Java と CORBA によるシステム開発

System Development with Java and CORBA

尾 島 良 司

要 約 Java と CORBA を使用したシステム開発が失敗するのは，CORBA の使い方に原因がある場合が多い．本稿では，失敗しない現実的な CORBA の使用方法について考察する．

本稿で CORBA の本質的な問題と考えているのは，ネットワークやライブラリのコストがかかることである．そこで，どのような方式を採用したとしても同等のコストがかかってしまうネットワーク通信に，CORBA の適用範囲を限定することを検討する．また，その際に考慮しなければならない点や実装の方式についても考察する．

Abstract CORBA (Common Object Request Broker Architecture) has become a system development environment frequently used by a developer or team of developers. However, the system development using both Java and CORBA may meet with failure when performing CORBA-based development in the same menner as Java language. This paper describes the usage of CORBA which does not result failure.

The essential problem of CORBA exists in the operational cost taken in the networking and the writing and maintaining library programs around CORBA objects.

In order to avoid the network operation cost, this paper proposes to set limits to the scope of CORBA objects running across network. Moreover, it discusses considerations taken into CORBA objects, and the implementation of a CORBA-based system.

1. はじめに

CORBA (Common Object Request Broker Architecture) は身近な環境となった．Linux をインストールして，デスクトップ環境を設定したら，実は内部では CORBA が使われていた，というのが現在の状況である．

Java は熟成を重ね，エンタープライズシステムのビジネスロジックを記述する言語となった．雑誌を見てもインターネットを覗いても，Java に関係する記事で溢れている．

CORBA と Java の両方を採用した開発も増えている．CORBA がインターオペラビリティを提供し，Java がポータビリティを提供する．Java＋CORBA は互いをカバーする優れた環境であり，ネットワークをまたがるシステムを作成するための最適なプラットフォームの一つである．

ところが，最適なはずの Java と CORBA を用いたシステム構築が失敗することがある．失敗の原因を調べてみると，CORBA の使い方に問題がある場合が目に付く．CORBA の使い方さえ間違えなければ，Java と CORBA によるシステム開発はかなりの割合で上手く行くのではないかと，筆者は考えている．

そこで，本稿では，筆者が今まで遭遇した，Java と CORBA によるシステム開発における問題とその解決案・回避案を，CORBA を中心に据えてまとめていく．

2. 前　　提

一般に，各種 CORBA プロダクトでは独自の機能拡張が行われている．しかし，機能を追加したからどのような開発のどのような局面でも有効である，というのは幻想でしかない．基本的なところでずれがある場合は，いくら機能を重ねても屋上屋を架すだけのことで，実際の開発の役には立たないのである．

多彩な機能に惑わされて間違った結論にたどり着くのを避けるため，本稿では CORBA の基本に立ち返って考察を行った．CORBA の基本とは通信と IDL(Interface Definition Language）によるインタフェース定義だと考え，そのレベルを超えるものは考察の対象外としている．

具体的には，プロダクト独自の拡張機能を使用した場合や，動的呼び出しや COS（Common Object Service）や OMA（Object Management Architecture）参照モデルを考慮していない．それらは，本稿であげた問題点を認識した上で初めて議論の対象となるのである．

3. CORBA の問題

本章では，CORBA を使った場合に遭遇する問題を列挙する．これらの問題をどのように回避・解決するのかは次の章で述べる．

3.1 通信のコスト

ネットワークを超えたメソッド呼び出しには，単一プロセスでのメソッド呼び出しと比べてコストが余分にかかる（図 1）．そこで，実際にどれだけのコストがかかるのか，Java でプログラムを作成して計測した[*1]．ORB は Java 2 SDK 1.2 に標準で含まれるものを使用した．結果は，CORBA の ORB 経由の場合は一回あたりの呼び出しが 10.77 ミリ秒，直接呼び出しの場合は 64.48 ナノ秒であった．単位をナノ秒に揃えると，10,770,000.00 と 64.48 であり，CORBA の ORB を経由すると，166,967 倍の時間がかかるという結果が出た．

