SDK互換性ガイド

Developer

目次

図、リスト

はじめに

Xcodeにはソフトウェア開発キット（SDK、Software Development Kit）が付属しており、iOSやMac OS Xの、ある特定のバージョン（開発環境と違っていても可）で動作するアプリケーションの開発に利用できます。この技術により、同一のバイナリ（実行可能ファイル）でありながら、ある新機能が動作環境のシステムに組み込まれていればそれを活用し、ない場合でもそれなりに対処するアプリケーションを開発できるのです。Appleフレームワークの中には、アプリケーションの開発に用いたSDKに基づき、自動的に振る舞いを変えて互換性を確保するものがあります。

> **注:** この文書では、iOSとMacOS Xの両方で動作するコードの開発方法については解説しません。Xcodeは、SDKを切り替えるだけでどちらのプラットフォームにも対応できるようになっていますが、iOSとMac OS Xのプログラムには、基本的な設計の違いがあります。詳細については、“Migrating from Cocoa”を参照してください。

この文書は、iOSやMac OS Xの、ある特定のバージョン（の範囲）で動作するアプリケーションを開発したい場合に参照するとよいでしょう。

この書類の構成

この文書は、次の章で構成されています。

- “SDKベースの開発の概要” （5 ページ）では、SDKを用いた開発作業について説明します。
- “プロジェクトの設定 - SDKベースの開発” （7 ページ）では、SDKを適切に利用するよう、プロジェクトを設定する手順を解説します。
- “SDKベースの開発” （12 ページ）では、弱くリンクされたクラス、メソッド、関数の使いかた、フレームワーク全体を弱くリンクする方法、SDK別に条件コンパイルする手順、廃止されたAPIを使っている箇所を見つける手順について説明します。

SDKベースの開発の概要

Appleでは、iOSやMac OS Xのバージョンごとに SDKを用意しています。これを利用すれば、開発環境とは異なるバージョンのオペレーティングシステム（OS）に対応したヘッダやライブラリを使って、アプリケーションを開発できます。たとえば、Mac OS Xバージョン10.4で動作するアプリケーションを、バージョン10.6の環境でも開発できる、ということです。

Xcode 3.2以降の開発環境にXcode Essentialsパッケージをインストールすると、これにMac OS X SDKも付属しています。Xcodeのリリースノートには、各リリースの対応SDKが列挙されています。iOS向けに開発する場合は、「iOS Dev Center」ウェブサイトからSDKをダウンロードしてください。

SDKベースの開発には次のような利点があります。

- ある特定のバージョンのOSに合わせて最適化したアプリケーションを構築し、同時に、将来のバージョンに対しても互換性を維持できます（ただし、新しい機能が自動的に使えるようになるわけではありません）。
- ある範囲のバージョンのOSに対応したアプリケーションを構築できます。旧バージョンのOSでも動作しますが、新バージョンであれば新たに組み込まれた機能も利用できます。したがって、新しいOSに移行したユーザは新機能を活用でき、まだ移行していないユーザも従来どおり使えることになります。

各バージョンのiOSやMac OS Xに対応したソフトウェアを開発、配布し、それぞれの機能を利用できるようにするためには、どのバージョンに対応したヘッダやライブラリ（すなわちSDK）を用いて構築するか、を指定する必要があります。ソフトウェアの動作に最低限必要な（最も古い）バージョンも指定できます。この考え方については、“ベースSDKと配布ターゲットの設定” （8 ページ）を参照してください。

フレームワークにおける振る舞いの選択

フレームワークは版を重ねるにつれて改善が進み、APIが新たに導入されたり廃止になったりしているほか、振る舞いが変わっていることもあります。Appleでは、互換性を損なうような変更は最小限にとどめるよう努めていますが、それでもフレームワークのバージョンによって振る舞いが変わることがあります。ごく稀ではありますが、フレームワークのバージョンに応じて、コードの記述を変えなければならないこともあります。

Appleのフレームワークでは時に、後方互換性を維持するため、アプリケーションの構築に使われたSDKのバージョンを調べ、旧SDKであれば振る舞いを変えることもあります。互換性に問題が生じることを、Appleがあらかじめ認識していた、あるいは後になって見つけた場合に、このような措置がとられます。

> **注:** バージョンによる動作の違いは、フレームワークのリリースノートに列挙されていますが、リファレンス資料にも載っているとは限りません。リリースごとの違いを把握しておきたい場合は、リリースノートを慎重に調べてください。

フレームワークは通常、アプリケーションがどのように構築されているか、を調べるため、これにリンクされているシステムフレームワークのバージョンを参照します。したがって、新しいSDKを使ってアプリケーションをリンクし直すと、振る舞いが変わり、互換性の問題が生じる恐れがあります。このような場合、アプリケーションを再構築したのが原因なので、同時に対処しなければなりません。したがって、些細なバグを修正するなど小規模な更新であれば、当初と同じ構築環境とライブラリを用意して、言い替えると、同じSDKを用いて構築するとよいでしょう。

