
メモリ管理プログラミングガイド

[Cocoa](#) > [Objective-C Language](#)



2011-09-28



Apple Inc.
© 2011 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, Instruments, Mac, Mac OS, Objective-C, and Xcode are trademarks of Apple Inc., registered in the United States 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。

目次

メモリ管理について 7

At a Glance 7

- 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 8
- 分析ツールを使ってメモリ管理上の問題を解決する 9

メモリ管理の方針 11

基本的なメモリ管理方法 11

- 簡単な例 11
 - autoreleaseを使って、releaseメッセージを「遅らせて」送信する 12
 - 参照により返されたオブジェクトを所有しない 13
- オブジェクトの所有権を放棄するdeallocを実装する 13
- Core Foundationの管理規則は似ているが、いくつか違いもある 14

実践的なメモリ管理 15

- アクセサメソッドを使ってメモリを管理しやすくする 15
 - アクセサメソッドでプロパティ値を設定する 16
 - 初期化メソッドやdeallocではアクセサメソッドを使わない 16
- 保持の循環を回避するために弱い参照を使う 17
- 使用中のオブジェクトの割り当て解除を避ける 19
- 希少なリソースはdeallocしない 20
- 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 20
- 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 21

自動解放プールの利用 23

- 自動解放プールについて 23
- ローカルな自動解放プールを使ってピーク時のメモリ消費量を減らす 24
- 自動解放プールとスレッド 25
- 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 25
- ガベージコレクション 26

書類の改訂履歴 27



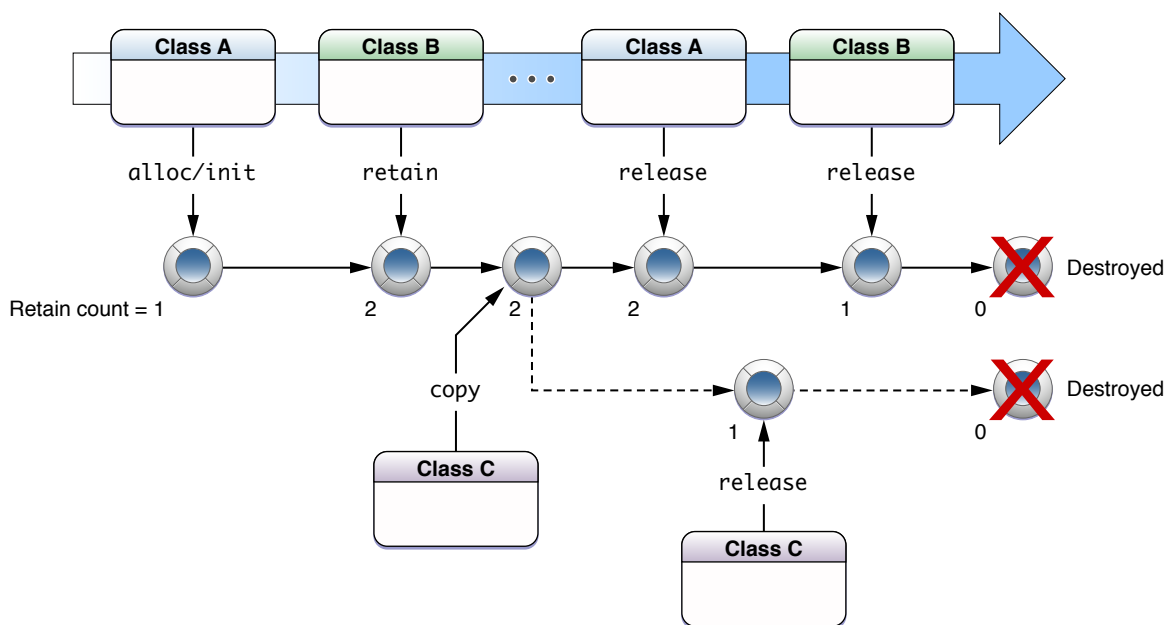
実践的なメモリ管理 15

図 1 循環参照の例 18

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。



At a Glance

Objective-Cにはアプリケーションメモリの管理手段が3つあります。

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いて実装されており、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。
3. **ガベージコレクション**の場合、どのオブジェクトがどのオブジェクトを所有しているか、システムが自動的に追跡します。その上で、どこからも参照されていないオブジェクトを、自動的に回収（ごみ集め）、解放するようになっています。MMRやARCとは異なる機構を用いており、Mac OS Xの実行時環境でしか利用できません（iOSでは不可）。ガベージコレクションについて詳しくは、『*Garbage Collection Programming Guide*』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](#)」（11 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。

「[実践的なメモリ管理](#)」（15 ページ）を参照してください。

自動解放プールは、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を用意して使うべき場合もあります。

「[自動解放プールの利用](#)」（23 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を見つけるためには、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。「[トレースデータの表示と分析](#)」を参照してください。

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）ほか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](#)」（19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。

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 = person.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 {
    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 = nil;
NSString *string = [[NSString alloc] initWithContentsOfFile:fileName
                                     encoding:NSUTF8StringEncoding error:&error];
if (string == nil) {
    // エラー処理...
}
// ...
[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
@synthesize firstName=_firstName, lastName=_lastName;
// ...
- (void)dealloc
{
    [_firstName release];
    [_lastName release];
    [super dealloc];
}
@end
```

重要： 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付けて考えるのはなりません。「[希少なリソースはdeallocしない](#)」（20 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。

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の命名規則は異なります。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。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 sharedInstance];
```

実践的なメモリ管理

「[メモリ管理の方針](#)」（11 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あります。

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 {
    NSNumber *_count;
}
@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;
}
```

「`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 numberWithInt: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。

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;
}
```

