

第 3 章

COMオブジェクトと COMインターフェイスの構築

概要

クライアントとサーバーの役割と相違点

一意な識別子(GUID、CLSID、REFIID)

インターフェイスクラスの構造とインターフェイスの構築

IUnknown インターフェイスの役割

レジストリの設定

COM のクライアントとインプロセスサーバーの例

今日のコンピュータ業界は狂騒状態にあり、特にインターネットに関しては誰もがすっかり夢中になっている。インターネット技術を顧客に提供することによって生み出される利益を見越して、インターネット上に新しい企業が相次いで登場した。すぐに話題になった企業も多く、インターネットビジネスで億万長者になった事業家もいる。この大ブームにより、新しい技術が次々と現れ、インターネットとインターネット技術に関する新しい略語が毎日のように生まれている。

コンピュータ技術においてめまぐるしい速度で変化するインターネット分野で、遅れずについていくのは控えめに言ってもかなりたいへんである。第1章「Microsoftのオブジェクト技術の概要」で触れたように、Microsoftは、過熱するインターネット市場の参入については競合他社より若干遅れたが、インターネット技術の革新に大きく貢献し、リーダーシップをとってきた。ここ数年にMicrosoftが世に送り出した膨大な技術、用語、略語に圧倒されているとしても、心配することはない。本書を読み進めていくとわかるが、幸いにもMicrosoftの新技术のほとんどはCOM(およびDCOM)を基盤としている。

第1章と第2章では、COMの定義とMicrosoftの技術戦略においてCOMが重要な役割を担っている背景を説明した。第3章以降では、これまでの説明に基づいて、実際にCOMコンポーネントを構築していく。段階を踏んでわかりやすく説明するので、心配せずについてきてほしい。

3.1 COMプログラミングを基礎から学ぶメリット

COM/DCOMの技術は複雑であるため、最初の段階では少し辛抱強く学習しなければならない。Microsoftは、OLEが不発に終わったことからこの事実を認識し、最大限の努力を払ってCOMのサポートをMFC(Microsoft Foundation Class)に組み込んだ。現在に至るまで、MFCは、C++プログラムでCOMをサポートするもっとも簡単な方法の一つであることは変わらない。このため、本書では、MFCライブラリを使うCOMアプリケーションの開発を説明することが多い。ただし、第3章ではまずCOMプログラミングの基本を学ぶ。

多くの面で、MFCはCOMアーキテクチャの機能をうまく隠してくれるため、ベースにはCOMの基本的な概念があることを認識せずに忘れてしまう。数学でも同じようなことがある。数学のあるコースで二次方程式の使い方を教わったとき、教授が授業にプログラム式の電卓の使用を許可したため、二次方程式やそのほかの数式をプログラムして電卓に入力した。この学期が終わって別の教授のコースを履修したが、ご想像のとおり、こちらの教授は電卓の使用を許さなかった。すっかり電卓に頼るようになっていた著者は、試行錯誤の結果、数日をかけて二次方程式やほかの数式の原理を学習し直すことになった。結局、自分で数式を解く方法を再度勉強しなければならなかったのである。電卓は、著者にとって、具体的な計算の理論や手順を抽象化して隠す階層を提供していたと言える。

電卓と同じように、MFCはプログラマに、COMの詳細を隠す階層の役割を果たす。MFCは必要な作業の大部分をプログラマの代わりにを行い、COMコンポーネントの実装の詳細を隠す。しかし、COMの基本を十分に学ばなければMFCが提供するこの抽象化

層に頼りすぎることになり、MFC ではプログラミングできないコンポーネントを注文する顧客が現れたようなときに COM の基本を大急ぎで学び直さなければならない羽目に陥る。このような事態を招かないように、本章で COM のプログラミングを自力で行う方法を学び、第 5 章「MFC による COM のプログラミング」で MFC を使う COM プログラミングについて説明する。MFC を使うと、第 3 章では自分で行う作業の大部分が MFC によって自動的に行われる。しかし、COM ベースのアプリケーションの内部処理を十分に理解するためには、最初に基本から学ぶことが大切だ。

注

COM の習得は非常にたいへんだ。COM は確かに複雑であるが、コンポーネント開発において COM により解決可能なさまざまな問題について考えると、それも納得できる。COM を習得するうえで重要なのは、くじけないことだ。COM のプログラミングでは、一般的なプログラミングにおいて馴染みの少ない考え方が必要になる。一度で習得できる人など誰もいない。十分に基礎を固め、実習を繰り返し、COM の世界に合わせて考え方を変える必要がある。COM の習得という目標を忘れずに、辛抱強く学習を進めてほしい。本書は、この目標を達成するためのガイドである。

3.2 クライアントとサーバー

コンピュータを使っていないときには、そのコンピュータは使われていない状態であるだけだ。これはまったく当然のように思えるが、論旨は完全に正しい。もう少しわかりやすく言えば、コンピュータ(または技術や開発ツールなど)に投資しても、使わなければ意味がない。

COM ではこのわかりきった考え方を強化して、オブジェクトモデルをクライアントとサーバーという 2 つの部分に概念的に分けている。図 3-1 では、この概念をもう少しわかりやすく表している。

注

クライアント/サーバーコンピューティングというすでにお馴染みの用語は、一般的には、複数のコンピュータにアプリケーションを分散し、各コンピュータに特定の機能を割り当てることを意味する。たとえば、典型的な 3 層構造のクライアント/サーバーアプリケーションアーキテクチャを例として考えると、最初のコンピュータに GUI、2 番目のコンピュータにアプリケーションのロジック、3 番目のコンピュータにデータベースを配置する構造が可能である。Microsoft は、業界標準のクライアント/サーバー方式の 1 つを採用して、巧みに修正した。では、Microsoft がクライアント/サーバー方式を COM に適用した方法を説明しよう。

図 3-1 は非常に単純である。1 つ以上の COM コンポーネントを保持しているのがサーバーで、そのコンポーネントを使うプログラムまたはアプリケーションがクライアントである。クライアントおよびサーバーは、同じプロセス内に存在し、同じコンピュータ上で動作する。続いて、図 3-2 に、クライアントとサーバーがやり取りする別の例を示す。

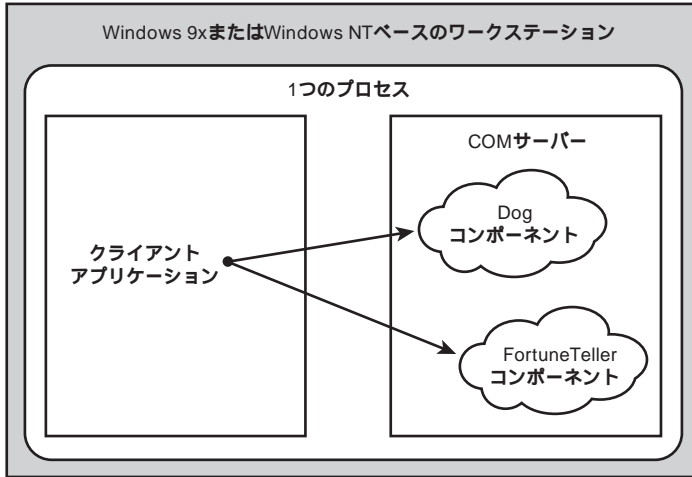


図 3-1 COMの基本的なクライアント/サーバーモデル

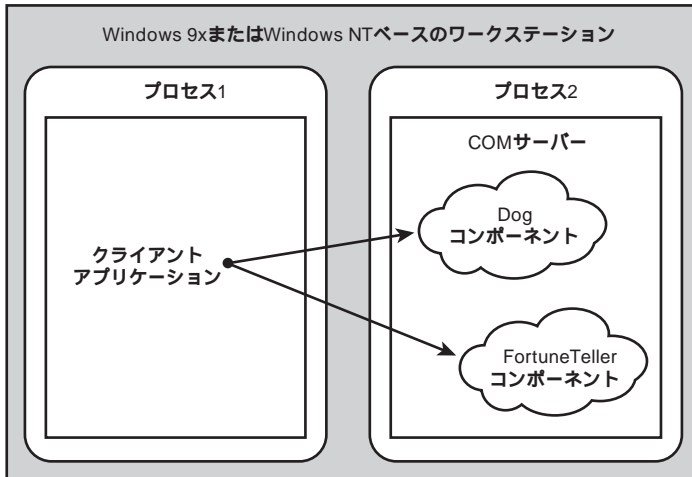


図 3-2 異なるプロセスに存在するCOMのクライアントとサーバー

図 3-2 は図 3-1 によく似ているが、クライアントとサーバーはそれぞれ別のプロセス内に置かれている。ただし、どちらも同じコンピュータ上で動作する。最後の例として、図 3-3 のクライアント/サーバーモデルを検討しよう。

図 3-3 では、クライアント/サーバーコンピューティングにおいてもっとも一般的な形式でクライアントとサーバーが分離されている。この場合、クライアントプログラムとCOMサーバーは別のコンピュータ上で実行され、ネットワークを介して相互に通信する。

標準的なCOMアーキテクチャをもっとも正確に表しているのは、どの図なのか。答えは、すべての図だ。より正確には、最初の2つの図に示すシステム構成はCOMで標準的に直接サポートされており、図 3-3 に示す構成はDCOMによってサポートされている。

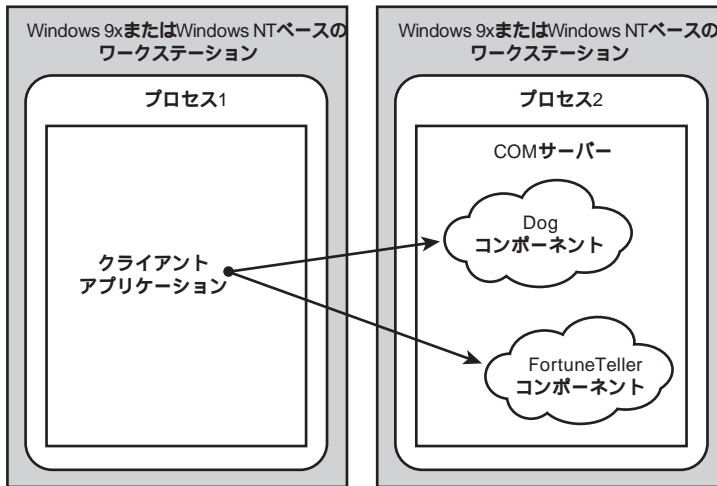


図 3-3 異なるコンピュータ上に存在する COM のクライアントとサーバー

クライアントは、COM コンポーネントが提供するサービスを必要とするプログラムである。つまり、COM クライアントは実行可能プログラムにすぎない。一方、COM サーバーはいくつかの異なる形態をとり、通常はどの形態をとるかによって、先ほどの3つの図に示したアーキテクチャのうち可能な構成が決まる。

