Cocoaメモリ管理プログラミングガイド

Cocoa > Objective-C Language

2009-08-18

Apple Inc.
© 2009 Apple Inc.
All rights reserved.

本書の一部あるいは全部を Apple Inc. から書面による事前の許諾を得ることなく複写複製（コピー）することを禁じます。また、製品に付属のソフトウェアは同梱のソフトウェア使用許諾契約書に記載の条件のもとでお使いください。書類を個人で使用する場合に限り 1 台のコンピュータに保管すること、またその書類にアップルの著作権表示が含まれる限り、個人的な利用を目的に書類を複製することを認めます。

Apple ロゴは、米国その他の国で登録された Apple Inc. の商標です。

キーボードから入力可能な Apple ロゴについても、これを Apple Inc. からの書面による事前の許諾なしに商業的な目的で使用すると、連邦および州の商標法および不正競争防止法違反となる場合があります。

本書に記載されているテクノロジーに関しては、明示または黙示を問わず、使用を許諾しません。本書に記載されているテクノロジーに関するすべての知的財産権は、Apple Inc. が保有しています。 本書は、Apple ブランドのコンピュータ用のアプリケーション開発に使用を限定します。

本書には正確な情報を記載するように努めました。ただし、誤植や制作上の誤記がないことを保証するものではありません。

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/

Apple, the Apple logo, Carbon, Cocoa, Mac, Mac OS, Objective-C, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

iPhone is a trademark of Apple Inc.

Apple Inc. は本書の内容を確認しておりますが、本書に関して、明示的であるか黙示的であるかを問わず、その品質、正確さ、市場性、または特定の目的に対する適合性に関して何らかの保証または表明を行うものではありません。その結果、本書は「現状有姿のまま」提供され、本書の品質または正確さに関連して発生するすべての損害は、購入者であるお客様が負うものとします。

いかなる場合も、Apple Inc. は、本書の内容に含まれる瑕疵または不正確さによって生じる直接的、間接的、特殊的、偶発的、または結果的損害に対する賠償請求には一切応じません。そのような損害の可能性があらかじめ指摘されている場合においても同様です。

上記の損害に対する保証および救済は、口頭や書面によるか、または明示的や黙示的であるかを問わず、唯一のものであり、その他一切の保証にかわるものです。 Apple Inc. の販売店、代理店、または従業員には、この保証に関する規定に何らかの変更、拡張、または追加を加える権限は与えられていません。

一部の国や地域では、黙示あるいは偶発的または結果的損害に対する賠償の免責または制限が認められていないため、上記の制限や免責がお客様に適用されない場合があります。 この保証はお客様に特定の法的権利を与え、地域によってはその他の権利がお客様に与えられる場合もあります。

目次

はじめに 7

対象読者 7
この書類の構成 7

メモリ管理規則 9

オブジェクトの所有権と破棄 11

オブジェクトの所有権ポリシー 11
簡易メソッドを使用したオブジェクトの取得 12
解放の遅延 13
参照で返されたオブジェクト 14
オブジェクトの所有権の取得 14
共有オブジェクトの有効性 15
保持(Retain)の循環 15
オブジェクトへの弱い参照 16
保持カウント 17
オブジェクトの割り当て解除 17
リソース管理 18

実践的なメモリ管理 19

基礎 19
簡単な例 20
アクセサメソッドの使用 21
resetメソッドの実装 22
よくある間違い 22
しばしば混乱を招くケース 23

自動解放プール(Autorelease Pools) 25

自動解放プールの概要 25
非AppKitプログラムの自動解放プール 26
自動解放プールとスレッド 27
自動解放プールのスコープとネストした自動解放プールの意味 27
Foundationにおける所有権ポリシーの保証 28
ガベージコレクション 29

アクセサメソッド 31

アクセサメソッドの宣言 31

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

アクセサメソッドの実装 31
方法1 32
方法2 32
方法3 32
値オブジェクトとコピー 33

オブジェクトコピーの実装 35

深いコピーと浅いコピー 35
独立したコピー 36
スーパークラスからのNSCopyingの継承 37
「alloc、init...」アプローチの使用 37
NSCopyObject()の使用 37
可変(Mutable)オブジェクトと不変(Immutable)オブジェクトのコピー 39

CocoaでのCore Foundationオブジェクトのメモリ管理 41

nibオブジェクトのメモリ管理 43

アウトレット 43
Mac OS Xデスクトップ 44
iPhone 45
最上位オブジェクト 45
メモリ警告 45

書類の改訂履歴 47

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

図、リスト

オブジェクトの所有権と破棄 11

図 1 保持の循環の例 16

自動解放プール(Autorelease Pools) 25

リスト 1 非AppKitプログラムのmain関数の例 26

オブジェクトコピーの実装 35

図 1 浅いコピーと深いコピーの両方を利用したインスタンス変数のコピー 36
図 2 コピー中の参照カウントの初期化 39

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

はじめに

どのようなプログラムを記述する場合も、リソースの効果的かつ効率的な管理を保証する必要があります。このようなリソースの 1 つにプログラムのメモリがあります。Objective-Cプログラムでは、自分が作成したオブジェクトは不要になった時点で確実に破棄しなければなりません。

複雑なシステムでは、オブジェクトが不要になったかどうかを正確に判断することが難しい場合もあります。Cocoaでは、その判断を容易にするための規則や原則をいくつか定義しています。

> **重要：** Mac OS v10.5以降では、ガベージコレクションを採用することによって自動メモリ管理を使用できます。ガベージコレクションについては『*Garbage Collection Programming Guide*』で説明しています。

対象読者

オブジェクトの所有権ポリシーと、参照カウントされる環境におけるオブジェクトの作成、コピー、保持、および破棄に関する手法について学ぶには、この文書を読む必要があります。

> **注：** Mac OS X v10.5以降をターゲットにした新規プロジェクトを開始する場合は、ここで説明する手法を使用する十分な理由がない限り、通常はガベージコレクションを使用します。

この文書では、オブジェクトの割り当てと初期化、および初期化メソッドの実装については詳しく説明しません。これらのタスクについては『*The Objective-C 2.0 Programming Language*』の「Allocating and Initializing Objects」で説明しています。

この書類の構成

この文書は次の項目で構成されています。

- 「メモリ管理規則」（9 ページ）では、オブジェクトの所有権と破棄に関する規則についてまとめます。
- 「オブジェクトの所有権と破棄」（11 ページ）では、基本となるオブジェクト所有権ポリシーについて説明します。
- 「実践的なメモリ管理」（19 ページ）では、メモリ管理の実践的な側面について説明します。
- 「自動解放プール(Autorelease Pools)」（25 ページ）では、Cocoaプログラムでの自動解放プール（割り当て解除を遅らせるメカニズム）の使用について説明します。
- 「アクセサメソッド」（31 ページ）では、アクセサメソッドの実装方法について説明します。

© 2009 Apple Inc. All Rights Reserved.

- 「オブジェクトコピーの実装」（35 ページ）では、オブジェクトのコピーに関連する問題について説明します。これには、深いコピーを実装するか浅いコピーを実装するかの判断、サブクラスでオブジェクトコピーを実装するためのアプローチなどが含まれます。
- 「CocoaでのCore Foundationオブジェクトのメモリ管理」（41 ページ）では、Cocoaコードでのcore Foundationオブジェクトのメモリ管理に関するガイドラインと手法を示します。
- 「nibオブジェクトのメモリ管理」（43 ページ）では、nibファイルに関連するメモリ管理の問題について説明します。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

メモリ管理規則

この章では、Objective-Cのメモリ管理規則についてまとめます。

以下は基本規則です。

- 「alloc」または「new」で始まる名前のメソッドや、「copy」を含む名前のメソッド（たとえば、alloc、newObject、mutableCopy）を使用してオブジェクトを作成した場合、またはオブジェクトにretainメッセージを送信した場合は、そのオブジェクトの所有権を取得できます。その場合は、releaseまたはautoreleaseを使用してオブジェクトの所有権を放棄する責任があります。それ以外の方法でオブジェクトを受け取った場合は、そのオブジェクトを解放して*はなりません*。

次の規則は基本規則から派生した規則や、特殊ケースに対処するための規則です。

- 基本規則の当然の結果として、受け取ったオブジェクトをインスタンス変数のプロパティとして格納する必要がある場合は、それを保持またはコピーしなければなりません（これは、「オブジェクトへの弱い参照」（16 ページ）で説明した弱い参照には当てはまりません。ただし、このようなケースは稀です）。
- 受け取ったオブジェクトは、それを受け取ったメソッド内では有効であることが通常は保証されています（ただし、マルチスレッドアプリケーション、およびいくつかの分散オブジェクトの状態では例外です。受け取ったほかのオブジェクトを変更する場合には注意しなければなりません）。そのメソッドは安全にオブジェクトを呼び出し側に返すこともできます。

  メッセージの正常な副次的作用としてオブジェクトが無効になるのを防ぐ必要がある場合は、releaseまたはautoreleaseと組み合わせてretainを使用します（「共有オブジェクトの有効性」（15 ページ）を参照）。
- autoreleaseは、単に「後でreleaseメッセージを送信する」ことを意味します（「後で」の定義については、「自動解放プール(Autorelease Pools)」（25 ページ）を参照してください）。

これらの規則の根拠については、「オブジェクトの所有権と破棄」（11 ページ）を参照してください。

重要： Core Foundationオブジェクトにも同様のメモリ管理規則があります（『*Memory Management Programming Guide for Core Foundation*』を参照）。しかし、CocoaとCore Foundationの命名規則は異なっています。特に、Core Foundationの規則の作成（『*Memory Management Programming Guide for Core Foundation*』の「The Create Rule」）は、Objective-Cオブジェクトを返すメソッドには適用されません。たとえば、次のコードでは、myInstanceの所有権を放棄する責任は*ありません*。

```
MyClass *myInstance = [MyClass createInstance];
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

オブジェクトの所有権と破棄

この章では、Objective-Cオブジェクトの所有権に関するポリシーと、オブジェクトを破棄する方法およびタイミングについて説明します。

オブジェクトの所有権ポリシーがCocoaでどのように実装されているかを十分に理解するには、「自動解放プール(Autorelease Pools)」（25 ページ）も合わせて読む必要があります。

オブジェクトの所有権ポリシー

Objective-Cプログラムでは、オブジェクトはほかのオブジェクトの作成と破棄を絶えず行っています。アプリケーションが必要以上のメモリを消費しないよう保証するには、不要になった時点でオブジェクトを破棄することが重要です。オブジェクトは大部分の時間を使用して、自身で使用するために何かを作成し、必要に応じてそれらを破棄します。しかし、オブジェクトがほかのオブジェクトにメッセージを送信して値を渡す場合、所有権の境界（そしてその結果として破棄の責任）があいまいになります。たとえば、多数のSprocketオブジェクトを含むThingamajigオブジェクトが1つあり、ほかのオブジェクトが次のメソッドを使用してこのSprocketオブジェクトにアクセスする場合を考えます。

