高度なメモリ管理
プログラミングガイド

Developer

目次

メモリ管理について 4

初めに 4

- 正しい方針で開発すればメモリ関係の問題は生じにくい 5
- 分析ツールを使ってメモリ管理上の問題を解決する 6

メモリ管理の方針 7

基本的なメモリ管理方法 7

- 簡単な例 8
- autoreleaseを使って、releaseメッセージを「遅らせて」送信する 8
- 参照により返されたオブジェクトを所有しない 9

オブジェクトの所有権を放棄するdeallocを実装する 10

Core Foundationの管理規則は似ているが、いくつか違いもある 11

実践的なメモリ管理 12

アクセサメソッドを使ってメモリを管理しやすくする 12

- アクセサメソッドでプロパティ値を設定する 13
- 初期化メソッドやdeallocではアクセサメソッドを使わない 14

保持の循環を回避するために弱い参照を使う 15

使用中のオブジェクトの割り当て解除を避ける 17

希少なリソースはdeallocしない 18

コレクションは自分に収容されるオブジェクトを所有する 19

所有権に関する処理は保持カウントを使って実装されている 20

自動解放プールブロックの利用 21

自動解放プールブロックについて 21

ローカルな自動解放プールブロックを使ってピーク時のメモリ消費量を減らす 22

自動解放プールブロックとスレッド 24

書類の改訂履歴 25

実践的なメモリ管理 12

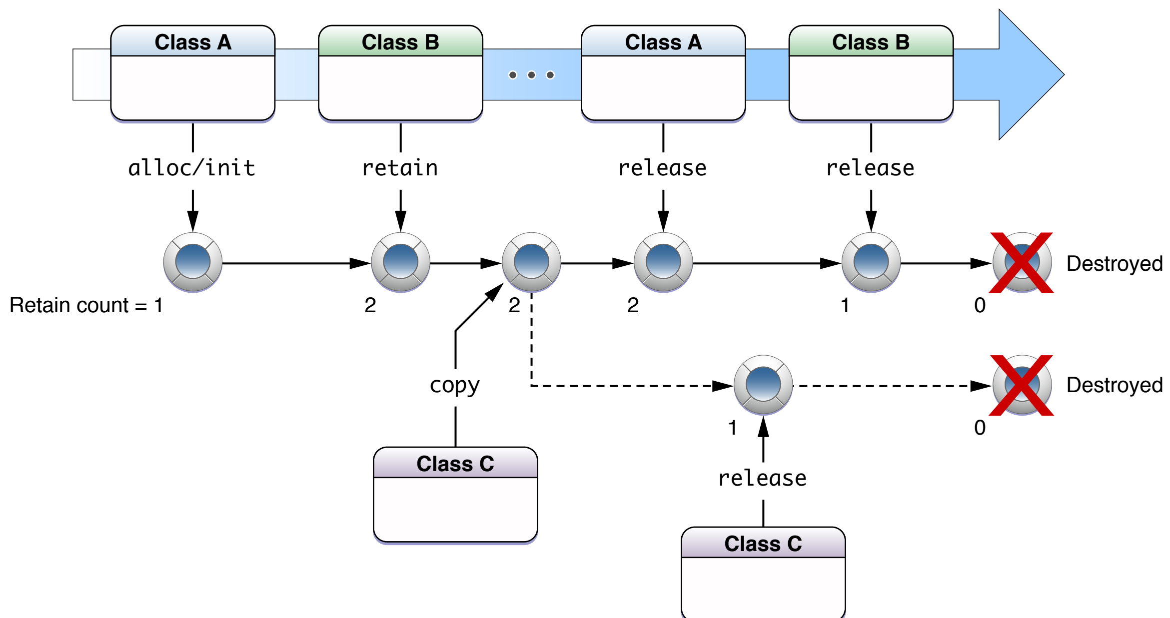
図 1 循環参照の例 16

© 2012 Apple Inc. All Rights Reserved.