3.2.1 インプロセスサーバー

図 3-1 では、クライアントとサーバーが同じプロセス内の同じアドレス空間に存在することがはっきりとわかる。COM では、クライアントと同じアドレス空間を共有するサーバーを、**インプロセスサーバー**と呼ぶ。インプロセスサーバーは、標準的な Windows の DLL としてまとめられる。COM は、必要に応じて DLL をロードし、クライアントがコンポーネントを使い終えたと判断したら DLL をアンロードする。コンポーネントを設計する場合、1つの DLL に複数のオブジェクトを入れたり、複数のコンポーネントをそれぞれの DLL に入れることができる。

3.2.2 ローカルサーバー

図 3-2 に示す構成を利用する際には、COM では DLL を使うことができない。クライアントのプロセスの外部から利用可能なコンポーネントは、**ローカルサーバー**に含まれているとみなされる。ローカルサーバーは、DLL ではなく完全な実行可能プログラム (EXE) として実装される。インプロセスサーバーと同様に、設計時には、1つのローカルサーバーに複数のオブジェクトを入れたり、各コンポーネントを1つのサーバープログラムに個別に入れることができる。

COMのランタイムライブラリ

少し話はそれるが、説明を進める前に注意しなければならないことがある。説明が遅れたが、COMにはランタイムライブラリがある。このライブラリはCOMで構築されている。COMのランタイムライブラリは、COMのサポートに必要なサービス、関数、インターフェイスを提供する。これは、COMコンポーネントを開発するたびに基本部分のコーディングから繰り返すことを避けるためだ。また、COMのランタイムライブラリは、複数のコンポーネントがプロセス内で対話するために必要なサービスもいくつか提供している。サービスの詳細については、サーバーの説明において言及する。

これで3種類のサーバーのうち2種類までは説明したが、COMではクライアントとサーバーの対話方法がまだよくわからない。COMでは、プロセス内または同じコンピュータ上のプロセス間の通信にLRPCを使う。

LRPC

COMは、同じコンピュータ上のプロセス間の通信に、LRPC(Lightweight Remote Procedure Call)という通信機構を利用する。LRPCは、後述するRPC(Remote Procedure Call)の小型版で、プロセス間通信を効率化するために設計された。LRPCはCOM固有の方式である。LRPCを使うことにより、COMでは完全に透過的なプロセス間通信が行われる。クライアントまたはオブジェクトが呼び出しを行うと、COMは呼び出しを捕捉し、LRPCを使って別のプロセス内の対象コンポーネント(またはサーバー)に渡す。第1章ではVisioの図面をWord文書に埋め込む例を紹介したが、埋め込みではWordとVisioなどのアプリケーション間の通信はこの方法で行われる。Wordの文書とVisioの図面は、どちらもCOMオブジェクトである。クライアント(Word)は、サーバー(Visio)が提供するCOMオブジェクトのインターフェイスポインタを取得する。COMのランタイムライブラリは、LRPCを使ってオブジェクトから別のオブジェクトへメッセージを転送する。

マーシャリング

クライアントからサーバーを呼び出しただけでは、必要な処理が半分しか終わっていない。この場合、呼び出したアプリケーションのアドレス空間にアクセスする必要がある。インターフェイスポインタを使う場合、ポインタが指す構造体を別のプロセスのアドレス空間にコピーして利用可能にしなければならない。この処理を**マーシャリング**という。

COMオブジェクトは、マーシャリングを実行するために、COMのランタイムライブラリにあるIMarshal **インターフェイス**を使う。IMarshalは、関数の呼び出しの前後で行われるパラメータのマーシャリングとアンマーシャリングに関連する処理を行う。具体的には、構造体をプロセス間でコピーし、必要な情報の参照を可能にする。1つのプロセス内のあるCOMコンポーネントから別のCOMコンポーネントのデータを参照するのは難しい処理であるが、COM、COMのランタイムライブラリ、LRPCがこの処理を見えないところで実行している。同じコンピュータ上にあるコンポーネント間の通信だけでも面倒な処理になるが、異なるコンピュータ上にあるコンポーネント間の通信はさらに難しい問題である。続いて、この問題とリモートサーバーの通信について説明する。

3.2.3 リモートサーバー

図 3-3 に示す構成では、クライアントからの要求をほかのコンピュータ上で稼働しているサーバーに転送するために、DCOM の技術を使う。クライアントとは異なるコンピュータ上に位置するサーバーを**リモートサーバー**と呼ぶ。DCOM のリモートサーバーには、DLL または実行可能プログラムのいずれかを設定可能だ。インプロセスサーバーやローカルサーバーと同様に、リモートサーバーにも、複数のオブジェクトのインスタンスを同時に置くことができる。

COM の優れた利点の 1 つは、クライアントは通信先のサーバーの種類(インプロセスサーバーまたはローカルサーバー)を意識する必要がないことだ。COM は、クライアントを目的のコンポーネントに接続するのに必要なオブジェクトと API の標準セットを提供している。API によって、インプロセスサーバーとローカルサーバーの実装に違いがあっても、それが隠される。インプロセスサーバーとローカルサーバーは実装が大きく異なる場合もあるが、1 つのクライアントを一貫した形式で作成するだけでよい。ここまでの説明により、COM においてローカルで通信が透過的に行われるしくみは理解できたはずだ。では、リモートの場合には透過的な通信がどのように行われるのだろうか。まず、プロキシという概念を理解する必要がある。

3.2.4 プロキシ

C++ で COM 以外の単純なプログラムを作成する場合、オブジェクトとの対話は、オブジェクトがコンパイラによってプログラムに組み込まれるものであれば簡単である。この場合、プログラムから単にオブジェクトのメソッドを呼び出すだけで、コンパイラが呼び出したメソッドのアドレスをプロセス内でマッピングする。オブジェクトが DLL 内に含まれている場合も処理手順は同じである。これは、オペレーティングシステムのプログラムローダーにより可能になる。プログラムローダーは、プログラムの起動時に必要な DLL がすべてメモリにロードされること、DLL の内部にあるメソッドの正しいアドレスがプログラム内の呼び出し点に結び付けられることを保証している。

この方式は COM のインプロセスサーバーにも応用可能である。COM のインプロセスサーバーは 1 つの DLL に含まれているため、COM のサブシステムはサーバーの DLL をロードし、前述と同様に結び付ける処理を実行する。しかし、ローカルまたはリモートの COM サーバーが異なるアドレス空間で動作している場合、クライアントが目的の COM コンポーネントをシームレスに呼び出す方法は多少わかりにくい。この場合、実際には、クライアントをローカルサーバーに直接結び付けることはできないため、プロキシを利用する。

COM のもっとも重要な役割の 1 つに、クライアントが特定のコンポーネントを要求した際に、必要なインターフェイスをサポートするオブジェクトをそのクライアントに確実に渡す処理がある。COM からクライアントに渡されるオブジェクト自体が、クライアントの要求したコンポーネントになる場合がもっとも単純である。COM サーバーがインプ

プロセスサーバーである場合がまさにこれだ。しかし、COM サーバーがローカルサーバーである(したがって別のアドレス空間に存在する)場合、COM はクライアントに**プロキシ**と呼ばれる特殊なオブジェクトを渡す。プロキシは、クライアントと同じアドレス空間に存在し、対応するコンポーネントがサポートしているすべてのインターフェイスに応答するオブジェクトである。クライアントからは、プロキシが目的のコンポーネントに見える。

対応するコンポーネントのインターフェイスのメソッドがクライアントから呼び出されると、プロキシは、メソッドのパラメータをすべて移動可能なデータ構造体のセットにまとめ、RPC を介して構造体のセットをローカルサーバーのプログラムに送る。

RPC

RPC は、OSF(Open Software Foundation)の DCE RPC 仕様で定められたプロセス間通信機構である。RPC を使うことにより、異なるコンピュータ上のアプリケーション間でさまざまなネットワーク通信機構を介した通信が可能になる。RPC は、クライアントとサーバーの間で通信を確立するためにほかの IPC(Interprocess Communication)機構を使う点が特徴的である。RPC では、リモートシステムとの通信に名前付きパイプ、NetBIOS、Windows ソケットを使うことができる。DCOM は、RPC を使うことにより、異なるコンピュータ上にあるプロセス内の COM コンポーネント間で通信を可能にしている。RPC によりアドレス空間をまたがる通信が可能であるため、プロキシは比較的容易に目的のローカルサーバーのアドレス空間にメソッドのパラメータを転送することができる。

サーバーでは、RPC スタブと呼ばれる小さなプログラムが RPC を受け取る。スタブは、受け取ったデータ構造体を元のメソッドのパラメータに分解し、該当するメソッドを実際のコンポーネントに対して呼び出す。メソッドの呼び出しが完了すると、コンポーネントはスタブに制御を戻す(戻り値がある場合はそのパラメータも返す)。スタブは、戻されたパラメータを移動可能な形式にまとめ、RPC を介してプロキシに返す。プロキシは、受け取ったデータを構造体から元の状態に分解し、クライアントに制御を戻す。クライアント、プロキシ、RPC スタブ、ローカルサーバーの関係を、**図 3-4**に示す。

この方式の優れた点は、**位置の透過性**が維持されることだ。つまり、クライアントは、COM コンポーネントとそのサーバーの位置を意識する必要がない。このように、DCOM では、クライアントプログラムは別のコンピュータ上で動作しているリモートサーバーを呼び出すことができるが、これを可能にしているのは何だろうか。そう、答えはプロキシオブジェクトと RPC である。位置の透過性は優れた効果を生み出す。

図 3-4と**図 3-1**を比べると、ローカルサーバーを使う場合は全体的にかなり複雑になることがわかる。もちろん、別のコンピュータ上にあるリモートサーバーを使う場合には、解決しなければならない技術的な問題がさらに増える。これらの理由により、ローカルサーバーおよびリモートサーバーをまったく新規に構築するのはかなり難しい。本章ではインプロセスの COM サーバーの構築方法だけを説明し、MFC を使ってローカルサーバーを構築する方法は第4章以降に譲る。DCOM のリモートサーバーを MFC を使って構築する方法については、本書の後半で説明する。