```
- (NSArray *)sprockets
```

この宣言には、返された配列を誰が解放すべきかについて何も記述されていません。しかし、Thingamajigオブジェクトがインスタンス変数を返す場合は、そのThingamajigオブジェクトが配列に責任を持つのが妥当です。一方、新たにThingamajigオブジェクトを作成した場合は、作成者にその新しいオブジェクトを破棄する責任があります。しかし、これが混乱の原因になる場合があります。「破棄」は「削除」や「割り当て解除」を意味するものです。

あるオブジェクトが別のオブジェクトを作成して、それをまた別のオブジェクトに渡す可能性があります（実際にこのようなことはよく行われます）。誰かがこの新しいオブジェクトを使い終わるまでそれを削除しないことが重要です。したがって、どのオブジェクトも1つ以上の所有者を持つ可能性がある場合は、オブジェクトの所有権という観点からメモリ管理を考える方が適しています。オブジェクトが少なくとも1つの所有者を持つ間は、そのオブジェクトは存在し続けます。オブジェクトが所有者を持たない場合は、ランタイムシステムが自動的にそれを破棄（割り当て解除）します。

オブジェクトを所有する場合としない場合、および所有者としての責任を明確にするために、Cocoaでは次のポリシーを設定しています。

- 自分が作成したオブジェクトはすべて自分が所有する。

  「alloc」または「new」で始まる名前のメソッドや「copy」を含む名前のメソッド（たとえば、`alloc`、`newObject`、`mutableCopy`）を使用してオブジェクトを「作成」する。

  オブジェクトの所有権を取得する別の方法については、「オブジェクトの所有権の取得」（14 ページ）を参照してください。

- オブジェクトを所有している場合は、それを使い終わったときに所有権を放棄する責任がある。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

releaseメッセージまたはautoreleaseメッセージを送信することによって、オブジェクトの所有権を放棄します（autoreleaseの詳細については、「解放の遅延」（13 ページ）を参照してください）。Cocoa用語では、オブジェクトの所有権を放棄することを一般にオブジェクトの「解放」と言います。

- 当然、オブジェクトを所有していなければ、オブジェクトの所有権を放棄することはできない。

このポリシーは、GUIベースのCocoaアプリケーションとコマンドラインのFoundationツールの両方に適用されます。

次の例を考えます。

```
Thingamajig *myThingamajig = [[Thingamajig alloc] init];
// ...
NSArray *sprockets = [myThingamajig sprockets];
// ...
[myThingamajig release];
```

この例はこのポリシーに適切に従っています。allocメソッドを使用してThingamajigオブジェクトを作成したので、その後でこのオブジェクトにreleaseメッセージを送信しています。Thingamajigオブジェクトからsprockets配列を取得した場合（自分がこの配列を「作成」したのではない場合）、この配列にはreleaseメッセージを送信してはなりません。

簡易メソッドを使用したオブジェクトの取得

ほとんどのクラスには、+className...という形式のメソッドがあります。これを使用して、そのクラスの新しいインスタンスを取得できます。これらのメソッドは「簡易コンストラクタ」と呼ばれ、クラスの新しいインスタンスを作成して初期化し、利用できるようにそのインスタンスを返します。このようにして作成したオブジェクトを解放する責任は受け取った側にあるように思えますが、前述の通り設定された所有権ポリシーに従うとそうではありません。クラスが新しいオブジェクトを作成するので、この新しいオブジェクトを破棄するのは*このクラス*の責任です。たとえば、次のコード例は*間違っています*。

```
Thingamajig *thingamajig = [Thingamajig thingamajig];
[thingamajig release]; // 間違い
```

このコードを試すとreleaseメッセージが送信されてすぐにエラーになるわけではありませんが、後で例外が発生します（「後で」の定義については、「自動解放プール(Autorelease Pools)」（25 ページ）を参照してください）。

ここで、どうしたらThingamajigクラスが所有権ポリシーを遵守できるかという問題が生じます。新規オブジェクトを解放する責任はありますが、受け取り側が所有権を要求する前にそのオブジェクトを解放してはいけません。これを示すためにthingamajigメソッドに考えられる2つの実装を検討します。

1. この実装は*間違っています*。新しいThingamajigオブジェクトへのクラスの参照は、thingamajigメソッドに制限されています。メソッドが返された後、クラスはこの新しいオブジェクトへの参照を失ってしまうので、releaseメッセージを送信して所有権を放棄することができません。

   ```
   + (Thingamajig *)thingamajig {
       id newThingamajig = [[Thingamajig alloc] init];
       return newThingamajig;
   }
   ```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

前述の通り設定された命名規則に従うと、呼び出し側には、返されたオブジェクトを解放する責任があることが知らされていないため、最終的にメモリリークを引き起こす可能性があります。

2. この実装も*間違*っています。クラスはこの新規オブジェクトの所有権を適切に放棄していますが、releaseメッセージが送信された後はこの新規Thingamajigオブジェクトは所有者を持たないのでただちにシステムによって破棄されます。

```
+ (Thingamajig *)thingamajig {
    id newThingamajig = [[Thingamajig alloc] init];
    [newThingamajig release];
    return newThingamajig; // ここではnewThingamajigは無効
}
```

Thingamajigクラスには、受け取り側がそれを使用したら、後から所有権を放棄できるようにオブジェクトにマークを付ける手段が必要です。Cocoaはそのためのメカニズム（「自動解放」と呼ばれる）を提供しています。これについては「解放の遅延」 （13 ページ）で説明します。

解放の遅延

NSObjectで定義されているautoreleaseメソッドは、それを受け取ったオブジェクトに後から解放するための印を付けます。オブジェクトを自動解放する（つまり、オブジェクトにautoreleaseメッセージを送信する）ことによって、autoreleaseを送信した側のスコープを越えてそのオブジェクトを所有しないことを宣言します。このスコープは現在の自動解放プールで定義されています（「自動解放プール(Autorelease Pools)」 （25 ページ）を参照）。

先に述べたsprocketsメソッドは次のように実装できます。

```
- (NSArray *)sprockets {
    NSArray *array;

    array = [[NSArray alloc] initWithObjects:mainSprocket,
                                auxiliarySprocket, nil];
    return [array autorelease];
}
```

別のメソッドがSprocketsオブジェクトの配列を取得した場合、そのメソッドでは、この配列は不要になったら破棄されるが、スコープ内ならどこででも安心して使用できることを前提にできます（「共有オブジェクトの有効性」 （15 ページ）を参照）。アプリケーションオブジェクトはコード用の呼び出しスタックの底を定義しているため、この配列を呼び出し側に返すこともできます。このように、autoreleaseメソッドを利用すると、どのオブジェクトも破棄の心配をせずにほかのオブジェクトを使用できます。

割り当てを解除した後でそのオブジェクトを解放するとエラーになるのと同様に、割り当てを解除した後でそのオブジェクトを後から解放するautoreleaseメッセージを何度も送信するとエラーになります。オブジェクトの作成回数（1回）にretainメッセージの送信回数を加えた回数だけ、releaseまたはautoreleaseをオブジェクトに送信すべきです（retainメッセージについては後で説明します）。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

参照で返されたオブジェクト

Cocoaには、オブジェクトが参照によって返されるように定義されたメソッドがいくつかあります。たとえば、次のメソッドのように、エラーが発生した場合にエラーに関する情報を保持するNSErrorオブジェクトを使用する例があげられます。

- initWithContentsOfURL:ofType:error: (NSDocument)
- initWithContentsOfURL:options:error: (NSData)
- initWithContentsOfFile:encoding:error: (NSString)

これらのメソッドでは、すでに説明したのと同じ規則が適用されます。これらのメソッドのどれを呼び出しても、自分がNSErrorオブジェクトを作成するわけではないためオブジェクトを所有していません。したがってオブジェクトを解放する必要はありません。

```
NSString *fileName = ...;
NSError *error;
NSString *string = [[NSString alloc] initWithContentsOfFile:fileName
                        encoding:NSUTF8StringEncoding error:&error];
if (string == nil) {
    // エラー処理
}
// ...
[string release];
```

何らかの理由で、返されたオブジェクトの所有権が基本規則に従っていない場合はそのメソッドのドキュメントにそのことが明記されています（例：dataFromPropertyList:format:errorDescription:）。

オブジェクトの所有権の取得

受け取ったオブジェクトを破棄したくない場合もあります。たとえば、インスタンス変数にオブジェクトをキャッシュする必要がある場合です。このような場合は、オブジェクトが不要かどうかを知っているのは自分だけであるため使用中にオブジェクトが破棄されないことを保証する機能が必要となります。これは、retainメッセージを使って行います。retainメッセージは、releaseメッセージまたはautoreleaseメッセージの逆のものです。オブジェクトを保持することによって、（ほかの誰もが同じ規則に従っていれば）そのオブジェクトを使い終わるまでは割り当て解除されないことを保証できます。たとえば、オブジェクトがmainSprocketを設定できるようにする場合は、次のようにそのSprocketを保持します。

```
- (void)setMainSprocket:(Sprocket *)newSprocket {
    [mainSprocket autorelease];
    mainSprocket = [newSprocket retain]; /* 新規のSprocketを要求する*/
    return;
}
```

これで、呼び出し側が保持しようとしているSprocketを引数としてsetMainSprocket:を呼び出すことができます。これは、このオブジェクトがほかのオブジェクトとSprocketを共有することを意味します。ほかのオブジェクトがこのSprocketを変更すると、このオブジェクトのmainSprocketも変更されます。それでよい場合もありますが、Thingamajigが固有のSprocketを持つ必要がある場合はこのメソッドでプライベートなコピーを作成します。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

```
- (void)setMainSprocket:(Sprocket *)newSprocket {
    [mainSprocket autorelease];
    mainSprocket = [newSprocket copy]; /* プライベートなコピーを作成する */
    return;
}
```

これらのメソッドはどちらも元のmainSprocketを自動解放します。これは、newSprocketとmainSprocketが同じオブジェクトで、Thingamajigがそれを所有する唯一のオブジェクトである場合に発生する問題（つまり、このような状況で、sprocketが解放されるとすぐに割り当てが解除され、それが保持されるかコピーされるとすぐにエラーが発生する）を回避します。次のコードはこの問題を解決します。