メモリ管理について

アプリケーションメモリの管理とは、プログラム実行時にメモリを割り当て、これを使い、用済みになったら解放する、というプロセスのことです。高品質のプログラムは、必要以上にメモリを使わないものです。Objective-Cでは、限りあるメモリリソースの所有権を、数多くのデータやコードに適切に配分する方法のこと、と考えてもよいでしょう。このガイドに従って学習を進めれば、オブジェクトのライフサイクルをきちんと管理し、不要になったら解放するなど、アプリケーションメモリの管理に必要な知識が身につきます。

メモリ管理は通常、個々のオブジェクトレベルで考えられていますが、実はここでの目標は、オブジェクトグラフを管理することです。メモリ上には本当に必要なオブジェクトしか置かないようにしたいはずです。



初めに

Objective-Cにはアプリケーションメモリの管理手段が2つあります。

2012-07-17 | © 2012 Apple Inc. All Rights Reserved.

1. このガイドで説明する方法は**MRR**（Manual Retain-Release、明示的に記述したコードによる獲得と解放）と呼ばれるもので、開発者自身がオブジェクトの生死を追跡し、メモリを管理することになります。「参照カウント」というモデルに基づき実装されており、Foundationクラス`NSObject`が、実行時環境と連携しながらその処理を行うようになっています。
2. **ARC**（Automatic Reference Counting、自動参照カウント）という方法では、システムはMRRと同じ「参照カウント」のシステムを使いますが、必要なメモリ管理メソッドの呼び出しを、コンパイル時に自動的に挿入するようになっています。新規プロジェクトではこのARC法をぜひ採用するようお勧めします。ARC法を採用すれば、この資料で説明する実装詳細を理解していなくても構いませんが、知っていれば役に立つこともあるでしょう。ARCについて詳しくは、『*Transitioning to ARC Release Notes*』を参照してください。

iOS用のコードを開発するのであれば、メモリ管理の機構を明示的に使わなければなりません（このガイドの主題）。さらに、ライブラリルーチンやプラグイン、共有コード（ガベージコレクションをするプロセス、しないプロセスのどちらにもロードされうるコード）を開発するのであれば、このガイドで説明するメモリ管理技術を活用する必要があるでしょう（もちろん、ガベージコレクションの有効/無効を切り替え、両方の場合について、Xcodeでコードをテストする必要があります）。

正しい方針で開発すればメモリ関係の問題は生じにくい

メモリ管理が不適切であるために生じる問題は、大きく分けて2種類あります。

- まだ使用中のデータを解放し、あるいは上書きしてしまう

  その結果、メモリの内容が破損し、アプリケーションがクラッシュするばかりでなく、最悪の場合にはデータを失うことになってしまいます。

- もう使っていないデータを解放しない（メモリリーク）

  メモリリークとは、割り当てたメモリが不要になったのに、解放されないままになることを言います。その結果、メモリ消費量が肥大し続け、システム性能が損なわれ、（iOSの場合は）アプリケーションが停止してしまうことになります。

しかしながら、参照カウントを中心にメモリ管理を考えるのは、あまり生産的とは言えません。最終的な目標ではなく、実装詳細の側からメモリ管理を検討することになりがちだからです。むしろ、オブジェクトの所有権やオブジェクトグラフの観点から理解する方がよいでしょう。

Cocoaでは名前づけ規約を工夫して、どのような場合にメソッドから返されたオブジェクトが自分の所有になるか、すぐに分かるようにしています。

“メモリ管理の方針”（7 ページ）を参照してください。

基本方針は単純ですが、メモリ管理を容易にし、プログラムの信頼性や頑健性を維持しつつ必要なリソースを最小限に抑えるためには、具体的な手順を踏んで開発を進める必要があります。

© 2012 Apple Inc. All Rights Reserved.

“実践的なメモリ管理” （12 ページ）を参照してください。

自動解放プールブロックは、releaseメッセージをオブジェクトに、「遅らせて」送信する仕組みです。オブジェクトの所有権は放棄するけれども、即座に割り当て解除されてしまうのも困る、という状況で役に立ちます（メソッドからオブジェクトを返す場合など）。また、独自に自動解放プールブロックを用意して使うべき場合もあります。

“自動解放プールブロックの利用” （21 ページ）を参照してください。

分析ツールを使ってメモリ管理上の問題を解決する

コード内に潜む問題をコンパイル時に見つけるためには、Xcodeに組み込みのClang Static Analyzerが役立ちます。

それでもメモリ管理上の問題が発生する場合は、問題箇所を特定、診断する、他のツールや技法を検討してください。

- Technical Note TN2239『*iOS Debugging Magic*』にはさまざまなツールや技法が記載されています。特にNSZombieは、誤って解放してしまったオブジェクトを検出するために有用です。
- Instrumentsを使うと、参照カウントに関するイベントを追跡し、メモリリークを検出できます。“Collecting Data on Your App”を参照してください。

© 2012 Apple Inc. All Rights Reserved.

