リフレクションを利用したCORBA言語インタフェースの改善

Improvement of a Language Interface for CORBA Using Reflection

藤枝 和宏 Kazuhiro FUJIEDA
渡部 卓雄 Takuo WATANABE
落水 浩一郎 Koichiro OCHIMIZU

北陸先端科学技術大学院大学 情報科学研究科
School of Information Science, Japan Advanced Institute of Science and Technology

概 要

本稿では、Pythonの持つリフレクション能力を利用して、CORBAのアプリケーション開発に必要なスタブとスケルトンを1)アプリケーションの実行時にインタフェースリポジトリから生成する手法と、2)アプリケーションの名前空間に取り込む手法を二つ示す。IDLコンパイラを用いる従来の手法と比べて、アプリケーションの開発手順が簡略化され、サーバ・クライアント間のインタフェースの不整合が生じる可能性が低くなる。

1 はじめに

CORBA(Common Object Request Broker Architecture)は、OMG(Object Management Group)によって策定された分散オブジェクト指向環境の標準規格である[1]。CORBAを利用すると、ネットワークを介してサービスを提供するサーバを分散オブジェクトとして実装することができて、クライアントは、そのサービスを分散オブジェクトのメソッド*を介して利用することができる。

分散オブジェクトとそのクライアントは、さまざまなプログラミング言語で記述することができる。オブジェクト指向言語で記述する場合には、クラス定義の枠組で分散オブジェクトの実装を記述し、言語のオブジェクトと同じ構文で分散オブジェクトを操作できる。

分散オブジェクトとプログラミング言語の間のインタフェースは、スタブとスケルトンと呼ばれるプログラムによって取られる。スタブは分散オブジェクトを操作する際に必要なプログラムであり、スケルトンは分散オブジェクトの実装に必要なプログラムである。これらは、**IDL(Interface Description Language)**で記述した分散オブジェクトの仕様からIDLコンパイラを用いて生成される。

CORBAのアプリケーション開発に広く用いられているC++やJavaでは、これらのプログラムをアプリケーションに取り込むために、コンパイルとリンク(Javaではサーバとクライアントへのクラスファイルの配置)が必要である。そのため、**CORBA**のアプリケーション開発はアプリケーションを実行可能にするまでの手順が煩雑であり、特に**IDL**ファイルの変更が生じる場合にはターンアラウンドが長くなる。

CORBAのアプリケーションをコンパイルやリンクの作業の不要なスクリプト言語を用いて記述することで、この問題はある程度解消できる。オブジェクト指向スクリプト言語**Python**[2]では、**Fnorb**[3]というCORBAの実装を利用することで、分散オブジェクトの実装とクライアントの両方をPythonで記述することができる。ただし、この場合でも、IDLコンパイラを用いてIDLファイルからスタブとスケルトンを生成し、アプリケーションに取り込む必要がある。

IDLコンパイラを用いてIDLファイルからスタブとスケルトンを生成する手法には、サーバとクライアントで同じ**IDL**ファイルの異なるバージョンを誤って用いたときに、インタフェースの不整合を起こす問題がある。アーキテクチャやプログラミング言語がサーバとクライアントで異なる場合には、

* CORBAではオペレーションと呼ばれる

スケルトンとスタブが異なるIDLコンパイラで別々に生成されるため、この問題が起こりやすい。スクリプト言語は、別の言語で記述されたサーバのテストプログラムの記述に用いられることが多く、この問題を引き起こす可能性が高い。

本稿では、**Python**の持つリフレクション能力を利用して、**CORBA**のインタフェースリポジトリに格納された分散オブジェクトの仕様から、アプリケーションの実行時に必要なスタブやスケルトンを生成して取り込む手法を示す。

本手法により、IDLコンパイラを起動する手順が省略できるので、開発手順が簡略化される。また、分散オブジェクトの実装時にIDLファイルをインタフェースリポジトリに転送することで、クライアントではインタフェースリポジトリから必要なスタブを直接生成できるので、インタフェースの不整合を防ぐことができる。

以下、本稿の構成は2節でPythonのリフレクション能力とインタフェースリポジトリを利用して、スタブやスケルトンを実行時に生成する手法を示す。次に、スタブやスケルトンをアプリケーションに取り込む手法として、3節でオブジェクトリファレンスを利用する手法を示し、4節でPythonのモジュール機構をカスタマイズする手法を示す。5節で関連研究について議論し、6節でまとめと今後の課題を述べる。

2 スタブ・スケルトンの実行時生成

リフレクションの可能なプログラミング言語では、プログラム自身を操作するプログラムや、プログラムの実行環境を操作するプログラムを記述することができる。前者を**Linguistic Reflection**、後者を**Behavioral Reflection**と呼ぶ[4]。Pythonではこの両者が利用可能である。**Python**の**Linguistic Reflection**と**CORBA**のインタフェースリポジトリを利用して、スタブとスケルトンを実行時に生成する手法を示す。