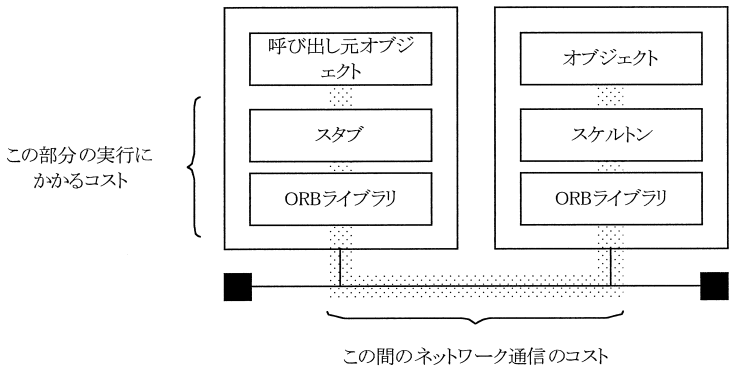

図 1　COBRA 呼び出しにかかる余計なコスト

もっとも，直接呼び出しの 64.48 ナノ秒は，同等の処理を行う C＋＋プログラムよりも速い結果であり[*2]，JIT（Just In Time）コンパイラがメソッドをインライン展

開した結果の数字だと考えられる．そこで，JIT コンパイラを使用しない場合についても同様の計測を行った．結果は CORBA の ORB 経由が 13,720,000 ナノ秒，直接呼び出しが 1,580 ナノ秒であり，8,696 倍となった．

この結果をまとめたのが表 1 である．166,967 倍と 8,696 倍のどちらの数字を採用するにせよ，CORBA 呼び出しは直接呼び出しに比べて桁が数桁異なるレベルでコストがかかるという結果となった．

表 1　ORB を経由した場合と直接呼び出しの計測結果

	ORB 経由	直接呼出し	比率
JIT あり	10,770,000.00	64.48	166,967
JIT なし	13,720,000.00	1,580.00	8,696

3.2　CORBA オブジェクトの問題

IDL で表現可能なものは，インタフェースと構造体だけである．IDL のインタフェースの実態である CORBA オブジェクトは， 通常ネットワークを超えた先に存在し，スタブ・スケルトンとライブラリの実行にかかるコストとネットワーク通信のコストがかかる．これらのコストを考慮すると，CORBA オブジェクトを Java のオブジェクトと同等に使用することには問題がある．

集合を管理する場合，Java であれば Collection Framework を，C++であれば STL（Standard Template Library）を使用する．同様の仕組みを CORBA で提供するなら，コレクション（コンテナ）もイテレータも CORBA オブジェクトとして実装することになる．コレクションやイテレータを使う場合はメソッド呼び出しを行う回数が多くなるので，CORBA オブジェクトを使用することはパフォーマンスに悪影響を及ぼす原因となる．

Java＋RMI（Remote Method Invocation）であれば，サーバで集中的にデータを管理する場合でも，集合を管理するサーバオブジェクトとキャッシュを行うローカルなイテレータという形に最適化することが可能である．これは RMI がオブジェクトの値渡しをサポートしているためであり，IDL では同様の表現を行うことが不可能である．

小さなオブジェクトを組み合わせて機能を実現するのが現在のオブジェクト指向である．呼び出しのコストを考慮すると，CORBA オブジェクトは通常のオブジェクトよりも大きな粒度にならざるを得ない．結果，イテレータなどのオブジェクト指向的なテクニックを適用できないのである．

3.3　構造体の問題

CORBA では引数や戻り値として構造体が使用できる．しかし，IDL の構造体は，オブジェクトの代替物として使用するには機能が不足している．

C では，構造体を使って複雑なデータ構造を表すことができる．これはポインタによるところが大であり，IDL では C のように構造体で複雑なデータを表すことはできない．IDL でも構造体のメンバに別の構造体を含むことはできるが，文字通り含むのであり，参照やポインタが追加されるわけではなく，構造体そのものが追加される．

社員構造体のメンバには部署構造体があり，部署構造体は社員構造体をメンバに持つ，という構成が IDL の構造体ではできない．社員は部署に所属しても所属しなくても良い，という場合の表現も難しい．sequence を使って長さ 0 か 1 の sequence にするしかないのだが，これはコードが複雑になる上に，効率が悪い方法でもある．

IDL の構造体では継承も使えない．インタフェース継承がないのでは，プログラムの抽象度は上がらず，オブジェクト指向のメリットを享受することはできない．any を使うという方法もあるが, 言語レベルでラッパを作成しないと多態性が実現できず, コーディングの負荷が大きくなる．型チェックが働かないという問題も発生する．