```
- (void)setMainSprocket:(Sprocket *)newSprocket {
    if (mainSprocket != newSprocket) {
        [mainSprocket release];
        mainSprocket = [newSprocket retain];
    }
}
```

共有オブジェクトの有効性

Cocoaの所有権ポリシーでは、受け取ったオブジェクトは通常、呼び出し側のメソッドのスコープ内で有効であると規定しています。また、受け取ったオブジェクトを現在のスコープから、解放される心配なしに返すこともできます。オブジェクトのgetterメソッドがキャッシュしたインスタンス変数を返すか計算した値を返すかは、アプリケーションにとって重要ではありません。必要な期間、オブジェクトが有効になっていることが重要なのです。

この規則には例外があります。たとえば、基本的なコレクションクラスはその内部に置かれているオブジェクトの存続期間を延長しようとはしません。可変配列からオブジェクトを削除すると以前取得したオブジェクトのコピーは無効になります。次に例を示します。

```
value = [array objectAtIndex:n];
[array removeObjectAtIndex:n];
// value は無効になっている
```

getterメソッドの1つを呼び出した後でオブジェクトを割り当て解除すると、また別の問題が発生します。

```
sprocket = [thingamajig mainSprocket];
[thingamajig release];
// sprocketは無効になっている
```

このような状況を防ぐには、sprocketを受け取ったらすぐにそれを保持し、使い終わったら解放します。呼び出し側がこのようにしてオブジェクトを保持するタイミングは必ずしも明確ではないので、オブジェクト自身が、現在の呼び出し側のスコープで有効な結果を返そうと努力する必要があります。ほとんどの場合、アクセサメソッドの実装方法を理解してアクセサメソッドを適切に実装すれば、どのような混乱も解消できます（「アクセサメソッド」 （31 ページ）を参照）。

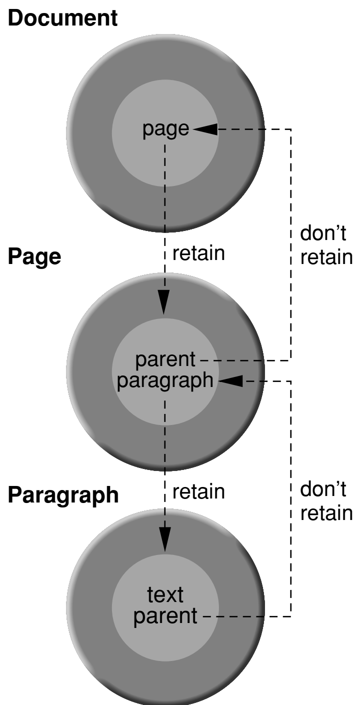
保持(Retain)の循環

場合によっては、2つのオブジェクトが循環参照することもあります。つまり、それぞれのオブジェクトが他方のオブジェクトを参照するインスタンス変数を含む場合です。たとえば、図1 （16 ページ）に示すようなオブジェクト関係を持つテキストプログラムを考えます。Documentオブジェクト

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

は、ドキュメント内のページごとにPageオブジェクトを1つ作成します。それぞれのPageオブジェクトは、それがどのドキュメントに属するかを追跡するインスタンス変数を1つ持っています。Documentオブジェクトがpageオブジェクトを保持し、Pageオブジェクトがdocumentオブジェクトを保持すると、どちらのオブジェクトも解放されることはありません。Documentの参照カウントはPageオブジェクトが解放されるまで0にならず、Pageオブジェクトはdocumentオブジェクトが割り当て解除されるまで解放されません。

図 1　　保持の循環の例



保持の循環に対する解決策は、「親」オブジェクトは「子」を保持していても、子オブジェクトは親を保持しないようにすることです。こうすれば、図 1（16 ページ）の場合、DocumentオブジェクトがPageオブジェクトを保持しても、PageオブジェクトはDocumentオブジェクトを保持しません。子から親への参照は弱い参照の一例です。これについては、「オブジェクトへの弱い参照」（16 ページ）で詳しく説明します。

オブジェクトへの弱い参照

オブジェクトを保持すると、そのオブジェクトへの「強い」参照が作成されます。オブジェクトへの強い参照がすべて解放されるまで、そのオブジェクトは割り当て解除できません。したがって、オブジェクトの存続期間はそのオブジェクトの強い参照の所有者によって決まります。場合によってはこのような動作が望ましくないこともあります。オブジェクトの割り当て解除を妨げずに、オブジェクトの参照を持ちたい場合もあります。このような場合は「弱い」参照を取得します。弱い参照は、オブジェクトを保持せずにオブジェクトへのポインタを格納することによって作成されます。

弱い参照は、それを利用しないと循環参照になってしまう場合に不可欠です。たとえばObject AとObject Bが相互に通信する場合、それぞれのオブジェクトは他方への参照を必要とします。それぞれが他方を保持すると、接続が切れるまでどちらのオブジェクトも割り当て解除できません。しか

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

し、いずれか一方のオブジェクトが割り当て解除されるまでは接続は切れません。これではどうしようもありません。循環を解消するには、一方のオブジェクトが従属する側に回り、他方への弱い参照を取得します。具体例としては、あるビュー階層内で、親ビューが子ビューを所有（保持）するが、子ビューは親ビューを所有しない場合があります。子の方では親を知る必要があるので、子は親への弱い参照を保持します。

Cocoaで弱い参照が使われている事例としては、テーブルデータソース、アウトラインビュー項目、通知オブザーバ、およびその他のターゲットやデリゲートがあります。ただし、それだけではありません。

重要： Cocoaでは、テーブルデータソース、アウトラインビュー項目、通知オブザーバ、およびデリゲートへの参照はすべて弱い参照です（たとえば、`NSTableView`オブジェクトはデータソースを保持していませんし、`NSApplication`オブジェクトはデリゲートを保持していません）。この文書では、この規約に対する例外についてのみ説明します。

弱い参照のみを保持するオブジェクトにメッセージを送信する場合は注意が必要です。割り当て解除されているオブジェクトにメッセージを送信すると、アプリケーションがクラッシュします。オブジェクトが有効であることを示す、明確に定義された状態を持つ必要があります。ほとんどの場合、弱い参照で参照されるオブジェクトは、循環参照の場合のように参照元のオブジェクトを知っているため、自分が割り当て解除されるときにほかのオブジェクトに通知する責任があります。たとえば、通知センターにオブジェクトを登録すると、通知センターはそのオブジェクトへの弱い参照を格納し、適切な通知が投稿されたときにオブジェクトにメッセージを送信します。オブジェクトが割り当て解除された場合は、通知センターからそのオブジェクトを登録解除して通知センターがもう存在しないオブジェクトにこれ以上メッセージを送信しないようにする必要があります。同様に、デリゲートオブジェクトが割り当て解除された場合は、他方のオブジェクトに`nil`引数を持つ`setDelegate:`メッセージを送信して、デリゲートのリンクを削除する必要があります。通常、これらのメッセージはオブジェクトの`dealloc`メソッドから送信されます。

保持カウント

通常は、オブジェクトに明示的に保持カウントを問い合わせる必要はありません（`retainCount`を参照）。関心のあるオブジェクトを保持しているフレームワークがわからないために、その結果を誤解することがよくあります。メモリ管理の問題をデバッグするときは、コードが所有権規則に従っているかを確認することだけに集中すべきです。

オブジェクトの割り当て解除

Cocoaは、「参照カウント」または「保持カウント」と呼ばれるメカニズムによって所有権ポリシーを実装しています。オブジェクトを作成すると、オブジェクトの保持カウントは1になります。オブジェクトに`retain`メッセージを送信すると、保持カウントが1つ増えます。オブジェクトに`release`メッセージを送信すると、保持カウントが1つ減ります（`autorelease`は、保持カウントのデクリメントを遅らせます）。

保持カウントが0になるとオブジェクトのメモリが回収されます。Cocoa用語では、これを「解放」または「割り当て解除」と呼びます。オブジェクトが割り当て解除されると自動的にそのオブジェクトの`dealloc`メソッドが呼び出されます。`dealloc`メソッドの役割は、オブジェクトが所有するメモリを解放してそのオブジェクトが保持しているすべてのリソース（オブジェクトインスタンス変数を含む）を破棄することです。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

クラスがオブジェクトインスタンス変数を持つ場合は、それらを解放し、その後親の実装を呼び出すdeallocメソッドを実装する必要があります。たとえば、Thingamajigクラスがnameとsprocketsというインスタンス変数を持つ場合は、次のようにdeallocメソッドを実装します。