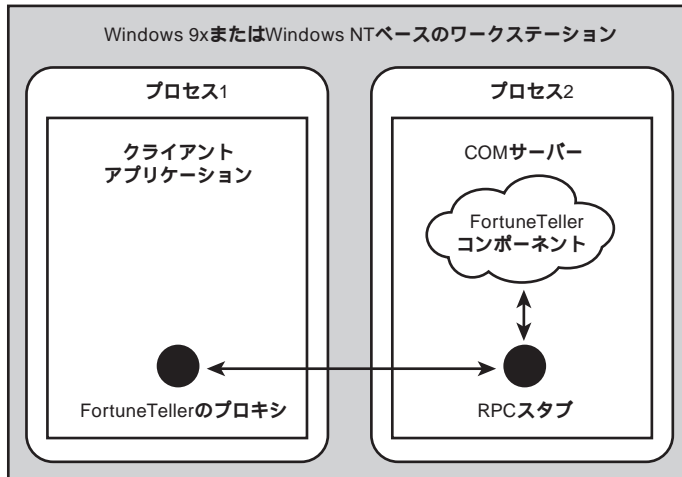


図 3-4 COM のローカルサーバーはプロキシと RPC スタブを使う

注

ここからは、クライアントとサーバーが同じコンピュータ上に存在する場合(特にインプロセスサーバーの場合)の問題を中心に説明する。リモートサーバーの構築方法の詳細については、第9章「分散オブジェクトの概要」を参照。第9章では、DCOMに関連する具体的な問題を詳しく説明する。

3.2.5 サーバーの役割

コンポーネントを置くサーバーの種類(インプロセスサーバーまたはローカルサーバー)に関係なく、COM仕様に準拠するためにサーバーがサポートしなければならない機能がいくつかある。機能の実装はサーバーの種類によって異なるが、機能的な要件は同じである。

ここでは、COMのインプロセスサーバーとローカルサーバーが扱う処理の一部について概要を説明する。ただし、あくまで概要の説明にとどめ、詳細は後述する。はっきりしない点があれば、後でサンプルプログラムを使って具体的な説明をするときに解決してほしい。

サーバーの登録

クライアントが特定のコンポーネントの使用をCOMに要求したとき、COMは要求先のサーバーの位置を認識している必要がある。この意味では、COMは、クライアントにぴったりのサーバーを探すお見合いゲームのようだ。COMをコンピュータ世界のChuck Woolery^{*1}にたとえるのは突飛だが、COMの処理はお見合いゲームよりは偶然に頼る割合がずっと少ない。COMコンポーネントは、人間ほど気まぐれではないからだ。

*1 米国のテレビ番組「The Dating Game」の司会者。

クライアントとサーバーを照合するために、COMは主にWindowsのレジストリを利用する。念のために補足するが、レジストリは巨大な階層構造をとる永続データである。Windowsオペレーティングシステムでは、頻繁にレジストリを使ってシステムの構成を管理し、アプリケーションをスムーズに機能させるために必要な情報の大部分を格納する。Windows 9xおよびWindows NTでは、オペレーティングシステム自体だけでなく、そのコンピュータ上にあるアプリケーションやハードウェアのドライバなど、初期設定や構成に関するデータがすべてレジストリに格納される。

COMは、レジストリを使って、システムにインストールされているコンポーネントのサーバーを追跡する。レジストリには、COM関連の情報として、クライアントから利用可能なコンポーネント、サーバーのDLLまたは実行可能プログラムの正確な位置、アプリケーションを実行するときに必要な詳細情報などが格納される。

サーバーに関する情報はなんらかの方法でレジストリに格納する必要があるが、この作業は開発者が責任を持って行わなければならない。サーバーを実装するときには、レジストリにサーバーの情報を格納するのに必要な関数を必ず実装する。サーバーの情報をレジストリに格納する処理を**サーバーの登録**と呼ぶ。逆に、この情報をレジストリから削除する処理は**サーバーの登録解除**と呼ばれる。サーバーおよびサーバーを構成するコンポーネントが不要になったときに、サーバーをシステムから削除する際に登録解除を実行する。COMサーバーの登録解除は、一般に、アプリケーションのアンインストール処理の一部として行われる。

サーバーの登録には2つの方法がある。最初の方法では、Regedit(またはRegedt32)というユーティリティで読み込み可能なスクリプトを作成し、レジストリに値を手動で格納する。この場合、登録用のスクリプトをサーバーに添付し、ユーザーまたはインストールプログラムによって実行する。登録用のスクリプトは簡単に変更したり、再実行することができるため、コンポーネントの開発中にはこの方法が最適である。

また、COMサーバーは自己登録機能を備えている。ローカルサーバーは、コマンドラインに/RegServerという引数を指定すると自己登録を行い、/UnregServerを指定すると自己登録解除を行う。/RegServerが指定された場合、サーバーは必要なエントリをレジストリに格納しなければならない。オペレーティングシステムがサポートしているWin32のレジストリAPIの関数を使って、レジストリに対する格納処理を行うのがもっとも一般的だ。

インプロセスサーバーの場合でも、処理は複雑になるが、自己登録をサポートすることができる。自己登録機能を持つインプロセスサーバーは、DllRegisterServer()とDllUnregisterServer()という2つのメソッドを外部に公開する。DllRegisterServer()は、Win32のレジストリAPIを使って、必要なエントリをすべてシステムレジストリに格納する。一方、DllUnregisterServer()は、DllRegisterServer()で格納したエントリを削除する。サーバーを登録するアプリケーションは、まずWin32APIのLoadLibrary()を使ってサーバーをメモリにロードし、GetProcAddress()を使って該当する関数のアドレスを取得し、関数を直接呼び出さなければならない。この方法の最大の欠点は、レジストリ登録用のスクリプトやローカルサーバーの自己登録を利用する場合と異なり、ユーザー

がコマンドラインから直接サーバーを登録できないことだ。インプロセスサーバーの自己登録では、サーバーの DLL をロードして登録関数を手動で呼び出すために、別のプログラムが必要になる。ただし、ありがたいことに、Windows NT および Windows 9x には Regsvr32 というユーティリティが付属しているため、サーバーの登録に利用することができる。このプログラムは、次のようにコマンドラインから実行する。

```
regsvr32 c:\myproject\bin\myserver.dll
```

クラスファクトリ

クラスファクトリについては、第 2 章「オブジェクト革命」で説明した。COM では、コンポーネントの生成と破棄にクラスファクトリを使う。したがって、サーバーは自分が扱うコンポーネントごとにクラスファクトリを 1 つずつ提供しなければならない。これは、どのサーバーも少なくとも 2 つの COM コンポーネント、つまり、処理を行うコンポーネントとそのコンポーネントのクラスファクトリをサポートすることを意味する。

クラスファクトリは、COM オブジェクトのライフサイクルの起点となるため、当然クラスファクトリオブジェクトの生成手順はほかの COM オブジェクトと異なる。では、すべてのクラスファクトリがクラスファクトリによって生成されるのであれば、最初のクラスファクトリはどのように生成されるのだろうか。この問題を考えると、ニワトリと卵の関係のように、最初のクラスファクトリを探す堂々巡りに陥る。

COM では、この問題を解決するために、サーバーに必ずブートストラップ機能を実装させてクラスファクトリオブジェクトを COM のサブシステムに送る。インプロセスサーバーとローカルサーバーでは機能の実装が異なることについてはすでに説明したが、この機能の実装がその一例である。

インプロセスサーバーをまったく新規に構築する場合、COM の要求に応じてクラスファクトリを COM に渡すだけで済む。クライアントが任意のコンポーネントとやり取りを開始すると、COM はレジストリを参照して正しいサーバーの位置を調べる。たとえば、あるプログラムが COM に対して新しい Dog オブジェクトを要求した場合、COM はレジストリを調べて Dog サーバーが C:\COM Servers\Dog.dll というインプロセスサーバーであることを認識する。COM は、サーバーがインプロセスサーバーであることを認識しているため、サーバーの DLL をロードした後、すべてのインプロセスサーバーに必ず実装されている DllGetObject() というメソッドを呼び出す。COM は、このメソッドの呼び出し時に、パラメータとして検索対象のクラスファクトリ名を渡す。この時点では、サーバーはクラスファクトリオブジェクトを生成し、COM のサブシステムに戻すだけでよい。実に単純である。

ローカルサーバーの場合は多少処理が異なる。COM が実行可能プログラムをロードして要求を出すだけでは済まないため、ローカルサーバーは起動時にすべてのクラスファクトリを自分で COM に登録しなければならない。COM への登録は、COM API の CoRegisterClassObject() というメソッドを呼び出すことによって行う。COM は、目的のクラスファクトリが登録されていることを確認してから、そのクラスファクトリを使っ

て新しいコンポーネントを生成する。明らかに、この処理は、インプロセスサーバーに比べると面倒である。インプロセスサーバーの場合、サーバーは、COM から特定のクラスファクトリに対して要求が発生したときに応答するだけだ。ローカルサーバーの場合、実際に使われるクラスファクトリがはっきりとわからないため、あらかじめ自分のクラスファクトリをすべて登録しなければならない。

COM コンポーネントの終了

すべてのサーバーが実装しなければならない最後の処理として、クリーンアップ処理のロジックがある。複数のプログラムがそれぞれ2~3のローカルサーバーを使うと、結果として多数のプログラムが同時に実行されることになる。したがって、ローカルサーバーが不要であると判断されたらすぐに破棄することは、システム全体のパフォーマンスを保つうえできわめて重要である。インプロセスサーバーの場合も同様だが、プログラムのアドレス空間にロードされる DLL によって発生するパフォーマンスの低下は、一般に、ローカルサーバーによって発生するパフォーマンスの低下ほど深刻ではない。

DLL と実行可能ファイルは本質的に異なるため、サーバーのアンロードの手順はインプロセスサーバーとローカルサーバーでは異なる。もちろん、アンロード方法の違いは、サーバーのクラスファクトリに対して COM からのアクセスを可能にする方式の違いと密接に関係している。

インプロセスサーバーでは、COM から要求された場合にのみ登録処理を行うだけでよい。COM は、定期的にサーバーの DLL に含まれる `DllCanUnloadNow()` を呼び出す。サーバーは、DLL がアンロード可能であると判断した場合には単に肯定応答を返し、なんらかの理由でアンロードできないと判断した場合にはアンロードの要求を単に拒否するだけである。実にわかりやすい。

ローカルサーバーをアンロードする場合、インプロセスサーバーよりさらに簡単だ。ローカルサーバーは完全なプログラムであるため、サーバーは自分自身を制御することができる。ローカルサーバーは自分自身のアンロードが可能であると判断すると、単に終了するだけである。このうえなく単純な処理だ。

ローカルサーバー(DLL ではない)は実行中のクラスオブジェクトを、「実行中」として登録し、特定の種類(CLSID)のオブジェクト用にプロセスを別途生成する余分な処理が不要であることを COM に通知する。

ローカルサーバーでは、サーバーが終了する前にサーバーのクラスファクトリを確実に破棄しなければならない。クラスファクトリの有効なポインタが存在している状態でサーバーが終了すると、コンピュータシステムがシャットダウンするおそれがある。システムが停止すると、ユーザーによるコンピュータのリブートが必要であるため、重要なデータの一部が失われる可能性がある。幸い、クラスファクトリの破棄は登録より簡単な処理であるが、`CoRegisterClassObject()` を使って COM に渡されたクラスファクトリはすべて、それぞれ `CoRevokeClassObject()` を呼び出して破棄しなければならない。

