスタンスに割り当てられたすべてのオブジェクトを自動的に解放します（ただし、デリゲートは除く）。そのため、完了ハンドラ内でこの処理を実行する必要はありません。

印刷エラーは、UIPrintErrorDomainというドメインと、UIPrintError.hで宣言されているエラーコードを持つNSErrorオブジェクトで表現されます。ほとんどすべての場合、これらのエラーコードはプログラミングエラーを示しているため、通常はユーザに知らせる必要はありません。しかし、（ファイルスキームのNSURLオブジェクトで指定された）ファイルを印刷しようとした結果として発生するエラーもあります。

# 印刷オプションの表示

UIPrintInteractionControllerには、ユーザに印刷オプションを表示するために次の3つのメソッドが宣言されており、それぞれアニメーションが付属しています。

- presentFromBarButtonItem:animated:completionHandler:は、ナビゲーションバーまたはツールバーのボタン（通常は印刷ボタン）からアニメーションでPopover Viewを表示します。
- presentFromRect:inView:animated:completionHandler:は、アプリケーションのビューの任意の矩形からアニメーションでPopover Viewを表示します。
- presentAnimated:completionHandler:は、画面の下端からスライドアップするページをアニメーション化します。

これらのうちの最初の2つは、iPadデバイス上で呼び出されることを想定しています。3つ目のメソッドは、iPhoneおよびiPod touchデバイス上で呼び出されることを想定しています。UI_USER_INTERFACE_IDIOMを呼び出して、その結果をUIUserInterfaceIdiomPadまたはUIUserInterfaceIdiomPhoneと比較することによって、デバイスタイプ（またはユーザインターフェイスイディオム）ごとに条件付きでコーディングすることができます。リスト 6-6にその例を示します。

**リスト 6-6**　現在のデバイスタイプに応じて印刷オプションを表示する

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    [controller presentFromBarButtonItem:self.printButton animated:YES
        completionHandler:completionHandler];
} else {
    [controller presentAnimated:YES completionHandler:completionHandler];
}
```

アプリケーションがiPhone上でiPad固有のメソッドを呼び出した場合（またはアニメーションを要求した場合）、デフォルトの動作では、画面の下端からスライドアップするページに印刷オプションが表示されます。アプリケーションがiPad上でiPhone固有のメソッドを呼び出した場合、デフォルトの動作では、現在のウインドウフレームからPopover Viewがアニメーションで表示されます。

印刷オプションがすでに表示されているときにpresent...メソッドの1つを呼び出した場合、UIPrintInteractionControllerは印刷オプションのビューまたはページを非表示にします。このメソッドを再度呼び出すと、印刷オプションが表示されます。

print-infoの値としてプリンタIDまたは両面印刷モードを割り当てた場合は、これらが印刷オプションのデフォルトとして表示されます（両面印刷コントロールを表示するには、プリンタに両面印刷機能が必要です）。印刷するページの範囲をユーザが選択できるようにする場合は、UIPrintInteractionControllerオブジェクトのshowsPageRangeプロパティをYESに設定する必要があります（デフォルト値はNO）。ただし、印刷可能なコンテンツをprintingItemsプロパティによって提供する場合、または総ページ数が1の場合は、showsPageRangeがYESであっても、印刷オプションにページ範囲コントロールは表示されません。

# プリンタ対応のコンテンツの印刷

iOSの印刷システムは、アプリケーションの関与をほとんど必要とせずに、特定のオブジェクトを受け取り、そのコンテンツを直接印刷します。これらのオブジェクトはNSDataクラス、NSURLクラス、UIImageクラス、およびALAssetクラスのインスタンスです。これらは画像データやPDFデータを保持または参照していなければなりません。画像データは、これらすべてのオブジェクトタイプに関与します。一方、PDFデータはNSURLオブジェクトによって参照されるか、NSDataオブジェクトによってカプセル化されます。これらのプリンタ対応オブジェクトには次の追加要件があります。

- 画像は、Image I/Oフレームワークがサポートする形式であること。サポートされる画像形式の一覧については、「サポートされる画像形式」（53 ページ）を参照してください。
- NSURLオブジェクトは、スキームとしてfile:、assets-library:、または登録済みプロトコルを持つNSDataを返すことができる任意のスキーム（QuickLookのx-apple-ql-id:スキームなど）を使用すること。
- ALAssetオブジェクトは、ALAssetTypePhoto型であること。

プリンタ対応オブジェクトを、UIPrintInteractionControllerの共有インスタンスのprintingItemプロパティまたはprintingItemsプロパティに割り当てます。単一のプリンタ対応オブジェクトはprintingItemプロパティに、プリンタ対応オブジェクトの配列はprintingItemsプロパティに割り当てます。アプリケーションが印刷可能なコンテンツに対してprintingItemsを使用した場合は、たとえ、複数のページが存在してshowsPageRangeプロパティがYESに設定されていても、ユーザはプリンタオプションビューでページ範囲を指定することはできません。

オブジェクトをこれらのプロパティに割り当てる前に、UIPrintInteractionControllerのクラスメソッドの1つの使用して、オブジェクトを検証する必要があります。たとえば、画像のUTIがあり、その画像がプリンタ対応であることを確認したい場合は、まずprintableUTIsクラスメソッドによってそれをテストできます。このメソッドは、対象の印刷システムで有効な一連のUTIを返します。