```
- (void)dealloc {
    [sprockets release];
    [name release];
    [super dealloc];
}
```

ほかのオブジェクトのdeallocメソッドを直接呼び出すことは決してしないでください。

重要： アプリケーションが終了するときは、終了時にプロセスのメモリが自動的にクリアされるため、オブジェクトにdeallocメッセージが送信されない場合があります。ただし、メモリ管理メソッドをすべて呼び出すよりも、オペレーティングシステムにリソースをクリーンアップさせる方がはるかに効率的です。これはdeallocメソッドをどのように実装するかに影響します（「リソース管理」 （18 ページ）を参照）。

リソース管理

通常、ファイル記述子、ネットワーク接続、およびバッファ/キャッシュなどの希少なリソースは、deallocメソッドでは管理しません。特に、deallocが期待どおりのタイミングで呼び出されることを前提に、クラスを設計してはいけません。deallocの呼び出しは、バグやアプリケーションのクラッシュのために後回しにされたり実行されない場合もあります。

希少なリソースを扱うインスタンスを持つクラスの場合は、リソースが必要でなくなったら、その時点でインスタンスに「クリーンアップ」を伝えるようにアプリケーションを設計する必要があります。通常は、インスタンスを解放するとdeallocが呼び出されます。ただし、deallocが呼び出されなくてもその他の問題に悩まされることはありません。

deallocの先頭でリソース管理を実行しようとすると、いくつかの問題が発生します。以下に示します。

1. オブジェクトグラフを破棄するときの順序依存。

   オブジェクトグラフの破棄メカニズムには本来順番がありません。通常は特定の順序を期待（および取得）することができますが、脆弱性があります。オブジェクトが予期せずに自動解放プールに入ると破棄の順番が変わることがあります。そのために予期しない結果が生じる場合があります。

2. 希少なリソースを回収できない。

   メモリリークは修正しなければならないバグですが、一般に致命的なものではありません。しかし、希少なリソースを解放したいときにそれが解放されない場合は、かなり深刻な問題になります。たとえば、アプリケーションでファイル記述子が不足するとユーザはデータを保存できません。

3. 実行中のロジックを間違ったスレッドでクリーンアップする。

   オブジェクトが予期しないタイミングで自動解放プールに入ると、それが発生したスレッドのプールで割り当て解除が行われます。1つのスレッドからしかアクセスしてはならないリソースの場合は、すぐに致命的なエラーになります。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

実践的なメモリ管理

この章では、メモリ管理について実践的観点から説明します。

いくつかの単純な規則に従うことで、メモリ管理は容易になります。この規則を守らないと、ほぼ間違いなく、ある時点でメモリリークや解放済みのオブジェクトにメッセージを送信したことによるランタイム例外が発生します。

基礎

アプリケーションのメモリ消費をできるだけ低く抑えるには、使っていないオブジェクトを削除する必要があります。ただし、使用中のオブジェクトを削除しないように注意する必要があります。したがって、オブジェクトがまだ有用であるという印を付けるメカニズムが必要です。こうした理由から、多くの点でメモリ管理は「オブジェクトの所有権」の側面からが一番良く理解できます。

- 1つのオブジェクトは1つ以上の所有者を持つ。

  例えとして、共同所有のアパートを考えてみてください。

- オブジェクトの所有者がいなくなると、そのオブジェクトは破棄される。

  上記の例えをさらに広げて、地域住民に人気のないアパートを考えてみてください。所有者が誰もいなくなると、アパートは壊されます。

- 関心を持っているオブジェクトが破棄されないようにするには、自分が所有者になる必要がある。

  新しいアパートを建てるか、既存のアパートに出資することができます。

- 関心がなくなったオブジェクトが破棄されるようにするには、所有権を放棄する。

  アパートを売ることができます。

このモデルをサポートするために、Cocoaでは「参照カウント」または「保持カウント」と呼ばれるメカニズムを提供しています。どのオブジェクトにも保持カウントがあります。オブジェクトは作成されると保持カウント`1`になります。保持カウントが`0`になると、そのオブジェクトは割り当て解除（破棄）されます。次のようなさまざまなメソッドを使用して、保持カウントを操作します（所有権の取得や放棄）。

`alloc`

: オブジェクトにメモリを割り当てて、保持カウントを`1`にして返します。

  `alloc`または`new`で始まる任意のメソッドでオブジェクトを作成すると、そのオブジェクトを所有できます。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

`copy`
: オブジェクトをコピーして、保持カウントを1にして返します。

  オブジェクトをコピーすると、そのコピーを所有できます。これは、copyという単語を含む任意のメソッドに当てはまります。ここで「copy（コピー）」とは、返されるオブジェクトを指します。

`retain`
: オブジェクトの保持カウントを1つ増やします。

  オブジェクトの所有権を取得します。

`release`
: オブジェクトの保持カウントを1つ減らします。

  オブジェクトの所有権を放棄します。

`autorelease`
: 今後どこかの段階で、オブジェクトの参照カウントを1つ減らします。

  今後どこかの段階で、オブジェクトの所有権を放棄します。

メモリ管理の規則は、次のようにまとめることができます（「メモリ管理規則」（9 ページ）も参照）。

- 1つのコードブロック内では、copy、alloc、およびretainを使用した回数と、releaseおよびautoreleaseを使用した回数は等しくなければならない。
- 「alloc」または「new」で始まる名前のメソッドや、「copy」を含む名前のメソッド（たとえば、alloc、newObject、mutableCopy）を使用して作成した場合、または オブジェクトにretainメッセージを送信した場合にのみ、そのオブジェクトを所有できる。
- 自分が所有するインスタンス変数を解放するためにdeallocを実装する。
- （独自のdeallocメソッド内で親の実装を呼び出す場合以外は）deallocを直接呼び出してはならない。

ほとんどのクラスには、+className...という形式のメソッドがあります。これを使用して、そのクラスの新しいインスタンスを取得できます。これらのメソッドは「簡易コンストラクタ」と呼ばれ、クラスの新しいインスタンスを作成して初期化し、利用できるようにそのインスタンスを返します。簡易コンストラクタやその他のアクセサメソッドから返されたオブジェクトは所有できません。

簡単な例

次の簡単な例は、オブジェクトを作成するためにallocを使用する場合、簡易コンストラクタを使用する場合、アクセサメソッドを使用する場合の違いを示しています。

最初の例では、allocを使用して新しい文字列オブジェクトを作成します。したがって、このオブジェクトは解放しなければなりません。

```
- (void)printHello {
    NSString *string;
    string = [[NSString alloc] initWithString:@"Hello"];
    NSLog(string);
    [string release];
}
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

2番目の例では、簡易コンストラクタを使用して新しい文字列オブジェクトを作成します。この場合は追加の処理を行う必要はありません。

```
- (void)printHello {
    NSString *string;
    string = [NSString stringWithFormat:@"Hello"];
    NSLog(string);
}
```

3番目の例では、アクセサメソッドを使用して文字列オブジェクトを取得します。簡易コンストラクタの場合と同様に、追加の処理を行う必要はありません。

```
- (void)printWindowTitle {
    NSString *string;
    string = [myWindow title];
    NSLog(string);
}
```

アクセサメソッドの使用

飽き飽きすると思われるかもしれませんが、一貫してアクセサメソッドを使用すればメモリ管理の問題に直面する回数はかなり減ります。コード全体にわたって、インスタンス変数に`retain`と`release`を使用していると、ほぼ間違いなく誤りを犯します。

Counterオブジェクトにカウントをセットする場合を考えます。

```
@interface Counter :NSObject {
    NSNumber *count;
}
```

カウントを取得したり設定したりするために、2つのアクセサメソッドを定義します。getアクセサでは、変数を返すだけなので`retain`や`release`の必要はありません。

```
- (NSNumber *)count {
    return count;
}
```

setメソッドでは、ほかの誰もが同じ規則に従っている場合、新しいカウントは常に破棄される可能性があることを前提にしなければなりません。したがって、破棄されないようにするには`retain`メッセージを送信してこのオブジェクトの所有権を取得する必要があります。また、古いカウントオブジェクトに`release`メッセージを送信して、所有権を放棄しなければなりません（Objective-Cでは`nil`にメッセージを送信することが許されています。このため、カウントがまだ設定されていなくてもこのコードは動作します）。この2つが同じオブジェクトだった場合に備えて（誤ってそれが割り当て解除されないように）、`[newCount retain]`の後でreleaseメッセージを送信する必要があります。

```
- (void)setCount:(NSNumber *)newCount {
    [newCount retain];
    [count release];
    // 新たに代入する
    count = newCount;
}
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

これらの例では、アクセサメソッドの単純な側面を示しました。アクセサメソッドについては、「アクセサメソッド」（31 ページ）でさらに詳しく説明します。

Counterはオブジェクトインスタンス変数を持つため、deallocメソッドも実装する必要があります。インスタンス変数にreleaseメッセージを送信してそれらの所有権を放棄し、最後に親の実装を呼び出す必要があります。

```
- (void)dealloc {
    [count release];
    [super dealloc];
}
```

resetメソッドの実装

カウンタをリセットするメソッドを実装する場合を考えます。それには2つの選択肢があります。1つ目は、簡易コンストラクタを使用して新しいNSNumberオブジェクトを作成する方法です。この場合は、retainメッセージまたはreleaseメッセージは必要ありません。どちらの場合も、このクラスのsetアクセサメソッドを使用する点に注目してください。

```
- (void)reset {
    NSNumber *zero = [NSNumber numberWithInteger:0];
    [self setCount:zero];
}
```

2つ目は、allocを使用してNSNumberインスタンスを作成する方法です。この場合は、allocとreleaseを対応させます。

```
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [self setCount:zero];
    [zero release];
}
```

よくある間違い

以下のセクションではよくある間違いを示します。

アクセサが使われていない

次の例は、単純なケースではほぼ確実にうまくいきます。ただし、アクセサメソッドを避けようとしているためほとんどの場合どこかの段階で間違いが生じます。

```
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [count release];
    count = zero;
}
```

キー値監視（『*Key-Value Observing Programming Guide*』を参照）を使用している場合は、このような変数の変更はKVO準拠ではない点にも注意してください。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

インスタンスのリーク

```
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [self setCount:zero];
}
```

（allocからの）新しい数字の保持カウントは1ですが、このメソッドのスコープ内でreleaseと対応していません。この新しい番号が解放されないとすると、メモリリークが生じます。

所有していないインスタンスにreleaseが送信された

```
- (void)reset {
    NSNumber *zero = [NSNumber numberWithInteger:0];
    [self setCount:zero];
    [zero release];
}
```

ほかのretain呼び出しがなければ、現在の自動解放プールが解放された後の次のcountへのアクセスでこのコードは失敗します。簡易コンストラクタメソッドは自動解放オブジェクトを返します。したがって、releaseを送信する必要はありません。autoreleaseによるreleaseが送信されると、保持カウントが0になり、そのオブジェクトは解放されます。次にカウントにアクセスすると、解放済みのオブジェクトにメッセージを送信することになります（通常、これはSIGBUS 10エラーになります）。

しばしば混乱を招くケース

配列、辞書、集合などのコレクションにオブジェクトを追加または設定すると、コレクションがそのオブジェクトの所有権を取得します。オブジェクトがコレクションから削除されたりコレクション自体が解放されたりすると、コレクションは所有権を放棄します。したがって、たとえば、数値の配列を作成する場合、次のいずれかを実行できます。

```
NSMutableArray *array;
NSUInteger i;
// ...
for (i = 0; i < 10; i++) {
    NSNumber *convenienceNumber = [NSNumber numberWithInteger:i];
    [array addObject:convenienceNumber];
}
```

この場合は、allocを呼び出していません。このため、releaseを呼び出す必要がありません。新しい数字を保持する必要はありません(convenienceNumber)。それは配列が行うからです。

```
NSMutableArray *array;
NSUInteger i;
// ...
for (i = 0; i < 10; i++) {
    NSNumber *allocedNumber = [[NSNumber alloc] initWithInteger:i];
    [array addObject:allocedNumber];
    [allocedNumber release];
}
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

この場合は、forループのスコープ内でallocと対応がとれるように、allocedNumberにreleaseメッセージを送信する*必要があります*。addObject:によって数値が追加されたときに配列はそれを保持するため、数値が配列内にある間はそれは割り当て解除されません。