以上の説明で、COM サーバーのアンロードについてはかなり理解できたはずだが、これだけではまだ十分ではない。サーバーが自分自身のアンロードの可能性を判断する条件

とは、正確には何を意味するのであろうか。簡単に言えば、サーバーは、内部のオブジェクトカウンタとロックカウンタの値がすべて0になると、アンロードすることができる。このカウンタは、第1章と第2章で簡単に説明した参照カウントである。参照カウントの詳細については、後述する。

3.3 GUIDによるクラスの識別

クライアントとサーバーについて理解した後は、COM コンポーネントの構築に関する説明を続ける。

基本的に、COM のプログラミングでは、ネットワークを介して共有可能なコンポーネントを構築し、コンポーネントからオブジェクトの機能を公開するインターフェイスを取り出すという2つの作業を行う。クラスとインターフェイスは、COM においてもっとも重要な概念であり、COM のサブシステムはこの2種類の管理と操作に深く関与する。

何かを管理しなければならない場合、管理する対象に名前を付けると作業が容易になる。たとえば、スーパーマーケットの在庫管理を例として考えるとき、棚にある各商品の量をどのように把握すればよいただろうか。通常は、「朝食時に牛乳をかけて食べる明るいピンク色のリング状の商品が6箱ある」とは考えない。代わりに、それぞれの商品に名前(ネオンポップシリアルなど)を付けて、名前で商品を識別するはずだ。もちろん、これは、目からうろこが落ちるほど斬新な方法ではない。つまるところ、日常的には誰もが名前という概念を使っているのだ。

COM では、コンポーネントおよびインターフェイスを名前によって参照する。ただし、コンポーネントやインターフェイスに付けられる名前は、普段慣れ親しんでいる名前とは似ても似つかない。たとえば、コンポーネントには EDFF2BC0-1FAF-11d0-8B7B-9493759B380C、このコンポーネントがサポートするインターフェイスには EDFF2BD1-1FAF-11d0-8B7B-9493759B380C という名前が付けられる。COM で使われるこの特殊な形式の名前を、GUID (Globally Unique Identifier) と呼ぶ(「ジーユーアイディ」または「グイド」と発音する)。

GUID がこのような特殊な形式をとるのには十分な理由がある。こうしている今でも、本書の読者をはじめとする数多くのプログラマによって、COM および DCOM をベースにコンポーネントが多数作成されており、それぞれが自分のコンポーネントに名前を付けようとしている。仮に、すべてのプログラマが自分のコンポーネントに文字列の名前を付けることを Microsoft が許すと、いったいどのような混乱が生じるかを考えてほしい。いつかどこかで名前の重複が発生するのは間違いない。

GUID の重要な点は、名前の重複を避けることと、すべてのコンポーネントとインターフェイスの名前を完全に一意にすることに尽きる。DCOM 技術により COM がインターネット対応になった今では、GUID の一意性がさらに重要になっている。Microsoft が提供する GUID の生成ユーティリティには高度なアルゴリズムが採用されており、個々のコンピュータで生成される GUID が全世界規模でも必ず一意であることを保証する。このアルゴリズムは、現在の日時とネットワークカードに組み込まれているアドレスの2つのカウンタの値を組み合わせ、コンポーネントやインターフェイスの一意な ID を生成する。

GUID は、コンポーネントを一意に識別するために、時間的かつ空間的にも完全に一意な 128 ビット(16 バイト)の定数として生成される。

このような理由により、GUID を使う必要がある。続いて、GUID を生成する方法と生成した GUID を使う方法を説明する。

3.3.1 GUIDの生成

Visual C++ 6.0 には、GUID の生成に利用可能な 2 つのユーティリティ(Uuidgen と Guidgen)が提供されている。Uuidgen はコマンドラインから実行し、Guidgen は GUI を介して操作するという違いはあるが、どちらも使いやすいユーティリティだ。新しい GUID を生成するときには、この 2 つのユーティリティのどちらかを使う必要がある。何があっても絶対に、自分で勝手に GUID を生成してはいけない。しつこいようだが、必ずこの 2 つのユーティリティのどちらかを使うこと。説明したとおり、GUID は全世界規模で一意であることを保証する必要がある。GUID が重複すると、COM のサブシステムの内部がめちゃくちゃになるおそれがある。GUID の生成は慎重に行ってほしい。

注

規則には必ず例外が存在する。したがって、GUID の生成には絶対に Uuidgen または Guidgen を使わなければならないというのも大げさすぎる。これらのユーティリティを使わない場合には GUID をほかの場所から入手することになるが、これも最終的には COM の API である CoCreateGuid() を呼び出した結果として GUID を取得することになる。名前からも明らかだが、この関数は新しい GUID を生成して返す。Guidgen ユーティリティでは、まさにこの関数を使って GUID を取得する。ただし、Uuidgen は CoCreateGuid() の代わりに、同じ機能を持つ UuidCreate() という関数を使う。UuidCreate() は、オペレーティングシステムの RPC エンジンに関連付けられている。CoCreateGuid() は、単に UuidCreate() の呼び出しをラップするように実装されてる。

Uuidgen による GUID の生成

新しい GUID を 1 つだけ生成するときは、コマンドラインで Uuidgen を引数なしで実行する。Uuidgen の出力結果をテキストファイルにリダイレクトして GUID を編集可能な形式で取得する場合は、コマンドライン引数として -o を指定すると生成された GUID が直接ファイルに出力される。

```
C:\>uuidgen -oNewGUID.txt
C:\>type NewGUID.txt
23c175b0-1fbf-11d0-8b7b-9493759b380c

C:\>
```

また、引数 -n を指定すると、複数の GUID を同時に生成することができる。この場合、生成する GUID の数を -n の後に指定する。

```
C:\uuidgen -n10
e4297790-1fbf-11d0-8b7b-9493759b380c
e4297791-1fbf-11d0-8b7b-9493759b380c
e4297792-1fbf-11d0-8b7b-9493759b380c
e4297793-1fbf-11d0-8b7b-9493759b380c
e4297794-1fbf-11d0-8b7b-9493759b380c
e4297795-1fbf-11d0-8b7b-9493759b380c
e4297796-1fbf-11d0-8b7b-9493759b380c
e4297797-1fbf-11d0-8b7b-9493759b380c
e4297798-1fbf-11d0-8b7b-9493759b380c
e4297799-1fbf-11d0-8b7b-9493759b380c

C:\
```

引数 `-n` と生成する GUID の数の間には空白を入れないので、注意する。同様に、引数 `-o` と GUID の出力先になるファイル名の間にも空白を入れない。

警告



Uuidgen を実行するコンピュータにネットワークカードが取り付けられていない場合、また、Uuidgen がカードのネットワークアドレスを読み取ることができない場合、“Warning: Unable to determine your network address. The UUID generated is unique on this computer only. It should not be used on another computer.” というメッセージが表示される。この場合、生成される GUID の一意性は保証されない。これは、ネットワークカードのないコンピュータを使っている別のプログラムが同じ GUID を生成する危険が高くなるためだ。GUID を生成する時刻まで一致している場合には、さらにこの危険性が高くなる。原則として、GUID の生成には、この警告メッセージが表示されるおそれのないコンピュータを使うべきである。

Uuidgen には、このほかにもソースコードに直接挿入可能な形式で GUID を出力するためのオプションがある。

Guidgen による GUID の生成

Guidgen は、機能的には Uuidgen とまったく同じであるが、単純なグラフィカルインターフェイスを提供する。Guidgen では、一度に生成可能な GUID は 1 つに限定され、GUID をテキストファイルに書き出すこともできない。ただし、Windows のクリップボードがサポートされており、GUID をソースコードのファイルに直接貼り付けることができる。

Guidgen は、コマンドライン引数をとらず、**図 3-5** に示すダイアログボックスを表示する。

図 3-5 からわかるように、Guidgen では GUID を生成するオプションの数が Uuidgen より多い。たとえば、Guidgen コーティリティでは、IMPLEMENT_OLECREATE というマクロ (MFC が GUID の宣言に使う) に対応した形式で GUID を生成することができる。

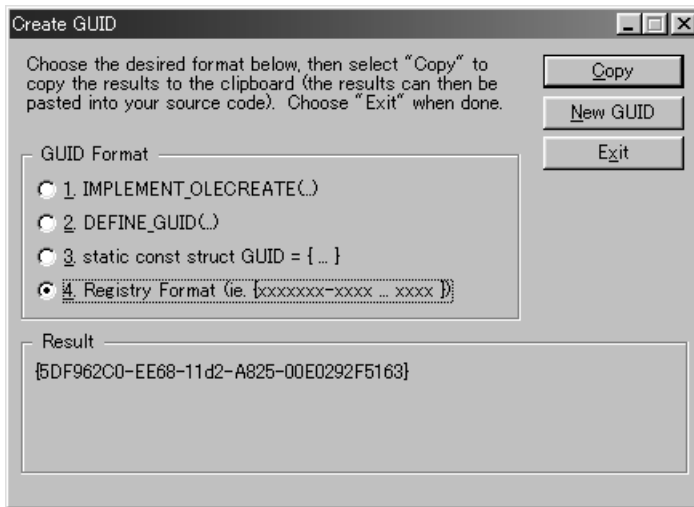


図 3-5 Guidgen ユーティリティ

警告

Uuidgen と異なり、Guidgen は、コンピュータのネットワークアドレスがわからなくても警告を表示しない。したがって、最初に Uuidgen を実行して GUID の生成が正しく実行できることを確認してから、Guidgen を実行することをお勧めする。

Guidgen は、特別優れたプログラムではないが、気に入らないところがあれば自由にカスタマイズすることができる。MFC には、Guidgen の完全なソースコードがサンプルプログラムの 1 つとして付属している(詳細については、Visual C++ のオンラインヘルプを参照)。

3.3.2 GUID の記述方法

GUID は、プログラマによる操作を前提として設計されているため、GUID の実際の構成を知る必要はないが、GUID の操作方法を理解する必要がある。つまり、GUID はそれぞれ、Wtypes.h というヘッダーファイルに定義されている構造体のいずれかに対応していることを認識しなければならない。Wtypes.h は、Visual C++ の \Microsoft Visual Studio\VC98\Include というディレクトリに格納されている。開発するアプリケーションのほとんどの部分では GUID の構成について意識する必要はないが、GUID の宣言と初期設定を行う部分では GUID の構造体についてある程度知識が必要だ。Microsoft は、この作業を支援するために多数のマクロを提供しているが、構造体の定義を実際に確認して処理の対象を把握するのに越したことはない。Wtypes.h というヘッダーファイルには、次に示す構造体が定義されている。

```
typedef struct _GUID
{
    DWORD Data1;
    WORD Data2;
    WORD Data3;
    BYTE Data4[ 8 ];
}GUID;
```