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


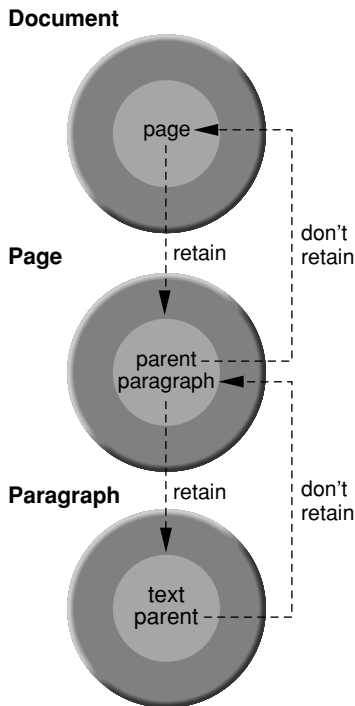
 **図 1**（18 ページ）に示すようなオブジェクトの関係があると、保持の循環が起こることがあります。Documentオブジェクトは、ドキュメント内のページごとにPageオブジェクトを1つ保持します。それぞれのPageオブジェクトは、それがどのドキュメントに属するかを追跡するプロパティを1つ持っています。DocumentオブジェクトがPageオブジェクトに対する強い参照を持ち、PageオブジェクトがDocumentオブジェクトに対する強い参照を持っていると、どちらのオブジェクトも割り当て解除されることはありません。Documentの参照カウントはPageオブジェクトが解放されるまで0にならず、Page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、ページや段落は誰の所有でもない、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（18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となります。

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。

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;
// heisenObjectは無効になっている。
```

オブジェクトをほか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。次に例を示します。

```
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が期待通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う問題が発生します。次に例を示します。

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。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。

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 numberWithInt: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 *allocatedNumber = [[NSNumber alloc] initWithInteger: i];
    [array addObject:allocatedNumber];
    [allocatedNumber release];
}
```

この場合は、`for`ループの範囲内で`alloc`と対応がとれるように、`allocatedNumber`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、将来のある時点で、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。

重要： 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分からないために、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。

自動解放プールの利用

自動解放プールは、`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）。通常は独自に自動解放プール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があります。

自動解放プールについて

自動解放プールは`NSAutoreleasePool`のインスタンスで、`autorelease`メッセージを受け取ったほかのオブジェクトを「保持」しています。自動解放プールを割り当て解除すると、保持している各オブジェクトに`release`メッセージが送られます。同じオブジェクトを重ねて自動解放プールに入れることもできます。オブジェクトは、プールに入れられた（`autorelease`メッセージが送信された）回数だけ、`release`メッセージを受信することになります。

自動解放プールはスタックに積み上げられますが、これが複数ある場合、「ネストして（入れ子になって）いる」と言うのが普通です。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。

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、アプリケーションに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。AppKitやUIKitなどのフレームワークでは、イベントループ（マウスクリック、タップなど）ごとに、先頭でプールを自動生成し、最後に解放するようになっています。したがって通常、明示的にプール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。

自動解放プールを別に用意して管理し、ループの末尾に到る前でも、適当な時点で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](#)」（25 ページ）を参照）。

NSAutoreleasePoolオブジェクトは通常通り、allocメッセージ、initメッセージで生成し、drainメッセージで破棄します。自動解放プールにautoreleaseまたはretainメッセージを送ると、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](#)」(26 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、ループ本体など)で破棄しなければなりません。

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。

ローカルな自動解放プール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、プールが破棄されるまでの間、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。ピーク時のメモリ消費量を減らすには、ローカルな自動解放プールを導入するとよいでしょう。プールを破棄すると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を使う例を以下に示します。

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

    NSAutoreleasePool *loopPool = [[NSAutoreleasePool alloc] init];

    NSError *error = nil;
    NSString *fileContents = [[[NSString alloc] initWithContentsOfURL:url
                                                                    encoding:NSUTF8StringEncoding
                                                                    error:&error] autorelease];

    /* 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 */

    [loopPool drain];
}
```

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、最後で破棄されます。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が破棄されるときに、これらのオブジェクトが解放されます。

自動解放プールが割り当て解除された時点で、それまで「自動解放」の対象であったオブジェクトは「破棄」されたものと考えて、以降の処理を実装する必要があります。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、次の例のように、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、自動解放プールが破棄された後にそのオブジェクトにautoreleaseを送信します。

```
- (id)findMatchingObject:(id)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 */
}
[subPool drain];
}

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が破棄されてからそれに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。

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。Foundation上でのみ動作するアプリケーションを開発している、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、(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

一般に、自動解放プールがネストしている、という言い回しをしますが、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。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自動解放プール用スタックの一番上にプッシュされます。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: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破棄されるまで現在最上位にあるプールはスコープから外れます（新しいプールが破棄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破棄された場合は、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破棄された場合は、スタック上でそれより上にあるすべての（未解放の）自動解放プールが破棄され、そのオブジェクトに適切なreleaseメッセージが送信されます。自動解放プールを使い終わったときに、それにdrain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つが破棄されるときにそのプールも破棄されます。

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破棄されます。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、破棄されたプールより上のすべてのプールも破棄されます（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、すでに破棄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。

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*』を参照）は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えるフレームワーク）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。また、通常、自動解放プールが解放される場所は、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。

書類の改訂履歴

この表は「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
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	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	誤字を訂正しました。
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	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。

日付	メモ
2006-05-23	この文書内の章を再編成し、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 <code>dealloc</code> 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。
2006-01-10	誤字を訂正しました。文書名を『Memory Management』から更新しました。
31.08.04	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。
06.06.03	「 自動解放プールの利用 」（23 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。
12.11.02	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。