これを理解するために、コレクションクラスを実装する人の立場になって考えてみます。管理対象のオブジェクトが管理下から消えないようにするために、それらが渡されたときにretainメッセージを送信します。オブジェクトが削除された場合は、それに対応するようにreleaseメッセージを送信しなければなりません。また、独自のdeallocメソッドで、残りのオブジェクトにreleaseメッセージを送信する必要があります。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

自動解放プール(Autorelease Pools)

この章では、アプリケーションでの自動解放プールの扱いの微調整に関する詳細を説明します。自動解放メカニズムの使用に関する一般的な情報は「オブジェクトの所有権と破棄」（11 ページ）を参照してください。

自動解放プールの概要

自動解放プールはNSAutoreleasePoolのインスタンスで、autoreleaseメッセージを受信したオブジェクトを「含んでいます」。自動解放プールが割り当て解除されると、プールはこれらの各オブジェクトにreleaseメッセージを送信します。1つのオブジェクトは何度でも自動解放プールに入れることができます。プールに入れられた回数だけ、オブジェクトはreleaseメッセージを受信します。このように、releaseの代わりにautoreleaseをオブジェクトに送信すると、そのオブジェクトの存続期間が、少なくともプール自身が解放されるまで延長されます（その間にオブジェクトが保持された場合は、さらに長くオブジェクトが存続することもあります）。

Cocoaは、常に自動解放プールが利用可能であることを前提にしています。自動解放プールが利用できない場合は、自動解放されたオブジェクトは解放されないためメモリリークが発生します。自動解放プールが利用できないときにautoreleaseメッセージを送信すると、Cocoaは適切なエラーメッセージを記録します。

通常のallocメッセージやinitメッセージを利用してNSAutoreleasePoolオブジェクトを作成し、releaseまたはdrainを利用して破棄します（自動解放プールにautoreleaseまたはretainを送信すると、例外が発生します）。releaseとdrainの違いについては、「ガベージコレクション」（29 ページ）を参照してください。自動解放プールは、常にそれを作成したときと同じコンテキスト（メソッドや関数の呼び出し、またはループ本体）で解放しなければなりません。

複数の自動解放プールは1つのスタックに配置されます。これらは一般に「ネストしている」と呼ばれます。新しい自動解放プールを作成すると、それがスタックの一番上に追加されます。プールが割り当て解除されると、それがスタックから削除されます。オブジェクトにautoreleaseメッセージが送信されると、そのオブジェクトは現在のスレッドの一番上のプールに追加されます。

自動解放プールをネストできるということは、任意の関数やメソッドに自動解放プールを含めることができることを意味します。たとえば、main関数で自動解放プールを作成して、別の自動解放プールを作成する別の関数を呼び出すことができます。あるいは、1つのメソッドで外側のループ用に1つの自動解放プールを持ち、内側のループ用にもう1つの自動解放プールを持つこともできます。自動解放プールをネストできることは明らかに有利です。ただし、例外が発生した場合は副作用があります（「自動解放プールのスコープとネストした自動解放プールの意味」（27 ページ）を参照）。

Application Kitは、自動的にマウスダウンイベントなどのイベントサイクル（またはイベントループの反復）の最初で自動解放プールを1つ作成し、最後でそれを解放します。したがって、通常はコードの側で自動解放プールについて心配する必要はありません。ただし、次の3つのケースでは独自の自動解放プールを作成して破棄します。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

- コマンドラインツールなどのApplication Kitに基づかないプログラムを記述している場合は、自動解放プールの組み込みサポートはありません。したがって、自動解放プールを自分で作成して破棄する必要があります。
- 別のスレッドを作成した場合は、そのスレッドの実行が始まったらすぐに固有の自動解放プールを作成する必要があります。さもないと、オブジェクトがリークします（詳細については「自動解放プールとスレッド」（27 ページ）を参照）。
- たくさんの一時オブジェクトを作成するループを記述する場合は、そのループ内に自動解放プールを作成して次の反復の前にこれらのオブジェクトを破棄することができます。これは、アプリケーションの最大メモリ占有量を減らすのに役立ちます。

自動解放プールは「インライン」で使われます。*通常は、自動解放プールをオブジェクトのインスタンス変数にしなければならないという理由はありません。*

非AppKitプログラムの自動解放プール

Application Kitに基づかないプログラムで自動解放メカニズムを有効にするのは簡単です。`main()`関数の最初で自動解放プールを作成し、最後でそれを解放するだけです。これは、XcodeのFoundation Toolテンプレートで使われているパターンです。これによって、このタスクが存続する間、自動解放プールが確立します。ただしこれは、このタスクの存続期間中に作成された自動解放オブジェクトが、このタスクが終了するまで破棄されないことも意味します。そのために、このタスクのメモリ占有量が不必要に増加する可能性があります。もっと狭いスコープでプールを作成することも考えられます。

多くのプログラムには、そのほとんどの処理を実行する最上位のループがあります。自動解放メカニズムを有効にするには、このループの最初で自動解放プールを作成し、最後でそれを解放します。

`main`関数は、リスト 1のコードのようになります。

リスト 1 非AppKitプログラムの`main`関数の例

```
void main()
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NSArray *args = [[NSProcessInfo processInfo] arguments];
    unsigned count, limit = [args count];

    for (count = 0; count < limit; count++)
    {
        NSAutoreleasePool *loopPool = [[NSAutoreleasePool alloc] init];
        NSString *fileContents;
        NSString *fileName;

        fileName = [args objectAtIndex:count];
        fileContents = [[[NSString alloc] initWithContentsOfFile:fileName]
autorelease];
        // これはstringWithContentsOfFile:を使用することと等価である

        /* ファイルを処理し、いくつかのオブジェクトを作成したり自動解放したりする */
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

```
        [loopPool release];
    }

    /* クリーンアップに必要な処理をすべて実行する */
    [pool drain];

    exit (EXIT_SUCCESS);
}
```

このプログラムはコマンドラインで渡されたファイルを処理します。このforループは一度に1つのファイルを処理します。NSAutoreleasePoolオブジェクトは、このループの最初で作成され、最後で解放されます。したがって、このループ内でautoreleaseメッセージが送信されたオブジェクト（fileContentsなど）はloopPoolに追加されます。そしてループの最後でloopPoolが解放されるときに、これらのオブジェクトも解放されます。さらに、forループのコンテキストで作成された自動解放オブジェクト（fileNameなど）は、たとえそれらに明示的にautoreleaseメッセージが送信されなくても、loopPoolが解放されるときに解放されます。

自動解放プールとスレッド

Cocoaアプリケーションでは、スレッドごとに固有のNSAutoreleasePoolオブジェクトスタックを管理しています。スレッドが終了すると、それに関連付けられているすべての自動解放プールが自動的に解放されます。Application Kitに基づくアプリケーションのメインスレッドでは自動解放プールが自動的に作成されて破棄されるため、通常コードではそれらを扱う必要がありません。ただし、Application Kitのメインスレッド以外でCocoa呼び出しを行う場合は独自の自動解放プールを作成する必要があります。これは、Foundationのみのアプリケーションを記述している場合やスレッドをデタッチする場合に当てはまります。

アプリケーションやスレッドが長時間存続し、大量の自動解放オブジェクトを生成する可能性がある場合は、（Application Kitのメインスレッドで行っているのと同様に）定期的に自動解放プールを破棄して作成すべきです。さもないと、自動解放オブジェクトが累積してメモリ占有量が増加します。デタッチしたスレッドでCocoa呼び出しを行わない場合は、自動解放プールを作成する必要はありません。

注： NSThreadの代わりにPOSIXスレッドAPIを使用して別のスレッドを作成する場合は、Cocoaがマルチスレッドモードになっていないと、Cocoa（NSAutoreleasePoolを含む）を使用できません。Cocoaがマルチスレッドモードになるのは、最初のNSThreadオブジェクトをデタッチした後だけです。別のPOSIXスレッドでCocoaを使用するには、すぐに終了させることができるNSThreadオブジェクトを少なくとも1つ、最初にアプリケーションでデタッチする必要があります。Cocoaがマルチスレッドモードになっているかどうかは、NSThreadクラスのisMultiThreadedメソッドによって確認できます。

自動解放プールのスコープとネストした自動解放プールの意味

自動解放プールは、リスト1（26 ページ）に示したように、コード内の囲み要素によってネストしているのが一般的です。ネストした自動解放プールはスタック上にあると考えることができます。その際、「一番内側の」自動解放プールがスタックの一番上にあります。前述のとおり、これ

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

はネストした自動解放プールの実際の実装です。プログラム内の各スレッドは1つの自動解放プールスタックを管理しています。自動解放プールを作成すると、それが現在のスレッドのスタックの一番上にプッシュされます。*オブジェクトが自動解放される（つまり、オブジェクトに*autorelease*メッセージが送信されるか、オブジェクトが引数として*addObject:*クラスメソッドに渡される）と、そのオブジェクトはスタックの一番上にある自動解放プールに置かれます。*

したがって、自動解放プールのスコープは、スタック内での位置と単純にそれが存在するという事実によって決まります。最上位のプールは、自動解放オブジェクトが追加されるプールです。別のプールが作成されると、この新しいプールが解放されるまで現在最上位にあるプールは事実上スコープから外れます（新しいプールが解放された時点で元のプールが再び最上位プールになります）。自動解放プール自身が解放された場合は、（明らかですが）それは永久にスコープから外れます。

スタックの最上位にない自動解放プールが解放された場合は、スタック上でそれより上にあるすべての（未解放の）自動解放プールが、そのオブジェクトと一緒に解放されます。自動解放プールを使い終わったときに、それにreleaseを送信するのを忘れても（このようなことはお勧めできません）、ネストの外側の自動解放プールの1つが解放されるときにそのプールも解放されます。

この動作は例外的な状況を意味しています。例外が発生してスレッドが突然現在のコンテキストから抜けると、そのコンテキストに関連付けられていたプールが解放されます。ただしそのプールがスレッドのスタックの最上位にない場合は、解放されたプールより上のすべてのプールも解放されます（処理中のオブジェクトもすべて解放されます）。その結果、スレッドのスタックの最上位の自動解放プールは、すでに解放された例外的な状況に関連するプールの下にあったプールになります。この動作があるので、例外ハンドラはautoreleaseが送信されたオブジェクトを解放する必要がありません。また、例外ハンドラで例外が再発生しない限り、例外ハンドラは自動解放プールにreleaseを送信する必要もありません。

Foundationにおける所有権ポリシーの保証

単純にオブジェクトを解放する代わりに自動解放プールを作成することで、一時オブジェクトの存続期間をそのプールの存続期間まで拡張できます。自動解放プールを割り当て解除した後は、そのプールが存続していた間に自動解放されたオブジェクトはすべて「破棄された」と見なすべきです。したがって、そのオブジェクトにメッセージを送信したり、そのオブジェクトをメソッドの呼び出し側に返してはなりません。

自動解放のコンテキストを越えて一時オブジェクトを使用しなければならない場合は、そのコンテキスト内でオブジェクトにretainメッセージを送信します。そして、自動解放プールが解放された後にそのオブジェクトにautoreleaseを送信します。