```
if ([[UIPrintInteractionController printableUTIs] containsObject:mysteryImageUTI])
    printInteractionController.printingItem = mysteryImage;
```

同様に、NSURLオブジェクトやNSDataオブジェクトをprintingItemプロパティやprintingItemsプロパティに割り当てる前に、UIPrintInteractionControllerのcanPrintURL:クラスメソッドやcanPrintData:クラスメソッドをこれらのオブジェクトに適用できます。これらのメソッドは、印刷システムが直接これらのオブジェクトを印刷できるかどうかを判断します。これらのメソッドの使用を、特にPDFの場合は、強くお勧めします。

リスト6-7に、NSDataオブジェクトにカプセル化されたPDF文書を印刷するコードを示します。それをprintingItemに割り当てる前に、そのオブジェクトの妥当性をテストします。また、ユーザに表示される印刷オプションにページ範囲コントロールを含めるように、Print Interaction Controllerに指示します。

**リスト 6-7**　ページ範囲の選択が可能な単一のPDF文書

```
- (IBAction)printContent:(id)sender {
    UIPrintInteractionController *pic = [UIPrintInteractionController
sharedPrintController];
    if  (pic && [UIPrintInteractionController canPrintData: self.myPDFData] )
{
        pic.delegate = self;
```

```
        UIPrintInfo *printInfo = [UIPrintInfo printInfo];
        printInfo.outputType = UIPrintInfoOutputGeneral;
        printInfo.jobName = [self.path lastPathComponent];
        printInfo.duplex = UIPrintInfoDuplexLongEdge;
        pic.printInfo = printInfo;
        pic.showsPageRange = YES:
        pic.printingItem = self.myPDFData;

        void (^completionHandler)(UIPrintInteractionController *, BOOL, NSError
 *) =
            ^(UIPrintInteractionController *pic, BOOL completed, NSError *error)
 {
              self.content = nil;
              if (!completed && error)
                  NSLog(@"FAILED! due to error in domain %@ with error code %u",
                   error.domain, error.code);
        };
        if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        [pic presentFromBarButtonItem:self.printButton animated:YES
            completionHandler:completionHandler];
        } else {
        [pic presentAnimated:YES completionHandler:completionHandler];
    }
}
```

一度に複数のプリンタ対応オブジェクトを送信する場合の手順も同じですが、オブジェクトの配列を`printingItems`プロパティに割り当てる必要があります。