場合によっては、アプリケーションの構築に用いたSDKとは無関係に、フレームワークの側で、新旧どちらの動作にするか設定できるようになっています。この機構は、デバッグ目的でのみ提供されていることが多いのですが、ある値を登録することにより、アプリケーション全体の振る舞いを変更できるようになっている場合もあります。`NSUserDefaults`のメソッド`registerDefaults:`を用いるだけで、簡単にこの機構をコードに組み込むことができます。

プロジェクトの設定 - SDKベースの開発

この章では、iOSおよびMac OS X用のSDKの設定方法と、特定のSDKを使うようXcodeプロジェクトを設定する手順を解説します。

SDKのヘッダファイルとスタブライブラリ

Xcodeをインストールすると、`/Developer/SDKs`というディレクトリが自動的に作成されます。この下はさらにいくつかのサブディレクトリに分かれており、それぞれ、ある特定のバージョンのiOSやMac OS Xに同梱されている、ヘッダファイルやスタブライブラリがすべて収録されています。Mac OS X SDKはメジャーバージョンに従って「`MacOSX10.6.sdk`」などという名前になっていますが、実際の中身は、その中で最新のマイナバージョンの内容になっています。

iOS用のXcodeインストーラはSDKを、プラットフォームに応じて、`/Developer/Platforms`ディレクトリ以下の、「`iPhoneOS.platform`」のような名前のディレクトリに置きます。さらにこの下には、当該プラットフォームに対応した`Developer/SDKs`ディレクトリがあります。iOS SDKは、マイナバージョンに従って、「`iPhoneOS4.2.sdk`」などという名前になっています。

プロジェクトで最新のSDKを選択すると、これに対応するOSで導入された、新しいAPIが使えるようになります。新しい機能を、システムアップデートにより追加したとしても、通常、それを反映した新しいヘッダファイルにはなりません。これに対し、SDKには新しいヘッダファイルが収録されています。

各`.sdk`ディレクトリの構成は、対応するOSのディレクトリ階層と同様になっています。最上位には、`usr`、`System`、`Developer`の各ディレクトリがあります。Mac OS Xの`.sdk`ディレクトリ以下には、`Library`ディレクトリもあります。以上の各ディレクトリは、さらにサブディレクトリに分かれており、Xcodeに付属する、対応するOS用のヘッダやライブラリが収録されています。

iOS/Mac OS X SDKのライブラリは、リンクに用いるだけのスタブです。実行形式コードはなく、エクスポートされたシンボルがあるだけです。したがって、SDKを使って開発したアプリケーションを動かすためには、実行環境として設定したバージョンのシステムが必要です。

ベースSDKと配布ターゲットの設定

Xcodeプロジェクトで用いるSDKを指定するため、構築に関する設定で、次の2項目を設定します。これにより、プロジェクトで使えるOS機能が決まります。

- **配布ターゲットの選択。** 開発したソフトウェアが動作する、最も古いOSの版を表します。デフォルト値は、ベースSDKに対応するOSのバージョンと同じになります。

  Xcodeのビルド変数`MACOSX_DEPLOYMENT_TARGET`（Mac OS Xに配布する場合）または`IPHONEOS_DEPLOYMENT_TARGET`（iOSに配布する場合）に、この設定値が格納されます。

  配布ターゲット以前のOSに組み込まれた機能は、無条件に使えます。

- **ベースSDKの選択。** 開発したソフトウェアでは、ベースSDKに対応するバージョン以前のOSに組み込まれている機能が使えます。デフォルト値は、Xcodeが対応する最新のOSになっています。

  Xcodeのビルド変数`SDKROOT`に、ベースSDKの設定値が格納されます。

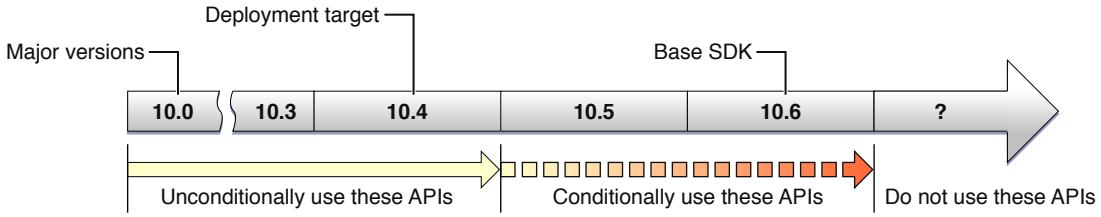
  配布ターゲットのバージョン以降、ベースSDKに対応するOSバージョンまでの機能は、開発したアプリケーションで利用できます。ただし、新機能が使えるかどうか、確認した上で実行する必要があります（"iOSにおける、弱くリンクされたクラスの使いかた"（12 ページ）、"弱くリンクされたメソッド、関数、シンボルの使いかた"（14 ページ）を参照）。