Pythonのプログラムの構成要素、すなわちモジュールやクラスやメソッドなどは、Pythonのオブジェクト†として実装されている。これらのオブジェクトの属性を参照、変更することで、プログラムの参照、改変が可能である。`new`モジュールや`exec`文を利用して、新たにこれらのオブジェクトを生成することもできる。

インタフェースリポジトリは、IDLファイルの内容を構文木の形で格納し、構文木の各ノードを分散オブジェクトとして提供するCORBAで規定されたサーバである。Fnorbを含むほとんどのCORBAの実装では、インタフェースリポジトリの実装と、IDLファイルの内容をインタフェースリポジトリに格納可能なIDLコンパイラが提供されている。

IDLファイルで記述される分散オブジェクトの仕様には、分散オブジェクトのインタフェースと、インタフェースで用いられる配列や構造体などのユーザ定義型の定義が含まれる。これらの定義はインタフェースリポジトリに格納される際に、リポジトリIDが割り当てられる。インタフェースリポジトリから定義を取り出す際には、リポジトリIDか定義の絶対スコープ名(図1の*Interface1*インタフェースでは::*Module1*::*Interface1*)のどちらかが必要である。

```
module Module1 {
    struct AtrType {
      string name;
      long value;
    };
    interface Interface1 {
      attribute AtrType attr1;
    };
};
```

図1　IDLファイルの例

インタフェースリポジトリから取り出した定義を元に、IDLコンパイラが生成するものと同様なスタブやスケルトンのプログラムを、PythonのLinguistic Reflectionを利用して生成することで、アプリケーションの実行時にスタブやスケルトンを生成することができる。そのためには、アプリケーションに必要な定義のリポジトリIDか絶対スコープ名が必要である。また、生成したスタブやスケルトンは、アプリケーションから参照できるように、その名前空間に取り込む必要がある。

3 オブジェクトリファレンスの利用

クライアントに必要な定義のリポジトリIDは、分散オブジェクトのオブジェクトリファレンスから取得できる。分散オブジェクトを利用するクライアントは、最初にCORBAのAPIかNamingService

† 正確には組込み型の値だが、オブジェクトと同様に扱える

を用いて分散オブジェクトのオブジェクトリファレンスを取得する。Fnorb を含むほとんどの CORBA の実装では、オブジェクトリファレンスに分散オブジェクトの型を表す ID として分散オブジェクトのインタフェースのリポジトリ ID が含まれている。このリポジトリ ID を元にインタフェースリポジトリから定義を取り出して、対応するスタブを生成することができる。インタフェースの定義の参照関係をたどることで、継承しているインタフェースやユーザ定義型など、インタフェースの利用に必要なスタブを順次生成することができる。生成したスタブは、分散オブジェクトを操作するプログラムから利用できるように、オブジェクトリファレンスの取得を実行したプログラムの名前空間に挿入する。

図 1 の IDL ファイルで定義されたインタフェース *Interface1* を利用するには、通常は IDL コンパイラを起動してスタブファイルを生成し、モジュールを取り込む import 文を用いて import *Module1* として明示的にプログラムに取り込まなければならない。上記の手法では、IDL ファイルの内容がインタフェースリポジトリに格納されているなら、IDL コンパイラの実行も import 文の実行も不要である (図 2)。

```
// オブジェクトリファレンスの取得
ior = open('server.ref', 'r').read())
// 文字列からオブジェクトリファレンスを得る API
srv = orb.string_to_object(ior)
// IDL の構造体に対応するスタブを利用して
atr = Module1.AttrType("hoge", 100)
// 属性の値を設定
srv._set_attr1(atr)
```

図 2　プログラミング例

名前空間を操作する際には、**Python** の提供している実行環境に関するプログラミング能力 (**Behavioral Reflection**) を利用する。具体的には、Python の例外処理の実装を参照する API を介してスタックフレームにアクセスして、オブジェクトリファレンスの取得を行ったプログラムの名前空間を取り出す。名前空間は Python の連想配列で実装されており、これを書き換えることで名前空間の内容を変更できる。

ところで、この手法には問題が二つある。

1. オブジェクトリファレンスの取得を行った名前空間でしか、スタブを利用することができない
   関数やメソッドを介して他のモジュールにオブジェクトリファレンスを渡すと、そのモジュールでは分散オブジェクトの操作にスタブを利用できない。
2. スケルトンを取り込むことができない
   この手法は、スケルトンの実行時生成には応用することができない。

4　モジュール機構のカスタマイズ

上記の問題に対処するために、スタブやスケルトンのモジュールを明示して取り込む方法も提供する。これを Python のモジュール機構をカスタマイズして実現する方法を示す。

4.1　Python のモジュール

Python のモジュールの概念は、ディレクトリやファイルに対応している。ディレクトリ Module1/ のファイル __init__.py にプログラムを記述して、import Module1 実行すると、このファイルで定義されたクラス Class1 を Module1.Class1 という名前で参照できる。このファイルが更新されたときには、組み込み関数 reload を用いて reload(Module1) とするとモジュールの内容を更新できる。