```
pic.printingItems = [NSArray arrayWithObjects:imageViewOne.image,
imageViewTwo.image, imageViewThree.image, nil];
```

# 単一の印刷フォーマッタを使用した印刷

印刷フォーマッタは、複数ページにわたる印刷可能なコンテンツをレイアウトするオブジェクトです。このオブジェクトは、コンテンツが表示される領域の余白を指定したり、コンテンツの開始ページを指定します。また、開始ページ、コンテンツ領域、およびレイアウトするコンテンツに基づいて、コンテンツの終了ページを計算します。

UIKitを利用すると、単一の印刷フォーマッタを印刷ジョブに割り当てることができます。これは、プレーンテキストやHTML文書を持つ場合に役立つ機能です。UIKitには、この種のテキストコンテンツ用の印刷フォーマッタ具象クラスがあるからです。また、このフレームワークは、特定のUIKitビューのコンテンツをプリンタに合った方法で印刷できるようにする印刷フォーマッタ具象サブクラスも実装しています。

以降の説明は単一のフォーマッタ（ページレンダラなし）の使い方に関係していますが、印刷フォーマッタについてのほとんどの情報は、ページレンダラと組み合わせて使われる印刷フォーマッタにも当てはまります。これについては、「1つ以上の印刷フォーマッタをページレンダラと一緒に使用する」（82 ページ）で説明します。

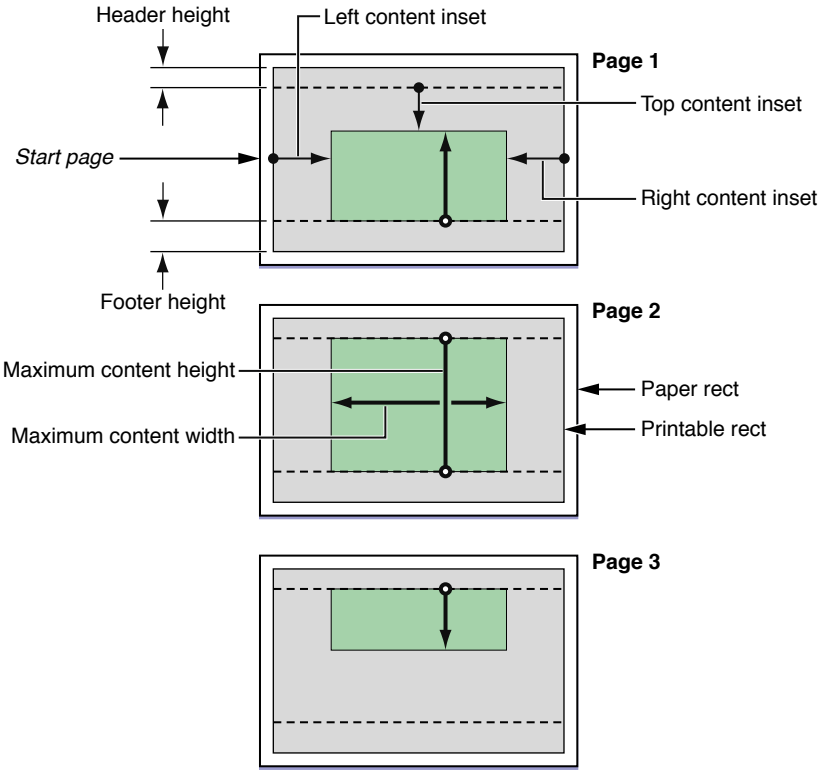
# 印刷ジョブのレイアウトプロパティの設定

印刷可能なコンテンツのページの領域を定義するために、UIPrintFormatterクラスは、具象サブクラス用に4つの重要なプロパティを宣言しています。これらのプロパティは、UIPrintPageRendererのfooterHeightプロパティやheaderHeightプロパティ、UIPrintPaperのpaperSizeプロパティやprintableRectプロパティと一緒に、複数ページの印刷ジョブのレイアウトを定義します。図 6-8 （77 ページ）は、このレイアウトを示しています。

| プロパティ | 説明 |
| --- | --- |
| contentInsets | 印刷可能領域の上、左、右の境界からの距離（ポイント単位）。これらの値は印刷されるコンテンツの余白に設定されます。ただし、これらの値よりもmaximumContentHeightやmaximumContentWidthの値が優先される場合もあります。上の余白は、印刷されるコンテンツの最初のページにのみ適用されます。 |
| maximumContentHeight | コンテンツ領域の最大の高さを表します。これは、ヘッダやフッタの高さに関係します。UIKitは、この値を、コンテンツ領域の高さから上の余白を引いた値と比較して、2つの値の小さいほうを使用します。 |
| maximumContentWidth | コンテンツ領域の最大幅を表します。UIKitは、この値を、コンテンツ領域の幅から左右の余白を引いた値と比較して、2つの値の小さいほうを使用します。 |
| startPage | 印刷用にコンテンツの描画を開始するページ。この値は0から始まります。印刷ジョブの最初のページの値は0です。 |

**注：** 印刷ジョブに単一のフォーマッタを使用している場合は(`printFormatter`)、`UIPrintPageRenderer`オブジェクトはありません。このため、印刷されるページにヘッダやフッタを付けることはできません。

**図 6-8** 複数ページの印刷ジョブのレイアウト




`UIPrintFormatter`は、図6-8に示したすべてのプロパティを使用して、印刷ジョブに必要なページ数を計算します。この値は、読み取り専用の`pageCount`プロパティに格納されます。

## テキスト文書やHTML文書の印刷

多くのアプリケーションには、ユーザが印刷したいと考えるテキストコンテンツが含まれています。コンテンツがプレーンテキストやHTMLテキストで、表示されているテキストコンテンツの元になっている文字列にアクセスできる場合は、`UISimpleTextPrintFormatter`または`UIMarkupTextPrintFormatter`のインスタンスを使用して、印刷用にテキストをレイアウトして描画できます。単にインスタンスを作成して、元になっている文字列で初期化し、レイアウトプロパティを指定します。次に、作成したこのインスタンスを、共有の`UIPrintInteractionController`インスタンスのインスタンス変数`printFormatter`に割り当てます。

リスト6-8に、`UIMarkupTextPrintFormatter`オブジェクトを使用してHTML文書を印刷する方法を示します。ここでは、四方に1インチの余白を持つようにページをフォーマットします。また、印刷されるコンテンツの幅が余白の内側で6インチ(`maximumContentWidth`)になるように指定しています。

**リスト 6-8**　HTML文書（ヘッダ情報なし）の印刷