Xcodeのビルド設定に使える値その他については、『*Xcode Project Management Guide*』の「Building for Multiple Releases of an Operating System」 in *Xcode Project Management Guide*、『*Xcode Build Setting Reference*』、『*Tools Workflow Guide for iOS*』の"Building and Running Apps"を参照してください。

アプリケーションをビルドすると、配布ターゲットがアプリケーションの`Info.plist`ファイル内の`MinimumOSVersion`エントリに反映されます。iOSアプリケーションの場合、`MinimumOSVersion`エントリは、App StoreがiOSのリリース要件を示すために使います。

図 2-1に、時間軸に沿った形で、配布ターゲットとベースSDKの関係を示します。

図 2-1 SDKの開発タイムライン



この図に示すプロジェクトでは、配布ターゲットがMac OS X v10.4、ベースSDKがMac OS X v10.6となっています（システムアップデートを含め、関連するリリースをすべて示してあります）。

この場合、Mac OS X v10.0以降、10.4の最新アップデートまでの機能は、すべて自由に使えます。Mac OS X v10.5および10.6の機能は、それが組み込まれているかどうか確認した上で使えます。

このように設定すると、コンパイル時および実行時に、次のような効果があります。コード中に現れるシンボルが:

- ベースSDKで未定義（より新しいOSで導入された、など）の場合、コンパイル時にエラーになります。
- ベースSDKに定義があるけれども、廃止になった旨の印がついている場合、コンパイル時に警告が現れます。
- 配布ターゲットに定義がある場合、通常どおりリンク、ビルドできます。実行時には:
  - 配布ターゲットよりも古いOSが稼働するシステムでは、そのシンボルがOSに定義されていない場合、ロードに失敗する恐れがあります。
  - 配布ターゲット以降であれば、そのOSで定義されていないシンボルはロード時に解決されず、ヌルポインタになります。コードはこのような状況も考慮しながら記述してください（“弱くリンクされたメソッド、関数、シンボルの使いかた”（14 ページ）、“iOSにおける、弱くリンクされたクラスの使いかた”（12 ページ）を参照）。

注: Mac OS X v10.6では、iOS Simulator SDK（バージョン3.0より古いもの）は使えません。さらに、Simulator SDKを使ってビルドしたバイナリは、ベースSDKと同じバージョンのOSでしか動作しません（これより古いものも新しいものも不可）。

廃止になったAPIを使っていないか、必ず確認してください。当面は使えていても、将来にわたって使えるという保証はありません。コード中に廃止されたAPIがあれば、コンパイラが警告を出します（“廃止されたAPIを使っている箇所の検索”（17 ページ）を参照）。

Xcodeは、ベースSDKの設定を変えると、ビルドに用いるヘッダやスタブライブラリだけでなく、他の機能の振る舞いも適切に調整します。たとえば、シンボルの検索、コードの補完、画面に表示するヘッダファイルは、開発環境が稼働しているOSではなく、ベースSDKのヘッダに応じて決まります。Xcode Quick Helpにも同様の機構があって、資料を参照しようとすれば、ベースSDKに対応したものが表示されるようになっています。

ベースSDKや配布ターゲットは、プロジェクトでまとめて設定できるほか、ビルド対象ごとに個別に設定することも可能です。この個別の設定は、プロジェクトの設定よりも優先します（ただし、シンボル定義や資料の検索など、ベースSDKの設定に応じて決まるXcodeの機能がいくつかあり、その動作は異なることがあります）。

弱いリンク(Weak Linking)とAppleフレームワーク

Xcodeのコンパイラは、Appleフレームワークのヘッダに定義されているシンボルに関して、有効性マクロと呼ばれるものを使って、シンボルを弱くリンクするか、強くリンクするかを判断します。このマクロは、ある機能がOSのどのバージョンから使えるようになったか、を表します。たとえば次の宣言に現れるマクロは、`enumerateObjectsUsingBlock:`メソッド（`NSArray`クラスに属するもの）が、Mac OS X v10.6以降、iOS 4.0以降で使えるようになったことを表します。

```
- (void)enumerateObjectsUsingBlock:(void (^)(id obj, NSUInteger idx, BOOL
*stop))block NS_AVAILABLE(10_6, 4_0);
```

フレームワークのあるシンボルが、弱くリンクするよう定義されていれば、実行時にこのシンボルがなくても、プロセスはそのまま動作を続行できます。スタティックリンカは、あるシンボルを参照するコードモジュールにおいて、それが弱くリンクされたシンボルであれば、そのようなものとして扱います。ダイナミックリンカは、この同じ情報を実行時に参照して、そのまま処理を続行できるかどうか判断します。コードモジュールに弱くリンクされたシンボルがあるけれども、フレームワークには存在しなかった場合、コードモジュールがそのまま処理を続行できるのは、当該シンボルを参照しない間に限ります。一方、フレームワークにも存在すれば、通常どおり処理を続行できます。