3.4 言語とのミスマッチ

CORBA は様々な言語をサポートしている．サポートする言語には，COBOL や C といったオブジェクト指向では無い言語も含まれる．そのため，オブジェクト指向を前提とすることができる Java と比べると，CORBA の表現力はどうしても弱くなる．

CORBA ではメソッドのオーバーロードができない．例外の継承ができない．ライブラリをサービスという大きなレベルでしか提供できない．集合を扱う機能が貧弱である．

メソッドのオーバーロードができないのは小さなことのように感じるかもしれない．しかし，setValue（Object o）と setValue（String s）を両方作成して柔軟な構造を作るのは，Java では当たり前のやり方である．setValue（Object o）という一つのメソッドだけを作成して，その内部で instanceof を使用して処理を振り分ける方式は，Java では通常は使われない．

例外の継承ができないのは，一つの例外で全てを表現するか，多数の例外を処理する try 節を書くかする必要があることを意味する．前者の方法を取ればコードが複雑になり，後者ならコーディング負荷が大きくなる．

4. Java と CORBA の使い方

本章では Java の場合を例にとって，CORBA をどのように使えば良いのかを考察する．

4.1 CORBA を使う場所

CORBA をコンポーネント分割の手段として使用することは薦められない．複数のコンピュータに分散できるというメリットは，ネットワークのコストで相殺されてしまうためである．CPU とメモリを十分に積んだコンピュータを使用すれば，パフォーマンスの問題はある程度までは解決できる．コンピュータのスペックを上げることを検討した方が良い．

複数言語でシステムを構築する場合のブリッジとして CORBA を採用することも問題がある．効率の良い別言語の呼び出し規格がある場合が多いためである．ほとんどの言語は C のライブラリを呼び出せる．本稿で実装言語として想定している Java であれば，C のライブラリを呼ぶことも，C から呼ばれることも可能である．複数言語を使う場合は，高級言語から低級言語を呼び出す場合がほとんどであり，ヘテロジーニアスな構成が採用されることは少ないことも根拠の一つである．

別のコンピュータを繋ぐという場合には，CORBA は最適な解である．プロセス間

を繋ぐ場合にも有効な場合がある．TCP/IP のソケットプログラミングや RPC（Remote Procedure Call）と比べて，コーディングが簡単で，ライブラリのコードが最適化されているためである．Java と比べるから表現力が低いという結論になるのであって，従来の通信方式と比べたならば CORBA は高機能な環境なのである．

ここまでの考察をまとめると，CORBA はネットワーク通信のための基盤であるということになる（図 2）．IDL はシステムのインタフェースを記述するものではなく，プロトコルを（形を変えて）記述するものとなるのである．

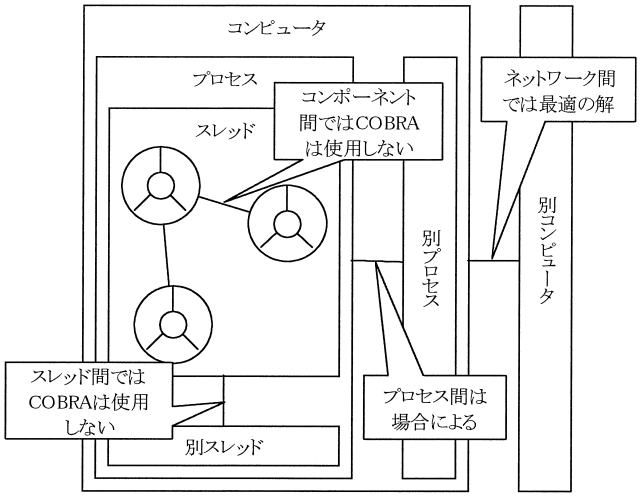

図 2　CORBA を使用する場所

オブジェクト指向設計をして，その結果を IDL で記述すれば良いという論を見ることがあるが，それでは常にネットワーク間通信を行うシステムとなり，パフォーマンスに問題が出る．設計を行った後に，ネットワーク分割を行う場所を決め，その部分の実装に合うように IDL を新規に書くというのが現実的な方法だと，筆者は考える．

IDL は表現力が低く，CORBA には制約が多い．インタフェースを表現するなら，IDL より Java の方が優れている．現在のところ，CORBA を適用するメリットがあるのは，ネットワーク通信を行う場合だけであると，筆者は考えている．