```
- findMatchingObject:anObject
{
    id match = nil;

    while (match == nil) {
        NSAutoreleasePool *subPool = [[NSAutoreleasePool alloc] init];

        /* 多くの一時オブジェクトを作成しているものを検索する */
        match = [self expensiveSearchForObject:anObject];

        if (match != nil) {
            [match retain]; /* matchを保持し続ける */
        }
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

```
        [subPool release];
    }

    return [match autorelease];   /* matchを自動解放して返す */
}
```

subpoolが有効な間にmatchにretainを送信し、subpoolが解放されてからそれにautoreleaseを送信することによって、matchをsubpoolから以前有効だったプールに移動できます。これによってmatchの存続期間が拡張し、ループの外でメッセージを受信したりfindMatchingObject:の呼び出し側に返すことができます。

ガベージコレクション

ガベージコレクションシステム（『*Garbage Collection Programming Guide*』）は、本来、自動解放プールを使用しませんが、ハイブリッドなフレームワーク（ガベージコレクトされる環境と、参照カウントされる環境で使用できるフレームワーク）を開発している場合、自動解放プールはガベージコレクタにヒントを与えるために役立ちます。

自動解放プールは、プールに追加されたオブジェクトの所有権を放棄したいときに解放されます。これは、しばしば、イベントサイクルの終わりや、たくさんの一時オブジェクトを作成するループ内などでその時点までに累積された一時オブジェクトを破棄するために有効です。また、通常、このような場所は、ガベージコレクションが保証できそうであるというヒントをガベージコレクタに与えるのにも役立ちます。

ガベージコレクトされる環境では、releaseは何も処理を行いません。このため、NSAutoreleasePoolには、参照カウントされる環境ではreleaseを呼び出したときと同じ動作をし、ガベージコレクトされる環境ではガベージコレクションをトリガする（ただし、前回のガベージコレクション以降に割り当てられたメモリが現在のしきい値を超えている場合）drainメソッドが提供されています。したがって、通常自動解放プールを破棄するにはreleaseではなくdrainを使用すべきです。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

アクセサメソッド

この章では、アクセサメソッドを使用しなければならない理由と、アクセサメソッドの宣言と実装の方法を説明します。

アクセサメソッドを使用する主な理由の1つに、カプセル化があります（『*Object-Oriented Programming with Objective-C*』の「Encapsulation」を参照）。参照カウントされる環境では、クラスの基本的なメモリ管理のほとんどをアクセサメソッドで行うと特にメリットがあります。

アクセサメソッドの宣言

通常、アクセサメソッドを宣言するには、次の例のように、Objective-Cの宣言済みプロパティ機能を使用します。

```
@property (copy) NSString *firstName;
@property (readonly) NSString *fullName;
@property (retain) NSDate *birthday;
@property NSInteger luckyNumber;
```

この宣言によってプロパティに対するメモリ管理セマンティクスが明確になります。

アクセサメソッドの実装

ほとんどの場合、Objective-Cの宣言済みプロパティ機能を使用して次のようにコンパイラにアクセサメソッドの合成を指示すれば、独自のアクセサメソッドの実装を避けることができます（避けるべきです）。

```
@synthesize firstName;
@synthesize fullName;
@synthesize birthday;
@synthesize luckyNumber;
```

独自の実装を提供する必要がある場合でも宣言済みプロパティを使用してアクセサを宣言しなければなりません。もちろん、その実装は宣言した仕様に合っていなければなりません（特に、デフォルトでは宣言済みプロパティはアトミックであることに注意してください。アトミック実装を提供しない場合、宣言で`nonatomic`を指定する必要があります）。

単純なオブジェクト値の場合、大まかに言うとアクセサの実装には次の3つ方法があります。

1. getterは値を返す前にその値を保持または自動解放し、setterは古い値を解放してから新しい値を保持（またはコピー）する。

2. getterは値を返し、setterは古い値を自動解放してから新しい値を保持（またはコピー）する。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

3. getterは値を返し、setterは古い値を解放してから新しい値を保持（またはコピー）する。

方法1

方法1では、getterによって返された値は呼び出し側のスコープ内で自動解放されます。

```
- (NSString*) title {
    return [[title retain] autorelease];
}

- (void) setTitle:(NSString*) newTitle {
    if (title != newTitle) {
        [title release];
        title = [newTitle retain]; // または、必要に応じてコピーする
    }
}
```

返されたオブジェクトは現在のスコープ内で自動解放されるため、プロパティ値が変更された場合は有効のままです。これによりアクセサがより堅牢になりますが、余分なオーバーヘッドのコストがかかります。getterメソッドが頻繁に呼び出される場合は、オブジェクトの保持と自動解放に伴う追加コストがパフォーマンスコストに見合わない場合もあります。

方法2

方法1と同様に、方法2でも自動解放を使用します。ただし今度はsetterメソッドでそれを行います。

```
- (NSString*) title {
    return title;
}

- (void) setTitle:(NSString*) newTitle {
    [title autorelease];
    title = [newTitle retain];
}
```

getterの呼び出し頻度がsetterよりもかなり多い場合は、方法2のパフォーマンスは方法1よりかなり優れています。

方法3

方法3では自動解放をまったく使用しません。

```
- (NSString*) title {
    return title;
}

- (void) setTitle:(NSString*) newTitle {
    if (newTitle != title) {
        [title release];
        title = [newTitle retain];
    }
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

```
}
```

方法3で使われているアプローチは、getterメソッドとsetterメソッドが頻繁に呼び出される場合に適しています。また、コレクションクラスなど、値の存続期間を長くしたくないオブジェクトにも適しています。

値オブジェクトとコピー

Objective-Cのコードでは、属性を表す値オブジェクトをコピーするのが一般的です。一般に値オブジェクトの代わりにC型の変数を使用できます。ただし、値オブジェクトには、よく使う操作のための便利なユーティリティをカプセル化しているという利点があります。たとえば、`NSString`オブジェクトは、エンコードと格納をカプセル化しているため文字ポインタの代わりに使われます。

値オブジェクトがメソッドの引数またはメソッドの戻り値として渡された場合は、そのオブジェクト自身ではなくコピーを使用するのが一般的です。たとえば、あるオブジェクトの`name`インスタンス変数に文字列を代入するために、次のメソッドを考えます。

```
- (void)setName:(NSString *)aName {
    [name autorelease];
    name = [aName copy];
}
```

`aName`のコピーを格納することは、元のオブジェクトとは独立していながら同じ内容を持つオブジェクトを作成することになります。その後でコピーに変更を加えても元のオブジェクトには影響しません。また、元のオブジェクトに変更を加えてもコピーには影響しません。同様に、インスタンス変数自身ではなくインスタンス変数のコピーを返すのが一般的です。たとえば、次のメソッドは`name`インスタンス変数のコピーを返します。

```
- (NSString *)name {
    return [[name copy] autorelease];
}
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

オブジェクトコピーの実装

この章では、オブジェクトのコピーのためにNSCopyingプロトコルのcopyWithZone:メソッドを実装する2つのアプローチについて説明します。

NSCopyingプロトコルのcopyWithZone:メソッドを実装することによってコピーを作成するには、2つの基本的なアプローチがあります。allocとinit...を使用する方法と、NSCopyObjectを使用する方法です。クラスに適した方法を選択するには、次の問いに対する答えを考える必要があります。

- 深いコピーが必要か、浅いコピーが必要か。
- NSCopyingの動作をスーパークラスから継承するか。

以降のセクションではこれらについて説明します。

深いコピーと浅いコピー

一般に、オブジェクトのコピーでは新しいインスタンスを作成してそれを元のオブジェクトの値で初期化する処理を行います。ポインタでないインスタンス変数（ブール値、整数、浮動小数点数など）の値をコピーする場合は簡単です。ポインタのインスタンス変数をコピーする場合は2つのアプローチがあります。1つ目のアプローチは、浅いコピーと呼ばれ元のオブジェクトのポインタ値をコピー先にコピーします。これによって、元のオブジェクトとコピー先のオブジェクトが参照先のデータを共有します。もう1つのアプローチは、深いコピーと呼ばれ、ポインタで参照されているデータを複製してコピー先のインスタンス変数に代入します。

インスタンス変数のsetメソッドの実装には、使用するコピーの種類を反映する必要があります。次のメソッドのようにsetメソッドが新しい値をコピーする場合は、インスタンス変数の深いコピーを実行することになります。

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue copy];
}
```

次のメソッドのようにsetメソッドが新しい値を保持する場合は、インスタンス変数の浅いコピーを実行することになります。

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue retain];
}
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

同様に、次の例のように、setメソッドで新しい値をコピーしたり保持したりせずにインスタンス変数に代入するだけの場合は、インスタンス変数の浅いコピーを実行することになります（ただし、このようなケースはめったにありません）。

```
- (void)setMyVariable:(id)newValue
{
    myVariable = newValue;
}
```

独立したコピー

元のオブジェクトから本当に独立したオブジェクトのコピーを作成するには、オブジェクト全体の深いコピーを実行しなければなりません。すべてのインスタンス変数を複製しなければなりません。インスタンス変数自体がインスタンス変数を持つ場合は、それらも複製する必要があります。多くの場合、アプローチを併用した方が有用です。データのコンテナと見なすことができるポインタインスタンス変数には一般に深いコピーを実行し、デリゲートのような高度なインスタンス変数には浅いコピーを実行します。

```
@interface Product :NSObject <NSCopying>
{
    NSString *productName;
    float price;
    id delegate;
}