廃止されたシンボルの場合、有効性マクロは特別な構文で、どのバージョンのOSで廃止になったか、を示します。いずれの場合も、そのシンボルのリファレンス資料には、使用可能かどうか、および、必要な場合は廃止に関する情報が記載されています。有効性マクロは、`/usr/include/`以下にある、`Availability.h`および`AvailabilityMacros.h`に定義されています。

Xcodeコンパイラは各有効性マクロを、ベースSDKおよび配布ターゲットに関するプロジェクトの設定と照らし合わせて解釈します。コンパイラはこの設定をもとに、`MAC_OS_X_VERSION_MIN_REQUIRED`マクロおよび`MAC_OS_X_VERSION_MAX_ALLOWED`マクロの値を適切に割り当てます。

たとえば、Xcodeで、配布ターゲット（最低限必要とするバージョン）を「`Mac OS X 10.5`」、ベースSDK（動作可能な最も新しいバージョン）を「`Mac OS X 10.6`」と設定したとします。するとコンパイラは、Mac OS X v10.6で導入されたインターフェイスは弱くリンクし、それより古いバージョンで定義されたインターフェイスは強くリンクします。その結果、アプリケーションはMac OS X v10.5で動作し、さらに、新しい機能が組み込まれてた動作環境ではそれも活用できることになります。

Important: 弱いリンクを利用するためには、Xcodeにおける配布ターゲットが、Mac OS X v10.2以降でなければなりません。これより古いMac OS Xでは弱いリンクができないのです。iOS向けにビルドする場合は、iOSのバージョンにかかわらず弱いリンクになります。

配布ターゲットよりも新しいiOSやMac OS Xで導入されたシンボルを使う場合は、あらかじめ、そのシンボルが使えるかどうか確認してください。使えない場合は、代替コードパスを与える必要があります。詳細については、“SDKベースの開発” （12 ページ）を参照してください。

makefileベースのプロジェクトの設定

makefileベースのプロジェクトであっても、コンパイルやリンクのコマンドに適当なオプションを追加することにより、SDKベースで開発する場合と同じことができます。makefileベースのプロジェクトでSDKを利用するためには、GCC 4.0以降が必要です。SDKの選択は、コンパイラの`-isysroot`オプション、リンカの`-syslibroot`オプションで行います。いずれの場合も、該当するSDKのディレクトリを完全パスで指定しなければなりません。makefileで配布ターゲットを設定するには、`ENVP=MACOSX_DEPLOYMENT_TARGET=10.4`のような形式のmakefile変数を使います。makefileでこの変数を使うときは、コンパイルやリンクのコマンドよりも前に置いてください。

プリフィックスファイルの設定

Xcodeにはプリフィックスファイルを扱う機能があります。これは、ソースファイルに明示的に記述しなくても、自動的にインクルードされるヘッダファイルのことです。多くのXcodeプロジェクトテンプレートは、プリフィックスファイルを自動生成するようになっています。たとえば、ある種のアプリケーションによく使われる、umbrellaフレームワークがそうなっています。プリフィックスファイルは、効率を考慮して、事前にコンパイルしてキャッシュに保存しておけるようになっており、大量の同じコードを（ソースファイルごとに）毎回コンパイルする必要がありません。記述子を追加することにより、アプリケーションが依存する、特定のフレームワークをインポートできます。

SDKベースの開発では、プリフィックスファイルも、該当するSDKを考慮に入れたものでなければなりません。すなわち、プリフィックスファイルに、umbrellaヘッダファイルを絶対パスで、`/System/Library/Frameworks/Cocoa.framework/Versions/A/Headers/Cocoa.h`のように設定してはいけない、ということです。このように絶対パスで記述すると、SDKの選択とは関係ない、開発環境のシステムに用意されたヘッダを参照することになってしまいます。

umbrellaフレームワークのヘッダをインクルードするためには、`#import <Framework/Framework.h>`という適切な記述子を、プリフィックスファイルに追加する必要があります。こうしておけば、コンパイラは常に、適切なSDKのディレクトリにあるヘッダを参照できます。たとえば、プロジェクト名がTestSDKで、`TestSDK_Prefix.pch`というプリフィックスファイルがある場合、各ソースファイルには、次の行を追加してください。

```
#import <Cocoa/Cocoa.h>
```

Objective-Cで開発している場合、`#include`（Cプログラムではこれを使うしかない）ではなく`#import`記述子を使うよう推奨します。`#import`であれば、同じヘッダファイルを繰り返しインクルードしてしまうことはありません。

SDKベースの開発

この章では、Xcodeのプロジェクトで利用できる、SDKベースの次のような開発技法について解説します。

- 弱くリンクされたクラス、メソッド、関数を使って、各種のバージョンのOS上で動作できるようにすること
- フレームワーク全体を弱くリンクすること
- SDKに応じて条件コンパイルすること
- 廃止になったAPIが使われていないか調べること
- OSやフレームワークのバージョンを実行時に判断すること