```
- (IBAction)printContent:(id)sender {
    UIPrintInteractionController *pic = [UIPrintInteractionController
sharedPrintController];
    pic.delegate = self;

    UIPrintInfo *printInfo = [UIPrintInfo printInfo];
    printInfo.outputType = UIPrintInfoOutputGeneral;
    printInfo.jobName = self.documentName;
    pic.printInfo = printInfo;

    UIMarkupTextPrintFormatter *htmlFormatter = [[UIMarkupTextPrintFormatter alloc]
        initWithMarkupText:self.htmlString];
    htmlFormatter.startPage = 0;
    htmlFormatter.contentInsets = UIEdgeInsetsMake(72.0, 72.0, 72.0, 72.0); // 1インチの
余白
    htmlFormatter.maximumContentWidth = 6 * 72.0;
    pic.printFormatter = htmlFormatter;
    [htmlFormatter release];
    pic.showsPageRange = YES;

    void (^completionHandler)(UIPrintInteractionController *, BOOL, NSError *) =
        ^(UIPrintInteractionController *printController, BOOL completed, NSError *error)
 {
            if (!completed && error) {
                NSLog(@"Printing could not complete because of error: %@", error);
            }
         };
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        [pic presentFromBarButtonItem:sender animated:YES
completionHandler:completionHandler];
    } else {
        [pic presentAnimated:YES completionHandler:completionHandler];
    }
}
```

印刷ジョブに単一の印刷フォーマッタを使用する場合は（つまり、1つのUIPrintFormatterオブジェクトをUIPrintInteractionControllerインスタンスのprintFormatterプロパティに割り当てる）、印刷される各ページにヘッダとフッタのコンテンツを描画することはできません。ヘッダとフッタを描画するには、UIPrintPageRendererオブジェクトに加えて、必要な印刷フォーマッタを使用する必要があります。詳細については、「1つ以上のフォーマッタをページレンダラと一緒に使用する」（82 ページ）を参照してください。

UISimpleTextPrintFormatterオブジェクトを使用してプレーンテキスト文書をレイアウトして印刷する手順も、ほとんど同じです。ただし、このオブジェクトのクラスには、フォント、色、印刷するテキストの配置を設定するプロパティが含まれています。

## ビュー印刷フォーマッタの使用

UIViewPrintFormatterクラスのインスタンスを使用して、いくつかのシステムビューのコンテンツをレイアウトして印刷できます。UIKitフレームワークは、そのビューに対してこれらのビュー印刷フォーマッタを作成します。多くの場合、表示用のビューの描画に使われるコードと同じもの

が、印刷用のビューの描画にも使われます。現時点では、ビュー印刷フォーマッタを使用してコンテンツを印刷できるシステムビューは、UIWebView、UITextView、およびMKMapView（MapKitフレームワーク）のインスタンスです。

UIViewオブジェクトのビュー印刷フォーマッタを取得するには、ビューに対してviewPrintFormatterを呼び出します。開始ページと任意のレイアウトプロパティを設定して、このオブジェクトを共有インスタンスUIPrintInteractionControllerのprintFormatterプロパティに割り当てます。あるいは、印刷する出力の一部をUIPrintPageRendererオブジェクトを使用して描画している場合は、このオブジェクトにビュー印刷フォーマッタを追加することもできます。リスト 6-9に、UIWebViewオブジェクトのビュー印刷フォーマッタを使用して、そのビューのコンテンツを印刷するコードを示します。

**リスト 6-9**　Web Viewのコンテンツの印刷