@end
```

たとえば、Productクラスでは`NSCopying`を採用しています。Productインスタンスは、このインターフェイスで宣言されているように、名前、価格、およびデリゲートを持っています。

Productインスタンスをコピーすると、`productName`は単純なデータ値を表すため深いコピーが作成されます。一方、`delegate`インスタンス変数は、両方（コピー先とコピー元）のProductに対して適切な機能を提供する、より複雑なオブジェクトです。したがって、コピー先とコピー元はこのデリゲートを共有します。図 1 （36 ページ）は、Productインスタンスとそのコピーのメモリ内のイメージを表しています。

図 1　浅いコピーと深いコピーの両方を利用したインスタンス変数のコピー

original	0xf2ae4
isa	0x8028
productName	0xf2bd8
price	0.00
delegate	0xe83c8

copy	0x104074
isa	0x8028
productName	0xe81f4
price	0.00
delegate	0xe83c8

`productName`のポインタ値が異なるのは、元のオブジェクトとコピーがそれぞれ固有の`productName`文字列オブジェクトを持っていることを示しています。`delegate`のポインタ値は同じです。これは、2つの製品オブジェクトが同じオブジェクトをデリゲートとして共有していることを示しています。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

スーパークラスからのNSCopyingの継承

スーパークラスがNSCopyingを実装していない場合は、クラスの実装で、クラスで宣言したインスタンス変数だけでなく、継承したインスタンス変数もコピーしなければなりません。一般に、そのための一番安全な方法は、alloc、init...、およびsetの各メソッドを使用することです。

一方、クラスがNSCopyingの動作を継承しており、追加のインスタス変数を宣言している場合は、copyWithZone:も使用する必要があります。このメソッドでは、スーパークラスの実装を呼び出して、継承したインスタンス変数をコピーしてから、追加のインスタンス変数をコピーします。新しいインスタンス変数をどのように扱うかは、スーパークラスの実装をどれだけ知っているかによります。スーパークラスがNSCopyObjectを使用しているか、または使用しているかもしれない場合は、allocとinit...を使用するときとは異なる方法でインスタンス変数を扱わなければなりません。

「alloc、init...」アプローチの使用

クラスがNSCopying動作を継承していない場合は、alloc、init...、およびsetの各メソッドを使用してcopyWithZone:を実装する必要があります。たとえば、「独立したコピー」（36 ページ）で説明したProductクラスのcopyWithZone:の実装は次のようになります。

```
- (id)copyWithZone:(NSZone *)zone
{
    Product *copy = [[[self class] allocWithZone:zone]
            initWithProductName:[self productName]
            price:[self price]];
    [copy setDelegate:[self delegate]];

    return copy;
}
```

継承したインスタンス変数に関連する実装の詳細はスーパークラスにカプセル化されているため、alloc、init...アプローチでNSCopyingを実装する方が一般に優れています。それには、setメソッドで実装されているポリシーを使用して、インスタンス変数に必要なコピーの種類を決定します。

NSCopyObject()の使用

クラスがNSCopying動作を継承している場合は、スーパークラスの実装がNSCopyObject関数を使用している可能性を検討する必要があります。NSCopyObjectは、インスタンス変数の値をコピーし、参照先のデータはコピーしない、まさしくオブジェクトの浅いコピーを作成します。たとえば、NSCellのcopyWithZone:の実装は、次のように定義できます。

```
- (id)copyWithZone:(NSZone *)zone
{
    NSCell *cellCopy = NSCopyObject(self, 0, zone);
    /* ここでその他の初期化が行われる */

    cellCopy->image = nil;
    [cellCopy setImage:[self image]];
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

```
    return cellCopy;
}
```

上の実装では、NSCopyObjectによって元のセルのまさしく浅いコピーが作成されます。この動作は、ポインタでないインスタンス変数をコピーしたり、浅くコピーする保持対象外のデータへのポインタであるインスタンス変数をコピーしたりする場合に適しています。保持されているオブジェクトのポインタインスタンス変数の場合は、それ以外の処理が必要です。

前述のcopyWithZone:の例で、imageは、保持されているオブジェクトへのポインタです。imageを保持する際のポリシーは、 setImage:アクセサメソッドの次の実装に反映されています。

```
- (void)setImage:(NSImage *)anImage
{
    [image autorelease];
    image = [anImage retain];
}
```

setImage:では、imageに再代入する前にそれを自動解放している点に注意してください。上のcopyWithZone:の実装で、setImage:を呼び出す前に、コピーのimageインスタンス変数が明示的にnilに設定されていない場合は、コピー先とコピー元によって参照されるimageは、対応する保持の処理が行われないまま解放されます。

たとえimageが正しいオブジェクトを指していても、概念的には初期化されません。allocとinit...によって作成されたインスタンス変数とは違って、初期化されないこれらの変数はnilになりません。これらの変数を使用する前には、明示的に初期値を代入する必要があります。この例では、cellCopyのimageインスタンス変数がnilに設定された後で、setImage:メソッドを使用して設定されています。

NSCopyObjectの効果はサブクラスの実装にも及びます。たとえば、NSSliderCellの実装では、次のように新しいtitleCellインスタンス変数がコピーされます。

```
- (id)copyWithZone:(NSZone *)zone
{
    id cellCopy = [super copyWithZone:zone];
    /* ここでその他の初期化が行われる */

    cellCopy->titleCell = nil;
    [cellCopy setTitleCell:[self titleCell]];

    return cellCopy;
}
```

ここでは、superのcopyWithZone:メソッドが次のような処理を実行することを前提としています。

```
id copy = [[[self class] allocWithZone:zone] init];
```

スーパークラスのcopyWithZone:メソッドが呼び出されて、継承したインスタンス変数がコピーされます。スーパークラスのcopyWithZone:メソッドを呼び出すときは、スーパークラスの実装でNSCopyObjectを使用している可能性がある場合は、新しいオブジェクトインスタンス変数が初期化されないことを前提にします。変数を使用する前に、明示的にそれらに値を代入します。この例では、setTitleCell:が呼び出される前に、titleCellが明示的にnilに設定されています。

NSCopyObjectを使用する場合は、オブジェクトの保持カウントの実装も考慮すべき点です。オブジェクトがインスタンス変数に保持カウントを格納する場合、copyWithZone:の実装で、コピー先の保持カウントを正しく初期化しなければなりません。図 2（39 ページ）はこの処理を示しています。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

図 2 コピー中の参照カウントの初期化

original	0xf2ae4
isa	0x8028
refCount	3
productName	0xf2bd8
price	0.00
delegate	0xe83c8

copy	0x104074
isa	0x8028
refCount	3
productName	0xf2bd8
price	0.00
delegate	0xe83c8

The copy produced by **NSCopyObject**

copy	0x104074
isa	0x8028
refCount	1
productName	0xe81f4
price	0.00
delegate	0xe83c8

The copy after unitialized instance variables are assigned in **copyWithZone:**

図 2（39 ページ）の最初のオブジェクトは、メモリ内のProductインスタンスを表しています。refCountの値は、このインスタンスが3回保持されたこと示しています。2番目のオブジェクトは、NSCopyObjectによって作成されたProductインスタンスのコピーです。そのrefCountの値は、元のオブジェクトと同じです。3番目のオブジェクトは、refCountが正しく初期化された後にcopyWithZone:から返されたコピーを表しています。copyWithZone:はNSCopyObjectを利用してコピーを作成した後で、refCountインスタンス変数に値1を代入します。copyWithZone:の呼び出し側は、暗黙的にこのコピーを保持し、それを解放する責任を負います。

可変(Mutable)オブジェクトと不変(Immutable)オブジェクトのコピー

「不変と可変」の概念をオブジェクトに適用すると、NSCopyingは元のオブジェクトが不変かどうかに関わらず不変コピーを作成します。不変クラスでは、NSCopyingを非常に効率的に実装できます。不変オブジェクトは変化しないため、それらを複製する必要はありません。その代わりに、NSCopyingは元のオブジェクトをretainするように実装できます。たとえば、不変の文字列クラスのcopyWithZone:は、次のように実装できます。

```
- (id)copyWithZone:(NSZone *)zone {
    return [self retain];
}
```

オブジェクトの可変コピーを作成するには、NSMutableCopyingプロトコルを使用します。可変コピーをサポートするために、オブジェクト自身が可変である必要はありません。このプロトコルではmutableCopyWithZone:メソッドを宣言します。可変コピーは、一般に、便利なNSObjectのmutableCopyメソッドを利用して呼び出されます。このメソッドは、デフォルトゾーンを引数としてmutableCopyWithZone:を呼び出します。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

CocoaでのCore Foundationオブジェクトのメモリ管理

多くのCore FoundationインスタンスとCocoaインスタンスは、CFStringオブジェクトとNSStringオブジェクトのように、単純に互いに型キャストできます。この章では、CocoaでCore Foundationオブジェクトを管理する方法について説明します。オブジェクトの所有権についての一般的な情報は「オブジェクトの所有権と破棄」（11 ページ）を参照してください。

重要： この章では、参照カウントされる環境におけるCocoaおよびCore Foundationの使用について説明します。ガベージコレクションを使用している場合は、セマンティクスが異なります（『*Garbage Collection Programming Guide*』を参照）。

Core Foundationのメモリ割り当てポリシーは、名前に「Copy」または「Create」を含む関数によって返された値は解放する必要があり、名前に「Copy」または「Create」を含まない関数によって返された値は解放してはならないというものです。

Core FoundationとCocoaで使われる規約は共によく似ています。allocation／retain／releaseの実装に互換性があるため、それぞれの環境の等価な関数やメソッドを、混在させて使用できます。したがって、

```
NSString *str = [[NSString alloc] initWithCharacters:...];
...
[str release];
```

これは、次の記述と同等です。

```
CFStringRef str = CFStringCreateWithCharacters(...);
 ...
CFRelease(str);
```

および

```
NSString *str = (NSString *)CFStringCreateWithCharacters(...);
 ...
[str release];
```

および

```
 NSString *str = (NSString *)CFStringCreateWithCharacters(...);
 ...
[str autorelease];
```

これらのコード例が示すように、一度作成されて型キャストされたオブジェクトは、CocoaまたはCore Foundationとして扱うことができ、それぞれの環境で「ネイティブ」のように見えます。

Core FoundationおよびCarbonのデータ型の扱いの詳細については、『*Carbon-Cocoa Integration Guide*』の「Interchangeable Data Types」のセクションを参照してください。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

nibオブジェクトのメモリ管理

Cocoaアプリケーションでは、アプリケーションが起動した瞬間から終了する瞬間までのさまざまな時点で、1つ以上のnibファイルがロードされ、それに含まれているオブジェクトが展開されます。これらのオブジェクトが不要になったときにそれを解放する責任は、開発対象のプラットフォームによって異なります。また、Mac OS Xでは、File's Ownerがどのクラスから派生しているかによって異なります。

nibファイルとそのメモリ管理セマンティクスの基本的な説明、およびnibに関連する用語（「アウトレット」、「File's Owner」、「最上位オブジェクト」など）の定義については、『*Resource Programming Guide*』の「Nib Files」を参照してください。