「弱いリンク」が使われる背景については"弱いリンク(Weak Linking)とAppleフレームワーク"（10ページ）を参照してください。

iOSにおける、弱くリンクされたクラスの使いかた

Xcodeプロジェクトで、弱くリンクされたクラスを使っている場合は、実行時に、その環境で該当するクラスが使えるかどうか、あらかじめ確認する必要があります。実行環境にないクラスを使おうとすると、ダイナミックリンカで実行時バインディングエラーが発生し、対応するプロセスが停止することがあります。

Xcodeプロジェクトで、ベースSDKがiOS 4.2以降であれば、`NSObjectclass`のメソッドを使って、弱くリンクされたクラスが実行環境で使えるかどうか確認してください。それには`NS_CLASS_AVAILABLE`というクラスの有効性マクロを使うと、簡単で効果的です。これはiOSの多くのフレームワークに用意されています。

> **Important:** `NS_CLASS_AVAILABLE`マクロに未対応のフレームワークもあるので、最新のiOSリリースノートで確認してください。

`NS_CLASS_AVAILABLE`マクロに対応したiOSフレームワークの場合、次の例のように、弱くリンクされたクラスの有無に応じて分岐してください。

```
if ([UIPrintInteractionController class]) {
```

```
    // インスタンスを生成して使う
} else {
    // 代替のコードを記述する
    //  （クラスが使えないので）
}
```

これがうまくいくのは、弱くリンクされたクラスが実行環境にない場合、これにメッセージを送ると、nilに送ったときと同じように振る舞うからです。弱くリンクされたクラスのサブクラスを定義した場合、そのスーパークラスが実行環境になければ、サブクラスもやはり、実行環境にないように見えます。

ここに示したようにclassメソッドを使うためには、フレームワークがNS_CLASS_AVAILABLEマクロに対応しているだけでは不充分です。これに加えて、下記の条件も満たす必要があります。そうすれば、先に示したコードは、クラスが存在しないバージョンのiOS上であっても、クラスの有無を安全にテストできるようになります。この設定は次のとおりです。

- Xcodeプロジェクトで用いるベースSDKは、iOS 4.2以降でなければなりません。ビルド設定エディタでは、ベースSDKはSDKROOTという変数に設定されています。
- プロジェクトの配布ターゲットはiOS 3.1以降でなければなりません。配布ターゲットはMACOSX_DEPLOYMENT_TARGETという変数に設定されています。
- プロジェクトで使うコンパイラは、LLVM-GCC 4.2以降、またはLLVMコンパイラ（Clang）1.5以降とします。C/C++コンパイラのバージョンは、GCC_VERSIONという変数に設定されています。
- プロジェクトの配布ターゲットで使えないフレームワークは、弱くリンクされていなければなりません。詳しくは『*Xcode Project Management Guide*』の“フレームワーク全体との弱いリンク”（16 ページ）および「Linking Libraries and Frameworks」 in *Xcode Project Management Guide* を参照してください。

Xcodeビルド設定エディタの使いかたについては、『*Xcode Project Management Guide*』の“Building Products”を参照してください。

Mac OS X（および上記の条件を満たさないiOSプロジェクト）では、classメソッドを使って、弱くリンクされたクラスの有無を判断することはできません。代わりにNSClassFromString関数を使って、次の例のように記述してください。

```
Class cls = NSClassFromString (@"NSRegularExpression");
if (cls) {
    // インスタンスを生成して使う
} else {
```

```
    // 代替のコードを記述する
    //  (クラスが使えないので)
}
```

弱くリンクされたメソッド、関数、シンボルの使いかた

プロジェクトで、弱くリンクされたメソッド、関数、外部シンボルを使っている場合は、実行時に、その環境で使えるかどうか、あらかじめ確認する必要があります。実行環境にないメソッドや関数、外部シンボルを使おうとすると、ダイナミックリンカで実行時バインディングエラーが発生し、対応するプロセスが停止することがあります。

たとえば、Xcodeプロジェクトで、ベースSDKをiOS 4.0と設定しているとします。すると、このバージョンのOSで実行すれば、当該バージョンに組み込まれた機能を使って動作しようとします。次に、このソフトウェアをiOS 3.1でも動かしたいとしましょう。もちろん、これより新しい版で導入された機能が使えないことは承知の上です。これは、配布ターゲットをバージョン3.1以前に設定することにより行います。

Objective-Cの場合、`instancesRespondToSelector:`メソッドで、メソッドセレクタの有無を調べることができます。たとえば、iOS 4.0で導入された`availableCaptureModesForCameraDevice:`メソッドを使う場合、次のようなコードになります。