```
- (void)printWebPage:(id)sender {
    UIPrintInteractionController *controller = [UIPrintInteractionController
sharedPrintController];
    if(!controller){
        NSLog(@"Couldn't get shared UIPrintInteractionController!");
        return;
    }
    void (^completionHandler)(UIPrintInteractionController *, BOOL, NSError *) =
       ^(UIPrintInteractionController *printController, BOOL completed, NSError *error)
 {
        if(!completed && error){
            NSLog(@"FAILED! due to error in domain %@ with error code %u",
            error.domain, error.code);
        }
    };
    UIPrintInfo *printInfo = [UIPrintInfo printInfo];
    printInfo.outputType = UIPrintInfoOutputGeneral;
    printInfo.jobName = [urlField text];
    printInfo.duplex = UIPrintInfoDuplexLongEdge;
    controller.printInfo = printInfo;
    controller.showsPageRange = YES;

    UIViewPrintFormatter *viewFormatter = [self.myWebView viewPrintFormatter];
    viewFormatter.startPage = 0;
    controller.printFormatter = viewFormatter;

    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        [controller presentFromBarButtonItem:printButton animated:YES
completionHandler:completionHandler];
    }else
        [controller presentAnimated:YES completionHandler:completionHandler];
}
```

# ページレンダラを使用したカスタムコンテンツの印刷

ページレンダラは、印刷ジョブの一部または全部を描画するUIPrintPageRendererのカスタムサブクラスのインスタンスです。これを使用するには、印刷ジョブのためにUIPrintInteractionControllerインスタンスを準備するときに、サブクラスを作成してプロジェクトに追加し、これをインスタンス化する必要があります。次に、このページレンダラを

UIPrintInteractionControllerのインスタンスのprintPageRendererプロパティに割り当てます。1つのページレンダラに、1つまたは複数の印刷フォーマッタを関連付けることができます。その場合、ページレンダラは、自身の描画と印刷フォーマッタの描画を混合します。

UIKit以外のAPI（Quartz APIなど）を使用して、画像やPDF文書を印刷可能なコンテンツとして描画している場合は、デフォルトの座標系（正の座標値が右上の方向になるように、yの原点を置く）を、UIKitのデフォルトの座標系に合うように「反転」させる必要が生じます。詳細については、「座標と座標変換」（14 ページ）を参照してください。

## ページレンダラ属性の設定

ページレンダラが、印刷する各ページのヘッダとフッタに描画する場合は、ヘッダやフッタの高さを指定する必要があります。それには、浮動小数点値（ポイント数を表す）を、ページレンダラのサブクラスが継承したheaderHeightプロパティとfooterHeightプロパティに割り当てます。これらのプロパティの高さの値が0の場合（デフォルト）、drawHeaderForPageAtIndex:inRect:メソッドとdrawFooterForPageAtIndex:inRect:メソッドは呼び出されません。