GUID の構造体は、4 つの属性で構成され、全体で 16 バイトの記憶域を使う。これまでに紹介した GUID の例を参照すると、この構造体の各フィールドが、生成された GUID のハイフンで区切られたそれぞれの部分にほぼ対応していることがわかる。

```
23c175b0-1fbf-11d0-8b7b-9493759b380c
```

Uuidgen および Guidgen が生成する GUID は 16 進形式で表される。したがって、1 バイトは 2 桁の 16 進数で表される。この GUID の場合、Data1 フィールドは 0x23c175b0、Data2 フィールドは 0x1fbf、Data3 フィールドは 0x11d0、Data4 フィールドは残りの部分 (0x8b7b9493759b380c) にそれぞれ対応する。唯一、注意しなければならないのは、GUID の 4 番目の領域になる 2 バイト (0x8b7b) が、Data4 に含まれていることだ。

GUID を使ってクラスとインターフェイスに名前を付ける方法を説明してきた。続いて、CLSID と IID というデータ型について説明する。CLSID と IID はどちらも、標準の GUID 構造体の typedef になる。より簡単に説明すると、CLSID と IID は、GUID が指す対象をより明確にするためのシンボル定数である。CLSID はクラスまたはコンポーネントを識別し、IID はインターフェイスを識別する GUID だ。原則として、クラス ID の変数名には接頭辞として CLSID_ が、インターフェイス ID の変数名には IID_ が付けられる。Chihuahua コンポーネントの例の場合、GUID の宣言は次のようになる。

```
CLSID CLSID_Chihuahua;
IID IID_IDog;

static const CLSID CLSID_Chihuahua =
{ 0x86ecd437, 0x1fd9, 0x11d0, { 0x8b, 0x7c, 0xe4, 0x45, 0xc9,
0xbd, 0x31, 0xc } };

static const IID IID_IDog =
{ 0x86ecd438, 0x1fd9, 0x11d0, { 0x8b, 0x7c, 0xe4, 0x45, 0xc9,
0xbd, 0x31, 0xc } };
```

CLSID と IID は実際には GUID であるため、この変数は次のような形式で宣言することもできる。

```
static const GUID CLSID_Chihuahua =
{ 0x86ecd437, 0x1fd9, 0x11d0, { 0x8b, 0x7c, 0xe4, 0x45, 0xc9,
0xbd, 0x31, 0xc } };

static const GUID IID_IDog =
{ 0x86ecd438, 0x1fd9, 0x11d0, { 0x8b, 0x7c, 0xe4, 0x45, 0xc9,
0xbd, 0x31, 0xc } };
```

これは、Guidgen を使って出力結果をクリップボードからカットアンドペーストした場合の形式の1つである。これらの定義は、Microsoft が提供する DEFINE_GUID というマクロを使ってもう少し見やすい形式で出力することができる。DEFINE_GUID は、宣言の対象となる変数の名前を最初のパラメータとして受け取り、その後 GUID の各部分(1番目が DWORD 型の値、2番目と3番目が WORD 型の値、4番目から11番目までが1バイトずつの値)を16進形式で指定する。たとえば、前述の CLSID_Chihuahua と IID_IDog の定義は、DEFINE_GUID マクロを使うと次に示す形式になる。

```
DEFINE_GUID(CLSID_Chihuahua,
            0x86ecd437, 0x1fd9, 0x11d0, 0x8b, 0x7c,
            0xe4, 0x45, 0xc9, 0xbd, 0x31, 0xc);

DEFINE_GUID(IID_IDog,
            0x86ecd438, 0x1fd9, 0x11d0, 0x8b, 0x7c,
            0xe4, 0x45, 0xc9, 0xbd, 0x31, 0xc);
```

DEFINE_GUID マクロを使うことが推奨される重要な理由が1つある。クラス ID やインターフェイス ID の宣言では、アプリケーション内の任意の場所からアクセス可能な、グローバルで静的な定数を設定するため、実際の GUID 自体が1つのソースファイルで一度だけ定義されることを確認しなければならない。

これは、実際には最初に思ったより難しい作業である。たとえば、一番簡単な方法として、GUID の定義をヘッダーファイルにそのまま格納し、GUID を必要とするプログラムでヘッダーファイルをインクルードさせるという方法が考えられるが、この方法は無効である。このヘッダーファイルを複数回インクルードすると、コンパイル時にその GUID の変数のシンボルが重複しているというエラーが発生する。この場合、目的はすべてのソースファイルで使用可能なヘッダーファイルに GUID を宣言することだが、実際には1つのソースファイルにのみ GUID を定義することになる。DEFINE_GUID は、この目的を支援するために複雑な処理を行う。

DEFINE_GUID は、デフォルトでは入力した GUID の各部分をすべて無視する。たとえば、次のように GUID を入力する。

```
DEFINE_GUID(IID_IDog,
            0x86ecd438, 0x1fd9, 0x11d0, 0x8b, 0x7c,
            0xe4, 0x45, 0xc9, 0xbd, 0x31, 0xc);
```

マクロはこの GUID を次のように展開する。

```
extern "C" const GUID IID_IDog;
```

言い換えると、GUID は宣言されるが、実際には定義されない。そこで、標準の OLE のヘッダーファイルをインクルードする前にプリプロセッサマクロの INITGUID を宣言すると、DEFINE_GUID マクロの定義は問題なく機能し、次のように展開される。

```
extern "C" const GUID IID_IDog = \
{ 0x86ecd438, 0x1fd9, 0x11d0, { 0x8b, 0x7c, 0xe4, 0x45, 0xc9,
0xbd, 0x31, 0xc } };
```

この方法では、GUID の宣言と定義が完全に行われている。ここで重要なのは、ソースコードにインクルードする GUID のそれぞれに対し、INITGUID を正確に 1 回だけ設定する必要があるということだ。INITGUID をまったく設定しなければ、リンクの処理は失敗し、プログラム内で宣言して使っている GUID の定義を検出できないというエラーが表示される。一方、複数のソースファイルで GUID のインクルードと INITGUID の設定を行うと、前述と同様にシンボルの重複問題が発生する。

3.4 HRESULT 型の戻り値

COM の API を使う際には、ほとんどの場合に COM で共通的に使われる HRESULT という戻り値を利用する。単純なエラーコードを戻す従来の API とは異なり、HRESULT は、4 バイトの符号なし long 型の値に収められる小型の構造体とみなされる。構造体であるため、HRESULT では、特定の操作の成功や失敗をきわめて柔軟に表すことができる。

注

Windows の初期のバージョンでは、一部のアプリケーションは SCODE 型を返していた。Windows と OLE の以前のバージョンでは、SCODE は HRESULT とは基本的に異なるエラー型である。一方、Windows 9x および Windows NT の現在のバージョンでは、SCODE と HRESULT は同じ型である。SCODE を使わないことが推奨されており、Microsoft が提供する新しいインターフェイスはすべて HRESULT 型を使っている。本書で紹介する例では 32 ビットのコードだけを扱っており、勝手ながら Windows 3.1 との互換性は考慮に入れていないため、本書では HRESULT 型だけを使う。

HRESULT は、成否フィールド、機能フィールド、説明フィールドの 3 つの部分から構成される。COM のサブシステムは、HRESULT からこの 3 つのフィールドのそれぞれを抽出するためのマクロを提供している。実際のプログラミングでは、HRESULT を参照して個々のフィールドを自分で見分けることを考えずに、Microsoft が提供するマクロを使うほうがよい。HRESULT 関連のマクロを使うと、たとえば、Microsoft が HRESULT の定義を変えた場合にも互換性を保持することができる。

HRESULT の成否フィールドは、HRESULT が処理の失敗または成功のどちらを通知しているかを示す。一般に、ある処理を実行しようとしたときにそれが失敗した場合、成否フィールドがエラーの発生を示している HRESULT を返す。このように、成否を説明フィールドで暗黙的に示すのではなく、専用のフィールドを使って通知する。この方法の長所の 1 つは、Windows またはコンポーネントが成功を示す複数のコードを定義できることだ。たとえば、1 つのコンポーネントにおいて、処理が無条件に成功したことを示すコードと、処理がある条件付きで成功したことを示すコードをそれぞれ別に定義することができる。

警告



成功した処理は HRESULT の複数のコードで示される可能性があるため、戻り値の HRESULT を、成功を表すのに広く使われている S_OK などの定数とただ単に照合するだけでは不十分である。代わりに、後述する FAILED() や SUCCEEDED() などのマクロを使う。

機能フィールドは、HRESULT が所属する大まかな機能区分を示す。Microsoft は、この機能区分を複数定義しているが、ユーザー独自の HRESULT 値として使用できるのは FACILITY_ITF に限られる。表 3-1 に、現在定義されている HRESULT の機能フィールドを示す。

表 3-1 HRESULT の有効な機能コード

機能フィールド	説明
FACILITY_NULL	すべての機能で共有できる完全に汎用的な HRESULT に使う。S_OK や S_FALSE などの HRESULT に使われる。
FACILITY_ITF	カスタムインターフェイス (Microsoft が提供するもの以外のインターフェイスなど) が返す HRESULT の機能フィールドは、必ず FACILITY_ITF になる。したがって、2つのインターフェイスがまったく同じ HRESULT を返し、それぞれの HRESULT が生成元のインターフェイスに応じて別の意味を持つ場合もある。
FACILITY_DISPATCH	OLE オートメーション技術に対応する HRESULT 用に予約されている。
FACILITY_RPC	RPC に対応する HRESULT 用に予約されている。
FACILITY_STORAGE	OLE の構造化ストレージインターフェイスに対応する HRESULT 用に予約されている。
FACILITY_WIN32	Win32 の API では、関数に応じてさまざまな種類のエラーコードを返す。このコードは、Win32 の標準のエラーコードを HRESULT 形式でグループ化するために使われる。
FACILITY_WINDOWS	前述のコードのいずれにも該当しない、Microsoft のそのほかのエラーコードに使われる。

HRESULT の説明フィールドは、ほとんどのプログラムで戻り値を受け取ったときに検討しなければならないものだ。説明フィールドは、実際に発生した状況を示す文脈依存のコードである。

Microsoft のコードから返された HRESULT を表すシンボル定数には、一定の標準形式で名前が付けられている。実際のプログラミングでも、インターフェイスにこの形式を採用することが多い。この名前の最初の部分は、通知する内容を機能フィールドより具体的に表す。2番目の部分は成否を、3番目の部分は具体的な状況を簡単に表す。たとえば、CLIPBRD_E_BAD_DATA では、E によってエラーが発生したことを示す。この HRESULT の生成元になった機能は、Microsoft のクリップボードの処理で不正なデータに関するなんらかのエラーが発生したことを表している。また、DRAGDROP_S_DROP の場合、ドラッグアンドドロップコンポーネントがドロップ処理の成功を示す値を戻したことがわかる。S_OK などの HRESULT は解釈が簡単で、処理は成功し、何もかもが OK であることを表している。