リスト 3-1　Objective-Cのメソッドが呼び出し可能かどうかの確認

```
if ([UIImagePickerController instancesRespondToSelector:
                @selector (availableCaptureModesForCameraDevice:)]) {
    // メソッドは使用可。
    // 動画のキャプチャが可能かどうかを調べ、
    // 可能ならばその機能を提供する。
} else {
    // メソッドは使用不可。
    // 静止画のキャプチャのみ行う代替コード。
}
```

動作環境がiOS 4.0以降であれば、`availableCaptureModesForCameraDevice:`を呼び出して、動画のキャプチャが可能かどうか判断できます。しかしiOS 3.1の場合、静止画のキャプチャしかできないと想定しなければなりません。

このコードをビルドすると、設定に応じて、次のような結果になるでしょう。

- ベースSDKとしてiphoneos3.1を指定した場合:

  ビルドに失敗します。availableCaptureModesForCameraDevice:メソッドが、このバージョンには定義されていないからです。

- ベースSDKとしてiphoneos4.0を指定した場合:
  - 配布ターゲットとしてiphoneos4.0を指定した場合: iOS 4.0以降でのみ動作し、それ以外では起動に失敗します。
  - 配布ターゲットとしてiphoneos3.1を指定した場合: iOS 4.0でもiOS 3.1でも動作しますが、それより古いシステムでは起動に失敗します。iOS 3.1で動かした場合、代替コードで画像をキャプチャするようになります。

Objective-Cで、あるプロパティの有無を調べるには、その値を取得するメソッドの名前（プロパティ名と同じ）をinstancesRespondToSelector:に渡してください。

弱くリンクされたCの関数の有無は、ある関数が使えない場合、リンカはそのアドレスとしてNULLを設定する、という事実を利用して調べます。すなわち、関数のアドレスを、NULLまたはnilと比較して、当該関数の有無を判断するのです。たとえば、配布ターゲットがMac OS X v10.5より古いプロジェクトで、CGColorCreateGenericCMYK関数を使う場合、次のようなコードになります。

リスト 3-2　Cの関数の有無を確認

```
if (CGColorCreateGenericCMYK != NULL) {
    CGColorCreateGenericCMYK (0.1,0.5.0.0,1.0,0.1);
} else {
    // 関数は使用不可。
    // 従来の方法でカラーオブジェクトを生成する代替コード
}
```

> **注:** 関数の有無を調べる際には、そのアドレスを明示的に、NULLまたはnilと比較してください。否定演算子（!）をアドレスに前置して調べることはできません。また、Cの場合、関数名をそのまま書いても、アドレス演算子を前置しても、同義として扱われることに注意してください。たとえば「&myFunction」という記述は、「myFunction」と同義です。

外部（extern）定数や通知名の有無は、そのアドレス（シンボル名そのものではない）を明示的に、NULLまたはnilと比較することにより調べます。

フレームワーク全体との弱いリンク

最近（配布ターゲットよりも後）追加されたフレームワークを使う場合、フレームワーク自身を、明示的に弱くリンクする必要があります。たとえば、（iOS 4.0以降使えるようになった）Accelerateフレームワークにリンクして、新しいシステム機能を使うとしましょう。さらに、配布ターゲットをiOS 3.1.3として、このバージョンのiOSを使っているユーザも、新機能は別として、アプリケーション自体は使えるようにしたい、とします。このような場合、Accelerateフレームワーク全体を弱くリンクする必要があります。

一方、配布ターゲットでも使用*可能*なフレームワークを使う場合は、実行環境にそのフレームワークが必要です（弱いリンクでは不可）。

フレームワークに弱くリンクする手順については、『*Xcode Project Management Guide*』の「Linking Libraries and Frameworks」 in *Xcode Project Management Guide* を参照してください。

SDKに応じた条件コンパイル

ひと組のソースコードを共通に使って、複数のベースSDKを対象としてビルドする場合、ベースSDKに応じて条件コンパイルする必要があります。`Availability.h`に定義されているマクロを使い、プリプロセッサ記述子で振り分けます。

> **注:** ヘッダファイル`Availability.h`があるのは、iOSおよびMac OS X v10.6以降です。`AvailabilityMacros.h`はこれより古くからあるもので、Mac OS X v10.2で導入されました。ファイルはいずれも`/usr/include`ディレクトリ以下にあります。

リスト 3-2のコードを、ベースSDKを`macosx10.4`としてコンパイルすることを考えましょう。コード中、（Mac OS X v10.5で導入された）`CGColorCreateGenericCMYK`関数を呼び出している箇所は、ないものとしてコンパイルしなければなりません。`CGColorCreateGenericCMYK`関数は参照できないため、コンパイル時にエラーになってしまうからです。

このコードをビルドするためには、`__MAC_OS_X_VERSION_MAX_ALLOWED`マクロを使って、次のように制御する必要があります。

- プロジェクトの対象がMac OS Xである（iOSではない）ことを確認する
- ベースSDKでは使えないコード部分を隠す

実際のコード例を以下に示します。`#if`の中で、シンボル`__MAC_10_5`ではなく、`1050`という数値と比較していることに注意してください。このシンボルが定義されていない、古いシステム上であっても、問題なくコンパイルできるようにするためです。