メモリ管理の方針

参照カウント環境におけるメモリ管理に使われる基本モデルは、NSObjectのプロトコルで定義されるメソッドと、標準的なメソッド命名規約から成ります。NSObjectクラスにもdeallocメソッドが定義されており、オブジェクトの割り当て解除の際、自動的に起動されるようになっています。この節では、Cocoaプログラムで正しくメモリを管理するために知っておくべき、基本的な方法を説明し、正しい使い方の例を紹介します。

基本的なメモリ管理方法

メモリ管理モデルは、オブジェクトの所有という概念に基づいています。1つのオブジェクトは1つ以上の所有者を持つ。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します。オブジェクトを所有する場合としない場合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**

  オブジェクトの作成は、「alloc」、「new」、「copy」、「mutableCopy」で始まる名前のメソッド（たとえば、alloc、newObject、mutableCopy）で行います。

- **retainメソッドでオブジェクトの所有権を獲得できる**

  一般に、（オブジェクト作成メソッドから）受け取ったオブジェクトは、そのメソッドから抜けるまで有効であることが保証されています。また、呼び出し元にそのオブジェクトを返しても安全です。retainメソッドは、（1）アクセサメソッドやinitメソッドの実装で、プロパティ値として格納するオブジェクトの所有権を得るため、（2）他の処理の副作用でオブジェクトが無効になるのを防ぐため（"使用中のオブジェクトの割り当て解除を避ける"（17 ページ）を参照）に使います。

- **所有するオブジェクトが不要になったら、その所有権を放棄する**

  オブジェクトの所有権の放棄は、releaseメッセージまたはautoreleaseメッセージを送ることによって行います。そのため、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。

- **自分が所有していないオブジェクトの所有権を放棄することはできない**

  これは前項までの規則の帰結にすぎませんが、念のためここで述べておきます。

© 2012 Apple Inc. All Rights Reserved.

簡単な例

以上の方針を適用する例として、次のコードを考えてみましょう。

```
{
    Person *aPerson = [[Person alloc] init];
    // ...
    NSString *name = aPerson.fullName;
    // ...
    [aPerson release];
}
```

Personオブジェクトを`alloc`メソッドで作成し、その後、不要になった時点で`release`メッセージを送信しています。（personの）nameは、所有するメソッドで作成したものではないので、`release`メッセージを送信していません。この例では、`autorelease`ではなく`release`メッセージを使っていることに注意してください。

autoreleaseを使って、releaseメッセージを「遅らせて」送信する

`release`メッセージの送信を遅らせたい（メソッドからオブジェクトを返すためなど）場合は、`autorelease`を使います。この方針で`fullName`メソッドを実装した例を示します。

```
- (NSString *)fullName {
    NSString *string = [[[NSString alloc] initWithFormat:@"%@ %@",
                                          self.firstName, self.lastName]
autorelease];
    return string;
}
```

`alloc`の戻り値であるstringは自分自身が所有しています。メモリ管理規則に従い、stringの所有権は、どこからも参照されなくなる前に放棄しなければなりません。しかしここで`release`を使うと、stringは呼び出し元に返す前に割り当て解除され、したがってこのメソッドは無効なオブジェクトを返すことになります。代わりに`autorelease`を使えば、所有権は放棄するけれどもすぐには割り当て解除されないので、メソッドの呼び出し元では戻り値stringを安全に使用できます。

`fullName`メソッドは次のように実装することも可能です。

```
- (NSString *)fullName {
```

© 2012 Apple Inc. All Rights Reserved.

```
    NSString *string = [NSString stringWithFormat:@"%@ %@",
                             self.firstName, self.lastName];
    return string;
}
```

基本規則により、stringWithFormat:の戻り値であるstringを所有してはいないので、呼び出し元に返しても安全です。

比較のため、*誤った例*も見てみましょう。

```
- (NSString *)fullName {
    NSString *string = [[NSString alloc] initWithFormat:@"%@ %@",
                                     self.firstName, self.lastName];
    return string;
}
```

名前づけ規約に従えば、fullNameメソッドの呼び出し元が戻り値stringを所有している、ということにはなりません。したがって呼び出し元がstringを解放することはなく、メモリリークが生じます。