ページレンダラがページのコンテンツ領域（ヘッダとフッタの間の領域）に描画する場合は、ページレンダラのサブクラスでnumberOfPagesメソッドをオーバーライドして、ページレンダラが描画するページ数を計算して返すようにします。ページレンダラに関連付けられている印刷フォーマッタが、ヘッダとフッタの間にあるコンテンツをすべて描画する場合は、印刷フォーマッタがページ数を計算してくれます。この状況は、ページレンダラがヘッダやフッタ領域だけを描画し、その他のコンテンツはすべて印刷フォーマッタが描画する場合に生じます。

ページレンダラに印刷フォーマッタが関連付けられていない場合は、印刷可能なコンテンツの各ページのレイアウトはすべてデベロッパに任されます。 レイアウトの寸法を計算するときは、UIPrintPageRendererのheaderHeight、footerHeight、paperRect、およびprintableRectの各プロパティを考慮します（最後の2つのプロパティは読み取り専用です）。ページレンダラが印刷フォーマッタを使用している場合は、レイアウトの寸法にUIPrintFormatterのcontentInsets、maximumContentHeight、およびmaximumContentWidthの各プロパティが含まれます。詳細については、「印刷ジョブのレイアウトプロパティの設定」（76 ページ）の図と説明を参照してください。

## 描画メソッドの実装

アプリケーションがページレンダラを使用して印刷可能なコンテンツを描画する場合、UIKitは、要求されたコンテンツの各ページに対して次のメソッドを呼び出します。UIKitがページの順番でこれらのメソッドを呼び出す保証はありません。さらに、ユーザがページのサブセットの印刷を要求した（つまり、ページ範囲を指定した）場合は、UIkitはサブセットに含まれないページに対してこれらのメソッドを呼び出しません。

drawPageAtIndex:inRect:メソッドは、その他の描画メソッドを以下に示す順番で呼び出します。印刷用に描画される内容を完全に制御したい場合は、アプリケーションでこのメソッドをオーバーライドできます。

| オーバーライド | 目的 |
| --- | --- |
| drawHeaderForPageAtIndex:inRect: | ヘッダにコンテンツを描画します。headerHeightが0の場合、このメソッドは呼び出されません。 |

| オーバーライド | 目的 |
| --- | --- |
| `drawContentForPageAtIndex:inRect:` | 印刷ジョブのコンテンツ（ヘッダとフッタの間の領域）を描画します。 |
| `drawPrintFormatter:forPageAtIndex:` | カスタム描画と印刷フォーマッタによって実行された描画を混合します。このメソッドは、指定されたページに関連付けられている印刷フォーマッタごとに呼び出されます。詳細については、「1つ以上のフォーマッタをページレンダラと一緒に使用する」（82 ページ）を参照してください。 |
| `drawFooterForPageAtIndex:inRect:` | フッタにコンテンツを描画します。footerHeightが0の場合、このメソッドは呼び出されません。 |

これらのメソッドはすべて、（`UIGraphicsGetCurrentContext`から返された）現在の表示用グラフィックスコンテキストに描画するように設定されています。各メソッドに渡される矩形（ヘッダ領域、フッタ領域、コンテンツ領域、およびページ全体を定義する矩形）の値は、左上角にあるページの原点に対して相対的な値です。

リスト 6-10に、`drawHeaderForPageAtIndex:inRect:`メソッドと`drawFooterForPageAtIndex:inRect:`メソッドの実装例を示します。`CGRectGetMaxX`と`CGRectGetMaxY`を使用して、印刷可能な矩形の座標系での`footerRect`矩形と`headerRect`矩形内のテキストの配置を計算します。

**リスト 6-10**　ページのヘッダとフッタの描画

```
- (void)drawHeaderForPageAtIndex:(NSInteger)pageIndex  inRect:(CGRect)headerRect
 {
    UIFont *font = [UIFont fontWithName:@"Helvetica" size:12.0];
    CGSize titleSize = [self.jobTitle sizeWithFont:font];
    //center title in header
    CGFloat drawX = CGRectGetMaxX(headerRect)/2 - titleSize.width/2;
    CGFloat drawY = CGRectGetMaxY(headerRect) - titleSize.height;
    CGPoint drawPoint = CGPointMake(drawX, drawY);
    [self.jobTitle drawAtPoint:drawPoint withFont: font];
}