4.2　インタフェース

ネットワーク通信の道具として CORBA を使うのであれば，IDL で定義するのはプロトコルということになる．いわゆる電文レベルの内容を IDL では記述することになる．

しかし，汎用機の電文を機械的にそのまま IDL に翻訳したのでは保守性が悪いので，通常はここで何らかの工夫をする．この時に問題となるのはネットワークのコストの高さと IDL の表現力の低さである．

最近のオブジェクト指向では，オブジェクトは組み合わせて使うもの，という前提がある．IDL のインタフェースを組み合わせて使うと，ネットワークのコストが指数

的に増えていくという問題が発生する．結果，IDL のインタフェースはこのような目的には適合しないことになる．

どこかのレベルで，オブジェクト指向をやめ，単なるモジュールとしてインタフェースを捉えなおすという作業を行わなければならない．オブジェクト指向はエレガントで表現力の高い方法であるが，CORBA の上で使用する場合は，コストの制約で全面採用できない技術なのである．

オブジェクト指向から構造化への切り替えは，イテレータをグレーゾーンとして，そこより一段抽象度が低いレベルで行うのが妥当だと考える．Java 2 の Collection API や C++の STL のレベルまで行くのは明らかに行き過ぎである．

そもそも，ネットワークのプロトコルを表現するという目的に使うのであれば，IDL はオブジェクト指向を用いなければ表現できないほどには複雑にはならない．IDL でシステム構造を表現しようとするから複雑になるのである．この意味でも，CORBA はネットワークを超えるための道具として使用するのが適切である．

4.3 構　造　体

IDL の構造体でデータ構造を表すのは難しい．IDL の構造体は引数をひとかたまりにして扱いやすくするためのもの，程度に捉えておくのが良い．とはいえ，既に設計済みのデータ構造を完全に無視して，ネットワーク部分だけのデータ構造を新たに設計するのはコストのかかる作業である．

設計で使用したデータ構造を使用する場合，エンティティそのものを構造体にマップすることは簡単なのだが，ポインタが無いためにエンティティ間の関係を IDL で表現するのは難しい作業となる．

ISAM のように，エンティティの間には構造や関係が無いと考えても良い．サーバ側ではデータベースの機能が使用できるので，ISAM の時のようなマッチングのためだけのコードを書く必要は無い．関係が無くなるのはあくまで IDL の手前でだけのことである．

この方式には，パフォーマンスの問題が残る．部署と所属する社員の一覧が欲しいとする．その場合，まず部署一覧を取得して，部署の一つ一つを引数にして所属する社員を取得することになる．リレーショナルデータベースの機能を使ったなら SELECT 一回の要求で取得できる内容が，部署が 100 あった場合は 100 回の要求になるのである．

関係を表すエンティティを必ず作成するという方式もある．部署を取得した場合は部署だけ，社員を検索した場合には社員だけの情報が得られ，所属を取得した場合には所属と一緒に部署と社員が取得できるという方式である．

この場合はリレーショナルデータベースのテーブル構造が複雑になるという問題が発生する．設計方法としては正しいのだが，リレーショナルデータベールのレベルでパフォーマンスに問題が出る可能性があるのである．

構造体を使わないという解決もある．サーバへの要求を行うとプリミティブな値の集合が返ってくるという方式である．この方式には CORBA クライアントではオブジェクト指向が使えなくなるという問題がある．

筆者は，この三つの方式（図 3）から，関係を表すエンティティを使用する方式を

選択するのが最良だと考える．データベースの機能を使え，オブジェクト指向設計からの変換が機械的に行え，CORBA 通信の回数が減り，オブジェクト指向のメリットが得られるためである．パフォーマンスの問題は，チューニングや意図的に最適化を崩すことでリカバリすれば良い．