参照により返されたオブジェクトを所有しない

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（引数型がClassName **やid *であるもの）。よくあるパターンとしては、initWithContentsOfURL:options:error: (NSData)、initWithContentsOfFile:encoding:error: (NSString) のように、発生したエラーに関する情報を保持するNSErrorオブジェクトを返すものがあります。

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、次のように、オブジェクトを解放する必要はありません。

```
NSString *fileName = <#Get a file name#>;
NSError *error;
NSString *string = [[NSString alloc] initWithContentsOfFile:fileName
                        encoding:NSUTF8StringEncoding error:&error];
if (string == nil) {
    // エラー処理...
}
```

2012-07-17 | © 2012 Apple Inc. All Rights Reserved.

```
// ...
[string release];
```

オブジェクトの所有権を放棄するdeallocを実装する

`NSObject`クラスには`dealloc`メソッドが定義されています。これは、オブジェクトの所有者がなくなると呼び出され、その結果、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。`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の所有権を含む）を破棄することです。

Personクラスの`dealloc`メソッドを実装する例を示します。

```
@interface Person : NSObject
@property (retain) NSString *firstName;
@property (retain) NSString *lastName;
@property (assign, readonly) NSString *fullName;
@end

@implementation Person
// ...
- (void)dealloc
    [_firstName release];
    [_lastName release];
    [super dealloc];
}
@end
```

© 2012 Apple Inc. All Rights Reserved.

Important: 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。

実装の*末尾*で、スーパークラスの実装を呼び出す必要があります。

システムリソースの管理を、オブジェクトの寿命と結びつけて考えてはなりません。"希少なリソースはdeallocしない" (18 ページ)を参照してください。

アプリケーションが終了する際、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。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、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。

Core Foundationの管理規則は似ているが、いくつか違いもある

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の作成規則(「The Create Rule」 in *Memory Management Programming Guide for Core Foundation* を参照)は、Objective-Cのオブジェクトを返すメソッドには適用されません。★注：後編集で修正：『*Memory Management Programming Guide for Core Foundation*』の「The Create Rule」★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];
```

© 2012 Apple Inc. All Rights Reserved.

実践的なメモリ管理

“メモリ管理の方針”（7 ページ）で説明した基本方針は単純ですが、メモリ管理を容易にし、プログラムの信頼性や頑健性を維持しつつ必要なリソースを最小限に抑えるためには、具体的な手順を踏んで開発を進める必要があります。

アクセサメソッドを使ってメモリを管理しやすくする

あるクラスのプロパティがオブジェクトである場合、値として設定されたオブジェクトが、使用中なのに割り当て解除されてしまうことは避けなければなりません。したがって、値として設定する際に、そのオブジェクトの所有権を要求する必要があります。また、（不要になったときには）その所有権を確実に放棄することも必要です。

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
@property (nonatomic, retain) NSNumber *count;
@end;
```

@propertyと記述すれば、2つのアクセサメソッドを宣言したことになります。通常はこの方法で、メソッドを生成するようコンパイラに指示すればよいのですが、実際にどのように実装されるのか、知っておくと役立つでしょう。

「get」アクセサは同期したインスタンス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;
}
```

© 2012 Apple Inc. All Rights Reserved.

「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、_countがまだ設定されていなくてもこの実装は動作します）。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;
}
```

アクセサメソッドでプロパティ値を設定する

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。第1の実装は、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を対応させせます。

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

第2の実装では、コンビニエンスコンストラクタでNSNumberオブジェクトを生成します。したがってretainメッセージやreleaseメッセージは必要ありません。

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

どちらの場合も、setアクセサメソッドを使用する点に注目してください。

© 2012 Apple Inc. All Rights Reserved.