リスト 3-3　プリプロセッサ記述子を使った条件コンパイル

```
#ifdef __MAC_OS_X_VERSION_MAX_ALLOWED
    // 対象がMac OS Xである（iOSではない）場合に限りコンパイルするコード
    // 「__MAC_10_5」ではなく「1050」と記述していることに注意
#if __MAC_OS_X_VERSION_MAX_ALLOWED >= 1050
    if (CGColorCreateGenericCMYK != NULL) {
        CGColorCreateGenericCMYK(0.1,0.5.0.0,1.0,0.1);
    } else {
#endif
    // 従来の方法でカラーオブジェクトを生成するコード
#if __MAC_OS_X_VERSION_MAX_ALLOWED >= 1050
    }
#endif
#endif
}
```

上記のコードは、プリプロセッサマクロを使っているだけでなく、より新しいベースSDKを指定してコンパイルし、一方で配布ターゲットはMac OS X v10.4以前であることも想定しています。実際、弱くリンクされたCGColorCreateGenericCMYKシンボルの有無を、呼び出す前に確認しています。これにより、新しいSDK用にビルドし、旧システムに配布した場合に起こりうる、実行時エラーを回避しています。シンボルの有無を実行時に検査する手順について詳しくは、“iOSにおける、弱くリンクされたクラスの使いかた”（12 ページ）および“弱くリンクされたメソッド、関数、シンボルの使いかた”（14 ページ）を参照してください。

廃止されたAPIを使っている箇所の検索

iOSやMac OS Xの進展に伴い、APIやこれを取り巻く技術も、開発者の要求に応じて変わることがあります。その過程で、効率に劣るインターフェイスは、新しいものに置き換えられます。ヘッダファイルの宣言に添えられている有効性マクロを使えば、廃止されたインターフェイスを使っている箇所を容易に見つけることができます。リファレンス資料でも、廃止されたインターフェイスにはその旨の印がついています。

> **注:** 「廃止された」といっても、直ちにフレームワークやライブラリから削除されてしまうわけではありません。より適切な代替手段がある旨を通知するだけです。廃止されたAPIも、依然として、コード中に記述することは可能です。しかしAppleでは、できるだけ早期に新しいインターフェイスに移行するようお勧めします。将来の版のOSでは、実際に削除される可能性があるからです。廃止されたAPIの代替インターフェイスについては、ヘッダファイルまたは該当する資料を参照してください。

配布ターゲットをMac OS X v10.5としてプロジェクトをコンパイルしたとき、廃止になったインターフェイスが使われていれば、コンパイラが警告を発します。警告には、廃止されたインターフェイス名と、それが使われているコード中の場所が記載されています。たとえば、廃止になった`HPurge`という関数が見つかった場合、エラーメッセージは次のようになります。

```
'HPurge' is deprecated (declared at /Users/steve/MyProject/main.c:51)
```

コード中、廃止になったAPIが使われている箇所を知りたい場合は、このような警告を調べればよいことになります。警告が多数ある場合は、Xcodeの検索フィールドを利用し、「deprecated」という字句が現れるものだけ抽出するとよいでしょう。

OSやフレームワークのバージョンの判定

ごく稀に、実行時にシンボルの有無を検査するだけでは、問題を解消できない場合があります。たとえば、あるメソッドの振る舞いがOSのバージョンによって異なる、あるいは以前からあったメソッドのバグが修復された、などの場合、この変化を考慮したコードを記述することが重要です。

OSのバージョンを確認する方法は、対象プラットフォームによって異なります。

- iOSのバージョンを実行時に検査するコードは次のようになります。

```
NSString *osVersion = [[UIDevice currentDevice] systemVersion];
```

- Mac OS Xのバージョンを実行時に調べるためには、Gestalt関数と「システムバージョンセレクタ」の定数を使います。

あるいは、あるフレームワークのバージョンを実行時に調べる、という方法もあり、多くのフレームワークで有効です。フレームワークが提供する、バージョンを表す大域定数を使って実装します。たとえばApplication Kitでは、（`NSApplication.h`に）`NSAppKitVersionNumber`という定数が宣言されており、Application Kitフレームワークのバージョンを判定できます。

```
APPKIT_EXTERN double NSAppKitVersionNumber;
#define NSAppKitVersionNumber10_0 577
#define NSAppKitVersionNumber10_1 620
#define NSAppKitVersionNumber10_2 663
#define NSAppKitVersionNumber10_2_3 663.6
#define NSAppKitVersionNumber10_3 743
#define NSAppKitVersionNumber10_3_2 743.14
#define NSAppKitVersionNumber10_3_3 743.2
#define NSAppKitVersionNumber10_3_5 743.24
#define NSAppKitVersionNumber10_3_7 743.33
#define NSAppKitVersionNumber10_3_9 743.36
#define NSAppKitVersionNumber10_4 824
#define NSAppKitVersionNumber10_4_1 824.1
#define NSAppKitVersionNumber10_4_3 824.23
#define NSAppKitVersionNumber10_4_4 824.33
#define NSAppKitVersionNumber10_4_7 824.41
#define NSAppKitVersionNumber10_5 949
#define NSAppKitVersionNumber10_5_2 949.27
#define NSAppKitVersionNumber10_5_3 949.33
```