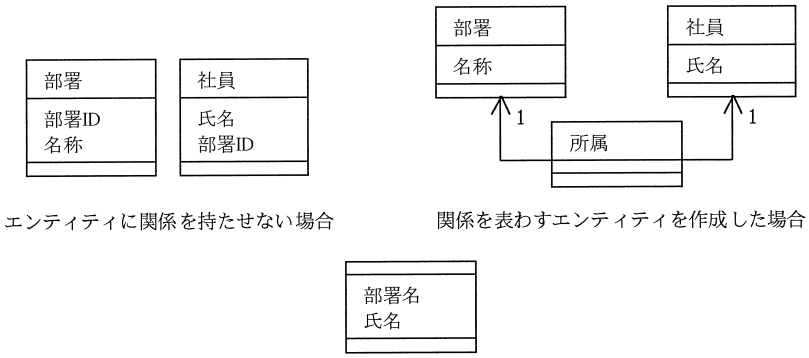

図 3　エンティティの関係の表現方法

4.4 ラ　ッ　パ

IDL をネットワークのプロトコルレベルの表現として使用するなら，データは構造体にマップされ，インタフェースは通信手順を表すものとなる．このような抽象度の低いインタフェースにプログラムが直接触れることは，コーディングの負荷を増大させることにつながる．この負荷を軽減するには，ラッパを作成することが効果的である．

ラッパを作成するなら，ラッパにどこまでの機能を持たせるかという問題を解決しなければならない．完全にローカルなオブジェクトであるかのように見えるレベルまで作りこむやり方もあり，簡単な型変換までとする方式もある．

オブジェクトの一意性を保証するところまで実装するのは，CORBA のラッパの場合はやりすぎである．オブジェクト指向データベースなどでは一意性を保証してくれるものもあるが，IDL の構造体を使う場合，それは困難な作業であり現実的ではない．

型変換だけしかしない場合は，ラッパの外側のプログラムに負荷がかかる．多態性などのオブジェクト指向の利点を得ることができないためである．これではラッパの意味が無い．

筆者は，一意性は保証しないが多態性は保証するというレベルのラッパを作成するのが良いと考える．実装負荷とオブジェクト指向の利点のバランスが取れているためである．

4.5 本稿とは別の考え方

OMA や COS（Common Object Services）は，ここに書かれている考え方とは全く逆の発想の元に作成されている．CORBA を多機能にして，通信部分の基盤ではなく，システム構築全般の基盤とする方向である．OMG 内部でも，CORBA を通信の

道具とするのか，フレームワークとするのかで意見が割れているようだが，現在のところは，多機能化の方向にある．技術や基盤の進歩次第では，本稿で懸念しているような問題は全て過去のものとなってしまうかもしれないので，将来を見越すなら多機能化の方向は正しいのかもしれない．

CORBA の手前では全くオブジェクト指向を使わないような開発をしているケースも多い．数値や文字などのプリミティブなデータ型だけを IDL で使用することにして，CORBA クライアントではオブジェクト指向を使わないのである．オブジェクト指向を使うことが目的ではないのだから，オブジェクト指向の採用を検討して見送った結果であれば，この考え方は現実的であり正しい．

5. おわりに

ここに書いたことは，あくまでも筆者の体験からの結論である．経験にこだわったのは，分散オブジェクトは理論やセールストークだけが先行している技術であり，書籍に書いてある内容そのままにはいかないと考えたためである．技術や環境が進化して，本稿で書いた問題が全て無意味になる日は近いのかもしれないが，それまでは痛い目に合いながら一つ一つこの目で確かめていこうと思う．

なお，CORBA を使用する場合の開発方法論や，ビジネスシステムで SYSTEM *v* ［nju :］を使用するような場合については，本技報の別の報告を参照していただきたい．

＊ 1　long の引数を取り，1 を足して返すだけのメソッドを作成して，CORBA を経由した場合は 1,000 回，経由しない場合は 100,000,000 回呼び出して計測した．環境は MMX　Pentium 166 MHz の Windows 98 上の Java プログラム（Java 2　SDK 1.2 を使用）である．計測は 5 回行い，その平均を結果とした．ただし，今回の計測では同一環境のコンピュータを 2 台揃えられなかったので，CORBA を経由したといってもサーバもクライアントも同じコンピュータの上で実行している．つまり，ネットワークのコストそのものは計測されていない．

＊ 2　PASCAL 呼び出しの仮想関数．VisualC＋＋6.0 で/O 2 オプションをつけてコンパイル．結果は一回あたり 102.46 ナノ秒だった．

執筆者紹介　尾　島　良　司（Ryoji Ojima）
1993 年法政大学卒業．同年日本ユニシス(株)入社．オープンシステムの設計・開発に従事．オブジェクト指向．Java，C++などの調査を経て，今日に至る．現在，生産技術部情報技術室に所属．