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（retainやreleaseの書き忘れ、インスタンス変数変更時のメモリ管理方法など）。

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

キー値監視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。

初期化メソッドやdeallocではアクセサメソッドを使わない

初期化メソッドと`dealloc`に限り、アクセサメソッドを使ってインスタンス変数の値を設定してはなりません。0を表す数値オブジェクトでカウンタオブジェクトを初期化したい場合は、次のように`init`メソッドを実装する必要があります。

```
- init {
    self = [super init];
    if (self) {
        _count = [[NSNumber alloc] initWithInteger:0];
    }
    return self;
}
```

0以外の値にも初期化できるようにしたい場合は、次のように`initWithCount:`メソッドを実装してください。

```
- initWithCount:(NSNumber *)startingCount {
    self = [super init];
    if (self) {
        _count = [startingCount copy];
    }
    return self;
```

© 2012 Apple Inc. All Rights Reserved.

```
}
```

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];
}
```

保持の循環を回避するために弱い参照を使う

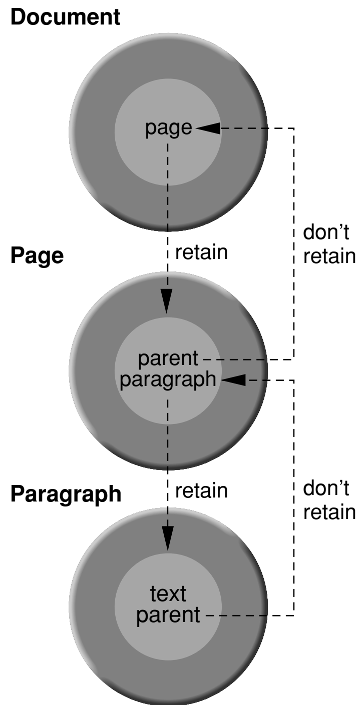
オブジェクトを保持すると、そのオブジェクトへの**強い**参照が作成されます。オブジェクトへの強い参照がすべて解放されるまで、そのオブジェクトは割り当て解除できません。2つのオブジェクトが循環参照している場合、**保持の循環**と呼ばれる問題が起こりえます。強い参照が相互に（直接、あるいは強い参照が連鎖して間接的に）結ばれている状況です

図1（16ページ）に示すようなオブジェクトの関係があると、保持の循環が起こることがあります。Documentオブジェクトは、ドキュメント内のページごとにPageオブジェクトを1つ保持します。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プロパティを1つ持っています。DocumentオブジェクトがPageオブジェクトに対する強い参照を持ち、Pageオブジェクトが

© 2012 Apple Inc. All Rights Reserved.

Documentオブジェクトに対する強い参照を持っていると、どちらのオブジェクトも割り当て解除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　　循環参照の例



保持の循環が生じたとき、これを解消するためには、弱い参照を使います。**弱い参照**とは、所有を伴わない関係、すなわち、参照元オブジェクトが参照先オブジェクトを保持しない参照関係のことです。

しかし、オブジェクトグラフを正しく維持するためには、どこかに強い参照がなければなりません（弱い参照しかないと、ページや段落は誰の所有でもないので、割り当て解除されてしまいます）。そこでCocoaでは、「親」オブジェクトは「子」オブジェクトに対して強い参照を持ち、子は親に対して弱い参照を持つ、という規約を設けています。

たとえば図 1 （16 ページ）の場合、DocumentオブジェクトはPageオブジェクトを強く参照（保持）し、一方PageオブジェクトはDocumentオブジェクトを弱く参照、すなわち保持しないことになります。

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。

© 2012 Apple Inc. All Rights Reserved.

弱い参照のみを持つ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。

使用中のオブジェクトの割り当て解除を避ける

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。

これには大きく分けて2つの例外があります。

1. オブジェクトがいずれかの基本コレクションクラスから削除される場合。

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

   オブジェクトが基本コレクションクラスから削除されると、`release`（`autorelease`ではない）メッセージが送信されます。削除されたオブジェクトの所有者がコレクションだけであれば、そのオブジェクト（この例では`heisenObject`）は即座に割り当て解除されます。

2. 「親オブジェクト」が割り当て解除される場合。

```
id parent = <#create a parent object#>;
// ...
heisenObject = [parent child] ;
[parent release]; // または、たとえば:self.parent = nil;
```

© 2012 Apple Inc. All Rights Reserved.