この定数の値を比較することにより、どのバージョンのApplication Kitと連携してコードが動作しているか、を判定できます。一般に、この大域定数の整数部分を取り出し、`NSApplication.h`に宣言されている定数と比較することになるでしょう。次に例を示します。

```
if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_0) {
  /* On a 10.0.x or earlier system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_1) {
  /* On a 10.1 - 10.1.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_2) {
  /* On a 10.2 - 10.2.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_3) {
  /* On 10.3 - 10.3.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_4) {
  /* On a 10.4 - 10.4.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_5) {
```

```
  /* On a 10.5 - 10.5.x system */
} else {
  /* 10.6 or later system */
}
```

同様に、Foundationの場合、（NSObjCRuntime.hに）NSFoundationVersionNumberという大域定数の宣言があり、バージョンに応じた値が設定されています。

他のオブジェクトやコンポーネントに対応する個々のヘッダにも、バグ修正や機能アップデートがあった場合、NSAppKitVersionNumberという形でバージョン番号が宣言されていることがあります。

書類の改訂履歴

この表は「*SDK互換性ガイド*」の改訂履歴です。

日付	メモ
2010-11-15	“iOSにおける、弱くリンクされたクラスの使いかた”（12 ページ）に、`NS_CLASS_AVAILABLE`マクロの使用方法についての説明を追加しました。 “プロジェクトの設定 - SDKベースの開発”（7 ページ）および“SDKベースの開発”（12 ページ）の記述を更新、改訂しました。 “弱くリンクされたメソッド、関数、シンボルの使いかた”（14 ページ）に、外部シンボルの有無の確認方法を追加しました。
2010-02-16	条件付きコンパイルマクロの使用方法についてのサンプルを修正しました。
2009-09-09	ビルド設定に関する情報をわかりやすくしました。
2009-05-12	『クロス開発プログラミングガイド』から文書名を変更しました。iOS SDKベースの開発についての情報を追加しました。テキストを大幅に改訂し、古くなった情報を削除しました。
2006-11-07	クロス開発と汎用バイナリについての詳細情報を追加しました。 「フレームワークのバージョンの判定」の章を追加しました。Mac OS X 10.3.0以前を対象としてビルドする場合、GCC 3.3が必要であることを明記しました。GCC 4.0でコンパイルする必要がある、Mac OS Xのバージョンに関する記述を修正しました。コマンドラインからの汎用バイナリの構築に関する詳細情報へのリンクを追加しました。サンプルの配布ターゲットを変更しました。

日付	メモ
2006-05-23	ビルド設定に関する指定を修正し、アーキテクチャ別のビルド設定機能に関する詳細情報を追加しました。 「makefileベースのプロジェクトの設定」で、LDFLAGSのビルド設定に関する指定を修正しました。アーキテクチャ別のビルド設定機能に関する詳細情報を追加しました。
2006-03-08	ユニバーサルI/O Kitドライバを開発する方法について説明したテクニカルノートTN2163へのリンクを追加しました。
2006-02-07	細かな訂正を行いました。 カーネル拡張の汎用バイナリ版の構築に関する情報を追加しました。 LDFLAGSの使いかたを明確にしました。
2005-11-09	クロス開発を使用した汎用バイナリの作成についてのセクションを追加しました。廃止されたAPIの特定についての情報を追加しました。誤りを訂正しました。 「クロス開発と汎用バイナリ」の章を追加しました。“廃止されたAPIを使っている箇所の検索”（17 ページ）を追加しました。クロス開発の設定に関する内容を、Xcodeプロジェクトか、makefileベースのプロジェクトかによって、セクションを分けて記述するように再編しました。“SDKに応じた条件コンパイル”（16 ページ）のサンプルコードを修正しました。
2005-08-11	シンボルの有無を確認するコードのバグを修正しました。配布ターゲットを設定する手順がXcode 2.1に対応するように更新しました。
2005-06-04	未定義の関数のチェックについての情報を更新しました。コマンドラインのプログラムからSDKをサポートする方法についての情報を追加しました。

日付	メモ
2003-09-16	Mac OS Xバージョン10.3で搭載されたクロス開発サポートの最終的な機能に対応するように、本書の全体を通して多数の変更を行いました。
2003-08-21	本書の初版。

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

Apple Japan
〒106-6140 東京都港区六本木 6
丁目10番1号 六本木ヒルズ
http://www.apple.com/jp