アウトレット

nibファイルをロードしてアウトレットを作成するときに、nibロードメカニズムは、（Mac OS X デスクトップとiPhoneのどちらの場合も）アクセサメソッドが存在する場合は常にそれを使用します。したがって、どのプラットフォームが開発対象の場合でも、通常はObjective-Cの宣言済みプロパティを使用してアウトレットを宣言する必要があります。

宣言の一般的な形式は次のようになります。

```
@property (attributes) IBOutlet UserInterfaceElementClass *anOutlet;
```

アウトレットの動作はプラットフォームに依存するため（「Mac OS Xデスクトップ」（44 ページ）および「iPhone」（45 ページ）を参照）、実際の宣言とは異なります。

- Mac OS Xでは次のように使用します。

  ```
  @property (assign) IBOutlet UserInterfaceElementClass *anOutlet;
  ```

- iPhone OS Xでは次のように使用します。

  ```
  @property (nonatomic, retain) Outlet UIUserInterfaceElementClass *anOutlet;
  ```

次に、それに対応するアクセサメソッドを合成するか、または宣言に従って実装します。そして、（iPhone OS内で）deallocで対応する変数を解放します。

最新のランタイムを使ってインスタンス変数を合成する場合、このパターンは同様に機能します。そして、すべての状況に渡って一貫性を保持します。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

Mac OS Xデスクトップ

nibファイルのFile's Ownerは、デフォルトで、nibファイル内のオブジェクトによって作成された非オブジェクトリソースだけでなく、nibファイル内の最上位オブジェクトを解放する責任があります。オブジェクトグラフのルートオブジェクトを解放すると、それに依存するすべてのオブジェクトが解放されます。アプリケーションのメインnibファイル（アプリケーションのメニュー、その他の項目を含む）のFile's Ownerは、グローバルなアプリケーションオブジェクトNSAppです。ただし、Cocoaアプリケーションが終了するときに、メインnib内の最上位オブジェクトは自動的にはdeallocメッセージを受けません。それは、NSAppが割り当て解除されているからです（「オブジェクトの割り当て解除」 （17 ページ）も参照）。言い換えると、メインnibファイル内にあっても最上位オブジェクトのメモリは管理する必要があります。

Application Kitでは、nibオブジェクトを正しく解放することを保証するために役立つ、次の2つの機能を提供しています。

- NSWindowオブジェクト（パネルを含む）はisReleasedWhenClosed属性を持ちます。この属性がYESに設定されていると、ウインドウは閉じるときに自分自身を解放します（その結果、ビュー階層内のすべての依存オブジェクトも解放されます）。Interface Builderでは、インスペクタの「Attributes」ペインの「Release When Closed」チェックボックスでこのオプションを設定します。
- nibファイルのFile's OwnerがNSWindowControllerオブジェクトの場合（ドキュメントベースのアプリケーションのドキュメントnibのデフォルト。NSDocumentはNSWindowControllerのインスタンスを管理することを思い出してください）は、File's Ownerが管理対象のウインドウを自動的に破棄します。

したがって、一般に、nibファイルの最上位オブジェクトを解放する責任はデベロッパにあります。ただし実際には、nibのFile's OwnerがNSWindowControllerのインスタンスの場合は、それが最上位オブジェクトを解放してくれます。オブジェクトの1つがnib自体をロードする場合（かつ、所有者がNSWindowControllerのインスタンスでない場合）、最上位オブジェクトごとにアウトレットを定義して、適切なタイミングでこれらの参照を使用して最上位オブジェクトを解放することもできます。すべての最上位オブジェクトに対してはアウトレットを持ちたくない場合は、NSNibクラスのinstantiateNibWithOwner:topLevelObjects:メソッドを使用して、nibファイルの最上位オブジェクトの配列を取得できます。

さまざまな種類のアプリケーションを検討すると、nibオブジェクトを破棄する責任の問題がより明確になります。ほとんどのCocoaアプリケーションは、シングルウインドウアプリケーションとドキュメントベースアプリケーションの2種類に分けられます。どちらの場合も、nibオブジェクトのメモリ管理はある程度までは自動的に処理されます。シングルウインドウアプリケーションの場合、メインnibファイルのオブジェクトは、アプリケーションが起動した瞬間から終了する瞬間までの間存続し、アプリケーションが終了すると解放されます。ただし、アプリケーションが終了するときに、メインnibファイルのオブジェクトに対して自動的にdeallocが呼び出される保証はありません。ドキュメントベースのアプリケーションでは、各ドキュメントウインドウは、ドキュメントnibファイルのメモリ管理を処理するNSWindowControllerオブジェクトによって管理されます。

nibファイルと最上位オブジェクトが複雑に配置されているアプリケーションもあります。たとえば、1つのアプリケーションが複数のウインドウコントローラ、ロード可能パネル、およびインスペクタを含む複数のnibファイルを持つ場合です。ただし、このようなケースのほとんどは、NSWindowControllerオブジェクトを使用してウインドウとパネルを管理したり、「Released When Closed」ウインドウ属性を使用すれば、メモリ管理の大部分を処理できます。ウインドウコントローラを使用しないことにして、「Release When Closed」属性を設定しない場合は、ウインドウが閉じるときにnibファイルのウインドウと最上位オブジェクトを明示的に解放する必要があります。ま

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

た、アプリケーションでインスペクタパネルを使用する場合は、（必要になった時点でそれがロードされた後は)通常、そのパネルはアプリケーションが起動した瞬間から終了する瞬間までの間存続します。インスペクタとそのリソースを破棄する必要はありません。

iPhone

最上位オブジェクト

nibファイル内のオブジェクトは、保持カウント1で作成され、その後自動解放されます。オブジェクト階層を再構築する際に、UIKitは`setValue:forKey:`を使用してオブジェクト間の接続を再構築します。このメソッドは利用可能なsetterメソッドを使用します。利用可能なsetterがない場合はデフォルトでオブジェクトを保持します。つまり、（「アウトレット」（43 ページ）で示したパターンに従うとすると）*アウトレットを持つオブジェクトは*いずれも有効のままになります。ただし、アウトレットに格納しない*最上位*オブジェクトがある場合は、`loadNibNamed:owner:options:`メソッドによって返された配列、またはその配列内のオブジェクトを保持して、これらのオブジェクトが早まって解放されるのを防ぐ必要があります。

メモリ警告

View Controllerは、メモリ警告(`didReceiveMemoryWarning`)を受信すると、現在は不要でも後から必要に応じて作成可能なリソースの所有権を放棄しなければなりません。そのようなリソースの1つに、View Controllerのビュー自身があります。ビューがスーパービューを持っていない場合は、そのビューは破棄されます（`didReceiveMemoryWarning`の実装の中で、`UIViewController`は`[self setView:nil]`を呼び出します）。

ただし、nibファイル内の要素へのアウトレットは通常保持されているため（「アウトレット」（43 ページ）を参照）、メインビューが破棄されたとしても、それ以上のアクションがなければアウトレットは破棄されません。これは特に問題ではありません。メインビューが再ロードされたときにアウトレットは単純に置き換えられるからです。ただし、`didReceiveMemoryWarning`の効果は減少します。

アウトレットの所有権が適切に放棄されることを保証するには、カスタムView Controllerで次のように`setView:`を実装します。

```
- (void)setView:(UIView *)aView {
    if (!aView) { // ビューはnilに設定されている
        // たとえば、アウトレットをnilに設定する
        self.anOutlet = nil;
    }
    // 最後に親の実装を呼び出す
    [super setView:aView];
}
```

残念なことに、現状ではこれによって別の問題が生じます。現在、`UIViewController`では（単純に変数を直接解放するのではなく）`setView:`アクセサメソッドを使用して`dealloc`メソッドを実装しているため、メモリ警告に応答するときだけでなく、`dealloc`内でも`self.anOutlet = nil`が呼び出されてしまいます。アウトレットの所有者がこのView Controllerだけだとすると、`dealloc`内でクラッシュが生じます。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

したがって、deallocでも次のようにアウトレットの変数をnilに設定する必要があります。

```
- (void)dealloc {
    // アウトレットを解放して、アウトレット変数をnilに設定する
    [anOutlet release], anOutlet = nil;
    [super dealloc];
}
```


2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

書類の改訂履歴

この表は「*Cocoaメモリ管理プログラミングガイド*」の改訂履歴です。

日付	メモ
2009-08-18	関連する概念へのリンクを追加しました。
2009-07-23	Mac OS Xでのアウトレットの宣言についてのガイダンスを更新しました。
2009-05-06	誤植を修正しました。
2009-03-04	誤植を修正しました。
2009-02-04	「Nibオブジェクト」に関する章を更新しました。
2008-11-19	ガベージコレクトされる環境での自動解放プールの使用に関するセクションを追加しました。
2008-10-15	欠落していた画像を修正しました。
2008-02-08	『Carbon-Cocoa Integration Guide』へのリンク切れを修正しました。
2007-12-11	誤植を修正しました。
2007-10-31	Mac OS X v10.5向けに更新しました。細かい誤植を修正しました。
2007-06-06	細かい誤植を修正しました。
2007-05-03	誤植を修正しました。
2007-01-08	nibファイルのメモリ管理に関する章を追加しました。
2006-06-28	deallocおよびアプリケーションの終了に関する注意を追加しました。
2006-05-23	この文書内の章を再編成し、フローを改善しました。「オブジェクトの所有権と破棄」を更新しました。
2006-03-08	オブジェクトの所有権とdeallocに関する説明を分かりやすくしました。アクセサメソッドの説明を別の章に移動しました。
2006-01-10	誤植を修正しました。文書名を『Memory Management』から更新しました。
2004-08-31	関連トピックのリンクを変更し、トピックの紹介を更新しました。
2003-06-06	「自動解放プール(Autorelease Pools)」（25 ページ）の自動解放プールが解放されると何が解放されるかの説明に、明示的および暗黙的自動解放オブジェクトの説明を追加しました。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.

日付	メモ
2003-06-03	「CocoaでのCore Foundationオブジェクトのメモリ管理」（41 ページ）に『*Integrating Carbon and Cocoa in Your Application*』へのリンクを追加しました。
2002-11-12	既存のトピックに改訂履歴を追加しました。改訂履歴を使って、トピックの内容への変更点を記録していきます。

2009-08-18 | © 2009 Apple Inc. All Rights Reserved.