```
// heisenObjectは無効になっている。
```

オブジェクトを他のオブジェクトから検索し、直接または間接に親オブジェクトを解放する、という状況になる場合があります。親を解放すると割り当て解除される状況で、この親が子の唯一の所有者であった場合は、子（この例ではheisenObject）も同時に割り当て解除されます（親のdeallocメソッドで、autoreleaseではなくreleaseメッセージが送られる場合を想定）。

このような状況を防ぐため、heisenObjectを受け取ったらすぐに保持し、使い終わったら解放してください。例を示します。

```
heisenObject = [[array objectAtIndex:n] retain];
[array removeObjectAtIndex:n];
// heisenObjectを使用...
[heisenObject release];
```

希少なリソースはdeallocしない

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問題が発生します。例を示します。

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。たとえば、オブジェクトが通常の方法で解放されると想定していたのに、実際には自動解放の対象になった場合、破棄の順番が変わり、予期しない結果が生じる恐れがあります。

2. 希少なリソースを回収できない。

© 2012 Apple Inc. All Rights Reserved.

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されると、それが発生したスレッドの自動解放プールブロック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。

コレクションは自分に収容されるオブジェクトを所有する

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

© 2012 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。

所有権に関する処理は保持カウントを使って実装されている

所有権に関する処理は、参照カウントを使って実装されています。通常、`retain`メソッドの呼び出し後は、これを「保持カウント」と呼びます。各オブジェクトに保持カウントがあります。

- 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。

  オブジェクトに`autorelease`メッセージを送信すると、現在の自動解放プールブロックの末尾で、保持カウントが1つ減ります。
- オブジェクトの保持カウントがゼロになると、割り当て解除されます。

> **Important:** 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。

自動解放プールブロックの利用

自動解放プールブロックは、オブジェクトの所有権は放棄するけれども、即座に割り当て解除されてしまうのも困る、という状況で役に立つ機構です（メソッドからオブジェクトを返す場合など）。通常は独自に自動解放プールブロックを作成しなくても構いませんが、それが必須である場合や、作成することにより利点が生じる場合もあります。

自動解放プールブロックについて

自動解放プールブロックは、次の例のように、`@autoreleasepool`というキーワードで表します。

```
@autoreleasepool {
    // 自動解放オブジェクトを生成するコード。
}
```

ブロック内で`autorelease`メッセージを受け取ったオブジェクトは、自動解放プールブロックの末尾で、`release`メッセージを受け取ります。ブロック内で`autorelease`メッセージが何度か送られた場合、その回数に応じて`release`メッセージを受け取ることになります。

他のコードブロックと同様、自動解放プールブロックも「入れ子」にすることができます。

```
@autoreleasepool {
    // . . .
    @autoreleasepool {
        // . . .
    }
    . . .
}
```

© 2012 Apple Inc. All Rights Reserved.

（通常、上とまったく同じようなコードを目にすることはないでしょう。多くの場合、あるソースファイルの自動解放プールブロック内で、他のソースファイルに記述されたコードを呼び出しており、その中に自動解放プールブロックがある、という形で入れ子が現れます。）このような場合、`release`メッセージが送られるのは、`autorelease`メッセージが送られた位置に対応する自動解放プールブロックの末尾です。

Cocoaは自動解放プールブロック内のコードが必ず実行されると想定します。そうでなければ、自動解放されたオブジェクトが実際に解放されることはないので、メモリリークが発生します（自動解放プールブロック外で`autorelease`メッセージが送信された場合、Cocoaは適切なエラーメッセージを記録します）。AppKitやUIKitなどのフレームワークは、イベントループで反復される各回の処理（マウスクリック、タップなど）を、自動解放プールブロック内で実行するようになっています。したがって通常、明示的に自動解放プールブロックを作成する必要はなく、そのためのコードを目にすることもありません。しかし次の3つの場合には、独自に自動解放プールブロックを作成するとよいでしょう。

- UIフレームワークを基盤としないプログラム（コマンドラインツールなど）を開発する場合。
- ループ内で大量の一時オブジェクトを作成する場合。

  ループ内で自動解放プールブロックを使って、次の反復が始まる前に一時オブジェクトを破棄します。このような形で自動解放プールブロックを使うと、アプリケーションの最大メモリ占有量を減らすことができます。
- 別のスレッドを生成する場合。

  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ブロック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“自動解放プールブロックとスレッド”（24 ページ）を参照）。

ローカルな自動解放プールブロックを使ってピーク時のメモリ消費量を減らす

多くのプログラムは、自動解放の対象となる一時オブジェクトを次々に作成します。したがって、ブロック末尾までの間、プログラムのメモリ消費量は増え続けることになります。多くの場合、イベントループの末尾に到るまで一時オブジェクトが蓄積していったとしても、大きなオーバーヘッドにはなりません。しかし状況によっては、一時オブジェクトが大量に生成され、メモリ消費量が著しく増えるため、より迅速に破棄しなければならないこともあります。この場合は独自に自動解放プールブロックを作成してください。ブロックの末尾で一時オブジェクトも解放され、通常は割り当て解除もされるので、メモリ消費量は減ります。

`for`ループ内でローカルな自動解放プールブロックを使う例を以下に示します。

© 2012 Apple Inc. All Rights Reserved.