以前にも説明したが、HRESULT からフィールドを抽出する場合には、Microsoft が提供する標準のマクロを使うことをお勧めする。表 3-2 に、HRESULT の操作に使用可能なマクロを示す。マクロの使用方法については、Visual C++のオンラインヘルプを参照。

表 3-2 HRESULT の操作に使用可能な標準マクロ

マクロ	説明
FAILED()	指定の HRESULT が処理の失敗を表している場合には True を、それ以外の場合には False を返す。
IS_ERROR()	指定の HRESULT の成否フィールドがエラーを表している場合には True を返す。機能的には FAILED() と同じである。
SUCCEEDED()	指定の HRESULT が処理の成功を表している場合には True を、それ以外の場合には False を返す。
HRESULT_SEVERITY()	HRESULT を入力として受け取り、HRESULT から成否フィールドを抽出して返す。
HRESULT_FACILITY()	HRESULT を入力として受け取り、HRESULT から機能フィールドを抽出して返す。
HRESULT_CODE()	HRESULT を入力として受け取り、HRESULT から説明フィールドを抽出して返す。
MAKE_hresult()	3つの値 成否を表す値、機能を表す値、説明コード を入力として受け取り、これらの値から HRESULT を生成して返す。

3.5 インターフェイスの詳細

この時点では、COM インターフェイスを C++ で表す方法がまだはっきりとわからない。結局、C++ では、**インターフェイス**を固有の機能としてサポートしていない。インターフェイスについてはすでに第 1 章と第 2 章で詳しく説明したが、復習のために再度説明する。第 2 章で説明したように、Microsoft が COM と OLE を設計したときの目的の 1 つは、言語に依存しないオブジェクトモデルを構築することであった。これは、COM オブジェクトがメモリに展開されたときの状態を規定するバイナリ仕様を作成することを意味する。さらに、この仕様には、ポインタや関数などの基本的な概念をサポートする任意の言語で実装できるように汎用性を持たせる必要があった。もちろん、COM のサポートをその言語に適した方法で隠すのは自由であるが、最終的には、COM オブジェクトとのやり取りおよび COM オブジェクトの処理を COM 準拠のプログラミング言語に共通のバイナリレベルで行うことになる。

COM オブジェクトとして利用可能な C++ のオブジェクトでは、VTable が中核になる。以前にも説明したように、インターフェイスは意味的に関連を持つ COM のメソッドの集合にすぎない。インターフェイスの VTable は、インターフェイスの各メソッドのポインタを 1 つのメモリテーブルの形にまとめたものだ。VTable の各エントリは、インターフェイスの各メソッドのアドレスに対応する。図 3-6 に、第 2 章で紹介した IDog インターフェイスの VTable の例を示す。

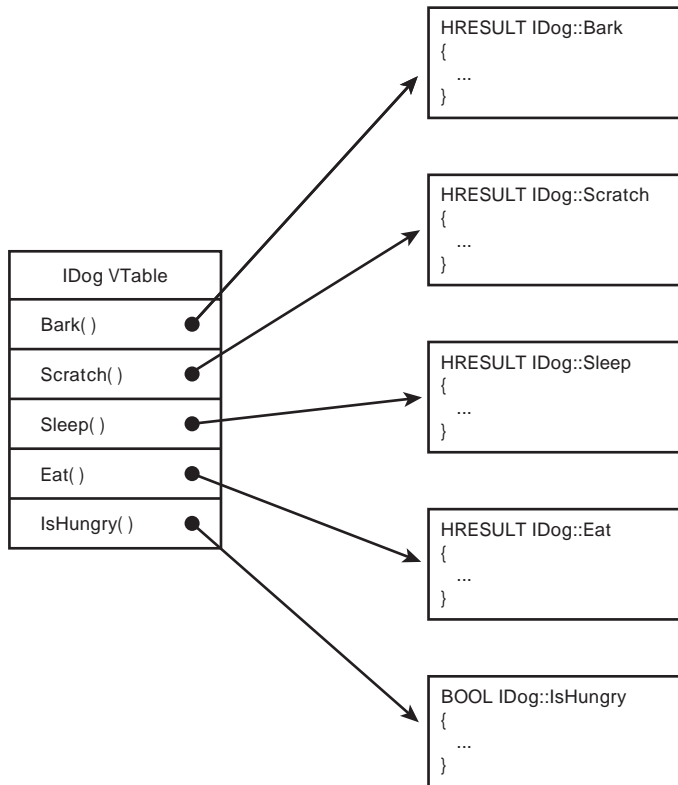


図3-6 IDog インターフェイスのVTable

COM インターフェイスのポインタは、インターフェイスの VTable のポインタを指すポインタにすぎない。インターフェイスは固有のプライベートデータを外部から見えない状態で保持し、プライベートデータの操作はインターフェイスのメソッドの呼び出しを介して行う。プライベートデータは、メモリ内では VTable の直後に格納される。図 3-7 に、IDog インターフェイスを例として、インターフェイス全体の配置図を示す。

このようにメモリ上に配置することの利点は、C++では仮想関数を持つ C++オブジェクトがまったく同じ形式で配置されるということだ。つまり、COM インターフェイスは、1つの C++の抽象基本クラスによって簡単に表すことができる。

このように COM と C++でオブジェクトの配置が一致しているのはもちろん偶然ではない。もともと、COM のバイナリ構造は、C++のバイナリ構造を直接のモデルとしている。これは、VTable という用語が C++に由来していることからわかる。VTable は、Virtual Function Table(仮想関数テーブル)の略で、COM が登場するずっと以前から C++の用語として存在していた。

しかし、C++とは異なり、COM の仕様では、オブジェクトをメモリ上でどのように表現するかが詳細かつ明示的に規定されている。一方、C++の仕様にはこのような規定はない。

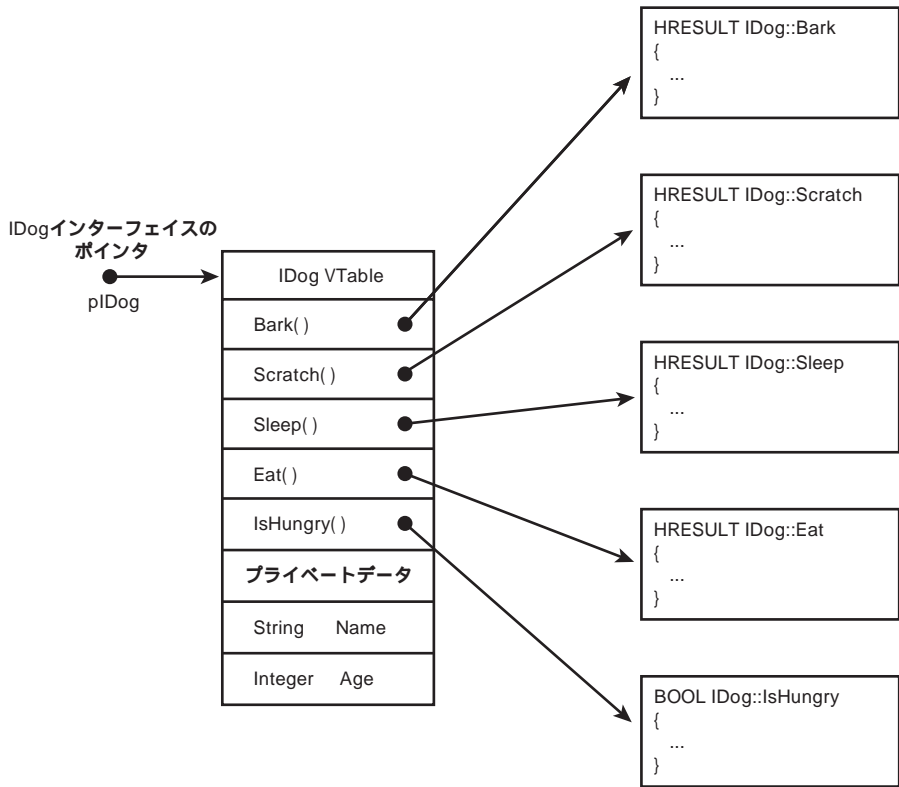


図 3-7 IDog インターフェイスのメモリ上の配置

COM インターフェイスを構築する場合、原則として、インターフェイスは純粹仮想関数を持つ C++ の抽象基本クラスとして記述する。コンポーネントがあるインターフェイスをサポートする場合、そのインターフェイスクラスから継承を行って必要なメソッドを実装するだけでよい。

ところで、COM が言語に依存しないのは COM オブジェクトが C++ オブジェクトと同じようにメモリに配置されるからではないだろうか。この考えは、完全に正しいとは言えない。COM のバイナリオブジェクトの仕様は C++ のオブジェクトの構造を直接の基盤としているが、C++ のオブジェクトは C++ の機能を完全にサポートするために付加的な要素を数多く保持している。個々の COM インターフェイスは 1 つの単純な C++ オブジェクトとして実装できるが、すべての C++ オブジェクトが COM インターフェイスになるわけではない。つまり、COM のバイナリオブジェクト仕様は、C++ のバイナリオブジェクト仕様のサブセットと考えることができる。

C++ におけるインターフェイスの宣言の例として、リスト 3-1 に IDog インターフェイスの宣言を示す。COM インターフェイスを実際に C++ で宣言する方法は、この例とは多少異なるが、ほとんど同じである。インターフェイスクラスを宣言する正確な方法については、Fortune2 というサンプルプログラムとともに説明する。

リスト 3-1 C++における IDog インターフェイスの宣言(IDog1.h)

```

class IDog : public IUnknown
{
public:
    virtual HRESULT Bark() = 0;
    virtual HRESULT Scratch() = 0;
    virtual HRESULT Sleep() = 0;
    virtual HRESULT Eat() = 0;
    virtual BOOL    IsHungry() = 0;
};

```

このコードを見てもわかるとおり、C++では簡単にインターフェイスを宣言することができる。

C++のコード例では、IDog インターフェイスを IUnknown インターフェイスから派生している。COM では、すべてのインターフェイスが IUnknown から派生する。使用言語が継承をサポートしていない場合、すべてのインターフェイスが IUnknown をサポートし、インターフェイスの VTable に正しいポインタがロードされることをプログラマが保証しなければならない。IUnknown インターフェイスは、コンポーネントが公開する IUnknown 以外のすべてのインターフェイスに対するアクセスを可能にする機能を提供する。つまり、クライアントは、必要なコンポーネントのインターフェイスの任意のポインタを取得すると、IUnknown を使ってコンポーネントが公開するほかのインターフェイスにアクセスすることができる。