4.2　モジュール機構のカスタマイズ

Python では import 文の実装が __import__ という組み込み関数として公開されている。この関数と reload 関数を再定義することにより、**Python** のモジュール機構の振る舞いを、**Python** のプログラムでカスタマイズできる (**Behavioral Reflection**)。モジュール機構をカスタマイズすることで、Python のモジュール機構を用いてスタブやスケルトンを実行時に生成して、アプリケーションに取り込むことができる。

具体的には、__import__ 関数を以下のように再定義する。

1. 指定されたモジュールが存在する場合には通常通り振る舞う
2. 存在しない場合には、モジュール名 (例えば *Module1*) を IDL の絶対スコープ名 (::*Module1*) に変換して、インタフェースリポジトリから IDL のモジュール定義を取得する。
3. モジュールに含まれる定義のスタブかスケルトンを生成する (Fnorb の IDL コンパイラに習い、モジュール名の最後が _skel の場合はスケ

ルトンとする)

4. 生成したスタブやスケルトンを格納したPythonのモジュールを作成する

生成されたモジュールに対してreloadが実行された場合には、インタフェースリポジトリから定義を読み込んでスタブやスケルトンを更新するようにreloadも再定義する。

上記のようにimport文の振る舞いをカスタマイズすると、同じIDLのモジュールについて、IDLコンパイラの生成したスタブやスケルトンを取り込むためのimport文と、実行時に生成して取り込むためのimport文がまったく同じになる。この性質により、変更の可能性がないIDLのモジュールについては、IDLコンパイラを用いてスタブやスケルトンを事前に生成しておくことで、実行時の自動生成を抑制し実行コストを削減できる。

5 関連研究

IDLコンパイラを用いたスタブやスケルトンの生成を廃止して、インタフェースリポジトリを利用することで、CORBAの言語インタフェースを実現するアプローチとしてはLuaORB [5] やCorbaScript [6] がある。

LuaORBは、リフレクションの可能なプログラミング言語Luaのメソッド呼び出しの振る舞いをカスタマイズする機構を用いて、CORBAの言語インタフェースを実現している。分散オブジェクトのオペレーションが実行されたときに、インタフェースリポジトリのオペレーションの定義を参照して、リクエストプロトコルを生成することで、分散オブジェクトへのアクセスを可能する。本稿のアプローチとは異なりスタブは生成されない。そのため、ユーザ定義型に対応するスタブも利用することができず、ユーザ定義型を扱うプログラミングが煩雑になる。

CorbaScriptはCORBAのアプリケーション記述用のスクリプト言語である。CorbaScriptでは、言語の名前空間とインタフェースリポジトリの名前空間が結合されており、言語側で未定義の名前にアクセスすると、対応する定義が自動的にインタフェースリポジトリから取り込まれる。このアプローチでは、スタブやスケルトンを取り込む記述が不要で、ユーザ定義型を扱うこともできる。ただし、既存の言語のリフレクションを利用している本稿とは異なり、言語から処理系まで新規に作成しているため、クラスライブラリが存在しないこと、言語の学習コストがかかることが問題である。

6 まとめ

本稿では、Pythonの提供する**Linguistic Reflection**を用いて、**CORBA**のアプリケーション開発に必要なスタブとスケルトンをインタフェースリポジトリを介して実行時に生成する手法、およびPythonの提供する**Behavioral Reflection**を用いて、生成されたスタブとスケルトンをアプリケーションの名前空間に取り込む二つの手法を示した。これらの手法により、IDLコンパイラを用いてスタブとスケルトンを生成する場合と比べて、アプリケーションの開発手順が簡略化し、サーバ・クライアント間のインタフェースの不整合を防ぐことができる。

本稿のアプローチでは、リフレクションの可能なプログラミング言語の持つ、アプリケーションの振る舞いを動的に変更できる特性が活用されていない。本稿のアプローチを元に、IDLファイルの変更に対して、アプリケーションを止めることなくその影響を吸収する機構を実現することについて、今後検討を進めていきたいと考えている。

謝辞

本研究の遂行にあたり、(株)PFU研究所 熊谷 章氏および東田 雅宏氏のご支援およびご討論に感謝します。また、本研究は特定領域研究(A)(1)「発展機構を備えたソフトウェア構成原理の研究」(課題番号09245105)の援助の下に実施された。

参考文献

[1] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.2*, February 1998. formal/98-02-01.

[2] Python language website. http://www.python.org.

[3] Martin Chilvers. *fnorb version 1.0*. CRC for Distributed Systems Technology, February 1999. http://www.dstc.edu.au/Fnorb.

[4] Graham Kirby, Ron Morrison, and David Stemple. Linguistic reflection in java. *Software: Practice and Experience*, Vol. 28, No. 10, pp. 1045–1077, 1998.

[5] Roberto Ierusalimschy, Renato Cerqueira, and Noemi Rodriguez. Using reflexivity to interface with CORBA. In *IEEE International Conference on Computer Languages (ICCL'98)*, May 1998.

[6] Laboratoire d'Informatique Fondamentale de Lille and Inc Object-Oriented Concepts. *OMG CORBA Scripting Language RFP Revised Submission*, December 1998. OMG TC Document orbos/98-12-09.