- (void)drawFooterForPageAtIndex:(NSInteger)pageIndex  inRect:(CGRect)footerRect
 {
    UIFont *font = [UIFont fontWithName:@"Helvetica" size:12.0];
    NSString *pageNumber = [NSString stringWithFormat:@"%d.", pageIndex+1];
    // フッタ矩形の右端にページ番号を置く
    CGSize pageNumSize = [pageNumber sizeWithFont:font];
    CGFloat drawX = CGRectGetMaxX(footerRect) - pageNumSize.width - 1.0;
    CGFloat drawY = CGRectGetMaxY(footerRect) - pageNumSize.height;
    CGPoint drawPoint = CGPointMake(drawX, drawY);
    [pageNumber drawAtPoint:drawPoint withFont: font];
}
```

## 1つ以上のフォーマッタをページレンダラと一緒に使用する

ページレンダラは、1つ以上の印刷フォーマッタと組み合わせて印刷可能なコンテンツを描画できます。たとえば、テキストコンテンツのページを印刷用に描画するために`UISimpleTextPrintFormatter`オブジェクトを使用し、各ページのヘッダに文書タイトルを描画するためにページレンダラを使用することができます。あるいは、2つの印刷フォーマッタを使用し、1つは最初のページの先頭にヘッダ（要約）情報を描画するために、もう1つは残りのコンテンツを描画するために使用することもできます。さらにこの2つの部分を区切る線を描画するためにページレンダラを使用することもできます。

先述のとおり、`UIPrintInteractionController`共有インスタンスの`printFormatter`プロパティに印刷フォーマッタを割り当てることによって、印刷ジョブに対して1つだけ印刷フォーマッタを使用できます。ただし、ページレンダラと複数の印刷フォーマッタを一緒に使用する場合は、それぞれの印刷フォーマッタをページレンダラに関連付けなければなりません。それには、次の2つのいずれかを実行します。

- 各印刷フォーマッタを、`printFormatters`プロパティに割り当てられている配列に入れる。
- `addPrintFormatter:startingAtPageAtIndex:`メソッドを呼び出して、各印刷フォーマッタを追加する。

印刷フォーマッタをページレンダラに関連付ける前に、印刷ジョブの開始ページ(`startPage`)などのレイアウトプロパティを必ず設定してください。これらのプロパティを設定すると、`UIPrintFormatter`は、印刷フォーマッタ用のページ数を計算します。`addPrintFormatter:startingAtPageAtIndex:`を呼び出して開始ページを指定した場合、その値は、`startPage`に割り当てられた任意の値を上書きします。印刷フォーマッタとレイアウト寸法については、「印刷ジョブのレイアウトプロパティの設定」（76 ページ）を参照してください。

ページレンダラは、`drawPrintFormatter:forPageAtIndex:`をオーバーライドして、自身による描画と、特定のページに指定された印刷フォーマッタによって実行される描画を統合することができます。まずページレンダラは、印刷フォーマッタが描画しないページ領域に描画します。次に、渡された印刷フォーマッタに対して`drawInRect:forPageAtIndex:`メソッドを呼び出して、ページの該当部分を描画させます。または、最初に印刷フォーマッタに描画させてから、印刷フォーマッタが描画したコンテンツの上に何かを描画することによって、「オーバーレイ」効果をもたらすこともできます。

# アプリケーションコンテンツの印刷テスト

iOS 4.2（およびそれ以降）のSDKでは、アプリケーションの印刷機能をテストできるPrinter Simulatorアプリケーションが提供されています。このアプリケーションは、一般的な種類の各種プリンタ（インクジェット、モノクロレーザ、カラーレーザなど）をシミュレートします。また、印刷されるページを「プレビュー(Preview)」アプリケーションで表示します。各ページの印刷可能領域を表示するように、環境設定を行うこともできます。また、Printer Simulatorは、印刷ジョブごとに印刷システムからの情報をログに記録します。Printer Simulatorは、ファイルシステムの次の場所にあります。

```
<Xcode>/Platforms/iPhoneOS.platform/Developer/Applications/PrinterSimulator.app
```

アプリケーションの印刷用コードをテストする場合は、`present...`メソッドに渡される完了ハンドラも実装して、この印刷システムから返されるエラーをすべて記録しなければなりません。通常、このエラーはプログラミングエラーであるため、アプリケーションのデプロイ前に捕捉する必要があります。詳細については、「印刷ジョブの完了とエラーへの応答」（72 ページ）を参照してください。

# 書類の改訂履歴

この表は「*iOS描画および印刷ガイド*」の改訂履歴です。

| 日付 | メモ |
| --- | --- |
| 2011-09-26 | 細かな変更をいくつか行いました。 |
| 2011-03-09 | 細かな変更をいくつか行いました。 |
| 2010-12-07 | 「ビットマップグラフィックスコンテキストへの描画」を画像関連の章に追加しました。 |
| 2010-11-15 | iOSの描画と印刷に関する概念、API、およびテクニックについて説明した文書の初版。 |

書類の改訂履歴