注

1 つの COM コンポーネントは複数のインターフェイスを公開することができる。実際、コンポーネントはすべて、少なくとも 2 つのインターフェイス(1 つは IUnknown)を公開する。複雑なコンポーネントの場合、公開するインターフェイスの数はさらに多くなる。たとえば、複雑な機能を持つ ActiveX コンポーネントの場合、15 種類以上のインターフェイスを公開する場合もある。複数インターフェイスのサポートは、意外に複雑な問題である。複数インターフェイスのサポートについては、第 5 章「MFC による COM のプログラミング」で説明する。

リスト 3-2 に、前述の C++のコード例と同じインターフェイスを C で記述する方法を示す。

リスト 3-2 C における IDog インターフェイスの宣言(IDog2.h)

```

// 宣言の開始
typedef struct IDog IDog;

// VTable の構造体
typedef struct
{

```

```

// IUnknown のメソッド
HRESULT (*QueryInterface )(IDog *this,
                           REFIID riid,
                           void **ppvObject);
ULONG (*AddRef )(IDog *this);
ULONG (*Release )(IDog *this);

// IDog のメソッド
HRESULT (*Bark)(void);
HRESULT (*Scratch)(void);
HRESULT (*Sleep)(void);
HRESULT (*Eat)(void);
BOOL (*IsHungry)(void);

} IDogVtable;

// IDog インターフェイスの実際の構造体
typedef struct
{
    struct IDogVtable *lpVtable;
} IDog;

```

同じインターフェイスでも、CのほうがC++のコードより少し複雑になっている。COM オブジェクトをCでプログラミングする場合、VTableを明示的に宣言して設定しなければならないため、C++のプログラミングより手間がかかる。Cでは、まずVTableの構造体そのものを宣言し、インターフェイスのメソッドに対応するポインタのエントリをすべて記述しなければならない。次に、インターフェイス自体の構造体を宣言しなければならないが、この構造体にはVTableのポインタだけが格納される。

Cではオブジェクトの継承がサポートされていないため、IUnknownインターフェイスのメソッドのポインタを、コンポーネントのVTableの構造体に直接手動で設定しなければならない。この処理を行うと、Cで作成したコンポーネントが、C++で作成したコンポーネントとまったく同じように機能する。

CによるCOMオブジェクトのプログラミングはC++ほど洗練されてはいないが、確かに可能ではある。ただし、本書では、C++による32ビットのCOM/DCOMプログラミングだけを想定する。C++を使うほうが、COMクライアントとCOMサーバーを構築する処理はるかに簡単であるからだ。さらに、C++はCOMのバイナリ構造をサポートしているため、実際にCOM/DCOMのプログラミングを行うときにはC++の使用が強く推奨される。

3.6 IUnknownの詳細

第2章でポリモーフィズムについて説明したときに、COMコンポーネントは必ずIUnknownインターフェイスを実装しなければならないと記述した。IUnknownインターフェイスはCOMの中でも特別な位置を占めているが、これは、COMではコンポーネントの直接のポインタをクライアントが取得することは決していないためだ。クライアントは、

インターフェイスポインタを介してCOMオブジェクトとやり取りする。IUnknownは、すべてのオブジェクトが必ず公開する唯一のインターフェイスである。COMのAPIとインターフェイスは、一般に、新しいオブジェクトを生成したときにIUnknownインターフェイスのポインタという形でクライアントに返す。これまでにオブジェクトのポインタの受け渡しを行うプログラムを作成した経験があれば、ここで考え方を改めてIUnknownインターフェイスのポインタをやり取りする方法を習得しなければならない。

IUnknownインターフェイスはきわめて単純だが、見ただけではわからない大きな力を秘めている。IUnknownの3つのメソッドは、COMオブジェクトモデル全体で広く使われている。

```
// IUnknown インターフェイス
class IUnknown
{
public:
    virtual HRESULT QueryInterface(REFIID riid,
                                   void **ppvObject) = 0;

    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};
```

IUnknownインターフェイスはきわめて単純に見えるが、これから説明するように見た目よりはるかに多くの機能を提供している。

3.6.1 QueryInterface()の使用

IUnknownインターフェイスのQueryInterface()メソッドは、COM全体でもっとも重要なメソッドであることは間違いない。COMにおけるオブジェクト間の対話はすべてインターフェイスを介して行われるが、アプリケーションやコンポーネントはこのインターフェイスポインタをどのように取得するのだろうか。現時点では、QueryInterface()を使うというのがその答えである。

QueryInterface()はパラメータとして、取得するインターフェイスを示すGUIDと取得したインターフェイスのポインタが返される変数のアドレスを受け取る。REFIIDは、インターフェイスのGUIDを指すC++のconst型(値が不変であることを示す)の参照である。パラメータとしてREFIIDを受け取るAPIやメソッドには、GUIDの構造体を値渡しではなく参照渡しでやり取りする。値渡しでは、GUIDの構造体をスタックにコピーしなければならないので処理がかなり遅くなるためだ。リスト3-3に、コンポーネントのIUnknownインターフェイスのポインタが与えられている場合に、QueryInterface()メソッドを使ってコンポーネントのインターフェイスを取得するコード例を示す。ここでは、IChihuahuaインターフェイスを取得している。

リスト 3-3 QueryInterface() を使ってインターフェイスポインタを取得する(QueryInt.cpp)

```

// IUnknown インターフェイスの有効なポインタが変数 pIUnknown に格納されている

HRESULT hResult = E_FAIL;
IChihuahua *pIChihuahua = NULL;

// コンポーネントの IChihuahua インターフェイスを照会する
hResult = pIUnknown->QueryInterface( IID_IChihuahua,
                                     (void**) &pIChihuahua);

// 処理は成功か
if ( FAILED(hResult) )
{
    // エラーの原因を特定する
    switch ( hResult )
    {
        // コンポーネントがこのインターフェイスをサポートしていない
        case E_NOINTERFACE:
            cout << "Component does not expose IChihuahua.\n";
            break;

        // COM がスレッドを初期化していない
        case CO_E_NOTINITIALIZED:
            cout << "COM not initialized.\n";
            break;

        // コンポーネントのメモリ不足
        case E_OUTOFMEMORY:
            cout << "Out of memory!\n";
            break;

        // そのほかのエラー
        default:
            cout << "Unrecognized error.\n";
    }

    return hResult;
}

```

この例では、インターフェイスポインタの取得時に発生する可能性のある問題が想定されている。もっとも発生する可能性が高いのは E_NOINTERFACE が示すエラーだ。E_NOINTERFACE は、コンポーネントが取得するインターフェイスをサポートしていない場合に返される。呼び出し元では、E_NOINTERFACE が返された場合の処理を決めなければならない。たとえば、ある重要なコンポーネントが IChihuahua を公開していないことがわかった場合、クライアントプログラムは、必要な機能が部分的に実装されていることを期待して IDog インターフェイスを取得するなどの処理を行うことができる。また、クライアントが処理を続行するためにどうしても IChihuahua を必要としている場合、E_NOINTERFACE が返されるとより深刻なエラー状態に陥ることになる。

3.6.2 AddRef()とRelease()による参照カウント

IUnknown には、QueryInterface() のほかに、AddRef() と Release() という 2 つのメソッドがある。AddRef() と Release() はペアで、COM コンポーネントの参照カウントの調整に利用する。コンポーネントの参照カウントは、コンポーネントが破棄可能かどうかを判断するために利用される。

一般的な COM ベースのアプリケーションでは、複数のコンポーネントまたは複数のクライアントのコードが同時に同じオブジェクトを使用したり、同じオブジェクトと対話するなどの状況がよく発生する。この場合、あるオブジェクトが未使用状態になったタイミングを、クライアントが判断しにくいという問題がある。

たとえば、あるクライアントアプリケーションが、1 つの Chihuahua コンポーネントを使っている。このクライアントアプリケーションは Kennel オブジェクトと Veterinarian オブジェクトから構成されており、それぞれが QueryInterface() を使って、Chihuahua コンポーネントの IDog インターフェイスポインタのポインタを取得する。Kennel クラスは Veterinarian クラスをまったく認識していない。また、Veterinarian クラスも Kennel クラスをまったく認識していない。では、このクライアントアプリケーション(または該当するサーバー)は、Chihuahua オブジェクトが未使用状態になったタイミングをどのように検出するのだろうか。Kennel オブジェクトは自分が Chihuahua オブジェクトを使い終えたことを認識しており、同様に Veterinarian クラスも Chihuahua オブジェクトを使い終えたことを認識している。しかし、Kennel および Veterinarian は、ほかのオブジェクトが該当する Chihuahua オブジェクトにアクセスしていないことを断定できない。では、Chihuahua オブジェクトが使用中であることを誰が認識しているのだろうか。

COM では、Chihuahua オブジェクト自体に自分を使用している外部エンティティを認識させることにより、この問題を解決している。各 COM オブジェクトは、自分を使用しているほかのオブジェクトおよびプログラムの数をリアルタイムで把握している。オブジェクトは、この参照カウントが 0 になると、未使用状態であり、破棄可能であることを認識する。

もちろん、Chihuahua オブジェクトが自分を使用しているオブジェクトをすべて認識しているというのは多少正確さに欠ける。各コンポーネントが内部の参照カウントを維持するのを支援するのは、クライアントの責任になる。クライアントは、この目的のために AddRef() メソッドと Release() メソッドを使う。

コンポーネントのクライアントは、いくつかの状況においてオブジェクトの参照カウントの管理に積極的に関係する。クライアントは、インターフェイスポインタのコピーを生成するたびにそのコンポーネントの AddRef() メソッドを呼び出して、オブジェクトの参照カウントをインクリメントする。一方、クライアントが持つインターフェイスポインタが破棄される(スコープ外に出るまたは上書きされる)たびにそのコンポーネントの Release() メソッドを呼び出して、オブジェクトの参照カウントをデクリメントしなければならない。AddRef() メソッドおよび Release() メソッドはどちらもパラメータをとらずに、戻り値として参照カウントを返す。

さらに、あるオブジェクトの参照カウントが 0 になると、そのコンポーネントをいつでも破棄できるということを忘れないでほしい。実際、ほとんどの COM コンポーネントは、直ちに自分自身で破棄処理を行う。C++ では、`Release()` メソッドの内部からオブジェクトの `this` ポインタを削除することによって破棄される。オブジェクトの参照カウントは、理論的には `Release()` の任意の呼び出しの後に 0 になる可能性があるため、インターフェイスポインタに対して `Release()` を呼び出した後にそのインターフェイスポインタを絶対に使用してはならない。インターフェイスポインタを解放したらすぐに `NULL` を設定するのが賢明だ。