```
NSArray *urls = <# An array of file URLs #>;
for (NSURL *url in urls) {

    @autoreleasepool {
        NSError *error;
        NSString *fileContents = [NSString stringWithContentsOfURL:url
                                         encoding:NSUTF8StringEncoding error:&error];
        /* 文字列を処理し、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 */
    }
}
```

このforループは一度に1つのファイルを処理します。自動解放プールブロック内でautoreleaseメッセージが送られたオブジェクト（たとえばfileContents）が、ブロック末尾で解放されます。

自動解放プールブロックの後では、自動解放されたオブジェクトは「破棄」されたものと考えなければなりません。当該オブジェクトにメッセージを送信したり、メソッドの呼び出し元に返したりすることはできません。自動解放プールブロック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、次の例のように、そのブロック内でオブジェクトにretainメッセージを送信します。そして、ブロックの後でそのオブジェクトにautoreleaseを送信します。

```
- (id)findMatchingObject:(id)anObject {

    id match;
    while (match == nil) {
        @autoreleasepool {

            /* 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 */
            }
        }
    }

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

2012-07-17 | © 2012 Apple Inc. All Rights Reserved.

```
}
```

自動解放プールブロック内で`match`に`retain`を送信し、自動解放プールブロック後に`autorelease`を送信する、という方法により、`match`の寿命を延ばし、ループ外でメッセージを受け取ったり、`findMatchingObject:`の呼び出し元に返したりすることができます。

自動解放プールブロックとスレッド

Cocoaアプリケーションでは、スレッドごとに自動解放プールブロックのスタックを管理しています。Foundation上でのみ動作するプログラムを開発している、あるいはスレッドをデタッチする場合は、独自の自動解放プールブロックを作成する必要があります。

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、（AppKitやUIKitのメインスレッドで行っているのと同様に）自動解放プールブロックを使用するべきです。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、自動解放プールブロックを使用する必要はありません。

> **注意:** `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。

© 2012 Apple Inc. All Rights Reserved.

書類の改訂履歴

この表は「*高度な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
2012-07-17	@autoreleasepoolブロックの概念を取り入れて、自動解放に関する記述を更新しました。
2011-09-28	ARCの導入に伴う変更を反映するように更新しました。
2011-03-24	わかりやすく、簡潔にするために大幅な改訂を行いました。
2010-12-21	可変コピーの命名規則を明確にしました。
2010-06-24	メモリ管理の原則について表現上の細かな変更を行い、簡潔さを強調しました。実践的なメモリ管理について細かな追加を行いました。
2010-02-24	iOS 3.0向けに、メモリ警告への対処についての説明を更新しました。「オブジェクトの所有権と破棄」の一部を改訂しました。
2009-10-21	実践的なメモリ管理にアクセサメソッドのセクションを追加しました。
2009-08-18	関連する概念へのリンクを追加しました。
2009-07-23	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	誤字を訂正しました。

© 2012 Apple Inc. All Rights Reserved.

日付	メモ
2009-03-04	誤字を訂正しました。
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	誤字を訂正しました。
2007-10-31	OS X v10.5向けに更新しました。細かな誤字を訂正しました。
2007-06-06	細かな誤字を訂正しました。
2007-05-03	誤字を訂正しました。
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	このドキュメント内の章を再編成し、わかりやすくしました。「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についての説明を明確にしました。アクセサメソッドの説明を別の章に移動しました。

© 2012 Apple Inc. All Rights Reserved.

日付	メモ
2006-01-10	誤字を訂正しました。ドキュメント名を『メモリ管理』から変更しました。
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	"自動解放プールブロックの利用"（21 ページ）の自動解放プールが解放されたときに解放されるものの説明に、明示的および暗黙的自動解放オブジェクトの説明を含めました。
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。

2012-07-17 | © 2012 Apple Inc. All Rights Reserved.

Apple Inc.
© 2012 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.

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

Apple, the Apple logo, Carbon, Cocoa, Instruments, Objective-C, OS X, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

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。