警告



`delete this` を実行するのは穏当ではないように思われるが、C++ ではこれは正しい処理である。

オブジェクトの `Release()` メソッドにおいて `delete this` の処理が実行された後、そのオブジェクトに関するコードをいっさい実行しないという点がきわめて重要だ。実際、安全のためにも、コンポーネントのコーディングでは、`delete this` の直後には常に `return` などの制御を返すステートメントを置くことが強く推奨される。

いくつか例を考えてみよう。コンポーネントが新しく生成されるたびに、クライアントには 1 つのインターフェイスが直接返される。または、コンポーネントのクラスファクトリを介して間接的に返される(この処理については後述する)。この時点では、インターフェイスのコピーは 1 つだけ公開されているため、このコンポーネントの参照カウントは 1 になる。

この後、`QueryInterface()` の呼び出しが成功するたびに、このオブジェクトの参照カウントは 1 つずつ増える。この処理をオブジェクトの `QueryInterface()` メソッドに実装するのは、コンポーネントの責任になる。もう少し一般的な言い方をすると、あるコンポーネントが任意のメソッド(`QueryInterface()` に限らない)からインターフェイスポインタを返すたびに、そのオブジェクトの参照カウントを必ずインクリメントしなければならない。

表 3-3 に、あるオブジェクトが生成されてから破棄されるまでに発生する一連のイベントの例を示す。インターフェイスポインタのコピー、上書きが行われ、最終的に解放されるまでの状況がはっきりとわかる。この例では、`pIUnknown` という変数に `IUnknown` インターフェイスのポインタが最初から格納されていると想定しているため、参照カウントは 1 から始まる。

参照カウントが複雑になるのはやむを得ない。しかし、コンポーネントのすべてのクライアントがインターフェイスポインタの使用状況を正確に追跡することはきわめて重要である。`AddRef()` または `Release()` の呼び出しを 1 つ省略しただけでも、大きな問題につながる場合がある。`Release()` の呼び出しを省略すると、メモリやリソースの不足が発生するおそれがある。`AddRef()` の呼び出しを省略すると、アプリケーションの一部でポインタが不正になり、無効なメモリ領域を参照するおそれがある(これはほぼ確実にプログラムのクラッシュにつながる)。

表 3-3 インターフェイスポインタの参照カウント

コード	参照カウント	説明
IDog *pIDog1 = 0; IDog *pIDog2 = 0; IChihuahua *pIChihuahua1 = 0;	1	この例で使う変数の宣言である。
pIUnknown->QueryInterface (IID_IDog, (void**) &pIDog1);	2	新しいIDog インターフェイスのポインタが返されることにより、オブジェクトの参照カウントが自動的に1つ増える。
pIDog2 = pIDog1; pIDog2->AddRef();	3	オブジェクトは、クライアントが pIDog1 のインターフェイスポインタのコピーを1つ生成したことを認識することができない。そのため、クライアントは、AddRef() を呼び出すことによって手動で参照カウントを1つ増やさなければならない。ただし、pIDog1 および pIDog2 はどちらも同じインターフェイスポインタを指すため、AddRef() を pIDog1 または pIDog2 のどちらに対して呼び出すかは重要ではない。
pIDog1->Release(); pIDog1 = NULL;	2	pIDog1 のインターフェイスポインタが NULL に設定されることを想定し、クライアントは Release() を呼び出して参照カウントを1つ減らす。
pIUnknown->QueryInterface (IID_IChihuahua, (void**) &pIChihuahua1);	3	新しいインターフェイスポインタが返されることにより、オブジェクトの参照カウントが自動的に1つ増える。
{ (void**) &pIChihuahua1; IChihuahua *pInner = 0; pInner = pIChihuahua1; pInner->AddRef();	4	ここから次のレベルのスコープが開始する。変数 pInner は、新しいスコープ内でのみ有効である。pIChihuahua からポインタがコピーされたため、クライアントは AddRef() を呼び出すことによって手動でオブジェクトの参照カウントを1つ増やさなければならない。
pInner->Release(); }	3	スコープが終わりに近づき、変数 pInner が破棄される。したがって、クライアントは pInner がスコープ外に出る前に Release() を呼び出す必要がある。
pIChihuahua1->Release(); pIDog2->Release();	1	クライアントがインターフェイスポインタを使い終えたため、各インターフェイスポインタに対して Release() が呼び出される。この時点で、オブジェクトの参照カウントは1に戻る。1は元の pIUnknown のポインタが使用中であることを示す。
pIUnknown->Release();	0	クライアントは、最後に残っているインターフェイスポインタを解放する。オブジェクトの参照カウントは0になり、このコンポーネントは自己破棄を行う。

Kraig Brockschmid(OLE プログラミングのバイブルと呼ばれる *Inside OLE*^{*2}の著者)、Don Box(COM の解説書として広く読まれている *Essential COM*^{*3}の著者)などの COM の権威は、参照カウントのパターンを**取得、使用、解放**とみなすことが多い。一般に、クライアントが新しいインターフェイスポインタを返す COM の API 関数やコンポーネントのメソッドを呼び出すと、クライアントはインターフェイスポインタを**取得**することになる。この場合、コンポーネント自身がポインタを返す前に自分の参照カウントを1つ増やすということが基本的な前提条件になる。続いて、クライアントは返されたインターフェイスポインタを**使用**して、一連の操作やなんらかの処理を実行する。最後に、クライアン

*2 *Inside OLE, Second Edition*(Microsoft Press、1995年)
邦訳は、長尾高弘訳『*Inside OLE 改訂新版*』(アスキー出版局、1996年)

*3 *Essential COM*(Addison Wesley Longman、1998年)
邦訳は、長尾高弘訳『*Essential COM*』(アスキー出版局、1999年)

トは、インターフェイスポインタを使い終わるとインターフェイスを解放し、それによって参照カウントが1つ減る。この3つの手順を常に念頭に置くと、最悪の事態を回避することができる。

3.6.3 IUnknownインターフェイスのポインタ

各コンポーネントの IUnknown インターフェイスのポインタには、2つの重要な特性を持たせる必要がある。何よりも重要なのは、各コンポーネントは、IUnknown インターフェイスのポインタ値を1つだけ保持し、この値はアプリケーションの実行中には一定の値をとることだ。この条件は非常に単純で、本章で示すサンプルプログラムでは簡単にこの条件を満たすことができるが、忘れないように注意する。MFC を使わない基本的な COM のサンプルプログラムを紹介するが、この例は非常に単純で、IUnknown 以外のインターフェイスを1つだけ公開するコンポーネントを扱う。

しかし、COM では、コンポーネントの内部構造については何も規定されていない。COM の世界では、コンポーネントが複数のサブオブジェクト(それぞれ1つのインターフェイスを扱う)で構成されることもある。実際、これは、複数のインターフェイスをサポートするために使う手法の1つだ。ここで、各コンポーネントの IUnknown インターフェイスのポインタは、そのコンポーネントの有効期間中には不変でなければならないという条件を思い出してほしい。

この条件は、主に、IUnknown インターフェイスのポインタが、クライアントプログラムの内部でオブジェクトを一意に識別するために使用可能であることから規定されている。たとえば、あるクライアントプログラムが IDog インターフェイスと IChihuahua インターフェイスという2つのインターフェイスポインタを取得した場合、これらのポインタの両方が同じオブジェクトに属しているのか、それぞれまったく別のコンポーネントを指すのかを区別する必要がある。

インターフェイスはすべて IUnknown から継承を行うため、どのインターフェイスも IUnknown がサポートする3つのメソッドに応答することができる。これは、クライアントプログラムが IDog インターフェイスと IChihuahua インターフェイスの両方に対して QueryInterface() を呼び出し、そのコンポーネントの IUnknown インターフェイスのポインタを要求できることを意味する。これらのインターフェイスポインタが同じオブジェクトに属するかどうかを判断するには、QueryInterface() メソッドからポインタを取得した後、単に2つのポインタを比較して同じかどうかを調べればよい。2つの IUnknown インターフェイスのポインタが同じなら、IDog インターフェイスと IChihuahua インターフェイスのポインタはどちらも同じオブジェクトのものである。2つのポインタが異なる場合、クライアントアプリケーションは2つのコンポーネントを処理していることがわかる。リスト 3-4 に、この比較処理のコード例を示す。

リスト 3-4 2つのインターフェイスポインタを比較して同じコンポーネントのものかどうかを調べる(Compare.cpp)

```

// IDog インターフェイスの有効なポインタが変数 pIDog に格納されている
// IChihuahua インターフェイスの有効なポインタが変数 pIChihuahua に格納されている

HRESULT result          = E_FAIL;
IUnknown* pDogIUnknown  = NULL;
IUnknown* pChihuahuaIUnknown = NULL;

// IDog から IUnknown インターフェイスのポインタを取得する
result = pIDog->QueryInterface(IID_IUnknown, (void**) &pDogIUnknown);

if (FAILED(result))
{
    // エラー処理
}

// IChihuahua から IUnknown インターフェイスのポインタを取得する
result = pIChihuahua->QueryInterface(IID_IUnknown,
                                     (void**) &pChihuahuaIUnknown);

if (FAILED(result))
{
    // エラー処理
}

// IUnknown インターフェイスのポインタを比較し、同じオブジェクトに属しているかどうかを判断する
if (pDogIUnknown == pChihuahuaIUnknown)
{
    cout << "The IDog and IChihuahua interfaces belong to "
          "the same component.\n";
}
else
{
    cout << "The IDog and IChihuahua interfaces DO NOT belong to "
          "the same component.\n";
}

```

1つのオブジェクトが常に同じ IUnknown インターフェイスのポインタを戻すのと同様に、1つのコンポーネントがサポートするインターフェイスのセットがそのオブジェクトの有効期間中は変わらないことも重要だ。あるコンポーネントが QueryInterface() を呼び出して有効な IDog インターフェイスのポインタを取得することができる場合には、同様に QueryInterface() を呼び出したときに E_NOINTERFACE のエラーが返されることは決してない。コンポーネントは、任意の方法でクライアントに返すインターフェイスポインタを取得することができる。しかし、ポインタを取得する方法に関係なく、コンポーネントは、常に、公開するインターフェイスに対する要求に応答可能でなければならない。

3.7 まとめ

本章では、COM オブジェクトの構築について、基本的な項目を説明した。最初に、COM では COM クライアントと COM サーバーに分けられること、COM サーバーにはインプロセスサーバー、ローカルサーバー、リモートサーバーの 3 種類があることを学んだ。また、C++ではインターフェイスが抽象基本クラス(純粹仮想関数が含まれる)として実装されるしくみについても説明した。本章で学んだ基本事項に基づいて、第 4 章「COM クライアントと COM サーバーの実装」では、COM クライアントと COM サーバーを実際に構築する。いよいよ、実践に入る。